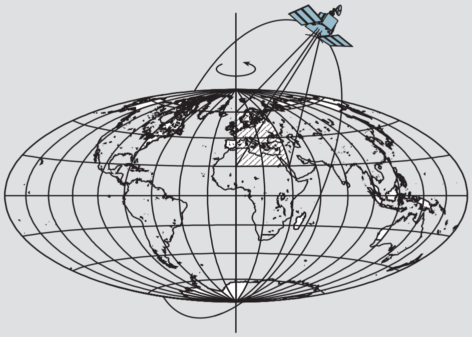
Implementation of Parallel Least-Squares Alorithms for Gravity Field Estimation

by

Jing Xie



Report No. 474

Geodetic and GeoInformation Science
Department of Civil and Environmental Engineering and Geodetic Science
The Ohio State University
Columbus, Ohio 43210-1275

March 2005

IMPLEMENTATION OF PARALLEL LEAST-SQUARES ALGORITHMS FOR GRAVITY FIELD ESTIMATION

By

Jing Xie

Report No.

Department of Civil and Environmental Engineering and Geodetic Science
The Ohio State University
Columbus, Ohio 43210

March 2005

ABSTRACT

NASA/GFZ's Gravity Recovery and Climate Experiment (GRACE) twin-satellite mission, launched in 2002 for a five-year nominal mission, has provided accurate scientific products which help scientists gain new insights on climate signals which manifest as temporal variations of the Earth's gravity field. This satellite mission also presents a significant computational challenge to analyze the large amount of data collected to solve a massive geophysical inverse problem every month. This paper focuses on applying parallel (primarily distributed) computing techniques capable of rigorously inverting monthly geopotential coefficients using GRACE data. The gravity solution is based on the energy conservation approach which established a linear relationship between the *in-situ* geopotential difference of two satellites and the position and velocity vectors using the high-low (GPS to GRACE) and the low-low (GRACE spacecrafts) satellite-to-satellite tracking data, and the accelerometer data from both GRACE satellites. Both the direct or rigorous inversion and the iterative (conjugate gradient) methods are studied. Our goal is to develop numerical algorithms and a portable distributed-computing code, which is potentially "scalable" (i.e., keeping constant efficiency with increased problem size and number of processors), capable of efficiently solving the GRACE problem and also applicable to other generalized large geophysical inverse problems.

Typical monthly GRACE gravity solutions require solving spherical harmonic coefficients complete to degree 120 (14,637 parameters) and other nuisance parameters. The accumulation of the 259,200 monthly low-low GRACE observations (with 0.1 Hz sampling rate) to normal equations matrix needs more than 55 trillion floating point operations (FLOPs) and ~1.7 GB central memory to store it. Its inversion adds ~1 trillion FLOPs. To circumvent this huge computational challenge, we use a 16 nodes SGI 750 cluster system with 32 733 MHz Itanium processors to test our algorithm. We choose the object-oriented Parallel Linear Algebra Package (PLAPACK) as the main tool and Message Passing Interface (MPI) as the underlying communication layer to build the parallel code. MPI parallel I/O technique is also implemented to increase the speed of transferring data between hard drive and memory. Furthermore, we optimize both the serial and parallel codes by carefully analyzing the cost of the numerical operations, fully exploiting the power of the Itanium architecture and utilizing highly optimized numerical libraries.

For direct inversion, we tested the implementations of the Normal equations Matrix Accumulation (NMA) method, that computes the design as well as normal equations matrix locally and accumulates them to global objects afterwards, and the Design Matrix Accumulation (DMA) approach, which forms small-size design matrices locally first and transfers them to global scale by matrix-matrix multiplication to obtain a global normal equations matrix. The creation of the normal equations matrix takes the majority of the entire wall clock time. Our preliminary results indicate that the NMA method is very fast but at present cannot be used to estimate extremely high degree and order coefficients due to the lack of central memory. The DMA method can solve for all geopotential coefficients complete to spherical harmonic degree 120 in roughly 30 minutes using 24 CPUs. The serial implementation of the direct inverse method takes about 7.5 hours for the same inversion problem using the same but only one processor.

In the realization of the conjugate gradient method on the distributed platform, the preconditioner is chosen as the block diagonal part of the normal equations matrix. The approximate computation of the variance-covariance matrix for the solution is also implemented. With significantly less arithmetic operations and memory usage, the conjugate gradient method only spends approximately 8 minutes (wall clock time) to solve for the gravity field coefficients up to degree 120 using 24 CPUs after 21 iterations, while the serial code runs roughly 3.5 hours to achieve the same results on a single processor.

Both the direct inversion method and the iterative method give good estimates of the unknown geopotential coefficients. In this sense, the iteration approach is better for the much shorter running time, but only an approximation of the estimated variance-covariance matrix is provided. Scalability of the direct and iterative method is also analyzed in this study. Numerical results show that the NMA method and the conjugate gradient method achieve good scalability in our simulation. While the DMA method is not as scalable as the other two for smaller problem sizes, its efficiency improves gradually with the increase of problem sizes and processor numbers. The developed codes are potentially transportable across different computer platforms and applicable to other generalized large geophysical inverse problems.

PREFACE

This report was prepared by Jing Xie, a graduate research associate in the Department of Civil and Environmental Engineering and Geodetic Science at the Ohio State University, under the supervision of Professor C. K. Shum. This research was partially supported by grants from NSF Earth Sciences program: EAR-0327633, NASA Office of Earth Science program: NNG04GF01G and NNG04GN19G.

This report was also submitted to the Graduate School of the Ohio State University as a thesis in partial fulfillment of the requirements for the Master of Science degree.

ACKNOWLEDGMENTS

First of all, I would like to thank Dr. C. K. Shum for giving me the opportunity and supporting me throughout my study and thesis writing, for all of his encouragement and advice, as well as acting as my supervisor in the thesis committee.

My appreciation also goes to Dr. Burkhard Schaffrin for reviewing this manuscript, for giving constructive and valuable suggestions, and for correcting many scientific and grammar errors on my thesis. In addition, I also would like to thank Dr. Shin-Chan Han for his comments and providing his serial code for my reference.

I also want to thank the Ohio Supercomputer Center, for providing the cluster machine used in this study under the project "Cluster Ohio" and offering instructive workshops about parallel processing.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION AND MOTIVATION

Accurate mapping of the Earth’s gravity field through space geodetic techniques is both a data processing (Input/Output) and computation intensive task. Several currently operating and planned dedicated satellite missions such as the CHAllenging Minisatellite Payload (CHAMP) (Reigber et al., 1999) with high-low satellite-to-satellite tracking (hl-SST), the Gravity Recovery And Climate Experiment (GRACE) (Tapley et al., 2004) with high-low and low-low satellite-to-satellite tracking (hl-SST and ll-SST), and the Gravity Field and Steady-State Ocean Circulation Explorer (GOCE) (Rebhan et al., 2000) with hl-SST and satellite gravity gradiometer (SGG) provide the promise for mapping the gravity field with unprecedented accuracy and resolution. Consequently, solving for the spherical harmonic coefficients of the geopotential model to a high degree and order requires the accumulation of a large amount of observations into non-sparse normal equations matrices, and the solution of such a large linear system requires enhanced computing power and efficient algorithms. For the current GRACE mission and the planned GOCE mission, one of the major challenges is the enormous computer processing power required to analyze the voluminous data already which is far beyond the capability of any type of single processor. For example, to solve for the spherical harmonic coefficients complete to degree 150 using GRACE measurements, approximately 23,000 unknown parameters need to be estimated. Meanwhile, huge amounts of observation data are continuously accumulated every day. Specifically, a total of approximately 518,400 ll-SST observations are collected over a period of 30 days with a sampling rate of 0.2Hz. GOCE is designed to recover gravity field coefficients complete to degree 300, which means that more than 90,000 parameterized unknowns are to be computed. So, besides dealing with such a huge volume of data, we also need to optimize numerical algorithms to solve for the unknown potential coefficients with high degree and order both rapidly and accurately. Short running time expedites debugging and testing of new algorithms. For the stringent task of processing GRACE data powerful supercomputers (e.g., Cray T3E) have been used (Gunter et al., 2001). However, traditional supercomputers are expensive and need dedicated support and maintenance. As an alternative, a commodity cluster with parallelized algorithms on many processor elements (PEs) is a fast and efficient option to solve this problem; e.g., the cluster can be built using contemporary processors (Intel Pentium 4 or Itanium, AMD Athlon or Opteron) and high speed interconnections (Myrinet) on a Linux platform with affordable cost and comparable computing performance. Meanwhile, in order to achieve best productivity, linear algebra packages, such as Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) need to be adapted to parallel platforms as well. Currently, Scalable Linear Algebra PACKage (ScaLAPACK) (Blackford et al., 1996) and the Parallel Linear Algebra Package (PLAPACK) (Van de Geijn, 1997) are among the most popular implementations.

The solution for the gravity field coefficients using the "brute force" method needs long waiting time on a serial implementation. Iterative methods can save considerable time and storage for the same problem. Klees et al. (2000) combine an iterative solution of the normal equations, using a Richardson-type iteration scheme, with the fast computation of the right-hand side of the normal equations in each iteration step by a suitable approximation of the design matrix to recover the GOCE gravity field. Kusche (2001) applied the multigrid solver to satellite data analysis using a space domain representation of disturbing gravity field and shows that the multigrid solver runs faster than the conjugate gradient solver with conventional pre-conditioners. On the other hand, a serial computing approach by Han (2004) implemented a more straightforward strategy for the observation equations based on the energy conservation principle (Jekeli, 1999). The advantage of this approach is that the *in-situ* potential difference observables have a simple linear relation to the unknown potential coefficients. A truncated Taylor series expansion of Legendre functions is used to avoid excessive computations when constructing the design matrix. No explicit appearance of the normal equations matrix saves limited memory for the computation and leaves space for higher degree and order gravity recovery. For the same reason, the conjugate gradient method (Schuh, 1996) is applied to solve the normal equations. A block-diagonal matrix which contains most of the power of the full normal equations matrix is constructed as a preconditioning matrix to increase the convergence rate. Also, this preconditioning matrix can be viewed as an approximation of the variance-covariance matrix. This algorithm has been demonstrated on the CRAY SV1 machine in a serial computation environment (Han, 2004).

In the parallel computing approach, Gunter (2000) applied a parallel least-squares algorithm with PLAPACK using simulated GRACE data. His work mainly concentrated on solving the over-determined least-squares problem and on comparing the QR factorization with the Cholesky decomposition. In his research, the design matrix is assumed to be available at the very beginning. Pail and Plank (2002) evaluate three numerical methods (preconditioned conjugate gradient method, semi-analytic approach, and distributed non-approximative adjustment) on the gravity recovery of GOCE gradiometer data and conclude that the three methods deliver nearly identical results with a preference for the non-approximate adjustment method.

In this study, we focus mainly on implementing a parallelization of the least-squares solver based on PLAPACK for the energy conservation approach (Jekeli, 1999) using simulated GRACE gravity data on a 16-node (32 PEs) Itanium cluster provided by the Ohio Supercomputing Center and located at the Laboratory for Space Geodesy and Remote Sensing, The Ohio State University. The parallel environment applied in this study includes Linux cluster, Message Passing Interface (MPI) (Snir et al., 1996), parallel Input/Output (I/O), and linear algebra package (PLAPACK). We choose PLAPACK over ScaLAPACK mainly because it can convert the formidable task of parallel coding to a relatively easy and manageable one using its object-oriented feature. And high level abstraction hides the detailed implementation of data distribution to PEs and their inter-communication through MPI.

Two methods are implemented in parallel mode:

- direct inversion method with no approximation ("brute force"),
- iterative method (conjugate gradient method).

The developed parallel method can be generalized to suit any gravity inversion problem or large linear system and is not limited to the current task. Parallel I/O is embedded in the process of generating the normal equations matrix which greatly reduces the I/O time by letting each processor work with only part of the whole data and then collect accumulated local contributions together to form the normal equations matrix. In a strict method, the normal equations matrix will be inverted with the parallel Cholesky solver using PLAPACK. This scheme is similar with the classical "Helmert blocking" technique in traditional Geodetic community (Wolf, 1978, Vaníček and Krakiwsky, 1986).

The outline of the thesis is as follows:

Chapter 2 overviews the basic model of GRACE gravity recovery and outlines the processing method with the requirements which guide the design of the parallel system. Least-Squares adjustment using Cholesky decomposition, conjugate gradient method and their respective benefits are described as well.

Chapter 3 gives an introduction and compares the hardware and software architecture of different parallel techniques. With a review of MPI and parallel I/O, PLAPACK is compared with ScaLAPACK on the internal mechanism of distributing data to nodes, inter-node communication and user-friendly features. In addition, an evaluation model for the parallel code is given at the end of this chapter.

Chapter 4 describes the detailed implementations of parallel computing for the direct and iterative methods, which include parallel I/O to deal with a large number of data files, different matrix accumulation methods due to hardware limitations (Normal equations Matrix Accumulation (NMA) and Design Matrix Accumulation (DMA)). A parallel Cholesky factorization approach for the normal equations matrix inversion and the parallel conjugate gradient method for the iterative approach are also presented.

Chapter 5 gives results for the parallel implementations of direct inverse and iterative methods. Simulated GRACE data based on the Earth's gravity field model EGM96 are generated for this study. The performance is evaluated based on different grid and node sizes. The accuracies of the estimated coefficients and their variance-covariance matrix are also taken into considerations. Serial implementations of the same problem are coded for comparison studies.

Chapter 6 summarizes the results and conclusions of this work. Future tasks of interest are discussed.

CHAPTER 2

A LEAST-SQUARES ALGORITHM AND ALTERNATIVE METHODS

2.1 The GRACE mission

Launched in March of 2002, The joint NASA/GFZ GRACE mission consists of two identical satellites flying about 220 kilometers apart in a near-circular orbit with a mean altitude of 500 km and mean inclination of 89 degree, suitable for the mapping of the temporal gravity field for climate change studies. The scientific data of the GRACE mission consist of Global Positioning System (GPS) high-low measurements primarily for orbit determination and atmospheric limb-sounding, 3-axes accelerometer data to measure the non-gravitational force, inter-satellite (low-low SST) K-band microwave tracking (range and range-rate) and altitude data. The low-low SST measurement with a precision of 0.1 μ m/sec (range-rate) represents an opportunity to achieve unprecedented accuracy for gravity mapping, especially for the time-variable gravity field. Accurate range and range-rate measurements between these two satellites could make the orbit determination more precise as well. The data are used to recover a global gravity field solution every 30 days in the designed mission span of 5 years. (GRACE Science Mission Requirement Document, 2000).

2.2 Estimation of gravity field parameters/coefficients

The Earth's gravity potential V is usually represented by spherical harmonic expansion as follows:

$$V = \frac{GM}{R}\left(1 + \sum_{n=2}^{\infty}\sum_{m=0}^{n}\left(\frac{R}{r}\right)^{n+1} \overline{P}_{nm}(\cos\theta)\left(\overline{C}_{nm}\cos m\lambda + \overline{S}_{nm}\sin m\lambda\right)\right) \qquad (2.1)$$

where GM is the gravitational constant times the mass of the Earth; R is the mean equatorial radius of the Earth; θ, λ, r are the spherical coordinates (geocentric co-latitude, longitude, and radius, in the Earth-Centered Earth-Fixed frame). $\overline{P}_{nm}$ denotes the normalized Legendre polynomial of degree n and order m; $\overline{C}_{nm}, \overline{S}_{nm}$ represent the normalized spherical harmonic coefficients (unknown parameters). The gravity recovery is based on a least-squares solution of the observation equations which link the unknown parameters (spherical harmonic coefficients) with the orbital observations (position and velocity vectors, range, range-rate, accelerometer observations, etc.)

The contemporary method of using the satellite measurements for gravity field modeling employs Newton's second law (Tapley, 1973). The nonlinear equation of motion is solved by numerical integration as an initial value problem. The best approximation of the Earth's gravity field is used as an initial model for the orbit determination and thus improved using the new observations with subsequent iterations. The relationship between the unknowns and the observables is non-linear thus making the construction of the design matrix complicated. The solution is sensitive to orbit dynamics and any change of the dynamic model will cause the re-processing of all the data due to non-linearity. In this method, the GPS high-low SST, KBR(K Band Range), accelerometer and other data are jointly used in both orbit determination and gravity recovery (Bettadpur, 2004).

To avoid such problems, the energy integral method (Visser et al., 2003) was applied for GOCE satellite. A similar energy conservation method (Jekeli, 1999; Han, 2004) employs the energy conservation principle to compute the *in-situ* potential differences of two GRACE satellites using KBR measurement, satellite orbits, accelerometer data, and altitude in the inertial (or non-rotating) coordinate system. The advantage of this method is the linear relationship between the potential differences and the unknown gravity coefficients. This will allow the change of any model without major re-processing of the data. Precise orbits are computed using the kinematic approach, the reduced-dynamic method, or the dynamic method can also be used. It is also found that the energy conservation model could easily incorporate observations of non-conservative accelerations (Han, 2004). This makes this model virtually suitable for different kinds of observations, e.g., observations from CHAMP, GRACE and GOCE.

2.3 Energy conservative principle

Here energy conservation formulas for the disturbing potential differences from GRACE are given without any proof. For detailed information, see Jekeli (1999) and Han (2004). The observation equation is as follows:

$$\mathrm{T}_{12} = \left|\dot{\boldsymbol{x}}_1^0\right| \delta\dot{\rho}_{12} + \mathrm{v}_1 + \mathrm{v}_2 + \mathrm{v}_3 + \mathrm{v}_4 + \delta \mathrm{VR}_{12} - \int \left(\boldsymbol{F}_2 \cdot \dot{\boldsymbol{x}}_2 - \boldsymbol{F}_1 \cdot \dot{\boldsymbol{x}}_1\right) dt - \delta \mathrm{V}_\mathrm{T} - \delta \mathrm{V}_0 \qquad (2.2)$$

where

$$\mathrm{v}_1 = \left(\dot{\boldsymbol{x}}_2^0 - \left|\dot{\boldsymbol{x}}_1^0\right| \boldsymbol{e}_{12}\right) \cdot \boldsymbol{\delta}\dot{\boldsymbol{x}}_{12},$$

$$\mathrm{v}_2 = \left(\boldsymbol{\delta}\dot{\boldsymbol{x}}_1 - \left|\dot{\boldsymbol{x}}_1^0\right| \boldsymbol{\delta e}_{12}\right) \cdot \boldsymbol{x}_{12}^0,$$

$$\mathrm{v}_3 = \boldsymbol{\delta}\dot{\boldsymbol{x}}_1 \cdot \boldsymbol{\delta}\dot{\boldsymbol{x}}_{12},$$

$$\mathrm{v}_4 = \frac{1}{2}\left|\boldsymbol{\delta}\dot{\boldsymbol{x}}_{12}\right|^2, \text{ and}$$

$$\delta \mathrm{VR}_{12} = \omega_e \left\{ \left(\mathrm{x}_1 \dot{\mathrm{x}}_2 - \mathrm{x}_2 \dot{\mathrm{x}}_1\right)\Big|_2 - \left(\mathrm{x}_1 \dot{\mathrm{x}}_2 - \mathrm{x}_2 \dot{\mathrm{x}}_1\right)\Big|_1 - \left(\mathrm{x}_1^0 \dot{\mathrm{x}}_2^0 - \mathrm{x}_2^0 \dot{\mathrm{x}}_1^0\right)\Big|_2 + \left(\mathrm{x}_1^0 \dot{\mathrm{x}}_2^0 - \mathrm{x}_2^0 \dot{\mathrm{x}}_1^0\right)\Big|_1 \right\}.$$

In the above equations, T_{12} is the disturbing potential difference between two satellites in the inertial frame, $\boldsymbol{x}$ is the position vector, $\dot{\boldsymbol{x}}$ is the velocity vector, and $\boldsymbol{F}$ is the net force vector including all non-conservative forces, measured by the accelerometer, such as drag, solar radiation pressure, etc. The superscript, 0, denotes a quantity based on the known reference field, and the symbol δ indicates an incremental quantity between the true field and the reference field. The sixth term $\delta \mathrm{VR}_{12}$ of the right hand side of equation (2.2) is the potential rotation difference between two satellites. The seventh term is the dissipating energy difference between the two satellites. The eighth term δV_T and the last term δV_0 are the incremental temporal variation of the gravitational potential and the incremental energy constant of the system, respectively (Han, 2004). In this formula, the *in-situ* disturbing potential difference is directly connected with the GRACE measurements of the position vector, velocity vector, relative position and velocity vectors, and range rate between the two satellites.

