A LIGHT-WEIGHT SHARING MECHANISM FOR SPARSE MATRIX COMPUTATIONS

Theodore Kubaska
A.B.-Sc.B (physics), Brown University, 1969
M.S. (physics), University of Maine, 1974

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

April 1995

The thesis, "A Light-Weight Sharing Mechanism for Sparse Matrix Computations" by Theodore Kubaska has been examined and approved by the following Examination Committee:

Steve. W. Otto
Assistant Professor
Thesis Advisor

Michael Wolfe
Associate Professor

J. E. Brandenburg
Scalable Systems Division
Intel Corporation

Abstract

A Light-Weight Sharing Mechanism for Sparse Matrix Computations

Theodore Kubaska, Master's Degree

Supervising Professor: Steve Otto

Although massively parallel MIMD machines now make it practical to solve large numerical problems, using these machines necessitates developing efficient methods for compute nodes to communicate with each other. This thesis describes a weakly coherent shared memory programming model for message-passing architectures. This method is applicable to iterative algorithms such as those used in finite element calculations. In this method, data on the boundaries of a domain decomposition are stored as aliases on the corresponding compute nodes. The calculation proceeds independently on each compute node. The method is termed weakly coherent because at certain points in the algorithm, data coherency is enforced with user-initiated calls. These aliased data points take part in independent calculations; they are both read and written. In addition, the construction of alias lists allows the pre-computing of the communication pattern and hence improves communication efficiency by coalescing messages.

Acknowledgments

I would like to thank Dr. Steve W. Otto for his encouragement, patience, and inspiration. He has kept me on track when I wandered into side issues and directed me to the pertinent literature. Thanks also to the members of my thesis committee, Dr. Michael Wolfe and Dr. J. E. Brandenburg. Extra special thanks to Lory Kubaska, who convinced me several times that it was worth it.

I would also like to thank Intel Corporation for the use of a number of Paragon XP/S supercomputers.

Table of Contents

Chapter 1 Introduction 1

Introduction 1
- Solving Partial Differential Equations 2
- The Finite Difference Method 2
- The Finite Element Method 2

Some Background Mathematics and Definitions 4
- Structured and Unstructured Grids 4
- Directed Graphs 5
- Laplace's Equation 5
- Finite Difference Grid and the Corresponding Matrix Equation 6
- Finite Element Grid and the Corresponding Matrix Equation 9

Parallel Computers 10
- Types of Parallel Computers 11
- Topology 12
- How the Mesh Appears in Real Life 15
- Speedup and Amdahl's Law 15
  - Amdahl's Law 16
  - Scaled Speedup 17
  - Fixed-Grain Speedup 17
- Parallel Programming 17

The Aliasing Method 18

Organization of this Thesis 22

Chapter 2 Solution Methods 24

Introduction 24

Serial Direct Methods 24

- Pivoting 25
- Ordering Schemes 26
- Fill-in 26
- Minimizing Fill-in 26
- Confining Fill-in 27
- Frontal Method 28

Serial Iterative Methods 30

- Stationary and Non-Stationary Methods 30
- Stationary Methods 31
  - Serial Jacobi Method 31
  - Serial Gauss-Seidel Method 32
  - Serial Successive Overrelaxation Method 33
- Non-Stationary Methods 34
  - Serial Conjugate Gradient Method 34
  - Preconditioned Conjugate Gradient 36
    - Condition Number and Convergence 36
    - Preconditioning 36

Chapter 3 Parallelization Techniques 38

Introduction 38

Data Parallelism 38

Control Parallelism 39

Parallel Direct Methods 39

- Serial Reordering Methods Concentrate on Reducing Fill-In 39
- The Tridiagonal Matrix: No Fill-In, but Requires Serial Elimination 39
- The Elimination Tree 40

Simple Example of an Elimination Tree 40
Definition of an Elimination Tree 41
Applying the Definition of an Elimination Tree 43

The Multifrontal Method 44
Nested Dissection 45
Parallel Multifrontal Method 47

Parallel Iterative Methods 48
Parallel Jacobi Method 49
Parallel Gauss-Seidel Method 51
Example of a Parallel Laplace Solver 51
The Parallel Jacobi Method and Laplace's Equation 52
The Parallel Gauss-Seidel Method and Laplace's Equation 54
Parallel Conjugate Gradient Method 55
Matrix-Vector Multiplication 57
Dot Product 59
The Owner Computes Method 59

Comparison of Direct and Iterative Methods 60

Chapter 4
Implementation of Aliasing 62

Introduction 62

Grid Partitioning 62

Information Needed to Construct Alias Groups 64

The Alias List 66

Making a Distributed Vector Coherent 66
The Send and Receive Lists 67
Example of Making a Vector Coherent 69

Chapter 5 Implementation of the Conjugate Gradient Method 73

Introduction 73

The Coefficient Matrix 73

Assembling the Matrix 73

Assembled vs. Unassembled Matrices 74

Conjugate Gradient Operations 76

Dot Product 76

Matrix-Vector Multiplication 79

Conjugate Gradient Algorithm 81

Setup Code 81

Conjugate Gradient Code 82

Testing for Convergence 83

Chapter 6 A Random Matrix 85

Introduction 85

The NAS CG Benchmark 86

Partitioning the Matrix 86

Determining the Location of Field Nodes 87

Example Partitioned over Six Compute Nodes 90

Aliasing Method is Inefficient for the NAS Matrix 90

What Happens During Matrix-Vector Multiplication 90

Longer Messages Than Needed 91

Performs Unneeded Additions 91

The NAS Matrix as a Limiting Case 92

Modifications Made for the NAS CG BenchMark 93

Balanced Secretaries 94

No Zeros 95

Chapter 7 Performance 98

Introduction 98

Laplace's Equation on a Rectangular Grid 98

Laplace: How the Data Were Obtained 99

Laplace: Results for the (x,y) Problem 100

Laplace: Message Passing in the (x,y) Problem 102

Laplace: Node Dependence in Vector Coherence 104

The NAS CG Benchmark 105

NAS CG: How the Data Were Obtained 105

NAS CG: What the Benchmark Measures 105

NAS CG: The Small Version 106

The Real Version 107

NAS CG: Message Passing 108

Chapter 8 Conclusions 111

Introduction 111

Weakly-Coherent Shared Memory 111

Efficient for Structured Problems 111

Local Indexing 112

No Distinguished Owner 112

Future Work 112

Appendix A The Finite Element Method 114

Introduction 114

What is a Finite Element? 114

Elemental Basis Functions 115

Minimum Residual Method 117

The Weak Formulation 118

The Galerkin Method 120

An Example of Solving Poisson's Equation 123

Elemental Matrix 124

The Right-Hand Side 125

An Example of Constructing Matrix Elements 127

Another Way of Accounting for Type 1 Boundary Conditions 130

The Solution 132

Appendix B Exact Solution of Laplace's Equation 133

Introduction 133

Statement of the Problem 133

Solving the Equation 133

Finite Element Method 137

Appendix C The Conjugate Gradient Method 139

Appendix D The NAS Benchmark 141

Introduction 141

Assumptions 141

Power Method 141

Output 142

The Matrix A 143

Row Decomposition of A 145

References 146

Biography 150

Index 151

List of Figures

Figure 1.1. Structured and Unstructured Grids 5
Figure 1.2. Example of a Directed Graph and Its Corresponding Matrix 6
Figure 1.3. Two-Dimensional Grid for Laplace's Equation 7
Figure 1.4. 6x6 Square Grid for Finite Differences 8
Figure 1.5. Quadrilateral Finite Element and Corresponding Matrix 10
Figure 1.6. Examples of three- and Four-Dimensional Hypercubes 14
Figure 1.7. Paragon XP/S Mesh Network 16
Figure 1.8. Dividing the Mesh Among Compute Nodes 19
Figure 1.9. Alias Groups 20
Figure 1.10. Pull-ins 21
Figure 1.11. Pull-ins Distributed among Alias Group Members 21
Figure 2.1. Block Tridiagonal Matrix 27
Figure 2.2. Demonstration of the Frontal Method 29
Figure 2.3. 4x4 Square Grid Used to Illustrate the Jacobi and Gauss-Seidel Methods 33
Figure 3.1. Tridiagonal Matrix Exhibiting no Fill-in 40
Figure 3.2. Elimination Trees 41
Figure 3.3. How Reordering Affects the Elimination Tree 42
Figure 3.4. Nested Dissection of a Linear Chain 45
Figure 3.5. Nested Dissection of a Two-Dimensional Directed Graph 46
Figure 3.6. Further Nested Dissection of a Two-Dimensional Directed Graph 47
Figure 3.7. Row Decomposition for Jacobi Iteration 50
Figure 3.8. Grid Decomposition 53
Figure 3.9. Grid Used for Parallel Gauss-Seidel Implementation. 54
Figure 3.10. Checkerboard Ordering 55
Figure 3.11. Finite Element Mesh Partitioned Among Compute Nodes 56
Figure 3.12. Pull-ins Distributed among Alias Group Members 58
Figure 3.13. Dividing the Mesh Among Compute Nodes (Owner Computes) 60
Figure 4.1. Distributed Grid Showing Alias Groups 63

Figure 4.2. Information Needed for Alias Group Construction 65
Figure 4.3. Example of an Alias List 67
Figure 4.4. Example of Send and Receive Lists 68
Figure 4.5. Global Description of Coherence 69
Figure 4.6. Local Description of Combination (Part A) 71
Figure 4.7. Local Description of Combination (Part B) 72
Figure 5.1. Construction of Assembled Matrix Elements 75
Figure 5.2. Unassembled vs. Assembled Matrix-Vector Multiplication 76
Figure 5.3. Formation of a Dot Product of Distributed Vectors 78
Figure 5.4. Matrix-Vector Multiplication with Coherence 80
Figure 6.1. Formation of Alias Groups when Matrix is Partitioned by Rows 89
Figure 6.2. NAS Matrix-Vector Multiplication 92
Figure 6.3. Updating Aliases when the Matrix is Distributed by Rows 93
Figure 6.4. Method 1: Row Decomposition 94
Figure 6.5. Modified vcomb() Routine. 96
Figure 6.6. vcomb() Does not Necessarily Involve Broadcasts 97
Figure 7.1. Total Flops vs. the Number of Compute Nodes in the Partition 100
Figure 7.2. Total Flops/Compute Node vs. the Number of Compute Nodes in the Partition 101
Figure 7.3. Time/Iteration vs. the Number of Compute Nodes in the Partition 101
Figure 7.4. Messages in First Stage of Making a Vector Coherent 103
Figure 7.5. Balanced Secretaries/No Zeros and Low Node Number Secretaries 106
Figure 7.6. Matrix-Vector Time vs. Compute Nodes (small version) 107
Figure 7.7. vcomb() time vs. Compute Nodes (small version) 107
Figure 7.8. Total Flops vs. the Number of Compute Nodes in the Partition 108
Figure 7.9. Total Flops/Compute Node vs. the Number of Compute Nodes in the Partition 108
Figure 7.10. Time/Iteration vs. the Number of Compute Nodes in the Partition 109
Figure A.1. The Solution Surface over One of the Triangular Elements 115
Figure A.2. The Elemental Basis Functions, $N_i(x,y)$, $N_j(x,y)$, and $N_k(x,y)$ 117
Figure A.3. The Nodal Basis Function for a Node 121
Figure A.4. The Nodal Basis Function for a Node 123
Figure A.5. Finite Element Nodes 1 through 7 with Elements I through VI 125
Figure A.6. Grid Used to Calculate Example Matrix Elements 127
Figure B.1. Offsetting the Boundary Conditions for Easier Application 134
Figure B.2. Solution of Laplace's Equation (Infinite Series) 136
Figure B.3. The Surface Divided into Finite Elements 137
Figure B.4. The Surface Divided Among Four Compute Nodes. 138

Figure B.5. Solution of Laplace's Equation (Finite Element) 138
Figure D.1. Sparse Storage Scheme for Matrix A 144

List of Tables

Table 7.1. Data for the (x,y) Grid Problem 102

Table 7.2. Data for the NAS CG Benchmark 110

Chapter 1 Introduction

Introduction

This thesis investigates how to solve large, sparse, linear systems of equations on massively-parallel computers. Such a linear system can be written as the matrix equation $\boldsymbol{Ax=b}$ where $\boldsymbol{A}$ is the sparse matrix, $\boldsymbol{x}$ is the unknown vector, and $\boldsymbol{b}$ is a known vector. $\boldsymbol{A}$ is sparse when most of its elements are zero.

The two matrices looked at in detail are the Laplace matrix (see Appendix A"The Finite Element Method" for a derivation of this matrix) and the NAS matrix (see Appendix D"The NAS Benchmark"). The Laplace matrix has about 6% nonzero elements for the size studied. The NAS matrix has about 1% nonzero elements.

A distributed-memory multicomputer is a parallel computer whose compute nodes have their own physical memory. Each compute node has its own address space. When compute nodes need to share information, they must exchange messages with each other. For a general description of parallel computer architecture, refer to "Parallel Computers" on page 10.

An effective way of achieving parallelism on a distributed-memory multicomputer is to partition the data among the compute nodes. Then, each compute node operates on its portion of the data set, communicating with other compute nodes as needed.

Partitioning the data among the compute nodes means that each compute node contains some matrix elements and some vector components. Some of the vector components exist on more than one compute node. Because the memory is distributed, a vector component that exists on more than one compute node actually exists as a separate copy on compute nodes that share the value.

The computation then proceeds in parallel on each compute node with each compute node using its own data set. During the course of the computation some of the shared values may attain different values. When the algorithm requires that the shared vector elements have the same values, the compute nodes must exchange information.

This type of operation (computation followed by communication when required by the algorithm) is termed *loosely synchronous*.

Solving Partial Differential Equations

In particular, this thesis investigates the solution of large, sparse, linear systems of equations that arise when using finite difference or finite element methods to solve boundary value partial differential equations. These methods are described later in this chapter.

The first step in solving a partial differential equation on a computer is to discretize the equation. Partial differential equations are discretized by setting up a grid of points and defining the solution to have a value at each point [30], [37]. Each point in the grid is given a numeric label, and the solution is a vector whose i^{th} component is the value at the grid point labeled i.

The matrix elements themselves designate the interaction between grid points. In particular, the matrix element a_{ij} gives the weight of the coupling between the grid points labeled i and j. The coupling described by a matrix element is the interaction among points that is also described by the derivative. The matrices are sparse because, at the level of approximation required for numerical work, the differential operator and consequently its corresponding matrix operator describe local interactions.

The Finite Difference Method

The finite difference method [33] consists of expressing the derivative at each point in the grid by a difference quotient. This approximation is obtained by writing a Taylor expansion for the derivative and retaining terms up to the chosen order in the grid spacing [25].

The finite difference approximation at each grid point is then substituted into the partial differential equation. The result is a set of simultaneous linear equations, one equation for each grid point. The equations are then solved for the solution vector; there is a component of the solution vector at each grid point.

The Finite Element Method

The finite element method consists of approximating the solution as a sum of carefully chosen functions called *basis functions*, substituting that sum into the partial

differential equation, and manipulating the result to obtain a set of algebraic equations, the solution of which provides the parameters of the approximate solution [35].

Specifically, for a two-dimensional problem, one writes the solution $u(x,y)$ as

$$u(x, y) = \Omega(x, y) + \sum_{i=1}^{N} a_i \phi_i(x, y)$$

where the ϕ_i(x,y) are the basis functions. These basis functions are defined to be zero on the boundary. Boundary conditions specifying $u(x,y)$ are satisfied by $\Omega(x,y)$. The solution $u(x,y)$ is said to be expanded in terms of the basis functions. The set of linearly independent, simultaneous, algebraic equations are solved for the a_i, the coefficients in this expansion.

There are N finite element nodes in the grid, and a basis function is associated with each finite element node. The set of algebraic equations contains as many equations as there are basis functions. Because of the way the basis functions are chosen, it turns out that the coefficient of a basis function in the expansion is also the value of the solution at the finite element node associated with that basis function. When the calculation is complete, one has values at all the finite element nodes, but one can also obtain values at non-nodal positions by using these values in the expansion.

The finite elements are the areas over which the basis functions are nonzero. In two dimensions these finite elements are usually polygons because polygons fill up space easily and because it's easier to make the basis function go to zero on a linear side. The finite elements are also often triangular because, in two dimensions, triangular elements are best at approximating irregular boundary geometry [6]. In three dimensions, the finite elements might be tetrahedra. The vertices of the polygons are the finite element nodes. If a finite element node is a vertex in one polygon, it is also a vertex in all adjacent polygons. Figure 1.1 on page 5 shows two grids that can also be interpreted as triangular finite element meshes.

The basis functions are chosen to have the following properties:

- There is one basis function associated with each finite element node.
- A basis function is nonzero only over the finite elements to which its node belongs.
- The basis functions are chosen to have simple mathematical properties, such as easy derivatives. They are often linear.

- The value of the basis function at the node with which it is associated is 1. This ensures that the coefficient of the basis function in the assignment is the solution at that finite element node.

The finite element method [5], [6], [23], [38] began in the field of structural engineering. With this method, a structure was first divided into individual components or elements, whose behavior was readily understood. Then, the behavior of the complete system was approximated by the assembled behavior of the components. In the 1940s, analogies between portions of a continuous solid and discrete elements such as bars and beams were made, and the technique became useful in aircraft structural design [38].

Refer to Appendix A "The Finite Element Method" for a detailed discussion of the finite element method.

Some Background Mathematics and Definitions

This section provides more information about grids and distinguishes between structured and unstructured grids. The section also introduces directed graphs and shows how they are related to the grids. Finally, the section defines Laplace's equation and shows how a matrix equation results from both finite difference and finite element solutions.

Structured and Unstructured Grids

With finite difference methods, the grid is *structured*; that is, the grid points are arranged in a regular, predictable pattern. With finite element methods, the grid is often *unstructured*; that is, different portions of the grid may have their points arranged differently to take into account the specific needs of the problem. Figure 1.1 illustrates the difference between structured and unstructured grids. The figure shows two grids consisting of triangular elements.

In a structured grid, the grid points form a regular pattern. To construct a structured grid, it is only necessary to know the pattern and then to repeat that pattern throughout the domain.

In an unstructured grid, there is no pattern. The exigencies of the problem end up determining the size, shape, and number of the triangles. For example, if the domain has a boundary with twists and turns, a large number of triangles may be necessary for adequate coverage. In addition, a particular region may be of more interest than another, and a denser set of triangles in that region provides for a more detailed solution. The

unstructured grid in the figure is adapted from a figure in the text by MacKeown and Newman [25].

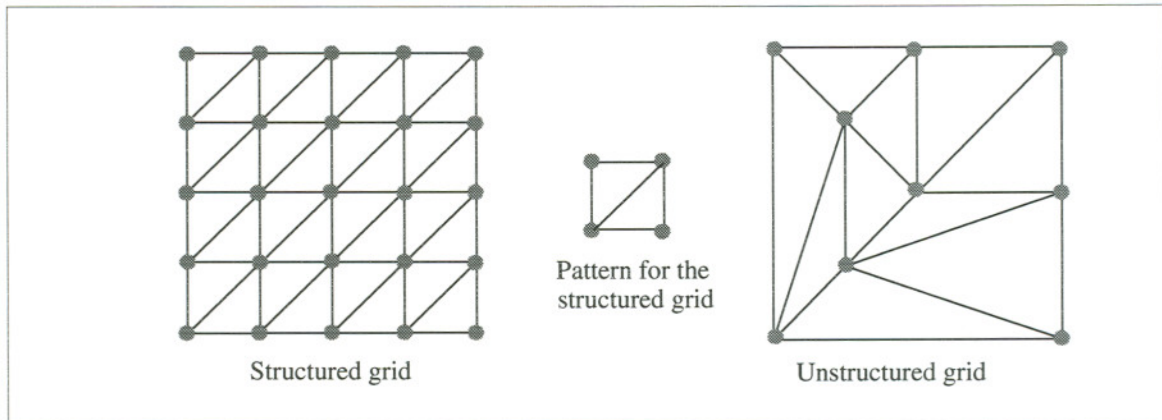


Figure 1.1. Structured and Unstructured Grids

The ordering of the grid is how the grid points are numbered; that is, which point on the grid is identified by 1, which point is identified by 2, etc. The ordering determines the pattern of nonzero elements in the corresponding matrix; that is, if grid points 1 and 2 interact, then matrix elements a_{12} and $a_{21}$are nonzero. The performance of some solution methods is dependent on where in a sparse matrix the nonzero elements occur. Hence, for these methods, reordering the grid is an important step. For more information about reordering, refer to "Ordering Schemes" on page 26.

Directed Graphs

This section defines the directed graph of a matrix [8]. A directed graph is a set of nodes (points) and edges (directed lines connecting the points). A directed graph node is a diagonal element of the matrix. There is one node for each diagonal element, even if the diagonal element is zero. Specifically, node i designates the diagonal element a_{ii}. An edge in the directed graph designates a nonzero, off-diagonal matrix element. Specifically, the line connecting nodes i and j, directed toward j, designates the matrix element a_{ij}, and the same line, directed toward i, designates the matrix element a_{ji}.

For example, Figure 1.2 shows a simple four-node directed graph and its corresponding matrix. Because the arrows are bidirectional, both a_{ij} and a_{ji} are nonzero.

Laplace's Equation

Laplace's equation with Dirichlet boundary conditions is chosen as a specific

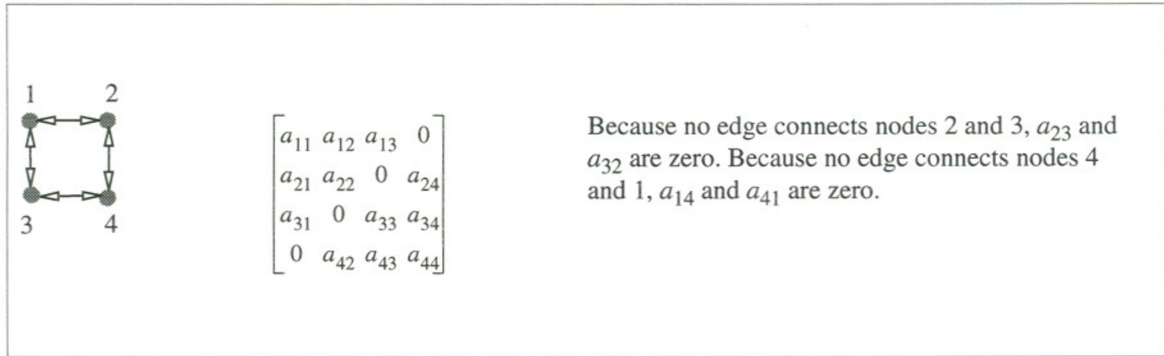


Figure 1.2. Example of a Directed Graph and Its Corresponding Matrix

example of a linear, second order partial differential equation. This is the equation that will be solved with the methods developed in this thesis. Laplace's equation in two dimensions and (*x*,*y*) coordinates is

$$\frac{\partial^2}{\partial x^2}u(x, y) + \frac{\partial^2}{\partial y^2}u(x, y) = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)u(x, y) = \nabla^2 u(x, y) = 0$$

The differential operator consisting of the sum of the second order partial derivatives is called the Laplacian and written as ∇^2. Dirichlet boundary conditions mean that the value of the function (in this case $u(x,y)$) is constrained to be a prescribed constant on the domain boundary.

Finite Difference Grid and the Corresponding Matrix Equation

This section shows how a matrix equation $\boldsymbol{Ax=b}$ results when a square grid is used in a second order, finite difference solution to Laplace's equation with Dirichlet boundary conditions. The section explicitly shows the matrix elements and the construction of the vector $\boldsymbol{b}$.

To see how a matrix equation results from this partial differential equation, define a domain as a two-dimensional grid and assign a value of u to each point in the grid. The grid spacing is the same in the x and y directions and is equal to Δ. This grid is shown in Figure 1.3.

The second order finite difference approximations for the second derivative (which can be obtained from a Taylor expansion) are

$$\frac{\partial^2}{\partial x^2}u = \left(\frac{1}{\Delta^2}\right) \cdot \left(u(x+\Delta, y) - 2u(x, y) + u(x-\Delta, y)\right)$$

$$\frac{\partial^2}{\partial y^2}u = \left(\frac{1}{\Delta^2}\right) \cdot \left(u(x, y+\Delta) - 2u(x, y) + u(x, y-\Delta)\right)$$

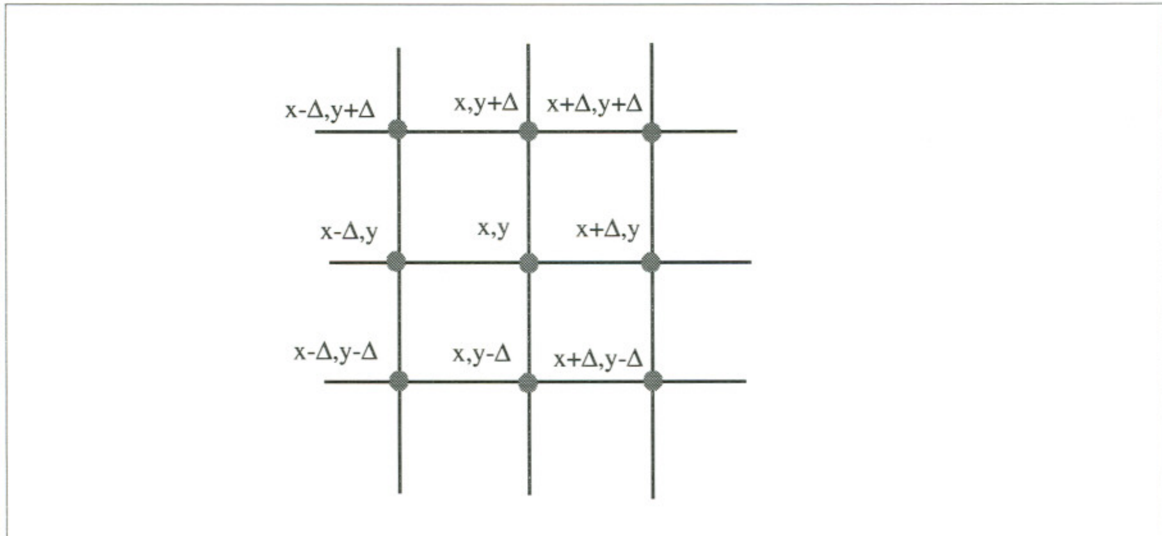


Figure 1.3. Two-Dimensional Grid for Laplace's Equation

Substitute these finite difference approximations for the second derivative into Laplace's equation and multiply through by Δ^2. The result is

$$-u(x, y+\Delta) - u(x-\Delta, y) + 4u(x, y) - u(x+\Delta, y) - u(x, y-\Delta) = 0$$

One such equation exists for every grid point (x,y). With this approximation, each point in the grid turns out to interact only with its nearest neighbors.

If the resulting set of linear equations is written as a matrix equation, the coefficients of each linear equation make up a row of the matrix. An interior node has five nonzero matrix elements in its row. These are the self-term (the diagonal element which is equal to 4) and the four nearest neighbors in the up, down, left and right directions (the off-diagonal terms which are equal to -1). The solution is a vector with an element defined at each grid point.

Figure 1.4 shows a 6x6 grid. The lines in the grid are shown as directed because for the finite difference Laplace example, the grid and the directed graph are the same. Hence, for this example the terms directed graph node and grid point are interchangeable. The boundary points are shown as darkly shaded circles, and the unknown points are shown as lightly shaded circles..

If the solution vector $\boldsymbol{x}$ is thought of as a 36-element vector (one element for each grid point), the matrix is 36x36 and the matrix equation to be solved is $\boldsymbol{Ax}=0$. There are 16 unknown elements in this vector, and they are labeled 1 through 16. The boundary conditions specify 20 vector elements of this vector, and they are labeled 17 through 36.

Another approach (the one taken in the figure) is to consider $\boldsymbol{x}$ as a 16-element

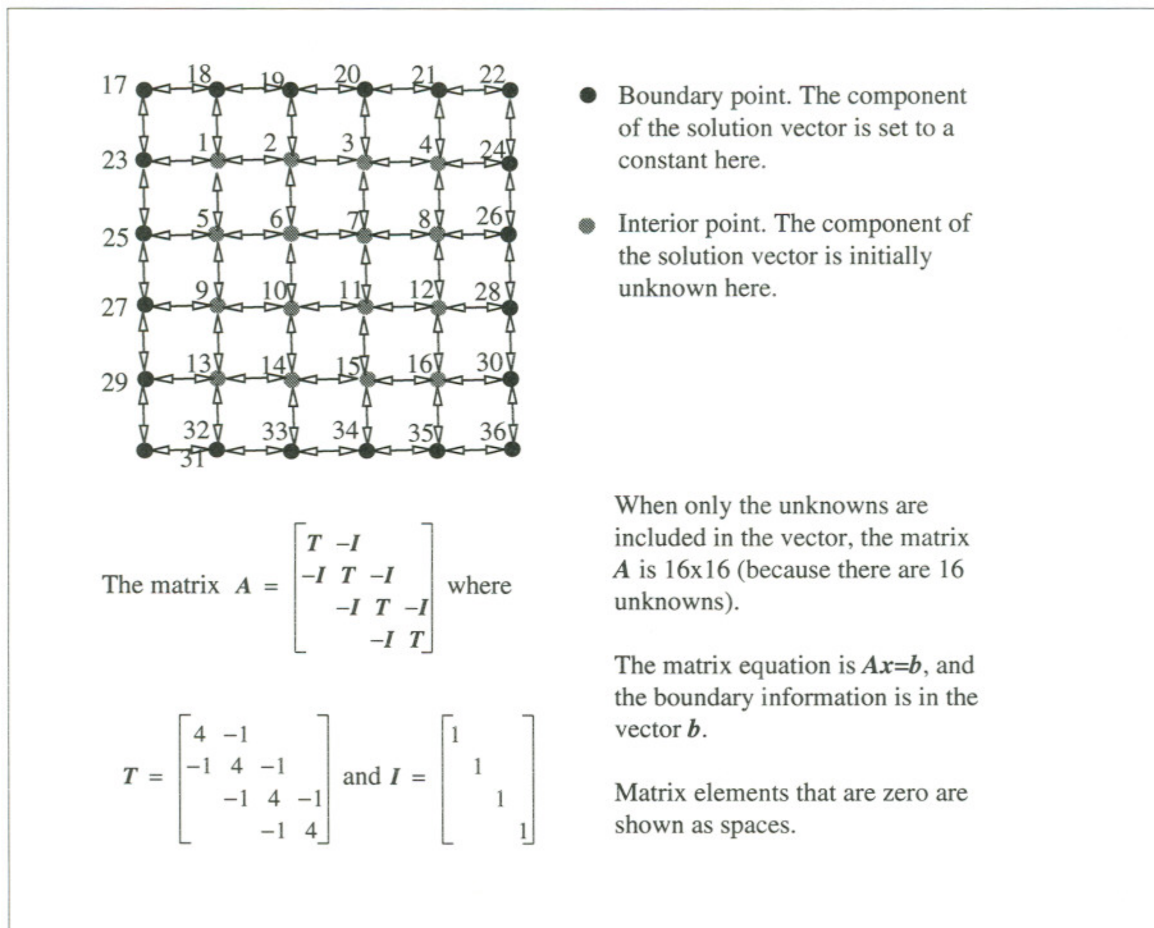


Figure 1.4. 6x6 Square Grid for Finite Differences

vector (one element for each point in the grid with an unknown value) and move the known values of $\boldsymbol{x}$ to the right-hand side. To see how this is done, consider the first six equations in the set of simultaneous linear equations shown as $\boldsymbol{Ax=b}$.

$$4x_1 - x_2 - x_5 = x_{23} + x_{18}$$

$$4x_2 - x_1 - x_3 - x_6 = x_{19}$$

$$4x_3 - x_3 - x_8 = x_{21} + x_{24}$$

$$4x_4 - x_2 - x_5 = x_{23} + x_{18}$$

$$4x_5 - x_1 - x_6 - x_9 = x_{25}$$

$$4x_6 - x_2 - x_5 - x_7 - x_{10} = 0$$

$$\vdots$$

Here, x_6 is the first element that is not directly connected to the boundary, and its corresponding equation (the sixth equation) is consequently the first equation with zero on the right-hand side. The vector $\boldsymbol{b}$ is ($x_{23}+x_{18}$, x_{19}, $x_{21}+x_{24}$, $x_{23}+x_{18}$, x_{25}, 0, ...).

The matrix $\boldsymbol{A}$ is then 16x16, and the matrix equation is $\boldsymbol{Ax=b}$, with $\boldsymbol{b}$ containing the known elements of $\boldsymbol{x}$ that were moved to the right. Figure 1.4 shows $\boldsymbol{A}$ as a 16x16 matrix, but rather than print out a 256-element matrix where most of the elements are zero, the matrix is shown in block form. Zero elements are shown as spaces because it makes the matrix more readable.

There is one equation for each unknown element of $\boldsymbol{x}$. If there were less equations than elements of $\boldsymbol{x}$, $\boldsymbol{x}$ would not have a unique solution, and the problem would be underdetermined. If there were more equations than elements of $\boldsymbol{x}$, the set of equations would not be linearly independent (the extra equations could be written as linear combinations of the others).

Finite Element Grid and the Corresponding Matrix Equation

The square grid shown in Figure 1.4 could also be considered as a set of quadrilateral finite elements. For a definition of a finite element, refer to "The Finite Element Method" on page 2. Each quadrilateral finite element provides contributions to 16 matrix elements as shown in Figure 1.5.

Figure 1.5 shows two quadrilateral finite elements, one made up of nodes 2, 3, 6, and 7, and the other made up of nodes 6, 7, 10, and 11. Each finite element provides one 4x4 elemental matrix and contributes to up to 16 matrix elements. The assembled matrix $\boldsymbol{A}$ is the sum of elemental matrices.

$$\boldsymbol{A} = \sum_{i=1}^{N} \boldsymbol{A}^{(i)}$$

where i is an index over the N elemental matrices and N is the number of finite elements. For an explicit calculation of the matrix elements used to solve Laplace's equation, refer to Appendix A, "The Finite Element Method."

In some formulations, the assembled matrix is never explicitly formed, but operations are performed on the elemental matrices. Storing the matrix as a set of elemental matrices is referred to as clique or finite-element storage [8].

Programming may be more straightforward when the matrix is unassembled and clique storage is used than when the matrix is assembled, but more calculation is involved. Refer to "Assembled vs. Unassembled Matrices" on page 74 for details about the

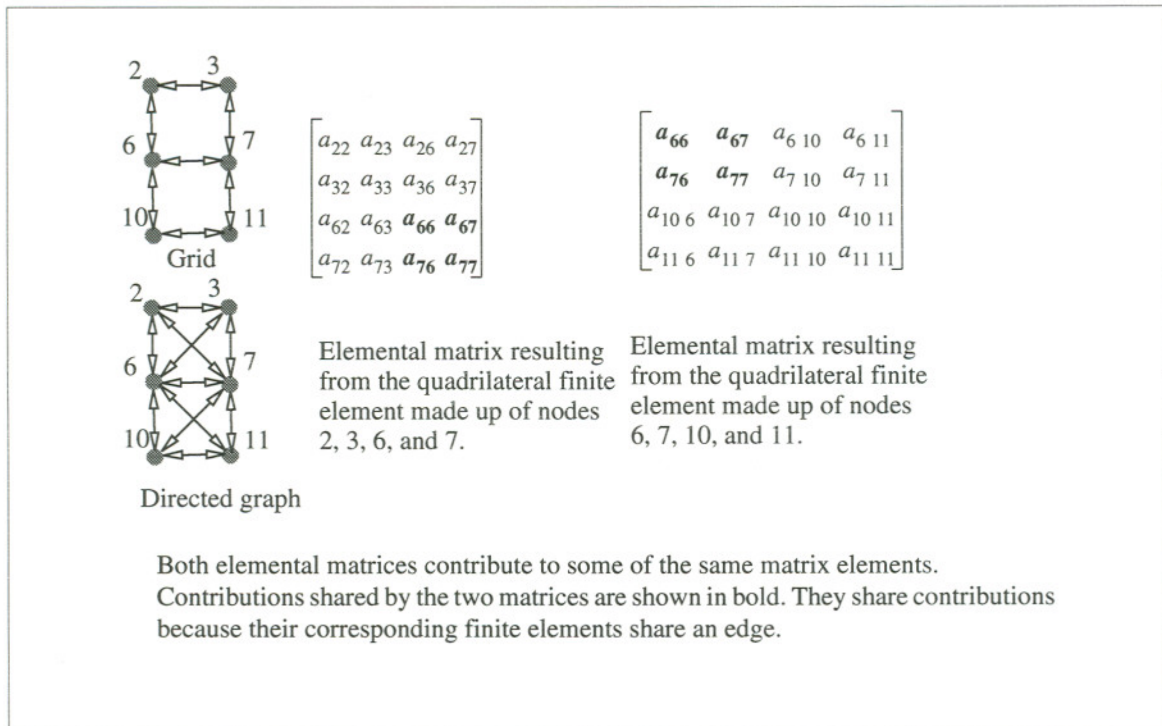


Figure 1.5. Quadrilateral Finite Element and Corresponding Matrix

differences in calculations when using assembled and unassembled matrices.

If, instead of quadrilateral elements, the finite element grid were constructed from triangular elements, an elemental matrix would be 3x3 and contribute at most to nine elements of the assembled matrix.

As with the finite difference method, the solution is again a vector with an element associated with each point in the grid.

Parallel Computers

Because the matrices in question tend to be very large, the sparse matrix equations in which they appear require enormous runtimes to solve. This thesis investigates how to make use of the considerable power of parallel computers to obtain solutions in reasonable time. The motivation behind parallel computing is to use more processors and have them work as a team.

Although many older computers were called sequential, they often had some degree of parallelism [1]. The Cray-1 had twelve pipelined functional units that could operate concurrently. The Cray-YMP features eight identical CPUs, each of which has parallel functional units. Massively parallel computers have up to hundreds or even thousands of processors called *compute nodes*. When these nodes cooperate to solve a

large problem, they achieve supercomputing performance.

Types of Parallel Computers

Flynn [11] divided architectures into the following four types, based on the number of their instruction and data streams.

- SISD has a single instruction stream and a single data stream.
- SIMD has a single instruction stream and multiple data streams.
- MIMD has multiple instruction streams and multiple data streams
- MISD has multiple instruction streams and a single data stream.

The conventional von Neumann architecture is SISD. A VLIW computer (VLIW stands for very long instruction word) extends SISD architecture by packing several instructions into one word. Different portions of the word tell different functional units what to do.

The processing elements of a SIMD machine stay synchronized with each other and execute the same instruction at the same time. The classic example is the ILLIAC-IV developed at the University of Illinois in the 1960's [19]. This machine had 64 processors, each with its own memory. A common control unit would broadcast an instruction to all 64 processors, and each processor would then execute the instruction simultaneously, each using data from its own memory.

A more recent example of a SIMD machine is the Connection Machine CM-200 developed by Thinking Machines. Here a front-end computer issues instructions to up to 64K 1-bit processors. Each processor then operates on data in its own local memory.

A MIMD machine has multiple instruction streams as well as multiple data streams. Such a machine has distinct programs operating on separate sets of data. This category contains most multiprocessor systems. Examples include Sequent's Symmetry, BBN's TC2000, Intel's Paragon supercomputer, and Thinking Machine's Connection Machine CM-5.

Flynn does not distinguish between shared and distributed memory. According to his taxonomy, Sequent's Symmetry and Intel's Paragon are both MIMD computers, yet they have very different architectures. The Symmetry machine has shared memory while the Paragon machine has distributed memory.

In a shared memory machine, different processors read and write the same locations in memory. The processors all share a common address space. The techniques for controlling access to shared memory are similar to those used by sequential

multiprocessing machines to control access by different processes.

The memory making up this shared address space may be centralized or distributed. In multiprocessor systems with uniform memory access (UMA), the shared memory is centralized. The drawback with UMA systems is that as the number of processors becomes large, designing hardware that can control access to the common memory pool while maintaining performance becomes increasingly difficult. In multiprocessor systems with non-uniform memory access (NUMA), the shared memory is physically distributed, but there is still one global address space.The consequence is that the time to access a memory location varies, depending on whether that memory location is local or remote to the processor.

In a multicomputer, the memory is physically distributed, and each processor has its own address space. Compute nodes communicate with each other by exchanging messages.

The work in this thesis was performed on Paragon XP/S systems of various sizes. The Paragon XP/S supercomputer has multiple instruction streams because each compute node runs its own program. Each compute node also has its own local memory containing its own set of data. Hence, there are also multiple data streams [28].

The MISD category rounds out the taxonomy. One view is that such a machine has not been built; another view calls pipelined processors MISD machines [1]. Pipelined processors break up an instruction into a number of steps and then run the same data through those steps. Another MISD possibility is a systolic array. Such a machine moves data from processor to processor, where each processor performs a different operation on the data.

Topology

The network topology defines how the compute nodes are connected. Three main characteristics of a network topology for a parallel computer are *diameter*, *bisection width*, and the *number of connections per node* [31].

The diameter is the largest distance between any two nodes and measures the longest distance a message must travel. The bisection width is the minimum number of edges that must be removed to divide the network in two, times the width of an edge. The width of an edge is the number of serial lines an edge contains. Bisection width measures how good data movement can be from one end of the network to another. Finally, the number of connections per node is the actual physical number of connections per compute node.

Many different topologies have been successful. The two discussed in this thesis, the hypercube and the two-dimensional mesh, have turned out historically to be the most important. The mesh architecture of the Paragon supercomputer traces its ancestry to the hypercube topology

of the iPSC supercomputers, also from Intel.

A high performance network achieves good data movement with a low diameter and a high bisection width. A constant number of connections per node contributes to scalability. Other desirable features are a low cost/performance ratio and a flat interconnect.

- *Scalability* means that a program running on a small number of nodes also runs on a large number of nodes. The same problem runs faster while larger collections of nodes permit the solution of larger problems in reasonable time. Several ways of measuring scalability exist, and they will be discussed in "Speedup and Amdahl's Law" on page 15.
- A low *cost/performance ratio* means that each compute node is priced low enough so that large numbers of them can be connected together to provide significant computing power. Each compute node of the Paragon XP/S supercomputer contains two i860 XP processors. One is dedicated to user processing; the other is dedicated to message-passing. The i860 XP microprocessor has a peak floating point performance of 75 MFLOPS double precision and 100 MFLOPS single precision [28]. Paragon XP/S systems of up to 1800 nodes are in operation.
- A *flat interconnect* means that programers need not be concerned with the actual physical layout of the compute nodes. That is, message-passing latency and bandwidth are independent of where the communicating compute nodes lie in the network. Although a flat interconnect never strictly exists, inasmuch as it is true, programs perform predictably on different subsets of nodes that are physically laid out differently.

A fully-connected network (one where every compute node is connected to every other compute node by a bidirectional link) appears at first to be the most desirable, but it has a high cost/performance ratio and is not scalable. A single bus provides a low cost/performance ratio, but the bus becomes overloaded with only a few compute nodes.

The dimension of a hypercube is the number of nearest neighbors for an individual node. For example, a three-dimensional hypercube has three nearest neighbors; a four-dimensional hypercube has four nearest neighbors, etc. Figure 1.6 shows the connectivity of a three-dimensional and a four-dimensional hypercube.The earlier iPSC supercomputers from Intel could have up to seven dimensions. This means that the largest iPSC supercomputer had 128 nodes, each with seven nearest neighbors.

A d-dimensional hypercube has a bisection width of $width * 2^{d-1}$. That is, a

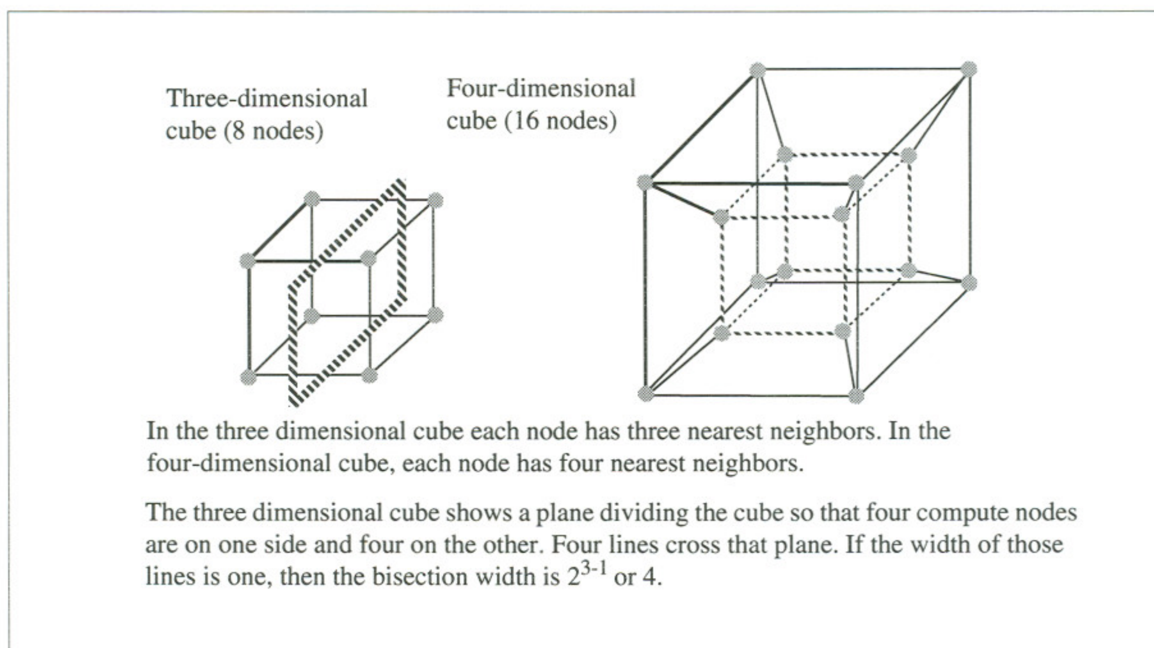


Figure 1.6. Examples of three- and Four-Dimensional Hypercubes

three-dimensional hypercube, when cut so that half the compute nodes are on one side of the cut and half on the other, has 2^2 or four lines crossing the midpoint. A four-dimensional hypercube has 2^3 or eight lines crossing the midpoint. Because the iPSC system has bit-serial communication, *width* is equal to one, and these numbers are also the bisection width.

Hypercubes are expanded by adding a dimension. Hypercubes go from two compute nodes to four to 16 to 32, etc. Every dimension requires another communication line for each compute node.

There are two difficulties with this scheme. The first is the additional hardware design and production that adding another communication link entails. The other is that the required expansion may be more than one really wants. For example, a ten-dimensional hypercube (1024 compute nodes) may not have enough compute nodes, but an eleven-dimensional hypercube (2048 compute nodes) may contain more than one's budget can justify.

Often, a dsigner is faced with the need for a fixed bisection width, whcih is related to the cost. A bisection width of 512 can be achieved by a ten-dimensional hypercube with serial lines ($2^{10-1} = 2^9 = 512$) or a 32x32 two-dimensional mesh with 16-bit lines (32*16 = 512).

A two-dimensional mesh has a low bisection width, but high scalability and a low cost/performance ratio. Intel moved to a mesh architecture to permit the building of larger,

more powerful machines and to increase the flexibility of choosing a configuration. Even for large machines, one can choose to add only a few extra compute nodes for added power.

How the Mesh Appears in Real Life

Although the topology is a two-dimensional mesh, the Paragon compute nodes actually reside in 16-slot card cages. The mesh can grow in both directions, but it cannot get higher than 16 nodes. This is because each cabinet has at most four card cages, and each card cage adds at most four compute nodes to the vertical direction.

The actual connection to the network is through Paragon mesh routing chips (MRCs) on the backplane, one for each compute node. These MRCs route message packets between any two nodes. Their peak performance is 175M bytes/second full duplex. Figure 1.7 illustrates the Paragon mesh network.

The Paragon supercomputer has several different usage models. One model is for the user to allocate a certain number of compute nodes in a user partition. The user can specify the length and width of that partition. The node numbers used by the user program are not the same as the physical node numbers shown in Figure 1.7. The numbering of allocated compute nodes in a user partition begins in the upper left-hand corner and proceeds left to right across the columns and then down the rows.

Speedup and Amdahl's Law

Speedup is a measure of the performance improvement attained when one uses more and more compute nodes to solve a problem. There are several ways of measuring speedup; this section distinguishes between the major ways and identifies the speedup measure used in Chapter 7, "Performance."

The simplest measure of speedup is the time the best sequential algorithm takes on one processor divided by the time the best parallel algorithm takes on n processors. Linear speedup is equal to the number of processors. That is, the application runs twice as fast on two processors, three times as fast on three processors, etc.

Speedup may in certain circumstances be supralinear [31]. For example, each compute node may have its own distinct cache, and the cache hit rate for n processors may be greater than that for one processor. Barring these special circumstances, speedup is typically less than linear, and the closer to linear it is, the more parallelizable the algorithm is considered.

