Improving Parallellism for the NEMO ocean model

Author: Hind Shouli
Student number: 1100874
Daily Supervisor: Prof. dr. ir. C. Vuik
MSc Science Education & Communication
22-7-2011

TU Delft Delft University of Technology

Content

Acknowledgements

This work could not have been finished without a lot of people who supported me in many different ways. I would like t thank them all.

At the first place I would like to thank my daily supervisor prof. dr. ir. Kees Vuik for his guidance and useful comments on my work. I also would like to thank dr. Jeroen Spandaw, as he is a member of the thesis committee.

Secondly, I thank John Donners of SARA Computing and Networking services for answering my questions during my research.

Finally I thank my parents, husband and daughters for providing me all support I need to finish my Masters.

Chapter 1 Introduction

Nucleus for European Modeling of the Ocean (NEMO) is a 3-dimensional ocean model that is used for oceanography, climate modeling as well as operational ocean forecasting. It includes submodels that describe sea-ice and biochemistry. Many processes are parameterized, e.g. convection and turbulence. NEMO is developed in France and the UK by the NEMO development team and is used by hundreds of institutes all over the world.

NEMO is composed of “engines” nested in an “environment”. The “engines” provide numerical solutions of ocean, sea-ice, tracers and biochemistry equations. The “environment” consists of the pre- and post- processing tools, the interface to the other components of the Earth System, the user-interface, etc.

The open-source code consists of 100k lines of code fully written in Fortran 90. The MPI (Message Passing Interface) paradigm is used to parallelize the code and depending on the configuration of the model, it can run on a single processor or scale up to 1000 or more processors.

1.1 Model description

NEMO is a finite- difference model with a regular domain decomposition and a tripolar grid to prevent singularities.

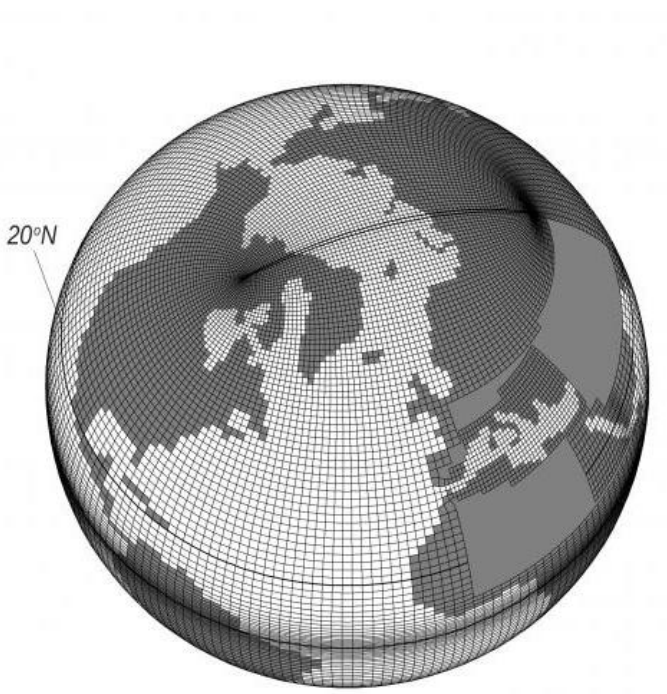

Figure 1: grid of NEMO-model.

The model calculates the incompressilble Navier-Stokes equations on a rotating sphere. The prognostic variables are the three-dimensional velocity, temperature and salinity and the surface height. The top of the ocean is implemented as a free surface, which requires the solution of an elliptic equation:

$$\left[\nabla \times \left[\frac{1}{H}\boldsymbol{k} \times \nabla\left(\frac{\partial \psi}{\partial t}\right)\right]\right]_z = [\nabla \times \bar{\boldsymbol{M}}]_z$$

to find $\frac{\partial \psi}{\partial t}$. Writing out this equation we get:

$$\frac{\partial}{\partial x}\left(\frac{k_3}{H}\frac{\partial \psi_t}{\partial x}\right) - \frac{\partial}{\partial x}\left(\frac{k_1}{H}\frac{\partial \psi_t}{\partial z}\right) - \frac{\partial}{\partial y}\left(\frac{k_2}{H}\frac{\partial \psi_t}{\partial z}\right) + \frac{\partial}{\partial y}\left(\frac{k_3}{H}\frac{\partial \psi_t}{\partial y}\right) = \frac{\partial \bar{M}_2}{\partial x} - \frac{\partial \bar{M}_1}{\partial y}$$

Since the vector $\boldsymbol{k} = (0, 0, 1)$ is the unit vector in z-direction, the equation becomes:

$$\frac{\partial}{\partial x}\left(\frac{k_3}{H}\frac{\partial \psi_t}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{k_3}{H}\frac{\partial \psi_t}{\partial y}\right) = \frac{\partial \bar{M}_2}{\partial x} - \frac{\partial \bar{M}_1}{\partial y}$$

For this purpose, it uses either a successive overrelaxation or a preconditioned conjugate gradient method. Both methods require the calculation of global variables, which incurs a lot of communication (both global and with nearest neighbors) when multiple processors are used.

1.2 Problem description

Both methods to calculate the free surface are of iterative nature, and are implemented as single subroutines of less than 200 lines. Their convergence rate is very slow and on average they need hundreds or even thousands of iterations to reach a satisfactory tolerance. At this moment, the scalability of NEMO is limited by the communication overhead of the free surface solver. As new computer architectures will consist of many more cores (100k+), it is imperative to improve scaling of NEMO.

The goal is to find new algorithms or to change the existing algorithms that improve the scalability of the NEMO model.

In numerical terms, this goal is dual: the algorithms should be changed in such a way that parallelism is improved, and algorithms should improve the convergence rate.

Chapter 2 Definitions and preliminary results

Here, an overview is given of relevant definitions, concepts and theorems used in later chapters.

2.1 Matrices

Consider the complex valued matrix A, with m rows and n columns, thus of size m x n. The element on the i-th row an j-column is denoted by a_{ij} .

The i-th row is denoted by:

$$a_{i*} = (a_{i1,}a_{i2}, \dots, a_{in}).$$

The j-th row is similarly denoted by:

$$a_{*j} = \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{pmatrix}$$

The transpose of matrix A is a matrix C whose elements are $c_{ij} = a_{ji}$, i=1, …, m, j=1, …, n. It is denoted by A^T.

A matrix is *square* if its size is $n \times n$. An important square matrix is the *Identity matrix I*, with ones at the main diagonal, i.e. $I_{ii} = 1, i = 1, \dots, n$ and $I_{ij} = 0$, if $i \neq j$.

The *inverse* of a matrix A, if it exists, is a matrix C such that: $AC = CA = I$. Denote the inverse of A by A^{-1}.

The *determinant* of A can be defined by the following recursive definition:

$$\det(A) = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} \det A_{1j},$$

where A_{1j} is an $(n-1) \times (n-1)$ matrix obtained by deleting the first row and the j-th column of A. Matrix A is said to be *singular* when $\det(A) = 0$. Otherwise it is *nonsingular*.

Eigenvalues

A complex scalar λ is called an *eigenvalue* of the square matrix A if a nonzero vector u of $\mathbb{C}^n$ exists such that $Au = \lambda u$. The vector u is called an *eigenvector* of A associated with λ. The set of all eigenvalues of A is called the *spectrum* of A and denoted by $\sigma(A)$.

It can be shown that a matrix is singular if zero is an eigenvalue. Otherwise, a matrix is nonsingular if and only if it has an inverse. That is why a nonsingular matrix is also called an *invertible* matrix.

Thus, the determinant of a matrix determines whether or not the matrix has an inverse. The maximum modulus of the eigenvalues is called the *spectral radius* and is denoted by $\rho(A)$

$$\rho(A) = \max_{\lambda \in \sigma(A)} |\lambda|$$

Types of matrices

- *Symmetric matrix*: $A^T = A$.
- *Hermitian matrix*: $A^H = A$. Where $A^H = \bar{A}^T = \overline{A^T}$.
- *Diagonal matrix*: $a_{ij} = 0$ for $j \neq i$. Notation: $A = \text{diag}(a_{11}, a_{22}, \dots, a_{nn})$.
- *Upper triangular matrix*: $a_{jj} = 0$ for $i > j$.
- *Lower triangular matrix*: $a_{jj} = 0$ for $i < j$.
- *Tridiagonal matrix*: $a_{ij} = 0$ for any pair i, j such that $|j - i| > 1$.
  Notation: $A = \text{tridiag}(a_{i,i-1}, a_{ii}, a_{i,i+1})$.
- *Column Permutation matrix*: the columns of A are a permutation of the columns of the identity matrix.
- *Block diagonal matrix*: generalizes the diagonal matrix by replacing each diagonal entry by a matrix. Notation: $A = \text{diag}(A_{11}, A_{22}, \dots, A_{nn})$.
- *Block tridiagonal matrix*: generalizes the tridiagonal matrix by replacing each nonzero entry by a square matrix. Notation: $A = \text{tridiag}(A_{i,i-1}, A_{ii}, A_{i,i+1})$.
- *Positive Definite*: $(Ax, x) > 0, \quad \forall x \in \mathbb{C}^n, \ x \neq 0$. Defined the same way for negative definite.
- *Positive semidefinite*: $(Ax, x) \geq 0, \quad \forall x \in \mathbb{C}^n, \ x \neq 0$. The same for negative semidefinite.
- *Reducible*: if there is a permutation matrix P such that PAP^T is block upper triangular. Otherwise, it is *irreducible*.

Matrix norms

For a general matrix A in $\mathbb{C}^{n \times m}$, we define the following special set of norms

$$\|A\|_p = \max_{x \in \mathbb{C}^m, x \neq 0} \frac{\|Ax\|_p}{\|A\|_p} = \max_{\|x\|_p = 1} \|Ax\|_p \qquad p \geq 1.$$

2.2 Orthogonal vectors

A set of vectors $G = \{a_1, a_2, \dots, a_r\}$ is said to be orthogonal if $(a_i, a_j) = 0$ when $i \neq j$. Where $(.,.)$ is an inner product. The set is called orthonormal if, in addition: $\|a_i\|_2 = 1, \quad i = 1, \dots, r$.

Every subspace admits an orthonormal basis which is obtained by taking any basis and "orthonormalizing" it. Given a linearly independent set of vectors $\{x_1, x_2, \dots, x_r\}$, the orthonormalization can be achieved by an algorithm known as the Gram-Schmidt procedure. This procedure consists of taking the first vector x_1, normalizing it to get q_1, and orthogonalize the vector x_2 against it. Then the second vector is normalized to q_2, then x_3 is orthogonalized against q_1 and q_2. This gives the following algorithm:

Algorithm 1: Gram Schmidt

1. Compute $r_{11} := \|x_1\|_2$. If $r_{11} = 0$ Stop, else compute $q_1 := \frac{x_1}{r_{11}}$.
2. For $j = 2, \dots, r$ Do:
3. Compute $r_{ij} := (x_j, q_i)$, for $i = 1, 2, \dots, j-1$
4. $\hat{q} := x_j - \sum_{i=1}^{j-1} r_{ij} \, q_i$
5. $r_{jj} := \|\hat{q}\|_2$,

6. If $r_{jj} = 0$ Stop, else $q_j := \frac{\hat{q}}{r_{jj}}$
7. End Do

This is the standard Gram-Schmidt process. There are alternative formulations of the procedure that have better numerical properties. The best known is the Modified Gram Schmidt, which can be found in Saad, Section 1.7, Algorithm 1.2.

Chapter 3 Basic Iterative Methods

Let

$$Au = b \quad (1)$$

be a linear system with an invertible matrix A.

Iterative methods of linear systems of equations are useful, if the number of iterations necessary is not too big, if A is sparse, or A has a special structure, or if a good guess for u is available.

We use the following general iteration:

$$u_{i+1} = Qu_i + s, \quad (i = 0,1,2,\ldots) \quad (2)$$

so that the system $u = Qu + s$ is equivalent to the original problem. Here Q is the *iteration matrix*. It is, however, not computed explicitly. It is possible to rewrite the process (2) as follows:

$$u_{i+1} = Qu_i + s$$

$$u_{i+1} - Qu_i = s$$

Then let $i \to \infty$

$$(I - Q)u = s$$

$$u = (I - Q)^{-1}s = A^{-1}b$$

The iterative process (2) is called *consistent*, if matrix $I - Q$ is not singular and if $(I - Q)^{-1}s = A^{-1}b$. If the iteration is consistent, the equation $(I - Q)^{-1}u = s$ has exactly one solution $u_{i\to\infty} \equiv u$.

The simplest iterative scheme is the *Richardson iteration*

$$u_{i+1} = u_i + \tau(b - Au_i) = (I - \tau A)u_i + \tau b \qquad (i = 0,1,\ldots)$$

with some acceleration parameter $\tau \neq 0$. In this case, $Q = I - \tau A$ and $s = \tau b$ in (1).

A usual way of constructing Q is to start with a *splitting*

$$A = \hat{A} - R$$

and to use the iteration

$$\hat{A}u_{i+1} = Ru_i + b.$$

Here

$$Q = (\hat{A})^{-1}R = I - (\hat{A})^{-1}A.$$

3.1 Jacobi and Gauss- Seidel

Two well-known examples for a splitting are the Jacobi and the Gauss-Seidel method.

Let $A \in \mathbb{R}^{N\times N}$ be a regular matrix, $A=(a_{mn}),\ m,n=1,\dots N$ with $a_{mm}\neq 0\ (m=1,2,\dots,N)$, $b \in \mathbb{R}^N$.

In the Jacobi method, we compute, starting with initial guess u_0 recursively the components of the (i+1)-th vector are computed the following way:

$$u_m^{\,i+1}=\frac{1}{a_{mm}}\left(b_m-\sum_{k=1(k\neq m)}^{N} a_{mn}u^i{}_k\right),\quad (m=1,\dots,N).$$

In Gauss-Seidel's method one uses, different from Jacobi's method, during the computation of $u_m^{\,i+1}$ the components computed previously in this iteration $u_1^{\,i+1},\dots,u_{m-1}^{\,i+1}$ instead of $u_1^{\,i},\dots,u_{m-1}^{\,i}$. Herewith, the approximations u^i should reach an exact solution u faster, if the process converges at all,

$$u_m^{\,i+1}=\frac{1}{a_{mm}}\left(b_m-\sum_{k=1}^{m-1}a_{mk}u^{i+1}{}_k-\sum_{k=m+1}^{N}a_{mk}u^i{}_k\right),\quad (m=1,\dots,N)$$

The iteration matrix Q for Jacobi is found by splitting A into the following form $A=D-C$, with a diagonal matrix D, which is equal to the main diagonal of A.

