NTNU
Norwegian University of
Science and Technology

Parallel Seismic Inversion for Shared Memory Systems

Andreas Dreyer Hysing

Master of Science in Computer Science

Submission date:	August 2010
Supervisor:	Anne Cathrine Elster, IDI
Co-supervisor:	Alf Birger Rustad, Statoil

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

Problem Description

Seismic processing applications typically process large amounts of data and are very computationally demanding. A wide variety of high performance applications have recently been modified to use newer multi-core architectures, often with considerable performance improvements. In this thesis, we will, in collaborations with Statoil, explore how a scientific application for seismic inversion can take advantage of multi-core programming on x86 architecture. The thesis will focus on most effective domain divisions, communication patterns and multithreaded scalability. Performance comparison with the original codes will be included, as well as an evaluation of the development effort required for implementing such techniques. At last the project will explore future trends in high performance computation for this type of application.

Assignment given: 15. February 2010
Supervisor: Anne Cathrine Elster, IDI

Abstract

Seismic processing applications provide invaluable tools for the oil industry. However, they typically process large amounts of data and are very computationally demanding. A wide variety of high performanc eapplications have therefore recently been modified to use newer multi-core architectures, often with considerable performance improvements.

In this thesis, we analyze a program for seismic inversion called CRAVA in collaboration with Statoil researchers and developers at Norsk Regnesentral(NR). CRAVA takes a new geostatistical approach to seismic inversion with beneficial properties. It was originally developed by NR in close collaboration with Statoil's researcher.

The focus in this thesis is on decreasing execution time and improving performance. Both serial and parallel techniques are applied. The serial optimization focused mainly on overall code efficiency without result changes. These changes included, avoiding boundary checks for each sample, but rather per volume and reusing FFTW plans. Open MP was used as a our parallel programming language for utilizing the computational power of modern multi-core processor. It involves spawing a set of threads that can be run in parallel on the available computational cores.

Our optimization and parallelization techniques are tested on an 8 core AMD Opteron system with 32GB RAM. The results are tested empiric against real world data sets from the industry. Our improved version is 10 times faster than the initial implementation on a costly cubic sampling step. This improvement consists of a 1.6 improvement in serial efficiency and a 4.5 speedup. Better cubic sampling lowered wall time with 25%. The seismic inversion step is also developed to exploit parallelism. Finally, ideas for future work are included.

Acknowledgments

This thesis was made possible by contributions by numerous different people. I would like to thank Dr. Anne Cathrine Elster for introducing me to high performance computing, and for teaching me what I know about distributed computing. I would also thank Dr. Elster for proposing, organizing and advising all aspects of my master thesis, and giving me the opportunity to attend International Supercomputer Conference 10 (ISC10) in Hamburg, Germany. ISC10 provided insights to high performance computing in the years to come.
From Statoil I would like to thank Dr. Alf Birger Raustad and Mr. Sebastian Ng for their contributions. Dr. Alf was the main contributor for opening of the program source code, while Mr. Sebastian Ng has provided valuable insight to field of geology, and how geophysicists work. He also showed me how Statoil integrates software into their workflow.
From The Norwegian Computing Centre I would like to thank Dr. Ragnar Hauge for maintaining a fast channel of communication with the developers. His insights lead the focus of development in the right direction, and gave explanation of mathematical models behind the program. In my opinion both parties have gained a better understanding of scientific software development through frequent exchange of information.

Andreas Dreyer Hysing

Contents

List of Algorithms

List of Figures

Listings

List of Tables

Glossary

API Application Programming Interface defines how software allows it to interact with other software. It consists of an API specification and an implementation. 8, 9

API Specification Application Programming Interface Specification consists of program routines and related data structures. It defines how the software interact with other software.. 8, 9

CPU Central Processing Unit. 5

CRAVA Conditioning Reservoir Variables to Amplitude Versus Angle data. 1, 2, 11, 12, 19, 20, 22, 23, 26, 29, 35, 38

Embarrassingly Parallel Problem A problem that can divided into multiple sub tasks with no dependency between sub tasks. 8

GCC GNU Compiler Collection. 2, 9, 33, 34

GPU Graphics Processing Unit. 3

heightmap Raster image used to store surface elevation data. 20

ICC Intel Compiler Collection. 2

MIMD Multiple Instruction stream, Multiple Data stream. 5

OpenCL Open Compute Language. 8

OpenMP Open Multi-Processing. 3, 8–10, 22, 25, 33, 34

RAM Random Access Memory. 2

SIMD Single Instruction, Multiple Data stream. 5, 8

Chapter 1

Introduction

Seismic inversion can be described as the task of analyzing elastic properties of seismic data reflected in a physical matter e.g. subsurface. This process is vital for scientists and engineers trying to understand the geology of an area.
The field of seismic inversion has traditionally been a field of great development of improved models and approximations. Traditional seismic inversion methods treats facies as deterministic [1], but seismic response is far too uncertain to give accurate approximations alone. Therefore geostatistical approaches with an error estimate in the model is a big advantage. By using a Bayesian geostatistical approach Conditioning Reservoir Variables to Amplitude Versus Angle data (CRAVA) is able to compute results including uncertainties.
At the time of writing the technique applied in CRAVA is quite new [2]. After a theoretical breakthrough it will take some time before there are made efficient implementations. It is therefore natural to expect there to be a great potential for better serial efficiency in CRAVA. Typically seismic inversion involves large data and therefore slow processing. With a low level of data dependence in models, and large amounts of computations it is reasonable to expect time savings by exploiting parallelism. The combination of potential for serial and parallel improvements makes CRAVA ideal as practical comparison of the two.

1.1 Thesis Description

The thesis description sets a framework for the thesis. It states

Seismic processing applications typically process large amounts of data and are very computationally demanding. A wide variety of high performance applications have recently been modified to use newer multi-core architectures, often with considerable performance improvements.
In this thesis, we will, in collaborations with Statoil, explore how a scientific application for seismic inversion can take advantage of multi-core programming on x86 architecture. The thesis will focus on most effective domain divisions,

communication patterns and multithreaded scalability. Performance comparison with the original codes will be included, as well as an evaluation of the development effort required for implementing such techniques. At last the project will explore future trends in high performance computation for this type of application.

1.2 Project Boundaries

Although the thesis description sets the focus for multiple important aspects were not covered. To solve this a set of practical requirements were made. These were focused on finding the best solution for engineers and geophysicists that works with CRAVA on a daily basis. The methodology for finding project requirements was meetings and discussions with both users and developers.

Compatible With Workstations CRAVA is in house software[1]. in Statoil. It should therefore be able to run both their computer workstations and cluster nodes with at least dual core X86 CPUs, and 32 gigabytes of Random Access Memory (RAM).

No Loss of Result Precision The program should deliver the same or better precision and results than the original program.

Focus on Computer Science When working with other fields it comes natural that design in one field affects the others. In high performance computing design changes often brings change in precision or execution times. The challenge when working cross is that it requires an overview and perspective of the problem. While this is theoretically possible, this thesis does not validate alternative results. Mainly because it requires statistical analysis of the results. Instead potential improvements will be commented when the author thinks changes could have significant impact.

Optimize the Common Case CRAVA contains numerous features that are not directly related to the main process of seismic inversion. Also CRAVA has different functionality affecting memory usage, and types of output. This thesis focuses on the process of seismic inversion with realistic workloads with no additional pre processing and post processing. The software is optimized for GNU Compiler Collection (GCC) 4.4.

Backwards Compatible The final CRAVA should keep the same functionality as it had when it started. The most important aspects are support for GCC, Intel Compiler Collection (ICC) and Microsoft Visual Studio. CRAVA also supports offloading buffers to disk to save main memory.

[1] In house software is only available within the company it is developed for.

1.3 Choosing OpenMP Threading Model

Finding the right threading model is vital for any large parallel program. A wrong choice of threading model is difficult to program and slows down development. It can even give worse than serial performance. In this thesis available hardware rules out steam processing techniques which require Graphics Processing Unit (GPU) processing. The alternatives are MPI, Open Multi-Processing (OpenMP) or other threading libraries. The author thinks MPI contains too much complexity for the task. Other thread libraries for CPU would probably make as small difference in terms of code changes and speed compared to OpenMP. The final choice is made on two important aspects. 57514 lines of C++ code is a strong argument for making as small changes to programming model as possible. As 3.2.2 explains OpenMP is perfect for injection of parallel codes into existing code. OpenMP ships with most modern compilers [3]. OpenMP is also the only programming model studied that makes it easy to create backwards compatible code. OpenMP even allows stepwise development from serial to parallel which is unique for the alternatives that were studied. Last but not least OpenMP is tested and proven technology in high performance computing. Bugs and performance is usually not a problem with well tested software.

1.4 Outline

Chapter 1 contains the thesis description and explanations for other design decisions for this thesis. It also argues for all design decisions that are not clearly defined in the thesis description together with argumentation for these decisions. It also includes explanations of tools.

Chapter 2 describes some aspects of the design for every modern PC that is important for scientific computing.

Chapter 3 explains the theoretical background. It contains a description of resources and software used in the project with focus on the particular problem of seismic inversion.
It also contains important theoretical concepts on parallel programming and competing technologies.

Chapter 4 describes motivations for of seismic inversion. It also describes the mathematical and computer science theory, and how CRAVA applies it.

Chapter 5 describes the state of the program at the start of this thesis. It describes the aspects of CRAVA that were improved. It also explains challenges during implementation. This chapter classifies each improvement as a serial or parallel optimization.

Chapter 6 describes benchmark results and reflections around the results.

Chapter 7 summerizes the conclusions, and also contains a short description of future work and alternative designs.

Chapter 2

Modern Computer Design

The modern computer is the result of a computer revolution. The main reason for this revolution has been the exponential growth in computing power. To achieve throughput the modern Central Processing Unit (CPU) has a high flock frequency in the range of Gigahertz, and sophisticated designs to allow as high instruction throughput as possible. One such design is instruction pipelining. Instead of operating on one program instruction at a time the modern CPU divides each program instruction into smaller sub tasks. CPUs consist of functional units. By performing different sub tasks of program instructions on different executing units the CPU can remain close to fully utilized at all times. Pipelining is vital to modern CPU design, but introduces problems. When an instruction stalls the pipeline is affected. A steady increase in clock speed has resulted in CPUs reading memory faster than main memory can provide. This leads to stalls every time the CPU fetches main memory. To prevent stalling pipelines a significant part of the CPU is used to decrease stalls. CPUs use multiple layers of high-speed cache which reauses frequently used registers. The second method to keep a high instruction throughput is algorithms that try to decrease the effect of memory stalls. Branch prediction with out of order execution minimizes the effect of branches by guessing the right execution a branch. The result is a CPU design where performance is bound by the most effective use of CPU cache and branch prediction.
For a long time new CPU designs outperformed older alternatives by increasing voltage and clock frequency. This continued until power consumption and heat hit a level that did not allow further growth. The phenomena have been named "the power wall". Now the only way to maintain a steady increase in throughput is with more executing units. This is a paradigm shift in computer architecture away from increasing serial throughput to increasing parallel throughput. Modern processors also have become parallel first by adding support for vector instructions. Vector Instructions executes on bundles of registers. Later designs put multiple processor cores on a single die. The modern processor does therefore support both Single Instruction, Multiple Data stream (SIMD) and Multiple Instruction stream, Multiple Data stream (MIMD) simultaneously.

2.1 Parallel Programming Concepts

Parallel processing enables programs to execute in a shorter time span. To the user the program has a lower wall clock time. In contrast the CPU measures CPU time, which is the time for one thread to perform a task. In a parallel processor wall time is not the same as CPU time because different sub tasks are running on different threads. The wall time is smaller than the sum of the CPU times for each thread. With a concept of parallel programs it of interest for program designers to compare parallel speed improvements by a more general concept of speedup. Speedup is

$$S(p) = \frac{t_s}{t_p} \tag{2.1}$$

where t_s is the wall time for the given task with the best serial algorithm while t_p is the wall time for the best parallel algorithm with p parallel processes. There are numerous practical aspects that affects program speedup, but the definition of speedup depends on a task and number of processes. If $S(p) = p$ the program has a linear speedup.

All programs contain dependencies which forces a certain order of execution. Speedup for mixed serial and parallel codes was first studied by Gene Amdahl [4], and resulted in what is known as Amdahl's law. Amdahl's law assumes that the parallel parts scales linearly. A portion f of the program is performed in parallel. The serial execution time is given by t_s. Parallel execution time with p processes would then be given by $t_p = (1-f)t_s + ft_s/p$. This gives a speedup of

$$S(p) = \frac{t_s}{(1-f)t_s + \frac{ft_s}{p}} = \frac{1}{(1-f) + \frac{f}{p}} \tag{2.2}$$

Amdahl's law tells us that the increase in speedup is only dependent on the serial fraction f and the number of processes p. With an unlimited number of processes Amdahl's law gives us maximum speedup of

$$\lim_{p \to \infty} S(p) = \frac{1}{1-f} \tag{2.3}$$

Amdahl originally coined his discovery as an argument against the multi-CPU systems of the 1960s. Today Amdahl's law is still valid, but there are no real alternatives to parallel programming. It should be noted that Amdahl's law only covers execution time. There is still a great potential for other aspects e.g. energy effectiveness, better software and different tasks.

Chapter 3

Parallel Programming

This chapter explains the alternatives for parallel computation. It explains the three best technologies for modern high performance computing, and puts them into a context. Advantages and disadvantages of each technology is explained, and compared to the alternatives for their optimal type of problems.

3.1 Parallel Programming Models

3.1.1 Cluster Computing

A computer cluster is a group of multiple computers coupled together by high speed networking for the purpose of running computationally intensive programs. Computer clusters uses middleware software to execute programs across multiple machines. This combines the power of its processors to achieve multiple orders of magnitude higher throughput than what is possible with a single computer. Cluster computers scales computing power linearly with available resources. Cluster can therefore be scaled to preferred size. But cluster computing comes with a high price. With additional resources follow additional expenses. Both purchase, power and tearing of hardware in a cluster computer scales linearly with the number of compute nodes.[1] Cluster computers have distributed memory machines with multiple memories across multiple physical computers. In addition high latency between nodes forces cluster programs to take into account the physical topology of network and memory when designing software. Running programs across clusters requires custom software, which also adds to the development costs.

[1]A compute node is one virtual or physical or virtual computer in a computer cluster.

3.1.2 GPGPU

Driven by the market of computer gaming the computer graphics card have become the fastest processors in the PC. The ability of programmable shaders on the GPU led scientists and engineers to use GPUs as an alternative to CPUs for programming. GPU vendors saw the market and started adapting graphics cards for other applications than graphics. This lead to development of the General Purpose Graphics Processing Unit named GPGPU. The design of the GPU was driven by pixel processing. Pixel processing is an Embarrassingly Parallel Problem. GPUs are therefore focused purely on SIMD through simple cores. Compared to CPUs GPUs have high speed programmable memory with high locality of the data. The disadvantage of GPU design is a lack of logic to detect branches and poor integer performance. Therefore a GPU design performs badly on serial parts and complex logic. Another problem with GPU programming is a fundamental change in software which breaks compatibility with old codebases. A new API called Open Compute Language (OpenCL) would in theory allow the same code to run both on CPUs and GPUs. Its implementations have been poor on performance or, non-existent on certain platforms for the first year. It is therefore a natural conclusion that GPGPU will be adopted primarily in new projects in the future.

3.2 Parallel Programming Technologies

3.2.1 MPI

Message Passing Interface, often shortened MPI, is a API Specification for message based communication for distributed memory systems. It has become the de facto standard for communication in high performance computing [5]. Because MPI is a API Specification numerous different implementations exits. Some of these specifications are developed by academia while other are developed and shipped by hardware vendors designed to exploit the capabilities of their systems. MPI version 1 features one-to-one and group messaging functions. The MPI version 2.0 added new features for dynamic process management, one-sided communication and parallel I/O [6]. Since MPI version 2 is backwards compatible of version 1 there is no need for porting MPI-1 code to MPI-2. MPI version 1 and version 2 are often shortened MPI-1 and MPI-2.
MPI solves communication between compute nodes and with explicit message passing between different nodes. The advantage of explicit communication is that the programmer has full control of where memory is located. A big disadvantage with MPI is that big programs with lots of communications tend to be cluttered by explicit communication. Another problem with MPI is that it is designed to challenge the complexity of distributed memory systems. Modern workstations are parallel, but a shared main memory. MPI requires considerable larger amounts of code than the equivalent OpenMP or native thread library [7].

3.2.2 OpenMP

OpenMP is a API Specification for shared memory parallelism. As the name suggests OpenMP is an open standard. It was developed by the OpenMP review board in an effort to standardize parallel programming on shared memory systems. Version 1.0 was released in October 1997. At the time of writing of this work the newest version is version 3.0. Many compiler vendors does only support older versions. Microsoft Visual C++ 2010 only supports version 2.5, while GCC introduced support with version 4.3.2 [3].
OpenMP enables shared memory processing by applying parallelism hints to the compiler. This design is based on the philosophy of communication by sharing, which is both the strength and weakness of OpenMP. An advantage of communication by sharing is that it does not require buffers because different threads are sharing the same address space.
OpenMP has a declarative take to parallelism. Instead of explicitly spawning threads OpenMP code declares parallel blocks. The API find the most effective way to spawn a thread groups to perform the block in parallel. Work in loops can be marked with work sharing directives to divide the work between threads. How OpenMP solves work sharing is implementation specific. Optionally one can take control of OpenMP and specify a scheduler and number of threads.
The programmer also has to specify the what data is shared between threads, and what data is available before and after parallel blocks.
A naive C implementation of matrix multiply with OpenMP might look like the example 3.1.

```
#include <omp.h>

int i, j, k;
#pragma omp parallel
{
#pragma for private(i, j)
    for(i = 0; i < rows_matrix1; i++){
        for(j = 0; j < columns_matrix2; j++){
            result[i][j] = 0.0
            for(k = 0; k < rows_matrix2; k++){
                result[i][j] += first[i][k] * second[k][j];
            }
        }
    }
}
```

Listing 3.1: Matrix multiply with OpenMP.

If the parallel blocks have a heterogeneous nature it is better to write the code as parallel sections shown in axample 3.2.

```
#include <omp.h>

int computation1(){
/* A function performing some computation */
}

int computation2(){
/* A function performing some other computation */
}
```

```

int main(int argc, char** arv){
#pragma omp parallel
    {
#pragma omp sections
        {
#pragma omp section
            computation1();
#pragma omp section
            computation2();
        }
    }
    /* serial code */
    return 0;
}
```

Listing 3.2: Task parallelism with OpenMP.

As the examples state, there is very little code needed to a serial code to perform a parallel computation. When there is communication between threads OpenMP becomes much more challenging to use. The programmer has to specify all variables to be shared, and perform `flush` directives to enforce memory coherence. OpenMP has no notion of group communication, and uses compiler directives for most of its functionality. Compiler directives have the benefit of being compiler specific. C++ defines that unknown commands are ignored by the compiler. This means that it is easy to do surgical incision in serial code and still remain compatible with compilers without OpenMP support. The disadvantage of directives is that most compilers are notoriously bad at reporting errors in directives [8].

3.3 Project State

3.3.1 CRAVA

CRAVA stands for Conditioning Reservoir Variables to Amplitude Versus Angle data. CRAVA is developed by The Norwegian Computing Centre in collaboration with Statoil. CRAVA uses seismic angle stacks and well logs calculate elastic properties and moreover estimate geological facies. General seismic inversion combines low frequency information from wells with higher frequency information from seismic data. Output from CRAVA is used for later processing or further analysis. CRAVAs typical user scenario is improving efficiency in an oil field. Different facies has different oil capacity. Use of well logs for estimation is both the advantage and drawback with CRAVA. Drilling wells is expensive, but is already a vital part of oil production. Because of drilling costs CRAVA modelling is not used for oil exploration.

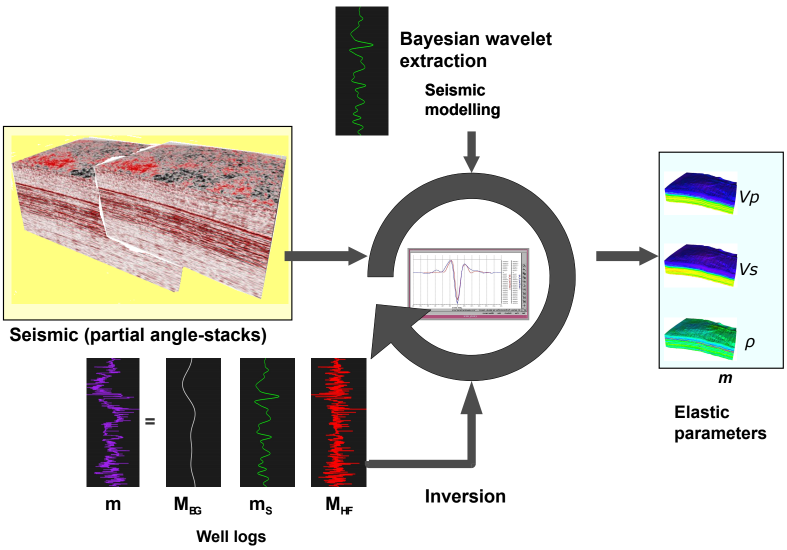


Figure 3.1: Input and Output of CRAVA. Printed with permission from Statoil.

CRAVA runs in either estimation mode or inversion mode. In estimation mode a project requires seismic angle stacks and well logs. In estimation the background model is resampled. All frequencies below 6Hz are treated as noise, and are cut off.

Wells can be interpolated with kriging because the model assumes a log-normal background model.

With wavelets, noise and background model from estimated, CRAVA can generate elastic parameters $\{V_p, V_s, \rho\}$ in a so called seismic inversion process. Elastic parameters are vital for later modelling. By using the elastic parameters and information from well logs CRAVA is able to generate facies probabilities from inversion results. Inversion and estimation can be combined into a single processing step. Combining estimation and inversion saves the process of writing background model to disk. For large reservoirs there is a key issue not perform more I/O than necessary.

3.3.2 Testing

CRAVA source code is bundled with 10 correctness tests. These tests compares variance and average numerical difference between output 3D seismic, wells and wavelets with pre calculated results from correct estimations and inversions. These correctness tests used throughout the development. Its most relevant properties are listed in A.2. For benchmarking it is important to have a test set with realistic workloads. A realistic workload should have the same wall times and proportion of wall times for different processing steps as the executions of the end user. Different processing steps have different scalability properties which makes small data sets improper to locate bottlenecks. The test used for profiling is parts of estimations done internally in Statoil on the Snorre oil reservoir in March 2010 from now on named the benchmark test. The relevant project settings for the benchmark test are listed in A.1.

