An Introduction to Parallel Programming

Tobias Wittwer

An Introduction to Parallel Programming

An Introduction to Parallel Programming

Tobias Wittwer

VSSD

First edition 2006

Published by:
VSSD
Leeghwaterstraat 42, 2628 CA Delft, The Netherlands
tel. +31 15 278 2124, telefax +31 15 278 7585, e-mail: hlf@vssd.nl
internet: http://www.vssd.nl/hlf
URL about this book: **http://www.vssd.nl/hlf/a019.htm**

ISBN-10 90-71301-78-8
ISBN-13 978-90-71301-78-0
NUR 958

Keywords: parallel programming

Foreword

Any computer user knows the craving for more computing power. It was this “need for speed” that made me choose high performance computing as subject of my master thesis. This gave me the opportunity to work with systems at Stuttgart’s high performance computing center, HLRS. I spent five months parallelising programs for gravity field modelling using OpenMP and MPI, and testing them on a Cray Opteron cluster, NEC TX-7, and NEC SX-6.

After graduating I moved to the Netherlands for my PhD research in the Physical and Space Geodesy Group of the Delft Institute of Earth Observation and Space System, at the faculty of Aerospace Engineering of the Delft University of Technology. My research topic, regional gravity field modelling, proved to be computationally intensive. Luckily, we had access to Teras and Aster, two supercomputers at the SARA supercomputing facility in Amsterdam. My programs were quickly parallelised, shortening program runs from hours to minutes.

Seeing my colleagues struggle with the limited power of their PCs gave me the idea of writing a tutorial about parallel programming, to give everyone the opportunity to easily parallelise her or his programs. More urge was added when our group decided to buy its own Linux cluster.Now all I needed was an example program - and when I had to write a program for spherical harmonic analysis to check some results, I had this as well.

A week of coding and writing ensued. Test computations, creating graphics, proofreading, and finetuning followed. My supervisor Prof. Dr.-Ing. Roland Klees reviewed the document and gave his approval. I hope that the finished product will be useful to the reader, and as enjoyable to read as it was to write.

Tobias Wittwer, Delft, November 2006

Contents

Chapter 1

Introduction

1.1 Goal

Many scientific computations require a considerable amount of computing time. This computing time can be reduced by distributing a problem over several processors. Multiprocessor computers used to be quite expensive, and not everybody had access to them. Since 2005, x86-compatible CPUs designed for desktop computers are available with two “cores”, which essentially makes them dualprocessor systems. More cores per CPU are to follow.

This cheap extra computing power has to be used efficiently, which requires parallel programming. Parallel programming methods that work on dual-core PCs also work on larger shared memory systems, and a program designed for a cluster or other type of distributed memory system will also perform well on your dual-core (or multi-core) PC.

The goal of this tutorial is to give an introduction into all aspects of parallel programming that are necessary to write your own parallel programs. To achieve this, it explains

- the various existing architectures of parallel computers,
- the software needed for parallel programming, and how to install and configure it,
- how to analyse software and find the points were parallelisation might be helpful,
- how to write parallel programs for shared memory computers using OpenMP,
- how to write parallel programs for distributed memory computers using MPI and ScaLAPACK.

This tutorial aims mainly at writing parallel programs for solving linear equation systems. I hope that it is also useful to give some help for parallelising programs for other applications.

1.2 Prerequisites

This introduction to parallel programming assumes that you

- work under Linux, as it is the most common platform for high performance computing,
- use the Intel `ifort` or GNU `gfortran` Fortran compiler, as these are freely available (Intel only for non-commercial purposes) OpenMP-capable compilers.

This tutorial should also be useful for people using different system configurations and/or programming languages. All the examples use above mentioned configuration, but can easily be adapted to other configurations. The example programs are written in Fortran, but should also be understandable for C programmers. OpenMP and MPI work very similar in C/C++, with only a slightly different syntax (`#OMP PARALLEL FOR` instead of `!$OMP PARALLEL DO`, different argument types). For ScaLAPACK programming, I recommend using Fortran, as ScaLAPACK can be a little awkward to use with C/C++.

1.3 Example Program

The example program SHALE implements spherical harmonical analysis (SHA) using least-squares estimation of the spherical harmonic coefficients. I consider SHA to be well suited as an example, as it is quite simple and understandable, can be run in various problem sizes, and offers several starting points for parallel implementation. The functional model can easily be exchanged for other functional models, making SHALE a good example for your own parallelised parameter estimation programs.

All versions of SHALE described in this tutorial are available from the author's website, `http://www.lr.tudelft.nl/psg` → Staff → Tobias Wittwer → personal homepage.

Chapter 2

System Architectures

A system for the categorisation of the system architectures of computers was introduced by Flynn (1972). It is still valid today and cited in every book about parallel computing. It will also be presented here, expanded by a description of the architectures actually in use today.

2.1 Single Instruction - Single Data (SISD)

The most simple type of computer performs one instruction (such as reading from memory, addition of two values) per cycle, with only one set of data or operand (in case of the examples a memory address or a pair of numbers). Such a system is called a *scalar computer*.

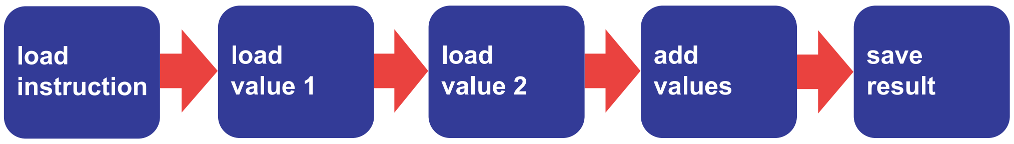

Figure 2.1: Summation of two numbers

Figure 2.1 shows, in a simplified manner, the summation of two numbers in a scalar computer. As a scalar computer performs only one instruction per cycle, five cycles are needed to complete the task - only one of them dedicated to the actual addition. To add n pairs of numbers, $n \cdot 5$ cycles would be required. To make matters even worse, in reality each of the steps shown in figure 2.1 is actually composed of several sub-steps, increasing the number of cycles required for one summation even more.

The solution to this inefficient use of processing power is *pipelining*. If there is one functional unit available for each of the five steps required, the addition still requires five cycles. The advantage is that with all functional units being busy at the same time, one result is produced every cycle. For the summation of n pairs of numbers, only $(n-1)+5$ cycles are then required. Figure 2.2 shows the summation in a pipeline.

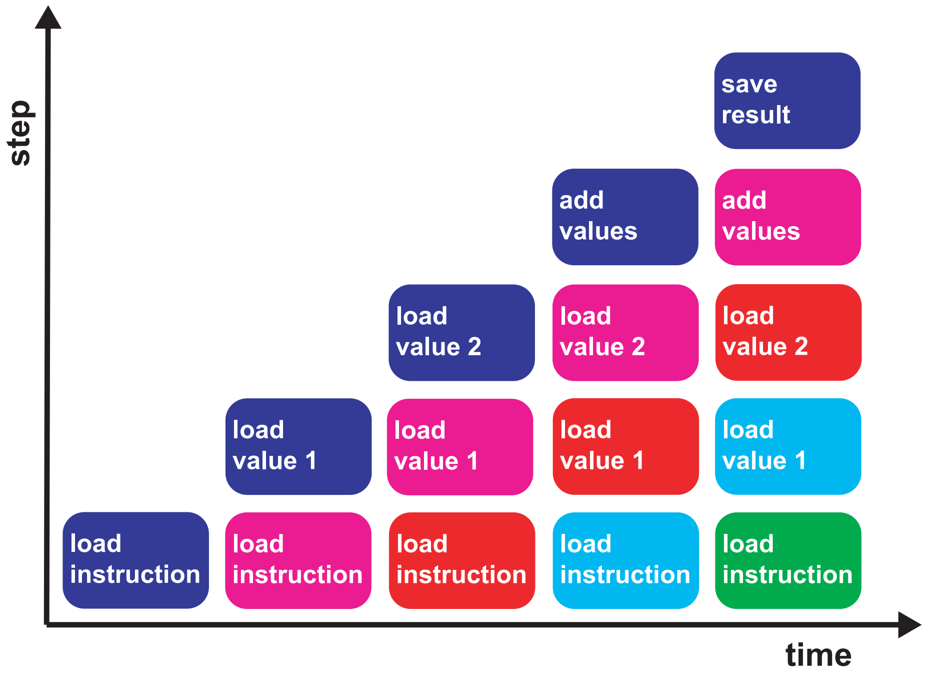

Figure 2.2: Summation of two numbers in a pipeline

As the execution of instructions usually takes more than five steps, pipelines are made longer in real processors. Long pipelines are also a prerequisite for achieving high CPU clock speeds. These long pipelines generate a new problem. If there is a branching event (such as due to an *if*-statements), the pipeline has to be emptied and filled again, and there is a number of cycles equal to the pipeline length until results are again delivered. To circumvent this, the number of branches should be kept small (avoiding and/or smart placement of *if*-statements). Compilers and CPUs also try to minimise this problem by “guessing” the outcome (branch prediction).

The power of a processor can be increased by combining several pipelines. This is then called a *superscalar* processor. Fixed-point and logical calculations (performed in the *ALU* - Arithmetic/Logical Unit) are usually separated from floating-point math (done by the *FPU* - Floating Point Unit). The FPU is commonly subdivided in a unit for addition and one for multiplication. These units may be present several times, and some processors have additional functional units for division and the computation of square roots.

To actually gain a benefit from having several pipelines, these have to be used at the same time. Parallelisation is necessary to achieve this.

2.2 Single Instruction - Multiple Data (SIMD)

The scalar computer of the previous section performs one instruction on one data set only. With numerical computations, we often handle larger data sets on which the same operation (the same instruction) has to be performed. A computer that performs one instruction on several data sets is called a *vector* computer.

Vector computers work just like the pipelined scalar computer of figure 2.2. The difference is that instead of processing single values, vectors of data are processed in one cycle. The number of values in a vector is limited by the CPU design. A vector processor than can simultaneously work with 64 vector elements can also generate 64 results per cycle - quite an improvement over the scalar processor from the previous section, which would require at least 64 cycles for this.

To actually use the theoretically possible performance of a vector computer, the calculations themselves need to be vectorised. If a vector processor is fed with single values only, it cannot perform decently. Just like with a scalar computer, the pipelines need to be kept filled.

Vector computers used to be very common in the field of high performance computing, as they allowed very high performance even at lower CPU clock speeds. In the last years, they have begun to slowly disappear. Vector processors are very complex and thus expensive, and perform poorly with non-vectorisable problems. Today's scalar processors are much cheaper and achieve higher CPU clock speeds. Vectorisation is not dead, though. With the Pentium III, Intel introduced *SSE* (Streaming SIMD Extensions), which is a set of vector instructions. In certain applications, such as video encoding, the use of these vector instructions can offer quite impressive performance increases. More vector instructions were added with SSE2 (Pentium 4) and SSE3 (Pentium 4 Prescott).

2.3 Multiple Instruction - Multiple Data (MIMD)

Up to this point, we only considered systems that process just one instruction per cycle. This applies to all computers containing only one processing core (with multi-core CPUs, single-CPU systems can have more than one processing core, making them MIMD systems). Combining several processing cores or processors (no matter if scalar or vector processors) yields a computer that can process several instructions and data sets per cycle. All high performance computers belong to this category, and with the advent of multi-core CPUs, soon all computers will. MIMD systems can be further subdivided, mostly based on their memory architecture.

2.4 Shared Memory

In MIMD systems with shared memory (SM-MIMD), all processors are connected to a common memory (*RAM* - Random Access Memory), as shown in figure 2.3. Usually all processors are identical and have equal memory access. This is called *symmetric multiprocessing* (SMP).