Here,

$$Q_{JAC}=D^{-1}C, s=D^{-1}b$$

In the Gauss-Seidel method, C is split into: $C=C_1+C_2$, with a lower triangular matrix C_1 and an upper triangular matrix C_2. Then the iteration matrix Q_{GS} for Gauss-Seidel becomes:

$$Q_{GS}=\left(D-C_1\right)^{-1}C_2,\ \ s=(D-C_1)^{-1}b.$$

Overrelaxation is based on the splitting

$$\omega A=(D-\omega C_1)-(\omega C_2+(1-\omega)D)$$

And the corresponding Successive Over Relaxation (SOR) method is given by the recursion

$$(D-\omega C_1)u_{i+1}=\left[\omega C_2+(1-\omega)D\right]u_i+\omega b$$

Note that when $\omega = 1$ we get Gauss-Seidel.

3.2 Convergence results for Jacobi and Gauss-Seidel

An iteration is called *convergent*, if the sequence $u_0,u_1,u_2,\dots$ converges to a limit u.

The iterative method $u_{i+1} = Qu_i + s$ $(i = 0,1,\ldots)$ converges if $\rho(Q) < 1$ [see Saad, Theorem 4.1, p. 104]. Here $\rho(Q)$ is the spectral radius of Q.

Since it is very expensive to compute the spectral radius of a matrix, sufficient conditions that guarantee convergence can be useful in practice. One such sufficient condition could be obtained by utilizing the inequality, $\rho(Q) \leq \|Q\|$ for any matrix norm. Thus for convergence, it is sufficient to have $\|Q\| < 1$.

Here we state the most important theorems about convergence for Jacobi, Gauss-Seidel and SOR.

Definition 1
Let A, M, N be three given matrices satisfying $A = M - N$. The pair of matrices M, N is a *regular splitting* of A, if M is nonsingular and M^{-1} and N are nonnegative.

We associate a regular splitting with the above splitting for Jacobi and Gauss-Seidel.

Theorem 1
Let M, N be a regular splitting of a matrix A. Then $\rho(M^{-1}N) < 1$ if and only if A is nonsingular and A^{-1} is nonnegative.

Proof: see Saad, p. 107-108

Definition 2
A matrix A is *strictly column diagonally dominant* if

$$|a_{jj}| > \sum_{\substack{i=1 \\ i \neq j}}^{i=n} |a_{ij}|, \quad j = 1, \ldots, n$$

A is *irreducibly diagonally dominant* if A is irreducible, and

$$|a_{jj}| \geq \sum_{\substack{i=1 \\ i \neq j}}^{i=n} |a_{ij}|, \quad j = 1, \ldots, n$$

with strict inequality for at least one j.

Theorem 2
If A is a strictly diagonally dominant or an irreducibly diagonally dominant matrix, then the associated Jacobi and Gauss-Seidel iterations converge for any u_0.

Proof: see Saad, p. 111

Theorem 3
If A is symmetric with positive diagonal elements and for $0 < \omega < 2$, SOR converges for any u_0 if and only if A is positive definite.

Stopping criteria

An iterative method should be stopped if the approximate solution is accurate enough. A good termination criterion is very important for an iterative method, because if the criterion is too weak, the approximate solution is useless, whereas if the condition is too severe the iterative method never stops or costs too much work.

For iterative methods that have a super linear convergence behavior most stopping criteria are based on the norm of the *residual* $r_i = b - Au_i$. Some examples of these stopping criteria are:

1) $\|b - Au_i\|_2 \leq \varepsilon$
2) $\frac{\|b-Au_i\|_2}{\|b-Au_0\|_2} \leq \varepsilon$
3) $\frac{\|b-Au_i\|_2}{\|b\|_2} \leq \varepsilon$
4) $\frac{\|b-Au_i\|_2}{\|u_i\|_2} \leq \varepsilon / \|A^{-1}\|_2$

The main disadvantage of the first one is that it is not scaling invariant. This implies that if $\|b - Au_i\|_2 \leq \varepsilon$ this does not hold for $\|100(b - Au_i)\|_2$. The second criterion is widely used, but may never be satisfied if the initial estimate is very good due to round off errors. The third criterion is a good one, while the fourth is in many cases useless since in general$\|A\|_2$ and $\|A^{-1}\|_2$ are not known.

Chapter 4 Krylov subspace methods

Recall the linear system:

$$Au = b$$

Define the subspace:

$$K_m(A, r_0) = span\{r_0, Ar_0, A^2 r_0, \dots, A^{m-1} r_0\}$$

Where $r_i = b - Au_i$ is called the *residual.* $K_m(A, r_0)$is called the *Krylov subspace* of dimension m corresponding to matrix A and initial residual r_0.

Solutions computed by basic iterative methods follow the recusion

$$u_{i+1} = u_i + B^{-1}(b - Au_i) = u_i + B^{-1} r_i$$

The generated u_i's are elements of $u_0 + K_m(B^{-1}A, B^{-1} r_0)$.

4.1 Arnoldi's Method

Arnoldi's procedure is an algorithm for building an orthogonal basis of the Krylov subspace K_m. In the algorithm, the vectors are orthogonalised by a standard Gram-Schmidt procedure. In exact arithmetic, one variant of the algorithm is as follows:

Algorithm 2: Arnoldi

1. Choose a vector v_1 such that $\|v_1\|_2 = 1$
2. For $j = 1,2, \dots, m$ Do:
3. Compute $h_{ij} = (Av_j, v_i)$ for $i = 1,2, \dots, m$
4. Compute $w_j := Av_j - \sum_{i=1}^{j} h_{ij} v_i$
5. $h_{j+1,j} = \|w_j\|_2$
6. If $h_{j+1,j} = 0$ then stop
7. $v_{j+1} = w_j / h_{j+1,j}$
8. EndDo

With the Modified Gram-Schmidt alternative the algorithm takes the following form:

Algortihm 3: Arnoldi- Modified Gram-Schmidt

1. Choose a vector v_1 of norm 1
2. For $j = 1,2, \dots, m$ Do:
3. Compute $w_j := Av_j$
4. For $i = 1,2, \dots, j$ Do:
5. $h_{ij} = (w_j, v_i)$
6. $w_j := w_j - h_{ij} v_i$
7. EndDo
8. $h_{j+1,j} = \|w_j\|_2$. If $h_{j+1,j} = 0$ Stop
9. $v_{j+1} = w_j / h_{j+1,j}$

10. EndDo

In practice, much can be gained by using Modified Gram-Schmidt or the Householder algorithm, which is from numerical point of view one of the most reliable orthogonalization techniques. For more details see Saad, page 149. Although the Householder algorithm is numerically more viable than Gram-Schmidt and Modified Gram-Schmidt, it is also more expensive.

When solving linear systems, the Modified Gram-Schmidt orthogonalization with a reorthogonalization strategy based on a measure of the level of cancellation [see Saad, page 148] is more than adequate in most cases.

4.2 The Symmetric Lanczos Algorithm

The symmetric Lanczos Algorithm can be viewed as a simplification of Arnoldi's method for the particular case when the matrix A is symmetric.

Denote by $\bar{\bar{H}}_m$ the (m+1) × m Hessenberg matrix whose nonzero entries h_{ij} are defined by Algorithm 2, and by H_m the matrix obtained from $\bar{H}_m$ by deleting its last row.

Theorem 4

Assume that Arnoldi's algorithm is applied to a real symmetric matrix A. Then the coefficients h_{ij} generated by the algorithm are such that:

$$h_{ij} = 0 \quad \text{for } 1 \leq i \leq j-1, \text{ and}$$

$$h_{j,j+1} = h_{j+1,j}, \qquad j = 1,2,\dots,m$$

In other words, the matrix H_m obtained from the Arnoldi process is tridiagonal and symmetric.

Proof: see Saad, page 173

The standard notation used to describe the Lanczos algorithm is obtained by setting

$$\alpha_j \equiv h_{ij}\ , \qquad \beta_j \equiv h_{j-1,j}\ .$$

And if T_m denotes the resulting H_m matrix, it is of the form,

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & \cdot \\ & & \beta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & \beta_m & \alpha_m \end{pmatrix}$$

This leads to the following form of the Modified Gram-Schmidt variant of Algorithm 3.

Algorithm 4: The Lanczos Algorithm

1. Choose an initial vector v_1 of norm unity. Set $\beta_1 \equiv 0, v_0 \equiv 0$
2. For $j = 1,2,\dots,m$Do:

3. $w_j := Av_j - \beta_j v_{j-1}$
4. $\alpha := (w_j, v_j)$
5. $w_j := w_j - \alpha_j v_j$
6. $\beta_{j+1} = \|w_j\|_2$. If $\beta_{j+1} = 0$ Stop
7. $v_{j+1} = w_j / \beta_{j+1}$
8. EndDo

In reality, exact orthogonality of these vectors is only observed at the beginning of the process. At some point the v_i's start losing their orthogonality rapidly.

The major differences with Arnoldi's method are that the matrix H_m is tridiagonal, and thus only three vectors must be stored in memory.

4.3 The Conjugate Gradient Algorithm

The Conjugate Gradient Algorithm is one of the best known iterative techniques for solving sparse Symmetric Positive Definite linear systems. Described in one sentence, the method is a realization of an orthogonal projection technique onto the Krylov subspace $K_m(A, r_0)$ where r_0 is the initial residual. Because A is symmetric, some simplifications resulting from the Lanczos recurrence will lead to more elegant results.

Given an initial guess u_0 to the linear system $Au = b$ and the Lanczos vectors $v_1 = \frac{r_0}{\|r_0\|_2}$, v_i, $i = 2, \ldots, m$ together with the tridiagonal matrix T_m, the approximate solution obtained from an orthogonal projection method onto K_m, is given by

$$u_m = u_0 + V_m y_m, \quad y_m = T_m^{-1}(\beta e_1)$$

Where V_m is the $n \times m$ matrix with column vectors $v_1, \ldots, v_m$, and $\beta = \|r_0\|_2$.

First write the LU factorization of T_m as $T_m = L_m U_m$. This factorization is of the form:

$$T_m = \begin{pmatrix} 1 & & & & \\ \lambda_2 & 1 & & & \\ & \ddots & \ddots & & \\ & & \ddots & \ddots & \\ & & \lambda_{m-1} & 1 & \\ & & & \lambda_m & 1 \end{pmatrix} \times \begin{pmatrix} \eta_1 & \beta_2 & & & \\ & \eta_2 & \beta_3 & & \\ & & \ddots & \ddots & \\ & & & \ddots & \\ & & & \eta_{m-1} & \beta_m \\ & & & & \eta_m \end{pmatrix}$$

Then it can be shown that u_m can be updated at each step as

$$u_m = u_{m-1} + \zeta_m p_m$$

where $\zeta_m = -\lambda_m \zeta_m$, $\zeta_1 = \|r_0\|_2$ and $p_m = \eta_m^{-1}[v_m - \beta_m p_{m-1}]$, $p_0 = 0$.

This gives rise to the direct version of the Lanczos algorithm for linear systems. This algorithm computes the solution of the tridiagonal system $T_m y_m = \beta e_1$ progressively by using Gaussian elimination without pivoting, which may be more prone to breakdown than the version with partial pivoting. For more details, see Saad, page 175 – 177.

The following proposition tells us that the residual vector for these algorithms is in the direction of v_{m+1}, and also that the vectors p_i are *A-orthogonal*, or *conjugate*.

Proposition 1

Let $r_m = b - Avu_m$, $m = 0, 1, 2, \dots$, be the residual vectors produced by the Lanczos and the Direct Lanczos algorithms and p_m, $m = 0, 1, 2, \dots$, the auxiliary vectors produced by the Direct Lanczos Algorithm. Then,

1. *Each residual vector r_m is such that $r_m = \sigma_m v_{m+1}$ where σ_m is a certain scalar. As a result the residual vectors are orthogonal to eacht other.*
2. *The auxiliary vectors p_i form an A-conjugate set, i.e., $(Ap_i, p_j) = 0$, for $i \neq j$.*

Proof: see Saad, page 177.

A consequence of the above proposition is that a version of the algorithm can be derived by imposing the orthogonality and conjugacy conditions. This gives the Conjugate Gradient algorithm. For the detailed derivation see Saad, page 177-178.

Algorithm 5: Conjugate Gradient

1. Compute $r_0 := b - Ax_0$, $p_0 := r_0$.
2. For $j = 0, 1, \dots$, until convergence Do:
3. $\quad \alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4. $\quad x_{j+1} := x_j + \alpha_j p_j$
5. $\quad r_{j+1} := r_j - \alpha_j Ap_j$
6. $\quad \beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7. $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
8. EndDo

It is important to note that the scalars α_j, β_j in this algorithm are different from those of the Lanczos algorithm. The vectors p_j are multiples of the p_j's of the Direct Lanczos.

In terms of storage, matrix A and the vectors x, p, Ap, and r must be stored.

Chapter 5 Preconditioning Techniques

Lack of robustness is a widely recognized weakness of iterative solvers, relative to direct solvers. This drawback hampers the acceptance of this type of methods in industrial applications despite their intrinsic appeal for very large linear systems. Both efficiency and robustness of iterative techniques can be improved by using preconditioning.

Preconditioning is a simple means of transforming the original linear system into one which had the same solution, but which is likely to be easier to solve with an iterative solver. When dealing with various applications, the reliability of iterative techniques depends much more on the quality of the preconditioner than on the particular Krylov subspace accelerators used.

First we will discuss the preconditioned version of the Conjugate Gradient algorithm without being specific about the particular preconditioner used. Later on we will discuss standard preconditioning techniques.

5.1 Preconditioned conjugate gradient method

Consider a matrix *A* that is Symmetric and Positive Definite (SPD) and assume that a preconditioner M is available. M approximates A in some sense, and is also SPD. Because the preconditioned algorithms will all require a linear system solution with the matrix M at each step, the only requirement on M is that it is inexpensive to solve the linear system $Mx = b$. Then, for example the following preconditioned system could be solved:

$$M^{-1}Ax = M^{-1}b$$

or

$$AM^{-1}u = b,\ x = M^{-1}u$$

Note that these two systems are no longer symmetric in general. The question here is, how to preserve symmetry?

