Winter 2001

New Sequential and Scalable Parallel Algorithms for Incomplete Factor Preconditioning

David A. Hysom
Old Dominion University

NEW SEQUENTIAL AND SCALABLE PARALLEL ALGORITHMS FOR INCOMPLETE FACTOR PRECONDITIONING

by

David A. Hysom
MS, Computer Science, Old Dominion University, 1997
BS, Sociology, The University of the State of New York, 1991

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December 2001

Approved by:

Alex Pothen (Director)

Michele Benzi (Member)

David Keyes (Member)

Linda Stals (Member)

Mohammad Zubair (Member)

ABSTRACT

NEW SEQUENTIAL AND SCALABLE PARALLEL ALGORITHMS FOR INCOMPLETE FACTOR PRECONDITIONING

David A. Hysom
Old Dominion University, 2001
Director: Dr. Alex Pothen

The solution of large, sparse, linear systems of equations $Ax = b$ is an important kernel, and the dominant term with regard to execution time, in many applications in scientific computing. The large size of the systems of equations being solved currently (millions of unknowns and equations) requires iterative solvers on parallel computers. Preconditioning, which is the process of translating a linear system into a related system that is easier to solve, is widely used to reduce solution time and is sometimes required to ensure convergence. Level-based preconditioning (ILU(ℓ)) has long been used in serial contexts and is widely recognized as robust and effective for a wide range of problems. However, the method has long been regarded as an inherently sequential technique. Parallelism, it has been thought, can be achieved primarily at the expense of increased iterations. We dispute these claims.

The first half of this dissertation takes an in-depth look at structurally based ILU(ℓ) symbolic factorization. There are two definitions of fill level, "sum" and "max," that have been proposed. Hitherto, these definitions have been cast in terms of matrix terminology. We develop a sequence of lemmas and theorems that provide graph theoretic characterizations of both definitions; these characterizations are based on the static graph of a matrix, $G(A)$.

Our Incomplete Fill Path Theorem characterizes fill levels per the sum definition; this is the definition that is used in most library implementations of the "classic" ILU(ℓ) factorization algorithm. Our theorem leads to several new graph-search algorithms that compute factors identical, or nearly identical, to those computed by the "classic" algorithm. Our analyses shows that the new algorithms have lower run time complexity than that of the previously existing algorithms for certain classes of matrices that are commonly encountered in scientific applications.

The second half of this dissertation presents a Parallel ILU algorithmic framework (PILU). This framework enables scalable parallel ILU preconditioning by combining concepts from domain decomposition and graph ordering. The framework can accommodate ILU(ℓ) factorization as well as threshold-based ILUT methods.

A model implementation of the framework, the Euclid library, was developed as part of this dissertation. This library was used to obtain experimental results for Poisson's equation, the Convection-Diffusion equation, and a nonlinear Radiative Transfer problem. The experiments, which were conducted on a variety of platforms with up to 400 CPUs, demonstrate that our approach is highly scalable for arbitrary ILU(ℓ) fill levels.

Dedicated to Phyllis Woods;
be kind to your secretaries.

ACKNOWLEDGEMENTS

Graduate school, like life, comes with neither an instruction booklet nor a web page. Or perhaps there really is an instruction booklet, only it has been hidden away and forgotten on a dusty shelf in a faculty lunchroom. Whatever, I share the common experience of many, that graduate school is like being left in a dark room with an elephant, a hitherto unimagined beast. Some of us stumble across a trunk and think, "aha, we've got it!" Others of us, groping leg or backside, are equally sure we have comprehended the beasts nature. Alone, we know little. But working together we slowly piece together the puzzle, an accurate picture of the elephant falls into place.

First and foremost, I acknowledge Florin Dobrian and Gary Kumfert, fellow students with whom, for several years, I shared a dark room with an elephant. Dinesh Kaushik and Kara Olson also wandered in from time to time, and told us interesting things about the elephant's toenails and tusk.

I would like to thank my advisor, Alex Pothen, for opening doors, for providing financial support, and for his meticulous insistence that we describe the elephant exactly. My only regret is that our educational system demanded that he spend much time perched on a stool in a cold, dim room writing grants. I wish he had been free to spend more time with us, figuring out the elephant.

I extend many thanks to my committee members. My dissertation has benefited greatly from your questions, comments, and editing. I am particularly grateful for the support of David Keyes, who spends more time in airplanes traveling between coasts than anyone I have ever known, but still finds time to walk into a darkened room, light a match, and point out elephant shadows on the wall.

I am tempted to thank the federal government, who approved many of Alex and David's grants, thus funding much of my graduate studies. But the government is, as some will remember, "we, the people," so I do not think any special recognition is in order. Instead, I urge any who are reading these words and also citizens of the United States of America to pat themselves on the back. Folks, I couldn't have done it without you.

Penultimately, I thank my parents and siblings, Lee, Bev, Stu, Ron, and Tom, for their encouragement and humor. Hey guys, I didn't mean everything I said.

You can go ahead and burn the tapes now, and don't forget to delete the email.

Finally, Lawrence Livermore National Laboratory provided a truly nurturing environment during my final 18 months of study. In particular, as an avid cyclist, I thank the Cycletrons, `http://www.llnl.gov/LLESA-groups/cycletrons/`, whose noon time rides provided much desperately needed stress release. Roll on!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 THE UNDERLYING THEME

This dissertation is about structure, by which I mean the theoretical relationship of paths in the graph of a matrix to the location of nonzero entries in an incomplete factor of the matrix . Experimentally, the matrices we will be concerned with arise from the discretization of systems of partial differential equations (PDEs) on grids or meshes. Discretization of such PDEs give rise, directly or indirectly, to linear systems of equations of the form

$$Ax = b.$$

Nowadays these systems are becoming very large, to the extent that we do not have computing resources to solve the problems by direct factorization, but must rely on iterative methods. In the most general terms, an iterative solution method begins with a guess at the actual solution, e.g., the zero vector. This guess is then refined (updated) during an iterative process by adding a correction vector, $p^{(i)}$, to the most recent solution vector

$$x^{(i+1)} = x^{(i)} + p^{(i)}.$$

The update vector can be considered to be a (complicated) function of the original matrix A and the right-hand side vector b. Iterations are terminated and the system is considered solved when an exit criterion is satisfied. Typically, the exit criterion is specified as a relative reduction in the residual norm

$$||r^{(i)}|| < rtol * ||r^{(0)}|| \quad \text{where} \quad r^{(i)} = b - Ax^{(i)}.$$

The choice of exit criterion is somewhat problematical. If the matrix A is ill conditioned a large reduction in the residual norm may translate to only a small reduction in the error norm. The error norm at the conclusion of the ith iteration is

$$||e^{(i)}|| = ||x - x^{(i)}||, \quad \text{where } x \text{ is the true solution.}$$

This dissertation is formatted in accordance with the SIAM Journal on Scientific Computing.

For experimental purposes, researchers frequently begin with a known solution z and generate an artificial right-hand side,

$$b = Az.$$

In such cases we can actually compute the true error (up to roundoff error in the machine), and hence can compare the residual reduction with the error reduction. While this may reveal information about the matrix's conditioning, such an artificial right-hand side also removes the influence of real-life boundary conditions and source terms, which may lead to higher frequency components in the solution than when z is simply chosen.

A large number of iterative methods, operating on various principles, have been devised and investigated over the years [1, 38, 45, 77]. The multi-authored "Templates" book [3], which is available online at `http://www.netlib.org`, provides a concise introduction and overview of both theory and implementation.

The convergence behavior (i.e., the required number of iterations) of the system being solved depends on the numerical properties of A; the quality of the initial guess; the specific iterative method of choice; and the right-hand side vector b. For PDEs the vector b typically represents boundary conditions and forcing terms. The numerical properties of A include definiteness, symmetry, condition number, the clustering and spread of the eigenvalues, and other properties.

Much past research has focused on the numerical properties of A and how these properties affect convergence. Quite a bit is known concerning the behavior of M-matrices and positive definite systems, although mathematicians remain somewhat puzzled (i.e., there is little theory) as to how the numerical properties of unsymmetric and indefinite systems affect convergence behavior.

This thesis is only peripherally concerned with the numerical properties of linear systems. Numerical properties make themselves known in the experimental sections wherein we report iteration counts, but nowhere in this thesis do we attempt to prove, in the numeric analytic sense, convergence bounds. However, the parallel ILU algorithms that are the topic of the second half of this dissertation can be interpreted in a sequential context as a matrix reordering followed by factorization. Hence, a plethora of known numerical results governing convergence behavior of ILU preconditioned systems is directly applicable.

1.2 PRECONDITIONING REVIEW

Mathematically, preconditioning can be summarized as the process of translating the system

$$Ax = b$$

into the related system

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

where the preconditioner M in some sense approximates A. This is a representation of *left* preconditioning; more generally we have

$$M_1^{-1}AM_2^{-1}(M_2x) = M^{-1}b,$$

where $M = M_1M_2 \approx A$. This is referred to as *split* preconditioning. If $M_2 = I$ the system degenerates to left preconditioning, and if $M_1 = I$ we arrive at *right* preconditioning. Although split, left, and right preconditioned systems may have identical spectra, they sometimes require different numbers of iterations for convergence; these phenomena are described in more detail in [77], and changing the preconditioner may change the norm in which convergence is most naturally measured.

The preconditioner M is usually derived in some fashion from the matrix A. Sometimes, as in multigrid methods, information from the underlying grid of unknowns and discretization scheme may also be used. M is usually not directly computed, nor is the matrix-matrix multiplication $M_1^{-1}A$ performed. In ILU methods one computes the factors L and U, where $LU = M$. In approximate inverse methods one may compute either M^{-1} or the factors L and U, where $U^{-1}L^{-1} = M^{-1}$.

The preceding description of preconditioning is abstract. What happens inside the computer for ILU preconditioning is this. Prior to the Krylov solve a preconditioner setup function is called. This function takes as input the matrix A and other parameters such as level or threshold (depending on the ILU method employed), and computes and returns a pair of incomplete factors of A. During each iteration of a Krylov solve a preconditioner application function is called. This function takes as input a vector, y, to which the preconditioner is applied, and the result is stored and returned as a vector z. For ILU(ℓ) the setup function has the form:

```
PC_Setup(const Matrix A_in,
         const integer level_in,
         Matrix L_out,
         Matrix U_out)
{
  //compute and return L_out and U_out, where
  //L_out and U_out are ILU(level_in)
  //                incomplete factors of A_in.
}
```

For left preconditioning the application function has the form:

```
PC_Apply(cont Matrix L_in,
         const Matrix U_in,
         const Vector y_in,
         Vector z_out)
{
   //solve: L_in * U_in * z_out = y_in
   //       for z_out
}
```

Most practical preconditioning methods can be placed in one of the classes [3, 77]:

- matrix-splitting, which includes Jacobi and SSOR;
- polynomial;
- Approximate Inverse;
- domain decomposition;
- Support-theory based;
- multilevel;
- ILU.

This taxonomy is somewhat arbitrary. ILU preconditioners, for example, can be considered the result of matrix splitting, and Block Jacobi is identical to zero-overlap Additive Schwarz.

Approximate Inverse, domain decomposition, support theory, and ILU preconditioning methods are discussed in separate subsections below. Brief descriptions of matrix-splitting and polynomial preconditioners follow.

A matrix can easily be split into strict lower triangular, diagonal, and upper triangular components, $A = E+D+F$. Jacobi preconditioning results from taking $M = D$, in which case the computation of M^{-1} is trivially easy.

SSOR preconditioning is defined by the splitting

$$M_{SSOR}(A) = (D - \omega L)D^{-1}(D - \omega F).$$

In practice it is common to take $\omega = 1$, since the determination of an optimal value for ω is a nontrivial task, apart from certain well studied cases with constant coefficients.

In polynomial preconditioning M is defined by $M^{-1} = s(A)$, where s is some polynomial, e.g., Neumann or Chebyshev, of low degree. Polynomial preconditioning has a long history and is of interest due to its inherent parallelism. However, the method has limitations that appear to preclude its effectiveness as a general high-performance method. See [8] for additional discussion and references.

Most preconditioners have both point and block formulations. One should note that *block* is an overloaded term and potentially misleading, having several common connotations. First, when beginning with a set of equations such as Navier-Stokes (NS) in CFD, we end up with multiple unknowns (aka, degrees of freedom (DOF)) associated with each grid point. For 2D NS for example, each gridpoint has five DOF, and with nodal ordering the resulting (sparse) matrix is consequently composed of dense 5×5 blocks. Thus, *block* may refer to a division of the matrix into equally sized square submatrices, such that each submatrix is either dense or zero. On the other hand, a matrix may be blocked into a number of square submatrices where the block size is arbitrary, and has nothing to do with any underlying equation(s); in this case the blocks will generally be sparse. A good discussion of these distinctions can be found in [19]. Finally, in Block Jacobi preconditioning the connotation is of a block diagonal matrix, with all off-diagonal entries set to zero.

1.2.1 APPROXIMATE INVERSE PRECONDITIONING

There has recently been considerable interest in *Approximate Inverse* (APINV) preconditioning, wherein the matrix $M^{-1} \approx A^{-1}$ is computed, either explicitly or in factored form. Depending on the specific algorithm, the inverse's sparsity pattern may be entirely determined before numerical computation begins, initially determined then updated as numerical computation progresses, or entirely determined during the course of numerical computation. APINV methods are attractive from a parallel viewpoint since, once formed, preconditioner application consists of easily parallelizable matrix-vector multiplications. However, while some algorithms for computing an approximate inverse are highly parallelizable (e.g., norm-minimization techniques such as SPAI), others are not. From a theoretical viewpoint, APINV preconditioning relies on the assumption that a matrix inverse, which is in general dense, can be well-represented by a sparse matrix. Justification for this assumption is so far mostly experimental; it has been shown [28] that, given a matrix with an irreducible sparsity pattern, numerical values can always be assigned such that the inverse is completely dense.

The earliest use of approximate inverses in parallel environments is credited to Benson [5] and Frederickson [33]. Chow, et. al., [21] present a concise survey of various APINV methods. A wealth of analysis, algorithms, experimental data, and additional references can be found in [8, 9, 22].

APPROXIMATE INVERSE AND ILU COMPARISON

The study of APINV preconditioning is newer and hence less developed than that of ILU, and perhaps partially for this reason comparisons between APINV and ILU are inconclusive. Some researchers report that, although APINV preconditioning is sometimes more robust and stable than ILU, it has not (yet) turned out to be cost-competitive with ILU factorization [22]. Others report the APINV is superior to ILU for a variety of problems [7, 8, 22]. After perusing numerous experimental studies, this author is of the opinion that such seemingly conflicting reports are mostly a reflection of the enormity of the experimental space and the numerous possible bases for comparison. Problems may be symmetric or not, positive definite or not, well or poorly conditioned, etc. Preconditioning for either ILU or APINV may be left, right, or split. Matrices may be reordered and/or scaled. Performance

comparison can be based on convergence (iteration count) or execution time, either of which can (and probably should) be considered in conjunction with a second dimension, the number of nonzeros in the preconditioner.

1.2.2 DOMAIN DECOMPOSITION AND GRAPH PARTITIONING

Briefly described, domain decomposition is the process of solving a system resulting from the discretization of one or more partial differential equations (PDEs) on a grid or mesh by partitioning the meshpoints into subdomains such that points within subdomains share physical locality. Each subdomain then constitutes a smaller problem that can be solved locally, although some subdomain solves may require boundary node information from other subdomains. Domain decomposition thus revolves around the idea of divide-and-conquer [77].

In parallel computing, *domain decomposition* is often used synonymously with *data decomposition*, which is the process of decomposing and distributing data structures in a distributed memory environment. On the other hand, domain decomposition is also used to refer to the collection of local solutions, used as a preconditioner for solution of a system of algebraic equations [79].

Domain decomposition terminology has been borrowed by researchers and implementors working in linear algebra and graph theory, and it is now common to refer to "subdomains" rather than "subgraphs," even for systems having no direct physical counterpart. See [57, 78] for a discussion of this development.

PARTITIONING, LOCALITY, AND COMMUNICATION COSTS

A key component implicit in domain decomposition methods is the principle that nodes inhabiting a common subdomain share locality. They should have, on average, higher connectivity with each other than with nodes in foreign subdomains. This locality may arise naturally, as is the case when formation of subdomains is guided by the physical features of a problem, or when behavior in different portions of a physical system are described by different PDEs. Locality may also be artificially imposed. A domain decomposition method may take as input a grid of unknowns or a system of linear equations, and divide the whole into parts through employment of a partitioning algorithm that attempts to minimize the number of cut edges, a cut edge being an edge that crosses subdomain boundaries.

In domain decomposition, cut edges represent scalar data dependencies. A cut edge indicates that the value associated with a boundary node in one subdomain is needed in the local solve of another subdomain. When there is a one-to-one mapping between subdomains and processors in a distributed computing environment these scalar data dependencies become communication dependencies.

DOMAIN DECOMPOSITION AND PARALLEL ILU COMPARISON

The Parallel ILU (PILU) algorithm proposed in the next section is similar to domain decomposition methods in its partitioning requirements. It is also similar in that the majority of computation is performed locally within subdomains. However, a cut edge in PILU represents a data (communication) dependency that involves the upper-triangular row of a matrix, whereas a cut edge in a domain decomposition method is typically interpreted as representing dependence on a value from a vector. It is then not surprising that PILU uses different data structures, and has different communication cost-analyses, than domain decomposition methods.

1.2.3 SUPPORT GRAPH PRECONDITIONERS

Over a decade ago, Vaidya proposed a family of preconditioners for M-matrices [80]. Later, Gremban, Miller, and Zagha [39, 40] extended the support graph theory that underlies the preconditioners, and constructed additional families of preconditioners. The Vaidya preconditioners have recently been implemented and tested experimentally [16], and analytic and experimental research is ongoing [10, 11].

The support graph preconditioners can be interpreted as members of the ILU family. One first forms a spanning tree of $G(A)$, then augments this graph with additional edges. This results in a preconditioner matrix that contains only a portion of the nonzero entries in A. A complete factorization of M is then performed. This resulting L and U factors are thus incomplete factors of the original matrix, A. A primary attraction of support graph theory is that it points the way to the development of new analytical tools for analyzing convergence.

1.2.4 ILU PRECONDITIONING

GRAPH THEORETIC CONSIDERATIONS

Modeling of sparse factorization by sequences called *elimination graphs* is credited to Parter for symmetric matrices [70], and Haskins and Rose for unsymmetric matrices [46]. Many researchers subsequently contributed to the theory, one of the more influential works being that of Rose and Tarjan [73], wherein the Fill Path Theorem was introduced. This theorem gives a static characterization of fill for complete factorization, *static* meaning that fill is completely described by the structure of the graph of the initial matrix, $G(A)$. This modeling and characterization led to the development of elimination trees, which form the basis for several practical algorithms for computing fill for the complete L and U factors of structurally symmetric matrices. Liu's survey article [59] provides an overview of many of these developments.

An elimination tree is the transitive reduction of the directed graph of the Cholesky factor L of a matrix A. This means that, if the factor contains a nonzero entry l_{ij}, then there is a fill path joining nodes i and j in $G(A)$, and a directed path joining nodes i and j in the elimination tree. Elimination trees are thus path-preserving, but not path-length preserving.

ILU ALGORITHMS

ILU preconditioners were first developed for M-matrices [65], a canonical example of which is the Laplacian discretized on a regular grid using central differencing with three point support in the x and y directions. More formally, A is an M-matrix if it is invertible, has all diagonal entries $a_{ii} > 0$, all off diagonal entries $a_{ij} \leq 0$, and all entries in the inverse (diagonal or otherwise) $a_{ij}^{-1} \geq 0$.

Many ILU variants have been developed over the years, most of which can be placed into either of two categories, structure-based ILU(ℓ) and threshold-based ILUT. In ILU(ℓ) the locations of permitted nonzero entries are determined in a symbolic phase; this is followed by a numeric phase, wherein the actual values are computed. This two-step procedure is analogous to that used for direct factorization of symmetric matrices, where the factor's structure is first computed through use of its elimination tree. In ILUT, symbolic and numeric factorization is interleaved

on a row by row basis. A row is updated from a previously factored row, and entries permitted in the factor, only if the pivots or entries are larger than some specified value. It is also common practice to put an upper limit on the number of permitted entries in each row regardless of numeric value, e.g., only five entries may be permitted in a row in addition to the number of entries in the corresponding row of A.

Although ILU preconditioning is widely used and considered robust and effective for a wide range of problems, the method can fail for a variety of reasons. The following summary is taken from Chow and Saad [20], which provides an excellent review and reference list. Existence and a form of stability can be proved for M-matrices [65], however, diagonal perturbations are required to help guarantee existence for the general positive definite case. Unsymmetric matrices are more problematical; here, the L and U factors may be much more poorly conditioned than A, and the solves unstable. These problems can worsen for indefinite matrices. Factorization may fail, or inaccuracies arise, due to zero or very small pivots. Excessive inaccuracies can be introduced if the dropping strategy results in the discard of too many nonzero entries. Even when factorization goes through, the factors may be far from diagonally dominant, resulting in unstable triangular solves.

ILU PRECONDITIONING AND PARALLELISM

Per the review by Chan and van der Vorst, [15], most researchers agree that parallel computation of ILU preconditioners necessitates trading off convergence for parallelism. The review identifies three methodologies for extracting or increasing parallelism in ILU methods: matrix reordering, replacement of ILU by a series expansion or polynomial preconditioner, and domain decomposition.

Numerous theoretical and experimental results have been reported regarding the interplay of ordering, parallelism, and convergence [6, 29, 31]. Most authors conclude that orderings that are "highly parallel," such as red-black in matrices whose graphs are two-colorable, result in increased iterations during iterative solution. Most of these studies, however, concentrate on preconditioners that have approximately the same number of nonzeros as the initial problem. When additional fill is permitted the "parallel" orderings can actually result in fewer iterations; this is clearly evident in Duff and Meurant's study in the results for ILUT and ILU(1) [29].

Permitting increased fill, of course, results in increased execution time in both the factorization phase and each application phase, and requires additional storage. For these reasons preconditioner effectiveness can best be measured by execution time, with a careful eye towards resource availability.

1.3 NEW CONTRIBUTIONS, RELATION TO EARLIER WORK, AND THESIS SUMMARY

The following lists the primary new contributions presented in this dissertation. Each contribution, and its relationship to previous work, is discussed in the subsequent paragraphs. This section also serves as a roadmap to the remainder of this work.

- Graph theoretic characterization of ILU(ℓ) fill.
- Graph-search algorithms for computing ILU(ℓ) fill and storage requirements.
- Scalable parallel ILU algorithmic framework (PILU).

1.3.1 GRAPH THEORETIC FILL CHARACTERIZATION

Chapter 2 of this work contains a sequence of lemmas and theorems that characterize where fill occurs in ILU(ℓ) factors. We show that fill can be determined "statically," by examining the graph of a matrix, $G(A)$, and that the nonzero structure of each row in the incomplete factors L and U of A can be determined independently. We develop characterizations for the two commonly used fill definitions, the *sum* rule and the *max* rule. Hitherto, these rules were cast in terms of matrix nomenclature, i.e, in terms of two nonzero entries in a matrix that cause a previously zero entry to "fill" during factorization. Our theorems show that the level of a fill entry corresponds to a path length in a graph.

Our fill theorems are generalizations of the original fill path theorem which, as discussed above (Section 1.2), did not encompass the concept of fill path length. We add the result that fill path lengths are determinants of fill levels in matrices.

Our characterization for fill in factors computed using the sum rule (Theorem 4) was known to D'Azevedo, Forsyth, and Tang [24], who reported a similar finding. There are two differences between their work and ours. First, although they were

clearly aware of the connection, they *defined* all fill levels in terms of path lengths in $G(A)$. We start with a far weaker premise. We only define level 0 fill entries, which are nonzero entries in the original matrix, A. We then use a graph theoretic model of row-oriented Gaussian elimination to prove the "if-and-only-if" relationship between path lengths in graphs and fill levels in matrices.

Second, D'Azevedo, Forsyth, and Tang's characterization used the concept of *reachable sets*, which is inherently dynamic in nature. A *reachable set* refers to the set of vertices that has been removed from a graph during Gaussian elimination. This concept is not needed as long as one assumes that an ordering is associated with the vertices. Employing reachable sets obscures the observation that the fill path lengths are static in nature, and hence the nonzero structure of each row in the matrix can be determined independently of that of any other row.

We also develop a static characterization of fill for ILU factors that are computed using the max rule. To the best of our knowledge there is no previous work in this regard. Finally, we show where the two characterizations coincide, and where they differ.

1.3.2 GRAPH-SEARCH ALGORITHMS

Chapter 3 contains several new algorithms for computing ILU(ℓ) structures. Previous algorithms ("classical ILU(ℓ)") for computing ILU(ℓ) structures operated by merging in previously factored rows of the U factor with the current row being factored. Our algorithms operate on a different principle: they determine structure by performing breadth first searches in the graph $G(A)$. These algorithms are a natural extension of the theorems developed in Chapter 2.

Determining fill by performing searches in graphs is not an entirely new concept. Eisenstat and Liu [32] used depth first searches to compute fill for complete L and U factors of matrices. The main thrust of their work was to speed up the symbolic factorization process of structurally unsymmetric matrices by making use of that portion of a matrix that might be symmetric. (Although the elimination tree leads to very fast algorithms for computing the structure of complete factors for symmetric matrices, there is no equivalent method for computing the structures of structurally unsymmetric matrices.) Gilbert and Liu [36] also presented an algorithm for computing the structure of complete L and U factors. Their algorithm

combines row-merging and graph-search features.

To the best of our knowledge, we are the first to design graph search algorithms for computing incomplete L and U factors. We prove that our new algorithms have lower run time complexity than the classical algorithm for matrices arising from PDE discretizations on 2D and 3D grids.

We also show how these algorithms can be modified to compute ILU(ℓ) storage requirements in $O(n)$ space. This is an advancement over current approaches, which either artificially limit the amount of fill by, for example, stipulating the maximum number of nonzero entries permitted in any row of the factor, or dynamically reallocate storage during the factorization process. Placing an artificial limit is disadvantageous since numerically large entries (which one would like to remain in the factor) may be arbitrarily dropped, thereby lowering preconditioner effectiveness. Dynamic reallocation is disadvantageous due to system call overhead time, and the possibility of memory fragmentation.

1.3.3 PARALLEL ILU ALGORITHMIC FRAMEWORK

Chapter 4 introduces a Parallel ILU preconditioning framework that can accommodate both ILU(ℓ) and ILUT factorization methods. Chapter 5 contains amplifying theory and results that show the method is scalable. Chapter 6 examines how partitioning and subdomain size effect the algorithms performance. Chapter 7 contains additional results. Our algorithm attains parallelism through a dual (global followed by local) reordering phase, and imposes a subdomain graph constraint that permits communication patterns to be determined prior to factorization, thus minimizing the possibility that long sequential dependency paths will arise during the factorization process.

Although developed independently, our preconditioning algorithm shares features with work reported by other researchers. Earlier attempts at parallel algorithms for preconditioning (including approaches other than incomplete factorization) are surveyed in [15, 27, 81]; orderings suitable for parallel incomplete factorizations have been studied inter alios in [6, 26, 29]. The surveys also describe the alternate approximate inverse approach to preconditioning.

Saad [77, Section 12.6.1] discusses a distributed ILU(0) algorithm that has the features of graph partitioning, elimination of interior nodes in a subdomain before

boundary nodes, and coloring the subdomains to process the boundary nodes in parallel. Only level zero preconditioners are discussed there, so that fill between subdomains, or within each subdomain, do not need to be considered. No implementations or results were reported, although Saad has informed us recently of a technical report [61] that includes an implementation and results. Our work, done independently, shows how fill levels higher than zero can be accommodated within this algorithmic framework. We also analyze our algorithm for scalability and provide computational results on the performance of PILU preconditioners. Our results show that fill levels higher than zero are indeed necessary to obtain parallel codes with scalability and good performance.

Karypis and Kumar [55] have described a parallel ILUT implementation based on graph partitioning. Their algorithm does not include a symbolic factorization, and they discover the sparsity patterns and the values of the boundary rows after the numerical computation of the interior rows in each subdomain. The factorization of the boundary rows is done iteratively, as in the discussion given above, where we show how the subdomain graph constraint might be relaxed. The partially filled graph of the boundary rows after the interior rows are eliminated is formed, and this graph is colored to compute a schedule for computing the boundary rows. Since fill edges in the boundary rows are discovered as these rows are being factored, this approach could lead to long dependency paths that are $\Theta(p)$. The number of boundary rows is $\Omega(N^{1/2})$ for 2D meshes, and $\Omega(N^{2/3})$ for 3D meshes with good aspect ratios. If the cost of factoring and communicating a boundary row is proportional to the number of rows, then this phase of their algorithm could cost $\Omega(p\sqrt{N})$, severely limiting the scalability of the algorithm (cf. the discussion in Section 4.3).

Recently Magolu monga Made and van der Vorst [62, 63] have reported variations of a parallel algorithm for computing ILU preconditioners. They partition the mesh, linearly order the subdomains, and then permit fill in the interior and the boundaries of the subdomains. The boundary nodes are classified with respect to the number of subdomains they are adjacent to, and are eliminated in increasing order of this number. Since the subdomains are linearly ordered, a "burn from both ends" ordering is employed to eliminate the subdomains. Our approaches are similar, except that we additionally order the subdomains by means of a coloring

to reduce dependency path lengths to obtain a scalable algorithm. They have provided an analysis of the condition number of the preconditioned matrices for a class of 2D second order elliptic boundary value problems. They permit high levels of fill (four or greater) as we do, and show that the increased fill permitted across the boundaries enables the condition number of the preconditioned matrix to be insensitive to the number of subdomains (except when the latter gets too great). We have worked independently of each other.

A different approach, based on partitioning the mesh into rectangular strips and then computing the preconditioner in parallel steps in which a "wavefront" of the mesh is computed at each step by the processors, was proposed by Bastian and Horton [4] and was implemented for shared memory multiprocessors recently by Vuik, van Nooyen, and Wesseling [84]. This approach has less parallelism than the one considered here.

1.4 CODE AVAILABILITY

As part of this dissertation, a model PILU implementation, the *Euclid library*, was designed and implemented. This library, which is implemented in C, has been tested on various experimental platforms, as detailed in Appendix B. The code is freely available for download, along with an interface to PETSc [2], at `http://www.cs.odu.edu~/pothen/software.html` or `http://www.cs.odu.edu/~hysom/Euclid/index.html`.

CHAPTER 2

INCOMPLETE FILL PATH THEOREMS

2.1 INTRODUCTION

Two *incomplete fill path theorems*, which are the primary theoretical contribution of this work, are generalizations of the original *fill path theorem*, due to Rose, Tarjan, and Leuker [72, 73]. The original theorem characterizes fill for the complete factors of a matrix, $A = LU$. It describes an intimate relationship between the structure of the graph of any given matrix, and the structure of the graph of that matrix's factors (for brevity, we sometimes abbreviate "the structure of the graph of a matrix" to "the structure of a matrix"):

Definition 1 *A* fill path *is a path joining two vertices i and j, all of whose interior vertices are numbered lower than the end vertices i and j.*

Theorem 2 *Let $F = L + U - I$ be the filled matrix corresponding to the complete factorization of A. Then $f_{ij} \neq 0$ if and only if there exists a fill path joining i and j in the graph $G(A)$ [72, 73].*

This theorem tells us that one can determine where fill will occur during factorization without actually performing the factorization. That is, fill locations are directly discernible from the initial graph. Hence, the theorem is said to provide a "static" characterization of fill.

(Regarding terminology, when describing the nonzero structure of a matrix's factors researchers have variously used the terms "fill," "fillin," and "fill-in." Usually, "fillin" or "fill-in" is used to describe entries that are zero in the original matrix but, due to the existence of a fill path, are allowed to become nonzero in the factor. We prefer to use the term "fill" to denote any nonzero entry in the factors. The term *filled matrix* denotes the matrix $F = L + U - I$, where L and U are either complete or incomplete factors of A.)

Application of Theorem 2 has resulted in the gradual development of the notion of elimination trees, and many algorithms of practical importance for direct methods. Liu provides a good overview of these developments in [59]. For symmetric

problems $A = LL^T$, the structure of L can be computed extremely quickly, in time essentially proportional to the number of nonzeros in the factor.

Just as in direct methods, it is possible to formulate static characterizations of fill for certain classes of ILU factors. ILU(ℓ) and IC(ℓ) factors are of interest since their products (although never explicitly formed during computation) have long been recognized as effective preconditioners for the iterative solution of important scientific problems, e.g., elliptic systems resulting from the discretization of second-order partial differential equations (PDEs). For symmetric positive definite problems, the structure of the ILU(ℓ) or IC(ℓ) factors can be completely determined prior to numerical factorization. For problems that require pivoting for numerical stability, and factorizations produced by methods that employ numerical criteria (e.g., ILUT), this is not possible. Since pivoting is dynamically determined during the factorization, any static prediction of structure must include all the nonzeros that would be present in *all possible* orderings. This is an upper bound, which, though it can be computed, is too big in practice [34].

In this chapter we develop new incomplete fill path theorems that characterize where fill occurs in ILU(ℓ) factors. We begin with an overview of existing ILU(ℓ) algorithms, and go on to discuss two rules (sum and max) that have been used to determine level assignments during ILU(ℓ) factorization. We then introduce a graph theoretic model for incomplete factorization that extends previously developed models for complete factorization. With this foundation in place, we present and prove several theorems that characterize incomplete fill.

2.2 CLASSICAL ILU(ℓ) FACTORIZATION

By *classical ILU(ℓ)* we refer to a family of widely known and implemented algorithms that compute ILU(ℓ) factors by mimicking direct factorization. These algorithms determine permitted fill based on the concept of a matrix entry's *level*. As in direct factorization, but unlike some ILU variants (e.g., ILUT), factorization is divided into two distinct phases. In the *symbolic phase* the locations of permitted nonzero fill entries are determined. In the subsequent *numeric phase* the values of the entries are computed. (In this work, when we write "ILU(ℓ)," we generally refer to the symbolic factorization phase.)

As in direct factorization, in their outermost loops classical ILU(ℓ) algorithms

may iterate over matrix rows, columns, or diagonal entries. Multifrontal approaches are also possible. Since the vast majority of current scientific codes are row (i.e., equation) oriented, for the remainder of this work we will be exclusively concerned with row-oriented algorithms.

Row-oriented ILU is said to be *upward looking*. That is, for every nonzero entry f_{ih} with $h < i$, row i is updated by merging in the upper-triangular portion of the previously factored row h. During this process a matrix entry f_{ij}, whose value was previously zero, may become nonzero (i.e, may "fill in") if there exists a nonzero entry f_{hj}. Here $h < i$, $h < j$, but j may be either greater or lesser than i. We say the fill entry f_{ij} is *caused by* the existence of the two entries f_{ih} and f_{hj}.

During ILU(ℓ) factorization all matrix entries are assigned an integer-based *level*. To get the ball rolling, all nonzero entries in the original matrix are assigned the level zero, and zero-valued entries are assigned the level infinity. (Actually, the assignment of "infinity" is a mathematical nicety; algorithmically, inside the computer, we simply ignore (do not allocate data structures for) zero entries.)

A potential fill entry f_{ij} is assigned a level based on the levels of its two causative entries (the rules used to assign levels are discussed in the next section). If the assigned level is not greater than ℓ, the entry is permitted to become nonzero during numeric factorization (mathematically, we say the entry is added to a sparsity set of permitted fill). Since a fill entry may have many different pairs of causative entries, and hence potentially be assigned many different levels, the tie-breaking rule is to assign it the lowest possible level.

Figure 1 contains a statement of the row-oriented CLASSIC-ILU algorithm. Nonzero matrix entries are represented by adjacency lists. If row j of matrix A contains a nonzero entry a_{ji}, then the adjacency list $adj(j)$ contains an element i. The compressed sparse row storage format (CSR), which is arguably the most commonly used data structure for computer matrix representation, is an adjacency list construct. The `computeWeight()` function in Step 10 is the subject of the next section.

2.3 FILL LEVEL ASSIGNMENT RULES

The level associated with a matrix entry f_{ij} in the matrix $F = L + U - I$, where $LU \approx A$, is denoted $level(i, j)$. As previously noted, entries in F corresponding

```
CLASSIC-ILU(A, ℓ)
1  # Initialization phase
2  for j = 1 to n
3      adj'(j) ← ∅
4      for t ∈ adj(j)
5          level(j, t) ← 0
6          insert t in adj'(j)
7      # Row-merge update phase
8      for each unprocessed i ∈ adj'(j) with i < j in ascending order
9          for t ∈ adj'(i) with t > i
10         wt = computeWeight(level(j, i), level(i, t))
11             if wt ≤ ℓ
12                 if t ∋ adj'(i)
13                     insert t in adj'(j)
14                     level(j, t) ← wt
15                 else
16                     level(j, t) ← min{level(j, t), wt)}
```

FIG. 1. CLASSIC-ILU *algorithm. The input matrix A contains n rows. The structure of a row a_{j*} is represented by the list $adj(j)$. The structure of a factor row f_{j*} is represented by the list $adj'(j)$.*

to nonzero entries in A are initially associated with the level zero, and numerically zero entries are associated with the level *infinity*.

There are two rules in the literature for assigning levels to fill entries that arise during factorization: the *sum* rule and the *max* rule. As before, we assume that fill entry f_{ij} is caused by previously admitted entries f_{ih} and f_{hj}. Mathematically, the sum rule states

$$level(i,j) = \min_{1 \le h < \min\{i,j\}} \{level(i,h) + level(h,j) + 1\}. \tag{1}$$

In words, this rule assigns a level which is the sum of the level of two causative entries, incremented by 1. All ILU(ℓ) implementations of which this author is aware make use of the sum rule when assigning levels during factorization. Intuitively, the sum rule is appealing since, by this rule, an entry's level is a direct indication of the minimum number of times any of its updates will be divided by a pivot value during the numeric factorization phase. Hence, for important classes of matrices (e.g., diagonally dominant) entries with higher levels are expected to be smaller in absolute value, and hence have less influence in establishing the factor's character.

