Universität Stuttgart

Geodätisches Institut

GOCE data analysis: Optimized brute force solutions of large-scale linear equation systems on parallel computers

Diplomarbeit im Studiengang

Geodäsie und Geoinformatik

an der Universität Stuttgart

Matthias Roth

Stuttgart, September 2010

Prüfer: Prof. Dr.-Ing. Nico Sneeuw
Universität Stuttgart

Betreuer: Dr.-Ing. Oliver Baur
Universität Stuttgart

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum	Unterschrift

The satellite mission GOCE (Gravity field and steady-state Ocean Circulation Explorer) was set up to determine the figure of the Earth with unprecedented accuracy. The sampling frequency is 1 Hz which results in a massive amount of data over the one year period the satellite is intended to be functional. From this data we can setup an overdeterminded linear system of equations to estimate the geopotential coefficients which are required for modelling the Earth's gravity field with spherical harmonics in the desired high resolution.

The linear system of equations is solved "brute-force" which means that the normal equation matrix has to be kept in memory as a whole. The normal equations matrix has a memory demand of up to 65 GByte, hence we need a computer providing a sufficient amount of memory and fast multiple processors for the computations to get them done in a reasonable time.

In this study, a program was written to compute the geopotential coefficients from simulated GOCE data, as GOCE real data were not available yet. As a first step, the program was optimized for the computations to become more efficient. As a second step, the program was parallelized to speed-up the computations by two different techniques: For a first version, the parallelization was done via OpenMP which can be used on shared memory systems which usally have a small number of processors. For the second version, MPI was used which is suited for a distributed memory architecture, hence can incorporate much more processors in the computations.

In summation, we gained a huge boost in efficiency of the program due to the optimization. Furthermore, huge speed-up was gained due to the parallelization. The more processors are incorporated in the computation, the more the overall efficiency drops because of increasing communication between the processors. Here we could show that for huge problems the MPI version is running more efficient than the OpenMP version.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a brief overview of the satellite mission GOCE and the motivation for the usage of super computers. Also different types of computer architectures are introduced as well as the systems used in this study.

Chapter 2 provides the mathematical background for the estimation of spherical harmonic coefficients. In this context, not only the necessary formulas are shown, but also relations which may become useful for a later extension of the program.

Chapter 3 addresses optimizations with respect to data handling and computations. In chapter 4 the parallelization of the program on a shared memory system using OpenMP is shown, whereas chapter 5 deals with the parallelization on a distributed memory system using MPI.

Chapter 6 concludes the work and gives some outline for future enhancement. The source code of the programs is listed in the Appendix.

1.1 The GOCE mission

The satellite mission GOCE [1] (*ESA*, 1999) was set up to measure the gravity field of the Earth from space with unprecedented accuracy. The satellite is in near free fall around the Earth in a low orbit of around 260 km height above ground. The low orbit was chosen that the signal of the Earth's gravity field is as strong as possible in the height of the satellite. In this low height exists still some atmosphere, hence the satellite needs to be propulsed steadily to keep its height. This steady push is produced by ion thrusters which accomplish this task with the necessary precision and small force, and at the same time do not require much fuel .

On board of the satellite is a 3D electrostatic gravity gradiometer mounted near the satellite's centre of mass. On each of the three axes two accelerometers are mounted. Each accelerometer contains a test mass of 320 g in form of a rectangular block with the size of $4\,\mathrm{cm} \times 4\,\mathrm{cm} \times 1\,\mathrm{cm}$ made of a platinum-rhodium-alloy. The test masses are held by electrostatic forces in the centre of the accelerometer. These forces are measured. Translations of the mass as well as rotations are registered. In order to perform high-accurate measurements it is necessary that the temperature is very stable and therefore is regulated very precisely.

The primary goal of the mission is the determination of a high-accurate, high-resolution model of the static Earth's gravity field. This model is deduced from the gradiometer data (Satellite

[1] Gravity field and steady-state Ocean Circulation Explorer

$L_{\max}$	# coefficients	RAM
10	118	0.2 MB
20	438	1.5 MB
50	2 598	51.5 MB
100	10 198	793.5 MB
120	14 638	1 634.8 MB
150	22 798	3 965.4 MB
200	40 398	12 451.2 MB
250	62 998	30 279.1 MB
300	90 598	62 622.1 MB

Table 1.1: *Number of coefficients and amount of memory for normal equations matrix storage dependent on maximum spectral resolution $L_{\max}$.*

Gravity Gradiometry – SGG) and GPS-observed satellite positions (Satellite-Satellite Tracking – SST). The data sampling rate is 1 Hz which results in 86 400 recorded datasets per day, around 2.6 million datasets per month and around 32 million datasets per year. (In this rough calculation it is not considered that there exists resting periods and periods where no data is available due to failure).

From this massive amount of data we can set up an overdetermined linear system of equations to estimate the coefficients which are required for the modelling of the Earth's gravity field with spherical harmonics in the desired high resolution. The number k of coefficients can be computed with

$$k = (L_{\max} + 1)^2 - 3,$$

where $L_{\max}$ is the maximum degree and order of the sherical harmonics. The memory requirement is dominated by the normal equations matrix. The necessary amount of memory can be computed with

$$\mathrm{RAM} = 8\,k^2\,\text{bytes}.$$

Table 1.1 shows the amount of memory dependent on the spectral resolution.

Such a huge amount of data cannot be handled by a desktop computer in a reasonable time. Also a desktop computer does not have the necessary amount of memory available. Therefore, the goal is to develop a program which uses a parallel computer for the computation of the spherical harmonic coefficients. First a version of the program for parallel computer systems with shared memory was implemented. However this program could run on one node of the computer only. To further speed up the program it is necessary to use more processors. Therefore a second version of the program was implemented for systems with distributed memory. A comparison of the perfomance of both program versions is conducted.

1.2 General words on high performance computing

The development of processors evolves rapidly. Within short periods faster and more energy efficient processors are introduced. Following the semi-annual publication of the *top500* list (Meuer et al., 2010), one can see that the development of super computers progresses of the same degree. With each new generation of super computers the computing power increases. Already the petaFLOPS[2]-mark is exceeded. The ratio of computation performance to energy consumption increases steadily, hence also the energy efficiency of the computers rises – but also the total power consumption increases as the demand of computing power grows steadily.

Nowadays most super computers are build from standard components. Standard components are mass produced which makes them cheap in comparison to especially developed components. But there also exist special computer architectures which are designed for special requirements.

There are several possibilities to group computers such as according to different memory concepts:

- shared memory systems,
- distributed memory systems,
- hybrid systems (e. g. cluster computers).

Hybrid systems are a mixture of shared and distributed memory.

1.2.1 Technical terms – processor, CPU, program, process, thread

Although often the terms *CPU*[3] and *processor* are used with the same meaning, they should be distinguished. A CPU can be seen as the "heart" of a computer which executes the instructions and performs integer calculations, whereas a processor can also be used for more general purposes. In this thesis it is not necessary to distinguish between CPUs and processors and also it is not of importance whether the processor cores are on the same chip or not.

Instructions for floating point calculations were added already to the early CPUs. But for scientific purposes these processors were too slow and therefore often special coprocessors for floating point operations were used. As the integration of circuits progressed, former separate parts (like the floating point coprocessor – now called FPU[4]) were moved to the same chip as the CPU to provide even faster execution of the operations. Also further extensions were invented and added to the processors to enhance their computation abilities. Starting with the MMX[5], followed by the SSE[6], those enhancements developed over the years in several versions, and still development progresses. Those extensions add the possibility for vector processing to the processor (but are used for vectorizing calculations of integer numbers mostly).

Altogether those components form the physical processor. Nonetheless, due to historical reasons people still refer to this as CPU (which actually is only part of a processor) while meaning processor. And also nowadays a physical processor (chip) normally holds more than one logical processor. To distinguish between the physical and logical processors the latter are often

[2]FLOPS = FLoating point Operations Per Second (for scientific applications floating point numbers with double precision are taken as a basis). 1 petaFLOPS equals 10^{15} FLOPS.

[3]CPU = Central Processing Unit

[4]FPU = Floating Point Unit

[5]MMX = Multi Media Extension

[6]SSE = Streaming SIMD (Single Instruction Multiple Data) Extensions

refered to as *cores*. In this terminology, one processor can consist of several cores. At the moment, desktop computers typically have 2–4 cores per processor (up to 6 cores for high end desktop computers).

For the terms *program*, *process* and *thread* the same distinguishing problem arises. In the past, as programs run on a single processor successively, a running program was called process. With the introduction of parallel computing things changed: a program can now be divided into several processes which in turn can run on several processors. Each process can be seen as an individual instance of a program with its own exclusively used memory area which is protected from the access of other processes. Threads are on the first sight the same as processes – often a thread is called a *lightweight process* – but threads are only sub instances of processes. One process can consist of many threads which have common access to the memory area of this process (Tanenbaum, 2001).

1.2.2 Shared memory systems

As the name already implies, on a shared memory system all CPUs have access to the same memory. This memory concept is quite common nowadays because many personal computers are equipped with multi core processors which usually share the whole installed memory.

This concept permits simple implementation of a multi-threaded program, which is due to simple communication between the individual CPUs. Messages are written in the memory where other processes can read them. On the other hand, there need to be effective mechanisms to protect the data in memory of the different processes against each other. When all processes are part of the same program, there needs to be a mechanism to protect data beeing overwritten by one process before another process is able to read it. The needed administrative expense grows the more processors are added to work on the same computation task. Depending on the type of task, at a certain point the administrative expense grows so much when one processor is added that the computing power provided by this processor is almost completely needed for the administrative workload. However, this type of system can be used in a very efficient way – especially because of its simplicity.

There exist different approaches how to connect the CPUs with the memory:

Bus system The most common system is the bus system because it is used in standard personal computers. All CPUs are connected via a common bus[7] to the memory (see figure 1.1).

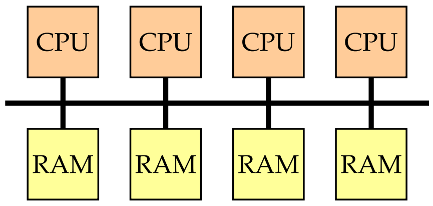

Figure 1.1: *Shared memory system – connection via a bus.*

[7]As the vehicle from lat. *omnibus* = for all.

The biggest advantage of the bus system is that it can be implemented without much design costs and that it can be extended easily by connecting further CPUs or other memory units to the bus.

The biggest disadvantage is that all CPUs have to share the bandwidth of the bus. Depending on the implementation it can also mean that only one CPU has access to the memory while at the same time the other CPUs have to wait.

Crossbar switch To get rid of the problems of the bus system, the crossbar switch was developed. With a crossbar switch each CPU has direct access to each memory unit (see figure 1.2).

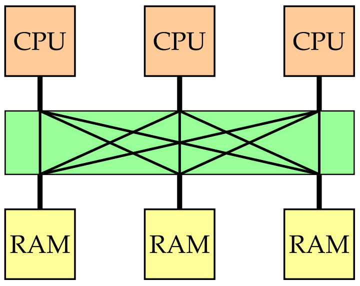

Figure 1.2: *Shared memory system – connection via crossbar switch.*

The crossbar switch makes it possible that each CPU can use the full bandwidth for memory access. Also possible waiting time decreases as one CPU can still access some memory banks when another CPU accesses other ones.

The disadvantage of using a crossbar switch is that the complexity of the circuits rises exponentially the more CPUs or memory banks are connected. There needs to be a direct connection from each CPU to every memory bank. Hence this type of memory connection is only suited for a very low number of CPUs.

1.2.3 Distributed memory systems

The most significant disadvantage of systems with shared memory is that these systems cannot be expanded arbitrarily. Such a disadvantage is nonexistant with distributed memory systems. On such systems each CPU has its own memory; memory access is restricted to the particular CPU.

This combination of CPU and memory is referred to as *node*. Communication between CPUs normally takes place over slow networks (see figure 1.3) – compared to the bandwidth of the communication with the memory. Distributed memory systems can be expanded nearly in any order. For example there exist systems with more than 100 000 CPUs. For such systems the term *massively parallel processing* exists.

On the other hand, a bottleneck is caused from the communication over the network. A network usually has longer latency than a memory bus. Before communcations can be established, a process has to wait for a bit because of technical reasons, hence the process is slowed down.

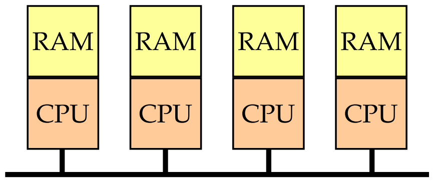

Figure 1.3: *Distributed memory system.*

At the same time the complexity of the programs gets higher as programmers have to tell the processes which of them should communicate with each other to send or receive data.

Parts of the program should be arranged in such a way that the amount of communication between CPUs is as low as possible.

1.2.4 Hybrid systems – cluster computers

The motivation for hybrid systems is to combine the advantages of shared memory systems and distributed memory systems. At first glance, hybrid systems look a lot like distributed memory systems. The difference lies in the computing nodes which work like a shared memory system each. Each node consists of several CPUs which share the node's memory. The computing nodes are connenected to each other via network which is usually much slower than the CPU-memory connection within a nodes (see figure 1.4).

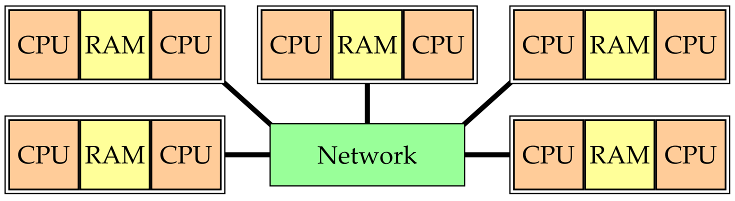

Figure 1.4: *Hybrid system – shared memory nodes are put together to form a cluster.*

Presently, the most common architecture to build super computers is such a hybrid one. Computers built in such way are usually called *cluster computers*.

There are big advantages: building such a computer is cheap as one can use standard components which are cheap due to mass production. Using standard components also makes it easy to exchange a defective node. Also the computers are easily extensible by adding more nodes, and the nodes can be exchanged by updated hardware.

Like distributed memory systems, a hybrid system has the disadvantages resulting from the relatively slow communication between nodes. This can be countered by using faster networking techniques. A further possibility is the consideration of the slow network connection while programming. When computational intensive parts of a program favor the local memory over the distributed memory the cost caused by the networking activity can be reduced considerably. Hence, the program does the computations in less time.

The range of different systems is huge – starting from low-cost selfmade systems of a few PCs connected by an Ethernet network, to high-end systems consisting of thousands of CPUs with the components packed tightly in so called *racks* and connected through special networking devices which enable fast communication.

1.3 Systems used for this study

Different computer systems were used for the implementation and testing of both versions of the program. These systems are presented next.

1.3.1 Systems for programming and simple testing

Local computer. As *local computer* I will refer to the desktop computer which was used to do the programming and also was sufficient for short tests which did not require much memory. The advantage of this computer is that new libraries could be installed and tested with different setting ad libitum.

This computer has a dual core processor. Hence, it was possible to write and test the shared memory version of the program. Further hardware characteristics are presented in table 1.2.

	local
type of processors	AMD Athlon 64 X2 4200+
# processors	1 (2 cores)
clock frequency	2.2 GHz
memory	1 GByte
operating system	Linux (SuSE)

Table 1.2: *Hardware characteristics of the local computer that was used for programming and testing.*

GIS-Cluster. A small cluster computer consisting of disused desktop computers was built to test the distributed memory version of the program. The different reasons for the setup of the cluster and how to proceed to build a cluster will be explained in section 5.2.1 in detail. The term *GIS* stands for "Geodätisches Institut Stuttgart".

The cluster consists of two desktop computers as this is enough for testing purposes. The specifications of the used computers are summarized in table 1.3. The frontend connects the cluster to the university network. It is also used as computing node.

1.3.2 High performance computers

Several high performance computing platforms are supported by the HLRS[8]. For this study, two systems were used: the SX-8 and its succeeding model, the SX-9. Both SX systems are built by NEC.

[8]HLRS = High Performance Computing Center Stuttgart (Höchstleistungsrechenzentrum Stuttgart).

	frontend	computing node
type of processors	Intel Pentium 4	
# processors	1 (1 core)	1 (1 core)
clock frequency	3.4 GHz	3.0 GHz
memory	1 GByte	1 GByte
operating system	Linux (Rocks Clusters)	

Table 1.3: *Hardware characteristics of the cluster that was used for testing the distributed memory version.*

Access to the SX systems is possible via their respective frontends. The frontends are used for cross-compiling programs for the SX systems as well as for the interaction with the batch system. The procedure for running a job on one of the SX systems is to submit a batch script to its scheduler. In this batch script the demands (amount of memory, number of nodes, number of processors, running time etc.) for the job as well as the program calls are defined. Depending on these demands the scheduler administers the jobs and executes them when the appropriate resources on the SX system are available.

The SX systems are so-called vector computers. The processors are no standard components but especially produced for the SX systems. Each processor has a vector unit which is capable for executing a floating point operation on multiple data. For scalar operations the processor also has a scalar unit which runs at half the clock speed of the vector unit.

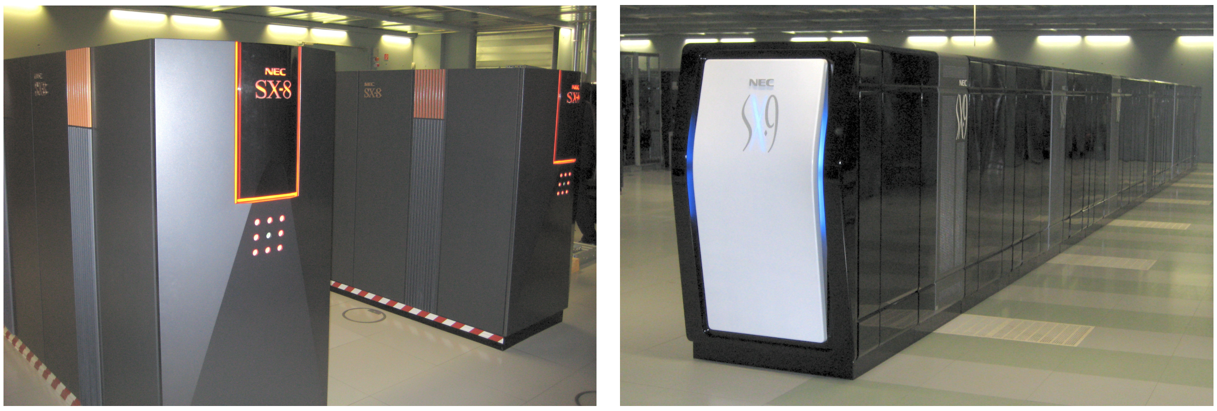

Figure 1.5: *HLRS NEC SX systems (left: SX-8, right: SX-9).*

NEC SX-8 and its frontends The SX-8 has two frontends called Asama[9] and A1. Their characteristics can be found in table 1.4. A picture of the SX-8 is displayed in figure 1.5. The vector unit of the SX-8 processors has four vector pipes, hence can handle four floating point numbers in parallel with a single instruction. The specifications of the SX-8 are listed in table 1.5.

NEC SX-9 and its frontends The SX-9 system, too, has two frontends. Accessible for us is only the one called Ontake[10]. Its specifications can be found in table 1.4. A picture of the SX-9

[9]Named after *Asama* (jap. 浅間山), a 2542 m high volcano on the main island Honshū, Nagano prefecture.
[10]Named after *Ontake* (jap. 御嶽山), with 3067 m the second highest volcano of Japan, located on Honshū, border of the prefectures Nagano and Gunma.

is shown in figure 1.5. The vector unit of the SX-9 processors has eight vector pipes, hence can handle eight floating point numbers in parallel with a single instruction. The specifications of the SX-9 can be found in table 1.5.

	SX-8 frontends		SX-9 frontend
	Asama	A1	Ontake
type of processors	Intel Itanium 2 (IA-64)		Intel XEON X7370 (x64)
# processors	32 (1 core)	32 (1 core)	4 (4 core)
clock frequency	1.5 GHz	1.5 GHz	2.93 GHz
memory	256 GByte	512 GByte	128 GByte
operating system	Linux (SuSE)	Linux (SuSE)	Linux (SuSE)

Table 1.4: *Hardware characteristics of the NEC SX frontends.*

	NEC SX-8	NEC SX-9
type of processors	NEC SX-8	NEC SX-9
# processors per node	8	16
# nodes	10	12
# processors	80	192
clock frequency	2 GHz (vector), 1 GHz (scalar)	3.2 GHz (vector), 1.6 GHz (scalar)
# vector pipes	4	8
memory per node	128 GByte	512 GByte
total memory	1.28 TByte	6 TByte
node-node interconnect	IXS 8 GByte/s per node	IXS 32 GByte/s per node
operating system	SUPER-UX	SUPER-UX
peak performance	1.2 TFLOPS	19.2 TFLOPS

Table 1.5: *Hardware characteristics of the NEC SX systems.*

Chapter 2

Methodology of gravity field recovery

2.1 From gravitational potential to gravitational gradients

In terms of spherical harmonics, the Earth's gravitational potential reads

$$V(\lambda, \varphi, r) = \frac{GM}{R} \sum_{l=0}^{\infty} \sum_{m=0}^{l} \left(\frac{R}{r}\right)^{l+1} \bar{P}_{lm}(\sin\varphi)\,(\bar{c}_{lm}\cos m\lambda + \bar{s}_{lm}\sin m\lambda) \tag{2.1}$$

(Heiskanen and Moritz, 1967). In this equation (λ, φ, r) are the spherical polar coordinates with eastern longitude λ, northern latitude φ and the distance from the centre of the Earth r. GM is the geocentric constant, R is the semi-major axis of a reference ellipsoid of revolution. Further, the latitude dependent functions $\bar{P}_{lm}(\sin\varphi)$ are the 4π-normalized Legendre functions of the first kind. The coefficients $\bar{c}_{lm}$ and $\bar{s}_{lm}$ are unknown geopotential parameters. $L = l_{\max}$ is the maximum spherical harmonic degree.

In the following, starting from equation 2.1 gravitational gradients will be derived. The derivation is according to Baur (2007).

Before proceeding, we substitute $\bar{P}_{lm}(\sin\varphi)(\bar{c}\cos m\lambda + \bar{s}\sin m\lambda)$ in order to save space during the derivations later on. The substitutions are as follows:

$$e_{lm}(\lambda, \varphi) := \bar{P}_{lm}(\sin\varphi) \begin{cases} \cos m\lambda & m \geq 0 \\ \sin m\lambda & m < 0 \end{cases}$$

are called the surface spherical harmonics and

$$\bar{v}_{lm} := \begin{cases} \bar{c}_{lm} & m \geq 0 \\ \bar{s}_{lm} & m < 0 \end{cases}$$

are the coefficients. The lower bound of the summation over n has to be changed to accommodate the substitution. With this, equation 2.1 reads

$$V(\lambda, \varphi, r) = \frac{GM}{R} \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \left(\frac{R}{r}\right)^{l+1} e_{lm}(\lambda, \varphi)\bar{v}_{lm}\,. \tag{2.2}$$

2.1.1 Base vectors and their derivatives

Any point $\mathbf{x}$ given in polar coordinates on a sphere can be expressed in its cartesian counterpart

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} r\cos\varphi\cos\lambda \\ r\cos\varphi\sin\lambda \\ r\sin\varphi \end{pmatrix}$$

From this equations, the unit vectors $\mathbf{e}_\lambda$, $\mathbf{e}_\varphi$ and $\mathbf{e}_r$, which form the local north oriented coordinate frame as outlined in figure 2.1, can be obtained. To do this we first define

$$\mathbf{f}_i = \left(\frac{\partial x_1}{\partial i}\mathbf{e}_1 + \frac{\partial x_2}{\partial i}\mathbf{e}_2 + \frac{\partial x_3}{\partial i}\mathbf{e}_3\right), \quad \text{with} \quad i = \lambda, \varphi, r$$

and we get the unit vectors

$$\mathbf{e}_\lambda = \frac{\mathbf{f}_\lambda}{\|\mathbf{f}_\lambda\|} = (-\sin\lambda\,\mathbf{e}_1 + \cos\lambda\,\mathbf{e}_2 + 0\,\mathbf{e}_3)\ , \tag{2.3}$$

$$\mathbf{e}_\varphi = \frac{\mathbf{f}_\varphi}{\|\mathbf{f}_\varphi\|} = (-\sin\varphi\cos\lambda\,\mathbf{e}_1 - \sin\varphi\sin\lambda\,\mathbf{e}_2 + \cos\varphi\,\mathbf{e}_3)\ , \tag{2.4}$$

$$\mathbf{e}_r = \frac{\mathbf{f}_r}{\|\mathbf{f}_r\|} = (\cos\varphi\cos\lambda\,\mathbf{e}_1 + \cos\varphi\sin\lambda\,\mathbf{e}_2 + \sin\varphi\,\mathbf{e}_3)\ . \tag{2.5}$$

Later, the partial derivations of the unit vectors are needed (equations 2.3 to 2.5). The most difficult to derive is $\frac{\partial}{\partial\lambda}\mathbf{e}_\lambda$, as one needs to extend the equation with $\cos^2\varphi + \sin^2\varphi = 1$ as follows

$$\begin{aligned}
\frac{\partial}{\partial\lambda}\mathbf{e}_\lambda &= \frac{\partial}{\partial\lambda}(-\sin\lambda\mathbf{e}_1 + \cos\lambda\mathbf{e}_2) \\
&= -\cos\lambda\mathbf{e}_1 - \sin\lambda\mathbf{e}_2 \\
&= -\cos\lambda\left(\cos^2\varphi + \sin^2\varphi\right)\mathbf{e}_1 - \sin\lambda\left(\cos^2\varphi + \sin^2\varphi\right)\mathbf{e}_2 \\
&= -\cos\varphi\mathbf{e}_r + \sin\varphi\mathbf{e}_\varphi
\end{aligned}$$

The other partial derivations of the unit vectors are more easy to derive, so only the results are given. Here all of them are listed:

$$\begin{aligned}
&\frac{\partial}{\partial\lambda}\mathbf{e}_\lambda = -\cos\varphi\mathbf{e}_r + \sin\varphi\mathbf{e}_\varphi && \frac{\partial}{\partial\lambda}\mathbf{e}_\varphi = -\sin\varphi\mathbf{e}_\lambda && \frac{\partial}{\partial\lambda}\mathbf{e}_r = \cos\varphi\mathbf{e}_\lambda \\
&\frac{\partial}{\partial\varphi}\mathbf{e}_\lambda = 0 && \frac{\partial}{\partial\varphi}\mathbf{e}_\varphi = -\mathbf{e}_r && \frac{\partial}{\partial\varphi}\mathbf{e}_r = \mathbf{e}_\varphi \\
&\frac{\partial}{\partial r}\mathbf{e}_\lambda = 0 && \frac{\partial}{\partial r}\mathbf{e}_\varphi = 0 && \frac{\partial}{\partial r}\mathbf{e}_r = 0
\end{aligned}$$

2.1.2 Gravitational acceleration vector

In Baur (2007) it is shown that the gradient operator for an equation given in polar coordinates reads

$$\text{grad} = \left(\mathbf{e}_\lambda\frac{1}{r\cos\varphi}\frac{\partial}{\partial\lambda} + \mathbf{e}_\varphi\frac{1}{r}\frac{\partial}{\partial\varphi} + \mathbf{e}_r\frac{\partial}{\partial r}\right). \tag{2.6}$$

Now we are able to compute the gravitational acceleration vector according to

$$\operatorname{grad} V = \left(\mathbf{e}_\lambda \frac{1}{r\cos\varphi}\frac{\partial V}{\partial\lambda} + \mathbf{e}_\varphi \frac{1}{r}\frac{\partial V}{\partial\varphi} + \mathbf{e}_r \frac{\partial V}{\partial r}\right) = \frac{GM}{R^2}\sum_{l=0}^{\infty}\sum_{m=-l}^{l}\left(\frac{R}{r}\right)^{l+2}\Bigg\{ \tag{2.7}$$

$$\mathbf{e}_\lambda \frac{1}{\cos\varphi}\frac{\partial}{\partial\lambda} e_{lm}(\lambda,\varphi)\bar{v}_{lm} + \mathbf{e}_\varphi \frac{\partial}{\partial\varphi} e_{lm}(\lambda,\varphi)\bar{v}_{lm} - \mathbf{e}_r (l+1) e_{lm}(\lambda,\varphi)\bar{v}_{lm}\Bigg\},$$

which commonly is represented with

$$\operatorname{grad} V = \begin{pmatrix} V_\lambda \\ V_\varphi \\ V_r \end{pmatrix}.$$

