



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

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 performance applications 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 spawning 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

Abstract	i
Acknowledgments	iii
Table of Contents	vii
Table of Algorithms	ix
Table of Figures	xi
Table of Listings	xiii
Table of Tables	xv
1 Introduction	1
1.1 Thesis Description	1
1.2 Project Boundaries	2
1.3 Choosing OpenMP Threading Model	3
1.4 Outline	3
2 Modern Computer Design	5
2.1 Parallel Programming Concepts	6
3 Parallel Programming	7
3.1 Parallel Programming Models	7

CONTENTS

3.1.1	Cluster Computing	7
3.1.2	GPGPU	8
3.2	Parallel Programming Technologies	8
3.2.1	MPI	8
3.2.2	OpenMP	9
3.3	Project State	11
3.3.1	CRAVA	11
3.3.2	Testing	12
3.3.3	Profiling Techniques	12
4	Case Study: Seismic Inversion	15
4.1	Seismic Acquisition	15
4.2	Geostatistical processing	15
4.3	Statistical Model	16
4.3.1	Computational Costs	17
5	Performance Optimizations of CRAVA	19
5.1	Resampling	20
5.1.1	Implementation	20
5.2	Seismic Inversion	22
5.3	FFTW Settings Store	26
6	Results and Future Work	29
6.1	Resampling	29
6.1.1	Parallel Scalability	30
6.2	Seismic Inversion	33
6.2.1	Parallel Scalability	34
6.3	FFTW Settings Store	35

7	Conclusions and Future Work	37
7.1	Comparison of Serial and Parallel Optimization	37
7.2	Future Work	38
7.2.1	Seismic Inversion	38
	Bibliography	40
	Appendices	40
A	Tests	43
A.1	Benchmark Test	43
A.2	Correctness Test	44
B	Code	47
B.1	Resampling	47
B.2	Seismic Inversion	89
B.3	FFTW Settings Store	113
B.4	OpenMP Header	120
C	Summarizing Poster	121

CONTENTS

List of Algorithms

1	Old approach for sampling grids with per sample outside mode . .	21
2	New approach for sampling grids outside mode.	22
3	Parallel seismic inversion approach.	23
4	FFTW settings store algorithm.	27

LIST OF ALGORITHMS

List of Figures

3.1	Input and Output of CRAVA.	11
5.1	Resampling area of interest.	19
5.2	Computation pipelining.	24
5.3	Dependency graph for seismic inversion sub tasks.	24
6.1	Resampling scalability	31
6.2	Parallel Seismic Inversion CPU time without error calculation. . .	32
6.3	Parallel Seismic Inversion wall time.	34
C.1	Thesis poster displayed the NTNU booth at ISC10.	122

LIST OF FIGURES

Listings

3.1	Matrix multiply with OpenMP.	9
3.2	Task parallelism with OpenMP.	9
3.3	Initial program profiling of the Snorre field on system 6.1.1. . . .	12
5.1	Example of a spinlock.	25
5.2	Typical use scenario for FFTW.	26
B.1	Old FFTGrid class definition. Contains resampling functionality.	47
B.2	New FFTGrid class definition. Contains resampling functionality.	52
B.3	Old volume resampling functions.	61
B.4	New volume resampling functions.	65
B.5	Old sampler in volume function header.	70
B.6	New sampler in volume function header.	74
B.7	Old sampler in volume function.	81
B.8	New sampler in volume function.	83
B.9	New Seismic Inversion.	89
B.10	Old Seismic Inversion.	104
B.11	FFTW setting store definition.	113
B.12	FFTW settings store.	116
B.13	Include of OpenMP with fallback for compilers without OpenMP.	120

LISTINGS

List of Tables

6.1	Average timing results of sampler functions on system 6.1.1 with 1 active core.	30
6.2	Overall serial improvements on resampling for the benchmark test on system 6.1.1.	30
6.3	Resampling timings and parallel fractions on 6.1.1	31
6.4	4 core system at NTNU type workstation.	33
6.5	4 core system at NTNU type cluster node.	33
6.6	2 core system at Statoil type workstation.	33
6.7	8 core system at Statoil type cluster node.	34
6.8	Average CPU time per sample used on seismic inversion.	35
6.9	Time usage of FFTW plan generation on selected correctness tests on system 6.1.1.	35
A.1	Benchmark tests most important characteristics.	43
A.2	Correctness tests most important characteristics.	44
A.3	Correctness tests most important characteristics continued.	45

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 software¹. 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.

¹In 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 clock 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 reuses 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

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

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

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

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

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

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.

¹A 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 led 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 exist. Some of these specifications are developed by academia while others 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.

```
1 #include <omp.h>
2
3 int i, j, k;
4 #pragma omp parallel
5 {
6 #pragma for private(i, j)
7     for(i = 0; i < rows_matrix1; i++){
8         for(j = 0; j < columns_matrix2; j++){
9             result[i][j] = 0.0
10            for(k = 0; k < rows_matrix2; k++){
11                result[i][j] += first[i][k] * second[k][j];
12            }
13        }
14    }
15 }
```

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.

```
1 #include <omp.h>
2
3 int computation1() {
4     /* A function performing some computation */
5 }
6
7 int computation2() {
8     /* A function performing some other computation */
9 }
```

```
10 |
11 | int main(int argc, char** arv){
12 | #pragma omp parallel
13 |     {
14 | #pragma omp sections
15 |     {
16 | #pragma omp section
17 |         computation1();
18 | #pragma omp section
19 |         computation2();
20 |     }
21 | }
22 | /* serial code */
23 | return 0;
24 | }
```

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. CRAVA's 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.

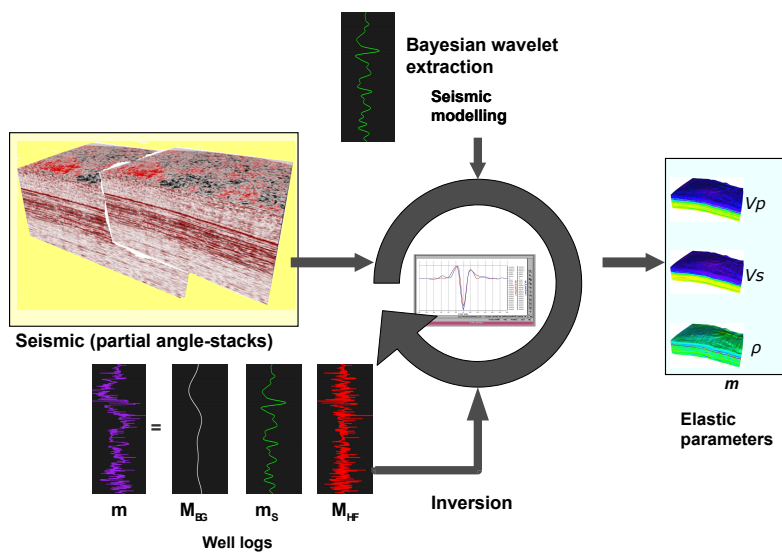


Figure 3.1: Input and Output of CRAVA. Printed with permission from Statoil.

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 expectation	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.

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

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

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

Equation 4.1 can be written on matrix form as

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

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].

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

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

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

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.

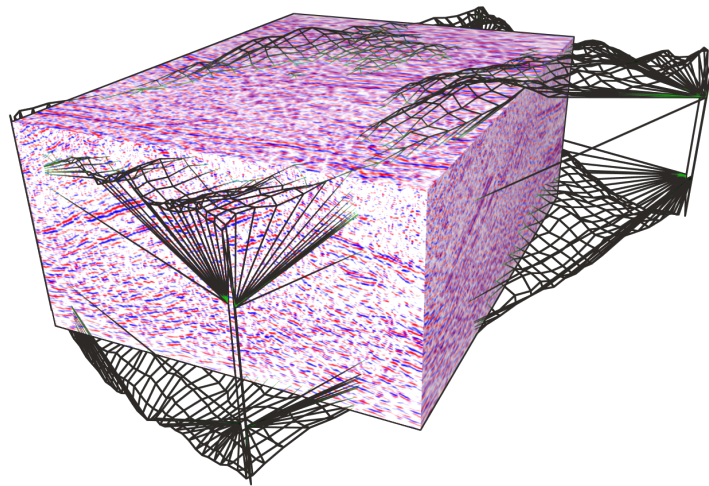


Figure 5.1: Sampling source file shown as seismic cube. Volume of interest shown as wire frame.

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 interpolation, 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

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

The sampler function applied in CRAVA is defined as

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

5.1.1 Implementation

Along the border of the volume some grid cells $s_{i\pm 1,j,k} s_{i,j\pm 1,k} s_{i,j,k\pm 1}$ 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 specified by the outside mode.

Border :

If $s_{i\pm 1,j,k}$, $s_{i,j\pm 1,k}$ or $s_{i,j,k\pm 1}$ 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: (ni, nj, nk)
  do
    (x, y, t) ← getCordinates(i, j, k)
    dobs ← dobs ∪ 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
     $(x, y, t) \leftarrow$  getCoordinates( $i, j, k$ )
    outputGrid  $\leftarrow$  outputGrid  $\cup$  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 calculates 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.

<p>Data: E_t time dependant error Fourier transformed.</p> <p>Data: t time value in range of valid time values T Fourier transformed.</p> <p>Result: D_{obs} observed sample points d_{obs} Fourier transformed.</p> <pre> begin $D_{obs} \leftarrow \emptyset$ for t Input: T do $E_t \leftarrow \text{calculateError}(t)$ parallel for $D \leftarrow$ Input: D_{obs} sharing E_t do Perform calculation with $E_{f,g}$ and D end parallel for end end </pre>

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

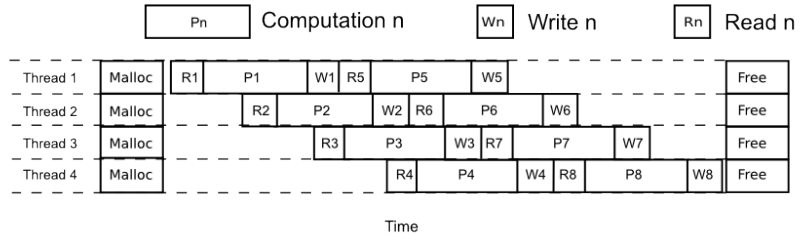


Figure 5.2: Computation pipelining: R_n : Read block n , C_n compute block n , W_n : write block n .

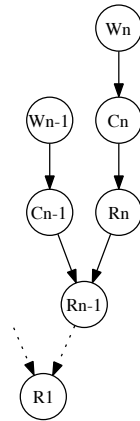


Figure 5.3: Dependency graph for seismic inversion sub tasks.

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.

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

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. `threadprivate` 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.

```

1  int wCnt = 0;
2  /* specifying a round robin scheduler is needed to
3   * ensure that each thread "gives away" access to the next
4   * thread by increasing wCnt.
5   */
6  #pragma omp parallel for schedule(static, 1)
7  for (int i = 0; i < nxp_ * nyp_; i++){
8     /* ordered read and computations */
9
10    while(wCnt != traceNr); /* infinite loop waits for wCnt */
11    /* ordered write */
12    wCnt = wCnt + 1
13 }

```

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.

```
1 /* 1D Fourier transform. The alternative is
2  * 3D Fourier transform.
3  */
4 fftwnd_plan settings;
5 settings = fftw_create_plan(1, width, FFTW_FORWARD, FFTW_ESTIMATE);
6
7 /* One of many Fourier transforms with the settings just
8  * created.
9  */
10 fftw_one_real_to_complex(settings, real_data, complex_data);
11
12 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 safe¹. For a pseudo code of implementation see 4.

¹A 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 \leftarrow find(storage, *s*)
 if *result* = \emptyset **then**
 lock(writeMutex)
 result \leftarrow find(storage, *s*)
 if *result* = \emptyset **then**
 result \leftarrow 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 \leftarrow \emptyset
 while *result* = \emptyset **do**
 result \leftarrow lookupSettings(*s*)
 end
 lock(*result*)
 c \leftarrow FFTW_complexToReal(*result*, *r*)
 unlock(*result*)
end

Data: *FFTSettings* *s*, real values *r*
Result: The complex values *c*

Fouriertransform begin
 result \leftarrow \emptyset
 while *result* = \emptyset **do**
 result \leftarrow lookupSettings(*s*)
 end
 lock(*result*)
 r \leftarrow 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 refactoring 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

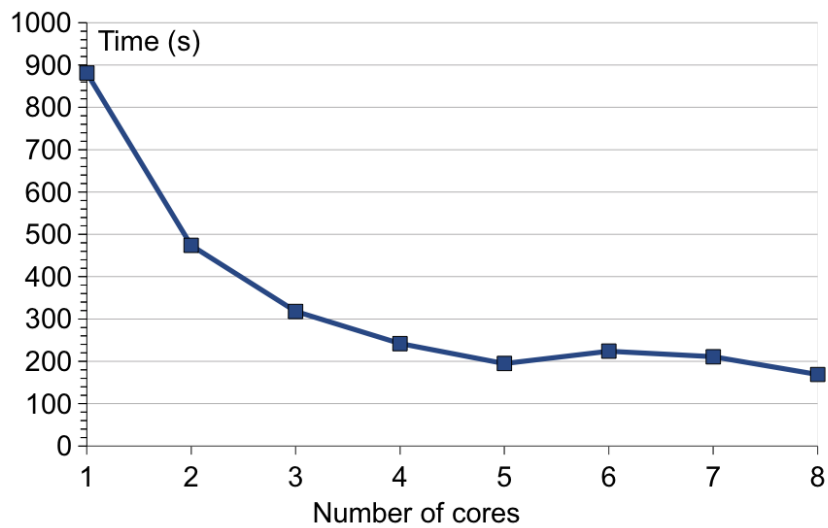


Figure 6.1: Resampling timings on 6.1.1.

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

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: Resampling timings and parallel fractions on 6.1.1

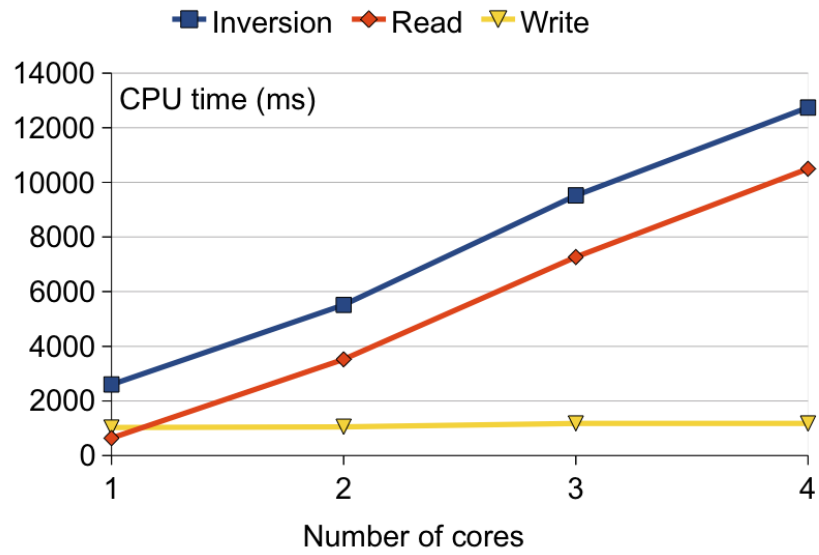


Figure 6.2: Parallel Seismic Inversion CPU time without error calculation.

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 6.1.1¹. Since the speedup model we introduced 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 consistent 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

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

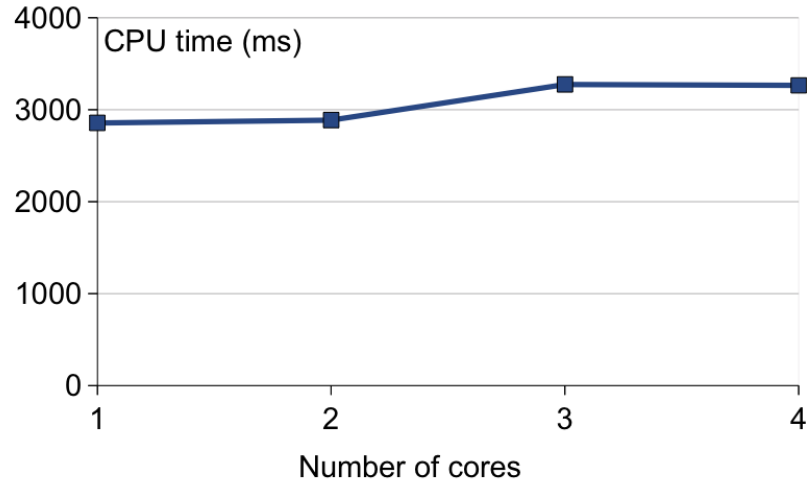


Figure 6.3: Parallel Seismic Inversion wall time.

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 Hat 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.

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

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.

CHAPTER 6. RESULTS AND FUTURE WORK

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 therefore. 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 potential for improvement, but is easily destroyed by a bad micro level code. Low level optimisations are easy to implement, 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 already 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 descending level of granularity. As the the granularity grows one have to do qualified choices of what and how to profile. Alongside profiling improvements should be done to the current bottleneck 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 practice parallel programming enforces 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 improvements 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 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 N th iteration performs reads and writes. If serial reads and writes are performed every N th 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.

APPENDIX A. TESTS

Name		two cubes four wells pred nokrig
Estimation		No
Grid	dimen-	112 * 112 * 280
sions(x*y*z)		
Kriging		No
Seismic angles		16°, 28°
Number of Wavelets		0
Number of Wells		4
Name		two cubes four wells pred krig
Estimation		No
Grid	dimen-	112 * 112 * 250
sions(x*y*z)		
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	dimen-	112 * 112 * 375
sions(x*y*z)		
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	dimen-	15 * 15 * 200
sions(x*y*z)		
Kriging		No
Seismic angles		16°, 28°
Number of Wavelets		2
Number of Wells		4
Name		two cubes one well faciesprob
Estimation		No
Grid	dimen-	15 * 15 * 500
sions(x*y*z)		
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

```
1 #ifndef FFTGRID_H
2 #define FFTGRID_H
3
4 #include <assert.h>
5 #include <string>
6
7 #include "fft/include/fftw.h"
8 #include "definitions.h"
9
10 class Corr;
11 class Wavelet;
12 class Simbox;
13 class RandomGen;
14 class GridMapping;
15
16 class FFTGrid{
17 public:
18
19     FFTGrid(int nx, int ny, int nz, int npx, int nyp, int nzp);
20     FFTGrid(FFTGrid * fftGrid);
21     FFTGrid() {} //Dummy constructor needed for FFTFileGrid
22     virtual ~FFTGrid();
23
24     void setType(int cubeType) {cubetype_ = cubeType;}
25     void setAngle(float angle) {theta_ = angle;}
26
27     int fillInFromSegY(SegY * segy, Simbox *simbox,
28         bool nopadding = false ); // No mode
29
30     int fillInFromStorm(Simbox *
31         actSimBox, StormContGrid * grid,
32         const std::string & parName,
33         bool scale = false,
34         bool nopadding = false);
```

```

34 void                                     // No mode
    fillInConstant(float value); //
35     No mode
    fftw_real* fillInParameter(Corr* corr, int minIntFq,
36                               float gradI, float gradJ); //
37     No mode
    void fillInErrCorr(Corr* parCorr, //
38                               float gradI, float gradJ); //
39     No mode
    virtual void fillInComplexNoise(RandomGen * ranGen); //
40     No mode/randomaccess
41 void fillInTest(float value1, float value2); //
42     No mode /DEBUG
    void fillInFromArray(float *value);
43
44 virtual fftw_complex getNextComplex(); //
45     Accessmode read/readandwrite
    virtual float getNextReal(); //
46     Accessmode read/readandwrite
    virtual int getNextComplex(fftw_complex); //
47     Accessmode write/readandwrite
    virtual int getNextReal(float); //
48     Accessmode write/readandwrite
    float getRealValue(int i, int j, int k, bool
49     extSimbox = false); // Accessmode randomaccess
50     float getRealValueCyclic(int i, int j, int k);
51     float getRealValueInterpolated(int i, int j, float
52     kindex, bool extSimbox = false);
53     fftw_complex getComplexValue(int i, int j, int k, bool
54     extSimbox = false) const;
55     virtual int setRealValue(int i, int j, int k, float
56     value, bool extSimbox = false); // Accessmode randomaccess
57     int setComplexValue(int i, int j, int k,
58     fftw_complex value, bool extSimbox = false);
59     fftw_complex getFirstComplexValue();
60     virtual int square(); //
61     No mode/randomaccess
62     virtual int expTransf(); //
63     No mode/randomaccess
64     virtual int logTransf(); //
65     No mode/randomaccess
66     virtual void realAbs();
67     virtual int collapseAndAdd(float* grid); //
68     No mode/randomaccess
69     virtual void fftInPlace(); //
70     No mode/randomaccess
71     virtual void invFFTInPlace(); //
72     No mode/randomaccess
73
74     virtual void add(FFTGrid* fftGrid); //
75     No mode/randomaccess
76     virtual void subtract(FFTGrid* fftGrid);
77     // No mode/randomaccess
78     virtual void changeSign(); // No mode/
79     randomaccess
80     virtual void multiply(FFTGrid* fftGrid); //
81     pointwise multiplication!
82     bool consistentSize(int nx, int ny, int nz, int
83     nxp, int nyp, int nzp);

```

APPENDIX B. CODE

```

69  int      getCounterForGet() const {return (
counterForGet_);}
70  int      getCounterForSet() const {return (
counterForSet_);}
71  int      getNx()      const {return (nx_);}
72  int      getNy()      const {return (ny_);}
73  int      getNz()      const {return (nz_);}
74  int      getNxp()     const {return (nxp_);}
75  int      getNyp()     const {return (nyp_);}
76  int      getNzp()     const {return (nzp_);}
77  int      getRNxp()    const {return (rnxp_);}
78  int      getcsize()   const {return (csize_);}
79  int      getrszize()  const {return (rszize_);}
80  float    getTheta()   const {return (theta_);}
81  float    getScale()   const {return (scale_);}
82  bool     getIsTransformed() const {return (
istransformed_);}
83  enum     gridTypes{CTMISSING,DATA,PARAMETER,
COVARIANCE, VELOCITY};
84  enum     accessMode{NONE,READ,WRITE,READANDWRITE,
RANDOMACCESS};
85  virtual void multiplyByScalar(float scalar); //No
mode/randomaccess
86  int      getType()    const {return (cubetype_);}
87  virtual void setAccessMode(int mode){assert (mode>=0);}
88  virtual void endAccess(){}
89  virtual void writeFile(const std::string & fileName ,
90                      const std::string & subDir ,
91                      const Simbox * simbox ,
92                      const std::string sgrilabel = "
NO_LABEL" ,
93                      const float z0 =
0.0 ,
94                      GridMapping * depthMap =
NULL ,
95                      GridMapping * timeMap =
NULL ,
96                      const TraceHeaderFormat & thf =
TraceHeaderFormat(
TraceHeaderFormat::SEISWORKS))
;
97  //Use this instead of the ones below.
98  virtual void writeStormFile(const std::string & fileName ,
99                      const Simbox * simbox , bool expTrans = false ,
bool ascii = false , bool
padding = false , bool
flat = false); //No mode/
randomaccess
100 virtual int writeSegyFile(const std::string & fileName ,
const Simbox * simbox , float z0 ,
101                      const TraceHeaderFormat &thf =
TraceHeaderFormat(
TraceHeaderFormat::
SEISWORKS)); //No mode/
randomaccess
102 virtual int writeSgrifile(const std::string & fileName ,
const Simbox * simbox , const std::string label);
103 virtual void writeAsciiFile(const std::string & fileName)
;
104 virtual void writeAsciiRaw(const std::string & fileName);
105 virtual void writeResampledStormCube(GridMapping *
gridmapping , const std::string & fileName ,

```