In contrast, the max rule for level assignment states

$$level(i,j) = \min_{1 \le h < \min\{i,j\}} \max\{level(i,h), level(h,j)\} + 1. \tag{2}$$

This method is intuitively appealing (particularly to computer scientists) due to its recursive flavor. To compute an ILU(ℓ) factor using the max rule, one can perform an ILU(1) factorization ℓ times. The input for the first iteration is the structure on the initial matrix A, the input for subsequent iterations in the structure (sparsity set) computed during the previous iteration. At the commencement of each iteration, all entries in the input sparsity set are considered as level zero entries. (Note that the sum and max rules always produce identical level assignments for ILU(1) factorizations. For matrices arising from 2D, five-point discretizations, the rules also produce identical results for ILU(2) factorizations.)

From a numerical viewpoint, however, this rule has less to recommend itself, since with this rule an entry's level is *not* indicative of the minimum number of times any of its updates will be divided by a pivot value. With the max rule the minimum number of pivot divisions for a level ℓ entry ranges between ℓ and 2^{ℓ}.

The origins of the sum and max rules are difficult to pin down precisely because the foundational ideas were developed gradually, over many years. Historically, incomplete factorization structures were specified either by considering gridpoint operators associated with discretizations of PDEs on regularly structured, naturally ordered grids; by examining the "banded" appearance (as would be seen in a Matlab spy plot) of the resulting matrix [13, 15, 66, 68, 69, 83]; or by considering matrix splittings. More accurate factorizations were arrived at by specifying larger stencils for the factor, or by permitting the inclusion of additional diagonal bands, or by adding the structure of the remainder matrix to the sparsity set.

The term "incomplete factorization" appears to have been coined by Meijerink and Van der Vorst [65]. A discussion of recursive factorization, which can be shown equivalent to the max rule, is presented by Axelsson [1], who attributes its origin to Gustafsson [43]. The first clear statement of the sum rule that we have been able to locate was enunciated by D'Azevedo, Forsyth, and Tang [24].

With reference to line 10 of Algorithm 1, we are primarily interested in two weighting functions that are functional counterparts to the max and sum level assignment rules. The sum weighting function is

$$\texttt{computeWeight}(\text{level}(i,h),\text{level}(h,j)) = \text{level}(i,h) + \text{level}(h,j)) + 1. \qquad (3)$$

The max weighting function is

$$\texttt{computeWeight}(\text{level}(i,h),\text{level}(h,j)) = \max\{\text{level}(i,h),\text{level}(h,j)\} + 1. \qquad (4)$$

To distinguish between levels computed using the max or sum functions, we sometimes write "S-level" or "M-level" in place of the more general term, "level." Similarly, we may write S-level(i,j) or M-level(i,j) instead of level(i,j).

2.4 GRAPH THEORETIC ILU(ℓ) MODEL

Parter [70], and later Rose and Tarjan [73], developed graph theoretic vertex elimination processes that model complete Gaussian Elimination. In this section we formalize a similar model for structurally based incomplete factorization.

A directed graph of a matrix $G(A) = (V, E)$ has vertex set V, which contains a vertex for every row in the matrix, and edgeset E, which contains a directed edge $\langle i, j \rangle$ for every nonzero entry a_{ij}. Edges are weighted, and the weight of an edge $\langle i, j \rangle$ is denoted level(i, j) (or sometimes more specifically, S-level(i, j) or M-level(i, j)). $G(A)$ is an ordered graph such that, if matrix row i is numbered less than matrix row j, then vertex i is ordered before vertex j.

All edges in $G(A)$ are assigned the weight zero. Given any two edges $\langle i, h \rangle$ and $\langle h, j \rangle$ that form a directed path $P(i, j)$, a weight can be assigned to the hypothetical edge $\langle i, j \rangle$ using either of the previously discussed weighting functions.

(Although we use brackets to indicate directed graph edges, for clarity we omit the brackets when specifying edge weights, i.e., we write level(i, j) instead of level$(\langle i, j \rangle)$. By a path's "length," we refer to the number of edges contained in the path. When a path $P(i, j)$ contains a single edge we have the equivalence: $P(i, j) = \langle i, j \rangle$.)

The *partial elimination process* is a sequence of graphs that models Gaussian elimination. The initial graph in the sequence, G_0, is identical to the graph of the matrix, $G(A) = (V, E)$. We assume the vertex set V contains n vertices. The graph G_{i+1}, for $0 < i \leq n$, is formed by examining all pairs of edges in G_i that form directed paths of length two: i, h, j, with $h < \min\{i, j\}$. For each such path $P(i, j)$, a directed edge $\langle i, j \rangle$ is inserted in E_{i+1} if and only if `computeWeight`(level(i, h), level(h, j)) is not greater than ℓ. If the hypothetical edge $\langle i, j \rangle$ has already been inserted, its weight is adjusted to the minimum of its present weight and the newly calculated weight. Hence, we denote the partial elimination process as

$$G(A) = G_0, G_1, G_2 \ldots, G_n = G_*. \tag{5}$$

For specificity, we use a superscript "S" to indicate when edge weights were calculated using the sum rule, e.g., G_*^S. Similarly, a superscript "M" indicates that edge weights were calculated using the max rule, e.g., G_*^M.

This partial elimination process models ILU(ℓ) factorization since matrix fill entries created or updated when row i is factored (iteration i in Algorithm 1) correspond exactly to edges inserted or updated during the formation of graph G_i. Hence we have $G_* = G(F)$.

The sequence of graphs defined above differs from previous models for complete factorization in at least one important aspect. The models for complete factorization are based on bordering methods, in which outer-product updates are performed while marching down a matrix's diagonal. Accordingly, one vertex is eliminated (removed) from V_i during the formation of graph $G_{i+1} = (V_{i+1}, E_{i+1})$ from graph $G_i = (V_i, E_i)$.

Row oriented factorization requires that we leave the vertex set intact, i.e, V_i and V_{i+1} are identical for $0 \leq i < n$. While it is possible to formulate a graph theoretic construct based on bordering for incomplete factorization, such a construct would not model the operation of the CLASSIC-ILU algorithm.

(Since our vertex sets remain constant, one might wonder whether the construct in Equation 5 is properly called an "elimination" model. Fortunately, good computer scientists, like politicians, are adept in the art of overloaded meanings. In the present situation, V_0 can be regarded as an unprocessed pool of vertices. When each graph G_{i+1} is formed from graph G_i a single vertex is processed, and thus *eliminated* from the pool of unprocessed vertices.)

2.5 STRUCTURAL CHARACTERIZATIONS

This section contains a collection of definitions, observations, lemmas, and theorems that provide static characterization of incomplete S-level and M-level fill. We also introduce the concept of 1-alternating fill paths, which are particular configurations of fill paths for which the S-level and M-level characterizations coincide.

Figure 2 provides a pictorial summary of the interconnections of this chapter's results.

2.5.1 STATIC CHARACTERIZATION OF S-LEVEL FILL

This section's first result tells us that nontrivial fill paths can always be decomposed into shorter fill paths.

Lemma 3 *Any fill path $P(i,j)$ that contains two or more edges can be uniquely decomposed into two fill paths, $P(i,h)$ and $P(h,j)$, each of which contains at least a single edge.*

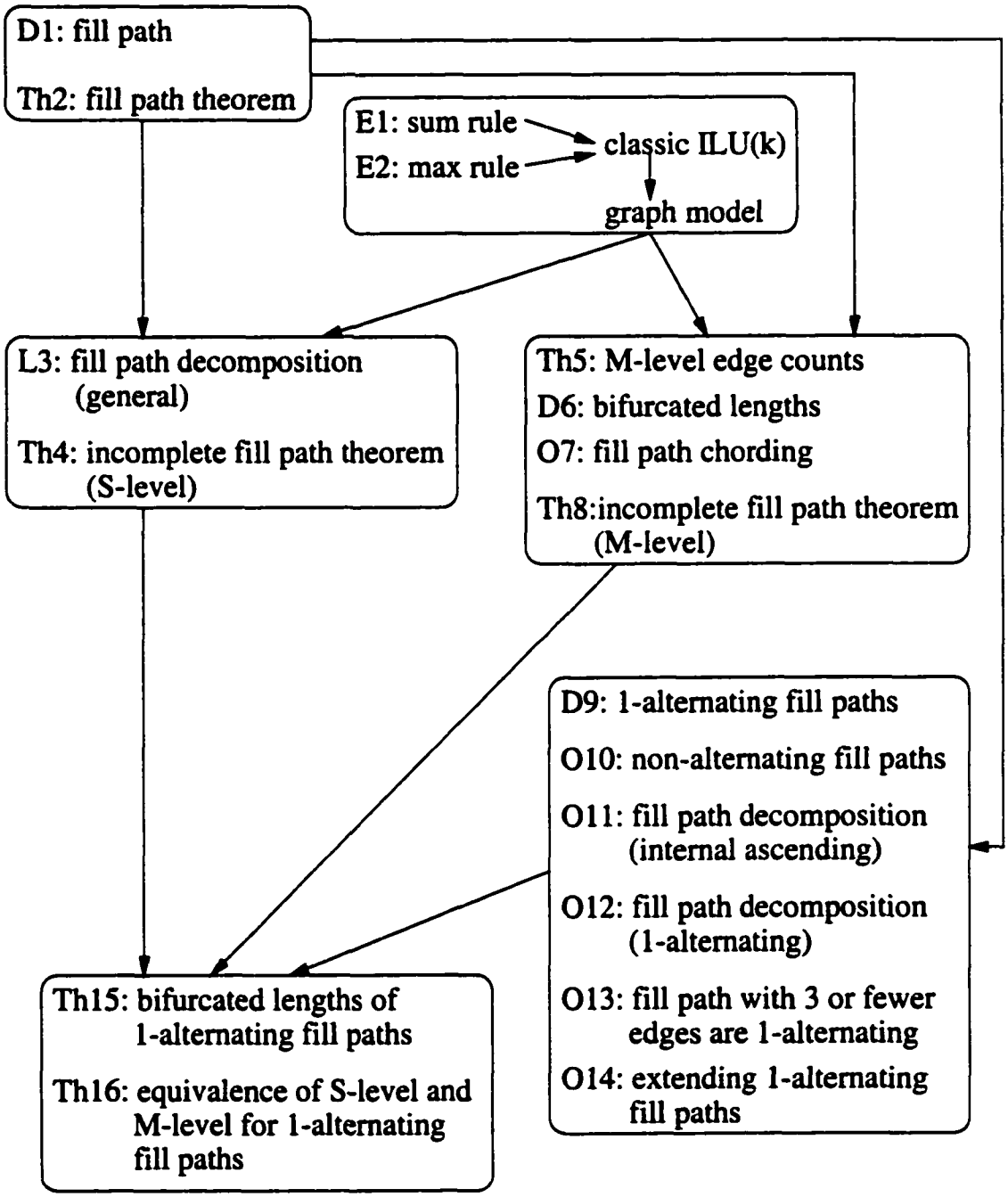


FIG. 2. *Relationships of definitions, theorems, and observations.*

Proof. Given a fill path $P(i,j)$ containing two or more edges, let h denote the highest numbered interior vertex on the path. The $P(i,h)$ section of this path is a fill path by the choice of h, since all intermediate vertices on this section are numbered lower that h. Similarly, the $P(h,j)$ section of this path is also a fill path. Thus, the fill path can clearly be decomposed in two subpaths, both of which are fill paths (existence).

To show uniqueness, suppose there exists some other decomposition. Let g be an interior vertex on the $P(i,j)$ path, distinct from h, such that both $P(i,g)$ and $P(g,j)$ sections are fill paths. Then h, which is also on the $P(i,j)$ path, must either be situated between vertices i and g, or between vertices g and j. Without loss of generality, assume vertex h is situated between vertices i and g. Then by Definition 1, $P(i,g)$ is not a fill path, since the path contains an interior vertex that is numbered higher than one of the end vertices. □

The next theorem, which is this section's main result, provides a static characterization of fill for classical ILU(ℓ) factors that are computed using the sum rule for level assignment.

Theorem 4 *Let $G(A) = (V,E)$ be the graph of a square matrix A, and let $\langle i,j \rangle$ be a permitted edge in G_*^S. Then S-level$(i,j) = k$ if and only if there exists a shortest fill path of length $k+1$ that joins i and j in $G(A)$.*

Proof. If there is a shortest fill path of length $k+1$ joining i and j in $G(A)$, we prove the result, that an edge $\langle i,j \rangle$ with S-level$(i,j) = k$ exists in G_*^S, by induction on u, which is the length of the fill path.

The base case $u = 1$ is immediate, since, by the construction in Section 2.4, a fill path of length one in the graph $G(A)$ is an edge $\langle i,j \rangle$ in $G(A)$, and edges in $G(A)$ are assigned level zero, and are also edges in G_*^S.

Now assume that the result is true for all lengths u less than $k+1$; we show it is also true for shortest paths of length $u = k+1$. Let $P(i,j)$ be a shortest fill path joining vertices i and j in $G(A)$, and let this path have length $u = k+1$.

Let h denote the highest numbered interior vertex on the fill path $P(i,j)$. We claim that the $P(i,h)$ section of this path is a shortest fill path in $G(A)$ joining i and h. This section is a fill path by the choice of h and Lemma 3. If there were a fill path joining i and h that was shorter than the $P(i,h)$ section, we would be able to

concatenate it with the $P(h,j)$ section to form a shorter $P(i,j)$ fill path. Hence the $P(i,h)$ section is a shortest fill path joining i and h. Similarly, the $P(h,j)$ section of this path is the shortest fill path joining h and j.

Since each of these sections has fewer than $k+1$ edges, and is a shortest fill path, the inductive hypothesis applies. Denote the number of edges in the $P(i,h)$ ($P(h,j)$) section of this path by v (w), where $v+w=u=k+1$. By the inductive hypothesis the edge $\langle i,h\rangle$ is a fill edge of level $v-1=k_1$, and the edge $\langle h,j\rangle$ is a fill edge of level $w-1=k_2$. Now by the sum rule for updating fill levels, when the vertex h is eliminated, we have a fill edge $\langle i,j\rangle$ of level

$$k_1+k_2+1=(v-1)+(w-1)+1=v+w-1=u-1=(k+1)-1=k.$$

Now we prove the converse. Suppose that $\langle i,j\rangle$ is a fill edge of level k in G_*^S; we show the result that there exists a shortest fill path in $G(A)$ of length $u=k+1$ edges by induction on the level k.

The base case $k=0$ is immediate since, by the construction in Section 2.4, the edge $\langle i,j\rangle$ constitutes a trivial fill path of length one.

Assume that the result is true for all fill levels less than k. Let the fill edge $\langle i,j\rangle$ with S-level$(i,j)=k$ be created in G_*^M, when vertex i is eliminated, by the previously existing edges $\langle i,h\rangle$ and $\langle h,j\rangle$. Let the edge $\langle i,h\rangle$ have level k_1 and the edge $\langle h,j\rangle$ have level k_2. By the sum rule for computing levels, we have that $k_1+k_2+1=k$. By the inductive hypothesis, there is a shortest fill path of length $v=k_1+1$ joining i and h, and such a path of length $w=k_2+1$ joining h and j. Concatenating these paths, we find a fill path joining i and j of length

$$v+w=(k_1+1)+(k_2+1)=k_1+k_2+2=k+1.$$

We need to prove that the $P(i,j)$ fill path in the previous paragraph is a shortest fill path between i and j. Consider any other pair of edges $\langle i,g\rangle$ and $\langle g,j\rangle$ in G_i^M that causes the fill edge $\langle i,j\rangle$ when vertex i is eliminated. By the choice of the vertex h, if the level of the edge $\langle i,g\rangle$ is k_1', and that of $\langle g,j\rangle$ is k_2', then $k_1'+k_2'\geq k$.

The inductive hypothesis applies to the $P(i,g)$ and $P(g,j)$ sections, and hence the sum of their lengths is at least $k+1$. □

D'Azevedo, Forsyth, and Tang [24] *defined* the (sum) level of a fill edge $\langle i,j\rangle$ using the length criterion employed here, and hence were aware of the connection

between matrix entry levels and fill path lengths. However, they did not postulate or prove this connection as a theorem, as we have done. In passing, we note that their work centers around a novel algorithm that combines an ordering technique with ILU factorization. They consider $G(A)$ to be initially unordered, and one vertex is ordered during each elimination step. They define fill levels in terms of path lengths through vertices in reachable sets, with a reachable set consisting of vertices already eliminated and ordered.

As we will show in the next chapter, the static characterization of S-level fill can be applied to develop new algorithms for computing ILU(ℓ) factors, to analyze the amount of fill for simple structured graphs, and to analyze ILU(ℓ) run-time complexities.

2.5.2 STATIC CHARACTERIZATION OF M-LEVEL FILL

We now turn our attention towards M-level fill entries and their associated fill paths. While a fill edge with S-level$(i,j) = k$ corresponds to a fill path with exactly $k+1$ edges, this section's first result says that a fill edge with M-level$(i,j) = k$ corresponds to a fill path that may contain anywhere between $k+1$ and 2^k edges. Figure 3 illustrates the intuition underlying this claim. Two very simple graphs are shown, both of which contain fill paths that correspond to level $k = 3$ fill edges. The fill path in the Figure 3(a) contains $3+1 = k+1 = 4$ edges, while the fill path in Figure 3(b) contains $2^3 = 2^k = 8$ edges.

Theorem 5 *Let $G(A) = (V,E)$ be the graph of a square matrix A, let $\langle i,j \rangle$ be a permitted edge in G_*^M with M-level$(i,j) = k$, and let $P(i,j)$ be a corresponding fill path in $G(A)$. Let u represent the number of edges in the path $P(i,j)$. Then $k+1 \leq u \leq 2^k$.*

Proof. We argue by induction on the fill edge's level, k. The base case $k = 0$ is immediate since, by the construction specified in Section 2.4, a fill edge of level zero corresponds to a fill path that contains $u = 1$ edges. In this case $k+1 = 1 \leq u \leq 2^k = 1$, so the result is true.

Now assume the result is true for all edges whose M-level is less than k; we show it is also true for edges with level k. Let h be the vertex whose elimination creates the fill edge $\langle i,j \rangle$ of M-level k. Let the edge $\langle i,h \rangle$ have M-level k_1 and

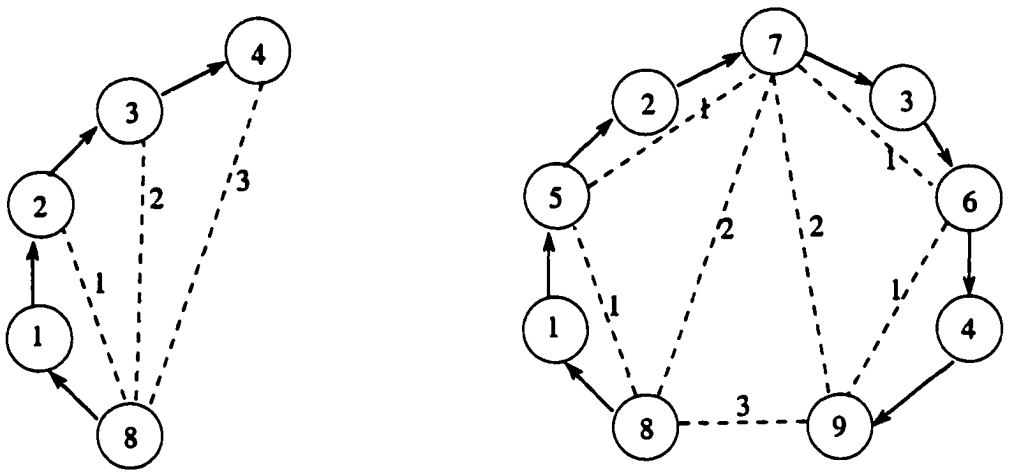


FIG. 3. *M-level fill level and path length relationships. Edges in $G(A)$ are drawn with solid lines. Edges in G_*^M are drawn with dashed lines, and labeled with their levels. The vertex numbering indicates elimination ordering. Both graphs contain an M-level $k = 3$ fill edge. The corresponding fill path in the graph on the left, 8,1,2,3,4, contains $3 + 1 = k + 1 = 4$ edges. The corresponding fill path in the graph on the right, 8,1,5,2,7,3,6,4,9, contains $2^3 = 2^k = 8$ edges.*

the edge $\langle h, j \rangle$ have M-level k_2. By the max rule for computing levels, we have that $\max\{k_1, k_2\} + 1 = k$, hence both k_1 and k_2 are less than k, so the inductive hypothesis applies. Also, either $k_1 = k - 1$ or $k_2 = k - 1$ or both. Without loss of generality, assume $k_1 = k - 1$.

Let v represent the number of edges in the fill path joining i and h in $G(A)$, and w the number of edges in the fill path joining h and j in $G(A)$. By the inductive hypothesis, $k_1 + 1 \leq v \leq 2^{k_1}$, and $k_2 + 1 \leq w \leq 2^{k_2}$. When h is eliminated these paths are concatenated, resulting in the fill path $P(i, j)$ whose length u is bounded:

$$(k_1 + 1) + (k_2 + 1) \leq u \leq 2^{k_1} + 2^{k_2}.$$

To make the left-hand side as small as possible, assume $k_1 = k - 1$ and $k_2 = 0$, which is possible if $P(h, j)$ contains a single edge. In this case

$$u = (k_1 + 1) + (k_2 + 1) = ((k - 1) + 1) + (0 + 1) = k + 1.$$

To make the right-hand side as large as possible, let $k_1 = k - 1$ and $k_2 = k - 1$. In this case

$$u = 2^{k_1} + 2^{k_2} = 2^{(k-1)} + 2^{(k-1)} = 2^k.$$

Therefore, $k + 1 \leq u \leq 2^k$. □

Not only is there wide latitude in fill path lengths associated with M-level fill edges, but it is also the case that a fill path $P(i, j)$ in $G(A)$ that is associated with a fill edge $\langle i, j \rangle$ in G_*^M may not be the shortest fill path (that is, the fill path containing the fewest number of edges) that connects vertices i and j in $G(A)$. Figure 4 illustrates this point. The figure shows a fill edge with M-level(i, j)=3 that arises due to the existence of a fill path that contains eight edges. Vertices i and j are also connected by a fill path that only contains five edges; however, this fill path would cause $\langle i, j \rangle$ to have M-level$(i, j) = 4$.

Hence, when fill is computed using the max rule, it appears that there is no necessary connection between fill levels and path lengths (where "length" indicates, as we use the term, the number of edges in a path). These observations suggest the need for a definition of *path length* that does not strictly depend on the number of edges in the path. Accordingly, we introduce the concept of *bifurcated length*, which is recursive in nature. In the following definition the phrase "unique fill subpaths" refers to the unique decomposition stated in Lemma 3.

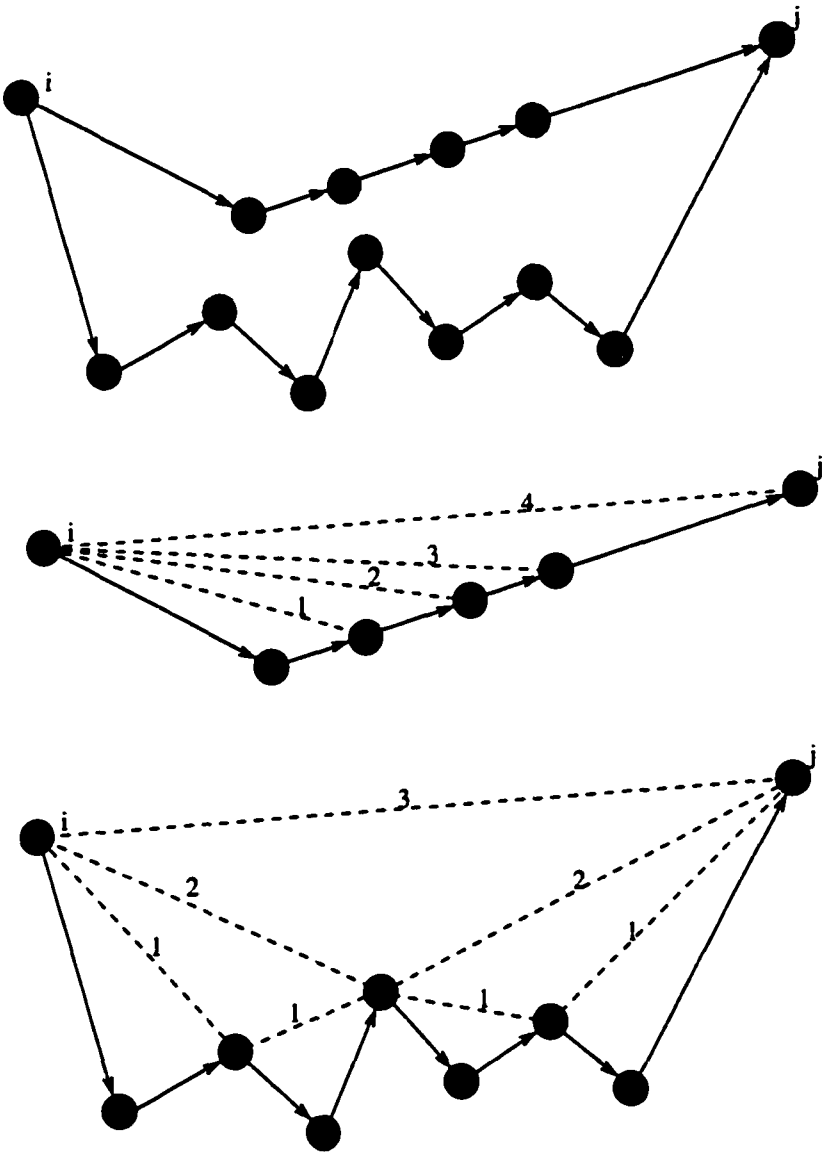


FIG. 4. *Bifurcated path lengths and edge counts. Top: a graph in which vertices i and j are connected by two fill paths. In the middle and bottom, the paths are shown separately, with bifurcated path lengths indicated by dashed lines. The path in the middle contains fewer edges but has a larger bifurcated length than the path at the bottom. Vertex ordering is indicated by vertical placement: vertices that are lower on the page are assumed to be ordered before vertices placed higher on the page.*

Definition 6 *A fill path containing a single edge has bifurcated length zero. A fill path containing two or more edges, whose unique fill subpaths have bifurcated lengths v and w, has bifurcated length $u = \max\{v, w\} + 1$.*

Heretofore, we have used the phrase "shortest fill path" to indicate, of all possible fill paths connecting two vertices in a graph, the (possibly nonunique) path containing the fewest number of edges. When discussing bifurcated lengths we use an analogous phrase, "fill path with shortest bifurcated length." This term indicates, of all possible fill paths connecting two vertices in a graph, the (possibly nonunique) path *whose bifurcated length is the smallest possible.*

A *chord* of a path is an edge that joins two non-consecutive vertices on the path. If an edge is added to a graph such that the shortest fill path $P(i, j)$ is chorded, the result will be that vertices i and j are joined by a shorter fill path than previously, and hence the corresponding S-level(i, j) will be reduced. This concept does not transfer to the study of bifurcated path lengths.

Observation 7 *A fill path may be chorded, and its bifurcated length unchanged.*

Figure 5 shows a fill path that contains 8 edges and has bifurcated length 4. After chording, the resulting shorter fill path contains only 7 edges, however, its bifurcated length is unchanged.

The next theorem provides a static characterization of M-level fill.

Theorem 8 *Let $G(A) = (V, E)$ be the graph of a square matrix A, and let $\langle i, j \rangle$ be a permitted edge in G_*^M. Then M-level$(i, j) = k$ if and only if there exists a fill path joining vertices i and j in $G(A)$ with bifurcated length k, and this path has the shortest bifurcated length amongst all fill paths between i and j.*

Remark. In contrast to Theorem 4, here there is no "+1" difference between bifurcated path lengths and M-levels. This is because the "+1" is incorporated into the definition of bifurcated path lengths.

Proof. If there is a fill path with shortest bifurcated length k joining i and j in $G(A)$, we prove the result, that an edge $\langle i, j \rangle$ with M-level$(i, j) = k$ exists in G_*^M, by induction on u, which is the bifurcated length of the fill path.

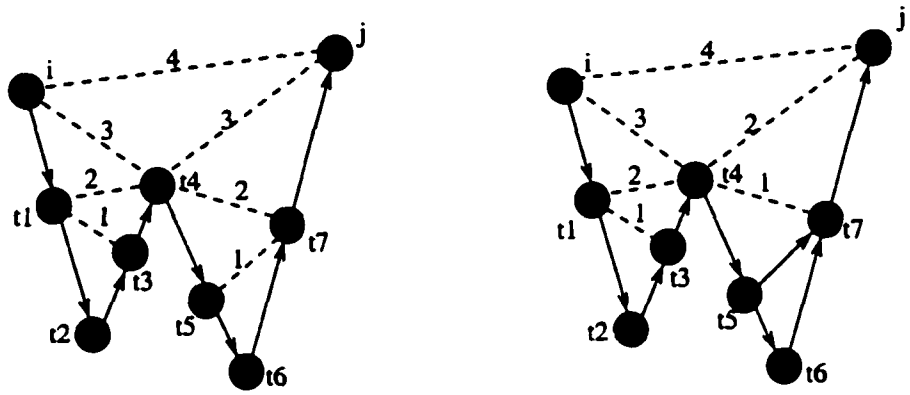


FIG. 5. *A fill path may be chorded, and its bifurcated length unchanged. Left: fill path* $P(i,j) = i, t_1, t_2, t_3, t_4, t_5, t_6, t_7, j$ *in* $G(A)$ *contains 8 edges and has bifurcated length 4. Right: the sub path* t_5, t_6, t_7 *has been chorded in* $G(A)$*; i and j are now connected by the shorter fill path* $P(i,j) = i, t_1, t_2, t_3, t_4, t_5, t_7, j$*. Ths path contains 7 edges, but the bifurcated length of* $P(i,j)$ *remains 4. Edges in* $G(A)$ *are drawn with solid lines. Edges in* G_*^M *are drawn with dashed lines, and labeled with their levels. Vertex ordering is indicated by vertical placement: vertices that are lower on the page are assumed to be ordered before vertices placed higher on the page.*

The base case $u = 0$ is immediate, since, by the construction in Section 2.4 and Definition 6, a path with bifurcated length zero corresponds to an original edge in $G(A)$.

Now assume the result is true for all fill paths with bifurcated length u less than k. We will prove that the result is true when the bifurcated length of a fill path is $u = k$.

Let $P(i, j)$ be a fill path with shortest bifurcated length that joins i to j in $G(A)$, and let the bifurcated length of this path be $u = k$. Let h be the highest-numbered interior vertex in this fill path. Then $P(i, h)$ and $P(h, j)$ are also fill paths by Lemma 3.

Let the bifurcated length of the fill path $P(i, h)$ be v and let the bifurcated length of the fill path $P(h, j)$ be w. By Definition 6, the bifurcated path length of $P(i, j)$ is $\max\{v, w\} + 1$, so the bifurcated lengths of v and w are both less than k. Note that either v or w (or both) is equal to $k - 1$. Without loss of generality, assume that w is less than $k - 1$. Then it must be that $v = k - 1$, and therefore the fill path $P(i, h)$ has the shortest bifurcated length possible.

Now suppose there is a path $P'(h, j)$ whose bifurcated length is less than w. Then we can freely replace the path $P(h, j)$ with the path $P'(h, j)$, and the bifurcated length of the path $P(i, j)$ will be unchanged.

Thus $P(i, j)$ is decomposable into two subpaths, $P(i, h)$ and $P(h, j)$, both of which are fill paths and have shortest bifurcated lengths less than k. Hence, the inductive hypothesis applies, so there exists a fill edge $\langle i, h \rangle$ with M-level v, and a fill edge $\langle h, j \rangle$ with M-level w. By the max level rule, when vertex i is eliminated, the fill edge $\langle i, j \rangle$ is created with M-level$(i, j) = \max\{v, w\} + 1 = k$.

Now we prove the converse. Suppose that $\langle i, j \rangle$ is a fill edge with M-level$(i, j) = k$ in G_*^M; we show the result that there exists a fill path $P(i, j)$ in $G(A)$ with shortest bifurcated length $u = k$ by induction on k, the edge's level.

The base case $k = 0$ is immediate, since, by the construction in Section 2.4, a fill edge with level zero corresponds to a fill path that contains a single edge, and by Definition 6 this path has bifurcated length zero.

Now assume the result is true for all fill edges with M-level less than k; we show it is also true for fill edges with M-level equal to k.

Assume the fill edge $\langle i,j \rangle$ with M-level$(i,j) = k$ is created, when vertex i is eliminated from G_i^M, by the previously existing edges $\langle i,h \rangle$ and $\langle h,j \rangle$.

Let the edge $\langle i,h \rangle$ have M-level$(i,h) = k_1$ and the edge $\langle h,j \rangle$ have M-level$(h,j) = k_2$. By the max rule for computing levels, we have that $\max\{k_1,k_2\}+1 = k$. Then both fill edges $\langle i,h \rangle$ and $\langle h,j \rangle$ have levels less than k, so the inductive hypothesis applies. Thus there exists a fill path that connects vertices i and h and has shortest bifurcated length $v = k_1$, and a fill path that connects vertices h and j and has shortest bifurcated length $w = k_2$. Additionally, either $k_1 = k-1$ or $k_2 = k-1$ or both. Without loss of generality, assume $k_1 = k-1$.

Now from Definition 6, the bifurcated length of the fill path $P(i,j)$ is

$$u = \max\{v,w\}+1 = \max\{k_1,w\}+1 = \max\{k-1,w\}+1 = k.$$

We also need to prove that the $P(i,j)$ fill path has the shortest bifurcated length amongst all fill paths connecting vertices i and j in $G(A)$. Suppose there were a path $P'(i,j)$ in $G(A)$ that had a shorter bifurcated length, that is, a bifurcated length u' less than k. From the first part of this proof, the edge $\langle i,j \rangle$ in G_*^M would then have an M-level less than k, which contradicts the premise that the fill edge $\langle i,j \rangle$ has M-level$(i,j) = k$. □

2.5.3 SIMILARITY OF S-LEVEL AND M-LEVEL FILL FOR MONOTONIC FILL PATHS

Some graphs have the property that an ILU(ℓ) factorization employing the sum rule computes factors identical to those computed when factorization employs the max rule. This property is an attribute, e.g., of graphs whose associated matrices arise from the discretization of partial differential equations on naturally ordered, structured grids, for factorization levels of three or less (we will say much more about this class of graphs in the following chapter). For these graphs, the shortest fill path connecting any two vertices i and j, and the fill path with shortest bifurcated length connecting the same two vertices i and j, are always identical when level$(i,j) \leq 3$. A consequence (which is the main result of this section) is that M-level(i,j) = S-level(i,j) for such cases. To capture and generalize the particular feature responsible for this consonance of level assignment, we define *1-alternating fill paths.*

As a preliminary, an *ascending path* is a path $(t_1, \ldots, t_k)$ that contains at least two vertices, with $t_h < t_{h+1}$ for $1 \leq h < k$. Similarly, a *descending path* is a path $(t_1, \ldots, t_k)$ that contains at least two vertices, with $t_h > t_{h+1}$ for $1 \leq h < k$.

Definition 9 *A fill path $P(i,j)$ is* 1-alternating *if it has one of the following forms.*
(i) A single edge, $\langle i, j \rangle$.
(ii) An edge $\langle i, h \rangle$ with $i > h$, concatenated with an ascending path $P(h, j)$.
(iii) A descending path $P(i, h)$ concatenated with an edge $\langle h, j \rangle$ with $h < j$.
(iv) A descending path $P(i, h)$ concatenated with an ascending path $P(h, j)$.

Note that forms (ii) and (iii) are restricted forms of form (iv). We call a 1-alternating path *internal-ascending* if it is either of form (*ii*), or consists of a single edge $\langle i, j \rangle$ with $i < j$. We call a 1-alternating path *internal-desending* if it is either of form (*iii*), or consists of a single edge $\langle i, j \rangle$ with $i > j$. Figure 6 illustrates the different species of 1-alternating fill paths, and the difference between 1-alternating and non-alternating fill paths.

By way of building up to this section's main result, and as an aid to intuition, several observations concerning properties of 1-alternating fill paths follow.

Observation 10 *A fill path is non-alternating if the path contains a sequence of interior vertices, $t_f, \ldots, t_g, \ldots, t_h$, such that $t_f < t_g$ and $t_g > t_h$.*

In the non-alternating path at the bottom right of Figure 6, $t_1 < t_2$ and $t_2 > t_3$.

Observation 11 *If $P(i,j) = i, t_1, t_2, \ldots, j$ is an internal-ascending fill path that contains at least three edges, and h is any interior vertex on the path with $h > t_1$, then $P(i, h)$ is also an internal-ascending fill path.*

The truth of this observation follows immediately from Definitions 1 and 9. Referring to the $P(i,j)$ fill path illustrated in the top left portion of Figure 6, this observation says that the paths $P(i, t_2)$ and $P(i, t_3)$ are internal-ascending fill paths. Note, however, that neither $P(t_1, j)$ nor $P(t_2, j)$ is a fill path. A similar observation holds for internal-descending paths.

The next observation is based on the fact that the highest numbered interior vertex of a 1-alternating path is necessarily adjacent to one of the end points of the path.

