Profiling, Optimization and Parallelization of a Seismic Inversion Code

Bent Ove Stinessen

Master of Science in Computer Science

Submission date:	July 2011
Supervisor:	Anne Cathrine Elster, IDI
Co-supervisor:	Alf Birger Rustad, Statoil

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Descriptions

This master thesis looks at HPC optimization and parallelization techniques and applies them to a seismic inversion code named *CRAVA* (Conditioning Reservoir variables to Amplitude Versus Angle data). The project starts by investigating sources of performance impacts and suitable tools for profiling and analyzing applications such as CRAVA. Sections of CRAVA are then classified according to the seven dwarfs taxonomy (UC Berkeley), and optimized and parallelized using libraries (e.g. FFTW) and/or OpenMPI/OpenMP. This project is in collaboration with Statoil.

Assignment given: 17. January 2011
Supervisor: Anne Cathrine Elster, IDI

Abstract

Modern chip multi-processors offer increased computing power through hardware parallelism. However, for applications to exploit this parallelism, they have to be either designed for or adapted to the new processor architectures. Seismic processing applications usually handle large amounts of data that are well suited for the task-level parallelism found in multi-core shared memory computer systems.

In this thesis, a large production code for seismic inversion is profiled and analyzed to find areas of the code suitable for parallel optimization. These code fragments are then optimized through parallelization and by using highly optimized multi-threaded libraries.

Our parallelizations of the linearized AVO seismic inversion algorithm used in the application, scales up to 24 cores, with almost linear speedup up to 16 cores, on a quad twelve-core AMD Opteron system. Overall, our optimization efforts result in a performance increase of about 60 % on a dual quad-core AMD Opteron system.

The optimization efforts are guided by the Seven Dwarfs taxonomy and proposed benchmarks. This thesis thus serves as a case study of their applicability to real-world applications.

This work is done in collaborations with Statoil and builds on previous works by Andreas Hysing, a former HPC-Lab master student, and by the author.

Acknowledgements

I would like to thank Dr. Anne C. Elster for being my supervisor, introducing me to high-performance computing, and helping me realize this thesis even though she's on sabbatical leave.

I would also like to thank Dr. Alf B. Rustad, Statoil Research, for being my co-supervisor, great technical assistance and interesting discussions.

Additionally, I would like to thank Statoil Research Centre Rotvoll for allowing me access to a real-world dataset and their resources. Also, thanks to the Norwegian Computing Center and its researchers for making the source code available.

Many thanks also go to HPC-Lab Post Doc Dr. Ian Karlin for valuable help and comments, and former HPC-Lab student Andreas Hysing for comments and sharing experiences from his work.

Finally, I would like to thank my fellow HPC-LAB students for support and generally good atmosphere throughout this stressful year.

Trondheim, June 30, 2011

Bent Ove Stinessen

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Glossary

ACML	AMD Core Math Library
ALU	Arithmetic and Logic Unit
AMD	Advanced Micro Devices
API	Application Programming Interface
ATLAS	Automatically Tuned Linear Algebra Software
AVO	Angle Versus Offset
BCSR	Block compressed sparse row
BLAS	Basic Linear Algebra Subroutines
DFT	Discrete Fourier Transform
DRAM	Dynamic Random Access Memory
FFT	Fast Fourier Transforms
FFTW	Fastest Fourier Transform in the West
GB	Gigabytes
GCC	GNU Compiler Collection
GEMM	GEneral Matrix Multiply
GPGPU	General-purpose computation on GPUs
GPL	GNU General Public License
GPS	Global Positioning System
GPU	Graphics processing unit

HPC	High-Performance Computing
I/O	Input / Output
ILP	Instruction-level parallelism
LAPACK	Linear Algebra PACKage
LSB	Linux Standards Base
MIMD	Multiple instructions, multiple data
MISD	Multiple instructions, single data
MPI	Message Passing Interface
NR	Norwegian Computing Center (Norsk Regnesentral)
NUMA	Non-uniform memory access
OS	Operating System
PDE	Partial differential equation
SISD	Single instruction, single data
SPMD	Single program, multiple data
SSE	Intel's Streaming SIMD-Extensions
TAU	Tuning and Analysis Utilities
UMA	Uniform memory access

Chapter 1

Introduction

Reflection seismology is a method of exploration geophysics. It can be used to estimate the structure of the Earth's interior from reflected seismic waves. Marine reflection seismology is of great benefit to the oil and gas industry, as it enables geophysicists to locate reservoirs in the seabed potentially containing hydrocarbons.

Conditioning Reservoir variables to Amplitude Versus Angle data (CRAVA) is a computer program developed by the Norwegian Computing Center (NR) in collaboration with Statoil, that performs *seismic inversion*. Seismic inversion computes elastic properties of a target area from seismic reflection data by inverting the physics equations for wave propagation. The process is the *inverse* of computing the data observations (seismograms) from the wave equations, or *seismic modeling* [36]. CRAVA uses a relatively new geo-statistical approach to the inversion problem, which is much faster than previous approaches. Additionally, the new approach enables quantification of the inherent uncertainties in the results, something that the older deterministic methods do not support.

The current trend in microprocessor architecture design is increased parallelism, and is a consequence of the increasingly difficult task of improving efficiency in the traditional uniprocessor architectures, as they approach the physical limits for frequency and power. Current microprocessors have several processing cores, each of which can sustain its own program execution. Using all the cores collectively to solve a problem can drastically improve program performance.

In this thesis, the seismic inversion algorithm in CRAVA is optimized for multi-core processors through parallelization, which in contrast to previous work results in good speedup. Additionally, optimized and multi-threaded libraries for linear algebra and Fast Fourier Transforms are applied to CRAVA.

1.1 Project goals

Statoil is one of the largest commercial users of high-performance computing (HPC) in Norway. Despite the fact that CRAVA runs on parallel hardware, it was not designed to use the available parallelism. Previous works by Andreas Hysing [28] and the author [44] successfully optimized sections of CRAVA for parallel hardware. However, the application has a large code base, and a great potential for further optimization exists.

The main goal of this thesis is to optimize CRAVA for modern multi-core architectures. To achieve this goal, the original code is profiled to locate areas suitable for optimization. The relevant sections of code will be classified according to the *seven dwarfs* taxonomy and proposed benchmarks [4], before optimization techniques are applied to them. The optimization techniques will include use of multi-threaded libraries and parallelization though OpenMP.

OpenMPI is not considered in this thesis due to the implicitly required algorithm decompositions, which are complex tasks and outside the scope of this thesis.

1.2 Outline

The rest of the thesis is structured as follows:

Chapter 2: Relevant background material and concepts important for this thesis related to parallel computing and optimization techniques are presented to give the reader the proper context. This chapter is in part adapted from the author's fall specialization project [44].

Chapter 3: The seismic inversion tool CRAVA is presented in more detail, and some key aspects of seismology are explained. Previous work is presented and the state of CRAVA at the beginning of this thesis is defined. This chapter is in part adapted from the author's fall specialization project [44].

Chapter 4: The profiling data for CRAVA are presented and suitable optimization techniques are explained and applied. Additional background material is presented for coherency.

Chapter 5: The effects of the performance optimizations applied to CRAVA in this thesis are shown. Each technique's results are compared to the original state, and the overall effects are summarized. The scalability of the optimized application is shown in results from a scaling-test.

Chapter 6: Conclusions made from the performance optimizations to CRAVA are presented. The contributions of this thesis are listed, along with recommendations for the application and ideas for future work.

Appendix A: Details of the various tests used in the thesis are listed.

Appendix B: Some additional timings and detailed results, supplementing the results in Chapter 5.

Appendix C: Relevant source code of the array transformation routines, and the seismic inversion process.

Chapter 2

Parallel Computing and Context

This chapter introduces background material and concepts used in this thesis related to parallel computing and optimization techniques. First, some basic concepts of parallel computing are presented, followed by how different forms of parallelism are manifested in hardware. Next, software profiling tools are introduced, the seven dwarfs are explained, and finally, optimization techniques are presented.

2.1 Parallel computing theory

As computers became increasingly more powerful, it became possible for larger and more complex problems to run on them. However, some problems are simply too large to run on a single computer. The solution for some of these problems is parallel computing. With multiple processors collaborating, large problems can be solved in a feasible amount of time. The computational capability of computers continue to increase, along with the computational demand. Contributors to the demand include problems that cannot be solved in a reasonable amount of time on current computers, called *grand challenge problems* [48]. An example is global weather forecasting. Weather predictions must be timely for the results to be useful. When more weather data can be processed in the available time period, weather predictions become more accurate and detailed.

2.1.1 Speedup

Two sections of a program can be run in parallel if they do not have any dependencies between them (e.g. data dependencies). If the sections are of equal computational size, the problem would ideally execute in $\frac{1}{p}$th of the sequential run-time on p computing units, an effect known as linear or ideal speedup [48]. A problem that can be immediately divided into completely independent parts is called embarrassingly parallel [48], and is straightforward to parallelize. Displaying the Mandelbrot set is an example of an embarrassingly parallel problem [48]. Some problems are not possible to decompose without introducing communication and synchronization between the parts, for example heat distribution problems using stencil based algorithms [48].

Parallel programs may contain one or more sections that cannot be parallelized. As Gene Amdahl observed in 1967, these sections limit the maximum speedup of the parallel program [1]. He formulated the parallel speedup as the equation:

$$S(p) = \frac{1}{f + \frac{1-f}{p}} = \frac{p}{1 + (p-1)f} \tag{2.1}$$

Where S is the speedup factor, f is the serial fraction of the program, and p is the number of processors. From Equation 2.1 it follows that the maximum speedup approaches $\frac{1}{f}$ regardless of how many processors are used. The limiting effect for various parallel percentages are shown in Figure 2.1.

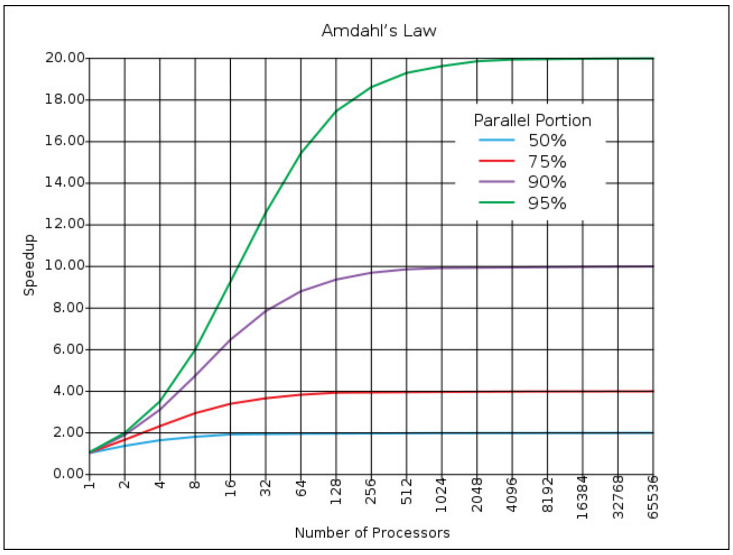

Figure 2.1: Amdahl's law for speedup given various percentages of parallelizable code. Figure from Wikipedia [46].

What Amdahl's law expresses is the maximum possible speedup, however actual speedup is usually less, and can be expressed as [48]:

$$S(p) = \frac{T_s}{T_p} \tag{2.2}$$

Where T_s is the observed time of the best sequential implementation, and T_p is the observed time of the parallel implementation using p processors. While speedup is usually lower than ideal, observed speedup can occasionally be greater than the number of processors used, an effect known as super-linear speedup [48]. Possible reasons for super-linear speedup are a sub-optimal sequential implementation, or caching effects. For example, when the program code or data fits in the increased amounts of memory or cache that come with using more processors, the need for disk or memory traffic is reduced. Another source of super-linear speedup can be better use of special purpose arithmetic instructions, for example fused multiply-accumulate which is often used in digital signal processing.

2.1.2 Memory architectures

There are two basic types of memory architectures for parallel computers [48]:

Shared memory systems have a single address space, reachable by all the processors in the system. A shared address space is convenient for programming, since all processors have access to the same data. The programmer does not have to do extra work to decompose the data and distribute it between processors. In small systems, memory interconnects like a bus (e.g. Intel's Front-side bus) or a switch (e.g. a crossbar switch [1]) is used to connect processors and memory modules. Unfortunately this does not scale very well, and in larger systems a bus quickly becomes a bottleneck, while switches require more space than feasible. Distributed shared memory is often a solution for larger systems: Each processor can still reach all of the memory, but access speeds vary. Architectures with varying memory access speeds are called non-uniform memory access (NUMA) architectures as opposed to uniform memory access (UMA) architectures, where the access speeds are constant for all memory locations. A multi-processor system is often a NUMA architecture, where each processor socket has its own memory modules, but can access the other processor sockets' memory through interconnection links (e.g. HyperTransport [2] links). The memory access time is

[1]A crossbar switch is a mesh of buses that provides routes between every processor and every memory module.

[2]http://www.hypertransport.org

dependent on the number of links a request must pass through.

In **distributed memory systems** each compute node has its own memory, and the nodes are connected through network interconnects. Sharing data between nodes requires network communication. Programming distributed memory systems is more complex than programming shared memory systems, because data transfers have to be explicitly handled by the programmer. For performance reasons it is preferable to operate on local memory as much as possible, reducing the network communication to a minimum.

2.1.3 Forms of parallelism

A common way to classify computers is by Flynn's taxonomy [22]. Flynn classified computers in terms of instruction streams and data streams, and called the traditional single-processor computer a *single instruction stream-single data stream* (SISD) computer. Multi-processor computers that can execute one program per processor, and each processor applies the program's instruction stream to different data, are classified as *multiple instruction streams-multiple data streams* (MIMD). In addition to these two extremes, two more classifications are defined: *single instruction stream-multiple data streams* (SIMD) and *multiple instruction streams-single data stream* (MISD). The term *stream* is commonly dropped from the definitions in newer publications [35].

Modern general purpose processors commonly have SIMD instructions in the form of short vector operations. In SIMD, the same instruction is applied to the multiple data elements in a short vector simultaneously. An example of SIMD instructions is Intel's Streaming SIMD-Extensions (SSE). SSE now has good compiler support and is implemented in many modern architectures (including architectures from other vendors, like AMD).

While SIMD is *data parallelism*, MIMD is *task parallelism.* Task parallelism means that multiple tasks can run simultaneously. Modern MIMD computers include multi-processor systems and multi-core systems. A popular programming model for MIMD machines is *single program-multiple data* (SPMD). In SPMD the same program is run simultaneously on all computing units, but each processor operates on different data. One of the benefits of SPMD is that only one program needs to be developed. Normally, the single program contains code that is executed only by certain computers in the group of computers, depending on the unit's identity within the group. A classic example of the SPMD programming model is MPI programs.

The MISD architecture class is not in use today, unless one specifically classifies pipelined architectures in this group [48], but as Flynn notes, MISD existed in the ancient plug-board machines.

2.2 Parallel hardware

Parallelism exists in different forms as described in Section 2.1.3, and different hardware exploits parallelism in different ways. Some of the most common technologies and hardware are described in this section. The performance of parallel hardware is usually measured in the number of floating-point operations performed per second, or *flops*. The maximum number of flops the hardware can yield is called its peak performance. Achieving peak performance is heavily dependent upon the problem and algorithms used, and experienced performance is often lower than peak due to sub-optimal use of the hardware.

2.2.1 Instruction-level parallelism

In 1965 Gordon Moore predicted that the amount of transistors we can cheaply place on a chip will double approximately every two years. For over 15 years, at the end of the 20th century, this was manifested in about 50 % performance increase every year as measured by the integer SPEC2006 [3] benchmark test [4]. Many of the extra transistors were used to exploit instruction level parallelism (ILP) [43], which led to rapidly increasing clock frequencies. ILP denominate concepts that enable the processor to issue more instructions in the same amount of time, while still preserving the sequential programming model . One of the concepts is *pipelining*.

In pipelining the instruction processing is split into smaller steps or micro-operations called stages. An instruction is partially processed at each stage, and follows a path of stages until completed. The benefit becomes apparent when an instruction is done with one stage and moves on to the next one, since a new instruction can then start processing in the first stage, thus increasing the number of instructions executed in a period of time. Instructions can be issued at the speed of the slowest stage in the pipeline, and the processor clock frequency is increased to match this speed. The slowest stage can limit the pipeline's performance if the gap up to the rest of the

[3]http://www.spec.org/cpu2006/

stages is large. The micro-operations can be generic and common for several instructions. Stages can also be duplicated to create multiple parallel data paths, enabling multiple instructions to execute at the same stage in the pipeline simultaneously. However, duplicating stages is complicated and requires much control logic.

The number of stages in a pipeline is called the pipeline depth. If a pipeline gets very deep, it can experience performance problems due to stalls and flushes. A *stall* of the pipeline occurs when an instruction I_1 is dependent on the result of an earlier instruction I_2 that has not yet finished executing. Instruction I_1 has to wait until the result from I_2 is ready before it can be executed further, and the pipeline stalls. One technique to reduce the performance penalty from pipeline stalls is out-of-order execution, where instructions are rearranged on-the-fly to keep the pipeline busy while an instruction is waiting for data.

A pipeline *flush* happens when a branch in the instruction stream (program) jumps to a different place in the stream. Since the instructions following the branch instruction have already started execution in the pipeline, a jump in the instruction stream means they have to be stopped or flushed from the pipeline. A pipeline flush means that many clock cycles are wasted on useless computation, and the number of instructions completed per cycle decreases. One technique to reduce pipeline flushing is speculative execution or branch prediction, where the processor predicts the outcome of a branch and starts its execution.

2.2.2 Multi-core

Much of the information found in this section is based on references Asanovic *et al.* [4] and Sodan *et al.* [43].

The amount of ILP is limited due to inherent dependencies in programs, and architecture designs eventually hit the *ILP wall*, where further performance gains were diminishing. Architectures also hit the *power wall*; the point where the heat generated from adding more transistors or turning up the clock frequency is more than what can be dissipated by conventional cooling. Additionally, the effect of increasing the clock frequency was limited by the performance gap between the processor and memory, known as the *memory wall*.

The ILP wall, power wall and memory wall comprise the *brick wall*. Due to the brick wall the fast uniprocessor was not an ideal design for further performance gain. The focus instead turned to task parallelism. By duplicating the control and execution units, multiple instruction streams or *threads* could run practically independent of each other. This enabled further performance increases without hammering the brick wall by simply increasing the number of processing cores (*multi-core*). The old sequential programs would still run as normal, but parallel programs were needed to unleash the full power within the cores.

The complexity of the cores is a deciding factor for the performance and energy consumption in new architectures. For highly parallel applications, simple processing cores are ideal (e.g. for graphics processing). However, for serial applications, more complex processing cores are better suited. From Amdahl's law we know that the serial part of a program can limit parallel performance. A heterogeneous mix of simple and complex cores, might be better suited for applications with both parallel and serial fractions than a homogeneous collection of simple or complex cores. The STI [4] Cell BE processor for instance, has one general purpose processor and eight synergistic processing elements.

The first multi-core architectures simply put multiple single-core processors on one chip, but more recent architectures are designed around the idea of cores. The current core designs are rather complex in nature, and often have private inner caches as well as shared outer caches. Ensuring cache coherency with the increasing number of cores will require a lot of die area and energy. For this reason, increasing the number of complex cores is expected to level out in diminishing returns rather quickly. Yet, the future chip multi-processors are expected to reach 100s, maybe 1000s of processing cores. Simpler core designs are one way to achieve these *many-core* processors. Some prototypes experiment with message passing protocols between the cores, resembling a small cluster [20].

2.2.3 Clusters

Clusters are the traditional workhorses of parallel computing. A cluster consists of multiple computers (nodes) connected by a high-speed network. A parallel algorithm is used to delegate a part of the problem to each node, and the problem is collectively computed by the cluster [48]. To achieve high utilization of the hardware, high locality in the computation and little communication is important. A relatively powerful cluster can be built using inexpensive commodity workstation hardware (Beowulf cluster) [48],

[4]STI is an alliance between Sony, Toshiba and IBM.

but supercomputer clusters are often specially designed using high performance components. The nodes of a cluster are usually made of homogeneous hardware, but recently installations of heterogeneous clusters have emerged. Several of the top positions on the TOP500 list [5] are now heterogeneous clusters, mixing traditional CPUs with GPUs or Cell processors.

Clusters are usually programmed using the SPMD programming model, and communication between the nodes is often handled using the Message Passing Interface (MPI) [48]. MPI is an API specification and many implementations exist. Vendors often maintain MPI implementations specially tuned to their hardware. At the base of MPI is the concept of communication groups, defined as subsets of all the processors. Within a group, each processor gets an identity that is used for sending and receiving messages. These identities can be assigned to form virtual topologies in the group that fits the problem. MPI has both point-to-point (e.g. send, receive) and collective (e.g. broadcasts, reductions) communication methods. Definable data-types to simplify the use of complex data patterns are also available.

Clusters make it possible to work with large datasets, since single machines do not have to process all the data. Machines are usually designed for compute-intensive applications, where performance is measured in number of arithmetic operations per second. However applications can also be data-intensive, or heavily I/O bound, which make them dependent on the ability to process more data. With datasets in the terabyte or petabyte scale, distributing the data for every computation quickly becomes impractical. New architectures are necessary and the focus of a still developing research area. A possible solution might be similar to what Google is using for their Internet search [7]: The dataset is partitioned over groups of processors which become responsible for its part. Computations on the data is sent to the responsible group instead of transferring the data around. A computation request might use the MapReduce abstraction [15].

2.2.4 GPGPU

In the recent years general purpose computation on graphics processing units (GPGPU) has emerged as an attractive platform for high-performance computing. The GPU was developed as a co-processor to the CPU that accelerated graphics rendering and game physics [34]. It revolutionized computer games and has become increasingly more powerful since its introduction.