```

106                                     const Simbox *simbox
                                     , const int
                                     format, bool
                                     expTrans);
107 virtual void writeCravaFile(const std::string & fileName,
    const Simbox * simbox);
108 virtual void readCravaFile(const std::string & fileName,
    std::string & errText, bool nopadding = false);
109
110 virtual bool isFile() {return(0);} // indicates wether
    the grid is in memory or on disk
111
112 static void setOutputFlags(int format, int domain) {
    formatFlag_ = format; domainFlag_ = domain;};
113 static void setOutputFormat(int format) {formatFlag_ =
    format;};
114 int getOutputFormat() {return(formatFlag_);}
115 static void setOutputDomain(int domain) {domainFlag_ =
    domain;};
116 int getOutputDomain() {return(domainFlag_);}
117 static void setMaxAllowedGrids(int maxAllowedGrids) {
    maxAllowedGrids_ = maxAllowedGrids ;}
118 static int getMaxAllowedGrids() { return
    maxAllowedGrids_ ;}
119 static int getMaxAllocatedGrids() { return
    maxAllocatedGrids_ ;}
120 static void setTerminateOnMaxGrid(bool terminate) {
    terminateOnMaxGrid_ = terminate ;}
121 static int findClosestFactorableNumber(int leastint);
122
123
124 virtual void createRealGrid(bool add = true);
125 virtual void createComplexGrid();
126
127 //This function interpolates seismic in all missing traces inside
    area, and mirrors to padding.
128 //Also interpolates in traces where energy is lower than treshold
    .
129 virtual void interpolateSeismic(float energyTreshold = 0)
    ;
130
131 void checkNaN(); //NBNB Ragnar: For debug purpose
    . Negative number = OK.
132 float getDistToBoundary(int i, int n, int np);
133 virtual void getRealTrace(float * value, int i, int j);
134 virtual int setRealTrace(int i, int j, float *value);
135 std::vector<float> getRealTrace2(int i, int j);
136
137
138 static void reportFFTMemoryAndWait(const std::string &
    msg) {
139     LogKit::LogFormatted(LogKit::HIGH, "%s: %2
        d grids, %10.2f MB\n", msg.c_str(),
        nGrids_, FFTMemUse_/(1024.0f*1024.0f))
        ;
140     float tmp;
141     std::cin >> tmp;
142     LogKit::LogFormatted(LogKit::HIGH, "Memory
        used %4.0f MB, used outside grid %4.0
        f MB\n", tmp, tmp-FFTMemUse_/(1024.0f
        *1024.0f));
143 }

```

APPENDIX B. CODE

```

144
145 protected:
146     //int          setPaddingSize(int n, float p);
147     int           getFillNumber(int i, int n, int np );
148
149     int           getXSimboxIndex(int i){return(getFillNumber(
150         i, nx_, nxp_ ));}
151     int           getYSimboxIndex(int j){return(getFillNumber(
152         j, ny_, nyp_ ));}
153     int           getZSimboxIndex(int k);
154     void          computeCircCorrT(Corr* corr, fftw_real*
155         CircCorrT);
156     void          makeCircCorrTPosDef(fftw_real* CircularCorrT
157         ,int minIntFq);
158     fftw_complex* fft1DzInPlace(fftw_real* in);
159     fftw_real*    invFFT1DzInPlace(fftw_complex* in);
160
161     //Interpolation into SegY and sgrid
162     float         getRegularZInterpolatedRealValue(int i, int
163         j, double z0Reg,
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```



```

179
180     int                cnxp_ ;                // size in x direction
        for storage inplace algorithm (complex grid) nxp_/2+1
181     int                rnxp_ ;                // expansion in x
        direction for storage inplace algorithm (real grid) 2*(nxp_
        /2+1)
182
183     int                csize_ ;                // size of complex grid ,
        cnxp_*nyp_*nzp_
184     int                rsize_ ;                // size of real grid
        rnxp_*nyp_*nzp_
185     int                counterForGet_ ;        // active cell in grid
186     int                counterForSet_ ;        // active cell in grid
187
188     bool                istransformed_ ;        // true if the grid
        contain Fourier values (i.e complex variables)
189
190     fftw_complex        * cvalue_ ;            // values of complex
        parameter in grid points
191     fftw_real           * rvalue_ ;            // values of real
        parameter in grid points
192
193     static int          formatFlag_ ;          // Decides format of
        output (see ModelSettings).
194     static int          domainFlag_ ;          // Decides domain of
        output (see ModelSettings).
195
196     static int          maxAllowedGrids_ ;     // The maximum number of
        grids we are allowed to allocate.
197     static int          maxAllocatedGrids_ ;   // The maximum number of
        grids that has actually been allocated.
198     static int          nGrids_ ;              // The actually number
        of grids allocated (varies as crava runs).
199     static bool         terminateOnMaxGrid_ ; // If true , terminate
        when we try to allocate more than maxAllowedGrids.
200     bool                add_ ;                 // Tells whether we
        should change nGrids_ or not
201
202     static float        maxFFTMemUse_ ;
203     static float        FFTMemUse_ ;
204
205 };
206 #endif

```

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

```

1  #ifndef FFTGRID_H
2  #define FFTGRID_H
3
4  #include "lib/timekit.hpp"
5  #include "src/timings.h"
6  #include "src/simbox.h"
7  #include "fft/include/fftw.h"
8  #include "src/locks.h"
9
10 class Corr ;
11 class Wavelet ;
12 class Simbox ;
13 class RandomGen ;
14 class GridMapping ;
15
16 #define AT2D(a, b, width) ((a)*(width) + (b))

```

APPENDIX B. CODE

```

17 //define PROFILING
18 #ifdef PROFILING
19 #include "src/profiling.h"
20 #endif
21
22 class FFTGrid{
23 public:
24   FFTGrid(int nx, int ny, int nz, int nxp, int nyp, int nzp);
25   FFTGrid(FFTGrid * fftGrid);
26   FFTGrid() {} //Dummy constructor needed for FFTFileGrid
27   virtual ~FFTGrid();
28   enum gridTypes{CTMISSING,DATA,PARAMETER,
29                 COVARIANCE,VELOCITY};
30
31   void setType(int cubeType) {cubetype_ = cubeType;}
32   void setAngle(float angle) {theta_ = angle;}
33
34   int fillInFromSegY(SegY * segy, Simbox *simbox,
35                     bool nopadding = false ); // No mode
36
37   int fillInFromStorm(Simbox *
38                       actSimBox,
39                       StormContGrid * grid,
40                       const std::string & parName,
41                       bool scale =
42                         false ,
43                       bool
44                         nopadding = false); // No mode
45
46   void fillInConstant(float value); //
47   // No mode
48   fftw_real* fillInParamCorr(Corr* corr,int minIntFq,
49                             float gradI, float gradJ);//
50   // No mode
51   void fillInErrCorr(Corr* parCorr, //
52                     float gradI, float gradJ); //
53   // No mode
54   virtual void fillInComplexNoise(RandomGen * ranGen); //
55   // No mode/randomaccess
56
57   void fillInTest(float value1, float value2); //
58   // No mode /DEBUG
59   void fillInFromArray(float *value);
60
61   virtual fftw_complex getNextComplex() ; //
62   // Accessmode read/readandwrite
63   virtual float getNextReal() ; //
64   // Accessmode read/readandwrite
65   virtual int setNextComplex(fftw_complex); //
66   // Accessmode write/readandwrite
67   virtual int setNextReal(float); //
68   // Accessmode write/readandwrite
69   float getRealValue(int i, int j, int k, bool
70                     extSimbox = false); // Accessmode randomaccess
71   float getRealValueCyclic(int i, int j, int k);
72   float getRealValueInterpolated(int i, int j, float
73                                   kindex, bool extSimbox = false);
74   virtual fftw_complex getComplexValue(int i, int j, int k, bool
75                                       extSimbox = false);
76   virtual int setRealValue(int i, int j, int k, float
77                          value, bool extSimbox = false); // Accessmode randomaccess

```

```

59 | virtual int          setRealValueOrPad(int i, int j, int k, float
      |         value);
60 | virtual int          setComplexValue(int i, int j, int k,
      |         fftw_complex value, bool extSimbox = false);
61 | fftw_complex         getFirstComplexValue();
62 | virtual int          square(); //
      |         No mode/randomaccess
63 | virtual int          expTransf(); //
      |         No mode/randomaccess
64 | virtual int          logTransf(); //
      |         No mode/randomaccess
65 | virtual void         realAbs();
66 | virtual int          collapseAndAdd(float* grid); //
      |         No mode/randomaccess
67 | virtual void         fftInPlace(); //
      |         No mode/randomaccess
68 | virtual void         invFFTInPlace(); //
      |         No mode/randomaccess
69 |
70 |
71 | virtual void         add(FFTGrid* fftGrid); //
      |         No mode/randomaccess
72 | virtual void         subtract(FFTGrid* fftGrid);
      |         // No mode/randomaccess
73 | virtual void         changeSign(); // No mode/
      |         randomaccess
74 | virtual void         multiply(FFTGrid* fftGrid); //
      |         pointwise multiplication!
75 | bool                consistentSize(int nx, int ny, int nz, int
      |         nxp, int nyp, int nzp);
76 | int                  getCounterForGet() const {return(
      |         counterForGet_);}
77 | int                  getCounterForSet() const {return(
      |         counterForSet_);}
78 | int                  getNx() const {return(nx_);}
79 | int                  getNy() const {return(ny_);}
80 | int                  getNz() const {return(nz_);}
81 | int                  getNxp() const {return(nxp_);}
82 | int                  getNyp() const {return(nyp_);}
83 | int                  getNzp() const {return(nzp_);}
84 | int                  getRNxp() const {return(rnxp_);}
85 | int                  getcsize() const {return(csize_);}
86 | int                  getsrsize() const {return(rsize_);}
87 | float                getTheta() const {return(theta_);}
88 | float                getScale() const {return(scale_);}
89 | bool                 getIsTransformed() const {return(
      |         istransformed_);}
90 | enum                 accessMode{NONE, READ, WRITE, READANDWRITE,
      |         RANDOMACCESS};
91 | virtual void         multiplyByScalar(float scalar); //No
      |         mode/randomaccess
92 | int                  getType() const {return(cubetype_);}
93 | virtual void         setAccessMode(int mode);
94 | virtual void         endAccess(){}
95 | virtual void         writeFile(const std::string & fileName,
      |         const std::string & subDir,
96 |         const Simbox * simbox,
97 |         const std::string sgridLabel = "
      |         NO_LABEL",
98 |         const float z0 =
99 |         0.0,

```

APPENDIX B. CODE

```

100         GridMapping      * depthMap =
101             NULL,
102         GridMapping      * timeMap =
103             NULL,
104         const TraceHeaderFormat & thf =
105             TraceHeaderFormat(
106                 TraceHeaderFormat::SEISWORKS)
107             ;
108 //Use this instead of the ones below.
109 virtual void writeStormFile(const std::string & fileName,
110     const Simbox * simbox, bool expTrans = false,
111     bool ascii = false, bool
112     padding = false, bool
113     flat = false); //No mode/
114     randomaccess
115 virtual int writeSegyFile(const std::string & fileName,
116     const Simbox * simbox, float z0,
117     const TraceHeaderFormat &thf =
118         TraceHeaderFormat(
119             TraceHeaderFormat::
120             SEISWORKS)); //No mode/
121     randomaccess
122 virtual int writeSgriFile(const std::string & fileName,
123     const Simbox * simbox, const std::string label);
124 virtual void writeAsciiFile(const std::string & fileName)
125 ;
126 virtual void writeAsciiRaw(const std::string & fileName);
127 virtual void writeResampledStormCube(GridMapping *
128     gridmapping, const std::string & fileName,
129     const Simbox *simbox
130     , const int
131     format, bool
132     expTrans);
133 virtual void writeCravaFile(const std::string & fileName,
134     const Simbox * simbox);
135 virtual void readCravaFile(const std::string & fileName,
136     std::string & errText, bool nopadding = false);
137
138 virtual bool isFile() {return(0);} // indicates wether
139     the grid is in memory or on disk
140
141 static void setOutputFlags(int format, int domain) {
142     formatFlag_ = format; domainFlag_ = domain;};
143 static void setOutputFormat(int format) {formatFlag_ =
144     format;};
145 int getOutputFormat() {return(formatFlag_);}
146 static void setOutputDomain(int domain) {domainFlag_ =
147     domain;};
148 int getOutputDomain() {return(domainFlag_);}
149 static void setMaxAllowedGrids(int maxAllowedGrids) {
150     maxAllowedGrids_ = maxAllowedGrids ;};
151 static int getMaxAllowedGrids() { return
152     maxAllowedGrids_
153     ;};
154 static int getMaxAllocatedGrids() { return
155     maxAllocatedGrids_
156     ;};
157 static void setTerminateOnMaxGrid(bool terminate) {
158     terminateOnMaxGrid_ = terminate ;};
159 static int findClosestFactorableNumber(int leastint);
160
161 virtual void createRealGrid(bool add = true);
162 virtual void createComplexGrid();

```

```

132
133 //This function interpolates seismic in all missing traces inside
134 //Also interpolates in traces where energy is lower than threshold
135 virtual void interpolateSeismic(float energyTreshold = 0)
136 ;
137 void checkNaN(); //NBNB Ragnar: For debug purpose
138 // Negative number = OK.
139 float getDistToBoundary(int i, int n, int np);
140 virtual void getRealTrace(float * value, int i, int j);
141 virtual int setRealTrace(int i, int j, float *value);
142 std::vector<float> getRealTrace2(int i, int j);
143
144 static void reportFFTMemoryAndWait(const std::string &
145 msg) {
146     LogKit::LogFormatted(LogKit::HIGH, "%s: %2
147     d grids, %10.2f MB\n", msg.c_str(),
148     nGrids_, FFTMemUse_/(1024.0f*1024.0f))
149     ;
150     float tmp;
151     std::cin >> tmp;
152     LogKit::LogFormatted(LogKit::HIGH, "Memory
153     used %4.0f MB, used outside grid %4.0
154     f MB\n", tmp, tmp-FFTMemUse_/(1024.0f
155     *1024.0f));
156 }
157 bool GetAllValidIndices(const Simbox& simbox,
158 const SegY& segy, int*& lowerTop, int*& topBorder, int*&
159 botBorder, int*& upperBot) const;
160 virtual bool allowsRandomWrite() const;
161 virtual bool allowsRandomRead() const;
162
163 protected:
164 float getValueByIndex(const Simbox* simbox, const
165 SegY* segy, int i, int j, int k) const;
166
167 template <NRLib::OutsideMode borderCases>
168 int sampleSEGY(SegY * segy, Simbox *simbox, bool
169 nopadding = false ); // No mode
170 //int setPaddingSize(int n, float p);
171 static int getFillNumber(int i, int n, int np );
172 static int getXSimboxIndex(int i, int nx, int npx){
173     return(FFTGrid::getFillNumber(i, nx, npx));}
174 static int getYSimboxIndex(int j, int ny, int nyp){
175     return(FFTGrid::getFillNumber(j, ny, nyp));}
176 int getXSImboxIndex(int i) const{return(
177     getFillNumber(i, nx_, npx_));}
178 int getYSImboxIndex(int j) const{return(
179     getFillNumber(j, ny_, nyp_));}
180 int getZSimboxIndex(int k) const;
181 void computeCircCorrT(Corr* corr, fftw_real*
182     CircCorrT);
183 void makeCircCorrTPosDef(fftw_real* CircularCorrT
184     ,int minIntFq);
185 fftw_complex* fft1DzInPlace(fftw_real* in);
186 fftw_real* invFFT1DzInPlace(fftw_complex* in);
187
188 //Interpolation into SegY and sgri

```

APPENDIX B. CODE

```

172 float          getRegularZInterpolatedRealValue(int i, int
173           j, double z0Reg,
174                                     double
175                                     dzReg,
176                                     int
177                                     kReg,
178                                     double
179                                     z0Grid
180                                     ,
181                                     double
182                                     dzGrid
183                                     );
184
185 //Supporting functions for interpolateSeismic
186 int          interpolateTrace(int index, short int *
187   flags, int i, int j);
188 void        extrapolateSeismic(int imin, int imax, int
189   jmin, int jmax);
190
191 /// Called from writeResampledStormCube
192 void        writeSegyFromStorm(StormContGrid *data, std
193   ::string fileName);
194 void        makeDepthCubeForSegy(Simbox *simbox, const
195   std::string & fileName);
196 int         cubetype_;           // see enum gridtypes
197   above
198 float       theta_;             // angle in angle gather
199   (case of data)
200 float       scale_;            // To keep track of the
201   scalings after fourier transforms
202 int         nx_;               // size of original grid
203   in lateral x direction
204 int         ny_;               // size of original grid
205   in lateral y direction
206 int         nz_;               // size of original grid
207   in depth (time)
208 int         nxp_;              // size of padded FFT
209   grid in lateral x direction
210 int         nyp_;              // size of padded FFT
211   grid in lateral y direction
212 int         nzp_;              // size of padded FFT
213   grid in depth (time)
214 int         cnxp_;             // size in x direction
215   for storage inplace algorithm (complex grid) nxp_/2+1
216 int         rnxp_;            // expansion in x
217   direction for storage inplace algorithm (real grid) 2*(nxp_
218   /2+1)
219 int         csize_;           // size of complex grid,
220   cnxp_*nyp_*nzp_
221 int         rsize_;           // size of real grid
222   rnxp_*nyp_*nzp_
223 int         counterForGet_;    // active cell in grid
224 int         counterForSet_;    // active cell in grid
225 bool        istransformed_;    // true if the grid
226   contain Fourier values (i.e complex variables)
227
228 fftw_complex * cvalue_;       // values of complex
229   parameter in grid points

```

```

204   fftw_real          * rvalue_;           // values of real
      parameter in grid points
205
206   static int         formatFlag_;         // Decides format of
      output (see ModelSettings).
207   static int         domainFlag_;        // Decides domain of
      output (see ModelSettings).
208
209   static int         maxAllowedGrids_;    // The maximum number of
      grids we are allowed to allocate.
210   static int         maxAllocatedGrids_; // The maximum number of
      grids that has actually been allocated.
211   static int         nGrids_;            // The actually number
      of grids allocated (varies as crava runs).
212   static bool        terminateOnMaxGrid_; // If true, terminate
      when we try to allocate more than maxAllowedGrids.
213   bool               add_;               // Tells whether we
      should change nGrids_ or not
214
215   static float       maxFFTMemUse_;
216   static float       FFTMemUse_;
217
218 };
219
220 #define SETVAL(i, j, k, value) rvalue_[(i) + rnxp*(j) + (k)*
      rnxp_nyp] = value
221
222 template <NRLib::OutsideMode borderCases>
223 int FFTGrid::sampleSEGY(SegY* segy, Simbox *simbox, bool padding){
224   const int nx = nx_;
225   const int ny = ny_;
226   const int nz = nz_;
227   const int nxp = nxp_;
228   const int nyp = nyp_;
229   const int nzp = nzp_;
230   const int rnxp = rnxp_;
231   const int rnxp_nyp = rnxp*nyp;
232   double x = 0.0, y = 0.0, z = 0.0;
233   float distx = 0.0f, disty = 0.0f, distz = 0.0f, value = 0.0f,
      val1 = 0.0f, val2 = 0.0f;
234   int outsideTraces = 0;
235   int refi = 0, refj = 0, refk = 0, i, j, k;
236   segy->setOutsideMode(borderCases);
237   createRealGrid(!padding);
238   add_ = !padding;
239   float* meanvalue= new float [nyp*nxp];
240
241   if(borderCases == NRLib::ZERO){
242   // #pragma omp parallel for private(i, j, x, y, z, val1, val2)
      default(shared)
243     for(j=0; j<ny; j++){
244       for(i=0; i<nx; i++){
245         simbox->getCoord(getXSimboxIndex(i), getYSimboxIndex(j), 0,
            x, y, z);
246         val1 = segy->GetValue(x,y,z);
247         simbox->getCoord(getXSimboxIndex(i), getYSimboxIndex(j), nz
            -1, x, y, z);
248         val2 = segy->GetValue(x,y,z);
249         if(((val1 == 0 && val2 == 0) || (val1 == RMISSING && val2
            == RMISSING)) &&
250           segy->GetGeometry()->IsInside(static_cast<float>(x),
            static_cast<float>(y)) == false){