2.1.3 Gravitational gradients

Now we have all parts together to obtain the gravitational gradient tensor

$$\operatorname{grad}(\operatorname{grad} V) = \left(\mathbf{e}_\lambda \frac{1}{r\cos\varphi}\frac{\partial}{\partial\lambda} + \mathbf{e}_\varphi \frac{1}{r}\frac{\partial}{\partial\varphi} + \mathbf{e}_r \frac{\partial}{\partial r}\right)\Bigg[\frac{GM}{R^2}\sum_{l=0}^{\infty}\sum_{m=-l}^{l}\left(\frac{R}{r}\right)^{l+2}\Bigg\{$$

$$\mathbf{e}_\lambda \frac{1}{\cos\varphi}\frac{\partial}{\partial\lambda} e_{lm}(\lambda,\varphi)\bar{v}_{lm} + \mathbf{e}_\varphi \frac{\partial}{\partial\varphi} e_{lm}(\lambda,\varphi)\bar{v}_{lm} - \mathbf{e}_r (l+1) e_{lm}(\lambda,\varphi)\bar{v}_{lm}\Bigg\}\Bigg]. \tag{2.8}$$

We get

$$\begin{aligned}
\operatorname{grad}(\operatorname{grad} V) = & \frac{GM}{R^3}\sum_{l=0}^{\infty}\sum_{m=-l}^{l}\left(\frac{R}{r}\right)^{l+3}\Bigg\{ \qquad\qquad (2.9)\\
& \mathbf{e}_\lambda\otimes\mathbf{e}_\lambda\left(\frac{1}{\cos^2\varphi}\frac{\partial^2}{\partial\lambda^2} e_{lm}(\lambda,\varphi)\bar{v}_{lm} - \tan\varphi\frac{\partial}{\partial\varphi} e_{lm}(\lambda,\varphi)\bar{v}_{lm} - (l+1)e_{lm}(\lambda,\varphi)\bar{v}_{lm}\right)\\
& + \mathbf{e}_\lambda\otimes\mathbf{e}_\varphi\left(\frac{1}{\cos\varphi}\frac{\partial^2}{\partial\lambda\partial\varphi} e_{lm}(\lambda,\varphi)\bar{v}_{lm} + \frac{\tan\varphi}{\cos\varphi}\frac{\partial}{\partial\lambda} e_{lm}(\lambda,\varphi)\bar{v}_{lm}\right)\\
& + \mathbf{e}_\lambda\otimes\mathbf{e}_r\left(-\frac{l+2}{\cos\varphi}\frac{\partial}{\partial\lambda\partial\varphi} e_{lm}(\lambda,\varphi)\bar{v}_{lm}\right)\\
& + \mathbf{e}_\varphi\otimes\mathbf{e}_\lambda\left(\frac{1}{\cos\varphi}\frac{\partial^2}{\partial\lambda\partial\varphi} e_{lm}(\lambda,\varphi)\bar{v}_{lm} + \frac{\tan\varphi}{\cos\varphi}\frac{\partial}{\partial\lambda} e_{lm}(\lambda,\varphi)\bar{v}_{lm}\right)\\
& + \mathbf{e}_\varphi\otimes\mathbf{e}_\varphi\left(\frac{\partial^2}{\partial\varphi^2} e_{lm}(\lambda,\varphi)\bar{v}_{lm} - (l+1)e_{lm}(\lambda,\varphi)\bar{v}_{lm}\right)\\
& + \mathbf{e}_\varphi\otimes\mathbf{e}_r\left(-(l+2)\frac{\partial}{\partial\varphi} e_{lm}(\lambda,\varphi)\bar{v}_{lm}\right)\\
& + \mathbf{e}_r\otimes\mathbf{e}_\lambda\left(-\frac{l+2}{\cos\varphi}\frac{\partial}{\partial\lambda\partial\varphi} e_{lm}(\lambda,\varphi)\bar{v}_{lm}\right)\\
& + \mathbf{e}_r\otimes\mathbf{e}_\varphi\left(-(l+2)\frac{\partial}{\partial\varphi} e_{lm}(\lambda,\varphi)\bar{v}_{lm}\right)\\
& + \mathbf{e}_r\otimes\mathbf{e}_r\left((l+2)(l+1)e_{lm}(\lambda,\varphi)\bar{v}_{lm}\right)\Bigg\}.
\end{aligned}$$

These terms usually are summarized in the 3×3 array

$$\operatorname{grad}(\operatorname{grad} V) = \begin{pmatrix} V_{\lambda\lambda} & V_{\lambda\varphi} & V_{\lambda r} \\ V_{\varphi\lambda} & V_{\varphi\varphi} & V_{\varphi r} \\ V_{r\lambda} & V_{r\varphi} & V_{rr} \end{pmatrix}.$$

In equation 2.9 it can be seen that

$$V_{\lambda\varphi} = V_{\varphi\lambda}, \quad V_{\lambda r} = V_{r\lambda}, \quad V_{\varphi r} = V_{r\varphi}$$

holds true. Hence the matrix is symmetric. Furthermore

$$V_{\lambda\lambda} + V_{\varphi\varphi} + V_{rr} = 0$$

holds true (Laplace condition).

Of course, in practical use the spectral resolution depends on the data sensitivity. Hence, the series in equation 2.9 is truncated at $L = l_{\max}$. For the GOCE mission, $l_{\max} = 250$ is expected. In this thesis only the radial element of the gravitational tensor, V_{rr}, is used. (Nonetheless the program was prepared to incorporate the other terms easily). The radial component reads

$$V_{rr}(\lambda, \varphi, r) = \frac{GM}{R^3} \sum_{l=0}^{L} \sum_{m=0}^{l} \left(\frac{R}{r}\right)^{l+3} (l+1)(l+2)\bar{P}_{lm}(\sin\varphi)\,[\bar{c}_{lm}\cos m\lambda + \bar{s}_{lm}\sin m\lambda]\,. \qquad (2.10)$$

2.2 Coordinate frames

For the analysis of GOCE data, several coordinate frames must be considered (*EGG-C*, 2009). The origin of the Local Orbit Reference Frame (LORF) lies in the nominal mass centre of the satellite GOCE, axis $\mathbf{e}_x$ is pointing in flight direction (along-track), axis $\mathbf{e}_z$ is pointing away from the Earth centre in near radial direction (quasi-radial) and axis $\mathbf{e}_y$ completes the orthogonal, right-handed reference frame (cross-track). The gradiometer is mounted near the mass centre of GOCE and has its own Gradiometer Reference Frame (GRF) in which the gravity gradients are provided. Origin and axes of the GRF are according to the axes of the gradiometer. Ideally, LORF and GRF would coincide. However, in reality the two systems deviate from each other within a few degrees. In this study, for simplicity we presume that LORF and GRF are identically.

The position of the satellite is given with respect to the Earth-Fixed Reference Frame (EFRF) with its origin in the geocenter, axis $\mathbf{e}_1$ piercing the intersection of the equator and the Greenwich meridian, axis $\mathbf{e}_3$ piercing the north pole and axis $\mathbf{e}_2$ to complete the orthogonal, right-handed reference frame. And finally the spherical harmonics are evaluated in the Local North Oriented Frame (LNOF) with axis $\mathbf{e}_\varphi$ pointing to the North, axis $\mathbf{e}_\lambda$ pointing to the East and axis $\mathbf{e}_r$ is radial. The different coordinate systems are outlined in figure 2.1.

To do the necessary coordinate transformations, quaternions are provided with the GOCE real data. These quaternions describe the rotation of coordinates from the Inertial Reference Frame (IRF) to the GRF and also from the EFRF to the IRF. Further the spherical coordinates of the satellite are given, so that also the position with respect to the LNOF can be obtained.

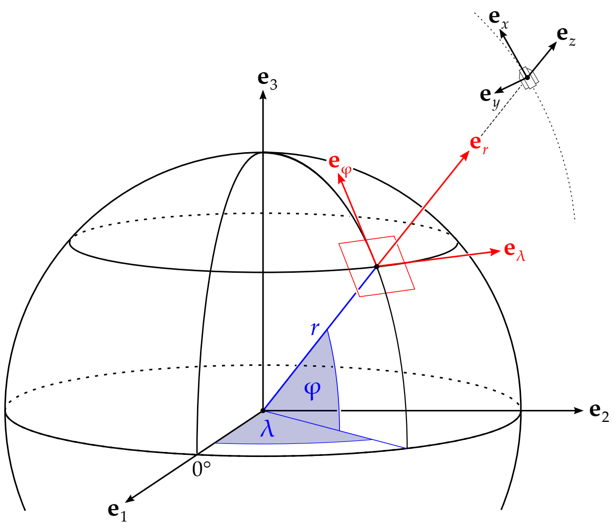

Figure 2.1: *Reference frames: Earth Fixed Reference Frame* ($\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$)*, Local North Oriented Frame* ($\mathbf{e}_\lambda$, $\mathbf{e}_\varphi$, $\mathbf{e}_r$) *and Local Orbit Reference Frame* ($\mathbf{e}_x$, $\mathbf{e}_y$, $\mathbf{e}_z$).

The simulated GOCE data, the program works with, is already given in the proper coordinate frame. Hence, these transformations are not necessary for the simulated data. This section serves as a reminder when later on the GOCE real data will be preprocessed for the use with the program as one has to consider these coordinate transformations at that time.

2.3 Estimation of geopotential parameters

There are expected around $n = 30$ million observations from GOCE for one year, and it is hoped that more data will be gathered as the fuel consumption of the satellite is less than expected. Hopefully no other problems occur like the failing of the main computer in February 2010 and the failing of the backup computer in July 2010. Luckily the experts from ESA were able to solve the problem with a software patch which bypasses the failed chip (*European Space Agency*, 2010).

As we want to estimate up to $u = 90\,598$ unknown coefficients for $l_{\max} = 300$ we have an overestimated system of equations. Hence we can make use of a least squares adjustment to calculate the unknown parameters.

The observation equations can be written in matrix form as

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{e} \tag{2.11}$$

where we have the observations $\mathbf{y}$ which consist of the product between the design matrix $\mathbf{A}$ and the coefficients $\mathbf{x}$ as well as the added residual errors $\mathbf{e}$. The design matrix is filled for each observation with one line of the partial derivatives of equation 2.10 with respect to the coefficients $\bar{c}_{lm}$ and $\bar{s}_{lm}$

$$\left.\begin{array}{l} \frac{\partial V_{rr}}{\partial \bar{c}_{lm}} \\ \frac{\partial V_{rr}}{\partial \bar{s}_{lm}} \end{array}\right\} = \frac{GM}{R^3} \sum_{l=0}^{L} \sum_{m=0}^{l} \left(\frac{R}{r}\right)^{l+3} (l+1)(l+2)\bar{P}_{lm}(\sin\varphi) \left\{ \begin{array}{l} \cos m\lambda \\ \sin m\lambda \end{array}\right. . \tag{2.12}$$

The minimization of the square sum of the residuals

$$\min_x \|\mathbf{e}\|_2 = \min_x \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2$$

results in the least squares estimate of the coefficients

$$\hat{\mathbf{x}} = \left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)^{-1}\mathbf{A}^\mathsf{T}\mathbf{y} = \mathbf{N}^{-1}\mathbf{b} \tag{2.13}$$

where $\mathbf{N} = \mathbf{A}^\mathsf{T}\mathbf{A}$ is called the normal equation matrix and $\mathbf{b}$ is defined as $\mathbf{b} = \mathbf{A}^\mathsf{T}\mathbf{y}$.

There exist other possibilities to solve the least squares problem, for example with iterative solvers which do not need the explicit assembly of the normal equations matrix and hence do not have the need for very much memory. But in our case we want the possibility to get accuracy information of the parameter estimation. For this we need the normal equations matrix $\mathbf{N}$ explicitly, so that we are able to obtain the variance-covariance matrix of the parameter estimate

$$D(\hat{\mathbf{x}}) = \hat{\sigma}^2\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)^{-1} = \hat{\sigma}^2\mathbf{N}^{-1}, \tag{2.14}$$

$$\hat{\sigma}^2 = \frac{\hat{\mathbf{e}}^\mathsf{T}\hat{\mathbf{e}}}{n-u}, \qquad \hat{\mathbf{e}} = \left(\mathbf{A}\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)^{-1}\mathbf{A}^\mathsf{T} - \mathbf{I}\right)\mathbf{y}. \tag{2.15}$$

Chapter 3

Computational optimization

In this chapter, the efforts which have been done to improve the efficency of the implementations in general are presented. For the linear algebra part of the program it is convenient to have with the BLAS[1] and LAPACK[2] two powerful numerical libraries at hand. Those libraries contain nearly all tools one needs for computing in the realms of linear algebra. For many computer architectures specially optimized versions are available. These versions usually get the best performance out of a computer they are designed for and accelerate the computation significantly in comparison to the standard version.

For several architectures (e. g. x86 and x86-64) ATLAS[3] is available. We also used this bundle of libraries on the local computer. ATLAS contains a wide range of BLAS and LAPACK functions which are compiled with specific settings according to the computer architecture. The architecture is determined automatically during the installation.

The specific optimizations and changes for the different approaches to parallelize the program are shown in the chapters on OpenMP and MPI (chapters 4 and 5).

3.1 Data – format and input

The simulated GOCE data are provided in ASCII format. These data are organized in two files. The first file contains the satellite-satellite tracking (SST) information (satellite positions, kinematic orbit) with respect to the LNOF. In particular the data are organized as one line per dataset with the entries printed in table 3.1. The second file contains the satellite gravity gradiometry (SGG) information. Again the data is organized as one line per dataset, the entries are listed in table 3.2.

t_{SST}	time
x_1, x_2, x_3	position [m]
$\dot{x}_1, \dot{x}_2, \dot{x}_3$	velocity [m/s]
$\ddot{x}_1, \ddot{x}_2, \ddot{x}_3$	acceleration [m/s^2]
λ, φ, r	spherical coordinates [rad, rad, m]

Table 3.1: *SST data format (simulated data).*

[1]BLAS = Basic Linear Algebra Subprograms

[2]LAPACK = Linear Algebra PACKage

[3]ATLAS = Automatically Tuned Linear Algebra Software

t_{SGG}	time
$T_{\mathrm{xx}}, T_{\mathrm{xy}}, T_{\mathrm{xz}}, T_{\mathrm{yy}}, T_{\mathrm{yz}}, T_{\mathrm{zz}}$	gradient tensor components [E]

Table 3.2: *SGG data format (simulated data).*

	lines	SX-8	SX-9
ASCII:	50 000	19.51 s	25.00 s
	100 000	38.80 s	49.52 s
	200 000	77.58 s	97.32 s
	500 000	>30 min	251.28 s
binary:	100 000	0.43 s	0.71 s
	500 000	0.68 s	1.44 s

Table 3.3: *Time to read data on the NEC SX systems.*

It is possible that the data of one file start or end at different timestamps. Hence the data have to be synchronized. This task includes searching for the file which data start later and for the file which data end earlier. Only the data common to both files are to be used for the analysis.

When ASCII data are read by a computer, they have to be interpreted and converted to the binary format the computer uses before they can be used in computations. In most programming languages the necessary functions are available to read ASCII files and convert the data while reading. With a PC the reading and converting happen in a negligible amount of time, which is smaller than 2 seconds for 500 000 lines on a 2.2 GHz Athlon CPU.

After porting the program to the NEC SX systems, it turned out that the scalar unit of the CPUs is quite slow in comparison to the CPU of the desktop computer. For instance, it took around 39 s to read 100 000 data records on the SX-8 (see table 3.3). From this, one would expect that reading of 500 000 lines of ASCII data will take around 190 s. But sometimes the SX-8 shows a weird behavior, as it is possible that different programs are executed on the same node while requesting more CPUs than present. When too many processes share the same node, performance drops for each process. As a matter of fact, data reading on the SX-9 is a bit slower than on the SX-8. Altogether, these findings show that the scalar unit of the CPUs of the SXs is not optimized for integer operations.

As mentioned before, for GOCE real data analysis the number of data records per day is 86 400, which amounts to around 2.6 millon data records per month and roughly 32 millon data records per year. It would take too long to convert this amount of data with the SX-8 from ASCII to binary format. Therefore it was decided to convert the data beforehand on a local PC to binary format and to write a corresponding input funtion.

3.1.1 Byte order

While converting the data, another problem occured. The NEC SX systems take binary data in the so called big-endian format which means that the byte of the highest order is first. In contrary, PCs save binary data in the so called little-endian format which means that the byte of the lowest order is first.

adress (byte)	1000	1001	1002	1003
little-endian	0xCD	0xAB	0x34	0x12
big-endian	0x12	0x34	0xAB	0xCD

Table 3.4: *Byte order of the 32-bit integer number 305 441 741 (hexadecimal: 0x12 34 AB CD).*

Basically, no special advantages or disadvantages are known why to choose the one or the other byte order for hardware implementation. But the byte order should be considered when binary data are used. In particular, this is true if it becomes necessary to move data from one architecture to another.

In table 3.4, an example is given for data in big- and little-endian format. The number 305 441 741 (hexadecimal: 0x12 34 AB CD) shall be put to the address 1000 in memory as a 32-bit integer number. It also shows one of the benefits of the big-endian format: the byte order is like "normal", so numbers can be read easily by humans without reordering the bytes. It also applies to floating point numbers that the byte order can be simply reversed to get the other format, only there may be a different amount of bytes used to represent that numbers. A function to do the byte swapping is provided in code 3.1.

```
double swapEndian_double(double in) {
  int siz = sizeof(double);
  double tmp;
  char *tmp_ptr, *in_ptr;
  int i;
  in_ptr  = (char*)&in;
  tmp_ptr = (char*)&tmp;
  for (i = 0; i < siz; i++) {
    tmp_ptr[i] = in_ptr[siz - 1 - i];
  }
  return tmp;
}
```

Code 3.1: *Function for swapping the bytes of a double precision floating point number between little- and big-endian*

Because a desktop CPU can convert data quite fast, it was decided to convert the data on such a PC and save them in big-endian format. If the data are read again on a PC, the program reorders the bytes of the values back to little-endian format. And if the data are read on a SX system there is no need for conversions anymore.

3.1.2 Data storage

Another, although less important, reason to change the input format of the data is that the scalar unit of the SX processors gets a significant workload to put the single values of one line to the respective array in memory. This was not measured in particular as this part of the process takes only a few seconds. But nonetheless it is slower than the entire conversion of the data on a PC. Therefore it was decided to write the data columns to single files, one file per column, while converting the data from the ASCII format to the big-endian binary format.

The aformentioned measures allow to read the binary files directly to the different arrays in the memory without the need of converting the data. The amount of time to read and convert the data on the SX systems could be reduced severly this way to be less than one second for 500 000 datasets (see table 3.3).

As mentioned earlier, the data have to be synchronized before using them. This task is done before converting the data to binary format. Hence, one can be sure that the saved binary data are synchronized.

3.2 Design matrix assembly

During the setup of the design matrix **A**, the 4π-normalized Legendre functions of the first kind, $\bar{P}_{lm}(\sin\varphi)$, have to be evaluated. There exist several ways to compute the $\bar{P}_{lm}(\sin\varphi)$. Here, the algorithm according to Heiskanen and Moritz (1967) is used, which was shown to be numerically stable to a sufficient degree and order:

$$\begin{aligned}
\bar{P}_{0,0}(\sin\varphi) &= 1, \quad \text{(all } \bar{P} \text{ with at least one negative index is set to zero)},\\
\bar{P}_{m,m}(\sin\varphi) &= W_{m,m}\cos\varphi \bar{P}_{m-1,m-1}(\sin\varphi),\\
\bar{P}_{l,m}(\sin\varphi) &= W_{l,m}\left[\sin\varphi \bar{P}_{l-1,m}(\sin\varphi) - W_{l-1,m}^{-1}\bar{P}_{l-2,m}(\sin\varphi)\right]\\
&= W_{l,m}\sin\varphi \bar{P}_{l-1,m}(\sin\varphi) - W_{l,m}^{*}\bar{P}_{l-2,m}(\sin\varphi),\\
\text{with} \quad W_{1,1} &= \sqrt{3},\\
W_{m,m} &= \sqrt{\frac{2m+1}{2m}},\\
W_{l,m} &= \sqrt{\frac{(2l+1)(2l-1)}{(l+m)(l-m)}} = \sqrt{\frac{4l^2-1}{l^2-m^2}},\\
W_{l,m}^{*} &= W_{l,m}\cdot W_{l-1,m}^{-1} = \sqrt{\frac{(2l+1)(l+m-1)(l-m-1)}{(l^2-m^2)(2l-3)}}.
\end{aligned}$$

One can see easily that the factors W are only dependent on degree l and order m. Because of this, it is possible to compute these factors beforehand and store them into a table in the memory for later use. Also the trigonometric functions are only evaluated once for every φ and again the values are stored in memory. The later computation of the Legendre functions is accelerated considerably.

The original function for computing the Legendre functions was split into two. One for making the independent initialization computations for the factors W, and the other one for the actual calculations of the Legendre function for every φ. Looking at the indices of the factors W shows that we need to compute $l_{\max}$ values for $W_{m,m}$, $l_{\max}\frac{l_{\max}+1}{2}$ values for $W_{l,m}$ and $W_{m,m}$, $l_{\max}\frac{l_{\max}-1}{2}$ values for $W_{l,m}^{*}$.

Per φ the number of values computed for the Legendre function equals $(l_{\max}+1)\frac{l_{\max}+2}{2}$. A further gain in computing speed can be accomplished because we do not need to address values in memory which is time consuming, but can revert to the last two computed values which still reside in the registers of the processor. This is also shown in figure 3.1, where the arrows indicate the used values. It is easy to imagine how confusing the necessary index calculations of the first method are.

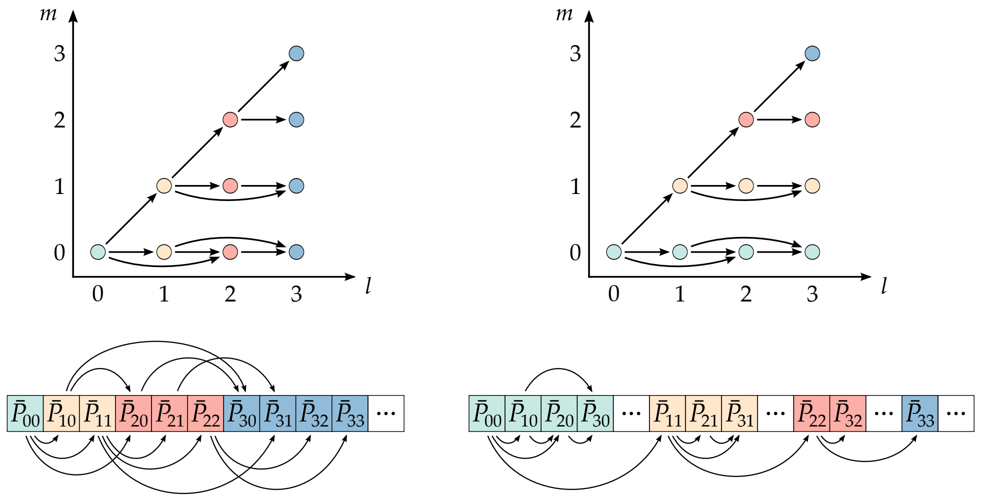
m
l
0
1
2
3
$\bar{P}_{00}$ $\bar{P}_{10}$ $\bar{P}_{11}$ $\bar{P}_{20}$ $\bar{P}_{21}$ $\bar{P}_{22}$ $\bar{P}_{30}$ $\bar{P}_{31}$ $\bar{P}_{32}$ $\bar{P}_{33}$ ⋯
$\bar{P}_{00}$ $\bar{P}_{10}$ $\bar{P}_{20}$ $\bar{P}_{30}$ ⋯ $\bar{P}_{11}$ $\bar{P}_{21}$ $\bar{P}_{31}$ ⋯ $\bar{P}_{22}$ $\bar{P}_{32}$ ⋯ $\bar{P}_{33}$ ⋯

Figure 3.1: *Computing the Legendre functions, same color indicates same program loop. Left: old method with complicated indexing, right: new method with simple indexing.*

Further potential to speed up the computations can be achieved with rearranging the equation 2.12 to

$$\left.\begin{array}{c}\frac{\partial V_{rr}}{\partial \bar{c}_{lm}}\\ \frac{\partial V_{rr}}{\partial \bar{s}_{lm}}\end{array}\right\} = \sum_{m=0}^{L} \left\{\begin{array}{c}\cos m\lambda\\ \sin m\lambda\end{array}\right\} \sum_{l=m}^{L} \underbrace{\frac{GM}{R^3}\left(\frac{R}{r}\right)^{l+3}(l+1)(l+2)}_{=:B_l} \bar{P}_{lm}(\sin\varphi) \tag{3.1}$$

which mainly was swapping both summation indices. With a clever arrangement of the summands and factors, it becomes possible to separate further parts of the computation. For example it is only necessary to compute the factors B_l per observation once for all $l = 0, \ldots, L$, save them for later usage and access only the precomputed values in memory. The same factors B_l are used in the summation over m several times. As this also applies for the Legendre functions $\bar{P}_{lm}(\sin\varphi)$ the summation over l ($l = m, \ldots L$) only requires the multiplication of trigonometric functions with pre-evaluated values which makes it very fast. Here also the trigonometric functions need only be computed once for $m = 0, \ldots, L$, but as the same value is only needed in the following loop over l, we can just store the specific values and not a whole vector.

One line of the design matrix consists of the partial derivatives with respect to the coefficients $\bar{c}_{lm}$ and $\bar{s}_{lm}$ in an alternating fashion. While assembling the design matrix, we take into account that some values are not needed. Those regarding $\bar{s}_{l0}$ are non-existant (see equation 2.1) and those regarding $\bar{c}_{1m}$ and $\bar{s}_{1m}$ are zero by definition (origin of coordinate system resides in Earth's centre of mass). However, one also has to keep in mind how the values are stored in memory to compute the correct index into the different arrays for retrieving the needed values. Some more minor computation tricks done are not mentioned, as they should be standard knowledge of a programmer.

3.3 Normal equations system setup

Equation 2.13 shows that the normal equations system dominates the required amount of memory as it has to be kept in the memory as a whole. The normal equations matrix of a linear system of equations is symmetric, therefore one could expect that only one triangle of the matrix needs to be computed and kept in memory. However, the BLAS functions require that the space for the whole matrix has to be allocated. Still only one triangle is computed, not needed elements of the memory contain arbitrary values.

The size of the design matrix as a whole gets too large, so completely putting it into the memory is out of question. Instead we assembled the normal equations matrix by splitting the observations into j blocks. The size of one of these blocks is chosen to be negligible in terms of memory requirements. The assembling of the normal equations matrix $\mathbf{N}$ and vector $\mathbf{b}$ works as follows

$$\mathbf{N} = \mathbf{A}^\mathsf{T}\mathbf{A} = \mathbf{A}_1{}^\mathsf{T}\mathbf{A}_1 + \ldots + \mathbf{A}_j{}^\mathsf{T}\mathbf{A}_j = \sum_{i=1}^{j} \mathbf{A}_i{}^\mathsf{T}\mathbf{A}_i = \sum_{i=1}^{j} \mathbf{N}_i\,, \tag{3.2}$$

$$\mathbf{b} = \mathbf{A}^\mathsf{T}\mathbf{y} = \mathbf{A}_1{}^\mathsf{T}\mathbf{y}_1 + \ldots + \mathbf{A}_j{}^\mathsf{T}\mathbf{y}_j = \sum_{i=1}^{j} \mathbf{A}_i{}^\mathsf{T}\mathbf{y}_i = \sum_{i=1}^{j} \mathbf{b}_i\,. \tag{3.3}$$

A graphic impression is given in figure 3.2.

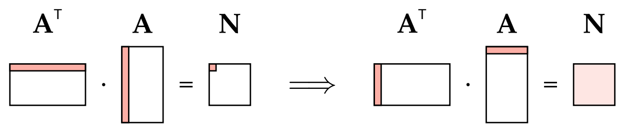

Figure 3.2: *The computation of the normal equations system is done by working on* $\mathbf{A}$ *linewise.*

In a first implementation we chose the dimension k of the single blocks first, afterwards we calculated the number of blocks j so that $n = k\,j$ holds true. Hence, the number of observations had to be an integer multiple of k and j. It was uncomfortable to handle how to read a specific amount of datasets with this implementation, so some improvements have been done.

With the improved implementation it is possible to read an arbitrary number of datasets. There is still the number of blocks j. But now this number as well as the size of a block k is computed by the program considering a specific memory size for the design matrix which is the only information to be provided by the user. Also the dimension of the last block is allowed to be smaller than k.