3.3.3 Profiling Techniques

```
Section                              CPU time                Wall time
--------------------------------------------------------------------------
Loading seismic data       113.64     2.17 %    1203.00     17.19 %
Resampling seismic data   1604.98    30.59 %    1606.00     22.95 %
Wells                        7.48     0.14 %       8.00      0.11 %
Prior expection            584.82    11.15 %     585.00      8.36 %
Prior correlation            5.07     0.10 %       5.00      0.07 %
Building stochas. model   1182.95    22.55 %    1183.00     16.91 %
Inversion                  561.00    10.69 %     659.00      9.42 %
Parameter filter           878.14    16.74 %     879.00     12.56 %
Rest                       308.30     5.88 %     869.00     12.42 %
--------------------------------------------------------------------------
Total                     5246.39   100.00 %    6997.00    100.00 %

Total CPU  time used in CRAVA:   5246 seconds
Total Wall time used in CRAVA:   6997 seconds
```

Listing 3.3: Initial program profiling of the Snorre field on system 6.1.1.

In an early phase of the thesis CRAVA was benchmarked. CRAVA uses 23% of all time on a resampling process. Except resampling no single processing step takes up more than 20% of the execution time. Unfortunately seismic inversion is one of the least significant processing steps. It should noted that CPU time and wall time were not consistent on the loading seismic data task. That step consists of large disk reads. Later tests showed that it varied greatly depending on how many other users were using the disk system on the cluster.

Chapter 4

Case Study: Seismic Inversion

This chapter describes seismic inversion, and mathematical and statistical models needed for seismic inversion. The most relevant maths is studied in terms of memory consumption and memory dependencies.

4.1 Seismic Acquisition

In general seismic inversion is used to gather a quantitative property descriptor of a reservoir based on measured seismic response. The seismic response consists of amplitude. The resulting property descriptor usually contains elastic parameters can for instance be shear-wave velocity V_s, pressure wave velocity V_p and density ρ. These are used by geophysicist to predicting the location of different facies in the reservoir [9]. Categorization of facies has high importance for the petroleum industry. On marine fields the seismic response, is acquired by an observation ship. The observation ship sends out multiple sounds with frequency diagram shaped like a wavelet. The most common used zero-facies wavelet in use is the Ricker wavelet [10]. These sounds are reflected in the reservoir, and the angle ϕ, amplitude and offset of the reflected sound signal is registered by sensors from the ship. The raw seismic from the sensory has multiple position-dependent angles. This raw seismic is then processed in by prestack migration. Prestack migration gathers up closely related angles in groups called angle stacks. An angle stack consists of multiple traces with closely related reflection angles. After prestack migration we assume a constant reflection angle ρ for each angle stack.

4.2 Geostatistical processing

The model assumes weak contrast approximation [11] of the Zoeppritz equations [12]. We then assume that elastic parameters $V_p(x, y, t)$ and $V_s(x, y, t)$ are stationary . Based on this assumption it is possible define seismic data as a convolution.

$$d_{obs}(x,y,t,\theta) = \int \omega(\tau,\theta)c(x,y,t-\tau,\theta)d\tau + e(x,y,t,\theta) \tag{4.1}$$

In this model ω is an angle dependent and position independent wavelet, $\tau = |t_2 - t_1|$ is a distance along time axis, the full integral is the synthetic seismic and e is an angle and location dependent error. The integral makes up the background model.

4.3 Statistical Model

Based on the assumption that $\{V_p(xt), V_s(x,t), \rho(x,t)\}$ are log-normal random field their joint distribution is given by a multi-normal distribution

$$\vec{m}(x,y,t) \sim \mathcal{N}\left(\boldsymbol{\mu}_m(x,y,t), \Sigma_m(x_1,y_1,t_1;x_2,y_2,t_2)\right) \tag{4.2}$$

Equation 4.1 can be written on matrix form as

$$\begin{aligned} d_{obs} &= WADm + e \\ &= Gm + e \end{aligned} \tag{4.3}$$

All values are discrete representations of variables in 4.2 and 4.1. We solve this equation to find m. The equation has linear properties.

It is possible to estimate 4.2 as the time and position discrete distribution [2].

$$\begin{aligned} \vec{d}_{obs} &\sim \mathcal{N}_{n_d}\left(\boldsymbol{\mu}_d, \Sigma_d\right) \\ \boldsymbol{\mu}_d &= \vec{G}\boldsymbol{\mu}_m \\ \Sigma_d &= \vec{G}\Sigma_m\vec{G}^T + \Sigma_e \end{aligned} \tag{4.4}$$

It is then possible to obtain $\vec{m}$ and $\vec{d}_{obs}$ based standard theory for Gaussian distributions [2]

$$\begin{aligned} \mu_{m|d_{obs}} &= \mu_m + \Sigma_m\vec{G}^T\Sigma_d^{-1}(d_{obs} - \mu_d) \\ \Sigma_{m|d_{obs}} &= \Sigma_m - \Sigma_m\vec{G}^T\Sigma_d^{-1}\vec{G}\Sigma_m \end{aligned} \tag{4.5}$$

μ_d is the seismic response of μ_m, and $\Sigma_{d,m}$ is the covariance matrix between observations and logarithmic parameters and observations. d_{obs} is the sample observed from seismic.

4.3.1 Computational Costs

A naive implementation of the model is upper bound by finding by matrix inverse Σ_d^{-1}. By using Gauss-Jordan elimination the complexity of finding the matrix inverse is $O(n^3)$. By transforming 4.5 to the Fourier domain the problem is reduced to solving independently for each frequency component. The inversion in the Fourier domain is performed on much smaller matrices of size $numberofangles * numberofangles$. All the matrix inverse can therefore be performed in $O(n)$ time. The whole process is upper bound by the 3D Fourier transform. It takes in $O(n * log(n))$ time to perform fast Fourier transform. Fourier transform is known to have optimized implementations on the major hardware platforms. These advantages have an impact on how the scientists integrate the inversion in the workflow. Faster estimation allows them to process more data to get more accurate results.

Chapter 5

Performance Optimizations of CRAVA

This chapter describes different optimizations to CRAVA. Then follows a description of improvements. Each improvement contains the motivation, and details of implementation of that improvement.

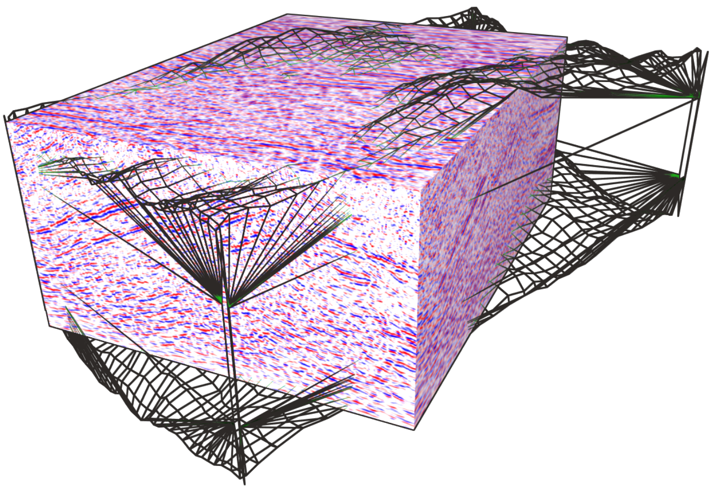


Figure 5.1: Sampling source file shown as seismic cube. Volume of interest shown as wire frame.

5.1 Resampling

The assumption of a multi-normal distribution requires that sampled with a consistent 2 times derivative sampling function. CRAVA does resampling by an orthogonal tricubic interpolation method. Sampling cubic is a compromise between noise in the Fourier domain often related to low order polynomial interoperation, and increasingly heavy computations involved with higher order polynomial interpolation. The cubic interpolation function is performed for all grid cells in the sampling grid. Every grid cell in seismic data has a position (x, y, t) and a unique index (i, j, k) in the sample grid. All grid cells are equally spaced. Therefore a sample point $s_{i,j,k}$ has a position relative to the lower left corner of the seismic cube given by

$$(x, y, t) = \left(\frac{ix}{n_i} + x_0, \frac{jy}{n_j} + y_0, \frac{kt}{n_k} + z_0 \right) \tag{5.1}$$

The sampler function applied in CRAVA is defined as

$$\begin{aligned}
d_{obs}(i, j, k) &= ax^2 + by^2 + ct^2 + dx + ey + ft + g \\
a &= \frac{s_{i-1,j,k} + s_{i+1,j,k} - 2s_{i,j,k}}{2} \\
b &= \frac{s_{i,j-1,k} + s_{i,j+1,k} - 2s_{i,j,k}}{2} \\
c &= \frac{s_{i,j,k-1} + s_{i,j,k+1} - 2s_{i,j,k}}{2} \\
d &= \frac{s_{i+1,j,k} - s_{i-1,j,k}}{2} \\
e &= \frac{s_{i,j+1,k} - s_{i,j-1,k}}{2} \\
f &= \frac{s_{i,j,k+1} - s_{i,j,k-1}}{2} \\
g &= s_{i,j,k}
\end{aligned} \tag{5.2}$$

5.1.1 Implementation

Along the border of the volume some grid cells $s_{i\pm1,j,k} s_{i,j\pm1,k} s_{i,j,k\pm1}$ will be missing. CRAVA treats missing grid cells with in with complex sampling logic. This logic is listed in 5.1.1 in order or descending priority.

CRAVA applies a volume of interest to all seismic data. The volume of interest is designed to limit the seismic volume that is needed for estimation or inversion. In petrol engineering one limits the scope of calculations by defining the surfaces. The volume of interest is defined by a bounding rectangle with a top and bottom cap defined by a heightmap. The area of inversion is the intersection of the volume of interest and the seismic data named the inversion volume.
The sampler can treat the outside volume in three different fashions depending

Inside :
If the grid cells are available the sampler remains unchanged.

Outside :
If the grid cell $s_{i,j,k}$ is outside the volume the value is specificed by the outside mode.

Border :
If $s_{i\pm1,j,k}$, $s_{i,j\pm1,k}$ or $s_{i,j,k\pm1}$ are outside the volume these terms are set to 0.
If $s_{i+1,j,k}$ is outside the volume that term is set to $s_{i-1,j,k}$
If $s_{i,j+1,k}$ is outside the volume that term is set to $s_{i,j-1,k}$
If $s_{i,j,k+1}$ is outside the volume that term is set to $s_{i,j,k-1}$
If $s_{i-1,j,k}$ is outside the volume that term is set to $s_{i+1,j,k}$
If $s_{i,j-1,k}$ is outside the volume that term is set to $s_{i,j+1,k}$
If $s_{i,j,k-1}$ is outside the volume that term is set to $s_{i,j,k+1}$

on an outside mode given as input parameter to the sampler. The outside mode is constant per grid.

```
begin
    for (i, j, k) Input: (n_i, n_j, n_k)
    do
        (x, y, t) ← getCordinates(i, j, k)
        d_obs ← d_obs ∪ sampleFunction(x, y, t, mode)
    end
end
```

Algorithm 1: Old approach for sampling grids with per sample outside mode. The outside mode is handled per sample as a parameter to the sampling function.

- Samples outside inversion volume returns 0.0.
- Samples outside inversion volume returns a value outside the normal range.
- Samples outside inversion volume returns the value closest along the time axis inside inversion volume.

The old sampling function performs necessary boundary testing against volume of interest and seismic data. Doing boundary testing per sample in the sampling function gives suboptimal performance. With an overall knowledge of the problem it is possible to do boundary checks per inversion volume. Combined with different sampler functions for areas inside, outside and along the border for the inversion volume removes the need for boundary checks per sample.
The first step is to detect the borders of the inversion volume. When the borders position of the inversion volume has been detected it is necessary to expand the border grid cells adjacent to the border. Border detection is done by expanding

the top and bottom grid cell along the six closest grid cells perpendicular to the x, y and t axis. There are a total of four borders in a inversion volume. These are two borders between volume of interest and the seismic data, and two along the edges of the seismic data.
With all borders detected the algorithm iterate over all volumes and apply the correct sampling function for each volume. Sampling inside and outside the inversion volume is performed in parallel. The level of parallelism is per sample on OpenMP version 3.0 and per x axis on older versions.

Outside Mode

Since the outside mode is constant per grid it introduces unnecessary complexity to check outside mode per sample. Removing the outside mode totally is unfortunately not an option. Because the revised sampler now sets the outside mode once before sampling the grid. The sample function just performs the outside mode that is set for the sample class. This saves three branches in the sampler.

```
begin
    setOutsideMode(mode)
    for (i, j, k) Input: (n_i, n_j, n_k)
    do
        (x, y, t) ← getCordinates(i, j, k)
        outputGrid ← outputGrid ∪ sampleFunction(x, y, t)
    end
end
```

Algorithm 2: New approach for sampling grids. The outside mode is handled per grid with a statefull sampling function.

5.2 Seismic Inversion

The initial profiling results showed that seismic inversion of low importance for optimization. It is not a time-consuming processing step compared to the others. The reason why there still was put focus on seismic inversion is because of its general importance. While all other processing steps might change significant in nature in different implementations the seismic inversion model CRAVA applies would have to performed the same way in all implementations.
The parallel features of seismic inversion are best understood by analyzing dependencies in 4.5. Solving equation 4.3 involves a matrix inversion. This matrix inversion is, as earlier explained, most efficiently solved in the Fourier domain. The error term e is given by the prior distribution of m and e which has to be calculated. CRAVA calulates e in the Fourier domain. Because the prior error model e is time dependant this dependency is also present in the Fourier domain.

In this thesis, this dependence was maintained by serial error calculation.

Data: E_t time dependant error Fourier transformed.
Data: t time value in range of valid time values T Fourier transformed.
Result: D_{obs} observed sample points d_{obs} Fourier transformed.

begin
    $D_{obs} \leftarrow \emptyset$
    for t **Input**: T
    do
        $E_t \leftarrow$ calculateError(t)
        parallel for $D \leftarrow$ **Input**: D_{obs} *sharing* E_t
        do
            Perform calcuation with $E_{f,g}$ and D
        end parallel for
    end
end

Algorithm 3: Parallel seismic inversion approach.

A technical difficulty is that CRAVA is designed to be able to read and write working copies of grids to secondary storage. This feature compromises speed for the benefit of less main memory usage. At the time of writing hard disks are the major source of secondary memory. Hard disk introduces delay of I/O. I/O costs are the sum of the time of moving hard disk head, rotate disk into position and data transfer. Time of moving the hard disk head and rotating data into position is called seek time and rotation delay respectively. The sum of seek time and rotational delay is access time. An average hard disk has access times in an order of milliseconds [7]. With a clock rate of 2GHz that would mean that access times alone would be equivalent to 12 000 000 clock cycles that are wasted in case of a stall. It is therefore highly favourable to cut access time as much as possible. It is reasonable to assume a scheduler that cuts down time on sequential writes to a minimum. Sequential I/O can not be performed in parallel. However since there is no dependency between different computations these steps can be overlapped. Overlapping I/O and computation is best performed through computation pipelining [13]. Computation pipelining is just the same technique applied to computation task as used on instructions in a CPU. Pipelining exploits that a set of sequential task to be performed in parallel by splitting them into sub tasks. 5.2 shows a simple pipeline with sub tasks read R_n, computation C_n and write W_n. Read and write has to be performed sequentially. we notice that two sequential sub tasks per task requires two points of synchronization.
Seismic inversion has the same sub tasks as 5.2. In sub task R_n temporary variables are read. Then follows one sub task C_n with seismic inversion. The last sub task W_n writes back the result.
A dependency graph between sub tasks would show that reads are dependent on previous reads. The same is the case with writes.

It is reasonable to assume a constant I/O cost for writing and reading. Since there are no $R_n \rightarrow W_{n-1}$ dependency reads and writes can be overlapped. As shown in figure 5.2 average sequential time of for one iteration is the highest

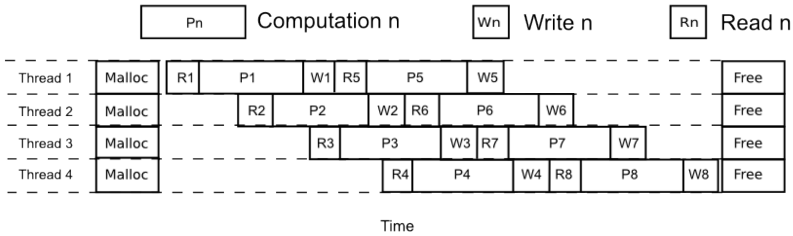


Figure 5.2: Computation pipelining: R_n: Read block n, C_n compute block n, W_n: write block n.

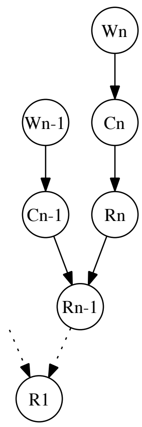


Figure 5.3: Dependency graph for seismic inversion sub tasks.

of the R_n and W_n. Assuming I/O and computation time is constant, and computation is performed in parallel with p processes, the computation the speedup is.

$$
\begin{aligned}
t_{I/O} &= max\left(t_R, t_W\right) \\
t_p &= max\left(t_{I/O}, \frac{t_c}{p}\right) \\
t_s &= \left(t_{I/O} + t_c\right) \\
s(p) &= \frac{t_s}{t_p} \\
s(p) &= \frac{t_{I/O} + t_c}{max\left(t_{I/O}, \frac{t_c}{p}\right)}
\end{aligned}
\tag{5.3}
$$

If $t_{I/O} > t_c$ the task is I/O bound. With infinitely many processors $s(p)$ gives a theoretical speedup limit of

$$
\lim p \rightarrow \infty s(p) = \frac{t_{I/O} + t_c}{t_{I/O}} \tag{5.4}
$$

5.4 also shows that $s(p)$ grows linear until $t_{I/O} = \frac{t_c}{p}$. That is a clear upper limit for speedup. Increasing the number of processors after the point where $t_{I/O}$ equals t_c will lead to more time spent waiting for sequential I/O. Care should be taken about the upper bound 5.4 when implementing this approach.

Spinlocks are used to enforce sequential I/O for reads and writes. OpenMP allows to enforce serial execution of blocks within parallel parts by using the `ordered` directive. Unfortunately according to specification [14] only up to one `ordered` can be performed per iteration. With both reading and writing in serial for each iteration it would be hard to maintain a single `ordered` block.
Instead of allocating a single global memory large enough to for all threads this implementation uses `threadprivate` memory. `threadtprivate` memory is the only way to keep values between multiple parallel blocks. `threadprivate` was introduced with OpenMP version 3.0. For old compilers without version 3.0 seismic inversion falls back to serial execution.

```
int wCnt = 0;
/* specifyig a round robin scheduler is needed to
 * ensure that each thread "gives away" access to the next
 * thread by increasing wCnt.
 */
#pragma omp parallel for schedule(static, 1)
for(int i = 0; i < nxp_ * nyp_; i++){
  /* ordered read and computations */

  while(wCnt != traceNr); /* infinate loop waits for wCnt */
  /*  orderd write */
  wCnt = wCnt + 1
}
```

Listing 5.1: Example of a spinlock.

5.3 FFTW Settings Store

Throughout the program Fourier and inverse Fourier transform is performed numerous places using the software package FFTW. FFTW is one of the fastest general implementations of Fourier transform on modern computers [15]. FFTW applies different solvers for different Fourier Transform properties. For larger Fourier transforms it divides the case into a combination of smaller ones. By dividing a case into smaller cases FFTW is able to choose the solver that exploits the memory hierarchy of the computer it is running on in a best possible manner. However the particular case of Fourier transform might not be known before run time. FFTW calculates the best particular known approach to solve the Fourier transform at run time which is a time consuming operation.

```
/* 1D Fourier transform. The alternative is
 * 3D Fourier transform.
 */
fftwnd_plan settings;
settings = fftw_create_plan(1, width, FFTW_FORWARD, FFTW_ESTIMATE);

/* One of many Fourier transforms with the settings just
 * created.
 */
fftw_one_real_to_complex(settings, real_data, complex_data);

fftw_destroy_plan(plan);
```

Listing 5.2: Typical use scenario for FFTW.

Because generating approaches is so time consuming every program and should then reuse the plan FFTW generates. CRAVA did not reuse plans. With knowledge of the particular problem CRAVA solves was natural to assume that only a handful different settings were used throughout the program. To solve this problem of redundant plan calculation all FFTW calls were replaced with calls to a wrapper class. The wrapper class would perform the Fourier transforms with the smallest number of plans needed by reusing old settings when a particular similar transform had been performed before. Since CRAVA is a multithreaded application the wrapper would have to be thread safe[1]. For a pseudo code of implementation see 4.

[1] A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads.

```
FFTSettings contains width, height, depth and precision
FFTResult contains a FFTSetting and a mutex

Data: FFTSettings s
Result: The settings and one mutex result
lookupSettings begin
    result ← find(storage, s)
    if result = ∅ then
        lock(writeMutex)
        result ← find(storage, s)
        if result = ∅ then
            result ← FFTW_result(s)
            insert(storage, result)
        end
        unlock(writeMutex)
    end
end

Data: FFTSettings s, complex values c
Result: The real values r
InverseFouriertransform begin
    result ← ∅
    while result = ∅ do
        result ← lookupSettings(s)
    end
    lock(result)
    c ← FFTW_complexToReal(result, r)
    unlock(result)
end

Data: FFTSettings s, real values r
Result: The complex values c
Fouriertransform begin
    result ← ∅
    while result = ∅ do
        result ← lookupSettings(s)
    end
    lock(result)
    r ← FFTW_realToComplex(result, c)
    unlock(result)
end
```

Algorithm 4: FFTW settings store algorithm.

The algorithm performs the Fourier transform for a given setting S by waiting until the plan for that mutex has been released or created. Therefore the number of simultaneous Fourier transforms that equals the number of unique settings. Thread safety is handled by locking resources. To keep the storage consistent without duplicates adding a new FFTW plan requires the write mutex. It is vital that the FFTW settings storage is significantly faster than generating new plans each time.

Chapter 6

Results and Future Work

In this thesis, we have looked at performance improvements to CRAVA with special attention on seismic inversion and parallel programming on multi-core CPUs. we have highlighted challenges of different implantations and how we solve these. This chapter tests scalability of the improved CRAVA in terms of memory usage and parallel speedup. We compare the results with theoretical models. We also predict scalability on future hardware.

6.1 Resampling

The resampler process was improved on three different levels.

- *Better sampling* By dividing the sampling area into different volumes with different complexity. A division of the sampling volume removed unnecessary border checks.
- *Code efficiency* By removing unnecessary branching and calculations and pre calculating partial results. To some extent calculations could also be completely removed without affecting the end result.
- *Parallel sampling* By removing dependencies and performing sampling in parallel.