[5]List of the most powerful computers systems in the world. http://top500.org

Graphics calculations are naturally parallel, which has resulted in GPU designs resembling symmetric multi-core processors [6].

A new market opened up for GPUs when the HPC community became interested in using them for general purpose computation. GPU vendors quickly provided programming interfaces and adapted the hardware to computing standards. The most used programming frameworks are NVIDIA's CUDA [40] and Khronos Group's OpenCL [38], which is embraced by AMD (ATI). Both are based on the C/C++ programming languages. A close relationship between frameworks and vendors can be a problem for portability [6], which is the case for e.g. CUDA.

Recent GPUs have improved support for floating-point arithmetic and double-precision numbers [19]. GPUs are optimized to run compute-intensive SPMD programs with little or no synchronization [6]. The programs (or *kernels*) are run in parallel on several multi-processors. Each multi-processor has a number of computing cores which process one thread each. Spawning threads on a GPU costs very little overhead, and several thousand threads can easily run at once. Collections of threads (called *warps*) are run simultaneously on the multi-processors, and can be quickly switched around to hide memory latency. To achieve full memory bandwidth, requests have to be *coalesced* [40]. Meaning that several threads access different addresses within the same memory bank.

The general difference between the CPU and GPU architectures can be seen in Figure 2.2. The GPU has fewer transistors devoted to caching data and flow control, but much more to data processing [40].

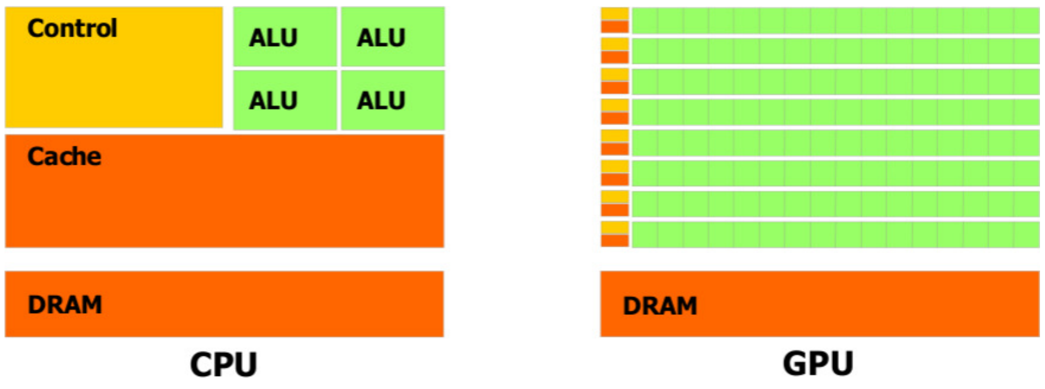

Figure 2.2: Comparison of the CPU and GPU architectures. Figure used with with permission from NVIDIA [40].

A single GPU today can have a theoretical peak performance of several teraflops [6, 40]. The fastest GPUs have the potential for about 10 times more flops than the currently fastest multi-core CPUs, but achieving this performance is heavily dependent on the problem and the algorithm used. Important factors that can limit throughput are synchronization, memory accesses and compute-intensity. One study found that on a selection of kernels tuned for both platforms, the GPU only performed 2.5x better than the CPU on average [34]. And that was without including the costs of moving data between the host and GPU, which can substantially limit the performance advantage [14]. These costs are unavoidable when the GPU is used as a co-processor, but can be reduced if the kernels are highly compute-intensive.

2.3 Profiling

Profiling an application involves analyzing and mapping certain characteristics of the program. These characteristics can be the most frequently called functions, cache utilization or where the program spends the most of its time. Profiling information can help determine which sections of a program to prioritize when optimizing.

Usually a program has some sections where it spends more time compared to other sections. An often quoted trend of this fact is the *90/10-rule*: 90 percent of the execution time is spent in just 10 percent of the code [30]. The 90/10-rule is common for data-intensive applications with large loop structures. An optimization within the most used 10 percent of the code will have a much greater effect on the overall run-time than an optimization within the remaining 90 percent.

Programs that can not be classified using the 90/10-rule can still benefit from profiling. However, their run-time usage is more evenly distributed throughout the program, and more optimization effort is usually needed to achieve large speedups.

2.3.1 Types of profilers

There are three subcategories of profilers: event-based, statistical and instrumenting. They differ in how data is gathered, their accuracy, and how they affect program execution.

Event-based profilers use properties of the programming language to listen for events. The language usually supports “trapping” events, such as function calls and object creation. They introduce minor overhead and cause the program to run at a slightly reduced speed. The run-time of the profiler's own code can cause additional slowdown.

Statistical profilers use operating system features to interrupt the program execution and capture run-time information. Sampling the program counter can tell you exactly where execution is in the program at given intervals. Based on the sampling a statistical approximation of the program's run-time profile can be generated. Statistical profilers are by design not entirely accurate, but allow the program to run at near full speed.

Instrumenting profilers insert additional code in the program to perform data gathering. The additional code can severely alter run-time and result in large slowdowns, but the upside is that they can produce entirely accurate data. The instrumentation code can be added in several ways. Manual instrumentation involves that the programmer manually writes code to collect data or time sections. Special tools can also inject this code automatically. The compiler might alternatively support automatic instrumentation, like the GNU profiler gprof [21]. Finally, the instrumentation can occur at run-time through the use of special dynamically linked libraries, or altering the binary executable.

2.3.2 Available profilers

Many different performance profilers exists, and a selected few are presented in this section for insight and relevance. While many commercial and vendor supplied profilers exists (e.g. Intel VTune and AMD CodeAnalyst), the profiling tools described here are all open source with either a GPL or BSD-style license.

Valgrind

Valgrind [6] is an instrumentation framework for dynamic analysis tools [16]. It comes bundled with several tools for analyzing memory usage and program behavior. One of those is Memcheck which uncovers heap data bugs, including memory leaks. Another one is Cachegrind, which profiles cache utilization. Callgrind is a modified version of Cachegrind that also creates a graph over the function calls. Valgrind also has two tools for detecting errors in multi-threaded programs (e.g. race conditions), named Helgrind and DRD.

[6]http://valgrind.org

Valgrind uses dynamic binary instrumentation to profile a program. Meaning it injects instrumentation commands directly into an intermediary representation of the binary executable. The instrumented code is run on a CPU emulator within the Valgrind core. The CPU emulator approach can profile programs or libraries for which no source code is available. Depending on which tools are used the slowdown factor due to profiling can range from 5 to 100 or more [16]. The output from Valgrind is more useful if the program is compiled with debug information, which is rarely the case with external libraries.

The output of Cachegrind and Callgrind can be visualized in KCachegrind [7]. It is a separate tool and not part of Valgrind. KCachegrind offers a graphical user interface which makes the results much easier to review. Features include an interactive visual call-graph, maps over calls to or from a certain function, and annotated source code. The instruction count is used to relate data to time usage, but no actual timing occurs.

TAU

Tuning and Analysis Utilities [8] (TAU) is a parallel performance system from the University of Oregon. It features a framework and toolkit for performance instrumentation, measurement, analysis and visualization of large-scale parallel applications [42]. TAU is an instrumenting profiler, and supports a multitude of instrumentation approaches, including manual, dynamic and completely automatic. Scripts to automate the instrumentation during compilation make TAU easy to use. The profiling is completely customizable, and it is possible to accurately define which files or functions to profile. There is also support for analyzing memory usage and I/O activity.

Several visualizing programs exists for TAU. From the simple terminal text representation of *pprof* to the graphical representations of the very powerful *ParaProf*. TAU supports most of the popular programming languages, including C/C++, Java and Python. TAU was designed with parallel programs in mind, and support for threads, MPI and multiple nodes is present.

In an attempt to reduce performance penalties due to profiling, TAU stops tracking a function if it is called more than 100 000 times and uses less than 10 µs per call. These numbers can be changed if necessary.

[7]http://kcachegrind.sourceforge.net/
[8]http://tau.uoregon.edu

GNU gprof

Gprof is the GNU Profiler, and is included in the GNU Binutils collection [9]. It analyses profiling data from a program compiled and linked with GCC and the flag *-pg*. Gprof uses both instrumentation and statistical profiling. Every function is instrumented to log function calls, and a statistical time usage estimate is collected through sampling. The use of sampling means the time usage estimates are subject to statistical errors [21]. Although gprof produces readable textual output, the output of gprof is better visualized in KProf [10]. Similiarly to KCachegrind, KProf is not part of gprof but an external visualization tool.

Gprof does not support multi-threaded applications. Due to the sampling method used, which relies on a timed kernel signal, only the main thread is profiled if several threads are spawned. The spawned threads do not react to the kernel signal, since they do not know what signal to listen for. There exists a relatively simple workaround for this behavior using a dynamically loaded wrapper for pthreads [27].

2.4 Seven dwarfs

In 2006 a group of Berkeley researchers sat down to discuss the impact of multi-core on microprocessor design. In their paper [4] they make several recommendations for the future of chip multi-processors. One of them is to use a collection of "dwarfs" instead of traditional benchmarks to design and evaluate parallel programming models and architectures. The dwarfs are algorithmic kernels that captures patterns of computation and communication. The reasoning is that new architectures and programming models that perform well on these patterns will be well suited for applications of the future. Another key point is that optimizing and parallelizing every computer problem is non-trivial and very complex. By instead focusing on generic methods, a large number of applications can benefit from the same optimization efforts. The dwarfs represent these generic methods.

In the Berkeley paper [4], 13 dwarfs are identified. The first seven dwarfs originate from a presentation by Phil Colella, in which he identifies seven numerical methods he believes will have future importance for scientific computation. The seven dwarfs are presented below along with a short description based on Asanovic *et al.* [4].

[9]http://www.gnu.org/software/binutils/
[10]http://kprof.sourceforge.net/

1. **Dense Linear Algebra** Linear algebra on dense matrices and vectors. This dwarf is often computationally bound (matrix-matrix) and benefit greatly from block algorithms that use the cache to keep the CPU occupied, as well as vector instructions. The benchmark used for the Top 500 list is based on dense linear algebra (Linpack benchmark [11]).

2. **Sparse Linear Algebra** Linear algebra on sparse matrices and vectors. A sparse matrix contains many zero values and is usually stored in a compressed format to reduce storage and bandwidth requirements (e.g. block compressed sparse row (BCSR) format). Sparse linear algebra is often limited by memory bandwidth.

3. **Spectral Methods** Spectral methods solve problems numerically when the data are in the frequency domain, as opposed to time or spatial domains. Typically, only the transformation of basis is classified as a spectral method, e.g. the Fast Fourier Transform. Spectral methods are often limited by memory latency.

4. **N-Body Methods** N-body methods depend on interactions between many discrete points. Methods include particle methods where every point is dependent on all other points, e.g. predicting the motion of astronomical bodies in space. This dwarf can lead to $O(n^2)$ calculations and is computationally bound.

5. **Structured Grids** Data represented in regular multidimensional grids, where the points are conceptually updated together. Points in close spatial proximity is often accessed with a 5 or 7 point stencil for two or three-dimensional grids. The dwarf is currently more memory bandwidth limited.

6. **Unstructured Grids** Conceptually similar to structured grids, but data point locations and connectivity to neighboring points are explicitly defined. Updates typically include multiple levels of memory reference indirection. The dwarf is limited by memory latency.

7. **Monte Carlo** Calculations depend on statistical results of repeated random trials. The dwarf is considered embarrassingly parallel, and has very little communication.

While the original seven dwarfs apply mostly to high performance computing, the Berkeley researchers identify an additional six for other application

[11]http://www.netlib.org/benchmark/hpl/

areas. These areas include embedded systems, machine-learning, general-purpose computing, databases, graphics and games. The additional dwarfs are *Combinational Logic*, *Graph traversal*, *Dynamic Programming*, *Backtrack and Branch+Bound*, *Construct Graphical Models* and *Finite State Machine.* More information about these dwarfs is found in the paper [4].

2.5 Optimization techniques

Just because a program is run on hardware that supports parallelism, does not automatically mean that program execution time will decrease. The program must either be coded or compiled for parallelism, with the exception of hardware parallelism like ILP. Sometimes the parallelization is handled automatically by the compiler, but usually it requires an implementation effort by the developer. Exceptions exist where external libraries can contain parallelism or automatically parallelize requests, but the program is completely serial.

Adapting a program to better use the hardware is called optimization. Examples of optimizations are eliminating branches and excessive function calls. Some compilers can also make use of developer supplied hints to create a better machine code structure, e.g. GCC's *builtin_expect()* (popularly referred to as likely/unlikely) hints on conditional branches. One optimization technique for parallel hardware is parallelization, and means to introduce parallelism in the program. Parallelizations can be done at many levels, ranging from automated compiler vectorization to manual decomposition of the problem.

2.5.1 OpenMP

OpenMP is a portable API for easy multi-treaded parallelization. It is based on work-sharing between concurrent threads, and provides compiler pragmas (hints) to identify parallelism. All the details of creating and destroying threads, load balancing and synchronization are managed by the compiler. OpenMP has become the de facto standard for easy shared-memory parallel programming [5]. Figure 2.3 shows the OpenMP threading model. When an annotated parallel section is encountered, a number of worker threads are spawned and work is shared between them. Originally only loop-level and predefined task parallelism were supported. However, version 3 adds support for irregular parallelism and dynamic tasks [5]. In OpenMP, it is up to the programmer to ensure dependencies in tasks are handled, unlike for instance the similar framework SMPSs that feature a dependency aware task programming model [41]. The programmer needs to be careful to avoid flow dependencies and race conditions, and if necessary, override the default settings for data sharing and provide synchronization and locking.

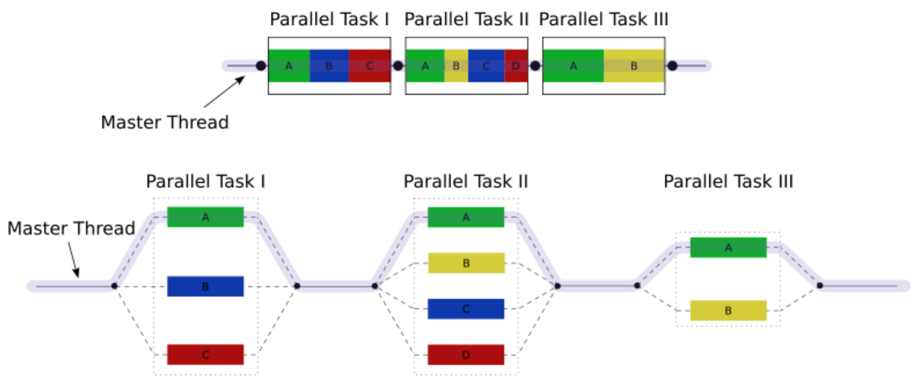

Figure 2.3: The threading model of OpenMP. The serial program execution is on top, and the threaded execution at the bottom. Figure from Wikipedia [47].

The use of compiler directives allows a serial program to be parallelized incrementally. Injecting parallelism into an existing program can be done with a relatively small implementation effort, reducing the chance of introducing unintentional bugs. If the parallel OpenMP program is compiled without OpenMP support it will compile just like the serial program, enabling easy debugging and portability. OpenMP officially supports C/C++ and Fortran, and is implemented in many of the most popular compilers. GCC has supported OpenMP 2.5 since version 4.2 and OpenMP 3.0 since version 4.4.

2.5.2 Libraries

Another way to increase performance is by using externally maintained libraries for common kernels. Using libraries often reduce development time since less functionality have to be implemented. Libraries are often heavily optimized by experts on the specific problems, sometimes even deploying auto-tuning. By using dynamically loaded libraries, the same application can achieve good performance portability through platform specific highly tuned kernel routines.

Auto-tuning

Auto-tuning is a technique to create portable code that adapts to the hardware for maximum performance [4]. The adaptation is achieved through testing different versions of the code and either through brute force or heuristics determining the best settings. These settings can be tile sizes, versions of SSE, different algorithms or implementations that favor the target hardware. Auto-tuning can be performed at either compile-time or run-time. Performing the tuning at run-time limits performance, but can be the best choice for certain applications (e.g. sparse linear algebra). A library auto-tuned at compile-time can take hours to install, but can offer near optimal performance to all programs using it. Performance can even trump hand-tuned libraries from the vendors. In the recent years, more auto-tuners for parallel codes have started appearing [29].

2.5.3 BLAS and LAPACK

The most used application programming interface for matrix and vector operations in libraries is the Basic Linear Algebra Subprograms (BLAS) standard [33, 18, 17]. It defines subprograms, or functions, for the most used linear algebra operations. The functions are grouped in "levels" where each level has increasing input dimensionality and complexity order [17]. The different levels of BLAS are listed in Table 2.1. The BLAS standard comes with a reference implementation, but is also independently implemented in various libraries. CPU vendors often maintain a tailored implementation for their specific architectures. Common optimizations include cache-tiling, SIMD-vectorization and threading for multi-core.

Table 2.1: The different levels of BLAS.

BLAS level	**Type of operations**
Level 1	Vector operations, incl. scalar-vector operations.
Level 2	Matrix-vector operations
Level 3	Matrix-matrix operations

BLAS routines are often used to build larger libraries, e.g. the Linear Algebra PACKage (LAPACK) [2]. LAPACK provides routines for solving linear equation systems, eigenvalue problems and more. Since it uses the architecture specific routines of BLAS (when available) it will often achieve good utilization of hardware.

Compatibility with C/C++

The original implementations of BLAS and LAPACK are written in FORTRAN. It is fully possible to link FORTRAN object files with C/C++, and the libraries even support this by providing C interfaces. However, FORTRAN uses a different memory layout for matrix arrays than C/C++. In local memory, multidimensional arrays are saved as pseudo-multidimensional arrays. Pseudo-multidimensional means that the array is really stored contiguously in a one dimensional array, but is accessed in a multidimensional fashion. The compiler translates multidimensional indexes in the program code into strided memory addresses in the pseudo-multidimensional array. The size of the stride is defined in the programming language as a function of array size. In FORTRAN, indexes are translated in column-major order, while C/C++ and many modern languages use a row-major order translation. The difference between the two is illustrated in Figure 2.4.

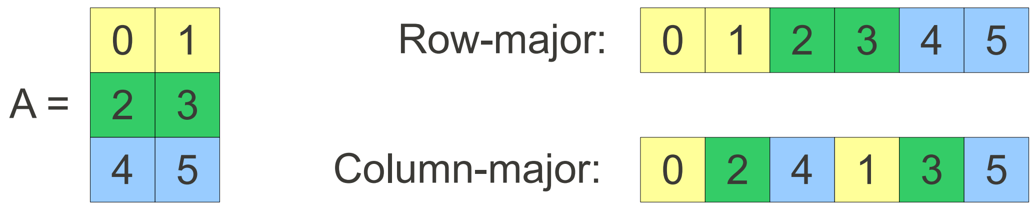

Figure 2.4: The difference between Column-major order and Row-major order. Row-major stores the rows after each other, while column-major stores the columns after each other.

In C and C++ multidimensional arrays are arrays of arrays. If the dimensions are defined at compile-time this is just an abstraction of pseudo-multidimensional arrays. If the arrays are dynamic (on the heap) the multidimensional array will be an array of pointers to arrays. The memory structure is not predictable for dynamic arrays, and the code has to dereference the pointers to reach the data.

There exists several versions of LAPACK made for C++, e.g. CPPLAPACK and LAPACK++. However, these versions of the library require the use of a special matrix data type. While this can be very advantageous when writing a new program from scratch, it is far more complicated to convert an existing application. Converting is complicated because every reference, indexing and use of the arrays needs to be carefully changed to avoid introducing bugs.

ACML

ACML [12] is AMD's core math library. It is copyrighted by AMD, but is available for free for anyone to use. ACML contains BLAS and LAPACK routines optimized for AMD processors, as well as some FFT routines and a random number generator. Both single and multi-core versions are available. ACML contains some FORTRAN code and use the column-major order memory layout for pseudo-multidimensional arrays.

ATLAS

ATLAS [13] [45] is a portable BLAS implementation based on auto-tuning. It is open source (GNU General Public License) and often regarded as the best optimized version of BLAS. Some LAPACK routines are also available. ATLAS is built on a C-port of BLAS (CBLAS) and supports both column-major order and row-major order memory layout for pseudo-multidimensional arrays.

2.5.4 FFTW

FFTW [14] [25] is a free open source (GPL) library for fast Fourier transforms (FFT), which is the most efficient type of discrete Fourier transform (DFT). FFTW is known as one of the fastest implementations of the FFT for CPUs [23]. It can compute transforms for arrays of arbitrary size in $O(n \log n)$ time. FFTW uses a form of auto-tuning to achieve good efficiency on different hardware. By measuring the performance, FFTW can choose the most optimal algorithm before performing the transformation. If the number of repeated transforms is sufficiently large, this technique can yield very good performance. However for single or few repeated transforms, it is usually faster to let FFTW estimate the best algorithm. Estimation is many times faster than measuring and hardly impacts the transformation time at all. The estimated or best measured algorithm is returned from the library as an "execution plan", which is a transformation scheme and can be used for any number of transforms.

[12]http://developer.amd.com/libraries/acml/
[13]http://math-atlas.sourceforge.net/
[14]http://fftw.org/

Chapter 3

CRAVA

CRAVA is an application for performing elastic seismic inversion. It is open-source and written in C and C++. CRAVA was developed by the Norwegian Computing Center (NR) in collaboration with Statoil.

Seismic, or geophysical, inversion involves determining the interior structure of the earth from data obtained at the surface. In seismic inversion, posterior (sub-sea) elastic parameters are estimated from seismic reflection data and well logs. These parameters are vital for use in geological modeling. Seismic inversion has been used for quite some time [36], and has traditionally been treated as a deterministic problem. One of the features that sets CRAVA apart from other implementations is the use of a relatively new Bayesian (statistical) approach to the inversion process, which provide quantifiable uncertainties and errors [12].