2.4 Direct inverse method

The linear relationship between the *in-situ* potential difference and the unknown spherical harmonic coefficients is described as:

$$T_{12}(r_{1,}\theta_1,\lambda_1,r_{2,}\theta_2,\lambda_2) = T_1(r_{1,}\theta_1,\lambda_1) - T_2(r_{2,}\theta_2,\lambda_2)$$

$$= \frac{GM}{R}\sum_{m=0}^{N_{max}}\sum_{n=m}^{N_{max}}\left\{\begin{array}{l}\left\{\left(\frac{R}{r_1}\right)^{n+1}\bar{P}_{nm}(\cos\theta_1)\cos m\lambda_1 - \left(\frac{R}{r_2}\right)^{n+1}\bar{P}_{nm}(\cos\theta_2)\cos m\lambda_2\right\}\bar{C}_{nm} + \\ \left\{\left(\frac{R}{r_1}\right)^{n+1}\bar{P}_{nm}(\cos\theta_1)\sin m\lambda_1 - \left(\frac{R}{r_2}\right)^{n+1}\bar{P}_{nm}(\cos\theta_2)\sin m\lambda_2\right\}\bar{S}_{nm}\end{array}\right\} \quad (2.3)$$

where GM is the gravitational constant times the mass of the Earth; R is the Earth's mean equatorial radius; (r_1,θ_1,λ_1) and (r_2,θ_2,λ_2) are the coordinates of the 1st and 2nd satellite, respectively; $\bar{P}_{nm}$ is the fully-normalized, associated Legendre function of degree n and order m; $\bar{C}_{nm}$ and $\bar{S}_{nm}$ are the unknown spherical harmonic coefficients of degree n and order m (Han, 2004). N_{max} is the maximum degree of the unknown coefficients.

According to the Gauss-Markov (GM) model, the linear observation equations can be expressed as:

$$\boldsymbol{y} = \boldsymbol{A\xi} + \boldsymbol{e},\ \ \boldsymbol{e} \sim (0, \sigma^2\boldsymbol{P}^{-1}) \quad (2.4)$$

$$\boldsymbol{\xi}^0 = \left[\bar{C}_{2,0}\bar{C}_{3,0}\ldots\bar{C}_{N\max,0}\right]^T$$

$$\boldsymbol{\xi}^1 = \left[\bar{C}_{2,1}\bar{C}_{3,1}\ldots\bar{C}_{N\max,1}\bar{S}_{2,1}\bar{S}_{3,1}\ldots\bar{S}_{N\max,1}\right]^T$$

....

$$\boldsymbol{\xi}^m = \left[\bar{C}_{m,m}\bar{C}_{m+1,m}\ldots\bar{C}_{N\max,m}\bar{S}_{m,m}\bar{S}_{m+1,m}\ldots\bar{S}_{N\max,m}\right]^T \text{ for } 2 < m \le N_{\max}.$$

Here $\boldsymbol{A}$ is the design matrix; $\boldsymbol{\xi}$ is the unknown parameter vector, ordered as $[\xi^0 \quad \xi^1 \quad \cdots \quad \xi^m]^T$ so that the number of the unknown parameters is $n_{coef} = (N_{max}+1)^2 - 4$; $\boldsymbol{y}$ is the observation vector, and $\boldsymbol{e}$ is the random error vector. Note that unknown parameters are arranged in an order-wise sequence, which will form a nearly block diagonal normal equations matrix. The matrix will be strictly block-diagonal if observations are equally weighted and regularly distributed along the parallel (longitude), but not necessary for latitude. We assume the weight matrix $\boldsymbol{P}$ to be the identity matrix, which means that all observations are uncorrelated or even independent. We form the local design matrices on each computing node and write the normal equations as:

$$\boldsymbol{N}\hat{\boldsymbol{\xi}} = \boldsymbol{c}, \quad \boldsymbol{N} = \boldsymbol{A}^T\boldsymbol{A}, \quad \boldsymbol{c} = \boldsymbol{A}^T\boldsymbol{y}. \quad (2.5)$$

The matrix $\boldsymbol{N}$ is usually both symmetric and positive-definite, which implies that the inverse exists for $\boldsymbol{N}$.

Directly inverting the full matrix $\boldsymbol{N}$ is a computationally demanding task. Since the normal equations matrix is symmetric and positive-definite, the Cholesky factorization can be chosen for the task. The normal matrix $\boldsymbol{N}$ is decomposed to a lower (or upper) triangular matrix $\boldsymbol{L}$ (resp. $\boldsymbol{L}^{\mathrm{T}}$), namely

$$\boldsymbol{N} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}. \qquad (2.6)$$

The equation can be solved successively by two triangular updates using the factorized matrix $\boldsymbol{L}$ as:

$$\boldsymbol{L}\boldsymbol{b} = \boldsymbol{c} \text{ and} \qquad (2.7)$$

$$\boldsymbol{L}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}, \qquad (2.8)$$

where, from now on, $\boldsymbol{x}$ replaces $\hat{\boldsymbol{\xi}}$.

2.5 Least-squares algorithm

The common procedure of converting space gravity measurements usually consists of several steps, namely

1. collecting data,
2. forming observation equations and combining physical and dynamic models,
3. constructing normal equations, and
4. solving the normal equations.

Usually steps (3) and (4) are the most computationally intensive tasks. In step (2), the design matrix of the linear equations is computed from the Legendre functions to maximum degree and order (e.g., 150) for different latitudes, which is very time-consuming. Forming the normal equations matrix in step (3) requires the computation of the dyadic products among all row vectors of the design matrix. This step is computationally intensive for observations with very large row number. Finally, to invert a large full matrix in step (4) is also computationally demanding.

The implementation of this method, however, is straightforward (GS 651, 2002). The normal equations matrix is formed by the direct sum of contributions from every single or group of observations (since we assume that all observations are independent). Let

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 & \boldsymbol{A}_2 & \cdots & \boldsymbol{A}_{\mathrm{n}} \end{bmatrix}^{\mathrm{T}} \qquad (2.9)$$

be the design matrix with n blocks $\boldsymbol{A}_{\mathrm{i}}^{\mathrm{T}}$ (i = 1,2,…,n); then:

$$\boldsymbol{N} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{A} = \sum_{\mathrm{i=1}}^{\mathrm{n}} \boldsymbol{A}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{A}_{\mathrm{i}} = \sum_{\mathrm{i=1}}^{\mathrm{n}} \boldsymbol{N}_{\mathrm{ii}} \qquad (2.10)$$

and

$$\boldsymbol{c} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{y} = \sum_{\mathrm{i=1}}^{\mathrm{n}} \boldsymbol{A}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{y} = \sum_{\mathrm{i=1}}^{\mathrm{n}} \boldsymbol{c}_{\mathrm{i}}. \qquad (2.11)$$

The task of accumulating $\boldsymbol{N}$ is very suitable for parallel processing, since every computing node could allocate part of the whole design matrix and compute the local contribution. The final normal equations matrix is obtained by summing up every local contribution.

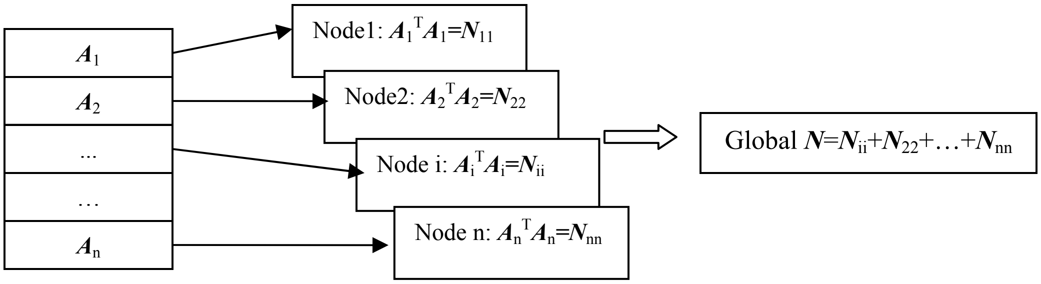

A_1
A_2
...
…
A_n
Node1: $A_1^T A_1 = N_{11}$
Node2: $A_2^T A_2 = N_{22}$
Node i: $A_i^T A_i = N_{ii}$
Node n: $A_n^T A_n = N_{nn}$
Global $N = N_{ii} + N_{22} + \ldots + N_{nn}$

Figure 2.1 Creation of the global normal equations matrix ***N***

The accumulation is made up of rank-one (BLAS Level 2) or rank-k (BLAS Level 3) updates, which usually can be finished at one time. BLAS 2 operations (Matrix-vector multiplications) are usually less efficient than the BLAS 3 routines (Matrix-matrix multiplications). The optimal performance could be achieved by carefully designing the distribution of the ***A*** matrix to nodes and utilizing the BLAS 3 rank-k update. Generally, $k \leq n_i$, where n_i is the number of rows from the design matrix distributed to node i. Meanwhile, symmetric BLAS routines could also reduce the number of operations. The details will be given in Chapter 4.

The floating point operation number for generating the lower part of the normal equations matrix from the design matrix is (2n-1)(m+1)m/2, the floating point operation number for the Cholesky decomposition is only $m^3/3$. Where n is the number of observations and m is the number of parameters. When the number of observations is large, the majority of total time is spent on the accumulation of the normal matrix. The total number of operations is roughly on the order of $O(nm^2+m^3/3)$.

For ill-conditioned systems, the QR decomposition or the Singular Value Decomposition (SVD) will provide a more accurate approach to any least-squares solution. It can be built based on our existing code in the near future.

2.6 Iterative methods

The direct solution of $\boldsymbol{Nx}=\boldsymbol{c}$ becomes impractical if ***N*** is very large, or if ***N*** is a sparse matrix. Iterative methods are alternatives which may replace the direct method, thereby generating a sequence of approximate solutions. Iterative methods usually only involve matrix-vector computations. The efficiency of the method is evaluated on how fast the convergence can be achieved.

The conjugate gradient (CG) method is one of the commonly used iterative methods which could achieve near-optimal performance with a carefully chosen pre-conditioner. It is most efficient if ***N*** is a square, symmetric, positive-definite (or positive-semidefinite) matrix. In this section, following the method of steepest descent, the conjugate gradient method will be briefly introduced. For details, please refer to Golub and van Loan (1996).

2.6.1 The method of steepest descent

First a quadratic function is defined as:

$$\mathrm{f}(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{x} - \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + \mathrm{d} \qquad (2.12)$$

where ***N*** is a matrix, ***c*** and ***x*** are vectors, and d is a scalar. f (***x***) is minimized by the solution of ***Nx***=***c***, if ***N*** is a symmetric and positive-definite matrix. This is simply because of the derivative being:

$$\mathrm{f}'(\boldsymbol{x}) = \boldsymbol{N}\boldsymbol{x} - \boldsymbol{c}. \qquad (2.13)$$

Let the gradient be equal to zero, then ***Nx***=*c* is obtained. The solution of ***Nx***=***c*** is a critical point of f (***x***); therefore, finding an **x** that minimizes f (***x***) amount to solving the normal equations ***Nx***=***c***. From a geometric point of view, that ***N*** is positive-definite, means that there exists a single value for ***x*** which lies at the bottom of a paraboloid to minimize f (***x***).

The method of steepest descent starts with an arbitrary value $\boldsymbol{x}_0$, and slides down to the bottom of the paraboloid. The search stops only when the computed solution is close enough to the real one.

When taking a step, the direction in which f decreases most quickly is chosen, which is the opposite direction of $\mathrm{f}'(\boldsymbol{x}) = \boldsymbol{N}\boldsymbol{x} - \boldsymbol{c}$. For the *i*th step, let the error vector be defined as:

$$\boldsymbol{e}_{\mathrm{i}} = \boldsymbol{x}_{\mathrm{i}} - \boldsymbol{x}. \qquad (2.14)$$

It indicates how far we are away from the solution. The discrepancy vector may be defined as:

$$\boldsymbol{r}_{\mathrm{i}} = \boldsymbol{c} - \boldsymbol{N}\boldsymbol{x}_{\mathrm{i}} = -\boldsymbol{N}\boldsymbol{e}_{\mathrm{i}} = -\mathrm{f}'(\boldsymbol{x}) \qquad (2.15)$$

The discrepancy vector could be thought as being the error vector in ***x*** transformed by ***N*** to the range space of $\boldsymbol{A}^{\mathrm{T}}$, and can be treated as the direction of steepest descent. Therefore, the sequence of $\boldsymbol{x}_{\mathrm{i}}$ can be obtained as:

$$\boldsymbol{x}_{\mathrm{i+1}} = \boldsymbol{x}_{\mathrm{i}} + \alpha_{\mathrm{i}}\boldsymbol{r}_{\mathrm{i}} \qquad (2.16)$$

where α_i should minimize f along a line; i.e., $\left.\frac{\mathrm{df}(\boldsymbol{x})}{\mathrm{d}\alpha_{\mathrm{i}}}\right|_{\boldsymbol{x}_{\mathrm{i+1}}} = 0$: (2.17)

$$\left.\frac{\mathrm{df}(\boldsymbol{x})}{\mathrm{d}\alpha_{\mathrm{i}}}\right|_{\boldsymbol{x}_{\mathrm{i+1}}} = \mathrm{f}'(\boldsymbol{x}_{\mathrm{i+1}})^{\mathrm{T}}\left.\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\alpha_{\mathrm{i}}}\right|_{\boldsymbol{x}_{\mathrm{i+1}}} = \mathrm{f}'(\boldsymbol{x}_{\mathrm{i+1}})^{\mathrm{T}}\boldsymbol{r}_{\mathrm{i}} = -\boldsymbol{r}_{\mathrm{i+1}}^{\mathrm{T}}\boldsymbol{r}_{\mathrm{i}}. \qquad (2.18)$$

The above expression is set equal to zero, which tells us that α is chosen so that $\boldsymbol{r}_{\mathrm{i}}$ and $\mathrm{f}'(\boldsymbol{x}_{\mathrm{i+1}})$ are orthogonal. Thus:

$$\boldsymbol{r}_{\mathrm{i+1}} = \boldsymbol{c} - \boldsymbol{N}\boldsymbol{x}_{\mathrm{i+1}} = \left(\boldsymbol{c} - \boldsymbol{N}\left(\boldsymbol{x}_{\mathrm{i}} + \alpha_{\mathrm{i}}\boldsymbol{r}_{\mathrm{i}}\right)\right) = (\boldsymbol{c} - \boldsymbol{N}\boldsymbol{x}_{\mathrm{i}}) - \alpha_{\mathrm{i}}\boldsymbol{N}\boldsymbol{r}_{\mathrm{i}} = \boldsymbol{r}_{\mathrm{i}} - \alpha_{\mathrm{i}}\boldsymbol{N}\boldsymbol{r}_{\mathrm{i}}, \qquad (2.19)$$

$$\boldsymbol{r}_{\mathrm{i+1}}^{\mathrm{T}}\boldsymbol{r}_{\mathrm{i}} = \boldsymbol{r}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{r}_{\mathrm{i}} - \alpha_{\mathrm{i}}\boldsymbol{r}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{r}_{\mathrm{i}} = 0, \qquad (2.20)$$

$$\alpha_{\mathrm{i}} = \frac{\boldsymbol{r}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{r}_{\mathrm{i}}}{\boldsymbol{r}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{r}_{\mathrm{i}}}. \qquad (2.21)$$

Summarizing, the method of steepest descent is as follows:

$$\boldsymbol{r}_{\mathrm{i}} = \boldsymbol{c} - \boldsymbol{N}\boldsymbol{x}_{\mathrm{i}}, \tag{2.22}$$

$$\alpha_{\mathrm{i}} = \frac{\boldsymbol{r}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{r}_{\mathrm{i}}}{\boldsymbol{r}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{r}_{\mathrm{i}}}, \tag{2.23}$$

$$\boldsymbol{x}_{\mathrm{i+1}} = \boldsymbol{x}_{\mathrm{i}} + \alpha_{\mathrm{i}}\boldsymbol{r}_{\mathrm{i}}. \tag{2.24}$$

The above algorithm requires two matrix-vector multiplications per iteration; one multiplication can be replaced by the following relationship after computing $\boldsymbol{r}_0$:

$$\boldsymbol{r}_{\mathrm{i+1}} = \boldsymbol{r}_{\mathrm{i}} - \alpha_{\mathrm{i}}\boldsymbol{N}\boldsymbol{r}_{\mathrm{i}} \tag{2.25}$$

The disadvantage of using the above relationship is that there is no feedback from $\boldsymbol{x}_{\mathrm{i}}$; round-off errors may accumulate to cause $\boldsymbol{x}_{\mathrm{i}}$ to converge to some point near $\boldsymbol{x}$.

The convergence of steepest descent is prohibitively slow if the condition number of $\boldsymbol{N}$ is large; i.e., if the eigenvalue ratio $\lambda_1(\boldsymbol{N})/\lambda_{\mathrm{n}}(\boldsymbol{N})$ is large (Golub and van Loan, 1996). Geometrically, $\mathrm{f}(\boldsymbol{x})$ forms an elongated hyperellipsoid; the lowest point lies in a relatively flat trough with steep side-walls, with the steepest descent bouncing back and forth between the sides of the trough while making little progress down the valley.

2.6.2 The method of conjugate gradient

Sometimes, the method of steepest descent will repeat the directions of earlier steps. If we can choose a set of orthogonal directions ($\boldsymbol{d}_0, \boldsymbol{d}_1, \cdots, \boldsymbol{d}_{\mathrm{n}}$) (with $\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{d}_{\mathrm{j}} = 0$ for i≠j) which never repeat themselves, the iteration will be finished after a certain number of steps. In general, for each step:

$$\boldsymbol{x}_{\mathrm{i+1}} = \boldsymbol{x}_{\mathrm{i}} + \alpha_{\mathrm{i}}\boldsymbol{d}_{\mathrm{i}}, \text{ or: } \boldsymbol{e}_{\mathrm{i+1}} = \boldsymbol{e}_{\mathrm{i}} + \alpha_{\mathrm{i}}\boldsymbol{d}_{\mathrm{i}}. \tag{2.26}$$

To use the above expression, we need to find the step size α_{i} and the search direction $\boldsymbol{d}_{\mathrm{i}}$. Using the idea that

$$\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{e}_{\mathrm{i+1}} = 0. \tag{2.27}$$

We obtain, formally, for the step size

$$\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\left(\boldsymbol{e}_{\mathrm{i}} + \alpha_{\mathrm{i}}\boldsymbol{d}_{\mathrm{i}}\right) = 0 \tag{2.28}$$

$$\alpha_{\mathrm{i}} = -\frac{\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{e}_{\mathrm{i}}}{\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{d}_{\mathrm{i}}} = \frac{\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}^{-1}\boldsymbol{r}_{\mathrm{i}}}{\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{d}_{\mathrm{i}}}. \tag{2.29}$$

Unfortunately, we cannot easily use the above formula without knowing $\boldsymbol{N}^{-1}$; if we knew it, the problem would already be solved.

To overcome this difficulty, we set the step size so that the search direction becomes $\boldsymbol{N}$-orthogonal instead of just orthogonal:

$$\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{d}_{\mathrm{j}} = 0 \text{ for } \mathrm{i} \neq \mathrm{j} \tag{2.30}$$

Similar to the procedure (2.18) in the steepest descent method, we obtain from (2.15) and (2.26):

$$\left.\frac{\mathrm{df}(\boldsymbol{x})}{\mathrm{d}\alpha_{\mathrm{i}}}\right|_{\boldsymbol{x}_{\mathrm{i}+1}} = \mathrm{f}'(\boldsymbol{x}_{\mathrm{i}+1})^{\mathrm{T}}\left.\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\alpha_{\mathrm{i}}}\right|_{\boldsymbol{x}_{\mathrm{i}+1}}$$

$$= \mathrm{f}'(\boldsymbol{x}_{\mathrm{i}+1})^{\mathrm{T}}\boldsymbol{d}_{\mathrm{i}} \qquad (2.31)$$

$$= -\boldsymbol{r}_{\mathrm{i}+1}^{\mathrm{T}}\boldsymbol{d}_{\mathrm{i}}$$

$$= \boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{e}_{\mathrm{i}+1} = 0.$$

Then, following the same steps as in (2.27-2.29), it is easy to arrive at:

$$\alpha_{\mathrm{i}} = -\frac{\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{e}_{\mathrm{i}}}{\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{d}_{\mathrm{i}}} = \frac{\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{r}_{\mathrm{i}}}{\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{d}_{\mathrm{i}}}. \qquad (2.32)$$

Next, we need to define a suitable set of ***N***-orthogonal search directions. We recall that in the method of steepest descent, discrepancies are chosen for the search directions. Here, we will start from this set of discrepancies ($\boldsymbol{r}_0, \boldsymbol{r}_1, \boldsymbol{r}_2, \cdots \boldsymbol{r}_{\mathrm{n}-1}$) to construct the search directions $\boldsymbol{d}_{\mathrm{i}}$ by the Gram-Schmidt process:

$$\boldsymbol{d}_{\mathrm{i}} = \boldsymbol{r}_{\mathrm{i}} + \sum_{\mathrm{k}=0}^{\mathrm{i}-1}\beta_{\mathrm{ik}}\boldsymbol{d}_{\mathrm{k}}. \qquad (2.33)$$

Here β_{ik} is defined for $\mathrm{i} > \mathrm{k}$, and can be derived using (2.30) by transposing and multiplying $\boldsymbol{N}\boldsymbol{d}_{\mathrm{j}}$ for $\mathrm{i} > \mathrm{j}$ on both sides:

$$\beta_{\mathrm{ij}} = -\frac{\boldsymbol{r}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{d}_{\mathrm{j}}}{\boldsymbol{d}_{\mathrm{j}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{d}_{\mathrm{j}}}. \qquad (2.34)$$

The disadvantage of this method is that all the old search vectors must be kept to construct the new one. However, the dependency of the above formula can be reduced to one step by applying the properties to be proved below.

