Brigham Young University

BYU ScholarsArchive

Faculty Publications

2005-04-01

Prioritized Multiplicative Schwarz Procedures for Solving Linear Systems

Nathaniel Powell
powell.nathan@gmail.com

Kevin Seppi
Brigham Young University, kseppi@byu.edu

Quinn O. Snell
snell@cs.byu.edu

David Wingate

Follow this and additional works at: https://scholarsarchive.byu.edu/facpub

Part of the Computer Sciences Commons

Original Publication Citation

Prioritized Multiplicative Schwarz Procedures for Solutions to General Linear Systems, David Wingate, Nathaniel Powell, Quinn Snell, Kevin Seppi, Proceedings of the 25 International Parallel and Distributed Processing Symposium, April 25.

BYU ScholarsArchive Citation

Powell, Nathaniel; Seppi, Kevin; Snell, Quinn O.; and Wingate, David, "Prioritized Multiplicative Schwarz Procedures for Solving Linear Systems" (2005). *Faculty Publications*. 388.
https://scholarsarchive.byu.edu/facpub/388

This Peer-Reviewed Article is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Faculty Publications by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.

Prioritized Multiplicative Schwarz Procedures for Solving Linear Systems

David Wingate, Nathaniel Powell, Quinn Snell, Kevin Seppi
Computer Science Department
Brigham Young University
Provo, Utah 84602
{wingated,nep8,snell,kseppi}@cs.byu.edu

Abstract

We describe a new algorithm designed to quickly and robustly solve general linear problems of the form $Ax = b$. We describe both serial and parallel versions of the algorithm, which can be considered a prioritized version of an Alternating Multiplicative Schwarz procedure. We also adopt a general view of alternating Multiplicative Schwarz procedures which motivates their use on arbitrary problems (even which may not have arisen from problems that are naturally decomposable) by demonstrating that, even in a serial context*, algorithms should use many, many partitions to accelerate convergence; having such an over-partitioned system also allows easy parallelization of the algorithm, and scales extremely well. We present extensive empirical evidence which demonstrates that our algorithm, with a companion subsolver, can often improve performance by several orders of magnitude over the subsolver by itself and over other algorithms.*

1. Introduction

This work introduces the concept of prioritization applied to Alternating Multiplicative Schwarz (AMS) procedures [11] as an effective method for solving problems of the form $Ax = b$, where A is a large, sparse matrix.

AMS procedures are a form of problem decomposition that were originally motivated by trying to solve continuous PDEs with irregular domains. The principle is to divide the problem into smaller subproblems (which are still continuous), then repeatedly sweep over the subproblems, solving each one and updating interface values between problems. These PDEs were not necessarily discretized, and in some cases, the subproblems could be solved directly. The technique was then generalized to discretized PDEs, where a matrix A resulting from a discretization process is broken up into submatrices, each of which correspond directly to some part of the original problem [10].

In contrast, we explore the use of AMS-like procedures for arbitrary matrices, regardless of where they originated. In that sense, this work reverses the traditional view of AMS procedures: instead of starting with a known problem and generating a matrix representing its discretization, we begin with a matrix, and force a decomposition upon it, even if the underlying problem is not naturally decomposable. This means that the decomposition may or may not correspond to anything recognizable in the underlying problem. The results are excellent, in some cases accelerating solution times by many orders of magnitude.

The centerpiece of this paper is the development of *prioritized* AMS procedures, in which subdomains are selected according to some priority metric. The principle behind this is the observation that variables with large error strongly influence the solution vector. Corrections of large errors propagate additional error backwards through the graph defined by the A matrix. Our algorithm processes variables in priority order by constructing a priority queue using a metric derived from the residual. Such prioritized solution methods have shown to provide tremendous speedups in other contexts [9][12] for two reasons: first, they focus computation in regions of the problem which are expected to be maximally productive, and second, they can sometimes avoid processing large parts of the problem.

It is well-known that AMS procedures are naturally parallelizable. However, most researchers using AMS procedures as a domain decomposition method split the matrix into a small number of partitions, typically equal to the number of processors. One of the central points of this work is the fact that, even on a single processor, dramatic improvements are produced if a large number of partitions are used to solve a problem. We demonstrate that in a parallel context, far more partitions than processors should be used. This allows the same performance benefits to be gained on each individual processor, and improves the load-balancing and cache coherency of the algorithm.