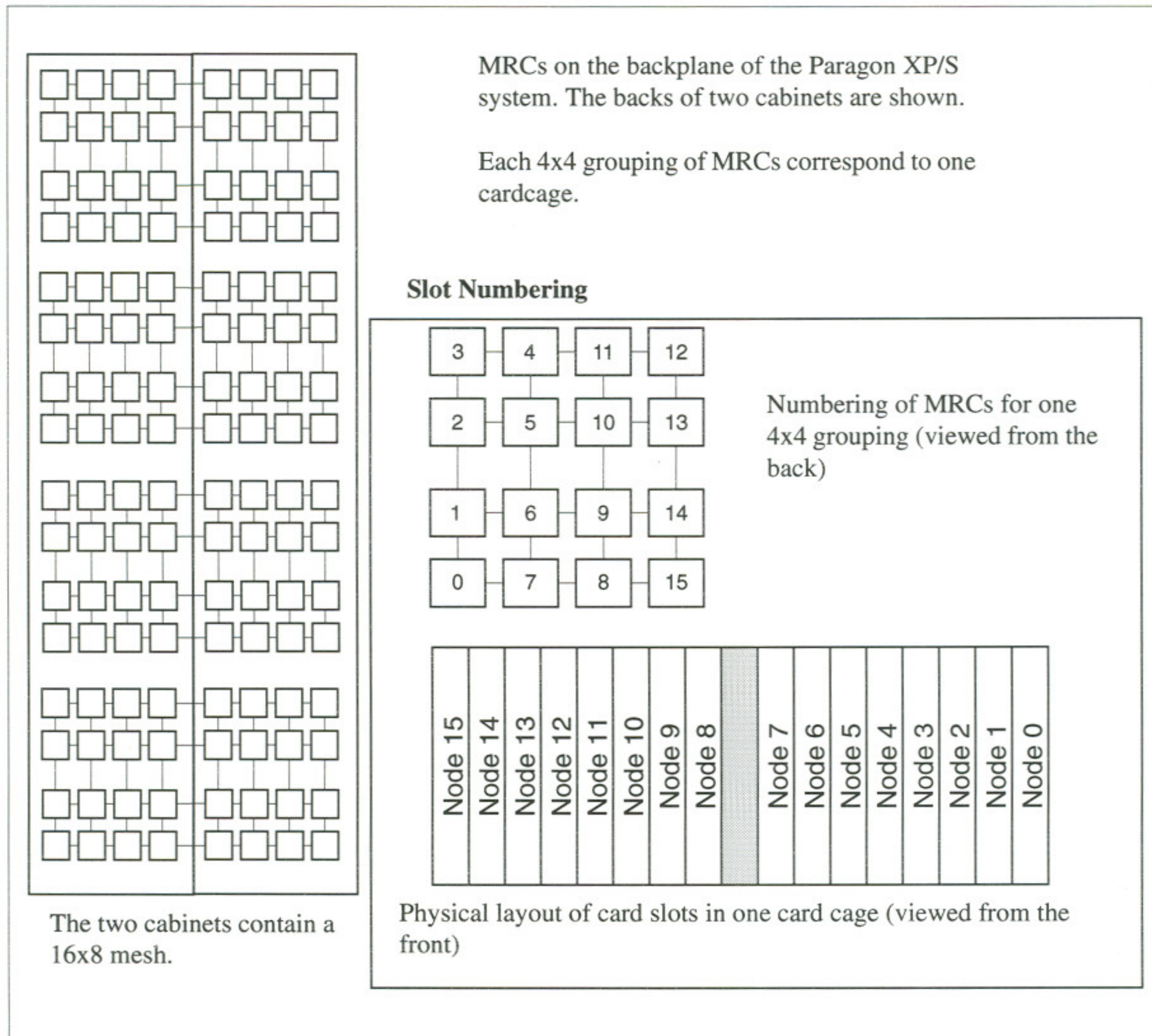


Figure 1.7. Paragon XP/S Mesh Network

Amdahl's Law

In any application, a certain percentage of operations must be performed sequentially. Often I/O consists of sequential operations. *Amdahl's Law* [2] expresses the maximum speedup S as a function of the fraction of the operations that must be sequential f, and the number of processors p. When f is zero, the speedup is the number of processors.

$$S \leq \frac{1}{f + (1-f)/p}$$

In the limit as p goes to infinity, the maximum speedup is $1/f$. For example, this means that if 10% of the operations must be sequential, the speedup can never be greater than 10, no matter how many processors are used.

This is not as bad as it seems. For many applications, the percentage of required

sequential operations decreases as the problem size increases. The sequential operations are usually independent of the problem size, sometimes slowly growing with problem size. This sequential overhead can be amortized over larger and larger problem sizes, thus increasing the speedup.

f is called the amdahl fraction. If n is the problem size and $f(n)$ approaches 0 as n approaches infinity, the the speedup S approaches p [27]. This is then linear speedup.

$$\lim_{f \to 0} S \leq \frac{1}{f + (1-f)/p} = \frac{p}{fp + 1 - f} = \frac{p}{1 + (p-1)f} = p$$

Scaled Speedup

With a multicomputer, more processors means that more memory is available and that a larger problem can be solved; but because a multicomputer has no shared memory, the large data sets required by large problems may not fit into the memory on a single processor. So one cannot actually run the best sequential algorithm on one processor. Instead, one must extrapolate the time that the large problem would take on the single processor. Scaled speedup is this extrapolated time divided by the measured time on n processors.

Fixed-Grain Speedup

Another way of presenting results is to increase the problem size when the number of compute nodes increases. For example, if *size* is the size of a problem on one processor, then $size*p$ is the size on p processors. This ensures that each processor is always doing the same amount of work and is referred to as speedup with a fixed grain size, distinguished from speedup with a fixed problem size.

Parallel Programming

Good parallel programming is characterized by load balancing, overlapping computation with communication, and making maximum use of data locality.

Load balancing means that all the compute nodes are kept busy. No cycles are wasted by having some compute nodes wait for others to finish. Load balancing can be static; that is, the problem is divided up evenly among the compute nodes. Sometimes dynamic load balancing techniques are employed. As the calculation proceeds and the amount of work remaining for each compute node changes, the work load is redistributed to restore an even balance.

Computation and communication overlap can be achieved programmatically and architecturally. For example, asynchronous message passing calls allow the programmer to post a receive and then continue with the computation until the actual received data are required. The compute node of the Paragon supercomputer has two processors, one of which is dedicated to message passing. The computational and communication processors share an address space.

In data parallel programming (see"Data Parallelism" on page 38), the data set is divided among the compute nodes. If a compute node needs data that exists on another compute node, it must exchange messages. By partitioning the data such that each compute node gets most of what it needs and designing the algorithm to use local data as much as possible, the communication/computation ratio is kept low.

The Aliasing Method

This thesis describes an aliasing method applicable to both structured and unstructured grids. In this method, the matrix elements and vectors are distributed among the compute nodes of a multicomputer. This section defines the aliasing method.

Figure 1.9 shows a directed graph divided among compute nodes. The dotted lines designate the compute node boundaries.

The directed graph is divided among the compute nodes along the lines of the graph; that is, no compute node boundary crosses a line in the directed graph. When the dividing line encounters a node of the directed graph, that node is duplicated so that a copy exists on both compute nodes. When several compute node boundaries come together, a copy of the directed graph node exists on each of those compute nodes sharing the boundary. Figure 1.9 shows an instance when three compute nodes share a boundary.

The nodes of the directed graph designate diagonal elements, but they also have vector elements associated with them. Defining a vector on the graph means that each directed graph node has a vector component associated with it. This vector has as many components as there are nodes in the directed graph. For example, node i of the directed graph, as well as designating the diagonal matrix element a_{ii}, has associated with it the ith component of all vectors defined on the graph. When the directed graph node is duplicated on compute nodes sharing a boundary, any associated vector components are also duplicated.

An alias group is a set of copies of one directed graph node (including its associated vector components), each copy residing on a different compute node. Figure

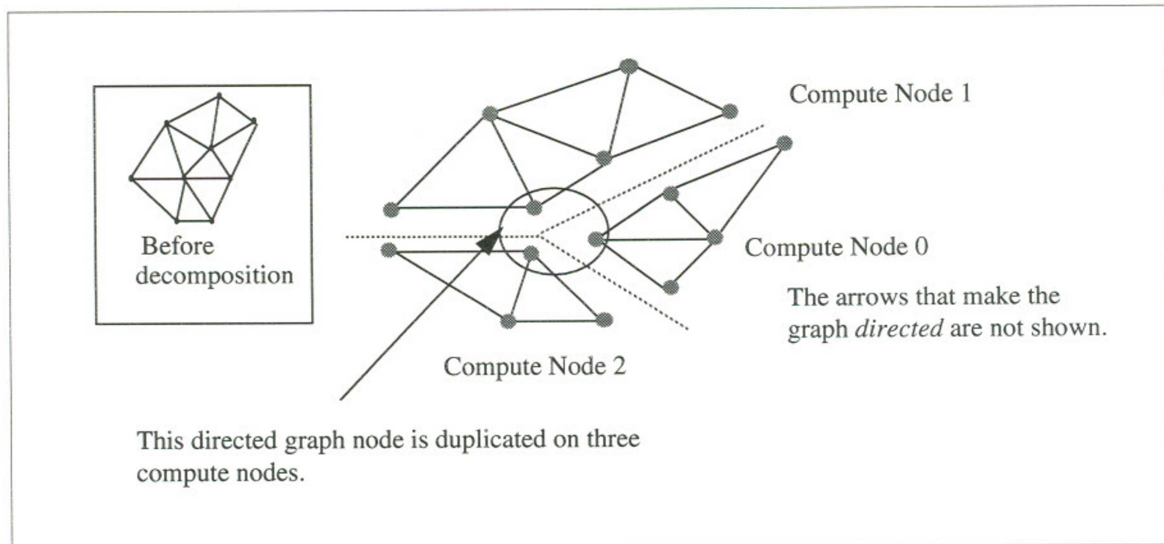


Figure 1.8. Dividing the Mesh Among Compute Nodes

1.9 shows the directed graph of Figure 1.9 with five alias groups identified. Alias groups have the following characteristics:

- Each compute node can both read and modify the values associated with its copy. This means that an alias group acts like *shared memory.*

  The memory used by an alias group is really physically distributed among the compute nodes having members in the alias group. But each compute node reads and writes its copy without worrying about what effect that action has on the other compute nodes.

- When one compute node modifies a value associated with a member of one of its alias groups, that modification is not *immediately* communicated to the other compute nodes that also have members in the same alias group. The guarantee is that the modification will be communicated when it needs to be. This means that the alias group sometimes acts like *weakly coherent* shared memory.

- Note that the different copies making up one vector component may be updated independently. This is different from what one normally considers weakly coherent shared memory Hence, there must exists some method of combining the different values into a single coherent value.

This thesis uses the aliasing method as part of the parallelization of the conjugate gradient method. The conjugate gradient method is an iterative method used to solve a set of simultaneous linear equations. The conjugate gradient method is introduced in "Non-Stationary Methods" on page 34 and described in detail in Appendix C, "The

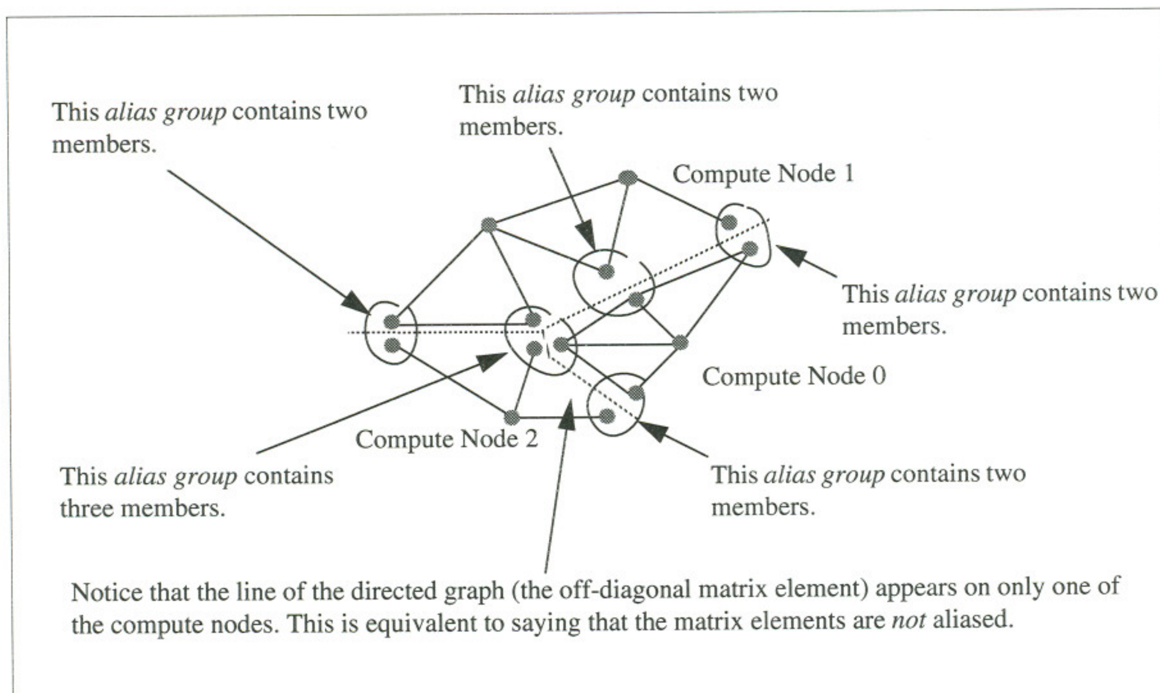


Figure 1.9. Alias Groups

Conjugate Gradient Method." Its parallel aspects are discussed in the section "Parallel Conjugate Gradient Method" on page 55.What is important for this section is to recognize that each step of a conjugate gradient iteration contains a matrix-vector multiplication. This section summarizes how the aliasing method is used to perform these operations.

A matrix-vector multiplication can be thought of as an operation on the directed graph. To see how, consider the matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{Ax}$. The vector $\boldsymbol{y}$ has components associated with each node of the directed graph as does the vector $\boldsymbol{x}$. When the matrix vector multiplication is performed the value of $\boldsymbol{y}$ changes. Each component of $\boldsymbol{y}$ gets contributions from every component of $\boldsymbol{x}$ to which it is connected by a matrix element. That is, the ith component of $\boldsymbol{y}$ is

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

A *pull-in* is defined as the contribution one vector component obtains from another vector component, weighted by the connecting matrix element (shown as a line). Figure 1.10 shows the component y_1 with three pull-ins, one from x_2 weighted by a_{12}, one from x_3 weighted by a_{13}, and one from x_4 weighted by a_{14}.

At each step of a conjugate gradient iteration, a matrix vector multiplication is performed. When this operation is executed in parallel, each compute node performs its pull-ins locally. The result is that each member of an alias group gets a value different

$y_1 = a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4$

The vector component y_1 contains contributions from the vector components x_2, x_3, and x_4. These contributions are weighted by the matrix elements designated by the connecting lines.

The term $a_{11}x_1$ is the self-term.

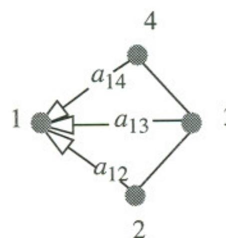


Figure 1.10. Pull-ins

from that of the other members; this is the same as saying that the alias group becomes *incoherent*. After a matrix-vector multiplication, the correct value for the members of an alias group is the sum of all the values in the alias group. Performing that sum and ensuring that it is received by all the compute nodes with a member in the alias group is called making the alias group *coherent*. Figure 1.10 illustrates what happens to the alias group for $y_1$during a matrix-vector multiplication.

Node 1 of the directed graph has six pull-ins. All six pull-ins do not occur on the same compute node. The figure shows that compute node 0 performs three pull-ins, compute node 1 performs two pull-ins, and compute node 2 performs one pull-in.

At the completion of the matrix-vector multiplication, each member of the alias group for y_1 contains a portion of the resulting value for y_1. The final value is the sum of the values contributed by each compute node.

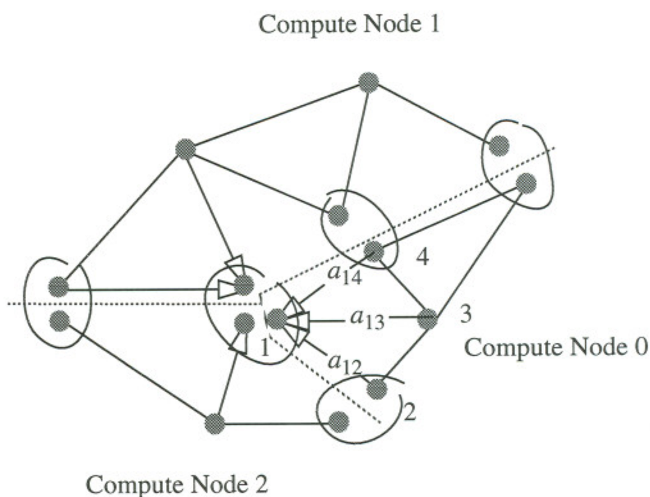


Figure 1.11. Pull-ins Distributed among Alias Group Members

In summary, this thesis uses the aliasing method to achieve an efficient parallelization of the conjugate gradient method. The thesis investigates how alias groups are constructed, stored, managed, and made coherent.

Organization of this Thesis

This thesis has eight chapters and four appendices. They are as follows:

- Chapter 1, "Introduction" gives an overview of the investigation. Some general terminology is defined. A relationship is made and a distinction is drawn between the grids used to discretize the equations and the directed graphs of the associated matrices. In addition, an overview of parallel architecture is presented with emphasis on the Paragon XP/S supercomputer.
- Chapter 2, "Solution Methods"discusses serial direct and serial iterative solution methods for sparse matrix equations. The direct methods are based on Gaussian elimination, and the iterative methods are based on the conjugate gradient algorithm.
- Chapter 3, "Parallelization Techniques"discusses how the serial methods described in the previous chapter can be adapted to parallel machines.
- Chapter 4, "Implementation of Aliasing"discusses how the grid is partitioned among the compute nodes of a parallel machine and how this affects the matrix and vector elements. Work proceeds independently on each compute node. When shared vector elements become incoherent, communication is required. Aliases are defined and shown to be a way of facilitating this communication.
- Chapter 5, "Implementation of the Conjugate Gradient Method" describes the implementation in enough detail to allow for further work. Individual routines and data structures are described.
- Chapter 6, "A Random Matrix" presents a limiting case where the aliasing method, although still working and producing correct answers, becomes inefficient. The Numerical Aerodynamic Simulation (NAS) Conjugate Gradient (CG) benchmark [3] is shown to provide a random matrix—a sparse matrix whose nonzero elements are randomly distributed throughout the matrix.
- Chapter 7, "Performance" presents data taken from solutions of the Laplace matrix and the random matrix from the NAS CG benchmark.
- Chapter 8, "Conclusions" summarizes the results and suggests possible extensions.

- Appendix A"The Finite Element Method" presents the mathematical background of the finite element method. Specifically, the method is applied to Laplace's equation, a two-dimensional, second order, linear partial differential equation.
- Appendix B, "Exact Solution of Laplace's Equation" presents an exact, analytic solution of Laplace's equation in continuous space. The purpose is to compare this exact solution with the answer obtained using finite elements and the conjugate gradient method.
- Appendix C, "The Conjugate Gradient Method" lists the steps of the conjugate gradient method.
- Appendix D, "The NAS Benchmark" describes the details of the NAS CG benchmark.

Chapter 2 Solution Methods

Introduction

This chapter gives an overview of direct and iterative solution methods for sparse matrix equations.

- *Direct methods* obtain a solution after a predictable number of operations. Intermediate steps do not approximate the solution. If it were not for roundoff errors, this solution would be completely accurate.
- *Iterative methods* generate a sequence of approximations that converge to the solution. The computation stops when the solution meets some specified accuracy or after a specified number of iterations. The number of operations is not predictable and depends on stopping criteria. Intermediate steps do approximate the solution.

Serial Direct Methods

The direct methods described in this thesis are based on variations of *Gaussian elimination*. For example, *LU factorization* takes the matrix equation $\boldsymbol{Ax} = \boldsymbol{b}$ and turns it into $\boldsymbol{LUx} = \boldsymbol{Ly} = \boldsymbol{b}$. If $\boldsymbol{A}$ is an nxn matrix, $\boldsymbol{x}$, and $\boldsymbol{b}$ are n-length vectors. $\boldsymbol{U}$ is an upper triangular matrix, and $\boldsymbol{L}$ is a lower triangular matrix.

Solving the set of equations $\boldsymbol{Ly} = \boldsymbol{b}$ for $\boldsymbol{y}$ is called *forward reduction* or *forward elimination* and is performed as follows:

$$y_1 = \frac{b_1}{l_{11}} \qquad y_i = \frac{b_i - \sum_{j=1}^{i-1} l_{ij} y_j}{l_{ii}} \qquad i=2,\ldots,n$$

where y_i and b_i are the ith elements of the $\boldsymbol{y}$ and $\boldsymbol{b}$ vectors and l_{ij} is the ij^{th} element of the matrix L. Once $\mathbf{y}$ is known, the matrix equation $\boldsymbol{Ux}=\boldsymbol{y}$ can easily be solved. This step is called *back substitution* and is performed as follows:

$$x_n = \frac{y_n}{u_{nn}} \qquad x_i = \frac{y_i - \sum_{j=i+1}^{n} u_{ij}x_j}{u_{ii}} \qquad i=1,\ldots,n-1$$

where x_i and y_i are the ith elements of the $\boldsymbol{x}$ and $\boldsymbol{y}$ vectors and u_{ij} is the ij^{th} element of the matrix U.

This LU factorization is performed with row operations. A row operation consists of multiplying one row of the matrix by some constant and adding the result to another row. The row operations are designed to zero out or eliminate the lower triangular part of $\boldsymbol{A}$ and thus form $\boldsymbol{U}$. The constants used as the row multipliers form the elements of $\boldsymbol{L}$, and usually, these are stored in the newly-zeroed locations of $\boldsymbol{A}$.

LU factorization proceeds in stages. Each stage eliminates the matrix elements below the diagonal in one column. The factor multiplying the row is chosen so that when it multiplies the diagonal element in the row (called the *pivot element*), the value is such that when it is added to an element in the column below, zero results. A different factor is chosen for each nonzero element in the column below.

LU factorization proceeds left to right, zeroing out a column at a time until there are no more. There is one less stage than there are columns in the matrix. The *remaining submatrix* consists of the columns to the right of the column being eliminated and the rows below the row containing the pivot [7]. Gaussian elimination occurs when the corresponding operation is performed on the vector on the right-hand side of the equation.

Pivoting

The Gaussian elimination algorithm requires division by the pivot element; and if the pivot element is zero or small, numerical instability and inaccuracy results [32]. Pivoting means rearranging the rows and columns of a matrix to bring an acceptable matrix element into the pivot position.

Interchanging rows or columns of the matrix does not modify the matrix equation. However, when columns in $\boldsymbol{A}$ are interchanged, the corresponding rows in $\boldsymbol{x}$ must also be interchanged; and when rows in $\boldsymbol{A}$ are interchanged, the corresponding rows in $\boldsymbol{b}$ must also be interchanged.

Interchanging rows is called *partial row pivoting*. Gaussian elimination with partial row pivoting is considered to be a stable algorithm, but that consideration is based more on experience than rigorous analysis [8].

Interchanging both rows and columns is called *full pivoting*. Gaussian elimination with full pivoting is usually not used because it involves more matrix element comparisons and is not needed.

Ordering Schemes

Direct methods rely heavily on *ordering schemes*. An ordering scheme is a method by which the points on a grid are numbered. For example, in Figure 1.4 on page 4, the ordering scheme is to number the grid points designating unknown values of the solution vector, starting with 1 and proceeding left to right on each row. Then, the grid points designating known values of the solution vector are numbered, also left to right and down the rows.

Choosing a different ordering scheme does not alter the problem being solved. It only changes the pattern of nonzero elements in the matrix. If the grid is reordered so that i goes to k and j goes to l, the matrix element a_{kl} now has the value that a_{ij} used to have. The corresponding reordering must occur for the vectors as well as the matrix elements. What was called x_i is now called x_k and what was called x_j is now called x_l.; what was called b_i is now called b_k and what was called b_j is now called b_l.

Changing the pattern of nonzero elements in the matrix can be very beneficial. It may reduce the operation count and storage requirements of the solution. It accomplishes this by reducing *fill-in*.

Fill-in

A major concern of direct sparse matrix methods is *fill-in*. Fill-in occurs when in the course of the calculation a matrix element that was zero becomes nonzero [15]. With sparse matrices, only their nonzero elements need be stored, resulting in a significant decrease in memory requirements. A good sparse storage scheme has to allow for fill-in, and a good direct algorithm manages fill-in. Managing fill-in means minimizing it, confining it to certain regions in the matrix, or some combination of the two.

Minimizing Fill-in

Minimizing fill-in means using some reordering strategy to minimize the number of zero elements that become nonzero during the course of Gaussian elimination. An effective way of minimizing fill-in is due to Markowitz [8], [26]. At each stage of

Gaussian elimination, the *Markowitz criterion* chooses as a pivot a particular nonzero entry from the remaining submatrix such that the least number of matrix elements will be modified in the subsequent elimination operation.

Minimizing fill-in is a local approach. That is, the reordering is performed at each stage of the elimination; and this reordering may be optimal for minimizing fill-in at that stage, but is not necessarily optimal for the entire elimination which consists of many elimination stages.

The pivot element chosen by the Markowitz criterion must also be large enough to ensure numerical stability. Usually, its magnitude is specified to be that of a specified drop tolerance times the largest possible pivot. Typical *drop tolerances* are 0.1 and 0.01.

Confining Fill-in

Confining fill-in means permuting the matrix to some particular form and then using the properties of that form to more easily obtain a solution. Because this method deals with the matrix as a whole, it is considered a global approach.

For example, if the matrix is block tridiagonal (as shown in Figure 2.1), instead of eliminating each entire column, one can choose an algorithm that factors only the diagonal blocks. In this algorithm, the off-diagonal blocks appear only as part of matrix-vector multiplications [8].

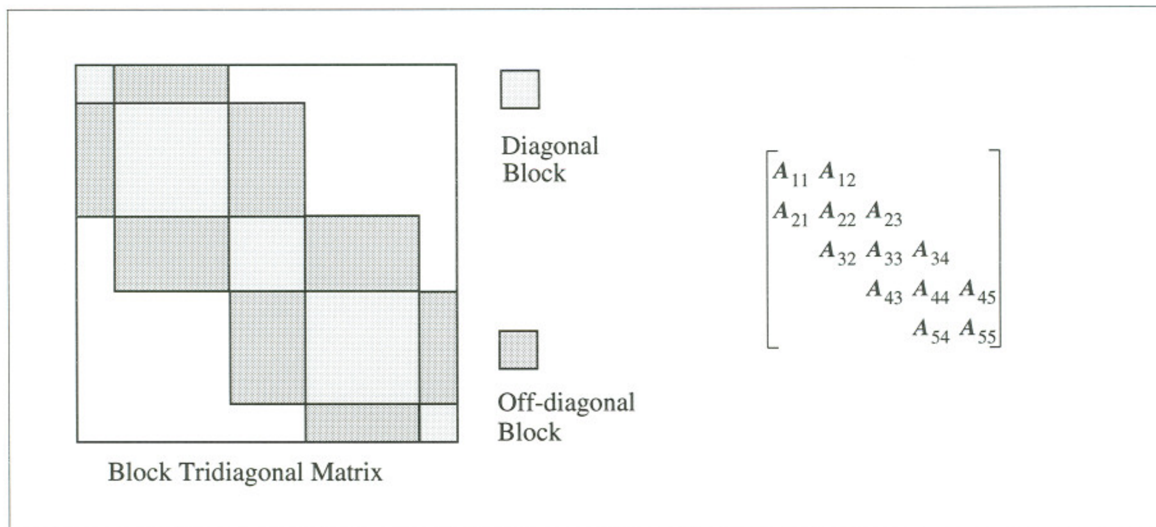


Figure 2.1. Block Tridiagonal Matrix

Frontal Method

Because the matrices in question may be very large, the frontal method was developed to take advantage of memory hierarchy. The frontal method factors the matrix in such a way that the higher levels of the memory hierarchy (where the access is most efficient) get used most frequently. This is possible because it is not necessary to have access to the entire matrix to factor it. At each stage of the factorization only a portion of the matrix is needed.

Assume the matrix is so large that it cannot fit into main memory. A portion of the matrix must always remain on disk. When this occurs, the solver must bring into main memory only the portion of the matrix needed for a particular stage of the algorithm, then replace that portion with the portion needed for the next stage. Such a solver is called an out-of-core solver.

One can think of the frontal method as a window that moves over the matrix. Only the portion of the matrix in the window is needed for the current stage of factorization.

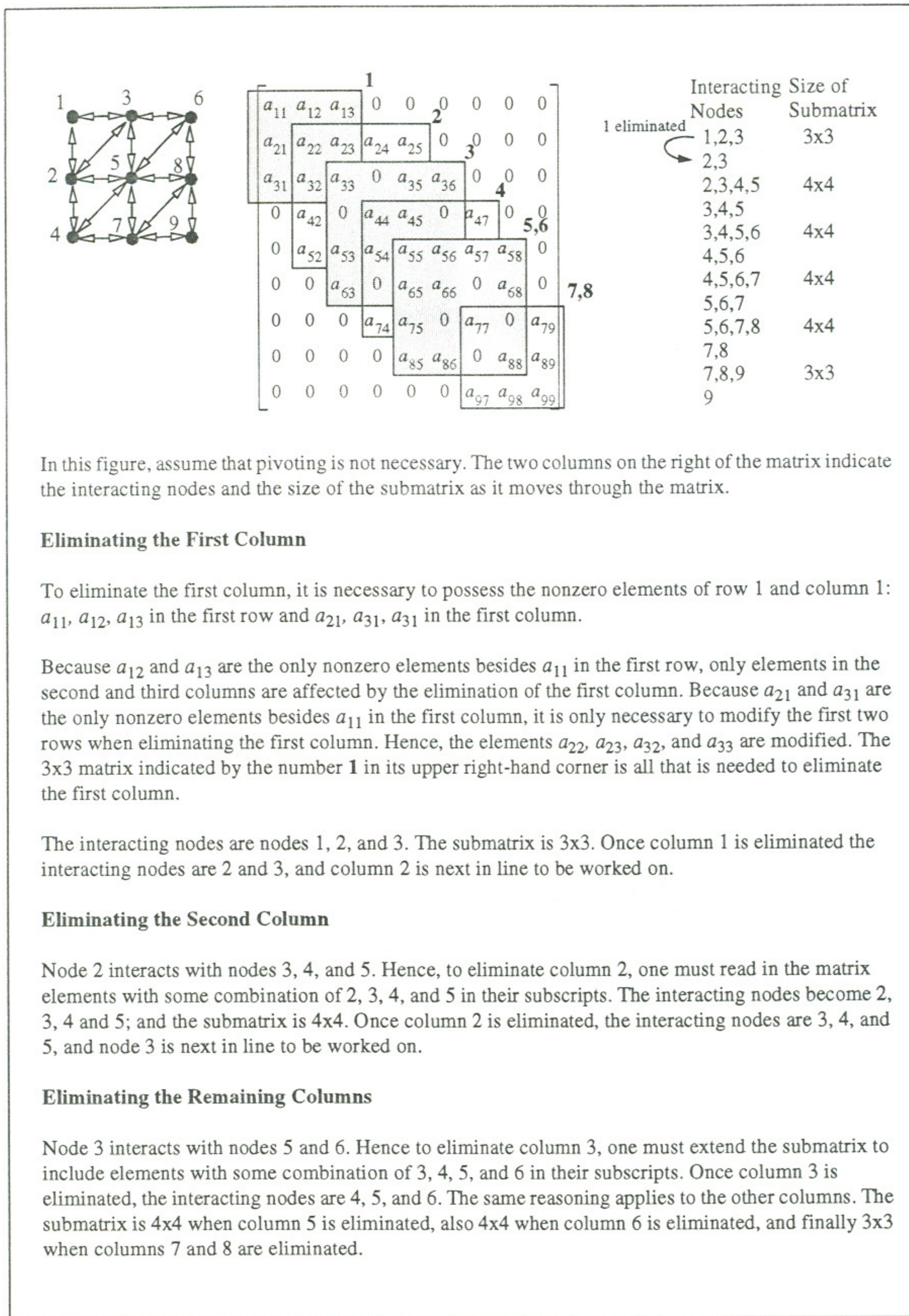
For example, to eliminate a particular column, it is only necessary to have access to the nonzero elements in the row and column containing the pivot. It is convenient to also have the nonzero elements in the rows that are affected by the elimination. Otherwise, one must save up the modifications to these elements and then apply them when the elements become available.

Figure 2.2 shows a nine-point directed graph and its corresponding matrix. A 3x3 matrix consisting of the elements common to the first three columns and the first three rows is all that is needed to eliminate the first column. Only this submatrix is needed to eliminate column 1 because node 1 does not have any interactions with any nodes other than 2 and 3.

The second submatrix has all the elements common to rows 2 through 5 and columns 2 through 5. This submatrix is all that is needed to eliminate column 2. Only this submatrix is needed because node 2 does not have any interactions with nodes other than 3, 4, and 5.

Figure 2.2 shows all the submatrices that are needed to factor the matrix. The submatrices are called frontal submatrices because they are thought of as a wavefront moving through the matrix. As different matrix elements are read into memory and different columns eliminated, the wavefront moves through the matrix.

The bold numbers at the upper right corner of each frontal submatrix indicate the columns that can be eliminated when that submatrix is in memory. The two columns of



In this figure, assume that pivoting is not necessary. The two columns on the right of the matrix indicate the interacting nodes and the size of the submatrix as it moves through the matrix.

Eliminating the First Column

To eliminate the first column, it is necessary to possess the nonzero elements of row 1 and column 1: a_{11}, a_{12}, a_{13} in the first row and a_{21}, a_{31}, a_{31} in the first column.

Because a_{12} and a_{13} are the only nonzero elements besides a_{11} in the first row, only elements in the second and third columns are affected by the elimination of the first column. Because a_{21} and a_{31} are the only nonzero elements besides a_{11} in the first column, it is only necessary to modify the first two rows when eliminating the first column. Hence, the elements a_{22}, a_{23}, a_{32}, and a_{33} are modified. The 3x3 matrix indicated by the number **1** in its upper right-hand corner is all that is needed to eliminate the first column.

The interacting nodes are nodes 1, 2, and 3. The submatrix is 3x3. Once column 1 is eliminated the interacting nodes are 2 and 3, and column 2 is next in line to be worked on.

Eliminating the Second Column

Node 2 interacts with nodes 3, 4, and 5. Hence, to eliminate column 2, one must read in the matrix elements with some combination of 2, 3, 4, and 5 in their subscripts. The interacting nodes become 2, 3, 4 and 5; and the submatrix is 4x4. Once column 2 is eliminated, the interacting nodes are 3, 4, and 5, and node 3 is next in line to be worked on.

Eliminating the Remaining Columns

Node 3 interacts with nodes 5 and 6. Hence to eliminate column 3, one must extend the submatrix to include elements with some combination of 3, 4, 5, and 6 in their subscripts. Once column 3 is eliminated, the interacting nodes are 4, 5, and 6. The same reasoning applies to the other columns. The submatrix is 4x4 when column 5 is eliminated, also 4x4 when column 6 is eliminated, and finally 3x3 when columns 7 and 8 are eliminated.

Figure 2.2. Demonstration of the Frontal Method

numbers in the right-hand side of the figure show the interacting nodes and the size of their corresponding submatrix. The text in the figure describes the steps in detail.

The wavefront determines the access pattern to the matrix elements; consequently, an ordering scheme that results in a small wavefront would allow the frontal method to make the most efficient use of memory hierarchy. Ideally, the submatrix currently being worked on resides in the smallest, fastest cache.

The *multifrontal method* is an extension of the frontal method that is applicable to parallel processors [10]. In the multifrontal method, several wavefronts can be moving through the matrix independently. See “Parallel Direct Methods” on page 39 for more information about multifrontal methods.

Serial Iterative Methods

Iterative methods have several advantages over direct methods. First of all, iterative methods usually require less storage. For large matrices, the fill-in and the subsequent increase in operation count in direct methods is significant. Because iterative methods do not have fill-in, the sparse storage scheme used by the matrix need not accommodate additional values during the course of the algorithm.

In addition, iterative methods are often faster than direct methods. Usually, the answer at each iteration is tested against some desired measure of accuracy. For example, in a relaxation method one may check on how much the new answer differs from the old, either in absolute magnitude or as a percentage and accept an answer that changes less than a set value. One need not work to an accuracy greater than required by the problem being worked on.

Iterative methods also often have simpler vectorization and parallelization properties. On sequential machines the primary goal of reordering the matrix is to decrease storage requirements (less fill-in) and decrease work (less operation count). Techniques that achieve these goals do not generally also promote parallelism as will be seen in“Parallel Direct Methods” on page 39.With iterative methods, however, parallelization can be as straightforward as assigning different nodes of the grid to different processors.

Stationary and Non-Stationary Methods

Iterative methods divide into stationary and non-stationary methods [4]. The

stationary methods discussed in this thesis are the Jacobi method itself, the Gauss-Seidel method, and the Successive Overrelaxation method. The non-stationary methods discussed are the conjugate gradient method and the preconditioned conjugate gradient method.

Stationary Methods

Stationary methods are called stationary because, although the solution vector changes with each iteration, the information used to update this vector remains the same. A stationary iterative method can be described by the matrix equation

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{B}\boldsymbol{x}^{(k)} + \boldsymbol{c}$$

where $\boldsymbol{x}$ is the solution vector and the superscript in parentheses is the iteration count. For example, $\boldsymbol{x}^{(k+1)}$ is the value of the solution vector $\boldsymbol{x}$ at the $(k+1)^{\text{th}}$ iteration. The equation shows how the value at the $(k+1)^{\text{th}}$ iteration depends on the value at the kth iteration. $\boldsymbol{B}$ is the *iteration matrix*, which is calculated from the original $\boldsymbol{A}$. Neither $\boldsymbol{B}$ nor $\boldsymbol{c}$ (a constant vector) depend on the iteration count k.

The iteration matrix is written for the stationary methods described below in terms of $\boldsymbol{L}$, $\boldsymbol{U}$, and $\boldsymbol{D}$ where $\boldsymbol{A}=\boldsymbol{L}+\boldsymbol{U}+\boldsymbol{D}$. $\boldsymbol{L}$ is lower triangular with zeros on the diagonal, $\boldsymbol{U}$ is upper diagonal with zeros on the diagonal, and $\boldsymbol{D}$ is diagonal [33].

The iteration matrix itself is important because its properties determine the convergence of the method. If the spectral radius of $\boldsymbol{B}$ is less than one, then the method converges; the smaller the spectral radius, the faster the convergence [33]. The spectral radius of a symmetric matrix is defined as the absolute value of its largest eigenvalue.

Serial Jacobi Method

In the Jacobi method, each element of $\boldsymbol{x}$ is updated from the old estimate of $\boldsymbol{x}$. To derive an equation for the Jacobi iteration, first recall the matrix form of the set of simultaneous linear equations,

$$0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$$

which in component form is

$$0 = b_i - \sum_{j=1}^{n} a_{ij}x_j = b_i - \sum_{j=1}^{i-1} a_{ij}x_j - a_{ii}x_i - \sum_{j=i+1}^{n} a_{ij}x_j$$

Notice that the above equation is written is such a way as to break out the term $a_{ii}x_i$. If that term is brought to the left-hand side of the equation and then both sides divided by a_{ii}, the result is

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^{n} a_{ij}x_j\right)$$

Now put an initial guess for $\boldsymbol{x}$ into the left-hand side of the above equation. Then, calculate a new x using the same equation. The value of $\boldsymbol{x}$ at each iteration is called the iterate of $\boldsymbol{x}$. The iteration number is shown as superscript in parentheses. The following equation shows the value for the $(k+1)^{th}$ iterate of $\boldsymbol{x}$, given the kth iterate [4], [33].

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

The equation shows the value for the $(k+1)^{th}$ iterate.

In matrix notation, the Jacobi iterates are defined as $\boldsymbol{x}^{(k+1)} = -\boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})\boldsymbol{x}^{(k)} + \boldsymbol{D}^{-1}\boldsymbol{b}$. This form identifies $-\boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})$ as the iteration matrix [33].

Figure 2.3 shows a 4x4 grid where the grid points are numbered left to right. The nodes are updated in numerical order, and there is a vector defined on the grid. This vector has 16 elements, with one element associated with every grid point. In one iteration, each of the 16 nodes is updated.

In the Jacobi method there are actually two vectors, an old one and a new one constructed from the old one. After one iteration, the new vector becomes the old and another new vector is formed.

Serial Gauss-Seidel Method

The Gauss-Seidel method differs from the Jacobi method by making use of the updated values of $\boldsymbol{x}$ as soon as they are available. Hence, when implementing the Gauss-Seidel method, the vector iterate can be updated in place. Consequently, the Gauss-Seidel method has less storage requirements than the Jacobi method and converges more rapidly.

If there are n unknowns, the equation for each unknown is

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

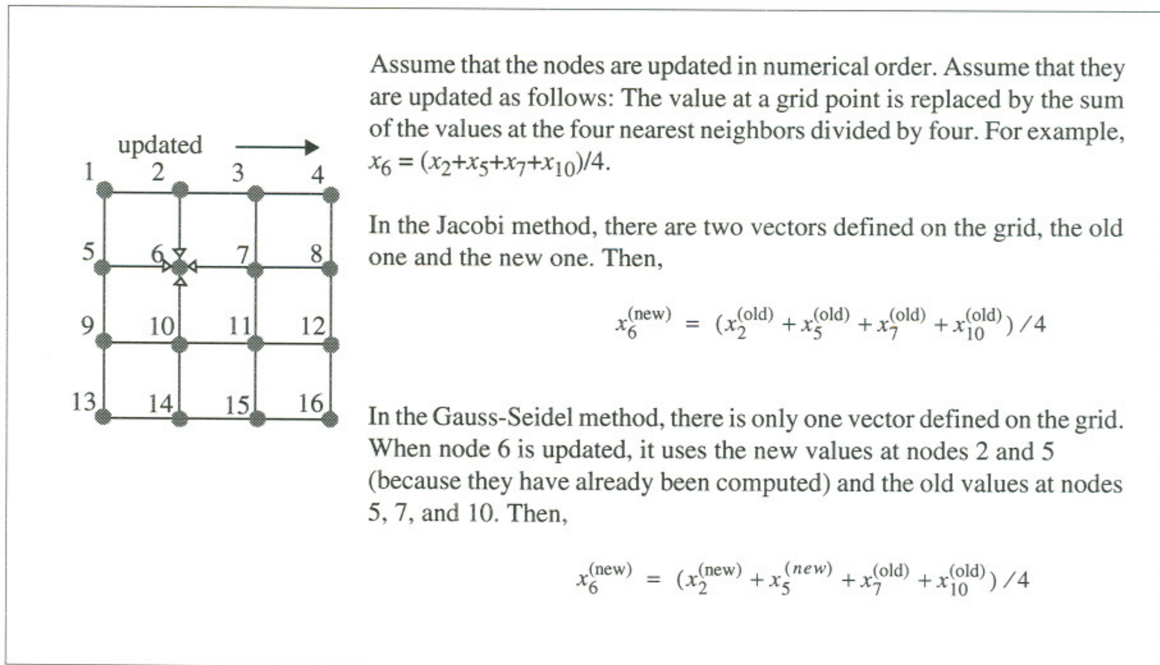


Figure 2.3. 4x4 Square Grid Used to Illustrate the Jacobi and Gauss-Seidel Methods

where the superscripts in parentheses again identify the iteration number. The vector elements whose index is less than the vector element being updated use the value of the current iteration [4], [33].

In matrix notation, the Gauss-Seidel iterates are defined as $\boldsymbol{x}^{(k+1)} = -(\boldsymbol{L}+\boldsymbol{D})^{-1}\boldsymbol{U}\boldsymbol{x}^{(k)} + (\boldsymbol{L}+\boldsymbol{D})^{-1}\boldsymbol{b}$. This form identifies $-(\boldsymbol{L}+\boldsymbol{D})^{-1}\boldsymbol{U}$ as the iteration matrix [33].

Serial Successive Overrelaxation Method

In cases where the Jacobi and Gauss-Seidel methods converge, but converge too slowly, the successive overrelaxation method (SOR) may be employed. This method consists of first computing the Gauss-Seidel iterate $\hat{\boldsymbol{x}}$, but then considering that to be an intermediate value. Then, the actual iterate obeys the following equation.

$$x_i^{(k+1)} = x_i^{(k)} + \omega(\hat{x}_i^{(k+1)} - x_i^{(k)})$$

where ω is a parameter added to accelerate the convergence. When ω is 1, SOR is the same as the Gauss-Seidel method. Determining the optimal ω is usually either impossible or prohibitively expensive, but a good ω is typically in the range $1 < \omega < 2$.

In matrix notation, the SOR iterates are defined as

$$x^{(k+1)} = (I+\omega D^{-1}L)^{-1}((1-\omega)I-\omega D^{-1}U)x^{(k)} + (I+\omega D^{-1}L)^{-1}\omega D^{-1}b$$

where $(I+\omega D^{-1}L)^{-1}((1-\omega)I-\omega D^{-1}U)$ is the iteration matrix and I is the identity matrix (all elements are zero, except those along the diagonal which are 1) [33].

Non-Stationary Methods

Non-stationary methods are called non-stationary because the information used to update the solution vector changes with each iteration. The non-stationary methods described in this thesis are variations of the *conjugate gradient method*. Each iteration of the conjugate gradient method generates an approximation to the solution, a residual, and a search direction used to obtain the next approximation [4].

Serial Conjugate Gradient Method

The conjugate gradient method is due to Hestenes and Stiefel [18]. It is actually a direct method that yields an exact solution in n steps; but in practice, it is considered an iterative method, because of roundoff errors.

The conjugate gradient method solves the matrix equation $Ax=b$ when A is real, nxn, symmetric and positive definite.The matrix A is positive definite if $x^TAx > 0$ for all $x > 0$. If A is positive definite, convergence is guaranteed; the solution improves with each iteration and is exact after n iterations [32]. When A is not symmetric, variations of the conjugate gradient method such as the biconjugate gradient method and the conjugate gradient squared method can be used [4].

The conjugate gradient method begins by realizing that minimizing the function

$$q(x) = \frac{1}{2}(x \cdot Ax) - (x \cdot b) + c$$

is equivalent to solving $Ax=b$. This can be seen by setting the derivative of $q(x)$ with respect to x to zero and solving for x.

The method entails minimizing $q(x)$ along a set of search directions. A complete solution minimizes along n search directions in n steps, but usually an acceptable answer is obtained in less than n steps. Each step updates the solution vector x as follows:

$$x^{(k+1)} = x^{(k)} + \lambda^{(k+1)}g^{(k)}$$

where k is the iteration number, $\boldsymbol{x}^{(k)}$ is the solution vector at the kth iteration, λ is a constant, and $\boldsymbol{g}$ is the search direction. As can be seen from the equation, the current step (iteration k+1) uses the solution vector and the search direction from the previous step (iteration k), to calculate a new λ and then a new solution vector.

This calculation is performed by substituting the expression for $\boldsymbol{x}^{(k+1)}$ into the equation defining $q(\boldsymbol{x})$ and then minimizing with respect to $\lambda^{(k+1)}$. The λ that minimizes $q(\boldsymbol{x})$ for the $(k+1)^{\text{th}}$ iteration is

$$\lambda^{(k+1)} = \frac{\boldsymbol{r}^{(k)} \cdot \boldsymbol{g}^{(k)}}{\mathbf{g}^{(k)} \cdot \boldsymbol{A}\boldsymbol{g}^{(k)}}$$

where $\boldsymbol{r}^{(k)}$ is the residual at the kth iteration. If the solution $\boldsymbol{x}^{(k)}$ were exact, the residual would be zero. The residual is defined as

$$\boldsymbol{r}^{(k)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}$$

At the beginning of each step, one must choose a good search direction. One must first ensure that the search direction is one along which $q(\boldsymbol{x})$ is decreasing. In addition, one wants a search direction that, when $q(\boldsymbol{x})$ is minimized along this direction, the minimizations along the other directions are not destroyed[28] [16].

The first requirement is satisfied by ensuring that the search direction $\boldsymbol{g}^{(k+1)}$ is not orthogonal to the residual $\boldsymbol{r}^{(k)}$. This can be seen by substituting the expression for the updated value of the solution vector ($\boldsymbol{x}^{(k+1)}$) into $q(\boldsymbol{x})$ with the minimizing value of $\lambda^{(k+1)}$

$$q(\boldsymbol{x}^{(k+1)}) = q(\boldsymbol{x}^{(k)} + \lambda^{(k+1)}\boldsymbol{g}^{(k)}) = q(\boldsymbol{x}^{(k)}) - \frac{(\boldsymbol{g}^{(k)} \cdot \boldsymbol{r}^{(k)})^2 \cdot (\boldsymbol{g}^{(k)} \cdot \boldsymbol{A}\boldsymbol{g}^{(k)})}{2}$$

and noting that if $\boldsymbol{g} \cdot \boldsymbol{r}$ is nonzero, $q(\boldsymbol{x}^{(k+1)})$ is less that $q(\boldsymbol{x}^{(k)})$. The second requirement is satisfied if each new search direction is orthogonal with respect to the matrix $\boldsymbol{A}$ to all the previous search directions. That is,

$$\boldsymbol{g}^{(i)} \cdot \boldsymbol{A}\boldsymbol{g}^{(j)} = 0 \qquad \text{for} \qquad i \neq j$$

Search directions chosen in this way are said to be $\boldsymbol{A}$-orthogonal or conjugate with respect to $\boldsymbol{A}$.

Appendix C, "The Conjugate Gradient Method" describes the steps of the conjugate gradient method in detail and in a form suitable for a computer program. The major work in each step consists of a matrix-vector product and two dot products. When the vector $\boldsymbol{g}$ (the search direction) is defined as the negative gradient of q, the method is

the method of steepest descent.

Preconditioned Conjugate Gradient

Preconditioning is a way of accelerating convergence of the conjugate gradient method. The conjugate gradient method does always give (barring round-off error) an exact answer in n steps; but the preconditioned conjugate gradient method reaches an acceptable answer sooner, where acceptable is usually defined as a small enough residual.

Condition Number and Convergence

The rate of convergence of the conjugate gradient method depends on the *condition number* of $\boldsymbol{A}$. The smaller the condition number, the faster the convergence [22].

The condition number of a matrix is defined as $\|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\|$ where $\|\boldsymbol{A}\|$ is the matrix L2 norm. For a symmetric matrix, the matrix L2 norm is its spectral radius (the absolute value of its largest eigenvalue). The spectral radius is denoted as $\rho(\boldsymbol{A})$. Hence, the condition number of a matrix is also defined as the absolute value of the largest eigenvalue of the matrix times the absolute value of the largest eigenvalue of the corresponding inverse matrix. This is $\rho(\boldsymbol{A})\rho(\boldsymbol{A}^{-1})$, which is equal to the absolute value of the matrix's largest eigenvalue divided by the absolute value of its smallest eigenvalue, $\lambda_{max}(\boldsymbol{A})/\lambda_{min}(\boldsymbol{A})$. Defined theis way, the condition number is ≥ 1.