To find out how much domain division and code efficiency affects performance we added timing to all sampler functions took average time spent in each function on a 4 core system 6.1.1. The number of active cores was limited to one to remove threading effects. The results are surprisingly clear. Low level code efficiency is hard to achieve in precise. Most of the work focus had been on removing branches. The results reveals that modern X86 CPUs has so good branching algorithms that these optimizations make marginal, if any, difference. In our case it made a difference of 9% on the timing results. Along the border the

Name	time (ns)	% improvement
Original sampler	0.192320	0.0
Inside sampler	0.175631	9.5
Border sampler	0.237923	-19.17
Outside sampler	0.000818	235.11

Table 6.1: Average timing results of sampler functions on system 6.1.1 with 1 active core.

revised version is producing the exact same results as the original sampler. In this volume the performance went down by 19%. It can be explained from major refactring that was made to the old sampler. The new sampler uses multiple layers of functions doing specific sub tasks of sampling. These small changes ends making up making a negative difference.
Sampling only in the inside inversion volume ended up being the significant factor on serial performance. The bechmarks shows over 200% improvement. We notice that timing results outside the volume are so small that it is impossible to measure accurate. However we can conclude is that sampling outside the volume is orders of magnitude faster than inside the volume of interest.
What difference sampling makes depends on how much of the inversion volume falls within the volume of interest. In addition other effects will remove some of the improvement trends. Every point in the volume of interest has to be converted to UTM-coordinates and further to cell grid indices in the source file. Therefor timing the implementation is the only reliable solution. Pure serial improvement was tested by running the benchmark test with 1 active core. The results shows that serial seismic inversion had changed from 4405.00 to 2764.00 seconds. That is a 59% improvement in wall time.

Name	CPU time	wall time
Resampling old	4402.72	4405.00
Resampling new	2764.70	2764.00

Table 6.2: Overall serial improvements on resampling for the benchmark test on system 6.1.1.

6.1.1 Parallel Scalability

Parallel scalability was tested by running the benchmark test increasing the number of active processor cores. The results show close to linear scalability for the first 5 cores. At 6 and 7 cores the wall time is increasing to somewhere between the performance of 4 and 5 active cores. Running the test again with 7 active processors gave the same trend. We therefore conclude that on this specific combination of software, hardware and tests there is some effect that makes resampling scale worse than expected with 6, and 7 active cores. We speculate in that there might be some disadvantageous cache pattern.
Amdahl's law gives us a parallel fraction f between 0.89 and 0.97. Applying an

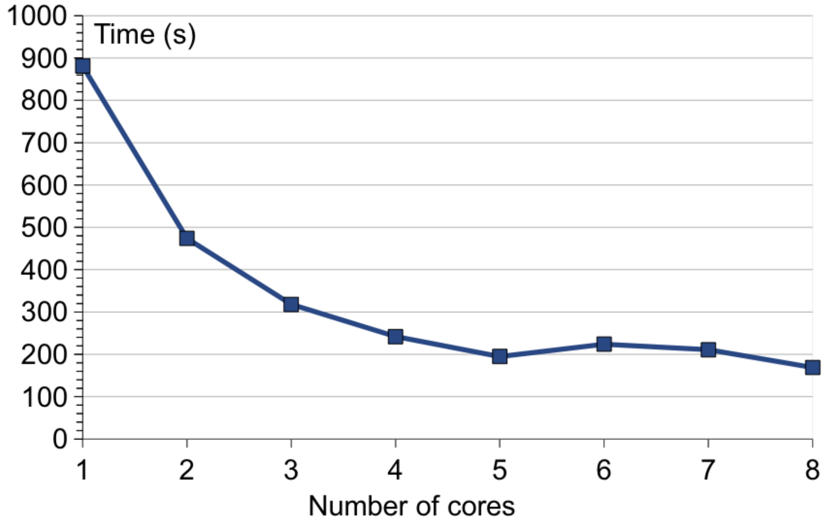


Figure 6.1: Resampling timings on 6.1.1.

arithmetic average of the parallel fractions gives a new f' Solving Amdahl's law with f' suggests a maximum speedup $\lim p \to \infty s(p) = 13.9$.

Cores	time (s)	$s(p)$	f
1	881	1	N/A
2	474	1.86	0.92
3	318	2.77	0.96
4	242	3.64	0.97
5	195	4.52	0.97
6	224	3.93	0.89
7	211	4.18	0.89
8	169	5.21	0.92

Table 6.3: Resamping timings and parallel fractions on 6.1.1

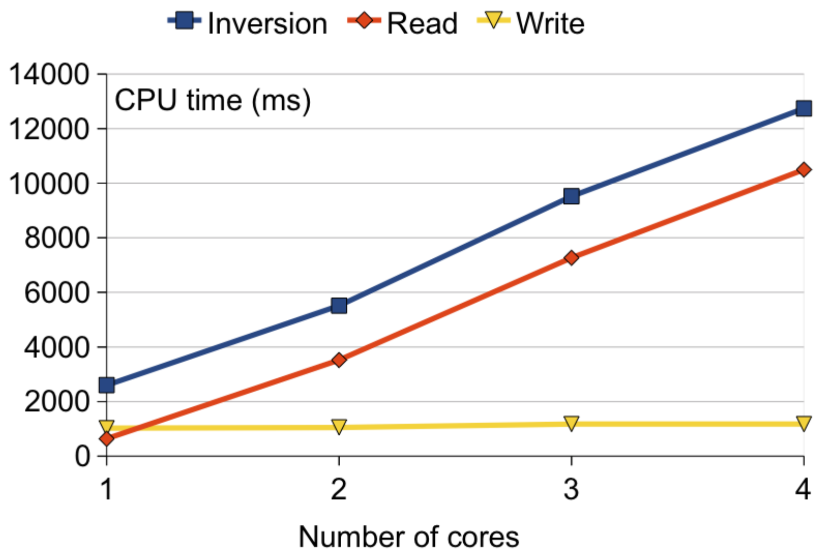


Figure 6.2: Parallel Seismic Inversion CPU time without error calculation.

Name	HPC20
Operating System	Linux kernel 2.6.32-23-generic
OpenMP version	3.0
Compiler	GCC version 4.4.1 (Ubuntu 4.4.1-4ubuntu9)
CPU	Intel Core i5 2.67GHz
Memory	4GB

Table 6.4: 4 core system at NTNU type workstation.

Name	tesla.idi.ntnu.no
Operating System	Linux
OpenMP version	3.0
Compiler	GCC version 4.4.1 (Ubuntu 4.4.1-4ubuntu9)
CPU	Intel Core i7 3.20GHz
Memory	16GB

Table 6.5: 4 core system at NTNU type cluster node.

Name	tr-lx627804
Operating System	Linux
OpenMP version	2.5
Compiler	GCC version 4.1 2071124 (Red Hat 4.1.2-42)
CPU	AMD Opteron 2216 2.4GHz
Memory	32GB

Table 6.6: 2 core system at Statoil type workstation.

6.2 Seismic Inversion

Chapter 5.2 introduced computation pipelining, and showed how it can be applied to seismic inversion. Speedup was benchmarked on a 4 core system 6.1.1[1]. Since the speedup model we introducesd in chapter 5.2 depends on wall times of each sub task we chose to time these tasks.

The results exposes an interesting effect. The sequential blocks enforces correct ordering sub tasks. When the number of cores scales up the time per thread used on reading increases proportional with the number of threads. The end result is a process that runs without overlapping computation at the same speed as serial execution.
According to the specification only the ordered block would execute sequentially but is not concistent with the results. This is also the root of the problems we experienced. With a declarative programming model the programmer relies on software to perform the task he wants on a much larger scale than imperative

[1]None of the systems available for running the benchmark test the necessary GCC 4.4

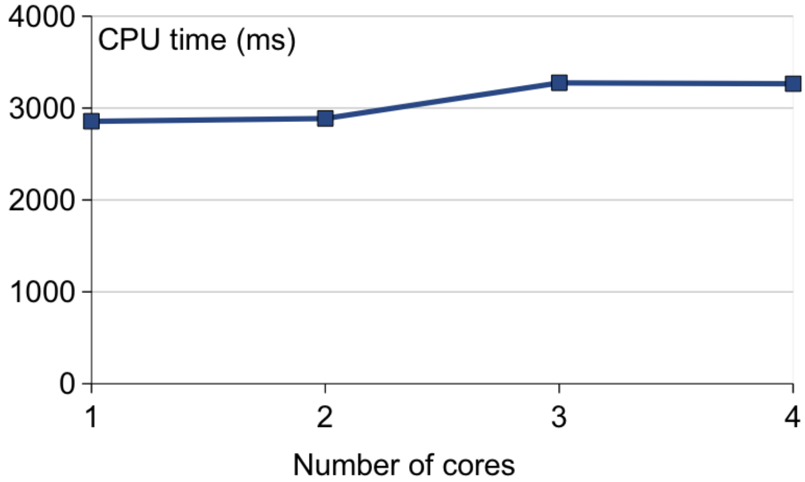


Figure 6.3: Parallel Seismic Inversion wall time.

programming. Most of technical details are hidden for the programmer. Whenever these hidden details does not perform as expected one are enforced to try another approach. It does not help that parallelism is easily turned on and off if the problem only occurs when the program runs in parallel.

6.2.1 Parallel Scalability

Even though we were not able to implement a fully functional version of a pipelined seismic inversion model it is possible to calculate how speedup ideally scales. According to the speedup model for inversion 5.3 parallel speedup hits a roof line when time spent on I/O passes time spent on computation. Based on the timing results for a single core upper limit of seismic inversion.

Name	Eddie
Operating System	Red Had Enterprise Linux Server release 5.4
OpenMP version	2.5
Compiler	GCC version 4.1.2 20080704 (Red Hat 4.1.2-46)
CPU	AMD Opteron 8216 2.4Ghz
Memory	128GB

Table 6.7: 8 core system at Statoil type cluster node.

Process	Time (ms)
R_n	1.085034
W_n	1.744898
$t_{I/O}$	1.744898
C_n	4.425170
t_c	4.425170

Table 6.8: Average CPU time per sample used on seismic inversion on system 6.1.1 of A.2 scaled up to 588 samples per trace.

Test name	Fourier transforms	Time cached (ns)	Time no cached (ns)
Corr. test 7	75505	0.1253021	15.6696014

Table 6.9: Time usage of FFTW plan generation on selected correctness tests on system 6.1.1.

$$\begin{aligned} s_p &= \frac{t_{I/O} + t_c}{t_{I/O}} \\ \lim p \to \infty s_p &= 5.42517 \end{aligned} \tag{6.1}$$

For the Intel Core i5 CPU that were benchmarked it is clear that the assumption that computation of seismic inversion should be much larger than I/O had been too optimistic. With the timings on this system the theoretical speedup stalls with six CPU cores. When taking into account that we in addition perform serial error calculation the speedup would be even lower. This task does simply not contain enough computation to scale as a linear function, in fact at all, with more than 5 processors.

6.3 FFTW Settings Store

The FFTW setting store was tested by turning caching of FFTW plans completely on and off. The number of Fourier transforms and time usage of the different approaches were logged. Two out of three initial assumptions were right. Generation of FFTW plans is a slow task that is possible to share significantly faster than it is possible to create. Maximum six unique FFTW plans occurs in CRAVA. The problem is that there is not performed enough Fourier transforms for plan generation to become a time consumer. When scaling up to real size problems it is natural to expect that the use of Fourier transforms would scale up with the problem size. The problem is that the number of Fourier transforms for large projects does not scales faster than the project size. The full scale benchmark executes Fourier and inverse Fourier transform 11430959 times. On the quad core that would take 179 seconds or 2 minutes and 59 seconds. For a program that runs for over an hour that is insignificant.

The time used on wrapping FFTW plans could have been more effective elsewhere. The challenge is to come up with that result. The conclusion is therefor. Understand what is meant by scalability for this project, and how does it affect proportions of the program.

Chapter 7

Conclusions and Future Work

The slow low level border sampler showed the importance showed the importance of the details when doing large structural changes, and the FFTW store showed us the importance of keeping the overall perspective when doing microlevel improvements. The conclusion is therefore that all program optimizations are connected. Algorithmic changes has a great potensial for improvement, vut is easily destroyed by a bad micro level code. Low level optimisations are easy to impement, but might be irrelevant when looked at in a context e.g. the inside volume sampler. We think the solution lies in an incremental process with benchmarking, evaluation and development. But profiling everything of a program is not the solution. Tool assisted program profiling can often be at least ten times as slow as ordinary program execution. When a profiling allready takes over an hour any extensive test is unrealistic. But profiling is done on top level process steps lacks the detail to explain small changes. Especially if they are evenly distributed along the larger. The methodology should be to do the highest level of profiling first followed by decending level of granularity. As the the granularity grows oue have to doe qualified choices of what and how to profile. Alonside profiling improvements should be done to the current bottlenetck in each step.

7.1 Comparison of Serial and Parallel Optimization

An important of parallel programming consists in distributing dependent values between threads. Dependency checking is a manual process that requires an understanding of the problem which lends itself to serial improvements. Therefore parallel and serial improvements are related, is best performed simultaneously. Instead it is easy to compare serial and parallel in terms of results. Adding parallelism gave 5.5 times speedup. Compared to the old 59 % speed improvement of serial execution. The parallel parts might not have hit their theoretical limit and might speedup even more on better hardware. In the other hand the serial improvements delivers the same improvement on all hardware. Another metric to compare serial and parallel programming is ease of implementation. We think that parallel programming is much more challenging than serial programming. Bad programming with use of common state and not re-entrant code can make parallel programming practically impossible. All work during this project has shown that parallel programming is more sensitive to bad coding. In pracice parallel programming enfoces a strict set of rules for good programming style with correct encapsulation and no hidden state. Stateful code suffers state explosion, and is hard to maintain in a parallel paradigm.

This leads to the conclusion that parallel programming is a great tool, but as any other optimization, it should be used when it is the fastest way to lower wall time.

7.2 Future Work

With parallel seismic inversion and resampling it is natural that the next step to be the stochastic model process. Building stochastic model is the new largest processing step. One can continue to improve processes as long as speed imrovements legitimate further development.

7.2.1 Seismic Inversion

There is still a potential for further speedup of seismic inversion. A natural first step would be to move error calculation out as a sub task prior to the inversion. It would require an extra amount of memory equal to the number of grid cells along x and y axis. The benefit would be a larger parallel fraction in the inversion.
CRAVA is is limited by ineffective I/O. Instead of simple getter and setter functions around memory and disk there might be possible to group multiple values together, or even use OS-dependent I/O to speed up implementation. A good cache analysis can reveal optimal read patterns for more efficient use of cache can also be studied.
There is possible with a combination where only every Nth iteration performs reads and writes. If serial reads and writes are performed every Nth iteration all other iterations would read and write to main memory, and can therefore skip N times as many sequential blocks as the current version.

Bibliography

[1] P Dahle, B. Fjellvoll, R. Hauge, O. Kolbjørnsen, A.R. Syversveen, and M. Ulvmoen. CRAVA User Manual version 0.9.8, March 2010. 1

[2] Arild Buland, Odd Kolbjørnsen, and Henning Omre. Rapid spatially coupled AVO inversion in the Fourier domain. *Geophysics*, 68(3):824–836, 2003. 1, 16

[3] OpenMP ARB. OpenMP.org » OpenMP Compilers. World Wide Web electronic publication, August 2010. 3, 9

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM. 6

[5] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 105, New York, NY, USA, 2006. ACM. 8

[6] William Gropp. *Using Mpi-2.* MIT Press, Cambridge, 1999. 8

[7] William Stallings. *Operating Systems.* Pearson Prentice Hall, Upper Saddle River, 2009. 8, 23

[8] Rohit Chandra. *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers, San Francisco, 2001. 10

[9] J. Pendrel. Seismic Inversion:A Critical Tool in Reservoir Characterization. *Scandinavian Oil-Gas Magazine*, No. 5/6:19–22, 2006. 15

[10] S.K. Upadhyay. *Seismic reflection processing: with special reference to anisotropy.* Springer, Berlin, 2004. 15

[11] K. Aki and P. G. Richards. *Quantitative Seismology: Theory and Methods.* W. H. Freeman and Co., 1980. 15

[12] R. H. Stolt and A. B. Weglein. Migration and inversion of seismic data. *Geophysics*, 50:2458–2472, 1985. 15

[13] Barry Wilkinson. *Parallel Programming.* Pearson/Prentice Hall, Upper Saddle River, 2005. 23

[14] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 2008. retrieved at: 21. August 2010 18:00 GMT+1. 25

[15] Matteo Frigo and Steven G. Johnson. FFTW: An Adaptive Software Architecture For The FFT. pages 1381–1384. IEEE, 1998. 26

Appendices

Appendix A

Tests

A.1 Benchmark Test

Reservoir source	Snorre oil field
Estimation	No
Surface format	.storm
Number of layers	350
Output format	.storm
Number of seismic angles	3
Seismic angles	10°, 20°, 30°
Wells	
Number of wells total	7
Number of wells used	3
Number of wells used for Synthetic Vs	1
Well format	RMS
Seismic	
Grid dimensions(x*y*z)	$216 * 189 * 420$
padding ratio (x/y/z)	0.0/0.0/0.2
Seismic angles	0°, 10°, 20°
Total data size	10.9 GB
Seismic data format	SEG Y
Wavelets:	
Number of wavelets	3

Table A.1: Benchmark tests most important characteristics.

A.2 Correctness Test

Name	two cubes forward modelling
Estimation	No
Grid dimensions(x*y*z)	$15 * 15 * 500$
Kriging	No
Seismic angles	0°, 10°
Number of Wavelets	2
Number of Wells	0
Name	two cubes one well pred nokrig
Estimation	Yes
Grid dimensions(x*y*z)	$15 * 15 * 250$
Kriging	No
Seismic angles	16°, 28°
Number of Wavelets	2
Number of Wells	1
Name	two cubes one well pred krig
Estimation	No
Grid dimensions(x*y*z)	$15 * 15 * 500$
Kriging	Yes
Seismic angles	16°, 28°
Number of Wavelets	2
Number of Wells	1
Name	two cubes one well sim krig
Estimation	No
Grid dimensions(x*y*z)	$15 * 15 * 250$
Kriging	Yes
Seismic angles	16°, 28°
Number of Wavelets	2
Number of Wells	1

Table A.2: Correctness tests most important characteristics.

Name	two cubes four wells pred nokrig
Estimation	No
Grid dimensions(x*y*z)	$112 * 112 * 280$
Kriging	No
Seismic angles	16°, 28°
Number of Wavelets	0
Number of Wells	4
Name	two cubes four wells pred krig
Estimation	No
Grid dimensions(x*y*z)	$112 * 112 * 250$
Kriging	Yes
Seismic angles	16°, 28°
Number of Wavelets	2
Number of Wells	4
Name	two cubes four wells pred nokrig corr dir
Estimation	No
Grid dimensions(x*y*z)	$112 * 112 * 375$
Kriging	Yes
Seismic angles	16°, 28°
Number of Wavelets	2
Number of Wells	4
Name	two cubes four wells pred nokrig localwavelet
Estimation	No
Grid dimensions(x*y*z)	$15 * 15 * 200$
Kriging	No
Seismic angles	16°, 28°
Number of Wavelets	2
Number of Wells	4
Name	two cubes one well faciesprob
Estimation	No
Grid dimensions(x*y*z)	$15 * 15 * 500$
Kriging	No
Seismic angles	0°, 20°
Number of Wavelets	2
Number of Wells	1

Table A.3: Correctness tests most important characteristics continued.