From (2.26), we can obtain:

$$\boldsymbol{e}_{\mathrm{i}} = \boldsymbol{e}_0 + \sum_{\mathrm{j}=0}^{\mathrm{i}-1}\alpha_{\mathrm{j}}\boldsymbol{d}_{\mathrm{j}}. \qquad (2.35)$$

If after n steps, the solution converges to $\boldsymbol{x}$ (cutting down on the error in every step), this would mean $\boldsymbol{e}_{\mathrm{n}} = 0$. Substituting this into (2.35) by setting i=n, we get:

$$\boldsymbol{e}_0 = -\sum_{\mathrm{j}=0}^{\mathrm{n}-1}\alpha_{\mathrm{j}}\boldsymbol{d}_{\mathrm{j}}. \qquad (2.36)$$

Putting (2.36) into (2.35), we obtain:

$$\boldsymbol{e}_{\mathrm{i}} = -\sum_{\mathrm{j}=\mathrm{i}}^{\mathrm{n}-1}\alpha_{\mathrm{j}}\boldsymbol{d}_{\mathrm{j}}. \qquad (2.37)$$

Pre-multiplying the above equation by $\boldsymbol{d}_{\mathrm{k}}^{\mathrm{T}}\boldsymbol{N}$(k<i) results in:

$$\boldsymbol{d}_{\mathrm{k}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{e}_{\mathrm{i}} = -\sum_{\mathrm{j}=\mathrm{i}}^{\mathrm{n}-1}\alpha_{\mathrm{j}}\boldsymbol{d}_{\mathrm{k}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{d}_{\mathrm{j}} = 0 \qquad (2.38)$$

$$\boldsymbol{d}_{\mathrm{k}}^{\mathrm{T}}\boldsymbol{r}_{\mathrm{i}} = 0 \ \text{ for } \ \mathrm{k} < \mathrm{i}. \qquad (2.39)$$

Formula (2.39) tells us that *each discrepancy vector* $\boldsymbol{r}_j$ *ought to be orthogonal to all the old search directions* $\boldsymbol{d}_i$. We can obtain another important property by pre-multiplying the relation (2.33) by $\boldsymbol{r}_j$:

$$\boldsymbol{d}_i^T\boldsymbol{r}_j = \boldsymbol{r}_i^T\boldsymbol{r}_j + \sum_{k=0}^{i-1}\beta_{ik}\boldsymbol{d}_k^T\boldsymbol{r}_j = \boldsymbol{r}_i^T\boldsymbol{r}_j \text{ for } i \le j \tag{2.40}$$

using (2.38), and thus:

$$\begin{cases} 0 = \boldsymbol{r}_i^T\boldsymbol{r}_j, \ i < j; \\ \boldsymbol{d}_i^T\boldsymbol{r}_i = \boldsymbol{r}_i^T\boldsymbol{r}_i, \ i = j; \end{cases} \tag{2.41}$$

This means that *each discrepancy vector is orthogonal to all previous discrepancy vectors*.

Multiplying the second equation of (2.26) by –$\boldsymbol{N}$, we get:

$$\boldsymbol{r}_{j+1} = \boldsymbol{r}_j - \alpha_j\boldsymbol{N}\boldsymbol{d}_j. \tag{2.42}$$

Multiplying $\boldsymbol{r}_i^T$ on both sides and rearranging the terms leads to:

$$\boldsymbol{r}_i^T\boldsymbol{N}\boldsymbol{d}_j = \frac{1}{\alpha_j}\left(\boldsymbol{r}_i^T\boldsymbol{r}_j - \boldsymbol{r}_i^T\boldsymbol{r}_{j+1}\right), \tag{2.43}$$

and with (2.34), resp. (2.42) to:

$$\beta_{ij} = -\frac{\boldsymbol{r}_i^T\boldsymbol{N}\boldsymbol{d}_j}{\boldsymbol{d}_j^T\boldsymbol{N}\boldsymbol{d}_j} = -\frac{\left(\boldsymbol{r}_i^T\boldsymbol{r}_j - \boldsymbol{r}_i^T\boldsymbol{r}_{j+1}\right)}{\alpha_j\boldsymbol{d}_j^T\boldsymbol{N}\boldsymbol{d}_j}.$$

Considering (2.40) and (2.41), this means:

$$\beta_{ij} = \begin{cases} \dfrac{\left(\boldsymbol{r}_i^T\boldsymbol{r}_{j+1} - \boldsymbol{r}_i^T\boldsymbol{r}_j\right)}{\boldsymbol{d}_j^T\boldsymbol{r}_j - \boldsymbol{d}_j^T\boldsymbol{r}_{j+1}} = -\dfrac{\left(\boldsymbol{r}_i^T\boldsymbol{r}_i\right)}{\alpha_{i-1}\boldsymbol{d}_{i-1}^T\boldsymbol{N}\boldsymbol{d}_{i-1}}, & i = j+1; \\ 0, & i > j+1; \end{cases} \tag{2.44}$$

and equation (2.33) becomes:

$$\boldsymbol{d}_i = \boldsymbol{r}_i + \sum_{k=0}^{i-1}\beta_{ik}\boldsymbol{d}_k = \boldsymbol{r}_i + \beta_{i,i-1}\boldsymbol{d}_{i-1} \tag{2.45}$$

with

$$\beta_{i,i-1} = \beta_i = \frac{\boldsymbol{r}_i^T\boldsymbol{r}_i}{\boldsymbol{d}_{i-1}^T\boldsymbol{r}_{i-1}} = \frac{\boldsymbol{r}_i^T\boldsymbol{r}_i}{\boldsymbol{r}_{i-1}^T\boldsymbol{r}_{i-1}}, \tag{2.46}$$

β_i being an abbreviation of $\beta_{i,i-1}$. Summarizing the above, the method of conjugate gradients is given by:

$$\boldsymbol{d}_0 = \boldsymbol{r}_0 = \boldsymbol{c} - \boldsymbol{N}\boldsymbol{x}_0 \ , \tag{2.47}$$

$$\alpha_i = -\frac{\boldsymbol{d}_i^T\boldsymbol{r}_i}{\boldsymbol{d}_i^T\boldsymbol{N}\boldsymbol{d}_i} = \frac{\boldsymbol{r}_i^T\boldsymbol{r}_i}{\boldsymbol{d}_i^T\boldsymbol{N}\boldsymbol{d}_i} \ , \tag{2.32}$$

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i\boldsymbol{d}_i, \tag{2.26}$$

$$\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i\boldsymbol{N}\boldsymbol{d}_i \ , \tag{2.42}$$

$$\beta_{i+1} = \frac{\boldsymbol{r}_{i+1}^T\boldsymbol{r}_{i+1}}{\boldsymbol{r}_i^T\boldsymbol{r}_i}, \tag{2.46}$$

$$\boldsymbol{d}_{\mathrm{i}+1} = \boldsymbol{r}_{\mathrm{i}+1} + \beta_{\mathrm{i}+1}\boldsymbol{d}_{\mathrm{i}}\,. \tag{2.45}$$

Generally, the convergence of the method of conjugate gradients is faster than steepest descent. If there were no round-off errors for computers (i.e., the floating point number is represented by an infinite number of digits), then the number of iterations for an exact solution would be at most, the number of distinct eigenvalues of the normal equations matrix. Duplicate eigenvalues would make this method even faster (Van der Sluis and Van der Vorst, 1986).

Practically, orthogonality among the discrepancy vectors may not hold after several iterations due to computer rounding errors. Thus, only a finite number of steps can be guaranteed. Usually, we will treat this method as a common iteration method; the termination will be jointly determined by a maximum iteration number and the norm of the discrepancy vector.

2.6.3 Preconditioning

Preconditioning is a technique to improve the condition number of a matrix within a certain task. If ***M*** is a symmetric positive-definite matrix that approximates ***N***, but is easy to invert, then:

$$\boldsymbol{M}^{-1}\boldsymbol{N}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{c} \tag{2.48}$$

should have the same solutions as ***Nx*=*c***. However, if the condition number $\kappa(\boldsymbol{M}^{-1}\boldsymbol{N}) << \kappa(\boldsymbol{N})$, or the eigenvalues of $\boldsymbol{M}^{-1}\boldsymbol{N}$ are more clustered than those of ***N***, (2.48) can be solved faster than the original problem. ***M*** is called the "preconditioning matrix".

A preconditioned conjugate gradient method can be easily derived and is listed below:

$$\boldsymbol{r}_0 = \boldsymbol{c} - \boldsymbol{N}\boldsymbol{x}_0 \ , \tag{2.49}$$

$$\boldsymbol{d}_0 = \boldsymbol{M}^{-1}\boldsymbol{r}_0\,, \tag{2.50}$$

$$\alpha_{\mathrm{i}} = \frac{\boldsymbol{r}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{M}^{-1}\boldsymbol{r}_{\mathrm{i}}}{\boldsymbol{d}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{N}\boldsymbol{d}_{\mathrm{i}}} \ , \tag{2.51}$$

$$\boldsymbol{x}_{\mathrm{i}+1} = \boldsymbol{x}_{\mathrm{i}} + \alpha_{\mathrm{i}}\boldsymbol{d}_{\mathrm{i}}\,, \tag{2.52}$$

$$\boldsymbol{r}_{\mathrm{i}+1} = \boldsymbol{r}_{\mathrm{i}} - \alpha_{\mathrm{i}}\boldsymbol{N}\boldsymbol{d}_{\mathrm{i}} \ , \tag{2.53}$$

$$\beta_{\mathrm{i}+1} = \frac{\boldsymbol{r}_{\mathrm{i}+1}^{\mathrm{T}}\boldsymbol{M}^{-1}\boldsymbol{r}_{\mathrm{i}+1}}{\boldsymbol{r}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{M}^{-1}\boldsymbol{r}_{\mathrm{i}}}\,, \tag{2.54}$$

$$\boldsymbol{d}_{\mathrm{i}+1} = \boldsymbol{M}^{-1}\boldsymbol{r}_{\mathrm{i}+1} + \beta_{\mathrm{i}+1}\boldsymbol{d}_{\mathrm{i}}\,. \tag{2.55}$$

The effectiveness of the preconditioning matrix is determined by the condition number of the new matrix $\boldsymbol{M}^{-1}\boldsymbol{N}$ and, sometimes, the clustering of the eigenvalues. Geometrically, finding a preconditioning matrix is equal to stretching the elongated ellipsoidal quadratic form until it appears more spherical, and the eigenvalues are close to each other. The ideal preconditioning matrix would be ***N*** itself, and the new matrix becomes the identity matrix; its quadratic form is perfectly spherical, and its eigenvalues are all the same. However, if we can invert the ***N*** matrix so easily, any iteration method would be useless. Therefore, to find a suitable preconditioning matrix that can balance the faster convergence of the iteration and, at the same time, keep the cost of computing $\boldsymbol{M}^{-1}\boldsymbol{r}$ low enough, is a very tough task.

The simplest preconditioning matrix is a matrix that only contains the diagonal part of ***N***. It costs almost nothing to invert, but its performance is also mediocre. There exist many other preconditioning matrices; some are simple while others are sophisticated. Depending on the application and the sophistication, different preconditioning matrices can be adopted.

For the normal equations (2.5) with

$\boldsymbol{N} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$

the computation of ***N*** is the most time-consuming part of the computation. In the conjugate gradient method, $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ is never explicitly computed. Instead, the product of $\boldsymbol{Nd}$ is obtained by first computing $\boldsymbol{Ad}$, then $\boldsymbol{A}^{\mathrm{T}}(\boldsymbol{Ad})$. This converts one matrix-matrix multiplication and one matrix-vector multiplication into two matrix-vector multiplications.

2.7 Comparison

The advantage of the direct inverse method is that there is no approximation. Secondly this method is straight-forward which makes the coding simple, and there is no need to do preprocessing or preconditioning. Meanwhile, this method has some drawbacks as well. Since we need to get the inverse of the normal equations matrix ***N***, we have to generate the ***N*** matrix first. The procedure of accumulating the ***N*** matrix is the most time-consuming part of the whole work. At the same time, with the increasing number of unknown coefficients, a huge memory is required to store the normal equations matrix ***N***. In PLAPACK there is no special routine for a symmetric matrix, so a full matrix has to be stored in the memory. For example, for the maximum degree of 120, a memory space of 1.7GB is required for the normal equations matrix ***N***. For the Linux cluster used in this study, only 4 GB memory shared by 2 processors on one computing node is available which limits this method for recovering higher degree geopotential coefficients. In addition, huge amounts of operations are needed to fulfill the construction and inversion of the normal equations matrix ***N***. For $N_{max} = 120$, about $O(10^{14})$ floating-point operations are needed (Han, 2004). But after optimizing and applying efficient BLAS-3 subroutines, this problem can be partially solved.

As an alternative to the traditional direct inverse method, the conjugate gradient method has many valuable features. First of all, the normal equations matrix ***N*** is not explicitly used in the approach. By treating a set (row or block) of data by as many operations as possible, a vector instead of a matrix is obtained as the result; that dramatically reduces the operations needed and the memory space required, and hence the clock time occupied. A preconditioning matrix is added to increase the convergence rate of this method. Here the preconditioning matrix is chosen as ***M***, where ***M*** is the block diagonal part of the normal equations matrix ***N***. Since ***M*** is a sparse matrix, $\boldsymbol{M}^{-1}$ is easy to compute and as block-diagonal matrix again, only needs a small memory to store. This feature of $\boldsymbol{M}^{-1}$ also makes the multiplication by $\boldsymbol{M}^{-1}$ not time-consuming at all. So the extra work of creating the preconditioning matrix is trivial. On the other hand, the small memory occupation comes at a price. In order to avoid the appearance of the normal equations matrix ***N***, many overheads exit in the iteration loop which results in a vast volume of repeated computations.

CHAPTER 3

PARALLEL TECHNIQUES

3.1 Overview

More than two decades ago parallel processing was introduced into the supercomputing community, either to reduce the running time needed to solve a problem or to increase the size of problems that can be solved. Its popularity in the field of scientific computations mainly lies in the fact that it can solve data-dense problems and computation-dense problems faster than by applying serial method. At the same time concentrated computing resources like processors, memories, etc make it possible to deal with large-scale problems that are already beyond the capability of sequential programming. In a parallel environment, the multiple processors may all reside on the same computer, or they may be spread across separate computers. When they are spread across separate computers, each computer is referred to as a node.

Traditionally, programs are written for serial computers which we all have become familiar with. In this situation, only one instruction is executed at a time using one processor. But when turning to the parallel approach, many identical or different instructions are running on multiple processors at the same time.

General procedures of parallel processing include: 1) dividing a large task into several smaller tasks that can be worked on simultaneously; 2) allocating these small tasks onto several processors as desired; 3) running tasks on some parallel programming environment; 4) thorough communication and coordination among these processors to get the result. We can see clearly that parallelized computing speeds up the execution of a program. Theoretically a program performed on n processors may get n times faster than that performed on a single processor. But for many occasions, that is not always true. Some added costs associated with parallelism need to be considered. Cooperation and communication between different processors is one source for this extra price. When the problem size is too small, the performance of parallel programming is even worse than the sequential one due to this overhead (Nagel, 1999). Synchronization among multiple processors performing identical or different instructions is another kind of overhead of parallelism, needed to be managed. Besides these, administering a parallel computing environment is more complicated than administering a serial environment. Then there must be a performance gain large enough to justify the overhead of parallelism and show the full potential of benefits of parallelism.

Various parallel architectures of supercomputers have been invented over the years. There are several different methods to classify the parallel computers and no single taxonomy can be used to fit all of them. Machine architecture and memory architecture are the two most commonly used classifications.

3.2 Machine architecture

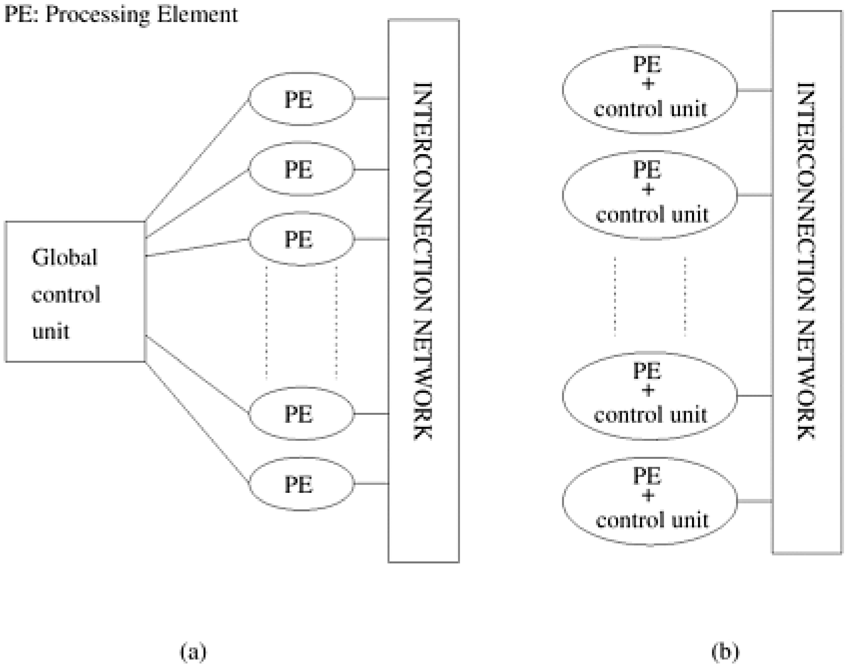


Figure 3.1: (a) A typical SIMD architecture and (b) a typical MIMD architecture.

According to the relationship between the instruction and the data, Flynn (1974) gives four basic classes as follows:

- SISD: Single Instruction Stream Single Data Stream

A SISD computer is not a parallel computer. It is the conventional sequential computer that can only execute one set of data by only one processor at one time.

- SIMD: Single Instruction Stream Multiple Data Stream

A SIMD computer is characterized by the fact that all processors perform the same instruction and each processor works on its own data piece at the same time (Figure 3.1 (a)) (Grama et al., 2003). This feature naturally meets the requirement of the vector computation, which consists in executing the same operation on every vector element. Because of the inherent synchronization in this model, interconnection between processors is not hard to administrate. Vector machines certainly belong to this class.

- MISD: Multiple Instruction Stream Single Data Stream

A MISD computer is not very practical at the present time.

- MIMD: Multiple Instruction Stream Multiple Data Stream

A MIMD computer refers to a model with parallel execution in which each processor can operate a sequential program independently of the others (Figure 3.1(b)) (Grama et al., 2003). Processors can execute multiple job streams simultaneously. Meanwhile each processor can perform any operation regardless of what other processors do. These characteristics greatly fit the concept of parallelism: Dividing a large task into several smaller jobs and getting them operated on by multiple processors simultaneously. But synchronization and loading balance should be considered carefully.

Beside these four basic categories, a subset of MSMD is introduced as:

- SPMD: Single Program Multiple Data Stream

The difference of SPMD and MIMD is that all processors operate with the same instruction on different data sets. It is the most popular model in the parallel computing field.

3.3 Memory architecture

According to the method that processors accumulate with other processors, there are three major classes:

- Shared memory system

The shared memory systems employ a limited number of powerful independent processors, each with access to the common memory. Only one processor can access the shared memory location at a time. Synchronization is achieved by controlling tasks' reading from and writing to the shared memory. The speed of data sharing is the speed of memory access. So it is very fast. But sometimes the communication bandwidth may cause a bottleneck. Hence the number of processors varies from two to tens. Figure 3.2 illustrates a typical shared memory system architecture.

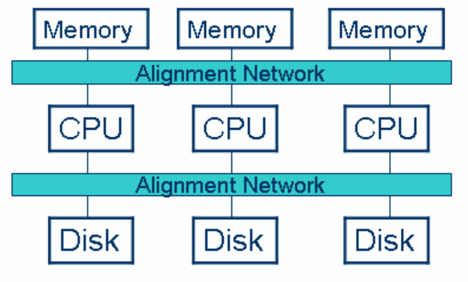


Figure 3.2 Shared memory architecture

- Distributed memory system

The distributed memory systems may use an unlimited number of powerful independent processors, each equipped with its own memory. The number of processors varies from two to thousands. In general, a message is constructed on one processor and is sent through an interconnection network to another processor, which then must accept and act upon the message. Although the overhead in handling each message may be high, there are typically few restrictions on how large the message can be. Thus, it can yield high bandwidth which can transmit a large piece of a data set from one processor to another in a very effective way. However, to minimize the need for expensive message passing operations, data structures within a parallel program must be spread across the processors so that most data referenced by each processor are in its local memory. Figure 3.3 illustrates a typical distributed memory system architecture.

