RC 25077, November 10, 2010
Computer Science/Mathematics

IBM Research Report

Parallel Preconditioners for Sparse Iterative Methods: A short tutorial

Anshul Gupta

Business Analytics and Mathematical Sciences
IBM T. J. Watson Research Center
anshul@watson.ibm.com

Parallel Preconditioners for Sparse Iterative Methods: A short tutorial

Anshul Gupta
Business Analytics and Mathematical Sciences
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
anshul@watson.ibm.com

1 Definition

Iterative methods for solving sparse systems of linear equations are potentially less memory and computation intensive than direct methods, but often experience slow convergence or fail to converge at all. The robustness and the speed of Krylov subspace iterative methods is improved, often dramatically, by *preconditioning*. Preconditioning is a technique for transforming the original system of equations into one with an improved distribution (clustering) of eigenvalues so that the transformed system can be solved in fewer iterations. A key step in preconditioning a linear system $Ax = b$ is to find a nonsingular *preconditioner* matrix M such that the inverse of M is as close to the inverse of A as possible and solving a system of the form $Mz = r$ is significantly less expensive than solving $Ax = b$. The system is then solved by solving $(M^{-1}A)x = M^{-1}b$. This particular example shows what is known as *left preconditioning*. There are two other formulations, known as *right preconditioning* and *split preconditioning*; the basic concept, however, is the same. Other practical requirements for successful preconditioning are that the cost of computing M itself must be low and the memory required to compute and apply M must be significantly less than that for solving $Ax = b$ via direct factorization.

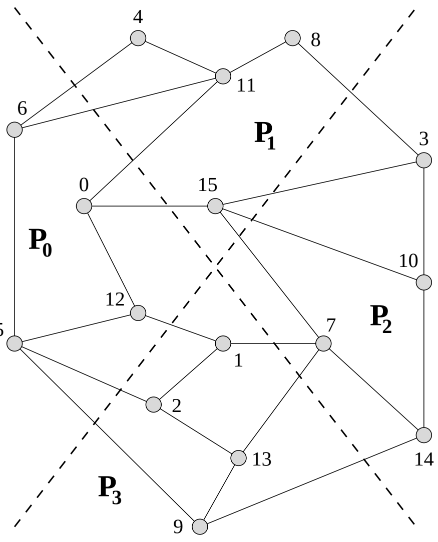
2 Discussion

Preconditioning methods are being actively researched and have been for a number of years. There are several classes of preconditioners; some are more amenable to being computed and applied in parallel than others. This article gives an overview of the generation (i.e., computing M in parallel) and application (i.e., solving a system of the form $Mz = r$ in parallel) of the most commonly used parallel preconditioners. Note that the topic of parallel preconditioning can easily fill a whole volume. This short article just scratches the surface of this rich research area and introduces the basic parallelization techniques.

While solving a sparse linear system $Ax = b$ in parallel, the matrix A and the vectors x and b are typically partitioned. The partitions are assigned to tasks that are executed by individual processes or threads in a parallel processing environment. Both the creation and the application of a preconditioner in parallel is affected by the underlying partitioning of the data. A commonly used effective and natural way of partitioning the data involves partitioning the graph, of which the coefficient matrix is an adjacency matrix. Other than partioning the problem for parallelization, the graph view of the matrix plays a useful role in many aspects of solving

$$
\begin{array}{c|cccccccccccccccc|c}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \\
\hline
0 & X & & & & & & & & & & & X & X & & & X & P_0 \\
1 & & X & X & & & & & X & & & & & X & & & & P_3 \\
2 & & X & X & & & X & & & & & & & & X & & & P_3 \\
3 & & & & X & & & & & X & & X & & & & & X & P_2 \\
4 & & & & & X & & X & & & & & X & & & & & P_1 \\
5 & & & X & & & X & X & & & X & & & X & & & & P_0 \\
6 & & & & & X & X & X & & & & & X & & & & & P_0 \\
7 & & X & & & & & & X & & & & & & X & X & X & P_2 \\
8 & & & & X & & & & & X & & & X & & & & & P_1 \\
9 & & & & & & X & & & & X & & & & X & X & & P_3 \\
10 & & & & X & & & & & & & X & & & & X & X & P_2 \\
11 & X & & & & X & & X & & X & & & X & & & & & P_1 \\
12 & X & X & & & & X & & & & & & & X & & & & P_0 \\
13 & & & X & & & & & X & & X & & & & X & & & P_3 \\
14 & & & & & & & & X & & X & X & & & & X & & P_2 \\
15 & X & & & X & & & & X & & & X & & & & & X & P_1 \\
\hline
\end{array}
$$

(a) A 16 x 16 symmetric sparse matrix



(b) The associated graph and its four partitions

Figure 1: A 16×16 sparse matrix with symmetric structure and its associated graph partitioned among four tasks.

sparse linear systems. Figure 1 illustrates a partitioning of the rows of a matrix among four tasks based on a 4-way partitioning of its graph.

