SCALABLE HYBRID SPARSE LINEAR SOLVERS

A Thesis in
Computer Science and Engineering
by
Keita Teranishi

© 2004 Keita Teranishi
Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2004
The thesis of Keita Teranishi was reviewed and approved* by the following:

Padma Raghavan  
Associate Professor of Computer Science and Engineering  
Thesis Adviser  
Chair of Committee

Mahmut Kandemir  
Associate Professor of Computer Science and Engineering

Lyle Long  
Professor of Aerospace Engineering

Panagiotis Michaleris  
Associate Professor of Mechanical Engineering

Paul Plassmann  
Associate Professor of Computer Science and Engineering

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

In many large-scale simulations that depend on parallel processing to solve problems of scientific interest, the application time can be dominated by the time for the underlying sparse linear system solution. This thesis concerns the development of effective sparse solvers for distributed memory multiprocessors using a hybrid of direct and iterative sparse solution methods. More specifically, we accelerate the convergence of an iterative solution method, namely the method of Conjugate Gradients (CG) using an incomplete Cholesky preconditioner. The latter is an approximation to the sparse matrix factor used in a direct method. Our parallel incomplete factorization scheme can support a range of fill-in to provide flexible preconditioning that can meet the requirements of a variety of applications. We have also developed special techniques that allow the effective application of such preconditioners on distributed memory multiprocessors; the relatively large latencies of interprocessor communication on such parallel computers make conventional schemes using parallel substitution extremely inefficient.

The first part of the dissertation focuses on the design of a parallel tree-based left-looking drop-threshold incomplete Cholesky factorization scheme using extensions of techniques from direct methods. The second part concerns modifications to the incomplete Cholesky factor to enable its efficient application as a preconditioner; these modifications concern selectively replacing certain triangular submatrices in the factor by their approximate inverses. We develop a ‘Selective Inversion’ (SI) scheme based on explicit inversion of selected submatrices and another variant using Selective Sparse Approximate Inversion (SSAI). The final part of the dissertation concerns latency-tolerant application of our ICT-SI and ICT-SSAI preconditioners by selectively using parallel matrix-vector multiplication instead of parallel substitution.

We analyze the computation and communication costs of all our schemes for model sparse matrices arising from finite difference methods on regular domains in two and three dimensions. We also provide extensive empirical results on the performance of our methods on such model matrices and others from practical applications. Our results demonstrate that both our ICT-SI and ICT-SSAI hybrid solvers are significantly more reliable than other preconditioned CG solvers. Furthermore, although their scalability lags that of some simpler schemes, they can still be the method of choice for matrices that require relatively strong preconditioning for CG to converge. Our analysis and experiments indicate that ICT-SSAI is more scalable than ICT-SI; however, our experiments indicate that this scalability is achieved at the expense of a slight decrease in preconditioning quality.

We have thus developed scalable and reliable hybrid solvers that can potentially provide significant improvements in the performance of modeling and simulation applications.
# Table of Contents

List of Tables ................................................. vii

List of Figures .................................................. viii

Acknowledgments ............................................... x

Chapter 1. Introduction ........................................ 1

Chapter 2. Sparse Linear System Solution ...................... 3
  2.1 Dense Direct Methods ..................................... 3
  2.2 Sparse Direct Methods .................................... 4
    2.2.1 Ordering ............................................. 4
    2.2.2 Symbolic Factorization ............................... 7
    2.2.3 Numeric Factorization ................................ 8
      2.2.3.1 Left-Looking and Multifrontal Schemes ........ 8
    2.2.4 Triangular Solution ................................ 10
    2.2.5 Parallel Sparse Direct Methods ..................... 10
  2.3 Iterative Methods ......................................... 11
    2.3.1 Conjugate Gradients ................................. 12
    2.3.2 Preconditioning ..................................... 13
    2.3.3 Incomplete Cholesky Preconditioning ............... 14
    2.3.4 Parallel Preconditioned Conjugate Gradients ...... 15
  2.4 Hybrid Sparse Linear Solution .......................... 15

Chapter 3. Parallel Incomplete Cholesky Preconditioners .... 16
  3.1 Left-Looking Drop-Threshold Incomplete Factorization ... 16
    3.1.1 Overview ........................................... 16
    3.1.2 Distributed Asynchronous Left-looking Fan-in Scheme 18
    3.1.3 Relationship to Domain-Decomposition and Multi-Level Linear Solvers .. 21
  3.2 Analysis of ICT Factorization Costs ..................... 22
    3.2.1 Analysis of Sequential ICT Factorization ............ 23
      3.2.1.1 Computational Costs at a Supernode ............ 23
      3.2.1.2 Computational Costs: Model $K \times K$ Grids ... 24
      3.2.1.3 Computational Costs: Model $K \times K \times K$ Grids .. 27
3.2.2 Analysis of Parallel ICT Factorization ............................................. 29
  3.2.2.1 Computational Costs at a Supernode .................................... 30
  3.2.2.2 Communication Costs a Supernode .................................... 30
  3.2.2.3 Computational Costs: Model $K \times K$ Grids ..................... 30
  3.2.2.4 Communication Costs: Model $K \times K$ Grids ..................... 33
  3.2.2.5 Computational Costs: Model $K \times K \times K$ Grids .......... 35
  3.2.2.6 Communication Costs: Model $K \times K \times K$ Grids .......... 36
3.3 Summary ............................................................................................................. 38

Chapter 4. Modifications to Enable Latency Tolerant Application of ICT Preconditioners
4.1 Introduction ......................................................................................................... 39
4.2 Parallel ICT Using Selective Inversion (ICT-SI) .................................................. 39
4.3 Parallel ICT Using Selective Sparse Approximate Inversion (ICT-SSAI) ............... 40
  4.3.1 Sparse Approximate Inverse Preconditioners ...................................... 40
  4.3.2 Selective Inversion Using Sparse Approximate Inversion ................. 43
  4.3.3 The ICT-SSAI Algorithm and its Implementation .............................. 44
4.4 Analysis of Parallel ICT with Selective Inversion (ICT-SI) ................................. 44
  4.4.1 Computational Costs at a Supernode .............................................. 45
  4.4.2 Communication Cost at a Supernode .............................................. 45
  4.4.3 Computational Costs: Model $K \times K$ Grids ................................. 46
  4.4.4 Communication Costs: Model $K \times K$ Grids ................................. 48
  4.4.5 Computational Costs: Model $K \times K \times K$ Grids ......................... 48
  4.4.6 Communication Costs: Model $K \times K \times K$ Grids ......................... 49
4.5 Analysis of Parallel ICT with Selective Sparse Approximate Inversion
  (ICT-SSAI) .............................................................................................................. 51
  4.5.1 Computational Costs at a Supernode .............................................. 51
  4.5.2 Communication Costs at a Supernode .............................................. 52
    4.5.2.1 Costs of Sparse Approximate Inversion (SAI) ...................... 52
    4.5.2.2 Costs of Sparse Matrix Vector Multiplication ...................... 52
    4.5.2.3 Costs of Rank-$k$ Updates ..................................................... 53
    4.5.2.4 Communication Costs of ICT-SSAI at a Supernode ............. 53
  4.5.3 Computational Costs: Model $K \times K$ Grids ..................................... 54
  4.5.4 Communication Costs: Model $K \times K$ Grids ..................................... 55
  4.5.5 Computational Costs: Model $K \times K \times K$ Grids .......................... 57
  4.5.6 Communication Costs: Model $K \times K \times K$ Grids .......................... 59
4.6 Summary .............................................................................................................. 61
  5.1 Distributed Memory Multiprocessors .................................. 62
  5.2 Test Matrices ............................................................. 63
    5.2.1 Model Grid Problems ............................................. 63
    5.2.2 Sparse Matrices from Practical Applications .................. 64
  5.3 Preconditioning Schemes ............................................... 65

Chapter 6. An Empirical Evaluation of Parallel Schemes to Construct Preconditioners .................................................. 68
  6.1 Scalability of Preconditioner Construction .......................... 68
  6.2 Performance on Sparse Matrices from Applications ............... 71
  6.3 Summary ....................................................................... 71

Chapter 7. Latency Tolerant Schemes for Applying ICT Preconditioners .......................................................... 73
  7.1 On Selectively Replacing Parallel Substitution by Parallel Matrix-Vector Multiplication ................................. 73
  7.2 One-way and Two-way Data Mapping for Reducing Communication Latency Costs ............................................ 76
  7.3 Analysis of Communication Costs ...................................... 77
  7.4 Empirical Study: Performance of Methods to Apply ICT Preconditioners ................................................... 83
    7.4.1 Performance of Preconditioned CG ............................. 84
    7.4.2 Communication Costs .............................................. 87
  7.5 Summary ....................................................................... 90

Chapter 8. On the Performance of Hybrid ICT-SI/ICT-SSAI Solvers .................................................. 91
  8.1 Test Conditions ............................................................ 91
  8.2 Scalability Results ......................................................... 92
  8.3 Reliability and Quality of Hybrid Linear Solvers on Sparse Matrices from Applications ................................... 92
  8.4 Summary ..................................................................... 107

Chapter 9. Conclusions ......................................................... 110

References ........................................................................ 118
List of Tables

5.1 Description of sparse test matrices used for testing the scalability of preconditioner construction. ................................................. 64
5.2 Description of sparse test matrices used for testing the scalability of preconditioner application. ................................................. 65
5.3 Description of sparse matrices from practical applications. ............ 66
8.1 Description of ten test matrices. ................................................. 98
8.2 Description of additional three test cases. ..................................... 100
9.1 Summary of computational and communication costs (dominant terms) for parallel ICT factorization schemes on sparse matrices from 2 and 3 dimensional model grid problems using $P$ processors. ................. 112
9.2 Scalability of parallel sparse linear solvers for sparse matrices from two-dimensional model grid problems. .............................. 113
9.3 Scalability of parallel sparse linear solvers for sparse matrices from three-dimensional model grid problems. ......................... 114
9.4 Summary of the performance of parallel sparse linear solvers for three large sparse matrices from applications. ......................... 115
List of Figures

2.1 A left-looking Cholesky factorization scheme. ...................... 3  
2.2 A sequence of five vertex-elimination steps for the graph of a small $5 \times 5$ sparse matrix. ....................................... 5  
2.3 A sequence of vertex-elimination steps after the original matrix is permuted. 6  
2.4 A supernodal tree and its corresponding elimination tree. .......... 7  
2.5 Left-looking sparse Cholesky factorization. .......................... 8  
2.6 Data dependencies in supernodal left-looking Cholesky factorization. 9  
2.7 Data dependencies in multifrontal Cholesky factorization. .......... 10  
2.8 The method of Conjugate Gradients. ................................. 12  
2.9 Preconditioned Conjugate Gradients. .................................. 13  
3.1 Basic left-looking drop-threshold IC factorization scheme (ICT). 17  
3.2 Sequential scheme for drop-threshold IC factorization using the supernodal tree. ............................................. 19  
3.3 Parallel drop-threshold IC factorization using the supernodal tree. 20  
3.4 Two levels of separators of an domain (graph) using nested dissection and the supernodal elimination tree. .......................... 21  
3.5 The separators (sides) in a model 2-dimensional grid. ............... 24  
3.6 Subgrids with different borders (separators). ....................... 25  
3.7 The 3 separator planes for the 3D cubic grid. .......................... 27  
3.8 Supernodes and the local-phase subtree (colored) computed by processor 0 in a 4 processor implementation of parallel ICT. ............ 31  
4.1 Parallel ICT factorization with Selective Inversion (ICT-SI). ........ 41  
4.2 Matrix associated with a distributed supernode for ICT-SSAI and ICT-SI. 43  
6.1 The cost of preconditioner construction. .............................. 69  
6.2 Time for constructing preconditioners on a suite of 21 sparse matrices. 70  
6.3 Time for preconditioner construction for augustus7 using 4-64 processors. 72  
7.1 Distributed forward solution at a supernode (A) with selective inversion and the one-way data mapping (DM-1) using 4 processors. ....... 75  
7.2 Triangular solution with 1-way mapping (DM-1). ........................ 77
7.3 Distributed forward solution at a supernode (A) with selective inversion and the two-way data mapping (DM-2) using 4 processors. 78
7.4 Triangular solution with 2-way mapping (DM-2). 79
7.5 Separators for 2 dimensional grid. 81
7.6 The 3 separator planes for the 3D cubic grid. 82
7.7 Number of iterations of the four preconditioning schemes. 85
7.8 Total time for applying the preconditioner over all iterations. 86
7.9 Time (in seconds) to apply the preconditioner in one CG iteration. 88
7.10 The number of messages sent by a preconditioner in each CG iteration. 89
8.1 The total time for the solution of the 2D and 3D grids. 93
8.2 The number of iterations for the solution of the 2D and 3D grids. 94
8.3 The time for PCCG iterations on the 2D and 3D grids. 95
8.4 The ratio of number of nonzero elements in preconditioners relative to the number of nonzero elements in the coefficient matrix from 2 dimensional grids. 96
8.5 The ratio of number of nonzero elements in preconditioners relative to the number of nonzero elements in the coefficient matrix from 3 dimensional grids. 97
8.6 The success ratio of preconditioned iterative schemes: convergence within 600 iterations (left) and within 2000 iterations (left). 100
8.7 Number of iterations to solve the 10 and 13 problems. 101
8.8 Total time for solving a single right-hand side vector associated with the 10 matrix problems (relative to the direct Cholesky solver). 102
8.9 Total time for solving a single right-hand side vector associated with the 13 matrix problems (relative to the direct Cholesky solver). 103
8.10 Total time for solving 10 right-hand side vectors associated with the 10 matrix problems (relative to the direct Cholesky solver). 104
8.11 Total time for solving 10 right-hand side vectors associated with the 13 matrix problems (relative to the direct Cholesky solver). 105
8.12 The nonzero elements distribution of bodyy5. 106
8.13 The average memory cost of the parallel solution schemes relative to the memory required to store A (set to 1). 106
8.14 Total solution time using parallel preconditioned CG for augustus7: the dashed line indicates the time required by the sparse direct solver with 64 processors. 108
8.15 The number of iterations to solve augustus7. 108
8.16 Total solution time for 10 right-hand sides using parallel preconditioned CG for augustus7: the dashed line indicates the time required by the sparse direct solver with 64 processors. 109
Acknowledgments

I wish to thank my advisor Padma Raghavan for her guidance. I have had her as my advisor since I started working on my masters degree at the University of Tennessee. Padma has always been available to answer and discuss my questions. Her enthusiasm and broad knowledge in scientific computing has motivated me to pursue my career in research in computational science and engineering.

I thank Esmond Ng who offered me an internship during the summer of 2002 at the National Energy Research Scientific Computing Center in the Lawrence Berkeley National Laboratory. He guided me while I was developing my code on the IBM SP Seaborg supercomputer, and has watched over my progress since the internship.

The final test code would not have been possible without help from Barry Smith and Satish Balay at the Argonne National Laboratory. Barry was advising me during my appointment at the Argonne National Laboratory in Fall 2002. He has provided me accounts on the Jazz Linux Cluster and has supported my efforts to complete this thesis.

I gratefully acknowledge Edmond Chow of Lawrence Livermore National Laboratory who has explained the performance characteristics of his parallel sparse approximate inverse code (Parasails); his explanation helped my analysis of my preconditioning methods on model problems.

I thank Michele Benzi of Emory University for providing his sparse matrix collection. I also thank Jack Dongarra from the University of Tennessee for offering an opportunity to work on a project related to LAPACK and ScaLAPACK when I was in my junior year in the undergraduate program. This project was the main reason for my interest in scientific computing and computational science and engineering.

I thank Mahmut Kandemir, Panagiotis Michaleris, Lyle Long, and Paul Plassmann for serving on my doctoral committee.

I truly thank my mother Masako Teranishi for her constant support during my nine years of study away from home in the U.S.A.
Chapter 1

Introduction

The solution of large sparse linear systems is required in many scientific and engineering applications. Consequently, efficient and scalable sparse solution methods have been studied for many years. More recently, with the wide-spread use of multiprocessor computers and networks of workstations, the development of scalable parallel solvers for sparse linear systems has became a more critical research issue. Solution methods can be broadly classified as iterative or direct methods, and we attempt to construct an efficient solution scheme as a hybrid of both classes of methods. Our work primarily concerns the solution of Symmetric Positive Definite (SPD) linear systems. However, our results can also be extended to the solution of non-symmetric linear systems using standard techniques [9, 98].

Consider the solution of a linear system $Ax = b$ where $A$ is a large, sparse, SPD matrix. An iterative solution method such as Conjugate Gradients (CG) [58] requires no extra memory beyond the coefficient matrix and a few vectors, and it allows a natural scalable parallel implementation. The major weakness of such an iterative method is its lack of robustness, i.e. convergence can be slow and a sufficiently accurate solution is often not achieved within a limited number of iterations. On the other hand, a direct method (based on matrix factorization) is robust, but computational requirements and memory overheads grow non-linearly with the dimension of the matrix. Furthermore, the efficient implementation of a direct solver is inherently complex because of the necessity of controlling and managing fill-in, i.e. zeroes in the coefficient matrix that become non-zeroes during factorization.

Preconditioning is a well-known technique that is used to accelerate the convergence of an iterative method by changing the spectral properties of the coefficient matrix. Improvements in the convergence rate typically offset the overheads associated with the construction and application of the preconditioner. Ideally, a preconditioned iterative method requires less memory than a direct method, and is more robust than an iterative method.

One of the most general-purpose preconditioning schemes is an incomplete Cholesky (IC) preconditioner [24, 77, 78]. This scheme constructs an incomplete factor $L$, which
is an approximation to the factor $L$, where $A = LL^T$. $\hat{L}$ is constructed by discarding some nonzero entries of $L$ during factorization, and applied using triangular solution in every iteration of Conjugate Gradients. Despite its popularity in the uniprocessor environment, parallel IC preconditioning (with mechanisms to support a wide range of fill) poses several challenges. First, if parallel substitution is used to apply such a preconditioner, then the relatively large latencies of interprocessor communication on distributed memory multiprocessors, lead to an inefficient implementation. Second, the implementation of a parallel incomplete factorization scheme is inherently complicated because of the problem of managing nonzero elements that are introduced and dropped during factorization. We construct an efficient hybrid direct-iterative solver by exploiting techniques from parallel sparse direct solvers to address these issues.

The rest of the dissertation is organized as follows. Chapter 2 provides background on the solution of sparse linear systems and the computing environment used in our experiments. Chapter 3 describes our parallel IC factorization scheme and analyzes its performance on some model problems. Chapter 4 concerns modifications to our parallel IC factorization scheme to enable a latency-tolerant application. Chapter 5 describes the environment in which our experiments were conducted and an overview of parallel sparse linear solvers developed by other researchers. Chapter 6 provides an empirical study of the performance of our parallel IC factorization schemes described in Chapters 3 and 4. Chapter 7 describes methods for latency-tolerant application of our IC preconditioner with analytical results on model problems and an empirical evaluation of their performance. Chapter 8 concerns an extensive empirical evaluation of the performance of our hybrid solver on a large suite of sparse matrices, and it includes comparisons with other types of parallel solvers. Chapter 9 contains concluding remarks and discusses potential extensions of our methods.
Sparse Linear System Solution

In this chapter, we review methods for the solution of sparse linear systems. More specifically, we focus on methods related to incomplete Cholesky (IC) preconditioning. Our overview concerns techniques for the computer solution of linear systems [31, 45], sparse direct methods [33, 41], and techniques for the iterative solution of sparse linear systems [9, 98].

2.1 Dense Direct Methods

A direct method for Symmetric Positive Definite (SPD) matrices is based on the Cholesky factorization $A = LL^T$, a symmetric variant of LU factorization for general square matrices. After factorization, the solution $x$ is obtained through triangular solutions, $Ly = b$ and $L^T x = y$. As shown in Figure 2.1, the algorithm consists of a triply nested loop. Consequently, 6 variants of the implementation can be obtained by permuting the loop order [30, 31].

For dense matrices, Dongarra et al. investigated the performance implementation corresponding to each permutation of the nested loop [30] on vector processors. Their

```
1: for i = 1 to N do
2:   for j = 1 to i - 1 do
3:     for k = i to N do
4:       A(k, i) ← A(k, i) - L(i, j) · L(k, j)
5:     end for
6:   end for
7:   A(i, i) ← √A(i, i)
8: for k = i + 1 to N do
9:   L(k, i) ← A(k, i) / A(i, i)
10: end for
11: end for

Fig. 2.1. A left-looking Cholesky factorization scheme.
```
effort was focused on increasing data reuse within the vector registers so that floating point operations are performed at near machine peak execution rates. Their ideas were extended further to allow effective use of the memory hierarchy for RISC architectures. The current state-of-art implementations [4, 52] use a block algorithm where the active portion of the matrix is packed within the cache, and computed using efficient dense kernels [28, 29].

2.2 Sparse Direct Methods

A sparse direct method takes advantage of the zeroes that comprise a large fraction of the matrix elements to store and operates only on nonzero entries. It can result in order of magnitude reductions in memory requirements and computational costs. For instance, the computational costs of sparse Cholesky factorization is \( O(N^{1.5}) \) for a certain class of sparse matrices whereas using the dense scheme would cost \( O(N^3) \) operations [41]. Despite the relative simplicity of the original dense Cholesky factorization algorithm, the design of a sparse factorization scheme is significantly complex primarily because of extra features needed to control and manage nonzeroes that are introduced during sparse factorization (called fill-in) [33, 41]. Most common techniques attempt to carefully reduce the amount of fill introduced in the triangular factor \( L \), and they adapt very efficient dense matrix techniques for the actual computation of \( L \) [33, 41, 85]. For SPD matrices, those techniques have been extensively studied in recent decades, and efficient implementations are available in several software packages [7, 80, 93]. In general, the solution process can be divided into the following four steps [41]

1. Ordering: to compute a symmetric permutation \( P \) of \( A \) in order to reduce the number of nonzeroes in \( L \), where \( PAP^T = LL^T \).
2. Symbolic factorization: to determine the structure of \( L \).
3. Numeric factorization: to compute the Cholesky factor \( L \).
4. Triangular solution: to solve two linear systems, \( Ly = Pb \) and \( L^T z = y \) to obtain \( x = P^T z \).

2.2.1 Ordering

The ordering step computes a symmetric permutation \( P \), so that the Cholesky factor \( L \), \( (PAP^T = LL^T) \), incurs less fill. The problem of computing an optimal permutation is known to be NP-complete [109].
Ordering heuristics operate on $G(A)$, the graph of a symmetric sparse matrix $A$. $G(A)$ is undirected with as many vertices as the number of columns of $A$; an edge $(i,j)$ represents the nonzero entry $A(i,j)$. Cholesky factorization of $A$ can be simulated by a sequence of vertex-elimination steps in $G(A)$ [41]. Figure 2.2 describes the vertex-elimination steps for the graph induced by a 5-by-5 matrix, where factorization of column $i$ is structurally equivalent to eliminating a vertex numbered $i$. Once a vertex is eliminated, its adjacent vertices are pairwise connected to each other; extra edges denoted by dashed-lines in $G(A)$ and ⊘ in $A$ (in Figure 2.2) correspond to fill-in.

![Graph of a 5-by-5 sparse matrix with vertex-elimination steps](image)

Fig. 2.2. A sequence of five vertex-elimination steps for the graph of a small $5 \times 5$ sparse matrix.

The amount of fill-in can be reduced by renumbering vertices. Figure 2.3 shows the vertex-elimination steps with the renumbered vertices for the example shown in Figure 2.2. It indicates that renumbering avoids all fill-in incurred by the elimination with the original vertex numbering.
The process of vertex-elimination is best described using the notion of fill-paths proposed by Rose et al. [95]. A path between \( i \) and \( j \) in \( G(A) \) is called a fill-path if all intermediate vertices are numbered less than \( \min(i,j) \); now \( L(i,j) \neq 0 \) if and only if \( G(A) \) contains a fill path between \( i \) and \( j \) [95].

Two major classes of fill-reducing ordering schemes can be explained easily in terms of fill-paths. The first class consists of methods that reduce fill-in by a greedy approach [2, 39, 72, 86]. This approach simulates Cholesky factorization on \( G(A) \) by selecting vertices that introduce a minimal number of fill edges or a minimal number of operations in their rank-1 update at each elimination step. Upon deletion of a vertex, its higher numbered neighbors are made pairwise adjacent, and the process is repeated until all vertices are eliminated. The second class uses a divide-and-conquer process. A vertex separator of \( G(A) \) is computed to partition the original graph into multiple subgraphs. The vertices in the vertex separator are assigned higher numbers to destroy fill-paths between the subgraphs induced by the partitioning process. This process is then applied recursively to each subgraph [40, 42]. Hybrid orderings that utilize both classes of methods have also been studied [6, 54].
2.2.2 Symbolic Factorization

Symbolic factorization analyzes the re-ordered graph $G(PAP^T)$ to determine the nonzero structure of $L$. Knowing the structure of $L$ can help to determine if there is sufficient memory to store $L$ or not, and to pre-allocate data structures for $L$ to allow efficient numeric factorization. The structure of $L$ is not computed by the simulation of factorization sequences through vertex elimination. Instead, an important data structure called the elimination tree is used to reduce the cost of such analysis.

The elimination tree [74] defines the column dependencies of the Cholesky factorization of $A$. Each node of the tree represents a column/row and an edge represents the first off-diagonal element of each column. The tree itself is a transitive reduction of the undirected graph $G(F)$ where $F = L + L^T$. The tree is computed efficiently using disjoint set and union operations [73, 74] with computational cost $O(|A| \alpha(|A|, N))$, where $|A|$ is the number of nonzeros in $A$ and $\alpha(|A|, N)$ is an inverse of Ackermann’s function which is less than the constant of 5 in practice. Therefore, the cost of elimination tree construction is nearly linear with respect to $|A|$. A supernode [8, 76, 90] is a set of contiguous columns that have the same nonzero row pattern. The supernode finding algorithm contracts multiple nodes (in a path) of the elimination tree into a single node. Each supernode can be represented as a dense submatrix to allow the use of dense matrix kernels [28, 29] to perform efficient numeric factorization.

A supernode finding algorithm can be implemented efficiently in conjunction with elimination tree construction as shown by Liu and Peyton [76]. Their algorithm is based on the contraction of the chain of tree nodes where every node except the one at the lowest level has only one child. Figure 2.4 shows a simple example of supernodal-tree and its corresponding elimination-tree. The supernodal tree enables numeric factorization to be performed by supernodal blocks obtained through a post-order traversal. In addition, a few supernodes can be combined to form a larger supernodal matrix [8] to further utilize efficient dense matrix kernels at the expense of extra memory and floating point operations.

Fig. 2.4. A supernodal tree and its corresponding elimination tree.
2.2.3 Numeric Factorization

Numeric factorization involves computing elements of $L$ using the data structure determined by symbolic factorization. The algorithm uses a post-order traversal of the supernodal tree; each small dense matrix associated with a supernode is factored using dense matrix techniques.

Consider the left-looking sparse Cholesky factorization shown in Figure 2.5. As described in lines 3–5, the left-looking update only needs to access the sparse columns that modify the current column $A(:,i)$. In a typical sparse Cholesky factorization, the management of such data dependencies is handled using either a left-looking or a multifrontal scheme based on the supernodal tree.

```
1: for $i = 1$ to $N$ do
2:     for $j = 1$ to $i - 1$ do
3:         if $L(i,j) \neq 0$ then
4:             $A(:,i) ← A(:,i) - L(i,j) * L(i:N,j)$
5:         end if
6:     end for
7:     $L(i,i) ← \sqrt{A(i,i)}$
8:     $L(i + 1 : N,i) ← A(i + 1 : N,i)/L(i,i)$
9: end for
```

Fig. 2.5. Left-looking sparse Cholesky factorization.

2.2.3.1 Left-Looking and Multifrontal Schemes

The left-looking scheme [41, 85] uses the earlier columns (column to the left) for the modification of the current columns before computing the Cholesky factor of current column. With a supernodal tree, the modification is performed by some columns associated with descendant nodes of the subtree rooted at the current node as sketched in Figure 2.6. The updating operation is performed by dense blocks to replace sparse scalar-vector multiplication by dense matrix-matrix multiplication and thus improve the performance from better data reuse in a cache [85]. The result of the matrix-matrix product is usually stored in a temporary buffer and added to the current supernodal matrix by sparse matrix addition.
Fig. 2.6. Data dependencies in supernodal left-looking Cholesky factorization.