This chapter presents CRAVA in more detail, and explains concepts used in seismic processing. First, seismic data is introduced along with the acquisition process. Then, a more thorough look at CRAVA and its features is presented, followed by the testing scheme used in this thesis. Finally, we present previous work on CRAVA, and its state before the optimizations of this thesis began.

3.1 Marine seismic data acquisition

Seismic waves are energy waves that travel through the ground. They can be the result of an earthquake, an explosion or artificial generation. Seismic acquisition is the process of gathering wave data over time, for further processing. Gathered seismic data can be used for the prediction of earthquakes, detection of explosions and mapping subsurface geological features [37].

To obtain seismic data for a marine survey, a vessel drags a seismic source in the water behind it. This source is most often air guns that shoot pressurized air towards the seabed [32]. The frequency of these pressure waves take the shape of wavelets [1], and are in part reflected at boundaries between two media with different acoustic impedances [37], termed *facies*. The reflections move at an inverse angle of the source wave, and are recorded by hydrophones, which is underwater microphones sensitive to pressure [32]. Several hydrophones are attached to a *streamer* a few meters under the waterline and towed behind the acquisition vessel.

Before the seismic data can be used with CRAVA, they need to be preprocessed (pre-stack migration) [9]. The responses with similar angles are grouped together in so-called partial angle stacks, and treated as the same angle ϕ. GPS is used to relate the data to geological locations. An illustration of the setup can be seen in Figure 3.1.

3.2 CRAVA overview

CRAVA is an acronym for "Conditioning Reservoir variables to Amplitude Versus Angle data." Angle data refers to the way the angles between downward and upward traveling seismic waves change according to the increasing distance (offset) between the seismic source and consecutive hydrophones [3]. Amplitude refers to the signal intensity of the reflection in the partial angle stacks, as recorded by the hydrophones. These amplitude data contain noise and measurement uncertainty, which are taken into account by the Bayesian linearized AVO (Angle Versus Offset) inversion method. The main advantage of the method is that uncertainties in the results can be quantified [3, 12]. Typical datasets are tens to hundreds of gigabytes (GB), and the program can use several hours in normal operation for the larger sets.

[1]A wavelet is a wave-like oscillation with an amplitude that starts out at zero, increases, and then decreases back to zero.

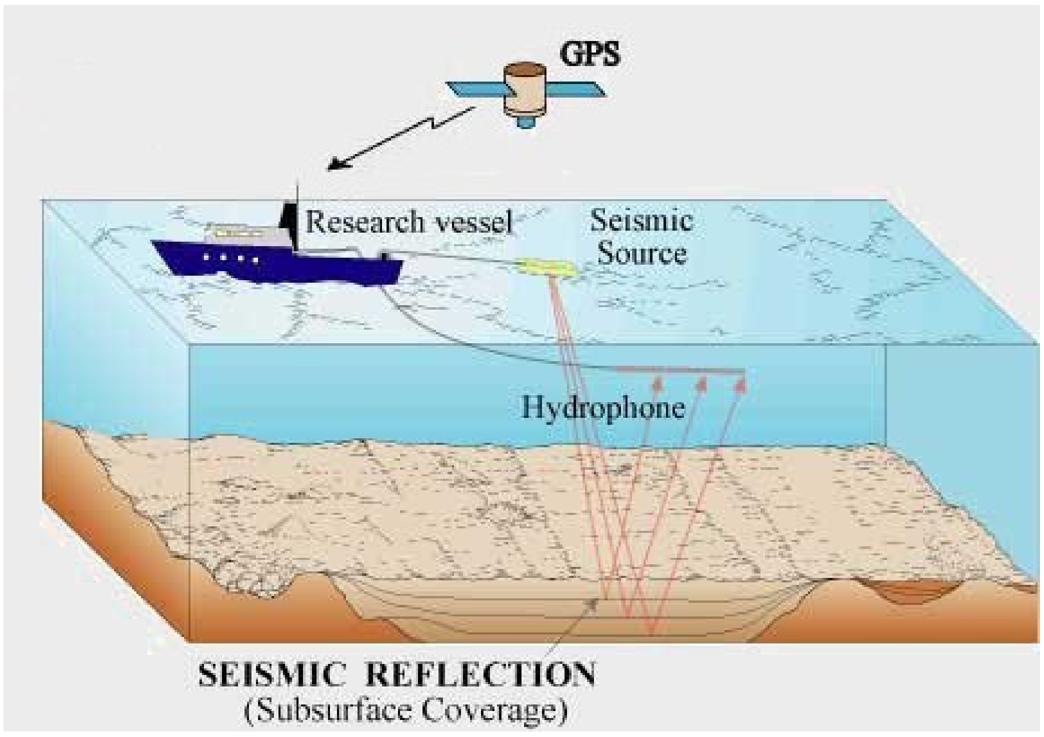

Figure 3.1: The seismic acquisition process.
Original illustration courtesy of the U.S. Geological Survey. (Public domain)

One output of CRAVA is the inverted elastic parameters V_p (pressure-wave velocity), V_s (shear-wave velocity), and ρ (density). These parameters can be used for further geophysical analyses and modeling. Pressure waves are compressional waves in the same direction as the energy propagation, and can travel through any material. Shear waves are transverse waves that displace the material perpendicular to the energy propagation direction. The difference is illustrated in Figure 3.2.

3.3 Features

CRAVA has three modes: inversion, estimation and forward modeling. The primary mode is inversion. CRAVA uses a new geo-statistical inversion method, which transforms the problem to the Fourier domain. Here the inversion problem can be solved independently for each frequency component, before it is transformed back to time domain. The approach reduces time complexity from $O(n^{2.x})$ to $O(n \log n)$, making it faster than the traditional

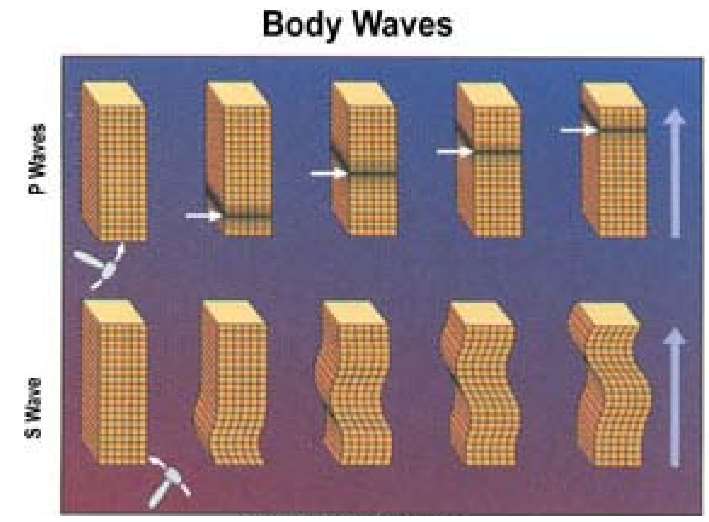

Figure 3.2: The difference between pressure (P) and shear (S) waves. The large arrow shows the energy propagation direction, while the little hammers indicate the displacement direction of the wave.
Original illustration courtesy of the U.S. Geological Survey. (Public domain)

inversion methods, while enabling the use of moderate computer resources [12, 13, 31]. The elastic parameters generated in inversion can either be predicted as the "most probable" values, or stochastically simulated using a Monte Carlo method [2] [11].

Before the inversion, prior background models are constructed for the earth, based on log data from drilling wells. The logs contain basic information about which facies exists (e.g. sand or shale), and at what depth. Inversion then predicts the posterior distributions matching the seismic data, while updating the models. The posterior distributions can be predicted analytically due to the linearized relationship between the AVO data and the elastic model parameters. The analytical method exploits the fact that reflection strength depends on the reflection angle and material properties where the reflections take place [9].

The estimation mode checks the quality of the data and estimates missing information for inversion, but does not perform the actual inversion. Forward modeling is a minor feature that generates synthetic seismic data from the background model. This mode is a joint estimation and inversion mode,

[2] Computational algorithm that relies on repeated sampling of a random number generator. It is useful for modeling phenomena with significant uncertainty in inputs.

which is also used for quality assurance. CRAVA will, when possible, estimate any information not supplied. Additional steps in CRAVA include kriging the results to the wells and the commonly used facies probabilities generation. Kriging is a form of geo-statistical interpolation, while facies probabilities are cubes with the statistical probability for each of the facies. The different facies have different oil capacities, and finding areas containing e.g. sand, is more interesting than areas of shale, since the probability for oil is higher in sand.

3.4 Testing

The CRAVA source code is bundled with a test suite containing ten tests, detailed in Table A.1. These tests run the working-copy of CRAVA against small synthetic datasets, and compare the output to previously generated results. Any anomalies in the computed data are reported as errors, and if measurable the difference is shown. The tests are used throughout this project to verify that any changes do not compromise accuracy or correctness. It is worth noting, that using compiler optimization flags changes the floating-point rounding, and CRAVA reports the variances as errors. The variances are however small, and since rounding errors naturally occur with small floating-point numbers [26], these errors are ignored.

The tests' datasets are small compared to real datasets and only suited for verifying correctness. For benchmarking a much larger dataset is used. This dataset originates from a Statoil-operated oilfield on the Norwegian continental shelf. Details are listed in Table A.2, and the test is referred to as the *benchmark test* in this thesis.

3.5 Previous work and current state

Andreas D. Hysing worked on optimizing CRAVA in his master's thesis [28]. His work involved optimizing and parallelizing the re-sampling algorithm, and parallelizing the inversion step. OpenMP was used to parallelize the code for shared memory systems. Re-sampling showed a 60 % serial speedup and good parallel scalability. The parallelization of inversion did not show any parallel speedup. A FFT settings store was tested to cache settings for fast Fourier transformations and support re-use of *execution plans* (as defined in Section 2.5.4). It achieved a small improvement in run-time, but did not significantly impact overall application performance.

In the author's fall specialization project, on which this thesis is based, two sections of CRAVA were profiled and parallelized [44]. The sections were *Prior expectation* (background model) and *Building the Stochastic model*. Additionally, some optimizations were made to the FFT settings store. Good local speedups were achieved resulting in 3.8x and 2.3x speedups for the two sections, and about 18% overall improvement for normal operation [3].

Listing 3.1: Original time usage of CRAVA at the start of this thesis.

```
Section                              CPU time                 Wall time
--------------------------------------------------------------------------
Loading seismic data         123.30     0.83 %         141.00      4.26 %
Resampling seismic data     1976.60    13.32 %         280.00      8.46 %
Wells                         12.34     0.08 %          12.00      0.36 %
Prior expection              606.36     4.09 %         136.00      4.11 %
Prior correlation              7.38     0.05 %           7.00      0.21 %
Building stochas. model      741.51     5.00 %         313.00      9.46 %
Inversion                  10751.13    72.45 %        1754.00     53.01 %
Parameter filter             429.33     2.89 %         429.00     12.96 %
Rest                         191.08     1.29 %         237.00      7.16 %
--------------------------------------------------------------------------
Total                      14839.04   100.00 %        3309.00    100.00 %
```

Listing 3.1 shows the different sections of the program and their original time usage. The timings includes the previous works mentioned above by Hysing and the author. To obtain these timings, the benchmark test was used on a test system with a total of 8 cores, introduced in Section 4.1. Inversion is now the most time consuming part by far, with parameter filtering on a distant second.

[3]When comparing normal operation on the same system as used in this thesis with OpenMP 3.0 and ignoring time used to load data. Ref. Listings B.16 and B.17 in [44].

Chapter 4

Performance Analysis and Optimizations

In this chapter, we present implementation details of our performance optimizations. First, we describe our test and development environments. Then we present the profiling methods used and tools tested. Next, for each *dwarf* found in CRAVA, the dwarf is explained and its performance data from profiling presented, before optimization techniques are applied to it. Finally, the seismic inversion algorithm is studied in more detail and its performance data presented, before the algorithm is optimized through parallelization.

4.1 Test and development environments

In this thesis, we use the real-world dataset explained in Section 3.4 for testing. To satisfy the memory requirement for this dataset, a compute node from a large cluster is used. This test system is detailed in Table 4.1, while the workstation system used for day-to-day development is described in Table 4.2. Since the test system only contained an old version of GCC and installing a newer version was impossible due to permissions, it was necessary to cross-compile CRAVA on the development system to be able to use OpenMP 3.0. Compiling CRAVA returns an executable binary file. For the executable file from the development system to run on the test system, which run a different version of Linux and accompanying libraries, LSB can be used.

Linux Standards Base [1] (LSB) is a project to increase application compatibility between Linux distributions. It specifies standard libraries, file system

[1] *http://www.linuxfoundation.org/collaborate/workgroups/lsb*

Table 4.1: Benchmark test system.

System	**Cluster compute node**
CPU(s)	Quad-Core AMD Opteron 2356 (Barcelona)
Frequency	2.3 GHz
Cores	8 (2 sockets × 4)
Memory	32 GB
Instruction set	amd64 (x86_64)
OS	Red Hat Enterprise Linux Server release 5.4
GCC	4.1.2
OpenMP	2.5
LSB	3.1

Table 4.2: Development system.

System	**HPC-LAB workstation**
CPU	Intel Core i7 930 (Nehalem)
Frequency	2.8 GHz
Cores	4 (8 with HyperThreading)
Memory	6 GB
Instruction set	x86_64
OS	Ubuntu 10.04
GCC	4.4.3
OpenMP	3.0
LSB	2.0, 3.0, 3.1, 3.2, 4.0

layout and more. The idea is that when an application is LSB compliant, it can run on any distribution that is also LSB compliant of the same version or newer, even in binary form. Thanks to LSB it has become easier to distribute binary programs for Linux without having a multitude of versions or restrictions on distributions.

To get a fully LSB compliant executable, a special compiler wrapper is used. This wrapper tells the compiler to use the proper LSB linker and libraries. On CRAVA this wrapper caused a myriad of compiler errors, so a simpler solution was used: The program was linked statically, which means that all the code for external library calls are included in the executable file. By statically linking CRAVA it can run independent of system libraries, as opposed to dynamic linking where the system libraries and versions are required to match the executable. An advantage of dynamic linking is the possibility of platform specific implementations of libraries, potentially auto-tuned to the system.

For CRAVA, libraries like the standard C++ library and OpenMP are statically linked, which require the entire pthreads library to be statically linked as well. The GNU C++ compiler has to be persuaded to statically link the standard C++ library, and doing so causes known complications if the program manually loads dynamic libraries with *dlsym*. Luckily CRAVA does not, and the resulting binary is fully self-contained. The binary was examined with the LSB compliance checker and found to be compatible with many different Linux distributions, including Red Hat 5 and the test system.

4.2 Profiling

CRAVA generates usage reports like the one in Listing 3.1, giving a coarse view of the time used by the application. The sections in these reports can represent large amounts of code, and more information is needed to determine where to focus the optimization effort. To make good decisions it is important to have performance data that is as accurate as possible. Therefore, it is vital to let CRAVA run mostly unaffected by the profiling code.

Slow network storage I/O accounts for a large percentage of CRAVA's total time usage. The unpredictable network latency results in very fluctuating timings for certain parts of the program. To reduce the impact of the network latency and obtain more consistent timings, an extra run-through of the program is performed before timings are recorded. This extra run-through lets the operating system cache most of the input data in memory, which helps speed up consecutive runs and provide persistent I/O timings.

In this thesis, several profiling methods were tested and used. Below follows a short description and discussion of the different methods.

Valgrind

Valgrind with the tool Callgrind produce informative call graphs of an application. It uses the number of instructions to relate functions to time use, which gives an indication of the time usage distribution but not exact timings. An indication can be useful for most programs, but when tested on CRAVA it was found to be impractical. In fact, due to the increased memory requirement, the test system (Table 4.1) could not run CRAVA in Callgrind. Another contributing factor is the implied slowdown as explained in Section

2.3.2, which makes testing ineffective. Due to these issues Valgrind was not used for profiling in this thesis.

TAU

TAU was used to profile the matrix code. All functions related to matrix operations are located in one source file, that is well suited for automatic profiling. Automation is advantageous due to the large number of functions. TAU was chosen because of ease of use and good visual representation of results. TAU is supposed to work with both static and dynamic linking. Static linking is used by default, and dynamic linking can be chosen through parameters. Despite several attempts, the compiler scripts never worked with static linking. No static linking made the compiler based instrumentation useless since CRAVA is statically linked for practical reasons. By digging through example code we revealed the internal TAU tools used for instrumenting. Using these tools directly in a Makefile generated valid profiling code, but the resulting application became many times slower and reported a possible error in performance data output. It turns out that the TAU profiler dumps profiling data to disk when all active timers are stopped. The compiler scripts inserts a top level timer automatically, but since we perform the instrumentation "manually", this timer is non-existent. Through testing it became apparent that this timer had to be in the same file as the rest of the profiling code. The final solution injects a top level timer into the instrumented source file, which is called from the main program at the start and end. This top level timer allowed the application to run with about 50 % increase in time use, and to dump data to disk only once.

GNU gprof

We attempted to use gprof on the matrix code. Since it is fully integrated into the GNU compiler collection, instrumenting the code was relatively easy. The instrumented application ran with only a minor performance impact. However, the output of gprof was deemed less useful than the other options, because it was at a function level granularity, which was not detailed enough at times. Another contributing factor to the low usefulness of gprof output, was the fact that the results are prone to statistical errors.

Manual instrumentation

Using detailed profiling like TAU on the whole application was not feasible due to the increase in time usage. A simple profiling framework was needed for testing during development. The framework used in this thesis was originally built by A. Hysing [28], and has been adapted slightly for this thesis. Instrumentation code is manually put in strategic places throughout the code, measuring time usage with simple timers. The timings are passed to the framework along with an identifier, and the framework creates a log of all the timers in a human readable format. This framework allows for variable granularity and detailed profiling where necessary. If for instance a function contains several for-loops, uncovering the time distribution is an easy task (albeit somewhat tiresome). The timing function used is the OpenMP wallclock timer, *omp_get_wtime()*.

Since each FFT call must pass through the settings store, it can easily be timed there. Our FFT timing was implemented in a slightly ineffective way requiring an extra linked list, and caused the detailed view of time usage for FFTW execution plans to become unstable. The detailed view being unstable means that the time usage per individual plan is not always accurate in one run, but can be statistically established over several consecutive runs. However, the total execution time used is always accurate.

4.3 Dwarfs

Two *dwarfs* are easily identified in CRAVA: *dense linear algebra* and *spectral methods*. Both are further detailed and optimized in the following sections.

4.3.1 Dense linear algebra

The dense linear algebra dwarf covers calculations with dense matrices and vectors. Dense matrix operations are usually computationally bound and can be optimized for data-level parallelism through SIMD and task-level parallelism through cache-tiling.

CRAVA uses dense matrix operations in several sections, but the use is mostly concentrated in the *Parameter filter* section. The original matrix code were simple naive implementations. NR expressed early that this was known to be an inefficient solution and it was planned to be fixed. However, it is included in this thesis because it directly classifies as a dwarf.

The original code was profiled with TAU and the top time-consuming functions are listed in Table 4.3. The top three functions account for the vast majority of time usage, and the rest of the functions use even less than the fourth. The timings relate directly to the timings in Listing 3.1, where the total time usage was 3309 seconds.

Table 4.3: Top time consuming matrix operation functions in CRAVA.

Function	**Calls**	**Time per call**	**Time total**
lib_matr_prod	26	13247 ms	344 s
lib_matrAXeqBMatR	8	21245 ms	170 s
lib_matrCholR	932	12 ms	11 s
lib_matrAxeqbR	79414	0.01 ms	0.8 s
Total			**525.8 s**

Lib_matr_prod performs matrix-matrix multiplication. This is a BLAS level 3 operation and matches the routine *dgemm* (double-precision general matrix multiply). The GEMM routine is very generic and is the basis for most of the other BLAS level 3 routines. It calculates the new value of matrix C from the matrix product of matrices A and B, and optionally the old value of C. The input matrices can be either transformed or not, and the new and old value scaled if needed (α and β). As shown in the formula:

$$C \leftarrow \alpha AB + \beta C$$

Lib_matrAXeqBMatR solves a set of linear equation systems

$$AX = B$$

for real numbers. The input matrix A is a symmetric positive definite, lower Cholesky factorized matrix. This matches the LAPACK routine *dpotrs*, which can solve

$$Ax_j = b_j$$

systems for each column j in X and B.

Lib_matrCholR performs a lower Cholesky factorization of a real symmetric positive definite matrix A.

$$A = LL^T$$

This matches the LAPACK routine *dpotrf*, which returns the matrix L.

Optimization

To optimize the dense linear algebra in CRAVA, BLAS and LAPACK libraries are used. CRAVA is coded in C/C++ and could theoretically use C++ versions of LAPACK, but converting a large application like CRAVA to a new matrix data-type would be a huge task and not considered for this thesis.

Since CRAVA is mostly run on AMD processors, the AMD Core Math Library (ACML) was chosen as the best suited library. ACML contains some FORTRAN code, which is not always straightforward to mix with C/C++ routines when the memory layout of the routines differ, as described in Section 2.5.3. ACML use the column-major order memory layout for pseudo-multidimensional arrays, which makes it incompatible with CRAVA which uses mostly dynamic multidimensional arrays. However, this incompatibility can be can be solved by transforming the matrices before and after calls to ACML. A transformation will obviously incur a time penalty, which will limit the overall optimization effect of a library.

In this thesis, two functions are used to convert a generic matrix from the CRAVA format to ACML format, and back. In reality the data is copied into a new one-dimensional array , passed to ACML and finally, copied back into the original matrix. This array conversion allows CRAVA to make full use of the BLAS and LAPACK routines of ACML.

Additionally, CRAVA is tested with a version of the Automatically Tuned Linear Algebra Software (ATLAS) [45] BLAS library, auto-tuned for the test system. ATLAS supports both column-major order and row-major order memory layout for pseudo-multidimensional arrays. However, it does not support dynamic arrays, and still requires a transformation from the CRAVA format. The transformation routines for ATLAS can use *memcpy* to speed up the data copying.

