

Faculty of Electrical Engineering, Mathematics, and  
Computer Science  
Delft University of Technology

**Implementation of the Deflated Pre-conditioned  
Conjugate Gradient Method applied to the  
Poisson equation related to bubbly flow on the  
GPU**

Literature Study

Rohit Gupta 1542702

Supervisors: Prof.dr.ir. C. Vuik,  
Ir. C.W.J. Lemmens

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Problem Definition</b>	<b>2</b>
<b>3</b>	<b>Iterative Solution Methods</b>	<b>3</b>
3.1	Discretization of Partial Differential Equations . . . . .	3
3.1.1	Finite Difference Method . . . . .	4
3.1.2	Finite Element Method . . . . .	5
3.1.3	Finite Volume Method . . . . .	8
3.2	Basic Iterative Methods at a Glance . . . . .	10
3.2.1	JACOBI . . . . .	10
3.2.2	GAUSS-SEIDEL . . . . .	10
3.2.3	Block Relaxation Schemes . . . . .	11
3.2.4	Preconditioning . . . . .	14
3.2.5	Convergence . . . . .	14
3.3	Conjugate Gradient . . . . .	15
3.3.1	Arnoldi Orthogonalization . . . . .	15
3.3.2	Lanczos Method . . . . .	16
3.3.3	Conjugate Gradient Algorithm . . . . .	17
3.3.4	Preconditioning applied to Conjugate Gradient . . . . .	19
3.4	Preconditioning Techniques . . . . .	21
3.4.1	ILUT approach and implementation issues . . . . .	26
3.5	Domain Decomposition . . . . .	28
3.5.1	Direct Solution and Schur Complement . . . . .	30
3.5.2	Schwarz Alternating Procedures . . . . .	35
3.5.3	Schur Complement Approaches . . . . .	41
3.6	Deflation . . . . .	44
3.6.1	Subdomain Deflation . . . . .	46
<b>4</b>	<b>Parallel Iterative Methods</b>	<b>46</b>
4.1	Parallel Implementations . . . . .	47
4.1.1	Matrix-Vector Products . . . . .	47
4.1.2	Level Scheduling: The case of 5-Point Matrices . . . . .	50
4.2	Preconditioning in Parallel . . . . .	50
4.2.1	Multi-Coloring . . . . .	50
4.2.2	Multi-Elimination ILU . . . . .	52
<b>5</b>	<b>Scientific Computing on GPUs</b>	<b>53</b>
5.1	Background Information on GPU Architecture and Programming Model . . . . .	53
5.1.1	Internal Organization . . . . .	54
5.1.2	Execution of Threads . . . . .	54
5.1.3	Memory Model . . . . .	54
5.1.4	Usability for Scientific Computing . . . . .	56
<b>6</b>	<b>Parallel Iterative Methods on the GPU</b>	<b>56</b>
6.1	Mixed Precision Techniques on the GPU . . . . .	56
6.2	Sparse Matrix Vector Products- SpMV's . . . . .	56
6.3	Conjugate Gradient . . . . .	58
6.4	Preconditioning . . . . .	58
6.5	Multi-GPU Implementations . . . . .	60

<b>7</b>	<b>Research Questions</b>	<b>61</b>
7.1	Conjugate Gradient on GPU vs CPU . . . . .	61
7.2	Preconditioning Approaches . . . . .	61
7.2.1	Preconditioned Conjugate Gradient on GPU vs CPU . . . . .	61
7.2.2	Diagonal . . . . .	61
7.2.3	IC(0) . . . . .	62
7.2.4	MultiGrid . . . . .	62
7.2.5	IP Preconditioning . . . . .	62
7.3	Deflation Approaches . . . . .	62
7.3.1	Deflated PCG on GPU vs CPU . . . . .	62
7.3.2	DPCG on Many GPUs . . . . .	62
7.4	Precision Statistics . . . . .	62
<b>8</b>	<b>Preliminary Results</b>	<b>62</b>
8.1	Conjugate Gradient - Basic Version . . . . .	62

# 1 Introduction

In this work we look at the implementation of a numerical solution of a linear Partial Differential Equation(PDE), resulting from the mathematical modeling of physical systems and in particular bubbly flows. The PDEs have been discretized through the use of finite elements. The linear systems we are interested in are of the form

$$Ax = b, A \in \mathbb{R}^{n \times n}, n \in \mathbb{N} \quad (1)$$

that arise from such discretizations, where  $n$  is the number of degrees of freedom and is also called the dimension of  $A$ . Also  $A$  is symmetric positive definite (SPD), i.e.,

$$A = A^T, y^T Ay > 0 \forall y \in \mathbb{R}^{n \times n}, y \neq 0. \quad (2)$$

The linear system given by (1) is usually sparse and ill-conditioned. This means that there are few non-zero elements per row of  $A$  and also that the condition number  $\kappa$  is usually large. Put in other words, the ratio of the largest eigenvalue to the smallest is large and this leads to slow convergence of an iterative method.

$$\kappa\{A\} = \frac{\lambda_n}{\lambda_1} \quad (3)$$

where  $0 \leq \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$  are eigenvalues of matrix  $A$ .

Solving the system (1) by direct methods is also an option but it is usually not memory-wise or computationally efficient. Though these methods are robust and generally applicable but they also tend to be prohibitively expensive. The sparsity of the matrix  $A$  necessitates the use of efficient storage methods and computation with 'iterative methods'. The term 'iterative method' refers to a wide range of techniques that use iterates, or successive approximations to obtain more accurate solutions to a linear system at each iteration step.

Krylov subspace methods, especially the Conjugate Gradient Method is the prominent choice for solving such systems. However the convergence of these methods depends heavily on  $\kappa(A)$ . In order to avoid more and more iterations, as  $\kappa(A)$  rises with increasing problem sizes, the matrix  $A$  is *preconditioned* to bring down the condition number from  $\kappa(A)$  to  $\kappa(M^{-\frac{1}{2}}AM^{\frac{1}{2}})$  which is equivalent to  $\kappa(M^{-1}A)$ . The coefficient matrix  $A$  is multiplied by  $M^{-1}$ . The original system (1) then looks like,

$$M^{-1}Ax = M^{-1}b, \quad (4)$$

where  $M$  is symmetric and positive definite just like  $A$ .  $M^{-1}$  is chosen in such a way that the cost of the operation  $M^{-1}y$  with a vector  $y$  is computationally cheap. However, sometimes preconditioning might also not be enough. In that case we use second level of preconditioning or Deflation in order to reduce  $\kappa(A)$ .

In this work some previous results [Tang, 2008] are used for implementation. The focus is to implement these methods on the Graphical Processing Unit (GPU).

Recently Scientific Computing has largely benefitted from the data parallel architecture of graphical processors. Many interesting problems which are computationally intensive are ideally suited to the GPU. Especially matrix calculations. It is only intuitive to use them for solution of discretized partial equations. With the advent of the Component Unified Device Architecture (CUDA) paradigm of computing available on NVIDIA GPU devices, it has become easier to write such applications. More time can be spent in exploring the 'What If?' scenarios that are of scientific importance rather than understading device specifics. We briefly define the problem of Bubbly flows followed by a background on Iterative Methods. Subsequently we introduce the Parallelization of Iterative methods

that are used on cluster machines and coded in MPI, HPF and such technologies. After introducing some of the Architectural details of the GPU and a glimpse of the programming techniques we discuss the recent work that has been done on the GPU with respect to solving iteratively the linear systems emerging from discretizations of PDEs. We conclude this document by presenting the research questions that the implementation will explore and try to answer.

## 2 Problem Definition

Computations of Bubbly flows is the main application for this implementation. Understanding the dynamics and interaction of bubbles and droplets in a large variety of processes in nature, engineering, and industry are crucial for economically and ecologically optimized design. Bubbly flow occur, for example, in chemical reactors, boiling, fuel injectors, coating and volcanic eruptions.

Two phase flows are complicated to simulate, because the geometry of the problem typically varies with time, and the fluids involved have very different material properties. Following from the previous work [Tang, 2008] we consider stationary and time-dependent bubbly flows, where the computational domain is always a unit square or unit cube filled with a fluid to a certain height. The bubbles and droplets in the domain are always chosen such that they are located in a structured way and have equal radius,  $s$ , at the starting time.

Mathematically bubbly flows are modelled using the Navier Stokes equations including boundary and interface conditions, which can be approximated numerically using operator splitting techniques. In these schemes, equations for the velocity and pressure are solved sequentially at each time step. In many popular operator-splitting methods, the pressure correction is formulated implicitly, requiring the solution of a linear system (1) at each time step. This system takes the form of a Poisson equation with discontinuous coefficients (also called the 'pressure(-correction) equation') and Neumann boundary conditions, i.e.,

$$-\nabla \cdot \left( \frac{1}{\rho(x)} \nabla p(x) \right) = f(x), x \in \Omega, \quad (5)$$

$$\frac{\partial p(x)}{\partial n} = g(x), x \in \partial\Omega, \quad (6)$$

where  $\Omega, p, \rho, x$  and  $n$  denote the computational domain, pressure, density, spatial coordinates, and the unit normal vector to the boundary,  $\partial\Omega$ , respectively. Right-hand sides  $f$  and  $g$  follow explicitly from the operator-splitting method, where  $g$  is such that mass is conserved, leading to a singular but compatible linear system (1).

A typical sequence of steps for Deflated Preconditioned CG algorithm can be outlined here as described in [Tang, 2008]. This method is numerically more stable although it is derived from the work discussed in [Saad, Yeung, Erhel, and Guyomarc'h, 2000].

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  $j:=0, \dots$ , until convergence do
3.  $\hat{w}_j := PAp_j$
4.  $\alpha_j := \frac{(\hat{r}_j, y_j)}{(p_j, \hat{w}_j)}$
5.  $\hat{x}_{j+1} := \hat{x}_j + \alpha_j p_j$
6.  $\hat{r}_{j+1} := \hat{r}_j - \alpha_j \hat{w}_j$

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.  $x_{it} := Qb + P^T x_{j+1}$

$A$  is the coefficient matrix.  $M$  is the preconditioning matrix.  $r_j$  is the residual at  $j^{th}$  step and  $p_j$  is the new search direction every step.  $x_j$  is the solution we seek for the linear system  $Ax = b$ .

Solving the linear system (1), that is a discretization of (5), within an operator splitting approach is a bottleneck in the fluid-flow simulation, since it typically consumes the bulk of the computing time. In order to accelerate the convergence of the iterative Preconditioned Conjugate Gradient Algorithm with deflation we propose to use the GPU. The challenge lies in optimizing the computation on the GPU in such a way so as to extract the maximum computation throughput it can deliver. We discuss some of the challenges involved in the Research Questions section of this document.

### 3 Iterative Solution Methods

In this section a brief introduction of the discretization methods for Partial Differential equations are discussed followed by a glimpse of the basic iterative methods.

#### 3.1 Discretization of Partial Differential Equations

To solve differential equations a suitable approximation must be taken. Expressing them in the form of equations involving a finite number of unknowns, we can translate them into a problem of solving a sparse linear system. The discretization involves some boundary conditions.

Consider the example of the Poisson's Equation:

$$\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} = f, \text{ for } \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \Omega \quad (7)$$

where  $\Omega$  is a bounded, open domain in  $\mathbb{R}^2$ . Here,  $x_1, x_2$  are the two space variables. The above equation is to be satisfied only for points that are located at the interior of the

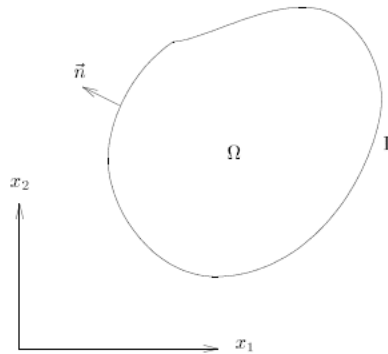


Figure 1: Domain  $\Omega$  for Poisson's Equation

domain  $\Omega$ . Equally important are the conditions that must be satisfied on the boundary  $\Gamma$  of  $\Omega$ . These are called *boundary conditions*. Three common types of boundary conditions are:

$$\text{Dirichlet Condition} \quad u(x) = \phi(x) \quad (8)$$

$$\text{Neumann Condition} \quad \frac{\partial u}{\partial \vec{n}}(x) = 0 \quad (9)$$

$$\text{Cauchy Condition} \quad \frac{\partial u}{\partial \vec{n}}(x) + \alpha(x)u(x) = \gamma(x) \quad (10)$$

The vector  $\vec{n}$  refers to a unit vector that is normal to  $\Gamma$  and directed outwards. For a given vector  $\vec{v}$ , with components  $v_1$  and  $v_2$ , the directional derivative  $\frac{\partial u}{\partial \vec{v}}$  is defined by

$$\frac{\partial u}{\partial \vec{v}}(x) = \lim_{h \rightarrow 0} \frac{u(x + h\vec{v}) - u(x)}{h} \quad (11)$$

$$= \frac{\partial u}{\partial x_1}(x)v_1 + \frac{\partial u}{\partial x_2}(x)v_2 \quad (12)$$

$$= \nabla u \cdot \vec{v} \quad (13)$$

### 3.1.1 Finite Difference Method

The finite difference method is based on local approximations of the partial derivatives in a Partial Differential Equation, which are derived by low order Taylor series expansions. It is particularly simple for uniform meshes. There are a number of "fast solvers" for constant coefficient problems, which can deliver solution in a logarithmic time per grid point.

The simplest way to approximate the first derivative of a function  $u$  at the point  $x$  is via the formula

$$\left( \frac{du(x)}{dx} \right) \approx \frac{u(x+h) - u(x)}{h}. \quad (14)$$

When  $u$  is differentiable at  $x$ , then the limit of the above ratio when  $h$  tends to zero is the derivative of  $u$  at  $x$ . For a function whose fourth derivative  $\frac{d^4 u}{dx^4}$  exists in the neighborhood of  $x$ , we have by Taylor's Formula

$$u(x+h) = u(x) + h \frac{du}{dx} + \frac{h^2}{2} \frac{d^2 u}{dx^2} + \frac{h^3}{6} \frac{d^3 u}{dx^3} + \frac{h^4}{24} \frac{d^4 u}{dx^4}(\xi_+) \quad (15)$$

for some  $\xi_+$  plus in the interval  $(x, x+h)$ . Therefore the above approximation (14) satisfies

$$\frac{du}{dx} = \frac{u(x+h) - u(x)}{h} + \frac{h}{2} \frac{d^2 u}{dx^2} + O(h^2). \quad (16)$$

replacing by  $-h$  in (15) we get

$$u(x-h) = u(x) - h \frac{du}{dx} + \frac{h^2}{2} \frac{d^2 u}{dx^2} - \frac{h^3}{6} \frac{d^3 u}{dx^3} + \frac{h^4}{24} \frac{d^4 u}{dx^4}(\xi_-) \quad (17)$$

adding them up we get the centered difference approximation of the second derivative. The dependence of this derivative on values of  $u$  at the points involved in the approximation is often represented by a "stencil".

The approximation shown first (14) is forward rather than centered. Also a backward approximation could be devised by replacing  $h$  by  $-h$  in that equation. The two formulas can be merged to get the *centered difference* formula:

$$\frac{du(x)}{dx} \approx \frac{u(x+h) - u(x-h)}{2h} \quad (18)$$

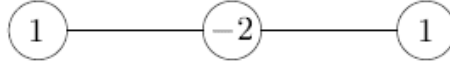


Figure 2: The three point stencil for the centered difference approximation to the second order derivative

### Finite Differences for 1-D problems

Consider the one-dimensional equation

$$-u''(x) = f(x) \text{ for } x \in (0, 1) \quad (19)$$

$$u(0) = u(1) = 0 \quad (20)$$

The interval  $[0, 1]$  can be discretized uniformly by taking  $n + 2$  points

$$x_i = i \times h, i = 0, \dots, n + 1 \quad (21)$$

where  $h = \frac{1}{n+1}$ . Because of the Dirichlet boundary conditions, the values at  $u(x_0)$  and  $u(x_{n+1})$  are known. At every other point, approximation of  $u_i$  is sought for the exact solution  $u(x_i)$ .

If centered difference approximation (17) is used, then by (19) expressed at the point  $x_i$ , the unknowns  $u_i, u_{i-1}, u_{i+1}$  satisfy the relation

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 f_i \quad (22)$$

### Finite Differences for 2-D problems

Similar to the previous case, consider a simple problem in 2D.

$$\left( \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} \right) = f \text{ in } \Omega \quad (23)$$

$$u = 0 \text{ on } \Gamma \quad (24)$$

where  $\Omega$  is now the rectangle  $(0, l_1) \times (0, l_2)$  and  $\Gamma$  its boundary. Both directions can be discretized uniformly in both dimensions by taking  $n_1 + 2$  and  $n_2 + 2$  points in each direction  $(x_1, x_2)$ .

Hence the function is defined on

$$x_{1,i} = i \times h_1, i = 0, \dots, n_1 + 1, x_{2,j} = j \times h_2, j = 0, \dots, n_2 + 1 \quad (25)$$

where

$$h_1 = \frac{l_1}{n_1 + 1}, h_2 = \frac{l_2}{n_2 + 1} \quad (26)$$

### 3.1.2 Finite Element Method

The finite element method is best suited for handling complex geometries (and boundaries) with relative ease. The finite element method is best illustrated with the solution of a simple Partial Differential Equation in a two dimensional space. Consider again Poisson's equation (7) with the Dirichlet boundary condition (8) where  $\Omega$  is a bounded open domain in  $\mathbb{R}^2$  and  $\Gamma$  its boundary. Then the Laplacian Operator can be defined as.

$$\Delta = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} \quad (27)$$

Now by the virtue of the Greens theorem (proof dicussed in [Saad, 1996] ) we have

$$\int_{\Omega} \nabla v \cdot \nabla u dx = - \int_{\Omega} v \Delta u dx + \int_{\Gamma} v \frac{\partial u}{\partial \vec{n}} ds. \quad (28)$$

For the system we want to solve in (23) we must take approximations of  $u$  and that over a finite dimensional space. Another consideration is that we must be able to solve the system numerically with this formulation. A weak formulation, to extract a system of equations that yield the solution of the problem, is in place.

$$a(u, v) \equiv \int_{\Omega} \nabla u \cdot \nabla v dx = \int_{\Omega} \left( \frac{\partial u}{\partial x_1} \frac{\partial v}{\partial x_1} + \frac{\partial u}{\partial x_2} \frac{\partial v}{\partial x_2} \right) dx, \quad (29)$$

$$(f, v) \equiv \int_{\Omega} f v dx. \quad (30)$$

The weak formulation of the initial problem consists of selecting a subspace of reference  $V$  of  $L^2$  and then defining the following problem.

$$\text{Find } u \in V \text{ such that } a(u, v) = (f, v), \forall v \in V. \quad (31)$$

In order to understand the usual choices for the space  $V$  , the definition of the weak problem only requires the dot products of the gradients of  $u$  and  $v$  and the functions  $f$  and  $v$  to be  $L_2$  integrable. The most general  $V$  under these conditions is the space of all functions whose derivatives up to the first order are in  $L_2$ . This is known as  $H^1(\Omega)$ . However, this space does not take into account the boundary conditions. This however is tru only in case of homogenous Dirichlet Boundary Conditions. The functions in  $V$  must be restricted to have zero values on  $\Gamma$ . The resulting space is called  $H_0^1(\Omega)$ . The finite element method consists of approximating the weak problem by a finite dimensional problem obtained by replacing  $V$  with a subspace of functions that are defined as low-degree polynomials on small pieces (elements) of the original domain.

Consider a region  $\Omega$  in the plane which is triangulated as shown in Figure 3

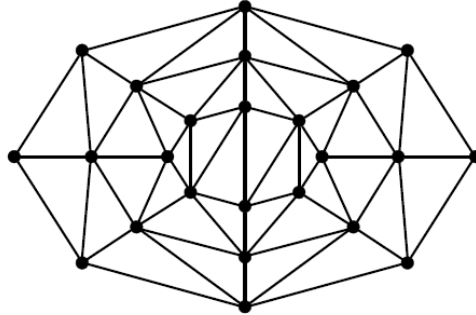


Figure 3: Finite Element triangulation of a domain

In this example, the domain is simply an ellipse but the external enclosing curve is not shown. The original domain is thus approximated by the union  $\Omega_h$  of  $m$  triangles  $K_i$ ,

$$\Omega_h = \bigcup_{i=1}^m K_i \quad (32)$$

For the triangulation to be valid, these triangles must have no vertex that lies on the edge of any other triangle. The *mesh size*  $h$  is defined by

$$h = \max_{i=1,\dots,m} \text{diam}(K_i) \quad (33)$$

where  $\text{diam}(K)$ , the diameter of a triangle  $K$ , is the length of its longest side. Then the finite dimensional space  $V_h$  is defined as the space of all functions which are piecewise linear and continuous on the polygonal region  $\Omega_h$ , and which vanish on the boundary  $\Gamma$ . More specifically,

$$V_h = \{\phi|_{\Omega_h}, \text{continuous}, \phi|_{\Gamma_h} = 0, \phi_{K_j} \text{ linear } \forall j\}. \quad (34)$$

Here,  $\phi|_X$  represents the restriction of the function  $\phi$  to the subset  $X$ . If  $x_j, j = 1, \dots, n$  are the nodes of the triangulation, then a function  $\phi_j$  in  $V_h$  can be associated with each node  $x_j$ , so that the family of functions  $\phi_j$ s satisfies the following conditions:

$$\phi_j(x_i) = \delta_{ij} = \begin{cases} 1 & \text{if } x_i = x_j \\ 0 & \text{if } x_i \neq x_j \end{cases} \quad (35)$$

$$\left. \begin{matrix} \\ \\ \end{matrix} \right\} \quad (36)$$

These conditions define  $\phi_i, i = 1, \dots, n$  uniquely. In addition, the  $\phi_i$ s form a basis of the space  $V_h$ . Each function of  $V_h$  can be expressed as

$$\phi(x) = \sum_{i=1}^n \xi_i \phi_i(x). \quad (37)$$

The finite element approximation consists of writing the Galerkin condition (31) for functions in  $V_h$ . This defines the approximate problem:

$$\text{Find } u \in V_h \text{ such that } a(u, v) = f(u, v), \forall v \in V_h. \quad (38)$$

Since  $u$  is in  $V_h$ , there are  $n$  degrees of freedom. By the linearity of  $a$  with respect to  $v$ , it is only necessary to impose the condition  $a(u, \phi_i) = (f, \phi_i)$  for  $i = 1, \dots, n$ . This results in  $n$  constraints. Writing the desired solution  $u$  in the basis  $\{\phi_i\}$  as

$$u = \sum_{i=1}^n \xi_i \phi_i(x) \quad (39)$$

and substituting in (38) gives the linear problem

$$\sum_{i=1}^n \alpha_{ij} \xi_i = \beta_j \quad (40)$$

where

$$\alpha_{ij} = a(\phi_i, \phi_j), \beta_j = (b, \phi_j). \quad (41)$$

The above equations form a linear system of equations

$$Ax = b \quad (42)$$

in which the coefficients of  $A$  are the  $\alpha_{ij}$ s; those of  $b$  are the  $\beta_j$ s.