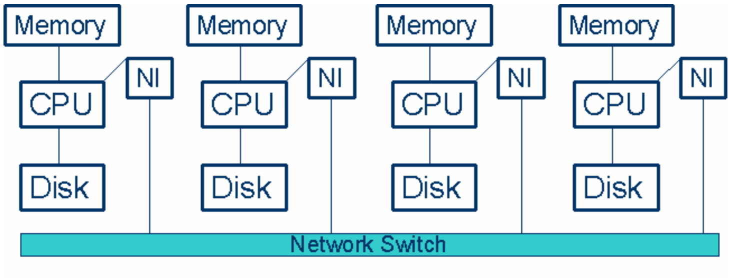


Figure 3.3 Distributed memory architecture

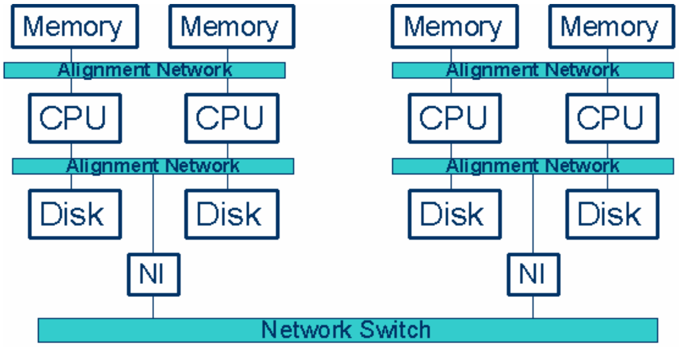


Figure 3.4 Combination system architecture

- Combination system

The combination systems have many computing nodes that are organized in a distributed manner. In particular, on each node two or four processors are linked as a shared memory system. High speed communication is available among those nodes.

Figure 3.4 illustrates a typical combination system architecture. At present this model is widely used on many kinds of parallel platforms.

For some extremely large linear systems, even the total distributed memory still cannot handle the whole problem. Hence, the out-of-core mechanism was proposed.

In Geodetic Science applications, parallel computing is usually focused on solving those large dense linear systems that traditional sequential computers or even vector supercomputers cannot accommodate. More details are found in Chapter 4.

3.4 Message Passing Interface – MPI

From the discussion above, we may already know the importance of communication to the whole parallel system. It may be relatively easier for the shared memory system, because all processors have the ability to access the public memory space. But when dealing with distributed or combination systems, each processor has its own memory. Then one processor cannot directly reference to another processor's local memory. So it is crucial to choose a suitable type of communication and arrange the communication explicitly. For shared memory systems, OpenMP is a good choice. Here we mainly focus on Message Passing Interface (MPI) (Snir et al., 1996) which is the most widely used message passing library in distributed and combination systems.

In May 1994, after two years' work, the first version of MPI was introduced to the public by a group of sixty people representing forty different organizations. It is available online at http://www.mcs.anl.gov/Projects/mpi/standard.html. MPI has been developed to be a standard message-passing library for distributed memory parallel computing. Now a days it has been implemented on a wide variety of parallel machines. Both FORTRAN and C interfaces are available in MPI-1. C++ and FORTRAN 90 are also generated in MPI-2. All parallelism is explicit: The programmer is in charge of generating the parallelism of the whole program and then applying suitable MPI commands. The parallel paradigm used here belongs to the SPMD (Single Program Multiple Data) category.

The first important object in MPI is the communicator which refers to the communication environment. It defines the scope on which a specific parallel execution happens. Processors can communicate only if they are in the same communicator. At most cases only one communicator is needed and it includes all the processors the programmer wants to use. Each processor has an identifying number, known as rank, within the communicator, which starts with zero. This ID number is used to distinguish the source and destination of messages. If one processor belongs to two different communicators, its two ranks may not be the same.

Point - to - point communication is the basic communication paradigm applied in MPI (Figure 3.5) (Ohio Supercomputer Center (OSC), 2003). It involves two processors: One processor sends an array of data elements to another processor. Sending operations usually requires the sending process to specify the data location, size, type and the destination. Receiving operations should match a corresponding send-operation and provide enough memory space for the upcoming message. There are two different ways to achieve this kind of communication:

- Blocking Sends and Receives

Blocking send and receive functions blocks the calling process. It can not return until the message transmission is completed. The meaning of completion to send is that the sent data can be reused after sending; to receive means that data received can be safely access. For sends, the data must be successfully sent or safely copied to system buffer space so that the variables passed to the send routine can be safely reused or overwritten. For receives, the data must be safely stored in the receive buffer so that it is ready for use. Here the blocking concept guarantees that message passing is successfully finished. But it also brings the possibilities of delays or even deadlock to the communication. In a deadlock situation each processor is blocked and waits for the other processor to act first. Then a careful administration of the communication is required.

- Non-blocking Sends and Receives

It is an alternative way to invoke sends and receives. Non-blocking means that a processing continues even if the message has not been transmitted successfully. So, it is deadlock-free and easy to arrange for the communication. But on the other hand, rewriting and reusing the data elements involved in a non-blocking send-and-receive routine may not be safe enough, which brings complexity to the coding.

There are four communication modes for both blocking sends and non-blocking sends (OSC, 2003).

- Synchronous send: It completes when the receiving processor starts operating the matching receive. It is useful when we want to know the receiving condition.
- Buffered send: It always completes.
- Standard send: Message sent without the information about the receiving status. It is the general-purpose send mode in MPI.
- Ready send: Sending when the receive processor is ready. It always completes.

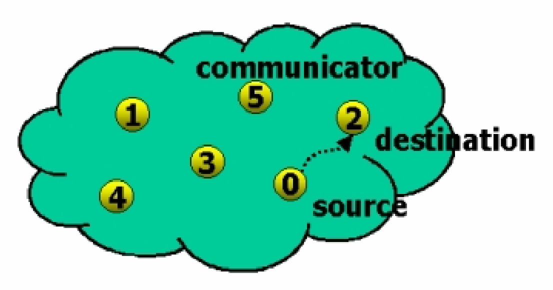


Figure 3.5 Point-to-point communication

For a more complicated communication involving multiple processors, MPI turns to collective communication. During this kind of communication all processors belonging to a communicator must call the collective routines at the same time. Some major operations it carries out are:

- Barrier: It works like a red-green light to make synchronization among all processors. It may result in some execution delay.

- Broadcast: It is a one-to-all communication. Message is passed by one root processor to all the other processors in the same communicator.
- Scatter: It is a one-to-all communication. The root processor divides the target message into several equal length chunks and sends each chunk to each processor in the communicator according to the rank order.
- Gather: It is an all-to-one communication. Each processor, including the root processor, sends a data set to the root processor. Then the root processor collects these data according to the rank order.
- All Gather: After finishing the gather routine, it broadcasts the result message from the root processor to all the other processors to let them own a copy.
- Reduce: First gathering data from each processor, it then reduces these data (sum, max or user defined operation) to a single value and stores it in the root processor.
- Other advanced operations.

As processors become faster and computers become more parallelized, the I/O part of a code often dominates the majority of the total wall clock time that the code needs to run. This bottleneck dramatically decreases the speed and efficiency of a parallel program. One solution to this problem is to parallelize I/O. Here we implement MPI parallel I/O because it also serves as the communicator in PLAPACK which is the parallel library used in this study. MPI parallel I/O is a new feature of MPI - 2, but this version is not completely realized yet. The version used in this study is the widely implemented MPI-1 which is also imported with parallel I/O functionality.

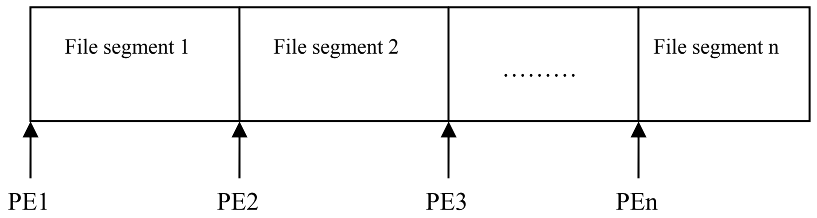


Figure 3.6 MPI parallel I/O

MPI parallel I/O implements the logical decomposition .Multiple processors read data from a single file at the same time and write data to a single file at the same time. It is like sending data to the disk and receiving data from the disk. Each processor has its own “view” to the file. The view defines the range of data that is accessible to this processor. The data type of the target file and the partition of the file are also included in the view. Figure 3.6 illustrates the concept of MPI parallel I/O.

3.5 PLAPACK

Accompanying the development for supercomputer hardware, corresponding numerical analysis packages with general purpose libraries and user-friendly programming interface and infrastructure were also developed for the scientific community. However, unlike the sequential implementation of numerical algorithms, this approach was greatly limited by the complexity of the hardware architecture, which usually needs to carefully distribute and allocate the data to the array of processing units and its corresponding memory. Inter-processor communication is a sophisticated operation and an additional component in parallel computing. All these extra tasks make the debugging process of the parallel code time-consuming. Thus, selecting a suitable numerical package before starting any parallel programming becomes a valuable effort for the overall project.

There are several popular parallel numerical packages available in the scientific community. LAPACK (Anderson et al., 1999) is a high performance and portable linear algebra library widely used in the sequential and central memory platforms. ScaLAPACK, based on LAPACK and proposed in the early nineties, exports the LAPACK libraries to the distributed memory parallel computers. It is designed for heterogeneous computing and is portable on any computer that supports Message Passing Interface (MPI) or Parallel Virtual Machine (PVM). The eventual goal of this package is to be suitable both in central memory and distributed memory systems (Blackford et al., 1996). Another significant effort to transform the sequential libraries to parallel libraries is the PLAPACK project. In this implementation, high level abstraction enables an object-based coding style, a “MPI-like” programming interface hiding the detail distribution and indexing task, and provides a very straightforward method for the user group (Alpatov et al. 1997). In this study we choose PLAPACK because, in its parallel implementation, high-level abstraction enables an object-based coding style that hides the details of matrix and vector distributions and indexing task, and thus provides a very straightforward user interface.

In order to achieve scalable performance, two-dimensional data distribution is required. Traditionally, there are three ways to partition and assign the matrix to processors: Blocked, wrapped or cyclic, and block wrapped.

The conventional matrix decomposition method is called “Two-dimensional block cyclic decomposition”, which obeys the following procedures:

- Dividing up the global array into blocks with m row and n columns, thereby thinking of the global array as composed only of these blocks.
- Distributing the first row of array blocks across the fist row of the processor grid in order. In the event of running out of processor grid columns, cycling back to the first column.
- Repeating this for the second row of array blocks, with the second row of the processor grid. In the event of running out of processor gird rows, cycling back to the first processor row and repeating it.

On the other hand, PLAPACK employs a new, application-centric distribution for dense matrices, called Physical-Based Matrix Distribution (PBMD). The authors argue that one should never try to decompose a matrix before considering how to decompose the physical problem; the distribution of the matrix to the processors is related to the distribution of the problem to be solved. For an abstraction of the problem described by the linear system $\boldsymbol{Nx}=\boldsymbol{c}$, the rules of physical-based matrix distribution are:

- The column of the matrix $\boldsymbol{N}$ should be assigned to the same column of processors as the corresponding elements of the vector $\boldsymbol{x}$.
- The row of the matrix $\boldsymbol{N}$ should be assigned to the same column of processors as the corresponding elements of the vector $\boldsymbol{c}$.

The method adopted by PLAPACK can obtain a better load balance than other Linear Algebra libraries. For example, if we consider a matrix-vector multiplication, using the traditional method, the elements of the vector $\boldsymbol{c}$ will invariably be distributed differently than the vector $\boldsymbol{x}$; extra communication is needed to redistribute the subvectors. In contrast, physical-based matrix distribution can be arranged to leave the result vector $\boldsymbol{c}$ distributed like the vector $\boldsymbol{x}$; no additional communication is needed.

Another benefit that physically based matrix distribution retains is for wrapped data distribution. By decomposing the vectors into more subvectors than processors, and applying either row or column wrapping, the matrix is wrapped, although tighter in one dimension than the other.

The PLAPACK package adopts an object-oriented coding style, such as the Message Passing Interface infrastructure. Like other popular Linear Algebra libraries, PLAPACK is also built on Level 1, 2, 3 BLAS and MPI. Data distribution also simplifies the infrastructure implementation. Thus, the code can be written in a natural way, and we do not need to attempt to recognize the algorithm and code as it is self-explanatory. But PLAPACK does not use the traditionally sequential coding method. Through high-level abstraction, PLAPACK code can be written in a more natural way, just by describing what we want to do and hiding all the technical details, thereby reducing the amount of code required. Moreover, it is easy to learn and convenient to debug.

Parallel processing requires efficient data distribution to computation nodes in order to achieve fast and efficient inter-node communication and loading balance. PLAPACK can do most of the work if one can make the abstraction of the problem to a linear system of equations. Inevitably, some overhead is added through the use of PLAPACK. However, the overhead can be kept to a minimum by handling the problem wisely. Specifically, inverting the normal equations matrix through the Cholesky or the QR decomposition method can be implemented very efficiently through its in-core algorithm. Gunter (2000) illustrates the efficiency and performance of PLAPACK via a traditional method for gravity field modeling. PLAPACK also provides the out-of-core algorithm for solving the linear system of equations that cannot fit into the physical memory of the computer node.

3.6 Performance theories of parallel processing

Gaining a better performance over a serial approach is the final goal of parallel processing. But as we stated above, doubling the computation resource cannot speed up the performance twice in many cases. There are many aspects that have influence on the execution of a parallel program. The interaction and communication among processors dominate the majority of the overheads associated with parallel programming. The fact that some processors may be at their idling status during the execution also contributes to this extra price. Synchronization at some point makes those processors that finish their task sooner, wait for others to end their task. Uneven loading among all computing nodes brings the same problem. When using the interacting mode to input some parameters needed for the program, usually only one processor is enough to work on. After that, this processor can broadcast the received information to all others. So, when only one processor is working on some serial or unparallelized job, other processors are left idling. Excess computation is another source for the overheads contracted with parallelism. Sometimes the best known serial algorithm is hard to be parallelized, and we have to apply parallelism on a poorer serial code. Excess computation is the difference in necessary operations between the algorithm of parallel programming and the best serial programming of the same problem.

Several metrics are introduced to measure the performance of parallel computation. Execution time is the direct indicator of system performance. It is easy to start and stop timing in the code without affecting the performance of the program. Beside execution time, measures such as speed-up, efficiency and scalability are also commonly referred to.

Speed-up: For the same problem, we denote T_s as the execution time of the best known serial algorithm, and let T_p be the execution time of a parallel algorithm applying p processing elements. Speed-up S(p*)* is defined as the ratio of these two times.

$$S(p) = \frac{T_s}{T_p}. \tag{3.1}$$

Theoretically, speed-up S(p) can not exceed the number of the processors used in a parallel algorithm.

Efficiency: It is defined as the speed-up over the number of processors implemented. In an ideal situation, the efficiency can reach 1 when the speed-up reaches p. But in practice, processors assign their time not only on the computation of the problem, but also on communication, idling and excess computation as stated before. So it varies between 0 and 1. We define the efficiency as E(p).

$$E(p) = \frac{S(p)}{p}. \tag{3.2}$$

Scalability: Usually we write and debug a parallel program in a small “scale”, which means that we only use a small amount of input data and run them on a small number of processors. But the final target is to solve a much bigger problem on a larger number of processors. So, scalability is an important factor when we examine the performance of a parallel system. It observes a parallel algorithm’s capacity to boost up its performance with increased processing elements. It also reflects a parallel system's ability to utilize increasing processing resources effectively (Grama et al., 2003).

Scalability is referred to the performance when both the problem size and the number of processors are increasing. A parallel algorithm is called scalable if the efficiency is kept constant as the problem size and the number of processors are increasing. In many cases, with a fixed number of processors, efficiency will increase as the problem size increases. On the other hand, with a fixed problem size, efficiency usually decreases as the number of processors increases. We can see these characters on examples in Chapter 5.

Amdahl’s law (Amdahl, 2000) is a simple, but useful model that provides the relationship between speed-up of a parallel application and the increase of the computing elements. We use *f* to denote the fraction of the serial component of a parallel algorithm; then, (1-*f*) is the fraction of the remaining parallel part. Let S and P represent the execution time of the serial and parallel segments of the whole problem running on one processor; then:

$$T_s = S + P, \tag{3.3}$$

$$T_p = S + \frac{P}{p}, \tag{3.4}$$

$$f = \frac{S}{T_s} = \frac{S}{S+P}, \tag{3.5}$$

$$1 - f = \frac{P}{T_s} = \frac{P}{S+P}. \tag{3.6}$$

By putting (3.5) into (3.4), T_p can be rewritten as:

$$T_p = f \cdot T_s + \frac{(1-f)\cdot T_s}{p}. \tag{3.7}$$

Then the speed-up S(p) is

$$S(p) = \frac{T_s}{T_p} = \frac{p}{f\cdot(p-1)+1}, \tag{3.8}$$

and the efficiency E(p) is

$$E(p) = \frac{S(p)}{p} = \frac{1}{f\cdot(p-1)+1}. \tag{3.9}$$

From equation (3.9) we can see that, when the problem size is fixed, the fraction factor *f* is constant. With the number of processors increasing, the efficiency decreases.

From what was discussed above, we can see that there are too many sources that play roles on the performance of a parallel system. In the hardware field, there are factors such as machine architecture, system design and operating system which are out of the user's control. At the user's level, the choices of language and corresponding compiler, program structure and the algorithm vary from one user to another; so, implementing the same parallel machine to solve the same problem by different people, different performances are to be expected. It is hard to build such a model that covers all of these influences completely. One fair solution is to track the system performance by applying execution time, while varying one or several elements of the parallel computation, such as the number of the processors or the problem size. So, in this study we use the execution time as our main measure to evaluate the effectiveness of different algorithms.

3.7 Computational environment

There are mainly four types of hardware configurations that support parallel computing. Among them, cluster has been viewed as a new member in the high performance computing (HPC) family in the past ten years. It belongs to the combination system class. Figure 3.4 shows how two nodes are connected with a high-speed network, and on each node there are two processors that share memory and hard drive. Compared with traditional HPC systems, cluster machines have many advantages. First of all, they are cheap and powerful. Usually they are constructed from commodity PC components with high performance/price ratio. Their theoretical performance approaches 1 GFLOP/sec. Secondly it is easy to build large-scale clusters with hundreds, or even thousands of processors. This type of supercomputer has the capability of computing faster and more precisely, and manipulating vast amounts of data. Moreover, the implementation of the high-speed inter-processor network helps to achieve the full potential that a cluster machine provides. All these characteristics make it currently one of the fastest growing parallel systems. In this study the implemented cluster machine is provided by the Cluster Ohio project, which is supported by the Ohio Supercomputer Center (OSC), the Ohio Board of Regents, and the OSC Statewide Users Group. It is a SGI 750 system with 16 computing nodes. On each node there are two 733 MHz Intel Itanium processors. Among these 16 nodes, one serves as the front end node that is in charge of interactive use, compiling and further tasks of this sort. All other 15 computing nodes are designated for serial and parallel jobs. A high speed system area network (SAN) connects each computing node with the other. It is a private network with no outside access, so that message passing can be performed efficiently and overhead can be minimized to a certain degree (Figure 3.7). The Itanium 2 cluster at OSC with 112 computing nodes for parallel jobs is also utilized for larger scale cases.

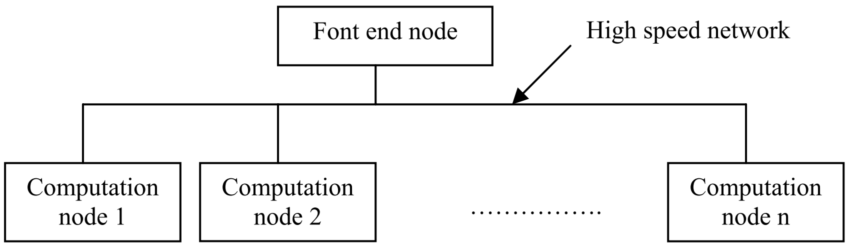


Figure 3.7 Architecture of cluster

CHAPTER 4

PARALLEL IMPLEMENTATIONS

4.1 Parallelism vs. linear algebra

Below is a small example of vector-matrix multiplication using PLAPACK. It will give an idea of how linear algebra computations fit into the concept of parallel processing, and how parallel computing is different from traditional serial computing.

In equation ***y*** = ***Ax***, we let ***A*** be a 4 x 4 matrix, ***x*** be a 4 x 1 vector, ***y*** a 4 x 1 vector. It can be rewritten as:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_{00} \\ a_{10} \\ a_{20} \\ a_{30} \end{bmatrix} x_0 + \begin{bmatrix} a_{01} \\ a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} x_1 + \begin{bmatrix} a_{02} \\ a_{12} \\ a_{22} \\ a_{32} \end{bmatrix} x_2 + \begin{bmatrix} a_{03} \\ a_{13} \\ a_{23} \\ a_{33} \end{bmatrix} x_3 \qquad (4.1)$$

From equation (4.1) we can see that the first column of matrix ***A*** only needs to be rescaled with the first element of the vector ***x***, and the same happens with the other columns of ***A*** and the other elements of ***x***.