One method to preserve the symmetry of the original system, is to have M available in the form of an incomplete Cholesky factorization, i.e., when $M = LL^T$. The splitting the preconditioner between left and right, and solve:

$$L^{-1}AL^{-T}u = L^{-1}b,\quad x = L^{-T}u$$

which involves a SPD matrix.

Define the *M-inner product*: $(x, y)_M \equiv (Mx, y) = (x, My)$.

An alternative method is to replace the usual Euclidian inner product in the Conjugate Gradient algorithm by the M-inner product or the A-inner product, since the matrix $M^{-1}A$ is self-adjoint for both the M-inner product and A-inner product:

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M(M^{-1}A)y) = (x, M^{-1}Ay)_M$$

It can be shown that the iterates are identical. For more details see Saad, page 245 – 247.

The Preconditioned Conjugate Gradient algorithm becomes as follows:

Algorithm 6: Precondtioned Conjugate Gradient

1. Compute $r_0 := b - Ax_0, z_0 = M^{-1}r_0,\ p_0 := z_0$.
2. For $j = 0, 1, \dots$, until convergence Do:
3. $\alpha_j := (r_j, z_j)/(Ap_j, p_j)$
4. $x_{j+1} := x_j + \alpha_j p_j$
5. $r_{j+1} := r_j - \alpha_j Ap_j$
6. $z_{j+1} := M^{-1}r_{j+1}$
7. $\beta_j := (r_{j+1}, z_{j+1})/(r_j, z_j)$
8. $p_{j+1} := z_{j+1} + \beta_j p_j$
9. EndDo

5.2 Preconditioning techniques

Consider the *preconditioned system* associated with the splitting $A = M - N$

$$M^{-1}Ax = M^{-1}b \quad (3)$$

and the iteration of the form:

$$x_{k+1} = Gx_k + f$$

Where $f = M^{-1}b$ and $G = I - M^{-1}A$ (see Chapter 3).

In theory, any general splitting in which M is non-singular can be used. Ideally, M should be close to A in some sense. Because a linear system with the matrix M must be solved at each step of the iterative procedure, a practical requirement is that these solution steps should be inexpensive. In general, a Krylov subspace method can be used to solve a preconditioned system like (3).

One of the simplest ways of defining a preconditioner is to perform an incomplete factorization of the original matrix A. This entails a decomposition of the form $A = LU - R$ where L and U have the same nonzero structure as the lower and upper parts of A respectively, and R is the *residual* or *error* of the factorization. This incomplete factorization is known as ILU(0). Next we will discuss the general class of incomplete factorization techniques which are discussed in the next section.

5.3 ILU factorization preconditioners

A general Incomplete LU (ILU) factorization process computes a sparse lower triangular matrix L and an upper triangular matrix U such that the residual $R = LU - A$ satisfies certain constraints, such as having zero entries in some locations. Such an algorithm can be derived by performing Gaussian elimination and dropping some elements in predetermined nondiagonal positions.

Theorem 5

Let A be an M-matrix and let A_1 be the matrix obtained from the first step of Gaussian elimination. Then A_1 is an M-matrix.

Proof: see Saad, page 269.

The elements to drop can be specified by choosing some non-zero pattern in advance. The only restriction on the zero pattern is that it should exclude diagonal elements because this assumption was used in the proof of Theorem 5.

Hence, for any zero pattern P, such that

$$P \subset \{(i,j) |\, i \neq j; i,j \leq n\} \qquad (4)$$

an ILU factorization, ILU_P can be computed as follows.

Algorithm 7: General Static Pattern ILU, KIJ Version

1. For $k = 1, \ldots, n-1$ Do
2. For $i = k+1, \ldots, n$ and if $(i,k) \notin P$ Do
3. $a_{ik} := a_{ik}/a_{kk}$
4. For $j = k+1, \ldots, n$ and for $(i,j) \notin P$ Do
5. $a_{ij} := a_{ij} - a_{ik}a_{kj}$
6. End Do
7. End Do
8. End Do

The following result can be proved:

Theorem 6

Let A be an M-matrix and P a given zero pattern defined as in (4). Then Algorithm 7 is feasible and produces an incomplete factorization,

$$A = LU - R$$

which is a regular splitting of A.

Proof: see Saad, page 270.

A more practical version of this algorithm but an equivalent one is the next:

Algorithm 8 General ILU factorization, IKJ Version

1. For $i = 2, \ldots, n$ Do:
2. For $k = 1, \ldots, i-1$ and if $(i,k) \notin P$, Do:
3. $a_{ik} := a_{ik}/a_{kk}$
4. For $j = k+1, \ldots, n$ and for $(i,j) \notin P$, Do:
5. $a_{ij} := a_{ij} - a_{ik}a_{kj}$
6. End Do
7. End Do
8. End Do

The following proposition assures us that Algorithm 7 and 8 are indeed equivalent:

Proposition 2

Let P be a zero pattern satisfying the condition (4). Then the ILU factors by the KIJ-based Algorithm 7 and the IKJ-based Algorithm 8 are identical if they can both be computed.

Proof: see Saad, page 272.

Above proposition is only true for static pattern ILU. If the pattern is dynamically determined as the Gaussian elimination algorithm proceeds, the patterns obtained with different versions of Gaussian elimination can be different.

Our incomplete factorization is about $A = LU - R$. The following proposition describes how matrix R looks like.

Proposition 3

Algorithm 8 produces factors L and U such that $A = LU - R$ in which $-R$ is the matrix of the elements that are dropped during the incomplete elimination process. When $(i, j) \in P$, an entry r_{ij} of R is equal to the value of $-a_{ij}$ obtained at the completion of the k loop in Algorithm 8. Otherwise, $r_{ij} = 0$.

Zero Fill-in (ILU(0))

The incomplete LU factorization technique with no fill-in, denoted by ILU(0), consists of taking the zero pattern P to be precisely the zero pattern of A. in general terms: construct any pair of matrices L and U so that the elements of A - LU are zero in the locations of the nonzero elements of A ($NZ(A)$).

The standard ILU(0) is defined constructively using Algorithm 8 with P the pattern equal to the zero pattern of A.

5.4 Incomplete Cholesky Conjugate Gradient

A specific form of an incomplete factorization with no fill-in elements is the Incomplete Cholesky factorization. Combined with the Preconditioned Conjugate Gradient Algorithm we get ICCG(0). A more detailed description of this algorithm is found in [2], p. 68-70.

When calculating the Cholesky factorization, the zero elements in de band of A become non zero elements in the band of L. These elements are called *fill-in elements*. The Cholesky factorization used for preconditioning is called incomplete because these fill-in elements are discarded.

Define the set of all pairs of off-diagonal matrix entries denoted by:

$$Q_N = \{(i,j) | i \neq j, 1 \leq i \leq N, 1 \leq j \leq N\}$$

where N denotes the dimension of matrix A: $A \in \mathbb{R}^{N \times N}$.

Then the subset Q of Q_N are the places (i, j) where L should be zero. Knowing this, the following theorem can be proved:

Theorem 7

If A is a symmetric M-matrix, there exists for each $Q \subset Q_N$ (with the property that (i, j) implies $(j, i) \in Q$), a uniquely defined lower triangular matrix L and a symmetric nonnegative matrix R with $l_{ij} = 0$ if $(i, j) \in Q$ and $r_{ij} = 0$ if $(i, j) \notin Q$, such that the splitting $A = LL^T - R$ leads to a

convergent iterative process

$$LL^T u^{i+1} = Ru^i + b \ \textit{for each choice } u^0 \ ,$$

where $u^i \to u = A^{-1}b$.

Proof: see[2]; p 69.

After the matrix L is constructed, it is used in Algorithm 2. When in this algorithm multiplications with L^{-1} and L^{-T} are necessary, this is never done by forming L^{-1} or L^{-T}, since these are full matrices. Since *L* is a lower triangular matrix, it is easy to see that solving the linear system $Lz = r$ is cheaper than calculating $z = L^{-1}r$.

In this case we compute a slightly adapted incomplete Cholesky factorization which is cheaper because less memory is used. This variant is mathematically equivalent to the described one.

Consider the following decomposition:

$$A = LD^{-1}L^T - R$$

where the elements of the lower triangular matrix L and diagonal matrix D satisfy the following rules:

1) $l_{ij} = 0$ for all (i,j) where $a_{ij} = 0 \ i > j$,
2) $l_{ii} = d_{ii}$,
3) $(LD^{-1}L^T)_{ij} = a_{ij}$ for all (i,j) where $a_{ij} \neq 0 \ i \geq j$.

Let m be the number of grid points in x-direction. Then A has the following form:

$$A = \begin{pmatrix} a_1 & b_1 & & c_1 & & & & \\ b_1 & a_2 & b_2 & & c_2 & & & \\ & \ddots & \ddots & \ddots & & \ddots & & \\ c_1 & & b_m & a_{m+1} & b_{m+1} & & c_{m+1} & \\ & \ddots & & \ddots & \ddots & \ddots & & \ddots \\ & & & & & & & \end{pmatrix}$$

If the elements of L are given as follows:

$$L = \begin{pmatrix} \tilde{d}_1 & & & & & \\ \tilde{b}_1 & \tilde{d}_2 & & & \emptyset & \\ & \ddots & \ddots & & & \\ \tilde{c}_1 & & \tilde{b}_m & \tilde{d}_{m+1} & & \\ & \ddots & & \ddots & \ddots & \\ & & & & & \end{pmatrix}$$

then the elements of L can be calculated by the following expressions:

$$\left.\begin{aligned} \tilde{d}_i &= a_i - \frac{b_{i-1}^2}{\tilde{d}_{i-1}} - \frac{b_{i-m}^2}{\tilde{d}_{i-m}} \\ \tilde{b}_i &= b_i \\ \tilde{c}_i &= c_i \end{aligned}\right\} \ i = 1, \dots, N.$$

Elements that are not defined are replaced by zeros.

Using this Cholesky decomposition as preconditioner M in the Preconditioned Conjugate Gradient algorithm is called the Incomplete Cholesky Conjugate Gradient (0) (ICCG(0)). The 0 stands for zero extra diagonals allowed with fill-in.

5.4 Level of fill in and ILU(P)

The accuracy of the ILU(0) incomplete factorization may be insufficient to yield an adequate rate of convergence. More accurate incomplete LU factorizations are often more efficient as well as more reliable. These factorizations will differ from ILU(0) by allowing some fill-in. For example, ILU(1) results from taking *P* to be the zero pattern of the product LU of the factors L, U obtained from ILU(0).

This construction can be extended to general sparse matrices by introducing the concept of *level of fill*. A level of fill is attributed to each element that is processed by Gaussian elimination and dropping will be based on the value of the level of fill.

Definition 3

The *initial level of fill* of an element a_{ij} of a sparse matrix A is defined by

$$\text{lev}_{ij} = \begin{cases} 0; & \text{if } a_{ij} \neq 0, \text{or } i = j \\ \infty; & \text{otherwise.} \end{cases}$$

Each time this element is modified in line 5 of Algorithm 8, its level of fill must be updated by

$$\text{lev}_{ij} = \min\{\text{lev}_{ij}, \text{lev}_{ik} + \text{lev}_{kj} + 1\} \quad (5)$$

Using the definition of zero patterns introduces earlier, the zero pattern for ILU(*p*) is the set

$$P_p = \{(i,j) |\, lev_{ij} > p\}$$

where lev_{ij} is the level of fill value after all updates (5) have been performed.

Here follows a practical implementation of the ILU(*p*) factorization.

Algorithm 9: ILU(*p*)

1. For all nonzero elements a_{ij} define $lev(a_{ij}) = 0$
2. For $i = 2, \dots, n$, Do:
3. For each $k = 1, \dots, i-1$ and for $lev(a_{ik}) \leq p$, Do
4. Compute $a_{ik} := a_{ik}/a_{kk}$
5. Compute $a_{i*} := a_{i*} - a_{ik} a_{k*}$
6. Update the levels of fill of the nonzero a_{ij}'s using (5)
7. End Do
8. For any $a_{ij} \in a_{i*}$ with $lev(a_{ij}) > p$: $a_{ij} = 0$
9. End Do

5.5 Threshold strategies and ILUT

Incomplete factorizations which rely on level of fill do not take into account the numerical value of the elements that are dropped, since the structure of A is only what matters. This can cause some difficulties for realistic problems that arise in many applications. There are a few methods available which do drop elements according to their magnitude rather than their locations. With these

techniques the zero pattern P is determined dynamically. The following algorithm describes the ILUT approach. ILU(0) is a special case of this strategy.

Algortihm 10: ILUT

1. For $i = 2, \dots, n$, Do
2. $w := a_{i*}$
3. For $k = 1, \dots, i-1$ and when $w_k \neq 0$, Do
4. $w_k := w_k / a_{kk}$
5. If 'dropping rule' satisfied set $w_k = 0$
6. If $w_k \neq 0$ then
7. $w := w - w_k * u_{k*}$
8. End If
9. End Do
10. Apply a 'dropping rule' to row w
11. $l_{ij} := w_j$ for $j = 1, \dots, i-1$
12. $u_{ij} := w_j$ for $j = i, \dots, n$
13. $w := 0$
14. End Do

In the factorization ILUT(p, τ) the follwing dropping rules are used:

- In line 5: $w_k = 0$ if $w_k < \tau_i$, where $\tau_i = \tau \|a_{i*}\|_2$
- In line 10: First apply the first dropping rule, then keep only the p largest elements in the L part of the row and the p largest elements in the U part of the row in addition to the diagonal element, which is always kept.

Roughly speaking, p can be viewed as a parameter that helps control memory usage, while τ helps to reduce computational costs.

5.6 Approximate inverse preconditioners

Consider the following incomplete LU factorization of sparse matrix A:

$$A = LU + E$$

where E is the error. The preconditioned matrices associated with the different forms of preconditioning are similar to

$$L^{-1}AU^{-1} = I + L^{-1}EU^{-1}$$

When the matrix A is diagonally dominant, then L and U are well-conditioned and the size of $L^{-1}EU^{-1}$ remains confined within reasonable limits. Its eigenvalues are nicely clustered around the origin.