### 3.1.3 Finite Volume Method

The finite volume method is geared toward the solution of conservation laws of the form:

$$\frac{\partial u}{\partial t} + \nabla \cdot \vec{F} = Q. \quad (43)$$

In the above equation,  $\vec{F}(u, t)$  is a certain vector function of  $u$  and time, possibly nonlinear. This is called the flux vector. The source term  $Q$  is a function of space and time. We now apply the principle used in the weak formulation, described before. Multiply both sides by a test function  $w$ , and take the integral

$$\int_{\Omega} w \frac{\partial u}{\partial t} dx + \int_{\Omega} w \nabla \cdot \vec{F} dx = \int_{\Omega} w Q dx. \quad (44)$$

Then integrate by part for the second term on the left-hand side to obtain

$$\int_{\Omega} w \frac{\partial u}{\partial t} dx - \int_{\Omega} \nabla w \cdot \vec{F} dx + \int_{\Gamma} w \vec{F} \cdot \vec{n} ds = \int_{\Omega} w Q dx. \quad (45)$$

Consider now a control volume consisting, for example, of an elementary triangle  $K_i$  in the two-dimensional case, such as those used in the finite element method. Take for  $w$  a function  $w_i$  whose value is one on the triangle and zero elsewhere. The second term in the above equation vanishes and the following relation results:

$$\int_{K_i} \frac{\partial u}{\partial t} dx + \int_{\Gamma_i} \vec{F} \cdot \vec{n} ds = \int_{K_i} Q dx \quad (46)$$

The above relation is the basis of the finite volume approximation. To go a little further, the assumptions will be simplified slightly by taking a vector function  $\vec{F}$  that is linear with respect to  $u$ . Specifically, assume

$$\vec{F} = \begin{pmatrix} \lambda_1 u \\ \lambda_2 u \end{pmatrix} \equiv \vec{\lambda} u. \quad (47)$$

Note that, in this case, the term  $\nabla \cdot \vec{F}$  in (43) becomes  $\vec{F}(u) = \vec{\lambda} \cdot \nabla u$ . In addition, the right-hand side and the first term in the left-hand side of (46) can be approximated as follows:

$$\int_{K_i} \frac{\partial u}{\partial t} dx \approx \frac{\partial u_i}{\partial t} |K_i|, \quad \int_{K_i} Q dx \approx q_i |K_i|. \quad (48)$$

Here,  $|K_i|$  represents the volume (in two dimensional volume is considered to mean area) of  $K_i$ , and  $q_i$  is some average value of  $Q$  in the cell  $K_i$ . These are crude approximations but they serve the purpose of illustrating the scheme. The finite volume (45) yields

$$\frac{\partial u_i}{\partial t} |K_i| + \vec{\lambda} \cdot \int_{\Gamma_i} u \vec{n} ds = q_i |K_i|. \quad (49)$$

The contour integral

$$\int_{\Gamma_i} u \vec{n} ds \quad (50)$$

is the sum of the integrals over all edges of the control volume. Let the value of  $u$  on each edge  $j$  be approximated by some average  $\bar{u}_j$ . In addition,  $s_j$  denotes the length of each edge and a common notation is

$$\vec{s}_j = s_j \vec{n}_j. \quad (51)$$

Then the contour integral is approximated by

$$\vec{\lambda} \cdot \int_{\Gamma_i} u \vec{n} ds \approx \sum_{edges} \bar{u}_j \vec{\lambda} \cdot \vec{n} s_j = \sum_{edges} \bar{u}_j \vec{\lambda} \cdot \vec{s}_j. \quad (52)$$

The situation in the case where the control volume is a simple triangle is depicted in Figure 4

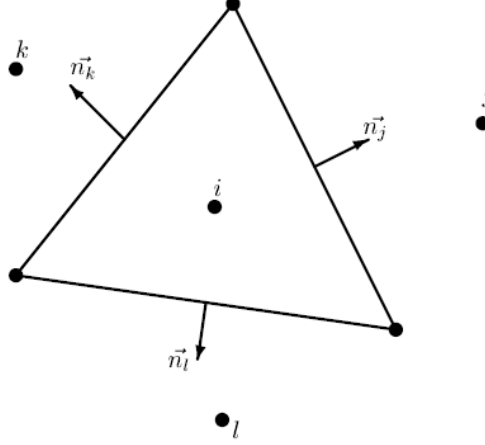


Figure 4: Finite Volume Cell associated with node  $i$  and neighboring cells.

The unknowns are the approximations  $u_i$  of the function  $u$  associated with each cell. These can be viewed as approximations of  $u$  at the centers of gravity of each cell  $i$ . This type of model is called *cell-centered* finite volume approximations.

The value  $\bar{u}_j$  required in (52) can be taken simply as the average between the approximation  $u_i$  of  $u$  in cell  $i$  and the approximation  $u_j$  in the cell  $j$  on the other side of the edge

$$\bar{u}_j = \frac{1}{2}(u_j + u_i). \quad (53)$$

This gives

$$\frac{\partial u_i}{\partial t} |K_i| + \frac{1}{2} \sum_j (u_i + u_j) \vec{\lambda} \cdot \vec{s}_j = q_i |K_i|. \quad (54)$$

One further simplification takes place by observing that

$$\sum_j \vec{s}_j = 0 \quad (55)$$

and therefore

$$\sum_j u_i \vec{\lambda} \cdot \vec{s}_j = u_i \vec{\lambda} \cdot \sum_j \vec{s}_j = 0. \quad (56)$$

This yields

$$\frac{\partial u_i}{\partial t} |K_i| + \frac{1}{2} \sum_j u_j \vec{\lambda} \cdot \vec{s}_j = q_i |K_i|. \quad (57)$$

In the above equation, the summation is over all neighboring cells  $j$ . One problem with such simple approximations is that they do not account for large gradients of  $u$  in the components. In finite volume approximations, it is typical to exploit upwind schemes which are more suitable in such cases.

### 3.2 Basic Iterative Methods at a Glance

Considering the methods to approximate a solution to the system

$$Ax = b \quad (58)$$

where  $x$  is an unknown vector,  $b$  is a known vector, and  $A$  is a known matrix of coefficients. To begin we consider two methods that are most notable.

These methods might take a large number of iterations to converge to a solution and might not be useful in many cases. However, convergence for finite difference discretization of Elliptic Partial Differential Equations has been extensively studied.

#### 3.2.1 JACOBI

The Jacobi iteration is based on the idea of splitting up  $A$  into  $D$  and  $E$  and  $F$ .

$$A = D - E - F \quad (59)$$

in which  $D$  is the diagonal of  $A$ ,  $-E$  its strict lower part, and  $-F$  its strict upper part, It is always assumed that the diagonal entries of  $A$  are all nonzero.

The Jacobi iteration determines the  $i$ -th component of the next approximation so as to annihilate the  $i$ -th component of the residual vector. In the following,  $\xi_i$  denotes the  $i$ -th component of the iterate  $x_k$  and  $\beta_i$  the  $i$ -th component of the right-hand side  $b$ . Thus, writing

$$(b - Ax_{k+1})_i = 0 \quad (60)$$

in which  $(y)_i$  represents the  $i$ -th component of the vector  $y$ , yields

$$a_{ii}\xi_i^{(k+1)} = - \sum_{j=1, j \neq i}^n a_{ij}\xi_j^{(k)} + \beta_i, \quad (61)$$

or

$$\xi_i^{(k+1)} = \frac{1}{a_{ii}} \left( \beta_i - \sum_{j=1, j \neq i}^n a_{ij}\xi_j^{(k)} \right) \quad i = 1, \dots, n \quad (62)$$

This is a component-wise form of the Jacobi iteration. All components of the next iterate can be grouped into the vector  $x_{k+1}$ . The above notation can be used to rewrite the Jacobi iteration (62) in vector form as

$$x_{k+1} = D^{-1}(E + F)x_k + D^{-1}b \quad (63)$$

#### 3.2.2 GAUSS-SEIDEL

The Gauss-Seidel iteration corrects the  $i$ -th component of the current approximate solution, in the order  $i = 1, 2, \dots, n$ , again to annihilate the  $i$ -th component of the residual. However, this time the approximate solution is updated immediately after the new component is determined. The newly computed components  $\xi_i^{(k)}$ ,  $i = 1, 2, \dots, n$  can be changed within a working vector which is redefined at each relaxation step. Thus, since the order is  $i = 1, 2, \dots, n$ , the result at the  $i$ -th step is

$$\beta_i - \sum_{j=1}^{i-1} a_{ij}\xi_j^{(k+1)} - a_{ii}\xi_i^{(k+1)} - \sum_{j=i+1}^n a_{ij}\xi_j^{(k)} = 0, \quad (64)$$

which leads to the iteration,

$$\xi_i^{(k+1)} = \frac{1}{a_{ii}} \left( - \sum_{j=1}^{i-1} a_{ij} \xi_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} \xi_j^{(k)} + \beta_i \right), i = 1, \dots, n \quad (65)$$

the defining equation (64) can be written as

$$b + Ex_{k+1} - Dx_{k+1} + Fx_k = 0, \quad (66)$$

which leads immediately to the vector form of the Gauss-Seidel iteration

$$x_{k+1} = (D - E)^{-1}Fx_k + (D - E)^{-1}b. \quad (67)$$

Computing the new approximation in (63) requires multiplying by the inverse of the diagonal matrix  $D$ . In (67) a trinaangular system must be solved with  $D - E$ , the lower triangular part of  $A$ . Thus, the new approximation in a Gauss-Seidel step can be determined either by solving a triangular system with the matrix  $D - E$  or from the relation (64).

A backward Gauss-Seidel iteration can also be defined as

$$(D - F)x_{k+1} = Ex_k + b, \quad (68)$$

which is equivalent to making the coordinate corrections in the order  $n, n - 1, \dots, 1$ . A Symmetric Gauss-Seidel Iteration consists of a forward sweep followed by a backward sweep. The Jacobi and the Gauss-Seidel iterations are both of the form

$$Mx_{k+1} = Nx_k + b = (M - A)x_k + b, \quad (69)$$

in which

$$A = M - N \quad (70)$$

is a splitting of  $A$ , with  $M = D$  for Jacobi,  $M = D - E$  for forward Gauss-Seidel, and  $M = D - F$  for backward Gauss-Seidel.

### 3.2.3 Block Relaxation Schemes

Block relaxation schemes are generalizations of the point relaxation schemes described above. They update a whole set of components at each time, typically a subvector of the solution vector, instead of only one component. The matrix  $A$  and the right-hand side and solution vectors are partitioned as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1p} \\ A_{21} & A_{22} & A_{23} & \cdots & A_{2p} \\ A_{31} & A_{32} & A_{33} & \cdots & A_{3p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & A_{p3} & \cdots & A_{pp} \end{pmatrix}, x = \begin{pmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \\ \vdots \\ \xi_p \end{pmatrix}, b = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \beta_p \end{pmatrix}, \quad (71)$$

in which the partitionings of  $b$  and  $x$  into subvectors  $\beta_i$  and  $\xi_i$  are identical and compatible with the partitioning of  $A$ . Thus, for any vector  $x$  partitioned as in (71),

$$(Ax)_i = \sum_{p_j=1} A_{ij} \xi_j, \quad (72)$$

in which  $(y_i)_i$  denotes the  $i$ -th component of the vector  $i$  according to the above partitioning. The diagonal blocks in  $A$  are square and assumed nonsingular. Now define, similarly to the scalar case, the splitting

$$A = D - E - F \quad (73)$$

with

$$D = \begin{pmatrix} A_{11} & & & \\ & A_{22} & & \\ & & \ddots & \\ & & & A_{pp} \end{pmatrix}, E = - \begin{pmatrix} O & & & \\ A_{21} & O & & \\ \vdots & \vdots & \ddots & \\ A_{p1} & A_{p2} & \cdots & O \end{pmatrix}, F = - \begin{pmatrix} O & A_{12} & \cdots & A_{1p} \\ & O & \cdots & A_{2p} \\ & & \ddots & \vdots \\ & & & O \end{pmatrix} \quad (74)$$

With these definitions, it is easy to generalize the previous two iterative procedures defined earlier, namely, Jacobi and Gauss-Seidel. For example, the block Jacobi iteration is now defined as a technique in which the new subvectors  $\xi_i^{(k)}$  are all replaced according to

$$A_{ii}\xi_i^{k+1} = ((E + F)x_k)_i + \beta_i \quad (75)$$

or,

$$\xi_i^{k+1} = A_{ii}^{-1}((E + F)x_k)_i + A_{ii}^{-1}\beta_i, i = 1, \dots, p, \quad (76)$$

which leads to the same equation as before,

$$x_{k+1} = D^{-1}(E + F)x_k + D^{-1}b, \quad (77)$$

except that the meanings of  $D$ ,  $E$ , and  $F$  have changed to their block analogues.

A general block Jacobi iteration can be defined as follows. Let  $V_i$  be the  $n \times n_i$  matrix

$$V_i = [e_{m_i(1)}, e_{m_i(2)}, \dots, e_{m_i(n_i)}] \quad (78)$$

and

$$W_i = [\eta_{m_i(1)}e_{m_i(1)}, \eta_{m_i(2)}e_{m_i(2)}, \dots, \eta_{m_i(n_i)}e_{m_i(n_i)}], \quad (79)$$

where each  $e_j$  is the  $j$ -th column of the  $n \times n$  identity matrix, and  $\eta_{m_i(j)}$  represents a weight factor chosen so that

$$W_i^T V_i = I \quad (80)$$

It must be noted in above that  $n_i$  denotes the size of  $V_i$ . When there is no overlap, i.e., when the  $S_i$ s form a partition of the whole set  $\{1, 2, \dots, n\}$ , then define  $\eta_{m_i(j)} = 1$ . Let  $A_{ij}$  be the  $n_i \times n_j$  matrix

$$A_{ij} = W_i^T A V_j \quad (81)$$

and define similarly the partitioned vectors

$$\xi_i = W_i^T x, \beta_i = W_i^T b. \quad (82)$$

Note that  $V_i W_i^T$  is a projector from  $\mathbb{R}^n$  to the subspace  $K_i$  spanned by the columns  $m_i(1), \dots, m_i(n_i)$ . In addition, we have the relation

$$x = \sum_{s_i=1} V_i \xi_i. \quad (83)$$

The  $n_i$  dimensional vector  $W_i^T x$  represents the projection  $V_i W_i^T x$  of  $x$  with respect to the basis spanned by the columns of  $V_i$ . The action of  $V_i$  performs the reverse operation. That means  $V_i y$  is an extension operation from a vector  $y$  in  $K_i$  (represented in the basis consisting of the columns of  $V_i$ ) into a vector  $V_i y$  in  $\mathbb{R}^n$ . The operator  $W_i^T$  is termed a *restriction operator* and  $V_i$  is an *prolongation operator*. Each component of the Jacobi

iteration can be obtained by imposing the condition that the projection of the residual in the span of  $S_i$  be zero, i.e.,

$$W_i^T \left[ b - A \left( V_i W_i^T x_{k+1} + \sum_{j \neq i} V_j W_j^T x_k \right) \right] = 0 \quad (84)$$

Remember that  $\xi_j = W_j^T x$ , which can be rewritten as

$$\xi_i^{k+1} = \xi_i^{(k)} + A_{ii}^{-1} W_i^T (b - Ax_k). \quad (85)$$

This leads to the following algorithm:

1. For  $K = 0, 1, \dots$ , until convergence Do:
2. For  $i = 1, 2, \dots, p$  Do:
3. Solve  $A_{ii} \delta_i = W_i^T (b - Ax_k)$ .
4. Set  $x_{k+1} := x_k + V_i \delta_i$
5. EndDo
6. EndDo

As was the case with the scalar algorithms, there is only a slight difference between the Jacobi and Gauss-Seidel iterations. Gauss-Seidel immediately updates the component to be corrected at step  $i$ , and uses the updated approximate solution to compute the residual vector needed to correct the next component. However, the Jacobi iteration uses the same previous approximation  $x_k$  for this purpose. Therefore, the block Gauss-Seidel iteration can be defined algorithmically as follows:

General Gauss Seidel Iteration

1. Until convergence Do:
2. For  $i = 1, 2, \dots, p$  Do:
3. Solve  $A_{ii} \delta_i = W_i^T (b - Ax)$
4. Set  $x := x + V_i \delta_i$
5. EndDo
6. EndDo

From the point of view of storage, Gauss-Seidel is more economical because the new approximation can be overwritten over the same vector. Also, it typically converges faster. On the other hand, the Jacobi iteration has some appeal on parallel computers since the second Do loop, corresponding to the different blocks, can be executed in parallel. Although the point Jacobi algorithm by itself is rarely a successful technique for real-life problems, its block Jacobi variant, when using large enough (overlapping) blocks, can be quite attractive especially in a parallel computing environment.

### 3.2.4 Preconditioning

The Jacobi and Gauss-Seidel iterations are of the form

$$x_{k+1} = Gx_k + f, \quad (86)$$

in which

$$G_{JA}(A) = I - D^{-1}A \quad (87)$$

$$G_{GS}(A) = I - (D - E)^{-1}A, \quad (88)$$

for the Jacobi and Gauss-Seidel iterations, respectively. Moreover, given the matrix splitting

$$A = M - N, \quad (89)$$

where  $A$  is associated with the linear system

$$Ax = b \quad (90)$$

a *linear fixed point iteration* can be defined by the recurrence

$$x_{k+1} = M^{-1}Nx_k + M^{-1}b \quad (91)$$

which has the form (86) with

$$G = M^{-1}N = M^{-1}(M - A) \quad (92)$$

$$= I - M^{-1}A \quad (93)$$

$$f = M^{-1}b. \quad (94)$$

For example, for the Jacobi iteration,  $M = D, N = A - D$ , while for Gauss-Seidel iteration,  $M = D - E, N = M - A = F$ .

The iteration  $x_{k+1} = Gx_k + f$  can be viewed as a technique for solving the system

$$(I - G)x = f \quad (95)$$

Since  $G$  has the form  $G = I - M^{-1}A$ , this system can be rewritten as

$$M^{-1}Ax = M^{-1}b \quad (96)$$

The above system which has the same solution as the original system is called a *preconditioned system* and  $M$  is the *preconditioning matrix* or a *preconditioner*. In other words, a *relaxation scheme is equivalent to a fixed point iteration on a preconditioned system*.

### 3.2.5 Convergence

The basic iterative methods discussed above define a sequence of iterates of the form

$$x_{k+1} = Gx_k + f, \quad (97)$$

in which  $G$  is a certain *iteration matrix*. If the above iteration converges, its limit  $x$  satisfies

$$x = Gx + f. \quad (98)$$

In the case where the above iteration arises from the splitting  $A = M - N$ , it is easy to see that the solution to the above system is identical to that of the original system  $Ax = b$ . Indeed, in this case the sequence (97) has the form

$$x_{k+1} = M^{-1}Nx_k + M^{-1}b \quad (99)$$

and its limit satisfies

$$Mx = Nx + b \quad (100)$$

or  $Ax=b$ . So convergence exists.

The conditions under which convergence happens can be summed up by saying that for any initial vector  $x_0$  the iteration (97) converges only if spectral radius of  $G$  is less than 1.

Also the convergence factor or rate of convergence is equal to the spectral radius of the iteration matrix  $G$ .

### 3.3 Conjugate Gradient

The Conjugate Gradient method is an important method for solving sparse linear systems. It is based on the idea of using a projection method on Krylov Subspaces  $\mathcal{K}_m$  to find an approximate solution  $x_m$  to

$$Ax = b \quad (101)$$

This is in turn done by imposing a Petrov Galerkin condition

$$b - Ax_m \perp \mathcal{L}_m, \quad (102)$$

where  $\mathcal{L}_m$  is another subspace of dimension  $m$ . Here,  $x_0$  represents an arbitrary initial guess to the solution. The subspace  $\mathcal{K}_m$  is written as

$$\mathcal{K}_m(A, r_0) = \text{span}\{r_0, Ar_0, A^2r_0, \dots, A^{m-1}r_0\}, \quad (103)$$

where  $r_0 = b - Ax_0$ .

When there is no ambiguity,  $\mathcal{K}_m(A, r_0)$  will be denoted by  $\mathcal{K}_m$ . The different versions of Krylov subspace methods arise from different choices of the subspace  $\mathcal{L}_m$  and from the ways in which the system is preconditioned.

Two broad choices for  $\mathcal{L}_m$  give rise to the best known techniques. The first is simply  $\mathcal{L}_m = \mathcal{K}_m$  and the minimum-residual variation  $\mathcal{L}_m = A\mathcal{K}_m$ .

An important assumption, for The Conjugate Gradient Method, is that the coefficient matrix  $A$  is Symmetric Positive Definite (SPD).

#### 3.3.1 Arnoldi Orthogonalization

The Arnoldi method is an orthogonal projection method onto  $\mathcal{K}_m$  for general non-Hermitian matrices. The procedure was introduced in 1951 as a means of reducing a dense matrix into Hessenberg form. Arnoldi presented his method in this manner but hinted that the eigenvalues of the Hessenberg matrix obtained from a number of steps smaller than  $n$  could provide accurate approximations to some eigenvalues of the original matrix. It was later discovered that this strategy leads to an efficient technique for approximating eigenvalues of large sparse matrices.