The multifrontal scheme [34, 75] does not directly access descendant supernodes for updating a current supernodal matrix. Unlike a left-looking scheme, a dense lower trapezoidal submatrix corresponding to columns in a supernode is extended to a triangular matrix called the *frontal matrix*. In each frontal matrix, a dense Cholesky factorization is described as follows:

\[
\begin{bmatrix}
A_{11} & 0 \\
A_{12} & F
\end{bmatrix} \rightarrow \begin{bmatrix}
L_{11} & 0 \\
L_{12} & \hat{F}
\end{bmatrix}.
\]

\(A_{11}\) and \(A_{12}\) represent block elements that correspond to current supernodal matrix, and \(F\) accumulates the rank-\(k\) update by \(L_{12}\) as follows:

\[
\hat{F} = F - L_{12}L_{12}^T.
\]

The benefit of the multifrontal scheme is that the update is performed within the same dense matrix buffer. It saves memory accesses of the earlier columns to typically provide a faster implementation. However, extra storage is required to store \(\hat{F}\) in each frontal matrix at ancestors. Figure 2.7 illustrates the data access pattern of the multifrontal scheme.
Fig. 2.7. Data dependencies in multifrontal Cholesky factorization.

The performances of both methods vary with the nonzero pattern of the sparse matrix. The multifrontal scheme is more efficient in terms of the floating point operation rate, but the left-looking scheme requires less memory. Ng and Peyton [85], and Rothberg [96] claim that the left-looking scheme is more efficient for some classes of sparse matrices.

2.2.4 Triangular Solution

The supernodal tree can also be used to support suitable traversal to implement triangular solution. The forward solution is, like the numeric factorization, performed from the leaf nodes to the root node in a post-order traversal sequence, and the backward-solution is performed according to a pre-order traversal of the tree from the root node to the leaf nodes. Like numeric factorization, dense matrix kernel can be exploited to access each supernodal matrix, but the performance gain is not substantial because triangular solution requires far fewer operations than numeric factorization (the number of operations is proportional to the number of nonzeros).

2.2.5 Parallel Sparse Direct Methods

Parallelizing sparse direct methods has been an important research issue for many years [3, 53]. A parallel implementation comprises parallel computation of the four main
steps. Among these four steps, the numeric factorization has received greater attention for parallelization because it is typically the most expensive part of the solution process.

A numeric factorization is parallelized using the supernodal tree [38, 53, 91]. The tree is recursively mapped to processors by assigning disjoint subsets of processors of size proportional to the computational costs of subtrees, until a single processor is assigned to a subtree. The latter is called the local phase subtree and it corresponds to a sequential version of sparse Cholesky factorization. The supernodes located above the root of a local phase subtree are computed by a parallel dense matrix algorithms [16, 106]. Scalable numerical factorization have been achieved by several implementations [7, 55, 80, 93].

In contrast, it is very difficult to achieve a good efficiency for parallel triangular solution. Parallel substitution for a submatrix at a supernode is often inefficient due to the large latency of interprocessor communication. One solution concerns a clever pipelining to overlap communication and computation [35, 67, 68]. Another approach uses inversion of $L$ to allow matrix vector multiplication instead of substitution. Alvarado et al. developed the parallel partitioned inverse scheme, which splits $L$ into a number of easily invertible block elementary matrices [1], and Raghavan proposed the Selective Inversion [92] scheme in which dense triangular submatrices at distributed supernodes are explicitly inverted.

2.3 Iterative Methods

Iterative methods iteratively refine a solution vector initialized to $x_0$ using suitable matrix-vector multiplications. The main benefit of the iterative methods is that the memory requirement is no more than the space for the coefficient matrix $A$. In addition, if convergence is fast, the total computational cost is expected to be far smaller than for direct methods.

There are two major classes of the methods: stationary and Krylov subspace methods [9, 45, 98]. Stationary methods are based on constructing an iteration matrix $M$ from the coefficient usually by matrix-splitting. The convergence rate of stationary methods depends on the spectral radius of the iteration matrix $M$ given by $\rho(M)$, where $\rho(M)$ is the maximum magnitude of the eigenvalue of $M$. The Successive Over Relaxation (SOR) [110] is the most popular stationary method; it performs well in practice, especially when the user can select the relaxation parameter to be close to the optimal value.

Krylov subspace methods such as the Conjugate Gradients [58] (CG), form another popular class of iterative methods. CG is guaranteed to reach convergence at $N$ iterations, where $N$ is the dimension of a matrix. However, in finite precision arithmetic, it typically fails to converge. Additionally, the convergence behavior depends on the spectral properties of the coefficient matrix. In practice, Krylov subspace methods are implemented with “preconditioning.” Preconditioning is an operation aimed at improving the spectral property of the coefficient matrix. With the advent of preconditioning, the CG method became a more popular iterative solution than SOR. In the rest of the section, our discussion focuses on the CG method and related issues.
2.3.1 Conjugate Gradients

The Conjugate Gradients method [58] (CG) computes a solution $x_i$ by minimizing the A-norm, $(x - x_i)^T A (x - x_i)$, over all vectors in a Krylov subspace defined by the span of vectors $\{r_0, Ar_0, A^2 r_0, \ldots, A^{i-1} r_0\}$; $r_0$ is a initial residual obtained as $b - Ax_0$ (where $x_0$ is the initial guess to the solution). At successive iterations, the solution is refined by $x_i \leftarrow x_{i-1} + \alpha_i p_i$ where $p_i^T A p_j = 0$ for all $i \neq j$ (this property is called conjugate or A-orthogonal). The CG method is described in Figure 2.8. The method only requires a single matrix-vector multiplication and a few vector dot products per iteration.

1: $r_0 \leftarrow Ax_0 - b$
2: $p_0 \leftarrow r_0$
3: for $i = 1$ until convergence do
4: \quad $q_i \leftarrow Ap_i$
5: \quad $\gamma_i \leftarrow r_i^T r_i$
6: \quad $\alpha_i \leftarrow \frac{\gamma_i}{p_i^T q_i}$
7: \quad $x_{i+1} \leftarrow x_i + \alpha_i p_i$
8: \quad $r_{i+1} \leftarrow r_i + \alpha_i q_i$
9: \quad $\beta_{i+1} \leftarrow \frac{r_{i+1}^T r_{i+1}}{\gamma_i}$
10: \quad $p_{i+1} \leftarrow r_{i+1} + \beta_{i+1} p_i$
11: end for

Fig. 2.8. The method of Conjugate Gradients.

According to the analysis by Paige [89], the A-norm error during CG iterations is defined as:

$$
\| x_{i+1} - x^* \|^2_A \leq \left( \frac{\lambda_{n-i} - \lambda_1}{\lambda_{n-i} + \lambda_1} \right)^2 \| x_0 - x^* \|^2_A,
$$

where $\lambda_i$ is the $i$th smallest eigenvalue of $A$ and $x^*$ is the real solution. This implies that the CG iteration converges quickly if eigenvalues are highly clustered.

The overall convergence rate of the CG iteration is related to the condition number of the coefficient matrix ($\kappa(A)$) defined as:

$$
\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1}
$$
2.3.2 Preconditioning

The CG method often fails to converge to the solution because of loss of orthogonality due to errors in finite precision computer arithmetic [27, 47]. In addition, a bad distribution of eigenvalues (for example, uniform distribution) makes the convergence very slow [27, 100]. To alleviate such deficiencies, the original linear system is preconditioned to obtain a new linear system suitable for the CG iteration.

If $M$ is a preconditioning matrix, the preconditioner is applied to the system $Ax = b$ in the following manner:

\[ MAx = Mb. \]

Therefore, the convergence rate of the CG method for the new system is defined by:

\[
\frac{\sqrt{\kappa(MA)} - 1}{\sqrt{\kappa(MA)} + 1},
\]

where $\kappa(MA)$ is smaller than $\kappa(A)$. In practice, the preconditioner $M$ is not directly applied to the coefficient matrix because the result of the matrix product $MA$ is typically denser than $A$. Instead, $M$ is applied to the residual vector during the CG iterations as described in Figure 2.9.

Fig. 2.9. Preconditioned Conjugate Gradients.

The preconditioner $M$ can be any matrix, but it is usually an approximation of $A^{-1}$. Once $M$ is computed, $M$ is applied repeatedly in every CG iteration. One of the
simplest preconditioning methods is obtained by applying the inverse of the diagonal elements; it is called Jacobi preconditioning. Many good reviews of preconditioning schemes are available [15, 32, 98].

### 2.3.3 Incomplete Cholesky Preconditioning

To improve the convergence of CG, the ideal preconditioner is the exact inverse of \( A \), which is obtained applying the exact Cholesky factor of \( A \). A possible way to take advantage of this observation is to use \(^\sim L\), an approximation of the Cholesky factor as a preconditioner. This is called “incomplete Cholesky preconditioning” (henceforth IC) [24, 77, 78, 81]. An IC preconditioner is obtained by ignoring some or all of fill-in during the numeric factorization of \( A \). Note that the IC factorization is easily extended to incomplete LU factorization for unsymmetric matrices [98].

IC preconditioners have two major variants: level-of-fill IC and drop-threshold IC. The level-of-fill IC is best described using the notion of fill-paths in the graph model of sparse Cholesky factorization [41] (as explained in the previous sections). IC with level-of-fill (henceforth ICF(\( k \))) retains any fill element derived from a fill-path of length \( \leq k \). When \( k = 0 \), it implies no fill; the nonzero pattern of \( A \) is used for incomplete factorization. In general, the nonzero pattern can be computed before the numeric factorization and the nonzero pattern can be utilized for static storage allocation. However, techniques such as the supernodal elimination tree are not available to efficiently determine the nonzero pattern.

The drop-threshold approach (ICT(tol)) [81, 112] is purely numeric; only elements in \(^\sim L\) that satisfy a drop threshold condition defined in terms of tol, are retained. The drop-tolerance can be simply a non-negative value or non-negative factor such as the 1-norm. In addition to the numerical drop-tolerance, other constraints such as the number of nonzero entries allowed in every column [97] can be imposed. The major drawback of ICT is that the nonzero pattern cannot be determined before numeric factorization. However, a major strength lies in its ability to support a wide range of fill-in by suitable selections of the drop-threshold.

Despite the effectiveness of IC preconditioners, the existence of an IC factor is guaranteed only for a special classes of SPD matrices [77, 78]. This implies that IC factorization for an arbitrary SPD matrix may fail due to negative pivot elements, resulting in an indefinite preconditioner. Even such breakdowns does not occur, the incomplete factor can be unstable due to very small pivot elements. These bad pivot elements make the preconditioning less effective as shown by the analysis by Elman [36], and the empirical study by Chow and Saad [23].

The problem with pivot elements can be handled by adding a multiple of the identity matrix to the original matrix [77, 78], replacing bad pivots by an arbitrary positive value [81], and moving off-diagonal positive elements to diagonal elements [51]. In the unsymmetric case, pivoting can be applied to bring off-diagonal elements with a large magnitude to the diagonal [98].
2.3.4 Parallel Preconditioned Conjugate Gradients

Parallel CG is relatively easy to implement because each iteration comprises matrix-vector multiplication and vector dot-products operations. A simple parallel matrix-vector multiplication requires a single collective communication such as MPI \texttt{ALLTOALL} in MPI [48, 102]. The methods can be further tuned, using asynchronous communication [49, 105]. Efficient parallel iterative methods are found in many software packages [11, 104]. The main challenge is that of parallel preconditioning. The key challenge concerns effectively adapting conventional preconditioning schemes that have been successful on uniprocessors to the parallel multiprocessors. Issues such as reducing communication overheads and maximizing concurrency need to be addressed. Additionally, some approaches with good parallel efficiency may degrade the convergence properties, thus increasing the number of iterations. For example, incomplete factorization with a mechanism to support a wide range of fill (such as a drop-threshold scheme), which is considered the most general-purpose scheme on sequential machines, is not considered to be feasible on distributed memory multiple processors. The main reason is that applying such preconditioner requires parallel substitution, which is inefficient, and thus the various implementation techniques have been attempted for efficient parallel preconditioning [59, 60, 63, 69, 83, 99]. In recent years, this has lead to the development of sparse approximate inverse preconditioners which compute an approximation $M \approx A^{-1}$, directly. Hence, $M$ can be applied by a sparse matrix-vector multiplication, which is easy to parallelize. $M$ can be computed by several manners such as orthogonal projection [14, 65], and solving multiple least squares problems [19, 21, 50].

2.4 Hybrid Sparse Linear Solution

As shown earlier in this chapter, direct and iterative methods have different strengths and weaknesses. Direct methods would be the primary choice if the linear system is ill-conditioned and sufficient memory resources are available. On the other hand, if the system is well-conditioned, iterative methods can be more efficient in terms of both memory usage and computational time.

Incomplete Cholesky preconditioning is a popular and robust preconditioning scheme for SPD systems. Unlike other preconditioning schemes, an IC preconditioner is equivalent to a direct method if no dropping is performed, and its implementation can be tuned by several direct method techniques. We therefore attempt to develop a hybrid of direct and iterative methods by developing a highly efficient IC-preconditioned CG solver (ICCG). To construct a good hybrid solver, the amount of fill in $\hat{L}$ must be parameterized so that the method can handle a wide range of sparse problems, i.e., $\hat{L}$ can comprise no more nonzeros than $|A|$ or large fraction of those in $L$, depending on application needs. This can be naturally achieved by the drop-tolerance IC preconditioner. Earlier, Ng et al. implemented an efficient blocked ICT for uniprocessors, and showed that their approach allows efficient hybrids for a large variety of sparse problems [84]. In this dissertation, we focus on developing such hybrids for multiprocessor architectures.
Chapter 3

Parallel Incomplete Cholesky Preconditioners

In this chapter, we consider the development of an incomplete Cholesky factorization scheme for distributed memory multiprocessors as a basis for our general purpose direct-iterative hybrid solver. As discussed in Chapter 2, incomplete Cholesky factorization (IC) [24, 77, 78] is a general-purpose preconditioning scheme. Our design handles factorization using the supernodal elimination tree, a data structure that is used in the implementation of sparse direct solvers.

Section 3.1 describes the design of our parallel incomplete Cholesky factorization, where incomplete factors are constructed using a drop-threshold approach. Section 3.2 analytically investigates the performance of our factorization scheme for sparse matrices that arise from model two and three dimensional finite difference grids. Section 3.3 provides a brief summary of this chapter.

3.1 Left-Looking Drop-Threshold Incomplete Factorization

3.1.1 Overview

Drop-threshold incomplete Cholesky (ICT) [84, 112] is typically more robust than the its symbolic counterpart. The effectiveness of ICT essentially stems from the approximation expressed in numerical terms. Another advantage is that the amount of fill can be parameterized using a wide range of drop threshold conditions. At first glance, an implementation of ICT appears to be a small modification of the complete Cholesky factorization in Figure 3.1; it is almost the same as the complete Cholesky factorization except for the application of the drop tolerance condition. However, in practice, the incompleteness of the triangular factor $\hat{L}$ changes the nature of the ICT scheme. The primary reason is that the distribution of nonzero entries in the incomplete factor, $\hat{L}$, cannot be predicted before numeric factorization. This makes symbolic factorization less effective than when it is used for complete Cholesky factorization.

Figure 3.1 shows a left-looking ICT factorization. As described earlier in Chapter 2, sparse Cholesky factorization can be implemented using either left-looking, right-looking, or multifrontal schemes with different performance characteristics, and the same
holds for ICT factorization. The quality of ICT preconditioner is largely determined by the drop tolerance condition, and is often independent of techniques for efficient implementation. For this reason, the main focus of the parallel implementation lies in improving computational efficiency and saving memory overheads. According to the empirical study of complete Cholesky factorization by Ng and Peyton [85], both left-looking and multifrontal schemes achieve similar computational efficiency. However, the memory usage of the multifrontal scheme is higher than the left-looking scheme. This is due to an overhead from the stack storage required to accumulate rank-$k$ updates. When one considers the fact that preconditioned CG is typically used for large problems for which direct methods are not suitable due to large memory requirements, the left-looking approach is a natural choice for implementing ICT [84]. Our sequential ICT factorization scheme is shown in Figure 3.2. The factorization is guided by the supernodal elimination tree, which represents the data dependency of columns during complete factorization. The factorization uses a dense submatrix, $S$, associated with a supernode to improve cache-performance through data locality and data reuse via BLAS [28, 29]. In greater detail, the factorization at a supernode $i$ can be viewed as the computation with respect to the supernodal matrix $S_i$ where

$$S_i = \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}.$$ 

Now, $A_{11}$ ($A_{21}$) corresponds to a triangular (rectangular) part of matrix associated with the supernode. The factorization at the submatrix starts with left-looking update by
After the left-looking update is applied to $A_{11}$ and $A_{21}$ to yield $\hat{A}_{11}$ and $\hat{A}_{21}$, $L_{11}$ is computed by dense Cholesky factorization on $\hat{A}_{11}$. Then, $L_{21}$ is obtained by solving $L_{11}L_{21} = \hat{A}_{21}^T$. Finally, the drop-tolerance condition is applied to elements in $L_{11}$ and $L_{21}$ to obtain the final $\hat{L}_{11}$ and $\hat{L}_{21}$ portion of the incomplete factor as shown below:

$$
\begin{bmatrix}
\hat{A}_{11} \\
\hat{A}_{21}
\end{bmatrix} \rightarrow
\begin{bmatrix}
L_{11} \\
L_{21}
\end{bmatrix} \rightarrow
\begin{bmatrix}
\hat{L}_{11} \\
\hat{L}_{21}
\end{bmatrix}.
$$

The submatrices $\hat{L}_{11}$ and $\hat{L}_{21}$ are no longer dense after application of the drop tolerance condition. Hence, the data structure is in the form of sparse columns. This factor also affects the scheme for performing the left-looking update. In complete factorization, the left-looking update can be tuned by cache-efficient dense matrix data structures associated with earlier supernodes (called sup-sup update) instead of computing by a single column [85]. Our implementation still allows blocked (dense) structure for the supernode before dropping, but the earlier columns are not blocked. Therefore, the update can be applied to a block submatrix, but the source of the update is accessed as a sequence of sparse columns. This can be regarded as a col-sup update to utilize the cache in a limited fashion.

### 3.1.2 Distributed Asynchronous Left-looking Fan-in Scheme

Our parallel ICT is implemented with the aid of the supernodal elimination tree much like the complete Cholesky factorization. Using a subtree to subcube mapping [38], the tree is divided into local-phase subtrees at a certain level from the root. For instance, with a $P$ processor implementation on a binary-tree like supernodal elimination tree, the local phase subtrees are at the $\log_2 P$ level. Each of the subtrees is assigned to an individual processor, which then performs a sequential ICT factorization as described in Section 3.1.1 and Figure 3.2. Supernodes above the root of the local-phase subtree are assigned to multiple processors, containing descendant local phase subtrees. Thus, as shown in Figure 3.3, such supernodes comprise data parallel computation. More precisely, the supernodal matrix is partitioned in one-dimensional column block cyclic distribution. At each step, a column block is computed using updates by previous columns associated with lower levels of the tree. Such update is computed in each of the processors individually as shown in lines 5–8 and 20–23 in Figure 3.3. Every active processor checks its own local subtree, and the update is accumulated in its own buffer space. As indicated in lines 9 and 24 in Figure 3.3, once all updates in each processor are
1: Construct a supernodal tree $T$ using techniques from direct methods
2: Traverse the supernodal tree $T$ in post-order
3: for each supernode $i$ to the root do
   4: Allocate temporary space for the supernodal matrix $S_i$ at $i$
   5: for each column $\hat{L}_j$ in an update queue at $i$ do
      6: Perform the left-looking update with column $\hat{L}_j$ to $S_i$
      7: Put $\hat{L}_j$ into the update queue at $S_k$ if it will update $S_k$ next
   8: end for
9: Apply dense Cholesky factorization on the diagonal block $\hat{A}_{11}$ of $S_i$
10: Apply dense triangular solution to compute the off-diagonal block
11: for each off-diagonal element, $\alpha$ in $S_i$ do
12: if $\alpha < \text{tol}$ then
13: Drop $\alpha$ from $S_i$
14: end if
15: end for
16: Allocate sparse matrices $\hat{L}_{11}$ and $\hat{L}_{21}$ for $S_i$
17: Put each column into the update queue $S_k$ if it will update $S_k$ next
18: Copy the nonzero elements of $S_i$ to $\hat{L}_{11}$ and $\hat{L}_{21}$
19: Free the temporary space for $S_i$
20: end for

Fig. 3.2. Sequential scheme for drop-threshold IC factorization using the supernodal tree.
1: for each supernode \( i \) to the root do
2: Allocate buffer space \( B \)
3: for each block column \( s_j \) in \( S_i \) do
4: if \( s_j \) is assigned to me then
5: for each column \( \hat{L}_k \) in an update queue at \( S_i \) do
6: Perform the left-looking update with column \( \hat{L}_k \) to \( s_j \)
7: Put \( \hat{L}_k \) into the update queue at \( S_l \) (\( l > j \)) if it will update \( S_l \) next
8: end for
9: Receive and gather \( B \) from other processors
10: Compute block column \( s_j \)
11: for each off-diagonal element, \( \alpha \) in \( s_j \) do
12: if \( \alpha < \text{tol} \) then
13: Eliminate \( \alpha \) from \( s_j \)
14: end if
15: end for
16: Allocate a sparse matrix \( \hat{L}_j \) for column block \( j \)
17: Put each column \( j \) into the update queue \( S_l \) if it will update \( S_l \) next
18: Copy the nonzero elements of \( s_j \) to \( \hat{L}_j \)
19: else
20: for each column \( \hat{L}_k \) in an update queue at \( s_i \) do
21: Perform the left-looking update with column \( k \) and accumulate in \( B \)
22: Put \( \hat{L}_k \) into the update queue at \( S_l \) if it will update \( S_l \) next
23: end for
24: Asynchronously send the content of \( B \) to the the owner of \( s_j \)
25: end if
26: end for
27: release buffer space \( B \)
28: end for

Fig. 3.3. Parallel drop-threshold IC factorization using the supernodal tree.
completed, every processor except the owner of column block $s_j$ sends the result to the owner of $s_j$. This operation is performed by non-blocking asynchronous communication such as MPI\_ISend in MPI [48, 102]. Because the ISend operation is non-blocking, processors can start computing the updates for the next column block without waiting for the completion of the ISend. Therefore, these processors perform some useful work while one processor is engaged in gathering the updates and performing the incomplete factorization. The drop threshold condition is applied to each individual column block; the condition is the relative magnitude of the corresponding diagonal element. Thus, there is no interprocessor communication at this step. The placement of the computed columns into the queue for forthcoming updates is almost the same as in the sequential algorithm.

Fig. 3.4. Two levels of separators of an domain (graph) using nested dissection and the supernodal elimination tree.

3.1.3 Relationship to Domain- Decomposition and Multi-Level Linear Solvers

One major characteristic of our IC preconditioner is the use of the supernodal elimination tree. The tree is constructed after a fill-reducing ordering, such as nested dissection [40, 42] is applied to the coefficient matrix. Consider the matrix as the graph
of a domain as shown in Figure 3.4; the supernode at the root of the tree corresponds to
the separator, $S_1$, which partitions the graph into two subgraphs. The supernodes at the
second level of the tree correspond to the separators $S_2$ and $S_3$. This process is repeated
recursively until the subgraphs are of small size. The subtrees rooted at supernode,
$S_1$ in Figure 3.4, represent subdomains of the graph partitioned by separator $S_1$. This
observation shows that our ICT preconditioner is related (at a superficial level) to the
domain-decomposition solvers [101] based on partitioning domains recursively. With this
similarity, the Schur complement in the domain decomposition scheme is equivalent to
the diagonal block submatrix of a supernodal matrix after the left-looking update.

Another interesting aspect is that the Schur complement is created in every step
of the tree traversal; thus, the method is very similar to the multi-level linear solver [101,
107, 111]. In ICT, the incomplete factor is applied as a preconditioner using triangular
solution of the entire factor. Alternatively, the diagonal submatrix at each supernode
can be applied as a preconditioner to a selected subset of the elements of right-hand side
vector. In other words, our ICT can be viewed as a sequence of multiple linear solutions
of such subsystems. The solution of each subsystem can be improved using different
types of preconditioners [101, 107], and this hierarchical form could potentially improve
the entire solution.

3.2 Analysis of ICT Factorization Costs

The performance of our ICT factorization is perhaps best understood by starting
with the costs of computation and communication for the submatrix associated with a
single supernode. Recall that each supernode is associated with a supernodal matrix and
the diagonal submatrix is computed first and then applied to the off-diagonal block. This
off-diagonal block is reserved to compute rank-$k$ updates to construct the submatrices
associated with the supernodes at the higher levels of the supernodal tree, i.e. at ancestor
supernodes. Once the cost at a supernode is known, it can then be used in recursive
equations associated with the supernodal tree to determine the cost of the overall ICT
factorization. Our analysis is similar to the performance modeling of complete sparse
Cholesky factorization [41], in terms of computing the costs at supernodes. In this
section, we investigate the performance of our ICT factorization for sparse matrices that
arise from finite difference five-point $K \times K$ and seven-point $K \times K \times K$ grids for both
sequential and parallel implementations. We assume that the matrix is ordered by an
“optimal” nested dissection [40, 42, 71], and we ignore the cost for applying the drop
threshold condition.
3.2.1 Analysis of Sequential ICT Factorization

3.2.1.1 Computational Costs at a Supernode

The computational cost at a supernode is split into costs for the three different parts: (1) computing the diagonal block, (2) computing the off-diagonal block using the diagonal, and (3) rank-$k$ updates by the off-diagonal block to submatrices at ancestor supernodes. We first analytically derive the cost of each of the three steps, and use them to determine the total computational cost at a supernode. The following assertions on costs for basic operations can be obtained from standard textbooks [45].

- The cost of dense Cholesky factorization of an $n \times n$ matrix is $\frac{1}{3}n^3 + O(n^2)$ flops.
- The cost of dense triangular solution with an $n \times n$ triangular matrix is $n^2$ flops.
- The cost of a symmetric rank-1 update with a sparse column of $c$ nonzero elements is $c^2$ flops.

Consider the cost of rank-1 update with $n$ sparse columns where each column contains $c$ nonzero elements. The exact cost of such operation is $nc^2$, but it is an overestimate because some of the target elements will be dropped. To better approximate the reduction of costs from the application of a drop threshold condition, we use a factor of $\frac{1}{2}$ as an average case.

Next consider costs at an $(m+n) \times n$ supernodal matrix, where the matrix comprises an $n \times n$ diagonal submatrix and an $m \times n$ off-diagonal submatrix, is found by summing the cost of Cholesky factorization on the diagonal submatrix $A_{chol}(n)$, triangular solution of the diagonal submatrix with $m$ right-hand side vectors $A_{off}(m,n)$, and left-looking update using the $m \times n$ off-diagonal submatrix $A_{update}(m,n,\alpha)$. Assuming that every off-diagonal column has $\alpha$ nonzero elements, we have the costs of the three operations as:

$$A_{chol}(n) = \frac{1}{3}n^3 + O(n^2)$$
$$A_{off}(m,n) = mn^2$$
$$A_{update}(m,n,\alpha) = \frac{1}{2}n(\frac{\alpha}{m})^2 = \frac{1}{2}n\alpha^2.$$ 

Therefore, the total cost at the supernode $A_s(m,n,\alpha)$ is:

$$A_s(m,n,\alpha) = A_{chol}(n) + A_{off}(m,n) + A_{update}(m,n,\alpha)$$
$$= \frac{1}{3}n^3 + mn^2 + \frac{1}{2}n\alpha^2 + O(n^2).$$
3.2.1.2 Computational Costs: Model $K \times K$ Grids

We consider the computational cost for a matrix associated with a $K \times K$ grid ordered by nested dissection, where the grid is bordered along $i$ sides. Figure 3.5 shows the bordered sides for different subgrids created by separators and Figure 3.6 shows the four different cases that we will use in our analysis. We use recursive equations to aggregate the costs associated with the supernodal matrices that corresponds to a set of “+” shaped separators. Each “+” shaped separators are equivalent to supernodes at consecutive two levels of the supernodal tree; the sizes of the associated supernodal matrices are determined by the sizes of separators which borders related to each subgrid.

![Figure 3.5](image)

Fig. 3.5. The separators (sides) in a model 2-dimensional grid.