The goal of this research is to speed up solution times by building on top of existing algorithms. However, our modi-

IEEE COMPUTER SOCIETY

fied AMS procedures are not accelerators (in the sense defined by Hageman and Young [7]), nor can they be considered preconditioners (as an additive Schwarz procedure can be). No simple linear algebra expression exists which captures the essence of the algorithm or which facilitates easy analysis. For those reasons, we present a more procedural exposition. Since we treat AMS procedures as an algorithmic accelerator for other subsolvers, and not as a preconditioner, we use lambda calculus notation to explicitly recognize the fact that our algorithm must be used in conjunction with another subsolver. This subsolver may be *any* solution method, including iterative methods, such as GMRES, CGS, TFQMR, BiCGSTAB, or direct methods, such as more traditional (I)LU decomposition or Gaussian elimination [2]. Thus, we have named the resulting algorithm GPS(λ), which is short for General Prioritized Solver, where λ is the companion subsolver. The parallel implementation is called PGPS(λ), which is short for Parallel General Prioritized Solver.

Because we wish to think about AMS procedures without regard to the underlying problem, we take a perspective akin to partial differentiation on the matrix A: we force arbitrary subsolvers to solve for a certain set of variables, while holding the rest of the variables constant. We make no assumptions about the form of the matrix; for example, we do not assume that the matrix has a diagonal structure (with any sized bandwidth). Instead of a traditional block decomposition, we adopt more general partitioning notation, and rely on general graph partitioners to decompose the problem and to determine interface edges. Thus, the ideas of overlap and interface value communication are not treated in the traditional way. However, if a naive partitioning is used (in which each partition is composed of a contiguous block of variables), and the matrix is already in a strongly diagonal form, then the result is mathematically equivalent to traditional AMS procedures, which is why we consider our algorithm a slight generalization of them.

Empirical results demonstrate that the serial version of the final algorithm almost always improves performance for a variety matrix types and sizes, and for a variety of subsolvers. Performance is often improved by several orders of magnitude, but for different reasons with different matrix/solver combinations. Improved performance is preserved in the parallel version, and excellent parallel scalability is demonstrated.

2. The GPS(λ) Algorithm

Here, we present the serial version of the GPS(λ) algorithm. The next section discusses parallel extensions. The core ideas underlying the GPS(λ) algorithm are quite simple: we partition the variables, we select a subset of variables based on priority, we solve the subset while holding all other variables constant (using any linear system solver, denoted λ), and then we update any quantities that depended upon variables in the subset. The entire process is repeated until convergence. There are four details that merit attention: 1) How can an arbitrary subsolver be made to solve for some variables, while holding others constant? 2) How should partitions be created? 3) How should partitions be selected in order to maximize the efficiency of the algorithm? 4) To what tolerance should the subproblem be solved? The following subsections address each question.

A few pieces of notation will be used throughout the paper. Let P be a partitioning of variable indices, with each $p \in P$ being a set of variable indices. Let $r = Ax - b$ be the global residual, and $r_p = A_p x_p - b_p$ be residual of a subproblem. Let ξ_G denote $\|r\|_2^2$, and ξ_p denote $\|r_p\|_2^2$.

Folding/Extracting One idea motivating the GPS(λ) algorithm is the idea of solving for a subset of all variables, while holding the rest of the variables constant. We term our method *folding and extracting*, and is an approach compatible with any sub-solver. The key observation is the fact that if certain variables are held constant, their values may be temporarily folded into the right-hand side vector, and a new sub-problem created.

As an example, consider the system $Ax = b$ where A is a 2x2 matrix. Suppose that we wish to solve only for variable 0 while holding variable 1 constant. We may expand this system (using array subscript notation to index vectors and matrices) and, since $x[1]$ is constant, rewrite it as

$$\begin{aligned} A[0,0]x[0] &= b[0] - A[0,1]x[1] &= b'[0] \\ A[1,0]x[0] &= b[1] - A[0,1]x[1] &= b'[1] \end{aligned}$$

where b' is the new vector that results from the folding operation. This yields a set of two equations with one unknown. Either may be used to solve for variable 0, although since our estimate for $x[1]$ is not guaranteed to be correct, different equations will result in different answers (see Section 7 for more discussion on this point).

