Diss. ETH 13928

Real-Time Finite Elements: A Parallel Computer Application

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH
for the degree of
Doctor of Technical Sciences

presented by

Alexander Clemens Rhomberg
Dipl. El.-Ing. ETH
born 27th March 1971
citizen of Hausen am Albis, ZH

accepted on the recommendation of

Prof. Dr. Gerhard Tröster, examiner
Dr. Hans Eberle, co-examiner

10th January 2001

Contents

Abstract **V**

Zusammenfassung **VII**

1. Introduction **1**
  - 1.1. The LASSO project 1
  - 1.2. Anatomical model building 6
  - 1.3. Determining mechanical properties 8
    - 1.3.1. Measuring method 8
    - 1.3.2. Material law 9
    - 1.3.3. Numerical methods 9
    - 1.3.4. The measuring instrument 10
  - 1.4. Tissue deformation modelling 11
  - 1.5. Graphical modelling of organ appearance 12
    - 1.5.1. Organ specific textures 12
    - 1.5.2. Illumination models 13
  - 1.6. Force feedback manipulator 14
  - 1.7. Structure of the dissertation 15
  - 1.8. Images 17

2. Simulation of Mechanical Behaviour **25**
  - 2.1. Different methods 26
    - 2.1.1. Neural networks 26
    - 2.1.2. Mass-spring models 27
  - 2.2. Finite Element Method 29
    - 2.2.1. Overview 29
    - 2.2.2. Static and linear FE calculation 29
    - 2.2.3. Dynamic calculations 31
    - 2.2.4. Implicit integration 32
    - 2.2.5. Explicit integration 33
    - 2.2.6. Solvers for linear equations 35
    - 2.2.7. Solving explicit integration 36
    - 2.2.8. Collisions 37
  - 2.3. Summary 38

3. Fast Finite Elements **39**
- 3.1. Existing approaches to fast FE calculation 39
- 3.2. Fast algorithms 41
- 3.3. Parallelisation of the problem 43
- 3.4. Topology of the parallel computer 44
- 3.5. Distribution of the FE model 47
- 3.6. Sensitive parameters in communication 48
- 3.7. Reducing communication requirements 49
  - 3.7.1. Using local communication 50
  - 3.7.2. Mixing local and global communication 53
- 3.8. Dynamic collision detection 55
  - 3.8.1. Coarse grain/fine grain method 55
  - 3.8.2. Prediction 56

4. Computation and Communication Requirements **57**
- 4.1. Implicit integration 58
- 4.2. Explicit integration 60
  - 4.2.1. Using stiffness matrices 61
  - 4.2.2. Direct vector calculation 61
  - 4.2.3. Single point integration with hourglass control . 62
- 4.3. Estimated numbers 63

5. Parallel Hardware **65**
- 5.1. Processing elements 66
  - 5.1.1. Digital signal processor 66
  - 5.1.2. Multi-threaded processor 67
  - 5.1.3. General purpose processors 67
  - 5.1.4. Cluster of workstations 68
  - 5.1.5. Parallel super-computer 69
  - 5.1.6. Vector super-computer 69
  - 5.1.7. The case for fast processors 69
- 5.2. Communication 72
  - 5.2.1. Dedicated point to point links 73
  - 5.2.2. Switched network 73
  - 5.2.3. Available hardware 74
- 5.3. Final hardware implementation 75
  - 5.3.1. Overall system design 76
  - 5.3.2. Finite Element hardware 78
  - 5.3.3. Alpha processor nodes 78
  - 5.3.4. The Myrinet network 80

6. Parallel Program **83**
- 6.1. Overview 84
- 6.2. Topology simulator 85
- 6.3. Communication interface 86
  - 6.3.1. Local communication 87
  - 6.3.2. Global communication 88
- 6.4. Data transfer 89
- 6.5. Synchronisation 90
  - 6.5.1. Synchronising the FE computer with I/O 90
  - 6.5.2. Synchronising components of the FE engine . . . 92
  - 6.5.3. Using SMP systems 94
- 6.6. Scheduling 98
  - 6.6.1. The schedule files 98
  - 6.6.2. Scheduling for local communication 99
  - 6.6.3. Scheduling for general communication 102
- 6.7. The partition tool 102
- 6.8. Distributing Elements 105
  - 6.8.1. Optimising criteria 106
  - 6.8.2. Obtaining a distribution 107
  - 6.8.3. Optimisation 109
  - 6.8.4. Genetic adaptation 110
- 6.9. Verification 112
- 6.10. The importance of optimising compilers and optimised data layouts 113

7. The Communication System **115**
- 7.1. Communication requirements 117
- 7.2. Different communication systems 120
- 7.3. Protocols used 125
  - 7.3.1. Communication speed 127
- 7.4. FE protocol 130
- 7.5. Implementation 132
  - 7.5.1. Sending from the host 132
  - 7.5.2. Main loop of the NIC program 134
  - 7.5.3. Sending by the NIC 134
  - 7.5.4. Receiving by the NIC 135
  - 7.5.5. Receiving by the host 136
  - 7.5.6. End to end coordination 136
  - 7.5.7. Initialisation 136
  - 7.5.8. Deadlocks 137

7.6. PCI speed 138

8. Results and Outlook 141

8.1. Overall performance and speedup 141
8.2. Conclusions 145
8.3. Outlook 145
    8.3.1. Scalability 145
    8.3.2. The future of the simulator 147

Glossary 149

Abstract

This dissertation shows that it is possible to use the explicit Finite Element Method in real-time for surgery simulation, achieving yet unknown levels of realism in the simulation of the mechanical behaviour of human organs.

Only well trained surgeons can perform minimally invasive operations. Current training methods, i.e. the inexpensive but ethically questionable operations on pigs, are to be replaced by Virtual Reality based simulators. The surgeon moves manipulators that provide force feedback and expects to feel the reaction of the organs and see their movements on the display.

The modelling of the mechanical behaviour of organs has high computational requirements. To obtain a physically correct simulation, the Finite Element Method (FEM) is used. Movements are calculated by frequently determining new positions of the simulated organs. Stability criteria require a new calculation every 100μs, in order to obtain a simulation in real-time. At this rate, the fastest processors available in mid 2000 are only able to process about 60 Elements. The performance required to simulate models consisting of several hundred or thousand Elements can be achieved with a parallel computer consisting of several fast processors connected over a high speed network.

In this dissertation, methods are presented to adapt a custom-developed FE program and a parallel computer to each other in order to obtain maximum performance. As the calculations are distributed to several processors, communication between, and synchronisation of, different parts is required. These tasks should take the least possible amount of time away from the FE calculation. In the system described, we maximise the time available for computation by

- hiding the communication latency by overlapping computation and communication,
- analysing the flow of data in order to reduce the amount of information that is communicated,
- optimising communication performance by using application-specific protocols,
- integrating synchronisation into the communication, and

- exploiting the abilities of the network to free the main processors almost completely from work related to communication.

By integrating these methods, the problem can be efficiently distributed to several processors, despite the closely coupled parallel calculation and the short time-steps. Each of the processors accelerates the computation optimally. In this way, we obtain a cost effective solution for the calculation of Finite Elements in real-time.

Zusammenfassung

Diese Dissertation zeigt, dass es möglich ist, die explizite Finite Elemente Methode in der Chirurgie-Simulation in Echtzeit einzusetzen und so das mechanische Verhalten von menschlichen Organen mit einem bisher nicht gekannten Realitätsgrad zu simulieren.

Minimal invasive Operationen unter Verwendung von Endoskopen können nur von geübten Chirurgen durchgeführt werden. Die heute üblichen Ausbildungsmethoden, vor allem das ethisch umstrittene Operieren von Schweinen sollen ersetzt werden durch einen Simulator unter Verwendung von Methoden der Virtuellen Realität. Der Chirurg bewegt Manipulatoren mit Kraftrückführung und erwartet, in Echtzeit die Reaktion der Organe zu fühlen und auf dem Bildschirm zu sehen.

Die Modellierung der Bewegungen der Organe erfordert eine hohe Rechenleistung. Um eine physikalisch korrekte Simulation zu erhalten, wird die Methode der Finiten Elemente (FEM) verwendet. Bewegungen können berechnet werden, indem die Positionen der Organe immer wieder neu ermittelt werden. Durch die Physik gegebene Stabilitätsbedingungen erfordern eine neue Berechnung dieser Positionen alle 100μs, damit die Simulation in Echtzeit durchgeführt werden kann. Bei dieser Wiederholungsrate benötigt die Berechnung eines einzelnen Elementes bereits eine hohe Rechenleistung. Der schnellste Prozessor, der Mitte 2000 erhältlich war, ist in der Lage, ca. 60 Elemente in Echtzeit zu berechnen. Die Leistung, die benötigt wird, um Simulationen mit mehreren hundert oder tausend Elementen zu berechnen, kann mit einem Parallelrechner bestehend aus vernetzten schnellen Computern erreicht werden.

In dieser Arbeit werden Methoden vorgestellt, wie ein Parallelrechner und das für den Simulator entwickelte FE Programm aneinander angepasst werden können, um die maximale Rechenleistung zu erhalten. Die durch die Parallelisierung benötigte Kommunikation und Synchronisation der einzelnen Prozesse soll einen möglichst kleinen Verlust an Rechenleistung bewirken. Im vorgestellten System maximieren wir die zum Rechnen zur Verfügung stehende Zeit, indem wir

- die Kommunikation und die Berechnungen überlappen,
- durch Analyse des Datenflusses die zu kommunizierende Datenmenge reduzieren,

- durch eigens entwickelte Protokolle die Kommunikationsleistung für die auftretenden kleinen Datenpakete optimieren,
- die Synchronisation in den Datenaustausch integrieren und
- durch optimalen Einsatz des Netzwerks und durch das Nutzen seiner Möglichkeiten den Prozessor praktisch vollständig von der Kommunikation entlasten.

Durch diese Methoden kann die FE Berechnung trotz der engen Koppelung der Rechnung und der kurzen Zeitschritte auf viele Prozessoren verteilt werden, wobei jeder die Rechnung optimal beschleunigt. Dadurch erreichen wir eine kosteneffektive Lösung zur Berechnung von Finiten Elementen in Echtzeit.

1

Introduction

This dissertation describes my contribution to an interdisciplinary project with the goal of building an interactive simulator based on Virtual Reality (VR) for diagnostic laparoscopy, hence the name LASSO (LAparoscopic Surgery SimulatOr).

It was my task to speed up the calculation of the mechanical behaviour of the simulated human organs so that their reaction to mechanical stimuli could be calculated in real-time as is needed for an interactive simulation.

The project comprises many other tasks which I will describe later in this introduction.

1.1. The LASSO project

Endoscopic operations, also termed keyhole surgery, have recently become a popular technique for the diagnosis and treatment of many kinds of human diseases and injuries. The basic concept of endoscopic surgery is to minimise damage to the surrounding healthy tissue that is caused by surgery on internal organs. In traditional open surgery, relatively large cuts have to be made in the healthy skin and tissue that blocks access to the operation site. These cuts take a long time to heal and present the danger of post-operative complications such as infections. By employing minimal invasive surgery techniques, the cuts can be replaced by small perforation holes. Trocar hulls inserted into these holes serve as entry points for optical and surgical instruments into the abdominal wall for laparoscopic interventions. The small spatial extent of the tissue injury and the careful selection of the entry points results in a major reduction of complications and recovery time.

The price for these benefits is paid by the surgeon, who loses direct contact with the operation site. The necessary visual information is provided by a specialised camera (the endoscope) and presented on a screen in only two dimensions, which is counter-intuitive to normal hand-eye coordination. While preliminary systems that use stereo optics are already available, today's surgery is usually performed under mono-scopic conditions. The external control of the surgical instruments reduces the tactile feedback and limits the manipulative freedom usually available in open surgery. This way, even simple tasks such as cutting, suturing or tying knots pose unusual challenges for the inexperienced surgeon.

Performing operations under these conditions demands specific skills which can only be obtained through extensive training. The training emphasis should be shifted from the traditional approach, where the apprentice learns from the master and hones his skills on a real patient under supervision, to simulator based training (Dawson & Kaufman 1998). The basic visio-spatial and manipulative skills can be learned by using inexpensive, traditional training devices such as the Pelvi-trainer (see Storz, GmbH) or the POP unit (see Optimist, Handelsgesellschaft mbH) which utilise synthetic or animal tissue mounted in rigid cases. These inexpensive and effective units allow the trainee to learn how to navigate under conditions of mono-scopic visual feedback, as well as perform basic manipulative procedures of an intervention. While the surgeon develops competence in completing a particular task, the lack of real-life effect limits training in dexterity and surgical problem solving. More importantly, as there is a lack of realistic tissue reactivity, the trainee is unable to learn the techniques of haemostasis (as the training tissue does not bleed realistically) and therefore cannot experience the complexities of abnormal anatomy or pathological situations. While experiments on animals are sometimes used for testing new surgical techniques, practical as well as ethical reasons strongly restrict their use for common surgery training.

Virtual Reality technology has already made a significant impact on medical education and training. Early applications were exclusively based on simulated three-dimensional visualisation of the anatomy. Comprehensive volumetric visualisation systems, such as the VOXEL-MAN program (Höhne et al. 1995) allow the interactive exploration of 3D virtual anatomical models coupled with a wide range of auxiliary information about radiology, pathology or functional behaviour. The availability of high-resolution cryosectional data covering the whole human body, the Visible Human dataset of the US National Library of Medicine (Ackerman 1998), strongly accelerated the widespread development and use of virtual three dimensional anatomical models in different applications in the field of medical education and simulation.

Virtual Reality also holds an enormous potential for supporting medical training using surgery simulator systems (Satava & Jones 1998). The development of a full-scale surgery simulator depends on a much broader technological basis than just 3D organ visualisation. The following fields are also key issues in the development of a complete Virtual Reality based surgery simulator:

- construction of the underlying anatomical model,
- visualisation of the operation site,
- modelling of organ behaviour,
- force feedback for mediating haptic sensation.

Ever increasing computational power, as well as current achievements in the field of interactive computer graphics and virtual reality, have already led to the rapid development of sophisticated surgery simulators over the past years. These systems offer an appealing way to provide adequate training without direct patient involvement.

Although some attempts for the simulation of open surgical procedures have already been made (Reinig et al. 1996, Suzuki et al. 1998, Bro-Nielsen et al. 1998), major problems[1] in providing realistic interaction with organs in open surgery have prevented widespread development and use of such systems until now.

Interventions which do not require direct free-hand contact with the operation site are much more suitable for available devices for human-machine interaction. Aside from micro-surgical procedures, such as those applied to eye surgery (Sagar et al. 1994, Schill et al. 1999), interventional radiology (Hahn et al. 1998) or epidural anaesthesia (Stredney et al. 1996), most successful applications have been developed in

---

[1] It is still impossible to provide a realistic force feedback, as the movements of the hands are not constrained.

the field of endoscopic diagnosis and surgery. A wide range of VR simulator systems have been proposed and implemented during the past few years. Some of them are restricted to purely diagnostic endoscopic investigations (Alyassin & Lorensen 1998, Vining 1996, Satava 1996)), while others allow the training of surgical procedures for laparoscopic (Cover et al. 1993, Kühnapfel et al. 1995, Cotin et al. 1996, Baur et al. 1998, Downes et al. 1998) or arthroscopic (Ziegler et al. 1995, Gibson et al. 1997) interventions.

While most of the developed systems require the use of relatively powerful workstations or even small supercomputers, the possibility of using low-end personal computers for surgery simulation has also been investigated (Alyassin & Lorensen 1998, Tseng et al. 1998, Daane et al. 1995). During the past years not only numerous academic research projects have been reported, but the first industrial products have been successfully launched (Delp et al. 1996, Bro-Nielsen et al. 1998, Sutton et al. 1997, O'Toole et al. 1998).

The basic advantage of VR-based endoscopic simulators is their potential for providing a realistic and configurable training environment that bridges the gap between basic training and performing the actual interventions on patients, without restricting repetitive training. They potentially allow the simulated organs to behave in a bio-mechanically authentic manner where the tissues deform and react in a realistic fashion. Due to advances in image rendering hardware, nearly photo-realistic visualisation of the operation site will become possible in the foreseeable future. However, the simulator systems proposed to date do not achieve the necessary levels of realism. Therefore, it is of major importance to explore the current limits of realism in endoscopic surgery simulation and to analyse the potential for further development. These limits show up in many areas:

- No three dimensional anatomical models exist that are based on a real anatomy, or even patient specific data.

- The graphical appearance of organs in current simulators is far from realistic.

- The mechanical behaviour of organs is calculated with simplistic methods due to limitations in available computational power.

- The elastomechanical properties of living tissue are largely unknown.

- Current force feedback manipulators do not support enough degrees of freedom, cannot be arranged close together and are not strong enough.

The LASSO project attempts to meet these long-term research objectives in the following areas (Székely et al. 2000):

- Construction of a detailed geometric model of the anatomical site to be simulated.
- Study of the visual appearance of internal organs based on synthetic organ textures resulting from the systematic analysis of intra-operative images and from using different surface visualisation techniques.
- Development of a framework for full-scale 3D FE Modelling techniques for a physically correct simulation of elastic abdominal tissue deformation.
- Systematic study of the elastic material properties of living tissue.
- Speeding up the simulation of the mechanical behaviour to real-time performance by parallelisation.
- Integration of force feedback devices that provide the necessary haptic feedback to the surgeon.

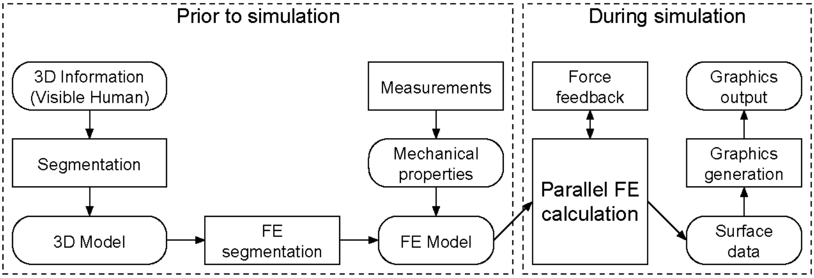


Figure 1.1: *Flow of control information in the LASSO simulator*

Figure 1.1 shows the flow of data in the LASSO simulator. Anatomical data, obtained from the Visible Human project, is analysed and three dimensional anatomical models of the simulated organs and surroundings are built. The mechanical properties of living tissue are measured with a special device. The FE model is constructed from these mechanical properties and the anatomical models which are segmented into Elements (see figure 2.5 on page 31). During a simulation, a parallel interactive FE calculation that runs in real-time simulates the mechanical behaviour of the organs. The surgeon moves the organs with a force-feedback device through which the reactions can also be felt, and the new positions of the organs are used to generate photo-realistic graphics in real-time.

1.2. Anatomical model building

The research in this field was conducted mainly by Johannes Hug, more details can be found in descriptions of his work (Hug et al. 1999).

A realistic simulation of the elastic deformation of abdominal organs and the resulting forces is only possible if it is based on a detailed anatomical model. The motion and deformation of organs is, in many cases, determined by morphological structures of small spatial extent, such as ligaments, which are difficult to place correctly when generating an anatomical model. Customary radiological imaging procedures cannot provide the necessary information due to serious constraints in image contrast and resolution, as those small structures might not even be visible on these images (Székely et al. 1998). The Visible Human Female dataset (Ackerman 1998), however, offers a consistent source of anatomical and morphological information and provides a high resolution dataset with excellent tissue contrast. Consequently, this dataset has been selected to form the basis of the anatomical model-making. The data obtained in three dimensional medical imaging and the Visible Woman data are both just a collection of voxels (volumetric pixel) with colour or intensity information, i.e. they list colour or intensity data for points in the three dimensional space. In order to obtain an anatomical model, they have to be segmented, i.e. the voxels have to be assigned to specific organs.

Existing methods for segmenting images can be grouped into the following three major methodical categories (Hug et al. 1999):

- Fully automatic tools that are easy to operate, but lack precision and completeness. They have been used extensively for the segmentation of single objects, however their applicability to multi-object systems is still an open question. (Kelemen 1998, Kelemen et al. 2000)

- Manual segmentation, which is highly insensitive to noise, can tolerate missing information, and is sufficiently precise. It is, however, extremely time consuming and tedious, with limited reproducibility. Nevertheless, in many ambiguous situations, it is advantageous to use manual segmentation.

- In between these two extremes, a whole family of semi-automatic tools have been developed (Fischler et al. 1981, Kass et al. 1988) that combine the advantages of computational support for precise border detection with the benefits of manual manipulation.

Organ definition is inherently a three-dimensional task. For the LASSO project, a slice-by-slice technique was employed, that utilises the familiar two-dimensional outlining methodology and a simple and well known interface for visualisation and manipulation.

To evaluate segmentation techniques, preliminary studies with MRI data from a female volunteer were conducted. The results of these tests, however, were unsatisfactory with respect to precision and consistency.

To obtain the volumetric data, a powerful segmentation environment was established by incorporating a multi-user segmentation system with an underlying anatomical database.

All abdominal organs influencing the elasto-mechanical behaviour and visual appearance of the operation site during gynaecological laparoscopic interventions were segmented. The inner surface of the abdominal cavity was also included. This multi-organ surface is of primary importance during simulation as it delineates the maximum spatial extent of the intervention. It surrounds the potential space where the gas is insufflated at the beginning of the surgery and wherein the surgeon places the instruments.

The Visible Human Female dataset proved to be suitable for the detailed definition of the abdominal organ geometry (Székely et al. 1998). The excellent visual fidelity and resolution provided by the colour cryosections allowed the definition of anatomy in the required quality, which no other source was able to provide. Limitations arose primarily from the post mortem anatomical changes. Even though a high quality anatomical dataset resulted from the work with the Visible Human Female data, this is only one step towards the fast and efficient creation of patient-specific digital anatomy datasets.

1.3. Determining mechanical properties

Realistic modelling of the deformation of tissues can be performed only with the knowledge of the elastic properties of living tissue. An analytical description of the mechanical behaviour can only be used if its parameters are known. The selection of the constitutive equation describing the elastic behaviour of a specific organic material (the material law) should therefore be followed by the determination of the actual numerical values of material parameters.

Since significant differences are expected between the mechanical properties of both dead and living, human and animal tissue, the measurements must be performed in vivo on patients. In addition, accurate data cannot be obtained from standard setups that require preparation of tissue, which results in significantly different mechanical properties. A number of experiments have been performed on a variety of tissues (Fung 1993), but there exists precious little data from in-vivo measurements (Moulton et al. 1995, Kühnapfel et al. 1999, Carter et al. 1999).

1.3.1. Measuring method

The measurement of mechanical properties of living human tissue presents several problems. Imaging methods, e.g. MRI elastography (Muthupillai et al. 1996) or sonoelastic imaging (Gao 1995) can only be applied within a limited strain range, while classical methods of bio-mechanical testing (Fung 1967) are difficult to implement under in-vivo conditions. It must be ensured that no injuries are caused to the patient. The measurement device has to be safe, easy to use and sterilisable. In addition, the instrument has to keep track of the boundary conditions during measurements because the contact of the instrument with the tissue to be measured can lead to global motion of the organ under investigation.

One technique that provides accurate data and overcomes the problems mentioned above is tissue aspiration (Aoki et al. 1997) in conjunction with the inverse FE Method. A tube is put against the target tissue and the air pressure in this tube is lowered. The ensuing vacuum fixes the organ to the tube, resulting in well defined boundary conditions, and causes some small deformation of the tissue inside the tube. The implemented measurement process is based on varying the pressure in the tube and determining the functions $z(r,t)$ and $P(t)$, where t is time, $P(t)$ is pressure inside the tube and $z(r,t)$ is the profile of the

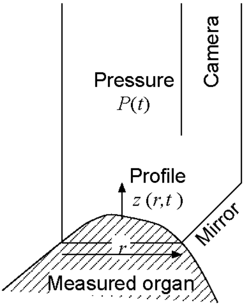


Figure 1.2: *How the mechanical properties of tissue are measured*

deformation, as illustrated in figure 1.2. Assuming there is both axisymmetry and homogeneous material in the investigated area, a complete description of the deformation is given by the measured profile $z(r,t)$. The inverse FE Method is then applied to find the material parameters from these deformations.

1.3.2. Material law

The material law is one of the most critical points in the modelling of tissue deformation. Living tissue is a nonlinear, inhomogeneous anisotropic material with viscoelastic properties. For this project, we do not try to account for all these properties in our material model, but instead a method is developed which allows the identification of material parameters once a material law has been chosen. The chosen material law has a Neo-Hookean strain energy function W (Sussman & Bathe 1987):

$$W = \mu(J_1 - 3) + \frac{1}{2}\kappa(J_3 - 1)^2 \tag{1.1}$$

Where μ is the material parameter to be determined. κ is a known material parameter and J_1 and J_3 are the first and third Jacobi invariant, respectively.

1.3.3. Numerical methods

The parameter identification is performed by a minimisation of squared differences between measured and simulated load-displacement data. An FE model, using the current estimate of the material parameters, is constructed with the same profile as the undeformed surface of the tissue under investigation. The aspiration tube is modelled as an undeformable object that touches the surface. The pressure function $P(t)$ that was measured during the experiment is then applied in the FE model. The reaction of the model is calculated with the explicit method as described in section 3.2 and the profile is compared to the measured data. The difference between measurements and calculation is then used to compute a new approximation of the material parameter using the Levenberg-Marquardt algorithm as described in (Kauer et al. 1999).

1.3.4. The measuring instrument

A vision-based device was developed to perform the measurements. This device permits controlled variation of the pressure and tracks the profiles of the resulting deformation. A small mirror placed beside the aspiration hole reflects the profile to a camera placed at the outer end of the tube (fig. 1.3).

This method allows accurate and rapid measurement of the desired profile. An optical fibre fixed in the tube illuminates the scene. The profiles are extracted from the images in real-time at a rate of 25Hz with

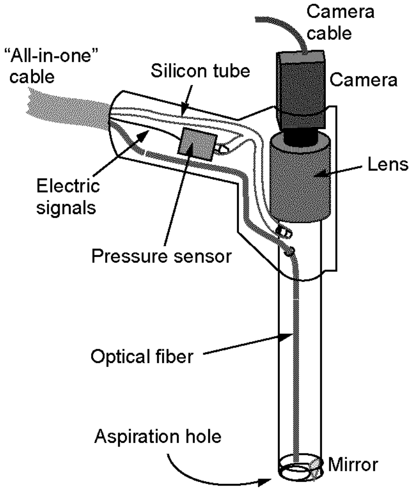


Figure 1.3: *The measurement device*

a resolution of 30 μm. Real-time extraction of the profiles avoids storage problems and can lead to real-time material parameter estimation in the future. The measuring instrument and the applied image analysis procedure are both described in more detail in (Vuskovic et al. 1999).

1.4. Tissue deformation modelling

Realistic simulation of tissue behaviour during interventions is one of the most challenging research areas in surgical simulation. While there are already first attempts in the literature for highly simplified modelling of complex interactions with organs, such as clipping, cutting or suturing (Kühnapfel et al. 1999, Pflesser et al. 1998, Basdogan et al. 1999, Baur et al. 1998, Voß et al. 1999), our project concentrated on the problem of organ deformations.

Real-time simulation of elastic tissue deformation is a major obstacle to developing systems for soft-tissue surgery. Different methods in use for deformation modelling include:

- Free-form deformation techniques from computer graphics (Barr 1984, Sederberg & Parry 1986) use parametric interpolative models for deformation estimation of the primitives. While the analogy to physical deformation processes is not always obvious, such techniques have become very popular in surgical simulators (Baur et al. 1998, Basdogan et al. 1998) due to the resulting fast deformation calculation.

- Different physically inspired approximations have also been used for tissue deformation modelling. Most popular are mass-spring models (Kühnapfel et al. 1995, Downes et al. 1998, Boux de Casson & Laugier 1999) but other alternatives such as space filling spheres (Suzuki et al. 1998) or the ChainMail algorithm (Gibson 1997) have also been implemented.

- Elastically deformable surface models used in computer graphics and computer vision (Terzopoulos et al. 1987) calculate surface deformations by solving linear elasticity equations. These models allow simulation of tissue deformation based on physical principles. Full 3D extensions of these techniques (Cotin et al. 1996, Bro-Nielsen & Cotin 1996) represent the first attempts for FE based tissue deformation modelling. Since these are linear

FE approaches, they are not feasible to applications where big deformations and rotations appear.

Today, the nonlinear FE Method has been implemented in simulators for very special cases (Sagar et al. 1994). Due to its physical background, we use this approach in our work even though it requires high computational power. The remaining chapters of this dissertation concentrate on the calculation of the nonlinear FE method and how the necessary computational power can be achieved to obtain real-time performance.

1.5. Graphical modelling of organ appearance

Providing correct visual information is indispensable in laparoscopic simulation as a means of closing the gap between VR-based surgical training and surgery on a real patient. Even though visual feedback is the most important information channel available to the surgeon, current surgery simulation systems often use standard methods for visualisation that do not provide realistic images. Whereas visualisation for laparoscopy simulation involves the treatment of a fairly wide range of topics (Soferman et al. 1998), our project concentrated on the computation of organ specific texture and the development of accurate illumination models.

1.5.1. Organ specific textures

All organ surfaces are covered by some micro-structure, which provides information about the type of the tissue and its relative smoothness. Methods to simulate such textures are available in almost all visualisation packages. Texturing, however, does more than increase realism. Perceptual psychologists have recognised the importance of surface texture as a cue to space perception (Gibson 1950, Haber & Henderson 1980). Texturing is also of fundamental relevance in representing pathological tissue. Therefore, simulating textures is important because one of the objectives of training with a laparoscopy simulator is to improve diagnostic skills.

Before this project, organ specific texturing was typically accomplished by using interactive painting methods (Sellberg et al. 1995), by mapping real laparoscopic images (Kühnapfel et al. 1995), or by interpolation based on the Visible Human data set (Reinig et al. 1996, Knapp et al. 1997). Even though all of these methods have been successfully applied, they have disadvantages that make them unsuitable for sophisticated surgery training systems. One specific virtual patient is not sufficient for training. What is needed is a system that allows medical experts to define arbitrary anatomies including pathologies by overriding default attributes of a generic anatomical model. Therefore, texture generation is to be kept independent of a specific data set. A large texture database will be provided, grouped into organs and pathologies, as well as a set of algorithms that allows automatic generation of texture maps for specific needs in a reasonable amount of time. Geometry independent organ specific textures without blood vessels can be generated with statistical methods from pictures taken during an actual laparoscopic intervention. This is described in more detail in (Meier 1999).

To generate organ specific blood vessels on top of the textures, a method was developed based on L-systems (Prusinkiewicz & Lindenmayer 1990), which are widely used in computer graphics to generate artificial plants. Basically, in an L-system, simple structures become more and more complex as their parts are recursively divided and modified according to given stochastic rules. While this is suitable for generating trees, what we need is a simulation of the biological growth process of vessels. This procedure is based on simplified models of the angiogenesis, the formation of new vessels directed by perfusion demand of growing tissue as well as physical laws of blood circulation. Aside from the ability to naturally handle vascular networks, physiologically based modelling of vessel growth can naturally adapt to organ-specific vessel formation by varying the parameters.

1.5.2. Illumination models

Ray-tracing is a technique that results in images of the highest quality. However, real-time ray-tracing cannot be implemented in the near future due to its high demands on computational power, so some simplification is necessary.

A wide range of rendering strategies currently exists that each use different illumination models. Experience has shown that artificial laparoscopic images of high quality can be produced by means of a fairly simplified ray-tracing approach. Since in laparoscopy the position of the camera and the light source are always identical, shadow computations are not required. The light source can be represented by an infinitely small point that radially emits energy that does not attenuate with distance, simulating automatic regulation of light intensity. Finally, the computation of reflection and refraction on transparent surfaces can be neglected without noticeable degradation of image quality. Due to the above simplifications, the resulting illumination models do not need ray-tracing, but can also be evaluated in real-time by means of a scan-line based method. In this project, the graphics were implemented on an SGI Onyx2/InfiniteReality machine. The scan-line methods were approximated on the standard OpenGL interface that is provided by this machine. More detail can be found in (Meier 1999).

1.6. Force feedback manipulator

Although the tactile information mediated by the surgical instruments during laparoscopic surgery is limited, force feedback is an indispensable component of any realistic simulation environment (Chen & Marcus 1998) as it also helps to replace the missing depth information. At the start of this project, no really satisfactory technical solutions were available for providing tactile and force feedback in the simulation of open surgery. However, during minimally invasive operations, haptic information is provided by mechanical manipulators, making the implementation of simulated surgical instruments with similar properties an important part of the simulator project.

Many different force feedback manipulators are available today. Besides general purpose devices, specialised instruments for laparoscopic surgery simulators have been developed and some of them are even commercially available (Immersion 1995). During laparoscopy, the following four degrees of freedom of the manipulators are required (fig. 1.4):

- The pivoting of the trocar, which is the entry point of the instrument into the body. Two angles of tilt can be measured (α and β).
- Translation of the surgical instrument through the trocar into the body (d).
- Rotation of the instrument along its longitudinal axis (γ).

A specific instrument may add additional degrees of freedom, such as the opening and closing of scissor blades. A more detailed analysis of the force feedback requirements in laparoscopic surgery can be found in (Baumann 1997).

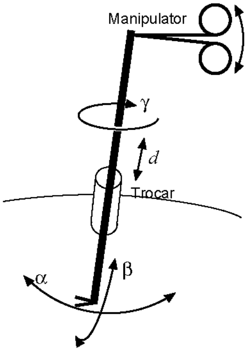


Figure 1.4: *The degrees of freedom of the manipulator*

In our system, we built the simulated surgical instruments on the basis of the commercially available PHANToM device (Massie & Salisbury 1994) of Sensable Devices (SensAble Devices 1996). In this case the manipulative degrees of freedom provided do not match our needs and can only be converted into a subset of those listed above. In the first version of the simulator, no force feedback is provided for rotation of the instrument. A physical phantom of the female abdomen which provides realistic geometry and external control of the endoscope, based on simple position tracking without force feedback, as well as specialised manipulators (Baumann 1997), will be added in the next version.

1.7. Structure of the dissertation

In chapter 2, the simulation of the mechanical behaviour of human organs is analysed and the resulting equations of the FE Method are outlined. Chapter 3 describes methods to speed up the FE calculations. It concentrates on the parallelisation of the calculation and on the reduction of additional tasks required due to the distribution of the calculation to a parallel computer. Chapter 4 analyses the requirements to communication and computation of different versions of the FE Method.

In Chapter 5, different hardware solutions for both computation and communication are compared and the selected hardware is described. Chapter 6 explains the implementation of the software, how the overhead of the parallelisation is minimised and how communication, synchronisation of the different parts and computation is scheduled concurrently. Chapter 7 contains a description of the communication system, i.e. the layer between the communication calls of the parallel FE program and the network hardware. Chapter 8 shows speed measurements and performance comparisons of different computers and other FE programs, followed by the conclusions and an outlook.