**Lemma 3.1.** Consider the $N \times N$ sparse matrix associated with the model five-point finite difference $K \times K$ grid ($N = K^2$). Assume an optimal nested dissection ordering is used for fill-reduction and $\alpha$ elements are retained for every $K$ elements in a column. Then, the computational cost of the sequential ICT factorization is no more than $\frac{11}{6}K^3 + O(K^2 \log_2 K)$.

**Proof.** Given a matrix associated with a $K \times K$ grid with $i$ bordered sides, let $A_{seq}(K, K, i, \alpha)$ denote the total computational cost of the sequential implementation of
our ICT factorization and let $M_{seq}(K, K, i, \alpha)$ be the computational cost of supernodal matrices associated with the “+” shaped separators. The cost of ICT factorization on a $K \times K$ grid is given by the following recursive equations.

$$A_{seq}(K, K, 0, \alpha) = 4A_{seq}(\frac{K}{2}, \frac{K}{2}, 2, \alpha) + M_{seq}(K, K, 0, \alpha) \quad (3.1)$$

$$A_{seq}(K, K, 2, \alpha) = A_{seq}(\frac{K}{2}, \frac{K}{2}, 2, \alpha) + 2A_{seq}(\frac{K}{2}, \frac{K}{2}, 3, \alpha) + A_{seq}(\frac{K}{2}, \frac{K}{2}, 4, \alpha) + M_{seq}(K, K, 2, \alpha) \quad (3.2)$$

$$A_{seq}(K, K, 3, \alpha) = 2A_{seq}(\frac{K}{2}, \frac{K}{2}, 3, \alpha) + 2A_{seq}(\frac{K}{2}, \frac{K}{2}, 4, \alpha) + M_{seq}(K, K, 3, \alpha) \quad (3.3)$$

$$A_{seq}(K, K, 4, \alpha) = 4A_{seq}(\frac{K}{2}, \frac{K}{2}, 4, \alpha) + M_{seq}(K, K, 4, \alpha) \quad (3.4)$$

In these recursive equations, $A_{seq}(K, K, 0, \alpha)$ represents the total ICT factorization cost of the sparse matrix of a $K \times K$ grid. The rest of this section considers expressions for $M_{seq}(K, K, 4, \alpha)$, which is then used to solve $A_{seq}(K, K, 4, \alpha)$.

The separators on a $K \times K$ grid bordered by 4 sides create three supernodal matrices: two $(3K + \frac{K}{2}) \times \frac{K}{2}$ matrices associated with the two small separators and a $(4K + K) \times K$ matrix associated with the large separator. The computational cost of a $(3K + \frac{K}{2}) \times \frac{K}{2}$ supernodal matrix, $M_{small}$, is obtained using the formula described in Section 3.2.1.1 (with new $\alpha$ defined in terms of $K$). Then, $M_{small}$ is:
\[ M_{\text{small}} = A_s(3K, \frac{K}{2}, \alpha) = A_{\text{chol}}(K/2) + A_{\text{off}}(3K, K/2) + A_{\text{update}}(3K, K/2, \alpha) = \frac{19}{24} K^3 + \frac{9}{4} \alpha^2 K + O(K^2) \]

Similarly, the computational cost of the large supernodal matrix, \( M_{\text{large}} \), is given by:

\[ M_{\text{large}} = A_s(4K, K, \alpha) = A_{\text{chol}}(4K) + A_{\text{off}}(4K, K) + A_{\text{update}}(4K, K, \alpha) = \frac{13}{3} K^3 + 8 \alpha^2 K + O(K^2) \]

The cost of computing all three supernodal matrices is therefore:

\[ M_{\text{seq}}(K, K, 4, \alpha) = 2M_{\text{small}} + M_{\text{large}} = \frac{71}{12} K^3 + \frac{25}{2} \alpha^2 K + O(K^2) \]

This result is applied to the recursive expression below:

\[ A_{\text{seq}}(K, K, 4, \alpha) = 4A_{\text{seq}}(K, K, 4, \alpha) + M_{\text{seq}}(K, K, 4, \alpha), \]

which is expanded to obtain:

\[ A_{\text{seq}}(K, K, 4, \alpha) = \left(\frac{71}{12} K^3 + \frac{25}{2} \alpha^2 K\right) + \left(\frac{71}{24} K^3 + \frac{25}{4} \alpha^2 K\right) + \cdots + \frac{1}{2 \log_2 K} \left(\frac{71}{12} K^3 + \frac{25}{2} \alpha^2 K\right) \]

This is essentially a geometric series:

\[ A_{\text{seq}}(K, K, 4, \alpha) = \left(\frac{71}{12} K^3 + \frac{25}{2} \alpha^2 K\right) + \sum_{j=0}^{\log_2 K-1} \left(\frac{1}{2}\right)^j + O(K^2 \log_2 K). \]

The equation thus simplifies to:

\[ A_{\text{seq}}(K, K, 4, \alpha) = \left(\frac{71}{6} K^3 + 25 \alpha^2 K\right)(1 - \frac{1}{2 \log_2 K}) = \left(\frac{71}{6} K^3 + 25 \alpha^2 K\right)(1 - \frac{1}{K}) + O(K^2 \log_2 K). \]

Hence, the total computational cost of ICT is (with \( \frac{1}{K} \to 0 \) for large \( K \)):

\[ A_{\text{seq}}(K, K, 4, \alpha) \approx \frac{71}{6} K^3 + 25 \alpha^2 K + O(K^2 \log_2 K). \]

\[ A_{\text{seq}}(K, K, i, \alpha) \] for other values of \( i \) can be found in the same manner. These results are only different in terms of constants associated with the \( K^3 \) and \( \alpha^2 K \) The value of \( M_{\text{seq}}(K, K, i, \alpha) \) is largest for \( i = 4 \); consequently \( A_{\text{seq}}(K, K, 4, \alpha) \) can be treated as an upper bound for \( A_{\text{seq}}(K, K, 0, \alpha) \). Thus, the computational cost of the ICT is no more than \( \frac{71}{6} K^3 + O(K^2 \log_2 K) \) for \( \alpha \ll K. \square \)

The higher-order cost of computing sparse Cholesky factorization for this problem (converting a multiply-add pair as two operations) is \( \frac{371}{12} K^3 \) (see analysis in [41]). Thus, the leading term of our ICT is smaller than that of Cholesky factorization.
3.2.1.3 Computational Costs: Model $K \times K \times K$ Grids

The analysis for sparse matrices associated with $K \times K \times K$ grids is similar to preceding the analysis for 2-dimensional grids. Figure 3.7 illustrates the three different separator planes that divide the original grid into eight cubic subgrids. Note that each subgrid is bordered by either 3, 4, 5 or 6 planes.

![Fig. 3.7. The 3 separator planes for the 3D cubic grid.](image)

**Lemma 3.2.** Consider the $N \times N$ sparse matrix associated with the model seven-point finite difference $K \times K \times K$ grid ($N = K^3$). Assume an optimal nested dissection ordering is used for fill-reduction and $\alpha$ elements are retained for every $K$ elements in a column. Then, the computational cost of the sequential ICT factorization is no more than $\frac{145}{14} K^6 + O(K^4)$.

**Proof.** Let $A_{seq}(K,K,i,\alpha)$ to denote the computational cost of the ICT factorization, and let $M_{seq}(K,K,i,\alpha)$ be the computational cost of supernodal matrices associated with the separators for a $K \times K \times K$ grid bordered by $i$ planes. Then, the costs of ICT factorization can be modeled by recursive equations as follows.
We now compute \( M_{seq}(K, K, 6, \alpha) \) which is the computational cost of 7 supernodal matrices associated with 3 separator planes on a \( K \times K \times K \) grid with 6 borders. The supernodal matrices associated with these separators are characterized by the 4 smallest separators (\( \frac{K}{2} \times \frac{K}{2} \) plane), 2 medium separators (\( K \times \frac{K}{2} \) plane) and the largest \( K \times K \) separator. The submatrices associated with the smallest separators consist of a \( \frac{K^2}{4} \times \frac{K^2}{4} \) diagonal block matrix and a \( \frac{5}{2}K^2 \times \frac{5}{4}K^2 \) off-diagonal block. As in Lemma 3.1, the drop condition is applied to retain \( \alpha \) nonzero elements for every \( K \) elements. The computational cost of this supernodal matrix, \( M_{small} \) is:

\[
M_{small} = A_s(\frac{5}{2}K^2, \frac{K^2}{4}, \alpha) = A_{chol}(\frac{K^2}{4}) + A_{off}(\frac{5K^2}{2}, \frac{K^2}{4}) + A_{update}(\frac{5K^2}{2}, \frac{K^2}{4}, \alpha)
\]

\[
= \frac{1}{3}(\frac{K^2}{4})^3 + \frac{5K^6}{32} + \frac{1}{2}K^6 + \frac{1}{2}(\frac{\alpha 5K^2}{2})^2 + O(K^4)
\]

\[
= \frac{31}{192}K^6 + \frac{25}{32}(\alpha^2 K^4) + O(K^4).
\]

The supernode associated with \( K \times \frac{K}{2} \) plane separator consists of a \( (4K^2 + \frac{K^2}{2}) \times \frac{K^2}{2} \) matrix. The computational cost, \( M_{medium} \), is given by:

\[
M_{medium} = A_s(4K^2, \frac{K^2}{4}, \alpha) = A_{chol}(\frac{K^2}{2}) + A_{off}(4K^2, \frac{K^2}{2}) + A_{update}(4K^2, \frac{K^2}{2}, \alpha)
\]

\[
= \frac{1}{3}(\frac{K^2}{2})^3 + \frac{1}{2}K^6 + \frac{1}{2}(\frac{\alpha 4K^2}{2})^2 + O(K^4)
\]

\[
= \frac{25}{24}K^6 + 4\alpha^2 K^4 + O(K^4).
\]
The supernodal matrix associated with \( K \times K \) plane separator has a \((6K^2 + K^2) \times K^2\) matrix. Thus, \( M_{\text{large}} \) is:

\[
M_{\text{large}} = A_s(5K^2, K^2, \alpha) = A_{\text{chol}}(K^2) + A_{\text{off}}(6K^2, K^2) + A_{\text{update}}(6K^2, K^2, \alpha)
\]

\[
= \frac{1}{3}(K^2)^3 + 6K^6 + \frac{1}{2}(K^2)(6\alpha K^2)^2 + O(K^4)
\]

\[
= \frac{19}{3}(K^6) + 18\alpha^2 K^4 + O(K^4).
\]

Now the cost of computing all supernodal matrices for the three different separators is:

\[
M_{\text{seq}}(K, K, 6, \alpha) = 4M_{\text{small}} + 2M_{\text{medium}} + M_{\text{large}} = \frac{145}{16}K^6 + \frac{233}{8}\alpha^2 K^4 + O(K^4).
\]

We now compute \( A_{\text{seq}}(K, K, 6, \alpha) \), applying \( M_{\text{seq}}(K, K, 6, \alpha) \) to Equation 3.9 to yield a geometric series:

\[
A_{\text{seq}}(K, K, 6, \alpha) = (\frac{145}{16}K^6 + \frac{233}{8}\alpha^2 K^4) + \cdots + (\frac{1}{8})^{\log_2 K} (\frac{145}{128}K^6 + \frac{233}{64}\alpha^2 K^4) + O(K^4)
\]

\[
= (\frac{145}{16}K^6 + \frac{233}{8}\alpha^2 K^4) \sum_{j=0}^{\log_2 K-1} \left(\frac{1}{8}\right)^j + O(K^4 \log_2 K).
\]

The solution is:

\[
A_{\text{seq}}(K, K, 6, \alpha) = (\frac{145}{14}K^6 + \frac{233}{7}\alpha^2 K^4)(1 - (\frac{1}{8})^{\log_2 K}) + O(K^4 \log_2 K)
\]

\[
= (\frac{145}{14}K^6 + \frac{233}{7}\alpha^2 K^4)(1 - (\frac{1}{K})^3) + O(K^4 \log_2 K).
\]

For a large \( K \), we assume that \((\frac{1}{K})^3\) is negligible to simplify the expression for \( A_{\text{seq}}(K, K, 6, \alpha) \) as follows:

\[
A_{\text{seq}}(K, K, 6, \alpha) = \frac{145}{14}K^6 + \frac{233}{7}\alpha^2 K^4 + O(K^4).
\]

Once again, \( A_{\text{seq}}(K, K, 6, \alpha) \) can be regarded as an upper bound because \( M_{\text{seq}}(K, K, i, \alpha) \) is the largest for \( i = 6 \). Therefore, the computational cost of the sequential ICT is no more than \( \frac{145}{14}K^6 + O(K^4) \) for \( \alpha \ll K \).

The higher-order cost of computing sparse Cholesky factorization for this problem (converting a multiply-add pair as two operations) is \( \frac{1077}{14}K^6 \) (see analysis in [41]). Thus, the leading term of our ICT is far smaller than that of Cholesky factorization.

3.2.2 Analysis of Parallel ICT Factorization

Our parallel ICT factorization schemes include communication costs in addition to computational costs. We analyze these two costs separately. We first discuss these costs at a single supernode, and apply them to determine the costs for entire scheme for matrices from model \( K \times K \) and \( K \times K \times K \) finite difference grids.
3.2.2.1 Computational Costs at a Supernode

The computational cost of our parallel ICT at a supernode using \( p \) processors is somewhat similar to the analysis for the sequential implementation described in Section 3.2.1.1. The main difference is that the cost is typically scaled by the number of processors \( p \). Thus, the cost of basic operations for an \((m + n) \times n\) supernodal matrix are given by:

\[
A_{\text{chol}}(n, p) = \frac{1}{3p} n^3 + O\left(\frac{n^2}{p}\right)
\]

\[
A_{\text{off}}(m, n, p) = \frac{1}{p} mn^2
\]

\[
A_{\text{update}}(m, n, \alpha, p) = \frac{1}{2p} n \left(\frac{\alpha}{m}\right)^2 = \frac{1}{2p} n \alpha^2.
\]

The total cost at the supernode is therefore:

\[
A_s(m, n, \alpha, p) = \frac{1}{3p} n^3 + \frac{1}{p} mn^2 + \frac{1}{2p} n \alpha^2 + O\left(\frac{n^3}{p}\right).
\]

3.2.2.2 Communication Costs a Supernode

We consider the communication costs of basic operations at an \((m + n) \times n\) supernodal matrix assigned to \( p \) processors using our ICT which employs a parallel fan-in factorization algorithm as described in Figure 3.3. The fan-in scheme computes both diagonal and off-diagonal submatrices by column blocks at the same time. The communication with respect to a column block occurs once between the owner of the column block and other processors as depicted in lines 9 and 24 of Figure 3.3. The number of messages propagated in each processor is, thus, no less than the number of column blocks in the supernode. In addition, the fact that each message contains the result of updates for a single column block \( s_j \) implies that the volume of all messages is no less than the size of the entire supernode. Hence, the communication cost of each processor at the supernode is bounded by \( nt_s \) for the latency and by \((\frac{1}{2}n^2 + mn)t_w\) for the communication volume. The total communication cost is given therefore by:

\[
C_s(m, n, \alpha, p) = nt_s + (\frac{1}{2}n^2 + mn)t_w.
\]

3.2.2.3 Computational Costs: Model \( K \times K \) Grids

**Lemma 3.3.** Consider the \( N \times N \) sparse matrix associated with the model seven point finite difference \( K \times K \) grid \((N = K^2)\). Assume an optimal nested dissection ordering is used for fill-reduction and \( \alpha \) elements are retained for every \( K \) elements in a column. Then, the computational cost of the ICT factorization using \( P \) processors is no more than \( \frac{71}{6P} K^3 + O\left(\frac{K^2}{P} \log_2 K\right) \).
In incomplete factorization, these matrices will not be dense due to drops.

Fig. 3.8. Supernodes and the local-phase subtree (colored) computed by processor 0 in a 4 processor implementation of parallel ICT.
Proof. The computational costs for parallel ICT factorization are derived using recursive equations similar to those in Lemma 3.1 with the difference that now supernodes are assigned to multiple processors until the $\log_2 P$ level of the supernodal tree. In other words, the number of processors assigned to each supernode is halved at each level until a local-phase subtree is assigned to a single processor. The computational costs for the entire grid is recursive costs in terms of the costs for supernodes associated with the “+” shaped separators. However, a key difference is that a single processor only computes supernodes in a path from the root of the local-phase tree to the root of the entire supernodal tree. This is illustrated in Figure 3.8 for a 4 processor mapping. Once this cost model for supernodes, henceforth $M_{\text{dist}}(K, K, i, \alpha, p)$ is defined, the entire factorization cost can be found using recursive equations. Consider the cost at a “+” shaped separator involving costs for a large and small matrix. For $i = 4$, there is a $(3K + \frac{K}{2}) \times \frac{K}{2}$ distributed on $\frac{p}{2}$ processors and a $(4K + K) \times K$ matrices distributed on $p$ processors. The corresponding computational costs $M_{\text{small}}$ and $M_{\text{large}}$ are given as follows:

$$M_{\text{small}} = A_s(3K, \frac{K}{2}, \alpha, \frac{p}{2}) = \frac{19}{12p}K^3 + \frac{9}{2p}K\alpha^2 + O\left(\frac{K^2}{p}\right)$$

$$M_{\text{large}} = A_s(4K, K, \alpha, p) = \frac{13}{3p}K^3 + \frac{8}{p}\alpha^2 K + O\left(\frac{K^2}{p}\right).$$

As a result, the cost at a single processor with respect to the “+” shaped separators is:

$$M_{\text{dist}}(K, K, 4, \alpha, p) = M_{\text{small}} + M_{\text{large}} = \frac{71}{12p}K^3 + \frac{25}{2p}K\alpha^2 + O\left(\frac{K^2}{p}\right).$$

The recursive equations for the cost of parallel ICT factorization, $A_{\text{dist}}(K, K, i, \alpha, p)$, using $M_{\text{dist}}(K, K, i, \alpha, p)$ are as follows.

$$A_{\text{dist}}(K, K, 0, \alpha, P) = A_{\text{dist}}(\frac{K}{2}, \frac{K}{2}, 2, \alpha, P/4) + M_{\text{dist}}(K, K, 0, \alpha, P)$$ (3.10)

$$A_{\text{dist}}(K, K, 2, \alpha, P) = \frac{1}{4}A_{\text{dist}}(\frac{K}{2}, \frac{K}{2}, 2, \alpha, P/4) + \frac{1}{2}A_{\text{dist}}(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/4) +$$

$$+ \frac{1}{4}A_{\text{dist}}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) + M_{\text{dist}}(K, K, 2, \alpha, P)$$ (3.11)

$$A_{\text{dist}}(K, K, 3, \alpha, P) = \frac{1}{2}A_{\text{dist}}(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/4) + \frac{1}{2}A_{\text{dist}}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) +$$

$$+ M_{\text{dist}}(K, K, 3, \alpha, P)$$ (3.12)

$$A_{\text{dist}}(K, K, 4, \alpha, P) = A_{\text{dist}}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) + M_{\text{dist}}(K, K, 4, \alpha, P)$$ (3.13)

These recursive equations can be expressed as non-recursive equations in $\log_2 P = \log_4 P$ terms of $M_{\text{dist}}$ and the term for computing the local-phase subtrees, $A_{\text{seq}}$. For $i = 4$, the non-recursive form is given by:
\[ A_{\text{dist}}(K, K, 4, \alpha, P) = M_{\text{dist}}(K, K, 4, \alpha, P) + M_{\text{dist}}(K/2, K/2, 4, \alpha, P/4) + \]
\[ \cdots + M_{\text{dist}}(K/(2^\log_4 P - 1), K/(2^\log_4 P - 1), 4, \alpha, 4) + O\left(\frac{K^2}{P} \log_4 P\right) + \]
\[ A_{\text{seq}}(K/2^\log_4 P, K/2^\log_4 P, 4, \alpha) \]

Thus, sequential ICT factorization is performed on a local-phase subtree associated with a \(K/2^\log_4 P \times K/2^\log_4 P\) subgrid. We now apply the geometric series for the terms of \(M_{\text{dist}}\) to obtain:

\[ A_{\text{dist}}(K; K; 4; P) = \left(\frac{71}{12P}K^3 + \frac{25}{2P} \alpha^2 K\right) \sum_{j=0}^{\log_4 P - 1} \left(\frac{1}{2}\right)^j + O\left(\frac{K^2}{P} \log_4 P\right) \]

\[ A_{\text{seq}}(K/2^\log_4 P, K/2^\log_4 P, 4, \alpha) \]

\[ = \left(\frac{71}{6P}K^3 + \frac{25}{P} \alpha^2 K\right)\left(1 - \left(\frac{1}{2}\right)^{\log_4 P}\right) + O\left(\frac{K^2}{P} \log_4 P\right) \]

\[ \frac{71}{6P^{3/2}}K^3 + \frac{25}{P^{3/2}} \alpha^2 K + O\left(\frac{K^2}{P} (\log_2 K - \log_4 P)\right) \]

The equation above can be simplified to yield:

\[ A_{\text{dist}}(K, K, 4, \alpha, P) = \left(\frac{71}{6P}K^3 + \frac{25}{P} \alpha^2 K\right)\left(1 - \frac{1}{\sqrt{P}}\right) + \]

\[ \frac{71}{6P^{3/2}}K^3 + \frac{25}{P^{3/2}} \alpha^2 K + O\left(\frac{K^2}{P} \log_2 K\right) \]

\[ \approx \frac{71}{6P}K^3 + \frac{25}{P} \alpha^2 K + O\left(\frac{K^2}{P} \log_2 K\right) \]

Since \(A_{\text{dist}}(K, K, i, \alpha, P)\) is the largest for \(i = 4\), the computational cost is no more than \(\frac{71}{6P}K^3 + O\left(\frac{K^2}{P} \log_2 K\right)\). □

### 3.2.2.4 Communication Costs: Model \(K \times K\) Grids

**Lemma 3.4.** Consider the \(N \times N\) sparse matrix associated with the model five-point finite difference \(K \times K\) grid \((N = K^2)\). Assume an optimal nested dissection ordering is used for fill-reduction and \(P\) processors are used. Then, the number of messages sent during ICT factorization is no more than \(3K\) and associated volume is no more than \(\frac{49}{6}K^2\).

**Proof.** Let \(C(K, K, i, \alpha, P)\) be the communication costs of parallel ICT on a matrix associated with a \(K \times K\) grid bordered along \(i\) sides with \(\alpha\) nonzero elements for every
$K$ elements. The corresponding recursive equation is constructed using $Q(K, K, i, \alpha, p)$, the communication cost of supernodes associated with the “+$” shaped separators on a $K \times K$ grid with $p$ processors.

Using the expression in Section 3.2.2.2, the communication costs for the two supernodal matrices associated with the “+$” shaped separators are given by:

\[
Q_{\text{small}} = C_s(3K, \frac{K}{2}, p) = \frac{1}{2}Kt_s + (\frac{1}{8}K^2 + \frac{3}{2}K^2)t_w = \frac{1}{2}Kt_s + \frac{13}{8}K^2t_w
\]

\[
Q_{\text{large}} = C_s(4K, K, p) = Kt_s + (\frac{1}{2}K^2 + 4K^2)t_w = Kt_s + \frac{9}{2}K^2t_w
\]

\[
Q(K, K, 4, \alpha, p) = Q_{\text{small}} + Q_{\text{large}} = \frac{3}{2}Kt_s + \frac{49}{8}K^2t_w.
\]

$C(K, K, i, \alpha, P)$ is then defined as follows.

\[
C(K, K, 0, \alpha, P) = C(\frac{K}{2}, \frac{K}{2}, 2, \alpha, P/4) + Q(K, K, 0, \alpha, P)
\]

\[
C(K, K, 2, \alpha, P) = \frac{1}{4}C(\frac{K}{2}, \frac{K}{2}, 2, \alpha, P/4) + \frac{1}{2}C(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/4) + \frac{1}{4}C(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) + Q(K, K, 2, \alpha, P)
\]

\[
C(K, K, 3, \alpha, P) = \frac{1}{2}C(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/4) + \frac{1}{2}C(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) + Q(K, K, 3, \alpha, P)
\]

\[
C(K, K, 4, \alpha, P) = C(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) + Q(K, K, 4, \alpha, P)
\]

Consider expanding $C(K, K, 4, \alpha, P)$ to its non-recursive form as:

\[
C(K, K, 4, \alpha, P) = Q(K, K, 4, \alpha, P) + Q(K/2, K/2, 4, \alpha, P/4) + \cdots + Q(K/(2^{\log_4 P - 1}), K/(2^{\log_4 P - 1}), 4, \alpha, 4).
\]

This simplified to a geometric series as:

\[
C(K, K, 4, \alpha, P) = \frac{3}{2}K \sum_{j=0}^{\log_4 P - 1} \left(\frac{1}{2}\right)^j t_s + \frac{49}{8}K^2 \sum_{j=0}^{\log_4 P - 1} \left(\frac{1}{4}\right)^j t_w.
\]

which can be solved to yield:

\[
C(K, K, 4, \alpha, P) = 3K(1 - \frac{1}{\sqrt{P}})t_s + \frac{49}{6}K^2(1 - \frac{1}{P})t_w.
\]

For large $P$, we can ignore $\frac{1}{\sqrt{P}}$ terms and $\frac{1}{P}$ terms to obtain:

\[
C(K, K, 4, \alpha, P) = 3Kt_s + \frac{49}{6}K^2t_w. \square
\]
3.2.2.5 Computational Costs: Model $K \times K \times K$ Grids

Lemma 3.5. Consider the $N \times N$ sparse matrix associated with the model seven-point finite difference $K \times K \times K$ grid ($N = K^3$). Assume an optimal nested dissection ordering is used for fill-reduction and $\alpha$ elements are retained for every $K$ elements in a column. Then, the computational cost of the parallel ICT factorization using $P$ processors is no more than $\frac{145}{14P} + O(K^4)$.

Proof. The costs model for a grid bordered by 6 planes, $M_{\text{dist}}(K, K, 6, \alpha, P)$, includes the costs associated with three different supernodal matrices assigned to $P/4$, $P/2$, and $P$ processors. The computational costs of these three matrices ($M_{\text{small}}$, $M_{\text{medium}}$, and $M_{\text{large}}$) are specified below:

$$M_{\text{small}} = A_s\left(\frac{5}{2}K^2, \frac{K^2}{4}, \alpha, \frac{p}{4}\right) = \frac{31}{48p} K^6 + \frac{25}{8p} \alpha^2 K^4$$

$$M_{\text{medium}} = A_s(4K^2, \frac{K^2}{2}, \alpha, \frac{p}{2}) = \frac{25}{12p} K^6 + \frac{8}{p} \alpha^2 K^4$$

$$M_{\text{large}} = A_s(6K^2, K^2, \alpha, p) = \frac{19}{3p} K^6 + \frac{18}{p} \alpha^2 K^4$$

Using these expressions, we obtain:

$$M_{\text{dist}}(K, K, 6, \alpha, p) = \frac{145}{16p} K^6 + \frac{233}{8p} \alpha^2 K^4 + O\left(\frac{K^4}{p}\right).$$

The recursive equations for the parallel ICT factorization are:

$$A_{\text{dist}}(K, K, 0, \alpha, P) = A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 2, \alpha, P/8\right) + M_{\text{dist}}(K, K, 0, \alpha, P)$$ (3.18)

$$A_{\text{dist}}(K, K, 3, \alpha, P) = \frac{1}{8} A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 3, \alpha, P/8\right) + \frac{3}{8} A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 4, \alpha, P/8\right)$$

$$+ \frac{3}{8} A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 5, \alpha, P/8\right) \cdot \frac{1}{8} A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8\right) + M_{\text{dist}}(K, K, 3, \alpha, P)$$ (3.19)

$$A_{\text{dist}}(K, K, 4, \alpha, P) = \frac{1}{4} A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 4, \alpha, P/8\right) + \frac{1}{2} A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 5, \alpha, P/8\right)$$

$$+ \frac{1}{4} A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8\right) + M_{\text{dist}}(K, K, 4, \alpha, P)$$ (3.20)

$$A_{\text{dist}}(K, K, 5, \alpha, P) = \frac{1}{2} A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 5, \alpha, P/8\right) + \frac{1}{2} A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8\right)$$

$$+ M_{\text{dist}}(K, K, 5, \alpha, P)$$ (3.21)

$$A_{\text{dist}}(K, K, 6, \alpha, P) = A_{\text{dist}}\left(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8\right) + M_{\text{dist}}(K, K, 6, \alpha, P).$$ (3.22)
These recursive equations can be expanded to their non-recursive forms using $M_{dist}$ and the cost for the local-phase subtree. $A_{dist}(K, K, 6, \alpha, P)$ is now expressed as follows:

$$A_{dist}(K, K, 6, \alpha, P) = M_{dist}(K, K, 6, \alpha, P) + M_{dist}(K/2, K/2, 6, \alpha, P/8) + \cdots + M_{dist}(K/2^{\log_2 P-1}, K/2^{\log_2 P-1}, 6, \alpha, 8) + A_{seq}(K/2^{\log_2 P}, K/2^{\log_2 P}, K/2^{\log_2 P}, 6, \alpha)$$

This form can be simplified to yield:

$$A_{dist}(K, K, 6, \alpha, P) = \left(\frac{145}{14P}K^6 + \frac{233}{7P} \alpha^2 K^4\right) + O\left(\frac{K^4}{P}\right) + A_{seq}(K/2^{\log_2 P}, K/2^{\log_2 P}, K/2^{\log_2 P}, 6, \alpha)$$

For $\alpha \ll K$, the final expression is:

$$A_{dist}(K, K, 6, \alpha, P) = \frac{145}{14P}K^6 + \frac{233}{7P} \alpha^2 K^4 + O\left(\frac{K^4}{P}\right)$$

Observe that the costs are the same as $A_{seq}(K, K, 6, \alpha, P)/P$.