Preconditioning

Preconditioning [22] means transforming the matrix equation $\boldsymbol{Ax=b}$ into another matrix equation $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}}$ such that the condition number of $\tilde{\boldsymbol{A}}$ is less than the condition number of $\boldsymbol{A}$, resulting in faster convergence. $\tilde{\boldsymbol{A}}$ is related to $\boldsymbol{A}$ as $\tilde{\boldsymbol{A}} = \boldsymbol{S}^T\boldsymbol{AS}$.

The actual transformation is usually not performed during the iteration because the transformation can destroy the sparsity of $\boldsymbol{A}$. This means that it is not necessary to know S or $\boldsymbol{S}^T$ (the transpose of $\boldsymbol{S}$) explicitly, nor to perform the matrix multiplication $\boldsymbol{S}^T\boldsymbol{AS}$. Instead, the transformation is performed implicitly within the preconditioned conjugate gradient algorithm. This implicit transformation requires that one solve a set of simultaneous linear equations of the form $\boldsymbol{Qx = y}$, where Q is $(\boldsymbol{SS}^T)^{-1}$. The matrix $\boldsymbol{Q}^{-1}=\boldsymbol{SS}^T$ is called the *splitting matrix*. Specifically, the step involves solving $\boldsymbol{Q}\tilde{\boldsymbol{r}}^{(k+1)} = \boldsymbol{r}^{(k+1)}$ for $\tilde{\boldsymbol{r}}^{(k+1)}$ where $\mathbf{r}$ is the residual vector.

Note that $\boldsymbol{Q}^{-1}\boldsymbol{A}$ and $\tilde{\boldsymbol{A}}$ are related by a similarity transformation $\boldsymbol{S}\tilde{\boldsymbol{A}}\boldsymbol{S}^{-1} = \boldsymbol{Q}^{-1}\boldsymbol{A}$ as

follows:

$$\hat{A} = S^T A S$$

$$S\hat{A} = SS^T A S$$

$$S\hat{A}S^{-1} = SS^T A = Q^{-1} A$$

they have equal eigenvalues and, consequently, equal condition numbers. Note that if $\boldsymbol{Q}=\boldsymbol{A}$, $\boldsymbol{Q}^{-1}\boldsymbol{A}$ is the identity matrix, and its condition number is 1. This is the smallest condition number that a matrix can have, and so the best $\boldsymbol{Q}$ one can get is $\boldsymbol{A}$ itself. However, this is not a practical choice because, if $\boldsymbol{Q} = \boldsymbol{A}$, the algorithm requires that one solve the original problem in order to precondition it! Ideally, $\boldsymbol{Q}$ is a matrix that approximates $\boldsymbol{A}$ but is easy to solve. Choosing an optimal $\boldsymbol{Q}$ remains an area of active research.

An example of a simple preconditioning is diagonal scaling where $\boldsymbol{S}$ is a diagonal matrix whose diagonal elements are the inverse square roots of the diagonal elements of $\boldsymbol{A}$.

$$S = diag((\sqrt{a_{11}})^{-1}, \ldots, (\sqrt{a_{nn}})^{-1})$$

Such an $\boldsymbol{S}$ makes the diagonal elements of $\tilde{\boldsymbol{A}}$ equal to 1. The off-diagonal elements are

$$\hat{a}_{ij} = \frac{a_{ij}}{\sqrt{a_{ii}a_{jj}}}$$

However, this preconditioning does not change the condition number if the main diagonal elements of $\boldsymbol{A}$ are identical as in the Poisson or Laplace matrix [15].

Chapter 3 Parallelization Techniques

Introduction

This chapter continues the discussion of solution methods and describes how direct and iterative methods are implemented on parallel machines. The parallel direct method discussed in detail is the multifrontal method developed by Duff and Reid [8]. Parallel implementations of the Jacobi and Gauss-Seidel methods are also discussed. Particular attention is paid to the conjugate gradient method.

Parallelization techniques divide into two categories: data parallelism and control parallelism.

Data Parallelism

Data parallelism is defined as using multiple functional units to apply the same operation simultaneously to a data set [31]. To implement data parallelism on a multicomputer, one first partitions the data among the compute nodes and then runs the same program on each compute node. Each compute node owns its own portion of the data set and has its own copy of the same program. This program runs on each compute node and performs the same operations on the portion of the data set owned by that compute node. Such a program is called an SPMD program where SPMD stands for Single Program, Multiple Data.

The individual copies of the SPMD program need not run in synchronization with each other; but at certain points in the algorithm, synchronization often turns out to be necessary. For this reason, the operating system has calls that provide barrier synchronization. That is, when a compute node reaches such a call in the program, it cannot proceed any further until all compute nodes have reached that point.

For example, in the parallel Jacobi method, the domain is divided among the different compute nodes; each compute node executes the Jacobi algorithm on its portion of the data. Message-passing deals with the portion of the domain on the boundary

between compute nodes.

Other examples of data parallel algorithms are the Gauss-Seidel and conjugate gradient methods.

Control Parallelism

Control parallelism means applying different operations to possibly different data. Control parallelism divides the tasks to be performed while data parallelism divides the data.

A tree search is a good example of control parallelism. Consider the search of a solution tree. Here, the leaves of the tree contain possible solutions to the problem. The solution tree may be quite involved and contain many branches. Control parallelism is implemented by first dividing the solution tree into many more sections than there are compute nodes. Then, one compute node (the manager node) assigns sections to other compute nodes. When a particular compute node finishes its task (follows its portion of the tree down to the leaves), it reports its answer and requests another task from the manager. This method is also sometimes called a manager/worker decomposition.

Parallel Direct Methods

This section describes parallel programs that use direct methods. Techniques that work well for serial programs do not necessarily work well for parallel programs.

Serial Reordering Methods Concentrate on Reducing Fill-In

The reordering methods previously described in "Serial Direct Methods" on page 24 focused on reducing fill-in. Fill reduction is desirable because it lowers operation count and memory requirements. In a parallel setting, reordering is still performed, but the goal is more than fill reduction. In fact, focusing solely on fill reduction may actually reduce opportunities for parallelism. The tridiagonal matrix is an example of a matrix ordering that exhibits no fill-in, yet prevents parallelization.

The Tridiagonal Matrix: No Fill-In, but Requires Serial Elimination

The tridiagonal matrix (shown in Figure 3.1) is an example of how an ordering useful in a serial implementation is inappropriate for a parallel implementation [13].

Recall that eliminating a column means performing matrix operations such that matrix elements below the diagonal are set to zero. Figure 3.1 shows that when column 1 is eliminated, only the diagonal element of column 2 is modified; when column 2 is eliminated, only the diagonal element of column 3 is modified, etc. Hence, a tridiagonal matrix exhibits no fill-in. If fill reduction were one's sole goal, reordering a matrix so that it becomes tridiagonal would be success.

However, the elimination algorithm for a tridiagonal matrix turns out to be a serial one. Because column 2 is modified when column 1 is eliminated, column 2 cannot be eliminated until the elimination of column 1 is complete; because column 3 is modified when column 2 is eliminated, column 3 cannot be eliminated until the elimination of column 2 is complete, etc. The result is that for a tridiagonal matrix the columns must be eliminated in order.

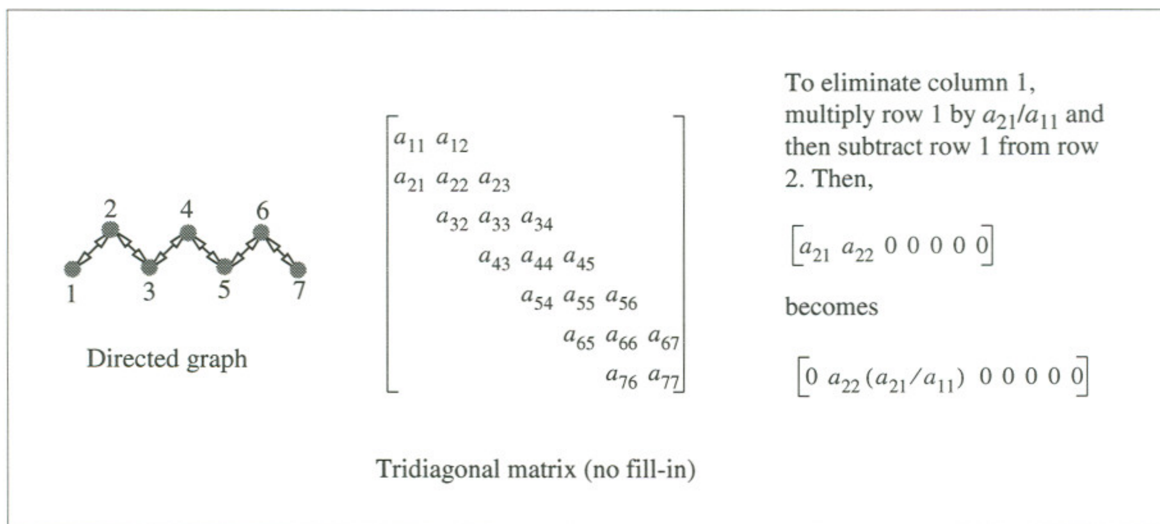


Figure 3.1. Tridiagonal Matrix Exhibiting no Fill-in

The Elimination Tree

The *elimination tree* shows the order in which columns must be eliminated. The elimination tree is defined in terms of tree construction algorithm [10], [13].

Simple Example of an Elimination Tree

The formal definition of an elimination tree has more meaning if one considers a simple example first. The elimination tree for the tridiagonal matrix shown in Figure 3.1 is a linear chain and is shown in Figure 3.2. For this example, each column in the matrix is a

node of the elimination tree. Node 2 is the parent of node 1; node 3 is the parent of node 2, etc.

For good parallelism, the goal is to reorder the matrix so that the elimination tree has more than one node at a level. Nodes at the same level in an elimination tree indicate columns that can be eliminated in parallel. Columns at a lower level must be eliminated before columns at a higher level.

Figure 3.2 contrasts a long, thin elimination tree (the seven-node linear chain) with a short, broad elimination tree (a three-level binary tree). The first level of the second elimination tree consists of nodes 1, 2, 4, and 5; this means that columns 1, 2, 4, and 5 can be eliminated in parallel. The second level consists of nodes 3 and 6; this means that columns 3 and 6 can be eliminated in parallel. Because node 7 is a single column and is the last column, it does not require elimination. Although there is no systematic measure of what makes a good parallel ordering, in a qualitative sense, short, broad elimination trees are superior to tall, narrow ones [13].

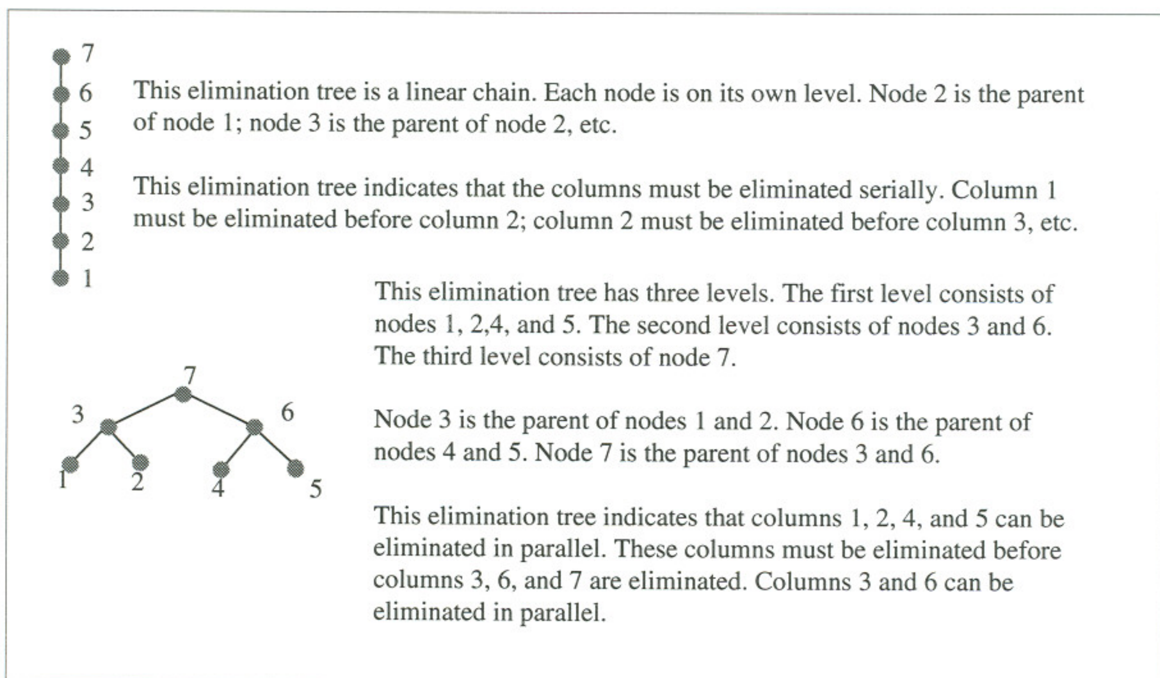


Figure 3.2. Elimination Trees

Figure 3.3 shows how one would reorder the tridiagonal matrix shown in Figure 3.1 so that its elimination tree becomes the three-level binary tree shown in Figure 3.2.

Definition of an Elimination Tree

The elimination tree is defined as a construction algorithm. Before presenting this

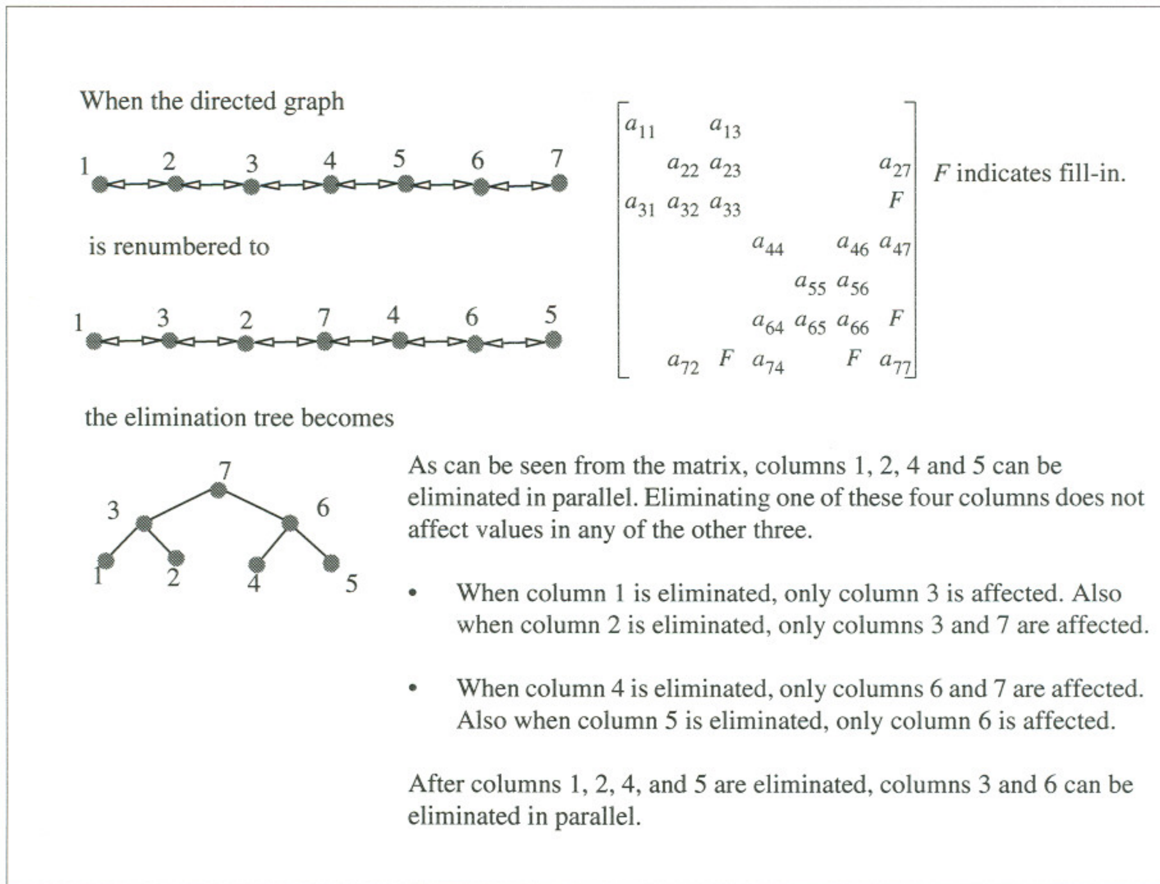


Figure 3.3. How Reordering Affects the Elimination Tree

algorithm, make two assumptions about the matrix $\boldsymbol{A}$. These assumptions are not required, but they make the definition simpler. Assume that the matrix $\boldsymbol{A}$ is symmetric; also, assume that the diagonal elements of $\boldsymbol{A}$ are all acceptable pivots.

Also, define the *sparsity structure* of a matrix. The sparsity structure of a matrix is a record of whether an element is zero or nonzero. It does not record the actual value of the matrix element. To define an elimination tree, one only needs the sparsity structure of $\boldsymbol{L}$, lower triangular part of $\boldsymbol{A}$.

The elimination tree is constructed as follows:

For columns $j = 1$ to n, where n is the number of columns in $\boldsymbol{A}$,

1. Beginning with the diagonal element a_{jj} in the sparsity structure of $\boldsymbol{L}$, go down the column j until the first nonzero matrix element is encountered. Assume that this nonzero matrix element is in row i.
2. Make a connection between column j and the row i.

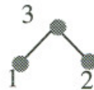
3. Do not actually perform the elimination of column j, but change the sparsity structure of $\boldsymbol{L}$ to how it would look if column j were eliminated. This is to account for fill-in that may occur when column j is eliminated.

Applying the Definition of an Elimination Tree

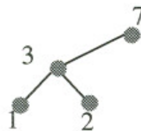
To see how the construction algorithm for an elimination tree works, apply it to the tridiagonal matrix shown in Figure 3.3.

1. Go down column 1. Note that a_{31} is nonzero. Make a connection between nodes 1 and 3.

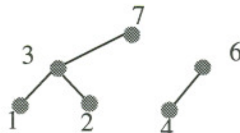
2. Go down column 2. Note a_{32} is nonzero. Make a connection between nodes 2 and 3.



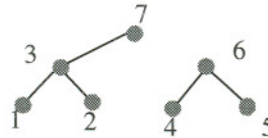
3. Go down column 3, Note that a_{37} is nonzero. This nonzero element is the result of fill-in that occurred when column 2 was eliminated. Make a connection between nodes 3 and 7.



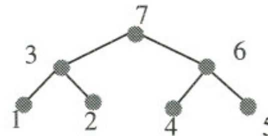
4. Go down column 4, Note that a_{64} is nonzero. Make a connection between nodes 4 and 6.



5. Go down column 5, Note that a_{65} is nonzero. Make a connection between nodes 5 and 6.



6. Go down column 6, Note that a_{67} is nonzero. This nonzero element is the result of fill-in that occurred when column 4 was eliminated. Make a connection between nodes 6 and 7.



The Multifrontal Method

The multifrontal method is a generalization of the frontal method introduced in the "Frontal Method" on page 28.

To recap, the frontal method recognizes that one need not access the entire matrix to factor a portion of it. The frontal method brings into memory only the portion of the matrix that is needed to factor the column currently being worked on. This method is useful when solving large matrices that do not fit into memory. It's called the frontal method because, with some imagination, one can think of the portion of the matrix that must be in memory as a wavefront that moves through the matrix.

The multifrontal method is just like the frontal method except that there are several fronts. The matrix is reordered so that different columns can be worked on independently. When the multifrontal method is implemented on parallel MIMD processors, a separate front is assigned to each processor.

As an example, once again consider a tridiagonal matrix reordered so that its elimination tree is the three-level binary tree shown in Figure 3.3. As shown in Figure 3.3, columns 1 2, 4, and 5 can be eliminated in parallel. At this stage there are four fronts in the matrix. The second stage is to eliminate the two columns 3 and 6; this second stage has two fronts.

Nested Dissection

Nested dissection is a method of ordering a directed graph so that the multifrontal method can be used.

For example, once again consider a linear directed graph. Figure 3.4 shows how this directed graph is dissected and numbered. The first dissection divides the graph into two regions, each containing three nodes. The dividing region contains one node. The second dissection divides each of the divided regions into two regions. The new dividing regions each contain one node.

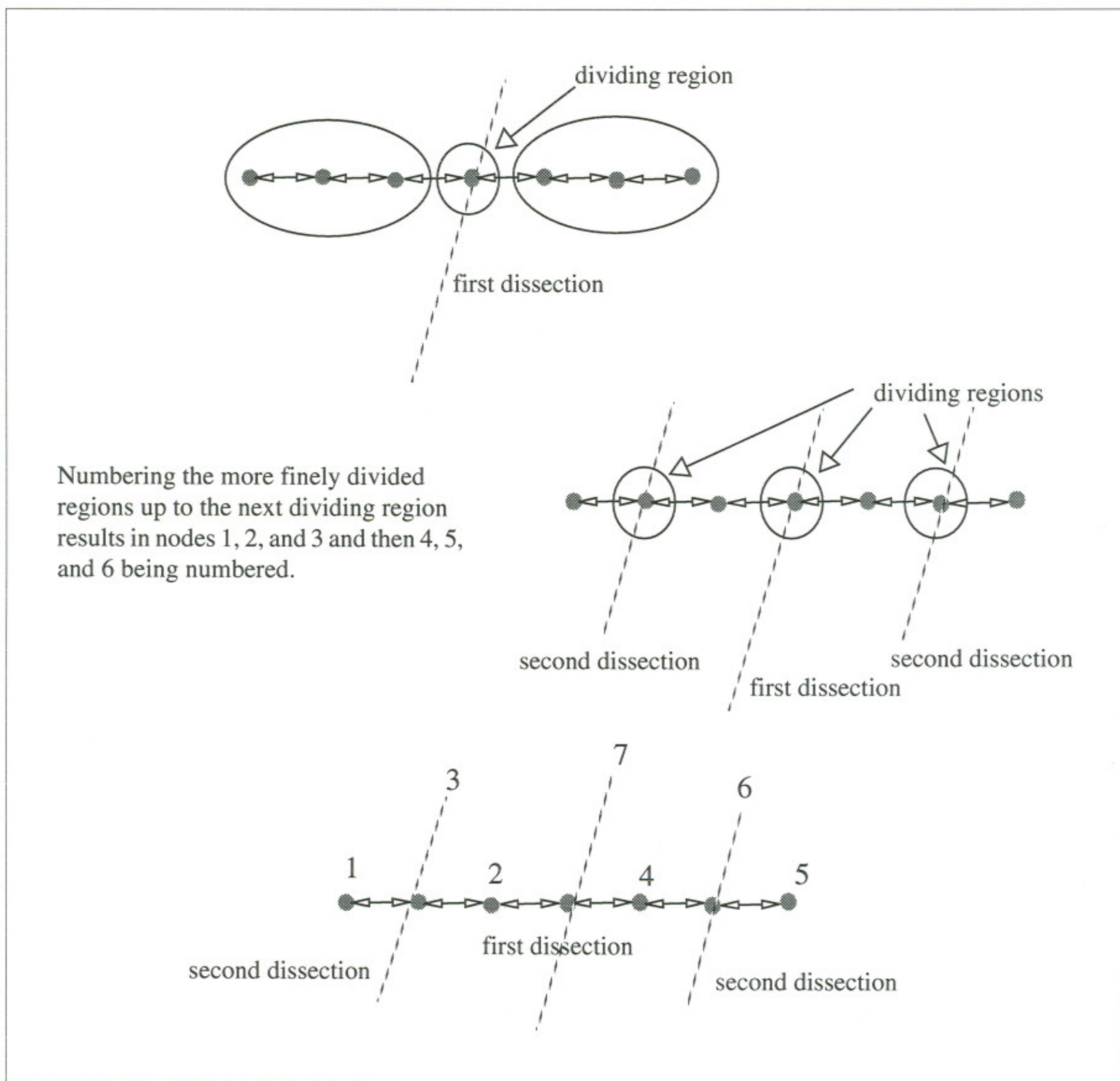


Figure 3.4. Nested Dissection of a Linear Chain

When the graph is numbered, the most finely divided regions up to the next dividing region are numbered first. Then, the next level of dividing regions are numbered,

followed by the next level, etc. In Figure 3.4, the most finely divided regions are nodes 1 and 2; the next dividing region is node 3. Then, again, at the same level, the most finely divided regions are nodes 4 and 5; the next dividing region is node 6. The next level is the remaining node, node 7.

Figure 3.5 shows a more realistic two-dimensional directed graph divided into seven regions. When one divides one region by another, one must ensure that no connection exists between the divided regions. For example, in Figure 3.5, regions 1 and 2 are divided by region 3, and there are no connections between nodes in region 1 with nodes in region 2.

Now assume a directed graph in a square region. Also assume that the connections are local; that is, they do not extend across a dividing region. For example, there is no connection between regions 1 and 4 because these regions are divided by region 7.

Dividing the graph into seven regions results in the matrix on the right. The most finely divided regions (1, 2, 4, and 5) are along the diagonal. The interface regions are pushed down and to the right. The higher up in the nesting hierarchy the region is, the further down and the further right its matrix elements are.

Region 1 has cross-terms with regions 3 and 7.

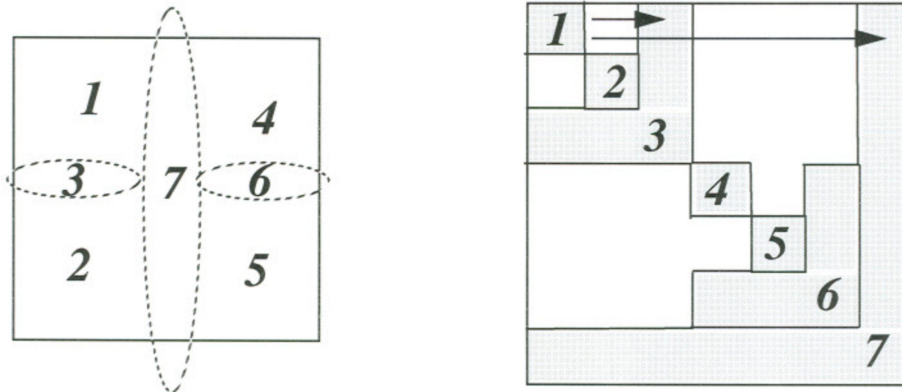


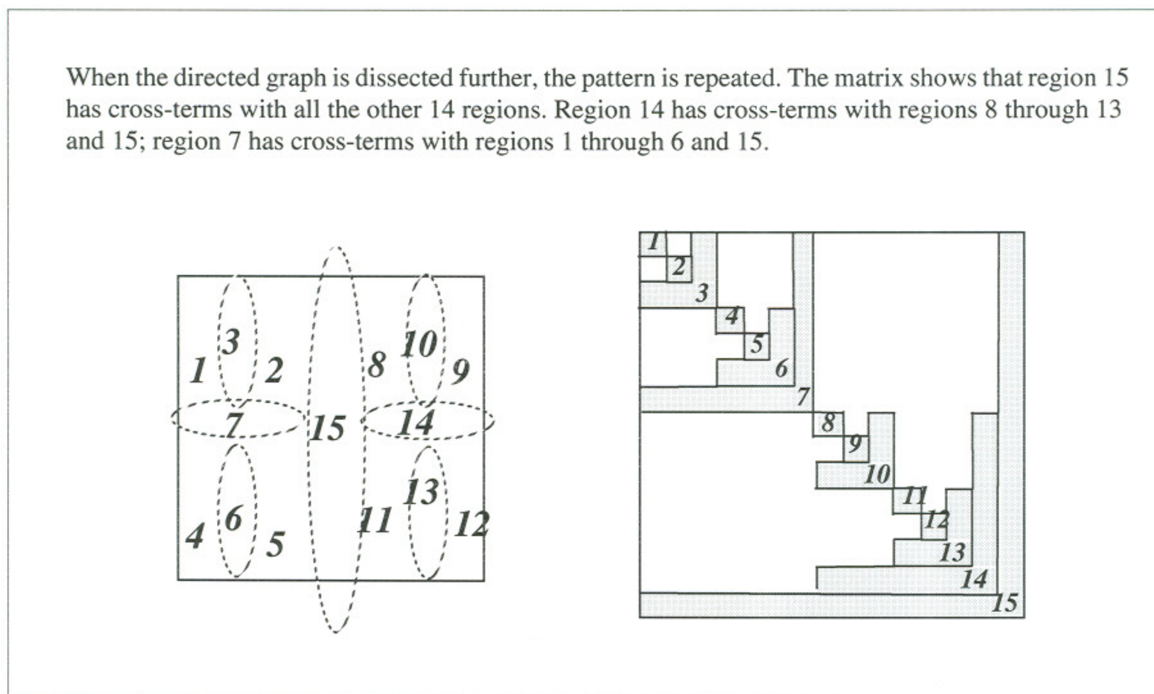
Figure 3.5. Nested Dissection of a Two-Dimensional Directed Graph

Also note that in Figure 3.4, the most finely divided region was a single node in the directed graph; this was the same as a single column in the matrix. In Figure 3.5, the dissection is not necessarily so fine. Each dissected region may actually contain several directed graph nodes. When that occurs the regions are referred to as supernodes [13].

The nodes within a supernode are numbered consecutively. That is, if the nodes in Figure 3.5 are supernodes, then the directed graph nodes in region 1 are numbered first, then the directed graph nodes in region 2, etc. When the multifrontal method is applied to

the matrix in Figure 3.5, it begins with four fronts, then goes to two.

One can continue the dissection as shown in Figure 3.6. With very large matrices, one can stop dissecting when the number of most finely divided regions is at least as large as the number of compute nodes. When the multifrontal method is applied to the matrix in Figure 3.6, it begins with eight fronts, then goes to four, then two.



When the directed graph is dissected further, the pattern is repeated. The matrix shows that region 15 has cross-terms with all the other 14 regions. Region 14 has cross-terms with regions 8 through 13 and 15; region 7 has cross-terms with regions 1 through 6 and 15.

Figure 3.6. Further Nested Dissection of a Two-Dimensional Directed Graph

Parallel Multifrontal Method

With finite element problems, one can dissect and renumber the finite element mesh itself and not have to form the directed graph. This is because finite element problems are local. That is, if there is an interaction between finite element nodes i and j (shown as a_{ij}), then finite element nodes i and j are also physically close to each other in the mesh.

Dissecting the finite element mesh is referred to as *substructuring* [21]. Nesting dissections (as in Figures 5 and 6) is an extension to substructuring that prepares the matrix for multifrontal techniques.

When the matrix is ordered as in Figures 5 and 6, the supernodes are also matrix blocks. Because the blocks at the bottom of the elimination tree describe interior portions of the directed graph while the upper blocks describe interfaces, the bulk of the factorization work occurs at the bottom of the tree where the parallelism is greatest.

For example, if a two-dimensional finite element mesh is divided into seven regions as in Figure 3.5, the matrix appears in block form as follows:

$$\begin{bmatrix} A_{11} & & A_{13} & & & & A_{17} \\ & A_{22} & A_{23} & & & & A_{27} \\ A_{31} & A_{32} & A_{33} & & & & A_{37} \\ & & & A_{44} & & A_{46} & A_{47} \\ & & & & A_{55} & A_{56} & A_{57} \\ & & & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{bmatrix}$$

The elimination tree looks like the three-level binary tree discussed previously and shown in Figure 3.3. The matrix shows that the columns in blocks A_{11}, A_{22}, A_{44}, and A_{55} can be eliminated in parallel.

As the columns in A_{11} are being eliminated, the columns in A_{13} and A_{17} are being modified. As the columns in A_{22} are being eliminated, the columns in A_{23} and A_{27} are being modified. This means that blocks A_{11} and A_{22} must be eliminated before the 3-7 blocks (those blocks with 3 or 7 as one of their indices). And the compute node eliminating those 3-7 blocks must receive information from the compute nodes eliminating the columns in A_{11} and A_{22}. Similarly, blocks A_{44} and A_{55} must be eliminated before blocks with 6 or 7 as one of their indices. The elimination is seen to follow the structure of the elimination tree rising toward its root.

Parallel Iterative Methods

Iterative methods are characterized by the following procedure. One begins with an initial guess for the solution $\boldsymbol{x}$. At each iteration step one calculates another value for $\boldsymbol{x}$ based on the old value. The more iteration steps one performs, the more accurate $\boldsymbol{x}$ becomes.

Data parallelism (see “Data Parallelism” on page 38) can be introduced when the new $\boldsymbol{x}$ is calculated. The vector $\boldsymbol{x}$ is divided up among the compute nodes, each compute node becoming responsible for a certain number of elements of $\boldsymbol{x}$. Each compute node calculates a new $\boldsymbol{x}$ from the old $\boldsymbol{x}$ for the elements of $\boldsymbol{x}$ that it maintains. When doing this the compute node may need some elements of $\boldsymbol{x}$ from other compute nodes, and this comprises a communication cost.

Parallel Jacobi Method

Recall the Jacobi method, introduced in the "Serial Jacobi Method" on page 31. As shown in that section, the calculation that one wants to distribute among the compute nodes is

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

Because the right-hand side of the equation is not modified during the iteration, the elements of $\boldsymbol{x}$ can be updated in parallel.

The matrix element a_{ij} quantifies how much of x_j (from the kth iterate) is present in x_i (for the $(k+1)^{\text{th}}$ iterate). If the matrix is completely dense (completely dense means that the matrix has no zero elements), then to form the $(k+1)^{\text{th}}$ iterate of $\boldsymbol{x}$, one must have access to all the elements of $\boldsymbol{x}$.

If $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U}$ where $\boldsymbol{L}$ is lower triangular with zeros on the diagonal, $\boldsymbol{U}$ is upper diagonal with zeros on the diagonal, and $\boldsymbol{D}$ is diagonal, then $\boldsymbol{x}^{(k+1)} = -\boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})\boldsymbol{x}^{(k)} + \boldsymbol{D}^{-1}\boldsymbol{b}$. Note that each step of the Jacobi iteration consists essentially of a matrix-vector multiplication—when $\boldsymbol{b}$ is zero, it is exactly a matrix-vector multiplication.

Consider a row decomposition of the iteration matrix, which is shown in Figure 3.7. In a row decomposition each compute node contains a set of contiguous rows of the matrix and corresponding elements of $\boldsymbol{b}$. Because the figure assumes the matrix is dense, it also shows $\boldsymbol{x}$ duplicated on each compute node. With a row decomposition, a matrix-vector multiplication modifies only a portion of the vector. The operation updates only those vector elements whose index is the same as one of the rows of the matrix the compute node contains; but because the matrix is dense, the compute node needs the entire $\boldsymbol{x}$ vector to perform its update. The Jacobi method requires that this operation be performed at each step in the iteration. In preparation for the next step, each compute node must broadcast its updated portion of $\boldsymbol{x}$ to all the other compute nodes.

In summary, each compute node must contain enough elements of $\boldsymbol{x}$ to calculate $\boldsymbol{x}^{(k+1)}$. When the matrix has no nonzero elements, this means each compute node must contain all the elements of $\boldsymbol{x}$. When the matrix has nonzero elements, it means that if a compute node is updating x_i and a_{ij} is nonzero, then the compute node must also have access to x_j.

Having access to an element of $\boldsymbol{x}$ means being able to read it, not necessarily to write it. In the dense matrix example, a compute node has to be able to read all of $\boldsymbol{x}$, but

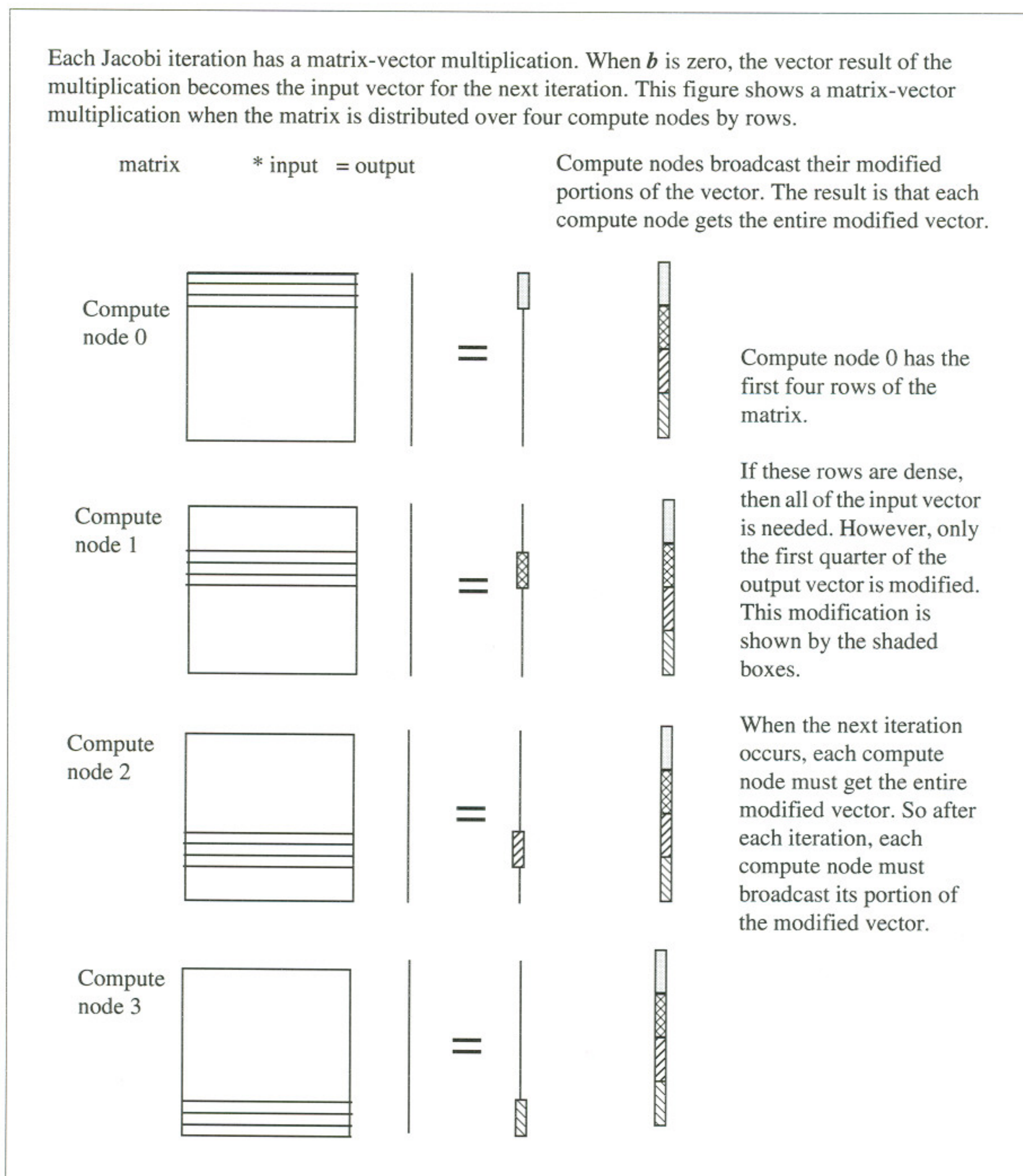


Figure 3.7. Row Decomposition for Jacobi Iteration

the compute node need only write those elements of x whose indices are the same as the indices of the rows the compute node was assigned.

The steps of the Jacobi iteration are as follows:

1. Guess an initial value of x.
2. Perform the operation, modifying certain elements of x.
3. Send the updated elements of x to those compute nodes that need them

4. Receive the updated elements of x needed for the next iteration.

Parallel Gauss-Seidel Method

Recall the Gauss-Seidel method, introduced in the "Serial Gauss-Seidel Method" on page 32. As shown in that section, the calculation that one wants to distribute among the compute nodes is

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

Because the right-hand side of the equation is modified during the iteration, the method has a significant serial component. To calculate $x_i^{(k+1)}$, it is necessary to calculate all the values $x_j^{(k+1)}$ where j is less than i.

For example, consider the row decomposition discussed previously in the section "Parallel Jacobi Method" on page 49. As in that section, each compute node modifies only a portion of the vector $\boldsymbol{x}$.

However, the operation on a particular compute node is serialized. For example, if compute node 0 updates elements 1 through i, then x_1 must be updated before x_2, which must be updated before x_3, etc. Finally, x_{i-1} must be updated before x_i.

In addition, the operation across the compute nodes is also serialized. For example, if compute node 1 updates elements i+1 through j, then x_{i+1} cannot be updated until all the values x_1 though x_i are updated on compute node 0 and communicated to compute node 1. Similarly, compute node 2 cannot start updating its values until compute node 1 has completed; compute node 3 cannot start until compute node 2 has completed, etc.

For most problems of physical interest, however, parallelization of the Gauss-Seidel method is possible. This is because physical problems often derive from a partial differential equation. A vector governed by such an equation has local interactions, and the resulting matrix is sparse.

The next section presents Laplace's equation as an example of a partial differential equation to be solved by the Jacobi or Gauss-Seidel methods. When this differential equation is written as a matrix equation $\boldsymbol{Ax} = \boldsymbol{b}$, the matrix $\boldsymbol{A}$ is sparse and has the form shown in Figure 1.4 on page 8.

Example of a Parallel Laplace Solver

Recall the two-dimensional Laplace's equation introduced in "Laplace's Equation"

on page 5

$$\nabla^2 \phi = \frac{\partial^2}{\partial x^2}\phi + \frac{\partial^2}{\partial y^2}\phi = 0$$

The finite difference approximation (as shown in "Finite Difference Grid and the Corresponding Matrix Equation" on page 6) is

$$\phi(x, y) = \frac{1}{4}(\phi(x, y+\Delta) + \phi(x-\Delta, y) + \phi(x+\Delta, y) + \phi(x, y-\Delta))$$

This equation shows how the ϕ at each grid point is related to the ϕ's at the neighboring grid points. There are four neighboring grid points. When the discretized Laplace's equation is written as a matrix equation $A\phi = 0$, each row of A contains only five nonzero elements, including the diagonal.

The above equation can be used to calculate the value of ϕ for each iteration cycle. Evaluating the equation is the same as multiplying ϕ by the iteration matrix. Whether one is using the Jacobi or the Gauss-Seidel method depends on which iteration cycle the ϕ values are chosen from.

When the Jacobi method is used to calculate the value of ϕ for the $(k+1)^{\text{th}}$ iteration, the values of $\phi(x,y)$ on the right-hand side are those for the kth iteration.

$$\phi^{(k+1)}(x, y) = \frac{1}{4}(\phi^{(k)}(x, y+\Delta) + \phi^{(k)}(x-\Delta, y) + \phi^{(k)}(x+\Delta, y) + \phi^{(k)}(x, y-\Delta))$$

Recall that the Gauss-Seidel method requires that the new values of ϕ are used as soon as they are available. Which grid points these new values are associated with depends on the order of update. When the grid points are numbered and updated consecutively right to left, top to bottom, the equation for the Gauss-Seidel method becomes

$$\phi^{(k+1)}(x, y) = \frac{1}{4}(\phi^{(k+1)}(x, y+\Delta) + \phi^{(k+1)}(x-\Delta, y) + \phi^{(k)}(x+\Delta, y) + \phi^{(k)}(x, y-\Delta))$$

The Parallel Jacobi Method and Laplace's Equation

The example discussed in this section is for a multicomputer with physically distributed memory and no shared address space.

The first step is to divide the grid up among the compute nodes. Because the iteration matrix for Laplace's equation connects only neighboring grid points, communication among compute nodes need only occur along the boundary of the

division. Figure 3.8 shows two possible grid decompositions. Note that the vertical

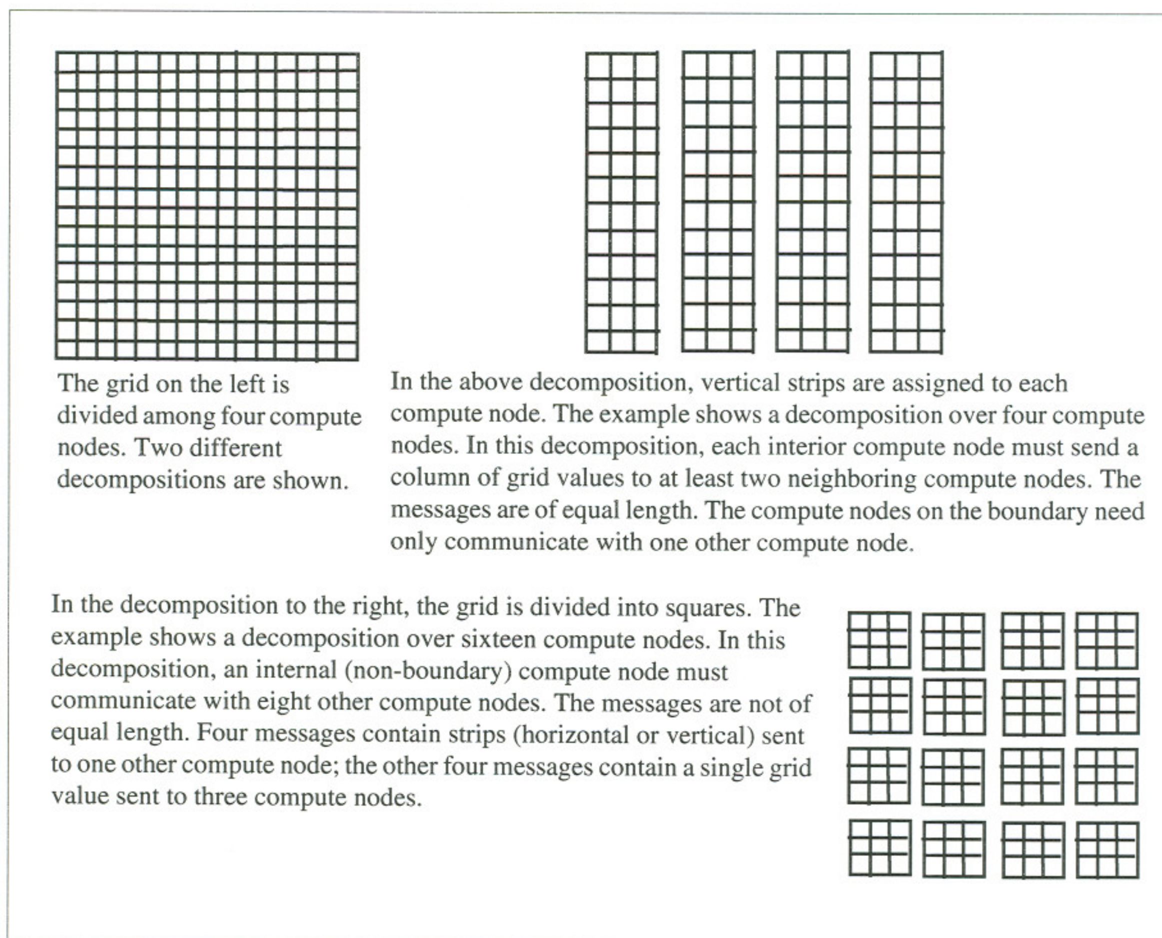


Figure 3.8. Grid Decomposition

decomposition results in fewer messages of longer length than the square decomposition. A small number of long messages is generally a desirable feature in a multicomputer communication pattern. However, as the grid gets very large and the messages get excessively long, a square decomposition is more appropriate.

The values needed from the neighboring compute nodes are kept in a guard buffer (sometimes called a shadow buffer). Figure 3.10 shows a vertical decomposition with guard buffers. The filled-in circles show the grid values that are updated by the compute node. The hollow circles are the values from the neighboring compute nodes that are needed to perform the update. Before the calculation is performed in each iteration cycle, the compute node receives values from the boundaries of adjacent compute nodes into its guard buffer and sends the values of its own boundaries to the guard buffers of its neighboring compute nodes. The arrows in Figure 3.10 indicate the direction of these messages.

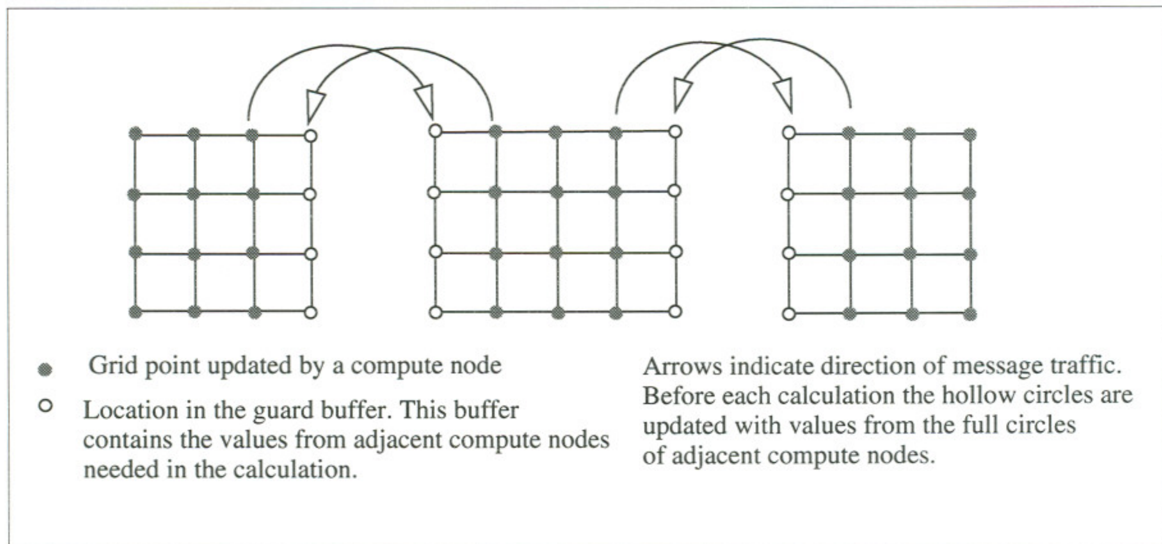


Figure 3.9. Grid Used for Parallel Gauss-Seidel Implementation.