1.8. Images

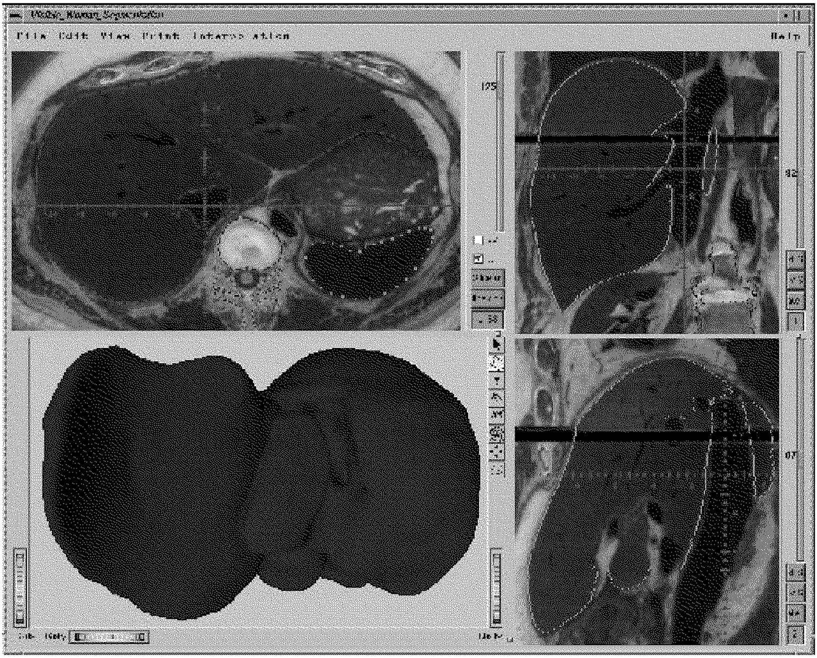


Figure 1.5: *The segmentation tool used to derive three-dimensional models of the organs of the visible human female. An overview of segmentation can be found in section 1.2*

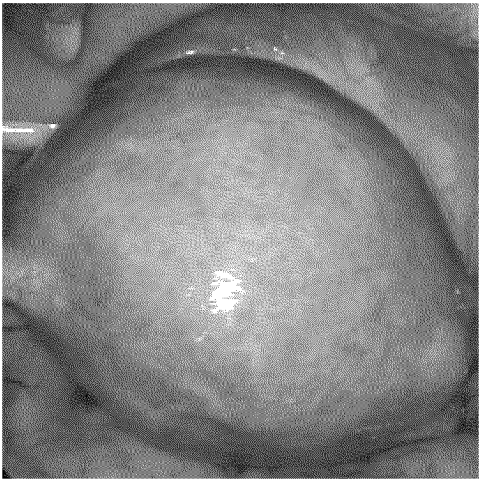


Figure 1.6: *The corpus uteri photographed during an operation*

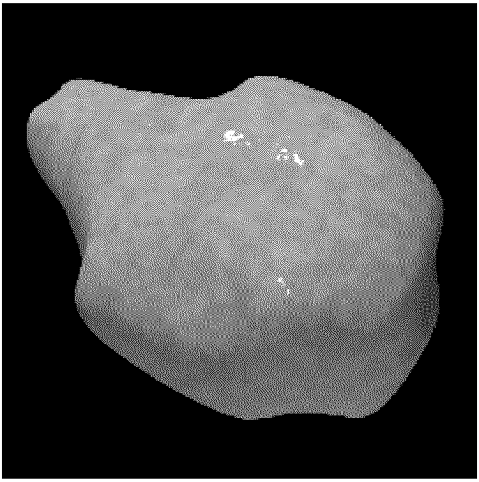


Figure 1.7: *A picture of the corpus uteri rendered by the graphics front-end. An overview of the generation of graphics can be found in section 1.5*

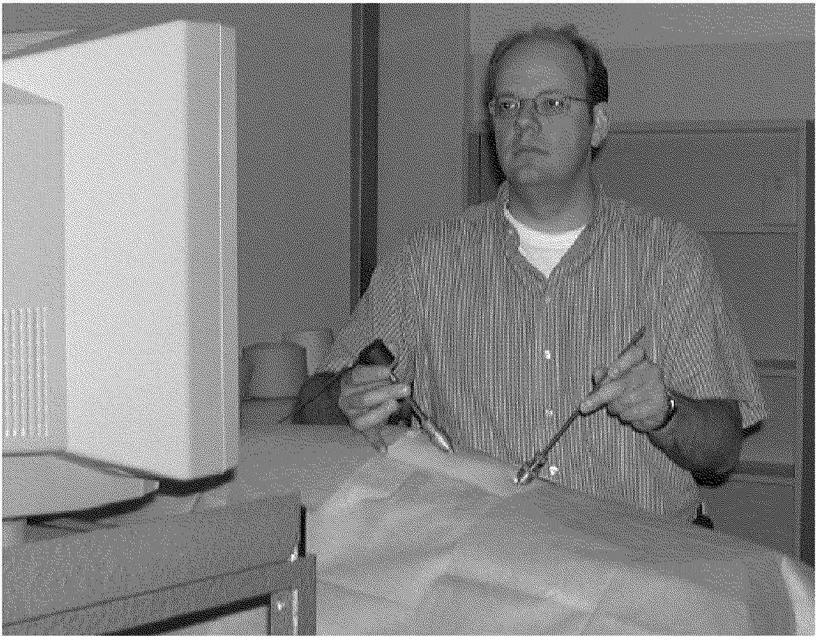


Figure 1.8: *A surgeon in front of the simulator. In his left hand, he holds the force-feedback manipulator, with his right hand he can change the position of the virtual endoscope*

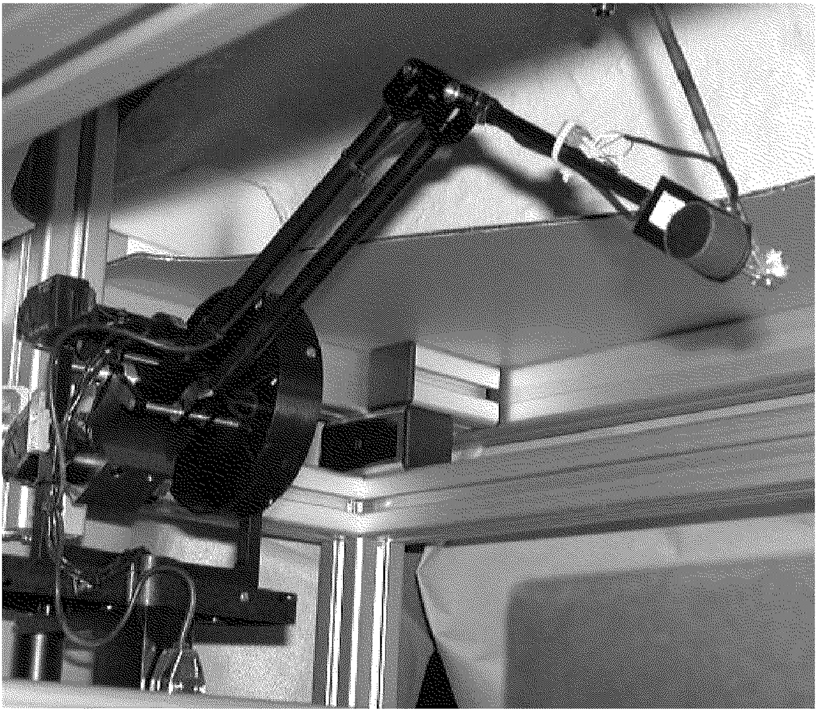


Figure 1.9: *The "intestines" of the virtual patient. The diagnostic stick at the top moves the PHANToM force-feedback device on the left. More information about force-feedback can be found in section 1.6*

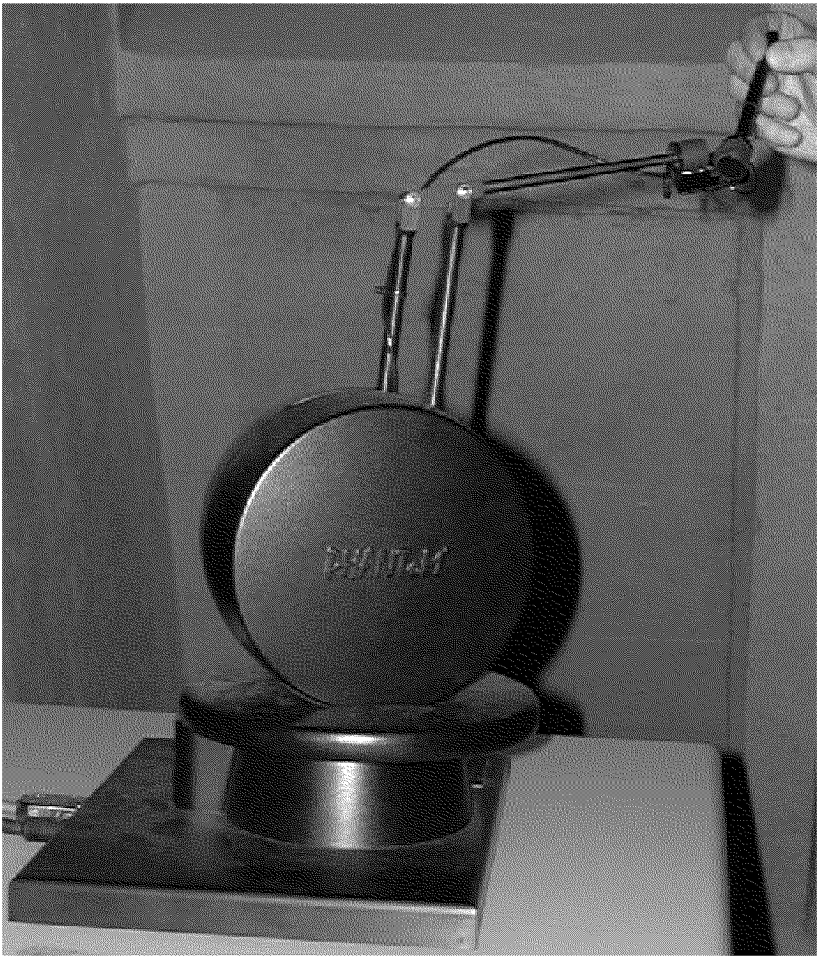


Figure 1.10: *The unmodified PHANToM force-feedback device*

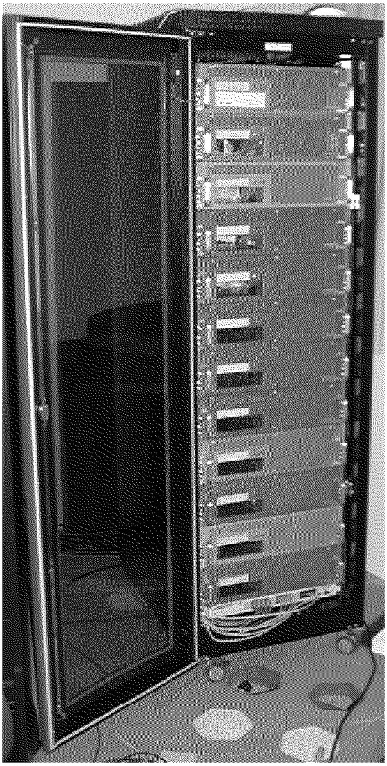


Figure 1.11: *The parallel computer with 12 dual Alpha motherboards in a standard 19" rack. This computer is described in detail in chapter 5*

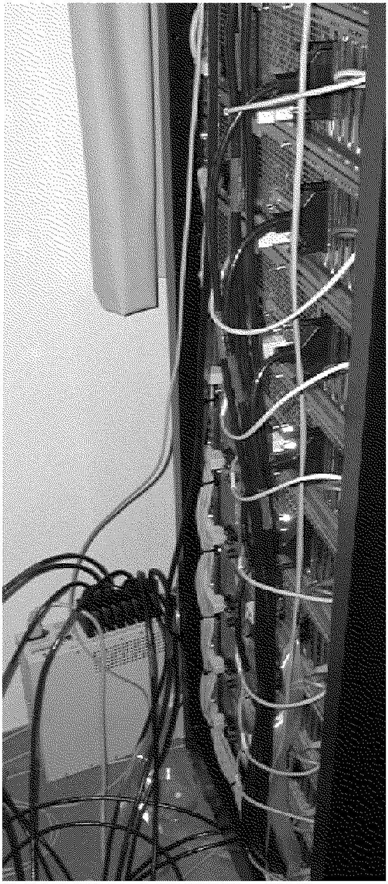


Figure 1.12: *The back of the parallel computer. The Myrinet switch is standing on the floor*

Figure 1.13: *The ONYX II parallel computer used to generate the photo-realistic graphics. The FE parallel computer can be seen on the right.*

2

Simulation of Mechanical Behaviour

At the core of the surgery simulator is the calculation of the mechanical behaviour of the organs in reaction to physical stimuli. This task requires expertise in different fields:

- The simulation of the mechanical behaviour of soft tissue is an issue of ongoing research in mechanics.
- The resulting algorithms are computationally intensive and must be optimised to allow efficient calculation.
- Even after minimising the workload with improved algorithms, knowledge in high performance computing is needed to obtain a real-time implementation.

In this chapter, I will take a look at different methods and list the calculations that have to be performed for the chosen method. At the beginning of the project, we had not chosen any particular method, but rather a set of criteria that were used to compare different simulation methods:

- The calculation of the mechanics must be fast enough to enable interactive manipulation of the model by the surgeon.
- The mechanical properties should closely match those of the real organ. The surgeon should not feel or see a difference in behaviour between the simulated and real tissue.
- Avenues for future improvements have to be kept open. These include using different models, (e.g. specific to the anatomy and pathologies of a patient), and simulating surgery instead of only allowing diagnostic laparoscopy. This means that the applied method must not make cutting a priori impossible. Any method that only simulates a surface (not a full body) is therefore not applicable.

2.1. Different methods

There are different methods to simulate the mechanical behaviour of the organs. Some methods are based on imitation and training, such as neural networks, other methods are based on physics such as mass-spring models or the FE Method.

2.1.1. Neural networks

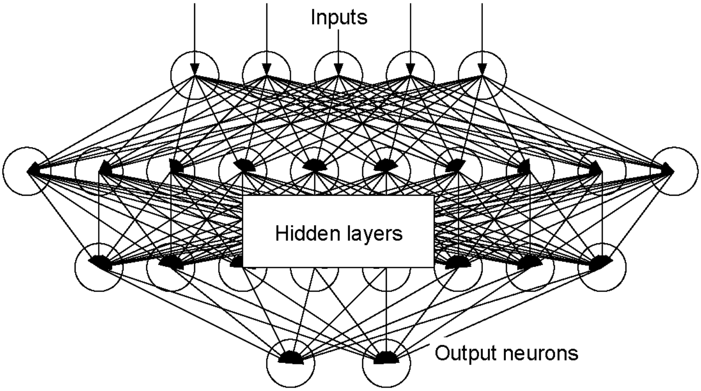


Figure 2.1: *Neural network*

Neural networks consist of several interconnected layers of neurons (fig. 2.1). An input layer is fed data, that propagates through hidden layers of neurons. Each neuron calculates a response to its inputs and feeds this response to the next layer of neurons. The output neurons finally calculate the final reaction of the network to the inputs. The network is trained to react correctly to a series of known stimuli by testing its reaction to a stimulus and then correcting the internal parameters according to the error in the output (Mehra & Wah 1992). Given unknown stimuli, the network can then extrapolate or interpolate the reactions.

In the case of surgical simulation, the inputs would be the position of the force feedback device and the old positions of the simulated organs, and the network would compute the new positions of the organs and the forces applied to the force feedback device. One of the challenges is the acquisition of sufficient training data, as thousands of sets would be required. This is difficult at best and it would also require a different method of mechanical modelling, such as the FE Method, to generate simulated behaviour. The major drawback, however, is the inability of a neural network to extrapolate beyond a certain point. It is impossible to perform any surgery in a neural network not trained for that case. In the same way, the network must be trained anew for changes in the anatomy. As neural networks do not provide an open path for future enhancements as required in our simulator prototype, we did not investigate them further.

2.1.2. Mass-spring models

The demand for real-time performance has forced most researchers to develop or adapt simplistic models of elastic deformation to the needs of surgery simulators. Mass-spring models have frequently been used as a method with relatively low performance requirements.

In a mass-spring model, the mass is concentrated in a number of nodes which are connected with springs, hence the name of the method. Mechanical reactions are computed by applying forces to the nodes, which cause them to move and pass the forces on to other nodes connected by springs. This method was first applied to real-time surgery simulation by Cover et al. (1993).

A main advantage of the mass-spring model is the possibility of unevenly distributing the nodes, i.e. the nodes are concentrated in areas of interest. The number of nodes and connecting springs can be adapted

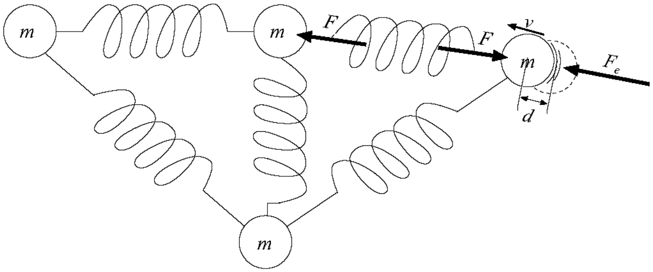


Figure 2.2: *Mass-spring model: The external force* F_e *causes a node with mass* m *to move. After a time* Δt*, the node will pick up a speed of* $v = \frac{F_e \Delta t}{m}$*, which will cause a displacement* $d = v\Delta t$*. This in turn produces a reaction force* $F = dk$ *by the spring with constant* k*, passing the movement on and countering the external force.*

to the available computational power. The first such models were used to simulate only the surface of organs. For surgery simulation, such models are unusable because they do not show volumetric behaviour, i.e. if one side of an organ is pushed, the other side does not react. In addition, the interior of a surface model is not defined. It is not possible to make incisions for simulating surgical procedures since the models are just hollow shells. The desired volumetric behaviour can only be achieved with a full-body model that models the interior of an organ and thereby connects the opposite sides of the organ. Kühnapfel et al. (1995) use a surface/volumetric mass-spring model in their KISMET simulation system.

It is difficult to obtain all the parameters, i.e. spring constants and masses, for arbitrarily interconnected nodes. Springs can be traced through the body of an organ in an irregular fashion, i.e. a node can be connected with many others in the interior or on the other side of the body and springs can also cross each other. This allows for efficient transfer of forces but complicates the calculation of the parameters. If a regular structure of nodes and springs is used, the parameters are simple to derive from the physical properties, but a regular model requires more nodes and springs and therefore more computational power than an irregular model. In order to simulate cutting, the interior of an organ must be modelled with a similar precision as the exterior. In (Deussen et al. 1995), a method for determining the optimal positions of the nodes and the distribution of the masses is presented. This paper shows how the authors were forced to use stochastic methods to find an optimal configuration of the spring network.

The mass-spring method can lead to simple models with low computational requirements, but such models do not show realistic behaviour and they do not allow cutting. The calculation of models with realistic behaviour requires a power similar to the FE Method, which is widely used in physical simulation.

2.2. Finite Element Method

2.2.1. Overview

The Finite Element Method is widely used to simulate complex problems in physics (Bathe 1996). In principle, a problem with a complexity too high to find a closed solution is divided into many small problems which can be solved individually. By combining individual solutions, an approximative solution to the original problem can be found. In mechanical simulation, a model with arbitrary shape is modelled by many small Elements. An Element is a body with one or more dimensions whose mechanical properties are known. Its corners are called nodes. The deformation of the Element is an invertible function of the forces applied to its nodes.

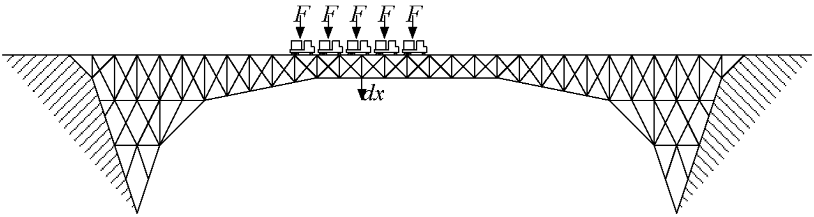


Figure 2.3: *A truss bridge: Basic FE application*

2.2.2. Static and linear FE calculation

The early implementations of Finite Elements used a linear relationship between deformation and forces. They were mainly used to calculate static problems. An example of such a problem is a bridge constructed of trusses (fig. 2.3). If a force is applied to the bridge, the FE Method finds the deformations and inner forces of the resulting stable state. In a static linear system, this information can be calculated by solving a set of thousands or millions of sparse linear equations.

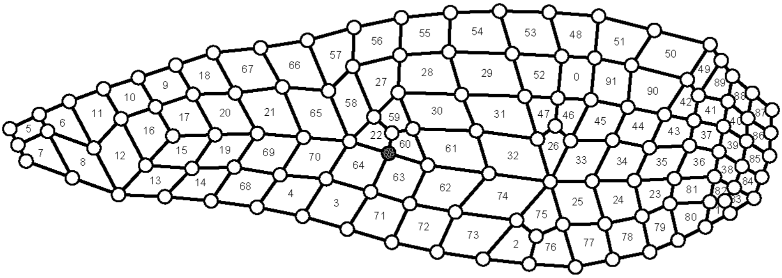


Figure 2.4: *A FE model. The circles are the nodes, the numbered quadrangles are the Elements*

The *Elements* of an FE model are connected by *nodes*. The positions of all nodes are stored in a vector $\mathbf{x}$. The calculation is performed using the *displacements* from the original position $\mathbf{x}_0$ in the displacement vector $\mathbf{u} = \mathbf{x} - \mathbf{x}_0$. For each Element E, the vector of the displacements of its nodes $\mathbf{u}^E$ are *gathered* from the global vector $\mathbf{u}$. Given $\mathbf{u}^E$, we can calculate the *strains* ε at any point in the Element by multiplying $\mathbf{u}^E$ with the transformation matrix $\mathbf{B}$ for that point.

$$\varepsilon = \mathbf{B}\mathbf{u}^E \tag{2.1}$$

The *stress* vector σ is related to the strain vector through Hooke's law:

$$\sigma = \mathbf{C}\varepsilon \tag{2.2}$$

where $\mathbf{C}$ is the material matrix. The forces $\mathbf{f}^E$ that Element E acts on the nodes can be found by transforming the stresses back and integrating over the volume of the Element.

$$\mathbf{f}^E = \int_V \mathbf{B}^T \sigma \, dV = \left(\int_V \mathbf{B}^T \mathbf{C} \mathbf{B} \, dV \right) \mathbf{u}^E \tag{2.3}$$

The integral can be computed numerically and stored in the *Element stiffness matrix* $\mathbf{K}^E$. All Element matrices can be assembled to obtain the global stiffness matrix $\mathbf{K}$, that is the elements of the Element-matrices are *scattered* and then added to the global matrix. We then derive the central equation (2.6), from which we can compute the displacements $\mathbf{u}$ of all nodes given a vector of external forces $\mathbf{f}_{ext}$:

$$\mathbf{K}^E = \int_V \mathbf{B}^T\mathbf{C}\mathbf{B}\,dV \tag{2.4}$$

$$\mathbf{K} = \sum_E \mathrm{scatter}(\mathbf{K}^E) \tag{2.5}$$

$$\mathbf{K}\mathbf{u} = \mathbf{f}_{ext} \tag{2.6}$$

2.2.3. Dynamic calculations

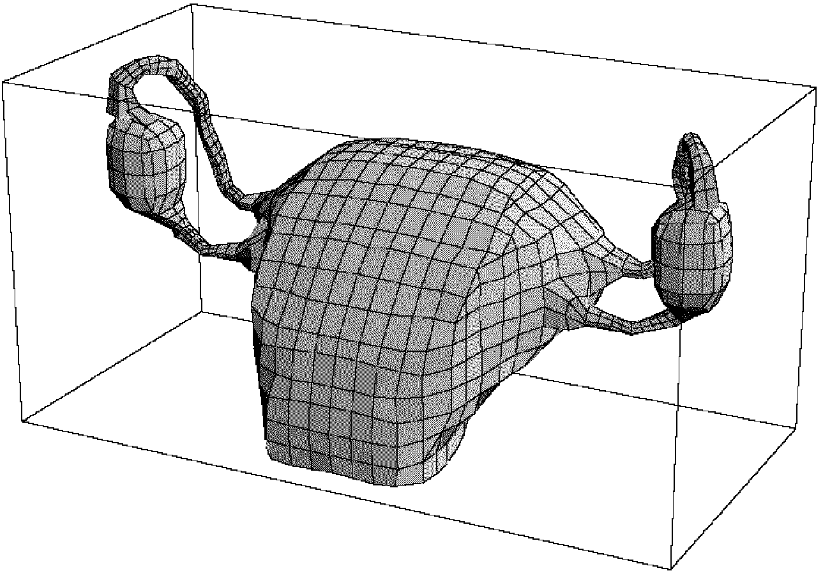


Figure 2.5: *Our FE model: Three dimensional uterus*

Static calculation cannot be used to model the movements of organs during an operation, dynamic calculation is needed. Mass and attenuation are added to the static equation (2.6) to simulate time dependent behaviour. The equation for attenuated harmonic motion is:

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{D}\dot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{f}_{ext} \tag{2.7}$$

where $\mathbf{M}$ is the mass matrix containing the masses of all Elements, $\mathbf{D}$ is the attenuation matrix with the attenuation factors between pairs of nodes and $\mathbf{K}$ is the global stiffness matrix constructed in the last section. Standard integration methods such as Central Differences or Newark are used to obtain $\ddot{\mathbf{u}}$ and $\dot{\mathbf{u}}$.

2.2.4. Implicit integration

For implicit calculation, a suitable integration method such as Newark is implemented to obtain an implicit discrete time version of equation (2.7). The equation is solved for the next time-step $t + \Delta t$.

$$\mathbf{M}\,{}^{t+\Delta t}\ddot{\mathbf{u}} + \mathbf{D}\,{}^{t+\Delta t}\dot{\mathbf{u}} + \mathbf{K}\,{}^{t+\Delta t}\mathbf{u} \;=\; {}^{t+\Delta t}\mathbf{f}_{ext} \tag{2.8}$$

For Newark integration, ${}^{t+\Delta t}\ddot{u}$ and ${}^{t+\Delta t}\dot{u}$ are substituted as follows:

$$\begin{aligned} {}^{t+\Delta t}\dot{u} &= \frac{\alpha}{\beta \Delta t}({}^{t+\Delta t}u - {}^{t}u) - \left(\frac{\alpha}{\beta} - t\right){}^{t}\dot{u} \\ &\quad -\Delta t \left(\frac{\alpha}{2\beta} - 1\right){}^{t}\ddot{u} && (2.9) \\ {}^{t+\Delta t}\ddot{u} &= \frac{1}{\beta \Delta t^2}({}^{t+\Delta t}u - {}^{t}u) - \frac{1}{\beta \Delta t}{}^{t}\dot{u} - \left(\frac{1}{2\beta} - 1\right){}^{t}\ddot{u} && (2.10) \end{aligned}$$

where α and β are constant parameters of the integration. An equation of the sort

$$\hat{\mathbf{K}}\,{}^{t+\Delta t}\mathbf{u} = \mathbf{f}\left({}^{t}\mathbf{u}, {}^{t}\dot{\mathbf{u}}, {}^{t}\ddot{\mathbf{u}}\right) \tag{2.11}$$

follows, where $\hat{\mathbf{K}}$ and $\mathbf{f}()$ can be determined from (2.8 – 2.10). To find ${}^{t+\Delta t}\mathbf{u}$, this set of linear equations has to be solved. If $\hat{\mathbf{K}}$ is constant, it could be inverted to obtain new values for ${}^{t+\Delta t}\mathbf{u}$ quickly:

$${}^{t+\Delta t}\mathbf{u} = \hat{\mathbf{K}}^{-1}\mathbf{f}\left({}^{t}\mathbf{u}, {}^{t}\dot{\mathbf{u}}, {}^{t}\ddot{\mathbf{u}}\right) \tag{2.12}$$

If only the movements and reactions of the surface of an object are needed, the inverted global stiffness matrix $\hat{\mathbf{K}}^{-1}$ could be condensed so that reaction forces for nodes in the interior of the organ are not calculated (Bro-Nielsen 1998). These simplifications work only for problems where the global stiffness matrix $\mathbf{K}$ remains constant. This is the case as long as movements are small, material behaves in a linear way and no changes, as for example caused by cutting, are made to the connectivity of the Elements. None of these restrictions are true for our problem. Organs can be moved large distances. Further, we want to be able to simulate the behaviour of non-linear material. To do this, the stiffness matrix has to be recalculated often, so the shortcuts, which are themselves computationally intensive, result in no gain.

Implicit integration is unconditionally stable, independent of the size of the time-step Δt. Unconditionally stable does not mean exact. To obtain realistic behaviour, Δt still has to be reasonably small.

A set of linear equations is not necessarily solvable. Singularities and the lack of numerical precision can cause the solver to fail. When simulating the mechanical behaviour of human organs, large deformations and displacements occur, which cannot be represented in linear calculations. There are also collisions which change connectivity and a possibility that some tissue moves freely, resulting in a set of equations that cannot be solved, because the boundary conditions are not sufficiently well defined. A method that involves solving a set of equations to find a stable state will fail in these cases. A different method must be chosen, which is more robust in such circumstances.

2.2.5. Explicit integration

By simply integrating forward in time, badly behaved problems which cause the failure of implicit methods can still be simulated. With implicit integration, the new displacements ${}^{t+\Delta t}\mathbf{u}$ are found by getting a stable solution from equations (2.8 – 2.11). In explicit integration, the new displacements are found directly from previous displacements and movements (eq. 2.18). Table 2.1 lists the differences between explicit and implicit calculation.

	Implicit	Explicit
New positions found	by solving equations	by direct calculation
Stability	Unconditionally stable	Conditionally stable
Solvability	Only some problems solvable	Always solvable
Computational power required	Moderate	High

Table 2.1: *Differences between implicit and explicit integration*

In explicit integration, the main requirement is the rate at which the calculations are repeated, as defined by the time-step Δt. If an organ is touched, the resulting shock wave is propagated at the speed of sound. This speed depends on the properties of the simulated materials and reaches about ten meters per second in the human tissue. In explicit integration, deformations are propagated from an Element to its neighbours only. Therefore, the shock wave may not skip an Element between two time-steps. The explicit integration method is therefore only stable if small enough time-steps Δt are chosen (Flanagan & Belytschko 1984). An estimation for the critical time-step Δt_{max} is given as:

$$\Delta t_{max} = \frac{\Delta L_{min}}{c} \tag{2.13}$$

where c is the speed of sound in the medium and ΔL_{min} is the shortest distance that can be traced through material between two not directly connected Elements.

By substituting ${}^{t}\dot{u}$ and ${}^{t}\ddot{u}$ in the dynamic equation (2.16) using the Central Differences equations (2.14,2.15), we can determine the explicit equation (2.18) for the new displacement ${}^{t+\Delta t}\mathbf{u}$.

$$ {}^{t}\dot{u} = \frac{{}^{t+\Delta t}u - {}^{t-\Delta t}u}{2\Delta t} \tag{2.14} $$

$$ {}^{t}\ddot{u} = \frac{{}^{t-\Delta t}u - 2\,{}^{t}u + {}^{t+\Delta t}u}{\Delta t^2} \tag{2.15} $$

$$ \mathbf{M}\,{}^{t}\ddot{\mathbf{u}} + \mathbf{D}\,{}^{t}\dot{\mathbf{u}} + \mathbf{K}\,{}^{t}\mathbf{u} = {}^{t}\mathbf{f}_{ext} \tag{2.16} $$

$$ \begin{aligned} \left(\frac{\mathbf{M}}{\Delta t^2} + \frac{\mathbf{D}}{2\Delta t}\right) \cdot {}^{t+\Delta t}\mathbf{u} = {}^{t}\mathbf{f}_{ext} + \frac{\mathbf{M}}{\Delta t^2} \cdot (2\,{}^{t}\mathbf{u} - {}^{t-\Delta t}\mathbf{u}) \\ - \frac{\mathbf{D}}{2\Delta t} \cdot {}^{t-\Delta t}\mathbf{u} + \mathbf{K}\,{}^{t}\mathbf{u} \end{aligned} \tag{2.17} $$

$$ \begin{aligned} {}^{t+\Delta t}\mathbf{u} = \hat{\mathbf{M}}^{-1}\Bigg({}^{t}\mathbf{f}_{ext} + \frac{\mathbf{M}}{\Delta t^2} \cdot (2\,{}^{t}\mathbf{u} - {}^{t-\Delta t}\mathbf{u}) \\ - \frac{\mathbf{D}}{2\Delta t} \cdot {}^{t-\Delta t}\mathbf{u} + \mathbf{K}\,{}^{t}\mathbf{u} \Bigg) \end{aligned} \tag{2.18} $$

$$ \text{where} \qquad \hat{\mathbf{M}} = \frac{\mathbf{M}}{\Delta t^2} + \frac{\mathbf{D}}{2\Delta t} \tag{2.19} $$

2.2.6. Solvers for linear equations

An exploration of the FE Method must include a look at numerical solvers. Static FE problems result in equations such as (2.6), while dynamic problems lead to (2.11). $\mathbf{u}$ is the unknown vector of displacements. The problem is to find a solution to the set of linear equations. The $\hat{\mathbf{K}}$ and $\mathbf{K}$ matrices are sparse, with most of their values being zero. The reason for this is that only Elements that are connected interact with each other, so only values of $\hat{\mathbf{K}}$ representing these interactions are nonzero. This is an important property that should be exploited when storing matrices and solving equations.

There are two general methods to find a solution for large sets of linear equations. The matrices $\mathbf{K}$ and $\hat{\mathbf{K}}$ in equations (2.6) and (2.11), respectively, can be wholly or partly inverted, normally by using Gaussian Elimination, or an iterative scheme can be applied. The most widely used iterative method is the Conjugated Gradient algorithm (Pommerell 1992).

2.2.7. Solving explicit integration

For explicit calculations, the equations to be solved are shown in (2.18). The mass of an Element can be concentrated in the nodes instead of being distributed throughout the Element, resulting in a diagonal mass matrix $\mathbf{M}$. Likewise, the physical attenuation can be substituted by a mere numerical attenuation[1], resulting in a diagonal attenuation matrix. Thus, the matrix $\hat{\mathbf{M}}$ becomes diagonal and can easily be inverted. As it is also constant, the inversion of $\hat{\mathbf{M}}$ can be calculated when the data is loaded. The multiplication of the diagonal matrices $\mathbf{M}$, $\mathbf{D}$ and $\hat{\mathbf{M}}^{-1}$ with $\mathbf{u}$ needs just one multiplication per value of $\mathbf{u}$. During each time-step, the new displacements can therefore be calculated for each node i individually with this much simpler equation:

$$ {}^{t+\Delta t}u_i = \alpha_i \left({}^{t}f_i + \beta_i \, {}^{t}u_i + \gamma_i \, {}^{t-\Delta t}u_i + \left\{ \mathbf{K} \, {}^{t}\mathbf{u} \right\}_i \right) \tag{2.20} $$

where α_i, β_i, and γ_i are constants which can be extracted from equation (2.18) at start-up time, and $\{\mathbf{K} \, {}^{t}\mathbf{u}\}_i$ is the part of the resulting vector of the multiplication related to the node i. For this multiplication, we can also take advantage of the fact that only a few of the elements of $\mathbf{K}$ are nonzero.

The resulting equations are similar to those calculated in mass-spring models. The mass has been concentrated at the nodes and every pair of nodes of an Element is connected by a spring. There are, however, fundamental differences to the commonly used mass-spring methods such as those described in (Terzopoulos et al. 1987): The parameters of the material matrix $\mathbf{C}$ can be easily determined when we know the physical properties of the material. Explicit FE is volumetric and much more realistic due to the fine grain. Further, the computational power needed is orders of magnitude higher for the FE Method than for the simple mass-spring models.

Speed of explicit integration As explained in section 2.2.5, a shock wave may not skip any Element between two time-steps. The maximum time-step is given by equation (2.13). As our mechanical simulation is used in an interactive environment, and therefore has to be calculated in real-time, we try to keep the rate of integration requirements as low

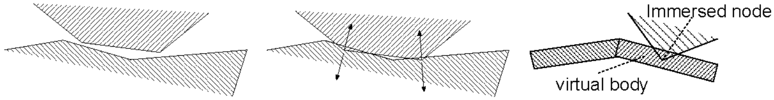
---

[1] An attenuation without physical base that attenuates the absolute speed of a node, rather than the speed relative to its neighbours. This attenuation only helps to stabilise the system by absorbing energy.

as possible. However, the laws of physics do not permit arbitrarily low rates. The tradeoff between good spatial resolution and an acceptable time-step led to a minimal feature size of $\Delta L_{min} = 10\text{mm}$. Given that the speed of sound in the modelled human tissue material is 10m/s, this results in a minimum time-step of $100\mu s$.

2.2.8. Collisions

When unconnected parts of the FE model move, collisions can occur. These collisions are not predictable and have to be detected and handled specially. A collision occurs when a node is immersed in an organ (figs. 2.6(a) & 2.6(b)). An immersion is found when a node is located below a surface. This is checked by testing if the node is inside a virtual body below that surface (fig. 2.6(c)).

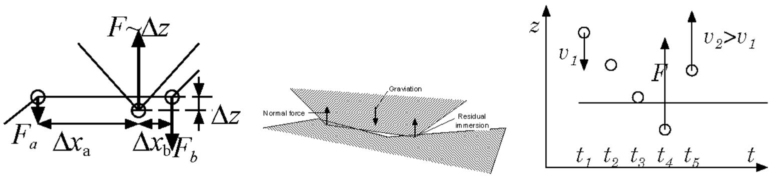


(a) No collision (b) Immersion (c) Virtual bodies below surfaces

Figure 2.6: *Collision detection*

The reaction to a collision is modelled by applying a force to both the nodes of the surface and to the immersed node. The forces are normal to the surface, in opposite directions. The forces are distributed to the nodes of the surface involved according to their distance from the immersion point, as is seen in figure 2.7(a), where $F_a + F_b = F$ and $\Delta x_a F_a = \Delta x_b F_b$. Friction forces are not considered, as human tissue has low friction.

This collision handling results in a residual immersion when objects constantly touch, e.g. when one lies on top of the other (fig. 2.7(b). This immersion of objects has to be taken into account when a scene is rendered. The ratio of immersion depth to repelling force must be chosen with care. A small repelling force results in a large immersion of touching object. If the repelling force is too strong, energy could be added to the system. This is illustrated in figure 2.7(c), where an object moves towards a surface ($t_1 - t_3$). In discrete-time calculation, the col-



(a) Reaction forces (b) Residual Immersion (c) Added Energy

Figure 2.7: *Collision handling*

lision is only detected when a considerable immersion has occurred and a large repelling force is applied (t_4). This force can cause the object to move away at a greater speed than at the beginning (t_5).

2.3. Summary

We have chosen the FE Method as it is a well known volumetric simulation method. The properties of the material enter the calculation in a straightforward way in the form of the material equation; there is no need for heuristical or manual determination of parameters. The possibility to use measured parameters leads to a highly realistic behaviour of the simulated organs. The fully volumetric calculation with its regular structure can be used to calculate the effects of cutting and thus to simulate interactive operations in the future.

