Optimizing and parallelizing X-ray spectral analysis software for shared memory systems

Utrecht University

Department of Mathematics

Tiemen Schut

February 5, 2010

Supervisor at Utrecht University: Prof. Dr. R.H. Bisseling
Supervisor at SRON: Dr. J. Kaastra

Abstract

This thesis discusses the optimization and parallelization of an existing program written in Fortran 90. After profiling the program with different use-cases, two parts that will be optimized are selected. One of these is essentially a matrix-vector multiplication, the other calculates absorption of photons due to photoionization. Parallelization will be done using openMP, since the program must be optimized for shared memory systems and incremental parallelization must be supported.

At the end of this thesis, execution time is reduced to 89% of the original for the use-case that used the optimized parts least of all use-cases and using only one processor, improving to 28% of the original for the use-case that used the optimized parts most and using four processors.

Preface

This thesis is written to complete my master study Scientific Computing at the Department of Mathematics at Utrecht University. The research presented was done at the High Energy Astrophysics (HEA) division of the Netherlands Institute for Space Research (SRON).

During my internship at SRON, I researched, implemented, and tested optimization and parallelization of an existing computer program used to perform spectral analysis on X-ray spectra. Reducing execution time of this program is important, as more and more data has to be processed with increasing precision of research instruments.

The research done resulted in a faster version of SPEX, which is already used within SRON.

I would like to thank SRON for giving me this opportunity. Not only has SRON provided a place and subject for my research, SRON also supported me during the full course of my study, allowing me to keep a regular job while my working hours had to be adjusted regularly to support changing schedules.

My research was supervised by Rob Bisseling from the Department of Mathematics and Jelle Kaastra from SRON, both of whom I would like to thank for their support and constructive feedback. I would like to thank them particularly for having the patience to read through an early version of this thesis, which turned out to be a *bit* longer than it was supposed to.

I would also like to thank Jelle de Plaa, who provided feedback and always had time when I had a question. Thanks to Rob Detmers, Yan Grange, and Jelle Kaastra for providing use-cases for SPEX, allowing me to easily test SPEX with different data and parameters.

Finally, I would like to thank Ashley Cowles for being there, even when I was often tired and not much fun to be around.

Contents

List of Abbreviations

AGN	Active Galaxy Nuclei
ARB	Architecture Review Board
ED	Engineering Division
EOS	Earth Oriented Science
EPIC	European Photon Imaging Camera
ESA	European Space Agency
FITS	Flexible Image Transport System
GCC	GNU Compiler Collection
HEA	High Energy Astrophysics
LEA	Low Energy Astrophysics
MPI	Message Passing Interface
NWO	Nederlandse Organizatie voor Wetenschappelijk Onderzoek - Dutch Organization for Scientific Research
openMP	open Multi-Processing
RAM	Random-Access Memory
SAC	Science Advisory Committee
SPEX	Spectral X-ray analysis
SPMD	Single Program Multiple Data
SR&T	Sensor Research & Technology
SRON	Space Research Organization Netherlands
XSPEC	X-ray SPEctral fitting package

Part I

Introduction, background, and initial benchmarks

Chapter 1

Introduction

1.1 SRON Netherlands Institute for Space Research

SRON is a Dutch agency under NWO, the Dutch Organization for Scientific Research. SRON develops satellite instruments for both astrophysical and earth-oriented science and performs scientific research using data from these and other instruments. The mission statement of SRON is to provide

> "the ensemble of knowledge and skills, both technically and scientifically, required to perform a principal role in the scientific utilization of space."

1.1.1 Organization and history

Space research in the Netherlands became organized in the sixties mainly in Utrecht, Leiden en Groningen. In 1983, this resulted in the founding of SRON with the former name Space Research Organization Netherlands. This name was changed to SRON Netherlands Institute for Space Research in 2005, keeping the abbreviation "SRON" for familiarity.

The institute has facilities at two locations in the Netherlands. The largest and main one is located in Utrecht next to the University, the other one is located at the University of Groningen.

SRON is divided in five divisions. Three of these are focused on research using observational data, either astrophysical or earth and planetary data, the other two are focused on technological research and development.

1.1.2 Research and development

Two of the five divisions within SRON are focused on research and development. The Engineering Division (ED) focuses on providing skilled manpower and innovative technology for space instrument and sensor research projects. Expert areas are the design of analogue and digital electronics, software design and engineering, design and manufacturing of mechanical components, mechatronics, and assembly.

The Sensor Research & Technology (SR&T) division focuses on developing advanced sensors for upcoming space missions. Having the in-house expertise and facilities to design and manufacture innovative and new sensors enables early access to new observational data for SRON.

1.1.3 Earth and Planetary Science

The division Earth and Planetary Science (EPS) focuses on two main lines of research: the physics and chemistry of the Earth's atmosphere and the dynamics of the solid Earth and oceans. Since a few years EPS also looks at planets other than the Earth. Until recently EPS was called Earth Oriented Science (EOS).

For atmospheric research, the present activities include calibration and interpretation of measurements from the GOME (Global Ozone Monitoring Experiment) and SCIAMACHY (SCanning Imaging Absorption SpectroMeter for Atmospheric CartograpHY) instruments. Both instruments are carried on satellites launched by the European Space Agency (ESA), GOME on the European Remote Sensing Satellite (ERS) and SCIAMACHY on the ENVIronmental SATellite (ENVISAT).

1.1.4 Low Energy Astrophysics

The two astrophysical divisions of SRON are divided by the spectral range of their observations. The Low Energy Astrophysics (LEA) division focuses on the construction and operation of instruments performing astronomical measurements in the infrared and sub-millimeter ranges, i.e. from 2.5 to 1000 μm. Observations in this range can be done by satellite, but also by planes and balloon platforms.

In recent years, LEA has been mostly concerned with the development of the Heterodyne Instrument for the Far Infrared (HIFI). This instrument has been developed by over 25 institutes in 12 countries. SRON is the principal investigator institute for the HIFI instrument, making SRON responsible for the overall management, systems engineering, instrument level assembly and testing, and system-wide quality assurance. HIFI was launched as an instrument on the Herschel Space Observatory on the 14th of May, 2009.

1.1.5 High Energy Astrophysics

The High Energy Astrophysics (HEA) division focuses on the design, development, and building of space instruments for X-ray and Gamma ray astrophysics and subsequently on scientific data analysis from these instruments. Since radiation in the X-ray and Gamma ray spectrum is mostly absorbed by the Earth's atmosphere these observations are always done from space.

Data analysis performed by HEA includes data from the Chandra X-ray observatory named in honor of the Indian-American physicist Subrahmanyan Chandrasekhar and the X-ray Multi-Mirror Mission - Newton (XMM-Newton) named in honor of Sir Isaac Newton. Both satellites were launched in 1999.

Currently HEA is working together with SR&T and the ED on the development of cryogenic microcalorimeters for the future International X-ray Observatory (IXO) mission. IXO will be a new X-ray telescope with joint participation from NASA (National Aeronautics and Space Administration), ESA, and JAXA (Japan Aerospace eXploration Agency). Aim is to launch IXO in the year 2021.

SRON also participates in the ASTRO-H project, which is a planned X-ray observatory developed mainly by JAXA. Launch is currently scheduled for the summer of 2014.

Furthermore, HEA develops and maintains spectral analysis software (SPEX). All

work in this thesis is on SPEX, and more background information on SPEX can be found in Section 1.2.

1.2 SPEX

SPEX is a software package developed at SRON for analysis and interpretation/visualization of cosmic X-ray spectra. Its features include spectral modelling, fitting, graphical display and output to files. The current version of SPEX uses mainly Fortran 90 code, with the exception of some external libraries.

1.2.1 History

In the early seventies, R. Mewe started developing spectral code for X-ray emission of optically thin plasmas at the Space Research Laboratory Utrecht [Mewe, 1974]. In 1985 [Mewe et al., 1985] and 1986 [Mewe et al., 1986], two papers were published focusing on theoretical calculations on X-ray spectra. These works, together with the work by Raymond and Smith [Raymond and Smith, 1977], formed the basis of cosmic X-ray spectroscopy for years.

In the following years the code was further developed, resulting in the XSPEC (X-ray SPEctral fitting) package in 1992. The code was named *meka*, after the lead authors Mewe and Kaastra.

Around this time work on SPEX (SPEctral X-ray analysis) started. Main goal was to create an improved, updated, and more extensive version of the code. SPEX should offer spectral simulation and fitting and offer better possibilities regarding plotting and tabular output of data.

A first version of SPEX was released in 1992 [Kaastra et al., 1996]. This version used updated *meka* code at its core. This first version of SPEX was written in Fortran 77, with the exception of some external libraries. Plotting was done using the pgplot package [Pearson, 1985]. For user input, a menu structure was used.

Shortly after the release of this first version, it was discovered that the measured spectrum of the Centaurus cluster of galaxies was inconsistent with the results predicted by the *meka* code. This resulted in a cooperation with D. Liedahl, producing an updated version of the code [Liedahl et al., 1994], named *mekal*, now named after Mewe, Kaastra and Liedahl.

This version of SPEX lasted for about ten years with only minor updates now and then. However, in 2004 SPEX version 2 was released, where main differences with the previous version are

- This version uses Fortran 90 instead of Fortran 77, which offers more flexibility and backwards compatibility with Fortran 77 if desired.
- The menu structure is replaced by a command structure. This makes it easier for an experienced user to execute complex commands.
- Small updates to the *mekal* code.
- Various additions to the model fitting possibilities.

1.2.2 Usage

Although SPEX is developed and maintained within SRON, it is used by different people all over the world as it is freely available at the SPEX website[1]. This website also offers the complete manual and some simple tutorial files.

1.3 This thesis

The research presented in this thesis does not focus on the functionality of SPEX, but on its implementation. Technological advancement in the field of X-ray spectroscopy results in more detailed spectra and thus more data, resulting in more computationally intensive processing for analysis. However, for the last few years the raw processing power of regular computers - with a single processor - has hardly increased. Instead, computer manufacturers turn to multi-core technology, increased cache sizes and other optimization techniques. However, for a program to optimally utilize these features it has to be written with these features in mind.

The work presented in this thesis attempts to optimize SPEX, using three different real-world analysis runs for benchmarking. The physical background of these use-cases will be discussed briefly in Section 3.3.

1.3.1 Goals of this thesis

The main practical goal of this thesis is to optimize the SPEX program, such that scientists working with the program will be able to analyze data faster. However, to do so, research and experiments will be conducted on mainly the following topics:

Efficient cache usage Cache is relatively small memory close to a processing unit. This is explained in more detail in Section 2.2. Due to the speed of cache memory, it is important to write code that is *cache-friendly*.

Parallel processing As modern computers almost always incorporate two or more processing cores it can be a very lucrative optimization technique to start using multiple cores simultaneously.

1.3.2 Constraints

As always, there are some practical constraints to this project.

First of all, all code should be written in Fortran 90/95, with the exception of used third-party software libraries. This is mainly because of maintenance reasons, because it is easier to maintain code written in a single language, and because a lot of Fortran experience is available at SRON.

On the topic of optimization through parallel programming, one has to keep in mind that the program will mainly run on normal consumer hardware. This means that any parallelism should be implemented for a shared memory system, i.e. a multicore processor system using a single main memory unit.

For testing purposes SRON has a quad-core system available. This means that it is currently impossible to test scaling of a parallel implementation beyond four processors.

[1] http://www.sron.nl/spex

Chapter 2

Background

The intention of this chapter is to give background information on some of the topics discussed in this thesis. This starts with a brief background on cosmic X-ray spectroscopy in relation to SPEX in Section 2.1. This is mainly a physical background for those unfamiliar with cosmic X-ray spectroscopy.

Section 2.2 briefly discusses the possibilities regarding software optimization. Furthermore, it defines some additional constraints to this thesis regarding methods that will or will not be used.

Section 2.3 discusses possibilities for parallel programming in Fortran. It also decides on the best choice for parallel programming in Fortran regarding the research presented in this thesis.

2.1 Cosmic X-ray spectroscopy

Although the spectrum of sunlight was observed through a prism by many, William Hyde Wollaston is the first known to discover dark lines in the solar spectrum in 1802 [Wollaston, 1802]. Wollaston surmised that these dark lines were boundaries between the colors of the rainbow, and since his interest was mainly in the colors himself he did not investigate further.

The dark lines were rediscovered by Joseph von Fraunhofer in 1814 [von Fraunhofer, 1815]. Fraunhofer mapped the positions and labeled the most intense lines with letters of the alphabet. Even though Fraunhofer did not investigate the source of the dark lines further, they are still called *Fraunhofer lines*. Figure 2.1 shows a spectrum of the Sun with the dark Fraunhofer lines marked.

The correct explanation for these Fraunhofer lines is generally attributed to Gustav Robert Kirchhoff in 1859 [Kirchhoff, 1859]. Kirchhoff used a light source which had no dark lines in its spectrum. He let light from this source pass through a flame containing sodium, and again the dark Fraunhofer D lines appeared. This led Kirchhoff to the conclusion that the dark lines in the Sun's spectrum are caused by elements in the Sun's atmosphere absorbing these wavelengths, thus creating dark lines [Kirchhoff and Bunsen, 1863].

There was however no theoretical explanation why certain elements absorb certain wavelengths until Niels Bohr introduced his model for atoms [Bohr, 1913]. The Bohr model describes atoms as a diffuse cloud of negatively charged electrons cir-

cling a small positively charged nucleus. These electrons can only travel around the nucleus in special orbits at a certain discrete set of distances from the nucleus. If an atom now collides with a photon, and the energy of the photon equals the energy between two adjacent orbits of an electron, this electron will absorb the photon and start travelling in a higher orbit, which explains the Fraunhofer lines.

It can be argued that Kirchhoff's experiments were the discovery of spectroscopy, as spectroscopy can be described as the study of the interaction between electromagnetic radiation and matter as a function of wavelength. In cosmic spectroscopy this can be described as looking at a star, galaxy, or some other source through anything (the matter) between source and observer and recording the observed spectrum.

There are two distinct ways to measure an X-ray spectrum. The first is similar to experiments with a prism, but the prism is replaced with a diffraction grating. Such a grating has the same effect as a prism, i.e. the direction of light (or photons in general) that passed through the grating depends on the wavelength (or energy) of the light. In instruments using this measurement technique incoming photons are counted per location, since the location after the grating relates to the energy, thus creating a spectrum.

A grating works the way it does because each opening in the grating will act as a point source, causing the light exiting the grating to be made up of the interfering components from each opening. This also causes one of the possible drawbacks of a grating: harmonics. This means that light with a certain wavelength will not only exit the grating at one certain angle but also at multiples of this angle.

The second way to measure an X-ray spectrum is by using a CCD (Charge-Coupled Device). CCDs are used in consumer digital cameras as well, but in the case of X-ray spectroscopy a CCD can be used to create a spectrum by measuring the energy (or wavelength) of incoming photons.

In both cases the measurements are influenced by the used equipment. For cosmic X-ray spectroscopy, this means that the observed spectrum will be influenced by the instrument. Very likely, the optics on the instrument (mirrors, focusing lenses, etc.) have some influence on the spectrum. This influence on the spectrum is called the *instrument response* and is usually represented by a response matrix. Figure 2.2 tries to clarify this concept.

The usual way of analyzing a measured spectrum is a technique called *forward folding*. With this technique, a model is chosen that represents the observation best. This model is then *folded* with the instrument response. The result of this operation is then compared to the observed spectrum and through a fitting procedure the parameters of the model are varied to get a best solution, usually in the least-squares sense. The parameters of the best-fit solution can then be used to do any necessary scientific analysis.

Refer to [Bland-Hawthorn and Cecil, 1997] for a more detailed and technical background on spectroscopy.

2.2 Software optimization

Software optimization can be seen as the process of modifying software such that quantities like execution speed or memory usage are reduced. Thus software opti-

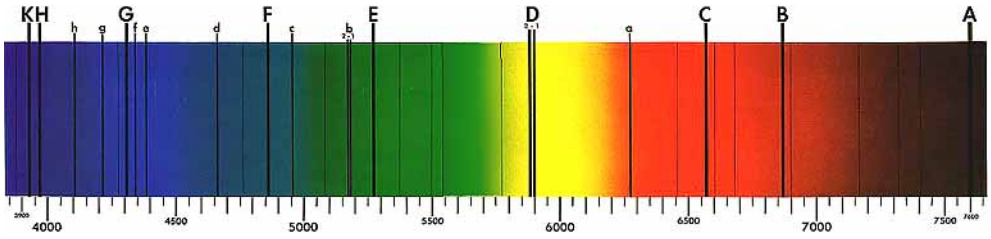

Figure 2.1: The visible light spectrum of the Sun. The dark Fraunhofer lines are clearly visible and marked with letters.

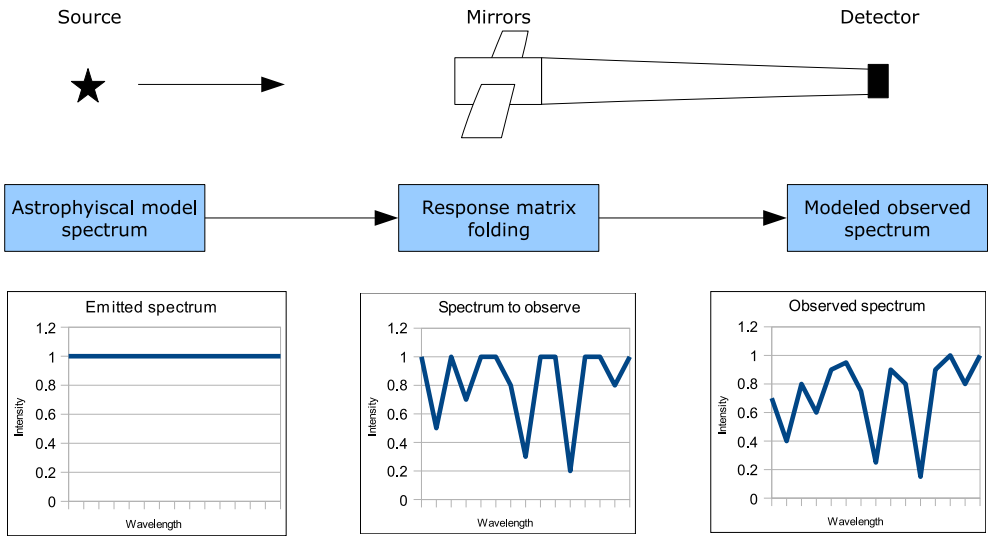

Figure 2.2: A simplified view of the path X-ray beams travel from source to observer. Note that the graphs are added purely for explanatory purposes, they have no real physical meaning whatsoever.

mization is not a goal in itself, it is a means to reach the goal of reducing execution time or memory usage. It may even be so that optimizing the execution time is mutually exclusive with the optimization of memory usage. For SPEX, the main concern is execution speed.

One can distinguish different *levels* of optimization. The highest level is the design or algorithm of the software. After all, a badly implemented smart algorithm may very well outperform a very well implemented but worse algorithm.

Below the algorithmic level is optimization at source code level. At this level, one may try to optimize the code such that it uses the available processing power to its best capabilities. This level also includes avoiding of obvious slowdowns, such as the use of many unnecessary indirect memory references. However, such optimizations may have a cost in readability and thus maintainability of the code and should therefore be considered with caution, in particular because the used compiler may be able to handle such optimizations transparently.

After optimization at the source code level, one may use an optimizing compiler to achieve further optimizations. However, usually the programmer has little control over the optimization actions performed by the compiler, except by specifying whether or not the compiler should perform certain optimizations.

At the lowest level one could optimize by programming using assembly code instead of using a high-level programming language. This allows the programmer to write code specific to a certain CPU, thus being able to use features specific to this CPU. This technique has a big drawback; the written code may run very inefficiently or not at all on another processor. Furthermore, assembly code is hard to read and therefore difficult to maintain. Note that this level of optimization will not be used in this thesis, but it is mentioned for completeness.

2.3 Parallel programming with Fortran

There are various options to choose from when parallelizing existing program code. This section takes a look at some of the features of a few of the most known of these options, thus making a good choice possible.

2.3.1 Requirements

Before being able to compare parallel programming possibilities it is important to know what is required by SPEX for such a parallel programming framework.

First of all, the resulting code should be readable and maintainable. It should be possible for someone without a firm background in parallel programming to understand what is going on and to be able to make small functional changes. This will ensure that the work presented in this thesis keeps its practical value for the years to come.

The used parallel framework must support shared memory parallelism. This is a very firm requirement, as SPEX is only used on regular desktop systems, not on distributed memory supercomputers.

It must be possible to compile and execute the resulting program code on both Linux and Mac OS-X systems, and preferably also on Microsoft Windows systems.

The latter is not a definite requirement if the parallel version can be compiled by a compiler which does not support parallelism, to result in a sequential version.

Since all work is done on existing code and considering the size of the existing codebase, it is desirable that the used framework supports incremental parallelization, i.e. it should be possible to parallelize just part of the program without altering anything else.

Obviously, the Fortran language must be supported.

2.3.2 Co-Array Fortran

Co-Array Fortran is an extension to Fortran 95, created by Robert Numrich and John Reid [Numrich and Reid, 1998]. It is currently included in the latest draft for the next revision of the Fortran standard [ISO/IEC, 2009].

The main feature of Co-Array Fortran is that it really integrates with the Fortran language. There are no external libraries or other dependencies, as it is supposed to become part of the language itself. This makes Co-Array Fortran easy to read and maintain and, becoming part of the official Fortran standard, it will be supported for many years to come.

Co-Array Fortran programming is done in Single Program Multiple Data (SPMD) style. Theoretically, the standard supports both shared memory systems and distributed memory systems, although this depends on the implementation.

The main drawback of Co-Array Fortran is that the only free compiler currently supporting Co-Array Fortran is the G95[1] compiler. However, this compiler only supports Co-Array Fortran on a cluster of computers connected by a network, although support for shared memory systems is in the design stage. This is essentially a deal-breaker, although Co-Array Fortran may become a good candidate in the future.

2.3.3 openMPI

OpenMPI (open Message Passing Interface) is an open-source implementation of the MPI standard [MPI Forum, 2008]. It is developed as a merger of different other MPI implementations, aiming to combine the parts these implementations excelled in into a single implementation.

OpenMPI completely depends on external libraries, thus removing the need for support at the level of the compiler. However, it does require additional runtime binaries to run the compiled application, which arguably makes the process of executing the application more complex.

Although openMPI is mainly used on distributed memory systems, it supports shared memory systems as well. OpenMPI is supported on Linux and OS-X; support for Microsoft Windows systems is planned but not yet implemented. Both the C/C++ and Fortran programming languages are supported by openMPI.

In general, openMPI is relatively complex. However, the programmer can control many details. This decreases readability of openMPI code and therefore makes

[1] http://www.g95.org/

it harder to maintain, especially by someone without a firm background in parallel programming. Moreover, openMPI and MPI in general are not really suited for incremental parallelism.

2.3.4 openMP

Although in name very similar to openMPI, openMP (open Multi-Processing) differs from openMPI in almost all other aspects. The openMP specification is developed from scratch; a first Fortran-only version was released in 1997 [OpenMP ARB, 1997], but a C/C++ version of the specification followed shortly after [OpenMP ARB, 1998]. The most recent specification is version 3 and incorporates both C/C++ and Fortran in a single specification [OpenMP ARB, 2008].

According to the openMP specification, it

> "specifies a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in C, C++ and Fortran programs." [OpenMP ARB, 2008, p. 1]

Thus, openMP supports shared memory parallelism. Furthermore, because of the use of compiler directives, a compiler with special openMP support will be necessary to compile openMP programs. Luckily, most modern compilers support a version of openMP. OpenMP version 3 is not yet widely supported, but the latest installments of GCC (GNU Compiler Collection) and the Intel Fortran compiler do support it, among others.

All compile-time parallel tasks are defined using openMP compiler directives. These directives are formatted such that they look like regular comments to a compiler without openMP support. Run-time operations such as controlling the number of threads or querying the current thread identifier are performed using library routines or environment variables.

OpenMP is relatively simple, as the programmer does not have to deal with message passing; this is done automatically by the compiler. However, if the need arises, it is possible for the programmer to do message passing manually. This offers flexibility by giving the choice between simplicity and full control.

Finally, openMP supports incremental parallelism. It is possible to parallelize a part of existing program code without altering any other part of the program; only the compiler will have to be instructed to use openMP.

2.3.5 Conclusions

OpenMP offers the best solution for the current problem. It is widely supported, can be used on shared memory systems, supports incremental parallelism and is relatively easy to read and maintain.

Chapter 3

Benchmarking SPEX

3.1 Introduction

In order to give quantitative real-world results of any optimization, a measure of performance is needed. Furthermore, to determine the performance bottlenecks of the software, a measure of performance of individual parts of the system is needed. These measures will be referred to as benchmarks.

For benchmarking, the Fortran StopWatch[1] module developed by William Mitchell at the National Institute of Standards and Technology will be used. This module is used by calling special subroutines on entry and exit of subroutines of SPEX. This allows the StopWatch module to keep track of the time spent in individual routines and, consequently, to keep track of the total execution time.

Note that there is a small overhead in calling the StopWatch related subroutines. In practice this means that a subroutine with a relatively small execution time compared to the overhead caused by StopWatch routines should not be timed individually.

3.2 Test environment

Fluctuations in results caused by anything but changes in the program must be kept to a minimum. To achieve this, the same compiler and hardware will be used throughout the entire project.

3.2.1 The compiler

For all tests the source code is compiled using the Intel Fortran compiler 11.0 professional edition[2]. Currently, three different compilers are used to compile the SPEX software. These are the Intel compiler, the G95 compiler[3] and the Lahey-Fujitsu compiler[4].

Three different compilers are used to create three different executables. However, the G95 compiler does not support openMP and therefore cannot be used. The Lahey-Fujitsu compiler does support openMP as does the Intel compiler. However, the research presented in this thesis is not about differences between compilers, thus

[1] http://math.nist.gov/StopWatch/
[2] Official website: http://www.intel.com/cd/software/products/asmo-na/eng/282048.htm
[3] Official website: http://www.g95.org/
[4] Official website: http://www.lahey.com/

a choice must be made.

The Intel compiler is chosen because past experiences with this compiler are very good. Furthermore, an SRON-wide license is available allowing anyone to compile the software if desired.