### 3.2.2.6 Communication Costs: Model $K \times K \times K$ Grids

**Lemma 3.6.** Consider the $N \times N$ sparse matrix associated with the model seven-point finite difference $K \times K \times K$ grid ($N = K^3$). Assume an optimal nested dissection ordering is used for fill-reduction and that $P$ processors are used. Then, the number of messages sent by a single processor is no more than $\frac{7}{3}K^2$ and the corresponding data volume is no more than $\frac{317}{30}K^2$.

**Proof.** Let $C(K, K, i, \alpha, P)$ be the communication cost of computing parallel ICT factorization of a matrix associated with a $K \times K \times K$ grid with $i$ borders using $P$ processors. The recursive equation is constructed in terms of $Q(K, K, i, \alpha, P)$, the communication cost of supernodes associated with three separator planes on $K \times K \times K$ grid. For $i = 6$, $Q(K, K, 6, \alpha, P)$ is given by:
\[ Q_{small} = C_s(\frac{5}{2}K^2, \frac{K^2}{4}, \alpha, \frac{P}{4}) = \frac{K^2}{4}t_s + \frac{21}{32}K^2t_w \]

\[ Q_{medium} = C_s(4K^2, \frac{K^2}{2}, \alpha, \frac{P}{2}) = \frac{K^2}{2}t_s + \frac{17}{8}K^4t_w \]

\[ Q_{large} = C_s(6K^2, K^2, \alpha, P) = K^2t_s + \frac{13}{2}K^4t_w \]

\[ Q(K, K, 6, \alpha, P) = Q_{small} + G_{medium} + Q_{large} = \frac{7}{4}K^2t_s + \frac{317}{32}K^4t_w. \]

\[ Q(K, K, i, \alpha, P) \] for other values of \( i \) can be established similarly. Now the total communication costs for the entire factorization is specified by the recursive equations, \( C(K, K, i, \alpha, P) \), below:

\[ C(K, K, 0, \alpha, P) = C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 3, \alpha, P/8) + Q(K, K, 0, \alpha, P) \quad (3.23) \]

\[ C(K, K, 3, \alpha, P) = \frac{1}{8}C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 3, \alpha, P/8) \]
\[ + \frac{3}{8}C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 4, \alpha, P/8) + \frac{3}{8}C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 5, \alpha, P/8) \]
\[ + \frac{1}{8}C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + Q(K, K, 3, \alpha, P) \quad (3.24) \]

\[ C(K, K, 4, \alpha, P) = \frac{1}{4}C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 4, \alpha, P/8) + \frac{1}{2}C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 5, \alpha, P/8) \]
\[ + \frac{1}{4}C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + Q(K, K, 4, \alpha, P) \quad (3.25) \]

\[ C(K, K, 5, \alpha, P) = \frac{1}{2}C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + \frac{1}{2}C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) \]
\[ + Q(K, K, 5, \alpha, P) \quad (3.26) \]

\[ C(K, K, 6, \alpha, P) = C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + Q(K, K, 6, \alpha, P). \quad (3.27) \]

For \( i = 6 \), \( C(K, K, i, \alpha, P) \) is expressed using a geometric series as:

\[ C(K, K, 6, \alpha, P) = Q(K, K, 6, \alpha, P) + Q(K/2, K/2, K/2, 6, \alpha, P/8) \]
\[ + \cdots + Q(K/2^{\log_8 P}, K/2^{\log_8 P}, K/2^{\log_8 P}, 6, \alpha, 8) \]
\[ = \frac{7}{4}K^2 \log_8 P^{-1} \sum_{j=0}^{P-1} \left( \frac{1}{4} \right)^j + \frac{317}{32}K^4 \log_8 P^{-1} \sum_{j=0}^{P-1} \left( \frac{1}{16} \right)^j. \]

This is equivalent to:

\[ C(K, K, 6, \alpha, P) = \frac{7}{3}K^2(1 - \left( \frac{1}{P} \right)^{2/3})t_s + \frac{317}{30}K^4(1 - \left( \frac{1}{P} \right)^{4/3})t_w. \]
For large $P$, $(\frac{1}{P})^{2/3}$ terms and $(\frac{1}{P})^{4/3}$ terms can be ignored to yield:

$$C(K, K, 6, \alpha, P) = \frac{7}{3}K^2 t_s + \frac{317}{30}K^4 t_w.$$  

This expression is a somewhat loose upper bound for a $K \times K \times K$ grid because the constants associated with $t_w$ are smaller for grids with fewer borders than 6. □

Our analysis also indicates that the communication costs grow faster on 3-dimensional problems than on 2-dimensional problems. Both the latency and the volume of communication are independent of the number of processors.

### 3.3 Summary

We have developed a parallel ICT factorization scheme that can form the first stage of a general purpose hybrid linear solver. The factorization is derived from extensions of several techniques used in sparse direct solvers. We provide an analysis of the costs of computation and interprocessor communication for sparse matrices associated with regular $K \times K$ and $K \times K \times K$ grids. Our analysis indicates that ICT is computationally more efficient than (complete) sparse Cholesky in terms of lower constants associated with the dominant terms. Savings in arithmetic operations are greater for $K \times K \times K$ grids than for $K \times K$ grids.
Chapter 4

Modifications to Enable Latency Tolerant Application of ICT Preconditioners

4.1 Introduction

This chapter discusses how extra computations can be utilized to enable a latency-tolerant application of ICT preconditioners on distributed memory multiprocessors. In the previous chapter, we explained our parallel ICT factorization using techniques derived from those for parallel sparse direct solvers. To apply such a preconditioner at each CG iteration, a natural implementation is to use a parallel sparse triangular solution based on a traversal of the supernodal tree with a parallel substitution scheme at each supernode assigned to multiple processors. However, with such a conventional substitution scheme, triangular solution is not efficient because the substitution algorithm is intrinsically sequential and it additionally suffers from the large latency of interprocessor communication. A natural scheme to alleviate this problem is to replace distributed substitution by distributed matrix-vector multiplication which can be executed at near machine peak speeds; we discuss such schemes for preconditioner application in Chapter 7. This chapter concerns the computations required to adapt our parallel ICT factors to enable the use of such latency-tolerant schemes for preconditioner application. Section 4.2 describes how selected triangular submatrices can be explicitly inverted. Section 4.3 proposes a variant when inversion is performed using sparse approximate inverse techniques. Finally, Section 4.4 and 4.5 contain an analysis of the costs of the ICT factorization with these post-processing steps for model grid problems.

4.2 Parallel ICT Using Selective Inversion (ICT-SI)

Distributed substitution that can be a performance bottleneck is required to solve subsystem associated with triangular portions of each supernodal matrix distributed to more than one processor. As explained in Chapter 3, our ICT factorization at a supernode concerns a pair of block submatrices shown below:

\[
\begin{bmatrix}
A_{11} \\
A_{21}
\end{bmatrix}
\]
In this form, $A_{11}$ is triangular and $A_{21}$ is rectangular. After factorization, we have 

$$\begin{bmatrix}
L_{11} \\
L_{21}
\end{bmatrix}.$$ 

Now the part that corresponds to $\hat{L}_{22}$ obtained by dropping elements from $L_{11}$ is used in distributed substitution. Selective Inversion (SI) [92] concerns replacing $\hat{L}_{11}$ by its inverse. To include this aspect, our algorithm performs a parallel explicit matrix inversion to compute $L_{11}^{-1}$ instead of applying a drop threshold condition to $L_{11}$ and $L_{21}$ immediately. Then, the drop threshold condition is applied to $L_{11}^{-1}$ and $L_{21}$ to filter the off-diagonal elements of $L_{11}^{-1}$ and $L_{21}$. Finally, the algorithm obtains the supernodal matrix of the form:

$$\begin{bmatrix}
\hat{L}_{11}^{-1} \\
\hat{L}_{21}
\end{bmatrix}.$$ 

The details of our scheme at a supernode are described in Figure 4.1. The inversion incurs an extra computation for the numeric factorization, but, in Chapter 7, we will show that the improvements in applying the preconditioner are substantial [94, 103].

### 4.3 Parallel ICT Using Selective Sparse Approximate Inversion (ICT-SSAI)

Although the Selective Inversion (SI) technique described above can achieve substantial performance improvements for subsequent application of the preconditioner, the explicit matrix inversion makes it difficult to achieve a scalable scheme for preconditioner construction. More specifically, the method suffers from high computational and communication costs on matrices that arise from discretization of the three dimensional domains (with large supernodal matrices at the top levels of the supernodal tree). The inefficiency is primarily due to parallel dense matrix computation to perform Cholesky factorization followed by explicit matrix inversion prior to dropping elements.

As an alternative to explicit matrix inversion in SI, we now propose the use of sparse approximate inverse techniques [14, 19, 50, 65] to reduce both computational and communication costs. As a result, the new preconditioner can be viewed as an incomplete factorization applied to the global matrix, within which sparse approximate inverse preconditioning is applied to some selected triangular submatrices.

#### 4.3.1 Sparse Approximate Inverse Preconditioners

Sparse approximate inverses (SAI) [14, 19, 50, 65] emerged in the last decade as an alternative to parallel preconditioning with incomplete matrix factors, mainly to
for each supernode \( i \) to the root do

Allocate buffer space \( B \)

for each block column \( s_j \) in \( S_i \) do

if \( s_j \) is assigned to me then

for each column \( L_k \) in an update queue at \( S_i \) do

Perform the left-looking update with column \( L_k \) to \( s_j \)

Put \( L_k \) into the update queue at \( S_l \) (\( l > j \)) if it will update \( S_l \) next

end for

Receive and gather \( B \) from other processors

Apply dense Cholesky factorization to \( s_j \)

else

for each column \( L_k \) in an update queue at \( s_i \) do

Perform the left-looking update with column \( k \) and accumulate in \( B \)

Put \( L_k \) into the update queue at \( S_l \) if it will update \( S_l \) next

end for

Asynchronously send the content of \( B \) to the owner of \( s_j \)

end if

end for

Perform parallel matrix inversion for dense distributed matrix \( L_i \)

for each local block column \( s_j \) in \( L_i \) do

for each element \( \alpha \) in \( s_j \) do

if \( \alpha < \text{tol} \) then

Drop \( \alpha \) from \( L_i \)

end if

end for

Allocate a sparse columns \( L_j \)

Put each column \( j \) into the update queue \( S_l \) if it will update \( S_l \) next

Copy the nonzero elements of \( s_j \) to \( L_j \)

end for

release buffer space \( B \)

end for

Fig. 4.1. Parallel ICT factorization with Selective Inversion (ICT-SI).
overcome the inefficiencies in applying the latter. The efficiency of SAI lies in using parallel sparse matrix-vector multiplication in its application, which parallelizes effectively compared to parallel substitution. Observe that these methods directly compute approximate inverses of $A$; we are now reviewing these techniques prior to describing techniques that use SAI with ICT to selectively invert portions of our incomplete Cholesky preconditioner. The methods of sparse approximate inverses fall into 2 classes: (i) Frobenius norm minimization and (ii) incomplete biconjugation. The former [21, 50, 65] aims at minimizing $\| I - AM \|_F$ where $A$ is the coefficient matrix, and $M$ is preconditioner matrix. This problem can be reduced to the solution of multiple least-square problems to solve that:

$$\min \| I - AM \|_F^2 \approx \sum_{j=1}^{n} \min \| e_j - Am_j \|_2^2.$$ 

In the expression above, $e_j$ is canonical vector and $m_j$ is $j$-th column of $M$. If $A$ is Symmetric and Positive Definite (SPD), the method tries to minimize $\| I - LG \|_F$, where $L$ is a Cholesky factor of $A$ and $G$ is a lower triangular preconditioner matrix. Since each least-squares solution can be computed independently, the method is suitable for the parallel implementation as shown by Grote and Huckle [50], and Chow [21]. The drawback of this method is that the solution of least-squares problems in every row/column tends to converge to local minimizer, and that the preconditioner may not be robust for matrices with very large dimension (as noted by Chow [21]). Another problem is that the method requires users to provide the distribution of nonzero elements of $M$ before sparse approximate inverse inversion. In particular, this problem is handled using the nonzero structure of $A^T$ ($i = 1, 2, 3\ldots$) to achieve an efficient preconditioner construction and its application [21, 20].

The second method, “incomplete biconjugation,” was proposed by Benzi [14]. Consider $A = LDU$, where $L$ is unit lower triangular, $D$ is diagonal, and $U$ is unit upper triangular, then $A^{-1}$ can be factored as $A^{-1} = U^{-1}D^{-1}L^{-1} = ZD^{-1}W^T$, where $Z = U^{-1}$ and $W = L^{-T}$ are unit upper triangular matrices ($Z = W^T$ if $A$ is symmetric). Next, using generalized Gram-Schmidt orthogonalization, approximations to $Z$ and $W$ are computed. As in incomplete factorization, selected elements are discarded based on a drop condition specified by the user. This method does not require the sparsity pattern of $Z$ and $W$ because the nonzero elements are determined dynamically. The orthogonalization algorithm, however, incurs some data dependencies between columns/rows somewhat like sparse matrix factorization algorithms. Consequently, its parallel implementation requires an ordering of the coefficient matrix to reduce these data dependencies [13].
4.3.2 Selective Inversion Using Sparse Approximate Inversion

In our ICT-SI scheme described Section 4.2 explicit matrix inversion was used for triangular dense submatrices associated with distributed supernodes. This can lead to high computation and communication overheads because dense matrix computations are used for a triangular submatrix which may eventually become very sparse from dropping. In addition, explicit matrix inversion usually produces a dense matrix even if the original matrix before the inversion is sparse [43]. Both observations indicate that ICT-SI wastes computation and storage to compute incomplete factors. Storing and computing on many zeroes will increase overheads even if dense matrix methods take advantage of the memory hierarchy for a more efficient implementation [28, 29]. The original SI scheme has another weakness in terms of parallel computation; the communication cost of the parallel dense matrix algorithm grows proportionally with the dimensions of submatrix; there are some data dependencies between rows/columns that prevent the use of asynchronous communication to improve performance. We, therefore, apply sparse approximate inverse techniques to compute the triangular submatrices that are explicitly inverted in ICT-SI. This scheme aims to exploit the benefits from sparsity of the several supernodes towards the top of the supernodal tree and the scalability of the parallel sparse approximate inverse implementations. Figure 4.2 illustrates the difference between ICT-SI and our new scheme. Our scheme applies SAI only at distributed supernodes; we call our scheme ICT with Selective Sparse Approximate Inversion (ICT-SSAI).

Fig. 4.2. Matrix associated with a distributed supernode for ICT-SSAI and ICT-SI.
4.3.3 The ICT-SSAI Algorithm and its Implementation

The ICT-SSAI scheme is once again based on the supernodal tree. We use the parallel sparse approximate preconditioner using the Frobenius norm minimization, which is available in the Parasails package developed by Chow [21, 37]. As described in the previous section, this method can achieve good efficiency and is therefore suitable for our parallel ICT-SSAI implementation.

The new algorithm is but a variant of the original ICT-SI scheme; it differs only in the treatment of the triangular submatrices. At a supernode with a pair of block submatrices, 
\[
\begin{bmatrix}
A_{11} \\
A_{21}
\end{bmatrix},
\]
are updated by previous columns in the left-looking scheme are applied to yield:
\[
\begin{bmatrix}
\hat{A}_{11} \\
\hat{A}_{21}
\end{bmatrix}.
\]
In the original SI scheme, parallel dense Cholesky factorization, \( \hat{L}_{11} \hat{L}_{11}^T \leftarrow \hat{A}_{11} \), is performed followed by the parallel dense matrix inversion, \( \hat{L}_{11}^{-1} \leftarrow \hat{L}_{11} \) and subsequent application of a drop threshold scheme. The new scheme directly computes a factored sparse approximate inverse (say \( G \)) of \( \hat{L}_{11} \). Next, the off-diagonal block is computed by distributed sparse matrix multiplication of the sparse approximate inverse, \( \hat{L}_{21} \leftarrow G \hat{A}_{21}^T \). The new incomplete factor is thus of the form:
\[
\begin{bmatrix}
G \\
\hat{L}_{21}
\end{bmatrix}.
\]
This change results in the use of sparse data structures for \( A_{11} \) and \( G \). Now our left-looking update scheme constructs a sparse matrix directly by means of integer array operations to repeatedly merge two different sparse columns using their column indices. On distributed supernodes, an additional data exchange between processors is necessary after the left-looking update scheme on individual processors.

4.4 Analysis of Parallel ICT with Selective Inversion (ICT-SI)

In this section, we analyze the performance of our parallel ICT Selective Inversion (ICT-SI) for sparse matrices associated with \( K \times K \) and \( K \times K \times K \) grids. The analysis is quite similar to our earlier analysis for parallel ICT described in Chapter 3. Once again, we use recursive equations associated with separators and corresponding submatrices at nodes of the supernodal tree. As in parallel ICT factorization, ICT-SI takes advantage of the original sequential ICT algorithm for the local-phase subtrees; our new SI feature is used only at the supernodes distributed across multiple processors.
4.4.1 Computational Costs at a Supernode

We examine the cost of ICT-SI on an \((m+n)\times n\) submatrix at distributed supernode assigned to \(p\) processors. This includes dense Cholesky factorization, computation of the off-diagonal block, rank-1 updates, and the extra explicit matrix inversion needed for ICT-SI.

For an \(n \times n\) matrix on \(p\) processors, the computational cost of explicit matrix inversion is given by (in terms of \(\text{ops}\)):

\[
A_{\text{inv}}(n,p) = \frac{1}{3p}n^3 + O(n^2).
\]

Assuming that \(\alpha\) elements are retained after applying the drop tolerance condition, we now have the cost at a supernode, henceforth \(A_{s,SI}(m,n,\alpha,p)\), as:

\[
A_{s,SI}(m,n,\alpha,p) = A_{\text{inv}}(n,p) + A_{\text{chol}}(n,p) + A_{\text{off}}(m,n,p) + A_{\text{update}}(m,n,\alpha,p)
\]

\[
= \frac{1}{3p}n^3 + \left(\frac{1}{3p}n^3 + \frac{1}{p}mn^2 + \frac{1}{2p}n\alpha^2\right)
\]

\[
= \frac{2}{3p}n^3 + \frac{1}{p}mn^2 + \frac{1}{2p}n\alpha^2.
\]

4.4.2 Communication Cost at a Supernode

The communication cost of ICT-SI includes the costs for the original parallel ICT and the one for the extra parallel matrix inversion. The communication cost of the parallel matrix inversion is different from that of parallel Cholesky factorization because the inversion algorithm is based on a fan-out scheme. Now a single processor (the owner of a column) sends a message to all the other processors for each column. Thus, for an \(n \times n\) matrix on \(p\) processors, every processor sends its own column to other processors \(n/p\) times. This indicates that the latency cost at a single processor is \((n/p \times (p-1))t_s \approx nt_s\) and the communication volume is \(\frac{1}{2p}n^2t_w\) (equivalent to the size of local submatrix).

The communication cost at an \((m+n) \times n\) supernodal matrix is therefore:

\[
C_{s,SI}(m,n,\alpha,p) = (nt_s + \frac{1}{sp}n^2t_w) + (nt_s + (\frac{1}{2}n^2 + mn)t_w)
\]

\[
= 2nt_s + (\frac{1}{2p}n^2 + \frac{1}{2}n^2 + mn)t_w.
\]

For large \(p\), \(\frac{1}{2p}\) terms are negligible. Thus, we have:

\[
C_{s,SI}(m,n,\alpha,p) \approx 2nt_s + (\frac{1}{2}n^2 + mn)t_w.
\]
4.4.3 Computational Costs: Model $K \times K$ Grids

**Lemma 4.1.** Consider the $N \times N$ sparse matrix associated with the model five-point finite difference $K \times K$ grid ($N = K^2$). Assume an optimal nested dissection ordering is used for fill-reduction and that $P$ processors are used. Then, the computational cost of the ICT-SI factorization is no more than \( \frac{38}{3P} K^3 + O(\frac{K^2}{P} \log_2 K) \).

**Proof.** Consider a sparse matrix associated with a $K \times K$ model grid on $P$ processors. When we assume that the drop condition retains $\alpha$ elements for every $K$ elements in a column, we can use the recursive equations similar to Equations 3.10–3.13 to determine the cost of the entire ICT-SI scheme; the main difference lies in the costs associated with two supernodal matrices at each “+” shaped separators. $M_{SI}(K, K, i, \alpha, P)$ is different from $M_{dist}(K, K, i, \alpha, P)$ in Equations 3.10–3.13 due to the additional matrix inversion. The computational costs for the two supernodal matrices (on $\frac{p}{2}$ and $p$ processors) associated with the “+” shaped separators on a $K \times K$ grid with 4 bordered sides are:

- $M_{small,SI} = A_{s,SI}(3K, \frac{K}{2}, \alpha, \frac{p}{2}) = \frac{5}{3p} K^3 + \frac{9}{2p} \alpha^2 K + O(\frac{K^2}{p})$
- $M_{large,SI} = A_{s,SI}(4K, K, \alpha, p) = \frac{14}{3p} K^3 + \frac{8}{p} \alpha^2 K + O(\frac{K^2}{p})$

Thus, the total cost at the supernode is:

- $M_{SI}(K, K, 4, \alpha, p) = M_{small,SI} + M_{large,SI} = \frac{19}{3p} K^3 + \frac{25}{2p} \alpha^2 K$

Consequently, costs of ICT-SI factorization are given by the following recursive equations:

- $A_{SI}(K, K, 0, \alpha, P) = A_{SI}(\frac{K}{2}, \frac{K}{2}, 2, \alpha, P/4) + M_{SI}(K, K, 0, \alpha, P)$ (4.1)
- $A_{SI}(K, K, 2, \alpha, P) = \frac{1}{4} A_{SI}(\frac{K}{2}, \frac{K}{2}, 2, \alpha, P/4) + \frac{1}{2} A_{SI}(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/4) + M_{SI}(K, K, 2, \alpha, P)$ (4.2)
- $A_{SI}(K, K, 3, \alpha, P) = \frac{1}{2} A_{SI}(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/4) + \frac{1}{2} A_{SI}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) + M_{SI}(K, K, 3, \alpha, P)$ (4.3)
- $A_{SI}(K, K, 4, \alpha, P) = A_{SI}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) + M_{SI}(K, K, 4, \alpha, P)$ (4.4)

Each equation can be solved by expanding it to a non-recursive form that comprises $\log_4 P$ terms and expressions for $M_{SI}(K, K, i, \alpha, P)$ and the cost for the local-phase subtrees, $A_{seq}(K/2^{\log_4 P}, K/2^{\log_4 P}, i, \alpha)$. For $i = 4$, the new equation is given by:
\[ A_{SI}(K, K, 4, \alpha, P) = M_{SI}(K, K, 4, \alpha, P) + M_{SI}(K/2, K/2, 4, \alpha, P/4) + \cdots + M_{SI}(K/(2^{\log_4 P - 1}), K/(2^{\log_4 P - 1}), 4, \alpha, 4) + A_{seq}(K/2^{\log_4 P}, K/2^{\log_4 P}, 4, \alpha). \]

This equation simplifies to:

\[ A_{SI}(K, K, 4, \alpha, P) = (\frac{38}{3P}K^3 + \frac{25}{P} \alpha^2 K) \sum_{j=0}^{\log_4 P - 1} \left(1 - \frac{1}{2}\right)^j + O(\frac{K^2}{P} \log_4 P) \]

\[ A_{seq}(K/2^{\log_4 P}, K/2^{\log_4 P}, 4, \alpha) \]

\[ = (\frac{19}{3P}K^3 + \frac{25}{P} \alpha^2 K)(1 - \frac{1}{2} \log_4 P) + \frac{71}{6P^{3/2}}K^3 + \frac{25}{P^{3/2}} \alpha^2 K + O(\frac{K^2}{P} \log_2 K). \]

This yields:

\[ A_{SI}(K, K, 4, \alpha, P) = (\frac{38}{3P}K^3 + \frac{25}{P} \alpha^2 K)(1 - \frac{1}{2} \log_4 P) + \frac{71}{6P^{3/2}}K^3 + \frac{25}{P^{3/2}} \alpha^2 K + O(\frac{K^2}{P} \log_2 K) \]

\[ = \frac{38}{3P}K^3 + \frac{25}{P} \alpha^2 K - \frac{5}{6P^{3/2}}K^3 + O(\frac{K^2}{P} \log_2 K). \]

As mentioned earlier in Chapter 3, \( A_{SI}(K, K, i, \alpha, P) \) has the same growth factor regardless of \( i \); the only difference is in the constants multiplying the \( K^3 \) and \( \alpha^2 K \) terms.

Thus, the computational cost of the ICT-SI is no more than \( \frac{38}{3P}K^3 + O(\frac{K^2}{P} \log_2 K) \). \( \square \)

Alternately, \( A_{SI}(K, K, i, \alpha, P) \) can be found by adding the total cost of the original parallel ICT to the cost of matrix inversion performed at all the distributed supernodes assigned to a single processor. The cost for the matrix inversion for the entire distributed phase, \( A_{inv}(K, K, P) \), is obtained by:

\[ A_{inv}(K, K, P) = \frac{5}{12P}K^3 \sum_{j=0}^{\log_4 P - 1} \left(1 - \frac{1}{\sqrt{P}}\right)^j + O(\frac{K^2}{P} \log_4 P), \]

which is equivalent to

\[ A_{inv}(K, K, P) = \frac{5}{6P}K^3(1 - \frac{1}{\sqrt{P}}) + O(\frac{K^2}{P} \log_4 P). \]

Adding \( A_{inv}(K, K, P) \) to \( A_{dist}(K, K, 4, \alpha, P) \) in Lemma 3.3, we have:

\[ A_{SI}(K, K, 4, \alpha, P) = \frac{71}{6P}K^3 + \frac{25}{P} \alpha^2 K + \frac{5}{6P}K^3(1 - \frac{1}{\sqrt{P}}) \]

\[ = \frac{38}{3P}K^3 + \frac{25}{P} \alpha^2 K - \frac{5}{6P^{3/2}}K^3 + O(\frac{K^2}{P} \log_2 K). \]
4.4.4 Communication Costs: Model $K \times K$ Grids

**Lemma 4.2.** Consider the $N \times N$ sparse matrix associated with the model five-point finite difference $K \times K$ grid ($N = K^2$). Assume an optimal nested dissection ordering is used for fill-reduction. Then, the number of messages sent by a single processor during ICT-SI with $P$ processors is no more than $6K$ and the corresponding volume of communication is no more than $9K^2$.

**Proof.** Using the cost of ICT-SI at a supernode (in Section 4.4.2), the costs at the two supernodes associated with “+” shaped separators of a $K \times K$ grid using $p$ processors is given by:

$Q_{small,SI} = C_{s,SI}(3K, \frac{K}{2}, \alpha, p/2) = Kt_s + \left(\frac{1}{4}K^2 + \frac{3}{2}K^2\right)t_w = Kt_s + \frac{7}{4}K^2t_w$

$Q_{large,SI} = C_{s,SI}(4K, K, \alpha, p) = 2Kt_s + (K^2 + 4K^2)t_w = 2Kt_s + 5K^2t_w$

$Q_{SI}(K, K, 4, \alpha, p) = Q_{small,SI} + Q_{large,SI} = 3Kt_w + \frac{27}{4}K^2t_w$.