Appendix B

Code

B.1 Resampling

```
#ifndef FFTGRID_H
#define FFTGRID_H

#include <assert.h>
#include <string>

#include "fft/include/fftw.h"
#include "definitions.h"

class Corr;
class Wavelet;
class Simbox;
class RandomGen;
class GridMapping;

class FFTGrid{
public:

  FFTGrid(int nx, int ny, int nz, int nxp, int nyp, int nzp);
  FFTGrid(FFTGrid * fftGrid);
  FFTGrid() {} //Dummy constructor needed for FFTFileGrid
  virtual ~FFTGrid();

  void setType(int cubeType) {cubetype_ = cubeType;}
  void setAngle(float angle) {theta_ = angle;}

  int                  fillInFromSegY(SegY * segy, Simbox *simbox,
      bool nopadding = false );              // No mode

  int                  fillInFromStorm(Simbox             *
      actSimBox,
                                        StormContGrid     * grid,
                                        const std::string & parName,
                                        bool                scale =
                                            false,
                                        bool
                                            nopadding = false);
```

```
                                          // No mode
  void                  fillInConstant(float value);              // No mode
  fftw_real*            fillInParamCorr(Corr* corr,int minIntFq,
                                         float gradI, float gradJ);// No mode
  void                  fillInErrCorr(Corr* parCorr,              // No mode
                                       float gradI, float gradJ);  // No mode
  virtual void          fillInComplexNoise(RandomGen * ranGen);   // No mode/randomaccess

  void                  fillInTest(float value1, float value2);   // No mode /DEBUG
  void                  fillInFromArray(float *value);

  virtual fftw_complex getNextComplex() ;                        // Accessmode read/readandwrite
  virtual float         getNextReal() ;                           // Accessmode read/readandwrite
  virtual int           setNextComplex(fftw_complex);             // Accessmode write/readandwrite
  virtual int           setNextReal(float);                       // Accessmode write/readandwrite
  float                 getRealValue(int i, int j, int k, bool extSimbox = false); // Accessmode randomaccess
  float                 getRealValueCyclic(int i, int j, int k);
  float                 getRealValueInterpolated(int i, int j, float kindex, bool extSimbox = false);
  fftw_complex          getComplexValue(int i, int j, int k, bool extSimbox = false) const;
  virtual int           setRealValue(int i, int j, int k, float value, bool extSimbox = false); // Accessmode randomaccess
  int                   setComplexValue(int i, int j ,int k, fftw_complex value, bool extSimbox = false);
  fftw_complex          getFirstComplexValue();
  virtual int           square();                                 // No mode/randomaccess
  virtual int           expTransf();                              // No mode/randomaccess
  virtual int           logTransf();                              // No mode/randomaccess
  virtual void          realAbs();
  virtual int           collapseAndAdd(float* grid);              // No mode/randomaccess
  virtual void          fftInPlace();                             // No mode/randomaccess
  virtual void          invFFTInPlace();                          // No mode/randomaccess


  virtual void          add(FFTGrid* fftGrid);                    // No mode/randomaccess
  virtual void          subtract(FFTGrid* fftGrid);
                          // No mode/randomaccess
  virtual void          changeSign();                     // No mode/randomaccess
  virtual void          multiply(FFTGrid* fftGrid);               // pointwise multiplication!
  bool                  consistentSize(int nx,int ny, int nz, int nxp, int nyp, int nzp);
```

```
int                   getCounterForGet() const {return(
    counterForGet_);}
int                   getCounterForSet() const {return(
    counterForSet_);}
int                   getNx()     const {return(nx_);}
int                   getNy()     const {return(ny_);}
int                   getNz()     const {return(nz_);}
int                   getNxp()    const {return(nxp_);}
int                   getNyp()    const {return(nyp_);}
int                   getNzp()    const {return(nzp_);}
int                   getRNxp()   const {return(rnxp_);}
int                   getcsize()  const {return(csize_);}
int                   getrsize()  const {return(rsize_);}
float                 getTheta()  const {return(theta_);}
float                 getScale()  const {return(scale_);}
bool                  getIsTransformed() const {return(
    istransformed_);}
enum                  gridTypes{CTMISSING,DATA,PARAMETER,
    COVARIANCE,VELOCITY};
enum                  accessMode{NONE,READ,WRITE,READANDWRITE,
    RANDOMACCESS};
virtual void          multiplyByScalar(float scalar);       //No
    mode/randomaccess
int                   getType() const {return(cubetype_);}
virtual void          setAccessMode(int mode){assert(mode>=0);}
virtual void          endAccess(){}
virtual void          writeFile(const std::string & fileName,
                                const std::string & subDir,
                                const Simbox      * simbox,
                                const std::string   sgriLabel = "
                                    NO_LABEL",
                                const float         z0        =
                                    0.0,
                                GridMapping       * depthMap  =
                                    NULL,
                                GridMapping       * timeMap   =
                                    NULL,
                                const TraceHeaderFormat & thf  =
                                    TraceHeaderFormat(
                                    TraceHeaderFormat::SEISWORKS))
                                    ;
//Use this instead of the ones below.
virtual void          writeStormFile(const std::string & fileName,
     const Simbox * simbox, bool expTrans = false,
                                     bool ascii = false, bool
                                         padding = false, bool
                                         flat = false);//No mode/
                                         randomaccess
virtual int           writeSegyFile(const std::string & fileName,
    const Simbox * simbox, float z0,
                                    const TraceHeaderFormat &thf =
                                         TraceHeaderFormat(
                                        TraceHeaderFormat::
                                        SEISWORKS));    //No mode/
                                        randomaccess
virtual int           writeSgriFile(const std::string & fileName,
    const Simbox * simbox, const std::string label);
virtual void          writeAsciiFile(const std::string & fileName)
    ;
virtual void          writeAsciiRaw(const std::string & fileName);
virtual void          writeResampledStormCube(GridMapping *
    gridmapping, const std::string & fileName,
```

```
                                                    const Simbox *simbox
                                                    , const int
                                                    format, bool
                                                    expTrans);
virtual void         writeCravaFile(const std::string & fileName,
     const Simbox * simbox);
virtual void         readCravaFile(const std::string & fileName,
   std::string & errText, bool nopadding = false);

virtual bool         isFile() {return(0);}    // indicates wether
     the grid is in memory or on disk

static void          setOutputFlags(int format, int domain) {
   formatFlag_ = format;domainFlag_=domain;};
static void          setOutputFormat(int format) {formatFlag_ =
   format;}
int                  getOutputFormat() {return(formatFlag_);}
static void          setOutputDomain(int domain) {domainFlag_ =
   domain;}
int                  getOutputDomain() {return(domainFlag_);}
static void          setMaxAllowedGrids(int maxAllowedGrids) {
   maxAllowedGrids_ = maxAllowedGrids ;}
static int           getMaxAllowedGrids()   { return
   maxAllowedGrids_   ;}
static int           getMaxAllocatedGrids() { return
   maxAllocatedGrids_ ;}
static void          setTerminateOnMaxGrid(bool terminate) {
   terminateOnMaxGrid_ = terminate ;}
static int           findClosestFactorableNumber(int leastint);


virtual void         createRealGrid(bool add = true);
virtual void         createComplexGrid();

//This function interpolates seismic in all missing traces inside
     area, and mirrors to padding.
//Also interpolates in traces where energy is lower than treshold
     .
virtual void         interpolateSeismic(float energyTreshold = 0)
   ;

void                 checkNaN(); //NBNB Ragnar: For debug purpose
   . Negative number = OK.
float                getDistToBoundary(int i, int n , int np);
virtual void         getRealTrace(float * value, int i, int j);
virtual int          setRealTrace(int i, int j, float *value);
std::vector<float>   getRealTrace2(int i, int j);


static void          reportFFTMemoryAndWait(const std::string &
   msg) {
                       LogKit::LogFormatted(LogKit::HIGH, "%s: %2
                           d grids, %10.2f MB\n", msg.c_str(),
                           nGrids_, FFTMemUse_/(1024.0f*1024.0f))
                           ;
                       float tmp;
                       std::cin >> tmp;
                       LogKit::LogFormatted(LogKit::HIGH, "Memory
                            used %4.0f MB, used outside grid %4.0
                           f MB\n", tmp, tmp-FFTMemUse_/(1024.0f
                           *1024.0f));
                     }
```

```
protected:
  //int                  setPaddingSize(int n, float p);
  int                    getFillNumber(int i, int n, int np );

  int                    getXSimboxIndex(int i){return(getFillNumber(i, nx_, nxp_ ));}
  int                    getYSimboxIndex(int j){return(getFillNumber(j, ny_, nyp_ ));}
  int                    getZSimboxIndex(int k);
  void                   computeCircCorrT(Corr* corr,fftw_real* CircCorrT);
  void                   makeCircCorrTPosDef(fftw_real* CircularCorrT,int minIntFq);
  fftw_complex*          fft1DzInPlace(fftw_real*  in);
  fftw_real*             invFFT1DzInPlace(fftw_complex* in);

  //Interpolation into SegY and sgri
  float                  getRegularZInterpolatedRealValue(int i, int j, double z0Reg,
                                                          double dzReg, int kReg,
                                                          double z0Grid, double dzGrid);

  //Supporting functions for interpolateSeismic
  int                    interpolateTrace(int index, short int * flags, int i, int j);
  void                   extrapolateSeismic(int imin, int imax, int jmin, int jmax);

  /// Called from writeResampledStormCube
  void                   writeSegyFromStorm(StormContGrid *data, std::string fileName);
  void                   makeDepthCubeForSegy(Simbox *simbox,const std::string & fileName);

  int                    cubetype_;           // see enum gridtypes above
  float                  theta_;              // angle in angle gather (case of data)
  float                  scale_;              // To keep track of the scalings after fourier transforms
  int                    nx_;                 // size of original grid in lateral x direction
  int                    ny_;                 // size of original grid in lateral y direction
  int                    nz_;                 // size of original grid in depth (time)
  int                    nxp_;                // size of padded FFT grid in lateral x direction
  int                    nyp_;                // size of padded FFT grid in lateral y direction
  int                    nzp_;                // size of padded FFT grid in depth (time)
```

```

  int                    cnxp_;              // size in x direction for storage inplace algorithm (complex grid) nxp_/2+1
  int                    rnxp_;              // expansion in x direction for storage inplace algorithm (real grid) 2*(nxp_/2+1)

  int                    csize_;             // size of complex grid, cnxp_*nyp_*nzp_
  int                    rsize_;             // size of real grid rnxp_*nyp_*nzp_
  int                    counterForGet_;     // active cell in grid
  int                    counterForSet_;     // active cell in grid

  bool                   istransformed_;     // true if the grid contain Fourier values (i.e complex variables)

  fftw_complex         * cvalue_;            // values of complex parameter in grid points
  fftw_real            * rvalue_;            // values of real parameter in grid points

  static int             formatFlag_;        // Decides format of output (see ModelSettings).
  static int             domainFlag_;        // Decides domain of output (see ModelSettings).

  static int             maxAllowedGrids_;   // The maximum number of grids we are allowed to allocate.
  static int             maxAllocatedGrids_; // The maximum number of grids that has actually been allocated.
  static int             nGrids_;            // The actually number of grids allocated (varies as crava runs).
  static bool            terminateOnMaxGrid_; // If true, terminate when we try to allocate more than maxAllowedGrids.
  bool                   add_;                // Tells whether we should change nGrids_ or not

  static float           maxFFTMemUse_;
  static float           FFTMemUse_;

};
#endif
```

Listing B.1: Old FFTGrid class definition. Contains resampling functionality.

```
#ifndef FFTGRID_H
#define FFTGRID_H

#include "lib/timekit.hpp"
#include "src/timings.h"
#include "src/simbox.h"
#include "fft/include/fftw.h"
#include "src/locks.h"

class Corr;
class Wavelet;
class Simbox;
class RandomGen;
class GridMapping;

#define AT2D(a, b, width) ((a)*(width) + (b))
```

```
//#define PROFILING
#ifdef  PROFILING
#include "src/profiling.h"
#endif

class FFTGrid{
public:
  FFTGrid(int nx, int ny, int nz, int nxp, int nyp, int nzp);
  FFTGrid(FFTGrid * fftGrid);
  FFTGrid() {} //Dummy constructor needed for FFTFileGrid
  virtual ~FFTGrid();
  enum          gridTypes{CTMISSING,DATA,PARAMETER,COVARIANCE,VELOCITY};

  void setType(int cubeType) {cubetype_ = cubeType;}
  void setAngle(float angle) {theta_ = angle;}

  int                 fillInFromSegY(SegY * segy, Simbox *simbox, bool nopadding = false );            // No mode

  int                 fillInFromStorm(Simbox               *actSimBox,
                                      StormContGrid      * grid,
                                      const std::string & parName,
                                      bool                  scale = false,
                                      bool                  nopadding = false);            // No mode
  void                fillInConstant(float value);                     // No mode
  fftw_real*          fillInParamCorr(Corr* corr,int minIntFq,
                                      float gradI, float gradJ);// No mode
  void                fillInErrCorr(Corr* parCorr,                     // No mode
                                   float gradI, float gradJ);   // No mode
  virtual void        fillInComplexNoise(RandomGen * ranGen);    // No mode/randomaccess

  void                fillInTest(float value1, float value2);    // No mode /DEBUG
  void                fillInFromArray(float *value);

  virtual fftw_complex getNextComplex() ;                        // Accessmode read/readandwrite
  virtual float       getNextReal() ;                            // Accessmode read/readandwrite
  virtual int         setNextComplex(fftw_complex);              // Accessmode write/readandwrite
  virtual int         setNextReal(float);                        // Accessmode write/readandwrite
  float               getRealValue(int i, int j, int k, bool extSimbox = false);  // Accessmode randomaccess
  float               getRealValueCyclic(int i, int j, int k);
  float               getRealValueInterpolated(int i, int j, float kindex, bool extSimbox = false);
  virtual fftw_complex getComplexValue(int i, int j, int k, bool extSimbox = false);
  virtual int         setRealValue(int i, int j, int k, float value, bool extSimbox = false);  // Accessmode randomaccess
```

```
virtual int            setRealValueOrPad(int i, int j, int k, float
    value);
virtual int            setComplexValue(int i, int j ,int k,
    fftw_complex value, bool extSimbox = false);
fftw_complex           getFirstComplexValue();
virtual int            square();                              //
    No mode/randomaccess
virtual int            expTransf();                           //
    No mode/randomaccess
virtual int            logTransf();                           //
    No mode/randomaccess
virtual void           realAbs();
virtual int            collapseAndAdd(float* grid);           //
    No mode/randomaccess
virtual void           fftInPlace();                          //
    No mode/randomaccess
virtual void           invFFTInPlace();                       //
    No mode/randomaccess


virtual void           add(FFTGrid* fftGrid);                 //
    No mode/randomaccess
virtual void           subtract(FFTGrid* fftGrid);
                        // No mode/randomaccess
virtual void           changeSign();                 // No mode/
    randomaccess
virtual void           multiply(FFTGrid* fftGrid);            //
    pointwise multiplication!
bool                   consistentSize(int nx,int ny, int nz, int
    nxp, int nyp, int nzp);
int                    getCounterForGet() const {return(
    counterForGet_);}
int                    getCounterForSet() const {return(
    counterForSet_);}
int                    getNx()    const {return(nx_);}
int                    getNy()    const {return(ny_);}
int                    getNz()    const {return(nz_);}
int                    getNxp()   const {return(nxp_);}
int                    getNyp()   const {return(nyp_);}
int                    getNzp()   const {return(nzp_);}
int                    getRNxp()  const {return(rnxp_);}
int                    getcsize() const {return(csize_);}
int                    getrsize() const {return(rsize_);}
float                  getTheta() const {return(theta_);}
float                  getScale() const {return(scale_);}
bool                   getIsTransformed() const {return(
    istransformed_);}
enum                   accessMode{NONE,READ,WRITE,READANDWRITE,
    RANDOMACCESS};
virtual void           multiplyByScalar(float scalar);       //No
    mode/randomaccess
int                    getType() const {return(cubetype_);}
virtual void           setAccessMode(int mode);
virtual void           endAccess(){}
virtual void           writeFile(const std::string & fileName,
                                 const std::string & subDir,
                                 const Simbox       * simbox,
                                 const std::string    sgriLabel = "
                                     NO_LABEL",
                                 const float          z0        =
                                     0.0,
```

```
                                    GridMapping      * depthMap  = NULL,
                                    GridMapping      * timeMap   = NULL,
                                    const TraceHeaderFormat & thf  = TraceHeaderFormat(TraceHeaderFormat::SEISWORKS));
  //Use this instead of the ones below.
  virtual void          writeStormFile(const std::string & fileName, const Simbox * simbox, bool expTrans = false,
                                       bool ascii = false, bool padding = false, bool flat = false);//No mode/randomaccess
  virtual int           writeSegyFile(const std::string & fileName, const Simbox * simbox, float z0,
                                      const TraceHeaderFormat &thf = TraceHeaderFormat(TraceHeaderFormat::SEISWORKS));   //No mode/randomaccess
  virtual int           writeSgriFile(const std::string & fileName, const Simbox * simbox, const std::string label);
  virtual void          writeAsciiFile(const std::string & fileName);
  virtual void          writeAsciiRaw(const std::string & fileName);
  virtual void          writeResampledStormCube(GridMapping * gridmapping, const std::string & fileName,
                                                const Simbox *simbox, const int format, bool expTrans);
  virtual void          writeCravaFile(const std::string & fileName, const Simbox * simbox);
  virtual void          readCravaFile(const std::string & fileName, std::string & errText, bool nopadding = false);

  virtual bool          isFile() {return(0);}    // indicates wether the grid is in memory or on disk

  static void           setOutputFlags(int format, int domain) {formatFlag_ = format;domainFlag_=domain;};
  static void           setOutputFormat(int format) {formatFlag_ = format;}
  int                   getOutputFormat() {return(formatFlag_);}
  static void           setOutputDomain(int domain) {domainFlag_ = domain;}
  int                   getOutputDomain() {return(domainFlag_);}
  static void           setMaxAllowedGrids(int maxAllowedGrids) {maxAllowedGrids_ = maxAllowedGrids ;}
  static int            getMaxAllowedGrids()   { return maxAllowedGrids_   ;}
  static int            getMaxAllocatedGrids() { return maxAllocatedGrids_ ;}
  static void           setTerminateOnMaxGrid(bool terminate) {terminateOnMaxGrid_ = terminate ;}
  static int            findClosestFactorableNumber(int leastint);


  virtual void          createRealGrid(bool add = true);
  virtual void          createComplexGrid();
```

```
  //This function interpolates seismic in all missing traces inside
      area, and mirrors to padding.
  //Also interpolates in traces where energy is lower than treshold
      .
  virtual void          interpolateSeismic(float energyTreshold = 0)
      ;

  void                  checkNaN(); //NBNB Ragnar: For debug purpose
      . Negative number = OK.
  float                 getDistToBoundary(int i, int n , int np);
  virtual void          getRealTrace(float * value, int i, int j);
  virtual int           setRealTrace(int i, int j, float *value);
  std::vector<float>    getRealTrace2(int i, int j);


  static void           reportFFTMemoryAndWait(const std::string &
      msg) {
                          LogKit::LogFormatted(LogKit::HIGH, "%s: %2
                              d grids, %10.2f MB\n", msg.c_str(),
                              nGrids_, FFTMemUse_/(1024.0f*1024.0f))
                              ;
                          float tmp;
                          std::cin >> tmp;
                          LogKit::LogFormatted(LogKit::HIGH, "Memory
                               used %4.0f MB, used outside grid %4.0
                              f MB\n", tmp, tmp-FFTMemUse_/(1024.0f
                              *1024.0f));
                        }
  bool                  GetAllValidIndices(const Simbox& simbox,
      const SegY& segy, int*& lowerTop, int*& topBorder, int*&
      botBorder, int*& upperBot) const;
  virtual bool          allowsRandomWrite() const;
  virtual bool          allowsRandomRead() const;

protected:
  float                 getValueByIndex(const Simbox* simbox, const
      SegY* segy, int i, int j, int k) const;

template <NRLib::OutsideMode borderCases>
  int                   sampleSEGY(SegY * segy, Simbox *simbox, bool
       nopadding = false );               // No mode
  //int                 setPaddingSize(int n, float p);
  static int            getFillNumber(int i, int n, int np );
  static int            getXSimboxIndex(int i, int nx, int nxp){
      return(FFTGrid::getFillNumber(i, nx, nxp));}
  static int            getYSimboxIndex(int j, int ny, int nyp){
      return(FFTGrid::getFillNumber(j, ny, nyp));}
  int                   getXSimboxIndex(int i) const{return(
      getFillNumber(i, nx_, nxp_));}
  int                   getYSimboxIndex(int j) const{return(
      getFillNumber(j, ny_, nyp_));}
  int                   getZSimboxIndex(int k) const;
  void                  computeCircCorrT(Corr* corr,fftw_real*
      CircCorrT);
  void                  makeCircCorrTPosDef(fftw_real* CircularCorrT
      ,int minIntFq);
  fftw_complex*         fft1DzInPlace(fftw_real*  in);
  fftw_real*            invFFT1DzInPlace(fftw_complex* in);

  //Interpolation into SegY and sgri
```

```
float                   getRegularZInterpolatedRealValue(int i, int
    j, double z0Reg,
                                                    double
                                                        dzReg,
                                                     int
                                                        kReg,
                                                    double
                                                        z0Grid
                                                        ,
                                                        double

                                                        dzGrid
                                                        );

//Supporting functions for interpolateSeismic
int                     interpolateTrace(int index, short int *
    flags, int i, int j);
void                    extrapolateSeismic(int imin, int imax, int
    jmin, int jmax);

/// Called from writeResampledStormCube
void                    writeSegyFromStorm(StormContGrid *data, std
    ::string fileName);
void                    makeDepthCubeForSegy(Simbox *simbox,const
    std::string & fileName);
int                     cubetype_;          // see enum gridtypes
    above
float                   theta_;             // angle in angle gather
     (case of data)
float                   scale_;             // To keep track of the
    scalings after fourier transforms
int                     nx_;                // size of original grid
     in lateral x direction
int                     ny_;                // size of original grid
     in lateral y direction
int                     nz_;                // size of original grid
     in depth (time)
int                     nxp_;               // size of padded FFT
    grid in lateral x direction
int                     nyp_;               // size of padded FFT
    grid in lateral y direction
int                     nzp_;               // size of padded FFT
    grid in depth (time)

int                     cnxp_;              // size in x direction
    for storage inplace algorithm (complex grid) nxp_/2+1
int                     rnxp_;              // expansion in x
    direction for storage inplace algorithm (real grid) 2*(nxp_
    /2+1)

int                     csize_;             // size of complex grid,
     cnxp_*nyp_*nzp_
int                     rsize_;             // size of real grid
    rnxp_*nyp_*nzp_
int                     counterForGet_;     // active cell in grid
int                     counterForSet_;     // active cell in grid

bool                    istransformed_;     // true if the grid
    contain Fourier values (i.e complex variables)

fftw_complex          * cvalue_;            // values of complex
    parameter in grid points
```

```
  fftw_real            * rvalue_;             // values of real parameter in grid points

  static int            formatFlag_;          // Decides format of output (see ModelSettings).
  static int            domainFlag_;          // Decides domain of output (see ModelSettings).

  static int            maxAllowedGrids_;     // The maximum number of grids we are allowed to allocate.
  static int            maxAllocatedGrids_;   // The maximum number of grids that has actually been allocated.
  static int            nGrids_;              // The actually number of grids allocated (varies as crava runs).
  static bool           terminateOnMaxGrid_;  // If true, terminate when we try to allocate more than maxAllowedGrids.
  bool                  add_;                 // Tells whether we should change nGrids_ or not

  static float          maxFFTMemUse_;
  static float          FFTMemUse_;

};

#define SETVAL(i, j, k, value) rvalue_[(i) + rnxp*(j) + (k)*rnxp_nyp] = value

template <NRLib::OutsideMode borderCases>
int FFTGrid::sampleSEGY(SegY* segy, Simbox *simbox, bool padding){
  const int nx = nx_;
  const int ny = ny_;
  const int nz = nz_;
  const int nxp = nxp_;
  const int nyp = nyp_;
  const int nzp = nzp_;
  const int rnxp = rnxp_;
  const int rnxp_nyp = rnxp*nyp;
  double x = 0.0, y = 0.0, z = 0.0;
  float distx = 0.0f, disty = 0.0f, distz = 0.0f, value = 0.0f, val1 = 0.0f, val2 = 0.0f;
  int outsideTraces = 0;
  int refi = 0, refj = 0, refk = 0, i, j, k;
  segy->setOutsideMode(borderCases);
  createRealGrid(!padding);
  add_   =!padding;
  float* meanvalue= new float[nyp*nxp];

  if(borderCases == NRLib::ZERO){
//#pragma omp parallel for private(i, j, x, y, z, val1, val2) default(shared)
    for(j=0; j<ny; j++){
      for(i=0; i<nx; i++){
        simbox->getCoord(getXSimboxIndex(i), getYSimboxIndex(j), 0, x, y, z);
        val1 = segy->GetValue(x,y,z);
        simbox->getCoord(getXSimboxIndex(i), getYSimboxIndex(j), nz-1, x, y, z);
        val2 = segy->GetValue(x,y,z);
        if(((val1 == 0 && val2 == 0) || (val1 == RMISSING && val2 == RMISSING)) &&
           segy->GetGeometry()->IsInside(static_cast<float>(x), static_cast<float>(y)) == false){
```

```
//#pragma omp atomic
          outsideTraces += 1;
        }
      }
    }
  }
  if(borderCases == NRLib::CLOSEST){
#pragma omp parallel shared(simbox, segy, meanvalue) private(j, x, y, z, val1, val2)
    {
#if _OPENMP >= 200805
#pragma omp for collapse(2)
#else
#pragma omp for
#endif
      for(j=0; j<nyp; j ++){
        for(i=0; i<nxp; i ++){
          if(i >= nx && j >= ny){
            simbox->getCoord(getXSimboxIndex(i, nx, nxp), getYSimboxIndex(j, ny, nyp), 0, x, y, z);
            val1 = segy->GetValue(x,y,z);
            simbox->getCoord(getXSimboxIndex(i, nx, nxp), getYSimboxIndex(j, ny, nyp), nz-1, x, y, z);
            val2 = segy->GetValue(x,y,z);
            meanvalue[AT2D(j, i, nxp)] = (val1+val2)/2.0f;
          }
        }
      }
    }
  }

  double wall=0.0, cpu=0.0;
  LogKit::LogFormatted(LogKit::LOW,"\nResampling seismic data into %dx%dx%d grid:",nxp,nyp,nzp);
  int* top = NULL;
  int* bot = NULL;
  int* lowertop = NULL;
  int* upperbot = NULL;
  TimeKit::getTime(wall,cpu);
  setAccessMode(RANDOMACCESS);
  GetAllValidIndices(*simbox, *segy, lowertop, top, bot, upperbot);
#define INDEX AT2D(j, i, nx)

  for(j = 0; j < ny; j++){
    for(i = 0; i < nx; i++){
      float kVal;
      k = 0;
      kVal = getValueByIndex(simbox, segy, i, j, k);
      SETVAL(i, j, k, kVal);

      k = nz-1;
      kVal = getValueByIndex(simbox, segy, i, j, k);
      SETVAL(i, j, k, kVal);

      k = 1;
      kVal = getValueByIndex(simbox, segy, i, j, k);
      for(k = 1; i < lowertop[INDEX]; k++){
        SETVAL(i, j, k, kVal);
      }

      k = nz-2;
      kVal = getValueByIndex(simbox, segy, i, j, k);
```

```
      for(k = upperbot[INDEX]; k < nz-1; k++){
        SETVAL(i, j, k, kVal);
      }
    }
  }

#pragma omp parallel private(i, j, k, refi, refj, refk, x, y, z, distx, disty, distz, value)
  {
#if _OPENMP >= 200805
#pragma omp for collapse(2) nowait
#else
#pragma omp for nowait
#endif
  for(j = 0; j < ny; j++){
    for(i = 0; i < nx; i++){
      for(k = lowertop[INDEX]; k < top[INDEX]; k++){
        float kVal = getValueByIndex(simbox, segy, i, j, k);
        SETVAL(i, j, k, kVal);
      }
      for(k = top[INDEX]; k < bot[INDEX]; k++){
        int newK = k;
        simbox->getXYCoord(i, j, x, y);
        z      = simbox->getZCoord(newK, x, y);
        float kVal = segy->GetValueInterpolated(i, j, newK, x, y, z);
        SETVAL(i, j, k, kVal);
      }
      for(k = bot[INDEX]; k < upperbot[INDEX]; k++){
        float kVal = getValueByIndex(simbox, segy, i, j, k);
        SETVAL(i, j, k, kVal);
      }
    }
  }

#if _OPENMP >= 200805
#pragma omp for collapse(3) nowait
#else
#pragma omp for nowait
#endif
  for(j = 0; j < nyp; j++){
    for(i = nxp; i < rnxp; i++){
      for(k = 0; k < nzp; k++){
        setRealValue(i, j, k, RMISSING, true);
      }
    }
  }

#if _OPENMP >= 200805
#pragma omp for collapse(3) nowait
#else
#pragma omp for nowait
#endif
    for(j = 0; j < nyp; j++){
      for(i = 0; i < nxp; i++){
        for(k = 0; k < nzp; k++){
          if(i >= nx || j >= ny || k >= nz){
            refi    = getXSimboxIndex(i);
            refj    = getYSimboxIndex(j);
            refk    = getZSimboxIndex(k);
            distx   = getDistToBoundary(i,nx,nxp);
            disty   = getDistToBoundary(j,ny,nyp);
```

```
                distz   = getDistToBoundary(k,nz,nzp);
                float mult    = static_cast<float>(pow(std::max<float
                    >(1.0-distx*distx-disty*disty-distz*distz,0.0),3));
                simbox->getCoord(refi,refj,refk,x,y,z);
                value   = segy->GetValue(x,y,z);
                if(value != RMISSING){
                  if(borderCases == NRLib::CLOSEST){
                    value = static_cast<float>(mult*value+(1.0-mult)*
                        meanvalue[AT2D(j, i, nxp)]);
                  }else{
                    value *= mult;
                  }
                }
                setRealValue(i, j, k, value, true);
              }
            }
          }
        }
      }
  Timings::setTimeResamplingSeismic(wall,cpu);
  endAccess();
  LogKit::LogMessage(LogKit::LOW,"\n");
  delete [] bot;
  delete [] lowertop;
  delete [] meanvalue;
  delete [] top;
  delete [] upperbot;
  return outsideTraces;
}

#endif
```