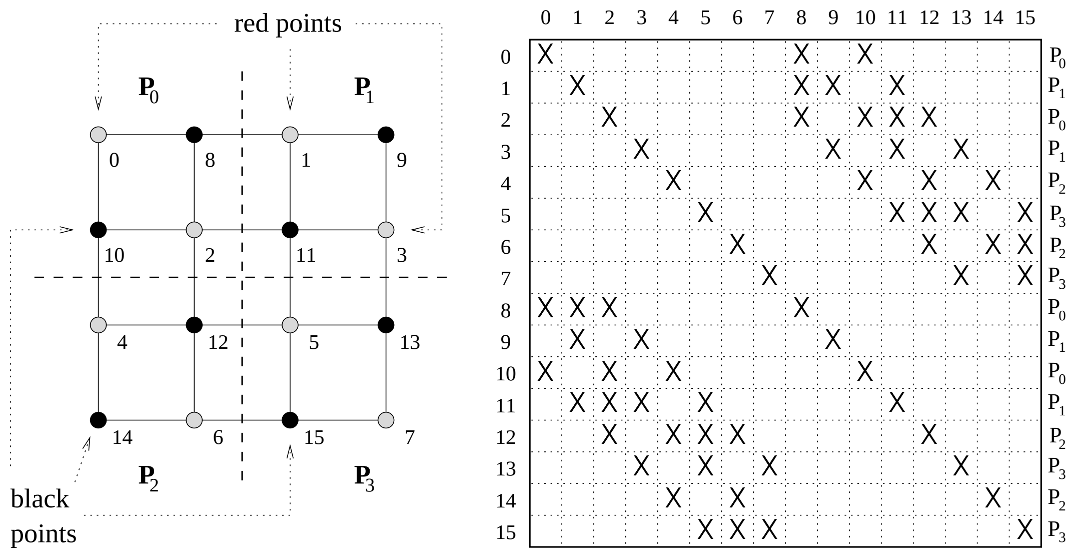
2.1 Simple preconditioners based on stationary methods

Stationary iterative methods are relatively simple algorithms that start with an initial guess of the solution (like all iterative methods) and attempt to converge towards the actual solution by repeated application of a correction equation. The correction equation uses the current residual and a fixed (stationary) operator or matrix, which is an approximation of the original coefficient matrix. While stationary iterative methods themselves have poor convergence properties, the approximating matrix can serve as a preconditioner for Krylov subspace methods.

2.1.1 Jacobi and block-Jacobi preconditioners

One of the simplest preconditioners is the point-Jacobi preconditioner, which is nothing but the diagonal D of the matrix A of coefficients. Applying the preconditioner in parallel is straightforward. It simply involves division with the entries (or multiplication with their inverses) of the part of the diagonal corresponding to the portion of the matrix that each thread or process is responsible for. In fact, scaling the coefficient matrix by the diagonal so that the scaled matrix has all 1's on the diagonal is equivalent to Jacobi preconditioning.

A block-Jacobi preconditioner is made up of non-overlapping square diagonal blocks of the coefficient matrix. These block may be of the same or different sizes. These blocks are usually factored or inverted (independently, in parallel) during the preconditioner construction phase so that the preconditioner can be applied inexpensively during the Krylov solver's iterations.



(a) A 4 X 4 grid with red−black ordering (b) Corresponding coefficient matrix

Figure 2: The sparse matrix corresponding to a 4×4 finite-difference grid with red-black ordering, partitioned among four parallel tasks.