The explicit integration in time has been chosen over implicit calculation as it can better handle changing and missing boundary conditions, which happen when objects collide or parts of objects fall freely. In contrast to the implicit integration, there is no danger that the simulation stops because no solution for new positions is found. However, explicit calculation needs time-steps that are much smaller than needed for smoothly animated graphics to remain stable.

3

Fast Finite Elements

The small time-steps needed for explicit FE calculation lead to a large amount of operations that have to be performed. After outlining existing work to speed up FE calculations, I will briefly explain the methods we developed to reduce the amount of work in every time-step. Since the resulting demand in computational power is too high for a standard workstation, we developed a distributed implementation on a parallel computer. A goal of this distribution is scalability, meaning that the size of the problem that can be handled should grow linearly with the number of processors. In our application, the major challenge is the large amount of data that has to be exchanged between processors. I will describe a communication topology which handles locality of the data well, and then go on to identify the parameters of both the communication system and the FE calculations that can be tuned to minimise communication and, with it, reduce the processing time lost to communication.

3.1. Existing approaches to fast FE calculation

The many methods developed to accelerate the calculation of the FE Method can be divided into two different approaches; the specialist in continuum mechanics reduces the complexity of the calculation and finds simpler algorithms, whilst the computer scientist adapts the given problems to the fastest computers available. In the literature, we can see the changes in supercomputer architecture, following the transition from the vector-computer of the 80's to the parallel computers of the 90's. The FE problems described in the literature encompass many fields such as magnetics, fluid dynamics, heat transfer or structural mechanics. The methods applied to one field can often be adapted to an other one.

A major field of FE research concentrates on fast calculation with sparse matrices (Manzini 1994) and solving sets of linear equations (Zhang & Lei 1991) on fast computers. Some researchers built computers targeted at fast FE calculation. Many different approaches were taken, such as specialised vector computers (Taylor et al. 1995), parallel machines (Loendorf 1985, Storaasli & Ransom 1987, Amano et al. 1990), and systolic arrays (Hammond & Law 1988, Evans 1994). Researchers have investigated problems related to FE calculation such as Domain Decomposition (Al-Nasra & Nguyen 1991) or Mesh Generation (Saxena & Perucchio 1992, Le-Tallec et al. 1996) to see if they can be set up with parallel computers. The highest interest, however, lies in solving the problem at hand in the fastest possible manner. While a few researchers use specialised parallel solvers to find solutions to their equation systems (Zois 1988, Baddourah & Nguyen 1994), the majority implements some variant of the Conjugated Gradient (CG) method, parallelised with domain decomposition. This means that the FE model is divided into several sub-models, at least one per processor.

A wide variety of experimental and commercial parallel and vector supercomputers was used, such as the MARK III (Nour-Omid et al. 1987), clusters of VAX computers (Hajjar & Abel 1988), the FPS-T20 (Yalamanchili et al. 1992), the Cray Y-MP (Yagawa & Shioya 1993, Chatterjee et al. 1994), the nCUBE2 system (Yagawa & Shioya 1993, Barragy et al. 1994), the IBM SP-2 (Wyrzykowski et al. 1999), or the Cray T3E (Carey et al. 1999). More recently, parallel portable environments such as PVM or MPI (Annamalai et al. 1999), are being used. Other work can be found on Element-by-Element (EBE) calculation (Hayes 1989, Davis & Carey 1995), where some of the calculation is done on a per-Element basis. EBE calculation can naturally be parallelised efficiently, given that the number of Elements is normally some orders of magnitude larger than the number of processors.

The field of research most pertinent to our project, however, is the explicit integration in time of dynamic Finite Elements (Belytschko & Gilbertsen 1987, Malone 1990). This method has been applied successfully to the massively parallel connection machine (Farhat et al. 1990, Carey & Shen 1995), demonstrating that explicit FE calculation scales well with large numbers of processors.

Once we narrow the field to research on soft tissue simulation in real-time, the literature becomes much more sparse. In (Hansen & Larsen 1998), the simulation of brain tissue by mixing static and implicit dynamic calculation is described. One of the best references that I could find is (Bro-Nielsen 1998), which describes real-time FE calculation of soft tissue for surgery simulation. However, the described method cannot be applied to our problem. While Bro-Nielsen uses linear Finite Elements with pre-inverted global stiffness matrices and pseudo-static approaches or implicit integration, we require nonlinear calculation due to large geometric movements and nonlinear material, which excludes tampering with the stiffness matrix, and explicit integration because of the much higher probability of finding a solution for situations with badly defined boundary conditions.

The reason for the lack of literature on simulation of soft tissue deformation in real-time is simple; the explicit FE calculation in real-time has only recently made the transition from *impossible* to *very interesting* in the eyes of a parallel computer specialist. To someone not willing or not able to implement custom methods and communication protocols, the problem will stay in the impossible area for some time to come. The reason for this is the amount of calculation and communication that has to be performed within the $100\mu s$ time-step. Standard methods of data exchange between and synchronisation of parallel processors are not fast enough, as measurements show in later chapters. This is caused by the granularity of the data that is transferred. A processor sends some 10 MByte/s through the network, divided into packets of only a few hundred bytes. The communication networks available today at a reasonable price cannot deliver data with small enough latency, and small packets cause significant drops in available bandwidth, as can be seen on page 128. Additionally, standard methods need significant processing time per packet.

3.2. Fast algorithms

There are two ways to speed up FE calculation. By parallelising the calculation and distributing it to a parallel computer, we accelerate the execution of a given algorithm. Choosing and optimising the right algorithm reduces the total power required.

To decrease the amount of work to be performed per time-step, we have to take a close look at the formulas that need to be calculated. The differential equation that must be solved to perform the dynamic FE calculation is shown in eq. (2.7). The result, if central differences are used for integration, is shown in (2.20). These equations can be calculated by Element and by node, respectively. The most time consuming task is the calculation of the internal Element forces $\mathbf{Ku}$, in particular the construction of the global stiffness matrix $\mathbf{K}$. However we do not even require a global matrix $\mathbf{K}$. Looking back to equation (2.5), we see that $\mathbf{K}$ is assembled from the individual Element stiffness matrices $\mathbf{K}^E$. Instead of multiplying the global displacement vector with the global stiffness matrix, we can do this multiplication for each Element and accumulate the resulting vector, eliminating the need for a global matrix:

$$\{\mathbf{K}\,\mathbf{u}\}_i = \sum_{E\in\mathbb{E}_i} \left\{\mathbf{K}^E\mathbf{u}^E\right\}_i \tag{3.1}$$

where $\mathbb{E}_i$ is the set of Elements that are connected to node i. For all other Elements, the corresponding lines in the Element stiffness matrix pertaining to node i contain only zeroes. With equation (3.1), we not only eliminated a lot of work, we also removed the last global part of equation (2.20), which is important as we are aiming for parallel processing.

In explicit FE calculations, incremental formulations are normally used. In an incremental formulation, only the current position, velocity and acceleration are known, and no direct relation exists between the actual stresses and the initial configuration. This means that a body does not remember its stress free position and residual deformations appear after forces have been applied and released in a closed load path, even if the material is assumed to be elastic. This leads to the accumulation of errors during long simulations, which drastically decreases the realism and stability of a calculation. To avoid the accumulation of such errors, an absolute formulation (Hutter et al. 2000) is needed which avoids residual deformation for a closed load path.

As we use nonlinear material and need to be able to work with large displacements of parts of our FE model, we have to recalculate the Element stiffness matrices frequently, which is an expensive operation. We can avoid the necessary matrix-matrix multiplication by calculating the stresses and strains for every integration point, instead of deter-

```
for all Elements $E$ do
    $u^E \overset{g}{=} u$(gather the positions)
    calcule the deformation gradient
    calcule the hourglass modal displacements
    calcule the invariants of the deformation Tensor
    calculate the pressure
    calculate the nominal stress tensor
    calculate the generalised hourglass forces
    project the stresses and the hourglass forces onto the internal forces $f^E$
    Scatter the internal forces: $f \overset{s}{=} f - f^E$
```

Algorithm 1: *Recovery of Elements*

mining the Element stiffness matrix $\mathbf{K}^E$. This way, we can directly *recover* an Element, i.e. determine the reaction forces as a function of the Element's deformation. The volume integral (2.3) must be evaluated, which is usually performed by a numerical 8-point quadrature. This calculation is time-consuming and can be replaced by a reduced volume integration. If the volume integration is simply reduced to a single point, information is lost and Elements can be deformed in bizarre ways (Maenchen & Sack 1964), thus requiring so called *hourglass control* (Belytschko & Ong 1984). The classical methods are based on incremental schemes with the same drawbacks as discussed above. Therefore, reduced volume integration based on absolute strain formulation has been developed (Hutter et al. 2000). Algorithm 1 shows the recovery of all Elements in a time-step.

3.3. Parallelisation of the problem

As will be shown in chapter 4, the performance requirements of the FE calculation are too high for standard computers. In section 3.2, I wrote that the resulting equations can be calculated by Element and by node and therefore *no global representation is needed.* This makes it easy to parallelise the problem by simply distributing the Elements and the nodes among processors.

In a parallel computer, communication is necessary, as processors do not have access to all information. First, we have to determine what data has to be communicated. As shown in equations (2.1) to (2.3), to recover an Element, the current positions of all its nodes have to be known. To find the new position of a node with eq. (2.20), the forces that are applied to this node by the surrounding Elements (3.1) and by the collision handling have to be added together.

To obtain correct results, data has to be exchanged during every time-step. This makes the explicit FE calculation a challenge for parallelisation, as it has low complexity[1]. If we compare the length of a time-step (100μs) with the latencies (>40μs, cf. p. 129) of standard communication methods, such as MPI or TCP/IP, we realise that special measures have to be taken. In addition, communication should use as little computational power as possible.

On a parallel computer it might be advantageous to duplicate a simple calculation, if a reduction of the communication requirements can be achieved. In order to find such cases, the equations have to be examined. The most time consuming part is the recovery of Elements shown in algorithm 1. The calculation of the position updates requires only a multiplication per vector element (eq. 2.20). It takes less time to perform this operation (requiring some 10 ns) on several processors, than on just one processor and distribute the new position to other processors, which takes some microseconds to finish.

3.4. Topology of the parallel computer

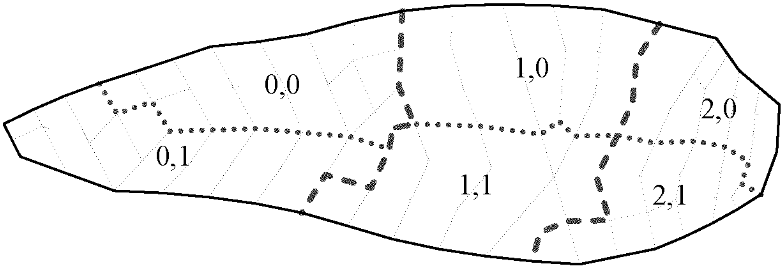


Figure 3.1: *A two dimensional example distributed to six processors. The numbers indicate the processor coordinates (x,y). Dashed and dotted lines show the borders between processors.*

---

[1] The complexity rating is a measure of the number of operations executed between exchanges of data. High complexity means many operations.

The three dimensional problem domain is divided into disjunctive sub-domains, each associated with a processor. While every Element is assigned to only one processor, the nodes the Elements are connected to can reside on more than one processor. Processors that keep track of the same node have to be able to exchange data. The division into sub-domains adds surfaces in the interior of the FE model (the dashed and dotted lines in fig. 3.1). All nodes on these additional surfaces reside on more than one processor, so minimising these surfaces also reduces the load on communication caused by parallelisation.

We divide the three dimensional object into cubes, which divide the three dimensional space almost optimally, and distribute them to the processors. The processors have to be able to communicate along the surfaces, edges and corners of these cubes. By connecting the surfaces, we obtain a three dimensional mesh of processors (fig. 3.2). The *local channels* connecting neighbours of this mesh handle communication between the surfaces and, as is shown later, also the edges and corners without additional overhead. For other irregular communication[2], an interconnection structure between arbitrary processors is required. This is called the *global bus*. This way, we get a partitioning of the problem, that is close to optimal with respect to communication requirements, while maintaining a regular and easy to use topology.

We define an invertible *placement function* $c{:}p \mapsto (x, y, z)$ that provides the coordinates of every processor in the three dimensional mesh. The functions c_x, c_y, and c_z determine the individual coordinates in the x, y, and z direction, respectively. The set of possible values for $c_x(p)$ is $\mathbb{X} = \{x \in \mathbb{N} | 1 \leq x \leq x_{max}\}$, the definitions of $\mathbb{Y}$ and $\mathbb{Z}$ are similar. The mesh is fully populated if, for every coordinate triplet $(x, y, z) \in \mathbb{X} \times \mathbb{Y} \times \mathbb{Z}$, there is exactly one processor p such that $c(p) = (x, y, z)$. We will only work with fully populated meshes.

A *distance function* $d{:}(p_1, p_2) \mapsto \delta$ helps us work with this mesh. This distance function and its one dimensional equivalents, d_x, d_y, and d_z, are described as:

$$d_x(p_1, p_2) = |c_x(p_1) - c_x(p_2)| \tag{3.2}$$

$$d_y(p_1, p_2) = |c_y(p_1) - c_y(p_2)| \tag{3.3}$$

$$d_z(p_1, p_2) = |c_z(p_1) - c_z(p_2)| \tag{3.4}$$

$$d(p_1, p_2) = d_x(p_1, p_2) + d_y(p_1, p_2) + d_z(p_1, p_2) \tag{3.5}$$

---

[2] collision detection and handling, haptic and optic display

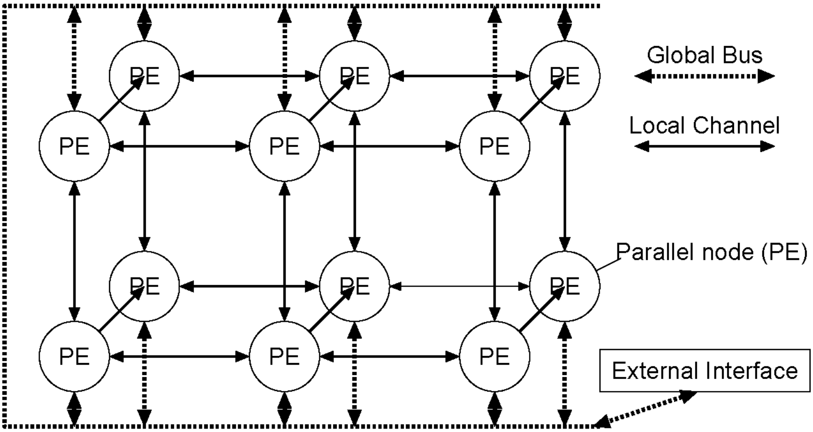


Figure 3.2: *12 processors in a three dimensional mesh with global and local communication*

The function $D:\mathbb{P} \mapsto \delta$ finds the maximum distance between the processors in a set. It is also exists in a version for each dimension:

$$D(\mathbb{P}) = \max_{(p_1,p_2)} d(p_1,p_2) \qquad p_1,p_2 \in \mathbb{P} \tag{3.6}$$

$$D_x(\mathbb{P}) = \max_{(p_1,p_2)} d_x(p_1,p_2) \qquad p_1,p_2 \in \mathbb{P} \tag{3.7}$$

$$D_y(\mathbb{P}) = \max_{(p_1,p_2)} d_y(p_1,p_2) \qquad p_1,p_2 \in \mathbb{P} \tag{3.8}$$

$$D_z(\mathbb{P}) = \max_{(p_1,p_2)} d_z(p_1,p_2) \qquad p_1,p_2 \in \mathbb{P} \tag{3.9}$$

We now define a neighbour operator $\rightleftharpoons$ which is true if two processors are *neighbours* in the mesh. This operator also exists in one dimensional versions:

$$p_1 \leftrightharpoons p_2 \qquad \longleftrightarrow \qquad d(p_1,p_2) = 1 \tag{3.10}$$

$$p_1 \overset{x}{\leftrightharpoons} p_2 \qquad \longleftrightarrow \qquad d(p_1,p_2) = d_x(p_1,p_2) = 1 \tag{3.11}$$

$$p_1 \overset{y}{\leftrightharpoons} p_2 \qquad \longleftrightarrow \qquad d(p_1,p_2) = d_y(p_1,p_2) = 1 \tag{3.12}$$

$$p_1 \overset{z}{\leftrightharpoons} p_2 \qquad \longleftrightarrow \qquad d(p_1,p_2) = d_z(p_1,p_2) = 1 \tag{3.13}$$

Any two neighbours are connected by a direct communication channel.

3.5. Distribution of the FE model

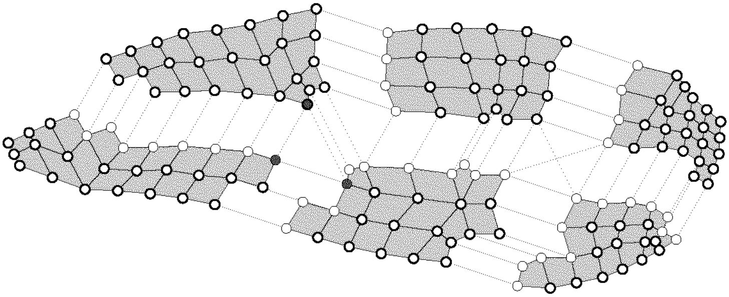


Figure 3.3: *The FE model from page 30 divided into six sub-domains, with dashed lines indicating where communication becomes necessary*

The complete FE problem consists of a set of nodes $\mathbb{K}$ and a set of Elements $\mathbb{E}$ that have to be distributed to a set of processors $\mathbb{P}$. Every Element $E \in \mathbb{E}$ is connected with a set of nodes $\mathbb{K}_E \subseteq \mathbb{K}$. From this, we can find a subset of Elements $\mathbb{E}_k = \{E | k \in \mathbb{K}_E\}$ for every node k. We define an *assignment function* $a{:}p \mapsto \mathbb{E}_p$ that assigns a set of Elements $\mathbb{E}_p \subseteq \mathbb{E}$ to every processor $p \in \mathbb{P}$ such that

$$\bigcup_{p \in \mathbb{P}} \mathbb{E}_p = \mathbb{E} \qquad \text{and} \qquad \mathbb{E}_{p_i} \cap \mathbb{E}_{p_j} = \emptyset \quad (p_i \neq p_j) \tag{3.14}$$

To determine the inner forces of its Elements, a processor p needs to know the positions of the associated nodes in the set $\mathbb{K}_p = \bigcup_{E \in \mathbb{E}_p} \mathbb{K}_E$. The set $\mathbb{P}_k = \{p | k \in \mathbb{K}_p\}$ contains all processors that need the information for node k.

There are two major aspects to an optimised distribution. On one hand, we have to distribute the computation evenly among the processors, and on the other hand, the communication requirements should be minimised. In the parallel FE program, we expect to overlap communication and computation to completely hide communication latency. First, the number of Elements assigned to a processor is determined such that the load is balanced. Next, the Elements are distributed in a way that minimises communication. As an approximation, the amount of data that has to be exchanged corresponds to the average number of processors that require the information of a node: $\sum_k |\mathbb{P}_k| \,/\, |\mathbb{K}|$. Dividing the FE model into sub-domains assigned to processors, with cuts where there are communication channels, is a good approximation of an optimal distribution.

3.6. Sensitive parameters in communication

When an application runs on a parallel computer, time is lost when data is exchanged and the parallel processors are synchronised. To be able to reduce this loss of time, we have to identify the parameters this time is sensitive to.

When data is sent from one processor to another, three components are involved, i.e. the sending processor, the receiving processor and the communication network. Accordingly, these three components can cause loss of processing time. It takes time for a processor to send data. When the receiver needs the data to be able to continue its work, it has to wait until the data arrives. The receiver also has to identify and process received data correctly, as that data might come from different senders.

The speed of and the load on the communication network determines how much time passes until data arrives at a receiver. The longer this latency, the longer the receiver has to wait for data it needs. While we can influence the raw speed of a network only by choosing fast hardware, measurements show that the granularity of the transferred data influences the bandwidth that can actually be reached (cf. page 128). For small packets, the bandwidth grows almost linearly with the size of the packet until the maximum bandwidth is asymptotically reached. This is due to the constant processing time each packet requires when it is sent, forwarded and received. Some of this time is spent by processors that reside on the network cards, some is used by the main processor, time which is no longer available for FE calculation.

We can now identify parameters that contribute to the parallelisation overhead and should be optimised:

- The total amount of data communicated contributes to the load of the network.
- The number of packets should be minimised as each requires constant processing time.
- Communication and computation should run concurrently in order to reduce the time a processor is idle, waiting for data.

- The communication should be spread evenly over the time-step to avoid performance losses due to peak loads.
- The time a processor spends sending data and identifying and processing received data should be minimised.
- The latency and per-packet overhead of the communication network should be minimised.

The first two points are addressed in this chapter, the scheduling of communication and computation is discussed in chapter 6, and the optimisation of the network is described in chapter 7.

3.7. Reducing communication requirements

Each processor p of the parallel computer is assigned a certain set of Finite Elements $\mathbb{E}_p$. In every time-step, the current positions of the nodes in the FE model are used to determine the forces an Element exerts on its nodes. For every node $k \in \mathbb{K}_p$, the resulting force f_k has to be determined by adding the force vectors from each Element $E \in \mathbb{E}_k$ that contributes to that node:

$$\mathbf{f}^E = \text{recover}(\mathbf{u}^E) \tag{3.15}$$

$$f_k = \sum_{E \in \mathbb{E}_k} f_k^E \tag{3.16}$$

$${}^{t+\Delta t}u_k = f(f_k, {}^{t}u_k, {}^{t-\Delta t}u_k) \tag{3.17}$$

Elements in $\mathbb{E}_k$ might be assigned to different processors, so communication might be needed to get the terms of the sum in eq. (3.16). Once the resulting force acting on a node is known, we can directly calculate its new position.

For a given topology and distribution of Elements, we can calculate the necessary communication bandwidth. Looking at processor p, we can determine what data it has to send to other processors. By adding up the amount of data every processor sends, we can determine the necessary aggregate bandwidth of the network. A node k requires communication if it is assigned to more than one processor, as all of the processors in $\mathbb{P}_k$ need the resulting force. Therefore, we determine the number of vectors that have to be sent by one processor as:

$$n_V^p = \sum_{k \in \mathbb{K}_p} (|\mathbb{P}_k| - 1) \tag{3.18}$$

If processor p_a sends data to processor p_b, we know that the same amount is sent from p_b to p_a due to the symmetrical nature of communication:

$$n_{p_a \to p_b} = n_{p_b \to p_a} = |\mathbb{K}_{p_a} \cap \mathbb{K}_{p_b}| \tag{3.19}$$

where $n_{p_a \to p_b}$ is the amount of data that is sent from processor p_a to processor p_b. We can therefore conclude that each processor receives exactly the same amount of data as it sends.

Numerical Example As an example, a processor with six direct neighbours is assigned a regular cube of $n \times n \times n$ Elements. Each node in a corner of the cube resides on eight processors and must therefore be sent to seven other processors. The nodes on the edges of the cube have to be sent three times and those on the surface have to be sent once. In this case $n_V^p = 7 \cdot 8 + 3 \cdot 12(n-1) + 6(n-1)^2 = 6n^2 + 24n + 26$. For 125 Elements ($n = 5$), n_V^p becomes 296, i.e. 296 vectors are sent by that processor in each time-step.

3.7.1. Using local communication

Of course the distribution is not random, but optimally adapted to the topology, as described in section 3.4. To determine the communication requirements in that case, we take a closer look at the requirements of every node. We can separate the nodes into different sets, depending on where their information is needed. *Internal nodes* are those that are only used on one processor: $\mathbb{K}_p^I = \{k \in \mathbb{K}_p \big| \, |\mathbb{P}_k| = 1\}$. The rest of the nodes are classified depending on the directions in the 3D mesh in which their data has to be exchanged. $\mathbb{K}_p^X$, $\mathbb{K}_p^Y$, and $\mathbb{K}_p^Z$ are the set of nodes that have to be communicated in x, y, and z direction, respectively. The nodes in $\mathbb{K}_p^{YZ}$ have to be sent in both y and z directions:

$$\begin{aligned} \mathbb{K}_p^X &= \{k \in \mathbb{K}_p | D_x(\mathbb{P}_k) = D(\mathbb{P}_k) = 1\} & (3.20) \\ \mathbb{K}_p^{YZ} &= \{k \in \mathbb{K}_p | D_y(\mathbb{P}_k) = D_z(\mathbb{P}_k) = 1 \wedge D_x(\mathbb{P}_k) = 0\} & (3.21) \end{aligned}$$

Other sets can be derived in a similar way.

The last set consists of the remaining nodes that cannot be handled by direct communication, because the distance in at least one direction is greater than one:

$$\mathbb{K}_p^{irr} = \left\{ k \in \mathbb{K}_p | \max_{dir=x,y,z} D_{dir}(\mathbb{P}_k) > 1 \right\} \tag{3.22}$$

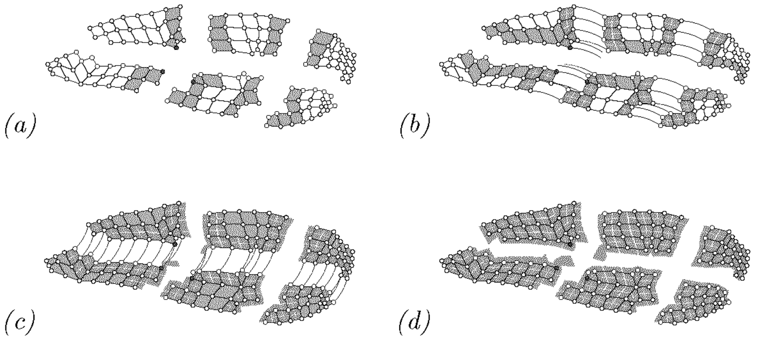


Figure 3.4: *Working principle of a communication using only local channels. The Elements on different processors are moved apart.*
(a): *At the beginning of a time-step, the Elements on the x borders are recovered (dark shade means ongoing computation).*
(b): *Some forces of computed Elements on their nodes are now known, indicated by a light shade in the respective corners. Now the processors exchange results of the x border nodes; dashed arcs show communication. Concurrently, the Elements on the y border are recovered.*
(c): *Finally, the "inner" Elements are recovered, while the last communication (y, shown as arcs) is in progress. The communicated forces include components received during step (b).*
(d): *After the last group of Elements is computed and all incoming forces have been added, all nodes have the correct resulting force.*

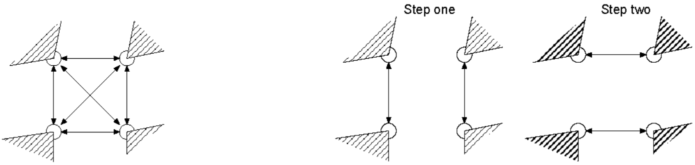
Whenever a node is represented on more than two processors, or on two which are not connected by a direct channel, the local communication is not directly sufficient to exchange data. The information can either be sent on global channels, or it can be routed through processors which are connected directly with each other.

The number of communications stays the same as in (3.18), as long as packets are only routed through processors that need to know about the data themselves. Data is routed through a number of channels equal to the distance of the associated set of processors. The advantage of using local communication channels rather than global channels is that most of the data can be sent through the faster local channels. This is, however, paid for by additional overhead in processors that not only read data but also have to send it on.

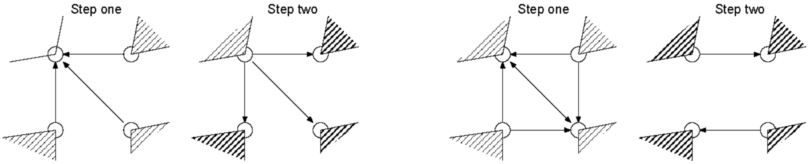
To reduce the amount of data exchanged, we have to look at the FE calculation more closely. To determine the new positions of a node k in equation (3.17), a processor does not need all partial forces f_k^E, knowing the resulting vector f_k is sufficient. That is, instead of forwarding partial forces, a processor calculates and forwards the *partial sum* of forces. Let us look at a node $k \in \mathbb{K}_p^{XY}$. It resides on four processors, p_1 to p_4, where $p_1 \overset{x}{\leftrightharpoons} p_2$ (p_1 and p_2 are neighbours in x direction), $p_3 \overset{x}{\leftrightharpoons} p_4$, $p_1 \overset{y}{\leftrightharpoons} p_3$, and $p_2 \overset{y}{\leftrightharpoons} p_4$. Each processor p_i calculates those parts of the total force it knows about ($f_k^{p_i}$) and sends them to the neighbour in x direction. By convention, all processors first send in x direction, then in y direction and, if necessary, in z direction. In this example, p_1 now has both $f_k^{p_1}$ and $f_k^{p_2}$ and forwards the sum in y direction to p_3. p_3 received $f_k^{p_4}$ before, calculated $f_k^{p_3}$ itself and can determine f_k as $(f_k^{p_1} + f_k^{p_2}) + f_k^{p_3} + f_k^{p_4}$, as can the others. At the end of just two local communication steps, all four involved processors have the complete sum f_k. This is illustrated in figure 3.4. This way, the communication is reduced to:

$$\begin{aligned} n_V^p = & \left|\mathbb{K}_p^X \cup \mathbb{K}_p^Y \cup \mathbb{K}_p^Z\right| + 2\left|\mathbb{K}_p^{XY} \cup \mathbb{K}_p^{XZ} \cup \mathbb{K}_p^{YZ}\right| \\ & + 3\left|\mathbb{K}_p^{XYZ}\right| + \sum_{n \in \mathbb{K}_p^{irr}} \left|\mathbb{P}_k - 1\right| \end{aligned} \tag{3.23}$$

Numerical Example For our processor with $n \times n \times n$ Elements, the communication amounts to $n_V^p = 3 \cdot 8 + 2 \cdot 12(n-1) + 6(n-1)^2 = 6n^2 + 12n + 6$. For the 125 Elements, this amounts to 216 vectors that have to be sent, a reduction of 27% compared to the example on page 50. We also get the additional advantage, that only local communication



(a) All to All (b) Local only



(c) One point (d) Two points

Figure 3.5: *Different communication schemes in 2D illustrations. Arrows show communication directions, lightly striped Elements are recovered just before data is exchanged, darker shaded Elements were recovered earlier.*

is used without the need for routing data through a processor.

3.7.2. Mixing local and global communication

The previous section showed the communication structure using only local communication. If both local and global communication are available, different communication methods can be considered. Instead of sending data through neighbours, processors not directly connected by local channels can exchange data over the global bus. Envision a node k in $\mathbb{K}_p^{XYZ}$ where $\mathbb{P}_k$ consists of eight processors such that $D_{dir}(\mathbb{P}_k) = 1$ for all directions. On these processors, the node is an element of $\mathbb{K}_{p_i}^{XYZ}$, so every processor has to send the information pertaining to the node three times. The total number of messages sent over local channels is 24, and three times the latency λ_ℓ of a local channel is required until all processors have received all data (fig. 3.5(b)). If only one processor collects the data, calculates the result and sends it out to all others, the necessary communication is reduced to six local and eight global messages. The partial forces of four nodes are sent over the global bus, the forces of three nodes are sent over local channels, while one is calculated on processor p. The results are determined by processor p and are sent with another four global and three local messages to the other processors. With this method, the maximum latency is twice that for global communication ($2\lambda_g$), assuming that global latency is larger than local latency ($\lambda_g > \lambda_\ell$) (fig. 3.5(c)). Several methods requiring different numbers of local and global messages and different latencies can be envisioned (fig. 3.5 (a & d)). In order to compare them, we again envisage a processor which is assigned $n \times n \times n$ Elements. This processor p, holds a total of n^3 Elements and $(n+1)^3$ nodes. The sizes of the different subsets are as follows:

$$\left|\mathbb{K}_p^I\right| = (n-1)^3 \qquad \left|\mathbb{K}_p^X\right| = \left|\mathbb{K}_p^Y\right| = \left|\mathbb{K}_p^Z\right| = 2(n-1)^2$$
$$\left|\mathbb{K}_p^{XYZ}\right| = 8 \qquad \left|\mathbb{K}_p^{XY}\right| = \left|\mathbb{K}_p^{XZ}\right| = \left|\mathbb{K}_p^{YZ}\right| = 4(n-1)$$

Depending on the method used, we get different amounts of packets sent over the local and global communication paths for different nodes. All methods have in common that the calculation of internal nodes in $\mathbb{K}_p^I$ does not require any communication and that exactly one vector is sent over local channels for every node in $\mathbb{K}_p^X \cup \mathbb{K}_p^Y \cup \mathbb{K}_p^Z$.

Method	Vectors on local bus	Vectors on global bus	maximal latency
All to all (fig. 3.5(a))	$6(n+1)^2$	12n+20	λ_g
Local only (fig. 3.5(b))	$6(n+1)^2$	0	$3\lambda_\ell$
One point (fig. 3.5(c))	$6n^2$	$6n+2$	$2\lambda_g$
Two points (fig. 3.5(d))	$6(n^2+2n-1)$	8	$\lambda_g+\lambda_\ell$

Table 3.1: *Latency and bandwidth comparison of different communication schemes for a processor with n^3 Elements*

A suitable method can be chosen when the properties of the network and the relative latencies of global and local channels are known. In our case, it proved to be optimal to use the all to all method, as the latency of the network was high and enough bandwidth was available.

3.8. Dynamic collision detection

The detection of collisions is a major task, as every node of the model attached to an outer surface has to be checked against every surface of the model. This task can consume a significant fraction of the total computation time. By using special properties of the dynamic calculation, a significant amount of time can be saved.

3.8.1. Coarse grain/fine grain method

To reduce the amount of operations as well as the communication bandwidth required to perform collision detection, a two level operation is performed. The FE model is divided into sub-domains by the parallelisation, one sub-domain is assigned to each node of the parallel computer. The first step is to compare the bounding boxes of these sub-domains. If they are disjunctive, no further checking must be carried out. Only if they overlap does additional information have to be exchanged and tests that become more and more accurate are performed in subsequent steps.

If additional properties of the model are known, e.g. that node/surface pairs cannot collide or sub-domains cannot collide with each other, the checking can be further reduced.

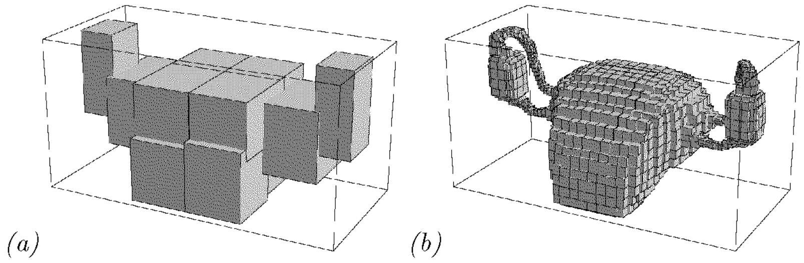


Figure 3.6: *Bounding Boxes of the uterus' surface. The whole model is about 12cm from left to right*
(a): *The bounding boxes of the model split into twelve sub-domains*
(b): *The bounding boxes of all surfaces*

3.8.2. Prediction

While the method of using different levels of granularity reduces the workload, the predictive method increases the average time between checks. When a node does not collide with a surface, in the same step an approximate distance between node and surface is calculated. This distance, together with other known properties of the model such as current velocity and maximum acceleration, can be used to determine a minimum time frame in which no collision can occur. In our model, the 100μs time-step is short, while the speeds at which parts move are comparatively slow. We can safely assume that vertical acceleration is less than one g, i.e. the acceleration of a freely dropping object. The horizontal speed is constrained by the movements of the surgeon, which reach at most a speed of 1m/s. So, even for small distances between possible contact partners, we might find that hundreds of time-steps have to go by before a collision can occur. A horizontal distance of 10mm between two possible collision partners ensures 100 time-steps without contact. During this time, these contact partners can be safely ignored. This prediction reduces the workload considerably.

4

Computation and Communication Requirements

To determine whether the mechanic simulation can be done within our limits of execution time and implementation costs, we first had to assess the performance requirements, with respect to computation and communication. The performance estimation was made both for implicit integration with Conjugated Gradients and for explicit integration. At the end of this chapter, we can reexamine the estimated requirements by looking at the model that is actually used in the simulator.

Our FE model consists of $n_{e\ell} = 1840$ hexaeder Elements, each with eight nodes or $df_{e\ell} = 24$ degrees of freedom. Each node is, on average, connected to $n_{eav} = 5.6$ Elements. These Elements together have $df_{eav} = 61.4$ degrees of freedom. All in all, there are $n_k = 2528$ nodes. The location of each node is specified by three dimensional coordinates, giving the complete model $df_g = 3n_k = 7584$ degrees of freedom. With full integration, an Element has $n_{int} = 8$ integration points, which can be reduced by known integration schemes.