```

APPENDIX B. CODE

```

251 // #pragma omp atomic
252     outsideTraces += 1;
253     }
254     }
255     }
256     }
257     if (borderCases == NRLib::CLOSEST){
258 #pragma omp parallel shared(simbox, segy, meanvalue) private(j, x,
259     y, z, val1, val2)
260     {
261 #if _OPENMP >= 200805
262 #pragma omp for collapse(2)
263 #else
264 #pragma omp for
265 #endif
266     for (j=0; j<nyp; j++){
267         for (i=0; i<nxp; i++){
268             if (i >= nx && j >= ny){
269                 simbox->getCoord(getXSimboxIndex(i, nx, nyp),
270                 getYSimboxIndex(j, ny, nyp), 0, x, y, z);
271                 val1 = segy->GetValue(x,y,z);
272                 simbox->getCoord(getXSimboxIndex(i, nx, nyp),
273                 getYSimboxIndex(j, ny, nyp), nz-1, x, y, z);
274                 val2 = segy->GetValue(x,y,z);
275                 meanvalue[AT2D(j, i, nyp)] = (val1+val2)/2.0f;
276             }
277         }
278     }
279     }
280     }
281     double wall=0.0, cpu=0.0;
282     LogKit::LogFormatted(LogKit::LOW, "\nResampling seismic data into
283     %dx%dx%d grid: ", nxp, nyp, nzp);
284     int* top = NULL;
285     int* bot = NULL;
286     int* lowertop = NULL;
287     int* upperbot = NULL;
288     TimeKit::getTime(wall, cpu);
289     setAccessMode(RANDOMACCESS);
290     GetAllValidIndices(*simbox, *segy, lowertop, top, bot, upperbot);
291 #define INDEX AT2D(j, i, nx)
292     for (j = 0; j < ny; j++){
293         for (i = 0; i < nx; i++){
294             float kVal;
295             k = 0;
296             kVal = getValueByIndex(simbox, segy, i, j, k);
297             SETVAL(i, j, k, kVal);
298             k = nz-1;
299             kVal = getValueByIndex(simbox, segy, i, j, k);
300             SETVAL(i, j, k, kVal);
301             k = 1;
302             kVal = getValueByIndex(simbox, segy, i, j, k);
303             for (k = 1; i < lowertop[INDEX]; k++){
304                 SETVAL(i, j, k, kVal);
305             }
306             k = nz-2;
307             kVal = getValueByIndex(simbox, segy, i, j, k);
308             SETVAL(i, j, k, kVal);

```



```

309     for(k = upperbot[INDEX]; k < nz-1; k++){
310         SETVAL(i, j, k, kVal);
311     }
312 }
313 }
314 }
315 #pragma omp parallel private(i, j, k, refi, refj, refk, x, y, z,
    distx, disty, distz, value)
316 {
317 #if _OPENMP >= 200805
318 #pragma omp for collapse(2) nowait
319 #else
320 #pragma omp for nowait
321 #endif
322     for(j = 0; j < ny; j++){
323         for(i = 0; i < nx; i++){
324             for(k = lowertop[INDEX]; k < top[INDEX]; k++){
325                 float kVal = getValueByIndex(simbox, segy, i, j, k);
326                 SETVAL(i, j, k, kVal);
327             }
328             for(k = top[INDEX]; k < bot[INDEX]; k++){
329                 int newK = k;
330                 simbox->getXYCoord(i, j, x, y);
331                 z = simbox->getZCoord(newK, x, y);
332                 float kVal = segy->GetValueInterpolated(i, j, newK, x, y, z
    );
333                 SETVAL(i, j, k, kVal);
334             }
335             for(k = bot[INDEX]; k < upperbot[INDEX]; k++){
336                 float kVal = getValueByIndex(simbox, segy, i, j, k);
337                 SETVAL(i, j, k, kVal);
338             }
339         }
340     }
341 }
342 #if _OPENMP >= 200805
343 #pragma omp for collapse(3) nowait
344 #else
345 #pragma omp for nowait
346 #endif
347     for(j = 0; j < nyp; j++){
348         for(i = nxp; i < rnxp; i++){
349             for(k = 0; k < nzp; k++){
350                 setRealValue(i, j, k, RMISSING, true);
351             }
352         }
353     }
354 }
355 #if _OPENMP >= 200805
356 #pragma omp for collapse(3) nowait
357 #else
358 #pragma omp for nowait
359 #endif
360     for(j = 0; j < nyp; j++){
361         for(i = 0; i < nxp; i++){
362             for(k = 0; k < nzp; k++){
363                 if(i >= nx || j >= ny || k >= nz){
364                     refi = getXSimboxIndex(i);
365                     refj = getYSimboxIndex(j);
366                     refk = getZSimboxIndex(k);
367                     distx = getDistToBoundary(i, nx, nxp);
368                     disty = getDistToBoundary(j, ny, nyp);

```

APPENDIX B. CODE

```

369         distz = getDistToBoundary(k,nz,nzp);
370         float mult = static_cast<float>(pow(std::max<float>
          >(1.0-distx*distx-disty*disty-distz*distz,0.0),3));
371         simbox->getCoord(refi,refj,refk,x,y,z);
372         value = segy->GetValue(x,y,z);
373         if(value != RMISSING){
374             if(borderCases == NRLib::CLOSEST){
375                 value = static_cast<float>(mult*value+(1.0-mult)*
          meanvalue[AT2D(j,i, nxp)]);
376             }else{
377                 value *= mult;
378             }
379         }
380         setRealValue(i,j,k,value,true);
381     }
382 }
383 }
384 }
385 }
386 Timings::setTimeResamplingSeismic(wall,cpu);
387 endAccess();
388 LogKit::LogMessage(LogKit::LOW,"\\n");
389 delete [] bot;
390 delete [] lowertop;
391 delete [] meanvalue;
392 delete [] top;
393 delete [] upperbot;
394 return outsideTraces;
395 }
396
397 #endif

```

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

```

1  /* Unrelated code */
2
3  int
4  FFTGrid::fillInFromSegY(SegY* segy, Simbox *simbox, bool padding)
5  {
6      assert(cubetype_ != CTMISSING);
7
8      createRealGrid(!padding);
9      add_ =!padding;
10     int i,j,k,refi,refj,refk;
11     float distx,disty,distz,mult;
12     double x,y,z;
13     float* meanvalue = NULL;
14     bool isParameter=false;
15
16     if(cubetype_==PARAMETER) isParameter=true;
17
18     int outMode = SegY::MISSING;
19     if(cubetype_ == DATA)
20         outMode = SegY::ZERO;
21     else if(cubetype_ == PARAMETER)
22         outMode = SegY::CLOSEST;
23
24     fftw_real value = 0.0;
25     float val1,val2;
26
27     meanvalue= static_cast<float*>(fftw_malloc(sizeof(float)*nyp_*
          nxp_));

```

```

28
29 int outsideTraces = 0;
30 for (j=0;j<nyp_;j++) {
31     for (i=0;i<nxp_;i++) {
32         refi = getXSimboxIndex(i);
33         refj = getYSimboxIndex(j);
34         refk = 0;
35         simbox->getCoord(refi, refj, refk, x, y, z);
36         val1 = segy->GetValue(x, y, z, outMode);
37         refk = nz_-1;
38         simbox->getCoord(refi, refj, refk, x, y, z);
39         val2 = segy->GetValue(x, y, z, outMode);
40         meanvalue[i+j*nxp_] = static_cast<float>((val1+val2)/2.0);
41         if ((outMode == SegY::ZERO && val1 == 0 && val2 == 0) ||
42             (outMode != SegY::ZERO && val1 == RMISSING && val2 ==
43                 RMISSING)) {
44             if (cubetype_ == DATA || (i < nx_ && j < ny_)) //Count
45                 padding traces only for data.
46                 if (segy->GetGeometry()->IsInside(static_cast<float>(x),
47                     static_cast<float>(y)) == false)
48                     outsideTraces++;
49         }
50     }
51 }
52
53 double wall=0.0, cpu=0.0;
54 TimeKit::getTime(wall, cpu);
55
56 LogKit::LogFormatted(LogKit::LOW, "\nResampling seismic data into
57     %dx%d grid:", nxp_, nyp_, nzp_);
58 setAccessMode(WRITE);
59
60 float monitorSize = std::max(1.0f, static_cast<float>(nyp_*nzp_)
61     *0.02f);
62 float nextMonitor = monitorSize;
63 printf("\n 0%%      20%%      40%%      60%%      80%%
64     100%%");
65 printf("\n |      |      |      |      |      |      |      |
66     ");
67 printf("\n ^");
68
69 for (k = 0; k < nzp_; k++)
70 {
71     for (j = 0; j < nyp_; j++)
72     {
73         for (i = 0; i < nxp_; i++)
74         {
75             if (i < nxp_)
76             {
77                 refi = getXSimboxIndex(i);
78                 refj = getYSimboxIndex(j);
79                 refk = getZSimboxIndex(k);
80                 simbox->getCoord(refi, refj, refk, x, y, z);
81                 distx = getDistToBoundary(i, nx_, nxp_);
82                 disty = getDistToBoundary(j, ny_, nyp_);
83                 distz = getDistToBoundary(k, nz_, nzp_);
84                 mult = static_cast<float>(pow(std::max<double>(1.0-
85                     distx*distx-disty*disty-distz*distz, 0.0), 3));
86                 value = segy->GetValue(x, y, z, outMode);
87                 if (value != RMISSING) {
88                     if (isParameter)

```

APPENDIX B. CODE

```

81         value = static_cast<float>(mult*value+(1.0-mult)*
82             meanvalue[i+j*nxp_]);
83     else
84         value *= mult;
85     }
86     else
87         value=RMISSING;
88
89     setNextReal(value);
90
91     } //for k,j,i
92     if (nyp_*k + j + 1 >= static_cast<int>(nextMonitor))
93     {
94         nextMonitor += monitorSize;
95         printf("^");
96         fflush(stdout);
97     }
98 }
99 }
100 LogKit::LogFormatted(LogKit::LOW, "\n");
101 endAccess();
102 fftw_free(meanvalue);
103
104 Timings::setTimeResamplingSeismic(wall,cpu);
105 return(outsideTraces);
106 }
107
108 int
109 FFTGrid::getFillNumber(int i, int n, int np )
110 {
111     // for the series          i = 0,1,2,3,4,5,6,7
112     // GetFillNumber(i, 5 , 8) returns 0,1,2,3,4,4,1,0 (cut
113     // middle, i.e 3,2)
114     // GetFillNumber(i, 4 , 8) returns 0,1,2,3,3,2,1,0 (copy)
115     // GetFillNumber(i, 3 , 8) returns 0,1,2,2,1,1,1,0 (drag
116     // middle out, i.e. 1)
117
118     int refi = 0;
119     int BeloWnp, AbovEn;
120
121     if (i < np)
122     {
123         if (i < n)
124             // then it is in the main cube
125             refi = i;
126         else
127         {
128             // Get cyclic extention
129             BeloWnp = np-i-1;
130             AbovEn = i-n+1;
131             if( AbovEn < BeloWnp )
132             {
133                 // Then the index is closer to end than start.
134                 refi=std::max(n-AbovEn,n/2);
135             }else{
136                 // The it is closer to start than the end
137                 refi=std::min(BeloWnp,n/2);
138             }//endif
139         }//endif
140     }//endif
141     else

```

```

140 {
141     // This happens when the index is larger than the padding size
142     // this happens in some cases because rnxp_ is larger than nxp_
143     // and the x cycle is of length rnxp_
144     refi=IMISSING;
145 }//endif
146 return(refi);
147 }
148
149 int
150 FFTGrid::getZSimboxIndex(int k)
151 {
152     int refk;
153
154     if(k < (nz_+nzp_)/2)
155         refk=k;
156     else
157         refk=k-nzp_;
158
159     return refk;
160 }
161
162
163 float
164 FFTGrid::getDistToBoundary(int i, int n, int np )
165 {
166     // for the series i = 0,1,2,3,4,5,6,7
167     // GetFillNumber(i, 5 , 8) returns 0,0,0,0,0,p,r,p p is
168     // between 0 and 1, r is larger than 1
169     // GetFillNumber(i, 4 , 8) returns 0,0,0,0,p,r,r,p p is
170     // between 0 and 1, r's are larger than 1
171     // GetFillNumber(i, 3 , 8) returns 0,0,0,p,r,r,r,p p is
172     // between 0 and 1, r's are larger than 1
173
174     float dist = 0.0;
175     float taperlength = 0.0;
176     int BeloWnp, AbovEn;
177
178     if (i < np)
179     {
180         if (i < n)
181             // then it is in the main cube
182             dist = 0.0;
183         else
184         {
185             taperlength = static_cast<float>((std::min(n,np-n)/2.1)) ;//
186             // taper goes to zero at taperlength
187             BeloWnp = np-i;
188             AbovEn = i-(n-1);
189             if( AbovEn < BeloWnp )
190             {
191                 // Then the index is closer to end than start.
192                 dist = static_cast<float>(AbovEn/taperlength);
193             }
194             else
195             {
196                 // The it is closer to start than the end (or identical to
197                 // )
198                 dist = static_cast<float>(BeloWnp/taperlength);
199             }
200         }
201     }
202 }//endif
203 }//endif
204 }//endif

```

APPENDIX B. CODE

```
197     else
198     {
199         // This happens when the index is larger than the padding size
200         // this happens in some cases because rnxp_ is larger than nxp_
201         // and the x cycle is of length rnxp_
202         dist=RMISSING;
203     }//endif
204     return(dist);
205 }
206
207 /* Unrelated code */
```

Listing B.3: Old volume resampling functions.

```
1  #include <assert.h>
2  #include "src/fftgrid.h"
3  #include "lib/random.h"
4  #include "nrlib/iotools/fileio.hpp"
5  #include "src/corr.h"
6  #include "src/fftwlock.h"
7  #include "src/gridmapping.h"
8  #include "src/omp_op.h"
9  #include "src/tasklist.h"
10 #include "src/io.h"
11 #include <string>
12
13 using NRLib::OutsideMode;
14
15 #define AT2D(a, b, width) ((a)*(width) + (b))
16 // #define PROFILING
17 #ifdef PROFILING
18 #include "src/profiling.h"
19 #include "nrlib/iotools/logkit.hpp"
20 #define RESAMPLING 5
21 #endif
22
23 /* Unrelated code */
24
25 int
26 FFTGrid::fillInFromSegY(SegY* segy, Simbox *simbox, bool padding)
27 {
28 #ifdef PROFILING
29     double wall = 0;
30 #pragma omp master
31     wall = omp_get_wtime();
32 #endif
33     assert(cubetype_ != CTMISSING);
34     switch(cubetype_){
35     case DATA:
36         sampleSEGY<NRLib::ZERO>(segy, simbox, padding);
37         break;
38     case PARAMETER:
39         sampleSEGY<NRLib::CLOSEST>(segy, simbox, padding);
40         break;
41     default:
42         sampleSEGY<NRLib::MISSING>(segy, simbox, padding);
43     }
44 #ifdef PROFILING
45     wall = omp_get_wtime() - wall;
46     NRLib::Prof::setName("fillInFromSegY\n", RESAMPLING);
47     NRLib::Prof::trackTime(wall, RESAMPLING);
48 #endif
```

```

49     return 0;
50 }
51
52 /* Unrelated code */
53
54 int
55 FFTGrid::getFillNumber(int i, int n, int np )
56 {
57     // for the series          i = 0,1,2,3,4,5,6,7
58     // GetFillNumber(i, 5 , 8) returns 0,1,2,3,4,4,1,0 (cut
59     // middle, i.e 3,2)
60     // GetFillNumber(i, 4 , 8) returns 0,1,2,3,3,2,1,0 (copy)
61     // GetFillNumber(i, 3 , 8) returns 0,1,2,2,1,1,1,0 (drag
62     // middle out, i.e. 1)
63
64     int refi    = i;
65     int BeloWnp = np-i-1;
66     int AbovEn  = i-n+1;
67
68     if (i < np)
69     {
70         if (i >= n)
71         {
72             refi = (AbovEn < BeloWnp ? std::max(n-AbovEn,n>>1) : std::min(
73                 BeloWnp,n>>1));
74         } //endif
75     } //endif
76     else
77     {
78         // This happens when the index is larger than the padding size
79         // this happens in some cases because rnxp_ is larger than nxp_
80         // and the x cycle is of length rnxp_
81         refi=IMISSING;
82     } //endif
83     return refi;
84 }
85
86 int
87 FFTGrid::getZSimboxIndex(int k) const
88 {
89     int refk;
90
91     if(k < (nz_+nzp_)/2)
92         refk=k;
93     else
94         refk=k-nzp_;
95
96     return refk;
97 }
98
99 float
100 FFTGrid::getDistToBoundary(int i, int n, int np )
101 {
102     // for the series          i = 0,1,2,3,4,5,6,7
103     // GetFillNumber(i, 5 , 8) returns 0,0,0,0,0,p,r,p p is
104     // between 0 and 1, r is larger than 1
105     // GetFillNumber(i, 4 , 8) returns 0,0,0,0,p,r,r,p p is
106     // between 0 and 1, r's are larger than 1
107     // GetFillNumber(i, 3 , 8) returns 0,0,0,p,r,r,r,p p is
108     // between 0 and 1, r's are larger than 1

```

APPENDIX B. CODE

```

105 float dist      = 0.0;
106 float taperlength = 0.0;
107 int  BeloWnp   = np-i;
108 int  AbovEn    = i-(n-1);
109 if (i < np)
110 {
111     if (i >= n)
112     {
113         taperlength = std::min(n,np-n)/2.1f; // taper goes to zero at
114         taperlength
115         if( AbovEn < BeloWnp )
116         {
117             // Then the index is closer to end than start.
118             dist = static_cast<float>(AbovEn/taperlength);
119         }
120         else
121         {
122             // The it is closer to start than the end (or identical to
123             )
124             dist = static_cast<float>(BeloWnp/taperlength);
125         } //endif
126     } //endif
127 } else {
128     // This happens when the index is larger than the padding size
129     // this happens in some cases because rnxp_ is larger than nxp_
130     // and the x cycle is of length rnxp_
131     dist=RMISSING;
132 } //endif
133 return dist;
134 }
135
136 /* Unreleated code */
137
138 static bool max3x3(const int* srcBorder, int* dstBorder, int nx,
139 int ny){
140     int i, j, index, botzid, convid, convi, convj;
141     const int pSize = nx*ny;
142     botzid = reduceMin(srcBorder, pSize, dstBorder);
143     for(i = 0; i < pSize; i++) dstBorder[i] = botzid;
144     for(i = 0; i < nx; i++){
145         for(j = 0; j < ny; j++){
146             index = AT2D(j, i, nx);
147             for(convi = std::max(i-1, 0); convi < std::min(i+1, nx);
148                 convi++){
149                 for(convj = std::max(j-1, 0); convj < std::min(j+1, ny);
150                     convj++){
151                     if((convi == i && convj != j) || (convj == j && convi !=
152                         i)){
153                         convid = AT2D(convj, convi, nx);
154                         dstBorder[index] = std::max(srcBorder[convid],
155                             dstBorder[index]);
156                     }
157                 }
158             }
159         }
160     }
161     return true;
162 }
163
164 static bool min3x3(const int* srcBorder, int* dstBorder, int nx,
165 int ny){
166     int i, j, index, topzid, convid, convi, convj;

```



```

159     const int pSize = nx*ny;
160     topzid = reduceMax(srcBorder, pSize, dstBorder);
161     for(i = 0; i < pSize; i++) dstBorder[i] = topzid;
162     for(i = 0; i < nx; i++){
163         for(j = 0; j < ny; j++){
164             index = AT2D(j, i, nx);
165             for(convi = std::max(i-1, 0); convi < std::min(i+1, nx);
166                 convi++){
167                 for(convj = std::max(j-1, 0); convj < std::min(j+1, ny);
168                     convj++){
169                     if((convi == i && convj != j) || (convj == j && convi !=
170                         i)){
171                         convid = AT2D(convj, convi, nx);
172                         dstBorder[index] = std::min(srcBorder[convid],
173                             dstBorder[index]);
174                     }
175                 }
176             }
177         }
178     }
179     return true;
180 }
181
182 bool FFTGrid::GetAllValidIndices(const Simbox& simbox, const SegY&
183     segy, int*& lowerTop, int*& topBorder, int*& botBorder, int*&
184     upperBot) const{
185     const int nx = nx_;
186     const int ny = ny_;
187     const int nz = nz_;
188     const int pSize = nx*ny;
189     double x, y, topz, botz, cx, cy;
190     int i, j, xind, yind, index = 0;
191
192     if(botBorder == NULL) botBorder = new int[pSize];
193     if(topBorder == NULL) topBorder = new int[pSize];
194     if(lowerTop == NULL) lowerTop = new int[pSize];
195     if(upperBot == NULL) upperBot = new int[pSize];
196     bool valid = (botBorder != NULL &&
197         topBorder != NULL &&
198         lowerTop != NULL &&
199         upperBot != NULL);
200     if(!valid) return false;
201
202     const NRLib::Surface<double>* bot = &simbox.GetBotSurface();
203     const NRLib::Surface<double>* top = &simbox.GetTopSurface();
204     //Got deadlocks when parallizing this loop. I dont know why.
205     for(i = 0; i < nx; i++){
206         for(j = 0; j < ny; j++){
207             index = AT2D(j, i, nx);
208             simbox.getXYCoord(i, j, x, y);
209             cx = x, cy = y;
210             segy.GetXYID(xind, yind, cx, cy);
211             topz = top->GetZ(x, y);
212             botz = bot->GetZ(x, y);
213             botBorder[index] = simbox.getZIndex(x, y, topz);
214             topBorder[index] = simbox.getZIndex(x, y, botz);
215         }
216     }
217     int* newBotBorder = new int[pSize];
218     int* newTopBorder = new int[pSize];
219     #pragma omp parallel sections

```

APPENDIX B. CODE

```

215     {
216         #pragma omp section
217         min3x3(botBorder, lowerTop, nx, ny);
218         #pragma omp section
219         max3x3(botBorder, newBotBorder, nx, ny);
220         #pragma omp section
221         min3x3(topBorder, newTopBorder, nx, ny);
222         #pragma omp section
223         max3x3(topBorder, upperBot, nx, ny);
224     }
225     for(int i = 0; i < nx; i++){
226         for(int j = 0; j < ny; j++){
227             if(i == 0 || j == 0 || i >= nx || j >= ny){
228                 lowerTop[index] = 1;
229                 newTopBorder[index] = nz-1;
230                 newBotBorder[index] = 1;
231                 upperBot[index] = nz-1;
232             }
233         }
234     }
235     delete topBorder;
236     delete botBorder;
237     topBorder = newTopBorder;
238     botBorder = newBotBorder;
239     return valid;
240 }
241
242 float FFTGrid::getValueByIndex(const Simbox* simbox, const SegY*
243     segy, int i, int j, int k) const{
244     #ifdef PROFILING
245         double wall = omp_get_wtime();
246     #endif
247     double x, y, z;
248     int refk = getZSimboxIndex(k);
249     simbox->getXYCoord(i, j, x, y);
250     z = simbox->getZCoord(refk, x, y);
251     return segy->GetValue(x, y, z);
252     #ifdef PROFILING
253     wall = omp_get_wtime() - wall;
254     NRLib::Prof::trackTime(wall, GETVALUEBYINDEX);
255     #endif
256 }
257 /* Unrelated code */