2.1.2 Gauss-Seidel preconditioner

Let the coefficient matrix A be represented by a three-way splitting as $L + D + U$, where L is the strictly lower triangular part of A, D is a diagonal matrix that consists of the principal diagonal of A, and U is the strictly upper triangular part of A. The Gauss-Seidel preconditioner is defined by

$$M = (D + L)D^{-1}(D + U). \tag{1}$$

A system $Mz = r$ is then trivially solved as $y = (D + L)^{-1}r$, $w = Dy$, and $z = (D + U)^{-1}w$. Thus, applying the Gauss-Seidel preconditioner in parallel involves solving a lower and an upper triangular system in parallel. In general, equation i of a lower triangular system can be solved for the i-th unknown when equations $1 \ldots i - 1$ have been solved. Similarly, equation i of an $N \times N$ upper triangular system can be solved after equations $i + 1 \ldots N$ have been solved. Since the matrices L and U in our case are sparse, the i-th equation while computing $y = (D + L)^{-1}r$ depends only on those unknowns that have nonzero coefficients in the i-th row of L. Similarly, the i-th equation while computing $x = (D + U)^{-1}w$ depends only on those unknowns that have nonzero coefficients in the i-th row of U. As a result, while solving both these systems, multiple unknowns may be computed simultaneously—those that do not depend on any unsolved unknowns. This is an obvious source of parallelism. This parallelism can be maximized by reordering the rows and columns of A (and hence those of L and U) in a way that maximizes the number of independent equations at each stage of the solution process.

Figure 2 illustrates one such ordering, known as red-black ordering, that can be used to parallelize the application of the Gauss-Seidel preconditioner for a matrix arising from a finite-difference discretization. The vertices of the graph corresponding to the matrix are assigned colors such that no two neighboring vertices have the same color. All vertices and the corresponding rows and columns of the matrix are numbered first, followed

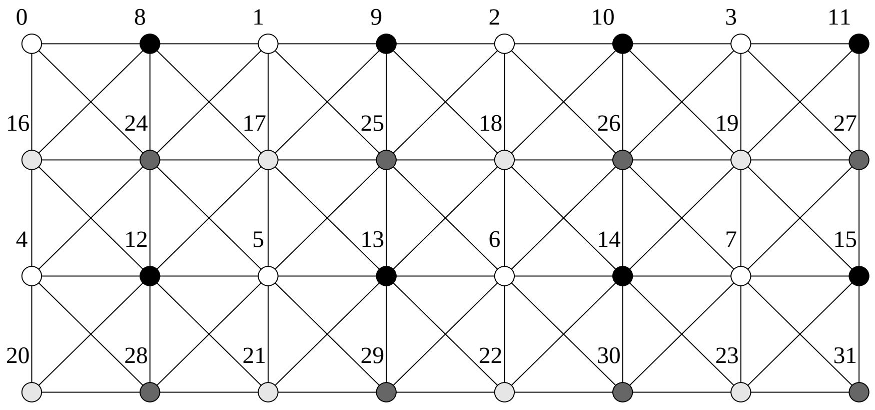


Figure 3: Multicolored ordering of a hypothetical finite-element graph using four colors.

by those of the other color. Assignment of matrix rows to tasks is based on a partitioning of the graph. With red-black ordering, each triangular solve is performed in two phases. During the lower triangular solve, first, all red unknowns are computed in parallel because they are all independent. After this step, all black unknowns can be computed. The order is reversed during the upper triangular solve. On a distributed-memory platform, each computation phase is followed by a communication phase. During a communication phase, each process communicates the values of the unknowns corresponding to graph vertices on the partition boundaries with its neighboring processes.

The idea of red-black ordering can be extended to general sparse matrices and their graphs, for which more than two colors may be required to ensure that no neighboring vertices have the same color. The triangular solves are then performed in as many parallel phases as the number of colors. A multicolored ordering with four colors is illustrated in Figure 3. For improved cache performance, block variants of multicolored ordering can be constructed by assigning colors to clusters of graph vertices and ensuring that no two clusters that have an edge connecting them are assigned the same color.

The reader is cautioned that often the convergence of a Krylov subspace method is sensitive to the ordering of matrix rows and columns. While red-black and multicolor orderings enhance parallelism, they may result in a deterioration of the convergence rate in some cases.

2.1.3 SOR preconditioner

A significant increase in convergence rate may be obtained by a modification of Equation 1 as follows, with $0 < \omega < 2$:

$$M = (\frac{D}{\omega} + L)\frac{\omega D^{-1}}{2-\omega}(\frac{D}{\omega} + U), \tag{2}$$

although determining an optimal value of ω can be expensive. The preconditioner specified by Equation 2 is known as successiver overrelaxtion or SOR preconditioner. Its symmetric formulation, when $L = U$, is known as symmetric SOR or SSOR preconditioner. The issues in parallel application of the SOR preconditioner are identical to those for the Gauss-Seidel preconditioner.