At the same time, we also notice that

$$y_i = \sum_{j=0}^{3} a_{i,j} x_j \text{ for } i = 0,1,2,3. \qquad (4.2)$$

It means that the first row of the matrix ***A*** only corresponds to the first element of the vector ***y***, and the same is true again with the other rows of ***A*** and other elements of ***y***.

Based on the *Physically Based Matrix Distribution* (PBMD) concept adopted by PLAPACK, we start with the distributions of the vectors ***x*** and ***y***. We use 4 processors in this sample case. Usually a two-dimensional topology is applied to the whole desired computing units, and we rank them either by row major order or column major order. We take the column major order in this case and start from 0. Thus vector ***x*** and ***y*** are assigned to the computing grid in column major as shown in Figure 4.1.a. We project the indices of ***x*** to the top and the indices of ***y*** to the left, and then the distribution of the matrix ***A*** is determined. The result after this action is shown in Figure 4.1.b. After the blocks of the matrix ***A*** moved to their right locations in the node mesh, we still need to do something more to fix the appearance of the elements of the vector ***x*** to make the local matrix-vector computation possible. For processor 0, the first column of the block matrix $\boldsymbol{A}_{00,}$ namely $[a_{00}\ a_{20}]^T$, needs to be rescaled with $x_{0,}$ and x_0 does exists on processor 0. So, nothing needs to be done with x_0. But, for the other column of $\boldsymbol{A}_{00}$, it is a different situation. It is required to meet x_1 on processor 0, but x_1 only exits on processor 1. So, we have to get a copy of x_1 from processor 1, and do the same thing on other processors when necessary. By now, on each node, local matrix-vector multiplication is well defined and can commence (Figure 4.1.c).

After local matrix-vector multiplication, each processor owns a part of the result of ***y***. Summing within each row of the nodes, the final result of ***y*** is obtained within one column of nodes. The last step is to distribute the ***y*** vector to all nodes just like it was done with the ***x*** vector at the beginning (Figures 4.1.d and e).

Above is a general description of a parallel matrix-vector multiplication. In practice, target matrices and vectors are decomposed into blocks. The size of each block is known as distribution blocking size. Each element in this example is replaced by sub-vectors and matrix blocks.

4.2 Block scheme

During these days the hierarchical memory architecture is a dominant feature in almost all hardware platforms. The reason is that the time spent on data transfers between memory and registers is much longer than that of data operations on processor registers. There are three levels of caches used in the Itanium system to organize the data stream between memory and registers to achieve high performance. In this architecture, it is important to increase the ratio of computation over data movement. That means we need to apply an increased number of operations on a data set before moving on to a new data set. So, in order to meet the requirement of the hardware part and to fit naturally into the memory architecture, higher-level linear algebra operations are preferred. In this way we may expect better performance, closer to the theoretical peaks that are provided by the hardware. Thus the block approach is straightforward.

For current linear algebra numerical libraries, Basic Linear Algebra Subprograms (BLAS) has been treated as a standard. Almost all linear algebra packages mentioned in the last chapter are based on BLAS. It contains three levels of linear algebra operations. BLAS-1 is for vector-vector computations; BLAS-2 is for vector-matrix computations; BLAS-3 is for matrix-matrix computations. It is obvious that higher-level BLAS operations match the hierarchical memory architecture better.

In this study, we apply a block approach to implement higher-level BLAS operations, such as partitioning the matrices in question into several sub-matrices, etc.

4.3 Direct inverse parallel approach

The processing scheme includes the creation of a local normal equations matrix ***N*** and a vector ***c*,** or only a local design matrix on each computing node, then transforming the local contributions to global objects and, finally, estimating the parameters by inversion using the Cholesky decomposition.

4.3.1 Creation of local contribution

Input/Output often becomes the bottleneck in computing using fast processors. MPI parallel I/O is very efficient in such situations. Furthermore, each set of observations can be worked with independently, making parallelism straight-forward.

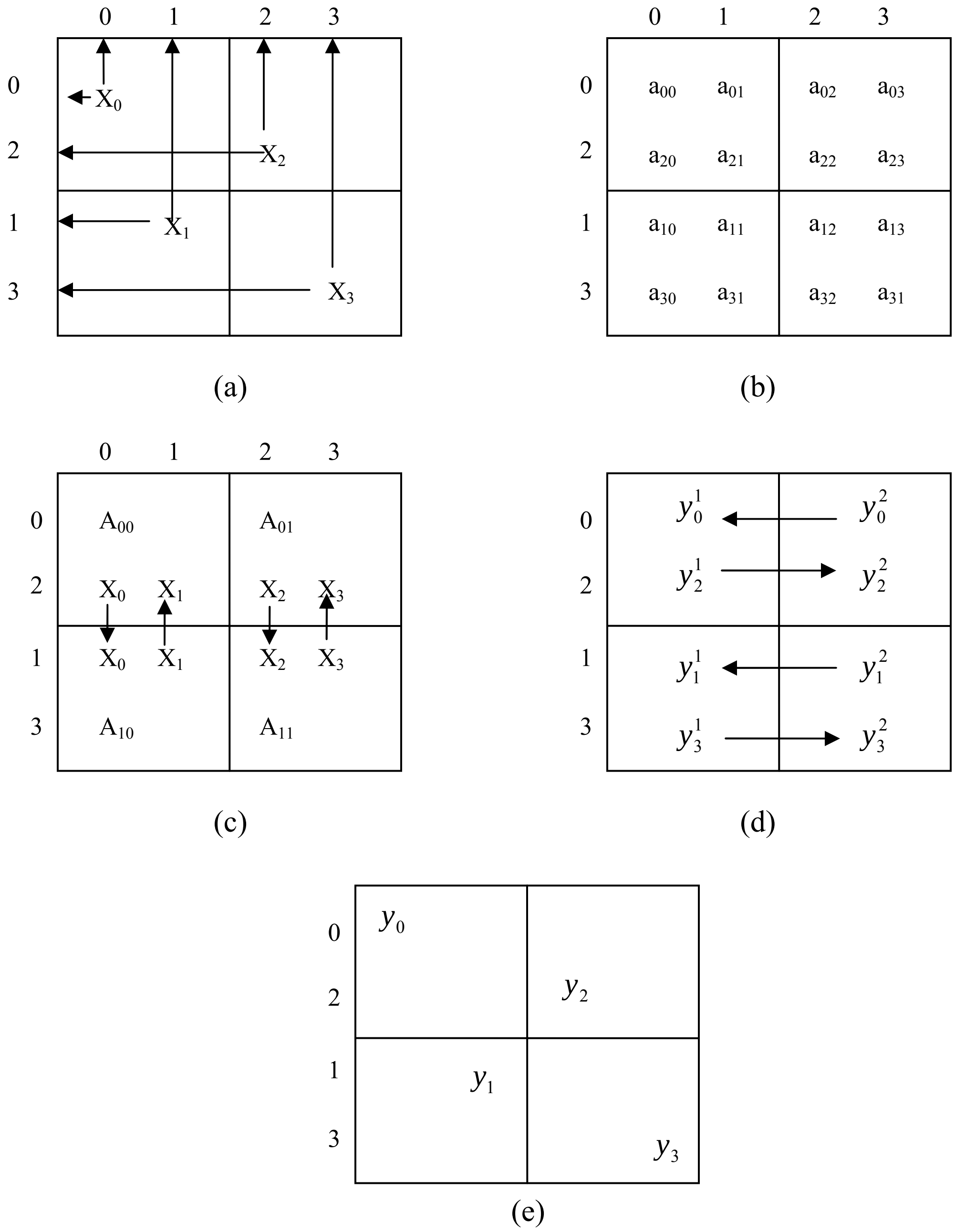


Figure 4.1 Illustration of matrix-vector multiplication

We divide all observations into smaller subsets, and as evenly as possible among a number of processors we desire. On each processor a view is defined to determine from where in the whole data file this processor begins its reading. Usually a local file pointer is placed on that designated position. In this way a huge set of data is decomposed into several small subsets of data and distributed among several processors. The parallel I/O technique of MPI is used in this case. Every processor operates on its assigned data simultaneously which greatly reduces the time spent on input. For each set of observations a series of computations is applied.

In particular, for latitude values of two satellites, the Legendre function is computed recursively. For other terms such as $\left(\frac{R}{r}\right)^{n+1}$, $\cos(m\lambda)$ and $\sin(m\lambda)$ in equations (4.3) and (4.4), recursive formulas are implemented for faster processing, simply because the trigonometric and power operations are more expensive than simple additions and multiplications. At the same time, new results are derived from old results. This avoids the unnecessary CPU usage to compute the early steps. For example, in the process of forming the design matrix ***A***, the value of $\left(\frac{R}{r}\right)^{n+1}$ is needed for every degree; instead of computing it directly using the intrinsic function in programming language, e.g., $\mathrm{pow}\left(\frac{R}{r}, n+1\right)$, the recursive formula below is adopted. The same approach is applied to the sine and cosine operations. In our test, this consideration eventually saves a lot of CPU time. The rule of thumb is to use as simple operations (addition, multiplication, subtraction) as possible.

$$\left(\frac{R}{r}\right)^{n+1} = \left(\frac{R}{r}\right)^{n} \cdot \left(\frac{R}{r}\right) \tag{4.3}$$

$$\begin{aligned} \cos(m\lambda) &= 2\cos\lambda\cos(m-1)\lambda - \cos(m-2)\lambda \\ \sin(m\lambda) &= 2\cos\lambda\sin(m-1)\lambda - \sin(m-2)\lambda \end{aligned} \tag{4.4}$$

By combining these terms a row vector of the design matrix ***A*** is formed, and its contributions to both the normal equations matrix ***N*** and the vector ***c*** are computed accordingly. We assume unit weights. As the level-2 matrix-vector operation is not as efficient as the level-3 matrix-matrix operation, a blocking algorithm is applied to this step. A well optimized BLAS-3 operation is chosen to construct the local design matrix ***A***. So, a block row instead of a row vector of the design matrix ***A*** is created. After repeating this procedure with every set of data on every node, several local normal equations matrices and corresponding vectors are obtained.

Also because of the independency of the observations we can distribute the work among the various computing nodes for their respective contributions to the normal equations matrix ***N*** and the vector ***c***. By accumulating all these contributions, we can generate the respective global normal equations matrix ***N*** and the corresponding vector ***c*** by equation (4.5), where p is the number of processors used:

$$\boldsymbol{N} = \sum_{i=1}^{p} \boldsymbol{A}_i^{T} \boldsymbol{A}_i, \boldsymbol{c} = \sum_{i=1}^{p} \boldsymbol{A}_i^{T} \boldsymbol{y}_i \tag{4.5}$$

Due to the memory limitation of our test platform, two methods were proposed for the problem namely the Normal equations Matrix Accumulation (NMA) and the

Design Matrix Accumulation (DMA). Figure 4.2 shows that NMA (left) begins with local computations on each node. For example, on processor P_I, after a local design matrix $\boldsymbol{A}_i$ is formed, local BLAS2/3 routines are applied to get the local normal equations matrix $\boldsymbol{N}_i$ and the vector $\boldsymbol{c}_{i,}$ respectively. By transferring local information to global objects, the final global normal equations matrix $\boldsymbol{N}$ and the vector $\boldsymbol{c}$ are obtained. The right side of Figure 4.2 illustrates the DMA approach: A global design matrix $\boldsymbol{A}$ is constructed after the creation of local design matrices (with $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$ created on processor P_I and P_J respectively), followed by the creation of the global normal equations matrix $\boldsymbol{N}$ and the global vector $\boldsymbol{c}$.

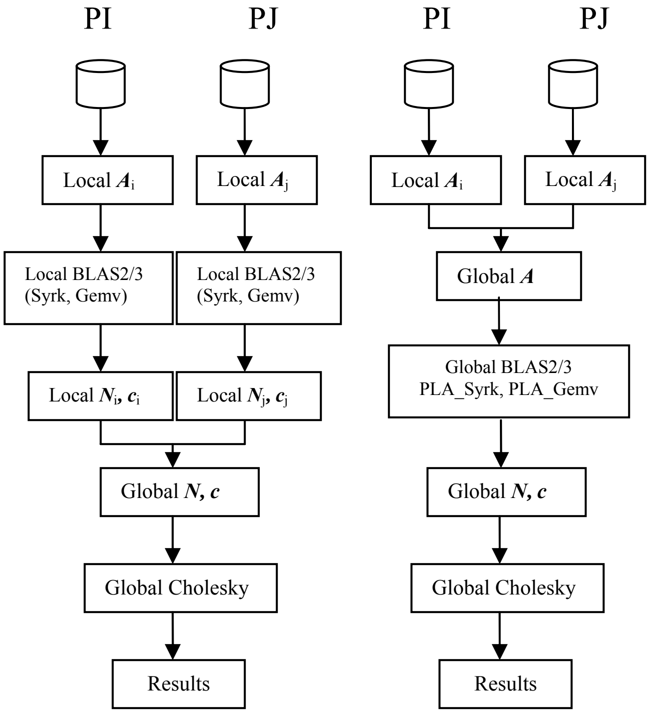

PI
PJ
PI
PJ
Local $\boldsymbol{A}_i$
Local $\boldsymbol{A}_j$
Local $\boldsymbol{A}_i$
Local $\boldsymbol{A}_j$
Local BLAS2/3 (Syrk, Gemv)
Local BLAS2/3 (Syrk, Gemv)
Global $\boldsymbol{A}$
Global BLAS2/3 PLA_Syrk, PLA_Gemv
Local $\boldsymbol{N}_i$, $\boldsymbol{c}_i$
Local $\boldsymbol{N}_j$, $\boldsymbol{c}_j$
Global $\boldsymbol{N, c}$
Global $\boldsymbol{N, c}$
Global Cholesky
Global Cholesky
Results
Results

Figure 4.2 Two implementations of parallel least-squares, Normal equations Matrix Accumulation method (NMA, left) and Design Matrix Accumulation method (DMA, right), (P_I and P_J stand for processor I and processor J)

Normal equations matrix accumulation means that the matrix $\boldsymbol{N}$ is formed on separate computation nodes and accumulated afterwards. The computation is independent on every node. In this way, usually a best performance could be achieved, based on the locally optimized BLAS-3 symmetric rank-k update (Syrk) call. However, this method requires a huge amount of memory. For example, to compute geopotential coefficients up to degree 120, a memory space of 14,637x14, 637*8 = 1,713,934,152 byte is required only for the $\boldsymbol{N}$ matrix alone. To circumvent the memory shortage, the Design Matrix

Accumulation method is adopted which, after forming local design matrices from individual data subsets, accumulates them to a global design matrix ***A*****,** and then the normal equations matrix ***N*** is created by using the global matrix ***A***. Since ***N*** is distributed among the computation nodes, this greatly alleviates the large demand for memory. In general, a large number of computation nodes will put less memory stress on the computation system. The distribution, on the other hand, would inevitably increase the communication overhead so that PLA_Syrk is slower than the Local Syrk. However further tuning and optimizing the parallel design and communication could minimize such effect.

4.3.2 Transformation of the local contributions to global objects

Moving local objects (matrix and vector) distributed on each node to global objects is very tricky in PLAPACK. PLAPACK provides this function through its own Application Interface (API). However, calling these APIs could cause inconsistencies or even deadlock if not properly used. Instead of these APIs, an alternative is to use the Copy function embedded in PLAPACK and a special object, called "multiscalar", which serves as a carrier in this particular situation. "Multiscalar" represents a non-distributed matrix exiting as a unit entirely on one node. It also can be duplicated on one row or column of nodes, or even on every node. For our specific problem, we first copy local information to a "multiscalar" on each node. Then reduce these "multiscalars" to one new "multiscalar" which is created conformal to the target global matrix ***N*** and the vector ***c***. The final step is to copy from this "multiscalar" to the global objects ***N*** and ***c***.

There is one drawback to this alternate way of transferring information from local to global level. Here, we need to create, on each processor, a matrix with the same size as the normal equations matrix ***N***. Considering the already existing local normal equations matrix which has the same size, there are now two large-size matrices on one processor which only has 4 GB memory shared with another CPU. So the lack of memory reduces our use of this method to smaller size problems. The API method is adopted in our DMA approach.

4.3.3 Cholesky decomposition

Cholesky decomposition is chosen to solve the normal equations to estimate the gravity model coefficients. Compared with QR factorization, when the number of observations is greater than the number of unknown parameters, the number of operations using Cholesky is almost half that of QR. Hence, we used the parallel Cholesky solver provided by PLAPACK.

4.4 Conjugate gradient parallel approach

As we can see, the direct inverse parallel approach requires huge memory occupation for its dense matrix accumulation and inversion. This drawback highly restricts its implementation to the recovery of spherical harmonic coefficients of higher degree and order. By now, the highest degree and order we can solve with it is 120. On the other hand, the conjugate gradient method is a good alternative. It avoids the accumulation and inversion of the normal equations matrix, thus makes the computation less time consuming. Meanwhile its requirement of memory is much smaller than that of the previous method. So it is suitable for very high degree and order gravity field estimation.

The first step is to obtain the ***c*** vector, the diagonal blocks in matrix ***N*** for the preconditioning matrix ***M***, and the $\boldsymbol{N}\boldsymbol{x}_0$ vector which are all prerequisites for the whole computation. Then we construct the preconditioning matrix ***M*** and its inverse to make the iteration converge faster. The final step is to update the estimated parameters through the iteration (equations (2.49)-(2.55)).

4.4.1 Preparation

As applied in the previous case, parallel I/O is also implemented here. The whole vast data set is evenly divided into *p* small subsets; here *p* is the number of processors we want to use. These subsets are distributed onto each node, and each node works on its own designated subset at the same time.

Compared with Han's algorithm (Han, 2004), we do not use the Taylor expansion of the Legendre function to speed up. Since for parallel computing, the time spent on exact the Legendre function is trivial, we just apply the exact recursive Legendre formula without approximation, which makes coding and debugging much easier.

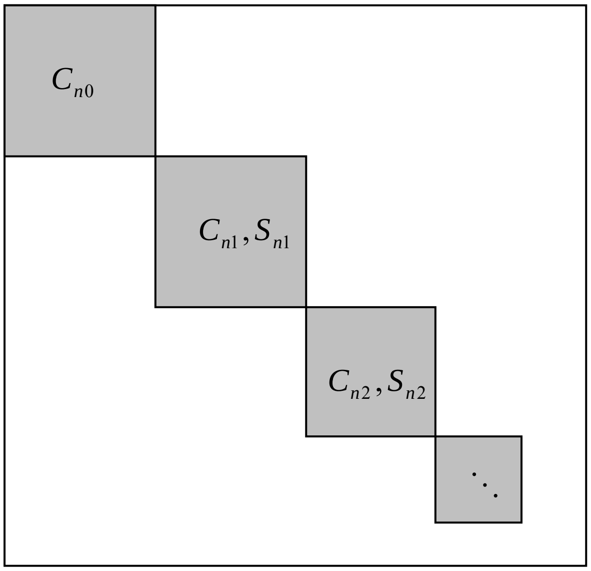


Figure 4.3 Structure of the preconditioning matrix ***M***

The great advantage of the conjugate gradient method lies in the fact that there is no huge matrix (e.g., the normal matrix) to be accumulated and inverted; no obvious matrix appears in the whole process. So, in this step two vectors are formed for further operations: $\boldsymbol{c}$ and $\boldsymbol{Nx}_0$, and a sparse matrix $\boldsymbol{M}$ which collects the blocks from the diagonal of $\boldsymbol{N}$ that acts as the preconditioning matrix in later computation. Since the off-diagonal elements of the preconditioning matrix $\boldsymbol{M}$ are zero, we store $\boldsymbol{M}$ as a one-dimensional vector in row major, which can save a huge memory space.

According to equation (4.5), the vector $\boldsymbol{c}$ is the sum of p local vectors $\boldsymbol{c}_\mathrm{i}$. Thus, on each node we first compute the local vectors $\boldsymbol{c}_{\mathrm{i},}$ respectively, and then accumulate all local contributions to the global $\boldsymbol{c}$**.** Here the construction of the $\boldsymbol{c}$ vector is the same as that from the NMA method in the direct inverse parallel approach.