```

Listing B.4: New volume resampling functions.

```

1  #ifndef SEGY_HPP
2  #define SEGY_HPP
3
4  #include <fstream>
5  #include <string>
6  #include <vector>
7
8  #include "traceheader.hpp"
9  #include "commonheaders.hpp"
10 #include "../volume/volume.hpp"
11 #include "../segy/segygeometry.hpp"
12 #include "../segy/segytrace.hpp"
13
14 namespace NRLib {
15
16 const int segyIMISSING = -99999;
17 class SegYTrace;
18 class SegYGeometry;
19 class BinaryHeader;
20 class TextualHeader;
21
22
23 class SegY{
24 public:
25
26     /// Constructor for reading
27     /// Read only the headers on top of the file
28     /// \param[in] fileName Name of file to read data from
29     /// \param[in] z0
30     /// \throw IOError if the file can not be opened.
31     SegY(const std::string      & fileName ,
32         float                   z0 ,
33         const TraceHeaderFormat & traceHeaderFormat);
34
35
36     /// Constructor for reading unknown format
37     /// Read only the headers on top of the file
38     /// \param[in] fileName Name of file to read data from
39     /// \param[in] z0
40     /// \param[in] thf Vector of pointers to possible
41     /// traceheaderformats. If NULL, default list is used.
42     /// \throw IOError if the file can not be opened.
43     /// \throw FileFormatError if the traceheaderformat can not be
44     /// recognized.
45     SegY(const std::string      & fileName ,
46         float                   z0 ,
47         std::vector<TraceHeaderFormat *> thf = std::vector<
48         TraceHeaderFormat *>(0) ,
49         bool                     searchStandardFormats =
50         true);
51
52     /// Constructor for writing
53     /// \param[in] fileName Name of file to write data to
54     /// \throw IOError if the file can not be opened.
55     SegY(const std::string      & fileName ,
56         float                   z0 ,
57         int                     nz ,
58         float                   dz ,
59         const TextualHeader      & ebcDicHeader ,
60         const TraceHeaderFormat & traceHeaderFormat =
61         TraceHeaderFormat(TraceHeaderFormat::SEISWORKS));
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

APPENDIX B. CODE

```

58 ~SegY();
59
60 //>>>Begin read all traces mode
61 void ReadAllTraces(const NRLib::Volume *
62     volume,
63     double zPad,
64     bool onlyVolume = false);
65 //< Read all traces
66 //with header
67 float GetValue(double x,
68     double y,
69     double z,
70     int outsideMode =
71     segyIMISSING);
72
73 std::vector<float> GetAllValues(); //< Return vector with
74     all values.
75 void CreateRegularGrid();
76 const SegyGeometry * GetGeometry(void) { return geometry_
77     ;} //Only makes sense after command above.
78 //<<<End read all trace mode
79
80 //>>>Begin read single trace mode
81 const SegYTrace * GetNextTrace(double zPad = 0,
82     const NRLib::Volume *
83     volume = NULL,
84     bool onlyVolume = false);
85 //<<<End read single trace mode
86
87 //>>>Begin write mode
88 void SetGeometry(const SegyGeometry *
89     geometry);
90 void StoreTrace(float x,
91     float y,
92     const std::vector<float>
93     data,
94     const NRLib::Volume *
95     volume,
96     float topVal
97     =0.0f,
98     float baseVal
99     =0.0f);
100
101 // Write single trace to file
102 void WriteTrace(const TraceHeader &
103     traceHeader,
104     const std::vector<float> &
105     data,
106     const NRLib::Volume *
107     volume,
108     float topVal=0.0f,
109     float baseVal=0.0f);
110
111 // Write single trace to internal memory
112 void WriteTrace(float x,
113     float y,
114     const std::vector<float>
115     data,
116     const NRLib::Volume * volume
117     ,

```

```

99         float
100             topVal=0.0f,
101             float
102             baseVal=0.0f);
103     void WriteAllTracesToFile(); ///< Use only
104     after writeTrace with x and y as input is used for the whole
105     cube
106     ///<<<<End write mode
107
108     // int checkError(char * errText)
109     // {if(error_ > 0) strcpy(errText, errMsg_);return(error_);}
110     ///< Return (possibly upper limit for) number of traces
111
112     size_t GetNTraces() const { return nTraces_ ;}
113     size_t GetNz() const { return nz_ ;}
114     float GetDz() const { return dz_ ;}
115
116     enum OutsideModes{MISSING, ZERO, CLOSEST};
117
118     size_t FindNumberOfTraces(void);
119     static size_t FindNumberOfTraces(const std::string
120     & fileName,
121                                     const
122                                     TraceHeaderFormat
123                                     *
124                                     traceHeaderFormat
125                                     = NULL);
126
127     SegyGeometry * FindGridGeometry();
128     static SegyGeometry * FindGridGeometry(const std::string
129     & fileName,
130                                             const
131                                             TraceHeaderFormat
132                                             *
133                                             traceHeaderFormat
134                                             = NULL);
135
136     TraceHeaderFormat GetTraceHeaderFormat(){return
137     traceHeaderFormat_};
138     static TraceHeaderFormat FindTraceHeaderFormat(const std::string
139     & fileName);
140
141 private:
142     ///< void ebcDicHeader(std::string& outstring);
143
144     bool ReadHeader(TraceHeader * header);
145     ///< Trace header
146     SegYTrace * ReadTrace(const NRLib::Volume * volume,
147     double zPad,
148     bool &
149     duplicateHeader,
150     bool onlyVolume,
151     bool &
152     outsideSurface,
153     bool writevalues
154     = true,
155     double *
156     outsideTopBot = NULL);
157     ///< Read single trace
158     from file

```

APPENDIX B. CODE

```

136 //Note: If outsideTopBot == NULL, lack of data on top or bot will
      throw exception.
137 //      Otherwise, outsideTopBot[0] will be top lack, [1] for
      bottom,
138 //      [2] is x-coord, [3] is y-coord. Allocate outside.
139
140 void WriteMainHeader(const TextualHeader&
      ebcDicHeader); ///< Quasi-dummy at the moment.
141 void ReadDummyTrace(std::fstream &file, int
      format, size_t nz);
142 ///< Used to find correct trace header format.
143 bool CompareTraces(TraceHeader *header1,
      TraceHeader *header2, int &delta, int &deltail, int &deltaxl)
      ;
144
145
146 bool TraceHeaderOK(std::fstream &file, const
      TraceHeaderFormat *headerFormat);
147 void FindDeltaLXL(TraceHeader *t1,
      TraceHeader *t2, TraceHeader *t3, float &dil, float &dxl,
      bool x);
148 void CheckTopBotError(const double * tE,
      const double * bE); ///< Summarizes lack of data at top and
      bottom.
149
150 TraceHeaderFormat traceHeaderFormat_;
151
152 SegyGeometry * geometry_; ///< Parameters
      to find final index from i and j
153 BinaryHeader * binaryHeader_; ///<
154
155 bool singleTrace_; ///< Read one
      and one trace
156 bool simboxOnly_; ///<
157 bool checkSimbox_; ///<
158
159 std::vector<SegYTrace*> traces_; ///< All traces
160 size_t nTraces_; ///< Holds the
      number of traces. May be an estimate if not all read.
161
162 int datasize_; ///< Bytes per
      datapoint in file.
163
164 size_t nz_; ///< Number of
      time samples
165
166 float z0_; ///< Top of segy
      cube
167 float dz_; ///< Sampling
      density in time
168
169 std::fstream file_;
170 std::string fileName_;
171
172 float rmissing_;
173
174 };
175
176
177
178
179 } // namespace NRLib

```

```
180
181 #endif
```

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

```

1  #ifndef SEGY_HPP
2  #define SEGY_HPP
3
4  #include <string>
5  #include <vector>
6
7  #include "traceheader.hpp"
8  #include "commonheaders.hpp"
9  #include "../volume/volume.hpp"
10 #include "segygeometry.hpp"
11 #include "segytrace.hpp"
12 #ifdef _OPENMP
13 #include <omp.h>
14 #else
15 #define omp_get_wtime() 0
16 #pragma message("OpenMP not found. Profiling will be invalid.");
17 #endif
18
19 ///#define PROFILINGFINEGRAINED
20 ///#define PROFILING
21 #ifdef PROFILING
22 #include "../src/profiling.h"
23 extern void NRLib::Prof::setName(const std::string& name, int TID
    );
24 extern void NRLib::Prof::initProfiling();
25 extern void NRLib::Prof::trackTime(double wall, const int TID);
26 extern void NRLib::Prof::writeProfilingLog();
27 #endif
28
29 namespace NRLib {
30
31 #define GetTraceValueUnchecked(xind, yind, zind) \
32     traces_[(yind)*nx_ + (xind)]->GetValueUnchecked(zind)
33
34
35 class SegYTrace;
36 class SegyGeometry;
37 class BinaryHeader;
38 class TextualHeader;
39 enum OutsideMode{MISSING = -99999, ZERO = 0, CLOSEST = 2};
40 #define AT2D(a, b, width) ((a)*(width) + (b))
41
42 class SegY{
43 public:
44     /// Constructor for reading
45     /// Read only the headers on top of the file
46     /// \param[in] fileName Name of file to read data from
47     /// \param[in] z0
48     /// \throw IOError if the file can not be opened.
49     SegY(const std::string & fileName,
50         float z0,
51         const TraceHeaderFormat & traceHeaderFormat);
52
53     SegY(const std::string & fileName,
54         float z0,
55         const TraceHeaderFormat & traceHeaderFormat,
56         const SegyGeometry* geom);

```

APPENDIX B. CODE

```

57
58 SegY(const std::string      & fileName ,
59      float                 z0 ,
60      const TraceHeaderFormat & traceHeaderFormat ,
61      const NRLib::Volume*   volume ,
62      double                 zPad ,
63      bool                   onlyVolume);
64
65
66 /// Constructor for reading unknown format
67 /// Read only the headers on top of the file
68 /// \param[in] fileName Name of file to read data from
69 /// \param[in] z0
70 /// \param[in] thf Vector of pointers to possible
71 /// traceheaderformats. If NULL, default list is used.
72 /// \throw IOError if the file can not be opened.
73 /// \throw FileFormatError if the traceheaderformat can not be
74 /// recognized.
75 SegY(const std::string      & fileName ,
76      float                 z0 ,
77      std::vector<TraceHeaderFormat *> thf = std::vector<
78      TraceHeaderFormat *>(0) ,
79      bool                   searchStandardFormats =
80      true);
81
82 SegY(const std::string      & fileName ,
83      float                 z0 ,
84      const NRLib::Volume*   volume ,
85      double                 zPad ,
86      bool                   onlyVolume ,
87      std::vector<TraceHeaderFormat *> thf = std::vector<
88      TraceHeaderFormat *>(0) ,
89      bool                   searchStandardFormats =
90      true);
91
92 SegY(const std::string      & fileName ,
93      float                 z0 ,
94      const SegyGeometry*    geom ,
95      std::vector<TraceHeaderFormat *> thf = std::vector<
96      TraceHeaderFormat *>(0) ,
97      bool                   searchStandardFormats =
98      true);
99
100 /// Constructor for writing
101 /// \param[in] fileName Name of file to write data to
102 /// \throw IOError if the file can not be opened.
103 SegY(const std::string      & fileName ,
104      float                 z0 ,
105      int                   nz ,
106      float                 dz ,
107      const TextualHeader   & ebcDicHeader ,
108      const TraceHeaderFormat & traceHeaderFormat =
109      TraceHeaderFormat(TraceHeaderFormat::SEISWORKS));
110
111 SegY(const std::string      & fileName ,
112      float                 z0 ,
113      int                   nz ,
114      float                 dz ,
115      const TextualHeader   & ebcDicHeader ,
116      const SegyGeometry    * geom ,

```



```

109     const TraceHeaderFormat & traceHeaderFormat =
110         TraceHeaderFormat(TraceHeaderFormat::SEISWORKS);
111     ~SegY();
112
113     size_t          GetLegalIndex(size_t traceID, int
114         sampleID) const { return traces_[traceID]->GetLegalIndex(
115         sampleID); }
116     float          GetValueInVol(int xind, int yind, int
117         zind, double x, double y, double z, float v1) const;
118     float          GetValueInterpolated(int xind, int yind
119         , int zind, double x, double y, double z) const;
120     void          report(std::ofstream& stream, int xInd,
121         int yInd, int zInd) const {
122         if(static_cast<unsigned int>(xInd) >=
123             geometry_->GetNx() ||
124             static_cast<unsigned int>(yInd)
125             >= geometry_->GetNy()){
126             stream << "Overbounce\n";
127         }
128         if(xInd < 0 || yInd < 0 || zInd < 0){
129             stream << "Underbounce\n";
130         }
131     }
132     float          GetValueUnchecked(double x,
133         double y,
134         double z);
135     void          GetSample(int xind,
136         int yind,
137         int zind,
138         int xoffset,
139         int yoffset,
140         int zoffset,
141         float v1,
142         float& a,
143         float& b) const;
144     float          GetTraceValue(int xind,
145         int yind,
146         int zind) const;
147     std::vector<float> GetAllValues(); //< Return vector with
148         all values.
149     const SegyGeometry * GetGeometry(void) { return geometry_
150         ;} //Only makes sense after command above.
151     //<<<<End read all trace mode
152
153     //>>>>Begin read single trace mode
154     const SegYTrace * GetNextTrace(double zPad = 0,
155         const NRLib::Volume *
156         volume = NULL,
157         bool
158         onlyVolume = false);
159
160     //<<<<End read single trace mode
161
162     //>>>>Begin write mode
163     void          StoreTrace(float x,
164         float y,
165         const std::vector<float>
166         data,
167         const NRLib::Volume *
168         volume,

```

APPENDIX B. CODE

```

155         float                topVal
156         =0.0f ,
157         float                baseVal
158         =0.0f);
159     void WriteTrace(const TraceHeader &
160         traceHeader ,
161         const std::vector<float> &
162         data ,
163         const NRLib::Volume *
164         volume ,
165         float
166         topVal=0.0f ,
167         float
168         baseVal=0.0f);
169     void WriteTrace(float x ,
170         float y ,
171         const std::vector<float>
172         data ,
173         const NRLib::Volume * volume
174         ,
175         float
176         topVal=0.0f ,
177         float
178         baseVal=0.0f);
179     void WriteAllTracesToFile(); ///< Use only
180     after writeTrace with x and y as input is used for the whole
181     cube
182     ///<<<<End write mode
183
184     // int checkError(char * errText)
185     // {if(error_ > 0) strcpy(errText , errMsg_);return(error_);}
186     // Return (possibly upper limit for) number of traces
187
188     void ClapIndex(double x, double y, int& xInd
189     , int& yInd);
190     size_t GetNTraces() const { return nTraces_ ;}
191     size_t GetNz() const { return nz_ ;}
192     size_t GetNx() const { return nx_ ;}
193     float GetDz() const { return dz_ ;}
194     float GetZ0() const { return z0_ ;}
195
196     size_t FindNumberOfTraces(void);
197     static size_t FindNumberOfTraces(const std::string
198     & fileName ,
199     const
200     TraceHeaderFormat
201     *
202     traceHeaderFormat
203     = NULL);
204
205     SegyGeometry * FindGridGeometry();
206     static SegyGeometry * FindGridGeometry(const std::string
207     & fileName ,
208     const
209     TraceHeaderFormat
210     *
211     traceHeaderFormat
212     = NULL);

```

```

193 | TraceHeaderFormat      GetTraceHeaderFormat() { return
      |     traceHeaderFormat_ };
194 | static TraceHeaderFormat FindTraceHeaderFormat(const std::string
      |     & fileName);
195 |
196 | void                   ReadAllTraces(const NRLib::Volume *
      |     volume,
197 |                                     double           zPad,
198 |                                     bool
      |                                     onlyVolume); ///<
      |                                     Read all traces with
      |                                     header
199 | float                 GetValue(double x, double y, double z)
      |     const;
200 | bool                  GetXYID(int& xind, int& yind, double& x
      |     , double& y) const;
201 | bool                  GetZID(int xind, int yind, int& zind,
      |     double x, double y, double& z) const;
202 | bool                  GetID(int& xind, int& yind, int& zind,
      |     double& x, double& y, double& z) const;
203 | void                  setOutsideMode(OutsideMode mode);
204 | private:
205 |     OutsideMode       mode_;
206 |     float             outsideVal_;
207 |     double            (*outsideSampler_)(const SegY* trace,
      |     int xind, int yind, int zind, double z);
208 |
209 | void                  CreateRegularGrid();
210 | void                  SetGeometry(const SegyGeometry *
      |     geometry);
211 | void                  searchStdFmt(std::vector<
      |     TraceHeaderFormat*> thf, std::string fileName);
212 | void                  initSingle(const std::string      &
      |     fileName,
213 |                                     float                z0
      |     ,
214 |                                     int                  nz
      |     ,
215 |                                     float                dz
      |     ,
216 |                                     const TextualHeader  &
      |     ebcDicHeader,
217 |                                     const TraceHeaderFormat &
      |     traceHeaderFormat);
218 | void                  commonInit(const std::string & fileName
      |     , float z0);
219 | //void                ebcDicHeader(std::string& outstring);
      |     ///<
220 | bool                  ReadHeader(TraceHeader * header);
      |     ///< Trace header
221 | SegYTrace             * ReadTrace(const NRLib::Volume * volume,
      |     double           zPad,
222 |                                     bool
      |                                     &
223 |                                     duplicateHeader,
224 |                                     bool           onlyVolume,
225 |                                     bool           &
      |                                     outsideSurface,
226 |                                     bool           writevalues
      |                                     = true,
227 |                                     double           *
      |                                     outsideTopBot = NULL);
      |     ///< Read single trace

```

APPENDIX B. CODE

```

228 //Note: If outsideTopBot == NULL, lack of data on top or bot will
      //      throw exception.
229 //      Otherwise, outsideTopBot[0] will be top lack, [1] for
      //      bottom,
230 //      [2] is x-coord, [3] is y-coord. Allocate outside.
231
232 void WriteMainHeader(const TextualHeader&
      ebcDicHeader); ///< Quasi-dummy at the moment.
233 void ReadDummyTrace(std::fstream &file, int
      format, size_t nz);
234 ///< Used to find correct trace header format.
235 bool CompareTraces(TraceHeader *header1,
      TraceHeader *header2, int &delta, int &deltail, int &deltax1)
      ;
236
237
238 bool TraceHeaderOK(std::fstream &file, const
      TraceHeaderFormat *headerFormat);
239 void FindDeltaLXL(TraceHeader *t1,
      TraceHeader *t2, TraceHeader *t3, float &dil, float &dx1,
      bool x);
240 void CheckTopBotError(const double *tE,
      const double *bE); ///< Summarizes lack of data at top and
      bottom.
241
242 TraceHeaderFormat traceHeaderFormat_;
243
244 SegyGeometry * geometry_; ///< Parameters
      // to find final index from i and j
245 BinaryHeader * binaryHeader_; ///<
246
247 bool singleTrace_; ///< Read one
      // and one trace
248 bool simboxOnly_; ///<
249 bool checkSimbox_; ///<
250
251 std::vector<SegYTrace*> traces_; ///< All traces
252 size_t nTraces_; ///< Holds the
      // number of traces. May be an estimate if not all read.
253
254 int datasize_; ///< Bytes per
      // datapoint in file.
255
256 size_t nz_; ///< Number of
      // time samples
257
258 float z0_; ///< Top of segy
      // cube
259 float dz_; ///< Sampling
      // density in time
260
261 std::fstream file_;
262 std::string fileName_;
263 float rmissing_;
264
265 //Precomputed private values;
266 void setZero();
267 void copyGeometryData(const SegyGeometry* geom);
268 double x0_;
269 double y0_;
270 double dx_;

```

```
271 | double dy_;  
272 | double dxHalf_;  
273 | double dyHalf_;  
274 | size_t nx_;  
275 | size_t tracesSize_;  
276 |  
277 | };  
278 |  
279 | } // namespace NRLib  
280 |  
281 | #endif
```

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

APPENDIX B. CODE

```

1  /* old segy.cpp */
2
3  /* Unrelated code */
4
5  /* The old sampler function */
6  float
7  SegY::GetValue(double x, double y, double z, int outsideMode)
8  {
9      int i, j;
10     float xind, yind;
11     float value;
12     double x0 = geometry_>GetX0()+0.5*geometry_>GetDx()*geometry_>
        GetCosRot()-0.5*geometry_>GetDy()*geometry_>GetSinRot();
13     double y0 = geometry_>GetY0()+0.5*geometry_>GetDy()*geometry_>
        GetCosRot()+0.5*geometry_>GetDx()*geometry_>GetSinRot();
14     double sx = (x-x0)*geometry_>GetCosRot() + (y-y0)*geometry_>
        GetSinRot() + 0.5*geometry_>GetDx();
15     double sy = -(x-x0)*geometry_>GetSinRot() + (y-y0)*geometry_>
        GetCosRot() + 0.5*geometry_>GetDy();
16     if (geometry_!=NULL)
17     {
18         int ok = geometry_>FindContIndex(static_cast<float>(x),
            static_cast<float>(y), xind, yind);
19
20         i = static_cast<int>(xind);
21         j = static_cast<int>(yind);
22         size_t nx = geometry_>GetNx();
23         size_t ny = geometry_>GetNy();
24
25         size_t index;
26
27         index = j*nx+i; //NBNB er dette rett??
28
29         if (traces_[index]!=0 && ok==1 && z>=z0_ && z<=z0_+nz_*dz_)
30         {
31             size_t zind = static_cast<size_t>(floor((z-z0_)/dz_)); //
                NBNB irap grid rounding different
32
33             float v1 = traces_[index]>GetValue(zind);
34             if (v1 == rmissing_ && outsideMode == CLOSEST)
35             {
36                 zind = traces_[index]>GetLegalIndex(zind);
37                 v1 = traces_[index]>GetValue(zind);
38                 if (traces_[index]>GetValue(zind-1) == rmissing_)
39                     z = z0_+zind*dz_; // Want edge value, hence 0/1
                        dz_ added
40             else // (0.5 would give center of
                cell).
41                 z = z0_+(zind+0.99f)*dz_;
42             }
43             if (v1 != rmissing_)
44             {
45                 // Computes interpolated value ax^2+by^2+cz^2+dx+ey+fz+g.
46                 // abcdefg estimated from closest point and its closest
                neighbours.
47                 size_t maxInd = nx*ny - 1;
48                 float v0, v2, a, b, c, d, e, f, g;
49
50                 // Along x:
51                 v0 = rmissing_;
52                 v2 = rmissing_;
53                 if (index >= 1 && traces_[index-1] != NULL)

```

```

54     v0 = traces_[index-1]->GetValue(zind);
55     if (index+1 <= maxInd && traces_[index+1] != NULL)
56         v2 = traces_[index+1]->GetValue(zind);
57     if (v0 == rmissing_)
58     {
59         a = 0;
60         if (v2 == rmissing_)
61             d = 0;
62         else
63             d = v2 - v1; // Using unit coordinates in each
64                          direction.
65     }
66     else if (v2 == rmissing_)
67     {
68         a = 0;
69         d = v1 - v0;
70     }
71     else
72     {
73         a = (v2+v0-2*v1)/2.0f;
74         d = (v2-v0)/2.0f;
75     }
76     // Along y:
77     v0 = rmissing_;
78     v2 = rmissing_;
79     size_t tmpInd;
80     if(index >= nx) {
81         tmpInd = index - nx;
82         if (tmpInd <= maxInd && traces_[tmpInd] != NULL)
83             v0 = traces_[tmpInd]->GetValue(zind);
84     }
85     tmpInd = index + nx;
86     if (tmpInd <= maxInd && traces_[tmpInd] != NULL)
87         v2 = traces_[tmpInd]->GetValue(zind);
88     if (v0 == rmissing_)
89     {
90         b = 0;
91         if (v2 == rmissing_)
92             e = 0;
93         else
94             e = v2 - v1; // Using unit coordinates in each
95                          direction.
96     }
97     else if (v2 == rmissing_)
98     {
99         b = 0;
100        e = v1 - v0;
101    }
102    else
103    {
104        b = (v2+v0-2*v1)/2.0f;
105        e = (v2-v0)/2.0f;
106    }
107    //Along z:
108    v0 = traces_[index]->GetValue(zind-1);
109    v2 = traces_[index]->GetValue(zind+1);
110    if (v0 == rmissing_)
111    {
112        c = 0;
113        if (v2 == rmissing_)
114            f = 0;
115        else

```

APPENDIX B. CODE

```

114         f = v2 - v1; //Using unit coordinates in each direction
115     }
116     else if (v2 == rmissing_)
117     {
118         c = 0;
119         f = v1 - v0;
120     }
121     else
122     {
123         c = (v2+v0-2*v1)/2.0f;
124         f = (v2-v0)/2.0f;
125     }
126     g = v1;
127     double dx = geometry_>GetDx();
128     double dy = geometry_>GetDy();
129     float ux = static_cast<float>(sx/dx) - static_cast<float>(
130         floor(sx/dx) + 0.5);
131     float uy = static_cast<float>(sy/dy) - static_cast<float>(
132         floor(sy/dy) + 0.5);
133     float uz = static_cast<float>((z-z0_)/dz_) - static_cast<
134         float>(floor((z-z0_)/dz_) + 0.5);
135     value = a*ux*ux+b*uy*uy+c*uz*uz+d*ux+e*uy+f*uz+g;
136 }
137 else
138 {
139     if (outsideMode == ZERO)
140         value = 0;
141     else
142         value = rmissing_;
143 }
144 }
145 else
146 {
147     if (outsideMode == ZERO)
148         value = 0;
149     else
150         value = rmissing_;
151 }
152 }
153 else
154     value = rmissing_;
155 return(value);
}

```

Listing B.7: Old sampler in volume function.