We use the recursive equations similar to Equation 3.2.2.4, but after replacing $Q$ and $C$ by $Q_{SI}$ and $C_{SI}$. Then, the total communication cost for ICT-SI on a $K \times K$ grid bordered on 4 sides, henceforth $C_{SI}(K, K, 4, \alpha, P)$, is:

$C_{SI}(K, K, 4, \alpha, P) = Q_{SI}(K, K, 4, \alpha, P) + Q_{SI}(K/2, K/2, 4, \alpha, P/4) + \cdots + Q(K/(2^{\log_4 P - 1}), K/(2^{\log_4 P - 1}), 4, \alpha, 4)$.

This is expressed using a geometric series as follows:

$C(K, K, 4, \alpha, P) = 3K \sum_{j=0}^{\log_4 P - 1} \left(\frac{1}{2}\right)^jt_s + \frac{27}{4}K^2 \sum_{j=0}^{\log_4 P - 1} \left(\frac{1}{4}\right)^jt_w$.

This expression simplifies to:

$C(K, K, 4, \alpha, P) = 6K(1 - \frac{1}{\sqrt{P}})t_s + 9K^2(1 - \frac{1}{P})t_w$.

Thus, for a large $P$:

$C(K, K, 4, \alpha, P) = 6Kt_s + 9K^2t_w$.

This indicates the communication costs of parallel ICT-SI are greater than those of parallel ICT in terms of constants that multiply the dominant terms.

4.4.5 Computational Costs: Model $K \times K \times K$ Grids

As shown in Lemma 3.2 of Section 3.2.2.5, the computational costs for sparse matrices of $K \times K \times K$ grids are determined using recursive equations based on the costs of three supernodal matrices associated with three separating planes. A similar analysis is used for our ICT-SI. We again assume that the drop condition retains $\alpha$ elements for every $K$ elements in a column.
Lemma 4.3. Consider the $N \times N$ sparse matrix associated with the model seven-point finite difference $K \times K \times K$ grid ($N = K^3$). Assume an optimal nested dissection ordering is used for fill-reduction and that $P$ processors are involved to execute the factorization. Then, the computational cost of ICT-SI factorization is no more than $\frac{76}{7P}K^6 + O\left(\frac{K^4}{P}\right)$.

Proof. We can obtain the total cost by using the costs of parallel ICT from Lemma 3.5, $A_{\text{dist}}(K, K, i, \alpha, P)$, and the cost of the matrix inversion for distributed nodes of the tree. Let $A_{\text{inv}}(K, K, K, P)$ be the cost of the matrix inversion at all distributed supernodes on a path from the local-phase tree to the root of the supernodal tree. Then, we have:

$$A_{\text{inv}}(K, K, K, P) = \frac{7}{16P}K^6 \sum_{j=0}^{\log_8 P-1} \left(\frac{1}{8}\right)^j + O\left(\frac{K^4}{P}\right),$$

which is equivalent to

$$A_{\text{inv}}(K, K, K, P) = \frac{1}{2P}K^6\left(1 - \frac{1}{P}\right) + O\left(\frac{K^4}{P}\right).$$

We now have $A_{SI}(K, K, K, 6, \alpha, P)$ as:

$$A_{SI}(K, K, K, 6, \alpha, P) = A_{\text{dist}}(K, K, 6, \alpha, P) + A_{\text{inv}}(K, K, K, P)$$

$$= \frac{145}{14P}K^6 + \frac{233}{7P^2}2^{\alpha}K^4 + \frac{1}{2P}K^6\left(1 - \frac{1}{P}\right) + O\left(\frac{K^4}{P}\right)$$

$$= \frac{76}{7P}K^6 - \frac{1}{2P^2}K^6 + \frac{233}{7P^2}2^{\alpha}K^4 + O\left(\frac{K^4}{P}\right).$$

Once again, we can use the expression as an upper bound for $A_{SI}(K, K, K, 0, \alpha, P)$. □

4.4.6 Communication Costs: Model $K \times K \times K$ Grids

Lemma 4.4. Consider the $N \times N$ sparse matrix of dimension associated with the model seven-point finite difference $K \times K \times K$ grid ($N = K^3$). Assume an optimal nested dissection ordering is used for fill-reduction. Then, the number of messages sent by a single processor is no more than $7K^2$ during ICT-SI factorization with $P$ processors and the volume of communication is no more than $\frac{53}{9}K^2$.

Proof: Consider the costs associated with 3 separating planes and thus 3 supernode associated with the separators of a grid with 6 borders.
The communication costs of the entire scheme can be expressed using the recursive simplifies to:

\[
Q_{\text{small}} = C_{S,I}(\frac{5}{2}K^2, \frac{K^2}{4}, \alpha, p, K^2) = \frac{1}{2}K^2 t_s + \left(\frac{1}{16}K^4 + \frac{5}{8}K^4\right) t_w = \frac{1}{2}K^2 t_s + \frac{11}{16}K^4 t_w
\]

\[
Q_{\text{medium}} = C_{S,I}(4K^2, \frac{K^2}{2}, \alpha, p) = K^2 t_s + \left(\frac{1}{4}K^4 + 2K^4\right) t_w = K^2 t_s + \frac{9}{4}K^4 t_w
\]

\[
Q_{\text{large}} = C_{S,I}(6K^2, K^2, \alpha, p) = 2K^2 t_s + (K^4 + 6K^4) t_w = 2K^2 t_s + 7K^4 t_w
\]

\[
Q_{SI}(K, K, 6, \alpha, p) = Q_{\text{small}} + Q_{\text{medium}} + Q_{\text{large}} = \frac{7}{2}K^2 t_s + \frac{159}{16}K^4 t_w.
\]

The communication costs of the entire scheme can be expressed using the recursive equations below:

\[
C_{SI}(K, K, 0, \alpha, P) = C_{SI}(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/8) + Q_{SI}(K, K, 0, \alpha, P)
\]  

\[
C_{SI}(K, K, 3, \alpha, P) = \frac{1}{8}C_{SI}(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/8)
\]  

\[
+ \frac{3}{8}C_{SI}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/8) + \frac{3}{8}C_{SI}(\frac{K}{2}, \frac{K}{2}, 5, \alpha, P/8)
\]

\[
+ \frac{1}{8}C_{SI}(\frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + Q_{SI}(K, K, 3, \alpha, P)
\]  

\[
C_{SI}(K, K, 4, \alpha, P) = \frac{1}{4}C_{SI}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/8) + \frac{1}{2}C_{SI}(\frac{K}{2}, \frac{K}{2}, 5, \alpha, P/8)
\]

\[
+ \frac{1}{4}C_{SI}(\frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + Q_{SI}(K, K, 4, \alpha, P)
\]  

\[
C_{SI}(K, K, 5, \alpha, P) = \frac{1}{2}C_{SI}(\frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + \frac{1}{2}C_{SI}(\frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8)
\]

\[
+ Q_{SI}(K, K, 5, \alpha, P)
\]  

\[
C_{SI}(K, K, 6, \alpha, P) = C_{SI}(\frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + Q_{SI}(K, K, 6, \alpha, P).
\]

They are almost the same as Equations 3.23–3.27 for parallel ICT in Section 3. The entire communication cost for ICT-SI with 6 bordered planes, \(C_{SI}(K, K, 6, \alpha, P)\), simplifies to:

\[
C_{SI}(K, K, 6, \alpha, P) = Q_{SI}(K, K, 6, \alpha, P) + Q_{SI}(K/2, K/2, 6, \alpha, P/4) +
\]

\[
\ldots + Q(K/(2^{\log_2 P - 1}), K/(8^{\log_4 P - 1}), K/(8^{\log_4 P - 1}), 6, \alpha, 8)
\]

\[
= \frac{7}{2}K^2 \sum_{j=0}^{\log_8 P - 1} \left(\frac{1}{4}\right)^j t_s + \frac{159}{16}K^4 \sum_{j=0}^{\log_8 P - 1} \left(\frac{1}{16}\right)^j t_w.
\]

This can be further simplified to:

\[
C_{SI}(K, K, 6, \alpha, P) = 7K^2 \left(1 - \left(\frac{1}{P}\right)^{2/3}\right) t_s + \frac{53}{5}K^4 \left(1 - \left(\frac{1}{P}\right)^{4/3}\right) t_w.
\]
Thus, for a large $P$:

$$C_{SI}(K, K, K, 6, \alpha, P) = 7K^2 t_s + \frac{53}{5} K^4 t_w.$$  

As seen in the analysis of parallel ICT in Chapter 3, the communication cost of ICT-SI depends on the grid size, but it has larger constants associated with the dominant terms due to the additional costs of parallel matrix inversion.

### 4.5 Analysis of Parallel ICT with Selective Sparse Approximate Inversion (ICT-SSAI)

The parallel ICT-SSAI differs from ICT-SI with respect to the overheads at distributed supernodes from the ICT-SI because sparse approximate inversion is used instead explicit inversion.

#### 4.5.1 Computational Costs at a Supernode

In ICT-SSAI, the computation at a supernode is the form of three basic operations: sparse approximate inversion, sparse matrix-vector multiplication, and sparse rank-$K$ updates. The first two operations make the performance model for ICT-SSAI different from that of ICT-SI. We now consider the cost model for these newly introduced operations, and then estimate the costs at an $(m + n) \times n$ supernodal matrix.

- The cost of computing a sparse approximate inverse of an $n \times n$ sparse matrix with $p$ processors is $\frac{1}{3p} c^3 n$ flops; we ignore the non-dominant term, $O(\frac{c^2 n}{p})$, and we assume that the sparse matrix has $c$ elements ($c < n$) in every row/column. The expression is based on costs for Parasails (developed by Chow [21]).

- The cost of sparse matrix-vector multiplication of an $n \times n$ matrix with $m$ vectors is $\frac{2}{p} c mn$ flops; we assume that the sparse matrix has $c$ elements ($c < n$) in every row/column.

Then, we can model the cost of a $p$ processor implementation at an $(m + n) \times n$ supernodal matrix by the equation (assuming that the $n \times n$ diagonal submatrix has $\alpha$ elements in every column, and that the $m \times n$ off-diagonal submatrix has $\alpha$ elements in each column of both submatrices). Thus, the off-diagonal submatrix has the different degree of sparsity from the diagonal submatrix. The computational cost is:

$$A_{s, SSAI}(m, n, \alpha, p) = A_{sai}(n, \alpha, p) + A_{smv}(m, n, \alpha, p) + A_{update, p}(m, n, \alpha, p)$$

$$= \frac{1}{3p} \alpha^3 n + \frac{2}{p} \alpha mn + \frac{1}{2p} \alpha^2 n.$$  

Note that the third term represents for the cost of the sparse rank-$k$ updates discussed earlier in Section 3.2.1.1.
4.5.2 Communication Costs at a Supernode

The communication cost at a supernode comprises the sum of costs for the three basic operations: sparse approximate inversion, sparse matrix vector multiplication, and parallel sparse rank-$k$ updates. Unlike ICT and ICT-SI, each operation requires separate communication. We thus begin by analyzing the cost of each of these basic operations, and then using them to estimate the total cost at an $(m+n) \times n$ supernodal matrix distributed on $p$ processors.

4.5.2.1 Costs of Sparse Approximate Inversion (SAI)

The sparse approximate inverse algorithm [21] performs communication before computing numerical values of the approximate inverse. The first communication step is to determine, at every processor, the nonzero elements it requires from other processors (in order to avoid further communication during to computation of the inverse). Next, each processor sends its local nonzero elements including row/column indices, and prepares to receive nonzero elements from other processors. The number of messages sent by each processor varies because it depends on the distribution of nonzero elements to processors. In our scheme, an $n \times n$ sparse matrix is distributed in a simple column block fashion on $p$ processors. We assume that each processor communicates with a half of the total number of processors (on average) to perform such data exchange, and that a quarter of the local matrix data is transferred over all messages. A similar data exchange is performed for gathering non-local right-hand side elements for parallel sparse matrix-vector multiplication for computing the off-diagonal block submatrix.

For the first step, every processor starts by sending $2(p-1)$ messages, and the size of all messages is $2(p-1)$. Then, $\frac{p}{2}$ messages in the second step are sent to exchange nonzero elements twice; once for the nonzero indices and again for corresponding numerical values, for a total of $\frac{1}{4p}an \times 2 = \frac{1}{2p}an$. Assuming $p - 1 \approx p$, the communication costs are given by:

$$C_{sai}(m, \alpha, p) = 3pt_s + (2p + \frac{1}{2p}an)t_w.$$  

4.5.2.2 Costs of Sparse Matrix Vector Multiplication

Sparse approximate inversion of the triangular submatrix at a supernode is followed by using its parallel sparse matrix-vector multiplication step to factor the $m \times n$ off-diagonal submatrix at a supernode. During the computation, our implementation stores
off-diagonal elements in a dense matrix buffer where the total average costs correspond to \( n \) vectors of size \( m \). The communication pattern of the matrix-vector multiplication is quite similar to that of the sparse approximate inverse computation. More precisely, the right-hand side vector is distributed in a simple row/column block. Before performing multiplication with its local elements in the matrix, every processor sends some of its local right-hand side elements if other processors need to access to such elements. At the same time, every processor invokes a receive operation to get these non-local elements. Once all the non-local elements are received, the multiplication is performed in an embarrassingly parallel manner. In the worst case, each processor receives the entire non-local right-hand side from the rest of the processors. However, for our analysis, we assume that each processor receives only a half of its local right-hand side vector and that sends the vector to only half of all the other processors.

Assuming the data exchange for multiple vectors are handled at once, the number of messages sent by a single processor is exactly the same as in sparse approximate inverse. Thus, the communication costs are given by:

\[
C_{\text{smv}}(m, n, \alpha, p) = pt_s + \frac{1}{p}mnt_w.
\]

### 4.5.2.3 Costs of Rank-\( k \) Updates

In our distributed implementation, the parallel rank-\( k \) updates create the distributed sparse matrix assigned to the ancestor supernode. The scheme initiates rank-\( k \) updates with respect to local off-diagonal elements before the next communication step. In the latter, each processor sends some of its matrix product to other processors to prepare a sparse matrix for the sparse approximate inverse computation in the ancestor supernode. For an \((m + n) \times n\) supernodal matrix with \( p \) processors, the rank-\( k \) update is performed on the \( m \times n \) off-diagonal matrix for the next supernodal matrix distributed on \( 2p \) processors. For this instance, we assume that the local rank-\( k \) updates create a \( m \times m \) matrix where every row/column has \( \alpha \) elements. As a result, the number of messages sent by a single processor is \( 2(2p - 1) \): two messages (index and values for nonzero elements) for each destination. The amount of data sent by a single processor is \( 2\frac{2p-1}{2p} \alpha n \) for indices and values of nonzero elements. For simplicity, we let \( 2p - 1 \approx 2p \).

Now the communication costs for the rank-\( k \) updates at the supernode are:

\[
C_{\text{update}}(m, n, \alpha, p) = 4pt_s + 2\alpha m t_w.
\]

### 4.5.2.4 Communication Costs of ICT-SSAI at a Supernode

Based on the cost model of three basic operations, the communication costs at a \((m + n) \times n\) supernodal matrix on \( p \) processor is given by the following equation.

\[
C_{s, \text{SSAI}}(m, n, \alpha, p) = C_{\text{sai}}(n, \alpha, p) + C_{\text{smv}}(m, n, \alpha, p) + C_{\text{update}}(m, n, \alpha, p)
\]

\[
= 8pt_s + (2p + \frac{1}{2p} \alpha n + \frac{1}{p}mn + 2\alpha m)t_w.
\]
4.5.3 Computational Costs: Model $K \times K$ Grids

Lemma 4.5. Consider the $N \times N$ sparse matrix associated with the model five-point finite difference $K \times K$ grid ($N = K^2$). Assume an optimal nested dissection ordering is used for fill-reduction and that the drop threshold condition retains $\alpha$ nonzeros for every $K$ elements. Then, the computational cost of ICT-SSAI using $P$ processors is $O\left( \frac{K^3}{P^{1.5}} + \frac{K^2 \log_2 K}{P} \right)$.

Proof. Consider the computational costs of ICT-SSAI at two different supernodes associated with a “+” shaped separator of a $K \times K$ grid with 4 borders:

$M_{\text{small,SSAI}} = A_{\text{sai}}(K/2, \alpha, p/2) + A_{\text{smv}}(K/2, 3K, \alpha, p/2) + A_{\text{update}}(3K, K/2, \alpha, p/2)$

$= \frac{1}{24} \alpha^3 K + \frac{3}{p} \alpha K^2 + \frac{9}{2p} \alpha^2 K$

$M_{\text{large,SSAI}} = A_{\text{sai}}(K, \alpha, p) + A_{\text{smv}}(K, 4K, \alpha, p) + A_{\text{update}}(4K, K, \alpha, p)$

$= \frac{1}{3p} \alpha^3 K + \frac{8}{p} \alpha K^2 + \frac{8}{p} \alpha^2 K.$

The total cost associated with the “+” shaped separators is given by:

$M_{\text{SSAI}}(K, K, 4, \alpha, p) = M_{\text{small,SSAI}} + M_{\text{large,SSAI}} = \frac{9}{24p} \alpha^3 K + \frac{11}{p} \alpha K^2 + \frac{25}{2p} \alpha^2 K.$

Now the cost of the entire ICT-SSAI factorization, $A_{\text{SSAI}}(K, K, i, \alpha, P)$ can be determined by solving the equations below:

$A_{\text{SSAI}}(K, K, 0, \alpha, P) = A_{\text{SSAI}}(\frac{K}{2}, \frac{K}{2}, 2, \alpha, P/4) + M_{\text{SSAI}}(K, K, 0, \alpha, P)$  \hspace{1cm} (4.10)

$A_{\text{SSAI}}(K, K, 2, \alpha, P) = \frac{1}{4} A_{\text{SSAI}}(\frac{K}{2}, \frac{K}{2}, 2, \alpha, P/4) + \frac{1}{2} A_{\text{SSAI}}(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/4) +

\frac{1}{4} A_{\text{SSAI}}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) + M_{\text{SSAI}}(K, K, 2, \alpha, P)$  \hspace{1cm} (4.11)

$A_{\text{SSAI}}(K, K, 3, \alpha, P) = \frac{1}{2} A_{\text{SSAI}}(\frac{K}{2}, \frac{K}{2}, 3, \alpha, P/4) + \frac{1}{2} A_{\text{SSAI}}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4)$

$+ M_{\text{SSAI}}(K, K, 3, \alpha, P)$  \hspace{1cm} (4.12)

$A_{\text{SSAI}}(K, K, 4, \alpha, P) = A_{\text{SSAI}}(\frac{K}{2}, \frac{K}{2}, 4, \alpha, P/4) + M_{\text{SSAI}}(K, K, 4, \alpha, P).$  \hspace{1cm} (4.13)

Once $A_{\text{SSAI}}(K, K, 4, \alpha, P)$ and other $M_{\text{SSAI}}(K, K, i, \alpha, P)$ are found, they are applied to the equations for $i = 3, 2, 0$ to derive $A_{\text{SSAI}}(K, K, 0, \alpha, P)$. For $i = 4$, Equation 4.13 is expanded to a non-recursive equation with $\log_4 P$ terms and the term for the cost of the local-phase subtrees:
\(A_{SSAI}(K, K, 4, \alpha, P) = M_{SSAI}(K, K, 4, \alpha, P) + M_{SSAI}(K/2, K/2, 4, \alpha, P/4) + \ldots + M_{SSAI}(K/(2 \log_4 P - 1), K/(2 \log_4 P - 1), 4, \alpha, 4) + A_{seq}(K/2 \log_4 P, K/2 \log_4 P, 4, \alpha).\)

We now apply a geometric series for the terms of \(M_{SSAI}\) to yield:

\[
A_{SSAI}(K, 4, \alpha, P) = \left(\frac{9}{24P} \alpha^3 K\right) \sum_{j=0}^{\log_4 P - 1} (1 - \frac{1}{4})^j + (\frac{11}{P} \alpha K^2 + \frac{25}{2P} \alpha^2 K) \sum_{j=0}^{\log_4 P - 1} (1 - \frac{1}{2})^j +
\]

\[
A_{seq}(K/2 \log_4 P, K/2 \log_4 P, 4, \alpha)
\]

\[
= \frac{1}{2P} \alpha^3 K (1 - \frac{1}{4} \log_4 P^4) +
\]

\[
(\frac{22}{P} \alpha K^2 + \frac{25}{P} \alpha^2 K)(1 - \frac{1}{2} \log_4 P^4) +
\]

\[
\frac{71}{6P1.5} K^3 + \frac{25}{P1.5} \alpha^2 K + O((\log_2 K - \log_2 P) \frac{K^2}{P}).
\]

This expression simplifies to:

\[
A_{SSAI}(K, 4, \alpha, P) = \frac{1}{2P} \alpha^3 K (1 - \frac{1}{P}) + (\frac{22}{P} \alpha K^2 + \frac{25}{P} \alpha^2 K)(1 - \frac{1}{\sqrt{P}}) +
\]

\[
\frac{71}{6P1.5} K^3 + \frac{25}{P1.5} \alpha^2 K + O((\log_2 K - \log_4 P) \frac{K^2}{P}).
\]

The expression indicates that the dominant terms are \(\frac{K^3}{P1.5}\) and \(\frac{K^2 \log_2 K}{P}\) for \(\alpha \ll K\) and \(P < K\). Therefore, the computational cost of ICT-SSAI grows as \(O\left(\frac{K^3}{P1.5} + \frac{K^2 \log_2 K}{P}\right)\),

which is lower than the costs for ICT-SI (\(O\left(\frac{K^3}{P}\right)\)). \(\Box\)

**4.5.4 Communication Costs: Model \(K \times K\) Grids**

**Lemma 4.6.** Consider the \(N \times N\) sparse matrix associated with the model finite difference 2 dimensional \(K \times K\) grid \((N = K^2)\). Assume an optimal nested dissection ordering is used for fill-reduction and \(\alpha\) elements are retained for every \(K\) elements in a column. Then, the number of messages sent by a single processor during ICT-SSAI using \(P\) processors is no more than \(16P\), and the communication volume is no more than \((3P + 11\alpha K) + (\frac{3}{4P} \alpha K + \frac{9}{2P} K^2) \log_4 P\).
PROOF. Consider the communication costs at the supernodes associated with the “+” shaped separators on a $K \times K$ grid bordered by 4 sides. Applying the cost at a supernode (derived in Section 4.5.2), the costs for two different supernodal matrices are given by:

$$Q_{\text{small,SSAI}} = C_{s,\text{SSAI}}(3K, K/2, \alpha, p/2)$$

$$= C_{\text{sai}}(K/2, \alpha, P/2) + C_{\text{smv}}(K/2, 3K, \alpha, P/2) + C_{\text{update}}(3K, K/2, \alpha, P/2)$$

$$= 4pt_s + (p + \frac{1}{4p} \alpha K + \frac{1}{2p} K^2 + 3\alpha K)t_w$$

$$Q_{\text{large,SSAI}} = C_{s,\text{SSAI}}(4K, K, \alpha, p)$$

$$= C_{\text{sai}}(K, \alpha, P) + C_{\text{smv}}(K, 4K, \alpha, P) + C_{\text{update}}(4K, K, \alpha, P)$$

$$= 8pt_s + (2p + \frac{1}{2p} \alpha K + \frac{4}{p} K^2 + 8\alpha K)t_w$$

Therefore, the total communication costs at the “+” shaped separators are:

$$Q_{\text{SSAI}}(K, K, 4, \alpha, p) = Q_{\text{small,ssai}} + Q_{\text{large,ssai}}$$

$$= 12pt_s + (3p + \frac{3}{4p} \alpha K + \frac{9}{2p} K^2 + 11\alpha K)t_w.$$
4.5.5 Computational Costs: Model $K \times K \times K$ Grids

**Lemma 4.7.** Consider the $N \times N$ sparse matrix $N$ associated with the model finite difference 3 dimensional $K \times K \times K$ grid ($N = K^3$). Assume an optimal nested dissection ordering is used for fill-reduction. Then, the computational cost of ICT-SSAI factorization with $P$ processors is $O\left(\frac{K^6}{P^2}\right)$.

**Proof.** We again assume that the drop condition retains $\alpha$ elements for every $K$ elements in a column. The costs of supernodes associated with the three separator planes shown in Figure 3.7 for a grid with 6 borders ($i = 6$) are given by:

$$M_{small,SSAI} = A_s,SSAI\left(\frac{5}{2}K^2, \frac{K^2}{4}, \alpha, P\right)$$

$$= A_{sai}\left(\frac{K^2}{4}, \alpha, P/4\right) + A_{smv}\left(\frac{K^2}{4}, \frac{5}{2}K^2, \alpha, P/4\right) + A_{update}\left(\frac{5}{2}K^2, \frac{K^2}{4}, \alpha, P/4\right)$$

$$= \frac{4}{3P}\left(\frac{\alpha K^2}{K}\right)^3 K^2 \left(\frac{K^2}{4}\right) + 8 \frac{\alpha K^2}{P} \frac{K^2}{2} \frac{K^2}{4} + \frac{4}{2P}\left(\frac{\alpha K^2}{K}\right)^2 \frac{K^2}{4}$$

$$= \frac{1}{192P} \alpha^3 K^5 + \frac{5}{4P} \alpha K^5 + \frac{25}{8P} \alpha^2 K^4$$

$$M_{medium,SSAI} = A_s,SSAI\left(4K^2, \frac{K^2}{2}, \alpha, P\right)$$

$$= A_{sai}\left(\frac{K^2}{2}, \alpha, P/2\right) + A_{smv}\left(\frac{K^2}{2}, 4K^2, \alpha, P/2\right) + A_{update}(4K^2, \frac{K^2}{2}, \alpha, P/2)$$

$$= \frac{2}{3P}\left(\frac{\alpha K^2}{K}\right)^3 K^2 \left(\frac{K^2}{2}\right) + 4 \frac{\alpha K^2}{P} \frac{K^2}{2} \frac{K^2}{2} + \frac{1}{P}\left(\frac{\alpha K^2}{K}\right) \frac{K^2}{2}$$

$$= \frac{1}{24P} \alpha^3 K^5 + \frac{4}{P} \alpha K^5 + \frac{8}{P} \alpha^2 K^4$$

$$M_{large,SSAI} = A_s,SSAI\left(6K^2, \frac{K^2}{2}, \alpha, P\right)$$

$$= A_{sai}(K^2, \alpha, P) + A_{smv}(K^2, 6K^2, \alpha, P) + A_{update}(6K^2, \frac{K^2}{2}, \alpha, P)$$

$$= \frac{1}{3P}\left(\frac{\alpha K^2}{K}\right)^3 K^2 + 2 \frac{\alpha K^2}{P} \frac{K^2}{2} K^2 + \frac{2}{P}\left(\frac{\alpha K^2}{K}\right)^2 K^2$$

$$= \frac{1}{3P} \alpha^3 K^5 + \frac{12}{P} \alpha K^5 + \frac{16}{P} \alpha^2 K^4.$$ 

Summing the three terms above, the cost is obtained as:

$$M_{SSAI}(K, K, 6, \alpha, P) = \frac{73}{192P} \alpha^3 K^5 + \frac{69}{4P} \alpha K^5 + \frac{233}{8P} \alpha^2 K^4.$$ 

$M_{SSAI}(K, K, i, \alpha, P)$ for $i = 0, 3, 4, 5$ can be determined similarly. Then, the costs of ICT-SSAI for a $K \times K \times K$ grid are given by:
\[
A_{SSAI}(K, K, 0, \alpha, P) = A_{SSAI}(K/2, K/2, K/2, 2, \alpha, P/8) + M_{SSAI}(K, K, 0, \alpha, P) \quad (4.14)
\]

\[
A_{SSAI}(K, K, 3, \alpha, P) = \frac{1}{8} A_{SSAI}(K/2, K/2, K/2, 3, \alpha, P/8) + \frac{3}{8} A_{SSAI}(K/2, K/2, K/2, 4, \alpha, P/8) + M_{SSAI}(K, K, 3, \alpha, P) \quad (4.15)
\]

\[
A_{SSAI}(K, K, 4, \alpha, P) = \frac{1}{4} A_{SSAI}(K/2, K/2, K/2, 4, \alpha, P/8) + \frac{1}{2} A_{SSAI}(K/2, K/2, K/2, 5, \alpha, P/8) + M_{SSAI}(K, K, 4, \alpha, P) \quad (4.16)
\]

\[
A_{SSAI}(K, K, 5, \alpha, P) = \frac{1}{2} A_{SSAI}(K/2, K/2, K/2, 5, \alpha, P/8) + \frac{1}{2} A_{SSAI}(K/2, K/2, K/2, 6, \alpha, P/8) + M_{SSAI}(K, K, 5, \alpha, P) \quad (4.17)
\]

\[
A_{SSAI}(K, K, 6, \alpha, P) = A_{SSAI}(K/2, K/2, K/2, 6, \alpha, P/8) + M_{SSAI}(K, K, 6, \alpha, P). \quad (4.18)
\]

We solve the equation for \( i = 6 \) to provide an upper bound of the cost on the \( K \times K \times K \) grid (for \( i = 0 \)) as:

\[
A_{SSAI}(K, K, 6, \alpha, P) = M_{SSAI}(K, K, 6, \alpha, P) + M_{SSAI}(K/2, K/2, K/2, 6, \alpha, P/8) + \cdots + M_{SSAI}(K/2^{\log_8 P-1}, K/2^{\log_8 P-1}, K/2^{\log_8 P-1}, 6, \alpha, 8) + A(K/2^{\log_8 P}, K/2^{\log_8 P}, K/2^{\log_8 P}, 6, \alpha),
\]

where the last term corresponds to the cost of the local-phase subtrees. This equations is simplified to a geometric series as:

\[
A_{SSAI}(K, K, 6, \alpha, P) = \frac{73}{192P} \alpha^3 K^5 \log_8 P - 1 \sum_{j=0}^{\log_8 P - 1} \left( \frac{1}{32} \right)^j + \left( \frac{69}{4P} \alpha K^5 \right)
\]

\[
+ \frac{233}{8P} \alpha^3 K^4 \sum_{j=0}^{\log_8 P - 1} \left( \frac{1}{8} \right)^j
\]

\[
+ \frac{145}{14P^2} K^6 + \frac{233}{7P^2} \alpha^2 K^4 + O\left( \frac{K^4}{P} \right),
\]

which yields:

\[
A_{SSAI}(K, K, 6, \alpha, P) = \frac{73}{186P} \alpha^3 K^5 (1 - \frac{1}{P^{3/3}}) + \left( \frac{138}{7P} \alpha K^5 + \frac{233}{7P} \alpha^2 K^4 \right) (1 - \frac{1}{P})
\]

\[
+ \frac{145}{14P^2} K^6 + \frac{233}{7P^2} \alpha^2 K^4 + O\left( \frac{K^4}{P} \right).
\]

The expression above indicates that the costs grow as \( O\left( \frac{K^6}{P^2} \right) \) for \( \alpha \ll K \) and \( P < K \). Therefore, the computational cost of ICT-SSAI is lower than the costs of ICT-SI \( O\left( \frac{K^6}{P^2} \right) \). \( \square \)
4.5.6 Communication Costs: Model $K \times K \times K$ Grids

**Lemma 4.8.** Consider the $N \times N$ sparse matrix associated with the model seven-point finite difference $K \times K \times K$ grid ($N = K^3$). Assume an optimal nested dissection ordering is used for fill-reduction and that $\alpha$ elements are retained for every $K$ elements in a column. Then, the number of messages sent by a single processor during parallel ICT-SSAI factorization using $P$ processors is no more than $16P$ and the corresponding communication volume is no more than $4P + \left(\frac{7}{4P} \alpha K^3 + \frac{25}{2P} K^4\right) + \frac{1864}{15} \alpha K^3$.