In general, given an $n \times n$ matrix A and a partition p, the equation for variable $i \notin p$ is:

$$\sum_{j \in p} A[i,j]x[j] + \sum_{j \notin p} A[i,j]x[j] = b[i]$$
$$\sum_{j \in p} A[i,j]x[j] = b[i] - \sum_{j \notin p} A[i,j]x[j] = b'[i]$$

This yields n equations in $|p|$ unknowns, which is an over constrained system. We will select a subset of the equations with which to work; we adopt the convention that we will use the equations corresponding to the variable indices in the partition. We define a *selector matrix* as

$$K_p[i,j] = \begin{cases} 1 & i \in p \wedge i = j \\ 0 & \text{otherwise} \end{cases}$$

Then, to select a subset of equations from A, x and b, we let $A' = K_p A K_p^T$, $x' = K_p x$, and $b' = K_p b$.

Note that A' is still an $n \times n$ matrix, but with many empty rows and columns. If row i in A is empty, column i will also be empty, as will the corresponding entries in both both x' and b'. The entire system may therefore be compacted by eliminating such zero entries, and mapping variable indices from the original system to new variable indices in the compacted system: $A_p = \texttt{compact}(A')$, $x_p = \texttt{compact}(x')$, and $b_p = \texttt{compact}(b')$. The entire system $A_p x_p = b_p$ may now be passed to an arbitrary subsolver.

Other authors define similar reduction procedures in terms of *restriction* and *prolongation* operators [6][10]. We use our notation for simplicity of exposition.

Partitioning There are several principles governing the creation of partitions. Partitioning variables is different than a traditional block decomposition on the matrix A, because a partition may contain *any* subset of variables. Block decompositions effectively stipulate that variables be partitioned contiguously.

There are still some constraints on the partitioning. First, partitions must ensure that no sub-problem is under constrained. This is equivalent to ensuring that every submatrix has no empty rows or columns (unless both are empty), which is equivalent to stipulating that the binary relation defined by the sub-matrix be total and surjective. General graph partitioning algorithms, such as multi-level k-way partitioners [8], or k-way partitioners using spectral bisection, do not necessarily generate partitions that satisfy this criterion. However, both conditions are satisfied for any partitioning when the matrix has a non-zero diagonal. In this situation, it is possible to create "naive" partitions where each partition simply contains $n/|P|$ contiguous variables.

Although in theory an arbitrary partitioning of the variables could be used, the principle of partitioning is to group related variables together. The idea is that if one variable has a high priority, the related variables in the partition will also have a high priority, and we can solve them all simultaneously without incurring the overhead of prioritizing each variable individually. A good way to accomplish is to use a partitioning which minimizes the edge cut, which is a common capability of existing k-way partitioning algorithms.

Prioritization Once partitions are defined, the next question is determining the order in which partitions should be processed. There are two options: first, a naive sequential method may be used. Although simple, this method is surprisingly effective for well-ordered matrices, as demonstrated in Section 6. A second option is to process partitions in priority order.

For many of the experiments, we therefore define a priority queue which orders partitions. The queue has a priority metric associated with it, which we denote $H(p)$. For reasons that will be explained shortly, we use the 2-norm-squared of the residual vector of each sub-problem as the priority of that partition: $H(p) = \xi_p = \|r_p\|_2^2$. In principle, many different priority metrics could be used, but this is left to future research.

For the purposes of the algorithm, the priority queue must support three operations: `insert`, `pop`, and `reprioritize`. The third operation is necessary because the priority of a partition will often change when it is deep in the queue, as other partitions, upon which it depends, are solved.

Subproblem tolerance The final issue involves determining the tolerance to which subproblems should be solved. There are three possibilities: specify a maximum number of iterations, specify a subproblem tolerance, or both.

Specifying a subproblem tolerance merits some discussion. Assume that we wish to solve $Ax = b$ to tolerance of ϵ, using a 2-norm measure on the residual vector. Clearly, it is not sufficient to solve each subproblem to within ϵ, because the combination of the residual norms of all of the subproblems may yield an overall residual norm which is greater than the global tolerance.