The time-step Δt_{im} for implicit integration may be, at most, as large as the time between displaying two frames, i.e. $\Delta t_{im} \leq \Delta t_{graphics} = 20\text{ms}$. While implicit integration is always stable, large time-steps reduce the precision of the result. Another possible value for the time-step is the rate at which force feedback data has to be calculated to provide a smooth interface to the surgeon; a reasonable value is $\Delta t_{haptic} = 1\text{ms}$. The time-step for explicit integration is easier to choose, as it depends on physical properties of the model. In our case, it is $\Delta t_{ex} = 100\mu\text{s}$.

The Conjugated Gradient (CG) method is an iterative solver which takes n_{it} iterations to arrive at a result with a reasonably small error. n_{it} differs in every time-step depending on the input data. Since the result of the previous time-step is used as an initial position for the next one, n_{it} will be large if many parts of the model move.

4.1. Implicit integration

Implicit integration based on the CG solver, consists of two time consuming parts: constructing the global stiffness matrix and finding a solution. The number of operations per second N_{CG} and the required communication bandwidth C_{CG} can be calculated as follows:

$$N_{CG} = \frac{n_{e\ell} N_K + n_{it} N_\varrho}{\Delta t_{im}} \tag{4.1}$$

$$C_{CG} = \frac{C_K + n_{it} C_\varrho}{\Delta t_{im}} \tag{4.2}$$

N_K	Operations for constructing the Element stiffness matrix $\mathbf{K}_E$
N_ϱ	Operations per iteration step
C_K	Data that is transferred to redistribute the global stiffness matrix
C_ϱ	Communicated data for calculating the error ϱ

The basis for calculating of the Element stiffness matrix is equation (2.4). Expanded by the Piola-Kirchhoff stress matrix $\hat{\mathbf{S}}$ (Bathe 1996) we obtain:

$$\mathbf{K}^E = \int_V \mathbf{B}_L^T \mathbf{C} \mathbf{B}_L \, dV + \int_V \mathbf{B}_N^T \hat{\mathbf{S}} \mathbf{B}_N \, dV \tag{4.3}$$

The calculation of the individual matrices of equation 4.3 can be found in a more elaborate documentation of the FE Method (Bathe

1996). Here I only list the number of operations required. In table 4.1 AOPs stand for the number of additions and MOPs for the number of multiplications.

Calculation	AOPs	MOPs
$\mathbf{B}_L$, $\mathbf{B}_N$, $\hat{\mathbf{S}}$	$25df_{e\ell}$	$31df_{e\ell}$
$\mathbf{B}_L^T\mathbf{C}\,\mathbf{B}_L$	$5df_{e\ell}(df_{e\ell}+6)$	$6df_{e\ell}(df_{e\ell}+6)$
$\mathbf{B}_N^T\hat{\mathbf{S}}\,\mathbf{B}_N$	$2df_{e\ell}(df_{e\ell}/3+3)$	$df_{e\ell}(df_{e\ell}+9)$
$\mathbf{K}^E$	$df_{e\ell}^2$	
miscellaneous	$5df_{e\ell}$	$7df_{e\ell}$
total	$6.\bar{6}df_{e\ell}^2+66df_{e\ell}$	$7df_{e\ell}^2+83df_{e\ell}$
8 Node Hexaeder	5424	6024

Table 4.1: *Operations needed to determine the Element stiffness matrix* $\mathbf{K}_E$ *for an Element with* $df_{e\ell}$ *degrees of freedom*

These operations have to be executed for every integration point. Thus, the total number of floating point operations (additions and multiplications) is:

$$N_K = n_{int}(13.\bar{6}df_{e\ell}^2 + 149df_{e\ell}) \tag{4.4}$$

N_K amounts to 90,000 floating point operations for an eight-node hexaeder Element with eight volume integration points.

One iteration of the BiCGSTAB (Pommerell 1992) version of the CG method requires two matrix-vector multiplications and 16 vector-vector operations. The matrix-vector multiplications constitute the most time consuming part. Generally, the vectors are of size df_g and the matrix has the dimensions $df_g \times df_g$. The matrix is only sparsely populated. On average, only df_{eav} values of a row are populated. Thus the two matrix-vector multiplications require $2(2df_{eav}-1)df_g$ operations. The calculation of the correction vectors amounts to another $20df_g$ operations. The total number of operations per iteration step are:

$$N_\varrho = 2df_g(2df_{eav}+9) \tag{4.5}$$

Since the amount of communication bandwidth required depends on the distribution of the Elements among the processors, I can only make an approximation here. There are two approaches to the parallel CG calculation. We can work with sub-domains in which case no global stiffness matrix is assembled. As a result, C_K is zero and all the communication load is moved to C_ϱ, such that the amount of data exchanged per iteration approximately equals the communication bandwidth required by the explicit integration discussed later. Alternatively, we can assemble the global stiffness matrix and assign a number of rows to each processor. This requires a considerable amount of data to be transferred when setting up the iterations, but no data exchange is required during an iteration. The amount of data communicated for each value of the global stiffness matrix depends on the partitioning of the model. We will assume that for each nonzero value, there is one data transfer, so that:

$$C_K \approx df_{eav} df_g \tag{4.6}$$

For each iteration, all processors need to know the current error. This error is determined from the partial errors calculated by the processors. There are two ways to make sure that all processors know the total error. All partial errors can be sent to one processor, which calculated the resulting error and sends it back to the others, requiring $2(n_p - 1)$ values to be transferred. However, the latency can be reduced if all processors accumulate the partial errors, which requires that every partial result is sent to every other processor, so that:

$$C_\varrho = (n_p - 1)^2 \tag{4.7}$$

4.2. Explicit integration

The computation of a time-step takes fewer operations with explicit integration than with implicit integration. However, the time-steps are much smaller for explicit integration, resulting in higher overall requirements, as I will show in this chapter.

The communication bandwidth needed in explicit integration is the same regardless of the method used to obtain the force resulting from the deformation of an Element — it was discussed in section 3.7. In the optimised implementation, shown in section 3.7.2, the data of every node is communicated $2(|\mathbb{P}_k| - 1)$ times. A statistical analysis of a distribution of the FE model among 12 processors shows that the position of a node is on average needed by $n_{pav} = 1.36$ processors. Thus, communication bandwidth C_{ex} is:

$$C_{ex} = \frac{2(n_{pav} - 1)df_g}{\Delta t_{ex}} \tag{4.8}$$

The calculation of the forces required to determine the new position of a node can be refined. Instead of multiplying the displacements with the stiffness matrix, direct vector calculation or reduced integration with hourglass control can be applied. These refinements will be discussed next.

4.2.1. Using stiffness matrices

If we calculate the stiffness matrix for every Element in each time-step, we can multiply the displacements with the stiffness matrix to determine the resulting force. This requires $df_{e\ell}(df_{e\ell} - 1)$ operations. The number of operations required to obtain the Element stiffness matrix is the same as for implicit integration shown in equation (4.4). To determine the force, we add the Element force vectors to the global force vector, which requires $df_g(n_{eav} - 1)$ operations. Finally we can calculate the new positions with an addition and a multiplication for each of the df_g degrees of freedom. The number of operations amounts to:

$$N_{ex1} = \frac{n_{e\ell}(N_K + df_{e\ell}^2 - df_{e\ell}) + df_g(n_{eav} + 1)}{\Delta t_{ex}} \tag{4.9}$$

4.2.2. Direct vector calculation

Rather than calculating a stiffness matrix we can calculate the stresses, strains and forces directly, as described in section 3.2. Table 4.2 shows how this reduces the number of operations.

Calculation	AOPs	MOPs
$\mathbf{B}_L$, $\mathbf{B}_N$, $\hat{\mathbf{S}}$	$25df_{e\ell}$	$31df_{e\ell}$
$\mathbf{B}_L^T(\mathbf{C}(\mathbf{B}_L\mathbf{u}))$	$11df_{e\ell}+24$	$12df_{e\ell}+36$
$\mathbf{B}_N^T(\hat{\mathbf{S}}(\mathbf{B}_N\mathbf{u}))$	$5df_{e\ell}+9$	$6df_{e\ell}+27$
$\mathbf{K}^E\mathbf{u}$	$df_{e\ell}$	
miscellaneous	$5df_{e\ell}$	$7df_{e\ell}$
total	$47df_{e\ell}+33$	$56df_{e\ell}+63$
8 Node Hexaeder	1161	1407

Table 4.2: *Operations needed per integration point for direct vector calculation*

The total number of operations executed for all integration points is:

$$N_{Ku} = n_{int}(103df_{e\ell}+96) \tag{4.10}$$

$$N_{ex2} = \frac{n_{e\ell}N_{Ku} + df_g(n_{eav}+1)}{\Delta t_{ex}} \tag{4.11}$$

4.2.3. Single point integration with hourglass control

The work described in (Hutter 1999) concentrates on further reducing the amount of operations required to obtain the forces from the displacements. The procedure we finally used is his *total hourglass control* which integrates at a single point ($n_{int}=1$). By analysing every part of the calculation, and removing parts which do not change from time-step to time-step, he could reduce the number of operations considerably. The implemented program requires 816 floating point operations instead of more than 20'000 operations required by equation (4.11) to calculate the forces. The number of operations required for explicit integration with total hourglass control is:

$$N_{ex3} = \frac{816\cdot n_{e\ell} + df_g(n_{eav}+1)}{\Delta t_{ex}} \tag{4.12}$$

4.3. Estimated numbers

To estimate the required computing performance, we have to make some assumptions for the parameters of the implicit integration. We are assuming that the global stiffness matrix is recalculated once every graphics frame ($\Delta t_{im} = 20\text{ms}$) and that $n_{it} = 100$ iterations are executed every frame. This allows for calculating several linear solutions with the same global stiffness matrix. The computational requirements for our 1840 Element model are shown in table 4.3.

Method (explained in section)	Ops/time-step	Ops/s
Implicit/CG (4.1)	339 M	17 GFLOPS
Explicit/matrix (4.2.1)	167 M	1.7 TFLOPS
Explicit/vector (4.2.2)	38 M	379 GFLOPS
Explicit/hourglass (4.2.3)	1.6 M	16 GFLOPS

Table 4.3: *Operations required for implicit and several explicit integration methods*

Table 4.4 shows the estimated communication requirements for different methods running on our parallel computer consisting of 12 dual processor nodes. The requirements for all explicit methods are similar.

Method	values/time-step	Bytes/s
Implicit	477 k	191 MByte/s
Explicit	5.5 k	437 MByte/s

Table 4.4: *Communication requirements for explicit and implicit integration*

As explained in chapter 2, we cannot use the Conjugated Gradient method with implicit integration, we have to use real-time explicit integration instead. From the numbers in table 4.3 we can conclude that only with total hourglass control the performance requirements for explicit integration were brought down to a manageable level.

To determine the number of processors required to deliver the required computational power, we assume that a highly tuned program can utilise a third of the theoretical peak floating point performance of a processor. For the RISC processors we use, this amounts to two floating point operations every three clock cycles. As our Alpha 21264A processors run at 666 MHz, we can expect a performance of 444 MFLOPS per processor. At this speed, we would require 36 processors to achieve our goal. With the 24 processors at our disposal, we can come close to achieving the original target.

An examination of a distribution of our FE model to the 12 nodes of our parallel computer shows that 40 kByte of data have to be exchanged in every time-step, or 400 MByte per second. This amounts to an average of 34 MByte/s per node that is sent and received. As we cannot expect to reach the maximum performance, the network needs a raw performance of at least a Gigabit per second per node.

5

Parallel Hardware

The parallel hardware consists of processing elements connected by a communication network. Numerous different hardware architectures are possible. In the first part of this chapter, we will take a look at how well different architectures are suited for real-time FE calculation. The second part of the chapter contains a description of the processing elements and the communication network that make up the LASSO hardware.

Commercially available communication networks are based on open standards such as Ethernet, HIPPI, or SCI, or custom implementations such as Myrinet. We could also build a custom communication network optimised for the requirements of the parallel FE computation.

We can also see several options for the processing elements. Standard components produced in large quantities, such as workstation processors and DSPs, deliver a lot of raw computational power. There are also less widely used or experimental designs such as the Intelligent RAM (IRAM), vector processors, or multi-threaded processors.

5.1. Processing elements

As found in chapter 4, the processing elements must be able to perform several billion floating point operations per second. This performance is achieved by running an adequate number of processing elements in parallel. The FE calculation must be performed with 64 bit arithmetic in order to avoid stability problems induced by a lack of numerical precision.

The information that must be stored for a node is contained in four three dimensional vectors and two scalars, a total of 112 byte. The total model with 2528 nodes needs 283 kByte for the floating point data plus at most the same amount for administrative information. As a processing element p does only need the information for the nodes in $\mathbb{K}_p$, we can be sure that no processing element needs to store more than 500 kByte of data.

Additional requirements were price, power consumption and space. The LASSO computer is located next to the ONYX II computer that renders the graphics, in a room with limited space and cooling capacity. The LASSO computer must fit in at most two 19” racks and it must not emit more than 10 kW of excess heat. We could spend 250 kCHF for the processing elements and the communication network to come as close to the original performance goals as possible.

5.1.1. Digital signal processor

DSPs are optimised for signal processing tasks such as Filters or Fast Fourier Transformations. Most are only capable of Fix-Point arithmetic, while a subset can also perform Floating-Point computations. A typical DSP has fast on-chip memory with a high bandwidth bus between the internal memory and the arithmetic unit, while the off-chip memory interface provides only a moderate bandwidth. The clock speed of the fastest signal processor has, for the last few years, been about a third of the clock speed of the fastest workstation CPU.

In the highly parallel parts of the FE computation, such as the recovery of Elements, several DSPs could carry out the task of a high-end workstation processor at a comparable price, power consumption and space. However, for sequential and data dependent parts, the DSPs would be much slower. At the time of evaluation, no DSP was capable of the required 64 bit floating point arithmetic. The necessity of constructing hardware and a suitable programming environment kept us from verifying the necessity of 64 bit precision in detail.

5.1.2. Multi-threaded processor

The multi-threaded processor (Tsai et al. 1999) is an experimental design, but it is shortly evaluated here because the technology is likely to be used in mainstream processors in a few years. Currently, the high-end general purpose processors use heavily pipelined designs with several parallel (super-scalar) execution units with high clock speeds. It takes several clock cycles until a computation is finished and the result is written back. If this result is needed in one of the next instructions, the execution is stalled until the computation is finished, thereby degrading performance and leaving several pipeline stages empty. This effect is countered with techniques such as optimising compilers that try to keep pipelines filled, and out of order instruction execution, where the processor postpones instructions for which operands are not ready and issues others which operate on different data.

An additional problem are conditional branches in the code. As the direction the branch must take is usually not known until immediately before the branch, the pipelines might become completely empty. Processors try to circumvent this with branch prediction, where the most likely branch is executed. Correctly predicted branches can then be processed at full speed, while mispredictions incur a penalty of several cycles.

The multi-threaded processor provides a way to eliminate these problems. It can work on multiple threads at the same time. Each thread has its own set of registers. The processor switches threads every time an instruction is issued. When a new instruction of a thread is issued, the last one has had enough time to complete, so pipeline stalls are avoided, and the available computation units on the processor are rarely idle.

In FE calculation, different Elements could be assigned to different threads, so that the time consuming recovery is calculated for several Elements simultaneously, with maximum performance. Other parts of the calculation are executed for each node or for each surface. Groups of nodes or surfaces can be assigned to different threads. All in all, the multi-threaded processor seems to be an interesting design for explicit FE computation once it becomes widely available.

5.1.3. General purpose processors

General purpose processors, which make up the core of personal computers and workstations, are designed to handle arbitrary tasks at high speed. At present they exceed 1 GHz clock speed, leaving other computation devices far behind. Being in a field of strong competition, they are in the centre of technological innovation, yielding ever higher performance. They also have fast high precision floating point units, which are needed for the task at hand. Altogether, general purpose processors deliver high performance at low cost.

A general purpose processor is not able to perform any computation without memory and I/O subsystems. The computational performance needed for the LASSO project can be reached by running several general purpose processors in parallel. This can be achieved by embedding each processor in a system and connecting the systems with a communication network. As a small number of processors (typically two or four) can share the memory and I/O subsystem without significant performance loss for each processor, we can connect symmetrical multiprocessing (SMP) workstations containing two or four processors with a communication network. Finally, the simplest programming model and the fastest communication is reached when all processors are connected to one virtual shared memory in a parallel super computer.

5.1.4. Cluster of workstations

A cluster of workstations is a parallel computer where several standard workstations with one or more processors are interconnected with a high speed network such as Myrinet or Fast Ethernet. In the last few years, Beowulf clusters that run the free GNU/Linux operating system (Sterling et al. 1995) have become popular. The key-acronym is COTS: Commodity Off The Shelf components provide the buyer with the good price-performance ratio of the commodity market. Using widely available tools, these systems are easy to set up, leaving time for the development of the parallel program.

As the LASSO hardware was bought near the end of the project, we were able to capitalise on the fast improvements in processor technology, enabling us to buy high performance workstations for a moderate price. Using a Beowulf cluster has additional advantages. Available standard message passing libraries, such as the message passing interface (MPI), make it simple to port software to the cluster, turning it into a powerful machine for different parallel computing needs. Even in our project, the use of MPI makes it simple to run the same program on a development computer and on the target Beowulf cluster.

In a cluster of workstations, information is exchanged with message passing; a processor sends a message to a peer that has to issue a receive call before it can use the received data. If a workstation contains more than one processor, the communication is no longer uniform; depending on the location of the communication partners, the shared memory or the communication network is used to exchange data. This additional complexity can either be hidden by the message passing layer in order to get a uniform programming model for the parallel program, or it can be passed on to the higher levels in order to obtain maximum performance.

5.1.5. Parallel super-computer

Similar to a Beowulf cluster, a parallel super-computer contains fast general purpose processors connected with a communication network. This communication network tightly couples the processors by providing high bandwidth and low latency. The memory in such a computer can often be accessed uniformly as shared memory, providing a method to exchange data between processors that is simpler to use than message passing. Because additional hardware must be built specifically for such a parallel computer, it is significantly more expensive than a cluster of workstations with the same amount of processor. Such computers are often shared by several users running batch processes. The interactive nature of our simulation makes such sharing rather difficult. We therefore decided against such a computer.

5.1.6. Vector super-computer

Vector super-computers comprise one or several custom-designed vector processors that are able to process vectors containing thousands of elements at great speed. Vector super-computers were widely used in the eighties but have since been replaced by computers with many standard processors running in parallel. The vectors appearing in explicit FE calculation contain at most 24 numbers, a size at which vector computers do not excel. Additionally, the cost and size requirements cannot be met by such a computer, which kept us from further evaluation.

5.1.7. The case for fast processors

To get the required performance, we can chose from different processors with different price-performance ratios. For fast processors, the prices increase more than linearly with speed. The nonlinear relationship of prices and processor speed is illustrated in figure 5.1 with the example of AMD processors. The cost of a complete system is made up of much

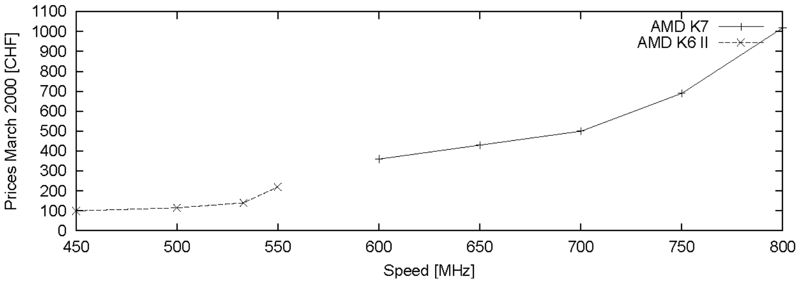


Figure 5.1: *Processor speed vs. price*

more than just the price of a processor, however the nonlinear price-speed ratio holds for most components. We therefore had to make a tradeoff between many slow and a few fast systems to reach the desired performance.

An examination of the partitioned model shows that the communication load for slow systems remains high and requires the same kind of performance as the fast systems. Thus the cost of the communication system per node stays the same and defines a lower bound for the price per node. A parallel computer constructed of many inexpensive systems has several drawbacks, such as increased consumption of power and space, increased probability of failure of a component and the lack of wide PCI buses with an ensuing negative impact on communication bandwidth. The main problem, however, is that the slower the systems are, the lower the overall performance that can be reached, despite constant peak performance.

In a simulation, I compared virtual parallel computers with processors of different speeds to find out the impact of the speed differences of processors. Three different processor models are used: The “fast” processor, the “moderate” processor, and the “slow” processor. In the first system (fig. 5.2), 24 fast processors are used. The moderate processor delivers half the performance of the fast processor, we therefore would have to use 48 moderate processors to get the same raw performance. The slow processor has a quarter of the speed of the fast processor, so 96 slow processor deliver the same peak performance as

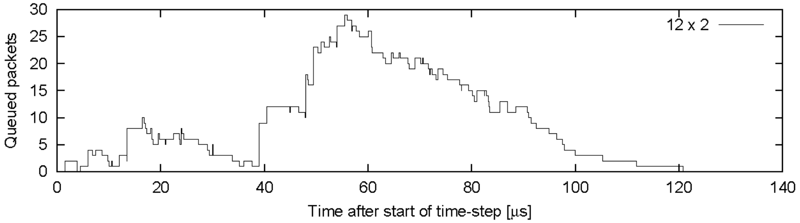


Figure 5.2: *Network occupation for 12 fast two-way SMP systems*

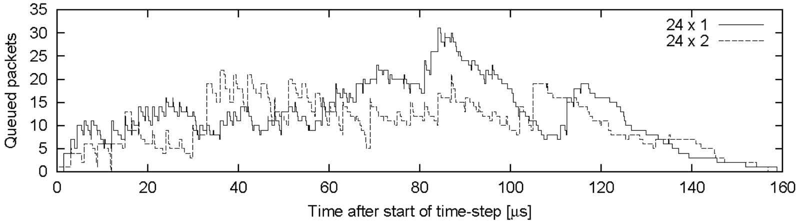


Figure 5.3: *Network occupation for 24 fast single processor nodes and 24 moderate two-way SMP nodes*

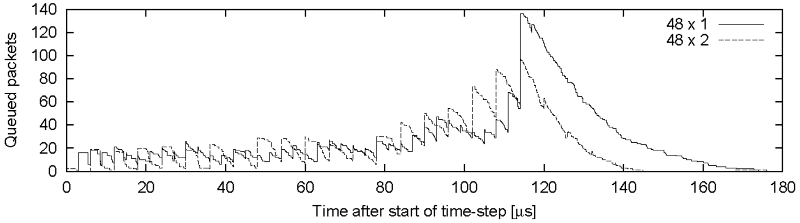


Figure 5.4: *Network occupation for 48 nodes consisting of moderate single processor systems or slow SMP systems*

24 fast processors do. The fast and moderate processors were simulated in single processor and two-way SMP systems, the slow processor was only simulated in a two-way SMP configuration. All systems have the same communication system, with the performance characteristics of an existing network.

For the comparison, I generated distributions of the FE model for the given number of processors. The distributions were used to extract communication information; for every processor, a list was generated showing the sequence of Elements that are recovered and data that is sent. This sequence was used to predict occupation of the communication network. In this prediction, the recovery of Elements causes a delay depending on the speed of the processors, and the transfer of a packet occupies the network. A packet starts occupying the network when it is sent, and the duration of the occupation is determined by the bandwidth of the network and the size of the packet.

Figures 5.2 to 5.4 show the predicted occupation of the network during a time-step. We can see that with slower processors, the exchange of data delays the end of the time-step significantly. There are several reasons for this: the slower processors need more time until results are ready and can be communicated, so the peak load is moved towards the end of a time-step. The smaller sub-domain processed by a node has a bigger percentage of surface nodes, so the number of Elements not needed for communication decreases, no work remains once all data transfers are started. Finally, the time needed for data transfer does not decrease despite a lower load per processor, because packets just become smaller and the available bandwidth decreases accordingly. We can conclude, that it is best to use the fastest components available when building a parallel computer.

5.2. Communication

As determined in chapter 4, the communication network must have a bandwidth of at least one Gbit/s. Latency should be only a few microseconds, and sending and receiving data should consume only minimal processing time. Ideally, a processor should be able to directly store data in the memory of a remote processor.

Processing time can be lost at different points during communication: when data is prepared to be sent, when a processor waits for data before it can go on with its work, and when a processor needs to interpret received data. The first and the last reason depend mainly on the communication software and are discussed in chapter 7. How much of the work needed for sending and receiving is taken off the main processor depends on the abilities and flexibility of the network hardware. The middle part depends on the networking hardware and the scheduling of corresponding send and receive calls. The faster the individual components are, the smaller the latency becomes, and the higher the bandwidth, the lower the occupation of the lines.

5.2.1. Dedicated point to point links

Any pair of processors must be able to exchange data. We can use dedicated point to point links between neighbours in the three dimensional mesh, or we can use a switched network.

Dedicated point to point links require a cable for every local channel. To implement this in hardware, we would require a maximum of seven cables out of a processor, i.e. one to each neighbour and one for global communication. One major drawback is the amount of cabling required, e.g. a $4 \times 4 \times 4$ processor machine would require 144 cables for local communication. A second problem is the connection of the computers to the network. We could put six network cards into every workstation, each of which is only active a fraction of the time. The price of six network cards that individually fulfil the specifications would be prohibitive.

Instead of using six network cards, we could use some that have more than one connection. One suggestion is shown in (Rhomberg et al. 1998), i.e. a card with a single network interface followed by a switch that routes the traffic to one of six neighbours. The receiver has to select one of the incoming channels to determine the neighbour from which data can be received. Such a network interface card would have to be custom made, and it would therefore be useless for anything else.

With dedicated lines for local communication, it is possible to simply send the data as noted in section 6.3. If data is received over a line, the sender is known. If the sequence in which data is sent is fixed, the data can be identified without additional information. This minimises the amount of data that is sent and eliminates the need to send data in packets.

5.2.2. Switched network

While dedicated links provide the means to exchange raw data, a switched network is used to send data with attached information through a network of wires and switches. Data with different meanings and destinations can be sent simultaneously over the same network. On one hand, this introduces additional overhead for the generation of a packet to be sent and for the detection of the purpose of a received packet, as well as latency when the packet is moved through switches. On the other hand, this method reduces the amount of cabling and network cards required and provides the means to handle global and local communication in almost the same way. By optimising the topology of the network, the total bandwidth also increases with the amount of processors in the system. In the same way, the number of switches on the path between any two processors can be kept to a minimum, when the 3D structure is kept in mind. Furthermore, if a single card with the attached network is not enough to handle all the local and global communication, the load could be split between more cards, which ideally reside on separate PCI buses, connected to different networks.

5.2.3. Available hardware

A number of vendors sell hardware adhering to different standards, with widely varying features. Many of the cards can be used both with dedicated links, although with the mentioned high requirements to hardware and cabling, and as switched networks.

HIPPI is an abbreviation for HIgh Performance Parallel Interface. It was originally intended as a fast point to point link connecting supercomputers. It sports a parallel line that transfers 100 MByte/s or 200 MByte/s over point to point links, which can be connected to larger networks with crossbar switches. A number of ANSI standards specify different parts of the network. While the parallel cable spans distances of up to 25 m, there is also a serial specification, which allows distances of up to 10 km over optical fibres. A newer projects aims to develop HIPPI-6400, a network with a bandwidth of 6.4 GBit/s. Although HIPPI aimed to expand into the commodity Gigabit market, it was never widely used as a cluster interconnect.

Gigabit Ethernet is an extension of the popular Ethernet and Fast Ethernet, which are the most widely used standards for LANs. Gigabit Ethernet is an IEEE standard (802.3z/802.3ab) and runs at a speed of 1 GBit/s. It is mainly used to connect Fast Ethernet networks with servers and as a backbone.

SCI is short for Scalable Coherent Interface, a standard for a fast network for clusters. It was designed to model a computer bus in a distributed environment. It has point to point links with a bandwidth of up to 1 GByte/s. The basic communication function implemented in SCI is a remote memory access; i.e. a part of the user space memory is mapped to the SCI card, and any stores to that memory are automatically recognised, sent over the network and stored at the appropriate place in the user space of a different processing node. This sounds *exactly* like what we need. However, at the time the first network was bought for our project, the existing SCI implementations were expensive and, according to other users at the ETH, were not yet mature enough for commercial applications.

Myrinet is a Gigabit LAN/SAN network developed by a spin-off company from Caltech. It features full duplex links with 1.28 GBit/s bandwidth in each direction that connect to switches and flexible NICs. While the design of the switches is fast and simple, with wormhole routing, the interface cards are unusual (Boden et al. 1995). Next to the network and PCI interface, Myrinet network cards have a specialised RISC processor that handles outgoing and incoming data. Full development tools are available for this processor, allowing the owner of such a card to develop custom low-level protocols. In the time until the final cluster was purchased, Myrinet not only proved to be able to deliver high performance communication in our application, it also was about to become the standard high-speed cluster interconnect network.

5.3. Final hardware implementation

In the end, we decided to buy and construct a Beowulf cluster consisting of Alpha-processor nodes, connected by a Myrinet network. We chose the Alpha processors, because they deliver the highest floating point performance of all available microprocessors. High single processor performance has many advantages:

- It reduces the number of nodes necessary and the space they take up.
- Consequently, the overall communication requirements are reduced.
- Serial parts of the computation are completed more quickly.

- The lag from the beginning of a time-step until the first data is exchanged is reduced, and thus the communication load can be more evenly spread over the time of a time-step.

The choice of Myrinet networks was also based on several reasons:

- It fit the specifications in respect to speed, price, availability, power and reliability.
- The design with the special processor allows the implementation of the envisaged fast user level protocol.

5.3.1. Overall system design

The complete software system for the LASSO surgery simulator consists of a number of asynchronous processes (cf. fig. 5.5). Every set of two processes can exchange data, via shared memory if they reside on the same computer, via messages sent over an external network otherwise. As we use a modular concept, the processes can be distributed on different computers and replaced by dummy programs for test purposes.

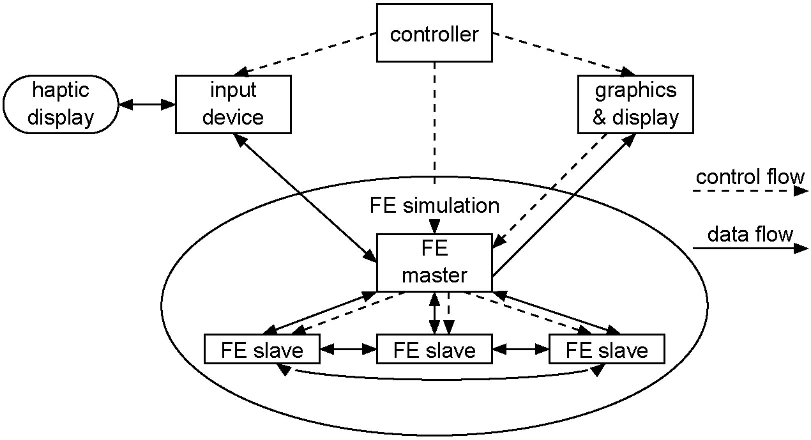


Figure 5.5: *Concurrent processes in the LASSO system*

At the core of the simulator is the FE simulation. It consists of a master which is close to the controlling process and the slaves which compute the mechanical behaviour of the organs in parallel. The master gets position information from the input devices and forwards the positions of the organ surfaces to the graphics device and information about reaction forces to the haptic display. The haptic display either connects to a program providing a pre-programmed sequence of inputs for development, to a mouse which is used for position input, or to a haptic display designed to provide the surgeon with tactile feedback. The graphics device uses the positions of the organ surfaces to calculate a photo-realistic image of the operation scene. Whenever a new image is to be calculated, the graphics device sends a request for the current positions to the FE master that answers with up to date information, so latency between manual input and visual feedback is minimised. The controller starts all other processes and sends them the appropriate configuration files, and then waits for any exceptions or for the command to terminate the simulation.

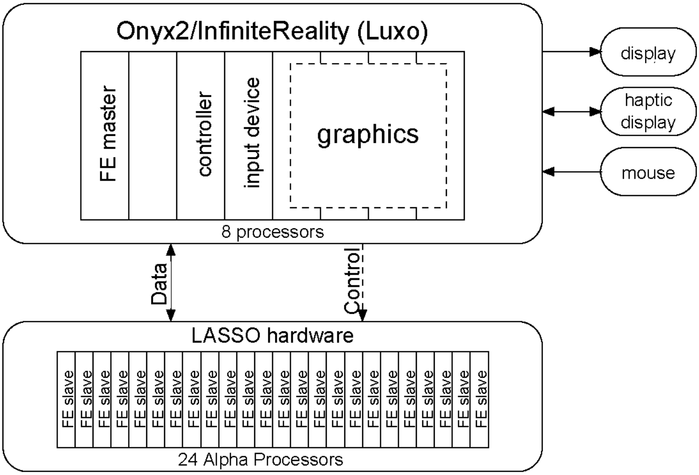


Figure 5.6: *System hardware*

All the parts of the surgery simulator except the FE engine run on an eight processor SMP Onyx2/InfiniteReality computer by SGI, Inc (cf. fig. 5.6). It has specialised render pipes for accelerated graphics calculation, needed to render the visual feedback. Despite improvements in 3D graphics accelerators for personal computers (mainly for games), the methods used for generation of the graphics are far beyond the capabilities of graphics cards and use half of the processing power of the SGI computer. The other half controls the data transfer to and from the LASSO hardware and the haptic display. During development of the surgery simulator, these four processors were used to calculate a small FE model.

5.3.2. Finite Element hardware

The hardware used for FE calculation is a Beowulf cluster (Sterling et al. 1995). The cluster comprises 12 workstations in 3U rack-mount cases, all together in a 19” rack. An image can be seen on page 22. Every workstation contains an UP2000 dual processor motherboard with two Alpha 21264 processors by Alpha-Processor, Inc., running at 667 MHz, 256 MByte of RAM, and a 9 GByte harddisk. This adds up to a total of 3 GByte of RAM and over 100 GByte of harddisk space, with a theoretical aggregate peak performance of 32 GFLOPS. The boards are connected with a standard switched 100 Mbit/s Fast Ethernet network and with a Myrinet network which provides 1280 Mbit/s full duplex bandwidth per link (fig. 5.7). The Myrinet switch is a full crossbar 16-port switch, with a maximum bisection bandwidth of 16×1280 Mbit/s. The Ethernet is used for standard services such as booting via the network, network file system (NFS), remote procedure calls (RPC) and other administrative tasks. Parallel programs running on the system can use the Myrinet network for data exchange between computers. A terminal server allows to control the consoles of headless nodes over the Ethernet network.

5.3.3. Alpha processor nodes

The individual nodes are built with UP2000 Dual Alpha motherboards by Alpha-Processor, Inc. (cf. fig. 5.8). In such a system, two Alpha processors of the third generation (21264A) are connected to the main memory and the I/O subsystem via the 21272 chip-set. The chip-set consists of a C-chip, two P-chips and four D-chips. The D-chips provide access to the main memory, each controlling a 64 bit wide portion for a total memory bus width of 256 bit. The C-chip interfaces with the processors, while the P-chips each control an individual 33 MHz PCI bus. The chip-set is running at a speed of 83 MHz, so that the main memory can be accessed with a bandwidth of over 2.6 GByte/s. Of the two PCI-buses, one is 32 bits wide and is used for the Fast Ethernet connection and for other cards, if any. The 64 bit bus, with its peak bandwidth

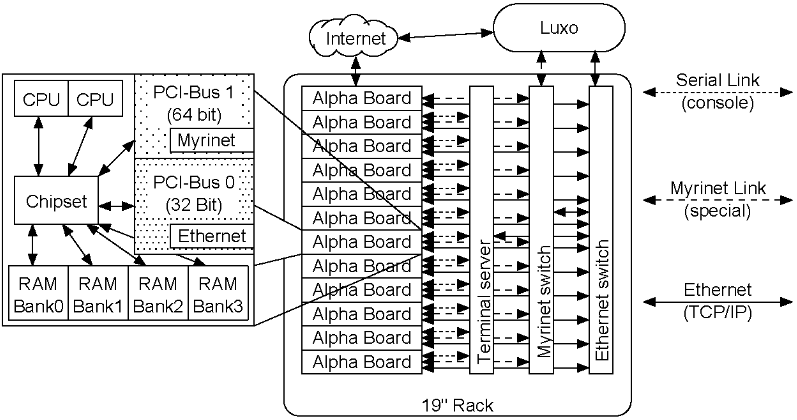


Figure 5.7: *Parallel Hardware*

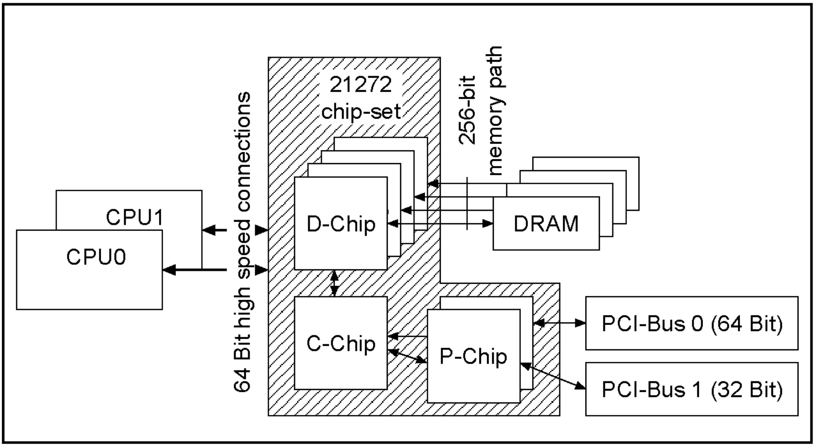


Figure 5.8: *UP2000 Motherboard overview*

of 264 MByte/s, is exclusively used for the Myrinet card. The motherboard also features IDE and SCSI hard-disk controllers next to the standard I/O interfaces such as parallel, serial, mouse and keyboard.