**Proof.** The recursive analysis for the communication cost is again based on the costs at supernodal matrices associated with 3 separator planes that partition the grid into 8 subgrids. The costs at these 3 supernodal matrices (using the result in Section 4.5.2) are:

\[
Q_{\text{small,SSAI}} = C_{s,\text{SSAI}}(\frac{5}{2} K^2, \frac{K^2}{4}, \alpha, p/4) = Q_{\text{sai}}(\frac{K^2}{4}, \alpha, p/4) + Q_{\text{smv}}(\frac{5}{2} K^2, \alpha, p/4) + Q_{\text{update}}(\frac{5}{2} K^2, \frac{K^2}{4}, \alpha, p/4)
\]

\[= 2pt_s + \left(\frac{p}{2} + \frac{1}{8p} \alpha K^3 + \frac{5}{2p} K^4 + \frac{25}{2} \alpha K^3\right)t_w\]

\[
Q_{\text{medium,SSAI}} = C_{s,\text{SSAI}}(4K^2, \frac{K^2}{2}, \alpha, p/4) = Q_{\text{sai}}(\frac{K^2}{2}, \alpha, p/2) + Q_{\text{smv}}(4K^2, \alpha, p/2) + Q_{\text{update}}(4K^2, \frac{K^2}{2}, \alpha, p/2)
\]

\[= 4pt_s + (p + \frac{1}{4p} \alpha K^3 + \frac{4}{p} K^4 + 32 \alpha K^3)t_w\]

\[
Q_{\text{large,SSAI}} = C_{s,\text{SSAI}}(6K^2, K^2, \alpha, p) = Q_{\text{sai}}(K^2, \alpha, p) + Q_{\text{smv}}(6K^2, \alpha, p) + Q_{\text{update}}(6K^2, K^2, \alpha, p)
\]

\[= 8pt_s + (2p + \frac{1}{2p} \alpha K^3 + \frac{6}{p} K^4 + 72 \alpha K^3)t_w\]

Consequently, the total cost of all three supernodal matrices is:

\[
Q_{\text{SSAI}}(K, K, 6, \alpha, P) = 14pt_s + \left(\frac{7}{2} P + \frac{7}{8P} \alpha K^3 + \frac{25}{2P} K^4 + \frac{233}{2} \alpha K^3\right)t_w.
\]

The communication costs for $i = 0, 3, 4, 5$ are determined similarly. Then, the communication cost for the entire matrix is:
\[ C_{SSAI}(K, K, 0, \alpha, P) = C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 3, \alpha, P/8) + Q_{SSAI}(K, K, 0, \alpha, P) \] (4.19)

\[ C_{SSAI}(K, K, 3, \alpha, P) = \frac{1}{8} C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 3, \alpha, P/8) \]

\[ + \frac{3}{8} C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 4, \alpha, P/8) + \frac{3}{8} C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 5, \alpha, P/8) \]

\[ + \frac{1}{8} C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + Q_{SSAI}(K, K, 3, \alpha, P) \] (4.20)

\[ C_{SSAI}(K, K, 4, \alpha, P) = \frac{1}{4} C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 4, \alpha, P/8) + \frac{1}{2} C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 5, \alpha, P/8) \]

\[ + \frac{1}{4} C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + Q_{SSAI}(K, K, 4, \alpha, P) \] (4.21)

\[ C_{SSAI}(K, K, 5, \alpha, P) = \frac{1}{2} C(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + \frac{1}{2} C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) \]

\[ + Q_{SSAI}(K, K, 5, \alpha, P) \] (4.22)

\[ C_{SSAI}(K, K, 6, \alpha, P) = C_{SSAI}(\frac{K}{2}, \frac{K}{2}, \frac{K}{2}, 6, \alpha, P/8) + Q_{SSAI}(K, K, 6, \alpha, P) \] (4.23)

Each recursive equation is solved expanding it to a non-recursive form. For \( i = 6 \), the non-recursive form of \( C_{SSAI}(K, K, 6, \alpha, P) \) is given by:

\[ C_{SSAI}(K, K, 6, \alpha, P) = \sum_{j=0}^{\log_2 P-1} \left( \left( \frac{1}{8} \right)^j \right) t_s + \]

\[ \frac{7}{2} P \sum_{j=0}^{\log_2 P-1} \left( \frac{1}{8} \right)^j \left( \frac{7}{8} \alpha K^3 + \frac{25}{2P} K^4 \right) \sum_{j=0}^{\log_2 P-1} \left( \frac{1}{2} \right)^j + \]

\[ \frac{233}{2} \alpha K^3 \sum_{j=0}^{\log_2 P-1} \left( \frac{1}{16} \right)^j t_w. \]

This simplifies to:

\[ C_{SSAI}(K, K, 6, \alpha, P) = 16P(1 - \frac{1}{P}) t_s + \]

\[ (4P(1 - \frac{1}{P}) + \left( \frac{7}{4P} \alpha K^3 + \frac{25}{P} K^4 \right)(1 - \frac{1}{\sqrt[3]{P}}) + \]

\[ \frac{1864}{15} \alpha K^3 \left( 1 - \frac{1}{P^{4/3}} \right) t_w. \]

As in the 2-dimensional case, ICT-SSAI is more latency tolerant than parallel ICT and ICT-SI because its latency cost is independent of the grid size. Furthermore, the communication volume of ICT-SSAI is asymptotically smaller (at \( \frac{K^4}{P} \)) than for parallel ICT and ICT-SI (at \( K^4 \)) for \( \alpha \ll K \).
4.6 Summary

We have developed two different modification schemes (SI and SSAI) to ICT to allow a latency tolerant application. These modification to ICT is based on extra computation to selectively invert a portion of each distributed supernodal submatrix. ICT-SI performs explicit dense matrix inversion for such triangular submatrices, while ICT-SSAI takes advantage of the sparsity of such submatrices to selectively use sparse approximate inversion. Our performance analysis indicates that ICT-SSAI is more efficient than ICT-SI in terms of both computational costs ($O(\frac{K^3}{P^{1.5}} + \frac{K^2 \log_2 K}{P})$ vs. $O(\frac{K^3}{P})$ for 2-dimensional problems and $O(\frac{K^6}{P^2})$ vs. $O(\frac{K^6}{P})$ for 3-dimensional problems) and interprocessor communication latencies ($16P$ vs. $6K$ for 2-dimensional problems and $16P$ vs. $7K^2$ for 3-dimensional problems).
Chapter 5

Multiprocessors, Sparse-Matrix Test Suites, and Solution Schemes Used for Empirical Evaluations

This chapter describes the conditions under which our experiments were performed to evaluate the performance of our parallel ICT schemes; the results of our experiments are in Chapters 6, 7 and 8. Section 5.1 provides a brief description of distributed memory multiprocessors. Section 5.2 introduces the test suite of sparse matrices, and Section 5.3 provides the overview of the parallel sparse linear solvers that were used in our experiments.

5.1 Distributed Memory Multiprocessors

Throughout this dissertation, our discussions focus on implementations on distributed memory multiprocessors. These computers have a number of processors connected with either a memory bus or an interconnection network. Unlike shared memory multiprocessors, each processor has its own address space and remote memory access is through explicit message passing libraries such as MPI [48, 102], where the user’s program directly controls a data exchange.

In such distributed memory machines, interprocessor communication costs can critically affect application performance. The communication costs of sending $\sigma$ words (typically 4 bytes per word for 32 bit architecture and 8 bytes for 64 bit architecture) between two processors can be modeled as:

$$t_{comm} = t_s + \sigma t_w$$

where $t_s$ is the latency or startup time, and $t_w$ is the time to transfer a single word. Both $t_s$ and $t_w$ can be approximated by constants specific to a given multiprocessor. More information on the communication cost model can be found in [46].

For the empirical results provided later in Chapters 6, 7, and 8, we used the Jazz Linux cluster at Argonne National Laboratory (Chapters 6 and 8) and the IBM SP located at NERSC, Lawrence Berkeley National Laboratory (Chapter 7).
The Jazz Linux Cluster. The Jazz Linux cluster [66] located at Argonne National Laboratory consists of 350 2.4 GHz Intel Xeon nodes with Myrinet2000 [82] interconnect running with the Red Hat Linux operating system. The Xeon processor is a 32-bit superscalar processor with two floating point computation units (4.8 Gflops at peak) and higher memory-bandwidth (3.2 Gbytes per second) than other older Pentium CPUs. Each computational node has either 1 or 2 Gbytes memory. Myrinet2000 is a high-performance packet communication system which achieves less than 10 microsecond latency and 16 nanoseconds per word data transfer (4 bytes per word, 250 Mbytes/sec bandwidth) with the GM-MPI implementation (optimized for Myrinet2000). The Jazz Linux cluster was ranked 235 among the top 500 supercomputers list in June 2004 [79].

The IBM SP at NERSC. The IBM SP is a distributed memory multiprocessor with multiple nodes of 16-way 375Mhz Power3 processor SMP and 16–64 Gbyte memory. Each node works as a stand-alone machine, and each processor in the node is able to execute programs individually. Therefore, a processor located in a single SMP node can be regarded as a single node of distributed multiprocessors. A parallel program implemented with MPI emulates message passing within a shared memory system if a message passing occurs between processors in the same SMP node.

The Power3 processor is a 64-bit superscalar architecture that contains 8 scalar units: 2 floating point, 3 fixed point, 2 load store and 1 branch unit. Its theoretical peak speed for floating point computation is 1.5 Gflops and its peak memory bandwidth is 16 Gbytes per second. The size of level-1 cache is 64 and 32 Kbytes for data and instruction respectively, and the size of level-2 cache is 8 Mbytes. The interconnection between nodes is implemented by a bidirectional multistage under 300 nano seconds with 80 nodes (at the hardware level). The communication latency of the message passing by MPI is 20 microsecond regardless of the location of processors, and the data transfer rate is less than 24 nanoseconds per word (8 bytes per word, 350 Mbytes/sec bandwidth). The IBM SP at NERSC has 380 nodes and was ranked 14 among the top 500 supercomputers list in June 2004 [79].

5.2 Test Matrices

5.2.1 Model Grid Problems

For the scalability studies, we use matrices from finite difference $K \times K$ and $K \times K \times K$ model grids distributed across 1 through 64 processors. As shown in Tables
5.1 and 5.2, we increase $K$ with the number of processors to keep the arithmetic costs fixed per processor. Thus, the problem sizes grow along with the number of processors. The problem sizes given in Table 5.1 are used for the experiments on preconditioner construction and the entire linear solution process (this includes preconditioner construction and preconditioned CG iterations) studied in Chapter 6 and 8. These problem sizes are scaled based on the computational costs of numeric factorization of the sparse direct solver. Table 5.2 describes the problem sizes for the experiments on preconditioner application presented in Chapter 7. These problem sizes are scaled based on the computational cost of the triangular solution of the sparse direct solver.

<table>
<thead>
<tr>
<th>Processors ($P$)</th>
<th>2-dimensional grids</th>
<th>3-dimensional grids</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$K$</td>
<td>$N$</td>
</tr>
<tr>
<td></td>
<td>$(10^6)$</td>
<td>$(10^6)$</td>
</tr>
<tr>
<td>1</td>
<td>200</td>
<td>40000</td>
</tr>
<tr>
<td>2</td>
<td>250</td>
<td>62500</td>
</tr>
<tr>
<td>4</td>
<td>312</td>
<td>97344</td>
</tr>
<tr>
<td>8</td>
<td>392</td>
<td>153664</td>
</tr>
<tr>
<td>16</td>
<td>490</td>
<td>240110</td>
</tr>
<tr>
<td>32</td>
<td>618</td>
<td>386884</td>
</tr>
<tr>
<td>64</td>
<td>780</td>
<td>624100</td>
</tr>
</tbody>
</table>

Table 5.1. Description of sparse test matrices used for testing the scalability of preconditioner construction. $N$ is the matrix dimension, $|A|$ denotes the number of nonzeros in the matrix, and $|L|$ denotes number of nonzeros in $L$ where $A = LL^T$.

### 5.2.2 Sparse Matrices from Practical Applications

For a comparative study of the performance of parallel preconditioning schemes, we choose a suite of symmetric positive definite sparse matrices from the Matrix Market [87] and the University of Florida Sparse Matrix Collection [25]. Some indefinite matrices are modified by diagonal shifting by a small factor (1.0001) to assure positive definiteness. We use 1–16 processors to test our ICT factorization schemes with these matrices; for the largest problem, *augustus7*, we used 4 to 64 processors (a sparse direct solver requires at least 16 processors for this problem).
5.3 Preconditioning Schemes

In recent years, several parallel preconditioning techniques have been developed to take advantage of the large memory space and computational power of distributed memory multiprocessors. We tested the following preconditioners in addition to our ICT factorization schemes (ICT-SI and ICT-SSAI): symmetric SOR (SSOR) [11, 26], block Jacobi [11, 98] (Bjacobi), level-of-fill IC (ICF(0)) [60, 61], algebraic multigrid (AMG) [56], and sparse approximate inverse (Parasails) [21].

SSOR [26, 110] and Block Jacobi [98] preconditioners are obtained from PETSc version 2.1.6 [11, 26]. A few iterations of SSOR (a iterative method) are applied as a preconditioner for each CG iteration. The Block Jacobi preconditioner from PETSc employs a block diagonal data distribution to avoid any interprocessor communication. While the original Jacobi preconditioning is applied as an inverse of diagonal elements in the given matrix, the blocked version allows other classes of preconditioners or factors associated with each block diagonal submatrix; any sequential preconditioning algorithm can be plugged into each block. Hence, this scheme can be seen as two different preconditioning schemes for the entire matrix (Jacobi preconditioning) and the preconditioner for each diagonal block; it is thus a “two-level” preconditioning introduced by Kolotilina and Yeremin [64]. We test a sequential ICF(0) algorithm with natural ordering for the experiments.

Table 5.2. Description of sparse test matrices used for testing the scalability of preconditioner application. \(N\) is the matrix dimension, \(|A|\) denotes the number of nonzeroes in the matrix, and \(|L|\) denotes number of nonzeroes in \(L\) where \(A = LL^T\).
| Matrix           | \(N\)  | \(|A|\)     | Description                                                                 |
|------------------|--------|------------|-----------------------------------------------------------------------------|
| augustus7        | 1060864| 9318876    | Diffusion equation from 3D mesh                                             |
| augustus5        | 134144 | 645028     | Diffusion equation from 3D mesh                                             |
| bcsstk15         | 3948   | 117816     | Module of offshore platform                                                 |
| bcsstk16         | 4884   | 290378     | U.S. Army Corps of Engineers dam                                            |
| bcsstk25†        | 15439  | 252241     | Columbia Center (Seattle) 76-story skyscraper                               |
| bcsstk35†        | 30237  | 1450163    | Automobile seat frame and body attachment                                   |
| bcsstk36†        | 23052  | 1143140    | Automobile shock absorber assembly                                          |
| bcsstk39†        | 46772  | 2089294    | Shuttle solid rocket booster                                                |
| bodyy5           | 18589  | 129281     | From NASA, collected by Alex Pothen                                         |
| bodyy6           | 19366  | 1347481    | From NASA, collected by Alex Pothen                                         |
| cfd2             | 123440 | 3087898    | CFD: pressure matrix                                                       |
| dis120           | 14400  | 184204     | Visible surface computation                                                 |
| engine           | 143571 | 2424822    | Engine head, Linear tetrahedral elements                                    |
| nasasrb          | 54870  | 2677324    | Shuttle rocket booster                                                     |
| msc23052         | 23052  | 1154814    | Shock absorber                                                             |
| poisson3         | 16130  | 361668     | CFD: pressure matrix                                                       |
| qa8fin†          | 66127  | 1660579    | Finite element analysis for 3D acoustics                                   |
| qa8flk           | 66127  | 1660579    | Finite element analysis for 3D acoustics                                   |
| s1rmq4m1         | 5489   | 281111     | Cylindrical shell uniform triangle mesh                                    |
| s1rmt3m1         | 5489   | 219521     | Cylindrical shell uniform triangle mesh                                    |
| spa120           | 14400  | 184804     | Visible surface computation                                                 |
| xerox2000c1      | 6000   | 148050     | Colloidal analysis                                                         |

Table 5.3. Description of sparse matrices from practical applications. \(N\) is the matrix dimension, \(|A|\) is the number of nonzeros in the matrix and †indicates that the matrix is originally indefinite before the application of diagonal shifting.
The parallel level-of-fill IC (ICF(0)) preconditioner is from the BlockSolve95 package \[60, 61\]. The method exploits a parallel clique finding algorithm to improve cache performance and a multicoloring algorithm \[62\] to achieve a good parallel efficiency. Those graph theoretic techniques reduce the data dependencies for both preconditioner construction and its application.

The parallel algebraic multigrid (AMG) algorithm is from BoomerAMG \[56\], which is a part of the Hypre package \[37\]. Like BlockSolve95, the method applies parallel graph heuristics to build a coarse grid.

We use Parasails \[21, 37\] for the sparse approximate inverse preconditioner, which is also a part of the Hypre package. It is based on Frobenius norm minimization where each column is computed using least square minimization. Parasails takes advantage of the static memory allocation based on the nonzero structure of $A^i$ where $i$ is given as a user-defined parameter. To observe the performance for a range of the sparsity of the preconditioner, we used three different fill parameters (0, 1 as default, and 2) with the following dropping parameters (0.1 for pre-processing and 0.1 for post-processing) to measure the performance of preconditioning construction and preconditioning quality.

Our parallel ICT-SI and ICT-SSAI are computed with 3 different drop threshold conditions (0.001, 0.01 as default, and 0.1) with proper diagonal shifts (from 0.0 to 0.3). For ICT-SSAI scheme, we use Parasails \[21, 37\] for computing distributed diagonal submatrices with 1 level of fill and 0.1 for the both of dropping parameters. In addition, we measured the performance of a sparse direct solver from DSCPACK \[93\] that supports parallel Cholesky factorization and triangular solution.

The nested dissection ordering for fill-reduction and parallel data distribution is performed on ICT schemes and DSCPACK, and is regarded as a part of the preconditioner construction process. All preconditioners are used with the parallel CG implementation in PETSc \[11\].
Chapter 6

An Empirical Evaluation

of

Parallel Schemes to Construct Preconditioners

This chapter concerns the performance evaluation of various schemes for preconditioner construction. Our parallel ICT factorization schemes are evaluated in conjunction with other parallel preconditioners and a sparse direct solver. Preconditioner construction may comprise a major cost of sparse linear solution when it is applied to solve for one or a few right-hand side vectors; however if the same preconditioner is used to solve a long sequence of right-hand side vectors, then the costs for its construction can be better amortized. Evaluating the performance is, thus, necessary to guide the use of our ICT preconditioners.

Section 6.1 examines the scalability of preconditioner construction using model grid problems with scaled sizes. Section 6.2 reports on experiments with matrices from applications followed by a brief summary in Section 6.3. All the experiments were performed on the Jazz Linux Cluster [66] described earlier in Section 5.1.

6.1 Scalability of Preconditioner Construction

The time for preconditioner construction with scaled matrices of model grids is provided in Figure 6.1; ideal scalability should lead to a line parallel to the X-axis. This figure indicates that ICT-SSAI is more scalable than ICT-SI as claimed in the analysis in Chapter 4, whereas ICT-SI and Cholesky show significant performance loss for 64 processors. Yet, ICT-SSAI is not as efficient as block Jacobi and Parasails; however they are not robust as reported later in Chapter 8.

The AMG method behaves like ICT-SI and Cholesky for 3-dimensional problems. This inefficiency is caused by the coarsening algorithm of AMG which does not guarantee a reduction of the matrix size by half (or a fixed constant) in every coarsening step (unlike geometric multigrid). According to [17, 56], the coarsening quality of AMG depends both on sparsity pattern and numerical values of nonzero elements in the coefficient matrix. Therefore, the algorithm may store a large number of restricted submatrices (Schur complements) and result in a large number of arithmetic operations as well as storage space. The method thus fails to achieve the scalable performance as geometric multigrid can deliver for model grid problems.
Fig. 6.1. The cost of preconditioner construction for 2-dimensional grids (top) and 3-dimensional grids (bottom) on the Jazz Linux cluster.
Fig. 6.2. Time for constructing preconditioners on a suite of 21 sparse matrices with 1 (top), 2, 4, 8, and 16 (bottom) processors. The values in the Y-axis represent the performance relative to the sequential sparse direct solver (Cholesky).
6.2 Performance on Sparse Matrices from Applications

We report on the performance of preconditioner construction on sparse matrices listed in Table 5.3. We first plot the performance of all matrices except augustus7 in Figure 6.2 as stacked bars; each color represents the performance of a single matrix, and is ordered as in Table 5.3. The values on the Y-axis express the time relative to that for computing sparse Cholesky factorization using a single processor on each matrix instance. Our methods are not showing very good parallel speedups compared to ICF and Parasails primarily because of the complicated data structures such as the supernodal tree and higher interprocessor communication. Another interesting aspect is that ICT-SI is more efficient than ICT-SSAI; this differs from our analysis in Chapter 4. We conjecture that the ordering and subsequent symbolic factorization create relatively small supernodal matrices toward top levels of the tree. Consequently, the parallel sparse matrix algorithms associated with such smaller supernodal matrices cannot obtain high efficiencies. On the other hand, these supernodal matrices are small enough to exploit dense matrix computation while avoiding indirect memory access and other integer operations to construct a sparse matrix data structure. AMG behaves like ICT schemes due to the intricacies in its data structures for the implementation of graph heuristics to support irregular grids.

We report on the parallel speedups for augustus7 in Figure 6.3 to observe that our ICT factorization schemes achieve reasonable speedups; we expect such speedups for large matrices where the computational costs dominate over the communication cost. More specifically, ICT-SSAI (ICT-SI) achieve 14 (4) fold speedup on increasing processors from 4 to 64. This indicates that ICT-SSAI can potentially outperform ICT-SI for the preconditioner construction for large matrices.

6.3 Summary

Our experiments with scaled model problems indicate that the scalability of our ICT-SSAI scheme is very close to the ideal. Experiments with a suite of matrices from practical applications reveal that ICT-SI is slightly more efficient than ICT-SSAI on small matrices; however, both ICT schemes are not able to achieve good speedups for small matrices. The experiment with augustus7 (the largest matrix in our test suite) shows that ICT-SSAI achieves a fourteen-fold speedup from 4 to 64 processors, while ICT-SI factorization only attains a four-fold speedup. This result confirms the advantage of ICT-SSAI over ICT-SI for sufficiently large matrices and is consistent with the results of our analysis in Chapter 4.
Fig. 6.3. Time for preconditioner construction for augustus7 using 4–64 processors.
Chapter 7

Latency Tolerant Schemes
for
Applying ICT Preconditioners

This chapter considers the parallel application of an incomplete Cholesky (IC) preconditioner, \( \hat{L} \), to accelerate Conjugate Gradients (CG). Application using sparse triangular solution using substitution is straightforward on uniprocessors. However, on distributed memory multiprocessors, a triangular solution with parallel substitution is inefficient due to the relatively high latencies of interprocessor communication. For a traditional parallel substitution scheme, henceforth the TS scheme, the latency cost grows with the dimension of the sparse matrix [94]. Thus, IC preconditioning has long been considered to be unsuitable for parallel CG method. This problem led to the popularity of a relatively new class of sparse approximate inverse preconditioners [14, 21, 50, 65]. Our methods exploit matrix inversion schemes that are applied selectively to submatrices in the incomplete factor, \( \hat{L} \). The original idea called Selective Inversion (SI) [92] was developed for parallel sparse direct solvers when the matrix is factored once, and used to solve for a series of right-hand side vectors. We now focus on an implementation with an enhancement for applying our incomplete factor preconditioners from ICT-SI and ICT-SSAI schemes.

The rest of chapter is organized as follows. Section 7.1 describes the main idea of the SI scheme to replace parallel substitution by matrix-vector multiplication. Section 7.2 shows how the SI scheme is adapted for incomplete factorization, and considers a new data mapping scheme to further reduce communication overheads. Section 7.3 contains an analysis of the communication costs of our schemes for model grid problems. Section 7.4 provides an empirical study of our parallel schemes on the IBM SP. Finally, Section 7.5 contains a summary of this chapter.

7.1 On Selectively Replacing Parallel Substitution by Parallel Matrix-Vector Multiplication

Let \( A = LL^T \), where \( L \) is the complete Cholesky factor computed by a typical sparse Cholesky factorization. The factorization is computed by two different steps:
symbolic and numeric factorization. As explained in Chapter 2, symbolic factorization constructs a supernodal tree for Cholesky factor $L$ that determines the algorithmic sequence for numeric factorization and the nonzero pattern of $L$. At the same time, the supernodal tree partitions $L$ into a set of small dense matrix blocks. On distributed memory multiprocessors, the data distribution of $L$ is determined by the supernodal tree where two different types of parallelism are exploited: task-parallelism with respect to the local-phase subtrees and data parallelism with respect to supernodes above the local-phase subtrees [38].

