Institut für
Technische Informatik und
Kommunikationsnetze

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

# Parallel Image Registration in Distributed Memory Environments

Master Thesis of Michael Kuhn

MA-2004-05

Nizhny Novgorod, 3. Oktober 2004

Supervisor: Prof. Victor P. Gergel
Co-Supervisors: Placi Flury, Ulrich Fiedler
Professor: Bernhard Plattner

# Abstract

Image registration is the task of finding a spatial transformation that optimally maps one image onto another one. Thereby, usually a global optimum on a similarity function (metric) is searched. Out of the huge variety of different registration problems, many are computationally intensive and require large amounts of memory. None of the most prevalent software libraries for image registration provides parallel computing methods for distributed memory environments, although the task has successfully been parallelized in several projects. Therefore, this thesis addresses the extension of one of the most important libraries with a framework for parallel image registration in distributed memory systems, such as clusters of workstations. The developed framework is based on a distributed calculation of the metric function (as opposed to parallel optimization). The distribution is coordinated by a generic module which serves as the basis for the parallelization of new metric functions and at the same time hides the parallelization details from the user. Maximum flexibility with respect to the underlying hardware is ensured by an abstract communication layer and a caching mechanism allows to process even large images without exceeding the physical memory of the participating nodes. By interacting with the numerous components provided by the library, the framework can address a rich variety of different registration problems, which is exemplarily demonstrated in three scenarios. Based on these scenarios experiments have been carried out on different platforms. They show that in some cases a near linear speedup can be reached even on a large number of processes despite of the generic character of the methods and that the framework performs particularly well in data intensive applications because of the avoidance of disk swapping due to the combination of caching and the accumulated memory capacity in a cluster.

In der Bildregistrierung (Image Registration) geht es darum, eine räumliche Transformation zu finden, welche ein Bild optimal auf ein anderes abbildet.

Dazu wird üblicherweise ein globales Optimum auf einer Vergleichsfunktion (Metrik) gesucht. Aus der grossen Auswahl verschiedener Registrierungsprobleme sind viele rechenaufwendig und haben einen grossen Speicherbedarf. Keine der am weitesten verbreiteten Software Bibliotheken für Bildregistrierung verfügt über Parallel-Computing-Methoden für Umgebungen mit verteiltem Speicher, obwohl das Problem in verschiedenen Projekten erfolgreich parallelisiert wurde. Deshalb widmet sich diese Arbeit der Erweiterung einer der wichtigsten Bibliotheken mit einem Framework für parallele Bildregistrierung für Systeme mit verteiltem Speicher, wie zum Beispiel Workstation-Clusters. Das entwickelte Framework basiert auf einer verteilten Berechnung der Vergleichsfunktion (im Gegensatz zu paralleler Optimierung). Die Verteilung wird koordiniert durch ein generisches Modul, welches als Basis für die Parallelisierung neuer Metriken dient und gleichzeitig die Parallelisierungsdetails vor dem Benutzer verbirgt. Ein abstrakter Kommunikationslayer garantiert maximale Flexibilität im Bezug auf die zu Grunde liegende Hardware-Architektur, und durch einen Caching-Mechanismus wird es möglich, auch grosse Bilder zu bearbeiten ohne dabei die Arbeitsspeicher-Kapazität der beteiligten Knoten zu übersteigen. Die Interaktion mit den zahlreichen Komponenten, welche die Bibliothek zur Verfügung stellt, erlaubt den Einsatz des Framework in einer breiten Auswahl von Problemen im Gebiet der Bildregistrierung. Dies wird anhand dreier Beispiel-Szenarios exemplarisch demonstriert. Aufbauend auf diesen Szenarios wurden Experimente auf verschiedenen Plattformen durchgeführt. Diese zeigen, dass trotz des generischen Charakters der Methoden in gewissen Fällen ein beinahe linearer Speedup erreicht werden kann, und dass das Framework in datenintensiven Problemen besonders gute Ergebnisse erzielt. Dies, da durch eine Kombination von Caching und der akkumulierten Speicherkapazität eines Clusters das in sequenziellen Programmen auftretende Disk-Swapping vermieden werden kann.

# Contents

# Chapter 1

# Introduction

Image registration is the process of establishing a point-by-point correspondence between two images of a scene. The images can thereby be acquired by different sensors or by the same sensor in different points in time or from a different point of view. Image registration is applied in several fields, such as in medical imaging, in stereo vision applications, for motion analysis, object localization or image fusion.

The correspondence between the images is defined by a spatial transformation, which is usually computed by an optimization process. Thereby an optimizer searches a global optimum on a function which defines a similarity measure (*metric function*[1]) between the images dependent on their relative position.

For more profound information about image registration please refer to Section 2.1.

Different software packages, libraries and frameworks exist that carry out image registration. Some of the most prevalent tools are the Automated Image Registration (AIR) package [1], the Flexible Image Registration Toolbox (FLIRT) [2], the VTK CISG Registration Toolkit [3], the Image Processing Toolbox of Mathworks (for Matlab) and the Insight Segmentation and Registration Toolkit (ITK) [4]. Among these projects, ITK is the most generic and extensive approach. Apart from the Image Processing Toolbox of Mathworks, all the above mentioned packages are open source projects that are freely available. However, only the Insight Segmentation and Registration Toolkit can be used free of charge in commercial applications.

Dependent on the application, image registration can be a resource demanding task. The main problems are the sometimes large amounts of data,

[1] the similarity function is often called metric function, even though it does not necessarily meet the definition of a metric in the mathematical sense

that can easily exceed the physical memory size of a state of the art computer, as well as the search space, which can reach a considerable size[2] in some problems. Exploring large search spaces and data sets inherently causes long computing times. What makes things even worse is the fact, that for large images, computers can run out of available physical memory. The resulting disk swapping causes a considerable increase in computing time.

Parallel computing has proven to be an efficient technique to solve both, memory and performance problems. Most specialists dealing with image registration are not familiar with parallel and distributed computing techniques. To bridge the gap between specialists of different disciplines, Squyres et al. [5] as well as Seinstra and Koelma [6] propose to integrate parallel computing methods into image processing libraries. Despite of the fact, that such methods have successfully been applied to specific image registration problems, no generic software modules exist that address the image registration task by means of parallel computing.

This thesis addresses the problem of parallelizing the image registration task and thereby tackles performance as well as memory issues. The goal was to develop methods that act as generic modules which can be used in a multitude of registration problems. This should be achieved by extending an existing library. To account for the diversity of the addressed registration problems the methods had to be designed for clusters of workstations (which are cheap and highly available) rather than for dedicated parallel computers.

Parallelization methods for optimization tasks can be divided into coarse and fine grained. While in coarse grained methods multiple independent function evaluations are parallely executed, fine grained methods calculate the basic computational steps in parallel. According to Eldred and Schimel [7], the parallel efficiency for coarse grained methods is usually better, and best performance and scalability can be achieved by combining coarse and fine grained methods.

Applying this scheme to image registration shows that there are basically two strategies: parallelizing the optimization method or parallelizing the metric function evaluation. These strategies can also be combined. Approaches based on both, parallel optimization as well as parallel metric function evaluation have successfully been realized before.

When applying coarse grained methods, memory issues still remain critical, since each function evaluation requires the whole data set. Furthermore, coarse grained methods usually scale only within a limited range. Several

[2]For deformable problems, search spaces of up to $9.8 \times 10^6$ parameters have been reported. However, more typical are search spaces from about six to a few hundred parameters.

voxel based registration techniques are well suited for fine grained parallel computation based on parallel metric function evaluation. This thesis concentrates on exactly these techniques.

A generic module for parallel metric computation has been developed, which extends the basic registration framework of the ITK library. Together with the huge diversity of registration components within ITK, this module allows the use in a multitude of applications. A caching mechanism has been introduced that takes advantage of the fact that each processor works on a partial image only when parallely evaluating a metric function. Enabling this mechanism allows to treat even large data size problems with a moderate amount of memory.

The developed methods have been integrated into three different scenarios and thereby successfully been tested for suitability in two and three dimensional, rigid and non-rigid as well as intra- and inter-modal registration. On two different cluster systems, more that 1000 experiments have been carried out. Thereby the speedup[3] compared to the equivalent scenarios composed of standard ITK modules as well as scalability issues have been examined. The analysis of time spent in different parts of the code further allowed to gain an insight into strengths and deficiencies of the developed framework.

It could be shown, that in some cases, a speedup of more than 15 can be reached on 16 and of over 50 on 64 processors. The use of caching, together with the accumulated memory capacity of a cluster made it even possible to reach parallel efficiencies[4] of far more than 100% compared to a sequential program that causes the operating system to start disk swapping.

As a conclusion from these results, it can be said that the ITK library was successfully extended by a generic framework for parallel image registration, which allows to treat even memory intensive problems.

The rest of this thesis is structured as follows:

**Background:** This chapter gives background information to *image registration*, *parallel computing* and *ITK* and thereby introduces the most important terms.

**Related Work:** The Related Work chapter first discusses the motivation of the project in more detail, based on work previously done in the area. Then an overview of alternatives to parallel computing to address the performance and memory problems in image registration is given. Finally, other projects that address image processing and registration by means of parallel computing are reviewed.

[3] See Section 2.2 for a definition.

[4] See Section 2.2 for a definition.

**Problem Statement:** The Problem statement chapter explains the problem tackled by this thesis in detail.

**Requirement Specification:** This chapter defines the exact scope of the thesis. It covers requirements and restrictions on the support of registration methods and the underlying hardware as well as an the extendibility of the developed methods.

**Distributed Registration Framework:** The developed methods are presented and explained in the Distributed Registration Framework chapter. Concepts as well as implementation are discussed in-depth. A reader that intends to use or extend the developed modules is strongly encouraged to read this chapter.

**Evaluation:** The Evaluation chapter discusses the methodology as well as the results of the evaluation of the developed framework.

**Future Work:** This chapter outlines, how the framework can further be improved.

**Conclusion:** The Conclusion chapter gives a concise review of what has been reached by this thesis and what has been left over to future projects.

# Chapter 2

# Background

## 2.1 Image Registration

Image Registration is the process of finding a spatial transformation that establishes optimal correspondence between two (or more) images. This is illustrated in Figures 2.1 and 2.2.

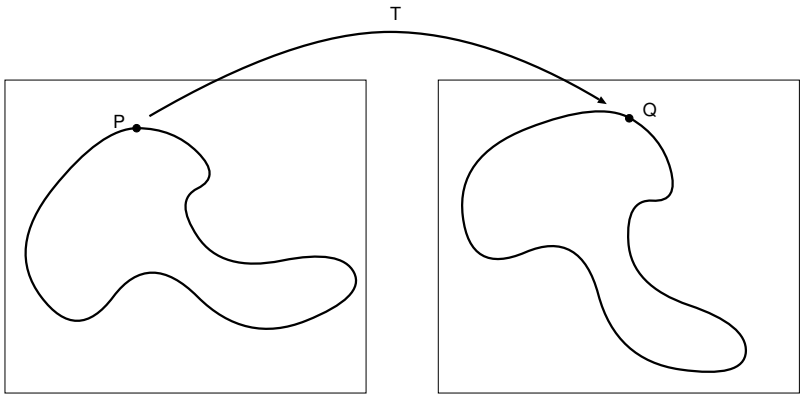



Figure 2.1: The goal of image registration is to find the transformation that optimally maps one image onto another one.

There are many different fields where image registration is applied, such as medicine, remote sensing, computer vision or cartography to name just a few. Two examples can be found in Figures 2.3 and 2.4. Some of the main goals are the combination of data acquired by different sensors, optimally aligning images or objects recorded from different viewpoints, analyzing changes in images taken at different points in time or matching an object model into a

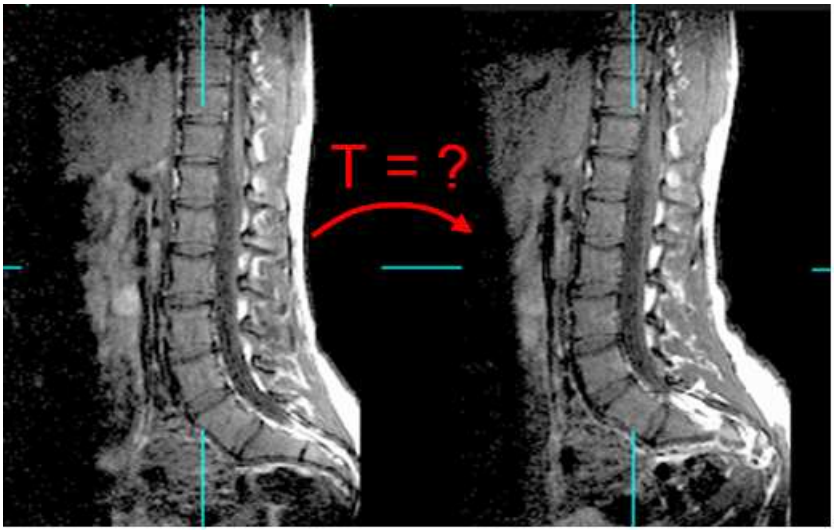



Figure 2.2: Goal: Find the (deformable) transformation that maps the image to the left onto the image to the right. (image modified from http://www-ipg.umds.ac.uk/p.edwards/warping/spapp.html, accessed 1 October 2004)

given scene.

A typical example for image registration is the combination of the information contained in an magnetic resonance imaging (MRI) and a computed tomography (CT) scan, which makes it necessary that these scans are optimally aligned.

Because of the diversity of applications and of the goals aimed at by the use of image registration, a variety of methods have been developed to address the task. Despite this variety, there are some main components common to most of these methods.

Some kind of *similarity measure* (*metric*) is applied to match *features* of the images to each other. The diversity of features used is again huge, ranging from raw pixel intensities to data structures produced by sophisticated object recognition methods. A global optimum of the similarity measure dependent on the parameters of the *spatial transformation* used during mapping is searched according to an *optimization strategy*. The spatial transformation is thereby restricted to the distortions that are to be expected for a given application. During optimization, one image (*fixed image*) is usually kept fixed while the other image (*moving image*) is permanently being transformed. This process is depicted in Figure 2.5.

According to the spatial transformation used during mapping, *rigid*, *non-rigid* or *deformable registration* methods are distinguished. While in rigid

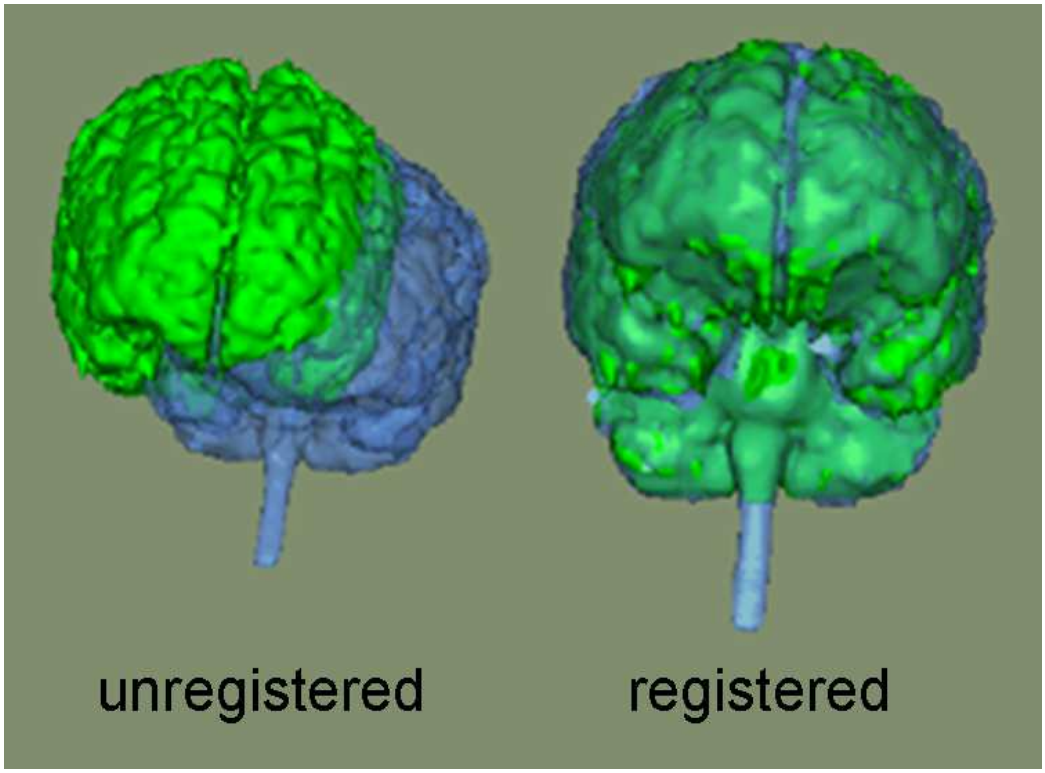



Figure 2.3: Example of a medical application: Applying the spatial transformation found during registration of the unregistered brain images (left) brings them into correspondence (right).

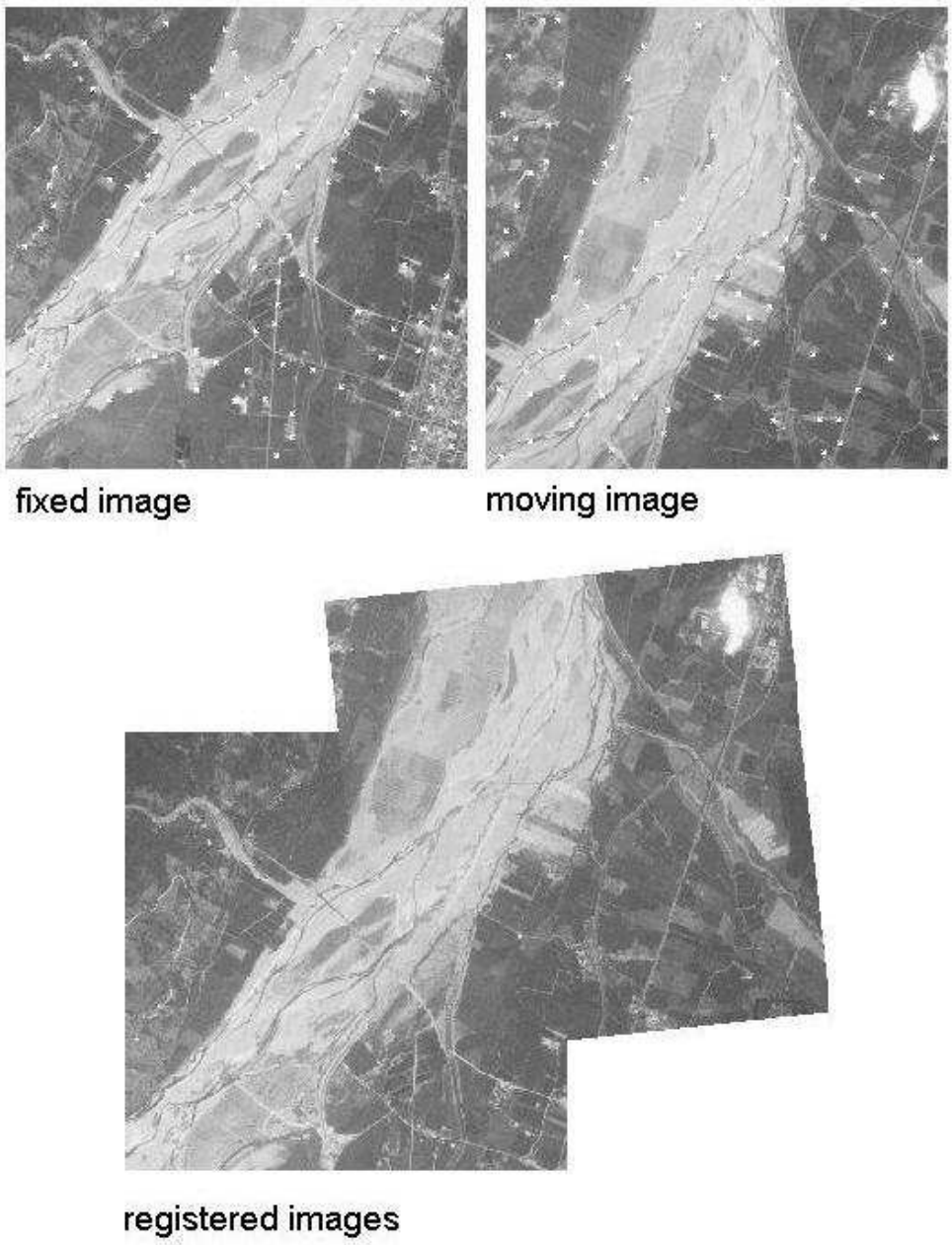



Figure 2.4: Example of an application in cartography.

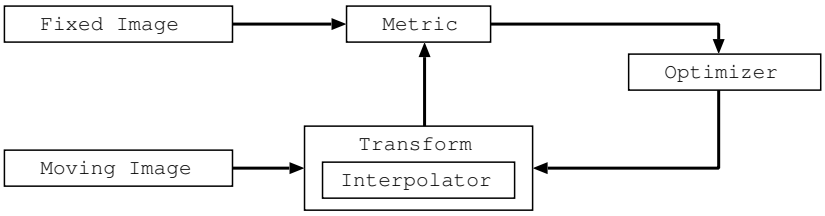



Figure 2.5: Registration is basically an optimization process involving the following steps: 1) transform the moving image and compare it to the fixed image using a metric function. 2) dependent on value and derivative of the metric function, an optimizer calculates a new transformation. 3) back to 1 until a stop condition of the optimizer is reached.

registration the transformation only consists of rotations and translations, no such restriction exists in non-rigid registration. A special kind of non-rigid registration is deformable registration, where elastic models define how an image is allowed to be distorted.

Further, registration methods are often classified according to the type of sensors involved. In *intra-modal* registration the images result from the same type of sensors. This is not the case in *inter-modal* (or *multi-modal*) registration, an example of which is the combination of MRI and CT data mentioned before. Moreover, there are methods that match intensity data to geometrical models. They are called *model based* registration methods.

Finally it is possible to distinguish between *area-based* and *feature-based* algorithms. An explicit feature-extraction step allows feature matching in feature-based methods. This step is omitted in area-based methods where raw image data is used for correlation estimation. This project only covers a special kind of area based methods that works by direct pixel intensity comparisons. Such methods are called *intensity based*.

## 2.2 Distributed and Parallel Computing

Because of the ever growing need for computation speed, lots of approaches to combine the power of several processors have been developed in past. Thereby a variety of different architectures have been proposed.

A popular classification scheme for parallel computing architectures was defined by Flynn. In a computer there are two basic types of information: data and instructions. Flynn's taxonomy distinguishes four types of computer systems according to the number of streams of each type of information

[8]:

- *Single Instruction Single Data* (SISD): these are usual sequential computers
- *Multiple Instruction Single Data* (MISD): such systems are rather unusual, however there exist some kind of pipeline architectures that fit this definition.
- *Single Instruction Multiple Data* (SIMD): a typical example for such an architecture is a vector processor
- *Multiple Instruction Multiple Data* (MIMD): there are several systems matching this definition, a cluster of workstations for example.

Often the class of MIMD systems is further subdivided into *shared memory* and *distributed memory* systems. In shared memory systems all processors work on the same, shared address space. In distributed memory systems, on the other hand, each processor has assigned its own memory which other processors cannot directly access. The most prevalent means of communication in such systems is message passing.

Systems based on message passing often build up on libraries that take care of the involved low level details, such as buffering, data type conversion in heterogeneous clusters or error handling. The most important libraries are the *Parallel Virtual Machine* (PVM) and different implementations of the *Message Passing Interface* (MPI) standard.

While Flynn's taxonomy classifies parallel hardware architectures, there is a similar scheme for parallel programming models. It distinguishes between *single program multiple data* (SPMD) and multiple program multiple data (MPMD) programs. While in SPMD architectures the same program is run on every processing node, different pieces of software carry out the calculation in the MPMD case.

MPMD architectures are typical in applications where entirely different calculations have to be carried out on different nodes. Often, these calculations are thereby applied to the same set of data. This kind of parallel architecture is also referred to *task parallelism*. SPMD programs are usually applied when the same operations are applied to different parts of the data. This case is also referred to as *data parallelism*.

Often, one process slightly differs from the rest in the SPMD case. This can be achieved by adding conditional statements that assign a special task to the process with rank 0. The *rank* of a process is a unique identifier that is assigned to each process. Usually, process ranks are integers that range

from 0 to the $n-1$ where $n$ is the number of processes participating in the calculation.

The methods developed throughout this project are designed for distributed memory MIMD systems. The process with rank 0 has a special task assigned and the modules can be deployed in SPMD as well as MPMD programs.

The quality of a parallel program is often measured in terms of two parameters: *parallel speedup* and *parallel efficiency*. Parallel speedup is defined as

$$S_N = T_S/T_N$$

where $T_S$ is the execution time of the best sequential algorithm[1] and $T_N$ is the execution time on $N$ processors.

Parallel efficiency is defined as

$$E_N = S_N/N$$

where $N$ is the number of processors.

An example should show the meaning of these measures: If the fastest sequential algorithm executes in 8 seconds ($T_S = 8$), and the parallel algorithm takes 2 seconds on 5 processors ($N = 5$, $T_5 = 2$) we get a speedup $S_5$ of 4. A speedup of 4 using 5 processors can be considered as an efficiency of 80%, excatly as denoted by the measure $E_5$.

There are different kinds of parallelization overhead that inherently limit speedup. Often this overhead is due to the necessary interprocess communication an is mainly characterized by the number and sizes of messages in distributed memory systems, and by synchronization issues in shared memory systems. Further, parallel programs may perform poorly because of unequally loaded nodes. Nodes might stay idle because they are waiting for others to complete their task. Sophisticated load balancing algorithms were proposed to handle this problem. Finally, there is limiting factor which is known as *Amdahl's Law*. Amdahl's Law states that the maximum speedup that can be reached by a parallel program is

$$S \leq \frac{1}{f + (1-f)/N}$$

[1]Note that it is difficult to define $T_S$, because it is not clear which processor (a single processor in the parallel environment? the fastest available serial processor?) should be taken as reference and because it is not generally possible to define the best sequential algorithm because of several reasons such as memory versus performance trade-off, dependency on different problem properties or changes over time.

where N is the number of processors and f is the non-parallelizable (and hence $1 - f$ the parallelizable) fraction of the code [9]. This is a theoretical limit, in practice this speedup will not be reached because of the parallelization overhead mentioned before.

Amdahl's law can be modified to take into account the problem size $W$ (which is basically the same as $T_S$) as well as any overhead $T_{po}$ caused by the parallelization [10]:

$$\begin{aligned} S_N &= \frac{W}{fW + (1-f)(W/N) + T_{po}} \\ &= \frac{N}{1 + (N-1)f + \frac{NT_{po}}{W}} \\ &\rightarrow \frac{1}{f + \frac{T_{po}}{W}} \text{ for } N \rightarrow \infty \end{aligned}$$

These equations show, that the non-parallelizable fraction, as well as the parallelization overhead should be minimized in order to achieve optimal speedup.

## 2.3 ITK and its Architecture

The Insight Segmentation and Registration Toolkit (ITK) is an open-source image processing library written in C++. It was designed to support the Visual Human Project [11] and mainly focuses on medical image segmentation and registration. ITK was developed by six principal organizations and funded by the National Library of Medicine at the National Institutes of Health. However, many other organizations and individuals have been helping to evolve the software. Even though ITK is already a large package, it is still rapidly growing. Reasons for that are the fact that ITK is open source and can be used free of charge even in commercial applications, as well as the good support and documentation provided by the main developers.

Different technical features have been incorporated into ITK. These are motivated by a design and implementation philosophy outlined in the ITK Style Guide [12] and the ITK Readme [13]. The following sections are intended to give an overview of the most important architectural features. Some general design decisions are covered by Section 2.3.1. A discussion of the core feature of the ITK architecture, a pipeline mechanism for data processing, follows in Section 2.3.2. Section 2.3.3 finally gives an overview of the architecture of the basic registration framework within ITK. A de-

tailed overview of ITK and its concepts and classes can be found in the ITK Software Guide [4].

### 2.3.1 General Features

ITK is designed to be compilable on multiple platforms, that is on several important C++ compilers and on different operating systems such as Windows, Unix or MacOSX.

To ensure a common build procedure on all platforms, CMake is used. CMake is a cross-platform build environment that allows to translate generic makefiles into makefiles for a particular compiler.

Besides the support for different operating systems and compilers, ITK further provides bindings to other programming languages, namely TCL, Python, and Java.

To keep the algorithms and modules applicable to a large variety of different problems, ITK makes extensive use of generic programming techniques. To achieve high performance and at the same time keep the architecture generic, many modules are templatized, which allows to make decisions at compile rather than at run time. The `itk::Image` class, for example, is templatized over the pixel type and the number of dimensions. In addition to pixel-type images, other data representations such as point sets and meshes are supported. In a generic programming model, there are objects that store, and objects that operate on data. Objects that store data are also referred to as *containers* and objects that operate on it as *algorithms*. Often, a third class of objects, the so called *iterators* are introduced. Iterators provide a generic interface for algorithms to access data within containers and are an important part of the programming model of ITK.

Another part of this programming model are the so called *smart pointers*, which have been introduced because C++ does not take care about memory allocation and freeing. Most objects within ITK are referenced by smart pointers that realize reference counting and automatically release memory that is not used any longer. Further features are object factories, which allow extensions of the system at run-time, a command-observer design pattern used for event handling and a pipeline architecture for data processing. This pipeline architecture is the core concept within ITK and it is therefore discussed in more detail in the following section.

### 2.3.2 Pipeline Architecture and Data Streaming

Basically, two types of objects are involved in data processing: *data objects* such as images and meshes that represent data, and *process objects* that

operate on data objects. These classes are connected by a data-flow pipeline mechanism as depicted in Figure 2.6. At the beginning of a pipeline, there is a process object usually referred to as *source* (a file-reader, for example). Its output is an ITK data object. To this data object, another process object (a *filter*) is attached which again produces a data object and so on. The pipeline is finally terminated by a so called *mapper*, a special process object that writes data to a file or any other output system.

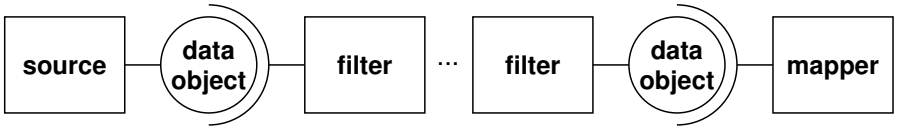



Figure 2.6: Illustration of the pipeline mechanism within ITK: process objects are connected to data objects, which in turn are connected to other process objects again.

This architecture does not only describe the flow of data, but also incorporates some sophisticated concepts. It takes care of memory management and makes sure the pipeline is always up to date while guaranteeing that only those sections are executed that have changed. It further enables data streaming and multi threading (see below).

Within the pipeline, there is some trade off between memory efficiency and execution speed. That is, once a filter has calculated its output, this output can be kept in memory for re-usage, or the memory can be released and the filter has to re-execute if the output is needed again at a later point in time. Therefore, each filter has a release data flag which optionally can be set by the user. This allows to optimally adjust the pipeline behavior to the application needs.

Execution of the filters is triggered by the mapper's `Update()` method. This update request is delegated in *upstream* direction to the data object at the mapper's input, from there to the filter producing this data object and so on (see Figure 2.7). To avoid redundant processing, the pipeline keeps track of any modifications on process and data objects and makes sure a filter is only executed when necessary. As soon as the update request reaches the source of the pipeline, filters that need to execute do so in turn in *downstream* direction, that is in direction from source to mapper. A filter executes when its `GenerateData()` method is invoked. In Figure 2.8 this process is illustrated in a simplified way, i.e. by neglecting the interactions between data and process objects.

An important feature of the ITK pipeline is its capability to process data

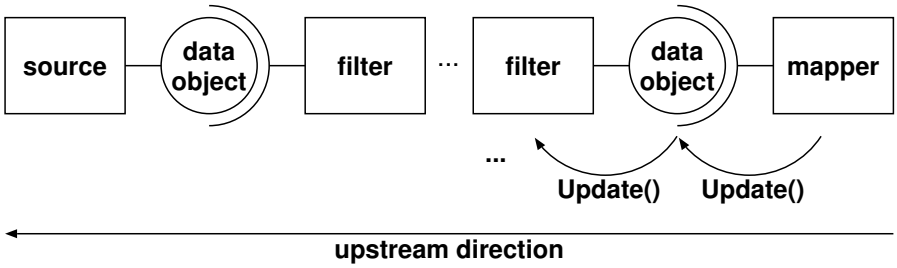



Figure 2.7: Illustration of the interaction between process and data objects for the case of the `Update()` method. As soon as a mapper is asked to update its output, it delegates this update request to the data object at its input. This data object in turn delegates the request to the process object by which it is generated, and so on.