1. Choose a vector  $v_1$  of norm 1
2. For  $j = 1, 2, \dots, m$  Do:

3. Compute  $h_{ij} = (Av_j, v_i)$  for  $i = 1, 2, \dots, j$
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

At each step, the algorithm multiplies the previous Arnoldi vector  $v_j$  by  $A$  and then orthonormalizes the resulting vector  $w_j$  against all previous  $v_i$ s by a standard Gram-Schmidt procedure. It stops if the vector  $w_j$  vanishes.

### 3.3.2 Lanczos Method

The symmetric Lanczos algorithm can be viewed as a simplification of the Arnoldi method for the particular case when the matrix is symmetric. When  $A$  is symmetric, then the Hessenberg matrix  $H_m$  becomes symmetric tridiagonal. This leads to a three-term recurrence in the Arnoldi process and short-term recurrences for solution algorithms such as FOM and GMRES. On the theoretical side, there is also much more to be said on the resulting approximation in the symmetric case.

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

$$\alpha_j \equiv h_{jj}, \quad \beta_j \equiv h_{j-1,j}, \quad (104)$$

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

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & & \\ \beta_2 & \alpha_2 & \beta_3 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \beta_{m-1} & \alpha_{m-1} & \beta_m & \\ & & & \beta_m & \alpha_m & \end{pmatrix}. \quad (105)$$

This leads to the following form of 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.  $\alpha_j := (w_j, v_j)$
  5.  $w_j := w_j - \alpha_j v_j$
  6.  $\beta_{j+1} := \|w_j\|_2$ . If  $\beta_{j+1} = 0$  then Stop
  7.  $v_{j+1} := w_j/\beta_{j+1}$
8. EndDo

It is rather surprising that the above simple algorithm guarantees, at least in exact arithmetic, that the vectors  $v_i, i = 1, 2, \dots$ , are orthogonal. 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 global orthogonality rapidly. The major practical differences with the Arnoldi method are that the matrix  $H_m$  is tridiagonal and, more importantly, that only three vectors must be stored, unless some form of reorthogonalization is employed.

### 3.3.3 Conjugate Gradient Algorithm

The Conjugate Gradient method is a realization of an orthogonal projection technique onto the Krylov Subspace  $\mathcal{K}_m(A, r_0)$  where  $r_0$  is the initial residual.

First we derive the Arnoldi Method for the case when  $A$  is symmetric. Given an initial guess  $x_0$  to the linear system  $Ax = b$  and the Lanczos vectors  $v_i, i = 1, \dots, m$  together with the tridiagonal matrix  $T_m$ , the approximate solution obtained from an orthogonal projection method onto  $\mathcal{K}_m$ , is given by

$$x_m = x_0 + V_m y_m, y_m = T_m^{-1}(\beta e_1). \quad (106)$$

We now have the Lanczos method for linear systems

1. Compute  $r_0 = b - Ax_0$ ,  $\beta := \|r_0\|_2$ , and  $v_1 := r_0/\beta$
2. For  $j = 1, 1, \dots, m$  Do:
3.  $w_j = Av_j - \beta_j v_{j-1}$  ( If  $j = 1$  set  $\beta_1 v_0 \equiv 0$ )
4.  $\alpha_j = (w_j, v_j)$
5.  $w_j := w_j - \alpha_j v_j$
6.  $\beta_{j+1} = \|w_j\|_2$ . If  $\beta_{j+1} = 0$  set  $m := j$  and go to 9
7.  $v_{j+1} = w_j/\beta_{j+1}$
8. EndDo
9. Set  $T_m = \text{tridiag}(\beta_i, \alpha_i, \beta_{i+1})$ , and  $V_m = [v_1, \dots, v_m]$ .
10. Compute  $y_m = T_m^{-1}(\beta e_1)$  and  $x_m = x_0 + V_m y_m$

We can write  $LU$  factorization of  $T_m$  as  $T_m = L_m U_m$ . The matrix  $L_m$  is unit lower bidiagonal and  $U_m$  is upper bidiagonal. Thus, the factorization of  $T_m$  is of the form

$$T_m = \begin{pmatrix} 1 & & & & \\ \lambda_2 & 1 & & & \\ & \lambda_3 & 1 & & \\ & & \lambda_4 & 1 & \\ & & & \lambda_5 & 1 \end{pmatrix} \times \begin{pmatrix} \eta_1 & \beta_2 & & & \\ & \eta_2 & \beta_3 & & \\ & & \eta_3 & \beta_4 & \\ & & & \eta_4 & \beta_5 \\ & & & & \eta_5 \end{pmatrix}. \quad (107)$$

The approximate solution is then given by,

$$x_m = x_0 + V_m U_m^{-1} L_m^{-1}(\beta e_1). \quad (108)$$

Letting

$$P_m \equiv V_m U_m^{-1} \quad (109)$$

and

$$z_m = L_m^{-1} \beta e_1, \quad (110)$$

then,

$$x_m = x_0 + P_m z_m. \quad (111)$$

Note that,  $p_m$ , the last column of  $P_m$ , can be computed from the previous  $p_i$ 's and  $v_m$  by the simple update

$$p_m = \eta^{-1}[v_m - \beta_m p_{m-1}]. \quad (112)$$

Here  $\beta_m$  is a scalar computed from the Lanczos Algorithm, while  $\eta_m$  results from the  $m$ -th Gaussian Elimination step on the tridiagonal matrix, i.e.,

$$\lambda_m = \frac{\beta_m}{\eta_{m-1}}, \quad (113)$$

$$\eta_m = \alpha_m - \lambda_m \beta_m. \quad (114)$$

Also from the structure of  $L_m$  we have

$$z_m = \begin{bmatrix} z_{m-1} \\ \zeta_m \end{bmatrix}, \quad (115)$$

in which  $\zeta_m = -\lambda_m \zeta_{m-1}$ . As a result,  $x_m$  can be updated at each step as

$$x_m = x_{m-1} + \zeta_m p_m \quad (116)$$

where  $p_m$  is defined above.

This brings us to the direct version of Lanczos algorithm for linear systems.

1. Compute  $r_0 = b - Ax_0$ ,  $\zeta_1 = \beta = \|r_0\|_2$ , and  $v_1 = r_0/\beta$
2. Set  $\lambda_1 = \beta_1 = 0$ ,  $p_0 = 0$
3. For  $m = 1, 2, \dots$ , until convergence Do :
4. Compute  $w = Av_m - \beta v_{m-1}$  and  $\alpha_m = (w, v_m)$
5. If  $m > 1$  then compute  $\lambda_m = \frac{\beta_m}{\eta_{m-1}}$  and  $\zeta_m = -\lambda_m \zeta_{m-1}$
6.  $\eta_m = \alpha_m - \lambda_m \beta_m$
7.  $p_m = \eta_m^{-1}(v_m - \beta_m p_{m-1})$
8.  $x_m = x_{m-1} + \zeta_m p_m$
9. If  $x_m$  has converged then Stop
10.  $w = w - \alpha_m v_m$
11.  $\beta_{m+1} = \|w\|_2$ ,  $v_{m+1} = w/\beta_{m+1}$
12. EndDo

This algorithm computes the solution of the tridiagonal system  $T_m y_m = \beta e_1$  progressively by using Gaussian elimination without pivoting. However, partial pivoting can also be implemented at the cost of having to keep an extra vector. In fact, Gaussian elimination with partial pivoting is sufficient to ensure stability for tridiagonal systems. Observe that the residual vector for this algorithm is in the direction of  $v_{m+1}$  due to equation (106). Therefore, the residual vectors are orthogonal to each other. Likewise, the vectors  $p_i$  are  $A$  orthogonal, or conjugate.

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 which we now derive. The vector  $x_{j+1}$ , the solution at iteration  $j+1$ , can be written as,

$$x_{j+1} = x_j + \alpha_j p_j. \quad (117)$$

Therefore the residual vectors must satisfy the recurrence

$$r_{j+1} = r_j - \alpha_j A p_j. \quad (118)$$

If the  $r_j$ 's are to be orthogonal, then it is necessary that  $(r_j - \alpha_j Ap_j, r_j) = 0$  and as a result

$$\alpha_j = \frac{(r_j, r_j)}{(Ap_j, r_j)} \quad (119)$$

Also, it is known that the next search direction  $p_{j+1}$  is a linear combination of  $r_{j+1}$  and  $p_j$ , and after rescaling the  $p$  vectors appropriately, it follows that

$$p_{j+1} = r_{j+1} + \beta_j p_j. \quad (120)$$

Thus, a first consequence of the above relation is that

$$(Ap_j, r_j) = (Ap_j, p_j - \beta_{j-1} p_{j-1}) = (Ap_j, p_j) \quad (121)$$

because  $Ap_j$  is orthogonal to  $p_{j-1}$ . Then, (119) becomes  $\alpha_j = (r_j, r_j)/(Ap_j, p_j)$ . In addition, writing that  $p_{j+1}$  as defined by (120) is orthogonal to  $Ap_j$  yields

$$\beta_j = -\frac{(r_{j+1}, Ap_j)}{(p_j, Ap_j)} \quad (122)$$

$\beta_j$  can also be written as

$$\frac{1}{\alpha_j} \frac{(r_{j+1}, (r_{j+1} - r_j))}{(Ap_j, p_j)} = \frac{(r_{j+1}, r_{j+1})}{(r_j, r_j)} \quad (123)$$

Putting these together we have the algorithm for Conjugate Gradient.

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

### 3.3.4 Preconditioning applied to Conjugate Gradient

Efficiency and robustness of iterative techniques can be improved by using *preconditioning*. Preconditioning is simply a means of transforming the original linear system into one which has the same solution, but which is likely to be easier to solve with an iterative solver.

Consider a matrix  $A$  that is symmetric and positive definite and assume that a preconditioner  $M$  is available. The preconditioner  $M$  is a matrix which approximates  $A$  in some yet-undefined sense. It is assumed that  $M$  is also Symmetric Positive Definite. From a practical point of view, the only requirement for  $M$  is that it is inexpensive to solve linear systems  $Mx = b$ . This is because the preconditioned algorithms will all require a linear system solution with the matrix  $M$  at each step. Then, for example, the following preconditioned system could be solved:

$$M^{-1}Ax = M^{-1}b \quad (124)$$

or

$$AM^{-1}u = b \quad (125)$$

$$x = M^{-1}u \quad (126)$$

These two systems are no longer symmetric in general. To preserve symmetry one can devise  $M$  such that its Cholesky factorization can be written, that is:

$$M = LL^T, \quad (127)$$

Then a simple way to preserve symmetry is to split the preconditioner between left and right, i.e., to solve

$$L^{-1}AL^{-T}u = L^{-1}b, x = L^{-T}u \quad (128)$$

which involves a Symmetric Positive Definite matrix. However, it is not necessary to split the preconditioner in this manner in order to preserve symmetry. Observe that  $M^{-1}A$  is self-adjoint for the  $M$ -inner product,

$$(x, y)_M \equiv (Mx, y) = (x, My) \quad (129)$$

since

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M(M^{-1}A)y) = (x, M^{-1}Ay)_M \quad (130)$$

Therefore, an alternative is to replace the usual Euclidean inner product in the Conjugate Gradient algorithm by the  $M$ -inner product. If the CG algorithm is rewritten for this new inner product, denoting by the original residual and by  $r_j = b - Ax_j$  the original residual and by  $z_j = M^{-1}r_j$  the residual for the preconditioned system the following sequence can be written .

1.  $\alpha_j := (z_j, z_j)_M / (M^{-1}Ap_j, p_j)_M$
2.  $x_{j+1} := x_j + \alpha_j p_j$
3.  $r_{j+1} := r_j - \alpha_j Ap_j$  and  $z_{j+1} := M^{-1}r_{j+1}$
4.  $\beta_j := (z_{j+1}, z_{j+1})_M / (z_j, z_j)_M$
5.  $p_{j+1} := z_{j+1} + \beta_j p_j$

Since  $(z_j, z_j)_M = (r_j, z_j)$  and  $(M^{-1}Ap_j, p_j)_M = (Ap_j, p_j)$ , the  $M$ -inner products do not have to be computed explicitly.

We have the preconditioned iteration for the CG algorithm as follows

1. Compute  $r_0 := b - Ax_0$ ,  $z_0 = M^{-1}r_0$ , and  $p_0 := z_0$
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

### 3.4 Preconditioning Techniques

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 known as ILU(0) is rather easy and inexpensive to compute. On the other hand, it often leads to a crude approximation which may result in the Krylov subspace accelerator requiring many iterations to converge. To remedy this, several alternative incomplete factorizations have been developed by allowing more fill-in in  $L$  and  $U$ . In general, the more accurate ILU factorizations require fewer iterations to converge, but the preprocessing cost to compute the factors is higher. However, if only because of the improved robustness, these trade-offs generally favor the more accurate factorizations. This is especially true when several systems with the same matrix must be solved because the preprocessing cost can be amortized.

The preconditioned versions of some Krylov subspace methods have been discussed in the previous section on CG with a generic preconditioner  $M$ . In theory, any general splitting in which  $M$  is nonsingular can be used. Ideally,  $M$  should be close to  $A$  in some sense. However, note that a linear system with the matrix  $M$  must be solved at each step of the iterative procedure. Therefore, a practical and admittedly somewhat vague requirement is that these solutions steps should be inexpensive.

Consider a general sparse matrix  $A$  whose elements are  $a_{ij}, i, j = 1, \dots, n$ . A general Incomplete LU (ILU) factorization process computes a sparse lower triangular matrix  $L$  and a sparse upper triangular matrix  $U$  so the residual matrix  $R = LU - A$  satisfies certain constraints, such as having zero entries in some locations. We first describe a general ILU preconditioner geared toward  $M$ -matrices. Then we discuss the ILU(0) factorization, the simplest form of the ILU preconditioners.

A general algorithm for building Incomplete LU factorizations can be derived by performing Gaussian elimination and dropping some elements in predetermined nondiagonal positions.

Let  $A$  be an  $M$ -matrix and let  $A_1$  be the matrix obtained from the first step of Gaussian elimination. It can be shown that  $A_1$  is also an  $M$ -matrix. If we remove the first row and first column of  $A_1$  then the resulting  $(n - 1) \times (n - 1)$  matrix is also an  $M$ -matrix.

Assume now that some elements are dropped from the result of Gaussian Elimination outside of the main diagonal. Any element that is dropped is a nonpositive element which is transformed into a zero. Therefore, the resulting matrix  $\tilde{A}_1$  is such that

$$\tilde{A}_1 = A_1 + R, \tag{131}$$

where the elements of  $R$  are such that  $r_{ii} = 0, r_{ij} \geq 0$ . Thus,

$$A_i \leq \tilde{A}_1 \tag{132}$$

and the off-diagonal elements of  $\tilde{A}_1$  are nonpositive. Since  $A_1$  is an  $M$ -matrix,  $\tilde{A}_1$  is also an  $M$ -matrix. The process can now be repeated on the matrix  $\tilde{A}(2:n, 2:n)$ , and then continued until the incomplete factorization of  $A$  is obtained. The above arguments shows that at each step of this construction, we obtain an  $M$ -matrix and that the process does not break down.

The elements to drop at each step have not yet been specified. This can be done statically, by choosing some non-zero pattern in advance. The only restriction on the zero pattern is that it should exclude diagonal elements. Therefore, for any zero pattern set  $P$ , such that

$$P \subset \{(i, j) | i \neq j; 1 \leq i, j \leq n\}, \tag{133}$$

an Incomplete LU factorization,  $ILLU_P$ , can be computed as follows.

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

The For loop in line 4 should be interpreted as follows: For  $j = k + 1, \dots, n$  and *only for those indices  $j$  that are not in  $P$  execute the next line.* In practice, it is wasteful to scan  $j$  from  $k + 1$  to  $n$  because there is an inexpensive mechanism for identifying those in this set that are in the complement of  $P$ .

Now consider a few practical aspects. An ILU factorization based on the form of the previous Algorithm is difficult to implement because at each step  $k$ , all rows  $k + 1$  to  $n$  are being modified. However, ILU factorizations depend on the implementation of Gaussian elimination which is used. Several variants of Gaussian elimination are known which depend on the order of the three loops associated with the control variables  $i, j$  and  $k$  in the algorithm. Thus, previous Algorithm is derived from what is known as the  $k, i, j$  variant. In the context of Incomplete LU factorization, the variant that is most commonly used for a row-contiguous data structure is the  $i, k, j$  variant. It is used for dense matrices.

Adapting this version for sparse matrices is easy because the rows of  $L$  and  $U$  are generated in succession. These rows can be computed one at a time and accumulated in a row-oriented data structure such as the CSR format. This constitutes an important advantage. Based on this, the general ILU factorization takes the following form.

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

It is not difficult to see that this more practical IKJ variant of ILU is equivalent to the KIJ version which can be defined from the first algorithm in this section.

Note that this is only true for a static pattern ILU. If the pattern is dynamically determined as the Gaussian elimination algorithm proceeds, then the patterns obtained with different versions of GE may be different.

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 the following,

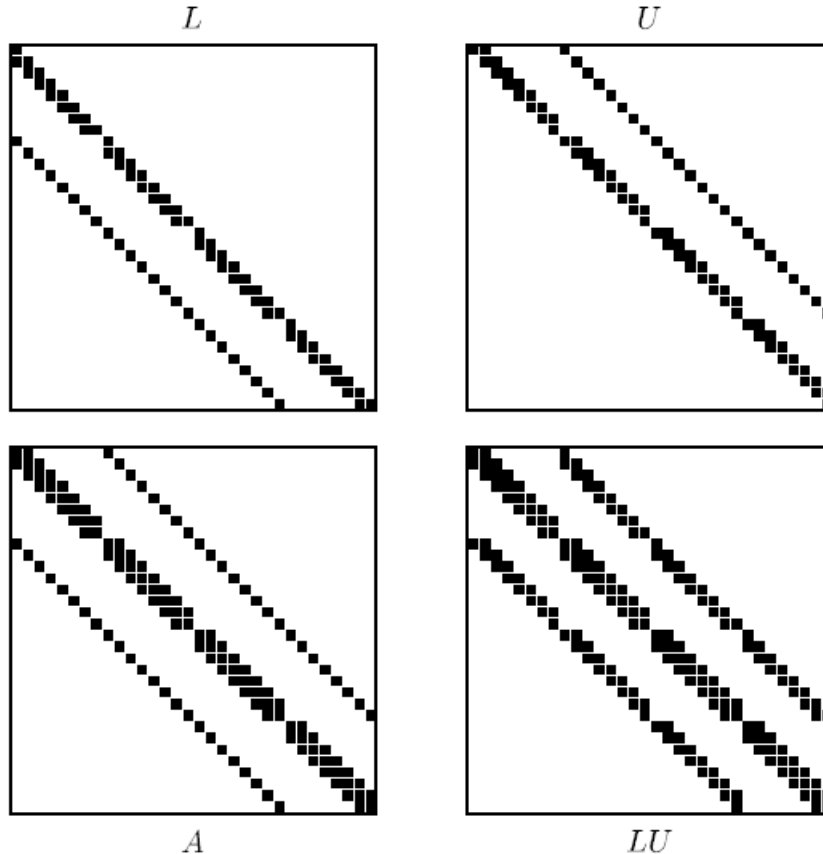


Figure 5: The ILU(0) factorization for a five-point matrix.

we denote by  $b_{i,*}$  the  $i$ -th row of a given matrix  $B$ , and by  $NZ(B)$ , the set of pairs  $(i, j)$ ,  $1 \leq i, j \leq n$  such that  $b_{i,j} \neq 0$ .

The incomplete factorization ILU(0) factorization is best illustrated by the case for which it was used originally, namely, for 5-point and 7-point matrices related to finite difference discretization of PDEs. Consider such a matrix  $A$  as illustrated in the bottom left corner of Figure 5. The  $A$  matrix represented in this figure is a 5-point matrix of size  $n = 32$  corresponding to an  $n_x \times n_y = 8 \times 4$  mesh. Consider now any lower triangular matrix  $L$  which has the same structure as the lower part of  $A$ , and any matrix  $U$  which has the same structure as that of the upper part of  $A$ . Two such matrices are shown at the top of Figure 5. If the product  $LU$  is performed, the resulting matrix would have the pattern shown in the bottom right part of the figure. It is impossible in general to match  $A$  with this product for any  $L$  and  $U$ . This is due to the extra diagonals in the product, namely, the diagonals with offsets  $n_x - 1$  and  $-n_x + 1$ . The entries in these extra diagonals are called *fill-in elements*. However, if these fill-in elements are ignored, then it is possible to find  $L$  and  $U$  so that their product is equal to  $A$  in the other diagonals. This defines the ILU(0) factorization in general terms: Any pair of matrices  $L$  (unit lower triangular) and  $U$  (upper triangular) so that the elements of  $A - LU$  are zero in the locations of  $NZ(A)$ . These constraints do not define the ILU(0) factors uniquely since there are, in general, infinitely many pairs of matrices  $L$  and  $U$  which satisfy these requirements. However, the standard ILU(0) is defined constructively using the above Algorithm with the pattern  $P$  equal to the zero pattern of  $A$ .

1. For  $i = 2, \dots, n$  Do:

2. For  $k = 1, \dots, i - 1$  and for  $(i, k) \in NZ(A)$  Do:
3. Compute  $a_{ik} = a_{ik}/akk$
4. For  $j = k + 1, \dots, n$  and for  $(i, j) \in NZ(A)$ , Do:
5. Compute  $a_{ij} := a_{ij} - a_{ik}a_{kj}$ .
6. EndDo
7. EndDo
8. EndDo

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 more accurate factorizations will differ from ILU(0) by allowing some fill-in. Thus, ILU(1) keeps the first order fill-ins, a term which will be explained shortly. To illustrate ILU( $p$ ) with the same example as before, the ILU(1) factorization results from taking  $P$  to be the zero pattern of the product  $LU$  of the factors  $L, U$  obtained from ILU(0). This pattern is shown at the bottom right of Figure 5. Pretend that the original matrix has this augmented pattern  $NZ_1(A)$ . In other words, the fill-in positions created in this product belong to the augmented pattern  $NZ_1(A)$ , but their actual values are zero. The new pattern of the matrix  $A$  is shown at the bottom left part of Figure 6. The factors  $L_1$  and  $U_1$  of the ILU(1) factorization are obtained by performing an ILU(0) factorization on this augmented pattern matrix. The patterns of  $L_1$  and  $U_1$  are illustrated at the top of Figure 6. The new LU matrix shown at the bottom right of the figure has now two additional diagonals in the lower and upper parts.