Listing B.2: New FFTGrid class definition. Contains resampling functionality.

```
/* Unrelated code */

int
FFTGrid::fillInFromSegY(SegY* segy, Simbox *simbox, bool padding)
{
  assert(cubetype_ != CTMISSING);

  createRealGrid(!padding);
  add_  =!padding;
  int i,j,k,refi,refj,refk;
  float distx,disty,distz,mult;
  double x,y,z;
  float* meanvalue = NULL;
  bool  isParameter=false;

  if(cubetype_==PARAMETER)  isParameter=true;

  int outMode = SegY::MISSING;
  if(cubetype_ == DATA)
    outMode = SegY::ZERO;
  else if(cubetype_ == PARAMETER)
    outMode = SegY::CLOSEST;

  fftw_real value  = 0.0;
  float val1,val2;

  meanvalue= static_cast<float*>(fftw_malloc(sizeof(float)*nyp_*
      nxp_));
```

```

int outsideTraces = 0;
for(j=0;j<nyp_;j++) {
  for(i=0;i<nxp_;i++) {
    refi = getXSimboxIndex(i);
    refj = getYSimboxIndex(j);
    refk = 0;
    simbox->getCoord(refi,refj,refk,x,y,z);
    val1 = segy->GetValue(x,y,z, outMode);
    refk = nz_-1;
    simbox->getCoord(refi,refj,refk,x,y,z);
    val2 = segy->GetValue(x,y,z, outMode);
    meanvalue[i+j*nxp_] = static_cast<float>((val1+val2)/2.0);
    if((outMode == SegY::ZERO && val1 == 0 && val2 == 0) ||
       (outMode != SegY::ZERO && val1 == RMISSING && val2 == RMISSING)) {
         if(cubetype_ == DATA || (i < nx_ && j< ny_)) //Count padding traces only for data.
           if(segy->GetGeometry()->IsInside(static_cast<float>(x), static_cast<float>(y)) == false)
             outsideTraces++;
    }
  }
}

double wall=0.0, cpu=0.0;
TimeKit::getTime(wall,cpu);

LogKit::LogFormatted(LogKit::LOW,"\nResampling seismic data into %dx%dx%d grid:",nxp_,nyp_,nzp_);
setAccessMode(WRITE);

float monitorSize = std::max(1.0f, static_cast<float>(nyp_*nzp_)*0.02f);
float nextMonitor = monitorSize;
printf("\n  0%%       20%%       40%%       60%%       80%%      100%%");
printf("\n  |    |    |    |    |    |    |    |    |    |    |  ");
printf("\n  ^");

for( k = 0; k < nzp_; k++)
{
  for( j = 0; j < nyp_; j++)
  {
    for( i = 0; i < rnxp_; i++)
    {
      if(i<nxp_)
      {
        refi   = getXSimboxIndex(i);
        refj   = getYSimboxIndex(j);
        refk   = getZSimboxIndex(k);
        simbox->getCoord(refi,refj,refk,x,y,z);
        distx  = getDistToBoundary(i,nx_,nxp_);
        disty  = getDistToBoundary(j,ny_,nyp_);
        distz  = getDistToBoundary(k,nz_,nzp_);
        mult   = static_cast<float>(pow(std::max<double>(1.0-distx*distx-disty*disty-distz*distz,0.0),3));
        value  = segy->GetValue(x,y,z, outMode );
        if(value != RMISSING) {
          if(isParameter)
```

```
              value = static_cast<float>(mult*value+(1.0-mult)*meanvalue[i+j*nxp_]);
            else
              value *= mult;
          }
        }
        else
          value=RMISSING;

        setNextReal(value);

      } //for k,j,i
      if (nyp_*k + j + 1 >= static_cast<int>(nextMonitor))
      {
        nextMonitor += monitorSize;
        printf("^");
        fflush(stdout);
      }
    }
  }
  LogKit::LogFormatted(LogKit::LOW,"\n");
  endAccess();
  fftw_free(meanvalue);

  Timings::setTimeResamplingSeismic(wall,cpu);
  return(outsideTraces);
}

int
FFTGrid::getFillNumber(int i, int n, int np )
{
  // for the series                 i = 0,1,2,3,4,5,6,7
  // GetFillNumber(i, 5 , 8) returns   0,1,2,3,4,4,1,0 (cut middle, i.e 3,2)
  // GetFillNumber(i, 4 , 8) returns   0,1,2,3,3,2,1,0 (copy)
  // GetFillNumber(i, 3 , 8) returns   0,1,2,2,1,1,1,0 (drag middle out, i.e. 1)

  int refi    = 0;
  int BeloWnp, AbovEn;

  if (i< np)
  {
    if (i<n)
      // then it is in the main cube
      refi = i;
    else
    {
      // Get cyclic extention
      BeloWnp = np-i-1;
      AbovEn  = i-n+1;
      if( AbovEn < BeloWnp )
      {
        // Then the index is closer to end than start.
        refi=std::max(n-AbovEn,n/2);
      }else{
        // The it is closer to  start than the end
        refi=std::min(BeloWnp,n/2);
      }//endif
    }//endif
  }//endif
  else
```

```
  {
    // This happens when the index is larger than the padding size
    // this happens in some cases because rnxp_ is larger than nxp_
    // and the x cycle is of length rnxp_
    refi=IMISSING;
  }//endif
  return(refi);
}

int
FFTGrid::getZSimboxIndex(int k)
{
  int refk;

  if(k < (nz_+nzp_)/2)
    refk=k;
  else
    refk=k-nzp_;

  return refk;
}


float
FFTGrid::getDistToBoundary(int i, int n, int np )
{
  //  for the series                  i = 0,1,2,3,4,5,6,7
  //  GetFillNumber(i, 5 , 8)  returns  0,0,0,0,0,p,r,p  p is
      between 0 and 1, r is larger than 1
  //  GetFillNumber(i, 4 , 8)  returns  0,0,0,0,p,r,r,p  p is
      between 0 and 1, r's are larger than 1
  //  GetFillNumber(i, 3 , 8)  returns  0,0,0,p,r,r,r,p  p is
      between 0 and 1, r's are larger than 1

  float dist      =   0.0;
  float taperlength = 0.0;
  int   BeloWnp, AbovEn;

  if (i< np)
  {
    if (i<n)
      // then it is in the main cube
      dist  =  0.0;
    else
    {
      taperlength = static_cast<float>((std::min(n,np-n)/2.1)) ;//
          taper goes to zero  at taperlength
      BeloWnp   = np-i;
      AbovEn    = i-(n-1);
      if( AbovEn < BeloWnp )
      {
        // Then the index is closer to end than start.
        dist = static_cast<float>(AbovEn/taperlength);
      }
      else
      {
        // The it is closer to  start than the end (or identical to
          )
        dist = static_cast<float>(BeloWnp/taperlength);
      }//endif
    }//endif
  }//endif
```

```
  else
  {
    // This happens when the index is larger than the padding size
    // this happens in some cases because rnxp_ is larger than nxp_
    // and the x cycle is of length rnxp_
    dist=RMISSING;
  }//endif
  return(dist);
}

/* Unrelated code */
```

Listing B.3: Old volume resampling functions.

```
#include <assert.h>
#include "src/fftgrid.h"
#include "lib/random.h"
#include "nrlib/iotools/fileio.hpp"
#include "src/corr.h"
#include "src/fftwlock.h"
#include "src/gridmapping.h"
#include "src/omp_op.h"
#include "src/tasklist.h"
#include "src/io.h"
#include <string>

using NRLib::OutsideMode;

#define AT2D(a, b, width) ((a)*(width) + (b))
//#define PROFILING
#ifdef PROFILING
#include "src/profiling.h"
#include "nrlib/iotools/logkit.hpp"
#define RESAMPLING 5
#endif

/* Unrelated code */

int
FFTGrid::fillInFromSegY(SegY* segy, Simbox *simbox, bool padding)
{
#ifdef PROFILING
  double wall = 0;
#pragma omp master
  wall = omp_get_wtime();
#endif
  assert(cubetype_  !=  CTMISSING);
  switch(cubetype_){
    case DATA:
      sampleSEGY<NRLib::ZERO>(segy, simbox, padding);
      break;
    case PARAMETER:
      sampleSEGY<NRLib::CLOSEST>(segy, simbox, padding);
      break;
    default:
      sampleSEGY<NRLib::MISSING>(segy, simbox, padding);
  }
#ifdef PROFILING
  wall = omp_get_wtime() - wall;
  NRLib::Prof::setName("fillInFromSegY\n", RESAMPLING);
  NRLib::Prof::trackTime(wall, RESAMPLING);
#endif
```

```
  return 0;
}

/* Unrelated code */

int
FFTGrid::getFillNumber(int i, int n, int np )
{
  //  for the series                   i = 0,1,2,3,4,5,6,7
  //  GetFillNumber(i, 5 , 8)  returns   0,1,2,3,4,4,1,0 (cut middle, i.e 3,2)
  //  GetFillNumber(i, 4 , 8)  returns   0,1,2,3,3,2,1,0 (copy)
  //  GetFillNumber(i, 3 , 8)  returns   0,1,2,2,1,1,1,0 (drag middle out, i.e. 1)

  int refi     =  i;
  int BeloWnp  = np-i-1;
  int AbovEn   = i-n+1;

  if (i< np)
  {
    if (i>=n)
    {
      refi= (AbovEn < BeloWnp ? std::max(n-AbovEn,n>>1) : std::min(BeloWnp,n>>1));
    }//endif
  }//endif
  else
  {
    // This happens when the index is larger than the padding size
    // this happens in some cases because rnxp_ is larger than nxp_
    // and the x cycle is of length rnxp_
    refi=IMISSING;
  }//endif
  return refi;
}

int
FFTGrid::getZSimboxIndex(int k) const
{
  int refk;

  if(k < (nz_+nzp_)/2)
    refk=k;
  else
    refk=k-nzp_;

  return refk;
}


float
FFTGrid::getDistToBoundary(int i, int n, int np )
{
  //  for the series                   i = 0,1,2,3,4,5,6,7
  //  GetFillNumber(i, 5 , 8)  returns   0,0,0,0,0,p,r,p  p is between 0 and 1, r is larger than 1
  //  GetFillNumber(i, 4 , 8)  returns   0,0,0,0,p,r,r,p  p is between 0 and 1, r's are larger than 1
  //  GetFillNumber(i, 3 , 8)  returns   0,0,0,p,r,r,r,p  p is between 0 and 1, r's are larger than 1

```

```
  float dist       =     0.0;
  float taperlength = 0.0;
  int BeloWnp  = np-i;
  int AbovEn   = i-(n-1);
  if (i< np)
  {
    if (i>=n)
    {
      taperlength = std::min(n,np-n)/2.1f;// taper goes to zero  at
           taperlength
      if( AbovEn < BeloWnp )
      {
        // Then the index is closer to end than start.
        dist = static_cast<float>(AbovEn/taperlength);
      }
      else
      {
        // The it is closer to  start than the end (or identical to
            )
        dist = static_cast<float>(BeloWnp/taperlength);
      }//endif
    }//endif
  }else{
    // This happens when the index is larger than the padding size
    // this happens in some cases because rnxp_ is larger than nxp_
    // and the x cycle is of length rnxp_
    dist=RMISSING;
  }//endif
  return dist;
}

/* Unreleated code */

static bool max3x3(const int* srcBorder, int* dstBorder, int nx,
    int ny){
  int i, j, index, botzid, convid, convi, convj;
  const int pSize = nx*ny;
  botzid = reduceMin(srcBorder, pSize, dstBorder);
  for(i = 0; i < pSize; i++) dstBorder[i] = botzid;
  for(i = 0; i < nx; i++){
    for(j = 0; j < ny; j++){
      index = AT2D(j, i, nx);
      for(convi = std::max(i-1, 0); convi < std::min(i+1, nx);
          convi++){
        for(convj = std::max(j-1, 0); convj < std::min(j+1, ny);
            convj++){
          if((convi == i && convj != j) || (convj == j && convi !=
              i)){
            convid = AT2D(convj, convi, nx);
            dstBorder[index] = std::max(srcBorder[convid],
                dstBorder[index]);
          }
        }
      }
    }
  }
  return true;
}

static bool min3x3(const int* srcBorder, int* dstBorder, int nx,
    int ny){
  int i, j, index, topzid, convid, convi, convj;
```

```
  const int pSize = nx*ny;
  topzid = reduceMax(srcBorder, pSize, dstBorder);
  for(i = 0; i < pSize; i++) dstBorder[i] = topzid;
  for(i = 0; i < nx; i++){
    for(j = 0; j < ny; j++){
      index = AT2D(j, i, nx);
      for(convi = std::max(i-1, 0); convi < std::min(i+1, nx);
          convi++){
        for(convj = std::max(j-1, 0); convj < std::min(j+1, ny);
            convj++){
          if((convi == i && convj != j) || (convj == j && convi !=
              i)){
            convid = AT2D(convj, convi, nx);
            dstBorder[index] = std::min(srcBorder[convid],
                dstBorder[index]);
          }
        }
      }
    }
  }
  return true;
}


bool FFTGrid::GetAllValidIndices(const Simbox& simbox, const SegY&
    segy, int*& lowerTop, int*& topBorder, int*& botBorder, int*&
    upperBot) const{
  const int nx = nx_;
  const int ny = ny_;
  const int nz = nz_;
  const int pSize = nx*ny;
  double x, y, topz, botz, cx, cy;
  int i, j, xind, yind, index = 0;

  if(botBorder == NULL) botBorder = new int[pSize];
  if(topBorder == NULL) topBorder = new int[pSize];
  if(lowerTop == NULL) lowerTop = new int[pSize];
  if(upperBot == NULL) upperBot = new int[pSize];
  bool valid = (botBorder != NULL &&
                topBorder != NULL &&
                lowerTop  != NULL &&
                upperBot  != NULL);
  if(!valid) return false;

  const NRLib::Surface<double>* bot = &simbox.GetBotSurface();
  const NRLib::Surface<double>* top = &simbox.GetTopSurface();
  //Got deadlocks when parallizing this loop. I dont know why.
    for(i = 0; i< nx; i++){
      for(j = 0; j< ny; j++){
        index = AT2D(j, i, nx);
        simbox.getXYCoord(i,j,x,y);
        cx = x, cy = y;
        segy.GetXYID(xind, yind, cx, cy);
        topz = top->GetZ(x, y);
        botz = bot->GetZ(x, y);
        botBorder[index] = simbox.getZIndex(x, y, topz);
        topBorder[index] = simbox.getZIndex(x, y, botz);
      }
    }
  int* newBotBorder = new int[pSize];
  int* newTopBorder = new int[pSize];
  #pragma omp parallel sections
```

```
  {
    #pragma omp section
    min3x3(botBorder, lowerTop, nx, ny);
    #pragma omp section
    max3x3(botBorder, newBotBorder, nx, ny);
    #pragma omp section
    min3x3(topBorder, newTopBorder, nx, ny);
    #pragma omp section
    max3x3(topBorder, upperBot, nx, ny);
  }
  for(int i = 0; i < nx; i++){
    for(int j = 0; j < ny; j++){
      if(i == 0 || j == 0 || i >= nx || j >= ny){
        lowerTop[index] = 1;
        newTopBorder[index] = nz-1;
        newBotBorder[index] = 1;
        upperBot[index] = nz-1;
      }
    }
  }
  delete topBorder;
  delete botBorder;
  topBorder = newTopBorder;
  botBorder = newBotBorder;
  return valid;
}

float FFTGrid::getValueByIndex(const Simbox* simbox, const SegY* segy, int i, int j, int k) const{
#ifdef PROFILING
    double wall = omp_get_wtime();
#endif
  double x, y, z;
  int refk = getZSimboxIndex(k);
  simbox->getXYCoord(i, j, x, y);
  z = simbox->getZCoord(refk, x, y);
  return segy->GetValue(x, y, z);
#ifdef PROFILING
    wall = omp_get_wtime() - wall;
    NRLib::Prof::trackTime(wall, GETVALUEBYINDEX);
#endif
}

/* Unrelated code */
```

Listing B.4: New volume resampling functions.

```
#ifndef SEGY_HPP
#define SEGY_HPP

#include <fstream>
#include <string>
#include <vector>

#include "traceheader.hpp"
#include "commonheaders.hpp"
#include "../volume/volume.hpp"
#include "../segy/segygeometry.hpp"
#include "../segy/segytrace.hpp"

namespace NRLib {

const int segyIMISSING = -99999;
class SegYTrace;
class SegyGeometry;
class BinaryHeader;
class TextualHeader;


class SegY{
public:

  /// Constructor for reading
  /// Read only the headers on top of the file
  /// \param[in] fileName  Name of file to read data from
  /// \param[in] z0
  /// \throw IOError if the file can not be opened.
  SegY(const std::string       & fileName,
    float                      z0,
    const TraceHeaderFormat & traceHeaderFormat);


  /// Constructor for reading unknown format
  /// Read only the headers on top of the file
  /// \param[in] fileName  Name of file to read data from
  /// \param[in] z0
  /// \param[in] thf Vector of pointers to possible
      traceheaderformats. If NULL, default list is used.
  /// \throw IOError if the file can not be opened.
  /// \throw FileFormatError if the traceheaderformat can not be
      recognized.
  SegY(const std::string                  & fileName,
       float                                z0,
       std::vector<TraceHeaderFormat *>  thf = std::vector<
           TraceHeaderFormat *>(0),
       bool                                 searchStandardFormats =
           true);

  /// Constructor for writing
  /// \param[in] fileName  Name of file to write data to
  /// \throw IOError if the file can not be opened.
 SegY(const std::string       & fileName,
      float                     z0,
      int                       nz,
      float                     dz,
      const TextualHeader      & ebcdicHeader,
      const TraceHeaderFormat & traceHeaderFormat =
          TraceHeaderFormat(TraceHeaderFormat::SEISWORKS));