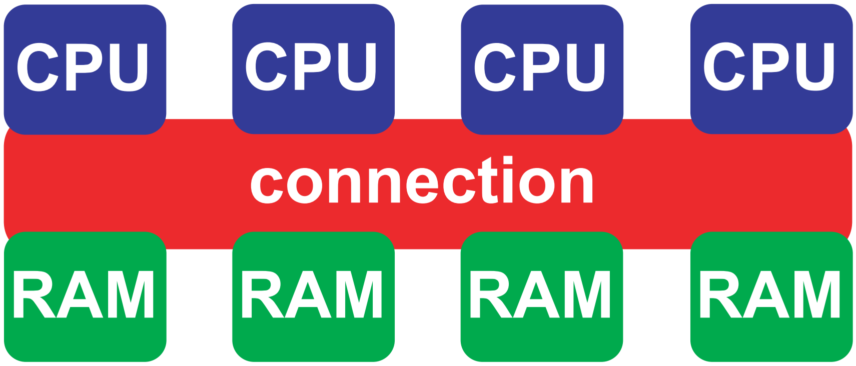

Figure 2.3: Structure of a shared memory system

The connection between processors and memory is of predominant importance. Figure 2.4 shows a shared memory system with a bus connection. The advantage of a bus is its expandability. A huge disadvantage is that all processors have to share the bandwidth provided by the bus, even when accessing different memory modules. Bus systems can be found in desktop systems and small servers (frontside bus).

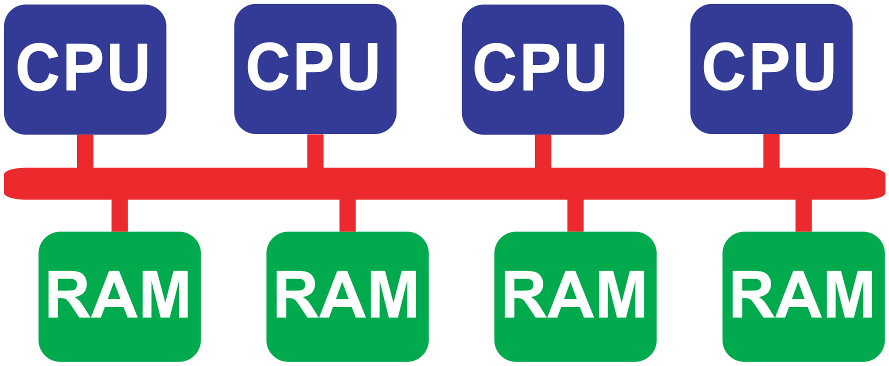

Figure 2.4: Shared memory system with bus

To circumvent the problem of limited memory bandwidth, direct connections from each CPU to each memory module are desired. This can be achieved by using a crossbar switch (figure 2.5). Crossbar switches can be found in high performance computers and some workstations.

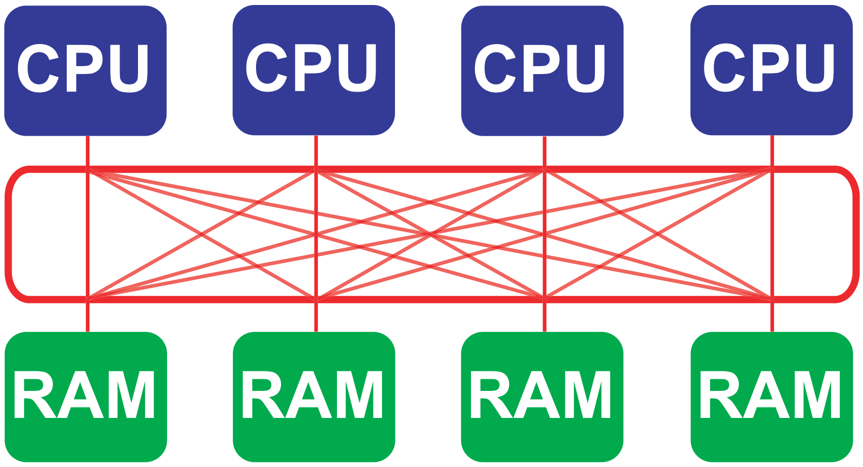

Figure 2.5: Shared memory system with crossbar switch

The problem with crossbar switches is their high complexity when many connections need to be made. This problem can be weakened by using multi-stage crossbar switches, which in turn leads to longer communication times. For this reason, the number of CPUs and memory modules than can be connected by crossbar switches is limited.

The big advantage of shared memory systems is that all processors can make use of the whole memory. This makes them easy to program and efficient to use. The limiting factor to their performance is the number of processors and memory modules that can be connected to each other. Due to this, shared memory-systems usually consist of rather few processors.

2.5 Distributed Memory

As could be seen in the previous section, the number of processors and memory modules cannot be increased arbitrarily in the case of a shared memory system. Another way to build a MIMD-system is distributed memory (DM-MIMD).

Each processor has its own local memory. The processors are connected to each other (figure 2.6). The demands imposed on the communication network are lower than in the case of a shared memory system, as the communication between processors may be slower than the communication between processor and memory.

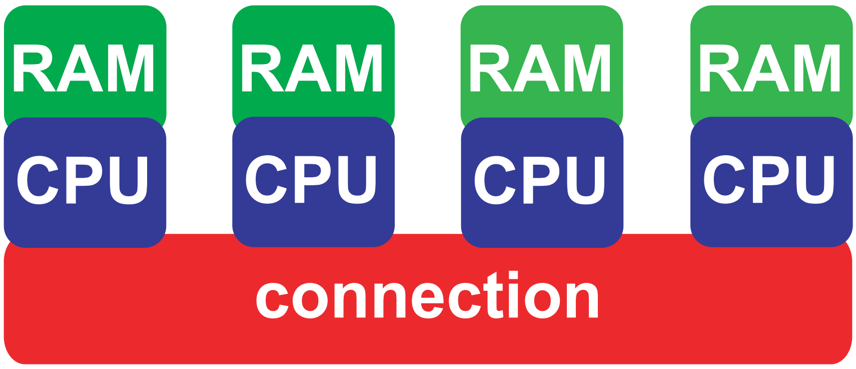

Figure 2.6: Structure of a distributed memory system

Distributed memory systems can be hugely expanded. Several thousand processors are not uncommon, this is called *massively parallel processing* (MPP). To actually use the theoretical performance, much more programming effort than with shared memory systems is required. The problem has to be subdivided into parts that require little communication. The processors can only access their own memory. Should they require data from the memory of another processor, then these data have to be copied. Due to the relatively slow communications network between the processors, this should be avoided as much as possible.

2.6 ccNUMA

The two previous sections showed that shared memory systems suffer from a limited system size, while distributed memory systems suffer from the arduous communication between the memories of the processors. A compromise is the ccNUMA (cache coherent non-uniform memory access) architecture.

A ccNUMA system (figure 2.7) basically consists of several SMP systems. These are connected to each other by means of a fast communications network, often crossbar switches. Access to the whole, distributed or non-unified memory is possible via a common cache.

A ccNUMA system is as easy to use as a true shared memory system, at the same time it is much easier to expand. To achieve optimal performance, it has to be made sure that local memory is used, and not the memory of the other modules, which is only accessible via the slow communications network. The modular structure is another big advantage of this architecture. Most ccNUMA system consist of modules that can be plugged together to get systems of various sizes.

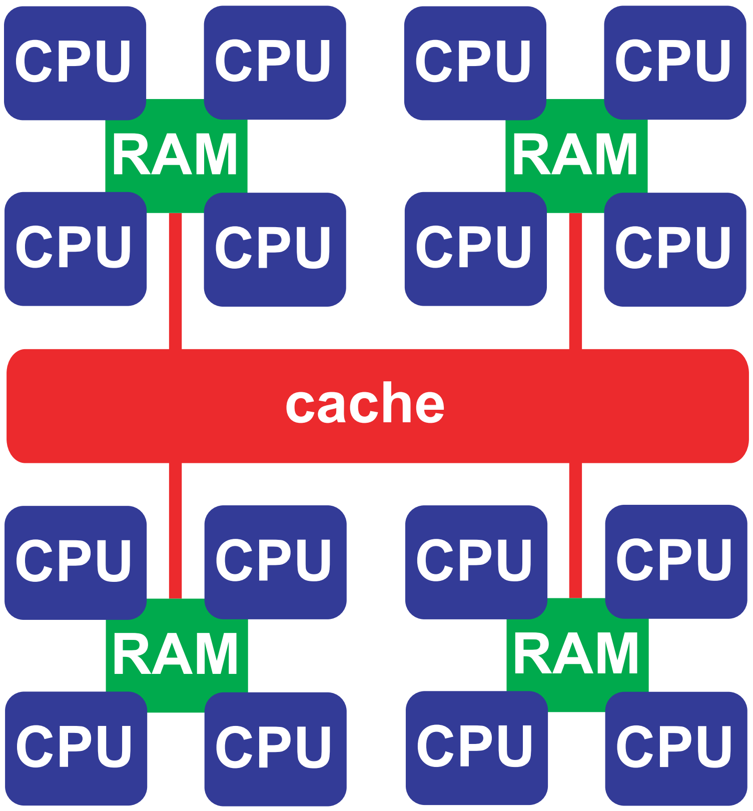

Figure 2.7: Structure of a ccNUMA system

2.7 Cluster

For some years now clusters are very popular in the high performance computing community. A cluster consists of several cheap computers (nodes) linked together. The simplest case is the combination of several desktop computers - known as a *network of workstations* (NOW). Most of the time, SMP systems (usually dual-CPU system with Intel or AMD CPUs) are used because of their good value for money. They form *hybrid* systems. The nodes, which are themselves shared memory systems, form a distributed memory system (figure 2.8).

The nodes are connected via a fast network, usually *Myrinet* or *Infiniband*. Gigabit Ethernet has approximately the same bandwidth of about 100 MB/s and is a lot cheaper, but the latency (travel time of a data package) is much higher. It is about 100 μs for Gigabit Ethernet compared to only 10 - 20 μs for Myrinet. Even this is a lot of time. At a clock speed of 2 GHz, one cycle takes 0.5

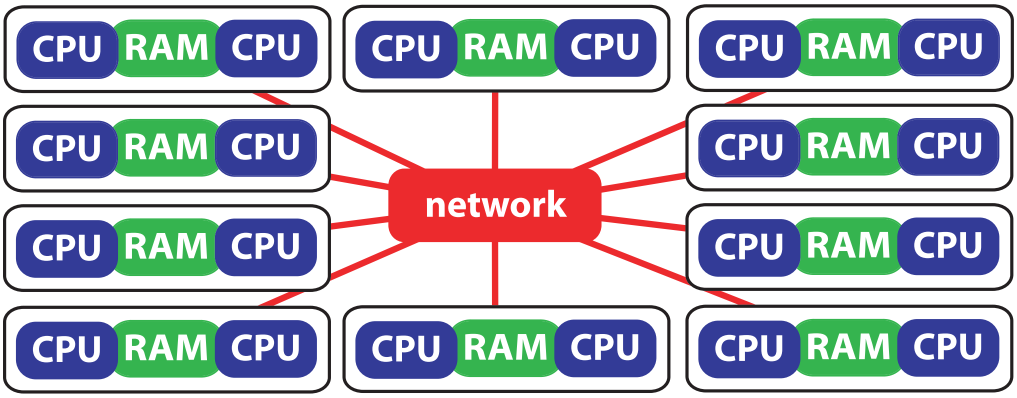

Figure 2.8: Structure of a cluster of SMP nodes

ns. A latency of 10 μs amounts to 20,000 cycles of travel time before the data package reaches its target.

Clusters offer lots of computing power for little money. It is not that easy to actually use the power. Communication between the nodes is slow, and as with conventional distributed memory systems, each node can only access its local memory directly. The mostly employed PC architecture also limits the amount of memory per node. 32 bit systems cannot address more than 4 GB of RAM, and x86-64 systems are limited by the number of memory slots, the size of the available memory modules, and the chip sets. Despite these disadvantages, clusters are very successful and have given traditional, more expensive distributed memory systems a hard time. They are ideally suited to problems with a high degree of parallelism, and their modularity makes it easy to upgrade them.

In recent years, the cluster idea has been expanded to connecting computers all over the world via the internet. This makes it possible to aggregate enormous computing power. Such a widely distributed system is known as a *grid*.