```

1  /* new segy.cpp */
2
3  /* Unrelated code */
4
5  // #define PROFILING
6
7  const float segyRMISSING = -99999.0f;
8
9  static double noop(const NRLib::SegY* trace, int xind, int yind,
10     int zind, double z){
11     return z;
12 }

```



```

13 static double outsideClosest(const NRLib::SegY* trace, int xind,
14     int yind, int zind, double z){
15     float z0 = trace->GetZ0();
16     float dz = trace->GetDz();
17     int nx = trace->GetNx();
18     size_t index = AT2D(yind, xind, nx);
19     zind = trace->GetLegalIndex(index, zind);
20     float value;
21     value = trace->GetTraceValue(xind, yind, zind-1);
22     if (value == segyRMISSING){
23         return z0+zind*dz; // Want edge value, hence 0/1
24         dz_ added // (0.5 would give center of
25     } else{ // cell).
26         return z0+(zind+0.99f)*dz;
27     }
28 }
29 using std::ios_base;
30 namespace NRLib{
31 /* A set of calculations are common for all sampling functions.
32 these are calculated upon entry. */
33 void
34 SegY::copyGeometryData(const SegyGeometry* geometry_){
35     dx_ = geometry_>GetDx();
36     dy_ = geometry_>GetDy();
37     dxHalf_ = dx_*0.5;
38     dyHalf_ = dy_*0.5;
39     x0_ = geometry_>GetX0()+0.5*dx_*geometry_>GetCosRot()-0.5*dy_*
40     geometry_>GetSinRot();
41     y0_ = geometry_>GetY0()+0.5*dy_*geometry_>GetCosRot()+0.5*dx_*
42     geometry_>GetSinRot();
43     nx_ = geometry_>GetNx();
44     time_t ny = geometry_>GetNy();
45     tracesSize_ = nx_*ny;
46 }
47 /* Unrelated code */
48 float
49 SegY::GetValueInVol(int xind, int yind, int zind, double x, double
50     y, double z, float v1) const{
51 #ifdef PROFILING
52 #ifdef PROFILINGFINEGRAINED
53     double wall = omp_get_wtime();
54 #endif
55 // Computes interpolated value ax^2+by^2+cz^2+dx+ey+fz+g.
56 // abcdefg estimated from closest point and its closest
57     neighbours.
58     float a = 0.0f;
59     float b = 0.0f;
60     float c = 0.0f;
61     float d = 0.0f;
62     float e = 0.0f;
63     float f = 0.0f;
64     float g = v1;
65 // Along x:
66     GetSample(xind, yind, zind, 1, 0, 0, v1, a, d);

```

APPENDIX B. CODE

```

67
68 // Along y:
69 GetSample(xind, yind, zind, 0, 1, 0, v1, b, e);
70
71 // Along z:
72 GetSample(xind, yind, zind, 0, 0, 1, v1, c, f);
73
74 float sx = (x-x0_)*geometry_>GetCosRot() + (y-y0_)*geometry_>
    GetSinRot() + dxHalf_;
75 float sy = -(x-x0_)*geometry_>GetSinRot() + (y-y0_)*geometry_>
    GetCosRot() + dyHalf_;
76 float ux = static_cast<float>(sx/dx_) - static_cast<float>(floor(
    sx/dx_) + 0.5);
77 float uy = static_cast<float>(sy/dy_) - static_cast<float>(floor(
    sy/dy_) + 0.5);
78 float uz = static_cast<float>((z-z0_)/dz_) - static_cast<float>(
    floor((z-z0_)/dz_) + 0.5);
79 float returner = a*ux*ux + b*uy*uy + c*uz*uz + d*ux + e*uy + f*uz
    + g;
80
81 #ifndef PROFILING
82 #ifdef PROFILINGFINEGRAINED
83     wall = omp_get_wtime() - wall;
84     Prof::trackTime(wall, GETVALUEINVOL);
85 #endif
86 #endif
87     return returner;
88 }
89
90 /* This function performs finds two sample values along a given
    axis. */
91 void
92 SegY::GetSample(int xind, int yind, int zind, int xoffset, int
    yoffset, int zoffset, float v1, float& a, float& b) const{
93     float v0 = 0.0f;
94     float v2 = 0.0f;
95     float mulV0 = 1.0f;
96     v0 = GetTraceValue(xind - xoffset, yind - yoffset, zind - zoffset
    );
97     v2 = GetTraceValue(xind + xoffset, yind + yoffset, zind + zoffset
    );
98     if(v0 == rmissing_){
99         mulV0 = 0.0f;
100     }
101     if(v2 == rmissing_){
102         a = 0;
103         b = (v1-v0)*mulV0;
104         return;
105     }
106     a = mulV0*(v2 + v0 - 2*v1)/2.0f;
107     b = mulV0*(v2 - v0)/2.0f + (!mulV0)*(v2-v1);
108 }
109
110 /* Unrelated code */
111
112 /* This functions performs tricubic interpolation used in CRAVAs
    sampler for SEG Y.
113 * This is the "unsafe" version without borderchecks. For the safe
    version see GetValue
114 */
115 float

```

```

116 SegY::GetValueInterpolated(int xind, int yind, int zind, double x,
    double y, double z) const
117 {
118 #ifdef PROFILING
119 #ifdef PROFILINGFINEGRAINED
120     double wall = omp_get_wtime();
121 #endif
122 #endif
123     float a = GetTraceValueUnchecked(xind-1, yind, zind);
124     float b = GetTraceValueUnchecked(xind, yind-1, zind);
125     float c = GetTraceValueUnchecked(xind, yind, zind-1);
126     float d = GetTraceValueUnchecked(xind+1, yind, zind);
127     float e = GetTraceValueUnchecked(xind, yind+1, zind);
128     float f = GetTraceValueUnchecked(xind, yind, zind+1);
129     float g = GetTraceValueUnchecked(xind, yind, zind);
130     float sx = (x-x0_)*geometry_>GetCosRot() + (y-y0_)*geometry_>
        GetSinRot() + dxHalf_;
131     float sy = -(x-x0_)*geometry_>GetSinRot() + (y-y0_)*geometry_>
        GetCosRot() + dyHalf_;
132     float ux = static_cast<float>(sx/dx_) - static_cast<float>(floor(
        sx/dx_) + 0.5);
133     float uy = static_cast<float>(sy/dy_) - static_cast<float>(floor(
        sy/dy_) + 0.5);
134     float uz = static_cast<float>((z-z0_)/dz_) - static_cast<float>(
        floor((z-z0_)/dz_) + 0.5);
135     float returner = a*ux*ux + b*uy*uy + c*uz*uz + d*ux + e*uy + f*uz
        + g;
136 #ifdef PROFILING
137 #ifdef PROFILINGFINEGRAINED
138     wall = omp_get_wtime() - wall;
139     Prof::trackTime(wall, GETVALUEINTERPOLATED);
140 #endif
141 #endif
142     return returner;
143 }
144
145 /* This function checks and sets the Outside mode. It should be
    performed once per grid. */
146 void SegY::setOutsideMode(OutsideMode mode){
147     mode_ = mode;
148     switch(mode_){
149     case CLOSEST:
150         outsideSampler_ = &outsideClosest;
151         outsideVal_ = rmissing_;
152         break;
153     case MISSING:
154         outsideSampler_ = &noop;
155         outsideVal_ = rmissing_;
156         break;
157     case ZERO:
158         outsideSampler_ = &noop;
159         outsideVal_ = 0.0f;
160         break;
161     default:
162         break;
163     }
164 }
165
166 bool SegY::GetZID(int xind, int yind, int& zind, double x, double y
    , double& z) const{
167     if (z>=z0_ && z<=z0_+nz_*dz_)
168     {

```

APPENDIX B. CODE

```

169     zind = static_cast<size_t>(floor((z-z0_)/dz_)); //NBNB irap
170         grid rounding different
171     z = outsideSampler_(this, xind, yind, zind, z);
172     return true;
173 }
174 return false;
175 }
176 float SegY::GetTraceValue(int xind, int yind, int zind) const{
177     size_t index = (yind)*nx_ + (xind);
178     if(index >= 0 && index < static_cast<size_t>(tracesSize_) &&
179         traces_[index] != NULL && zind >= static_cast<int>(traces_[
180             index]->GetStart()) && zind < static_cast<int>(traces_[index
181             ]->GetEnd())){
182         return traces_[((yind)*nx_ + (xind))]->GetValueUnchecked(zind);
183     }else{
184         return rmissing_;
185     }
186 }
187 bool SegY::GetXYID(int& xind, int& yind, double& x, double& y)
188     const{
189     float xfloatid = 0, yfloatid = 0;
190     geometry_->FindContIndex(static_cast<float>(x), static_cast<
191         float>(y), xfloatid, yfloatid);
192     xind = static_cast<int>(xfloatid);
193     yind = static_cast<int>(yfloatid);
194     return true;
195 }
196 float SegY::GetValue(double x, double y, double z) const{
197     #ifdef PROFILING
198     #ifdef PROFILINGFINEGRAINED
199     double wall = 0;
200     wall = omp_get_wtime();
201     #endif
202     #endif
203     int xind = 0;
204     int yind = 0;
205     int zind = 0;
206     bool validID = GetID(xind, yind, zind, x, y, z);
207     float v1;
208     v1 = GetTraceValue(xind, yind, zind);
209     float value;
210     if (validID && v1 != rmissing_){
211         value = GetValueInVol(xind, yind, zind, x, y, z, v1);
212     }else{
213         value = outsideVal_;
214     }
215     #ifdef PROFILING
216     #ifdef PROFILINGFINEGRAINED
217     wall = omp_get_wtime() - wall;
218     Prof::trackTime(wall, GETVALUE);
219     #endif
220     #endif
221     return value;
222 }
223 bool SegY::GetID(int& xind, int& yind, int& zind, double& x,
224     double& y, double& z) const{
225     return GetXYID(xind, yind, x, y) && GetZID(xind, yind, zind, x,
226     y, z);
227 }

```

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

Listing B.8: New sampler in volume function.

B.2 Seismic Inversion

```

1  /* Unrelated code */
2  #include "src/locks.h"
3  #include "src/fftwlock.h"
4
5  //#define PROFILING
6  #ifdef PROFILING
7  #include <sstream>
8  using std::stringstream;
9  #include "src/profiling.h"
10 #define INVERSIONLOG 7
11 #define INVERSIONREADLOG 8
12 #define INVERSIONWRITELOG 11
13 #define INVCPUTIMELOG 9
14 #define INVERSIONLOOPLOG 10
15 #endif
16
17
18 #define _USE_MATH_DEFINES
19 #include <cmath>
20 #include <ctime>
21
22
23 static int classCnt = 0;
24 LOCK_T lock;
25
26 static fftw_complex** reduceVar = NULL;
27 static fftw_complex errCorrUnsmoothVal = {0.0f, 0.0f};
28 static fftw_complex** errVar = NULL;
29 static fftw_complex * ijkAns = NULL;
30 static fftw_complex * ijkDataMean = NULL;
31 static fftw_complex * ijkMean = NULL;
32 static fftw_complex * ijkMean2 = NULL;
33 static fftw_complex * ijkRes = NULL;
34 static fftw_complex * ijkRes2 = NULL;
35 static fftw_complex * ijkRes3 = NULL;
36 static fftw_complex** KScC = NULL;
37 static fftw_complex** KS = NULL;
38 static fftw_complex** margVar = NULL;
39 static fftw_complex parSPartialCorrVal = {0.0f, 0.0f};
40 static fftw_complex** parVar2 = NULL;
41 static fftw_complex** parVar = NULL;
42
43 #if _OPENMP >= 200805
44 #pragma omp threadprivate(parSPartialCorrVal, ijkMean, ijkMean2,
45     ijkDataMean, ijkAns, ijkRes, ijkRes2, ijkRes3, margVar, KScC,
46     KS, parVar2, parVar, errVar, reduceVar)
47 #endif
48 Crava::Crava(Model * model, SpatialWellFilter * spatwellfilter)
49 {
50     // Since we are using global variables for some of the methods we
51     // want them to be globally blocking to maintain
52     // reentrant properties.
53     // That is maintained by locking the affected methods upon entry.
54     // To be able to use locks we have to initialize them before usage
55     // , and destroy them after usage.
56     // This is solved by adding a class counter (named classCnt) in
57     // all CRAVA constructors that ensures that the first crava
58     // instance

```

```

54 // initializes the lock, and the last crava instance destroys the
    lock (named lock).
55 #pragma omp critical (CRAVA)
56 {
57     if (classCnt == 0) {
58         omp_init_lock(&lock);
59     }
60     classCnt += 1;
61 }
62 Utils::writeHeader("Building Stochastic Model");
63
64 double wall=0.0, cpu=0.0;
65 TimeKit::getTime(wall,cpu);
66
67 model_          = model;
68 nx_             = model->getBackAlpha()->getNx();
69 ny_             = model->getBackAlpha()->getNy();
70 nz_             = model->getBackAlpha()->getNz();
71 nxp_            = model->getBackAlpha()->getNxp();
72 nyp_            = model->getBackAlpha()->getNyp();
73 nzp_            = model->getBackAlpha()->getNzp();
74 lowCut_        = model->getModelSettings()->getLowCut();
75 highCut_       = model->getModelSettings()->getHighCut();
76 wnc_           = model->getModelSettings()->getWNC(); //
77     white noise component see crava.h
78 energyTreshold_ = model->getModelSettings()->
    getEnergyThreshold();
79 ntheta_        = model->getModelSettings()->getNumberOfAngles
    ();
80 fileGrid_      = model->getModelSettings()->getFileGrid();
81 outputGridsSeismic_ = model->getModelSettings()->
    getOutputGridsSeismic();
82 outputGridsElastic_ = model->getModelSettings()->
    getOutputGridsElastic();
83 writePrediction_ = model->getModelSettings()->
    getWritePrediction();
84 krigingParameter_ = model->getModelSettings()->
    getKrigingParameter();
85 nWells_        = model->getModelSettings()->getNumberOfWells
    ();
86 nSim_          = model->getModelSettings()->
    getNumberOfSimulations();
87 wells_         = model->getWells();
88 simbox_        = model->getTimeSimbox();
89 meanAlpha_     = model->getBackAlpha();
90 meanBeta_      = model->getBackBeta();
91 meanRho_       = model->getBackRho();
92 correlations_  = model->getCorrelations();
93 random_        = model->getRandomGen();
94 seisWavelet_   = model->getWavelets();
95 A_             = model->getAMatrix();
96 postAlpha_     = meanAlpha_; // Write over the input
    to save memory
97 postBeta_      = meanBeta_; // Write over the input
    to save memory
98 postRho_       = meanRho_; // Write over the input
    to save memory
99 fprob_         = NULL;
100 thetaDeg_     = new float[ntheta_];
101 empSNRatio_   = new float[ntheta_];
102 theoSNRatio_  = new float[ntheta_];
103 modelVariance_ = new float[ntheta_];

```

APPENDIX B. CODE

```

103 | signalVariance_ = new float[ntheta_];
104 | errorVariance_ = new float[ntheta_];
105 | dataVariance_ = new float[ntheta_];
106 | scaleWarning_ = 0;
107 | scaleWarningText_ = "";
108 | errThetaCov_ = new double*[ntheta_];
109 | sigmamdnw_ = NULL;
110 | for(int i=0;i<ntheta_;i++) {
111 |     errThetaCov_[i] = new double[ntheta_];
112 |     thetaDeg_[i] = static_cast<float>(model->getModelSettings()
113 |         ->getAngle(i)*180.0/M_PI);
114 | }
115 | seisData_ = NULL;
116 | fftw_real * corrT = NULL; // = fftw_malloc(2*(nzp_/2+1)*sizeof(
117 |     fftw_real));
118 | // Double-use grids to save memory
119 | FFTGrid * parSpatialCorr = NULL; // Parameter correlation
120 | FFTGrid * errCorrUnsmooth = NULL; // Error correlation
121 |
122 | if(!model->getModelSettings()->getForwardModeling())
123 | {
124 |     seisData_ = model->getSeisCubes();
125 |     model->releaseGrids();
126 |     correlations_ ->createPostGrids(nx_,ny_,nz_,npx_,npy_,nzp_,
127 |         fileGrid_);
128 |     parPointCov_ = correlations_ ->getPriorVar0();
129 |     parSpatialCorr = correlations_ ->getPostCovAlpha(); // Double-
130 |         use grids to save memory
131 |     errCorrUnsmooth = correlations_ ->getPostCovBeta(); // Double-
132 |         use grids to save memory
133 |     // NBNB nzp_*0.001*corr ->getdt() = T lowCut = lowIntCut*
134 |         omega = lowIntCut/T
135 |     int lowIntCut = int(floor(lowCut_*(nzp_*0.001*correlations_ ->
136 |         getdt())));
137 |     // computes the integer whis corresponds to the low cut
138 |         frequency.
139 |     float corrGradI, corrGradJ;
140 |     model->getCorrGradIJ(corrGradI, corrGradJ);
141 |     corrT = parSpatialCorr->fillInParamCorr(correlations_,lowIntCut
142 |         ,corrGradI, corrGradJ);
143 |     if(spatwellfilter!=NULL)
144 |     {
145 |         parSpatialCorr->setAccessMode(FFTGrid::RANDOMACCESS);
146 |         for(int i=0;i<nWells_;i++)
147 |             spatwellfilter->setPriorSpatialCorr(parSpatialCorr, wells_[
148 |                 i], i);
149 |         parSpatialCorr->endAccess();
150 |     }
151 |     correlations_ ->setPriorCorrTFiltered(corrT,nz_,nzp_); // Can
152 |         has zeros in the middle
153 |     errCorrUnsmooth->fillInErrCorr(correlations_,corrGradI,
154 |         corrGradJ);
155 |     if((model->getModelSettings()->getOtherOutputFlag() & IO::
156 |         PRIORCORRELATIONS) > 0)
157 |         correlations_ ->writeFilePriorCorrT(corrT,nzp_); // No
158 |         zeros in the middle
159 |     }
160 | else
161 | {
162 |     model->releaseGrids();

```



```

151     }
152
153     // reality check: all dimensions involved match
154     assert(meanBeta_ ->consistentSize(nx_,ny_,nz_,nxp_,nyp_,nzp_));
155     assert(meanRho_ ->consistentSize(nx_,ny_,nz_,nxp_,nyp_,nzp_));
156
157     for(int i=0 ; i< ntheta_ ; i++)
158     {
159         if(!model->getModelSettings()->getForwardModeling())
160             assert(seisData_[i]->consistentSize(nx_,ny_,nz_,nxp_,nyp_,
161                 nzp_));
161         assert(seisWavelet_[i]->consistentSize(nzp_));
162     }
163
164     if(!model->getModelSettings()->getForwardModeling())
165     {
166         parSpatialCorr->fftInPlace();
167         computeVariances(corrT,model->getModelSettings());
168         scaleWarning_ = checkScale(); // fills in scaleWarningText_ if
169             needed.
170         fftw_free(corrT);
171         if(simbox_->getIsConstantThick() == false)
172             divideDataByScaleWavelet();
173         errCorrUnsmooth->fftInPlace();
174         for(int i = 0 ; i < ntheta_ ; i++)
175         {
176             seisData_[i]->setAccessMode(FFTGrid::RANDOMACCESS);
177             seisData_[i]->fftInPlace();
178             seisData_[i]->endAccess();
179         }
180     }
181
182     if ((model->getModelSettings()->getEstimateFaciesProb() && model
183         ->getModelSettings()->getFaciesProbRelative())
184         || model->getModelSettings()->getUseLocalNoise())
185     {
186         // meanAlpha_ ->setAccessMode(FFTGrid::READ);
187         meanAlpha2_ = copyFFTGrid(meanAlpha_);
188         // meanAlpha2_ ->endAccess();
189         // meanBeta_ ->setAccessMode(FFTGrid::READ);
190         // meanRho_ ->setAccessMode(FFTGrid::READ);
191
192         meanBeta2_ = copyFFTGrid(meanBeta_);
193         meanRho2_ = copyFFTGrid(meanRho_);
194     }
195
196     meanAlpha_ ->fftInPlace();
197     meanBeta_ ->fftInPlace();
198     meanRho_ ->fftInPlace();
199
200     Timings:: setTimeStochasticModel(wall,cpu);
201 }
202
203 Crava::~Crava()
204 {
205     delete [] thetaDeg_;
206     delete [] empSNRatio_;
207     delete [] theoSNRatio_;
208     delete [] modelVariance_;
209     delete [] signalVariance_;
210     delete [] errorVariance_;
211     delete [] dataVariance_;

```

APPENDIX B. CODE