3.2.2 Hardware

All tests are conducted on *Galacticus*. This is a machine with an Intel Xeon 5335 quad-core processor. Galacticus has two gigabytes of RAM (Random-Access Memory) and about 120 GB (Gigabyte) of harddisk space.

The Intel Xeon 5335 processor has two levels of cache. Each of the four processor cores has access to 64 KB (KiloByte) of level 1 (L1) cache and 4 MB (MegaByte) of level 2 (L2) cache, although there is a total of 8 MB L2 cache available [Intel Corporation, 2009a].

3.3 Use-cases

For testing and benchmarking purposes three different use-cases are available. It would not be sufficient to test with a single use-case, since different data and different options given to the program may result in completely different benchmark results. To give some physical background, the backgrounds of these use-cases are briefly discussed in the following sections.

3.3.1 Case 1

This use-case concerns data from the European Photon Imaging Camera (EPIC), which is an instrument on XMM Newton. The source is a cluster of galaxies named RXC J1539.5-8335.

Clusters of galaxies

Clusters of galaxies are the largest gravitationally bound systems known in the Universe. In the early Universe, just after the Universe began to exist, all mass was almost homogeneously distributed. However, there were small deviations in this homogeneity. Since the laws of gravity cause places which have accumulated more mass than their surroundings to attract even more mass, the homogeneity decreased and mass was concentrated in certain places more and more.

Such a distribution of mass results in a web-like structure, as can be seen in Figure 3.1. The knots in this web contain a relatively high concentration of mass, indicating that there is a big chance many stars and galaxies have formed there. These knots are known as clusters of galaxies. Typically such a cluster contains hundreds of galaxies in a volume with a radius of 1 Mpc[5]. The total mass of such a cluster is around 10^{14} to 10^{15} masses of the Sun.

X-ray spectroscopy

About 75% to 80% of the mass in clusters of galaxies consists of dark matter. About a quarter of the remaining mass is contained in stars, the remaining matter is diffuse gas between the stars and galaxies. Because of the strong gravitational fields in a

[5] 1 Mpc (MegaParsec) = 3.26 million light years

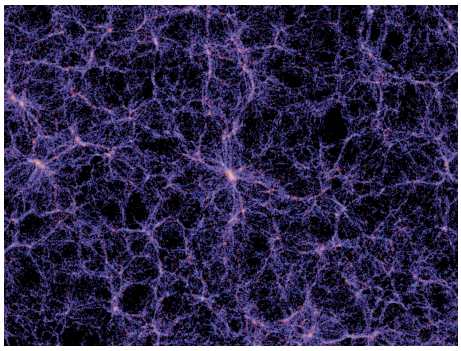

Figure 3.1: The distribution of mass (stars and galaxies) in the Universe. Color in the image relates to the concentration of mass, where lighter colors indicate more mass. This clearly shows the web-like structure of the Universe. The width of the image covers about 400 Mpc. Image courtesy: the millennium simulation; `http://www.mpa-garching.mpg.de/galform/virgo/millennium/`

cluster of galaxies, all the gas will be compressed and heated to temperatures of 10^7 to 10^8 Kelvin. Because of this temperature, the gas will radiate X-rays.

These X-rays are observed by EPIC on the XMM Newton satellite, which is a CCD-based instrument. Because the amount of photons seen by the detector is small, it is possible to determine the energy for every received photon, and thus a spectrum can be created. Using spectral analysis on this data makes it possible to retrieve information about the temperature and composition of the gas.

3.3.2 Case 2

Every galaxy contains a supermassive black hole at its center. These black holes typically have up to a billion times the mass of the Sun. A galaxy is considered active if the black hole is attracting gas and thereby forms an accretion disc. The gas in this disc is hot (about 10^4 to 10^6 Kelvin) and radiates in the ultraviolet and X-ray wavelengths. The power of this radiation is such that a large part of the gas in its path can be blown away from the center of the galaxy and in some cases even out of the galaxy.

This gas is often enriched, meaning that it contains more metal than average. Thus, these winds enable an AGN to enrich its galaxy, which influences the evolution of the galaxy. Another reason for the importance of these outflows of gas is the fact that they may blow away any other gas close to the black hole, locally preventing the formation of new stars. If this happens, accretion on the black hole will slowly stop due to the lack of gas in its vicinity. This feedback mechanism is currently a hot topic in astronomy.

These outflows can be studied with the XMM and Chandra satellites, which contain grating-based instruments. These satellites have a spectral resolution high enough to show the required absorption and emission lines, which tells something about the state of the gas and how fast it is travelling from or to us.

The data in this use-case is from an AGN known as Mrk 509. This AGN is too far from Earth for a nice photograph, but an artist impression of an AGN can be seen in Figure 3.2.

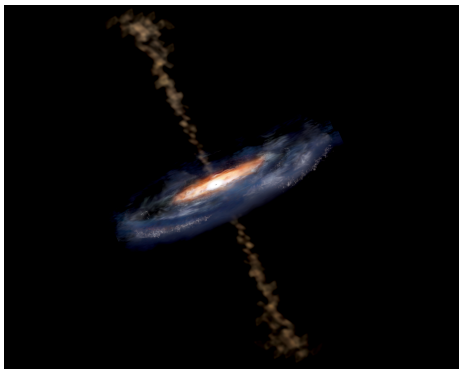

Figure 3.2: Artist impression of an AGN. Image credit: Aurore Simonnet, Sonoma State University.

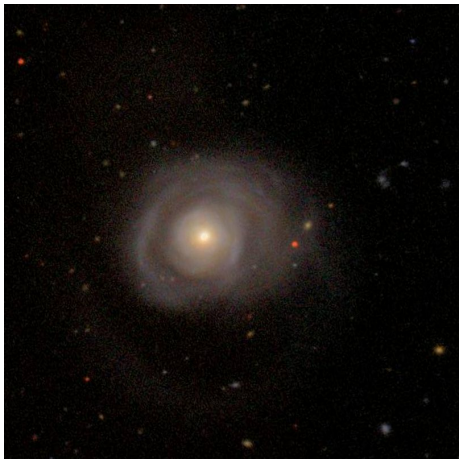

Figure 3.3: Optical image of NGC5548, taken by the SDSS telescope.

3.3.3 Case 3

Seyfert galaxies are a subclass of active galactic nuclei, named after Carl Keenan Seyfert, who first identified this class of galaxies [Seyfert, 1943]. They are characterized by a very bright nucleus and spectra with bright emission lines for hydrogen, helium, nitrogen, and oxygen.

This use-case looks at one Seyfert galaxy in particular, named NGC5548. A picture of this galaxy taken by the Sloan Digital Sky Survey (SDSS) telescope can be seen in Figure 3.3.

Data in this use-case is from the Low Energy Transmission Grating Spectrometer (LETGS), which is one of the instruments on the Chandra X-ray Observatory.

3.4 SPEX profile results

This section shows the results of the first tests, run with the unaltered version of SPEX. To be as accurate as possible all tests are run three times and results are averaged.

Since the total amount of subroutines profiled can be impractically large, only the most time-consuming are shown. The other subroutines are accumulated in a single number for clarity.

This section begins with a short introduction on the goals of profiling in this particular case. After that, profile results for each of the three use-cases are presented and briefly discussed. Finally, from these profiles two subroutines are chosen that appear most suitable for optimization.

3.4.1 Introduction

These tests have two important goals. First, they may give insight in the possible difference between test cases. Tests may show that different test cases result in about the same time spent in the same subroutines. However, more likely they will show that different test cases result in a completely different profile.

Second, knowing how much time different subroutines use gives an upper limit to speedup after optimizations of the specific subroutine, with speedup S defined as

$$S = \frac{T_1}{T_2},$$

where T_1 is the execution time of the original program and T_2 that of the optimized program. This can be used to determine an upper limit on S as

$$S_{\max} = \frac{p}{1 + f(p-1)}, \tag{3.1}$$

where $0 < f < 1$ is the fraction of time spent in the part of the program that is not optimized and p is how many times faster the optimized part has become. Thus, if a program takes 100 seconds before optimization and the time spent in the part that can be optimized is 60 seconds, maximum speedup depending on the amount of times the optimized part can be made faster is as in Figure 3.4.

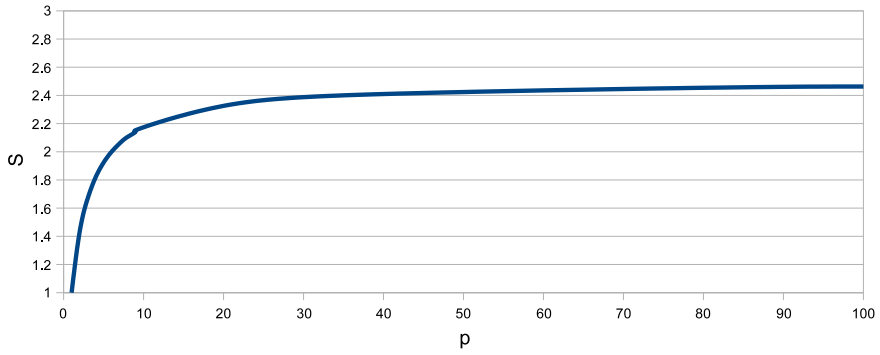

Figure 3.4: Max speedup with $f = 0.4$.

Clearly, there is a limit to the maximum speedup $S_{\max}$ depending on the amount of the program that can be optimized. This limit is

$$\lim_{p \to \infty} \frac{p}{1 + f(p-1)} = \frac{1}{f}.$$

Equation 3.1 is actually a more generic case of *Amdahls law* [Amdahl, 1967] which describes the same situation for the parallelizing process instead of the general optimization process as

$$S = \frac{1}{f + \frac{1-f}{P}}, \tag{3.2}$$

where f is the fraction of the program that can be parallelized and P is the number of processors used. Suppose 60% of a program can be parallelized and four processors are available, then the achievable speedup is

$$S = \frac{1}{0.4 + \frac{1-0.4}{4}} \approx 1.81.$$

3.4.2 Case 1

Results for the first use-case can be seen in Figure 3.5. The subroutine *freebound* uses almost 40% of the time. Most of the subroutines seen in Figure 3.5 are concerned with calculating the model spectrum, with the exception of *conresp_mat*, which multiplies the model spectrum with the response matrix as discussed briefly in Section 2.1.

In this particular use-case a relatively large amount of time is spent in other subroutines than the five taking most time, approximately 25%.

3.4.3 Case 2

Results for the second use-case can be seen in Figure 3.6. Notice that three of the top time-consuming subroutines in the profile for case 2 were also present in the profile for case 1, namely *conresp_mat*, *sigfot*, and *siglin*.

Subroutines that did not appear in the profile for case 1 are *ini_xabsfile* and *model*. Most time is spent in subroutines *sigfot* and *conresp_mat*, approximately 75%.

3.4.4 Case 3

Results for the third case can be seen in Figure 3.7. About 70% of the total execution time in this case is spent in *sigfot*. Another approximately 8% is spent in *conresp_mat*. Both of these routines are seen in the profile for all three use-cases.

3.4.5 Conclusions

As was shown in Equation 3.1 the best speedup can be achieved by optimizing that part of the program which takes the most time, if one assumes that it is unknown which part can be optimized how much and therefore assumes that all parts can be optimized equally.

Another criterium for selecting the right subroutines to optimize is the importance of this subroutine in all three use-cases. For instance, *freebound* takes a lot of time in one of the use-cases, but does not show up in the other two.

The above criteria lead to choosing the subroutines *conresp_mat* and *sigfot* for optimization. Both of these show up in all three of the use-cases as one of the five subroutines using most computing time, therefore optimizing these routines has practical value. Furthermore, *conresp_mat* performs matrix-vector multiplication, which is a very common task in scientific computing. Thus, the research done on this particular subroutine may very well have academic value outside SPEX.

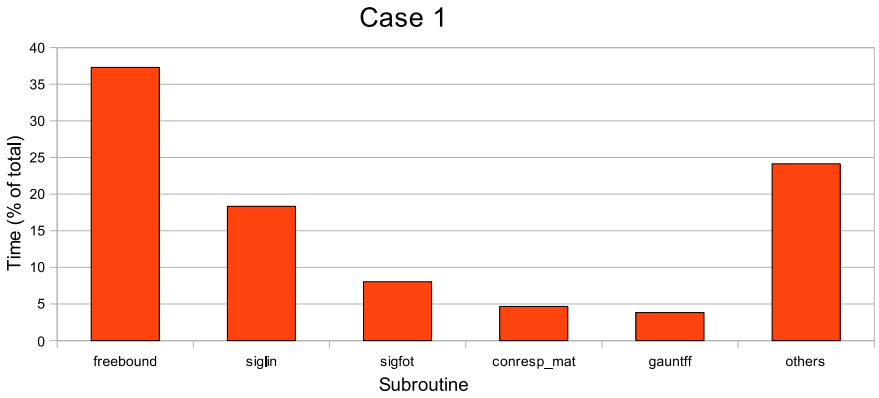

Figure 3.5: Program profile for use-case 1.

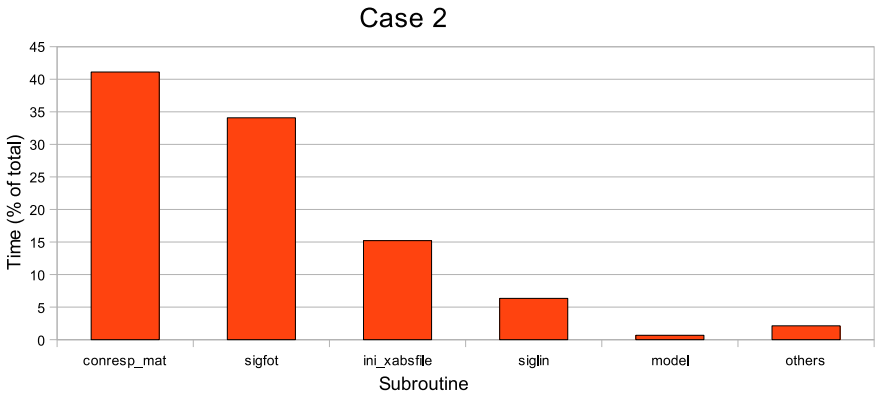

Figure 3.6: Program profile for use-case 2.

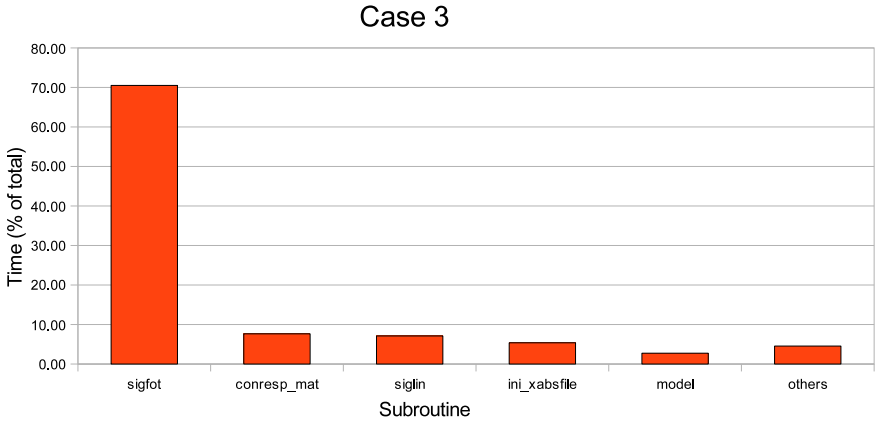

Figure 3.7: Program profile for use-case 3.

Part II

Matrix-vector multiplication

Chapter 4

Introduction

This part of the thesis focuses on optimization and parallelization of the matrix-vector multiplication used in SPEX. It starts with background information on this process, first on the physical background in this chapter. It then continues with an analysis of the current algorithm and implementation in Chapter 5.

Chapter 6 continues by proposing possible optimizations for the current version. These proposals are implemented and tested in Chapter 7. These optimizations are all sequential, i.e. no parallel processing is used yet.

A first parallel version is introduced in Chapter 8. Chapter 9 then suggests improvements to this first parallel version, which are implemented and tested in Chapter 10.

Finally, Chapter 11 and Chapter 12 provide conclusions and recommendations based on the work done in this part.

4.1 Background

Recall that X-ray spectra are usually analyzed by fitting a model spectrum to the observed spectrum, where the model spectrum has to be multiplied with a response matrix to correct for the instrument response. As instruments tend to get higher and higher resolutions, this process becomes increasingly time consuming.

In order to keep the sizes of the model spectrum and the response matrix limited, Kaastra proposed a method which essentially splits the model spectrum and the response matrix in two parts [Kaastra et al., 2007], but reduces the amount of elements needed. This method is explained in the following section.

4.1.1 Model binning

Consider a model of the spectrum of an X-ray source $f(E)$, where E is the energy of the received photons. Typically, $f(E)$ has units of for instance photons $\mathrm{m}^{-2}\mathrm{s}^{-1}\mathrm{keV}^{-1}$, i.e. photons per area per time per energy. To reconstruct how this spectrum is observed by an instrument it is convolved with the instrument response:

$$s(c) = \int R(c, E) f(E) \mathrm{d}E, \tag{4.1}$$

where $s(c)$ is the constructed observed spectrum, depending on c, the channel or pixel of the instrument with units of counts (photons) $\mathrm{s}^{-1}\mathrm{keV}^{-1}$; $R(c, E)$ is the

instrument response, which depends on both the channel and the energy and has units of for instance m^2, i.e. a measure of area.

Unfortunately, the model spectrum and instrument response are usually very complex, making it impossible in almost all cases to calculate the integral in Equation (4.1) analytically. Therefore, the model spectrum is usually evaluated on a limited set of energies corresponding to the energies in the response matrix. This turns the problem into a discrete one that can be analyzed numerically, changing the integral into a summation:

$$S_i = \sum_j R_{ij} F_j. \tag{4.2}$$

In this equation, index j corresponds to a bin in the set of energies used to evaluate the model spectrum, index i corresponds to a data channel. Note that Equation (4.2) is essentially a matrix-vector multiplication.

Define E_j as the centroid of bin j in the model grid, E_{1j} and E_{2j} as the upper and lower boundary of bin j and the bin width ΔE as $\Delta E = E_{2j} - E_{1j}$. The classical method of evaluating the model spectrum for the given set of energies is to evaluate the model at the bin centroid E_j, essentially taking

$$F_j = f(E_j)\Delta E_j.$$

This can be seen as the zeroth order approximation of the function $f(E)$ on the interval $[E_{1j}, E_{2j}]$ as

$$f_0(E) = N\delta(E - E_j),$$

where N is the total photon count in this bin and δ is the delta function. Intuitively, one can see that the worst-case scenario for this approximation is when the true model $f(E)$ is in fact a very small peak at one of the bin boundaries. This can be solved by taking the first order approximation, where instead of taking the total photon count at the centroid of the bin the total count is taken at the average energy as follows:

$$f_1(E) = N\delta(E - E_a),$$

where E_a is the average energy in the bin, i.e.

$$E_a = \frac{\int_{E_{1j}}^{E_{2j}} f(E)E\mathrm{d}E}{\int_{E_{1j}}^{E_{2j}} f(E)\mathrm{d}E} = \frac{\int_{E_{1j}}^{E_{2j}} f(E)E\mathrm{d}E}{N}. \tag{4.3}$$

This approximation will give the exact result in the worst-case scenario for the zeroth-order approximation (a small peak at one of the bin boundaries). However, this approximation has its worst-case scenario in the case of a small peak at both bin boundaries. This method is better, since either the precision can be increased or the number of bins can be decreased relative to the zeroth order approximation f_0 using the first order approximation f_1. This method is used in the current version of SPEX.

4.1.2 Consequences regarding the response matrix

The method described in the previous section results in a model spectrum which not only has a value representing the photon count for each bin, but also a value representing the average energy in this bin. This means there are now two vectors representing the model spectrum, while the observed spectrum S_i and the response matrix R_{ij} still have a "classical" representation.

However, the average energy of the bin has to be taken into account somehow, otherwise there is no difference with the zeroth order approximation, the amount of bins cannot be reduced and therefore there is no gain. Fortunately, [Kaastra et al., 2007] showed that

$$S_i = \sum_j \left[R_{ij} F_j + \delta R_{ij} (E_a - E_j) F_j \right], \tag{4.4}$$

where R_{ij} is the classical response and δR_{ij} is its derivative with respect to the energy. Note that this doubles required computation compared to equation (4.2). However, in practice the amount of bins needed can be reduced by a factor between 10 and 100 [Kaastra et al., 2007, p. 105], thus reducing the total computation time by a factor between 5 and 50.

On a practical note, some users still use the zeroth order approximation, i.e. $\delta R_{ij} = 0$ for various reasons. This has to be taken into account when optimizing the program.

4.1.3 Sectors and regions

This section tries to clarify some of the terminology used in the current implementation of the program.

Sky sectors

An observation often consists of multiple sources, which may need to be modeled differently. For instance, a bright point source may be superimposed on a cluster of galaxies.

In order to facilitate this possibility, the relevant part of the sky is divided into *sectors*. In the above example one could define two sectors, one for the point source and one for the cluster of galaxies. These sectors may overlap and will often use a different model.

Detector regions

Similarly as with sky sectors where the sky is divided into different parts, the detector instrument may be divided into different regions. In practice, this means that for a certain region only part of the pixels from the detector is taken into account.

Sectors and regions and the response matrix

When sky sectors and detector regions are used there will have to be a response matrix for every couple of sectors and regions. In order to calculate the "final" model spectrum every model corresponding to a sector has a contribution for every region, thus making the operation not just a single matrix-vector multiplication, but a summation of matrix-vector multiplications, depending on the number of sectors and regions present.

4.1.4 Storage format of the response matrix

The instrument response matrix is stored in a FITS (Flexible Image Transport System) file. This is a very common file format in the field of astronomy. It is designed especially for scientific data and includes facilities for storing of for instance spatial calibration information and image origin metadata [Wells et al., 1981].

In general, but not always, response matrices are sparse. This means that it

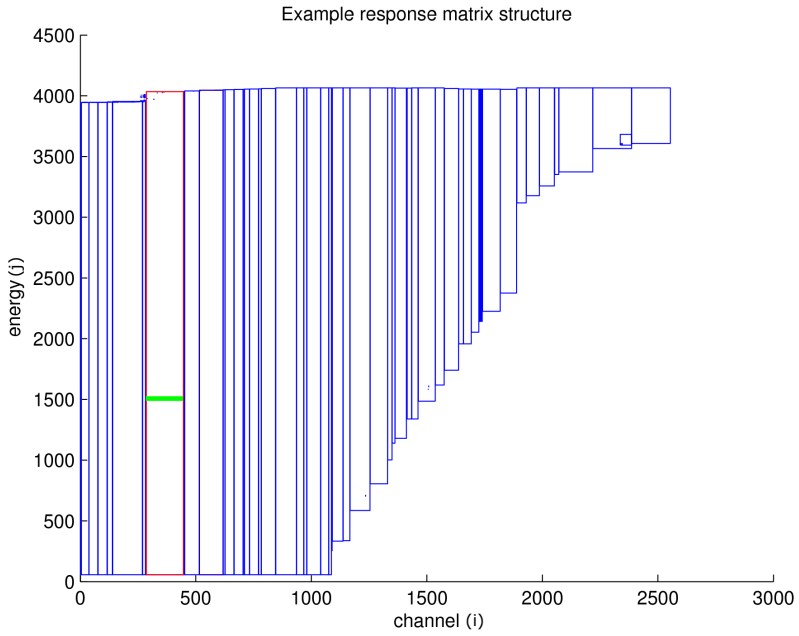

Figure 4.1: This figure shows the matrix layout of a response matrix for use-case 2. The blue outlined areas are bounding boxes around response components, one of these is highlighted in red. The green line represents a single response group within a response component. Note that response components in general will not be square, but they can overlap.

is inefficient to save the matrix as a twodimensional array. Furthermore, some meta-information is necessary about sectors and regions. Although the matrix may be sparse, its fill is not random. Generally, the non-zero elements are clustered together. It would therefore be inefficient to store every element as a triplet of value, row and column.

Instead, the matrix is divided into *components*, where every component is defined as a continuous part along the energy axis of the matrix. The component then has at most one response *group* for every energy, where a response group is defined as a vector of non-zero elements defined between two channels. In terms of R_{ij}, every component is a submatrix R_{kl}, where $0 \leq k, l \leq i_{\max}, j_{\max}$, while a response group G_l is one row-vector of a component R_{kl}, i.e. $G_l = R_{.l}$. Figure 4.1 tries to clarify this.

Chapter 5

Current implementation and algorithm

The current implementation of the matrix-vector multiplication consists of four nested subroutines. The top-level subroutine, *eval*, initializes memory for the total spectrum, i.e. the spectrum summed over all sky sectors. It then evaluates the model for this sector and calls the next subroutine, *conresp*.

Subroutine *conresp* contains two loops. The outer one loops over all instruments, because multiple instruments may be used to look at the same sky sector. The inner one loops over all detector regions within an instrument. Within the inner loop the next subroutine is called.

This next subroutine, *conresp_mat*, loops over all components in the response matrix. It then checks whether this component is in the right sector and region, and if so it calls the final subroutine.

This is *conresp_comp_mat*, which loops over the response groups of the current component. It then re-bins the model spectrum if necessary, and performs its part of the matrix-vector multiplication.

5.1 Algorithm

The current algorithm is shown in detail in Algorithm 1. In this algorithm, most variable names are related to those described in Sections 4.1.1 and 4.1.2. For clarity, that is:

$N(E)$ Total photon count for energy bin E.

$W(E)$ The weighted distance from the average energy in the bin, i.e.

$$W(E) = (E_a - E_c)N(E),$$

where E_a is the average energy as defined in Equation 4.3 and E_c is the energy at the center of bin E.

E_j, E_k Model re-binning boundaries, i.e. if the model is defined on a finer grid than the response matrix, E_j and E_k define which model bins describe a single bin in the response matrix.

$\vec{s}$ The constructed, simulated version of the observed spectrum.

R The response matrix.

ΔE Width of energy bin E.

$A(i)$ Area scale factor for energy group i.

$c1(i), c2(i)$ Channel range for energy group i.