One problem with the construction defined in this illustration is that it does not extend to general sparse matrices. It can be generalized 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. Any form of GE can be used to illustrate. The rationale is that the level of fill should be indicative of the size: the higher the level, the smaller the elements. A very simple model is employed to justify the definition: A size of  $\epsilon^k$  is attributed to any element whose level of fill is  $k$ , where  $\epsilon < 1$ . Initially, a nonzero element has a level of fill of one (this will be changed later) and a zero element has a level of fill of  $\infty$ . An element  $a_{ij}$  is updated in line 5 of IKJ variant of GE by the formula

$$a_{ij} = a_{ij} - a_{ik} \times a_{kj}. \quad (134)$$

If  $lev_{ij}$  is the current level of the element  $a_{ij}$ , then our model tells us that the size of the updated element should be

$$a_{ij} := \epsilon^{lev_{ij}} - \epsilon^{lev_{ik}} \times \epsilon^{lev_{kj}} = \epsilon^{lev_{ij}} - \epsilon^{lev_{ij}+lev_{kj}}. \quad (135)$$

Therefore, roughly speaking, the size of  $a_{ij}$  will be the maximum of the two sizes  $\epsilon^{lev_{ij}}$  and  $\epsilon^{lev_{ij}+lev_{kj}}$ , and it is natural to define the new level of fill as,

$$lev_{ij} := \min\{lev_{ij}, lev_{ik} + lev_{kj}\}. \quad (136)$$

In the common definition used in the literature, all the levels of fill are actually shifted by  $-1$  from the definition used above. This is purely for convenience of notation and to conform with the definition used for ILU(0). Thus, initially  $lev_{ij} = 0$  if  $a_{ij} = 0$ , and  $lev_{ij} = \infty$  otherwise. Thereafter, define recursively

$$lev_{ij} := \min\{lev_{ij}, lev_{ik} + lev_{kj} + 1\}. \quad (137)$$

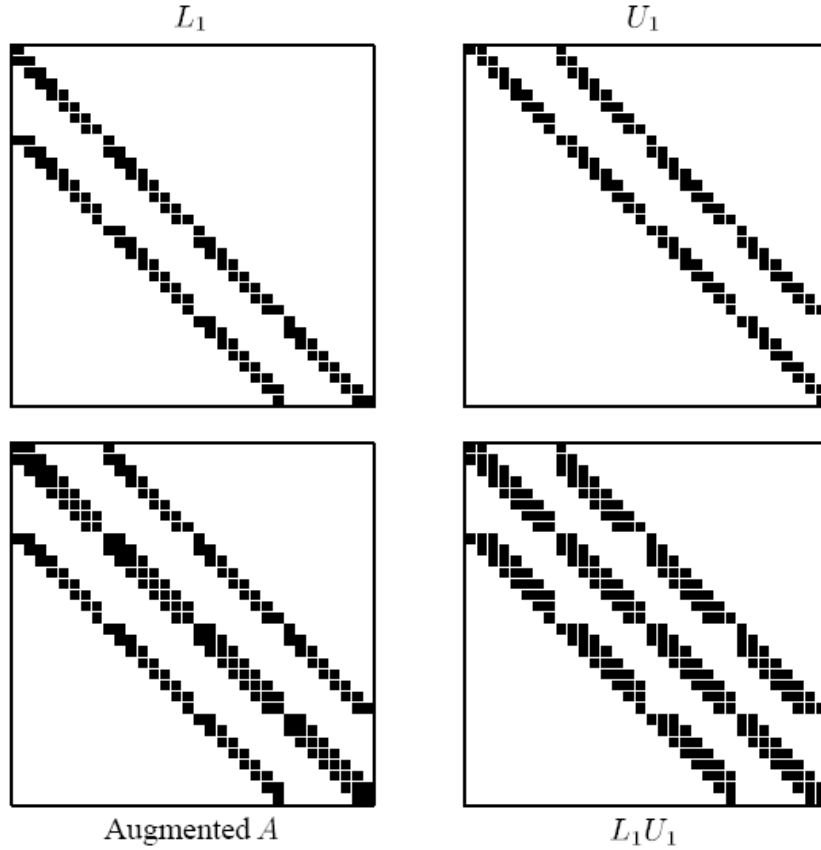


Figure 6: The ILU(1) factorization

Observe that the level of fill of an element will never increase during the elimination. Thus, if  $a_{ij} \neq 0$  in the original matrix  $A$ , then the element in location  $i, j$  will have a level of fill equal to zero throughout the elimination process. The above systematic definition gives rise to a natural strategy for discarding elements. In ILU( $p$ ), all fill-in elements whose level of fill does not exceed  $p$  are kept. So using the definition of zero patterns introduced earlier, the zero pattern for ILU( $p$ ) is the set

$$P_p = \{(i, j) | lev_{ij} > p\}, \quad (138)$$

where  $lev_{ij}$  is the level of fill value after all updates (137) have been performed. The case  $p = 0$  coincides with the ILU(0) factorization and is consistent with the earlier definition.

In practical implementations of the ILU( $p$ ) factorization it is common to separate the symbolic phase (where the structure of the  $L$  and  $U$  factors are determined) from the numerical factorization, when the numerical values are computed. Here, a variant is described which does not separate these two phases. In the following description,  $a_{i*}$  denotes the  $i$ -th row of the matrix  $A$ , and  $a_{ij}$  the  $(i, j)$ -th entry of  $A$ .

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  $a'_{i,j}$ s using (137)
7. EndDo
8. Replace any element in row  $i$  with  $lev(a_{ij}) > p$  by zero
9. EndDo

### 3.4.1 ILUT approach and implementation issues

There are a number of drawbacks to the above algorithm. First, the amount of fill-in and computational work for obtaining the  $ILU(p)$  factorization is not predictable for  $p > 0$ . Second, the cost of updating the levels can be quite high. Most importantly, the level of fill-in for indefinite matrices may not be a good indicator of the size of the elements that are being dropped. Thus, the algorithm may drop large elements and result in an inaccurate incomplete factorization, in the sense that  $R = LU - A$  is not small. Experience reveals that *on the average* this will lead to a larger number of iterations to achieve convergence, although there are certainly instances where this is not the case. The techniques indicated below have been developed to remedy these three difficulties, by producing incomplete factorizations with small error  $R$  and a controlled number of fill-ins.

A generic ILU algorithm with threshold can be derived from the IKJ version of Gaussian elimination, by including a set of rules for dropping small elements. In what follows, applying a dropping rule to an element will only mean replacing the element by zero if it satisfies a set of criteria. A dropping rule can be applied to a whole row by applying the same rule to all the elements of the row. In the following algorithm,  $w$  is a full-length working row which is used to accumulate linear combinations of sparse rows in the elimination and  $w_k$  is the  $k$ -th entry of this row. As usual,  $a_{i*}$  denotes the  $i$ -th row of  $A$ .

ILUT

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

The ILU(0) can be viewed as a special case of the above algorithm. The dropping rule for ILU(0) is to drop elements that are in positions not belonging to the original structure of the matrix.

In an ILU( $p, \tau$ ),  $p$  is the parameter that helps control memory usage, while  $\tau$  helps to reduce computational cost.

In the factorization ILU( $p, \tau$ ), the following rule is used.

1. In line 5, an element  $w_k$  is dropped (i.e., replaced by zero) if it is less than the relative tolerance  $\tau_i$  obtained by multiplying  $\tau$  by the original norm of the  $i$ -th row (e.g., the 2-norm).
2. In line 10, a dropping rule of a different type is applied. First, drop again any element in the row with a magnitude that is below the relative tolerance  $\tau_i$ . 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.

Few of the Implementation details are worth noting for this method of factorization. Listed below are the challenges to an efficient implementation.

1. Generation of the linear combination of rows of  $A$ .
2. Selection of the  $p$  largest elements in  $L$  and  $U$ .
3. Need to access the elements of  $L$  in increasing order of columns.

To solve the first issue one can use a clever storage pattern as summarized in [Saad, 1996]. For the second hurdle heapsort or a variation on quick sort could be utilized. Finally to speed-up the access to the elements of  $L$  can be done by storing them in a binary search tree.

There could be problems with the implementation of ILUT case. They could be summarized as follows:-

1. The ILUT procedure encounters a zero pivot;
2. The ILUT procedure encounters an overflow or underflow condition, because of an exponential growth of the entries of the factors;
3. The ILUT preconditioner terminates normally but the incomplete factorization preconditioner which is computed is unstable.

To remedy the problems that might arise with the ILUT approach an ILUTP approach might be used in generating a factorization.

ILUTP("P" stands for pivoting) uses a permutation array  $perm$  to hold the new orderings of the variables, along with the reverse permutation array. At step  $i$  of the elimination process the largest entry in a row is selected and is defined to be the new  $i$ -th variable. The two permutation arrays are then updated accordingly. The matrix elements of  $L$  and  $U$  are kept in their original numbering. However, when expanding the  $L - U$  row which corresponds to the  $i$ -th outer step of Gaussian elimination, the elements are loaded with respect to the new labeling, using the array  $perm$  for the translation. At the end of the process, there are two options. The first is to leave all elements labeled with respect to the original labeling. No additional work is required since the variables are already in this form in the algorithm, but the variables must then be permuted at each preconditioning step. The second solution is to apply the permutation to all elements of  $A$  as well as  $L/U$ . This does not require applying a permutation at each step, but rather produces a permuted solution which must be permuted back at the end of the iteration phase. The

complexity of the ILUTP procedure is virtually identical to that of ILUT. A few additional options can be provided. A tolerance parameter called *permtol* may be included to help determine whether or not to permute variables: A nondiagonal element  $a_{ij}$  is candidate for a permutation only when  $tol \times |a_{ij}| > |a_{ii}|$ . Furthermore, pivoting may be restricted to take place only within diagonal blocks of a fixed size. If we assume that the size of the blocks is named as *mbloc* then a value of  $ofmbloc \geq n$  indicates that there are no restrictions on the pivoting. A state-of-the-art Multilevel scheme using ILU preconditioners is also discussed in [Bollhöfer and Saad, 2006]. To solve special matrices stored in the *sparse skyline format (SSK)* an ILUTS method could be used to factorize. Thus savings in storage could be leveraged. Also the symmetric nature of such matrices could result in a symmetric preconditioner.

### 3.5 Domain Decomposition

Domain decomposition methods refer to a collection of techniques which revolve around the principle of divide-and-conquer. Consider the problem of solving the Laplace Equation on an L-shaped domain

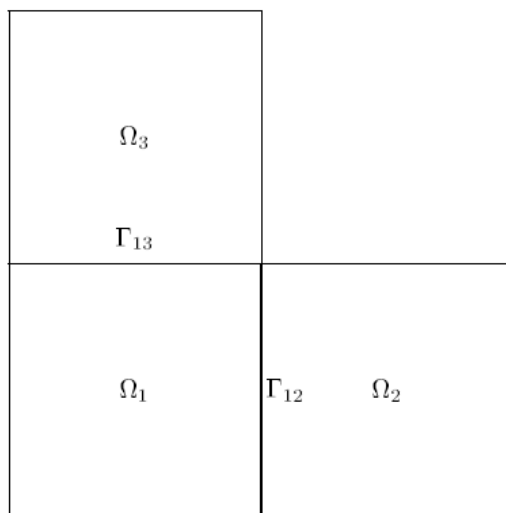


Figure 7: An L-shaped domain subdivided into three subdomains

partitioned as shown in Figure 7. Domain decomposition or substructuring methods attempt to solve the problem on the entire domain

$$\Omega = \bigcup_{i=1}^s \Omega_i, \quad (139)$$

from problem solutions on the subdomains  $\Omega_i$ . There are several reasons why such techniques can be advantageous. In the case of the above picture, one obvious reason is that the subproblems are much simpler because of their rectangular geometry. For example, fast solvers can be used on each subdomain in this case. A second reason is that the physical problem can sometimes be split naturally into a small number of subregions where the modeling equations are different (e.g., Eulers equations on one region and Navier-Stokes in another). Substructuring can also be used to develop out-of-core solution techniques. As already mentioned, such techniques were often used in the past to analyze very large mechanical structures. The original structure is partitioned into pieces, each of which is small enough to fit into memory. Then a form of block-Gaussian elimination is used to solve the global linear system from a sequence of solutions using subsystems.

In order to review the issues and techniques in use and to introduce some notation, assume that the following problem is to be solved:

$$\Delta u = f \text{ in } \Omega \quad (140)$$

$$u = u_\Gamma \text{ on } \Gamma = \partial\Omega. \quad (141)$$

Domain decomposition methods are all implicitly or explicitly based on different ways of handling the unknowns at the interfaces. From the PDE point of view, if the value of the solution is known at the interfaces between the different regions, these values could be used in Dirichlet-type boundary conditions and we will obtain  $s$  uncoupled Poisson equations. We can then solve these equations to obtain the value of the solution at the interior points. If the whole domain is discretized by either finite elements or finite difference techniques, then this is easily translated into the resulting linear system.

Assume that the problem associated with the domain shown in Figure 7 is discretized with centered differences. We can label the nodes by subdomain as shown in Figure 8.

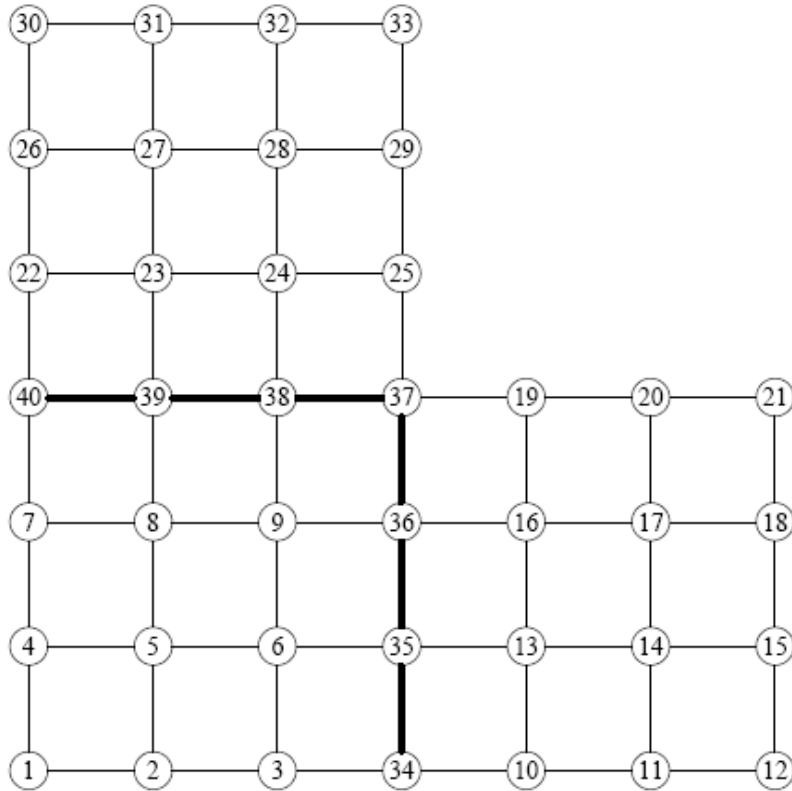


Figure 8: Discretization of the problem for L-shaped geometry

Note that the interface nodes are labeled last. As a result, the matrix associated with this problem will have the structure shown in Figure 9. For a general partitioning into  $s$  subdomains, the linear system associated with the problem has the following structure:

$$\begin{pmatrix} B_1 & & & E_1 \\ & B_2 & & E_2 \\ & & \ddots & \vdots \\ & & & B_s & E_s \\ F_1 & F_2 & \cdots & F_s & C \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \\ g \end{pmatrix} \quad (142)$$

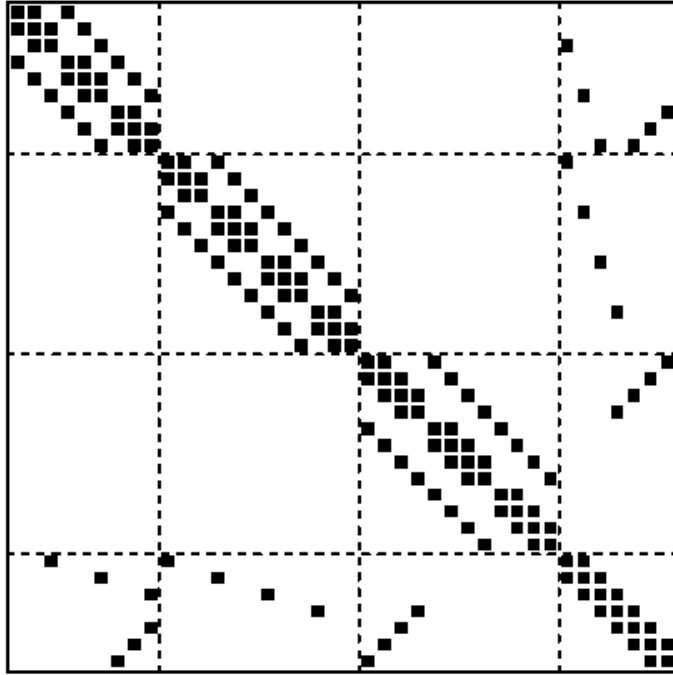


Figure 9: Matrix associated with the finite difference mesh of the Figure 8

where each  $x_i$  represents the subvector of unknowns that are interior to subdomain  $\Omega_i$  and  $y$  represents the vector of all interface unknowns. It is useful to express the above system in the simpler form,

$$A \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix} \text{ with } A = \begin{pmatrix} B & E \\ F & C \end{pmatrix} \quad (143)$$

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

### 3.5.1 Direct Solution and Schur Complement

Consider the linear system written in the form (143), in which  $B$  is assumed to be non-singular. From the first equation the unknown  $x$  can be expressed as

$$x = B^{-1}(f - Ey) \quad (144)$$

Upon substituting this into the second equation, the following *reduced system* is obtained:

$$(C - FB^{-1}E)y = g - FB^{-1}f. \quad (145)$$

The matrix

$$S = C - FB^{-1}E \quad (146)$$

is called the *Schur complement* matrix associated with the  $y$  variable. If this matrix can be formed and the linear system (145) can be solved, all the interface variables  $y$  will become available. Once these variables are known, the remaining unknowns can be computed, via (144). Because of the particular structure of  $B$ , observe that any linear system solution with it decouples in  $s$  separate systems. The parallelism in this situation arises from this natural decoupling. A solution method based on this approach involves four steps:

1. Obtain the right-hand side of the reduced system (144).
2. Form the Schur complement matrix (145).
3. Solve the reduced system (144).
4. Back-substitute using (143) to obtain the other unknowns.

One linear system solution with the matrix  $B$  can be saved by reformulating the algorithm in a more elegant form. Define

$$E' = B^{-1}E \quad \text{and} \quad f' = B^{-1}f. \quad (147)$$

The matrix  $E'$  and the vector  $f'$  are needed in steps (1) and (2). Then rewrite step (4) as

$$x = B^{-1}f - B^{-1}Ey = f' - E'y, \quad (148)$$

which gives the following algorithm

BLOCK GAUSSIAN ELIMINATION

1. Solve  $BE' = B$ , 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 a practical implementation, 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 in  $E_i$  will be zero. These zero columns correspond to interfaces that are not adjacent to subdomain  $i$ . Therefore, any efficient code based on the above algorithm should start by identifying the nonzero columns.

PROPERTIES OF SCHUR COMPLEMENT

If  $A$  be a nonsingular matrix partitioned as in (143) and such that the submatrix  $B$  is nonsingular and let  $R_y$  be the restriction operator onto the interface variables, i.e, the linear operator defined by

$$R_y \begin{pmatrix} x \\ y \end{pmatrix} = y. \quad (149)$$

Then the following properties are true.

1. The Schur complement matrix  $S$  is nonsingular.
2. If  $A$  is SPD, then so is  $S$ .
3. For any  $y$ ,  $S^{-1}y = R_yA^{-1}\begin{pmatrix} 0 \\ y \end{pmatrix}$ .

The first property indicates that a method that uses the above block Gaussian elimination algorithm is feasible since  $S$  is nonsingular. A consequence of the second property is that when  $A$  is positive definite, an algorithm such as the Conjugate Gradient algorithm can be used to solve the reduced system (145). Finally, the third property establishes a relation which may allow preconditioners for  $S$  to be defined based on solution techniques with the matrix  $A$ .

SCHUR COMPLEMENT FOR VERTEX BASED PARTITIONINGS

The partitioning used in Figure 8 is edge-based, meaning that a given edge in the graph does not straddle two subdomains. If two vertices are coupled, then they must belong to the same subdomain. From the graph theory point of view, this is perhaps less common than vertex-based partitionings in which a vertex is not shared by two partitions (except when domains overlap). A vertex-based partitioning is illustrated in Figure 10. We will call interface edges all edges that link vertices that do not belong to the same subdomain. In the case of overlapping, this needs clarification. An overlapping edge or vertex belongs to the same subdomain. Interface edges are only those that link a vertex to another vertex which is not in the same subdomain already, whether in the overlapping portion or elsewhere. Interface vertices are those vertices in a given subdomain that are adjacent to an interface edge. For the example of the figure, the interface vertices for subdomain one (bottom, left subsquare) are the vertices labeled 10 to 16. The matrix shown at the bottom of Figure 10 differs from the one of Figure 9, because here the interface nodes are not relabeled the last in the global labeling as was done in Figure 8. Instead, the interface nodes are labeled as the last nodes in each subdomain. The number of interface nodes is about twice that of the edge-based partitioning.