When the original matrix A is not diagonally dominant, L^{-1} and U^{-1} may have very large norms, causing the error $L^{-1}EU^{-1}$ to be very large and thus adding large perturbations to the identity matrix.

This can be remedied by trying to find a preconditioner that does not require solving a linear system. For example, preconditioning the original system by a matrix M which is a direct approximation to the inverse of A.

This can be translated to the minimizing problem:

Find M that minimizes:

$$F(M) = \|I - AM\|_F^2 = \sum_{j=1}^{n} \left\|e_j - Am_j\right\|_2^2$$

where $\|.\|_F$ is the *Frobenius norm* defined as:

$$\|A\|_F^2 = \sum_{i=1}^{m}\sum_{j=1}^{n} \left|a_{ij}\right|^2$$

in which A is a m x n-matrix, and e_j and m_j are the j-th columns of the identity matrix and of the matrix M.

Finding M can be done in two different ways:

1) Minimizing the function $F(M) = \|I - AM\|_F^2$ globally as a function of a sparse matrix, e.g. by a gradient-type method.
2) Minimizing the individual functions $\left\|e_j - Am_j\right\|_2^2$, $j = 1, 2, \ldots, n$.

Although the second approach is more appealing to parallel computers, parallelism can also be exploited in the first one. For more details on the solutions, see Saad, p.298-301.

5.7 Block preconditioners

For block-tridiagonal matrices arising from the discretization of elliptic problems, block preconditioning is a popular technique.

Consider a block-tridiagonal matrix blocked in the form

$$A = \begin{pmatrix} D_1 & E_2 & & & \\ F_2 & D_2 & E_3 & & \\ & \ddots & \ddots & \ddots & \\ & & F_{m-1} & D_{m-1} & E_m \\ & & & F_m & D_m \end{pmatrix}$$

Let D be the block-diagonal matrix consisting of the diagonal blocks D_i, L the block strictly-lower triangular matrix consisting of the sub-diagonal blocks F_i, and U the block strictly-upper triangular matrix consisting of the super-diagonal blocks E_i. Then:

$$A = L + D + U.$$

A Block ILU preconditioner is defined by

$$M = (L + \Delta)\Delta^{-1}(\Delta + U),$$

where L and U are the same as above, and Δ is a block diagonal matrix whose blocks Δ_i are defined by the recurrence:

$$\Delta_i = D_i - F_i \Omega_{i-1} E_i,$$

in which Ω_j is some sparse approximation to Δ_j^{-1}, since using explicit inverses of the block diagonal matrix can cause some difficulties because the diagonal structure is lost.

An important particular case is when the diagonal blocks D_i of the original matrix are tridiagonal, while the co-diagonal blocks E_i and F_i are diagonal. Then, only the tridiagonal part of the inverse must be kept in the recurrence above:

$$\begin{aligned} \Delta_1 &= D_1, \\ \Delta_i &= D_i - F_i \Omega_{i-1}^{(3)} E_i, \quad i = 1, \ldots, m, \end{aligned}$$

where $\Omega_k^{(3)} = (\Delta_k^{-1})_{i,j} \;\; \text{for } |i-j| \leq 1$ is the tridiagonal part of Δ_k^{-1}.

The inverse of a tridiagonal matrix can easily be obtained via the Cholesky factorization. Let Δ be a tridiagonal matrix of dimension l in the form:

$$\Delta = \begin{pmatrix} \alpha_1 & -\beta_2 & & & \\ -\beta_2 & \alpha_2 & -\beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & -\beta_{l-1} & \alpha_{l-1} & -\beta_l \\ & & & -\beta_l & \alpha_l \end{pmatrix}$$

With its Cholesky factorization: $\Delta = LDL^T$, with $D = \text{diag}\{\delta_i\}$ and

$$L = \begin{pmatrix} 1 & & & & \\ -\gamma_2 & 1 & & & \\ & \ddots & \ddots & & \\ & & -\gamma_{l-1} & 1 & \\ & & & -\gamma_l & 1 \end{pmatrix}$$

The inverse is: $\Delta^{-1} = L^{-T} D^{-1} L^{-1}$.

It can be shown that the columns of L^{-T} can be obtained by the following recurrence:

$$\begin{aligned} c_1 &= e_1 \\ c_j &= e_j + \gamma_j c_{j-1}, \qquad \text{for } j \geq 2 \end{aligned}$$

Then the inverse of Δ becomes:

$$\Delta^{-1} = L^{-T} D^{-1} L^{-1} = \sum_{j=1}^{l} \frac{1}{\delta_j} c_j c_j^T.$$

Chapter 6 Parallel Implementations

Parallel computing is fast becoming an inexpensive alternative to the standard supercomputer approach for solving large scale problems that arise in scientific and engineering applications. Because of the increased importance of three-dimensional models, iterative methods are starting to play a major role in many application areas, since sparse direct methods for solving these problems are associated with high costs. Another advantage of iterative techniques above direct methods is that they are far easier to implement on parallel computers because they only require a small set of computational kernels. Currently, there is a large effort to develop new practical iterative methods that are efficient in parallel environments and are also robust. However, these two requirements seem to be in conflict.

There are two traditional approaches for developing parallel iterative techniques:

1. Extract parallelism whenever possible from standard algorithms. These methods are in general easier to understand since the underlying structure did not change.
2. Develop alternative algorithms which have enhanced parallelism.

First we describe two methods of the first kind. After that we pass to the second approach.

6.1 Multiprocessing and Distributed Computing

The six major forms of parallelism are (1) multiple functional units; (2) pipelining; (3) vector processing; (4) multiple vector pipelines; (5) multiprocessing; and (6) distributed computing. Here we describe the latter two.

A multiprocessor system is a computer, or a set of several computers, consisting of several processing elements (PE's), each consisting of a CPU, a memory, etc. These PE's are connected to one another with some communication medium, either a bus or a multi-storage network. There are numerous of possible configurations, like the Distributed Memory Architectures. A more general form of multiprocessing is Distributed Computing, in which the processors are actually linked by some Local Area Network. In heterogeneous networks of computers, the processors are separated by relatively large distances and that had a negative impact on the performance of distributed applications. This approach is cost effective only for large applications, in which a high volume of computation can be performed before more data has to be exchanged.

Distributed Memory Architectures

A Distributed Memory System consists of a large number of identical processors which have their own memories and which are interconnected in a regular topology where processors are linked to other neighboring processors which in turn are linked to neighboring processors, etc.

In 'Message Passing' models, there is no global synchronization of the given parallel tasks. Instead, computations are data driven because a processor performs a given task only when the operands it requires become available. The programmer must program all the data exchanges explicitly between processors.

Chapter 7 Parallel preconditioners

The methods described in this chapter are suitable when the desired goal is to maximize parallelism. When developing parallel preconditioners, one should be aware that the benefits of increased parallelism are not outweighed by the increased amount of computations. The main goal is to find preconditioning techniques that have a high degree parallelism, as well as good intrinsic qualities.

7.1 Introduction

Specifically for parallel environments, a number of alternative preconditioning techniques is developed. Here we will discuss three types of these techniques. The first and simplest one is the Jacobi or block Jacobi approach. This preconditioner is often not very useful, since the number of iterations of the resulting iteration tends to be much larger than the more standard preconditioners such as ILU. To enhance performance a second level of preconditioning is applied called *polynomial preconditioning*. The second technique discussed is 'Multicoloring'. The idea is to color the nodes such that two adjacent nodes do not have the same color. Doing this, we are able to determine nodes with the same color simultaneously. The third strategy is 'Domain Decomposition' which is a generalization of 'partitioning' techniques.

In the algorithms described, we roughly distinguish two types: those which can be termed coarse-grain and those which can be termed fine-grain. In coarse-grain algorithms, the parallel tasks are relatively big and may involve solution of small linear systems. In fine-grain parallelism, the subtasks can be elementary floating-point operations.

7.2 Block Jacobi

A generalization of the "point" relaxation schemes are the block relaxation schemes which update a subvector containing a set of unknowns in one time step.

The matrix A and the right-hand side and solution vectors are partitioned as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1p} \\ A_{21} & A_{22} & A_{23} & \cdots & A_{2p} \\ A_{31} & A_{32} & A_{33} & \cdots & A_{3p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & A_{p3} & \cdots & A_{pp} \end{pmatrix}, \qquad x = \begin{pmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \\ \vdots \\ \xi_p \end{pmatrix}, \qquad b = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \beta_p \end{pmatrix}$$

in which the partitionings of b and x into subvectors β_i and ξ_i are identical and compatible with the partitioning of A. The diagonal block in A are square and assumed nonsingular.

Generalizing the scalar case, define the splitting:

$$A = D - E - F$$

with

$$D = \begin{pmatrix} A_{11} & & & \\ & A_{22} & & \\ & & \ddots & \\ & & & A_{pp} \end{pmatrix}, E = -\begin{pmatrix} O & & & \\ A_{21} & O & & \\ \vdots & \vdots & \ddots & \\ A_{p1} & A_{p2} & \cdots & O \end{pmatrix}, F = -\begin{pmatrix} O & A_{12} & \cdots & A_{1p} \\ & O & \cdots & A_{2p} \\ & & \ddots & \vdots \\ & & & O \end{pmatrix}$$

Then at each block Jacobi iteration, the updates are defined by

$$\xi_i^{(k+1)} = A_{ii}^{-1}((E+F)x_k)_i + A_{ii}^{-1}\beta_i, \quad i = 1, \dots, p$$

With finite difference approximates of PDE's it is standard to block the variables and the matrix by partitioning along whole lines of the mesh. A block can also correspond with a few consecutive lines of the mesh. In this case corresponding matrices A_{ii} are block-tridiagonal instead of tridiagonal. As a result, solving linear systems with A_{ii} may be much more expensive. On the other hand, the number of iterations required to achieve convergence often decreases rapidly as the block-size increases.

7.3 Polynomial preconditioning

When handling a problem that is solved by 10 000 up to 100 000 processors, polynomial preconditioning is a good technique to improve the convergence rate, by reducing the inner products calculated. For relatively a little amount of processers (0 – 1000) this method is not interesting enough.

In polynomial preconditioning the matrix M is defined by

$$M^{-1} = s(A)$$

where s is a polynomial, typically of low degree. Thus, the original system is replaced by the preconditioned system

$$s(A)Ax = s(A)b$$

which is then solved by a conjugate gradient-type technique. Note that $s(A)$ and A commute and, as a result, the preconditioned matrix is the same for right of left preconditioning. In addition, the matrix $s(A)$ or $As(A)$ does not need to be formed explicitly since $As(A)v$ can be computed for any vector v from a sequence of matrix-by-vector products.

The polynomial s can be selected to be optimal in some sense, and leads to the use of Chebychev polynomials. The criterion that is used makes the preconditioned matrix $s(A)A$ as close as possible to the identity matrix in some sense. If we want to make the spectrum of the preconditioned matrix as close as possible to that of the identity we have to solve a minimizing problem with use of Chebyshev polynomials. For A is Symmetric Positive Definite and some interval $[\alpha, \beta]$ which contains the eigenvalues of A, this leads to the Chebyshev acceleration algorithm.

Let $\theta \equiv \frac{\beta+\alpha}{2}$ be the center and $\delta \equiv \frac{\beta-\alpha}{2}$ be the mid-width of interval $[\alpha, \beta]$.

Algorithm 11: Chebyshev Acceleration

1. $r_0 = b - Ax_0; \sigma_1 = \delta/\theta;$

2. $\rho_0 = 1/\sigma_1; d_0 = \frac{1}{\theta} r_0;$
3. For $k = 0, \ldots,$ until convergence Do:
4. $x_{k+1} = x_k + d_k$
5. $r_{k+1} = r_k - A d_k$
6. $\rho_{k+1} = (2\sigma_1 - \rho_k)^{-1};$
7. $d_{k+1} = \rho_{k+1}\rho_k d_k - \frac{2\rho_{k+1}}{\delta} r_{k+1}$
8. End Do

If the estimates α and β are obtained by Greshgorin, it is possible that $\alpha \leq 0$ although A is SPD. In this case the Least Squares polynomials approach is applied. For more details see Saad, p. 359 – 361.

7.4 Multicoloring

The problem addressed by multicoloring is to determine a coloring of the nodes of the adjacency graph of a matrix such that any two adjacent nodes have different colors. For a 2-dimensional finite difference grid (5-point operator), this can be achieved with two colors, typically referred to as red-black-ordering.

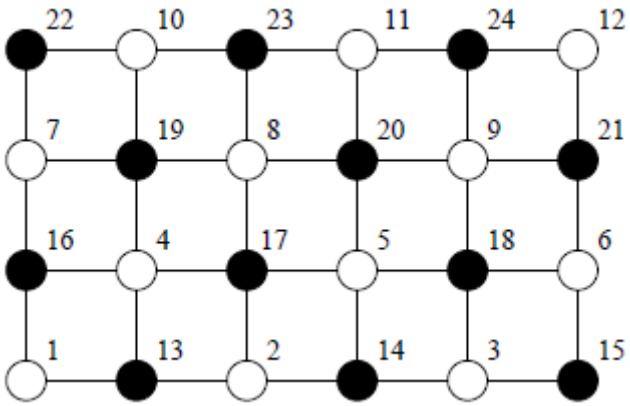

Figure 2: Red-black coloring and labeling of the nodes of a 6 x 4 grid.

Assume that the unknowns are labeled by listing the red unknowns first together, followed by the black ones, as in the figure above. Since the red nodes are not coupled with other black ones and vice versa, the system that results from this ordering will have the structure:

$$\begin{pmatrix} D_1 & F \\ E & D_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \qquad (6)$$

In which D_1 and D_2 are diagonal matrices. The reordered matrix is shown below.

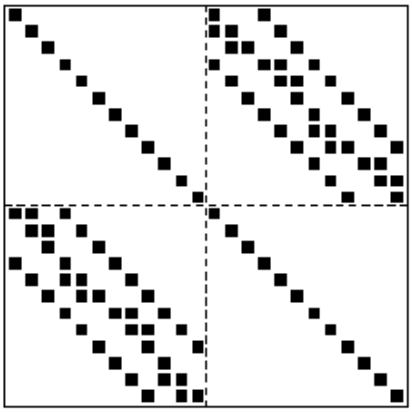

Figure 3: Matrix associated with the red-black ordering of figure 2.