Algorithm 1 The original implementation of the matrix-vector multiplication

```
1: reset ← true
2: for all i_sect in sectors do
3:    N⃗, Ē_a ← evaluate model for i_sect
4:    for all i_instr in instruments do
5:       for all regions (i_reg) in i_instr do
6:          if reset = true then
7:             s⃗ ← 0
8:          end if
9:          for all components (i_comp) in i_reg do
10:            if i_comp(sector) = i_sect AND i_comp(region) = i_region then
11:               for i_eg = 1 to n_eg do
12:                  N_new = Σ_{E=E_j}^{E_k} N(E)
13:                  W_new = (Σ_{E=E_j}^{E_k} W(E)) + ⟨N_{E_j...E_k}, E_{j...k} − E_c⟩
14:                  if area_scale = true then
15:                     N_new ← N_new A(i_eg)
16:                     W_new ← W_new A(i_eg)
17:                  end if
18:                  if renorm ≠ 1 then
19:                     N_new ← N_new · renorm
20:                     W_new ← W_new · renorm
21:                  end if
22:                  for i_channel = c1(i_eg) to c2(i_eg) do
23:                     s(i_channel) = s(i_channel) + N_new R_{i_channel,i_eg} + W_new δR_{i_channel,i_eg}
24:                  end for
25:               end for
26:            end if
27:         end for
28:      end for
29:   end for
30:   reset ← false
31: end for
```

Some of the operations in Algorithm 1 have not yet been discussed in detail. One of these is the re-binning of the input spectrum in lines 12 and 13. Re-binning the total photon count (N) is straightforward, but re-binning of the average energy ($W(E)$) deserves some more explanation.

Recall that $W(E)$ is defined as $W(E) = (E_a - E_j)N(E)$ for a single bin E. Let E_A be the average energy in the re-binned situation, E_c the center of the new bin and N_{tot} the total photon count of the new bin, i.e.

$$N_{\text{tot}} = \sum_{\text{bins}} N(E).$$

The re-binned average energy W_{new} will then be

$$W_{\text{new}} = N_{\text{tot}}(E_A - E_c). \tag{5.1}$$

The definition of E_a in Equation 4.3 is for a continuous function, this is however easily transformed in a discrete function for E_A as

$$E_A = \frac{\sum_{\text{bins}} N(E)E_a}{\sum_{\text{bins}} N(E)}.$$

Inserting this in Equation 5.1 gives

$$W_{\text{new}} = \sum_{\text{bins}} N(E) \left(\frac{\sum_{\text{bins}} N(E)E_a}{\sum_{\text{bins}} N(E)} - E_c \right) = \sum_{\text{bins}} N(E)E_a - \sum_{\text{bins}} N(E)E_c.$$

This can be written in terms of the original $W(E)$ by adding zero:

$$\sum_{\text{bins}} N(E)E_j - \sum_{\text{bins}} N(E)E_j,$$

which gives

$$W_{\text{new}} = \sum_{\text{bins}} N(E)E_a - \sum_{\text{bins}} N(E)E_c + \sum_{\text{bins}} N(E)E_j - \sum_{\text{bins}} N(E)E_j.$$

Reordering this gives

$$W_{\text{new}} = \sum_{\text{bins}} N(E)(E_a - E_j) + \sum_{\text{bins}} N(E)(E_j - E_c) = \sum_{\text{bins}} W(E) + \langle N, E_j - E_c \rangle,$$

where the dot product is with N and E_j for the relevant bins, E_c is a scalar. This is exactly what is done on line 13 of Algorithm 1.

5.2 Inspection of the matrices

To be able to optimize the matrix-vector multiplications, it is important to know the layout of the matrices. After all, the matrices may very well have properties which can be used to optimize the program.

In this section, matrices from all use-cases are inspected. Doing so should give a better understanding of the layout of the different components in the matrix, the general fill of the matrix, and any other special properties that may show up.

5.2.1 Case 1

Case 1 uses only one instrument region and one sky sector, meaning there are only two matrices, R_{ij} and δR_{ij}. However, in use-case 1 the matrix δR_{ij} is not used (it is all zeros), so there is only one matrix with non-zero elements. Figure 5.1 shows the fill of the matrix and the layout of the response components.

Table 5.1 shows some more detailed information about the matrix. This shows that there is some, but very little, overlap as 29 values are at a location in the matrix which already was non-zero. Furthermore, Figure 5.1 shows only three distinctive components, while Table 5.1 clearly states there are actually 23. The twenty components which are not clearly visible in Figure 5.1 are all very small components containing only a few values on or very close to a border between two of the larger components.

Note that use-case 1 actually performs the same calculations on seven different datasets. The seven corresponding matrices however show so much similarity that it would be redundant to show them all here.

Response groups	2297
Components	23
Values	113617
Non-zero elements	113588
Energy bins	2091
Channels	230
Fill	23.00 %

Table 5.1: Properties of the response matrix for use-case 1.

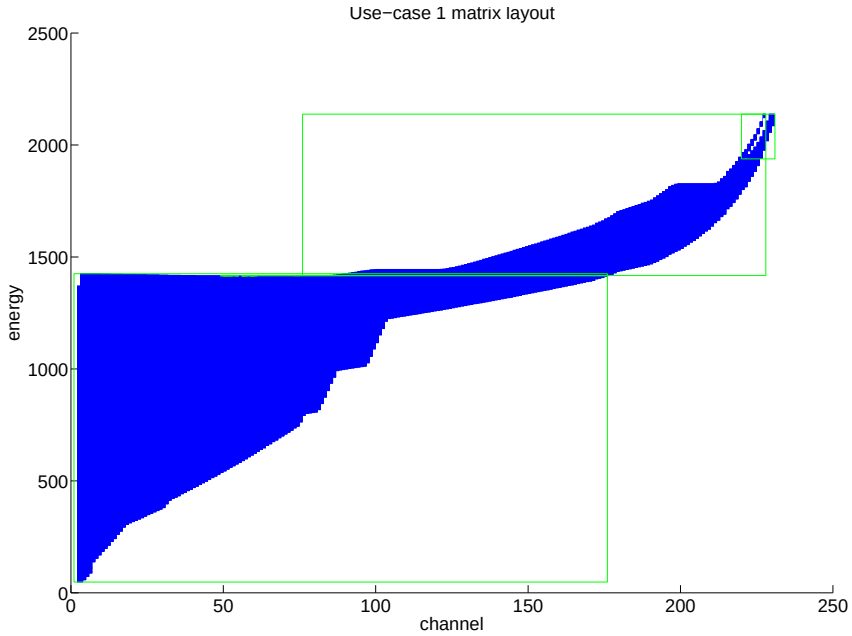

Figure 5.1: Matrix fill for use-case 1. Blue elements denote non-zeros, the green boxes are bounding boxes around the components in the response matrix. Note that although the image appears square, the matrix is not.

	Region 1	Region 2	Region 3	Region 4
Response groups	171720	110698	67164	48856
Components	166	75	234	91
Values	6748510	5968305	2539149	2308146
Non-zero elements	6748510	5968305	2539149	2308146
Energy bins	4010	4010	2886	2726
Channels	2554	2607	2211	2212
Fill	65.89 %	57.09 %	39.79 %	38.28 %

Table 5.2: Properties of the response matrices for use-case 2.

5.2.2 Case 2

Case 2 uses only one instrument and one sky sector, but four regions. Thus, there will be four times two matrices. However, just as with use-case 1 the matrix δR_{ij} is all zeros, so there are only four matrices with non-zero elements. Figure 5.2 shows the layout of two of the four response matrices for use-case 2, Table 5.2 gives some more detailed information on these matrices. Note that the other two response matrices have a layout similar to those shown in Figure 5.2.

Note that these matrices, contrary to the one from use-case 1, have no overlap, i.e. the number of non-zeros in every matrix is equal to the number of values defined in the input file. Furthermore, these matrices are far more dense than the one from use-case 1, especially those for regions 1 and 2. They also are a lot bigger, with non-zero element counts in the order of millions.

Again, many of the components are relatively small while some of the components are relatively big. This is most easy to see in Figure 5.2 for region 3 where many small components are visible in the upper left of the matrix. Note that the order in which the components are stored is random.

All response matrices for use-case 2 have a somewhat triangular layout. However, none of them - with the possible exception of the region 3 response matrix - is truly triangular.

5.2.3 Case 3

Case 3 is the only one that uses both the regular matrix R_{ij} and matrix δR_{ij}. However, fill for these matrices is identical, only the values are different. Case 3 uses only one instrument, sector and region, so there are two matrices in total.

Figure 5.3 shows the fill of the response matrices for case 3 and the layout of the response components. Note that where the matrices from use-cases 1 and 2 were somewhat similar, this one is completely different. This is unfortunate as this may make it more difficult to do optimizations based on layout of the matrices.

Table 5.3 shows some more detailed information on the response matrix. Again, there is no overlap between components. Contrary to the matrices for use-cases 1 and 2 this matrix is quite sparse with a fill of only 1.42 %. The matrix is also non-square, with the amount of energy bins an order of magnitude bigger than the amount of channels.

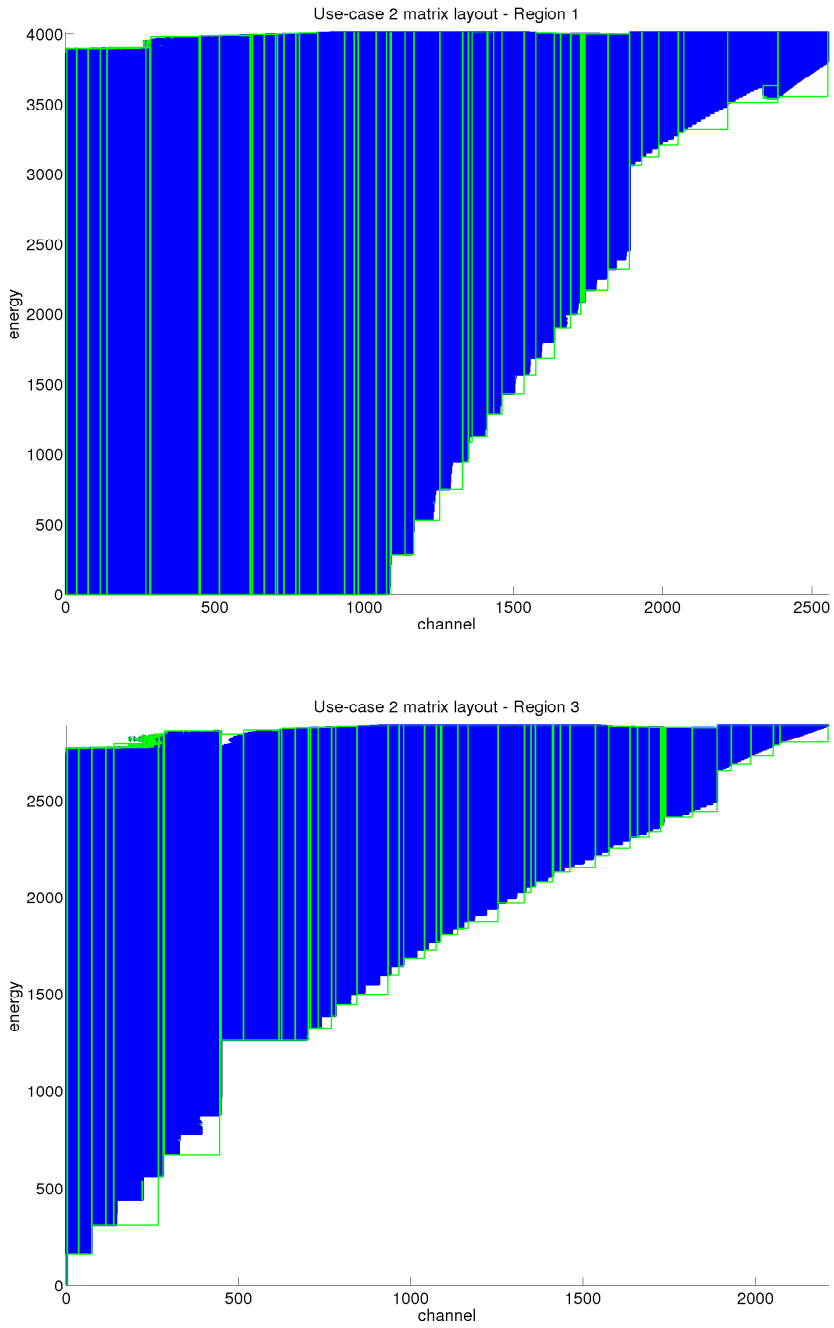

Figure 5.2: Matrix fill for use-case 2, regions 1 and 3. Blue elements indicate non-zeros, the green boxes are bounding boxes around the components in the response matrix.

Response groups	143231
Components	9
Values	2853383
Non-zero elements	2853383
Energy bins	36693
Channels	5479
Fill	1.42 %

Table 5.3: Properties of the response matrix for use-case 3.

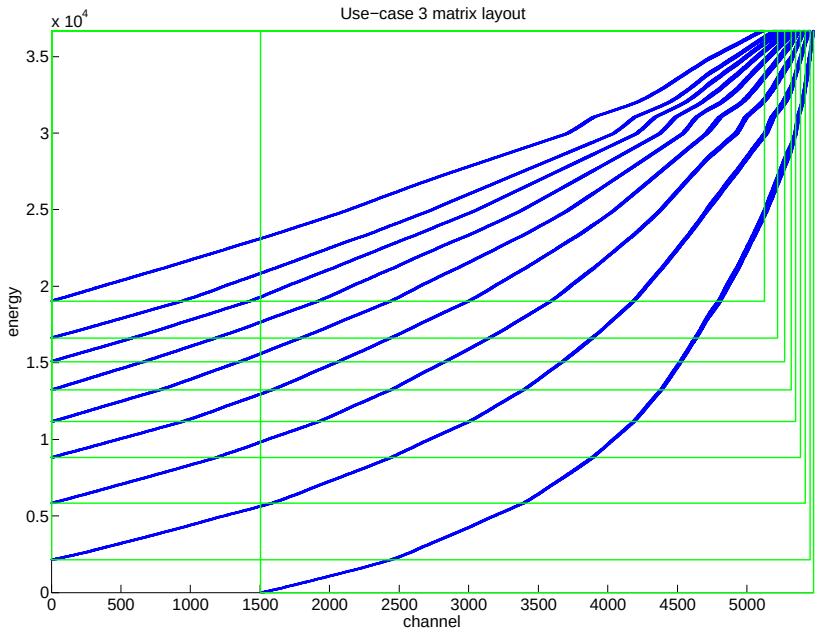

Figure 5.3: Matrix fill for use-case 3. Blue elements denote non-zeros, the green boxes are bounding boxes around the components in the response matrix. Note that although the image appears square, the matrix is not.

	Case 1	Case 2	Case 3
conresp_mat	76.55	438.94	4.56
total	1640.12	1067.66	59.74
$S_{\max}$	1.05	1.701	1.08

Table 5.4: Reference measurement results in seconds. Maximum speedup $S_{\max}$ is defined as $\frac{1}{f}$, where f is the fraction of the program that will not be optimized.

5.3 Reference measurements

Chapter 3 benchmarked SPEX. However, in this chapter only a profile of SPEX was given with percentages spent in different subroutines. This makes comparisons with an improved version difficult. Therefore, this section presents reference measurements with absolute values.

All optimizations presented in this part are mainly concerned with the matrix-vector multiplication. Thus, in order to keep the information presented clear, only timings for the subroutine *conresp_mat* and the entire program time are discussed.

All use-cases were run three times and the averaged execution times are used. If one of the test results showed a value suspiciously different from the other two it is rerun. Results are shown in Table 5.4, including the maximum achievable speedup $S_{\max}$.

Chapter 6

Proposed optimization methods

This chapter proposes some possible optimizations with regard to Algorithm 1 and the matrix layouts shown in Chapter 5.2. Note that all proposed optimizations must satisfy the following criteria:

1. The dataformat in which the matrices are stored should not have to change. This is due to the fact that the matrices are usually delivered by those who build the corresponding instrument. However, this does not mean that it is impossible to make changes to the datastructure once it is loaded in memory.
2. All optimizations have to be implemented in Fortran 90 and should compile with the Intel Fortran compiler without difficulties.

6.1 Checking the use of δR

Although the use of δR can theoretically improve performance by an order of magnitude (recall Section 4.1.2), this is only used in use-case 3. Obviously, the best solution to this is to make people aware of this possibility and have them generate the appropriate datasets. It may seem best to make people aware of this by just stopping support for this case, thereby forcing them to create new matrices. However, this would also mean that it becomes impossible to analyze "old" data with newer versions of SPEX, which is undesirable.

Therefore, the algorithm should be changed such that it does not perform any calculations on δR if this matrix is all zeros. This will also make it possible to stop storing all these values in memory the entire run of the program, reducing memory requirements for the response matrix to about half the original.

6.2 Response component ordering

Recall from Chapter 5.2 that currently the order in which the response components of the matrix are stored is apparently random. This implies that memory access - specifically of the result vector $\vec{s}$ - will not be continuous, which may result in cache inefficiency.

Consider the response matrix for use-case 2, region 3 (Figure 5.2) in a simplified version (Figure 6.1). In this figure, the response components are stored and

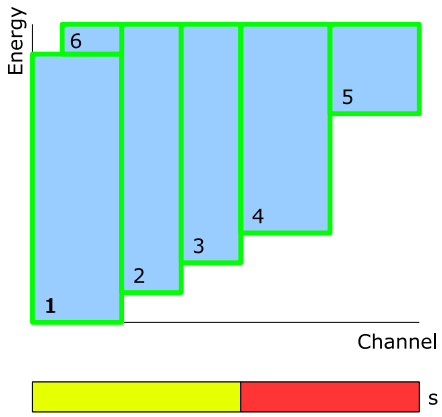

Figure 6.1: Simplified response matrix example. The black numbers indicate the order in which the response components are stored and processed.

processed from left to right, with the exception of component 6. Now suppose that about half of the result vector $\vec{s}$ fits in the processors cache memory. Then, when the first component is processed, the first half of the result vector will be in cache (the green part in Figure 6.1). This is good, as the same part of this vector is needed for components 2 and 3 as well. Once component 4 is processed, the second half of the result vector will be loaded in the processors cache memory, which can then be reused for component 5. Unfortunately, when processing component 6 again the first part of $\vec{s}$ is needed but it is no longer in the cache memory, meaning that there is unnecessary access to the main memory.

Obviously, this effect will be even worse if over 200 components are stored in random order as is the case with the response matrix of region 3 from use-case 2. However, the described process is a simplification of a very complex system and it is impossible to predict whether it is possible to diminish this effect.

A possible solution to this is to sort the response components in the response matrix, either from left to right or from right to left. Sorting can be done using the lowest or highest channel value in a component or the average of the two. However, it is expected that due to the irregularity in the response components there will be no performance difference depending on the sort criterium.

Use-case 2 will probably benefit most from this proposal, as the relevant response matrices have relatively many components.

6.3 Increasing precision

Inspection of the response matrix for use-case 3 shows that the absolute value of R varies between 10^{-15} and 10^{-3}, while the absolute value of δR varies between 10^{-15} and 10! While such small values are representable with single precision floating point values, rounding errors will be relatively large if operations include a combination of larger and smaller numbers, as is the case with $\vec{s}$. For example, the difference between the 1.0 and the smallest single precision floating point value larger than 1.0 is already $1.19209290 \cdot 10^{-07}$.

It may not be immediately clear why increasing precision can result in an in-

crease in performance. After all, increased precision will increase memory usage, which may cause more cache misses. However, recall that *conresp_mat* is part of a fitting procedure. This means that it may be so that increased precision will result in faster convergence of the fit, which will result in shorter execution time, not only for *conresp_mat*, but for every subroutine that is part of the fitting procedure.

6.4 Parallelizing

Once the sequential algorithm has been updated it is time to start looking into a possible parallel version of the algorithm. There are some natural possibilities for data-level parallelism ingrained in the way the response matrices are stored. For instance, the data is already divided into chunks at the response component level and at the response group level for all matrices. Note that parallelism at the region or sector level is not really an option as not all use-cases have multiple sectors or regions.

Since parallelization is a relatively big subject, a separate chapter (Chapter 8) will be devoted to this.

Chapter 7

Optimizations and implementation

This chapter implements and tests the optimizations proposed in Chapter 6.

7.1 Checking the use of δR

As shown before in Section 5.2, only use-case 3 uses δR; use-cases 1 and 2 only use R, i.e. δR is all zeros for these cases. Although using both R and δR is theoretically more efficient [Kaastra et al., 2007], not all datasets are adjusted to this, especially those from older instruments. Unfortunately, the current matrix-vector multiplication algorithm in SPEX does not take this into account. Instead, it always multiplies the model spectrum with both R and δR, even if the latter is all zeros and thus contributes nothing to the result.

This means that for cases where δR is not used it should be relatively easy to improve the execution speed by checking the values in δR at the time the datafile is read. Doing so one can set a flag in memory indicating whether the matrix is all zeros or not, and during the matrix-vector multiplication one can check that flag and act accordingly.

7.1.1 Proposed algorithm

The proposed algorithm is presented as Algorithm 2. Note that it is very similar to the original algorithm (Algorithm 1). All variables are the same in both algorithms, refer to Section 5.1 for details.

This proposed algorithm is expected to give better results, because it checks whether δR is non-zero and only performs relevant computations if it is. Ultimately, one would expect this to halve the computation time. However, in practice this will not happen due to overhead in for instance looping and if-clauses.

Thus, it can be expected that this algorithm will be slightly slower for use-case 3. However, it can also be expected that this solution will be significantly faster for use-cases 1 and 2.

Algorithm 2 The revised algorithm for matrix-vector multiplication

reset $\leftarrow$ **true**
for all i_{sect} in sectors **do**
  $\vec{N}, \vec{E}_a \leftarrow$ evaluate model for i_{sect}
  for all i_{instr} in instruments **do**
    for all regions (i_{reg}) in i_{instr} **do**
      if reset = **true then**
        $\vec{s} \leftarrow 0$
      end if
      for all components (i_{comp}) in i_{reg} **do**
        if i_{comp}(sector) = i_{sect} AND i_{comp}(region) = i_{region} **then**
          for $i_{\text{eg}} = 1$ to n_{eg} **do**
            $N_{\text{new}} = \sum_{E=E_j}^{E_k} N(E)$
            if area_scale = **true then**
              $N_{\text{new}} \leftarrow N_{\text{new}} A(i_{\text{eg}})$
            end if
            if renorm $\neq 1$ **then**
              $N_{\text{new}} \leftarrow N_{\text{new}} \cdot \text{renorm}$
            end if
            for $i_{\text{channel}} = c1(i_{\text{eg}})$ to $c2(i_{\text{eg}})$ **do**
              $s(i_{\text{channel}}) = s(i_{\text{channel}}) + N_{\text{new}} R_{i_{\text{channel}}, i_{eg}}$
            end for
            if $\delta R \neq 0$ **then**
              $W_{\text{new}} = \left(\sum_{E=E_j}^{E_k} W(E)\right) + \left\langle N_{E_{j \ldots E_k}}, E_{j \ldots k} - E_c \right\rangle$
              if area_scale = **true then**
                $W_{\text{new}} \leftarrow W_{\text{new}} A(i_{\text{eg}})$
              end if
              if renorm $\neq 1$ **then**
                $W_{\text{new}} \leftarrow W_{\text{new}} \cdot \text{renorm}$
              end if
              for $i_{\text{channel}} = c1(i_{\text{eg}})$ to $c2(i_{\text{eg}})$ **do**
                $s(i_{\text{channel}}) = s(i_{\text{channel}}) + W_{\text{new}} \delta R_{i_{\text{channel}}, i_{eg}}$
              end for
            end if
          end for
        end if
      end for
    end for
  end for
  reset $\leftarrow$ **false**
end for

7.1.2 Implementation

First of all, the implementation will have to check whether δR is all zeros or not. Natural place to do this is when the data is read from disk. This way, this check has to be done only once, after which a flag can be set in the response data structure, which can then be used during matrix-vector multiplication.

In practice this is very simple, as one can check while reading data from disk for every energy group whether the response is all zeros. If this is the case for all energy groups within a component, a component-wide flag is set indicating that for this particular component δR is not used.

Listing 7.1 shows the relevant source code (*read_resp_mat.f90*) from SPEX. This listing shows the main loop which reads data from disk which loops over all components. At the beginning of every loop iteration, the local variable *allzero* and the variable *c%usedresp*, which is part of the component structure, are set to true (lines 3,4). Thus, by default δR will be used. Only if, for all of the energy groups in this component, the check in lines 9 and 10 shows that all values are 0, the variable *c%usedresp* will be set to false and allocated space for δR will be deallocated (lines 14-21).

```
1  do i = 1,r%ncomp
2      c => r%comp(i)
3      allzero = .true.
4      c%usedresp = .true.
5      do j = 1,c%neg
6          [...] ! read data from disk for the current energy group
7
8          ! perform zero check
9          if (.not.(MAXVAL(c%dresp(j)%data).eq.0.
10             .and.0..eq.MINVAL(c%dresp(j)%data))) then
11             allzero = .false.
12         endif
13     enddo
14     if (allzero .eq. .true.) then
15         c%usedresp = .false.
16         do j = 1,c%neg
17             if (associated(c%dresp(j)%data)) then
18                 deallocate (c%dresp(j)%data)
19             endif
20         enddo
21     endif
22 enddo
```

Listing 7.1: Checking δR for non-zeros. Note that *[...]* indicates some code has been removed for clarity.

Furthermore, the implementation of subroutine *conresp_mat* will have to be changed according to Algorithm 2. For comparison, the original code can be seen in Listing 7.2. This program code relates directly to Algorithm 1, with the exception of some local variables used for efficiency and readability. Obviously, this implementation is very inefficient if δR is not used. After all, it always computes the variable *wie* (called W in Algorithms 1 and 2), and it always computes the contribution of δR.