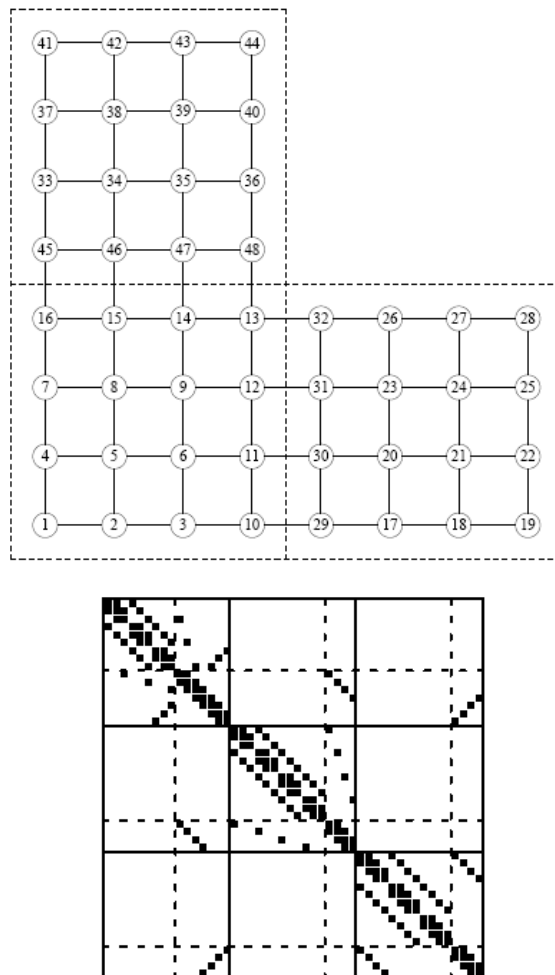


Figure 10: Discretization of problem for L-shaped domain subdivided into three

Consider the Schur complement system obtained with this new labeling. It can be written similar to the edge-based case using a reordering in which all interface variables

are listed last. The matrix associated with the domain partitioning of the variables will have a natural  $s$ -block structure where  $s$  is the number of subdomains. For example, when  $s = 3$  (as is the case in the above illustration), the matrix has the block structure defined by the solid lines in the figure, i.e.,

$$A = \begin{pmatrix} A_1 & A_{12} & A_{13} \\ A_{21} & A_2 & A_{23} \\ A_{31} & A_{32} & A_3 \end{pmatrix}. \quad (150)$$

In each subdomain, the variables are of the form

$$z_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \quad (151)$$

where  $x_i$  denotes interior nodes while  $y_i$  denotes the interface nodes associated with subdomain  $i$ . Each matrix  $A_i$  will be called the local matrix. The structure of  $A_i$  is as follows:

$$A_i = \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix} \quad (152)$$

in which, as before,  $B_i$  represents the matrix associated with the internal nodes of subdomain  $i$  and  $E_i$  and  $F_i$  represent the couplings to/from external nodes. The matrix  $C_i$  is the local part of the interface matrix  $C$  defined before (231), and represents the coupling between local interface points. A careful look at the matrix in Figure 10 reveals an additional structure for the blocks  $A_{ij}$ ,  $j \neq i$ . Each of these blocks contains a zero sub-block in the part that acts on the variable  $x_j$ . This is expected since  $x_i$  and  $x_j$  are not coupled. Therefore,

$$A_{ij} = \begin{pmatrix} 0 \\ E_{ij} \end{pmatrix}. \quad (153)$$

In addition, most of the  $E_{ij}$  matrices are zero since only those indices  $j$  of the subdomains that have couplings with subdomain  $i$  will yield a nonzero  $E_{ij}$ . Now we write the part of the linear system that is local to subdomain  $i$ , as

$$B_i x_i + E_i y_i = f_i \quad (154)$$

$$F_i x_i + C_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i. \quad (155)$$

The term  $E_{ij} y_j$  is the contribution to the equation from the neighboring subdomain number  $j$ , and  $N_i$  is the set of subdomains that are adjacent to subdomain  $i$ . Assuming that  $B_i$  is nonsingular, the variable  $x_i$  can be eliminated from this system by extracting from the first equation  $x_i = B_i^{-1}(f_i - E_i y_i)$  which yields, upon substitution in the second equation,

$$S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i - F_i B_i^{-1} f_i, \quad i = 1, \dots, s \quad (156)$$

in which  $S_i$  is the local Schur complement

$$S_i = C_i - F_i B_i^{-1} E_i. \quad (157)$$

When written for all subdomains  $i$ , the equations (156) yield a system of equations which involves only the interface points  $y_j$ ,  $j = 1, 2, \dots, s$  and which has a natural block structure associated with these vector variables

$$S = \begin{pmatrix} S_1 & E_{12} & E_{13} & \cdots & E_{1s} \\ E_{21} & S_2 & E_{23} & \cdots & E_{2s} \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ E_{s1} & E_{s2} & E_{s3} & \cdots & S_s \end{pmatrix} \quad (158)$$

The diagonal blocks in this system, namely, the matrices  $S_i$ , are dense in general, but the off-diagonal blocks  $E_{ij}$  are sparse and most of them are zero. Specifically,  $E_{ij} \neq 0$  only if subdomains  $i$  and  $j$  have at least one equation that couples them. A structure of the global Schur complement  $S$  has been described which has the following important implication: *For vertex-based partitionings, the Schur complement matrix can be assembled from local Schur complement matrices (the  $S_i$ 's) and interface-to-interface information (the  $E_i$ s). A similar idea will be exploited for finite element partitionings.*

### SCHUR COMPLEMENT FOR FINITE-ELEMENT PARTITIONINGS

In finite-element partitionings, the original discrete set  $\Omega$  is subdivided into subsets  $\Omega_i$ , each consisting of a distinct set of elements. Given a finite element discretization of the domain  $\Omega$ , a finite dimensional space  $V_h$  of functions over  $\Omega$  is defined, e.g., functions that are piecewise linear and continuous on  $\Omega$ , and that vanish on the boundary  $\Gamma$  of  $\Omega$ . Consider now the Dirichlet problem on  $\Omega$  and recall that its weak formulation on the finite element discretization can be stated as follows :

$$\text{Find } u \in V_h \text{ such that } a(u, v) = (f, v), \forall v \in V_h, \quad (159)$$

where the bilinear form  $a(., .)$  is defined by

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v dx = \int_{\Omega} \left( \frac{\partial u}{\partial x_1} \frac{\partial v}{\partial x_1} + \frac{\partial u}{\partial x_2} \frac{\partial v}{\partial x_2} \right) dx \quad (160)$$

It is interesting to observe that since the set of the elements of the different  $\Omega_i$ s are disjoint,  $a(., .)$  can be decomposed as

$$a(u, v) = \sum_{i=1}^s a_i(u, v), \quad (161)$$

where

$$a_i(u, v) = \int_{\Omega_i} \nabla u \cdot \nabla v dx. \quad (162)$$

In fact, this is a generalization of the technique used to assemble the stiffness matrix from element matrices, which corresponds to the extreme case where each  $\Omega_i$  consists of exactly one element. If the unknowns are ordered again by subdomains and the interface nodes are placed last as was done previously, immediately the system shows the same structure,

$$\begin{pmatrix} B_1 & & & E_1 \\ & B_2 & & E_2 \\ & & \ddots & \vdots \\ & & & B_s & E_s \\ F_1 & F_2 & \cdots & F_s & C \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \\ g \end{pmatrix} \quad (163)$$

where each  $B_i$  represents the coupling between interior nodes and  $E_i$  and  $F_i$  represent the coupling between the interface nodes and the nodes interior to  $\Omega_i$ . Note that each of these matrices has been assembled from element matrices and can therefore be obtained from contributions over all subdomain  $\Omega_i$  that contain any node of  $\Omega_i$ . In particular, assume that the assembly is considered only with respect to  $\Omega_i$ . Then the assembled matrix will have the structure

$$A_i = \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix}. \quad (164)$$

where  $C_i$  contains only contributions from local elements, i.e., elements that are in  $\Omega_i$ . Clearly,  $C$  is the sum of the  $C_i$ s,

$$C = \sum_{i=1}^s C_i \quad (165)$$

The Schur complement associated with the interface variables is such that

$$S = C - FB^{-1}E = C - \sum_{i=1}^s F_i B_i^{-1} E_i = \sum_{i=1}^s C_i - \sum_{i=1}^s F_i B_i^{-1} E_i = \sum_{i=1}^s [C_i - F_i B_i^{-1} E_i]. \quad (166)$$

Therefore, if  $S_i$  denotes the local Schur complement

$$S_i = C_i - F_i B_i^{-1} E_i, \quad (167)$$

then the above proves that,

$$S = \sum_{i=1}^s S_i, \quad (168)$$

showing again that the Schur complement can be obtained easily from smaller Schur complement matrices.

### 3.5.2 Schwarz Alternating Procedures

The original alternating procedure described by Schwarz consisted 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. This procedure is called the Multiplicative Schwarz procedure. In matrix terms, this is very reminiscent of the block Gauss-Seidel iteration with overlap defined with the help of projectors. The analogue of the block-Jacobi procedure is known as the Additive Schwarz procedure.

#### MULTIPLICATIVE SCHWARZ PROCEDURE

In the following, assume that each pair of neighboring subdomains has a non-void overlapping region. The boundary of subdomain  $\Omega_i$  that is included in subdomain  $j$  is denoted by  $\Gamma_{i,j}$

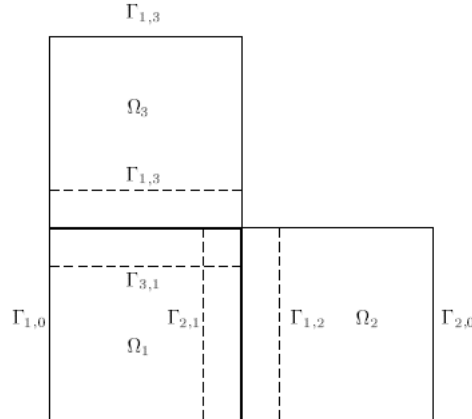


Figure 11: An L-shaped domain subdivided into three overlapping subdomains

This is illustrated in Figure 11 for the L-shaped domain example. Each subdomain extends beyond its initial boundary into neighboring subdomains. Call  $\Gamma_i$  the boundary of  $\Omega_i$  consisting of its original boundary (which is denoted by  $\Gamma_{i,0}$ ) and the  $\Gamma_{i,j}$ 's and denote by  $u_{ji}$  the restriction of the solution  $u$  to the boundary  $\Gamma_{ji}$ . Then the Schwarz Alternating Procedure can be described as follows.

SAP - Schwarz Alternating Procedure

1. Choose an initial guess  $u$  to the solution
2. Until convergence Do:
3. For  $i = 1, \dots, s$  Do:
4. Solve  $\Delta u = f$  in  $\Omega_i$  with  $u = u_{ij}$  in  $\Gamma_{ij}$ .
5. Update  $u$  values on  $\Gamma_{ji}$
6. EndDo
7. EndDo

The algorithm sweeps through the  $s$  subdomains and solves the original equation in each of them by using boundary conditions that are updated from the most recent values of  $u$ . Since each of the subproblems is likely to be solved by some iterative method, we can take advantage of a good initial guess. It is natural to take as initial guess for a given subproblem the most recent approximation. Going back to the expression (154) of the local problems, observe that each of the solutions in line 4 of the algorithm will be translated into an update of the form

$$u_i := u_i + \delta_i, \quad (169)$$

where the correction  $\delta_i$  solves the system

$$A_i \delta_i = r_i \quad (170)$$

Here,  $r_i$  is the local part of the most recent global residual vector  $b - Ax$ , and the above system represents the system associated with the problem in line 4 of the algorithm when a nonzero initial guess is used in some iterative procedure. The matrix  $A_i$  has the block structure (152). Writing

$$u_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \delta_i = \begin{pmatrix} \delta_{x,i} \\ \delta_{y,i} \end{pmatrix}, r_i = \begin{pmatrix} r_{x,i} \\ r_{y,i} \end{pmatrix}, \quad (171)$$

the correction to the current solution step in the algorithm leads to

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} := \begin{pmatrix} x_i \\ y_i \end{pmatrix} + \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix}^{-1} \begin{pmatrix} r_{x,i} \\ r_{y,i} \end{pmatrix} \quad (172)$$

After this step is taken, normally a residual vector  $r$  would have to be computed again to get the components associated with domain  $i + 1$  and to proceed with a similar step for the next subdomain. However, only those residual components that have been affected by the change of the solution need to be updated. Specifically, employing the same notation used in equation (154), we can simply update the residual  $r_{y,j}$  for each subdomain  $j$  for which  $i$  in  $N_j$  as

$$r_{y,j} := r_{y,j} - E_{ji} \delta_{y,i}. \quad (173)$$

This amounts implicitly to performing Step 5 of the above algorithm. Note that since the matrix pattern is assumed to be symmetric, then the set of all indices  $j$  such that  $i \in N_j$ , i.e.,  $N_i^* | i \in N_i$ , is identical to  $N_i$ . Now the loop starting in line 3 of the previous Algorithm and called *domain sweep* can be restated as follows.

MULTIPLICATIVE SCHWARZ Sweep - Matrix Form

1. For  $i = 1, \dots, s$  Do:
2. Solve  $A_i \delta_i = r_i$
3. Compute  $x_i := x_i + \delta_{x,i}, y_i := y_i + \delta_{y,i}$ , and set  $r_i := 0$
4. For each  $j \in N_i$  Compute  $r_{y,j} := r_{y,j} - E_{ji} \delta_{y,i}$
5. EndDo

Considering only the  $y$  iterates, the above iteration would resemble a form of Gauss-Seidel procedure on the Schur complement matrix (158). In fact, it is mathematically equivalent, provided a consistent initial guess is taken.

It is interesting to interpret the Schwarz alternating procedure, or rather its discrete version, in terms of projectors. For this we follow the model of the overlapping block-Jacobi technique. Let  $S_i$  be an index set

$$S_i = \{j_1, j_2, \dots, j_{n_i}\}, \quad (174)$$

where the indices  $j_k$  are those associated with the  $n_i$  mesh points of the interior of the discrete subdomain  $\Omega_i$ . Note that as before, the  $S_i$ s form a collection of index sets such that

$$\bigcup_{i=1, \dots, s} S_i = \{1, \dots, n\} \quad (175)$$

and the  $S_i$ s are not necessarily disjoint. Let  $R_i$  be a *restriction operator* from  $\Omega$  to  $\omega_i$ . By definition,  $R_i x$  belongs to  $\Omega_i$  and keeps only those components of an arbitrary vector  $x$  that are in  $\Omega_i$ . It is represented by an  $n_i \times n$  matrix of zeros and ones. The matrices  $R_i$  associated with the partitioning of Figure 9 are represented in the three diagrams of Figure 12, where each square represents a nonzero element (equal to one) and every other element is a zero. These matrices depend on the ordering chosen for the local problem. Here, boundary nodes are labeled last, for simplicity. Observe that each row of each  $R_i$  has exactly one nonzero element (equal to one). Boundary points such as the nodes 36 and 37 are represented several times in the matrices  $R_1, R_2$  and  $R_3$  because of the overlapping of the boundary points. Thus, node 36 is represented in matrices  $R_1$  and  $R_2$ , while 37 is represented in all three matrices.

From the linear algebra point of view, the restriction operator  $R_i$  is an  $n_i \times n$  matrix formed by the transposes of columns  $e_j$  of the  $n \times n$  identity matrix, where  $j$  belongs to the index set  $S_i$ . The transpose  $R_i^T$  of this matrix is a *prolongation operator* which takes a variable from  $\Omega_i$  and *extends* it to the equivalent variable in  $\Omega$ . The matrix

$$A_i = R_i A R_i^T \quad (176)$$

of dimension  $N_i \times n_i$  defines a restriction of  $A$  to  $\Omega_i$ . Now a problem associated with  $A_i$  can be solved which would update the unknowns in the domain  $\Omega_i$ . With this notation, the multiplicative Schwarz procedure can be described as follows:

1. For  $i = 1, \dots, s$  Do
2.  $x := x + R_i^T A_i^{-1} R_i (b - Ax)$
3. EndDo

We change notation and rewrite step 2 as

$$x_{new} = x + R_i^T A_i^{-1} R_i (b - Ax). \quad (177)$$

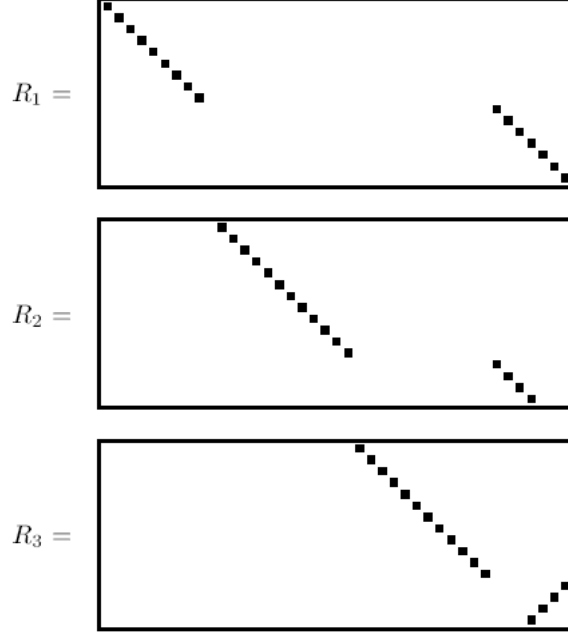


Figure 12: Patterns of the three matrices  $R_i$  associated with the partitioning as shown in the finite difference mesh

If the errors  $d = x_* - x$  are considered where  $x_*$  is the exact solution, then notice that  $b - Ax = A(x_* - x)$  and, at each iteration the following equation relates the new error  $d_{new}$  and the previous error  $d$ ,

$$d_{new} = d - R_i^T A_i^{-1} R_i A d. \quad (178)$$

Starting from a given  $x_0$  whose error vector is  $d_0 = x_* - x$ , each sub-iteration produces error vector which satisfies the relation

$$d_i = d_{i-1} - R_i^T A_i^{-1} R_i A d_{i-1}, \quad (179)$$

for  $i = 1, \dots, s$ . As a result,

$$d_i = (I - P_i) d_{i-1} \quad (180)$$

in which

$$P_i = R_i^T A_i^{-1} R_i A. \quad (181)$$

Observe that  $P_i \equiv R_i^T A_i^{-1} R_i A$  is a projector since

$$(R_i^T A_i^{-1} R_i A)^2 = R_i^T A_i^{-1} (R_i A R_i^T) A_i^{-1} R_i A = R_i^T A_i^{-1} R_i A. \quad (182)$$

### MULTIPLICATIVE SCHWARZ PRECONDITIONING

Because of the equivalence of the multiplicative Schwarz procedure and a block Gauss-Seidel iteration, it is possible to recast one Multiplicative Schwarz sweep in the form of a global fixed-point iteration of the form  $x_{new} = Gx + f$ . Recall that this is a fixed-point iteration for solving the preconditioned system  $M^{-1}Ax = M^{-1}b$  where the preconditioning matrix  $M$  and the matrix  $G$  are related by  $G = I - M^{-1}A$ . To interpret the operation associated with  $M^{-1}$ , it is helpful to identify the result of the error vector produced by this iteration with  $x_{new} - x_* = Q_s(x - x_*)$ . This comparison yields,

$$x_{new} = Q_s x + (I - Q_s) x_*, \quad (183)$$

and therefore,

$$G = Q_s f = (I - Q_s)x_*. \quad (184)$$

Hence, the preconditioned matrix is  $M^{-1}A = I - Q_s$ .

So it can be stated that the multiplicative Schwarz procedure is equivalent to a fixed-point iteration for the preconditioned problem

$$M^{-1}Ax = M^{-1}b, \quad (185)$$

in which

$$M^{-1}A = I - Q_s \quad (186)$$

$$M^{-1}b = (I - Q_s)x_* = (I - Q_s)A^{-1}b. \quad (187)$$

The transformed right-hand side in the proposition is not known explicitly since it is expressed in terms of the exact solution. However, a procedure can be found to compute it. In other words, it is possible to operate with  $M^{-1}$  without invoking  $A^{-1}$ . Note that  $M^{-1} = (I - Q_s)A^{-1}$ .  $M^{-1}$  and  $M^{-1}A$  can be computed recursively.

Define the matrices

$$Z_i = I - Q_i \quad (188)$$

$$M_i = Z_i A^{-1} \quad (189)$$

$$T_i = P_i A^{-1} = R_i^T A_i^{-1} R_i \quad (190)$$

for  $i = 1, \dots, s$ . Then  $M^{-1} = M_s$ ,  $M^{-1}A = Z_s$ , and the matrices  $Z_i$  and  $M_i$  satisfy the recurrence relations

$$Z_1 = P_1, Z_i = Z_{i-1} + P_i(I - Z_{i-1}), i = 2, \dots, s \quad (191)$$

and

$$M_1 = T_1, M_i = M_{i-1} + T_i(I - AM_{i-1}), i = 2, \dots, s. \quad (192)$$

Note that (191) yields immediately the important relation

$$Z_i = \sum_{j=1}^i P_j Q_{j-1} \quad (193)$$

If the relation (192) is multiplied to the right by a vector  $v$  and if the vector  $M_i A^{-1}v$  is denoted by  $z_i$ , then the following recurrence results.

$$z_i = z_{i-1} + T_i(V - Az_{i-1}). \quad (194)$$

Since  $z_s = (I - Q_s)A^{-1}v = M^{-1}v$ , the end result is that  $M^{-1}v$  can be computed for an arbitrary vector  $v$ , by the following procedure.

MULTIPLICATIVE SCHWARZ PRECONDITIONER

1. Input:  $v$ ; Output:  $z = M^{-1}v$
2.  $z := T_1 v$
3. For  $i = 2, \dots, s$  Do:
4.  $z := z + T_i(v - Az)$

5. EndDo

By a similar argument, a procedure can be found to compute vectors of the form  $z = M^{-1}Av$ . In this case an operator algorithm can be shown:

MULTIPLICATIVE SCHWARZ PRECONDITIONED OPERATOR

1. Input:  $v$ , Output:  $z = M^{-1}Av$ .
2.  $z := P_1v$
3. For  $i = 2, \dots, s$  Do
4.  $z := z + P_i(v - z)$
5. EndDo

In summary, the Multiplicative Schwarz procedure is equivalent to solving the preconditioned system

$$(I - Q_s)x = g \tag{195}$$

where the operation  $(I - Q_s)v$  can be computed from the operator Algorithm and  $g = M^{-1}b$  can be computed from preconditioning Algorithm.