2.8 Multiple Instruction - Single Data (MISD)

The attentive reader may have noticed that one system architecture is missing: *Multiple Instruction - Single Data* (MISD). Such a computer is neither theoretically nor practically possible in a sensible way. Openshaw et al. (1999) write: "We found it hard to figure out why you would want to do this (the simultaneous manipulation of one data set with several operations) unless you are a computer scientist interested in weird computing! It is a highly specialised and seemingly a very restrictive form of parallelism that is often impractical, not to mention useless, as the basis for a general-purpose machine."

2.9 Some Examples

This section presents a few common multiprocessor/multi-core architectures. A much more extensive and detailed description is given in the "Overview of recent supercomputers", which is updated once a year, and available on-line at `http://www.phys.uu.nl/~euroben/`.

Twice a year, a list of the 500 fastest computers in the world is published. The ranking is based on the LINPACK benchmark. Although this is an old benchmark with little practical reference, the Top 500 list gives a good overview of the fastest computers and the development of supercomputers. The list can be viewed on-line at `http://www.top500.org`.

2.9.1 Intel Pentium D

The Intel Pentium D was introduced in 2005. It is Intel's first dual-core processor. It integrates two cores, based on the NetBurst design of the Pentium 4, on one chip. The cores have their own caches and access the common memory via the frontside bus. This limits memory bandwidth and slows the system down in the case of memory-intensive computations. The Pentium D's long pipelines allow for high clock frequencies (at the time of writing up to 3.73 GHz with the Pentium D 965), but may cause poor performance in the case of branches. The Pentium D is not dual-CPU-capable. This capability is reserved for the rather expensive Xeon CPU. The Pentium D supports SSE3 and x86-64 (the 64bit-extension of the x86 instruction set).

2.9.2 Intel Core 2 Duo

Intel's successor to the Pentium D is similar in design to the popular Pentium M design, which in turn is based on the Pentium III, with ancestry reaching back to the Pentium Pro. It abandons high clock frequencies in the favour of more efficient computation. Like the Pentium D, it uses the frontside bus for memory access by both CPUs. The Core 2 Duo supports SSE3 and x86-64.

2.9.3 AMD Athlon 64 X2 & Opteron

AMD's dual-core CPUs Athlon 64 X2 (single-CPU only) and Opteron (depending on model up to 8 CPUs in one system possible) are very popular CPUs for Linux clusters. They offer goodt performance at affordable prices and reasonable power consumption. Each core has its own HyperTransport channel for memory access, making these CPUs well suited for memory-intensive applications. They also support SSE3 and x86-64.

2.9.4 IBM pSeries

The pSeries is IBM's server- and workstation line based on the POWER processor. The newer POWER processors are multi-core designs and feature large caches. IBM builds shared memory systems with up to 32 CPUs. One large pSeries installation is the JUMP cluster at Kernforschungszentrum Jülich, Germany (`http://jumpdoc.fz-juelich.de`).

2.9.5 IBM BlueGene

BlueGene is an MPP (massively parallel processing) architecture by IBM. It uses rather slow 700 MHz PowerPC processors. These processors form very large, highly integrated distributed memory systems, with fast communication networks (a 3D-Torus, like the Cray T3E). At the time of writing, position one and three of the Top 500 list were occupied by BlueGene systems. The fastest system, BlueGene/L (`http://www.llnl.gov/asc/computing_resources/bluegenel/`), consists of 131,072 CPUs, and delivers a performance of up to 360 TeraFLOPS.

2.9.6 NEC SX-8

The NEC SX-8 is the one of the few vector supercomputers in production at the moment. It performs vector operations at a speed of 2 GHz, with eight operations per clock cycle. One SX-8 node consists of eight CPUs, up to 512 nodes can be connected. The biggest SX-8 installation is, at the time of writing, the 72-node system at Höchstleistungsrechenzentrum Stuttgart (HLRS), Germany.

2.9.7 Cray XT3

Cray is the most famous name in supercomputing. Many of its designs were known not only for their performance, but also for their design. The Cray XT3 is a massively-parallel system using AMD's Opteron CPU. The biggest installation of an XT3 is "Red Storm" at Sandia National Laboratories (`http://www.sandia.gov/ASC/redstorm.html`) with 26,544 dual-core Opteron CPUs, good for a performance of more than 100 TFLOPS and the second position in the November 2006 Top 500 list.

2.9.8 SGI Altix 3700

The SGI Altix 3700 is a ccNUMA system using Intel's Itanium 2 processor. The Itanium 2 has large caches and good floating point performance. Being ccNUMA, the Altix 3700 is easy to program. Aster at SARA, Amsterdam, the Netherlands (`http://www.sara.nl/userinfo/aster/description/index.html`) is an Altix 3700 with 416 CPUs.

Chapter 3

Software

3.1 Compiler

3.1.1 OpenMP

OpenMP (OpenMP, 2002) is a standard for writing multithreaded programs. Multithreaded means that parts of a program are run in parallel. Data exchange between threads is done implicitly by accessing memory. OpenMP has the advantage that it is easy to use for parallelising loops. Its disadvantage is that it is only usable on shared memory systems. Since Version 9.1, the Intel compilers also support Cluster OpenMP, an OpenMP version that uses message passing.

The OpenMP standard can be downloaded from `http://www.openmp.org`. OpenMP has to be supported by the compiler. At the moment, there are two freely available (Intel only for non-commercial purposes) OpenMP-capable compilers: Intel icc/ifort and GNU gcc/gfortran.

3.1.2 Intel Fortran

The Intel compilers icc (C/C++) and ifort (Fortran 77/90/95) are available from Intel's website, `http://www.intel.com`. After registration, you will receive an e-mail with license code and download link. After downloading, ifort is installed by

```
tar xfz l_fc_c_9.1.036.tar.gz
cd l_fc_c_9.1.036
./install.sh
```

Default installation directory is `/opt/intel/fce/9.1.036`. Add the binary directory (in the example `/opt/intel/fce/9.1.036/bin`) to your `PATH`. It may also be necessary to add `/opt/intel/fce/9.1.036/lib` to your dynamic linker search path. How to do this depends on your Linux distribution, usually it is done by editing `/etc/ld.so.conf` and running `ldconfig`.

The Intel compilers come with extensive documentation. A few common options are shown here:

- `-L/`*`path`*: Search path for linking against libraries,
- `-xW`: enable optimisations for SSE2-capable CPUs (Pentium 4, newer AMD CPUs),
- `-xP`: enable optimisations for SSE3-capable CPUs (newer Pentium 4s, Intel Core and newer). Although some AMD CPUs support SSE3, programs compiled with `-xP` will not run on these due to processor detection,
- `-O3`: full optimisations,
- `-openmp`: enable OpenMP.

There are many other options, especially concerning optimisations. Feel free to experiment with these. In my experience, they offer little performance benefit with decent code, especially if most computations are done by library routines (BLAS, LAPACK).

Since version 9.1, the Intel compilers also support Cluster OpenMP, a version of OpenMP for distributed memory systems. Unfortunately, it is not freely available at this time.

3.1.3 gfortran

gfortran is a rewrite of the GNU g77 Fortran compiler. It is available from `http://gcc.gnu.org/wiki/GFortran`. It supports Fortran 77, 90 and 95. The newest versions also have support for OpenMP. If you're lucky, your distribution may already come with a current version of gfortran. If not, you should look for prebuilt binary packages, or you may have to compile gfortran from source. The gfortran website will give further assistance.

Common compile options are:

- `-L/`*`path`*: Search path for linking against libraries,
- `-O3`: full optimisations,
- `-fopenmp`: enable OpenMP.

3.2 BLAS & LAPACK

BLAS (basic linear algebra software) and LAPACK (linear algebra package) are standards for linear algebra routines. They are widely used and available in optimised versions for various machines.

3.2.1 Reference BLAS

The reference BLAS (Lawson et al., 1979) is the reference implementation of the BLAS standard. It is usually slower than machine-optimised versions, but can be used if no optimised libraries are accessible. It is available from `http://www.netlib.org/blas/`. Installation is done by

```
mkdir BLAS
mv blas.tgz BLAS
cd BLAS
tar xfz blas.tgz
ifort -O3 -c *.f (or whatever compiler/options you want to use)
ar r libblas.a *.o
```

You can then link against BLAS with `-lblas`.

3.2.2 Reference LAPACK

The reference LAPACK (Anderson et al., 1999) is the reference implementation of the LAPACK standard. Its performance is heavily dependent on the underlying BLAS implementation. LAPACK is available from `http://www.netlib.org/lapack/`. Installation is done by

```
tar xfz lapack.tgz
cd LAPACK
cp INSTALL/make.inc.LINUX make.inc
```

edit the `make.inc` file according to the compiler you use. Change `BLASLIB` to point your BLAS library. Change `LAPACKLIB` to `liblapack.a`. A simple make should then compile LAPACK. You can now link against LAPACK with `-llapack -lblas`.

3.2.3 Intel MKL

The Intel MKL (Math Kernel Library) implements (among others functionality, such as FFT) the BLAS and LAPACK functionality. It is optimised for Intel CPUs. The non-commercial version of the MKL for Linux is available from Intel's Website, `http://www.intel.com`. After registration, you receive an e-mail with the license code and a download link. After downloading, the following commands with install the MKL:

```
tar xfz l_mkl_p_8.1.1.004.tgz
cd l_mkl_p_8.1.1.004
./install.sh
```

The default installation directory is `/opt/intel/mkl/8.1.1`. The actual libraries are in the subdirectories `lib/32` (for 32-bit systems), `lib/64` (for Itanium systems), and `lib/em64t` (for x86-64 systems). Linking against the MKL depends on your architecture, for x86_64 it is done by `-lmkl_lapack -lmkl_em64t -lguide`. It may be necessary to add the MKL library path to the search path for the dynamic linker, for example by editing `/etc/ld.so.conf` and running `ldconfig`.

3.2.4 AMD ACML

The AMD ACML (AMD Core Math Library) is AMD's optimised version of BLAS and LAPACK, and also offers some other functionality (e.g. FFT). It is, after free registration, available from `http://developer.amd.com/acml.jsp`. For multi-core or multiprocessor systems using the gfortran or Intel Fortran compiler, you should use the ACML built with gfortran. Installation is done by

```
tar xfz acml-3-5-0-gfortran-64bit.tgz
./install-acml-3-5-0-gfortran-64bit.sh
```

The default installation directory is `/opt/acml3.5.0`. The parallelised libraries are located in the `lib/gfortran64_mp` subdirectory. They can be linked against with `-lacml -lacml_mv -lgfortran -lgomp`.

3.2.5 Goto BLAS

The Goto BLAS (Goto et al., 2006) is a very fast BLAS library, probably the fastest on the x86 architecture. It is available from `http://www.tacc.utexas.edu/resources/software`. Installation is done by

```
tar xfz GotoBLAS-1.07.tar.gz
cd GotoBLAS
./quickbuild.32bit or ./quickbuild.64bit
```

The quickbuild script should automatically find the best compiler. After compilation, the library can be used by linking with `-lgoto -lpthread`.

The Goto BLAS does not come with a LAPACK library. It is possible to build the reference LAPACK with use of the Goto BLAS library.

```
tar xfz lapack.tgz
mv LAPACK LAPACK_goto
cd LAPACK_goto
cp INSTALL/make.inc.LINUX make.inc
```

edit the `make.inc` file according to the compiler you use. Change LOADOPTS to `-lpthread`. Change `BLASLIB` to point to the Goto BLAS. Change `LAPACKLIB` to `liblapack_goto.a`. A simple `make` should then compile LAPACK. Link against this version with `-llapack_goto -lgoto -lpthread`.

A LAPACK library built with Goto BLAS is fast and, like Goto BLAS, multithreaded. The number of threads used can be selected by setting the `OMP_NUM_THREADS` environment variable to the desired value.