The two Alpha processors are on a daughter-card with 4 MByte of second-level cache. The processing core is connected with 64 kByte of 64 bit wide first level cache. The second level cache is connected with a separate 128 bit wide bus, while a third bus which is 64 bit wide, provides the connection to the chip-set. This bus is kept narrow to reduce pin count, but it runs at speeds of up to 333 MHz, so the high memory bandwidth can be used effectively. The main processing core contains a floating point ALU with a multiplication and addition pipeline. The execution unit can issue four instructions in every cycle, i.e. two floating point and two integer instructions. An out-of-order instruction execution unit and sophisticated branch prediction help to keep those pipelines filled.

5.3.4. The Myrinet network

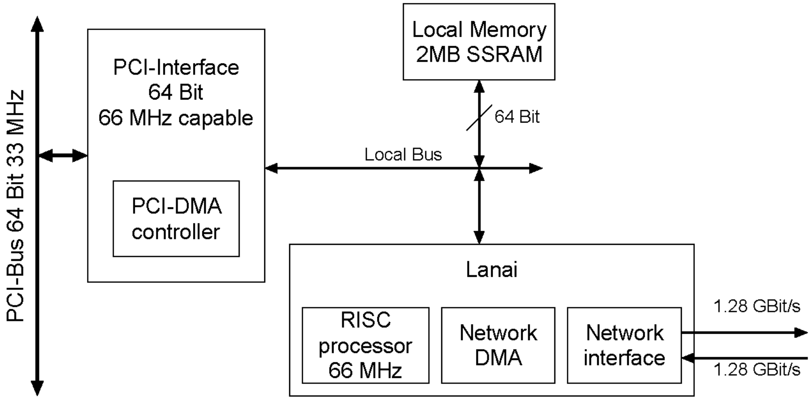


Figure 5.9: *Myrinet network card*

In each node of the parallel computer, there is one PCI64A Myrinet card (cf. fig. 5.9). A 64 bit 66 MHz capable PCI interface connects the card to the computer, automatically adapting to the size and speed of the actual PCI bus. In our case, it runs in 64 bit 33 MHz mode. A PCI DMA controller can transfer data directly between the computer's main memory and the fast local memory on the network card. It can autonomously process a chain of DMA transfers in both directions. The memory on the card can also be accessed directly from the PCI bus. On the network side, there is another DMA controller which can transfer data between the local memory and the network interface, and the Lanai, a special purpose RISC processor running at 66 MHz. The local memory, 2 MByte of synchronous SRAM clocked at 132 MHz, is used to exchange data between the two ends. The high clock speed of the SRAM allows access from both the Lanai and the PCI bus at full speed, so the memory can effectively be used like dual ported RAM.

The lifetime of a communication packet begins when a processor encounters a command to send data. The data is collected and transferred to the Myrinet card. Once a packet is assembled in the local memory, complete with headers, the Lanai initiates a DMA which copies the packet to the outgoing network interface. The headers required by the Myrinet include a routing header, followed by two bytes of type information. Once the DMA transfer is finished, a so called tail flit is automatically added, indicating the end of a packet.

When the packet arrives at a switch, the first byte of the routing header is stripped off and examined and the rest is transferred to the outgoing port that is indicated by that first byte with wormhole routing. In this way, each switch encountered on the way strips off one byte, until the packet arrives at the destination network card. After all routing bytes are stripped, the type field becomes the head of the packet and is transferred by DMA to the local memory, along with the payload. Flow control on the links uses backpressure, a simple **stop** and **go** protocol. When the short slack buffer of a receiver is full, a **stop** signal is sent back, which is followed by a **go** signal once the slack buffer becomes sufficiently empty. The Lanai receives a signal indicating that a packet was transferred to the local memory and starts to process it. It will first examine the packet type, reject unknown packet types and initiate some sort of reaction for known types. It can either inform the main processor of the arrival of data with an interrupt, or it can first start a DMA to move the packet to main memory and tell the processor about it later, or allow it to find out by polling that location.

6

Parallel Program

When a program is distributed among the processors of a parallel computer, overhead occurs since data must be exchanged and the processors must be synchronised. This causes a loss of time, that can be minimised by adapting the program to the parallel computer. For the FE program, the communication provides an implicit synchronisation. Communication and computation are executed concurrently, hiding the communication latency and minimising the time spent waiting for data.

Figure 6.1 shows the flow of control information for the parallel program. The control information is generated offline, and is used to control the data flow of the parallel program. Given the FE model and the structure of the parallel computer, an optimised assignment function is found (cf. section 3.5). The partition program then creates a schedule file for each processor. This schedule describes the order in which data is to be computed, sent and received during every time-step.

In this chapter, we will first examine the parallel FE program: the topology simulator used to run the program on different platforms, the communication interface, the way data transfer and synchronisation is handled, and the schedule information. I will then explain how the partition program generates the schedule files and how an optimised

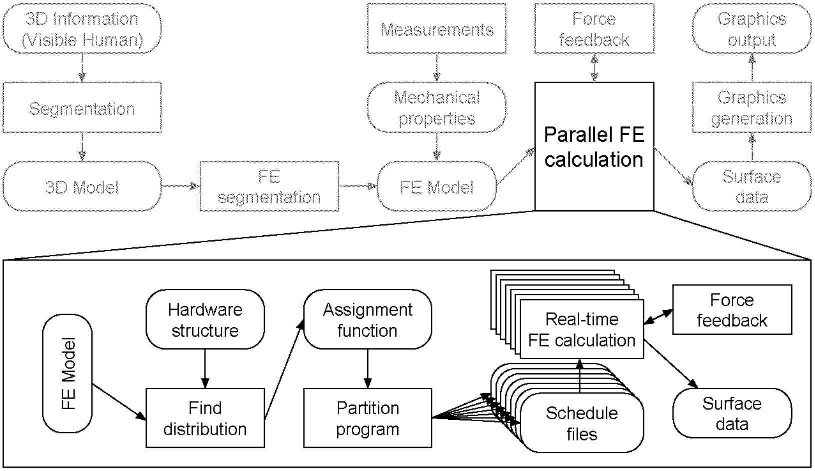


Figure 6.1: *Flow of control information of the parallel FE calculation*

distribution of the FE model is found. At the end of this chapter, there is a description of the method used to verify the schedule files and the parallel FE program, and a section about the importance of optimising compilers.

6.1. Overview

Figure 6.2 shows the control flow of the parallel FE program. First, the prepared schedule file is read and data structures are initialised accordingly. Administrative data, such as pointers and communication headers, that stays the same during the execution of the program is pre-calculated. During the simulation, one time-step after the other is computed. In a time-step, requests for graphics data and positions of the manipulators are read, if available. Data is sent back if necessary. Then the *batches* listed in the schedule file are processed. A batch consists of commands to recover Elements, to send and receive data, and to calculate the new positions of nodes.

Figure 6.3 shows the different layers of communication and synchronisation in the parallel FE computer. The communication interface described in section 6.3 is simple and efficient; the time needed to send

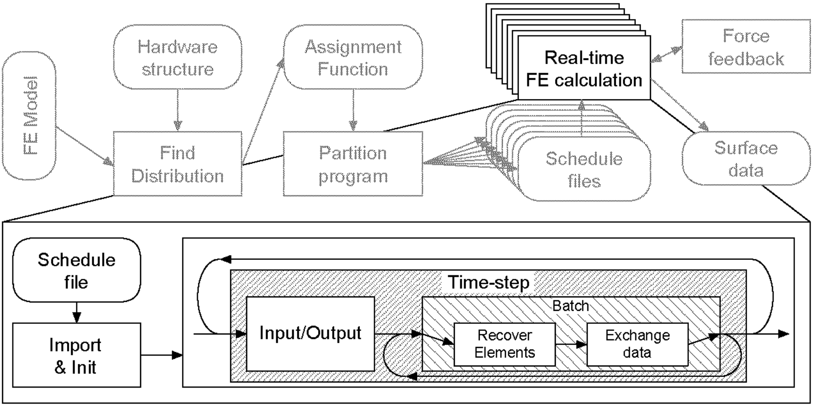


Figure 6.2: *Data flow of the parallel FE program*

and receive data is minimised. Since the processors may need different amounts of time to process their data, they have to be synchronised in order to ensure the correct interpretation of received data. Finally, the parallel computer has to exchange data with the display computer that controls the input and output devices. Again, synchronisation ensures correct interpretation of exchanged data.

6.2. Topology simulator

I first wrote a parallel simulator that was used for the development of the parallel FE software. This simulator makes the testing of the parallel FE software possible, and it speeds up calculations of FE models, because the computation can be executed in parallel on a cluster of workstations or on a parallel computer (fig. 6.4). The early implementations of the laparoscopy simulator, running on the 8 processor Onyx II computer, used this simulator. The same program was then compiled and linked with a different communication library to run on the target parallel computer described in chapter 5.

The simulator consists of an initialisation procedure and a communication layer and works in a master-slave principle. The master helps to set up communication channels, provides input and handles output, while the slaves do the number crunching. At startup, the master sends initialisation information to the slaves and helps to set up local com-

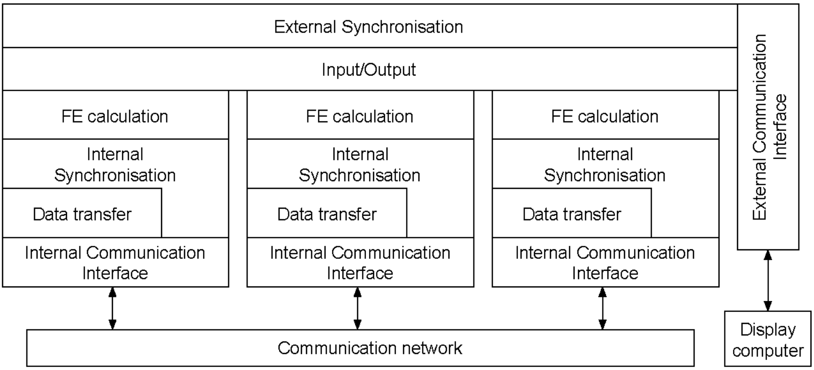


Figure 6.3: *Layers of synchronisation and communication in an example with three processors*

munication channels between neighbours in a virtual three dimensional mesh.

While a simulation runs, the slaves exchange data through local and global communication. The communication layer sends this data to other slaves over the available hardware, such as Ethernet or shared memory. The master handles data transfer to other parts of the laparoscopy simulator, such as force feedback and display, and it can collect debugging information.

6.3. Communication interface

The communication interface is a consistent API that provides the same functions to the FE program, regardless whether the simulator or the parallel FE computer is used to run the program. To transfer data between processors, local and global communication is provided. Local communication is used to transfer data between neighbours in the three dimensional mesh, while global communication provides the means to transfer data between any two processors (cf. fig. 3.2 on page 46). Data transfer on local channels adheres to a fixed schedule, a property that can be exploited when optimising local communication, while the global communication must be able to handle arbitrary unscheduled data exchanges.

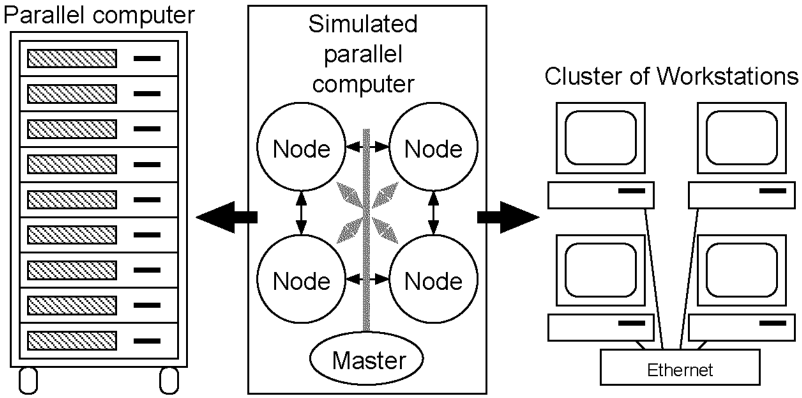


Figure 6.4: *The topology simulator provides the means to run the same parallel program on different platforms. The simulator provides local and global communication over the available communication networks.*

6.3.1. Local communication

A local communication channel provides a direct communication link between two neighbours in the three dimensional mesh of the parallel computer. In every time-step, a processor calculates the same Elements and sends the same data. By sending this data over a direct link, the order is preserved and data can be identified by the order in which it is received. The interface of local communication consists of four calls: `Queue()`, `Send()`, `Reset()`, and `Receive()`.

When data is sent, it has to be gathered and copied to the network interface card (NIC), either by the processor or by direct memory access (DMA). For the small packet sizes involved, the average packet contains less than 500 Bytes, the overhead to set up a DMA transfer does not pay off (cf. section 7.6). Therefore, the `Queue()` call writes data directly to the NIC, combining this copy operation with the gathering of data. The `Send()` call is used to add a header to the packet and start the transfer of data over the network. On the receiving side, data is directly written into a buffer in user space. For each time-step, data with the same meaning is written to the same location. The `Receive()` call polls this location until the data arrives. The `Reset()` function, called by the sender at the end of every time-step, prompts the distribution of a reset

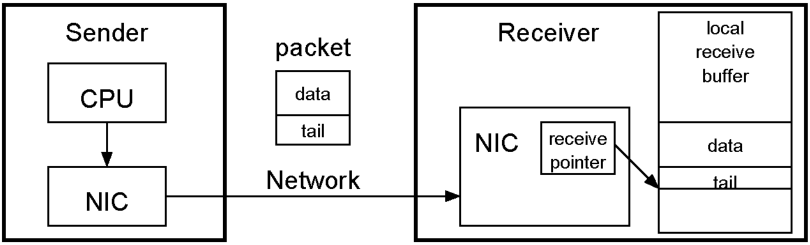


Figure 6.5: *Local communication data flow. The tail is needed for end-to-end synchronisation.*

packet to all neighbours. The neighbours react by resetting their input buffers, thus getting ready for the data of a new time-step.

When dedicated links are used for local communication, the data need not be sent in packets. In this case, the `Queue()` call directly causes the sending of data, and the `Send()` call becomes a no-op.

Figure 6.5 illustrates the data flow in local communication. A tail containing the number of the current time-step is added to the data, providing end-to-end synchronisation. The receiver checks for the arrival of data by polling until the correct tail arrives.

6.3.2. Global communication

Global communication provides a method to send data from one processor to any other processor. Several processors might send packets simultaneously to the same receiver, causing the packets to arrive in arbitrary order at their destination. To ensure that the data is processed correctly, packets need identification information. In the implementation used for the FE simulation, this information consists of the packet's length and of an address in the receiving processor's memory. The `StartPacket()` call is used to initialise a packet and to provide a destination address. Data is then queued and the packet is sent as in local communication, with a `Queue()` call and a `SendPacket()` call. In the receiving NIC, the packet is analysed and stored at the given address in the receiving buffer (fig. 6.6). For every piece of data sent on the global bus, space is allocated in this receiving buffer when the parallel computation is initialised. Although packets arrive in varying

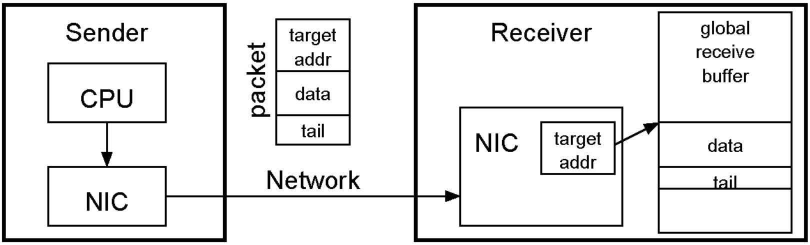


Figure 6.6: *Data flow in global communication. The data is written to the provided address in the receiver's buffer.*

orders, they are sorted without processor interference and the receiver finds the data at the expected location, after using the `Receive()` call to poll until the correct tail is read.

6.4. Data transfer

The communication interface provides methods to store data in the memory of remote processors. The buffers used for communication are statically allocated, and constantly overwritten with new information. We must ensure that no data is destroyed before it was read by the receiving processor, i.e. we must send some kind of acknowledge back to free a buffer. An additional message just for this acknowledge would consume processor time and network bandwidth and should be avoided. Instead, we can profit from our knowledge about the symmetric nature of communication; if node a sends to node b, we know that node b sends data to node a in the same time-step. Thus if a receives data from b in time-step $n+1$, it knows that b has finished step n and therefore read the data received in that step.

As shown in figure 6.7, we can simply use two alternate receive buffers for each message. Buffer 0 is used in even time-steps, buffer 1 is used in odd time-steps. Let us assume that n is even and that we are at the start of step $n+2$. In order to arrive here, we had to finish step $n+1$. Thus, we can be sure that all communication partners did at least reach step $n+1$, meaning that all of them did finish step n. We are now free to write information to the buffer 0 of a communication partner, as this buffer was last used in step n.

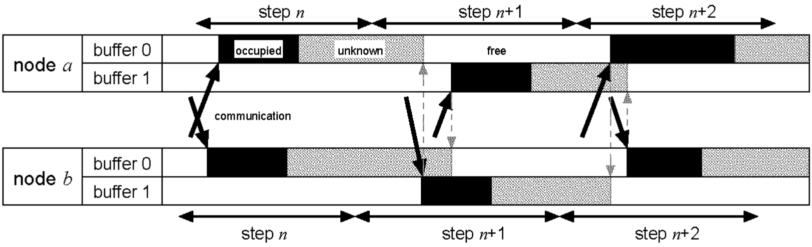


Figure 6.7: *Double buffering and symmetric communication is used to assure data integrity. Dark arrows show communication, light arrows indicate that the buffer pointed to is known to be free at that time.*

6.5. Synchronisation

In our parallel computer, many components run at different speeds. Even the parts of the parallel computer that perform the FE computation spend different amounts of time for their tasks, because no two tasks are exactly the same. Figure 6.8 shows the different components that are of interest here. The synchronisation between processors and NICs is the task of the communication interface and is therefore discussed in chapter 7.

6.5.1. Synchronising the FE computer with I/O

During a simulation, the FE computer has to be synchronised with I/O, that is the force feedback device and the display computer. The FE computation runs at the rate of 10’000 steps per second. The graphics engine should produce 24 images per second for smooth animation and the force feedback device needs to update the forces fed back to the user at least 500 times per second. To produce a new image, the display computer needs the current positions of the surface nodes. Since the time it takes the display computer to generate a single image varies, the display computer rather than the parallel computer has to decide when new position information is required. When the display computer encounters a need for new surface data, it sends a corresponding request to the FE computer. To render an image correctly, the display engine needs consistent information. That is, all positions must be taken from the same time-step. The nodes of the parallel FE computer might work

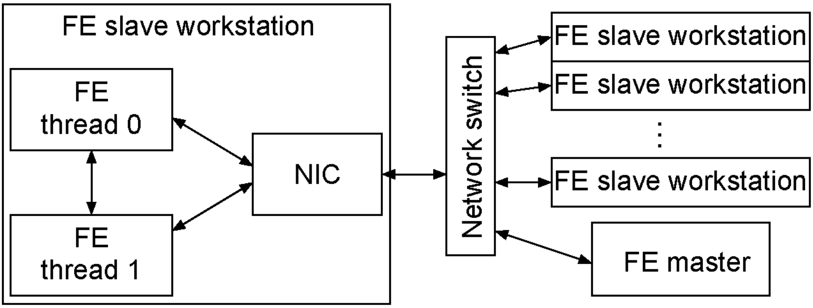


Figure 6.8: *Components that run asynchronously from the point of view of one processor. All components shown have to be synchronised with this processor. Within the same workstation, there is another processor occupied with a second thread and the NIC with its own processor. Through the network, the processor is connected to the other nodes of the parallel computer and to the display computer that provides input and output.*

on different time-steps, so a snapshot of the current positions on all nodes might be inconsistent. To get a complete set of surface positions, the display engine sends a request for the positions at a time-step in the near future, d steps after the current time-step k. This message is sent to the one node of the parallel computer responsible for coordination with the display computer. It notes its own current time-step as having number k and sends a request for the surface data of time-step $k + d$ to all others. As one node after the other reaches the end of time-step $k + d$, it sends off the corresponding surface data to the display computer. This in turn waits until all data has arrived before starting to render the new image.

Compared to graphics output, the method used for handling force feedback has two minor differences. First, the data flows in both directions; forces are sent to the manipulator and positions are returned. Second, the exchange of manipulator data occurs more often but uses less bandwidth than graphics data, as packets consist only of three numbers, either a three dimensional position or a force vector. Again, all processors must use the new positions in the same time-step or the calculation becomes unstable. As with display data, one node receives the new positions for time-step $k+d$ at time-step k and sends them on

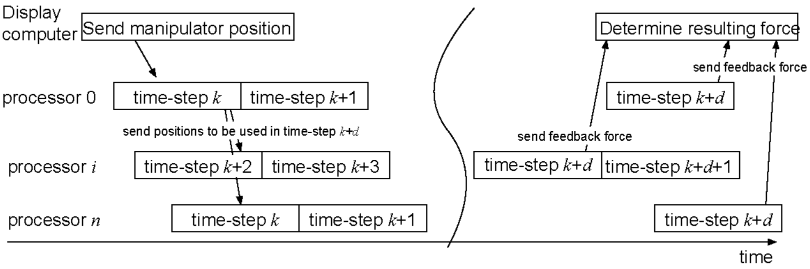


Figure 6.9: *Exchange of forces between the display computer and the parallel computer. The new input is sent to processor 0 at time-step k and hence forwarded to the others. As each node reaches time-step $k + d$, it uses the new input and sends back the resulting force. The master waits until all answers have arrived and calculates the resulting force for the manipulator. Display information is handled similarly, with the difference that only a request is sent instead of input.*

(fig. 6.9). When the nodes reach time-step $k + d$, the processors start using the new position and send back their reaction forces. The number d corresponds to the longest distance of any node from the coordination node, as explained in the next section.

6.5.2. Synchronising components of the FE engine

The processors of the parallel computer have to be synchronised to ensure that the communicated data is always processed in the correct time-step. The simplest way is barrier synchronisation; every node computes a time-step, waits until all others are done as well, and then proceeds to the next time-step. However, barrier synchronisation is a time consuming operation, as shown in figure 6.10. For a barrier synchronisation, all nodes have to report their readiness either to all others or to a central node that in turn allows the nodes to continue once all are ready. In standard implementations, this procedure alone takes a large part of the total time allowed for a time-step.

We use a loose form of synchronisation, that is more difficult to control but which uses less overhead than barrier synchronisation. By examining the communication, we can see that the same pattern is repeated in every time-step, and that the amount of data exchanged

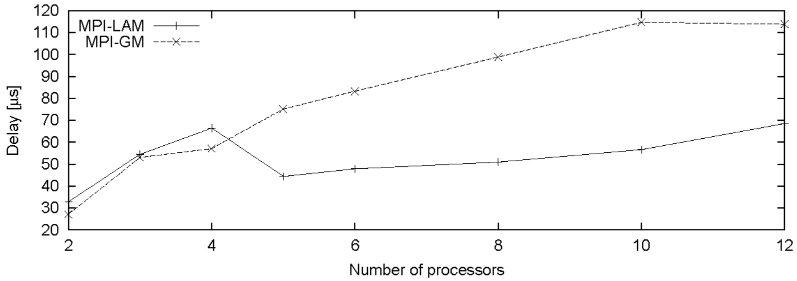


Figure 6.10: *Speed of MPI Barrier synchronisation for different numbers of processors. Two MPI implementations, with their implementation dependent differences, were measured on our parallel computer, MPI-LAM over Ethernet and MPI-GM, an MPI implementation that uses the GM low level layer on Myrinet. Chapter 7 contains more information about these MPI implementations.*

between any pair of nodes is the same for both directions. When two nodes exchange data, either both send partial forces or one sends a force and the other answers with a new position. In the second case, the synchronisation is automatically provided within a time-step, but in the first case, some analysis is necessary. In time-step $n-1$, data is sent by node a to node b, as illustrated in figure 6.11. a then waits until data arrives from b and proceeds to time-step n where it sends new data. As soon as the new values from b arrive, node a is sure that b has completed time-step $n-1$ and is computing step n. The difference in simulation time between a and b is therefore guaranteed to be less than one time-step.

With this knowledge, we can calculate the maximum difference in simulation time in the parallel computer as illustrated in figure 6.12. Any two nodes that exchange data during normal FE calculation are always less than a time-step away from synchronous operation. The maximum difference between a pair of processors can thus be found by determining the smallest distance in communication steps between them. For example in a computer with 12 nodes in a $2 \times 2 \times 3$ arrangement using only local communication, the maximum difference in simulation time is four time-steps.

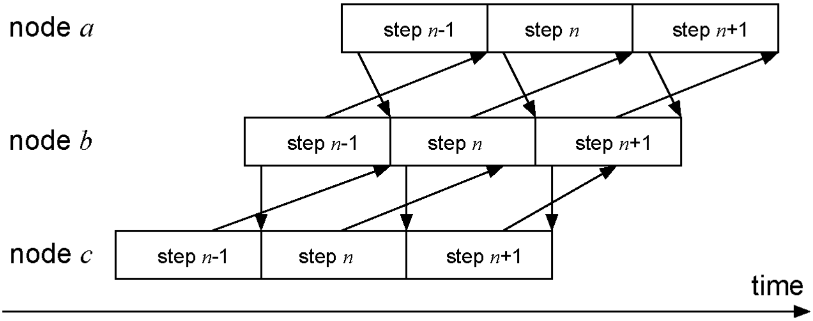


Figure 6.11: *Synchronisation method. Arrows denote data exchanges. Node a can only complete step n once the data from node b has arrived. b can only send that data during time-step n, so the difference in simulation time between a and b cannot become more than one time-step.*

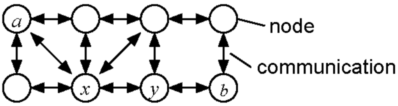


Figure 6.12: *The maximum difference in simulation time in this eight node parallel system is three time-steps between nodes a and b, as there is at most a difference of one between each of the pairs* (a, x), (x, y) *and* (y, b).

6.5.3. Using SMP systems

The hardware we use consists of 12 SMP boards, each containing two processors (cf. section 5.3). The parallel computer that is formed by connecting these systems with a high speed communication network does not provide a homogeneous communication infrastructure. The bandwidth, latency, and communication method for two processors depends on whether they reside on the same board or not. This can be handled in several ways; for example, we can have the communication API handle the differences and treat all processors equally, getting a parallel computer with 24 nodes, or we can run two threads on each board and treat it as just one node of the parallel computer.

6.5.3.1. Homogeneous programming model

If all processors are treated equally, the computer is simple to program, because only one method of communication is used. Any two processors exchange data through the communication layer with one set of functions, whether the processors reside on the same board or not. With respect to performance, this approach has several drawbacks that are described in more detail below:

- The communication library needs additional complexity to decide whether off-board or on-board communication is used.
- Small sub-models are needed, resulting in decreased overall performance.
- The high bandwidth between the two processors on a board is not used.

Communication library The processes running on a node treat all communication in the same way. The difference between on-board and off-board communication is shielded from the implementation by the communication library. This adds complexity to the library, but with only marginal loss of performance. The bigger problem is the sharing of network resources. In a zero-copy environment that avoids context switches, the communication library runs in user space. If there is only one NIC per system, it is shared by the processors, and access to its resources has to be controlled. The problem of sharing resources between processors persists for non-homogeneous methods, but if we attack it at a higher level, we can use additional knowledge about the usage of network resources.

Sub-model size The FE problem is parallelised by cutting the model into sub-models, one per node of the parallel processor. The communication requirements are proportional to the surface area of the sub-models. If twice the number of sub-models is used, the distribution will be different and more off-board communication will be needed. Figure 5.3, if compared to figure 5.2, shows how smaller sub-models have a negative impact on performance.

Bandwidth usage If all communication channels are treated the same way, the differences in bandwidth are ignored. On the workstations we use, the memory bandwidth is 2.6 GByte/s and memory latencies are less than a microsecond. The network has a bandwidth of up to 120 MByte/s, and latencies of a few microseconds. The performance of memory is thus higher by more than an order of magnitude. Both the memory and the NIC are shared by the two processors, but the arbitration for memory access is controlled by hardware, and cache coherence protocols can bring the bandwidth between the two processors to more than half of the memory bandwidth. When accessing the NIC, the processors have to compete for bandwidth, and the arbitration executed in software takes additional time.

6.5.3.2. Multi-threaded programming model

We can treat each SMP board as one node of the parallel computer. In order to take optimal advantage of all processors, we have to make sure that the workload of a node is well distributed between its two processors. We can run either a process or a thread per node. Processes run in their own memory space, independent of each other, and they require special means, such as explicitly allocated shared memory, to exchange data with each other. Threads use the same memory space, with only the stack being local to a thread. If one thread modifies a global variable, all of the others see the modification immediately. We can depend on the operating system to assign the threads or processes to run on different processors.

I chose to use threads since they automatically share data. To make sure different threads do not accidentally try to modify data at the same time, we have to keep track of the shared resources. All memory locations that are modified by one thread and read by another are shared resources, and so is the NIC. They must be managed in order to avoid inconsistent data. The shared resources are:

- the forces in each node that are accumulated during a time-step,
- the positions of the nodes that are updated when all forces are known,
- the counter that is incremented every time-step, and
- the communication network.

The communication network is a special case, as the low level implementation determines whether a call must be treated as a shared resource. Standard communication APIs are generally not thread-safe, so each call to send or receive a message has to be protected from other simultaneous accesses. This protection will cause a loss of time when multiple threads try to access the communication interface concurrently.

The communication library designed for the FE calculation has the advantage that all local channels and the global channel operate almost independently of each other. Multiple threads can use different outgoing channels simultaneously. The channels are processed by the same NIC, but they use different memory spaces within the card. The outgoing packets are serialised by the processor on the network card. Because an independent global channel is provided to every thread, global communication is no shared resource that has to be controlled. Since receiving data is completely free from interaction with the NIC in that the receiver just reads from memory, it is automatically thread safe.

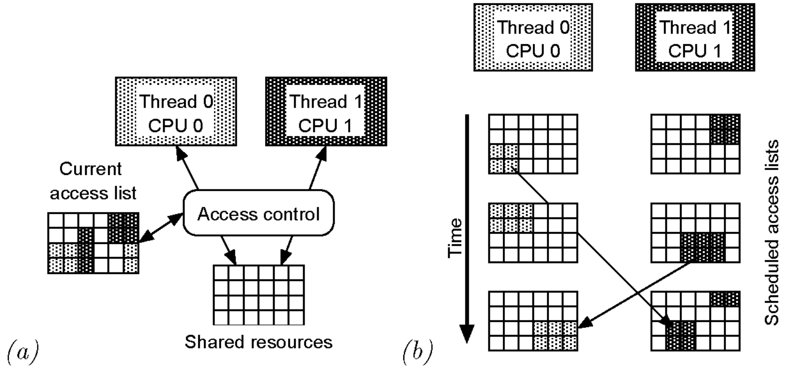


Figure 6.13: *Methods to control access to shared resources.*
(a): *An access control method keeps track of what shared resource is assigned to which thread and delays accesses by other threads to an assigned resource until it is freed.*
(b): *Shared resources are assigned to a thread in fixed order. If a resource changes owners, as marked by arrows, the new owner must wait until the current owner frees the resource.*

Shared resources are managed by assigning them to a thread until it no longer needs them and then passing them on to some other thread (fig. 6.13b). When the schedule file for a node of the parallel computer is written, each thread is assigned a sequence of batches to be calculated in every time-step. If a batch a needs data that is computed by a batch b in a different thread, a *depends* on b. A batch can only access the desired shared resources once the batches it depends on are completed.

When a batch starts running, it first performs calculations that do not write to shared resources, namely the time consuming recovery of the Elements, and thereby delays the need to access shared resources as long as possible. The batch then waits for the batches it depends on. Each batch has a number that notes the time-step in which it was last completely finished; so the completion of a batch can be checked by examining this number.

Batches are scheduled in a way that minimises the waiting time for dependencies; the shared resources will normally be freed long before their availability is checked. After the dependency check, a batch is free to modify the shared resources: it may add the recovered forces to the forces vector, calculate the updated positions of the nodes, and send data over local channels. When all remaining work of the batch is completed, the number of the current time-step is written into a field, marking this batch as finished and thus freeing the shared resources. By using the knowledge of the model during the partition stage, we can determine a strict order of access and avoid the need for arbitration of simultaneous accesses.

6.6. Scheduling

During a time-step, each node of the parallel computer recovers the Elements assigned to it and exchanges data with other processors. As these tasks remain the same in every time-step, the order in which Elements are recovered and data is exchanged is given in a fixed schedule. The main task of scheduling communication and computation is to hide the communication latency. This is illustrated in figure 6.14.

6.6.1. The schedule files

The list of tasks that must be processed in each time-step is written by the partitioning program to a schedule file. This file contains a sequence of commands that is parsed when the FE simulation is started. The commands are grouped in batches. A batch consists of five parts, all of them optional, as illustrated in figure 6.15: A processor first has

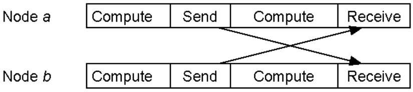


Figure 6.14: *The aim of scheduling: concurrent communication and computation.*

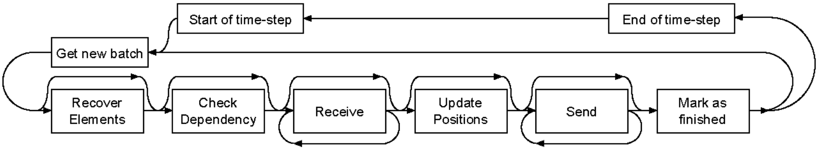


Figure 6.15: *The time-step loop of each thread*

to *recover* Elements, to get the force information required for subsequent steps. If any of the shared resources needed by that batch are currently assigned to another thread, it waits until these *dependencies* are resolved. It then *receives* data from its peers and adds received forces to the accumulated force vector. Once all forces of a node are accumulated, the new *positions* can be determined, along with the initial forces needed for the next time-step. Data is then *sent* to other processors.

6.6.2. Scheduling for local communication

Scheduling has to take the difference in performance of global and local communication into account. If we use primarily local channels, as described in section 3.7.1, the need for routing through processors can be avoided by summing up continuously and sending intermediate resulting forces. The schedule for local communication tries to optimally place data transfers and data forwarding within a time-step, in order to use the communication channels evenly during a time-step, and to lose only a minimal amount of time when waiting to receive data.

6.6.2.1. Using local channels sequentially

The first approach of scheduling is designed for a system where the NIC handles the local channels one after the other, i.e. sequentially, not concurrently. As a result, the communication on the local channels is also scheduled sequentially, which gives the basis for scheduling the Elements.

In a time-step, first all data exchanges in both x-directions are completed, then the data exchanges in the y-directions, before communication in the z-directions is started. To obtain all the data that must be sent to the neighbour in the positive x direction (X+), all Elements at the X+ border of the sub-model must be recovered. The set of these Elements is $\mathbb{E}_p^{\mathrm{X}+}$, where

$$\begin{aligned} \mathbb{K}_p^{\mathrm{X}+} = \{ k \in \mathbb{K}_p \mid D_x(\mathbb{P}_k) = 1 \\ \wedge (\forall p_k \in \mathbb{P}_k : c_x(p) \leq c_x(p_k)) \} \end{aligned} \tag{6.1}$$

$$\mathbb{E}_p^{\mathrm{X}+} = \{ e \in \mathbb{E}_p \mid \mathbb{K}_p^{\mathrm{X}+} \cap \mathbb{K}_e \neq \emptyset \} \tag{6.2}$$

After the Elements in $\mathbb{E}_p^{\mathrm{X}+}$ are calculated, the force of the nodes in $\mathbb{K}_p^{\mathrm{X}+}$ are sent in positive x direction. Next, the negative x direction X$-$ is processed in a similar way, with the difference that the pool of Elements is reduced by those already recovered in this time-step:

$$\begin{aligned} \mathbb{K}_p^{\mathrm{X}-} = \{ k \in \mathbb{K}_p \mid D_x(\mathbb{P}_k) = 1 \\ \wedge (\forall p_k \in \mathbb{P}_k : c_x(p) \geq c_x(p_k)) \} \end{aligned} \tag{6.3}$$

$$\mathbb{E}_{\mathrm{pool}} = \mathbb{E}_p \setminus \mathbb{E}_p^{\mathrm{X}+} \tag{6.4}$$

$$\mathbb{E}_p^{\mathrm{X}-} = \{ e \in \mathbb{E}_{\mathrm{pool}} \mid \mathbb{K}_p^{\mathrm{X}-} \cap \mathbb{K}_e \neq \emptyset \} \tag{6.5}$$

The y border is processed next, starting with the recovery of the Elements in $\mathbb{E}_p^{\mathrm{Y}+}$ and $\mathbb{E}_p^{\mathrm{Y}-}$. The calculation of these sets is similar to that of $\mathbb{E}_p^{\mathrm{X}-}$ in equations (6.3) – (6.5). Before the communication in the y directions can start, all values from neighbours in the x directions must be received. With the received data, the partial forces for nodes in $\mathbb{K}_p^{XY}$ and $\mathbb{K}_p^{XYZ}$, that is the nodes at the corners and edges of the sub-model, can be computed and included in the Y+ and Y$-$ communication as described in section 3.7.1.