The Parallel Gauss-Seidel Method and Laplace's Equation

The section "Parallel Gauss-Seidel Method" on page 51 discussed how the requirement that each element of $\boldsymbol{x}$ is updated using updated elements of $\boldsymbol{x}$ as soon as they are available introduces a serial component to the calculation that makes the algorithm difficult to parallelize. In a serial implementation, the requirement can be simply satisfied by updating the vector $\boldsymbol{x}$ in place.

In a parallel implementation, one approach is to update $\boldsymbol{x}$ in place for each iteration using the values that are locally available. Then, as in the Jacobi method, at the end of each iteration, each compute node must send its updated $\boldsymbol{x}$ elements to the compute nodes that need these values for the next iteration. This approach, however, does not exactly follow the rules. When one compute node is using a value from its guard buffer to update one of its $\boldsymbol{x}$ values, that value may already have been updated on the compute node it came from, but the new value has not yet made it into the guard buffer.

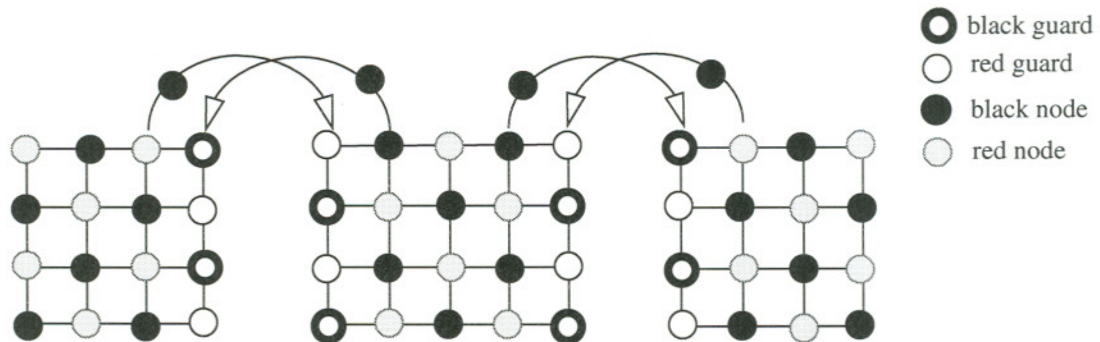
The Gauss-Seidel rules can be more strictly followed by modifying the update order. For example, consider checkerboard ordering (also called red-black ordering) [37], [12]. This update ordering is illustrated in Figure 3.10.

Here the grid is divided into two kinds of grid points, often called red and black. The pattern is like that of a checkerboard. The red values are updated using only black values, and the black values are updated using only red values. So when the compute nodes are updating their red values, they do not have to concerned about a black value

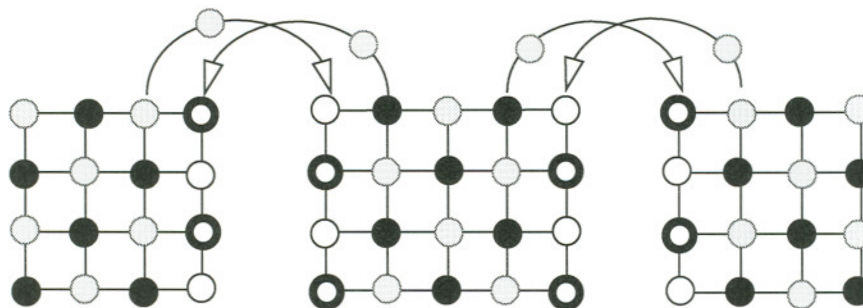
The darkly shaded grid points (black ones)are updated using only values from the lightly shaded grid points (red ones).

The Gauss-Seidel rules can be observed by checkerboard ordering. An iteration step is as follows:

1. Communicate the darkly shaded (black) values on the boundary.



2. Update the lightly shaded (red) values. These grid points use only the darkly shaded (black) values.
3. Communicate the lightly shaded (red) values on the boundary.



4. Update the darkly shaded (black) values. These grid points use only the lightly shaded (red) values.

Figure 3.10. Checkerboard Ordering

changing on another compute node

Parallel Conjugate Gradient Method

This section describes how the operations of the conjugate gradient method are performed on a multicomputer. The conjugate gradient method was introduced in the section "Non-Stationary Methods" on page 34. The detailed steps of the method are described in Appendix C, "The Conjugate Gradient Method."

As with the Gauss-Seidel method, the domain is a two-dimensional grid. The

points on the grid are partitioned among the compute nodes and indexed locally. A global index would be one that is unique over the set of compute nodes participating in the calculation. Local indexing allows one to modify the mesh without having to communicate among the compute nodes. For example, one may wish to refine the finite element mesh. Refining the mesh means dividing the domain up into smaller finite elements. With a local indexing scheme, one can refine the mesh independently on each compute node.

Unlike the Gauss-Seidel method, the domain is typically unstructured. The difference between structured and unstructured grids was previously described in "Structured and Unstructured Grids" on page 4.

Even though the grid is unstructured, the nonzero pattern of the resulting matrix is not random. Recall that a matrix element designates an interaction between grid points. This interaction is local; that is, only grid points physically close to each other interact and contribute to nonzero matrix elements. If whether a matrix element is nonzero is a random decision, the matrix is called unstructured.

Figure 3.11 shows a directed graph, a grid, or a finite element mesh. For the purposes of this discussion, they are the same. The graph is assumed to be for a symmetric matrix, meaning that all connecting lines should have arrows at both ends; but since this is a global characteristic, it is omitted from the figure.

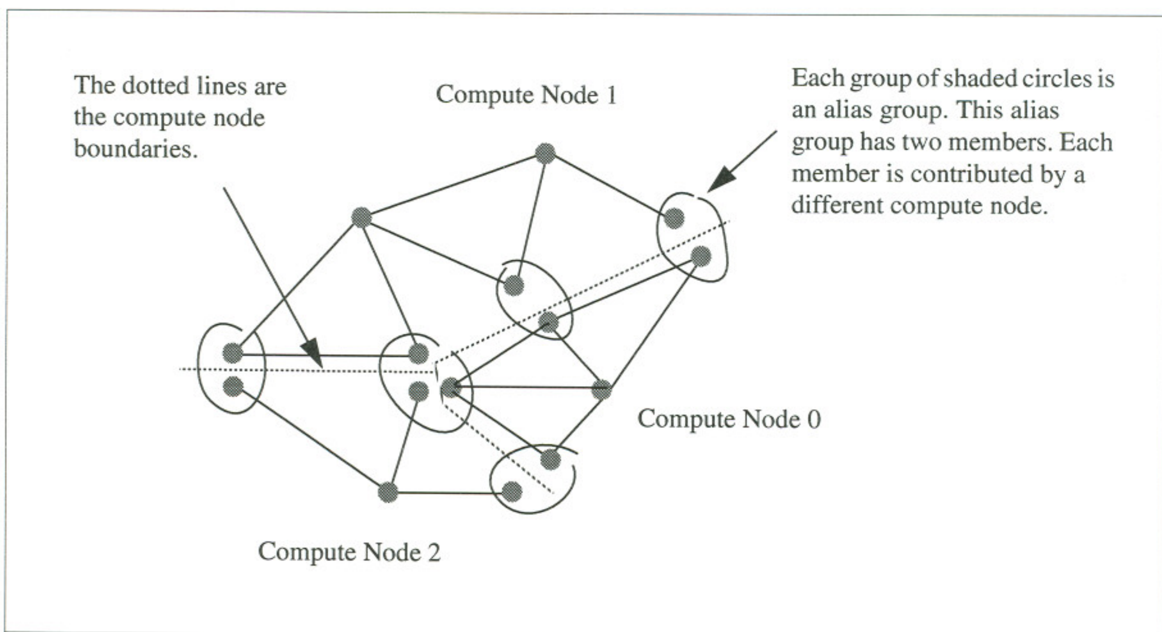


Figure 3.11. Finite Element Mesh Partitioned Among Compute Nodes

Figure 3.11 also illustrates how the matrix elements and the vector components are

distributed among the compute nodes. The matrix element is designated by a connecting line. Note that along the compute node boundary, this line exists on only one side. This means that the matrix element is stored on one compute node and not the other. Which compute node has the matrix element is an arbitrary decision. This decision is discussed in later chapters.

A vector component is associated with each grid point. If the grid has several vectors, each grid point has several vector components, one from each vector. Unlike the matrix elements, the vector components on the compute node boundaries are duplicated on the compute nodes sharing the boundaries.

Recall the definition of an alias group from "The Aliasing Method" on page 18. There, an alias group is defined as a set of copies of one directed graph node (including its associated vector components), each copy residing on a different compute node. Figure 3.11 shows five alias groups.

As previously discussed in "The Aliasing Method" on page 18, an alias group acts like weakly coherent, shared memory. A vector component that is aliased has several copies, one on each compute node. As a compute node performs its calculations, it uses its copy of the vector component, reading and modifying it as necessary. These copies potentially attain different values; this is the same as saying that the alias group may become incoherent. At certain points in the calculation, it becomes necessary for the members of an alias group to once again have the same value; this is the same as saying that the alias group must be made coherent.

As previously mentioned in "The Aliasing Method" on page 18, each iterative step of the conjugate gradient method has one matrix-vector multiplication and two dot products. A parallelization of the conjugate gradient method entails parallelizing these operations.

Matrix-Vector Multiplication

Think of the matrix-vector multiplication as a set of pull-ins. Recall from "The Aliasing Method" on page 18 that a pull-in is defined as the contribution one vector component obtains from another vector component, weighted by the connecting matrix element.

Figure 3.12 shows how the pull-ins are distributed among the compute nodes. The indicated alias group has three members. Each member has its own set of pull-ins on its own compute node.

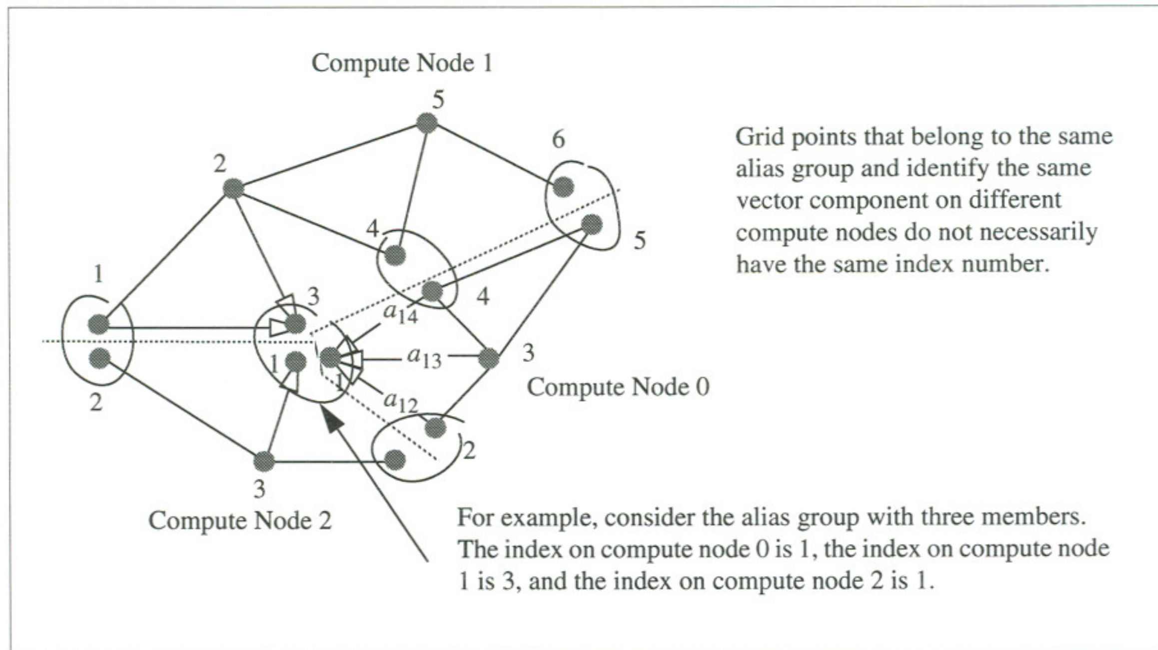


Figure 3.12. Pull-ins Distributed among Alias Group Members

Each compute node performs its pull-ins independently of the other compute nodes. For example, consider the matrix-vector multiplication ***y=Ax*** and the alias group indicated in Figure 3.12. This alias group consists of three members, y_1 on compute node 0, y_3 on compute node 1, and y_1 on compute node 2.

When the matrix-vector multiplication is performed, y_1 on compute node 0 becomes $y_1 = a_{11}x_1 + a_{12}x_2 + a_{13}x_3$. The term $a_{11}x_1$ is the self-term, which comes from the diagonal element. The term $a_{12}x_2$ designates the pull-in along the line connecting grid points 1 and 2. The term $a_{13}x_3$ designates the pull-in along the line connecting grid points1 and 3. In addition, y_3 on compute node 1 (which is really the same vector component as y_1 on compute node 0) becomes $y_3 = a_{32}x_2 + a_{31}x_1$; and y_1 on compute node 2 (which is also the same vector component as y_1 on compute node 0) becomes $y_1 = a_{13}x_3$. After the matrix-vector multiplication, the members of this alias group are incoherent. Each member of the alias group has a different value, which is a portion of final value for that vector element.

The task then remains to make the alias group coherent. This is performed by a *coherence operation*. The coherence operation needed in this case is a routine that sums up the values in an alias group and then replaces the value of each member of the alias group with that sum. The coherence operation is specifically requested by the user program.

Dot Product

The dot product of two n-dimensional vectors is defined as

$$x \cdot y = \sum_{i=1}^{n} x_i y_i$$

Note that the dot product is a scalar value. When the vector is distributed over compute nodes, the dot product is global over the entire set of participating compute nodes.

Once again, consider Figure 3.12. This figure shows nine grid points, five of which are aliased. Assume that each grid point contains two vector elements; that is, point i contains x_i and y_i. Compute node 0 calculates $x_1y_1+x_2y_2+x_3y_3+x_4y_4+x_5y_5$; compute node 1 calculates $x_1y_1+x_2y_2+x_3y_3+x_4y_4+x_5y_5+x_6y_6$; and compute node 2 calculates $x_1y_1+x_2y_2+x_3y_3$.

Because the dot product is a global value, the next step is logically to sum the contributions from each compute node, but when that is done, the aliased terms in the sum are duplicated. For example, x_1y_1 on compute node 0 is really the same term as x_3y_3 on compute node 1. This problem is avoided if when the compute node perform their sums, only one term from an alias group contributes. This is achieved by designating, for each alias group, one compute node as the *secretary.*

If, in Figure 3.12, the compute node with the lowest node number is designated the secretary, then compute node 0 includes x_1y_1 in its partial sum; but compute node 1 does not include its x_3y_3, and compute node 2 does not include its x_1y_1.

So a secretary is defined for each alias group. Each alias group is a set of directed nodes, each from a different compute node. One of those compute nodes is designated the secretary. How the designation is made is arbitrary. In the example presented in this section, the secretary for each alias group is the compute node with the lowest compute node number, chosen from the compute nodes contributing a directed graph node to the alias group. For some problems, (as will be seen in Chapter 6, "A Random Matrix") choosing the secretary this way is unbalenced. Becasue the secretary performs additional work, one should attempt to evenly distribute the secretary designation among the compute nodes.

The Owner Computes Method

In the *owner-computes* method, each compute node contains the matrix elements it needs to perform its portion of the matrix vector multiplication. The compute node does not contain the value of the vector component on the other end of the pull-in and must

obtain that value in a message from the owner of that vector component.

Here is the reason why the method is called owner computes. Consider the matrix-vector multiplication $\boldsymbol{y}=\boldsymbol{Ax}$. If the vector component y_i resides on a compute node (that compute node is called the owner of y_i), then that compute node calculates all the terms in y_i. In contrast, in the aliasing method, portions of y_i are calculated by all the compute nodes belonging to the alias group for y_i, followed by communication to make y_i coherent.

Figure 3.13 shows the same mesh as that in Figure 3.12, as it would appear for an

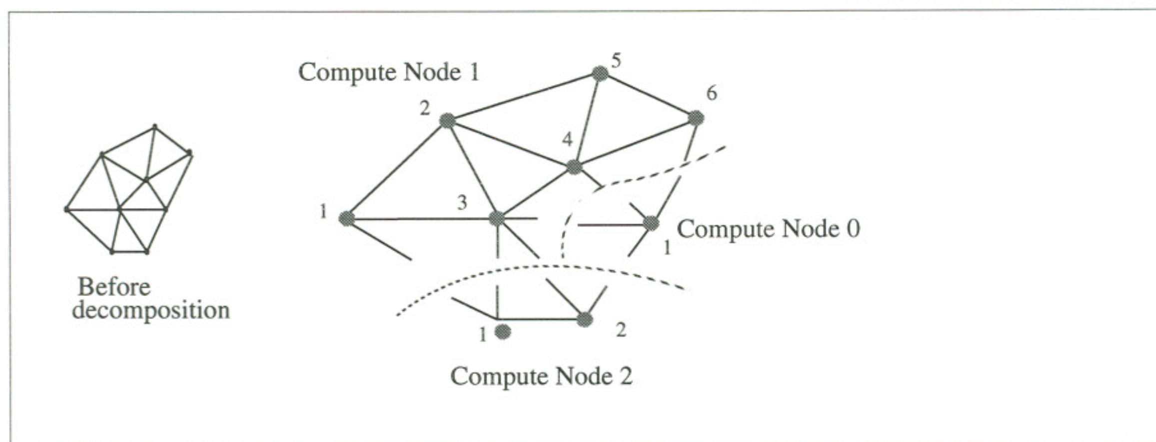


Figure 3.13. Dividing the Mesh Among Compute Nodes (Owner Computes)

owner-computes computation. Note that the vector components are not aliased. For example, when compute node 2 calculates y_1, it needs three matrix elements. These are shown as lines connected to the shaded circle labeled 1. One of those lines is connected to the shaded circle labeled 2, also on compute node 2. This pull-in requires no communication. The other two lines are connected to shaded circles on compute node 1. Pull-ins along these lines require communication with compute node 1. Compute node 2 must receive the values y_1 and y_3 from compute node 1.

Comparison of Direct and Iterative Methods

The differences between direct and iterative methods become most apparent for large problems. With large problems, the fill-in resulting from direct methods increases greatly (the matrix becomes essentially dense) [34]; and the number of operations becomes prohibitively large. Iterative methods, however, do not change the matrix once it has been constructed; the number of operations in each iterative step remains the same.

Round-off characteristics are different as well. With direct methods, roundoff

errors accumulate. The more singular the matrix is, the more serious are the roundoff errors. With iterative methods, preconditioning techniques can be used to reduce roundoff error [30].

The parallelization properties of direct and iterative methods are different as well. Parallel direct methods more closely resemble control parallelism than data parallelism. Although data are partitioned among the compute nodes as in data parallelism, the procedures of rising in an elimination tree and keeping track of what compute nodes are participating at what level are control operations. Parallel iterative methods exhibit straightforward data parallelism.

Another important difference is simply the commitment on the part of the user. When a user chooses a direct method, the user commits to see the process to its completion. Unless the program completes, there is no answer; it's all or nothing. However, with an iterative method, each step advances the current value closer to the solution. A user can choose to perform the amount of work required to achieve the desired level of accuracy.

Chapter 4 Implementation of Aliasing

Introduction

This chapter describes how a grid is partitioned among the compute nodes, how alias groups arise from that partitioning, and how alias groups are made coherent. The chapter presents an explicit example of a finite element triangular grid. This example takes the finite element grid, divides it among four compute nodes, and shows how alias groups develop. Alias groups were first defined in "The Aliasing Method" on page 18; they were further discussed in "Parallel Conjugate Gradient Method" on page 55.

Grid Partitioning

A grid is partitioned by assigning a portion of the grid points to each compute node. If the grid is constructed beforehand, this assignment can be supplied as part of an input file. If the grid is developed by the parallel program, each compute node can appropriate a portion of the grid and assign itself grid points as they are created.

This section presents a simple example of a grid and shows how it appears when partitioned over four compute nodes.

Figure 4.1 shows an 11x11 grid being cut equally into quarters. The cut goes through a line of grid points. When the cut line encounters a grid point, the grid point splits into two, creating alias groups. Each alias group contains two grid points. Where the cut lines meet (in the center), the grid point is split into four; and that alias group contains four grid points. Figure 4.2 shows the grid after the alias groups have been created.

Hereafter, the term *field node* is used to mean either a grid point or a finite element node with all its associated vector components. When a vector is defined on the grid, a component of the vector is associated with each field node, and there are as many components as there are field nodes. The field nodes are uniquely identified by (x,y) coordinates. The grid coordinates are in the range 0 through 1, making the corners of the grid (0,0), (0,1), (1,0), and (1,1).

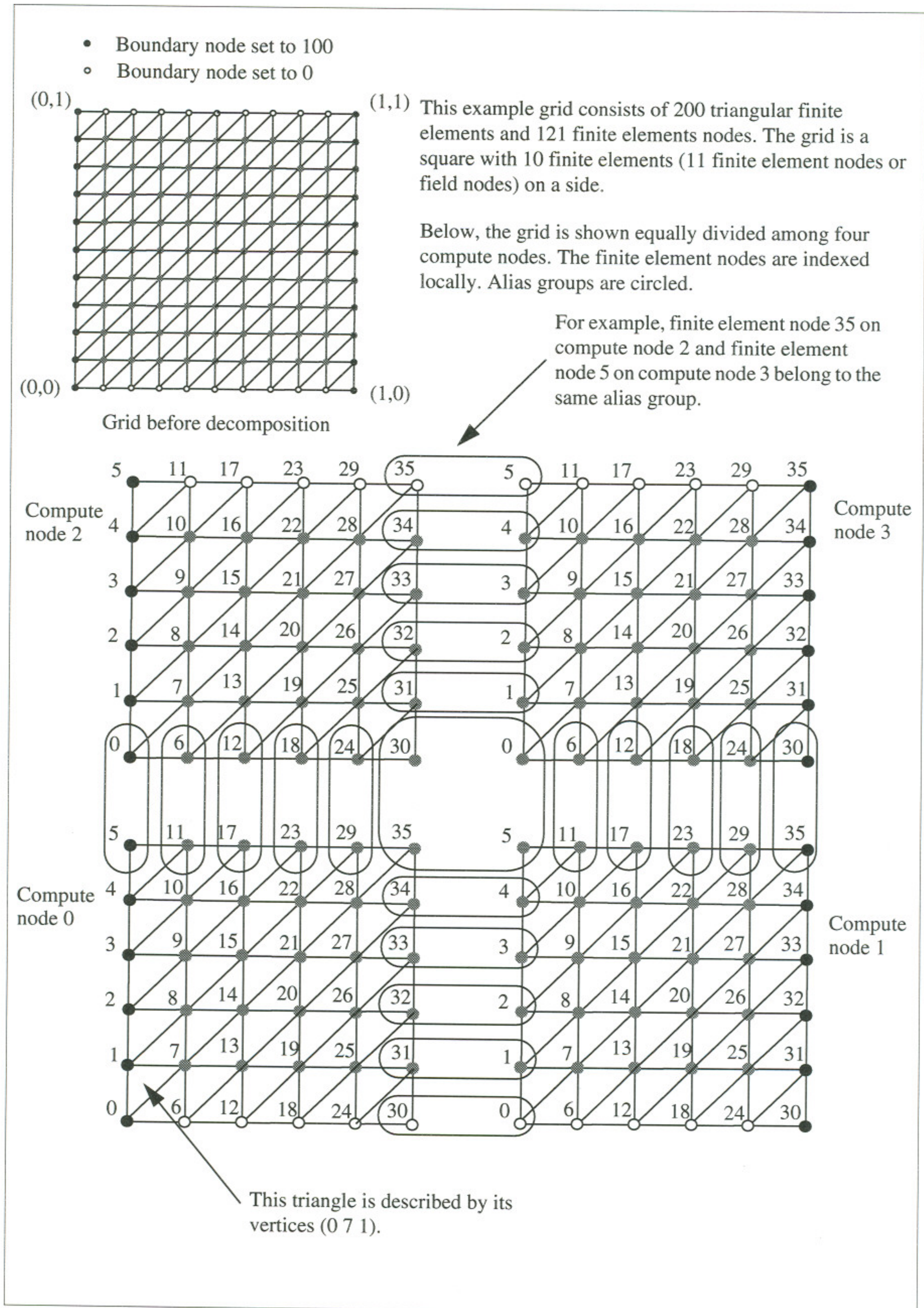


Figure 4.1. Distributed Grid Showing Alias Groups

Each field node has four values associated with it. Each field node has an x coordinate, a y coordinate, and a b value. The b value is the component of the vector $\boldsymbol{b}$ (from the matrix equation $\boldsymbol{Ax=b}$) that is assigned to that field node. Later, the solution vector $\boldsymbol{x}$ will have a value associated with each field node.

Each distributed vector ($\boldsymbol{b}$ and later the solution $\boldsymbol{x}$) has 121 (11 * 11) components. Each compute node has 36 components (labeled 0 through 35). Of the 36 field nodes on each compute node, 11 belong to alias groups. In the figure, the circles indicate alias groups. Because the field nodes are indexed locally, vector components from the same vector belonging to the same alias group may have different indices.

An alias group is really one field node shared by the compute nodes belonging to the alias group. In Figure 4.1, there are 121 unique field nodes. When the duplicates from the alias groups are included, each compute node has 36 field nodes, for a total of 36*4 or 144 field nodes handled by the entire set of compute nodes. Each compute node also has 50 triangles. A triangle is described by the local indices of its vertices. For example, (0 7 1) and (0 6 7) describe triangles.

Information Needed to Construct Alias Groups

Each compute node must be given a set of field nodes. Some of these field nodes are duplicated on different compute nodes; but at this stage, what field nodes are duplicated and what compute nodes they are duplicated on is not known.

One needs to somehow uniquely identify a field node over the set of compute nodes. This can be done with a global index or, as in the example presented in this chapter, with (x,y) coordinates.

Figure 4.2 shows the information needed to construct alias groups for the finite element triangular grid example. This figure also shows the information that defines the grid partitioning (the grid topology) and the boundary conditions.

Figure 4.2 shows that the example has four compute nodes with 36 finite element nodes (field nodes) and 50 finite elements (triangles) on each compute node. It then lists the (x,y) coordinates of each field node followed by the field node numbers that define the triangles themselves. Finally, it lists the boundary conditions as field node number and the constrained value it contains. These are Dirichlet boundary conditions.

`4`	*number of compute nodes*
`36`	*number of finite element nodes on compute node 0*
`50`	*number of finite elements (triangles) on compute node 0*
`0 0 0 0`	*x coordinate, y coordinate, b value, global index—36 of these*

```
0 0.1 0 1
0 0.2 0 2
0 0.3 0 3
0 0.4 0 4
0 0.5 0 5
0.1 0 0 11
     •
     •
```

After 36 lines showing x, y, f, global index, connectivity information appears. (0 7 1) describes a triangle whose vertiices are field nodes 0, 7, and 1.

```
0 7 1
0 6 7
1 8 2
     •
     •
```

Now the boundary conditions must be specified. Compute node 0 has 11 field nodes on the boundary. The specification is field node number followed by constrained value.

`11`	*Number of boundary field nodes on compute node 0.*
`6 0`	*Field node number, constrained value*

```
12 0
18 0
24 0
30 0
0 100
     •
     •
```

Now begin the data for compute node 1.

```
36
50
0.5 0 0 55
0.5 0.1 0 56
0.5 0.2 0 57
     •
     •
```

Figure 4.2. Information Needed for Alias Group Construction

The Alias List

To construct the alias groups, one must determine what field nodes belong to what alias groups by comparing the (x,y) coordinates of all the field nodes on all the nodes. Although this operation is time consuming, it need only be performed once. The comparison results in alias lists. These lists identify to each compute node which of its field nodes are aliased, the compute nodes to which they are aliased, and the local indices on those compute nodes.

The alias lists are constructed as follows. Each compute node gathers up the (x,y) coordinates of all its field nodes into a data structure (call it a *mailbag*). Then, the set of allocated compute nodes is thought of as a ring, and each compute node sends its *mailbag* to the compute node on one side and receives a *mailbag* from a compute node on the other side. This is one cycle. After *np*-1 cycles (where *np* is the number of compute nodes), every *mailbag* has visited every compute node.

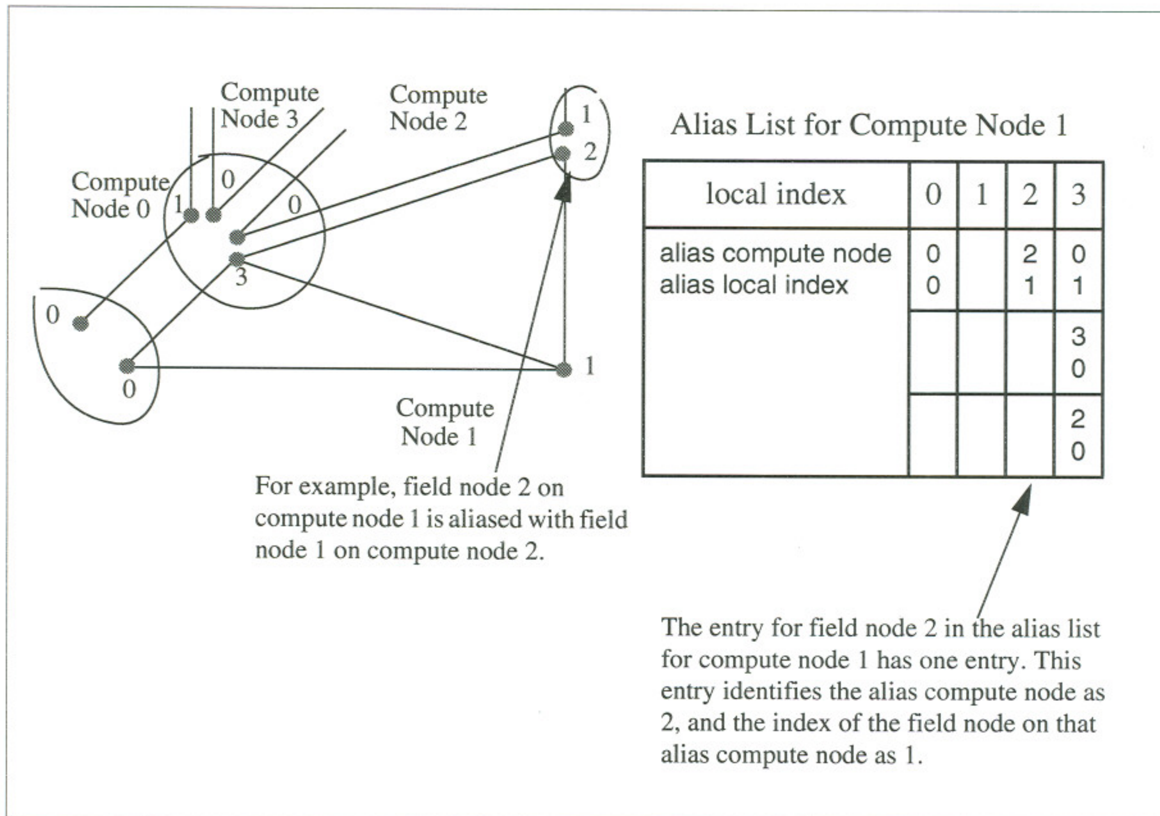
After a compute node receives a *mailbag* from another compute node, it compares the values in that *mailbag* with the values in its own database and constructs its alias list. Figure 4.3 shows a portion of a grid and the corresponding alias list for compute node 1.

The alias list is an array called *aliases*[]. This array exists on each compute node. It is an array of structures. There is one entry for every field node on the compute node. The entry points to a linked list containing alias information for that field node on that compute node. The pointer is NULL if the field node is not aliased. An entry in the linked list contains the node number of the compute node to which the field node is aliased and the index on that compute node.

Making a Distributed Vector Coherent

A distributed vector is a vector, each component of which is associated with a field node. Some of these field nodes may be aliased. Alias group coherence was first defined in "The Aliasing Method" on page 18. When this coherence is needed, was previously discussed in "Matrix-Vector Multiplication" on page 57.

A distributed vector is made coherent by applying a coherence operation to the field nodes in an alias group and replacing the value of the vector component at each aliased field node with the result. For example, if the coherence operation is a sum, each vector component in an alias group is replaced by the sum of all the vector components belonging to the alias group. The coherence operation was first introduced in "Matrix-Vector Multiplication" on page 57.



local index	0	1	2	3
alias compute node alias local index	0 0		2 1	0 1
				3 0
				2 0

Figure 4.3. Example of an Alias List

The distributed vector is made coherent as follows. All compute nodes send their aliased elements to the secretary of each alias group. The secretary is one of the compute nodes in the alias group. The secretary is chosen arbitrarily. The need for a secretary was previously discussed in "Dot Product" on page 59. The secretary performs the specified coherence function. Then, the secretary sends the answer back to the other nodes.

The Send and Receive Lists

Each compute node must maintain a send list and a receive list. A send list consists of a list of compute nodes and the vector element indices whose values need to be sent to those nodes. For example, in Figure 4.3, if compute node 0 is the secretary for all alias groups that it belongs to, then, when the distributed vector is made coherent, compute node 1 sends its components 0 and 3 to compute node 0; compute node 2 sends its component 0 to compute node 0; and compute node 3 sends its component 0 to compute node 0. A receive list consists of a list of compute nodes and the vector element indices whose values will be received from those nodes. These send and receive lists can be calculated locally from the alias list.

There are two arrays *get_slist*[] (an array of send lists) and *get_rlist*[] (an array of receive lists). The index in each array indicates the compute node to which the current node will send or receive data. Each element in the array contains a pointer to the start of a linked list. Each element in the linked list contains the local index (the index on this compute node) and the remote index (the index on the other compute node). If no data are to be sent to or received from that node, the pointer is NULL.

Figure 4.4 shows an example of the send and receive lists. This example is for the same grid shown in Figure 4.3.

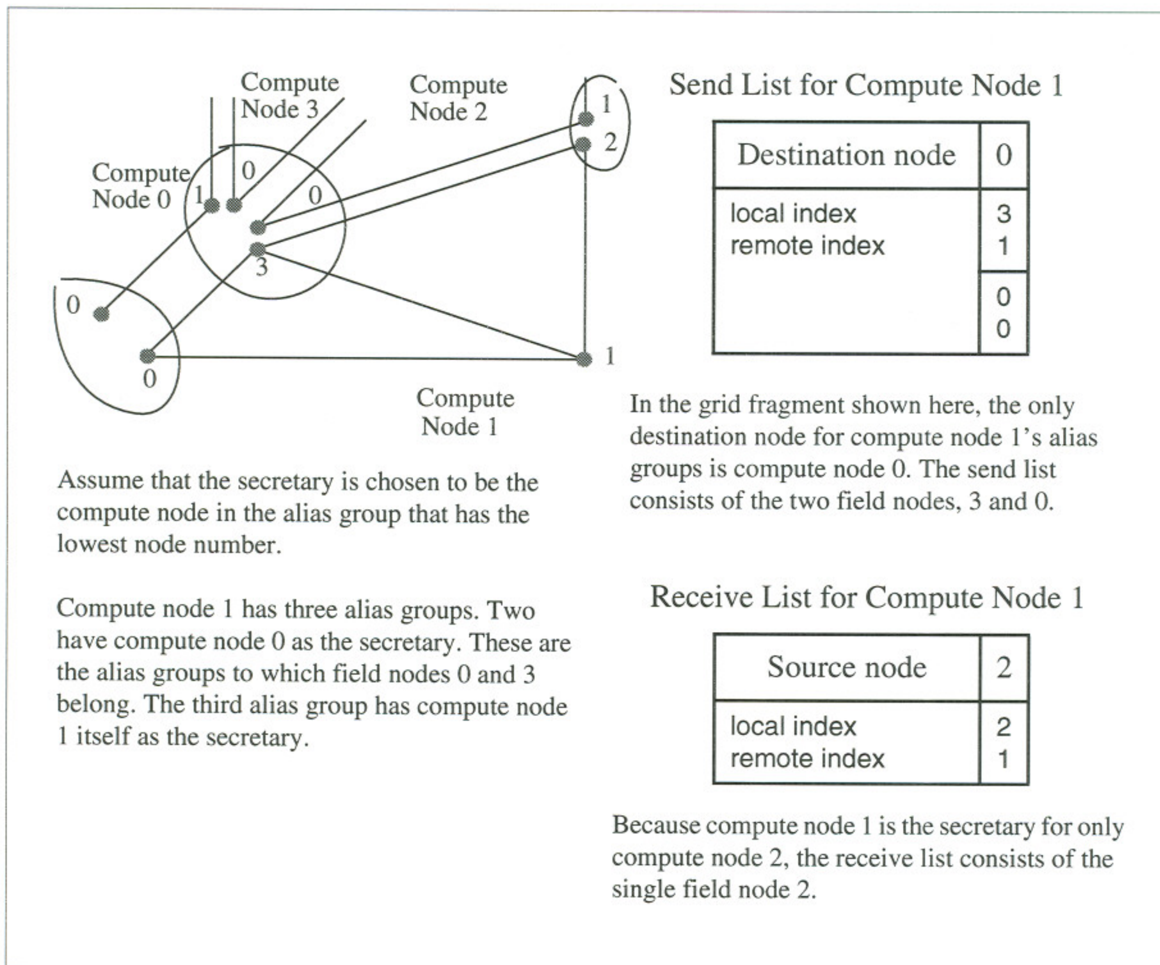


Figure 4.4. Example of Send and Receive Lists

In summary, making a distributed vector coherent consists of the following steps:

1. A compute node sends data to its send list. This means that the compute node sends data to the secretaries of its alias groups.

2. A compute node receives data from its receive list. This means that the compute node receives data from the members of the alias groups to which it is the secretary.
3. The secretaries perform the specified coherence operation.
4. A compute node sends the answer to its receive list. This means that the secretaries send the answer to the other members of its alias groups.
5. A compute node receives the answer from its send list. This means that the non-secretaries obtain the answer.

Example of Making a Vector Coherent

Figure 4.5 illustrates in detail what happens when a distributed vector is made coherent. The figure shows a vector distributed over four compute nodes. This vector has nine components, indexed globally from 0 to 8. No single compute node has all the components. Compute node 0 has the first five; and it indexes them locally from 0 to 4. Compute node 1 has the 1st, 2nd, 5th, and 6th components (counting starts form 0); and it indexes them from 0 through 3.

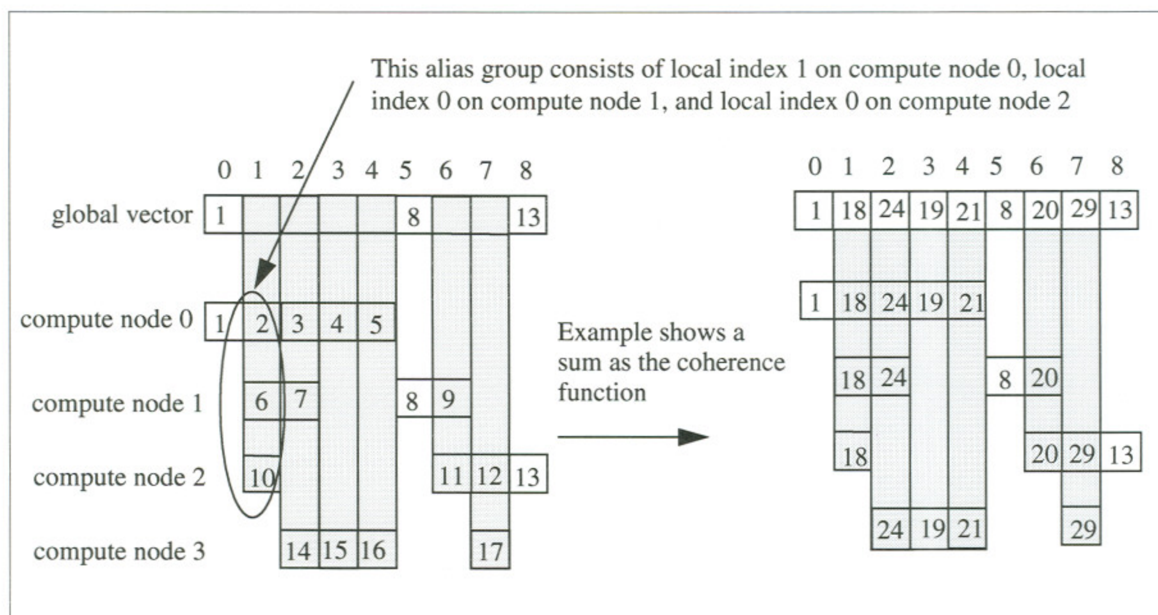


Figure 4.5. Global Description of Coherence

In Figure 4.5, the numbers inside the boxes are the values of the vector components. Note that these values are different on different compute nodes. If the coherence function is a sum, the result, after making the vector coherent, is that the value

on each component in the alias group is summed and the new value appears on each compute node. This operation has no effect on the values of vector components that are not aliased.

Figures 6 and 7 describe in more detail how the vector shown in Figure 4.5 is made coherent. An explicit library call must be made to make a distributed vector coherent. Usually, this call is within a loop. It is more efficient to allocate the memory needed for the call before the loop begins and to free this memory after the loop ends. For example,

```
init_vcomb(t_double, &out_sendlist, &out_recvlist,
           &send_corres, &nbr_send_corres,
           &recv_corres, &nbr_recv_corres);
```

Loop begins ...

```
vcomb((char *)y, dsum, t_double,
      out_sendlist, out_recvlist,
      send_corres, nbr_send_corres,
      recv_corres, nbr_recv_corres);
```

Loop ends ...

```
reset_vcomb(out_sendlist, out_recvlist,
            send_corres, recv_corres,
            nbr_send_corres, nbr_recv_corres);
```

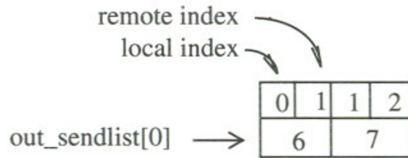
The parameters in the example code have the following meanings:

- In **init_vcomb**(), *t_double* is an **enum** variable that identifies the type as **double**. *out_sendlist*[] and *out_recvlist*[] are the send and receive buffers; *send_corres*[] and *recv_corres*[] are integer arrays that contain the compute node numbers that the calling node corresponds with. Each element of *out_sendlist*[] and *out_recvlist*[] points to a buffer to be sent to or received from another compute node. For example, *out_sendlist*[*i*] points to data that will be sent to compute node *send_corres*[*i*].
- In **vcomb()**, *y* is the vector to be made coherent and *dsum* is a pointer to the combination function, which is a **double** sum. Pointers to the buffers allocated in **init_vcomb**() are also provided.
- In **reset_vcomb()**, the buffers allocated in **init_vcomb**() are released. .

Compute Node 1

This figure shows the detailed send and receive lists that compute node 1 has for the decomposition shown in Figure 4.4. The *out_sendlist*[] is what is sent to the secretary. The indices are not sent, just the values. It is not necessary to send the indices because the receiving node can determine them from its own alias lists.

1. Send out the send lists.



send_corres[0] = 0

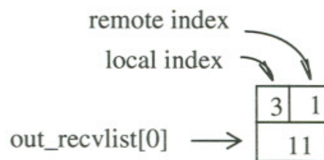
nbr_send_corres = 1

Compute node 1 has two field nodes aliased with compute node 0, and compute node 0 is the secretary for those field nodes. The local indices (the indices on compute node 1) of these field nodes are 0 and 1. The corresponding indices on compute node 0 are 1 and 2. The values on compute node 1 are 6 and 7.

Each compute node maintains a send list for each of the other compute nodes. In this example, compute node 1 has a send list for compute node 0 because compute node 0 is the secretary for compute node 1's aliased field nodes.

Compute node 1 also has an aliased field node with compute node 2. It is itself the secretary for that alias group. Consequently compute node 1 has a receive list.

2. Receive the receive lists.



recv_corres[0] = 2

nbr_recv_corres = 1

After the receive, this field contains the value 11.

Compute node 1 has two field nodes aliased with compute node 2, and one field node aliased with compute node 3. It is the secretary for one of the field nodes aliased with compute node 2 (index 3 on compute node 1 and index 1 on compute node 2).

Compute node 1's receive list for compute node 2 is compute node 2's send list for compute node 1.

Note that in this example *out_recvlist*[0] shows only one value. If compute node 1 were the secretary for more than one alias with compute node 2, those aliased values would be in *out_recvlist*[0] also. Compute node 1 would know that each value is a **double** and would skip 8 bytes to get subsequent entries.

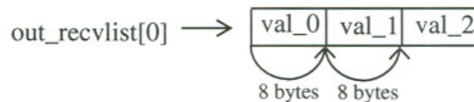


Figure 4.6. Local Description of Combination (Part A)

This figure continues the description of the steps involved in a combination begun in Figure 4.6.

3. Perform the function. Then, send out the receive list.

out_recvlist[0] ⟶

3	1
20	

recv_corres[0] = 2

nbr_recv_corres = 1

Compute node 1 takes its value of 9 (at local index 3) and adds it to 11, the value that came from compute node 3. The result is that out_recvlist[0] now contains the value 20.

Then, compute node 1 sends *out_recvlist*[0] to compute node 2.

4. Receive the send lists.

out_sendlist[0] ⟶

0	1	1	2
18		24	

send_corres[0] = 0

nbr_send_corres = 1

Compute node 1 now receives its send lists. The send lists were sent to the secretary node, where the values were modified in accordance with the coherence function. These are now the coherent values.

Figure 4.7. Local Description of Combination (Part B)

Chapter 5 Implementation of the Conjugate Gradient Method

Introduction

This chapter describes how the conjugate gradient algorithm is programmed with the aliasing method. The chapter describes how the matrix is constructed from input data and then partitioned among the compute nodes. The major operations of the conjugate gradient algorithm are a matrix-vector multiplication and a dot product. This chapter describes how these operations are performed using aliases.

The Coefficient Matrix

The coefficient matrix for finite element problems has three important characteristics. It is large, sparse, and symmetric. Because it is large, it makes sense to store it as a distributed matrix. Because it is sparse, it does not make sense to store the global matrix as a large 2x2 array. Because it is symmetric, one can use the conjugate gradient algorithm.

One can take advantage of the symmetry to use less storage space but this would be at the expense of additional calculation. The work presented here does not take advantage of matrix symmetry to reduce storage requirements.

Assembling the Matrix

After the lists (the alias list, the send list, and the receive list) are formed, it is necessary to form the matrix. The matrix is stored in assembled form. The difference between assembled and unassembled matrices was discussed in "Finite Element Grid and the Corresponding Matrix Equation" on page 9.

The matrix is first assembled as much as possible on individual compute nodes. The complete assembly of the matrix requires communication among the compute nodes. If the unassembled parts of an assembled matrix element lie on different compute nodes,

one of those compute nodes is chosen to contain the matrix element. Figure 5.1 describes how much assembly can be performed on a distributed finite element mesh without communication.

To assemble the matrix it is necessary to know both the connectivity of the finite element mesh and the (x,y) coordinates of each point. This information is part of the initial data. Finite elements that are next to each other contribute their elemental matrix elements to form assembled matrix elements. This makes the connectivity information important for assembly.

The routine **mat_local_assemble()** constructs the assembled matrix as much as it can locally. The arguments of this routine specifiy *ne*, the number of finite elements on the compute node and *ni*[], *nj*[], and *nk*[], integer arrays that define the connectivity of the finite element mesh. Figure 5.1 shows how the assembled matrix is constructed.