```

210     if(fprob_!=NULL) delete fprob_ ;
211
212     for(int i = 0;i<ntheta_ ;i++)
213         delete [] errThetaCov_[i];
214     delete [] errThetaCov_ ;
215
216     if(postAlpha_!=NULL) delete postAlpha_ ;
217     if(postBeta_!=NULL) delete postBeta_ ;
218     if(postRho_!=NULL) delete postRho_ ;
219
220     const int nx = nx_ ;
221     const int ny = ny_ ;
222     if(sigmamdnew_!=NULL)
223     {
224         int i, j;
225 #pragma omp parallel if(nx*ny > SIMPLEOVERHEADLIMIT) private(i, j)
226 #pragma omp for
227         for(i=0;i<nx;i++)
228         {
229             for(j=0;j<ny;j++)
230             {
231                 if((*sigmamdnew_)(i, j)!=NULL)
232                 {
233                     for(int ii=0;ii<3;ii++)
234                         delete [] (*sigmamdnew_)(i, j)[ii];
235                     delete [] (*sigmamdnew_)(i, j);
236                 }
237             }
238         }
239         delete sigmamdnew_ ;
240     }
241
242     //This is the global lock destructor. See the comments in the
243     //constructor for a full description.
244 #pragma omp critical (CRAVA)
245 {
246     classCnt --=1;
247     if(classCnt == 0){
248         omp_destroy_lock(&lock);
249     }
250 }
251
252 /* This comment replaces 873 lines not related to inversion */
253
254 static void fillErrorMatrix(float wnc, const double** errThetaCov,
255                             double scale, const fftw_complex* errMult1, const fftw_complex*
256                             errMult2, const fftw_complex* errMult3, int matrixSize,
257                             fftw_complex** errVar){
258     for(int l = 0; l < matrixSize; l++){
259         for(int m = 0; m < matrixSize; m++){ // Note we multiply
260             kWNorm[l] and comp.conj(kWNorm[m]) hence the + and not a
261             minus as in pure multiplication
262             errVar[l][m].re = static_cast<float>(
263                 0.5f*(1.0f-wnc)*errThetaCov[l][m]*scale*( errMult1[l].re*
264                     errMult1[m].re + errMult1[l].im* errMult1[m].im) +
265                 0.5f*(1.0f-wnc)*errThetaCov[l][m]*scale*( errMult2[l].re*
266                     errMult2[m].re + errMult2[l].im* errMult2[m].im));
267         }
268     }
269     for(int l = 0; l < matrixSize; l++){

```

```

263     errVar[1][1].re += static_cast<float>(wnc*errThetaCov[1][1] *
264         errMult3[1].re * errMult3[1].re);
265     errVar[1][1].im = 0.0f;
266 }
267 for(int l = 0; l < matrixSize; l++){
268     for(int m = l+1; m < matrixSize; m++){
269         errVar[1][m].im = static_cast<float>(
270             0.5f*(1.0f-wnc)*(errThetaCov[1][m]*scale)*(-errMult1[1].
271                 re*errMult1[m].im + errMult1[1].im*errMult1[m].re) +
272             0.5f*(1.0f-wnc)*(errThetaCov[1][m]*scale)*(-errMult2[1].
273                 re*errMult2[m].im + errMult2[1].im*errMult2[m].re));
274     }
275 }
276 for(int l = 0; l < matrixSize; l++){
277     for(int m = 0; m < l; m++){
278         errVar[1][m].im = static_cast<float>(
279             0.5f*(1.0f-wnc)*(errThetaCov[1][m]*scale)*(-errMult1[1].
280                 re*errMult1[m].im + errMult1[1].im*errMult1[m].re) +
281             0.5f*(1.0f-wnc)*(errThetaCov[1][m]*scale)*(-errMult2[1].
282                 re*errMult2[m].im + errMult2[1].im*errMult2[m].re));
283     }
284 }
285 }
286 }
287 #define PROCESS_DATA(TID) \
288     double ijkErrLamRe = static_cast<float>(fabs(
289         errCorrUnsmoothVal.re)); \
290     fillErrorMatrix(wnc_, const_cast<const double**>(
291         errThetaCov_), ijkErrLamRe, errMult1, errMult2,
292         errMult3, ntheta_, errVar); \
293     lib_matrProdCpx(K, parVar2, ntheta_, 3, 3, KS); \
294     lib_matrProdAdjointCpx(KS, K, ntheta_, 3, ntheta_, margVar)
295     ; \
296     lib_matrAddMatCpx(errVar, ntheta_, ntheta_, margVar); \
297     if(lib_matrCholCpx(ntheta_, margVar) == 0){ \
298         lib_matrAdjoint(KS, ntheta_, 3, KSc); \
299         lib_matrAXeqBMatCpx(ntheta_, margVar, KS, 3); \
300         lib_matrProdCpx(KSc, KS, 3, ntheta_, 3, reduceVar); \
301         lib_matrSubtMatCpx(reduceVar, 3, 3, parVar2); \
302         lib_matrProdMatVecCpx(K, ijkMean2, ntheta_, 3, ijkDataMean
303             ); \
304         for(i = 0; i < ntheta_; i++){ \
305             ijkRes2[i].re = ijkRes[i].re; \
306             ijkRes2[i].im = ijkRes[i].im; \
307             ijkRes3[i].re = ijkRes[i].re; \
308             ijkRes3[i].im = ijkRes[i].im; \
309         } \
310         lib_matrSubtVecCpx(ijkDataMean, ntheta_, ijkRes2); \
311         lib_matrProdAdjointMatVecCpx(KS, ijkRes2, 3, ntheta_,
312             ijkAns); \
313         lib_matrAddVecCpx(ijkAns, 3, ijkMean2); \
314         lib_matrProdMatVecCpx(K, ijkMean2, ntheta_, 3, ijkRes2); \
315         lib_matrSubtVecCpx(ijkRes2, ntheta_, ijkRes3); \
316     }
317 }
318 int
319 Crava::computePostMeanResidAndFFTCov()
320 {
321     // This method is globally blocking.
322     // Two independant calls for computePostMeanResidAndFFTCov() from
323     // two (in)dependant independent instances will execute in
324     // serial.

```

APPENDIX B. CODE

```

312 //
313 // computePostMeanResidAndFFTCov() is designed this way because
      openMP requires threadprivate variables to be static.
314 // computePostMeanResidAndFFTCov() exploits threadprivate to be
      able to perform expensive calls once.
315 #ifdef PROFILING
316 double wtime = omp_get_wtime();
317 double ptimeAccum = 0.0;
318 double invLoop = 0.0;
319 double readTimeAccum = 0.0;
320 double writeTimeAccum = 0.0;
321 #endif
322
323 omp_set_lock(&lock);
324 Utils::writeHeader("Posterior model / Performing Inversion");
325 if(seisData_ == NULL) return 1;
326 double wall=0.0, cpu=0.0;
327 TimeKit::getTime(wall,cpu);
328 int i,j,k,l;
329 const float lowCut = lowCut_;
330 const double simboxMinRelThick = simbox_>getMinRelThick();
331 const float highCut = highCut_;
332 const double lz = simbox_>getLz();
333 const int ntheta = ntheta_;
334 const int nzp = nzp_;
335 const int nz = nz_;
336 const int cnxp = nxp_/2+1;
337 const int nyp_cnxp = nyp_*cnxp;
338 const float delta = static_cast<float>((nz*1000.0f)/(lz*nzp));
339 const float monitorSize = std::max(1.0f, static_cast<float>(nzp_)
      *0.02f);
340 float nextMonitor = monitorSize;
341
342 Wavelet * diff1Operator = new Wavelet(Wavelet::
      FIRSTORDERFORWARDIFF,nz_,nzp_);
343 Wavelet * diff2Operator = new Wavelet(diff1Operator, Wavelet::
      FIRSTORDERBACKWARDIFF);
344 Wavelet * diff3Operator = new Wavelet(diff2Operator, Wavelet::
      FIRSTORDERCENTRALDIFF);
345
346 diff1Operator->fft1DInPlace();
347 delete diff2Operator;
348 diff3Operator->fft1DInPlace();
349
350 Wavelet ** errorSmooth = new Wavelet*[ntheta];
351 Wavelet ** errorSmooth2 = new Wavelet*[ntheta];
352 Wavelet ** errorSmooth3 = new Wavelet*[ntheta];
353
354 for(l = 0; l < ntheta ; l++){
355     std::string angle = NRLib::ToString(thetaDeg_[l], 1);
356     std::string fileName;
357     seisData_[l]->setAccessMode(FFTGrid::READANDWRITE);
358     if (seisWavelet_[0]->getDim() == 1) {
359         errorSmooth[l] = new Wavelet(seisWavelet_[l], Wavelet::
            FIRSTORDERFORWARDIFF);
360         errorSmooth2[l] = new Wavelet(errorSmooth[l], Wavelet::
            FIRSTORDERBACKWARDIFF);
361         errorSmooth3[l] = new Wavelet(errorSmooth2[l], Wavelet::
            FIRSTORDERCENTRALDIFF);
362         fileName = std::string("ErrorSmooth_") + angle + IO::
            SuffixGeneralData();
363         errorSmooth3[l]->printToFile(fileName);

```

```

364     errorSmooth3[1]->fft1DInPlace();
365
366     fileName = IO::PrefixWavelet() + angle + IO::
367         SuffixGeneralData();
368     seisWavelet_[1]->printToFile(fileName);
369     seisWavelet_[1]->fft1DInPlace();
370
371     fileName = std::string("FourierWavelet_") + angle + IO::
372         SuffixGeneralData();
373     seisWavelet_[1]->printToFile(fileName);
374     delete errorSmooth[1];
375     delete errorSmooth2[1];
376 }
377 }
378 delete[] errorSmooth;
379 delete[] errorSmooth2;
380
381 meanAlpha_ ->setAccessMode(FFTGrid::READANDWRITE); // Note
382 meanBeta_  ->setAccessMode(FFTGrid::READANDWRITE); // the top
383             five are over written
384 meanRho_   ->setAccessMode(FFTGrid::READANDWRITE); // does not
385             have the initial meaning.
386
387 FFTGrid * parSpatialCorr = correlations_ ->getPostCovAlpha();
388 // NB! Note double usage of postCovAlpha
389 FFTGrid * errCorrUnsmooth = correlations_ ->getPostCovBeta();
390 // NB! Note double usage of postCovBeta
391 FFTGrid * postCovAlpha = correlations_ ->getPostCovAlpha();
392 FFTGrid * postCovBeta = correlations_ ->getPostCovBeta();
393 FFTGrid * postCovRho = correlations_ ->getPostCovRho();
394 FFTGrid * postCrCovAlphaBeta = correlations_ ->
395     getPostCrCovAlphaBeta();
396 FFTGrid * postCrCovAlphaRho = correlations_ ->
397     getPostCrCovAlphaRho();
398 FFTGrid * postCrCovBetaRho = correlations_ ->getPostCrCovBetaRho
399     ();
400 parSpatialCorr ->setAccessMode(FFTGrid::READANDWRITE); //
401     after the processing
402 errCorrUnsmooth ->setAccessMode(FFTGrid::READANDWRITE); //
403 postCovRho ->setAccessMode(FFTGrid::WRITE);
404 postCrCovAlphaBeta ->setAccessMode(FFTGrid::WRITE);
405 postCrCovAlphaRho ->setAccessMode(FFTGrid::WRITE);
406 postCrCovBetaRho ->setAccessMode(FFTGrid::WRITE);
407
408 // long int timestart, timeend;
409 // time(&timestart);
410
411 LogKit::LogFormatted(LogKit::LOW, "\nBuilding posterior
412     distribution:");
413 LogKit::LogMessage(LogKit::HIGH, "\n 0%      20%      40%
414     60%      80%      100% \
415     \n | | | | | | | | | | | | | | \
416     \n ^");
417
418 fftw_complex * errMult1 = new fftw_complex[ntheta];
419 fftw_complex * errMult2 = new fftw_complex[ntheta];
420 fftw_complex * errMult3 = new fftw_complex[ntheta];
421 fftw_complex** K = new fftw_complex*[ntheta];
422 for(int iter = 0; iter < ntheta; iter++){
423     K[iter] = new fftw_complex[3];
424 }

```

APPENDIX B. CODE

```

414
415 // Memory is allocated once per thread which means that each
416 // thread has their own memory area.
417 // Noticed that all these variables are marked threadprivate and
418 // therefor store data between parallel
419 // blocks
420 #if _OPENMP >= 200805
421 #pragma omp parallel
422 {
423 #endif
424     reduceVar = new fftw_complex*[3];
425     errVar = new fftw_complex*[ntheta];
426     ijkAns = new fftw_complex[3];
427     ijkDataMean = new fftw_complex[ntheta];
428     ijkMean = new fftw_complex[3];
429     ijkMean2 = new fftw_complex[3];
430     ijkRes = new fftw_complex[ntheta];
431     ijkRes2 = new fftw_complex[ntheta];
432     KSc = new fftw_complex*[3]; // cc - complex conjugate (and
433 // transposed)
434     KS = new fftw_complex*[ntheta];
435     margVar = new fftw_complex*[ntheta];
436     parVar2 = new fftw_complex*[3];
437     parVar = new fftw_complex*[3];
438     ijkRes3 = new fftw_complex[ntheta];
439     for(int iter = 0; iter < ntheta; iter++){
440         errVar[iter] = new fftw_complex[ntheta];
441         KS[iter] = new fftw_complex[3];
442         margVar[iter] = new fftw_complex[ntheta];
443     }
444     for(int iter = 0; iter < 3; iter++){
445         reduceVar[iter] = new fftw_complex[3];
446         KSc[iter] = new fftw_complex[ntheta];
447         parVar2[iter] = new fftw_complex[3];
448         parVar[iter] = new fftw_complex[3];
449     }
450 #if _OPENMP >= 200805
451 #pragma omp single copyprivate(parVar)
452 #endif
453 {
454     for(int iter = 0; iter < 3; iter++){
455         for(int iter2 = 0; iter2 < 3; iter2++){
456             parVar[iter][iter2].re = parPointCov_[iter][iter2];
457             parVar[iter][iter2].im = 0.0;
458         }
459     }
460 #if _OPENMP >= 200805
461 #endif
462 }
463 #endif
464 // Each thread performs a small part of the serial code and
465 // stores their part of the result.
466 // Explicitly specifying the same scheduler for all parallel
467 // blocks makes the result of parallel blocks
468 // to be shared between parallel regions.
469 //
470 // Note:
471 // * The default scheduler is implementation dependant.
472 // * schedule(static, 1) == round robin. thread 0, 1, 2, 3, ...
473 //

```

```

471 #ifndef PROFILING
472     invLoop = omp_get_wtime();
473 #endif
474     for(k = 0; k < nzp; k++){
475         fftw_complex kD = diff1Operator->getCamp(k);
476         // defines content of kD
477         if (seisWavelet_[0]->getDim() == 1) { //1D-wavelet
478             if( simbox_->getIsConstantThick() == true)
479             {
480                 // defines content of K=WDA
481                 fillkW(k, errMult1); //
482                 errMult1 used as dummy
483                 lib_matrProdScalVecCpx(kD, errMult1, ntheta); //
484                 errMult1 used as dummy
485                 lib_matrProdDiagCpxR(errMult1, A_, ntheta, 3, K); //
486                 defines content of (WDA) K
487
488                 // defines error-term multipliers
489                 fillkWNorm(k, errMult1, seisWavelet_); //
490                 defines input of (kWNorm) errMult1
491                 fillkWNorm(k, errMult2, errorSmooth3); //
492                 defines input of (kWD3Norm) errMult2
493                 lib_matrFillOnesVecCpx(errMult3, ntheta); //
494                 defines content of errMult3
495             } //simbox_->getIsConstantThick() == false
496             }else{
497                 fftw_complex kD3 = diff3Operator->getCamp(k); //
498                 defines kD3
499
500                 // defines content of K = DA
501                 lib_matrFillValueVecCpx(kD, errMult1, ntheta); //
502                 errMult1 used as dummy
503                 lib_matrProdDiagCpxR(errMult1, A_, ntheta, 3, K); //
504                 defines content of ( K = DA )
505
506                 // defines error-term multipliers
507                 lib_matrFillOnesVecCpx(errMult1, ntheta); // defines
508                 content of errMult1
509                 for(l=0; l < ntheta; l++)
510                     errMult1[l].re /= seisWavelet_[l]->getNorm();
511
512                 lib_matrFillValueVecCpx(kD3, errMult2, ntheta); // defines
513                 content of errMult2
514                 for(l=0; l < ntheta; l++)
515                 {
516                     errMult2[l].re /= errorSmooth3[l]->getNorm(); // defines
517                     content of errMult2
518                     errMult2[l].im /= errorSmooth3[l]->getNorm(); // defines
519                     content of errMult2
520                 }
521                 fillInverseAbskWRobust(k, errMult3); // defines
522                 content of errMult3
523             } //simbox_->getIsConstantThick()
524         }
525
526         // Log progress
527         if (k > static_cast<int>(nextMonitor)){
528             nextMonitor += monitorSize;
529             LogKit::LogMessage(LogKit::LOW, "^");
530         }
531
532     bool sequentialInput = meanAlpha_->allowsRandomRead();

```

APPENDIX B. CODE

```

518     sequentialInput = meanBeta_ ->allowsRandomRead();
519     sequentialInput = meanRho_ ->allowsRandomRead();
520     sequentialInput = parSpatialCorr ->allowsRandomRead();
521     sequentialInput = errCorrUnsmooth ->allowsRandomRead();
522     for(int iter = 0; iter < ntheta; iter++){
523         sequentialInput &= seisData_[iter] ->allowsRandomRead();
524     }
525     bool sequentialOutput = postCovAlpha ->allowsRandomWrite();
526     sequentialOutput &= postCovBeta ->allowsRandomWrite();
527     sequentialOutput &= postCovRho ->allowsRandomWrite();
528     sequentialOutput &= postCrCovAlphaBeta ->allowsRandomWrite
529         ();
530     sequentialOutput &= postCrCovAlphaRho ->allowsRandomWrite
531         ();
532     sequentialOutput &= postCrCovBetaRho ->allowsRandomWrite
533         ();
534     sequentialOutput &= postAlpha_ ->allowsRandomWrite();
535     sequentialOutput &= postBeta_ ->allowsRandomWrite();
536     sequentialOutput &= postRho_ ->allowsRandomWrite();
537     for(int iter=0; iter < ntheta; iter++){
538         sequentialOutput &= seisData_[iter] ->allowsRandomWrite();
539     }
540     int readSpinlock = 0;
541     int writeSpinlock = 0;
542     #if _OPENMP >= 200805
543     #pragma omp parallel for ordered private(j) default(shared)
544         schedule(static, 1)
545     #endif
546     for(j = 0; j < nyp_cnxp; j++){
547         int idI = k;
548         int idJ = j/cnxp;
549         int idK = j%cnxp;
550         //START READ
551         // A ordered section ensures sequential ordering
552         // Sequential ordering is essential in this part in order to
553         // retain sequential I/O to disks.
554         while(sequentialInput && readSpinlock != j);
555     #ifndef PROFILING
556         const double readTime = omp_get_wtime();
557     #endif
558     #pragma omp ordered
559     errCorrUnsmoothVal = errCorrUnsmooth ->getComplexValue(idK,
560         idJ, idI, true);
561     ijkMean[0] = meanAlpha_ ->getComplexValue(idK, idJ, idI, true)
562         ;
563     ijkMean[1] = meanBeta_ ->getComplexValue(idK, idJ, idI, true)
564         ;
565     ijkMean[2] = meanRho_ ->getComplexValue(idK, idJ, idI, true)
566         ;
567     parSpatialCorrVal = parSpatialCorr ->getComplexValue(idK, idJ
568         , idI, true);
569     for(int iter = 0; iter < ntheta; iter++){
570         ijkRes[iter] = seisData_[iter] ->getComplexValue(idK, idJ,
571             idI, true);
572     }
573     #ifndef PROFILING
574     readTimeAccum += omp_get_wtime() - readTime;
575     #endif
576     #pragma omp atomic
577     readSpinlock += 1;
578     #pragma omp flush(readSpinlock)
579     //END READ

```



```

569 //START COMPUTE
570
571 for(int iter = 0; iter < 3; iter++){
572     ijkMean2[iter].im = ijkMean[iter].im;
573     ijkMean2[iter].re = ijkMean[iter].re;
574 }
575 float ijkParLamRe = fabs(parSPartialCorrVal.re);
576 for(int iter = 0; iter < 3; iter++){
577     for(int iter2 = 0; iter2 < 3; iter2++){
578         parVar2[iter][iter2].re = static_cast<fftw_real>(parVar[
579             iter][iter2].re * ijkParLamRe);
580         parVar2[iter][iter2].im = static_cast<fftw_real>(parVar[
581             iter][iter2].im * ijkParLamRe);
582     }
583 }
584 float realFrequency = delta*std::min(k, nzp-k); // the
585     physical frequency
586 bool current = (realFrequency > lowCut*simboxMinRelThick &&
587     realFrequency < highCut); // inverting only relevant
588     frequencies
589 for(int iter = 0; iter < 3; iter++){
590     ijkMean2[iter].im = ijkMean[iter].im;
591     ijkMean2[iter].re = ijkMean[iter].re;
592 }
593 if(current){
594     PROCESS_DATA(omp_get_thread_num());
595 }
596
597 // END COMPUTE
598 // START WRITE
599 // A spinlock is used to force serial execution without use
600 // of the ordered because ordered can only
601 // be used once per iteration
602 // A spinlock works by continuously testing a condition until
603 // it fails. This is more resource demanding,
604 // than using locks based on interrupts.
605 // When there is nothing better to use the resources on a
606 // spinlock is as good as any lock.
607 while(sequentialOutput && writeSpinlock != j);
608 #ifndef PROFILING
609 const double writeTime = omp_get_wtime();
610 #endif
611 postCovAlpha->setComplexValue(idK, idJ, idI, parVar2[0][0],
612     true);
613 postCovBeta ->setComplexValue(idK, idJ, idI, parVar2[1][1],
614     true);
615 postCovRho ->setComplexValue(idK, idJ, idI, parVar2[2][2],
616     true);
617 postCrCovAlphaBeta->setComplexValue(idK, idJ, idI, parVar2
618     [0][1], true);
619 postCrCovAlphaRho ->setComplexValue(idK, idJ, idI, parVar2
620     [0][2], true);
621 postCrCovBetaRho ->setComplexValue(idK, idJ, idI, parVar2
622     [1][2], true);
623 for(int iter=0;iter<ntheta; iter++){
624     seisData_[iter]->setComplexValue(idK, idJ, idI, ijkRes3[
625         iter], true);
626 }
627 postAlpha_->setComplexValue(idK, idJ, idI, ijkMean2[0], true)
628 ;
629 postBeta_->setComplexValue(idK, idJ, idI, ijkMean2[1], true)
630 ;

```

APPENDIX B. CODE