After the last data is sent in the z directions, the remaining Elements are computed. Then, the data coming from neighbours in the z directions is received, and the complete forces and the new positions of the nodes are calculated. This leads to algorithm 2, that is executed

```
while the simulator is running
    initialise force vector
    determine and add collision forces
    recover Elements in $\mathbb{E}_p^{X+}$
    send forces for the nodes in $\mathbb{K}_p^{X+}$ in X+ direction
    recover Elements in $\mathbb{E}_p^{X-}$
    send forces for the nodes in $\mathbb{K}_p^{X-}$ in X− direction
    recover Elements on $Y$ border $\left(e \in \mathbb{E}_p^{Y+} \cup \mathbb{E}_p^{Y-}\right)$
    receive data from both $x$ directions
    add received forces to force vector
    send forces for the nodes in $\mathbb{K}_p^{Y+}$ and $\mathbb{K}_p^{Y-}$
    recover Elements on $Z$ border $\left(e \in \mathbb{E}_p^{Z+} \cup \mathbb{E}_p^{Z-}\right)$
    receive data from both $y$ direction
    add received forces to force vector
    send forces for the nodes in $\mathbb{K}_p^{Z+}$ and $\mathbb{K}_p^{Z-}$
    recover remaining Elements $\left(E \in \mathbb{E}_p \setminus \bigcup_{dir} \mathbb{E}_p^{dir}\right)$
    receive data from both $z$ directions
    compute resulting forces
    compute new positions
```

Algorithm 2: *Algorithm for local communication usage*

during every time-step. All the sets in this section are computed in advance and stored in the schedule files.

The described method has the disadvantage that the communication is not evenly distributed during a time-step. Consider 64 Elements, a cube of 4 × 4 × 4 Elements, on a processor with six neighbours. Of the 125 nodes, 50 are on the x border. 32 Elements must be recovered to make all data for x communication available. 16 more Elements are needed for communication in the y directions. Of the remaining ones, eight Elements are needed for z communication. Only the last eight Elements do not contribute to any node on the surface of the cube. At the beginning of a time-step, communication is idle, while in the last quarter of the time-step, two thirds of the data must be transferred.

6.6.2.2. Using local channels concurrently

We can schedule with a much finer granularity to get a more evenly distributed usage of local channels. The nodes are given a priority according to the number of times their data is sent over the network. The nodes in $\mathbb{K}_p^{XYZ}$ get the highest priority as they have to be sent three times, while nodes in $\mathbb{K}_p^{Y}$ are only transferred once and therefore get a lower priority. The nodes in a set are put into a batch and scheduled according to their priority. Each Element e is then assigned to the first batch that contains a node in $\mathbb{K}_e$. Intermediate processors have to receive, add, and send the partial forces of the nodes that are sent more than once. This can either be scheduled between the processing of two batches, or those nodes can be attached to a batch with lower priority after their forces were sent once.

With this method, the load on the communication network is more evenly spread over a time-step than with the method described in the last section.

6.6.3. Scheduling for general communication

As shown in section 3.7.2, the communication load can be reduced if global channels are used for data that is needed on more than two processors. First, one of the schemes illustrated in figure 3.5 is chosen. As in the last section, the nodes are assigned a priority according to the number of times their data must be transferred during a time-step. Nodes that are sent to the same destinations are grouped together in batches. Batches of the same priority are arranged to get an optimal usage of communication hardware. Each Element is assigned to the batch that needs the results of the Element's recovery first.

6.7. The partition tool

The partition tool uses the description of the FE model, a distribution of Elements to processors, and the topology of the parallel computer to generate a schedule file for every processor. When determining a schedule, the program tries to spread the communication evenly over the time-step, with sends scheduled early and receives scheduled late. If the schedule files are to be generated for multiple threads, the partition program also minimises the expected time the threads spend waiting for shared resources.

The partition tool starts by reading the FE model given in a standard FE file format and an additional file that provides the distribution of Elements to processors (fig. 6.16). This distribution file can be written by hand or generated by an optimising algorithm as described in

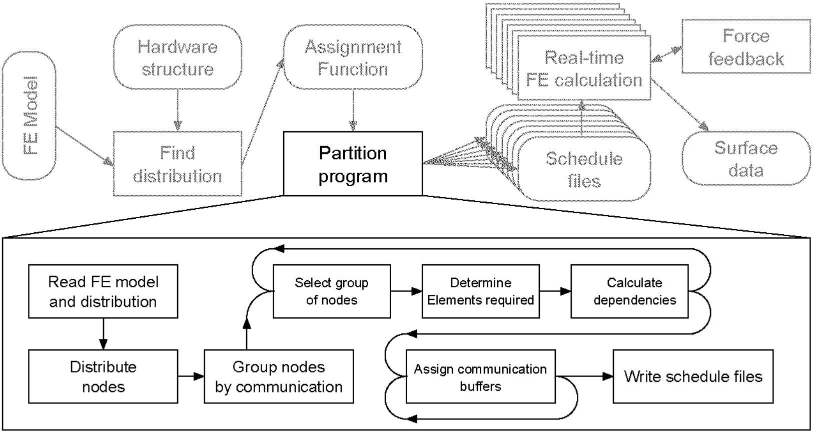


Figure 6.16: *Control flow of the partition tool*

section 6.8.3. The information of the FE model file is used to determine the relations between Elements and nodes. The sets described in section 3.5 are constructed, namely $\mathbb{K}_p$, $\mathbb{E}_p$, $\mathbb{E}_k$, and $\mathbb{P}_k$ for all $p \in \mathbb{P}$ and $k \in \mathbb{K}$. For each processor, a sequence of batches, containing Elements to be recovered and data to be sent and received, is then found. For a processor p, the program divides the nodes into different priority groups depending on the number of times they have to be forwarded. A node that will be forwarded by another processor gets a higher priority than a node that is just needed by one neighbour. Within a priority group $\mathbb{K}_p^i$, sets of nodes $\mathbb{K}_{p,p_j}^i$ are created that have to be sent to the same communication peer p_j:

$$\mathbb{K}_{p,p_j}^i = \mathbb{K}_p^i \cap \mathbb{K}_{p_j} \tag{6.6}$$

The sets $\mathbb{K}_{p,p_j}^i$ build the basis for the batches. In each priority group, a sequence of batches is determined. If a schedule for multi-threaded computation is needed, the main criterion for finding a sequence of batches is the usage of shared resources; the expected time that passes between accesses to a shared resource by different threads must not exceed a certain minimum. A second criterion is the load balancing; all threads should be assigned the same number of Elements. Out of the $n!\,2^{n-1}$ possible ways to assign n batches to two threads, the program finds the best solution with an exhaustive search. As the number of pos-

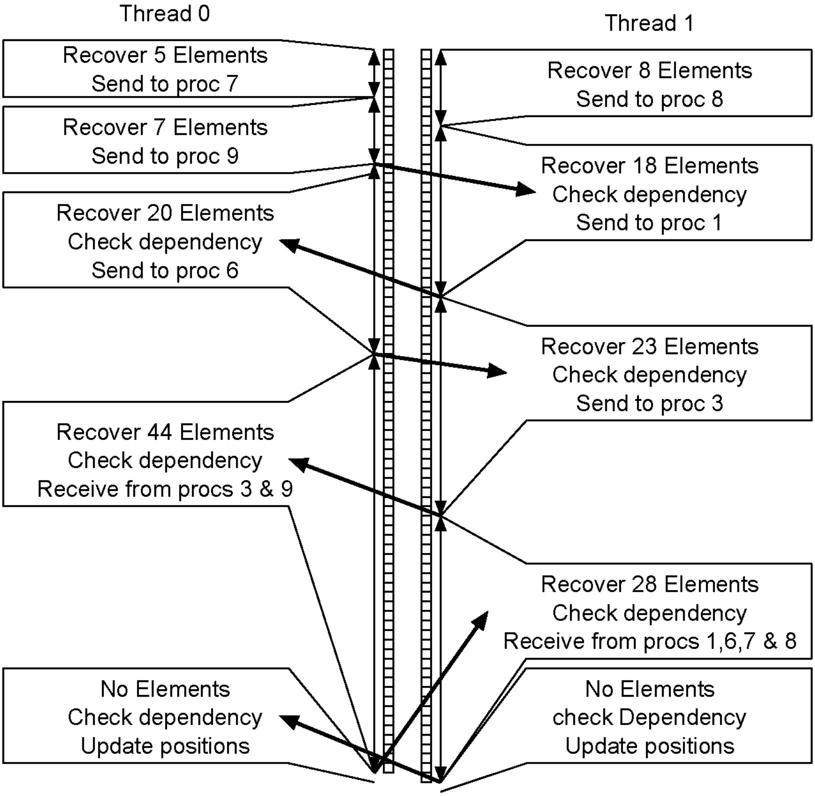


Figure 6.17: *An example schedule for two threads on one node of the parallel computer. Fat arrows show dependencies. A batch can only be finished once the batch it depends on is finished. The two columns in the centre show the Elements recovered in each batch. This recovery is the most time consuming part, therefore the number of Elements in a batch is taken as an estimate of the time a batch needs.*

sibilities grows very quickly for larger n, this search can be terminated after finding a sequence that is good enough.

At this point, the sequence of send operations is known. The program adds to each batch all the Elements and receive commands that are needed to collect the data for the batch's send operation. Access to shared resources is tracked, and dependency information is added to the batches. Then the remaining Elements and receive commands are distributed evenly among the threads, so all threads recover the same number of Elements during a time-step. A last batch is added to each thread, containing commands to calculate the new positions of the nodes and the initial forces needed for the next time-step. An example schedule is seen in figure 6.17.

When the sequence of all data transfers is known, the buffers for global communication can be assigned. For each receive command, an offset in the receive buffer is calculated where the received data is to be found. The send commands get the same offset so the data will be written to the correct address.

In a final step, the information is written to the schedule files.

6.8. Distributing Elements

The first FE model we used contained 21 Elements. We distributed it by hand to four processors. The distribution of a larger model to more processors is not so simple. With optimal load balancing, there are more than 10^{1960} ways to assign the 1840 Elements of our model to 12 parallel nodes. A method must be found to get a good distribution. We regard an assignment function $a{:}p \mapsto \mathbb{E}_p$ as good if:

- the amount of data that is transfered is minimised, and
- the load is optimally balanced.

Partitioning the mode consisting of 1840 Element by hand is not feasible for several reasons:

- The model is three dimensional and thus difficult to display on two dimensional media. Special tools would have to be used to make the results of a partition visible to a human.
- A division into eight or more sub-models requires cuts in three dimensions in order to minimise communication. These divisions are hard to visualise.

- It is difficult for a human to see whether one partition is better than another.
- A lot of work is required to find one distribution. If a different anatomy is used or the number of processors changes, the work must be done again.

In order to allow a human to find an effective assignment function, a lot of effort would have to be put into tools to help him. This effort is better spent writing a program that finds an optimised solution.

6.8.1. Optimising criteria

To optimise a problem with a computer, one must be able to determine the value of a possible solution. The qualitative criteria were described above, but a computer needs a quantitative measurement of how well the criteria are met.

Load balancing A parallel computer can only be as fast as its slowest node. Load balancing tries to equalise the time the processors spend in a time-step. Many tasks contribute to that time, some of them an unpredictable amount. However, more than half of the time-step is spent recovering the Elements, a time that is proportional to the number of Elements recovered. Moreover, the time spent with other tasks, such as collision detection or communication, is also roughly proportional to the number of Elements. Thus, we can take the number of Elements assigned to a processor as a measurement of load and balance it by assigning an equal number of Elements to every processor.

The number of Elements is not necessarily divisible by the number of processors, some will have more Elements, some less. The node that is assigned the largest number of Elements determines the speed of the parallel computation. We conclude that a distribution has good load balancing if no processor is assigned more Elements than necessary:

$$\forall p : |\mathbb{E}_p| \leq \left\lceil \frac{|\mathbb{E}|}{|\mathbb{P}|} \right\rceil \tag{6.7}$$

A distribution should be rejected unless it meets this load balancing condition.

Communication The bandwidth a distribution requires is equal to the sum of the bandwidths of the nodes. The bandwidth of a node k

is determined by the number of processors $|\mathbb{P}_k|$ that process the node. A node that resides on only one processor causes no data transfer and needs no bandwidth. If three or more processors are involved, the node gets an additional bandwidth penalty. I computed the bandwidth a processor p needs as:

$$n^p_{comm} = \sum_{k \in \mathbb{K}_p} f(|\mathbb{P}_k|) \begin{cases} f(1) = 0 \\ f(2) = 4 \\ f(n) = 4 + n \quad (n > 2) \end{cases} \tag{6.8}$$

This function gives a number that can be used to estimate the communication requirements of a distribution. Lower numbers are better. As processors exchange data in parallel, only the processor with the highest bandwidth requirements determines the communication requirements of a distribution.

6.8.2. Obtaining a distribution

We need a method to derive a reasonable distribution from the FE model that can be used as a starting point for optimisation. The distribution must meet the load balancing criterion of eq. 6.7 and should have low communication cost. Assigning Elements connected by nodes to different processors incurs a communication cost. Because connections between Elements and their geometric distance is closely related, we can derive a distribution from the geometric positions of the Elements.

For each Element, we calculate the arithmetic mean of the positions of its corners, thereby obtaining the position of the Element's centre (fig. 6.18(a)).

$$pos_{el} = \frac{\sum_k pos_k}{n_{\text{nodes}}} \tag{6.9}$$

We can now group Elements that are close together (Algorithm 3). This is done by making cuts in separate coordinate directions, as shown in figure 6.18. For an $n_x \times n_y \times n_z$ mesh of processors, we first sort all the Elements by their z coordinates. Then they are divided into n_z equal blocks (fig. 6.18(d)). Each of these blocks is sorted by their y coordinates and subdivided by n_y (fig. 6.18(e)). The procedure is then repeated for the x direction (fig. 6.18(f)). As a result, we obtain a distribution with an equal number of Elements per processor and with low communication cost.

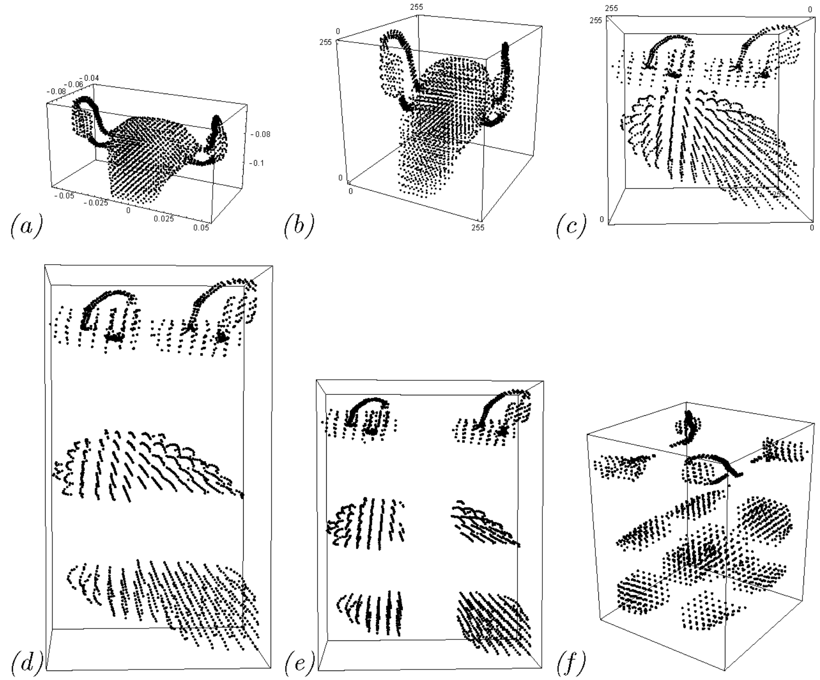


Figure 6.18: *The steps from the geometric positions of the Elements to a distribution*
(a): *The centres of all Elements*
(b): *The logical positions obtained from* (a) *by normalising all coordinates.*
(c): *The logical positions from* (b) *modified with additional knowledge to obtain a better distribution.*
(d): *The Elements are sorted with respect to their logic coordinate in* z *direction and divided into three groups.*
(e): *Each of the three groups is divided along the* y *coordinate.*
(f): *After another division in* x *direction, we get twelve sub-models that are assigned to the twelve nodes of the parallel computer.*

By normalising the coordinates, we can work within known ranges (fig. 6.18(b)). We can also apply additional knowledge about the form of the FE model to obtain a better distribution (fig. 6.18(c)).

```
sort $z$ coordinates of all Elements $\mathbb{E}$
cut into $n_z$ sets $(\mathbb{E}^z_1, \ldots, \mathbb{E}^z_{n_z})$ of at most $\lceil n_{e\ell}/n_z \rceil$ Elements
for all sets $\mathbb{E}^z_i$
    sort $y$ coordinates of $\mathbb{E}^z_i$
    cut into $n_y$ sets $(\mathbb{E}^y_1, \ldots, \mathbb{E}^y_{n_y})$ of at most $\lceil |\mathbb{E}^z_i|/n_y \rceil$ Elements
    for all sets $\mathbb{E}^y_j$
        cut into $n_x$ sets $(\mathbb{E}^x_1, \ldots, \mathbb{E}^x_{n_x})$ of at most $\lceil |\mathbb{E}^y_j|/n_x \rceil$ Elements
        assign these sets to the processors with $(x, y, z)$ coordinates of $(1, j, i)$ to $(n_x, j, i)$
```

Algorithm 3: *Obtaining a distribution from positions*

6.8.3. Optimisation

To optimise a problem with a computer, we need an evaluation function, an optimisable representation of the solution, and an optimiser. The evaluation function is a method to compare possible solutions, either by giving an absolute value representing the solution's worth or by finding the better of two solutions. A good representation can always be transformed to a possible solution despite any modifications of the representation's values.

For us, possible solutions are those that distribute an equal number of Elements to every processor, such that each Element is assigned exactly once, fulfilling eqs. (3.14) and (6.7). As an example of a bad representation, we keep a list of processor numbers that shows to which processor each Element is assigned. If this list is modified freely, the load balancing condition can be violated. Likewise, if each processor is assigned a list of Elements, unconstrained modification could result in Elements that are assigned exactly once.

A good representation is described in section 6.8.2. The positions of the Elements can not only be used to determine a good starting point for optimisation. Once the relationships between nodes and Elements are extracted from the FE model, the physical properties are

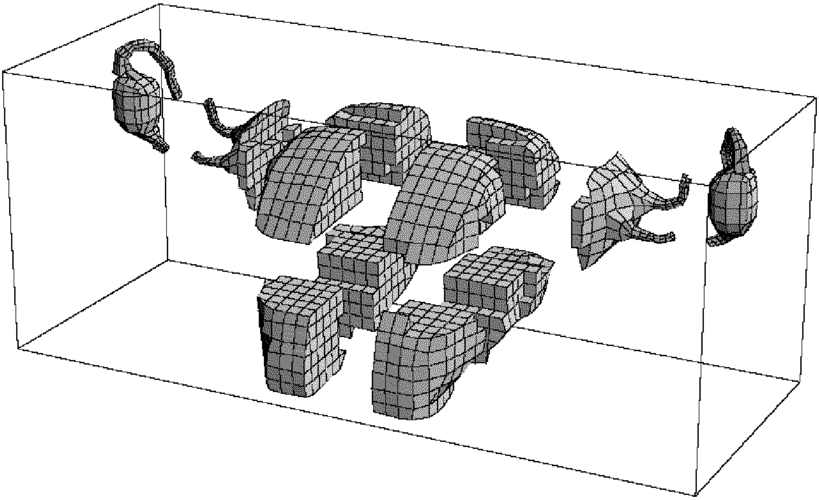


Figure 6.19: *The uterus partitioned for twelve processors*

no longer important for the optimisation. As we have seen in section 6.8.2, the positions can still be helpful in obtaining a reasonably good distribution. We can now optimise this distribution by moving around Elements. As the positions of the Elements change, we call them *logical* positions. The logical positions are derived from the physical positions (figs. 6.18(a–c)) and then modified during optimisation. No matter how the logical positions are modified, algorithm 3 will still find a distribution that fulfils the load balancing condition.

We use the bandwidth described in section 6.8.1 to compare the value of different solutions.

6.8.4. Genetic adaptation

Once we know an evaluation function and a modifiable representation, we need an optimiser. I chose to use genetic adaptation (Lawrence 1991). The logical position of an Element, consisting of three logical coordinates, forms a *gene*. The logical positions of all Elements constitute the *genome* of an *individual*.

The steps in genetic adaptation are shown in figure 6.20. Starting with a large pool of individuals, some of them are randomly modified by *mutation*. Then random pairs of individuals are selected and their genomes are mixed in a *cross-over* step, resembling the way bacteria exchange their genes. Finally, the fitness of each individual is calculated, and the fittest reproduce, replacing the weaker ones. I applied this method to determine an optimal distribution by finding the best logical position for each Element.

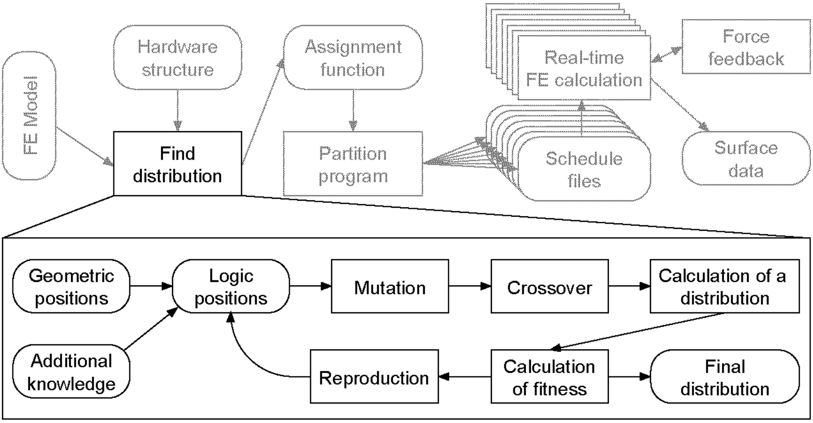


Figure 6.20: *Control flow of the genetic adaptation program*

During mutation, random coordinates are chosen. Each coordinate is modified with a random number with Gaussian distribution. The probability that a certain coordinate is modified, and the parameters of the Gaussian random number generators can be changed to modify the behaviour of the optimisation.

For cross-over, all individuals are paired. For each pair of individuals, a random cross-over point between two genes is determined. The genomes of these individuals are then cut and swapped at the cross-over point.

The method used for reproduction is called "binary tournament selection". The fitness of all individuals is calculated. Algorithm 3 is used to find a distribution from the logical positions, and equation 6.8 is used to calculate the communication cost used as the fitness function. Pairs of individuals are pitted against each other, and a copy of the fitter individual takes the place of the weaker.

With genetic adaptation, I was able to find a distribution for the FE model with 21 Elements that was better than the distribution found by hand. Applied to the model with 1840 Elements, the optimisation reduced the communication cost of the already good starting position by 18%. As no step in the generation of distribution requires interaction, I could easily generate optimised distributions for different numbers of processors.

6.9. Verification

After programming the parallel FE simulation and generating a schedule file for each processor, we are ready to start the calculation. Yet, we cannot know if there are any errors in the program or in the schedule file. When data is transferred, we cannot be sure that it is interpreted and processed correctly. As the forces sent and received are floating point numbers, they are difficult to distinguish if something goes wrong.

I therefore implemented a method to verify the correctness of the parallel FE program and the scheduling files. The additional code for the verification is kept to a minimum, because new code adds new bugs, and code that is replaced by the verification code cannot be tested this way. It was also important to add no delay to the run time system. This is accomplished by switching the verification on or off at compile-time. If the verification is switched off, no delay is added to the compiled program. At only two vital places in the time-step loop was verification code added: whenever two forces are added together, and when the new positions are calculated from the resulting forces.

The smallest entity transferred by the FE program contains either the position or the force of a node. It consists of three floating point values with a total size of 24 bytes. For verification, we use those 24 bytes for a different purpose. Instead of adding up the forces provided by the Elements, we store the global indices of the Elements that contribute to that node. We divide the vector into 12 two-byte fields, the first field counting how many of the other fields are filled, and the remaining fields listing the global indices of the Elements that contribute forces to be added to the force vector in the real calculation. This works with up to 11 Elements contributing to a node and with up to 2^{16} Elements contributing to a force in the complete model. This is more than we need.

The send routines are not modified. They take data out of the global force vector and send it, unaware that they are sending lists of Elements instead of floating point values. Normally, the local vectors and the received vectors are added when the data is received. During verification, the lists are joined instead.

When a new position is to be calculated, the list of Elements is copied to the position vector. At the end of a time-step, the list of Elements in the position vector of every node on every processor is stored to a file. A Perl program compares this list to the original FE model with which we began our work. If the correct Elements are listed for every node, we know that:

- the schedule files are correct,
- there were no communication errors,
- all forces were properly initialised, and
- all nodes had their positions updated.

By adding random delays in the computation, we can also stress proper operation of synchronisation and control of shared resources for multithreading.

6.10. The importance of optimising compilers and optimised data layouts

Near the end of the project, I decided to try and improve the execution speed of the recovery of an Element. The algorithm was already provided and well optimised with respect to the number of floating point operations. At the beginning of this task, a time of 15μs was needed to process one Element[1] on the first test system, an Alpha 21164A running at 500 MHz. The early estimations projected a time of 3μs per Element with a 500 MHz processor. I was looking for an improvement by a factor of five through more efficient processing of the same algorithms. The original program was written in C++, and Compaq had just made available a C compiler that produces vastly faster code. First, I built a measurement platform and ported the critical sections of the code to C. The original program, taking 15μs per Element, was compiled with the standard compiler (g++) and ran at a pitiful 66 MFLOPS, less than 7% of the peak performance. The first run through the improved compiler (ccc), with options set for maximal speed, cut the time down to 8μs.

---

[1] The calulation of one Element comprises the calculation of the inner forces from the current displacement and the addition of these forces to the global force vector.

Code	Runtime per Element
Original code, compiled with g++, running on 21164A @ 500 MHz	$15\mu s$
Original code, compiled with ccc, running on 21164A @ 500 MHz	$8\mu s$
Optimised code, compiled with ccc, running on 21164A @ 500 MHz	$3\mu s$
Optimised code, compiled with ccc, running on 21264A @ 666 MHz	$2\mu s$
Optimised code, compiled with g++, running on 21264A @ 666 MHz	$5.5\mu s$

Table 6.1: *Execution speeds with different compilers and the effect of optimised code*

I then started to examine the generated assembler code using a simple self-written tool to find inefficient code. The inefficient code is produced when the compiler does not optimally schedule vector operations due to possible aliasing. More information about the aliasing problem can be found in (Aho et al. 1986).

I identified passages in the C code that resulted in inefficient code and changed them by adding temporary variables and arrays and by inlining a subroutine that was only called once. In the end, a single Element was calculated in $3\mu s$ on the 500 MHz 21164A. The test on a node of the parallel computer with its 666 MHz 21264A processor yielded a time of only $2\mu s$ per Element, attaining 500 MFLOPS, more than a third of the theoretical peak performance.

The difference between the two systems is slightly bigger than the difference in main processor clock speed, because the newer processor also features a higher average number of instructions per second. The last comparison of table 6.1 shows that the improvement is mainly due to the compiler that produces efficient code as long as it does not encounter potential aliasing.

7

The Communication System

In order to keep the FE program portable, layers of abstraction are inserted between the application programs and the communication hardware. In this chapter, we examine the events started off by the FE program when it calls a function to exchange data between processors. Libraries and protocols, which we will call "communication systems", exist in many variations, each with different design goals.

A communication system ensures the correct delivery of data, correcting possible errors induced by deficiencies of the underlying network. Several things could happen to the data exchanged in a parallel computer:

- Some bits could be delivered incorrectly.
- An intermediate station (router) or the designated receiver could be unable to receive data due to a failure or too high a load.
- Data could be lost.

- Data could arrive in a different order than it was sent.
- In a multitasking system, the network resources must be protected from simultaneous access by multiple tasks.

There are ways to work around these problems. *Error detection/correction* takes care of incorrectly received bits. *Dynamic routing* is used to bypass failing or overloaded routers, and *flow control* ensures that no part drowns in data. Lost data is detected by the receiver, causing a *retransmission*. If necessary, the data is then *reordered* and delivered to the receiving program in the same order it was sent. Failure of a receiving computer can not be handled by the communication system alone. In a parallel computer, the tasks assigned to a failing node have to be reassigned to the rest of the computer, requiring *fault tolerant* parallel computing. The parallel program has to provide for this in some way, unless redundant nodes can take over. In order to handle simultaneous access to the communication system by different tasks, some sort of *access control* is required.

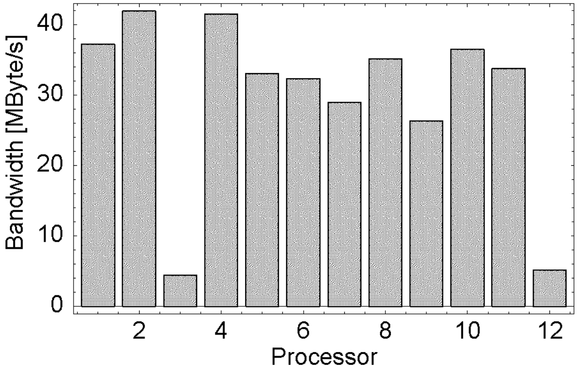


Figure 7.1: *Required communication bandwidth for each processor*

The more errors and undesired behaviour a communication system can handle, the more complex it becomes, often at the expense of performance. By analysing the needs of our system, we can determine the amount of flexibility and performance required, and thus choose the tradeoffs for a suitable communication system.

In the following sections, we will first determine how much flexibility and performance the communication system needs. A short description

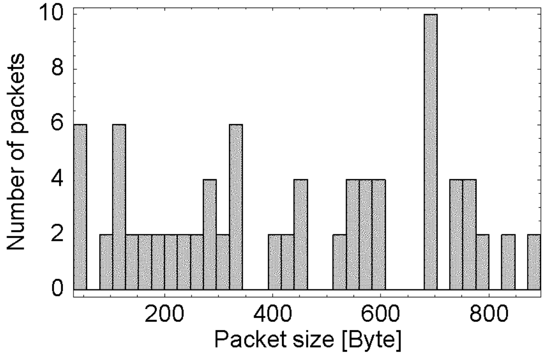


Figure 7.2: *Sizes of packets transferred in a time-step. Average size of a packet is 424 Bytes*

of different communication methods follows. The protocols that were used in the simulators are listed, along with a performance comparison. Finally, I will explain the protocol that was designed for the FE computer, along with a description of the implementation.

7.1. Communication requirements

Figure 7.1 shows the communication bandwidth required by the traffic generated by each processor, and figure 7.2 shows the sizes of the packets that are transferred during the FE simulation. The numbers were obtained from the schedule files of the distribution used in the simulation. In figure 7.1, processors number three and twelve need significantly less bandwidth than the others; those processors calculate the ovaries that can be seen to the extreme left and right in figure 6.19 on page 110. These packet sizes are rather small when compared to conventional communication systems and yield less than optimal behaviour. This is illustrated by the bandwidth measurements of figure 7.4 on page 128.

In the following paragraphs, we will examine the requirements of the FE simulation with respect to the above mentioned tasks of a communication system.

Dynamic routing and fault tolerance The LASSO system is not mission critical; we can accept an unexpected failure of the program or downtime to repair a component, as long as it only happens rarely. Because the computers and networks we use are reliable, the expected rate of failure is low enough and no redundant components are needed.

If we look a the system overview in figure 5.7 on page 79, we can see that there are no redundant resources that could replace a failing component automatically. There are two separate communication networks connecting all the nodes of the computer, but they serve different purposes. Neither could take over for the other in case of a failure. The Ethernet is not fast enough to handle the data generated by the FE computation. It is used to boot the nodes and to handle administrative requests. The Myrinet can move the necessary amount of data, but it can only be initialised once the computer is booted, and it cannot handle traffic other than FE data as long as the simulation is running. If a part fails, it must be repaired or replaced before the simulation can be restarted.

Error detection/correction The short links provide a high signal to noise ratio at the receiver. The Myrinet links use forward error correction to detect and correct simple errors. Overall, the bit error rate (BER) is lower than 10^{-15}, according to the manufacturer. We transfer less than 40 MByte/s per node, resulting in a total of about 4 Gbit/s, so an error is expected at most every 70 hours. We can choose to ignore the possibility of errors and take the risk that the simulator might have to be restarted, should an error occur. If we do implement error detection/correction at a higher network layer, it must not have a noticeable impact on performance as long as no error occurs. Once an error is detected, correcting the error can take a comparatively long time, even causing a noticeable delay, as transmission errors are rare.

Retransmission/reordering The assumption that there are no transmission errors together with the link level flow control ensure that no packets are lost. When all data between two nodes is transmitted over the same wires and no data is lost, the data arrives at the destination in order. Neither retransmission nor reordering is required.

Flow control Flow control is required to prevent the links from being overloaded and the receivers from being overflown. The link level flow control is needed if two nodes try to send data over the same link. Only one packet at a time can be routed through the output part of a switch. The other packet is stopped by the switch. The Myrinet hardware uses backpressure to control the flow of data at the link level.

When determining the requirements for end-to-end flow control in the parallel FE system, we can once more profit from the fact that the data flow is known completely before the program is started. The nodes of the parallel computer are synchronised as described in section 6.5, and we can calculate the maximum amount of data exchanged in a time-step. If the receive buffers are large enough, which they are in our program, *no explicit flow control is needed.* The communication pattern itself provides the synchronisation of, and flow control between, the nodes of the parallel computer.

Fault tolerance In the improbable event that a node of the parallel computer fails, our system will stop. If fault tolerance is built into a parallel computer, the remaining nodes are able to take over from the failing one. This can be accomplished either by activating a redundant node reserved for such an event, or by distributing the workload among the remaining nodes. If a node fails, it must be repaired or replaced. The computation could be adapted to run on fewer nodes, however, this increases the workload per node, reducing the speed of the computation. This is not acceptable since the FE simulation must be calculated in real-time. Additionally, this adaptation would need the intervention of an operator to generate a new distribution and the corresponding schedule files. In any case, the running program would fail and therefore would not benefit from any code that handles such a problem gracefully. If a redundant node was available, the simulation could simply be restarted using that node. The low probability of a failure, given the small number of components, does not justify the cost of such a redundant node. If there is a failure, it must be repaired before normal operation can be resumed.

Access control We distinguish between two types of tasks[1] running concurrently: *processes* running in their own memory spaces and *threads* running in shared memory.

Multiple *processes* might try to access the same system resources simultaneously. Those processes could execute completely different programs. We will therefore assume that the simultaneous access is not

---

[1] We will assume that these tasks are either threads or processes running under UNIX.

coordinated. Processes run in different memory spaces, so user level libraries are not affected. In the layers between user space and network hardware, the requests must be serialised, as the network card will not be able to handle simultaneous requests.

A process can consist of several *threads*, which run in the same memory space and thus share all global variables. For the sake of simplicity, we will assume that the operation of these threads, and their accesses to the communication resources, are somehow coordinated. Due to the shared memory space, user level libraries are directly affected and might not be able to handle multiple simultaneous requests. In those cases, some access control has to be implemented by the calling program or an additional intermediate layer.

We know that only one process accesses the network resources during the FE simulation, but this process consists of multiple threads.

Requirements Let us summarise how the communication system can meet the discussed requirements:

- Bandwidth of 30 MByte/s.
- Latency of a few microseconds.
- Must handle small packets efficiently.
- Requires no dynamic routing.
- Requires no additional error detection due to highly reliable network.
- Network preserves order of data, no reordering is necessary.
- Requires no fault tolerance.
- Is only accessed by one process, and must handle multiple concurrent threads of that process.
- Should use only minimal processor time.