2.2 Preconditioners based on incomplete factorization

A class of highly effective but conceptually simple preconditioners is based on incomplete factorization methods. Recall that iterative methods are used in applications where a factorization of the form $A = LU$ is not feasible because the triangular factor matrices L and U are much denser than A, and therefore, too expensive to compute and store. The idea behind incomplete factorization is to perform a factorization of A along the lines of a regular Gaussian elimination or Cholesky factorization, but dropping a large fraction of nonzero entries from the triangular factors along the way. Depending on the underlying factorization method, incomplete factorization is referred to as ILU (incomplete LU) or IC (incomplete Cholesky) factorization. Due to dropping, the resulting triangular factors $\tilde{L}$ and $\tilde{U}$ are much sparser than L and U and are computed with significantly less computing effort. Entries are chosen for dropping using some criteria that strive to keep the inverse of $M = \tilde{L}\tilde{U}$ as close to the inverse of A as possible. Devising effective dropping criteria has been an active area of research. Incomplete factorization methods can be broadly classified as follows based on the the dropping criteria.

2.2.1 Static-pattern incomplete factorization

A static-pattern incomplete factorization method is one in which the locations of the entries that are kept in the factors and those that are dropped are determined a priori, based only on the structure of A. The simplest form of incomplete factorizations are ILU(0) and IC(0), in which the structure of $\tilde{L} + \tilde{U}$ is identical to structure of A; i.e., only those factor entries whose locations coincide with those of nonzero entries in the original matrix are saved. A generalization of static-pattern incomplete factorization is a level-κ incomplete factorization, referred to as ILU(κ) or IC(κ) factorization in the literature. The structure of a level-κ incomplete factorization is computed symbolically as follows. Initially, *level*$(i, j) = 0$ if $a_{ij} \neq 0$; otherwise, *level*$(i, j) = \infty$. This is followed by an emulation of factorization where each numerical update step of the form $a_{ij} = a_{ij} - a_{ik}.a_{kj}$ is replaced by updating *level*$(i, j) =$ *min*(*level*(i, j), *level*(i, k) + *level*(k, j) + 1). During the subsequent numerical factorization phases, entries in locations with a level greater than κ are dropped.

In graph terms, upon the completion of symbolic factorization, *level*(i, j) is 1 less than the length of the shortest path between vertices i and j. During the computation and application of an ILU(κ) or IC(κ) preconditioner, the computation corresponding to a vertex in the graph requires data associated with vertices that are up to $\kappa + 1$ edges away from it. For example, in the graph shown in Figure 4, data exchange is required among vertex pairs (p, q) and (q, r) for $\kappa = 0$. For $\kappa = 1$, additional exchanges among vertex pairs such as (s, t) and (u, v) are required. The figure also illustrates that in a parallel environment, data associated with $\kappa + 1$ layers of vertices adjacent to a partition boundary needs to be exchanged with a neighboring task.

Both the computation per vertex of the graph and the data exchange overhead per task in each iteration of a Krylov solver using a level-κ incomplete factorization preconditioner increase as κ increases. On the other hand, the overall number of iterations typically declines. The optimum value of κ is problem dependent.

2.2.2 Threshold-based incomplete factorization

Although it permits a relatively easy and fast parallel implementation, static-pattern incomplete factorization is robust for a few classes of problems only, including those with diagonally dominant coefficient matrices. It can, and often does delete fill entries of large magnitudes that do not happen to be located in the predetermined locations. The resulting large error can make the preconditioner ineffective. Threshold-based incomplete fac-

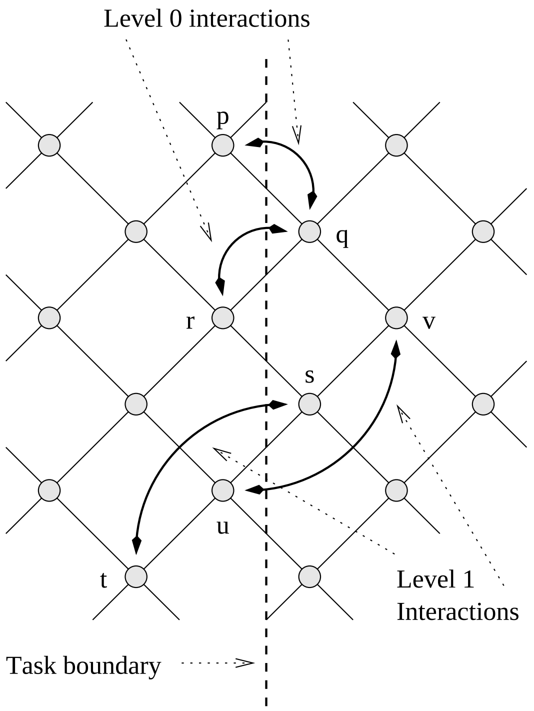


Figure 4: Illustration of data exchange across a task boundary when level 0 and level 1 fill is permitted in incomplete factorization.

torization rectifies this problem by dropping entries from the factors as they are computed. Regardless of their locations, entries greater in magnitude than a user-defined threshold τ are kept and the others are dropped. Typically, a second threshold γ is also used to limit the factors to a predetermined size. If n_i is the number of nonzeros in row (or column) i of the coefficient matrix, then at most γn_i entries (those with the largest magnitudes) are permitted in row (or column) i of the incomplete factor.