In the initial implementation the BLAS function `DGEMM` was used for matrix-matrix multiplication and summation

$$\mathbf{N}_i = \mathbf{A}_i{}^\mathsf{T}\mathbf{A}_i + \mathbf{N}_{i-1}\,, \quad \text{with} \quad \mathbf{N}_0 = \mathbf{0}\,.$$

We replaced this function by `DSYRK` which only computes one triangle of the symmetrical normal equations matrix and therefore is much more efficient, `DSYRK` is specially made for multiplying a matrix with its transposed (or the other way round).

The function `DGEMV` is used for the matrix-vector multiplication and summation

$$\mathbf{b}_i = \mathbf{A}_i{}^\mathsf{T}\mathbf{y}_i + \mathbf{b}_{i-1}\,, \quad \text{with} \quad \mathbf{b}_0 = \mathbf{0}\,.$$

The allocated memory for $\mathbf{N}$ respective $\mathbf{b}$ has to be initialized with zeros beforehand, which can be easily achieved with using the function `calloc` which overwrites the allocated memory with zeroes (instead of the function `malloc` which would just allocate the memory).

3.3.1 BLAS function calls

Different versions of BLAS are implemented for different computer architectures to put the special features of the hardware to full potential. Originally, BLAS was programmed in Fortran, but meanwhile an interface for C followed. This C interface was used for the shared memory version of the program. Here, the double precicion function calls we used in our program are presented (please refer to the *Netlib Repository* (2010) for a list of all function calls).

DGEMM $\mathbf{C} \leftarrow \alpha \mathbf{A}^{\mathsf{T}} \mathbf{B} + \beta \mathbf{C}$

```
cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans,
            m, n, k, α, A, lda, B, ldb, β, C, ldc);
```

with

α, β	scalars
`m, n, k`	integers, representing matrix dimensions
`A`	`k` × `m` matrix
`lda`	integer, leading dimension of `A`
`B`	`k` × `n` matrix
`ldb`	integer, leading dimension of `B`
`C`	`m` × `n` matrix
`ldc`	integer, leading dimension of `C`

DSYRK $\mathbf{C} \leftarrow \alpha \mathbf{A}^{\mathsf{T}} \mathbf{A} + \beta \mathbf{C}$

```
cblas_dsyrk(CblasRowMajor, CblasUpper, CblasTrans,
            n, k, α, A, lda, β, C, ldc);
```

with

α, β	scalars
`n, k`	integers, matrix dimensions
`A`	`k` × `n` matrix
`lda`	integer, leading dimension of `A`
`C`	`n` × `n` symmetric matrix
`ldc`	integer, leading dimension of `C`

DGEMV $\mathbf{y} \leftarrow \alpha \mathbf{A}^{\mathsf{T}} \mathbf{x} + \beta \mathbf{y}$

```
cblas_dgemv(CblasRowMajor, CblasTrans,
            m, n, α, A, lda, x, incx, β, y, incy);
```

with

α, β	scalars
`m, n`	integers, matrix dimensions
`A`	`m` × `n` matrix
`lda`	integer, leading dimension of `A`
`x`	vector of length `m`
`incx`	increment to reach the address of the next element of vector `x`
`y`	vector of length `n`
`incy`	increment to reach the address of the next element of vector `y`

3.4 Normal equations system solution

The normal equations system is symmetric positive definite. Hence it can be solved using the LAPACK routine `DPOSV` which computes the solution of a linear system of equations $\mathbf{Ax} = \mathbf{b}$ (*Netlib Repository*, 2010). As the normal equations matrix is symmetric, the function only needs the upper or lower triangular part of that matrix.

There are different implementations of LAPACK installed on the different used computer systems. Unfortunately the name of the DPOSV function and also the function call is different on each system.

Local computer: a version of CLAPACK is installed, the function call is similar to the CBLAS function calls.

DPOSV	solve a normal equations system

```
clapack_dposv(CblasRowMajor, CblasUpper, n, nrhs, A, lda, b, ldb);
```

with

`n`	integer, the number of linear equations
`nrhs`	the number of right hand sides
`A`	symmetric `n` × `n` matrix, holds the upper triangular part of that matrix
`lda`	integer, leading dimension of `A`
`b`	`n` × `nrhs` matrix (can hold multiple right hand side vectors), holds the results on return!
`ldb`	integer, leading dimenasion of `b`

Frontends, NEC SX systems: the call must be made to the Fortran LAPACK library.

On the frontends the function call is `DPOSV(uplo, n, nrhs, A, lda, b, ldb, info)`, however on the SX systems the name is changed to `dposv_` while the parameters remain the same.

All parameter must be pointers. When the parameter `CblasUpper` was used with the BLAS functions to assemble the normal equations system, the variable `uplo` has to contain the letter `"L"` for the function to work in the same way. The parameter `info` gives back the status information of the finished function call.

3.5 Makefile

For speeding up the development process it is very helpful to have a capable makefile at hand. A makefile contains the parameters for the program `make`. The parameters tell `make` how to compile and link a program, but can also control other tasks like copying data or running the program with commandline options.

Especially a makefile is handy because the options for compiling and linking a program often are lengthy. For example, the commandlines to compile, link and execute the program on the local computer would look like this:

compiling: `gcc -c main.c parser.c functions.c performance.c`,
linking: `gcc -o main main.o parser.o functions.o performance.o -lrt -lm -L/usr/local/ATLAS/lib/Linux_HAMMER64SSE_2 -llapack -lcblas -latlas`
and running: `./main`.

With a makefile it is possible to congregate different tasks within one so-called *target*. This makes it easy to port a program to different hardware architectures, maybe with the use of different programming libraries which depend on the specific hardware architecture. Within the target all necessary information to build the program like compiler, linker, parameters, libraries etc. is gathered.

As an example the makefile for the shared memory version of the program is explained here (see section A.3). In the appendix also the makefiles for the other program versions are provided.

In the file seven targets are included (`main`, `run`, `asama`, `asama_run`, `sx8`, `data` and `clean`). The target `main` is the default target. If the program `make` is called without further options, this target will be built. In our case the program will be compiled and linked for the local computer. It is also possible to call `make main` which would do the same. One can also list more than one target, e. g. call `make clean run` which first removes all object files and executables, next the program is compiled and linked and finally is executed.

Very practical is the possibility to declare variables which hold often used phrases. For example it is possible to specify the used compiler in a variable at a location, easy to find on top of the makefile, and refer to this variable later on. For example we use `gcc` as compiler on the local computer, therefore we assign it to the variable `CC_LOCAL` and refer to it with `$(CC_LOCAL)` later in the file. This also helps to prevent mistakes when a specific option should be changed at a later time. Now the option needs to be changed only once and changes take place globally.

The target `.PHONY` is not callable. It lists all the other targets which should be executed everytime. For example within the target `clean` no file called `clean` is created, hence such a file will never exist. Therefore this target will be executed evertime someone calls `make clean`. However, if somehow a file with this name comes to reside in the directory of the makefile, `make` will consider this file as up-to-date and the target would not be executed. To avoid this problem, the specific target can be put on the list of the target `.PHONY`.

3.6 Commandline options

To test different sets of parameters, commandline options can be called necessary for a program. Commandline options make it possible to change parameters of the program quickly without the need to recompile the program. Also for the NEC SX systems this is quite important as the access is in batch mode and the programs get executed at a later time specified by the scheduler. If one would change the program and recompile it, this new version would get executed instead. Also for making several consecutive tests in a row, every set of parameters would need one special version of the program which holds those parameters. To avoid getting things confused, a commandline parser was implemented.

For systems of the x86 architecture the library *getopt* is available. This library makes parsing the commandline easy. Unfortunately this library is not available for the SX systems, so we had to use a simpler version which does not allow long names of the parameters. The possible parameters for the pragram are shown in table 3.5. Now it is possible to compile the program once and register different jobs at the schedulers of the SX systems which only differ in their commandline paramters. This makes testing an easier task.

parameters	description
`--sstfile, -a <file>`	SST file; default: `../data/sc7/sc7_sst_coord_localsph`
`--sggfile, -b <file>`	SGG file; default: `../data/sc7/sc7_tensor_inv_localsph`
`--maxlm, -c <max>`	spectral resolution, maximum degree and order; default: 50
`--lines, -d <n>`	number of datasets to read; 0 = all, default: 100 000
`--outfile, -o <file>`	the file in which the estimated coefficients will be written; default: `./test.koeff`

Table 3.5: *Commandline parameters. For the SX systems only the short version is availabe.*

3.7 Runtime measurement routines

For testing purposes it is necessary to know how a program performs its different tasks. With this knowledge it is possible to detect the parts of the program which takes much of the overall runtime. These parts should be addressed first for speed-up considerations.

There exist several possibilities to gain knowledge to a certain degree about runtimes of a program. The most simple time measuring tool is the program `time` which is distributed with Linux and other UNIX-like operating systems. During a program run it measures the real time (which is the time that can be measured with a stop watch from the start to the end of the program), the user time (which is the projected runtime of the program to one busy processor) and the system time (which is the time that the operating system took for its processes, also projected to one processor).

Another possibility would be the use of a profiler program. Here special preparations are necessary as the program needs to be compiled with special options. For the different computers we used, different profiler programs are available. The results of these different profilers would be not directly comparable. Hence they were not used.

To achieve the goal – measuring the runtimes of different parts of the program – it was decided to write a small collection of functions which would use the system clock of a computer and at the same time make it possible to reset, start and stop several timers as well as taking a interim time. Also it should be possible to stop a running timer and start it again while continuing to count the runtime starting at the last stop.

On systems like the local computer or the frontends which are built with standard processors and are using Linux as an operating system, the system clock is read with the function clock_gettime. The time is given back with a resolution of nanoseconds. However the NEC SX systems use their special processors and have SUPER-UX as operating system. Here a function gettimeofday is available which returns the time with a microsecond resolution (which still is sufficient for the purpose of measuring the runtime).

For example, to measure the time it takes for assembling the design matrix **A**, the start and stop commands are placed before respectively behind the part of the program where this is done. The same is done before and behind the part where the normal equations matrix **N** is computed. Like this the runtime of all the parts of interest can be measured without having a major effect on the runtime itself as the function calls to read the system clock are fast. A small code snippet can be found in code 3.2. The timer functions themselves can be found in the files performance.h and performance.c which are provided in the appendix, section A.5.

```
Ttimer T1;

reset_timer(&T1);
for (i = 0; i < 1000; i++) {
  start_timer(&T1);
  do_something();
  stopp_timer(&T1);
  do_other_things();
}
printf("␣Time␣for␣'do_something()'␣:␣%1.4d␣s\n", totaltime_timer(&T1);
```

Code 3.2: *Example for the function calls to measure the runtime of program parts.*

3.8 Optimization results

Tests of the optimization were made continuously to see the efficency of the changes to the program immediately. Here only three special versions are picked out.

- The first version is a very early one where no optimizations were done yet. The data is read from ASCII-files, also still the routine DGEMM was used for adding up the normal equations matrix.
- The second version switched to the faster routine DSYRK and also the Legendre functions are computed with pre-evaluated values, though still ASCII data were used.
- Lastly, the third version is the actual version (also provided in the appendix) which incorporates all the optimizations mentioned within this chapter, in particular, the optimized Legendre functions with rearranged summations, binary data input and use of DSYRK.

The runtime results are shown in table 3.6. A speed-up of 1 would mean, that the program runs at the same speed as the basic version.

A large speed-up especially for the assembly of the design matrix could be achieved. Although a big part of the overall speed-up is clearly due to the use of DSYRK, the use of the routine DGEMM earlier can be counted as a beginner's mistake. The runtime measurements show that the time is dominated by the routine DSYRK, but as this routine is part of the BLAS library, it cannot be optimized by the user easily. The only possibility to gain more speed is a parallelization of the program, that more than one processor can participate in the computations. This topic will be covered by the next chapters.

program features:

version	input	**A**	**N**
1	ASCII	not optimized	`DGEMM`
2	ASCII	pre-evaluation	`DSYRK`
3	binary	pre-evaluation, reordered summations	`DSYRK`

parameters: $l_{\max} = 20$, $n = 100\,000$

version	input [s]	**A** [s]	**N** [s]	overall [s]	speed-up
1	1.83	11.36	27.91	41.11	1
2	1.79	2.83	5.91	10.55	3.90
3	0.77	0.60	5.95	6.63	6.20

parameters: $l_{\max} = 30$, $n = 100\,000$

version	input [s]	**A** [s]	**N** [s]	overall [s]	speed-up
1	1.83	25.67	156.91	184.20	1
2	2.06	6.83	25.35	33.77	5.45
3	0.88	1.07	25.52	27.59	6.68

Table 3.6: *Runtime results for three different program versions and different spectral resolutions $l_{\max}$ (tested on the local computer).*

Chapter 4

Parallelization with OpenMP

In the previous chapter, parts of the program were optimized to gain the most speed and computational efficiency while running the program on a single processor. With this effort a limit is reached where not much progress can be expected anymore. Or in other terms, it would cost too much effort to even get a small gain in the program's efficiency.

Hence, the idea of parallel computing is considered. OpenMP, the method introduced by this chapter, is useful for computers with shared memory architecture and at the same time quite easy to use.

4.1 Open Multi-Processing (OpenMP)

OpenMP is a programming interface which is developed together by different producers of hardware and compilers (e. g. AMD, Intel, IBM, HP, NEC) starting in the year 1997 (*OpenMP – Wikipedia*, 2010)[1].

OpenMP was developed to get an easy possibility to program multi-processor systems with a shared memory architecture. Nowadays, most of the commonly used compilers for C, C++ and Fortran (for example the GNU compiler collection) are equipped with this interface. This is also due to the growing spread of multi-processor (or rather multi-core) systems in office and home use.

Parallelization with OpenMP is taking place on a thread or loop level. With OpenMP, special compiler directives, called *pragmas*, are defined. Those directives instruct the compiler to build machine code which can run in parallel for those parts of the program which are enclosed by the directives. The pragmas are constructed in such a way that they look like regular comments to compilers which cannot handle them. Hence, a program can still be compiled by such a compiler, but in this case, it will only be able to run on a single processor.

Programming with OpenMP is quite a simple task, as the pragmas just need to be inserted into an already existing program source code. There is usually no more need for further changes in the source code.

In code 4.1, an OpenMP version of the well known "hello world"-example is given. In this example the parameters `omp parallel` instruct the compiler to build a parallel program. The program (in this context also called "master thread") forks at this point into several threads which run in parallel. Without further instructions the program forks into as many threads as

[1]This Wikipedia-article is recommended by the official OpenMP website, see (*OpenMP*, 2010)

processors are available. This means for this example that the phrase "Hello world!" is printed once by each processor.

```
int main(int argc, char* argv[]) {
  #pragma omp parallel
  printf("Hello␣world!\n");
  return 0;
}
```

Code 4.1: *OpenMP parallelized "Hello world!"-example*

The next example, see code 4.2, demonstrates the parallelization of a `for`-loop. It just prints a text on the screen. The parameters `omp parallel for` instruct the program to split the loop into several parts and distribute these parts to several threads. The loop is executed only once in total. This is a good method to initialize an array. Another example is the computation of several values in a loop, although a precondition is that those values are indepentend to each other. They cannot base on an earlier computation within this loop.

```
int main(int argc, char *argv[]) {
  int i;
  #pragma omp parallel for
  for (i = 0; i < 1000; i++) {
    printf("Hello␣this␣is␣no.␣%d\n", i);
  }
  return 0;
}
```

Code 4.2: *Mit OpenMP parallelisiertes `for`-Schleife*

There exist many more parameters for OpenMP which are only covered as far as needed in the following. For a deeper study on the possibilities of OpenMP, please refer to appropriate technical literature or to the manifold tutorials on the Internet.

4.2 Program adaption

As OpenMP is easy to use, the adaption of the program also went quite easy. Actually there were only a few things necessary to change the program from its single-threaded version to the OpenMP multi-threaded one.

A flowchart for the shared memory version of the program is displayed in figure 4.1. The flowchart shows no differences to the single threaded version of the program. Nonetheless a few differences exist in the two program versions:

- At first the OpenMP header files have to be included into the program by inserting the line `#include <omp.h>`.
- All variables are shared between the threads by default. This produces a problem when one variable is a pointer to some memory area. Eventually this variable becomes overwritten and therefore the memory area inaccessible. Such variables have to be private for each thread. They have to be defined globally and become thread private just after their definition with the pragma `omp threadprivate()`. In code 4.3, for example, the variables `a` and `b` should be private for each thread. The pragma becomes `omp threadprivate(a, b)`. Later, the memory allocation is done in a parallel part of the program.

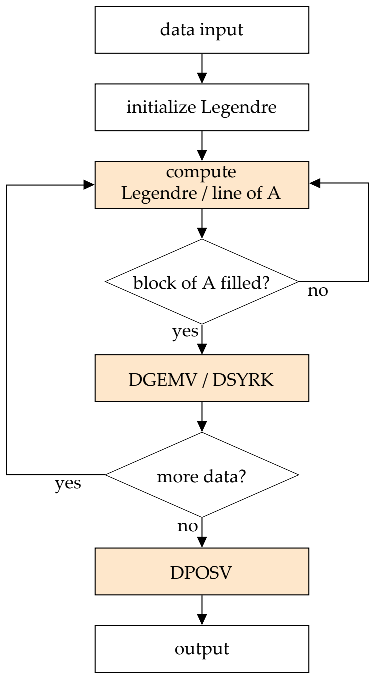

Figure 4.1: *Flowchart of the OpenMP-parallelized version of the program.*

```
#include <omp.h>

static double *a = NULL; *b = NULL;
#pragma omp threadprivate(a, b)

int main(int argc, char *argv[]) {

  #pragma omp parallel
  {
    a = (double*)malloc(sizeof(double) * 10);
    b = (double*)malloc(sizeof(double) * 20);
  }

  return 0;
}
```

Code 4.3: *OpenMP thread private memory allocation.*

- The part of the program which is to be executed in parallel is also surronded by the appropriate pragma. In case of the program, the design matrix is built within a `for`-loop, hence, `omp parallel for schedule(guided)` is used. The `schedule(...)` tells the compiler to build the program with a certain schedule algorithm for distributing parts of the loop. The `guided` algorithm breaks the loop into large chunks of contiguous iterations which are assigned to the threads. In direction to the end of the loop the chunk size decreases. As we deal with a huge amount of data, this algorithm is the most efficient one.

- While compiling, it is crucial to link the parallelized BLAS libraries to the program. Otherwise only the design matrix part of the program will be executed in parallel and the computing-intensive multiplying of the design matrix blocks with their transposed versions will be executed in a single thread. This would be a waste of ressources.

4.3 Parallelization results

Amdahl's law (Amdahl, 1967) states that every parallelized program also includes some parts which run sequentially. Hence, if k times the amount of processors are used, the program will not run k times as fast, usually. The time t_k which a parallelized program needs for its execution can be calculated from the time which is needed for the sequential part t_s and the time which is needed for the parallel part t_p as follows:

$$t_k = t_s + \frac{t_p}{k},$$

or in another representation:

$$\text{speed-up} = \frac{1}{r_s + r_p/k}$$

where $r_s + r_p = 1$ and r_s represents the ratio of the sequential part of the program. Still, Amdahl's law is only a rough description of the situation as overhead is not considered which is produced by the required communication of the processes.

For the OpenMP version of the program, the timer functions introduced in section 3.7 were used to measure the time for loading the data, the time for assembling the design matrix, the time for setting up the normal equations system as well as the overall time required by the program (also often called *wallclock time*, the time which can be measured with a wall clock).

4.3.1 Local computer and frontend

On the local computer the testing of the shared memory version could only be performed with maximally two processors. Therefore the local computer was used for small and quick tests during the development. More elaborate testing was done at the frontend Asama which consists of 32 processors. As some of the processors are occupied by other processes (management of the SX systems, other users testing their programs) a maximum of 21 processors was used by our program.

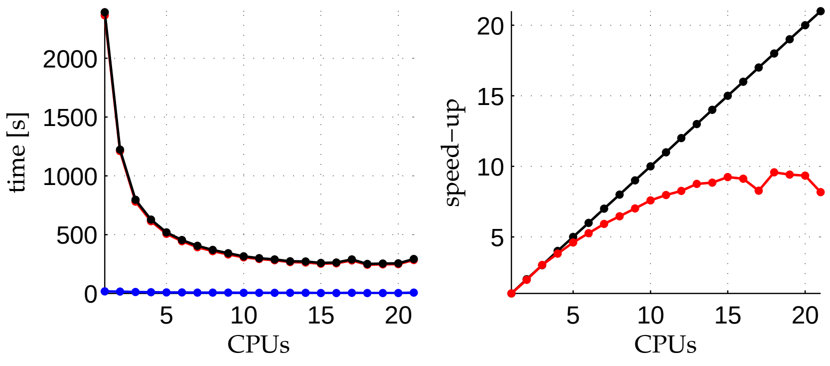

Figure 4.2: *Shared memory version with OpenMP on the frontend Asama, left: time for design matrix assembly (blue); time for normal equations system setup (red); overall time (black); right: ideal speed-up (black); achieved speed-up (red). Parameters for the test:* $l_{\max} = 70$, $n = 500\,000$.

Figure 4.2 illustates the results of the runtime measuremant and the speed-up. Both graphs show that the program scales well, but at a certain point no gain in speed can be achieved anymore with incorporationg more CPUs. This can be seen in the graph on the right hand side of the figure. The gap between the ideal speed-up and the true speed-up grows the more CPUs are used. At a certain point, the speed-up is completely used up for the communication overhead. Usually in computer engineering, resources are handled in powers of two. Hence, for the frontend Asama this point can be set at 16 CPUs. But one could also discuss if this point should be set earlier (e. g. at 8 CPUs).

4.3.2 SX systems

Further runtime results were gathered on both NEC SX systems, SX-8 and SX-9. The results are displayed in figures 4.3 and 4.4. For a job exactly the same number of processors was requested from the scheduler as there would be processes. The scheduler of the SX systems also provides some time measurements, which are the real time (the same as overall time or wallclock time), the user time (sum of the time used on all processors), the vector time (sum of the time used in the vector unit of all processors) and the system time (the time which the operating system needs for its tasks and which is negligible short usually). These time measurements give information about the overall efficiency of the program. The efficiency of a program can be seen in the user time or the speed-up. Ideal would be a straight line parallel to the x-axis for the user time or a speed-up factor close to the number of used processors.

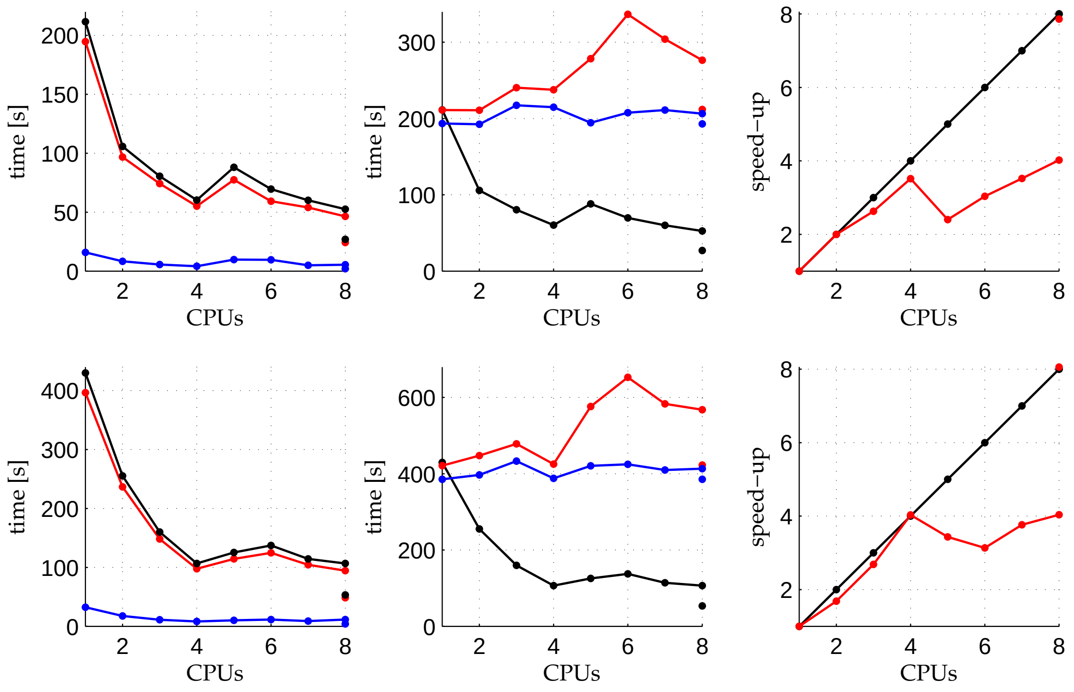

Figure 4.3: *Shared memory version with OpenMP on the SX-8. Left: time for design matrix assembly (blue); time for normal equations system setup (red); overall time (black). Middle: user time (red); vector time (blue); real time (black). Right: ideal speed-up (black); true speed-up (red). The top graphics use $n = 250\,000$ lines of data, the bottom graphics $n = 500\,000$, in both cases $l_{\max} = 50$. Single dot: exclusive use of a single node.*

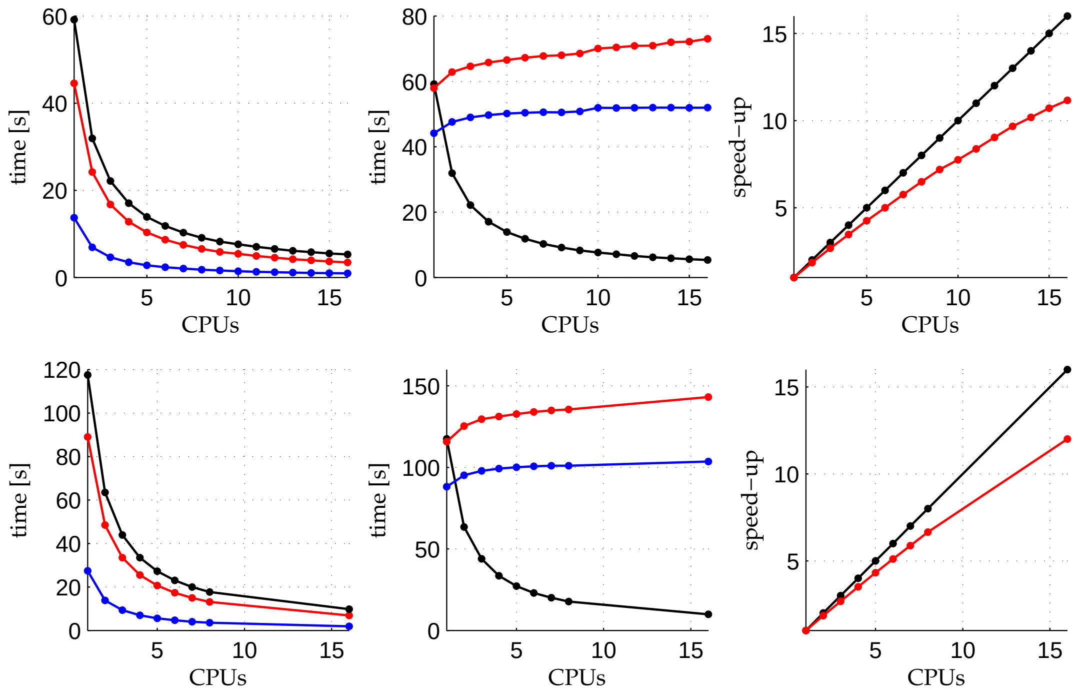

Figure 4.4: *As in figure 4.3, but for the SX-9.*

The SX-8 shows a weird behaviour the more processors are used. An explanation can be found in the way the scheduler for the SX-8 works. There exist two different types of jobs: parallel and single/multi ones. Parallel jobs are used if not a complete node is needed. Several other jobs are allowed to run on the same node then. In contrast, a single job obtains the exclusive access to a complete node, respectively a multi job to two and more nodes. In the parallel job mode the scheduler executes up to twelve processes on one node which only has eight processors. Hence, the processes block each other with high probability. The probability that a owned process gets hindered by foreign ones decreases if fewer owned processes are started. This can be nicely seen in figure 4.3, as up to four processors the SX-8 shows an expected behaviour.

For the program test with eight CPUs, a node was also requested in the single job mode. Here, the situation differs completely as the program has an exclusive access to the ressources of one node. The SX-8 scales very well in the single job mode, which can be seen in the speed-up which is close to ideal.

The SX-9 shows a near idealistic behaviour. The nodes of the SX-9 consists of 16 processors which are also shared by a maximum of 16 processes. Because of the weird behaviour of the SX-8, the time measurements obtained from the SX-9 are more meaningful. The user time is slowly rising, in a linear fashion though. This indicates that the parallelization scales well. The point where the usage of more processors would be giving no more benefit to the parallelization cannot be reached at the SX-systems as the SX-8 has only 8 CPUs per node and the SX-9 only 16 CPUs per node.