Instead, it is necessary to *allocate error* to different subproblems. If we assume that error is allocated uniformly, for instance, then $\left(\sum_{p \in P} \xi_p\right)^{1/2} < \epsilon$, and by the uniformity assumption, $(|P|\xi_p)^{1/2} < \epsilon$, meaning that $\xi_p^{1/2} < \frac{\epsilon}{|P|^{1/2}}$. This states that if error is allocated uniformly, the subproblem tolerance must be less than ϵ to ensure that the global tolerance will be achieved.

However, this is only one way of allocating error to partitions. Other methods, including methods which dynamically allocate error as the algorithm progresses, are left to future research. In our algorithm, we specify both a subdomain tolerance, as well as a maximum number of subdomain iterations.

2.1. Algorithm Details

Here, we present the final algorithm which incorporates the issues discussed in this section. We will briefly discuss an optimization related to incremental error computations, and then present the final algorithm.

Incremental error computations As previously noted, many different stopping criteria could be used with the GPS(λ) algorithm. We have presented the most common, which is the 2-norm of the residual vector (this is also the criteria that was used in all of our experiments). The extension to other stopping criteria is straightforward, but messy, and has been omitted. An advantage of the 2-norm of the

Algorithm 1 $\texttt{GPS}(\lambda)$

Initialization

```
1: r := b − Ax
2: ξ_G := ||r||_2^2
3: for all p ∈ P do
4:    A_p := create_submatrix(p, A)
5:    (x_p, b_p) := fold(p, A, x, b)
6:    r_p := b_p − A_p x_p
7:    ξ_p := ||r_p||_2^2
8:    PQ.insert(p)
```

Main Loop

```
1: repeat
2:    // Solve the highest priority partition
3:    p := PQ.pop()
4:    (x_p, b_p) := fold(p, A, x, b)
5:    x_p := λ(A_p, x_p, b_p)
6:    (x, r) := extract(p, x, x_p, r, r_p)
7:    r_p := b_p − A_p x_p
8:
9:    // Update the global residual
10:   ξ_G := ξ_G − ξ_p
11:   ξ_p := ||r_p||_2^2
12:   ξ_G := ξ_G + ξ_p
13:
14:   // Update partition and variable residuals
15:   for all i ∈ VarDep(p) do
16:      p' := var_to_part(i)
17:      ξ_p' := ξ_p' − r[i]^2
18:      ξ_G := ξ_G − r[i]^2
19:      r[i] := b[i]− < A[i, ∗]^T, x_i >
20:      ξ_p' := ξ_p' + r[i]^2
21:      ξ_G := ξ_G + r[i]^2
22:
23:   // Reprioritize partitions
24:   for all p' ∈ PartDep(p) do
25:      PQ.reprioritize(p')
26: until ξ_G < ϵ^2
```

Algorithm 2 $\texttt{fold}(p, A, x, b)$

```
1: for all i ∈ p do
2:    i' := gv_to_lv(p, i)
3:    b_p[i'] := b[i]
4:    x_p[i'] := x[i]
5:    for all j ∈ A[i] do
6:       if j ∉ p then
7:          b_p[i'] := b_p[i'] − x[j]A[i, j]
8: return x_p, b_p
```

Algorithm 3 $\texttt{extract}(p, x, x_p, r, r_p)$

```
1: for all i ∈ p do
2:    i' := gv_to_lv(p, i)
3:    x[i] := x_p[i']
4:    r[i] := r_p[i']
5: return x, r
```

residual vector is that efficient incremental computation can be derived, which is compatible with the fact that, due to the prioritization aspects of GPS(λ), variables do not always need to be processed. Practically, squaring and square root operations can be avoided if the 2-norm squared is compared to the tolerance squared; our algorithm employs this. We use this technique for computing global error estimates, as well as maintaining error estimates for the sub-problems. Empirically, the technique is stable, but suffers from round-off errors, which is remedied by periodically performing a full residual norm calculation (not shown in Algorithm 1).

Full algorithm Algorithm 1 shows the full pseudocode for the serial GPS(λ) algorithm. Here, $\texttt{VarDep}(p)$ returns the *variable dependents* of a partition p, which is the set of all variables with an edge to any variable in p. Similarly, $\texttt{PartDep}(p)$ returns the *partition dependents* of a partition p, which is the set of all partitions which contain at least one variable in $\texttt{VarDep}(p)$. The gv_to_lv function ("global-variable-to-local-variable") maps variable indices from A to submatrix indices in A_p, and the var_to_part function maps variables to their owning partitions.