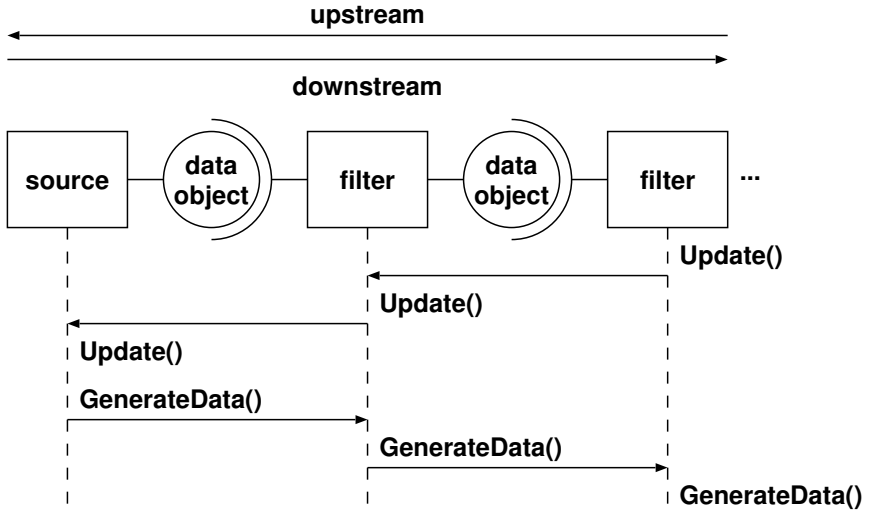



Figure 2.8: Illustration of the pipeline execution: First, update requests propagate upstream from the mapper to the source, and then filters execute in turn by invoking their `GenerateData()` method.

in pieces. This concept, which is named *streaming*, permits to apply the filters within the pipeline on sub-images and to reassemble the whole image at the end of the pipeline. Therefore, even very large images can be processed without exceeding physical memory size. Obviously, not every filter can be applied to partial images, because of global data dependencies. Therefore, only parts of the toolkit are streaming capable. Even though some metric functions are suited to be applied to partial images, none of the registration methods within ITK supports streaming at the time of this writing.

Filters that support streaming are well suited for parallel computation. Therefore, they usually support multi-threaded execution. The basic concept, splitting data into pieces for processing, is the same. Thus, *multi-threading* comes at little cost when building up on the streaming architecture. A filter simply has to implement the `ThreadedGenerateData()` method instead of the `GenerateData()` method in order to be multi-threading capable.

The processing of sub-images is realized by a concept consisting of three different image regions. These regions are the *largest possible region*, which represents the entire dataset, the *buffered region*, which is a contiguous block of memory allocated by filters to hold their output and the *requested region* which is the part of the dataset that a filter is asked to process. An `ImageRegionSplitter` allows to divide the image region into several requested regions which are then processed one after the other.

The `Update()` method of data objects, invokes three methods:

- `DataObject::UpdateOutputInformation()`
- `DataObject::PropagateRequestedRegion()`
- `DataObject::UpdateOutputData()`

The output information consists of meta-data which allows to map the pixel coordinate system to a position in real world. It has to be changed when an image is shrunk, for example.

As indicated by its name, `PropagateRequestedRegion()` asks a preceding filter for an output of requested size. Sometimes a filter cannot satisfy this request and produces a larger output. Further, a filter might need a larger region at its input in order to generate its output. This occurs, for example, when some additional boundary pixels are required by a filter kernel.

The third method called by `Update()` is `UpdateOutputData()`, which determines whether a particular filter needs to execute. Once a filter executes, all filters downstream have to do so as well.

### 2.3.3 Basic Registration Framework

Since the multitude of registration problems asks for different methods to solve them, ITK provides different registration frameworks, each suitable for some particular applications.

There exist four registration frameworks within ITK, three of which are only suited for specialized tasks. Two of these specialized frameworks treat different deformable registration problems and the third makes model to image registration[2] possible. Besides these very specialized methods, a *basic image registration framework* allows to solve a multitude of more general registration tasks. It provides intensity based algorithms for registration problems defined by global transforms as well as for some simple deformable problems. This project only covers the basic registration framework, the architecture of which is discussed within this section.

As seen in Section 2.1, four main modules are required to solve the registration task, namely a similarity measure (or *metric*), a *transform* that defines the search space, an *optimizer*, and an *interpolation method*. These four modules also make up the main part of the basic registration framework within ITK. Besides them, there is a controller class, the `ImageRegistrationMethod`. The `ImageRegistrationMethod` sets up the necessary interconnections between the other modules and it is possible to register an observer to it, which reacts to events generated during the registration process.

The four main modules are more or less freely interchangeable. This flexibility allows to create a multitude of registration setups suited for lots of different applications.

From the user's point of view, the modules are assigned to the `ImageRegistrationMethod` class, which hides the further details. It keeps a reference to each module and then sets up the necessary interconnections. That is, it assigns the transform and interpolator to the metric and the metric to the optimizer (see Figure 2.9). The optimizer does the actual registration task and its execution is wrapped by the controller's `StartRegistration()` method.

The choice of the transform is mainly affected by the nature of the overall problem, that is by the distortion that is to be expected between the images. To avoid holes and overlaps, the transform is applied in *inverse* direction. Each pixel in the fixed image is transformed and the intensity value at the transformed point is read from the moving image. Transforms within ITK work on world coordinates as opposed to pixel coordinates. Image meta-data, such as pixel spacing and origin thereby allow to convert pixel positions into

[2] see Section 2.1.

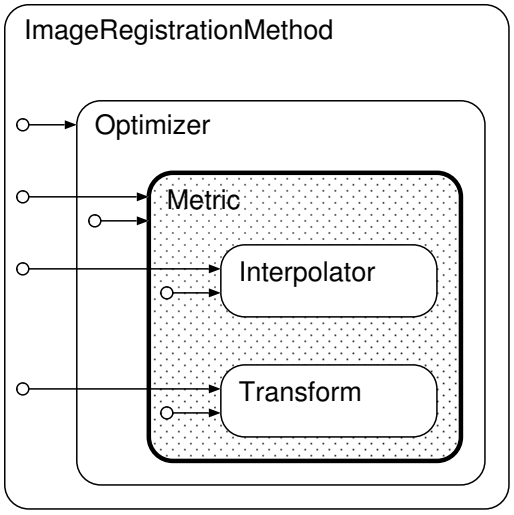



Figure 2.9:

world coordinates. Using world coordinates for example guarantees that even non-isotropic images are correctly treated.

ITK provides three interpolation methods: nearest neighbor, linear and bspline. The choice of the interpolator mainly affects the quality of the solution. Methods that ask for sub-pixel accuracy, for example, require sophisticated interpolation methods. Such methods, however, are not well suited for real time applications because of the high computational cost. More accurate interpolators further result in smoother cost functions as illustrated in Figure 2.10. This can be of advantage during the optimization process.

The optimizer directly operates on the transform parameters. Therefore, the transform restricts the set of possible optimizers, that is, not all transform-optimizer pairs optimally work together. At the time of this writing, all optimization algorithms implemented in ITK work iteratively. Most of them are gradient based optimization methods, but there are others as well, an evolutionary algorithm, for example.

The decision which metric function to use is often the most crucial one, and greatly depends on the characteristics of the image sensors. While all cost functions within ITK can be applied in intra-modal problems, only the subset of cost functions based on mutual information can be used in inter-modal applications. The metric further greatly affects the catch range of the optimizer, and its choice is therefore dependent on the possibility for a sophisticated initialization and on choice of the optimizer. The aim of this project is, to make it possible to parallely calculate the metric functions of

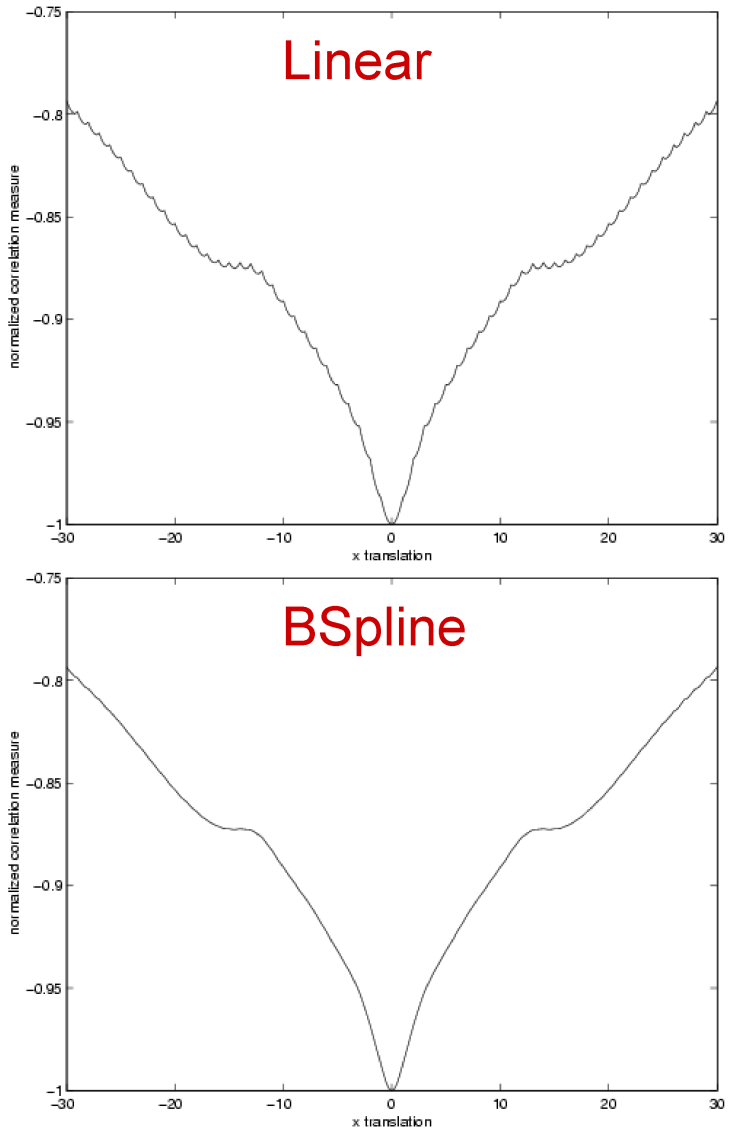



Figure 2.10: The metric function along one transform parameter for linear (top) and bspline interpolation (bottom). (images from http://www.itk.org/CourseWare/Training/RegistrationMethodsOverview.pdf)

ITK. They are therefore going to be presented in more detail now. At the time of this writing, there exist 11 of them:

- `GradientDifferenceImageToImageMetric`
- `MattesMutualInformationImageToImageMetric`
- `MeanReciprocalSquareDifferenceImageToImageMetric`
- `MeanSquaresImageToImageMetric`
- `MutualInformationImageToImageMetric`
- `NormalizedCorrelationImageToImageMetric`
- `CorrelationCoefficientHistogramImageToImageMetric`
- `MeanSquaresHistogramImageToImageMetric`
- `MutualInformationHistogramImageToImageMetric`
- `NormalizedMutualInformationHistogramImageToImageMetric`
- `KullbackLeiberCompareHistogramImageToImageMetric`

Whereas the metric functions related to mutual information can be applied in inter- as well as in intra-modal registration, all the other functions are only applicable in intra-modal registration.

The most common cost function is the mean squares metric, which pixel-wise sums up the squared differences of the intensity values. The mean reciprocal metric works on a pixel by pixel basis as well, however, instead of adding the squared differences, the differences are added after passing them through the bell-shaped function $\frac{1}{1+x^2}$. The gradient difference metric compares image gradients instead of intensities, the normalized correlation metric is principally a cross correlation approach and the mutual information metrics are based on a statistical method that minimizes the joint information of the overlaid images measured by entropies of the intensity distribution functions. Those metrics carrying the term 'Histogram' in their name are all derived from a common superclass and operate on joint histogram data of the two images. The computational complexity differs greatly for different metrics. For most optimization strategies, the metric value as well as the derivative at a given position have to be evaluated. Obviously, the derivative calculation becomes more time-consuming the more transform parameters are involved. That is, for rigid transforms, for example, the derivative calculation is less complex than for general global transforms.

An extension of the basic image registration framework described above is the multi-resolution registration framework. It consists of the same basic modules but operates on two so called image pyramids instead of two simple images as its input. An image pyramid represents an image at different resolution levels. The framework takes care of all the necessary steps involved when switching from one to the next resolution level. Multi-resolution registration can considerably reduce computing time for large input images, and often the optimization process becomes more robust since there are less undesired local optima for low resolution images (refer also to Section 3.2).

# Chapter 3

# Related Work

## 3.1 Motivation

A huge diversity of different methods have been developed to address the image registration problem. Some of them demand for long calculation times. Dependent on the application, the main reasons for this computational expense are the sometimes large image sizes [14, 15, 16], the large number of parameters involved in the transform that creates a huge search space [15, 17] and the fact, that finding a global optimum on a highly non-linear metric function is a complex task that asks for time consuming optimization methods [18, 19].

For registrations based on linear transformations, encountered execution times on a single processor on state of the art hardware at the respective time of writing were 3-4 minutes for a resolution of $128 \times 128 \times 21$ pixels [20] (1998), between 10 and 80 minutes for an image resolution of $125 \times 125 \times 250$ pixels [16] (2003) and about 45 minutes for approximately $256 \times 256 \times 50$ pixels [21] (1998). In deformable applications, the search space can be significantly larger, which results in a considerable increase in computational complexity [15]. In such problems, computing times of up to 330 minutes were reported by Rohlfing and Maurer [15] in 2003 and in 1996 Christensen et al. [17] stated that the calculation of a deformable registration problem with $9.8 \times 10^6$ transform parameters on a $128 \times 128 \times 100$ voxel volume can take up to 7 days on a 150 MHz MIPS R4400.

Besides increasing the number of arithmetic operations, large data sizes can also cause problems with the memory constraints given by off-the-shelf computers [16, 21]. Image resolutions of more than $500 \times 500 \times 500$ pixels are not unusual in medical applications [16, 22]. Such images require storage in the order of the available physical memory on a typical workstation computer.

Exceeding physical memory causes disk swapping and therefore a drastic increase in computing time. The problem of data size is likely to even increase in future, because of ever more data produced by ever better acquisition devices [14] and because of a trend from 2D to 3D image processing [21].

In some registration applications, long computing times are unacceptable. Among them are real time applications such as in range data analysis [23] or sensor fusion [24], applications that have to cope with the huge amount of data daily produced by a large amount sensors [14] and clinical applications that ask for fast imaging in intra-operative situations as well as to cut down time between scan and surgery [15, 21, 25].

Besides an enormously higher computing power, parallel computing can also provide a solution to memory problems by using the aggregated memory capacity provided by all nodes participating in the computation [14, 26, 27]. This encourages the use of parallel computing in image registration applications when performance is a problem. Since image processing experts are usually not quite familiar with parallel computing, different groups suggested to hide the parallelization details of image processing tasks by the development of parallel image processing libraries, such as presented in [5], [28] and [6]. At the time of this writing, no such libraries exist that provide generic building blocks for a multitude of registration applications in a distributed memory parallel processing environment.

## 3.2 Alternative Approaches to Cut Down Computing Time

Reducing execution time is an issue in image registration applications, and parallel computing is a possible solution to it, as outlined in the preceding section. However, parallel computing asks for expensive resources. Therefore it is a must to consider other alternatives that cut down computing times prior to applying parallel computing methods. In image registration, this can be achieved by data as well as search space reduction [14].

Two main strategies are applied to reduce data sizes: Registration of sub-images only [14] and multi-resolution registration as applied in [14], [20] or [21]. The idea of the multi-resolution approach is to start at a low image resolution to roughly approach the final solution and to successively refine the image resolution to eventually reach the required accuracy.

Another idea to reduce data is the application of some kind of feature extraction prior to registration, instead of working on full pixel intensity values. This, however, can lead to information loss which might be undesirable

in applications that ask for high accuracy [20].

The most important step in search space reduction is the application of an appropriate optimization method, that efficiently explores the search space. To avoid getting stuck in a local optimum on the metric function, often global optimization methods such as genetic algorithms are applied as in [14], [18] or [19]. These methods, however, are usually more time consuming than local optimization methods [18, 19]. Another approach to avoid local optima is to provide an initial guess transform prior to applying a local optimization method [16, 19, 21]. Finally, it is possible to rely on the multi-resolution approach described above, which is more robust with respect to getting trapped in a local optimum, since less of them exist at a low resolution level [20, 21].

Further search space reduction techniques can be found in deformable problems. Examples are successive grid refinement [15] or compensation for global differences by applying a registration based on global transforms prior to the deformable registration [21].

All of these methods provide efficient ways to optimize image registration performance. However, they also show certain drawbacks. Data reduction techniques such as sub-image registration or feature extraction might cause less accurate registration results and search space reduction techniques do not solve the memory problems caused by large data sets. The same holds for the multi-resolution approach as soon as it comes to working on the highest resolution level. These drawbacks can be overcome by applying parallel computing methods.

## 3.3 Parallel and Distributed Computing in Image Processing and Registration

Even though communication overhead is one of the most important reasons that limits the speedup of parallel applications, many authors state, that image processing tasks are good candidate for parallel computing because of the large data sizes involved [5, 9, 26, 27, 29, 30]. The inherent overhead of data distribution can only be overcome by parallel file system [5].

However, the fact that many image processing tasks are inherently parallel [5, 28, 31] often outweighs the problems caused by the data amounts and allows to achieve near linear speedup [5, 28]. Often, data parallelism is applied to account for the localized operations in low level algorithms [9] and to cope with the involved data amounts [29]. Different strategies have been proposed to reduce transmission times of large amounts of data. These

include using IP-Multicast [32, 33] for communication involving many nodes and compressing data prior to transmitting it, which sometimes allows to achieve an increased throughput. However this greatly depends on data and system properties [34, 35].

Image registration methods exhibit components that have some degree of natural parallelism. Typically, either the optimization method or the cost function is parallelized. However, other approaches exist as well, such as parallely calculating general matrix operations [24], solving differential equations [17] or decomposing wavelets [36].

Among optimization methods, the most popular candidates for parallelization are evolutionary algorithms, as applied in [14], [18] and [19]. The main reason for this popularity is the inherent parallel character of such algorithms as pointed out by Salomon et al. [19]. However, other optimization methods, such as the simplex algorithm, have successfully been parallelized as well [15].

Many intensity based cost functions work on a pixel-by-pixel basis, exactly as typical low level image algorithms do. Approaches that take advantage of this inherently parallel nature of metric functions are presented in [15], [21], [23] and [25].

Eldred and Schimel [7] discuss the problem of parallel optimization from a more theoretical point of view. To avoid the scaling problems of parallel optimization methods and at the same time taking advantage of their parallel nature, a combination with parallel function evaluation is proposed. This approach has been applied to image registration by Rohlfing and Maurer [15] by combining parallel optimization with parallel metric calculation.

Parallel image processing has been deployed on different hardware architectures. While some of them take advantage of the high number of processing units in an SIMD system [17, 24, 36], others trust in the higher flexibility of dedicated MIMD architectures [15, 19, 21]. Further, several approaches build upon clusters of workstations [14, 18, 23, 25, 39, 40]. These systems usually provide less computing power than dedicated architectures, however their low cost, the high availability and the simplicity of adding further nodes outweigh this disadvantage in many applications [37].

According to Fleury et al. [38], shared memory systems are better suited for image processing applications because they avoid excessive memory to memory movements, and Downton and Crooks [9] state, that, while shared memory systems have successfully been applied in practice, there are problems in reaching real-time performance with clusters of workstations. Other examples however show, that cluster and grid computing can be an alternative to shared memory systems [5, 14, 18, 23, 25, 28, 29, 30, 39, 40]. Considering future trends, the authors of [9] state, that MIMD systems built

from common microprocessors are likely to rule out dedicated parallel processors and they expect a trend towards switch based networks (as opposed to point-to-point interconnections) and towards topology independent parallel algorithms.

# Chapter 4

# Problem Statement

As shown in Section 3.1, there are several factors that can cause image registration to take more computing time than desired for a particular application. The most important factors are the large data sizes and search spaces. Even though there are several methods that reduce computing times on single processor systems, there are applications where it makes sense to use parallel computing methods. The main reasons that speak in favor of parallel computing are memory issues and the often inherent parallel character of the problem.

The goal of this project is to speed up the image registration task by means of parallel computing methods in a distributed memory environment. Generic methods applicable in various applications should be developed in a way that they are available to a wide audience. Thereby particular attention should be paid to applications involving large amounts of data.

The methods have to be integrated into the basic image registration framework of ITK (see Section 2.3). That way, it is ensured that the resulting framework is generally applicable and available to a wide range of users. The implementation finally has to be investigated with respect to performance and scalability.

As seen before, there are two main strategies to parallelize the image registration task: Parallelizing the optimization method or parallelizing the cost function evaluation. When applying parallel optimization, each processing node requires a full copy of each, the fixed and the moving image. These images have to be distributed as well as stored in main memory. When, on the other hand, the load of calculating the cost function is distributed, each processing node typically works on a partial image only. Therefore, in principle only one copy of each, the fixed and the moving image has to be distributed and only partial images have to be kept in memory at each processing node. Efficiently coping with large data sizes is a main goal of this

project. Therefore, the developed methods should be based on distributed metric calculation rather than parallel optimization. Further, only intensity based metric functions has to be considered. Such functions are usually well suited for parallelization as shown in Section 3.3.

Despite of the advantageous characteristics of parallel metric calculation when large data sizes are involved, network traffic and memory issues are still likely to become a serious problem. These issues have to be considered particularly carefully and sophisticated data management techniques should be incorporated if necessary.

To achieve the outlined goals, the project is organized in the following four major subtasks:

- specification of the detailed requirements of the implemented software (functional specification),
- design of an architecture for distributed image registration that fits into the existing toolkit (ITK),
- implementation of the proposed architecture,
- investigation of the performance increase and the scalability.

Each of these subtasks is going to be discussed separately in the following sections.

## 4.1 Requirement Specification

First, a requirement specification with a clear interaction and service model has to be defined. This model is the foundation for identifying the possibilities for parallelization and distribution and helps to delimit the scope of the thesis. The requirement specification should enfold the set of registration problems that has to be covered by the developed methods. Further it has to define the requirements on scalability, on the amount and kind of data that has to be supported and on hardware properties such as constraints on heterogeneous computer systems differing in processing power and memory, or on constraints and requirements of the communication subsystem.

## 4.2 Architecture Design

An architecture for distributed image registration based on parallel metric calculation that fits smoothly into the existing part of ITK has to be designed.

Thereby the overall architecture of the toolkit should be taken into account, allowing flexible extensions of the proposed architecture in future. Such an extension could be the combination of the distributed metric calculation with parallel optimization or the development of further distributed cost functions. Besides pure computational speedup, the architecture should take into account memory issues and network traffic.

The design of such an architecture involves the following steps:

- studying the architecture of ITK and the characteristics of all cost functions used in the basic image registration framework of the toolkit. Methods only applied in deformable and model based registration should only roughly be looked at, keeping future extensions in mind. The cost functions should mainly be investigated with respect to their feasibility for parallel calculation (local vs. global data dependencies) and with respect to their relations to other methods within the toolkit. Such relations can be similar mathematical characteristics as well as relations within C++ (inheritance).

- based on above investigation of ITK as well as the previously defined requirements, a design should be specified that exactly states which parts of ITK should be implemented in a distributed manner and how this should be done.

- above design should be refined by stating how to deal with issues concerning the underlying hardware as they are to be expected according to the requirement specification. Particular attention should be paid to data management issues.

- based on these considerations an architecture has to be proposed that addresses all issues discussed before.

## 4.3 Implementation

Even though making the code official part of ITK is not the goal of this project, it should be written such that an integration can be achieved at a later point in time. Therefore, the code should be written with the ITK Style Guide in mind, and tests on different platforms, as they are supported by ITK, should be carried out during the implementation process. The requirements and restrictions of ITK should also be considered when choosing a particular interprocess communication infrastructure.

## 4.4 Evaluation

The implemented architecture has to be investigated with respect to scalability and parallel efficiency for scenarios that cover the previously defined requirements. Memory consumption as well as computation time should be considered theoretically as well as in experiments.

A sophisticated methodology has to be thought of that allows to quantitatively assess the implementation. This methodology should define which parts are evaluated, to what they are compared and how the measurements are carried out.

# Chapter 5

# Requirement Specification

## 5.1 Image Registration Methods

The existing approaches to parallel image registration, as presented in Section 3.3, all focus on particular problems. None of them provides the flexibility to solve a wide variety of different registration tasks. To create such a flexible parallel image registration toolkit is exactly what this project aims at. The idea is to extend a widely used image registration library, the Insight Segmentation and Registration Toolkit (ITK), with parallel computing methods. ITK is mainly thought for and used in medical applications, however its generic architecture allows the use in other fields as well.

As shown in Section 2.3.3, there exist several registration frameworks within ITK. During this project only the most generally applicable of them, the basic image registration framework, has to be extended. Further, only the load of calculating intensity based metric functions is addressed by this project, as stated in Section 4.

The characteristics of such functions are quite diverse. While some of them have local data dependencies only, others work on intermediate representations gathered from the whole image. The architecture of the methods developed throughout this project have to cover all of them.

To ensure maximum flexibility of the resulting framework, any of the further components required in image registration, namely transform, interpolator and optimizer, have to be supported by the distributed methods. Particular attention should be paid to those components that cause high computational complexity.

It is important to note that the methods mentioned before need to be covered only by the design. Not all configurations have to be evaluated, and mainly in the case of metric functions, not all of them have to be imple-

mented. In a first step, a prove of concept implementation with a simple cost function, such as the mean squares metric, should be realized. Then, a metric function applicable in inter-modal registration problems, i.e. one based on mutual information, can be parallelized.

## 5.2 Distributed Architectures

It cannot be expected that there is dedicated parallel hardware around in typical (i.e. medical) application fields of ITK. One option to overcome this problem is by remote access to high performance computers. However, this is typically expensive and makes dependent from the provider of the system. As mentioned in Section 3.3, cluster computing can provide a powerful and affordable alternative to dedicated parallel hardware. Therefore, the target environment on which the extended library should be applicable are clusters of workstations. Typical cluster sizes might range from a few, say three or four, computers within a laboratory in a hospital up to a hundred and more nodes in research labs. Any of these clusters should be supported by the architecture. Because of the typically large amounts of data involved, a fast network (100 Mbps and more) will be indispensable and can be considered a prerequisite during architecture design. As a topology, a datagram based network is assumed, which virtually connects all nodes to each other and which uses the IP protocol to address them. Further, it can be expected that the nodes within the cluster are located close to each other, that is in the same building or complex, which results in low network delays. It is assumed that TCP is used at the transport layer, which leads to reliable communication links.

Unfortunately, clusters of workstations have the disadvantage of unequally loaded nodes, as seen in Section 3.3. For an effective operation in a real world application, sophisticated load-balancing algorithms therefore have to be provided. In a first step only a simple static load balancing scheme needs to be implemented. This assumes identical workstations within the cluster that are equally loaded at any time. However, the architecture should be designed such, that the extension with advanced dynamic load balancing methods is possible in future. This allows the use in heterogeneous clusters with unequally loaded nodes.

## 5.3 Requirements on Data to be Processed

The diversity of typical applications of ITK asks for different image types and methods, such as 2D rigid or 3D deformable registration. The parallel framework is supposed to cover the same types of data as the basic registration framework of ITK.

As mentioned before, parallel computing is usually interesting where large volumes of data are involved. Therefore, images that contain up to several hundred millions of pixels (and hence require several hundred megabytes for storage) have to be processable.

As shown in Section 3.3, data compression prior to transmission over a network can increase overall throughput. This, however, greatly depends on the data involved and on the underlying network. Therefore the architecture should allow to add compression methods developed for particular applications and particular data in future.

## 5.4 Requirements on Extensibility

ITK is a rapidly evolving library. Therefore, it is essential to keep the architecture flexible, allowing extensions in future.

As mentioned earlier, the architecture has to be designed such that metric functions added to the toolkit in future can profit from the parallel computing methods developed throughout this project. This will in general not be possible without extra work, but the additional effort required to integrate a new function into the distributed framework should be as low as possible.

It would be interesting to design the architecture such that support for multi-threading, as it is provided by other ITK methods, can be added at little cost in future. That way, shared memory systems could easily take advantage of the developed methods.

## 5.5 Summary

### 5.5.1 Image Registration Methods

**Supported by the Architecture**

- only intensity based methods
- only the load of calculating the metric functions needs to be parallelized
- only the basic image registration framework

- all metric functions within the basic registration framework
- all transformation methods within the basic registration framework
- all optimization strategies within the basic registration framework
- all interpolation methods within the basic registration framework

**Covered by the Implementation**

- at least one typical metric function has to be parallelized
- several combinations of modules should be tested

### 5.5.2 Distributed Architectures

**Supported by the Architecture**

- clusters of workstations
  - from 3 to 100 hosts
  - connected by a switched network allowing a throughput of 100 Mbps and more
  - using the IP protocol to address hosts
  - hosts located not far from each other
  - reliable communication links (assuming TCP as transport protocol)
  - unequal processing power and load

**Covered by the Implementation**

Experiments have to be carried out on at least one typical cluster according to the description above. Only a static load balancing scheme has to be implemented, i.e. processing power and load can assumed to be identical for all hosts.

### 5.5.3 Requirements on Data to be Processed

**Covered by the Architecture**

- 2 and 3 dimensional data
- large images (several hundred millions of pixels)

- use of the streaming capabilities of the toolkit to process large data
- possibility to preprocess data by compression methods prior to sending

**Covered by the Implementation**

- support for 2 and 3 dimensional data
- streaming support for large data
- no compression methods have to be implemented

### 5.5.4 Requirements on Extensibility

- other registration frameworks and methods do not have to be considered
- it should be possible to add new metric functions at little cost
- if possible with reasonable effort, the architecture should be designed such that it can later be extended to fully support the multi-threading capabilities of the toolkit.

# Chapter 6

# Distributed Registration Framework

## 6.1 Overview

The distributed registration framework extends the existing registration framework by adding support for parallel metric calculation in a distributed memory environment. The distributed metric calculation is organized in a master-slave manner as illustrated in Figure 6.1. The master process is located on the same node as the image data. It is responsible for data distribution as well as for the communication between the existing registration framework and the slave nodes and it hides the distribution details from the user. The slaves do the major part of the metric function evaluation and work in parallel which leads to the targeted speedup. To each slave a region of the fixed image is assigned. For this part, the respective slave calculates some intermediate metric value. In intensity based metric functions, such an intermediate value is calculated based on the comparison of each pixel of the assigned fixed image region with the respective pixel of the moving image. The master provides the slaves with the data they need to carry out their calculations. It further coordinates all steps required to collect and process the partial results and passes the final result to the registration framework.

Besides the assigned fixed image region, each slave is provided with the whole moving image to make sure it has all necessary data at its disposal in order to calculate the intermediate metric value. Typically, only a small part of this image is accessed, however. Therefore a caching mechanism that can optionally be enabled has been introduced. This caching mechanism together with the subdivision of the fixed image makes sure that even for large data size problems memory consumption in the slaves is kept low.

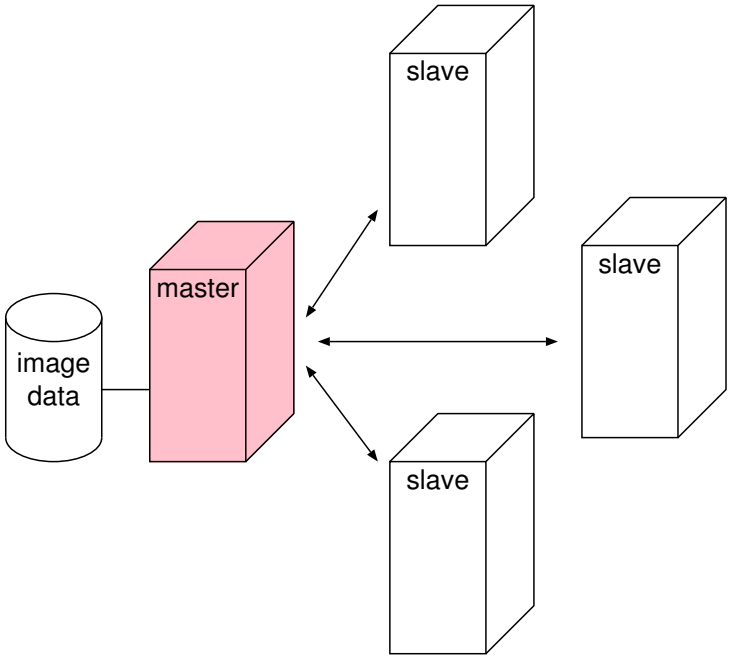



Figure 6.1: There is exactly one master process. It distributes the image data to the slaves and coordinates the distributed calculation.

Within the whole distributed framework, an abstract interface is used for all communication tasks. This ensures maximum flexibility with respect to the underlying hardware and communication subsystem.

These concepts are implemented using the following basic classes:

- `DistributedImageToImageMetric`
- `RegistrationCommunicator`
- `CacheImage`

The core element of the distributed registration framework is the `DistributedImageToImageMetric` class, which is divided into a master and a slave part. It is derived from the `itk::ImageToImageMetric` class which serves as a superclass to all similarity functions used within the basic registration framework of ITK.

The `DistributedImageToImageMetric` class is an abstract superclass. In order to create a specialized similarity function, some virtual methods have to be implemented by derived classes. These functions define the way the actual similarity measure is calculated and therefore determine the actual behavior of the metric.

Communication is realized through an abstract class called `RegistrationCommunicator`, which provides an interface for all the necessary communication tasks. Specialized implementations of this interface allow to adapt the registration framework to different parallel hardware and communication libraries. At the moment, only one specialization based on MPI is realized.

A `CacheImage` organizes its data in a block structure and makes sure that only recently accessed blocks are kept in main memory, while others are stored on disk. A user can choose whether images are stored as cache- or as usual images, which perform better as long as memory is not critical.

## 6.2 Distributed Metric

### 6.2.1 Concepts

The `DistributedImageToImageMetric` is the core of the distributed registration framework. It is a closed module which can be plugged into the basic registration framework, and which consists of a master and a slave part. The master part coordinates the whole calculation and thereby hides the distribution details from the user while the slaves do the actual work. This basic setup is illustrated in Figure 6.2.