ADDITIVE SCHWARZ PRECONDITIONER

The additive Schwarz procedure is similar to a block-Jacobi iteration and consists of updating all the new (block) components from the same residual. Thus, it differs from the multiplicative procedure only because the components in each subdomain are not updated until a whole cycle of updates through all domains are completed. The basic Additive Schwarz iteration would therefore be as follows:

1. For  $i = 1, \dots, s$  Do
2. Compute  $\delta_i = R_i^T A_i^{-1} R_i(b - Ax)$
3. EndDo
4.  $x_{new} = x + \sum_{i=1}^s \delta_i$

The new approximation (obtained after a cycle of the  $s$  substeps in the above algorithm are applied) is

$$x_{new} = x + \sum_{i=1}^s R_i^T A_i^{-1} R_i(b - Ax). \tag{196}$$

Each instance of the loop redefines different components of the new approximation and there is no data dependency between the subproblems involved in the loop.

The preconditioning matrix is rather simple to obtain for the additive Schwarz procedure. Using the matrix notation defined in the previous section, notice that the new iterate satisfies the relation

$$x_{new} = x + \sum_{i=1}^s T_i(b - Ax) = \left( I - \sum_{i=1}^s P_i \right) x + \sum_{i=1}^s T_i b. \tag{197}$$

Thus, using the same analogy as in the previous section, this iteration corresponds to a fixed-point iteration  $x_{new} = Gx + f$  with

$$G = I - \sum_{i=1}^s P_i, f = \sum_{i=1}^s T_i b. \tag{198}$$

With the relation  $G = I - M^{-1}A$  between  $G$  and the preconditioning matrix  $M$ , the result is that

$$M^{-1}A = \sum_{i=1}^s P_i \quad (199)$$

and

$$M^{-1} = \sum_{i=1}^s P_i A^{-1} = \sum_{i=1}^s T_i. \quad (200)$$

Now the procedure for applying the preconditioned operator  $M^{-1}$  becomes:

**ADDITIVE SCHWARZ PRECONDITIONER**

1. Input:  $v$ ; Output:  $z = M^{-1}v$ .
2. For  $i = 1, \dots, s$  Do:
3. Compute  $z_i := T_i v$
4. EndDo
5. Compute  $z := z_1 + z_2 \dots + z_s$ .

Note that the do loop can be performed in parallel. Step 5 sums up the vectors  $z_i$  in each domain to obtain a global vector  $z$ . In the nonoverlapping case, this step is parallel and consists of just forming these different components since the addition is trivial. In the presence of overlap, the situation is similar except that the overlapping components are added up from the different results obtained in each subdomain.

The procedure for computing  $M^{-1}Av$  is identical to the one above except that  $T_i$  in line 3 is replaced by  $P_i$ .

### 3.5.3 Schur Complement Approaches

Schur complement methods are based on solving the reduced system (145) by some preconditioned Krylov subspace method. Procedures of this type involve three steps.

1. Get the right-hand side  $g' = g - FB^{-1}f$ .
2. Solve the reduced system  $Sy = g'$  via an iterative method.
3. Back-substitute, i.e., compute  $x$  via (144)

The different methods relate to the way in which step 2 is performed. First observe that the matrix  $S$  need not be formed explicitly in order to solve the reduced system by an iterative method. For example, if a Krylov subspace method without preconditioning is used, then the only operations that are required with the matrix  $S$  are matrix-by-vector operations  $w = Sv$ . Such operations can be performed as follows.

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

The above procedure involves only matrix-by-vector multiplications and one linear system solution with  $B$ . Recall that a linear system involving  $B$  translates into  $s$ -independent linear systems. Also note that the linear systems with  $B$  must be solved exactly, either by a direct solution technique or by an iterative technique with a high level of accuracy.

While matrix-by-vector multiplications with  $S$  cause little difficulty, it is much harder to precondition the matrix  $S$ , since this full matrix is often not available explicitly. There have been a number of methods, derived mostly using arguments from Partial Differential Equations to precondition the Schur complement. Here, we consider only those preconditioners that are derived from a linear algebra viewpoint.

### INDUCED PRECONDITIONERS

One of the easiest ways to derive an approximation to  $S$  is to exploit (149), the properties of the Schur Complement and the intimate relation between the Schur complement and Gaussian elimination. This proposition tells us that a preconditioning operator  $M$  to  $S$  can be defined from the (approximate) solution obtained with  $A$ . To precondition a given vector  $v$ , i.e., to compute  $w = M^{-1}v$ , where  $M$  is the desired preconditioner to  $S$ , first solve the system

$$A \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ v \end{pmatrix}, \quad (201)$$

then take  $w = y$ . Use any approximate solution technique to solve the above system. Let  $M_A$  be any preconditioner for  $A$ . Using the notation defined earlier, let  $R_y$  represent the restriction operator on the interface variables, as defined in (149). Then the preconditioning operation for  $S$  which is induced from  $M_A$  is defined by

$$M_S^{-1}v = R_y M_A^{-1} \begin{pmatrix} 0 \\ v \end{pmatrix} = R_y M_A^{-1} R_y^T v. \quad (202)$$

Observe that when  $M_A$  is an exact preconditioner, i.e., when  $M_A = A$ , then according to (149),  $M_S$  is also an exact preconditioner, i.e.,  $M_S = S$ . This induced preconditioner can be expressed as

$$M_S = (R_y M_A^{-1} R_y^T)^{-1}. \quad (203)$$

It may be argued that this uses a preconditioner related to the original problem to be solved in the first place. However, even though the preconditioning on  $S$  may be defined from a preconditioning of  $A$ , the linear system is being solved for the interface variables. That is typically much smaller than the original linear system. For example, GMRES can be used with a much larger dimension of the Krylov subspace since the Arnoldi vectors to keep in memory are much smaller. Also note that from a Partial Differential Equations viewpoint, systems of the form (201) correspond to the Laplace equation, the solutions of which are Harmonic functions. There are fast techniques which provide the solution of such equations inexpensively.

In the case where  $M_A$  is an ILU factorization of  $A$ ,  $M_S$  can be expressed in an explicit form in terms of the entries of the factors of  $M_A$ . This defines a preconditioner to  $S$  that is induced canonically from an incomplete LU factorization of  $A$ . Assume that the preconditioner  $M_A$  is in a factored form  $M_A = L_A U_A$ , where

$$L_A = \begin{pmatrix} L_B & 0 \\ F U_B^{-1} & L_S \end{pmatrix} \text{ and } U_A = \begin{pmatrix} U_B & L_B^{-1} E \\ 0 & U_S \end{pmatrix}. \quad (204)$$

Then, the inverse of  $M_A$  will have the following structure:

$$M_A^{-1} = U_A^{-1} L_A^{-1} = \begin{pmatrix} \star & \star \\ 0 & U_S^{-1} \end{pmatrix} \begin{pmatrix} \star & 0 \\ \star & L_S^{-1} \end{pmatrix} = (\star \quad \star \quad \star \quad U_S^{-1} L_S^{-1}) \quad (205)$$

where a star denotes a matrix whose actual expression is unimportant. Recall that by definition,

$$R_y = (0 \ I), \quad (206)$$

where this partitioning conforms to the above ones. This means that

$$R_y M_A^{-1} R_y^T = U_S^{-1} L_S^{-1} \quad (207)$$

and, therefore, according to (203),  $M_S = L_S U_S$ . The  $L$  and  $U$  factors for  $M_S$  are the  $(2, 2)$  blocks of the  $L$  and  $U$  factors of the ILU factorization of  $A$ . An important consequence of the above idea is that the parallel Gaussian elimination can be exploited for deriving an ILU preconditioner for  $S$  by using a general purpose ILU factorization. In fact, the  $L$  and  $U$  factors of  $M_A$  have the following structure:

$$A = L_A U_A - R \quad (208)$$

$$L_A = \begin{pmatrix} L_1 & & & & & \\ & L_2 & & & & \\ & & \ddots & & & \\ & & & L_s & & \\ F_1 U_1^{-1} & F_2 U_2^{-1} & \dots & F_s U_s^{-1} & L & \end{pmatrix} \quad (209)$$

$$U_A = \begin{pmatrix} U_1 & & & L_1^{-1} E_1 \\ & U_2 & & L_2^{-1} E_2 \\ & & \dots & \vdots \\ & & & U_s & L_s^{-1} E_s \\ & & & & U \end{pmatrix}. \quad (210)$$

Each  $L_i, U_i$  pair is an incomplete LU factorization of the local  $B_i$  matrix. These ILU factorizations can be computed independently. Similarly, the matrices  $L_i^{-1} E_i$  and  $F_i U_i^{-1}$  can also be computed independently once the LU factors are obtained. Then each of the matrices

$$\tilde{S}_i = C_i - F_i U_i^{-1} L_i^{-1} E_i, \quad (211)$$

which are the approximate local Schur complements, is obtained. Note that since an incomplete LU factorization is being performed, some drop strategy is applied to the elements in  $\tilde{S}_i$ . Let  $T_i$  be the matrix obtained after this is done,

$$T_i = \tilde{S}_i - R_i. \quad (212)$$

Then a final stage would be to compute the ILU factorization of the matrix (158) where each  $S_i$  is replaced by  $T_i$ .

### PRECONDITIONING VERTEX-BASED SCHUR COMPLEMENTS

We now discuss some issues related to the preconditioning of a linear system with the matrix coefficient of (158) associated with a vertex-based partitioning. As was mentioned before, this structure is helpful in the direct solution context because it allows the Schur complement to be formed by local pieces. Since incomplete LU factorizations will utilize the same structure, this can be exploited as well.

Note that multicolor SOR or SSOR can also be exploited and that graph coloring can be used to color the interface values  $y_i$  in such a way that no two adjacent interface variables will have the same color. In fact, this can be achieved by coloring the domains. In the course of a multicolor block-SOR iteration, a linear system must be solved with the diagonal blocks  $S_i$ . For this purpose, it is helpful to interpret the Schur complement. Call

$P$  the canonical injection matrix from the local interface points to the local nodes. If  $n_i$  points are local and if  $m_i$  is the number of the local interface points, then  $P$  is an  $n_i \times m_i$  matrix whose columns are the last  $m_i$  columns of the  $n_i \times n_i$  identity matrix. Then it is easy to see that

$$S_i = (P^T A_{loc,i}^{-1} P)^{-1} \quad (213)$$

If  $A_{loc,i} = LU$  is the LU factorization of  $A_{loc,i}$  then it can be verified that

$$S_i^{-1} = P^T U^{-1} L^{-1} P = P^T U^{-1} P P^T L^{-1} P, \quad (214)$$

which indicates that in order to operate with  $P^T L^{-1} P$ , the last  $m_i \times m_i$  principal submatrix of  $L$  must be used. The same is true for  $P^T U^{-1} P$  which requires only a back-solve with the last  $m_i \times m_i$  principal submatrix of  $U$ . Therefore, only the LU factorization of  $A_{loc,i}$  is needed to solve a system with the matrix  $S_i$ . Interestingly, approximate solution methods associated with incomplete factorizations of  $A_{loc,i}$  can be exploited.

### 3.6 Deflation

Deflation is an attempt to treat the bad eigenvalues resulting in the preconditioned matrix.

$$M^{-1} A x = M^{-1} b \quad (215)$$

$M^{-1} A$ , where  $M^{-1}$  is a symmetric positive definite (SPD) preconditioner and  $A$  is the symmetric positive definite (SPD) coefficient matrix. This operation reduces the convergence iterations for the Preconditioned Conjugate Gradient (PCG) method and makes it more robust.

The original linear system

$$A x = b \quad (216)$$

can be solved by employing the splitting

$$x = (I - P^T) x + P^T x \quad (217)$$

Simplifying we get

$$x = (I - P^T) x + P^T x \Leftrightarrow x = Q b + P^T x \quad (218)$$

$$\Leftrightarrow A x = A Q b + A P^T x \quad (219)$$

$$\Leftrightarrow b = A Q b + P A x \quad (220)$$

$$\Leftrightarrow P b = P A x, \quad (221)$$

where

$$P := I - A Q, Q := Z E^{-1} Z^T, E := Z^T A Z. \quad (222)$$

where

$E \in \mathbb{R}^{k \times k}$  is the invertible Galerkin Matrix,  
 $Q \in \mathbb{R}^{n \times n}$  is the correction Matrix,  
and  $P \in \mathbb{R}^{n \times n}$  is the deflation matrix.

Also it is given that  $A$  is an SPSD coefficient matrix as given in (216) and  $Z \in \mathbb{R}^{n \times k}$ , with full rank and  $k < n - d$  is given.

The  $x$  at the end of the expression is not necessarily a solution of the original linear system (216), since it might consist of components of the null space of  $PA$ ,  $\mathcal{N}(PA)$ .

Therefore this 'deflated' solution is denoted as  $\hat{x}$  rather than  $x$ . The deflated system is now,

$$PA\hat{x} = Pb \quad (223)$$

The Preconditioned deflated version of the Conjugate Gradient Method can now be presented. The deflated method (223) can be solved using a symmetric positive definite (SPD) preconditioner,  $M^{-1}$ . We therefore now seek a solution to

$$\tilde{P}\tilde{A}\hat{x} = \tilde{P}\tilde{b}, \quad (224)$$

where

$$\tilde{A} := M^{-\frac{1}{2}}AM^{-\frac{1}{2}}, \hat{x} := M^{\frac{1}{2}}\hat{x}, \tilde{b} := M^{-\frac{1}{2}}b, \quad (225)$$

and

$$\tilde{P} := I - \tilde{A}\tilde{Q}, \tilde{Q} := \tilde{Z}\tilde{E}^{-1}\tilde{Z}^T, \tilde{E} := \tilde{Z}^T\tilde{A}\tilde{Z}, \quad (226)$$

where  $\tilde{Z} \in \mathbb{R}^{n \times k}$  can be interpreted as a preconditioned deflation-subspace matrix. The resulting method is called the Deflated Preconditioned Conjugate Gradient (DPCG) method [Tang, 2008].

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  $j:=0, \dots$ , until convergence do
3.  $\hat{w}_j := PAp_j$
4.  $\alpha_j := \frac{(\hat{r}_j, y_j)}{(p_j, \hat{w}_j)}$
5.  $\hat{x}_{j+1} := \hat{x}_j + \alpha_j p_j$
6.  $\hat{r}_{j+1} := \hat{r}_j - \alpha_j \hat{w}_j$
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.  $x_{it} := Qb + P^T x_{j+1}$

It can be seen that  $\tilde{P}$  or  $M^{\frac{1}{2}}$  are never calculated explicitly. Hence the linear system is often denoted by

$$M^{-1}PA\hat{x} = M^{-1}Pb \quad (227)$$

Some Observations:

All known properties of Preconditioned Conjugate Gradient (PCG) also hold for DPCG, where  $PA$  can be interpreted as the coefficient matrix  $A$  in (124). Moreover if  $P = I$  is taken the algorithm above reduces to the PCG algorithm.

Careful selection of Deflation vectors is required for this method to prove useful. Two methods, one based on eigenvector (of  $M^{-1}A$ ) based subspace for  $Z$  and the other based on an arbitrary choice of the deflation subspace, are worth to mention.

However to calculate the eigenvectors itself could be computationally intensive so an arbitrary choice which closely resembles the part of the eigenspace is the way out. In short the ideal deflation method should satisfy the following criteria:

- The deflation-subspace matrix  $Z$  must be sparse;
- The deflation vectors approximate the eigenspace corresponding to the unfavorable eigenvalues;
- The cost of constructing deflation vectors is relatively low;
- The method has favorable parallel properties;
- The approach can be easily implemented in an existing PCG code.

Subdomain Deflation based choice for Deflation vectors seems to emerge as a close match.

### 3.6.1 Subdomain Deflation

In Subdomain deflation, the deflation vectors are chosen in an algebraic way. The computational domain is divided into several subdomains, where each subdomain corresponds to one or more deflation vectors.

Consider application of Subdomain deflation to Poisson Equation with discontinuous coefficients(also called the 'pressure(-correction) equation') and Neumann boundary conditions, i.e.,

$$-\nabla \cdot \left( \frac{1}{\rho(x)} \nabla p(x) \right) = f(x), x \in \Omega, \quad (228)$$

$$\frac{\partial}{\partial n} p(x) = g(x), x \in \partial\Omega, \quad (229)$$

where  $\Omega, p, \rho, x$ , and  $n$  denote the computational domain, pressure, density, spatial coordinates, and the unit normal vector to the boundary  $\partial\Omega$ , respectively.  $g$  is such that mass is conserved.

Now assume that the computational domain,  $\Omega$  is divided into several subdomains,  $\Omega_j$ , where each  $\Omega_j$  corresponds to one deflation vector, consisting of ones for grid points in the interior of the discretized subdomain,  $\Omega_{h_j}$ , and zeroes for other grid points. Then, subdomain deflation is effective, if each subdomain,  $\Omega_j$ , corresponds to exactly one constant part of the coefficient,  $\rho$ . In this case, the subspace spanned by the deflation vectors is proved to be almost equal to the eigenspace associated with the smallest eigenvalues.

## 4 Parallel Iterative Methods

There have been two traditional approaches in parallelizing solutions through iterative methods.

- Extracting Parallelism from Standard algorithms.
- Devise new algorithms that have inherently better parallelism.

We will look at the first approach and discuss some of the approaches of the latter kind in this section. For our problem we use Conjugate Gradient Method. The main operations in Preconditioned Conjugate Gradient Algorithm are:

1. Preconditioner setup.
2. Matrix vector multiplications.

3. Vector updates.
4. Dot products.
5. Preconditioning operations.

Operations 1 and 5 are the potential bottlenecks followed by Matrix Vector Multiplications.

For speeding up the main operations in preconditioned methods, Level Scheduling could be put to use. We take it up in the following sections.

## 4.1 Parallel Implementations

In this section we look at the implementation of standard operations and how the matrices could be stored in an efficient way to speed up common operations.

### 4.1.1 Matrix-Vector Products

Matrix-by-vector multiplications (sometimes called Matvecs for short) are relatively easy to implement efficiently on high performance computers. We will cover sparse Matvec operations for a few different storage formats.

One of the most general schemes for storing sparse matrices is the Compressed Sparse Row storage format. Recall that the data structure consists of three arrays: a real array  $A(1 : nnz)$  to store the nonzero elements of the matrix row-wise, an integer array  $JA(1 : nnz)$  to store the column positions of the elements in the real array  $A$ , and, finally, a pointer array  $IA(1 : n + 1)$ , the  $i$ -th entry of which points to the beginning of the  $i$ -th row in the arrays  $A$  and  $JA$ . To perform the matrix-by-vector product  $y = Ax$  in parallel using this format, note that each component of the resulting vector can be computed independently as the dot product of the  $i$ -th row of the matrix with the vector

CSR Format -Dot Product Form

1. Do  $i = 1, n$
2.  $k1 = ia(i)$
3.  $k2 = ia(I + 1) - 1$
4.  $y(i) = \text{dotproduct}(a(k1 : k2), x(ja(k1 : k2)))$
5. EndDo

Line 4 of the above algorithm computes the dot product of the vector with components  $a(k1), a(k1 + 1), \dots, a(k2)$  with the vector with components  $x(ja(k1)), x(ja(k1 + 1)), \dots, x(ja(k2))$ . The fact that the outer loop can be performed in parallel can be exploited on any parallel platform. On some shared-memory machines, the synchronization of this outer loop is inexpensive and the performance of the above program can be excellent. On distributed memory machines, the outer loop can be split into a number of steps to be executed on each processor. Thus, each processor will handle a few rows that are assigned to it. It is common to assign a certain number of rows (often contiguous) to each processor and to also assign the component of each of the vectors similarly. The part of the matrix that is needed is loaded in each processor initially. When performing a matrix-by-vector product, interprocessor communication will be necessary to get the needed components of the vector  $x$  that do not reside in a given processor.

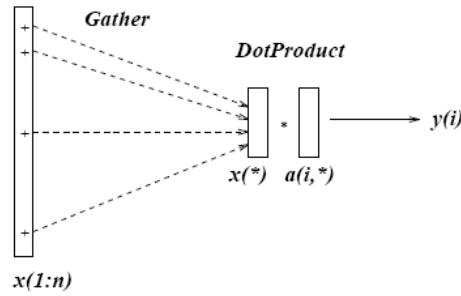


Figure 13: Illustration of the row-oriented matrix-by-vector multiplication

The indirect addressing involved in the second vector in the dot product is called a *gather* operation. The vector  $x(ja(k1 : k2))$  is first gathered from memory into a vector of contiguous elements. The dot product is then carried out as a standard dot-product operation between two dense vectors.

Now assume that the matrix is stored by columns (CSC format). The matrix-by-vector product can be performed by the following algorithm.

1.  $y(1 : n) = 0.0$
2. Do  $i = 1, n$
3.  $k1 = ia(i)$
4.  $k2 = ia(i + 1) - 1$
5.  $y(ja(k1 : k2)) = y(ja(k1 : k2)) + x(j) * a(k1 : k2)$
6. EndDo

The above code initializes  $y$  to zero and then adds the vectors  $x(j) \times a(1 : n, j)$  for  $j = 1, \dots, n$  to it. It can also be used to compute the product of the transpose of a matrix by a vector, when the matrix is stored (row-wise) in the CSR format. Normally, the vector  $y(ja(k1 : k2))$  is gathered and the SAXPY operation is performed in vector mode. Then the resulting vector is scattered back into the positions  $ja(*)$ , by what is called a *Scatter* operation.

MATVECS in Diagonal Format

The above storage schemes are general but they do not exploit any special structure of the matrix. The diagonal storage format was one of the first data structures used in the context of high performance computing to take advantage of special sparse structures. Often, sparse matrices consist of a small number of diagonals in which case the matrix-by-vector product can be performed by diagonals. For sparse matrices, most of the  $2n - 1$  diagonals are zero. Recall that the matrix is stored in a rectangular *arraydiag*( $1 : n, 1 : ndiag$ ) and the offsets of these diagonals from the main diagonal may be stored in a small integer array *offset*( $1 : ndiag$ ).

1. Do  $i = 1, n$
2.  $tmp = 0.0$
3. Do  $j = 1, ndiag$

4.  $tmp = tmp + \text{diag}(i, j) * x(i + \text{offset}(j))$
5. EndDo
6.  $y(i) = tmp$
7. EndDo

One drawback with diagonal storage is that it is not general enough. For general sparse matrices, we can either generalize the diagonal storage scheme or reorder the matrix in order to obtain a diagonal structure. The simplest generalization is the Ellpack-Itpack Format.