```

614     postRho_  ->setComplexValue(idK, idJ, idI, ijkMean2[2], true)
615     ;
615 #ifdef PROFILING
616     writeTimeAccum += omp_get_wtime() - writeTime;
617     ptimeAccum += omp_get_wtime() - readTime;
618 #endif
619     // Release the lock.
620 #pragma omp atomic
621     writeSpinlock += 1;
622 #pragma omp flush(writeSpinlock)
623     //END WRITE
624 }
625 }
626 #ifdef PROFILING
627     invLoop = omp_get_wtime() - invLoop;
628 #endif
629
630     LogKit::LogMessage(LogKit::LOW, "\n");
631     // Parallel memory cleanup. Each threads cleans up their local
632     // copy of threadprivate memory.
633     // All calls in parallel blocks happens the same times as the
634     // number of threads
635 #if _OPENMP >= 200805
636 #pragma omp parallel private(j)
637 #endif
638 {
639     for(int iter = 0; iter < ntheta; iter++){
640         delete [] errVar[iter];
641         delete [] KS[iter];
642         delete [] margVar[iter] ;
643     }
644     for(int iter = 0; iter < 3; iter++){
645         delete [] KScC[iter];
646         //delete [] parVar[iter];
647         delete [] parVar2[iter];
648         delete [] reduceVar[iter];
649     }
650     delete [] errVar;
651     delete [] ijkAns;
652     delete [] ijkDataMean;
653     delete [] ijkMean;
654     delete [] ijkMean2;
655     delete [] ijkRes;
656     delete [] ijkRes2;
657     delete [] ijkRes3;
658     delete [] KS;
659     delete [] KScC;
660     delete [] margVar;
661     //delete [] parVar;
662     delete [] parVar2;
663     delete [] reduceVar;
664 }
665 for(i = 0; i < ntheta; i++){
666     delete errorSmooth3[i];
667     delete [] K[i];
668 }
669 delete diff1Operator;
670 delete diff3Operator;
671 delete [] errMult1;
672 delete [] errMult2;
673 delete [] errMult3;
674 delete [] errorSmooth3;

```

```

673 delete [] K;
674
675 // time(&timeend);
676 // LogKit::LogFormatted(LogKit::LOW,"\n Core inversion finished
        after %ld seconds ***\n",timeend-timestart);
677 // these does not have the initial meaning
678 meanAlpha_ = NULL; // the content is taken care of by
        postAlpha_
679 meanBeta_ = NULL; // the content is taken care of by
        postBeta_
680 meanRho_ = NULL; // the content is taken care of by
        postRho_
681 parSpatialCorr = NULL; // the content is taken care of by
        postCovAlpha
682 errCorrUnsmooth = NULL; // the content is taken care of by
        postCovBeta
683
684 postAlpha_->endAccess();
685 postBeta_->endAccess();
686 postRho_->endAccess();
687
688 postCovAlpha->endAccess();
689 postCovBeta->endAccess();
690 postCovRho->endAccess();
691 postCrCovAlphaBeta->endAccess();
692 postCrCovAlphaRho->endAccess();
693 postCrCovBetaRho->endAccess();
694
695 postAlpha_->invFFTIInPlace();
696 postBeta_->invFFTIInPlace();
697 postRho_->invFFTIInPlace();
698
699 for (l=0;l<ntheta;l++)
700     seisData_[l]->endAccess();
701
702 //Finish use of seisData_, since we need the memory.
703 if((outputGridsSeismic_ & IO::RESIDUAL) > 0)
704 {
705     if (simbox_->getIsConstantThick() != true)
706         multiplyDataByScaleWaveletAndWriteToFile("residuals");
707     else
708     {
709         for (l=0;l<ntheta;l++)
710         {
711             std::string angle = NRLib::ToString(thetaDeg_[l],1);
712             std::string sgriLabel = " Residuals for incidence angle "+
                angle;
713             std::string fileName = IO::PrefixResiduals() + angle;
714             seisData_[l]->setAccessMode(FFTGrid::RANDOMACCESS);
715             seisData_[l]->invFFTIInPlace();
716             seisData_[l]->writeFile(fileName, IO::
                PathToInversionResults(), simbox_, sgriLabel);
717             seisData_[l]->endAccess();
718         }
719     }
720 }
721 for (l=0;l<ntheta;l++){
722     if (seisData_[l] != NULL) delete seisData_[l];
723     seisData_[l] = NULL;
724 }
725 delete [] seisData_;
726 seisData_ = NULL;

```

APPENDIX B. CODE

```

727 LogKit::LogFormatted(LogKit::DEBUGLOW, "\nDEALLOCATING: Seismic
728 data\n");
729 if(model_>getVelocityFromInversion() == true) { //Conversion
730     undefined until prediction ready. Complete it.
731     postAlpha_>setAccessMode(FFTGrid::RANDOMACCESS);
732     postAlpha_>expTransf();
733     GridMapping * tdMap = model_>getTimeDepthMapping();
734     const GridMapping * dcMap = model_>getTimeCutMapping();
735     const Simbox * timeSimbox = simbox_;
736     if(dcMap != NULL)
737         timeSimbox = dcMap->getSimbox();
738     tdMap->setMappingFromVelocity(postAlpha_, timeSimbox);
739     postAlpha_>logTransf();
740     postAlpha_>endAccess();
741 }
742
743 //NBNB Anne Randi: Skaler traser ihht notat fra Hugo
744
745 if(model_>getModelSettings()->getUseLocalNoise())
746 {
747     correlations_>invFFT();
748     correlations_>createPostVariances();
749     correlations_>FFT();
750     correctAlphaBetaRho(model_>getModelSettings());
751 }
752
753 if(writePrediction_ == true)
754     ParameterOutput::writeParameters(simbox_, model_, postAlpha_,
755     postBeta_, postRho_,
756     outputGridsElastic_, fileGrid_, -1, false);
757
758 writeBWPredicted();
759
760 Timings::setTimeInversion(wall, cpu);
761 omp_unset_lock(&lock);
762 #ifdef PROFILING
763 stringstream ss;
764 ss << "Seismic inversion [cnxp: ";
765 ss << cnxp;
766 ss << ", nyp: ";
767 ss << nyp_;
768 ss << ", nzp: ";
769 ss << nzp_;
770 ss << "] wallclock time.";
771 wtime = omp_get_wtime() - wtime;
772 NRLib::Prof::setName(ss.str(), INVERSIONLOG);
773 NRLib::Prof::setName("Seismic inversion time reading.",
774     INVERSIONREADLOG);
775 NRLib::Prof::setName("Seismic inversion time writing.",
776     INVERSIONWRITELOG);
777 NRLib::Prof::setName("Seismic inversion inner loop CPU time.",
778     INVCPUTIMELOG);
779 NRLib::Prof::setName("Seismic Inversion loop time.",
780     INVERSIONLOOPLOG);
781 NRLib::Prof::trackTime(wtime, INVERSIONLOG);
782 NRLib::Prof::trackTime(readTimeAccum, INVERSIONREADLOG);
783 NRLib::Prof::trackTime(writeTimeAccum, INVERSIONWRITELOG);
784 NRLib::Prof::trackTime(ptimeAccum, INVCPUTIMELOG);
785 NRLib::Prof::trackTime(invLoop, INVERSIONLOOPLOG);

```

```

782 #endif
783
784     return(0);
785 }
786 /* Below this comment were 1227 lines of code not relevant for
    inversion */

```

Listing B.9: New Seismic Inversion.

```

1  /* Above this comment the original file had ~3000 lines of code not
    related to inversion */
2
3  int
4  Crava::computePostMeanResidAndFFTCov()
5  {
6      Utils::writeHeader("Posterior model / Performing Inversion");
7
8      double wall=0.0, cpu=0.0;
9      TimeKit::getTime(wall,cpu);
10     int i,j,k,l,m;
11
12     fftw_complex * kW          = new fftw_complex[ntheta_];
13
14     fftw_complex * errMult1    = new fftw_complex[ntheta_];
15     fftw_complex * errMult2    = new fftw_complex[ntheta_];
16     fftw_complex * errMult3    = new fftw_complex[ntheta_];
17
18     fftw_complex * ijkData      = new fftw_complex[ntheta_];
19     fftw_complex * ijkDataMean = new fftw_complex[ntheta_];
20     fftw_complex * ijkRes       = new fftw_complex[ntheta_];
21     fftw_complex * ijkMean      = new fftw_complex[3];
22     fftw_complex * ijkAns       = new fftw_complex[3];
23     fftw_complex  kD,kD3;
24     fftw_complex  ijkParLam;
25     fftw_complex  ijkErrLam;
26     fftw_complex  ijkTmp;
27
28     fftw_complex** K = new fftw_complex*[ntheta_];
29     for(i = 0; i < ntheta_; i++)
30         K[i] = new fftw_complex[3];
31
32     fftw_complex** KS = new fftw_complex*[ntheta_];
33     for(i = 0; i < ntheta_; i++)
34         KS[i] = new fftw_complex[3];
35
36     fftw_complex** KScC = new fftw_complex*[3]; // cc - complex
    conjugate (and transposed)
37     for(i = 0; i < 3; i++)
38         KScC[i] = new fftw_complex[ntheta_];
39
40     fftw_complex** parVar = new fftw_complex*[3];
41     for(i = 0; i < 3; i++)
42         parVar[i] = new fftw_complex[3];
43
44     fftw_complex** margVar = new fftw_complex*[ntheta_];
45     for(i = 0; i < ntheta_; i++)
46         margVar[i] = new fftw_complex[ntheta_];
47
48     fftw_complex** errVar = new fftw_complex*[ntheta_];
49     for(i = 0; i < ntheta_; i++)
50         errVar[i] = new fftw_complex[ntheta_];
51

```

APPENDIX B. CODE

```

52 | fftw_complex** reduceVar = new fftw_complex*[3];
53 | for(i = 0; i < 3; i++)
54 |     reduceVar[i] = new fftw_complex[3];
55 |
56 | Wavelet * diff1Operator = new Wavelet(Wavelet::
57 |     FIRSTORDERFORWARDIFF, nz_, nzp_);
58 | Wavelet * diff2Operator = new Wavelet(diff1Operator, Wavelet::
59 |     FIRSTORDERBACKWARDIFF);
60 | Wavelet * diff3Operator = new Wavelet(diff2Operator, Wavelet::
61 |     FIRSTORDERCENTRALDIFF);
62 |
63 | diff1Operator->fft1DInPlace();
64 | delete diff2Operator;
65 | diff3Operator->fft1DInPlace();
66 |
67 | Wavelet ** errorSmooth = new Wavelet*[ntheta_];
68 | Wavelet ** errorSmooth2 = new Wavelet*[ntheta_];
69 | Wavelet ** errorSmooth3 = new Wavelet*[ntheta_];
70 |
71 | int cnxp = nxp_/2+1;
72 |
73 | for(l = 0; l < ntheta_ ; l++)
74 | {
75 |     std::string angle = NRLib::ToString(thetaDeg_[l], 1);
76 |     std::string fileName;
77 |     seisData_[l]->setAccessMode(FFTGrid::READANDWRITE);
78 |     if (seisWavelet_[0]->getDim() == 1) {
79 |         errorSmooth[l] = new Wavelet(seisWavelet_[l], Wavelet::
80 |             FIRSTORDERFORWARDIFF);
81 |         errorSmooth2[l] = new Wavelet(errorSmooth[l], Wavelet::
82 |             FIRSTORDERBACKWARDIFF);
83 |         errorSmooth3[l] = new Wavelet(errorSmooth2[l], Wavelet::
84 |             FIRSTORDERCENTRALDIFF);
85 |         fileName = std::string("ErrorSmooth_") + angle + IO::
86 |             SuffixGeneralData();
87 |         errorSmooth3[l]->printToFile(fileName);
88 |         errorSmooth3[l]->fft1DInPlace();
89 |
90 |         fileName = IO::PrefixWavelet() + angle + IO::
91 |             SuffixGeneralData();
92 |         seisWavelet_[l]->printToFile(fileName);
93 |         seisWavelet_[l]->fft1DInPlace();
94 |
95 |         fileName = std::string("FourierWavelet_") + angle + IO::
96 |             SuffixGeneralData();
97 |         seisWavelet_[l]->printToFile(fileName);
98 |         delete errorSmooth[l];
99 |         delete errorSmooth2[l];
100 |     }
101 |     else { //3D-wavelet
102 |         /* 49 lines of unrellated comments */
103 |     }
104 | }
105 | delete [] errorSmooth;
106 | delete [] errorSmooth2;
107 |
108 | meanAlpha_->setAccessMode(FFTGrid::READANDWRITE); // Note
109 | meanBeta_->setAccessMode(FFTGrid::READANDWRITE); // the top
110 |     five are over written
111 | meanRho_->setAccessMode(FFTGrid::READANDWRITE); // does not
112 |     have the initial meaning.

```

```

103 FFTGrid * parSpatialCorr = correlations_ ->getPostCovAlpha();
104 // NB! Note double usage of postCovAlpha
104 FFTGrid * errCorrUnsmooth = correlations_ ->getPostCovBeta();
105 // NB! Note double usage of postCovBeta
105 FFTGrid * postCovAlpha = correlations_ ->getPostCovAlpha();
106 FFTGrid * postCovBeta = correlations_ ->getPostCovBeta();
107 FFTGrid * postCovRho = correlations_ ->getPostCovRho();
108 FFTGrid * postCrCovAlphaBeta = correlations_ ->
108     getPostCrCovAlphaBeta();
109 FFTGrid * postCrCovAlphaRho = correlations_ ->
109     getPostCrCovAlphaRho();
110 FFTGrid * postCrCovBetaRho = correlations_ ->getPostCrCovBetaRho
110     ();
111 parSpatialCorr ->setAccessMode(FFTGrid::READANDWRITE); //
111     after the processing
112 errCorrUnsmooth ->setAccessMode(FFTGrid::READANDWRITE); //
113 postCovRho ->setAccessMode(FFTGrid::WRITE);
114 postCrCovAlphaBeta ->setAccessMode(FFTGrid::WRITE);
115 postCrCovAlphaRho ->setAccessMode(FFTGrid::WRITE);
116 postCrCovBetaRho ->setAccessMode(FFTGrid::WRITE);
117
118 // Computes the posterior mean first below the covariance is
118     computed
119 // To avoid to many grids in mind at the same time
120 double priorVarVp, justfactor;
121
122 int cholFlag;
123 // long int timestart, timeend;
124 // time(&timestart);
125 float realFrequency;
126
127 LogKit::LogFormatted(LogKit::LOW, "\nBuilding posterior
127     distribution:");
128 float monitorSize = std::max(1.0f, static_cast<float>(nzp_)*0.02f
128     );
129 float nextMonitor = monitorSize;
130 std::cout
131     << "\n 0%      20%      40%      60%      80%      100%"
132     << "\n |      |      |      |      |      |      |"
133     << "\n ^";
134
135 for(k = 0; k < nzp_; k++)
136 {
137     realFrequency = static_cast<float>((nz_*1000.0f)/(simbox_->
137         getLz()*nzp_)*std::min(k, nzp_-k)); // the physical
137         frequency
138     kD = diff1Operator->getCAmp(k); // defines
138         content of kD
139     if (seisWavelet_[0]->getDim() == 1) { //1D-wavelet
140         if (simbox_->getIsConstantThick() == true)
141         {
142             // defines content of K=WDA
143             fillkW(k, errMult1); //
143                 errMult1 used as dummy
144             lib_matrProdScalVecCpx(kD, errMult1, ntheta_); //
144                 errMult1 used as dummy
145             lib_matrProdDiagCpxR(errMult1, A_, ntheta_, 3, K); //
145                 defines content of (WDA) K
146
147             // defines error-term multipliers
148             fillWNorm(k, errMult1, seisWavelet_); //
148                 defines input of (kWNorm) errMult1

```

APPENDIX B. CODE

```

149     fillkWNorm(k, errMult2, errorSmooth3);           //
150         defines input of (kWD3Norm) errMult2
151     lib_matrFillOnesVecCpx(errMult3, ntheta_);       //
152         defines content of errMult3
153 }
154 else //simbox_ ->getIsConstantThick() == false
155 {
156     kD3 = diff3Operator->getCAmp(k);                 // defines kD3
157
158     // defines content of K = DA
159     lib_matrFillValueVecCpx(kD, errMult1, ntheta_); //
160         errMult1 used as dummy
161     lib_matrProdDiagCpxR(errMult1, A_, ntheta_, 3, K); //
162         defines content of ( K = DA )
163
164     // defines error-term multipliers
165     lib_matrFillOnesVecCpx(errMult1, ntheta_);       // defines
166         content of errMult1
167     for(l=0; l < ntheta_; l++)
168         errMult1[l].re /= seisWavelet_[l]->getNorm(); // defines
169         content of errMult1
170
171     lib_matrFillValueVecCpx(kD3, errMult2, ntheta_); // defines
172         content of errMult2
173     for(l=0; l < ntheta_; l++)
174     {
175         errMult2[l].re /= errorSmooth3[l]->getNorm(); // defines
176         content of errMult2
177         errMult2[l].im /= errorSmooth3[l]->getNorm(); // defines
178         content of errMult2
179     }
180     fillInverseAbskWRobust(k, errMult3);             // defines
181         content of errMult3
182 } //simbox_ ->getIsConstantThick()
183 }
184
185 for( j = 0; j < nyp_; j++) {
186     for( i = 0; i < cnxp; i++) {
187         ijkMean[0] = meanAlpha_->getNextComplex();
188         ijkMean[1] = meanBeta_->getNextComplex();
189         ijkMean[2] = meanRho_->getNextComplex();
190
191         for(l = 0; l < ntheta_; l++)
192         {
193             ijkData[l] = seisData_[l]->getNextComplex();
194             ijkRes[l] = ijkData[l];
195         }
196
197         ijkTmp = parSpatialCorr->getNextComplex();
198         ijkParLam.re = float ( sqrt(ijkTmp.re * ijkTmp.re));
199         ijkParLam.im = 0.0;
200
201         for(l = 0; l < 3; l++)
202             for(m = 0; m < 3; m++)
203             {
204                 parVar[l][m].re = parPointCov_[l][m] * ijkParLam.re;
205                 parVar[l][m].im = 0.0;
206                 // if(l!=m)
207                 // parVar[l][m].re *= 0.75; //NBNB OK DEBUG TEST
208             }
209     }
210 }

```



```

201 priorVarVp = parVar[0][0].re;
202 ijkTmp      = errCorrUnsmooth->getNextComplex();
203 ijkErrLam.re = float(sqrt(ijkTmp.re * ijkTmp.re));
204 ijkErrLam.im = 0.0;
205
206 if(realFrequency > lowCut_*simbox_->getMinRelThick() &&
    realFrequency < highCut_) // inverting only relevant
    frequencies
207 {
208     for(l = 0; l < ntheta_; l++)
209         for(m = 0; m < ntheta_; m++)
210             { // Note we multiply kWNorm[l] and comp.conj(
                kWNorm[m]) hence the + and not a minus as in pure
                multiplication
211                 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));
212                 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));
213                 if(l==m) {
214                     errVar[l][m].re += float(wnc_*errThetaCov_[l][m]
                        * errMult3[l].re * errMult3[l].re);
215                     errVar[l][m].im = 0.0;
216                 }
217                 else {
218                     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));
219                     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));
220                 }
221             }
222
223     lib_matrProdCpx(K, parVar, ntheta_, 3, 3, KS);
                // KS is defined here
224     lib_matrProdAdjointCpx(KS, K, ntheta_, 3, ntheta_,
                margVar); // margVar = (K)S(K)' is defined here
225     lib_matrAddMatCpx(errVar, ntheta_, ntheta_, margVar);
                // errVar is added to margVar = (WDA)S(
                WDA)' + errVar
226
227     cholFlag=lib_matrCholCpx(ntheta_, margVar);
                // Choleskey factor of margVar
                is Defined
228
229     if(cholFlag==0)
230     { // then it is ok else posterior is identical to
                prior
231
232         lib_matrAdjoint(KS, ntheta_, 3, KSc);
                // WDASc is adjoint of
                WDAS
233         lib_matrAXeqBMatCpx(ntheta_, margVar, KS, 3);
                // redefines WDAS
234         lib_matrProdCpx(KSc, KS, 3, ntheta_, 3, reduceVar);
                // defines reduceVar

```

APPENDIX B. CODE