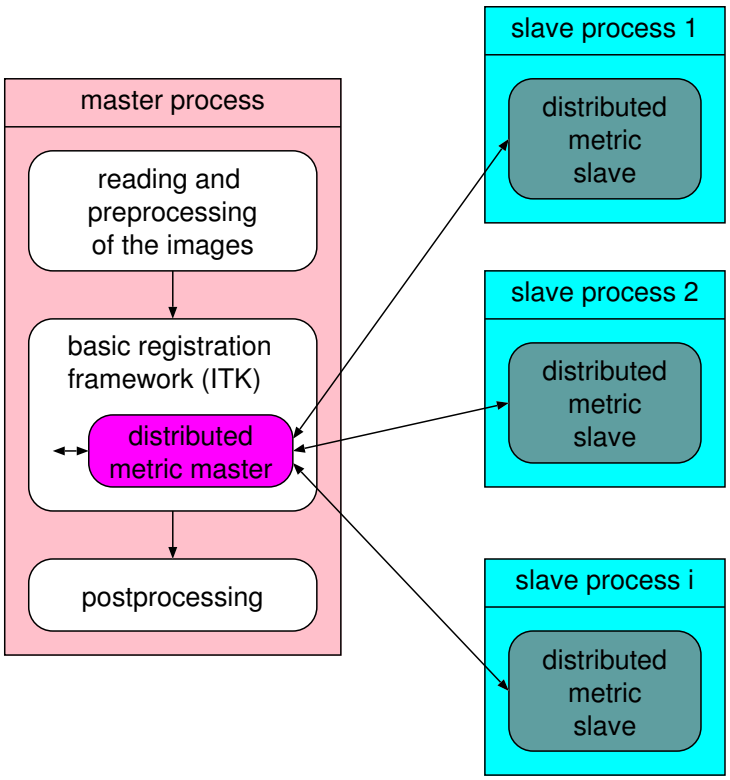



Figure 6.2: The metric master is plugged into an ITK program as any other metric. A minimal slave program which instantiates and starts the metric slave has to be written in addition. The master takes care of the communication with the basic registration framework and the slaves.

The whole registration process consists of two major stages: Initialization and optimization. They are treated separately in the following sections.

### Initialization

During initialization, the master provides the slaves with the data they need to carry out their calculations. A natural way to divide data for computation is to assign a part of the fixed image to each slave. Since the fixed image does not change in the course of the optimization, it can be distributed over the slaves at the beginning of the registration process. The slaves then all the time operate on the identical sub image. Unfortunately, for the moving image things are more complicated, since it is permanently transformed during the registration process. There are basically three strategies on how to deal with the moving image:

- for each slave: During each iteration calculate the part of the moving image needed for the current transform and send that piece of data to the slave that is not available at the slave's node.
- for each slave: Based on some user provided hints calculate the maximum region of the moving image that is required to carry out the whole registration and send this region to the respective slave.
- broadcast the whole moving image to each slave before the registration process starts.

In the first approach, the amount of data to be sent to each slave is minimized. However, during each iteration the part of the image that needs to be sent has to be calculated for each slave. Since the transformations involved can be quite complex, calculating this part is rather expensive, even though it can easily be computed in parallel, since it has to be done on a per slave basis. Further, image data has to be sent during each iteration, which causes a considerable amount of parallel overhead and thus reduces the possible achievable speedup according to Amdahl's law[1]. Most data sent during the ongoing registration process is data that has already been sent earlier[2]. Its amount greatly depends on the initial position and it should obviously be minimized.

[1] Any communication results in parallelization overhead. The extended version of Amdahl's law (Section 2.2) states, that this overhead inherently limits the achievable speedup. Since this overhead occurs in each iteration, it can make up a large part of the program.

[2] Note, that it has been *sent* by the master. The receiver was, however, another slave, therefore it has to be resent.

It would of course be desirable to know the total region required by each slave in advance. Then this region could be sent at the beginning of the registration process and no further calculation and communication overhead occurs. However, it is a highly non-trivial task to calculate this region in advance. Only sophisticated user input would allow to do that. The resulting interface most likely became very complicated and hence difficult to handle and prone to mistakes.

Distributing the whole moving image to each slave at the beginning of the registration run guarantees that enough data is available at each node, involves much less computations and eases the interface as well as the implementation. At a first glance this approach seems to require enormous amounts of data to be sent over the network. However, a tree structured communication scheme allows to broadcast a message to $N$ hosts in only $T_M \cdot \log_2(N)$ seconds, where $T_M$ is the time required to send the message to a single host[3]. When optimizing the send process at the network layer, the time required can even be reduced to $1 \cdot T_M$ when using multi-cast techniques such as IP-Multicast. Assuming an identical network bandwidth at each node, it becomes apparent that the bottleneck of the data distribution is the master node. Therefore, rather than minimizing the amount of data to be sent to each slave, the amount of data sent from the master node should be minimized. A further advantage of this approach is, that there is little effort required to adapt the distributed framework to profit from a parallel file system. Only an image reading capability has to be added to the slave to achieve such a support. Considering these facts, it was decided to go for the broadcasting approach, which results in a data distribution as illustrated in Figure 6.3.

Keeping the whole moving image in each slave is not quite optimal with respect to memory usage. While this does not bother as long as small images are involved, it can become a serious issue in large size problems. In each slave, typically only a small part of the moving image (in the order of the size of the fixed image part) is accessed. To account for that, a caching mechanism has been introduced, which is discussed in detail in section 6.4.

For some metric functions, besides the fixed and the moving image, a third image, the gradient image of the moving image, is required in the slaves. Using the broadcasting approach, this image can be calculated locally in the slave node prior to starting the optimization process. When bspline interpolation is applied, finally a forth image, holding the bspline coefficients

[3]Tree structured communication works as follows: First, process 0 sends the message to process 1. Then process 0 sends the message to process 2 and process 1 sends it to process 3. Then all four processes send the message to another four processes and so on.

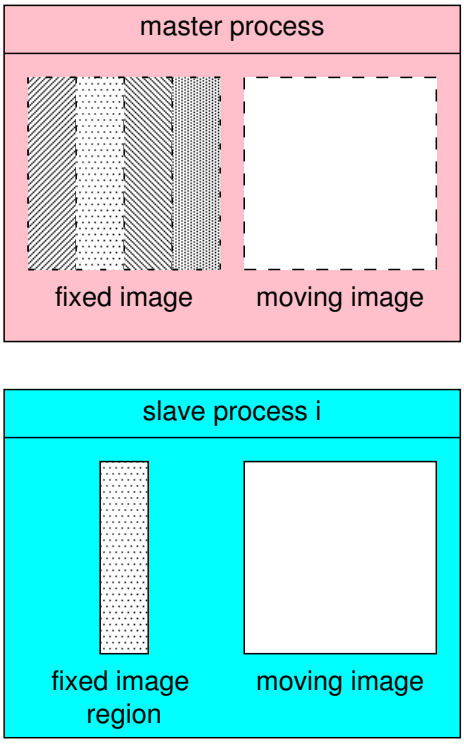



Figure 6.3: Simplified illustration of the data distribution after the metric initialization. The images in the master node are drawn in dashed lines, because dependent on the application they can be removed from memory after initialization. In reality, there might be a gradient and a coefficient image on the slave side dependent on the choice of metric and interpolator. These images are typically large compared to the moving image. See Section 6.4 for details.

needs to be kept in the slaves. Typically, the gradient as well as the coefficient image are large compared to the moving and the fixed image. Since the access characteristics are equal to those for the moving image, the same caching mechanism can be applied in order to cut down memory usage.

#### Optimization phase

After distributing the image data, the framework is ready to start the optimization process. All the different optimizers within ITK work iteration based. During each iteration an optimizer dependent number of metric values and derivatives is requested from the metric function. Whenever a new value (or derivative[4]) is required, the optimizer addresses a request to the metric master including the transform parameters for which the function has to be evaluated. The master then passes the transform parameters to the slaves and asks them to calculate the partial value associated with their fixed image region. Calculating the partial value can potentially involve several stages, however, usually only one stage is required. First, the slaves calculate intermediate results and transmit them to the master. The master in turn processes these results and, if required, distributes the processed intermediate results to the slaves. Based on these results, further intermediate results are calculated and again transmitted to the master. This process is repeated an arbitrary (metric dependent) number of times. From the last set of intermediate results the master finally computes the actual metric value and passes it to the optimizer. The steps required in order to calculate a metric value are depicted in Figure 6.4

The optimization process ends as soon as any of the optimizer's stop conditions is reached. At this point the registration process terminates. The slaves get informed about that and quit.

### 6.2.2 Implementation

#### Initialization

The `DistributedImageToImageMetric` consists of a master and a slave part, which are implemented within the same class. The master part is identified by the process with rank zero. Any instance of the metric running on a process with a rank unequal to zero automatically becomes a slave.

The data distribution is realized in the `Initialize()` method. While for the master process, this method is automatically called by the registration

[4]For simplicity only the process involved to calculate a metric value is discussed. Calculating derivatives works analogously.

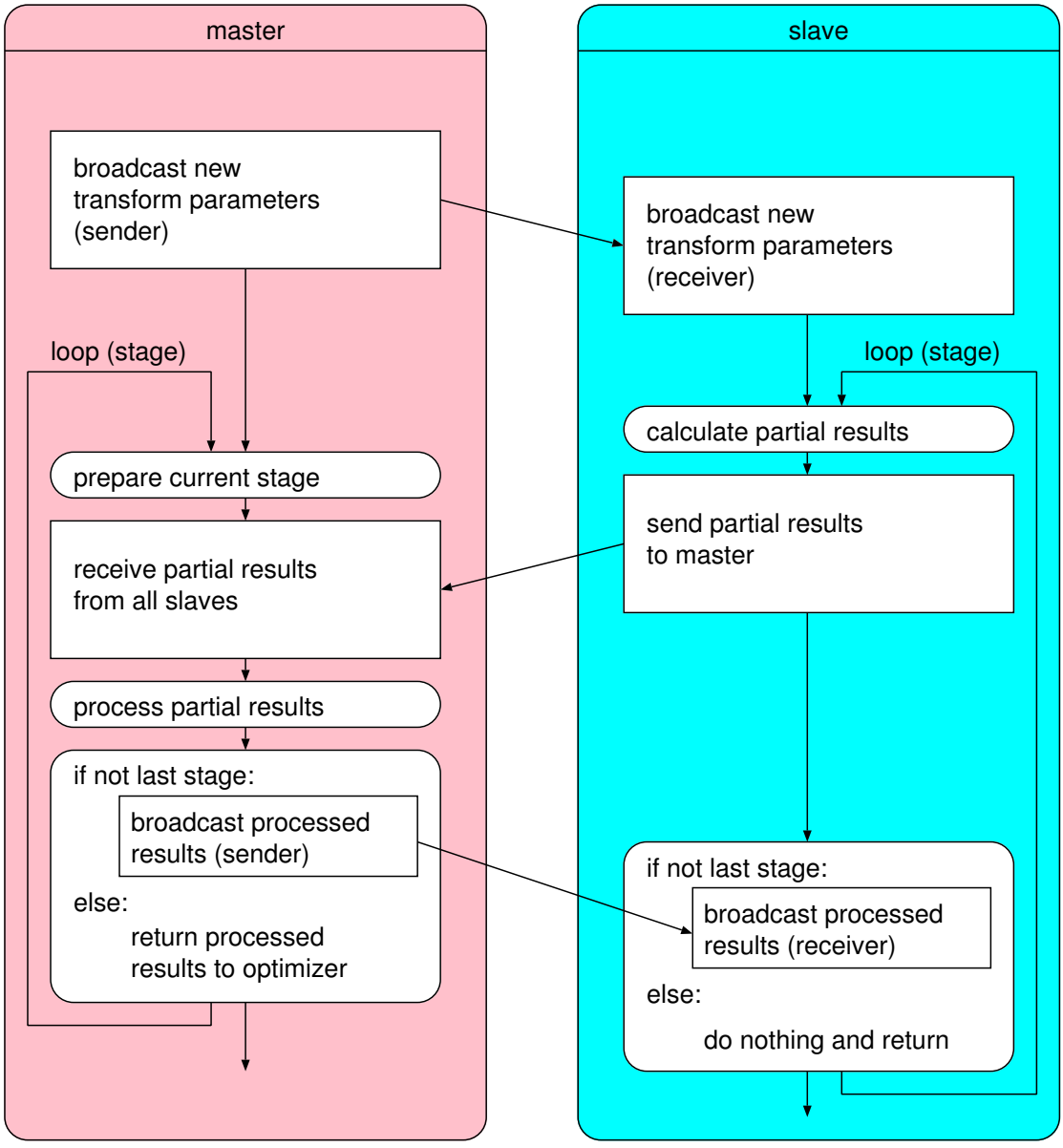



Figure 6.4: Metric value calculation.

framework it has to be called explicitly by the user for the slave processes.

The fixed image is subdivided into slices along the outermost image dimension[5], as illustrated in Figure 6.5(a). This eases the implementation over a subdivision along multiple axis, which can be rather complicated, since any number of slaves (and hence image parts) has to be supported (see Figure 6.5(b)).

In case a gradient image is required, this is computed in the slave's `Initialize()` method as well. Computing it parallely in all slaves takes approximately the same amount of time as calculating it once in the master node, but avoids the overhead incurred by sending it to the slaves afterwards.

### Optimization

For the slaves, the optimization phase is implemented within the `StartSlave()` method, which has to be explicitly called by the user. After the slaves have been started, they enter a loop that waits for commands from the master.

During the optimization process, the optimizer repeatedly (at least once per iteration) calls the master's

```
GetValue(TransformParameters parameters)
```

method[6].

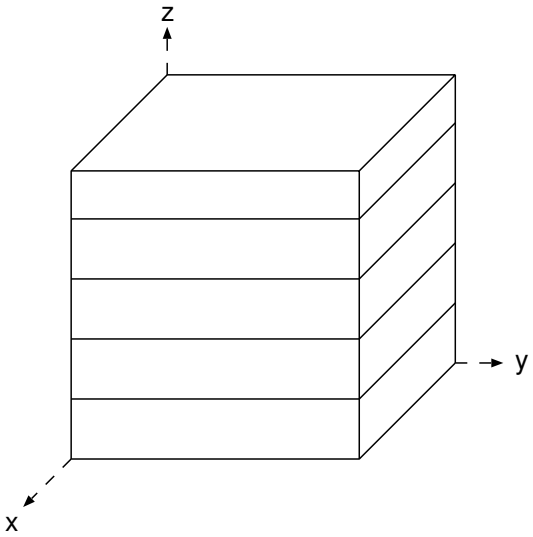
The master now broadcasts a command which informs the slaves that a new metric value has to be calculated. Then the new transform parameters are transmitted. At this point, the master as well as the slave enter a loop that accounts for the possibility that the parallel computation consists of several stages. These loops look as follows:

slave:

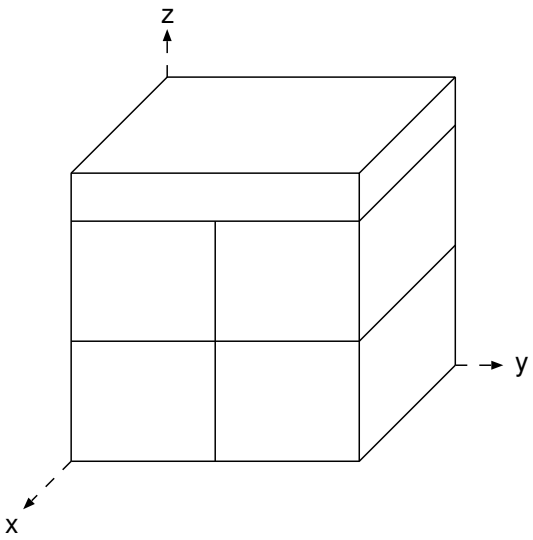
```
for (int i = 0; i < m_NumberOfStages; i++) {
    this->DistributedGetValue(region,
        m_DistributionController->GetLocalProcessId(), i);
    this->IntermediateResultsAllReduce(i)
}
```

[5]The $y$-dimension in 2D and the $z$-dimension in 3D

[6]For simplicity, only the procedure for calculating a metric value is discussed. The procedure for calculating derivatives is analogously and the involved functions are discussed in Section 6.2.3.




(a) slicing along the outermost dimension




(b) splitting along mutliple dimensions

Figure 6.5:

master:

```
for (int i = 0; i < m_NumberOfStages; i++) {
    this->BeforeDistributedGetValue(i);
    this->IntermediateResultsAllReduce(i);
}
```

with the method `IntermediateResultsAllReduce(int stage)`:

```
if (this->Master() ) {
    this->ReceiveIntermediateResultsFromSlaves(stage);
    this->AfterDistributedGetValue(stage);
    this->SendProcessedIntermediateResultsToSlaves(stage);
} else { // slave
    this->SendIntermediateResultsToMaster(stage);
    this->ReceiveProcessedIntermediateResultsFromMaster(stage);
}
```

The slaves first calculate the intermediate results for the current stage by invoking their `DistributedGetValue()` method and then send them to the master. To ease the send process, all intermediate results are transmitted as an array of type double. The master first receives the intermediate results from all the slaves, then processes them and finally redistributes the processed results to the slaves, if the current stage is not yet the last one.

The `IntermediateResultsAllReduce()` method was introduced, because in some cases this operation can be performed more efficiently than shown here. If a more efficient solution exists, a derived metric class can realize it by overriding `IntermediateResultsAllReduce()`.

After exiting the loops, the final metric value is stored on the metric master. The master's `GetValue()` function now returns this value to the registration framework. The slave, on the other hand, waits for a new command to be processed.

Above steps are repeated for each iteration during the optimization process. Eventually, the optimizer reaches one of its stop conditions and the optimization process - and the registration run - terminates. At that time, the slaves are waiting for another command to be processed. Therefore, an exit command is sent to all the slaves which now terminate.

### 6.2.3 Creating new Metric Functions

The basic idea to create a new metric function is to derive a specialized class from the `DistributedImageToImageMetric`. However, there are two

strategies to calculate metric derivatives: Based on finite differences or based on the Jacobian[7] of the transform.

When using the finite difference method, the derivatives are calculated by evaluating the metric function at slightly displaced positions. Thus all calculations base on the `GetValue()` method and no further operations have to be parallelized.

For some metric functions, it is possible to calculate the derivative based on analytical considerations using the Jacobian of the transform and the gradient image of the moving image[8]. This allows to calculate the derivative by traversing the fixed image only once, which results in an enormous speedup compared to the finite difference methods, which repeatedly calls the `GetValue()` method. Since this derivative calculation does not build up on the `GetValue()` method, two more parallelized methods have been introduced. These methods, which are called `GetDerivative()` and `GetValueAndDerivative()`[9] work analogously to the `GetValue()` method that was described before.

To account for the two strategies for derivative calculation, two more classes have been introduced, which are both derived from `DistributedImageToImageMetric`:

- `DistributedImageToImageMetricFDDerivatives`
- `DistributedImageToImageMetricSpecializedDerivatives`

The complete class diagram for the distributed metric framework thus looks as shown in Figure 6.6.

Rather than directly derive the new metric class from `DistributedImageToImageMetric`, it should be derived from any of the latter two classes, depending on how the derivatives are going to be calculated.

In any case, the following three pure virtual methods have to be overridden:

- `DistributedGetValue()` (slave)
- `BeforeDistributedGetValue()` (master)
- `AfterDistributedGetValue()` (master)

[7] A matrix containing the first order derivatives at a given position

[8] To discuss the mathematical details is beyond the scope of this report

[9] Often both, the value and the derivative are required. It is usually sufficient to traverse the fixed image once if they are calculated in the same method.


Figure 6.6: Class diagram for the distributed metric modules. New metric functions should either be derived from `DistributedImageToImageMetricFDDerivatives` (for derivative calculation based on finite differences) or from `DistributedImageToImageMetricSpecializedDerivatives` (for derivative calculation by a user defined method).

The core method is the `DistributedGetValue()` method, which calculates the partial results for each stage[10].

The partial results have to be stored in the class member `m_IntermediateResults`, which is an array of type `double`. The first array index is used by the master to identify the process. For the slaves, and therefore within the `DistributedGetValue()` method, this index is always zero. The second array index denotes the stage and the third index is used to identify the n-th value of the intermediate result for the current stage[11]. The number of pixels considered during the calculation of the partial result have to be stored in the `m_NumberOfPixels` member variable.

The method `BeforeDistributedGetValue()` allows to perform some steps that prepare the master to receiving the partial results. Often, no special steps have to be carried out. In this case, the method can just consist of an empty implementation.

The method `AfterDistributedGetValue()` finally processes the partial results provided by the slaves. In the final stage, the result of the processing step has to be stored in the `m_Measure` member variable. It is the value stored in this variable, which is returned to the registration framework as the value of the metric function.

For many metric functions these are all the steps that have to be done to make them available to the distributed framework. However, there are some further cases that have to be considered. If a metric is derived from the `DistributedImageToImageMetricSpecializedDerivatives` class, it has further to implement the `DistributedGetDerivative()`, the `BeforeDistributedGetDerivative()`, the `AfterDistributedGetDerivative()`, and optionally the `DistributedGetValueAndDerivative()` method.

As mentioned earlier, the `IntermediateResultsAllReduce()` method can be overridden in order to profit from a more efficient method to process and redistribute the intermediate results.

Further, in some cases the predefined format for storage of intermediate results might not be suited. In order to properly treat a custom format for intermediate results, the following communication methods have to be overridden:

- `SendIntermediateResultsToMaster()`
- `ReceiveIntermediateResultsFromSlaves()`

[10] usually only one stage is involved, however sometimes it might make sense to calculate the metric value in a multiple stage process

[11] An intermediate result can consist of several values, that are transmitted as arrays. The third index points to the n-th value inside this array

- `SendProcessedIntermediateResultsToSlaves()`
- `ReceiveProcessedIntermediateResultsFromMaster()`

### 6.2.4 Implemented Metric Functions

Based on above guidelines, the following metric functions have been implemented:

1. `DistributedMeanSquaresImageToImageMetric`
2. `DistributedMeanSquaresImageToImageMetric2`
3. `DistributedHistogramImageToImageMetric` (abstract)
4. `DistributedMeanSquaresHistogramImageToImageMetric`
5. `DistributedMutualInformationHistogramImageToImageMetric`
6. `MPIDistributedHistogramImageToImageMetric` (abstract)
7. `MPIDistributedMutualInformationHistogramImageToImageMetric`

The first two metric functions both implement a mean squares[12] measure. While the first metric calculates derivatives based on finite differences, the second one uses a more efficient method based on a gradient image and the Jacobian of the transform.

The third metric serves as an abstract superclass for metric functions that operate on the joint histogram of the two images. The `DistributedMeanSquaresHistogramImageToImageMetric` as well as the `MutualInformationHistogramImageToImageMetric` directly build upon this superclass. As indicated by their names, they define a mean squares measure and a measure based on mutual information, respectively[13]. The mutual information metric is applicable in inter- as well as intra-model applications.

The histogram based metric functions incur a large communication overhead, as will be shown in Section 7.5.3. In the `MPIDistributedHistogramImageToImageMetric`, the

[12]Squared error or the pixel intensities divided by the number of pixels considered in order to normalize the value.

[13]There are two reasons that legitimate the existence of this third mean squares metric: First, it often requires a lower number of multiplications than the other approaches. Second, since in a histogram several intensity values can be subsumed into one bin, the result is not necessarily the same as for the other approaches.

`IntermediateResultsAllReduce()` method was overridden, which allowed to improve the situation. The `MPIDistributedMutualInformationHistogramImageToImageMetric` derives from this modified superclass. Since these classes make direct use of MPI routines, the prefix `MPI` has been added.

## 6.3 Communication Subsystem

### 6.3.1 Concepts

ITK does not provide any interprocess communication means[14]. As seen in Section 2.2, there exist several libraries designed for parallel computing in cluster environments. Dependent on environment and application these libraries show different advantages and disadvantages[15]. Therefore an abstract communication layer was introduced, which defines an interface that is independent from the underlying communication library. Any interprocess communication within the distributed framework is carried out through this abstract communication layer. This is illustrated in Figure 6.7.

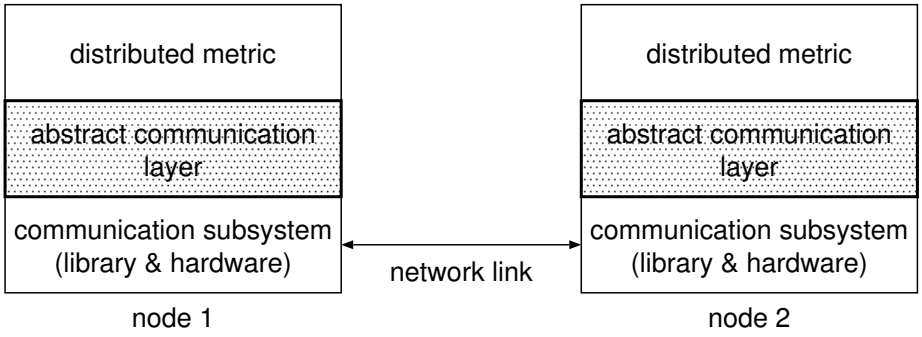



Figure 6.7: Any communications operations are carried out through an abstract communication layer.

The communication interface mainly consists of a (small) subset of the functionality defined in the MPI-1 standard [41]. MPI exhibits four basic design principles:

[14]This is the state at the time of this writing.

[15]Most MPI implementations, for example, have limited support for heterogeneous networks or dynamic process creation. Better support can be found in PVM. However, MPI is a standard while PVM is developed by a single institution.

**Communication operations:** Functions for point-to-point and collective communication.

**Data types:** The data types of messages are known. This allows to compensate for different architectures (e.g. big endian versus little endian) of the participating nodes.

**Communicators:** Messages can only be sent within communicators. A communicator consists of an identifier and a group of processes. The communicator concept allows to develop modular programs (and libraries) using MPI.

**Topologies:** To optimize communication paths, the underlying topology is considered by MPI.

From these four, the first three concepts were integrated into the communication layer.

For all the basic data types supported by C++, communication operations for point-to-point communication as well as for broadcasting have been defined. All of them assume that the message buffer can be reused after returning from the send call. Whether this is achieved by buffering or by waiting for the message to be delivered is left to the implementation of the interface. Besides the communication functionality for basic data types, operations to transmit image data and meta data are provided. The necessary data type conversions have to be done by the specific implementation of the interface.

In order to support modular programming, concepts to group processes and to form communication entities have been incorporated into the communication layer. There further exist methods to initialize the whole communication subsystem and to cleanly shut it down.

### 6.3.2 Implementation

The abstract communication layer is realized by a pure virtual class named `RegistrationCommunicator`. The `RegistrationCommunicator` defines a communicator in the sense of MPI, that is, messages sent within the communicator can only be received by the group of processes associated to it. The `RegistrationCommunicator` exhibits different communication methods for point-to-point communication, broadcasting and image transmission. Methods used to send messages of basic data types are overloaded for all C++ standard data types, hiding the conversion to data types of the underlying communication system from the user.

In order to realize interprocess communication, an implementation has to be provided for this abstract interface. Based on the given hardware and possibly a communication library, a class can be derived from the abstract interface, as illustrated in Figure 6.8. At the moment, only one specialization, based on MPI and called `MPIRegistrationCommunicator`, exists. Since MPI implementations exist for many different architectures this class is a quite generic implementation that fits the needs of most applications.

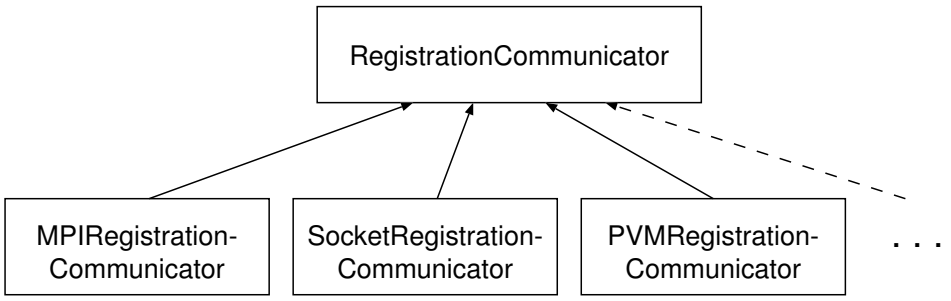



Figure 6.8: The abstract superclass (`RegistrationCommunicator`) allows to derive different specialized communicator classes.

## 6.4 Caching

### 6.4.1 Concepts

During initialization, each slave receives the whole moving image, as explained in Section 6.2.1. Typically it however works on a small part[16] of the fixed image only. This part is mapped onto the moving image by a iteration dependent transform. Usually the size of the transformed region is in the order of the size of the fixed image region and it typically does not alter very much from iteration to iteration as depicted in Figure 6.9. Therefore, data access can generally be considered local for succeeding iterations and most regions of the moving image will never be accessed. Consequently there is little sense in keeping the whole moving image in main memory, particularly if memory is a critical resource. The local data access characteristics encourage the use of caching to make optimal use of the available memory. Therefore, a `CacheImage` class has been introduced that takes care of the necessary data management, i.e. that decides which parts of the image are kept in main memory and which parts are stored on disk.

[16]The more processors involved the smaller the fixed image part on a single slave.

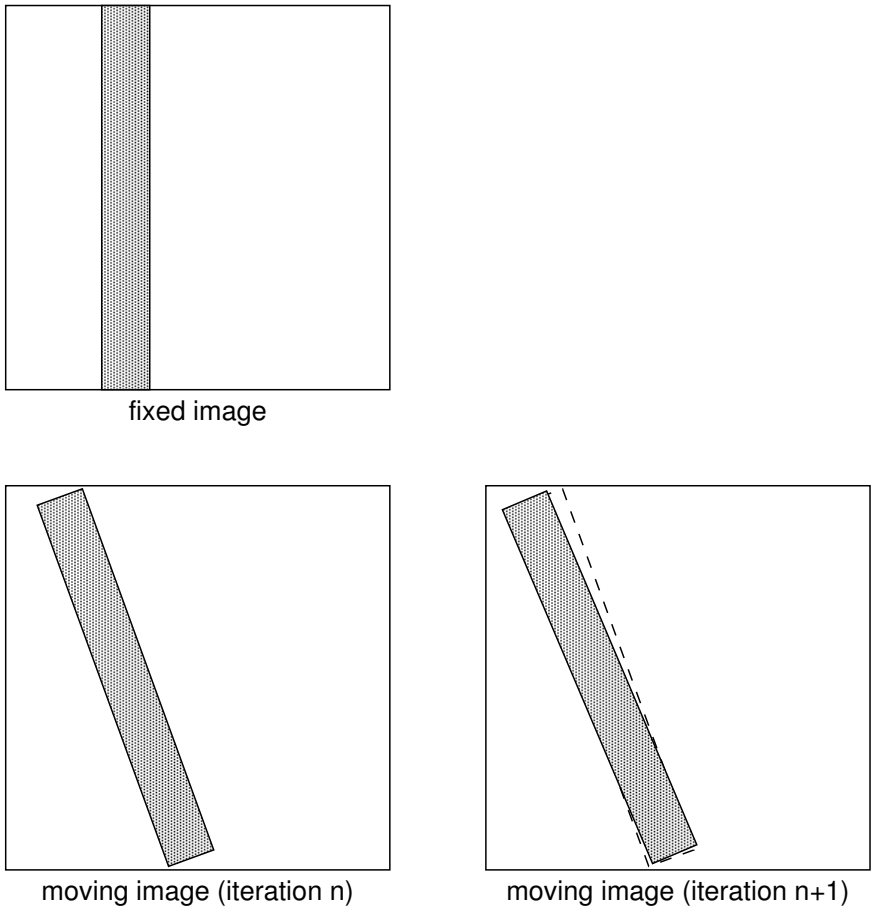



Figure 6.9: The region of fixed image and its corresponding region of the moving image after applying the transform for iteration n are illustrated on the left side. On the right side, the transformed fixed image region for iteration n + 1 (dark) as well as the region for iteration n (dashed) are indicated. Since the transform will only slightly change from iteration n to iteration n + 1 the region on the moving image will only slightly change, too.

In order to efficiently move data portions from main memory to disk and vice versa, data is preferably stored in a block structure rather than line by line as this is done in ITK. This is depicted in Figure 6.10. Therefore, the `CacheImage` organizes its data in blocks. Whenever a pixel is accessed, it is checked whether the corresponding block resides in main memory. If not, the new block is loaded from disk and the block that has not been accessed for the longest time is released from main memory.

To avoid occupying large amounts of swap space that could be used by other processes, data on disk is stored in a regular file. Memory mapped access is used in order to perform read and write operations on the file.

In case of the moving image, the `CacheImage` is integrated into a `CacheInterpolator`. Three of them exist, in accordance with the three interpolator types (nearest neighbor, linear and bspline) present in ITK. In case of the gradient image, the cache image is directly located in the metric slaves.

The gradient image is large compared to the moving image, because a pixel consists of a floating point value for each derivative. Typically it consumes between 8 and 24 times as much memory as the moving image[17]. Caching this image reduces the memory consumption during the ongoing optimization process. However, the memory peak resulting during the creation of the gradient image can already cause serious problems. A system might run out of physical memory which causes swapping and slows down the creation process. What makes things worse is the fact that there are operating systems[18], that do not allow a process to return freed memory to the operating system. Therefore, the process size will not drop even during optimization, where due to caching only small amounts of memory are used. The unused memory (which is most likely swapped out) reduces the swap space available to other processes. Because of these problems, the memory peak occurring during gradient image generation should be avoided. The streaming concept of ITK allows to create images in pieces. Creating the gradient image block by block and directly writing these blocks to disk avoids the undesirable memory peak.

In memory uncritical problems, the framework performs better without caching. Therefore, it is by default disabled. In order to enable caching, a cached interpolator has to be provided in case of the moving image, and a flag has to be set in case of the gradient image.

[17] A pixel of the moving image is typically represented by one or two bytes and a derivative typically by a `double` value which takes 8 bytes. In 2D there are 2 derivatives yielding 16 bytes, and in 3D there are 3 derivatives yielding 24 bytes per pixel.