As a last test of the program, the behaviour for different $L = l_{\max}$ was analyzed. As expected the program shows an exponentially increasing run-time for increasing $l_{\max}$, as can be seen in figure 4.5.

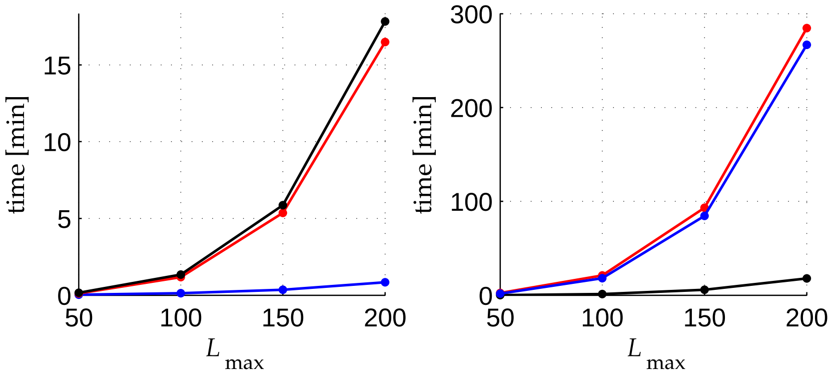

Figure 4.5: Shared memory version with OpenMP on the SX-9. Left: time for design matrix assembly (blue); time for normal equations system setup (red); overall time (black). Right: user time (red); vector time(blue); real time (black). Parameters were: 16 CPUs, $n = 500\,000$ lines of data.

4.3.3 Numerical precision of the computations

Just to be sure that the numerical precision is within acceptable range, the resolved spherical harmonic coefficients are compared to those of the EGM96[2] which are the input parameters of the simulated GOCE data used in this study.

Two different approaches are chosen. First, the degree-error root mean squares (DE-RMS) cumulate the error of each degree l:

$$\text{DE-RMS}_l = \sqrt{\frac{\sum\limits_{m=-l}^{l} (v_{lm}^{\text{ref}} - \hat{v}_{lm})^2}{2l+1}}\,,$$

with $v_{lm}^{\text{ref}} = \bar{c}_{lm}^{\text{ref}}$ for $l \geq 0$ and $v_{lm}^{\text{ref}} = \bar{s}_{lm}^{\text{ref}}$ for $l < 0$ beeing the reference coefficients of the EGM96. Analogously, the $\hat{v}_{lm}$ which assemble the estimated geopotential parameters. Secondly, the relative empirical errors read

$$e_{v_{lm}}^{\text{rel}} = \left\| \frac{v_{lm}^{\text{ref}} - \hat{v}_{lm}}{v_{lm}^{\text{ref}}} \right\| .$$

They show the precision of each coefficient. Because of the GOCE orbit inclination of 96.7° there exist polar gaps. These are the areas over the poles of the Earth where the satellite cannot pass by. Because of the polar gaps the coefficients with low order cannot be estimated with high presicion. Hence, there is a wedge visible along the vertical axis of the graphic on the right hand side of figure 4.6. For this reason, coefficients up to order $m = 20$ were neglected for the calculation of the DE-RMS values.

It can be verified that the numerical precision is within acceptable range (see figure 4.6). Hence, the conclusion can be drawn that the OpenMP version of the program produces good results in terms of spherical harmonic coefficient recovery.

[2]EGM96 = Earth Gravitational Model 1996

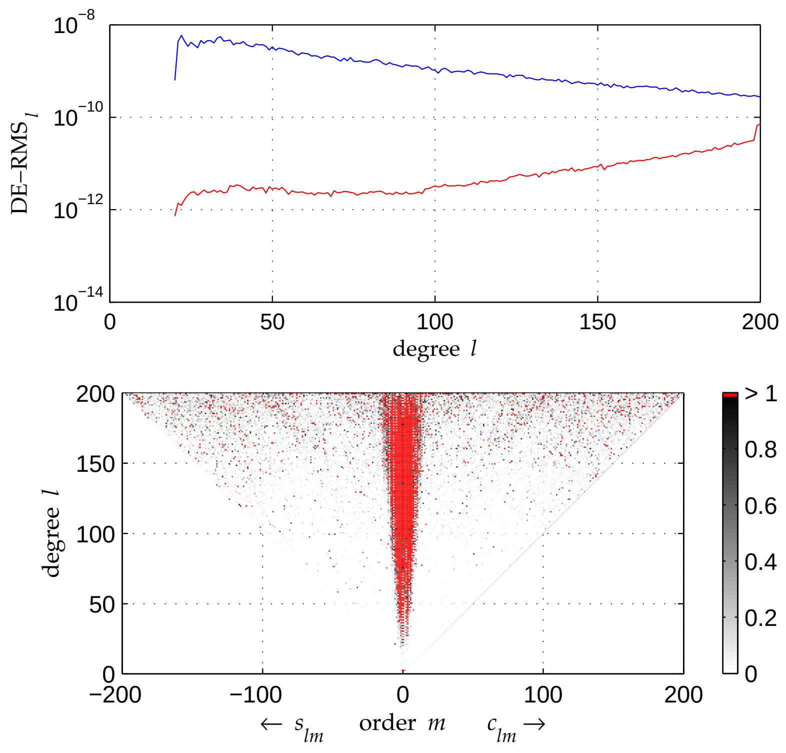

Figure 4.6: *OpenMP version of the program, degree and order $L = l_{\max} = 200$. Left: EGM96 signal (blue), DE-RMS values (red). Right: Relative empirical errors.*

Chapter 5

Parallelization with MPI

The last chapter showed how to parallelize the program at a shared memory architecture with OpenMP. This approach already accelerated the program considerably. However, it is only possible to incorporate one computing node into the computations with the shared memory version of the program. With the goal to speed up the program further, a new program version was implemented which makes it possible to use several computing nodes at the same time.

For this distributed version of the program it is necessary to establish a communication between the single processes as each process has only access to its own memory area. For the communication the library MPI is used which is introduced in this chapter. Fortunately with PBLAS[1] and ScaLAPACK[2] further libraries exist which provide similar functions as BLAS and LAPACK. Though the difference is that these functions aim for the processing of vectors and matrices which are distributed over the processes.

Programming now gets considerably more complex as the programmer has to tell the processes which of them hold which part of a matrix or a vector. Also the communication has to be programmed and the matrices and vectors have to be distributed. A method how this can be achieved is also described in this chapter.

5.1 Introduction

5.1.1 Message Passing Interface (MPI)

The MPI standard has been developed since 1992 by a group of several companies (e. g. IBM, Intel, Cray etc.), laboratories and universities. The project is hosted at the Argonne National Laboratory, Illinois, USA. There exist a variety of implementations for different hardware architectures. At the moment the MPI standard is available in version 2 (*MPI*, 2010).

MPI is one of several methods of parallelizing programs on distributed memory systems (it works also on shared memory systems though). In contrary to OpenMP it needs much more programming work to be done. Another difference is that the program will not run by a direct call anymore. It has to be called via the program `mpirun`[3]. A direct comparison to OpenMP shows the complexity of MPI clearly. While an existing program is parallelized with OpenMP

[1]PBLAS = Parallel Basic Linear Algebra Subprograms.
[2]ScaLAPACK = Scalable Linear Algebra PACKage.
[3]There exist other aliases for it like `orterun`.

very easily just by entering some adequate `pragma` directives, the parallelization with MPI often needs a restructuring of the program code and several additions to it.

A minimal outline of a MPI program is shown in code 5.1. To use MPI, the header file of the MPI library has to be included. In the program flow first MPI must be initialized. Then it is useful to retrieve the number of all processes as well as the rank of the actual process. At the end of the program MPI has to be finalized.

```
#include "mpi.h"

int main(int argc, char* argv[]) {
  int procs, myid;         // number of processes, process rank

  /* Start MPI */
  MPI_Init(&argc, &argv);

  /* get number of processes and rank/name of this process*/
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  printf("Hello␣from␣process␣%d␣of␣%d\n", myid, procs);

  /* Shut down MPI */
  MPI_Finalize();

  return 0;
}
```

Code 5.1: *Minimal outline of a MPI program.*

5.1.2 Scalable Linear Algebra PACKage (ScaLAPACK)

The devolpment of ScaLAPACK started in the mid-1990's with the first version released in 1995 (Blackford et al., 1996). ScaLAPACK, together with PBLAS, provides a subset of BLAS and LAPACK routines which are redesigned for distributed memory computers. In figure 5.1 the hierarchy of the needed libraries is depicted. MPI can be exchanged by other communication libraries, depending on the architecture of the computer and network infrastructure.

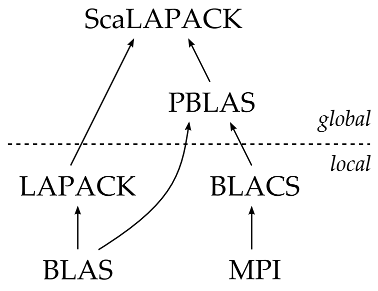

Figure 5.1: *ScaLAPACK software hierarchy.*

BLAS, LAPACK and MPI were already introduced. Together with BLACS[4] those libraries are running locally on the same process. BLACS bundles some of the functions of the underlying communication library (here MPI) to provide a more convenient communication interface for

[4]BLACS = Basic Linear Algebra Communication Subprograms.

the needs of PBLAS and ScaLAPACK, also to make it easier to port a program to different hardware architectures.

ScaLAPACK and PBLAS handle linear algebra computations globally. They break the problem down to smaller portions which are solved by the single processes locally using LAPACK and BLAS. Here, communication is necessary as different parts of matrices and vectors are stored in the memory area of different processes.

The usage of ScaLAPACK needs more initialization to be done before the computations can start. In code 5.2, a minimal example of a ScaLAPACK program is given (Petersen and Arbenz, 2004). Again, MPI has to be initialized first. Then a so-called *BLACS context* has to be requested. After that a processing grid has to be set up. Here it is ideal to have a quadratic grid, however this is not possible every time. So the program calculates the grid size to be close to quadratic with the given number of processes. The grid setup is finished by initializing it. Usually, the coordinates of the process in the processing grid should be retrieved afterwards, as those coordinates will be used during the setup and distribution of the matrices and vectors, which is shown later in detail. For finishing the program, in reverse order first the processing grid is released and afterwards MPI is finalized.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "cblas.h"
#include "mpi.h"

int main(int argc, char* argv[]) {
  /* MPI and BLACS stuff */
  int procs, myid;                    // number of processes, process rank
  int ctxt, prow, pcol, myrow, mycol;  // BLACS stuff

  MPI_Init(&argc, &argv); // Start MPI

  MPI_Comm_size(MPI_COMM_WORLD, &procs); // get number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);  // get process id

  Cblacs_get(0, 0, &ctxt); // get BLACS context

  /* Initialize "optimal" 2D processing grid (so that prow <= pcol) */
  pcol = sqrt(procs);
  while (procs % pcol != 0) {
    pcol--;
  }
  prow = procs / pcol;
  if (prow > pcol) {
    pcol = prow; prow = procs / pcol;
  }

  /* Initialize the prow x pcol process grid */
  Cblacs_gridinit(&ctxt, "Row-major", prow, pcol);
  Cblacs_pcoord(ctxt, myid, &myrow, &mycol);

  if (myid == 0) { // only the main process does this
    printf("␣␣-->␣%d␣processes␣used␣in␣a␣%d␣x␣%d␣grid\n", procs, prow, pcol);
  }

  /*
      ... COMPUTATIONS ...
  */

  Cblacs_gridexit(ctxt); // release process grid
```

```
    MPI_Finalize(); // shut down MPI

    return 0;
}
```

Code 5.2: *Minimal example for a ScaLAPACK program with automatical calculation of the processing grid size.*

5.1.3 Block-cyclic distribution

ScaLAPACK demands a special arrangment of the matrices and vectors in the memory of the computing nodes in the processing grid. The matrices and vectors are split into blocks which are distributed over the single processes. The distribution scheme is called *block-cyclic* (Blackford et al., 1997).

The block-cyclic scheme can be explained the best with an example. The following matrix shall be distributed.

$$\begin{array}{c|c|c|c|c|c|c|c|}
 & \multicolumn{7}{c}{\xrightarrow{n}} \\
\hline
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
m \downarrow & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
 & 15 & 16 & 17 & 18 & 19 & 20 & 21 \\
\hline
 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\
\hline
 & 29 & 30 & 31 & 32 & 33 & 34 & 35 \\
\hline
 & 36 & 37 & 38 & 39 & 40 & 41 & 42 \\
\hline
 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \\
\hline
\end{array}$$

LAPACK was programmed in Fortran, hence, the organization of array elements in the memory is different to programs which are written in C. While the matrices in a C-program are stored line-by-line, matrices are organized column-by-column by Fortran-programs. Hence, this has also to be considered for our program. The matrix in this examples resides column-by-column in the memory. The single elements are ordered like this in the memory:

1	8	15	22	29	36	43	2	9	16	23	30	37	44	3	10	...

In this example, the matrix is split into blocks of the size 2×2 and distributed to the processes. The first line of blocks of the matrix is distributed over the first line of the processes in the processing grid by sending the first block to process 0 of this line, the second block is sent to process 1 and so forth. If there are no further processes in this line but still blocks of the matrix left, one starts at process 0 again.

The second line of blocks of the matrix is distributed over the second line of the processes in the processing grid in the same fashion. If there are no further lines in the processing grid but still lines of blocks of the matrix left, one starts over at the first line of the processing grid.

In this example a processing grid of size 2×3 is chosen. Hence the matrix is split into the following blocks, coloured to show the affiliation to the different processes:

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42
43	44	45	46	47	48	49

Those blocks are distributed to the processes and stored into local matrices as follows:

(0, 0)

1	2	7
8	9	14
29	30	35
36	37	42

(0, 1)

3	4
10	11
31	32
38	39

(0, 2)

5	6
12	13
33	34
40	41

(1, 0)

15	16	21
22	23	28
43	44	49

(1, 1)

17	18
24	25
45	46

(1, 2)

19	20
26	27
47	48

Again, it is of importance to consider the different storing schemes of Fortran- and C-programs. The data is organized in the local memory as follows:

(0, 0)	1	8	29	36	2	9	30	37	7	14	35	42
(0, 1)	3	10	31	38	4	11	32	39				
(0, 2)	5	12	33	40	6	13	34	41				
(1, 0)	15	22	43	16	23	44	21	28	49			
(1, 1)	17	24	45	18	25	46						
(1, 2)	19	26	47	20	27	48						

There exists a function `numroc_` which calculates the size of a local matrix. It is important to know this size during memory allocation as there should not be allocated less memory than required. It would lead to a faulty program. On the other hand, it would be a waste of memory to allocate more memory than necessary.

The source code A.1 in the appendix reproduces this example. The distribution scheme of a vector is similar; a standing vector is distributed to only one column in the processing grid, a lying vector is distributed to a row accordingly.

5.2 Cluster installation

5.2.1 The GIS-Cluster

Handling with the high performance computers at the HLRS showed some huge drawbacks. As the SX-8, and even more the SX-9, are high-efficient systems, many users want to execute their computations on them. Therefore, the job queues are long which implies usually a long waiting time until the job is executed. This is a huge problem when one is in the programming or testing stage where constant feedback is necessary to see if a function works well or if there are still some bugs left.

On these systems, it is not possible to watch how a job is performing at runtime. Instead, when the program finishes, some statistics are returned. Those statistics are mean values and sometimes can be a bit difficult to interprete. On a local computer it is possible to observe the processes during runtime, for example with the program `top` which returns some process values like processor load, memory usage etc. in real-time. This makes the process more transparent to the user, as it can be seen how the load of the ressources changes while the program is running. It is also easier to learn in detail how a cluster computer works and how the processes and software components depend on each other.

Hence, it was decided to build a cluster computer for testing purposes. There exist several Linux distributions which are specially built for cluster computing. The Linux distribution *Rocks Clusters* of the University of California (*Rocks Clusters*, 2010) proved to be very beginner friendly and makes installing and running a computing cluster quite easy. For a test, all what is needed are some PCs with a network card, sufficient memory and a hard disk drive, then of course the Rocks Clusters software and some networking hardware (a switch or hub) as well as one additional network card for the frontend node to connect the cluster to the university network. Eventually there is also the need of a net boot CD[5] if the PCs are not able to boot over the network by themselves.

The general layout of the cluster is illustrated in figure 5.2. In the end, it was decided that a minimal cluster is sufficient for the intended use (only testing, no real workload). Thus, the GIS-cluster consists only of two disused PCs from the institute. The hardware specifications of those PCs are listed in table 5.1. For the cluster interconnecting network an old 10BaseT Ethernet hub was used. The additional network card for the frontend node got "rescued" from the electronic scrap of the university.

	name	CPU	RAM	HDD	network cards
frontend	cluster	Intel Pentium IV, 3.4 GHz	1 GB	70 GB	2
node	compute-0-0	Intel Pentium IV, 3.0 GHz	1 GB	30 GB	1

Table 5.1: *Hardware used for the cluster nodes.*

[5]e. g. from `http://etherboot.org/`

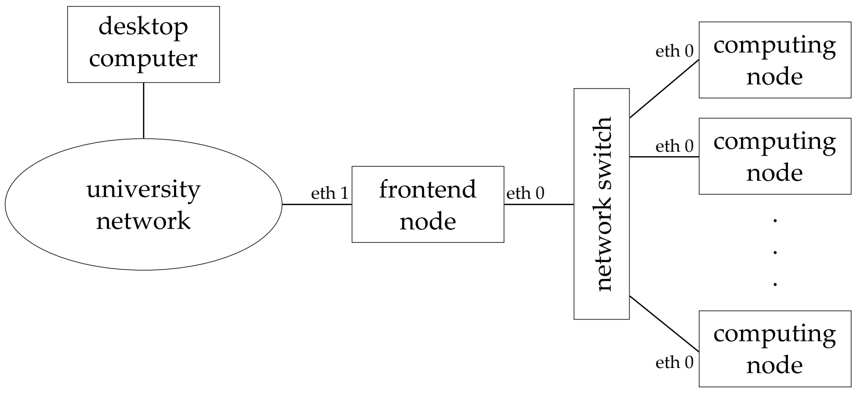

Figure 5.2: *General layout of the GIS-cluster.*

5.2.2 Brief installation guide for a Rocks Cluster

This short guide references to version 5.3 of the Rocks Clusters Linux distribution. It is recommended to have a look at the much more detailed official documentation before installing a cluster. The guide given here covers mainly the purpose to document what was done to install our cluster. But it might become useful if someone wants to build a computing cluster.

Installation of the frontend node. The software comes in several packages – called *rolls* by the makers of Rocks Clusters. There exist two versions of the software, one supports the i386 architecture (e. g. Athlon, Pentium and Xeon CPUs) which was also used for the GIS-Cluster, the other one supports the x86_64 architecture. At least four CDs are needed which hold the necessary rolls to install the frontend node: Kernel/Boot-Roll, Core-Roll, OS-Roll disk 1 and 2.

Some more important notes should be made before the installation is started: When the installation progresses all the data on the hard disk drive are erased! There where some problems during the installation at PCs with Intel CPU with HyperThreading enabled, so this should be disabled in the BIOS.

The installation itself is fairly easy – first, the BIOS options should be set so that the first boot device of the frontend node is the CD drive. Then the computer is booted from the Kernel/Boot Roll CD. When the splash screen appears, the option `build` has to be entered to install the cluster software. After booting, the rolls which should be installed have to be registered. For a computing cluster, the rolls according to table 5.2 should be installed. For the registering process, the CDs are changed and the rolls chosen accordingly.

Roll	on CD
kernel	Kernel/Boot-Roll
base, ganglia, hpc, web-server	Core-Roll
os	OS-Roll – disk 1
os	OS-Roll – disk 2

Table 5.2: *Mandatory rolls for installing a computing cluster.*

Then, some more questions are asked, e. g. which network address the frontend should show to the outside network (address for the device `eth1`). The network address can also be obtained through `dhcp`, but only during installing, afterwards this adress will be fixed! At this stage of installing, the CDs are requested again to copy the chosen rolls to the hard disk drive. Afterwards, the installation runs completely automatically.

Installation of the computing nodes. When the frontend node is installed and rebooted, one has to log in as *root* and start the program `insert-ethers` then select the option `Compute`. Afterwards, the first computing node, which will get its boot image over the network, is switched on and will install itself. When the installation has finished at this node, the next node is switched on etc. If there is no option in the PC's BIOS to boot over network, a net boot CD can be used. The cluster's growth can be watched over the network with the help of the ganglia tool. This can be done by entering the IP address of the frontend with an added `/ganglia` in a web browser, for example `http://192.168.6.25/ganglia`.

5.2.3 Numerical libraries installation

BLAS and LAPACK (respectively PBLAS and ScaLAPACK) functions are used for the program. Accordingly, we need to compile them also for our cluster. All these libraries are available at the Netlib Repository[6]. Although most of the libraries are not optimized for a specific hardware architecture, these reference implementations work for our purpose sufficiently.

BLAS, LAPACK, BLACS, SCALAPACK. Netlib provides an installer for ScaLAPACK (version 0.96) which makes downloading and compiling of the underlying libraries very easy. For an installation on the Rocks Cluster the archive has to be downloaded, the contents extracted and the setup script executed with the command line `./setup.py -downblas -downblacs -downlapack -f90=gfortran`. With these parameters the setup script downloads and compiles BLAS, BLACS and LAPACK. After this is done, ScaLAPACK will be compiled. The parameter `-f90=...` is needed to tell the script which Fortran compiler to use. Instead of downloading the reference implementations also optimized implementations can be used. Such optimized libraries are able to bring out more performance of the installed hardware. As our cluster is only a test environment for the programs and no productive computations are done, we decided against the trouble of installing optimized libraries.

After the compiling is finished, the libraries can be found in the subdirectory `lib`. For an easier usage, the libraries are copied to a new directory `LIBS`, e. g. in the home folder, and are renamed according to table 5.3. The renaming is necessary to link the libraries to the program easily later on.

CBLAS. Our program is written with the language C, so also CBLAS, the C-interface to the BLAS library, is necessary. It can be downloaded from Netlib as well. After extracting the files there is a bit of editing to be done.

[6]`http://www.netlib.org/`

after compiling	$\longrightarrow$	renamed
`blacs.a`		`libblacs.a`
`blacsC.a`		`libblasCinit.a`
`blacsF77.a`		`libblasF77init.a`
`librefblas.a`		`libblas.a`
`libreflapack.a`		`liblapack.a`
`libscalapack.a`		`libscalapack.a`
`cblas_LINUX.a`		`libcblas.a`

Table 5.3: *Renaming scheme for BLAS, LAPACK & Co.*

First, in the directory of CBLAS a link, named `Makefile.in`, to the architecture dependent makefile should be created. In our case, the command line was `ln -s Makefile.LINUX Makefile.in`. Afterwards, some parameters in the file `Makefile.in` have to be edited: `CBDIR` should point to the base directory of the CBLAS installation files, `BLLIB` should point to the earlier compiled BLAS library and `FC` should name the Fortran compiler (again `gfortran`).

After this is completed, `make alllib` is called to compile CBLAS. The compiled library can be found in the subdirectory `lib/LINUX`. It is copied to the same directory `LIBS` as before and renamed according to table 5.3. Also the header file `cblas.h` needs to be copied to the same directory. This file can be found within the CBLAS source code folder.

5.3 Program adaption

A flowchart of the distributed memory version of the program is given in figure 5.3. It can easily be seen that this program version is more complex. The distributed memory version of the program needs several initializations to be done before a communication between the processes is established (a skeletal structure for initializing MPI and a processing grid was shown in code 5.2). Beside those initializations (and finalizations at the end of the program), several other steps get necessary.

5.3.1 Data input and distribution.

In general, the data input works in the same manner as for the shared memory version. Due to performance reasons, only one process reads the data from the hard disk drive. After the data are read, they will be sent to all the other processes with the help of the MPI function `MPI_Bcast`.

Starting with version 2 of MPI there exists a parallelized file input routine named `MPI_File_read` which reads the file in a parallelized manner, giving the data line by line to alternating processes. But this idea was discarded as it would need major changes in the structure of the program, also, after the ASCII to binary conversion, reading the data and distributing them is not time critical anymore and the memory usage for storing all the data with each process is not that high.

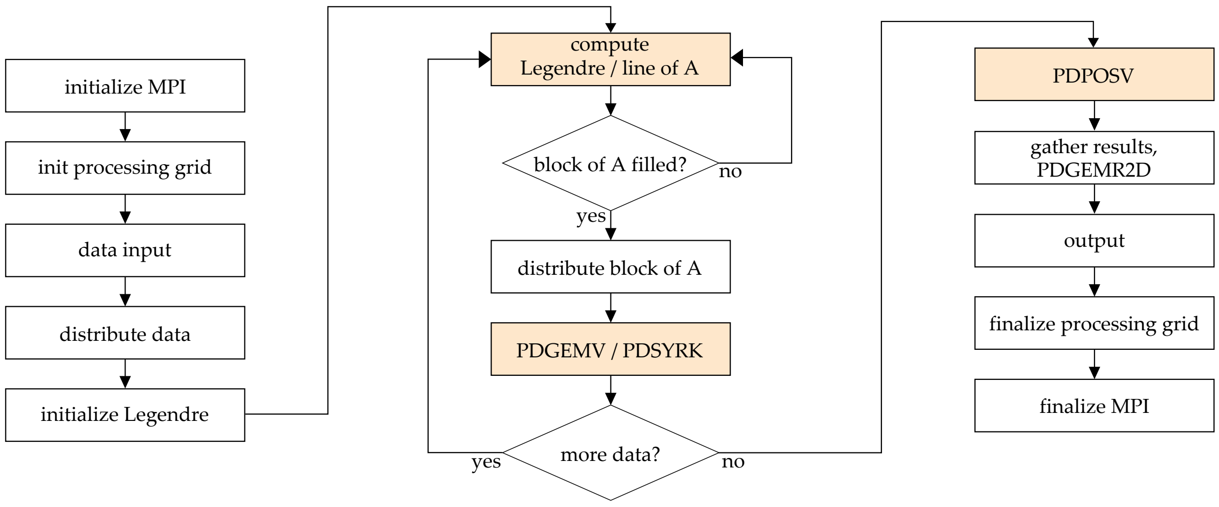

Figure 5.3: *Flowchart of the MPI-parallelized version of the program.*

5.3.2 Initializing a distributed matrix

One point not shown in the flowchart of the distributed memory version is the initialization necessary for the distributed matrices and vectors. There exist two handy functions named `numroc_` which returns the size of a local matrix and `descinit_` which does the initializing of the distributed matrix.

First the size of the matrix part stored at the process is calculated. This is done by two calls of the function `numroc_`, one call for each matrix dimension. After the local size of the matrix is obtained, the second step becomes possible and the proper amount of memory can be allocated. The last step is to tell ScaLAPACK how the matrix will be distributed with the routine `descinit_`.

NUMROC	compute local size of matrix

```
loc_n = numroc_(n, nb, iproc, isrcproc, nprocs);
```

with

`loc_n`	output: number of rows/columns of the local part of the matrix
`n`	the number of rows/columns of the distributed matrix
`nb`	size of the blocks the distributed matrix is split into
`iproc`	coordinate of the process whose local array row or column is to be determinded
`isrcproc`	coordinate of the process that possesses the first row or column of the distributed data
`nprocs`	total number of processes in the row or column over which the matrix is distributed

All variables are pointers to integer numbers.

DESCINIT	initialize array descriptor of distributed matrix