3.2.6 ATLAS

The ATLAS (automatically tuned linear algebra software, (Whaley et al., 2005)) contains the BLAS and a subset of LAPACK. It automatically optimises the code for the machine on which it is compiled. It is available from `http://www.netlib.org/atlas/`. Installation is done by

```
tar xfz atlas3.6.0.tgz
cd ATLAS
make config cc=icc
```
(or whatever C compiler you want to use)

When asked for the options, manually choose icc if you want to use the Intel compiler. Do

`make install arch=Linux_HAMMER64SSE2_4` (or whatever your architecture may be). After compiling you can link against the multihreaded ATLAS with `-llapack -lptf77blas -lptcblas -latlas`. Note that the number of threads used by ATLAS is determined during compilation and cannot be set manually.

3.3 MPI

MPI (message passing interface) is a standard for message passing, the parallelisation method most commonly used for distributed memory architectures. There are many implementations of MPI for different system architectures, communication networks, and operating systems. The MPI standard is defined by the MPI forum and can be found on-line at `http://www.mpi-forum.org/`.

3.3.1 Open MPI

Open MPI (Gabriel et al., 2004) emerged from a number of other MPI implementations. The goal is to create the best MPI library. Open MPI supports a number of network architectures, among them shared memory, Ethernet, Myrinet, and Infiniband. Detailed information on the installation and configuration can be found on the Open MPI website, `http://www.open-mpi.org/`. Only short descriptions can be given here. After downloading, do:

```
bzip2 -d openmpi-1.1.1.tar.bz2
tar xf openmpi-1.1.1.tar
cd openmpi-1.1.1
./configure --prefix=/opt (or wherever you want Open MPI to be placed)
make all install
```

You can then compile MPI programs with `mpicc`, `mpic++`, and `mpif90`. Using these commands automatically links against the required MPI libraries. Programs are started with the `mpirun` command.

```
mpirun --hostfile hosts -np 2 ./program
```

The hostfile should contain the nodes, one node per file. For multiprocessor nodes, you may want to add `slots=`*number of processors*. Don't due this if you use multithreaded programs, use `OMP_NUM_THREADS` instead. You can also directly specify execution hosts by `--host n01,n02,...`.

3.3.2 MPICH

MPICH (Gropp et al., 1996a,b) is a popular implementation of the MPI standard. More information, software packages and installation instructions can be found at `http://www-unix.mcs.anl.gov/mpi/mpich2/`. MPICH comes with an excellent installation guide.

3.3.3 MVAPICH

MVAPICH (Liu et al., 2004) is an MPICH-based MPI implementation for Infiniband networks. It is available from `http://nowlab.cse.ohio-state.edu/projects/mpi-iba/`. Installation is done in the following way:

```
tar xfz mvapich2-0.9.5.tar.gz
cd mvapich2-0.9.5
./configure --enable-threads=multiple
make
```

After installation, you can compile your applications with `mpicc`, `mpic++` and `mpif90`.

3.4 BLACS

BLACS (Dongarra et al., 1995) are the basic linear algebra communication subprograms. They are used as communication layer by ScaLAPACK. BLACS itself makes use of PVM (parallel virtual machine) or MPI. Here, only the MPI version is discussed.

BLACS is available from `http://www.netlib.org/blacs/`. Make sure that you also download the patch for MPIBLACS. Installation is then done by

```
tar xfz mpiblacs.tgz
tar xfz mpiblacs-patch03.tgz
cd BLACS
cp BMAKES/Bmake.MPI-LINUX Bmake.inc
```

Edit the `Bmake.inc` file. Set `BTOPdir` to the correct path, add the `lib` prefix to the library names, and fill in the correct parameters for the MPI library you want to use. For ifort, `INTFACE` has to be changed to `-DAdd_`. When using an MPI library that follows the MPI2-standard (such as Open MPI), `TRANSCOMM` has to be set to `-DUseMpi2`. You may also need to change the compiler-related parameters. If everything is filled in correctly, you can start the compilation with

```
make mpi
```

If you get compile errors, it may be necessary to add symbolic links to MPI include files in the BLACS `SRC/MPI/INTERNAL` directory.

3.5 ScaLAPACK

ScaLAPACK (Blackford et al., 1997) is a library for linear algebra on distributed memory architectures. It implements routines from the BLAS and LAPACK standards. ScaLAPACK makes it possible to distribute matrices over the whole memory of a distributed memory machine, and use routines similar to the standard BLAS and LAPACK routines on them.

ScaLAPACK is available from `http://www.netlib.org/scalapack/`. After downloading, follow these steps for installation:

```
tar xfz scalapack-1.7.4.tgz
cd scalapack-1.7.4
```

Edit the `SLmake.inc` file. Modify the `home`, MPI and BLACS statements to point to the right directories and libraries. Change the compiler parameters according to your compiler. For ifort, `CDEFS` has to be changed to `-DAdd_`. You also need to specify which BLAS library to use, preferably an optimised BLAS. A simple

```
make
```

should then compile ScaLAPACK. Linking against ScaLAPACK also involves linking against BLACS, BLAS, possibly LAPACK, and MPI. When using Open MPI and Goto BLAS, the following library statements are required:

```
-lscalapack -lblacsF77init_MPI-LINUX-0 -lblacs_MPI-LINUX-0
-lblacsF77init_MPI-LINUX-0 -llapack -lgoto -lguide -lpthread
```

ScaLAPACK programs are run like MPI programs, usually by starting them with `mpirun`.

Chapter 4

Performance Analysis

4.1 Timing

Before parallelising a program, we first need to know which parts of a program need the most computation time. It does not make sense to spend a lot of time and effort parallelising program parts that contribute only very little to the total runtime.

When timing a program, there are three different time spans to be considered:

- *wall time*: The time span a "clock on the wall" would measure, which is the time elapsed between start and completion of the program. This is usually the time to be minimised.
- *user time*: The actual runtime used by the program. If this is significantly smaller than the wall time, the program has to wait a lot, for example for computation time allocation or data from the RAM or (in the worst case) from the harddisk. These are indications for necessary optimisations. When using more than one CPU, the user time should be higher than the wall time, indicating that the CPUs work in parallel.
- *system time*: Time used not by the program itself, but by the operating system, e.g. for allocating memory or harddisk access. System time should stay low.

Wall, user, and system time can be measure with the Unix command `time`:

```
time ./shale
real 3m13.535s
user 3m11.298s
sys 0m1.915s
```

`time` only measures the total runtime used by the program. For the performance analysis, we want to know the runtime required by individual parts of a program. There are several programming language and operating system dependent methods for measuring time inside a program. Both MPI and OpenMP have their own, platform independent functions for time measurement. `MPI_Wtime()` and `omp_get_wtime()` return the wall time in seconds, the difference between the results of two such function calls yields the runtime elapsed between the two function calls.

4.2 Profiling

A more advanced method of performance analysis is called *profiling*. For profiling, the program has to be built with information for the profiler. This is done with the switch `-pg` for gfortran and `-p` for Intel Fortran.

After compilation, the program has to be run regularly. This program run creates the file `gmon.out` required by the profiler `gprof`. Executing `gprof` *`program`* `> prof.txt` creates a text file with the profiling information.

The first item contained in the file is the *flat profile*. It lists all function/subroutine calls, the time used for them, the percentage of the total time, and the number of calls, among other information. The second item is the *call tree*, a listing of all routines call by the subroutines of the program.

4.3 Measuring Performance

The runtime measurements described above only give an absolute measurement of the computation time used by the program. It is also interesting to know how efficient these compuations actually are. The efficiency is the ratio between the actual performance and the theoretical performance of a system.

The floating-point performance of a computer is expressed in FLOPS - floating point operations per second. The theoretical performance of a superscalar computer is calculated as follows:

$$R_{peak} = n_{cores} \cdot n_{FPU} \cdot f, \tag{4.1}$$

where n_{core} is the number of computing cores of the computer, n_{FPU} is the number of floating-point units per core, and f is the clock frequency. For a Pentium D 830 (two cores, two FPUs per core, clock frequency 3 GHz), R_{peak} results to $2 \cdot 2 \cdot 3 \cdot 10^9$ FLOPS = 12 GFLOPS (GigaFLOPS).

The SGI Altix 3700 “Aster” at SARA, Amsterdam is equipped with 416 Itanium 2 CPUs (single-core) clocked at 1.3 GHz, resulting in an R_{peak} of $416 \cdot 4 \cdot 1.3 \cdot 10^9$ FLOPS, or 2.16 TFLOPS (TeraFLOPS).

There are several methods for measuring performance. For a dense matrix-matrix operation (such as performed by the DGEMM routine of BLAS), the number of floating-point operations required (the flop-count) is

$$n_{flop} = 2mnk, \tag{4.2}$$

with m being the number of rows of the first matrix, n being the number of columns of the second matrix, and k being the number of columns of the first matrix and the number of rows of the second matrix. The flop count of the DSYRK routine (as used in $\mathbf{N} = \mathbf{A}^T\mathbf{A}$, see chapter 5) is

$$n_{flop} = m(m+1)n, \tag{4.3}$$

with n being the number of rows of $\mathbf{A}$, and m being the number of columns of $\mathbf{A}$ and the number of rows and columns of $\mathbf{N}$.

Performance can then by calculated by $R = \frac{n_{flop}}{\text{wall time}}$, and efficency by calculating the ratio $\frac{R}{R_{peak}}$.

For Intel CPUs, Intel provides a performance measurement tool called VTUNE. The Linux version is available free of charge for non-commercial purposes from Intel’s website, `http://www.intel.com`. VTUNE offers the functionality of a profiler, but can also measure the number of integer and floating point operations during a program call.

Many supercomputers come with integrated counters for measuring performance. These make it very simple to assess performance. If you are lucky to have access to such a machine, the system’s documentation should contain information about performance measurement.

Please note that the goal of parallelisation and optimisation is not to maximise the efficiency, but to minimise the runtime required for a program. Sometimes, one algorithm may be less efficient than another, but also require a smaller number of floating point operations, resulting in a shorter runtime.

Chapter 5

SHALE - a program for spherical harmonic analysis

5.1 Spherical harmonic analysis

Spherical harmonic analysis (SHA) is Fourier analysis on the sphere. Spherical harmonics are the most popular base functions used for data analysis on the sphere. Due to the almost spherical shape of planets, spherical harmonics are a natural choice of base function for global data analysis. They are used in many fields such as geophysics, physics, and geodesy.

SHALE, the program described here, uses disturbing potential values in discrete points to compute a spherical harmonic presentation of the earth's gravity potential:

$$V = \frac{GM}{R} \sum_{n=0}^{n_{max}} \sum_{m=0}^{n} \left(\left(\bar{c}_{n,m} \cos\left(m\lambda\right) + \bar{s}_{n,m} \sin\left(m\lambda\right) \right) \bar{P}_{n,m}\left(\cos\vartheta\right) \right), \tag{5.1}$$

where GM is the geocentric gravitational constant, R is the earth radius, n is the spherical harmonic degree, m is the spherical harmonic order, λ is the longitude and ϑ the colatitude (polar distance) of a point on the sphere, the $\bar{P}_{n,m}$ are the normalised associated Legendre functions of the first kind, and $\bar{c}_{n,m}$ and $\bar{s}_{n,m}$ are the unknown spherical harmonic coefficients which we want to estimate. A problem of maximum degree n_{max} has

$$u = n_{max}^2 + 2 \cdot n_{max} + 1 \tag{5.2}$$

unknown coefficients.