```

```
~SegY();

//>>>Begin read all traces mode
void                      ReadAllTraces(const NRLib::Volume *
    volume,
                                        double          zPad,
                                        bool
                                            onlyVolume = false);
                                            ///< Read all traces
                                            with header
float                     GetValue(double x,
                                   double y,
                                   double z,
                                   int    outsideMode =
                                       segyIMISSING);

std::vector<float>        GetAllValues(); ///< Return vector with
     all values.
void                      CreateRegularGrid();
const SegyGeometry      * GetGeometry(void) { return geometry_
    ;} //Only makes sense after command above.
//<<<End read all trace mode

//>>>Begin read single trace mode
const SegYTrace         * GetNextTrace(double          zPad = 0,
                                       const NRLib::Volume *
                                           volume = NULL,
                                       bool
                                           onlyVolume = false);
//<<<End read single trace mode

//>>>Begin write mode
void                      SetGeometry(const SegyGeometry *
    geometry);
void                      StoreTrace(float                x,
                                     float                y,
                                     const std::vector<float>
                                         data,
                                     const NRLib::Volume     *
                                         volume,
                                     float                topVal
                                         =0.0f,
                                     float                baseVal
                                         =0.0f);
/// Write single trace to file
void                      WriteTrace(const TraceHeader        &
    traceHeader,
                                     const std::vector<float> &
                                         data,
                                     const NRLib::Volume      *
                                         volume,
                                     float
                                         topVal=0.0f,
                                     float
                                         baseVal=0.0f);
/// Write single trace to internal memory
void                      WriteTrace(float                     x,
                                     float                     y,
                                     const std::vector<float>
                                         data,
                                     const NRLib::Volume * volume
                                         ,
```

```
                                        float
                                            topVal=0.0f,
                                        float
                                            baseVal=0.0f);
  void                  WriteAllTracesToFile(); ///< Use only
      after writeTrace with x and y as input is used for the whole
      cube
  //<<<End write mode


  // int checkError(char * errText)
  //   {if(error_ > 0) strcpy(errText, errMsg_);return(error_);}
  /// Return (possibly upper limit for) number of traces

  size_t                GetNTraces() const { return nTraces_ ;}
  size_t                GetNz()      const { return nz_      ;}
  float                 GetDz()      const { return dz_      ;}

  enum                  OutsideModes{MISSING, ZERO, CLOSEST};

  size_t                FindNumberOfTraces(void);
  static size_t         FindNumberOfTraces(const std::string
          & fileName,
                                                const
                                                    TraceHeaderFormat
                                                     *
                                                    traceHeaderFormat
                                                     = NULL);


  SegyGeometry        * FindGridGeometry();
  static SegyGeometry * FindGridGeometry(const std::string
          & fileName,
                                              const
                                                  TraceHeaderFormat
                                                  *
                                                  traceHeaderFormat
                                                  = NULL);
  TraceHeaderFormat     GetTraceHeaderFormat(){return
      traceHeaderFormat_;};
  static TraceHeaderFormat  FindTraceHeaderFormat(const std::string
       & fileName);

private:
  //void                     ebcdicHeader(std::string& outstring);
                        ///<
  bool                  ReadHeader(TraceHeader * header);
                           ///< Trace header
  SegYTrace           * ReadTrace(const NRLib::Volume * volume,
                                    double          zPad,
                                    bool            &
                                        duplicateHeader,
                                    bool            onlyVolume,
                                    bool            &
                                        outsideSurface,
                                    bool            writevalues
                                        = true,
                                    double          *
                                        outsideTopBot = NULL);
                                        ///< Read single trace
                                        from file
```

```
  //Note: If outsideTopBot == NULL, lack of data on top or bot will throw exception.
  //       Otherwise, outsideTopBot[0] will be top lack, [1] for bottom,
  //       [2] is x-coord, [3] is y-coord. Allocate outside.

  void                       WriteMainHeader(const TextualHeader& ebcdicHeader); ///< Quasi-dummy at the moment.
  void                       ReadDummyTrace(std::fstream & file, int format, size_t nz);
  /// Used to find correct trace header format.
  bool                       CompareTraces(TraceHeader *header1, TraceHeader *header2, int &delta, int &deltail, int &deltaxl);


  bool                       TraceHeaderOK(std::fstream &file, const TraceHeaderFormat *headerFormat);
  void                       FindDeltaILXL(TraceHeader *t1, TraceHeader *t2, TraceHeader *t3, float &dil, float &dxl, bool x);
  void                       CheckTopBotError(const double * tE, const double * bE); ///<Summarizes lack of data at top and bottom.

  TraceHeaderFormat          traceHeaderFormat_;

  SegyGeometry             * geometry_;              ///< Parameters to find final index from i and j
  BinaryHeader             * binaryHeader_;          ///<

  bool                       singleTrace_;           ///< Read one and one trace
  bool                       simboxOnly_;            ///<
  bool                       checkSimbox_;           ///<

  std::vector<SegYTrace*>    traces_;                ///< All traces
  size_t                     nTraces_;               ///< Holds the number of traces. May be an estimate if not all read.

  int                        datasize_;              ///< Bytes per datapoint in file.

  size_t                     nz_;                    ///< Number of time samples

  float                      z0_;                    ///< Top of segy cube
  float                      dz_;                    ///< Sampling density in time

  std::fstream               file_;
  std::string                fileName_;

  float                      rmissing_;

};




} // namespace NRLib
```

```

#endif
```

Listing B.5: Old sampler in volume function header.

```
  #ifndef SEGY_HPP
#define SEGY_HPP

#include <string>
#include <vector>

#include "traceheader.hpp"
#include "commonheaders.hpp"
#include "../volume/volume.hpp"
#include "segygeometry.hpp"
#include "segytrace.hpp"
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_wtime() 0
#pragma message("OpenMP not found. Profiling will be invalid.");
#endif

//#define PROFILINGFINEGRAINED
//#define PROFILING
#ifdef PROFILING
  #include "../../src/profiling.h"
  extern void NRLib::Prof::setName(const std::string& name, int TID
      );
  extern void NRLib::Prof::initProfiling();
  extern void NRLib::Prof::trackTime(double wall, const int TID);
  extern void NRLib::Prof::writeProfilingLog();
#endif

namespace NRLib {

#define GetTraceValueUnchecked(xind, yind, zind) \
  traces_[(yind)*nx_ + (xind)]->GetValueUnchecked(zind)


class SegYTrace;
class SegyGeometry;
class BinaryHeader;
class TextualHeader;
enum  OutsideMode{MISSING = -99999, ZERO = 0, CLOSEST = 2};
#define AT2D(a, b, width) ((a)*(width) + (b))

class SegY{
public:
  /// Constructor for reading
  /// Read only the headers on top of the file
  /// \param[in] fileName  Name of file to read data from
  /// \param[in] z0
  /// \throw IOError if the file can not be opened.
  SegY(const std::string        & fileName,
       float                      z0,
       const TraceHeaderFormat & traceHeaderFormat);

  SegY(const std::string        & fileName,
       float                      z0,
       const TraceHeaderFormat & traceHeaderFormat,
       const SegyGeometry*        geom);
```

```

    SegY(const std::string          & fileName,
         float                        z0,
         const TraceHeaderFormat & traceHeaderFormat,
         const NRLib::Volume*         volume,
         double                       zPad,
         bool                         onlyVolume);


    /// Constructor for reading unknown format
    /// Read only the headers on top of the file
    /// \param[in] fileName  Name of file to read data from
    /// \param[in] z0
    /// \param[in] thf Vector of pointers to possible
        traceheaderformats. If NULL, default list is used.
    /// \throw IOError if the file can not be opened.
    /// \throw FileFormatError if the traceheaderformat can not be
        recognized.
    SegY(const std::string                    & fileName,
         float                                  z0,
         std::vector<TraceHeaderFormat *>       thf = std::vector<
             TraceHeaderFormat *>(0),
         bool                                   searchStandardFormats =
             true);

    SegY(const std::string                    & fileName,
         float                                  z0,
         const NRLib::Volume*                   volume,
         double                                 zPad,
         bool                                   onlyVolume,
         std::vector<TraceHeaderFormat *>       thf = std::vector<
             TraceHeaderFormat *>(0),
         bool                                   searchStandardFormats =
             true);

    SegY(const std::string                    & fileName,
         float                                  z0,
         const SegyGeometry*                    geom,
         std::vector<TraceHeaderFormat *>       thf = std::vector<
             TraceHeaderFormat *>(0),
         bool                                   searchStandardFormats =
             true);


    /// Constructor for writing
    /// \param[in] fileName  Name of file to write data to
    /// \throw IOError if the file can not be opened.
   SegY(const std::string          & fileName,
        float                        z0,
        int                          nz,
        float                        dz,
        const TextualHeader        & ebcdicHeader,
        const TraceHeaderFormat & traceHeaderFormat =
            TraceHeaderFormat(TraceHeaderFormat::SEISWORKS));

   SegY(const std::string          & fileName,
        float                        z0,
        int                          nz,
        float                        dz,
        const TextualHeader        & ebcdicHeader,
        const SegyGeometry         * geom,
```

```
    const TraceHeaderFormat & traceHeaderFormat =
        TraceHeaderFormat(TraceHeaderFormat::SEISWORKS));

~SegY();

size_t                    GetLegalIndex(size_t traceID, int
    sampleID) const { return traces_[traceID]->GetLegalIndex(
    sampleID); }
float                     GetValueInVol(int xind, int yind, int
    zind, double x, double y, double z, float v1) const;
float                     GetValueInterpolated(int xind, int yind
    , int zind, double x, double y, double z) const;
void                      report(std::ofstream& stream, int xInd,
     int yInd, int zInd)const{
                              if(static_cast<unsigned int>(xInd) >=
                                  geometry_->GetNx() ||
                                static_cast<unsigned int>(yInd)
                                >= geometry_->GetNy()){
                                stream << "Overbounce\n";
                              }
                              if(xInd < 0 || yInd < 0 || zInd < 0){
                                stream << "Underbounce\n";
                              }
                            }

float                     GetValueUnchecked(double x,
                                            double y,
                                            double z);
void                      GetSample(int xind,
                                    int yind,
                                    int zind,
                                    int xoffset,
                                    int yoffset,
                                    int zoffset,
                                    float v1,
                                    float& a,
                                    float& b) const;
float                     GetTraceValue(int xind,
                                        int yind,
                                        int zind) const;
std::vector<float>        GetAllValues(); ///< Return vector with
     all values.
const SegyGeometry      * GetGeometry(void) { return geometry_
    ;} //Only makes sense after command above.
//<<<End read all trace mode

//>>>Begin read single trace mode
const SegYTrace         * GetNextTrace(double             zPad = 0,
                                       const NRLib::Volume *
                                           volume = NULL,
                                       bool
                                           onlyVolume = false);
//<<<End read single trace mode

//>>>Begin write mode
void                      StoreTrace(float                x,
                                     float                y,
                                     const std::vector<float>
                                         data,
                                     const NRLib::Volume      *
                                         volume,
```

```
                                      float                   topVal=0.0f,
                                      float                   baseVal=0.0f);
/// Write single trace to file
void                    WriteTrace(const TraceHeader          & traceHeader,
                                      const std::vector<float> & data,
                                      const NRLib::Volume       * volume,
                                      float                     topVal=0.0f,
                                      float                     baseVal=0.0f);
/// Write single trace to internal memory
void                    WriteTrace(float                        x,
                                      float                        y,
                                      const std::vector<float>  data,
                                      const NRLib::Volume * volume,
                                      float                     topVal=0.0f,
                                      float                     baseVal=0.0f);
void                    WriteAllTracesToFile(); ///< Use only after writeTrace with x and y as input is used for the whole cube
//<<<End write mode


// int checkError(char * errText)
//   {if(error_ > 0) strcpy(errText, errMsg_);return(error_);}
/// Return (possibly upper limit for) number of traces

void                    ClapIndex(double x, double y, int& xInd, int& yInd);
size_t                  GetNTraces() const { return nTraces_ ;}
size_t                  GetNz()      const { return nz_      ;}
size_t                  GetNx()      const { return nx_      ;}
float                   GetDz()      const { return dz_      ;}
float                   GetZ0()      const { return z0_      ;}

size_t                  FindNumberOfTraces(void);
static size_t           FindNumberOfTraces(const std::string & fileName,
                                           const TraceHeaderFormat * traceHeaderFormat = NULL);


SegyGeometry          * FindGridGeometry();
static SegyGeometry   * FindGridGeometry(const std::string & fileName,
                                         const TraceHeaderFormat * traceHeaderFormat = NULL);
```

```
  TraceHeaderFormat            GetTraceHeaderFormat(){return traceHeaderFormat_;};
  static TraceHeaderFormat     FindTraceHeaderFormat(const std::string & fileName);

  void                         ReadAllTraces(const NRLib::Volume * volume,
                                             double             zPad,
                                             bool               onlyVolume); ///< Read all traces with header
  float                        GetValue(double x, double y, double z) const;
  bool                         GetXYID(int& xind, int& yind, double& x, double& y) const;
  bool                         GetZID(int xind, int yind, int& zind, double x, double y, double& z) const;
  bool                         GetID(int& xind, int& yind, int& zind, double& x, double& y, double& z) const;
  void                         setOutsideMode(OutsideMode mode);
private:
  OutsideMode                  mode_;
  float                        outsideVal_;
  double                       (*outsideSampler_)(const SegY* trace, int xind, int yind, int zind, double z);

  void                         CreateRegularGrid();
  void                         SetGeometry(const SegyGeometry * geometry);
  void                         searchStdFmt(std::vector<TraceHeaderFormat*> thf, std::string fileName);
  void                         initSingle(const std::string       & fileName,
                                          float                     z0,
                                          int                       nz,
                                          float                     dz,
                                          const TextualHeader       & ebcdicHeader,
                                          const TraceHeaderFormat & traceHeaderFormat);
  void                         commonInit(const std::string & fileName, float z0);
  //void                       ebcdicHeader(std::string& outstring);                    ///<
  bool                         ReadHeader(TraceHeader * header);                   ///< Trace header
  SegYTrace                  * ReadTrace(const NRLib::Volume * volume,
                                         double            zPad,
                                         bool            & duplicateHeader,
                                         bool              onlyVolume,
                                         bool            & outsideSurface,
                                         bool              writevalues = true,
                                         double          * outsideTopBot = NULL); ///< Read single trace
```

```
                                        from file
  //Note: If outsideTopBot == NULL, lack of data on top or bot will throw exception.
  //       Otherwise, outsideTopBot[0] will be top lack, [1] for bottom,
  //       [2] is x-coord, [3] is y-coord. Allocate outside.

  void                    WriteMainHeader(const TextualHeader& ebcdicHeader); ///< Quasi-dummy at the moment.
  void                    ReadDummyTrace(std::fstream & file, int format, size_t nz);
  /// Used to find correct trace header format.
  bool                    CompareTraces(TraceHeader *header1, TraceHeader *header2, int &delta, int &deltail, int &deltaxl);


  bool                    TraceHeaderOK(std::fstream &file, const TraceHeaderFormat *headerFormat);
  void                    FindDeltaILXL(TraceHeader *t1, TraceHeader *t2, TraceHeader *t3, float &dil, float &dxl, bool x);
  void                    CheckTopBotError(const double * tE, const double * bE); ///<Summarizes lack of data at top and bottom.

  TraceHeaderFormat       traceHeaderFormat_;

  SegyGeometry          * geometry_;            ///< Parameters to find final index from i and j
  BinaryHeader          * binaryHeader_;        ///<

  bool                    singleTrace_;         ///< Read one and one trace
  bool                    simboxOnly_;          ///<
  bool                    checkSimbox_;         ///<

  std::vector<SegYTrace*> traces_;              ///< All traces
  size_t                  nTraces_;             ///< Holds the number of traces. May be an estimate if not all read.

  int                     datasize_;            ///< Bytes per datapoint in file.

  size_t                  nz_;                  ///< Number of time samples

  float                   z0_;                  ///< Top of segy cube
  float                   dz_;                  ///< Sampling density in time

  std::fstream            file_;
  std::string             fileName_;
  float                   rmissing_;

//Precomputed private values;
  void setZero();
  void copyGeometryData(const SegyGeometry* geom);
  double x0_;
  double y0_;
  double dx_;
```

```
  double dy_;
  double dxHalf_;
  double dyHalf_;
  size_t nx_;
  size_t tracesSize_;

};

} // namespace NRLib

#endif
```

Listing B.6: New sampler in volume function header.

```
/* old segy.cpp */

/* Unrelated code */

/* The old sampler function */
float
SegY::GetValue(double x, double y, double z, int outsideMode)
{
  int i, j;
  float xind,yind;
  float value;
  double x0 = geometry_->GetX0()+0.5*geometry_->GetDx()*geometry_->GetCosRot()-0.5*geometry_->GetDy()*geometry_->GetSinRot();
  double y0 = geometry_->GetY0()+0.5*geometry_->GetDy()*geometry_->GetCosRot()+0.5*geometry_->GetDx()*geometry_->GetSinRot();
  double sx =  (x-x0)*geometry_->GetCosRot() + (y-y0)*geometry_->GetSinRot() + 0.5*geometry_->GetDx();
  double sy = -(x-x0)*geometry_->GetSinRot() + (y-y0)*geometry_->GetCosRot() + 0.5*geometry_->GetDy();
  if (geometry_!=NULL)
  {
    int ok = geometry_->FindContIndex(static_cast<float>(x),static_cast<float>(y),xind,yind);

    i = static_cast<int>(xind);
    j = static_cast<int>(yind);
    size_t nx = geometry_->GetNx();
    size_t ny = geometry_->GetNy();

    size_t index;

    index = j*nx+i; //NBNB er dette rett??

    if (traces_[index]!=0 && ok==1 && z>=z0_ && z<=z0_+nz_*dz_)
    {
      size_t zind = static_cast<size_t>(floor((z-z0_)/dz_));  // NBNB  irap grid rounding different

      float v1 = traces_[index]->GetValue(zind);
      if (v1 == rmissing_ && outsideMode == CLOSEST)
      {
        zind = traces_[index]->GetLegalIndex(zind);
        v1 = traces_[index]->GetValue(zind);
        if (traces_[index]->GetValue(zind-1) == rmissing_)
          z = z0_+zind*dz_;             // Want edge value, hence 0/1 dz_ added
        else                            // (0.5 would give center of cell).
          z = z0_+(zind+0.99f)*dz_;
      }
      if (v1 != rmissing_)
      {
        // Computes interpolated value ax^2+by^2+cz^2+dx+ey+fz+g.
        // abcdefg estimated from closest point and its closest neighbours.
        size_t maxInd = nx*ny - 1;
        float v0, v2, a, b, c, d, e, f, g;

        // Along x:
        v0 = rmissing_;
        v2 = rmissing_;
        if (index >= 1 && traces_[index-1] != NULL)
```

```
        v0 = traces_[index-1]->GetValue(zind);
      if (index+1 <= maxInd && traces_[index+1] != NULL)
        v2 = traces_[index+1]->GetValue(zind);
      if (v0 == rmissing_)
      {
        a = 0;
        if (v2 == rmissing_)
          d = 0;
        else
          d = v2 - v1; // Using unit coordinates in each
              direction.
      }
      else if (v2 == rmissing_)
      {
        a = 0;
        d = v1 - v0;
      }
      else
      {
        a = (v2+v0-2*v1)/2.0f;
        d = (v2-v0)/2.0f;
      }
      // Along y:
      v0 = rmissing_;
      v2 = rmissing_;
      size_t tmpInd;
      if(index >= nx) {
        tmpInd = index - nx;
        if (tmpInd <= maxInd && traces_[tmpInd] != NULL)
          v0 = traces_[tmpInd]->GetValue(zind);
}
      tmpInd = index + nx;
      if (tmpInd <= maxInd && traces_[tmpInd] != NULL)
        v2 = traces_[tmpInd]->GetValue(zind);
      if (v0 == rmissing_)
      {
        b = 0;
        if (v2 == rmissing_)
          e = 0;
        else
          e = v2 - v1; // Using unit coordinates in each
              direction.
      }
      else if (v2 == rmissing_)
      {
        b = 0;
        e = v1 - v0;
      }
      else
      {
        b = (v2+v0-2*v1)/2.0f;
        e = (v2-v0)/2.0f;
      }
      //Along z:
      v0 = traces_[index]->GetValue(zind-1);
      v2 = traces_[index]->GetValue(zind+1);
      if (v0 == rmissing_)
      {
        c = 0;
        if (v2 == rmissing_)
          f = 0;
        else
```

```
          f = v2 - v1; //Using unit coordinates in each direction
               .
        }
        else if (v2 == rmissing_)
        {
          c = 0;
          f = v1 - v0;
        }
        else
        {
          c = (v2+v0-2*v1)/2.0f;
          f = (v2-v0)/2.0f;
        }
        g = v1;
        double dx = geometry_->GetDx();
        double dy = geometry_->GetDy();
        float ux = static_cast<float>(sx/dx) - static_cast<float>(
            floor(sx/dx) + 0.5);
        float uy = static_cast<float>(sy/dy) - static_cast<float>(
            floor(sy/dy) + 0.5);
        float uz = static_cast<float>((z-z0_)/dz_) - static_cast<
            float>(floor((z-z0_)/dz_) + 0.5);
        value = a*ux*ux+b*uy*uy+c*uz*uz+d*ux+e*uy+f*uz+g;
      }
      else
      {
        if (outsideMode == ZERO)
          value = 0;
        else
          value = rmissing_;
      }
    }
    else
    {
      if (outsideMode == ZERO)
        value = 0;
      else
        value = rmissing_;
    }
  }
  else
    value = rmissing_;

  return(value);

}
```

Listing B.7: Old sampler in volume function.

```
/* new segy.cpp */

/* Unrelated code */

//#define PROFILING

const float segyRMISSING = -99999.0f;

static double noop(const NRLib::SegY* trace, int xind, int yind,
    int zind, double z){
  return z;
}