From Chapter 2 we know that $\boldsymbol{Nx}_0 = \boldsymbol{A}^\mathrm{T}\boldsymbol{Ax}_0$.Following equation (4.5) again, we can get:

$$\boldsymbol{A}^\mathrm{T}\boldsymbol{Ax}_0 = [\boldsymbol{A}_1{}^\mathrm{T}\boldsymbol{A}_2{}^\mathrm{T}...\boldsymbol{A}_\mathrm{p}{}^\mathrm{T}]\begin{bmatrix}\boldsymbol{A}_1\\ \boldsymbol{A}_2\\ \cdots\\ \boldsymbol{A}_\mathrm{p}\end{bmatrix}\boldsymbol{x}_0 = [\boldsymbol{A}_1{}^\mathrm{T}\boldsymbol{A}_1 + \boldsymbol{A}_2{}^\mathrm{T}\boldsymbol{A}_2 + ... + \boldsymbol{A}_\mathrm{p}{}^\mathrm{T}\boldsymbol{A}_\mathrm{p}]\boldsymbol{x}_0$$

$$= \boldsymbol{A}_1{}^\mathrm{T}\boldsymbol{A}_1\boldsymbol{x}_0 + \boldsymbol{A}_2{}^\mathrm{T}\boldsymbol{A}_2\boldsymbol{x}_0 + ... + \boldsymbol{A}_\mathrm{p}{}^\mathrm{T}\boldsymbol{A}_\mathrm{p}\boldsymbol{x}_0 . \qquad (4.6)$$

So, we first need to make sure that every processor owns a copy of the vector $\boldsymbol{x}_0$. Then, after local design matrix $\boldsymbol{A}_\mathrm{i}$ is formed, we get local $\boldsymbol{N}_\mathrm{ii}\mathbf{x}_0$ by two matrix-vector multiplications. Hence we not only reduce the huge volume of computations needed for matrix-matrix multiplication, but also avoid the appearance of a dense normal equations matrix $\boldsymbol{N}$. For $\mathrm{N}_{\mathrm{max}}$=120, the size of $\boldsymbol{N}$ would be 14637x14637. The final step is then to gather all local vectors $\boldsymbol{N}_\mathrm{ii}\boldsymbol{x}_0$ and form the global vector $\boldsymbol{Nx}_0$.

The $\boldsymbol{M}$ matrix actually is the block diagonal part of the normal equations matrix $\boldsymbol{N}$ as shown in Figure 4.3 and will be stored as a vector in the memory. We choose this sparse matrix as a preconditioning matrix to make the system converge faster. The normal equations matrix $\boldsymbol{N}$ is a block diagonal dominant matrix, and the $\boldsymbol{M}$ matrix contains most of the information of the normal equations matrix **N**. So, we can use the inverse of the preconditioning matrix $\boldsymbol{M}$ to approximate the variance-covariance matrix of $\boldsymbol{x}$ which is essentially $\boldsymbol{N}^{-1}$. At the same time, the inversion of $\boldsymbol{M}$ is very easy to perform due to its block diagonal pattern (Han, 2004).

Since we need to recompute $\boldsymbol{Nd}$ in every iteration (equation (3.32)), we need to store the assigned input data on each local memory. On each computing node, from this local memory row numbers of observations are operated at one time to form a local design matrix $\boldsymbol{A}_\mathrm{i}$ each time. Then this $\mathrm{n}_{\mathrm{row}}$ x $\mathrm{n}_{\mathrm{coef}}$ matrix is divided into $\mathrm{N}_{\mathrm{max}}$+1 blocks. The number of columns in every block matches the dimension of the corresponding diagonal block of the normal equations matrix, which makes the matrix-matrix multiplication well defined. After that, a symmetric rank-k update operation is applied on these block matrices one by one. The results are stored on a local $\boldsymbol{M}_\mathrm{i}$ on row major. This procedure is illustrated in Figure 4.4. Thus all processors contribute to the global $\boldsymbol{M}$. The sum of all local $\boldsymbol{M}_\mathrm{i}$ amounts to the global $\boldsymbol{M}$, just as in the situation when we use the local $\boldsymbol{c}_\mathrm{i}$ vectors to create the global $\boldsymbol{c}$ vector. But since we still need to invert the single blocks in $\boldsymbol{M}$, we just leave these local blocks on each processor. By now, we have two global vector objects: $\boldsymbol{c}$ and $\boldsymbol{Nx}_0$, and p local blocks: $\boldsymbol{M}_\mathrm{i}$.

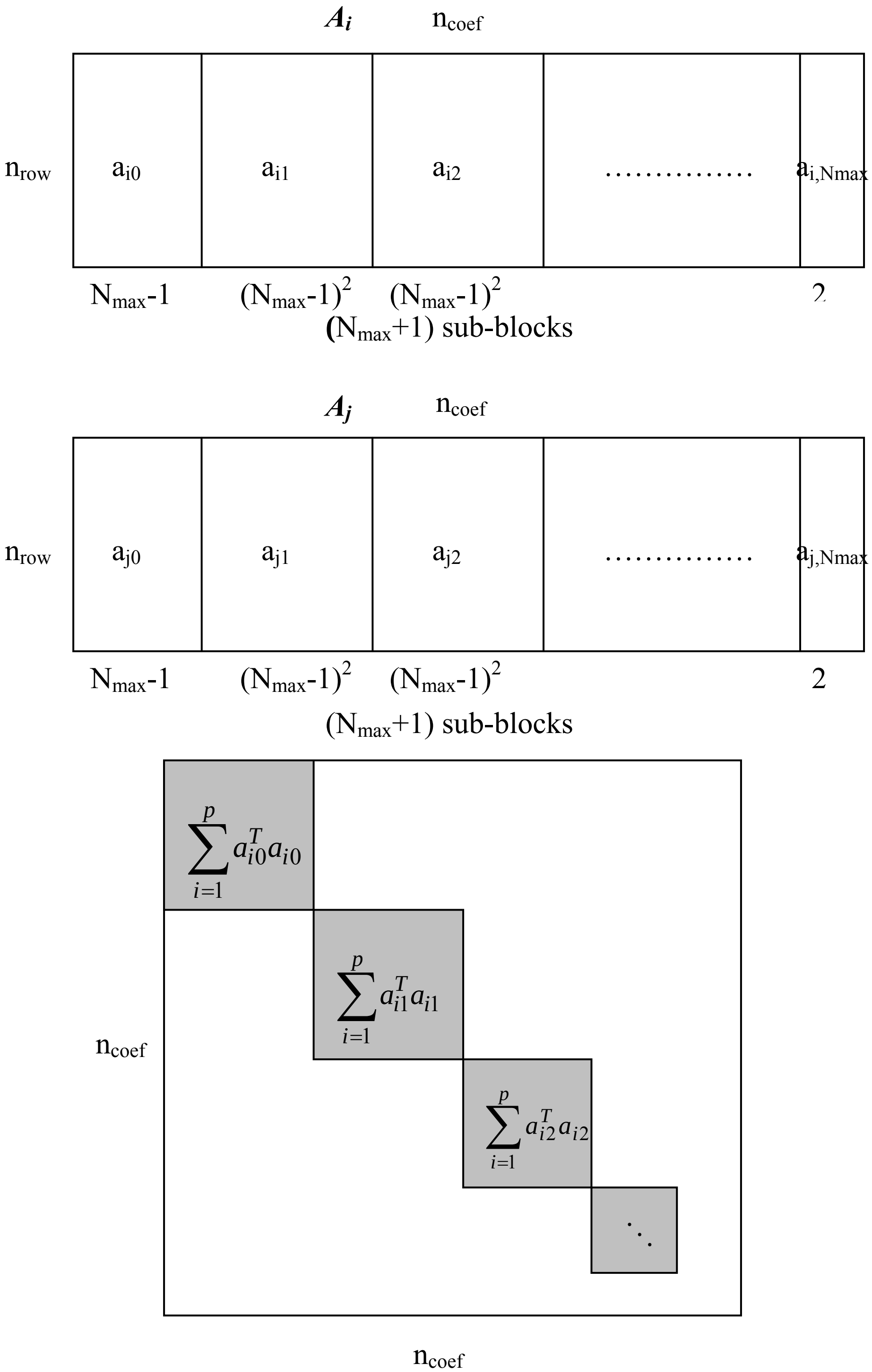


Figure 4.4 Creation of the local $\boldsymbol{M}_i$

4.4.2 Preconditioning

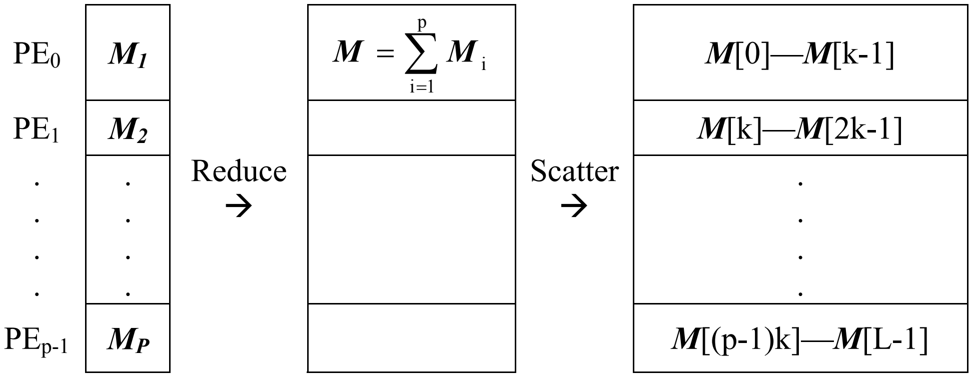

PE_0
PE_1
PE_{p-1}
$\boldsymbol{M_1}$
$\boldsymbol{M_2}$
$\boldsymbol{M_P}$
Reduce
$\boldsymbol{M} = \sum_{i=1}^{p} \boldsymbol{M}_i$
Scatter
$\boldsymbol{M}$[0]—$\boldsymbol{M}$[k-1]
$\boldsymbol{M}$[k]—$\boldsymbol{M}$[2k-1]
$\boldsymbol{M}$[(p-1)k]—$\boldsymbol{M}$[L-1]

Figure 4.5 Creation of the global $\boldsymbol{M}$ (p: number of processors, k: length of the segment of $\boldsymbol{M}_i$ on each processor, L: length of $\boldsymbol{M}$)

In this subsection we work on the creation of global conditioning matrix $\boldsymbol{M}$ and its inverse. Here the conditioning matrix $\boldsymbol{M}$ is a sparse matrix with non-zero elements only on its block-diagonal part. To save memory space, it is stored in a vector style in row major. So we cannot apply global inversion and multiplication operations provided by PLAPACK on this special format object. We have to manage the data distribution among processors and do these computations locally by ourselves. As we stated before, at this stage on each processor a local $\boldsymbol{M}_i$ already exists. We collect all these local contributions in the root processor and accumulate to get the final $\boldsymbol{M}$. But this object only exists on the root processor. When we apply parallel inversion processing, we want these blocks distributed among all processors evenly, and then do the inversion block by block, simultaneously on each node. The multiplication of $\boldsymbol{M}^{-1}$ also requires the assignment of blocks. So, by scattering, each processor receives almost the same number of blocks as the others (Figure 4.5). After that inversion, the matrix-vector multiplication can go smoothly. When the multiplication of $\boldsymbol{M}^{-1}$ is called for, the target vector also needs to be divided into blocks before we distribute these blocks to corresponding processors to make the computations correctly.

4.4.3 Iteration

This part is relatively easy to handle when compared with the previous two tasks. In the iteration all variables are global objects except the conditioning matrix. Thus we can take advantage of those easy-to-use functions provided by PLAPACK. We do not need to take care of the data distribution, message transformation, etc.

The whole scheme is shown in Figure 4.6.

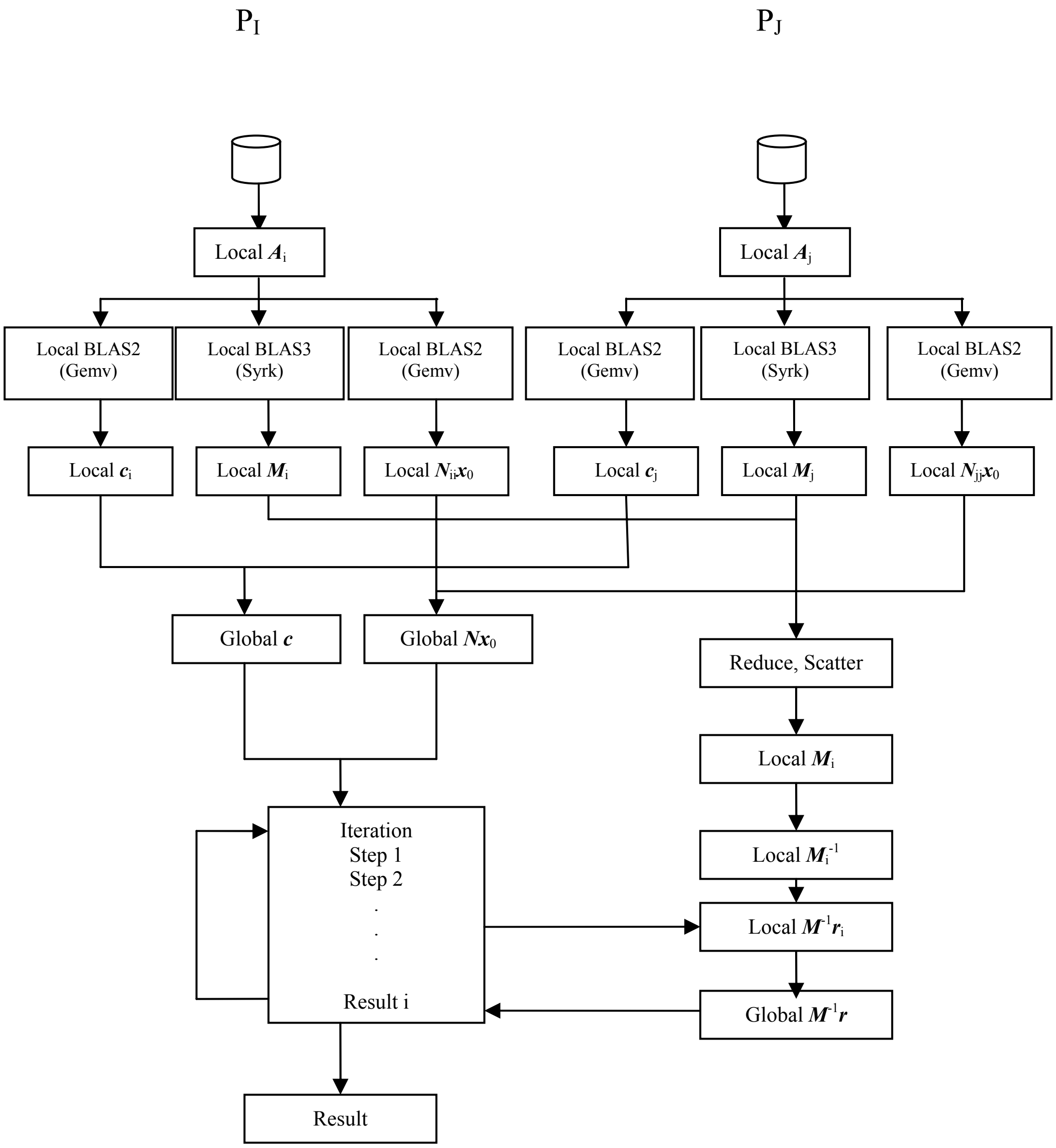


Figure 4.6 Parallel iteration scheme

CHAPTER 5

RESULTS AND ANALYSIS

In this chapter results from two independent gravity field recovery strategies are presented. In order to get a fair evaluation of the effectiveness of these strategies, they are tested on the same simulated GRACE data set and run on the same parallel platform. A comparison is given according to three major aspects: the quality of the estimated spherical harmonic coefficients and their variance – covariance information, performance (mainly the computation time), and other aspects like the ability to handle non-gravitational parameters, which is relevant in real GRACE mission data processing.

5.1 Results from the direct inverse method

The results shown in this section are based on a 30-day simulated GRACE orbit with 10-second time interval and an inclination of 89 degrees. The disturbing potential differences in equation (2.3) are computed from the simulated orbit using the EGM96 model. The computation platform is a 16-node SGI 750 system which is built on Intel Itanium processors. This system consists of 1 front end node and 15 computing nodes. On each node there are 2 processors sharing 4 GB memory. A high speed network (Myrinet) connects all these 16 nodes.

Figure 5.1 illustrates clearly that, when solving the same problem, the parallel solution is faster than the serial solution. As expected, for up to degree and order 30, parallel computing performs five times faster compared to the same task executed in a serial environment. Running on 24 processors, less than 20 seconds is needed for the parallel code, while for serial computing the whole running time is more than 100 seconds. For degree and order 120, as much as 3.5 hours are required for the serial job, which is nearly 14 times slower than the time for parallel execution. In Figure 5.1, the abscissa describes steps involved with generating the least-squares solution. In particular, step 1) includes input and creation of the matrix ***N*** and the vector ***c***, step 2) covers the actual computation, and step 3) the output.

Figure 5.1 shows that most of the time is spent on the input and the creation of the normal equations matrix ***N***. The floating point operations for generating the normal equations matrix from the design matrix account to (2n-1)(m+1)m/2, or approximately $O(nm^2)$, where n is the number of observations and m is the number of parameters. The floating point operations of the Cholesky decomposition amount to only $m^3/3$. When the number of observations is large and the number of parameters is small, the majority of the total time is spent on accumulating the normal equations matrix.

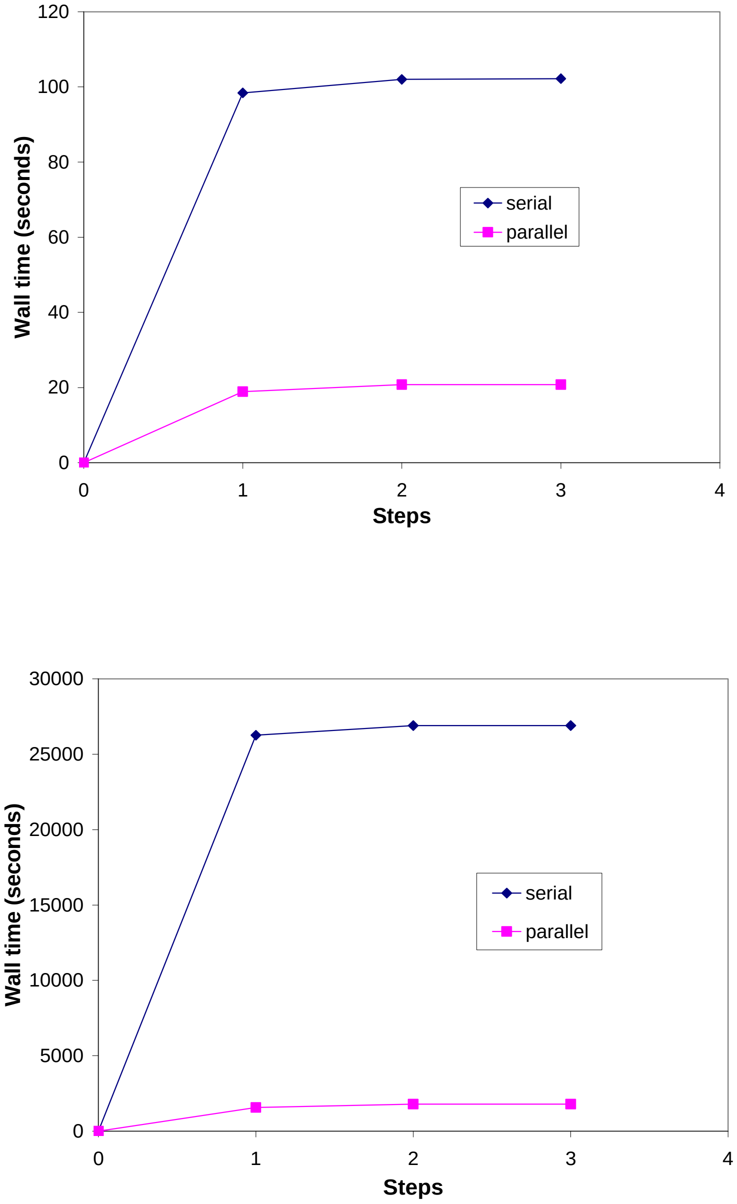
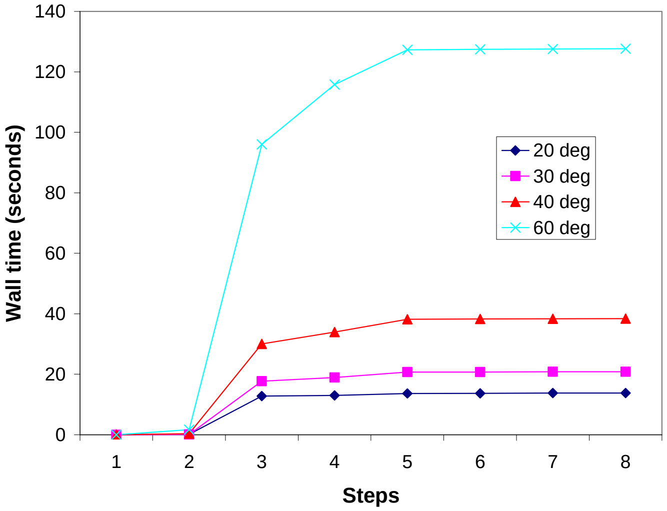


Figure 5.1 Execution time of serial and parallel direct inversion approaches. N_{max} =30 (top), DMA method ; N_{max}=120 (bottom), DMA method, 1 month of data, using 12 nodes (24 CPUs)

To compute the geopotential coefficients to a higher degree, say 120, the design matrix method (DMA) has to be used because of the limitation of the system memory. The normal equations matrix accumulation method (NMA), in the present configuration, can only solve the problem up to degree 60 (Figure 5.2(a)). The steps in Figure 5.2(a) are:

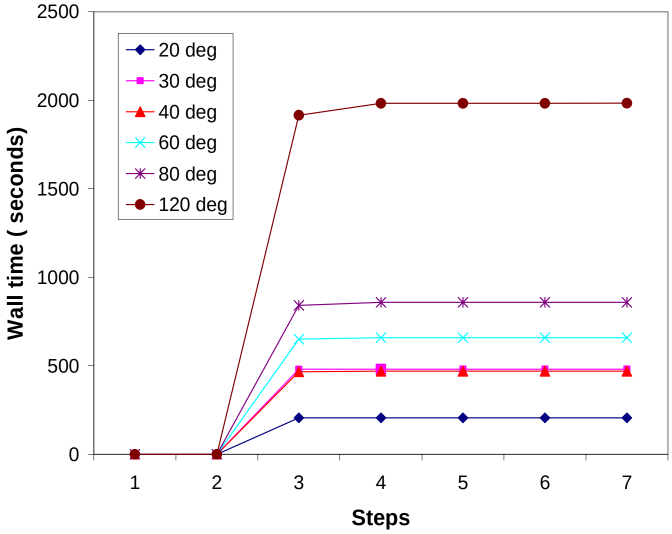
1) Initialization; 2) Preparing for reading; 3) Forming the local normal equations matrix $\boldsymbol{N}_{ii}$ and the local vector $\boldsymbol{c}_i$; 4) Going from local to global; 5) Cholesky decomposition; 6) First triangular solver; 7) Second triangular solver; 8) Output. The steps in Figure 5.2(b) are: 1) Initialization; 2) Preparing for reading; 3) Forming the local design matrix $\boldsymbol{A}_i$, the global $\boldsymbol{A}$**,** and the global normal equations matrix $\boldsymbol{N}$ with the global vector $\boldsymbol{c}$; 4) Cholesky decomposition; 5) First triangular solver; 6) Second triangular solver; 7) Output. Figure 5.2 shows that the NMA method is much faster than the DMA method. The total time spent for degree 60 using the NMA method is almost five times faster than when using the DMA method. The library used for the local BLAS-3 Syrk operation is highly optimized on the Itanium platform to fully utilize the CPU power. For global operations, communications are needed for different computation nodes. Beside the communication, the global Syrk operation is a combination of both the Syrk and Gemm call. The BLAS-3 Gemm call needs to compute the full matrix, which is less effective than the symmetric rank-k update. There are still possibilities to further optimize the DMA method code by carefully analyzing the code structure.