7.2. Different communication systems

There are many different network technologies, protocols and programming interfaces used for communication between computers. We use the seven layer Open System Interconnection (OSI) reference model described in (Koelmans 1981) to see where different protocols and interfaces fit in and how they can be compared:

Name	Description	Example
Physical Layer 1	Provides the electrical, mechanical, functional and procedural characteristics required to move data bits between each end of the communication channel. Describes the actual hardware connectors, wires, patch panels and other interconnect devices. Includes parameters such as voltage levels, number of pins on a connector, baud rates, etc.	HIPPI-PH, RS-232
Data Link Layer 2	Ensures the reliability of the transmission medium by providing error checking and retransmission when required, as well as flow control, and sequencing capabilities used by the Network Layer.	Ethernet, SLIP/PPP
Network Layer 3	Describes the way data is directed from node to node, based on prevailing conditions. Provides the means to establish, maintain, and terminate the switched connections between the end users' systems. Responsible for splitting the data into packets and the efficient routing of information. Routers and Gateways are examples of equipment used by this layer.	IP part of TCP/IP
Transport Layer 4	Provides end-to-end control and information interchange with the level of reliability requested by the user. Often called the host-to-host layer. This layer insulates the higher levels from the lower levels, thereby allowing communication equipment to change without requiring modifications in upper layers and the applications using those layers. At this layer, several streams of messages can be multiplexed into a single channel.	TCP part of TCP/IP

Session Layer 5	Manages the dialogue between the two cooperating applications by providing the services needed to establish the communication, synchronise the data flow, and terminate the connection in an orderly fashion. Sometimes called the Data Flow Control layer because it is responsible for establishing connections between two applications or processes, reestablishing connections in case of failure, and maintaining data flow.	Sockets and related functions
Presentation Layer 6	Allows the application to interpret the meaning of the exchange of information. Format conversions that allow otherwise incompatible devices to communicate. (E.g. big-endian/little-endian)	SMTP/POP, HTTP, FTP, SNMP etc.
Application Layer 7	Serves the end user directly by providing the distributed information services to support the applications and manage the communications.	E-Mail, WWW

Table 7.1: *The seven OSI layers*

The requirements of our system with respect to the high level layers are quite small, as we define both the sending and the receiving side of a data transfer ourselves and therefore do not have to adhere to a standard protocol such as HTTP. Table 7.2 shows how the high level layers relate to the FE program and the LASSO system.

Layer	FE program	LASSO system
Session	Initialisation of channels at startup	Initialisation of communication
Presentation	None	Format conversions between different computers (byte order)

Table 7.2: *Requirements to the higher level layers*

The physical layer was discussed in chapter 5. The Myrinet network defines not only the physical properties, it also defines the data link layer, that is, data encoding, flow control, and error correction and detection on the link layer. In our environment consisting completely of Myrinet hardware, the Myrinet network also provides end-to-end routing. In addition, the NIC can also calculate a checksum that can be used for error checking in the implementation of the transport layer.

On the Myrinet hardware, different protocols and programming interfaces can be used. We will compare the most widely used protocols with our own protocol and library.

TCP/IP TCP/IP is the protocol at the heart of the Internet. The Internet Protocol (IP) implements the network layer of the OSI model. It is a connectionless protocol, that performs the task of routing messages or packets between nodes on a network. The Transaction Control Protocol (TCP), implements the transport layer. It is a connection-oriented protocol that ensures that data is delivered error-free, by providing flow control, error correction and acknowledgement of received data. As a unit, TCP/IP takes a message to be transmitted, breaks the information into packets, and sends the packets to the remote computer where they are re-assembled into the original message. Each packet contains a piece of the message plus an ID tag, containing the addresses of the sending and receiving computers, as well as a sequence number describing the place of the packet within the message. Since each packet has its own addressing information, it can travel independently in a network. The packets may travel on different paths and arrive out of order. They may even be damaged or lost, requiring retransmission. The sequencing information allows the receiving computer to reconstruct the original message.

For our application, the features of TCP/IP are not required. The back-end extracts the routing information from the IP packets and delivers them reliably to the target computer, in order and without dropping a packet or changing a bit. The TCP/IP stack of the receiving computer only has to check that everything went perfectly and put the original message back together. TCP/IP has the major advantage of portability. It is built into every contemporary network-enabled operating system.

MPI The Message Passing Interface (MPI) is a standard that was published in April 1994 (*MPI: a message-passing interface standard* 1994). It was the outcome of a community effort to try to define both the syntax and semantics of a library of message-passing routines that would be useful to a wide range of users and of which efficient implementations could be realised on a wide range of parallel computers. The main advantage of such a standard is portability. MPI provides vendors with a clearly defined set of routines that allows them to run existing MPI-based software.

With MPI, a pool of computers participate in executing a parallel program and each of them can exchange data with any other without explicitly opening a channel first. With regard to our problem, MPI provides efficient and reliable communication between any pair of nodes in a portable way. We have used MPI on all of our test platforms, since it allows us to simply recompile the program for new platforms and to iron out any quirks before we attack the efficiency of the communication.

Sockets Sockets are the standard method of communication between computers on the Internet. Two computers that want to exchange data both open a socket and then establish a reliable communication channel between them. To use that channel, the socket is used like a file, that is read from and written to. Sockets are designed for communication between pairs of computers that are part of a much larger system, e.g. all computers that are connected to the Internet. They only allow the exchange of data along channels that were set up prior to communication. In a parallel system where each node can talk to every other, either a central master node would have to relay the data, or channels between every pair of nodes that exchange data would have to be set up, which amounts to $\frac{n(n-1)}{2}$ channels for n nodes.

Our first implementation of a parallel framework for FE computation used sockets on the Sun workstations of the Electronics Laboratory. However, the implementation of sockets in the SGI flavour of UNIX, IRIX, proved to be so slow as to be unusable. Therefore this method, and the maximum portability it provided, was replaced by MPI.

GM The GM Message Passing System is the standard low-level interface for Myrinet networks, replacing the earlier MyriAPI. It provides a number of communication commands which can be used directly or as a base for TCP/IP or MPI implementations. GM was designed to have low latency, low CPU overhead and high bandwidth. It can provide simultaneous memory-protected user-level access to several user-level applications. It provides reliable ordered communication and can route around network faults automatically. A mapper determines the network structure and the changes to it, e.g. due to the failure of a host. We use this mapper to obtain a static network map and the routing headers for our protocol. GM is a light-weight communication layer that lacks some of the features of a complete communication protocol. It can only transfer messages between DMA-able memory, which might be quite limited, depending on the operating system. Accordingly, it does not support direct gather and scatter operations. This can be overcome by layering another protocol (such as MPI) on top of GM.

FE computer protocol The protocol we have implemented fulfils the requirements described in section 7.1. The system spans OSI layers 3 to 6, whereby all of the layers are more or less reduced. In our communication system, data is directly stored into the memory of the receiving computer, that does not need to process the data until it is required for the next computation. At that time, the schedule determines where the data can be found.

7.3. Protocols used

Figure 7.3 shows the different combinations of communication protocols and hardware that can be used with our parallel FE program. The FE program uses a communication layer that implements the simple API described in section 6.3. It interfaces different lower level protocols:

MPI Several different implementations of MPI were used in this project. On the Onyx II with its shared memory, a proprietary MPI implementation is used, while the MPI-LAM implementation (Nevin 1996) is on top of TCP/IP on Ethernet networks. The MPI-CH implementation (Gropp et al. 1996) sends data over the Myrinet with the help of the GM low level interface. All MPI implementations adhere to the same standard; the same communication layer can therefore be used for all of them.

Sockets Sockets can be used as an ubiquitous communication method. The performance of sockets is usually poorer than that of MPI, yet it remains an universal communication method for inhomogeneous systems.

GM The GM low level interface can be used directly without MPI-CH. Doing so, the portability of MPI is lost, since the GM interface is restricted to Myrinet networks. As there are no intermediate layers, the direct GM implementation is slightly faster than MPI.

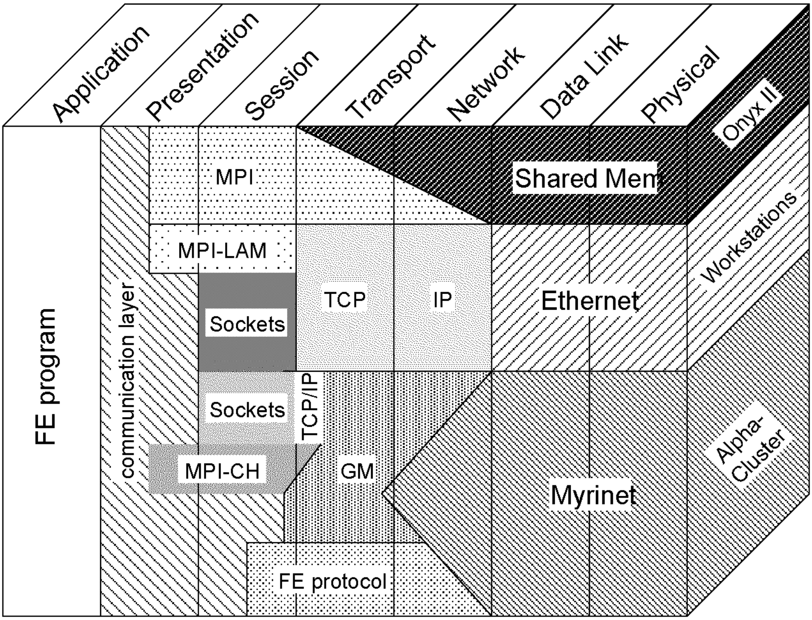


Figure 7.3: *Communication methods in relation to the OSI model*

FE protocol As the API of the communication layer was designed for the FE protocol, this implementation is the fastest, with only a light-weight layer between the application and the hardware. The communication layer consists of a minimal set of functions that allow the application to talk almost directly to the network hardware.

7.3.1. Communication speed

The figures 7.4 through 7.8 show performance measurements on the target parallel computer with three different communication systems: the FE lines show the speed of the special light-weight protocol used for the FE simulation. Of the two MPI implementations I tested, MPI-LAM uses the Fast Ethernet, and MPI-CH transfers data over the Myrinet network. All measurements were made between two processors with packet sizes that are typical for our parallel FE computation.

Figure 7.4 shows the bandwidth in half duplex mode, where one processor sends a block of data split into packets to the other processor, that starts sending the packets back after it has received all of them. Figure 7.5 shows the bandwidth in full duplex mode, with both processors sending and receiving simultaneously. Figure 7.6 shows the latency, that is the time from the start of a packet transfer until the packet has completely arrived at the receiver. The MPI-LAM latency is $80\mu s$ to several hundred microseconds and was not included in order to keep the differences between the faster protocols visible. Figures 7.7 and 7.8 show the time spent in the `Send()` and the `Receive()` calls. For the receive time, it is assumed that the data has arrived at the receiving processor before the receive command is called. Note that the FE protocol spends a constant time in the receive call, as it only checks one memory address and does no further data dependent processing.

The data in figures 7.4 through 7.8 show the comparatively high performance of the FE communication protocol running on Myrinet. The focus on small packets and the knowledge about the data flow allowed us to implement a communication layer that handles the data transfers for the FE computation significantly better than standard methods. The packet size required to reach half of the peak bandwidth is only 196 bytes, compared to 1963 bytes required with MPI. For the packets that appear in the actual FE simulation as shown in figure 7.2, the average bandwidth with MPI is 20 MByte/s, while the FE protocol reaches an average bandwidth of 111 MByte/s. This improvement

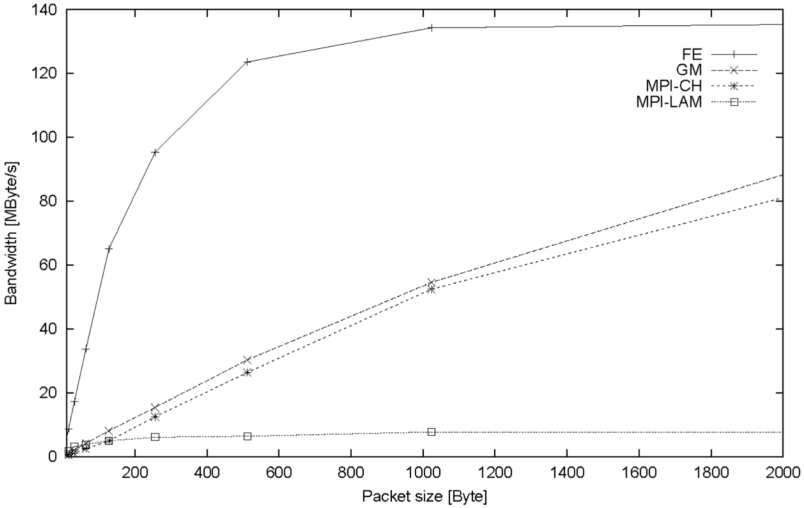


Figure 7.4: *Communication bandwidth for half duplex mode*

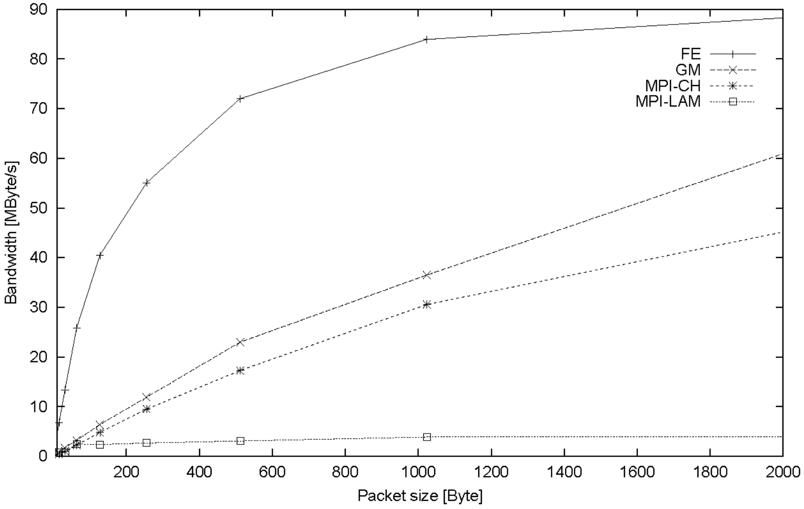


Figure 7.5: *Communication bandwidth for full duplex mode*

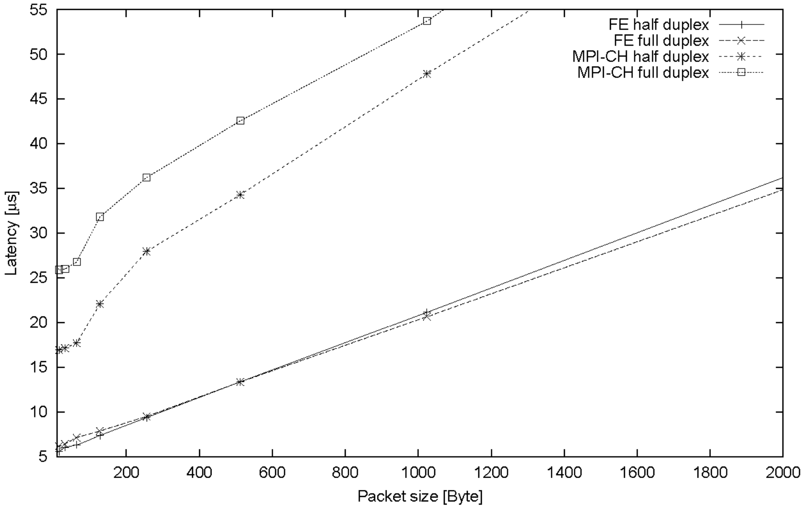


Figure 7.6: *Communication latency*

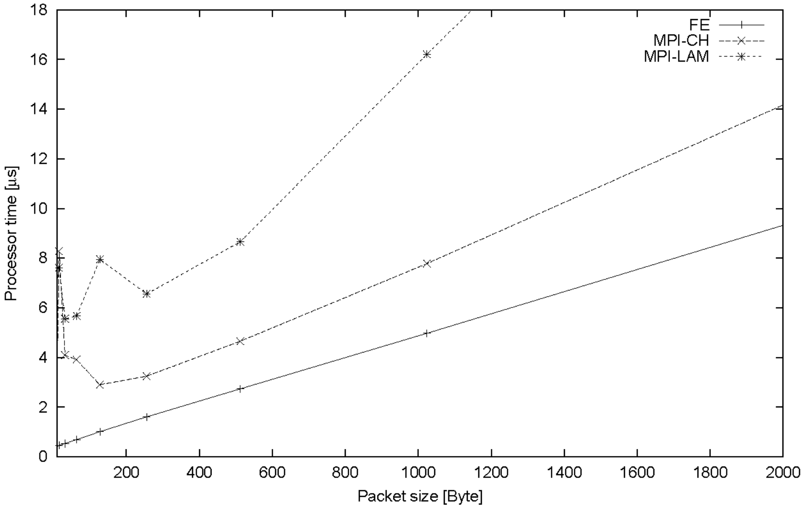


Figure 7.7: *Processor time spent in* `Send()`

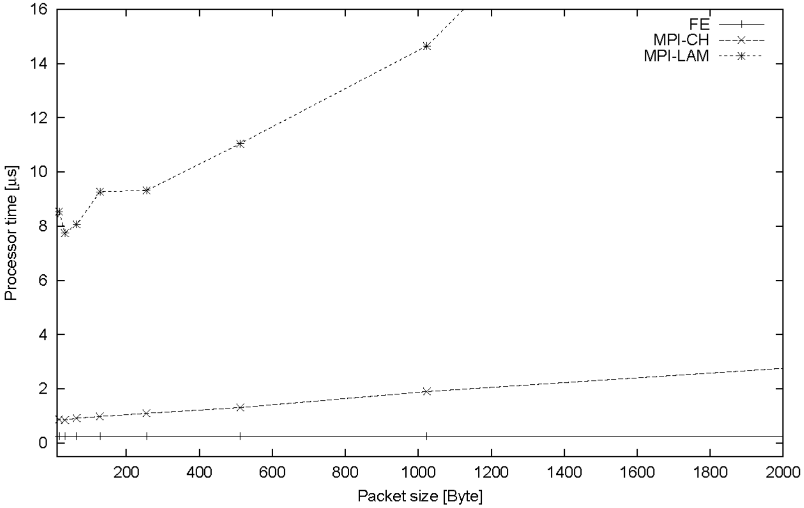


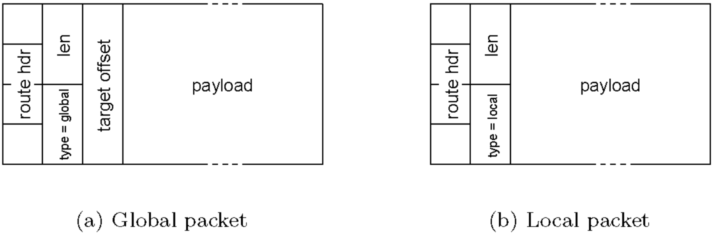
Figure 7.8: *Processor time spent in* `Receive()`

manifests itself in the overall throughput of the FE simulation that is shown in chapter 8.

7.4. FE protocol

There are two slightly different versions of the protocol, one for local and one for global communication. On the old version of the Myrinet card (PCI32), the local version was more efficient, while the new Myrinet version (PCI64A) processes global communication so quickly that everything can now be handled this way. A global packet consists of a routing header (mandated by Myrinet), the data header with two bytes indicating the type, two bytes indicating the length of the payload and four bytes for the target offset, followed by the payload (fig. 7.9(a)). The local packet consists of the routing header, the type, and a two byte length, followed by the payload (fig. 7.9(b)).

The routing header is consumed by the Myrinet switches along the way. The type is the first information that arrives at the destination NIC. The type can contain six different values for local communication, corresponding to the six possible senders or directions, plus one



(a) Global packet (b) Local packet

Figure 7.9: *Global and local packets*

value indicating global communication. If the receiver recognises a local packet, it is appended to the buffer assigned to the indicated direction. If a global packet is read from the network interface, the target address is computed by adding the given offset to the base address of the global receive space in main memory, and a DMA transfer is started accordingly in order to forward the data.

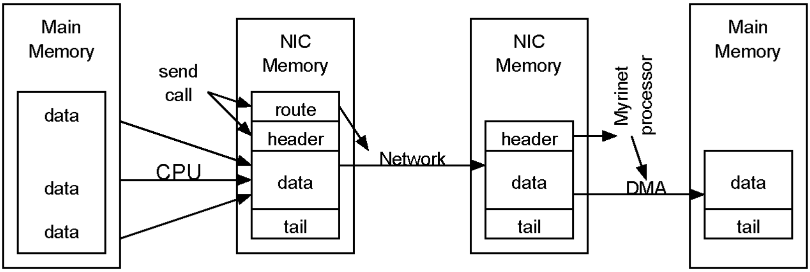


Figure 7.10: *Flow of information through the network. The main program gathers the data to be sent and writes it to the NIC. The send call adds the header and routing information. The NIC then forwards the packet to the network, which consumes the routing information. The remainder is written into the receiving NIC's memory, where the header is read to find out where the data is to be stored in the main memory of the receiver. The tail is read by the receiver to check for new data.*

As the receive buffers for local communication are refilled every time-step, an additional command is needed that resets the write pointer to the start of the buffer. The receive buffers are controlled by the sending processor, by issuing a reset command. A reset is implemented by sending a packet with length zero and no payload. If a network card receives such a packet, it resets the receive pointer of the corresponding local channel, in order to get ready to receive data for a new time-step. Whenever a processor starts a new time-step, a reset message is sent over all the local channels of that processor.

The target address for global communication is explicitly controlled by the sender, as opposed to local communication, where addresses are implicit. Therefore, no such reset method is required on global communication. The sender just reuses addresses to overwrite memory locations used in earlier time-steps.

7.5. Implementation

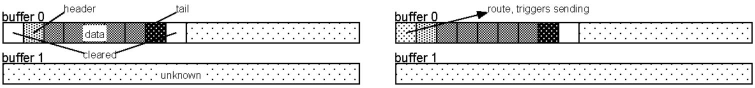
In this section, I will describe the implementation of the FE communication interface. At the heart of this interface is the Myrinet control program (MCP), that runs on the processor of the Myrinet card. The implementation described here runs on the newer Myrinet cards (PCI64A), that we use in the cluster. An earlier test implementation used network cards (PCI32) with much slower processors, which were also more complicated to program due to unavoidable bit-twiddling.

7.5.1. Sending from the host

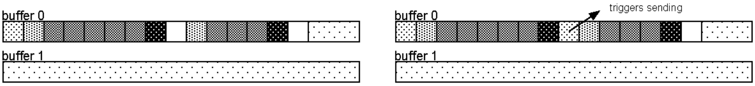
To reduce latency, the host writes data directly to the memory on the NIC, transferring 8 bytes per PIO operation. We could also use DMA to move data to the NIC, but the PIO operations reduce latency and avoid consuming the limited memory that can be used for DMA.

There are four buffers for the outgoing data in the memory of the NIC, i.e. two per thread. A thread uses one of the two buffers during a time-step and switches to the unused buffer when a new time-step begins.

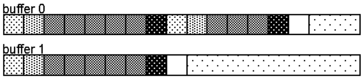
When a thread needs to send data, it writes the pre-processed data header to the second field of the current buffer and leaves the first field empty. The data is then written to the next consecutive fields and the eight bytes that immediately follow are cleared (fig. 7.11(a)). By writing the prepared routing header to the first field, the sending of the packet is triggered (fig 7.11(b)). Additional packets in the same time-step are written to consecutive fields in the NIC's memory (figs.



(a) First packet queued (b) First packet sent



(c) Second packet queued (d) Second packet sent



(e) Next time-step, first packet

Figure 7.11: *The send buffers for outgoing data on the NIC, as packets are sent.*

7.11(c), 7.11(d)). The first packet of the next time-step is written to the first fields of the formerly unused buffer (fig 7.11(e)).

During time-step n, the buffer that was used in time-step $n-2$ is reused. This could cause failure if unsent data from time-step $n-2$ was overwritten. The symmetrical nature of communication in our system makes this impossible. Figure 7.12 shows how we can reuse the same buffer without the need to check the status of send buffers with time-consuming PCI read operations.

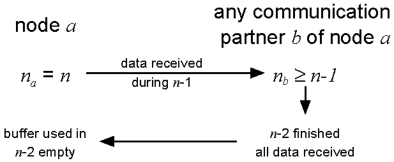


Figure 7.12: *When node a enters time-step n ($n_a = n$), we know that all communication partners have entered time-step $n-1$ and have therefore finished step $n-2$, as explained in section 6.5.2 on page 92. To finish step $n-2$, they had to receive all the data sent in that step, including the information sent by node a. It follows that all data in the buffer used during step $n-2$ was sent, and that the buffer can be reused.*

7.5.2. Main loop of the NIC program

The processor on the NIC runs a program, the MCP, that processes outgoing and incoming packets. The main loop of this program checks memory locations and status bits to determine whether a packet was received or if a packet can be sent, and the corresponding subroutine can be called (fig. 7.13). Everything runs sequentially, no interrupts are used to react to status changes.

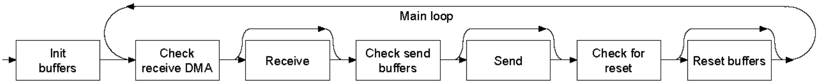


Figure 7.13: *The main loop of the MCP*

7.5.3. Sending by the NIC

The NIC constantly monitors the first field of the two active send buffers. If one of them becomes non-zero and no packet is currently being transferred, the packet in that buffer is forwarded to the network; the length field is read and the send DMA transfer is set up accordingly (fig. 7.14). As a single processor on the NIC services both host processors, the packets are serialised automatically and there is no potential for access conflicts. If several packets are ready to be sent, a non-zero header will be recognised in the next round of the main loop.

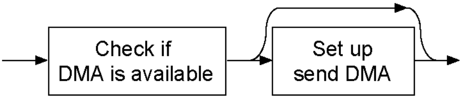


Figure 7.14: *The MCP sends data*

7.5.4. Receiving by the NIC

Whenever a packet arrives at the NIC, a DMA transfer has already been set up that writes it to the receive buffer on the card. When a packet is completely transferred, the DMA transfer is stopped and a flag signals the completion of the transfer. After recognising the active status of this flag, the program enters receive mode. In receive mode, a new DMA transfer is set up that allows the next packet to be stored directly behind the current one while it is being processed. Then the type is examined; unknown packets are skipped. If the type is correct, the target address in the main memory is calculated from the offset provided in the packet, and the corresponding information is stored in the next DMA header of the PCI DMA descriptor chain. If the PCI DMA controller is idle at that moment, it will start to copy the packet to main memory, otherwise it will automatically do so after finishing the transfer of the packets that were already on the chain (fig. 7.15).

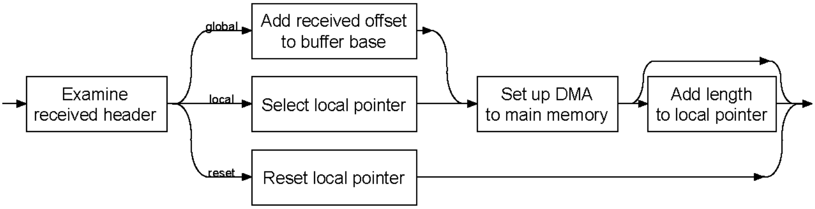


Figure 7.15: *The response to a received packet depends on its type*

7.5.5. Receiving by the host

The receiving processor does not need to do anything in order to receive data. The data is directly stored in user memory space and is accessible to the program. The schedule informs the program where to find the data at the appropriate time. As no data is changed by the processor, there is no possibility for collisions when multiple threads call the receive function simultaneously.

7.5.6. End to end coordination

When a program encounters a command to receive data, it may only process the newly received data after the complete packet was received. To ensure this, I implemented a simple end-to-end protocol; the sender appends an eight byte sequence number to every packet. This is the number of the current time-step. The receiver checks the buffer of the packet it currently needs to process and polls this location until the correct number appears.

7.5.7. Initialisation

The initialisation procedure was kept simple, and no overhead was added to normal operation. The NIC initialises the buffers, informs the host that it is ready and then starts the main loop. The nodes use the normal communication functions in order to coordinate startup. A program can be started on any subset of the cluster. On each node, a program is started that needs as parameters the total number of nodes involved, as well as its own ID. The list of cluster nodes and the route between every pair of nodes is extracted from a network map file automatically generated at compile time. The node numbered zero takes over and starts to send out initialisation packets to all other nodes. Every initialisation packet contains a magic number indicating initialisation, an address where the answer is to be sent, and the cluster-ID of node zero. The other nodes, that are supposed to participate in the simulation that is currently being started, wait for this packet and then answer with a packet containing their cluster-ID and their ID in the program that is being started. Once enough answers arrive at node zero, a complete list of participants and their names is constructed. This list is sent to the other nodes. At the end of this procedure, every node knows the address of every other node involved in the current computation.

7.5.8. Deadlocks

Deadlocks can occur when more entities try to use a resource than can be handled, and some of these entities are blocked until the resource is freed. A deadlock occurs, if this freeing of the resource depends upon one of the blocked entities to free its resource. The three types of deadlocks discussed here are the routing deadlock, where multiple packets sent simultaneously through some routers block each other, the buffer deadlock, where full receive buffers of two communication peers cause a stall to propagate back to the senders, and the dependency deadlock, where two peers wait for each other to send data.

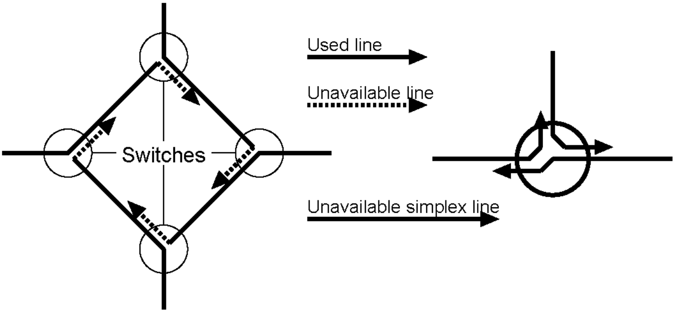


Figure 7.16: *Routing deadlock*

The routing deadlock as shown in figure 7.16 can not occur in our system as we only have a single crossbar switch and full duplex lines.

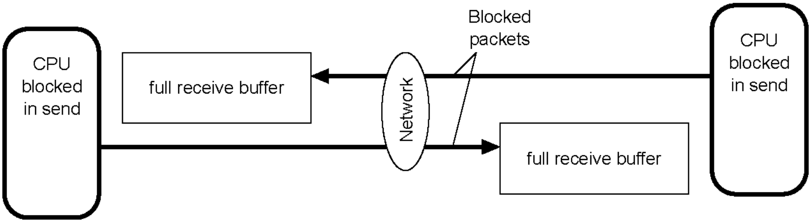


Figure 7.17: *Buffer deadlock*

The buffer deadlock shown in figure 7.17 will not happen because neither the send nor the receive functions block at any stage. The main processor and the NIC start a transaction without waiting for it to finish. This works because the packets are small and the amount of data sent never exceeds the amount of receive buffer space available.

In a dependency deadlock, both participants wait for each other to release the control over a shared resource, or both participants wait for each other to send data. This can not occur, because the order in which shared resources are assigned to threads and the order in which data is sent and received is strictly defined in the schedule files prior to the start of the program. A dependency deadlock cannot be caused or prohibited by the network components, the high level program must take care of such possibilities.

7.6. PCI speed

To make sure that the method of PIO writes to PCI memory (instead of using DMA) did not have a negative impact on overall performance, I made some measurements of PCI performance. If we take a look at the results in table 7.3, we can see that the old Myrinet card (PCI32) already showed good write performance, and the newer 64 bit card (PCI64A) almost reaches the theoretical peak of 264 MByte/s when writing 8-byte words to consecutive memory locations on the PCI cards.

	PCI32 NIC in 32/33 PCI Bus	PCI64A NIC in 64/33 PCI Bus
Write	82.8 MByte/s	241.1 MByte/s
Read	4.7 MByte/s	5.8 MByte/s
Read/Write	3.9 MByte/s	3.9 MByte/s

Table 7.3: *Measured PCI Bandwidth*

It also becomes evident that the performance for read and alternating read and write operations is poor. They should be avoided. Reading a single value appears to take more than a microsecond.

In the second measurement, we overlapped computation and writing data to the PCI card to see whether sending packets slows down the processor. As can be seen in figure 7.18, packets of up to one kilobyte can be written this way without significant impact on calculation performance. Only the transfer of the longest packets appearing in our

FE computation reduce computational performance by a noticeable amount.

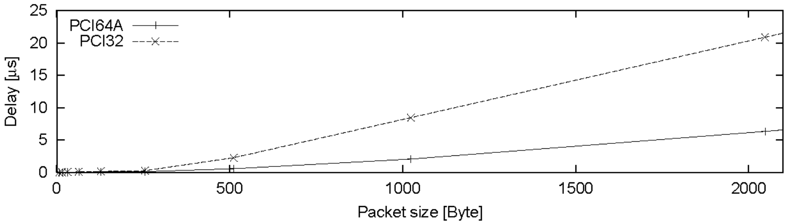


Figure 7.18: *Delay for writing to PCI bus during computation*

8

Results and Outlook

8.1. Overall performance and speedup

Figure 8.1 shows the overall throughput of different systems with a varying number of processors. The systems used were the SGI ONYX II running the standard MPI, and the Alpha cluster using three different communication systems: MPI-LAM over Fast Ethernet, MPI-GM over Myrinet and the custom FE protocol. The Alpha cluster was measured with two configurations; single threaded or with two threads per node (the graphs with the SMP attribute). The uterus FE model with 1840 Elements was used, and the time to finish a time-step was measured. Figure 8.2 shows the speedup of the parallel program, with the time measured for a single processor as reference. Figure 8.3 shows the fraction of the theoretical peak performance that was reached, demonstrating the validity of the optimistic numbers assumed in chapter 4; that a third of the peak performance could be reached.

In these measurements, the fastest node sends and receives 54 MByte/s; the total bandwidth used in that configuration is 225 MByte/s. It is obvious that MPI-LAM using the Fast-Ethernet cannot deliver that amount of data. The MPI-GM shows good performance for a small number of processors, but as the packets get smaller and

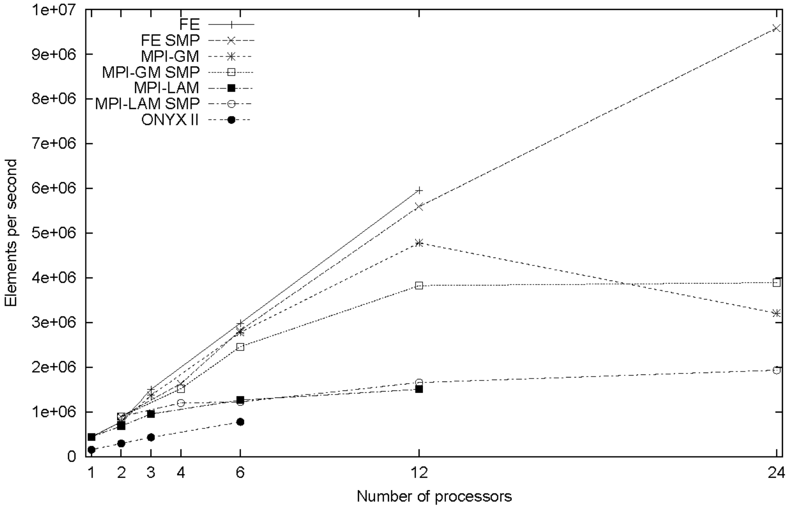


Figure 8.1: *Overall throughput*

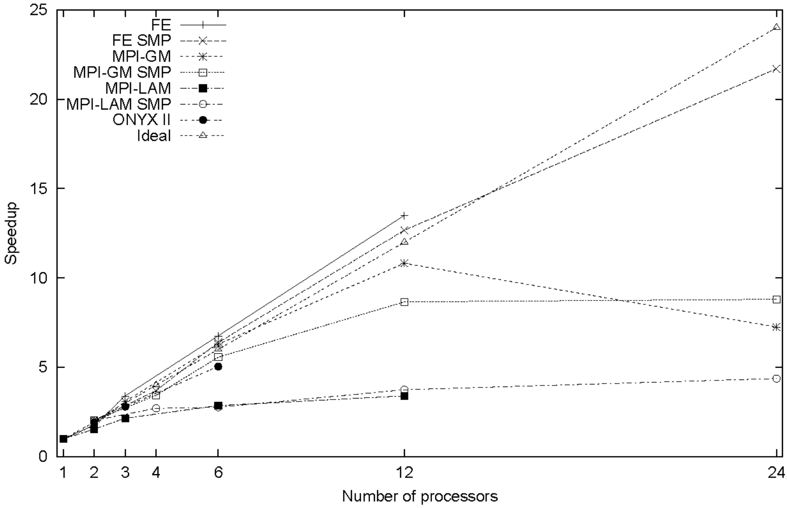


Figure 8.2: *Speedup*

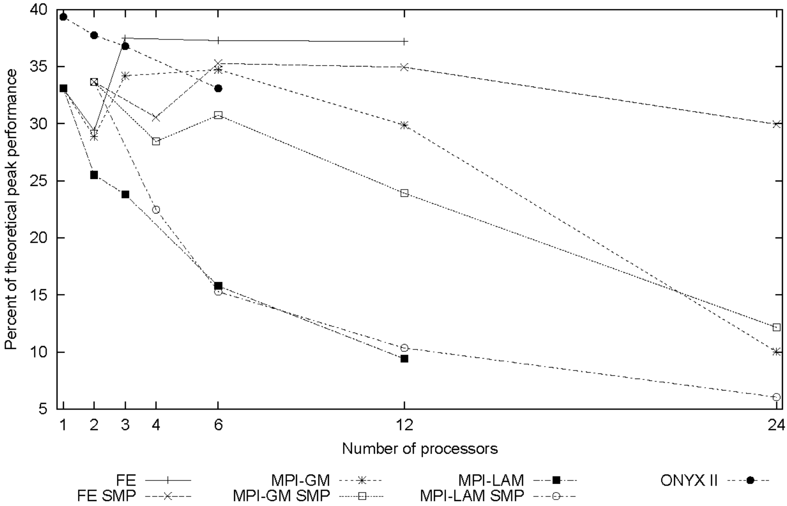


Figure 8.3: *Percentage of theoretical peak floating point performance reached*

the times allowed for delivery get shorter, the performance decreases. For 12 parallel nodes, the improvements in communication performance reached by the custom protocol manifest themselves in an improvement in overall throughput of almost 150%. The multi-threaded configurations show lower performance than the single threaded versions, due to the higher load on the network and the time lost by the synchronisation of the two threads. One measurement shows that the choice of SMP boards was correct. In figure 8.1, four measurements were taken with 24 processors. For three of those measurements, a 12 × 2 SMP configuration was used. Only the MPI-GM data point was measured in a 24×1 configuration. While the SMP configurations use 12 sub-domains, the model was divided into 24 sub-domains for the MPI-GM measurement. Consequently, the sub-domains of a parallel node are smaller and performance degrades.