[18] among them most variants of UNIX

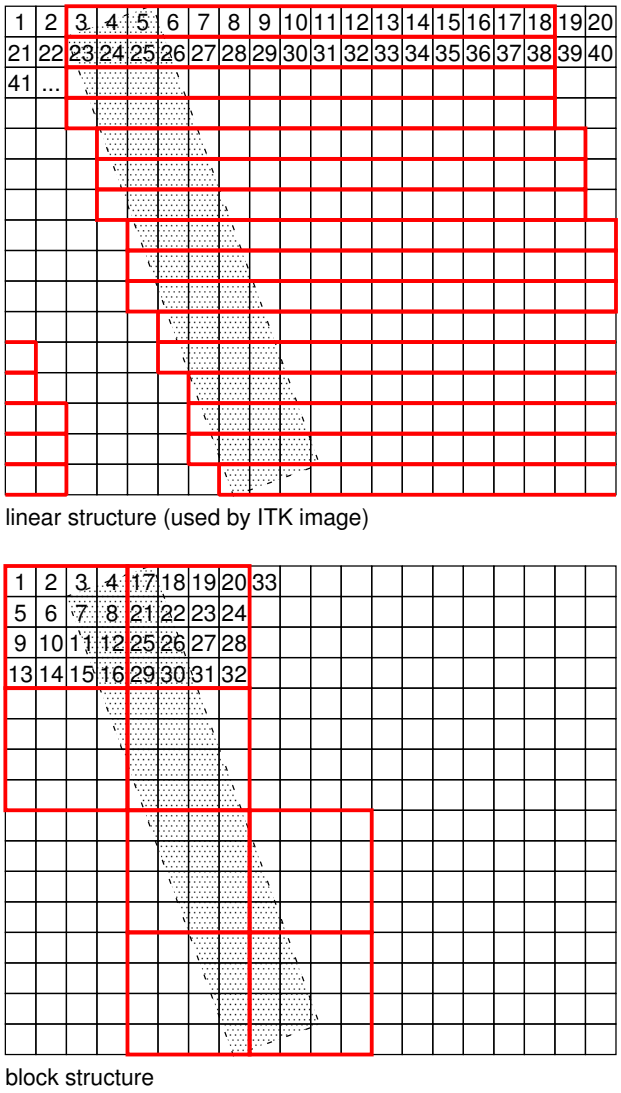



Figure 6.10: **Top:** Linear data structure as it is used by the standard ITK image. Pixels are stored line by line in a single huge array, as indicated by the increasing numbers. When data for the shadowed region should be kept in memory, and it should be possible to move blocks of 16 pixels to and from disk, the area marked with bold lines is affected. **Bottom:** The same is illustrated for an image that stores data in blocks of 4 by 4 pixels. In the first case, $16 \times 16 = 256$ pixels are affected, in the second case only $8 \times 16 = 128$ pixels. In reality, both, image as well as block size is typically much larger than in this illustration.

### 6.4.2 Implementation

Pixels that do not belong to the fixed image are accessed after applying a transform. Therefore, any pixel values are retrieved by the image's `GetPixel()` method and no iterators are used. This fact considerable eases the implementation of an image that stores data in blocks, since no complicated iterators that need to traverse these blocks have to be implemented. Rather, any index calculation can be carried out within the `GetPixel()` method. Index calculations turn out to be more complex than for usual image data structures, despite the use of index tables to speed them up. Consequently pixel access performs worse when `CacheImage`s are used.

The block handling is also realized in the `GetPixel()` method. The whole data is written into an ordinary file, block by block. Memory mapped file access allows to efficiently move blocks to and remove them from main memory. A `FileMapper` class wraps the mapping process in order to hide the differences of memory mapped file access in Windows and Unix. Whenever a pixel is accessed and the corresponding block does not reside in memory, the respective region of the file is mapped to a new view. The view of the block that has not been accessed for the longest time is unmapped.

Caching of the moving image is realized by new interpolator classes, namely

- `NearestNeighborCacheInterpolateImageFunction`
- `LinearCacheInterpolateImageFunction`
- `BSplineCacheInterpolateImageFunction`

Two main reasons led to the decision that the cache image should be created within the interpolator and not within the metric class. A first reason is, that the bspline interpolator works on a coefficient image, rather than directly on the moving image. This coefficient image is the one that has to be cached, and it has to be created inside of the interpolator. Creating the cached image inside the nearest neighbor and the linear interpolator allows to treat all the cases almost identically. A second reason that speaks in favor of wrapping the cache image inside the interpolator are several type conflicts that would have had to be resolved when storing the image in the metric.

Caching of the gradient image is realized inside the metric class. In order to avoid the undesired memory peak, the gradient image is created block by block and these blocks are directly written disk. Since the gradient filter applied in ITK image metrics does not support streaming it was replaced by a combination of a smoothing filter (`itk::DiscreteGaussianImageFilter`)

and a simpler gradient filter (`itk::GradientImageFilter`) that is streaming capable.

# Chapter 7

# Evaluation

When evaluating a parallel realization of an algorithm, several measures are of interest. The measures used to assess the methods developed throughout this project are briefly introduced within the following section. Then a software library (MPE) that allows the examination of performance parameters within a parallel program is presented followed by the discussion of the methodology, the scenarios used for, and the results of the evaluation of the distributed registration framework.

## 7.1 Measures

Usually, people are interested in the reduction of computing time in the first place. Recall from Section 2.2, that reduction in computing time is defined as parallel speedup

$$S_N = T_S/T_N \tag{7.1}$$

and that another, closely related measure called parallel efficiency

$$E_N = S_N/N \tag{7.2}$$

has been introduced. In these formulas, $N$ denotes the number of processors, $T_N$ the execution time of the parallel program on $N$ processors, and $T_S$ the execution time of the best sequential algorithm on one processor. Note that it is generally difficult to define $T_S$, because of several reasons. Which algorithm performs best can depend on several factors, the best algorithm can change over time (since new, better algorithms are developed) and it is not clear on which hardware the time is measured. Therefore, in practice $T_S$

usually denotes the execution time of a good sequential algorithm[1].

The parallel efficiency usually depends on the number of processors as well as on the problem size $W$ (using the definition of [42](p. 129), $W$ becomes basically equal to $T_S$[2]). Due to non parallelizable code and an increasing communication as well as synchronization overhead, the parallel efficiency decreases with an increasing number of processors. On the other hand, an increasing problem size often increases efficiency, since the calculation time increases relative to the overhead. A system that allows to keep the efficiency at a fixed value for an increasing number of processors, if the problem size is adjusted accordingly, is called *scalable*. The function that defines how $W$ has to be increased with respect to $N$ in order to keep the efficiency fixed is called *isoefficiency function*. To look at this phenomena more closely, let us define the *total overhead*

$$T_o = N \cdot T_N - W, \tag{7.3}$$

which subsumes any work that is not incurred by the fastest known sequential algorithm. As seen before, the total overhead is a function of the problem size as well as the number of processors: $T_o(N, W)$. Using equation 7.3, the parallel execution time can be written as a function of the problem size, the number of processes and the total overhead function:

$$T_N = \frac{W + T_o(W, N)}{N}$$

Substituting $T_N$ in equations 7.1 and 7.2 results in the following expression for the parallel efficiency:

$$E_N = \frac{1}{1 + T_o(N, W)/W} \tag{7.4}$$

From equation 7.4 it becomes apparent, that the efficiency does not change if $T_o(N, W)/W$ is constant. Replacing $E_N$ in 7.4 with the desired value $E$ for the efficiency and reformulating the equation yields:

$$W = \frac{E}{1 - E} T_o(N, W) \tag{7.5}$$

[1]For the evaluation of the parallel registration framework, $T_S$ was the time used by an equivalent program composed of standard ITK modules.

[2]The authors of [42] state, that problem size should be proportional to the total number of basic computational steps involved, which is proportional to $T_S$. If the unit to measure $W$ is normalized accordingly, $W$ thus becomes equal to $T_S$. This is arguable when factors like disk swapping slow down the computation.

This is an implicit representation of the *isoefficiency function*. Often equation 7.5 can be reformulated such that $W$ is expressed as a function of $N$, which results in an explicit representation of the isoefficiency function.

The isoefficiency function defines the ease with which the efficiency of a parallel system can be kept fixed by increasing the problem size. It can therefore be considered as a measure for *scalability*. A "small" isoefficiency function denotes that a little increase of the problem size is sufficient in order to maintain the efficiency for an increasing number of processors, which means that the algorithm scales well.

With a growing problem size, memory requirements usually increase. Therefore, memory requirements have to be considered carefully when analyzing scalability. Often it is assumed that the amount of available memory grows equally to the number of processors (this is quite reasonable for cluster computing). It is then possible to change above definition of a scalable system by adding a memory constraint: A system is *scalable* if the efficiency can be kept fixed for a growing number of processes by increasing the problem size such that the memory requirement per processor is constant (or decreasing).

## 7.2 The Multi Processing Environment (MPE) Library

The *Multi Processing Environment* (MPE) library [43] provides features for graphics output, debugging and performance analysis of MPI programs. It is an open source library developed at the Argonne National Laboratory. Here, only a brief introduction to the functionality for performance analysis is given.

The performance analysis using MPE is based on logging of events using highly accurate time stamps. There is an automatic mode that encloses every MPI routine with calls to the MPE library which in turn creates an entry in a log file for the beginning and the end of the routine. Further, it is possible to create user defined events by inserting according calls to the MPE library into the code. MPE gathers all the information logged by the different processes, corrects the possible misalignment and drift of clocks and finally creates a single log file.

Typically a log file generated by MPE is viewed by some graphical tool that displays all the logged events as time-lines. The most popular viewer is *Jumpshot*, which comes with the MPE distribution. An example of a time-line representation of an MPE log file using *Jumpshot* is shown in Figure 7.1.

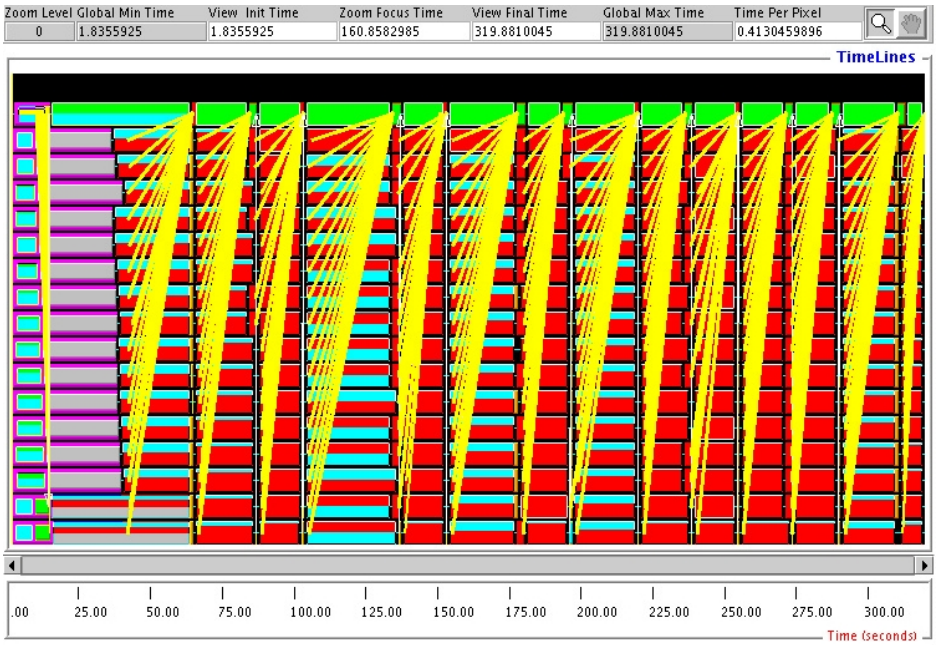



Figure 7.1: A screenshot of the Jumpshot output of an MPE log file. The rows represent the single processes and the x-direction the elapsed time since process start. Colored boxes depict states (such as computing time, broadcast operations, etc.) and arrows indicate point to point communications.

Inserting timing statements into a program obviously incurs some overhead. In our case, this overhead is small with respect to computing times and can safely be neglected during performance analysis. However, when developing an application based on the parallel registration framework this overhead is not desired. MPE logging is only in effect if MPE is linked to the program. Since this decision is made at link time, no overhead occurs if MPE is not linked. In order to disable custom calls to MPE routines, a compiler directive has been defined that allows to switch these calls on and off at compile time and which can be set using CMake[3] during the configuration of the parallel registration framework.

## 7.3 Methodology

The goal of the evaluation was to examine the developed methods with respect to speedup, efficiency and scalability. When analyzing the scalability, memory issues were taken into account.

These measures were acquired for three different sample scenarios. A sample scenario consists of a combination of metric function, optimizer, transform and interpolator with their respective parameters.

Speedup and efficiency can be quantified for a fixed problem size by measuring the execution times of the parallel program as well as of the "best" sequential algorithm. The "best" sequential algorithm required to get the time $T_S$ is considered to be the equivalent algorithm implemented using standard ITK modules. $T_S$ is measured on a single node of the workstation cluster. Execution time is measured as wall-clock time, which is the time (such as the system time) elapsed between the process start and its termination. Using wall-clock time is common practice when evaluating parallel performance.

In order to evaluate the scalability, it is necessary to adjust the problem size and to closely watch the memory consumption, as explained in Section 7.1. Therefore, for each scenario memory usage as well as speedup has to be examined for an increasing problem size. The increase in problem size can be quantified by measuring the increase of $T_S$ [4]. To modify the problem size, obviously the size of the input images can be changed. For larger images more pixels have to be compared and consequently problem size increases. An other option to get a larger problem size is to ask for higher accuracy. This can be achieved by adjusting the optimizer parameters that define the

[3] A makefile generation tool

[4] This is arguable in cases where the sequential program is slowed down due to disk swapping.

stop conditions. Basically, a higher accuracy asks for a larger number of iterations and consequently problem size increases.

Obviously it would be interesting to gain a deeper insight into the performance parameters than by just measuring the total execution time and memory consumption. Using the `MPIRegistrationCommunicator` as the communication interface (see Section 6.3 for details) a program based on the parallel registration framework becomes an MPI application which allows to use MPE to gain such an insight. By automatic profiling, MPE allows to measure the overhead of the MPI routines (which is mainly communication overhead). Customized MPE calls further permit to measure time spent in particular parts of the code, such as the non parallelizable fraction $f$, the initialization phase, work carried out by the optimizer or time spent by the processes waiting for each other. Figure 7.2 shows for the case of the optimization phase, how the parallel algorithm can be subdivided into different parts.

During the evaluation, the following values have been analyzed using MPE:

- time spent in the initialization phase
- time spent to send the moving and the fixed image
- time spent by the optimizer
- time spent to calculate the intermediate values and derivatives
- idle time of the slaves (waiting for new parameters)

Unlike the measures described in Section 7.1, these values do not provide the possibility to compare the parallel framework with other parallel algorithms. However, they give insight into strengths and deficiencies of the parallel framework and allow to derive conclusions on how to further improve it.

### Measurement Conditions

Experiments were carried out on Unix workstations at ETH Zurich (Tardis cluster) as well as on a Windows cluster at Nizhny Novgorod State University (NNSU). The Tardis cluster consists of up to 78 identical Sun-Blade-100 workstations with 512 MB RAM and about 30 Sun-Blade-1500 workstations. Experiments should only be carried out on the Sun-Blade-100 machines. They are connected by a 100 Mbit Ethernet.

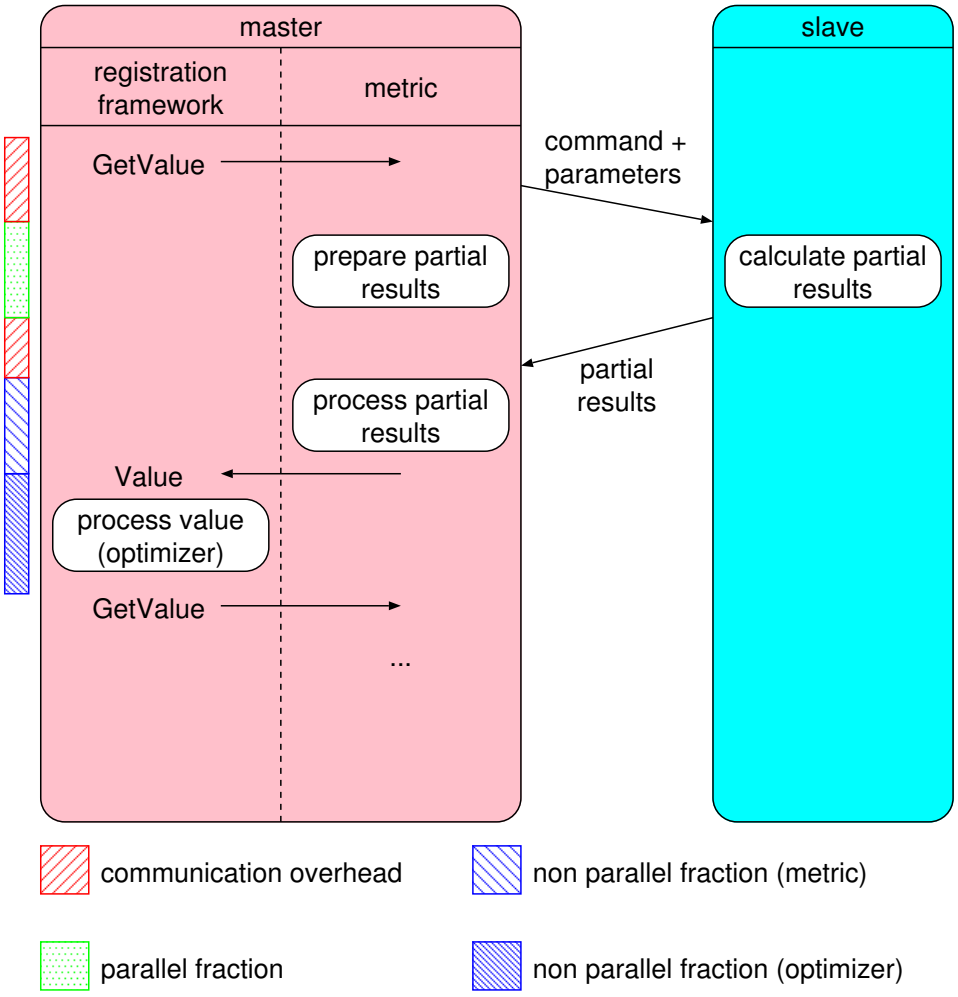

master
registration framework
metric
GetValue
prepare partial results
process partial results
Value
process value (optimizer)
GetValue
...
command + parameters
partial results
slave
calculate partial results
communication overhead
non parallel fraction (metric)
parallel fraction
non parallel fraction (optimizer)

Figure 7.2:

The Windows cluster at NNSU consists of 2 servers with 4 Pentium-III 700 MHz CPUs each, 12 servers with 2 Pentium-III 1 GHz CPUs each, and 12 workstations with a single Pentium-IV 1.3 GHz CPU. The different machines have 512, 256 and 128 MB of RAM, respectively. The server machines are connected with a 1000 Mbit, and the workstations with a 100 Mbit Ethernet network.

Any experiments for speedup and efficiency were carried out at least three times. The workstations at ETH were not guaranteed to be free of interference originating from other users. Further, any jobs had to be run with lowest priority (nice 19). This caused a significant load imbalance. Since the parallel framework does not provide dynamic load balancing, only the result with the lowest interference (i.e. the experiment with the lowest wall-clock time) was taken.

On the windows cluster at NNSU eight Pentium-III 1 GHz machines (i.e. 16 processors) could be reserved to carry out the experiments. Since they were supposed to be free of interfering load[5], averages times were taken for the results acquired by the experiments on this system.

The master-slave architecture of the parallel registration framework causes one process (the master process) to be idle most of the time. This process can be run on the same processor as one of the slave processes, as illustrated in Figure 7.3 (a). In this setup, $N+1$ process*es* (1 master and $N$ slaves) are run on $N$ process*ors*.

Unfortunately, it was cumbersome to automatically run a large number of experiments like that at the Unix cluster at ETH. Therefore, the $N+1$ processes were run on $N+1$ processors as illustrated in Figure 7.3 (b). Speedup and efficiency values were, however, calculated with respect to $N$[6]. Some specially set up experiments proved, that no change in performance can be observed between the two configurations.

The Windows cluster also caused difficulties to run experiments as shown in Figure 7.3, since each node consists of two CPUs there. The operating system would automatically place the second process on the second CPU. Therefore, only the experiments on 16 processors (with 17 processes) were run in the configuration as shown in Figure 7.3 (a).

During all the experiments, MPE logging was enabled. The overhead with respect to timing is supposed to be negligible. However, the memory

[5]Unfortunately this was not entirely true as will be shown in Section 7.4.2. However, the load was the same for all experiments.

[6]Parallel speedup and efficiency are defined with respect to the number of process*ors*. Strictly speaking, they would therefore have had to be related to $N+1$ when using configuration (b). However the results would have become difficult to interpret and would not have reflected the actual character of the parallelization.

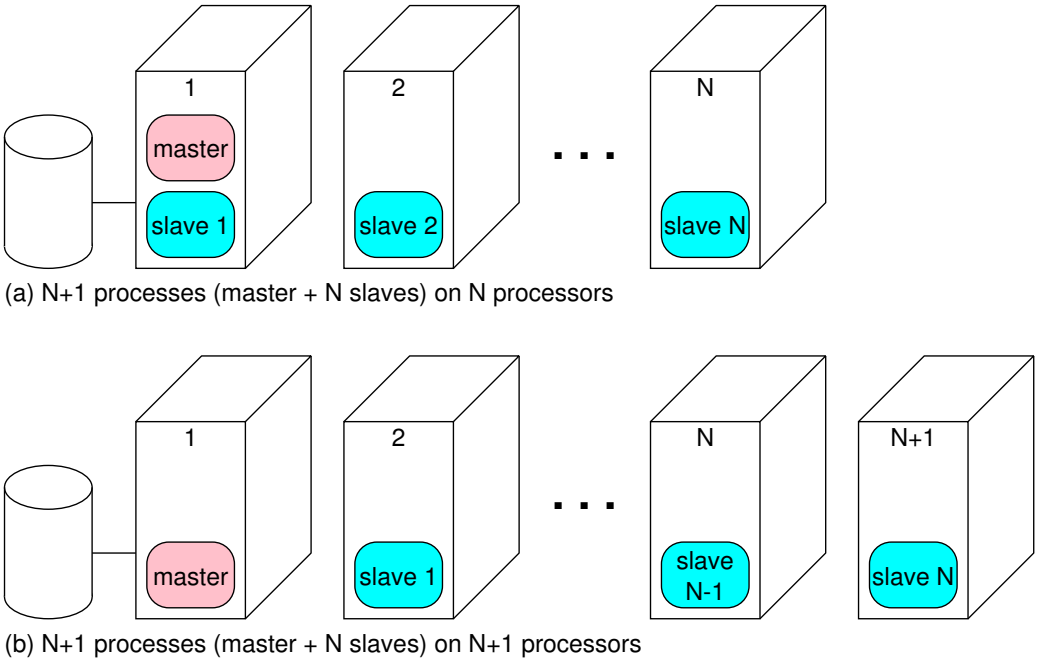



Figure 7.3: The parallel framework can be run in two different configurations. In configuration (a), N slaves sit on N processors. Thereby, one processor is shared between a slave an the master. Since the master is idle most of the time, performance is not affected. In configuration (b), each process sits on its own CPU. Experiments have been carried out with configuration (b) (except of those on 16 processors at NNSU).

measurements might have been influenced by the logging process, since for performance reasons, log files are kept in main memory until process termination.

## 7.4 Sample Scenarios

Three sample scenarios addressing different registration tasks have been created to evaluate the parallel registration framework. These scenarios are briefly presented in the following sections.

### 7.4.1 Three Dimensional Rigid Problem

The first scenario addresses a rigid registration problem in 3 dimensions. The scenario consists of the following components:

- `itk::QuaternionRigidTransform`
- `itk::QuaternionRigidTransformGradientDescentOptimizer`
- `itk::LinearInterpolateImageFunction`
- `itk::MeanSquaresImageToImageMetric` (sequential)
- `DistributedMeanSquaresImageToImageMetric2` (parallel)

Two micro CT images[7] of a mouse femur taken from different points of view serve as the input to the program. The original images have a resolution of $512 \times 512 \times 1006$ pixels encoded with 2 bytes (65536 gray levels). This results in approximately 500 MB per image. For the experiments, the images were downscaled to the following resolutions:

- scale 1: $52 \times 52 \times 101$ pixels (534 KB)
- scale 2: $65 \times 65 \times 127$ pixels (1048 KB)
- scale 4: $82 \times 82 \times 160$ pixels (2102 KB)
- scale 8: $103 \times 103 \times 202$ pixels (4186 KB)
- scale 16: $130 \times 130 \times 254$ pixels (8384 KB)

[7]The images originate from Steven Boyd, PhD, Department of Mechanical and Manufacturing Engineering, University of Calgary, Canada.

- scale 32: $163 \times 163 \times 320$ pixels (16606 KB)
- scale 64: $205 \times 205 \times 403$ pixels (33079 KB)

The most important property of the scenario is the large amount of data when using full resolution. The optimizer requires gradients which are calculated by the metric by a closed form rather than by finite differences. Therefore, a gradient image has to be calculated in the initialization step. This causes a rather large non-parallelizable fraction of the code[8] as well as increased memory requirements because of the large gradient image.

### 7.4.2 Two Dimensional Problem based on Bspline Deformable Transform

The second scenario addresses a simple deformable problem in two dimensions. It consists of the following components:

- `itk::BSplineDeformableTransform`
- `itk::RegularStepGradientDescentOptimizer`
- `itk::BSplineInterpolateImageFunction`
- `itk::MeanSquaresImageToImageMetric` (sequential)
- `DistributedMeanSquaresImageToImageMetric2` (parallel)

The deformation field is defined by a bspline deformable transform. An example of such a deformation field is illustrated in Figure 7.4. For the scenario, grid-points on a grid of size $5 \times 5$ were displaced to well defined positions. Since third order bsplines were used, the total number of transform parameters was $(5 + 3) \times (5 + 3) = 64$.

Two slices of a brain image (one of which was intentionally deformed) taken from the ITK examples serve as the input to the program. The original images have a size of $221 \times 257$ pixels encoded with 1 byte (256 gray-levels). For the experiments, they were upscaled to the following resolutions:

- scale 1: $221 \times 257$ (56 KB)
- scale 2: $313 \times 364$ (112 KB)

[8] The whole gradient image is calculated on all the slaves, which is basically equivalent to calculating it once non-parallelized. This step could be parallelized however (see Section 8).

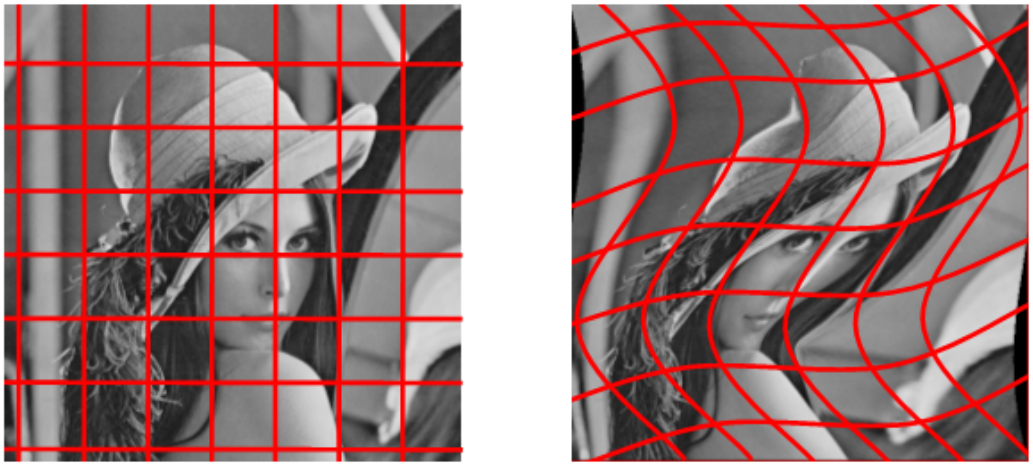


Figure 7.4: The bspline deformable transform allows to define a deformation field by displacing grid-points. The deformation in between the grid points is calculated using bspline interpolation.(image modified from http://www.cs.rpi.edu/courses/spring04/imagereg/lectureBSplines.ppt (29 September 2004))

- scale 4: $442 \times 514$ (222 KB)
- scale 8: $626 \times 727$ (445 KB)
- scale 16: $884 \times 1028$ (888 KB)

Due to the computationally intensive bspline interpolator, it takes rather long to calculate a single metric value (and derivative) despite of the small images. A gradient as well as a coefficient image are required due to the metric and the interpolator, resulting in rather large memory requirements compared to the small input images.

### 7.4.3 Two Dimensional Affine Problem

The third scenario addresses a two dimensional affine problem. It consists of the following components:

- `itk::AffineTransform`
- `itk::OnePlusOneEvolutionaryOptimizer`
- `itk::LinearInterpolateImageFunction`

- `itk::MutualInformationHistogramImageToImageMetric` (sequential)
- `DistributedMutualInformationHistogramImageToImageMetric` (parallel)

The input images were again taken from the ITK examples. They represent slices of a brain and one of them was intentionally deformed. The following resolutions were used for the experiments:

- scale 1: $221 \times 257$ (56 KB)
- scale 2: $313 \times 364$ (112 KB)
- scale 4: $442 \times 514$ (222 KB)
- scale 8: $626 \times 727$ (445 KB)
- scale 16: $884 \times 1028$ (888 KB)

Due to the use of a mutual information based similarity function, the scenario is applicable to inter-modal registration problems. Because of the typically rather noisy character of mutual information based metrics, evolutionary optimization algorithms are supposed to be a good choice. The applied one plus one evolutionary optimizer does not require any gradients to be calculated. Since further, a relatively simple interpolator was used and the images are rather small, the time to calculate one metric value becomes low. However, the optimizer requires a large amount of iterations in order to achieve satisfying results which causes the overall computing time to be large.

## 7.5 Results

The developed parallel registration framework was evaluated in more than 1000 experiments. The following sections try to summarize the results. They can be compared to results reached by other projects addressing image registration by cluster computing. The approach of Chalermwat [14] reached efficiencies of about 85% on 8, between 70% and 80% on 16, between 45% and 60% on 32, and between 20% and 50% on 64 processors. Other projects achieved efficiencies of approximately 70% on 8 and 50% on 14 [25] or of 85% to 90% on 8 and between 60% and 70% on 16 processors [23]. Further, the

results can be compared to the "ideal" speedup of $N$ on $N$ processors[9]. To facilitate this task, a curve that illustrates this "ideal" case has been added to all speedup and efficiency diagrams.

### 7.5.1 3D Rigid Scenario

The three dimensional rigid problem was the one that was evaluated most extensively. The behavior was analyzed with and without caching for small and rather large images[10] (see Section 7.4 for details about the image sizes) and for different numbers of optimizer iterations.

The experiments for scale 1 to 32 have been carried out with 100 iterations. For the experiments with scale 64, both, the Windows machines at NNSU as well as the Unix workstations at ETH run into disk swapping, which was intended in order to analyze the behavior of caching. Therefore, experiments with scale 64 were computed with only 50 iterations. This avoided overly long execution times[11], which do not only make experiments very time consuming but also make the results prone to interference due to other tasks and prevent other users from using expensive resources[12].

The resulting speedup for scales 1 to 64 and 100 iterations (only 50 in case of scale 64) without caching has been summarized in Figure 7.5. Figure 7.6 shows the corresponding efficiencies for these experiments. Note the extraordinary shape of the curves for scale 64 for the experiments at ETH and for scales 16, 32 and 64 for the experiments at NNSU, which are due to the disk swapping caused by the sequential reference program. This phenomena is discussed in more detail in the caching part of this section. The wall-clock times of the sequential program to which the speedup and efficiency values are related are illustrated in Figure 7.7.

To demonstrate the effect of increasing the problem size by increasing the number of iterations (which leads to more accurate results), the experiments for scales 1 to 16 have further been carried out for 200 iterations. The respective results can be found in Figures 7.8 (speedup) and 7.9 (efficiency).

These results show, that for the three dimensional rigid scenario, an increase in problem size due to larger images does only slightly improve parallel

[9]The word "ideal" is used, even though parallelization sometimes allows to reach a higher speedup.

[10]Note that the maximum file size was about 32 MB, which only a 16-th of the original image size

[11]For 50 iterations, the execution time for the sequential program was still in the order of 8 hours on the Windows cluster, and in the order of 10 hours on the Unix machines.