```
descinit_(desc, m, n, mb, nb, irsrc, icsrc, ictxt, lld, info);
```

with

`desc`	output: pointer to an array of integer of the length 9, the array descriptor for the distributed matrix.
`m`	number of rows in the distributed matrix
`n`	number of columns in the distributed matrix
`mb`	number of rows in one block of the matrix
`nb`	number of columns in one block of the matrix
`irsrc`	first row in processing grid which holds the first block of the matrix
`icsrc`	first column in processing grid which holds the first block of the matrix
`ictxt`	the BLACS context handle determined at the initialization of the processing grid
`lld`	leading dimension of the local array storing the local blocks of the distributed matrix
`info`	output: status information.

All variables are pointers to integer numbers.

5.3.3 Design matrix assembly and redistribution

For the OpenMP version of the program we made great efforts to optimize the computations. We found that the version which computes one line of the design matrix per process runs fastest, because part of the computations are the same for each matrix entry of one line. This effort done there should not go to waste so it was decided to reuse that part in the MPI version of the program. But here the situation differs. All the processes are not computing one block of lines together but every process computes a block, which does not contain full lines, for itself as the memory is not shared. Also the blocks of the matrix are expected to be block-cyclic distributed for the PBLAS- and ScaLAPACK-routines. With this new situation new problems occur. These problems have to be seen from the viewpoint of a process in the processing grid:

- on which data should I work,
- which of the blocks I computed have to be sent to other processes,
- which blocks do I receive from other processes,
- to which process should I send a specific matrix block,
- from which process do I receive a specific block?

An example solution of these problems is illustrated in figure 5.4.

At first, each process computes one block-line of matrix **A**. Then, for each block, the global coordinates are computed. From the global coordinates, in turn, the new local coordinates in the process grid are computed. With this the process can find out if the block can be kept or has to be sent to another process and from which process a block will be received instead. For this approach, the network traffic will be the higher the more processes are involved. Because, the more processes are used the more blocks have to be exchanged over the network. In conclusion this approach implies a fast network connection for the computing nodes to transfer this large quantity of data.

The coordinates of a local block are depending on the respective process grid. In our case the local coordinates have to be transformed from one processing grid to another. The grid in which we compute the matrix entries line-by-line is also just a line of processes. The grid in which we compute the matrix-matrix multiplications and also solve the linear system of equations may be two dimensional.

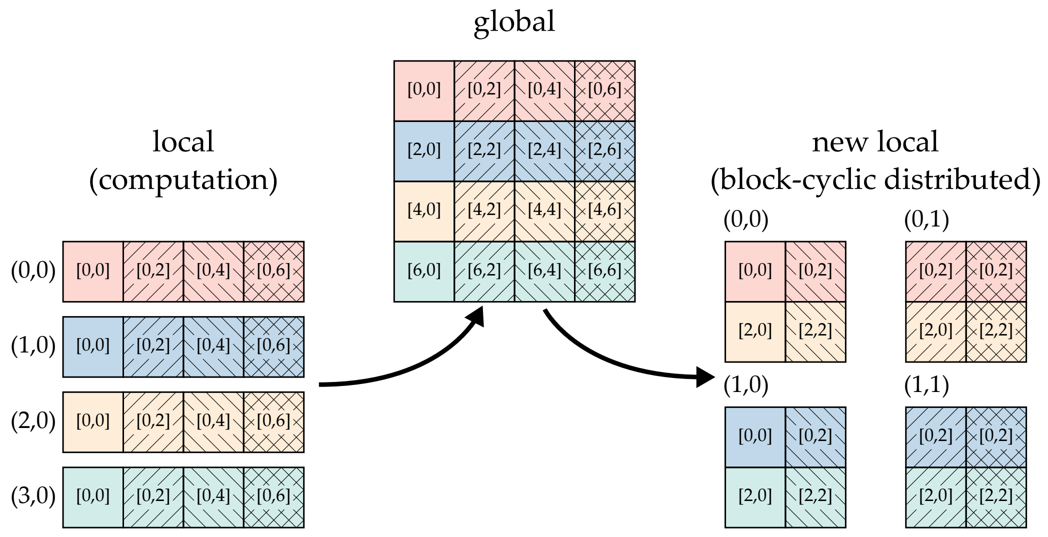

Figure 5.4: *Distribution scheme of a block of the design matrix* **A***: from local over global to local coordinates of a* 2×2 *processing grid.*

Starting from the global coordinates of a block, it is easy to calculate its local coordinates. The same applies the other way round. As the calculation is the same for both dimensions, functions are only implemented for one dimension and are called twice by a 2D wrap around function. The one-dimensional function `glob2loc` calculates the local coordinate of a block from its global coordinate. The function `loc2glob` reverses this calculation and computes the global coordinate of a local block. The sorce code to the one-dimensional functions and their according 2D wrappers can be found in the files `distribute.h` and `distribute.c` which are provided in the appendix, section A.4.

The BLACS library provides functions to send and receive a matrix block. For these functions it is necessary to specify at the sending process which is the receiving process and the other way round. The functions are called `DGESD2D` for sending, respectively `DGERV2D` for receiving. But the receiving process knows the sending process by its number only, not by its processing grid coordinates. Hence, the BLACS function `BLACS_PCOORD` is needed to calculate the coordinates of this process first before calling the function for receiving.

And again, the fact has to be considered that Fortran-programs organize the elements of a matrix row-by-row in the memory and C-programs do it line-by-line. Hence, the elements of a block have to be reorganized according to the Fortran scheme during the redistribution task to achieve a block-cyclic distribution of the matrix.

`DGESD2D`	point to point send

```
Cdgesd2d(ctxt, m, n, A, ldA, rdest, cdest);
```

with

`ctxt`	array of integer, BLACS context
`m, n`	integer, number of matrix rows/columns to be operated on
`A`	pointer to the first element of the submatrix to be sent
`ldA`	integer, distance of two elements in a matrix row
`rdest, cdest`	integer, row/column of the receiving process

DGERV2D	point to point receive

```
Cdgerv2d(ctxt, m, n, A, ldA, rsrc, csrc);
```

with

`ctxt`	array of integer, BLACS context
`m`, `n`	integer, number of matrix rows/columns to be operated on
`A`	pointer to the first element of the array to receive the incomming matrix into
`ldA`	integer, distance of two elements in a matrix row
`rsrc`, `csrc`	integer, row/column of the sending process

BLACS_PCOORD	calculates the grid coordinates of a process

```
Cblacs_pcoord(ctxt, pnum, prow, pcol);
```

with

`ctxt`	array of integer, BLACS context
`pnum`	integer, number of the process
`prow`, `pcol`	pointer to an integer, row/column of the process in the processing grid

5.3.4 Normal equations system setup

For setting up the normal equations system, the PBLAS library contains the equivalents of the functions of the BLAS library for distributed computing. The function `PDGEMV` is used to compute the vector **b** and the function `PDSYRK` to compute the normal equations matrix **N**. Both function expect the matrices and vectors to be block-cyclic distributed. This was handled in the previous section.

Although both functions work similar to their BLAS pendants, the function calls need more parameters to describe how the matrices and vectors are distributed over the processing grid. Again, the descriptions of the functions are reduced to the case needed in the program.

PDSYRK	$\mathbf{C} \leftarrow \alpha \mathbf{A}^{\mathsf{T}} \mathbf{A} + \beta \mathbf{C}$

```
pdsyrk_("L", "T", n, k, α, locA, iA, jA, descA,
        β, locC, iC, jC, descC);
```

with

α, β	scalars
`n`, `k`	integers, dimensions of the distributed matrices (global)
`locA`	local part of distributed matrix `A` (matrix `A` is of size `k` × `n`)
`iA`, `jA`	integer, first row/column of the distributed matrix `A` in the processing grid
`descA`	the array descriptor of the distributed matrix `A`
`locC`	local part of distributed matrix `C` (matrix `C` is of size `n` × `n`)
`iC`, `jC`	integer, first row/column of the distributed matrix `C` in the processing grid
`descC`	the array descriptor of the distributed matrix `C`

All parameters are pointers.

PDGEMV	$\mathbf{y} \leftarrow \alpha \mathbf{A}^{\mathsf{T}} \mathbf{x} + \beta \mathbf{y}$

```
pdgemv_("T", m, n, α, locA, iA, jA, descA,
        locx, ix, jx, descx, incx, β, locy, iy, jy, descy, incy);
```

with

α, β	scalars

`m`, `n`	integers, matrix dimensions
`locA`	local part of distributed matrix `A` (matrix `A` is of size `m` × `n`)
`iA`, `jA`	integer, first row/column of the distributed matrix `A` in the processing grid
`descA`	the array descriptor of the distributed matrix `A`
`locx`	local part of distributed vector `x` (vector `x` is of size `m` × `1`)
`ix`, `jx`	integer, first row/column of the distributed vector `x` in the processing grid
`descx`	the array descriptor of the distributed vector `x`
`incx`	integer, increment to reach the address of the next element of vector `x`
`locy`	local part of distributed vector `y` (vector `y` is of size `n` × `1`)
`iy`, `jy`	integer, first row/column of the distributed vector `y` in the processing grid
`descy`	the array descriptor of the distributed vector `y`
`incy`	integer, increment to reach the address of the next element of vector `y`

All parameters are pointers.

5.3.5 Normal equations system solution

The normal equations system is solved by `PDPOSV` which is the ScaLAPACK pendant to the LAPACK routine `DPOSV`. Again, the distributed function needs more parameters to describe how the matrix and vector are distributed.

After solving the normal equations system, also the vector containing the estimated coefficients is found to be distributed over the processing grid. To output the results, it is the best to gather them at one process first, and then writing the results to a file. The gathering can be done with the routine `PDGEMR2D` which is used to redistribute a matrix from one processing grid to another.

PDPOSV solve a normal equations system

```
pdposv_("L", n, nrhs, locN, iN, jN, descN,
        locb, ib, jb, descb, info);
```

with

`n`	integer, the number of linear equations
`nrhs`	the number of right hand sides in `b`
`locA`	local part of the lower triangular part of the symmetric distributed matrix `A` (matrix `A` is of size `n` × `n`)
`iA`, `jA`	integer, first row/column of the distributed matrix `A` in the processing grid
`descA`	the array descriptor of the distributed matrix `A`
`locb`	local part of the distributed matrix `b` (matrix `b` is of size `n` × `nrhs` and can hold mutiple right hand side vectors), holds the results on return!
`ib`, `jb`	integer, first row/column of the distributed matrix `b` in the processing grid
`descb`	the array descriptor of the distributed matrix `b`
`info`	returns status information

All parameters are pointers.

PDGEMR2D redistribute a matrix to a different processing grid

```
pdgemr2d_(m, n, locA, iA, jA, descA, locB, iB, jB, descB, ctxt);
```

with

`m`, `n`	integer, the number of rows/columns od the distributed matrix

`locA`	local part of the distributed matrix `A` (matrix `A` is of size `m` × `n`)
`iA`, `jA`	integer, first row/column of the distributed matrix `A` in the processing grid
`descA`	the array descriptor of the distributed matrix `A`
`locB`	local part of the distributed matrix `B` (matrix `B` is also of size `m` × `n`)
`iB`, `jB`	integer, first row/column of the distributed matrix `B` in the processing grid
`descB`	the array descriptor of the distributed matrix `B`
`ctxt`	the BLACS context enfolding at least all processors included in either `A` context or `B` context.

All parameters are pointers.

5.4 Further considerations

5.4.1 Block size of a distributed matrix

The size of the blocks, into which a distributed matrix is split, also matters. For each transferred block exists some head information for communication purposes. When the block size is chosen too small, many blocks have to be transferred. With such a setting, the communication overhead will be high which in turn results in higher runtime of the program. On the other hand, when the block size is chosen too big, during the transfer of one block other communication gets blocked eventually. This results in also slowing down the program.

After testing it was found that a blocksize between 48×48 and 104×104 results in significantly lower runtime with a slight optimum at a size of 88×88. This also is shown in figure 5.5. For the graph on the left hand side a more coarse stepping was used for the blocksize (multiples of 2) with some refinements in between. The right hand side shows the part where the runtime is optimal with further refinement steps.

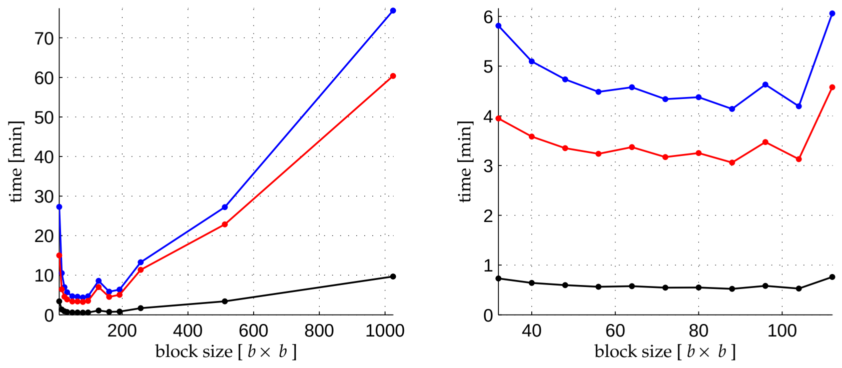

Figure 5.5: *Runtime depending on the block size of the distributed matrices (black: real time; red: vector time; blue: user time).*

The tests were done on the SX-8 only but on a complete node in exclusive use. It is still to test, if for the SX-9 another block size would be better suited. Nonetheless, it was decided to split the distributed matrices in blocks of the size 88×88 on both SX systems.

5.4.2 Linking a program which includes ScaLAPACK

In the appendix, the makefiles for the program versions are provided. Especially the makefile for the MPI version of the program needs some further comments.

After compiling a program, it is linked with the necessary libraries in order to produce an executable file. The C-compiler is usually capable to link a program (for linking a library the parameter `-l` is available). Usually, the order in which the libraries are specified to the linker does not matter. But for the ScaLAPACK library with its adjacent libraries like BLACS, LAPACK and BLAS a specific order has to be followed.

On the GIS-Cluster the order would be as follows:

```
  -lscalapack -lblacs -lblacsCinit -lblacsF77init -lblacs -llapack -lcblas -lblas -lgfortran
```

Please note that `-lblacs` appears twice! Linking for the SX systems is similar but the names differ: `-lblacsF77init` is exchanged with `-lblacsF90init` and the Fortran library `-lgfortran` is exchanged with `-f90lib`. The other libraries are named equally.

5.5 Runtime results

The local computer and the frontends do not run MPI and the 10BaseT Ethernet network of the GIS-Cluster is just too slow. Hence it is not possible to get reasonable results on that cluster. The tests to get MPI runtime results were done at the SX systems. Because of the long queues on the SX systems, it was not possible to execute all the prepared tests.

5.5.1 Varying number of CPUs

The tests were started with two processes which are the minimum for a program which uses MPI. The single processor result from the OpenMP version is used here as a reference. With MPI it is possible to use several nodes of the SX systems. Hence on both systems up to 32 CPUs were used. This equals four nodes on the SX-8 and two nodes at the SX-9.

Figures 5.6 and 5.7 displey the results of the time measurements. For the MPI version a new timer is introduced which measures the duration of the design matrix distribution. The seemingly weird behaviour in the figures for up to eight CPUs is due to the processing grid layout. For the SX-8 this is overlayed by the usual problem with the parallel job mode. In both figures the measured time is low when the prosessing grid is more quadratic. For odd numbers the processing grid is linear (e. g. 1×7 for seven CPUs) which is inefficient for the routines of PBLAS and ScaLAPACK. The design matrix assembly is still following the expected behaviour.

Compared to the SX-9, the MPI version of the program runs on the SX-8 with a higher efficiency for more CPUs. The SX-9 seems to have a problem here. A comparison with the OpenMP results shows that the MPI version needs around four to five times as long as the OpenMP version, which is not acceptable.

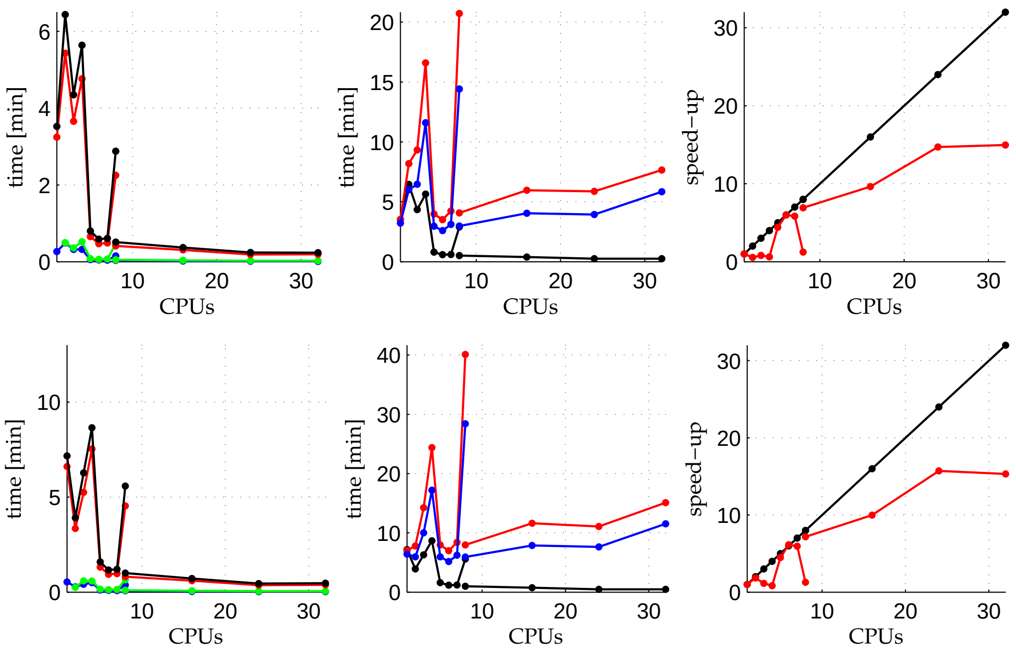

Figure 5.6: Distributed memory version with MPI on the SX-8. Left: design matrix assembly (blue); distribution of the design matrix (green); normal equations system setup (red); overall time (black). Middle: user time (red); vector time(blue); real time (black). Right: ideal speed-up (black); true speed-up (red). Parameters: $l_{\max} = 50$, top: $n = 250\,000$, bottom: $n = 500\,000$. Please note the gap in the graphs; here the job mode is switched to exclusive use.

5.5.2 Large-scale problems

The previous test was repeated for a higher degree and order of $l_{\max} = 200$ and $n = 500\,000$ lines of data to analyze if the results depend on a specific degree and order. The test were done on the SX-9. The results are depicted in figure 5.8.

Another test was done to analyze if the program's efficiency gets better for higher $l_{\max}$. Like the OpenMP version, the MPI version was tested up to a degree and order of $l_{\max} = 200$ on the SX-9, the results are shown in figure 5.9. Now the situation differs. For $l_{\max} = 200$ the MPI version is about 10 % faster than the OpenMP version. It seems that the MPI version is better suited for large scale problems. A test for $L = 250$ was launched for the MPI and the OpenMP version to analyze this further, but due to problems of the scheduler after a waiting time of two days the queue was reset, hence those jobs were not executed.

This better performance can also be seen in table 5.4 where a direct comparison of OpenMP and MPI is possible. The results show that MPI is more efficient than OpenMP for large-scale problems and the other way round for lower-scale problems. The turning point is between a problem size of $l_{\max} = 150, \ldots, 200$.

A further test was conducted on the SX-8, starting with one complete node (8 CPUs) up to four nodes (32 CPUs in total). The intension of this test was to analyze how the efficiency of the program will be for different $l_{\max}$ and a different number of nodes. It can be seen in the graph for the real time in figure 5.10 that – like expected – the more nodes are used the faster the computation is done. For computations which run in parallel, also the user time

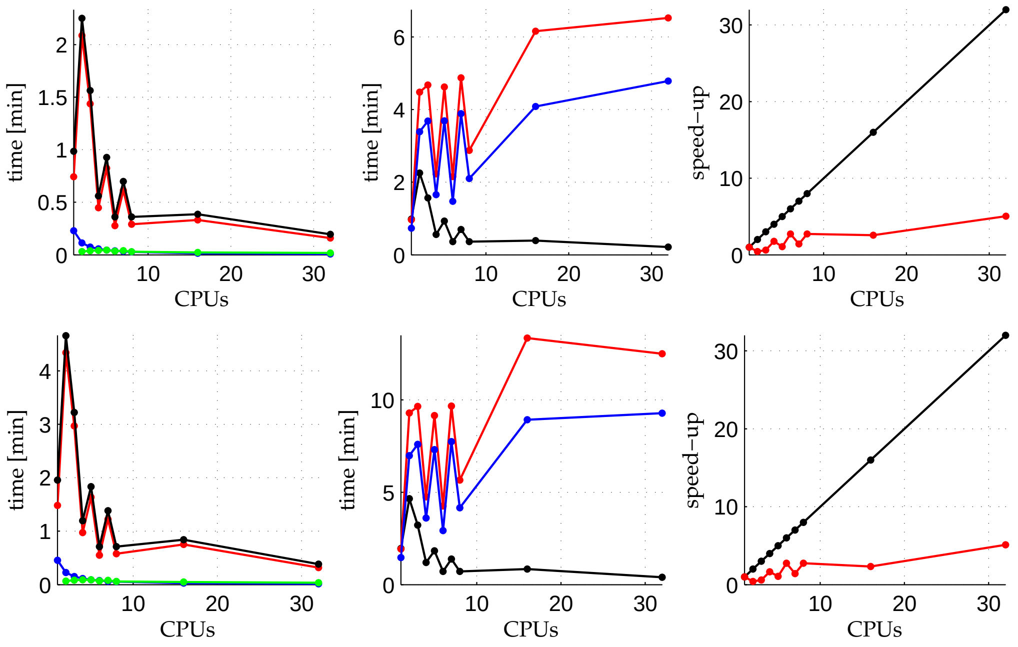

Figure 5.7: *As in figure 5.6, but for the SX-9.*

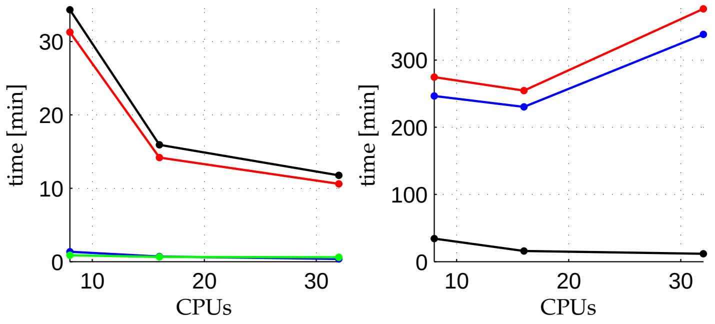

Figure 5.8: *As in figure 5.6, but for the SX-9 and with* $l_{\max} = 200$.

(accumulated time over all used processors) is important. For the computation to be at an equal efficiency with a variing number of processors the user time should be equal. A comparison of the user times reveals, that this matches for one and two nodes only. For three respectively four nodes the user time increases. In terms of efficiency the computation should be done incorporationg only two nodes at maximum of the SX-8. The increasing user time is mostly due to the increasing communication between the processes.

5.5.3 Numerical precision of the computation

Once again, it is verified that the numerical precision is within acceptable range. The results are displayed in figure 5.11. The graphics are visually identical to the previous test with OpenMP (see section 4.3.3). Hence, also the same conclusions can be drawn that the program produces good results in terms of spherical harmonic coefficients recovery.

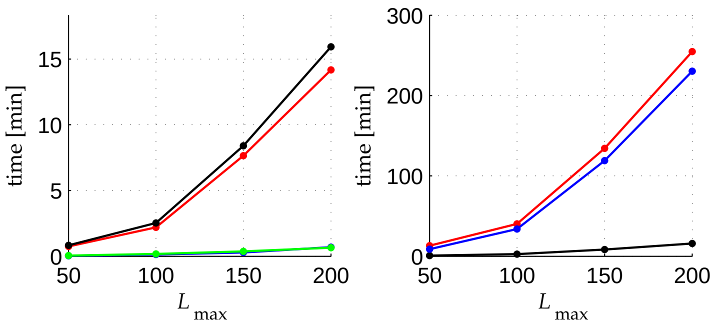

Figure 5.9: *Distributed memory version with MPI on the SX-9. Colors are according to figure 5.6, fixed parameters: 16 CPUs, $n = 500\,000$.*

l_{max}	— OpenMP —		— MPI —		speed-up
	real time [s]	user time [s]	real time [s]	user time [s]	(OpenMP : MPI)
50	10	142	50	791	0.20
100	81	1272	151	2414	0.54
150	352	5596	504	8055	0.70
200	1070	17081	955	15278	1.12

Table 5.4: *Comparison of the parallelization between OpenMP and MPI, SX-9, $n = 500\,000$, 16 CPUs.*

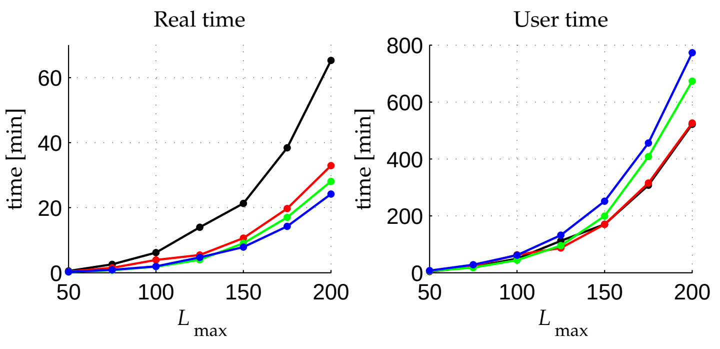

Figure 5.10: *SX-8: runtime depending on maximum degree and order l_{max} for a different number of whole nodes. One node (black); two nodes (red); three nodes (green); four nodes (blue); fixed parameters: $n = 256\,000$.*

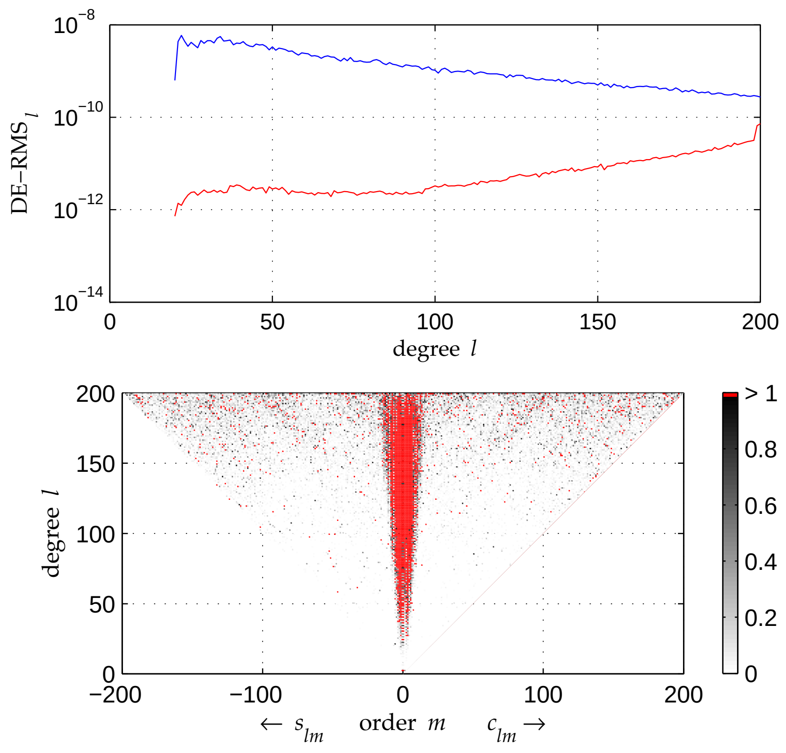

Figure 5.11: *MPI version, $L = l_{\max} = 200$. Left: EGM96 signal (blue), DE-RMS values (red). Right: Relative empirical errors.*

Chapter 6

Conclusions and outlook

Several achievements could be made. First, the program was optimized, which offered a huge gain in efficiency. The biggest speed-up was achieved by rearranging the algorithm to compute the Legendre function which also involved changing the procedure for setting up the design matrix. Some minor improvement like converting the data from ASCII to binary became necessary because of the different computer architectures. After the possibilities for further optimization got very small, the program was changed to incorporate parallel computations for a further speed-up of the BLAS/LAPACK part of the program. This was done with two techniques.

The first technique used was OpenMP which is mostly a extension to the compiler. After compilation, the program can be executed on shared memory systems. A clear advantage of OpenMP is its simplicity. It is easy to use in an existing program, also the compiled program can be called like any other program from the command line. The disadvantages are that the program is limited to one computing node with shared memory and that the computing node needs a sufficient amount of memory to hold the normal equations matrix.

The second technique used was MPI which is more complex to use, as communication is not done automatically as with OpenMP. An advantage of MPI is that it is possible to use both, shared memory and distributed memory systems. Opposed to OpenMP, it is possible to use more than one computing node with a large number of processors. Also it got possible to distribute the normal equations matrix over all the used nodes. Each of those nodes needs only to hold part of the matrix. Because of that, the memory demand per node is divided by the number of used nodes. A disadvantage is that the program has to be run with a tool like `mpirun` which provides the necessary interface to the communication network.

For the distributed memory version of the program it became necessary to include ScaLAPACK with its associated libraries. Here MPI and a computing grid were initialized, an optimal blocksize for distributing the matrices was found and also an algorithm to distribute the matrix was developed.

The effort done for optimization and parallelization helped to improve the efficiency of the program considerably. Also attention was paid to make the program as modular as possible, so that future changes and extensions can be implemented easily. The program source code is provided in the appendix.

A comparison between the OpenMP and MPI versions indicated that the OpenMP version is better suited for lower $l_{\max}$ while the MPI version shows better efficiency for higher $l_{\max}$. This leads to an idea to combine both concepts and have OpenMP doing the intra-node work and MPI the inter-node communication. Maybe this could improve the speed even further, especially if even more nodes get involved with the computations. This could be the next step on this project.

Some further improvement might be done for the MPI version at the design matrix setup. Here, it would be possible to save the matrix elements already in the Fortran scheme. This would need a new computing scheme for the indices, but on the other hand, the reordering step which is necessary at the moment could be skipped. It remains to be tested if this would lead to a gain in speed. Another possibility would be to completely eliminate the step of the design matrix distribution and to compute the values at the process which also stores this part of the matrix. Maybe this could also lead to a gain in speed. On the other hand, this would mean to rewrite a bigger part of the program.

Appendix A

Source code

A.1 Block-cyclic distribution – an example

The following code demonstrates the example of the block-cyclic distribution of section 5.1.3.

bsp_block-cyclic.c