3. Parallel Implementation

The enhancements of partitioning and prioritization naturally facilitate an efficient parallel implementation. This section discusses one way in which the GPS(λ) algorithm can be parallelized, and discusses some of additional issues encountered when moving to a parallel architecture.

In our algorithm, each processor owns (or is responsible for) a set of partitions. These partitions are termed "local" partitions, which contain "local" variables. All other partitions and variables are "foreign." In a corresponding generalization of notation, we will use ξ_{Pr} to denote the sum of all ξ_p for p local to processor Pr.

To create PGPS(λ), we adopted an architecture with some asynchronous aspects as well as some synchronous aspects. The basic algorithm is substantially the same as the serial algorithm, except for two major changes which merit detailed discussion. First, since partitions are assigned to processors, there is the issue of communicating the values of variables between processors. Secondly, there is the issue of termination.

3.1. Variable Communication

The PGPS(λ) algorithm embodies the same core concepts as the serial version, except that the data and computation has been distributed. First, each processor selects

IEEE COMPUTER SOCIETY

the local partition p with the highest priority and solves it. Then, the priorities and residuals of any other local partitions which depend on p are recomputed. Now, however, PGPS(λ) must execute steps that GPS(λ) did not need to: it must communicate the values in p to all other processors which depend upon p, and it must receive and process updates from other processors.

The values in a partition do not necessarily need to be communicated to *every* other processor. In fact, the processors which need the new values for partition p are partitions which have some local variable that depends upon a value from a variable in p. We term this set `ProcDep`(p), and is typically very small in sparse matrices.

Communicating new values can be accomplished with a single message sent to each processor in `ProcDep`(p). To avoid starvation of processors, a processor receives a maximum of n messages, where n is the number of processors.

Receiving an update message from a foreign processor is conceptually the same as the solving a local subproblem associated with a partition p. The various quantities that depend on variables within p must be recomputed: partition priorities, partition error estimates, the global error estimate, and variable residuals.

3.2. Termination Detection

Since a processor is only responsible for a subset of the variables, it can only compute a subset of the overall error. Each processor's error estimate must be combined to form a global error estimate, which can be used to determine when to terminate.

Although there are many ways to compute such a global error estimate, an effective way is a periodic reduction. In our implementation, all processors reduce the local ξ_{Pr} error values every 100 iterations. The number of iterations between synchronizations is a tunable parameter. Theoretically, if it is set too high, the algorithm performs unnecessary work, and if it is set too low, there can be significant overhead. Empirically, however, it had very little impact on the solution times.

3.3. Discussion

There are several advantages of this quasi-synchronous architecture. First, since processors communicate changed values only to the processors that need it, and since `ProcDep`(p) is typically very different for different partitions, processors can work at their own rate. In addition, the overall volume of communication is fairly small, since partitions are usually selected to contain no more than about 100 variables. Naturally, all master/slave bottlenecks are avoided.

The primary disadvantage is that many small messages are generated, which can incur high latency penalties. In addition, the synchronous computation of error means that processors cannot work *completely* at their own rate. This creates some inefficiencies that could be remedied with a more sophisticated termination detection.

3.4. Full Algorithm

Algorithm 4 shows the full PGPS(λ) algorithm. Note that processors now operate only on local partitions. The PQ contains only local partitions, and whenever a partition changes (either as a result of popping off the local PQ and solving, or as a result of a message from a foreign processor), only local errors and residuals are recomputed.

Here, `LocalVarDep`(p) returns the *local variable dependents* of a partition p, which is the same as `VarDep`(p), except that it contains only variables local to Pr. `LocalPartDep`(p) returns the *local partition dependents* of a partition p, and only contains local partitions. The `ProcDep`(p) function returns a set of *processor dependents* of a partition p, which is the set of all processors that own one variable which depends upon one variable in p.

Lines 2-5 show pseudocode for the simple synchronous termination test. Lines 18-38 constitute the bulk of the difference between PGPS(λ) and GPS(λ), but is conceptually simple: each partition update message received triggers processing that is the same as the processing which happens which a local partition changes.

4. Algorithm Design Details

There are many issues surrounding the design of an algorithm such as GPS(λ). This section briefly examines some of the more theoretical aspects, but we stress that space precludes a full analysis. Here, we only briefly consider convergence and miscellaneous design details.