```

235 //double hj=1000000.0;
236 //if (reduceVar [0][0].im!=0)
237 // hj = MAXIM(reduceVar [0][0].re/reduceVar [0][0].im
    //,-reduceVar [0][0].re/reduceVar [0][0].im); //
    NBNB DEBUG
238 lib_matrSubtMatCpx (reduceVar ,3 ,3 , parVar );
    // redefines parVar as the
    posterior solution
239
240 lib_matrProdMatVecCpx (K, ijkMean , ntheta_ , 3,
    ijkDataMean); // defines content of
    ijkDataMean
241 lib_matrSubtVecCpx (ijkDataMean , ntheta_ , ijkData);
    // redefines content of ijkData
242
243 lib_matrProdAdjointMatVecCpx (KS, ijkData ,3 ,ntheta_ ,
    ijkAns); // defines ijkAns
244
245 lib_matrAddVecCpx (ijkAns , 3, ijkMean);
    // redefines ijkMean
246 lib_matrProdMatVecCpx (K, ijkMean , ntheta_ , 3,
    ijkData); // redefines ijkData
247 lib_matrSubtVecCpx (ijkData , ntheta_ , ijkRes);
    // redefines ijkRes
248 }
249
250 // quality control DEBUG
251 if (priorVarVp*4 < ijkAns [0].re*ijkAns [0].re + ijkAns
    [0].re*ijkAns [0].re)
252 {
253     justfactor = sqrt (ijkAns [0].re*ijkAns [0].re +
        ijkAns [0].re*ijkAns [0].re)/sqrt (priorVarVp);
254 }
255
256 postAlpha_ ->setNextComplex (ijkMean [0]);
257 postBeta_ ->setNextComplex (ijkMean [1]);
258 postRho_ ->setNextComplex (ijkMean [2]);
259 postCovAlpha->setNextComplex (parVar [0][0]);
260 postCovBeta ->setNextComplex (parVar [1][1]);
261 postCovRho ->setNextComplex (parVar [2][2]);
262 postCrCovAlphaBeta->setNextComplex (parVar [0][1]);
263 postCrCovAlphaRho ->setNextComplex (parVar [0][2]);
264 postCrCovBetaRho ->setNextComplex (parVar [1][2]);
265
266 for (l=0;l<ntheta_ ;l++)
267     seisData_ [l]->setNextComplex (ijkRes [l]);
268 }
269
270 // Log progress
271 if (k+1 >= static_cast<int>(nextMonitor))
272 {
273     nextMonitor += monitorSize;
274     std::cout << "^";
275     fflush (stdout);
276 }
277 }
278 std::cout << "\n";
279
280 // time(&timeend);
281 // LogKit::LogFormatted (LogKit::LOW, "\n Core inversion finished
    after %ld seconds ***\n", timeend-timestart);
282 // these does not have the initial meaning

```

```

283 meanAlpha_ = NULL; // the content is taken care of by
      postAlpha_
284 meanBeta_ = NULL; // the content is taken care of by
      postBeta_
285 meanRho_ = NULL; // the content is taken care of by
      postRho_
286 parSpatialCorr = NULL; // the content is taken care of by
      postCovAlpha
287 errCorrUnsmooth = NULL; // the content is taken care of by
      postCovBeta
288
289 postAlpha_ ->endAccess();
290 postBeta_ ->endAccess();
291 postRho_ ->endAccess();
292
293 postCovAlpha ->endAccess();
294 postCovBeta ->endAccess();
295 postCovRho ->endAccess();
296 postCrCovAlphaBeta ->endAccess();
297 postCrCovAlphaRho ->endAccess();
298 postCrCovBetaRho ->endAccess();
299
300 postAlpha_ ->invFFTInPlace();
301 postBeta_ ->invFFTInPlace();
302 postRho_ ->invFFTInPlace();
303
304 for(l=0;l<ntheta_;l++)
305     seisData_[l] ->endAccess();
306
307 //Finish use of seisData_, since we need the memory.
308 if((outputGridsSeismic_ & IO::RESIDUAL) > 0)
309 {
310     if(simbox_ ->getIsConstantThick() != true)
311         multiplyDataByScaleWaveletAndWriteToFile("residuals");
312     else
313     {
314         for(l=0;l<ntheta_;l++)
315         {
316             std::string angle = NRLib::ToString(thetaDeg_[l],1);
317             std::string sgridLabel = "Residuals for incidence angle "+
                angle;
318             std::string fileName = IO::PrefixResiduals() + angle;
319             seisData_[l] ->setAccessMode(FFTGrid::RANDOMACCESS);
320             seisData_[l] ->invFFTInPlace();
321             seisData_[l] ->writeFile(fileName, IO::
                PathToInversionResults(), simbox_, sgridLabel);
322             seisData_[l] ->endAccess();
323         }
324     }
325 }
326 for(l=0;l<ntheta_;l++)
327     delete seisData_[l];
328 LogKit::LogFormatted(LogKit::DEBUGLOW, "\nDEALLOCATING: Seismic
    data\n");
329
330 if(model_ ->getVelocityFromInversion() == true) { //Conversion
      undefined until prediction ready. Complete it.
331     postAlpha_ ->setAccessMode(FFTGrid::RANDOMACCESS);
332     postAlpha_ ->expTransf();
333     GridMapping * tdMap = model_ ->getTimeDepthMapping();
334     const GridMapping * dcMap = model_ ->getTimeCutMapping();
335     const Simbox * timeSimbox = simbox_;

```

APPENDIX B. CODE

```

336     if(dcMap != NULL)
337         timeSimbox = dcMap->getSimbox();
338
339     tdMap->setMappingFromVelocity(postAlpha_, timeSimbox);
340     postAlpha_->logTransf();
341     postAlpha_->endAccess();
342 }
343
344 //NBNB Anne Randi: Skaler traser ihht notat fra Hugo
345
346 if(model_->getModelSettings()->getUseLocalNoise())
347 {
348     correlations_->invFFT();
349     correlations_->createPostVariances();
350     correlations_->FFT();
351     correctAlphaBetaRho(model_->getModelSettings());
352 }
353
354 if(writePrediction_ == true)
355     ParameterOutput::writeParameters(simbox_, model_, postAlpha_,
356                                     postBeta_, postRho_,
357                                     outputGridsElastic_, fileGrid_
358                                     , -1, false);
359
360 writeBWPredicted();
361
362 delete [] seisData_;
363 delete [] kW;
364 delete [] errMult1;
365 delete [] errMult2;
366 delete [] errMult3;
367 delete [] ijkData;
368 delete [] ijkDataMean;
369 delete [] ijkRes;
370 delete [] ijkMean;
371 delete [] ijkAns;
372 delete [] diff1Operator;
373 delete [] diff3Operator;
374
375 for(i = 0; i < ntheta_; i++)
376 {
377     delete [] K[i];
378     delete [] KS[i];
379     delete [] margVar[i];
380     delete [] errVar[i];
381     delete [] errorSmooth3[i];
382 }
383 delete [] K;
384 delete [] KS;
385 delete [] margVar;
386 delete [] errVar;
387 delete [] errorSmooth3;
388
389 for(i = 0; i < 3; i++)
390 {
391     delete [] KScC[i];
392     delete [] parVar[i];
393     delete [] reduceVar[i];
394 }
395 delete [] KScC;
396 delete [] parVar;
397 delete [] reduceVar;

```

```
396 |  
397 |     Timings::setTimeInversion(wall,cpu);  
398 |     return(0);  
399 | }  
400 | /* Below this comment were 1225 lines of code not rellevant for  
    | inversion */
```

Listing B.10: Old Seismic Inversion.

B.3 FFTW Settings Store

```

1  #ifndef FFTWLOCK_H_
2  #define FFTWLOCK_H_
3
4  #include "src/locks.h"
5  #include "fft/include/fftw.h"
6  #include "fft/include/rfftw.h"
7  #include "nrlib/iotools/logkit.hpp"
8
9  #include <iterator>
10 #include <map>
11 #include <string>
12
13 class FftwSettings{
14     int nx, ny, nz;
15     fftw_direction direction;
16     int *n;
17     int rank;
18     int flags;
19     int dimensions;
20 public:
21
22     FftwSettings():
23     nx(0),
24     ny(0),
25     nz(0),
26     direction(FFTW_FORWARD),
27     n(NULL),
28     rank(0),
29     flags(0),
30     dimensions(0)
31     {}
32
33     FftwSettings(int nx, int ny, int nz, fftw_direction direction,
34                 int flags){
35         this->nx = nx;
36         this->ny = ny;
37         this->nz = nz;
38         this->direction = direction;
39         this->dimensions = 3;
40         this->n = NULL;
41         this->rank = 0;
42         this->flags = flags;
43     }
44
45     FftwSettings(int rank, const int* n, fftw_direction direction,
46                 int flags):
47     nx(0),
48     ny(0),
49     nz(0),
50     direction(direction),
51     n(NULL),
52     rank(rank),
53     flags(flags),
54     dimensions(1){
55         this->n = new int[rank];
56         for(int i = 0; i < rank; i++) this->n[i] = n[i];
57     }
58
59     FftwSettings(const FftwSettings& s){
60         nx = s.nx;

```

```

59     ny = s.ny;
60     nz = s.nz;
61     dimensions = s.dimensions;
62     direction = s.direction;
63     n = new int[s.rank];
64     for(int i = 0; i < s.rank; i++) n[i] = s.n[i];
65     rank = s.rank;
66     flags = s.flags;
67 }
68
69 ~FftwSettings(){
70     if(n != NULL){
71         delete [] n;
72     }
73 }
74
75 std::string toString() const;
76
77 int getNx() const{ return nx; }
78 int getNy() const{ return ny; }
79 int getNz() const{ return nz; }
80 const int* getN() const{ return n; }
81 int getRank() const{ return rank; }
82 int getFlags() const{ return flags; }
83 fftw_direction getDirection() const{ return direction; }
84 int getDimension() const{ return dimensions; }
85
86 friend bool operator<(const FftwSettings& a, const FftwSettings&
87     b){
88     bool result =
89         a.dimensions < b.dimensions ||
90         a.direction < b.direction ||
91         a.flags < b.flags ||
92         a.nx < b.nx ||
93         a.ny < b.ny ||
94         a.nz < b.nz;
95     for(int i = 0; i < a.rank && i < b.rank; i++){
96         result |= (a.n[i] < b.n[i]);
97     }
98     return result;
99 }
100 bool operator()(const FftwSettings* a, const FftwSettings* b)
101     const{
102     return (*a) < (*b);
103 }
104 bool operator()(const FftwSettings& a, const FftwSettings& b)
105     const{
106     return a < b;
107 }
108 };
109
110 class FftwLock{
111     LOCK_T plan_l_;
112     fftwnd_plan plan_;
113 public:
114     FftwLock(){
115         omp_init_lock(&plan_l_);
116         plan_ = NULL;
117     }
118     FftwLock(const FftwSettings& s):

```

APPENDIX B. CODE

```

118     plan_(NULL)
119     {
120         omp_init_lock(&plan_l_);
121         switch(s.getDimension()){
122             case 3:
123                 plan_ = rfftw3d_create_plan(s.getNx(), s.getNy(), s.getNz()
124                     , s.getDirection(), s.getFlags());
125                 break;
126             // case 2:
127             //     plan_ = rfftw2d_create_plan(s.nx, s.ny, s.dir, s.flags);
128             //     break;
129             case 1:
130                 plan_ = rfftwnd_create_plan(s.getRank(), s.getN(), s.
131                     getDirection(), s.getFlags());
132                 break;
133             default:
134                 break;
135         }
136     }
137
138     ~FftwLock(){
139         omp_destroy_lock(&plan_l_);
140         if(plan_ != NULL) fftwnd_destroy_plan(plan_);
141     }
142
143     void lock(){
144         omp_set_lock(&plan_l_);
145     }
146
147     void unlock(){
148         omp_unset_lock(&plan_l_);
149     }
150
151     fftwnd_plan getPlan() const{ return plan_; } //Figure out why
152     //the return type need to be consted.
153 };
154
155 class FftwStore{
156     double wall;
157     std::map<const FftwSettings*, FftwLock*, FftwSettings> rToC;
158     std::map<const FftwSettings*, FftwLock*, FftwSettings> cToR;
159     int cachehits;
160     LOCK_T insert_l_;
161
162     FftwLock* getPlanAndLock(const FftwSettings& s, std::map<const
163         FftwSettings*, FftwLock*, FftwSettings>* m){
164         std::map<const FftwSettings*, FftwLock*, FftwSettings>::
165             iterator it = m->find( &s );
166         if(it == m->end()){
167             FftwLock* l;
168             omp_set_lock(&insert_l_);
169             it = m->find( &s );
170             if(it == m->end()){
171                 const FftwSettings* key = new FftwSettings(s);
172                 l = new FftwLock(s);
173                 //NRLib::LogKit::LogFormatted(NRLib::LogKit::HIGH, "NEW: %s
174                 \n", s.toString().c_str());
175                 if(l->getPlan() != NULL){
176                     m->insert(std::pair<const FftwSettings*, FftwLock*>(key,
177                         l));
178                 }else{

```



```

173     //NRLib::LogKit::LogFormatted(NRLib::LogKit::HIGH, "Failing
        cache");
174     delete l;
175     l = NULL;
176     }
177     }else{
178     //NRLib::LogKit::LogFormatted(NRLib::LogKit::HIGH, "Second:
        %s\n", s.toString().c_str());
179 #pragma omp atomic
180     cachehits += 1;
181     l = it->second;
182     }
183     omp_unset_lock(&insert_l_);
184     return l;
185     }
186 #pragma omp atomic
187     cachehits += 1;
188     return it->second;
189     }
190
191 public:
192     FftwStore();
193
194     ~FftwStore();
195
196     int getCacheHits(){ return cachehits; }
197
198
199     void one_complex_to_real(fftw_complex* cData, fftw_real* rData,
        const FftwSettings& s);
200
201     void one_real_to_complex(fftw_real* rData, fftw_complex* cData,
        const FftwSettings& s);
202
203     int getHits() const{ return cachehits; }
204     int getNumberOfSettings() const{ return rToC.size() + cToR.size()
        ; }
205 };
206
207 FftwStore* getFFTStorage();
208
209 #endif

```

Listing B.11: FFTW setting store definition.

```

1 #include "src/fftwlock.h"
2 #include "src/profiling.h"
3 #include "src/locks.h"
4 #include "nrlib/iotools/logkit.hpp"
5 #include <sstream>
6
7 #define SETTINGSTHRESHOLD 1
8 #define FFTW
9 #define PROFILINGFFTW 1022
10
11 using namespace NRLib;
12 using std::string;
13 using std::stringstream;
14
15 static FftwStore storage;
16
17 string FftwSettings::toString() const{

```

APPENDIX B. CODE

```

18 | stringstream ss;
19 | ss << "[nx: ";
20 | ss << nx;
21 | ss << string(", ny: ");
22 | ss << ny;
23 | ss << string(", nz: ");
24 | ss << nz;
25 | ss << string(", direction: ");
26 | ss << static_cast<int>(direction);
27 | ss << string(", rank: ");
28 | ss << rank;
29 | ss << string(", flags: ");
30 | ss << flags;
31 | ss << string(", dim: ");
32 | ss << dimensions;
33 | ss << "]";
34 | return ss.str();
35 | }
36 |
37 | FftwStore* getFFTStorage(){
38 |     return &storage;
39 | }
40 |
41 |
42 | FftwStore::FftwStore(): cachehits(0)
43 | {
44 |     wall = 0.0;
45 |     omp_init_lock(&insert_l_);
46 | }
47 |
48 | FftwStore::~~FftwStore(){
49 | #pragma omp critical (io)
50 | {
51 | #ifdef PROFILINGFFTW
52 |     LogKit::SetFileLog( string("fftw.log"), PROFILINGFFTW, false);
53 | #endif
54 | #ifndef FFTW
55 |     LogKit::LogMessage(PROFILINGFFTW, "FFTW Lookup\n");
56 | #else
57 |     LogKit::LogMessage(PROFILINGFFTW, "FFTW Dump no reuse of
58 |         fftw_plans \n");
59 | #endif
60 |     LogKit::LogFormatted(PROFILINGFFTW, "FFTW cacheHits: %i\n",
61 |         getHits());
62 |     LogKit::LogFormatted(PROFILINGFFTW, "FFTW settings: %i\n",
63 |         getNumberOfSettings());
64 |     LogKit::LogFormatted(PROFILINGFFTW, "FFTW wall time: %i ms\n",
65 |         static_cast<int>(1000*wall));
66 |     if(getNumberOfSettings() > SETTINGSTHRESHOLD){
67 |         LogKit::LogMessage(PROFILINGFFTW, "FFTW SETTINGS:\n");
68 |         LogKit::LogMessage(PROFILINGFFTW, "FFTW real to complex:\n");
69 |         int cntr = 0;
70 |         for(std::map<const FftwSettings*, FftwLock*, FftwSettings>::
71 |             const_iterator it = rToC.begin(); it != rToC.end(); it++)

```

```

72     for(std::map<const FftwSettings*, FftwLock*, FftwSettings>::
        const_iterator it = cToR.begin(); it != cToR.end(); it++)
        {
73         LogKit::LogFormatted(PROFILINGFFTW, "FFTW\t%s\n", it->first
            ->toString().c_str());
74         cntr++;
75         if(cntr > 20) break;
76     }
77 }
78 #ifndef PROFILINGFFTW
79     LogKit::EndLog();
80 #endif
81 }
82
83     std::map<const FftwSettings*, FftwLock*, FftwSettings>::iterator
        it;
84     for(it = rToC.begin(); it != rToC.end(); it++){
85         delete it->first;
86         delete it->second;
87     }
88     for(it = cToR.begin(); it != cToR.end(); it++){
89         delete it->first;
90         delete it->second;
91     }
92     omp_destroy_lock(&insert_l_);
93 }
94
95 void FftwStore::one_complex_to_real(fftw_complex* cData, fftw_real
        * rData, const FftwSettings& s){
96     double curWall = 0;
97     #pragma omp master
98     curWall = omp_get_wtime();
99
100    #ifndef FFTW
101        FftwLock* fl = NULL;
102        while(fl == NULL) fl = getPlanAndLock(s, &cToR);
103
104        fl->lock();
105    #pragma omp master
106        curWall = omp_get_wtime() - curWall;
107        wall += curWall;
108        rfftwnd_one_complex_to_real((fl->getPlan()), cData, rData);
109    #pragma omp master
110        curWall = omp_get_wtime();
111        fl->unlock();
112    #pragma omp master
113        curWall = omp_get_wtime() - curWall;
114    #else
115        rfftwnd_plan p2;
116        if(s.getDimension() > 1){
117            p2 = rfftw3d_create_plan(s.getNx(), s.getNy(), s.getNz(), s.
                getDirection(), s.getFlags());
118        } else {
119            p2 = rfftwnd_create_plan(1, s.getN(), s.getDirection(), s.
                getFlags());
120        }
121    #pragma omp master
122        curWall = omp_get_wtime() - curWall;
123        wall += curWall;
124        rfftwnd_one_complex_to_real(p2, cData, rData);
125    #pragma omp master
126        curWall = omp_get_wtime();

```

APPENDIX B. CODE

```

127     fftwnd_destroy_plan(p2);
128 #pragma omp master
129     curWall = omp_get_wtime() - curWall;
130 #endif
131     wall += curWall;
132 }
133
134 void FftwStore::one_real_to_complex(fftw_real* rData, fftw_complex
    * cData, const FftwSettings& s){
135     double curWall = 0.0;
136 #pragma omp master
137     curWall = omp_get_wtime();
138 #ifndef FFTW
139
140     FftwLock* fl = NULL;
141     while(fl == NULL) fl = getPlanAndLock(s, &rToC);
142     fl->lock();
143 #pragma omp master
144     curWall = omp_get_wtime() - curWall;
145     wall += curWall;
146     rfftwnd_one_real_to_complex((fl->getPlan()), rData, cData);
147 #pragma omp master
148     curWall = omp_get_wtime();
149     fl->unlock();
150 #pragma omp master
151     curWall = omp_get_wtime() - curWall;
152 #else
153     rfftwnd_plan p1;
154     if(s.getDimension() > 1){
155         p1 = rfftw3d_create_plan(s.getNx(), s.getNy(), s.getNz(), s.
            getDirection(), s.getFlags());
156     }else{
157         p1 = rfftwnd_create_plan(1, s.getN(), s.getDirection(), s.
            getFlags());
158     }
159 #pragma omp master
160     curWall = omp_get_wtime() - curWall;
161     wall += curWall;
162     rfftwnd_one_real_to_complex(p1, rData, cData);
163 #pragma omp master
164     curWall = omp_get_wtime();
165     fftwnd_destroy_plan(p1);
166 #pragma omp master
167     curWall = omp_get_wtime() - curWall;
168 #endif
169     wall += curWall;
170 }

```

Listing B.12: FFTW settings store.

B.4 OpenMP Header

```
1  #ifndef __LOCKS_H__
2  #define __LOCKS_H__
3
4  #ifdef _OPENMP
5      #include <omp.h>
6      #define LOCK_T omp_lock_t
7  #else
8      #define LOCK_T void*
9      #define omp_init_lock(a) 0
10     #define omp_set_lock(a) 0
11     #define omp_unset_lock(a) 0
12     #define omp_destroy_lock(a) 0
13     #define omp_test_lock(a) 0
14     #define omp_get_num_threads() 1
15     #define omp_get_thread_num() 0
16     #define omp_get_max_threads() 1
17     #define omp_get_wtime() 0.0
18 #endif
19
20 static const int BASICOVERHEADLIMIT = 1000;
21 static const int SIMPLIOVERHEADLIMIT = BASICOVERHEADLIMIT/2;
22 static const int MEDIUMOVERHEADLIMIT = BASICOVERHEADLIMIT/4;
23 void init_omp();
24
25 #endif
```

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

Appendix C

Summarizing Poster

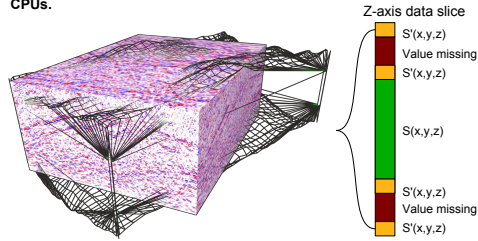
Parallelization Techniques for Seismic Inversion Codes

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

Dept. of Computer and Information 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 V_p (pressure-wave velocity), V_s (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.

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



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

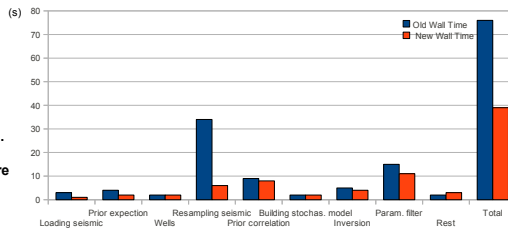
$$S_{x,y,z} = d_{x-1,y,z} \cdot x^2 + d_{x,y-1,z} \cdot x^2 + d_{x,y,z-1} \cdot z^2 + d_{x+1,y,z} \cdot x + d_{x,y+1,z} \cdot y + d_{x,y,z+1} \cdot z + d_{x,y,z}$$

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

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 expectation has gone down from 34 to 9 seconds. Total Walltime went from 76 to 37 seconds.

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

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 dependence on two of three data axes.

- Redundant boundary conditions resulted in unwanted pipeline stalls.

- Data padding was performed by redundant sampling with a cubic fadeoff.

Solution:

- Paralyze independent seismic traces with openMP.

- Recognize boundaries, remove boundary checks inside volume in sample function.

- Add custom cases for padding.



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