The revised version, based on Algorithm 2, can be seen in Listing 7.3. Note that this listing only contains the body of the subroutine, all declarations are the same as in Listing 7.2. The code in this listing is a direct translation of Algorithm 2. Both listings only include the loop on the energy grid. Loops on instruments, sectors, regions, and components are currently in different subroutines, as explained in Chapter 5.

```
1  subroutine conresp_comp_mat(neg,eg,sener,wener,c,ch,area_scal,renorm,s)
2      !-- convolves the model spectrum SENER with the response
3      !-- matrix C of a component for the matrix-type of response;
4      !-- the convolved spectrum is added to S
5      use response, only : rcomp,rgroup
6      implicit none
7
8      integer,              intent(in)    :: neg
9      real, dimension(neg), intent(in)    :: eg
10     real, dimension(neg), intent(in)    :: sener
11     real, dimension(neg), intent(in)    :: wener
12     type (rcomp), pointer               :: c
13     type (rcomp), pointer               :: ch
14     logical, intent(in)                 :: area_scal
15     real, intent(in)                    :: renorm
16     real, dimension(:), intent(inout)   :: s
17
18     integer                            :: i,i1,i2
19     real                               :: sie,wie
20     type (rgroup), pointer             :: g,h
21
22     do i = 1,c%neg
23         g => c%group(i)
24         h => ch%group(i)
25         if (g%nc.eq.0) cycle
26         i1 = g%ie1
27         i2 = g%ie2
28
29         !-- re-bin the input spectrum
30         sie = sum(sener(i1:i2))
31         wie = sum(wener(i1:i2))
32               + dot_product(sener(i1:i2),eg(i1:i2)-g%eg)
33
34         !-- prescale the flux with effective area factor if needed
35         if (area_scal) then
36             sie = sie * g%relarea
37             wie = wie * g%relarea
38         endif
39
40         !-- multiply the model spectrum by the instrument
41         !-- renormalization
42         if (renorm.ne.1.) then
43             sie = sie * renorm
44             wie = wie * renorm
45         endif
46
47         !-- do the convolution
48         s(g%ic1:g%ic2) = s(g%ic1:g%ic2) + sie * h%resp(:)
49                                         + wie * h%dresp(:)
50     enddo
51 end
```

Listing 7.2: Original version of *conresp_mat*.

```
1  do i = 1,c%neg
2      g => c%group(i)
3      h => ch%group(i)
4      if (g%nc.eq.0) cycle
5      i1 = g%ie1
6      i2 = g%ie2
7
8      !-- re-bin the input spectrum
9      sie = sum(sener(i1:i2))
10
11     !-- prescale the flux
12     if (r%area_scal) then
13         sie = sie * g%relarea
14     endif
15
16     !-- multiply the model spectrum by the instrument
17     !-- renormalisation
18     if (r%renorm(ireg).ne.1.) then
19         sie = sie * r%renorm(ireg)
20     endif
21
22     !-- do the convolution
23     s_local(g%ic1:g%ic2) = s_local(g%ic1:g%ic2) + sie * h%resp(:)
24
25     if (c%usedresp.eq..true.) then
26         !-- rebin the input spectrum
27         wie = sum(wener(i1:i2))
28               + dot_product(sener(i1:i2),eg(i1:i2)-g%eg)
29
30         !-- prescale the flux
31         if (r%area_scal) then
32             wie = wie * g%relarea
33         endif
34
35         !-- multiply the model spectrum by the instrument
36         !-- renormalisation
37         if (r%renorm(ireg).ne.1.) then
38             wie = wie * r%renorm(ireg)
39         endif
40
41         !-- do the convolution
42         s_local(g%ic1:g%ic2) = s_local(g%ic1:g%ic2) + wie * h%dresp(:)
43     endif
44 enddo
```

Listing 7.3: Revised version of *conresp_mat*.

7.1.3 Results

Test results for all three use-cases compared to the original program and are shown in Figure 7.1. These results are as expected, i.e. most subroutines use approximately the same amount of time as before, with the exception of *conresp_mat* for use-cases 1 and 2, which has become faster. The increase in speed is not a factor two, which can be explained by introduced overhead.

7.2 Using double precision

As discussed in Section 6.3, increasing precision may increase performance. This leaves the question: Which variables are to be put in double precision? The response matrix itself could be loaded in memory entirely as double precision variables. However, since the response matrices account for most of the memory used by SPEX, that would almost double the program's memory profile. Furthermore, errors in the response matrix itself will not accumulate, in the worst-case there will be a small error when loading the response matrix in memory. Thus, this would gain very little increase in precision at the cost of twice the memory footprint.

It would be more sensible to use double precision for the result vector, that is, the calculated spectrum $\vec{s}$. This keeps additional memory requirements low and should ensure that errors do not accumulate.

7.2.1 Results

Figure 7.2 shows test results for all use-cases when $\vec{s}$ is stored as a double precision floating point value. All use-cases show a degree of improvement in speed, although the improvement for use-case 1 is decidedly smaller than that for cases 2 and 3. As discussed in 5.2 the response matrices for use-case 1 contain relatively few non-zero elements. Hence, the accumulated error will have been smaller when using single precision, and consequently the gain when using double precision is smaller. To give an idea of how much faster the fitting process can be, use-case 2 used to perform 3324 evaluations, but when using a double precision variable for $\vec{s}$ this decreases to 2679 evaluations.

7.3 Response component ordering

As discussed in Section 6.2 efficiency of *conresp_mat* can possibly be further increased when ordering the response components in memory. Theoretically, mainly use-case 2 will benefit from this, as the response matrices for this case contain the most components. On the other hand, having the most components also means that sorting will take more time. Luckily, sorting has to be done only once, when the data is read from disk.

7.3.1 Sorting algorithm

Many different algorithms exist to facilitate sorting of lists in computer science. Some are relatively simple, but also relatively slow, while others may be more complex but faster in general or faster for lists with specific properties.

Region 3 of use-case 2 contains 234 components, which is the most in all available use-cases. For a sorting algorithm this should not be much, so it seems best to use a simple sorting algorithm, thereby sacrificing some performance for reduced

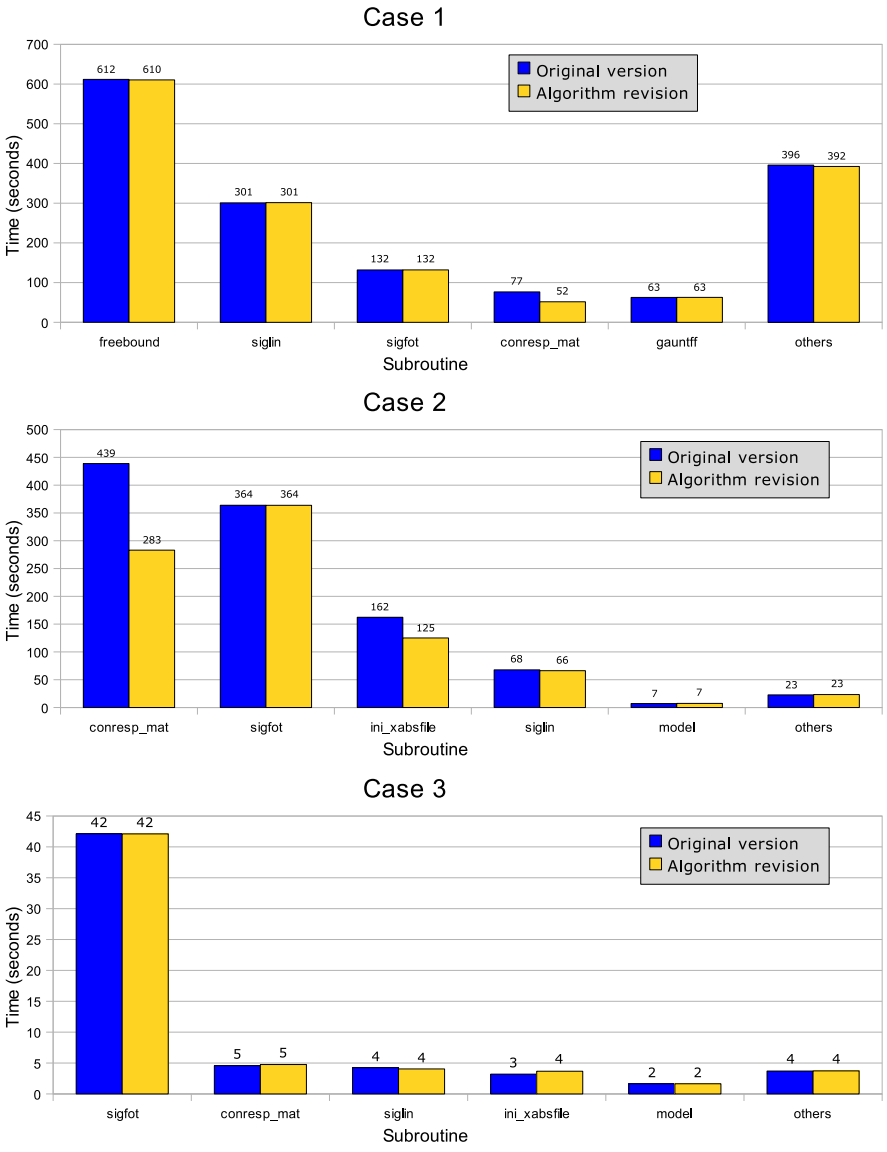

Figure 7.1: Tests results after implementation of the revised algorithm.

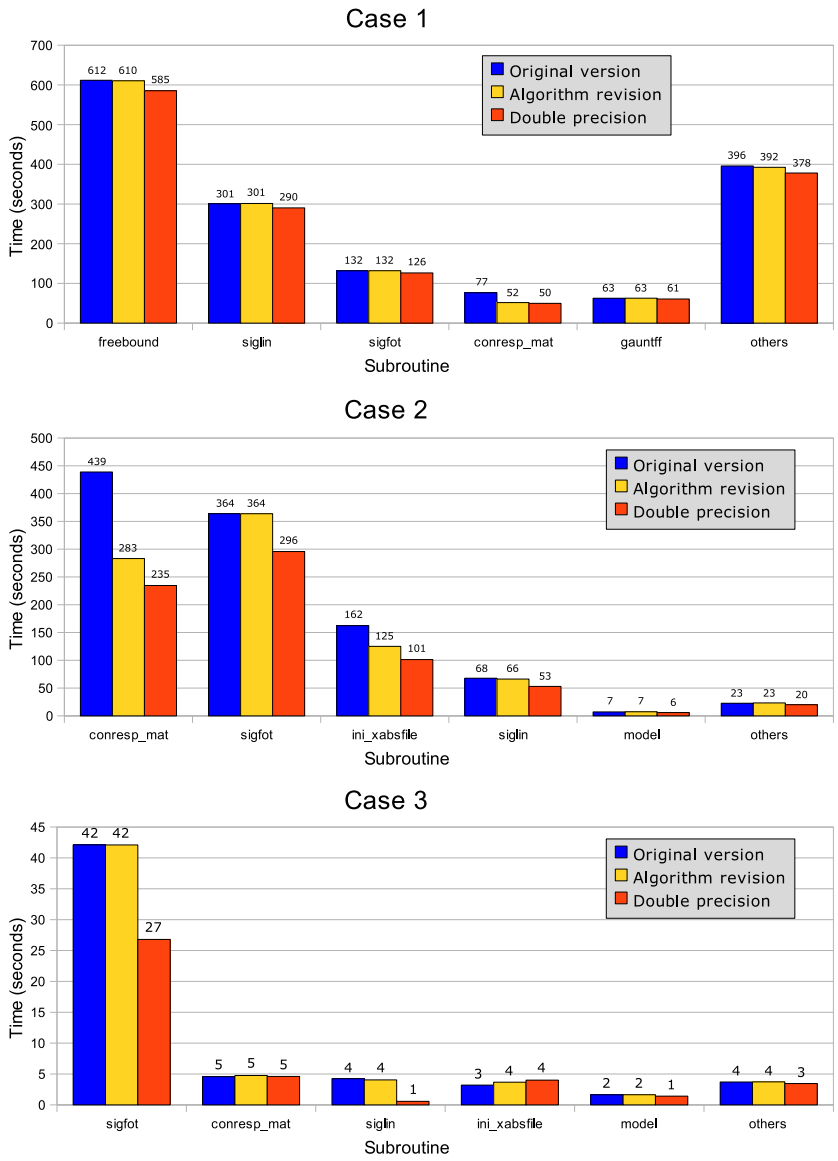

Figure 7.2: Test results with double precision $\vec{s}$ compared to the original program and the latest version which still used single precision.

implementation time and complexity.

One of the oldest and most used sorting algorithms is known as *bubble sort*. The first known analysis of the bubble sort dates from 1956 [Astrachan, 2003]. The algorithm is fairly simple, effectively it traverses the list to be sorted and swaps adjacent items if they are in the wrong order until no more swaps need to be done, indicating that the list is sorted. This has a worst-case complexity of $O(n^2)$, and is therefore not suited for sorting large lists.

7.3.2 Sorting criteria

Obviously, one can sort the response components on different criteria. However, the most logical would be to sort such that either the minimum or the maximum channel represented in each component is increasing.

Components from different regions may be stored in the same structure, so sorting should also be such that components from the same region are adjacent. In other words, one would sort first by minimum or maximum channel in a component and then by region.

7.3.3 Implementation

Sorting is implemented directly after all data has been read from disk. The current implementation can bee seen in Listing 7.4. Note that this implementation can hardly be called efficient, however, it is very easy to read and maintain and it is not expected that sorting will have a significant contribution to the overall processing time.

The implementation in Listing 7.4 consists of two main loops. Each of these loops is terminated if and only if the logical variable *swapped* is false at the end of a loop iteration. If this variable is false, it indicates that no swaps were necessary for the entire list, thus sorting is complete. In the body of both loops adjacent items are swapped if they are in the wrong order according to the applicable sorting criterium.

7.3.4 Results

With previous tests the entire use-cases were run three times and the most time-consuming subroutines were shown individually in the results. However, for the influence of the ordering of the components on performance of the matrix-vector multiplication, one is only interested in performance of said multiplication. Thus, for these tests the program was altered such that it terminated after 100 matrix-vector multiplications, showing the execution time per multiplication.

The results for these tests, shown in Table 7.1, show that there is hardly any influence on performance of the matrix-vector multiplication. Only use-case 1 is approximately 10% slower when the response components are sorted by increasing highest channel present. So apparently, sorting *can* have an influence on performance, but the default ordering for use-case 1 is already optimal by chance.

That there is an influence becomes more apparant when looking at the results for a random order in Table 7.1. Note that a random order does not have to be the worst possible ordering, however, both use-cases 2 and 3 show a definite decrease in performance when the response components are ordered randomly.

```
1  do
2      swapped = .false.
3      do i = 1,r%ncomp-1
4          if (minval(r%comp(i)%group(:)%ic1)
5              .gt. minval(r%comp(i+1)%group(:)%ic1)) then
6          co = r%comp(i)
7          r%comp(i) = r%comp(i+1)
8          r%comp(i+1) = co
9          swapped = .true.
10         endif
11     enddo
12     if (swapped .eq. .false.) exit
13 enddo
14 do
15     swapped = .false.
16     do i = 1,r%ncomp-1
17         if (r%comp(i)%iregion .gt. r%comp(i+1)%iregion) then
18         co = r%comp(i)
19         r%comp(i) = r%comp(i+1)
20         r%comp(i+1) = co
21         swapped = .true.
22         endif
23     enddo
24     if (swapped .eq. .false.) exit
25 enddo
```

Listing 7.4: Sorting the response components.

	Case 1	Case 2	Case 3
default	0.472	21.801	28.719
increasing min channel	0.484	21.799	28.630
increasing max channel	0.551	21.728	28.563
random order	0.500	22.077	29.038

Table 7.1: Timings (in milliseconds) with different sorting of response components. Default means no sorting is applied, i.e. sorting is the same as it was on disk. Min and max channel indicate the lowest or highest channel present in the response component.

What cannot be read from the table with results is the time it takes to sort the response components. For use-cases 1 and 3 this time is relatively small (< 1 second) as they contain only a few response components. For use-case 2, which contains four regions with the amount of components varying from 75 to 234, sorting takes about 5 seconds. This can be considered long, especially since there is hardly any gain.

In conclusion, it is best to leave out sorting of the components for now. However, it may be that once a parallel algorithm is developed sorting is an option again. After all, it may be so that sorting makes it possible to "restrict" processors to certain parts of the result vector $\vec{s}$. If sorting then turns out to result in a significant increase in performance it will be worthwile to look into a faster sorting algorithm.

7.4 Conclusions

Section 5.3 presented baseline measurements for all three use-cases, including the maximum attainable speedup when optimizing only the matrix-vector multiplication. Table 7.2 shows this data again, but now includes the results after all optimizations

	Case 1	Case 2	Case 3
original mat-vec	76.55	438.94	4.56
original total	1640.12	1067.66	59.74
$S_{\max}$	1.05	1.701	1.08
optimized mat-vec	49.77	234.56	4.61
optimized total	1546.52	715.01	41.12
achieved S	1.06	1.49	1.45

Table 7.2: Test results after sequential optimization compared with the original results. $S_{\max}$ is the originally predicted maximum speedup, achieved S is the speedup calculated with these results.

in this chapter.

Table 7.2 shows that use-cases 1 and 3 are faster than one would expect based on the predicted maximum speedup. This is due to the fact that some calculations were converted from single precision to double precision, which improved the entire fitting algorithm instead of just the matrix-vector multiplication, thereby speeding up execution at the algorithmic level.

Use-case 2, although not improved beyond the expected maximum speedup, still comes relatively close to the maximum speedup. Execution time for the matrix-vector multiplication has been almost halved, mainly by taking into account whether δR is used or not.

Ordering of the response components has some influence as well, albeit arbitrarily small in most cases. However, it may be that this influence is bigger when ordering is compared with a "worst-case ordering", which has not been done. This is something to keep in mind when parallelizing.

After optimizing the sequential algorithm and implementation the achieved speedup is better than expected.

Chapter 8

A first parallel version

8.1 Introduction

Now that a solid sequential algorithm has been implemented it is time to start looking at parallelization. All parallelization topics are discussed in this chapter, starting with a short introduction to openMP and shared memory parallelism. With this background, a first and simple parallel version will be made. This first version will then be experimented and improved upon.

Note that in this chapter only the matrix-vector multiplication will be parallelized. This means that measurements from Table 7.2 can be used as baseline measurements and new maximum speedups can be calculated. These can be seen in Table 8.1.

As Table 8.1 shows use-case 2 has the most to gain by parallelizing the matrix-vector multiplication.

	Case 1	Case 2	Case 3
mat-vec	49.77	234.56	4.61
total	1546.52	715.01	41.12
$S_{\max}$ for $P = 2$	1.016	1.196	1.059
$S_{\max}$ for $P = 4$	1.025	1.326	1.092

Table 8.1: Baseline tests before parallelization. P is the number of processors used to calculate the maximum speedup $S_{\max}$.

8.1.1 Testing methods

Obviously, once a parallel version has been created, all tests will have to be run multiple times. One must test the parallel version with one, two, and four processors, and all tests should be run three times. Running all use-cases will then take quite a while, so it would be nice if it was possible to test performance another way.

To do so, a counter will be introduced in SPEX which counts how many times the matrix-vector multiplication is performed. The program will then be made to terminate after a given amount of multiplications and performance will be calculated as the average time used for a matrix-vector multiplication.

The amount of multiplications done before stopping and calculating the average

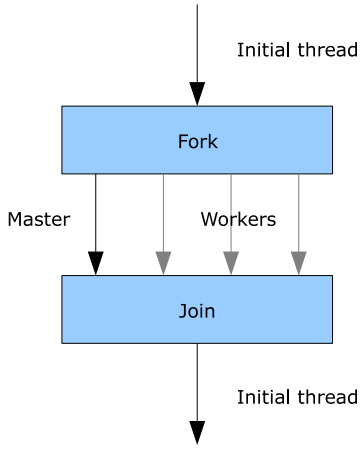

Figure 8.1: The fork-join model as used by openMP.

should not be too low as to preserve precision, but neither should it be too high as this would defeat the purpose. It is decided to perform 200 evaluations. Note that this may include more matrix-vector multiplications, as this will do matrix-vector multiplications for every region and sector. Also note that for use-case 3 this means that the entire use-case will be run, as the use-case in itself is relatively short and performs only about 150 matrix-vector multiplications.

8.2 A short introduction to openMP

The following section is meant to be a very short introduction to openMP to anyone with some programming experience. It is by no means complete; its aim is to provide enough background for the reader to understand the following sections easily without prior openMP knowledge.

8.2.1 The fork-join model

A program running on a computer can consist of multiple threads. Each thread can be seen as a separate entity, able to independently execute instructions. Threads share most of their memory with the main program (or process), but can also have individual memory to store variables private to a thread. When multiple threads are executed simultaneously on multiple processors or processor cores, the encompassing program is called multithreaded or parallel.

OpenMP employs the fork-join model. This means that it essentially starts with a single thread, called the initial thread. This thread forks worker threads at the beginning of a parallel region in the program. At the end of this parallel region the worker threads are joined with the initial thread. This is illustrated in Figure 8.1.

8.2.2 Features of openMP

The openMP API (Application Programming Interface) consists of a collection of compiler directives, library routines, and environment variables. Together, these allow the programmer to control shared memory parallelism through the following means:

- Create teams of threads for parallel processing

- Declare variables such that they are either private to a thread or shared for all threads
- Distribute work among the members of a team of threads in specific ways
- Synchronize threads

Each class (compiler directives, library routines, and environment variables) within the API serves a different purpose. Compiler directives are used by the programmer to indicate which part or parts of the program can be executed in parallel, how the work should be distributed, which variables are private or shared, and so forth.

Library routines can be used by the programmer to query the system status during the execution of the program or to change certain parameters. For example, a programmer could use the library routines to check during execution of the program how many processors are available and take some action depending on the outcome.

The openMP environment variables can be used by anyone who is going to execute an openMP program to control the environment said program will be executed in. For example, a user may use this to limit the maximum number of processors an openMP program may use, or, if applicable, how work should be distributed among threads.

8.2.3 A simple example

A simple openMP "Hello world" example can be seen in Listing 8.1. This example does not use environment variables, but it does show the use of openMP library routines and compiler directives.

Use of an openMP library routine can be seen on line 5 of Listing 8.1. The function *omp_get_thread_num()* is called here to retrieve the id of the thread currently executing this part of the program. As a team of threads was created on line 4, with the compiler directive *!$omp parallel private(id)*, multiple threads will be executing the same code simultaneously. By default, all variables are shared among threads, which is why variable *id* is explicitly marked as private.

A synchronizing directive can be seen on line 7 of Listing 8.1 (*!$omp barrier*). This directive indicates that no code after this directive may be executed before all threads have reached the barrier. This ensures that all threads will always print their hello messages before the thread with id zero prints the total number of threads in lines 8 to 11.

Unlike a barrier, which is a single directive, a parallel region must always have a corresponding ending directive. This directive, *!$omp end parallel*, can be seen on line 12. After this directive, the program will continue executing like a regular program with a single thread.

8.3 A first parallel version

Main question for a first parallel implementation is which part of the algorithm is to be executed in parallel. The current datastructure gives two natural options: at the component or at the energy group level. The first possibility, the component level, has the advantage that each component contains many calculations, therefore introducing little overhead. However, because there may be only a few components, load balancing may not be ideal. Consider for instance use-case 3, which contains

```
1  program hello90
2  use omp_lib
3  integer:: id, nthreads
4      !$omp parallel private(id)
5      id = omp_get_thread_num()
6      write (*,*) 'Hello World from thread', id
7      !$omp barrier
8      if ( id .eq. 0 ) then
9          nthreads = omp_get_num_threads()
10         write (*,*) 'There are', nthreads, 'threads'
11     end if
12     !$omp end parallel
13 end program
```

Listing 8.1: A simple openMP programming example.

only 9 components. Distributing these on 4 processors and assuming that each component will contain approximately the same number of elements will result in one processor evaluating only a single component, while the other 3 will compute two components' contributions.

The other natural parallelization possibility, the energy group level, does not have this disadvantage, as there will always be so many energy groups that load balancing should not be an issue. However, this will induce the overhead of distributing many relatively small tasks among processors.

Since it is unknown how much overhead will be introduced by either option, both options will be left open for now.

8.3.1 Parallelizing at the component level

To create a first parallel version only two things have to be added to the algorithm. First, the parallel section must be defined and the work must be distributed. Since openMP has a special construct to automatically distribute work that is now in a Fortran do-loop, this will be only two compiler directives; one at the start of the parallel region and one at the end.

Second, openMP must be instructed that updates to the result vector $\vec{s}$ may be done only by one thread at a time. After all, if this restriction is not implemented, it is possible that multiple threads fetch $\vec{s}$ from memory simultaneously. Both threads now have the same copy of $\vec{s}$, thus after updating the updates of one thread will be lost. This is known as a *race condition*. Normally, one would have to use a locking mechanism to provide mutual exclusion. However, using openMP one can use the *critical* directive, which indicates that only one thread at a time may execute the region enclosed within at the same time.

However, preventing the race condition with a *critical* directive will be sub-optimal at best. After all, this means that most of the computations are now in a critical section, abandoning the advantages of parallel programming. To solve the race condition when updating $\vec{s}$ in a more elegant way there are in general two main approaches. First, one could define a private variable, have each thread update their own private variable, and add these individual results together at the end of the parallel section. The second possibility is to use the openMP *reduction* clause, which is meant exactly for situations like these. A reduction clause can be used to tell the compiler that a certain variable will be updated by all threads and that the compiler

should prevent race conditions.

In general, the latter option has preference, as one can assume that the compiler is best at optimizing at low levels in order to get the best performance. However, the reduction clause does not support pointers.

In order to get the best of both worlds, a temporary variable $\vec{s}_{\text{local}}$ can be introduced which is not a pointer but the same size as the regular $\vec{s}$. This temporary variable can then be used in a reduction clause.

Parallel implementation

The algorithm essentially has not changed from Algorithm 2 and will therefore not be presented again here. However, specific openMP additions to the source code will be discussed. The parallelized source can be seen in Listing 8.2.

The parallel region in Listing 8.2 is defined in lines 2 (start of the parallel region) and 53 (end of the parallel region). With the start of the parallel region, it is also defined which variables are private for each thread. By default all variables are shared among threads, private variables have to be marked as such explicitly, or the default has to be changed to private. In this case, all variables that are assigned and used only in the parallel section are marked as private, with the exception of the local result vector $\vec{s}_{\text{local}}$, which is kept shared.