```

```
static double outsideClosest(const NRLib::SegY* trace, int xind,
    int yind, int zind, double z){
      float z0 = trace->GetZ0();
      float dz = trace->GetDz();
      int nx = trace->GetNx();
      size_t index = AT2D(yind, xind, nx);
      zind = trace->GetLegalIndex(index, zind);
      float value;
      value = trace->GetTraceValue(xind, yind, zind-1);
      if (value == segyRMISSING){
        return z0+zind*dz;             // Want edge value, hence 0/1
            dz_ added
      }else{                            // (0.5 would give center of
          cell).
        return z0+(zind+0.99f)*dz;
      }
}

using std::ios_base;

namespace NRLib{

/* A set of calculations are common for all sampling functions.
    these are calculated upon entry. */
void
SegY::copyGeometryData(const SegyGeometry* geometry_){
  dx_ = geometry_->GetDx();
  dy_ = geometry_->GetDy();
  dxHalf_ = dx_*0.5;
  dyHalf_ = dy_*0.5;
  x0_ = geometry_->GetX0()+0.5*dx_*geometry_->GetCosRot()-0.5*dy_*
      geometry_->GetSinRot();
  y0_ = geometry_->GetY0()+0.5*dy_*geometry_->GetCosRot()+0.5*dx_*
      geometry_->GetSinRot();
  nx_ = geometry_->GetNx();
  time_t ny = geometry_->GetNy();
  tracesSize_ = nx_*ny;
}

/* Unrelated code */

float
SegY::GetValueInVol(int xind, int yind, int zind, double x, double
    y, double z, float v1) const{
#ifdef PROFILING
#ifdef PROFILINGFINEGRAINED
    double wall = omp_get_wtime();
#endif
#endif
  // Computes interpolated value ax^2+by^2+cz^2+dx+ey+fz+g.
  // abcdefg estimated from closest point and its closest
      neighbours.
  float a = 0.0f;
  float b = 0.0f;
  float c = 0.0f;
  float d = 0.0f;
  float e = 0.0f;
  float f = 0.0f;
  float g = v1;

  // Along x:
  GetSample(xind, yind, zind, 1, 0, 0, v1, a, d);
```

```

  // Along y:
  GetSample(xind, yind, zind, 0, 1, 0, v1, b, e);

  // Along z:
  GetSample(xind, yind, zind, 0, 0, 1, v1, c, f);

  float sx =  (x-x0_)*geometry_->GetCosRot() + (y-y0_)*geometry_->
      GetSinRot() + dxHalf_;
  float sy = -(x-x0_)*geometry_->GetSinRot() + (y-y0_)*geometry_->
      GetCosRot() + dyHalf_;
  float ux = static_cast<float>(sx/dx_) - static_cast<float>(floor(
      sx/dx_) + 0.5);
  float uy = static_cast<float>(sy/dy_) - static_cast<float>(floor(
      sy/dy_) + 0.5);
  float uz = static_cast<float>((z-z0_)/dz_) - static_cast<float>(
      floor((z-z0_)/dz_) + 0.5);
  float returner = a*ux*ux + b*uy*uy + c*uz*uz + d*ux + e*uy + f*uz
       + g;

#ifdef PROFILING
#ifdef PROFILINGFINEGRAINED
    wall = omp_get_wtime() - wall;
    Prof::trackTime(wall, GETVALUEINVOL);
#endif
#endif
  return returner;
}

/* This function performs finds two sample values along a given
    axis. */
void
SegY::GetSample(int xind, int yind, int zind, int xoffset, int
    yoffset, int zoffset, float v1, float& a, float& b) const{
  float v0 = 0.0f;
  float v2 = 0.0f;
  float mulV0 = 1.0f;
  v0 = GetTraceValue(xind - xoffset, yind - yoffset, zind - zoffset
      );
  v2 = GetTraceValue(xind + xoffset, yind + yoffset, zind + zoffset
      );
  if(v0 == rmissing_){
    mulV0 = 0.0f;
  }
  if(v2 == rmissing_){
    a = 0;
    b = (v1-v0)*mulV0;
    return;
  }
  a = mulV0*(v2 + v0 - 2*v1)/2.0f;
  b = mulV0*(v2 - v0)/2.0f + (!mulV0)*(v2-v1);
}

/* Unrelated code */

/* This functions performs tricubic interpolation used in CRAVAs
    sampler for SEG Y.
 * This is the "unsafe" version without borderchecks. For the safe
     version see GetValue
 */
float
```

```
SegY::GetValueInterpolated(int xind, int yind, int zind, double x, double y, double z) const
{
#ifdef PROFILING
#ifdef PROFILINGFINEGRAINED
    double wall = omp_get_wtime();
#endif
#endif
  float a = GetTraceValueUnchecked(xind-1, yind, zind);
  float b = GetTraceValueUnchecked(xind, yind-1, zind);
  float c = GetTraceValueUnchecked(xind, yind, zind-1);
  float d = GetTraceValueUnchecked(xind+1, yind, zind);
  float e = GetTraceValueUnchecked(xind, yind+1, zind);
  float f = GetTraceValueUnchecked(xind, yind, zind+1);
  float g = GetTraceValueUnchecked(xind, yind, zind);
  float sx =  (x-x0_)*geometry_->GetCosRot() + (y-y0_)*geometry_->GetSinRot() + dxHalf_;
  float sy = -(x-x0_)*geometry_->GetSinRot() + (y-y0_)*geometry_->GetCosRot() + dyHalf_;
  float ux = static_cast<float>(sx/dx_) - static_cast<float>(floor(sx/dx_) + 0.5);
  float uy = static_cast<float>(sy/dy_) - static_cast<float>(floor(sy/dy_) + 0.5);
  float uz = static_cast<float>((z-z0_)/dz_) - static_cast<float>(floor((z-z0_)/dz_) + 0.5);
  float returner = a*ux*ux + b*uy*uy + c*uz*uz + d*ux + e*uy + f*uz + g;
#ifdef PROFILING
#ifdef PROFILINGFINEGRAINED
    wall = omp_get_wtime() - wall;
    Prof::trackTime(wall, GETVALUEINTERPOLATED);
#endif
#endif
  return returner;
}

/* This function checks and sets the Outside mode. It should be performed once per grid. */
void SegY::setOutsideMode(OutsideMode mode){
  mode_ = mode;
  switch(mode_){
    case CLOSEST:
      outsideSampler_ = &outsideClosest;
      outsideVal_ = rmissing_;
      break;
    case MISSING:
      outsideSampler_ = &noop;
      outsideVal_ = rmissing_;
      break;
    case ZERO:
      outsideSampler_ = &noop;
      outsideVal_ = 0.0f;
      break;
    default:
      break;
  }
}

bool SegY::GetZID(int xind, int yind, int& zind, double x, double y, double& z) const{
  if (z>=z0_ && z<=z0_+nz_*dz_)
  {
```

```
    zind = static_cast<size_t>(floor((z-z0_)/dz_)); //NBNB irap grid rounding different
    z = outsideSampler_(this, xind, yind, zind, z);
    return true;
  }
  return false;
}

float  SegY::GetTraceValue(int xind, int yind, int zind) const{
  size_t index = (yind)*nx_ + (xind);
  if(index >= 0 && index < static_cast<size_t>(tracesSize_) && traces_[index] != NULL && zind >= static_cast<int>(traces_[index]->GetStart()) && zind < static_cast<int>(traces_[index]->GetEnd())){
    return traces_[(yind)*nx_ + (xind)]->GetValueUnchecked(zind);
  }else{
    return rmissing_;
  }
}
  bool SegY::GetXYID(int& xind, int& yind, double& x, double& y) const{
    float xfloatid = 0, yfloatid = 0;
    geometry_->FindContIndex(static_cast<float>(x), static_cast<float>(y), xfloatid, yfloatid);
    xind = static_cast<int>(xfloatid);
    yind = static_cast<int>(yfloatid);
    return true;
  }

  float SegY::GetValue(double x, double y, double z) const{
#ifdef PROFILING
#ifdef PROFILINGFINEGRAINED
  double wall = 0;
    wall = omp_get_wtime();
#endif
#endif
    int xind = 0;
    int yind = 0;
    int zind = 0;
    bool validID = GetID(xind, yind, zind, x, y, z);
    float v1;
    v1 = GetTraceValue(xind, yind, zind);
    float value;
    if (validID && v1 != rmissing_){
      value = GetValueInVol(xind, yind, zind, x, y, z, v1);
    }else{
      value = outsideVal_;
    }
#ifdef PROFILING
#ifdef PROFILINGFINEGRAINED
    wall = omp_get_wtime() - wall;
    Prof::trackTime(wall, GETVALUE);
#endif
#endif
    return value;
  }

  bool SegY::GetID(int& xind, int& yind, int& zind, double& x, double& y, double& z) const{
    return GetXYID(xind, yind, x, y) && GetZID(xind, yind, zind, x, y, z);
  }
```

```
} /* NRLib */
```

Listing B.8: New sampler in volume function.

B.2 Seismic Inversion

```
/* Unrelated code */
#include "src/locks.h"
#include "src/fftwlock.h"

//#define PROFILING
#ifdef PROFILING
#include <sstream>
using std::stringstream;
#include "src/profiling.h"
#define INVERSIONLOG 7
#define INVERSIONREADLOG 8
#define INVERSIONWRITELOG 11
#define INVCPUTIMELOG 9
#define INVERSIONLOOPLOG 10
#endif


#define _USE_MATH_DEFINES
#include <cmath>
#include <ctime>


static int classCnt = 0;
LOCK_T lock;

static fftw_complex** reduceVar = NULL;
static fftw_complex errCorrUnsmoothVal = {0.0f, 0.0f};
static fftw_complex** errVar = NULL;
static fftw_complex * ijkAns = NULL;
static fftw_complex * ijkDataMean = NULL;
static fftw_complex * ijkMean = NULL;
static fftw_complex * ijkMean2 = NULL;
static fftw_complex * ijkRes = NULL;
static fftw_complex * ijkRes2 = NULL;
static fftw_complex * ijkRes3 = NULL;
static fftw_complex** KScc = NULL;
static fftw_complex** KS = NULL;
static fftw_complex** margVar = NULL;
static fftw_complex parSpartialCorrVal = {0.0f, 0.0f};
static fftw_complex** parVar2 = NULL;
static fftw_complex** parVar = NULL;

#if _OPENMP >= 200805
#pragma omp threadprivate(parSpartialCorrVal, ijkMean, ijkMean2,
    ijkDataMean, ijkAns, ijkRes, ijkRes2, ijkRes3, margVar, KScc,
    KS, parVar2, parVar, errVar, reduceVar)
#endif

Crava::Crava(Model * model, SpatialWellFilter * spatwellfilter)
{
  // Since we are using global variables for some of the methods we
      want them to be globally blocking to maintain
  // reentrant properties.
  // That is maintained by locking the affected methods upon entry.
  // To be able to use locks we have to initalize them before usage
      , and destroy them after usage.
  // This is solved by adding a class counter (named classCnt) in
      all CRAVA constructors that ensures that the first crava
      instance
```

```
  // initializes the lock, and the last crava instance destroys the lock (named lock).
#pragma omp critical (CRAVA)
  {
    if(classCnt == 0){
      omp_init_lock(&lock);
    }
    classCnt +=1;
  }
  Utils::writeHeader("Building Stochastic Model");

  double wall=0.0, cpu=0.0;
  TimeKit::getTime(wall,cpu);

  model_               = model;
  nx_                  = model->getBackAlpha()->getNx();
  ny_                  = model->getBackAlpha()->getNy();
  nz_                  = model->getBackAlpha()->getNz();
  nxp_                 = model->getBackAlpha()->getNxp();
  nyp_                 = model->getBackAlpha()->getNyp();
  nzp_                 = model->getBackAlpha()->getNzp();
  lowCut_              = model->getModelSettings()->getLowCut();
  highCut_             = model->getModelSettings()->getHighCut();
  wnc_                 = model->getModelSettings()->getWNC();      // white noise component see crava.h
  energyTreshold_      = model->getModelSettings()->getEnergyThreshold();
  ntheta_              = model->getModelSettings()->getNumberOfAngles();
  fileGrid_            = model->getModelSettings()->getFileGrid();
  outputGridsSeismic_ = model->getModelSettings()->getOutputGridsSeismic();
  outputGridsElastic_ = model->getModelSettings()->getOutputGridsElastic();
  writePrediction_     = model->getModelSettings()->getWritePrediction();
  krigingParameter_    = model->getModelSettings()->getKrigingParameter();
  nWells_              = model->getModelSettings()->getNumberOfWells();
  nSim_                = model->getModelSettings()->getNumberOfSimulations();
  wells_               = model->getWells();
  simbox_              = model->getTimeSimbox();
  meanAlpha_           = model->getBackAlpha();
  meanBeta_            = model->getBackBeta();
  meanRho_             = model->getBackRho();
  correlations_        = model->getCorrelations();
  random_              = model->getRandomGen();
  seisWavelet_         = model->getWavelets();
  A_                   = model->getAMatrix();
  postAlpha_           = meanAlpha_;          // Write over the input to save memory
  postBeta_            = meanBeta_;           // Write over the input to save memory
  postRho_             = meanRho_;            // Write over the input to save memory
  fprob_               = NULL;
  thetaDeg_            = new float[ntheta_];
  empSNRatio_          = new float[ntheta_];
  theoSNRatio_         = new float[ntheta_];
  modelVariance_       = new float[ntheta_];
```

```
signalVariance_       = new float[ntheta_];
errorVariance_        = new float[ntheta_];
dataVariance_         = new float[ntheta_];
scaleWarning_         = 0;
scaleWarningText_     = "";
errThetaCov_          = new double*[ntheta_];
sigmamdnew_           = NULL;
for(int i=0;i<ntheta_;i++) {
  errThetaCov_[i]   = new double[ntheta_];
  thetaDeg_[i]      = static_cast<float>(model->getModelSettings()->getAngle(i)*180.0/M_PI);
}
seisData_  = NULL;

fftw_real * corrT = NULL; // =  fftw_malloc(2*(nzp_/2+1)*sizeof(fftw_real));

// Double-use grids to save memory
FFTGrid * parSpatialCorr   = NULL;  // Parameter correlation
FFTGrid * errCorrUnsmooth = NULL;  // Error correlation

if(!model->getModelSettings()->getForwardModeling())
{
  seisData_          = model->getSeisCubes();
  model->releaseGrids();
  correlations_->createPostGrids(nx_,ny_,nz_,nxp_,nyp_,nzp_,fileGrid_);
  parPointCov_     = correlations_->getPriorVar0();
  parSpatialCorr   = correlations_->getPostCovAlpha();  // Double-use grids to save memory
  errCorrUnsmooth = correlations_->getPostCovBeta();    // Double-use grids to save memory
  // NBNB   nzp_*0.001*corr->getdt() = T    lowCut = lowIntCut*domega = lowIntCut/T
  int lowIntCut = int(floor(lowCut_*(nzp_*0.001*correlations_->getdt())));
  // computes the integer whis corresponds to the low cut frequency.
  float corrGradI, corrGradJ;
  model->getCorrGradIJ(corrGradI, corrGradJ);
  corrT = parSpatialCorr->fillInParamCorr(correlations_,lowIntCut,corrGradI, corrGradJ);
  if(spatwellfilter!=NULL)
  {
    parSpatialCorr->setAccessMode(FFTGrid::RANDOMACCESS);
    for(int i=0;i<nWells_;i++)
      spatwellfilter->setPriorSpatialCorr(parSpatialCorr, wells_[i], i);
    parSpatialCorr->endAccess();
  }
  correlations_->setPriorCorrTFiltered(corrT,nz_,nzp_); // Can has zeros in the middle
  errCorrUnsmooth->fillInErrCorr(correlations_,corrGradI,corrGradJ);
  if((model->getModelSettings()->getOtherOutputFlag() & IO::PRIORCORRELATIONS) > 0)
    correlations_->writeFilePriorCorrT(corrT,nzp_);      // No zeros in the middle
}
else
{
  model->releaseGrids();
```

```
  }

  // reality check: all dimensions involved match
  assert(meanBeta_->consistentSize(nx_,ny_,nz_,nxp_,nyp_,nzp_));
  assert(meanRho_->consistentSize(nx_,ny_,nz_,nxp_,nyp_,nzp_));

  for(int i=0 ; i< ntheta_ ; i++)
  {
    if(!model->getModelSettings()->getForwardModeling())
      assert(seisData_[i]->consistentSize(nx_,ny_,nz_,nxp_,nyp_,
          nzp_));
    assert(seisWavelet_[i]->consistentSize(nzp_));
  }

  if(!model->getModelSettings()->getForwardModeling())
  {
    parSpatialCorr->fftInPlace();
    computeVariances(corrT,model->getModelSettings());
    scaleWarning_ = checkScale();  // fills in scaleWarningText_ if
         needed.
    fftw_free(corrT);
    if(simbox_->getIsConstantThick() == false)
      divideDataByScaleWavelet();
    errCorrUnsmooth->fftInPlace();
    for(int i = 0 ; i < ntheta_ ; i++)
    {
      seisData_[i]->setAccessMode(FFTGrid::RANDOMACCESS);
      seisData_[i]->fftInPlace();
      seisData_[i]->endAccess();
    }
  }

  if ((model->getModelSettings()->getEstimateFaciesProb() && model
      ->getModelSettings()->getFaciesProbRelative())
      || model->getModelSettings()->getUseLocalNoise())
  {
  //  meanAlpha_->setAccessMode(FFTGrid::READ);
    meanAlpha2_ = copyFFTGrid(meanAlpha_);
  //  meanAlpha2_->endAccess();
   // meanBeta_->setAccessMode(FFTGrid::READ);
  //  meanRho_->setAccessMode(FFTGrid::READ);

    meanBeta2_  = copyFFTGrid(meanBeta_);
    meanRho2_   = copyFFTGrid(meanRho_);
  }

  meanAlpha_->fftInPlace();
  meanBeta_ ->fftInPlace();
  meanRho_  ->fftInPlace();

  Timings::setTimeStochasticModel(wall,cpu);
}

Crava::~Crava()
{
  delete [] thetaDeg_;
  delete [] empSNRatio_;
  delete [] theoSNRatio_;
  delete [] modelVariance_;
  delete [] signalVariance_;
  delete [] errorVariance_;
  delete [] dataVariance_;
```

```
  if(fprob_!=NULL) delete fprob_;

  for(int i = 0;i<ntheta_;i++)
    delete[] errThetaCov_[i];
  delete [] errThetaCov_;

  if(postAlpha_!=NULL) delete  postAlpha_ ;
  if(postBeta_!=NULL)  delete  postBeta_;
  if(postRho_!=NULL)   delete  postRho_  ;

  const int nx = nx_;
  const int ny = ny_;
  if(sigmamdnew_!=NULL)
  {
   int i, j;
#pragma omp parallel if(nx*ny > SIMPLEOVERHEADLIMIT) private(i, j)
#pragma omp for
    for(i=0;i<nx;i++)
    {
      for(j=0;j<ny;j++)
      {
        if((*sigmamdnew_)(i,j)!=NULL)
        {
          for(int ii=0;ii<3;ii++)
            delete [] (*sigmamdnew_)(i,j)[ii];
          delete [] (*sigmamdnew_)(i,j);
        }
      }
    }
    delete sigmamdnew_;
  }

//This is the global lock destructor. See the comments in the constructor for a full description.
#pragma omp critical (CRAVA)
  {
    classCnt -=1;
    if(classCnt == 0){
      omp_destroy_lock(&lock);
    }
  }
}

/* This comment replaces 873 lines not related to inversion */

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) +
          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);


  //  long int timestart, timeend;
  //  time(&timestart);

  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[l].im  /= errorSmooth3[l]->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 readSpinlock = 0;
     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++){
       int idI = k;
       int idJ = j/cnxp;
       int idK = j%cnxp;
       //START READ
       // A ordered section ensures sequential ordering
       // Sequential ordering is esential in this part in order to retain sequential I/O to disks.
       while(sequentialInput && readSpinlock != j);
#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
#pragma omp atomic
       readSpinlock += 1;
#pragma omp flush(readSpinlock)
       //END READ
```

```
        //START COMPUTE

        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());
        }

        // END COMPUTE
        // START WRITE
        // 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(sequentialOutput && 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)
        //END WRITE
      }
    }
#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;

//  time(&timeend);
// LogKit::LogFormatted(LogKit::LOW,"\n Core inversion finished after %ld seconds ***\n",timeend-timestart);
// 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();
  }

  //NBNB Anne Randi: Skaler traser ihht notat fra Hugo

  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);
}
/* Below this comment were 1227 lines of code not rellevant for
    inversion */
```

Listing B.9: New Seismic Inversion.

```
/* Above this comment the original file had ~3000 lines of code not
     rellated to inversion */

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

  double wall=0.0, cpu=0.0;
  TimeKit::getTime(wall,cpu);
  int i,j,k,l,m;

  fftw_complex * kW           = new fftw_complex[ntheta_];

  fftw_complex * errMult1     = new fftw_complex[ntheta_];
  fftw_complex * errMult2     = new fftw_complex[ntheta_];
  fftw_complex * errMult3     = new fftw_complex[ntheta_];

  fftw_complex * ijkData      = new fftw_complex[ntheta_];
  fftw_complex * ijkDataMean  = new fftw_complex[ntheta_];
  fftw_complex * ijkRes       = new fftw_complex[ntheta_];
  fftw_complex * ijkMean      = new fftw_complex[3];
  fftw_complex * ijkAns       = new fftw_complex[3];
  fftw_complex   kD,kD3;
  fftw_complex   ijkParLam;
  fftw_complex   ijkErrLam;
  fftw_complex   ijkTmp;

  fftw_complex**  K  = new fftw_complex*[ntheta_];
  for(i = 0; i < ntheta_; i++)
    K[i] = new fftw_complex[3];

  fftw_complex**  KS  = new fftw_complex*[ntheta_];
  for(i = 0; i < ntheta_; i++)
    KS[i] = new fftw_complex[3];

  fftw_complex**  KScc  = new fftw_complex*[3]; // cc - complex
     conjugate (and transposed)
  for(i = 0; i < 3; i++)
    KScc[i] = new fftw_complex[ntheta_];

  fftw_complex**  parVar = new fftw_complex*[3];
  for(i = 0; i < 3; i++)
    parVar[i] = new fftw_complex[3];

  fftw_complex**  margVar = new fftw_complex*[ntheta_];
  for(i = 0; i < ntheta_; i++)
    margVar[i] = new fftw_complex[ntheta_];

  fftw_complex**  errVar = new fftw_complex*[ntheta_];
  for(i = 0; i < ntheta_; i++)
    errVar[i] = new fftw_complex[ntheta_];

```

```
  fftw_complex** reduceVar = new fftw_complex*[3];
  for(i = 0; i < 3; i++)
    reduceVar[i]= new fftw_complex[3];

  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
/* 49 lines of unrellated comments */
    }
  }
  delete[] errorSmooth;
  delete[] errorSmooth2;

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

```
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;
//   long int timestart, timeend;
//   time(&timestart);
float realFrequency;