4.3.2 Spectral methods

The spectral methods dwarf embodies algorithms that numerically solve partial differential equations (PDE), where data are in the frequency domain [39]. PDEs are often expressed in continuous space and time, and can not be solved without discretization. A spectral method provides a way to translate the equations into discrete versions which can be solved numerically. Spectral discretizations can yield approximations with high accuracy, as well as a low number of grid points to achieve the desired precision, resulting in low memory footprint [39].

The Fourier series lies at the core of many spectral methods, where data is represented in the frequency domain. In fact most spectral methods require a transformation of the data for the discretization to work [39]. For the methods based on the Fourier series, a discrete Fourier transform is performed, e.g. by using the FFT. A key point of the spectral methods dwarf is that after the change of basis, the computational pattern often resembles the Structured or Unstructured Grid dwarfs [39].

CRAVA uses a spectral method in its core step, inversion. A discrete representation is used to solve the differential wave equations in the frequency domain. As mentioned above, after the transformation of basis the rest of the computation in this dwarf will be of a different nature. This section will therefore focus mostly on the FFT, and the inversion itself will be examined in more detail in Section 4.4.

CRAVA uses the library FFTW [25] for its Fourier transforms. In previous works (Hysing [28] and Stinessen [44]), the FFTW execution plans were cached and re-used, with varying success. Because FFTs classify as the spectral methods *dwarf*, a decision was made to further investigate their performance. As described in Section 4.2 the settings store is used as a centralized timer for FFTs. Sadly the timing implementation caused a few inaccuracies, but through repeated runs the performance data established are listed in Table 4.4. The timings again relate directly to the timings in Listing 3.1, where the total time usage was 3309 seconds. Clearly the 3D transforms are the most dominant in CRAVA. In fact, only 4 execution plans account for over 99 % of the transformation time (top 2 from Table 4.4 with forward and backward transforms).

Table 4.4: Details and time usage of all the Fast Fourier Transforms in CRAVA, sorted by FFTW execution plan.

Plan dimensions	**Size**	**Transforms**	**Total time**
3D	$1344 \times 972 \times 336$	17	480 s
1D	336	11.7 mill.	51 s
1D (rest)	assorted	less than 1000	0.4 s
Total			**531.4 s**

Optimization

In the author's fall specialization project [44], an attempt was made to enable threading of FFTs by allowing multiple transforms to run in simultaneously. This had minor positive effect only in some cases. However, FFTW includes multi-threaded execution routines for plans, and in this thesis they are applied to the CRAVA code. Since most of the plan executions pass through the settings store, it is the natural place to inject the parallelism. Several different setups are tested:

- Running all transforms with multi-threading
- Running only the 3D transforms with multi-threading
- Running only the 3D transforms with multi-threading, but without the simultaneous execution from the fall specialization project [44].
- Use the measuring feature of FFTW to find the best algorithm for the repeated 1D transforms.

To use the multi-threaded execution routines of FFTW a version upgrade was necessary. CRAVA was originally packed with version 2.1.2. While this version does have basic threading support, OpenMP threading was not introduced before version 2.1.4. Some experimentation showed that the OpenMP threading was slightly faster, and FFTW version 2.1.5 (bug fix) was integrated in CRAVA. It should be noted that when FFTW was upgraded, the output of CRAVA changed slightly, causing the correctness tests to report small differences. The change in output is assumed to be due to rounding differences in the floating-point arithmetic [26] in the new library version and therefore ignored. Although the 2.1.x branch of FFTW is used in CRAVA, a newer version branch 3.x exists [25], promising over 20 % performance gain over 2.1. This performance gain is accomplished through a new code structure and more SIMD vectorization. The newer version was not considered for this thesis because of its new API, which would have required even more changes in CRAVA due to its incompatibility with 2.1.

4.4 Inversion

The objectives of geophysical inversion is to estimate elastic model parameters based on general knowledge and a set of measurements. These model parameters are $V_p(\mathbf{x}, t)$, $V_s(\mathbf{x}, t)$ and $\rho(\mathbf{x}, t)$, as explained in Chapter 3, where $\mathbf{x}$ is the lateral location and t is the two-way vertical seismic travel-time. The Bayesian linearized AVO inversion method used in CRAVA is based on a weak contrast approximation [9, 12]. The approximation relates the seismic reflection coefficients to the elastic medium, and is a linearization of the Zoepprits equations [9]. The following equations are paraphrased from Buland *et al.* [8, 9, 10] and Dahle *et al.* [12]. The continuous approximation reflectivity function is given as

$$\begin{aligned} c(\mathbf{x}, t, \theta) &= a_{V_p}(\mathbf{x}, t, \theta)\frac{\partial}{\partial t}\ln V_p(\mathbf{x}, t) \\ &+ a_{V_s}(\mathbf{x}, t, \theta)\frac{\partial}{\partial t}\ln V_s(\mathbf{x}, t) \\ &+ a_{\rho}(\mathbf{x}, t, \theta)\frac{\partial}{\partial t}\ln \rho(\mathbf{x}, t) \end{aligned}$$

where θ is the reflection angle. The inversion algorithm requires that a_{V_p}, a_{V_s} and $a_\rho$are defined in a prior known background model. This unknown model parameter vector is defined as

$$\mathbf{m}(\mathbf{x}, t) = [\ln V_p(\mathbf{x}, t), \ln V_s(\mathbf{x}, t), \ln \rho(\mathbf{x}, t)]^T$$

The seismic data is represented by the convolutional model

$$d_{obs}(\mathbf{x}, t, \theta) = \int s(\tau, \theta)c(\mathbf{x}, t - \tau, \theta)d\tau + e_d(\mathbf{x}, t, \theta)$$

where s is the wavelet, c is the synthetic seismic and e_d is an angle and location dependent error term.

$\mathbf{d}_{obs}$, $\mathbf{m}$, $\mathbf{s}$, $\mathbf{w}_{obs}$ are time discretizations of the seismic data, elastic parameters, seismic wavelet and well-log data, respectively. The error term in the measured data is assumed to be a zero mean Gaussian distribution. The other discretizations are then multi-Gaussian or multi-normal distributions. $\mathbf{m}$ is for example

$$\mathbf{m}|\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m \sim \mathcal{N}_{n_m}(\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$$

where n_m is the dimension of $\mathbf{m}$, $\boldsymbol{\mu}_m$ is the expectation vector and $\boldsymbol{\Sigma}_m$ is the covariance matrix. From this it can be construed that the posterior distribution is Gaussian

$$\mathbf{m}|\mathbf{d}_{obs} \sim \mathcal{N}_{n_m}(\boldsymbol{\mu}_{m|d_{obs}}, \boldsymbol{\Sigma}_{m|d_{obs}})$$

where the posterior expectation and covariance are

$$\begin{aligned} \boldsymbol{\mu}_{m|d_{obs}} &= \boldsymbol{\mu}_m + (\mathbf{SA\Sigma}_m^{'})^T \boldsymbol{\Sigma}_{d_{obs}}^{-1}(\mathbf{d}_{obs} - \boldsymbol{\mu}_{d_{obs}}) && (4.1) \\ \boldsymbol{\Sigma}_{m|d_{obs}} &= \boldsymbol{\Sigma}_m - (\mathbf{SA\Sigma}_m^{'})^T \boldsymbol{\Sigma}_{d_{obs}}^{-1} \mathbf{SA\Sigma}_m^{'} && (4.2) \end{aligned}$$

and $\mathbf{SA\Sigma}_m^{'}$ is the covariance between $\mathbf{d}_{obs}$ and $\mathbf{m}$.

Equations 4.1 and 4.2 are dependent on the inverse covariance matrix $\boldsymbol{\Sigma}_{d_{obs}}^{-1}$. This matrix has $n_\theta^2 n^2$ elements for a volume with n cells, and calculating the inverse matrix is an operation with complexity $O(n^{2.x})$. For any reasonably sized inversion volumes this matrix inversion becomes restrictively time consuming. However, the covariance function for a homogeneously correlated spatial variable is diagonalized by a 3D Fourier transform. In the Fourier domain each frequency component can be solved independently, thus having a complexity of $O(n)$. The inversion becomes upper bound by the Fourier transform which has a complexity of $O(n \log n)$.

Optimization

The inversion algorithm in CRAVA is by definition a spectral method. A partial differential equation is spatially discretized and solved numerically. After the discretization, the inversion calculation resembles the computational pattern of the *structured grid dwarf.* No n-point stencil is used in the computation, but data from each angle stack are fetched and a series of dense matrix operations are performed on them.

Early benchmarks revealed that inversion was the number one time-consumer in CRAVA. Listing 3.1 shows that inversion accounts for over 50 % of the wallclock time used. At first an attempt was made to work with the already parallelized inversion code from Hysing [28], but as his own results shows, this code does not scale at all, and even scales negatively for some metrics. The original serial version of CRAVA support using secondary storage (e.g. hard disk) for storing temporal data if it cannot fit all in memory. This secondary storage is implemented using serial I/O to and from the data structures. Serial I/O imposes restrictions on the efficiency of a parallel version, and limits the possible parallel speedup as Amdahl's law explains. The old parallelization respected these restrictions by using a pipelined execution model. This implementation would run the computation in parallel but the surrounding I/O serially [28].

However, the common usage of CRAVA does not involve the secondary storage feature, and serial I/O is not necessary when all data resides in memory. The pipeline is therefore not an ideal solution for normal use. An attempt was made to remove the sequential restrictions in the existing parallel implementation, but the attempt had negative effect. A slowdown factor of almost 2 was observed. Tracking down the cause of this effect proved difficult, and instead it was decided to start over from scratch. The original serial inversion code was reinstated, and work began to parallelize it.

Table 4.5: Time usage in the original serial inversion code.

Part	**Time usage**	**Percent**
Inversion loop	731 s	63.5 %
Stack FFT	92 s	8.0 %
I/O time	329 s	28.5 %
Total	**1152 s**	**100 %**

The original serial code was benchmarked and the time usage is detailed in Table 4.5. These timings are comparable to the timings in Listing 3.1, if its inversion section is ignored. For comparison, CRAVA's total time usage with the original serial inversion code would have been 2707 seconds. The table shows that the inversion loop is the largest part of the inversion section, followed by I/O. The inversion loop traverses every layer in the seismic angle stacks, gradually deeper into the posterior. For every layer, joint error multipliers are computed from the wavelets. Every location in the layer use these multipliers to construct an error matrix along with the location specific error correlation. Next, the inversion is performed individually for every location. All the angle stacks and parameters are calculated in one loop traversal, as per the $O(n)$ complexity.

Table 4.6 shows the original time usage within the inversion loop in Table 4.5. The core inversion loop is the largest single time consumer, but loading and storing data from and to memory also use a considerable amount of time. The memory is accessed though serial data access methods. The first step in parallelizing inversion is to use indexed access methods. Adding indexed access methods increased time usage by about 10 %. This increase was assumed to be a result of the extra bounds checks in the indexed methods.

Table 4.6: Time usage within the inversion loop in Table 4.5.

Part	**Time usage**	**Percent**
Loading data	280 s	38.3 %
Core inversion loop	317 s	43.4 %
Creating error matrix	63 s	8.6 %
Storing data	71 s	9.7 %
Total	**731 s**	**100 %**

For a parallelization of the inversion loop, some data structures needs to be thread specific. In the old parallelization, the core inversion loop was parallelized. In order to avoid allocation and deallocation of these resources for every iteration in the main inversion loop, the thread-private feature in OpenMP 3.0 was used [28]. This approach allowed the data to be allocated and deallocated once in each thread, and live across iterations. However, the prominent OpenMP version available in CRAVA's normal operational environment is 2.5, which lacks support for thread-private data. For this thesis we decided to try a different approach. By focusing on parallelizing the main inversion loop, the data structures can be created and removed inside the OpenMP parallel section, which will work in both version 2.5 and 3.0 of OpenMP. With this approach each layer is processed in parallel, and there is less chance of data collisions between the private caches of the cores (which can lead to thrashing in the caches). Additionally, the serial fraction in computing the error multipliers poses no problem with this approach, since that too is run locally by the thread responsible for that layer. The pseudo code for the new inversion algorithm is listed in Algorithm 4.1.

With the new parallel implementation of the inversion algorithm, the time usage increased by a factor of 3. At first this was believed to be caused by cache problems, but after tedious debugging, the reason emerged as mutual exclusion (mutex) locks around the indexed data access methods. They were likely put there to ensure data consistency, but are unnecessary with the new parallelization approach. Removing the mutex locks had great effect on time usage and speedup was achieved. The locks are also the main reason why introducing indexed data access methods increased time usage.

The inversion algorithm checks every layer for relevancy before processing. If the layer is not a relevant frequency, no inversion is performed, and the prior model is simply copied to the posterior model. When all the data resides in memory the prior and posterior grids are the same data structures, and the data copying is redundant. A quick test proved that of the 336 layers in the benchmark test, only 112 (33 %) are relevant frequencies and inverted. The

Algorithm 4.1 Pseudo code for the new parallel inversion algorithm.

```
1: parallel with $n$ threads do
2:   Allocate thread data
3:   worksharing for $k = 0$ to $d_z$ do
4:     /* Iterations divided amongst threads */
5:     Compute $error\_multipliers$ from wavelet
6:     for $j = 0$ to $d_y$ do
7:       for $i = 0$ to $d_x$ do
8:         if $relevant\_frequency$ then
9:           Load $data$ from $d_{i,j,k}$
10:          Fill $error\_matrix$ for $i, j, k$ with $error\_multipliers$
11:          Solve equations with $data$ and $error\_matrix$
12:          Store new $data$ to $d_{i,j,k}$
13:        else if $using\_file\_storage$ then
14:          Load data from $d_{pre_{i,j,k}}$
15:          Store data to $d_{post_{i,j,k}}$
16:        end if
17:        Update covariances for $i, j, k$
18:      end for
19:    end for
20:  end for
21:  Remove thread data
22: end parallel
23: Inverse FFT
24: Write parameters to disk
```

algorithm was modified to not perform this copying for irrelevant frequencies. However, one special case exists when the data is located in secondary storage where the prior and posterior grids are separate data structures. In this case a data copy is required, but the processing step is unnecessary. The change to the inversion algorithm along with the special case condition is represented in Algorithm 4.1 on lines 8 to 16.

Chapter 5

Results

In this chapter we present the results of applying the optimizations described in Chapter 4. New benchmark timings are presented and compared to the original timings. Metrics for the improvements are given and visualized, along with discussion of the results.

The different sections in Chapter 4 are systematically examined before the overall effects on CRAVA are presented. Timings are from benchmarks on the test system (see Table 4.1), except where stated otherwise. The timings for the scaling-test are from benchmark tests on the *scale-test system*, detailed in Table 5.1.

Table 5.1: Scale-test system

System	**Cluster application node**
CPU	Twelve-core AMD Opteron 6168 (Magny Cours)
Frequency	1.9 GHz
Cores	48 (4 sockets $\times$ 12)
Memory	512 GB
Instruction set	amd64 (x86_64)
OS	Red Hat Enterprise Linux Server release 5.6
GCC	4.1.2
OpenMP	2.5
LSB	4.0

5.1 Dense linear algebra dwarf

After ACML was enabled on the top three matrix operations as described in Section 4.3.1, the code was profiled with TAU a second time. As expected, a large improvement was observed. The new timings are listed in Table 5.2.

Table 5.2: Time usage of the optimized matrix operations, with comparison to the original time usage.

Function	**Time per call**	**Total time new**	**Total time before**	**Speedup**
lib_matr_prod	85 ms	2.2 s	344 s	156x
lib_matrAXeqBMatR	121 ms	1 s	170 s	170x
lib_matrCholR	0.9 ms	0.8 s	11 s	13.8x
C_to_FORTRAN	2 ms	2 s	-	-
FORTRAN_TO_C	0.6 ms	0.6 s	-	-
Total		**6.6 s**	**525 s**	**79.5x**

The section of CRAVA with the most matrix operations is *Parameter filter*. Using ACML resulted in a 98.1 % reduction in run-time (53.6x speedup) on this section, as shown in Section 5.4. The large speedup has to be viewed in context with the previous version. Since a naive numerical recipes implementation was used, direct comparison with a highly optimized library is unfair. The difference in time usage is visualized in Figure 5.1.

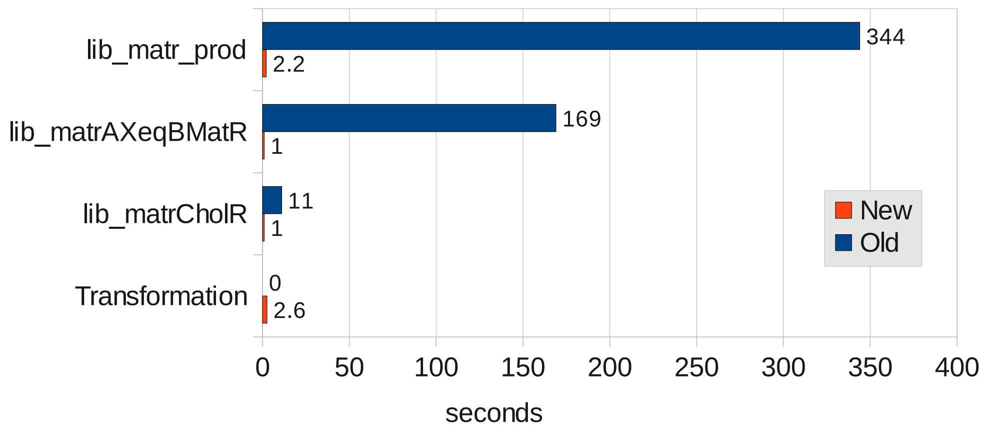

Figure 5.1: Comparison of the dense matrix time usage before and after use of ACML.

Compared to the new matrix operations, the array transformation routines use a large amount of time. However, when compared to the rest of the program, it is of insignificant size. The rest of the matrix operations in CRAVA, that was not considered for library calls, deal with very small matrices and vectors (as explained in Section 4.3.1). Most run in less than 5 microseconds, and the gains from optimization would be small. The limited amount of work for small matrices in CRAVA exemplifies that the size of the dataset is important to the parallel performance of the dense linear algebra *dwarf*. Both a serial version and multi-threaded version of ACML were tested with CRAVA. On the test system, the multi-threaded version performed slightly better than the serial, but as a scaling test (introduced in Section 5.4) uncovered, the multi-threaded version scaled negatively, and is not the best choice for the small matrix operation datasets in CRAVA.

The auto-tuned ATLAS version performed well, but overall were no better than ACML. The timings for ATLAS are listed in Table 5.3 along with a comparison to ACML. ATLAS uses more time on the actual matrix operations, but the array transformations are considerably faster due to the use of *memcpy*. The transformation routines are about 85 % faster than the routines used for ACML. Overall, the ATLAS implementation is about the same speed as the ACML implementation when including the array transformation time.

Table 5.3: Time usage of the matrix operations when ATLAS is used.

Function	**Time per call**	**Total time**	**Speedup**	$\frac{ATLAS}{ACML}$
lib_matr_prod	139 ms	3.6 s	95.6x	1.64
lib_matrAXeqBMatR	285 ms	2.3 s	73.9x	2.30
lib_matrCholR	0.6 ms	0.5 s	22x	0.63
multidim_to_pseudo	0.26 ms	0.3 s	-	0.15
pseudo_to_multidim	0.13 ms	0.1 s	-	0.17
Total		**6.8 s**	**77.2x**	**1.03**

5.2 Spectral methods dwarf

In the spectral methods *dwarf* the Fast Fourier Transforms of CRAVA were executed with multi-threading. FFTW supplies routines for threading with OpenMP. A few different setups were tested, with varying success. The 3D transforms benefited greatly from parallel execution, but most when given full access to the CPUs. From previous efforts, multiple 3D transforms were performed simultaneously. This limited the effect of the multi-threaded routines. Running multiple multi-threaded routines simultaneously would spawn more threads than available cores. Since spectral methods are limited by memory latency [4], it was speculated that the threads would end up competing for memory bandwidth or cache resources, resulting in the limited performance gains.

The smaller one-dimensional transforms did not show a large change from the multi-threaded routines. Most likely they do not parallelize completely (as noted in the documentation [24]), or are so small that the effect is negligible. Since these transforms are called several million times, we attempted to use the measuring feature of FFTW to find the best algorithm. The measuring feature increased the time used to create the plans by about 16 seconds, or 1/3 of the total plan execution time, only to have no improvement on the transformation time. For the final version only the 3D transforms are run with multi-threading, and the simultaneous transformations removed. All the transforms use the best estimated algorithm by FFTW. The new performance data is listed in Table 5.4.

Table 5.4: The new time usage of all the FFTs in CRAVA, with comparison to the original time usage.

Plan dimensions	**Size**	**Time usage new**	**Time usage before**	**Speedup**
3D	$1344 \times 972 \times 336$	56 s	480 s	8.6x
1D	336	51 s	51 s	-
1D (rest)	assorted	0.4 s	0.4 s	-
Total		**107.4 s**	**531.4 s**	**5x**

Overall the changes reduced transformation time by 80 %. The improvement is visualized in Figure 5.2. The sections of CRAVA most affected by the improvement in FFT time use are *Building of the stochastic model* and *Inversion.*

The results show that only the 3D transforms benefited from multi-threaded execution. While the one-dimensional transforms are relatively small and quick, their combined time usage can get dominating if enough transformations are performed. They are not be able to exploit parallelism available in the hardware and fall back to serial execution. These results are an indication that the *spectral methods dwarf* is not a one sided benchmark for parallel hardware and programming models. Even if the newer versions of FFTW advertise more SIMD vectorization, it would not exploit the current trend in processor architecture design of increased task level parallelism.