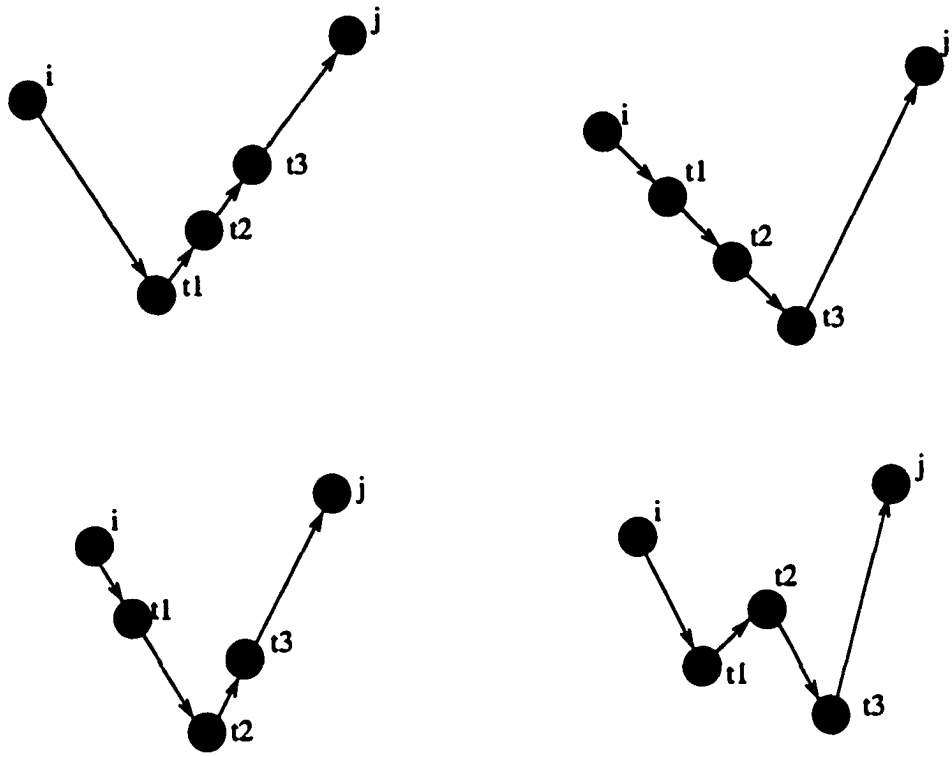


FIG. 6. *1-alternating and non-alternating fill paths. Top left: internal-ascending fill path. Top right: internal-descending fill path. Bottom left: internal-alternating fill path. Bottom right: non-alternating fill path. Here as elsewhere, vertical positioning of vertices is indicative of their relative orderings.*

Observation 12 *If $P(i,j)$ is a 1-alternating fill path containing $k+1$ edges, where $k \geq 1$, then the path can be uniquely decomposed into two 1-alternating fill paths $P(i,h)$ and $P(h,j)$. One of these fill paths will contain k edges, and the other a single edge.*

The existence and uniqueness of the decomposition was shown in Lemma 3. In that lemma's proof, we saw that the vertex h is necessarily the largest interior vertex on the $P(i,j)$ path. From Definition 9, this vertex is adjacent to either vertex i or vertex j, hence either $P(h,j)$ is a path containing a single edge, in which case the path $P(i,h)$ must contain k edges, or $P(i,h)$ is a path containing a single edge, in which case the path $P(h,j)$ must contain k edges. Referring again to Figure 6, the $P(i,j)$ fill path in the top left contains four edges, and can be decomposed into the fill paths (i,t_3) and (t_3,j), containing three edges and a single edge, respectively.

Observation 13 *Any fill path with three or fewer edges is 1-alternating.*

The truth for the one and two edge cases follows directly from definition 9. Now consider a fill path with three edges, i, t_1, t_2, j. Either $t_1 < t_2$, or $t_2 < t_1$; in either case the fill path is 1-alternating by Definition 9. As illustrated in the bottom right of Figure 6, paths with four or more edges are not necessarily 1-alternating.

Observation 14 *If $P(i,j)$ is any species of 1-alternating fill path, then*
(i) if $\langle h,t \rangle$ is an edge with $t > j$, then $P(i,t)$ is also a 1-alternating fill path;
(ii) if $\langle t,i \rangle$ is an edge with $t > i$, then $P(t,j)$ is also a 1-alternating fill path.

In top left of Figure 6, $P(i,t_3)$ is a 1-alternating fill path, and $\langle t_3,j \rangle$ is an edge. By this observation, $P(i,j)$ is therefore a 1-alternating fill path. This observation states a condition that permits a fill path to be extended while preserving its 1-alternating character. As such it is the complement of Observation 12, which says that any 1-alternating fill path can be decomposed.

Note that extending a 1-alternating path does not necessarily preserve any internal-descending or internal-ascending property it may possess. For example, if an internal-descending fill path $P(i,j)$ is extended by concatenation with an edge $\langle j,t \rangle$ with $t > j$, then the resulting fill path $P(i,t)$ is no longer internal-descending.

The following theorem establishes a relationship between path lengths and bifurcated lengths of 1-alternating fill paths.

Theorem 15 *Let $P(i,j)$ be a fill path that contains $k+1$ edges. The bifurcated length of $P(i,j)$ is k if and only if the fill path is 1-alternating.*

Proof. Suppose there exists a 1-alternating fill path that connects vertices i and j and contains $k+1$ edges. We prove the path has bifurcated length k by induction on k, the number of edges in the path.

The base case $k = 0$ is immediate since a fill path containing a single edge has bifurcated length zero by Definition 6. Now assume the result is true for all 1-alternating fill paths containing k or fewer edges; we show it is also true for 1-alternating fill paths containing $k+1$ edges.

Let h denote the highest numbered interior vertex on the path joining i and j. From Observation 12, h must be adjacent to either vertex i or vertex j. Without loss of generality, assume it is adjacent to vertex j.

Thus, $P(i,h)$ is a 1-alternating fill path containing k edges, and $P(h,j)$ is a 1-alternating fill path containing a single edge, so the inductive hypothesis applies to both subpaths.

By the inductive hypothesis, $P(i,h)$ has bifurcated length $k-1$, and $P(h,j)$ has bifurcated length zero. When these two paths are concatenated, the resulting path $P(i,j)$ has bifurcated length, by Definition 6, of

$$\max\{k-1,0\}+1 = k.$$

Now we prove the converse. Suppose vertices i and j are connected by a fill path that contains $k+1$ edges and has bifurcated length k. We show that the path is 1-alternating by induction on k, the number of edges in the path.

The base case $k = 0$ is immediate since a fill path containing a single edge is 1-alternating (Definition 6). Now assume the result is true for any fill path that contains j edges and has bifurcated length $j-1$, where $j \leq k$. We show the result is also true for paths that contain $k+1$ edges.

Let h denote the highest numbered interior vertex on the fill path joining i and j. Let m_1 be the number of edges in the $P(i,h)$ subpath, and m_2 the number of edges in the $P(h,j)$ subpath. Then $m_1 + m_2 = k+1$.

Let k_1 be the bifurcated length of the $P(i,h)$ subpath, and k_2 be the bifurcated length of the $P(h,j)$ subpath. Then $\max\{k_1,k_2\}+1 = k$. Hence, either $k_1 = k-1$ or $k_2 = k-1$ or both. Without loss of generality, suppose $k_1 = k-1$. Then from

Theorem 5, the $P(i,h)$ subpath must contain at least k edges, that is, $m_1 \geq k$. And since $m_1 + m_2 = k + 1$, it must contain exactly k edges, and m_2, the number of edges in the $P(h,j)$ subpath, must be 1.

Since $P(i,h)$ has bifurcated length $k-1$ and contains k edges, the inductive hypothesis applies, i.e, $P(i,h)$ is a 1-alternating fill path. Similarly, since $P(h,j)$ contains a single edge, and by definition 6 has bifurcated length zero, the inductive hypothesis applies.

Finally, by Observation 14, when the $P(i,h)$ path is concatenated with the $P(h,j)$ path, the resulting $P(i,j)$ fill path is 1-alternating. □

This chapter's final theorem formalizes the relationship between M-level and S-level fill that was alluded to in this Section's introduction.

Theorem 16 *Let $G(A) = (V,E)$ be the graph of a square matrix A, and let $\langle i,j \rangle$ be a permitted edge in G_*^M with M-level$(i,j) = k$. If the fill path with shortest bifurcated length that connects vertices i and j in $G(A)$ is 1-alternating, then M-level(i,j) = S-level(i,j).*

Proof. If $\langle i,j \rangle$ is a permitted edge in G_*^M with M-level k, and the fill path $P(i,j)$ with shortest bifurcated length is 1-alternating, then by Theorem 8 the path contains $k+1$ edges. By Theorem 5, there can be no shorter fill path (i.e, no fill path with fewer edges) connecting vertices i and j. Therefore, by Theorem 4, S-level$(i,j) = k$. □

CHAPTER 3

INCOMPLETE FILL PATH THEOREM APPLICATIONS

3.1 INTRODUCTION

This chapter contains a potpourri of algorithms and analyses that are logical consequences of the Incomplete Fill Path Theorem (Theorem 4). In sections 3.2 and 3.3 we present several algorithms that compute ILU(ℓ) structures by performing searches in the graph of a matrix. In section 3.4 we introduce the notion of *naturally ordered graphs*. In section 3.5 we show how these algorithms can be employed to permit computation of ILU(ℓ) storage requirements in space proportional to the number of rows in the matrix.

These graphs form the basis of the analyses in sections 3.6, 3.7, and 3.8, wherein we develop runtime complexity bounds and analyze fill densities. Finally, in section 3.9, we present an additional graph search algorithm that, for some classes of graph, has lower runtime complexity than either CLASSIC-ILU (Algorithm 1) or the graph search algorithms developed in earlier sections of this chapter.

3.2 COMPUTING UPPER TRIANGULAR STRUCTURES

In this and the following section we present new algorithms that compute ILU(ℓ) factors that are identical to those computed by CLASSIC-ILU. The new algorithms operate by using breadth first searches to find, per Theorem 4, shortest fill paths in $G(A)$ that contain at most $\ell + 1$ edges. Unlike CLASSIC-ILU, which requires the results of previously computed rows i to factor a current row j, our algorithms rely only on the static structure of $G(A)$, and hence have the novel feature that the structure of each row (or column) in the factor can be computed completely independently (i.e., in parallel).

GS-UROW, the subject of this section, computes ILU(ℓ) row structures for upper triangular factors. In graph theoretic terms, computing upper triangular row structures is equivalent to finding, for each vertex $i \in G(A)$, all vertices j such that $j > i$ and the vertices i and j are connected by a fill path containing $\ell + 1$ or fewer edges. For each such vertex j, an edge $\langle i, j \rangle$ is inserted in E', and $G(U) = (V, E')$. Our algorithms are presented in terms of adjacency list representations of graphs.

Inserting an element j in the adjacency list $adj(i)$ can be interpreted either as inserting an edge $\langle i, j \rangle$ in E', or as admitting the nonzero matrix entry f_{ij}.

(Concerning notation, we use $adj(i)$ to indicate edges in the graph $G(A) = (V, E)$. That is, $j \in adj(i)$ denotes the existence of the edge $\langle i, j \rangle \in E$. Similarly, we use $adj'(i)$ to indicate edges in $G(F)$, $G(U)$, or $G(L)$; $adj^T(i)$ to indicate edges in $G(A^T)$; and $adj^{T\prime}(i)$ to indicate edges in $G(F^T)$, $G(U^T)$, or $G(L^T)$.)

Consider a fill path $i, t_1, t_2, \ldots, t_k, j$, where $i < j$, and, per definition, $t_h < i$ for $1 \leq h \leq k$. Break this path into two subpaths: $P(i, t_k)$, which contains k edges, and $P(t_k, j)$, which contains a single edge. (Note that (i, t_k) is not necessarily a fill path.) GS-UROW's underlying design principle can be stated as follows. Given a graph $G(A)$, an initial vertex i, and a level ℓ, find all shortest paths $P(i, t_k)$ such that i is the largest vertex in the path, and the path contains ℓ or fewer edges. For each vertex t_k in each such path, if there also exists an edge $\langle t_k, j \rangle$ with $j > i$, insert the element (i, j) in the set of permitted fill (equivalently, insert j in $adj'(i)$, or, in matrix terminology, admit f_{ij} as a fill entry).

Now recall that breadth first search (BFS) finds a shortest path (a path containing the fewest number of edges) between a seed vertex i and any vertex t_h that is reachable from i (detailed descriptions of BFS can be found in many references; [23] is particularly readable). Thus, we can find the shortest subpaths $P(i, t_k)$ by performing BFS in the subgraph of $G(A)$ that is induced by the subset of vertices $\hat{V} = \{h | h \leq i\}$. (By a *subgraph induced by a subset of vertices* $\hat{V}$ we mean a graph $\hat{G} = (\hat{V}, \hat{E})$, where $\hat{E} = \{\langle u, v \rangle \in E | u, v \in \hat{V}\}$.) Finally, the vertices that terminate the fill paths are those vertices $j \in adj(t_k)$ and $j \in \hat{V}^C$, where $\hat{V}^C = \{j | j > i\}$. (Mnemonically, $\hat{V}^C$ is the "complementary" vertex set to the vertex set $\hat{V}$.)

Figure 7 contains a statement of the GS-UROW procedure. (Mnemonically, the name indicates that the algorithm performs graph searches (GS-) to compute upper triangular row-oriented structures (UROW).) This procedure, which is called once for every vertex $i \in V$ by GS-UPPER (Figure 8), takes as input a graph $G(A)$ in adjacency list representation; a level ℓ; a vertex i that corresponds to the row whose structure is sought; and an initially empty adjacency set, $adj'(i)$. When the procedure completes, $adj'(i)$ contains the structure of row i in U. (To reduce clutter, the *length*[] and *visited*[] arrays are assumed global in scope. The *visited*[] array is initialized once, in the GS-UPPER driver, discussed below.)

```
GS-UROW(G(A), ℓ, i, adj'(i))
1   # Initialization for BFS from vertex i
2   Q ← {i}
3   length[i] ← 0
4   visited[i] ← i
5   # BFS phase
6   while Q ≠ ∅
7       h ← Dequeue(Q)
8       for t ∈ adj(h) with visited[t] ≠ i
9           visited[t] ← i
10          if t < i and length[h] < ℓ
11              Enqueue(Q, t)
12              length[t] = length[h] + 1
13          if t > i
14              insert t in adj'(i)
```

FIG. 7. GS-UROW. *This procedure computes the structure of row i in the the factor's upper triangle. Here, as elsewhere in this work, indentation is used to demarcate code blocks.*

```
GS-UPPER(G(A), ℓ)
1   # Initialization
2   S ← ∅
3   for each vertex i ∈ V
4       visited[i] ← −1
5   # Compute structure of upper triangular row i
6   for each vertex i ∈ V
7       adj'(i) ← ∅
8       GS-UROW(G(A), ℓ, i, adj'(i))
9       insert adj'(i) in S
10  return S
```

FIG. 8. GS-UPPER *driver.*

GS-UROW uses a first-in, first-out queue that supports the standard ENQUEUE and DEQUEUE operations described in [23] and elsewhere. The queue is instantiated in Step 2 and seeded with a single vertex i. The value $length[j]$ indicates the length of a shortest path from the seed vertex i to the vertex j (Steps 3 and 12). The length of the path from a vertex to itself (Step 3) is defined as zero.

Vertices are marked with i when visited (Steps 4 and 9); in the interest of efficiency, this eliminates the necessity of having to mark every vertex as *unvisited* at the beginning of the procedure. (Were we to do so, the algorithm's runtime complexity would immediately be bounded below by $\Omega(n^2)$, where $|V| = n$, a situation we wish to avoid.)

The loop beginning in Step 6 employs BFS to search for previously unvisited vertices. Steps 11–13 discover the previously discussed (i, t_k) portions of potential fill paths. (We say "potential," since the paths are not necessarily part of any fill path.) Finally, Steps 13 and 14 discover the (t_k, j) portions of the fill paths. Due to the restriction "if $t > i$" in Step 14, an entry is only added to the adjacency list in Step 15 if the fill edge $\langle i, j \rangle$ is upper triangular.

The driver, GS-UPPER of Figure 8, performs global initializations in Steps 2–4. In Steps 6–9 an empty adjacency list is initialized for each vertex in the graph (row in the factor), and GS-UROW is called to compute the corresponding row structure. The algorithm returns the sparsity set S, which contains a set of adjacency lists that collectively represent the structure of the upper triangular factor.

3.3 COMPUTING LOWER TRIANGULAR STRUCTURES

For structurally unsymmetric matrices it is necessary to compute both lower and upper triangular factors. Our goal in this section is to design a symbolic factorization algorithm that operates similarly to GS-UROW in that it: (1) computes lower triangular row structures; (2) computes structures identical to those returned by CLASSIC-ILU.

To motivate interest in design goal (1), we note that in many scientific applications linear systems are developed by manipulating individual equations. For example, for two or three dimensional problems in the physical sciences, one or more equations is associated with each grid point. Each equation becomes a row in

a matrix, and since implementations frequently mimic the way we are accustomed to thinking (i.e, functions follow form), common practice is to develop, store, and pass the matrix to a linear solver through row-oriented data structures. In parallel implementations, each processor is typically assigned a rectangular section of a matrix, i.e., a contiguously numbered set of rows. Hence, efficient performance in the preconditioner application (triangular solve) phase requires that the L and U factors also be stored in a row-oriented format.

The first algorithm discussed in this section, GS-LOWER, is a driver that forms the graph $G(A^T)$, then calls GS-UROW. Unfortunately, the result is a column-oriented structure, hence, design goal (1) is not achieved. The second algorithm, GS-LROW, is a modified version of GS-UROW that permits computation of lower triangular row-oriented structures. However, we show that the returned adjacency lists may only contain a subset of the entries computed by CLASSIC-ILU. Hence, this algorithm fails to meet design goal (2). This prompts us to develop a *Multiple Search Rule*, which provides a means of modifying GS-LROW so that both design goals are achieved.

3.3.1 GS-LOWER

To find lower triangular fill edges we need to discover fill paths of the form $i, t_1, t_2, \ldots, t_k, h$, where $i > h$. The simplest way to do this is to "mentally reverse" the directed edges and search instead for fill paths of the form $h, t_k, \ldots, t_2, t_1, i$. But that is precisely what GS-UROW does. To make use of our previous algorithm we can form $G(A^T) = (V, E^T)$. This graph's edgeset contains a directed edge $\langle j, i \rangle$ for every nonzero entry a_{ij}. Analogously, in adjacency list representation, $adj^T(j)$ contains an entry i if $j \in adj(i)$.

Figure 9 presents the GS-LOWER driver that calls GS-UROW to compute lower triangular structures. The algorithm forms the transpose of the input graph in Steps 3–11. (Since an adjacency list representation of $G(A)$ can be interpreted as a representation of A, we use the term "transpose of the graph" as shorthand for "the graph of the transpose of the matrix.") Steps 13–16 invoke GS-UROW, which computes lower triangular column structures of L.

This algorithm computes structures L that are identical to those computed by CLASSIC-ILU. It has the drawbacks that (1) the transpose of the input must be

```
GS-LOWER(G(A), ℓ)
1   # Initialization
2   S ← ∅
3   # Form G(A^T)
4   G(A^T) ← ∅
5   for each vertex i ∈ V
6       visited[i] ← −1
7       adj^T(i) ← ∅
8       insert adj^T(i) in G(A^T)
9   for each vertex i ∈ V
10      for j ∈ adj(i)
11          insert i in adj^T(j)
12  # Compute structure of lower triangular column i
13  for each vertex i ∈ V
14      adj^T'(i) ← ∅
15      GS-UROW(G(A^T), ℓ, i, adj^T'(i))
16      insert adj'(i) in S
17  return S
```

FIG. 9. GS-LOWER.

computed; (2) if the factor is required in row-oriented format, the transpose of the output must also be computed.

3.3.2 GS-LROW

To design an algorithm that directly computes lower triangular factors in row-oriented format we use reasoning similar to that described in section 3.2 for GS-UROW. Given a graph $G(A)$, an initial vertex i, and a level ℓ, consider fill paths $i, t_1, t_2, \ldots, t_k, h$, where $i > h$, as composed of two subpaths, $P(i, t_k)$, containing ℓ or fewer edges, and $P(t_k, h)$, containing a single edge. (Again, note that $P(i, t_k)$ is not necessarily a fill path.) As before, we use BFS to find all shortest paths $P(i, t_k)$ that contain ℓ or fewer edges in the subgraph of $G(A)$ induced by the vertex set,

```
GS-LROW(G(A), ℓ, i, adj'(i))
1   # Initialization for BFS from vertex i
2   Q ← {i}
3   length[i] ← 0
4   secondLargest[i] ← −1
5   visited[i] ← i
6   # BFS phase
7   while Q ≠ ∅
8       h ← Dequeue(Q)
9       for t ∈ adj(h) with visited[t] ≠ i
10          visited[t] ← i
11          if t < i and length[h] < ℓ
12              Enqueue(Q, t)
13              length[t] = length[h] + 1
14              secondLargest[t] = max{secondLargest[h], t}
15          if t > secondLargest[h]
16              insert t in adj'(i)
```

FIG. 10. GS-LROW.

$\hat{V} = \{h | h \leq i\}$.

Previously, we discovered the single-edged subpaths by looking for terminating vertices $j \in \hat{V}^C$ (recall that $\hat{V}^C = \{j | j > i\}$). Here, however, the initial vertex is also the highest numbered vertex in any fill path, so all vertices in the fill paths are contained in the vertex set $\hat{V}$, hence we must conduct our entire search in the subgraph induced by $\hat{V}$. This leads to the design principle: concatenate a $P(t_k, h)$ single-edged subpath to a $P(i, t_k)$ subpath, *only if h is larger than the second largest vertex in the* $P(i, t_k)$ *path.*

This added requirement, that we keep track of the second largest vertex in each $P(i, t_k)$ path, is the motivation behind GS-LROW (Figure 10). This procedure is similar in operation to GS-UROW. As before, a queue is instantiated and seeded

with a single vertex i (Step 2), and we record the length of a path from the seed vertex i to a vertex j in $length[j]$ (Steps 3 and 13). We also record the numbering of the second largest vertex in the path $P(i,j)$ in the variable $secondLargest[j]$ (Steps 4 and 14).

The information about the second largest vertex is used in Step 15. This step ensures that t is only inserted in $adj'(i)$ if there is a path whose terminating vertices (i and t) are larger than any vertices interior to the path.

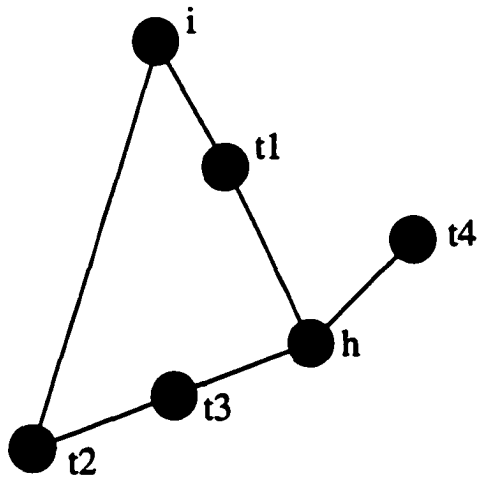
While GS-LROW returns a list of fill edges corresponding to shortest fill paths, as we now explain it can fail to discover some edges that "should" be included. By "should" we mean, with respect to our design goal (2), edges that *would* be included in a CLASSIC-ILU computation. Figure 11 illustrates what can go wrong. As amplified in the following discussion, fill edge $\langle i,h\rangle$ is not discovered since, even though the there exists a fill path i, t_2, t_3, h, the vertex h is first visited by a traversal of the shorter path, i, t_1, h, which is not a fill path. Similarly, fill edge $\langle i,t_4\rangle$ is not discovered since the shortest path from vertex i to t_4 is not a fill path.

The following discussion is summarized in the table in the lower half of the figure. Parentheses around entries in the columns titled *SecondLargest* and *Length* indicate read operations; entries without parentheses indicate write operations.

Assume we are performing a level three search from vertex i in the graph illustrated in Figure 11. That is, we are attempting to discover all shortest fill paths that contain four or fewer edges. Initially vertex i is enqueued (Step 1). When this vertex is dequeued (Step 2), vertex t_1 is discovered (Step 3). Since t_1 has not been previously visited, it is enqueued and the subsidiary information recorded: $length[t_1] \leftarrow 1$, which is the number of edges in the path $i, \ldots, t_1$; and $secondLargest[t_1] \leftarrow t_1$, indicating that t_1 is the second highest numbered vertex in this path. The fill edge $\langle i,j\rangle$ is also added to the sparsity set, since t_1 is larger than the second highest numbered vertex in the path $P(i,i)$. Vertex t_1 is compared against the *secondLargest* value, -1, that was read when i was dequeued.

Similarly, vertex t_2 is discovered, enqueued, and edge $\langle i,t_2\rangle$ added to the sparsity set (Step 4). Note that the ordering of Steps 3 and 4 is arbitrary; it would make no difference if t_2 were discovered before t_1.

Next, t_1 is dequeued (Step 5). Since the graph shown is undirected, vertex i is contained in $adj(t_1)$. Since vertex i was previously visited, no action will be taken.



Step	Vertex	Previously Visited?	Enqueued?	SecondLargest	Length	Fill Edge Added
1.	i		yes			
2.	dequeue i			(-1)	(0)	
3.	t1	no	yes	t1	1	$\langle i, t_1 \rangle$
4.	t2	no	yes	t2	1	$\langle i, t_2 \rangle$
5.	dequeue $t1$			(t1)	(1)	
6.	h	no	yes	t1	2	
7.	dequeue $t2$			(t1)	(1)	
8.	t3	no	yes	t3	2	
9.	dequeue h			(t1)	(2)	
10.	t4	no	yes	t1	3	
11.	dequeue $t3$			(t1)	(2)	
12.	h	yes	no			

FIG. 11. *Incompleteness of* GS-LROW. *When graph at top of picture is input to* GS-LROW, *neither the fill edge* $\langle i, h \rangle$ *that corresponds to the path* i, t_2, t_3, h, *nor the fill edge* $\langle i, t_4 \rangle$ *that corresponds to the path* i, t_2, t_3, h, t_4, *will be discovered. See text for additional discussion.*

(To reduce clutter, this and several similar steps have been omitted from the table.)

Searching from t_1 we discover vertex h, which is enqueued (Step 6), and subsidiary information written indicating that t_1 is the largest vertex in the path $i, \ldots, t_1, h$, which has length three. Since h is *not* larger than the second largest vertex in the path $i, \ldots, t_1$, no fill edge is discovered. Again, h is compared against the *secondLargest* value, t_1, that was read when t_1 was dequeued.

Similarly, vertex t_2 is dequeued (Step 7) and vertex t_3 subsequently enqueued (Step 8). Again, the ordering of steps 5–6 and steps 7–8 could be exchanged with no effects.

Next, vertex h is dequeued (Step 9), and the smallest vertex in the path $i, \ldots, h$ is read as t_1. Next, t_4 is encountered (Step 10). Since t_1 is larger than t_4, we know that $i, \ldots, h, t_4$ is not a fill path, so no fill edge is added. Since t_4 was not previously visited, it is enqueued.

Things go wrong after vertex t_3 is dequeued (Step 11). When h is encountered (Step 12), we "should" discover fill edge $\{i, h\}$, since h is larger than the second largest vertex in the path $i, \ldots, t_3$. However, since h was previously visited the $\langle i, h \rangle$ edge is not discovered. Additionally, since h has already been enqueued (when it was previously visited), it is not enqueued again. As a result, the fill path $i, \ldots, t_3, h, t_4$ will not be discovered.

This example suggests that need to modify GS-LROW to permit a vertex to be enqueued more than once. We need an algorithm whose operation encompasses the following *Multiple Search Rule*. This rule is illustrated in Figure 12.

> *Multiple Search Rule*: a vertex h should be re-queued if it was initially discovered via a path $P(i, h) = i, \ldots, g_1, \ldots, h$, where g_1 is the second largest vertex in the path $P(i, h)$, and on a subsequent visit it is discovered via a path $P'(i, h) = i, \ldots, g_2, \ldots, h$, where g_2 is the second largest vertex in the path $P'(i, h)$, and $g_2 < g_1$.

We can modify GS-LROW to take account of the Multiple Search Rule quite simply, by altering the conditional in Step 9. The revised formulation reads

```
for t ∈ adj(h)
    if visited[t] ≠ i or secondLargest[h] < secondLargest[t]
        (process t as before).
```

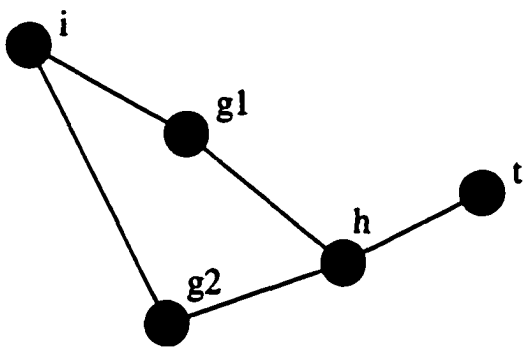


FIG. 12. *Multiple Search Rule. A vertex h should be re-queued if it was initially discovered via a path* $P(i,h) = i,\ldots,g_1,\ldots,h$, *where* g_1 *is the second largest vertex in the path* $P(i,h)$, *and on a subsequent visit it is discovered via a path* $P(i,h)' = i,\ldots,g_2,\ldots,h$, *where* g_2 *is the second largest vertex in the path* $P(i,h)'$, and $g_2 < g_1$. *Without this rule, the fill path* i, g_2, h, t *would not be discovered.*

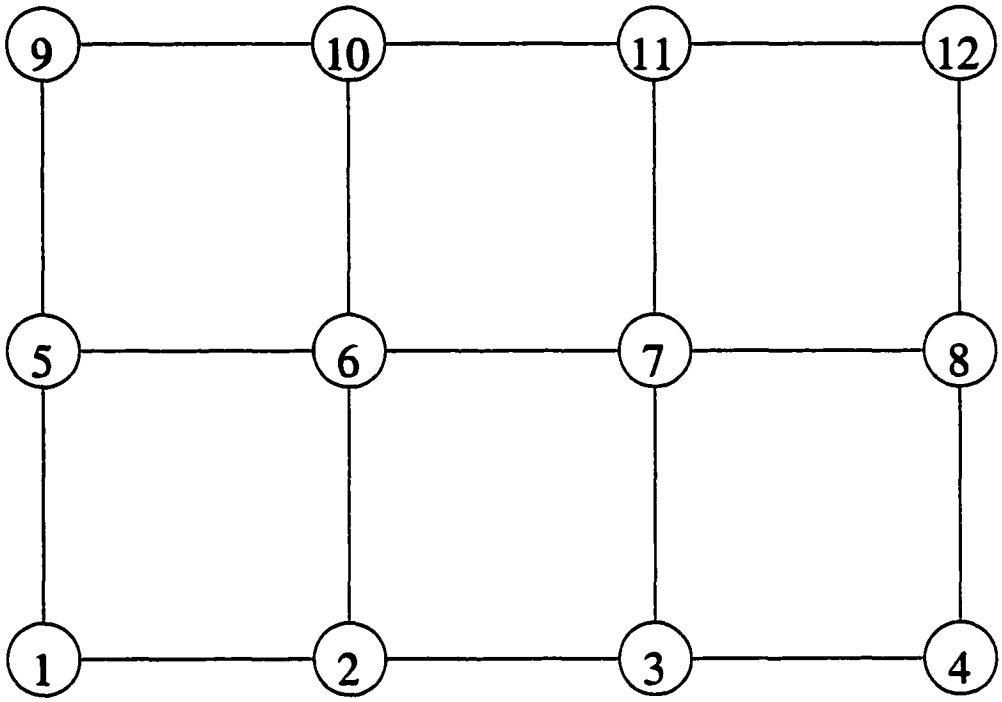


FIG. 13. *A curious object. This picture can be interpreted either as a grid (the vector x in the system $Ax = b$) or the graph of a matrix (the matrix A in the system $Ax = b$). If interpreted as a grid, the numbers indicate the gridpoint's natural ordering.*

3.4 GRAPHS AND GRIDS

In this section we introduce some terminology in order to clarify the distinction between graphs and grids. The object pictured in Figure 13 can represent either a grid (i.e., a collection of unknowns, each of which is associated with a Cartesian coordinate), or a graph of a matrix (i.e., a set of vertices and a set of edges). When the object is interpreted as a grid it represents the left-hand side vector in the system $Ax = b$. When the object is interpreted as the graph of a matrix it represents the matrix A in the system $Ax = b$.

For our purposes the connection between grid and matrix is that, if one starts with a 2D grid, then discretizes a PDE using a five-point stencil, then draws pictures of the grid and the graph, the pictures are identical, even though the underlying

mathematical and computational objects are quite different. That is, the grid corresponds to a vector, and the graph corresponds to a matrix.

It is common to speak of *naturally ordered grids* meaning, informally, that the gridpoints have been numbered from left to right and bottom to top. Natural ordering can be described more formally by reference to the Cartesian coordinate associated with each gridpoint. (In fact, this gives an easy way to extend the concept of natural ordering from structured to unstructured grids.) Although graphs, being sets of vertices and edges, have no attached spatial coordinates, it is useful to preserve the concepts of "natural order" and "spatiality" for matrices that derive from the discretization of PDEs on grids.

Henceforth, we use the term *structured graph* to refer to a graph of a matrix that arose from the discretization of a PDE or set of PDEs on a structured grid; by extension, we refer to the corresponding matrix as a *naturally ordered matrix*. The term *naturally ordered structured graph*, which we shorten to *naturally ordered graph*, indicates that the gridpoints were naturally ordered. These graphs are structurally symmetric (undirected), although their associated matrices may be numerically unsymmetric.

3.5 ILU(ℓ) MEMORY ALLOCATION

As noted in the previous chapter, it is possible to compute storage requirements for complete LU factors of symmetric matrices in time essentially proportional to the number of nonzeros in the factor and space proportional to the number of matrix rows. Hence, all needed data structures can be allocated and set up before factorization begins. Unfortunately, there is no known equivalent procedure for predicting ILU storage requirements. One practice is to guess at the number of nonzeros in the factor and initially allocate that much storage; if this proves insufficient, the factorization fails. In some ILU schemes, such as ILUT, an arbitrary limit is set on the number of nonzeros in each row. This ensures that adequate storage will be allocated and, unless a zero-pivot is encountered, the factorization will succeed. Another approach, for implementations coded in C or C++, is to dynamically reallocate storage when the initial guess is insufficient. However, this reallocation strategy can incur non-trivial overhead and can also fragment memory.

While we cannot offer a general solution to the storage allocation prediction

problem for ILU factorization, our graph-search algorithms can easily be modified to compute storage requirements for ILU(ℓ) factors using $O(n)$ space. The modification for GS-UROW (Figure 7) is accomplished as follows. Initialize a counter to zero. Change Step 15, which previously inserted an element in an adjacency list, to increment the counter. Return the counter's value. Similar modifications can be made to GS-LROW.

While these modifications permit computation of a factor's storage requirements in $O(n)$ space, the time complexity is identical to that required for actually performing symbolic factorization. It is an open question whether faster methods for computing ILU storage requirements can be devised.

Experimentally, we can use our revised procedures to provide a qualitative feel for the amount of fill likely to be generated for various matrix classes, orderings, and levels.

The fill density, which we denote as ρ, is the ratio of the number of nonzeros in F to the number of nonzeros in A. This density is an indication of memory requirements and—ignoring memory hierarchies, communication costs, etc.—can be interpreted as the amount of work required for preconditioner application.

Figure 14 shows how density grows as a function of level for 2D and 3D (five-point and seven-point) structured graphs. Two characteristics, which are common to all medium to large scale matrices we have examined, are apparent. First, fill amounts vary smoothly with level; there are no points in the graphs where fill ratios make unexpectedly large "jumps" between one factorization level and the next. Second, even in this small example, there tends to be a great many levels before complete factorization is achieved.

3.6 GS-UROW COMPLEXITY

In this section we formulate run time complexity bounds for GS-UROW for any graph whose vertices have bounded degree; these bounds are not tight. We also formulate tight run time complexity bounds for 2D and 3D structured graphs. The bounds are cast as functions of ℓ, the factorization level, and n, the number of vertices in the graph.

The runtime cost of a single iteration of GS-UROW is the cost of conducting a BFS from a vertex i. In general, the runtime cost of BFS on a graph $G = (V, E)$

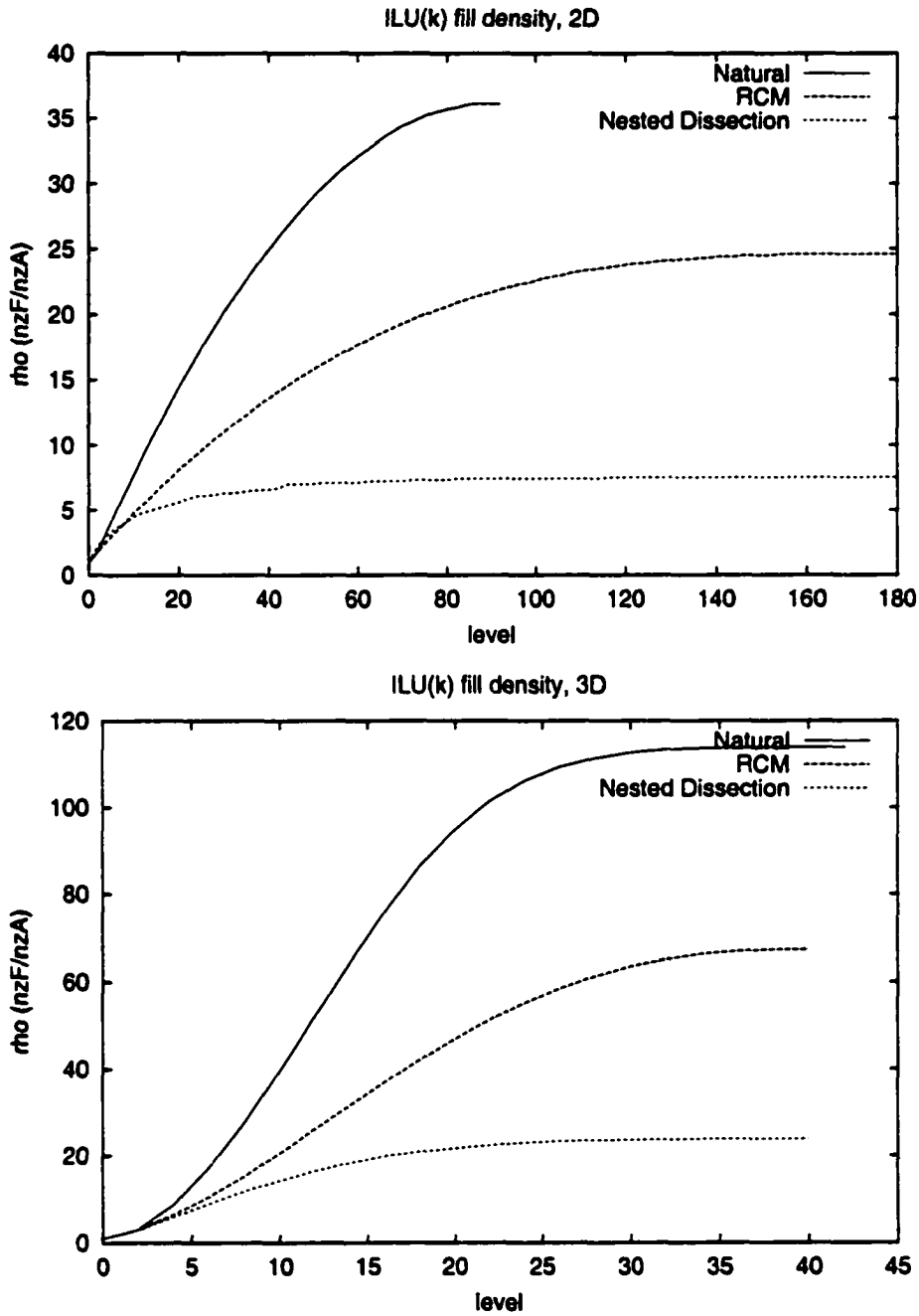


FIG. 14. *Fillin densities for naturally ordered structured graphs. Data is from 2D* 90×90 *graph with 8,100 unknowns (top), and a 3D* $20 \times 20 \times 20$ *graph with 8,000 unknowns (bottom).*

is known to be $O(V + E)$, that is, BFS runs in time linear in the size of the adjacency-list representation of G [23].