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  ^";

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()
    }

    for( j = 0; j < nyp_; j++) {
      for( i = 0; i < cnxp; i++) {
        ijkMean[0] = meanAlpha_->getNextComplex();
        ijkMean[1] = meanBeta_ ->getNextComplex();
        ijkMean[2] = meanRho_  ->getNextComplex();

        for(l = 0; l < ntheta_; l++ )
        {
          ijkData[l] = seisData_[l]->getNextComplex();
          ijkRes[l]  = ijkData[l];
        }

        ijkTmp        = parSpatialCorr->getNextComplex();
        ijkParLam.re = float ( sqrt(ijkTmp.re * ijkTmp.re));
        ijkParLam.im = 0.0;

        for(l = 0; l < 3; l++ )
          for(m = 0; m < 3; m++ )
          {
            parVar[l][m].re = parPointCov_[l][m] * ijkParLam.re;
            parVar[l][m].im = 0.0;
            // if(l!=m)
            //    parVar[l][m].re *= 0.75;   //NBNB OK DEBUG TEST
          }

```

```
        priorVarVp = parVar[0][0].re;
        ijkTmp     = errCorrUnsmooth->getNextComplex();
        ijkErrLam.re = float( sqrt(ijkTmp.re * ijkTmp.re));
        ijkErrLam.im = 0.0;

        if(realFrequency > lowCut_*simbox_->getMinRelThick() &&
            realFrequency < highCut_) // inverting only relevant
            frequencies
        {
          for(l = 0; l < ntheta_; l++ )
            for(m = 0; m < ntheta_; 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  = float( 0.5*(1.0-wnc_)*
                  errThetaCov_[l][m] * ijkErrLam.re * ( errMult1[
                  l].re *  errMult1[m].re +  errMult1[l].im *
                  errMult1[m].im));
              errVar[l][m].re += float( 0.5*(1.0-wnc_)*
                  errThetaCov_[l][m] * ijkErrLam.re * ( errMult2[
                  l].re *  errMult2[m].re +  errMult2[l].im *
                  errMult2[m].im));
              if(l==m) {
                errVar[l][m].re += float( wnc_*errThetaCov_[l][m]
                    * errMult3[l].re  * errMult3[l].re);
                errVar[l][m].im   = 0.0;
              }
              else {
                errVar[l][m].im  = float( 0.5*(1.0-wnc_)*
                    errThetaCov_[l][m] * ijkErrLam.re * (-
                    errMult1[l].re * errMult1[m].im + errMult1[l
                    ].im * errMult1[m].re));
                errVar[l][m].im += float( 0.5*(1.0-wnc_)*
                    errThetaCov_[l][m] * ijkErrLam.re * (-
                    errMult2[l].re * errMult2[m].im + errMult2[l
                    ].im * errMult2[m].re));
              }
            }

            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
```

```
                  //double hj=1000000.0;
                  //if(reduceVar[0][0].im!=0)
                  // hj = MAXIM(reduceVar[0][0].re/reduceVar[0][0].im
                      ,-reduceVar[0][0].re/reduceVar[0][0].im); //
                      NBNB DEBUG
                  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
                }

                // quality control DEBUG
                if(priorVarVp*4 < ijkAns[0].re*ijkAns[0].re + ijkAns
                    [0].re*ijkAns[0].re)
                {
                  justfactor = sqrt(ijkAns[0].re*ijkAns[0].re +
                      ijkAns[0].re*ijkAns[0].re)/sqrt(priorVarVp);
                }
            }
            postAlpha_->setNextComplex(ijkMean[0]);
            postBeta_ ->setNextComplex(ijkMean[1]);
            postRho_  ->setNextComplex(ijkMean[2]);
            postCovAlpha->setNextComplex(parVar[0][0]);
            postCovBeta ->setNextComplex(parVar[1][1]);
            postCovRho  ->setNextComplex(parVar[2][2]);
            postCrCovAlphaBeta->setNextComplex(parVar[0][1]);
            postCrCovAlphaRho ->setNextComplex(parVar[0][2]);
            postCrCovBetaRho  ->setNextComplex(parVar[1][2]);

            for(l=0;l<ntheta_;l++)
              seisData_[l]->setNextComplex(ijkRes[l]);
        }
      }
      // Log progress
      if (k+1 >= static_cast<int>(nextMonitor))
      {
        nextMonitor += monitorSize;
        std::cout << "^";
        fflush(stdout);
      }
    }
    std::cout << "\n";

    //  time(&timeend);
    // LogKit::LogFormatted(LogKit::LOW,"\n Core inversion finished
        after %ld seconds ***\n",timeend-timestart);
    // 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++)
  delete seisData_[l];
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();
  }

  //NBNB Anne Randi: Skaler traser ihht notat fra Hugo

  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();

  delete [] seisData_;
  delete [] kW;
  delete [] errMult1;
  delete [] errMult2;
  delete [] errMult3;
  delete [] ijkData;
  delete [] ijkDataMean;
  delete [] ijkRes;
  delete [] ijkMean ;
  delete [] ijkAns;
  delete    diff1Operator;
  delete    diff3Operator;

  for(i = 0; i < ntheta_; i++)
  {
    delete[]  K[i];
    delete[]  KS[i];
    delete[]  margVar[i] ;
    delete[] errVar[i] ;
    delete errorSmooth3[i];
  }
  delete[] K;
  delete[] KS;
  delete[] margVar;
  delete[] errVar  ;
  delete[] errorSmooth3;

  for(i = 0; i < 3; i++)
  {
    delete[] KScc[i];
    delete[] parVar[i] ;
    delete[] reduceVar[i];
  }
  delete[] KScc;
  delete[] parVar;
  delete[] reduceVar;
```

```

  Timings::setTimeInversion(wall,cpu);
  return(0);
}
/* Below this comment were 1225 lines of code not rellevant for
   inversion */
```

Listing B.10: Old Seismic Inversion.

B.3 FFTW Settings Store

```
#ifndef FFTWLOCK_H_
#define FFTWLOCK_H_

#include "src/locks.h"
#include "fft/include/fftw.h"
#include "fft/include/rfftw.h"
#include "nrlib/iotools/logkit.hpp"

#include <iterator>
#include <map>
#include <string>

class FftwSettings{
  int nx, ny, nz;
  fftw_direction direction;
  int *n;
  int rank;
  int flags;
  int dimensions;
public:

  FftwSettings():
  nx(0),
  ny(0),
  nz(0),
  direction(FFTW_FORWARD),
  n(NULL),
  rank(0),
  flags(0),
  dimensions(0)
  {}

  FftwSettings(int nx, int ny, int nz, fftw_direction direction, int flags){
    this->nx = nx;
    this->ny = ny;
    this->nz = nz;
    this->direction = direction;
    this->dimensions = 3;
    this->n = NULL;
    this->rank = 0;
    this->flags = flags;
  }

  FftwSettings(int rank, const int* n, fftw_direction direction, int flags):
  nx(0),
  ny(0),
  nz(0),
  direction(direction),
  n(NULL),
  rank(rank),
  flags(flags),
  dimensions(1){
    this->n = new int[rank];
    for(int i = 0; i < rank; i++) this->n[i] = n[i];
  }

  FftwSettings(const FftwSettings& s){
    nx = s.nx;
```

```
    ny = s.ny;
    nz = s.nz;
    dimensions = s.dimensions;
    direction = s.direction;
    n = new int[s.rank];
    for(int i = 0; i < s.rank; i++) n[i] = s.n[i];
    rank = s.rank;
    flags = s.flags;
  }

  ~FftwSettings(){
    if(n != NULL){
      delete[] n;
    }
  }

  std::string toString() const;

  int getNx() const{ return nx; }
  int getNy() const{ return ny; }
  int getNz() const{ return nz; }
  const int* getN() const{ return n; }
  int getRank() const{ return rank; }
  int getFlags() const{ return flags; }
  fftw_direction getDirection() const{ return direction; }
  int getDimension() const{ return dimensions; }

  friend bool operator<(const FftwSettings& a, const FftwSettings&
      b){
    bool result =
      a.dimensions < b.dimensions ||
      a.direction < b.direction ||
      a.flags < b.flags ||
      a.nx < b.nx ||
      a.ny < b.ny ||
      a.nz < b.nz;
    for(int i = 0; i < a.rank && i < b.rank; i++){
      result |= (a.n[i] < b.n[i]);
    }
    return result;
  }

  bool operator()(const FftwSettings* a, const FftwSettings* b)
      const{
    return (*a) < (*b);
  }
  bool operator()(const FftwSettings& a, const FftwSettings& b)
      const{
    return a < b;
  }
};

class FftwLock{
  LOCK_T plan_l_;
  fftwnd_plan plan_;
public:
  FftwLock(){
    omp_init_lock(&plan_l_);
    plan_ = NULL;
  }

  FftwLock(const FftwSettings& s):
```

```
  plan_(NULL)
  {
    omp_init_lock(&plan_l_);
    switch(s.getDimension()){
      case 3:
        plan_ = rfftw3d_create_plan(s.getNx(), s.getNy(), s.getNz()
            , s.getDirection(), s.getFlags());
        break;
//      case 2:
//        plan_ = rfftw2d_create_plan(s.nx, s.ny, s.dir, s.flags);
//        break;
      case 1:
        plan_ = rfftwnd_create_plan(s.getRank(), s.getN(), s.
            getDirection(), s.getFlags());
        break;
      default:
        break;
    }
  }

  ~FftwLock(){
    omp_destroy_lock(&plan_l_);
    if(plan_ != NULL) fftwnd_destroy_plan(plan_);
  }

  void lock(){
    omp_set_lock(&plan_l_);
  }

  void unlock(){
    omp_unset_lock(&plan_l_);
  }

  fftwnd_plan getPlan() const{ return plan_; }   //Figure out why
      the return type need to be consted.

};

class FftwStore{
  double wall;
  std::map<const FftwSettings*, FftwLock*, FftwSettings> rToC;
  std::map<const FftwSettings*, FftwLock*, FftwSettings> cToR;
  int cachehits;
  LOCK_T insert_l_;

  FftwLock* getPlanAndLock(const FftwSettings& s, std::map<const
      FftwSettings*, FftwLock*, FftwSettings>* m){
    std::map<const FftwSettings*, FftwLock*, FftwSettings>::
        iterator it = m->find( &s );
    if(it == m->end()){
      FftwLock* l;
      omp_set_lock(&insert_l_);
      it = m->find( &s );
      if(it == m->end()){
        const FftwSettings* key = new FftwSettings(s);
        l = new FftwLock(s);
        //NRLib::LogKit::LogFormatted(NRLib::LogKit::HIGH, "NEW: %s
            \n", s.toString().c_str());
        if(l->getPlan() != NULL){
          m->insert(std::pair<const FftwSettings*, FftwLock*>(key,
              l));
        }else{
```

```
        //NRLib::LogKit::LogFormatted(NRLib::LogKit::HIGH, "Failing cache");
          delete l;
          l = NULL;
        }
      }else{
        //NRLib::LogKit::LogFormatted(NRLib::LogKit::HIGH, "Second: %s\n", s.toString().c_str());
#pragma omp atomic
        cachehits += 1;
        l = it->second;
      }
      omp_unset_lock(&insert_l_);
      return l;
    }
#pragma omp atomic
    cachehits += 1;
    return it->second;
  }

public:
  FftwStore();

  ~FftwStore();

  int getCacheHits(){ return cachehits; }


  void one_complex_to_real(fftw_complex* cData, fftw_real* rData, const FftwSettings& s);

  void one_real_to_complex(fftw_real* rData, fftw_complex* cData, const FftwSettings& s);

  int getHits() const{ return cachehits; }
  int getNumberOfSettings() const{ return rToC.size() + cToR.size(); }
};

FftwStore* getFFTStorage();

#endif
```

Listing B.11: FFTW setting store definition.

```
#include "src/fftwlock.h"
#include "src/profiling.h"
#include "src/locks.h"
#include "nrlib/iotools/logkit.hpp"
#include <sstream>

#define SETTINGSTHRESHOLD 1
#define FFTW
#define PROFILINGFFTW 1022

using namespace NRLib;
using std::string;
using std::stringstream;

static FftwStore storage;

string FftwSettings::toString() const{
```

```
  stringstream ss;
  ss << "[nx: ";
  ss << nx;
  ss << string(", ny: ");
  ss << ny;
  ss << string(", nz: ");
  ss << nz;
  ss << string(", direction: ");
  ss << static_cast<int>(direction);
  ss << string(", rank: ");
  ss << rank;
  ss << string(", flags: ");
  ss << flags;
  ss << string(", dim: ");
  ss << dimensions;
  ss << "]";
  return ss.str();
}

FftwStore* getFFTStorage(){
  return &storage;
}


FftwStore::FftwStore(): cachehits(0)
{
  wall = 0.0;
  omp_init_lock(&insert_l_);
}

FftwStore::~FftwStore(){
#pragma omp critical (io)
  {
#ifdef PROFILINGFFTW
    LogKit::SetFileLog( string("fftw.log"), PROFILINGFFTW, false);
#endif
#ifndef FFTW
    LogKit::LogMessage(PROFILINGFFTW, "FFTW Lookup\n");
#else
    LogKit::LogMessage(PROFILINGFFTW, "FFTW Dump no reuse of
        fftw_plans \n");
#endif
    LogKit::LogFormatted(PROFILINGFFTW, "FFTW cacheHits: %i\n",
        getHits());
    LogKit::LogFormatted(PROFILINGFFTW, "FFTW settings: %i\n",
        getNumberOfSettings());
    LogKit::LogFormatted(PROFILINGFFTW, "FFTW wall time: %i ms\n",
        static_cast<int>(1000*wall));
    if(getNumberOfSettings() > SETTINGSTHRESHOLD){
      LogKit::LogMessage(PROFILINGFFTW, "FFTW SETTINGS:\n");
      LogKit::LogMessage(PROFILINGFFTW, "FFTW real to complex:\n");
      int cntr = 0;
      for(std::map<const FftwSettings*, FftwLock*, FftwSettings>::
          const_iterator it = rToC.begin(); it != rToC.end(); it++)
          {
        LogKit::LogFormatted(PROFILINGFFTW, "FFTW\t%s\n", it->first
            ->toString().c_str());
        cntr++;
        if(cntr > 20) break;
      }
      LogKit::LogMessage(PROFILINGFFTW, "FFTW complex to real:\n");
```

```
      for(std::map<const FftwSettings*, FftwLock*, FftwSettings>::
          const_iterator it = cToR.begin(); it != cToR.end(); it++)
          {
        LogKit::LogFormatted(PROFILINGFFTW, "FFTW\t%s\n", it->first
            ->toString().c_str());
        cntr++;
        if(cntr > 20) break;
      }
    }
#ifdef PROFILINGFFTW
    LogKit::EndLog();
#endif
  }

  std::map<const FftwSettings*, FftwLock*, FftwSettings>::iterator
      it;
  for(it = rToC.begin(); it != rToC.end(); it++){
    delete it->first;
    delete it->second;
  }
  for(it = cToR.begin(); it != cToR.end(); it++){
    delete it->first;
    delete it->second;
  }
  omp_destroy_lock(&insert_l_);
}

 void FftwStore::one_complex_to_real(fftw_complex* cData, fftw_real
      * rData, const FftwSettings& s){
  double curWall = 0;
#pragma omp master
  curWall = omp_get_wtime();

#ifndef FFTW
  FftwLock* fl = NULL;
  while(fl == NULL) fl = getPlanAndLock(s, &cToR);

  fl->lock();
#pragma omp master
  curWall = omp_get_wtime() - curWall;
  wall += curWall;
  rfftwnd_one_complex_to_real((fl->getPlan()), cData, rData);
#pragma omp master
  curWall = omp_get_wtime();
  fl->unlock();
#pragma omp master
  curWall = omp_get_wtime() - curWall;
#else
     rfftwnd_plan p2;
     if(s.getDimension() > 1){
     p2 = rfftw3d_create_plan(s.getNx(), s.getNy(), s.getNz(), s.
         getDirection(), s.getFlags());
     }else{
     p2 = rfftwnd_create_plan(1, s.getN(), s.getDirection(), s.
         getFlags());
     }
#pragma omp master
  curWall = omp_get_wtime() - curWall;
     wall += curWall;
     rfftwnd_one_complex_to_real(p2, cData, rData);
#pragma omp master
  curWall = omp_get_wtime();
```

```
      fftwnd_destroy_plan(p2);
#pragma omp master
    curWall = omp_get_wtime() - curWall;
#endif
      wall += curWall;
}

  void FftwStore::one_real_to_complex(fftw_real* rData, fftw_complex* cData, const FftwSettings& s){
    double curWall = 0.0;
#pragma omp master
    curWall = omp_get_wtime();
#ifndef FFTW

    FftwLock* fl = NULL;
    while(fl == NULL) fl = getPlanAndLock(s, &rToC);
    fl->lock();
#pragma omp master
    curWall = omp_get_wtime() - curWall;
    wall += curWall;
    rfftwnd_one_real_to_complex((fl->getPlan()), rData, cData);
#pragma omp master
    curWall = omp_get_wtime();
    fl->unlock();
#pragma omp master
    curWall = omp_get_wtime() - curWall;
#else
      rfftwnd_plan p1;
      if(s.getDimension() > 1){
      p1 = rfftw3d_create_plan(s.getNx(), s.getNy(), s.getNz(), s.getDirection(), s.getFlags());
      }else{
      p1 = rfftwnd_create_plan(1, s.getN(), s.getDirection(), s.getFlags());
      }
#pragma omp master
    curWall = omp_get_wtime() - curWall;
      wall += curWall;
      rfftwnd_one_real_to_complex(p1, rData, cData);
#pragma omp master
    curWall = omp_get_wtime();
      fftwnd_destroy_plan(p1);
#pragma omp master
    curWall = omp_get_wtime() - curWall;
#endif
      wall += curWall;
}
```

Listing B.12: FFTW settings store.

B.4 OpenMP Header

```
#ifndef __LOCKS_H__
#define __LOCKS_H__

#ifdef _OPENMP
  #include <omp.h>
  #define LOCK_T omp_lock_t
#else
  #define LOCK_T void*
  #define omp_init_lock(a) 0
  #define omp_set_lock(a) 0
  #define omp_unset_lock(a) 0
  #define omp_destroy_lock(a) 0
  #define omp_test_lock(a) 0
  #define omp_get_num_threads() 1
  #define omp_get_thread_num() 0
  #define omp_get_max_threads() 1
  #define omp_get_wtime() 0.0
#endif

static const int BASICOVERHEADLIMIT = 1000;
static const int SIMPLEOVERHEADLIMIT = BASICOVERHEADLIMIT/2;
static const int MEDIUMOVERHEADLIMIT = BASICOVERHEADLIMIT/4;
void init_omp();

#endif
```

Listing B.13: Include of OpenMP with fallback for compilers without OpenMP.

Appendix C

Summarizing Poster

Parallelization Techniques for Seismic Inversion Codes

Andreas D. Hysing (Master Student), Dr. Anne C. Elster (Advisor) and Alf Birger Raustad (Advisor)

Dept. of Computer and Inforamtion Science, Norwegian University of Science and Technology

This thesis focused on improving performance of a scientific application for seismic processing of geophysical data. The process performs a new process called seismic inversion to calculate earth model parameters Vp (pressure-wave velocity), Vs (shear-wave velocity), and ρ (density). Input data are amplitude cubes stacked at different angles and corresponding wavelets. Output are cubes of inverted elastic properties. The application also offers the possibility to investigate the accuracy in the results by calculating uncertainties, or by simulating (Monte Carlo) cubes of possible elastic properties.

Independant part was parallized using Parallelization Techniques for Seismic Inversion Codes Parallelization Techniques for Seismic Inversion Codes openMP utilize the full performance of all modern CPUs.

The test program CRAVA:

Stands for:
Conditioning Reservoir variables to Amplitude Versus Angle data.

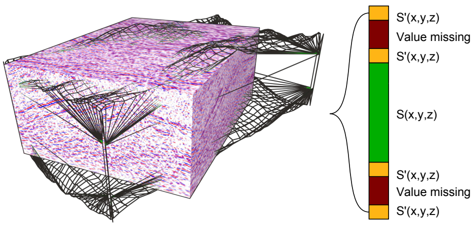
Developed by:
Norwegian Computing Center in collaboration with Statoil

Is built up of 128540 lines of C++ with and 57514 without libraries.

Utilizes boost and FFTW

Implements:
Buland, A. and Omre, H. (2003). Bayesian linearized AVO inversion

http://www.nr.no/pages/sand/area_res_char_crava



Seismic resampling is performed in the intersection between a seismic inversion volume and raw data from SEGY files. Sampling function is S(x,y,z). If some neighbors are missing the sampling falls back to S'(x,y,z)

$$s_{x,y,z} = d_{x-1,y,z} \cdot x^2 + d_{x,y-1,z} \cdot x^2 + d_{x,y,z-1} \cdot z^2 + d_{x+1,y,z} \cdot x + d_{x,y+1,z} \cdot y + d_{x,y,z+1} \cdot z + d_{x,y,z}$$

$$s'_{x,y,z} = d_{x-1,y,z} \cdot x + d_{x,y-1,z} \cdot x + d_{x,y,z-1} \cdot z$$

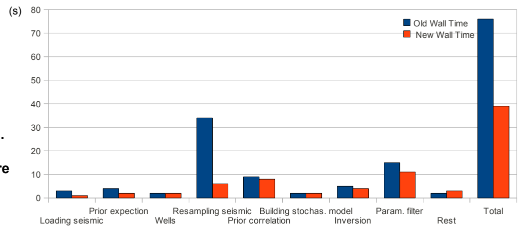
Program profiling results showed that 60% of wall time was used on cubic sampling the input grid from raw data. The area of interest was only between a top and bottom surface.
- **The program logic contained no data dependance on two of three data axies.**
- **Redundant boundary conditions resulted in unwanted pipeline stalls.**
- **Data padding was performed by redundant sampling with a cubic fadeoff.**

Solution:
- **Parallize independant semismic traces with openMP.**
- **Recognize boundaries, remove boundary checks inside volume in sample function.**
- **Add custom cases for padding.**

General speedup:

Only a few different properties were used for FFTW. Costly FFTW settings were cached in a thread safe settings store between independent computations. Resulting in ~ 20 cache hits.

Compile time parameters for functions were evaluated through extensive use of c++ templates.



Performance comparison of different phases of serial and optimized program. Prior expection has gone down from 34 to 9 seconds. Total Walltime went from 76 to 37 seconds.

HPC-Lab NTNU
Norwegian University of Science and Technology

Figure C.1: Thesis poster displayed the NTNU booth at ISC10.