After **mat_local_assemble()**, **mlist()** makes the send and receive lists for the matrix elements. These lists are then used by **mcomb()** to assemble the matrix. After assembly, the matrix elements are distributed as shown in Figure 5.4. That is, if field node i is aliased, the self-term a_{ii} is only present on the compute node that is the secretary for field node i. Cross-terms only appear on one of the dividing compute nodes. Figure 5.4 indicates the presence of a cross-term by a solid line. The dotted line shows where it would also occur if it were aliased. Finally, the global load vector f must be combined. These routines are called as follows:

```
mat_local_assemble(ne, ni,nj,nk, A, &Amax,Arow,Acol, f);
mlist(is_alias, Amax, Arow, Acol);
mcomb(A, Amax, Arow, Acol, dsum);

init_vcomb(t_double, &out_sendlist, &out_recvlist,
           &send_corres, &nbr_send_corres,
           &recv_corres, &nbr_recv_corres);

vcomb((char *)f, dsum, t_double,
      out_sendlist, out_recvlist,
      send_corres, nbr_send_corres,
      recv_corres, nbr_recv_corres);

reset_vcomb(out_sendlist, out_recvlist,
            send_corres, recv_corres,
            nbr_send_corres, nbr_recv_corres);
```

Assembled vs. Unassembled Matrices

The question now arises whether one should assemble the matrix to solve it. At a

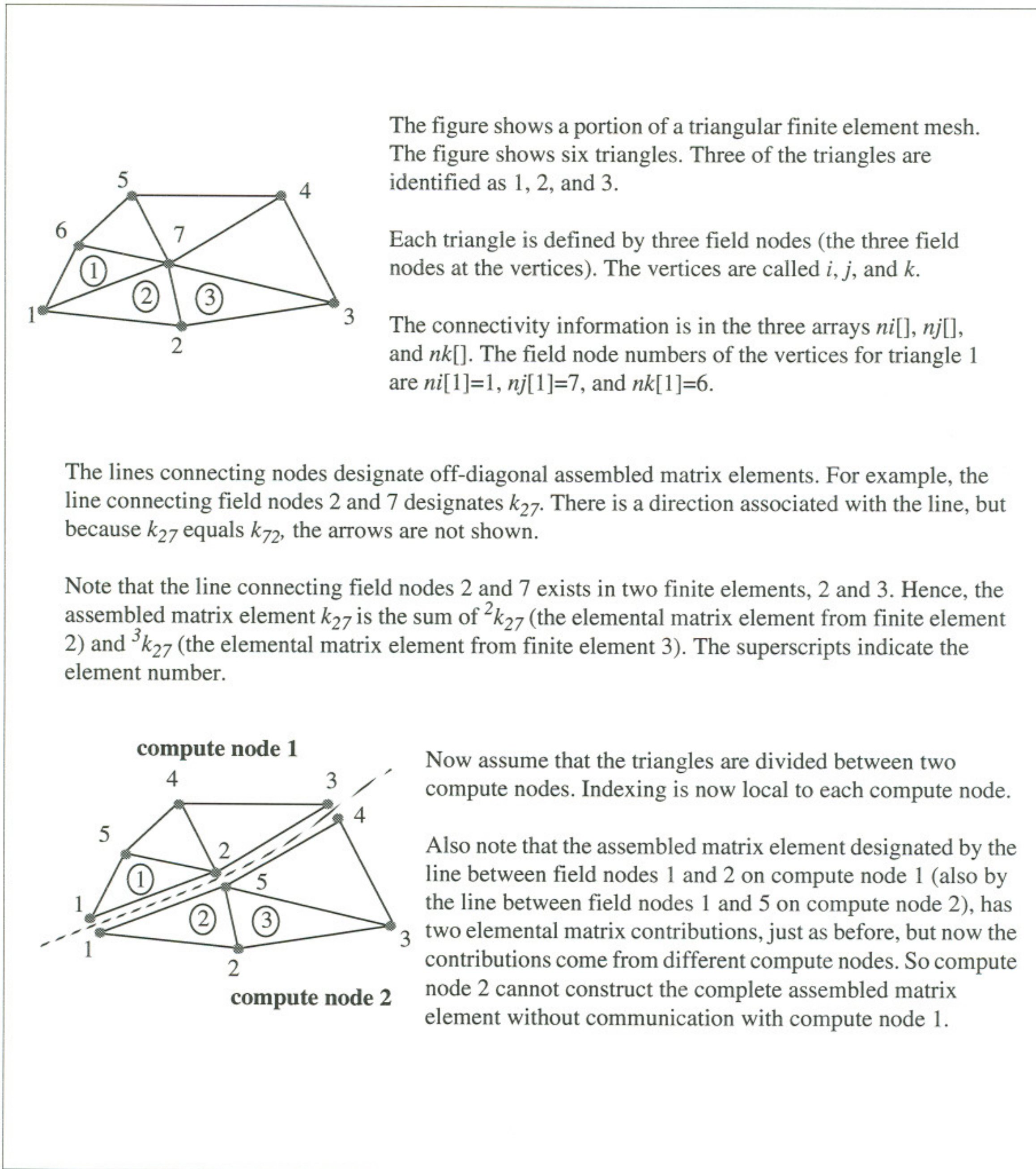
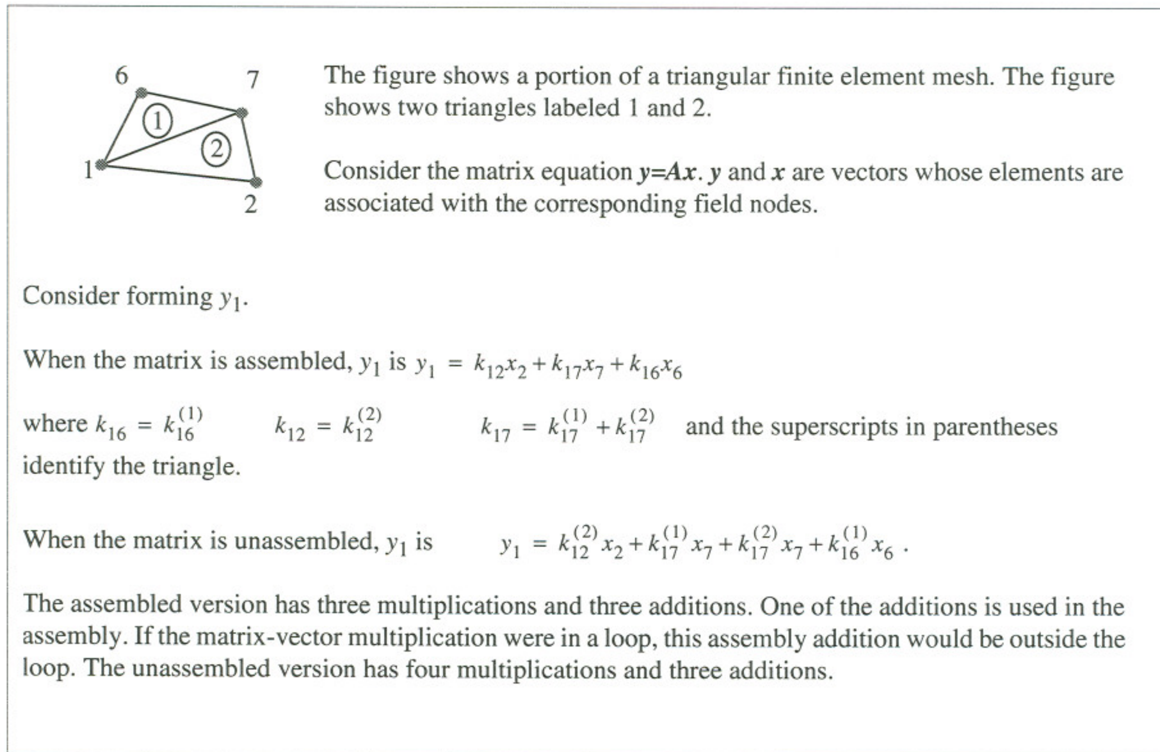


Figure 5.1. Construction of Assembled Matrix Elements

high enough level in the algorithm, it doesn't matter whether the matrix is stored as one assembled matrix or as a collection of 3x3 elemental matrices (called the unassembled matrix). During the iterative steps of the conjugate gradient algorithm, the only access to the matrix is through a routine that performs the matrix-vector multiplication. As far as the program, is concerned the storage mechanism is internal to this routine.

There is an important difference, however, between using the assembled matrix and using the unassembled matrix when one considers how the calculation is performed.

When the matrix is stored as 3x3 elemental matrices, one performs a number of multiplications followed by additions. When the matrix is stored as an assembled matrix, one performs additions followed by multiplications. There is less floating point work when the matrix is assembled. Figure 5.2 illustrates the difference between unassembled and assembled computations.



The figure shows a portion of a triangular finite element mesh. The figure shows two triangles labeled 1 and 2.

Consider the matrix equation $\boldsymbol{y}=\boldsymbol{Ax}$. $\boldsymbol{y}$ and $\boldsymbol{x}$ are vectors whose elements are associated with the corresponding field nodes.

Consider forming y_1.

When the matrix is assembled, y_1 is $y_1 = k_{12}x_2 + k_{17}x_7 + k_{16}x_6$

where $k_{16} = k_{16}^{(1)}$ $k_{12} = k_{12}^{(2)}$ $k_{17} = k_{17}^{(1)} + k_{17}^{(2)}$ and the superscripts in parentheses identify the triangle.

When the matrix is unassembled, y_1 is $y_1 = k_{12}^{(2)}x_2 + k_{17}^{(1)}x_7 + k_{17}^{(2)}x_7 + k_{16}^{(1)}x_6$.

The assembled version has three multiplications and three additions. One of the additions is used in the assembly. If the matrix-vector multiplication were in a loop, this assembly addition would be outside the loop. The unassembled version has four multiplications and three additions.

Figure 5.2. Unassembled vs. Assembled Matrix-Vector Multiplication

Conjugate Gradient Operations

As previously mentioned in "The Aliasing Method" on page 18, each iterative step of the conjugate gradient method has one matrix-vector multiplication and two dot products. The parallelization of these operations was discussed in "Parallel Conjugate Gradient Method" on page 55. This section shows how these operations are implemented, using the techniques described in Chapter 4, "Implementation of Aliasing."

Dot Product

This section describes how the dot product is formed when both participating vectors are distributed over the compute nodes. A dot product of two n-dimensional vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is a scalar s defined as

$$s = \boldsymbol{x} \cdot \boldsymbol{y} = \sum_{i=1}^{n} x_i y_i$$

On a single processor, such a dot product can be implemented in C code as follows:

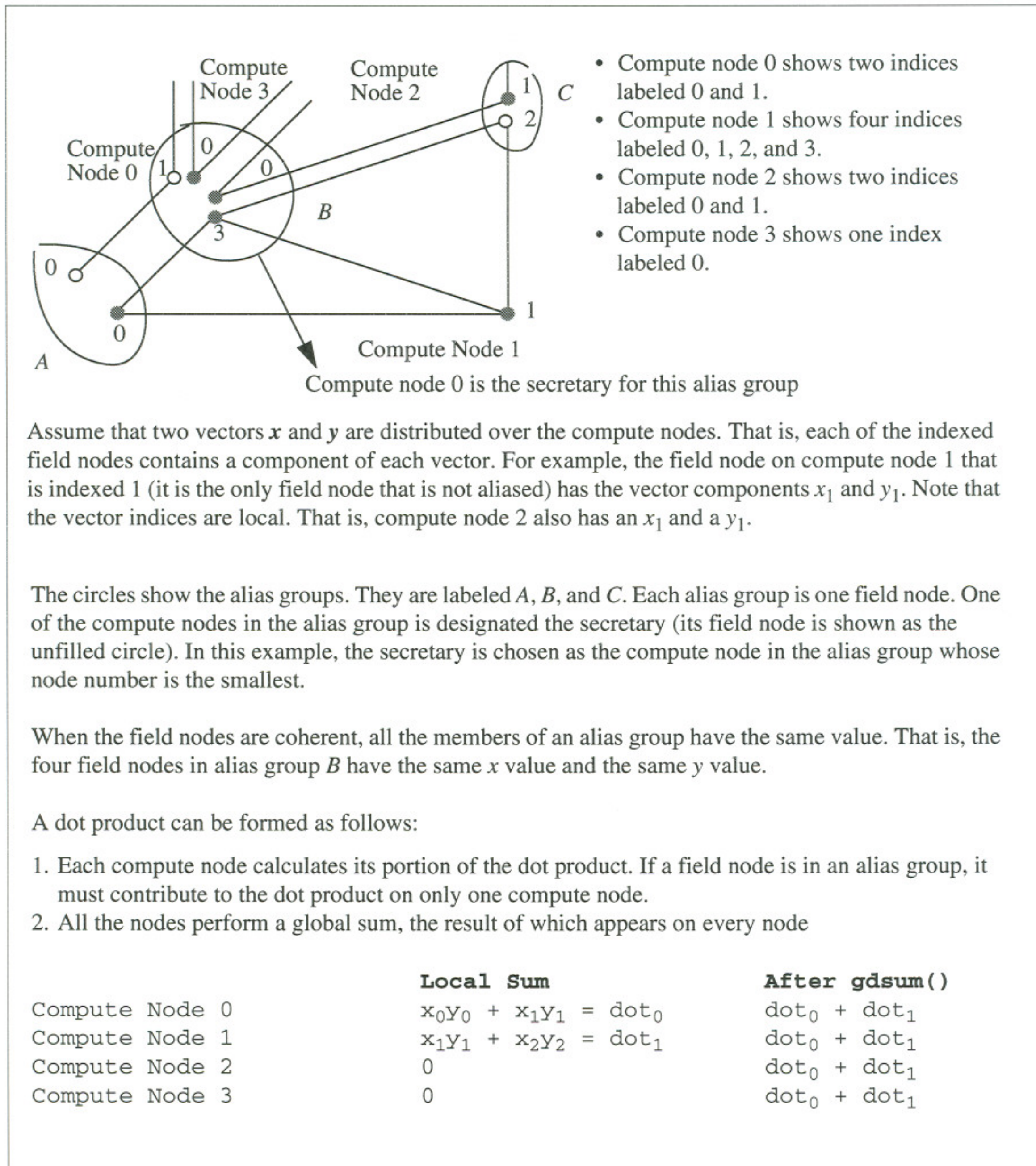
```
s=0;
for(i=0; i<n; i++)
    s += x[i]*y[i];
```

To form the dot product of $\boldsymbol{x}$ and $\boldsymbol{y}$, when they are vectors distributed over a set of compute nodes, each compute node performs its portion of the dot product. Then, a global routine sums the contributions from each compute node. The result appears on all the compute nodes. However, one must be careful not to include a contribution from each member of the alias group. Only one member of an alias group may contribute. In the example shown in Figure 5.3, this contribution comes from the secretary.

Here is some example code that forms a dot product of two distributed vectors. The code is written for clarity, not efficiency. The reason this code is not efficient is the presence of the conditional in the **for** loop. Conditionals can invalidate a pipeline, and in loops they do it again and again.

The dot product code is executed on every compute node. The values of *dot_value* on each compute node are different because each compute node has different components of $\boldsymbol{x}$ and $\boldsymbol{y}$. The routine **gdsum()** is a synchronous routine (supplied as part of the operating system's message-passing library); that is, all compute nodes must execute the routine, and no compute node can proceed past **gdsum()** until all compute nodes have completed the routine.

```
double dot(double *x, double *y,
              int *is_alias, int *is_secretary, int n)
{
int i;
double dot_value=0.0, dtemp;
    for(i=0; i<n; i++) {
        if(is_alias[i]){
            if(is_secretary[i])
                dot_value += x[i] * y[i];
            else continue;
        }
        else
            dot_value += x[i] * y[i];
    }
    gdsum(&dot_value, 1, &dtemp);
    return dot_value;
}
```



- Compute node 0 shows two indices labeled 0 and 1.
- Compute node 1 shows four indices labeled 0, 1, 2, and 3.
- Compute node 2 shows two indices labeled 0 and 1.
- Compute node 3 shows one index labeled 0.

Assume that two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are distributed over the compute nodes. That is, each of the indexed field nodes contains a component of each vector. For example, the field node on compute node 1 that is indexed 1 (it is the only field node that is not aliased) has the vector components x_1 and y_1. Note that the vector indices are local. That is, compute node 2 also has an x_1 and a y_1.

The circles show the alias groups. They are labeled *A*, *B*, and *C*. Each alias group is one field node. One of the compute nodes in the alias group is designated the secretary (its field node is shown as the unfilled circle). In this example, the secretary is chosen as the compute node in the alias group whose node number is the smallest.

When the field nodes are coherent, all the members of an alias group have the same value. That is, the four field nodes in alias group *B* have the same *x* value and the same *y* value.

A dot product can be formed as follows:

1. Each compute node calculates its portion of the dot product. If a field node is in an alias group, it must contribute to the dot product on only one compute node.
2. All the nodes perform a global sum, the result of which appears on every node

	Local Sum	**After gdsum()**
Compute Node 0	$x_0y_0 + x_1y_1 = dot_0$	$dot_0 + dot_1$
Compute Node 1	$x_1y_1 + x_2y_2 = dot_1$	$dot_0 + dot_1$
Compute Node 2	0	$dot_0 + dot_1$
Compute Node 3	0	$dot_0 + dot_1$

Figure 5.3. Formation of a Dot Product of Distributed Vectors

In the code, *is_alias*[] is a boolean array that indicates if the indexed field node is aliased or not. That is, *is_alias*[*i*] is 1 if field node *i* belongs to an alias group and 0 if it does not. Similarly *is_secretary*[*i*] is 1 if field node *i* is aliased and belongs to a secretary node and 0 otherwise.

The code for the dot product checks every field node. If the field node is not aliased, it accepts a contribution. If the field node is aliased, it accepts a contribution only if the field node also belongs to a secretary.

The routine **gdsum()** sums the value *dot_value* on each compute node. It returns the global sum in *dot_value*, and uses a temporary variable provided as *dtemp*. The 1 as the second argument indicates that *dot_value* is a scalar; a 2 would indicate that it was a two-element vector.

A faster dot product can be coded by using a high-performance KAI routine internally. This KAI routine is part of the Basic Math Library from Kuck and Associates, Inc. that is provided with the Paragon supercomputer operating system. It uses the KAI routine to calculate the dot product, then subtracts the contribution that resulted from including elements that were aliased but whose compute nodes were not secretaries. In this way, the conditional within the loop is avoided. For large vectors, this is faster than the example shown here.

Matrix-Vector Multiplication

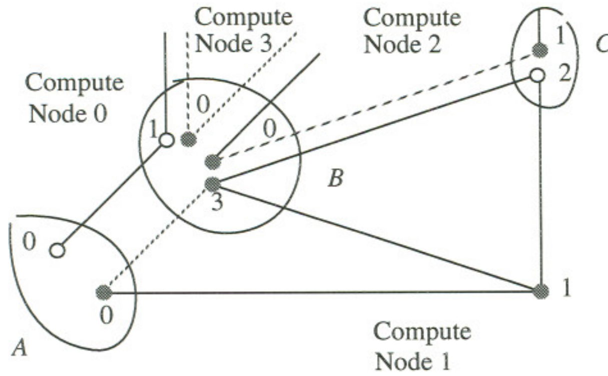
This section describes how the matrix-vector product is formed. when both participating vectors are distributed over the compute nodes.

As shown in Figure 5.4, the vector components are aliased and the matrix elements are not. Assume the matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{Ax}$ where both vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are distributed. Each field node has associated with it a component of $\boldsymbol{x}$ and a component of $\boldsymbol{y}$.

Each compute node performs its pull-ins and calculates its self-terms independently. (For a definition of pull-ins and self-terms, refer to "Matrix-Vector Multiplication" on page 57.) After the pull-ins and self-terms are calculated, the complete $\boldsymbol{y}$ element is still not present. The vector must be made coherent with a sum as the coherence operation.

Because $\boldsymbol{A}$ is sparse, it is not stored as a two-dimensional array. Instead, the matrix is stored as three one-dimensional arrays. These arrays are *A*[], *Arow*[], and *Acol*[]. *A*[] contains the actual matrix value; and *Arow*[] and *Acol*[] contain the row and column indices. For example, the matrix element stored in *A*[10] has row position *Arow*[10] and column position *Acol*[10]. This is not the most optimal storage, but it is the easiest to understand.

Another characteristic of the storage scheme is that equal *Arow*[] indices are grouped together. That is, *A*[] exhausts all the values in one row before moving on to the next. However, the row indices do not necessarily increase as the *Arow*[] index increases. That is, *A*[] may contain all of row 1, followed by row 4, followed by row 2. Once the value of a *Arow*[] index changes, it does not return to a previous value. For example, a



This figure describes how y is constructed from $y = Ax$. A is a two-dimensional matrix. x and y are vectors. Each field node contains an element of x and an element of y.

This figure is a portion of a triangular finite element grid. The lines between field nodes indicate possible matrix elements. That is, the line between field nodes 0 and 1 on compute node 0 designates a_{01} (also a_{10} because the matrix is symmetric).

The field nodes are aliased, but the matrix elements are not. The dotted lines indicate where aliased matrix elements would be if they were not aliased. For example, the dotted line between field nodes 0 and 3 on compute node 1 designates a_{03}. Because field node 0 on compute node 1 is also field node 0 on compute node 0 and field node 3 on compute node 1 is also field node 1 on compute node 0, a_{03} on compute node 1 is also a_{01} on compute node 0. Only the matrix element on compute node 0 is stored.

Assume that each compute node performs its pull-ins independently. The compute node also calculates the self-term $(a_{ii}x_i)$ for those field nodes for which it is the secretary. These self-terms are shown in bold. After the pull-ins and self-terms are calculated, the complete y element is not present. The values in an alias group must be summed, and the sum sent to all members of the alias group.

Compute Node	Field Node	Aliased?	Secretary?	Pull-ins and Self-Terms	After Combination compute node (value)
0	0	yes	yes	$\mathbf{a_{00}x_0} + a_{01}x_1$	$0(\mathbf{a_{00}x_0} + a_{01}x_1) + 1(a_{01}x_1)$
	1	yes	yes	$\mathbf{a_{11}x_1} + a_{10}x_0$	$0(\mathbf{a_{11}x_1} + a_{10}x_0)$ $+ 1(a_{31}x_1 + a_{32}x_2)$ $+ 2(...) + 3(...)$
1	0	yes	no	$a_{01}x_1$	$0(\mathbf{a_{00}x_0} + a_{10}x_0) + 1(a_{01}x_1)$
	1	no	no	$\mathbf{a_{11}x_1} + a_{10}x_0 + a_{12}x_2$ $+ a_{13}x_3$	$1(\mathbf{a_{11}x_1} + a_{10}x_0 + a_{12}x_2 + a_{13}x_3)$
	2	yes	yes	$a_{21}x_1 + \mathbf{a_{22}x_2} + a_{23}x_3$	$1(a_{21}x_1 + \mathbf{a_{22}x_2} + a_{23}x_3) + 2(...)$
	3	yes	no	$a_{31}x_1 + a_{32}x_2$	$0(\mathbf{a_{11}x_1} + a_{10}x_0)$ $+ 1(a_{31}x_1 + a_{32}x_2)$ $+2(...) + 3(...)$
2	0	yes	no	2(...)	$0(\mathbf{a_{11}x_1} + a_{10}x_0) +$ $1(a_{31}x_1 + a_{32}x_2) + 2(...) + 3(...)$
	1	yes	no	2(...)	$1(a_{21}x_1 + \mathbf{a_{22}x_2} + a_{23}x_3) + 2(...)$
3	0	yes	no	3(...)	$0(\mathbf{a_{11}x_1} + a_{10}x_0)$ $+ 1(a_{31}x_1 + a_{32}x_2)$ $+2(...) + 3(...)$

Figure 5.4. Matrix-Vector Multiplication with Coherence

portion of a small matrix might appear as follows:

```
A:    1.0  0.5 -1.5  0.5  1.5 -1.5 -0.5 -1.5 -1.5  3.0  0.0  0.0 -0.5
Arow:0    0    0    1    1    1    1    3    3    3    3    2    2
Acol:0    1    3    0    1    3    2    0    1    3    2    3    1
```

The following C code forms $\boldsymbol{y} = \boldsymbol{Ax}$ up to the point of combination. It is the portion where each compute node calculates its pull-ins and self-terms independently. *Amax* is the number of nonzero elements in $\boldsymbol{A}$ and hence the length of the three arrays. The routine **dvecinit**() zeroes out $\boldsymbol{y}$.

```
void mat_vect(double *A, long Amax, long *Arow, long *Acol,
              double *x, double *y)
{
int i;
   dvecinit(y,Amax);
   for(i=0; i<Amax; i++)
      y[Arow[i]] += A[i] * x[Acol[i]];
}
```

Conjugate Gradient Algorithm

The implementation consists of some setup code and the conjugate gradient code. The one-time code constructs the alias, send, and receive lists, then incorporates the boundary conditions. The conjugate gradient code makes an initial guess and then enters a loop that executes the conjugate gradient steps.

Setup Code

The alias, send, and receive lists are constructed by **vsync**(). The *is_alias*[] and *is_secretary*[] arrays are constructed by **get_is_alias**() and **get_is_secretary**(). *nn* is the number of field nodes. These routines are called as follows:

```
vsync();
get_is_alias(is_alias,nn);
get_is_secretary(is_secretary,nn);
```

Next, one must incorporate the boundary condition. The routine **add_constraints**() modifies the matrix to take into account boundary conditions. The arrays *c*[], *nc*[], and *nd*[] define the constraints. Again, *nn* is the number of field nodes.

```
add_constraints(A, &Amax,Arow,Acol, epsilon,
                f, c, nc, nd, is_alias, is_secretary, nn);
```

Conjugate Gradient Code

The equation to be solved is ***Ax=f***. Before the iteration begins, one must make an initial guess for the solution (***x***). Then, one calculates the initial residual (***residual***). If the initial guess is 0, the initial residual is ***f*** (***f*** - ***Ax*** = ***f*** - ***A****0 = ***f***). One also calculates ρ, the square of the initial residual. Then, one defines an initial search direction (***g***) as the direction of the residual (***g*** = ***f***).

Steps 1 and 2 consist of making an initial guess for ***x*** and defining the initial residual. If that residual is less than epsilon, stop.

```
for(i=0; i<nn; i++)
   residual[i] = f[i];
```

Steps 3 and 4 consist of forming the square of the initial residual and then setting the initial search direction ***g*** to the residual.

```
rho = dot(residual,f,is_alias, is_secretary, nn);
for(i=0; i<nn; i++)
   g[i] = residual[i];
```

Step 5 is the iteration. Each iteration has three major steps. The first is updating the solution vector. The solution vector is modified by adding to it a constant times the search direction. The second is to form a from the new solution vector. The third step is to choose a new search direction.

In Appendix C, "The Conjugate Gradient Method" the iteration consists of substeps identified by letters form A to I. The discussion that follows describes the $(k+1)^{th}$ iteration, using values calulated in the kth iteration. The first iteration (k=1) uses the intial values (k=0 values).

Each iteration has a matrix-vector multiplication and two dot products. The matrix-vector multiplication is the matrix ***A*** times the vector ***g***. It must be followed by a **vcomb**() to make the result coherent.

Step A is the matrix-vector multiplication.

```
init_vcomb(t_double, &out_sendlist, &out_recvlist,
           &send_corres, &nbr_send_corres,
           &recv_corres, &nbr_recv_corres);

for(iter = 0; iter < total_nn; iter++) {
   mat_vec(A,Amax,Arow,Acol, g,Ag);
   vcomb((char *)Ag, dsum, t_double,
      out_sendlist, out_recvlist,
      send_corres, nbr_send_corres,
      recv_corres, nbr_recv_corres);
```

Step B forms λ which is ρ divided by the first dot product.

```
gAg = dot(g,Ag,is_alias, is_secretary, nn);
lambda = rho/gAg;
```

Now that λ is known, the solution vector is ready to be updated. The result of the matrix-vector multiplication is also used to calulate a new residual.

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \lambda^{(k)}\boldsymbol{g}^{(k)}$$

$$\boldsymbol{residual}^{(k+1)} = \boldsymbol{residual}^{(k)} - \lambda^{(k)}\boldsymbol{Ag}^{(k)}$$

Steps C and D update the solution vector $\boldsymbol{x}$ and the residual. Step E saves the old ρ.

```
for(j=0; j<nn; j++) {
    x[j] += lambda * g[j];
    residual[j] -= lambda * Ag[j];
 }
 rho0 = rho;
```

Step F forms the new ρ, which is the residual squared. This is the second dot product. Step G forms β, which is the ratio of the new and old ρ's. β is used to update the search direction.

$$\boldsymbol{g}^{(k+1)} = \boldsymbol{residual}^{(k+1)} + \beta^{(k)}\boldsymbol{g}^{(k)}$$

Step H updates the search direction $\boldsymbol{g}$. Step I is a fail-safe; a loop terminates if the loop count exceeds a preset value.

```
rho = dot( residual,residual,is_alias, is_secretary, nn);
beta = rho/rho0;
for(j=0; j<nn; j++)
   g[j] = residual[j] + beta * g[j];
} /* end of conjugate gradient iteration */

reset_vcomb(out_sendlist, out_recvlist,
             send_corres, recv_corres,
             nbr_send_corres, nbr_recv_corres);
```

Testing for Convergence

The conjugate gradient iteration continues until the residual is less than a certain prescribed *epsilon* (10^{-5}). This test is performed in two places. The first is right after the

initial residual is set, after the initial guess for ***x***, on the off chance the initial guess is good enough. Then, the test is performed again as part of the iteration when ***x*** is updated.

The residual is a vector distributed across the compute nodes. The routine **gte()** first finds the largest element that is local to each compute node. If this element is less than epsilon, **gte()** sets a value *greater_than_epsilon* to 0 (FALSE). Then, **gte()** performs a global OR of *greater_than_epsilon* over all compute nodes and returns this value. The result is that **gte()** returns 1 if any element of the residual on any compute node is greater than *epsilon*.

gte() is used to provide a stopping condition to the conjugate gradient iteration. **gte()** is executed after each iteration, and the iteration continues if **gte()** returns a 1.

Chapter 6 A Random Matrix

Introduction

This chapter describes a limiting case where the aliasing method, although still working and producing correct answers, becomes inefficient. The inefficiency occurs because each compute node has several alias groups and because the alias groups are large.

The matrix investigated is the random matrix from the Numerical Aerodynamic Simulation (NAS) Conjugate Gradient (CG) benchmark [3]. A matrix is called random when whether a matrix element is nonzero is determined randomly.

That the nonzero matrix elements may be scattered throughout the matrix does not necessarily mean that the matrix is random. A suitable reordering can make the non-random nature of the matrix apparent. The aliasing method, however, does not require reordering. With the aliasing method, the non-randomness of a matrix is taken advantage of regardless of how the matrix is indexed. As can be seen from Chapter 4, "Implementation of Aliasing," only those vector components that need to be duplicated are duplicated, and they are duplicated on only those compute nodes that require them.

In summary, if the matrix has a random pattern of nonzeros, the aliasing method still works, but performance and efficiency are degraded. Such a linear system ends up with nearly every vector component belonging to an alias group. Solving such a system is characterized by a large number of long messages in each iteration step. When this occurs, it is more efficient to broadcast the vector than to construct alias lists and perform combinations.

The goal of parallel decomposition is to partition the directed graph into P regions, one region for each compute node. A matrix has structure if the decomposition can be done in such a way that the regions exhibit good locality. Good locality means the following:

- Most edges of the corresponding directed graph are internal to a region. That is, most field nodes in a region are not aliased.
- The number of other regions that any one region connects to is small compared to *P*. That is, the members of an alias group belong to only a few compute nodes, often only two.

The aliasing method works best when the number of messages is kept as low as possible. It is also better to have a few long messages than several shorter ones. Both of the above characteristics of good locality are not present in the NAS matrix. As will be seen, most field nodes are aliased, and most are aliased to all the other compute nodes.

The NAS CG Benchmark

This section describes the NAS CG benchmark [3]. This benchmark provides a matrix whose nonzero elements are randomly distributed throughout the matrix. What is described here is an older version of the benchmark than is currently offered. Refer to Appendix D, "The NAS Benchmark," for a detailed description of the NAS CG benchmark.

The Numerical Aerodynamic Simulation (NAS) Conjugate Gradient (CG) benchmark uses the power method to find an estimate of the smallest eigenvalue of a symmetric, positive definite, sparse matrix ***A***. The major part of the benchmark is spent in solving a matrix equation by the conjugate gradient method. The solution is performed 15 times, each time with different right-hand sides.

In addition to being symmetric, positive definite, and sparse, ***A*** is of order 14000 and has 1 853 104 nonzero elements. This comes out to about 132 nonzero elements in a row. The benchmark provides a Fortran program that generates ***A***. The benchmark also provides a test program whose ***A*** has order 1400. The matrix ***A*** has a random pattern of nonzero values.

Partitioning the Matrix

A simple row decomposition is used. Similar results are obtained for a column decomposition. In a row decomposition, the matrix is partitioned by rows among the compute nodes. For example, consider the sample NAS CG matrix partitioned over six compute nodes. This matrix has 1400 rows. Compute nodes 0 and 1 each get 234 rows, and compute nodes 2 through 5 each get 233 rows.

The difficulty is that when the NAS CG matrix has a row decomposition, just about every field node is aliased.

Determining the Location of Field Nodes

Now that the rows are distributed, it remains to determine on which compute nodes the field nodes reside. Recall that the conjugate gradient method requires a matrix-vector multiplication, $\boldsymbol{y} = \boldsymbol{Ax}$. Because of the row decomposition, every compute node starts off with a particular set of vector components. For example, if a compute node has rows k through l, then it also has vector components y_k through y_l and x_k through x_l.

When a compute node updates one of its y components, it needs access to some of the $\boldsymbol{x}$ components. Specifically, it needs access to those x components to which one of its $\boldsymbol{y}$ components is linked through a matrix element. This need can be seen from the following equation

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

where $k \le i \le l$ and n is the order of the matrix. This equation indicates that if a matrix element exists between y_i and x_j (a_{ij}) on a particular compute node (call it compute node n_1) and j is not one of the row numbers assigned to n_1, then x_j is aliased on n_1.

Although x_j does not lie within the range assigned to n_1, it does lie within the range assigned to some compute node (call it compute node n_2). Because the matrix is symmetric a_{ji} exists on n_2. This means that compute node n_2 must have access to x_i. Recall that the portion of the matrix-vector multiplication on n_1 did not require that x_i be aliased on n_1 because its index was within the range assigned to n_1; but the portion of the matrix-vector multiplication on n_2 does require a copy of x_i. So the x_i on n_1 belongs to an alias group, another member of which is the x_i on n_2.

To set up these alias groups, one must first assign some unique identification to the field nodes; they can be uniquely identified by global indices. These are the same global indices that identify the matrix elements. That is, a_{ii} is the self term for field node i; and a_{ij} is the matrix element that links field node i with field node j.

The algorithm for determining the location of the field nodes is as follows:

1. First note that, for the NAS matrix, the diagonal elements are nonzero. This means that the compute node that is assigned a row definitely has the vector component whose index is the same as the row; that is, the compute node

that has a_{ii} has the vector component i.

2. If the compute node has a_{ij}, then, field nodes i and j are also present on the compute node. This is because, in the aliasing method, there are no edges without attached field nodes in the portion of the directed graph on a compute node.

   If the compute node has a_{ij}, but not a_{ji}, then j is not within the range of row indices assigned to the compute node. The field nodes i and j must be present on that compute node; hence, j is aliased. It turns out that i must also be aliased, as is shown in the next paragraph.

   If the compute node does not have a_{ji}, then a_{ji} exists on some other compute node because the matrix is symmetric and a_{ij} being nonzero means a_{ji} is also nonzero. On that compute node, i is not within the row indices assigned to it. Because this compute node needs vector component i, i is aliased.

The result is that i and j are aliased on both compute nodes. This method determines, for a particular compute node, what vector components are aliased on that compute node. No communication is involved. The method does not determine the compute nodes to which the vector element is aliased. Communication is required for that information. How to determine what compute nodes belong to what alias groups was previously described in "The Alias List" on page 66.

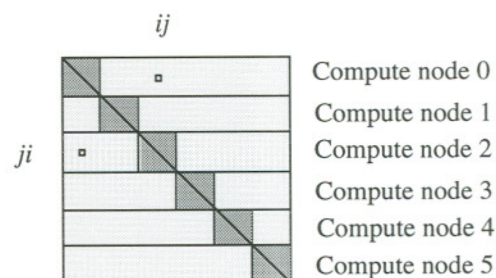
To see graphically what vector components are aliased, refer to Figure 6.1. The matrix in this figure is shown with a row decomposition over six compute nodes. The portion of rows assigned to a compute node has both darkly shaded and lightly shaded portions. The darkly shaded portion designates the area where the compute node has both cross-terms, a_{ij} and a_{ji}. The lightly shaded portion designates the area where the compute node has only one of the cross-terms, a_{ij}. If the value of this element (the a_{ij} in the lightly shaded portion) is nonzero, then the compute node must have copies of both vector elements i and j, and both belong to alias groups.

When is an alias *not* formed? Again consider the column portions in the lightly shaded regions of the matrix in Figure 6.1. If the portion of column j assigned to compute node 0 has all zero elements, then compute node 0 need not construct an alias for field node*s* i and j.

Note that throughout this discussion, the indices have been global over the entire set of compute nodes. To form an alias list, it is necessary that each field node have some unique identification. Otherwise there is no way to determine if it is duplicated on other

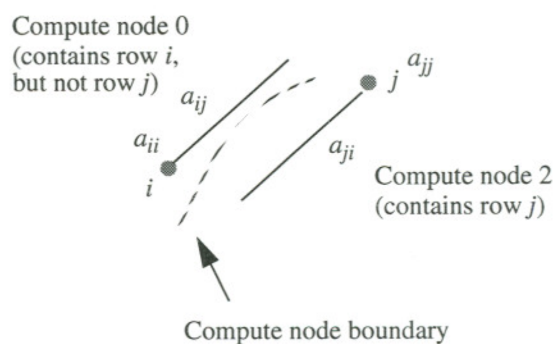
The off-diagonal terms in the darkly shaded areas designate internal edges in the directed graph. The small square box designates an off-diagonal term a_{ij} whose transpose is on a different compute node.

In the figure, the small square box designates a matrix element on compute node 0 whose transpose is on compute node 2.

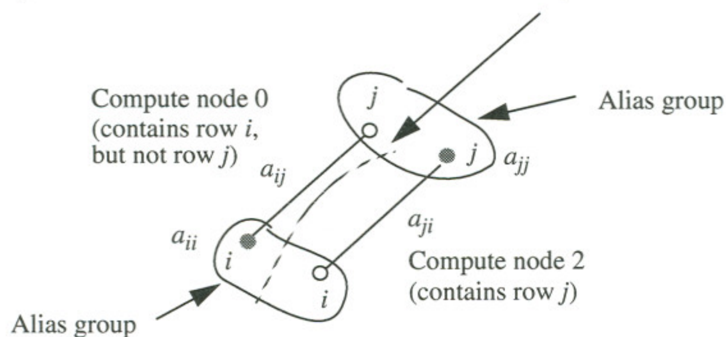


If a_{ij} is nonzero on compute node 0, then a_{ji} is also nonzero on compute node 2. Vector component j must be aliased on compute node 0, and vector component i must be aliased on compute node 2. Hence, both vector components i and j belong to alias groups on both compute nodes.

Step 1. The compute node that is assigned a row definitely has the vector component whose index is the same as the row; that is, the compute node that has a_{ii} has the vector component i.



Step 2. If the compute node has a_{ij}, but not a_{ji}, then j is aliased.



Step 2. If the compute node has a_{ji}, but not a_{ij}, then i is aliased.

The result is that i and j are aliased on both compute nodes.

Figure 6.1. Formation of Alias Groups when Matrix is Partitioned by Rows

compute nodes. For the NAS CG benchmark, this unique identification is the global index. For the actual calculation, local indices are used; the global index is used only to construct the alias list. Other types of problems may have other ways of achieving unique identification of field nodes. Physical problems may use the (x,y) coordinates of a field node.

Example Partitioned over Six Compute Nodes

When the sample NAS CG problem is partitioned over six compute nodes, compute node 0 has all zero elements in only one column, namely column 410. Consequently, of the possible 1400 field nodes, compute node 0 must have copies of 1399 of them! Every compute node has some columns with all zero elements. For the sample NAS problem partitioned over six compute nodes, these columns are as follows:

compute node	0	1	2	3	4	5
columns with no nonzero elements	410	499 1373	190 241 402 431	1013 1295	241	905

Recall the definition of the secretary and its role when making a distributed vector coherent. This role was previously described in "Making a Distributed Vector Coherent" on page 66. As stated in that section, how the secretary is chosen is arbitrary.

Assume that one chooses as the secretary for an alias group, the compute node with the lowest node number that has a vector component in the alias group. This means that, for the sample NAS problem partitioned over six compute nodes, compute node 0 turns out to be the secretary node for most of the field nodes. Specifically, compute node 0 is the secretary for 1399 field nodes. It is not the secretary for field node 410—compute node 1 turns out to be the secretary for this field node.

Aliasing Method is Inefficient for the NAS Matrix

The aliasing method applied to the NAS matrix with a row decomposition is inefficient. It sends longer messages than are needed and performs unneeded additions.

To see why, once again consider the sample NAS matrix distributed across six compute nodes and look closely at what happens between compute node 0 and compute node 1.

What Happens During Matrix-Vector Multiplication

Consider the row partitioning for compute node 1. If the global indices start at 0, compute node 1 has rows 234 through 467; compute node 1 also has 1398 field nodes (1400-2, because two of its column ortions contain only zeros). But when compute node 1 performs its matrix-vector multiplication, only field nodes 234 through 467 get nonzero values. All the other field nodes get zero values. This is illustrated in Figure 6.2.

Now to see why compute node 1 sends longer messages than needed and performs unneeded additions, consider in more detail how compute node 1 makes its field nodes coherent.

Longer Messages Than Needed

Consider the matrix-vector multiplication $\boldsymbol{Ax=y}$ and what happens on compute node 1. To make $\boldsymbol{y}$ coherent, compute node 1 first sends all aliased components to the corresponding secretaries (which for all components but component 410), is compute node 0). Of the 1400 components in the vector, compute node 1 does not maintain copies of two of them and is itself the secretary for one of them. Hence, compute node 1 sends 1397 components to compute node 0. It turns out that most of the values (the 1397 vector components) that compute node 1 sends to compute node 0 do not actually need to be sent. Only those from 234 through 467 (excluding 410, for a total of 233) need to be sent because the others are zero.

Performs Unneeded Additions

After compute node 0 receives values from compute node 1 (and other compute nodes for which it is the secretary), compute node 0 performs a sum, which turns out to be also unneeded. A replacement would be sufficient.

Here's why only a replacement is needed. Compute node 0 has itself modified $\boldsymbol{y}$ components 0 through 233. So it adds those to the vector components it received from compute node 1. Positions 0 through 233 in that vector (the one it received from compute node 1) are zero because compute node 1 did not modify them. So compute node 0 is adding zero to its components 0 through 233; these are unneeded additions.

As part of the matrix-vector multiplication, compute node 1 has itself modified only vector components 234 through 467. But those components were not modified by compute node 0. So compute node 0 is adding zero to the components 234 through 467 that it received from compute node 1; this is again an unneeded addition. After the sum, compute node 0 sends the answers to the other compute nodes. It sends 1397 values to compute node 1.

The result is that the aliasing method blindly applied to the NAS matrix sends longer messages than are needed and performs unneeded additions. Figure 6.3 illustrates the preceding discussion.

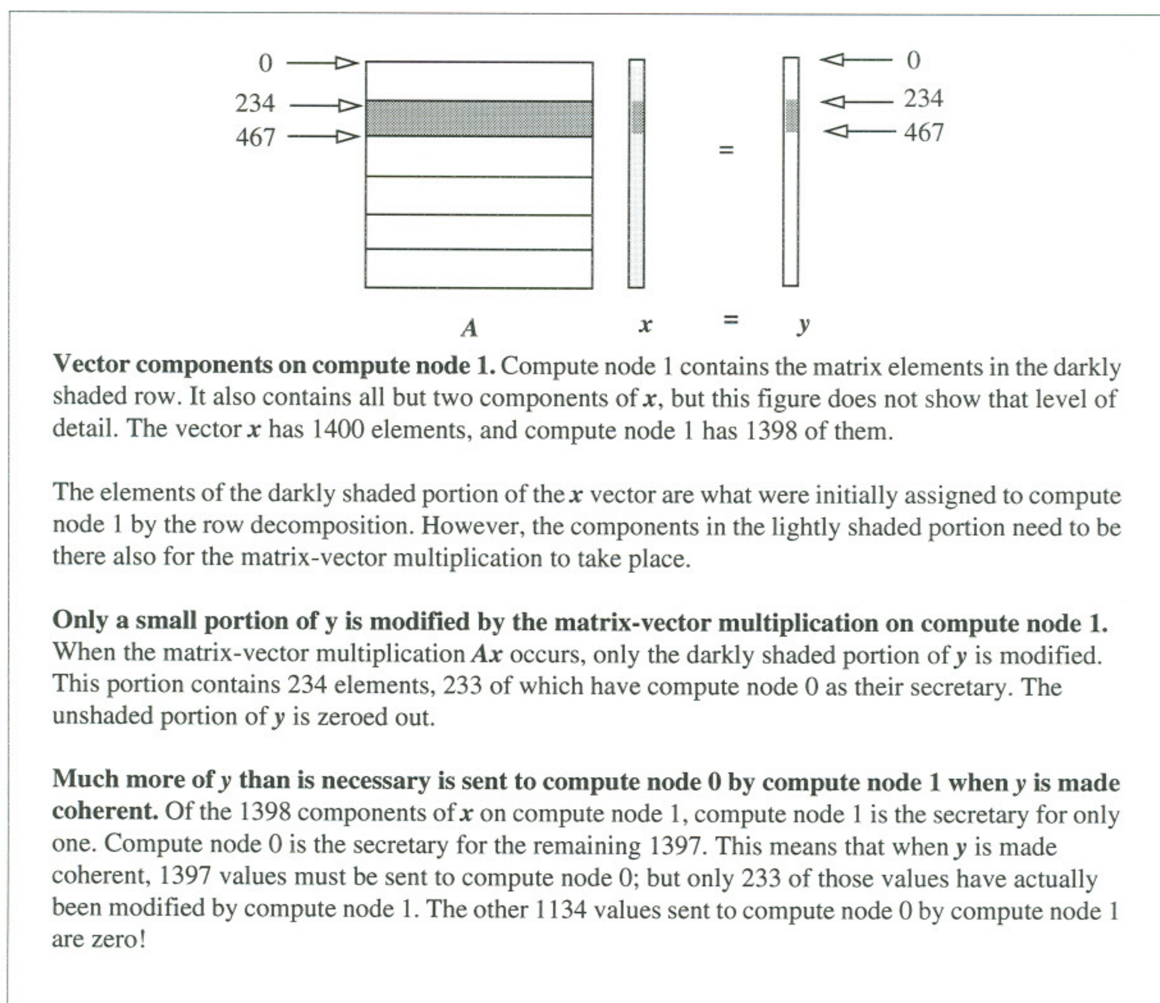


Figure 6.2. NAS Matrix-Vector Multiplication

The NAS Matrix as a Limiting Case

Because the NAS matrix is random, it presents a limiting case for the aliasing method. As seen in the previous discussion, the alias lists become overlong, and needless additions are performed.

Modifications can be made to the aliasing method to improve the performance. Figure 6.4 shows an example of a modification that turns out to be Method 1 of Lewis and van de Geijn [24].

Consider once again the matrix equation $Ax=y$. As before, distribute the rows of the matrix to the different compute nodes. In the figure, both vectors x and y are replicated on each of the compute nodes. After the matrix-vector multiplication a portion of y is modified on each compute node. Then, each compute node broadcasts its modified portion to the other compute nodes.

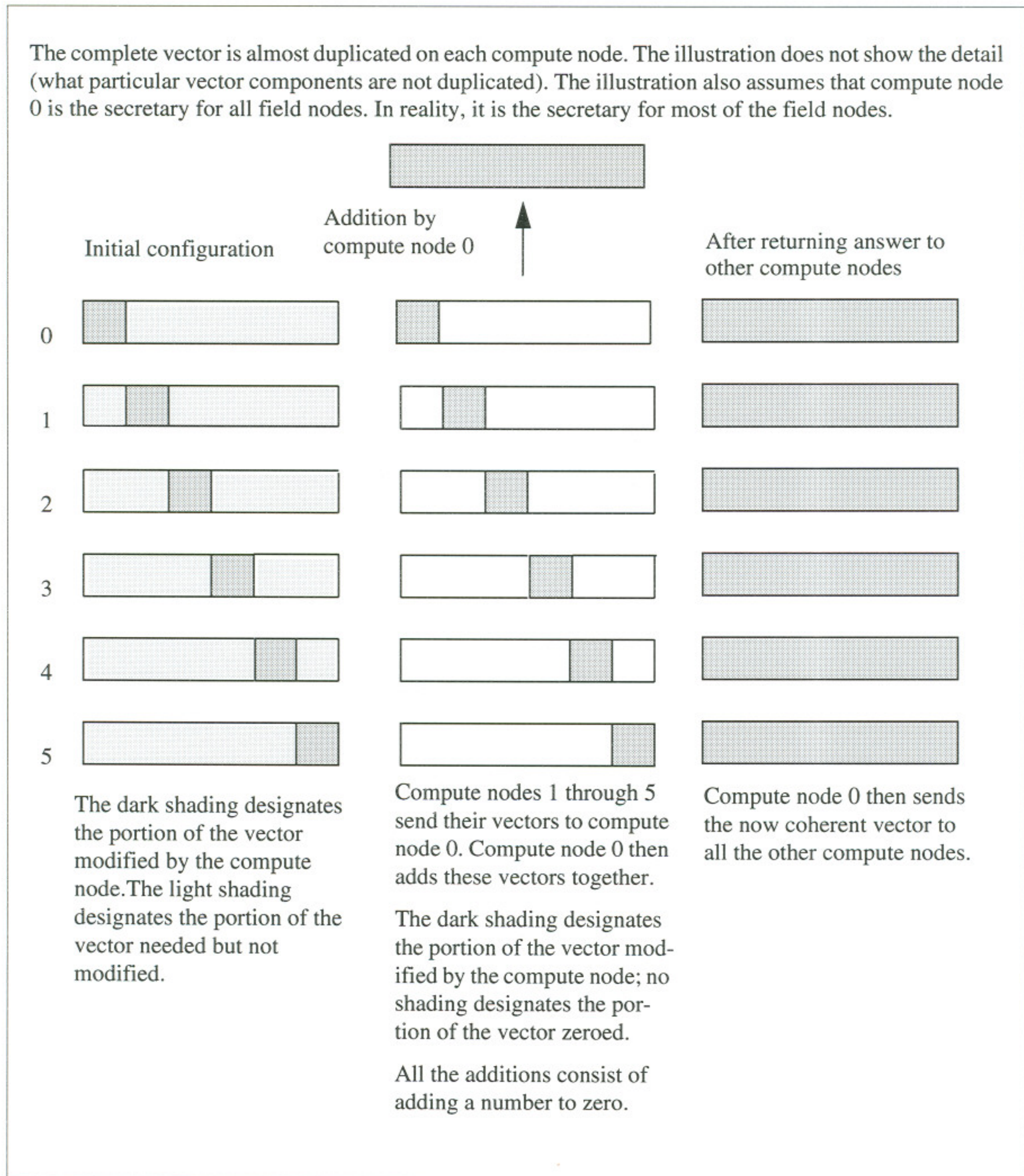


Figure 6.3. Updating Aliases when the Matrix is Distributed by Rows

Modifications Made for the NAS CG BenchMark

Modifications to the aliasing method were made to improve its performance when applied to the NAS matrix. These modifications make the aliasing method less general.