The Ellpack-Itpack (or Ellpack) format is of interest only for matrices whose maximum number of nonzeros per row,  $jmax$ , is small. The nonzero entries are stored in a real array  $ae(1:n, 1:jmax)$ . Along with this is integer array  $jae(1:n, 1:jmax)$  which stores the column indices of each corresponding entry in  $ae$ .

1. Do  $i = 1, n$
2.  $yi = 0$
3. Do  $j = 1, ncol$
4.  $yi = yi + ae(j, i) * x(jae(j, i))$
5. EndDo
6.  $y(i) = yi$
7. EndDo

The main difference between these loops and the previous ones for the diagonal format is the presence of indirect addressing in the innermost computation. A disadvantage of the Ellpack format is that if the number of nonzero elements per row varies substantially, many zero elements must be stored unnecessarily. Then the scheme becomes inefficient. As an extreme example, if all rows are very sparse except for one of them which is full, then the arrays  $ae$ ,  $jae$  must be full  $n \times n$  arrays, containing mostly zeros. This is remedied by a variant of the format which is called the *jagged diagonal format*.

A more general alternative to the diagonal or Ellpack format is the Jagged Diagonal (JAD) format. This can be viewed as a generalization of the Ellpack-Itpack format which removes the assumption on the fixed length rows. To build the jagged diagonal structure, start from the CSR data structure and sort the rows of the matrix by decreasing number of nonzero elements. To build the first jagged diagonal (j-diagonal), extract the first element from each row of the CSR data structure. The second jagged diagonal consists of the second elements of each row in the CSR data structure. The third, fourth, ..., jagged diagonals can then be extracted in the same fashion. The lengths of the successive j-diagonals decreases. The number of j-diagonals that can be extracted is equal to the number of nonzero elements of the first row of the permuted matrix, i.e., to the largest number of nonzero elements per row. To store this data structure, three arrays are needed: a real array  $DJ$  to store the values of the jagged diagonals, the associated array  $JDIAG$  which stores the column positions of these values, and a pointer array  $IDIAG$  which points to the beginning of each j-diagonal in the  $DJ$ ,  $JDIAG$  arrays.

A matrix-by-vector product with this storage scheme can be performed by the following code segment.

1. Do  $j = 1, ndiag$

2.  $k1 = \text{idiag}(j)$
3.  $k2 = \text{idiag}(j + 1)1$
4.  $len = \text{idiag}(j + 1)k1$
5.  $y(1 : len) = y(1 : len) + dj(k1 : k2) * x(j\text{diag}(k1 : k2))$
6. EndDo

Since the rows of the matrix  $A$  have been permuted, the above code will compute  $JAx$ , a permutation of the vector  $Ax$ , rather than the desired  $Ax$ . It is possible to permute the result back to the original ordering after the execution of the above program. This operation can also be performed until the final solution has been computed, so that only two permutations on the solution vector are needed, one at the beginning and one at the end. For preconditioning operations, it may be necessary to perform a permutation before or within each call to the preconditioning subroutines.

#### 4.1.2 Level Scheduling: The case of 5-Point Matrices

Consider an example which consists of a 5-point matrix associated with a  $4 \times 3$  mesh as represented in Figure 14. The lower triangular matrix associated with this mesh is represented in the left side of Figure 14. The stencil represented in the right side of Figure 14 establishes the data dependence between the unknowns in the lower triangular system solution when considered from the point of view of a grid of unknowns. It tells us that in order to compute the unknown in position  $(i, j)$ , only the two unknowns in positions  $(i - 1, j)$  and  $(i, j - 1)$  are needed. The unknown  $x_{11}$  does not depend on any other variable and can be computed first. Then the value of  $x_{11}$  can be used to get  $x_{1,2}$  and  $x_{2,1}$  simultaneously. Then these two values will in turn enable  $x_{3,1}, x_{2,2}$  and  $x_{1,3}$  to be obtained simultaneously, and so on. Thus, the computation can proceed in wavefronts. The steps for this wavefront algorithm are shown with dashed lines in Figure 14. Observe that the maximum degree of parallelism (or vector length, in the case of vector processing) that can be reached is the minimum of  $n_x, n_y$  the number of mesh points in the  $x$  and  $y$  directions, respectively, for 2-D problems. For 3-D problems, the parallelism is of the order of the maximum size of the sets of domain points  $x_{i,j,k}$ , where  $i + j + k = lev$ , a constant level  $lev$ . It is important to note that there is little parallelism or vectorization at the beginning and at the end of the sweep. The degree of parallelism is equal to one initially, and then increases by one for each wave reaching its maximum, and then decreasing back down to one at the end of the sweep. For example, for a  $4 \times 3$  grid, the levels (sets of equations that can be solved in parallel) are 1, 2,5, 3,6,9, 4,7,10, 8,11, and finally 12. The first and last few steps may take a heavy toll on achievable speed-ups. An implementation of this technique to speed up the iterations for GMRES is discussed in [Vuik, van Nooyen, and Wesseling, 1998].

The idea of proceeding by levels or wavefronts is a natural one for finite difference matrices on rectangles.

## 4.2 Preconditioning in Parallel

### 4.2.1 Multi-Coloring

Multicoloring could be useful for exploiting parallelism in iterative solution techniques.

Red-black ordering involves numbering the grid points of a discretized problem alternately with alternating colors. The condition being that two adjacent nodes do not have similar colors.

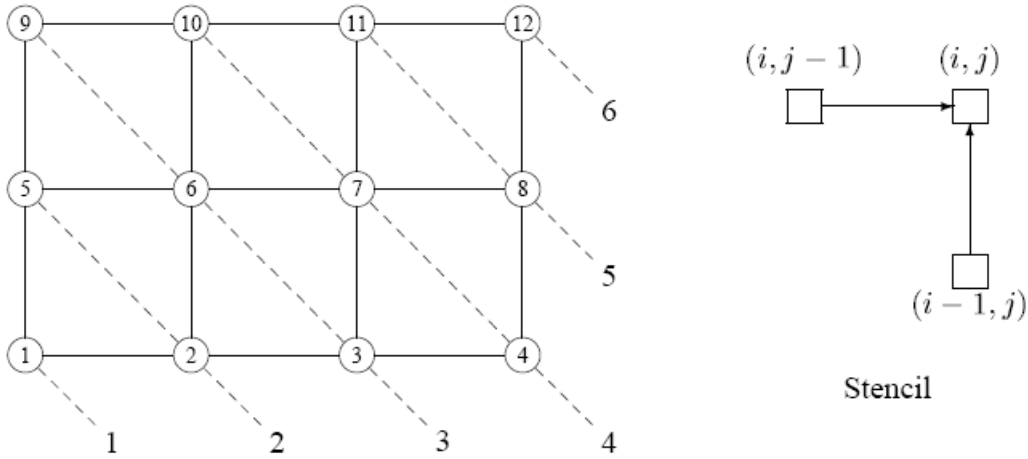


Figure 14: Level Scheduling for a  $4 \times 3$  grid problem.

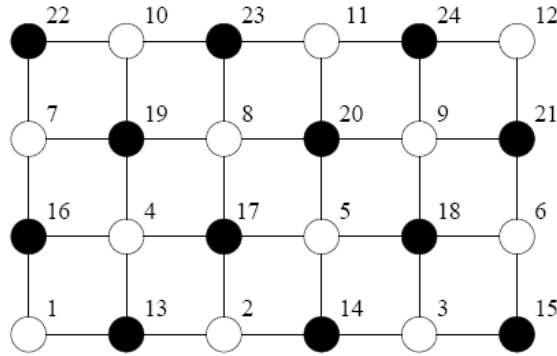


Figure 15: Red Black Ordering

Since the red nodes are not coupled with other red nodes and, similarly, the black nodes are not coupled with other black nodes, the system that results from this reordering will have the structure

$$\begin{pmatrix} D_1 & F \\ E & D_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \quad (230)$$

The easiest way to solve such a red-black system is to use SSOR or ILU(0) preconditioning. The linear system that arises from the forward solve in SSOR will have the form

$$\begin{pmatrix} D_1 & O \\ E & D_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \quad (231)$$

This system can be solved by performing the following sequence of operations:

1. Solve  $D_1 x_1 = b_1$ .
2. Compute  $\hat{b}_2 = b_2 - E x_1$ .
3. Solve  $D_2 x_2 = \hat{b}_2$ .

This consists of two diagonal scalings (operations 1 and 3) and a sparse matrix-vector product. Therefore, the degree of parallelism, is at least  $n/2$  if an atomic task is considered to be any arithmetic operation. The situation is identical with the ILU(0) preconditioning.

However, since the matrix has been reordered before  $ILU(0)$  is applied to it, the resulting LU factors are not related in any simple way to those associated with the original matrix. In fact, a simple look at the structure of the ILU factors reveals that many more elements are dropped with the red-black ordering than with the natural ordering. The result is that the number of iterations to achieve convergence can be much higher with red-black ordering than with the natural ordering.

Keeping in mind a greedy technique for coloring a graph. Given a general sparse matrix  $A$ , this inexpensive technique allows us to reorder it into a block form where the diagonal blocks are diagonal matrices. The number of blocks is the number of colors. For example, for six colors, a matrix would result with the structure shown in below

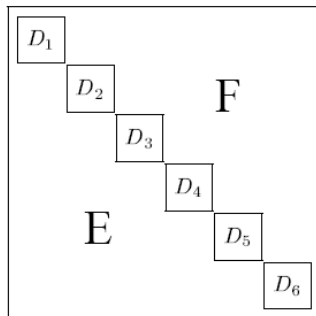


Figure 16: A six-color ordering of a general sparse matrix.

where the  $D$ s are diagonal and  $E$ ,  $F$  are general sparse. This structure is obviously a generalization of the red-black ordering.

Just as for the red-black ordering,  $ILU(0)$ , SOR, or SSOR preconditioning can be used on this reordered system. The parallelism of SOR/SSOR is now of order  $n/p$  where  $p$  is the number of colors. A loss in efficiency may occur since the number of iterations is likely to increase. A Gauss-Seidel sweep will essentially consist of  $p$  scalings and  $p - 1$  matrix-by-vector products, where  $p$  is the number of colors.

#### 4.2.2 Multi-Elimination ILU

The discussion in this section begins with the Gaussian elimination algorithm for a general sparse linear system. Parallelism in sparse Gaussian elimination can be obtained by finding unknowns that are independent at a given stage of the elimination, i.e., unknowns that do not depend on each other according to the binary relation defined by the graph of the matrix. A set of unknowns of a linear system which are independent is called an independent set. Thus, independent set orderings can be viewed as permutations to put the original matrix in the form

$$\begin{pmatrix} D & E \\ F & C \end{pmatrix} \quad (232)$$

in which  $D$  is diagonal, but  $C$  can be arbitrary. This amounts to a less restrictive form of multicoloring, in which a set of vertices in the adjacency graph is found so that no equation in the set involves unknowns from the same set. The rows associated with an independent set can be used as pivots simultaneously. When such rows are eliminated, a smaller linear system results, which is again sparse. Then we can find an independent set for this reduced system and repeat the process of reduction. The resulting second reduced system is called the second-level reduced system. The process can be repeated recursively a few times. As the level of the reduction increases, the reduced systems gradually lose their sparsity. A direct solution method would continue the reduction until the reduced

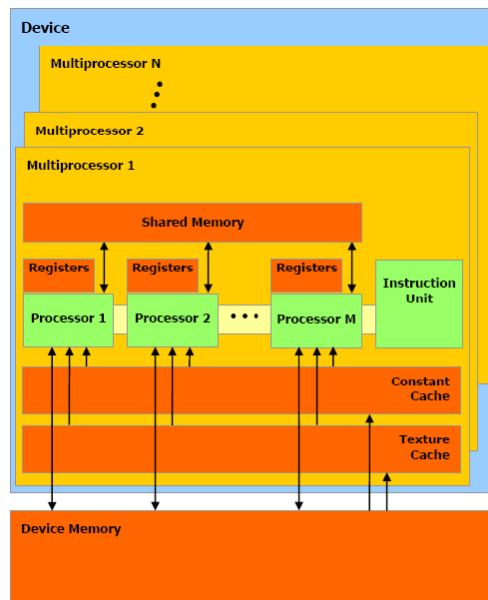
system is small enough or dense enough to switch to a dense Gaussian elimination to solve it.

## 5 Scientific Computing on GPUs

GPUs commercially available of the shelf, promise up to 1 teraflop (single precision) of compute power. The cost at which this performance is available is a couple of hundred euros. This makes it an attractive option already compared to setting up or sharing a cluster that might be hard to get access to or costlier to put up in the first place. Scientific computing problems could be restructured with some effort to work on the GPUs and it is not uncommon to get upto 20x speed-up with simple first implementation. Talking about implementation, it has become a lot easier with the advent of CUDA (Compute Unified Device Architecture) from NVIDIA to write C code that runs on the GPU. Earlier this was not the case. In order to harness the capabilities of the GPU C programs had to be adapted to the shader languages like Cg .The programmer had to understand the way how a GPU interprets textures and objects in a rendering environment and he had to explicitly mould the application code as if it were a rendering operation.

### 5.1 Background Information on GPU Architecture and Programming Model

The GPU is a SIMD processor (Single Instruction Multiple Data). What this means is that there is an army of processors waiting to crunch the problem computations, all the processors execute the exact same instructions but they do so in parallel without any dependence. The result is that if one of them is able to say execute only at a rate of 1 Ghz, then if a thousand of them do it simultaneously we get a teraflop straightforwardly. A GPU has some fixed number of processors within, these are called streaming multiprocessors (SMs). Each multiprocessor further has eight scalar processors (SPs).



A set of SIMD multiprocessors with on-chip shared memory.

Figure 17: GPU Architecture

### 5.1.1 Internal Organization

There is a layer of abstraction built upon the physical hardware inside a GPU card. The threads are internally arranged into blocks and there is a grid of such blocks. When executing a computation on the GPU the programmer has to specify how many of such threads exist in each block and how many such blocks exist in the grid. Blocks of threads are divided amongst the streaming multiprocessors. The threads and blocks have three dimensions. So each thread or block is identified by an index that has an  $x, y$  and  $z$  co-ordinate inside a block. Likewise a block inside a grid lives in 3D space. This comes in handy when using matrices (which are commonly used in scientific simulations), since each matrix element could be easily assigned to a thread and could be addressed by a thread Id.

### 5.1.2 Execution of Threads

Threads inside a block are grouped into warps. A multiprocessor on the GPU is assigned some number of blocks. The scheduler that picks up threads for execution, does so in granularity of a warp. So, if the warp size is say, 32, it will pick 32 threads with consecutive thread Ids and schedule them for execution in the next cycle.

Each thread executes on one of the scalar processors. The Multiprocessors are capable of executing a number of warps simultaneously. This number can vary from 512 upto 1024 on the GPUs depending on the type of card. At the time of issue from the scheduler the SMs are handed over a number of blocks to execute. These can vary on the requirement that each thread imposes in terms of registers and shared memory since they are limited on each multiprocessor.

For example suppose that maximum number of threads that can be scheduled on a multiprocessor is 768. Further if each warp is composed of 32 threads then we can have maximum of 24 warps. Now many possible schemes for division could be laid down:

1. 256 threads per block \* 3 blocks
2. 128 threads per block \* 6 blocks
3. 16 threads per block \* 48 blocks

Now each SM has a restriction on the number of blocks that can simultaneously run on it. So if the maximum number of blocks is say 8 then only the 1st and 2nd schemes could form a valid execution configuration.

### 5.1.3 Memory Model

Each multiprocessor has a set of memories associated with it. These memories have different access times. It must be noted that these memories are on the device (the GPU) and not on the DRAM available with the CPU. They are:

1. Register Memory

There are fixed number of registers that must be shared amongst the number of threads that are configured (by the previously discussed allocation of threads in blocks and grids). Registers are exclusive to each thread.

2. Shared Memory

Shared memory is accessible to all the threads within a block. It is the next best thing after registers since accessing it is cheaper than the global memory.

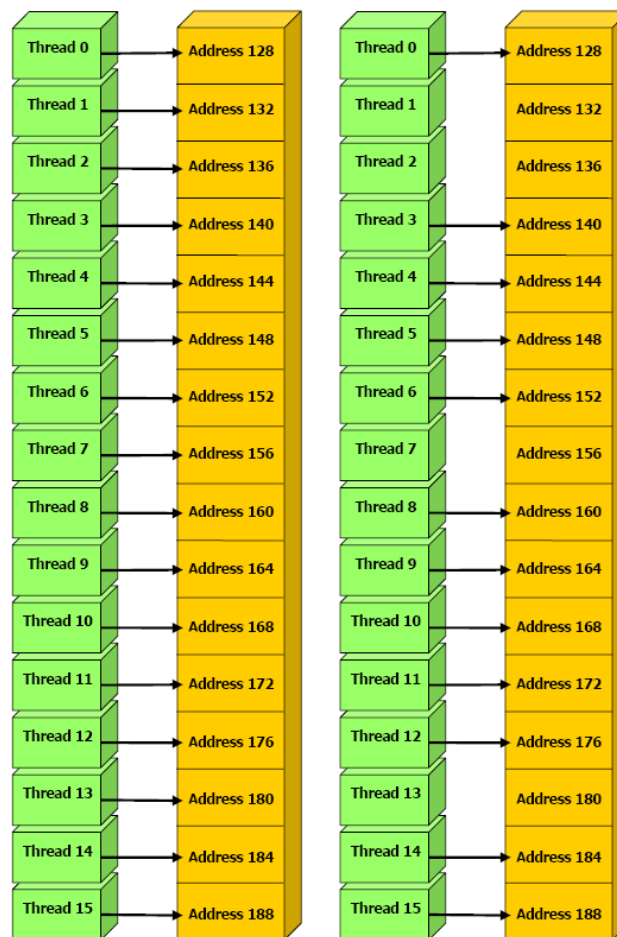
### 3. Texture Memory

Texture memory is read-only and could be read by all the threads across blocks on a single multiprocessor.

### 4. Global Memory

This memory is the biggest in size and is placed farthest from the threads executing on the multiprocessors. Its access times compared to the shared memory latency might be up to 200 times more.

An important consequence of the different access times of the available memories is that it often becomes important to hide the latency with available computation. Register Usage can also decide how many blocks (consequently how many threads) might execute in parallel. So suppose if we have 8192 registers and each thread (in a  $16 \times 16$  block) uses 10 registers then the maximum number of blocks that can execute on a SM is 3 since  $10 * 256 = 2560$  and  $8192/2560$  gives 3 (integer) as result. Increasing by only one register per thread reduces the number of blocks by 1.



Left: coalesced float memory access.  
Right: coalesced float memory access (divergent warp).

Figure 18: Coalesced Memory Accesses in a GPU

Global memory access is done in a regular fashion (regularly spaced or aligned) and is consistent within a warp (all accesses are at equal strides or increments) then the data can be brought from global memory in less number of instructions. This alignment called coalescing could yield performance benefits when data is neatly aligned.

### 5.1.4 Usability for Scientific Computing

Previous work [Bolz, Farmer, Grinspun, and Schröder, 2003] [Krüger and Westermann, 2003] has shown the suitability of the data parallel architectures for solving sparse matrices using iterative methods. However most of these approaches used the then available shader languages and texture manipulation paradigms to write the application. Some of the approaches [de Sturler and Loher, 1998] used other parallelization primitives like HPF for such solvers. A variety of interesting implementations have come up in the last few years which use the CUDA API for adapting the Iterative Solvers on the GPU. We discuss them next.

## 6 Parallel Iterative Methods on the GPU

Solving with FFTs[Shi, Cai, Hou, Ma, Tan, Ho, and Wang, 2009].

### 6.1 Mixed Precision Techniques on the GPU

It is possible to use double precision calculations for some part of the iterative method and use single precision for others thereby achieving a tradeoff that meets precision criteria and converges as good as the double precision case. At the same time the rate of convergence is also not affected very much. [Baboulin, Buttari, Dongarra, Kurzak, Langou, Langou, Luszczek, and Tomov, 2008] use a similar approach for direct and iterative solution method for sparse systems where in they pose the problem as the refinement of the solution  $x_{i+1}$  which can be written as:

$$x_{i+1} = x_i + M(b - Ax_i), \quad (233)$$

where  $M$  is the preconditioner and approximates  $S^{-1}$ . If we use right preconditioning then the system  $Ax = b$  reduces to the following

$$AMu = b, \quad (234)$$

$$x = Mu \quad (235)$$

Further they suggest calculating  $M$  using an iterative method and for the solution of the original system a Krylov Subspace method is employed. This system of Inner and Outer Iteration is now ready for Mixed Precision use. The idea here is that a single precision arithmetic matrix-vector product is used as a fast approximation of the double precision operator in the inner iterative solver. They have reported the results for a non-symmetric solver wherein the outer iteration is of a FMGRES and the inner one is a GMRES cycle.

They report a speedup of 15 for some selected test problems.

### 6.2 Sparse Matrix Vector Products- SpMVs

Using the prefix sum technique in a sparse matrix vector multiply it is possible to speedup this operation. The proposed method can achieve a scan operation with  $O(n)$  complexity for a problem size of  $n$ . It has been further explored [SenGupta, Harris, Zhang, and Owens, 2007] and extended as a primitive to implement operations like Sparse Matrix Vector Multiply which are of interest to us.

The idea is to build a balanced binary tree on the input data and sweep it to and from the root to compute the prefix sum. A binary tree with  $n$  leaves has  $d = \log_2 n$  levels, and each level  $d$  has  $2^d$  nodes. If we perform one add per node, then we will perform  $O(n)$  adds on a single traversal of the tree. The algorithm consists of two phases: the *reduce* phase (also known as the *up-sweep* phase) and the *down-sweep* phase. In the *reduce* phase, we traverse the tree from leaves to root computing partial sums at internal nodes of the

tree, as shown in Figure 19. This is also known as a parallel reduction, because after this phase, the root node (the last node in the array) holds the sum of all nodes in the array.