Successful and scalable parallel implementations of threshold-based incomplete factorization preconditioning use graph partitioning and graph coloring for balancing computation and minimizing communication among parallel tasks. The use of these two techniques in the context of sparse matrix computations has already been discussed earlier. Graph partitioning enables parallel tasks to independently compute and apply (i.e., perform forward and back substitution) the preconditioner for the matrix rows and columns corresponding to the internal vertices of the partition. An internal vertex and all its neighbors belong to the same partition. Graph coloring permits parallel incomplete factorization and forward and back substitution of matrix rows and columns corresponding to the boundary vertices. In the context of incomplete factorization, coloring is applied to a graph that includes only the boundary vertices (i.e., after the internal vertices have been eliminated) but includes the additional edges (fill-in) created as a result of the elimination of the internal vertices.

2.2.3 Incomplete factorization based on inverse-norm estimate

This is a relatively new class of incomplete factorization preconditioners in which the dropping criterion takes into account and seeks to minimize the growth of the norm of the inverse of the factors. These preconditioners have been shown to be more robust and effective than incomplete factorization with dropping based solely on the position or absolute value of the entries. The issues in the parallel generation and application of these preconditioners are very similar to those in threshold-based incomplete factorization—in both cases, the location of nonzeros in the factors cannot be determined a priori.

2.3 Sparse approximate inverse preconditioners

While the incomplete factorization preconditioners seek to compute $\tilde{L}$ and $\tilde{U}$ as approximations of the actual factors L and U of the coefficient matrix A, sparse approximate preconditioners seek to compute M^{-1} as an approximation to its inverse A^{-1}. The problem of computing M^{-1} is framed as the problem of minimizing the norm $\|I - AM\|$ or $\|I - MA\|$. To support parallelism, these minimization problems can be reduced to independent subproblems of computing the rows and columns of M^{-1}. Note that the actual inverse of a sparse A is dense, in general. It is therefore imperative that a number of entries be dropped in order to keep M^{-1} sparse. Just like incomplete factorization, the dropping can be structural (static), or based on values (dynamic), or both. In graph terms, while computing the rows or columns of M^{-1} in parallel, dropping is typically orchestrated in a way that confines the interaction to pairs of vertices that are either immediate neighbors or have short paths connecting them in the graph of A. Graph partitioning is used to facilitate load balance and minimize interaction among parallel tasks—both during the computation and the application of the preconditioner.

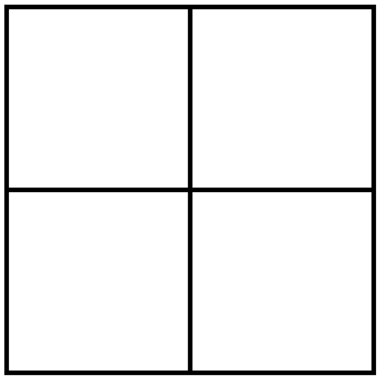
There are some important advantages to explicitly using an approximation of A^{-1} for preconditioning, rather than using A's approximate factors. First, the preconditioner computation avoids the kind of breakdowns that are possible in incomplete factorization due to small or zero (or negative, in case of incomplete Cholesky) diagonals. Secondly, the application of the preconditioners involves a straightforward multiplication of a vector with the sparse matrix M^{-1}, which may be simpler and more easily parallelizable than the forward and back substitutions with $\tilde{L}$ and $\tilde{U}$. However, just like $\tilde{L}$ and $\tilde{U}$, M^{-1} may be denser than A. Therefore, the graph of M^{-1} many have many more edges than the graph of A and multiplying a vector with M^{-1} may require more communication than multiplying a vector with A.