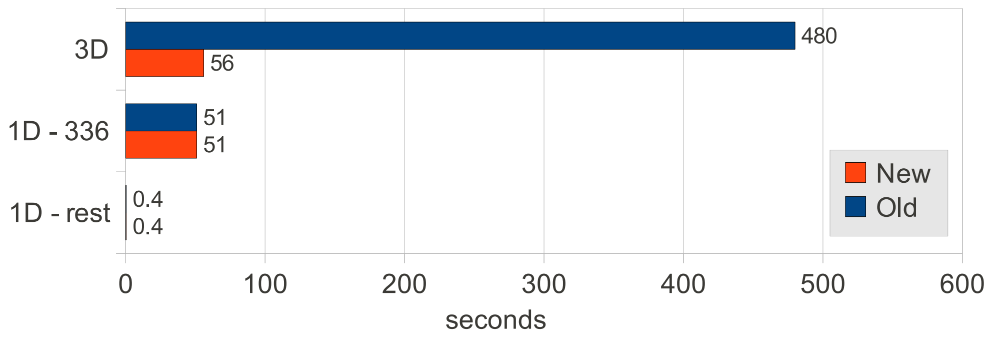

Figure 5.2: Comparison of the old and new FFT time usage in CRAVA.

5.3 Inversion

The inversion section of CRAVA is optimized through parallelization of the inversion algorithm. Additionally, the algorithm now skips data copying for irrelevant frequencies. The new time usage of the section is listed in Table 5.5.

Table 5.5: New time usage within the inversion section in CRAVA, compared to the time usage before optimization.

Part	**Time usage new**	**Percent**	**Time usage before**	**Speedup**
Inversion loop	54 s	13 %	731 s	13.5x
Stack FFT	13 s	3 %	92 s	7x
I/O time	343 s	84 %	329 s	-
Total	**410 s**	**100 %**	**1152 s**	**2.8x**

The improvement for “Stack FFT” originates from the multi-threaded Fourier transforms in Section 5.2. A 13.5x speedup is observed for the inversion loop, and the inversion section is now dominated by I/O with 84 % of the total time usage. The 14 seconds increase in I/O time is due to the network conditions and latency, and not due to our changes. The reason for the inversion loop's super linear speedup is the extra work performed in copying data for irrelevant frequencies in the original serial version. When the unnecessary data copying is dropped the inversion loop takes 336 seconds (as recorded in the scaling-test described below), resulting in a reasonable parallel speedup of 6.2x. The improvements to the inversion section are visualized in Figure 5.3.

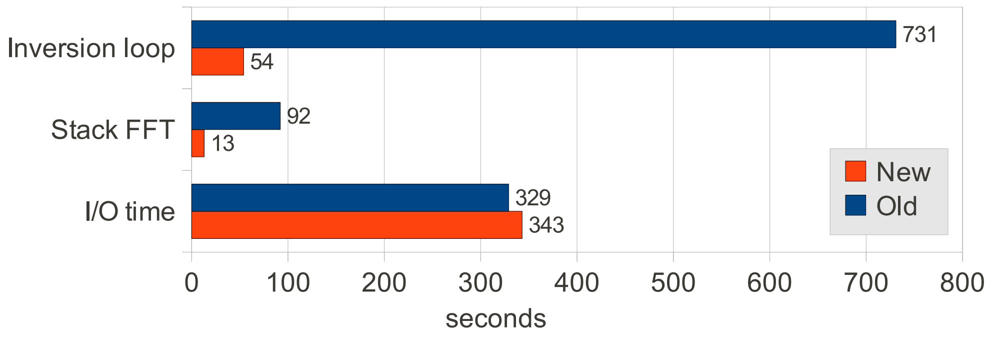

Figure 5.3: Comparison of the time usage within the inversion section of CRAVA before and after the optimization and new parallelization.

To test the parallel scalability of the new inversion algorithm, a scaling-test was performed on the *scale-test system* (detailed in Table 5.1). The optimized version of CRAVA was run with a varying number of threads, which is assumed to exactly match the number of processing cores used. The results are plotted in Figure 5.4 (detailed timings are available in Table B.1). The results show a good parallel speedup for the parallelized inversion algorithm, with near linear speedup up to around 16 cores. Above 16 cores the speedup decreases, and even turns slightly negative for about 32 or more cores. The scale-test system has 48 cores, but sadly our optimized version of CRAVA "hangs" (locks up and stops) at 40 or more cores, so the highest number of cores we are able to test is 38. From the graph we observe that the I/O time is approximately constant, and accounts for increasingly more of the total time usage in the inversion section.

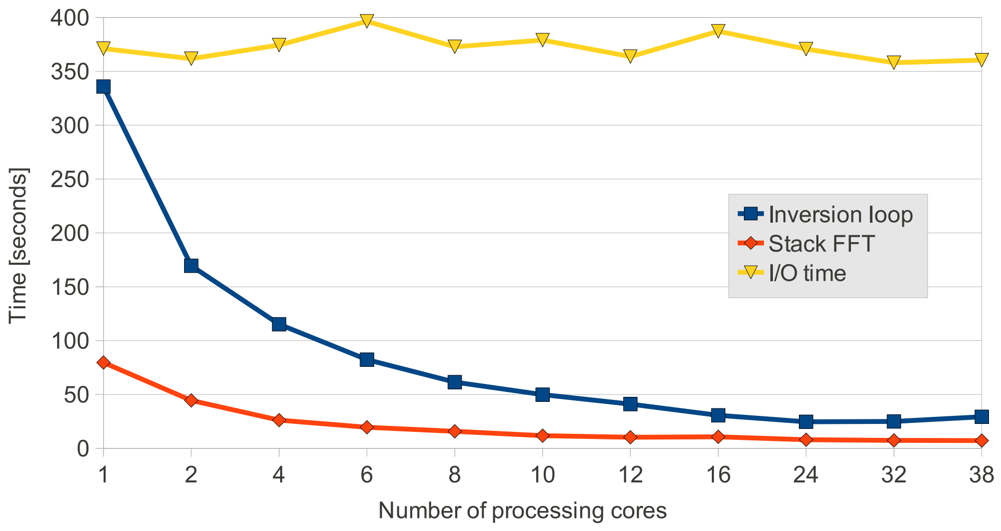

Figure 5.4: The parallel scalability of the new inversion algorithm parallelization. Inner time usage of the inversion section of CRAVA for various number of processing cores.

5.4 Overall effects

After all the optimizations in Chapter 4 were applied to the CRAVA code, another benchmark test was performed to measure the overall optimization effects. The new time usage of the sections within CRAVA are presented in Listing 5.1.

Listing 5.1: The new time usage within CRAVA after the optimizations applied in this thesis are applied.

```
Section                          CPU time                 Wall time
-----------------------------------------------------------------------
Loading seismic data       120.10     2.98 %        154.00    11.13 %
Resampling seismic data   1783.74    44.29 %        254.00    18.35 %
Wells                       11.01     0.27 %         11.00     0.79 %
Prior expection            605.22    15.03 %        135.00     9.75 %
Prior correlation            7.19     0.18 %          7.00     0.51 %
Building stochas. model    714.21    17.73 %        172.00    12.43 %
Inversion                  575.17    14.28 %        418.00    30.20 %
Parameter filter            35.09     0.87 %          8.00     0.58 %
Rest                       175.81     4.37 %        225.00    16.26 %
-----------------------------------------------------------------------
Total                     4027.56   100.00 %       1384.00   100.00 %
```

We observe that the time used within sections *Building stochastic model*, *Inversion* and *Parameter filter* has decreased from the original time usage in

Listing 3.1. These changes are presented in Table 5.6, along with the change in total wallclock time. A visualization of this table is seen in Figure 5.5.

Table 5.6: Comparison of time usage for the sections within CRAVA, that are affected by our optimization efforts, as well as the total time usage of CRAVA.

Section	**New time usage**	**Old time usage**	**Speedup**	**Improvement**
Building stochastic model	172 s	313 s	1.82x	45.05 %
Inversion	418 s	1754 s	4.20x	76.17 %
Parameter filter	8 s	429 s	53.63x	98.14 %
Total run time	1384 s	3309 s	2.39x	58.17 %

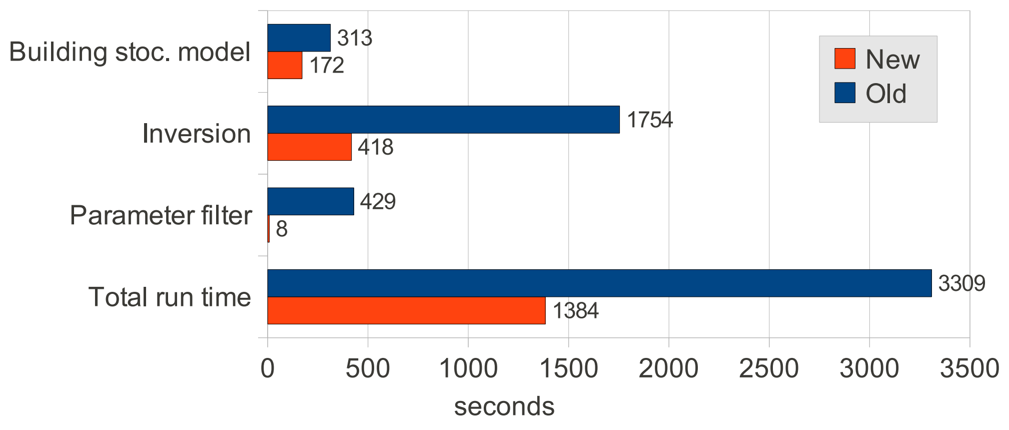

Figure 5.5: Comparison of the time usage within sections of CRAVA affected by our optimizations, as well as the total time usage of CRAVA.

It is important to note that the I/O times depend on the network conditions and are very unstable. The sections containing I/O are *Inversion*, *Rest* and *Loading seismic data*. As noted in Section 4.2, the impact of data input latency is reduced by letting the operating system cache the data in memory. A "cold run" without this cache use about 700 % (very unstable) more time loading the data (one example is found in Listing B.1).

The results from the scaling-test performed with our optimized version of CRAVA on the *scale-test system* (see Table 5.1), are plotted in Figure 5.6. The detailed timings are available in Table B.1, and a larger version of the graph is available in Figure B.1.

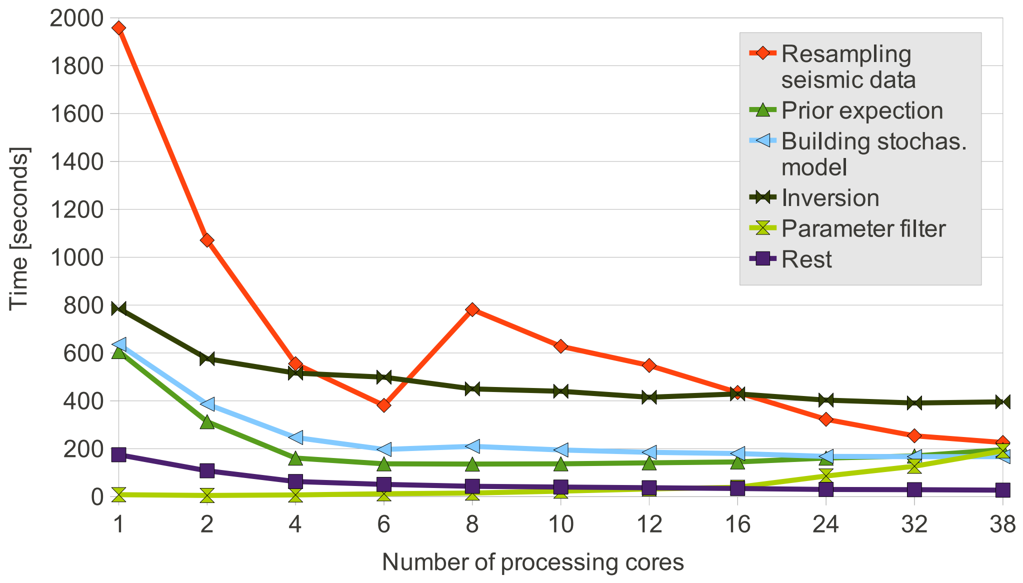

Figure 5.6: Time usage of selected sections of CRAVA in a scaling-test with various number of processing cores. Larger version in Figure B.1, and detailed timings in Table B.1.

The graph shows that for the re-sampling section, time use doubles from 6 to 8 cores. In fact, the section's time usage was observed to be very unstable after the profiling timing code was disabled. As both Listing 3.1 and Listing 5.1 show, the re-sampling time usage for 8 cores lies around 250 - 280 seconds on the *test system* (Table 4.1). A possible reason for the different time usage on the scale-test system is a different, less suited allocation of threads to cores and processor sockets.

Figure 5.6 reveals that the multi-threaded ACML routines scale negatively past 2 cores on the scale-test system. This trend is easier to see in the detailed timings in Table B.1. The results indicate that the small-sized matrices of CRAVA are not large enough to trump the overhead of thread creation and synchronization in the multi-threaded ACML routines.

Chapter 6

Conclusions and Future Work

Modern multi-core processors offer great computing power through hardware parallelism. However, for applications to exploit this parallelism they have to be either designed for or adapted to the new processor architectures. The main goal of this thesis was to further optimize a serial application for multi-core shared memory systems. The application is used for reservoir research in the oil and gas industry, and processes seismic data to produce parameters used for modeling the Earth's interior structure.

In this thesis, we optimized the seismic application CRAVA through OpenMP parallelization and use of external multi-threaded libraries. The optimization was guided by profiling and the ideas behind the *seven dwarfs* taxonomy [4]. Different profilers were tested and used on the application to locate sections of code suitable for performance optimization. We presented a way to parallelize the seismic inversion algorithm in CRAVA for OpenMP 2.5 or newer, and heavily optimized multi-threaded libraries were applied to dense linear algebra and three-dimensional Fast Fourier Transforms. Benchmarking tests were performed on modern AMD multi-core systems to measure the optimization effects on a large real-world dataset made available to us by Statoil.

6.1 Contributions

The contributions of this thesis are:

- A study of different profilers and their suitability for real-world applications like the seismic inversion tool CRAVA.
- A case study for the applicability of the *seven dwarfs* taxonomy and proposed benchmarks [4]. We observe from our results how the size of the datasets are of great importance to the parallel efficiency of the dwarfs on current computer systems, and that the small datasets can limit the effectiveness of optimizing architectures for the dwarfs.
- A way to parallelize the linearized AVO seismic inversion algorithm used in the CRAVA application, as well as a way to skip considerable amounts of irrelevant data copying within the algorithm.
- A performance improvement of about 60 % for the CRAVA application on a dual quad-core shared memory system.

6.2 Conclusions

For the CRAVA application we found that suitable profilers for testing sections during development were simple manual instrumentation frameworks. Additionally, for some instances where many function calls are grouped and examined together, larger automated profiling frameworks with selective profiling are useful. Important for all are that they do not impose a large overhead or excessively impact the timings.

In CRAVA we found the two *dwarfs* dense linear algebra and spectral methods. Furthermore, the spectral method dwarf's computation in the Fourier domain resembled the computational pattern of the structured grid dwarf. We demonstrated that CRAVA operate with data of very different sizes within these patterns, from large to small. Their percentage of total time usage were less than $1/3$, and our results showed small to no gains from parallelization for the smallest data, raising the question of real-life effectiveness from using of these dwarfs as parallel benchmarks.

Our results showed that seismic processing applications, such as the CRAVA seismic inversion code, can benefit greatly from multi-core optimization efforts. Through serial optimizations and parallelization, we managed to achieve a 13.5x speedup for the inversion algorithm that scales up to 24 cores, with almost linear speedup up to 16 cores, on a quad twelve-core shared memory system. By using multi-threaded libraries, we observed a mildly super linear speedup on 3D Fast Fourier Transforms, and small to no speedup for small 1D transforms. Due to small matrices, multi-threaded linear algebra library routines contributed to minimal speedup compared to the serial routines, and even scaled negatively for large number of processors. Overall, the optimization efforts resulted, as mentioned, in a performance increase of about 60 % on a dual quad-core shared memory system.

6.3 Future work

In this section, we present suggestions and ideas for future work, starting with recommendations for the CRAVA application:

- The version of CRAVA in this thesis, as well as previous work, have been distanced from the development and production version. Porting the optimizations into the latest version should be prioritized to better utilize the parallelism available in modern hardware.
- To achieve greater portability, a dynamically linked version of ATLAS, which can be tuned for each system, is recommended for CRAVA instead of architecture specific libraries like ACML.
- Unless there is a need to support tape drives or other media incapable of random data access, it will probably be beneficial to investigate the possibility of removing serial access methods for data structures and instead rely purely on indexed methods.
- Our results show that I/O is now a dominating part of CRAVA. This can partly be attributed to slow network storage, but a further analysis into disk access could be beneficial. Blocking, memory mapping and incremental continuous output are possible techniques to investigate. Another possibility worth exploring is parallel I/O through e.g. MPI-2 parallel I/O routines.
- As the current time usage of CRAVA is decreasing, it would be interesting to see how the optimizations scale with increasing dataset sizes. This investigation should involve a load-balancing analysis.

- Although not suited for everyday use, running CRAVA through an automated full-program profiler like Callgrind is probably a good idea before any further optimizations, to see how sections of code compares to others.

- The remaining optional stages of CRAVA, which have not been optimized yet, are good candidates for further performance increases. Calculation of *facies probabilities* and *kriging to wells*, for example, are large sections that can be examined. Their time usage is now very large compared to the optimized sections as seen in Listing B.2.

- Through our optimization efforts it became apparent that the inversion algorithm can be decomposed for a distributed memory cluster system. Distributed implementations of the 3D FFT already exists. Investigating the feasibility for distributing the remaining sections of CRAVA for cluster systems could be interesting as a way to increase the amount of memory available.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.

[2] Angerson, Bai, Dongarra, Greenbaum, McKenney, Du Croz, Hammarling, Demmel, Bischof, and Sorensen. LAPACK: A portable linear algebra library for high-performance computers. *SC Conference*, 0:2–11, 1990.

[3] Statoil ASA. Seismic inversion. `http://www.statoil.com/en/technologyinnovation/optimizingreservoirrecovery/imagingandmonitoring/seismicinversion/pages/default.aspx`. [Online; accessed Feb. 28 2011].

[4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[5] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404 –418, 2009.

[6] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[7] Randal E. Bryant. Data-intensive supercomputing: The case for DISC. Technical report, School of Computer Science, Carnegie Mellon University, 2007.

[8] Arild Buland, Odd Kolbjornsen, and Henning Omre. Rapid spatially coupled AVO inversion in the Fourier domain. *Geophysics*, 68(3):824–836, 2003.

[9] Arild Buland and Henning Omre. Bayesian linearized AVO inversion. *Geophysics*, 68(1):185–198, 2003.

[10] Arild Buland and Henning Omre. Joint AVO inversion, wavelet estimation and noise-level estimation using a spatially coupled hierarchical bayesian model. *Geophysical Prospecting*, 51(6):531–550, 2003.

[11] Norwegian Computing Center. Elastic inversion of seismic amplitudes - the CRAVA project. `http://www.nr.no/pages/sand/area_res_char_crava`. [Online; accessed Mar. 1 2011].

[12] Pål Dahle, Bjørn Fjellvoll, Ragnar Hauge, Odd Kolbjørnsen, Anne Randi Syversveen, and Marit Ulvmoen. *CRAVA User Manual*. Norwegian Computing Center, 0.9.8 edition, March 2009.

[13] Pål Dahle, Ragnar Hauge, Odd Kolbjørnsen, Ernesto Della Rossa, Fabio Luoni, and Alfonso Junio Marini. Geostatistical AVO inversion on a deep-water oil field. In *Petroleum Geostatistics 2007, Cascais, Portugal*. EAGE, Sept. 2007.

[14] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12. IEEE Press, 2008.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[16] Valgrind Developers. *Valgrind User Manual*, 3.6.0 edition, Oct. 2010.

[17] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16:1–17, March 1990.

[18] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14:1–17, March 1988.

[19] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming.

[20] Timothy G. Mattson et al. The 48-core SCC processor: the programmer's view. In *Supercomputing 2010.* IEEE, 2010.

[21] Jan Fenlason and Richard Stallman. *GNU gprof - The GNU Profiler.* Free Software Foundation, 1988. `http://www.skyfree.org/linux/references/gprof.pdf`.

[22] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, 1972.

[23] M. Frigo and S.G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381 –1384 vol.3, May 1998.

[24] M. Frigo and S.G. Johnson. *FFTW User's Manual*, 2.1.5 edition, Mar. 2003. `http://www.fftw.org/fftw2.pdf`.

[25] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216 –231, feb. 2005.

[26] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23:5–48, March 1991.

[27] Sam Hocevar. HOWTO: using gprof with multithreaded applications. `http://sam.zoy.org/writings/programming/gprof.html`, Dec 2004. [Online; last accessed May 4 2011].

[28] Andreas D. Hysing. Parallel Seismic Inversion for Shared Memory Systems. Master's thesis, HPC-Lab, NTNU, Aug. 2010.

[29] S. Kamil, Cy Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[30] Donald E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[31] Odd Kolbjørnsen, Ragnar Hauge, Anne Randi Syversveen, Bjørn Fjellvoll, and Pål Dahle. *IProgram documentation for CRAVA*. Norwegian Computing Center, Nov. 6th 2008.

[32] Christine E. Krohn. Seismic data acquisition. In David Havelock, Sonoko Kuwano, and Michael Vorländer, editors, *Handbook of Signal Processing in Acoustics*, pages 1545–1558. Springer New York, 2009.

[33] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.*, 5:308–323, September 1979.

[34] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460. ACM, 2010.

[35] M.D. McCool. Scalable programming models for massively multicore processors. *Proceedings of the IEEE*, 96(5):816 –831, May 2008.

[36] Peter Mora. Seismic inversion using fine grain parallel computers. *Stanford Exploration Program (SEP) 56*, pages 241–254, 1987.

[37] David C. Mosher, Vl B, Peter G. Simpkin, Lower Sackville, and Bc S. Status and trends of marine high-resolution seismic reflection profiling: Data acquisition, 1999.