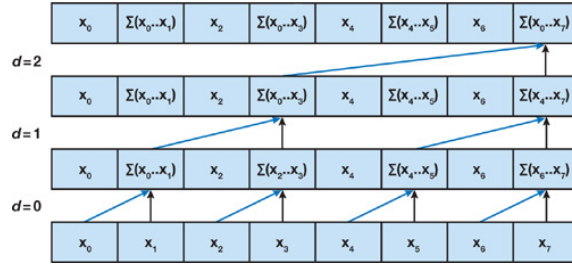


Figure 19: An Illustration of the Up-Sweep, or Reduce, Phase of a Work-Efficient Sum Scan Algorithm

In the *down-sweep* phase, we traverse back down the tree from the root, using the partial sums from the *reduce* phase to build the scan in place on the array. We start by inserting zero at the root of the tree, and on each step, each node at the current level passes its own value to its left child, and the sum of its value and the former value of its left child to its right child. The *down-sweep* is shown in Figure 20,

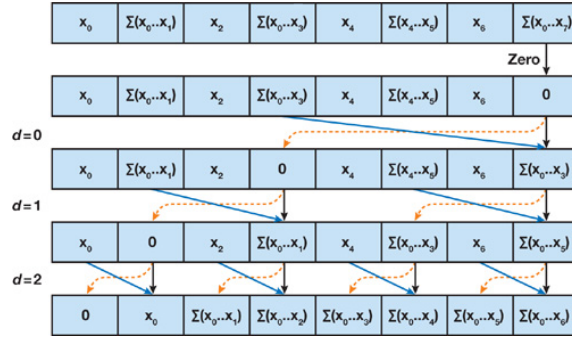


Figure 20: An Illustration of the Down-Sweep Phase of the Work-Efficient Parallel Sum Scan Algorithm

Details can be found out in the GPU Gems article available online [Harris, Sengupta, and Owens, 2007]. In the same document one can also find methods suggested by the authors to optimize the scan for the GPU by taking into account memory bank conflicts, shared memory and provisions for handling arbitrary array sizes (i.e. where  $n$  is not a power of 2).

NVIDIA released its SpMV library which uses a hybrid storage format for sparse matrices. It stores the matrices in a ELLpack-COO hybrid format, wherein the rows with more than a threshold number of non-zero elements are stored in the COO format. The rows with less than the threshold are stored in ELLPack format. Details can be found in [Bell and Garland, 2008]. In this extensive study they compare the performance of the kernels they have developed for the GPU *vis - a - vis* other architectures like STI Cell, and CPUs like Xeon, Opteron etc.

In a recent study by [Monakov and Avetisyan, 2009] they suggest a hybrid use of a blocked method and the ELL-COO format of the storing the sparse matrix. Then they perform vector multiply in order to extract more performance on both the fronts. Of course their method relies on an initial sweep on the matrix to find out the number of non-zero elements and the decision to divide the matrix into two different formats for storage. They suggest a dynamic programming approach to calculate optimal selection of blocks and also an heuristic approach based on greedy block selection.

The CUDA library can also be enriched with CUDAPP [Harris, Sengupta, Owens, Tseng, Zhang, and Davidson, 2009] which provides a routine *cusppSparseMatrix*, for sparse matrix vector multiply routine which comes in handy when solving through iterative methods. To use a method the user first declares a *Plan* in which he/she specifies the input output arrays, the number of elements etc.

[M. Baskaran and Bordawekar, 2008] demonstrate improvements over the methods discussed above [Bell and Garland, 2008] [Harris, Sengupta, Owens, Tseng, Zhang, and Davidson, 2009] by exploiting some of the architectural optimizations to the Sparse Matrix-Vector Multiplication code. In particular they center the optimization efforts on the following four points:

- Exploiting Synchronization-Free Parallelism,
- Optimized Thread Mapping,
- Aligned Global Memory Access;
- Data-Reuse.

### 6.3 Conjugate Gradient

[Georgescu and Okuda, 2007] discuss how conjugate gradient methods could be aligned to the GPU architecture. They also discuss the problems with precision and implementing preconditioners to accelerate convergence. In particular they state that for double precision calculations problems having condition numbers less than  $10^5$  may converge and give a speed-up also. They however warn that above a threshold value of the condition number the Conjugate Gradient Method will not converge. This last observation relates to the limited precision available on the GPU.

[Buatois, Caumon, and Levy, 2009] discuss their findings on implementing single precision iterative solvers on the GPU and show that for Jacobi preconditioning and a limited number of iterations the GPU is able to provide a solution of comparable accuracy but as the iterations increase the precision drops in comparison to the CPU.

They use the CG method exploiting some of the techniques like register blocking, vectorization and the Block Compressed Row storage (BCRS) to extract parallel performance on the GPU.

They alike other implementations try to maximize the throughput for the memory transactions by using  $4 \times 4$  blocks in the BCRS format. This format later proves beneficial for stripmining operations. Using such an arrangement complemented with the vector data types available on the GPU (for example *float4* that allows storing 4 32-bit floats to be stored at one index). By making arrays of such aggregate data types one can access data in chunks thereby saving address calculation. For e.g. in the case of a  $4 \times 4$  block storage an array of 4 *float4*'s can be used and accessing all elements is possible with a single address that of the array.

Further they suggest that individual elements within an aggregate data type could be allocated to multiple threads and such a pattern could be followed among the other elements of this 4-*float4* element array. Thereby providing speedups in reduction operations.

An important finding that is indicated in their results is that reordering of the type Cuthill-McKee did not show any influence on the implementations they executed for the GPU and the CPU.

### 6.4 Preconditioning

Techniques that are basically dependent on the Sparse Matrix Vector Multiply discussed in previous sections have been suggested in literature for accelerating Preconditioning

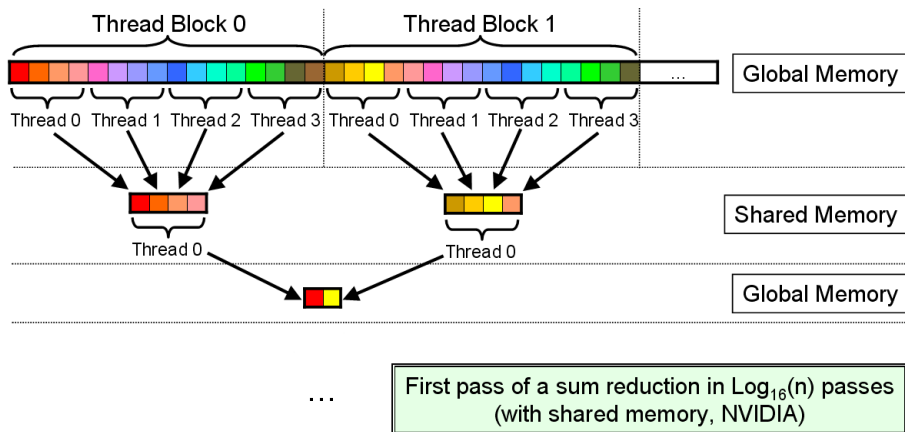


Figure 21: Sum reduction on a Scalar architecture with shared Memory

of Iterative Solvers like GMRES and Conjugate Gradient. [Wang, Klie, Parashar, and Sudan, 2009] use an LU Block Preconditioner, which has poor convergence qualities but is easier to parallelize, for solving a sparse linear system by the GMRES method. Coefficient

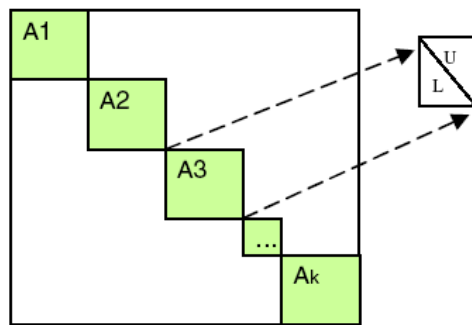


Figure 22: Block ILU preconditioner

matrix  $A$  is divided into equal sized sub-matrices which are then locally decomposed using ILU, as shown in Figure 22. The blocks shown in Figure 21 do not communicate to each other during the decomposition and also in solving it, this scheme fits well in the data parallel paradigm. A stream now is a collection of sub-matrices along the main diagonal.

In work published by [Asgari and Tate, 2009] they discuss how the use of a chebyshev polynomial based preconditioner could be utilised for achieving speedups in the Conjugate Gradient method for solving a linear system. The said preconditioner effectively reduces the condition number of the coefficient matrix thereby achieving convergence quickly. It approximates the inverse of the coefficient matrix with linear combinations of matrix-valued Chebyshev polynomials. This method uses only matrix multiplication and addition to compute the approximate inverse of the coefficient matrix, which makes it suitable for parallel platforms. In the implementation described in the paper, they use linear combinations of the first few chebyshev polynomials to build a preconditioner. The combination of Chebyshev preconditioner and Krylov subspace linear solver leads to a highly efficient solver on parallel platforms.

## 6.5 Multi-GPU Implementations

For getting better speed-ups in iterative methods for solutions of sparse systems [Cevahir, Nukada, and Matsuoka, 2009] [Ament, Knittel, Weiskopf, and Straßer, 2010] suggest the use of multiple GPUs mounted on a single main board. The CPU is responsible for synchronization and holds the arrays where GPUs read from and write to. In the first solution a Jagged Diagonal Format for storage of the matrix is employed along with use of constant cache and texture memory to store the vector used in Matrix-Vector Multiplication. Some additional padding is required in this method to make the data access coalesced and this is one of the main reasons cited for the demonstrated speed-up. The technique employed for solution is a inner-outer iteration based [Golub and Qiang, 1997] technique where the refinement or outer iteration runs on the CPU and the computation intensive inner iteration runs on the GPU.

In the second publication a new kind of preconditioner called the Incomplete Poisson Preconditioner is utilized to achieve speed-up across multiple GPUs. The idea stems from the inherently serial method of finding the approximation of the inverse of the coefficient matrix  $A$  which will then be used as a preconditioner. Since this method would pull back any possible speed-up on the GPU a new kind of preconditioner is suggested. Beginning with an SSOR like preconditioner of the type

$$M^{-1} = (I - LD^{-1})(I - D^{-1}L^T) \quad (236)$$

where  $L$  is the lower triangular part of  $A$  and  $D$  is the diagonal matrix containing diagonal elements of  $A$ . In turn it can be shown that

$$M = KK^T \quad (237)$$

where

$$K = I - LD^{-1} \quad \text{and} \quad K^T = I - D^{-1}L^T. \quad (238)$$

Considering a two-dimensional regular discretization and taking the case of an inner grid cell, the stencil of the  $i$ -th row of  $A$  is

$$\text{row}_i(A) = (a_{y-1}, a_{x-1}, a, a_{x+1}, a_{y+1}) \quad (239)$$

$$= (-1, -1, 4, -1, -1) \quad (240)$$

Hence, the stencils for  $L$ ,  $D^{-1}$ , and  $L^T$

$$\text{row}_i(L) = (-1, -1, 0, 0, 0) \quad (241)$$

$$\text{row}_i(D^{-1}) = (0, 0, 0.25, 0, 0) \quad (242)$$

$$\text{row}_i(L^T) = (0, 0, 0, -1, -1) \quad (243)$$

In the next step, after performing the operations for (238).

$$\text{row}_i(K) = (0.25, 0.25, 1, 0, 0) \quad (244)$$

$$\text{row}_i(K^T) = (0, 0, 1, 0.25, 0.25) \quad (245)$$

The final step is the matrix-matrix product  $KK^T$ , which is the multiplication of a lower and an upper triangular matrix. Each of the 3 coefficients in  $\text{row}_i(K)$  hits 3 coefficients in  $K^T$  but in different columns. The interleaved arrangement in such a row-column product introduces new non-zero coefficients in the result. The stencil of the inverse increases to

$$\text{row}_i(M^{-1}) = (m_{y-1}, m_{x+1, y-1}, m_{x-1}, m, m_{x+1}, m_{x-1, y+1}, m_{y+1}) \quad (246)$$

$$= (0.25, 0.0625, 0.25, 1.125, 0.25, 0.0625, 0.25) \quad (247)$$

Without going into too much detail here, the stencil enlarges to up to 13 non-zero elements in three dimensions for each row, which would almost double the computational effort in a matrix-vector product compared to the 7-point stencil in the original matrix. By looking again at the coefficients in  $row_i(M^{-1})$  it can be observed that the additional non-zero values are rather small compared to the rest of the coefficients. Furthermore, this nice property remains true in three dimensions, so they use an incomplete stencil assuming that these small coefficients only have a minor influence on the condition. They set them to zero and obtain the following 5-point stencil in two dimensions

$$row_i(M^{-1}) = (0.25, 0, 0.25, 1.125, 0.25, 0, 0.25) \quad (248)$$

Another important property of the incomplete formulation is the fact that symmetry is still preserved as the cancellation always affects two pair-wise symmetric coefficients namely

$$(m_{x+1,y-1}, m_{x-1,y+1}) \quad (249)$$

in two dimensions and

$$(m_{x+1,z-1}, m_{x-1,z+1})(m_{x+1,y-1}, m_{x-1,y+1})(m_{y+1,z-1}, m_{y-1,z+1}) \quad (250)$$

in three dimensions. From this it follows that the CG method still converges and hence the *IncompletePoisson* (IP) *preconditioner*. By calculating the product  $AM^{-1}$  and comparing the element-wise distance of this product with the Identity Matrix they further show that this preconditioner effectively reduces the condition number of the resulting matrix.

## 7 Research Questions

In this section we briefly outline what would be the main focus of the implementation phase of this work. Many of the techniques already explored for Iterative Methods will be used to find out their suitability for the GPU. At the same time we will contribute to the existing research for GPU based solutions by providing results on Deflation Method.

### 7.1 Conjugate Gradient on GPU vs CPU

It is possible to use FFT based solution of the Conjugate Gradient but it requires constant coefficient matrix which is not what we have in Bubbly Flow Problems. However some possibilities exist??? for which this method could also be explored as FFTs are very well optimized (CUFFT for CUDA) for the GPU. Using FISHpack for FFTs performance could be compared with CUFFT library on the GPU.

### 7.2 Preconditioning Approaches

#### 7.2.1 Preconditioned Conjugate Gradient on GPU vs CPU

In case of block preconditioners [Vuik and Frank, 2001] suggest that increasing the number of blocks leads to reduction in convergence. However an application of a Coarse-Grid Correction *a.k.a* Deflation using domain decomposition can result in acceleration of convergence. A question could be the choice of the number of blocks and the number of domains and how do they map to the processors available on the GPU.

#### 7.2.2 Diagonal

It is expected and also discussed in previous work that this kind of preconditioners are easy to parallelize, statistics will be reported for our method also.

### 7.2.3 IC(0)

Incomplete Cholesky based preconditioners are the first choice when looking for an effective preconditioner. However after evaluating the effectiveness of such a method a possible direction could be to test out the variants discussed earlier in section 3.4.

### 7.2.4 MultiGrid

[Feng and Li, 2008] utilise a geometric multi-grid method and show impressive speed-up by dividing the workload amongst the GPU and the CPU. [Göddecke, Strzodka, Mohd-Yusof, McCormick, Wobker, Becker, and Turek, 2008] bring out the requirements for code modification when considering GPUs or other multi-core platforms for acceleration of Multi-Grid Solvers.

### 7.2.5 IP Preconditioning

The Incomplete Posisson Preconditioner can also be tried along with Deflation to see if the convergence benefits from both of them and if yes, then how much.

## 7.3 Deflation Approaches

### 7.3.1 Deflated PCG on GPU vs CPU

SubDomain Deflation applied to a preconditioned Conjugate Gradient would be one of the things this work would be contributing uniquely. Also the application of Bubbly flows will be evaluated for this particular method of obtaining a solution.

### 7.3.2 DPCG on Many GPUs

Taking the cue from earlier works discussed in the previous section it would be beneficial to predict/test out the implementation of the solution we propose on a cluster of GPUs.

## 7.4 Precision Statistics

First we will experiment with the Single Precision floating point available on the GPU to see how much it affects convergence. A next step could be to see how Conjugate Gradient with deflation performs when mixed precision strategies are used. The performance will be reported both on the rate of convergence and with speed-ups over CPU and single-precision GPU code.

# 8 Preliminary Results

## 8.1 Conjugate Gradient - Basic Version

As a first step we have implemented CG on the GPU. We have tried with three different storage formats namely the CSR, DIA and Matrix-Free Format.

In this version we have largely relied on cublas and texture cache based(for CSR and DIA) optimizations. The CPU versions are in double precision and GPU versions in single precision arithmetic.

We get a speed-up of around 6 times but we expect to get even higher speedups by use of chare memory and improvements in memory access patterns.

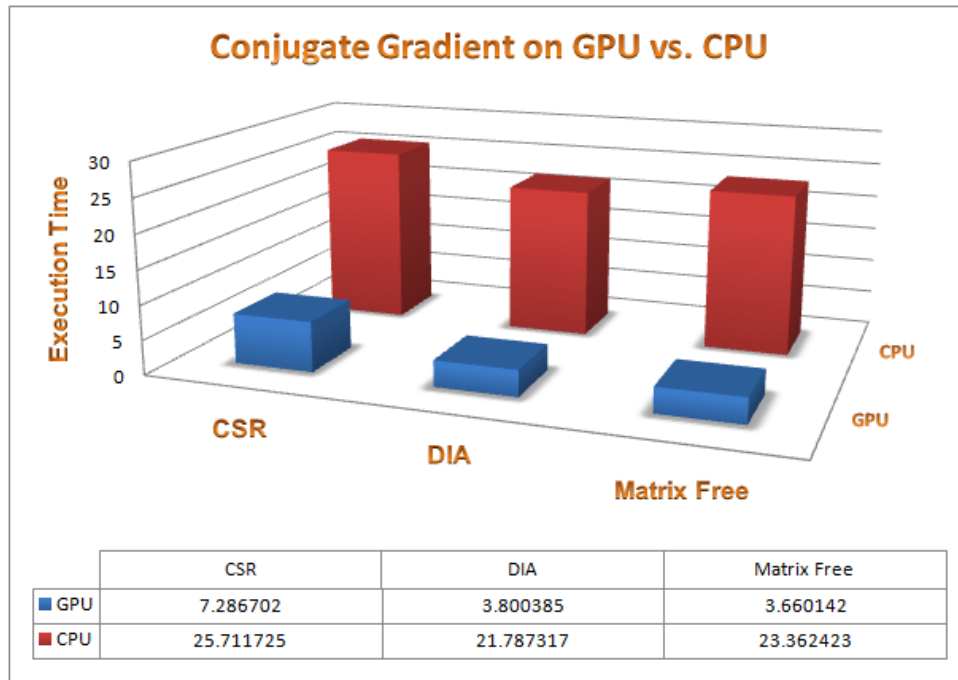


Figure 23: Conjugate Gradient on GPU vs CPU

## References

- M. Ament, G. Knittel, D. Weiskopf, and W. Straßer. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform. <http://www.vis.uni-stuttgart.de/~amentmo/docs/ament-pcgip-PDP-2010.pdf>, 2010.
- A. Asgasri and J. E. Tate. Implementing the chebyshev polynomial preconditioner for the iterative solution of linear systems on massively parallel graphics processors. <http://www.ele.utoronto.ca/~zeb/publications/>, 2009.
- M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *CoRR*, abs/0808.2794, 2008. URL <http://dblp.uni-trier.de/db/journals/corr/corr0808.html>. informal publication.
- N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-04, NVIDIA Corporation, December 2008.
- M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse-based ilus. *SIAM J. Sci. Comput.*, 27(5):1627–1650, 2006.
- J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 917–924, New York, NY, USA, 2003. ACM.
- L. Buatois, G. Caumon, and B. Levy. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3): 205–223, 2009.
- A. Cevahir, A. Nukada, and S. Matsuoka. Fast conjugate gradients with multiple GPUs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 893–903, Berlin, 2009. Springer-Verlag.

- E. de Sturler and D. Loher. Parallel iterative solvers for irregular sparse matrices in high performance fortran. *Future Gener. Comput. Syst.*, 13(4-5):315–325, 1998.
- Z. Feng and P. Li. Multigrid on GPU: Tackling power grid analysis on parallel SIMT platforms. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 647–654, Nov. 2008.
- S. Georgescu and H. Okuda. Conjugate gradients on graphic hardware. Under review in *Lecture Notes in Computer Science*, 2007.
- D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering (IJCSE)*, 4(1):36–55, 2008.
- G. H. Golub and Y. Qiang. Inexact preconditioned conjugate gradient method with inner-outer iteration. *SIAM J. Sci. Comput.*, 21:1305–1320, 1997.
- M. Harris, S. Sengupta, and J. D. Owens. *Parallel Prefix Sum (Scan) with CUDA*, 2007.
- M. Harris, S. Sengupta, J. D. Owens, S. Tseng, Y. Zhang, and A. Davidson. Cudpp. <http://gpgpu.org/developer/cudpp>, 2009.
- J. Kruger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM.
- M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical report, IBM Research Division, NY, USA, December 2008. <http://gpgpu.org/2009/04/13/optimizing-sparse-matrix-vector-multiplication-on-gpus>.
- A. Monakov and A. Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In *SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 289–297, Berlin, 2009. Springer-Verlag.
- Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Massachusetts, 1996.
- Y. Saad, M. Yeung, J. Erhel, and F. Guyomarc’h. A deflated version of the conjugate gradient algorithm. *SIAM J. Sci. Comput.*, 21(5):1909–1926, 2000.
- S. SenGupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for GPU computing. *Graphics Hardware*, 2007.
- J. Shi, Y. Cai, W. Hou, L. Ma, S. X.-D. Tan, P.-H. Ho, and X. Wang. GPU friendly fast poisson solver for structured power grid network analysis. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 178–183, New York, NY, USA, 2009. ACM.
- Y.M. Tang. *Two-Level Preconditioned Conjugate Gradient Methods with Applications to Bubbly Flow Problems*. PhD thesis, Delft Unniveristy Of Technology, 2008.
- C. Vuik and J. Frank. Coarse grid acceleration of a parallel block preconditioner. *Future Generation Computer Systems*, 17:933–940, 2001.
- C. Vuik, R.R.P. van Nooyen, and P. Wesseling. Parallelism in ILU-preconditioned GMRES. *Parallel Computing*, 24:1927–1946, 1998.

M. Wang, H. Klie, M. Parashar, and H. Sudan. Solving sparse linear systems on NVIDIA Tesla GPUs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 864–873, Berlin, 2009. Springer-Verlag.