By partially deriving equation 5.1 after the unknowns, we get the entries for the design matrix A that links observations i and unknowns $\bar{c}_{n,m}$, $\bar{s}_{n,m}$:

$$a_i^{\bar{c}_{n,m}} = \frac{GM}{R} \cos(m\lambda_i) \bar{P}_{n,m}(\cos\vartheta_i), \tag{5.3}$$

$$a_i^{\bar{s}_{n,m}} = \frac{GM}{R} \sin(m\lambda_i) \bar{P}_{n,m}(\cos\vartheta_i). \tag{5.4}$$

With the design matrix **A** and the observation vector **y**, we can compute the normal equation matrix **N** and the right-hand-side vector **b**:

$$\mathbf{N} = \mathbf{A}^T\mathbf{A}, \tag{5.5}$$

$$\mathbf{b} = \mathbf{A}^T\mathbf{y}. \tag{5.6}$$

The estimated coefficients are then obtained by computing

$$\hat{\mathbf{x}} = \mathbf{N}^{-1}\mathbf{b}. \tag{5.7}$$

Equation 5.1 holds only for points on the sphere with radius R. For points outside the sphere, the potential is

$$V = \frac{GM}{R} \sum_{n=0}^{n_{max}} \sum_{m=0}^{n} \left((\bar{c}_{n,m}\cos(m\lambda) + \bar{s}_{n,m}\sin(m\lambda)) \bar{P}_{n,m}(\cos\vartheta) \left(\frac{R}{r}\right)^{n+1} \right), \tag{5.8}$$

where r is the radius of the point. The partial derivatives are then

$$a_i^{\bar{c}_{n,m}} = \frac{GM}{R} \left(\frac{R}{r_i}\right)^{n+1} \cos(m\lambda_i) \bar{P}_{n,m}(\cos\vartheta_i), \tag{5.9}$$

$$a_i^{\bar{s}_{n,m}} = \frac{GM}{R} \left(\frac{R}{r_i}\right)^{n+1} \sin(m\lambda_i) \bar{P}_{n,m}(\cos\vartheta_i), \tag{5.10}$$

the estimation of the unknown coefficients is done as described in equations 5.5 to 5.7.

5.2 Direct method

5.2.1 Description

The "direct method" for estimating the unknown gravity field coefficients is implementing equations 5.5 to 5.7. The inversion of the normal equation matrix **N** is usually replaced by a decomposition of **N** and solving for the unknown coefficients.

The advantage of this method is its simplicity, and that the inverse $\mathbf{N}^{-1}$ can be formed to get error estimates. It is also easy to expand with other techniques such as regularisation for stabilising the equation system, and variance component estimation for weighting observation groups with different accuracies.

The big disadvantage of the direct method is the memory requirement. The normal equation matrix **N**, which is of the size $u \times u$ for u unknown parameters, has to be kept in memory. The design matrix **A** is even larger - it is of size $n \times u$ for n observations and u unknown parameters. Table 5.1 shows the resulting matrix sizes for some typical maximum degrees n_{max} and 100,000 observations.

n_{max}	u	size of **N**	size of **A**
20	441	1.5 MB	336.5 MB
50	2601	51.6 MB	1.9 GB
100	10201	793.9 MB	7.6 GB
200	40401	12.2 GB	30.1 GB
300	90601	61.2 GB	67.5 GB

Table 5.1: Number of unknowns and matrix sizes depending on n_{max}, for 100,000 observations

The design matrix **A** does not have to be kept in memory. Is is possible to build **A** for only a part of the observations, do the multiplication and add to **N**:

$$\mathbf{N} = \sum_{j=1}^{n_j} \mathbf{A}_j^T \mathbf{A}_j, \tag{5.11}$$

but with the disadvantage that **A** has to be built several times if, for example, residuals are to be calculated:

$$\hat{\mathbf{e}} = \mathbf{y} - \mathbf{A}\hat{\mathbf{x}}. \tag{5.12}$$

It is possible to compute $\mathbf{A}_j$ for only one observation at a time. This should be avoided, though, as the matrix multiplication $\mathbf{A}_j^T \mathbf{A}_j$ for an $\mathbf{A}_j$ of only one line is very inefficient.

Besides the memory requirement, another disadvantage of the direct method is the time-consuming dense matrix multiplication $\mathbf{N} = \mathbf{A}^T\mathbf{A}$.

5.2.2 Program structure

The structure of SHALE is shown in figure 5.1. The three computation-intensive tasks are shaded. The linewise build of $\mathbf{A}$ is done by a loop and the routine `build_a_line`. The matrix multiplication $\mathbf{N} = \mathbf{A}^T\mathbf{A}$ is done using the DSYRK routine of BLAS, $\mathbf{b} = \mathbf{A}^T\mathbf{y}$ is done using DGEMV. The solving of the linear equation systems is done by LAPACK's DPOSV routine.

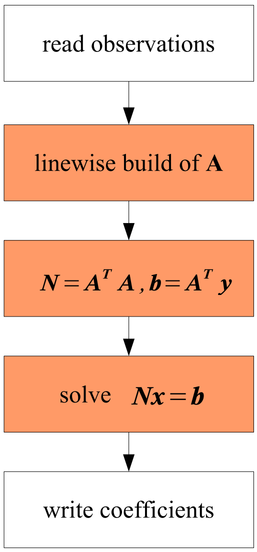

Figure 5.1: Structure of SHALE for direct method

The structure of SHALE as shown here has been implemented in SHALE V0.1 (for observations on the sphere as in eq. 5.1) and in SHALE V0.2 (for observations on and outside the sphere, eq. 5.8).

5.2.3 Program analysis

In the program structure (figure 5.1) three parts were highlighted as computationally intensive. In SHALE V0.3, timing was added to these parts. Time measurement is done using OpenMP's `omp_get_wtime()` function.

When being run, SHALE may produce the following output:

```
build A [s] :  1.693
build N [s] :  20.268
build b [s] :  0.072
solving [s] :  1.303
info :  0

total runtime [s]:  23.457
```

This shows that most of the computation time is required for the matrix multiplication $\mathbf{N} = \mathbf{A}^T\mathbf{A}$. Some computation time is also required setting up $\mathbf{A}$ and solving the linear equation system. Depending on the number of observations and unknowns, the numbers will change, but the general picture will stay the same.

5.2.4 Parallelisation with OpenMP

The performance analysis has shown three areas which can be parallelised to speed up the program: setup of design matrix $\mathbf{A}$, matrix multiplication $\mathbf{N} = \mathbf{A}^T\mathbf{A}$, and solving the linear equation system. SHALE V0.4 is parallelised in this way.

The matrix multiplication is done using the DSYRK routine of BLAS. With a multithreaded BLAS library (such as GotoBLAS or MKL), increasing the number of threads on a multiprocessor system should speed up the multiplication.

The number of threads is set to two with
`export OMP_NUM_THREADS=2` if BASH is used and
`setenv OMP_NUM_THREADS 2` in the case of C-Shell.

A new program run yields the following result:

```
build A [s] :  1.677
build N [s]:  10.894
build b [s]:  0.071
solving [s] :  0.883
info :  0

total runtime [s]:  13.646
```

Not only the time for the matrix multiplication, but also the time required for solving the equation system has been reduced. This is due to the fact that a LAPACK making use of a multithreaded BLAS library will also run faster on more processors.

Parallelising the setup of $\mathbf{A}$ requires a little more effort. Unparallelised, it looks like this:

```
do i=1,nobs
read(1,*) long, lat, rad, value
y(i) = value
long = long/rho
lat = lat/rho
call build_a_line(A,i,nobs,nmax,u,long,lat,rad,latold,pnm,rearth)

latold = lat
end do
```

This loop can be parallelised quite easily, as setup of the individual lines of *A* can be done independently. Reading in the observations is moved out of the loop, the coordinates of the points are also read into vectors. This makes it possible to execute the loop in parallel:

```
!$OMP PARALLEL DO
do i=1,nobs
call build_a_line(A,i,nobs,nmax,u,long(i),lat(i),rad(i),latold,
pnm,rearth)
latold = lat(i)
end do
!$OMP END PARALLEL DO
```

The `!$OMP PARALLEL DO` pragma indicates that the following do loop can be executed in parallel. But to our disappointment, a program parallelised in this way does not produce valid results. There are two reasons for this:

- Each thread needs its own private copy of the variable `latold`
- Each thread needs its own private array of Legendre functions `pnm`

Making a variable private to a thread is done by adding a `PRIVATE` statement to the `PARALLEL` pragma. This way, `latold` can be made private. `pnm` is slightly more complicated, as this is a dynamically allocated array. It can also be declared as private in a parallel block, but then has to be allocated by each thread. This results in the following code for the parallelised loop:

```
!$OMP PARALLEL PRIVATE(latold,pnm)
allocate(pnm(0:nmax,0:nmax))
latold = 1.6d0
!$OMP DO
do i=1,nobs
call build_a_line(A,i,nobs,nmax,u,long(i),lat(i),rad(i),latold,
pnm,rearth)
```

```
latold = lat(i)
end do
!$OMP END DO
deallocate(pnm)
!$OMP END PARALLEL
```

A run with the new program yields the desired result:

```
build A [s] :  1.277
```

Unfortunately, parallelisation did not result in a speedup close to 100%. This is due to the rather small workload. With larger problems, the resulting speedup will also be higher.

When parallelising loops, you have to make sure that loop iterations are independent from each other. A loop like

```
do i=1,n
a(i) = b(i)*a(i-1)
end do
```

cannot be parallelised. When calling subroutines inside parallelised loops, you also have to be sure that subroutine calls are independent from each other. Especially subroutines using the `SAVE` parameter for variables are dangerous.

Distribution of the loops over the threads is done by the operating system. The distribution is load-balanced, so a CPU busy with other tasks will get less loop iterations than an otherwise idle CPU.

To assess the quality of the parallelisation, a number of computations were done on a dual Opteron 280 system (two cores per CPU, 2.4 GHz), using the Goto BLAS 1.0.7 and the reference LAPACK as linear algebra libraries, for a problem of $n_{max} = 50$ and $n = 16,200$ observations. Figure 5.2 shows the resulting runtimes for the multiplication $\mathbf{N} = \mathbf{A}^T\mathbf{A}$ (DSYRK), the solving of $\mathbf{Nx} = \mathbf{b}$ (DPOSV), and the total program runtime. Figure 5.3 shows the resulting performance and efficiency of the DSYRK routine, computed according to equation 4.3. Even with this small problem, the runtime scales almost linearly, with DSYRK efficiency around 90%. This result showcases the efficiency of the Goto BLAS on this architecture.

5.2.5 Parallelisation with MPI and ScaLAPACK

As could be seen from the previous section, parallelisation with OpenMP is rather easy and straightforward, if independent loops are to be parallelised. OpenMP (with the exception of Cluster OpenMP) is limited to shared memory system. This is also true for BLAS and LAPACK, the two libraries that do the most workload in SHALE.

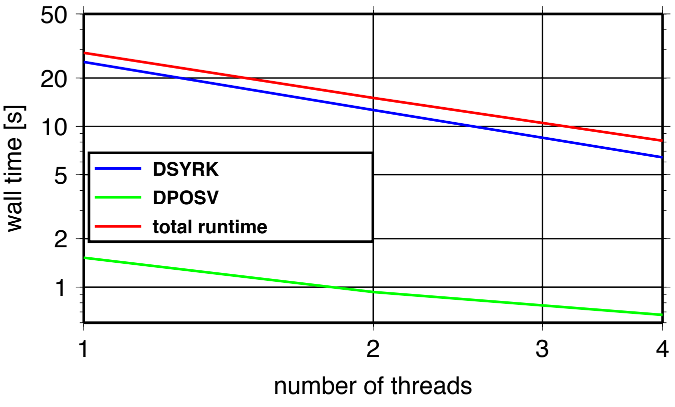

Figure 5.2: Runtimes of SHALE V0.4 with $n_{max} = 50$ and $n = 16,200$ observations

Distributed memory systems require the use of message passing methods for communication between the nodes. The most common message passing library is MPI. ScaLAPACK is a BLAS/LAPACK library for distributed memory system. ScaLAPACK makes use of BLACS as communication layer, which in turn makes use of MPI.

ScaLAPACK distributes matrices and vectors in blocks among all nodes taking part in a computation. How this distribution is actually done and how computations and communication need to be performed is not important to the user - he only needs to call the ScaLAPACK routines with the right parameters. ScaLAPACK follows the SPMD (single program - multiple data) approach of MPI. All processes run the same program. Node-dependent behaviour has to be controlled by if-statements checking for the process id. Input/Output statements, for example, should only be done by one process, usually the one with id 0.