Solution of Red-Black Systems

The easiest way to exploit red-black ordering is to use ILU(0) preconditioners for solving the block system (6). It appears that the number of iterations to achieve convergence can be much higher with red-black ordering than with natural ordering.

A second method solves the reduced system which involves only the black unknowns:

$$(D_2 - ED_1^{-1}F)x_2 = b_2 - ED_1^{-1}b_1.$$

This is again a sparse linear system with half as many unknowns. For easy problems, the reduced system can often be solved efficiently with only diagonal preconditioning. The computation of the reduced system is a highly parallel and inexpensive process. It is not necessary to form the reduced system, especially when D_1 is not diagonal, such as in Domain Decomposition methods. For example, applying the matrix to the vector x, can be performed using nearest-neighbor communication, and this can be more efficient than the standard approach of multiplying the vector with the Schur complement matrix $D_2 - ED_1^{-1}F$ and it can save storage.

7.5 Multi-elimination ILU

Here we will discuss a parallel algorithm for the Gaussian elimination for a general sparse matrix. Paralellism is obtained by finding unknowns that are independent of each other. A set of unknowns of a linear system which are independent is called an independent set. A few algorithms for finding independent set orderings of a general sparse graph are discussed in Saad, Chapter 3. Given this independent set, we permute our original matrix in such a way that we get the following structure:

$$\begin{pmatrix} D & E \\ F & C \end{pmatrix}$$

Where D is a diagonal matrix and C is arbitrary.

The rows associated with an independent set can be used as pivots simultaneously. When such rows are eliminated, we have reduces our system to a smaller one, which is again a sparse one. We call this the first-level reduces system. Then we repeat the process of reduction a few times again. As the level of reduction increases, the reduced systems gradually lose sparsity.

Let A_j be the matrix obtained at the j-th step of the reduction, j = 0, …, n, where n is the number of reduction steps taken and $A_0 = A$. Assume that an independent set ordering is applied to A_j and that the matrix is permuted accordingly as follows:

$$P_j A_j P_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix}$$

Where D_j is a diagonal matrix. When the unknowns of the independent set are eliminated, we get the next reduces matrix,

$$A_{j+1} = C_j - E_j D_j^{-1} F_j .$$

If we write: $C_j = A_{j+1} + E_j D_j^{-1} F_j$, this results in a block LU factorization:

$$P_j A_j P_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix} = \begin{pmatrix} I & O \\ E_j D_j^{-1} & I \end{pmatrix} \times \begin{pmatrix} D_j & F_j \\ O & A_{j+1} \end{pmatrix}$$

Thus, in order to solve a system with the matrix A_j, both a forward and backward substition need to be performed with the block matrices on the right –hand side of the above system. The backward solution involves solving a system with the matrix A_{j+1}. This block factorization approach can be used recursively until a system results that is small enough to be solved with a standard method.

7.6 ILUM

The successive reduction steps described above will give rise to matrices that become more and more dense due to the fill-in introduced by the elimination process. The fill-in elements will be dropped by using a simple dropping strategy. Here, an approximate version of the successive reduction steps can be used to provide an approximate solution $M^{-1}v$ to $A^{-1}v$ for any given v. This can be used to precondition the original linear system. The reduced matrix can then be written as:

$$A_{j+1} = \left(C_j - E_j D_j^{-1} F_j\right) - R_j$$

in which R_j is the matrix with the dropped elements in this reduction step.

The ILU block factorization becomes like:

$$P_j A_j P_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix} = \begin{pmatrix} I & O \\ E_j D_j^{-1} & I \end{pmatrix} \times \begin{pmatrix} D_j & F_j \\ O & A_{j+1} \end{pmatrix} + \begin{pmatrix} O & O \\ O & R_j \end{pmatrix}$$

It is not necessary to save the successive A_j matrices but only the last one that is generated. We need to store the sequence of sparse matrices:

$$B_{j+1} = \begin{pmatrix} D_j & F_j \\ E_j D_j^{-1} & O \end{pmatrix}$$

We refer to this incomplete factorization as ILUM (ILU with Multi-Elimination). We distinguish a preprocessing phase and a backward and forward solution phase. The preprocessing phase consists of a succession of n (number of reduction steps taken) applications of the following three steps:

(1) Finding the independent set ordering
(2) Permuting the matrix
(3) Reducing it

Algorithm 12: ILUM: Preprocessing Phase

1. Set $A_0 = A$.
2. For $j = 0, 1, \ldots, n-1$ Do
3. Find an independent set ordering permutation P_j for A_j;
4. Apply P_j to A_j to permute it into the form $P_j A_j P_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix}$;
5. Apply P_j to $B_1, \ldots, B_j$;
6. Apply P_j to $P_1, \ldots, P_{j-1}$;
7. Compute the matrices A_{j+1} and B_{j+1}.
8. End Do

In the backward and forward solution phase, the last reduced system must be solved but not necessary with high accuracy. It can be solved according to the level of tolerance allowed in the dropping strategy during the preprocessing phase. First, some notation is introduced:

We start by applying the product $P_{n-1}, P_{n-2}, \ldots, P_1$ to the right-hand side. We overwrite the result on the current solution N-vector x_0. Now partition this vector into:

$$x_0 = \begin{pmatrix} y_0 \\ x_1 \end{pmatrix}$$

According to the partitioning $P_j A_j P_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix}$. The forward step consists of transforming the second component of the right-hand side as

$$x_1 := x_1 - E_0 D_0^{-1} y_0.$$

As a result, x_1 is partitioned in the same manner as x_0 and the forward elimination is continued the same way. Thus at each step:

$$x_j = \begin{pmatrix} y_j \\ x_{j+1} \end{pmatrix}.$$

A forward elimination step defines the new x_{j+1} using the old x_{j+1} and y_j for $j = 0, \ldots, n-1$ while a backward step defines y_j using the old y_j and x_{j+1} for $j = n-1, \ldots, 0$.

Above gives the following algorithm:

Algorithm 13: ILUM: Forward and Backward Solutions

1. Apply global permutation to right-hand side b and copy into x_0.
2. For $j = 0, \ldots, n-1$ Do: [forward step]
3. $x_{j+1} := x_{j+1} - E_j D_j^{-1} y_j$
4. End Do
5. Solve with a relative tolerance ϵ:

6. $A_n x_n := x_n$.
7. For $j = n-1, \ldots, 0$ Do: [backward sweep]
8. $y_j := D_j^{-1}(y_j - F_j x_{j+1})$.
9. End Do
10. Permute the resulting solution vector back to the original ordering to obtain the solution x.

7.7 Distributed ILU and SSOR

Here we describe parallel variants of the Block Succesive Over-Relaxation (BSOR) and ILU(0) preconditioners which are suitable for distributed memory environments. First, a distributed matrix is a matrix whose entries are located in the memories of different processors in a multiprocessor system.

In the ILU(0) factorization, the LU factors have the same nonzero patterns as the original matrix, so that the references of the entries belonging to the external subdomains in the ILU(0) factorizations are identical with those of the matrix-by-vectors product operation with the matrix A. This is not the case for ILU(p) when $p>0$. Here we first define a global ordering, which is an ordering for the subdomains (subgraphs). This is based on the graph which describes the coupling between the subdomains: Two subdomains are coupled if and only if they contain at least a pair of coupled unknowns, one from each subdomain. Then, within each subdomain, define a local ordering. In those subdomains, we distinguish between points that are not coupled with nodes belonging to other subdomains, these are the interior points, and between interface points. These are local nodes that are coupled with at least one node which belongs to another processor. Each subdomain is mapped to the same processor.

Consider the rows associated with the interior nodes. Because the unknowns associated with these nodes are not coupled with variables from other processors, these nodes can be eliminated independently in the ILU(0) process. After this process, the interface nodes can be eliminated in a certain order. There are two natural choices for this order. The first one is based on the global ordering of the subdomains, but this is somewhat unnatural. A proper order can also be defined by replacing the PE -numbers with any labels, provided that any two neighboring processors have a different label. The most natural way to do this is by performing a multicoloring of the subdomains.

Define lab_j as the label of Processor number j.

Algorithm 14: Distributed ILU(0) factorization

1. In each processor P_i, $i = 1, \ldots, p$ Do:
2. Perform the ILU(0) factorization for interior local rows.
3. Receive the factored rows from the adjacent processors j with
4. $Lab_j < Lab_i$
5. Perform the ILU(0) factorization for the interface rows with
6. pivots received from the interior rows completed in step 2.
7. Send the completed interface rows to adjacent processors j with
8. $Lab_j > Lab_i$
9. End Do

In above algorithm, step 2 can be performed parallel. Once this ILU(0) factorization is completed, the preconditioned Krylov subspace algorithm will require a forward and backward sweep at each step. The distributed forward/ backward solution based on this factorization can be implemented as follows:

Algorithm 15: Distributed Forward and Backward Sweep

1. In each processor P_i, $i = 1, \dots, p$ Do:
2. Forward solve:
3. Perform the forward solve for the interior nodes.
4. Receive the updates values from the adjacent processors j
5. with $Lab_j < Lab_i$.
6. Perform the forward solve for the interface nodes.
7. Send the updated values of boundary nodes to the adjacent
8. processors j with $Lab_j > Lab_i$.
9. Backward solve:
10. Receive the updated values from the adjacent processors j
11. with $Lab_j > Lab_i$.
12. Perform the backward solve for the boundary nodes.
13. Send the updated values of boundary nodes to the adjacent
14. processors j with $Lab_j < Lab_i$.
15. Perform the backward solve for the interior nodes.
16. End Do.

In this algorithm, lines 3 and 15 can be computed parallel.

7.8 Incomplete Poisson Preconditioning

As shown in before, there is a price to pay for this parallelism in terms of convergence speed. Note that a simpler form of the equation we want to solve for the NEMO model, is the Poisson equation.

In [4] a description of a preconditioner for the Poisson equation can be found. This preconditioner depends on the sum decomposition of A into its lower triangular part L and its diagonal D. The approximation of the inverse then is:

$$M^{-1} = (I - LD^{-1})(I - D^{-1}L^T)$$

Applying this preconditioner to the PCG algorithm requires that the modified system is still symmetric positive definite, which in turn requires that the preconditioner is a symmetric real-valued matrix.

It can be shown that the preconditioner M can be written as:

$$M = KK^T$$

where

$$K = I - LD^{-1} \quad \text{and} \quad K^T = I - D^{-1}L^T. \qquad (6)$$

In the case of a two-dimensional regular discretization, focusing on an inner grid cell, the stencil of the i-th row of A is:

$$row_i(A) = \left(a_{j-1}, a_{i-1}, a, a_{i+1}, a_{j+1}\right) = (-1, -1, 4, -1, -1)$$

Hence, the stencils of the i-th row of L, D^{-1} and L^T can be written as:

$$row_i(L) = (-1, -1, 0, 0, 0)$$
$$row_i(D^{-1}) = (0, 0, 0.25, 0, 0)$$
$$row_i(L^T) = (0, 0, 0, -1, -1)$$

After performing the operations for (6):

$$row_i(K) = (0.25, 0.25, 0, 0, 0)$$
$$row_i(K^T) = (0, 0, 1, 0.25, 0.25)$$

Taking the matrix product $M = KK^T$ results in:

$$row_i(M^{-1}) = \left(m_{j-1},\ m_{i+1,i-1}, m_{i-1}, m,\ m_{i+1}, m_{i-1,j+1}, m_{j+1}\right)$$
$$= (0.25, 0.0625, 0.25, 1.125, 0.25, 0.0625, 0.25)$$

Observe that there are two fill-in elements. In the three-dimensional case the stencil enlarges from 7 non-zero elements up to 13 non-zero elements for each row, which would almost double the computational effort. By looking again, note that the additional non-zero values are rather small compared to the rest of the coefficients. This nice property remains true in three dimensions. So the fill-in elements are discarded:

$$row_i(M^{-1}) = (0.25, 0.25, 1.125, 0.25, 0.25)$$

What about the parallelism of this preconditioner? Since we can compute the non-zero elements of each row separately, the degree of parallelism of this preconditioner is N, where N is the number of unknowns or dimension of A.

Chapter 8 Domain Decomposition Methods

Domain decomposition methods refer to a collection of techniques which revolve around the principle of divide-and-conquer. Reasons why such techniques can be advantageous are various. First, the problem can become simpler because of the geometric forms of the subdomains. Second, the physical problem can sometimes be split naturally into a small number of subregions where the modeling equations are different. Third, Domain Decomposition makes parallel computing possible.

We will introduce the basic concepts of domain decomposition through an example from Partial Differential Equations.

8.1 Introduction

Consider the Laplace Equation on an L-shaped domain Ω partitioned as shown in the figure below.

$$\begin{cases} \Delta u = f \text{ in } \Omega \\ u = u_\Gamma \text{ on } \Gamma = \partial\Omega \end{cases}$$

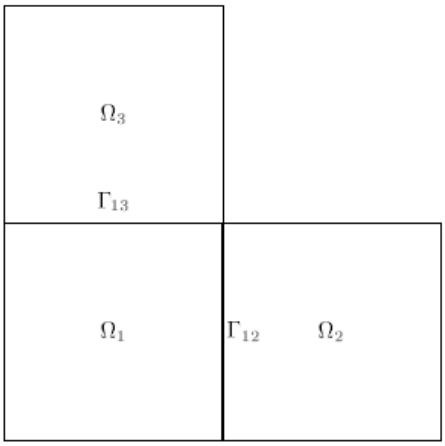

Figure 4 A L-shaped domain subdivided into three subdomains.

Domain decomposition methods are implicitly or explicitly based on different ways of handling the unknown at the interfaces. From PDE point of view, If the value of the solution is known at the interfaces, these values could be used in Dirichlet-type boundary conditions and we will obtain s uncoupled Poisson Equations. When we solve these equations, we obtain the value at the interior points.

Assume that the problem above is discretized with central differences. We can label the nodes as shown in the figure below; the interface nodes are labeled last.