What is needed for our analysis is an expression for the number of vertices and edges "touched" during a level ℓ BFS from a vertex i. We first consider a general bound for any graph whose vertices have bounded degree c (e.g., for 2D structured graphs $c = 5$; for 3D structured graphs $c = 7$.) When the initial seed vertex i is dequeued each of c adjacent vertices are examined. If all of these vertices are numbered less than i (which is certainly possible), all will be enqueued. At this point we have completed a level zero search (i.e., we have examined all paths of length one), at a cost of $O(c)$.

During the level one phase of the search each of possibly c vertices is dequeued, and for each of these we examine and enqueue at most c vertices. Thus, the cost for the level one phase is bounded by $O(c^2)$. During the level two phase of the search each of possibly c^2 vertices is dequeued, and for each we again examine and enqueue at most c vertices, so the cost for this phase is bounded by $O(c^3)$. Continuing in this repetitious line of reasoning, we conclude that the cost for a level ℓ search from a single vertex is bounded by $O(c^{(\ell+1)})$. For a matrix with n rows and columns, n breadth-first searches are conducted, so the total runtime cost is bounded by $O(nc^{(\ell+1)})$. In practice this bound is not at all tight since when a vertex i is dequeued typically some or all of the c vertices in its adjacency list will previously have been enqueued (visited).

Next, consider a 2D structured graph. As shown in Figure 15, it is easy to devise a formula for counting the number of vertices that are within a distance ℓ from a seed vertex i. Since there are exactly 4ℓ vertices at distance ℓ from a seed vertex i, the expression for the total number of vertices reachable by ℓ or fewer edges from i (i.e., all the vertices that will be enqueued during BFS) is the sum

$$1 + \sum_{k=1}^{\ell} 4k = 1 + 2\ell(\ell + 1). \tag{6}$$

Since at most four edges will be examined when each vertex is dequeued, and n searches are conducted, we arrive at the runtime complexity $O(n(\ell+1)^2) \approx O(n\ell^2)$. (The "+1" is needed since, when vertices at a distance ℓ are dequeued, no additional vertices are enqueued. However, all vertices adjacent to the dequeued vertices are examined, and entries corresponding to fill path lengths of $\ell + 1$ may be added to

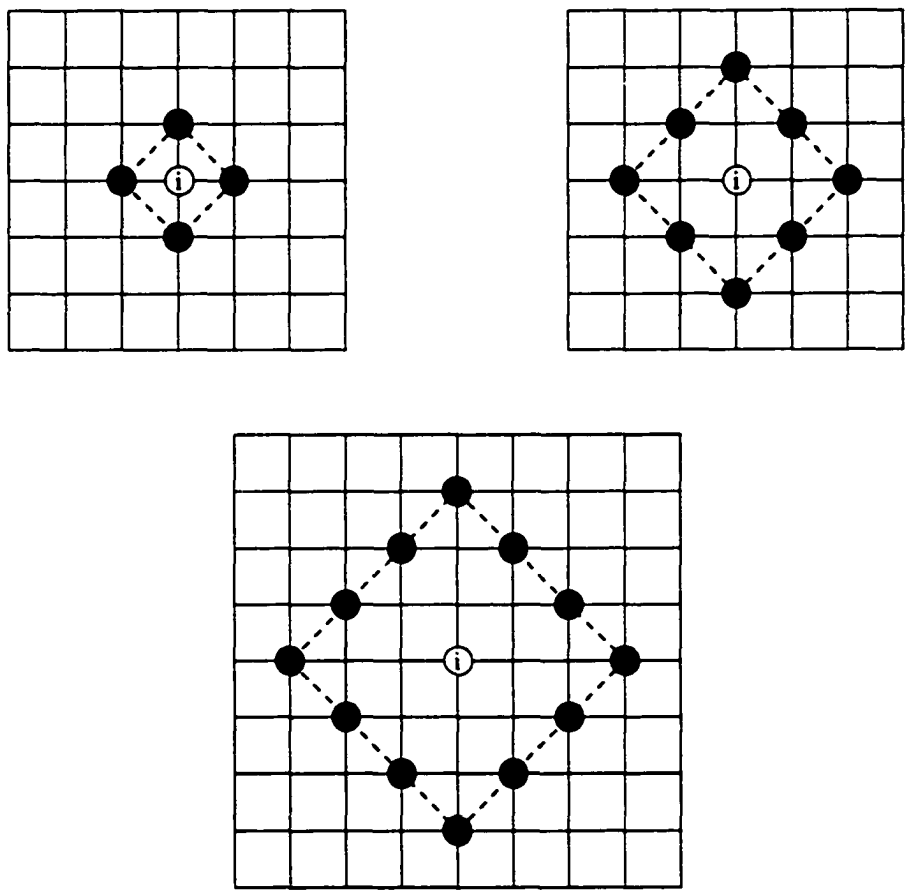


FIG. 15. *Number of vertices at distance ℓ from seed vertex i. Number of vertices at distance ℓ from seed vertex i in a 2D structured graph. Cases for $\ell = 1$, 2, and 3 are shown left to right and top to bottom. The dashed lines are added as an aid to the eye in discerning the patterns. We conclude from studying the patterns that there are 4ℓ vertices at ℓ edges distant from a given vertex i.*

TABLE 1
Symbolic factorization complexity upper bounds.

Algorithm	2D	3D	bounded degree c	general
CLASSIC-ILU	$O(n\ell^2)$	$O(n\ell^4)$	$O(nc^{2(\ell+1)})$	$O(nc^{2(\ell+1)})$
GS-UROW	$O(n\ell^2)$	$O(n\ell^3)$	$O(nc^{(\ell+1)})$	$O(ne^*)$

the sparsity set.)

For 3D, the number of vertices at a distance ℓ from a seed vertex i is

$$2 + 4\ell + \sum_{k=1}^{\ell-1} k. \tag{7}$$

Geometrically, these vertices form the faces of two pyramids joined at their bases (a rendering of which is beyond the artistic capabilities of this author). The total number of vertices within ℓ edges from i

$$\sum_{j=1}^{\ell} \left(2 + 4j + \sum_{k=1}^{j-1} k \right). \tag{8}$$

Since a search is conducted for each vertex in the graph, we arrive at a runtime complexity of $O(n\ell^3)$.

In all three cases (bounded degree; 2D structured; 3D structured), if the graphs are arbitrarily ordered, there may be a fill edge between vertex i and any dequeued vertex. Hence, the number of fill entries in any row of the matrix is bounded by same complexity as the search cost from the corresponding vertex. As shown in the next section, these bounds are not tight.

Finally, consider an arbitrary unsymmetric graph, about which we have no a priori structural information. The cost of the BFS from each vertex is bounded by the maximum of the number of vertices and edges in the matrix. Assuming the graph is connected, this gives an upper bound of $O(ne)$, where e is the number of edges in the matrix.

Table 1 summarizes the results in this section. The table also includes complexity results for CLASSIC-ILU, which are developed in Section 3.8.

3.7 FILL DENSITIES FOR NATURALLY ORDERED GRAPHS

For a 2D naturally ordered graph the analysis in the previous section shows that the number of nonzeros in any row in the matrix is bounded by $O(\ell^2)$. However, in Figure 23 in the following chapter we use geometrical reasoning to show that there are asymptotically 2ℓ nonzeros in the strict upper triangular portion of the matrix rows. Due to structural symmetry, there are also 2ℓ nonzeros in the strict lower triangular portion of matrix rows. Similar geometrical arguments show that there are asymptotically ℓ^2 nonzero entries in the strict upper and lower triangular portions of naturally ordered 3D graphs.

These results give us a theoretic means of estimating the fill density ρ. Asymptotically, rows in A contain five nonzeros. Asymptotically, rows in F contain $1+4\ell$ nonzeros (a diagonal entry, plus 2ℓ entries in the strict upper and lower portions of the row).

On the assumption that all matrix rows contain the same number of nonzero entries, ρ can be defined as the ratio of the number of nonzero entries in a row in F to the number of nonzero entries in a row in A. For 2D naturally ordered graphs this is

$$\frac{1+4\ell}{5}. \tag{9}$$

For 3D naturally ordered graphs the corresponding expression is

$$\frac{1+2\ell^2}{7}. \tag{10}$$

In Figure 16 we have plotted the predicted fill densities, and the actual fill densities for several sizes of 2D naturally ordered graphs. As the graphs get larger the actual densities more closely approach the predicted density. When fill levels are high, the average number of nonzeros per row in F progressively decreases, due to the grid boundaries (this explains the fall-off on the right-hand side of the plot). Were we dealing with a torus (a 2D grid with wrap-around on the sides and edges), this fall off would not occur.

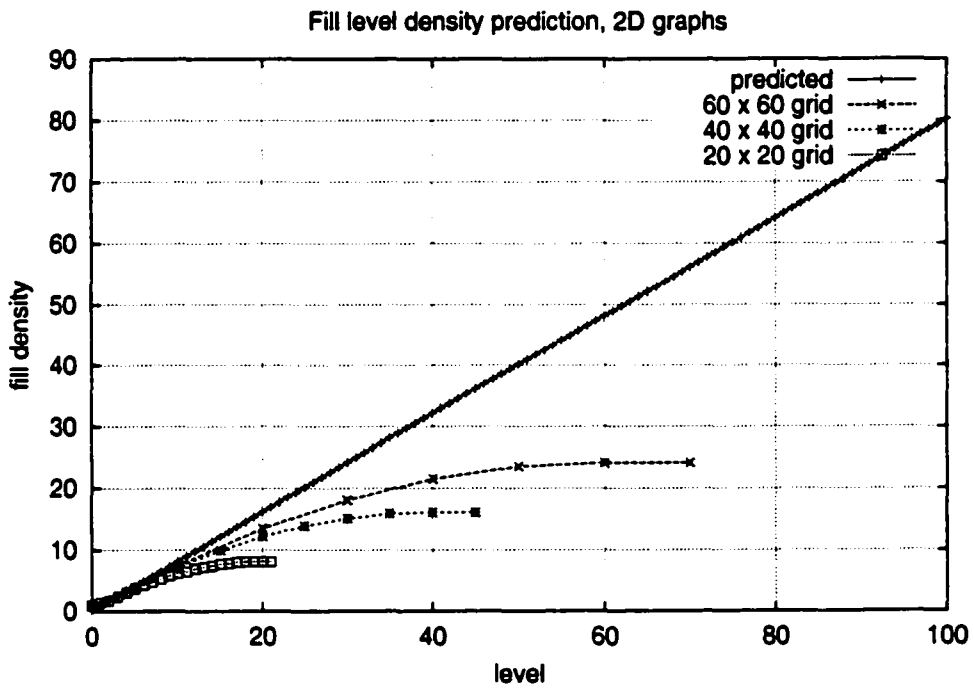


FIG. 16. *Predicted vs. actual fill densities for 2D naturally ordered graphs.*

3.8 CLASSIC-ILU COMPLEXITY

A statement of the CLASSIC-ILU algorithm appeared in chapter 2, Figure 1. CLASSIC-ILU's runtime complexity is determined by Steps 8 and 9. For every entry i in $adj'(j)$ with $i < j$ (i.e., elements in the lower triangular portion of row j), we "touch" every element k in $adj'(i)$ with $k > j$ (i.e., elements in the upper triangular portion of row i) (This analysis is equally applicable to complete as well as incomplete factors). If the upper and lower triangular portion of each row contains e entries, the run time complexity is thus $O(ne^2)$.

In Section 3.6 we showed that, for structurally symmetric graphs of bounded degree c, runtime complexity for GS-UROW $O(nc^{(\ell+1)})$. Since any of the vertices touched during a breadth first search can potentially result in fill entries, we conclude that the number of fill entries in either upper or triangular sections of the rows is bounded by $e = O(c^{(\ell+1)})$. Hence, the runtime complexity of CLASSIC-ILU is bounded by $O(nc^{2(\ell+1)})$.

For 2D naturally ordered grids we have shown that $e = 2\ell$. Hence, the runtime complexity is $O(n\ell^2)$. This is identical to the run time complexity derived for GS-UROW in section 3.6.

For 3D naturally ordered grids $e = O(\ell^2)$. Hence, the runtime complexity is $O(n\ell^4)$. This is of higher order than the $O(n\ell^3)$ run time complexity for GS-UROW that was derived in section 3.6.

Runtime complexity analyses for more general matrices, structure we have no a priori information is elusive. In general, we can only say that complexity is bounded by that for complete factorization. This has been shown to be $O(ne^*)$, where e^* is the number of edges in the filled graph $G(F)$ [72].

These complexity results are summarized in Table 1. The table also contains complexity results (developed in Section 3.6) for the GS-UROW Algorithm.

3.9 POTENTIAL-U GRAPH SEARCH ALGORITHM

For our final application we introduce a graph search algorithm that, for some classes of matrices, computes factors identical to those returned by GS-UROW, but with lower runtime complexity. The new algorithm, called POTENTIAL-U, is a logical extension of the ideas presented in previous sections of this chapter. Instead

of conducting a level ℓ search on the unrestricted graph $G(A_L)$ as does GS-UROW, POTENTIAL-U restricts its search by enqueuing only vertices that, on dequeue, are guaranteed to result in the discovery of (possibly previously discovered) fill edges.

POTENTIAL-U always computes a set S that is a subset of that computed by GS-UROW. For the structured graphs that were introduced in Section 3.4 the two sets are always identical. For all other classes of graphs the sets are identical for $\ell = 0, 1$, or 2. For higher levels, the set of edges returned by POTENTIAL-U may be only a proper subset of that returned by GS-UROW. (We will state these properties as theorems after introducing the algorithm). Hence, the two fill sets are only "potentially" identical, from which observation the algorithm takes its name. Before presenting the algorithm, we briefly discuss the observations and intuition behind its design.

Figure 17, top, shows all vertices visited, edges traversed, and fill edges discovered during the ith iteration of GS-UROW(G(A), 5), when $G(A)$ is a 2D naturally ordered graph. The bottom half of the figure indicates the vertices and edges that are part of the shortest fill paths associated with admitted fill entries; these are the vertices and edges that represent "useful" work. Two observations are apparent from inspection of these pictures. First, a great many of the vertices that are enqueued are not part of any of the fill paths associated with admitted edges. Visiting these vertices is hence "useless" work, which we would like to avoid if we could. Second, every vertex h that is part of a shortest fill path is adjacent to a vertex j such that $\langle i, j \rangle$ is an admitted edge.

These observations suggest a simple method of restricting the BFS phase of GS-UROW in order to avoid most of the "unnecessary" work. We introduce the *graph search constraint*: when conducting a BFS from vertex i, a vertex h is only enqueued if there exists an edge $\langle h, j \rangle$ in $G(A)$, and $h < i \leq j$. This constraint is implemented in the form of an $O(1)$ lookup table that is initialized in Steps 2 through 4 of the POTENTIAL-DRIVER, shown in Figure 18. This driver, like the driver for GS-UROW, calls the POTENTIAL-U algorithm once for each vertex in the graph. The POTENTIAL-U algorithm itself is identical to the GS-UROW algorithm

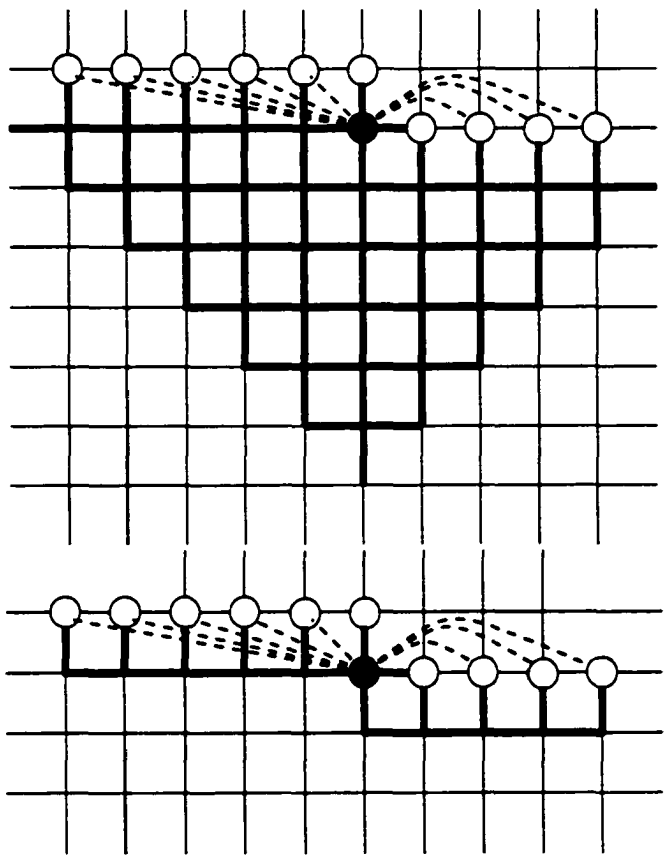


FIG. 17. *Vertices visited during* GS-UROW. *Top: vertices and edges visited during* GS-UROW, *level 5. All end point vertices of the bold edges are queued during the search. Bottom: the fill edges discovered during the search are indicated with dashed lines. The edges that actually participate in the shortest fill paths that lead to the discovered edges are bolded.*

```
POTENTIAL-DRIVER(G(A), ℓ)
1   # Initialization
2   S ← ∅
3   for each vertex i ∈ V
4       largestNabor[t] ← max{j : j ∈ adj(i)}
5       visited[i] ← −1
6   # Compute structure of upper triangular row i
7   for each vertex i ∈ V
8       adj'(i) ← ∅
9       POTENTIAL-U(G(A), ℓ, i, adj'(i))
10      insert adj'(i) in S
11  return S
```

FIG. 18. POTENTIAL-U *driver.*

(Figure 7), except that line 10, which formerly read:

if $t < i$ and $length[h] < \ell$

is replaced by

if $t < i$ and $length[h] < \ell$ and $largestNabor[t] \geq i$.

We next state and prove three simple theorems concerning the output of POTENTIAL-U.

Theorem 17 *For any graph G the set of fill edges returned by* POTENTIAL-U *is a subset of that returned by* GS-UROW.

Proof. This statement follows immediately from the graph search constraint. An edge is only discovered due to the existence of a vertex j that is adjacent to an enqueued vertex t. The effect of the graph search constraint is that only a subset of the vertices enqueued by GS-UROW are enqueued by POTENTIAL-U. □

Theorem 18 *For any graph G and $\ell = 0, 1,$ or 2, the set of fill edges returned by* POTENTIAL-U *is identical to that returned by* GS-UROW.

Proof. Consider a fill path $i, \ldots, h, j$, where $i < j$. It suffices to show that vertex h will be enqueued. The case for $\ell = 0$ is immediate, since here we are searching for fill paths of the form i, j, and vertex i is always enqueued (Step 2, Figure 7).

For $\ell = 1$ POTENTIAL-U searches for fill paths containing two edges. These paths must be of the form i, h, j, with $h < i < j$. Vertex i is always enqueued. Since vertex h is adjacent to vertex i, and $i \geq i$, it will also be enqueued.

For $\ell = 2$, POTENTIAL-U searches for fill paths containing three edges, which are of the form i, h_1, h_2, j. From the argument for $\ell = 1$, vertex h_1 will always be enqueued. When vertex h_1 is dequeued vertex h_2 will be discovered. Since vertex h_2 is adjacent vertex j, this vertex will also be enqueued. □

Theorem 19 *For any graph G and $\ell \geq 3$, the set of fill edges returned by* POTENTIAL-U *may be only a subset of that returned by* GS-UROW.

Proof. The proof is by construction. It suffices to show an example for $\ell = 3$ where this is true. Consider the level 3 fill path, i, h_1, h_2, h_3, j, and suppose that h_2 is the smallest vertex in the path. From the proof of the previous theorem, vertices i and h_1 will necessarily be enqueued. When vertex h_1 is dequeued vertex h_2 will be discovered. Since this vertex is not adjacent to any vertex $j \geq i$, the graph search constraint prohibits its enqueuement. Hence vertex h_3 will not be discovered or enqueued, and the terminating vertex in the fill path, level j, will not be discovered. □

Theorem 20 *Given a 2D or 3D naturally ordered graph,* GS-UROW *and* POTENTIAL-U *compute identical sparsity sets.*

Proof. The proof is geometrical in nature. Referring to the bottom half of Figure 17, all vertices h in all shortest fill paths are necessarily adjacent to vertices j such that $j > i$. □

For 2D naturally ordered graphs, POTENTIAL-U has run time complexity of $O(n\ell)$. For 3D naturally ordered graphs, POTENTIAL-U has run time complexity of $O(n\ell^2)$. Again, the arguments are geometrical in nature. Table 2 compares the run time complexities of POTENTIAL-U with other symbolic factorization algorithms discussed in this chapter.

TABLE 2

POTENTIAL-U *run time complexity comparison. Results for* CLASSIC-ILU *and* GS-UROW *apply for any ordering. The results for* POTENTIAL-U *for 2D and 3D are only guaranteed for natural ordering.*

Algorithm	2D	3D
CLASSIC-ILU	$O(n\ell^2)$	$O(n\ell^4)$
GS-UROW	$O(n\ell^2)$	$O(n\ell^3)$
POTENTIAL-U	$O(n\ell)$	$O(n\ell^2)$

CHAPTER 4

PARALLEL ILU

This chapter describes a parallel algorithm for computing incomplete factor (ILU) preconditioners. The algorithm attains a high degree of parallelism through graph partitioning and a two-level ordering strategy. Both the subdomains and the nodes within each subdomain are ordered to preserve concurrency. We show through an algorithmic analysis and through computational results that this algorithm is scalable. Experimental results include timings on three parallel platforms for problems with up to 20 million unknowns running on up to 216 processors. The resulting preconditioned Krylov solvers have the desirable property that the number of iterations required for convergence is insensitive to the number of processors for a given problem size, if the number of interior to boundary nodes is relatively large.

Material in this chapter has been published in the SIAM Journal of Scientific Computing [53]. This chapter provides and introduction to and overview of the research with which the remainder of this dissertation is concerned. Follow on chapters provide additional algorithmic and scalability details and results, and describe the model PILU implementation that was developed as part of this dissertation.

An outline of this chapter is as follows. Section 1 provides an overview and introduction to the PILU algorithm. Section 2 describes the steps in the parallel algorithm for computing the ILU preconditioner in detail and provides theoretical justification. The algorithm is based on the incomplete fill path theorem that was presented in Chapter 2. We also discuss the role that a subdomain graph constraint plays in the design of the algorithm, show that the preconditioners exist for special classes of matrices, and relate our work to earlier work on this problem. Section 3 contains an analysis that shows that the parallel algorithm is scalable for two-dimensional (2D-) and three-dimensional (3D-)model problems, when they are suitably ordered and partitioned. Section 4 contains computational results on Poisson and convection-diffusion problems. The first subsection shows that the parallel ILU algorithm is scalable on three parallel platforms; the second subsection reports convergence studies. We tabulate how the number of Krylov solver iterations and the number of entries in the preconditioner vary as a function of the preconditioner level for three variations of the algorithm. The results show that fill levels higher

than one are effective in reducing the number of iterations; the number of iterations is insensitive to the number of subdomains; and the subdomain graph constraint does not affect the number of iterations while it makes possible the design of a simpler parallel algorithm.

The background needed for ILU preconditioning may be found in several books; see, e.g., [1, 38, 45, 77]. A preliminary version of the material in this chapter was presented at Supercomputing '99 and was published in the conference proceedings [49], and as technical reports [51, 52].

4.1 INTRODUCTION

Incomplete factorization (ILU) preconditioning is currently among the most robust techniques employed to improve the convergence of Krylov space solvers for linear systems of equations. However, scalable parallel algorithms for computing ILU preconditioners have not been available until recently despite the fact that ILU has been heavily used in applications for more than twenty years [27]. We report the design, analysis, implementation, and computational evaluation of a parallel algorithm for computing ILU preconditioners.

Our parallel algorithm assumes that three requirements are satisfied.

- The adjacency graph of the coefficient matrix (or the underlying finite element or finite difference mesh) must have good edge separators, i.e., it must be possible to remove a small set of edges to divide the problem into a collection of subproblems that have roughly equal computational work requirements.

- The size of the problem must be sufficiently large relative to the number of processors so that the work required by the subgraph on each processor is suitably large to dominate the work and communications needed for the boundary nodes.

- The subdomain intersection graph (to be defined later) should have a small chromatic number. This requirement will ensure that the dependencies in factoring the boundary rows do not result in undue losses in concurrency.

4.2 ALGORITHMS

In this section we discuss the Parallel ILU (PILU) algorithm and its underlying theoretical foundations.

4.2.1 THE PILU ALGORITHM

Figure 19 describes the steps of the PILU algorithm at a high level; the algorithm is suited for implementation on both message-passing and shared-address space programming models.

The PILU algorithm consists of four major steps. In the first step, we create parallelism by dividing the problem into subproblems by means of graph partitioning. In the second step, we preserve the parallelism in the interior of the subproblems by locally scheduling the computations in each subgraph. In the third step, we preserve parallelism in the boundaries of the subproblems by globally ordering the subproblems through coloring a suitably defined graph. In the final step, we compute the preconditioner in parallel.

Before discussing the four steps in greater detail, an example may prove illuminating. Figure 20, top, shows a Matlab spy plot of a matrix arising from a five-point discretization on a 2D grid. The spy plot in the bottom half of the figure shows the same matrix, after it has been partitioned amongst nine processors (first step), and reordered (second and third steps). Figure 21 shows spy plots of the filled matrix, $F = L + U - I$, following factorization (fourth step). The top spy plot shows fill for PILU(4), and the bottom spy plot show fill for PILU(10).

Step 1: Graph partitioning. In the first step of PILU, we partition the adjacency graph $G(A)$ of the coefficient matrix A into p subgraphs by removing a small set of edges that connects the subgraphs to each other. Each subgraph will be mapped to a distinct processor that will be responsible for the computations associated with the subgraph.

An example of a model five-point grid partitioned into four subgraphs is shown in Figure 22. For clarity, the edges corresponding to the coefficient matrix elements (within each subgraph or between subgraphs) are not shown. The edges drawn correspond to fill elements (elements that are zero in the coefficient matrix but are nonzero in the incomplete factors) that join the different subgraphs.

Input: A coefficient matrix, its adjacency graph, and the number of processors p.
Output: The incomplete factors of the coefficient matrix.

1. Partition the adjacency graph of the matrix into p subgraphs (subdomains), and map each subgraph to a processor. The objectives of the partitioning are that the subgraphs should have roughly equal work, and there should be few edges that join the different subgraphs.

2. On each subgraph, locally order interior nodes first, and then order boundary nodes.

3. Form the subdomain intersection graph corresponding to the partition, and compute an approximate minimum vertex coloring for it. Order subdomains according to color classes.

4. Compute the incomplete factors in parallel.
   a. Factor interior rows of each subdomain.
   b. Receive sparsity patterns and numerical values of the nonzeros of the boundary rows of lower-numbered subdomains adjacent to a subdomain (if any).
   c. Factor boundary rows in each subdomain and send the sparsity patterns and numerical values to higher-numbered neighboring subdomains (if any).

FIG. 19. *High level description of the PILU algorithm.*

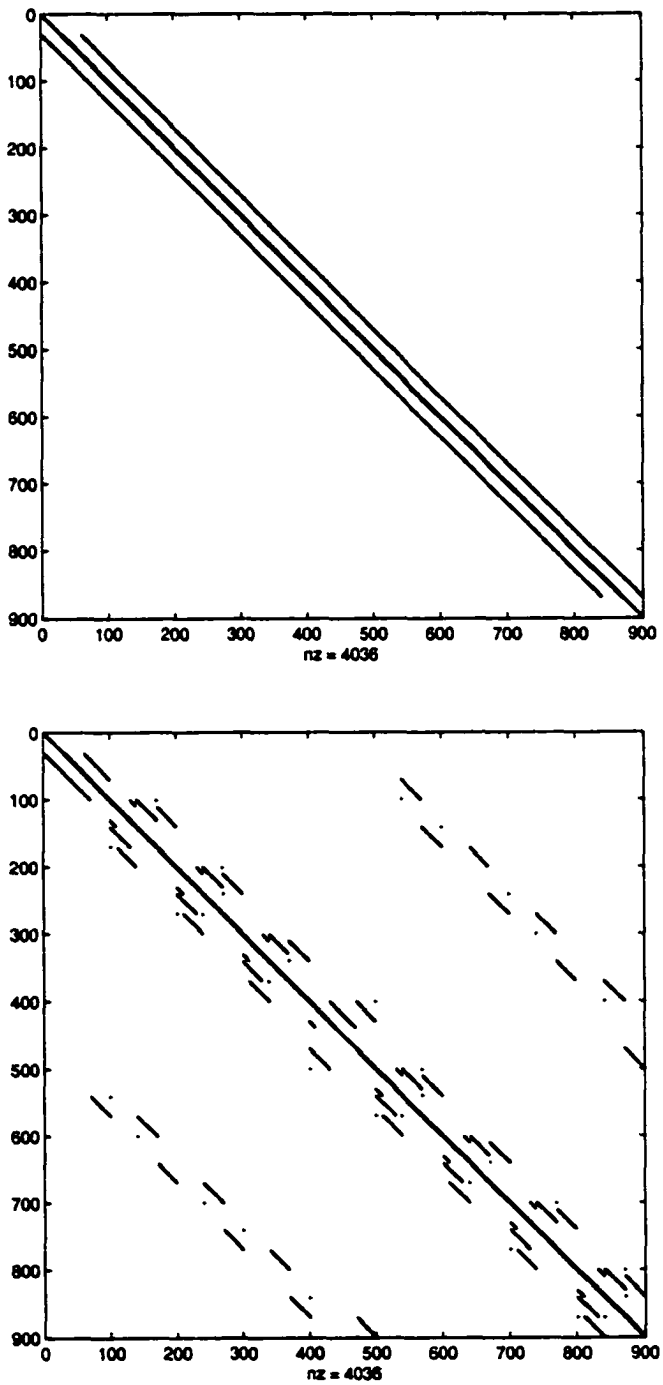


FIG. 20. *PILU ordering pattern, level zero. Top: spy plot of a naturally ordered matrix arising from a five-point stencil on a* 30×30 *grid; Bottom: matrix has been partitioned into nine subdomains and ordered per the PILU algorithm.*

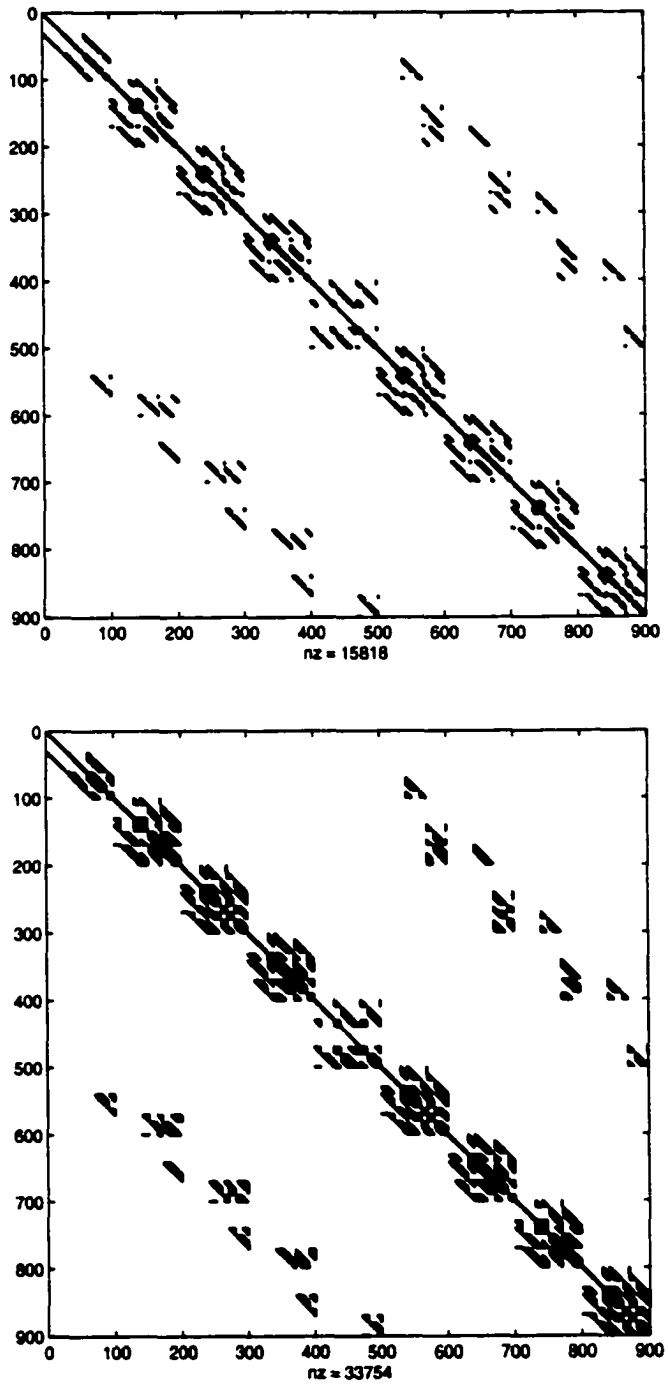


FIG. 21. *PILU ordering patterns, levels four and ten. Top: level four fill for matrix from Figure 20; Bottom: level ten fill.*

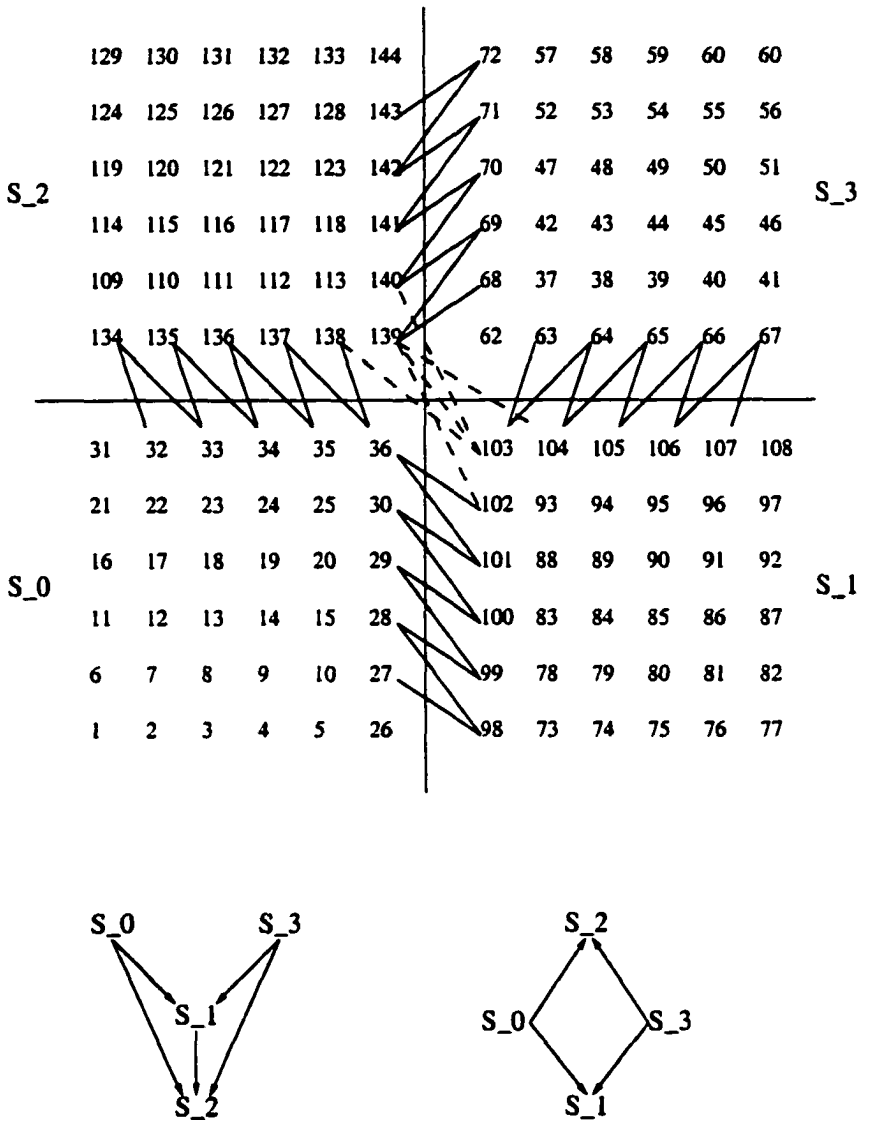


FIG. 22. *PILU partitioning, mapping, and vertex ordering. An example that shows the partitioning, mapping, and vertex ordering used in the PILU algorithm. The graph on the top is a regular* 12×12 *grid with a five-point stencil partitioned into four subdomains and then mapped on four processors. The subdomains are ordered by a coloring algorithm to reduce dependency path lengths. Only the level one and two fill edges that join the different subdomains are shown; all other edges are omitted for clarity. The figure on the bottom right shows the subdomain intersection graph when the subdomain graph constraint is enforced. (This prohibits fill between the boundary nodes of the subdomains* S_1 *and* S_2*, indicated by the broken edges in the top graph.) The graph on the bottom left shows the subdomain intersection graph when the subdomain graph constraint is not enforced.*

To state the objective function of the graph partitioning problem, we need to introduce some terminology. An edge is a *separator edge* if its endpoints belong to different subgraphs. A vertex in a subgraph is an *interior vertex* if all of its neighbors belong to that subgraph; it is a *boundary vertex* if it is adjacent to one or more vertices belonging to another subgraph. By definition, an interior vertex in a subgraph is not adjacent to a vertex (boundary or interior) in another subgraph. In Figure 22, the first 25 vertices are interior vertices of the subgraph S_0, and vertices numbered 26 through 36 are its boundary vertices. The goal of the partitioning is to keep the amount of work associated with the incomplete factorization of each subgraph roughly equal, while keeping the communication costs needed to factor the boundary rows as small as possible.