Table 8.1 shows a rough price-performance comparison of different systems. Exform is a program for explicit FE calculation developed at the ETH Zürich (Berg 1997). The comparison with the eight node ONYX II, with an architecture similar to that of the Origin, shows the improvements due to the specialised FE program and the optimised parallelisation. The Cluster of workstations again shows a significant improvement over the traditional shared memory supercomputer, because the commodity components deliver higher computational performance at a lower price. This comes at the cost of slower communication and increased programming overhead. The programming overhead for the cluster is higher because no simple shared memory model can be used and the processes have to use explicit communication calls to exchange data. While the supercomputer has direct high-bandwidth channels between its processors, the cluster must exchange data through NICs connected through PCI buses. By optimising the data exchange through these network cards and by scheduling communication and computation concurrently, near optimal performance was obtained despite these drawbacks, yielding an improvement in price-performance of three or-

System	Elements / s / CHF
Origin running Exform	0.015
ONYX II with the LASSO program	2.1
Alpha Cluster	38.3

Table 8.1: *Hardware price – performance ratio of different systems*

ders of magnitude over traditional methods.

8.2. Conclusions

The goal of the LASSO project was the construction of a working simulator for diagnostic laparoscopy with studies of feasibility in associated fields. We now have a working prototype with a force feedback manipulator with which a virtual corpus uteri can be deformed. The uterus can be felt through the manipulator and the deformations can be seen on a display. The main question that was open at the start of this project could be answered:

It is feasible to use the explicit Finite Element Method for real-time interactive surgery simulation.

The goal of the FE simulator was the design of a machine able to calculate the explicit FE Method in real-time. Although I did not quite reach the original goal of a $100\mu s$ time-step, I came very close, with the fastest configuration taking just under $200\mu s$ for a time-step. This amounts to a performance of 9.6 GFLOPS at the user's fingertips. I did not expect to reach a $100\mu s$ time-step, because I did not have enough processors according to the estimates in chapter 4. The three tasks that were necessary to reach this performance were:

- The analysis of the FE program and the generated code in order to optimise the single processor performance.
- The careful parallelisation that allows performance to scale almost linearly with the number of processors, given a communication network that is fast enough.
- The implementation of a communication system that delivers the necessary performance on a cluster of workstations, which is the most cost effective hardware solution that can provide the required computational performance.

8.3. Outlook

8.3.1. Scalability

Now that we have the measurements from a working prototype, we would like to apply the knowledge gained to make estimates for future hardware implementations. How does the parallel FE program scale in *speed* and in *size*. If the problem scales in speed, adding processors proportionally reduces the time it takes to calculate a single time-step, probably reaching the desired 100 microseconds. If the problem scales in size, we can use more processors to calculate a bigger model with a constant time-step.

A useful tool to analyse scalability is the *critical sub-domain size*. We know that when the sub-domains become smaller, the requirements to the communication system increase. Communication starts to become a dominant factor and less and less can be gained by adding processors. If sub-domains have less than critical size, the computation becomes inefficient; the actual speedup misses the ideal speedup by too high a margin. If we require 90% of the ideal speedup, we can see that with the custom communication system, we can barely reach this goal with 24 processors; the critical sub-domain size is 150 Elements.

The critical sub-domain size is constant as long as the ratio between the computational speed and the performance of the communication system is constant. If computational performance increases relatively, the critical size also increases. We can now use this critical size for scalability estimations.

Scalability in speed The measurements show that the critical sub-domain size is already reached. Adding more processors decreases sub-domain size and is therefore not an effective way to increase speed. Consequently we cannot reach the 100μs time-step with more processors; all we can do is wait for faster processor and networks. Fortunately, the technological progress will close this gap. In 4Q01, a year from now, components that are fast enough should be available, making it possible to build a computer that calculates a time-step within 100μs.

Scalability in size Only the size of the sub-domain and the exchange of data with the neighbours determines the speed of the computation. The size of the total model is irrelevant. Different parts of a big model are simulated independently by different groups of processors. Thus, the calculation should scale perfectly in size, if the critical sub-domain size is observed. In the model used, critical size was reached with 12 parallel nodes, with a node having at most four neighbours. In a larger system, nodes have up to six neighbours and need to transfer more data. I expect the critical size to triple because of this effect.

When the performance of available hardware increases by a factor of six compared to the LASSO hardware, a factor that should be reached within four years, it should be possible to build a computer that simulates an arbitrarily big model with 10'000 steps per second.

8.3.2. The future of the simulator

We now have a prototype that can already be used to get a first impression of the look and feel of laparoscopic surgery. In a few years, more computational performance will be available, allowing the addition of more features. Because of the flexibility of the FE Method, complexity is almost only limited by the available computational performance. Further development of laparoscopic surgery simulators will concentrate on four main fields:

- The generation of more detailed models,
- more accurate simulation of materials,
- improved graphics, and
- simulation of different tasks during surgery, such as cutting or using multiple manipulators.

In the not too distant future, a patient will be scanned thoroughly prior to surgery. An anatomical database and prior knowledge of the patients pathologies will be used to automatically segment the three dimensional data. With the help of a tissue database, models of the organs and the connecting ligaments will be created, with accurate material properties including anisotropy, visco-elasticity, or active parts such as muscles. This model will then be used in a simulation to train the surgical team prior to the actual operation. During this simulation, the surgeon will be able to cut tissue, suture severed blood vessels, remove material, and sew the repaired organ back together. The laparoscopic simulator will help to reduce costs, simplify training, and improve the chances of success of an operation.

Glossary

Scalars

$df_{e\ell}$	Degrees of freedom of an Element
df_{eav}	Average degrees of freedom that directly influence a node
df_g	Global degrees of freedom
Δt	Size of a time-step
Δt_{max}	Maximal time-step
$n_{p_a \rightarrow p_b}$	The amount of data that is sent from processor p_a to processor p_b
n_{eav}	Average number of Elements connected to a node
$n_{e\ell}$	Number of Elements in the model
n_{int}	Integration points per Element
n_k	Number of nodes in the model
n_V^p	Number of 3D vectors that have to be sent and received by processor p

Vectors

ε	Strains in an Element
$\mathbf{f}^E$	The Forces an Element exerts on its nodes due to deformation
$\mathbf{f}_{ext}$	External forces applied to the model
f_k	Resulting force in node k
f_k^E	Contribution to f_k by Element E
σ	Stresses in an Element
$\mathbf{u}$	Global vector of displacements, the difference between current and original nodal positions
$\dot{\mathbf{u}}$	Speed of $\mathbf{u}$ $(d\mathbf{u}/dt)$
$\ddot{\mathbf{u}}$	Acceleration of $\mathbf{u}$ $(d\mathbf{u}/dt^2)$
${}^t\dot{\mathbf{u}}$	Speed at time t
${}^t\ddot{\mathbf{u}}$	Acceleration at time t

$\mathbf{u}^E$	Displacements of the nodes of one Element E
u_k	Displacement of node k
$\mathbf{x}$	Global vector of current nodal positions
$\mathbf{x_0}$	Global vector of original nodal positions

Matrices

$\mathbf{B}$	Transformation matrix
$\mathbf{C}$	Material matrix
$\mathbf{D}$	Attenuation matrix
$\mathbf{K}$	Global stiffness matrix
$\mathbf{K}^E$	Element stiffness matrix
$\mathbf{M}$	Mass matrix
$\hat{\mathbf{M}}$	Diagonal matrix composed of the attenuation and mass matrices

Sets

$\mathbb{E}$	All Finite Elements
$\mathbb{E}_k$	The Elements node k is connected to
$\mathbb{E}_p$	The Elements calculated on processor p
$\mathbb{K}$	The nodes in the FE model
$\mathbb{K}_E$	The nodes attached to Element E
$\mathbb{K}_p$	The nodes the positions of which processor p must know
$\mathbb{P}$	Set of processors in the parallel computer
$\mathbb{P}_k$	The processors that contribute to node k
$\mathbb{X}$	Possible x coordinates of processors
$\mathbb{Y}$	Possible y coordinates of processors
$\mathbb{Z}$	Possible z coordinates of processors

Functions

$D(\mathbb{P})$	Maximum distance between any pair of processors in $\mathbb{P}$
$D_x(\mathbb{P})$	Maximum distance in x between any pair of processors in $\mathbb{P}$

$D_y(\mathbb{P})$	Maximum distance in y between any pair of processors in $\mathbb{P}$
$D_z(\mathbb{P})$	Maximum distance in z between any pair of processors in $\mathbb{P}$
$a(p)$	Assigns a set of Elements to processor p
$c(p)$	Placement function which provides the coordinates of processor p in the three dimensional mesh
$c_x(p)$	Determines the x coordinate of processor p
$c_y(p)$	Provides the y coordinate of processor p
$c_z(p)$	Gives the z coordinate of processor p
$d(p_1, p_2)$	Cartesian distance between processors p_1 and p_2
$d_x(p_1, p_2)$	Distance in x between processors p_1 and p_2
$d_y(p_1, p_2)$	Distance in y between processors p_1 and p_2
$d_z(p_1, p_2)$	Distance in z between processors p_1 and p_2

Abbreviations

API	Application Programming Interface
CG	Conjugate Gradients; an iterative solver for systems of linear equations
DMA	Direct Memory Access
EBE	Element by Element
FE	Finite Element
FEM	Finite Element Method
GFLOPS	10^9 Floating point operations per second
LASSO	LAparoscopic Surgery SimulatOr
MCP	Myrinet Control Program
MFLOPS	10^6 Floating point operations per second
MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
NIC	Network Interface Card
SMP	Symmetric Multiprocessing
TFLOPS	10^{12} Floating point operations per second
VR	Virtual Reality

Bibliography

Ackerman, M. (1998), 'The Visible Human Project', *Proceedings of the IEEE* **86**(3), 504–511.

Aho, A., Sethi, R. & Ullman, J. (1986), *Compilers; Principles, Techniques, and Tools*, Addison-Wesley, chapter 10.8, pp. 648–660.

Al-Nasra, M. & Nguyen, D. (1991), 'An Algorithm for Domain Decomposition in Finite Element Analysis', *Computers & Structures* **39**(4), 277–289.

Alyassin, A. & Lorensen, W. (1998), Virtual Endoscopy Software Application on a PC, *in* 'Proc. MMVR'98', IOS Press, pp. 84–89.

Amano, H., Boku, T. & Kudoh, T. (1990), '(SM)^2-II: A Large-Scale Multiprocessor for Sparse Matrix Calculations', *IEEE Trans. on Computers* **39**(7), 889–905.

Annamalai, V., Krishnamoorthy, C. & Kamakoti, V. (1999), 'High-speed applications in the automotive industry: adaptive finite element analysis on a parallel and distributed environment', *Parallel Computing* **25**(12), 1413–1434.

Aoki, T. et al. (1997), 'The Pipette Aspiration Applied to the Local Stiffness Measurement of Soft Tissues', *Annals of Biomedical Engineering* **25**, 581–587.

Baddourah, M. & Nguyen, D. (1994), 'Parallel-Vector Computations for Geometrically Nonlinear Finite Element Analysis', *Computers & Structures* **51**(6), 785–789.

Barr, A. (1984), 'Global and Local Deformations of Solid Primitives', *Computer Graphics* **18**(3), 21–30.

Barragy, E., Carey, G. & van Geijn, R. (1994), 'Performance and Sacalbility of Finite Element Analysis for Distributed Parallel Computation', *J. of Par. and Distr. Computing* **21**, 202–212.

Basdogan, C., Ho, C.-H. & Srinivasan, A. (1999), Simulation of Tissue Cutting and Bleeding for Laparoscopic Surgery Using Auxiliary Surfaces, *in* 'Proc. MMVR'99', IOS Press, pp. 38–44.

Basdogan, C., Ho, C.-H., Srinivasan, M., Small, S. & Dawson, S. (1998), Force interactions in Laparoscopic Simulations: Haptic Rendering of Soft Tissues, *in* 'Proc. MMVR'98:', IOS Press, pp. 385–391.

Bathe, K. (1996), *Finite Element Procedures*, Prentice Hall, Englewood Cliffs, New Jersey.

Baumann, R. (1997), Haptic Interface for Virtual Reality Based Laparoscopic Surgery Training Environment, PhD thesis No. 1734, Swiss Federal Institute of Technology, Lausanne.

Baur, C., Guzzoni, D. & Georg, O. (1998), Virgy: A Virtual Reality and Force Feedback Based Endoscopy Surgery Simulator, *in* 'Proc. MMVR'98', IOS Press, pp. 110–116.

Belytschko, T. & Gilbertsen, N. (1987), Concurrent and Vectorized Mixed Time, Explicit Nonlinear Structural Dynamics Algorithms, *in* A. Noor, ed., 'Parallel computations and their impact on mechanics', The American Society of Mechanical Engineers, pp. 279–289.

Belytschko, T. & Ong, S. (1984), 'Hourglass Control in Linear and Nonlinear Problems', *Computer Methods in Applied Mechanics and Engineering* **43**, 251–276.

Berg, H. (1997), Prozeßoptimierte numberische Verfahren zur Auslegung von wirkmedienunterstützten Umformvorgängen, PhD thesis No. 12394, ETH Zürich.

Boden, N., Cohen, D., Felderman, R., Kulawik, A., Seitz, C., Seizovic, J. & Wen-King, S. (1995), 'Myrinet: a gigabit-per-second local area network', *IEEE Micro* **15**(1), 29–36.

Boux de Casson, F. & Laugier, C. (1999), Modelling the Dynamics of a Human Liver for a Minimally Invasive Surgery Simulator, *in* C. Taylor & A. Colchester, eds, 'Second International Conference on Medical Image Computing and Computer-Assisted Intervention MICCAI'99', number 1679 *in* 'Lecture Notes in Computer Science', Springer Verlag, pp. 1156–1165.

Bro-Nielsen, M. (1998), 'Finite Element Modeling in Surgery Simulation', *Proceedings of the IEEE* **86**(3), 490–503.

Bro-Nielsen, M. & Cotin, S. (1996), 'Real-time volumetric deformable models for surgery simulation using Finite Elements and condensation', *Comp. Graphics Forum* **15**(3), 57–66.

Bro-Nielsen, M., Helfrick, D., Glass, B., Zeng, X. & Connacher, H. (1998), VR Simulation of Abdominal Trauma Surgery, *in* 'Proc. MMVR'98', IOS Press, pp. 117–123.

Carey, G. & Shen, Y. (1995), 'Simulation of fluid mixing using least-squares finite elements and particle tracing', *International Journal of Numerical Methods for Heat and Fluid Flow* **5**(6), 549–573.

Carey, G., McLay, R., Bicken, G., Barth, B., Swift, S. & Ardelea, A. (1999), 'Parallel finite element solution of three-dimensional Rayleigh-Benard-Marangoni flows', *International Journal for Numerical Methods in Fluids* **31**(1), 37–52.

Carter, F., Frank, T., Davies, P., McLean, D. & Cuschieri, A. (1999), 'Biomechanical Testing of Intra-abdominal Soft Tissues', *Medical Image Analysis*. submitted.

Chatterjee, A., Volakis, J. & Windheiser, D. (1994), 'Parallel computation of 3D electromagnetic scattering using finite elements and conformal ABCs', *IEEE Transactions on Magnetics* **30**(5), 3606–3609.

Chen, E. & Marcus, B. (1998), 'Force Feedback for Surgical Simulation', *Proceedings of the IEEE* **86**(3), 524–530.

Cotin, S., Delingette, H., Clément, J., Bro-Nielsen, M., Ayache, N. & Marescaux, J. (1996), Geometrical and Physical Representations for a Simulator of Hepatic Surgery, *in* 'Proc. MMVR'96', IOS Press, pp. 139–151.

Cover, A., Ezquerra, N., O'Brien, J., Rowe, R., Gadacz, T. & Palm, E. (1993), 'Interactively Deformable Models for Surgery Simulation', *IEEE Computer Graphics and Applications* **13**, 68–75.

Daane, P., Constantinou, P. & Hesselroth, G. (1995), A $100 Surgical Simulator for the IBM PC, *in* 'Proc. MMVR'95', IOS Press, pp. 79–80.

Davis, M. & Carey, G. (1995), Parallel element-by-element spectral multilevel techniques for finite elements, *in* D. Bailey, P. Bjorstad,

J. Gilbert, M. Mascagni, R. Schreiber, H. Simon, V. Torczon & L. Watson, eds, 'Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing', pp. 393–394.

Dawson, S. & Kaufman, A. (1998), 'The Imperative for Medical Simulation', *Proceedings of the IEEE* **86**(3), 479–483.

Delp, S., Loan, J., Basdogan, C., Buchanan, T. & Rosen, J. (1996), Surgical simulation: an emerging technology for military medical training, *in* R. Zajtchuk, F. Goeringer & S. Mun, eds, 'Proceedings of the National Forum: Military Telemedicine On Line Today Research, Practice, and Opportunities', pp. 29–34.

Deussen, O., Kobbelt, L. & Tucke, P. (1995), Using simulated annealing to obtain good nodal approximations of deformable bodies, *in* D. Terzopoulos & D. Thalmann, eds, 'Computer Animation and Simulation '95', pp. 30–43.

Downes, M., Cavusoglu, M., Gantert, W., Way, L. & Tendick, F. (1998), Virtual Environments for Training Critical Skills in Laparoscopic Surgery, *in* 'Proc. MMVR'98', IOS Press, pp. 316–322.

Evans, D. (1994), 'Systolic array for finite elements (SAFE)', *Elektrotehniski Vestnik* **61**(4), 222–225.

Farhat, C., Sobh, N. & Park, K. (1990), 'Transient Finite Element Computations on 65536 Prozessors: The Connection Machine', *Int. J. for Numerical Methods in Eng.* **30**, 27–55.

Fischler, M., Tenenbaum, J. & Wolf, H. (1981), 'Detection of Roads and Linear Structures in Low-Resolution Aerial Imagery Using a Multisource Knowledge Integration Technique', *Comp. Graphics Image Proc.* **15**, 201–223.

Flanagan, D. & Belytschko, T. (1984), 'Eigenvalues and Stable Time Steps for the Uniform Strain Hexahedron and quadrilateral', *Journal of Applied Mechanics, March* **51**, 35–40.

Fung, Y. (1967), 'Elasticity of soft tissues in simple elongation', *Am. J. Physiol.* **213**(6), 1532–1544.

Fung, Y. (1993), *Biomechanics: Mechanical Properties of Living Tissues*, Springer-Verlag, New York.

Gao, L. (1995), 'Sonoelasticity Imaging: Theory and experimental verification', *J. Acoust. Soc. Am.* **97**(6), 3875–3886.

Gibson, J. (1950), *The Perception of the Visual World*, The Riverside Press, Cambridge, Mass.

Gibson, S. (1997), 3D ChainMail: A Fast Algorithm for Deforming Volumetric Objects, *in* 'Proc. Symp. on Interactive 3D Graphics', Providence, RI, USA, pp. 149–154.

Gibson, S., Samosky, J., Mor, A., Fyock, C., Grimson, E., Kanade, T., Kikinis, R., Lauer, H., McKenzie, N., Nakajima, S., Ohkami, H., Osborne, R. & Sawada, A. (1997), Simulating Arthroscopic Knee Surgery using Volumetric Object Representations, Real-Time Volume Rendering and Haptic Feedback, *in* 'Proc. CVRMed'97', Springer-Verlag, pp. 369–378.

Gropp, W., Lusk, E., Doss, N. & Skjellum, A. (1996), 'A high-performance, portable implementation of the MPI message passing interface standard', *Parallel Computing* **22**(6), 789–828.

Haber, R. & Henderson, M. (1980), *The Psychology of Visual Perception*, 2nd edn, Holt, Rhinehart and Winston, New York.

Hahn, J., Kaufman, R., Winick, A., Carleton, T., Park, Y., Lindeman, R., Oh, K.-M., Al-Ghreimil, N., Walsh, R., Loew, M., Gerber, J. & Sankar, S. (1998), Training Environment for Inferior Vena Caval Filter Placement, *in* 'Proc. MMVR'98', IOS Press, pp. 291–297.

Hajjar, J. & Abel, J. (1988), 'Parallel Processing for Transient Nonlinear Structural Dynamics of Three-Dimensional Framed Structures Using Domain Decomposition', *Computers & Structures* **30**(6), 1237–1254.

Hammond, S. & Law, K. (1988), 'Architecture and Operation of a Systolic Engine for Finite Element Computations', *Computers & Structures* **30**(1), 365–374.

Hansen, K. & Larsen, O. (1998), Using region-of-interest based finite element modelling for brain-surgery simulation, *in* W. Wells, A. Colchester & S. Delp, eds, 'Medical Image Computing and Computer Assisted Intervention MICCAI'98', pp. 305–316.

Hayes, L. (1989), Advances and Trends in Element-by-Element Techniques, *in* A. Noor, ed., 'State-of-the-art surveys on computational mechanics', The American Society of Mechanical Engineers, pp. 219–236.

Höhne, K., Pflesser, B., Pommert, A., Riemer, M., Schubert, R. & Tiede, U. (1995), 'A new representation of knowledge concerning human anatomy and function', *Nature Med.* **1**(6), 506–511.

Hug, J., Brechbühler, C. & Székely, G. (1999), Tamed Snake: A Particle System for Robust Semi-automatic Segmentation, *in* C. Taylor & A. Colchester, eds, 'Second International Conference on Medical Image Computing and Computer-Assisted Intervention MICCAI'99', number 1679 *in* 'Lecture Notes in Computer Science', Springer Verlag, pp. 106–115.

Hutter, R. (1999), Total hourglass control – eine robuste FE Methode zur Simulation von weichen Geweben, PhD thesis No. 13238, Department of Mechanical Engineering, ETH Zürich.

Hutter, R., Hora, P. & Niederer, P. (2000), 'Total Hourglass Control for Hyperelastic Materials', *Computer methods in applied mechanical engineering*. In press.

Immersion, I. (1995), 'Laparoscopic Impulse Engine', Company brochure, Santa Clara, CA.

Kass, M., Witkin, A. & Terzopoulos, D. (1988), 'Snakes: Active Contour Models', *Int. J. Comp. Vision* **1**(4), 321–331.

Kauer, M., Vuskovic, V. & Dual, J. (1999), In vivo measurements of elastomechanical properties of organic tissue, *in* 'European Medical and Biological Engineering Conference, EMBEC'99', Vienna, Austria.

Kelemen, A. (1998), Elastic Model-Based Segmentation of 2-D and 3-D Neuroradiological Data Sets, PhD thesis No. 12907, Department of Electrical Engineering, ETH Zürich.

Kelemen, A., Székely, G. & Gerig, G. (2000), 'Elastic Model-Based Segmentation of 3-D Neuroradiological Data Sets', *IEEE Trans. Medical Imaging*. In press.

Knapp, D., Kerr, J. & Sellberg, M. (1997), Patient Specific Color Texture Mapping of CT-based Anatomical Surface Models Utilizing Cryosectional Data, *in* 'Medicine Meets Virtual Reality', IOS Press, pp. 608–617.

Koelmans, A. (1981), *The Open Systems Interconnection Reference Model*, Rijksuniversiteit Groningern.

Kühnapfel, U., Çakmak, H. & Maaß, H. (1999), Modelling for Endoscopic Surgery, *in* 'Proceedings of the IEEE Symposium on Simulation', Delft, The Netherlands.

Kühnapfel, U., Krumm, H., Kuhn, C., Hübner, M. & Neisius, B. (1995), Endosurgery Simulations with KISMET: A flexible tool for Surgical Instrument Design, Operation Room Planning and VR Technology based Abdominal Surgery Training, *in* B. Gröttrup, ed., 'Proc. Virtual reality World'95, Stuttgart', Computerwoche Verlag, München, pp. 165–171.

Lawrence, D., ed. (1991), *Handbook of genetic algorithms*, van Nostrand Reinhold, New York.

Le-Tallec, P., Mohammadi, B., Sabourin, T. & Saltel, E. (1996), Distributed CFD on cluster of workstations involving parallel unstructured mesh adaption, finite-volume-Galerkin approach and finite-elements, *in* A. Ecer, J. Periaux, N. Satofuka & S. Taylor, eds, 'Parallel Computational Fluid Dynamics', pp. 521–526.

Loendorf, D. (1985), Development and Use of an Asynchronous MIMD Coputer for Finite Element Analysis, *in* 'Algorithmically specialized parallel computers', pp. 213–222.

Maenchen, G. & Sack, S. (1964), 'The TENSOR Code', *Methods in Computational Physics* **3**, 181–210.

Malone, J. (1990), 'Parallel Nonlinear Dynamic Finite Element Analysis of Three-Dimensional Shell Structures', *Computers & Structures* **35**(5), 523–539.

Manzini, G. (1994), 'Sparse Matrix Computations on the Hypercube and Related Networks', *J. of Par. and Distr. Computing* **21**, 169–183.

Massie, T. & Salisbury, J. (1994), The PHANToM Haptic Interface: A Device for Probing Virtual Objects, *in* C. Radcliffe, ed., 'Dynamic Systems and Control', Vol. 1, ASME, pp. 295–301.

Mehra, P. & Wah, B. (1992), *Artificial Neural Networks: Concepts and Theory*, IEEE Computer Society Press, Los Alamitos.

Meier, V. (1999), Realistic Visualization of Abdominal Organs and its Application in Laparoscopic Surgery Simulation, PhD thesis No. 13215, Department of Electrical Engineering, ETH Zürich.

Moulton, M. et al. (1995), 'An Inverse Approach to Determining Myocardial Material Properties', *J. Biomechanics* **28**, 935–948.

MPI: a message-passing interface standard (1994), *International Journal of Supercomputer Applications and High Performance Computing* **8**(3), 169–416.

Muthupillai, R., Rossman, P., Lomas, D., Greanleaf, J., Riederer, S. & Ehman, R. (1996), 'Magnetic Resonance Imaging of Transverse Acoustic Strain Waves', *Magnetic Rasonance in Medicine* **36**, 266–274.

Nevin, N. J. (1996), The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster, Technical Report OSC-TR-1996-4, Ohio Supercomputing Center, Columbus, Ohio.

Nour-Omid, B., Raefsky, A. & Lyzenga, G. (1987), Solving Finite Element Equations on Concurrent Computers, *in* A. Noor, ed., 'Parallel computations and their impact on mechanics', The American Society of Mechanical Engineers, pp. 209–227.

O'Toole, O., Playter, R., Krummel, T., Blank, W., Cornelius, N., Roberts, W., Bell, W. & Raibert, M. (1998), Assessing skill and learning in surgeons and medical students using a force feedback surgical simulator, *in* W. Wells, A. Colchester & S. Delp, eds, 'Medical Image Computing and Computer Assisted Intervention MICCAI'98', pp. 899–909.

Pflesser, B., Tiede, U. & Höhne, K. (1998), Specification, Modelling and Visualization of Arbitrarily Shaped Cut Surfaces in the Volume

Model, *in* W. M. Wells, A. Colchester & S. Delp, eds, 'First International Conference on Medical Image Computing and Computer-Assisted Intervention MICCAI'98', number 1496 *in* 'Lecture Notes in Computer Science', Springer Verlag, pp. 853–860.

Pommerell, C. (1992), Solution of large unsymmetric systems of linear equations, PhD thesis No. 9838, Department of Electrical Engineering, ETH Zürich.

Prusinkiewicz, P. & Lindenmayer, A. (1990), *The algorithmic beauty of plants*, Springer-Verlag, New York.

Reinig, K., Rush, C., Pelster, H., Spitzer, V. & Heath, J. (1996), Real-Time Visually and Haptically Accurate Surgical Simulation, *in* 'Health Care in the Information Age', IOS Press and Ohmsha, pp. 542–545.

Rhomberg, A., Enzler, R., Thaler, M. & Tröster, G. (1998), Design of a FEM computation engine for real-time laparoscopic surgery simulation, *in* 'Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing', IEEE Comput. Soc, Los Alamitos, CA, USA, pp. 711–715.

Sagar, M., Bullivant, D., Mallinson, G., Hunter, P. & Hunter, I. (1994), 'A Virtual Environment and Model of the Eye for Surgical Simulation', *Comp. Graphics* **28**, 205–212.

Satava, R. (1996), 'Virtual Endoscopy: Diagnosis using 3-D Visualisation and Virtual Representation', *Surgical Endoscopy* **10**, 173–174.

Satava, R. & Jones, S. (1998), 'Current and Future Applications of Virtual Reality in Medicine', *Proceedings of the IEEE* **86**(3), 484–489.

Saxena, M. & Perucchio, R. (1992), 'Parallel FEM Algorithms Based on Recursive Spatial Decomposition—I. Automatic Mesh Generation', *Computers & Structures* **45**(5), 817–831.

Schill, M., Wagner, C., Hennen, M., Bender, H.-J. & Männer, R. (1999), EyeSi - A Simulator for Intra-ocular Surgery, *in* C. Taylor &

A. Colchester, eds, 'Second International Conference on Medical Image Computing and Computer-Assisted Intervention MICCAI'99', number 1679 *in* 'Lecture Notes in Computer Science', Springer Verlag, pp. 1166–1174.

Sederberg, T. & Parry, S. (1986), 'Free-Form Deformation of Solid Geometric Models', *Computer Graphics* **20**(4), 151–160.

Sellberg, M., Murray, D., Knapp, D., Teske, T., Lattie, K. & Vanderploeg, M. (1995), Virtual Human: An Automated Virtual Environment for Computer-Aided Instruction and Biomechanical Analysis, *in* 'Interactive Technology and the New Paradigm for Healthcare', IOS Press and Ohmsha, pp. 340–348.

SensAble Devices, I. (1996), 'PHANToM', Company brochure, Cambridge, MA.

Soferman, Z., Blythe, D. & Nugel, J. (1998), 'Advanced Graphics Behind Medical Virtual Reality:Evolution of Algorithms, Hardware, and Software Interfaces', *Proceedings of the IEEE* **86**(3), 531–553.

Sterling, T., Savarese, D., Becker, D., Fryxell, B. & Olson, K. (1995), Communication overhead for space science applications on the Beowulf parallel workstation, *in* 'Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing', pp. 23–30.

Storaasli, O. & Ransom, J. (1987), 'Structural Dynamic Analysis on a Parallel Computer: the Finite Element Machine', *Computers & Structures* **26**(4), 551–559.

Stredney, D., Sessana, D., McDonald, J., Hiemenz, L. & Rosenberg, L. (1996), A Virtual Simulation Environment for LEarning Epidural Anesthesia, *in* 'Proc. MMVR'96', IOS Press, pp. 164–175.

Sussman, T. & Bathe, K. (1987), 'A Finite Element Formulation for Nonlinear Incompressible Elastic and Inelastic Analysis', *Computers & Structures* **26**(1/2), 357–409.

Sutton, C., Mccloy, R., Middlebrook, A., Chater, P., Wilson, M. & Stone, R. (1997), MIST VR-a laparoscopic surgery procedures trainer and evaluator, *in* K. Morgan, H. Hoffman, D. Stredney & S. Weghorst, eds, 'Medicine Meets Virtual Reality', pp. 598–607.

Suzuki, N., Hattori, A., Ezumi, T., Uchiyama, A., Kumano, T., Ikamoto, A., Adachi, Y. & Takatsu, A. (1998), Simulator for virtual surgery using deformable organ models and force feedback system, *in* 'Proc. MMVR'98', IOS Press, pp. 227–233.

Székely, G., Bajka, M., Hug, J., Manestar, M., Groscurth, P. & Haller, U. (1998), Anatomical model generation for laparoscopic surgery simulation, *in* 'Proc. 2nd Visible Human Project Conf., Bethesda, MA', National Library of Medicine, pp. 45–46.

Székely, G., Brechbühler, C., Dual, J., Enzler, R., Hug, J., Hutter, R., Ironmonger, N., Kauer, M., Meier, V., Niederer, P., Rhomberg, A., Schmid, P., Schweitzer, G., Thaler, M., Vuskovic, V., Tröster, G., Haller, U. & Bajka, M. (2000), 'Virtual reality-based simulation of endoscopic surgery', *Presence* **9**(3), 310–333.

Taylor, V., Ranade, A. & Messerschmitt, D. (1995), 'SPAR: A New Architecture for Large Finite Element Computations', *IEEE Trans. on Computers* **44**(4), 531–545.

Terzopoulos, D., Platt, J., Barr, A. & Fleischer, K. (1987), 'Elastically Deformable Models', *Computer Graphics* **21**(4), 205–214.

Tsai, J.-Y., Huang, J., Amlo, C., Lilja, D. & Yew, P.-C. (1999), 'The Superthreaded Processor Architecture', *IEEE Transactions on Computers* **48**(9), 881–902.

Tseng, T., Lee, Y., Chan, Y., Wu, S. & Chiu, A. (1998), A PC-Based Surgical Simulator for Laparoscopic Surgery, *in* 'Proc. MMVR'98', IOS Press, pp. 155–160.

Vining, D. (1996), 'Virtual Endoscopy: Is It Really?', *Radiology* **200**, 30–31.

Voß, G., Hahn, J., Müller, W. & Lindeman, R. (1999), Virtual Cutting of Anatomical Structures, *in* 'Proc. MMVR'99', IOS Press, pp. 381–383.

Vuskovic, V., Blaser, R. & Spiga, A. (1999), A vision-based device for in vivo measurements of elastomechanical properties of soft tissue, *in* 'Proc. Workshop on Image Processing in Applied Mechanics', Warshaw, Poland.

Wyrzykowski, R., Sczygiol, N., Olas, T. & Kanevski, J. (1999), Parallel finite element modeling of solidification processes, *in* P. Zinterhof, M. Vajtersic & A. Uhl, eds, 'Parallel Computation. Proceedings of the 4th International ACPC Conference', pp. 183–195.

Yagawa, G. & Shioya, R. (1993), 'Parallel finite elements on a massively parallel computer with domain decomposition', *Computing Systems in Engineering* **4**(4), 495–503.

Yalamanchili, K., Anand, S. & Warner, D. (1992), 'Three-Dimensional Finite Element Analysis on a Hypercube Computer', *Computers & Structures* **42**(1), 11–20.

Zhang, W. & Lei, E. (1991), 'A Parallel Frontal Solver on the Alliant FX/80', *Computers & Structures* **38**(2), 203–215.

Ziegler, R., Mueller, W., Fischer, G. & Goebel, M. (1995), A Virtual Reality Medical Training System, *in* 'Proc. 1^{st} In. Conf. on Comp. Vision, Virtual Reality and Robotics in Medicine, CVRMed'95, Nice, Lecture Notes in Comp. Sci., Springer-Verlag', Vol. 905, Springer, Berlin (etc.), pp. 282–286.

Zois, D. (1988), 'Parallel Processing Techniques for FE Analysis: Stiffness, Loads and Stresses Evaluation', *Computers & Structures* **28**(2), 247–260.

Curriculum Vitae

Personal Information

Alexander Clemens Rhomberg
Born 27. March 1971
Citizen of Hausen am Albis, ZH, Switzerland

Education

1977–1983: Primary school in Wollerau, SZ and Hausen, ZH

1983–1989: Grammar school in Zürich Wiedikon, ZH

1989–1995: M. Sc. in Electronic Engineering at the Swiss Federal Institute of Technology (ETH) in Zürich

1995–2000: Ph. D. at the Swiss Federal Institute of Technology

Work

1991: Internship at Crypto AG, Cham, CH: Cryptographic algorithms

1991–1992: Internship at Alcatel STR AG, Zürich, CH: Measuring a GHz PLL

1993: Internship at ENST Brest, France: Programming in Fortran

1995–2000: Teaching and Research Assistant at the Electronics Laboratory of the ETH Zürich