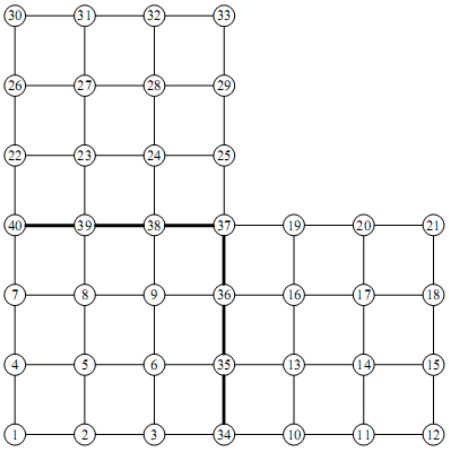

Figure 5: interface nodes are numbered last.

For a general partitioning into s subdomains, the linear system associated with the problem has the following structure:

$$\begin{pmatrix} B_1 & & & E_1 \\ & B_2 & & E_2 \\ & & \ddots & \vdots \\ & & B_s & E_s \\ F_1 & F_2 & \cdots & F_s & C \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \\ g \end{pmatrix}$$

Each x_i here represents the subvector of unknowns that are interior to subdomain Ω_i and y represents the vector of all interface unknowns. It is useful to express the above system in the simpler form:

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} B & E \\ F & C \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}$$

Here E represents the subdomain to interface coupling seen from the subdomains, while F represents the interface to subdomain coupling seen the interface nodes.

We distinguish two types of partitionings:

1. Vertex-based,
2. Element-by-element based

At the first one, works by dividing the origin vertex set into subsets of vertices. This means that each subset of vertices is mapped to the same processor. In the second one we map all the information that is related to an element to the same processor. The only restriction in this case is that no element should be split between two subdomains.

The figure below shows us an example for each type.

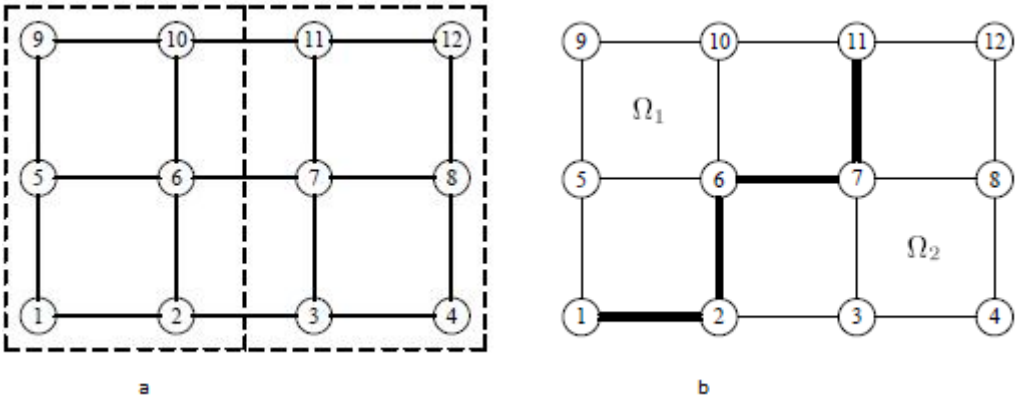

Figure 6: (a) Vertex-based, (b) element-based partitioning of a 4 x 3 mesh into two subregions.

8.2 Types of Techniques

Once we have a partitioning, the interface values can be obtained by block-Gaussian elimination which may be too expensive for large problems, and thus for most applications.

Another options are the Schwarz Alternating Procedures, methods that alternate between the subdomains, solving a new problem each time with boundary conditions updated from the most recent solutions.

The subdomains may overlap. This means that the Ω_i's are such that

$$\Omega = \bigcup_{i=1,s} \Omega_i \,, \qquad \Omega_i \cap \Omega_j \neq \emptyset$$

where Ω_i and Ω_j are neighbouring domains.

It is typical to quantify the extent of overlapping by the number of mesh-lines that are common to the two subdomains.

We can distinguish domain decomposition techniques by four features:

1. *Type of partitioning*. For example, should it occur along edges, or along vertices or by elements.
2. *Overlap*. Should sub-domains overlap or not, and if yes, how much?
3. *Processing of interface values.* Is the Schur complement approach (explained later) used? Should there be successive updates to the interface values?
4. *Subdomain solution.* Should the subdomain problems be solved exactly or approximately by an iterative method?

We classify the methods we discuss here in four distinct groups:

1. Direct methods and the substructuring approach are useful for introducing some definitions and for providing practical insight.
2. The Schwarz Alternating Procedures, a class of the simplest and oldest techniques.
3. Methods based on preconditioning the Schur Complement.

4. Solving $Ax = b$ using preconditioning derived from domain decomposition concepts.

8.3 Direct solution and the Schur complement

Recall the system that we obtained after applying domain decomposition an reordering the unknowns:

$$\begin{pmatrix} B & E \\ F & C \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}$$

Assuming B is non-singular, we are allowed to write x as: $x = B^{-1}(f - Ey)$. Substituting this back gives:

$$FB^{-1}(f - Ey) + Cy = g$$

$$FB^{-1}f - FB^{-1}Ey + Cy = g$$

$$(C - FB^{-1}E)y = g - FB^{-1}f \qquad (7)$$

The matrix

$$S = (C - FB^{-1}E)$$

is called the *Schur complement matrix* associated with the y variable. If this matrix can be formed, the system (7) can be solved, all the interface variables y will become available. Then the remaining unknowns can be computed via $x = B^{-1}(f - Ey)$. Recall that the matrix B is a block diagonal matrix of s blocks. Thus, parallelism arises naturally from the structure of B, which with its particular structure decouples every linear system into s separate systems.

By defining the following matrices, one linear system solution can be saved, when Algorithm 16 is applied.

Define: $E' = B^{-1}E$ and $f' = B^{-1}f$. Then for x we can write: $x = B^{-1}f - B^{-1}Ey = f' - E'y$, and we get the following algorithm:

Algorithm 16: Block Gaussian Elimination

1. Solve $BE' = E$, and $Bf' = f$ for E' and f', respectively
2. Compute $g' = g - Ff'$
3. Compute $S = C - FE'$
4. Solve $Sy = g'$
5. Compute $x = f' - E'y$.

In practice, all the B_i matrices are factored and then the systems $B_iE'_i = E_i$ and $B_if'_i = f_i$ are solved. In general, many columns of E_i will be zero. Therefore any efficient code based on the above algortihm should start by identifying the nonzero columns.

The following proposition tells us something about the properties of the Schur complement matrix.

Proposition 4

Let $A = \begin{pmatrix} B & E \\ F & C \end{pmatrix}$ *be a nonsingular matrix and be such that the submatrix* B *is nonsingular.*

Let $R_y \begin{pmatrix} x \\ y \end{pmatrix} = y$ *be the restriction operator onto the interface variables.*

Then the following properties are true:

1. *The Schur complement matrix* $S = C - FB^{-1}E$ *is nonsingular*
2. *If* A *is Symmetric Positive Definite, then so is* S.
3. *For any y,* $S^{-1}y = R_y A^{-1} \begin{pmatrix} 0 \\ y \end{pmatrix}$.

A consequence of the second property is that when A is positive definite, an algorithm such as Conjugate Gradient can be used to solve the reduced system

$$(C - FB^{-1}E)y = g - FB^{-1}f$$

The third property allows preconditioners for S to be defined based on solution techniques with the matrix A.

8.4 The Schur complement for vertex-based partitionings

For vertex-based partitioning the matrix A has a different structure from what we just have seen because of the numbering of the unknowns. Since there are no interface nodes, there is no renumbering. As a result, the Schur complement system also has a different structure. For vertex-based partitionings the Schur matrix S is assembled from local Schur complement matrices and interface-to-interface information. For more details see Saad, page 390-392.

8.5 Schwarz Alternating Procedures

The next procedure we will discuss is called the Multiplicative Schwarz procedure and it consists of three parts: alternating between two overlapping domains, solving the Dirichlet problem on one domain at each iteration, and taking boundary conditions based on the most recent solution obtained from the other domain.

Multiplicative Schwarz Procedure

Assume that each pair of neighboring subdomains has a non-empty overlapping region like in the figure below. The boundary of subdomain Ω_i that is contained in subdomain j is denoted by $\Gamma_{i,j}$.

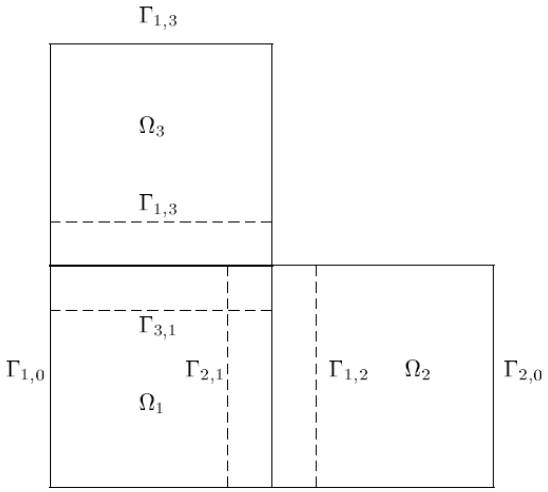

Figure 7: An L-shaped domain subdivided into three overlapping subdomains.

Call Γ_i the boundary of Ω_i consisting of its original boundary which is denoted by $\Gamma_{i,0}$ and the $\Gamma_{i,j}$'s. Denote by u_{ji} the restriction of the solution u to the boundary Γ_{ji}.

The following algorithm describes the Schwarz Alternating Procedure (SAP).

Algorithm 17: SAP- Multiplcative Schwarz Sweep-Matrix form

1. Choose an initial guess u to the solution
2. Until convergence Do:
3. For $i = 1, \dots, s$ Do:
4. Solve $A_i \delta_i = r_i$
5. Compute $x_i := x_i + \delta_{x,i}$, $y_i := y_i + \delta_{y,i}$, and set $r_i := 0$
6. For each $j \in N_i$ Compute $r_{y,j} := r_{y,j} - E_{ji}\,\delta_{y,i}$
7. End Do
8. End Do

Here r_i is the local part of the most recent global residual vector $b - Ax$, and A_i is the local matrix that is associated with the subdomain Ω_i. The structure of A_i is as follows:

$$A_i = \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix}$$

Finally we write:

$$u_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \quad \delta_i = \begin{pmatrix} \delta_{x,i} \\ \delta_{y,i} \end{pmatrix}, \quad r_i = \begin{pmatrix} r_{x,i} \\ r_{y,i} \end{pmatrix},$$

Here E_{ji} represents the coupling from internal subdomain j to external subdomain i. For more details on splitting up the Schur complement see Saad, page 391.

8.6 Schur complement Approaches

Schur complement methods are based on solving the reduced system:

$$(C - FB^{-1}E)y = g - FB^{-1}f$$

by some preconditioned Krylov subspace method. The first step in this solution procedure is getting the right-hand side $g^{'} = g - FB^{-1}f$. Then we solve the reduced system $Sy = g'$ via an iterative method, and we back-substitute the solution y to compute $x = B^{-1}(f - Ey)$.

In order to solve the reduced system by a direct method, S needs to be formed explicitly. But the full matrix S is often not available.

For matrix-by-vector operations $w = Sv$ this problem can be solved by the following procedure which only requires matrix-by-vector multiplications and one linear system solution:

1. Compute $v^{'} = Ev$,
2. Solve $Bz = v'$
3. Compute $w = Cv - Fz$.

Note that a linear system involving B translates into s independent linear systems which must be solved exactly by a direct solution method or an iterative method with high accuracy.

Preconditioning the matrix S is hard to do. Here, we discuss some preconditioners that can be used in this situation.

8.7 Induced Preconditioners

Proposition 4 tells us that a preconditioning operator M to S can be defined from the (approximate) solution obtained with A.

It can be shown that, if a preconditioner M_A to A is obtained, the preconditioning operator M_S can be induced from it as follows:

$$M_S = (R_y M_A^{-1} R_y^T)^{-1} \qquad (8)$$

Where R_y represents the restriction operator on the interface variables defined in Proposition 4.

In the special case where $M_A = L_A U_A$ the following proposition holds:

Proposition 5

Let $M_A = L_A U_A$ be an ILU preconditioner for A. then the preconditioner M_S for S induced by M_A as defined by (8), is given by

$$M_S = L_S U_S, \text{ with } L_{S=} R_y L_A R_y^T,\ U_S = R_y U_A R_y^T.$$

An important consequence of this proposition is that the parallel Gaussian elimination can be exploited for deriving an ILU preconditioner for S by using a general purpose ILU factorization.

Suppose we have an incomplete LU factorization of A, then we can approximate the local Schur complement matrices S_i by:

$$\widetilde{S}_i = C_i - F_i U_i^{-1} L_i^{-1} E_i,$$

Since an incomplete factorization is performed, some drop strategy is applied to the elements in $\widetilde{S}_i$. Let $T_i = \widetilde{S}_i - R_i$ be the matrix after the drop strategy is applied.

Finally, the ILU factorization of the matrix below can be computed.

$$\begin{pmatrix} T_1 & E_{12} & E_{13} & \cdots & E_{1s} \\ E_{21} & T_2 & E_{23} & \cdots & E_{2s} \\ \vdots & & \ddots & & \vdots \\ & & & \ddots & \vdots \\ E_{s1} & E_{s2} & E_{s3} & \cdots & T_s \end{pmatrix}.$$

where the E_{ij}'s are like described before.

8.8 Full matrix methods

Recall:

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} B & E \\ F & C \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}$$

Any technique that iterates this system is called a *full matrix method*. Preconditioners for A can be derived from approximating interface values in the same way that preconditioners for the Schur complement were derived from the LU factorization of A.

First we state a few simple relations between iterations involving A and S. After that we discuss preconditioning techniques.

Proposition 6

Let

$$L_A = \begin{pmatrix} I & O \\ FB^{-1} & I \end{pmatrix}, \quad U_A = \begin{pmatrix} B & E \\ O & I \end{pmatrix}$$

and assume that a Krylov subspace method is applied to the original system

$$\begin{pmatrix} B_1 & & & & E_1 \\ & B_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & B_s & E_s \\ F_1 & F_2 & \cdots & F_s & C \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \\ g \end{pmatrix}$$

with left preconditioning L_A and right preconditioning U_A, and with initial guess of the form

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} B^{-1}(f - Ey_0) \\ y_0 \end{pmatrix}.$$

then this preconditioned Krylov iteration will produce iterates of the form

$$\begin{pmatrix} x_m \\ y_m \end{pmatrix} = \begin{pmatrix} B^{-1}(f - Ey_m) \\ y_m \end{pmatrix}$$

In which the sequence y_m is the result of the same Krylov subspace method applied without preconditioning to the reduced linear system $Sy = g'$ with $g' = g - FB^{-1}f$ starting with the vector y_0.