There is a difficulty with modeling the communication costs associated with the boundary rows. In order to describe this difficulty, we need to relate this cost more precisely to the separators in the graph. Define the *higher degree* of a vertex v as the number of vertices numbered higher than v in a given ordering. We assume that upward-looking, row-oriented factorization is used. At each boundary between two subgraphs, elements need to be communicated from the lower numbered subgraph to the higher numbered subgraph. The number of these elements is proportional to the sum of the higher degrees (in the filled graph $G(F)$) of the boundary vertices in the lower numbered subgraph. But unfortunately, we do not know the fill edges at this point since we have neither computed an ordering of $G(A)$ nor computed a symbolic factorization. We could approximate by considering higher degrees of the boundary vertices in the graph $G(A)$ instead of the filled graph $G(F)$, but even this requires us to order the subgraphs in the partition.

The union of the boundary vertices on all the subgraphs forms a *wide vertex separator*. This means that the shortest path from an interior vertex in any subgraph to an interior vertex in another subgraph consists of at least three edges; such a path has *length* at least three. The communication cost in the (forward and backward) triangular solution steps is proportional to the sum of the sizes of the wide vertex separators. None of the publicly available graph partitioning software has the minimization of wide separators as its objective function, but it is possible to modify existing software to optimize this objective.

The goal of the partitioning step is to keep the amount of work associated with

each subgraph roughly equal (for load balance) while making the communication costs due to the boundaries as small as possible. As the previous two paragraphs show, modeling the communication costs accurately in terms of edge and vertex separators in the initial graph $G(A)$ is difficult, but we could adopt the minimization of the wide separator sizes as a reasonable goal. This problem is NP-complete, but there exist efficient heuristic algorithms for partitioning the classes of graphs that occur in practical situations. (Among these graph classes are 2D-finite element meshes and 3D-meshes with good aspect ratios.)

Step 2: Local reordering. In the second step, in each subgraph we order the interior vertices before the boundary vertices. This ordering ensures that during the incomplete factorization, an interior vertex in one subgraph cannot be joined by a fill edge to a vertex in another subgraph, as will be shown later. Fill edges between two subgraphs can join only their boundary vertices together. Thus interior vertices corresponding to the initial graph $G(A)$ remain interior vertices in the graph of the factor $G(F)$. The consequences of this are that the rows corresponding to the interior vertices in each subdomain of the initial problem $G(A)$ can be factored concurrently, and that communication is required only for factoring rows corresponding to the boundary rows. The reader can verify that in each subgraph in Figure 22 the interior nodes have been ordered before the boundary nodes.

The observation concerning fill edges in the preceding paragraph results from an application of the Theorems 2 and 4 from Chapter 2. Now consider the adjacency graph $G(A)$ and a partition $\Pi = \{S_0, \ldots, S_{p-1}\}$ of it into subgraphs (subdomains). Any path joining two interior nodes in distinct subdomains must include at least two boundary nodes, one from each of the subgraphs; since each boundary node is numbered higher than (at least one of) the path's end vertices (since these are interior nodes in the subgraph), this path cannot be a fill path. If two interior nodes belonging to separate subgraphs were connected by a fill path and the corresponding fill entry were permitted in F, the interior nodes would be transformed into boundary nodes in $G(F)$. This is undesirable for parallelism, since then there would be fewer interior nodes to be eliminated concurrently.

The local ordering step preserves interior and boundary nodes during the factorization and ensures that a subdomain's interior rows can be factored independently

of row updates from any other subdomain. Therefore, when subdomains have relatively large interior/boundary node ratios, and contain approximately equal amounts of computational work, we expect PILU to exhibit a high degree of parallelism.

Step 3: Global ordering. The global ordering phase is intended to preserve parallelism while factoring the rows corresponding to the boundary vertices. In order to explain the loss of concurrency that could occur during this phase of the algorithm, we need the concept of a subdomain intersection graph, which we shall call a subdomain graph for brevity.

The subdomain graph $S(G, \Pi) = (V_s, E_s)$ is computed from a graph G and its partition $\Pi = \{S_0, \ldots, S_{p-1}\}$ into subgraphs. The vertex set V_s contains a vertex corresponding to every subgraph in the partition; the edge set E_s contains edge $\langle S_i, S_j \rangle$ if there is an edge in G with one endpoint in S_i and the other in S_j. We can compute a subdomain graph $S(A)$ corresponding to the initial graph $G(A)$ and its partition. (This graph should be denoted $S(G(A), \Pi)$, but we shall write $S(A)$ for simplicity.) We could also compute a subdomain graph $S(F)$ corresponding to the graph of the factor $G(F)$. The subdomain graph $S(A)$ corresponding to the partition of the initial graph $G(A)$ (the top graph) in Figure 22 is shown in the graph at the bottom right in that figure.

We impose a constraint on the fill, *the subdomain graph constraint.* The subdomain graph corresponding to $G(F)$ is restricted to be identical to the subdomain graph corresponding to $G(A)$. This prohibits some fill in the filled graph $G(F)$: if two subdomains are not joined by an edge in the original graph $G(A)$, any fill edge that joins those subdomains is not permitted in the graph of the incomplete factor $G(F)$. The description of the PILU algorithm in Figure 19 assumes that the subdomain graph constraint is satisfied. This constraint makes it possible to obtain scalability in the parallel ILU algorithm. Later, we discuss how the algorithm should be modified if this constraint is relaxed.

Each subdomain's nodes (in $G(A)$) are ordered contiguously. Consequently, saying "subdomain r is ordered before subdomain s" is equivalent to saying "all nodes in subdomain r are ordered, and then all nodes in subdomain s are ordered." This permits $S(A)$ to be considered as a directed graph, with edges oriented from lower to higher numbered vertices.

Edges in $S(F)$ indicate data dependencies in factoring the boundary rows of the subdomains. If an edge in $S(F)$ joins r and s and subdomain r is ordered before subdomain s, then updates from the boundary rows of r have to be applied to the boundary rows of s before the factorization of the latter rows can be completed. It follows that ordering $S(F)$ so as to reduce directed path lengths reduces serial bottlenecks in factoring the boundary rows. If we impose the subdomain graph constraint, these observations apply to the subdomain graph $S(A)$ as well since then $S(A)$ is identical with $S(F)$.

We reduce directed path lengths in $S(A)$ by coloring the vertices of the subdomain graph with few colors using a heuristic algorithm for graph coloring, and then by numbering the subdomains by color classes. The boundary rows of all subdomains corresponding to the first color can be factored concurrently without updates from any other subdomains. These subdomains update the boundary rows of higher numbered subdomains adjacent to them. After the updates, the subdomains that correspond to the second color can factor their boundary rows. This process continues by color classes until all subdomains have factored their boundary rows. The number of steps it takes to factor the boundary rows is equal to the number of colors it takes to color the subdomain graph.

In Figure 22, let p_i denote the processor that computes the subgraph S_i. Then p_0 computes the boundary rows of S_0 and sends them to processors p_1 and p_2. Similarly, p_3 computes the boundary rows of subgraph S_3 and sends them to p_1 and p_2. The latter processors first apply these updates and then compute their boundary rows.

How much parallelism can be gained through subdomain graph reordering? We can gain some intuition through analysis of simplified model problems, although we cannot answer this question a priori for general problems and all possible partitions. Consider a matrix arising from a second order PDE that has been discretized on a regularly structured 2D grid using a standard five-point stencil. Assume that the grid is naturally ordered and that it has been partitioned into square subgrids and mapped into a square grid of p processors. In the worst case, the associated subdomain graph, which itself has the appearance of a regular 2D grid, can have a dependency path of length $2(\sqrt{p}-1)$. Similarly, a regularly structured 3D grid discretized with a seven-point stencil that is naturally ordered and then mapped

on a cube containing p processors can have a dependency path length of $3(\sqrt[3]{p}-1)$. However, regular 2D grids with the five-point stencil and regular 3D grids with the seven-point stencil are bipartite graphs and can be colored with two colors. If all subdomains of the first color class are numbered first, and then all subdomains of the second color class are numbered, the longest dependency path in S will be reduced to one. This discussion shows that coloring the subdomain graph is an important step in obtaining a scalable parallel algorithm.

Step 4: Preconditioner computation. Now that the subdomains and the nodes in each subdomain have been ordered, the preconditioner can be computed. We employ an upward-looking, row oriented factorization algorithm. The interior of each subdomain can be computed concurrently by the processors, and the boundary nodes can be computed in increasing order of the color classes. Either a level-based ILU(ℓ) or a numerical threshold based ILUT algorithm may be employed on each subdomain. Different incomplete factorization algorithms could be employed in different subdomains when appropriate, as in multiphysics problems. Different fill levels could be employed for the interior nodes in a subdomain and for the boundary nodes to reduce communication and synchronization costs.

4.2.2 RELAXING THE SUBDOMAIN GRAPH CONSTRAINT

Now we consider how the subdomain graph constraint might be relaxed. Given a graph $G(A)$ and a partition of it into subgraphs, we color the subdomain graph $S(A)$ and order its subdomains as before. Then we compute the graph $G(F)$ of an incomplete factor and its subdomain graph $S(F)$. To do this, we need to discover the dependencies in $S(F)$, but initially we have only the dependencies in $S(A)$ available. This has to be done in several rounds, because fill edges could create additional dependencies between the boundary rows of subdomains, which in turn might lead to further dependences. The number of rounds needed is the length of a longest dependency path in the subdomain graph $G(F)$, and this could be $\Omega(p)$. This discussion applies when an ILU(ℓ) algorithm is employed, with symbolic factorization preceding numerical factorization. If ILUT were to be employed, then symbolic factorization and numerical factorization must be interleaved, as would be done in a sequential algorithm.

We can then color the vertices of $S(F)$ to compute a schedule for factoring

the boundary rows of the subdomains. For achieving concurrency in this step the subdomain graph $S(F)$ should have a small chromatic number (independent of the number of vertices in $G(A)$). Note that the description of the PILU algorithm in Figure 19 needs to be modified to reflect this discussion when the subdomain graph constraint is relaxed.

The graph $G(F)$ in Figure 22 indicates the fill edges that join S_1 to S_2 as broken lines. The corresponding subdomain intersection graph $S(F)$ is shown on the lower left. The edge between S_1 and S_2 necessitates three colors to color $S(F)$: the subdomains S_0 and S_3 form one color class; S_1 by itself constitutes the second color class; and S_2 by itself makes up the third color class. Thus three steps are needed for the computation of the boundary rows of the preconditioner when the subdomain graph constraint is relaxed. Note that the processor responsible for the subdomain S_2 can begin computing its boundary rows when it receives an update from either S_0 or S_3, but that it cannot complete its computation until it has received the update from the subdomain S_1.

Theorem 4 has an intuitively simple geometric interpretation. Given an initial node i in $G(A)$, construct a topological "sphere" containing all nodes that are at a distance less than or equal to $k+1$ edges. Then a fill entry f_{ij} is admissible in an ILU(ℓ) factor only if j is within the sphere. Note that all such nodes j do not cause fill edges since there needs to be a fill path joining i and j. By applying Theorem 4, we can gain an intuitive understanding of the fill entries that may be discarded on account of the subdomain graph constraint. Referring again to Figure 22, we see that prohibited edges arise when two nonadjacent subdomains in $G(A)$ have nodes that are joined by a fill path of length less than $k+1$. No level zero edge is discarded by the constraint.

4.2.3 EXISTENCE OF PILU PRECONDITIONERS

The existence of preconditioners computed from the PILU algorithm can be proven for some classes of problems.

Meijerink and van der Vorst [65] proved that if A is an M-matrix, then ILU factors exist for any predetermined sparsity pattern, and Manteuffel [64] extended this result to H-matrices with positive diagonal elements. These results immediately show that PILU preconditioners with sparsity patterns based on level values exist

for these classes of matrices. This is true even when different level values are used for the various subdomains and boundaries.

Incomplete Cholesky (IC) preconditioners for symmetric problems could be computed with our parallel algorithmic framework using preconditioners proposed by Jones and Plassmann [54] and by Lin and Moré [58] on each subdomain and on the boundaries. The sparsity patterns of these preconditioners are determined by the numerical values in the matrix and by memory constraints. Lin and Moré have proved that these preconditioners exist for M- and H-matrices. Parallel IC preconditioners also can be shown to exist for M- and H-matrices. If the subdomain graph constraint is not enforced, then the preconditioner computed in parallel corresponds to a preconditioner computed by the serial algorithm from a reordered matrix.

4.3 PERFORMANCE ANALYSIS

In this section we present simplified theoretical analyses of algorithmic behavior for matrices arising from PDEs discretized on 2D grids with five-point stencils and 3D grids with seven-point stencils. Since our arguments are structural in nature, we assume ILU(ℓ) is the factorization method used.

We assume the grid has been block-partitioned, with each subdomain consisting of a square subgrid of dimension $c \times c$. We also assume the subdomain grid has dimensions $\sqrt{p} \times \sqrt{p}$, so there are p processors in total. There are thus $N = c^2 p$ nodes in the grid, and subdomains have at most $4c = 4\sqrt{\frac{N}{p}}$ boundary nodes.

If subdomain interior nodes are locally numbered in natural order and $\ell \ll c$, each row in the factor F asymptotically has 2ℓ (strict) upper triangular and 2ℓ (strict) lower triangular nonzero entries. The justification for this statement arises from a consideration of the incomplete fill path theorem; the intuition is illustrated in Figure 23.

Assuming that the classical ILU(ℓ) algorithm is used for symbolic factorization, both symbolic and numeric factorization of row j entails $4\ell^2$ arithmetic operations. This is because for each lower triangular entry f_{ji} in matrix row j, factorization requires an arithmetic operation with each upper triangular entry in row i.

A red-black ordering of the subdomain graph gives an optimal bipartite division. If red subdomains are numbered before black subdomains, our algorithm simplifies to the following three stages.

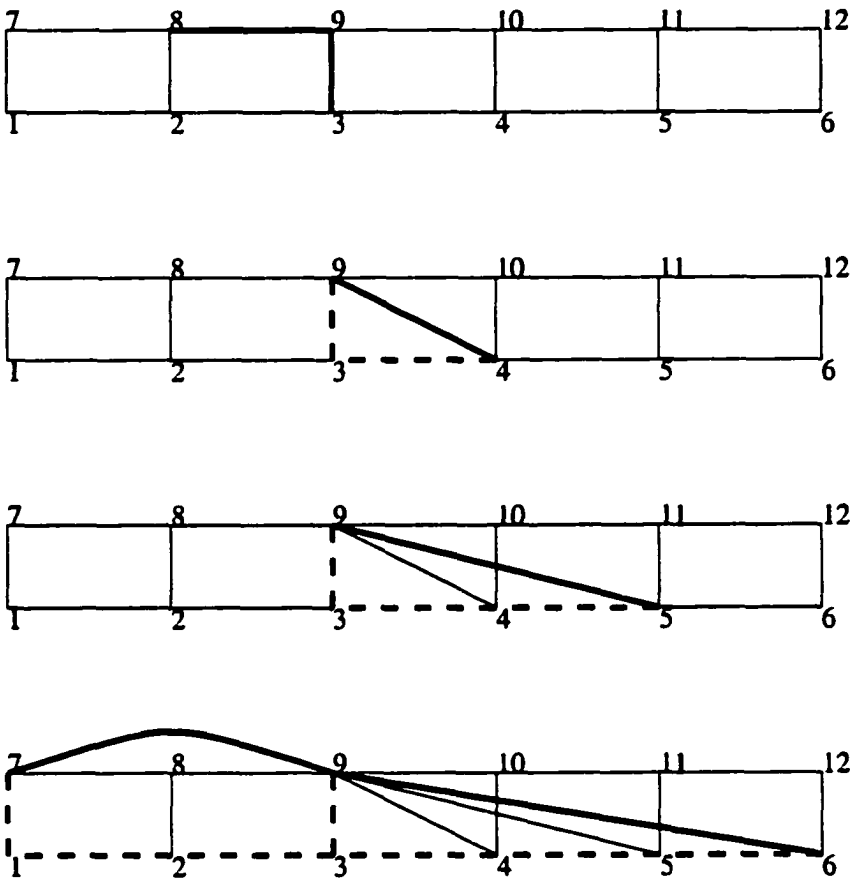


FIG. 23. *Counting lower triangular fill edges in a naturally ordered graph. We count the number of edges incident on vertex* 9. *Considering the graphs from top to bottom, we find that there are two level* 0 *edges; there is one level* 1 *edge, due to fill path* 9, 3, 4*; there is one level* 2 *edge due to fill path* 9, 3, 4, 5*; there are two level* 3 *edges, due to fill paths* 9, 3, 4, 5, 6 *and* 9, 3, 2, 1, 7. *We can generalize that two additional fill edges are created for every level greater than three, except near the boundaries. We conclude that asymptotically there are* $2k$ *lower triangular edges incident on a vertex in a level* k *factorization. Since the mesh corresponds to a structurally symmetric problem, there are* $2k$ *upper triangular edges incident on a vertex as well.*

1. Red processors eliminate all nodes; black processors eliminate interior nodes.
2. Red processors send boundary-row structure and values to black processors.
3. Black processors eliminate boundary nodes.

If these stages are nonoverlapping, the cost of the first stage is bounded by the cost of eliminating all nodes in a subdomain. This cost is $4\ell^2c^2 = \frac{4\ell^2N}{p}$.

The cost for the second stage is the cost of sending structural and numerical values from the upper-triangular portions of the boundary rows to neighboring processors. If $\ell \ll c$, the incomplete fill path theorem can be used to show that, asymptotically, a processor only needs to forward values from c rows to each neighbor. We assume a standard, noncontentious communication model wherein α and β represent message startup and per-word-transfer times, respectively. We measure these times in non-dimensional units of flops by dividing them by the time it takes to execute one flop. The time for an arithmetic operation is thus normalized to unity. Then the cost for the second step is $4(\alpha + 2\ell\beta c) = 4(\alpha + 2k\beta\sqrt{\frac{N}{p}})$.

Since the cost of factoring a boundary row can be shown to be asymptotically identical to that for factoring an interior row, the cost for eliminating the $4c$ boundary nodes is $(4\ell^2)(4c) = 16\ell^2\sqrt{\frac{N}{p}}$. Speedup can then be expressed as

$$\text{speedup} = \frac{4\ell^2N}{\frac{4\ell^2N}{p} + 4(\alpha + 2\ell\beta\sqrt{\frac{N}{p}}) + 16\ell^2\sqrt{\frac{N}{p}}}.$$

The numerator represents the cost for sequential execution, and the three terms in the denominator represent the costs for the three stages (arithmetic for interior nodes, communication costs, and arithmetic for the boundary nodes) of the parallel algorithm.

Three implications from this equation are in order. First, for a fixed problem size and number of processors, the parallel computational cost (the first and third terms in the denominator) is proportional to ℓ^2, while the communication cost (the second term in the denominator) is proportional to ℓ. This explains the increase in efficiency with level that we have observed. Second, if the ratio N/p is large enough, the first term in the denominator will become preeminent, and efficiency will approach 100%. Third, if we wish to increase the number of processors p by some factor while maintaining a constant efficiency, we need only increase the size of

the problem N by the same factor. This shows that our algorithm is scalable. This observation is not true for a direct factorization of the coefficient matrix, where the dependencies created by the additional fill cause loss in concurrency.

For the 3D case we assume partitioning into cubic subgrids of dimension $c \times c \times c$ and a subdomain grid of dimension $p^{1/3} \times p^{1/3} \times p^{1/3}$, which gives $N = c^3 p$. Subdomains have at most $6c^2$ boundary nodes. A development similar to that above shows that, asymptotically, matrix rows in the factor F have $2\ell^2$ (strict) upper and lower triangular entries, so the cost for factoring a row is $4\ell^4$. Speedup for this case can then be expressed as

$$\begin{aligned} \text{speedup} &= \frac{4\ell^4 N}{\frac{4\ell^4 N}{p} + 6(\alpha + 2\ell^2\beta(\frac{N}{p})^{1/3}) + 24\ell^4(\frac{N}{p})^{1/3}} \\ &= \frac{2\ell^4 N}{\frac{2\ell^4 N}{p} + 3(\alpha + 2\ell^2\beta(\frac{N}{p})^{1/3}) + 12\ell^4(\frac{N}{p})^{1/3}}. \end{aligned}$$

4.4 RESULTS

Results in this section are based on the following model problems.

Problem 1. Poisson's equation in two or three dimensions:

$$\Delta u = g.$$

Problem 2. Convection-diffusion equation with convection in the xy plane:

$$-\varepsilon \Delta u + \frac{\partial}{\partial x} e^{xy} u + \frac{\partial}{\partial y} e^{-xy} u = g.$$

Homogeneous boundary conditions were used for both problems. Derivative terms were discretized on the unit square or cube, using 3-point central differencing on regularly spaced $n_x \times n_y \times n_z$ grids ($n_z = 1$ for 2D). The values for ε in Problem 2 were set to 1/500 and 1/1000. The problem becomes increasingly unsymmetric, and more difficult to solve accurately as ε decreases. The right-hand sides of the resulting systems, $Ax = b$, were artificially generated as $b = A\hat{e}$, where $\hat{e}$ is the all-ones vector.

In addition to demonstrating that our algorithm can provide high degrees of parallelism, we address several other issues. We study the influence of the subdomain graph constraint on the fill permitted in the preconditioner and on the

convergence of preconditioned Krylov space solvers. We also report convergence results as a function of the number of nonzeros in the preconditioner.

4.4.1 PARALLEL PERFORMANCE

We now report timing and scalability results for preconditioner factorization and application on three parallel platforms:

- an SGI Origin2000 at NASA Ames Research Center (AMES);
- the Coral PC Beowulf cluster at ICASE, NASA Langley Research Center;
- a Sun HPC 10000 Starfire server at Old Dominion University (ODU).

Machine configuration information for these platforms appears in Appendix B. Both problems were solved using Krylov subspace methods as implemented in the PETSc [2] software library. Problem 1 was solved using the conjugate gradient method, and Problem 2 was solved using Bi-CGSTAB [82]. PETSc's default convergence criterion was used, which is five orders of magnitude (10^5) reduction in the residual of the preconditioned system. We used our own codes for problem generation, partitioning, ordering, and symbolic factorization.

Table 3 shows incomplete factorization timings for a 3D memory-scaled problem with approximately $91,125$ unknowns per processor. As the number of processors increases, so does the size of the problem. The coefficient matrix of the problem factored on 216 processors has about 19.7 million rows. ILU(2) was employed for the interior nodes, and ILU(1) was employed for the boundary nodes. Reading down any of the columns shows that performance is highly scalable, e.g., for the SGI Origin2000, factorization for 216 processors and 19.7 million unknowns required only 62% longer than the serial case. Scanning horizontally indicates that performance was similar across all platforms, e.g., execution time differed by less than a factor of two between the fastest (Origin2000) and slowest (Beowulf) platforms.

Table 4 shows similar data and trends for the triangular solves for the scaled problem. Scalability for the solves was not quite as good as for factorization; e.g., the solve with 216 processors took about 2.5 times longer than the serial case. This is expected due to the lower computation cost relative to communication and synchronization costs in triangular solution.

TABLE 3

Factorization Timing, 3D problem (SGI, Beowulf, Starfire). Scaled problem, 91,125 unknowns per processor, seven-point stencil, ILU(2) factorization on interior nodes, and ILU(1) factorization on boundary nodes. Dashes (-) for Beowulf and HPC 10000 indicate that the machines have insufficient cpus to perform the runs.

Procs	Origin2000 AMES	Beowulf (ICASE)	HPC 10000 (ODU)
1	2.04	2.27	2.13
8	2.44	3.11	2.43
27	2.96	4.06	2.97
64	3.11	4.64	-
125	3.18	-	-
216	3.32	-	-

We observed that the timings for identical repeated runs on the HPC 10000 and SGI typically varied by 50% or more, while repeated runs on the Beowulf were remarkably consistent.

Table 5 shows speedup for a constant-sized problem of 1.7 million unknowns. There is a clear correlation between performance and subdomain interior/boundary node ratios; this ratio needs to be reasonably large for good performance.

The performances reported in these tables are applicable to any PDE that has been discretized with a seven-point central difference stencil since the sparsity pattern of the symbolic factor depends on the grid and the stencil only.

4.4.2 CONVERGENCE STUDIES

Our approach for designing parallel ILU algorithms reorders the coefficient matrices whose incomplete factorization is being computed. This reordering could have a significant influence on the effectiveness of the ILU preconditioners. Accordingly, in this section we report the number of iterations of a preconditioned Krylov space solver needed to reduce the residual by a factor of 10^5.

We compare three different algorithms.

TABLE 4

Triangular solve timing, 3D problem (SGI, Beowulf, Starfire). Scaled problem, 91,125 unknowns per processor, seven-point stencil, ILU(2) factorization on interior nodes, ILU(1) factorization on boundary nodes. Dashes (-) for Beowulf and HPC 10000 indicate that the machines have insufficient cpus to perform the runs.

Procs	Origin2000 (AMES)	Beowulf (ICASE)	HPC 10000 (ODU)
1	.182	.187	.289
8	.431	.359	.515
27	.405	.508	.629
64	.472	.556	-
125	.610	-	-
216	.646	-	-

TABLE 5

Speedup for 3D constant-size problem (SGI). total of approximately 1.7 million unknowns; data is for ILU(0) factorization performed on the SGI Origin2000; "I/B ratio" is the ratio of interior to boundary nodes in each subdomain.

Procs	Unknowns/ Processor	I/B ratio	Time (sec.)	Efficiency (%)
8	216,000	9.3	2.000	100
27	64,000	6.0	0.846	70
64	27,000	4.3	.408	61
125	13,824	3.4	.307	42

- Constrained PILU(ℓ) is the parallel ILU(ℓ) algorithm with the subdomain graph constraint enforced.
- In unconstrained PILU(ℓ), the subdomain graph constraint is dropped, and all fill edges up to level ℓ between the boundary nodes of different subdomains are permitted, even when such edges join two nonadjacent subdomains of the initial subdomain graph $S(A)$.
- In Block Jacobi ILU(ℓ) (BJILU(ℓ)), all fill edges joining two different subdomains are excluded.

Intuitively, one expects, especially for diagonally dominant matrices, that larger amounts of fill in preconditioners will reduce the number of iterations required for convergence.

FILL COUNT COMPARISONS

For a given problem, the number of permitted fill edges is a function of three components: the factorization level, ℓ; the subdomain size(s); and the discretization stencil. While the numerical values of the coefficients of a particular PDE influence convergence, they do not affect fill counts. Therefore, our first set of results consists of fill count comparisons for problems discretized on a $64 \times 64 \times 64$ grid using a standard, seven-point stencil.

Table 6 shows fill count comparisons between unconstrained PILU(ℓ), constrained PILU(ℓ), and Block Jacobi ILU(ℓ) for various partitionings and factorization levels. The data shows that more fill is discarded as the factorization level increases, and as subdomain size (the number of nodes in each subdomain) decreases. These two effects hold for both constrained PILU(ℓ) and Block Jacobi ILU(ℓ) but are much more pronounced for the latter. For example, less than 5% of fill is discarded from unconstrained PILU(ℓ) factors when subdomains contain at least 512 nodes (so that the subgraphs on each processor are not too small), but up to 42% is discarded from Block Jacobi factors. Thus, one might tentatively speculate that, for a given subdomain size and level, PILU(ℓ) will provide more effective preconditioning than BJILU(ℓ). We have observed similar behavior for 2D problems also. For both 2D and 3D problems, when there is a single subdomain the factors returned by the three algorithms are identical. For the single subdomain

case, the ordering we have used corresponds to the natural ordering for these model problems.

An important observation to make in Table 6 is how the sizes (number of nonzeros) of the preconditioners depend on levels of fill. For the 3D problems considered here (cube with 64 points on each side, seven-point stencil), a level one preconditioner typically requires twice as much storage as the coefficient matrix A; when the level is two, this ratio is about three; when the level is three, it is about six; and when the level is four, it is about ten. For 2D problems (square grid with 256 points on a side, five-point stencil), the growth of fill with level is slower; the ratios are about 1.4 for level one, 1.8 for level two, 2.6 for level three, 3.5 for level four, 4.3 for level five, and 5.4 for level six.

In parallel computation fill levels higher than those employed in sequential computing are feasible since modern multiprocessors are either clusters or have virtual shared memory, and these have memory sizes that increase with the number of processors. Another point to note is that the added memory requirement for these level values is not as prohibitive as it is for a complete factorization. Hence it is practical to trade-off increased storage in preconditioners for reducing the number of iterations in the solver.

CONVERGENCE OF PRECONDITIONED ITERATIVE SOLVERS

The fill results in the previous subsection are not influenced by the actual numerical values of the nonzero coefficients; however, the convergence of preconditioned Krylov space solvers is influenced by the numerical values. Accordingly, Table 7 shows iterations required for convergence for various partitionings and fill levels for the three variant algorithms that we consider. The data in these tables can be interpreted in various ways; we begin by discussing two ways that we think are primarily significant.

First, by scanning vertically one can see how changing the number of subdomains, and hence, matrix ordering, affects convergence. The basis for comparison is the iteration count when there is a single subdomain. The partitioning and ordering for these cases is identical to, and our data in close agreement with, that reported by Benzi, Joubert, and Mateescu [6] for natural ordering. (They report

TABLE 6

Fill comparisons for the 64×64×64 grid. U denotes unconstrained, C denotes constrained, and B denotes Block Jacobi ILU(k) preconditioners. The columns headed "nzF/nzA" show the ratio of the number of nonzeros in the preconditioner to the number of nonzeros in the original problem and are indicative of storage requirements. The columns headed "constraint effects" present another view of the same data: here, the percentage of nonzeros in the constrained PILU(k) and Block Jacobi ILU(k) factors are shown relative to that for the unconstrained PILU(k). These columns show the amount of fill dropped due to the subdomain graph constraint.

Nodes per subdom.	Subdom. count	Level	nzF/nzA			Constraint effects (%)	
			U	C	B	C	B
262,144	1	0	1.00	1.00	1.00	100.00	100.00
		1	1.84	1.84	1.84	100.00	100.00
		2	3.22	3.22	3.22	100.00	100.00
		3	5.96	5.96	5.96	100.00	100.00
		4	9.73	9.73	9.73	100.00	100.00
32,768	8	0	1.00	1.00	0.99	100.00	98.64
		1	1.87	1.87	1.80	99.99	96.53
		2	3.36	3.35	3.12	99.96	92.91
		3	6.32	6.32	5.70	99.92	90.13
		4	10.50	10.49	9.19	99.89	87.56
4,096	64	0	1.00	1.00	0.96	100.00	95.93
		1	1.89	1.89	1.72	99.90	91.24
		2	3.45	3.44	2.91	99.62	84.36
		3	6.51	6.47	5.19	99.34	79.72
		4	10.81	10.70	8.17	99.06	75.61
512	512	0	1.00	1.00	0.90	100.00	90.50
		1	1.92	1.91	1.57	99.46	81.62
		2	3.59	3.52	2.53	98.05	70.35
		3	6.72	6.50	4.27	96.62	63.47
		4	10.96	10.43	6.32	95.20	57.69
64	4,096	0	1.00	1.00	0.80	100.00	79.64
		1	1.97	1.92	1.29	97.58	65.15
		2	3.73	3.42	1.86	91.67	49.79
		3	6.60	5.64	2.71	85.37	41.04
		4	10.01	7.76	3.35	77.56	33.45
8	32,768	0	1.00	1.00	0.58	100.00	57.92
		1	2.05	1.85	0.80	90.07	38.81
		2	3.98	2.55	0.87	64.14	21.84
		3	6.15	2.89	0.90	46.95	14.72
		4	7.40	2.90	0.90	39.26	12.23

TABLE 7

Iteration comparisons for the 64 x 64 x 64 grid. U denotes unconstrained, C denotes constrained, and B denotes Block Jacobi ILU(k) preconditioners. The starred entries (*) indicate that, since there is a single subdomain, the factor is structurally and numerically identical to the unconstrained PILU(k). Dashed entries (-) indicate the solutions either diverged or failed to converge after 200 iterations. For Problem 2, when $\varepsilon = 1/500$ the level zero preconditioners did not reduce the relative error in the solution by a factor of 10^5 at termination; when $\varepsilon = 1/1000$, the level one preconditioners did not do so either.

Nodes per subdom.	Subdom. count	Level	Problem 1			Problem 2					
						$\epsilon = 1/500$			$\epsilon = 1/1000$		
			U	C	B	U	C	B	U	C	B
262,144	1	0	43	*	*	19	*	*	-	*	*
		1	29	*	*	16	*	*	30	*	*
		2	24	*	*	8	*	*	32	*	*
		3	19	*	*	8	*	*	14	*	*
		4	16	*	*	6	*	*	8	*	*
32,768	8	0	45	45	53	32	32	26	-	-	-
		1	32	33	41	14	14	19	38	39	41
		2	27	29	37	11	11	17	38	38	66
		3	22	24	33	8	8	13	16	15	21
		4	19	21	29	7	7	13	10	11	18
4,096	64	0	43	43	55	33	33	49	-	-	-
		1	31	32	45	15	15	21	42	41	46
		2	25	27	41	12	11	22	24	28	78
		3	20	23	39	9	9	16	18	17	28
		4	17	20	36	8	8	19	11	12	27
512	512	0	41	41	56	28	28	67	-	-	-
		1	29	31	48	18	16	29	39	40	111
		2	25	26	46	11	12	36	21	21	106
		3	21	23	44	11	11	31	20	21	110
		4	18	21	43	9	12	34	13	14	70
64	4,096	0	43	43	64	28	28	-	63	63	-
		1	30	33	60	17	18	124	55	56	-
		2	26	30	58	13	15	115	25	28	-
		3	21	28	58	12	17	127	24	36	-
		4	17	28	58	10	17	132	11	27	-
8	32,768	0	46	46	83	43	43	-	83	83	-
		1	32	41	82	24	46	-	152	-	-
		2	25	40	82	11	45	-	13	115	-
		3	19	40	82	5	44	-	7	107	-
		4	16	40	82	4	45	-	6	111	-

results for Problem 2 with $\varepsilon = 1/500$ but not for $\varepsilon = 1/1000$.)

A pleasing property of both the constrained and unconstrained PILU algorithms is that the number of iterations increases only mildly when we increase the number of subdomains from one to 512 for these problems. This insensitivity to the number of subdomains when the number of nodes per subdomain is not too small confirms that the PILU algorithms enjoy the property of parallel algorithmic scalability. For example, Poisson's equation (Problem 1) preconditioned with a level two factorization and a single subdomain required 24 iterations. Preconditioning with the same level, constrained PILU(ℓ) on 512 subdomains needed only two more iterations. Similar results are observed for the convection-diffusion problems also. This property is a consequence of the fill between the subdomains that is included in the PILU algorithm.

Similar results have been reported in [62, 63] and the first paper includes a condition number analysis supporting this observation.

Increasing the level of fill generally has the beneficial effect of reducing the number of iterations needed; this influence is largest for the worse-conditioned convection-diffusion problem with $\varepsilon = 1/1000$. For this problem, level zero preconditioners do not converge for reasonable subdomain sizes. Also, even though level one preconditioners require fewer iteration numbers than level two preconditioners in some cases, when the PETSc solvers terminate because the residual norms are reduced by 10^5, the relative errors are larger than 10^{-5} for the former preconditioners. The relative errors are also large for the convection-diffusion problem with $\varepsilon = 1/500$ when the level is set to zero.

Second, scanning the data in Table 7 horizontally permits evaluation of the subdomain graph constraint's effects. Again, unless subdomains are small and the factorization level is high; constrained and unconstrained PILU(ℓ) show very similar behavior. Consider, for example, Poisson's equation (Problem 1) preconditioned with a level two factorization and 512 subdomains. The solution with unconstrained PILU(ℓ) required 25 iterations while constrained PILU(ℓ) required 26.

We also see that PILU(ℓ) preconditioning is more effective than BJILU(ℓ) for all 3D trials. (Recall that the single apparent exception, Problem 2, $\varepsilon = 1/500$, ILU(0) with 32,768 nodes per subdomain, has large relative errors at termination.) Again, the extremes of convergence behavior are seen for Problem 2 with $\varepsilon = 1/1000$. Here,

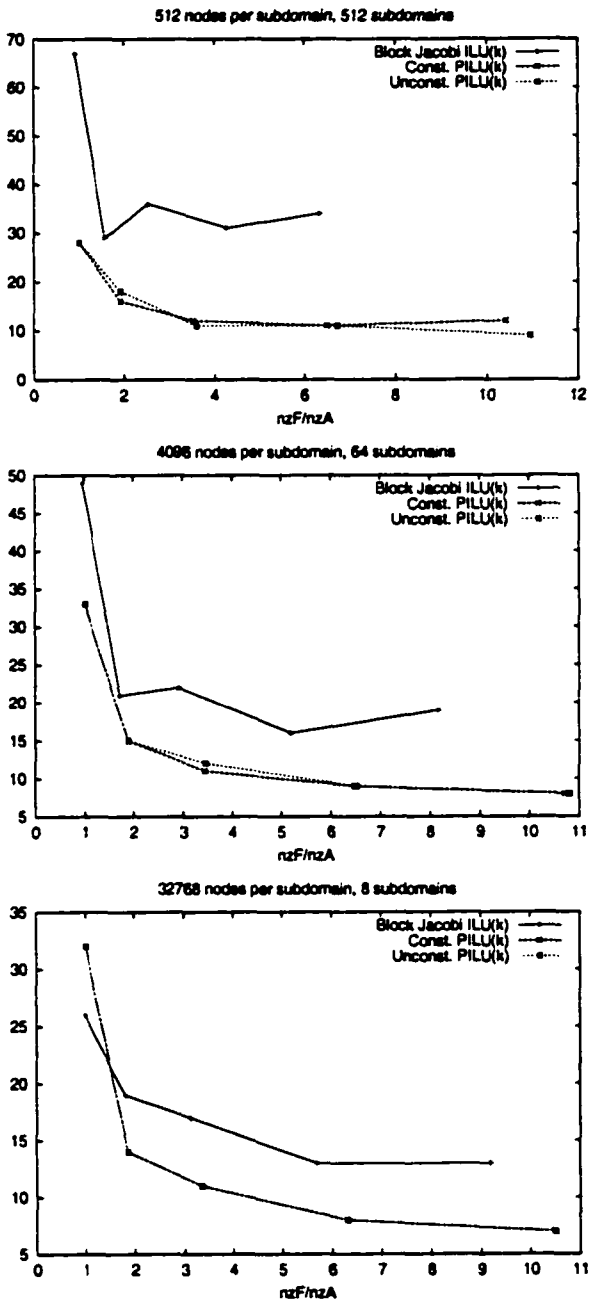


FIG. 24. *Convergence comparison for convection-diffusion problem.* $\epsilon = 1/500$ *on the* $64 \times 64 \times 64$ *grid. Data points are for levels* 0 *through* 4. *Data points for constrained and unconstrained PILU(k) are indistinguishable in the third graph.*

with level one preconditioners, BJILU(ℓ) suffers large relative errors at termination while the other two algorithms do not, when the number of subdomains is 64 or fewer.