[38] Aaftab Munshi. *The OpenCL Specification version 1.1*. Khronos OpenCL Working Group, 2010. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.

[39] Mark Murphy. Spectral methods. `http://parlab.eecs.berkeley.edu/wiki/_media/patterns/paraplop_g3_2.pdf`, May 2009. [Online; accessed May 20 2011].

[40] NVIDIA. *NVIDIA CUDA C Programming Guide Version 3.2*, Sep. 2010. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf` [Online; accessed May 10 2011].

[41] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 29 2008-oct. 1 2008.

[42] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, Summer 2006.

[43] Angela C. Sodan, Jacob Machina, Arash Deshmeh, Kevin Macnaughton, and Bryan Esbaugh. Parallelism via multithreaded and multicore cpus. *Computer*, 43:24–32, 2010.

[44] Bent Ove Stinessen. Profiling and Optimizing a Seismic Inversion Tool. Fall master/specialization project, HPC-Lab, NTNU, Jan. 2011.

[45] R.C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2):3 –35, 2001.

[46] Wikipedia. File:amdahlslaw.svg — wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg`. (Creative Commons) [Online; accessed May 10 2011].

[47] Wikipedia. File:fork join.svg — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=File:Fork_join.svg&oldid=339669228`. (Creative Commons) [Online; accessed May 11 2011].

[48] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Prentice-Hall, Inc., 2004.

Appendix A

Tests

This appendix present details of the various tests used in this thesis.

A.1 Correctness tests

The details of the tests used to verify correctness throughout development are listed in Table A.1. Note the small grid sizes.

Test	Angle stacks	Wells	Wavelets	Mode	Grid	Options
1	0°, 10°	0	2	forward mod.	$15 \times 15 \times 500$	
2	16°, 28°	1	2	prediction	$15 \times 15 \times 250$	
3	16°, 28°	1	2	prediction	$15 \times 15 \times 250$	Kriging BG from file
4	16°, 28°	1	2	simulation	$15 \times 15 \times 250$	1 realization Kriging
5	16°, 28°	4	0	prediction	$112 \times 112 \times 280$	
6	16°, 28°	4	2	prediction	$112 \times 112 \times 250$	Kriging BG from file
7	16°, 28°	4	2	prediction	$112 \times 112 \times 375$	Correlation dir.
8	16°, 28°	4	2	prediction	$112 \times 112 \times 250$	BG from file
9	0°, 20°	1	2	prediction	$75 \times 75 \times 336$	Facies prob. BG from file
10	16°, 28°	1	2	prediction	$15 \times 15 \times 250$	

Table A.1: Correctness tests.

A.2 Benchmark test

The details of the benchmark test and dataset are listed in Table A.2. Note the large memory requirement.

Mode	prediction
Angle stacks	$10°$, $20°$, $30°$
Seismic format	SEG Y
Wavelets	estimate (3)
Surface format	STORM
Grid dimensions	$1344 \times 972 \times 336$
Padding ratio	$0.03 \times 0.02 \times 0.12$
Wells	8
Well format	RMS
Facies	sand/shale
Data size	8.7 GB
Memory needed	23.9 GB
Output format	STORM

Table A.2: Benchmark test.

Appendix B

Additional Timings

This appendix present additional timings and detailed results.

B.1 Scaling-test

The detailed timings from the scaling-test on a 48-core shared memory system (Table 5.1) are listed in Table B.1, and visualized in Figure B.1, which is a larger version of the graph in Figure 5.6. Note how some timings increase in time usage as the number of cores increase.

Table B.1: Detailed timings from the scaling-test. All the timings are in seconds.

Section \ Cores	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**16**	**24**	**32**	**38**
Loading seismic data	90	97	97	97	96	95	94	95	98	94	96
Resampling seismic data	1958	1071	555	381	781	628	548	435	323	254	226
Wells	13	13	13	13	13	13	13	13	13	13	13
Prior expection	606	313	161	137	136	137	141	145	161	170	196
Prior correlation	8	9	9	8	9	8	9	9	8	8	9
Building stochas. model	636	386	247	197	210	195	185	180	168	167	167
Inversion	*786*	*576*	*516*	*499*	*450*	*440*	*415*	*429*	*403*	*391*	*396*
Parameter filter	8	5	7	12	15	23	32	39	86	127	191
Rest	175	108	63	51	43	40	37	34	30	29	27

Time usage within inversion section

Inner section \ Cores	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**16**	**24**	**32**	**38**
Inversion loop	335.9	169.5	115.2	82.4	61.5	49.9	41.1	30.7	24.7	25	29.3
Stack FFT	79.7	44.5	26.2	19.6	15.8	11.8	10.3	10.7	8	7.4	7.2
I/O time	371.1	361.7	374.4	396.4	372.8	379	363.7	387.2	370.7	358	360.4
Inversion total	*786.7*	*575.7*	*515.8*	*498.4*	*450.1*	*440.7*	*415.1*	*428.6*	*403.4*	*390.4*	*396.9*

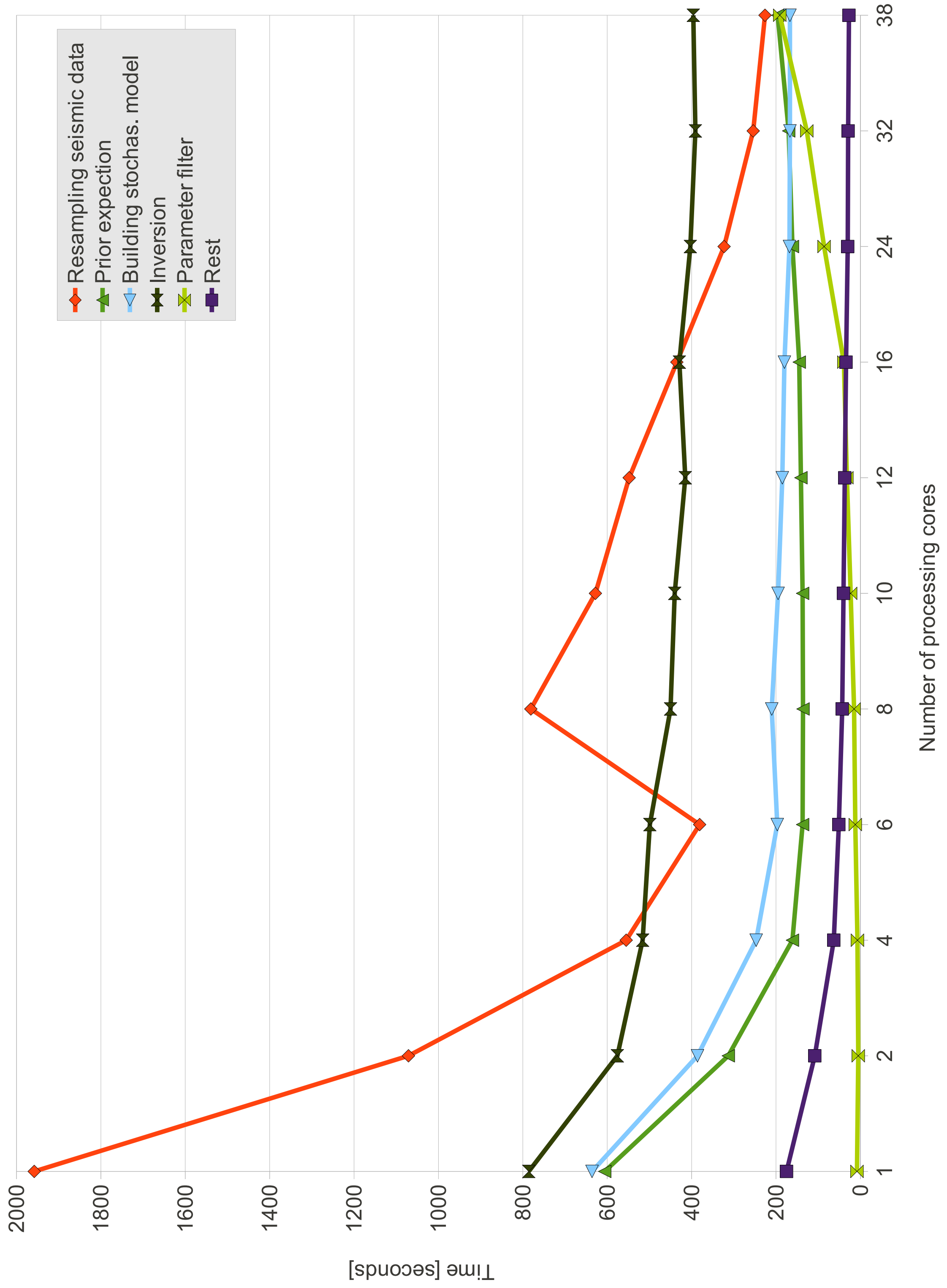

Figure B.1: Time usage of selected sections of CRAVA in a scaling-test with various number of processing cores.

B.2 Cold-run

Timings from our optimized version of CRAVA on a "cold run", meaning no data is cached in memory by the operating system. Note the large time usage of sections *Loading seismic data* and *Rest*, compared to runs where the data is cached in memory (e.g. Listing 5.1).

Listing B.1: The new time usage within CRAVA on a "cold run", or without any data cached in memory by the operating system.

```
Section                             CPU time                 Wall time
------------------------------------------------------------------------
Loading seismic data        131.15     2.46 %        1286.00    39.83 %
Resampling seismic data    3085.55    57.97 %         514.00    15.92 %
Wells                        11.16     0.21 %          11.00     0.34 %
Prior expection             598.37    11.24 %         135.00     4.18 %
Prior correlation             7.20     0.14 %           8.00     0.25 %
Building stochas. model     709.92    13.34 %         166.00     5.14 %
Inversion                   560.34    10.53 %         414.00    12.82 %
Parameter filter             35.56     0.67 %           9.00     0.28 %
Rest                        183.56     3.45 %         686.00    21.24 %
------------------------------------------------------------------------
Total                      5322.83   100.00 %        3229.00   100.00 %
```

B.3 Facies probabilities and kriging

Timings from our optimized version of CRAVA with the unoptimized *Facies probabilities* and *Kriging* features enabled are listed in Listing B.2. Note the large time usage of these sections compared to the others.

Listing B.2: The time usage within the optimized version of CRAVA with "Facies probabilities" and "Kriging" features enabled.

```
Section                            CPU time                   Wall time
-------------------------------------------------------------------------
Loading seismic data        120.72      1.29 %        120.00      2.78 %
Resampling seismic data    1785.94     19.09 %        253.00      5.86 %
Wells                        10.97      0.12 %         11.00      0.25 %
Prior expection             608.33      6.50 %        136.00      3.15 %
Prior correlation             7.17      0.08 %          7.00      0.16 %
Building stochas. model     747.94      7.99 %        200.00      4.63 %
Parameter filter             34.89      0.37 %          8.00      0.19 %
Facies probabilities       1006.59     10.76 %       1105.00     25.57 %
Kriging                    4069.85     43.49 %       1079.00     24.97 %
Rest                       4453.99     47.60 %       2069.00     47.88 %
-------------------------------------------------------------------------
Total                      9357.49    100.00 %       4321.00    100.00 %
```

Appendix C

Relevant Source Code

This appendix lists relevant source code from the thesis. First, the code for array transformation routines, then the inversion code.

C.1 Array transformation

The array transformation routines for ACML is listed in Listing C.1, and the routines for ATLAS is listed in Listing C.2.

Listing C.1: Array transformation routines for ACML, converting CRAVA dynamic arrays to FORTRAN pseudo-multidimensional format.

```
double * C_to_FORTRAN(int rows, int columns, double **Cmat, double *Fmat)
{
  /* Convert C style matrixes to FORTRAN 77 (Row-major to Column-major order)*/
  int i,j;
  if (Fmat == NULL)
  {
    Fmat = malloc(rows * columns * sizeof(double));
  }
  for (j = 0; j < columns; j++)
    for (i = 0; i < rows; i++)
      Fmat[j*rows + i] = Cmat[i][j];

  return Fmat;
}

double ** FORTRAN_to_C(int rows, int columns, double *Fmat, double **Cmat)
{
  /* Convert FORTRAN 77 style matrixes to C (Column-major to Row-major order)*/
  int i,j;
  if (Cmat == NULL)
  {
    // allocation
    return NULL;
  }
  for (j = 0; j < columns; j++)
    for (i = 0; i < rows; i++)
      Cmat[i][j] = Fmat[j*rows + i];

  return Cmat;
}
```

Listing C.2: Array transformation routines for ATLAS, converting CRAVA dynamic arrays to C pseudo-multidimensional arrays.

```c
/* Function to convert true multidimensional array of arrays to pseudo multidimensional
    (1d) arrays. */
/* Allocates output matrix if not supplied */
double *multidim_to_pseudo_array(int rows, int columns, double **multidim, double *
    pseudo)
{
  int i;
  if (pseudo == NULL)
  {
    pseudo = malloc(rows * columns * sizeof(double));
  }

  for (i = 0; i < rows; i++)
    memcpy(&pseudo[i*columns], multidim[i], columns*sizeof(double));

  return pseudo;
}
/* Function to convert pseudo multidimensional (1d) arrays to true multidimensional
    array of arrays. */
/* Allocates output matrix if not supplied (not implemented yet.) */
double **pseudo_to_multidim_array(int rows, int columns, double *pseudo, double **
    multidim)
{
  int i;
  if (multidim == NULL)
  {
    // allocation
    return NULL;
  }

  for (i = 0; i < rows; i++)
    memcpy(multidim[i], &pseudo[i*columns], columns*sizeof(double));

  return multidim;
}
```

C.2 Inversion code

The original inversion code is listed in Listing C.3, and our optimized version is listed in Listing C.4. Contact NR [1] (SAND dept.) for full source code.

Listing C.3: Original inversion code. Snippet from crava.cpp.

```cpp
static void fillErrorMatrix(float wnc, const double** errThetaCov, double scale, const
    fftw_complex* errMult1, const fftw_complex* errMult2, const fftw_complex* errMult3,
    int matrixSize, fftw_complex** errVar){
  for(int l = 0; l < matrixSize; l++){
    for(int m = 0; m < matrixSize; m++){       // Note we multiply kWNorm[l] and comp.
        conj(kWNorm[m]) hence the + and not a minus as in pure multiplication
      errVar[l][m].re = static_cast<float>(
          0.5f*(1.0f-wnc)*errThetaCov[l][m]*scale*( errMult1[l].re* errMult1[m].re +
              errMult1[l].im* errMult1[m].im) +
          0.5f*(1.0f-wnc)*errThetaCov[l][m]*scale*( errMult2[l].re* errMult2[m].re +
              errMult2[l].im* errMult2[m].im));
    }
  }
  for(int l = 0; l < matrixSize; l++){
    errVar[l][l].re += static_cast<float>(wnc*errThetaCov[l][l] * errMult3[l].re *
        errMult3[l].re);
    errVar[l][l].im  = 0.0f;
  }
  for(int l = 0; l < matrixSize; l++){
    for(int m = l+1; m < matrixSize; m++){
      errVar[l][m].im = static_cast<float>(
          0.5f*(1.0f-wnc)*(errThetaCov[l][m]*scale)*(-errMult1[l].re*errMult1[m].im +
              errMult1[l].im*errMult1[m].re) +
```