The *!$omp do reduction (+:s_local)* directive instructs the compiler to perform an addition reduction operation on the variable *s_local*, preventing the possible race condition. However, since the *reduction* clause does not support pointers, a local variable must be used. The results are copied to the original variable in line 54.

Results

Test results with this parallel implementation are shown in Figure 8.2.

All parallel runs, with the exception of use-case 1 with four processors, are faster than the sequential program. This is expected when more than one processor is used, however, it is not expected when the parallel version is used with only one processor. After all, the overhead introduced by openMP cannot be negative, therefore a parallel implementation run on one processor is always expected to be slower than a sequential version.

The only change that may have a positive influence when the program is run on one processor is the introduction of the local variable $\vec{s}_{\text{local}}$. Apparently, using a local variable for the vector operations is considerably faster than using the "regular" variable. This can be explained by the fact that the regular $\vec{s}$ is stored non-contiguously (that is, it is interleaved in memory with other properties), while $\vec{s}_{\text{local}}$ is.

Even though it would seem logical to conclude that the program performs better for use-cases with response matrices with more elements, this is not directly true. The problem is that use-case 1, which contains relatively few elements, also contains few components. Even worse, the number of elements in the components in use-case 1 is very unevenly distributed, resulting in a sub-optimal distribution of work among processors. This issue will be addressed in the next sections.

```
1  s_local = 0.
2  !$omp parallel private(icomp,c,ch,i,g,h,i1,i2,sie,wie)
3  !$omp do reduction (+:s_local)
4  do icomp = 1,r%ncomp
5    c => r%comp(icomp)
6    if (isect.ne.c%isector) cycle  !skip if not the right sector
7    if (ireg.ne.c%iregion) cycle   !skip if not the right region
8
9    if (r%share_comp) then
10     ch => r%comp(c%sh_comp)
11   else
12     ch => r%comp(icomp)
13   endif
14
15   do i = 1,c%neg
16     g => c%group(i)
17     h => ch%group(i)
18     if (g%nc.eq.0) cycle
19     i1 = g%ie1
20     i2 = g%ie2
21     !-- re-bin the input spectrum
22     sie = sum(sener(i1:i2))
23     !-- prescale the flux
24     if (r%area_scal) then
25       sie = sie * g%relarea
26     endif
27     !-- renorm
28     if (r%renorm(ireg).ne.1.) then
29       sie = sie * r%renorm(ireg)
30     endif
31     !-- do the convolution
32     s_local(g%ic1:g%ic2) = s_local(g%ic1:g%ic2)
33                 + sie * h%resp(:)
34     if (c%usedresp.eq..true.) then
35       !-- re-bin the input spectrum
36       wie = sum(wener(i1:i2))
37                 + dot_product(sener(i1:i2),eg(i1:i2)-g%eg)
38       !-- prescale the flux
39       if (r%area_scal) then
40         wie = wie * g%relarea
41       endif
42       !-- renorm
43       if (r%renorm(ireg).ne.1.) then
44         wie = wie * r%renorm(ireg)
45       endif
46       !-- do the convolution
47       s_local(g%ic1:g%ic2) = s_local(g%ic1:g%ic2)
48                 + wie * h%dresp(:)
49     endif
50   enddo
51 enddo
52 !$omp end do
53 !$omp end parallel
54 s%bin(:)%schan = s_local(:)
```

Listing 8.2: Parallel implementation of the matrix-vector multiplication.

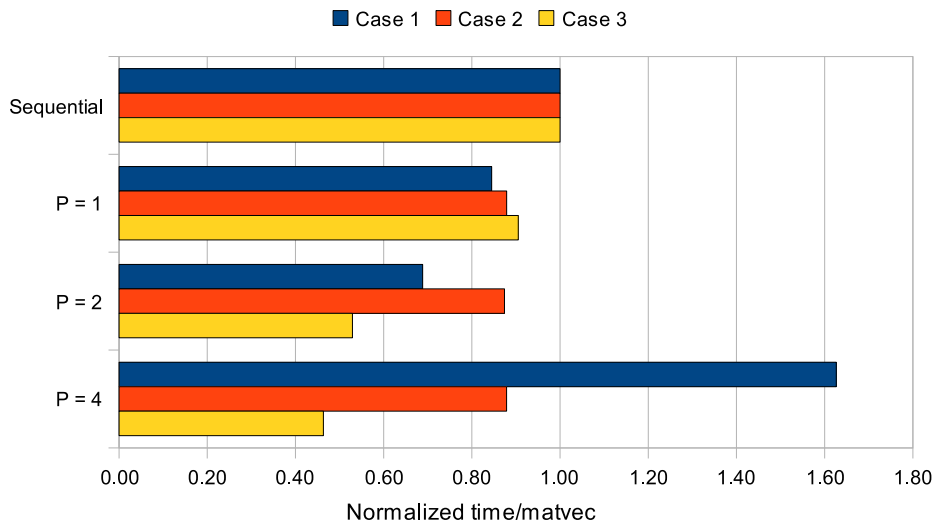

Figure 8.2: Measurement results for the first parallel implementation.

8.3.2 Parallelizing at the energy group level

The main advantage of parallelizing at the energy group level was supposed to be the increase in load balance. However, as Chapter 10 will show it is very well possible to increase the load balance of an implementation that is parallelized at the component level without all the additional overhead required when parallelizing at the energy group level. Because of this, parallelization at the energy group level will not be implemented.

Chapter 9

Proposed improvements to the parallel implementation

The purpose of this chapter is comparable to that of Chapter 7, but where Chapter 7 proposed optimizations for the sequential version, this chapter proposes improvements for the parallel version.

The proposed improvements will be implemented and put to the test in Chapter 10.

9.1 Response component ordering

As Section 7.3 showed, ordering of the response components did not have a significant positive influence on the performance of the sequential program. However, it did have some influence, which, together with the fact that it may very well be that there is a positive influence on a parallel version, are the reasons to try this again.

Recall that the original reason to order response components was availability of $\vec{s}$ in cache memory. There are two reasons to believe that this might work for the parallel version while it did not before.

First, by using a local variable $\vec{s}_{\text{local}}$ chances that all or part of the vector fits in cache are much bigger. After all, the local version is just a consecutive list of double precision variables, while the regular $\vec{s}$ is stored interleaved with other data.

The second reason is that when running on two or more processors, each processor has some private cache memory, which can be accessed faster than the shared cache memory. Thus, there is the possibility that by ordering components and assigning them to threads such that each thread will be mainly concerned with its own part of $\vec{s}_{\text{local}}$, the amount of data that must stay in cache for efficient operation will be reduced.

9.2 Creating smaller response components

One of the problems with use-cases 1 and 3 in particular is that it is impossible to distribute the data among threads in such a way that each thread processes approximately the same amount of matrix elements. This is due to the fact that the response matrices for these use-cases are divided in relatively few components,

and, for use-case 1, the amount of matrix elements in these components are very unevenly distributed.

This can be resolved by introducing a maximum size $n_{\max}$ for response components. If a component contains more than $n_{\max}$ elements, it should be split into the biggest possible even amount of components with less than $n_{\max}$ elements.

Obviously, splitting components could be done in different ways. One could split components horizontally, vertically, both, or any other more exotic way. To ensure that cache efficiency for $\vec{s}_{\text{local}}$ is optimal, it makes sense to split components only vertically, thereby ensuring that split components which access the same parts of $\vec{s}_{\text{local}}$ are never distributed among different threads.

On the other hand it seems logical to split components horizontally, because for every row in every component there is some overhead in certain operations such as renormalization, meaning that splitting the components vertically increases overhead while splitting horizontally does not. It cannot be predicted with certainty which way of splitting gives the best result, therefore it may be best to try both.

It is expected that doing so will yield significant improvements in parallel speedup for use-case 1 and 3, while no significant improvement, if any at all, is expected for use-case 2. After all, use-case 2 already contains enough components to allow for proper load-balancing.

9.3 Different scheduling

Until now, the only instruction given to the compiler with the *omp do* directive was the reduction clause, trying to prevent the use of critical sections. Another clause that may greatly affect performance is the *schedule* clause, which tells the compiler how the different loop iterations should be distributed among threads, which may be a powerful tool, especially in combination with ordered response components.

OpenMP supports the following four types of scheduling:

static The static scheduling method divides iterations in continuous chunks of specified and fixed size, which are then distributed among threads in a round-robin fashion. The last chunk may contain a smaller number of iterations. Choosing a chunksize of 1 will result in a cyclic distribution, while not specifying the chunksize will result in a block distribution. The cyclic and block distribution concept is clarified in Figure 9.1.

dynamic Just as with static scheduling, iterations are divided in continuous chunks of a specified size. These are then assigned to threads on individual thread requests. A thread will execute one chunk of iterations and then request a new one. Contrary to static scheduling, the chunksize defaults to 1 when it is not specified.

guided The guided scheduling type is similar to the dynamic one, i.e. threads request iteration chunks instead of having them assigned as with the static scheduling type. However, the size of each chunk is now proportional to the amount of iterations left in the queue divided by the number of threads. The minimum size of a chunk can be defined by the chunksize parameter; it defaults to 1.

auto Auto leaves the scheduling decision to the compiler. This means that the used scheduling will be implementation dependent. A chunksize cannot be

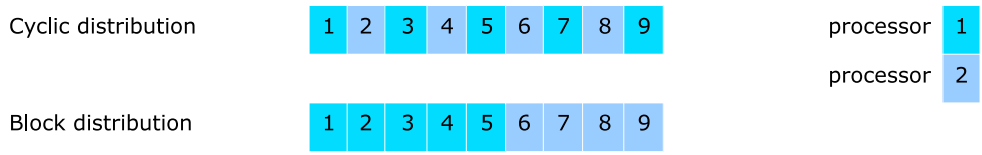

Figure 9.1: Illustrated cyclic and block distribution. Numbers in the table indicate iteration numbers in a loop scheduled with the openMP static scheduling method with chuncksize 1 (cyclic) or no chunksize specified (block).

specified with this scheduling parameter. This scheduling type was introduced in openMP 3.0.

runtime This is not really a scheduling type, but it is one of the possible options that can be specified. Specifying runtime as the scheduling parameter well let openMP decide the scheduling type at runtime through the *OMP_SCHEDULE* environment variable.

The current implementation does not specify a scheduling parameter and no default scheduling type is specified in the openMP specifications; the default is implementation specific [OpenMP ARB, 2008, p. 311]. The implementation of the used Intel Fortran compiler defaults to static scheduling with a chunksize approximately equal to the total amount of iterations divided by the number of available threads [Intel Corporation, 2009b, p. 1343-1344], i.e. a block distribution.

Recall from Section 5.2 that both use-case 1 and 2 use response matrices containing components that differ greatly in size. For instance, approximately half the components of the response matrix for region 3 of use-case 2 are very small compared to the other half. This means that, in a worst-case scenario where the response components are ordered by the amount of matrix elements they contain, the default scheduling is about the worst possible scheduling type one can imagine. After all, if for instance two threads are used, static scheduling will assign all large components to the first thread while all small components will be assigned to the second thread, which obviously causes huge load imbalance.

Note that this will also occur when the response components are stored in an ordered fashion as proposed in Section 9.1, albeit not in a completely worst-case way. This happens because the small response components are clustered together in a small part of the response matrix, causing them to be clustered in the ordered components as well, and thus chances are that the smaller components will be assigned to a single thread.

9.3.1 Manual iteration scheduling

Drawback of the scheduling types proposed in this section is that it cannot be guaranteed that each processor processes a continuous block of the local result vector $\vec{s}_{\text{local}}$, thus not fully utilizing the cache memory.

The only way to be absolutely sure about the distribution of components among processors is by doing said distribution manually. This should also save on some overhead, as this scheduling can probably be done just once at an initialization stage. Doing so would mean that the *omp do* directive is no longer necessary, which unfortunately also means that the reduction clause can no longer be used. Thus, when using manual iteration scheduling, adding the local results in $\vec{s}_{\text{local}}$ will have to be done manually as well.

To optimize load balance, iteration scheduling should be such that each thread gets assigned approximately the same number of non-zero elements in the response matrix. This does not imply that the result vector $\vec{s}$ will be nicely distributed among the available threads. Furthermore, using just the non-zero elements in the response matrix as a scheduling criterium is not completely optimal, as there is relatively more overhead for smaller components. Even so, this is the method of choice since it is a relatively easy criterium without any unknown factors, such as the ratio between the size of a component and its overhead.

Chapter 10

Implementing improvements to the parallel algorithm

This chapter deals with implementation and testing of the improvements proposed in Chapter 9.

10.1 Response component ordering

Response component ordering was already implemented for the sequential version in Section 7.3, but it was taken out again because no significant positive effects were shown. This implementation can now be almost directly reused as it was completely stand-alone, i.e. it did not interfere with any other parts of the program.

However, there are some complications for a parallel version, especially for use-case 2. This is because use-case 2 contains four regions with approximately the same number of components. The sorting algorithm sorted components not only in order of their location in the response matrix, but also in order of the regions, while the matrix-vector multiplication algorithm processes only one region at a time. This means that, with the current default scheduling algorithm, work will be distributed such that only one thread processes approximately one region.

For clarification, consider a situation where a response matrix contains two regions which contain ten components each, which are sorted by region. If these are processed using two threads, each thread will process ten components, but these ten components all belong to a single region. The program is parallelized at the component level and regions are processed sequentially, hence there will be no speedup.

To solve this problem, regions should be sorted in a cyclic fashion. Instead of sorting such that all components for region one are stored first, then all components for region two, then region three, and so forth, regions should be stored in a cyclic pattern. This will lead to a much better work distribution, as long as the different regions contain approximately the same number of components.

Test results for the sequential version showed it was in general most efficient to order the response components with the lowest channel in the components ascending, so this order will be used for the parallel version as well.

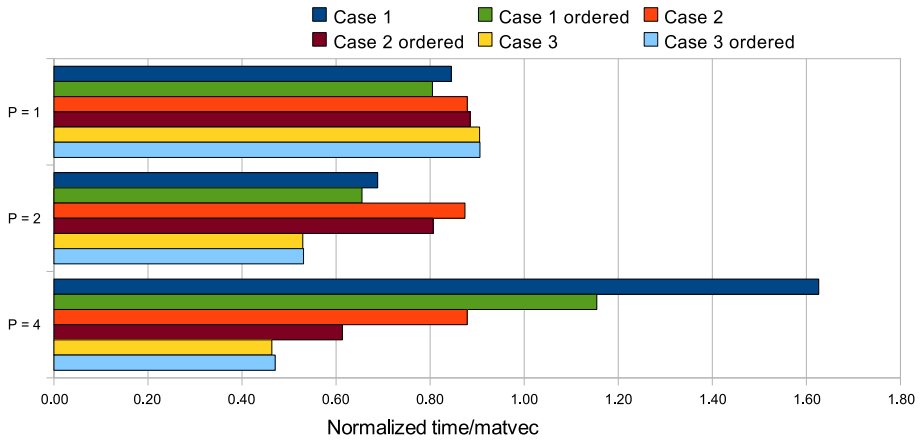

Figure 10.1: Test results with components ordered compared to the previous parallel version (Section 8.3.1). Note that timings are still normalized with respect to the sequential version even though results from the sequential version have not changed and are not shown.

10.1.1 Results

Figure 10.1 shows test results with the components ordered compared with the previous parallel version (Section 8.3.1), which had no specific ordering in the response components.

As can be seen in Figure 10.1, results for use-case 1 have improved significantly, especially when four processors are used. This shows that the expected increase in performance is indeed there as single threads access only a specific part of $\vec{s}_{\text{local}}$. However, parallel speedup for use-case 1 is still suboptimal, which is probably mainly caused by the fact that, given the current components in the response matrix for this case, it is impossible to achieve proper load balance.

Results for use-case 2 have improved as well. However, this is probably mostly caused by increased load-balance due to component ordering. Still, parallel speedup for this case is far from optimal. Given the distribution among the different response components for use-case 2 it should theoretically be possible to get near-optimal load balance. Unfortunately, by distributing the iterations in a block fashion as is currently done, the current version will not even come close to any theoretical optimum, because the sizes of the components differ greatly, correlating to their location in the matrix.

Use-case 3 shows no significantly different results when the response components are ordered, which can be explained by the fact that use-case 3 contains only 9 components.

Ordering the response components seems to be a step in the right direction, but more work is necessary to get near-optimal load balance and good parallel speedup.

10.2 Manual iteration scheduling

One possible way to improve load balance is by scheduling iterations manually, i.e. manually define which thread processes which loop iterations. Recall that currently

the openMP implementation default is being used, which means iterations are distributed among threads in a block fashion. This will be fine in many cases, but in all use-cases available this will not suffice, because not all components contain even approximately the same amount of work. Thus, loop iterations may be distributed evenly, but it does not follow from this that work is distributed evenly.

For use-case 2 there is another complication. This use-case contains multiple regions, but these regions do not contain the same amount of components. This means that, even with the cyclic region ordering introduced in Section 10.1, load distribution will be imbalanced.

Ideal scheduling will distribute iterations such that every thread processes approximately the same amount of matrix elements. For use-cases 1 and 3 this is fairly straightforward, as one can simply count the total number of elements in the matrix n_{total} and then assign components to the first thread until the number of elements in these components is greater than or equal to n_{total}/P, with P the total amount of threads that will be used. When done with the first thread, the same should be done with the second, starting with the first component not yet assigned to another thread. This should be repeated until all components are assigned.

This approach is easy since it results in a block distribution. The main difference is that the sizes of the blocks do not have to be approximately equal. Since every thread will always process a contiguous block of components the only thing that must be stored are starting and ending points for every processor.

Use-case 2, which contains four regions, presents some more problems. After all, it is only possible to achieve near optimal load-balance with a block-type distribution if:

- components are stored such that regions occur in a cyclic fashion,
- every region contains the same amount of components,
- every component contains approximately the same amount of matrix elements.

This is definitely not the true for use-case 2. A cyclic distribution may solve this problem, but it would not be possible to use a cyclic distribution and guarantee near optimal load-balance.

A better solution would be to remove the cyclic ordering by region from the response components and order them as was done in the sequential version in Section 7.3, i.e. store all components from one region adjacently. This section of the total components can then be considered a sub-matrix, which can be distributed as described above for use-cases 1 and 3.

10.2.1 Algorithm

The sorting algorithm will not be discussed here, as it is almost equal to the ones used in Section 7.3 and Section 10.1. Finding the first and last iteration that every thread should use is fairly straightforward, as Algorithm 3 shows. It looks more complex because the algorithm must distinguish between different regions, meaning that almost all variables are indexed by region.

The algorithm starts by finding the boundaries for every region, and by setting start and end of a region as initial values for n_{first} and n_{last} in lines 9 to 19 of Algorithm 3. It then continues by dividing the regions among processors, by counting the elements in every component and testing whether the elements from all components so far assigned to a processor exceed the total amount of elements divided

by the amount of processors available. If so, following components are assigned to the next processor, until all components are assigned. This is done in lines 20 to 31.

Note that manual iteration scheduling means the *!$omp do* directive is no longer necessary. Instead, each processor uses a customized do-loop based on their processor identifier. This also means that the *reduction* clause can no longer be used; this is replaced by a critical section after the do-loop. Relevant part of the source code can be seen in Listing 10.1.

Algorithm 3 Finding iteration bounds for every thread.

1: $n_{\text{reg}} \leftarrow$ number of regions in response matrix
2: $n_{\text{comp}} \leftarrow$ number of components in response matrix
3: $P \leftarrow$ number of processors available
4: $p_{\text{count}}(i) \leftarrow 1 \qquad \forall i \in [1, n_{\text{reg}}]$
5: $nz_{\text{total}}(i) \leftarrow$ amount of elements in region $i \qquad \forall i \in [1, n_{\text{reg}}]$
6: $nz_{\text{count}}(i) \leftarrow 0 \qquad \forall i \in [1, n_{\text{reg}}]$
7: $n_{\text{first}}(i, j) \leftarrow n_{\text{comp}}$ where $\{(i, j) : 1 \leq i \leq n_{\text{reg}} \times 1 \leq j \leq P\}$
8: $n_{\text{last}}(i, j) \leftarrow n_{\text{comp}}$ where $\{(i, j) : 1 \leq i \leq n_{\text{reg}} \times 1 \leq j \leq P\}$
9: **for all** components (i_{comp}) **do**
10: $i_{\text{reg}} \leftarrow i_{\text{comp}}$(region)
11: **for** $t = 1$ to P **do**
12: **if** $i_{\text{comp}} < n_{\text{first}}(i_{\text{reg}}, t)$ **then**
13: $n_{\text{first}}(i_{\text{reg}}, t) \leftarrow i_{\text{comp}}$
14: **if** $i_{\text{reg}} > 1$ **then**
15: $n_{\text{last}}(i_{\text{reg}}, t) \leftarrow i_{\text{comp}} - 1$
16: **end if**
17: **end if**
18: **end for**
19: **end for**
20: **for all** components (i_{comp}) **do**
21: $i_{\text{reg}} \leftarrow i_{\text{comp}}$(region)
22: $nz_{\text{count}} \leftarrow nz_{\text{count}} + nz(i_{\text{comp}})$
23: **if** $nz_{\text{count}}(i_{\text{reg}}) > \frac{nz_{\text{total}}(i_{\text{reg}})}{P}$ **then**
24: $n_{\text{last}}(i_{\text{reg}}, p_{\text{count}}(i_{\text{reg}})) \leftarrow i_{\text{comp}}$
25: $p_{\text{count}}(i_{\text{reg}}) \leftarrow p_{\text{count}}(i_{\text{reg}}) + 1$
26: **if** $p_{\text{count}}(i_{\text{reg}}) \leq P$ **then**
27: $n_{\text{first}}(i_{\text{reg}}, p_{\text{count}}(i_{\text{reg}})) \leftarrow i_{\text{comp}} + 1$
28: **end if**
29: $nz_{\text{count}}(i_{\text{reg}}) \leftarrow nz_{\text{count}}(i_{\text{reg}}) - \frac{nz_{\text{total}}(i_{\text{reg}})}{P}$
30: **end if**
31: **end for**

10.2.2 Results

The changes made in this section should achieve better load balance, which in turn should result in better parallel speedup. To see whether the changes succeeded in this, timings will be presented, but first of all the actual load balance should be inspected.

```
1  !$omp parallel private(icomp,c,ch,i,g,h,i1,i2,sie,wie,s_local)
2    s_local = 0.
3    p = omp_get_thread_num()
4    do icomp = r%nfirst(ireg,p+1),r%nlast(ireg,p+1)
5      ! perform calculations
6    enddo
7    !$omp critical
8      s%bin(:)%schan = s%bin(:)%schan + s_local(:)
9    !$omp end critical
10 !$omp end parallel
```

Listing 10.1: Using manual iteration scheduling.

Load balance

Table 10.1 shows how many matrix elements each thread processed for a given use-case when using 1, 2 or 4 threads. It also shows the load imbalance, defined as

$$\max\left\{w_p - \frac{w_s}{P} : 1 \leq p \leq P\right\},$$

where P is the total number of processors used, w_p is the work for a processor p, and w_s is the total amount of work.

Table 10.1 shows that load imbalance for use-case 1 decreases slightly when using manual iteration scheduling. Obviously it is virtually impossible to increase load-balance further, given the distribution of matrix elements among components for use-case 1.

Use-case 2 contains four regions which are processed sequentially. Therefore, load balance should be considered separately for each region. Table 10.1 shows that load balance improves greatly for use-case 2, up to orders of magnitude for region 3. Load balance is not optimal, but considering the total amount of elements for use-case 2 it gets very close.

For use-case 3 there is no difference in load balance when using manual iteration scheduling. This is not surprising, as use-case 3 contains only 9 components with approximately the same number of elements. Load balance could only be improved if the components of use-case 3 would be split into smaller components.

Thus, it is expected that execution time measurements will show improvements in parallel speedup for use-case 2, while performance for use-case 1 and 3 will be approximately unchanged.

Performance

Results from performance tests can be seen in Figure 10.2, which compares results with manual iteration scheduling with results from the previous version (Section 10.1).

Figure 10.2 shows that all use-cases are slightly faster when just one processor is used. This can be explained by the fact that there will be less overhead due to removal of the *!$omp do* directive. However, the difference is so small that it may also be caused by simple measurement uncertainties.

As expected, performance for use-cases 1 and 3 is approximately unchanged, while performance for use-case 2 has increased. Although the increase in parallel speedup

			Work (# of elements)		Load imbalance	
Case/region	P	Thread ID	openMP	Manual	openMP	Manual
Case 1	1	0	113617	113617	0	0
	2	0	92493	92480	35685	35672
		1	21124	21137		
	4	0	92487	92480	64083	64076
		1	6	2		
		2	125	1		
		3	20999	21134		
Case 2 region 1	1	0	6748510	6748510	0	0
	2	0	1044337	3692389	2329910	318134
		1	5704173	3056121		
	4	0	1044102	1748115	1686893	257147
		1	235	1944274		
		2	3207758	1439857		
		3	2496415	1616264		
Case 2 region 2	1	0	5968305	5968305	0	0
	2	0	5903154	3187778	2919002	203626
		1	65151	2780527		
	4	0	1608367	1608367	1492076	116291
		1	4294787	1579411		
		2	65151	1514670		
		3	0	1265857		
Case 2 region 3	1	0	2539149	2539149	0	0
	2	0	649604	1290460	619971	20886
		1	1889545	1248689		
	4	0	352079	649340	1253419	14553
		1	297525	641120		
		2	1339	628385		
		3	1888206	620304		
Case 2 region 4	1	0	2308146	2308146	0	0
	2	0	1607574	1295413	453501	141340
		1	700572	1012733		
	4	0	263648	604940	577037	113437
		1	1437838	690473		
		2	606660	496835		
		3	0	515898		
Case 3	1	0	2853383	2853383	0	0
	2	0	1597345	1597345	170654	170654
		1	1256038	1256038		
	4	0	913469	913469	200123	200123
		1	683876	683876		
		2	664694	664694		
		3	591344	591344		

Table 10.1: Load (im)balance with automatic openMP scheduling compared to load (im)balance with manual iteration scheduling. Load imbalance values are rounded to the nearest integer.