The triangular solution is directed by either post-order (forward solution) or pre-order (backward solution) traversal of the supernodal tree. The dense submatrix associated with a supernode is used to compute some components of the solution vector and vector updates to the right-hand side associated with ancestor supernodes. More precisely, the system at a supernode is of the form:

$$
\begin{bmatrix}
L_{11} & L_{21} \\
L_{21} & &
\end{bmatrix}
\begin{bmatrix}
x_1 \\
b_2 
\end{bmatrix} =
\begin{bmatrix}
b_1 \\
b_2 
\end{bmatrix}.
$$

The supernodal matrix composed of $L_{11}$ and $L_{21}$ is typically trapezoidal in shape. The triangular system $L_{11}x_1 = b_1$ must first be solved to obtain $x_1$. Next $x_1$ is used to compute $b_2 \leftarrow b_2 - L_{21}x_1$. The vector $x_1$ provides components of the solution vector associated with columns in the supernode. The vector $b_2$ is used to update components of the right-hand side vector associated with ancestor supernodes; the updates are propagated through computations at successive immediate ancestors or parent supernodes.

For a supernode located at higher levels of the tree, the triangular solution, $L_{11}x_1 = b_1$, implemented by parallel substitution algorithm suffers from a large communication latency. To alleviate the latency problem, Raghavan earlier proposed the Selective Inversion scheme that applies a matrix inversion for $L_{11}$ only for distributed supernodes during factorization [92]. As a result, the triangular solution, $L_{11}x_1 = b_1$, is replaced by a dense matrix-vector multiplication, $L_{11}^{-1}b_1 = x_1$. The SI scheme involves extra computational cost for inversion of some submatrices, but [92] reports that the additional cost for performing selective inversion for complete Cholesky factor is no more than 6% of the factorization cost for the model grid problems. Hence, the cost of the selective inversion can be offset by around 10–20 repeated triangular solutions [92].

The SI scheme can be applied for our parallel ICT factorization with a modification of the data structure (henceforth ICT-SI and ICT-SSAI) as described earlier in Chapter 4. The main difference between ICT-SI/ICT-SSAI and complete Cholesky factorization is that both $\hat{L}_{11}$ and $\hat{L}_{11}$ no longer exhibit dense structure due to dropping elements.
Fig. 7.1. Distributed forward solution at a supernode (A) with selective inversion and the one-way data mapping (DM-1) using 4 processors.
7.2 One-way and Two-way Data Mapping for Reducing Communication Latency Costs

We now consider the data mapping scheme for reducing communication costs of distributed sparse triangular solution. In the original SI algorithm [92] available in DSCPACK 1.0 [93], the distributed supernode is partitioned by 1-dimensional column-block-cyclic mapping, which is suitable for a fan-in scheme by a distributed multifrontal scheme for numeric factorization. Figure 7.1 illustrates the computations associated with a supernode (labeled A) for the forward solution using SI. Computations for backward solution are similar and not described separately. In a supernode (A), the trapezoidal matrix is assigned to 4 processors (0,1,2,3). Henceforth, we use DM-1 to denote such a data mapping and refer to it as one-way data mapping. The children supernodes of (A), (B) and (C), are mapped to processors (0,1) and (2,3) respectively as described in Figure 7.1. The components of the right hand side vectors \(b_1, b_2\) for supernode (A) are gathered from the updated portion of (B) and (C). This gather is performed by a group reduction operation such as \texttt{MPI_AllReduce} in MPI [48]; we refer this operation to \texttt{AllReduceSum}. Next, at supernode (A) \(x_1 \leftarrow L_{11}^{-1}b_1\) is computed on processors (0,1,2,3); we denote this by \texttt{ParallelTriangularMatVec}. As illustrated in Figure 7.1, this operation produces \(x_1\) scattered across the four processors. A second gather is needed to accumulate \(x_1\) and therefore the second \texttt{AllReduceSum} operation is performed. Finally, a parallel matrix vector multiplication (\texttt{ParallelMatVec}) is used to compute \(b_2 \leftarrow b_2 - L_{21} \times x_1\). The vector \(b_2\) is then available in a scattered form for computations at the parent of supernode (A) where the same process is repeated using the 8 processors assigned to (A) and its sibling. Thus, at each supernode, the communication latency costs are those associated with the number of messages required to perform two \texttt{AllReduceSum} operations. The algorithm is illustrated in Figure 7.2.

For the ICT-SI and ICT-SSAI methods, an incomplete factor \(\hat{L}\) is replaced for \(L\). Regardless of the incompleteness of \(\hat{L}\), the communication latency cost is the same as that for the (complete) Cholesky factor, \(L\).

We now describe a new two-way data mapping scheme, henceforth DM-2, which requires only a single \texttt{AllReduceSum} operation at a supernode. In this scheme, \(\hat{L}_{11}^{-1}\), the triangular part of the trapezoidal submatrix at (A) is mapped in “row-block-cyclic” form while the rectangular part \(\hat{L}_{21}\) is mapped in “column-block-cyclic” form. The block sizes for both mapping are kept the same. As in DM-1, \(b_1\) is first gathered from
Algorithm 1 (DM-1)

1: Let node $i$ involves $p_i$ processors and let $b = (b_1, b_2)^T$
2: for each distributed supernode $i$ (traversed in post-order on the tree) do
3:   $\text{All\_Reduce\_Sum}(b, p_i)$
4:   $x_1 \leftarrow \text{Parallel\_Triangular\_MatVec}(L_{11}, b_1, p_i)$
5:   $\text{All\_Reduce\_Sum}(x_1, p_i)$
6:   $b_2 \leftarrow b_2 \cdot \text{Parallel\_MatVec}(L_{21}, x_1, p_i)$
7: end for

Fig. 7.2. Triangular solution with 1-way mapping (DM-1).

children supernodes (B) and (C) using an $\text{All\_Reduce\_Sum}$ operation and a parallel matrix vector multiplication ($\text{Parallel\_Triangular\_MatVec}$) is used to compute $x_1 = \hat{L}^{-1}_{11} \times b_1$. With the “column-block-cyclic” mapping of $\hat{L}_{21}$, $b_2 = b_2 - \hat{L}_{21} \times x_1$ can be computed without the second gather operation required in DM-1; the components of $x_1$ are located on processors (0,1,2,3) in the manner required for $\text{Parallel\_MatVec}$. Thus, in DM-2, only one $\text{All\_Reduce\_Sum}$ operation is needed at each supernode. Consequently, this modification halves the number of messages sent at a processor and thus the latency costs. The algorithm is shown in Figure 7.3; Figure 7.4 shows the two-way data mapping and computational steps in DM-2.

7.3 Analysis of Communication Costs

We now summarize our analysis of the communication overhead for DM-1 and DM-2 for sparse matrices of model two and three dimensional grids. The two dimensional model problem is associated with a $K \times K$ five-point finite difference grid and results in a sparse matrix of rank $N = K^2$. For the three-dimensional grid, the sparse matrix of dimension $N = K^3$ corresponds to a $K \times K \times K$ seven-point finite-difference grid. We assume such model problems are being solved by $P$ processors using a nested-dissection fill-reducing ordering.

Once again, we assume that the multiprocessor has the following model of communication costs. The cost of sending $\sigma$ words in a single message is: $t_s + \sigma t_w$, where $t_s$ is the startup (or latency) and the $t_w$ is the per word transfer cost. The latency cost $t_s$ is typically orders of magnitude larger than $t_w$ (and the time for a single floating point operation). We also assume that $\text{All\_Reduce\_Sum}$ with $P$ processors can be implemented with $\log_2 P$ messages using a multidimensional hypercube type algorithm. The latency of communication is proportional to the number of messages sent, and the data transfer costs are proportional to the volume of data exchanged.
Fig. 7.3. Distributed forward solution at a supernode (A) with selective inversion and the two-way data mapping (DM-2) using 4 processors.
Algorithm 2 (DM-2)

1: Let node \( i \) involves \( p_i \) processors and let \( b = (b_1, b_2)^T \)
2: for each distributed supernode \( i \) (traversed in post-order on the tree) do
3: \( \text{AllReduce}_\text{Sum}(b, p_i) \)
4: \( x_1 \leftarrow \text{Parallel}_\text{Triangular}_\text{MatVec}(L_{11}^{-1}, b_1, p_i) \)
5: \( b_2 \leftarrow b_2 - \text{Parallel}_\text{MatVec}( L_{21}, x_1, p_i) \)
6: end for

Fig. 7.4. Triangular solution with 2-way mapping (DM-2).

Lemma 7.1. Consider the sparse linear system of dimension \( N \) associated with the model five-point, finite-difference \( K \times K \) grid \((N = K^2)\) and the seven-point \( K \times K \times K \) grid \((N = K^3)\). Assume a nested dissection ordering is used for fill-reduction and triangular solution is performed using \( P \) processors. With the TS scheme, the number of messages sent by a single processor in every triangular solution grows as \( O(\sqrt{K/P}) \) for the 2 dimensional grid and \( O\left(\frac{K^2}{P^{2/3}}\right) \) for the 3 dimensional grid.

Proof. Let \( C_2(K, P) \) be the number of messages required for the two-dimensional \( K \times K \) grid. The expression is computed by solving the recurrence:

\[
C_2(K, P) = 2\frac{K}{P} + C_2\left(\frac{K}{2}, \frac{P}{2}\right).
\]

The recurrence relation is obtained by observing that the root node of the compute-tree involves the solution of a triangular linear system of dimension \( K \) using \( P \) processors. The next level has two nodes, each of which involves the solution of a triangular system of dimension \( \frac{K}{2} \) using \( \frac{P}{2} \) processors. The total number of non-overlapping messages over these two levels is \( 2\frac{K}{P} \). After that there are four independent subtrees, each corresponds to a grid of size \( \frac{K}{2} \) using \( \frac{P}{4} \) processors. However, communication within each subtree is independent of that in other trees.

The recurrence is solved by observing that:

\[
C_2(K, P) = 2\frac{K}{P} \{1 + 2 + 2^2 + \cdots + 2^{\log_4 P - 2}\} + C_2\left(\frac{K}{2^{\log_4 P - 1}}, 1\right).
\]

In the expression above \( C_2\left(\frac{K}{2^{\log_4 P - 1}}, 1\right) = 0 \) as it denotes a local phase computation; the summation simplifies to \( 2\frac{K}{P} \times 2^{\log_4 P} \), which leads to \( O\left(\frac{K}{\sqrt{P}}\right) \) complexity. The
three-dimensional case can be analyzed similarly; let $C_3(K, P)$ be the number of non-overlapping messages required. Now the appropriate recurrence relation is:

$$C_3(K, P) = 3 \frac{K^2}{P} + C_3\left(\frac{K}{2}, \frac{P}{2^3}\right).$$

This recurrence yields $O\left(\frac{K^2}{P^{2/3}}\right)$. □

**Lemma 7.2.** Consider the sparse linear system of dimension $N$ associated with the model five-point, finite-difference 2 dimensional $K \times K$ grid ($N = K^2$) and the seven-point 3 dimensional $K \times K \times K$ grid ($N = K^3$). Assume that an optimal nested dissection ordering is used for fill-reduction and triangular solution is performed using $P$ processors. With the DM-1 scheme, the number of the messages sent at a processor is no more than $(\log_2 P)^2 + \log_2 P$. With the DM-2 scheme, the number of message is no more than $(\log_2 P)^2 + \log_2 P$.

**Proof.** A processor participates in distributed-phase computations on a path from the root of its local subtree to the root of the entire supernodal tree. The number of supernodes deployed along the path is exactly $\log_2 P$. At a supernode with $p$ processors, a processor requires $\log_2 p$ messages to perform a single *AllReduceSum* operation. Starting from the root supernode with $P$ processors, the number of processors participating in the computation is halved in every level. Hence the total number of messages sent by a processor with DM-1 scheme is given by:

$$2(\log_2 P) + 2(\log_2 P - 1) + 2(\log_2 P - 2) + \cdots + 4 + 2 = (\log_2 P)^2 + \log_2 P.$$  

The number of *AllReduceSum* for DM-2 scheme is exactly one half to lead a total of $(\log_2 P)^2 + \log_2 P$ messages. □

**Lemma 7.3.** Consider the sparse linear system of dimension $N$ associated with the model five-point finite-difference $K \times K$ grid ($N = K^2$) and the seven-point $K \times K \times K$ grid ($N = K^3$). Assume that an optimal nested dissection ordering is used for fill-reduction and the triangular solution is performed using $P$ processors with the DM-2 scheme. The number of elements sent by a processor is no more than $3K(\log_2 P)$ for 2-dimensional grids and no more than $\frac{7}{3}K^2(\log_2 P)$ for 3-dimensional grids.

**Proof.** In the root supernode, the size of triangular submatrix is equal to the size of separator of the grid. Therefore, it is $K$ for the two dimensional grid and $K^2$ for the three dimensional grid.
**2D Grids.** The size of separator for 2-dimensional grid is understood by the diagram in Figure 7.5 where two types of separators size of $K$ and $K/2$ divide the grid into four subgrids. In each subgrid, the new separators divide the grid into four, again.

The number of processors participating in a communication step is halved at each level, and hence the number of messages sent by a single \texttt{AllReduceSum} operation is reduced by 1. Therefore, the total amount of data sent by a processor with the DM-2 scheme is:

$$Volume(K, P) = Volume(K/2, P/4) + K \times \log_2 P + (K/2) \times (\log_2 P - 1)$$

It is equivalent to:

$$Volume = K \times \left\{ \log_2 P + (1/2)(\log_2 P - 1) + (1/2)(\log_2 P - 2) + (1/4)(\log_2 P - 3) + (1/4)(\log_2 P - 4) + (1/8)(\log_2 P - 5) + \ldots + 1/2(\log_4 P)(2) + 1/2(\log_4 P + 1)(1) \right\}$$

This yields:

$$Volume = K \times \sum_{i=1}^{\log_4 P} (1/2)^i (3 \log_2 P - 6(i - 1) - 1)$$

$$<K \times \sum_{i=1}^{\log_4 P} (1/2)^i (3 \log_2 P)$$

$$= K((3 \log_2 P) \times (1 - (1/2)^{\log_4 P})).$$
For large $P$, $(1/2)^{\log_4 P} \approx 0$. Thus, we have:

$$\text{Volume} < K(3 \log_2 P).$$

The DM-1 scheme requires exactly twice the communication steps in DI-2. Thus, the amount of communication volume is bounded by $6K \log_2 P$.

\[3D \text{ Grids.}\] The size of separators for 3-dimensional grid is shown in Figure 7.6 where three types of separators size of $K \times K$, $K/2 \times K$, and $K/2 \times K/2$ divide the grid into eight subgrids. In each subgrid, new separators further divide the grid into eight, and so on. As in the 2 dimensional case, the number of processors involved in computing a supernode is reduced by half at every level of the tree. Now we have the general formula:

$$\text{Volume}(K, P) = \text{Volume}(K/2, P/8) + K^2 \times \log_2 P$$

$$+ (K^2/2) \times (\log_2 P - 1) + (K^2/4) \times (\log_2 P - 1)$$

It is equivalent to:

$$\text{Volume} = K^2 \times \{\log_2 P + (1/2)(\log_2 P - 1) + (1/4)(\log_2 P - 2) +$$

$$(1/4)(\log_2 P - 3) + (1/8)(\log_2 P - 4) + (1/16)(\log_2 P - 5) +$$_{\cdots} +$$

$$(1/2)^{\log_8 P} (4) + (1/2)^{\log_8 P+1} (2) + (1/2)^{\log_8 P+2} (1)\}.$$
This yields:

\[ Volume = K^2 \times \prod_{i=1}^{\log_8 P} (1/4)^i (7 \log_2 P - 21(i - 1) - 4) \]

\[ < K^2 \times \prod_{i=1}^{\log_8 P} (1/4)^i (7 \log_2 P) \]

\[ = K^2 ((7 \log_2 P) \times 1/3(1 - (1/4)^{\log_8 P})). \]

For large \( P \), \((1/4)^{\log_8 P} \approx 0\). Thus, we have:

\[ Volume < K^2 \left( \frac{7}{3} \log_2 P \right). \]

The DM-1 scheme requires exactly twice of the communication steps of DM-2. Thus, the amount of communication volume is bounded by \( \frac{14}{3} K^2 \log_2 P \). \( \square \)

This lemma indicates that the communication volume grows with the size of grid and the number of processors. In terms of the matrix dimension \( (N) \), the growth of the communication volume is \( O(N^{1/2} \log_2 P) \) for two-dimensional grids and \( O(N^{2/3} \log_2 P) \) for three-dimensional grids. For large problems, the matrix dimension seems to be the main factor for the inefficiency of the triangular solution. However, the communication latency is more problematic than the communication bandwidth because of the recent developments in interconnection networks with higher bandwidth. Therefore, our scheme allows a practical implementation of CG with incomplete factor preconditioner. Observe that our methods are also applicable for complete Cholesky factors.

7.4 Empirical Study: Performance of Methods to Apply ICT Preconditioners

We report on the performance of 1-way data mapping Selective Inversion (DM-1), two way data mapping Selective Inversion (DM-2) and the traditional substitution method (TS) using drop-threshold incomplete Cholesky factor preconditioners (henceforth ICT-SI-1, ICT-SI-2, and ICT-TS) on 2- and 3-dimensional finite difference model grid problems. Our implementation was based on DSCPACK1.0 [93]. For the sake of completeness, we also provide results on the performance of the Parasails developed by Chow [21]. Parasails is a representative method from the class of parallel sparse approximate inverse preconditioners. We used different fill parameters to control the sparsity of the preconditioner to hold the same amounts of nonzero elements as our parallel ICT scheme described in Chapter 3 and 4. With all four preconditioning methods,
we used the parallel CG implementation with the default parameters from the PETSc package [11]. Our program is executed on the IBM SP at NERSC. As shown in Table 5.2 in Chapter 5, we increase the size of problems so that the amount of the arithmetic operations of a single triangular solution per processor is kept fixed as the number of processors is increased. For all methods, we report results with the number of nonzeroes in the preconditioner $|\tilde{L}|$ relative to $|L|$ set (as closely as possible) to .2 and .5 for 2 dimensional problems, and at .2 and .4 for 3 dimensional problems. This was achieved by experimenting with various values of the drop threshold parameters for IC and the levels of fill for Parasails. Despite our efforts, there are some minor variations in the fill because of the nature of the underlying algorithms; however, they have negligible impact on the reported performance measures.

7.4.1 Performance of Preconditioned CG

The performance of parallel preconditioned CG depends both on the quality of preconditioning and the time to apply the preconditioner at each iteration. The former can dramatically affect the total number of iterations to convergence while the latter can increase with the number of processors. Thus, both aspects can significantly affect the scaling of our preconditioned CG application. Our main focus is on scalability of different schemes for applying the preconditioner. Assessing the quality of preconditioning is beyond the scope of this experiment, but we do provide some limited results indicating intrinsic differences between the class of ICT methods and sparse approximate inverses.

Figure 7.7 shows the number of iterations for convergence of CG with ICT and Parasails. In terms of the number of iterations, ICT is better than the Parasails for 2-dimensional problems. We conjecture that dropping strategies and other techniques can be incorporated with our ICT method to decrease further the number of iterations [9, 51, 70, 77, 98]. For the 3-dimensional problems, the convergence from Parasails is better for larger problems. This is partly because the 3-dimensional problems are better conditioned than the 2-dimensional problems. Figure 7.8 provides the total time over all preconditioned iterations using Parasails or TS, ICT-SI-1, and ICT-SI-2 with IC. This quantity is the product of the time per iteration and the number of iterations to convergence. Parasails does better than ICT-SI-2 for the two-dimensional problem using 64 processors when the preconditioner is sparse (at 20% fill). ICT-SI-2 is significantly faster than Parasails using 64 processors for the two-dimensional problem at 50% fill and the three-dimensional problems at 20% and 40% fill in the preconditioner. This is despite the lower number of iterations incurred by Parasails and is primarily from
Fig. 7.7. Number of iterations of the four preconditioning schemes. The size of $|\tilde{L}|/|L| = 0.2$ (top left) and 0.5 (top right) for 2-dimensional grids and $|\tilde{L}|/|L| = 0.2$ (bottom left), and 0.4 (bottom right) for 3-dimensional grids.
Fig. 7.8. Total time for applying the preconditioner over all iterations. $|\frac{L}{L_0}| = 0.2$ (top left) and $0.5$ (top right) for 2-dimensional grids, and, $|\frac{L}{L_0}| = 0.2$ (bottom left), and $0.4$ (bottom right) for 3-dimensional grids.
the improved performance of ICT-SI-2 for the preconditioning. Our results clearly indicate that ICT-SI-2 makes parallel preconditioning with incomplete factors a viable and effective method.

We now discuss the performance gains achieved using the new 2-way mapping and ICT-SI-2. Figure 7.9 shows the time to apply the preconditioner in a single CG iteration, i.e. a single triangular solution step for IC and a parallel matrix vector multiplication for Parasails. Ideally, the scaled performance should not result in any increase in time with the number of processors and the problem size. As expected, the performance of ICT-TS is the worst; ICT-SI-2 performs better than ICT-SI-1 with larger numbers of processors. The relative performance gain for 3-dimensional problems is smaller because the incomplete factor for such problems is relatively denser for the same matrix dimension. The performance of both SI methods is comparable to that of Parasails even though the latter uses only a single distributed matrix vector product; recall that SI methods use a sequence of smaller distributed matrix-vector products on the tree. For the 3-dimensional problems, Parasails is slower than our scheme even though a matrix vector multiplication should be more efficient than the tree-based procedure in ICT-SI-1 and ICT-SI-2.

7.4.2 Communication Costs

Figure 7.10 shows the average and maximum number of messages sent at a processor per preconditioning step. The message counts are independent of the sparsity of the preconditioner for ICT-TS, ICT-SI-1, and ICT-SI-2, while they grow with the density of the preconditioner for Parasails. We have thus reported messages counts with the least dense preconditioners, i.e. at 20% fill with the least number of observed messages for the Parasails. Observe that for ICT-SI-1 and ICT-SI-2 these message counts are independent of matrix dimension; they are indeed a slow growing function of the number of processors as indicated in Lemma 4.1 an 4.2. As expected, the number of messages for ICT-SI-2 is approximately half those for ICT-SI-1. The maximum number of messages sent over all processors does not vary from the average number for ICT-SI-1 and ICT-SI-2 because of the underlying structured tree-based algorithm. However, in Parasails, the maximum number of messages sent at a processor can be significantly higher than the average. This is primarily because Parasails uses a simple row block data distribution where one processor may need to communicate with all other processors to access right-hand-side elements corresponding to its portion of the preconditioner. Thus our ICT-SI-2 scheme has a more balanced communication pattern. Even with the sparsest preconditioners (at 20% fill), the maximum number of messages sent at a processor in ICT-SI-2 is lower than that for Parasails.
Fig. 7.9. Time (in seconds) to apply the preconditioner in one CG iteration. $\frac{|L|}{|Z|} = 0.2$ (top left) and 0.5 (top right) for 2-dimensional grids, and $\frac{|L|}{|Z|} = 0.2$ (bottom left), and 0.4 (bottom right) for 3-dimensional grids.
Fig. 7.10. The number of messages sent by a preconditioner in each CG iteration. The average (left) and maximum (right) number of messages sent per processor for preconditioning in each CG iteration for 2-dimensional grids (top) and 3-dimensional grids (bottom). The message counts for the IC preconditioners do not change with fill but the message counts for Parasails grow with increasing values of $\frac{|L|}{|L|}$ (shown at 20% fill).
7.5 Summary

We have demonstrated how the SI scheme can be used to apply ICT preconditioner and we have developed a data-mapping scheme that can halve the communication costs of the original scheme. Our analysis shows that the number of messages of the SI scheme is independent of the size of matrix ($O((\log_2 P)^2)$), while traditional substitution requires $f (O(K\sqrt{\frac{P}{K}})$ messages for two-dimensional grids ( $O(K^2\frac{P}{K^{2/3}}$) messages for three-dimensional grids). Our empirical results indicate that for model grid problems the performance of CG preconditioned with ICT-SI is comparable to that of the CG with Parasails preconditioner.
Chapter 8

On the Performance
of
Hybrid ICT-SI/ICT-SSAI Solvers

In this chapter, we study the performance of linear system solution using our hybrid solvers with parallel ICT-SI and ICT-SSAI preconditioners for CG. More specifically, we evaluate these schemes based on total solution time, number of iterations, and memory usage. We also include comparisons with other parallel sparse linear solvers. Section 8.1 provides a brief description of the conditions under which our experiments were carried out. Section 8.2 examines the scalability of our schemes using scaled sparse matrices that arise from model two and three dimensional grids. Section 8.3 contains results for a suite of sparse matrices from practical applications, and Section 8.4 contains a brief summary of this chapter.

8.1 Test Conditions

The experiments were carried out under conditions described earlier in Chapter 5. The sparse grid problems are the same as those listed in Table 5.1; we, again, increase problem size with number of processors so that each processor in a parallel sparse direct solver performs the same amount of arithmetic work. Our test code is written in C and uses PETSc [11] for parallel CG iterations; other parallel preconditioners are obtained from Hypre (for Parasails and AMG), PETSc (for SSOR and Block Jacobi) and BlockSolve95 (for ICF(0)). The preconditioned CG iteration terminates when it either diverges or satisfies convergence criterion, or when preconditioner construction fails. In particular, the convergence of CG iteration is determined by a relative residual of the unpreconditioned system which satisfies the condition below:

\[
\frac{\|b - Ax\|}{\|b\|} < 10^{-8}.
\]

All the test codes are executed on Jazz Linux Cluster [66] at Argonne National Laboratory.
8.2 Scalability Results

We first provide the total time for solution in Figure 8.1. CG preconditioned by SOR, Block Jacobi, ICF(0) and Parasails show scalable performance, while the performance of ICT schemes and AMG on 3-dimensional grids degrade for large numbers of processors. The performance of our ICT schemes suffers from the higher cost of preconditioner construction as mentioned earlier in Chapter 5. The poor performance of AMG on 3 dimensional grids is due to the complex nature of its coarsening process.

Next, Figure 8.2 shows the number of iterations required; our ICT schemes require less iterations than other preconditioning schemes except AMG, which reaches convergence in a few iterations. ICT-SI requires less iterations than ICT-SSAI, but ICT-SSAI requires less time than ICT-SI because it incurs lower costs for preconditioner construction. The small numbers of iterations of ICT-SI and ICT-SSAI result in a scalable hybrid solver. Figure 8.3 shows the time for PCCG iteration where time for preconditioner construction is excluded from total solution time. This indicates that our schemes achieve scalable preconditioner application comparable to other preconditioning techniques on both 2 and 3 dimensional grids.

Figures 8.4 and 8.5 demonstrate how preconditioned CG schemes can save memory for the solution compared to a sparse direct solver; values in the Y-axis represent the number of nonzeroes in a preconditioner relative to the original coefficient matrix. The difference in the memory usage between ICT-SI and ICT-SSAI is less than 1% and we therefore provide the results for the ICT-SI scheme labeled as “ICT” in these figures. An interesting aspect is that AMG requires a larger amount of space to compute a matrix for 3-dimensional grids than the other preconditioning schemes (depicted in Figure 8.5); its memory usage also increases with the number of processors. Therefore, AMG is not a memory scalable preconditioner while other preconditioners such as SSOR saves a substantial amount of memory. SSOR has the best memory scalability because it requires no extra memory beyond the space for coefficient matrix. Block Jacobi, ICF(0), and Parasails require some extra memory, but no more than the size of coefficient matrix; the efficient memory use of Parasails stems from dropping elements before and after computing a preconditioner. Memory usage of ICT schemes grows slightly with the number of processors, but it is not as problematic as for AMG. Observe that a sparse direct solver on 64 processors requires a space more than 100 times of storage for a coefficient matrix for matrices from 3-dimensional grids.

8.3 Reliability and Quality of Hybrid Linear Solvers on Sparse Matrices from Applications

We investigate the performance of the various solvers on sparse SPD systems available in well-known collections from Matrix Market [87] and the University of Florida
Fig. 8.1. The total time for the solution of the 2 dimensional (top) and 3 dimensional grids (bottom) on the Jazz Linux cluster.
Fig. 8.2. The number of iterations for the solution of the 2 dimensional (top) and 3 dimensional grids (bottom).
Fig. 8.3. The time for PCCG iteration on the 2 dimensional (top) and 3 dimensional grids (bottom).
Fig. 8.4. The ratio of number of nonzero elements in preconditioners relative to the number of nonzero elements in the coefficient matrix from 2 dimensional grids on 1 (upper left), 4 (upper right), 16 (lower left), and 64 (lower right) processors.
Fig. 8.5. The ratio of number of nonzero elements in preconditioners relative to the number of nonzero elements in the coefficient matrix from 3 dimensional grids on 1 (upper left), 4 (upper right), 16 (lower left), and 64 (lower right) processors.
Sparse Matrix Collection [25]. The details of matrices are shown on Table 5.3 in Chapter 5. The experiments were conducted on the Jazz Linux cluster [66] using 1 to 16 processors for all matrices except \textbf{augustus7} (4–64 processors) whose dimension is in excess of one million.