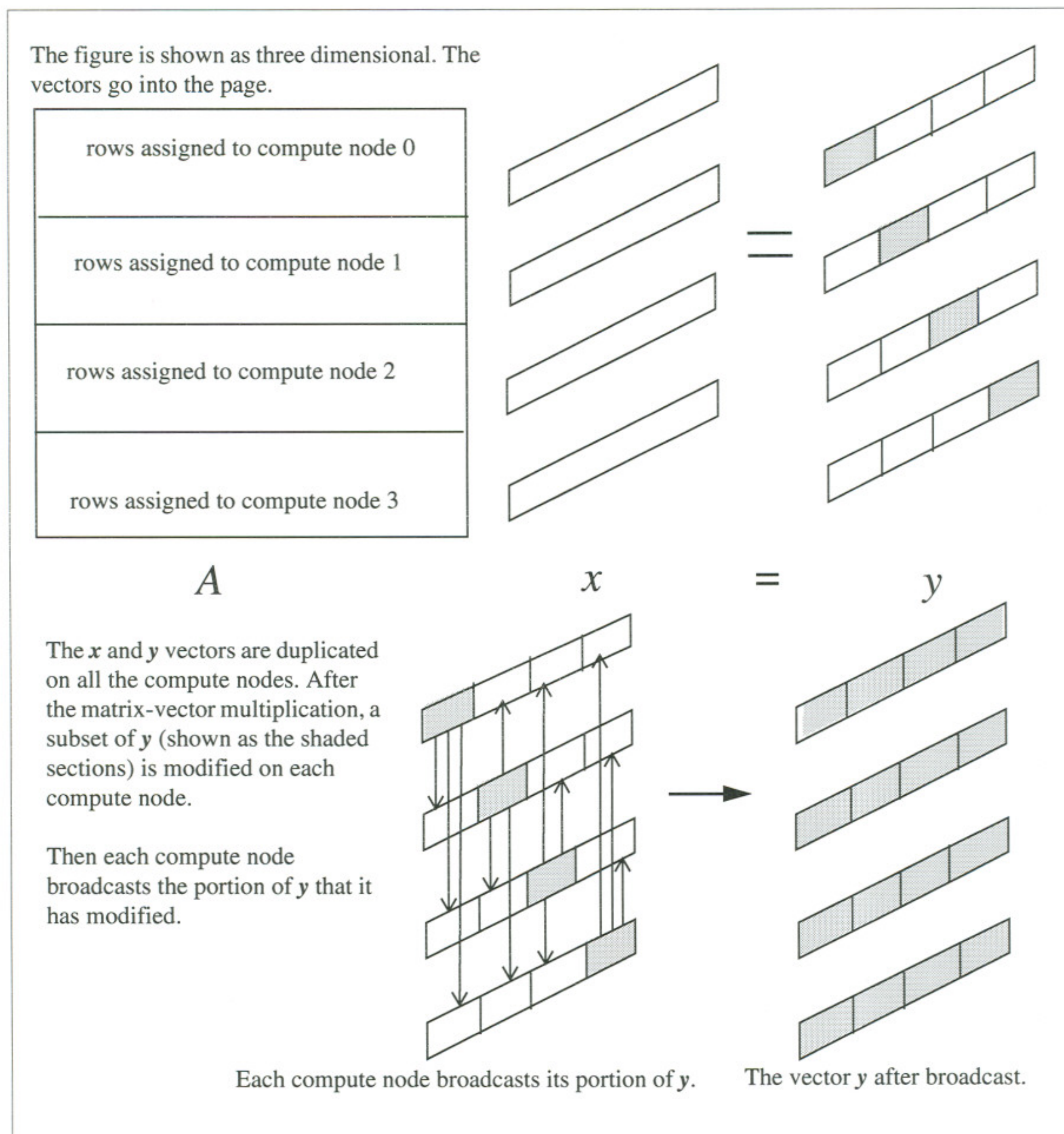


Figure 6.4. Method 1: Row Decomposition

These improvements include changing the method of determining what compute node becomes the secretary (called "balanced secretaries") and modifying the vector combination routine so that unneeded values are not sent (called "no zeros").

Balanced Secretaries

As was previously shown in the section "Example Partitioned over Six Compute Nodes" on page 90, if the secretary is chosen as the compute node in an alias group with the lowest node number, then compute node 0 ends up being the secretary for most of the

alias groups. Choosing the secretary this way is called low node number secretaries.

Using low node number secretaries creates a bottleneck when a distributed vector is made coherent. All the compute nodes send their portions of the vector to compute node 0, who then performs the coherence operation for overlapping portions, and then returns each now-coherent portion to the compute node it came from. This bottleneck can be avoided by using another method (called the "balanced secretaries" method) for choosing the secretary.

In the "balanced secretaries" method, if field node i is aliased, then the compute node that contains a_{ii} is defined to be the secretary. Because, with a row decomposition, every compute node knows what compute nodes contain what diagonal elements, this is easily done.

No Zeros

As was previously shown in the section "Performs Unneeded Additions" on page 91, with a row decomposition, only a portion of the vector is modified and only by the secretary. This is also illustrated in Figure 6.2. The consequence is that there is no need to do the initial send to a secretary, and there is no need for the secretary to perform the coherence operation.

The **vcomb()** routine described in "Example of Making a Vector Coherent" on page 69 was modified to remove the send to the secretary and the execution of the coherence operation. The modified **vcomb()** routine only performs the following actions: it sends its modified values to its receive list and then receives from its send list.

Figure 6.5 illustrates the **vcomb()** changes. Consider a matrix-vector multiplication. Compute node 1 contains matrix elements in the shaded portion of the matrix. It modifies vector elements in the darkly shaded portion of the vector; it is also the secretary for those elements. Compute node 1 needs the vector elements from the lightly shaded portion of the vector to perform the matrix vector multiplication, but compute node 1 does not modify any vector elements in the lightly shaded portion.

So after modifying the darkly shaded vector elements as shown in Figure 6.5, compute node 1 sends these elements to its receive list. (Because *only* compute node 1 modified these elements, it was not necessary to first receive contributions from non-secretary compute nodes; nor was it necessary to perform a coherence operation.) Then, in preparation for the next iteration, compute node 1 must receive from its send list.

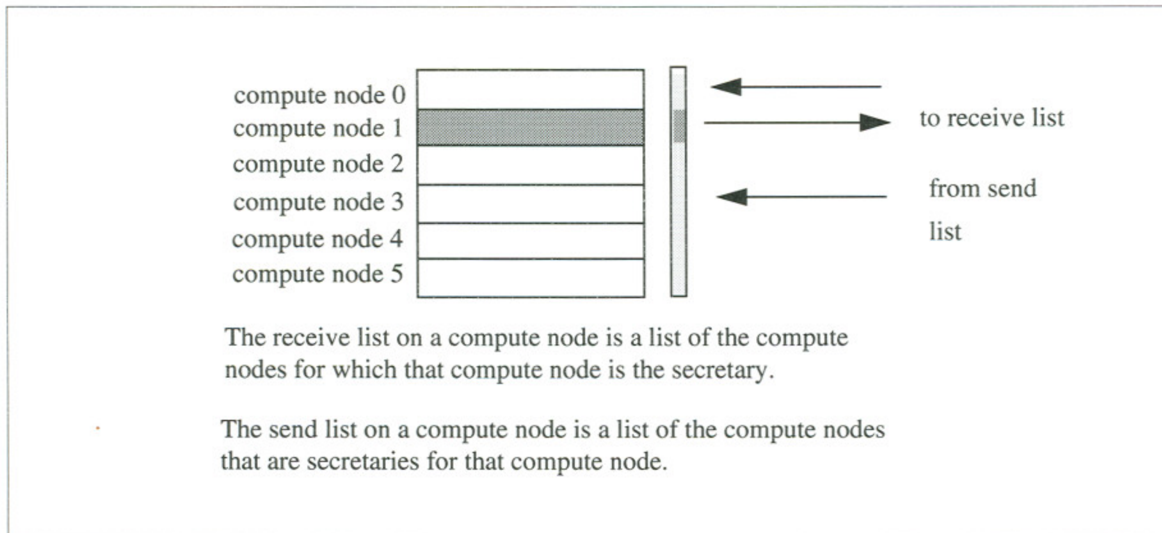


Figure 6.5. Modified vcomb() Routine.

This is to get the values of the components from the lightly shaded portion of the vector. The modified **vcomb()** has two steps: 1, send to the compute node's receive list; 2, receive from the compute node's send list.

When the compute node sends to its receive list, it does not necessarily perform a broadcast. If the portion of a column assigned to any compute node contains all zeros, then the vector component whose global index is the same as the global column index need not be recorded and maintained by the compute node.

For example, if the very lightly shaded portion of the matrix assigned to compute node 1 were all zeros as shown in Figure 6.5, then compute node 1 need not receive any vector components from compute node 0. Compute node 0 need not have compute node 1 on its receive list.

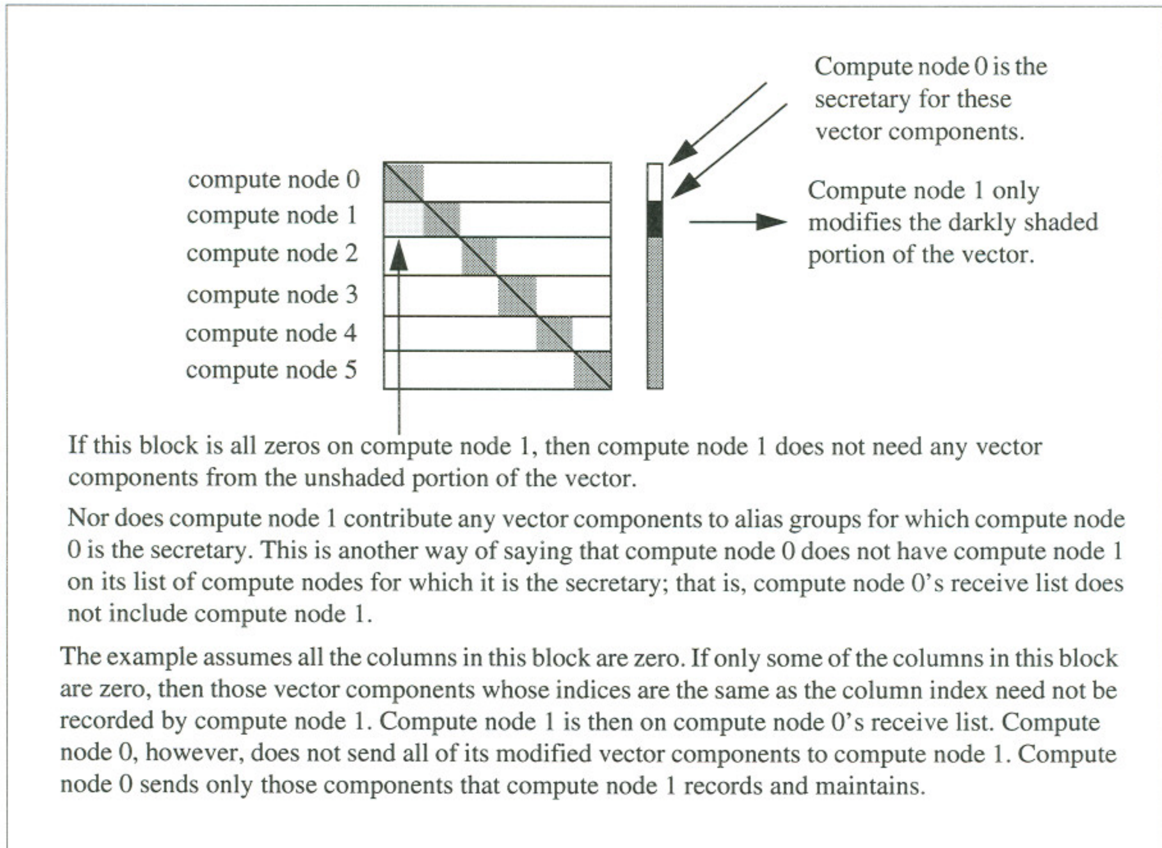


Figure 6.6. vcomb() Does not Necessarily Involve Broadcasts

Chapter 7 Performance

Introduction

Two specific problems were investigated using the aliasing method. This method was introduced in "The Aliasing Method" on page 18; its implementation was presented in Chapter 4, "Implementation of Aliasing."

- One was solving Laplace's equation on a rectangular (x,y) grid. Laplace's equation was previously defined and discussed in "Laplace's Equation" on page 5. The problem was set up as a finite element problem with triangular elements, and the conjugate gradient method was used to solve the resulting matrix equation.
- The other was running the The Numerical Aerodynamic Simulation (NAS) Conjugate Gradient (CG) benchmark. This benchmark provides the random matrix that was previously described in Chapter 6, "A Random Matrix" ; this random matrix has a random pattern of nonzero values and serves as a worst case example for the aliasing method. The benchmark itself is described in Appendix D, "The NAS Benchmark."

Code was written for and executed on a Paragon supercomputer from Intel Corporation, running Intel's Release 1.2 operating system.

Laplace's Equation on a Rectangular Grid

Meshes were constructed such that each compute node had 2500 finite elements. That is, the size of the problem was $2500P$ finite elements where P is the number of compute nodes. These meshes were generated by the program itself.

A finite element method was used to solve Laplace's equation with Dirichlet boundary conditions. The resulting matrix equation was solved by a conjugate gradient iteration. The method is implemented as an SPMD program with data parallel operations .

Laplace: How the Data Were Obtained

Data were obtained as follows:

1. Log into the Paragon supercomputer. A login shell then runs on the service node. The service node is physically just like a compute node, but it is in the service partition, has UNIX scheduling, and does not run parallel applications. Several users may share a service node, and there may be several service nodes.

2. Define a partition of compute nodes. These are the compute nodes that will be used to run the code. Use the **mkpart** command to do this. For example, the following command allocates a 10x10 partition called `ted`. The Paragon supercomputer used is configured such that a user has exclusive access to the nodes of a user-defined partition.

```
% mkpart -sz 10x10 ted
```

3. Load and run the program on the defined partition. The program requests the number of compute nodes in the *x* and *y* directions. (This input must be the same as the defined partition.) The program also requests the number of finite elements for the entire unpartitioned grid. The user response, 500x500, means that there are 50x50 elements per compute node for a 10x10 partition. If the partition had been 10x7, the specification would have been 500x350 finite elements on a side. The program is called `mdb`, and the letters do not mean anything.

```
% mdb -pn ted
Number of compute nodes on a side (x y): 10 10
Number of finite elements on a side (x y): 500 500
```

4. The program then creates the grid. The (*x,y*) dimensions of the entire grid are always 0 to1 in both directions. This means that when more compute nodes are used, the grid is finer. The more compute nodes that are used, the larger is the order of the coefficient matrix.

5. The program assigns the boundary conditions: 0 along the *x* boundary and 100 along the *y* boundary. The corners are constrained to be 100. This entails modifying the matrix elements as described in Appendix A, "The Finite Element Method."

6. The program calculates the matrix elements, then constructs the alias lists,

the send lists, and the receive lists.

7. The program performs the conjugate gradient iteration until the residual meets the prescribed epsilon, 10^{-5}. Because the total problem grows with the number of compute nodes, the number of iterations it takes to reach the same accuracy also increases. Theoretically, the conjugate gradient method yields the exact solution in *n* steps where *n* is the order of the matrix.

8. Finally the program prints out the statistics of the run for each compute node. Post-processing averages the appropriate numbers.

Laplace: Results for the (*x*,*y*) Problem

Figure 7.1 graphs the total flops vs. the number of compute nodes in the allocated partition. The speedup shown here is fixed-grain speedup, as defined in "Fixed-Grain Speedup" on page 17. The straight line shows how the total flops would increase if there were no dropoff due to increased communication.

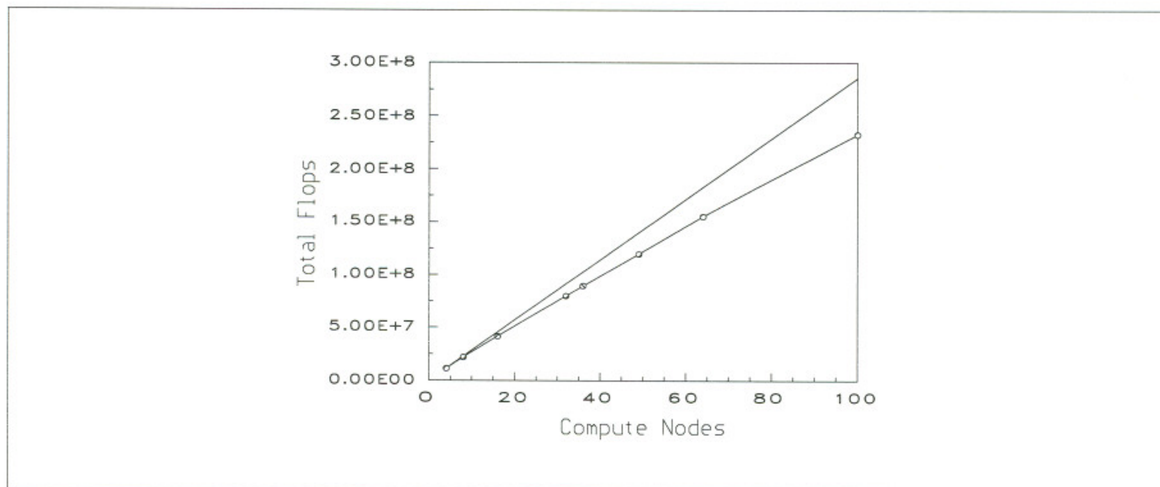


Figure 7.1. Total Flops vs. the Number of Compute Nodes in the Partition

Figure 7.2 graphs the total flops per compute node. Performance per compute node starts off at 2.9 Mflops for a four-node partition and ends up at 2.3 Mflops for a 100-node partition.This figure shows that the work done by each compute node remains essentially constant over the range of compute nodes, which is to be expected for a problem of fixed grain size.

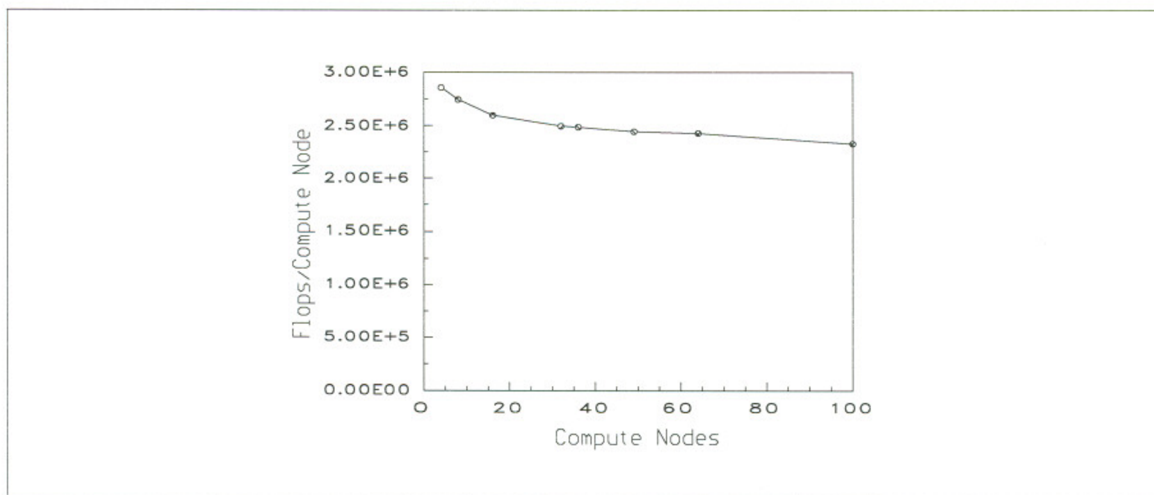


Figure 7.2. Total Flops/Compute Node vs. the Number of Compute Nodes in the Partition

Figure 7.3 graphs the time per iteration for three key operations vs. the number of compute nodes in the allocated partition. The three operations are the matrix-vector multiplication, the dot product (the time is for both dot products in each iteration), and the vector coherence.

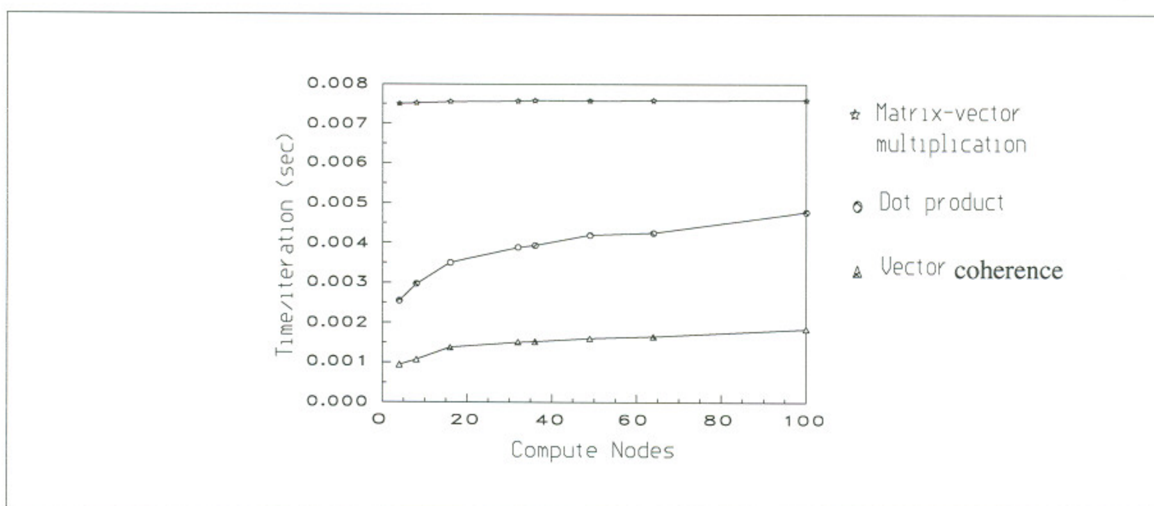


Figure 7.3. Time/Iteration vs. the Number of Compute Nodes in the Partition

Figure 7.3 shows that the matrix-vector multiplication remains flat. There is no communication involved in this operation and each compute node is doing the same amount of work for each run. The figure also shows that the time for a dot product increases with the number of compute nodes. The dot product is formed locally on each node and then the local values are added together. The Paragon supercomputer operating system provides a global call **gdsum()** that constructs a global sum. When **gdsum()** is

executed, each compute node contributes a floating point value to the sum, and each compute node ends up with the answer. The time for a dot product understandably increases with the number of compute nodes because the global operation involves more message passing.

Finally, Figure 7.3 shows that the coherence time increases slightly with the number of compute nodes; that is, as the number of compute nodes increases, the time it takes to make a vector coherent also increases. The reason for this node dependence is not readily apparent. A possible explanation for the increase of coherence time with larger partitions is just the increase in overall message traffic and ensuing contention that occurs in such cases.

The section "Laplace: Message Passing in the (x,y) Problem" on page 102 shows that, as more compute nodes are added, the amount of message passing per compute node does not increase. Although the total number of messages increases, the size and number of messages per compute node remains the same. The section "Laplace: Node Dependence in Vector Coherence" on page 104 describes in detail the care that was taken to have no node dependence in the coherence routine, **vcomb()**.

The data for the (x,y) grid problem are shown in Table 1. These are the data that are graphed in Figures 1, 2, and 3.

Table 7.1. Data for the (x,y) Grid Problem

Compute Nodes	Iterations	Combination/ iteration	Matrix-Vector Multiplication/ Iteration	Dot Product/ Iteration	Total Flops (Megaflops)
4	166	0.000946	0.007498	0.002550	11.424
8	288	0.001073	0.007516	0.002974	22.1962
16	327	0.001378	0.007553	0.003498	41.5697
32	513	0.001510	0.007567	0.003877	79.876
36	483	0.001526	0.007582	0.003933	89.474
49	561	0.001604	0.007575	0.004185	119.77
64	636	0.001651	0.007584	0.004245	155.39
100	790	0.001842	0.007590	0.004775	232.50

Laplace: Message Passing in the (x,y) Problem

Because of the fixed-grain size, each compute node sends and receives the same number of messages; but with more compute nodes, there are more of those messages.

Consider the following stages of making a vector coherent:

1. Compute nodes send to their send lists and receive from their receive lists.
2. Secretaries perform the coherence operation.
3. Compute nodes send to their receive lists and receive from their send lists.

A compute node is assigned a square of 51 finite element nodes (50 finite elements) on a side. This means that, during the first stage of making a vector coherent, an interior compute node receives three messages and sends three messages, as illustrated in Figure 7.4. The compute node receives from the compute nodes for which it is the secretary (its receive list) and sends to the compute nodes that are its secretaries (its send list). During the second stage, its also sends three messages; it sends to its receive list and receives from its send list.

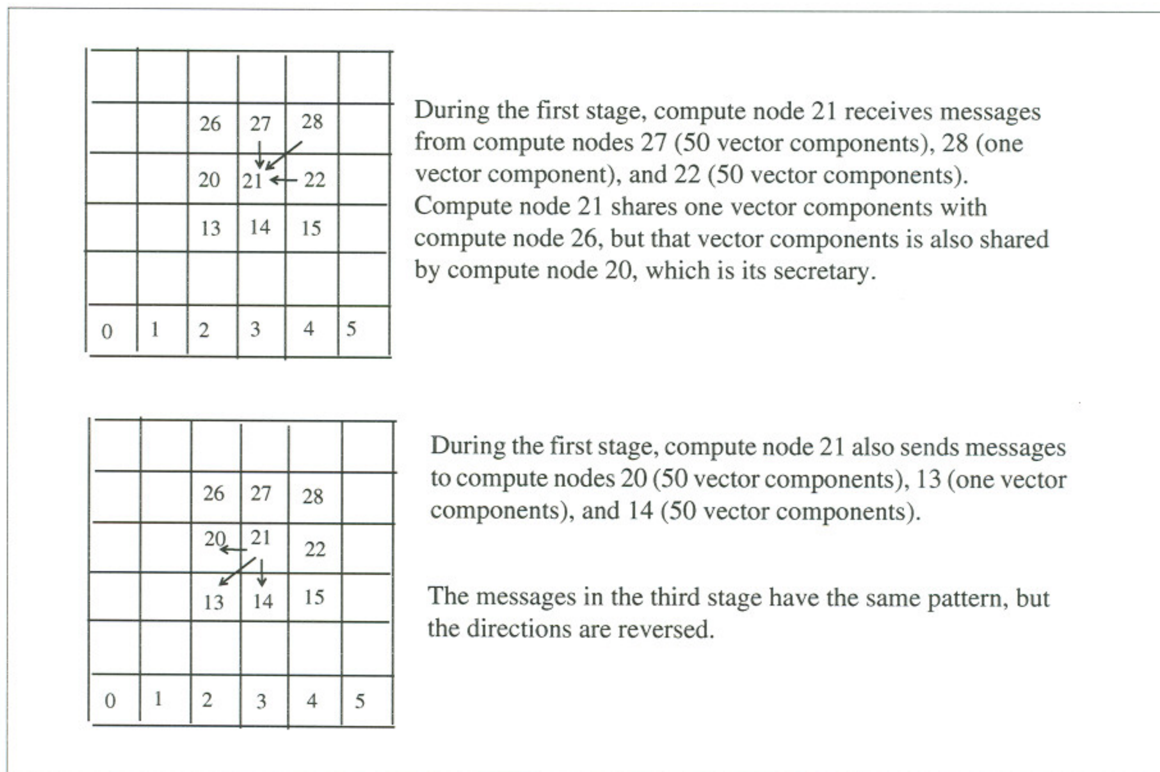


Figure 7.4. Messages in First Stage of Making a Vector Coherent

During the first stage, an interior compute node receives one vector component (a **double**) from the compute node to its top right, 50 vector components from the compute node to its top, and 50 vector components from the compute node to its right. One received message is eight bytes; the other two are 450 bytes each. The compute node sends three messages. It sends one vector component to the compute node to its bottom left, 50 vector components to the compute node to its bottom, and 50 vector components

to the compute node to its left. One sent message is eight bytes; the other two are 450 bytes each.

Laplace: Node Dependence in Vector Coherence

Early versions of **vcomb()** did have a node dependence. They used arrays whose dimensions were the number of compute nodes. **vcomb()** had a number of **for** loops that stepped through these arrays.

For example, the array *out_sendlist*[] was used by **vcomb()** to access its send list. *out_sendlist*[] is an array of pointers. If *out_sendlist*[*i*] is `NULL`, then the compute node did not send to compute node *i*; compute node *i* was not on that compute node's send list. If the compute node did send to *i*, then *out_sendlist*[*i*] contained a pointer to the data to be sent. An early version of **vcomb()** stepped through all the entries of *out_sendlist*[], ignoring those that were NULL and sending a message to the compute node whose node number was the index when the entry was a message pointer.

The latest version of **vcomb()** uses an *out_sendlist*[] that is only as large as the number of receiving compute nodes. Another array *send_corres*[] contains the node numbers of those compute nodes. Using *out_sendlist*[] in conjunction with *send_corres*[] allows the **for** loop to go only over the compute nodes to be sent to and not over all the nodes in the partition. For example, *send_corres*[*i*] contains the number of the compute node to which the current compute node will send the data pointed to by *out_sendlist*[*i*]. Making this change did slightly flatten the graph of coherence time/iteration vs. number of compute nodes.

In addition, there was some concern about efficient cache use. Early versions of **vcomb()** allocated a buffer for each message. These allocations occurred outside the iteration (the same buffers were used within the iteration); but there was no guarantee that these buffers were near each other in memory. Since every one was used during each iteration, it was desirable to make them contiguous. Increased locality for data structures used frequently increases the probability of taking cache hits rather than misses.

So the latest version of **vcomb()** allocates a large chunk of memory for its buffers. The pointers in *out_sendlist*[] and other arrays are now offsets into that large chunk. Making this change further flattened the graph of coherence time/iteration vs. number of compute nodes, but there is still an increase for small numbers of compute nodes and a gradual increase for large numbers of compute nodes.

The NAS CG Benchmark

The Numerical Aerodynamic Simulation (NAS) Conjugate Gradient (CG) benchmark comes in two versions: the small version (also called the debug or test version) and the real version. Recall from Chapter 6, "A Random Matrix," that the NAS matrix contains a random pattern of nonzeros. In the section "Aliasing Method is Inefficient for the NAS Matrix" on page 90, it was shown how poor performance is expected for NAS CG benchmark when implemented with the aliasing method.

NAS CG: How the Data Were Obtained

As with the (x,y) grid, a partition is allocated, and the program loaded on that partition. The program requires no user input.

```
% mkpart -sz 10x10 ted
% nas -pn ted
```

NAS CG: What the Benchmark Measures

The NAS benchmark has an outer loop that is executed 15 times and an inner loop that is executed 25 times. At the end of each outer loop iteration, a matrix-vector multiplication is performed. In addition to the matrix-vector multiplication, a **vcomb()** must be performed to make the answer coherent. The time for the 15 matrix-vector multiplications and the 15 **vcomb()**'s are averaged and graphed. Pseudocode for the NAS CG benchmark is as follows:

```
Start main timer
   do 15 times MAIN
      do 25 times CG iteration to solve Ax=b
         matrix-vector multiplication
         make the vector coherent
         two dot products
      enddo CG

      start matvec timer
         matrix-vector multiplication
      end matvec timer; accumulate matvec time

      start vcomb timer
         vector combination
      end vcomb timer; accumulate vcomb time

   enddo MAIN
End main timer
```

NAS CG: The Small Version

Figure 7.5 shows the performance for the small (debug) version of the NAS CG benchmark. The upper curve shows the performance for the program containing the balanced secretaries/no zeros **vcomb()**, and the lower curve is for the program containing the standard **vcomb()**. The section "Balanced Secretaries" on page 94 described the balanced-secretaries modification to **vcomb()**, and the section "No Zeros" on page 95 described the no-zeros modification to **vcomb()**. Note that the version with the modified **vcomb()** shows higher performance, even increasing with the number of compute nodes, until presumably the increased communication, fixed problem size, and resulting message contention do cause the curve to turn over. The section "NAS CG: Message Passing" on page 108 characterizes message passing for the NAS CG benchmark.

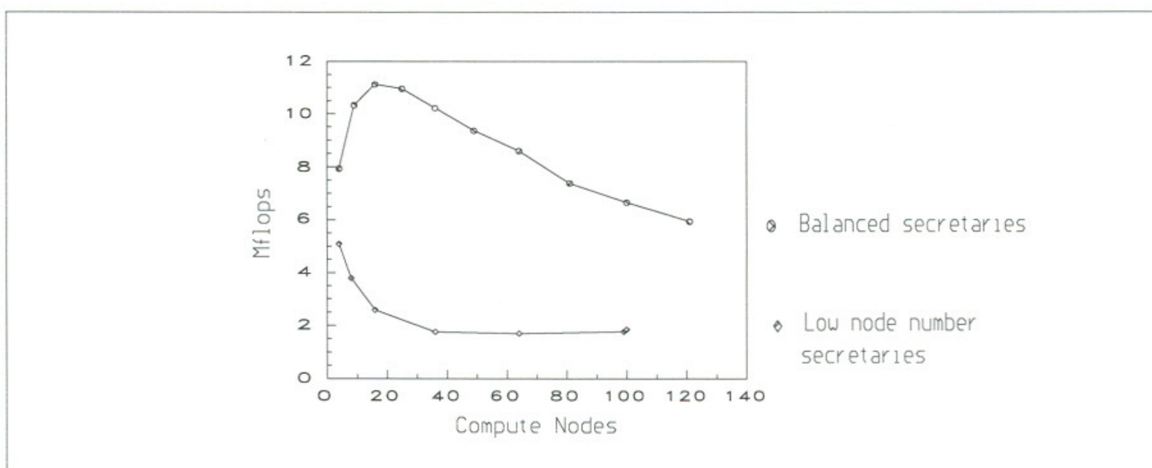


Figure 7.5. Balanced Secretaries/No Zeros and Low Node Number Secretaries

Figure 7.6 shows the matrix-vector multiplication time. Because this is a fixed-size problem and this time does not involve any communication, the matrix-vector multiplication time understandably decreases as the number of compute nodes increases and is the same for both versions of **vcomb()**.

Figure 7.7 shows the **vcomb()** time for the two versions of **vcomb()**. The balanced secretaries/no zeros **vcomb()** still increases with the number of compute nodes, but it is clearly faster than the standard **vcomb()**.

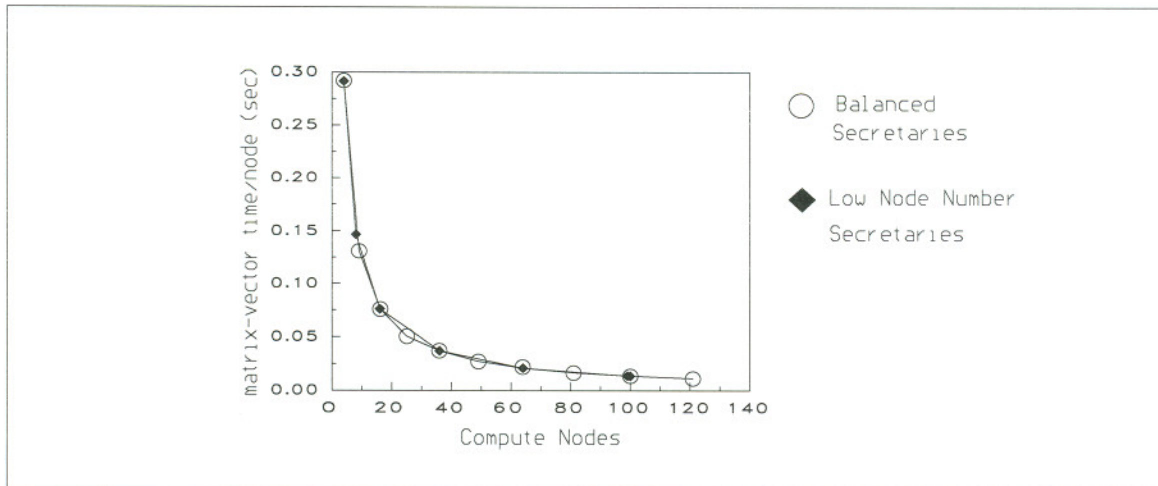


Figure 7.6. Matrix-Vector Time vs. Compute Nodes (small version)

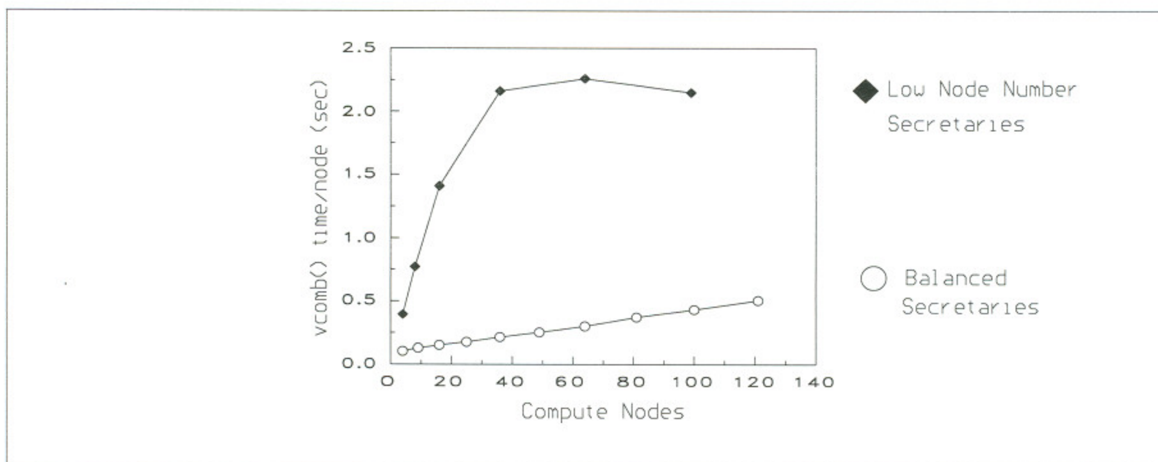


Figure 7.7. vcomb() Time vs. Compute Nodes (small version)

The Real Version

The real version of the benchmark has a matrix of order 14000 instead of 1400. Figure 7.8 shows the performance in Mflops for the real version of the NAS benchmark. This value increases with the number of compute nodes up to a partition size of about 50 compute nodes, at which point it decreases before increasing again at around 70 compute nodes.

Figure 7.9 shows the performance per compute node. This value decreases until it levels off at a partition size of about 70 compute nodes.

Figure 7.10 shows the matrix-vector multiplication time and the **vcomb()** time. The matrix-vector multiplication time decreases as the number of compute nodes increases. This is to be expected because the problem is of a fixed size. The shape of this curve is the same as that for the matrix-multiplication time for the debug version. The

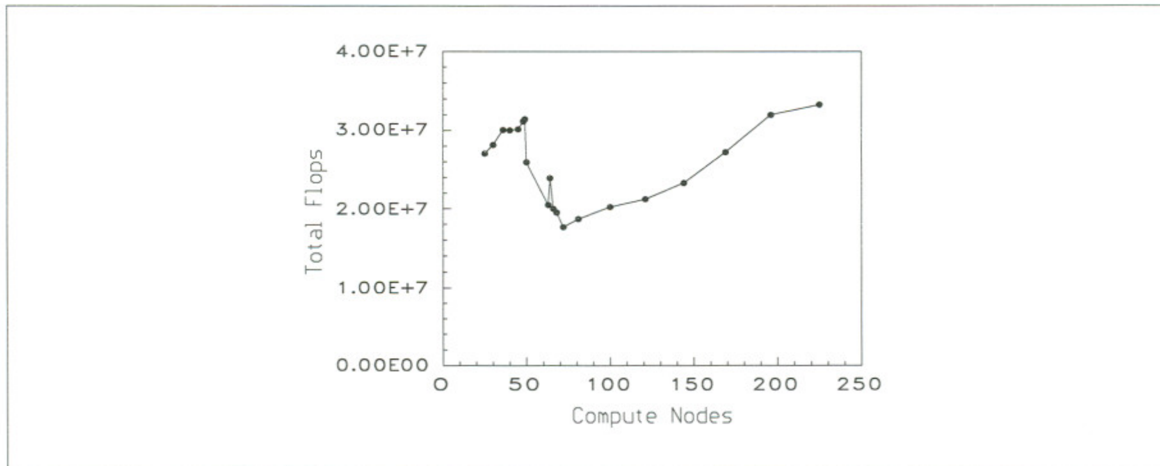


Figure 7.8. Total Flops vs. the Number of Compute Nodes in the Partition

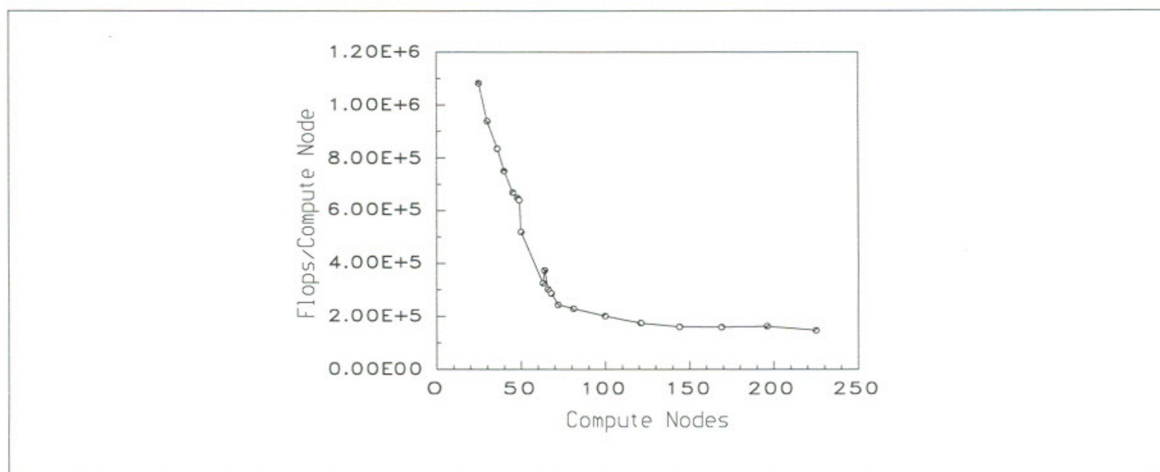


Figure 7.9. Total Flops/Compute Node vs. the Number of Compute Nodes in the Partition

vcomb() time increases slightly as the number of compute nodes increases. As before with the small NAS CG version, the increase in **vcomb()** time is attributed to increased message traffic. The section "NAS CG: Message Passing" on page 108 characterizes the NAS CG benchmark.

The data for the real version of the NAS CG benchmark are shown in Table 1. These are the data that are graphed in Figures 8, 9, and 10..

NAS CG: Message Passing

To get a general idea of the magnitude and number of messages, consider the following argument. Assume that most field nodes are aliased and also that they are aliased to every compute node. Then, each compute node sends and receives n-1 messages (there are n compute nodes).The messages have length

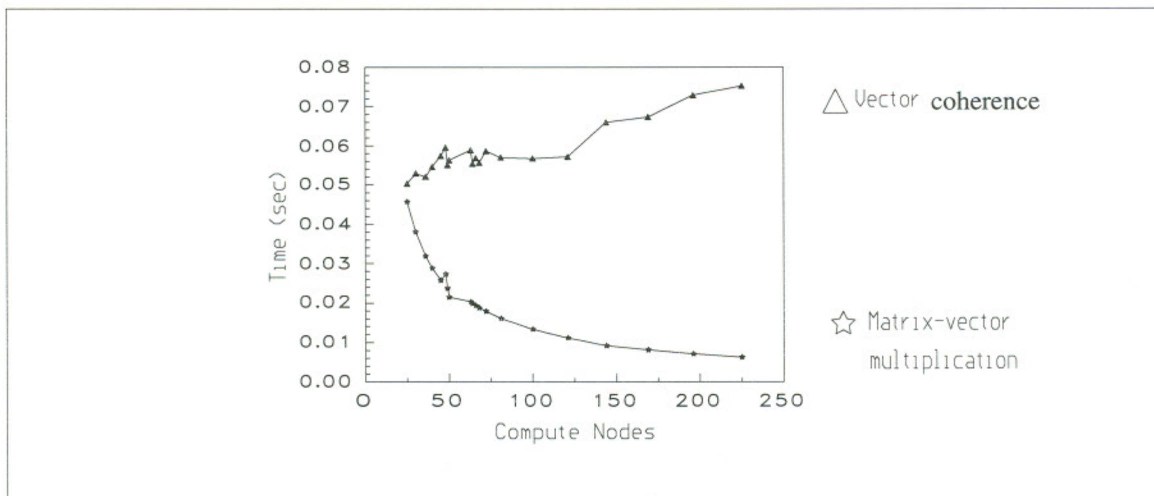


Figure 7.10. Time/Iteration vs. the Number of Compute Nodes in the Partition

$$\frac{(\text{order_of_matrix}) \cdot (\text{size_of_double})}{(\text{bytes/Kbyte}) \cdot n}$$

The small version has a matrix of order 1400; the large version has a matrix of order 14000. Then,

$$\frac{1400 \cdot 8}{1024 \cdot n} = \frac{11}{n}\text{Kbytes} \qquad \text{message length for small version}$$

$$\frac{14000 \cdot 8}{1024 \cdot n} = \frac{110}{n}\text{Kbytes} \qquad \text{message length for real version}$$

As the number of compute nodes increases, the size of each message decreases; but the number of messages increases. If $(n\text{-}1)/n$ approximates n, the total number of bytes participating in message traffic stays about the same.

$$(\frac{1400 \cdot 8}{1024 \cdot n}) \cdot (n-1) \approx 11\text{Kbytes} \qquad \text{total message traffic for small version}$$

$$(\frac{14000 \cdot 8}{1024 \cdot n}) \cdot (n-1) \approx 110\text{Kbytes} \qquad \text{total message traffic for real version}$$

In summary, there are n-1 messages of length 11/n Kbytes (sample) and 110/n Kbytes (real) where n is the number of compute nodes. The total number of bytes participating in message traffic remains the same at about 11 Kbytes (sample) and 110 Kbytes (real). Although the size of messages decreases, the number of messages increases.

Table 7.2. Data for the NAS CG Benchmark

Compute Nodes	Combination Time	Matrix-Vector Multiplication Time	Total Flops (Megaflops)	Megaflops/ Compute Node
25	0.050283	0.045734	27.061900	1.0824
30	0.052937	0.038023	28.148100	0.93827
36	0.052026	0.031847	30.062300	0.83506
40	0.054602	0.028800	30.016800	0.75042
45	0.057426	0.025749	30.154800	0.67010
48	0.059532	0.027286	31.153400	0.64902
49	0.055096	0.023765	31.420300	0.64123
50	0.056313	0.021501	25.927800	0.51855
63	0.058888	0.020371	20.489900	0.32523
64	0.055395	0.019865	23.941550	0.37408
66	0.056825	0.019348	20.003200	0.30307
68	0.055633	0.018801	19.512500	0.28694
72	0.0586633	0.017965	17.632800	0.24490
81	0.057009	0.016108	18.682300	0.23064
100	0.056756	0.013354	20.210100	0.20210
121	0.057184	0.011179	21.235000	0.17549
144	0.065955	0.009206	23.288700	0.16172
169	0.067275	0.008164	27.215300	0.16103
196	0.072839	0.007144	31.977300	0.16314
225	0.075236	0.006265	33.258000	0.14781

Chapter 8 Conclusions

Introduction

This thesis has presented an aliasing method that can be used to develop iterative algorithms to solve large, sparse linear systems of equations on massively parallel computers. This aliasing method was introduced in "The Aliasing Method" on page 18. The details of its implementation were presented in Chapter 4, "Implementation of Aliasing."

The method was applied to two problems. One was Laplace's equation on a rectangular grid (see "Laplace's Equation" on page 5); the other was the NAS CG benchmark (see "The NAS CG Benchmark" on page 86). The method gives correct answers for both problems, but performs better for the Laplace problem.

Weakly-Coherent Shared Memory

The aliasing method provides a useful programming interface. With the aliasing method, the programmer need not manage the details of guard buffers and message passing. The aliasing method provides a simpler view, namely that of weakly-coherent, shared memory.

Efficient for Structured Problems

This thesis has shown that for structured problems, the aliasing method is efficient. Structured problems were shown to be those representing physical problems and amenable to finite element methods. An example is Laplace's equation on a rectangular grid. Their coefficient matrices exhibited good locality, as defined in Chapter 6, "A Random Matrix." The aliasing method was shown not to be efficient for matrices with a random nonzero pattern. An example is the NAS CG benchmark.

Alias groups were defined and used in the algorithm that made a distributed vector coherent. An important consequence of alias groups is that vector components not needed to make a distributed vector coherent are not part of a message. Only the information that needs to be sent is sent; and this information is only sent to the compute nodes that need it. There is some overhead in limiting the size and number of messages this way; and that is in essence why the method is inefficient for random matrices.

Local Indexing

Another important feature of the aliasing method is local indexing. The vector components of a distributed vector are not numbered globally. This means that grid refinement can proceed without any need for global renumbering. If one decides to make the grid denser on one compute node, one can do that without affecting the numbering on any other compute node.

No Distinguished Owner

Another characteristic of alias groups is that there is no distinguished owner. The alias group is symmetric. Each compute node that is a member of an alias group reads and writes the vector component it has contributed to the alias group without taking into account who the other members of the alias group are. All compute nodes in the alias group equally own the vector component. The alias group is not a guard buffer; it acts like weakly coherent shared memory. This distinguishes the aliasing method from the owner computes method (see "The Owner Computes Method" on page 59).

We did define a secretary for the alias group. Some operations on the alias group required us to single out an alias group member. But the secretary is not the owner of the vector component. The secretary is just a construct used to perform operations (called coherence operations) on the alias group.

Future Work

This investigation focused on the actual iterative computation. A desirable method, however, would have overall performance, including the initialization. Two areas are designated for improvement.

No optimization was attempted for the initialization. Constructing the alias lists, send lists, and receive lists can be time consuming. In addition, applying the boundary conditions was also surprisingly time consuming. The method used was that described in "Another Way of Accounting for Type 1 Boundary Conditions" on page 130.

Finally, because the thesis was focused on the alias groups and their needed communication routines, the actual matrix-vector multiplication would also benefit from some attention. The sparse storage scheme used was an easy one to manipulate and understand, but not the most optimal one.

We do believe that this thesis has shown the aliasing method to be a useful and desirable programming model for certain kinds of problems typical of real-world applications; and for that reason merits further study.