On 2D domains, while PILU(ℓ) is more effective than BJILU(ℓ) for Poisson's equation, BJILU(ℓ) is sometimes more effective in the convection-diffusion problems.

We also examine iteration counts as a function of preconditioner size graphically. A plot of this data appears in Figure 24. In these figures the performance of the constrained and unconstrained PILU algorithms is often indistinguishable. We find again that PILU(ℓ) preconditioning is more effective than BJILU(ℓ) for 3D problems for a given preconditioner size; however, this conclusion does not always hold for 2D problems, especially for lower fill levels. As the number of vertices in the subdomains increases, higher fill levels become more effective in reducing the number of iterations needed for convergence. We find that fill levels as high as four to six can be the most effective when the subdomains are sufficiently large. Fill levels higher than these do not seem to be merited by these problems, even for the difficult convection-diffusion problems with $\varepsilon = 1/1000$, where a level four preconditioner reduces the number of iterations below ten.

CHAPTER 5

PARALLEL DESIGN AND SCALABILITY

In this chapter we examine several aspects related to PILU preconditioner scalability. First, we explain what we mean by our claim that PILU is scalable. Second, we discuss some details of the model PILU implementation (the Euclid library) that was developed as part of this dissertation. This discussion focuses on communication patterns and how they affect PILU's scalability; algorithmic design and implementation issues; and how PILU might be altered to yield better performance, e.g., by taking advantage of known symmetry when it exists. Third, we present experimental scalability results and develop an analytic result that states a scalability relationship between PILU and Block Jacobi ILU. The main thrust of this chapter is an examination of preconditioner setup costs, and per-iteration costs of applying the preconditioner.

Like many preconditioning techniques PILU consists of two distinct phases: preconditioner setup and preconditioner application. For Block Jacobi ILU preconditioning, setup consists of symbolic and numeric factorization and no communication is required. For PILU, setup additionally entails formation, coloring, and ordering of the subdomain graph; local reordering to place boundary nodes last; exchange of boundary node permutations; and setting up persistent communications that are used in triangular solves. Some of these steps require global communication (e.g., identification of boundary nodes for structurally unsymmetric problems), while others require only nearest neighbor communication (e.g., exchange of permutations and factorization). Communication costs can sometimes be lessened or dispensed with by taking advantage of special information, such as structural symmetry (with reference to the matrix), and regular structure (with reference to the processor grid).

In contrast to preconditioner setup, the application phase only requires nearest neighbor communication and is algorithmically much simpler. Unlike the setup phase, symmetry and processor grid structure do not help to lessen communication costs.

The scalability and communication costs examined in this chapter do not depend on the numerical properties of any particular problem. Given for example

a finite element grid, setup cost and application cost per iteration are identical whether one is solving a numerically well-behaved elliptic problem or a troublesome convection-diffusion problem. The costs involved derive entirely from combinatorial considerations.

In this chapter's first section we discuss the meaning of scalability, and how it applies to the PILU technique. In the second section we discuss details of the preconditioner setup phase. In the third section we discuss details of the preconditioner application (triangular solve) phase. The explanations and algorithms presented in the second and third sections expand on the high level description of PILU that was presented in Chapter 4. In the fourth section we summarize the influence of communication patterns on performance. In the fifth section we present experimental results for preconditioner setup and application on a per-iteration basis. In the final section we develop an analytic formula that permits comparison of PILU and Block Jacobi ILU performance.

In the interest of concreteness we explicitly reference MPI calls in the discussions in this chapter. However, PILU can of course be implemented using other message passing libraries, or in shared memory. Additionally, some of the algorithms presented in this chapter reflect specific solutions we developed in the course of coding our model PILU implementation; other, perhaps more efficient solutions are possible.

5.1 SCALABILITY AND PILU

Scalability is a term that is thrown about so freely by researchers and has so many connotations that there is a single implication to which all parties are likely to agree: to be scalable is good; to be unscalable is bad. A discussion of the various meanings of scalability, with references to many authors can be found in Chapter 4 of [56]. In this Chapter we are primarily interested in scalability with respect to *scaled problem size*. This is related to the concept of *scaled speedup*, whose origin is attributed to Gustafson [41, 42].

We claim that PILU has good scalability properties in that, if we hold problem size per processor constant while scaling the global problem size and processor

"When *I* use a word," Humpty Dumpty said in rather a scornful tone, "it means just what I choose it to mean—neither more nor less" [14].

count, (1) execution time for the preconditioner setup phase remains constant or grows very slowly, and (2) execution time for the preconditioner application phase remains constant or grows very slowly. By this definition, Block Jacobi ILU preconditioning (which is identical to additive Schwarz ILU with zero overlap) also has good scalability properties. The distinguishing feature of PILU is that the required number of iterations, for many problems we have examined to date, is lower for PILU than for Block Jacobi ILU.

5.2 PRECONDITIONER SETUP

PILU preconditioner setup involves three primary tasks, the first of which is divided into several stages.

1. Subdomain graph setup.
   (a) Exchange of row-ownership information.
   (b) Identification of boundary nodes.
   (c) Formation, coloring, and ordering of the subdomain graph.
   (d) Local reordering to place boundary nodes last in each subdomain.
   (e) Identification of nearest neighbors in the subdomain graph.
   (f) Exchange of boundary node permutations with nearest neighbors.
2. Symbolic and numeric factorization.
3. Setup persistent communications for the triangular solves.

SUBDOMAIN GRAPH SETUP

When Euclid is initialized through its interface with a linear solver library, it takes as input a matrix that is assumed to have been been partitioned and distributed such that each processor owns a consecutively numbered chunk of matrix rows (referred to as a subdomain), with each subdomain having a "large" interior/boundary node ratio. These assumptions are in accordance with common practice in computational physics and with the solver libraries to which Euclid interfaces (as of this writing, PETSc and Hypre).

Initially, each processor informs all others of the global numbering of its first locally owned row, and the number of locally owned rows. This is accomplished through calls to `MPI_Alltoall`, which result in each processor having an identical copy of the arrays *beg_row*[] and *row_count*[], each of which has as many bins as there are processors. The global number of the first row owned by processor P_u is stored in *beg_row*[P_u], and the number of P_u's locally owned rows is stored in *row_count*[P_u]. Given an external (non-locally owned) row i, a processor uses these arrays to identify the processor that owns the row i.

For structurally symmetric matrices, processors can completely identify both their local boundary rows and their neighbors in the subdomain graph by scanning the adjacency list representation of their locally owned rows. Processors look for entries of of the form $j \in adj(i)$, where i is a locally owned row and j is an external row. P_u can discern, by consulting the *beg_row*[] and *row_count*[] arrays, that (1) row i is a boundary row; (2) row j is owned by P_v and hence P_v is P_u's neighbor in the subdomain graph; (3) the factored matrix row i will need to be sent to P_v during factorization, if P_u is numbered less than P_v following subdomain graph coloring and ordering. By symmetry, P_v's adjacency lists must contain an entry of the form $i \in adj(j)$, which allows P_v to discern similar information concerning its boundary rows and subdomain graph neighbors.

For unsymmetric matrices identification of boundary rows requires a non-trivial communication step. Figure 25 illustrates why a processor P_u may not be able to identify all local boundary rows using purely local information. We have implemented a Boundary Node Identification algorithm that permits all processors to completely identify their local boundary rows in structurally unsymmetric cases.

In the Boundary Node Identification algorithm each processor initially identifies as many internal and external boundary nodes as possible by scanning once through its adjacency lists. Note that if P_u owns rows i and j but not k, and its adjacency list contains the entries $k \in adj(i)$ and $k \in adj(j)$, then the external boundary row k will be discovered twice. We implemented a SortedSet class to deal with this non-uniqueness problem. When an element (row index) is added by invoking this class's insert method, the element is inserted into a sorted list, or discarded if an identical element was previously inserted. Ensuring that the set of external row indices is sorted lessens the complexity of determining which processor owns which

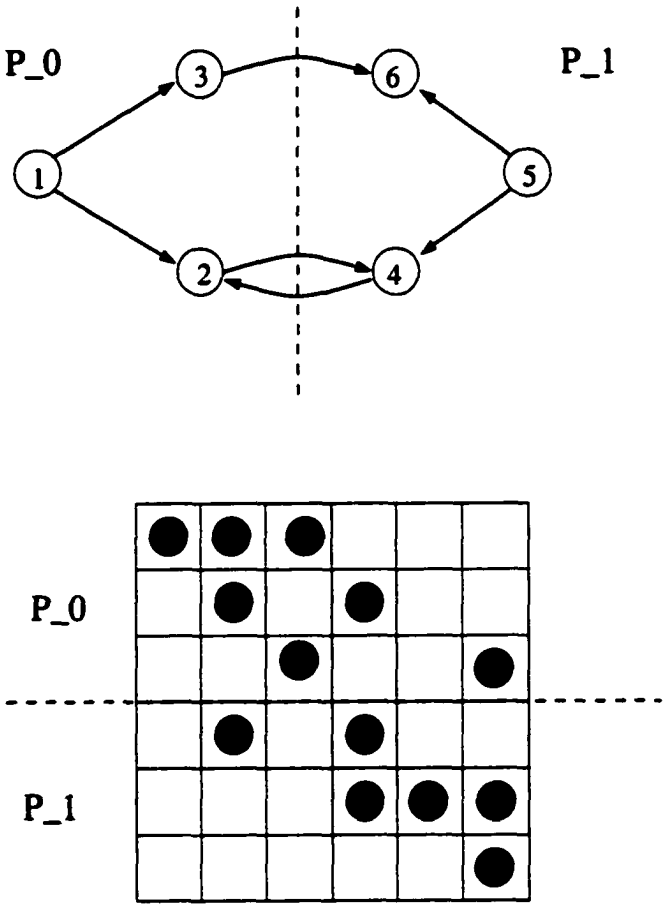


FIG. 25. *Identifying boundary nodes in unsymmetric graphs. The graph (top) of an unsymmetric matrix (bottom) has been partitioned amongst two processors. Nodes 2, 3, 4, and 6 are boundary nodes. Processor P_1 can discern, by scanning its local adjacency lists, that local node 4 and external node 2 are boundary nodes, but cannot discover that local node 6 is also a boundary node. Processor P_0 can discern, by scanning its local adjacency lists, that nodes 2 and 3 are local boundary nodes, and nodes 4 and 6 are external boundary nodes.*

rows, a point upon which we will not comment further.

Each processor must now tell all other processors of any external boundary nodes of which it is aware. This involves a call to `MPI_Alltoall`, wherein each processor sends every other processor a scalar value. A processor P_u sends a value 'x' to every other processor P_v, indicating the number of boundary nodes that P_u has identified as belonging to P_v. Processors can then allocate buffer space for receiving the lists of nodes (if any) from other processors, after which lists are exchanged via asynchronous `MPI_Isend` and `MPI_Irecv` calls. Finally, each processor merges its list of locally-identified boundary nodes with the lists of boundary nodes received from other processors.

In addition to identifying all local boundary rows, the Boundary Node Identification algorithm permits a processor to completely identify all its neighbors in the subdomain graph. P_v is P_u's neighbor if (1) P_u sent a nonzero datum to P_u during the all-to-all exchange, and/or (2) P_u received a nonzero value from P_v during the exchange.

The subdomain graph, which is the amalgamation of each processor's list of neighbors, is always symmetric. Suppose P_u owns matrix row i and P_v owns matrix row j. Then there will be an undirected edge $\{P_u, P_v\}$ in the subdomain graph if either or both of the matrix entries a_{ij} and a_{ji} exist.

Once each processor has identified its neighbors in the subdomain graph the global subdomain graph must be assembled, colored, and ordered. Our implementation uses an algorithm wherein each processor sends the root processor P_0 its subdomain neighbor adjacency list. P_0 then assembles the global subdomain graph and broadcasts this graph to all processors. Finally, each processor employs an identical algorithm to color and order the nodes in the graph.

Subdomain graph formation involves both a reduction and fan-out operation and hence, like the all-to-all communication in the previous step, is not perfectly scalable. However, given a priori information about the processor grid and discretization technique, communication costs can sometimes be entirely eliminated. For example, given a structured grid and the number of processors in the x, y, and z directions, one can design simple algorithms to identify all neighbors and color the subdomain graph. Note here that knowledge of the discretization technique is essential. For example, in a 2D processor grid with a five-point stencil discretization

of PDEs, each processor has (at most) four neighbors. For a nine-point discretization, however, a processor has (at most) eight neighbors. In the former case two colors will suffice for coloring, while the latter case require four colors.

Once all local boundary rows have been identified, each processor locally computes a permutation that orders its interior nodes first, then orders its boundary nodes. Our convention is that interior nodes maintain the same relative ordering before and after ordering, and similarly for boundary nodes; however, other options are possible. For example, one could apply an RCM or nested dissection ordering to the interior nodes, and/or unsymmetric row reorderings that may increase stability.

After local reordering processors must exchange the boundary node portion of their permutation with their neighbors in the subdomain graph. Our model implementation uses the following strategy. P_u sends to all its neighbors in the subdomain graph the total number of boundary nodes in its subdomain, and receives from all its neighbors the boundary node count in their subdomains. Processors can then allocate buffers of the proper length to receive the permutation lists from their neighbors. Finally, the actual permutation lists are exchanged via asynchronous send/receive pairs.

Common practice is to store permutation information in an array, for example, $j = oldToNew[i]$ indicates that matrix row i is to be permuted to position j. This approach would require all processors to maintain arrays of length m, where m is the global number of rows in the matrix. Since this approach is not scalable, our implementation stores local permutation information in a lookup array and external permutation information (that received from other processors) in a hash table.

FACTORIZATION

Figure 26 contains a detailed description of the PILU factorization algorithm. This is an expanded version of the high level description presented in Chapter 4, Figure 19. The more comprehensive version presented here includes details such as buffer allocation, additional communication pattern details, and access to external rows during factorization. The factorization proper (steps 1 and 3) can potentially employ many variants of ILU(ℓ) or ILUT.

For factoring interior rows our implementation uses algorithms that are essentially unchanged from their sequential counterparts. Scalable factoring of boundary

Input: Adjacency graph representation of a matrix A, and its corresponding subdomain graph object.
Output: The incomplete factors of the coefficient matrix.

1. Factor interior rows.
2. For all lower ordered neighbors in the subdomain graph: a. Receive nonzero counts for the upper triangular portion of the neighbor's boundary rows. b. Allocate buffer space to receive rows, and insert the buffer space addresses into a hash table. c. Wait for all external rows to arrive.
3. Factor boundary rows. a. Previously factored *locally owned rows* to be merged with the row being factored are referenced via standard sparse row storage scheme. b. Previously factored *external rows* to be merged with the row being factored are referenced via hash lookup.
4. For all higher ordered neighbors in the subdomain graph: a. Send the nonzero count for the upper triangular portion of this subdomain's boundary rows. b. Send the upper triangular column indices and values for this subdomain's boundary rows.
5. Wait for all sends to complete.

FIG. 26. *PILU factorization algorithm.*

rows, however, is a bit tricky. When factoring a matrix row i, common practice in the sequential world is to expand the row into a working vector of length n, where n is the number of global rows and columns in the matrix. However, this approach is inherently unscalable since we cannot expect processors to have sufficient resources to allocate arrays of length n. We therefore insert, retrieve, and update column elements in the row being factored through use of a SortedList object. This object only requires $O(c)$ storage, where c is the number of nonzeros in the factored row. The tradeoff is that the cost of the insert, retrieve, and update operations is potentially much more expensive than the $O(1)$ costs that would be incurred were we able to use an array.

Currently, our SortedList object uses a simple list traversal to locate column indices. This is arguably horrible, since it results in each lookup operation's cost being bounded by $O(c^2)$. This cost could be reduced by using a more complicated binary search, however, we believe our experimental results demonstrate that the cost in practice is unlikely to be excessive.

During factorization, matrix row indices and values are transmitted between processors. For simplicity, our implementation uses two send/receive pairs, one for the integer data and one for the floating point data. Packing the integer and floating point data into a single send/receive pair should reduce communication costs.

TRIANGULAR SOLVE SETUP

In the triangular solve setup phase each processor informs its neighboring processors in the subdomain graph of the vector indices it requires for completing its forward triangular solve (solving $Lw = x$ for w) and its backward triangular solve (solving $Uy = x$ for y). The lists of requested indices are exchanged a single time, in this setup phase; the corresponding values will be exchanged repeatedly, whenever preconditioner application is invoked by the Krylov solver. Figure 27 summarizes the triangular solve setup algorithm.

The triangular solve setup phase is similar to the subdomain graph setup phase in that, for structurally unsymmetric cases, a processor cannot necessarily identify which nodes it needs to receive or send to whom by scanning its adjacency list representation of the L and U factors. As before, if the filled matrix $F = L + U - I$

Input: The incomplete L and U factors in adjacency list representation, the corresponding subdomain graph object, and the number of processors p.
Output: On return, persistent communications have been initialized.

1. Initialize elements in arrays *counts_L*[] and *counts_U*[] of length p to zero.

2. Each processor P_u scans its adjacency lists of L and U to determine the boundary nodes required from other processors. (At the completion of this step, *count_L*$[P_v]$ or *count_U*$[P_v]$ can be nonzero only if P_u and P_v are neighbors in the subdomain graph.) a. For each external node i, where i is required for a forward triangular solve involving L, and i is owned by P_v, increment *count_L*$[P_v]$, and insert i in the list of nodes required from P_v. b. For each external node i, where i is required for a backward triangular solve involving U, and i is owned by P_v, increment *count_U*$[P_v]$, and insert i in the list of nodes required from P_v.

3. Processor P_u receives the number of nodes which its neighboring processors P_v require for their triangular solves. (These are the counts from the *counts_L*[] and *counts_U*[] arrays.)

4. Each Processor P_u allocates buffer space for receiving the lists of indices required by its neighboring processors P_v for their triangular solves.

5. Processors exchange their lists of requested indices via asynchronous send/receive pairs.

6. Each processor P_u allocates buffer space for receiving the values associated with the index lists it requested.

7. Start persistent communication for sending and receiving the values.

FIG. 27. *PILU triangular solve setup algorithm.*

is structurally symmetric, this information can be used to reduce communication costs. Note that F may be structurally unsymmetric even when A is both structurally and numerically symmetric. F may become unsymmetric for many reasons, e.g., due to roundoff error; when small values in A are discarded during factorization; when values in A are row scaled; or when ILUT is the factorization algorithm of choice.

5.3 PRECONDITIONER APPLICATION

Figure 28 contains a description of the preconditioner application (triangular solve) algorithm. The solves are similar to those used during the sequential solution of $LUy = x$ for y. The difference here is that values in the working vector w and the solution vector x that are associated with boundary nodes must be sent to and received from neighboring processors in the subdomain graph. Indices in w will be received from lower ordered neighbors, and sent to higher ordered neighbors. Indices in y will be received from higher ordered neighbors, and sent to lower ordered neighbors.

This nearest-neighbor communication uses the persistent communications that were established during the triangular solve setup phase. Additionally, as in the factorization phase, we overlap computation and communication. Unlike the factorization phase, where hash tables were used to access local copies of external data, in this phase we employ a global-to-local mapping strategy that permits the direct lookup of external values in an array. (This strategy is also used in the triangular solve setup phase.) Our mapping strategies, and the algorithms employed to send, receive, and access external vector values, were adopted from code and ideas developed by Edmond Chow for the mat-vec multiply functions in his ParaSails code [17].

5.4 PERFORMANCE EXPECTATIONS

Table 8 summarizes the communication patterns discussed in this chapter's preceding sections, and additional considerations discussed in this section. We have divided communication into four categories, and indicated which patterns our implementation employs in the various setup and application stages. The patterns

Input: The incomplete L and U factors, and a vector x (the vector to which the preconditioner is to be applied).
Output: The preconditioned vector y, where $LUy = x$.

1. Permute local portions of the the vector x.
2. Start receives from higher and lower ordered neighbors in the subdomain graph as appropriate.
3. Perform forward triangular solve, $Lw = x$ for w, on interior nodes.
4. Wait for values from w from lower ordered neighbors.
5. Perform forward triangular solve, $Lw = x$ for w, on boundary nodes.
6. Send values from w to higher ordered neighbors.
7. Wait for values in y from higher ordered neighbors.
8. Perform backward triangular solve, $Uy = w$ for y, on boundary nodes.
9. Send values from y to lower ordered neighbors.
10. Perform backward triangular solve, $Uy = w$ for y, on interior nodes.
11. Unpermute local portions of the preconditioned vector y.
12. Wait for all sends to complete.

FIG. 28. *Preconditioner application (triangular solves).*

TABLE 8

Communication pattern and scalability summary. The four communication patterns are listed, from left to right, with reference to increasing globality; the more global the pattern, the poorer we expect will be the algorithmic scalability. In the rightmost column, larger comp/comm ratios are also indicative of better scalability.

Setup task	**Communication Pattern**				**Comp/comm ratio**
	none	peer-to-peer	reduction	all-to-all	
exchange ownership information				x	low
identify boundary nodes				x	med
form, color, order subdomain graph			x		high
local reordering	x				infinite
identify neighbors in subdomain graph	x				infinite
exchange permutations		x			medium
symbolic and numeric factorization		x			low
triangular solve setup		x			low

are listed with reference to "increasing globality." The more global the pattern, the poorer we expect will be the algorithmic scalability. The first pattern is *none*, or purely local operation. The second pattern is *peer-to-peer*, which we have also referred to as nearest-neighbor, and is accomplished through calls to `MPI_Irecv` and `MPI_Isend`. The third pattern is *reduction*, which is a global operation to which each processor contributes data, with the result transmitted to all processors. The contributed data may be scalar, as in the case of inner product computation, or multi-valued, as in the case of subdomain graph formation. Our implementation uses calls to `MPI_Reduce` for scalar data. For subdomain graph formation we employ calls to `MPI_Irecv` and `MPI_Isend`, with all `MPI_Irecv` calls at the root processor, followed by `MPI_Bcast`. The fourth pattern is *all-to-all*, wherein each processor sends every other processor a unique piece of information; this pattern makes use of the `MPI_Alltoall` call.

In addition to communication patterns, performance and scalability are influenced by the computation/communication (comp/comm) ratio. In general, the larger the comp/comm ratio, the better the expected performance. Note that the concept of comp/comm ratio is orthogonal to the concept of granularity. Granularity refers to the amount of independent work that can be performed before or between communication stages. The comp/comm ratio, on the other hand, refers to the total amount of computation and the total amount of communication; it says nothing about how the computation and communication are interspersed.

Comp/comm ratios reflect globality as well as the total amount of communication. For example, if a processor must perform some computation then transmit the result to all other processors, we say that section of code has a lower comp/comm ratio than it would were the processor only required to transmit the result to a subset of the other processors.

Comp/comm ratios are a feature of hardware as well as software. For example, the Pentium chips in the Coral Beowulf cluster at ICASE are blazingly fast computationally, but are connected together by Fast Ethernet, which nowadays is considered somewhat slow . In contrast, the processors on ASCI Blue Pacific are computationally slower than the Pentiums, but the interconnect is faster. Hence we

Coral has a second interconnect network, Gigabit Ethernet, which is fast. However, our experiments and hence discussion only made use of the slower Fast Ethernet interconnect.

categorize the comp/comm ratio of the Coral cluster as high, and the comp/comm ratio of the ASCI Blue Pacific platform as low.

Amortization of the preconditioner setup time also influences expected performance. We amortize the setup time by dividing it over the total number of Krylov iterations. Linear systems whose solution requires a relatively small number of iterations—perhaps because we only wish to knock down their residuals by a couple of orders of magnitude—have poor amortization behavior. For these systems preconditioner setup time may become a performance limiter. On the other hand, applications involving nonlinear systems typically require the solution of a series of linear systems having similar structure. In these contexts subdomain graph setup, symbolic factorization, and triangular solve setup steps need only be performed when solving the first system. For subsequent systems this information can be reused, and we need only perform numeric factorization. This type of application has favorable amortization behavior. As previously noted, special symmetry and structural information can sometimes be used to strengthen amortization behavior.

An important caveat concerning our model implementation is here in order. Although we believe that our model PILU implementation is reasonably efficient, there are several places, some of which were pointed out in preceding sections, where, to ensure correctness and simplify the implementation, less than optimal algorithms and MPI communication patterns were employed. However, virtually all such shortcuts were taken during implementation of the preconditioner setup phase. In contrast, we believe that our implementation of the triangular solve (preconditioner application) phase is highly efficient.

5.5 EXPERIMENTAL RESULTS

Experimental scalability results for up to 400 processors on ASCI Blue Pacific appear in Figures 29 and 30 and in Tables 9 through 12 (machine configuration information for this platform appears in Appendix B). Results are for a PDE discretized with a five-point stencil (three-point central differencing) on a 2D grid. The problem was scaled from 65.5K unknowns to 26.2M unknowns; in all cases each processor was assigned a square 256 subgrid (65.5K unknowns per processor). Figures 29 and 30 are plots of the triangular solve timings that appear in the rightmost column of the tables, which are averages over 20 iterations of CG.

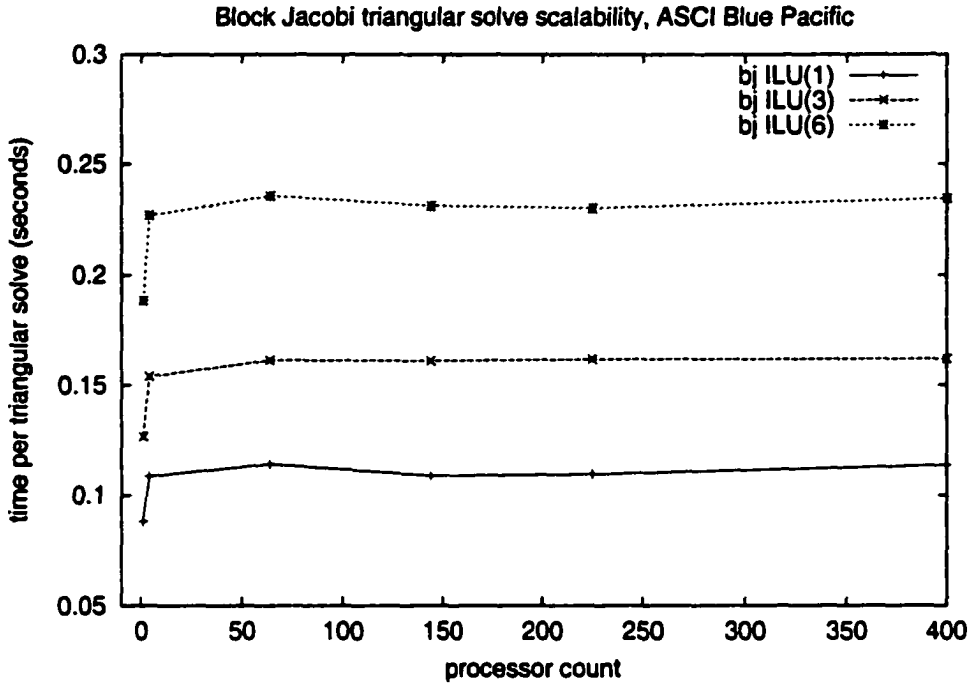


FIG. 29. *Block Jacobi triangular solve scalability (ASCI Blue).*

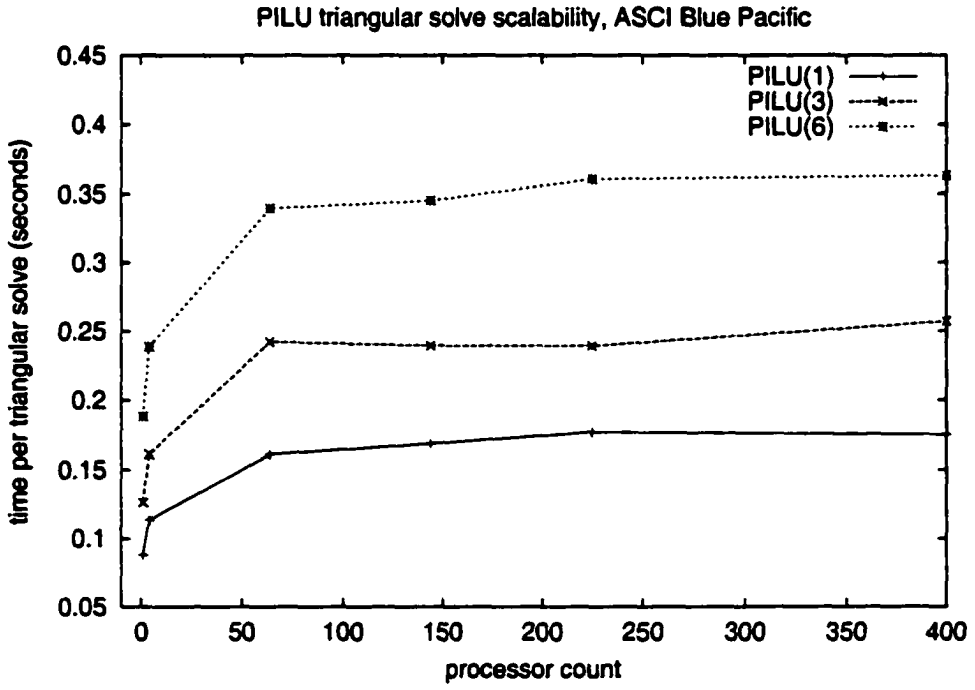


FIG. 30. *PILU triangular solve scalability (ASCI Blue).*

Each processor independently timed the various stages using `MPI_Wtime`; reported timings are the maximum over all processors.

Although these results are for structurally symmetric matrices arising from discretizations of PDEs on regularly structured grids with partitioning such that the subdomain graph itself was also structured and symmetric, none of this a priori "special" information was made visible during execution. In other words, for experimental purposes both the matrix and subdomain graph were assumed unsymmetric and unstructured. As discussed above, use of this special information would have resulted in a large diminution of the subdomain graph setup time and would also have reduced triangular solve setup times.

Figures 29 and 30 show that both PILU and Block Jacobi ILU triangular solves have nearly perfect scalability over a wide range of scaled problem sizes. The four-processor PILU case executed more quickly than the larger runs since in the four-processor case each subdomain had only two neighbors in the subdomain graph. For the larger runs most processors have four neighbors, and hence increased communication costs.

Our experience on ASCI Blue Pacific is that timings for identical runs show considerable variance. For our reported results each run (each horizontal entry in the tables) was repeated on three separate occasions. The figures in the tables are for the fastest of the three timings, with selection based on the "application per iteration" entry. Timing variance is readily discernible in the "subdomain graph setup" column in the tables. For a given number of processors all entries in this column should be identical, since the factorization level does not influence subdomain graph setup time.

5.6 ANALYTIC COMPARISON OF PILU AND BLOCK JACOBI

Results and discussion in this and other chapters show that both Parallel ILU and Block Jacobi ILU are highly scalable and effective preconditioning techniques. PILU is more powerful than Block Jacobi in the sense that, for preconditioners of approximately the same size (having similar nonzero counts) and level, PILU preconditioned systems require fewer iterations to converge. In contrast, Block Jacobi is far easier to implement and has lower setup cost and application cost per iteration. In practice the choice between PILU or Block Jacobi preconditioning is

TABLE 9

Scalability data, 400 processors (ASCI Blue). Timings in seconds. Local grid size is 256 x 256 (65.5K unknowns per processor, 26.6M global unknowns). Application timings are averages over 21 solves. The "Setup, solve" column represents the one-time cost for setting up communications for transmittal of boundary nodes; this stage is not applicable to Block Jacobi, hence the zero timings.

		Setup			Application
strategy	level	subdomain graph	factorization	solve	per iteration
PILU	0	1.39	1.70	0.84	0.1547
	1	1.59	2.04	1.03	0.1751
	2	1.77	2.74	1.46	0.2049
	3	1.53	3.68	1.91	0.2570
	4	1.49	5.24	2.48	0.2898
	5	1.64	7.23	3.54	0.3363
	6	1.65	9.63	4.75	0.3627
Block	0	0.01	1.40	0	0.0973
Jacobi	1	0.03	1.47	0	0.1134
	2	0.01	1.67	0	0.1305
	3	0.02	1.80	0	0.1619
	4	0.01	2.27	0	0.1887
	5	0.01	2.58	0	0.2119
	6	0.01	2.81	0	0.2343

TABLE 10

Scalability data, 225 processors (ASCI Blue). Timings in seconds. Local grid size is 256 x 256 (65.5K unknowns per processor, 14.7M global unknowns). Application timings are averages over 21 solves. The "Setup, solve" column represents the one-time cost for setting up communications for transmittal of boundary nodes; this stage is not applicable to Block Jacobi, hence the zero timings.

		Setup			Application
strategy	level	subdomain graph	factorization	solve	per iteration
PILU	0	1.27	1.62	0.80	0.1487
	1	1.27	2.00	1.02	0.1769
	2	1.58	2.62	1.34	0.2012
	3	2.14	3.62	1.81	0.2393
	4	1.47	5.21	2.55	0.2886
	5	1.35	7.23	3.64	0.3244
	6	1.41	9.32	4.46	0.3605
Block	0	0.01	1.31	0	0.0960
Jacobi	1	0.01	1.45	0	0.1092
	2	0.01	1.73	0	0.1294
	3	0.01	1.77	0	0.1617
	4	0.01	2.24	0	0.1861
	5	0.01	2.48	0	0.2076
	6	0.01	2.71	0	0.2299

TABLE 11

Scalability data, 64 processors (ASCI Blue). Timings in seconds. Local grid size is 256 x 256 (65.5K unknowns per processor, 4.2M global unknowns). Application timings are averages over 21 solves. The "Setup, solve" column represents the one-time cost for setting up communications for transmittal of boundary nodes; this stage is not applicable to Block Jacobi, hence the zero timings.

		Setup			Application
strategy	level	subdomain graph	factorization	solve	per iteration
PILU	0	1.20	1.49	0.44	0.1343
	1	1.05	1.87	0.64	0.1611
	2	1.37	2.45	1.16	0.1923
	3	1.24	3.41	1.47	0.2330
	4	1.23	5.11	2.17	0.2677
	5	1.05	6.94	2.87	0.3062
	6	1.35	9.47	4.27	0.3397
Block	0	0.01	1.22	0	0.0910
Jacobi	1	0.01	1.44	0	0.1091
	2	0.01	1.61	0	0.1247
	3	0.01	1.69	0	0.1605
	4	0.01	2.22	0	0.1803
	5	0.01	2.63	0	0.2089
	6	0.01	2.94	0	0.2355

TABLE 12

Scalability data, 4 processors (ASCI Blue). Timings in seconds. Local grid size is 256 x 256 (65.5K unknowns per processor, 262K global unknowns). Application timings are averages over 21 solves. The "Setup, solve" column represents the one-time cost for setting up communications for transmittal of boundary nodes; this stage is not applicable to Block Jacobi, hence the zero timings.

		Setup			Application
strategy	level	subdomain graph	factorization	solve	per iteration
PILU	0	0.99	1.38	0.32	0.0986
	1	0.99	1.66	0.42	0.1138
	2	0.99	2.11	0.54	0.1307
	3	0.99	2.76	0.80	0.1613
	4	0.99	3.98	1.17	0.1902
	5	1.00	5.33	1.75	0.2113
	6	0.99	7.11	2.52	0.2389
Block	0	0.01	1.26	0	0.0913
Jacobi	1	0.01	1.34	0	0.1084
	2	0.01	1.58	0	0.1234
	3	0.01	1.81	0	0.1540
	4	0.01	2.35	0	0.1788
	5	0.01	2.59	0	0.2033
	6	0.01	2.92	0	0.2270

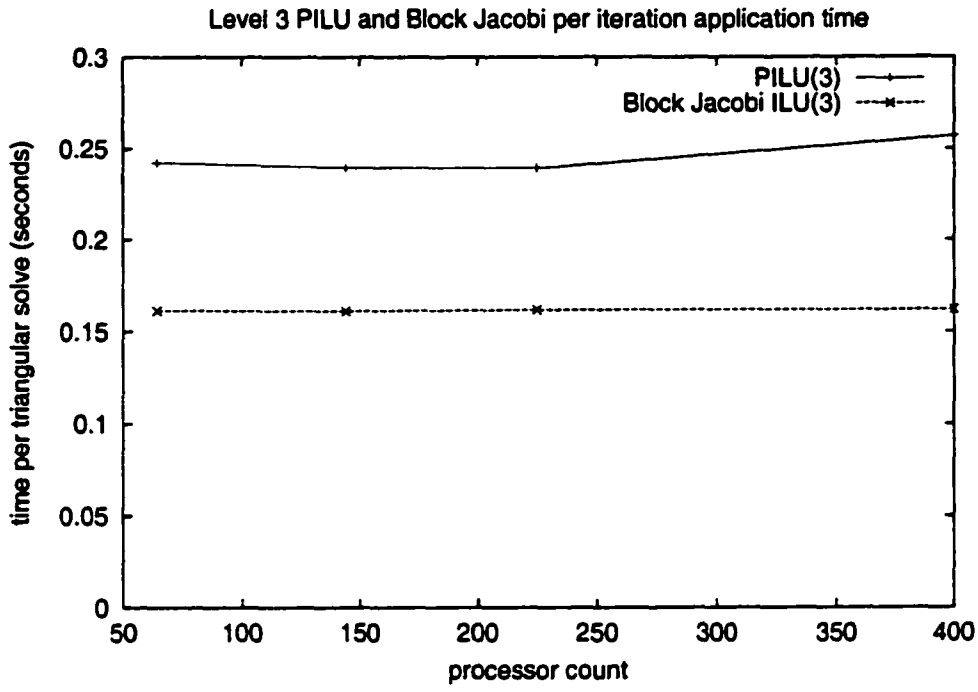


FIG. 31. *Experimental α computation (ASCI Blue). The upper line in the graph represents $c + \alpha$ values; the lower line represents c values; α is the vertical distance between the two lines, here approximately 0.08.*

likely to be determined experimentally. In this section, however, we formulate an analytic performance comparison expression.