[12]Even when using lowest priority for the jobs, a user will be disturbed due to disk swapping

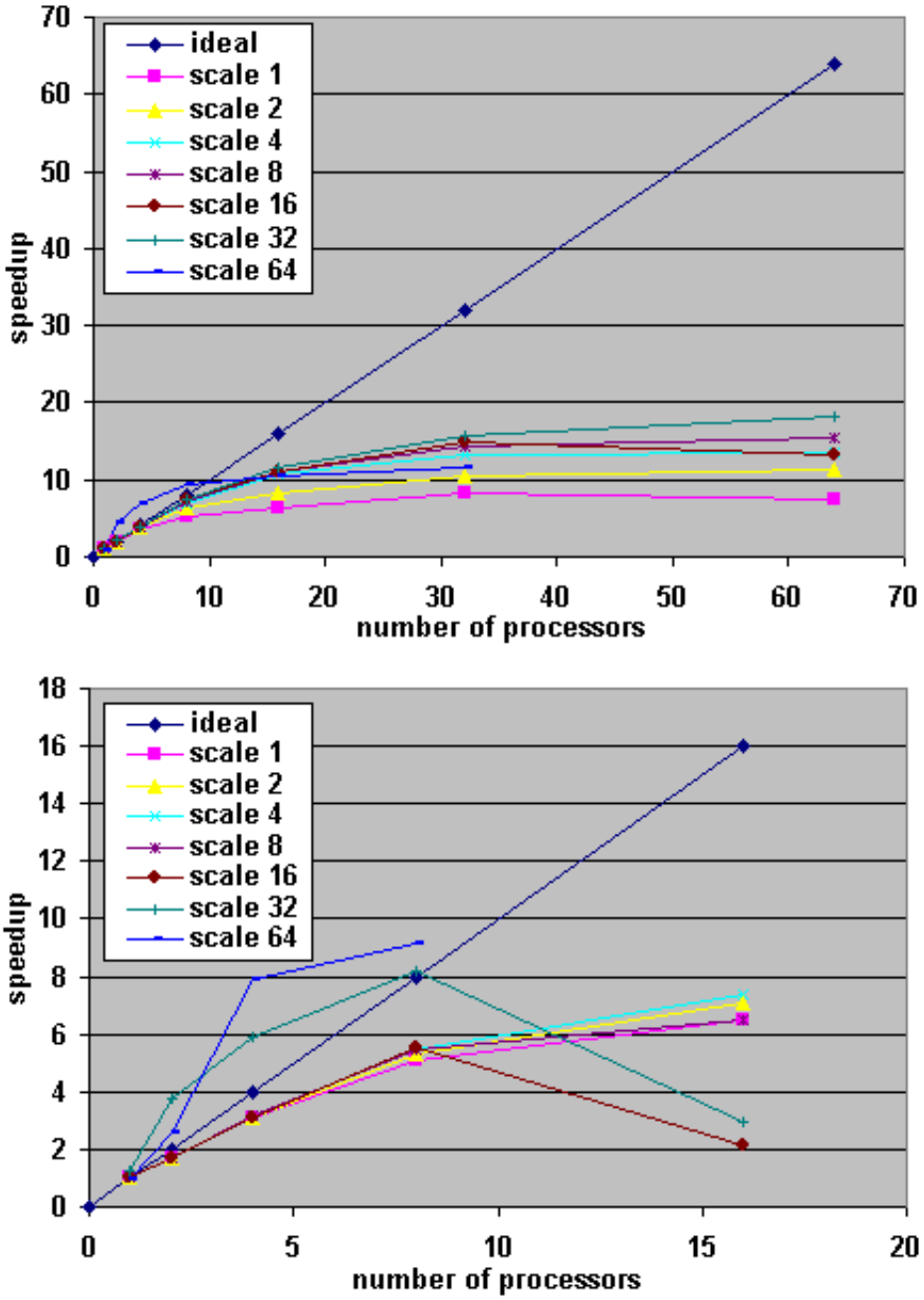



Figure 7.5: Speedup for the experiments at ETH (top) and NNSU (bottom). The results for scale 64 were carried out with only 50 iterations, the other experiments with 100 iterations. The extraordinary shape of the curves for scale 64 (ETH) and 16 to 64 (NNSU) will be explained in the caching part of this section.

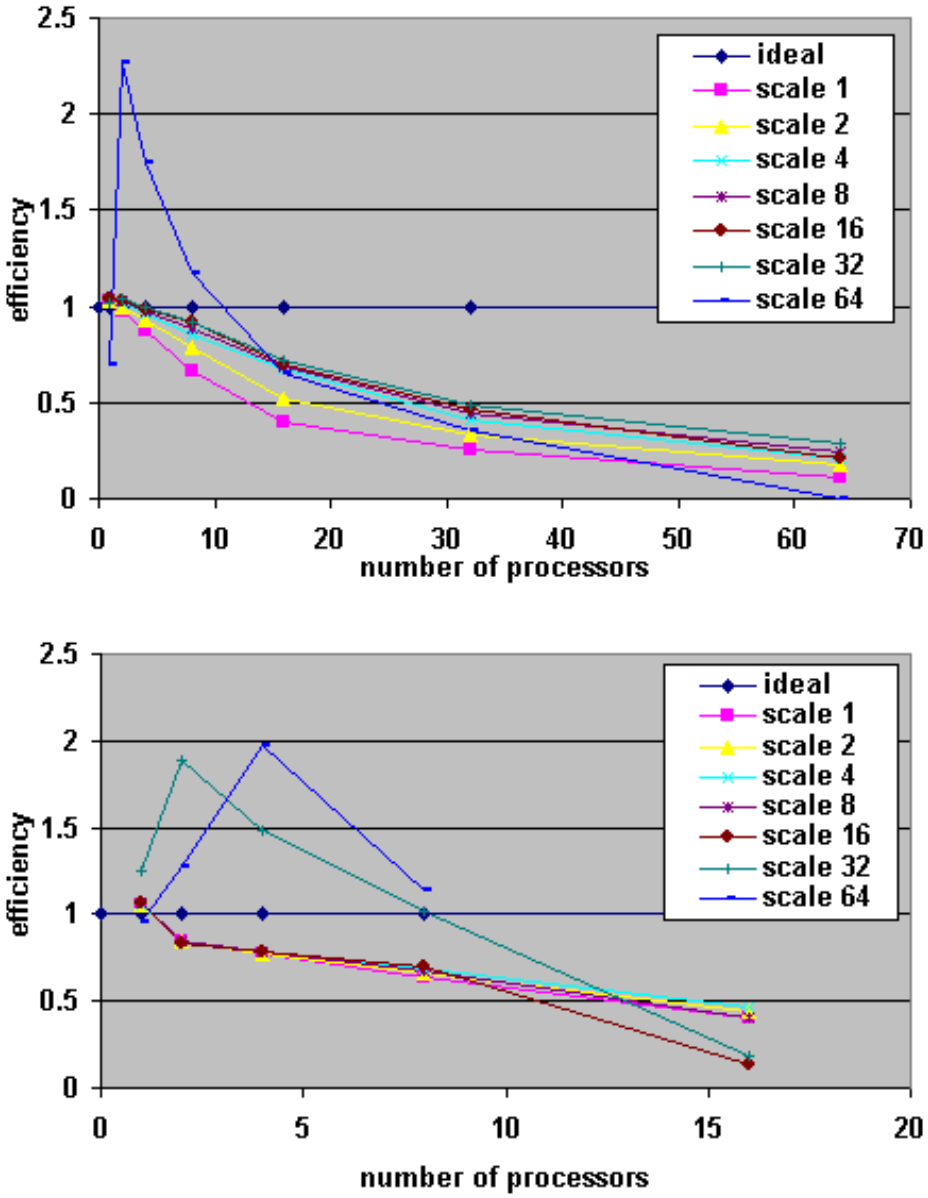



Figure 7.6: Efficiencies for the experiments at ETH (top) and NNSU (bottom). The results for scale 64 were carried out with only 50 iterations, the other experiments with 100 iterations. The extraordinary shape of the curves for scale 64 (ETH) and 16 to 64 (NNSU) will be explained in the caching part of this section.

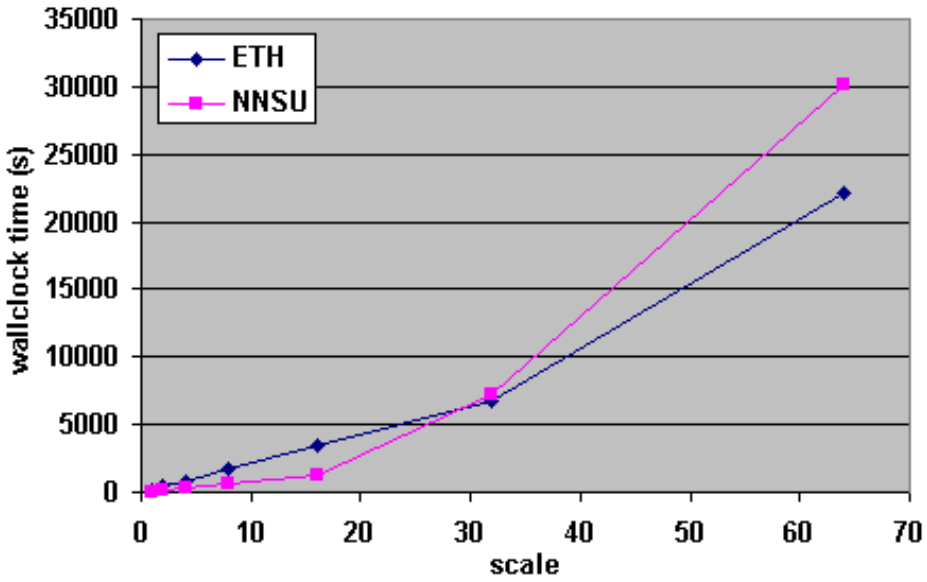



Figure 7.7: The wall-clock times for the sequential program of the three dimensional rigid scenario.

efficiency. Since larger images ask for more memory resources, scalability in the sense of the extended definition, which asks for constant memory consumption (see Section 7.1) is poor. When the problem size is increased by incrementing the number of iterations, memory requirements remain constant. At the same time a significant improvement of parallel efficiency can be observed as illustrated in Figure 7.10. According to the formal definition of Section 7.1, the program can therefore be considered scalable. This conclusion is, however, dangerous. Which number of iterations is reasonable is highly dependent on the problem and the required accuracy[13]. In the three dimensional rigid scenario that was evaluated, the improvement in accuracy stagnated between 200 and 300 iterations.

### Caching

The caching mechanism (see Section 6.4) was introduced to avoid the problem of an increasing per processor memory consumption for larger and larger data sizes, which inevitably causes the processing nodes to start disk swapping at a certain point. Any experiments using caching were carried out with the

[13]Factors such as the quality of the initial guess (if present), the optimization method, or the smoothness of the metric function greatly affect the convergence.

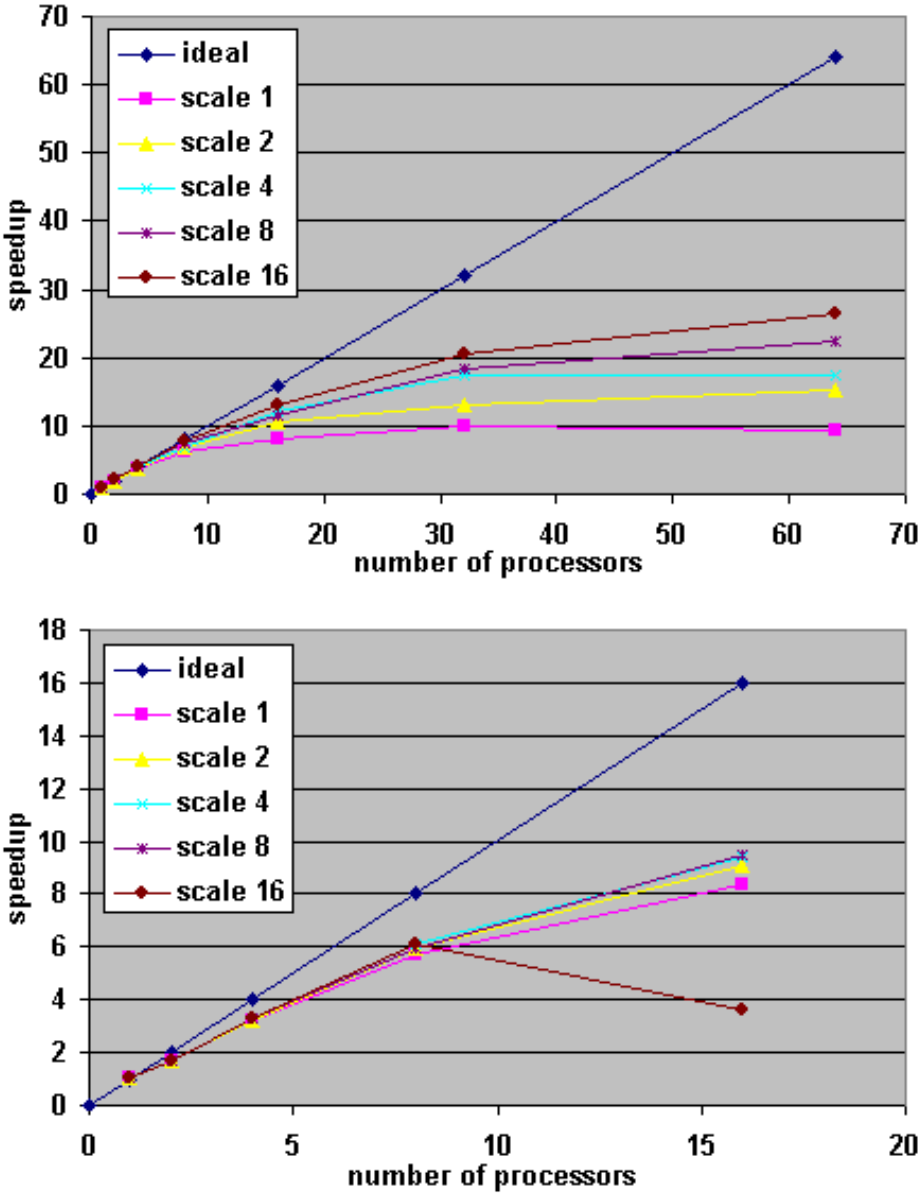



Figure 7.8: Speedup for the experiments at ETH (top) and NNSU (bottom) for the three dimensional rigid problem with 200 iterations. The extraordinary shape of the curve for scale 16 (NNSU) will be explained in the caching part of this section.

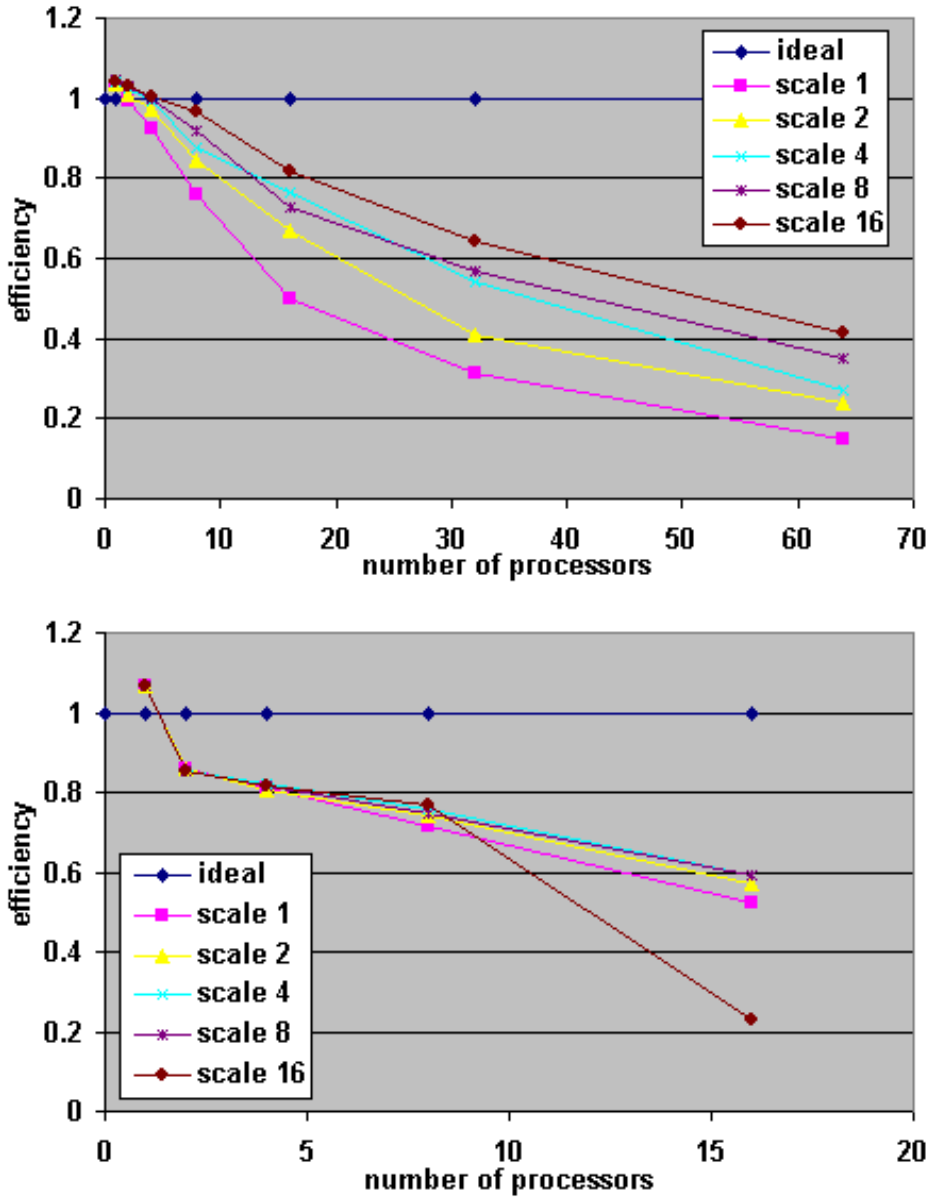



Figure 7.9: Efficiencies for the experiments at ETH (top) and NNSU (bottom) for the three dimensional rigid problem with 200 iterations. The extraordinary shape of the curve for scale 16 (NNSU) will be explained in the caching part of this section.

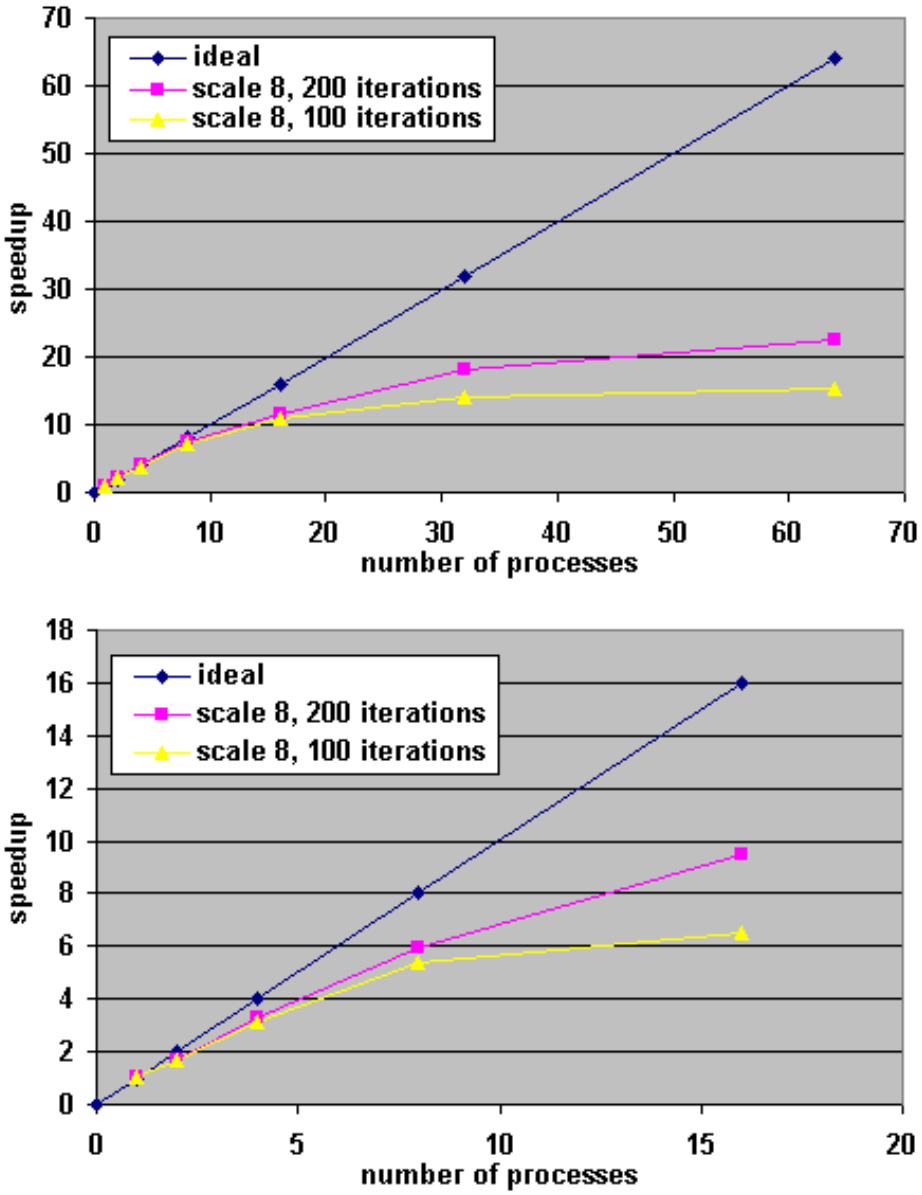



Figure 7.10: Comparison of the speedup of the three dimensional rigid scenario with 100 and 200 iterations at scale 8. The results of the experiments at ETH (top) as well as at NNSU (bottom) are shown.

following parameters:

- Cache block size for the gradient image: $32 \times 32 \times 32$ pixels (768 KB)
- Maximum number of cache blocks for the gradient image: 64
- Cache block size for the moving image: $64 \times 64 \times 64$ pixels (512 KB)
- Maximum number of cache blocks for the moving image: 64

Therefore, the maximum memory consumption caused by these two images should not exceed 80 MB ($64 \cdot (768 + 512)$ KB).

The per slave memory consumption as illustrated in Figure 7.11 shows, that without caching, the physical memory size is exceeded from scale 32 on the Windows machines at NNSU (256 MB RAM), and from scale 64 on the Unix workstations at ETH (512 MB RAM). On the Windows cluster, two processes were executed on each host for the experiments with 16 processors. Therefore, the experiments with scale 16 on 16 processors were already influenced by the consequences of disk swapping. This occurrence of disk swapping explains the extraordinary curves that were observed in Figures 7.5 and 7.6.

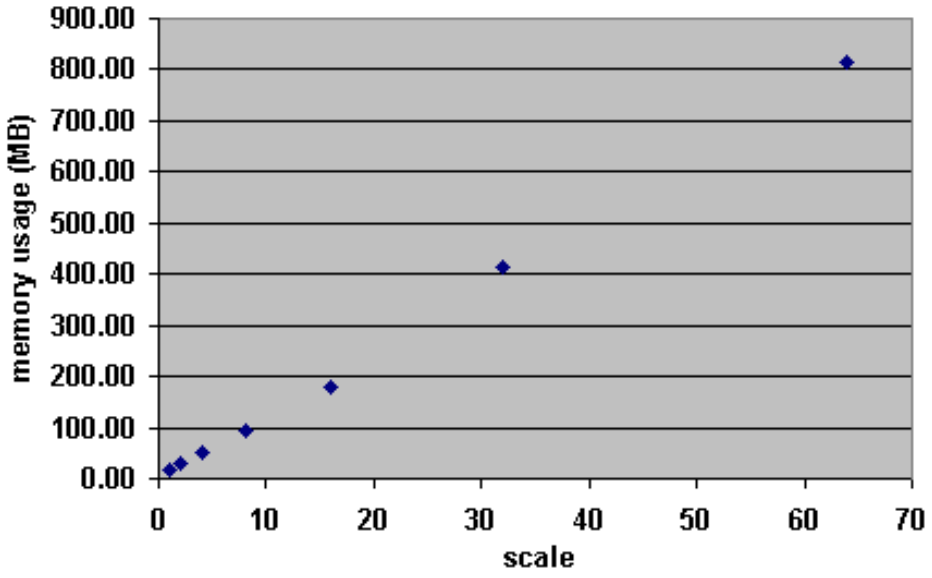



Figure 7.11: Memory usage in dependency of the scale. Note that the memory usage is the total process size. The values were taken from the Unix (SunOS) workstations at ETH.

As soon as the process size of the non-caching version exceeds the physical memory size, a caching-version of the program is supposed to increase performance. The results, which are summarized in Figures 7.12 and 7.13 confirmed this conjecture.

At a first glance, it might be surprising that efficiencies considerably larger than one can be observed, even in cases where no caching is applied (as seen in Figures 7.5 and 7.6). However, the fact that each host works on a partial image only does not only reduce the amount of disk accesses in case of caching, but can also do so in case of usual disk swapping. Analyzing the bone images used in this example showed, that the transform between the fixed and the moving image consisted of a rotation along the z-axis and a slight transformation. Therefore, a slice of the fixed image (which represents the sub-image in the slave) is mapped to a slice of the moving image as illustrated in Figure 7.14. As indicated in the figure, pixels are stored with increasing x, then y and finally z indices. Therefore, the moving image slice is an almost continuous block of memory when using standard ITK data representation. Consequently only little swapping is necessary as soon as the slice is small enough to fit into the physical memory.

To demonstrate the effect of a less optimally aligned data set, the fixed image was rotated around the y-axis. The resulting orientation of the images and the corresponding slices are illustrated in Figure 7.15. In this case, the moving image slice is no longer stored in a continuous memory block when using the ITK data structure. Therefore, a much larger amount of pages are required to represent this slice. Consequently, much more disk swapping is required in this setup and the non-caching version of the parallel program performs significantly worse. A comparison of the speedups reached on the Unix machines for the two data sets and the caching and the non-caching version of the program is shown in Figure 7.16. This figure also shows that the parallel framework can reach an enormous speedup compared to the sequential execution time of standard ITK methods as soon memory becomes critical, particularly when caching is enabled.

Experiments showed, that for a scale of 64, the maximum resident size of the slave processes was in the order 95 MB on the Windows machines and in the order of 160 MB on Unix, when caching was used. The difference most likely comes from different policies of the operating systems when it comes to freeing memory mapped to a file and from the larger physical memory size on the Unix machines in conjunction with this policy. With an increasing number of slaves, the per slave memory usage rapidly dropped (this is shown in Figure 7.17 for scale 64 and 32 on the Unix machines). Considering the

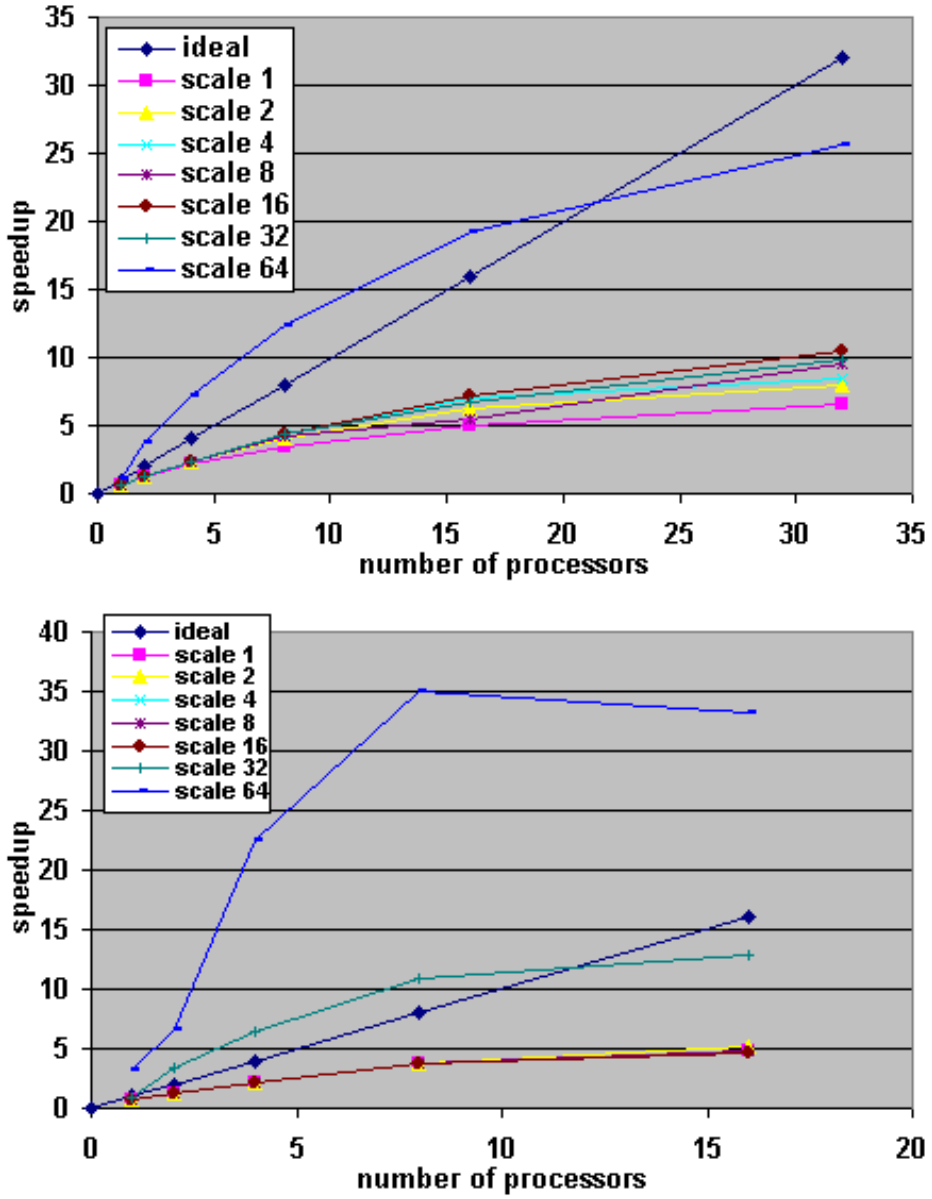



Figure 7.12: Speedup for the three dimensional rigid scenario with caching for the experiments at ETH (top) and NNSU (bottom). The results for scale 64 were carried out with only 50 iterations, the other experiments with 100 iterations. Note that some speedup values are higher than those of the "ideal" curve. This is due to the disk swapping which slowed down the sequential reference program.

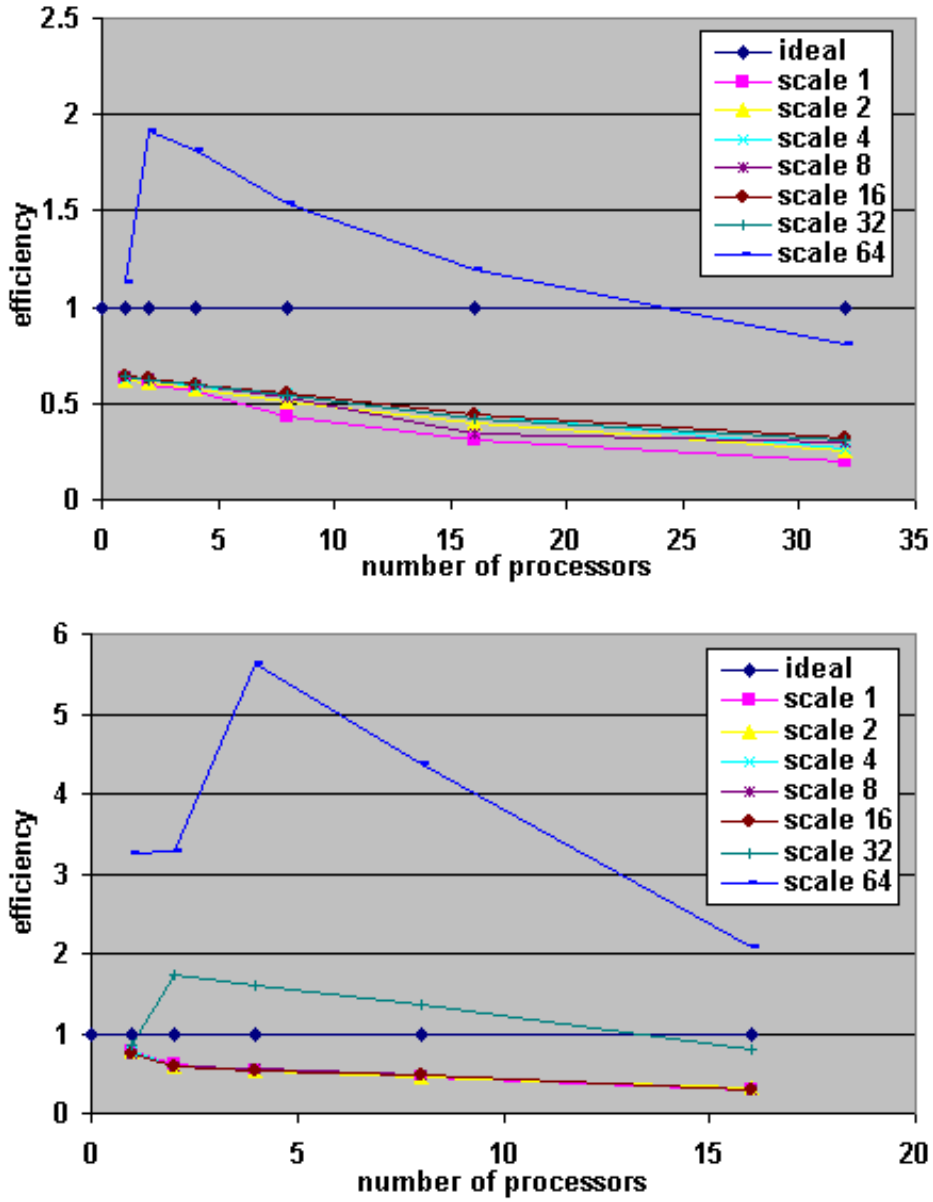



Figure 7.13: Efficiencies for the three dimensional rigid scenario with caching for the experiments at ETH (top) and NNSU (bottom). The results for scale 64 were carried out with only 50 iterations, the other experiments with 100 iterations. Note that some efficiency values are higher than 100%. This is due to the disk swapping that slowed down the sequential reference program.