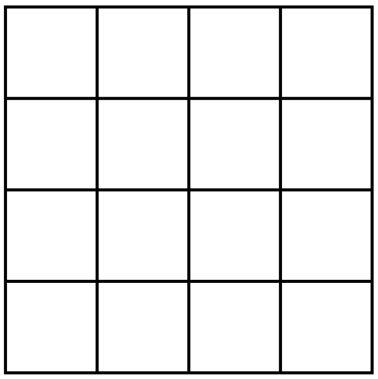
2.4 Multigrid preconditioners

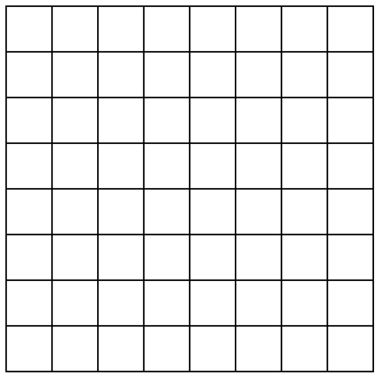
Multigrid methods are a class of iterative algorithms for solving partial differential equations (PDEs) efficiently, often by exploiting more problem-specific information than a typical Krylov subspace method. Each iteration of a multigrid solver is a somewhat complex recursive procedure. In many applications, the effectiveness of a multigrid algorithm can be substantially enhanced by using it to precondition a Krylov subspace method rather than using it as the solver. This is done by replacing the preconditioning step of the Krylov subspace solver with one iteration of the multigrid algorithm; i.e., treating the approximate solution obtained by an iteration of the multigrid algorithm as the solution with respect to a hypothetical preconditioner matrix.

2.4.1 Geometric multigrid

Multigrid methods were originally designed for solving elliptic PDEs by discretizing them using a hierarchy of regular grids of varying degrees of fineness over the same domain. For example, consider a domain D and a sequence of successively finer discretizations G_0, G_1, $\ldots$, G_m. Here G_0 is the coarsest discretization and G_m is the finest discretization over which the eventual solution to the PDE is desired. Figure 5 shows a square domain with $m = 3$. As the figure shows, the grid points in G_i are a subset of the grid points in G_{i+1}. A simple formulation of multigrid would work as follows:


(a) Discretization G_0


(b) Discretization G_1


(c) Discretization G_2

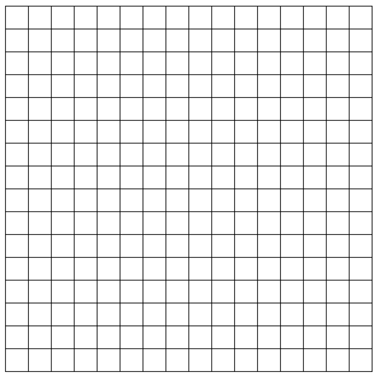

(d) Discretization G_3

Figure 5: Successively finer discretizations of a domain.

1. First, the linear system corresponding to discretization G_0 is solved. Since G_0 has a small number of points, the associated linear system is small and can be solved inexpensively by an appropriate direct or iterative method.

2. The solution at G_0 is interpolated to obtain an initial guess of the solution of the system corresponding to G_1, which is four times larger. Among various ways of *interpolation* (also knows as *prolongation*), a simple one involves approximating the value of the solution at a point that is in G_1 but not in G_0 by the average of the values of its neighbors.

3. Starting with the initial guess obtained by interpolation, a few steps of *relaxation* (also referred to as *smoothing*) are used to refine the solution at G_1. Often, the relaxation steps are simply a few iterations of a relatively inexpensive stationary method such as Jacobi or Gauss-Siedel.

4. The process of relaxation and interpolation continues from G_i to G_{i+1}, until $i + 1 = m$. After the relaxation at G_m, a first approximation x_0^m to the solution x^m of the the linear system $A^m x^m = b^m$ corresponding to G_m is obtained. Successively more accurate approximations $x_1^m, x_2^m, \ldots$ are obtained by $x_{j+1}^m = x_j^m + d_j^m$, where d_j^m is obtained by solving $Ad_j^m = r_j^m \equiv (b^m - Ax_j^m)$ via steps 5 and 6 below.