```
#include <stdio.h>
#include <stdlib.h>

void glob2loc(int glob, int b, int prowcol, int *loc, int *mrowcol) {
  // INPUT:
  //   glob ...... global row/column coordinate of matrix element
  //   b ......... row/column size of block matrix
  //   prowcol ... row/column size of process grid
  // OUTPUT:
  //   loc ....... local row/column coordinate of matrix element
  //   mrowcol ... row/column coordinate of process which holds the data

  int tmp;

  *mrowcol = (glob / b) % prowcol;
  tmp      = glob / (b * prowcol);
  *loc     = tmp * b + glob % b;
}

void glob2loc2D(int mglob, int nglob, int mb, int nb, int prow, int pcol,
                int *mloc, int *nloc, int *mrow, int *mcol) {
  // INPUT:
  //   mglob, nglob ... global coordinates of matrix element
  //   mb, nb ......... size of block matrix
  //   prow, pcol ..... size of process grid
  // OUTPUT:
  //   mloc, nloc ..... local coordinates of matrix element
  //   mrow, mcol ..... coordinates of process which holds the data

  glob2loc(mglob, mb, prow, mloc, mrow);
  glob2loc(nglob, nb, pcol, nloc, mcol);
}

int main(int argc, char *argv[]) {
  int m = 8, n = 5;              // 7 x 7 matrix
  int prow = 2, pcol = 2;        // 2 x 3 process grid
  int mb = 2, nb = 2;            // 2 x 2 matrix block
  int myrow, mycol, mrow, mcol; // coordinates of process on processing grid
  int mglob, nglob, mloc, nloc; // global/local coordinates of matrix element

  int *A = NULL;
  int i;

  A = (int*)malloc(m * n * sizeof(int)); // allocate memory for matrix
  for (i = 0; i < (m * n); i++) { // initialize matrix
    A[i] = i;
  }

  printf("matrix␣size␣.....␣%d␣x␣%d\n", m, n);
  printf("process␣grid␣....␣%d␣x␣%d\n", prow, pcol);
  printf("block␣size␣......␣%d␣x␣%d\n", mb, nb);


  for (mglob = 0; mglob < m; mglob += mb) { // run over global matrix elements
    for (nglob = 0; nglob < n; nglob += nb) { // increase for one block

      for (myrow = 0; myrow < prow; myrow++) { // simulate processing grid
        for (mycol = 0; mycol < pcol; mycol++) {

          glob2loc2D(mglob, nglob, mb, nb, prow, pcol, // compute local coordinates
                     &mloc, &nloc, &mrow, &mcol);

          // if the computed process coordinates are the same as "this" process:
          if ((myrow == mrow) && (mycol == mcol)) {
            printf("process␣(%d,␣%d)␣:␣global␣[%d,␣%d]␣=>␣local␣[%d,␣%d]\n",
                   mrow, mcol, mglob, nglob, mloc, nloc);
          }
        }
      }
    }
  }
  return 0;
}
```

A.2 Conversion of ASCII to binary data

makefile

```
# data conversion ASCII --> binary (local computer)
main:
        gcc -o ASCII2bin ASCII2bin.c dataio.c commandline.c
```

ascii2bin.c

```
#include <stdio.h>
#include <string.h>

#define ASCII2BIN

#include "dataio.h"
#include "commandline.h"

char out_sst_fil[][10] = {"sst_time", "x1", "x2", "x3", "x1p", "x2p", "x3p",
                          "x1pp", "x2pp", "x3pp", "lambda", "phi", "r"};
char out_sgg_fil[][10] = {"sgg_time", "Txx", "Txy", "Txz", "Tyy", "Tyz", "Tzz", "I1", "I2"
    , "I3"};


///////////////////////////////////////////////
int main(int argc, char* argv[]) {

  // commandline parameters
  struct t_cmd_line cmd_parameters;

 // data input parameters
```

```
double *time_sst = NULL,                    // sst-file
  *x1 = NULL, *x2 = NULL, *x3 = NULL,
  *x1p = NULL, *x2p = NULL, *x3p = NULL,
  *x1pp = NULL, *x2pp = NULL, *x3pp = NULL,
  *r = NULL, *phi = NULL, *lambda = NULL;
double *time_sgg = NULL,                    // sgg-file
  *Txx = NULL, *Txy = NULL, *Txz = NULL,
  *Tyy = NULL, *Tyz = NULL, *Tzz = NULL,
  *I1 = NULL, *I2 = NULL, *I3 = NULL;
double *sst_tmp[13] = {NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
     NULL, NULL};
double *sgg_tmp[13] = {NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL};

// dimension parameters
int N = 0;          // read all lines of data
int M = 1;          // number of initial memory cells
int numread;        // number of read lines of data

FILE *sst_fil, *sgg_fil, *fil;
int sst_chunksiz, sgg_chunksiz;
int i, j;
double tmp[20];
char filname[200];


///// INTERPRET COMMANDLINE PARAMETERS /////
cmd_parameters = interprete_commandline(argc, argv);
printf("␣␣-->␣command␣line␣parameters\n");
printf("␣␣␣␣␣␣␣SST␣file␣...............␣%s\n", cmd_parameters.in_sst_filename);
printf("␣␣␣␣␣␣␣SGG␣file␣...............␣%s\n", cmd_parameters.in_sgg_filename);


// EINLESEN DER DATEN
printf("␣␣-->␣input␣data\n");
printf("␣␣␣␣␣␣␣SST␣data:␣%s\n", cmd_parameters.in_sst_filename);
printf("␣␣␣␣␣␣␣SGG␣data:␣%s\n", cmd_parameters.in_sgg_filename);
if ((numread = input_sst_sgg(N, M,
                             cmd_parameters.in_sst_filename, &sst_tmp[0], &sst_tmp[1], &
                                 sst_tmp[2],
                             &sst_tmp[3], &sst_tmp[4], &sst_tmp[5], &sst_tmp[6],
                             &sst_tmp[7], &sst_tmp[8], &sst_tmp[9], &sst_tmp[10],
                             &sst_tmp[11], &sst_tmp[12],
                             cmd_parameters.in_sgg_filename, &sgg_tmp[0], &sgg_tmp[1], &
                                 sgg_tmp[2],
                             &sgg_tmp[3], &sgg_tmp[4], &sgg_tmp[5], &sgg_tmp[6],
                             &sgg_tmp[7], &sgg_tmp[8], &sgg_tmp[9])) <= 0) {
  return -1; // something went wrong
}
printf("␣␣␣␣␣␣␣#␣requested:␣%d␣--␣#␣read:␣%d\n", N, numread);

N = numread;
printf("␣␣-->␣maximal␣usable␣data␣entries␣n␣=␣%d\n", N);

// WRITING OF DATA
printf("␣␣-->␣writing␣data␣...\n", N);

for (j = 0; j < 13; j++) {
  sprintf(filname, "%s.bin_%s", cmd_parameters.in_sst_filename, out_sst_fil[j]);
  printf("␣␣␣␣␣␣␣-␣%s\n", filname);

  fil = fopen(filname, "wb");
  for (i = 0; i < N; i++) {
    tmp[0] =  swapEndian_double(sst_tmp[j][i]);
    fwrite(&tmp, sizeof(double), 1, fil);
  }
  fclose(fil);
}
```

```
  for (j = 0; j < 10; j++) {
    sprintf(filname, "%s.bin_%s", cmd_parameters.in_sgg_filename, out_sgg_fil[j]);
    printf("␣␣␣␣␣␣␣-␣%s\n", filname);

    fil = fopen(filname, "wb");
    for (i = 0; i < N; i++) {
      tmp[0] = swapEndian_double(sgg_tmp[j][i]);
      fwrite(&tmp, sizeof(double), 1, fil);
    }
    fclose(fil);
  }

  printf("␣␣␣␣␣␣...␣done.\n", N);

  return 0;
}
```

A.3 Shared memory version – OpenMP

makefile

```
CC_LOCAL = gcc  # local computer
CC_ASAMA = icc  # ASAMA
CC_SX8   = sxcc # SX8

FIL_SX8  = main.c commandline.c dataio.c legendre.c compute.c performance.c
OBJ      = main.o commandline.o dataio.o legendre.o compute.o performance.o
BIN      = main main_asama main_sx8 main_sx9

LIB_LOCAL = -lrt -lm -L/usr/local/ATLAS/lib/Linux_HAMMER64SSE2_2 -llapack -lptcblas -
    latlas # for parallel computing: -lptcblas instead of -lcblas
LIB_ASAMA = -lrt -lm -L/opt/MathKeisan/MKL/lib/64 -lmkl_lapack -lmkl_ipf
LIB_SX8   = -lcblas -llapack -lparblas -f90lib  # for parallel computing: -lparblas
    instead of -lblas

CMP_LOCAL = -O3 -fopenmp -DOpenMP
CMP_ASAMA = -O2 -openmp -I/opt/MathKeisan/MKL/include -DOpenMP
CMP_SX8   = -Popenmp -size_t64 -w none -DOpenMP

INC_SX8   = -I/SX/opt/mathkeisan/inst/include

# local computer
main:
        $(CC_LOCAL) $(CMP_LOCAL) -c $(patsubst %.o, %.c, $(OBJ))
        $(CC_LOCAL) $(CMP_LOCAL) -o main $(OBJ) $(LIB_LOCAL)

run: main
        time ./main --maxlm=10 --lines=10000

# ASAMA
asama:
        $(CC_ASAMA) $(CMP_ASAMA) -c $(patsubst %.o, %.c, $(OBJ)) -DASAMA
        $(CC_ASAMA) $(CMP_ASAMA) -o main_asama $(OBJ) $(LIB_ASAMA)

asama_run: asama
        time ./main_asama --maxlm=10 --lines=10000

# SX8
sx8:
        $(CC_SX8) $(CMP_SX8) $(INC_SX8) -c $(FIL_SX8) -DSX8
        $(CC_SX8) $(CMP_SX8) -o main_sx8 $(OBJ) $(LIB_SX8)

# SX9
```

```
sx9:
        $(CC_SX8) $(CMP_SX8) $(INC_SX8) -c $(FIL_SX8) -DSX8
        $(CC_SX8) $(CMP_SX8) -o main_sx9 $(OBJ) $(LIB_SX8)

#miscellaneous
clean:
        rm -f $(BIN) $(OBJ)

.PHONY: main run asama asama_run sx8 sx9 clean
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#if defined(OpenMP)
  #include <omp.h>
#endif

#if defined(SX8)
  #include <cblas.h>
#elif defined(ASAMA)  // ASAMA / A1:
  #include <mkl_types.h>
  #include <mkl_cblas.h>
  #include <mkl_lapack64.h>
  #include <getopt.h>
#else                 // LOCAL:
  #include "/usr/local/ATLAS/include/atlas_enum.h"
  #include "/usr/local/ATLAS/include/cblas.h"
  #include "/usr/local/ATLAS/include/clapack.h"
  #include <getopt.h>
#endif

#include "commandline.h"
#include "dataio.h"
#include "legendre.h"
#include "performance.h"

const double pi = 3.1415926535897932384626433832795028841971693993751058;

const double R  = 6378137.0;         // mean earth radius
const double GM = 3.986004418e+14;   // geocentric constant

static double *Bl = NULL;
static double *Wmm = NULL, *Wml_1 = NULL, *Wml_2 = NULL; // for initial values of Legendre
     function
static double *nL = NULL;
#ifdef OpenMP
  #pragma omp threadprivate(nL, Bl)
#endif

/*************************************************************
   MAIN FUNCTION
*************************************************************/

int main(int argc, char* argv[]) {

  // commandline parameters
  struct t_cmd_line cmd_parameters;

  FILE* fil;        // output
  double clm, slm;

  // data storage
  double *time_sst = NULL,                 // sst-file
```

```
    *x1 = NULL, *x2 = NULL, *x3 = NULL,
    *x1p = NULL, *x2p = NULL, *x3p = NULL,
    *x1pp = NULL, *x2pp = NULL, *x3pp = NULL,
    *r = NULL, *phi = NULL, *lambda = NULL;
  double *time_sgg = NULL,                 // sgg-file
    *Txx = NULL, *Txy = NULL, *Txz = NULL,
    *Tyy = NULL, *Tyz = NULL, *Tzz = NULL,
    *I1 = NULL, *I2 = NULL, *I3 = NULL;

  // dimension parameters
  int A_siz = 16;    // memory size (in MiB) for block of design matrix to build normal
      equations matrix

  int numread;   // number of read lines of data
  int numlp, cpus;

  int i, l, m, idx; // counter

  int koeffs, N_siz, A_blk_siz;        // number of coefficients, sizes of normal
      equations matrix and design matrix block
  int blk, blk_cnt, b;
  double *A_row = NULL;                // block of design mtrix
  double *N_mat = NULL, *x_vec = NULL; // normal equations matrix and vector of
      obeservations/coefficients

  double GMRRR;


#if defined(ASAMA) || defined(SX8)
  char uplo = 'L';
  int  nrhs = 1, info;
#endif

  // performance monitoring
  Ttimer timer_data, timer_ALL, timer_A, timer_N;

 ///// INTERPRET COMMANDLINE PARAMETERS /////
  cmd_parameters = interprete_commandline(argc, argv);
  printf("␣␣-->␣command␣line␣parameters\n");
  printf("␣␣␣␣␣␣␣SST␣file␣................␣%s\n", cmd_parameters.in_sst_filename);
  printf("␣␣␣␣␣␣␣SGG␣file␣................␣%s\n", cmd_parameters.in_sgg_filename);
  printf("␣␣␣␣␣␣␣max␣lm␣..................␣%d\n", cmd_parameters.max_lm);
  printf("␣␣␣␣␣␣␣lines␣...................␣%d\n", cmd_parameters.N);
  printf("␣␣␣␣␣␣␣out␣file␣................␣%s\n", cmd_parameters.out_filename);
  printf("␣␣␣␣␣␣␣local␣matrix␣block␣size␣.␣%d\n", cmd_parameters.block_m);

  // here we start performance monitoring
  reset_timer(&timer_ALL); start_timer(&timer_ALL);
  reset_timer(&timer_data);
  reset_timer(&timer_A);
  reset_timer(&timer_N);

  //// DATA INPUT ////
  start_timer(&timer_data);
  printf("␣␣-->␣input␣data\n");
  printf("␣␣␣␣␣␣␣SST␣data:␣%s\n", cmd_parameters.in_sst_filename);
  printf("␣␣␣␣␣␣␣SGG␣data:␣%s\n", cmd_parameters.in_sgg_filename);
  if ((numread = input_sst_sgg_bin(cmd_parameters.N, cmd_parameters.block_m,
                                   cmd_parameters.in_sst_filename, &time_sst, &x1, &x2, &
                                       x3, &x1p, &x2p, &x3p, &x1pp, &x2pp, &x3pp, &lambda,
                                        &phi, &r,
                                   cmd_parameters.in_sgg_filename, &time_sgg, &Txx, &Txy,
                                       &Txz, &Tyy, &Tyz, &Tzz, &I1, &I2, &I3)) <= 0) {
    return -1; // something went wrong
  }

  printf("␣␣␣␣␣␣␣#␣requested:␣%d␣--␣#␣read:␣%d\n", cmd_parameters.N, numread);
  cmd_parameters.N = numread; // update the count of datalines with the correct value
  printf("␣␣-->␣maximal␣usable␣data␣entries␣n␣=␣%d\n", cmd_parameters.N);
  stop_timer(&timer_data);

  //// INITIALIZING COMPUTATIONS ////
  // set up of normal equations matrix N and vector A'y
  koeffs    = (cmd_parameters.max_lm + 1) * (cmd_parameters.max_lm + 1) - 3; // c_1m, s_1m
       are not estimated
  N_siz     = koeffs * koeffs;                 // size of normal equations matrix
  A_blk_siz = (A_siz * 1024 * 1024) / (koeffs * sizeof(double));  // block size for
      setting up normal equations system
  if (A_blk_siz > cmd_parameters.N) {
    A_blk_siz = cmd_parameters.N;
  }

  printf("␣␣-->␣making␣N-Matrix,␣A'y-Vektor\n");
  printf("␣␣␣␣␣␣␣size␣A-Block␣:␣%8d␣*␣double␣=␣%6.2f␣MiB␣␣(%d␣lines)\n", A_blk_siz *
      koeffs, 1.0 * A_blk_siz * koeffs * sizeof(double) / 1024 / 1024, A_blk_siz);
  printf("␣␣␣␣␣␣␣size␣A'y␣␣␣␣␣:␣%8d␣*␣double␣=␣%6.2f␣MiB␣␣(max.␣degree/order:␣%d)␣\n",
      koeffs, 1.0 * koeffs * sizeof(double) / 1024 / 1024, cmd_parameters.max_lm);
  printf("␣␣␣␣␣␣␣size␣N␣␣␣␣␣␣␣:␣%8d␣*␣double␣=␣%6.2f␣MiB\n", N_siz, 1.0 * N_siz * sizeof(
      double) / 1024 / 1024);

  A_row = (double*)calloc(koeffs * A_blk_siz, sizeof(double));
  N_mat = (double*)calloc(N_siz,  sizeof(double));
  x_vec = (double*)calloc(koeffs, sizeof(double));


#ifdef OpenMP
  #pragma omp parallel
  {
    cpus = omp_get_num_threads();
  }
  printf("␣␣-->␣threads␣used:␣%d\n", cpus);
#endif

  printf("␣␣-->␣blocks␣to␣compute:␣%d\n", (int)ceil((double)cmd_parameters.N / (double)
      A_blk_siz));

  blk = A_blk_siz; blk_cnt = 0;
  numlp = (cmd_parameters.max_lm + 1) * cmd_parameters.max_lm / 2 + cmd_parameters.max_lm
      + 1;

#ifdef OpenMP
  #pragma omp parallel
#endif
  {
    nL = (double*)malloc(sizeof(double) * numlp);
    Bl = (double*)malloc(sizeof(double) * cmd_parameters.max_lm);
  }

  Wmm   = (double*)malloc(sizeof(double) * cmd_parameters.max_lm); // initialize Legendre
      functions (beforehand calculations)
  Wml_1 = (double*)malloc(sizeof(double) * cmd_parameters.max_lm * (cmd_parameters.max_lm
      + 1) / 2);
  Wml_2 = (double*)malloc(sizeof(double) * cmd_parameters.max_lm * (cmd_parameters.max_lm
      - 1) / 2);
  init_legendre1kind(cmd_parameters.max_lm, Wmm, Wml_1, Wml_2);

  GMRRR = GM / (R*R*R);

  for (i = 0; i < cmd_parameters.N; i += blk) {
    if (cmd_parameters.N - i >= A_blk_siz) { // calculate actual block size
      blk = A_blk_siz;
    } else {
      blk = cmd_parameters.N - i;
    }
    printf("Block␣#%3d␣-␣size:␣%8d␣%8d", blk_cnt, blk, i);
    if ((laptime_timer(&timer_ALL) > 10.0) && (i != 0))
      printf("␣␣␣␣␣--␣␣␣␣time:␣%1.0fs␣/␣~%1.0fs\n", laptime_timer(&timer_ALL),
          laptime_timer(&timer_ALL) * (cmd_parameters.N - i) / i);
    else
      printf("\n");

    start_timer(&timer_A);

    {
#ifdef OpenMP
  #pragma omp parallel for schedule(guided)
#endif
      for (b = 0; b < blk; b++) {
        compute_A_line(&(A_row[b * koeffs]), cmd_parameters.max_lm, cmd_parameters.N,
            koeffs,
                       i, b,
                       GMRRR, GM, R,
                       lambda, phi, r,
                       Bl, nL, Wmm, Wml_1, Wml_2);
      }

    } // end pragma omp parallel

    stop_timer(&timer_A);
    start_timer(&timer_N);

    // successive building of vector x_vec = A_mat * Tzz
    cblas_dgemv(CblasRowMajor, CblasTrans, blk, koeffs, 1.0, A_row, koeffs, Tzz+i, 1, 1.0,
         x_vec, 1);

    // successive building of normal equations matrix N = A' * A
    // cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, koeffs, koeffs, blk, 1.0,
        A_row, koeffs, A_row, koeffs, 1.0, N_mat, koeffs); // slow
    cblas_dsyrk(CblasRowMajor, CblasUpper, CblasTrans, koeffs, blk, 1.0, A_row, koeffs,
        1.0, N_mat, koeffs); // fast (only triangular matrix)

    stop_timer(&timer_N);

    blk_cnt++;
  }

  // solve "\hat{x} = (\transp{A} * A)^{-1} \transp{A} y"
  // x_vec(new) = N_mat * x_vec(old)
  printf("␣␣-->␣computing:␣x_solve␣=␣(N_mat)^(-1)␣*␣x_vec\n");

#if defined(SX8)
  dposv_(&uplo, &koeffs, &nrhs, N_mat, &koeffs, x_vec, &koeffs, &info);
#elif defined(ASAMA)
  DPOSV(&uplo, &koeffs, &nrhs, N_mat, &koeffs, x_vec, &koeffs, &info);
#else
  clapack_dposv(CblasRowMajor, CblasUpper, koeffs, 1, N_mat, koeffs, x_vec, koeffs);
#endif

  // OUTPUT
  printf("␣␣-->␣some␣coefficients:\n");

  fil = fopen(cmd_parameters.out_filename, "w");

  i = 0;
  for (l = 0; l <= cmd_parameters.max_lm; l++) {
    for (m = 0; m <= l; m++) {

      if (l == 0) {
        clm = x_vec[0]; slm = 0.0;
      } else if (l == 1) {
        clm = 0.0; slm = 0.0;
      } else if (m == 0) {
        clm = x_vec[l - 1]; slm = 0.0;
      } else {
        idx = 2 * (m * (cmd_parameters.max_lm + 1) - m * (m - 1) / 2 + l - m) -
            cmd_parameters.max_lm - 4;
        clm = x_vec[idx]; slm = x_vec[idx + 1];
      }

      fprintf(fil, "%3d␣%3d␣␣%24.16e␣%24.16e\n", l, m, clm, slm);
      if (i <= 10) {
        printf("%3d␣%3d␣␣%24.16e␣%24.16e\n", l, m, clm, slm);
      }
      i++;

    }
  }
  fclose(fil);

  stop_timer(&timer_ALL);

  // performance results
  printf("␣␣-->␣used␣CPU␣time:\n");
  printf("␣␣␣␣␣␣reading␣data␣............␣:␣%8.4f␣s\n", totaltime_timer(&timer_data));
  printf("␣␣␣␣␣␣computing␣of␣A␣..........␣:␣%8.4f␣s\n", totaltime_timer(&timer_A));
  printf("␣␣␣␣␣␣computing␣of␣N␣..........␣:␣%8.4f␣s\n", totaltime_timer(&timer_N));
  printf("␣␣␣␣␣␣OVERALL␣TIME␣............␣:␣%8.4f␣s\n", totaltime_timer(&timer_ALL));

  return 0;
}
```

A.4 Distributed memory version – MPI

makefile

```
CC     = mpicc
CC_SX8 = sxmpicc

OBJ     = main.o commandline.o dataio.o legendre.o compute.o performance.o distribute.o
SRC_SX8 = main.c commandline.c dataio.c legendre.c compute.c performance.c distribute.c
BIN     = mpi_test mpi_sx8 mpi_sx9

LIB = -lrt -L /home/rothms/LIBS -lscalapack -lblacs -lblacsCinit -lblacsF77init -lblacs -
    llapack -lcblas -lblas -lgfortran
LIB_SX8 = -lscalapack -lblacs -lblacsCinit -lblacsF90init -lblacs -llapack -lcblas -lblas
    -f90lib

CMP     = -O3
CMP_SX8 = -size_t64 -w none

INC_SX8 = -I /SX/opt/mathkeisan/inst/include

DEF     = -DCLUSTER -DMPI
DEF_SX8 = -DSX8 -DMPI

# GIS cluster
main:
        $(CC) $(CMP) -c $(patsubst %.o, %.c, $(OBJ)) $(DEF)
        $(CC) $(CMP) -o mpi_test $(OBJ) $(LIB)

run2: main
        time mpirun -np 2 -machinefile ../machines2 ./mpi_test --maxlm 10 --lines=10000
```

```
run4: main
        time mpirun -np 4 -machinefile ../machines2 ./mpi_test --maxlm 10 --lines=10000

# SX systems
sx8:
        $(CC_SX8) $(CMP_SX8) $(INC_SX8) -c $(SRC_SX8) $(DEF_SX8)
        $(CC_SX8) $(CMP_SX8) -o mpi_sx8 $(OBJ) $(LIB_SX8)

sx9:
        $(CC_SX8) $(CMP_SX8) $(INC_SX8) -c $(SRC_SX8) $(DEF_SX8)
        $(CC_SX8) $(CMP_SX8) -o mpi_sx9 $(OBJ) $(LIB_SX8)

# miscellaneous
clean:
        rm -f $(BIN) $(OBJ)

.PHONY: main run2 run4 sx8 sx9 clean
```

distribute.h

```
void glob2loc(int glob, int b, int prowcol, int *loc, int *mrowcol);
  // INPUT:
  //   glob ...... global row/column coordinate of matrix element
  //   b ......... row/column size of block matrix
  //   prowcol ... row/column size of process grid
  // OUTPUT:
  //   loc ....... local row/column coordinate of matrix element
  //   mrowcol ... row/column coordinate of process which holds the data

void glob2loc2D(int mglob, int nglob, int mb, int nb, int prow, int pcol,
                int *mloc, int *nloc, int *mrow, int *mcol);
  // INPUT:
  //   mglob, nglob ... global coordinates of matrix element
  //   mb, nb ......... size of block matrix
  //   prow, pcol ..... size of process grid
  // OUTPUT:
  //   mloc, nloc ..... local coordinates of matrix element
  //   mrow, mcol ..... coordinates of process which holds the data


void loc2glob(int loc, int mrowcol, int b, int prowcol,
              int *glob);
  // INPUT:
  //   loc ....... local row/column coordinate of matrix element
  //   mrowcol ... row/column coordinate of process which holds the data
  //   b ......... row/column size of block matrix
  //   prowcol ... row/column size of process grid
  // OUTPUT:
  //   glob ...... global row/column coordinate of matrix element

void loc2glob2D(int mloc, int nloc, int mrow, int mcol,
                int mb, int nb, int prow, int pcol,
                int *mglob, int *nglob);
  // INPUT:
  //   mloc, nloc ..... local coordinates of matrix element
  //   mrow, mcol ..... coordinates of process which holds the data
  //   mb, nb ......... size of block matrix
  //   prow, pcol ..... size of process grid
  // OUTPUT:
  //   mglob, nglob ... global coordinates of matrix element
```

distribute.c

```
void glob2loc(int glob, int b, int prowcol, int *loc, int *mrowcol) {
  int tmp;
  *mrowcol = (glob / b) % prowcol;
  tmp      = glob / (b * prowcol);
  *loc     = tmp * b + glob % b;
}

void glob2loc2D(int mglob, int nglob, int mb, int nb, int prow, int pcol,
                int *mloc, int *nloc, int *mrow, int *mcol) {
  glob2loc(mglob, mb, prow, mloc, mrow);
  glob2loc(nglob, nb, pcol, nloc, mcol);
}

void loc2glob(int loc, int mrowcol, int b, int prowcol,
              int *glob) {
  *glob = loc * prowcol + mrowcol * b;
}

void loc2glob2D(int mloc, int nloc, int mrow, int mcol,
                int mb, int nb, int prow, int pcol,
                int *mglob, int *nglob) {
  loc2glob(mloc, mrow, mb, prow, mglob);
  loc2glob(nloc, mcol, nb, pcol, nglob);
}
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#if defined(CLUSTER)
  #include <getopt.h>
  #include "../LIBS/cblas.h"
#endif

#if defined(SX8)
  #include <cblas.h>
#endif

#include "mpi.h"

#include "commandline.h"
#include "compute.h"
#include "dataio.h"
#include "legendre.h"
#include "performance.h"
#include "distribute.h"

const double pi = 3.14159265358979323846264338327950288419716939937510582;

const double R  = 6378137.0;         // mean earth radius
const double GM = 3.986004418e+14;   // geocentric constant