Let i_p represent the number of iterations required for convergence of some linear system when preconditioned with PILU, and let i_b represent the number of iterations required for convergence when the linear system is preconditioned with Block Jacobi ILU. We make the simplifying assumption that the time per iteration of an unpreconditioned Krylov method (such as CG) remains constant when problem size and processor number are scaled per our discussions throughout this chapter. Our experiments show that both Block Jacobi and PILU application time per iteration also remain (nearly) constant over a wide range of problem sizes. We therefore let c represent the cost per iteration of one iteration of a Krylov solve preconditioned with Block Jacobi, and $c + \alpha$ the cost per iteration of a similar iteration preconditioned with PILU.

TABLE 13
Experimental α values for 2D five-point problems (ASCI Blue).

Level	α
0	.0527
1	.0677
2	.0718
3	.0776
4	.1025
5	.1168
6	.1306

Experimental measurements of c and $c+\alpha$ are graphed in Figure 31. The results are for a 2D, 5-point stencil problem run on ASCI Blue Pacific with local grid size of 256 ($\approx$ 65K unknowns per processor). Pictorially, α is the distance between the two plotted lines. Table 13 lists values for α based on the experiments presented in the preceding section. Since the lines for preconditioner application timing for both PILU and Block Jacobi ILU were seen to remain nearly flat for processor counts ranging from 64 through 400, the values in the table were computed using data from the 225 processor runs.

Disregarding preconditioner setup time, total solution time for a scaled system when Block Jacobi preconditioning is employed is ci_b,and total solution time when PILU preconditioning is employed is $(c+\alpha)i_p$. PILU should then be the preconditioning method of choice when

$$i_p(c+\alpha) < i_b c$$

which upon rearranging gives

$$\frac{i_p}{i_b} < \frac{c}{c+\alpha}. \tag{11}$$

Now let us assume we have a given grid and processor topology upon which a series of PDEs will be discretized using a five-point stencil. The right-hand side of Equation 11 is totally independent of any of the PDEs; it is a function

of combinatorics only. Moreover, it remains constant as problem size is scaled upwards. It will only change if each processor's local grid size changes, or if some different discretization method is employed.

In contrast, the ratio on the left-hand side is determined by the numerical properties of the particular PDE (along with its boundary conditions) being solved, when global grid and processor topology are held constant. If we solve a particular PDE repeatedly, while scaling the problem size, the ratio will also be influenced by the global grid size. In either case, however, the left-hand side remains constant.

Another interpretation of Equation 11 is that the smaller the α, the greater will be the advantage in employing PILU. The value of α should shrink as subdomain/interior boundary node ratios increase.

5.7 END-USER SCALABILITY PERSPECTIVE

Much of the discussion in this chapter has centered around details that may remain opaque to end-users of the Euclid library. Faced with a multitude of preconditioning codes from which to choose, users need quick answers to questions such as "does the code scale for my problem?" and "is Euclid more effective than library *XYZ* in reducing solution time for my problem?"

These are difficult questions to answer in an analytic sense, for any except the simplest of problems. Since preconditioner setup time can be amortized over a large number of Krylov iterations, the main determinant of solution time is the number of Krylov iterations multiplied by the time for each Krylov iteration. The number of iterations depends on mathematical properties of the system being solved and the preconditioner. Timing for a single Krylov iteration is dependent on many structural factors of the matrix, as discussed in this and the following chapter, and defies simple summarization. In the future we hope to develop software that will quickly analyze matrix structural properties, and use this as a guide in determining timing per Krylov iteration.

Therefore, an answer to the question, "should I use Euclid?" is likely to remain largely experimental. However, the following guidance is pertinent.

If ILU preconditioning is not effective for sequentially-solvable problems that are similar to the problem of interest, then PILU is unlikely to be effective for larger problems that can only be solved in parallel, and users should investigate

other preconditioning approaches. If ILU preconditioning is effective for similar sequentially-solvable problems, and if it is known that the problem can be partitioned in such a way that subdomains have large interior/boundary node ratios, then PILU preconditioning is likely to be effective.

CHAPTER 6

PARTITIONING AND INTERIOR/BOUNDARY NODE RATIOS

In this chapter we are concerned with two issues that affect PILU performance. The first issue is partitioning, by which we mean the manner in which a matrix and its incomplete factor are distributed amongst the processors (here, as elsewhere, we use "processors" synonymously with "subdomains"). The second issue is subdomain size (the number of matrix rows or unknowns in each subdomain). Subdomain size is a function of both processor number and global problem size. For our purposes partitioning and subdomain size are similar in their performance effects since changes in either can modify subdomain interior/boundary node ratios.

For the scalability studies in this chapter, we are interested in examining what happens when we either (1) hold the problem size constant, while increasing the number of subodmains (processors); (2) hold the number of processors constant while increasing problem size; or (3) hold both problem size and subdomain count constant while varying the partitioning strategy. More fundamentally, we want to examine how performance is affected by varying interior/boundary node ratios.

6.1 PARTITIONING AND ORDERING BACKGROUND

In the purest sense, *partitioning* is the division of the elements in a set into two or more disjoint subsets (sometimes called blocks). In graph theory the set to be partitioned is frequently taken to be the set of vertices in a graph. Due to our construction wherein each vertex in a graph is associated with a unique row in a matrix, we speak of "partitioning a matrix," by which we mean the division of matrix rows amongst two or more subsets, called subdomains.

Within this work we have implicitly assumed what we might call a *domain decomposition constraint.* This is a constraint on how the set of vertices in $G(A)$ may be partitioned. Within each subdomain, every vertex is constrained to be edge-connected to at least one other vertex in the same subdomain, and subdomains are constrained to have large interior/boundary node ratios. The constraint is desirable since large interior/boundary node ratios are essential for getting good performance

out of PILU, and the largest ratios can usually be attained when nodes within a partition block are connected. However, the assumption is not strictly necessary, either theoretically or in our model implementation.

In the interests of full disclosure we note that the concept of partitioning a graph or matrix is distinct from the concept of ordering (a.k.a, reordering) a graph or matrix, though we have sometimes blurred the lines of demarcation. For example, we have said that "each subdomain's nodes (in $G(A)$) are ordered contiguously," and "each processor is assigned a rectangular section of a matrix, i.e., a contiguously numbered set of rows." In a more formal sense both these statements reflect the operations of (1) partitioning a graph or matrix, followed by (2) ordering the vertices or rows such that all members of each subset are numbered contiguously, and (3) mapping each subdomain to a processor. Note, however, that one can sometimes get good results by reversing steps (1) and (2), i.e., by performing a global ordering then partitioning. For example, for some matrices one can attain high interior/boundary node ratios by first computing an RCM ordering for $G(A)$, then employing the partitioning strategy of assigning contiguously numbered sections of matrix rows to subsets.

6.2 DOES PILU PARTITION?

The high level PILU description presented in Chapter 4, Figure 19 states that PILU begins by partitioning a matrix into subdomains and then distributing (mapping) the subdomains amongst the processors. In Chapter 5, however, we said that our model implementation takes as input a matrix that is assumed to already have been partitioned and distributed. Since readers justifiably may find this confusing, some clarification is in order.

The presentation in Chapter 4 is concerned with what we would theoretically like to be able to do in order to get the best possible advantage from PILU preconditioning. In contrast, the discussion in Chapter 5 is concerned with the model implementation, and is reflective of interface and memory limitations that must be dealt with during the software design process. Our model implementation was designed to conform to the following specifications.

1. When Euclid's services are invoked by a Krylov solver, the matrix A in the linear system $Ax = b$ may only be accessed by a generic interface abstraction of the form: `getRow(rowIN, lenOUT, const colOUT[], const valsOUT[])`.

2. Euclid should not make a duplicate copy of the matrix.

The `getRow()` abstraction implies that we are prohibited from directly accessing or manipulating the solver's private data structures. Although this may result in some loss of efficiency during factorization, it greatly eases the burden of designing interfaces between Euclid and solver libraries. For example, as of this writing PETSc's `petscmat.h` header file identifies 17 matrix storage formats. Employing the `getRow()` abstraction isolates Euclid from the details of these structures, which PETSc considers private and subject to change. The second specification is intended to reduce memory requirements.

To perform partitioning services, Euclid would need to duplicate the matrix (which could be accomplished using the `getRow()` interface), partition, and redistribute the copy. Unfortunately, this would give rise to added overhead during the triangular solves. Since we cannot partition and redistribute the matrix owned by the solver, the result would be that, whenever the solver calls Euclid with a vector to which the preconditioner is to be applied, the vector would have to be redistributed before the triangular solves could commence, and the preconditioned vector would have to be un-distributed before returning to the solver. These are details which, while in no way profound, raise complications that were judged beyond the purview of the model PILU implementation.

6.3 PARTITIONING AND INTERIOR/BOUNDARY NODE RATIO EFFECTS

Our finite difference matrix generator class produces matrices that are partitioned and distributed according to one of two strategies with respect to structured grids, as illustrated in Figure 32. A grid contains n nodes (unknowns) and is of dimension $\sqrt{n} \times \sqrt{n}$ for 2D problems, and $n^{1/3} \times n^{1/3} \times n^{1/3}$ for 3D problems. Matrices are distributed amongst p processors.

In *blocked* partitioning, subdomains consist of square (2D) or cubic (3D) subgrids. In this strategy the subdomain graph is itself regularly structured. For 2D

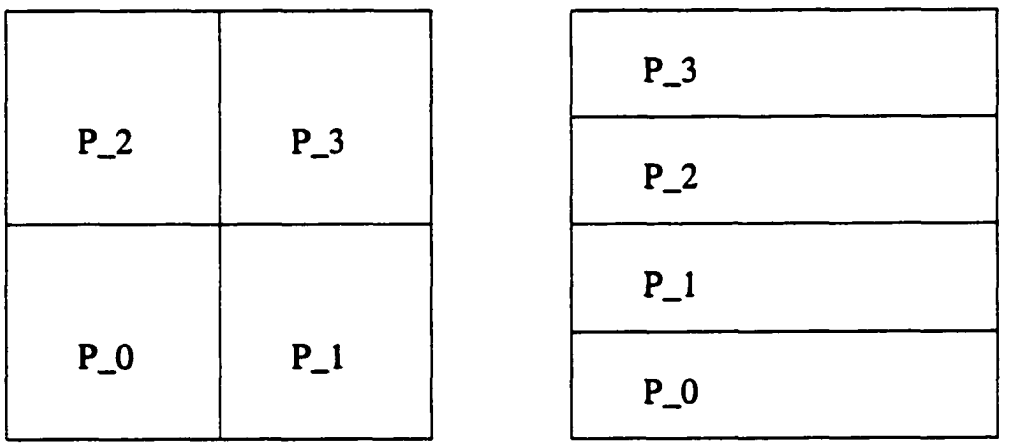


FIG. 32. *Block and striped partitioning strategy comparison. Block (left) and striped (right) partitioning strategies. An $n \times n$ grid (individual grid points are not shown) has been partitioned into four blocks and mapped to four processors.*

TABLE 14

Interior/boundary node ratios and subdomain graph dimensions for blocked and striped partitioning.

Partitioning Strategy	Subdomain Dimensions	Interior/boundary Node Ratio	Subdomain Graph Dimensions
2D blocked	$\sqrt{\frac{n}{p}} \times \sqrt{\frac{n}{p}}$	$\frac{1}{4}\sqrt{\frac{n}{p}}$	$\sqrt{p} \times \sqrt{p}$
2D striped	$\sqrt{n} \times \frac{\sqrt{n}}{p}$	$\frac{\sqrt{n}}{2p}$	$1 \times p$
3D blocked	$\sqrt[3]{\frac{n}{p}} \times \sqrt[3]{\frac{n}{p}} \times \sqrt[3]{\frac{n}{p}}$	$\frac{1}{6}\sqrt[3]{\frac{n}{p}}$	$\sqrt[3]{p} \times \sqrt[3]{p} \times \sqrt[3]{p}$
3D striped	$\frac{\sqrt[3]{n}}{p} \times \sqrt[3]{n} \times \sqrt[3]{n}$	$\frac{\sqrt[3]{n}}{2p}$	$1 \times p$

problems the subdomain graph has dimensions $\sqrt{p} \times \sqrt{p}$, and each subdomain asymptotically contains $4\sqrt{\frac{n}{p}}$ interior/boundary nodes. The interior/boundary node ratio is thus asymptotically $\frac{1}{4}\sqrt{\frac{n}{p}}$. These statistics, along with figures for 3D problems, are summarized in Table 14. Blocked partitioning is reasonably optimal in terms of maximizing interior/boundary node ratios.

In *striped* partitioning, subdomains consist of "long skinny" rectangles. The processor grid is of dimension $1 \times p$ for both 2D and 3D problems. For 2D, subdomains have dimensions $\sqrt{n} \times \frac{\sqrt{n}}{p}$, and contain $2\sqrt{n}$ boundary nodes. The interior/boundary node ratio is thus asymptotically $\frac{\sqrt{n}}{p}$. The striped partitioning strategy is less optimal than the blocked strategy in terms of maximizing the interior/boundary node ratios.

We can reason that a problem that has been stripe-partitioned will likely require more iterations to converge than when it has been block partitioned as follows. For Block Jacobi ILU, inter-subdomain coupling information is "lost" at interior/boundary nodes. Since stripe-partitioned problems have larger interior/boundary node ratios than their block-partitioned counterparts, we expect there will be greater coupling information loss in the former. Therefore, at least for well-behaved elliptic problems, we expect that striped partitioning will mandate a greater number of iterations, thus increasing execution time. In contrast, the preconditioners will contain fewer nonzeros, so application time per iteration will

shorten, which will decrease execution time.

PILU preconditioning preserves inter-subdomain coupling information, so it is less obvious whether changing from block to striped partitioning should affect iteration counts. We suspect there is some effect, since the different partitionings result in different matrix orderings and filled matrix structures. However, based on the evidence presented in chapter 4, we expect the reorderings to have scant effect when subdomains are relatively large. On the other hand, striped partitioning should definitely cause application time per iteration to increase, since the increased interior/boundary node count lessens parallelism and increases communication costs.

6.4 SUBDOMAIN SIZE AND INTERIOR/BOUNDARY NODE EFFECTS

Subdomain size (the number of unknowns per subdomain) is a function of both problem size and the number of subdomains into which the problem is partitioned. For a given partitioning strategy the "Interior/boundary Node Ratio" column in Table 14 shows this relationship.

Suppose we gradually decrease problem size while holding the partitioning strategy and subdomain count constant. We are interested in the comparative performance of PILU with Block Jacobi ILU on a per-iteration basis. Let t_b represent execution time for a single application of a Block Jacobi ILU preconditioner, and let t_p represent execution time for a single application of a PILU of the same level. We take as our metric the ratio t_b/t_p. As problem size increases the interior/boundary node ratio for PILU also increases, hence the triangular solve phase for PILU becomes relatively less expensive in terms of communication. We thus expect that the t_b/t_p ratio should also increase. (This relationship is shown experimentally in Figure 35, which is presented in the next section.)

6.5 EXPERIMENTAL RESULTS

Experiments in this section were conducted on the ASCI Blue Pacific platform. Machine configuration information for this platform appears in Appendix B.

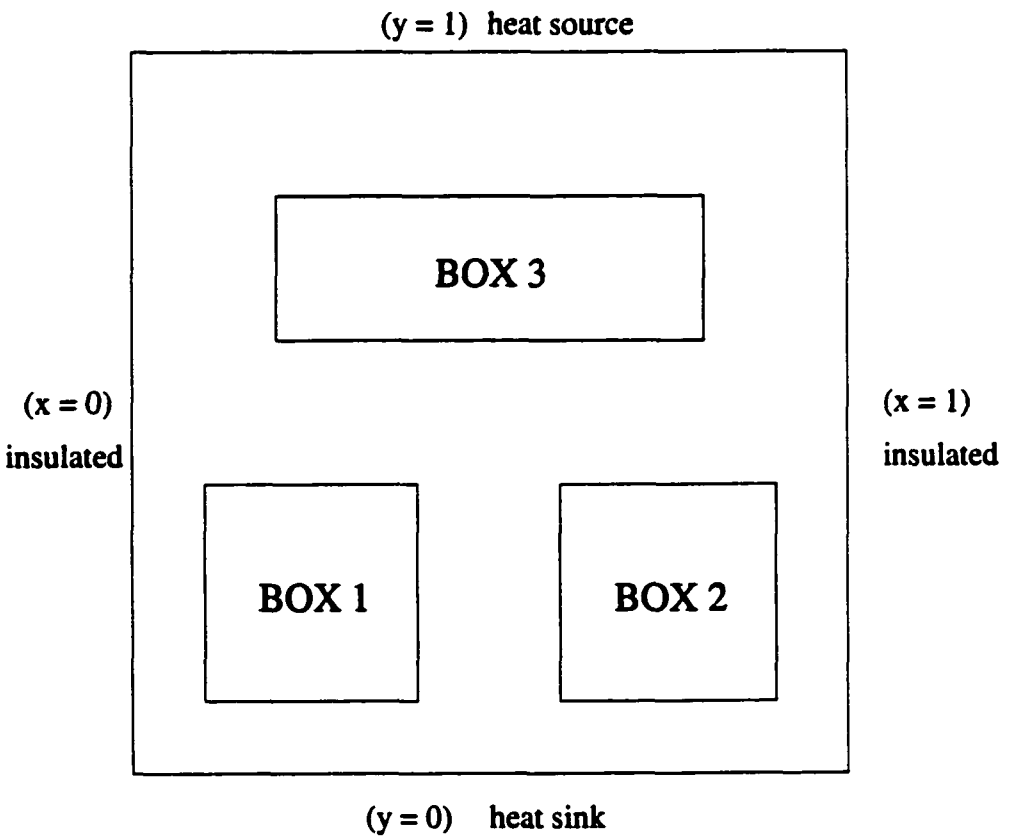


FIG. 33. *Laplacian 3-box problem description.*

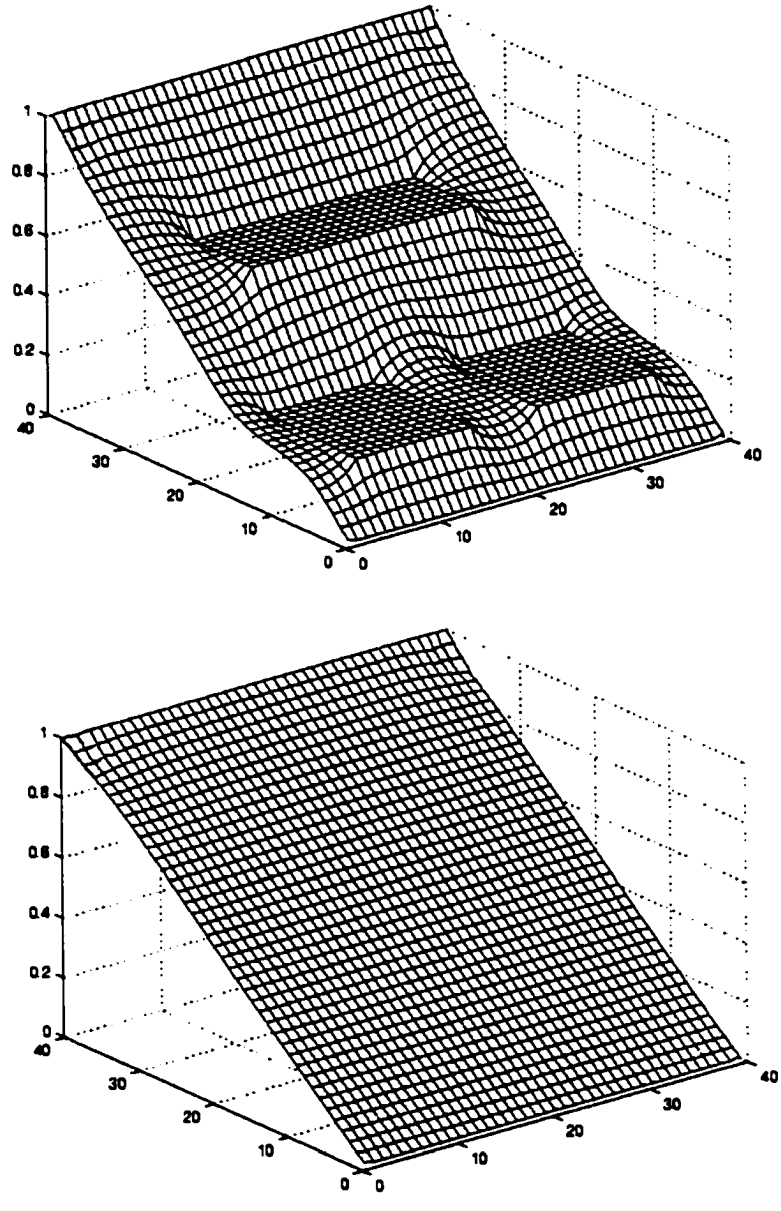


FIG. 34. *Solutions for the simple and 3-box Laplacian problems.*

6.5.1 PROBLEM DESCRIPTION

Although PILU was originally designed as a preconditioner for unsymmetric problems, in real life a great many problems of interest are, and will continue to be, elliptical in nature, symmetric, and positive definite. Some of these problems are quite difficult to solve efficiently, so it is altogether fitting to inquire how PILU performs on some of these seemingly simple problems.

Results in this section are based on the equation

$$\nabla \cdot \alpha \nabla u = 0,$$

discretized on the unit square with interior/boundary conditions

$$\begin{aligned} u(x,0) &= 0 \\ u(x,1) &= 1 \\ u_x(0,y) &= 0 \\ u_x(1,y) &= 0. \end{aligned}$$

This equation characterizes heat diffusion in a square plate with a heat source along one side, a heat sink along the opposite side, and perfect insulation along the two remaining sides. We examined two problems that are distinguished by the function $\alpha(x,y)$. The diffusivity constant for the *simple problem* is $\alpha(x,y) = 1$, which is Laplace's equation. The diffusivity constant for the *three-box problem* varies with respect to the location of three inset "boxes," and is illustrated in Figure 33. Figure 34 shows visualizations of the solutions of both problems.

It is instructive at the outset to get a feel for how difficult these problems are, compared to each other, and for varying ILU(ℓ) factorization levels. Table 15 shows comparative convergence data on a naturally ordered 100 × 100 grid (10K unknowns). The data shows that solving the 3-box problem is considerably more difficult than the simple problem, particularly with preconditioners with low factorization level values.

TABLE 15

Convergence comparison, simple and three-box problem. Results for discretization on a 100×100 *grid; convergence criterion was 1e8 residual reduction of preconditioned system; the Krylov method used was CG. The (*) indicates failure to converge after 5,000 iterations.*

	Problem	
Level	Simple	Three-box
1	526	*
3	62	220
5	33	121
7	25	84

6.5.2 EXPERIMENTAL RESULTS AND ANALYSIS

The simple and three-box problems were solved for two grid sizes, 350×350 (122.5K unknowns) and 700×700 (490K unknowns), on a 7×7 processor grid (49 processors) on the Coral cluster at ICASE. The problems were solved with both block and striped partitioning. Due to the nature of the problem—the placement and shape of the boxes for the three-box problem, and the boundary conditions for both problems—one expects that the orientation of the striped partitioning "skinny rectangles" should affect convergence behavior. We therefore used both x-striped and y-striped partitioning, where x- and y- indicate the orientation of the long dimension of the rectangular subdomains with respect to the standard labeling of the Cartesian coordinate axes.

Results appear in Tables 17 through 20. The problems were solved using PETSc's CG Krylov solver with Euclid preconditioning. Convergence criterion was 1e8 residual reduction of preconditioned system.

As in the previous chapter, no advantage was taken of problem symmetry or structure. The implication—recalling also that our preconditioner setup implementation has known inefficiencies—is that the timing values in both the factorization and total timing columns are higher than would be expected for a more optimized code. The trends indicated by the data in these columns, however, are most instructive, and should be a valid indicator of expectations for other implementations.

Preconditioner setup timings for both x-striped and y-striped partitionings should be in agreement, since the interior/boundary node ratios, and hence CPU operation count and message traffic, is identical in both cases. Similarly, the time per iteration (which is not shown, but can be calculated from data in the tables) should be similar. The orientation of the rectangular subdomains should only affect the number of iterations required for convergence. For y-striped partitioning, the long dimension of the subdomains is aligned in the the direction of flow so, particularly for Block Jacobi ILU, we expect this partitioning to converge faster (require fewer iterations) than the x-striped partitioning.

Table 16 lists the interior/boundary node ratios for the partitioning strategies and grid sizes reported on in this section. The larger ratios for the blocked cases indicate that these partitionings have higher comp/comm ratios, and thus we expect better performance.

TABLE 16
Interior/boundary node ratios for experimental problems.

Nodes per subdomain	Partitioning Strategy striped	blocked
2.5K	3.6	12.5
10K	7.1	25

Concerning x- and y-striped partitioning, from a physical viewpoint the most important coupling is parallel to the y-axis, that is, from the side where the source is located to the side where the sink is located. We therefore expect that y-striped partitioning should outperform x-striped partitioning in terms of iterations required for convergence, and this is in fact what the results show. (Since time per iteration for both x- and y-striped partitioning is identical, here the number of iterations directly reflects the comparative execution time.)

For nearly all cases block partitioning gave the fastest execution time. However, for a fill level of three, y-striped partitioning tended to outperform block partitioning; this was true for both PILU and Block Jacobi methods. For example, in Table 20, "BLOCKED PILU level 3" required 4282 iterations to resolve, while "y-STRIPED PILU level 3" only required 1318 iterations.

The higher level preconditioner conveyed greater benefit to the PILU runs than to the Block Jacobi ILU runs. For example, of the 12 combinations of partitioning (3 choices) and grid sizes (2 choices) and problem choice (2 choices), the fastest execution time for 11 of 12 cases for PILU was observed with level seven preconditioning. In contrast, for Block Jacobi preconditioning only three out of the 12 combinations was resolved fastest with level seven preconditioning.

Figure 35 shows relative performance of PILU and Block Jacobi ILU on a per-iteration basis for a block partitioned 2D problem on a 7×7 processor grid. As predicted, for small subdomain sizes (i.e., small interior/boundary node ratios) PILU performs poorly on a per-iteration basis compared to Block Jacobi ILU. As subdomain size increases we begin to see asymptotic behavior.

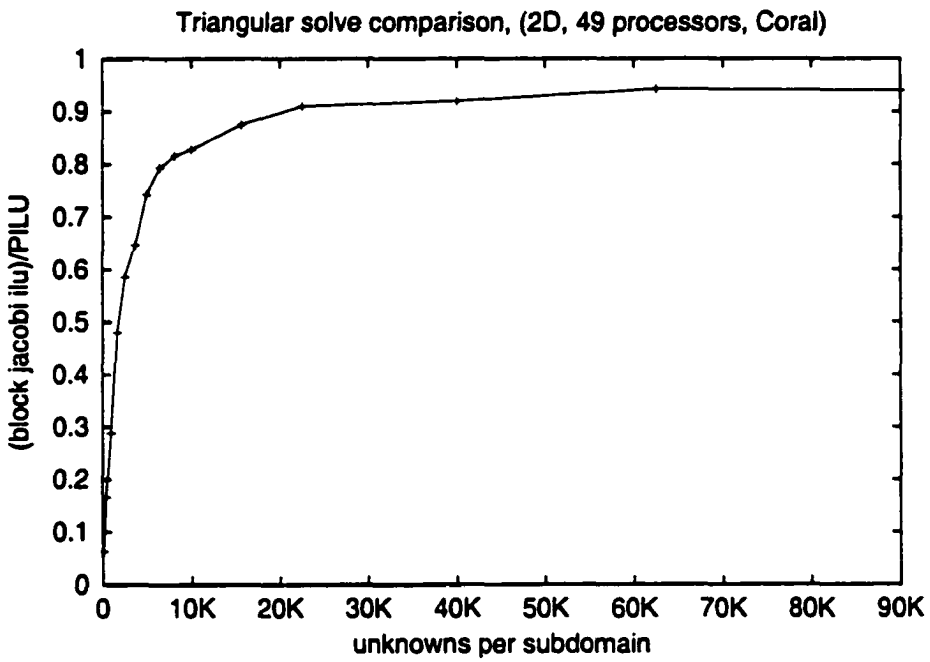


FIG. 35. *Relative performance of PILU and Block Jacobi ILU (ASCI Blue). Results are for a 7 × 7 processor grid on ASCI Blue Pacific. The plot shows t_b/t_p as a function of the number of local unknowns. (Global problem size varies from 4.9K to 4.4M unknowns).*

TABLE 17

Simple Problem (small), partitioning convergence effects (ASCI Blue). 122.5K unknowns. (*) indicates failure to converge after 5,000 iterations.

Partitioning Strategy	parallelization method	level	its	Timing (seconds) setup	solve	total
BLOCKED	pilu	1	4315	3.9	79.8	83.7
		3	221	4.4	4.7	9.1
		5	130	2.2	3.0	5.2
		7	93	3.7	2.4	6.1
	bj	1	*			
		3	364	.05	5.6	5.6
		5	239	.06	3.9	3.9
		7	222	.07	4.0	4.0
x-STRIPED	pilu	1	*			
		3	306	2.7	6.1	8.9
		5	203	6.2	4.8	11.0
		7	164	15.7	4.7	20.4
	bj	1	*			
		3	498	0.5	7.0	7.1
		5	429	0.5	6.5	6.6
		7	370	0.7	6.7	6.7
y-STRIPED	pilu	1	*			
		3	226	5.8	4.6	10.4
		5	174	6.2	4.3	10.6
		7	139	12.8	4.0	16.9
	bj	1	*			
		3	420	.05	5.8	5.9
		5	410	.06	6.5	6.5
		7	408	.08	7.1	7.2

TABLE 18

Simple Problem (large), partitioning convergence effects (ASCI Blue). 490K unknowns. (*) indicates failure to converge after 5,000 iterations.

Partitioning Strategy	parallelization method	level	its	Timing (seconds) setup	solve	total
BLOCKED	pilu	1	*			
		3	558	2.3	25.0	27.3
		5	238	4.3	12.4	16.7
		7	176	7.6	10.8	18.4
	bj	1	*			
		3	720	.19	28.3	28.5
		5	338	.25	15.8	16.0
		7	335	.30	17.9	18.2
x-STRIPED	pilu	1	*			
		3	568	5.5	28.5	34.0
		5	268	12.2	16.0	28.2
		7	214	25.2	15.3	40.6
	bj	1	*			
		3	1118	.20	44.1	44.3
		5	576	.25	26.6	26.9
		7	543	.30	29.0	29.3
y-STRIPED	pilu	1	*			
		3	404	5.4	20.3	25.7
		5	267	12.2	16.0	28.3
		7	209	25.0	15.1	40.1
	bj	1	*			
		3	564	.20	22.1	22.3
		5	512	.26	23.4	23.6
		7	500	.31	26.3	26.6

TABLE 19

Three-box problem (small), partitioning convergence effects (ASCI Blue). 122.5K unknowns. (*) indicates failure to converge after 5,000 iterations.

Partitioning Strategy	parallelization method	level	its	Timing (seconds) setup	solve	total
BLOCKED	pilu	1	*			
		3	1618	1.31	28.1	29.4
		5	556	2.1	11.0	13.1
		7	399	3.8	8.8	12.6
	bj	1	*			
		3	2645	.08	35.1	35.2
		5	922	.10	14.1	14.1
		7	819	.12	13.9	14.0
x-STRIPED	pilu	1	*			
		3	3099	5.8	62.7	68.5
		5	845	6.2	20.1	26.3
		7	629	15.9	18.2	34.1
		1	*			
		3	*			
		5	3388	.05	24.6	24.7
		7	1550	.07	26.5	26.6
y-STRIPED	pilu	1	*			
		3	801	5.7	16.1	21.8
		5	724	6.2	17.4	23.6
		7	494	12.9	14.2	27.1
	bj	1	*			
		3	1401	.05	18.7	18.8
		5	1580	.06	24.1	24.1
		7	1312	.07	22.2	22.3

TABLE 20

Three-box problem (large), partitioning convergence effects (ASCI Blue). 122.5K unknowns. (*) indicates failure to converge after 5,000 iterations.

Partitioning Strategy	parallelization method	level	its	Timing (seconds) setup	solve	total
BLOCKED	pilu	1	*			
		3	1618	1.31	28.1	29.4
		5	556	2.1	11.0	13.1
		7	399	3.8	8.8	12.6
	bj	1	*			
		3	2645	.08	35.1	35.2
		5	922	.10	14.1	14.1
		7	819	.12	13.9	14.0
x-STRIPED	pilu	1	*			
		3	3099	5.8	62.7	68.5
		5	845	6.2	20.1	26.3
		7	629	15.9	18.2	34.1
		1	*			
		3	*			
		5	3388	.05	24.6	24.7
		7	1550	.07	26.5	26.6
y-STRIPED	pilu	1	*			
		3	801	5.7	16.1	21.8
		5	724	6.2	17.4	23.6
		7	494	12.9	14.2	27.1
	bj	1	*			
		3	1401	.05	18.7	18.8
		5	1580	.06	24.1	24.1
		7	1312	.07	22.2	22.3

CHAPTER 7

SOLVING LARGE SYSTEMS

In chapter 5 we presented results with the intent of demonstrating that PILU scales well on a per-iteration basis. In chapter 6 we presented results with the intent of examining how partitioning and subdomain size affect boundary node ratios and hence PILU's performance. In this chapter we present results with the intent of answering the million dollar question, "does PILU perform really well at preconditioning large 2D and 3D systems?"

In chapters 5 and 6 our sample results were from the 2D realm. We chose to examine scalability and performance affects on 2D problems because one can perform meaningful scaling experiments with far fewer processors in much less time when using 2D as opposed to 3D grids. For example, for a 2D block partitioned problem one needs 9 processors in order to "fill the communication pipe," as opposed to 27 processors for 3D problems. (By "fill the pipe" we mean there is at least one processor that communicates with its neighbors to the maximum extent that any processor will ever have to do so as the problem is scaled upwards.)

The PC Coral cluster at ICASE tends to have much faster turnaround time and consistently delivers results with less variance than the ASCI Blue Pacific cluster at LLNL. For these reasons Coral is this author's platform of choice for gathering the extensive experimental data needed to identify statistical performance trends. However, the Coral cluster has far fewer CPUs than ASCI Blue Pacific. There are enough processors to identify scaling trends for 2D problems, but doing so for 3D cases is problematical.

In this chapter's first section we present complete solution results for a typical 3D convection-diffusion problem. In the second section we present complete solution results for a nonlinear radiative transport problem and the three-box problem introduced in chapter 6. In both sections our interest is in holding problem size per processor constant while varying the number of processors.

7.1 3D SYSTEMS

Results in this section are based on the convection diffusion equation:

$$-\varepsilon\Delta u + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} + \frac{\partial u}{\partial z} = g.$$

The diffusivity coefficient vector ε was 0.01 in the x and y directions and 1 in the z direction. Homogeneous boundary conditions were used for all trials. Derivative terms were discretized on the unit cube using 3-point central differencing on regularly spaced $n_x \times n_y \times n_z$ grids. The right-hand sides of the resulting systems, $Ax = b$, were artificially generated as $b = A\hat{e}$, where $\hat{e}$ is the all-ones vector. Systems were solved using PETSc's BICGSTAB Krylov method with Euclid preconditioning, with convergence criterion of 1e-8 residual reduction.

The local grid size for all runs was $40 \times 40 \times 40$, or 64,000 unknowns per processor. (We also experimented with local grid sizes of $30 \times 30 \times 30$ and $50 \times 50 \times 50$, and witnessed similar behavior.) As before, we made no use of known structural and symmetry information, and remind the reader that total execution time would be lower had we done so.

Tables 21 and 22 show iteration counts and timing results for PILU and Block Jacobi ILU for levels 0, 1, and 2. In all cases ILU(1) preconditioning gave the shortest execution time. Note that timings in the "per iteration" column are for one iteration of the preconditioned Krylov solve (i.e, the "solve" column divided by the "its" column). This differs from our usage in the tables in chapter 5, where we considered "per iteration" timing with respect to preconditioner application. Figure 36 illustrates scalability on a per-iteration basis for the preconditioned Krylov solve. The time per iteration does not scale as well as we would like (in which case we would see a flat line), however, the time required for one iteration only increases by approximately 50% when scaling from eight to 343 processors, (scaling global problem size from .5M to 22M). The line's slope is attributable to the global nature of the inner product computations in the Krylov solver. The PILU preconditioned iterations, although more expensive than those for Block Jacobi ILU, scale equally well.

Figure 37 plots total solution time (preconditioner setup plus the complete Krylov solve) as a function of processor count (equivalently, global problem size). PILU outperforms Block Jacobi as the problem is scaled upward, and more importantly has an increasing advantage as problem size increases.