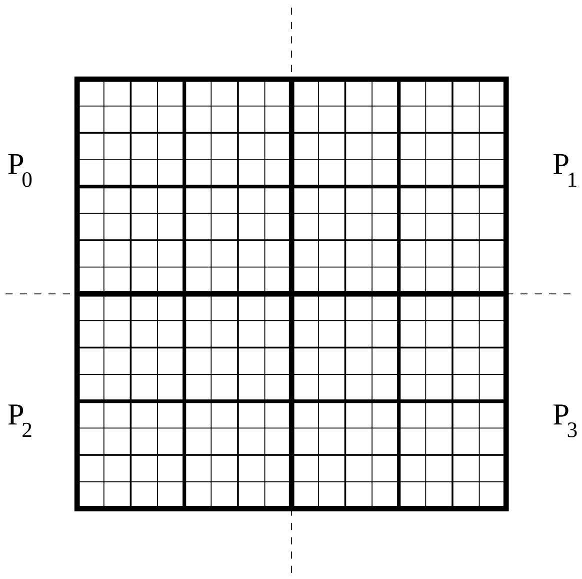


Figure 6: The domain and discretizations G_0–G_3 of Figure 5 mapped onto four parallel tasks.

5. The system $Ad_j^m = r_j^m$ in the j-th multigrid iteration is solved by a recursive process, in which the residual r_j^m corresponding to G_m is projected on to G_{m-1} and so on. The process of *projection* (also known as *restriction*) is the reverse of interpolation. At the end of the recursive projection steps, a relatively small system of equations $Ad_j^0 = r_j^0$ corresponding to G_0 is obtained, which is readily solved by an appropriate iterative or direct method.

6. The cycle of interpolation and relaxation is then repeated to obtain $d_j^1, d_j^2, \ldots, d_j^m$ starting with d_j^0 computed in step 5.

7. The process is stopped after k multigrid iterations if r_k^m is smaller than a user-defined threshold.

When the multigrid method is used as a preconditioner, then instead of repeating the multigrid cycle k times to solve the problem, the approximate solution after one cycle is substituted as the solution with respect to the preconditioner inside a Krylov subspace algorithm. Whether multigrid is used as a solver or a preconditioner, the parallelization process is the same.

The first step in implementing a parallel multigrid procedure is to partition the domain among the tasks such that each task is assigned roughly the same number of points of the finest grid. For example, Figure 6 shows the partitioning of the domain of Figure 5 and its discretizations G_0–G_3 into four parallel tasks. Unlike Figures 5 and 6, the domain in a real problem may be irregular and the partitioning may not be trivial. The partitioning of the domain implicitly defines a partitioning of the grids at all levels of discretization. It is easily seen that the parallel interpolation, relaxation, and projection at any grid level require a task to exchange information corresponding to the grid points along the partition boundary with its neighboring tasks. All computations corresponding to each partition's interior points can be performed independently.

2.4.2 Algebraic multigrid

The algebraic multigrid (AMG) method is a generalization of the hierarchical approach of the geometric multigrid method so that it is not dependent on the availability of the meshes used for discretizing a PDE, but can be used a black-box solver for a given linear system of equations. In the geometric multigrid method, successively finer meshes are constructed by a geometric refining of the coarser meshes. On the other hand, the starting point for AMG is the final system of equations (analogous to the finest grid), from which successively smaller (coarser) systems are constructed. The coefficients of a coarse system in AMG are only algebraically related to the coefficients of the finer systems, which is in contrast to the geometric relationship between successive grids in geometric multigrid.

Just like geometric multigrid, an AMG method can be used either as a solver or a preconditioner. In either case, the method involves two phases. In the first set-up phase, the hierarchy of coarse systems is constructed from the original linear system and the prolongation and restriction operators are defined. The second solution phase consists of the prolongation, relaxation, and restriction cycles.

Efficient parallelization of AMG is much harder than that of geometric multigrid. As usual, the basis of parallelization is a good partitioning of the graph corresponding to the coefficient matrix and assigning the partitions to individual tasks. The solution phase, in principle, can then be parallelized with computation corresponding to the interior nodes remaining independent and that involving the nodes at or close to the boundary involving exchange of data with neighboring partitions. The boundary communication can be more involved than in the case of geometric multigrid because of the irregularity of the graph and the fact that the sets of interacting boundary nodes and the interaction pattern can be different for prolongation, relaxation, and restriction. However, the main difficulty in parallelizing AMG is in the set-up phase.

Effective parallelization of the set-up phase is essential for the overall scalability of parallel AMG because this phase can account for up to one-fourth of the total execution time. The process of construction of a coarser linear system in the classical AMG approach relies on a notion of the ”strength” of dependencies among coefficients. This process is not only sequential in nature, but also involves a highly non-local pattern of interaction between the vertices of the graph of the coefficient matrix. Therefore, the algorithm must be adapted for parallelization, which can make the convergence rate and per iteration cost of AMG dependent on the number of parallel tasks. Care must be taken to ensure that parallelization does not adversely effect the convergence rate or the iteration complexity of AMG. Typical approaches to parallelizing coarsening in AMG rely on decoupling the partitions, using parallel independent sets (similar to multicoloring described earlier), or performing subdomain blocking, which starts the coarsening at the partition boundaries and then proceeds to the interior nodes. Note that the coarsening scheme has an impact on the inter-task interaction during the prolongation and restriction steps of the solution phase.