Parallelising SHALE with ScaLAPACK involves the following steps:

- initialising the BLACS process grid,
- initialising matrix descriptors and allocating memory accordingly,
- distributed setup of design matrix **A**,
- call of routines for matrix multiplication and solving the equation system,
- gathering the estimated parameters for output,
- exiting the BLACS process grid.

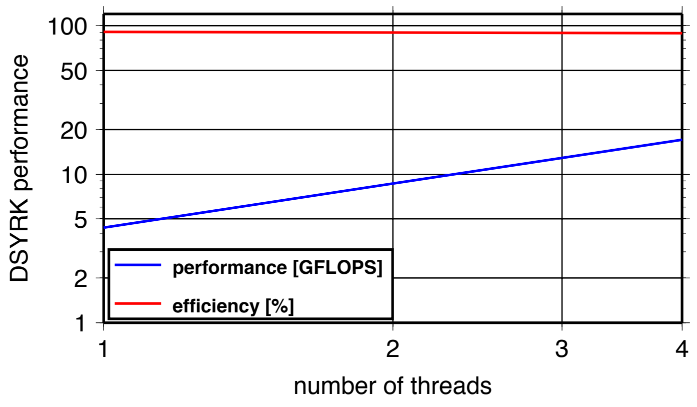

Figure 5.3: Performance of SHALE V0.4 with $n_{max} = 50$ and $n = 16,200$ observations

Initialising the BLACS process grid is done by

```
call blacs_pinfo(iam,nprocs)
call blacs_get( -1, 0, ictxt )
call blacs_gridinit(ictxt,'R',nprocs,1)
```

The variable `iam` contains the process id, `nprocs` the total number of processes. The `'R'` in `blacs_gridinit` with a process row size equal to the number of processors and a process column size of 1 defines a row-style process grid, as shown in figure 5.4. Such a grid has the advantage that vectors are distributed over all processes, and no process receives no vector elements. It is thus easy to use. The disadvantage is the lower performance compared to grids where the columns are also distributed (more than one process column).

ScaLAPACK needs a descriptor `desc` for each matrix. This descriptor contains information about the distribution of the matrix. Each process also needs to know how many rows and columns of a matrix are assigned to it, in order to allocate memory accordingly. In SHALE, local matrix size calculation and descriptor initialisation is done by the subroutine `calc_sizes_descinit`.

```
call blacs_gridinfo(ictxt,nprow,npcol,myrow,mycol)
call calc_sizes_descinit(blocksize,ictxt,myrow,mycol,nprow,npcol,
nobs,u,desca,ra,ca)
...
```

The distributed setup of **A** is pretty straightforward. The rows of **A** are distributed over the processes in a blockwise fashion, as shown in figure 5.5. Since ScaLAPACK uses a block-cyclic distribution, it is not that (in the case of two processes) the first process is assigned the first half

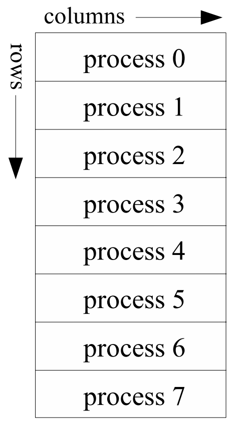

Figure 5.4: Row-style process grid

of the matrix, the second process the other half. Blocks are distributed in a round-robin fashion.

For setting up **A**, we need to know to which global row index the local row index in the loop corresponds. This information is provided by ScaLAPACK's `indxl2g` function. Except for also passing the size of the local **A** matrix, the `build_a_line` routine for setting up **A** can remain unchanged. As we have only one column of processes, each process holds all columns. The number and ordering of the unknown parameters, and thus the actual computation of one line of **A**, remains unchanged. We keep the OpenMP parallelisation for multithreaded program runs on SMP nodes, which means that `idx` also has to be private.

```
!$OMP DO
do i=1,ra
idx=indxl2g(i, blocksize, myrow, 0, nprow)
call build_a_line(A,i,nobs,ra,nmax,u,long(idx),lat(idx),rad(idx),
latold,pnm,rearth)
latold = lat(idx)
end do
!$OMP END DO
```

Matrix multiplication and linear equation solving is done by replacing the DSYRK, DGEMV, and DPOSV routines with ScaLAPACK's PDSYRK, PDGEMV, PDPOSV, and the appropriate parameters.

```
call PDSYRK('L','T',u,nobs,1.0d0,A,1,1,desca,0.0d0,N,1,1,descn)
call PDGEMV('T',nobs,u,1.0d0,A,1,1,desca,y,1,1,descy,1,0.0d0,
b,1,1,descb,1)
```

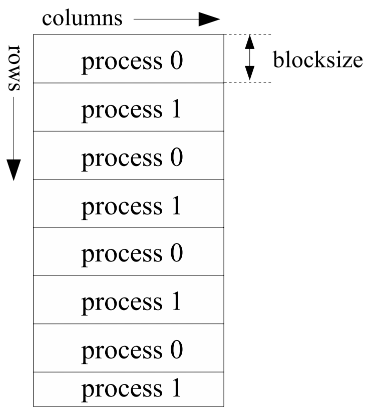

Figure 5.5: Matrix distribution over processes

```
call PDPOSV('L',u,1,N,1,1,descn,b,1,1,descb,info)
```

After solving, the estimated parameters are contained in the distributed vector *b*. For output, we need all coefficients inside one vector in process 0. This is done by writing all coefficients to the proper position in a full vector of estimated coefficients. This vector is then summed up by a global sum operation, as shown in figure 5.6.

```
x(:)  = 0.0d0
do i=1,rb
idx=indxl2g(i, blocksize, myrow, 0, nprow)
x(idx) = b(i)
end do
call dgsum2d(ictxt,'A',' ',1,u,x,1,-1,-1)
```

The final step is exiting the BLACS process grid. This is done by

```
call blacs_gridexit(ictxt)
call blacs_exit(0)
```

SHALE V0.5 is parallelised as described above. After compilation, we can start the program with

```
mpirun -np 1 ./shale
```

and may get output like this:

```
build A [s] :  1.739
build N [s] :  24.604
```

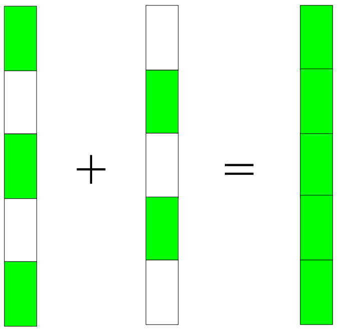

Figure 5.6: Summation of distributed coefficients

```
build b [s] :  0.076
solving [s] :  1.811
info :  0

total runtime [s]:  28.291
```

Another run with

```
mpirun -np 2 ./shale
```

yields

```
build A [s] :  1.765
build N [s] :  13.654
build b [s] :  0.072
solving [s] :  1.667
info :  0

total runtime [s]:  17.266
```

The problem size in the example is too small for $\mathbf{A}$ setup or solving benefiting significantly from the second CPU. For very small problems and slow interconnects, computation times may even increase. The matrix multiplication $\mathbf{N} = \mathbf{A}^T\mathbf{A}$ is sped up significantly, and may benefit even more for larger problem sizes.

An example problem with $n_{max} = 100$ and $n = 64,800$ observations was used on an Opteron Linux cluster (two Opteron 280 dual-core CPUs, clocked at 2.4 GHz, per node) with Infiniband interconnect. Open MPI 1.1.2, ScaLAPACK 1.7.4, and the Goto BLAS 1.0.7 were used as software packages. Figure 5.7 shows the resulting runtimes for the multiplication $\mathbf{N} = \mathbf{A}^T\mathbf{A}$ (PDSYRK), the solving of $\mathbf{Nx} = \mathbf{b}$ (PDPOSV), and the total program runtime. Figure 5.8 shows the resulting performance and efficiency of the PDSYRK routine. For 4 and 8 nodes, efficiency is around 60%. Using more than leads to a significant drop in efficency. Using 32 nodes actually slows the program down compared to 16 nodes, as too much time is spent communicating. Only the PDPOSV routine benefits from 32 nodes. Note that especially PDPOSV is sensitive to the selected ScaLAPACK block size. All results were obtained with a blocksize of 128.

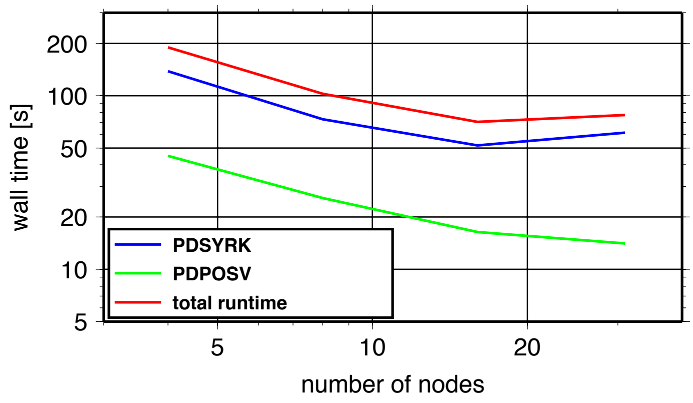

Figure 5.7: Runtimes of SHALE V0.5 with $n_{max} = 100$ and $n = 64,800$ observations

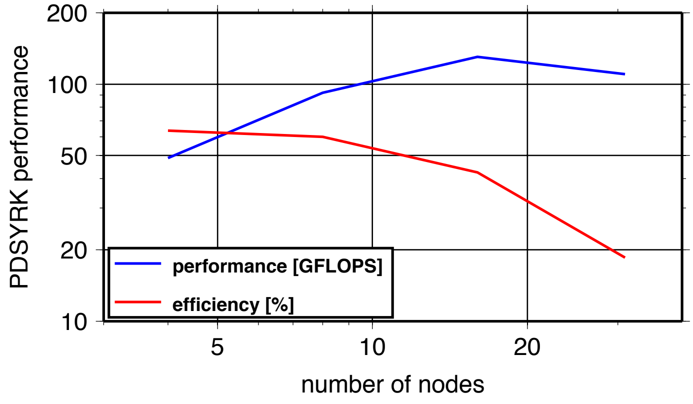

Figure 5.8: Performance of SHALE V0.5 with $n_{max} = 100$ and $n = 64,800$ observations

5.2.6 Some hints

In all computations, you should avoid *if*-statements, as branching can be quite costly in terms of computation time. Sometimes, it may be better to compute valuess and not use them at all rather than using *if*. Sometimes, it doesn't matter, as the processor's branch prediction may guess correctly (such as with the *if* in `build_a_line`, removing it does not make the program faster). If the outcome is already known at compile time, the compiler should remove the *if*-statement.

It is often sensible to rearrange loops. In SHALE, the summation of equation 5.8 has been rearranged, with the loop over the order m outside. This way, the values $\cos m\lambda$ and $\sin m\lambda$ have to be computed only once.

Values that are used several times should be precomputed and stored for later use. Memory is usually more abundant than computation time, especially if it concerns vectors of only a few kilobytes. In SHALE, the values of $\left(\frac{R}{r}\right)^{n+1}$, which are required in each loop of equation 5.8, are precomputed and stored in a vector. Often this can also be done with sqare root tables, and may lead to quite dramatic speedups.

In SHALE, the associated Legendre functions $\bar{P}_{n,m}$ are only computed if the latitude φ_i is different from the previous latitude φ_{i-1}. If the observations are sorted by latitude, this can improve the computation time considerably. This technique is used in many Fortran programs, often using Fortran's SAVE statement, which saves the values of variables between subroutine calls. Be careful, though: subroutines making use of save and being called by parallel threads will result in wrong results. Because of this, SHALE saves φ_{i-1} outside of the subroutine actually using it, with a private copy of the variable for every thread.

5.3 Conjugate gradient method

5.3.1 Description

The previous chapter presented the "direct method" for solving linear equation systems, in this case for spherical harmonic analysis. As has been said before, its drawbacks are the memory requirements and the time required for the matrix multiplication $\mathbf{N} = \mathbf{A}^T\mathbf{A}$. To circumvent these problems, iterative solving methods have been developed, with the "conjugate gradients" probably being the most popular method. With this method, it is not necessary to fully build $\mathbf{N}$. The computation only requires vector-vector operations, so very little memory is required.