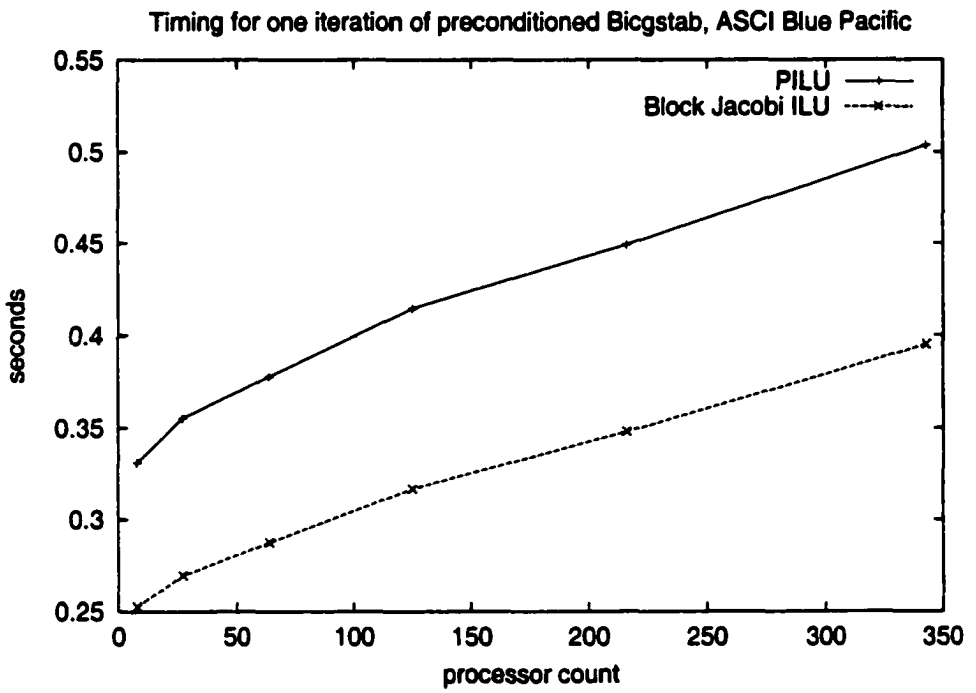


FIG. 36. *Scalability of 3D convection-diffusion problem (ASCI Blue). Data is for a single iteration of the preconditioned Krylov solve.*

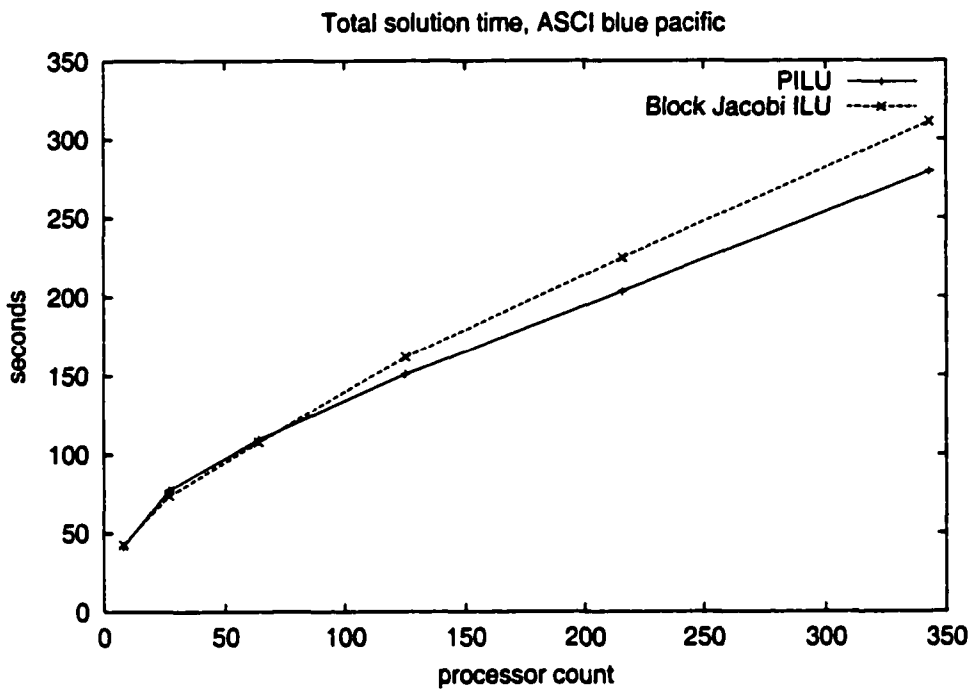


FIG. 37. *Convection-diffusion problem, total solution time (ASCI Blue).*

7.2 2D SYSTEMS

7.2.1 RADIATIVE TRANSPORT

Results in this section are for the nonlinear simplified 2D Radiative Transport problem

$$-\nabla \cdot (\alpha T^{\beta} \nabla T) = 0.$$

where $\beta = 2.5$ and $\alpha = 1.0$. The problem was uniformly discretized in the unit square with a five-point stencil and the boundary conditions

$$u(x,0) = 1.0$$
$$u(x,1) = 0.1$$
$$u_x(0,y) = 0$$
$$u_x(1,y) = 0.$$

This example, whose code is distributed with the PETSc source code [2], was executed on up to 400 processors. Each processor was assigned 10,000 grid points, so the largest instance solved contained 4 million unknowns. Solution in all cases required eight or nine Newton iterations and a linear system was solved each iteration. Preliminary investigations on smaller numbers of processors indicated that the shortest execution time was generally obtained with ILU(6) preconditioning.

Figure 38 shows execution timing result comparison when the linear systems were preconditioned using PILU and Block Jacobi ILU. Due to the long execution times for the larger problem sizes (e.g., the 400 processor run with Block Jacobi ILU preconditioning requires approximately 25 minutes), only results for a factorization level of six are reported. The results indicate that PILU increasingly outperforms Block Jacobi PILU as global problem size increases, and that the linear solves comprise the time-dominating kernel in this nonlinear problem.

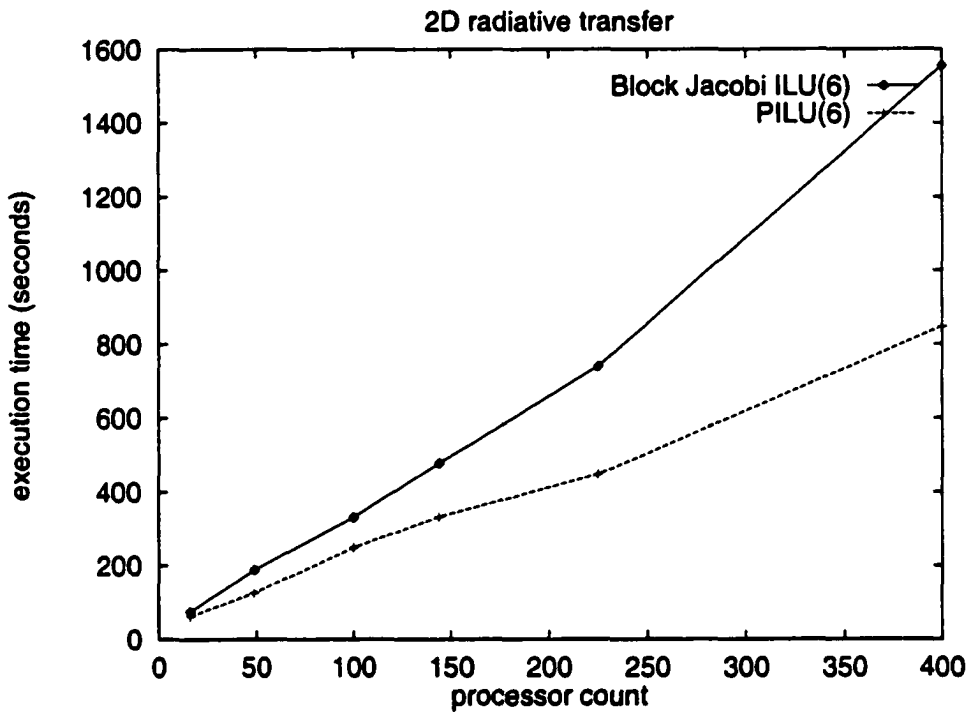


FIG. 38. *Simplified 2D Radiative Transfer problem (ASCI Blue). Problem size is 10K unknowns per processor. Results obtained from code distributed with PETSc. Solution for all problem sizes required eight or nine newton iterations, and a linear system was solved each iteration. Preconditioning used level four PILU or Block Jacobi ILU.*

7.2.2 THREE-BOX PROBLEM

Tables 23 and 24 show results for the three-box problem that was described in chapter 6, Figure 33. Convergence criteria was 1e-5 residual reduction of the preconditioned system. Each processor was assigned a 100 x 100 subgrid, for a total of 10,000 unknowns. The number of processors was scaled from 9 (90K global unknowns) to 225 (2.3M global unknowns). The systems were solved with PILU and Block Jacobi preconditioning, levels three through seven.

Level three or four preconditioners were most effective for the smaller numbers of processors. For the larger runs the level six or seven preconditioners gave the fastest execution time. PILU preconditioning appears to derive greater benefit from the higher levels for large numbers of processors. For example, for the 225 processor runs a level six preconditioner gave the fastest solution time (78.55 seconds), compared to a level seven preconditioner for PILU (63.75 seconds).

Block Jacobi ILU(ℓ) failed to converge after 2000 iterations for level three preconditioning for the larger runs (64 or more processors), while PILU achieved convergence in all cases.

7.2.3 OPTIMALITY

The condition of elliptically dominated problems scales like the inverse of the mesh spacing squared. For 2D problems on quasi uniform grids with N unknowns, this is $O(N)$; for 3D problems with N unknowns this is $O(N^{2/3})$. Incomplete factorization preconditioners generally improve the constants in these condition number scaling laws, but not the exponents. In contrast, multilevel methods such as geometric multigrid or algebraic multigrid can improve these condition numbers to $O(1)$. Therefore, ILU and its parallel forms cannot be expected to compete favorably with optimal multilevel preconditioners on arbitrarily large elliptical problems. However, PILU, being very general purpose, may be employed (recursively) as a "smoother" component in difficult multilevel problems, where simple multigrid smoothers like point Jacobi may be too weak. In this sense, an efficient parallel implementation of incomplete fatorization may be considered part of an optimal algorithm.

TABLE 21

3D convection-diffusion problem, PILU (ASCI Blue).

			Timing (seconds)				global
np	level	its	setup	solve	total	perIt	unknowns
	0	219	4.94	56.74	61.68	0.2591	512K
8	1	99	8.58	32.79	41.37	0.3312	
	2	89	19.08	37.88	56.96	0.4256	
	0	351	8.80	97.62	106.42	0.2781	1.mM
27	1	177	14.53	62.92	77.45	0.3555	
	2	149	31.25	67.61	98.87	0.4538	
	0	481	8.61	144.88	153.49	0.3012	4M
64	1	249	15.26	94.15	109.41	0.3781	
	2	219	33.90	106.65	140.55	0.4870	
	0	613	8.57	203.99	212.56	0.3328	8M
125	1	327	15.27	135.58	150.85	0.4146	
	2	281	33.63	145.38	179.01	0.5174	
	0	769	9.28	303.41	312.69	0.3946	14M
216	1	417	16.27	187.35	203.62	0.4493	
	2	351	34.92	195.19	230.11	0.5561	
	0	907	9.60	379.41	389.01	0.4183	2.2M
343	1	523	16.10	263.44	279.54	0.5037	
	2	445	35.27	269.84	305.12	0.6064	

TABLE 22

3D convection-diffusion problem, Block Jacobi ILU(ℓ)(ASCI Blue).

			Timing (seconds)				global
np	level	its	setup	solve	total	perIt	unknowns
8	0	275	1.42	57.05	58.48	0.2075	512K
	1	161	1.88	40.69	42.57	0.2527	
	2	163	2.78	51.78	54.56	0.3176	
27	0	411	1.39	89.17	90.55	0.2170	1.7M
	1	267	1.88	71.92	73.80	0.2694	
	2	253	2.75	86.01	88.76	0.3400	
64	0	617	1.31	144.92	146.23	0.2349	4M
	1	369	1.79	106.07	107.86	0.2875	
	2	351	2.64	123.88	126.53	0.3529	
125	0	751	1.33	206.21	207.54	0.2746	8M
	1	505	1.72	160.01	161.73	0.3169	
	2	465	2.62	177.80	180.42	0.3824	
216	0	913	1.32	268.52	269.84	0.2941	14M
	1	641	1.72	223.15	224.87	0.3481	
	2	635	2.51	268.94	271.44	0.4235	
343	0	1135	1.32	378.81	380.13	0.3338	22M
	1	783	1.67	309.48	311.14	0.3952	
	2	749	2.51	391.45	393.96	0.5226	

TABLE 23
2D three-box problem, PILU (ASCI Blue).

np	level	its	Timing (seconds) setup	solve	total	global unknowns
9	3	497	1.05	17.91	18.96	90K
	4	165	1.50	6.64	8.15	
	5	126	2.09	5.55	7.64	
	6	113	2.86	5.45	8.31	
	7	105	3.89	5.48	9.37	
36	3	544	1.15	24.37	25.52	360K
	4	332	1.83	18.18	20.01	
	5	257	2.30	13.37	15.67	
	6	216	3.28	13.11	16.39	
	7	191	4.70	12.74	17.44	
64	3	778	1.32	40.45	41.77	640K
	4	460	2.67	26.80	29.47	
	5	325	2.34	21.02	23.35	
	6	275	3.66	19.55	23.21	
	7	244	4.40	18.45	22.85	
100	3	1035	1.20	71.63	72.83	1M
	4	591	1.69	50.14	51.84	
	5	418	2.37	35.29	37.66	
	6	349	3.51	31.01	34.51	
	7	313	4.43	29.93	34.36	
144	3	1346	1.36	103.84	105.19	1.4M
	4	639	1.84	54.74	56.57	
	5	473	2.54	48.14	50.68	
	6	392	3.38	39.35	42.74	
	7	341	4.54	35.90	40.43	
225	3	1699	1.27	158.40	159.66	2.25M
	4	842	1.79	80.57	82.36	
	5	617	2.63	68.95	71.59	
	6	539	3.52	64.68	68.20	
	7	473	4.47	59.28	63.75	

TABLE 24

2D three-box problem, Block Jacobi ILU(ℓ)(ASCI Blue).

np	level	its	Timing (seconds) setup	solve	total	global unknowns
9	3	430	0.27	14.33	14.60	90K
	4	251	0.34	9.45	9.79	
	5	387	0.37	15.68	16.05	
	6	354	0.42	15.42	15.84	
	7	368	0.46	17.40	17.85	
36	3	1591	0.26	73.71	73.97	360K
	4	474	0.34	21.96	22.30	
	5	504	0.37	26.28	26.65	
	6	464	0.42	24.08	24.50	
	7	368	0.46	23.06	23.52	
64	3	2001	0.26	95.95	96.21	640K
	4	683	0.33	37.85	38.18	
	5	583	0.37	33.62	33.99	
	6	656	0.41	42.89	43.30	
	7	490	0.45	30.57	31.02	
100	3	2001	0.25	111.96	112.21	1M
	4	824	0.32	48.37	48.70	
	5	636	0.36	39.17	39.52	
	6	582	0.40	42.07	42.47	
	7	558	0.44	49.31	49.75	
144	3	2001	0.25	138.56	138.81	1.4M
	4	897	0.33	63.18	63.50	
	5	733	0.35	51.07	51.42	
	6	669	0.39	53.76	54.15	
	7	641	0.43	50.21	50.64	
225	3	2001	0.25	147.82	148.07	2.25M
	4	1267	0.31	98.27	98.58	
	5	1001	0.34	84.12	84.47	
	6	915	0.38	78.18	78.55	
	7	875	0.42	87.88	88.30	

CHAPTER 8

CONCLUSION AND FUTURE WORK

This thesis has presented new algorithms for computing ILU(ℓ) structure, predicting ILU(ℓ) storage requirements, and parallelizing preconditioner factorization and application for all ILU variants. The algorithms spring from a collection of lemmas and theorems, developed in Chapter 2, that elucidate the structure of ILU(ℓ) factors. Although we were not directly concerned with matrix numerical properties, the experimental results presented indicate that the parallel algorithm can be highly effective in practice.

The PILU algorithm is primarily intended for the preconditioning of scaled problems on large numbers of processors. Results presented indicate it should be far less effective from that standpoint of speedup, i.e., "solving a given problem size faster." The reason for this is that, as the number of processors increase, the boundary node ratio within each subdomain decreases, and we have shown that this degrades parallel performance. If one is only interested in obtaining speedup using a relatively small number of processors, however, a shared-memory implementation executed on an SMP node might give good results.

Our reported results are for a PILU implementation that uses MPI message passing for inter-processor communication. A shared-memory implementation in many respects would require far less software design effort. In such an implementation, for example, there could be a single global permutation vector, and the need for the explicit exchange of boundary node permutation information would be eliminated. Additionally, one could return to the sequential practice, discussed in Chapter 5, of performing the factorization by unpacking the row being factored in a working vector of length n. However, a shared memory implementation is unlikely to scale well to hundreds or thousands of processors.

It is possible that the concept of graph search algorithms could be extended to include numeric factorization. The goal would be to reduce factorization time by performing only those numerical updates that correspond to shortest path lengths. This would probably result in less effective preconditioners, in the sense that required iterations would increase, but this might be offset by a reduction in factorization time. However, we do not at present know how to design such an algorithm.

Although we have shown how to compute ILU(ℓ) storage requirements in $O(n)$ space, the method has the same run time complexity as actually performing the factorization. This stands in stark contrast to the e-tree storage prediction methods for complete factors of symmetric matrices, which operate in essentially $O(n)$ time as well as $O(n)$ space. It is an open question whether a similar result can be devised for incomplete factors.

The model PILU implementation has proven effective in practice; however, many improvements can be made. As discussed in Chapter 5, the factorization (setup) phase has known inefficiencies in communication and look-up of external boundary rows. Additionally, support should be added for complex numbers (researchers from two different national labs have expressed interest in complex PILU). Some PDE systems, such as the Navier Stokes equations, have multiple degrees of freedom associated with each gridpoint. This results in a "small" block structure that could be exploited to reduce both communication and computation time. Finally, the model implementation supports only ILU(ℓ) symbolic and numeric factorization. Support should be added for ILUT and other ILU variants. Fortunately, due to Euclid's object-oriented design, incorporating these additional factorization methods requires no alteration to Euclid's communication classes.

REFERENCES

[1] O. AXELSSON, *Iterative Solution Methods*, Cambridge University Press, Cambridge, UK, 1994.

[2] S. BALAY, W. D. GROPP, L. CURFMAN MCINNES, AND B. F. SMITH, *PETSc home page.* `http://www.mcs.anl.gov/petsc`, 2001.

[3] R. BARRETT, M. BERRY, T. F. CHAN, J. DEMMEL, J. DONATO, J. DONGARRA, V. EIJKHOUT, R. POZO, C. ROMINE, AND H. VAN DER VORST, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA, 1994.

[4] P. BASTIAN AND G. HORTON, *Parallelization of robust multigrid methods: ILU factorization and frequency decomposition method*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 1457–1470.

[5] M. W. BENSON, *Iterative solution of large scale linear systems*, tech. report, Lakehead University, Thunder Bay, ON, 1972.

[6] M. BENZI, W. JOUBERT, AND G. MATEESCU, *Numerical experiments with parallel orderings for ILU preconditioners*, Electronic Transactions on Numerical Analysis, 8 (1999), pp. 88–114.

[7] M. BENZI, J. MARIN, AND M. TUMA, *A two-level parallel preconditioner based on sparse approximate inverses*, in Iterative Methods in Scientific Computation IV, D. R. Kincaid and A. C. Elster, eds., vol. 5 of IMACS Series in Computational and Applied Mathematics, IMACS, 1999, pp. 167–178.

[8] M. BENZI AND M. TUMA, *A comparative study of sparse approximate inverse preconditioners*, Applied Numerical Mathematics, 30 (1999), pp. 305–340.

[9] ——, *Orderings for factorized sparse approximate inverse preconditioners*, SIAM J. Sci. Comput., 21 (2000), pp. 1851–1868.

[10] M. BERN, J. R. GILBERT, B. HENDRICKSON, N. NGUYEN, AND S. TOLEDO, *Support-graph preconditioners.* Submitted to SIAM J. Mat. Anal. and App., 29 pages, January 2001.

[11] E. G. BOMAN, D. CHEN, B. HENDRICKSON, AND S. TOLEDO, *Maximum-weight-basis preconditioners.* Submitted to the Journal on Numerical Linear Algebra, 29 pages, June 2001.

[12] A. M. BRUASET AND H. P. LANGTANGEN, *Object-oriented design of preconditioned iterative methods in Diffpack*, ACM Transactions on Mathematical Software, (1997), pp. 50–80.

[13] N. I. BULEEV, *A numerical method for the solution of two-dimensional and three-dimensional equations of diffusion*, Math. Sb., 51 (1960), pp. 227–238.

[14] L. CARROLL, *Through the Looking-Glass*, Project Gutenberg, Urbana, Illinois, 1991.

[15] T. CHAN AND H. VAN DER VORST, *Approximate and incomplete factorizations*, in Parallel Numerical Algorithms, D. E. Keyes, A. H. Samed, and V. Venkatakrishnan, eds., vol. 4 of ICASE/LaRC Interdisciplinary Series in Science and Engineering, Kluwer Academic Press, 1997, pp. 167–202.

[16] D. CHEN AND S. TOLEDO, *Implementation and evaluation of Vaidya's preconditioners.* Submitted to Electronic Transactions on Numerical Analysis, 20 pages, August 2001.

[17] E. CHOW, *Parasails home page.* `http://www.llnl.gov/CASC/parasails`, 2000.

[18] E. CHOW AND M. A. HEROUX, *Block preconditioning toolkit.* `http://www-users.cs.umn.edu/~chow/bpkit.html`, 1997.

[19] ——, *An object-oriented framework for block preconditioning*, ACM Trans. Math. Softw., 24 (1998), pp. 159–183.

[20] E. CHOW AND Y. SAAD, *Experimental study of ILU preconditioners of indefinite matrices*, J. Comput. Appl. Math, 86 (1997), pp. 387–414.

[21] ——, *Parallel approximate inverse preconditioners*, in Eighth SIAM Conference on Parallel Processing for Scientific Computing, March 1997. Minneapolis, MN, (CD-ROM proceedings).

[22] ——, *Approximate inverse preconditioners via sparse-sparse iterations*, SIAM J. Sci. Comput., 19 (1998), pp. 995–1023.

[23] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to algorithms*, McGraw-Hill, San Francisco, CA, 1990.

[24] E. F. D'AZEVEDO, P. A. FORSYTH, AND W.-P. TANG, *Towards a cost-effective ILU preconditioner with high level fill*, BIT, 32 (1992), pp. 442–463.

[25] S. DOI AND A. LICHNEWSKY, *A graph-theory approach for analyzing the effects of ordering on ILU preconditioning*, Tech. Report 1452, Institut National de Recherche in Informatique et en Automatique, Rocquencourt, BP105-78153, Le Chesnay Cedex, France, 1991.

[26] S. DOI AND T. WASHIO, *Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations*, Parallel Comput., 25 (1995), pp. 1995–2014.

[27] J. J. DONGARRA, I. S. DUFF, D. C. SORENSEN, AND H. A. VAN DER VORST, *Numerical Linear Algebra for High Performance Computers*, SIAM, Philadelphia, PA, 1998.

[28] I. S. DUFF, A. M. ERISMAN, C. W. GEAR, AND J. K. REID, *Sparsity structure and Gaussian elimination*, SIGNUM Newsletter, 23 (1988), pp. 2–9.

[29] I. S. DUFF AND G. A. MEURANT, *The effect of ordering on preconditioned conjugate gradients*, BIT, 29 (1989), pp. 635–657.

[30] I. S. DUFF AND J. REID, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Transactions on Mathematical Software, 9 (1983), pp. 302–325.

[31] V. EIJKHOUT, *Analysis of parallel incomplete point factorizations*, Linear Algebra and its Applications, 154–156 (1991), pp. 723–740.

[32] S. C. EISENSTAT AND J. W. H. LIU, *Exploiting structural symmetry in unsymmetric sparse symbolic factorization*, SIAM J. Mat. Anal. and App., 13 (1992), pp. 202–211.

[33] P. FREDERICKSON, *Fast approximate inversion of large sparse linear systems*, Tech. Report Math. Report 7, Lakehead University, Thunder Bay, Canada, 1975.

[34] A. GEORGE, J. R. GILBERT, AND W. H. LIU, *Graph Theory and Sparse Matrix Computation*, Springer-Verlag, 1993.

[35] A. GEORGE AND J. W. H. LIU, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, 1981.

[36] J. R. GILBERT AND J. W. H. LIU, *Elimination structures for unsymmetric sparse LU factors*, SIAM J. Mat. Anal. and App., 14 (1993), pp. 334–352.

[37] G. GOLUB AND J. M. ORTEGA, *Scientific Computing*, Academic Press, San Diego, 1993.

[38] A. GREENBAUM, *Iterative Methods for Solving Linear Systems*, SIAM, Philadelphia, 1997.

[39] K. GREMBAN, G. MILLER, AND M. ZAGHA, *Performance evaluation of a parallel preconditioner*, in 9th International Parallel Processing Symposium, Santa Barbara, April 1995, pp. 65–69.

[40] K. D. GREMBAN, *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*, PhD thesis, Carnegie Mellon University, October 1996. Technical Report CMU-CS-96-123.

[41] J. L. GUSTAFSON, *Reevaluating Amdahl's law*, Communications of the ACM, 31 (1988), pp. 532–533.

[42] J. L. GUSTAFSON, G. R. MONTRY, AND R. E. BENNER, *Development of parallel methods for a 1024-processor hypercube*, SIAM J. Sci. Comput., 9 (1988), pp. 609–638.

[43] I. GUSTAFSSON, *A class of first-order factorization methods*, BIT, 18 (1978), pp. 142–156.

[44] W. HACKBUSCH, *Iterative Solution of Large Sparse Systems of Equations*, vol. 95 of Applied Mathematical Sciences, Springer-Verlag, New York, U.S.A, 1993.

[45] W. HACKBUSCH AND G. WITTUM, eds., *Incomplete Decomposition ILU: Algorithms, Theory, and Applications*, Notes Numer. Fluid Mech. 41, Vieweg, Braunschweig, Wiesbaden, 1993.

[46] L. HASKINS AND D. J. ROSE, *Toward a characterization of perfect elimination digraphs*, SIAM J. Comput., 2 (1973), pp. 217–224.

[47] M. R. HESTENES AND E. L. STIEFEL, *Methods of conjugate gradients for solving linear systems*, J. Res. Nat. Bur. Standards, 49 (1952), pp. 409–436.

[48] D. HYSOM AND A. POTHEN, *Symmetric reduction redux: Incomplete factors*, in Fifth Copper Mountain Conference on Iterative Methods, March 1998.

[49] ——, *Efficient parallel computation of ILU(k) preconditioners*, in SC99, ACM, November 1999. published on CDROM, ISBN #1-58113-091-0, ACM Order #415990, IEEE Computer Society Press Order # RS00197.

[50] ——, *Parallel incomplete factorization preconditioning*, in Ninth SIAM Conference on Parallel Processing for Scientific Computing, March 1999. SIAM, Philadelphia (CDROM).

[51] ——, *Efficient parallel computation of ILU(k) preconditioners*, Tech. Report 2000-23, ICASE, NASA Langley Research Center, Hampton, VA, 2000.

[52] ——, *Parallel ILU ordering and convergence relationships: Numerical experiments*, Tech. Report 2000-24, ICASE, NASA Langley Research Center, Hampton, VA, 2000.

[53] ——, *A scalable parallel algorithm for incomplete factor preconditioning*, SIAM J. Sci. Comput., 22 (2001), pp. 2194–2215.

[54] M. T. JONES AND P. E. PLASSMANN, *An improved incomplete Cholesky factorization*, ACM Trans. Math. Software, 21 (1995), pp. 5–17.

[55] G. KARYPIS AND V. KUMAR, *Parallel threshold-based ILU factorization*, Tech. Report 061, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 5455, 1998.

[56] V. KUMAR, A. GRAMA, A. GUPTA, AND G. KARYPIS, *Introduction to Parallel Computing*, Benjamin/Cummings Publishing Company, Inc., 1994.

[57] S. KUZNETSOV, G. LO, AND Y. SAAD, *Parallel solution of general sparse linear systems*, Tech. Report IMSI-97-98, University of Minnesota, Minneapolis, 1997.

[58] C.-J. LIN AND J. J. MORÉ, *Incomplete Cholesky factorizations with limited memory*, SIAM J. Sci. Comput., 21 (1999), pp. 24–45.

[59] J. W. H. LIU, *The role of elimination trees in sparse factorization*, SIAM J. Mat. Anal. and App., 11 (1990), pp. 134–172.

[60] ——, *The multifrontal method for sparse matrix solution: theory and practice*, SIAM Review, 34 (1992), pp. 82–109.

[61] S. MA AND Y. SAAD, *Distributed ILU(0) and SOR preconditioners for unstructured sparse linear systems*, Tech. Report AHPCRC-94-027, Army High Performance Computing Research Center, University of Minnesota, Minneapolis, MN, 1994.

[62] M. MAGOLU MONGA MADE AND H. A. VAN DER VORST, *Parallel incomplete factorizations with pseudo-overlapped subdomains*, Parallel Computing, 27(8) (2001), pp. 989–1008.

[63] ——, *Spectral analysis of parallel incomplete factorizations with implicit pseudo-overlap*, Numer. Linear Algebra App., to appear, 9 (2002).

[64] T. A. MANTEUFFEL, *An incomplete factorization technique for positive definite linear systems*, Math. Comput., 34 (1980), pp. 307–327.

[65] J. MEIJERINK AND H. VAN DER VORST, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.

[66] ——, *Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems*, J. Comput. Phys., 44 (1981), pp. 134–155.

[67] B. M. MURRAY, *C++ Strategies and Tactics*, Addison-Wesley, 1993.

[68] T. A. OLIPHANT, *An implicit numerical method for solving two-dimensional time-dependent diffusion problems*, Quart. Appl. Math., 19 (1961), pp. 221–229.

[69] ——, *An extrapolation process for solving linear systems*, Quart. Appl. Math., 20 (1962), pp. 257–267.

[70] S. PARTER, *The use of linear graphs in Gauss elimination*, SIAM Review, 3 (1961), pp. 119–130.

[71] D. J. ROSE AND R. E. TARJAN, *Algorithmic aspects of vertex elimination on directed graphs*, Tech. Report STAN-CS-75-531, Computer Science Department, Stanford University, 1975.

[72] ——, *Algorithmic aspects of vertex elimination on directed graphs*, SIAM J. Appl. Math., 23 (1978), pp. 176–197.

[73] D. J. ROSE, R. E. TARJAN, AND G. S. LUEKER, *Algorithmic aspects of vertex elimination on directed graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

[74] Y. SAAD, *SPARSKIT: a basic tool kit for sparse matrix computations*, tech. report, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990. `http://www-users.cs.umn.edu/~saad/software.html`.

[75] ——, *Highly parallel preconditioners for general sparse matrices*, Tech. Report AHPCRC Preprint 92-087, University of Minnesota, Minneapolis, 1992.

[76] ——, *ILUT: A dual-threshold incomplete LU factorization*, Numer. Linear Algebra App., (1994), pp. 387–402.

[77] Y. SAAD, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 20 Park Plaza, Boston, MA 02116, 1996.

[78] Y. SAAD AND M. SOSONKINA, *Enhanced parallel multicolor preconditioning techniques for linear systems*, in SIAM conf. Parallel Processing for Scientific Computing, November 1999.

[79] B. F. SMITH, P. BJORSTAD, AND W. GROPP, *Domain Decomposition*, Cambridge University Press, 1996.

[80] P. M. VAIDYA, *Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners (unpublished manuscript).* A talk based on the manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computation, Octoboer 1991, Minneapolis.

[81] H. A. VAN DER VORST, *High performance preconditioning*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1174–1185.

[82] ——, *Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems*, SIAM J. Sci. Stat. Comput., 13 (1992), pp. 631–634.

[83] R. S. VARGA, *Factorization and normalized iterative methods*, in Boundary Problems in Differential Equations, R. E. Langer, ed., University of Wisconsin Press, Madison, WI, 1960, pp. 121–142.

[84] C. VUIK, R. R. P. VAN NOOYEN, AND P. WESSELING, *Parallelism in ILU-preconditioned GMRES*, Parallel Comput., (1998), pp. 1927–1946.

APPENDIX A

PREVIOUSLY PUBLISHED MATERIAL

The bulk of the material in Chapter 4 has been published in the SIAM Journal of Scientific Computing [53]. Permission to use this material has been granted by SIAM, per the following email, dated November 27, 2001.

> SIAM is pleased to grant you permission to include the article "A scalable parallel algorithm for incomplete factor preconditioning," by David Hysom and Alex Pothen, SIAM Journal on Scientific Computing 22 (2001), pp. 2194-2215, in your PhD thesis.
>
> Please indicate that portions of your thesis were published in this article and include the complete citation.
>
> The SIAM copyright transfer agreement you, or your coauthor on behalf of you, signed gives you the specific right to use the work in any other work also authored by you. This includes your thesis.
>
> Please let me know if you have any questions or if you need a hardcopy letter.
>
> Sincerely,
>
> Mary Rose Muccie
> Journals Publisher
> **muccie@siam.org**

APPENDIX B

EXPERIMENTAL PLATFORMS

Euclid, the model implementation of the PILU Algorithm, has been tested on a variety of platforms; this appendix summarizes the hardware characteristics of these various machines. Readers should note, however, that Euclid's development, and the experiments reported in this dissertation, spanned a period of over 2 1/2 years. As practitioners are aware, most large scale parallel platforms are in a continual state of flux. Both software (operating system, compilers, MPI implementations) and hardware are continually being modified, sometimes on a weekly basis. Therefore, the reported machine configurations may not reflect precisely the configuration in place when experiments were conducted. However, the Euclid code has, for several months, performed reliably and consistently on all platforms tested, and it is this author's belief that the statistics presented below, coupled with the experimental evidence reported in previous chapters of this dissertation, provide an accurate reflective of what end users can reasonably expect in terms of performance.

B.1 SGI ORIGIN2000

The SGI Origin2000 is located at NASA Ames Research Center (AMES). The cluster employs SGI's cc-NUMA architecture. The compute cluster *Steger* has 128 nodes, 256 CPUs and a total of 64GB system memory. Each processors is a MIPS RISC R10000 64-bit CPU, with a 32KB two-way set-associative instruction cache, a 32KB two-way set-associative data cache, and 4MB L2 cache. System bus bandwidth as measured by bisection bandwidth is reported at 40GB/sec sustained, 50GB/sec peak.

B.2 CORAL PC BEOWULF CLUSTER

The Coral PC Beowulf cluster, located at ICASE, NASA Langley Research Center, Hampton, Virginia, is a heterogeneous cluster consisting of 32 single CPU and 32 dual CPU compute nodes, built in four phases. The complete system has 54.5 GB of RAM and 1.2 TB of raw disk space. The communication network consists of two fast ethernet switches trunked together via dual Gigabit Ethernet, with a

Gigabit Ethernet uplink to a Gigabit Ethernet switch connecting the servers and one of the dual CPU 800 MHz Pentium III compute nodes. Coral's 32 dual CPU compute nodes are also connected via a high performance cLAN network in a full bandwidth configuration, delivering up to 113 Mbyte/s with MVICH latency under 14 microseconds. This cluster runs Red Hat Linux 7.1 updated to Linux kernel 2.4.3-12 and tuned for high network performance.

Phase I started with the 32 single CPU nodes and the front-end server. Each single CPU node has a 400 MHz Pentium II processor, 384 MB of 100 MHz RAM, a fast 6.5 GB local disk, a floppy drive and a fast ethernet card.

Phase II added 16 dual CPU nodes, Gigabit Ethernet network and two file servers. Each Phase II dual CPU node has two 500MHz Pentium III processors, 512 MB of 100 MHz RAM, a fast 14.4 GB local disk, a floppy drive and a fast ethernet card.

Phase III added another 16 dual CPU nodes and the 32-node cLAN network in a full bandwidth configuration. Each Phase III dual CPU node has two 800MHz Pentium III 'Coppermine' processors, 1 GB of 133 MHz RAM, a cLAN adaptor on a 64-bit PCI bus, a fast 30 GB local disk, a floppy drive and dual fast ethernet. The cLAN adaptors were also added to Phase II dual CPU nodes, so that a total of 64 CPUs are connected to this high performance network fabric.

Phase IV replaced 24 of the old Phase I nodes with new 1.7 GHz Pentium 4 nodes with 400 MHz memory (PC800 RDRAM). The new machines also have more memory (two have 2 GB each and the rest have 1 GB each). Initial uniprocessor tests confirm that these nodes are about four times faster than the Phase I nodes they replaced.

Experiments reported in this dissertation used the capabilities of phases I through III, and the MPICH library and fast ethernet for communication.

B.3 SUN HPC 10000 STARFIRE

The Sun HPC 10000 Starfire server is located at Old Dominion University, Norfolk, Virginia. The server contains 32 superscalar 64-bit RISC CMOS UltraSPARCTM-II processors. This shared-memory NUMA machine contains a total of 32 gigabytes of system memory. Each processor contains a 16KB direct mapped non-blocking L1 data cache, a 16KB direct mapped non-blocking L1 instruction cache, and a 4MB

L2 direct mapped cache. Each CPU can execute 4 instructions/clock cycle (9-stage pipeline), and has 4 integer, 2 floating-point, and 2 graphics execution units. CPUs are rated at up to to 1.3 Gbytes/sec memory transfers.

The system has a Gigaplane-XB Interconnect that adheres to Sun Microsystems' UltraSPARCTM Port Architecture (UPA) specification, which defines separate address and data paths. The Gigaplane-XB–a 16-byte wide, 16x16 non-blocking, true crossbar connects the system boards via one global data router and four global address buses. Separating the buses allows the data and address topologies to be independently optimized for their respective purposes. In contrast, the four global address buses allow four simultaneous address transfers or broadcasts throughout the system.

B.4 ASCI BLUE PACIFIC

The ASCI Blue Pacific IBM-SP cluster is located at Lawrence Livermore National Laboratory, Livermore, California. The machine is composed of 332 Mhz 604e 4-way SMP compute nodes. Each shared-memory node contains four CPUs, a single Network Interface, and a total of 1.5 GB main memory. Each CPU has one floating-point unit and one load/store unit; a 32KB L1 4 way associative cache with 32 byte cache lines and an LRU replacement scheme; and a 500KB L2 cache. Tests used IBM's MPI library in user-space mode. Node to node bi-directional bandwidth is 150 Mbyte/s.

VITA

David A. Hysom
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

David Hysom received a B.S. in Sociology from The University of the State of New York, 1991, and an M.S. in Computer Science from Old Dominion University in 1997. Much of his PhD academic career was supported by a GAANN fellowship from the Department of Education. He has published several papers and technical reports on the Parallel ILU algorithm, and has presented his work at several international conferences. One of his papers was a runner-up for the best paper at the Supercomputing '99 Conference. David currently holds a post-doc position in the Center for Applied Scientific Computing (CASC), at Lawrence Livermore National Laboratory (LLNL), where he is working on the SAMRAI and EMSolve projects.

Typeset using LaTeX.