The following extension of this proposition allows S to be preconditioned:

Proposition 7

Let $S = L_s U_S - R$ be an approximate factorization of S and define

$$L_A = \begin{pmatrix} I & O \\ FB^{-1} & L_s \end{pmatrix}, \quad U_A = \begin{pmatrix} B & E \\ O & U_S \end{pmatrix}.$$

Assume that a Krylov subspace method is applied to the original system (see Proposition 6) with left preconditioning L_A and right preconditioning U_A, and with initial guess of the form

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} B^{-1}(f - Ey_0) \\ y_0 \end{pmatrix}.$$

Then the preconditioned Krylov iteration will produce iterates of the form

$$\begin{pmatrix} x_m \\ y_m \end{pmatrix} = \begin{pmatrix} B^{-1}(f - Ey_m) \\ y_m \end{pmatrix}.$$

Moreover, y_m is the result of the same Krylov subspace method applied the reduced linear system $Sy = g - FB^{-1}f$, left preconditioned with L_S and right preconditioned with U_S, and starting with the vector y_0.

Proof 6-7: see Saad, page 411- 412.

Also there are two other versions in which S is allowed to be preconditioned from the left to the right. Thus, is M_S is a certain preconditioner for S, use the following factorizations

$$\begin{pmatrix} B & E \\ F & C \end{pmatrix} = \begin{pmatrix} I & O \\ FB^{-1} & M_S \end{pmatrix} \begin{pmatrix} I & O \\ O & M_S^{-1}S \end{pmatrix} \begin{pmatrix} B & E \\ O & I \end{pmatrix}$$

$$= \begin{pmatrix} I & O \\ FB^{-1} & I \end{pmatrix} \begin{pmatrix} I & O \\ O & SM_S^{-1} \end{pmatrix} \begin{pmatrix} B & E \\ O & M_S \end{pmatrix},$$

to derive the appropriate left or right preconditioners.

Although the previous results indicate a preconditioned Schur Complement iteration is mathematically equivalent to a certain preconditioned full matrix method, the practical benefits in iterating with the nonreduced system are that the operation Sx needs not to be performed explicitly.

8.9 Graph partitioning

One of the first things to do when solving a problem on a parallel computer, is deciding how to map the data into the processors. For shared memory and SIMD computers, directives are often provided to help the user input a desired mapping. Distributed memory computers require finding good mappings by the user. For more details on Graph partitioning techniques see Saad, Chapter 13.

8.10 Deflation

Deflation reduces the number of iterations for the Preconditioned Conjugate Gradient (PCG) method. This is done in [Gupta, p. 16] by treating the bad eigenvalues of the preconditioned matrix $M^{-1}A$ in the system:

$$M^{-1}Ax = M^{-1}b$$

where M^{-1} is a symmetric positive definite preconditioner (SPD) and A is the symmetric positive definite coefficient matrix.

The original linear system $Ax = b$ can be solved using the splitting:

$$x = (I - P^T)x + P^T x$$

This splitting is equivalent with:

$$\begin{aligned} &x = (I - P^T)x + P^T x \;\Leftrightarrow \\ &x = Qb + P^T x \;\Leftrightarrow \\ &Ax = AQb + AP^T x \;\Leftrightarrow \\ &b = AQb + PAx \;\Leftrightarrow \\ &Pb = PAx \end{aligned}$$

where $E \in \mathbb{R}^{k\times k}$, $Q \in \mathbb{R}^{n\times n}$, $Z \in \mathbb{R}^{n\times k}$ and $P \in \mathbb{R}^{n\times n}$ which is called the deflation matrix. The k columns of Z are called the deflation vectors.

The deflation vectors are chosen in a geometric way. The computational domain is divided into several subdomains, where each subdomain corresponds to one or more deflation vectors. The deflation vector that corresponds to subdomain Ω_i consists of ones on positions that correspond to interior gridpoints of Ω_i and zeroes for other gridpoints.

Considering the x in the last equation:

$$Pb = PAx$$

This x is not necessarily a solution of the original linear system $Ax = b$, since it might consist of components of the null space of PA. Therefore, this deflated solution is denoted by $\hat{x}$. The deflated system is now:

$$PA\hat{x} = Pb$$

The Deflated Preconditioned Conjugate Gradient method calculates the deflated solution $\hat{x}$ of the deflated preconditioned system:

$$M^{-1}PA\hat{x} = M^{-1}Pb$$

and afterwards the solution $\hat{x}$ is transformed by:

$$\hat{x} = Qb + P^T x$$

to the solution of the original system $Ax = b$.

Algorithm 18: Deflated Preconditioned Conjugate Gradient Algorithm

1. Select x_0. Compute $r_0 := b - Ax_0$ and $\hat{r}_0 = Pr_0$. Solve $My_0 = \hat{r}_0$ and set $p_0 := y_0$.
2. For $j = 0, 1, \ldots$, until convergence Do:
3. $\widehat{w}_j := PAp_j$
4. $\alpha_j := \frac{(\hat{r}_j, y_j)}{(p_j, \widehat{w}_j)}$
5. $\hat{x}_{j+1} := \hat{x}_j + \alpha_j p_j$
6. $\hat{r}_{j+1} := \hat{r}_j - \alpha_j \widehat{w}_j$
7. Solve $My_{j+1} = \hat{r}_{j+1}$
8. $\beta_j := \frac{(\hat{r}_{j+1}, y_{j+1})}{(\hat{r}_j, y_j)}$
9. $p_{j+1} := y_{j+1} + \beta_j p_j$
10. End For
11. $x_{it} := Qb + P^T x_{j+1}$

The selection of the deflation vectors has influence on the convergence of the algorithm. One can divide a simple squared domain into strips of little squares. These different divisions will have different convergence rates.

Chapter 9 Numerical Experiments

Like described in the summary of the literature study, it is hard to find good parallel methods that have good convergence qualities.

The goal of this research is finding a method that is parallelizable and has a convergence rate that is faster than the preconditioned conjugate gradient method used at this moment.

Therefore, implementations will be tested on simple *test problems*. When having good results on the simple test problem, we can apply our implementations to a less simple test problem that is more likely to the original NEMO-equation (1).

These test problems consider the equation described in the introduction:

$$\frac{\partial}{\partial x}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial x}\right)+\frac{\partial}{\partial y}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial y}\right)=\frac{\partial \bar{M}_2}{\partial x}-\frac{\partial \bar{M}_1}{\partial y}$$

in simpler forms on the unit square in 2-dimensions.

9.1 Test problems

In the first case the NEMO equation is simplified as much as possible, getting the Poisson equation. For this equation we take a simple, but not trivial, right-hand side. The Poisson equation is shown below:

$$-\frac{\partial^2 u}{\partial x^2}-\frac{\partial^2 u}{\partial y^2}=1$$

The computational domain is also simplified to the unit square. Test results on this first test problem can be found in Chapter 10.

The second test problem will involve the variable H that depends on x and y. The right-hand side of this equation will be zero. The computational region is the same as the first test problem: the unit square. The equation for the second test problem will be:

$$\frac{\partial}{\partial x}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial x}\right)+\frac{\partial}{\partial y}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial y}\right)=0$$

First, H is taken constant, then it will be varied which is more likely in practice.

In the last case the computational domain is made bigger.

Results for the second test problem are shown in Chapter 11.

9.2 Numerical Algorithms

At this moment, the Preconditioned Conjugate Gradient Method with diagonal scaling is applied to solve the Nemo equation. We start by applying the Conjugate Gradient Algorithm to our test problems. These results will be the basic results with which the other results will be compared.

By preconditioning the Conjugate Gradient (CG) Algorithm with an appropriate preconditioner, an improvement in convergence rate is expected. We choose the Incomplete Cholesky Decomposition to precondition the CG algorithm (ICCG(0)).

Since ICCG(0) is not parallelizable, an another preconditioner which has good parallel properties is applied: the Incomplete Poisson (IP) preconditioner. Here, the price paid for parallelism is clear when results are compared to ICCG(0).

By applying Deflation it is possible to reduce the number of iterations for de Preconditioned Conjugate Gradient Algorithm, hence we Deflate the Preconditioned CG, where CG is preconditioned by diagonal scaling and by the IP preconditioner. The latter preconditioner is more expensive than the first one. So based on the test results one can make a choice between the two preconditioners.

The original subroutine is fully written in Fortran 90. Our tests are computed in Matlab.

Chapter 10 The Poisson Equation

10.1 From Nemo to the Poisson equation

Recall the equation mentioned in the first chapter:

$$\left[\nabla \times \left[\frac{1}{H}\boldsymbol{k} \times \nabla\left(\frac{\partial \psi}{\partial t}\right)\right]\right]_z = [\nabla \times \bar{\boldsymbol{M}}]_z$$

Which should be solved for $\frac{\partial \psi}{\partial t}$.

In the equation H denotes the height of the ocean surface (or the depth of the ocean), M represents the collected contributions of the Coriolis, hydro-static pressure gradient, nonlinear and viscous terms and k is the unit vector in z-direction. All variables depend only on the x- and y-direction.

Writing out this equation we get:

$$\frac{\partial}{\partial x}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial y}\right) = \frac{\partial \bar{M}_2}{\partial x} - \frac{\partial \bar{M}_1}{\partial y}$$

On purpose of testing the results of the implementations, the equation is simplified to the *Poisson equation* on the unit square with *homogeneous* boundary conditions:

$$\begin{cases} -\Delta u = -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = 1 \text{ on } (0,1) \times (0,1), \\ u = 0 \quad \text{on the boudary} \end{cases} \quad (8)$$

The right hand side of the equation is made as simple as possible, but not trivial.

10.2 Discretization

Nemo is a finite difference model with a regular domain decomposition. Therefore the Poisson equation is discretized in the same way and the unit square is decomposed in a regular grid. This means that both the unit intervals in x- and y-direction are subdivided into N subintervals of length $h = \frac{1}{N}$. Thus the whole unit square is subdivided into little squares of $h \times h$.

The nodes in x-direction are numbered by i, and the nodes in y-direction are numbered by j. Hence an internal node $u_{i,j}$ has four neighbors, left $u_{i-1,j}$, right $u_{i+1,j}$, down $u_{i,j-1}$ and up $u_{i,j+1}$.

By using Taylors formula for sufficiently smooth u, it can be shown that for each internal node $u_{i,j}$, the Laplacian can be approximated by:

$$-\Delta u_{i,j} = \frac{1}{h^2}[4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}] \quad (9)$$

For nodes adjacent to the boundary, some unknowns can be eliminated. For example, an internal node adjacent to the lift boundary of the square, has one known neighbor, i.e. the left one $u_{i-1,j} = 0$ due to the given boundary condition given in (8).

Thus expression (9) for this node becomes:

$$-\Delta u_{i,j} = \frac{1}{h^2}[4u_{i,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}]$$

Putting this together in the discretization matrix A results is a sparse matrix with the following pattern:

$$A = \frac{1}{h^2}\begin{pmatrix} T & -I & & \\ -I & T & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & T \end{pmatrix}$$

where

$$I = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} \quad \text{and} \quad T = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix}$$

The dimension of I and T depends on the number of gridpoints in x-direction. If the number of gridpoints is n, then the matrix dimensions of I and T are $n \times n$. When having n gridpoints in x-direction, we have also n gridpoints in y-direction. Hence the total number of gridpoints is n^2, so the dimension of A then is $n^2 \times n^2$. For more details see [5] Chapter 3.

Note that matrix A is not only sparse but also symmetric positive definite. These are important properties when implementing iterative solution methods.

10.3 Test Results

First, we tested the Conjugate Gradient Algorithm, the Incomplete Cholesky Conjugate Gradient and the Incomplete Poisson Algorithms on the Poisson equation. The results are shown in the table below:

Dimension	CG	ICCG(0)	IP
9x9	5	4	5
100x100	25	9	14
900x900	64	20	34

Table 1: number of iterations needed for convergence of CG, ICCG(0) and IP.

Observe that both ICCG(0) and IP are doing better than CG. IP is slightly worse than ICCG(0), but that is the price paid to improve parallelism.

Deflation

Convergence speed can be graded up, with having good parallel qualities by applying Deflation. When using the Deflated Preconditioned Conjugate Gradient Algorithm, so called deflation vectors should be chosen. The selection of the deflation vectors has influence on the convergence of the algorithm. Hence, here the algorithm is tested with several selections of the deflations vectors. First, the computational domain is divided into horizontal strips. Each strip contains a few rows of gridpoints. Secondly, the (square) computational domain is divided into little squares.

Strip subdomains

For the 9×9 problem we divided the squared computational region into 3 strings of 3 rows for each string. The Deflated Preconditioned Conjugate Gradient Algorithm needed 6 iterations for convergence. Not a very good result. The results for the 100×100 and 900×900 problems are much more better. First the Deflated Preconditioned Conjugate Gradient Algorithm (DPCG) is tested with the simple preconditioner $M = D$, where D is a diagonal matrix that contains the diagonal elements of A. Secondly, the DPCG Algorithm is tested with a different preconditioner, the Incomplete Poisson Preconditioner (IPP). With this preconditioner, better results are expected, while the degree of parallelism remains the same: n for an $n \times n$-matrix.

For the 100×100 problem, where the region contains 10 gridpoints in each direction, the region is subdivided into 2 and in 5 strips:

Number of Strips	Iterations	
	M = D	IPP
2	22	11
5	20	11

Table 2: Number of iterations with DPCG Algorithm with two different preconditioners for a 10x10 domain subdivided into strips.

For the 900×900 problem:

Number of Strips	Iterations	
	M = D	IPP
2	48	
3	47	26
5	46	25
6	46	25
10	46	25
15	46	25
30		

Table 3: Number of iterations with DPCG Algorithm with two different preconditioners for a 30x30 domain subdivided into strips.

Observe that when the number of strips is low, thus when the strips are broad, the results are bad. Letting the strips become thinner gives better results. In all cases, the IPP preconditioner has faster convergence. Compare the best results with the former algorithms shows that the DPCG Algorithm with the Incomplete Poisson preconditioner gives the best results in terms of convergence and is highly parallel.