[1] http://www.nr.no

```
          0.5f*(1.0f-wnc)*(errThetaCov[l][m]*scale)*(-errMult2[l].re*errMult2[m].im + errMult2[l].im*errMult2[m].re));
    }
  }
  for(int l = 0; l < matrixSize; l++){
    for(int m = 0; m < l; m++){
      errVar[l][m].im = static_cast<float>(
          0.5f*(1.0f-wnc)*(errThetaCov[l][m]*scale)*(-errMult1[l].re*errMult1[m].im + errMult1[l].im*errMult1[m].re) +
          0.5f*(1.0f-wnc)*(errThetaCov[l][m]*scale)*(-errMult2[l].re*errMult2[m].im + errMult2[l].im*errMult2[m].re));
    }
  }
}
#define PROCESS_DATA(TID) \
        double ijkErrLamRe = static_cast<float>(fabs(errCorrUnsmoothVal.re)); \
        fillErrorMatrix(wnc_, const_cast<const double**>(errThetaCov_), ijkErrLamRe, errMult1, errMult2, errMult3, ntheta_, errVar); \
        lib_matrProdCpx(K, parVar2 , ntheta_, 3 ,3, KS); \
        lib_matrProdAdjointCpx(KS, K, ntheta_, 3 ,ntheta_, margVar); \
        lib_matrAddMatCpx(errVar, ntheta_,ntheta_, margVar); \
        if(lib_matrCholCpx(ntheta_,margVar) == 0){ \
          lib_matrAdjoint(KS,ntheta_,3,KScc); \
          lib_matrAXeqBMatCpx(ntheta_, margVar, KS, 3); \
          lib_matrProdCpx(KScc,KS,3,ntheta_,3,reduceVar); \
          lib_matrSubtMatCpx(reduceVar, 3, 3, parVar2); \
          lib_matrProdMatVecCpx(K,ijkMean2, ntheta_, 3, ijkDataMean); \
          for(i = 0; i < ntheta_; i++){ \
            ijkRes2[i].re = ijkRes[i].re; \
            ijkRes2[i].im = ijkRes[i].im; \
            ijkRes3[i].re = ijkRes[i].re; \
            ijkRes3[i].im = ijkRes[i].im; \
          } \
          lib_matrSubtVecCpx(ijkDataMean, ntheta_, ijkRes2); \
          lib_matrProdAdjointMatVecCpx(KS,ijkRes2, 3, ntheta_, ijkAns); \
          lib_matrAddVecCpx(ijkAns, 3,ijkMean2); \
          lib_matrProdMatVecCpx(K,ijkMean2, ntheta_, 3, ijkRes2); \
          lib_matrSubtVecCpx(ijkRes2, ntheta_, ijkRes3); \
        }

int Crava::computePostMeanResidAndFFTCov()
{
  // This method is globally blocking.
  // Two independant calls for computePostMeanResidAndFFTCov() from two (in)dependant independant instances will execute in serial.
  //
  // computePostMeanResidAndFFTCov() is designed this way because openMP requires threadprivate variables to be static.
  // computePostMeanResidAndFFTCov() exploints threadprivate to be able to perform expensive calls once.
  #ifdef PROFILING
  double wtime = omp_get_wtime();
  double ptimeAccum = 0.0;
  double invLoop = 0.0;
  double readTimeAccum = 0.0;
  double writeTimeAccum = 0.0;
  #endif

  omp_set_lock(&lock);
  Utils::writeHeader("Posterior␣model␣/␣Performing␣Inversion");
  if(seisData_ == NULL) return 1;
  double wall=0.0, cpu=0.0;
  TimeKit::getTime(wall,cpu);
  int i,j,k,l;
  const float lowCut = lowCut_;
  const double simboxMinRelThick = simbox_->getMinRelThick();
  const float highCut = highCut_;
  const double lz = simbox_->getlz();
  const int ntheta = ntheta_;
  const int nzp = nzp_;
  const int nz = nz_;
  const int cnxp  = nxp_/2+1;
  const int nyp_cnxp = nyp_*cnxp;
  const float delta = static_cast<float>((nz*1000.0f)/(lz*nzp));
  const float monitorSize = std::max(1.0f, static_cast<float>(nzp_)*0.02f);
  float nextMonitor = monitorSize;

  Wavelet * diff1Operator = new Wavelet(Wavelet::FIRSTORDERFORWARDDIFF,nz_,nzp_);
  Wavelet * diff2Operator = new Wavelet(diff1Operator,Wavelet::FIRSTORDERBACKWARDDIFF);
  Wavelet * diff3Operator = new Wavelet(diff2Operator,Wavelet::FIRSTORDERCENTRALDIFF);

  diff1Operator->fft1DInPlace();
  delete diff2Operator;
  diff3Operator->fft1DInPlace();

  Wavelet ** errorSmooth  = new Wavelet*[ntheta];
  Wavelet ** errorSmooth2 = new Wavelet*[ntheta];
  Wavelet ** errorSmooth3 = new Wavelet*[ntheta];

  for(l = 0; l < ntheta ; l++){
    std::string angle = NRLib::ToString(thetaDeg_[l], 1);
    std::string fileName;
    seisData_[l]->setAccessMode(FFTGrid::READANDWRITE);
    if (seisWavelet_[0]->getDim() == 1) {
      errorSmooth[l]  = new Wavelet(seisWavelet_[l],Wavelet::FIRSTORDERFORWARDDIFF);
      errorSmooth2[l] = new Wavelet(errorSmooth[l], Wavelet::FIRSTORDERBACKWARDDIFF);
      errorSmooth3[l] = new Wavelet(errorSmooth2[l],Wavelet::FIRSTORDERCENTRALDIFF);
      fileName = std::string("ErrorSmooth_") + angle + IO::SuffixGeneralData();
      errorSmooth3[l]->printToFile(fileName);
      errorSmooth3[l]->fft1DInPlace();

      fileName = IO::PrefixWavelet() + angle + IO::SuffixGeneralData();
      seisWavelet_[l]->printToFile(fileName);
      seisWavelet_[l]->fft1DInPlace();

      fileName = std::string("FourierWavelet_") + angle + IO::SuffixGeneralData();
      seisWavelet_[l]->printToFile(fileName);
      delete errorSmooth[l];
      delete errorSmooth2[l];
    }
  }
  delete [] errorSmooth;
  delete [] errorSmooth2;

  meanAlpha_->setAccessMode(FFTGrid::READANDWRITE);  //   Note
  meanBeta_ ->setAccessMode(FFTGrid::READANDWRITE);  //   the top five are over written
  meanRho_  ->setAccessMode(FFTGrid::READANDWRITE);  //   does not have the initial meaning.

  FFTGrid * parSpatialCorr      = correlations_->getPostCovAlpha(); // NB! Note double usage of postCovAlpha
  FFTGrid * errCorrUnsmooth     = correlations_->getPostCovBeta();  // NB! Note double usage of postCovBeta
  FFTGrid * postCovAlpha        = correlations_->getPostCovAlpha();
  FFTGrid * postCovBeta         = correlations_->getPostCovBeta();
  FFTGrid * postCovRho          = correlations_->getPostCovRho();
  FFTGrid * postCrCovAlphaBeta  = correlations_->getPostCrCovAlphaBeta();
  FFTGrid * postCrCovAlphaRho   = correlations_->getPostCrCovAlphaRho();
  FFTGrid * postCrCovBetaRho    = correlations_->getPostCrCovBetaRho();
  parSpatialCorr     ->setAccessMode(FFTGrid::READANDWRITE);  //   after the prosessing
  errCorrUnsmooth    ->setAccessMode(FFTGrid::READANDWRITE);  //
  postCovRho         ->setAccessMode(FFTGrid::WRITE);
  postCrCovAlphaBeta->setAccessMode(FFTGrid::WRITE);
  postCrCovAlphaRho  ->setAccessMode(FFTGrid::WRITE);
  postCrCovBetaRho   ->setAccessMode(FFTGrid::WRITE);

  LogKit::LogFormatted(LogKit::LOW,"\nBuilding␣posterior␣distribution:");
  LogKit::LogMessage(LogKit::HIGH, "\n␣␣0%␣␣␣␣␣␣␣20%␣␣␣␣␣␣␣40%␣␣␣␣␣␣␣60%␣␣␣␣␣␣␣80%␣␣␣␣␣␣100%␣\
␣␣␣␣␣␣\n␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣\
␣␣␣␣␣␣\n␣␣^");

  fftw_complex * errMult1 = new fftw_complex[ntheta];
  fftw_complex * errMult2 = new fftw_complex[ntheta];
  fftw_complex * errMult3 = new fftw_complex[ntheta];
  fftw_complex** K = new fftw_complex*[ntheta];
  for(int iter = 0; iter < ntheta; iter++){
    K[iter] = new fftw_complex[3];
  }

  // Memory is allocated once per thread which means that each thread thread has heir own memory area.
  // Noticed that all these variables are marked threadprivate and therefor store data between parallel
  // blocks
#if _OPENMP >= 200805
#pragma omp parallel
    {
#endif
        reduceVar = new fftw_complex*[3];
        errVar = new fftw_complex*[ntheta];
        ijkAns = new fftw_complex[3];
        ijkDataMean = new fftw_complex[ntheta];
        ijkMean = new fftw_complex[3];
        ijkMean2 = new fftw_complex[3];
        ijkRes = new fftw_complex[ntheta];
```

```
       ijkRes2 = new fftw_complex[ntheta];
       KScc = new fftw_complex*[3]; // cc - complex conjugate (and transposed)
       KS = new fftw_complex*[ntheta];
       margVar = new fftw_complex*[ntheta];
       parVar2 = new fftw_complex*[3];
       parVar = new fftw_complex*[3];
       ijkRes3 = new fftw_complex[ntheta];
       for(int iter = 0; iter < ntheta; iter++){
         errVar[iter] = new fftw_complex[ntheta];
         KS[iter] = new fftw_complex[3];
         margVar[iter] = new fftw_complex[ntheta];
       }
       for(int iter = 0; iter < 3; iter++){
         reduceVar[iter]= new fftw_complex[3];
         KScc[iter] = new fftw_complex[ntheta];
         parVar2[iter] = new fftw_complex[3];
         parVar[iter] = new fftw_complex[3];
       }
#if _OPENMP >= 200805
#pragma omp single copyprivate(parVar)
#endif
       {
         for(int iter = 0; iter < 3; iter++){
           for(int iter2 = 0; iter2 < 3; iter2++){
             parVar[iter][iter2].re = parPointCov_[iter][iter2];
             parVar[iter][iter2].im = 0.0;
           }
         }
       }
#if _OPENMP >= 200805
   }
#endif

    // Each thread performs a small part of the serial code and stores their part of the result.
    // Explicitly specifying the same scheduler for all parallel blocks makes the result of parallel blocks
    // to be shared between parallel regions.
    //
    // Note:
    // * The default scheduler is implementation dependant.
    // * schedule(static, 1) == round robin. thread 0, 1, 2, 3, ...
    //

#ifdef PROFILING
  invLoop = omp_get_wtime();
#endif
  for(k = 0; k < nzp; k++){
    fftw_complex kD = diff1Operator->getCAmp(k);                   // defines content of kD
    if (seisWavelet_[0]->getDim() == 1) { //1D-wavelet
      if( simbox_->getIsConstantThick() == true)
      {
        // defines content of K=WDA
        fillkW(k,errMult1);                                        // errMult1 used as dummy
        lib_matrProdScalVecCpx(kD, errMult1, ntheta);              // errMult1 used as dummy
        lib_matrProdDiagCpxR(errMult1, A_, ntheta, 3, K);          // defines content of (WDA) K

        // defines error-term multipliers
        fillkWNorm(k,errMult1,seisWavelet_);                       // defines input of  (kWNorm) errMult1
        fillkWNorm(k,errMult2,errorSmooth3);                       // defines input of  (kWD3Norm) errMult2
        lib_matrFillOnesVecCpx(errMult3,ntheta);                   // defines content of errMult3
 //simbox_->getIsConstantThick() == false
      }else{
        fftw_complex kD3 = diff3Operator->getCAmp(k);              // defines  kD3

        // defines content of K = DA
        lib_matrFillValueVecCpx(kD, errMult1, ntheta);             // errMult1 used as dummy
        lib_matrProdDiagCpxR(errMult1, A_, ntheta, 3, K); // defines content of ( K = DA )

        // defines error-term multipliers
        lib_matrFillOnesVecCpx(errMult1,ntheta);                   // defines content of errMult1
        for(l=0; l < ntheta; l++)
          errMult1[l].re /= seisWavelet_[l]->getNorm();

        lib_matrFillValueVecCpx(kD3,errMult2,ntheta);    // defines content of errMult2
        for(l=0; l < ntheta; l++)
        {
          errMult2[l].re  /= errorSmooth3[l]->getNorm(); // defines content of errMult2
          errMult2[1].im  /= errorSmooth3[1]->getNorm(); // defines content of errMult2
        }
        fillInverseAbskWRobust(k,errMult3);              // defines content of errMult3
      } //simbox_->getIsConstantThick()
    }

    // Log progress
    if (k > static_cast<int>(nextMonitor)){
      nextMonitor += monitorSize;
      LogKit::LogMessage(LogKit::LOW, "^");
    }

    bool sequentialInput = meanAlpha_->allowsRandomRead();
         sequentialInput = meanBeta_->allowsRandomRead();
         sequentialInput = meanRho_->allowsRandomRead();
         sequentialInput = parSpatialCorr->allowsRandomRead();
         sequentialInput = errCorrUnsmooth->allowsRandomRead();
    for(int iter = 0; iter < ntheta; iter++){
      sequentialInput &= seisData_[iter]->allowsRandomRead();
    }
    bool sequentialOutput =   postCovAlpha->allowsRandomWrite();
         sequentialOutput &=  postCovBeta ->allowsRandomWrite();
         sequentialOutput &=  postCovRho  ->allowsRandomWrite();
         sequentialOutput &=  postCrCovAlphaBeta->allowsRandomWrite();
         sequentialOutput &=  postCrCovAlphaRho ->allowsRandomWrite();
         sequentialOutput &=  postCrCovBetaRho  ->allowsRandomWrite();
         sequentialOutput &=  postAlpha_->allowsRandomWrite();
         sequentialOutput &=  postBeta_ ->allowsRandomWrite();
         sequentialOutput &=  postRho_  ->allowsRandomWrite();
    for(int iter=0;iter<ntheta;iter++){
      sequentialOutput &=    seisData_[iter]->allowsRandomWrite();
    }
    int writeSpinlock = 0;
#if _OPENMP >= 200805
#pragma omp parallel for ordered private(j) default(shared) schedule(static, 1)
#endif
    for(j = 0; j < nyp_cnxp; j++){
// A ordered section ensures sequential ordering
// Sequential ordering is esential in this part in order to retain sequential I/O to disks.
      int idI = k;
      int idJ = j/cnxp;
      int idK = j%cnxp;
#ifdef PROFILING
      const double readTime = omp_get_wtime();
#endif
#pragma omp ordered
{
      errCorrUnsmoothVal = errCorrUnsmooth->getComplexValue(idK, idJ, idI, true);
      ijkMean[0] = meanAlpha_->getComplexValue(idK, idJ, idI, true);
      ijkMean[1] = meanBeta_ ->getComplexValue(idK, idJ, idI, true);
      ijkMean[2] = meanRho_  ->getComplexValue(idK, idJ, idI, true);
      parSpartialCorrVal = parSpatialCorr->getComplexValue(idK, idJ, idI, true);
      for(int iter = 0; iter < ntheta; iter++){
        ijkRes[iter] = seisData_[iter]->getComplexValue(idK, idJ, idI, true);
      }
}
#ifdef PROFILING
      readTimeAccum += omp_get_wtime() - readTime;
#endif
      for(int iter = 0; iter < 3; iter++){
        ijkMean2[iter].im = ijkMean[iter].im;
        ijkMean2[iter].re = ijkMean[iter].re;
      }
      float ijkParLamRe = fabs(parSpartialCorrVal.re);
      for(int iter = 0; iter < 3; iter++){
        for(int iter2 = 0; iter2 < 3; iter2++){
          parVar2[iter][iter2].re = static_cast<fftw_real>(parVar[iter][iter2].re * ijkParLamRe);
          parVar2[iter][iter2].im = static_cast<fftw_real>(parVar[iter][iter2].im * ijkParLamRe);
        }
      }
      float realFrequency = delta*std::min(k, nzp-k); // the physical frequency
      bool current = (realFrequency > lowCut*simboxMinRelThick && realFrequency < highCut); // inverting only relevant frequencies
      for(int iter = 0; iter < 3; iter++){
        ijkMean2[iter].im = ijkMean[iter].im;
        ijkMean2[iter].re = ijkMean[iter].re;
      }
      if(current){
        PROCESS_DATA(omp_get_thread_num());
      }
```

```

        // A spinlock is used to force serial execution without use of the ordered because
            ordered can only
        // be used once per iteration
        // A spinlock works by continously testing a condition until it fails. This is
            more resouce demanding,
        // than using locks based on interrupts.
        // When there is nothing better to use the resouces on a spinlock is as good as
            any lock.
        while(writeSpinlock != j);
#ifdef PROFILING
        const double writeTime = omp_get_wtime();
#endif
        postCovAlpha->setComplexValue(idK, idJ, idI, parVar2[0][0], true);
        postCovBeta ->setComplexValue(idK, idJ, idI, parVar2[1][1], true);
        postCovRho  ->setComplexValue(idK, idJ, idI, parVar2[2][2], true);
        postCrCovAlphaBeta->setComplexValue(idK, idJ, idI, parVar2[0][1], true);
        postCrCovAlphaRho ->setComplexValue(idK, idJ, idI, parVar2[0][2], true);
        postCrCovBetaRho  ->setComplexValue(idK, idJ, idI, parVar2[1][2], true);
        for(int iter=0;iter<ntheta; iter++){
          seisData_[iter]->setComplexValue(idK, idJ, idI, ijkRes3[iter], true);
        }
        postAlpha_->setComplexValue(idK, idJ, idI, ijkMean2[0], true);
        postBeta_ ->setComplexValue(idK, idJ, idI, ijkMean2[1], true);
        postRho_  ->setComplexValue(idK, idJ, idI, ijkMean2[2], true);
#ifdef PROFILING
        writeTimeAccum += omp_get_wtime() - writeTime;
        ptimeAccum += omp_get_wtime() - readTime;
#endif
        // Release the lock.
#pragma omp atomic
        writeSpinlock += 1;
#pragma omp flush(writeSpinlock)
      }
    }
#ifdef PROFILING
  invLoop = omp_get_wtime() - invLoop;
#endif

  LogKit::LogMessage(LogKit::LOW, "\n");
  // Parallel memory cleanup. Each threads cleans up their local copy of threadprivate
      memory.
  // All calls in parallel blocks happends the same times as the number of threads
#if _OPENMP >= 200805
#pragma omp parallel private(j)
#endif
  {
        for(int iter = 0; iter < ntheta; iter++){
          delete[] errVar[iter];
          delete[] KS[iter];
          delete[] margVar[iter] ;
        }
        for(int iter = 0; iter < 3; iter++){
          delete[] KScc[iter];
          //delete[] parVar[iter];
          delete[] parVar2[iter];
          delete[] reduceVar[iter];
        }
        delete[] errVar;
        delete[] ijkAns;
        delete[] ijkDataMean;
        delete[] ijkMean;
        delete[] ijkMean2;
        delete[] ijkRes;
        delete[] ijkRes2;
        delete[] ijkRes3;
        delete[] KS;
        delete[] KScc;
        delete[] margVar;
        //delete[] parVar;
        delete[] parVar2;
        delete[] reduceVar;
  }
  for(i = 0; i < ntheta; i++){
    delete errorSmooth3[i];
    delete[] K[i];
  }
  delete    diff1Operator;
  delete    diff3Operator;
  delete[] errMult1;
  delete[] errMult2;
  delete[] errMult3;
  delete[] errorSmooth3;
  delete[] K;

  // these does not have the initial meaning
  meanAlpha_       = NULL; // the content is taken care of by  postAlpha_
  meanBeta_        = NULL; // the content is taken care of by  postBeta_
  meanRho_         = NULL; // the content is taken care of by  postRho_
  parSpatialCorr   = NULL; // the content is taken care of by  postCovAlpha
  errCorrUnsmooth  = NULL; // the content is taken care of by  postCovBeta

  postAlpha_->endAccess();
  postBeta_->endAccess();
  postRho_->endAccess();

  postCovAlpha->endAccess();
  postCovBeta->endAccess();
  postCovRho->endAccess();
  postCrCovAlphaBeta->endAccess();
  postCrCovAlphaRho->endAccess();
  postCrCovBetaRho->endAccess();

  postAlpha_->invFFTInPlace();
  postBeta_->invFFTInPlace();
  postRho_->invFFTInPlace();

  for(l=0;l<ntheta;l++)
    seisData_[l]->endAccess();

  //Finish use of seisData_, since we need the memory.
  if((outputGridsSeismic_ & IO::RESIDUAL) > 0)
  {
    if(simbox_->getIsConstantThick() != true)
      multiplyDataByScaleWaveletAndWriteToFile("residuals");
    else
    {
      for(l=0;l<ntheta;l++)
      {
        std::string angle      = NRLib::ToString(thetaDeg_[l],1);
        std::string sgriLabel = " Residuals for incidence angle "+angle;
        std::string fileName   = IO::PrefixResiduals() + angle;
        seisData_[l]->setAccessMode(FFTGrid::RANDOMACCESS);
        seisData_[l]->invFFTInPlace();
        seisData_[l]->writeFile(fileName, IO::PathToInversionResults(), simbox_,
            sgriLabel);
        seisData_[l]->endAccess();
      }
    }
  }
  for(l=0;l<ntheta;l++){
    if(seisData_[l] != NULL) delete seisData_[l];
    seisData_[l] = NULL;
  }
  delete[] seisData_;
  seisData_ = NULL;
  LogKit::LogFormatted(LogKit::DEBUGLOW,"\nDEALLOCATING: Seismic data\n");

  if(model_->getVelocityFromInversion() == true) { //Conversion undefined until
      prediction ready. Complete it.
    postAlpha_->setAccessMode(FFTGrid::RANDOMACCESS);
    postAlpha_->expTransf();
    GridMapping * tdMap = model_->getTimeDepthMapping();
    const GridMapping * dcMap = model_->getTimeCutMapping();
    const Simbox * timeSimbox = simbox_;
    if(dcMap != NULL)
      timeSimbox = dcMap->getSimbox();

    tdMap->setMappingFromVelocity(postAlpha_, timeSimbox);
    postAlpha_->logTransf();
    postAlpha_->endAccess();
  }

  if(model_->getModelSettings()->getUseLocalNoise())
  {
    correlations_->invFFT();
    correlations_->createPostVariances();
    correlations_->FFT();
    correctAlphaBetaRho(model_->getModelSettings());
  }

  if(writePrediction_ == true)
    ParameterOutput::writeParameters(simbox_, model_, postAlpha_, postBeta_, postRho_,
        outputGridsElastic_, fileGrid_, -1, false);

  writeBWPredicted();

  Timings::setTimeInversion(wall,cpu);
  omp_unset_lock(&lock);

#ifdef PROFILING
  stringstream ss;
  ss << "Seismic␣inversion␣[cnxp:␣";
  ss << cnxp;
  ss << ",␣nyp:␣";
  ss << nyp_;
  ss << ",␣nzp:␣";
  ss << nzp_;
  ss << "]␣wallclock␣time.";
  wtime = omp_get_wtime() - wtime;
  NRLib::Prof::setName(ss.str(), INVERSIONLOG);
  NRLib::Prof::setName("Seismic␣inversion␣time␣reading.", INVERSIONREADLOG);
  NRLib::Prof::setName("Seismic␣inversion␣time␣writing.", INVERSIONWRITELOG);
  NRLib::Prof::setName("Seismic␣inversion␣inner␣loop␣CPU␣time.", INVCPUTIMELOG);
  NRLib::Prof::setName("Seismic␣Inversion␣loop␣time.", INVERSIONLOOPLOG);
  NRLib::Prof::trackTime(wtime, INVERSIONLOG);
  NRLib::Prof::trackTime(readTimeAccum, INVERSIONREADLOG);
  NRLib::Prof::trackTime(writeTimeAccum, INVERSIONWRITELOG);
  NRLib::Prof::trackTime(ptimeAccum, INVCPUTIMELOG);
  NRLib::Prof::trackTime(invLoop, INVERSIONLOOPLOG);
#endif

  return(0);
}
```