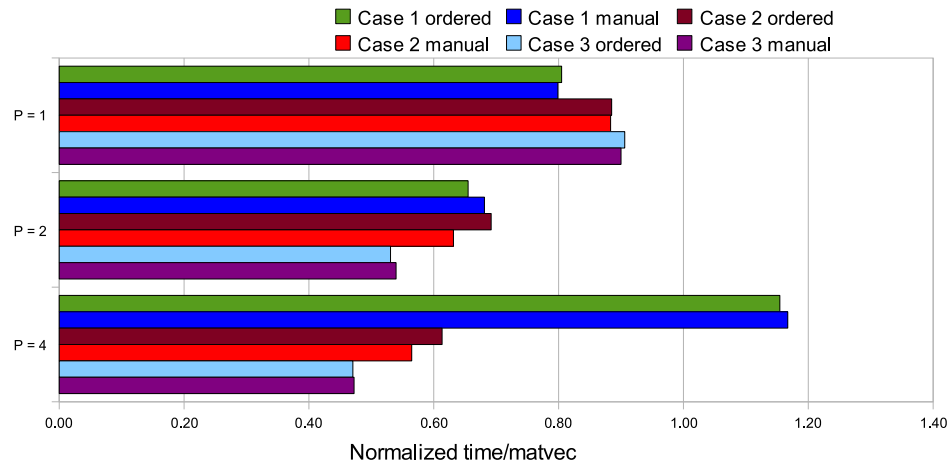

Figure 10.2: Performance of the matrix-vector multiplication with manual iteration scheduling compared to the previous version (Section 10.1).

seems very small, it is still about 9% when using two processors and 8% when using four processors.

10.3 Creating smaller response components

Recall from Section 9.2 that splitting relatively large components into smaller ones should enable improved load balancing and thus improved parallel speed-up, especially for use-cases 1 and 3. The question is when a component is relatively large.

As this is a load balancing issue it is proposed to split components if they contain more than $n_{\max}$ components, where $n_{\max}$ is defined to be a fraction f of the total amount of matrix elements n_{total}. Doing so makes sure that components can be distributed among processors in a relatively balanced fashion, depending on f.

10.3.1 Algorithms

Splitting components horizontally

Because data in components is stored in rows instead of columns, splitting the components horizontally will be simpler than splitting components vertically. An algorithm to split components this way can be relatively simple, the one that will be implemented can be seen in Algorithm 4. In this and other algorithms in this section $C(i)$ denotes component i in the response matrix, f is the fraction of the total amount of nonzero elements that determines whether a component should be split or not.

Essentially, Algorithm 4 performs one of two actions for every component. If the number of elements in a component is less than or equal to $nz_{\text{total}}f$ the component should not be split; it is directly copied from C to C_{new}. If the amount of elements in a component is greater than $nz_{\text{total}}f$, the component should be split. In this case, energy groups of this component are assigned to one component as long as the amount of elements assigned to this component is less than half of the elements of the original component; the remaining energy groups are assigned to another component.

The algorithm repeats until none of the components contains more than $nz_{\text{total}}f$ elements.

Algorithm 4 Splitting components horizontally

```
1: done ← false
2: nz_total ← total number of nonzero elements in matrix
3: while done = false do
4:    k ← 0
5:    nz_i ← nonzero elements in component i
6:    for i ← 0 to n_C do
7:       if nz_i ≤ nz_total f then
8:          C_new(k) ← C(i)
9:          k = k + 1
10:      else
11:         s ← 0
12:         for all energy groups G in component C(i) do
13:            s ← s + size(G)
14:            if s < nz_i/2 then
15:               add group G to C_new(k)
16:            else
17:               add group G to C_new(k + 1)
18:            end if
19:         end for
20:         k = k + 1
21:      end if
22:   end for
23:   if k = n_C then
24:      done ← true
25:   end if
26:   C = C_new
27: end while
```

Splitting components vertically

Splitting components vertically is very similar to splitting horizontally. However, because every energy group contains a row of elements it is not just a matter of assigning different energy groups to different components. Instead, energy groups will have to be split such that part of a group can be assigned to one component while the other part will be assigned to another component.

This adds additional bookkeeping to the process, because splitting energy groups means that the other properties of this group, such as the lowest or highest energy in the group, may change as well. Splitting one group into multiple groups also makes it harder to make a split such that the two parts of a split component both contain approximately the same amount of matrix elements.

Because of this, it is decided to make the vertical split always in the middle of a component's boundaries. This means that it is possible that an irregularly filled component is split sub-optimally and will be split more than absolutely necessary. However, it will also ensure a readable and relatively simple algorithm, and since data distribution in most components seems to be quite evenly spread this should not present any real problems.

		Vertical split			Horizontal split		
f	P	Case 1	Case 2	Case 3	Case 1	Case 2	Case 3
1	2	35672	683986	170654	35672	683986	170654
	4	64076	501428	200124	64076	501428	200124
0.5	2	22589	683986	170654	35672	683986	170654
	4	19198	501428	200124	17886	501428	200124
0.25	2	8295	683986	170654	12512	683986	170654
	4	19198	501428	200124	17786	501428	200124
0.1	2	8304	683986	114655	910	683986	131235
	4	7704	501428	60430	546	501428	85280
0.05	2	3664	221930	68522	910	230756	72996
	4	2192	239662	60430	546	240466	65525

Table 10.2: Load imbalance when splitting components vertically compared to load imbalance when splitting components horizontally. For use-cases with multiple regions the total load imbalance is shown.

The algorithm is very similar to that for horizontal splitting as shown in Algorithm 4 and will therefore not be discussed in detail.

10.3.2 Results

Load balance

Load imbalance results with different f can be seen in Table 10.2. This includes results for splitting horizontally and for splitting vertically.

Table 10.2 clearly shows there is very little difference in load balance when splitting horizontally or vertically. There is a small difference, which can be explained by the fact that splitting in another direction may result in a slightly different element count for the split components.

Load balance for use-case 1 improves most, up to an order of magnitude. This is as expected, as load balance for this use-case previously was very bad. Improvements for use-cases 2 and 3 are less and occur only when f gets smaller (0.1 for use-case 3, 0.05 for use-case 2).

From the results in Table 10.2 it can be expected that use-case 1 will be the one that benefits most from splitting the components, be it vertically or horizontally. For use-cases 2 and 3 there may be some improvement when f is small enough, but it is unclear whether the improvement will be enough to balance any additional overhead caused by having additional components.

Performance

Results for all use-cases with vertical component splitting can be seen in Figure 10.3. The same f-values have been used as with the load balance tests.

Results when splitting components vertically are disappointing. Figure 10.3 shows that in general performance *decreases* when splitting components vertically. The only plausible explanation is that because the algorithm calculates the vector to multiply with for every energy group within every component, splitting components vertically induces a lot of overhead as this vector has to be calculated twice for

every split. Unfortunately, this is not easily remedied because the multiplication vector can have different values depending on properties of the energy group, such as possible renormalization and scaling.

Vertical component splitting does appear to enable better parallel speedup. However, better speedup is not worth much when execution time has about tripled when using one processor.

This is not an issue when splitting components horizontally, which is reflected in the results shown in Figure 10.4. These results show definite improvements, especially for use-case 1, which is as expected. However, use-cases 2 and 3 show some, albeit very small, improvements as well when using $f = 0.05$.

10.4 Conclusions

Figure 10.5 shows the final results of parallelizing the matrix-vector multiplication compared to the optimized sequential version.

During the parallelizing process the result vector $\vec{s}$ was changed to a local variable. This caused the program to be considerably faster for all use-cases, even if only one processor is used. This is due to the fact that the original result vector was stored in memory interleaved with other properties belonging to every entry in the result, while the local variable is stored contiguously. Because of this, the use of cache memory is optimized and all versions show an improvement.

Still better performance was achieved by ordering the response components. One reason for this is that it caused access of the result vector to be sequential and every processor accesses a contiguous block of the result vector if the response components are ordered. Another reason is that ordering the response components may have inadvertently improved load balance.

Load balance was improved further with manual iteration scheduling, mainly for use-case 2. Use-cases 1 and 3 did not directly benefit from this due to their component structure. However, this step increased performance for use-case 2 by approximately 8%.

Even better load balance was achieved when components where split such that every component contains at most a fraction f of the total amount of elements. Splitting the components vertically introduced too much additional overhead, parallel speedup was better but overall performance degraded. However, splitting components horizontally had clear advantages, mainly for use-case 1.

Table 10.3 shows absolute results after optimization compared to the optimized sequential version, including maximum parallel speedups and achieved speedups. Unfortunately, achieved speedups are significantly less than the maximum speedups, which cannot be accounted for by the matrix-vector multiplication only. As one can expect, the overhead introduced mainly by sorting and splitting response components causes a huge performance penalty.

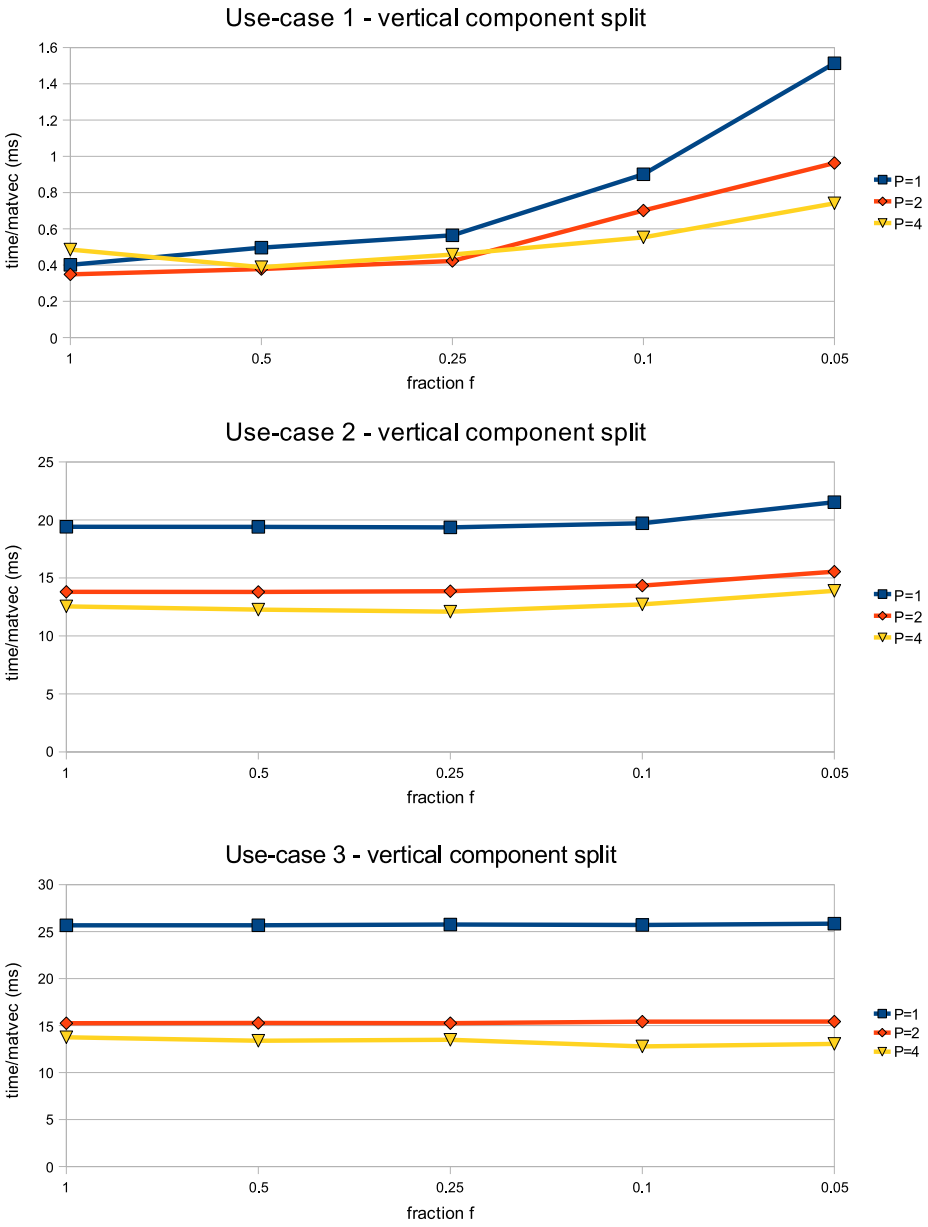

Figure 10.3: Test results for different f when splitting components vertically.

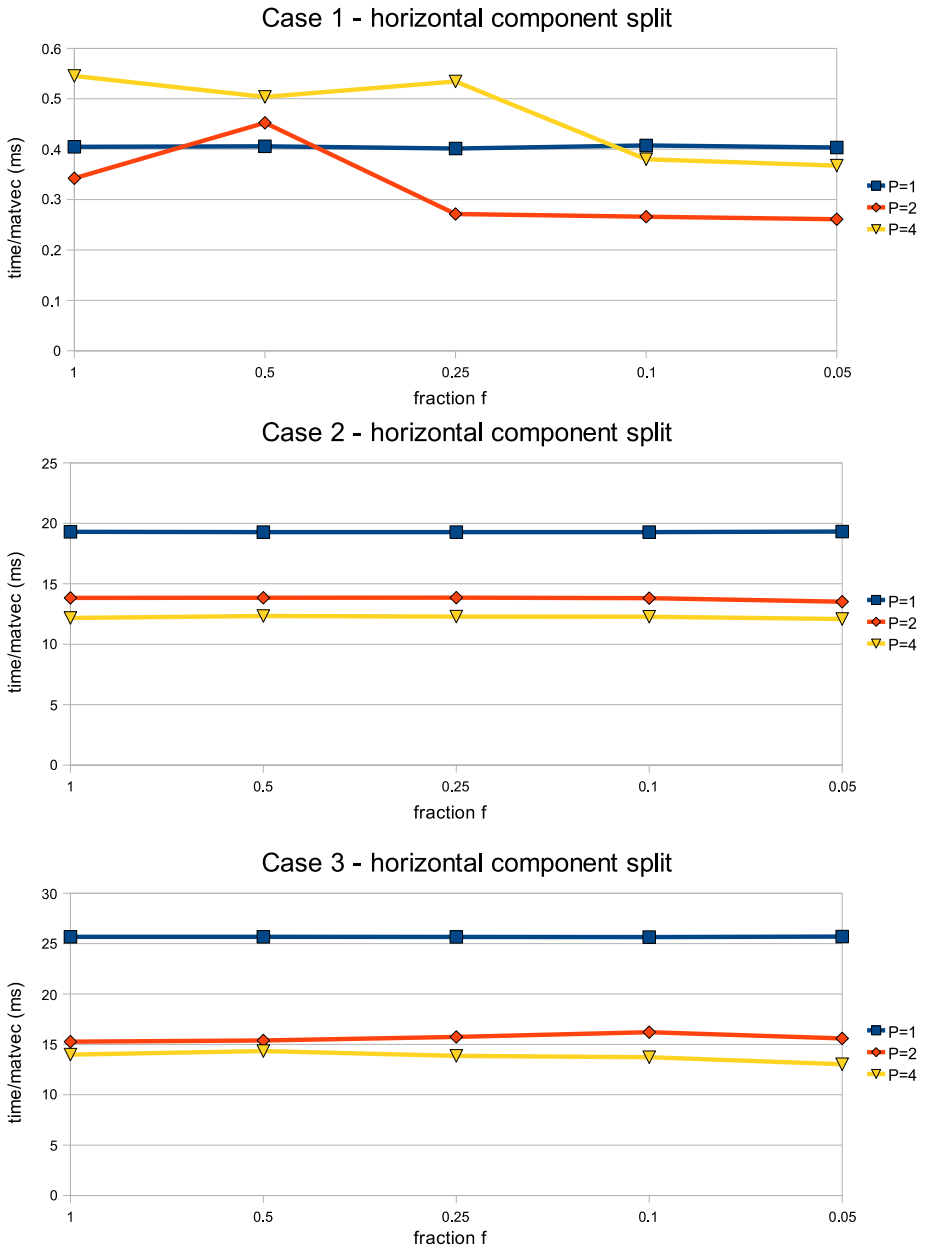

Figure 10.4: Test results for different f when splitting components horizontally.

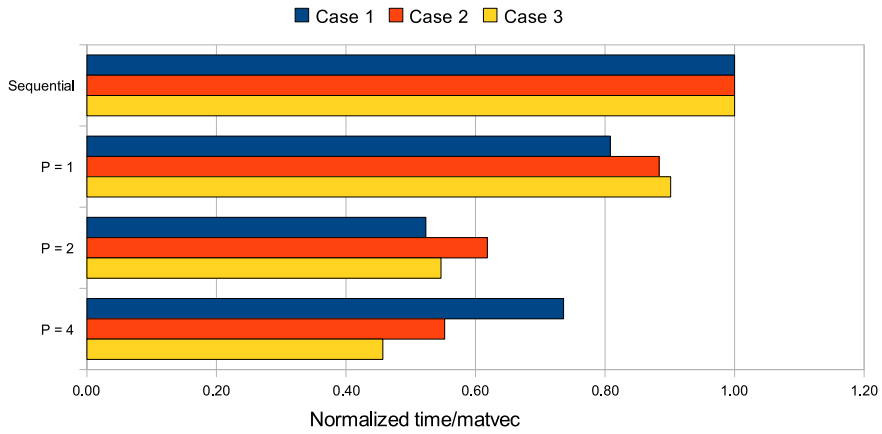

Figure 10.5: Final results of parallelizing the matrix-vector multiplication compared to the optimized sequential version.

	Case 1	Case 2	Case 3
sequential mat-vec	49.77	234.56	4.61
sequential total	1546.52	715.01	41.12
$S_{\max}$ for $P = 2$	1.016	1.196	1.059
$S_{\max}$ for $P = 4$	1.025	1.326	1.092
mat-vec with $P = 1$	43.95	206.20	3.927
total with $P = 1$	1579.88	731.14	41.69
mat-vec with $P = 2$	27.19	144.68	2.373
total with $P = 2$	1562.04	673.64	40.45
mat-vec with $P = 4$	25.24	129.88	2.061
total with $P = 4$	1558.25	652.90	40.00
achieved S for $P = 2$	0.990	1.061	1.017
achieved S for $P = 4$	0.992	1.095	1.028

Table 10.3: Absolute results after parallelization compared to the sequential (optimized) version.

Chapter 11

Conclusions

The sequential algorithm was optimized mainly by including a check to see whether the matrix δR was actually used. Because two out of three cases do not use δR, this optimization only improved execution times for these two use-cases.

Precision of the calculations was increased by using a double precision result variable. This resulted in faster convergence of the fitting procedure, which in turn caused all subroutines used by the fitting procedure to require less time. Note that the individual routines did not really execute faster, but because the fit converged faster and thus needed less iterations, subroutines were no longer executed as often as before.

Ordering the response components did not have much influence, although it may be possible that the original ordering was already good by accident.

A parallel version was created, which showed speedup even if only one processor was used. It was determined that this was due to the fact that cache use was optimized by using a local variable for the result vector.

Load-balance for the parallel version was improved by ordering the response components and manually scheduling iterations. Even so, load-balance was not yet optimal for most cases due to the component properties. By splitting large components load-balance was increased further.

Results for the complete program can ben seen in Figures 11.1, 11.2, and 11.3.

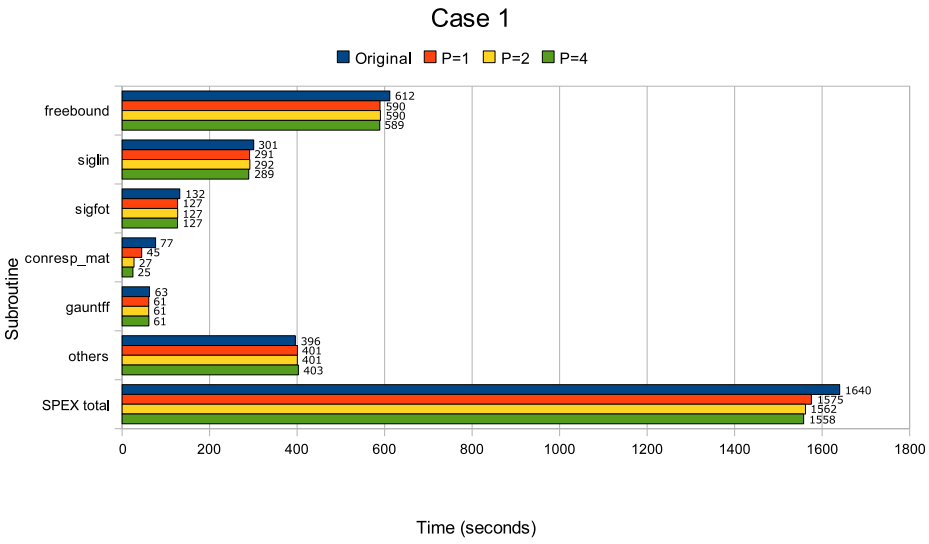

Figure 11.1: Results after optimization of the matrix-vector multiplication for use-case 1.

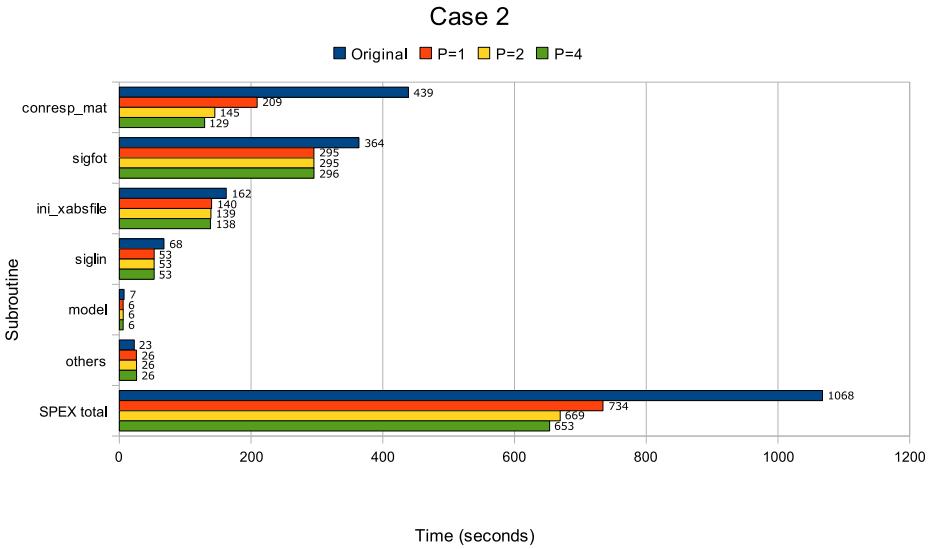

Figure 11.2: Results after optimization of the matrix-vector multiplication for use-case 2.

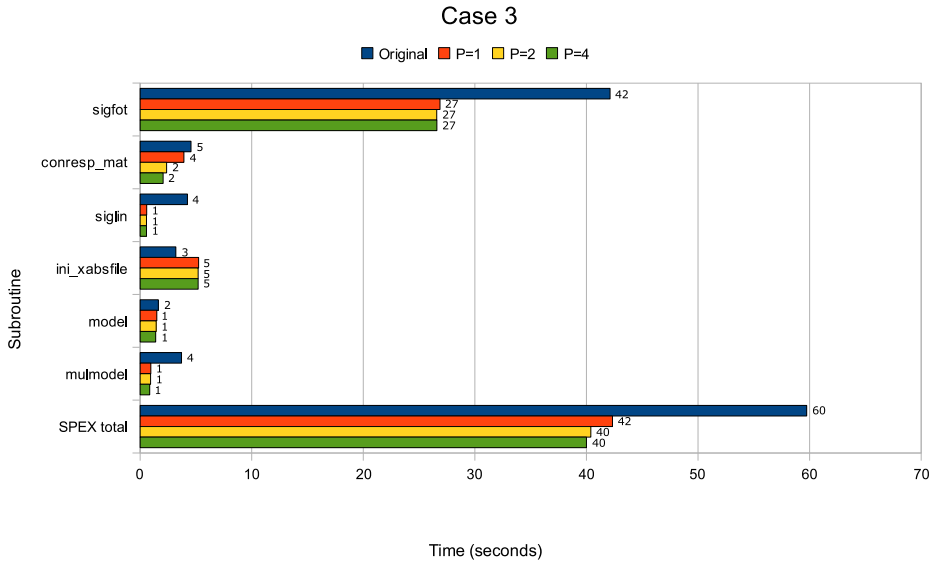

Figure 11.3: Results after optimization of the matrix-vector multiplication for use-case 3.

Chapter 12

Recommendations

This chapter presents some recommendations and possible improvements directed mainly at anyone wishing to improve on the current implementation with regard to the matrix-vector multiplication and the changes made in the previous chapters.

Sort algorithm

Currently components are ordered using a bubble sort algorithm. This is not a problem for matrices containing just a few components, but when matrices contain more components sorting may have too much impact on the total performance, especially for cases where the total execution time is relatively short.

Best solution to this would be if matrices where stored on disk in a sorted fashion. However, as matrices are often created by others this may not be a valid option.

Another option would be to use a more elegant and faster sort algorithm. There exist so many sort algorithms that discussing them here is not an option, however, the quicksort [Hoare, 1961] algorithm may be a good place to start.

Component splitting

Splitting large components is a relatively extensive process because a lot of memory has to be copied. This can possibly be improved by changing the way components are stored in memory such that everything is stored with pointers. However, using pointers for everything also means that all these pointers will have to be dereferenced sometime, which may cause a performance penalty.

Ideally, response components would be split in the file they are stored in on disk, such that they do not have to be split at runtime. Again, this is mainly relevant for cases where the total execution time is relatively short, and where response components are relatively large.

Improve response matrix storage on disk

Although better sorting and splitting algorithms may reduce the penalty introduced with these techniques, an optimal solution would be to pre-process the matrices stored on disk such that they are already sorted and split in smaller components if applicable. This would be useful because the same test will often be run multiple times, or a different test may use the same data.

Making sure that the response matrix is already of optimal form on disk would mean a significant performance increase, especially for use-cases 1 and 2.

Precalculating the multiplication vector

If possible, the multiplication vectors (*sie* and *wie*, the model spectrum) should be precalculated. Currently SPEX supports vectors which depend on certain properties of energy groups, therefore it is not possible to precalculate a single vector for the entire matrix. If it is scientifically reasonable to stop supporting this and precalculate a single vector for every matrix, performance could possibly be increased significantly.