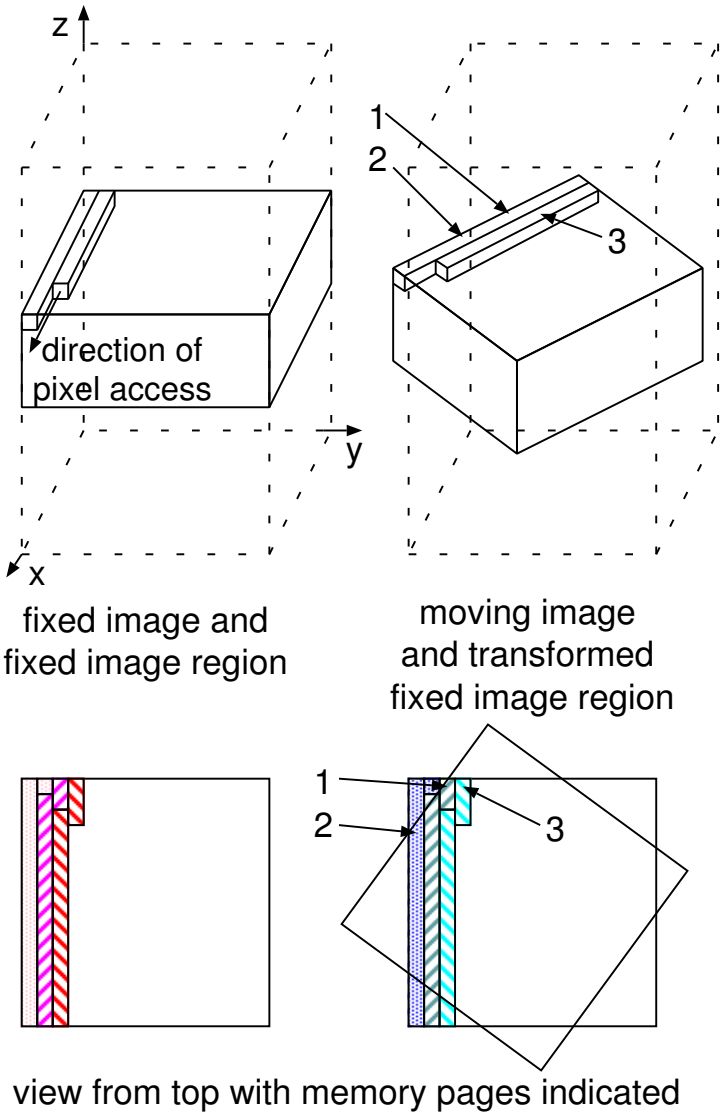



Figure 7.14: **Top:** Orientation of the images as well as of the fixed and the transformed fixed image region. **Bottom:** View from the top with the operating system memory pages indicated. The numbers indicate the order of the page accesses (NOT the pixel accesses). It can be seen, that first all the pages in the topmost layer are accessed, then those in the second layer from top and so on. Thus, the data accesses are restricted to a compact block in memory.

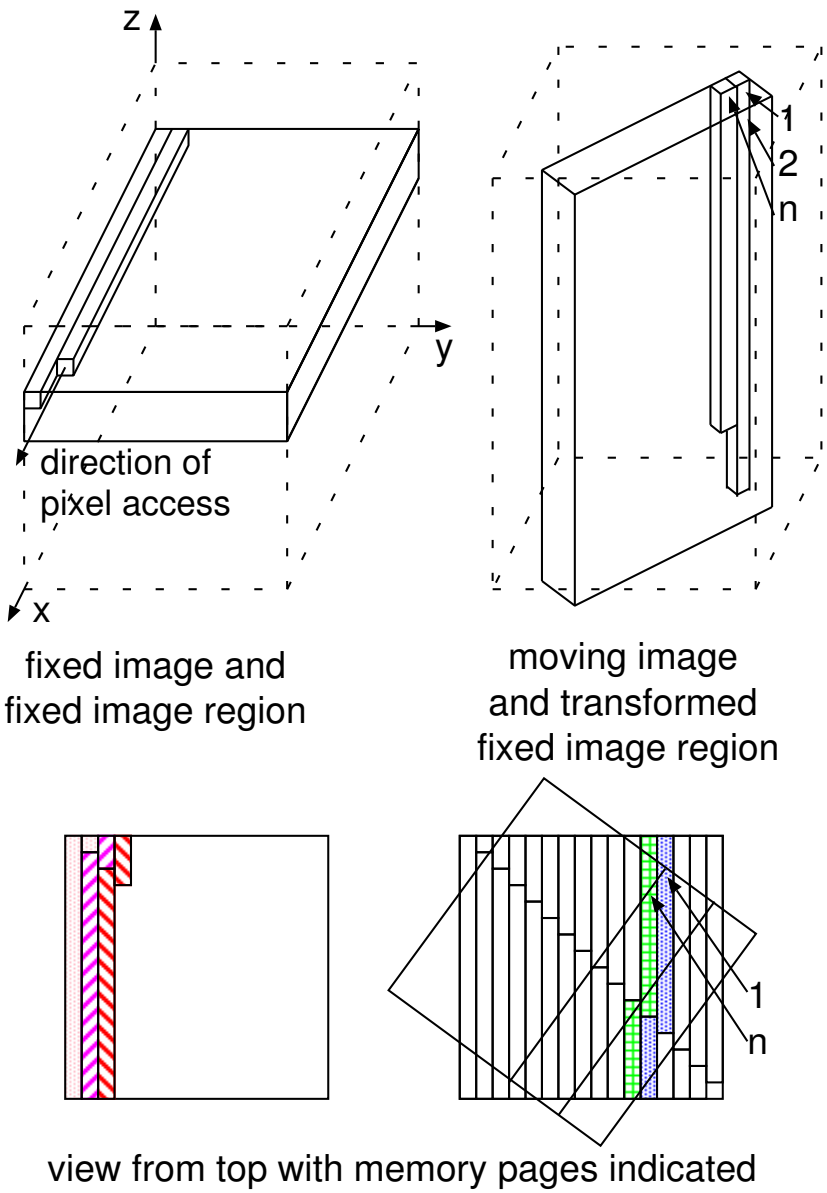



Figure 7.15: **Top:** Orientation of the images as well as of the fixed and the transformed fixed image region. **Bottom:** View from the top with the operating system memory pages indicated. The numbers indicate the order of the page accesses (NOT the pixel accesses). Compared to the area of the fixed image slice, a much larger number of memory pages is required per layer than in the situation in Figure 7.14. What's more, pixels are this time accessed in direction of the z-axis. Therefore, succeeding pixel accesses address different pages. The consequence are much more frequent swapping operations (and hence disk accesses) as soon as the total region does not fit into main memory.

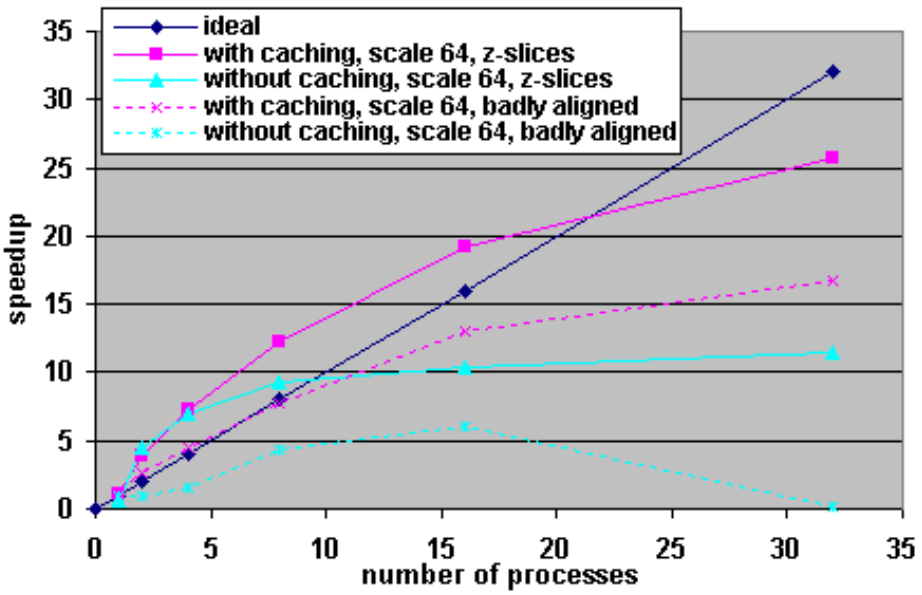



Figure 7.16: Comparison of the behavior of the parallel framework with and without caching for differently aligned data sets. When the resulting moving image regions approximately represent slices along the z-axis, both, the caching as well as the normal version of the parallel framework perform better than for less optimally aligned data sets and both can reach efficiencies of more than 100%. For both alignments, the caching version performs better (particularly on a large number of processors), and particularly for the badly aligned data sets the results are more predictable. The underlying data of this figure originates from the experiments at ETH.

(theoretically[14]) limited amount of memory and the high efficiencies (up to more than 500%) that seem[15] to significantly increase with an increasing problem size (see Figure 7.18), the architecture can be said to behave well with respect to scaling.

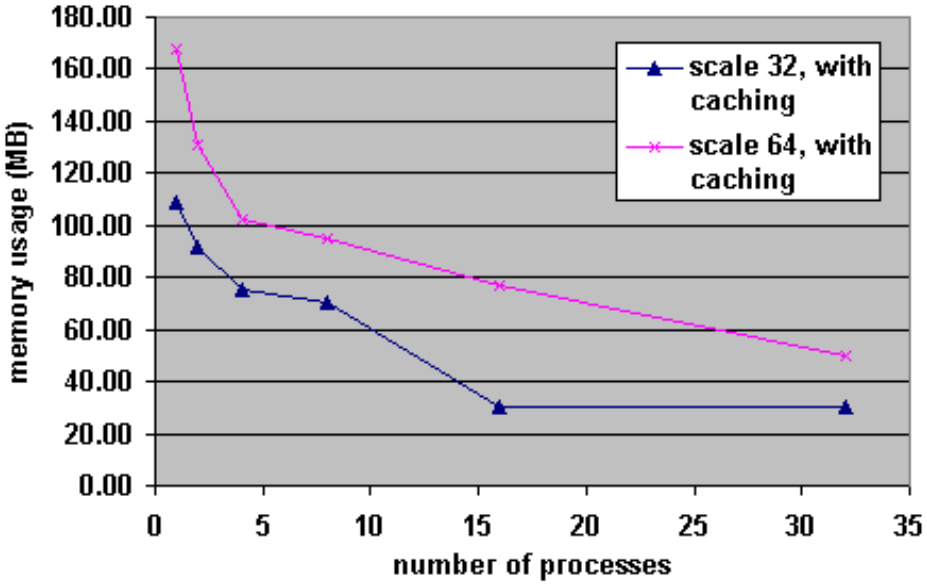



Figure 7.17: Memory usage in dependency of the number of processes when caching is enabled. Note that the memory usage is the total process size. The values were taken from the Unix (SunOS) workstations at ETH.

The results related to caching showed, that caching can significantly increase performance in certain situations. However, they also show, that it should not be used blindly. It is inevitable to think about the amount of data, and even about its approximate orientation in space in order to estimate the consequences. If not used carefully, caching might significantly reduce instead of increase performance.

[14]Experiments with a much lower number of cache image blocks showed, that this limitation also works in practice. For two processes and scale 16, for example, the total and resident process sizes were only 42 and 32 MB, respectively when using 10 blocks each, compared to 91 and 63 MB when using 64 blocks.

[15]Unfortunately, it was almost impossible to carry out experiments on larger scales which might have proved this assumption.

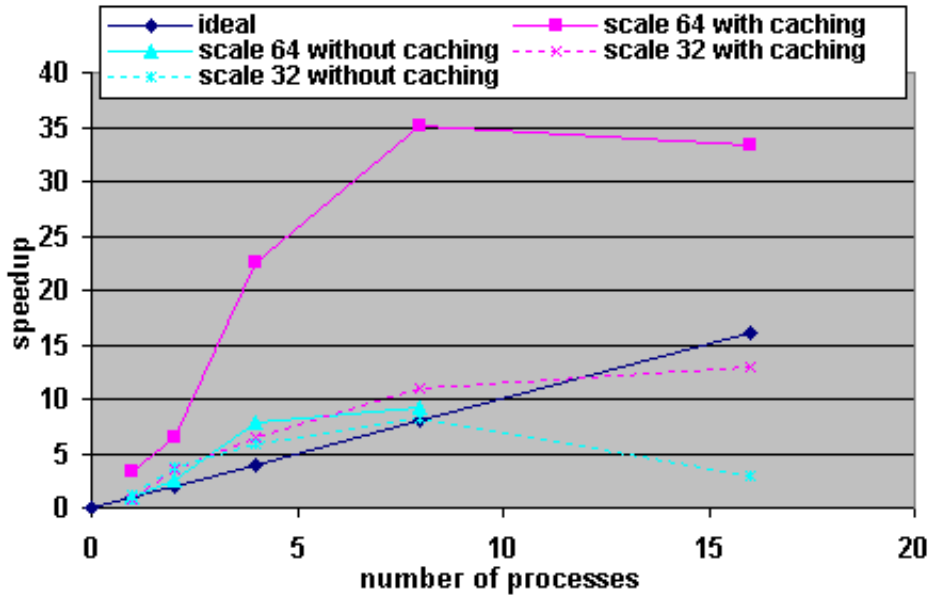



Figure 7.18: Comparison of the behavior of caching for an increasing data size based on the results from the experiments at NNSU. When the process size only slightly exceeds the physical memory size (scale 32), no significant performance difference can be observed between the non-caching and the caching version of the program. However, the situation drastically changes for an increasing data size (scale 64).

### Analysis of the MPE Log-Files

When analyzing the MPE log files, it becomes apparent that a considerable amount of time is spent in the initialization process. With approximately 65 seconds, or approximately 2% of the total execution time, this part seems small when looking at Figure 7.19. However, Amdahl's law states, that this already limits the maximally reachable speedup to about 50. Even worse, Figure 7.20 reveals, that due to a more complicated data distribution, the initialization phase takes even more time on a larger number of processes (approximately 135 seconds in this example with 33 processors) and it becomes also optically visible that the time spent for initialization significantly affects speedup, even though the overhead within the optimization phase is rather low. The long initialization phase also explains the significant improvement in speedup when increasing the number of iterations.

When looking more closely at this initialization phase, as this is done in Figure 7.21, it becomes apparent that particularly on a low or moderate number of processors, a significant part of this initialization phase consists of gradient image calculation and that further a lot of time is spent by sending image data. Section 8 will show, how the time for gradient image calculation can be reduced and methods will be presented that allow to send image data in a smaller amount of time.

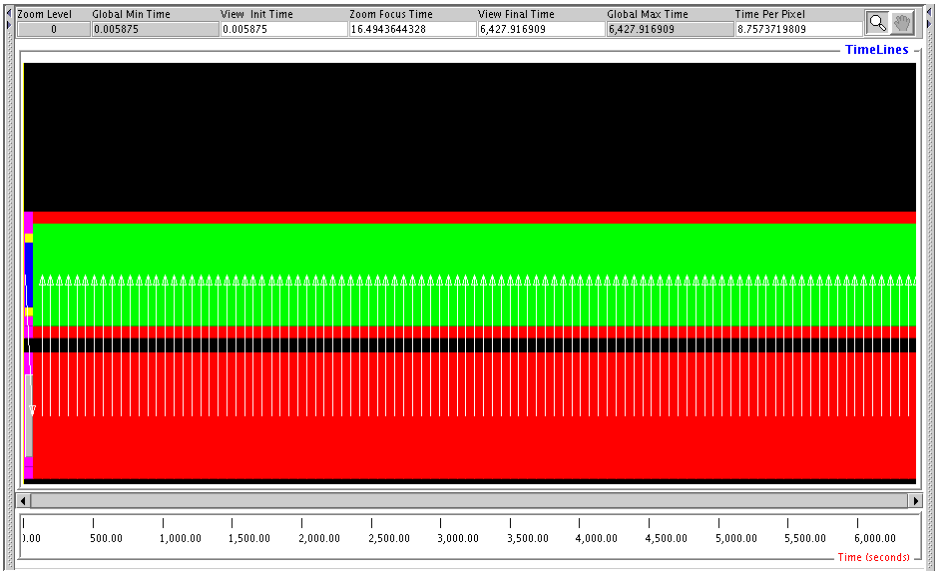



Figure 7.19: Jumpshot diagram for two processes. To the left, the initialization phase (violet) can be seen. It makes up only a small part of the total computation time.

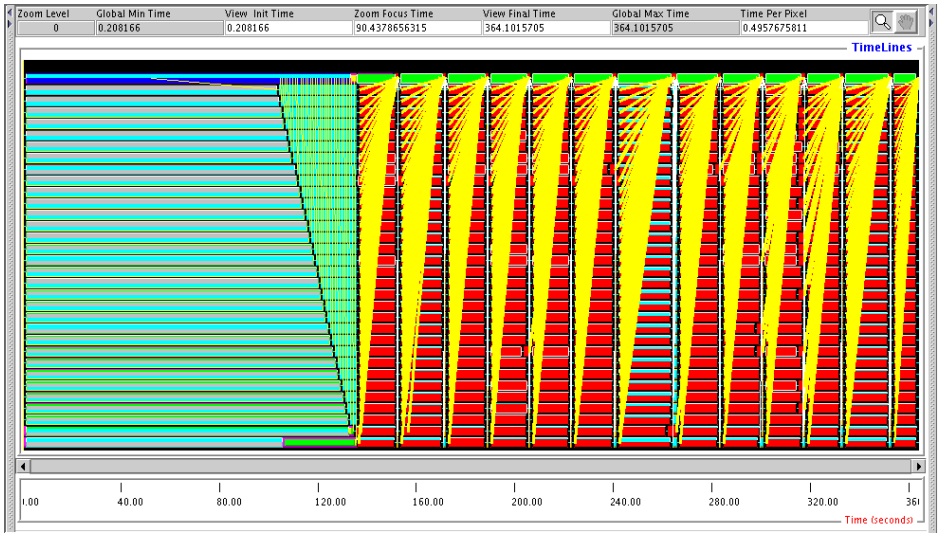



Figure 7.20: Jumpshot diagram for $N = 32$. The initialization phase to the left (mainly green) makes up a considerable part of the overall wall-clock time in this case.

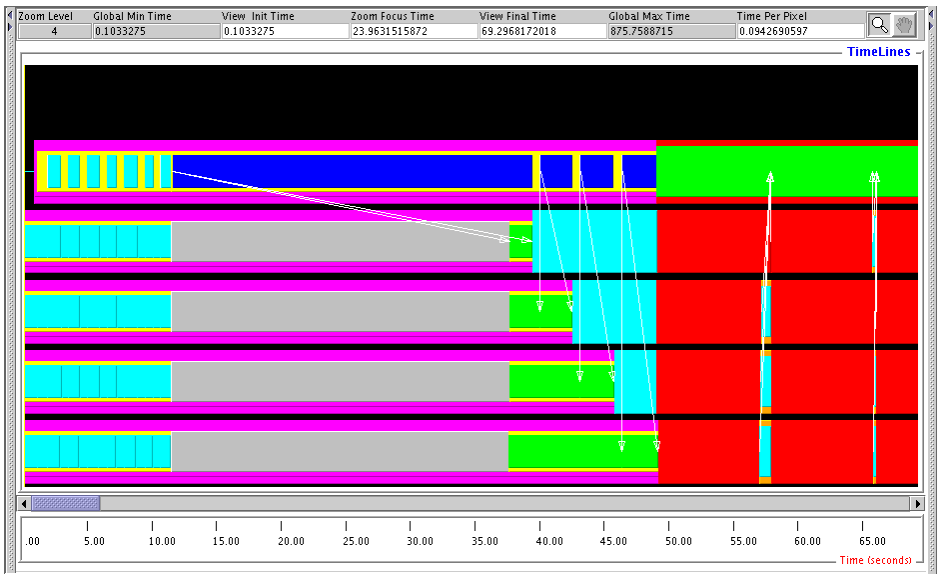



Figure 7.21: The initialization phase (violet) basically consists of sending the moving image (light blue to the left), calculating the gradient image (gray) and sending the fixed image regions (green).

The log files further show, that only a negligible part of the time is spent in the other non-parallelizable code segments, such as the optimizer and the post processing of the intermediate results. However, a significant overhead is generated by load imbalance. Figure 7.22 shows a case, where many slaves have to wait for one process that sits on a CPU that was (most likely) slowed down by other user processes. Further load imbalance is caused by a non equally distributed workload due to unequal region splitting (slice thickness can vary by one pixel, since it has to be an integer number) and due to regions that are not mapped onto the moving image and thus produce less work. This problem is is illustrated in Figure 7.23.

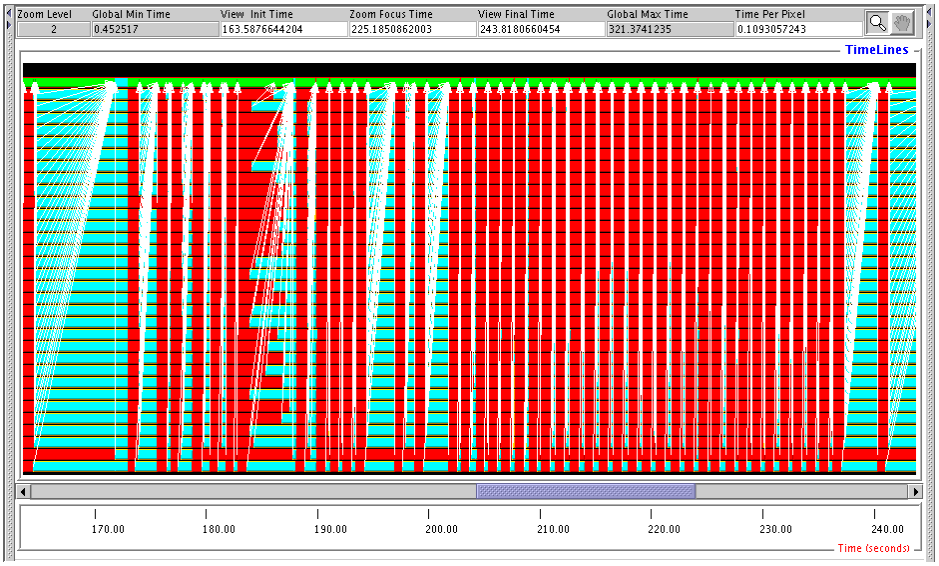



Figure 7.22: The second last processor is (most likely) not free of load from other users. Therefore, all the other processes have to wait for it (light blue). The large block that is mainly red (computing time) shows that the program could perform much better. The thin white lines within this block are the point to point operations that transmit the intermediate results.

### Summary

Speedup and efficiency figures for the three dimensional rigid example are in the order of those reached by other projects. The use of caching further allows to treat large data size problems with high efficiency on a large number of processes, due to a low memory usage.

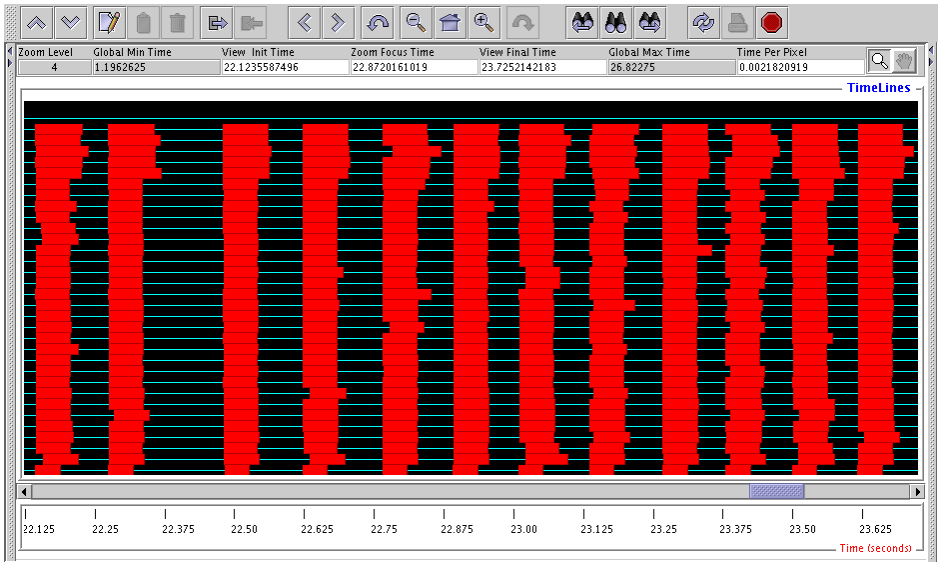



Figure 7.23: The computing times (red) of the first 5 slaves are longer, since they work on a larger image region. The image size in z-direction was 101 pixels, which resulted in 27 slices of 3 and 5 slices of 4 pixels width. The slice of the last slave was most likely not entirely mapped onto the moving image, which resulted in even less work.

The scenario is characterized by a long initialization phase which inherently limits the achievable speedup. Due to the large data sizes, the time for function evaluations becomes long compared to the time spent to send and process the intermediate results. Therefore, the parallel overhead during the optimization phase is low.

### 7.5.2 2D Bspline Deformable Scenario

The two dimensional bspline deformable problem was evaluated for scales 1 to 16 on 1 to 16 processors at NNSU and on 1 to 64 processors at ETH. The results are summarized in Figures 7.24 and 7.25. The wall-clock times of the sequential program can be found in Figure 7.26.

The experiments at ETH show a steady increase in efficiency when going to larger data sets. The (approximate) isoefficiency functions for 80%, 82% and 84% are shown in Figure 7.27. It is difficult to guess the exact extrapolation of the curves. From visual inspection, however, it can be expected that the slope will not extremely increase even in the area of 100 processors[16]. Considering problem size increase factors of about 2.5 or 3 to keep efficiency at the constant level of 80% when doubling the number of processors, it can be said that the methods scales well in this scenario. Unfortunately, per node memory consumption increases proportionally with the problem size. Therefore, the good scaling properties have to be relativized. However, even on the Windows machines with 256 MB of memory, the maximum "scale", such that no swapping occurs, would be approximately 100 (compared to a maximum of 16 that was evaluated). For even larger data sets, caching could be used in order to account for the high memory requirements. Thus, the scenario can be considered scalable despite of the increasing memory requirements for larger workloads.

#### Analysis of the MPE Log-Files

The inspection of the MPE log files showed a rather short initialization phase. The computationally intensive bspline interpolation caused long calculation times compared to the overhead incurred by transmitting the intermediate results. Therefore, the overall parallel overhead within the optimization phase was low, as depicted in Figure 7.28. The time spent in the optimizer was negligible, despite of the large number of parameters.

A careful reader might have noticed, that the results of the experiments on the NNSU cluster were worse compared to those on the workstations at

[16]Recall from the requirement specification, that 100 processors was the approximate maximum size that was expected for typical applications.

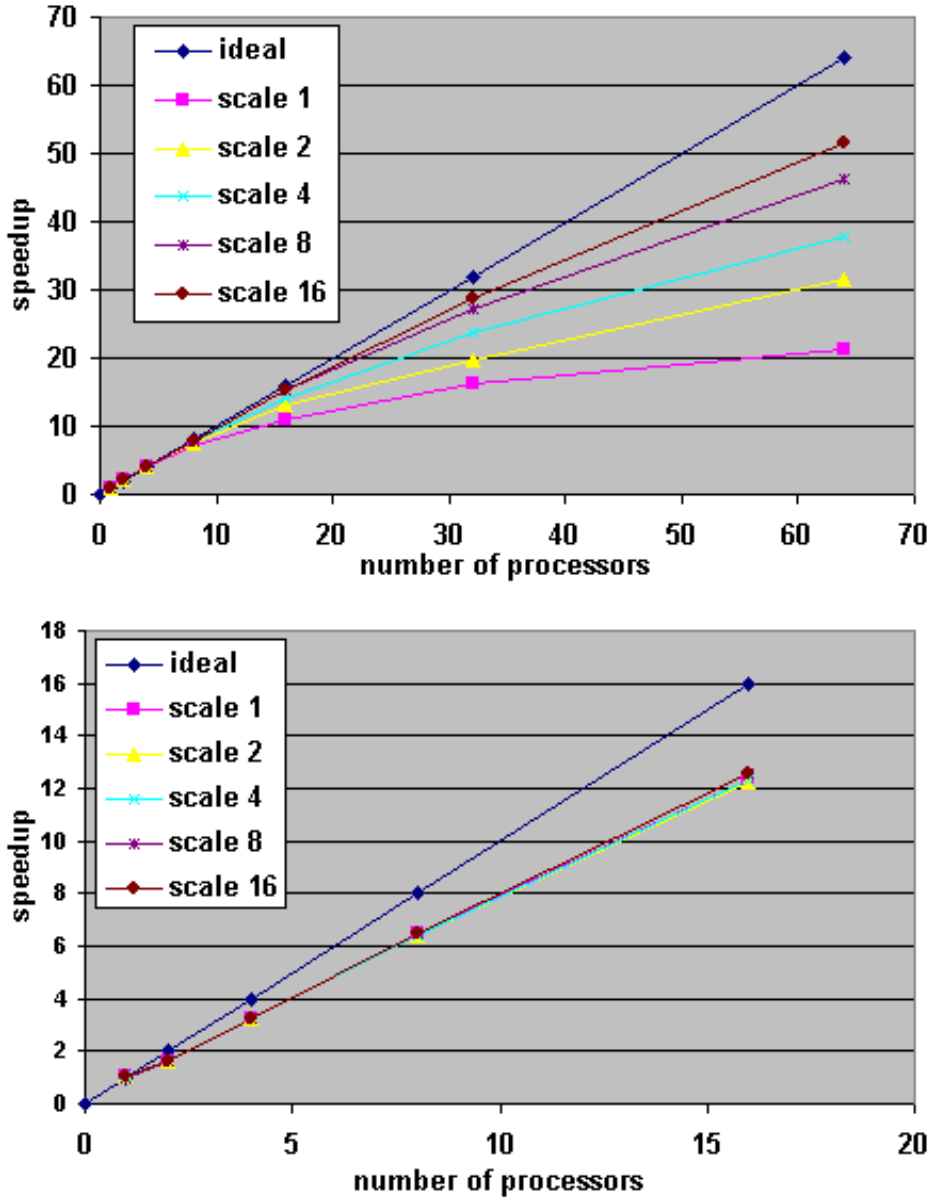



Figure 7.24: Speedup values for the two dimensional bspline deformable problem. In the experiments at ETH (top) speedup steadily increases for an increasing problem size. This is not the case for the experiments at NNSU (bottom). The reason will be explained during the discussion of the MPE log files later in this section.

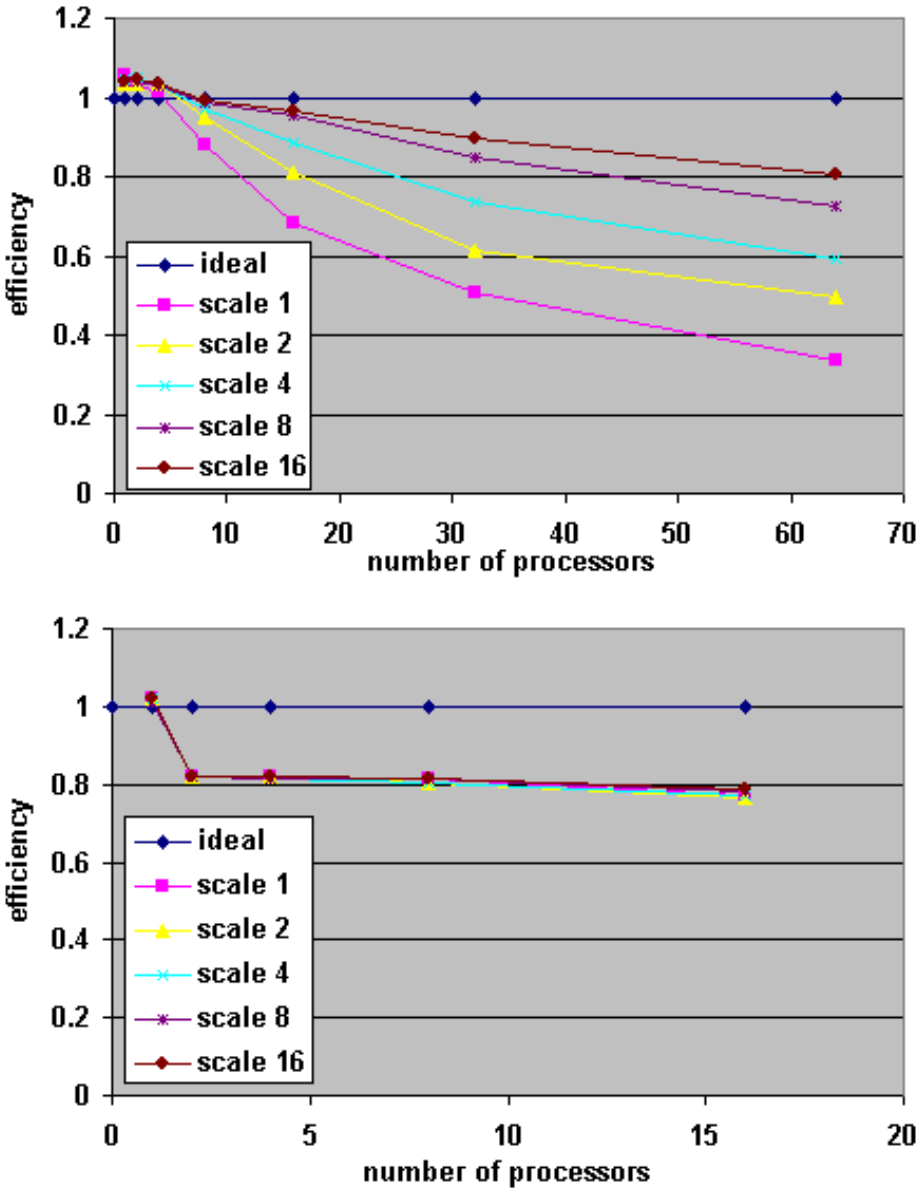



Figure 7.25: Efficiency values for the two dimensional bspline deformable problem. In the experiments at ETH (top) efficiency steadily increases for an increasing problem size. This is not the case for the experiments at NNSU (bottom). The reason will be explained during the discussion of the MPE log files later in this section.