/************************************************************
   MAIN FUNCTION
************************************************************/

int main(int argc, char* argv[]) {

  // commandline parameters
  struct t_cmd_line cmd_parameters;

  FILE* fil;        // Ausgabe
  double clm, slm;

  // MPI and BLACS stuff
  int procs, myid;                      // number of processes, process rank
  int ctxt, prow, pcol, myrow, mycol;  // BLACS stuff
  int info;
  MPI_Status status;

  int ZERO = 0, ONE = 1;

  // data storage
  double *time_sst = NULL,                    // sst-file
    *x1 = NULL, *x2 = NULL, *x3 = NULL,
    *x1p = NULL, *x2p = NULL, *x3p = NULL,
    *x1pp = NULL, *x2pp = NULL, *x3pp = NULL,
    *r = NULL, *phi = NULL, *lambda = NULL;
  double *time_sgg = NULL,                    // sgg-file
    *Txx = NULL, *Txy = NULL, *Txz = NULL,
    *Tyy = NULL, *Tyz = NULL, *Tzz = NULL,
    *I1 = NULL, *I2 = NULL, *I3 = NULL;

  int numread;

  double GMRRR;

  // Legendre functions
  int numlp;
  double *nL = NULL, *Bl = NULL;
  double *Wmm = NULL, *Wml_1 = NULL, *Wml_2 = NULL; // pre-evaluated values

  // matrix stuff
  int koeffs, N_siz, A_blk_siz, blk_cnt;
  double *A_row = NULL, *N_mat = NULL, *A_line = NULL, *x_vec = NULL,    *x_out = NULL;
  double alpha = 1.0, beta = 1.0;

  int locA_m, locA_n, locN_m, locN_n;  // local sizes
  int descA[9], descN[9],              // array descriptors
    descx_vec[9], descTzz[9],
    descAl[9], desc_out[9];

  // other stuff
  int i, b, bb, j, l, m, idx;          // counter
  double t;
  double *buf = NULL;                  // buffer for sending data
  int ii, jj, il, jl, procid, mglob, nglob, // calculation of block coordinates
    mloc_o, nloc_o, mloc_n, nloc_n,
    mrow, mcol, mrow_o, mcol_o;

  // performance monitoring
  Ttimer timer_data, timer_ALL, timer_A, timer_N, timer_dist, timer_solv;



  printf("Starting␣MPI...\n");
  // Start MPI
  MPI_Init(&argc, &argv);

  // get number of processes and rank/name of this process
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  if (procs < 2) {
    printf("ATTENTION!␣Please␣use␣at␣least␣two␣processes␣to␣run␣this␣program!\n");
    MPI_Finalize();
    return -1;
  }

  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  // get BLACS context
  Cblacs_get(0, 0, &ctxt);

  // Initialize "optimal" 2D processing grid (so that prow <= pcol)
```

```
pcol = sqrt(procs);
while (procs % pcol != 0) {
  pcol--;
}
prow = procs / pcol;
if (prow > pcol) {
  pcol = prow; prow = procs / pcol;
}

// Initialize the prow x pcol process grid
Cblacs_gridinit(&ctxt, "Row-major", prow, pcol);
Cblacs_pcoord(ctxt, myid, &myrow, &mycol);

if (myid == 0) { // only the main process does this
  printf("␣␣-->␣%d␣processes␣used␣in␣a␣%d␣x␣%d␣grid\n", procs, prow, pcol);
}


///// INTERPRET COMMANDLINE PARAMETERS /////
if (myid == 0) { // only the main process does this
  cmd_parameters = interprete_commandline(argc, argv);
  printf("␣␣-->␣command␣line␣parameters\n");
  printf("␣␣␣␣␣␣␣SST␣file␣................␣%s\n", cmd_parameters.in_sst_filename);
  printf("␣␣␣␣␣␣␣SGG␣file␣................␣%s\n", cmd_parameters.in_sgg_filename);
  printf("␣␣␣␣␣␣␣max␣lm␣..................␣%d\n", cmd_parameters.max_lm);
  printf("␣␣␣␣␣␣␣lines␣...................␣%d\n", cmd_parameters.N);
  printf("␣␣␣␣␣␣␣out␣file␣................␣%s\n", cmd_parameters.out_filename);
  printf("␣␣␣␣␣␣␣local␣matrix␣block␣size␣.␣%d␣x␣%d\n", cmd_parameters.block_m,
      cmd_parameters.block_n);
}
// send commandline parameters to the other processes
MPI_Bcast(&cmd_parameters, sizeof(cmd_parameters), MPI_BYTE, 0, MPI_COMM_WORLD);

// Here we start performance monitoring (only process 0!)
if (myid == 0) {
  reset_timer(&timer_ALL); start_timer(&timer_ALL);
  reset_timer(&timer_data);
  reset_timer(&timer_A);
  reset_timer(&timer_N);
  reset_timer(&timer_dist);
  reset_timer(&timer_solv);
}


///// READ DATA /////
if (myid == 0) { // only the main process does this
  start_timer(&timer_data);
  printf("␣␣-->␣input␣data\n");
  printf("␣␣␣␣␣␣␣SST␣data:␣%s\n", cmd_parameters.in_sst_filename);
  printf("␣␣␣␣␣␣␣SGG␣data:␣%s\n", cmd_parameters.in_sgg_filename);
  if ((numread = input_sst_sgg_bin(cmd_parameters.N, cmd_parameters.block_m * procs,
                                   cmd_parameters.in_sst_filename, &time_sst, &x1, &x2,
                                       &x3, &x1p, &x2p, &x3p, &x1pp, &x2pp, &x3pp, &
                                       lambda, &phi, &r,
                                   cmd_parameters.in_sgg_filename, &time_sgg, &Txx, &Txy
                                       , &Txz, &Tyy, &Tyz, &Tzz, &I1, &I2, &I3)) <= 0) {
    return -1; // something went wrong
  }
  printf("␣␣␣␣␣␣␣#␣requested:␣%d␣--␣#␣read:␣%d\n", cmd_parameters.N, numread);
  cmd_parameters.N = numread; // update the count of datalines with the correct value
  printf("␣␣-->␣maximal␣usable␣data␣entries␣n␣=␣%d\n", cmd_parameters.N);
}

if (myid == 0) {
  printf("␣␣-->␣distributing␣data␣...\n");
}
```

```
BCast_Data(cmd_parameters.N, cmd_parameters.block_m,
           &time_sst,
           &x1, &x2, &x3, &x1p, &x2p, &x3p, &x1pp, &x2pp, &x3pp, &lambda, &phi, &r,
           &time_sgg,
           &Txx, &Txy, &Txz, &Tyy, &Tyz, &Tzz, &I1, &I2, &I3);

printf("␣␣␣␣␣␣process␣[%d,␣%d]␣finished.\n", myrow, mycol);
MPI_Barrier(MPI_COMM_WORLD);
if (myid == 0) {
  stop_timer(&timer_data);
}


///// INITIALIZING COMPUTATIONS /////
koeffs      = (cmd_parameters.max_lm + 1) * (cmd_parameters.max_lm + 1) - 3; // c_1m,
    s_1m are not estimated
N_siz       = koeffs * koeffs;                                        // size of
    the normal equations matrix
A_blk_siz   = cmd_parameters.block_m * procs;                         // block
    size for building normal equations matrix

if (myid == 0) {
  printf("␣␣-->␣size␣A_row␣....................␣%d␣x␣%d␣␣(=␣%0.2f␣MiB)\n",
         A_blk_siz, koeffs, (double)A_blk_siz * (double)koeffs * sizeof(double) / 1024 /
              1024);
}
MPI_Barrier(MPI_COMM_WORLD);
locA_m = numroc_(&A_blk_siz, &cmd_parameters.block_m, &myrow, &ZERO, &prow);
locA_n = numroc_(&koeffs,    &cmd_parameters.block_n, &mycol, &ZERO, &pcol);
printf("␣␣␣␣␣␣local␣size␣(process␣[%d,␣%d])␣...␣%d␣x␣%d␣␣(=␣%0.2f␣MiB)\n",
       myrow, mycol, locA_m, locA_n, (double)locA_m * (double)locA_n * sizeof(double) /
           1024 / 1024);
MPI_Barrier(MPI_COMM_WORLD);

if (myid == 0) {
  printf("␣␣-->␣size␣N␣........................␣%d␣x␣%d␣␣(=␣%0.2f␣MiB)\n",
         koeffs, koeffs, (double)N_siz * sizeof(double) / 1024 / 1024);
}
MPI_Barrier(MPI_COMM_WORLD);
locN_m = numroc_(&koeffs, &cmd_parameters.block_m, &myrow, &ZERO, &prow);
locN_n = numroc_(&koeffs, &cmd_parameters.block_n, &mycol, &ZERO, &pcol);
printf("␣␣␣␣␣␣local␣size␣(process␣[%d,␣%d])␣...␣%d␣x␣%d␣␣(=␣%0.2f␣MiB)\n",
       myrow, mycol, locN_m, locN_n, (double)locN_m * (double)locN_n * sizeof(double) /
           1024 / 1024);
MPI_Barrier(MPI_COMM_WORLD);

// allocate local matrices
A_row = (double*)calloc(locA_m * locA_n, sizeof(double));
N_mat = (double*)calloc(locN_m * locN_n, sizeof(double));
x_vec = (double*)calloc(koeffs, sizeof(double));
x_out = (double*)calloc(koeffs, sizeof(double));

// define array descriptors
descinit_(descA, &A_blk_siz, &koeffs, &cmd_parameters.block_m, &cmd_parameters.block_n,
    &ZERO, &ZERO, &ctxt, &locA_m, &info);
descinit_(descN, &koeffs, &koeffs, &cmd_parameters.block_m, &cmd_parameters.block_n, &
    ZERO, &ZERO, &ctxt, &locN_m, &info);

descinit_(descTzz, &cmd_parameters.N, &ONE, &cmd_parameters.N, &ONE, &ZERO, &ZERO, &ctxt
    , &cmd_parameters.N, &info);
descinit_(descx_vec, &koeffs, &ONE, &cmd_parameters.block_m, &ONE, &ZERO, &ZERO, &ctxt,
    &locN_m, &info);
descinit_(desc_out, &koeffs, &ONE, &koeffs, &ONE, &ZERO, &ZERO, &ctxt, &koeffs, &info);
    // to gather x_vec to one process after solving

if (myid == 0) {
  printf("\n␣␣-->␣blocks␣to␣compute:␣%d\n", (int)ceil((double)cmd_parameters.N / (double
      )A_blk_siz));
}

// initialize Legendre functions
numlp = (cmd_parameters.max_lm + 1) * cmd_parameters.max_lm / 2 + cmd_parameters.max_lm
    + 1 + 1;

nL = (double*)malloc(sizeof(double) * numlp);
Bl = (double*)malloc(sizeof(double) * cmd_parameters.max_lm);

Wmm   = (double*)malloc(sizeof(double) * cmd_parameters.max_lm); // initialize Legendre
    functions
Wml_1 = (double*)malloc(sizeof(double) * cmd_parameters.max_lm * (cmd_parameters.max_lm
    + 1) / 2);
Wml_2 = (double*)malloc(sizeof(double) * cmd_parameters.max_lm * (cmd_parameters.max_lm
    - 1) / 2);

init_legendre1kind(cmd_parameters.max_lm, Wmm, Wml_1, Wml_2);

MPI_Barrier(MPI_COMM_WORLD);

///// COMPUTATIONS /////
A_line = (double*)calloc(A_blk_siz * koeffs, sizeof(double)); // for computing A
buf    = (double*)calloc(cmd_parameters.block_m * cmd_parameters.block_n, sizeof(double)
    );
blk_cnt = 0;
GMRRR   = GM / (R * R * R);

for (i = 0; i < cmd_parameters.N; i += A_blk_siz) {

  // define A-matrix array descriptor (can change from block to block)
  locA_m = numroc_(&A_blk_siz, &cmd_parameters.block_m, &myrow, &ZERO, &prow);
  locA_n = numroc_(&koeffs, &cmd_parameters.block_n, &mycol, &ZERO, &pcol);
  descinit_(descA, &A_blk_siz, &koeffs, &cmd_parameters.block_m, &cmd_parameters.block_n
      , &ZERO, &ZERO, &ctxt, &locA_m, &info);

  // computation of A
  if ((myrow == 0) && (mycol == 0) && ((blk_cnt % 10) == 0)) {
    printf("␣␣␣␣␣␣block␣#%4d␣-␣size:␣%8d␣%8d\n", blk_cnt, A_blk_siz, i);
  }

  if (myid == 0) {
    start_timer(&timer_A);
  }

  for (b = 0; b < cmd_parameters.block_m; b++) {
    compute_A_line(&(A_line[b * koeffs]), cmd_parameters.max_lm, cmd_parameters.N,
        koeffs,
                   i + cmd_parameters.block_m * myid, b,
                   GMRRR, GM, R,
                   lambda, phi, r,
                   Bl, nL, Wmm, Wml_1, Wml_2);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  if (myid == 0) {
    stop_timer(&timer_A);
    start_timer(&timer_dist);
  }

  // distribute the design matrix A
  for (procid = 0; procid < procs; procid++) {

    for (mloc_o = 0; mloc_o < cmd_parameters.block_m; mloc_o += cmd_parameters.block_m)
        { // run over local A matrix elements
      for (nloc_o = 0; nloc_o < koeffs; nloc_o += cmd_parameters.block_n) {
                          // increase for one block
```

```
            loc2glob2D(mloc_o, nloc_o, procid, 0,
                       cmd_parameters.block_m, cmd_parameters.block_n, procs, 1,
                       &mglob, &nglob);

            glob2loc2D(mglob, nglob, cmd_parameters.block_m, cmd_parameters.block_n, prow,
                pcol, // compute local coordinates
                       &mloc_n, &nloc_n, &mrow, &mcol);


            ii = min(cmd_parameters.block_m, cmd_parameters.block_m - mloc_o);
            jj = min(cmd_parameters.block_n, koeffs - nloc_o);

            if (procid == myid) {
              if ((myrow == mrow) && (mycol == mcol)) { // block stays with this process

                for (il = 0; il < ii; il++) {
                  for (jl = 0; jl < jj; jl++) {
                    A_row[(nloc_n + jl) * locA_m + (mloc_n + il)] = A_line[(mloc_o + il) *
                        koeffs + (nloc_o + jl)];
                  }
                }

              } else { // block is sent to another process

                Cdgesd2d(ctxt, jj, ii, &(A_line[mloc_o * koeffs + nloc_o]), koeffs, mrow,
                    mcol);

              }
            } else if ((myrow == mrow) && (mycol == mcol)) { // block is received from
                another process

              Cblacs_pcoord(ctxt, procid, &mrow_o, &mcol_o); // get the grid coordiantes of
                  the sending process
              Cdgerv2d(ctxt, jj, ii, buf, 1, mrow_o, mcol_o);
              for (il = 0; il < ii; il++) {
                for (jl = 0; jl < jj; jl++) {
                  A_row[(nloc_n + jl) * locA_m + (mloc_n + il)] = buf[il * jj + jl];
                }
              }

            }
          }
        }
      }
      MPI_Barrier(MPI_COMM_WORLD);
      if (myid == 0) {
        stop_timer(&timer_dist);
        start_timer(&timer_N);
      }

      pdsyrk_("L", "T", &koeffs, &A_blk_siz, &alpha, A_row, &ONE, &ONE, descA, &beta, N_mat,
           &ONE, &ONE, descN); // N(i) = N(i-1) + A(i)' * A(i)
      pdgemv_("T", &A_blk_siz, &koeffs, &alpha, A_row, &ONE, &ONE, descA, Tzz+i, &ONE, &ONE,
           descTzz, &ONE, &beta, x_vec, &ONE, &ONE, descx_vec, &ONE);

      if (myid == 0) {
        stop_timer(&timer_N);
      }

      blk_cnt++;
    }
    MPI_Barrier(MPI_COMM_WORLD);

    // solve the system of equations
    if (myid == 0) {
      start_timer(&timer_solv);
      printf("DPOSV\n");
    }
    pdposv_("L", &koeffs, &ONE, N_mat, &ONE, &ONE, descN, x_vec, &ONE, &ONE, descx_vec, &
        info);

    // retrieve the coefficients' vector from the other processes
    pdgemr2d_(&koeffs, &ONE, x_vec, &ONE, &ONE, descx_vec, x_out, &ONE, &ONE, desc_out, &
        ctxt, &info);

    if (myid == 0) {
      stop_timer(&timer_solv);

      // OUTPUT
      printf("␣␣-->␣some␣coefficients␣(output␣is␣written␣to␣'%s')\n", cmd_parameters.
          out_filename);
      fil = fopen(cmd_parameters.out_filename, "w");

      i = 0;
      for (l = 0; l <= cmd_parameters.max_lm; l++) {
        for (m = 0; m <= l; m++) {

          if (l == 0) {
            clm = x_out[0]; slm = 0.0;
          } else if (l == 1) {
            clm = 0.0; slm = 0.0;
          } else if (m == 0) {
            clm = x_out[l - 1]; slm = 0.0;
          } else {
            idx = 2 * (m * (cmd_parameters.max_lm + 1) - m * (m - 1) / 2 + l - m) -
                cmd_parameters.max_lm - 4;
            clm = x_out[idx]; slm = x_out[idx + 1];
          }

          fprintf(fil, "%3d␣%3d␣␣%24.16e␣%24.16e\n", l, m, clm, slm);
          if (i <= 10) {
            printf("%3d␣%3d␣␣%24.16e␣%24.16e\n", l, m, clm, slm);
          }
          i++;

        }
      }
      fclose(fil);

      // performance results
      stop_timer(&timer_ALL);
      printf("␣␣-->␣performance␣results:\n");
      printf("␣␣␣␣␣␣reading␣data␣..................␣:␣%8.4f␣s\n", totaltime_timer(&
          timer_data));
      printf("␣␣␣␣␣␣computing␣of␣A␣................␣:␣%8.4f␣s\n", totaltime_timer(&timer_A))
          ;
      printf("␣␣␣␣␣␣distributing␣A␣................␣:␣%8.4f␣s\n", totaltime_timer(&
          timer_dist));
      printf("␣␣␣␣␣␣computing␣of␣N␣................␣:␣%8.4f␣s\n", totaltime_timer(&timer_N))
          ;
      printf("␣␣␣␣␣␣solving␣equations␣system␣......␣:␣%8.4f␣s\n", totaltime_timer(&
          timer_solv));
      printf("␣␣␣␣␣␣OVERALL␣TIME␣..................␣:␣%8.4f␣s\n", totaltime_timer(&timer_ALL
          ));
    }

   ENDE:

    // wait for all processes to reach this point
    MPI_Barrier(MPI_COMM_WORLD);

    // release process grid
    Cblacs_gridexit(ctxt);
    // Shut down MPI
    MPI_Finalize();

    return 0;
  }


  // helper: return the minimum of two values
  int min(int x, int y) {
    if (x < y)
      return x;
    else
      return y;
  }
```

A.5 Commonly used files

commandline.h

```
struct t_cmd_line {
  char in_sst_filename[200];
  char in_sgg_filename[200];
#ifndef ASCII2BIN
  char out_filename[200];
  int max_lm;
  int N;
  int block_m;
  int block_n;
#endif
};

struct t_cmd_line interprete_commandline(int argc, char* argv[]);
```

commandline.c

```
#if defined(SX8)         // SX-8/9
#elif defined(ASAMA)       // ASAMA und A1
  #include <getopt.h>
#else                    // LOCAL
  #include <getopt.h>
#endif

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "commandline.h"

struct t_cmd_line interprete_commandline(int argc, char* argv[]) {

  // default values
  struct t_cmd_line cmdpar = {"../data/sc7/sc7_sst_coord_localsph",  // SST file
                              "../data/sc7/sc7_tensor_inv_localsph", // SGG file
#ifndef ASCII2BIN
                              "./test.koeff",                        // output file
                              20,       // maximum degree and order
                              64000,    // number of lines of data to read (0: all)
                              64, 64    // size of matrix block (m x n), ATTENTION: pdposv
                                   needs quadratic blocks!
#endif
  };
```

```
#ifndef SX8
  // commandline parameters
  static struct option long_options[] = {
    {"sstfile", required_argument, 0, 'a'},
    {"sggfile", required_argument, 0, 'b'},
#ifndef ASCII2BIN
    {"maxlm",   required_argument, 0, 'c'},
    {"lines",   required_argument, 0, 'd'},
    {"outfile", required_argument, 0, 'o'},
    {"block_m", required_argument, 0, 'm'},
    {"block_n", required_argument, 0, 'n'},
#endif
    {0, 0, 0, 0}
  };
  int option_index = 0;
#endif
  int command;

  // interprete commandline parameters
#ifdef SX8
#ifdef ASCII2BIN
  while ((command = getopt(argc, argv, "a:b:")) != -1) {
#else
  while ((command = getopt(argc, argv, "a:b:c:d:o:m:n:")) != -1) {
#endif
    switch (command) {
    case 'a':
      strncpy(cmdpar.in_sst_filename, optarg, 200);
      break;
    case 'b':
      strncpy(cmdpar.in_sgg_filename, optarg, 200);
      break;
#ifndef ASCII2BIN
    case 'c':
      sscanf(optarg, "%d", &cmdpar.max_lm);
      break;
    case 'd':
      sscanf(optarg, "%d", &cmdpar.N);
      break;
    case 'o':
      strncpy(cmdpar.out_filename, optarg, 200);
      break;
    case 'm':
      sscanf(optarg, "%d", &cmdpar.block_m);
      break;
    case 'n':
      sscanf(optarg, "%d", &cmdpar.block_n);
      break;
#endif
    default:
      break;
    }
  }
#else // on ASAMA and locally "getopt.h" is available (long names for commandline options)
#ifdef ASCII2BIN
  while ((command = getopt_long(argc, argv, "a:b:", long_options, &option_index)) != -1) {
#else
  while ((command = getopt_long(argc, argv, "a:b:c:d:o:m:n:", long_options, &option_index)
      ) != -1) {
#endif
    switch (command) {
    case 0 :
      break;
    case 'a' :
      strncpy(cmdpar.in_sst_filename, optarg, 200);
      break;
    case 'b' :
       strncpy(cmdpar.in_sgg_filename, optarg, 200);
      break;
#ifndef ASCII2BIN
    case 'c' :
      sscanf(optarg, "%d", &cmdpar.max_lm);
      break;
    case 'd' :
      sscanf(optarg, "%d", &cmdpar.N);
      break;
    case 'o':
      strncpy(cmdpar.out_filename, optarg, 200);
      break;
    case 'm' :
      sscanf(optarg, "%d", &cmdpar.block_m);
      break;
    case 'n' :
      sscanf(optarg, "%d", &cmdpar.block_n);
      break;
#endif
   default:
      break;
    }
  }
#endif

  return cmdpar;
}
```

compute.h

```
// computes one line of matrix A
void compute_A_line(double *A_line, int max_lm, int N, int koeffs, int i, int b,
                    double GMRRR, double GM, double R,
                    double *lambda, double *phi, double *r,
                    double *Bl, double *nL, double *Wmm, double *Wml_1, double *Wml_2);
```

compute.c

```
#include "legendre.h"
#include <math.h>

void compute_A_line(double *A_line, int max_lm, int N, int koeffs, int i, int b,
                    double GMRRR, double GM, double R,
                    double *lambda, double *phi, double *r,
                    double *Bl, double *nL, double *Wmm, double *Wml_1, double *Wml_2) {

  int ib, m, l, idx_0, idx_nL, idx;
  double lambda_ib, phi_ib, r_ib, R_rib, m_lambda_ib,
    BlP, cosml, sinml;

  ib = i + b;
  if (ib >= N) {
    for (idx_0 = 0; idx_0 < koeffs; idx_0++) {
      A_line[idx_0] = 0.0;
    }
  } else {
    lambda_ib = lambda[ib];
    phi_ib    = phi[ib];
    r_ib      = r[ib];
    R_rib     = R / r_ib;

    legendre1kind(max_lm, phi_ib, nL, Wmm, Wml_1, Wml_2);
    // output of Legendre functions is in order:
    // nL: 00,10,20,30,...,11,21,31,...,22,32,...
```

```
      idx_0 = 0;
      A_line[idx_0] = 2.0 * GM / pow(r_ib, 3);              // c_(0,0)

      BlP = pow(R_rib, 5);
      for (l = 2; l <= max_lm; l++) {                       // c_(2,0) ... c_(max_lm,0)
        Bl[l - 2] = GMRRR * BlP * (l+2) * (l+1);          // GMRRR * pow(R / r_ib, l+3) * (l+2)
             * (l+1); save for later
        BlP      *= R_rib;                                // pow(R / r_ib, l+3);
        A_line[idx_0 + l - 1] = Bl[l - 2] * nL[l];
      }

      idx_0 -= (max_lm + 4);                // further down: idx = b * koeffs + 2 * idx_nL -
           max_lm - 4;
      for (m = 1; m <= max_lm; m++) {       // c_(2,1), s_(2,1) ... c_(max_lm,max_lm), s_(
          max_lm,max_lm)
        m_lambda_ib = m * lambda_ib;
        cosml = cos(m_lambda_ib);
        sinml = sin(m_lambda_ib);

        for (l = m; l <= max_lm; l++) {
          if (l == 1)
            continue;

          idx_nL = (m * (max_lm + 1) - m * (m - 1) / 2 + l - m);
          idx    = idx_0 + 2 * idx_nL;

          BlP = Bl[l - 2] * nL[idx_nL];

          A_line[idx]     = BlP * cosml;
          A_line[idx + 1] = BlP * sinml;
        }
      }
    }
  }
```

dataio.h

```
// data input routines

// ASCII data (non-synchronized)
int input_sst_sgg(int n, int memchunksize,
                  char *sst_in_filename,
                  double **sst_time,
                  double **x1,     double **x2,   double **x3,
                  double **x1p,    double **x2p,  double **x3p,
                  double **x1pp,   double **x2pp, double **x3pp,
                  double **lambda, double **phi,  double **r,
                  char *sgg_in_filename,
                  double **sgg_time,
                  double **Txx, double **Txy, double **Txz,
                  double **Tyy, double **Tyz, double **Tzz,
                  double **I1,  double **I2,  double **I3);


// binary data (already synchronized!) in big-endian format
double swapEndian_double(double in);

int input_sst_sgg_bin(int n, int memchunksize,
                      char *sst_in_filename,
                      double **sst_time,
                      double **x1,     double **x2,   double **x3,
                      double **x1p,    double **x2p,  double **x3p,
                      double **x1pp,   double **x2pp, double **x3pp,
                      double **lambda, double **phi,  double **r,
                      char *sgg_in_filename,
                      double **sgg_time,
                      double **Txx, double **Txy, double **Txz,
                      double **Tyy, double **Tyz, double **Tzz,
                      double **I1,  double **I2,  double **I3);