When parallelizing AMG on large parallel machines, the number of parallel tasks may exceed the number of points in some of the grids at the coarsest levels. This situation requires special treatment. A commonly used work-around to this problem is *agglomeration*, in which neighboring domains are coalesced leaving some tasks idle during the processing of the coarsest levels. Another approach is to stop the coarsening when the number of coefficients per task becomes too small, which makes the behavior of the overall algorithm dependent on the number of parallel tasks.

2.5 Stochastic preconditioners

There has been a fair amount of research on algorithms for approximating the solution of linear systems based on random sampling of the coefficient matrix or on random walks in the graph corresponding to it. Most of these methods have been proven to work on a limited classes of linear systems only, such as symmetric diagonally dominant systems. A few practical solvers have been developed recently by using some of these techniques for preconditioning Krylov subspace methods. An attractive property of these methods is that they are usually trivially parallelizable. The quest for scalable massively parallel sparse linear solvers may prompt more active research into statistical techniques for preconditioning.

2.6 Matrix-free methods and physics-based preconditioners

Note that this article discusses preconditioners derived explicitly from the coefficient matrix A of the system $Ax = b$ that needs to be solved. In some applications, A is never constructed explicitly to save time and storage. Instead, it is applied implicitly to compute the matrix-vector products required in the Krylov subspace solver. In some of these cases, preconditioning is also applied implicitly, or the the knowledge of the physics of the application is utilized to construct the preconditioner, which cannot be derived from the coefficient matrix in a matrix-free method. Such preconditioners are called physics-based preconditioners. Due to the highly application-specific nature of matrix-free methods and physics-based preconditioners, these topics are not covered in further detail in this article.

3 Related Topics

- Krylov subspace methods
- Graph partitioning
- Linear algebra software
- Algebraic multigrid
- Domain decomposition
- Parallel algorithms for PDEs

4 Bibliographic Notes

The readers are referred to Saad's book [11] and the survey by Benzi [1] for a fairly comprehensive introduction to various preconditioning techniques. These do not cover parallel preconditioners and may not included some of the most recent work in preconditioning. However, these are excellent resources for gaining an insight into the state of the art of preconditioning circa 2002.

Hysom and Pothen [7] and Karypis and Kumar [8] cover the fundamentals of scalable parallelization of incomplete factorization based preconditioners. The work of Grote and Huckle [6] and Edmond Chow [3, 4] is the basis of modern parallel sparse approximate inverse preconditioners. Chow et al.'s survey [5] should give the readers a good overview of parallelization techniques for geometric and algebraic multigrid methods.

Last, but not the least, almost all parallel preconditioning techniques rely on effective parallel heuristics for two critical combinatorial problems—graph partitioning and graph coloring. The readers are referred to papers by Karypis and Kumar [9, 10] and Bozdag et al. [2] for an overview of these.

References

[1] Michele Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

[2] Doruk Bozdag, Assefaw H. Gebremedhin, Fredrik Manne, Eric G. Boman, and Umit V. Catalyurek. A framework for scalable greedy coloring on distributed memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535, 2008.

[3] Edmond Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing*, 21(5):1804–1822, 2000.

[4] Edmond Chow. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *International Journal of High Performance Computing Applications*, 15(1):56–74, 2001.

[5] Edmond Chow, Robert D. Falgout, Jonathan J. Hu, Raymond S. Tuminaro, and Ulrike Meier Yang. A survey of parallelization techniques for multigrid solvers. In Michael A. Heroux, Padma Raghavan, and Horst D. Simon, editors, *Parallel Processing for Scientific Computing*. SIAM, 2006.

[6] M. J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3), 1997.

[7] David Hysom and Alex Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal on Scientific Computing*, 22(6):2194–2215, 2000.

[8] George Karypis and Vipin Kumar. Parallel threshold-based ILU factorization. Technical Report TR 96-061, Department of Computer Science, University of Minnesota, 1996.

[9] George Karypis and Vipin Kumar. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical Report TR 97-060, Department of Computer Science, University of Minnesota, 1997.

[10] George Karypis and Vipin Kumar. Parallel algorithms for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.

[11] Yousef Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition.* SIAM, 2003.