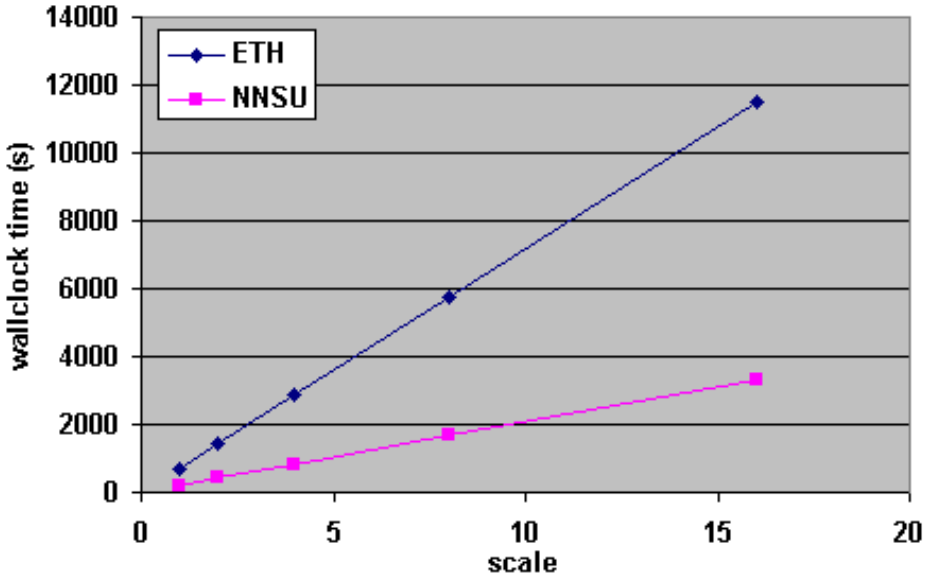



Figure 7.26: The wall-clock times of the two dimensional deformable scenario for increasing image sizes.

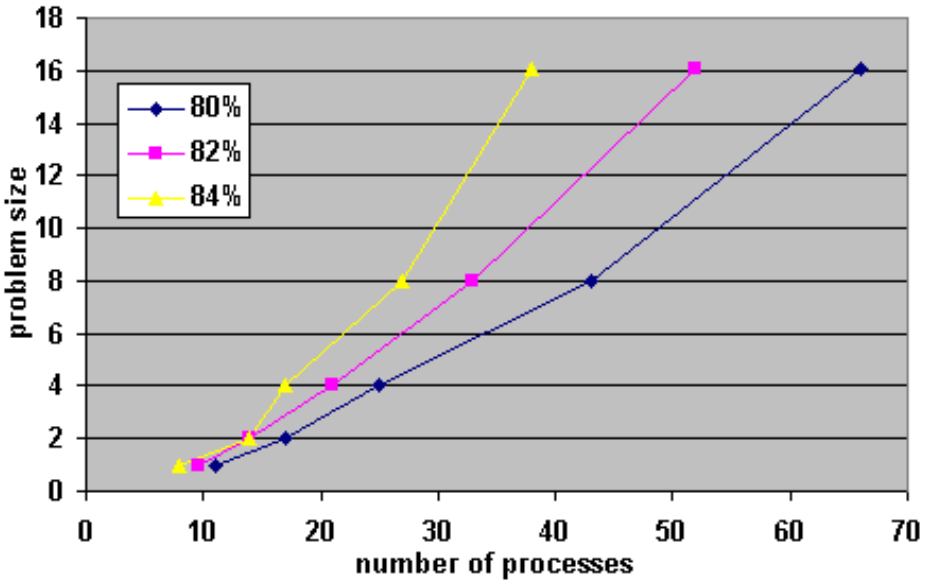



Figure 7.27: The isoefficiency functions for the two dimensional bspline deformable scenario and the experiments at ETH.

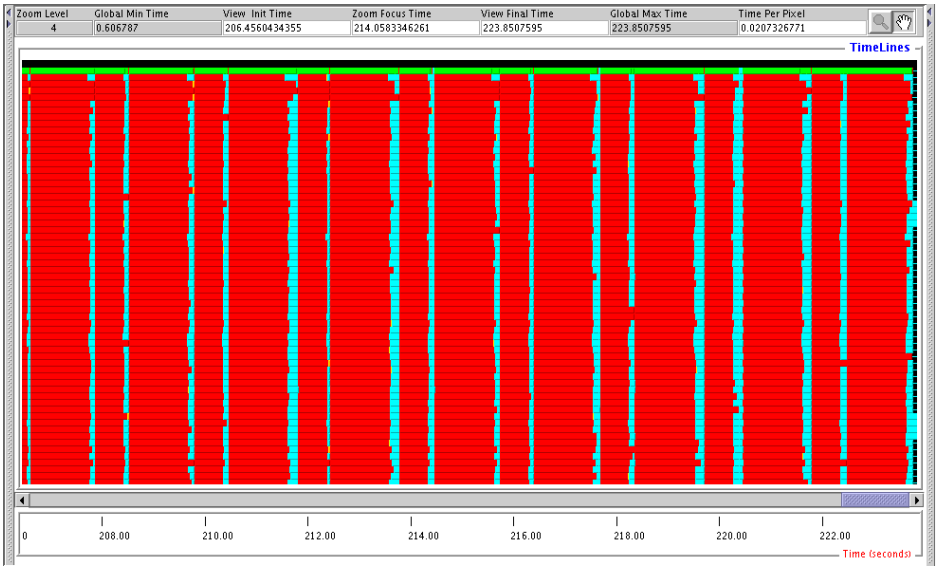



Figure 7.28: Even for 64 slaves, the communication overhead (light blue) is low compared to the computing times (red) in the bspline scenario.

ETH. They also showed a significant efficiency decrease when incrementing the number of processors from one to two. Inspection of the log files showed, that one host (the second one) performed slightly worse in all the examples (including those for the other scenarios)[17]. This is depicted in Figure 7.29. Due to the low overall overhead in the bspline deformable scenario, the influence of this slower node was particularly high in this case.

**Summary**

The bspline deformable problem is characterized by a short initialization phase as well as a small parallel overhead during optimization. Therefore, high efficiencies can be reached. Because of the small initialization phase, the problem scales well as long as memory is not critical. Most likely, caching allows to keep up this scalability even for memory intensive tasks, however, this has not experimentally been proved.

### 7.5.3 2D Affine Scenario

The two dimensional affine scenario was evaluated for scales 1 to 16 and 100 iterations. Since a histogram based metric was applied, the intermediate

[17]The reason is, that this node acts as a file server for the cluster.

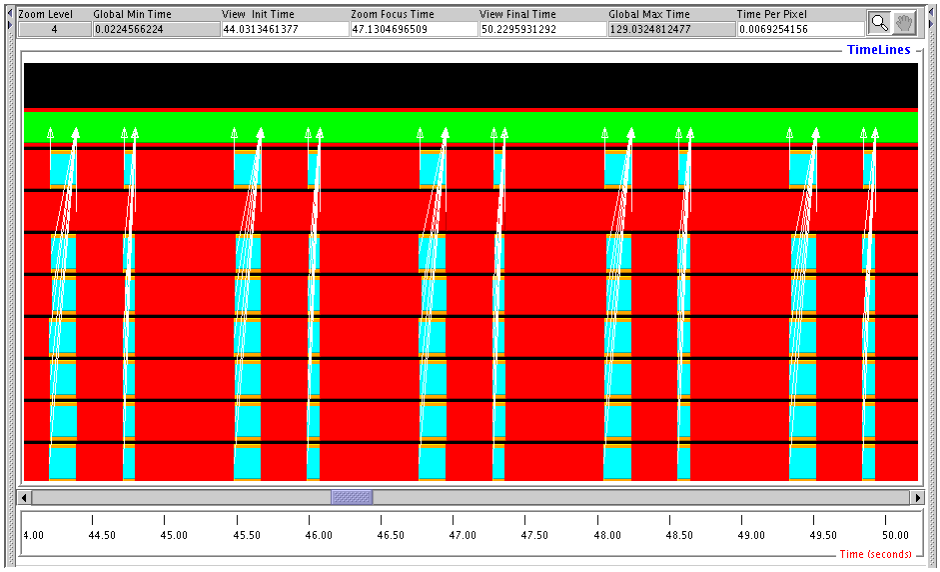



Figure 7.29: On the Windows cluster at NNSU, one node (the second one, 3rd row) is slightly more loaded than the others since it acts as a file server for the cluster. Therefore, the other nodes permanently have to wait (light blue) for this one.

results were large in this example (a whole $64 \times 64$ histogram), which resulted in a high communication overhead. Therefore, the effect of overriding the `IntermediateResultsAllReduce()` method (see Sections 6.2.2 and 6.2.3) was evaluated in this example.

The results of the variant without overriding the `IntermediateResultsAllReduce()` method are summarized in Figures 7.30 and 7.31 and the wall-clock times of the sequential program are illustrated in Figure 7.32. For low scales they show poor efficiencies. However, a significant increase of parallel efficiency can be achieved when going to larger problem sizes. For scale 16 the results of this example can already compete with those of other projects. The isoefficiency functions for 56%, 60% and 64% are shown in Figure 7.33.

### Overriding `IntermediateResultsAllReduce`

The treatment of intermediate results of the `DistributedImageToImageMetric` is not optimal when in each stage a large number of intermediate results has to be transmitted an processed. A large number of processors causes a large number of point-to-point

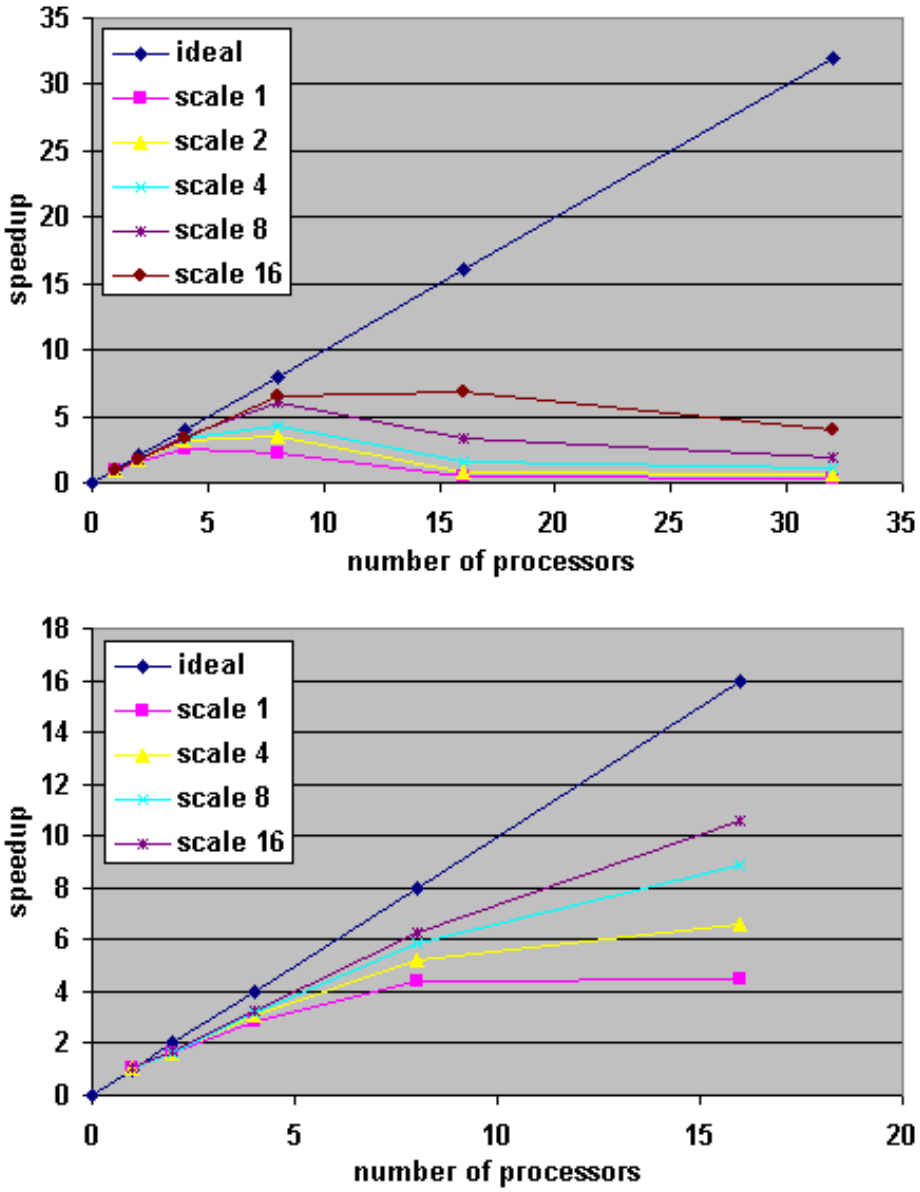



Figure 7.30: Speedup values for the two dimensional affine problem. In the experiments at ETH (top) speedup even decreases for a high number of processors due to the high communication overhead. This was not the case for the experiments at NNSU (bottom), where the nodes were connected by a faster 1Gbit Ethernet.

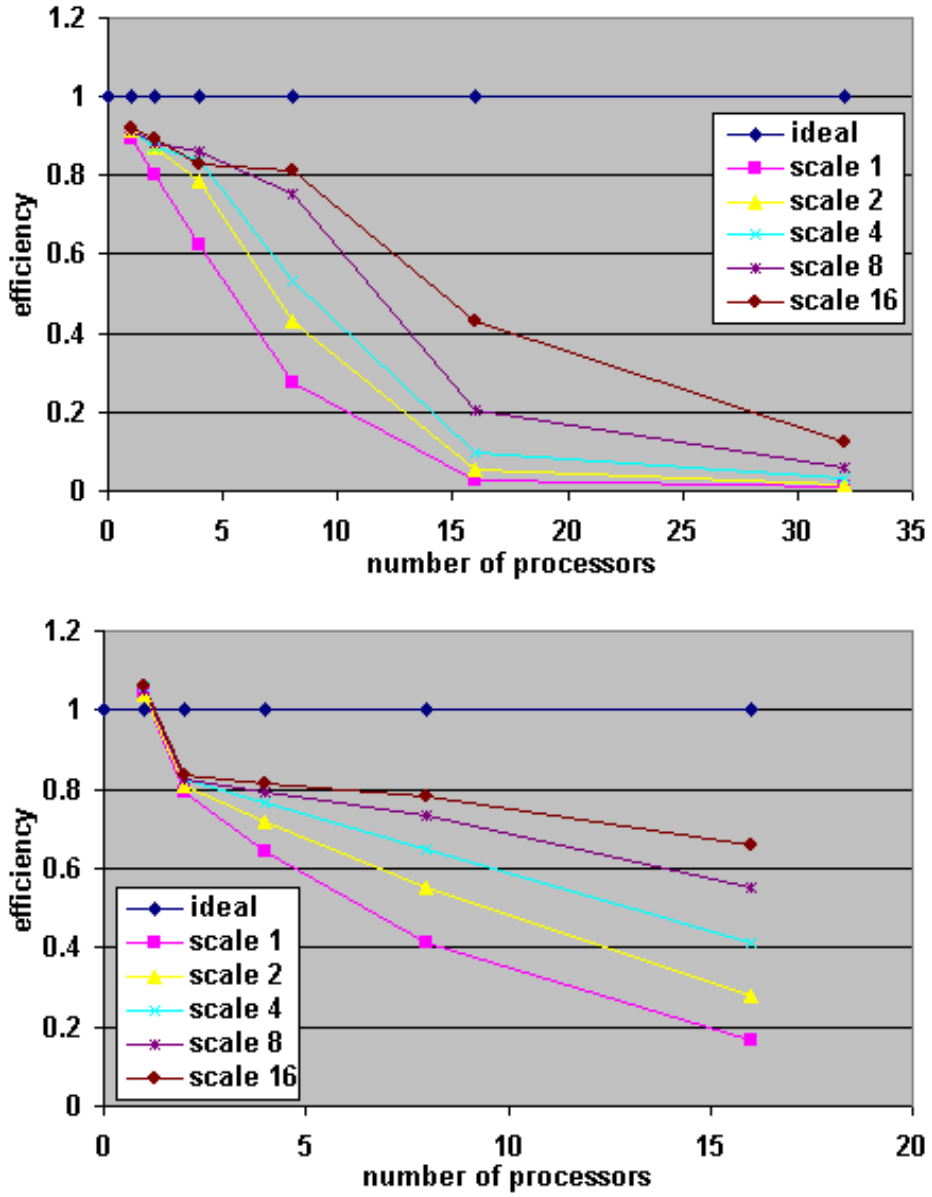



Figure 7.31: Speedup values for the two dimensional affine problem. The better results at NNSU (bottom) compared to those at ETH (top) mainly result from the faster 1Gbit interconnection network at NNSU.

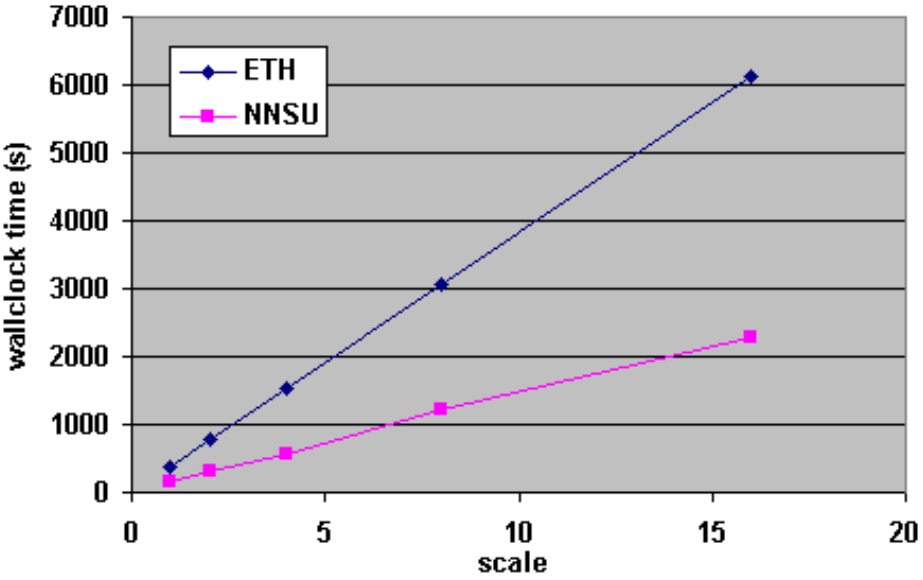



Figure 7.32: The wall-clock times of the sequential program for the two dimensional affine scenario.

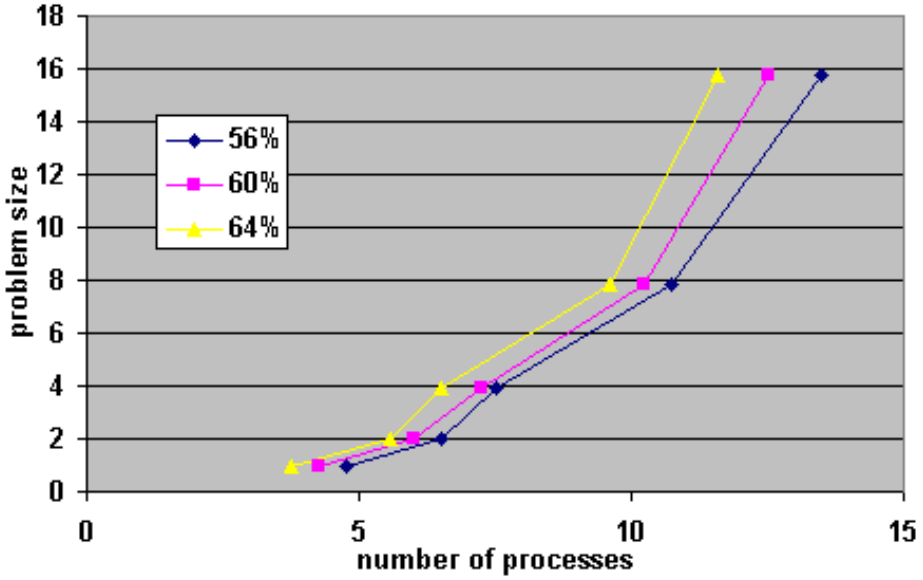



Figure 7.33: Isoefficiency of the two dimensional affine problem for the experiments at ETH.

messages, that all have to be received at the master node. All the results have then to be post processed by a singe node (the master node). Further, intermediate results are by default transmitted as values of type `double`, which is not always appropriate.

Post processing of the partial histograms produced by a histogram based metric consists of simply adding the respective bin frequencies. This can efficiently be done in a tree structure[18], which reduces the amount of messages that have to be received by the master from $N$ to $\log_2(N)$. Further, the histogram values represent frequencies that do not require a `double` value but can be represented by `integer`s, which reduces the message size by a factor of two.

Exactly these modifications (representing histogram bins using `integer`s and summing the bins up using a tree structure[19]) have been incorporated into the histogram metric by overriding the `IntermediateResultsAllReduce()` method. The results for this scenario with the modified metric function are shown in Figures 7.34 and 7.35. As can be seen, many of the efficiency values are lower than when using the non-modified metric, which might seem disappointing. When, however, the isoefficiency functions illustrated in Figure 7.36 are compared to those of the non-modified example (Figure 7.33), it becomes obvious that the scaling behavior has significantly improved. While the slopes of the curves rapidly increase in the non-modified case, those of the modified metric are almost constant.

### Analysis of the MPE Log-Files

The MPE log files of the non-modified metric showed, that on a large number of processors, the communication overhead dominates over the computing time during the optimization phase. This is illustrated in Figure 7.37 for 33 processors and scale 16. Reducing message size by better the use of a better suited data type and replacing the point to point communication by `MPI_Reduce()` allowed to improve the situation, as shown in Figure 7.38. However, the overhead is still considerable. Therefore, efficiency drastically increases for larger data sizes that lead to longer computing times at the same communication cost during the optimization phase.

[18] I.e. in a first step, any two neighboring processes add their histograms. The results are stored on any process with even rank. In a second step, these new histograms (only half of the number of the original partial histograms exists now) are again added, and the results are stored in the processes with ranks $0, 4, 8, \ldots$, and so on.

[19] Which can easily be achieved by calling `MPI_Reduce()` (at least when using the MPICH implementation of MPI)

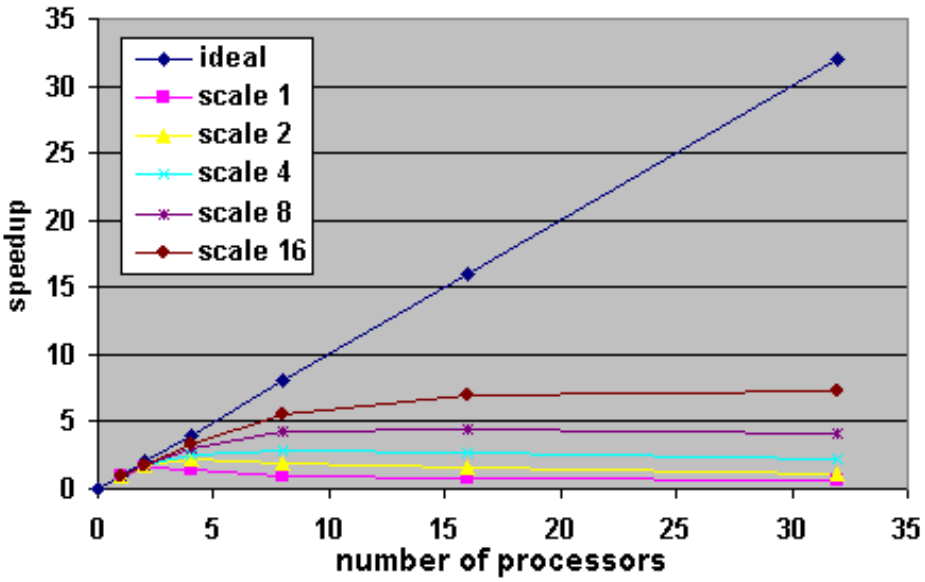



Figure 7.34: Speedup values for the modified two dimensional affine problem that uses `MPI_Reduce()` for the processing of the partial results. The experiments were carried out at ETH.

### Summary

For small problem sizes, parallel speedup and efficiency are poor in the two dimensional affine scenario. However, the situation significantly improves when going to larger data sizes. Introducing an improved processing of the intermediate histograms generated in the slaves resulted in isoefficiency functions that speak for good scalability properties.

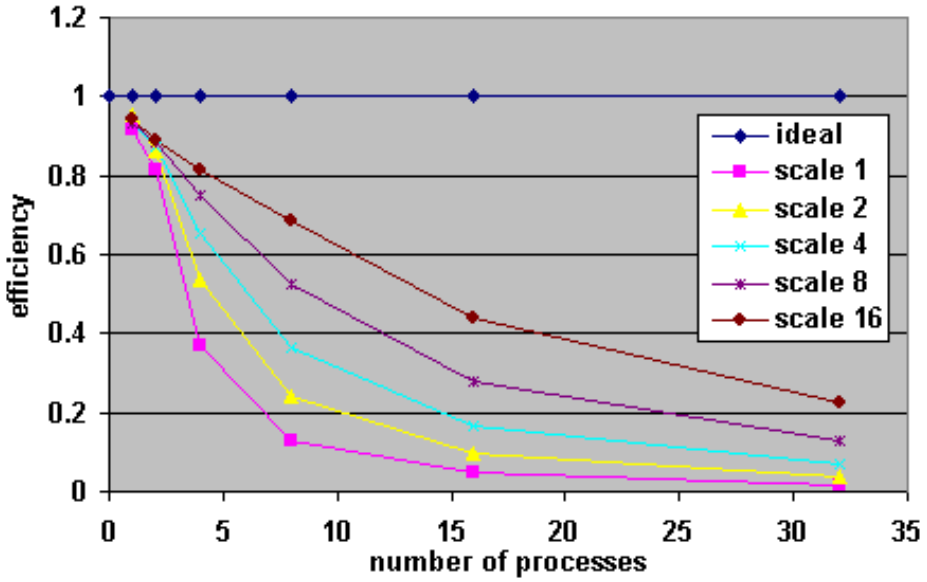



Figure 7.35: Efficiency values for the modified two dimensional affine problem that uses `MPI_Reduce()` for the processing of the partial results. The experiments were carried out at ETH.

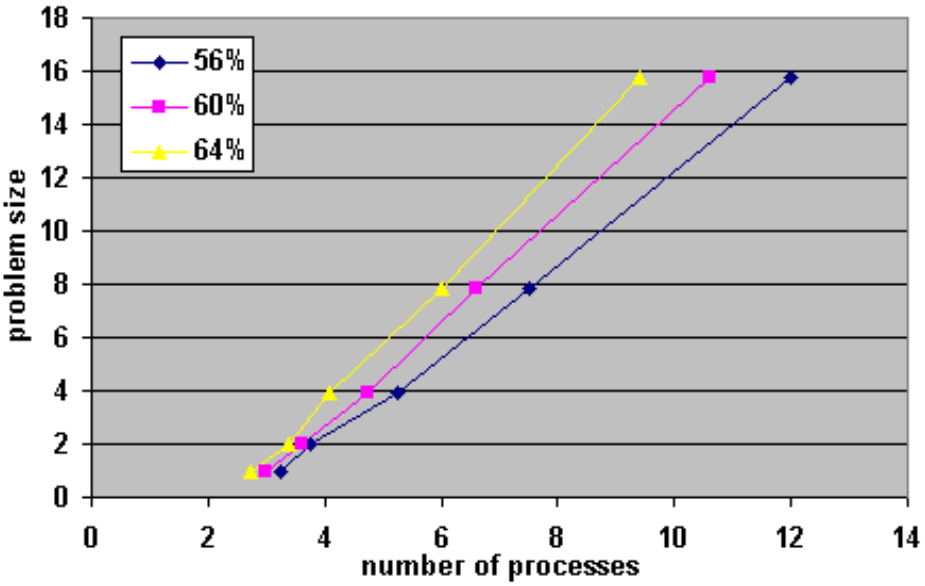



Figure 7.36: Isoefficiency functions for the modified two dimensional affine problem that uses `MPI_Reduce()` for the processing of the partial results. The almost linear functions indicate good scalability. The experiments were carried out at ETH.

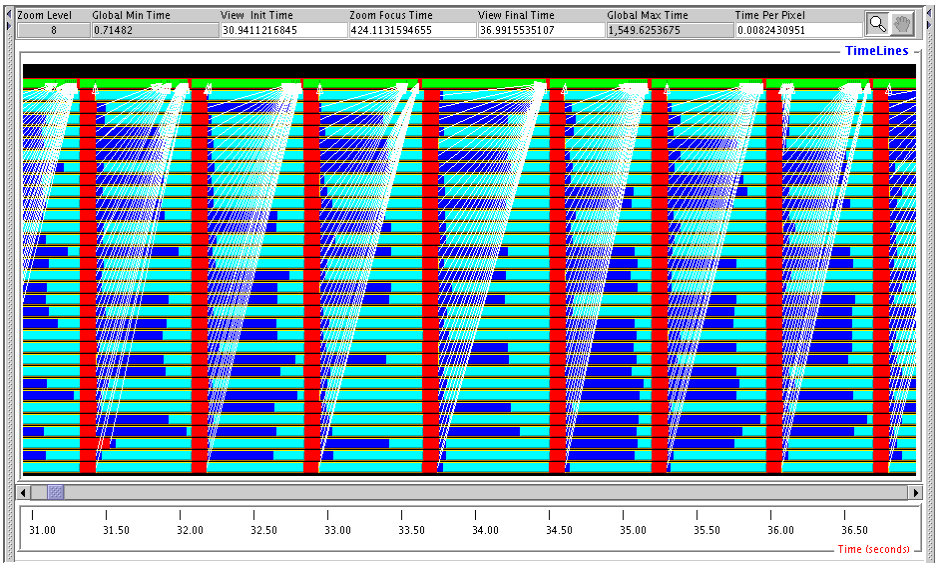



Figure 7.37: A large number of point to point communications (white arrows) that transmit messages of a considerable size ($64 \cdot 64 \cdot 8$ bytes) cause a lot of overhead and dominate over the computing time (red).

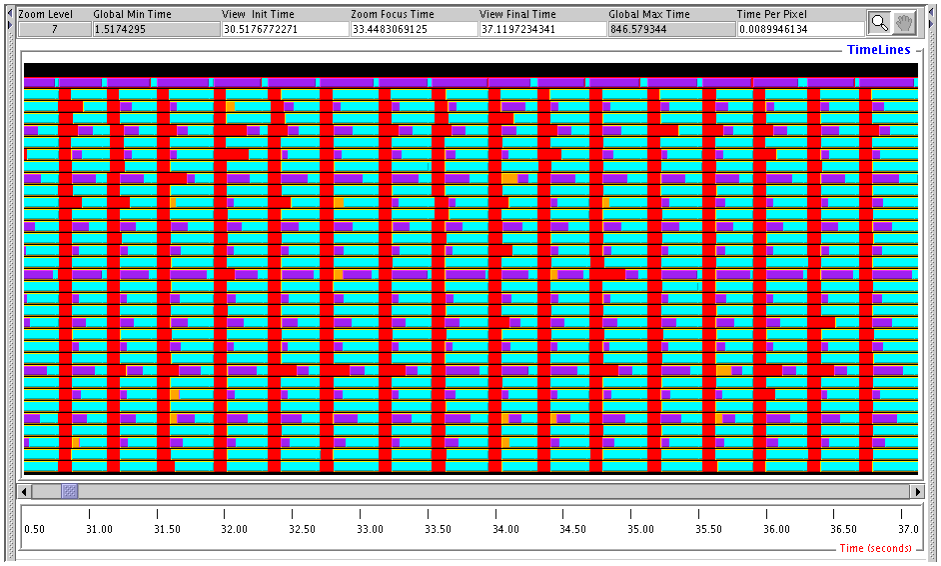



Figure 7.38: Using `MPI_Reduce()` and a better data representation, the communication overhead can considerably be reduced compared to the one depicted in Figure 7.37. However, it is still large compared to the computing times (red).

# Chapter 8

# Future Work

## 8.1 Reducing Parallel Overhead

### 8.1.1 Need for Dynamic Load Balancing

As already mentioned in section 5.2, dynamic load balancing is indispensable when deploying parallel computing methods on clusters of work stations in "real world" environments (such as a hospital). The experiments carried out on the Tardis cluster at ETH, which could not entirely be protected against interference caused by other users, showed that the parallel registration framework is no exception.

One way to extend the framework with dynamic load balancing is the use of progress monitoring inside the slaves. Whenever a certain percentage (say, 5%) of the assigned fixed image region is completed, a slave process could send a message to the master. As soon as one slave has finished its work, the master could send a part (say, the last 5%) of the fixed image region of the slowest process to this fastest process which in turn calculates the associated intermediate values. The slowest slave would then be informed about that and finish its calculation after 95% of its region[1]. Since typically the performance of a single node does not change very much from one iteration to the next, the newly distributed image regions should be kept in memory in order to minimize the cost of sending image data in succeeding iterations. In case caching is enabled, care has to be taken that data locality is preserved. This can for example be achieved by only allowing neighboring nodes to take over work from each other.

Depending on the data and the number of processes, the subdivision of

[1] Obviously, further parts can be redistributed, such that the slowest slave only requires to process 90% or 85% (etc.) of its region.

image regions could be done along the outermost (as already done for initial distribution) or the second last[2] image dimension.

Obviously it is not possible to integrate dynamic load balancing into the parallel registration framework without creating some new parallel overhead. As seen in Figure 7.22, an important reason that lowers efficiency for a large number of processes is synchronization overhead due to slow nodes. Therefore, the performance increase due to better loaded nodes is likely to far outweigh this newly introduced overhead.