(a) Normal equations matrix accumulation method (NMA)

Figure 5.2 Performance of the NMA and the DMA method for geopotential estimation. (Using 12 nodes, 24 CPUs)(continued)

Figure 5.2: continued



(b) Design matrix accumulation method (DMA)

The solution for the geopotential coefficients up to degree 120 using the DMA method takes about 33 minutes on 24 CPUs for an exact solution. The computation time is dramatically reduced by running on parallel mode. Table 5.1 shows the speed-up and efficiency of the NMA and DMA methods with 24 processors involved. When the problem size is small, say up to degree and order 30 or 40, the number of speed-up for the DMA method is smaller than one. This means that its execution time is even longer than that of the serial mode. The extra time spent on communication and message passing is so big that the advantage of parallelism is sacrificed. But with the increasing problem size, the speed-up is increasing. With constant CPU number, efficiency is also increasing. Table 5.2 presents different performances of both methods when solving for unknown coefficients to degree and order 60. The number of computing units varies from 6 to 12, to 24. With the increasing number of CPUs, the efficiencies of both methods drop. The speed-up of DMA method with 24 CPUs is even smaller than that with 12 CPUs. It means that the communication overhead associated with the increased number of CPUs is dominant in this small problem size case. We also can see some similar situations in Table 5.3 and Table 5.4 due to the same reason. It is obvious that the achievement by the NMA method is much better than that of the DMA method. With increasing memory for the computation nodes and an optimized data transformation algorithm, the NMA method could achieve best performance by taking advantage of locally well-tuned scientific libraries to further reduce communication costs.

	Speed-up (NMA)	Speed-up (DMA)	Efficiency (NMA)	Efficiency (DMA)
degree 20	3.101	0.208	0.129	0.009
degree 30	4.913	0.212	0.205	0.009
degree 60	14.43	2.766	0.601	0.115
degree 120	-	13.566	-	0.565

Table 5.1 Speed-up and efficiency of the NMA and DMA methods (Using 12 nodes, 24 CPUs)

	Speed-up (NMA)	Speed-up (DMA)	Efficiency (NMA)	Efficiency (DMA)
6 CPUs	5.536	3.046	0.923	0.508
12 CPUs	10.261	4.625	0.855	0.385
24 CPUs	14.43	4.396	0.601	0.183

Table 5.2 Speed-up and efficiency of the NMA and DMA methods (N_{max}=60)

Number of CPU	6	12	24
Wall time (seconds)	79.1	39.3	29.7
Total speed (Mflops/sec)	9,221.1	18,559.6	24,558.6
Speed/node (Mflops/sec)	1,536.8	1,546.6	1,023.3

(a) NMA method

Number of CPU	6	12	24
Wall time (seconds)	404.5	379.7	466.2
Total speed (Mflops/sec)	1803.2	1,921.0	1564.5
Speed/node (Mflops/sec)	300.5	160.1	65.2

(b) DMA method

Table 5.3 Comparison of performance in constructing the normal equations matrix (N_{max}=40, # unknown=1,677, # observations=259,201, # total Mflops=729,391.8)

Number of CPU	6	12	24
Wall time (seconds)	4.0	4.4	4.2
Total speed (Mflops/sec)	393.0	357.3	374.3
Speed/CPU (Mflops/sec)	65.5	29.8	15.6

(a) NMA method

Number of CPU	6	12	24
Wall time (seconds)	3.0	2.5	2.8
Total speed (Mflops/sec)	524.0	628.8	561.4
Speed/CPU (Mflops/sec)	87.3	52.4	23.4

(b) DMA method

Table 5.4 Comparison of performance in Cholesky decomposition (N_{max}=40, # parameters=1,677, # observations=259,201, # total Mflops=1,572)

Table 5.3 compares the time and speed for computing the normal equations matrix using both the NMA and DMA methods. The total speed is defined as the total number of million floating point operations for a certain problem divided by computation time (wall clock time). Speed per CPU is defined as total speed divided by number of CPU. For the NMA method, although the total speed is increased as computing elements increase, the computation time is reduced accordingly. The parallelized code for the NMA method achieves scalability at a certain level. On the other hand, for the DMA method, the speed drops and computation time increases when the number of CPU increases from 12 to 24, because the inter-node communication impedes performance. This suggests that increasing nodes alone does not necessarily boost speed. Parallel code has to be fine-tuned to fit the system. The DMA code used here is only our first version; we will further analyze and tweak our code to improve its performance.

The results on time and speed performance of the Cholesky decomposition are listed in Table 5.4. Since the NMA and the DMA methods use a similar approach for the Cholesky decomposition, the results also appear similar. The numbers listed in the table are only for reference showing that, compared to the overall computational scheme, time spent in this step is very short.

5.2 Results from the conjugate gradient method

A series of case tests with variation of parameters of the computation has been performed on the SGI 750 parallel system which is the same machine involved in the direct inverse approach. The same simulated GRACE data set has also been applied here for fairness reasons. Another almost identical observation record, except for an inclination of 87 degrees, has also been processed.

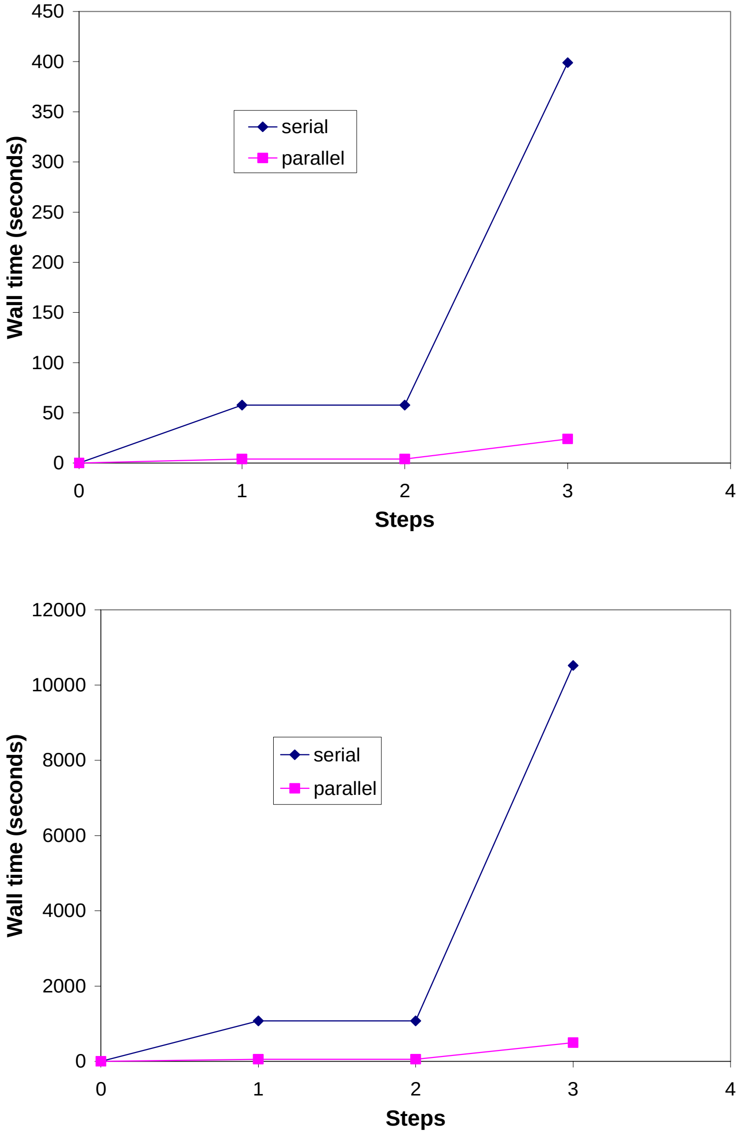


Figure 5.3 Execution time of serial and parallel computing when applying the iteration method. N_{max} =30 (top), 9 iterations, N_{max}=120 (bottom), 21 iterations. (1 month of data, using 12 nodes, 24 CPUs).

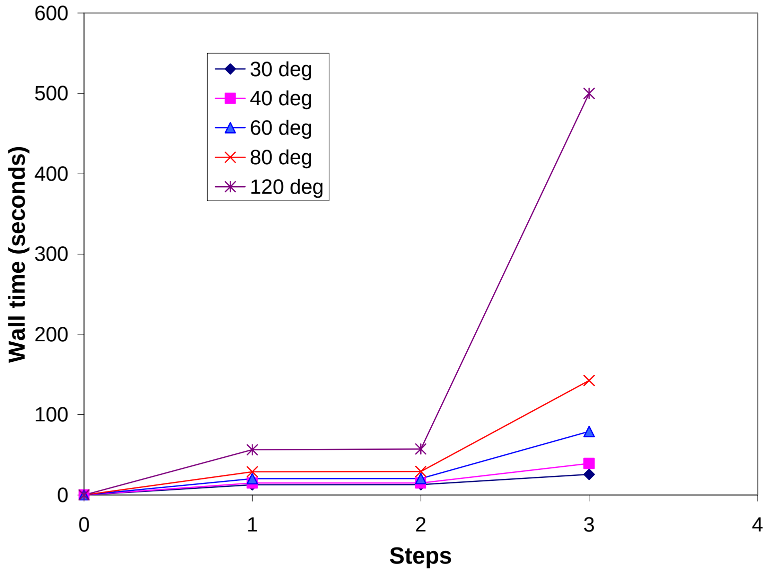


Figure 5.4 Performance of the iteration method for geopotential inversion (Using 12 nodes, 24 CPUs). Number of iterations: 9(degree 30), 10(degree 40), 11(degree 60), 12(degree 80), 20(degree 120).

Figure 5.3 shows the execution time of both a serial iteration strategy and a parallel iteration strategy when processing 30-day simulated GRACE data to solve for the unknown coefficients up to degree and order 30 and 120. For gravity recovery up to degree 30, running in parallel mode on 24 processors, the whole wall clock time is only 25 seconds. It is 16 times faster than that of the serial job. In Figure 5.3, the abscissa describes the steps involved with generating the iterative solution. Specifically, step 1) includes the system initialization and the creation of two global objects, $\boldsymbol{c}$ and $\boldsymbol{Nx}_0$, and local M_i, step 2) covers the creation of the preconditioning matrix $\boldsymbol{M}$ and its inverse, and step 3) the iteration. We can see from Figure 5.3 that for both serial and parallel approaches, no step occupies a huge amount of execution time, which is different from what we can see in the direct inverse method. No necessity for the accumulation of the normal equations matrix $\boldsymbol{N}$ is the significant advantage of this iteration method. Its direct significance lies in the fact that only matrix-vector computations are involved in this approach; thus the number of operations is reduced dramatically. We let the preconditioning matrix $\boldsymbol{M}$ be the block diagonal part of the normal equations matrix $\boldsymbol{N}$, its inverse $\boldsymbol{M}^{-1}$ is easily obtained with little computational effort. Accordingly, a nearly flat line reflects this part of work in Figure 5.3. We can also notice that the time spent on iteration is the dominant part of the whole computation time. This is so since that the matrix-vector multiplications are mainly located in this part, which is the most time-consuming operation in this method. With the problem size increasing, the number of iterations needed increases. In each iteration there are three matrix-vector multiplications, two in the equation (2.51), and one in the equation (2.54). So, approximately O(2nm + m^2) floating point operations are conducted in one iteration. Here m is the number of unknown parameters, and n is the number of observations. Figure 5.4 presents the various execution times when the problem size increases from degree 30 to degree 120, while keeping the computing nodes as 12. The abscissa is the same as in Figure 5.3. The time spent on iterations increases as the problem size and the number of iterations increase. On the other hand, times taken by the other two parts keep a relatively slow increasing rate.

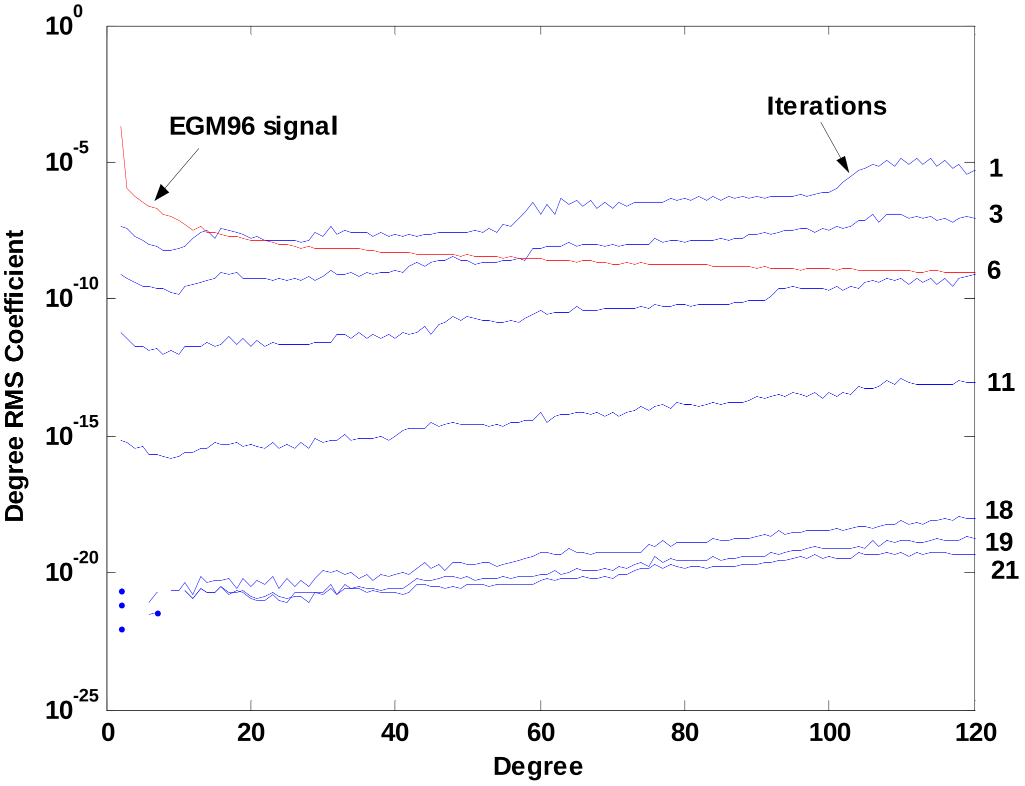


Figure 5.5 Degree RMS coefficient of EGM96 and the iteration behavior represented by the degree RMS coefficient given in equation (5.2) (Using 12 nodes, 24 CPUs)

In this study, EGM96 coefficients are treated as the “true” values, and the degree RMS coefficient is defined as:

$$c_n = \sqrt{\frac{\sum_{m=0}^{n}\left(\overline{C}_{nm}\right)^2 + \left(\overline{S}_{nm}\right)^2}{2n+1}} \tag{5.1}$$

Here $\overline{C}_{nm}$ and $\overline{S}_{nm}$ are the coefficients of our reference model EGM96. Then, the "degree RMS error" e^i_n defined in equation (5.2) gives us the mean value of the estimation error per degree in the i-th iteration. In this equation $\hat{\overline{C}}{}^i_{nm}$ and $\hat{\overline{S}}{}^i_{nm}$ are the intermediate adjusted values after the i-th iteration.

$$e^i_n = \sqrt{\frac{\sum_{m=0}^{n}\left(\hat{\overline{C}}{}^i_{nm} - \overline{C}_{nm}\right)^2 + \left(\hat{\overline{S}}{}^i_{nm} - \overline{S}_{nm}\right)^2}{2n+1}} \tag{5.2}$$

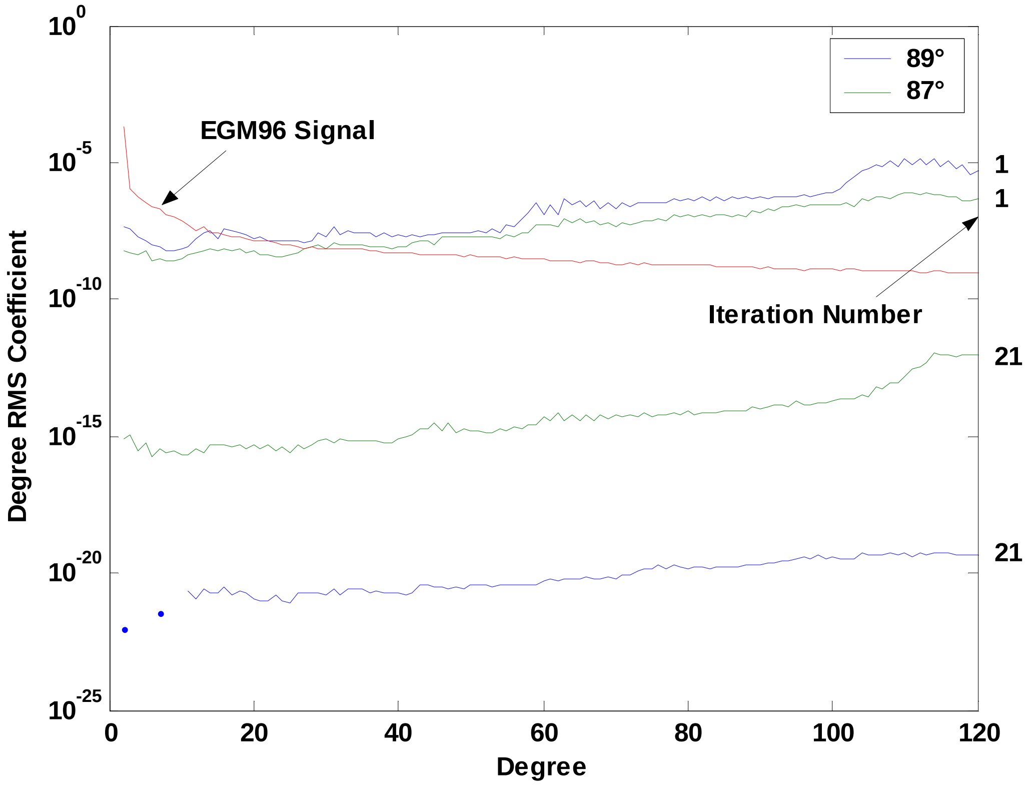


Figure 5.6 Degree RMS coefficient of EGM96 and the iteration behavior of two different data sets, one with an inclination of 87 degrees, the other with 89 degrees represented by the degree RMS coefficient given in equation (5.2) (Using 12 nodes, 24 CPUs)

Here initial values are set to zero and two keys are selected to control the flow of iteration. One is the largest number of iteration; the other is the RMS discrepancy from vector r_i, defined as:

$$(\text{discr})_i = \sqrt{\frac{|\boldsymbol{r}_i|^2}{n_{\text{coef}}}}\,,\ n_{\text{coef}} \text{ being the number of unknown coefficients.} \qquad (5.3)$$

Figure 5.5 shows the degree RMS coefficient of EGM96 and the iteration behavior characterized by e^i_n in equation (5.2). After 8 minutes, 21 iterations have been conducted with convergence to the" true" values.