Appendix A The Finite Element Method

Introduction

This appendix presents the mathematical background of the finite element method. The finite element method is described here as a numerical method for solving partial differential equations. Specifically, the method is applied to two-dimensional, second order, linear partial differential equations. The treatment here follows that in Bickford [5]. The description of the Galerkin approach follows Sewell[32].

What is a Finite Element?

The domain of the solution is divided into a number of finite elements, and the total solution is the sum of the solutions in each of these finite elements.

For example, consider a two-dimensional surface, the (x,y) plane. The solution at each point in the plane is graphed as a height above this plane, with the total solution forming a solution surface. Consider the plane as divided into a number of finite elements, and choose the simplest two-dimensional element, the triangle. The vertices of the triangles are called finite element nodes.

The portion of the solution surface above a finite element is called a *solution segment* and is approximated as a plane. The complete solution is approximated as a number of planar segments, one above each finite element. Figure A.1 shows the portion of the planar solution surface over one of the triangles.

The solution over an element is expressed in terms of *elemental basis functions*. It is planar if these elemental basis functions are linear. Non-linear basis functions are not discussed in this thesis.

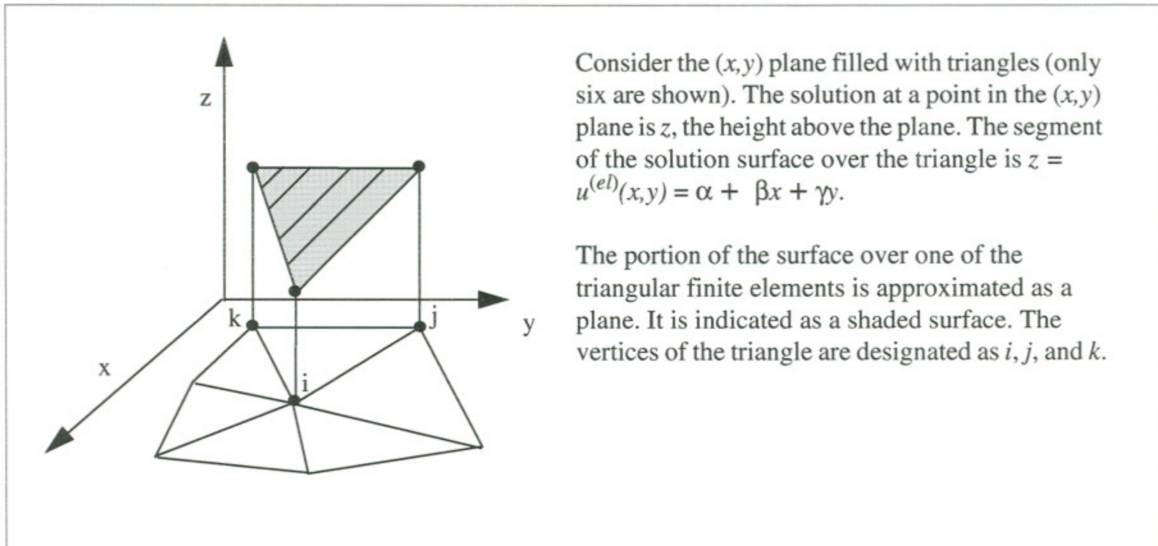


Figure A.1. The Solution Surface over One of the Triangular Elements

Elemental Basis Functions

The solution segment over a finite element is approximated as a plane. If the superscript *(el)* identifies a particular element labeled *el*, the solution segment can be written as

$$u^{(el)}(x, y) = \alpha + \beta x + \gamma y$$

where α, β, and γ are constants. To determine α, β, and γ, note that the equation must equal the values of $u^{(el)}(x,y)$ at the vertices of the triangle. The triangle has vertices *i*, *j*, and *k*. At vertex *i*, $u_e(x,y)$ is u_i; at vertex *j*, $u^{(el)}(x,y)$ is u_j; and at vertex *k*, $u^{(el)}(x,y)$ is u_k.

The values of $u^{(el)}(x,y)$ at each of the vertices result in three simultaneous linear equations. Each equation specifies $u^{(el)}(x,y)$ at the triangle's vertices.

$$u^{(el)}(x_i, y_i) = \alpha + \beta x_i + \gamma y_i = u_i$$

$$u^{(el)}(x_j, y_j) = \alpha + \beta x_j + \gamma y_j = u_j$$

$$u^{(el)}(x_k, y_k) = \alpha + \beta x_k + \gamma y_k = u_k$$

Use these equations to solve for α, β, and γ. Then, rearrange the equations so that $u^{(el)}(x,y)$ (the solution segment above the triangle) is written in terms of the values at the vertices (u_i, u_j, and u_k) and *elemental basis functions* (N_i, N_j, and N_k). The result is as follows:

$$u^{(el)}(x, y) = u_i N_i(x, y) + u_j N_j(x, y) + u_k N_k(x, y)$$

The elemental basis functions are defined as follows:

$$N_i = \frac{a_i + b_i x + c_i y}{2A^{(el)}} \qquad N_j = \frac{a_j + b_j x + c_j y}{2A^{(el)}} \qquad N_k = \frac{a_k + b_k x + c_k y}{2A^{(el)}}$$

where

$$a_i = x_j y_k - x_k y_j \qquad a_j = x_k y_i - x_i y_k \qquad a_k = x_i y_j - x_j y_i$$

$$b_i = y_j - y_k \qquad b_j = y_k - y_i \qquad b_k = y_i - y_j$$

$$c_i = x_k - x_j \qquad c_j = x_i - x_k \qquad c_k = x_j - x_i$$

and $A^{(el)}$ is the area of the elemental triangle calculated from the following determinant,

$$2A^{(el)} = \begin{vmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{vmatrix}$$

Figure A.2 illustrates the elemental basis functions N_i, N_j, and N_k. These functions are defined so that at vertex i (x_i,y_i), $N_i = 1$ and $N_j = N_k = 0$, at vertex j (x_j,y_j), $N_j = 1$ and $N_i = N_k = 0$, and at vertex k (x_k,y_k), $N_k = 1$ and $N_i = N_j = 0$

Define $\boldsymbol{N}$ as the vector whose components are the elemental basis functions.

$$\boldsymbol{N}^T = \left[N_i(x, y) \;\; N_j(x, y) \;\; N_k(x, y) \right]$$

where i, j, and k designate the vertices of the triangular finite element, and the superscript T stands for transpose, makeing $\boldsymbol{N}^{\mathrm{T}}$ a row vector. $\boldsymbol{b}^{(el)}$ and $\boldsymbol{c}^{(el)}$ are vectors defined as follows:

$$\boldsymbol{b}^{(el)T} = \left[b_i \;\; b_j \;\; b_k \right] \qquad \boldsymbol{c}^{(el)T} = \left[c_i \;\; c_j \;\; c_k \right]$$

where the superscript *(el)* indicates a vector defined on the finite element (the triangle). To define a vector on a finite element means that an component of the vector is defined on each finite element node of the finite element. When the finite elements are triangles, these vectors have three components.

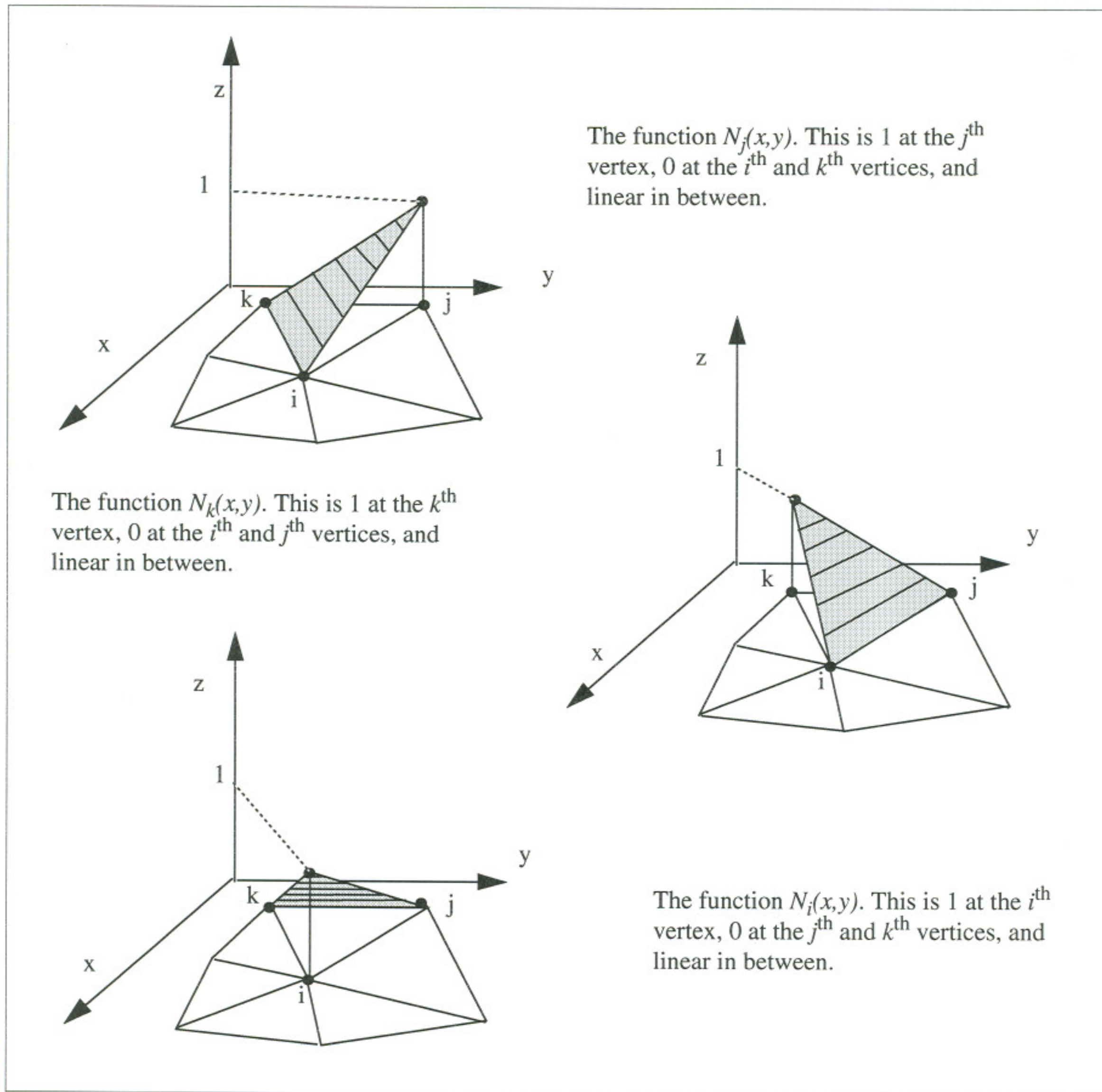


Figure A.2. The Elemental Basis Functions, $N_i(x,y)$, $N_j(x,y)$, and $N_k(x,y)$

The derivatives of N are

$$\frac{\partial N}{\partial x} = \frac{\boldsymbol{b}^{(el)}}{2A^{(el)}} \qquad \frac{\partial N}{\partial y} = \frac{\boldsymbol{c}^{(el)}}{2A^{(el)}}$$

Minimum Residual Method

The *method of weighted residuals* means that the residual must be orthogonal to N linearly independent weight functions where N is the number of finite element nodes. This is written as

$$\iint_D w_i(x, y) R(x, y) = 0$$

where w_i is the weight function, $R(x,y)$ is the residual, and i goes from 1 to N.

The residual is defined as follows. Arrange the equation to be solved so that it equals zero, substitute a function for the solution, and evaluate. How much the resulting value (called the *residual*) differs from zero is a measure of how good the solution is.

The weight function takes on specific forms to generate approximate solutions in specific ways.

The Weak Formulation

This section introduces the weak formulation. It shows how the method of weighted residuals can be combined with an integration by parts to weaken the continuity requirements on the solution. The result is an equation called the weak formulation that is later used to get the matrix equation.

With the *weak formulation*, a differential equation is satisfied in an average sense over some domain rather than everywhere. If the differential equation is written such that its right-hand side is zero, then an exact solution will always set the left-hand side to zero.

Multiply the left-hand side by some test function and then integrate over a domain. The resulting integral is zero for a series of test functions. In the equation,

$$\iint_D w(x,y)\,R(x,y)\,dA = 0$$

$R(x,y)$ is the actual differential equation and $w(x,y)$ is the test function.

To see how the weak formulation is developed, consider an example. Specifically, consider a two-dimensional Poisson's equation with Type 1 (Dirichlet) and Type 2 (Neumann) boundary conditions. Call the domain R and the boundary Γ which has two parts: Γ_1 and Γ_2. Γ_1 has Dirichlet boundary conditions and Γ_2 has Neumann boundary conditions.

Poisson's equation is

$$\nabla^2 u(x,y) + f(x,y) = 0$$ in R.

$$u(x,y) = r(s)$$ on Γ_1 where s is measured along the boundary. These Dirichlet conditions are also called *essential boundary conditions*.

$$\frac{\partial}{\partial n}u(x, y) = h(s) - \alpha(s)\,u(x, y)$$

on Γ_2 where s is measured along the boundary. Differentiation with respect to n means the derivative normal to the boundary and pointing out; that is

$$\frac{\partial}{\partial n}u(x, y) = \nabla\, \boldsymbol{u} \cdot \boldsymbol{n}$$

where $\boldsymbol{n}$ is a vector perpendicular to the boundary and pointing out of R. These are also called *natural boundary conditions.* They are sometimes called *suppressible boundary conditions* [23].

The test function $w(x,y)$ must have first partial derivatives defined in the domain D which includes R and Γ. It is also assumed that the test function is zero on the boundary Γ_1. Then, the equation for the weak formulation is as follows:

$$\iint_D w(x, y)\left(\nabla^2\, u(x, y) + f(x, y)\right)dA = 0$$

Remembering that D is the sum of R and Γ and noting that $w(x,y)$ is zero on Γ_1, the equation becomes

$$\iint_R w(x, y)\left(\nabla^2\, u(x, y) + f(x, y)\right)dA + \int_{\Gamma_2} \left(-\frac{\partial}{\partial n}u(x, y) + h(s) - \alpha(s)\,u(x, y)\right) w(x, y)\,ds = 0$$

Now perform an integration by parts for the term $w(x,y)\nabla^2\, u(x,y)$.

$$w(x, y)\,\nabla^2\, u(x, y) = \nabla\bullet\, (w(x, y)\,\nabla\, u(x, y)) - \nabla\, w(x, y) \bullet \nabla\, u(x, y)$$

Substitute the result into the integral over the surface R. Then, use the Gauss divergence theorem to turn the first term of the integration by parts into an integral over the boundary Γ. Again note that only Γ_2 contributes because w is 0 on Γ_1 by definition. $\boldsymbol{n}$ is the outward pointing normal from the boundary. For the sake of brevity the (x,y) has been left out of the function symbols.

$$\int_{\Gamma_2} ((w\nabla\, u) \bullet \boldsymbol{n})\,ds - \iint_R (\nabla\, w \bullet \nabla\, u - wf)\,dA + \int_{\Gamma_2} \left(-\frac{\partial u}{\partial n} + h(s) - \alpha(s)\,u\right) w(x, y)\,ds = 0$$

Notice that the first integral cancels the first term in the third integral. The gradient of u dotted into the outward pointing normal is in fact the normal derivative. So after making that cancellation, the equation becomes

$$\iint_R (\nabla w \bullet \nabla u - wf)\, dA - \int_{\Gamma_2} (h(s) - \alpha(s)\, u)\, w ds = 0$$

The above equation is referred to as the weak formulation. Note that the natural or suppressible boundary conditions are part of the equation. They do not have to be explicitly "added in." With no Type 2 boundary conditions, $h(s)=\alpha(s)=0$ and the equation becomes

$$\iint_R (\nabla w \bullet \nabla u - wf)\, dA = 0$$

The Galerkin Method

In the Galerkin approach, the solution is a linear combination of a set of M linearly independent functions plus another function that satisfies all the essential boundary conditions, $\Omega(x,y)$. The index M ranges over all the finite element nodes that are not constrained. The set of linearly independent functions is $\{\phi_1(x,y),...,\phi_M(x,y)\}$ and is abbreviated as $\{\phi\}$. These linearly independent functions vanish on Γ_1.The Galerkin approximation to the solution as follows:

$$u(x, y) = \Omega(x, y) + \sum_{j=1}^{M} u_j \phi_j(x, y)$$

The $\{\phi\}$are the *nodal basis functions*. These are not the elemental basis functions discussed earlier; but they are assembled from the elemental basis functions, as shown in Figure A.3. A basis function peaks at the node and drops to 0 at the neighboring nodes. The figure shows why these functions are sometimes called hat functions.

Note that there is one basis function for every finite element node but that a finite element node actually belongs to several finite elements. The nodal basis function for a particular finite element node is the sum of all the elemental basis functions associated with that finite element node with the additional requirement that the value at the finite element node remain 1.

Note that at a non-boundary node (for example, node i in Figure A.3), the coordinates are (x_i,y_i) and $u(x,y)$ is $u(x_i,y_i)$ or u_i. This means that the coefficients in the linear expansion are the values of the function at the finite element nodes. At a node satisfying an essential boundary condition, the ϕ_j's are zero and $u(x,y)$ is equal to $\Omega(x,y)$.

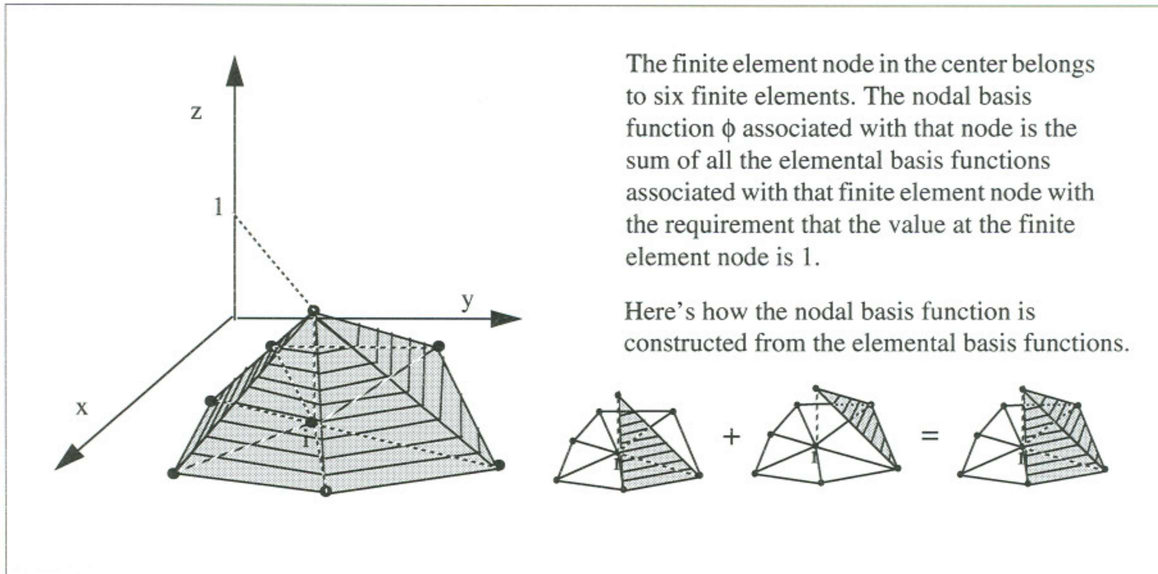


Figure A.3. The Nodal Basis Function for a Node

Now substitute the Galerkin approximation in for $u(x,y)$ in the weak formulation. Take terms involving products of terms containing $w(x,y)$ to the left-hand side and those involving $\Omega(x,y)$ or products that contain $w(x,y)$ in only one term to the right-hand side. The result is as follows:

$$\sum_{j=1}^{M}\left(\iint_R \nabla w \bullet \nabla \phi_j dA + \int_{\Gamma_2} \alpha w \phi_j ds\right)u_i = \iint_R (-\nabla w \bullet \nabla \Omega + fw)\, dA + \int_{\Gamma_2} (-\alpha\Omega w + hw)\, ds$$

The Galerkin method requires that the arbitrary function w be chosen from the set of ϕ_j's. There are then M versions of the above equation, one for each possible choice of w. Let the index i identify the equation with $w = \phi_i$. Then, the ith equation is

$$\sum_{j=1}^{N}\left(\iint_R \nabla \phi_i \bullet \nabla \phi_j dA + \int_{\Gamma_2} \alpha \phi_i \phi_j ds\right)u_i = \iint_R (-\nabla \phi_i \bullet \nabla \Omega + f\phi_i)\, dA + \int_{\Gamma_2} (-\alpha\Omega\phi_i + h\phi_i)\, ds$$

This equation can be rewritten as a matrix equation as follows:

$$\sum_{j=1}^{M} K_{ij} u_j = d_i \qquad \text{or} \qquad \mathbf{Ku} = \mathbf{d}$$

where

$$K_{ij} = \iint_R (\nabla \phi_i) \bullet (\nabla \phi_j)\, dA + \int_{\Gamma_2} \alpha \phi_i \phi_j ds$$

$$d_i = \iint_R (f\phi_i - (\nabla \phi_i) \bullet \nabla \Omega)\, dA + \int_{\Gamma_2} (h\phi_i - \alpha\Omega\phi_i)\, ds$$

The finite element choices for Ω, $\phi_1, ..., \phi_N$ are piecewise polynomial functions that are zero outside some small region of R. This ensures that $\boldsymbol{K}$ is sparse because $K_{ij} = 0$ unless the regions where ϕ_i and ϕ_j are nonzero overlap.

The above equations calculate an assembled matrix element. That's because ϕ functions in the equation for K_{ij} are the nodal basis functions, not the elemental basis functions. This assembled matrix element is a sum of elemental matrix elements, with a contribution from each finite element. To write down integrals for the elemental matrix elements, one must use the elemental basis functions and then sum these elemental matrix elements to get the assembled matrix element.

Assume that the surface is partitioned into n_{el} finite elements. Then, the assembled matrix element can be written as the sum of the elemental matrix elements over all the finite elements. Note that the second sum is over the portions of the Γ_2 line segments in the finite elements that have Γ_2 line segments.

$$K_{ij} = \sum_{el=1}^{n_{el}} k_{ij}^{(el)} = \sum_{el=1}^{n_{el}} \iint_{el} \nabla N_i^{(el)} \bullet \nabla N_j^{(el)} dA + \sum_{el=1}^{n_{el}} \int_{\Gamma_2^e} \alpha N_i^{(el)} N_j^{(el)} ds$$

An individual finite element contributes as many elemental basis functions as it has finite element nodes. In the above equation, these N's are indexed globally. That is, the subscript on each N is the global index of the finite element nodes in the finite element mesh. At each of the nodes in the element, one of the finite element's N functions has value 1 and the others are zero. Specifically, on the ith finite element node in the el^{th} finite element, the function $N_i^{(el)}$ is 1. The hat function ϕ_i is the sum of all the $N_i^{(el)}$, one from each finite element that contains the finite element node i.

The load vector (the inhomogeneous term), like the matrix element, can also be written as a sum. Again, the first sum is over the finite elements. The second sum is over the portions of the Γ_2 line segments in the finite elements that have such.

$$d_i = \sum_{el=1}^{n_{el}} d_i^{(el)} = \sum_{el=1}^{n_{el}} \iint_{el} (fN_i^{(el)} - \nabla \Omega \bullet \nabla N_i^{(el)}) dA + \sum_{el=1}^{n_{el}} \int_{\Gamma_2^{el}} (hN_i^{(el)} - \alpha \Omega N_i^{(el)}) ds$$

The requirement on $\Omega(x,y)$ is that it evaluate to a prescribed constant on Γ_1. Choose for $\Omega(x,y)$ a hat function built up from elemental basis functions that have their maxima on the prescribed finite element node. Figure A.4 shows such a function, called $\Omega_j(x,y)$whose maximum is at finite element node j. The total $\Omega(x,y)$ is the sum

$$\Omega(x, y) = \sum_i \Omega_i(x, y)$$

where i goes over the finite element nodes on Γ_1.

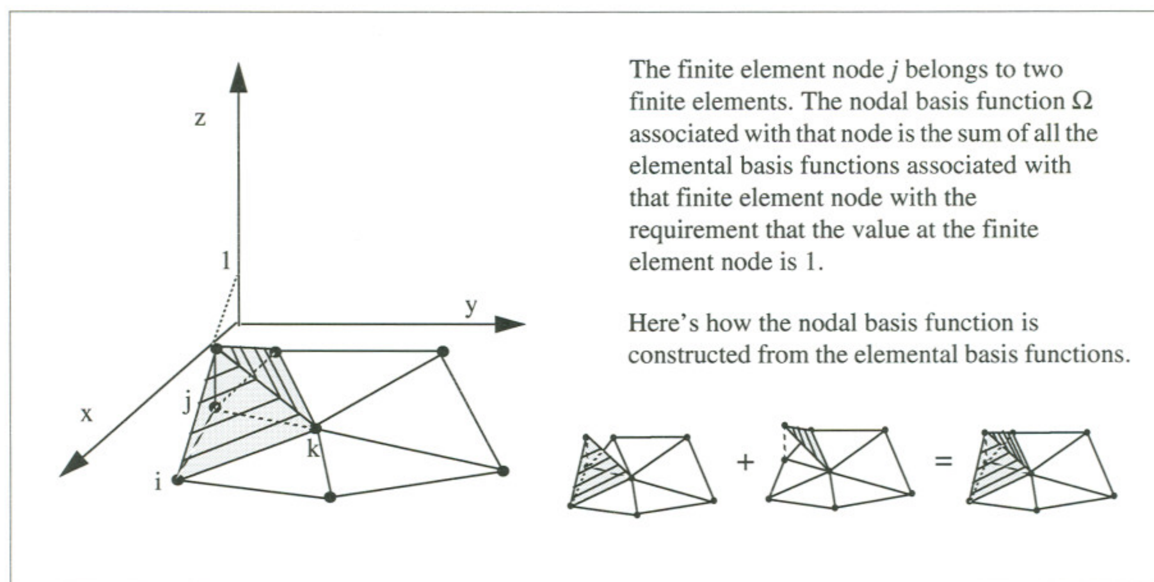


Figure A.4. The Nodal Basis Function for a Node

An Example of Solving Poisson's Equation

This section evaluates the integrals that make up the definitions of the matrix elements for a specific example. The example is Poisson's equation $\nabla^2 u(x, y) = f(x, y)$. The section shows how the assembled matrix element is constructed from the elemental matrix elements.

Assume that there are no Type 2 boundary conditions. This means α and h in the equations discussed in the "The Weak Formulation" on page 118 are both 0. Also, assume two dimensions and expand out the Cartesian form of the gradient. Now develop an expression for the assembled matrix elements, K_{ij}. Start with the following equation:

$$K_{ij} = \sum_{el=1}^{n_{el}} \iint_{el} \left(\frac{\partial N_i^{(el)}}{\partial x} \frac{\partial N_j^{(el)}}{\partial x} + \frac{\partial N_i^{(el)}}{\partial y} \frac{\partial N_j^{(el)}}{\partial y} \right) dA = \sum_{el=1}^{n_{el}} \iint_{el} (\nabla N_i^{(el)} \cdot \nabla N_j^{(el)})\, dA$$

Using the expressions for the derivatives of N developed earlier, this becomes

$$K_{ij} = \sum_{el=1}^{n_{el}} \iint_{el} \frac{(b_i^{(el)} b_j^{(el)} + c_i^{(el)} c_j^{(el)})}{4A^{(el)2}} dA$$

Because the integrand is constant, the equation becomes

$$K_{ij} = \sum_{el=1}^{n_{el}} \frac{(b_i^{(el)} b_j^{(el)} + c_i^{(el)} c_j^{(el)})}{4A^{(el)}}$$

Note above that the vectors $b^{(el)}$ and $c^{(el)}$ have as many components as nodes in the element. With triangular elements, these vectors have three components. The indices of these components, however, do not go from 1 to 3, but rather are global over the entire finite element mesh.

Elemental Matrix

A finite element contributes an elemental matrix to the assembled matrix. A triangular finite element would contribute a 3x3 elemental matrix

For example, Figure A.5 shows six finite element nodes. Here each finite element contributes a 3x3 elemental matrix. The assembled matrix elements for finite element nodes that exist in more than one finite element get local contributions from the elemental matrices they exist in. For example, the diagonal assembled matrix element K_{77} gets contributions from each of the six finite elements that contain finite element node 7 as one of its vertices.

As a more detailed example, consider the elemental matrices associated with finite elements (2)and (3). Superscripts indicate the finite element from which the matrix element comes. The elemental matrix associated with finite element (2) (with indices 1, 2, and 7) is

$$\begin{bmatrix} k_{11}^{(2)} & k_{12}^{(2)} & k_{17}^{(2)} \\ k_{21}^{(2)} & k_{22}^{(2)} & k_{27}^{(2)} \\ k_{71}^{(2)} & k_{72}^{(2)} & k_{77}^{(2)} \end{bmatrix}$$

The elemental matrix associated with element III (with indices 2,3, and 7) is

$$\begin{bmatrix} k_{22}^{(3)} & k_{23}^{(3)} & k_{27}^{(3)} \\ k_{32}^{(3)} & k_{33}^{(3)} & k_{37}^{(3)} \\ k_{72}^{(3)} & k_{73}^{(3)} & k_{77}^{(3)} \end{bmatrix}$$

The assembled matrix is the sum of the contributions from each elemental matrix. For example, the assembled matrix element K_{27} is $k_{27}^{(2)} + k_{27}^{(3)}$.

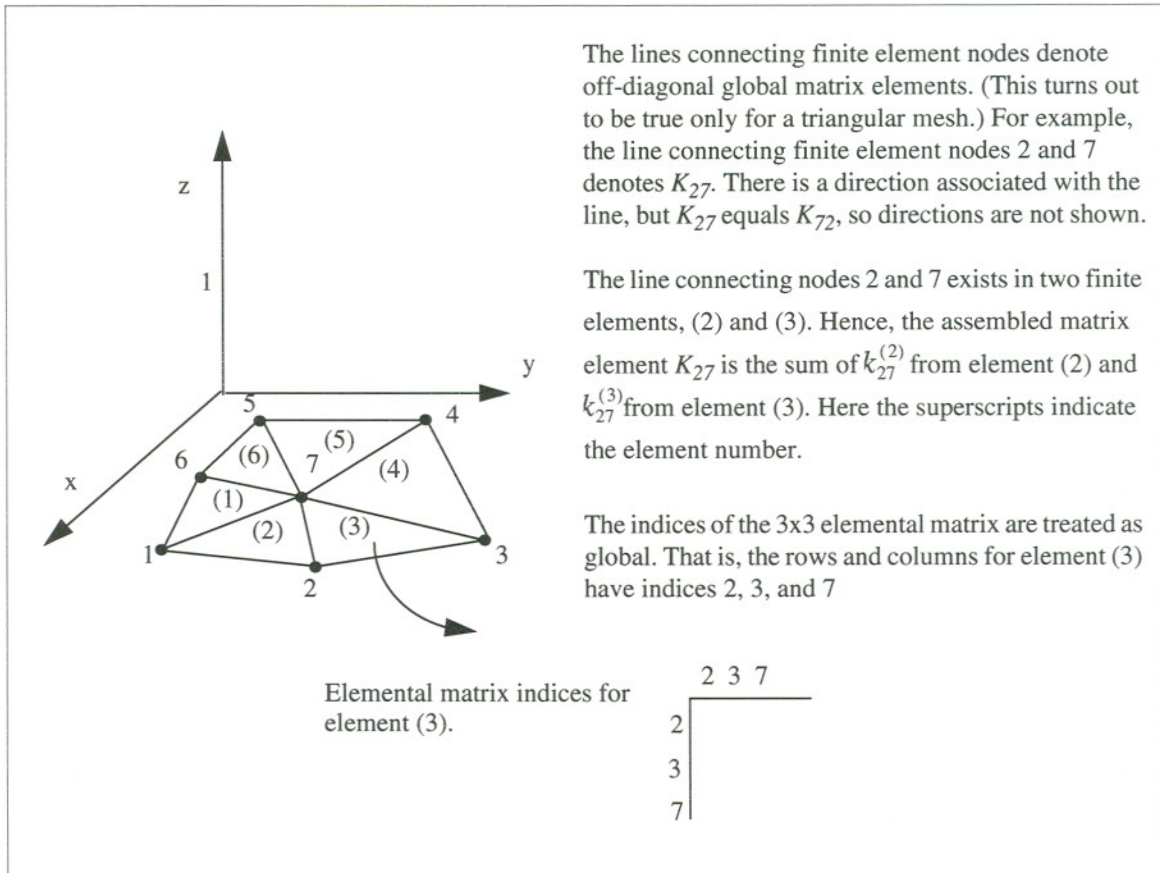


Figure A.5. Finite Element Nodes 1 through 7 with Elements I through VI

This method of constructing the assembled matrix can be written as follows:

$$K = \sum_{el=1}^{n_{el}} k^{(el)} = \sum_{el=1}^{n_{el}} \frac{(b^{(el)} b^{(el)T} + c^{(el)} c^{(el)T})}{4A^{(el)}}$$

Note that the right hand side is a sum of 3x3 matrices, but the answer is a matrix of larger order because the indices on the 3x3 matrices are global. Also note that the vector products in the above equation are outer products.

The above equation is the form used in the computations. The elemental matrices are formed and then fed into the assembled matrix.

The Right-Hand Side

Now consider the right-hand side. Again, because there are no Type 2 boundary conditions $\alpha = h = 0$. Then,

$$d_i = \sum_{el=1}^{n_{el}} \iint_{el} (fN_i^{(el)} - \nabla\ \Omega \bullet \nabla\ N_i^{(el)})\, dA$$

For ease of computation, rewrite this equation as a vector equation. The vector $\boldsymbol{N}^{(el)}$ consists of the three elemental basis functions associated with the triangular finite element *el*. If the finite element has indices *i*, *j*, and *k*,

$$\boldsymbol{N}^{(el)T} = (N_i^{(el)}, N_j^{(el)}, N_k^{(el)})$$

Now consider the first term in the integral (the one without the Ω).

$$\sum_{el=1}^{n_{el}} \iint_{el} \boldsymbol{N}^{(el)} f dA$$

This is a difficult integral to perform because *f* is a function of *x* and *y*. To make the integral easier, approximate *f* in a finite element as

$$f = N_i^{(el)} f_i + N_j^{(el)} f_j + N_k^{(el)} f_k = \boldsymbol{N}^{(el)T} \boldsymbol{f}^{(el)}$$

This approximation is called the *linear interpolant* of *f*. The vector $\boldsymbol{f}^{(el)}$ has three components, the value of *f* at each vertex of the triangular finite element. This vector is a constant and can be taken out of the integral.

$$\sum_{el=1}^{n_{el}} \iint_{el} \boldsymbol{N}^{(el)} \boldsymbol{N}^{(el)T} \boldsymbol{f} dA = \sum_{el=1}^{n_{el}} \left(\iint_{el} \boldsymbol{N}^{(el)} \boldsymbol{N}^{(el)T} dA \right) \boldsymbol{f}^{(el)}$$

The integral in parentheses is a constant. The result of evaluating the integral (which is not easy) [5] is

$$\sum_{el} \left(\iint_{el} \boldsymbol{N}^{(el)} \boldsymbol{N}^{(el)T} dA \right) \boldsymbol{f}^{(el)} = \sum_{el} \frac{A^{(el)}}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \boldsymbol{f}^{(el)}$$

The actual indices of $\boldsymbol{f}^{(el)}$, however, are the global indices. So when the sum over the finite elements is performed, one sums the components of $\boldsymbol{f}^{(el)}$ that have the same index.

Now return to the expression for d_i and examine the second term in the integral.

$$-\sum_{el=1}^{n_{el}} \iint_{el} \nabla \Omega \bullet \nabla N_i^{(el)} dA$$

Here the sum is over only those finite elements that have a Γ_1 boundary. The i must be one of the indices of the finite element that has the boundary finite element node as one of its indices. For each finite element, the Ω is the sum of N_j where j runs over the indices of the finite element nodes on the Γ_1 boundary that are in that finite element.

An Example of Constructing Matrix Elements

This section shows how the equations developed so far are used to calculate some specific matrix elements for a specific grid. The grid is shown in Figure A.6. This section calculates the matrix elements that link finite element node 4 with the other finite element nodes.

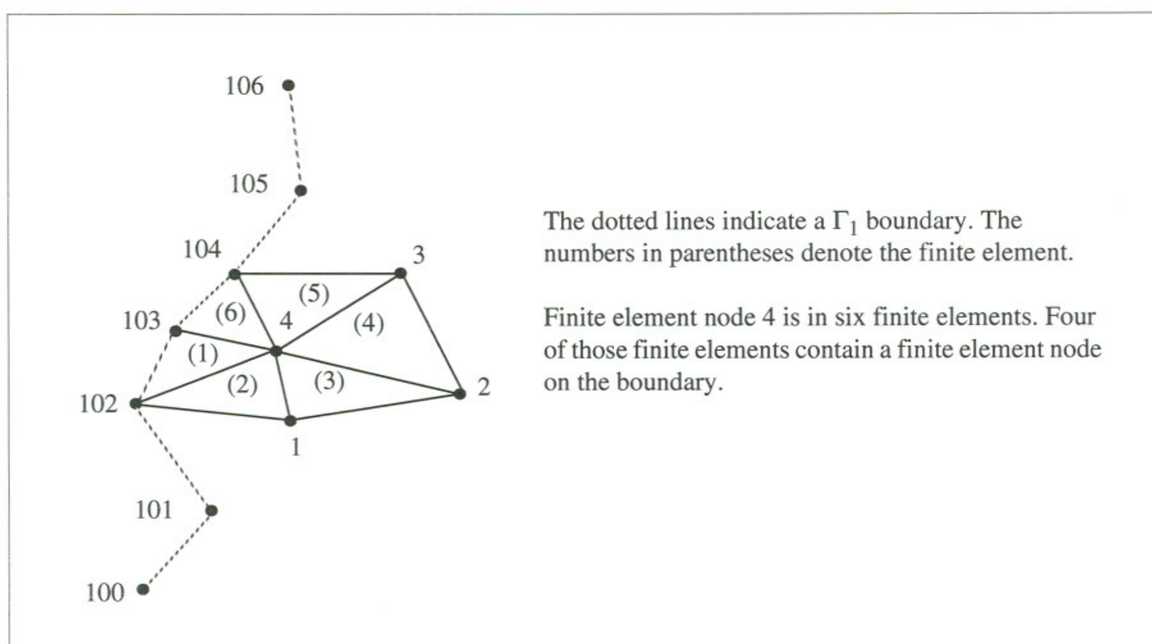


Figure A.6. Grid Used to Calculate Example Matrix Elements

Finite element node 4 has the value u_4 and is connected to six finite element nodes. The finite element nodes 1, 2,and 3 have unknown values; they are u_1, u_2, and u_3. The finite element nodes 102, 103, and 104 have constrained values; they are c_{102}, c_{103}, c_{104}. The fourth row of the matrix-vector multiplication $\boldsymbol{Ku}$ is

$$K_{41}u_1 + K_{42}u_2 + K_{43}u_3 + K_{44}u_4$$

K_{41} is an assembled matrix element that comes from the elemental matrix element $k_{14}^{(2)}$ and $k_{14}^{(3)}$ from finite elements (2) and (3); K_{42} is assembled from $k_{42}^{(3)}$ and $k_{42}^{(4)}$, etc. For example,

$$K_{41} = \int\int_{(2)} \left(\frac{\partial N_4^{(2)}}{\partial x}\frac{\partial N_1^{(2)}}{\partial x} + \frac{\partial N_4^{(2)}}{\partial y}\frac{\partial N_1^{(2)}}{\partial y}\right)dA + \int\int_{(3)} \left(\frac{\partial N_4^{(3)}}{\partial x}\frac{\partial N_1^{(3)}}{\partial x} + \frac{\partial N_4^{(3)}}{\partial y}\frac{\partial N_1^{(3)}}{\partial y}\right)dA$$

$$K_{42} = \int\int_{(3)} \left(\frac{\partial N_4^{(3)}}{\partial x}\frac{\partial N_2^{(3)}}{\partial x} + \frac{\partial N_4^{(3)}}{\partial y}\frac{\partial N_2^{(3)}}{\partial y}\right)dA + \int\int_{(4)} \left(\frac{\partial N_4^{(4)}}{\partial x}\frac{\partial N_2^{(4)}}{\partial x} + \frac{\partial N_4^{(4)}}{\partial y}\frac{\partial N_2^{(4)}}{\partial y}\right)dA$$

$$K_{43} = \int\int_{(4)} \left(\frac{\partial N_4^{(4)}}{\partial x}\frac{\partial N_3^{(4)}}{\partial x} + \frac{\partial N_4^{(4)}}{\partial y}\frac{\partial N_3^{(4)}}{\partial y}\right)dA + \int\int_{(5)} \left(\frac{\partial N_4^{(5)}}{\partial x}\frac{\partial N_3^{(5)}}{\partial x} + \frac{\partial N_4^{(5)}}{\partial y}\frac{\partial N_3^{(5)}}{\partial y}\right)dA$$

$$K_{44} = \int\int_{(1)} \left(\frac{\partial N_4^{(1)}}{\partial x}\frac{\partial N_4^{(1)}}{\partial x} + \frac{\partial N_4^{(1)}}{\partial y}\frac{\partial N_4^{(1)}}{\partial y}\right)dA + \int\int_{(2)} \left(\frac{\partial N_4^{(2)}}{\partial x}\frac{\partial N_4^{(2)}}{\partial x} + \frac{\partial N_4^{(2)}}{\partial y}\frac{\partial N_4^{(2)}}{\partial y}\right)dA$$

$$+\int\int_{(3)} \left(\frac{\partial N_4^{(3)}}{\partial x}\frac{\partial N_4^{(3)}}{\partial x} + \frac{\partial N_4^{(3)}}{\partial y}\frac{\partial N_4^{(3)}}{\partial y}\right)dA + \int\int_{(4)} \left(\frac{\partial N_4^{(4)}}{\partial x}\frac{\partial N_4^{(4)}}{\partial x} + \frac{\partial N_4^{(4)}}{\partial y}\frac{\partial N_4^{(4)}}{\partial y}\right)dA$$

$$+\int\int_{(5)} \left(\frac{\partial N_4^{(5)}}{\partial x}\frac{\partial N_4^{(5)}}{\partial x} + \frac{\partial N_4^{(5)}}{\partial y}\frac{\partial N_4^{(5)}}{\partial y}\right)dA + \int\int_{(6)} \left(\frac{\partial N_4^{(6)}}{\partial x}\frac{\partial N_4^{(6)}}{\partial x} + \frac{\partial N_4^{(6)}}{\partial y}\frac{\partial N_4^{(6)}}{\partial y}\right)dA$$

Now construct the right-hand side. d_4 has again contributions from six finite elements. This means that in the first of the two terms making up the right-hand side, there are six vectors $f^{(el)}$. Each of these vectors has three components, one for each of the finite element nodes in each of the six finite elements. For example,

$$\sum_{el} \frac{A^{(el)}}{12}\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix} f^{(el)} = \frac{A^{(1)}}{12}\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix}\begin{bmatrix}f_4\\f_{102}\\f_{103}\end{bmatrix} + \frac{A^{(2)}}{12}\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix}\begin{bmatrix}f_4\\f_1\\f_{102}\end{bmatrix} + \frac{A^{(3)}}{12}\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix}\begin{bmatrix}f_4\\f_2\\f_1\end{bmatrix} + \frac{A^{(4)}}{12}\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix}\begin{bmatrix}f_4\\f_3\\f_2\end{bmatrix}$$

$$+ \frac{A^{(5)}}{12}\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix}\begin{bmatrix}f_4\\f_{104}\\f_3\end{bmatrix} + \frac{A^{(6)}}{12}\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix}\begin{bmatrix}f_4\\f_{103}\\f_{104}\end{bmatrix}$$

To pick out the contributions to d_4, sum the appropriate components after the matrix-vector multiplication; each finite element contributes its first component from the previous sum to give

$$\frac{A^{(1)}}{12}(2f_4 + f_{102} + f_{103}) + \frac{A^{(2)}}{12}(2f_4 + f_1 + f_{102}) + \frac{A^{(3)}}{12}(2f_4 + f_2 + f_1) + \frac{A^{(4)}}{12}(2f_4 + f_3 + f_2)$$

$$+\frac{A^{(5)}}{12}(2f_4+f_{104}+f_3)+\frac{A^{(6)}}{12}(2f_4+f_{103}+f_{104})$$

To carry the example a little further, assume that *f* is defined as *F* at all finite element nodes and that the areas of all finite elements are equal. Then,

$$\frac{A}{12}(4+4+4+4+4+4)F = 2AF$$

Now consider the second half of the right-hand side. Finite element node 4 is connected to four finite elements that contain a finite element node on the boundary. These are finite elements (2), (1), (6), and (5). For Ω, take the sum of the elemental basis functions at each of the constrained finite element nodes multiplied by the value at that constrained finite element node.

$$-\sum_{el=1}^{n_{el}}\iint_{el}\nabla\,\Omega\bullet\nabla\,N_i^{(el)}dA = -\left(c_{102}\int\int_{(1)}\nabla\,N_{102}^{(1)}\bullet\nabla\,N_4^{(1)}dA + c_{102}\int\int_{(2)}\nabla\,N_{102}^{(2)}\bullet\nabla\,N_4^{(2)}dA\right)$$

$$-\left(c_{103}\int\int_{(1)}\nabla\,N_{103}^{(1)}\bullet\nabla\,N_4^{(1)}dA + c_{103}\int\int_{(6)}\nabla\,N_{103}^{(6)}\bullet\nabla\,N_4^{(6)}dA\right)$$

$$-\left(c_{104}\int\int_{(6)}\nabla\,N_{104}^{(6)}\bullet\nabla\,N_4^{(6)}dA + c_{104}\int\int_{(5)}\nabla\,N_{104}^{(5)}\bullet\nabla\,N_4^{(5)}dA\right)$$

This equation can be rewritten in terms of matrix elements as follows:

$$-\sum_{el=1}^{n_{el}}\iint_{el}\nabla\,\Omega\bullet\nabla\,N_i^{(el)}dA = -(K_{4,\,102}c_{102}+K_{4,\,103}c_{103}+K_{4,\,104}c_{104})$$

pfThe complete equation is then

$$K_{41}u_1+K_{42}u_2+K_{43}u_3+K_{44}u_4 = 2AF-(K_{4,\,102}c_{102}+K_{4,\,103}c_{103}+K_{4,\,104}c_{104}) = d_4$$

This is one of the many equations defining the solution of Poisson's equation on a finite element mesh. The complete set of equations is denoted by the matrix equation ***Ku=d***.

Another Way of Accounting for Type 1 Boundary Conditions

It is interesting and informative to point out that there is another way of arriving at the same set of equations. This is the method used in [5]. Assume that instead of

$$u(x, y) = \Omega(x, y) + \sum_{j=1}^{M} u_j \phi_j(x, y)$$

one substitutes

$$u(x, y) = \sum_{j=1}^{N} u_j \phi_j(x, y)$$

into the partial differential equation. But now the sum is over the entire set N of finite element nodes, even the constrained ones.

Once again, consider the matrix equation $\boldsymbol{Ku} = \boldsymbol{d}$. But now for $\boldsymbol{d}$, only include the first term on the right-hand side, not the term containing Ω. In the previous section, this term was $2AF$. The matrix equation now reads

$$\sum_{j=0}^{N} K_{ij} u_j = 2AF$$

Then, to incorporate a Type 1 boundary condition at finite element node i, set u_i to c_i, the constrained value. Set d_i to this constrained value. Then, set the matrix elements in row i to zero except for K_{ii} which is set to 1. The ith equation becomes

$$k_{ii} u_i = u_i = d_i = c_i$$

However, notice that now the symmetry of the matrix is destroyed. The symmetry can be regained by ensuring that the off-diagonal elements in the column whose diagonal element was set to 1 are also zero. Do that with row operations.

To set a particular column element to zero, multiply the row whose diagonal was set to 1 by the nonzero column element targeted to be zero and then subtract the result from the row containing that column element. Do this for all column elements in the column whose diagonal was set to 1.

One must, of course, be careful to subtract the appropriate terms from the load vector, the right-hand side of the equation. Consider the following example, The equation is a 4x4 matrix vector equation with u_2 constrained to the value c_2. Hence, K_{22} is set to 1 while the other elements in the row are set to 0.

$$\begin{bmatrix} K_{11} & 0 & K_{13} & K_{14} \\ 0 & 1 & 0 & 0 \\ K_{31} & 0 & K_{33} & K_{34} \\ K_{41} & 0 & K_{43} & K_{44} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} F_1 - (K_{12} \cdot c_2) \\ c_2 \\ F_3 - (K_{32} \cdot c_2) \\ F_4 - (K_{42} \cdot c_2) \end{bmatrix}$$

To set K_{12} in the matrix to zero, multiply row 2 by K_{12} and then subtract that row from row 1. The result is that K_{12} becomes zero and the other elements in row 1 are unchanged. We have to be careful, however, to subtract $K_{12}c_2$ from the first element of the right-hand side vector to retain the equality. The matrix vector equation below also shows the terms that must also be subtracted from the third and fourth terms of the load vector.

$$\begin{bmatrix} K_{11} & 0 & K_{13} & K_{14} \\ 0 & 1 & 0 & 0 \\ K_{31} & 0 & K_{33} & K_{34} \\ K_{41} & 0 & K_{43} & K_{44} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} F_1 - (K_{12} \cdot c_2) \\ c_2 \\ F_3 - (K_{32} \cdot c_2) \\ F_4 - (K_{42} \cdot c_2) \end{bmatrix}$$

To see how this develops for more than one boundary condition, assume now that u_1 is constrained to be c_1. The result is that K_{11} becomes 1 and the other elements in row 1 and in column 1 become zero. Setting the other elements of column 1 to zero required modification of the right-hand side as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & K_{33} & K_{34} \\ 0 & 0 & K_{43} & K_{44} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ F_3 - (K_{32} \cdot c_2) - (K_{31} \cdot c_1) \\ F_4 - (K_{42} \cdot c_2) - (K_{41} \cdot c_1) \end{bmatrix}$$

Notice that the matrix equation now contains on the right-hand side the terms that would have developed if the second right-hand side term (the one containing Ω had been retained). The difference is that the matrix is larger, containing explicit equations that define the Type 1 boundary conditions. In the previous method, the matrix would have been 2x2 instead of 4x4. It would not have contained the rows and columns that just have 1 on the diagonal.