### 8.1.2 Optimizing Initialization Phase

#### IP-Multicast

During initialization, the moving image is broadcasted to all slaves. The current implementations uses the `MPI_Bcast()` method to do so. `MPI_Bcast()` internally builds up on point-to-point communications. A tree structure allows to broadcast a message in $T_M \cdot \log_2(N)$ seconds (where $N$ is the number of processes and $T_M$ the time to send the message over the network once). It has been shown, that for large messages (such as messages consisting of image data definitely are), the use of IP-Multicast can considerably reduce transmission time, since only one message (instead of $\log_2(N)$) has to be sent by the root node [32, 33]. Using IP-Multicast can therefore reduce the initial communication overhead which inherently limits the achievable speedup according to Amdahl's law. A free implementation that integrates IP-Multicast into MPI can be found at `http://www.boulder.swri.edu/~ptamblyn/ais/`[3].

#### Data Compression

As stated in Section 3.3, compressing data prior to sending it over the network can sometimes increase throughput. This, however, greatly depends on the kind of data, the available computing power, and the communication subsystem. In case a configuration of these components speaks in favor of compression, a user can override the image send and receive methods of the `RegistrationCommunicator` by versions that compresses data prior to transmission (send) and uncompresses them after reception (receive). The goal of adding such compression methods is obviously to reduce the time spent in the initialization phase. Therefore, it is of particular interest if a large number of processors is involved in the computation.

[2] I.e. $x$ in 2D and $y$ in 3D.
[3] State 21.09.2004

### Parallel Gradient Image Calculation

As can be seen in Section 7.5.1, Figure 7.21, a considerable amount of time during initialization is consumed by gradient image calculation. Each slave calculates the whole gradient image. Consequently this work is carried out $n$ times (where $n$ is the number of slave processes). This is basically equivalent to non-parallelizable code which inherently limits the achievable speedup according to Amdahl's law. Since the gradient image filter used in the parallel framework is streaming capable (i.e. allows to treat images region by region), the work of calculating the gradient image could be parallelized, such that each slave only calculates one n-th ($\frac{1}{n}$) of the gradient image and broadcasts its part to the other slaves. Provided, that the interconnection network is fast, the speedup due to this parallelization probably outweighs the time lost be exchanging image data among the slaves[4].

## 8.2 Parallel Registration Framework and Shared Memory Systems

ITK has some built in support for parallel computing on shared memory systems. At the time of this writing, one metric function (`MatchCardinalityImageToImageMetric`) provides support for multi-threaded processing. In analogy with the `DistributedImageToImageMetric`'s `DistributedGetValue()`, `BeforeDistributedGetValue()` and `AfterDistributedGetValue()` methods, it owns the methods `ThreadedGetValue()`, `BeforeThreadedGetValue()` and `AfterThreadedGetValue()`. These methods carry out exactly the same work as their counterparts in the distributed framework.

Analyzing the architecture has shown, that a threaded metric superclass could be introduced, similar to the `DistributedImageToImageMetric` which serves as the superclass for all the metric classes of the parallel framework for distributed memory systems. Let us assume such a threaded metric superclass exists, is called `ThreadedImageToImageMetric` and derives from `itk::ImageToImageMetric`. Then, a special version of the

[4] An example should give a rough idea: The calculation time for an image of $130 \times 130 \times 254$ pixels (which yields a gradient image size of roughly 100 MB) took approximately 26 seconds (at ETH). Ideally 100 MB can be sent over a a 100 Mbps network in approximately 8 seconds. On $n = 16$ slaves, the calculation of one gradient image part takes about $26/16 = 1.6$ seconds. Thus, a parallel version ideally calculates the gradient image in $8 + 1.6 = 9.6$ instead of 26 seconds.

`DistributedImageToImageMetric` could be created, that keeps an instance of such a `ThreadedImageToImageMetric`. Within the slaves, the fixed and the moving image, as well as the interpolator could be assigned to the threaded metric member[5]. The `DistributedGetValue()` method would then become a simple wrapper around the `ThreadedGetValue()` method of the threaded metric that makes sure the results are copied into the data structures needed to transmit them over the network. The pre- and post processing methods could analogously be wrapped. That way, any threading capable metric could be equipped with the functionality of the distributed metric.

Instead of creating a new metric function deriving from the `DistributedImageToImageMetric`, new metric functions could be derived from the `ThreadedImageToImageMetric` and then be made available to the parallel registration framework for distributed memory systems by this wrapper module.

[5]Note that this does not involve copying large amounts of data, since only pointers to the images have to be copied.

# Chapter 9

# Conclusion

During this project, a generic parallel registration framework for distributed memory systems, such as clusters of workstations, has been developed. The framework builds upon and extends the ITK library, one of the most prevalent software toolkits in the area of image registration. Thereby the developed methods hide the parallelization details from the user and can thus be applied without in-depth knowledge about parallel computing. The multitude of modules available in the ITK library together with their universal character allows to embed the parallel framework into a large variety of different applications.

A generic module for distributed metric calculation has been developed, which allows to implement a multitude of intensity based metric functions with little effort. Exemplarily, metric functions that define mean squares as well as mutual information measures have been realized.

An abstract communication layer has been introduced to ensure that the methods can be deployed on different hardware architectures with little modifications. This layer has been implemented for the MPI standard, which allows the use of the unmodified framework on most distributed memory systems. Finally, a caching mechanism has been proposed to deal with memory intensive problems.

An extensive evaluation in more that 1000 experiments showed that:

- the framework can be applied in various registration problems, including:
  - two and three dimensional problems.
  - rigid and deformable problems.
  - inter- and intra-modal problems.

- high speedup values up to a large number of processors can be achieved in some problems.
- the framework scales well in most problems.
- in problems demanding for large amounts of memory, parallel efficiencies of more than 100% can be reached compared to sequential programs that cause disk swapping.
- extensions that improve performance in special situations can be added at little cost.
- the framework can be used in Windows as well as Unix clusters.

The evaluation, however, also revealed, that particularly in three dimensional problems, a long initialization phase inherently limits the reachable speedup and that the lack of dynamic load balancing negatively affects performance in heterogeneous clusters as well as in clusters that are not free of interfering load. The need for dynamic load balancing as well as the optimization of the initialization phase could be addressed by future projects. In addition, such projects could tackle problems related to the use of the parallel framework in shared memory systems.

In dieser Arbeit wurde ein generisches Framework entwickelt, welches parallele Bildregistrierung in Systemen mit verteiltem Speicher, wie zum Beispiel Workstation-Clusters, ermöglicht. Das Framework basiert auf und erweitert die ITK Bibliothek, welche eines der weitestverbreiteten Software-Toolkits im Bereich der Bildregistrierung darstellt. Die entwickelten Methoden verbergen die Parallelisierungsdetails vor dem Benutzer, so dass sie auch ohne tiefgründiges Wissen über Parallel-Computing eingesetzt werden können. Der universelle Charakter und die Vielzahl der von der ITK Bibliothek zur Verfügung gestellten Module erlauben es, das Framework in eine grosse Auswahl verschiedener Anwendungen zu integrieren.

Für die verteilte Berechnung der Metrikfunktion wurde ein generisches Modul entwickelt, welche es ermöglicht, mit geringem Aufwand eine Vielzahl von intensitätsbasierten Vergleichsfunktionen zu implementieren. Exemplarisch wurden Metriken realisiert, welche auf den Massen “Mean-Squares” und “Mutual Information” beruhen.

Ein abstrakter Kommunikationslayer wurde eingeführt, um sicherzustellen, dass lediglich kleine Modifikationen nötig sind um das Framework auf den

verschiedensten Hardware-Architekturen einzusetzen. Dieser Layer wurde für den MPI-Standard implementiert, wodurch die meisten Systeme mit verteiltem Speicher abgedeckt sind und sich eine Modifikation des Frameworks ganz erübrigt. Schliesslich wurde ein Caching-Mechanismus vorgeschlagen und umgesetzt, um auch memoryintensive Probleme behandeln zu können.

Eine umfangreiche Evaluation, bestehend aus mehr als 1000 Experimenten, hat gezeigt, dass:

- das Framework in verschiedenartigen Bildregistrierungsproblemen eingesetzt werden kann, darunter gehören:
  - zwei- und dreidimensionale Probleme.
  - Probleme welche auf starren Transformationen beruhen, sowie solche, in welchem die Transformation durch ein allgemeines Deformationsfeld gegeben ist.
  - inter- und intra-modale Probleme.
- in gewissen Problemen hohe Speedup-Werte auf bis zu einer grossen Anzahl von Prozessoren erreicht werden können.
- das Framework in den meisten Problemen gute Skalierungseigenschaften aufweist.
- in Problemen mit grossem Speicherbedarf im Vergleich zu sequenziellen Programmen, welche zu Disk-Swapping führen, eine parallele Effizienz von mehr als 100% erreicht werden kann.
- mit kleinem Aufwand Erweiterungen eingebracht werden können, welche die Performance in speziellen Situationen steigern.
- das Framework sowohl in Windows- als auch in Unix-Clustern einsetzbar ist.

Allerdings hat die Evaluation auch aufgezeigt, dass besonders in dreidimensionalen Problemen eine lange Initialisierungsphase den erreichbaren Speedup inhärent limitiert, und dass das Fehlen einer dynamischen Lastverteilungsstrategie die Performance in heterogenen Clustern, sowie in Clustern mit unterschiedlich belasteten Knoten, negativ beeinträchtigt. Zukünftige Projekte könnten sich deshalb sowohl der Entwicklung eines dynamischen Lastverteilsungsmechanismus als auch der Optimierung der Initialisierungsphase widmen. Ausserdem könnten solche Projekte Probleme angehen, welche sich im Zusammenhang mit der Verwendung des Frameworks in Shared-Memory-Systemen ergeben.

# Appendix A

# Original Task Description

## A.1 Introduction

Image registration is the process of establishing a point-by-point correspondence between two images of a scene. The images can thereby be acquired by different sensors (inter-modal registration) or by the same sensor in different points in time (intra-modal registration). The image registration task can be further divided into rigid and non-rigid registration depending on the transform allowed to map one image onto the other.

Image registration is applied in several fields, such as in medical imaging, in stereo vision applications, for motion analysis, object localization or image fusion.

From the algorithmic point of view, image registration is basically an optimization problem. Usually a cost function is used, that defines the quality of the correspondence between the images depending on their relative position. On this function, a global optimum is searched. In the course of the optimization, usually one image is kept fixed (fixed image) and the other one (moving image) is transformed until an optimal correspondence is found.

Different software packages, libraries and frameworks exist that carry out image registration. Some of the most prevalent tools are the Automated Image Registration (AIR) package ([1]), the Flexible Image Registration Toolbox (FLIRT) ([2]), the VTK CISG Registration Toolkit ([3]), the Image Processing Toolbox of Mathworks (for Matlab) and the Insight Segmentation and Registration Toolkit (ITK) [4]. Among these projects, ITK is the most generic and extensive approach. Apart from the Image Processing Toolbox of Mathworks, all the above mentioned packages are open source projects that are freely available. However, only the Insight Segmentation and Registration Toolkit can be used in commercial applications.

Particularly in 3D applications, image registration can be a computationally intensive task as shown in [16] and [2]. The main problems are the sometimes large amounts of data, e.g. a $\mu$CT image of a mouse femur as used in [16] has a size of approximately 500MB. Since a cost function often takes into account every single pixel (or voxel), such data amounts result in long calculation times. What makes things even worse is the fact, that for large images computers soon run out of available physical memory. The ensueing disk swapping causes considerable performance loss and an increase in calculation time.

The usage of distributed computing techniques can solve both, memory as well as performance problems. There exist several approaches that address the image registration problem, and optimization problems in general, with parallel computing techniques ([44], [14], [45], [19], [25], [7], [46], [47]). Parallization methods for optimization tasks are divided into coarse grained and fine grained methods. While in coarse grained methods multiple independent function evaluations are executed in parallel, fine grained methods calculate the basic computational steps in parallel. According to [7] the parallel efficiency for coarse grained methods is usually better, and best performance and scalability can be achieved by combining coarse and fine grained methods. When addressing the parallel computation of the image registration task, usually coarse grained approaches based on evolutionary algorithms are applied ([44], [14], [45], [19]). Evolutionary algorithms are often used in optimization problems and are well suited for a coarse grained distribution, since the same function has to be evaluated several times for different samples.

However, memory issues still remain critical when applying coarse grained methods. Furthermore, coarse grained methods usually scale only within a very limited range. Several voxel based registration techniques appear to be well suited for fine grained parallel computation, thus allowing to define promissing methods for addressing scalability and memory issues.

In this thesis, distributed computing techniques should be integrated into the ITK library and the performance increase as well as the scalability of fine grained methods should be investigated. The ITK library was chosen because of the above mentioned advantages over other libraries.

## A.2 The ITK Library and Distributed Image Registration

ITK is an open source C++ library mainly used in medical imaging for image segmentation and registration. It was founded in 1999 by the US Library of Medicine of the National Institutes of Health.

The basic image registration framework consists mainly of *transform*, *interpolator*, *optimizer*, and *metric* classes. All of these components are necessary to carry out image registration. The transform is needed to define how the moving image can be transformed (only translations, rigid transforms, affine transforms, non-linear transforms etc.) in the course of a registration run. The metric class is the cost function, that defines the quality of the correspondence dependent on the current transform parameters. Since the transformed pixels of the moving image will not exactly fit onto the pixel grid of the fixed image, interpolation is necessary. The optimizer defines a strategy to search for a global optimum in the cost function. A further class serves as a controller to the mentioned modules and sets up all the necessary interconnections between them (see figure A.1). Several optimizers, transforms, interpolators and metrics can be chosen from.

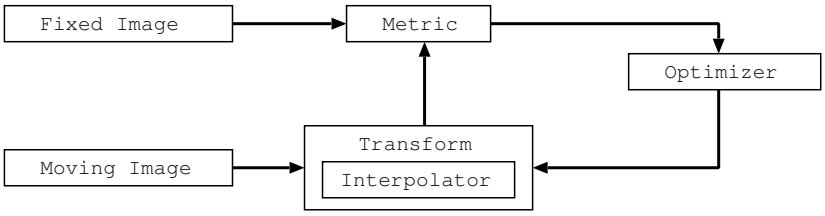



Figure A.1: Modules of the ITK Library.

Besides the basic image registration framework, ITK contains functionality for deformable and for model based registration. However, these parts should not be considered during this project.

All optimizers present in ITK (see [4]) have an optimization strategy that works iteratively. It thereby often steps along a direction related to the gradient of the cost function. Therefore, for each iteration, the cost function has to be evaluated and (usually) its gradient has to be computed. For large amounts of data, the calculation of the cost value and the gradients becomes time consuming. We therefore propose an approach that distributes the load of computing the image metrics.

Within the toolkit, there are 11 implementations of cost functions at the

time of this writing. These are:

- GradientDifferenceImageToImageMetric
- MattesMutualInformationImageToImageMetric
- MeanReciprocalSquareDifferenceImageToImageMetric
- MeanSquaresImageToImageMetric
- MutualInformationImageToImageMetric
- NormalizedCorrelationImageToImageMetric
- CorrelationCoefficientHistogramImageToImageMetric
- MeanSquaresHistogramImageToImageMetric
- MutualInformationHistogramImageToImageMetric
- NormalizedMutualInformationHistogramImageToImageMetric
- KullbackLeiberCompareHistogramImageToImageMetric

Whereas the metric functions related to mutual information can be applied in inter- as well as in intra-modal registration, all the other functions are only applicable in intra-modal registration.

The most common cost function is the mean squares metric, which pixelswise sums up the squared differences of the intensity values. The mean reciprocal metric works on a pixel by pixel basis as well, however, instead of adding the squared differences, the differences are added after passing them through the bell-shaped function $\frac{1}{1+x^2}$. The gradient difference metric compares image gradients instead of intensities, the normalized correlation metric is principally a cross correlation approach and the mutual information metrics are based on a statistical method that minimizes the joint information of the overlaid images measured by entropies of the intensity distribution functions. Those metrics carrying the term 'Histogram' in their name are all derived from a common superclass and operate on joint histogram data of the two images. The computational complexity differs greatly for different metrics. For most optimization strategies, the metric value as well as the derivative at a given position have to be evaluated. Obviously, the gradient calculation becomes more time-consuming the more transform parameters are involved (i.e. for rigid transforms, gradient calculation is less complex as in the case when general non-linear transforms are allowed).

While some metrics (like the mean squares or the mean reciprocal metric) only contain local data dependencies, this does not hold for other metrics, what makes it more difficult to calculate them in parallel.

As already stated before, within ITK it seems to be reasonable to distribute the load of metric calculations. Other approaches, like distributing the optimization process are possible as well but show some disadvantages. Most of the optimizers are based on a gradient descent idea. In principle, calculating these gradients can easily be distributed in a coarse grained manner by calculating the cost function for some slightly displaced positions on different machines. This approach makes sense for some cost functions, where analytical solutions for the gradient calculation are not available or not efficient. However, the scalability for this technique is very limited and each involved computer needs to work on the whole images which might cause memory problems as well as unnecessary high network traffic. For a 3D rigid registration working with 6 degrees of freedom (i.e. transform parameters), for example, 13 computers could be involved (for each parameter a displacement in both directions has to be calculated). In total, the calculations would be carried out on 13 copies of the fixed as well as the moving image and these copies would have to be distributed over the network. Other coarse grained methods raise the same problems.

To allow flexible scalability and to avoid the necessity of several image copies, the problem should be tackled by distributing the calculation of the image metrics in a fine grained manner. This essentially means to divide the image into partial images and to compute the metric value based on calculations applied to them. The single calculations should thereby be carried out on computers connected by a network. Except of the overhead, only one copy of each, the fixed and the moving image has to be distributed in this setup. Network traffic still might become a serious problem in such an environment, because of the large data amounts and the permanent displacement of the moving image (which probably makes up most of the overhead mentioned above). A good data management will be indispensable and it could be worth considering methods like data compression, or maybe even prediction to tackle the problem. Based on an architecture that allows distributed metric calculation, a combination with coarse grained methods could be interesting as proposed by [7]. However, this is not the aim of this project.

## A.3 Problem Statement

The master thesis consists of the following main tasks:

- specification of the detailed requirements of the implemented software

(functional specification),

- design of an architecture for distributed image registration that fits into the existing toolkit (ITK),
- implementation of the proposed architecture,
- investigation of the performance increase and the scalability.

### A.3.1 Requirement Specification

In order to set up a clear requirement specification the student starts with the definition of a clear interaction and service model. This model will be the foundation for identifying the possibilities for parallelization and distribution, but it will also help in delimiting the scope of the thesis. The deduced requirement specification will enfold the set of registration problems that have to be covered by the distributed registration methods developed during this project, i.e. inter– vs. intra–modal registration, rigid vs. non–rigid registration,etc. It further comprises the requirements on scalability, on the amount of data that is to be supported, and on hardware profiles e.g. on constraints on heterogenous computers systems differing in processing power and memory, or on constraints and requirements of the communication system.

### A.3.2 Architecture Design

An architecture for distributed image registration should be designed that fits smoothly into the existing part of ITK. The basic idea is to consider fine grained methods, i.e. the parallel calculation of cost functions. Thereby the overall architecture of the toolkit should be taken into account, allowing flexible extensions of the proposed architecture in future. Such an extension could be the combination of the distributed metric calculation with coarse grained methods or the development of further distributed cost functions.

Besides pure computational speedup, the architecture should take into account memory issues and network traffic. Memory issues have shown to be a major problem in [16], since for large amounts of data the physically available memory has been exceeded. Because of these large data amounts, data exchange over the network will most likely become a serious issue as well.

The design of such an architecture involves the following steps:

- studying the architecture of ITK and the characteristics of all cost functions used in the basic image registration framework of the toolkit.

Methods only applied in deformable and model based registration should only roughly be looked at, keeping future extensions in mind. The cost functions should mainly be investigated with respect to their feasibility for parallel calculation (local vs. global data dependencies) and with respect to their relations to other methods within the toolkit. Such relations can be similar mathematical characteristics as well as relations within C++ (inheritance).

- based on above investigation of ITK as well as the previously defined requirements, a design should be specified that exactly states which parts of ITK should be implemented in a distributed manner and how this should be done.

- above design should be refined by stating how to deal with issues concerning the underlying hardware as they are to be expected according to the requirement specification. Particular attention should be paid to data management issues.

- based on these considerations an architecture should be proposed that addresses all issues discussed before.

### A.3.3 Implementation

The previously specified architecture should be implemented such that it runs on all platforms supported by ITK. The choice of the interprocess communication infrastructre will be decided upon an anaylysis of the requirements and the design choice. A lightweight solution is however suggested. During the whole implementation process, tests on different platforms should be carried out.

The coding styles defined in the ITK Style Guide will be followed whenever possible. Making the code official part of ITK is not the main goal of the project since it is difficult to estimate the work necessary to do so. Therefore, only in case there is enough time left and the procedure proves not to be extraordinarily time consuming this work can be done within the project. However, the code should be written such that an integration into ITK can later be achieved.

### A.3.4 Evaluation

The implemented architecture should be investigated with respect to scalability and parallel efficiency for scenarios that cover the previously defined

requirements. Memory consumption as well as computation time should be considered theoretically as well as in experiments.

A sophisticated methodology has to be thought of that allows to quantitatively assess the implementation. This methodology should define which parts are evaluated, to what they are compared and how the measurements are carried out.

## A.4 Organization of the Work

- The student has to arrange weekly meetings with his supervisor. In these meetings the student is exepeced to inform on the progress of the work, on encountered problems and proposed solutions. It is mandatory that the student is well prepared for these meetings.

- By the end of the second week the student is expected to present the final timetable of the thesis. The rough timetable defined in advance should be refined and discussed with Prof. Gergel. The timetable has to identify important milestones of the project. Milestones are sub–goals that, upon reaching them, may implying decisions on further proceeding.

- By the end of the first 6 weeks the student has to issue a pre–version of the table of contents of the thesis documentation. It should be discussed with Prof. Gergel and sent to the supervisors at ETH such that they can add remarks.

- The student will have to make a short report at about the end of month three. The report will be presented to Prof. Platter. It has to give a brief overview on the current state of the project and the ongoing and planned steps. (The report will most likely be hold as a telephone conference.)

- The documentation of the thesis is carried out in parallel to the project's progress. Two intermediate reports should be issued and sent to the assistants at ETH, the first after 2 and the second after 4 months. The documentation can be made with any word processor, though we recommend to use LaTeX.

- After the end of the thesis, when back in Switzerland, the student is to present his achievements in a 15 minutes talk at a group/institute meeting at TIK. The presentation slides have to be sent to the assistants at ETH by the end of the project.

- The documentation of the project is to be written in English. The abstract and the executive summary however need to be written in both English and German. In the appendix the taks description and the timeplan have to be included. The student is to deliver 4 copies of the documentation.
- The whole work has to be archived on CDROM (including presentation slides). Please check that the documentation, the code, and all sources are available in a printable, or executable, and useable shape.

# Appendix B

# Time Table

**Week 1-4: 5 April - 2 May**

- Refine this time table.
  - *18 April: Hand in the refined time table.*
- Prepare a presentation to be held at NNSU
  - *15 April: Hold the presentation.*
- Set up the development environment:
  - make sure there is a linux and a windows environment available
  - install software
    - LaTeX
    - compiler (gcc, VC++)
    - ITK and related sofware (cmake, cvs?)
- Find out about related work:
  - image registration
  - which job to assign to which processor in a distributed environment
  - communication means in a distributed environment
- Define the requirement specification according to the task description.

**Week 5: 3 May - 9 May**

Analyze the ITK architecture.

- *9 May: Hand in the table of contents of the report.*

- ***Milestone:*** *All the information necessary to define how the final solution should look like is gathered.*

**Week 6-7: 10 May - 18 May**
Define an architecture for the distributed part that fits into the existing toolkit.

- ***Milestone:*** *Based on above defined architecture the exact requirements for the communication subsystem can now be seen.*

**Week 7: 19 May - 23 May**
Evaluate different methods to implement the communication means and choose a method suitable for the proposed architecture.

- *23 May: Hand in the first intermediate report.*

**Week 8-13: 24 May - 4 July**
Implementation of the basic architecture (framework for communication and distribution of classes that should be calculated in parallel)

- *Week 13: Phone conference with professor Plattner.*

**Week 14-16: 5 July - 25 July**
Implementation of one scenario as a proof of concept for the previously defined and implemented architecture.

- *25 July: Hand in the second intermediate report.*
- ***Milestone:*** *If the proof of concept was successful and enough time is remaining, proceed according to the timetable. If there is not enough time left, proceed with the evaluation such that at least one scenario is evaluated. If the proof of concept was not successful, reconsider the architecture.*

**Week 17-19: 26 July - 15 August**
Implementation of further scenarios.

**Week 20-21: 16 August - 29 August**
Definition of a methodology for the evaluation of the architecture

**Week 22-23: 30 August - 12 September**
Make theoretical calculations and carry out the experiments according to the defined evaluation methodology.

**Week 24-26: 13 September - 3 October**
Buffer for delays as well as time to prepare the presentation.

- *Week 26: Hand in the final report.*
- *Week 26: Hand in the presentation slides.*

# Bibliography


[1] *Automated Image Registration*, http://bishopw.loni.ucla.edu/, 5 Feb. 2004

[2] Jenkinson M., Smith S.: *Optimisation in Robust Linear Registration of Brain Images*, Technical Report TR00MJ2, Oxford Centre for Functional Magnetic Resonance Imaging of the Brain (FMRIB), 2000

[3] Hartkens D., Rueckert T., Schnabel J.A., Hawkes D.J., Hill D.L.G.: *VTK CISG Registration Toolkit: An Open Source Software Package for Affine and Nonrigid Registration of Single- and Multimodal 3D images*, Bildverarbeitung fr die Medizin, 2002

[4] Ibanez L., Schroeder W., Ng L., Cates J.: *The ITK Software Guide*, Kitware Inc., 2003

[5] Squyres J. M., Lumsdaine A., Stevenson R. L.: *A toolkit for parallel image processing*, SPIE Annual Meeting, San Diego, 1998

[6] Seinstra F. J., Koelma D.: *Transparent Parallel Image Processing by way of a Familiar Sequential API*, International Conference on Pattern Recognition (ICPR'00), Volume 4, Barcelona, Spain, 2000

[7] Eldred M.S., Schimel B.D.: *Extended Parallelism Models for Optimization on Massively Parallel Computers*, Proceedings of the 3rd World Congress of Structural and Multidisciplinary Optimization, Buffalo, NY, 1999

[8] Pacheco P. S.: *Parallel Programming with MPI*, Morgan Kaufmann Publishers, Inc., USA, 1997

[9] Downton A., Crookes D.: *Parallel Architectures for Image Processing*, Electronics and Communication Engineering Journal, vol. 10, No. 3, pp.139-151, 1998

[10] Hwang K., Zhiwei X.: *Scalable Parallel Computing*, The McGraw-Hill Companies, USA, 1998

[11] *The Visual Human Project*, http://www.nlm.nih.gov/research/visible/visible_human.html, accessed 6 July 2004

[12] The Insight Consortium: *ITK Style Guide*, www.itk.org/cgi-bin/cvsweb.cgi/Insight/Documentation/Style.pdf?cvsroot=Insight&rev=1.4, accessed 28 April 2004

[13] *An Introduction to the Insight Segmentation & Registration Toolkit*, http://www.itk.org/cgi-bin/viewcvs.cgi/README.html?cvsroot=Insight&rev=1.17, accessed 28 April 2004

[14] Chalermwat P.: *High Performance Automatic Image Registration for Remote Sensing*, Dissertation, George Mason University, Fairfax, Virginia, 1999

[15] Rohlfing T., Maurer C. R.: *Nonrigid Image Registration in Shared-Memory Multiprocessor Environments With Application to Brains, Breasts, and Bees*, IEEE Transactions on Information Technology in Biomedicine, Vol. 7, No. 1, 2003

[16] Moser S., Kuhn M.: *Development of Intra-Modality Image Registration Methods for Assessment of Micro-Structural Bone Adaption*, Student Thesis, University of Calgary, Canada, 2003

[17] Christensen G. E., Miller M. I., Vannier M. W., Grenander U.: *Individualizing Neuroanatomical Atlases Using a Massively Parallel Computer*, IEEE Computing, vol. 29, no. 1, 1996

[18] Jiang T., Fan Y.: *Parallel Genetic Algorithm for 3D Medical Image Analysis*, IEEE Int. Conf. on System, Man and Cybernetics, Tunesia, 2002

[19] Salomon M., Perrin G.R., Heitz F.: *Parallelizing differential evolution for 3D medical image registration*, research report, 2000

[20] Thevenaz P., Ruttimann U. E., Unser M.: *A Pyramid Approach to Sub-Pixel Registration Based on Intensity*, IEEE Transactions On Image Processing, vol. 7, no. 1, 1998

[21] Warfield S., Jolesz F. A., Kikinis R.: *A High Performance Computing Approach to the Registration of Medical Imaging Data*, Parallel Computing, Vol. 24, no. 9-10, pp. 1345-1468, 1998

[22] Baowei F., Wietholt Ch., Clough A. V., Dawson Ch. A., Wilson D. L.: *Automatic Registration and Fusion of High Resolution Micro-CT and Lung Perfusion SPECT Images of the Rat*, Engineering in Medicing and Biology Society, Proceedings of the 25th Annual International Conference of the IEEE, 2003

[23] Langis Ch., Greenspan M., Godin Guy: *The Parallel Iterative Closest Point Algorithm* Proceedings of the Third International Conference on 3D Digital Imaging and Modeling (3DIM), Quebec, Canada, 2001

[24] Li H, Zhou Y. T., Chellappa R.: *SAR/IR Sensor Image Fusion and Real-Time Implementation*, Signals, Systems and Computers, 1995

[25] Sermesant M., Clatz O., Li Z., Lantri S., Delingette H., Ayache N.: *A Parallel Implementation of Non-Rigid Registration Using a Volumetric Biomechanical Model*, WBIR, Philadelphia, PA, USA, 2003

[26] Chan K. L., Tsui W. M., Chan H. Y., Wong H. Y., Lai H. C.: *Parallelising Image Processing Algorithms*, IEEE Tencon'93, Bejing, China, 1993

[27] Riyait V. S., Lawlor M. A., Adams A. E., Hinton O., Sharif B.: *Real-Time Synthetic Aperture Sonar Imaging Using a Parallel Architecture*, IEEE Transactions on image processing, VOL. 4, NO. 7, July 1995

[28] Nicolescu C., Jonker P.: *Parallel low-level image processing on a distributed-memory system*, IPDPS Workshops, 2000

[29] Young J. K., Byung K. K.: *Load Balancing Algorithm for Parallel Vision System using COTS PCs and Networks*, ServiceRob 2001, Santorini, Greece, 2001

[30] Hastings S., Kurc T., Langella S., Catalyurek U., Pan T., Saltz J.: *Image Processing for the Grid: A Toolkit for Building Grid-enabled Image Processing Applications*, The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), 2003

[31] Erenyi I., Fazekas Z.: *Mapping Image Processing Algorithms onto Abstract Multiprocessors*, IEEE Colloquium on 'Mathematical Modelling and Simulation of Industrial and Economic Processes' (Digest No.1994/168), 1994

[32] Yuan X., Daniels S., Faraj A., Karwande A.: *Group Management Schemes for Implementing MPI Collective Communication over IP-Multicast*, The 6th International Conference on Computer Science and Informatics, Durham, NC, 2002

[33] Chen H. A., Carrasco Y. O., Apon A. W.: *MPI Collective Operations over IP Multicast*, Proceedings of the 3rd Workshop on Personal Computer-based Networks of Workstations (PC-NOW 2000), 2000

[34] Hanushevsky A., Trunov A., Cottrell L.: *Peer-to-Peer Computing for Secure High Performance Data Copying*, Computing in High Energy and Nuclear Physics CHEP'01, Beijing, 2001

[35] *How Network Compression Can Increase WAN Capacity Up to 4X*, http://www.packeteer.com/resources/prod-sol/Acceleration.pdf, accessed 27 April 2004

[36] LeMoinge J.: *Towards a Parallel Registration of Multiple Resolution Remote Sensing Data*, Proceedings of IGARSS'95, Firenze, Italy, 1995

[37] Shivaratri N. G., Krueger P., Singhal M.: *Load Distributing for Locally Distributed Systems*, IEEE Computer 25, pp 33-44, 1992

[38] Fleury M., Sava H., Downton A. C., Clark A. F.: *A Real-Time Parallel Image Processing Model*, Proceedings of IEE 6th International Conference on Image Processing and its Applications IPA'97, volume 1, pages 174-178, 1997.

[39] Lee C., Hsu C. C., Yai S. B., Huang W. C., Lin W.C.: *Distributed Robust Image Mosaics*, International Conference on Pattern Recognition ICPR98, 1998

[40] Giancaspro A., Candela L., Lopinto E., Lore V. A., Milillo G.: *SAR Images Co-registration Parallel Implementation*, Geoscience and Remote Sensing Symposium IGARSS, 2002

[41] Message Passing Interface Forum: *A Message Passing Interface Standard*, Technical report, 1995

[42] Kumar V., Grama A., Gupta A., Karypis G.: *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin-Cummings Publishing Company, Inc., Reedwood City, Canada, 1994

[43] *MPE - Multi Processing Environment*, http://www-unix.mcs.anl.gov/mpi/www/www4/MPE.html, accessed 29 September 2004

[44] Chalermwat P., El-Ghazawi T., LeMoigne J.: *GA-based Parallel Image Registrataion on Parallel Clusters*, IPPS/SPDP Workshops, San Juan, Puerto Rico, USA, 1999

[45] Chalermwat P., El-Ghazawi T.: *Multi-Resolution Image Registration Using Genetics*, ICIP, Kobe, Japan, 1999

[46] Pardalos P.M., Xue G., Panagiotopoulos P.D.: *Parallel Algorithms for Global Optimization Problems*, Solving Combinatorial Optimization Problems in Parallel, 1996

[47] Gergel V.P., Strongin R.G.: *Parallel Computing for Globally Optimal Decision Making*, PaCT, Novosibirsk, Russia, 2003

# Index