We tested another set of simulated GRACE data, with the inclination at 87 degrees, to see the effect of different global coverage. Figure 5.6 shows the convergence behavior when comparing the two sets of simulated GRACE data, with inclination at 89 degrees and at 87 degrees. We can see that the data set with 89 degrees inclination delivers a better result and a better convergence rate than that of the data set with 87 degrees inclination.

5.3 Comparison

In this section we examine the above two independent gravity recovery approaches in three categories. Computational performance is characterized in execution time, speed-up, efficiency, and scalability. Estimated harmonic coefficients with their variance-covariance information are compared to decide which approach gains better accuracy. Other concerns, such as flexibility and computing resources are addressed in the third category.

5.3.1 Performance

Execution time is chosen as a major measure to evaluate the effectiveness of these two numerical solutions of the gravity field estimation problem. Here the DMA method is referred to as the direct inverse method. Figure 5.7 clearly indicates that the parallel iterative solution runs much faster than the parallel direct iteration solution. This result reflects the huge difference in the computation cost between these two algorithms. The direct inverse algorithm applies normal equations matrix accumulation which has about $O(nm^2)$ floating point operations. For the Cholesky decomposition there are another $O(m^3)$ floating point operations to be added. Here n is the number of the observations, and m is the number of unknown coefficients. Massive computations are required which result in a long running time. Meanwhile, a much smaller amount of computations is involved in the iterative algorithm. Only $O(2nm + m^2)$ floating point operations for the matrix-vector multiplication are needed in each iteration. Another factor that has influence on the execution time are the communication costs. In the DMA method the creation and existence of the global normal equations matrix $\boldsymbol{N}$ not only brings a memory shortage problem, but also raises the communication price. While the normal equations matrix is constructed, massive local data are collected and summed to a global object. Since the global normal equations matrix $\boldsymbol{N}$ is distributed on all computing nodes, a huge amount of message passing is expected when applying operations on this global object. In contrast, the iterative approach only has vector-type global objects involved in the whole iteration. Less communication is needed to complete the operations in question. In this sense, the iteration algorithm is more efficient than its competitor.

Scalability is an important measure to exam a parallel algorithm's effectiveness. When we increase both the number of processors and the problem size, and the efficiency stays constant, we call such a system scalable. Table 5.5 illustrates the scalability of three methods. As we can see, the implementation of both the NMA method and the conjugate gradient method is nearly scalable. This character means that solving larger problems with more nodes may be as effective as solving smaller problems, without wasting computing resources, and at the same time achieves good load balance. For the DMA method, the time consumed by communication overhead is relatively bigger for smaller problem size, which causes a very low efficiency. As the problem size grows, the overhead only occupies a small percentage of the total time. Thus, the efficiency increases gradually, but the number is still smaller than for the other two methods, due to the large number of non-local computations.

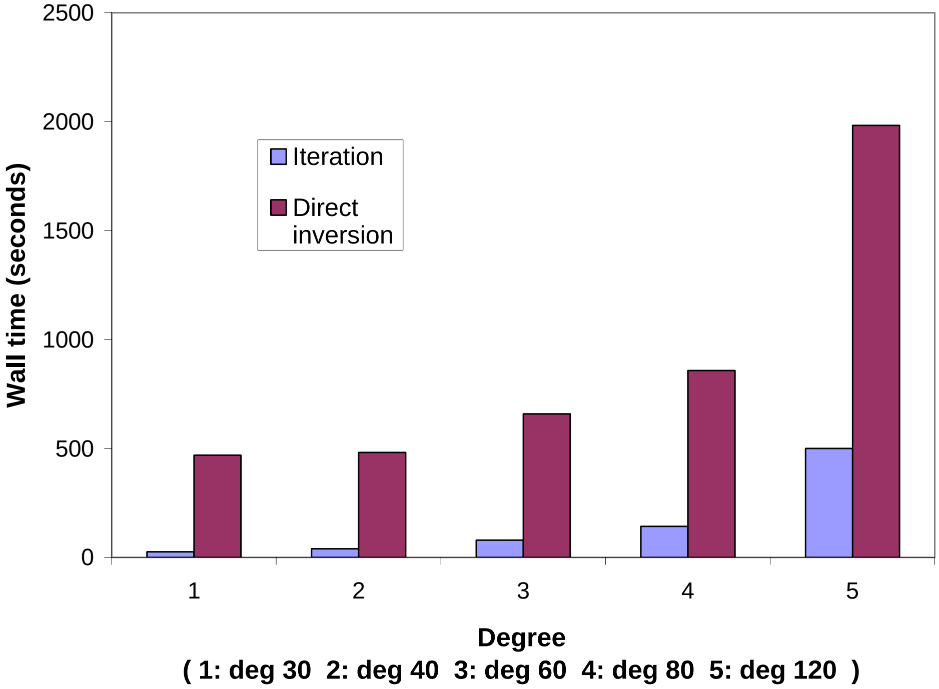


Figure 5.7 Execution time comparisons of the parallel iteration method and the parallel direct inverse method (DMA). (1 month of data, using 12 nodes, 24 CPUs).

	NMA Efficiency	DMA Efficiency	Conjugate Gradient Efficiency
20° 4 CPUs	0.943	0.156	0.750
30° 8 CPUs	0.876	0.196	0.734
60° 12 CPUs	0.855	0.385	0.684
120° 24 CPUs	-	0.565	0.881

Table 5.5 Scalability of three methods

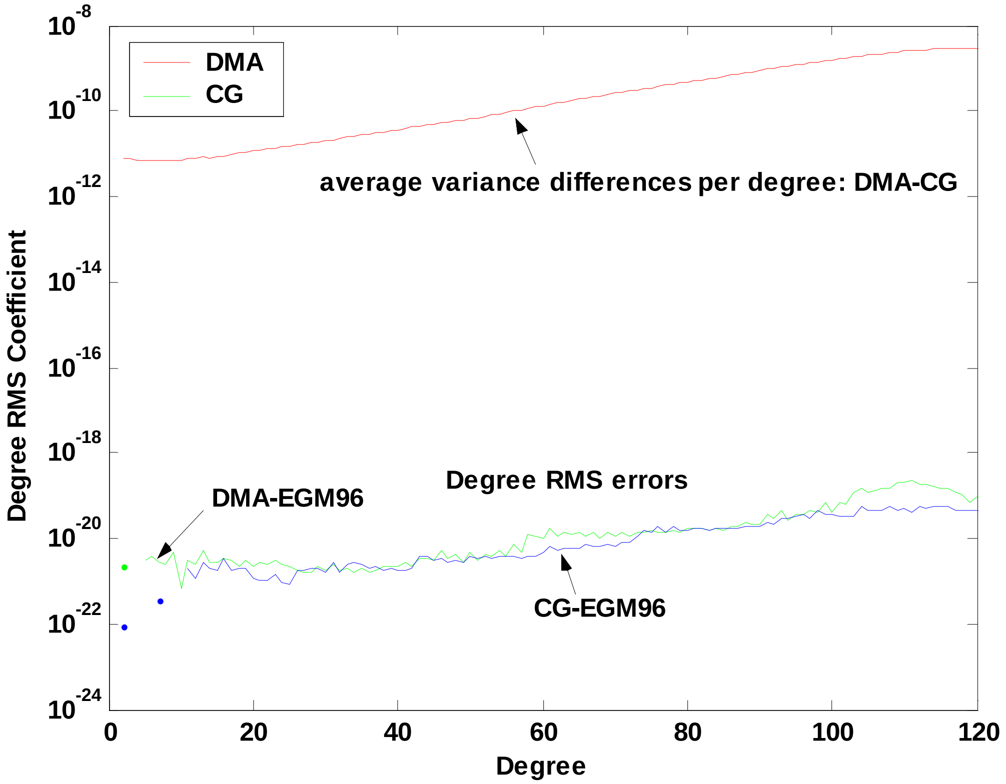


Figure 5.8 Degree RMS errors of the DMA method and the iteration method respectively (bottom), and the average variance differences per degree of these two methods (top). (Using 12 nodes, 24 CPUs)

5.3.2 Accuracy

No exact estimate for the variance-covariance information is provided by this iteration method, while it comes out with ease in the direct inverse method. However, the preconditioning matrix ***M*** is the block-diagonal part of the normal equations matrix ***N***. So, $\boldsymbol{M}^{-1}$ may be used to approximate the variance-covariance matrix $\boldsymbol{N}^{-1}$ to some degree. And ***M*** is a sparse matrix with non zero elements only on the diagonal-block part; thus, its inverse $\boldsymbol{M}^{-1}$ is easy to compute. Figure 5.8 illustrates that the "degree RMS errors" defined in equation (5.2) from both methods are nearly identical. The line on top of Figure 5.8 shows the average variance differences per degree given in equation (5.4) from the DMA method and the iteration method. The difference is small enough to indicate that both approaches can provide an equally accurate solution to the gravity field estimation problem. This conclusion is also consistent with other studies, e.g., Han (2004) and Pail and Plank (2002).

$$\sigma_n = \sqrt{\frac{\sum_{m=0}^{n} (\hat{\sigma}_{C_{nm}}\big|_{\text{direct}} - \hat{\sigma}_{C_{nm}}\big|_{\text{iteration}})^2 + (\hat{\sigma}_{S_{nm}}\big|_{\text{direct}} - \hat{\sigma}_{S_{nm}}\big|_{\text{iteration}})^2}{2n+1}} \qquad (5.4)$$

5.3.3 Other aspects

Computation resources form an important factor that we need to take into consideration when designing and evaluating a numerical algorithm. The direct inverse method gives us the exact solution to the inversion problem and provides an accurate variance-covariance matrix to the estimated parameters. But this accuracy comes at a price. Huge amounts of floating point operations lead to long CPU times. Meanwhile, PLAPACK cannot handle symmetric matrices in its memory; so, a full normal equations matrix has to be stored in the memory. For N_{max} =120, almost 2 GB memory is needed only for the storage of the normal equations matrix. This requirement indeed decreases the popularity of the parallel direct inverse method.

In this simulation study only gravitational coefficients are treated as unknown parameters. In real GRACE data processing, some non-gravitational parameters (e.g., nuisance parameter) are also needed to be taken care of. Nuisance parameters disappear from the equation system when the "reduced weight matrix" is used. This would indeed destroy the sparsity. In this way, the block-diagonal dominant feature of the normal equations matrix ***N*** no longer exists. Thus, the iteration method cannot produce results as good as in this simulation study. But this situation has little effect on the direct inverse method.

CHAPTER 6

CONCLUSIONS

6.1 Summary and conclusions

The parallel implementation of the least-squares algorithm for the solution of the gravitational coefficients based on the energy conservation method is presented in this study. Both a direct inversion and an iterative (conjugate gradient) method are implemented on a distributed computing environment—a 16-node SGI Itanium cluster. We have shown in the previous chapter that parallel code greatly reduces the computational time using the moderate commodity cluster. The simulated data are all based on one month of GRACE orbit and the EGM96 Earth gravity model. We chose PLAPACK over other numerical libraries for the study. Although it was found out later in the process that there are some drawbacks to using it, its object-oriented design and "physical based data distribution" architecture hide the detailed implementation of distributing data, which greatly releases the burden of explicitly managing and delivering data on the distributed nodes. Some explicit BLAS operations and MPI, MPI parallel I/O functions are utilized as well.

From a preliminary analysis, we see that data accumulation for the normal equations matrix is the most time-consuming and memory-demanding task; the inversion of the matrix using Cholesky decomposition does not need too much time. In the realization of a strict least-squares method, due to the physical memory limitation, we considered the Normal equations Matrix Accumulation (NMA) method and the Design Matrix Accumulation (DMA) method. NMA could potentially achieve better performance than DMA with the use of highly optimized local libraries, since it does not need any data communication when forming the local ***N*** matrix. However, this method becomes impractical with geopotential coefficients beyond degree 60. The NMA method also shows scalability to a certain extent. On the other hand, DMA method could solve for spherical harmonic coefficients up to degree 120, but at a much slower speed than the NMA method. The DMA method takes about 0.5 hours to solve for the spherical harmonic coefficients up to degree 120 using 24 CPUs. With careful optimization and code-tuning, we replace the unnecessary duplicate computations and complicated operations (e.g., trigonometry, power functions) with the recursive formulas that only use simple operations (addition, subtraction and multiplication). At the same time, we take advantage of the hierarchical architecture of the Itanium processors, and use BLAS-3 (matrix-matrix operations) whenever possible. By adjusting the "data I/O block size" (number of observations read in at one time), "PLAPACK distribution block size" and "PLAPACK algorithmic block size", better performance could be achieved. For example, when the matrix sizes are multiples of the "distribution block size" and "algorithm block size", PLAPACK will distribute and manipulate matrices in the most efficient manner. One direct example of this optimization is that the solution for the spherical harmonic coefficients up to degree 120 using serial code only takes around 7.5 hours. On one hand, PLAPACK eases our work, on the other hand, it also increases the program overhead (the consumption of CPU and memory by the package itself). The increase of communication costs in the DMA method through the high speed network slows down the program performance. The advantage of the direct method is that we use exact formulas in every step; hence, there is no approximation. It is very easy to obtain a full variance-covariance matrix.

The iteration (conjugate gradient) method is also being implemented in this study. The serial conjugate gradient method needs about 3.5 hour to compute spherical harmonic coefficients up to degree 120, while the parallel code takes only 8 minutes to finish 21 iterations on 24 CPUs with the same result. It also has the same accuracy compared with the strict least-squares method. The use of the inverse block-diagonal matrix from $\boldsymbol{N}$ for preconditioning and to approximate the variance-covariance matrix proves to be successful. One reason that the conjugate gradient method runs so fast is that it does not need to store the whole $\boldsymbol{N}$ matrix, and nearly all the computations are matrix-vector operations. This greatly reduces the memory usage and makes the separate local operations possible. The only communication happens at the stage of "reducing" local results; there is no communication during the local operations. This usually is not the case when doing global operations where a large amount of data is exchanged between nodes (DMA method). The other reason is that the number of operations in the conjugate gradient method is less than that of the direct inversion method.

Both implementations can be revised to become a general purpose least-squares solver for any geophysical inverse problem.

6.2 Recommendations

As mentioned above, PLAPACK and other numerical libraries (e.g., ScaLAPACK) also have some drawbacks. These packages require the use of full matrices, even though these matrices are symmetric. The inflexibility causes a big waste of system memory, and sometimes requires special design of the codes to circumvent the memory shortage problem. The communication routines in PLAPACK are not robust and sometimes cause deadlock (freeze) of the code when there are large amounts of communication traffic. Tricks have to be played to work around the problem. We only use those routines if we cannot find another way. Sometimes, direct use of MPI functions is more robust. Adding symmetric matrix support in PLAPACK would increase its flexibility and broaden its usage to even larger linear systems. If one wants to achieve even better performance, direct usage of MPI could reach the goal; but the code development cycle will be longer.

Not every code can be parallelized; the dependency of the data flow and the complexity of parallelization are main factors to consider. Procedures not depending on each other are the best suit for parallel realization, since less communication will occur. Improvement is expected by further analyzing and tuning the code.

For even larger problems that cannot fit into the physical memory, out-of-core methods could be implemented. In particular, for near-singular normal equations matrices, we need to implement a parallel version of the QR decomposition or the Singular Value Decomposition (SVD) method.

LIST OF REFERENCES

Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., Van de Geijn, R., Wu, Y.J. (1997), PLAPACK: Parallel Linear Algebra Package, In *Proceedings of the SIAM Parallel Processing Conference* (Also available at http://www.cs.utexas.edu/users/plapack/pubs.html).

Amdahl, G.M. (2000), Validity of the single processor approach to achieving large scale computing capabilities, *Readings In Computer Architecture*, Mark D. Hill, Norman P. Jouppi and Gurindar S. Sohi (eds.), Morgan Kaufmann Publishers Inc: San Francisco, CA, pp. 79–81.

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

Bettadpur, S. (2004), UTCSR Level-2 Processing Standards Document, Tech. Report GRACE 327-742, The Center for Space Research, University of Texas at Austin.

Blackford, L.S., Choi, J., Cleary, A., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. W., Whaley, R. C. (1996), ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance, In *Proceedings of Supercomputing '96*, published on CD-ROM format by IEEE Computer Society Press (also available at http://users.cs.cf.ac.uk/David.W.Walker/LINALG/sc96.html).

Flynn, M.J. (1972), Some computer organizations and their effectiveness, *IEEE Transactions on Computers*, Vol. 21, No.9, pp.948-960.

GS 651 (2002), Adjustment Computations II, class notes, The Ohio State University, Columbus, Ohio.

GRACE Science Mission Requirement Document (2000), Tech. Report GRACE 327-720, The Center for Space Research, University of Texas at Austin.

Grama, A., Karypis, G., Kumar, V., Gupta, A. (2003), *Introduction to Parallel Computing*, Second Edition, Addison-Wesley: Harlow, England / New York.

Golub, G., van Loan, C. (1996), *Matrix Computations*, Third edition, Johns Hopkins University Press: Baltimore, MD.

Gunter, B.C. (2000), Parallel least squares analysis of simulated GRACE data, Master's Thesis, Department of Aerospace Engineering and Engineering Mechanics, University of Texas at Austin.

Gunter, B.C., Tapley, B.D., van de Geijn, R.A. (2001), Advanced parallel least squares algorithms for GRACE data processing, In Proceedings of IAG 2001 Scientific Assembly on CD-ROM, Budapest, Hungary.

Han, S.-C. (2004), Efficient determination of the global gravity field from satellite-to-satellite tracking mission, *Celestial Mechanics and Dynamic Astronomy*, 88, pp.69-102.

Jekeli, C. (1999), The determination of gravitational potential differences from satellite-to-satellite tracking, *Celestial Mechanics and Dynamic Astronomy*, 75, pp.85-101.

Klees, R., Koop, R., Visser, P., van den IJssel, J. (2000), Efficient gravity field recovery from GOCE gravity gradient observations, *Journal of Geodesy,* 74, pp. 561-571.

Klees, R., Koop, R., van Geemert, R., Visser, P. (2001), GOCE gravity field recovery using massive parallel computing, *Gravity, Geoid, and Geodynamics 2000*, International Association of Geodesy Symposia, Vol. 123, Springer-Verlag: Berlin/ Heidelberg.

Kusche, J. (2001), Implementation of multigrid solvers for satellite gravity anomaly recovery, *Journal of Geodesy,* 74, pp. 773-782.

Nagel, P.B. (1999), Multiprocessor implementation of algorithms for multisatellite orbit determination, Doctoral dissertation, Department of Aerospace Engineering and Engineering Mechanics, University of Texas at Austin.

OSC (2003), *Parallel Programming with MPI*, Ohio Supercomputer Center (OSC), workshop notes, The Ohio State University, Columbus, Ohio.

Pail, R., Plank, G. (2002), Assessment of three numerical solution strategies for gravity field recovery from GOCE satellite gravity gradiometry implemented on a parallel platform, *Journal of Geodesy* , 76, pp. 462-474.

Rebhan, H., Aguirre, M., Johannessen, J. (2000): The gravity field and steady-state ocean circulation explorer mission - GOCE, *ESA Earth Observation Quarterly,* 66, pp. 6-11.

Reigber, Ch., Schwintzer, P., Luhr, H. (1999), CHAMP geopotential mission. *Boll. Geof. Teor. Appl.* 40, pp. 285–289.

Schuh, W.D. (1996), Tailored numerical solution strategies for the global determinations of the Earth's gravity field, *Mitteilungen d. Geodat. Inst. d. TU Graz*, No. 81, Graz, Austria.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. (1996), *MPI: The Complete Reference*, The MIT Press, Cambridge, MA.

Tapley, B.D. (1973), Statistical orbit determination theory. In: *Recent Advances in Dynamical Astronomy*, B.D. Tapley and V. Szebehely (eds.), D. Reidel Publ. Co., Holland, pp. 396-425.

Tapley, B.D., Bettadpur, S., Ries, J.C., Thompson, P.F., Watkins, M. (2004), GRACE measurements of mass variability in the Earth system, *Science*, Vol 305, Issue 5683, pp. 503-505.

Van de Geijn, R.A. (1997), *Using PLAPACK: Parallel Linear Algebra Package*, The MIT Press, Cambridge, MA.

Van der Sluis, A., Van der Vorst, H.A. (1986), The rate of convergence of conjugate gradients, *Numerische Mathematik* 48, No. 5, pp. 543-560.

Vaníček, P., Krakiwsky, E.J. (1986), *Geodesy: The Concepts,* Second edition, North Holland, Amsterdam, pp. 405-407.

Visser, P. N. A. M., Sneeuw, N., Gerlach, C. (2003), Energy integral method for gravity field determination from satellite orbit coordinates, *Journal of Geodesy*, 77, pp. 207-216.

Wolf, H. (1978), The Helmert block method, its origin and development, In *Proceedings of the Second International Symposium on Problems Related to the Redifinition of North American Geodetic Networks, Arlington, Va.*, pp.319-326.