The Solution

Solving the matrix equation developed in the previous sections results in the solution vector, **u**. This vector has as many elements as there are finite element nodes. Each element of the vector is the value of the solution at the finite element node. u_i is the solution at the node i. These values are also the coefficients used in the Galerkin approximation as follows:

$$u(x, y) = \Omega(x, y) + \sum_{j=1}^{M} u_j \phi_j(x, y)$$

Appendix B Exact Solution of Laplace's Equation

Introduction

This appendix presents an exact, analytic solution of Laplace's equation in continuous space. The purpose of this appendix is to compare this exact solution with the answer obtained using finite elements and the conjugate gradient method.

Statement of the Problem

Laplace's equation is a special case of Poisson's equation, which is as follows:

$$\nabla^2 u(x, y) + f(x, y) = 0$$

When $f(x,y)$ is zero, it is Laplace's equation. Assume that it is to be solved over a two-dimensional square whose side is 1. Also no type 2 boundary conditions and assume the following type 1 (Dirichlet) boundary conditions:

$$u(x,0) = u(x,1) = 0 \qquad 0 < x < 1$$
$$u(0,y) = u(1,y) = 100 \qquad 0 \le y \le 1$$

Figure B.3 shows the domain over which the equation is to be solved along with the boundary conditions. This domain is divided into 200 finite elements.

Solving the Equation

Notice that Laplace's equation is an elliptic partial differential equation. Use the separation of variables technique. Assume that the solution $u(x,y)$ can be written as

$$u(x, y) = u_x u_y$$

where u_x depends only on x and u_y depends only on y. Substitute this value for u in Laplace's equation, then divide by $u_x u_y$. The result is

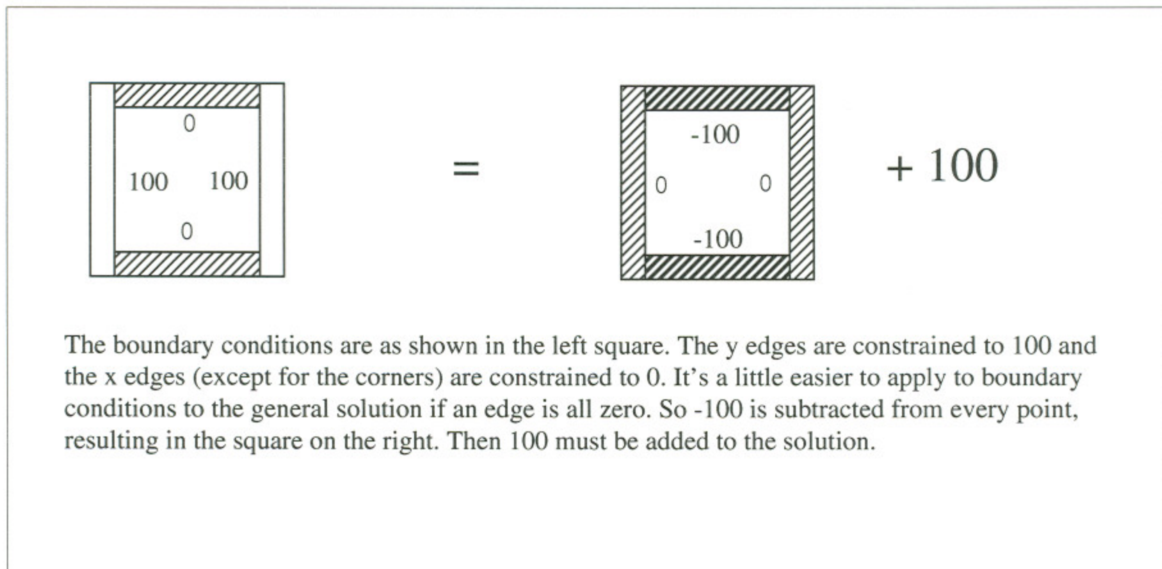
$$\frac{\frac{\partial^2 u_x}{\partial x^2}}{u_x} = -\frac{\frac{\partial^2 u_y}{\partial y^2}}{u_y}$$

Because one side of the equation depends only on *x* and the other only on *y*, they can only be equal if they are both equal to the same constant, which is called the separation constant. The result is that the two-dimensional differential equation can be written as two one-dimensional equations.

$$\frac{\partial^2 u_x}{\partial x^2} + R^2 u_x = 0 \qquad \frac{\partial^2 u_y}{\partial y^2} - R^2 u_y = 0$$

Now, for each equation, assume an exponential solution and solve the characteristic equation. The result is that a solution of u_x is a linear combination of exponential functions and a solution for u_y is a linear combination of sinusoidal functions.

First modify the boundary conditions as shown in Figure B.1. That is, arrange that one set of edges is constrained to be zero [17]. These are the y edges. The problem is unchanged, but you must remember to add 100 to the solution.



The boundary conditions are as shown in the left square. The y edges are constrained to 100 and the x edges (except for the corners) are constrained to 0. It's a little easier to apply to boundary conditions to the general solution if an edge is all zero. So -100 is subtracted from every point, resulting in the square on the right. Then 100 must be added to the solution.

Figure B.1. Offsetting the Boundary Conditions for Easier Application

Now consider u_x and write it as a linear combination of a sine and a cosine. The modified boundary conditions state that u must be zero at x=0 and x=1. This means that we can drop the cosine term from the solution and

$$u_x = A_2 \sin Rx = A_2 \sin n\pi x$$

Here n is any integer. The solution for u_y can be written as follows:

$$u_y = A_3 e^{n\pi y} + A_4 e^{-n\pi y}$$

To satisfy the remaining boundary conditions at y=0 and y=1, first write the solution as the product $u_x u_y$. Because the boundary condition is a constant at constant y and any x, show the solution as an infinite series. Absorb the constant A_2 from u_x into c_n.

$$u(x, y) = \sum_{n=1}^{\infty} c_n (A_3 e^{n\pi y} + A_4 e^{-n\pi y}) \sin n\pi x$$

Because of the boundary conditions, the term in parentheses must be the same for y=0 and y=1; these are the x edges, and both are -100. In the following equation, the right-hand side is the value of the term at y=0 and the left-hand side is the value of the term at y=1.

$$A_3 + A_4 = A_3 e^{n\pi} + A_4 e^{-n\pi}$$

Solving for A_4, you get

$$A_4 = \frac{-A_3 (1 - e^{n\pi})}{1 - e^{-n\pi}}$$

After substituting this value for A_4 in the expression for $u(x,y)$ and then absorbing A_4 into c_n, $u(x,y)$ becomes

$$u(x, y) = \sum_{n=1}^{\infty} c_n \left(e^{n\pi y} + \frac{(1 - e^{n\pi})}{e^{-n\pi} - 1} e^{-n\pi y} \right) \sin n\pi x$$

At y=0 or y=1, the equation must equal -100.

$$u(x, 0) = u(x, 1) = \sum_{n=1}^{\infty} c_n \left(e^{n\pi y} + \frac{(1 - e^{n\pi})}{e^{-n\pi} - 1} e^{-n\pi y} \right) \sin n\pi x = -100$$

Now need to get an expression for c_n, the one remaining constant. To do that, make use of the orthogonality of the sine functions. Multiply both sides by $\sin m\pi x$. Integrate over the period of the sine in x, which is 2. When the function is undefined for x > 1, the only nonzero part of the integral is from 0 to 1.

$$\int_0^1 \sin n\pi x \sin m\pi x dx = \frac{1}{2}\delta_{nm}$$

where δ is the Kronecker delta. This property brings out c_m. Then relabel m to n.

$$c_n\left(\frac{(e^{-n\pi} - e^{n\pi})}{e^{-n\pi} - 1}\right) = -2\int_0^1 100 \sin n\pi x dx$$

After evaluating the integral, solving for c_n, and then substituting that value into the infinite series form of *u(x,y)* (now remembering to add 100), *u(x,y)* can be written as

$$u(x, y) = \sum_{n=1}^{\infty} \frac{-200}{n\pi}(1-(-1)^n)\frac{e^{n\pi y}e^{-n\pi} - e^{n\pi x} + e^{-n\pi y} - e^{-n\pi y}e^{n\pi}}{e^{-n\pi} - e^{n\pi}}\sin n\pi x + 100$$

Graphing the Solution

To graph the solution, first set up a two-dimensional grid with 11 nodes in each dimension. The *x* coordinates range from 0.0 to 1.0 in increments of 0.1. The *y* coordinates have the same range. The total number of points is then 121.

For each of the 121 points, put the coordinate values in the expression for *u(x,y)* and sum 225 terms. Figure B.2 is a graph of *u(x,y)* calculated this way. The value of *u(x,y)* is shown as a surface over the *xy* plane.

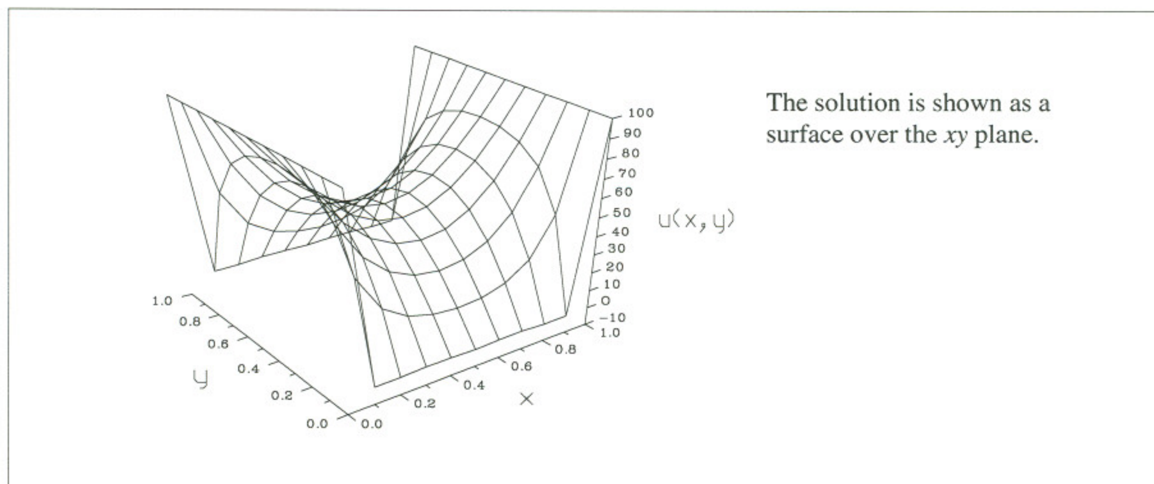


Figure B.2. Solution of Laplace's Equation (Infinite Series)

Finite Element Method

The finite element method is used to solve the same problem solved analytically in "Solving the Equation" on page 133. Consider the same two-dimensional square and divide it into 200 triangles as shown in Figure B.3. The boundary conditions are also the same as shown previously.

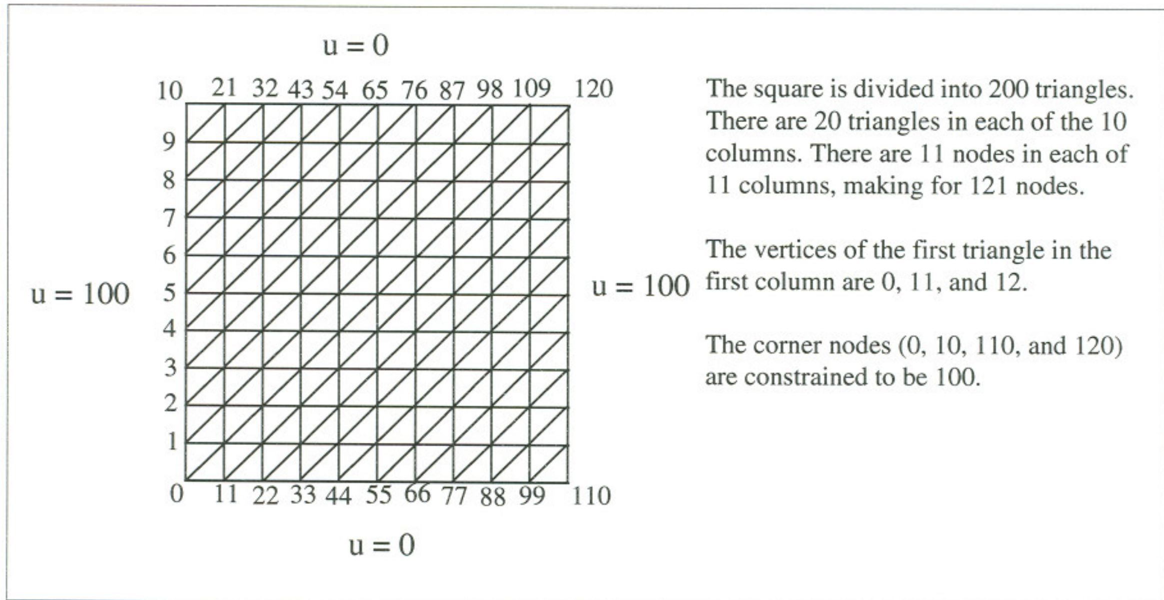


Figure B.3. The Surface Divided into Finite Elements

This problem was divided over four compute nodes as shown in Figure B.4. The figure shows two of the resulting alias groups. The solution is shown in Figure B.5.

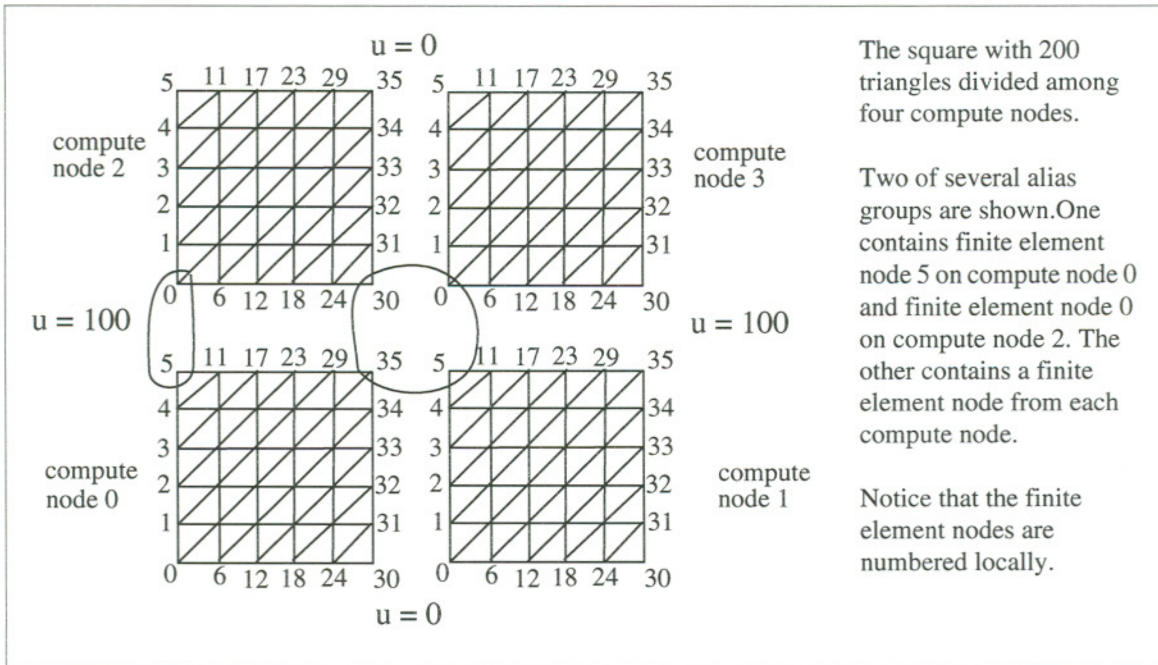


Figure B.4. The Surface Divided Among Four Compute Nodes.

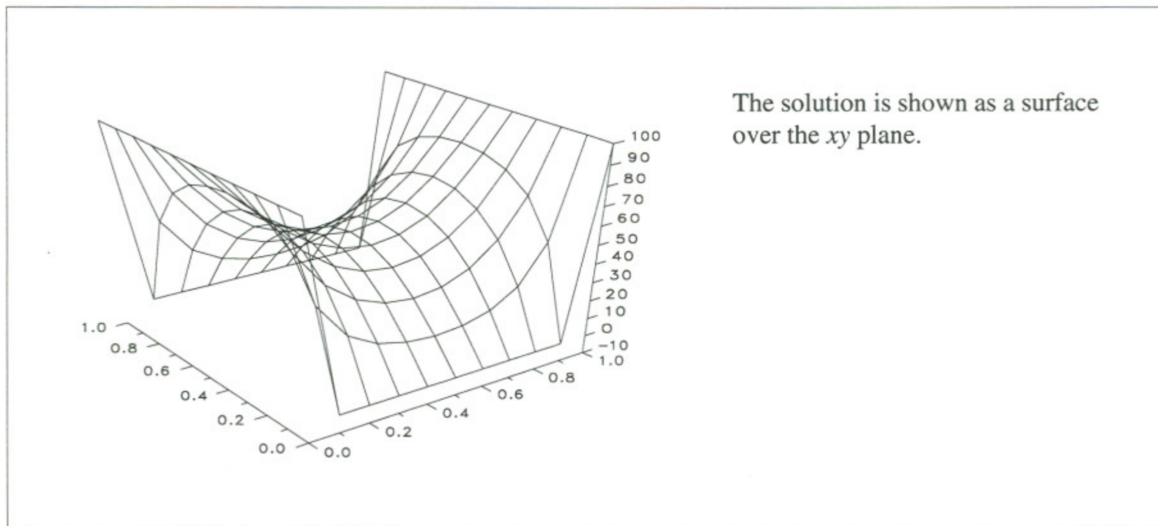


Figure B.5. Solution of Laplace's Equation (Finite Element)

Appendix C The Conjugate Gradient Method

This appendix lists the steps of the conjugate gradient method. The discussion follows the organization in the text by Gene Golub and James Ortega [15], but with a slightly different notation.

The conjugate gradient method is an iterative method for solving the matrix equation $\boldsymbol{Ax} = \boldsymbol{b}$ where $\boldsymbol{A}$ is the matrix of coefficients, $\boldsymbol{x}$ is the unknown vector, and $\boldsymbol{b}$ is a constant load vector. The iteration is carried out until the residual is less than a prescribed *epsilon*. The residual is defined as

$$residual = b - Ax$$

1. Make an initial guess for $\boldsymbol{x}$ and call it $\boldsymbol{x_0}$. 0 is an acceptable guess; set all elements of $\boldsymbol{x}$ to $\boldsymbol{x_0}$.

2. Set the initial value of the residual to $\boldsymbol{residual}^{(1)} = \boldsymbol{b} - \boldsymbol{Ax_0}$. The superscript in parentheses indicates the iteration number, which is initially 1. The value of $\boldsymbol{residual}^{(1)}$ is $\boldsymbol{b}$ if the initial $\boldsymbol{x_0}$ is 0. Check if the maximum element of $\boldsymbol{residual}^{(1)}$ is less than *epsilon* and if it is, stop.

3. Form $\rho^{(1)}$, which is the square of the residual. $\rho^{(1)} = \boldsymbol{residual}^{(1)} \cdot \boldsymbol{residua}^{(1)}$.

4. Form $\boldsymbol{g}^{(1)}$, the initial search direction. Set $\boldsymbol{g}^{(1)} = \boldsymbol{residual}^{(1)}$.

5. Now start the iteration. In the steps below the superscript *k* indicates the iteration number, which is initially 1.

   A. Form $\boldsymbol{A}_{\boldsymbol{g}}^{(k)} = \boldsymbol{Ag}^{(k)}$. This is athe matrix-vector multiplcation.

   B. Form $\lambda^{(k)} = \rho^{(k)} / (\mathbf{g}^{(k)} \cdot \boldsymbol{A}_{\boldsymbol{g}}^{(k)})$. This is the dot product.

   C. Use $\lambda^{(k)}$ to update the solution vector, $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \lambda^{(k)}\boldsymbol{g}^{(k)}$. This step updates the solution vector; it is a daxpy.

D. Calculate $\boldsymbol{residual}^{(k+1)} = \boldsymbol{residual}^{(k)} - \lambda^{(k)} \boldsymbol{A}_{\boldsymbol{g}}^{(k)}$, the new residual. Check if the maximum element of $\boldsymbol{residual}^{(k+1)}$ is less than *epsilon* and if it is, stop. The following equation shows how the expression for $\boldsymbol{residual}^{(k+1)}$ develops from the residual's definition and the updated solution vector. This step updates the residual; it is another daxpy.

$$\boldsymbol{residual}^{(k+1)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k+1)} = \boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x}^{(k)} + \lambda^{(k)} \boldsymbol{g}^{(k)})$$

$$\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)} - \lambda^{(k)} \boldsymbol{A}\boldsymbol{g}^{(k)} = \boldsymbol{residual}^{(k)} - \lambda^{(k)} \boldsymbol{A}_{\boldsymbol{g}}^{(k)}$$

E. Save the old $\rho^{(k)}$ as $\rho_0^{(k)}$; that is, set $\rho_0^{(k)} = \rho^{(k)}$.

F. Now form a new $\boldsymbol{residual}^{(k+1)}$ squared, based on the new $\boldsymbol{residual}^{(k+1)}$ calculated in Step D. That is, set $\rho^{(k+1)} = \boldsymbol{residual}^{(k+1)} \cdot \boldsymbol{residual}^{(k+1)}$. This step is the other dot product.

G. Form $\beta^{(k)}$, the ratio of the $\boldsymbol{residual}^{(k+1)}$ squared with its previous value, $\beta^{(k)} = \rho^{(k+1)} / \rho_0^{(k)}$.

H. Update $\boldsymbol{g}^{(k+1)}$; that is, set $\boldsymbol{g}^{(k+1)} = \boldsymbol{residual}^{(k+1)} + \beta^{(k)} \boldsymbol{g}^{(k)}$. This step updates the search direction; it is a daxpy.

I. If the iteration count is less than some pre-determined value, return to Step A. Ideally, you leave the loop when the answer is less than *epsilon*, but the count is there to prevent hangs.

The description in Golub and Ortega [15] uses a slightly different notation. The differences are listed below.

$\boldsymbol{residual} \rightarrow \boldsymbol{r}$
$\boldsymbol{g} \rightarrow \boldsymbol{p}$
$\lambda \rightarrow -\alpha$

After the initial setup, each loop contains one matrix-vector multiplication, two dot products, and three daxpys.

Appendix D The NAS Benchmark

Introduction

The Numerical Aerodynamic Simulation (NAS) Conjugate Gradient (CG) benchmark [3] uses the power method to find an estimate of the smallest eigenvalue of a symmetric, positive definite sparse matrix $\boldsymbol{A}$. This matrix has a random pattern of nonzero values.

Assumptions

First, assume that a sparse symmetric positive definite matrix $\boldsymbol{B}$ has a set of n linearly independent eigenvectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$. Then,

$$\boldsymbol{Bx} = \lambda\boldsymbol{x}$$

Also, assume that the eigenvectors are normalized so that the infinity norm is 1. The infinity norm is defined as the absolute value of the largest component of a vector. For example the infinity norm of the m-dimensional vector $\boldsymbol{v}$ is

$$\|\boldsymbol{v}\|_\infty = 1 \qquad \text{where} \qquad \|\boldsymbol{v}\|_\infty = max_{i \le m}[|v_i|]$$

Finally, the eigenvalues are ordered by size, and the largest eigenvalue is distinct. Note that because the matrix is positive definite, the eigenvalues are positive [36], and it is not necessary to specify that their absolute values are used.

$$\lambda_1 > \lambda_2 \ge \lambda_3 \ge \ldots \ge \lambda_n$$

Power Method

The power method is an iterative procedure for finding λ_1, the largest eigenvalue. The remainder of this section (not the entire appendix) describes this procedure.

1. First make an initial guess for $\boldsymbol{x}_1$. Call this initial guess $v^{(0)}$. With the NAS CG benchmark, the initial guess is all ones.

$$v^{(0)} = [1, 1, \ldots, 1]$$

2. Because the eigenvectors are linearly independent, $v^{(0)}$ can be written as a linear combination of the eigenvectors such that the coefficients are not all zero.

$$v^{(0)} = \sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i$$

3. If both sides of this sum are multiplied on the left by the matrix $\boldsymbol{B}$, what results is a sum whose elements are $\boldsymbol{B}$ times an eigenvector. Define this as $z^{(1)}$. Each element in the sum can then be written as the eigenvalue times its eigenvector as follows:

$$z^{(1)} = \boldsymbol{B}v^{(0)} = \boldsymbol{B}\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i \boldsymbol{x}_i$$

4. What the benchmark provides as starting data, however, is $\boldsymbol{B}^{-1}$, not $\boldsymbol{B}$. So set $\boldsymbol{A} = \boldsymbol{B}^{-1}$. Then, solve the following equation for $z^{(1)}$ using the conjugate gradient method. This step describes the major portion of the calculation.

$$\boldsymbol{B}^{-1}z^{(1)} = \boldsymbol{A}z^{(1)} = v^{(0)}$$

5. Now set the absolute value of the largest element of $z^{(1)}$ to μ_1 and form $v^{(1)}$ as follows:

$$v^{(1)} = \frac{z^{(1)}}{\mu_1} \qquad \text{so that} \qquad \|v^{(1)}\|_\infty = \left\|\frac{z^{(1)}}{\mu_1}\right\|_\infty = 1$$

If k is the iteration number, $\mu_k \to \lambda_1$ as $k \to \infty$. λ_1 is the largest eigenvalue of $\boldsymbol{B}$ and the smallest eigenvalue of $\boldsymbol{A}$ where $\boldsymbol{A}$ is the given matrix. So the power method solves for the smallest eigenvalue of $\boldsymbol{A}$.

Output

The NAS CG benchmark specifies performing the power method iteration 15 times. That is, k goes from 1 to 15. At each iteration k, one must calculate μ, the Aitken extrapolation of the last three iterates.

$$\mu = \mu_{k-2} - \frac{(\mu_{k-1} - \mu_{k-2})^2}{\mu_k - 2\mu_{k-1} - \mu_{k-2}}$$

At each of the 15 iterations, one prints the following: k (the iteration number), μ (the Aitken extrapolation), and $\| r \|$ (the Euclidean norm of the last conjugate gradient residual vector). The Euclidean norm is defined as

$$\| r \| = \left(\sum_{i=1}^{n} r_i^2 \right)^{1/2}$$

The Matrix *A*

A is symmetric, positive definite, and sparse. It is of order 14000 and has 1 853 104 nonzero elements. The benchmark provides a Fortran program that generates ***A***. A test version has order 1400.

The sparse matrix storage scheme is called *compressed row.* The matrix ***A*** is stored in three one-dimensional arrays. These arrays are *a*, *rowidx*, and *colstr.* Figure D.1 illustrates this sparse storage scheme.

a	An array containing the actual elements of ***A***. The elements are grouped by column. That is, *a* contains all the ***A*** elements in one column followed by all those in another column. Lower number columns appear before higher number columns. The elements with the same column number have different row numbers, and these are not sorted.
rowidx	An array containing the row indices of *a.* That is, *rowidx*[*i*] contains the row index (in ***A***) of *a*[*i*].
colstr	An array containing the index of *a* that begins the column whose number is the index of *colstr.*

For example, consider column *i.* If *colstr*[*i*+1] - *colstr*[*i*] is nonzero, then column *i* has *colstr*[*i*+1] - *colstr*[*i*] nonzero values, and *colstr*[*i*] is the index of *a* that contains the first nonzero value in column *i.*

That is, the first nonzero element in column *i* is *a*[*colstr*[*i*]], and its row index is *rowidx*[*colstr*[*i*]]. Its location in the matrix is *A*[*rowidx*[*colstr*[*i*]] [*i*]. The second nonzero element in column *i* (if there is one) is *a*[*colstr*[*i*] + 1]. Its location in the matrix is *A*[*rowidx*[*colstr*[*i*] + 1] [*i*].

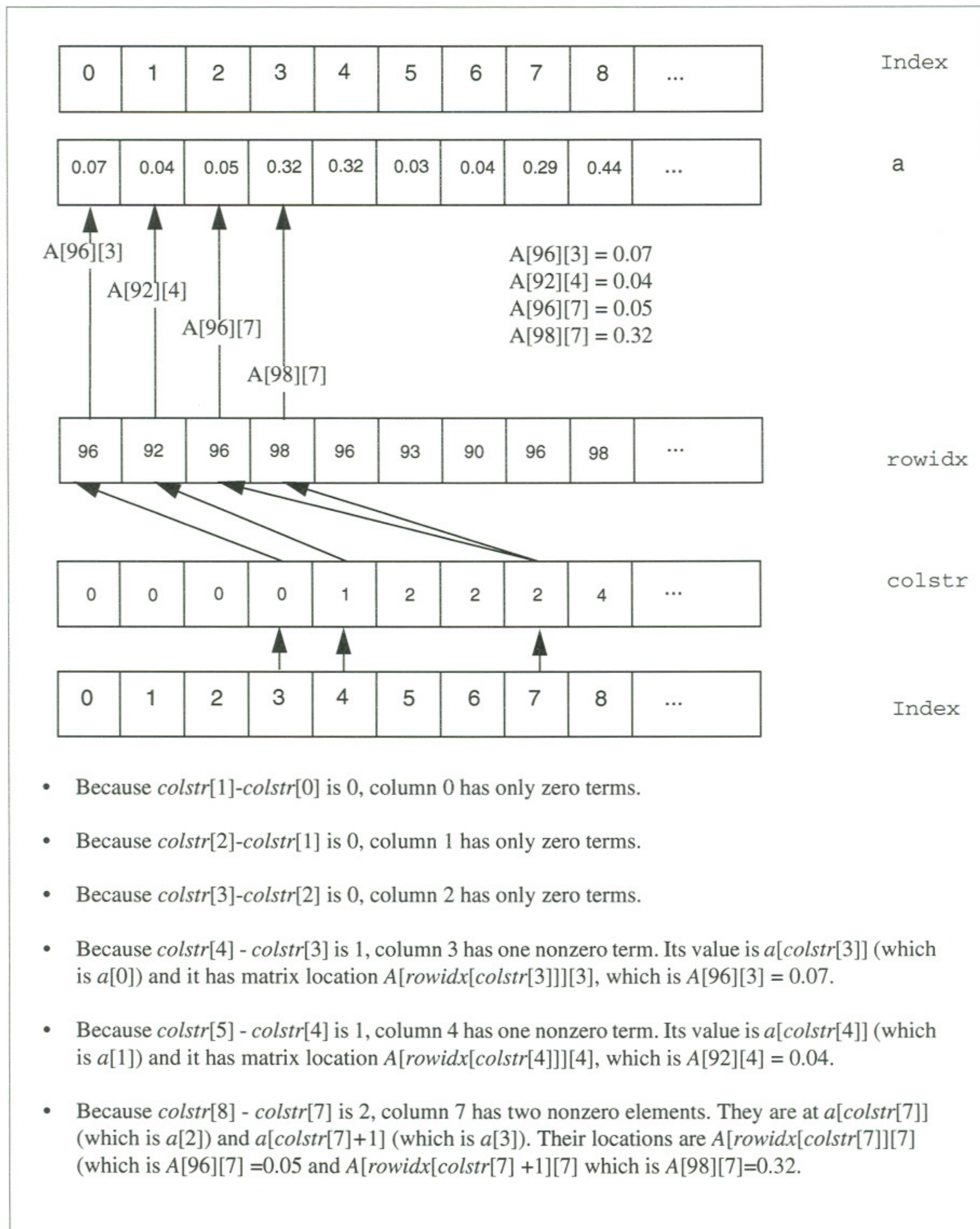


Figure D.1. Sparse Storage Scheme for Matrix A

Here is an algorithm (shown as C code) that prints out the elements of A. *I*n the code, *aorder* is the order of A, the number of rows (which is also the number of columns) in A.

```
for(j=0; j<aorder; j++) {
   myacol = j;
   for(i=colstr[j]; i<colstr[j+1]; i++) {
      myarow = rowidx[i];
      aval   = a[i];
      printf("A(%4d,%4d)= %7.5f\n",myarow,myacol,aval);
   }
}
```

The consequence of this storage scheme is that *a*[] and *rowidx*[] must have a length that is equal to the number of nonzero elements in ***A***. *colstr*[] must have a length that is equal to one more than the order of ***A***. The final value of *colstr*[], which is *colstr*[*aorder*] (indexing starts at 0) is such that *colstr*[*aorder*] - *colstr*[*aorder* - 1] is equal to the number of nonzero elements in the last column of ***A***, whose column index is *aorder* -1.

Row Decomposition of *A*

All the values of ***A*** do not appear on every node. ***A*** is distributed over the compute nodes. Deciding how to partition ***A*** can affect the performance of the algorithm.

The example here distributes ***A*** by rows over the compute nodes. How many rows each compute node gets is determined by the routine **get_range()**. This routine attempts to divide the rows equally among the compute nodes. If the number of compute nodes does not divide the number of rows evenly, each compute node beginning with compute node 0 gets an additional row until all the rows are used up.

For example, if 1400 nodes are divided among 13 compute nodes, compute nodes 0 through 8 each get 108 rows (9 * 108 = 972)and compute nodes 9 through 12 get 107 rows (4 * 107 = 428).

References

1. Almasi, George S., and Allan Gottlieb. 1989. *Highly parallel computing*. Redwood City: Benjamin/Cummings.

2. Amdahl, G. 1967. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the AFIPS Conference* 30:483-485,

3. Bailey, David, John Barton, Thomas Lasinski, and Horst Simon. January 1991. The NAS parallel benchmarks, Report RNR-91-002, NAS Systems Division, Applied Research Branch, NASA Ames Research Center.

4. Barrett, Richard, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. 1994. *Templates for the solution of linear systems: building blocks for iterative methods*. Philadelphia: SIAM.

5. Bickford, William B. 1990. *A first course in the finite element method*. Boston: Irwin.

6. Davies, A. J. 1980. *The finite element method: a first approach*, Oxford: Clarendon Press.

7. Dongarra, Jack J., Iain S. Duff, Danny C. Sorensen, and Henk A.van der Vorst. 1991. *Solving linear systems on vector and shared memory computers*. Philadelphia: SIAM.

8. Duff, I. S., A. M.Erisman, and J. K.Reid. 1986. *Direct methods for sparse matrices*. Oxford: Clarendon Press.

9. Duff, I. S., and J. K. Reid. 1983. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Softw*. 9:302-325.

10. Duff, I. S., 1986. Parallel implementation of multifrontal schemes. *Parallel Computing* 3:193-204.

11. Flynn, Michael J. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers* C-21:948-960,

12. Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon. D. Walker. 1988. *Solving problems on concurrent processors*. Englewood Cliffs: Prentice Hall.

13. Gallivan, K. A., Michael T. Heath, Esmond Ng, James M. Ortega, Barry W.Peyton, R. J. Plemmons, Charles H. Romine, A. H. Sameh, and Robert G.Voigt. 1990. *Parallel algorithms for matrix computations*. Philadelphia: SIAM.

14. George, Alan, and Joseph W. Liu. 1982. *Computer solution of large sparse positive definite systems*. Englewood Cliffs: Prentice-Hall.

15. Golub, Gene, and James M. Ortega. 1993. *Scientific computing, an introduction with parallel computing*, San Diego: Academic Press.

16. Golub, Gene, and Charles F. Van Loan. 1989. *Matrix computations*. Baltimore: The John Hopkins University Press.

17. Hawkinson, Stuart. 1994. Scalable Systems Division, Intel Corporation. Beaverton, Oregon. Private conversation.

18. Hestenes, M. R., and E. Stiefel. 1952. Methods of conjugate gradient for solving linear systems. *Journal Research National Bureau of Standards* 45:409-436.

19. Hord, Michael R. 1982. *The Iliac-IV: the first supercomputer*. Computer Science Press.

20. Irons, B. M. 1970. A frontal solution program for finite element analysis. *Internat. J. Numer Methods Engrg.* 2:231-239.

21. Jennings, Alan, and J. J. McKeown. 1992. *Matrix computation*. Chichester: John Wiley & Sons.

22. Kincaid, David, and Ward Cheney. 1991. *Numerical analysis*. Pacific Grove: Brooks/Cole.

23. Lewis, P. E., and J. P.Ward. 1991. *The finite element method, principles and applications*. Workingham: Addison-Wesley,

24. Lewis, John G., and Robert A.van de Geijn. 1993. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. *Proceedings Supercomputing '93* 484-492.

25. MacKeown, P. K., and D. J. Newman. 1987. *Computational techniques in physics.* Bristol: Adam Hilger.

26. Markowitz, H. M. 1957. The elimination form of the inverse and its application to linear programming. *Management Sci.* 3:255-269,

27. Moler, Cleve. 1987. A closer look at Amdahl's law. Intel Scientific Computers, Technical Note 02-0587 0288.

28. Otto, Steve W. 1994. Oregon Graduate Institute. Beaverton, Oregon. Class lecture.

29. Paragon XP/S Technical Summary. January 1994. Intel Supercomputer Systems Division. SSD9401R13N.

30. Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. 1992. *Numerical recipes in C.* Cambridge: Cambridge University Press.

31. Quinn, Michael J. 1994. *Parallel computing theory and practice.* New York: McGraw-Hill.

32. Sewell, Granville. 1988. *The numerical solution of ordinary and partial differential equations.* San Diego: Academic Press,

33. Smith, G. D. 1985. *Numerical solution of partial differential equations: finite difference methods.* Oxford: Clarendon Press.

34. Stoer, J., and R. Bulirsch. 1993. *Introduction to numerical analysis.* New York: Springer-Verlag.

35. Wait, R., and A. R. Mitchell. 1985. *Finite element analysis and applications.* Chichester: John Wiley & Sons.

36. Wilkinson, J. H. 1965. *The algebraic eigenvalue problem.* Oxford: Clarendon Press.

37. Zhu, Jianping. 1994. *Solving partial differential equation on parallel computers.* Singapore: World Scientific.

38. Zienkiewicz, O. C., and R. L. Taylor. 1989. *The finite element method, volume 1, basic formulation and linear problems*. London: McGraw-Hill.

Biography

Theodore Kubaska graduated from Brown University in 1969 with an A.B../Sc.B. in physics. He spent a summer with VISTA in Gary, Indiana, managed an undergraduate laboratory at a small college in Kentucky, and was a replacement physics teacher at a private school for almost a year.

Then, Ted moved to New Haven and worked at Yale for two years. He isolated, purified, and characterized bacterial enzymes for the Department of Molecular Biophysics. While living in New Haven, he married Lory Dawson, whom he had met in VISTA. Then, Ted and Lory moved to Orono, Maine where he became a graduate student in the Physics Department at the University of Maine. He received an M.S. in 1974.

After Maine, Ted and Lory moved to Troy, New York where Ted became a graduate student in the Physics Department at Rensselaer Polytechnic Institute. After five years at RPI, he accepted a job writing technical manuals for Data General in Westboro, Massachusetts. He worked there for three years and took courses at Worcester Polytechnic Institute.

In 1982 he moved to Portland, Oregon to work as a technical writer for Intel. For his company sabbatical he returned to Maine and taught a course on how to solve computational problems on parallel computers. He worked in a variety of divisions at Intel, and he is currently writing manuals for Intel's Scalable Systems Division. While at Intel, Ted enrolled at the Oregon Graduate Institute and studied computer science.

Index

A

Aitken extrapolation
- NAS CG 142

alias group 64
- characteristics 19
- construction 62, 66
- definition 18, 57
- inefficient for NAS matrix 90
- information needed for construction 64
- NAS has long messages 91
- NAS has unneeded additions 91
- shared memory 19, 57
- steps for making vector coherent 68
- symmetric 112
- used when making vector coherent 112
- weakly coherent 19, 57

alias list
- definition 66
- points to linked list 66
- used to calculated send and receive lists 67

aliasing
- still works for random matrix 85
- takes advantage of non-randomness of matrix 85

Amdahl's Law
- definition 16

assembled matrix 74
- definition 9
- **mat_local_assemble()** 74
- **mcomb()** 74
- **mlist()** 74
- not formed explicitly 9

B

balanced secretaries
- definition 94, 95
- NAS CG small 106

barrier synchronization 38

basis function
- as many as there are equations 3
- associated with node 3
- characteristics 3
- coefficient is value at node 3
- nodal 120
- zero on boundary 3

BBN 11

bisection width
- characteristic of network topology 12
- definition 12
- hypercube 13

boundary conditions
- Dirichlet 6
- natural 119
- suppressible 119
- Type 1 118
- Type 2 118

broadcast
- more efficient for random matrix 85

C

cache use
- memory allocations for efficient use 104

coherence
- definition for alias group 21
- example of making vector coherent 69
- steps for making vector coherent 68

coherence operation 112
- definition 58
- example as sum 66
- performed by secretary 67
- steps for making vector coherent 69

coherence time
- definition 102

communication
- overlap with computation 18

compressed row
- sparse storage 143

condition number
- definition 36

conjugate gradient
- choosing search direction 35
- convergence 83
- definition 34
- iterative method 19
- minimizing function 34
- NAS CG 86, 142
- non-random matrix 56
- non-stationary 34
- parallel 55
- positive definite matrix 34
- preconditioning 36
- residual 35
- search directions are ***A***-orthogonal 35
- search directions defined 34
- serial 34
- unstructured domain 56

Connection Machine CM-200 11

Connection Machine CM-5 11

constraints
- **add_constraints()** 81

contention
- message passing 102

control parallelism
- definition 39
- example of tree search 39
- manager node 39
- manager/worker decomposition 39

convergence
- conjugate gradient 83
- **gte()** 84

cost/performance ratio 13

Cray-1 10

Cray-YMP 10

D

data parallelism 18
- definition 38
- parallel iterative 48

diameter
- characteristic of network topology 12
- definition 12

direct methods
- parallel 39
- serial vs. parallel 39

directed graph
- definition 5

Dirichlet boundary conditions
- definition 6
- finite element triangular grid example 64

discretization
- partial differential equations 2

dot product
- definition 59
- global sum 77
- Kuck and Associates, Inc. 79
- single processor implementation 77
- time increases with number of compute nodes 102

drop tolerances 27

E

eigenvector
- linearly independent 142

elemental basis functions
- definition 116
- linear 114
- solution over element in terms of 114

elemental matrix 124

elimination tree
construction algorithm 42
definition 40
definition as algorithm 41
effects of reordering on 41
example for tridiagonal matrix 40
example of construction 43
parallelism at bottom 47
short broad vs. tall thin 41
tridiagonal matrix 43
two-dimensional graph 46

epsilon
conjugate gradient 100, 139

Euclidean norm 143

F

field node
definition 62

fill-in
confining 27
definition 26

finite difference
definition 2
derivative approximation 6

finite element
assembled matrix 9
basis functions 2
clique storage 9
definition 3, 114
dissection of mesh 47
elemental matrix 9
grid 9
mesh 3
solution segment 114
triangular 3
unassembled matrix 10
values at non-nodal positions 3

fixed grain speedup 17

flat interconnect
definition 13

Flynn taxonomy 11
not distiguish between shared and distributed memory 11

frontal method
definition 28, 44

full pivoting 26

fully-connected network 13

G

Galerkin method
definition 120
weak formulation 121

Gauss divergence theorem 119

Gaussian elimination
full pivoting not used 26
pivot element 25
remaining submatrix 25
see LU factorization
serial direct method 24
stable with partial pivoting 25
stages 25

Gauss-Seidel method
checkerboard ordering 54
compared to Jacobi 52
component form 32
matrix form 33
parallel 51
red-black ordering 54
serial 32
sparse 51
update equation 52
update order forces serialization 51
when to update guard 54

gdsum()
used in dot product 101

good locality 111

grid
finite difference 2
partitioning 62
structured 4
unstructured 4

grid refinement
no need for global renumbering 112

guard buffer
definition 53

guard buffers 111

H

hat function 122

hypercube
    bisection width 13
    dimension 13
    expansion 14

I

i860
    performance 13

incoherence
    definition for alias group 21

infinity norm
    definition 141
    NAS CG 141

integration by parts
    used in weak formulation 118

iPSC 13

iteration matrix
    definition 31
    matrix decomposition 49
    parallel 49

iterative method
    faster than direct 30
    less storage than direct 30
    parallelization 30

iterative methods
    parallel 48

J

Jacobi method
    broadcast vector 49
    compared to Gauss-Seidel 52
    component form 32
    dense matrix 49
    matrix form 32
    matrix-vector multiplication 49
    parallel 38, 49
    serial 31
    steps of 50
    update equation 52

K

Kuck and Associates, Inc.
    dot product 79

L

L2 norm
    used in condition number 36

Laplace matrix
    diagonal scaling does not help 37
    percentage nonzero 1

Laplace's equation 51, 111
    definition 5, 51
    finite difference approximation 52
    Gauss-Seidel method 54
    grid decomposition 53
    parallel Jacobi 52
    sparse matrix 51

Laplacian
    definition 6

linear interpolant
    definition 126

load balancing
    definition 17
    dynamic 17

load vector 122

local indexing 112
    ability to refine mesh 56

locality
    definition of good locality 85

loosely synchronous
    definition 2

low node number secretaries
    creates bottleneck for NAS CG 95
    definition 95

LU factorization
    back substitution 24
    definition 24
    forward elimination 24
    forward reduction 24
    row operation 25
    see Gaussian elimination
    stages 25

M

mailbag
    sending around a ring 66
    used to construct alias list 66

Markowitz criterion
    definition 27

matrix
- elements distributed 56
- partial differential equation 2
- sparse for partial differential equation 2
- structure 85

matrix element
- assembled 122
- elemental 122

matrix-vector multiplication 82

mesh routing chip 15

messages
- long messages desirable 53

MIMD
- definition 11

MISD
- definition 11
- example 12

mkpart
- creates partition 99

MRC 15

multicomputer
- definition 1

multifrontal method
- compared with frontal 44
- extension of frontal method 30
- generalization of frontal 44
- parallel 44
- tridiagonal matrix 44
- two-dimensional graph 47

N

NAS CG
- algorithm for determining field node location 87
- benchmark 86
- compute node 0 has only one nonzero column 90
- global vs. local index 88
- modifications to aliasing 93
- nonzero columns for sample problem over six compute nodes 90
- number of nonzero elements in matrix 86
- order of matrix 86
- pseudocode 105
- random matrix 86
- sample partitioned over six compute nodes 90

NAS CG benchmark
- has random matrix 85

NAS matrix
- diagonal element nonzero 87
- does not have good locality 86
- limiting case 92
- percentage nonzero 1

nested dissection
- definition 45
- one dimensional graph 45

no zeros
- definition 94
- NAS CG small 106

node
- service 99

node dependence
- existed in early versions of **vcomb()** 104
- reasno for not apparent 102

non-stationary methods
- definition 34

norm 141

NUMA
- definition 12

Numerical Aerodynamic Simulation, see NAS CG

O

ordering
- changing to reduce fill-in 26
- definition 26
- grid 5

out_sendlist
- pointers offsets into allocated chunk 104
- used by **vcomb()** 104

owner-computes method 59

P

Paragon
- MIMD 11
- two processors 18
- used for this work 12

Paragon supercomputer
- logging in 99

Paragon XP/S 11

parallel computers 10

partial pivoting 25

partition
    compute node 99

pivoting
    definition 25
    full 26
    operations 25
    partial 25

Poisson matrix
    diagonal scaling does not help 37

positive definite matrix
    conjugate gradient 34
    positive eigenvalues 141

power method
    steps of 141
    used in NAS CG 86

preconditioning
    definition 36
    diagonal scaling example 37

pull-in
    definition 20, 57
    parallel 58

R

random matrix
    definition 85

receive buffer
    used by receive list 70

receive list
    calculated from alias list 67
    definition 67
    example 68
    need for 67
    steps for making vector coherent 69

reordering
    goal more than fill reduction 39
    makes non-random nature apparent 85
    serial vs. parallel 39

residual 82
    definition 118, 139

row decomposition
    NAS CG 86

S

scalability
    definition 13

scaled speedup 17

secretary 112
    choosing 59
    definition 59
    perform coherence operation 67
    steps for making vector coherent 68

send buffer
    used by send list 70

send list
    calculated from alias list 67
    definition 67
    example 68
    need for 67
    steps for making vector coherent 68

send_corres
    used by **vcomb()** to decrease node dependence 104

Sequent 11

service node
    definition 99

service partition
    contains service nodes 99

shadow buffer
    definition 53

shared memory
    parallel computer 11

SIMD
    definition 11

similarity transformation
    definition 36

single bus network 13

Single Program Multiple Data, see SPMD 38

SISD
    definition 11

solution segment
    approximation to 115
    definition 114

sparse matrix
    storage 30

sparsity structure of matrix
    definition 42

spectral radius
  definition 36
  greater than 1 means convergence 31

speedup 15
  Amdahl's Law 16
  fixed grain 17

splitting matrix
  definition 36

SPMD program
  definition 38

stationary methods 31
  definition 31

structure
  matrix 85

structured grid 4

substructuring
  compared to nested dissection 47
  definition 47

successive overrelaxation method
  component form 33
  matrix form 33
  optimal ω 33
  serial 33

supernodes
  definition 46
  matrix blocks 47

Symmetry, Sequent
  MIMD 11

systolic array
  MISD 12

T

TC2000 11

topology
  characteristics 12
  definition for network 12

tridiagonal matrix
  column elimination is serial 40
  elimination tree 40
  no fill-in 39–40
  no parallelization 39–40

U

UMA
  definition 12

unassembled matrix
  definition 10

unstructured grid 4

unstructured matrix
  definition 56

V

vcomb()
  modifications due to no zeros 95

vector component
  aliased 57
  distributed 56
  incoherent 57

VLIW
  definition 11

W

weak formulation
  definition 118

weakly coherent 19

weakly-coherent, shared memory 111

weight function
  definition 118

weighted residuals
  method of 117