We first examine the quality of preconditioning based on the percentage of successes for solvers and the number of iterations to reach convergence for each instance. A scheme is a success if it converges within 600 or 2000 iterations. The percentage of successful instances is derived from the total of 105 trials with 5 different processor sizes (1, 2, 4, 8 and 16) and 21 matrices (all matrices except augustus7) for each scheme. The results in Figure 8.6 indicate that ICT-SI and ICT-SSAI reach convergence within 600 iterations in more than 90% of instances while other schemes fail to converge with many instances. Allowing a higher maximum iteration limit enables the other preconditioning schemes to reach convergence on more instances (except for AMG).

| Matrix    | \(N\)   | \(|A|\)       | Description                                           |
|-----------|---------|---------------|-------------------------------------------------------|
| bodyy5    | 18589   | 129281        | From NASA, collected by Alex Pothen                   |
| bodyy6    | 19366   | 1347481       | From NASA, collected by Alex Pothen                   |
| dis120    | 14400   | 184204        | Visible surface computation                           |
| poisson3  | 16130   | 361668        | CFD: pressure matrix                                  |
| qa8fm†    | 66127   | 1660579       | Finite element analysis for 3D acoustics              |
| qa8fk     | 66127   | 1660579       | Finite element analysis for 3D acoustics              |
| s1rmq4m1  | 5489    | 281111        | Cylindrical shell uniform triangle mesh               |
| s1rmt3m1  | 5489    | 219521        | Cylindrical shell uniform triangle mesh               |
| spa120    | 14400   | 184804        | Visible surface computation                           |
| xerox2000c1 | 6000    | 148050        | Colloidal analysis                                    |

Table 8.1. Description of ten test matrices that were successfully solved by all the preconditioned CG schemes. \(N\) is the matrix dimension, \(|A|\) is the number of nonzeroes in the matrix and †indicates that the matrix is originally indefinite.

We also present the number of iterations as stacked bars in Figure 8.7, where each color stands for a single matrix instance. These 10 matrices listed in Table 8.1 are solved by all the preconditioning schemes within 2000 iterations and the other 3 matrices in Table 8.2 are solved by all the preconditioners except AMG. The results show that ICT-SI and ICT-SSAI require fewer iterations than any other schemes except AMG. The latter
can solve only a small set of problems, but, in these instances, convergence is achieved with relatively fewer iterations. The comparison based on the 13 matrices indicates that ICT schemes require approximately 25% of iterations to solve all 13 matrices compared to other preconditioning techniques. Another interesting aspect is that ICT-SI requires fewer iterations than ICT-SSAI because explicit matrix inversion tends to produce more accurate approximations than sparse approximate inversion.

Next, we study the total time of preconditioned CG solution for a single right-hand side vector including the time to construct the preconditioner and to apply it to CG iterations. Figure 8.8 (8.9) summarizes the relative solution time of each method to the solution time of a sparse direct solver using a stack bar where each color stands for a single instance of the 10 (13) sparse matrices listed in Table 8.1 (8.1 and 8.2). Not surprisingly, a sparse direct solver outperforms all the parallel preconditioned CG schemes [44]. However, the performance gap between the direct solver and some preconditioned CG schemes such as SSOR, ICF(0) and Parasails decreases as the number of processors increase. Our ICT schemes show good performance throughout the instances, but the performance improvement for a large number of processors is not as substantial. This difference stems from the fact that ICT schemes and the parallel sparse direct solver suffers from relatively higher communication cost than other parallel preconditioning schemes. In other words, the improvements from reduced arithmetic operations per processor is insignificant compared to the communication cost. In addition to ICT schemes, Block Jacobi is not showing a good parallel speedup for a different reason. Typical implementations of Block Jacobi perform no communication between processors during preconditioner construction and its application. However, on a large number of processors, the simple diagonal block data distribution reduces the size of the diagonal block as the number of processors is increased. This in turn increases the number of iterations for large numbers of processors, especially for matrices such as bodyy5, which contain many nonzero entries far from its diagonal as illustrated in Figure 8.12.

We also investigate the performance when solving for multiple right-hand sides, a situation that often arises in time dependent PDEs and non-linear optimization. The results on solving 10 right-hand side vectors for the same coefficient matrix shown in Figure 8.10 (for the 10 matrices) and 8.11 (for the 13 matrices) indicate that the higher factorization costs of ICT schemes are justified by the smaller number of iterations. Needless to say, the direct solver outperforms other schemes further than the case of a single right-hand side.

For the sake of completeness, we provide the memory costs of all schemes in Figure 8.13. SSOR requires no extra space beyond the coefficient matrix, $A$ and other
Table 8.2. Description of additional three test cases that successfully solved all the preconditioned CG schemes except AMG. $N$ is the matrix dimension, $|A|$ is the number of nonzeros in the matrix.

| Matrix    | $N$   | $|A|$      | Description                                           |
|-----------|-------|-----------|-------------------------------------------------------|
| augustus5 | 134144 | 645028    | Diffusion equation from 3D mesh                       |
| cfd2      | 123440 | 3087898   | CFD: pressure matrix                                  |
| engine    | 143571 | 2424822   | Engine head, Linear tetrahedral elements              |

Fig. 8.6. The success ratio of preconditioned iterative schemes: convergence within 600 iterations (left) and within 2000 iterations (left).
Fig. 8.7. Number of iterations to solve the 10 (right) and 13 (left) problems. All the values are derived from an average of all processor sets.
Fig. 8.8. Total time for solving a single right-hand side vector associated with the 10 problems (relative to the direct Cholesky solver) with 1 (upper left), 4 (upper right), 8 (lower left), and 16 (lower right) processors.
Fig. 8.9. Total time for solving a single right-hand side vector associated with the 13 problems (relative to the direct Cholesky solver) with 1 (upper left), 4 (upper right), 8 (lower left), and 16 (lower right) processors.
Fig. 8.10. Total time for solving 10 right-hand side vectors associated with the 10 problems (relative to the direct Cholesky solver) with 1 (upper left), 4 (upper right), 8 (lower left), and 16 (lower right)
Fig. 8.11. Total time for solving 10 right-hand side vectors associated with the 13 problems (relative to the direct Cholesky solver) with 1 (upper left), 4 (upper right), 8 (lower left), and 16 (lower right)
Fig. 8.12. The nonzero elements distribution of body5.

Fig. 8.13. The average memory cost of the parallel solution schemes relative to the memory required to store A (set to 1).
preconditioning schemes require small amounts of extra memory beyond $A$. ICT schemes require a little larger space than the others, but comparable to Parasails and AMG.

We now evaluate the performance for **augustus7**. Figures 8.14 and 8.15 present the total solution time and the number of iterations for solving a single right-hand side vector using 4 to 64 processors. Note that AMG is not able to solve this problem, and that the direct solver was not able to solve the problem with less than 16 processors due to very high memory requirements. In Figure 8.14, the advantage of parallel preconditioned CG is more pronounced; all the schemes with 16 processors are able to solve the linear system faster than the sparse direct solver with 64 processors. Figure 8.15, also shows that ICT-SI and ICT-SSAI, again, spend much fewer number of iterations than other preconditioners. Consequently, when solving 10 right-hand sides, ICT-SSAI exhibits the best performance as demonstrated in Figure 8.16.

### 8.4 Summary

The experiments with model grid problems reveal that the ICT-SI and ICT-SSAI schemes are scalable even though the costs of preconditioner construction increase substantially. ICT-SSAI outperforms ICT-SI because preconditioner construction is faster. The experiments with matrices from practical applications indicate that our hybrid schemes are robust with performance comparable to other parallel preconditioning schemes. ICT schemes are especially suitable for the solution of multiple right-hand side vectors. Additionally, ICT-SSAI achieves the best performance for **augustus7** (whose size exceeds one million). In summary, parallel ICT-CG hybrid solvers are viable for large sparse linear systems that are not well-conditioned.
Fig. 8.14. Total solution time using parallel preconditioned CG for augustus7: the dashed line indicates the time required by the sparse direct solver with 64 processors.

Fig. 8.15. The number of iterations to solve augustus7.
Fig. 8.16. Total solution time for 10 right-hand sides using parallel preconditioned CG for augustus7: the dashed line indicates the time required by the sparse direct solver with 64 processors.
In this dissertation, we have developed a parallel hybrid direct-iterative sparse linear solver based on new schemes for parallel incomplete Cholesky preconditioning and its latency-tolerant application to accelerate the convergence of the method of Conjugate Gradients (CG). Our work demonstrates that scalable application of incomplete factorization preconditioners is indeed possible even on distributed memory multiprocessors where the latency of communication can be several orders of magnitude greater than the per-word transfer time between processors.

On distributed memory multiprocessors, traditional parallel substitution schemes to apply incomplete factor preconditioners become a performance bottleneck. This fact gave rise in the last decade to a new class of preconditioners that seek a sparse approximation to the inverse of the coefficient matrix $A$ to enable its application through highly efficient and latency-tolerant sparse matrix-vector multiplication. However, preconditioners from this class are typically less effective than preconditioning through incomplete factorization. Our new methods use an effective combination of drop-threshold incomplete factorization (ICT) coupled with selective inversion; the latter produces approximate inverses of certain submatrices in the incomplete factor. This, in turn, allows the replacement of latency-dominated parallel substitution by latency-tolerant parallel matrix vector multiplication when the preconditioner is applied at each iteration. In our methods, the selective inversion can be done explicitly (SI) following incomplete factorization or it can be done through the application of sparse approximate inversion (SSAI) techniques to selected submatrices. In addition, our drop-threshold scheme can allow a wide range of fill-in to meet the preconditioning needs of sparse linear systems from a variety of applications. Thus, we provide a highly-tunable and effective ICT-SI-CG or ICT-SSAI-CG hybrid solvers.

We analyzed our methods on model sparse matrices from finite-difference discretizations on regular two-dimensional ($K \times K$) and three-dimensional ($K \times K \times K$) grids; our results are summarized in Table 9.1. Note that preconditioner construction using ICT-SSAI is very efficient with computational costs for two-dimensional problems that grow as $O\left(\frac{K^2 \log_2 K}{P} + \frac{K^3}{P^{1.5}}\right)$ and communication costs that grow as $Pt_s + \frac{K^2}{P} \log_4 Pt_w$. 
The latter is independent of $K$ with respect to latency costs (which grow linearly with the number of processors) and is thus significantly better than ICT and ICT-SI. More importantly, as shown by analysis in Chapter 7, communication latencies for applying the preconditioner from both ICT-SI and ICT-SSAI methods are independent of the problem size and grow as $(\log_2 P)^2$; traditional methods using substitution have latency-costs that grow as $O\left(\frac{K}{\sqrt{P}}\right) \left(\frac{K^2}{P^{2/3}}\right)$ for two-dimensional (three-dimensional) model problems. These improvements were made possible primarily through our SI and SSAI techniques that allow the selective use of parallel matrix-vector multiplication to apply the incomplete factor preconditioners. Our empirical results substantiate our analysis and are summarized in Tables 9.2 and 9.3; although our ICT-SI and ICT-SSAI schemes have speedups that are lower than that of Parasails, the actual execution time can be lower from the improved quality of preconditioning (as indicated by the lower number of iterations). These results indicate that our methods should be particularly suitable for problems that require significant preconditioning to accelerate CG, especially when the preconditioner is constructed once and applied to solve a sequence of right-hand side vectors.

The performance of the various methods for the three largest problems in our sparse matrix test-suite is shown in Table 9.4. Observe that our methods provide effective preconditioning to allow CG to converge with significantly fewer iterations than preconditioners using IC with no more fill than in A, fixed-point iterative methods like SSOR and Jacobi and sparse approximate inversion (Parasails). Note that for these problems, AMG does not result in the convergence of CG. Our methods are also significantly more robust than the other schemes as indicated by the failure rate of under 10% for a test suite with 21 matrices. Despite the relatively high costs for preconditioner construction, our methods can lead to least time when the preconditioner costs are amortized over the solution of ten or more linear systems. Our methods thus provide parallel robust, memory-scalable and latency-tolerant solution for ill-conditioned sparse linear systems.

The remaining part of the chapter contains brief discussions of research that builds upon and extends the contribution in this thesis.

**Parallel ILU Preconditioning for Non-symmetric Matrices.** The techniques used for ICT-SI and ICT-SSAI can be applied to develop parallel ILU preconditioners that can be used to accelerate the convergence of Krylov subspace methods for non-symmetric matrices such as the Generalized Minimum Residual (GMRES) and Quasi-Minimum Residual (QMR) iterations [9, 98]. New issues that arise in this situation include pivoting to maintain stability and thus improve preconditioning quality.
<table>
<thead>
<tr>
<th>Method</th>
<th>Computational Cost</th>
<th>Communication Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Latency</td>
</tr>
<tr>
<td>ICT</td>
<td>$\frac{71 K^3}{6P}$</td>
<td>$3Kt_s$</td>
</tr>
<tr>
<td>ICT-SI</td>
<td>$\frac{38 K^3}{3P}$</td>
<td>$6Kt_s$</td>
</tr>
<tr>
<td>ICT-SSAI</td>
<td>$\frac{71 K^3}{P^1.5} + \frac{K^2 \log_2 K}{P^2}$</td>
<td>$16Pt_s$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Computational Cost</th>
<th>Communication Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Latency</td>
</tr>
<tr>
<td>ICT</td>
<td>$\frac{145K^6}{14P}$</td>
<td>$\frac{7}{3}K^2 t_s$</td>
</tr>
<tr>
<td>ICT-SI</td>
<td>$\frac{76K^6}{1P}$</td>
<td>$7K^2 t_s$</td>
</tr>
<tr>
<td>ICT-SSAI</td>
<td>$\frac{145K^6}{P^2}$</td>
<td>$16Pt_s$</td>
</tr>
</tbody>
</table>

Table 9.1. Summary of computational and communication costs (dominant terms) for parallel ICT factorization schemes on sparse matrices from 2 and 3 dimensional model grid problems using $P$ processors.
<table>
<thead>
<tr>
<th></th>
<th>SSOR</th>
<th>Jacobi</th>
<th>ICF</th>
<th>PS</th>
<th>ICT-SI</th>
<th>ICT-SSAI</th>
<th>AMG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of Iterations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>163</td>
<td>139</td>
<td>200</td>
<td>213</td>
<td>53</td>
<td>53</td>
<td>2</td>
</tr>
<tr>
<td>16 processors</td>
<td>545</td>
<td>463</td>
<td>706</td>
<td>503</td>
<td>133</td>
<td>150</td>
<td>2</td>
</tr>
<tr>
<td><strong>Time for Preconditioner Construction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>0.000</td>
<td>0.013</td>
<td>0.719</td>
<td>0.709</td>
<td>0.353</td>
<td>0.353</td>
<td>0.376</td>
</tr>
<tr>
<td>16 processors</td>
<td>0.000</td>
<td>0.004</td>
<td>0.308</td>
<td>0.319</td>
<td>0.337</td>
<td>0.329</td>
<td>0.334</td>
</tr>
<tr>
<td>Scaled Speedup</td>
<td>NA</td>
<td>47.34</td>
<td>37.41</td>
<td>35.53</td>
<td>16.7670</td>
<td>17.1436</td>
<td>17.99</td>
</tr>
<tr>
<td><strong>Time for Iterative Solution for single right-hand side</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>1.718</td>
<td>1.069</td>
<td>5.562</td>
<td>2.476</td>
<td>0.853</td>
<td>0.853</td>
<td>0.379</td>
</tr>
<tr>
<td>16 processors</td>
<td>0.253</td>
<td>1.557</td>
<td>8.148</td>
<td>2.120</td>
<td>1.494</td>
<td>1.470</td>
<td>0.545</td>
</tr>
<tr>
<td><strong>Total Solution Time for single right-hand side</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>1.718</td>
<td>1.081</td>
<td>6.281</td>
<td>3.184</td>
<td>1.207</td>
<td>1.207</td>
<td>0.755</td>
</tr>
<tr>
<td>16 processors</td>
<td>2.253</td>
<td>1.561</td>
<td>8.456</td>
<td>2.439</td>
<td>1.8307</td>
<td>1.7998</td>
<td>0.879</td>
</tr>
<tr>
<td>Scaled Speedup</td>
<td>12.20</td>
<td>10.98</td>
<td>11.89</td>
<td>20.89</td>
<td>10.54</td>
<td>10.73</td>
<td>13.74</td>
</tr>
<tr>
<td><strong>Total Solution Time for 10 right-hand sides</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>17.18</td>
<td>10.70</td>
<td>56.34</td>
<td>25.46</td>
<td>8.888</td>
<td>8.888</td>
<td>4.167</td>
</tr>
<tr>
<td>16 processors</td>
<td>22.53</td>
<td>15.58</td>
<td>81.79</td>
<td>21.52</td>
<td>15.27</td>
<td>15.03</td>
<td>5.787</td>
</tr>
<tr>
<td><strong>Failure Rate</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>16 processors</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 9.2. Scalability of parallel sparse linear solvers for sparse matrices from 2-dimensional model grid problems; PS stands for Parasails(1); tol = 0.01 is applied for ICT-SI and ICT-SSAI; level-of-fill for ICF is 0.
<table>
<thead>
<tr>
<th></th>
<th>SSOR</th>
<th>Jacobi</th>
<th>ICF</th>
<th>PS</th>
<th>ICT-SI</th>
<th>ICT-SSAI</th>
<th>AMG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Iterations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>37</td>
<td>34</td>
<td>46</td>
<td>54</td>
<td>22</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>16 processors</td>
<td>79</td>
<td>75</td>
<td>86</td>
<td>82</td>
<td>36</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>Time for Preconditioner Construction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>0.000</td>
<td>0.011</td>
<td>0.539</td>
<td>0.659</td>
<td>1.377</td>
<td>1.377</td>
<td>0.532</td>
</tr>
<tr>
<td>16 processors</td>
<td>0.000</td>
<td>0.002</td>
<td>0.198</td>
<td>0.256</td>
<td>2.108</td>
<td>0.703</td>
<td>3.744</td>
</tr>
<tr>
<td>Scaled Speedup</td>
<td>NA</td>
<td>76.33</td>
<td>43.55</td>
<td>41.16</td>
<td>10.45</td>
<td>31.32</td>
<td>2.274</td>
</tr>
<tr>
<td>Time for Iterative Solution for a single right-hand side</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>0.294</td>
<td>0.204</td>
<td>0.836</td>
<td>0.415</td>
<td>0.266</td>
<td>0.341</td>
<td></td>
</tr>
<tr>
<td>16 processors</td>
<td>0.181</td>
<td>0.148</td>
<td>0.548</td>
<td>0.232</td>
<td>0.347</td>
<td>0.384</td>
<td>1.363</td>
</tr>
<tr>
<td>Scaled Speedup</td>
<td>25.98</td>
<td>22.11</td>
<td>24.43</td>
<td>28.60</td>
<td>22.82</td>
<td>3.744</td>
<td></td>
</tr>
<tr>
<td>Total Solution Time for a single right-hand side</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>0.294</td>
<td>0.216</td>
<td>1.375</td>
<td>1.074</td>
<td>1.642</td>
<td>1.642</td>
<td>0.873</td>
</tr>
<tr>
<td>16 processors</td>
<td>0.181</td>
<td>0.150</td>
<td>0.746</td>
<td>0.488</td>
<td>2.455</td>
<td>1.087</td>
<td>5.107</td>
</tr>
<tr>
<td>Scaled Speedup</td>
<td>5.976</td>
<td>22.96</td>
<td>29.51</td>
<td>35.19</td>
<td>10.70</td>
<td>24.18</td>
<td>2.736</td>
</tr>
<tr>
<td>Total Solution Time for 10 right-hand sides</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>2.938</td>
<td>2.054</td>
<td>8.903</td>
<td>4.809</td>
<td>4.034</td>
<td>4.034</td>
<td>3.945</td>
</tr>
<tr>
<td>16 processors</td>
<td>1.810</td>
<td>1.481</td>
<td>5.676</td>
<td>2.578</td>
<td>5.851</td>
<td>4.538</td>
<td>17.37</td>
</tr>
<tr>
<td>Scaled Speedup</td>
<td>25.98</td>
<td>22.19</td>
<td>25.10</td>
<td>29.85</td>
<td>11.57</td>
<td>14.23</td>
<td>3.634</td>
</tr>
<tr>
<td>Failure Rate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 processor</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>16 processors</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 9.3. Scalability of parallel sparse linear solvers for sparse matrices from 3-dimensional model grid problems; PS stands for Parasails(1); $tol = 0.01$ is applied for ICT-SI and ICT-SSAI; level-of-fill for ICF is 0.
<table>
<thead>
<tr>
<th></th>
<th>SSOR</th>
<th>Jacobi</th>
<th>ICF</th>
<th>PS</th>
<th>ICT-SI</th>
<th>ICT-SSAI</th>
<th>AMG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of Iterations (Average of the three)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 processors</td>
<td>1459</td>
<td>1414.7</td>
<td>1226</td>
<td>538</td>
<td>171.7</td>
<td>213</td>
<td>NA</td>
</tr>
<tr>
<td>16 processors</td>
<td>1456</td>
<td>1475.7</td>
<td>1233.7</td>
<td>544.6</td>
<td>182</td>
<td>227.3</td>
<td>NA</td>
</tr>
<tr>
<td><strong>Relative Time for Preconditioner Construction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 processors</td>
<td>0.00</td>
<td>0.0003</td>
<td>0.846</td>
<td>0.168</td>
<td>0.323</td>
<td>0.273</td>
<td>NA</td>
</tr>
<tr>
<td>16 processors</td>
<td>0.00</td>
<td>0.0001</td>
<td>0.173</td>
<td>0.065</td>
<td>0.146</td>
<td>0.109</td>
<td>NA</td>
</tr>
<tr>
<td>Speedup</td>
<td>NA</td>
<td>3.01</td>
<td>4.89</td>
<td>2.58</td>
<td>2.21</td>
<td>2.50</td>
<td>NA</td>
</tr>
<tr>
<td><strong>Relative Total Solution Time for a single right-hand side</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 processors</td>
<td>0.577</td>
<td>0.657</td>
<td>2.467</td>
<td>0.709</td>
<td>0.466</td>
<td>0.411</td>
<td>NA</td>
</tr>
<tr>
<td>16 processors</td>
<td>0.126</td>
<td>0.109</td>
<td>0.552</td>
<td>0.204</td>
<td>0.225</td>
<td>0.154</td>
<td>NA</td>
</tr>
<tr>
<td>Speedup</td>
<td>4.598</td>
<td>6.057</td>
<td>4.472</td>
<td>3.474</td>
<td>2.073</td>
<td>2.674</td>
<td>NA</td>
</tr>
<tr>
<td><strong>Relative Total Solution Time for 10 right-hand sides</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 processors</td>
<td>4.079</td>
<td>4.189</td>
<td>11.032</td>
<td>3.716</td>
<td>0.958</td>
<td>1.001</td>
<td>NA</td>
</tr>
<tr>
<td>16 processors</td>
<td>1.317</td>
<td>1.072</td>
<td>3.901</td>
<td>1.205</td>
<td>0.675</td>
<td>0.520</td>
<td>NA</td>
</tr>
<tr>
<td>Speedup</td>
<td>3.097</td>
<td>3.907</td>
<td>2.828</td>
<td>2.253</td>
<td>1.419</td>
<td>1.92</td>
<td>NA</td>
</tr>
<tr>
<td><strong>Failure Rate for solving 21 matrices using 1–16 processors</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600 iterations</td>
<td>34.3 %</td>
<td>50.5 %</td>
<td>43.7 %</td>
<td>42.9 %</td>
<td>9.5 %</td>
<td>9.5 %</td>
<td>52.4 %</td>
</tr>
<tr>
<td>2000 iterations</td>
<td>45.7 %</td>
<td>35.2 %</td>
<td>23.8 %</td>
<td>4.8 %</td>
<td>9.5 %</td>
<td>9.5 %</td>
<td>52.4 %</td>
</tr>
</tbody>
</table>

Table 9.4. Summary of the performance of parallel sparse linear solvers for sparse matrices from applications; PS stands for Parasails(1); tol = 0.01 is applied for ICT-SI and ICT-SSAI; level-of-fill for ICF is 0. Execution times are relative to that for a 16 processor direct method using Cholesky factorization (set to 1.0).
Improving Efficiency of Preconditioner Construction. As mentioned in the empirical study of preconditioner construction in Chapter 6, ICT-SI factorization does not have ideally scalability due to relatively high communication costs. Although ICT-SSAI is better than ICT-SI, its performance does somewhat degrade with the number of processors. This performance bottleneck for ICT-SSAI is caused by the multiple executions of parallel sparse approximate inversion on each distributed supernodal matrix. This problem can be potentially alleviated by combining some distributed supernodes (for instance, a parent and its children) together to form a large supernodal submatrices. This may increase computational costs for preconditioner construction and the number of CG iterations due to sparse approximate inversion applied to a larger portion of the entire matrix. On the other hand, the number of sparse approximate inversion steps is reduced and the efficiency is improved due to a large matrix size. In the extreme case, all distributed supernodes could be merged to a single supernode to yield a parallel two-level preconditioner [101].

Object-Oriented Blocked Parallel IC Preconditioner. As described in Chapter 3 and 4, both ICT-SI and ICT-SSAI are integrated with some data structures such as trees, supernodes, matrix computation method for supernodal matrices. In fact, these ICT schemes can be tuned by replacing a new method to compute a diagonal submatrix and its corresponding off-diagonal submatrix. Such a plug-and-play scheme can be implemented using object-oriented methods. For instance, such techniques have been applied for parallel Jacobi preconditioning [10, 11, 104] to allow the user to choose an existing sequential preconditioning technique to be applied for every diagonal block submatrix. Recently, Chow employed this approach for a sequential ILU [22] and achieved a good performance for blocked sparse matrices. Therefore, extending these ideas to our parallel ICT preconditioning framework is natural. The main question is how to construct interface and software tools to enable such code integration with ease. It would be interesting to explore the use of the Common Component Architecture (CCA) [5] techniques to achieve this goal.

Application to Nonlinear Optimization and Eigenvalue Computation. It would be interesting to study the behavior of our preconditioning schemes for nonlinear numerical optimization where sparse linear systems are solved repeatedly. We are currently investigating the use of our methods with optimization methods such as the Gradient Projection-Conjugate Gradient method (GPCG) [18] and the Newton methods enhanced by trust region or line search [88], available in the TAO package [12] for
distributed memory multiprocessors. We plan to compare the performance of our ICT with other parallel preconditioned iterative methods for solving practical problems using TAO [12].

Another interesting application contains computing several eigenvalues and their corresponding eigenvectors of a sparse matrix. This problem occurs, for example, in Normal Coordinate Analysis [108]. Solution schemes such as shift-and-invert Lanczos are required to solve the system of a shifted matrix, \((A - \lambda)x = y\) where \(\lambda\) is either an eigenvalue or a shift parameter to find eigenvalues near the shift. We propose to solve this problem using ICT-SI/SSAI scheme with SLEPc [57], a parallel eigenvalue computation package built using PETSc [11].
References


Vita

Keita Teranishi was born in Ninohe, Iwate, Japan on December 29, 1974. He received his high school education at Kaisei high school in Tokyo, Japan from 1987 to 1993. After one year in the college of Liberal Arts at the International Christian University, Tokyo, Japan, he transferred to the University Tennessee, Knoxville in 1995. In 1998, he received the BS degree in computer science (mathematics minor) with the highest honor (Summa Cum Laude). In 2000, he received the MS degree in computer science from the University of Tennessee, Knoxville. In the same year, he joined the Ph.D. program in computer science at the Pennsylvania State University. During the Ph.D. study, he worked as an intern at NERSC, Lawrence Berkeley National Laboratory in the summer 2002 and MCS division, Argonne National Laboratory in the fall 2002. His main research interest is parallel scientific computing related to sparse matrix computations. He is also interested in application of the Grid computing techniques to enable modelings and simulations in various science and engineering disciplines.