If this can be done it should be combined with splitting components vertically. After all, overhead induced by calculating the multiplication vector would be gone while cache access is improved.

Part III

Calculating absorption due to photoionization

Chapter 13

Introduction

This part of the thesis focuses on optimization of the subroutines of SPEX that calculate absorption due to photoionization. This starts with a discussion of the physical background of the process, which will also explain why it is important in SPEX.

Chapter 14 gives an analysis of the current algorithm, and will inspect the data on which the algorithm operates. Chapter 15 will propose and implement possible optimizations to the algorithm. The resulting optimized algorithm will be parallelized in Chapter 16 and conclusions to this part of the thesis will be presented in Chapter 17.

13.1 Background

While the response matrix in the first part of this thesis is used to correct influences due to the instrument itself, this part is concerned with calculating the absorption due to photoionization in a medium between source and observer.

Imagine an ideal source and observer, with a certain gas in between. Fitting procedures in SPEX can be used to calculate the composition of this gas, because radiation with a specific wavelength will be absorbed by the gas. This causes Fraunhofer absorption lines, which was explained in Section 2.1, as well as continuum absorption structures.

The continuum absorption of radiation for an ionization stage i_i in a shell i_s of atom i_a can be measured experimentally or calculated theoretically. A lot of work has been done to create an empirical generic function - which depends on radiation energy E, with parameters depending on i_i, i_s and i_a - that calculates absorption ([Verner et al., 1995], [Verner and Yakovlev, 1995]). This work is the basis for the relevant routines used in SPEX.

The formula used in SPEX is derived from [Verner and Yakovlev, 1995], which states that the partial photoionization cross section σ for energy E can be described by:

$$\begin{aligned} \sigma(E) &= \sigma_0 F(E/E_0), \\ F(y) &= \left[(y-1)^2 + y_w^2\right] y^{-Q} \left(1 + \sqrt{y/y_a}\right)^{-P}, \end{aligned} \tag{13.1}$$

where σ_0, E_0, y_w, y_a and P are parameters depending on the current set of i_i, i_s and i_a, as will become clear in the next chapters, and $Q = 5.5 + l - 0.5P$, where

$l = 0$, 1, or 2 is the subshell orbital quantum number, which depends on i_s only. The result of Equation 13.1 is in Megabarn, which is a unit of area; 1 barn equals $10^{-28}m^2$. All parameters are real numbers.

Chapter 14

Current implementation and algorithm

The current implementation of the absorption calculation consists of two nested subroutines. The first, *sigfot*, is mainly concerned with initialization and flow control. After initialization of variables it contains three nested loops, which iterate on atoms, ions, and shells.

In the innermost loop *sigfot* calls subroutine *sig_shell*[1], which performs the actual calculation for every energy available in the model energy grid. Depending on i_i, i_s, and i_a the results may be multiplied by a correction factor.

Upon return to *sigfot*, the result vector $\vec{\sigma}$ is multiplied by the fitting parameter which describes how many ions of each kind are expected to be in the medium. Subsequently, the *transmission* is calculated using $\vec{\sigma}$ as

$$T(E) = e^{-\tau(E)}, \qquad \text{with } \tau(E) = \sum_{i_a} \sum_{i_i} \sum_{i_s} N_{i_a,i_i,i_s} \sigma_{i_a,i_i,i_s}(E).$$

In this, N_{i_a,i_i,i_s} is the fitting parameter. Furthermore, the current $\vec{\sigma}$ is added to a total for all combinations of i_a, i_i and i_i available in the used model, the maximum value of $\vec{\sigma}$ is stored, and the equivalent width EW, defined as

$$EW(i_a, i_i, i_s) = \int_0^\infty \left[1 - e^{-\tau_{i_a,i_i,i_s}(E)}\right] \text{ dE}$$

is calculated.

14.1 Algorithm

Since the exact algorithm used in SPEX is not extensively documented, the currently used algorithm has been reconstructed from the original source code. This results in Algorithm 5.

The following variables are used not only in this algorithm, but throughout the text in this part of the thesis as well:

explim A threshold, e^{-x} is considered zero if $x \geq$ explim. This variable has a fixed value of 34.

[1]Subroutine *sig_shell* is not profiled separately in SPEX. Its execution time is added to that of *sigfot*.

$n_{\mathbf{e}}$ Total number of energy bins in the model.

$n_{\mathbf{atoms}}$ Total atom count.

$n_{\mathbf{shells}}$ Maximum number of atomic shells.

δE_i Width of energy bin i.

E_i Center of energy bin i.

$A_{\mathbf{fot}}(i, \mathbf{ind})$ Photoionization parameters. The current atom/shell/ion combination is indicated by ind, the parameter by i.

$\mathbf{selected}(i_a)$ True if and only if atom i_a is selected in the model.

$\mathbf{column}(i)$ The density of ion i in the model.

$\mathbf{indion}(i_a, i_i)$ Table which contains indices for combinations of an atom i_a and ion i_i.

$\mathbf{index}(i_a, i_i, i_s)$ Table which contains indices for combinations of an atom i_a, ion i_i, and shell i_s.

Algorithm 5 starts with initialization of its result variables (lines 1-4). The first two, tau_{ph} and tra_{ph}, are vectors, the third, slabcont, is a two-dimensional array, and the fourth, elcol, is a scalar.

The algorithm continues by iterating on all possible atoms (line 5). The body of this loop is executed only if the current atom is selected in the model, which is checked on line 6. If the current atom is selected in the model, the algorithm will iterate on all possible ions for the current atom. The body of this second loop is only executed if the column density for the current ion is greater than 0 (lines 9-11). If so, the contribution to result variable elcol is calculated.

Continuing, the algorithm iterates on all possible atomic shells on line 16. Now, calculations have to be performed for every energy E_{i_e} on the used energy grid. However, calculations have to be performed only if $E_{i_e} \geq A_{\text{fot}}(1, \text{ind})$, which is represented on lines 19-21. The SPEX interpretation of Equation (13.1) can be seen on lines 22-23.

Note that line 24 of Algorithm 5 states that exceptions to the formula for $\sigma(i_e)$ must be handled. In practice, this means that if the current set of i_a, i_i, and i_s is equal to a certain predefined set, $\vec{\sigma}$ must be multiplied with a scalar. This is true for only two sets of i_a, i_i, and i_s, and will therefore not be included in the optimization process.

The calculated $\vec{\sigma}$ is used on lines 26-32 to populate and update the result variables.

14.1.1 Mathematical analysis

The function used in Algorithm 5 looks quite different from Equation (13.1). This section will show that both functions are actually the same.

The easiest way to show this is by starting with the function used in SPEX. Replacing some of the variable names used in Algorithm 5 with shorter ones for clarity gives:

$$\sigma(E) = \left((E - A_2)^2 + A_3\right) e^{A_4 + A_5 \log E + A_7 \log(A_6 + \frac{1}{\sqrt{E}})}. \tag{14.1}$$

Now, variables used in SPEX (A_i, E) should be expressed in terms of the variables used in Equation (13.1) $(y, E, y_w, y_a, \sigma_0, Q, P)$. Expressions for the variables used in SPEX are not documented, but in source code which was used to convert the

Algorithm 5 Calculating absorption due to photoionization - original algorithm

$\text{tau}_{\text{ph}}(i) \leftarrow 0 \quad \forall \quad i \in [0, n_{\text{e}})$
$\text{tra}_{\text{ph}}(i) \leftarrow 1 \quad \forall \quad i \in [0, n_{\text{e}})$
$\text{slabcont}(j, i) \leftarrow 0 \quad \forall \quad i \in [0, \max(index(a, i, s))], j \in [1, 3]$
$\text{elcol} \leftarrow 0$
for $i_a = 1$ to n_{atoms} **do**
  if selected(i_a) **then**
    for $i_i = 1$ to $i_a + 1$ **do**
      $\text{ion} \leftarrow \text{indion}(i_a, i_i)$
      if $\text{column(ion)} \leq 0$ **then**
        continue
      end if
      $\text{elcol} \leftarrow \text{elcol} + \text{column(ion)}(i_i + 1)$
      if $i_i > i_a$ **then**
        continue
      end if
      for $i_s = 1$ to n_{shells} **do**
        $\text{ind} \leftarrow \text{index}(i_s, i_i, i_a)$
        for $i_e = n_e$ to 1 step -1 **do**
          if $E_{i_e} < A_{\text{fot}}(1, \text{ind})$ **then**
            break
          end if
          $\sigma(i_e) \leftarrow \left((E_{i_e} - A_{\text{fot}}(2, \text{ind}))^2 + A_{\text{fot}}(3, \text{ind})\right)$
          $\cdot e^{A_{\text{fot}}(4,\text{ind}) + A_{\text{fot}}(5,\text{ind}) \log E_{i_e} + A_{\text{fot}}(7,\text{ind}) \log \left(A_{\text{fot}}(6,\text{ind}) + \frac{1}{\sqrt{E_{i_e}}}\right)}$
          handle exceptions to the formula for $\sigma(i_e)$
        end for
        $\text{tau}(i) \leftarrow \text{column(ion)} \cdot \sigma(i) \quad \forall \quad i \in [0, n_e)$
        $\text{tra}(i) \leftarrow e^{-\text{tau}(i)} \quad \forall \quad i \text{ such that } \text{tau}(i) < \text{explim}$
        $\text{tau}_{\text{ph}}(i) \leftarrow \text{tau}_{\text{ph}}(i) + \text{tau}(i) \quad \forall \quad i \in [0, n_e)$
        $\text{tra}_{\text{ph}}(i) \leftarrow \text{tra}_{\text{ph}}(i)\text{tra}(i) \quad \forall \quad i \in [0, n_e)$
        $\text{slabcont}(1, \text{ind}) \leftarrow A_{\text{fot}}(1, \text{ind})$
        $\text{slabcont}(2, \text{ind}) \leftarrow \max(\text{tau}(i) | i \in [0, n_e))$
        $\text{slabcont}(3, \text{ind}) \leftarrow \langle 1 - \text{tra}, \delta E \rangle$
      end for
    end for
  end if
end for

atomic data the following expressions where found:

$$A_2 = E_0 \tag{14.2}$$
$$A_3 = (E_0 y_w)^2 \tag{14.3}$$
$$A_4 = \log \sigma_0 + (3.5 + l - \frac{P}{2}) \log E_0 \tag{14.4}$$
$$A_5 = -5.5 - l \tag{14.5}$$
$$A_6 = \frac{1}{\sqrt{E_0 y_a}} \tag{14.6}$$
$$A_7 = -P. \tag{14.7}$$

Furthermore, recall from Section 13.1 that

$$Q = 5.5 + l - 0.5P, \tag{14.8}$$
$$y = \frac{E}{E_0}. \tag{14.9}$$

Now, combining Equation (14.1) with Equations (14.2) and (14.3) gives

$$\sigma(E) = \left((E - E_0)^2 + E_0^2 y_w^2\right) e^{A_4 + A_5 \log E + A_7 \log(A_6 + \frac{1}{\sqrt{E}})}.$$

Using Equation (14.9) and collecting E_0^2 results in

$$\sigma(E) = E_0^2 \left((y-1)^2 + y_w^2\right) e^{A_4 + A_5 \log E + A_7 \log(A_6 + \frac{1}{\sqrt{E}})}.$$

The first part of this equation already takes the desired form. Next, substitute Equation (14.4):

$$\sigma(E) = E_0^2 \left((y-1)^2 + y_w^2\right) e^{\log \sigma_0} e^{(3.5 + l - \frac{P}{2}) \log E_0} e^{A_5 \log E + A_7 \log(A_6 + \frac{1}{\sqrt{E}})},$$

which can also be written as

$$\sigma(E) = \sigma_0 E_0^2 \left((y-1)^2 + y_w^2\right) E_0^{3.5 + l - \frac{P}{2}} E^{A_5} \left(A_6 + \frac{1}{\sqrt{E}}\right)^{A_7}.$$

Substituting Equations (14.5) and (14.7) and expressing $E^{-5.5-l}$ in terms of y and E_0 gives

$$\sigma(E) = \sigma_0 E_0^2 \left((y-1)^2 + y_w^2\right) E_0^{3.5 + l - \frac{P}{2}} y^{-5.5-l} E_0^{-5.5-l} \left(A_6 + \frac{1}{\sqrt{E}}\right)^{-P}.$$

Now, by grouping all powers of E_0, this can be reduced to

$$\sigma(E) = \sigma_0 \left((y-1)^2 + y_w^2\right) y^{-5.5-l} E_0^{-\frac{P}{2}} \left(A_6 + \frac{1}{\sqrt{E}}\right)^{-P}.$$

This can be further reduced by multiplying with $y^Q y^{-Q} = 1$ and applying Equation (14.8), giving

$$\sigma(E) = \sigma_0 \left((y-1)^2 + y_w^2\right) y^{-Q} y^{5.5 + l - \frac{P}{2}} y^{-5.5-l} E_0^{-\frac{P}{2}} \left(A_6 + \frac{1}{\sqrt{E}}\right)^{-P},$$
$$\sigma(E) = \sigma_0 \left((y-1)^2 + y_w^2\right) y^{-Q} y^{-\frac{P}{2}} E_0^{-\frac{P}{2}} \left(A_6 + \frac{1}{\sqrt{E}}\right)^{-P}.$$

Using Equation (14.9), E_0 can be eliminated:

$$\sigma(E) = \sigma_0 \left((y-1)^2 + y_w^2\right) y^{-Q} E^{-\frac{P}{2}} E_0^{\frac{P}{2}} E_0^{-\frac{P}{2}} \left(A_6 + \frac{1}{\sqrt{E}}\right)^{-P},$$
$$\sigma(E) = \sigma_0 \left((y-1)^2 + y_w^2\right) y^{-Q} E^{-\frac{P}{2}} \left(A_6 + \frac{1}{\sqrt{E}}\right)^{-P}.$$

The only variable now left from the original SPEX implementation is A_6. Substituting Equation (14.6) gives

$$\sigma(E) = \sigma_0 \left((y-1)^2 + y_w^2\right) y^{-Q} E^{-\frac{P}{2}} \left(\frac{1}{\sqrt{E_0 y_a}} + \frac{1}{\sqrt{E}}\right)^{-P},$$
$$\sigma(E) = \sigma_0 \left((y-1)^2 + y_w^2\right) y^{-Q} \left(E^{\frac{1}{2}} \left(\frac{1}{\sqrt{E_0 y_a}} + \frac{1}{\sqrt{E}}\right)\right)^{-P},$$
$$\sigma(E) = \sigma_0 \left((y-1)^2 + y_w^2\right) y^{-Q} \left(1 + \sqrt{\frac{y}{y_a}}\right)^{-P},$$

which is the same as Equation (13.1), proving equality with the function used in the current implementation of SPEX.

Note that, although it is not shown in the analysis, SPEX does not use the Mb unit directly. Therefore, certain variables have been scaled appropriately.

This translation was originally done to prevent rounding errors and for performance reasons. After all, note that the SPEX version only uses natural logarithms and powers of e, while Equation 13.1 uses arbitrary exponents.

14.2 Data inspection

Before starting optimization it is important to have a general idea of the values of the used data. After all, it may be possible to exploit certain properties of this data. The main variables of interest are the different atomic properties (*afot(i,ind)*) and the energy grid (E_i). Furthermore, it may be interesting to take a closer look at the results. All data will be from use-case 3.

A sample from the tables with atomic data used in SPEX can be seen in Table 14.1. Note that this table gives a good impression of the data; if there are notable exceptions these will be mentioned when relevant. Two things are immediately obvious from the data in Table 14.1: first, A_3 is often zero, and second, it appears that A_5 only takes on a few different values. There is no apparent structure in the rest of the variables. In total, there are 2167 possible indices for *afot(i,:)*.

Figure 14.1 shows the energy grid for use-case 3. Apparently, the gridpoints are distributed fairly logarithmically, with the exception of the first and last few points. These first and last points are either very small or very large compared to the other gridpoints. The energy grid contains a total of 36792 points. Note that the energy grid may vary during the fitting procedure to accommodate velocity of the source. That is, if there is a large relative speed between source and observer, there will be redshift or blueshift, which are results of the Doppler effect. In SPEX this is accounted for by having the fitting procedure apply small shifts to the energy grid.

Not only the input data is of interest; it may be interesting to take a look at the output data as well, at least to understand what is going on. Figure 14.2 shows $\sigma(E)$ for a number of ions in use-case 3. Note that apparently $\sigma(E)$ only has meaningful values for energies above a certain threshold ($A_{\text{fot}}(1, \text{ind})$ in Algorithm 5), which is caused by the fact that the algorithm starts with the highest energy and decreases, until a threshold is reached. Remaining entries for $\sigma(E)$ are zero, because of the minimum energy needed to photoionize an atom, called the photoelectric effect [Einstein, 1905].

A_1	A_2	A_3	A_4	A_5	A_6	A_7
0.0135980	0.0010091	0	7.2470322	-5.5	6.5030589	-2.3877800
0.0245870	0.0020240	0	19.2450333	-5.5	7.1560950	-6.2179999
0.0544160	0.0040381	0	9.0587015	-5.5	3.2508118	-2.3877800
0.3060000	0.0911300	0	11.2309914	-5.5	0.3379327	-1.3380001
0.0309000	0.0029910	0	43.5595398	-5.5	10.4103222	-14.8000002
0.0244000	0.0109400	0.0000333	8.0543394	-6.5	1.6622968	-4.1500001
0.3250000	0.0837000	0	11.3125229	-5.5	0.3998968	-1.4420000
0.0479000	0.0039420	0	19.9722786	-5.5	4.1137781	-7.4889998
0.3440000	0.0841200	0	11.3145437	-5.5	0.3992180	-1.4280000
0.0645000	0.0078430	0	12.6350021	-5.5	2.1233382	-4.7540002

Table 14.1: Sample from the atomic data used in SPEX.

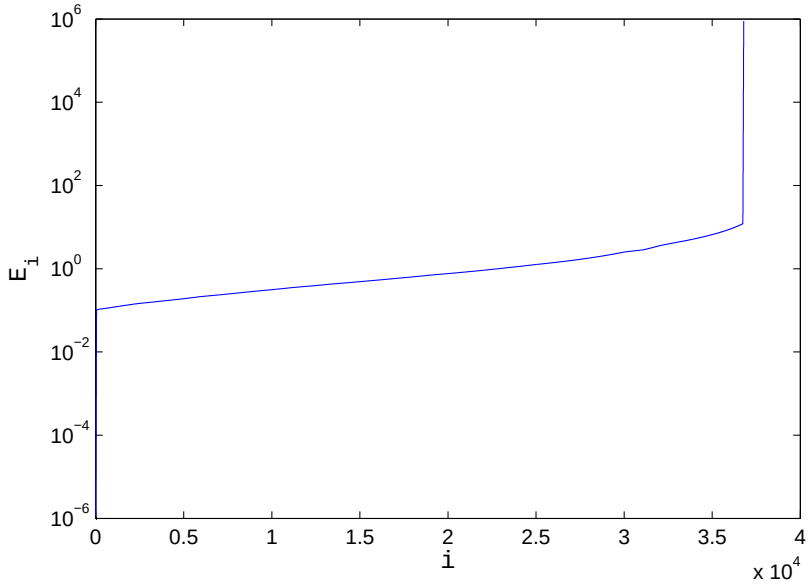

Figure 14.1: Energy grid E_i for use-case 3.

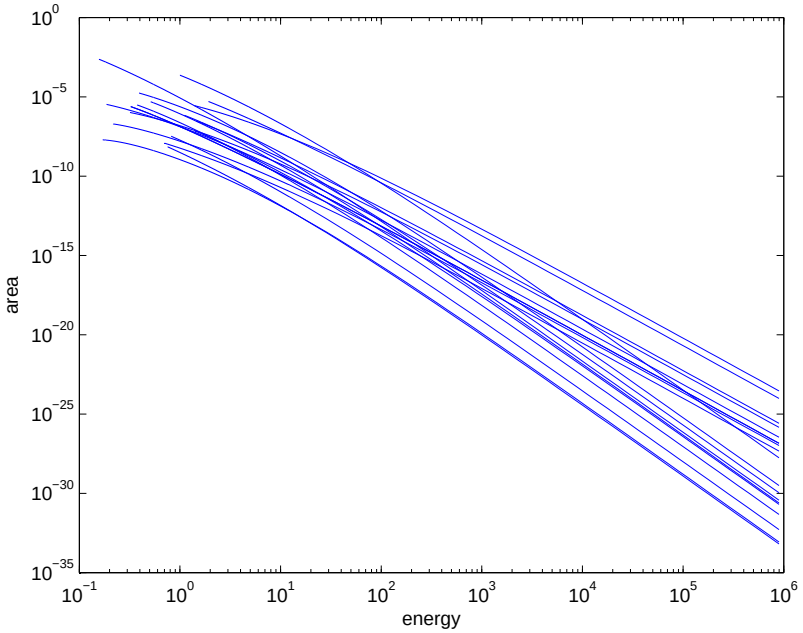

Figure 14.2: $\sigma(E)$ for a random collection of ions from use-case 3.

Chapter 15

Sequential optimization

This chapter describes an optimized sequential algorithm and compares its performance with the original algorithm. Because of time constraints, only use-case 3 is used for performance tests. The testing environment is unaltered, i.e. used hardware and software is the same as in Part II of this thesis.

15.1 Reference measurements

Since the optimization process for *sigfot* starts with the end result of Part II of this thesis, reference measurements at the beginning of this part are the same as the ones used in the conclusion of Part II. However, these measurements only provided total execution times for the individual subroutines, while interest is now mainly in the execution time per call to *sigfot*.

This results in the reference measurements being a single value. Running use-case 3 three times results in 94 calls to *sigfot* for every run, taking an average of 26.604 seconds total per run. Thus, the average execution time for a single call to *sigfot* is 283.021 milliseconds.

15.2 Optimized algorithm

15.2.1 Possible optimizations

Before presenting an optimized algorithm, this section will discuss possible optimizations.

Calculation of $\vec{\sigma}$

Obviously, one of the most time consuming parts of the current algorithm is the calculation of $\sigma(i_e)$ (recall Equation 14.1). This function is in itself already efficient. However, it may be possible to calculate some of the terms separately.

Parts of the function that qualify for this "precalculation" should depend on either i_e or *ind*, but not both. After all, if a term depends on both i_e and *ind*, precalculating all possibilities would require $36792 \cdot 2167$ floating point values, which equals roughly 300 megabyte. Not only is this a lot of memory, it would also be questionable whether doing precalculations like this would result in a faster algorithm. After all, fetching data from memory is relatively expensive, so it may be faster to have less memory access and more calculations.

Values in *afot* are constants. However, none of the terms in Equation 14.1 that would qualify for precalculation depend solely on *afot*. There are however two parts which depend only on i_e that could be candidate for precalculation, being $\log E_{i_e}$ and $\frac{1}{\sqrt{E_{i_e}}}$. These terms will not change during execution of *sigfot*, and can thus be calculated outside of the loops on ions, atoms and shells.

Calculation of *slabcont(i,j)*

Recall from Algorithm 5 that the variable *slabcont(i,j)* contains properties i of the calculated absorption for every ion j. These properties are relatively expensive to calculate, especially calculation of the inner product.

After some research it became clear that the values in *slabcont* are not used in the fitting procedure. They are only used after the fitting procedure, if results are printed to screen or file. This means effectively that all calculations that only influence *slabcont* are performed without reason, except at the end of the fitting procedure.

Luckily, SPEX already contains two global flags, *fastmode* and *ascmode*, which can be used to determine whether *slabcont* has to be calculated.

Calculation of $\mathrm{tau_{ph}}$ and $\mathrm{tra_{ph}}$

The variables $\mathrm{tau_{ph}}$ and $\mathrm{tra_{ph}}$ are used in the fitting procedure, i.e. they are used to construct the model spectrum. Algorithm 6 repeats the relevant part of Algorithm 5 for clarity.

Note that the most time-consuming part of Algorithm 6 will be calculation of the exponent of *tau*. After all, all other operators are either assignments or regular additions and multiplications. Also note that Algorithm 6 is performed for every ion available in the model. This makes calculation of $\mathrm{tra_{ph}}$ actually a product of powers of e. Because

$$e^a e^b = e^{a+b},$$

$\mathrm{tra_{ph}}$ can also be calculated as

$$\mathrm{tra_{ph}} = e^{-\mathrm{tau_{ph}}}.$$

After all, $\mathrm{tau_{ph}}$ already is the desired summation. Doing this makes it possible to take the exponentiation out of the iteration on atoms, shells, and ions. Thus, instead of calculating the exponent for every ion in the model, the exponent has to be calculated just once.

Furthermore, combining this with the fact that *slabcont* now has to be calculated only once allows for removal of some temporary variables, such as *tau* and *tra*.

Algorithm 6 Calculating $\mathrm{tau_{ph}}$ and $\mathrm{tra_{ph}}$ - original algorithm

1: $\mathrm{tau}(i) \leftarrow \mathrm{column(ion)} \cdot \sigma(i) \quad \forall \quad i \in [0, n_e)$
2: $\mathrm{tra}(i) \leftarrow e^{-\mathrm{tau}(i)} \quad \forall \quad i$ such that $\mathrm{tau}(i) < \mathrm{explim}$
3: $\mathrm{tau_{ph}}(i) \leftarrow \mathrm{tau_{ph}}(i) + \mathrm{tau}(i) \quad \forall \quad i \in [0, n_e)$
4: $\mathrm{tra_{ph}}(i) \leftarrow \mathrm{tra_{ph}}(i)\mathrm{tra}(i) \quad \forall \quad i \in [0, n_e)$

Using properties of *afot(i,j)*

Note that it was determined that *afot(3,j)* was often zero and that *afot(5,j)* only took on a few different values (Recall Table 14.1). This would suggest using different forms of the formula that calculates $\vec{\sigma}$. For instance, if *afot(3,j)* = 0, a version could be used that does not include the addition of *afot(3,j)*.

Unfortunately, creating special cases for neither *afot(3,j)* nor *afot(5,j)* will reduce the amount of logarithms or exponentiations in the used formula. Compared to the computation time required for a logarithm or exponentiation the removal of one addition will not contribute much. Furthermore, making these special cases will create a lot more code, making the program less readable and maintainable.

Because of these reasons, no special cases for the formula of $\vec{\sigma}$ depending on values in *afot(i,j)* will be created.