The conjugate gradient method is presented here not only because it has memory (and possibly runtime) benefits compared to the direct method, but also because it is much more complicated, making it more of a challenge to parallelise efficiently.

Iterative methods can be sped up considerably if some a-priori information about the results is known. The process of adding a-priori information is called "preconditioning". The preconditioner is obtained by partly building the normal matrix N. Spherical harmonics are well suited to preconditioning, as coefficients of different order, as well as the cosine- and sine-coefficients, are almost independent from each other, leading to normal matrix entries close to zero. This means, that only blocks of the normal matrix N have to be computed and kept in memory.

The block-diagonal preconditioning matrix $\mathbf{N}_{bd}$ has

$$n_{\mathbf{N}_{bd}} = (n_{max}+1)^2 + 2\sum_{n=1}^{n_{max}} n^2 \tag{5.13}$$

elements, which can be rewritten without the sum:

$$n_{\mathbf{N}_{bd}} = 2\left(\frac{n_{max}\,(n_{max}+1)\,(2n_{max}+1)}{6}\right) + (n_{max}+1)^2 \tag{5.14}$$

This leads to a much smaller memory requirement than for the full normal matrix $\mathbf{N}$. Table 5.2 compares the memory requirements for $\mathbf{N}$ and $\mathbf{N}_{bd}$ for various values of n_{max}. All other arrays required for the preconditioned conjugate gradient method do not exceed the size of the number of unknowns, not more than a few megabytes for n_{max} = 300.

n_{max}	u	size of $\mathbf{N}$	$n_{\mathbf{N}_{bd}}$	size of $\mathbf{N}_{bd}$
20	441	1.5 MB	6181	48 KB
50	2601	51.6 MB	88451	691 KB
100	10201	793.9 MB	686901	5.2 MB
200	40401	12.2 GB	5413801	41.3 MB
300	90601	61.2 GB	18180701	138.7 MB

Table 5.2: Number of unknowns and matrix sizes depending on n_{max}

Here, the conjugate gradient method with preconditioning and Schönauer smooting (Schönauer, 2000) is used, as implemented by Ditmar et al. (2003) for gravity field recovery from the GOCE satellite mission. It is too complex to explain here, but I want to show the equations behind it. Bold Latin letters correspond to vectors, Greek letters to scalar values:

1. $\mathbf{x}_0 = \tilde{\mathbf{x}}_0 = 0, \mathbf{b}_0 = \tilde{\mathbf{b}}_0 = \mathbf{A}^T\mathbf{y}, \mathbf{p}_0 = \tilde{\mathbf{p}}_0 = \mathbf{N}_{bd}^{-1}\mathbf{b}_0, k = 1$
2. $\mathbf{a}_k = \mathbf{N}\mathbf{p}_k = \mathbf{A}^T\mathbf{A}\mathbf{p}_k$
3. $\alpha_k = \frac{\mathbf{b}_k^T\mathbf{p}_k}{\mathbf{a}_k^T\mathbf{p}_k}$

4. $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$
5. $\mathbf{b}_{k+1} = \mathbf{b}_k - \alpha_k \mathbf{a}_k$
6. $\mathbf{e}_k = \mathbf{N}_{bd}^{-1}\left(\tilde{\mathbf{b}}_k - \mathbf{b}_{k+1}\right)$
7. $\gamma_k = -\frac{\mathbf{b}_{k+1}^T \mathbf{e}_k}{\left(\tilde{\mathbf{b}}_k - \mathbf{b}_{k+1}\right)^T \mathbf{e}_k}$
8. $\tilde{\mathbf{x}}_{k+1} = \mathbf{x}_{k+1} + \gamma_k \left(\tilde{\mathbf{x}}_k - \mathbf{x}_{k+1}\right)$
9. $\tilde{\mathbf{b}}_{k+1} = \mathbf{b}_{k+1} + \gamma_k \left(\tilde{\mathbf{b}}_k - \mathbf{b}_{k+1}\right)$
10. If $\frac{\|\tilde{\mathbf{b}}_{k+1}\|}{\|\tilde{\mathbf{b}}_0\|} < \varepsilon_1$ and difference $\left(\tilde{\mathbf{x}}_k, \tilde{\mathbf{x}}_{k+1}\right) < \varepsilon_2$, set $\mathbf{x} = \tilde{\mathbf{x}}_{k+1}$ and stop
11. $\tilde{\mathbf{p}}_{k+1} = \mathbf{N}_{bd}^{-1}\mathbf{b}_{k+1}$
12. $\beta_{k+1} = \frac{\mathbf{b}_{k+1}^T \tilde{\mathbf{p}}_{k+1}}{\mathbf{b}_k^T \tilde{\mathbf{p}}_k}$
13. $\mathbf{p}_{k+1} = \tilde{\mathbf{p}}_{k+1} + \beta_{k+1}\mathbf{p}_k$
14. $k = k+1$, go to step (2)

The stopping criteria ε_1 and ε_2 are usually set to 10^{-6}.

5.3.2 Program structure

The program structure for SHALE with the preconditoned conjugate gradient method as solver is shown in figure 5.9. Once again, computationally intensive steps are shaded. These are the build of the preconditioner and the vector-vector operations in each iteration.

5.3.3 Program analysis

The preconditioned conjugate gradient method as described above has been implemented in SHALE CG V0.2. A program run with a test data set and $n_{max} = 100$ leads to convergence in the fourth iteration. This is surprising at first, but, with noise-free data in a perfect distribution and a good preconditioner, not unexplicable.

A profiling run, as described in section 4.2, indicates that more than 90% of the runtime is required for building the preconditioner $\mathbf{N}_{bd}^{-1}$, but with very little time required for the blockwise inversion. Simply using two threads instead of one on a dual-core system (remember that a

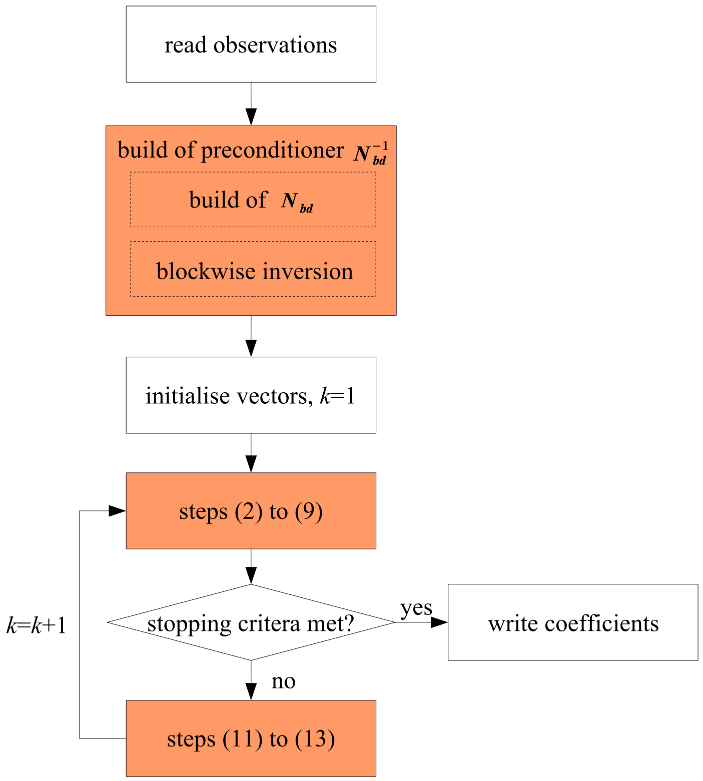

Figure 5.9: Structure of SHALE for preconditioned conjugate gradient method

multithreaded LAPACK library is used) should thus not result in a significantly faster runtime - and it does not.

About 4% of the runtime are spent for the linewise build of $\mathbf{A}$, the rest in a number of other routines.

5.3.4 Parallelisation with OpenMP

The performance analysis showed that most of the runtime, at least when the problem converges quickly, is required for building the matrix $\mathbf{N}_{bd}$. It thus makes sense to parallelise this part of the program first.

A closer look at the subroutine `build_precon` shows that it consists of a series of loops, with the outermost loop looping over the observations. Parallelising this loop is easy, done in the same way as with the linewise build of **A** for the direct method (5.2.4), making sure that all required variables are set PRIVATE.

```
!$OMP PARALLEL PRIVATE(latold,a_row,pnm,idx,offset,m,j,k)
allocate(pnm(0:nmax,0:nmax))
allocate(a_row(u))
latold=1.6d0
!$OMP DO
do i=1,nobs
...
end do
!$OMP END DO
deallocate(a_row)
deallocate(pnm)
!$OMP END PARALLEL
```

To our dismay, the build of $\mathbf{N}_{bd}$ is not sped up, and problem does not converge any more: the preconditioner was not built correctly. As all threads may access $\mathbf{N}_{bd}$ at the same time, some summations are not done correctly.

This problem can be circumvented by giving each thread its own copy of $\mathbf{N}_{bd}$, and combining them later.

```
threadnum = omp_get_thread_num()+1
...
N2(idx,threadnum) = N2(idx,threadnum) + a_row(j)*a_row(k)
...
do j=1,numthreads
do i=1,size_nbd
N(i) = N(i)+N2(i,j)
end do
end do
```

A program modified in this way (SHALE CG V0.4) still shows no performance improvement on a Pentium D. On a dual-CPU dual-core Opteron system, going from one to two threads reduces the computation time slightly. Four threads do not lead to a significant speedup.

The build of $\mathbf{N}_{bd}$ requires the processing of very small vectors with a lot of memory access. As both cores of the Pentium D use the same bus for memory access, this is the bottleneck. The Opteron benefits slightly from its independent HyperTransport connection to memory, but the intensive memory access still limits the performance that can be reached.

In SHALE CG V0.5, the build of the preconditioner has been modified to use blocks of **A**, as in equation 5.11. The blocks are distributed over the threads. This is more efficient, and now using a second thread actually delivers a performance gain. When using the Goto BLAS, you may need to set the environment variable `GOTO_NUM_THREADS` to 1.

```
!$OMP DO
do j=1,numblocks
do i=1,blocksize
idx = (j-1)*blocksize+i
if(idx.gt.nobs) then
a_block(i,:)  = 0.0d0
else
call build_a_line(a_block(1,i),nmax,u,long(idx),lat(idx),
rad(idx),latold,pnm,rearth)
latold = lat(idx)
end if
end do

call dsyrk('L','N',nmax+1,blocksize,1.0d0,a_block,u,1.0d0,
N2(1,threadnum),nmax+1)

...
```

Note that `a_block` has the dimensions $u \times blocksize$, and not $blocksize \times u$. This is due to Fortran's column-major array storage (arrays are stored column by column, not line by line as in C). `build_a_line` needs only one row of $\mathbf{A}$ at a time, which is achieved by making the rows the columns. This has to be kept in mind when building the product $\mathbf{A}^T\mathbf{A}$, which has to be done as $\mathbf{A}_j\mathbf{A}_j^T$. Other methods, such as explicitly passing all elements of one line (`a_block(i,:)`) or copying a line to the blocked $\mathbf{A}$ (`a_block(i,:) = a_row`) are much slower.

Another profiler run now shows that about 30% of the runtime are required for calling `build_a_line`, 25% for `dgemm_kernel` (the multiplication $\mathbf{A}^T\mathbf{A}$ in building the preconditioner), and 13% in `daxpy`. The linewise build of $\mathbf{A}$ is thus the logical next target for parallelisation. It has already been parallelised in the preconditioner build, and parallelising it inside the iteration block follows the same method, with individual vectors for each thread, and summation after the parallel block.

With these modification, the program is sped up by a factor of 1.8 when using the second core of a Pentium D, with $n_{max} = 100$ and 64,800 observations. The speedup should be even better when processing larger problems.

5.3.5 Parallelisation with MPI

The direct method was parallelised for distributed memory systems using ScaLAPACK. ScaLAPACK was the ideal method, as most of the workload was performed by the BLAS and LAPACK routines DSYRK and DPOSV.