Dimension	CG	ICCG(0)	IP	DPCG(D)	DPCG(IPP)
9x9	5	4	5	6	
100x100	25	9	14	20	11
900x900	64	20	34	46	25

Table 4: Number of iterations for CG, ICCG(0), IP, DPCG(D) and DPGC(IPP).

Square subdomains

The computational domains are now subdivided into little squares. Like in the former case, to preconditioners will be tested, the trivial diagonal one and the Incomplete Poisson Preconditioner.

The 100×100 case, with a computational domain contains 10 gridpoints in each direction:

Number of Squares	Iterations	
	M = D	IPP
4	19	10
25	10	7

Table 5: Number of iterations with DPCG Algorithm with two different preconditioners for a 10x10 domain subdivided into squares.

Note that the best case for the diagonal preconditioner, where the domain is divided into 25 squares of 2×2, is better than the best case of 5 strips with a diagonal preconditioner: the first one needs 10 iterations, while the second one needs the double: 20 iterations. In the case of the IPP: for strips 11 iterations needed, while for squares only 7 iterations are good enough for convergence.

Looking to the 900×900 problem where the computational domain contains 30 gridpoints in each direction:

Number of Squares	Iterations	
	M = D	IPP
9	36	23
25	27	16
100	15	10
225	11	8

Table 6: Number of iterations with DPCG Algorithm with two different preconditioners for a 30x30 domain subdivided into squares.

10.4 Conclusions

All test results are shown in the table below. For the deflated algorithm we took the best results, that is, when the domain is divided into squares. For the 10×10 domain this means the domain is divided into 25 squares and for the 900×900 domain there are 225 squares. The squares of both domains have dimensions 2×2.

Dimension	CG	ICCG(0)	IP	DPCG(D)	DPCG(IPP)
9x9	5	4	5	6	
100x100	25	9	14	10	7
900x900	64	20	34	11	8

Table 7: all results for several algorithms applied to the Poisson equation.

The deflated algorithms give very good results. The DPCG with the IP preconditioner is slightly better than DPCG with diagonal scaling. But one has to take into account that the IP preconditioner is more expensive.

Chapter 11 The homogeneous Nemo equation

For the second test problem we consider the homogeneous Nemo equation on the unit square with homogeneous boundary conditions:

$$\begin{cases} \frac{\partial}{\partial x}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial y}\right) = 0 & \text{on } (0,1)\times(0,1) \\ \frac{\partial \psi}{\partial t} = 0 & \text{on the boundary} \end{cases}$$

The only difference between this test problem and the first one, the Poisson equation, is that here the function H that also depends on x and y is involved. H represents the depth of the ocean.

11.1 Discretization

It can be shown that the discretization of the unknown ψ_t in the node (i,j)

$$\frac{\partial}{\partial x}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial x}\right)\Big|_{i,j} + \frac{\partial}{\partial y}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial y}\right)\Big|_{i,j} =$$

$$\frac{1}{h^2}\left\{-\left(\frac{1}{H_{i-\frac{1}{2},j}} + \frac{1}{H_{i+\frac{1}{2},j}} + \frac{1}{H_{i,j-\frac{1}{2}}} + \frac{1}{H_{i,j+\frac{1}{2}}}\right)\psi_{i,j} + \frac{1}{H_{i-\frac{1}{2},j}}\psi_{i-1,j} + \frac{1}{H_{i+\frac{1}{2},j}}\psi_{i+1,j} + \frac{1}{H_{i,j-\frac{1}{2}}}\psi_{i,j-1} + \frac{1}{H_{i,j+\frac{1}{2}}}\psi_{i,j+1}\right\}$$

The k-th row of A that corresponds with the i-th point on x-direction and with the j-th point on y-direction is shown below:

$$\left(0\cdots 0, \frac{1}{H_{i,j-\frac{1}{2}}}, 0\cdots 0, \frac{1}{H_{i-\frac{1}{2},j}}, -\left(\frac{1}{H_{i-\frac{1}{2},j}} + \frac{1}{H_{i+\frac{1}{2},j}} + \frac{1}{H_{i,j-\frac{1}{2}}} + \frac{1}{H_{i,j+\frac{1}{2}}}\right), \frac{1}{H_{i+\frac{1}{2},j}}, 0\cdots 0, \frac{1}{H_{i,j+\frac{1}{2}}} 0\cdots 0\right)$$

11.2 Constant Depth H

Here we start with the simplest form for H. That is when H has a constant value for the whole computational domain. Then H will be varied in two different ways. This is done in sections 11.3 and 11.4.

In order to compare results, we divide the computational domain again into a 3×3, 10×10, and a 30×30 grid. This results a before in a 9×9, 100×100 and a 900×900 matrix.

Again we apply the AG, ICCG(0), IP and DPCG Algorithms to our problem and compare the number of iterations needed to reach the desired convergence with an error $\varepsilon = 10^{-4}$. The initial solutionx_0 is random since the equation is homogeneous and the solution is the trivial zero solution.

Suppose $H = 5250$ for every x and y.

Dimension	CG	ICCG(0)	IP
9x9	5	3	4
100x100	19	7	10
900x900	57	18	31

Table 8: number of iterations for the homogeneous Nemo equation with constant depth, CG, ICCG(0) and IP.

Deflation: Strip subdomains

First we divide the region into strips, afterwards into squares. The Deflated Preconditioned Conjugate Gradient Algorithm is tested with two preconditioners, the diagonal scaling and the Incomplete Poisson preconditioner. This is only done for the 100×100 and the 900×900 problems.

The 100×100 case:

Number of Strips	Iterations	
	M = D	IPP
2	14	8
5	14	8
10	14	8

Table 9: number of iterations for the homogeneous Nemo equation with constant depth when dividing the 10×10 domain into strips and applying DPCG with two preconditioners.

The 900×900 case:

Number of Strips	Iterations	
	M = D	IPP
2	44	24
3	44	24
5	43	23
6	44	24
10	44	24
15	43	24
30	44	24

Table 10: number of iterations for the homogeneous Nemo equation with constant depth when dividing the 30×30 domain into strips and applying DPCG with two preconditioners.

Comparing the results for DPCG with table (..), we see that the IP preconditioner gives better results than DPCG with diagonal scaling. Applying the IP preconditioner with DPCG, gives a slightly better result, but not very much. Moreover one has to take into account the extra costs for DPCG with IP.

Square subdomains

Like the first test case we divide the region into a number of squares and look what happens to the number of iterations.

The 100×100 case:

Number of Squares	Iterations	
	M = D	IPP
4	14	8
25	7	5

Table 11: Number of iterations with DPCG Algorithm with two different preconditioners for a 10x10 domain subdivided into squares.

The 900×900 case:

Number of Ssquares	Iterations	
	M = D	IPP
9	34	18
25	25	14
100	15	9
225	11	8

Table 12: Number of iterations with DPCG Algorithm with two different preconditioners for a 30x30 domain subdivided into squares.

The test results for the square subdomains are much more better than the strips. Comparing these results with the other algorithms in the table below we see that DPCG given much more better results, when choosing the subdomains as squares.

Dimension	CG	ICCG(0)	IP	DPCG(D)	DPCG(IPP)
9x9	5	3	4		
100x100	19	7	10	7	5
900x900	57	18	31	11	8

Table ..: all results for several algorithms applied to the homogeneous Nemo equation with constant depth.

11.3 Variable Depth H (1)

A constant depth of the ocean is not very realistic. Let us vary with H, say, in the middle of the ocean there is a deep region. The region close to the boundary is not deep.

Choosing H is such a way that the square in the middle with corners on the following coordinates: (0.3,0.3), (0.3,0.7), (0.7,0.7) and (0.7,0.3) has depth $H = 5000$. The strips at the boundary have depth $H = 100$.

Dimension	CG	ICCG(0)	IP
9x9	7	5	7
100x100	56	17	33
900x900	201	55	108

Table 13: number of iterations for the homogeneous Nemo equation with variable depth H, CG, ICCG(0) and IP.

Deflation: Strip subdomains

Again, DPCG with the two different preconditioners is applied, leading to the following results:

For the 100x100 problem:

Number of Strips	Iterations	
	M = D	IPP
2	22	28

5	21	31
10	20	29

Table 14: number of iterations for the homogeneous Nemo equation with variable H when dividing the 10×10 domain into strips and applying DPCG with two preconditioners.

The 900x900 problem gives the following results

Number of Strips	Iterations	
	M = D	**IPP**
2	67	110
3	65	104
5	65	104
6	67	108
10	66	102
15	66	91
30	66	105

Table 15: number of iterations for the homogeneous Nemo equation with variable H when dividing the 30×30 domain into strips and applying DPCG with two preconditioners.

Square subdomains

The 100x100 problem:

Number of Squares	Iterations	
	M = D	**IPP**
4	19	30
25	12	23

Table 16: number of iterations for the homogeneous Nemo equation with variable H when dividing the 10×10 domain into squares and applying DPCG with two preconditioners.

The 900x900 problem:

Number of Squares	Iterations	
	M = D	**IPP**
9	47	88
25	37	69
100	20	45
225	18	43

Table 17: number of iterations for the homogeneous Nemo equation with variable H when dividing the 30×30 domain into squares and applying DPCG with two preconditioners.

Conclusions

It may not be surprising that also in this case again the DPCG algorithm with diagonal scaling for square subdomains gives the best results:

Dimension	CG	ICCG(0)	IP	DPCG(D)	DPCG(IPP)
9x9	7	5	7		
100x100	56	17	33	12	23
900x900	201	55	108	18	43

Table 18: all results for several algorithms applied to the homogeneous Nemo equation with variable H.

11.4 Variable Depth H (2)

One big step in the ocean is also not a very realistic image of the ocean. The second way of varying H is making the H going deeper in steps, to make it more realistic.

Let the square with dimensions (0.2x0.2) in the middle have depth $H = 5000$. The coordinates of the corners of this square are: (0.4;0.4), (0.6;0.4), (0.6;0.6) and (0.4;0.6). Around this square we take squares with width = 0.1 on each side, so we get four squares. The second square from inside has depth 3500, de third one a depth of 2000, then 500 en the outer one has depth 150.

Because the former test results give a good impression of the behavior of the number of iterations through the different results, we look here to the number of iterations for CG, ICCG(0) and IP and compare these results with DPCG with diagonal scaling and IP squares with dimensions 2×2. For the 100×100 problem this means that there are 25 squares and for the 900×900 this means we have 225 squares.

Dimension	CG	ICCG(0)	IP	DPCG(D)	DPCG(IPP)
9x9	6	3	6		
100x100	46	11	33	10	25
900x900	180	26	100	15	44

Table 19: all results for several algorithms applied to the homogeneous Nemo equation with variable H(2).

Note that these results do not differ very much from the results for the first variable H.

A 3600x3600 problem

To test the algorithms on a more realistic problem, the x-direction and y-direction of the unit square are divided into 60 subintervals. This results in a 3600x3600 matrix A. What about the number of iterations for this problem for the different algorithms used before? See the table below:

Dimension	CG	ICCG(0)	IP	DPCG(D)	DPCG(IPP)
3600x3600	370	49	201	15	53

Table 20: results for several algorithms applied to the homogeneous Nemo equation with variable H(2) for the 3600x3600 problem.

The results are in line with the former ones. DPCG with the diagonal scaling gives the best results.

11.5 Conclusions

A significant difference is noticed between the constant H and a variable H in general. When trying different variants of a variable H, the number of iterations do not vary that much. Compared to the Conjugate Gradient Algorithm, deflation can decrease the number of iterations till almost 70%. When involving the variable H, note that the diagonal scaling for the DPCG is much more faster that the IP preconditioner.

Chapter 12 Conclusions

In order to solve the Nemo equation:

$$\frac{\partial}{\partial x}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial x}\right)+\frac{\partial}{\partial y}\left(\frac{1}{H}\frac{\partial \psi_t}{\partial y}\right)=\frac{\partial \bar{M}_2}{\partial x}-\frac{\partial \bar{M}_1}{\partial y}$$

for $\frac{\partial \psi}{\partial t}$, the homogeneous variant on the unit square is tested by applying several iterative methods.

Starting with the Conjugate Gradient (CG) Algorithm, the convergence speed is upgraded by using preconditioners. The first one used is the Incomplete Cholesky (ICCG(0)) decomposition. This preconditioner gives very good results, but it is not parallelizable. On purpose to perform the numerical computations on several computer units, we tried to find algorithms and preconditioners that are suitable for parallel computations and have a faster convergence rate as well. The Incomplete Poisson Preconditioner is a useful preconditioner for this goal, because the results are good and the preconditioner is highly parallel. The degree of parallelism for this preconditioner is n for a $n \times n$ matrix A. The results are not as well as for the ICCG(0), but far more better than the CG algorithm.

Applying Deflation to the Preconditioned Conjugate Gradient Algorithm decreases the number of iterations. This is done for the various test problems, for both the diagonal scaling and Incomplete Poisson Preconditioner.

When H is variable, it is observed that the Deflated Conjugate Gradient Algorithm gives the best results when diagonal scaling is used for preconditioning. When dividing the computational domain into subdomains on purpose to use DPCG, the best way to do this is by dividing into squares.

Test are done for the homogeneous variant of the Nemo equation, but the results for the non-homogeneous variant for the real domain are not expected to differ from the results of our test problems.

Chapter 13 Further Research

The aim of Nemo is to improve the scalability of Nemo for a large amount of cores (100k+). Since we do not have results of the Deflated Preconditioned Conjugate Gradient algorithm for a large amount of computational units, the behavior of the algorithm for these kind of architectures should be explored.

Secondly, for this research, variable values for the function H are taken. Maybe more realistic values for this function are available for testing the algorithms.

References

[1] Saad, Y. (1996), '***Iterative methods for sparse linear systems'***

[2] Reader Scientific Computing, 2005 Delft Institute of Applied Mathematics.

[3] http://www.nemo-ocean.eu/

[4] Gupta, R. (2010), ***'Implementation of the Deflated Preconditioned Conjugate Gradient Method for Bubbly Flow on the Graphical Processing Unit (GPU)'***, Delft University of Technology

[5] Kan, J. van; Segal, A.; Vermolen, F. (2004), ***'Numerical Methods in Scientific Computing'***, Delft