// distribute the data to the processes (MPI version only)
#ifdef MPI
void BCast_Data(int n, int memchunksize,
                double **time_sst,
                double **x1, double **x2, double **x3,
                double **x1p, double **x2p, double **x3p,
                double **x1pp, double **x2pp, double **x3pp,
                double **lambda, double **phi, double **r,
                double **time_sgg,
                double **Txx, double **Txy, double **Txz,
                double **Tyy, double **Tyz, double **Tzz,
                double **I1, double **I2, double **I3);
#endif
```

dataio.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#ifdef MPI
  #include "mpi.h"
#endif

/************************************************************
   function for combined sst and sgg data input (NON-SYNCHRONIZED ASCII DATA)

   input:    data file, lenght of sequences, size of memory-step, sst data arrays
   output:   sst data
   return:   number of read data lines (> 0)
             if error: < 0
               -1 : can't open file
               -2 : can't reallocate memory
               -3 : timestamps could not be synchronized
************************************************************/
int input_sst_sgg(int n, int memchunksize,
                  char *sst_in_filename,
                  double **sst_time,
                  double **x1,     double **x2,   double **x3,
                  double **x1p,    double **x2p,  double **x3p,
                  double **x1pp,   double **x2pp, double **x3pp,
                  double **lambda, double **phi,  double **r,
                  char *sgg_in_filename,
                  double **sgg_time,
                  double **Txx, double **Txy, double **Txz,
                  double **Tyy, double **Tyz, double **Tzz,
                  double **I1,  double **I2,  double **I3) {
  FILE *sst_in_fil, *sgg_in_fil;
  int N      = 0,
    corr_sst = 0,
    corr_sgg = 0,
    M        = memchunksize,
    i        = 0;

  // do files exist?
  if ((sst_in_fil = fopen(sst_in_filename, "r")) == NULL) {
    return -1; // Can't open file "sst_in_filename"
  }
  if ((sgg_in_fil = fopen(sgg_in_filename, "r")) == NULL) {
    return -1; // Can't open file "sgg_in_filename"
  }

  // allocate memory - sst data
```

```
*sst_time = (double *)malloc(sizeof(double) * M);
*x1       = (double *)malloc(sizeof(double) * M);
*x2       = (double *)malloc(sizeof(double) * M);
*x3       = (double *)malloc(sizeof(double) * M);
*x1p      = (double *)malloc(sizeof(double) * M);
*x2p      = (double *)malloc(sizeof(double) * M);
*x3p      = (double *)malloc(sizeof(double) * M);
*x1pp     = (double *)malloc(sizeof(double) * M);
*x2pp     = (double *)malloc(sizeof(double) * M);
*x3pp     = (double *)malloc(sizeof(double) * M);
*lambda   = (double *)malloc(sizeof(double) * M);
*phi      = (double *)malloc(sizeof(double) * M);
*r        = (double *)malloc(sizeof(double) * M);
// allocate memory - sgg data
*sgg_time = (double *)malloc(sizeof(double) * M);
*Txx      = (double *)malloc(sizeof(double) * M);
*Txy      = (double *)malloc(sizeof(double) * M);
*Txz      = (double *)malloc(sizeof(double) * M);
*Tyy      = (double *)malloc(sizeof(double) * M);
*Tyz      = (double *)malloc(sizeof(double) * M);
*Tzz      = (double *)malloc(sizeof(double) * M);
*I1       = (double *)malloc(sizeof(double) * M);
*I2       = (double *)malloc(sizeof(double) * M);
*I3       = (double *)malloc(sizeof(double) * M);

while (1) {

  if (N + 1 >= M) { // enlarge memory area if necessary
    M += memchunksize;
    // sst data
    if ((*sst_time = (double *)realloc(*sst_time, sizeof(double) * M)) == NULL) { return
         -2; } // Can't reallocate
    if ((*x1       = (double *)realloc(*x1,       sizeof(double) * M)) == NULL) { return
         -2; } // ...
    if ((*x2       = (double *)realloc(*x2,       sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*x3       = (double *)realloc(*x3,       sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*x1p      = (double *)realloc(*x1p,      sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*x2p      = (double *)realloc(*x2p,      sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*x3p      = (double *)realloc(*x3p,      sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*x1pp     = (double *)realloc(*x1pp,     sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*x2pp     = (double *)realloc(*x2pp,     sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*x3pp     = (double *)realloc(*x3pp,     sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*lambda   = (double *)realloc(*lambda,   sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*phi      = (double *)realloc(*phi,      sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*r        = (double *)realloc(*r,        sizeof(double) * M)) == NULL) { return
         -2; }
    // sgg data
    if ((*sgg_time = (double *)realloc(*sgg_time, sizeof(double) * M)) == NULL) { return
         -2; } // Can't reallocate
    if ((*Txx      = (double *)realloc(*Txx,      sizeof(double) * M)) == NULL) { return
         -2; } // ...
    if ((*Txy      = (double *)realloc(*Txy,      sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*Txz      = (double *)realloc(*Txz,      sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*Tyy      = (double *)realloc(*Tyy,      sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*Tyz       = (double *)realloc(*Tyz,       sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*Tzz       = (double *)realloc(*Tzz,       sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*I1        = (double *)realloc(*I1,        sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*I2        = (double *)realloc(*I2,        sizeof(double) * M)) == NULL) { return
         -2; }
    if ((*I3        = (double *)realloc(*I3,        sizeof(double) * M)) == NULL) { return
         -2; }
  }


  if ((corr_sst == 0) && (corr_sgg == 0)) { // if both "corr" = 0:

    if (fscanf(sst_in_fil, "␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf",
               &(*sst_time)[N],
               &(*x1)[N],     &(*x2)[N],    &(*x3)[N],
               &(*x1p)[N],    &(*x2p)[N],   &(*x3p)[N],
               &(*x1pp)[N],   &(*x2pp)[N],  &(*x3pp)[N],
               &(*lambda)[N], &(*phi)[N],   &(*r)[N]) <= 0) { // ERROR while reading sst-
                   file
      printf("␣␣␣␣␣␣␣eof␣sst!\n");
      break;
    }
    if (fscanf(sgg_in_fil, "␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf",
               &(*sgg_time)[N],
               &(*Txx)[N], &(*Txy)[N], &(*Txz)[N],
               &(*Tyy)[N], &(*Tyz)[N], &(*Tzz)[N],
               &(*I1)[N],  &(*I2)[N],  &(*I3)[N]) <= 0) { // ERROR while reading sgg-
                   file
      printf("␣␣␣␣␣␣␣eof␣sgg!\n");
      break;
    }
  } else if (corr_sst != 0) { // if corr_sst != 0, only read sst and set corr_sst = 0;
      don't inc N
    if (fscanf(sst_in_fil, "␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf",
               &(*sst_time)[N - 1],
               &(*x1)[N - 1],     &(*x2)[N - 1],    &(*x3)[N - 1],
               &(*x1p)[N - 1],    &(*x2p)[N - 1],   &(*x3p)[N - 1],
               &(*x1pp)[N - 1],   &(*x2pp)[N - 1],  &(*x3pp)[N - 1],
               &(*lambda)[N - 1], &(*phi)[N - 1],   &(*r)[N - 1]) <= 0) {
      printf("␣␣␣␣␣␣␣eof␣sst!\n");
      break;
    }
    corr_sst = 0; N--;
  } else if (corr_sgg != 0) { // if corr_sgg > 0, only read sgg and set corr_sgg = 0;
      don't inc N
    if (fscanf(sgg_in_fil, "␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf␣%lf",
               &(*sgg_time)[N - 1],
               &(*Txx)[N - 1], &(*Txy)[N - 1], &(*Txz)[N - 1],
               &(*Tyy)[N - 1], &(*Tyz)[N - 1], &(*Tzz)[N - 1],
               &(*I1)[N - 1],  &(*I2)[N - 1],  &(*I3)[N - 1]) <= 0) {
      printf("␣␣␣␣␣␣␣eof␣sgg!\n");
      break;
    }
    corr_sgg = 0; N--;
  }

  if ((*sst_time)[N] != (*sgg_time)[N]) {
    printf("␣␣␣␣␣␣␣no␣synch:␣N␣=␣%2d␣␣-->␣sst_time␣=␣%6.6f␣sgg_time␣=␣%6.6f\n",
           N, (*sst_time)[N], (*sgg_time)[N]);
    if ((*sst_time)[N] > (*sgg_time)[N]) {
      if (corr_sst != 0)
        return -3; // dates can't be synchronized
      corr_sgg = 1;
    } else if ((*sst_time)[N] > (*sgg_time)[N]) {
        if (corr_sgg != 0)
          return -3; // dates can't be synchronized
        corr_sst = 1;
        }
    }

    N++;

#ifdef SX8
    if (N % 1000 == 0) {
      printf("␣␣%d␣Datenzeilen␣gelesen\n", N);
    }
#endif

    if ((N >= n) && (n != 0)) // break if n lines of data are read
      break;
  } // while
  fclose(sst_in_fil);
  fclose(sgg_in_fil);

  for (i = 0; i < N; i++) { // adapt the units
    (*Txx)[i] = (*Txx)[i] * 1e-9;
    (*Txy)[i] = (*Txy)[i] * 1e-9;
    (*Txz)[i] = (*Txz)[i] * 1e-9;
    (*Tyy)[i] = (*Tyy)[i] * 1e-9;
    (*Tyz)[i] = (*Tyz)[i] * 1e-9;
    (*Tzz)[i] = (*Tzz)[i] * 1e-9;
    (*I1)[i]  = (*I1)[i]  * 1e-9;
    (*I2)[i]  = (*I2)[i]  * 1e-9;
    (*I3)[i]  = (*I3)[i]  * 1e-9;
  }

  return N;
}


/***********************************************************
  Binary data is stored in big endian (as SX-8/SX-9 needs this format).
  For x86-architecture it has to be converted to little endian.
  This function does the conversion little --> big AND big --> little.
***********************************************************/
double swapEndian_double(double in) {
  int siz = sizeof(double);
  double tmp;
  char *tmp_ptr, *in_ptr;
  int i;
  in_ptr  = (char*)&in;
  tmp_ptr = (char*)&tmp;
  for (i = 0; i < siz; i++) {
    tmp_ptr[i] = in_ptr[siz - 1 - i];
  }
  return tmp;
}


/***********************************************************
   function for combined sst and sgg data input -- already SYNCHRONIZED BINARY DATA

   (data files must have the ending ".bin_*", don't specify the ending in *_in_filename!)
   Also no need for normalizing Txx, Txy, ... as it is done already
   The user has to take care that all data files contains the same amount of data!

   input:    data file, lenght of sequences, size of memory-step, sst data arrays
   output:   sst, sgg data
   return:   number of read data lines (> 0)
             if error: < 0
               -1 : can't open file
```

```
              -2 : can't reallocate memory
              -3 : timestamps could not be synchronized
***********************************************************/
int input_sst_sgg_bin(int n, int memchunksize,
                  char *sst_in_filename,
                  double **sst_time,
                  double **x1,     double **x2,   double **x3,
                  double **x1p,    double **x2p,  double **x3p,
                  double **x1pp,   double **x2pp, double **x3pp,
                  double **lambda, double **phi,  double **r,
                  char *sgg_in_filename,
                  double **sgg_time,
                  double **Txx, double **Txy, double **Txz,
                  double **Tyy, double **Tyz, double **Tzz,
                  double **I1,  double **I2,  double **I3) {
  FILE *fil;
  int N, m, M = 0;

  double *tmp = NULL;
  int i, j;
  char filname[200];
  char out_sst_fil[][10] = {"sst_time", "x1", "x2", "x3", "x1p", "x2p", "x3p", "x1pp", "
      x2pp", "x3pp", "lambda", "phi", "r"};
  char out_sgg_fil[][10] = {"sgg_time", "Txx", "Txy", "Txz", "Tyy", "Tyz", "Tzz", "I1", "
      I2", "I3"};
  char *buf = NULL;

  N = n; m = n;
  while (m % memchunksize != 0) {
    m++;
  }

  printf("␣␣-->␣changing␣memory␣allocation␣from␣%d␣to␣%d␣entries␣(chunk␣size:␣%d)\n", n, m
      , memchunksize);

  // SST-DATEN
  for (j = 0; j < 13; j++) {
    sprintf(filname, "%s.bin_%s", sst_in_filename, out_sst_fil[j]);
    printf("␣␣␣␣␣␣␣-␣%s\n", filname);

    if ((fil = fopen(filname, "rb")) == NULL) { // file exists?
      return -1; // Can't open file "filname"
    }

    if ((tmp = (double *)calloc(m, sizeof(double))) == NULL) { return -2; }; // allocate
        memory, if not enough: ERROR
    setvbuf(fil, buf, _IOFBF, 4 * 1024 * 1024); // full buffering of input files

    if ((M = fread(&(*tmp), sizeof(double), N, fil)) <= 0) { // ERROR while reading file
      printf("␣␣␣␣␣␣␣eof!\n");
      return -1;//      break;
    }

    for (i = N; i < m; i++) { // fill the allocated memory with zeroes
      tmp[i] = 0.0;
    }

#ifndef SX8
    for (i = 0; i < N; i++) {
      tmp[i] = swapEndian_double(tmp[i]);
    }
#endif
    switch (j) {
    case  0: *sst_time = tmp; break;
    case  1: *x1       = tmp; break;
    case  2: *x2       = tmp; break;
    case  3: *x3       = tmp; break;
    case  4: *x1p      = tmp; break;
    case  5: *x2p      = tmp; break;
    case  6: *x3p      = tmp; break;
    case  7: *x1pp     = tmp; break;
    case  8: *x2pp     = tmp; break;
    case 19: *x3pp     = tmp; break;
    case 10: *lambda   = tmp; break;
    case 11: *phi      = tmp; break;
    case 12: *r        = tmp; break;
    }

    fclose(fil);
  }

  // SGG-DATEN
  for (j = 0; j < 10; j++) {
    sprintf(filname, "%s.bin_%s", sgg_in_filename, out_sgg_fil[j]);
    printf("␣␣␣␣␣␣␣-␣%s\n", filname);

    if ((fil = fopen(filname, "rb")) == NULL) { // file exists?
      return -1; // Can't open file "filname"
    }

    tmp = (double *)calloc(n, sizeof(double)); // allocate memoty and fill with zeroes

    setvbuf(fil, buf, _IOFBF, 4 * 1024 * 1024); // full buffering of input files

    if ((M = fread(&(*tmp), sizeof(double), N, fil)) <= 0) { // ERROR while reading file
      printf("␣␣␣␣␣␣␣eof!\n");
      return -1;//      break;
    }

#ifndef SX8
    for (i = 0; i < N; i++) {
      tmp[i] = swapEndian_double(tmp[i]);
    }
#endif
    switch (j) {
    case  0: *sgg_time = tmp; break;
    case  1: *Txx      = tmp; break;
    case  2: *Txy      = tmp; break;
    case  3: *Txz      = tmp; break;
    case  4: *Tyy      = tmp; break;
    case  5: *Tyz      = tmp; break;
    case  6: *Tzz      = tmp; break;
    case  7: *I1       = tmp; break;
    case  8: *I2       = tmp; break;
    case  9: *I3       = tmp; break;
    }

    fclose(fil);
  }

  return N;
}


/*************************************************************
  Distribute the read data to the other processes
 ************************************************************/
#ifdef MPI
void BCast_Data(int n, int memchunksize,
                double **time_sst,
                double **x1, double **x2, double **x3,
                double **x1p, double **x2p, double **x3p,
                double **x1pp, double **x2pp, double **x3pp,
                double **lambda, double **phi, double **r,
                double **time_sgg,
                double **Txx, double **Txy, double **Txz,
                double **Tyy, double **Tyz, double **Tzz,
                double **I1, double **I2, double **I3) {

  int m;
  m = n;
  while (m % memchunksize != 0) {
    m++;
  }

  // send zero-filled datalength to the other processes
  MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);

  // allocate memory - sst data
  if (*time_sst == NULL) { *time_sst = (double *)malloc(sizeof(double) * m); }
  if (*x1       == NULL) { *x1       = (double *)malloc(sizeof(double) * m); }
  if (*x2       == NULL) { *x2       = (double *)malloc(sizeof(double) * m); }
  if (*x3       == NULL) { *x3       = (double *)malloc(sizeof(double) * m); }
  if (*x1p      == NULL) { *x1p      = (double *)malloc(sizeof(double) * m); }
  if (*x2p      == NULL) { *x2p      = (double *)malloc(sizeof(double) * m); }
  if (*x3p      == NULL) { *x3p      = (double *)malloc(sizeof(double) * m); }
  if (*x1pp     == NULL) { *x1pp     = (double *)malloc(sizeof(double) * m); }
  if (*x2pp     == NULL) { *x2pp     = (double *)malloc(sizeof(double) * m); }
  if (*x3pp     == NULL) { *x3pp     = (double *)malloc(sizeof(double) * m); }
  if (*lambda   == NULL) { *lambda   = (double *)malloc(sizeof(double) * m); }
  if (*phi      == NULL) { *phi      = (double *)malloc(sizeof(double) * m); }
  if (*r        == NULL) { *r        = (double *)malloc(sizeof(double) * m); }
  // allocate memory - sgg data
  if (*time_sgg == NULL) { *time_sgg = (double *)malloc(sizeof(double) * m); }
  if (*Txx      == NULL) { *Txx      = (double *)malloc(sizeof(double) * m); }
  if (*Txy      == NULL) { *Txy      = (double *)malloc(sizeof(double) * m); }
  if (*Txz      == NULL) { *Txz      = (double *)malloc(sizeof(double) * m); }
  if (*Tyy      == NULL) { *Tyy      = (double *)malloc(sizeof(double) * m); }
  if (*Tyz      == NULL) { *Tyz      = (double *)malloc(sizeof(double) * m); }
  if (*Tzz      == NULL) { *Tzz      = (double *)malloc(sizeof(double) * m); }
  if (*I1       == NULL) { *I1       = (double *)malloc(sizeof(double) * m); }
  if (*I2       == NULL) { *I2       = (double *)malloc(sizeof(double) * m); }
  if (*I3       == NULL) { *I3       = (double *)malloc(sizeof(double) * m); }

  // send the read data to the all nodes
  MPI_Bcast((*time_sst), m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*x1)      , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*x2)      , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*x3)      , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*x1p)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*x2p)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*x3p)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*x1pp)    , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*x2pp)    , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*x3pp)    , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*lambda)  , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*phi)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*r)       , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*time_sgg), m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*Txx)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*Txy)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*Txz)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*Tyy)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*Tyz)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*Tzz)     , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*I1)      , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*I2)      , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast((*I3)      , m, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  MPI_Barrier(MPI_COMM_WORLD); // Wait for all processes to finish
```

```
}
#endif
```

legendre.h

```
// Legendre functions

void init_legendre1kind(int kmax, double *Wmm, double *Wlm_1, double *Wlm_2);

void legendre1kind(int kmax, double lat, double *nL, double *Wmm, double *Wlm_1, double *
    Wlm_2);
```

legendre.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "legendre.h"

extern double pi;

/****************************************************************************
function for computing the normalized LEGENDRE functions of the first kind
as well as its first derivatives

input:    degree/order of computation, argument for calculation (latitude phi)

output:   normalized LEGENDRE functions                       nL
// not used at the moment:
//          normalized LEGENDRE functions - first derivatives   dnL
//          normalized LEGENDRE functions - second derivatives  ddnL
*************************************************************************** */

/* void legendre1kind(int kmax, double lat, double *nL, double *dnL, double *ddnL) {
  int l, m, k;
  double ff1, ff2, fk;
  double f1[2], f2[2];
  double eta, sigma, tau, ny;
  double sinlat, coslat, sinpilat, cospilat;

  sinlat   = sin(lat);
  coslat   = cos(lat);
  sinpilat = sin(pi/2 - lat);
  cospilat = cos(pi/2 - lat);

  // invariant coefficients
  f1[0] = sqrt(3.);
  f2[0] = sqrt(5.);
  f1[1] = sqrt(15.);
  f2[1] = f1[1] / 2.;

  // starting values
  nL[0] = 1.;
  nL[1] = f1[0] * sinlat;
  nL[2] = f1[0] * coslat;
  nL[3] = f2[0] * (1.5 * sinlat * sinlat - 0.5);
  nL[4] = f1[1] * sinlat * coslat;
  nL[5] = f2[1] * coslat * coslat;

  // normalized legendre function of the first kind; tested and right (second possibility)
  for (l = 3; l <= kmax; l++) {
    for(m = 0; m <= (l-2); m++) {
      ff1 = sqrt(4.*l * l - 1.) / sqrt(l*l - m*m);
      ff2 = sqrt((2.*l + 1.) * (l + m - 1.) * (l - m - 1.)) / sqrt((l*l - m*m)*(2.*l - 3.)
          );
```

```
        nL[(l+1) * l/2 + m] = ff1 * sinlat * nL[l * (l-1)/2 + m] - ff2 * nL[(l-1) * (l-2)/2
            + m];
      }

      for (k = 1; k <= 2; k++) {
        fk = sqrt(2*l + 1.) / sqrt(2. * (l-k+1.));

        nL[l * (l+1)/2 + l + 1 - k] = fk * coslat * nL[l * (l-1)/2 + l - k];
      }
    }

    // normalized legendre function of the first kind - 1st derivative; tested and right (
        second possibility)
    dnL[0] = 0;
    dnL[1] = sqrt(3) * sinpilat;
    dnL[2] = -sqrt(3) * cospilat;

    for (l = 2; l <= kmax; l++) {
      tau = sqrt(2*l + 1);
      ny  = sqrt((2*l + 1) / (2.0*l));

      for (m = 0; m <= l - 2; m++) {
        eta   = sqrt(((2*l + 1) * (2*l - 1)) / ((l + m) * (l - m) * 1.0));
        sigma = sqrt(((2*l + 1) * (l + m - 1) * (l - m - 1)) / ((2*l - 3) * (l+m) * (l-m) *
            1.0));


        dnL[(l+1)*l/2+m] = eta * (cospilat * dnL[l * (l-1)/2 + m] + sinpilat * nL[l * (l-1)
            /2 + m]) - sigma * dnL[(l-1) * (l-2)/2 + m];
      }
      dnL[(l+1)*l/2+l-1] = -tau * (-cospilat * dnL[l * (l-1)/2 + l - 1] - sinpilat * nL[l *
          (l-1)/2 + l - 1]);
      dnL[(l+1)*l/2+l]   = -ny * (-sinpilat * dnL[l * (l-1)/2 + l - 1] + cospilat * nL[l * (
          l-1)/2 + l - 1]);
    }

    // normalized legendre function of the first kind - 2nd derivative; tested and right (
        single examples)
    for (l = 0; l <= kmax; l++) {
      for (m = 0; m <= l; m++) {
        ddnL[(l+1)*l/2+m] = tan(lat) * dnL[(l+1) * l/2 + m] - (l * (l+1) - (m*m) / pow(cos(
            lat),2)) * nL[(l+1) * l/2 + m];
      }
    }
  }
  */

  void init_legendre1kind(int kmax, double *Wmm, double *Wlm_1, double *Wlm_2) {
    int l, m;
    int id_mm = 0, id_lm_1 = 0, id_lm_2 = 0;

    for (m = 0; m <= kmax; m++) {
      if (m != 0) {
        if (m == 1) {
          Wmm[id_mm++] = sqrt(3.0);
        } else {
          Wmm[id_mm++] = sqrt((2.0 * m + 1.0) / (2.0 * m));
        }
      }

      if ((m + 1) <= kmax) {
        Wlm_1[id_lm_1++] = sqrt((4.0 * (m + 1.0) * (m + 1.0) - 1.0) / ((m + 1.0) * (m + 1.0)
            - m * m));
      }

      for (l = m + 2; l <= kmax; l++) {
```

```
      Wlm_1[id_lm_1++] = sqrt((4.0 * l * l - 1.0) / (l * l - m * m));
      Wlm_2[id_lm_2++] = sqrt(((2.0 * l + 1.0) * (l - 1.0 + m) * (l - 1.0 - m)) / ((l * l
          - m * m) * (2.0 * l - 3.0)));
    }
  }
}


void legendre1kind(int kmax, double lat, double *nL, double *Wmm, double *Wlm_1, double *
    Wlm_2) {
// coefficients: 00, 10, 20, 30, ... 11, 21, 31, ... 22, 32, ...
  int l, m;
  int idx = 0, id_mm = 0, id_lm_1 = 0, id_lm_2 = 0;

  double sinlat, coslat;
  double tmp, tmp0, tmp1, tmp2;

  coslat = cos(lat);
  sinlat = sin(lat);

  // tested and right
  for (m = 0; m <= kmax; m++) {
    if (m == 0) {
      tmp = tmp0 = 1.0;
    } else {
      tmp = tmp0 = Wmm[id_mm++] * coslat * tmp;
    }

    if ((m + 1) <= kmax) {
      tmp1 = Wlm_1[id_lm_1++] * sinlat * tmp0;
    }

    for (l = m + 2; l <= kmax; l++) {
      tmp2 = Wlm_1[id_lm_1++] * sinlat * tmp1 - Wlm_2[id_lm_2++] * tmp0;

      nL[idx] = tmp0; tmp0 = tmp1; tmp1 = tmp2; idx++;
    }
    nL[idx] = tmp0; idx++;
    nL[idx] = tmp1; idx++;
  }
}
```

performance.h

```
// Allows to create severat timer for part of the code.
// Multiple start and stop is possible, time is counted again from the last stop.
//

#ifdef SX8
  #include <sys/time.h>
#else
  #include <time.h>
#endif

typedef struct {
  double total_time;
#ifdef SX8
  struct timeval start, stop;
#else
  struct timespec start, stop;
#endif
  int started;
} Ttimer;

void reset_timer(Ttimer *timer);

void start_timer(Ttimer *timer);
void stop_timer(Ttimer *timer);

// return laptime
double laptime_timer(Ttimer *timer);

// return total time
double totaltime_timer(Ttimer *timer);
```

performance.c

```
#include "performance.h"

void reset_timer(Ttimer *timer) {
  (*timer).total_time    = 0.0;
  (*timer).start.tv_sec  = 0;
  (*timer).stop.tv_sec   = 0;
#ifdef SX8
  (*timer).start.tv_usec = 0;
  (*timer).stop.tv_usec  = 0;
#else
  (*timer).start.tv_nsec = 0;
  (*timer).stop.tv_nsec  = 0;
#endif
  (*timer).started       = 0;
}

void start_timer(Ttimer *timer) {
  if ((*timer).started == 0) {
#ifdef SX8
    gettimeofday(&(*timer).start, NULL);
#else
    clock_gettime(CLOCK_REALTIME, &(*timer).start);
#endif
  }
  (*timer).started  = 1;
}

void stop_timer(Ttimer *timer) {
  if ((*timer).started == 1) {
#ifdef SX8
    gettimeofday(&(*timer).stop, NULL);
#else
    clock_gettime(CLOCK_REALTIME, &(*timer).stop);
#endif
  }
#ifdef SX8
  (*timer).total_time  += (double)((*timer).stop.tv_sec - (*timer).start.tv_sec) +
    (double)((*timer).stop.tv_usec - (*timer).start.tv_usec) * 1e-6;
  (*timer).start.tv_usec = 0;
  (*timer).stop.tv_usec  = 0;
#else
  (*timer).total_time  += (double)((*timer).stop.tv_sec - (*timer).start.tv_sec) +
    (double)((*timer).stop.tv_nsec - (*timer).start.tv_nsec) * 1e-9;
  (*timer).start.tv_nsec = 0;
  (*timer).stop.tv_nsec  = 0;
#endif
  (*timer).start.tv_sec = 0;
  (*timer).stop.tv_sec  = 0;
  (*timer).started      = 0;
}

double laptime_timer(Ttimer *timer) {
#ifdef SX8
  gettimeofday(&(*timer).stop, NULL);
  return (*timer).total_time + (double)((*timer).stop.tv_sec - (*timer).start.tv_sec) +
    (double)((*timer).stop.tv_usec - (*timer).start.tv_usec) * 1e-6;
#else
```

```
  clock_gettime(CLOCK_REALTIME, &(*timer).stop);
  return (*timer).total_time + (double)((*timer).stop.tv_sec - (*timer).start.tv_sec) +
    (double)((*timer).stop.tv_nsec - (*timer).start.tv_nsec) * 1e-9;
#endif
}

double totaltime_timer(Ttimer *timer) {
  return (double)(*timer).total_time;// / (double)CLOCKS_PER_SEC;
}
```

Bibliography

Amdahl, G. M. (1967), Validity of the single processor approach to achieving large-scale computing capabilities, *in* 'AFIPS Conference Proceedings', pp. 483–485.

Baur, O. (2007), Die Invariantendarstellung in der Satellitengradiometrie – Theoretische Betrachtungen und numerische Realisierung anhand der Fallstudie GOCE, Deutsche Geodätische Kommission, Series C 609, Munich, 101pp.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. (1997), 'Scalapack users' guide'.
URL: *http://www.netlib.org/scalapack/slug/*

Blackford, L. S., Choi, J., Cleary, A., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. (1996), 'Scalapack: A portable linear algebra library for distributed memory computers – design issuses and performance (technical paper)'.

EGG-C, The European GOCE Gravity Consortium (2009), GOCE Level 2 Product Data Handbook, GO-MA-HPF-GS-0110, 75pp.

ESA SP-1233 (1999), The four candidate Earth explorer core missions – gravity field and steady state ocean circulation mission, European Space Agency Report SP-1233(1), Granada.

European Space Agency (2010).
URL: *http://www.esa.int/esaLP/LPgoce.html*

Heiskanen, W. and Moritz, H. (1967), Physical Geodesy, H. W. Freeman and Company San Francisco.

Meuer, H., Strohmaier, E., Dongarra, J. and Simon, H. (2010), 'TOP500 Supercomputer Sites'.
URL: *http://www.top500.org* [May 2010]

MPI (2010).
URL: *http://www.mcs.anl.gov/research/projects/mpi/*

Netlib Repository (2010).
URL: *http://www.netlib.org*

OpenMP (2010).
URL: *http://openmp.org*

OpenMP – Wikipedia (2010).
URL: *http://en.wikipedia.org/wiki/OpenMP* (recommended by *OpenMP* (2010)) [July 2010]

Petersen, W. P. and Arbenz, P. (2004), Introduction to Parallel Computing, Oxford University Press.

Rocks Clusters (2010).
    URL: *http://www.rocksclusters.org*

Tanenbaum, A. S. (2001), Modern Operating Systems, Second Edition, Pearson Education, Inc.

Acknowledgement

I thank the High Performance Computing Center Stuttgart (HLRS) that I got the opportunity to use their computing facilities. Furthermore, I thank the staff members of the HLRS who gave me technical support which was very helpful.