As could be seen in the previous section, the conjugate gradient method behaves quite differently. Only small vectors are processed, with most of the runtime being consumed by the linewise setup of the design matrix **A**. Using ScaLAPACK would be of little benefit. It is better (and easier) to use pure MPI for distribution of the computations, as has been done in SHALE CG V0.6.

The MPI parallelisation adresses the same areas as the OpenMP parallelisation in the previous section: distribution of the linewise setup of **A** over the nodes. As with the direct method, the OpenMP parallelisation will be retained, as multithreaded programs usually run faster than singlethreaded programs. Note that the loop distribution shown here is not load-balanced. For the program to run efficiently, it has to be made sure that all processes can dedicate an equal amount of processing power to the problem.

For MPI programs, the header file mpif.h has to be included into the routines. As with BLACS, MPI first has to be initialised.

```
include 'mpif.h'
call mpi_init(ierr)
```

Two more function calls are necessary to get the total number of processes, and the process id (rank).

```
call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)
```

At the end of the program, the MPI communication has to be ended:

```
call mpi_finalize(ierr)
```

Parallelising the build of the preconditioner is actually quite easy. A process with rank $i+1$ is assigned every ith block. Within one process, the blocks are still distributed by OpenMP (figure 5.10).

```
!$OMP DO
do j=myrank+1,numblocks,nprocs
```

Each process then holds its part of $\mathbf{N}_{bd}$. All these parts have to be summed up to build the full $\mathbf{N}_{bd}$, and distributed to all processes. This can be done with one call to `mpi_allreduce`, which sums the elements of `N2` into `N` and distributes the results (figure 5.11):

```
call mpi_allreduce(N2,N,size_nbd,MPI_DOUBLE_PRECISION,MPI_SUM,
MPI_COMM_WORLD,ierr)
```

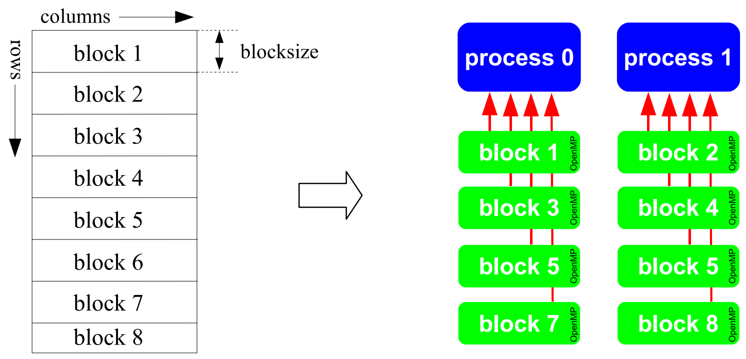

Figure 5.10: Blockwise distribution of **A** over processes

The blockwise inversion is very fast and done by each process. It would also be possible to give the full $\mathbf{N}_{bd}$ to only one process (using `mpi_reduce`), let that process do the inversion, then distribute the result $\mathbf{N}_{bd}^{-1}$. Distributing the blocks for inversion among the processes would be more complicated and would not lead to a significant runtime improvement. It is often beter do do small computations multiple times (once by each process) than to do it just once and distribute the results, as MPI communication is rather slow.

The loop over the lines of **A** in the initialisation step and inside the iterations is parallelised in the same way. The resulting program performs quite well on an Infiniband-equipped Opteron cluster. Doing an analysis up to degree 300 with 260,000 observations takes about 30 minutes on a Pentium D 830. Eight cluster nodes (each equipped with two Opteron 280 CPUs) need only two and a half minutes.

An interesting effect can be viewed, though, when comparing singlethreaded (one process per core, `OMP_NUM_THREADS` set to 1) and multithreaded (one process per node, `OMP_NUM_THREADS` set to 4) program runs. With eight nodes, preconditioner build is slightly faster during the multithreaded program run, while the actual solving is faster during the singlethreaded program run. This effect amplifies for higher node numbers, and with 30 nodes, the multithreaded program is faster, as the singlethreaded program needs more time for building the preconditioner than for solving the equation system. Summing up and distributing $\mathbf{N}_{bd}$ over all processes probably slows down the singlethreaded program, while the multithreaded program suffers from too small workloads for each thread.

This can be improved by changing the linewise processing of **A** to blockwise, as with the build of $\mathbf{N}_{bd}$ (figure 5.10). This has been done in SHALE CG V0.7. Both singlethreaded and multithreaded runs benefit from this. For 8 nodes, the multithreaded program is now as fast as the singlethreaded program, and it is faster and scales much better with more than eight nodes. The

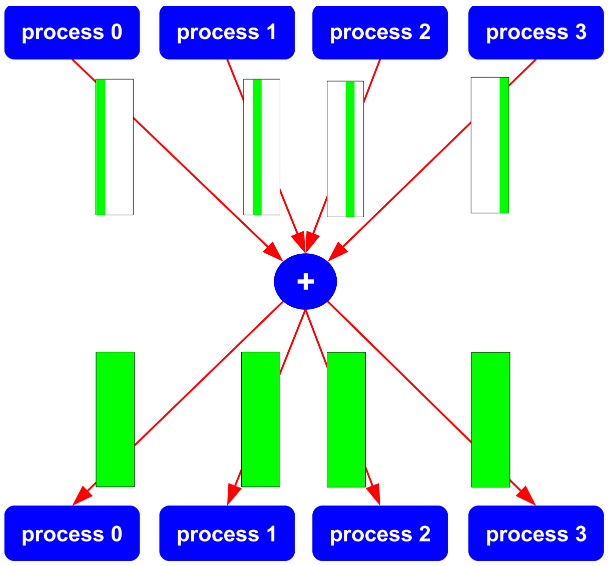

Figure 5.11: Summation and distribution of vector with `mpi_allreduce`

singlethreaded program is slower with 30 nodes than with 16 nodes. Figure 5.12 shows the time required for building the preconditioner, solving, and the total runtime, for 1 to 32 nodes. The problem size was $n_{max} = 300$ and $n = 259,200$ observations. The system used was a Linux cluster with dual-CPU Opteron 280 nodes (dual-core, 2.4 GHz) and Infiniband interconnect, 4 threads per node were used. The MPI library was Open MPI 1.1.2, with Goto BLAS 1.0.7 and the reference LAPACK providing the linear algebra routines. The solving scales almost linearly, while the preconditioner build benefits little from going from 16 to 32 CPUs. This also affects the total runtime.

5.4 Conclusions

The previous sections showed step-by-step, how two different programs can be parallelised using OpenMP, MPI, and ScaLAPACK. A number of conclusions can be drawn from this chapter:

- *If*-statements may or may not slow down the program, try and see. If it doesn't, keep the *if* if it avoids code-recycling.
- It is relatively simple to parallelise loops using OpenMP. Make sure all necessary variables are listed in the PRIVATE statement.

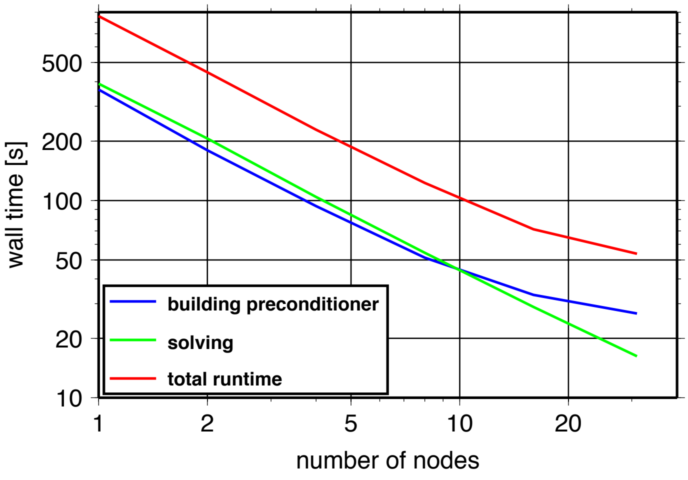

Figure 5.12: Runtimes of SHALE CG V0.7 with $n_{max} = 300$ and $n = 259,200$ observations

- Avoid simultaneous memory access to the same location by multiple threads. Give each thread its own variable, and combine them after the parallel region.
- Make sure each thread has a sufficient workload, otherwise starting/finishing threads will cost more time than the actual computation.
- Blockwise instead of linewise computations are a very good method for improving the efficiency of the program, while blocks can be made sufficiently small to avoid memory problems.
- ScaLAPACK is a good choice for parallelising programs that make extensive use of BLAS and LAPACK routines.
- When loops are to be distributed, it is usually better to use pure MPI.
- When using message passing, avoid communication as much as possible, especially with a slow interconnect (Ethernet).
- Multithreaded MPI programs will usually run faster than singlethreaded programs, as less explicit communication by message passing is required.

Bibliography

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D., 1999. LAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA.

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., Whaley, R. C., 1997. ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Ditmar, P., Klees., R., Kostenko., F., 2003. Fast and accurate computation of spherical harmonic coefficients from satellite gravity gradiometry data. Journal of Geodesy 76, 690-705.

Dongarra, J., Whaley, R. C., 1995. A user's guide to the BLACS v1.1, Computer Science Dept. Technical Report CS-95-281, University of Tennessee, Knoxville, TN.

Flynn, M.J., 1972. Some computer organisations and their effectiveness. IEEE Trans Computers 21, 948-960.

Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S., 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary.

Goto, K., van de Geijn, R., 2006. High-Performance Implementation of the Level-3 BLAS. ACM Transactions on Mathematical Software, submitted.

Gropp, W., Lusk, E., Doss, N., Skjellum, A., 1996. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22 (6), 789-828.

Gropp, W., Lusk, E., 1996. User's Guide for mpich, a Portable Implementation of MPI. Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, IL.

Lawson, C. L., Hanson, R. J., Kincaid, D., Krogh, F. T, 1979. Basic Linear Algebra Subprograms for FORTRAN usage. ACM Transactions on Mathematical Software 5, 308-323.

Liu, J., Wu, J., Panda, D. K., 2004. High Performance RDMA-Based MPI Implementation over InfiniBand. International Journal of Parallel Programming.

OpenMP Fortran Application Program Interface, Version 2.0, 2002. http://www.openmp.org.

Openshaw S., Turton I., 1999. High Performance Computing and the Art of Parallel Programming, An introduction for geographers, social scientists and engineers. Routledge, London. ISBN 0-415-15692

Schönauer, W., 2000. Architecture and use of shared and distributed memory parallel computers. Published by Willi Schönauer, Karlsruhe.

Whaley, R.C., Petitet, A., 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. Software: Practice and Experience 35 (2), 101-121.

Index

An Introduction to Parallel Programming

Many scientific computations require a considerable amount of computing time. This computing time can be reduced by distributing a problem over several processors. Multiprocessor computers used to be quite expensive, and not everybody had access to them. Since 2005, x86-compatible CPUs designed for desktop computers are available with two “cores”, which essentially makes them dualprocessor systems. More cores per CPU are to follow.

This cheap extra computing power has to be used efficiently, which equires parallel programming. Parallel programming methods that work on dual-core PCs also work on larger shared memory systems, and a program designed for a cluster or other type of distributed memory system will also perform on a dual-core (or multi-core) PC.

The goal of this tutorial is to give an introduction into all aspects of parallel programming that are necessary to write ones own parallel programs. To achieve this, it explains

- the various existing architectures of parallel computers,
- the software needed for parallel programming, and how to instal and configure it,
- how to analyse software and find the points where parallelisation might be helpful,
- how to write parallel programs for shared memory computers using OpenMP,
- how to write parallel programs for or distributed memory computers using MPI and ScaLA-PACK.

This tutorial mainly aims at writing parallel programs for solving linear equation systems.Hopefully it is also useful to give some help for parallelising programs for other applications.

Contents:

Published by VSSD
URL on this publication: http://www.vssd.nl/hlf/a019.htm

ISBN-10 90-71301-78-8
ISBN-13 978-90-71301-78-0

ISBN 90-71301-78-8
9 789071 301780