Listing C.4: Optimized inversion code. Snippet from crava.cpp.

```
inline static void fillErrorMatrix(float wnc, const double** errThetaCov, double scale, const fftw_complex* errMult1, const fftw_complex* errMult2, const fftw_complex* errMult3, int matrixSize, fftw_complex** errVar){
  for(int l = 0; l < matrixSize; l++ ) {
    for(int m = 0; m < matrixSize; m++ )
    {       // Note we multiply kWNorm[l] and comp.conj(kWNorm[m]) hence the + and not a minus as in pure multiplication
      float tmp = 0.5f*(1.0f-wnc)*errThetaCov[l][m] * scale;
      errVar[l][m].re  = static_cast<float>(
          tmp * ( errMult1[l].re *  errMult1[m].re +  errMult1[l].im *  errMult1[m].im) +
          tmp * ( errMult2[l].re *  errMult2[m].re +  errMult2[l].im *  errMult2[m].im));
      if(l==m) {
        errVar[l][m].re += static_cast<float>( wnc*errThetaCov[l][m] * errMult3[l].re  * errMult3[l].re);
        errVar[l][m].im   = 0.0f;
      }
      else {
        errVar[l][m].im  = static_cast<float>(
            tmp * (-errMult1[l].re * errMult1[m].im + errMult1[l].im * errMult1[m].re) +
            tmp * (-errMult2[l].re * errMult2[m].im + errMult2[l].im * errMult2[m].re));
      }
    }
  }
}

int Crava::computePostMeanResidAndFFTCov()
{
  Utils::writeHeader("Posterior␣model␣/␣Performing␣Inversion");

#ifdef PROFILING
  double invLoop;
  double innerInvLoop = 0.0;
  double timeReadingAccum = 0.0;
  double timeWritingAccum = 0.0;
  double timeErrorMatrAccum = 0.0;
  double cleanup;
  double startup = omp_get_wtime();
  double wtime = omp_get_wtime();
  double stackFFT;
  double invIO,invIOAccum = 0.0;
#endif
  double wall=0.0, cpu=0.0;
  TimeKit::getTime(wall,cpu);
  int i,j,k,l,m;

  fftw_complex * errMult1;
  fftw_complex * errMult2;
  fftw_complex * errMult3;

  fftw_complex * ijkData;
  fftw_complex * ijkDataMean;
  fftw_complex * ijkRes;
  fftw_complex * ijkMean;
  fftw_complex * ijkAns;
  fftw_complex   kD,kD3;
  fftw_complex   ijkTmp;

  fftw_complex**  K;
  fftw_complex**  KS;
  fftw_complex**  KScc; // cc - complex conjugate (and transposed)
  fftw_complex**  parVar;
  fftw_complex**  margVar;
  fftw_complex**  errVar;
  fftw_complex** reduceVar;

  Wavelet * diff1Operator = new Wavelet(Wavelet::FIRSTORDERFORWARDDIFF,nz_,nzp_);
  Wavelet * diff2Operator = new Wavelet(diff1Operator,Wavelet::FIRSTORDERBACKWARDDIFF);
  Wavelet * diff3Operator = new Wavelet(diff2Operator,Wavelet::FIRSTORDERCENTRALDIFF);

  diff1Operator->fft1DInPlace();
  delete diff2Operator;
  diff3Operator->fft1DInPlace();

  Wavelet ** errorSmooth  = new Wavelet*[ntheta_];
  Wavelet ** errorSmooth2 = new Wavelet*[ntheta_];
  Wavelet ** errorSmooth3 = new Wavelet*[ntheta_];

  int cnxp  = nxp_/2+1;

  for(l = 0; l < ntheta_ ; l++)
  {
    std::string angle = NRLib::ToString(thetaDeg_[l], 1);
    std::string fileName;
    seisData_[l]->setAccessMode(FFTGrid::READANDWRITE);
    if (seisWavelet_[0]->getDim() == 1) {
      errorSmooth[l]  = new Wavelet(seisWavelet_[l],Wavelet::FIRSTORDERFORWARDDIFF);
      errorSmooth2[l] = new Wavelet(errorSmooth[l], Wavelet::FIRSTORDERBACKWARDDIFF);
      errorSmooth3[l] = new Wavelet(errorSmooth2[l],Wavelet::FIRSTORDERCENTRALDIFF);
      fileName = std::string("ErrorSmooth_") + angle + IO::SuffixGeneralData();
      errorSmooth3[l]->printToFile(fileName);
      errorSmooth3[l]->fft1DInPlace();

      fileName = IO::PrefixWavelet() + angle + IO::SuffixGeneralData();
      seisWavelet_[l]->printToFile(fileName);
      seisWavelet_[l]->fft1DInPlace();

      fileName = std::string("FourierWavelet_") + angle + IO::SuffixGeneralData();
      seisWavelet_[l]->printToFile(fileName);
      delete errorSmooth[l];
      delete errorSmooth2[l];
    }
    else { //3D-wavelet
      /* Commented out code removed */
    }
  }
  delete[] errorSmooth;
  delete[] errorSmooth2;

  meanAlpha_->setAccessMode(FFTGrid::READANDWRITE);  //   Note
  meanBeta_ ->setAccessMode(FFTGrid::READANDWRITE);  //   the top five are over written
  meanRho_  ->setAccessMode(FFTGrid::READANDWRITE);  //   does not have the initial
       meaning.

  FFTGrid * parSpatialCorr     = correlations_->getPostCovAlpha(); // NB! Note double
       usage of postCovAlpha
  FFTGrid * errCorrUnsmooth    = correlations_->getPostCovBeta();  // NB! Note double
       usage of postCovBeta
  FFTGrid * postCovAlpha       = correlations_->getPostCovAlpha();
  FFTGrid * postCovBeta        = correlations_->getPostCovBeta();
  FFTGrid * postCovRho         = correlations_->getPostCovRho();
  FFTGrid * postCrCovAlphaBeta = correlations_->getPostCrCovAlphaBeta();
  FFTGrid * postCrCovAlphaRho  = correlations_->getPostCrCovAlphaRho();
  FFTGrid * postCrCovBetaRho   = correlations_->getPostCrCovBetaRho();
  parSpatialCorr     ->setAccessMode(FFTGrid::READANDWRITE);  //   after the prosessing
  errCorrUnsmooth    ->setAccessMode(FFTGrid::READANDWRITE);  //
  postCovRho         ->setAccessMode(FFTGrid::WRITE);
  postCrCovAlphaBeta->setAccessMode(FFTGrid::WRITE);
  postCrCovAlphaRho ->setAccessMode(FFTGrid::WRITE);
  postCrCovBetaRho  ->setAccessMode(FFTGrid::WRITE);

  // Computes the posterior mean first  below the covariance is computed
  // To avoid to many grids in mind at the same time
  double priorVarVp,justfactor;

  int cholFlag;
  float realFrequency;

  const bool usingFileStorage = meanAlpha_->isFile() || meanBeta_->isFile() || meanRho_->isFile()
                                || postAlpha_->isFile() || postBeta_->isFile() || postRho_->isFile();

  LogKit::LogFormatted(LogKit::LOW,"\nBuilding␣posterior␣distribution:");
  float monitorSize = std::max(1.0f, static_cast<float>(nzp_)*0.02f);
  float nextMonitor = monitorSize;
  std::cout
    << "\n␣␣0%␣␣␣␣␣␣␣20%␣␣␣␣␣␣␣40%␣␣␣␣␣␣␣60%␣␣␣␣␣␣␣80%␣␣␣␣␣␣100%"
    << "\n␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣␣␣|␣␣"
    << "\n␣␣^";

#ifdef PROFILING
  startup = omp_get_wtime() - startup;
  invLoop = omp_get_wtime();
#endif
#ifdef PROFILING
#pragma omp parallel default(none) \
     private(i,j,k,l,m,realFrequency,kD,errMult1,errMult2,errMult3,K,kD3,ijkMean,ijkData,ijkRes,ijkTmp,parVar,priorVarVp,errVar,KS,margVar,\
           cholFlag,KScc,reduceVar,ijkDataMean,ijkAns) \
     shared(diff1Operator,diff3Operator,errorSmooth3,cnxp,parSpatialCorr,errCorrUnsmooth,timeReadingAccum,timeErrorMatrAccum,postCovAlpha,postCovBeta,\
          postCovRho,postCrCovAlphaBeta,postCrCovAlphaRho,postCrCovBetaRho,timeWritingAccum,innerInvLoop,nextMonitor,monitorSize,std::cout,stdout)
#else
#pragma omp parallel default(none) \
     private(i,j,k,l,m,realFrequency,kD,errMult1,errMult2,errMult3,K,kD3,ijkMean,ijkData,ijkRes,ijkTmp,parVar,priorVarVp,errVar,KS,margVar,\
           cholFlag,KScc,reduceVar,ijkDataMean,ijkAns) \
     shared(diff1Operator,diff3Operator,errorSmooth3,cnxp,parSpatialCorr,errCorrUnsmooth,postCovAlpha,postCovBeta,\
          postCovRho,postCrCovAlphaBeta,postCrCovAlphaRho,postCrCovBetaRho,nextMonitor,monitorSize,std::cout,stdout)

#endif
{
  errMult1    = new fftw_complex[ntheta_];
  errMult2    = new fftw_complex[ntheta_];
  errMult3    = new fftw_complex[ntheta_];

  ijkData     = new fftw_complex[ntheta_];
  ijkDataMean = new fftw_complex[ntheta_];
  ijkRes      = new fftw_complex[ntheta_];
  ijkMean     = new fftw_complex[3];
  ijkAns      = new fftw_complex[3];

  K  = new fftw_complex*[ntheta_];
  KS  = new fftw_complex*[ntheta_];
  KScc  = new fftw_complex*[3]; // cc - complex conjugate (and transposed)
  parVar = new fftw_complex*[3];
  margVar = new fftw_complex*[ntheta_];
  errVar = new fftw_complex*[ntheta_];
  reduceVar = new fftw_complex*[3];

  for(i = 0; i < ntheta_; i++)
  {
    K[i] = new fftw_complex[3];
    KS[i] = new fftw_complex[3];
    margVar[i] = new fftw_complex[ntheta_];
    errVar[i] = new fftw_complex[ntheta_];
  }

  for(i = 0; i < 3; i++)
  {
    KScc[i] = new fftw_complex[ntheta_];
    parVar[i] = new fftw_complex[3];
    reduceVar[i]= new fftw_complex[3];
  }
#pragma omp for
  for(k = 0; k < nzp_; k++)
  {
    realFrequency = static_cast<float>((nz_*1000.0f)/(simbox_->getlz()*nzp_)*std::min(k,nzp_-k)); // the physical frequency
    kD = diff1Operator->getCAmp(k);                    // defines content of kD
    if (seisWavelet_[0]->getDim() == 1) { //1D-wavelet
```

```
      if( simbox_->getIsConstantThick() == true)
      {
        // defines content of K=WDA
        fillkW(k,errMult1);                                   // errMult1 used as dummy
        lib_matrProdScalVecCpx(kD, errMult1, ntheta_);        // errMult1 used as dummy
        lib_matrProdDiagCpxR(errMult1, A_, ntheta_, 3, K);    // defines content of (
            WDA)     K

        // defines error-term multipliers
        fillkWNorm(k,errMult1,seisWavelet_);                 // defines input of  (kWNorm)
            errMult1
        fillkWNorm(k,errMult2,errorSmooth3);                 // defines input of  (
            kWD3Norm) errMult2
        lib_matrFillOnesVecCpx(errMult3,ntheta_);            // defines content of
            errMult3

      }
      else //simbox_->getIsConstantThick() == false
      {
        kD3 = diff3Operator->getCAmp(k);          // defines  kD3

        // defines content of K = DA
        lib_matrFillValueVecCpx(kD, errMult1, ntheta_);       // errMult1 used as dummy
        lib_matrProdDiagCpxR(errMult1, A_, ntheta_, 3, K); // defines content of ( K =
            DA )

        // defines error-term multipliers
        lib_matrFillOnesVecCpx(errMult1,ntheta_);          // defines content of errMult1
        for(l=0; l < ntheta_; l++)
          errMult1[l].re /= seisWavelet_[l]->getNorm();  // defines content of errMult1

        lib_matrFillValueVecCpx(kD3,errMult2,ntheta_);    // defines content of errMult2
        for(l=0; l < ntheta_; l++)
        {
          errMult2[l].re  /= errorSmooth3[l]->getNorm(); // defines content of errMult2
          errMult2[l].im  /= errorSmooth3[l]->getNorm(); // defines content of errMult2
        }
        fillInverseAbskWRobust(k,errMult3);               // defines content of errMult3
      } //simbox_->getIsConstantThick()
    }

    const bool relevant_frequency = realFrequency > lowCut_*simbox_->getMinRelThick() &&
          realFrequency < highCut_; // inverting only relevant frequencies

#ifdef PROFILING
    const double innerLoop = omp_get_wtime();
#endif
    for( j = 0; j < nyp_; j++) {
      for( i = 0; i < cnxp; i++) {
#ifdef PROFILING
        const double timeReading = omp_get_wtime();
#endif
        ijkTmp       = parSpatialCorr->getComplexValue(i,j,k,true);
        float ijkParLamRe = fabs(ijkTmp.re);

        for(l = 0; l < 3; l++ )
          for(m = 0; m < 3; m++ )
          {
            parVar[l][m].re = static_cast<fftw_real>(parPointCov_[l][m] * ijkParLamRe);
            parVar[l][m].im = static_cast<fftw_real>(0.0f);
          }

#ifdef PROFILING
          timeReadingAccum += omp_get_wtime() - timeReading;
#endif

          if(relevant_frequency) // inverting only relevant frequencies
          {
#ifdef PROFILING
        const double timeReading = omp_get_wtime();
#endif
            ijkMean[0] = meanAlpha_->getComplexValue(i,j,k,true);
            ijkMean[1] = meanBeta_ ->getComplexValue(i,j,k,true);
            ijkMean[2] = meanRho_  ->getComplexValue(i,j,k,true);

            for(l = 0; l < ntheta_; l++ )
            {
              ijkData[l] = seisData_[l]->getComplexValue(i,j,k,true);
              ijkRes[l]  = ijkData[l];
            }

            priorVarVp = parVar[0][0].re;
            ijkTmp       = errCorrUnsmooth->getComplexValue(i,j,k,true);
            double ijkErrLamRe = static_cast<float>(fabs(ijkTmp.re));

#ifdef PROFILING
          timeReadingAccum += omp_get_wtime() - timeReading;
#endif

#ifdef PROFILING
            const double timeError = omp_get_wtime();
#endif
            fillErrorMatrix(wnc_, const_cast<const double**>(errThetaCov_),ijkErrLamRe,
                errMult1, errMult2, errMult3, ntheta_, errVar);
#ifdef PROFILING
              timeErrorMatrAccum += omp_get_wtime() - timeError;
#endif

              lib_matrProdCpx(K, parVar , ntheta_, 3 ,3, KS);                  // KS is
                   defined here
              lib_matrProdAdjointCpx(KS, K, ntheta_, 3 ,ntheta_, margVar); // margVar =
                   (K)S(K)' is defined here
              lib_matrAddMatCpx(errVar, ntheta_,ntheta_, margVar);             // errVar  is
                    added to margVar = (WDA)S(WDA)'  + errVar

              cholFlag=lib_matrCholCpx(ntheta_,margVar);                       // Choleskey
                   factor of margVar is Defined

              if(cholFlag==0)
              { // then it is ok else posterior is identical to prior

                lib_matrAdjoint(KS,ntheta_,3,KScc);                            //  WDAScc is
                      adjoint of WDAS
                lib_matrAXeqBMatCpx(ntheta_, margVar, KS, 3);                  // redefines
                     WDAS
                lib_matrProdCpx(KScc,KS,3,ntheta_,3,reduceVar);                // defines
                     reduceVar
                lib_matrSubtMatCpx(reduceVar,3,3,parVar);                      // redefines
                     parVar as the posterior solution

                lib_matrProdMatVecCpx(K,ijkMean, ntheta_, 3, ijkDataMean); //  defines
                     content of ijkDataMean
                lib_matrSubtVecCpx(ijkDataMean, ntheta_, ijkData);           //  redefines
                      content of ijkData

                lib_matrProdAdjointMatVecCpx(KS,ijkData,3,ntheta_,ijkAns); // defines
                     ijkAns

                lib_matrAddVecCpx(ijkAns, 3,ijkMean);                          // redefines
                     ijkMean
                lib_matrProdMatVecCpx(K,ijkMean, ntheta_, 3, ijkData);      // redefines
                     ijkData
                lib_matrSubtVecCpx(ijkData, ntheta_,ijkRes);                   // redefines
                     ijkRes
              }

#ifdef PROFILING
          const double timeWriting = omp_get_wtime();
#endif
          postAlpha_->setComplexValue(i,j,k,ijkMean[0],true);
          postBeta_ ->setComplexValue(i,j,k,ijkMean[1],true);
          postRho_  ->setComplexValue(i,j,k,ijkMean[2],true);

          for(l=0;l<ntheta_;l++)
            seisData_[l]->setComplexValue(i,j,k,ijkRes[l],true);
#ifdef PROFILING
          timeWritingAccum += omp_get_wtime() - timeWriting;
#endif
          }
          else if(usingFileStorage)
          {
            // Move data from mean to post if using File storage (not the same grids)
            ijkMean[0] = meanAlpha_->getComplexValue(i,j,k,true);
            ijkMean[1] = meanBeta_ ->getComplexValue(i,j,k,true);
            ijkMean[2] = meanRho_  ->getComplexValue(i,j,k,true);
            postAlpha_->setComplexValue(i,j,k,ijkMean[0],true);
            postBeta_ ->setComplexValue(i,j,k,ijkMean[1],true);
            postRho_  ->setComplexValue(i,j,k,ijkMean[2],true);
          }
#ifdef PROFILING
          const double timeWriting = omp_get_wtime();
#endif

          postCovAlpha->setComplexValue(i,j,k,parVar[0][0],true);
          postCovBeta ->setComplexValue(i,j,k,parVar[1][1],true);
          postCovRho  ->setComplexValue(i,j,k,parVar[2][2],true);
          postCrCovAlphaBeta->setComplexValue(i,j,k,parVar[0][1],true);
          postCrCovAlphaRho ->setComplexValue(i,j,k,parVar[0][2],true);
          postCrCovBetaRho  ->setComplexValue(i,j,k,parVar[1][2],true);

#ifdef PROFILING
          timeWritingAccum += omp_get_wtime() - timeWriting;
#endif
        }
      }
#ifdef PROFILING
      innerInvLoop += omp_get_wtime() - innerLoop;
#endif
      // Log progress
      if (k+1 >= static_cast<int>(nextMonitor))
      {
        nextMonitor += monitorSize;
        std::cout << "^";
        fflush(stdout);
      }
    }

    // Cleanup in each thread
    delete [] errMult1;
    delete [] errMult2;
    delete [] errMult3;
    delete [] ijkData;
    delete [] ijkDataMean;
    delete [] ijkRes;
    delete [] ijkMean ;
    delete [] ijkAns;


    for(i = 0; i < ntheta_; i++)
    {
      delete[]  K[i];
      delete[]  KS[i];
      delete[]  margVar[i] ;
      delete[] errVar[i] ;
    }
    delete[] K;
    delete[] KS;
    delete[] margVar;
    delete[] errVar  ;

    for(i = 0; i < 3; i++)
    {
      delete[] KScc[i];
      delete[] parVar[i] ;
      delete[] reduceVar[i];
    }
    delete[] KScc;
    delete[] parVar;
    delete[] reduceVar;
}
    std::cout << "\n";
#ifdef PROFILING
    invLoop = omp_get_wtime() - invLoop;
    cleanup = omp_get_wtime();
#endif

    // these does not have the initial meaning
    meanAlpha_       = NULL; // the content is taken care of by  postAlpha_
    meanBeta_        = NULL; // the content is taken care of by  postBeta_
    meanRho_         = NULL; // the content is taken care of by  postRho_
    parSpatialCorr   = NULL; // the content is taken care of by  postCovAlpha
    errCorrUnsmooth = NULL; // the content is taken care of by  postCovBeta

    postAlpha_->endAccess();
    postBeta_->endAccess();
    postRho_->endAccess();

    postCovAlpha->endAccess();
    postCovBeta->endAccess();
    postCovRho->endAccess();
    postCrCovAlphaBeta->endAccess();
    postCrCovAlphaRho->endAccess();
    postCrCovBetaRho->endAccess();

#ifdef PROFILING
    stackFFT = omp_get_wtime();
#endif
    postAlpha_->invFFTInPlace();
    postBeta_->invFFTInPlace();
```

```
  postRho_->invFFTInPlace();
#ifdef PROFILING
  stackFFT = omp_get_wtime() - stackFFT;
#endif

  for(l=0;l<ntheta_;l++)
    seisData_[l]->endAccess();

#ifdef PROFILING
  invIO = omp_get_wtime();
#endif
  //Finish use of seisData_, since we need the memory.
  if((outputGridsSeismic_ & IO::RESIDUAL) > 0)
  {
    if(simbox_->getIsConstantThick() != true)
      multiplyDataByScaleWaveletAndWriteToFile("residuals");
    else
    {
      for(l=0;l<ntheta_;l++)
      {
        std::string angle     = NRLib::ToString(thetaDeg_[l],1);
        std::string sgriLabel = " Residuals for incidence angle "+angle;
        std::string fileName  = IO::PrefixResiduals() + angle;
        seisData_[l]->setAccessMode(FFTGrid::RANDOMACCESS);
#ifdef PROFILING
  const double fftTime = omp_get_wtime();
#endif
        seisData_[l]->invFFTInPlace();
#ifdef PROFILING
  stackFFT += omp_get_wtime() - fftTime;
#endif
        seisData_[l]->writeFile(fileName, IO::PathToInversionResults(), simbox_, sgriLabel);
        seisData_[l]->endAccess();
      }
    }
  }
  for(l=0;l<ntheta_;l++)
    delete seisData_[l];
  LogKit::LogFormatted(LogKit::DEBUGLOW,"\nDEALLOCATING: Seismic data\n");
#ifdef PROFILING
  invIOAccum += omp_get_wtime() - invIO;
#endif

  if(model_->getVelocityFromInversion() == true) { //Conversion undefined until prediction ready. Complete it.
    postAlpha_->setAccessMode(FFTGrid::RANDOMACCESS);
    postAlpha_->expTransf();
    GridMapping * tdMap = model_->getTimeDepthMapping();
    const GridMapping * dcMap = model_->getTimeCutMapping();
    const Simbox * timeSimbox = simbox_;
    if(dcMap != NULL)
      timeSimbox = dcMap->getSimbox();

    tdMap->setMappingFromVelocity(postAlpha_, timeSimbox);
    postAlpha_->logTransf();
    postAlpha_->endAccess();
  }

  if(model_->getModelSettings()->getUseLocalNoise())
  {
    correlations_->invFFT();
    correlations_->createPostVariances();
    correlations_->FFT();
    correctAlphaBetaRho(model_->getModelSettings());
  }

#ifdef PROFILING
  invIO = omp_get_wtime();
#endif
  if(writePrediction_ == true)
    ParameterOutput::writeParameters(simbox_, model_, postAlpha_, postBeta_, postRho_,
                                     outputGridsElastic_, fileGrid_, -1, false);

  writeBWPredicted();
#ifdef PROFILING
  invIOAccum += omp_get_wtime() - invIO;
#endif

  delete [] seisData_;
  for(i = 0; i < ntheta_; i++)
  {
    delete errorSmooth3[i];
  }
  delete[] errorSmooth3;

  delete    diff1Operator;
  delete    diff3Operator;

  Timings::setTimeInversion(wall,cpu);

#ifdef PROFILING
  cleanup = omp_get_wtime() - cleanup - stackFFT - invIOAccum;
  wtime = omp_get_wtime() - wtime;

  stringstream ss;
  ss << "Seismic inversion [cnxp: ";
  ss << cnxp;
  ss << ", nyp: ";
  ss << nyp_;
  ss << ", nzp: ";
  ss << nzp_;
  ss << "] wallclock time.";

  NRLib::Prof::trackTime(invLoop, INVERSION_INV_LOOP);
  NRLib::Prof::trackTime(innerInvLoop, INVERSION_INNER_INV_LOOP);
  NRLib::Prof::trackTime(timeReadingAccum, INVERSION_TIME_READING);
  NRLib::Prof::trackTime(timeWritingAccum, INVERSION_TIME_WRITING);
  NRLib::Prof::trackTime(timeErrorMatrAccum, INVERSION_TIME_ERROR_MATR);
  NRLib::Prof::trackTime(cleanup, INVERSIONCLEANUP);
  NRLib::Prof::trackTime(startup, INVERSION_STARTUP);
  NRLib::Prof::trackTime(stackFFT, INVERSION_STACK_FFT);
  NRLib::Prof::trackTime(invIOAccum, INVERSION_IO);
  NRLib::Prof::trackTime(wtime, INVERSIONLOG);

  NRLib::Prof::setName(ss.str(), INVERSIONLOG);
  NRLib::Prof::setName("Inversion I/O time", INVERSION_IO);
  NRLib::Prof::setName("Inversion Stack FFT", INVERSION_STACK_FFT);
  NRLib::Prof::setName("Inversion startup", INVERSION_STARTUP);
  NRLib::Prof::setName("Inversion cleanup", INVERSIONCLEANUP);
  NRLib::Prof::setName("Inversion creating error matr", INVERSION_TIME_ERROR_MATR);
  NRLib::Prof::setName("Inversion time spent writing", INVERSION_TIME_WRITING);
  NRLib::Prof::setName("Inversion time spent reading", INVERSION_TIME_READING);
  NRLib::Prof::setName("Inversion inner Loop", INVERSION_INNER_INV_LOOP);
  NRLib::Prof::setName("Inversion Loop", INVERSION_INV_LOOP);
#endif
  return(0);
}
```