15.2.2 Algorithm

The changes proposed above lead to Algorithm 7. Note that this algorithm is slightly more complex than the original algorithm (Algorithm 5), but added complexity is mainly in the form of conditional statements which make sure only the absolute necessary is calculated.

Since the algorithm directly implements the suggestions from the previous sections and is largely the same as the original algorithm it will not be discussed in detail.

15.3 Test results

Recall that the original implementation resulted in a computation time of 283.021 milliseconds for every call to *sigfot*. This section shows how much is gained by the improvements implemented in Section 15.2.

Test results can be seen in Figure 15.1 and Table 15.1. The test results show a good improvement with a speedup of 2.43.

Algorithm 7 Calculating absorption due to photoionization - optimized algorithm

$\text{tau}_{\text{ph}}(i) \leftarrow 0 \qquad \forall \quad i \in [0, n_{\text{e}})$
if !fastmode and ascmode **then**
  $\text{slabcont}(j, i) \leftarrow 0 \qquad \forall \quad i \in [0, \max(index(a, i, s))), j \in [1, 3]$
end if
$\text{elcol} \leftarrow 0$
$\text{invsqrtE}_i \leftarrow \frac{1}{\sqrt{E_i}} \qquad \forall \quad i \in [0, n_{\text{e}})$
$\text{logE}_i \leftarrow \log E_i \qquad \forall \quad i \in [0, n_{\text{e}})$
for $i_a = 1$ to n_{atoms} **do**
  if $\text{selected}(i_a)$ **then**
    for $i_i = 1$ to $i_a + 1$ **do**
      $\text{ion} \leftarrow \text{indion}(i_a, i_i)$
      if $\text{column(ion)} \leq 0$ **then**
        continue
      end if
      $\text{elcol} \leftarrow \text{elcol} + \text{column(ion)}(i_i + 1)$
      if $i_i > i_a$ **then**
        continue
      end if
      for $i_s = 1$ to n_{shells} **do**
        $\text{ind} \leftarrow \text{index}(i_s, i_i, i_a)$
        $\sigma(i_e) \leftarrow 0 \qquad \forall \quad i \in [0, n_{\text{e}})$
        for $i_e = n_e$ to 1 step -1 **do**
          if $E_{i_e} < A_{\text{fot}}(1, \text{ind})$ **then**
            break
          end if
          $\sigma(i_e) \leftarrow \text{column(ion)} \left(\left(E_{i_e} - A_{\text{fot}}(2, \text{ind})\right)^2 + A_{\text{fot}}(3, \text{ind}) \right)$
$\cdot e^{A_{\text{fot}}(4,\text{ind}) + A_{\text{fot}}(5,\text{ind}) \text{logE}_{i_e} + A_{\text{fot}}(7,\text{ind}) \log\left(A_{\text{fot}}(6,\text{ind}) + \text{invsqrtE}_{i_e}\right)}$
          handle exceptions to the formula for $\sigma(i_e)$
        end for
        $\text{tau}_{\text{ph}}(i) \leftarrow \text{tau}_{\text{ph}}(i) + \sigma(i) \qquad \forall \quad i \in [0, n_e)$
        if !fastmode and ascmode **then**
          $\text{slabcont}(1, \text{ind}) \leftarrow A_{\text{fot}}(1, \text{ind})$
          $\text{slabcont}(2, \text{ind}) \leftarrow \max(\text{tau}(i) | i \in [0, n_e))$
          $\text{slabcont}(3, \text{ind}) \leftarrow \langle 1 - \text{tra}, \delta E \rangle$
        end if
      end for
    end for
  end if
end for
$\text{tra}_{\text{ph}}(i) \leftarrow e^{-\text{tau}_{\text{ph}}} \qquad \forall \quad i \in [0, n_e)$

Version	Time (ms)	Speedup
Original	283.021	1.00
Optimized	116.662	2.43

Table 15.1: Comparison of the execution time and speedup of *sigfot* for the original and the optimized version.

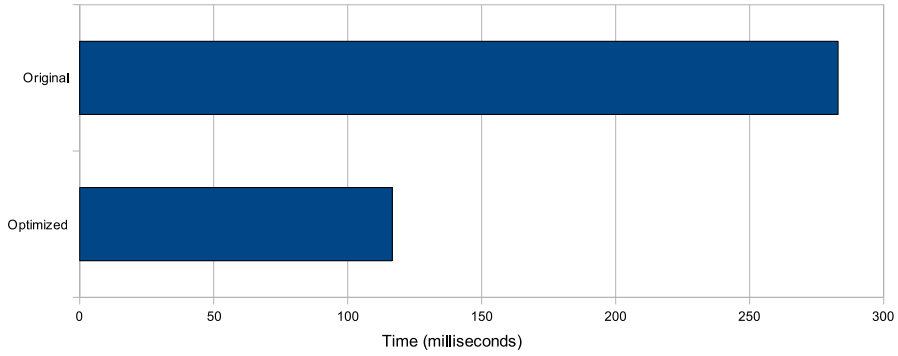

Figure 15.1: Comparison of the execution time of *sigfot* for the original and the optimized version.

Chapter 16

Parallelization

This chapter will extend optimization of *sigfot* with a simple parallel version.

16.1 Parallel algorithm and implementation

The optimized algorithm (Algorithm 7) contains a number of nested loops. At each level of these nested loops parallelization is theoretically possible. However, to keep overhead by parallelization to a minimum, it is generally best to parallelize the outer loop.

If the outer loop (the one that iterates on all atoms) is chosen for parallelization, there is a small problem with load balancing. If one would use the default openMP scheduling type, which means distributing loop iterations in a cyclic fashion (recall Section 9.3), load balance will be optimal if every iteration contains the same amount of work. However, mainly because some atoms may not be selected in the model, this will generally not be the case.

One way to solve this would be to manually create a distribution of iterations for every call to *sigfot*, tailored to create near-ideal load balance. An easier and more versatile option however may be to use the *dynamic* scheduling clause of openMP. This clause tells the compiler that it must dynamically assign loop iterations to threads, theoretically creating near-ideal load balance at the cost of some overhead in iteration distribution.

Another problem, similar to one of the problems with parallelization in Part II of this thesis (Section 8.3), is the fact that some of the result variables of *sigfot* are summations of result variables from the inner loop. As before, this results in a *race condition*, and can be solved by use of the openMP *reduction* clause.

Using these techniques for parallelization makes sure the algorithm can stay practically identical to Algorithm 7, with the exception of added openMP clauses. Therefore, the parallel algorithm will not be discussed in detail. However, it may be interesting to take a closer look at the actual source code.

The relevant part of the source code for *sigfot* can be seen in Listing 16.1. Note that variables *inv_sqrt_ef* and *log_ef* have a special status, which results in these variables keeping their values across subroutine calls. The source code translates almost directly back to Algorithm 7, with the obvious exception of the parallelization constructs.

Parallelization of *sigfot* using openMP involves only a few lines of code. Two of these (lines 13 and 14 of Listing 16.1) start the parallel processing, another two (lines 47 and 48) end the parallel processing, and two local variables are used (lines 11, 12, 49, and 50) to prevent a possible race condition.

The first openMP directive in Listing 16.1 is the *omp parallel* directive. This directive indicates that from this line on a team of threads will execute all code, until the *omp end parallel* directive is encountered. Furthermore there are the *default(shared)* and *private(...)* clauses, which tell the compiler which variables should be the same for every thread and which variables should be private to a thread. The only variables made private in this case are the ones that are assigned a value in the parallel section, with exception of the variables that appear in the *reduction* clause later on.

The second openMP directive makes sure the work is actually distributed among threads. This is done by the *omp do* directive, which tells the compiler that the iterations of the following do-loop must be distributed among all available threads. As discussed before in this section, iterations will be distributed dynamically, indicated in the source code with the *schedule(dynamic)* clause. Finally, this openMP directive contains the *reduction* clause, telling the compiler which operations will be used to update which variables by multiple threads simultaneously.

16.2 Test results

Test results of the parallel version compared to the optimized sequential version can be seen in Figure 16.1 and Table 16.1.

Note that the parallel version using just one thread is slightly slower than the sequential version (approximately 2%). This is as expected, as there obviously is some overhead in the parallel version. This overhead will be caused by two factors. First, there will be some overhead due to the openMP instructions, even though no parallelization is performed when running with just one thread. Second, there will be some overhead because additional local variables are used to prevent a race condition.

The achieved parallel speedup can be considered pretty good, especially for an implementation created so easily.

```
1  tauph = 0.
2  if (eqwidth) slabcont = 0.
3  elcol = 0.
4
5  allocate(inv_sqrt_ef(ne))
6  allocate(log_ef(ne))
7  inv_sqrt_ef = 1. / sqrt(ef)
8  log_ef = log(ef)
9
10 !-- loop over all ions and subshells
11 tauph_local = 0.
12 elcol_local = 0.
13 !$OMP PARALLEL DEFAULT(shared), PRIVATE(iz,jz,is,ie,ion,ind,sig)
14 !$OMP DO SCHEDULE(dynamic) REDUCTION(+:tauph_local,elcol_local)
15 do iz = 1,nz
16     if (ielsel(iz)) then
17         do jz = 1,iz+1
18             ion = indion(jz,iz)
19             if (column(ion).le.0.) cycle
20             elcol_local = elcol_local + column(ion) * float(jz-1)
21             if (jz.gt.iz)
22             do is = 1,ns
23                 ind = index(is,jz,iz)
24                 if (ind.le.0) cycle
25                 sig = 0.
26                 do ie = ne,1,-1
27                     if (ef(ie).lt.afot(1,ind)) exit
28                     sig(ie) = column(ion)*((ef(ie)-afot(2,ind))**2 &
29                     +afot(3,ind))*exp(afot(4,ind)+afot(5,ind) &
30                     *log_ef(ie)+afot(7,ind)*log(afot(6,ind) &
31                     +inv_sqrt_ef(ie)))
32                 enddo
33
34                 ! handle exceptions to sig(ie)
35                 [...]
36
37                 tauph_local = tauph_local + sig
38                 if (eqwidth) then
39                     slabcont(1,ind) = afot(1,ind)
40                     slabcont(2,ind) = maxval(sig)
41                     slabcont(3,ind) = dot_product(1.-exp(-sig),de)
42                 endif
43             enddo
44         enddo
45     endif
46 enddo
47 !$OMP END DO
48 !$OMP END PARALLEL
49 tauph = tauph_local
50 elcol = elcol_local
51 traph = exp(-tauph)
52
53 deallocate(inv_sqrt_ef(ne))
54 deallocate(log_ef(ne))
```

Listing 16.1: Parallel version of the optimized algorithm for calculating absorption due to photoionization. Note that *[...]* indicates some code has been removed for clarity.

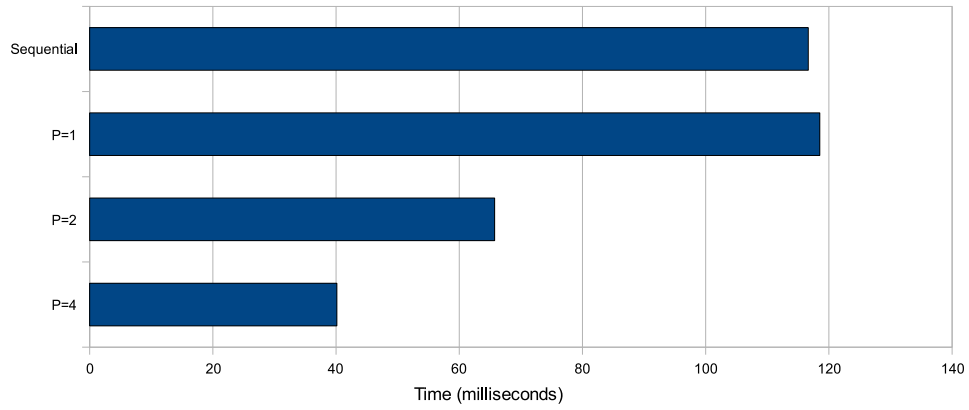

Figure 16.1: Test results of the parallel version using 1, 2, or 4 processors compared to the sequential version.

Version	Time (ms)	Parallel speedup
Sequential	116.660	1.00
Parallel (P=1)	118.527	0.98
Parallel (P=2)	65.741	1.77
Parallel (P=4)	40.135	2.91

Table 16.1: Parallel speedup when using 1, 2, or 4 processors.

Chapter 17

Conclusions

By carefully looking at the algorithm and source code it was possible to take some expensive operations out of inner loops, thereby reducing execution time. Furthermore it was discovered that results from certain calculations were used only very selectively, while they were always calculated. By making these calculations conditional, execution time was reduced further. Finally, it turned out that parts of the formula for $\vec{s}$ did not depend on the loop iterations, and thus could be calculated outside of the loops.

These sequential optimizations resulted in a speedup of 2.43 for use-case 3.

This optimized algorithm was transformed to a parallel algorithm using openMP. This introduced some overhead, causing the parallel version using only one thread to be approximately 2% slower than the sequential version. However, when using two or four threads parallel speedups of respectively 1.77 and 2.91 were achieved, in addition to the speedup of 2.43 that was already achieved with the sequential version.

This results in a total speedup of 4.31 when using two threads or 7.05 when using a machine capable of handling four threads.

Part IV

Final test results and conclusions

Chapter 18

Test results

Final test results for use-cases 1, 2, and 3 can be seen in Figures 18.1, 18.2, and 18.3 respectively. These results include work done in Parts II and III.

As these results show, use-case 1 gained the least of all from the optimizations implemented in this thesis. This is as expected; after all, the most time consuming routines used for use-case 1 were not selected for optimization. Still, when using only one processor use-case 1 executes in approximately 89% of the time needed originally. When using four processors this is further reduced to 86%.

Speedup for use-case 2 is a lot better. When using 1 processor execution time is 51% of the original, when using four processors this is only 37%. This is mainly because a lot of time was spent in the subroutines that were optimized, but also because side-effects of these optimizations resulted in the fitting procedure converging faster.

Use-case 3 gains most with the implemented optimizations; execution time is 43% of the original when using one processor and 28% when using four processors. This is mainly because the routine that was improved most also was the routine that consumed the most time for use-case 3.

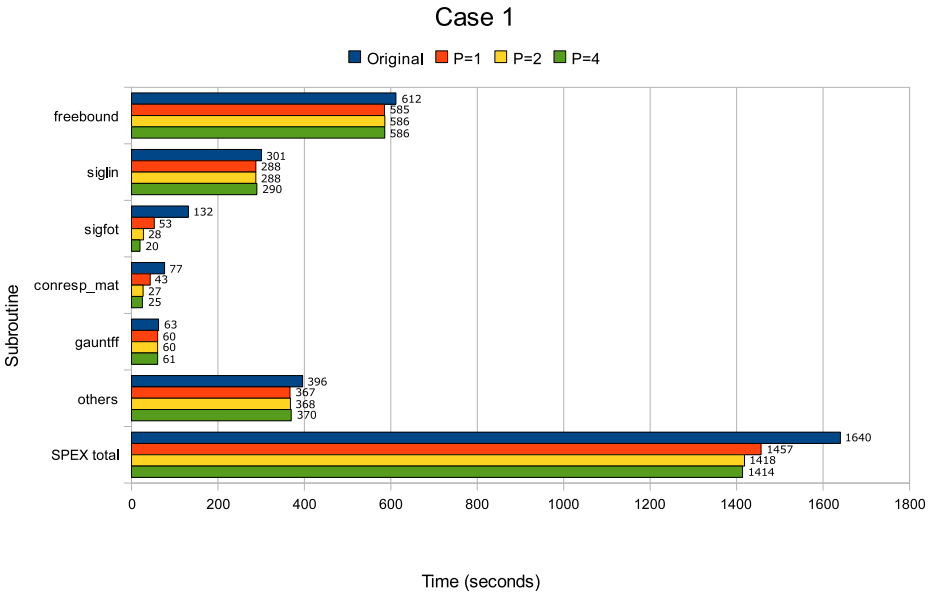

Figure 18.1: Final test results for use-case 1.

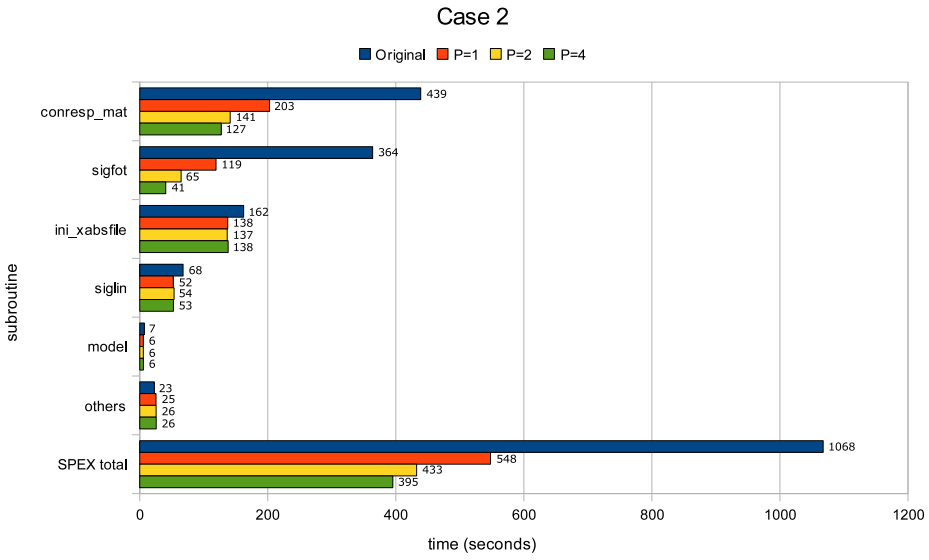

Figure 18.2: Final test results for use-case 2.

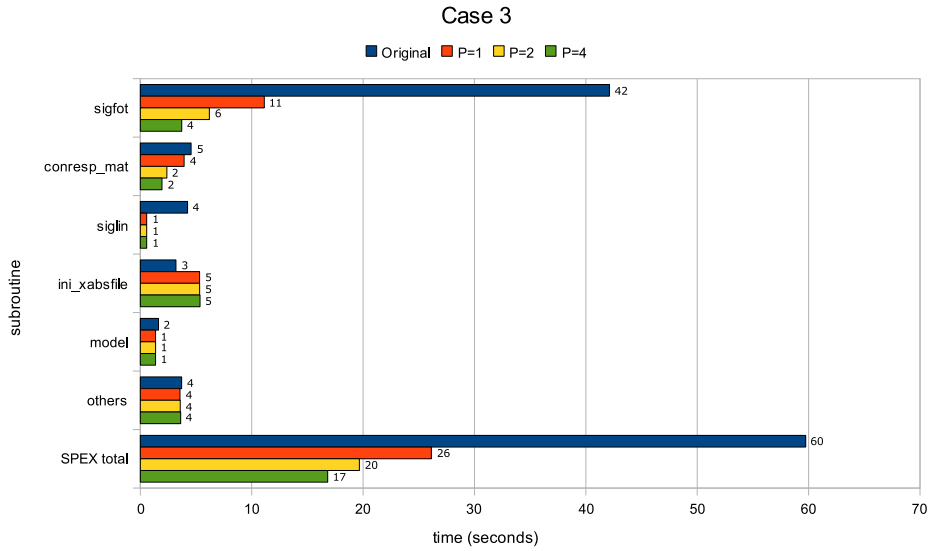

Figure 18.3: Final test results for use-case 3.

Chapter 19

Conclusions

The main goal of this thesis was to improve the speed of a program used for spectral analysis for X-ray astronomy called SPEX. To do this, measurements were done to show how much time was spent in individual subroutines using three different use-cases. These three use-cases all resulted in different measurements. However, these measurements were used to select two subroutines that were used in all three use-cases and contributed significantly to the total execution time of the program.

The first of these subroutines essentially carried out a matrix-vector multiplication, which is part of a fitting procedure in SPEX. While optimizing this subroutine, it was discovered that if precision of this multiplication was increased, the encompassing fitting routine converged faster, thus making all use-cases faster. The matrix-vector multiplication was parallelized after sequential optimization. The main problem with parallelization was making sure that calculations were spread evenly over different processors, i.e. achieve good load-balance. This was done by slightly altering the way the matrix was stored in memory and by manually distributing parts of the matrix to different processors.

The second subroutine that was optimized calculated absorption of X-rays due to photoionization. This subroutine was optimized mainly by either postponing calculations until they were really necessary or by calcuting parts in advance. This was easier to parallelize; a simple parallel version showed a speedup of 1.77 when using 2 processors and 2.91 when using 4 processors.

For all parallelizations openMP was used. OpenMP is mainly suited for shared memory systems, and provides easy incremental parallelization. Furthermore, even compilers that do not support openMP can compile a program using openMP, although this will not result in a parallel program. OpenMP turned out to be the right choice and has been shown to result in very readable and maintainable program code, while at the same time resulting in good performance.

Improvements made to these two parts of SPEX in this thesis will probably also apply to other parts of SPEX. Using the work from this thesis as a guideline chances are that performance can be increased further, especially for use-cases that focus on other parts than the two optimized routines.

Bibliography

[Amdahl, 1967] Amdahl, G. (1967). Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, number 30, pages 483–385.

[Astrachan, 2003] Astrachan, O. (2003). Bubble Sort: An Archeological Algorithmic Analysis. *SIGSCE*, pages 19–23.

[Bland-Hawthorn and Cecil, 1997] Bland-Hawthorn, J. and Cecil, G. (1997). Classical Spectroscopy. *Atomic, Molecular and Optical Physics: Atoms and Molecules*, 29B.

[Bohr, 1913] Bohr, N. (1913). On the constitution of atoms and molecules, parts i, ii, and iii. *Philosophical Magazine*, 26.

[Einstein, 1905] Einstein, A. (1905). On a heuristic viewpoint concerning the production and transformation of light. *Annalen der Physik*, 17:132–148.

[Hoare, 1961] Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Communications of the ACM*.

[Intel Corporation, 2009a] Intel Corporation (2009a). *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1*, chapter 2.3, page 62.

[Intel Corporation, 2009b] Intel Corporation (2009b). *Intel Fortran Compiler User and Reference Guides*.

[ISO/IEC, 2009] ISO/IEC (2009). JTC1/SC22/WG5/N1776: Information technology — Programming languages — Fortran. For review, International Organization for Standardization, Geneva, Switzerland.

[Kaastra et al., 1996] Kaastra, J., Mewe, R., and Nieuwenhuijzen, H. (1996). SPEX: a new code for spectral analysis of X-ray and UV spectra. *11th Colloquiem on UV and X-ray spectroscopy of astrophysical and laboratory plasmas*, pages 411–414.

[Kaastra et al., 2007] Kaastra, J., Mewe, R., and Raassen, T. (2007). SPEX User's Manual. Available at http://www.sron.nl/spex/.

[Kirchhoff, 1859] Kirchhoff, G. R. (1859). Über den Zusammenhang zwischen Emission und Absorption von Licht und Wärme. *Monatsberichte der Akademie der Wissenschaften zu Berlin*, pages 783–787.

[Kirchhoff and Bunsen, 1863] Kirchhoff, G. R. and Bunsen, R. (1863). Untersuchungen über das Sonnenspektrum und die Spektren der chemischen Elemente. *Abh. kgl. Akad. Wiss*, page 1861.

[Liedahl et al., 1994] Liedahl, D., Osterheld, A., Mewe, R., and Kaastra, J. (1994). New calculations of Fe spectra in high-temperature plasmas. In *The advanced satellite for cosmology and astrophysics symposium*, Tokyo, Japan.

[Mewe, 1974] Mewe, R. (1974). *Solar Physics*, 22:459–491.

[Mewe et al., 1985] Mewe, R., Gronenschild, E. H. B. M., and van den Oord, G. H. J. (1985). *Astronomy and Astrophysics supplement series*, 62:197–254.

[Mewe et al., 1986] Mewe, R., Lemen, R., and van den Oord, G. H. J. (1986). *Astronomy and Astrophysics supplement series*, 65:511–536.

[MPI Forum, 2008] MPI Forum (2008). MPI: A Message-Passing Interface Standard Version 2.1.

[Numrich and Reid, 1998] Numrich, R. W. and Reid, J. (1998). Co-Array Fortran for parallel programming. *ACM Fortran forum*, 17(2):1–31.

[OpenMP ARB, 1997] OpenMP ARB (1997). OpenMP Fortran Application Program Interface. Available at http://www.openmp.org/.

[OpenMP ARB, 1998] OpenMP ARB (1998). OpenMP C and C++ Application Program Interface. Available at http://www.openmp.org/.

[OpenMP ARB, 2008] OpenMP ARB (2008). OpenMP Application Program Interface. Available at http://www.openmp.org/.

[Pearson, 1985] Pearson, T. (1985). *PGPLOT Graphic subroutine library*. Pasadena: Institute of Technology.

[Raymond and Smith, 1977] Raymond, J. and Smith, B. (1977). *Astrophysical Journal supplement series*, 35:419–439.

[Seyfert, 1943] Seyfert, C. K. (1943). Nuclear Emission in Spiral Nebulae. *The Astrophysical Journal*, 97:28–40.

[Verner et al., 1995] Verner, D. A., Ferland, G. J., Torista, K. T., and Yakovlev, D. G. (1995). Atomic data for astrophysics. II. New analytic fits for photoionization cross section of atoms and ions. *The Astrophysical Journal*, 465:487–498.

[Verner and Yakovlev, 1995] Verner, D. A. and Yakovlev, D. G. (1995). Analytic fits for partial photoionization cross sections. *Astronomy and Astrophysics supplement series*, 109:125–133.

[von Fraunhofer, 1815] von Fraunhofer, J. (1814-1815). Bestimmung des Brechungs- und Farbenzerstreutuungs-Vermögens verschiedener Glassarten, in Bezug auf die Vervollkommung achromatischer Fernröhre. *Denkschriften der Königlichen Akademie der Wissenschaften zu München*, V:193–226.

[Wells et al., 1981] Wells, D. C., Greisen, E. W., and Harten, R. H. (1981). FITS - a Flexible Image Transport System. *Astronomy and Astrophysics supplement series*, 44:363–370.

[Wollaston, 1802] Wollaston, W. H. (1802). A method of examining refractive and dispersive powers, by prismatic reflection. *Philosophical Transactions of the Royal Society of London*, 92:365–380.