4.1. Convergence

Technically, the introduction of prioritization into the solution process moves the GPS(λ) algorithm into the class of chaotic (or asynchronous) relaxation algorithms. Proofs of convergence of such algorithms have traditionally imposed strict requirements on the problems to be solved; a typical condition is that $\rho(A) < 1$ [3][5][4].

In principle, the convergence of GPS(λ) should therefore be governed by two factors: first, A must satisfy any constraints that the subsolver requires. If CGS is selected as a subsolver, for instance, then A must be symmetric positive definite. Second, it seems that A should satisfy any additional constraints that asynchronous relaxations impose.

Algorithm 4 PGPS(λ)

Main Loop

```
1:  repeat
2:    // Synchronously compute global error
3:    if time to resync then
4:      ensure all messages have been received
5:      ξ_G = Reduce(ξ_Pr)
6:
7:    // Solve the highest priority partition
8:    // and update the global residual
9:    same as serial
10:
11:   // Update partition and variable residuals
12:   // and reprioritize partitions
13:   for all i ∈ LocalVarDep(p) do
14:     same as serial
15:   for all p' ∈ LocalPartDep(p) do
16:     same as serial
17:
18:   // Send new partition values
19:   for all i ∈ ProcDep(p) do
20:     Send(i, p, x_p)
21:
22:   // Receive new values
23:   for i = 1 to nproc do
24:     if no messages waiting then
25:       break
26:     (p', x_p') = Recv()
27:     // Update partition and variable residuals
28:     for all i ∈ LocalVarDep(p') do
29:       p'' := var_to_part(i)
30:       ξ_p'' := ξ_p'' − r[i]^2
31:       ξ_Pr := ξ_Pr − r[i]^2
32:       r[i] := b[i]− < A[i,*]^T, x_i >
33:       ξ_p'' := ξ_p'' + r[i]^2
34:       ξ_Pr := ξ_Pr + r[i]^2
35:
36:     // Reprioritize partitions
37:     for all p'' ∈ LocalPartDep(p') do
38:       PQ.reprioritize(p'')
39: until ξ_G < ε^2
```

Empirically however, the GPS(λ) algorithm is unique in that neither constraint appears to be fully in force. In our experiments, GPS(λ) robustly converged for almost *all* matrices tested (assuming that preconditioner succeeded on the sub-problems), each of which was non-convergent (that is, $\rho(A) \geq 1$). In addition, at least in theory, the GPS(λ) algorithm actually *relaxes* some of the constraints imposed on solvers and matrices. The issues of convergence merit detailed attention, so they are largely left for future research. However, the idea that GPS(λ) relaxes some requirements merits an example. Consider the matrix

$$A = \begin{bmatrix} 3 & 2 & 1 & 0 \\ 2 & 3 & 0 & 4 \\ 0 & 0 & 3 & 2 \\ 0 & 0 & 2 & 3 \end{bmatrix}$$

and a right-hand side of $b = [1, 1, 1, 1]^T$. Although A is positive definite, it is not symmetric. A solver that requires that the matrix be SPD, such as CG, diverges when attempting to solve $Ax = b$. However, if we define two partitions $p_1 = \{1, 2\}$ and $p_2 = \{3, 4\}$, then $A_1 = A_2 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix}$ both of which are positive definite. CG can therefore solve both sub-problems, meaning that GPS(CG) can be used to solve the entire system.

The rate of convergence of GPS(λ) is unknown. Since it is technically a chaotic relaxation algorithm, all known convergence proofs simply state that if certain conditions relating to the matrix are met, and all variables are relaxed an infinite number of times, the system will converge. No guaranteed rates are known except in very specific circumstances, but empirically, we have observed excellent rates.

Additional material and analysis on convergence of AMS procedures can be found in [6].

4.2. Miscellaneous Details

Currently, determining when to stop the subsolver is largely heuristic. As noted, GPS(λ) uses a combination of a subproblem tolerance and a small maximum number of subproblem iterations. Although sufficient for convergence, both methods leave something to be desired, because both numbers are static. Some initial experimentation has indicated that dynamically changing the subproblem tolerance (or the maximum number of subdomain iterations) can yield improved performance. This is consistent with basic principles: it is good to work quickly across many different parts of a problem, to quickly reduce large errors. However, as time progresses, it becomes more important to solve subproblems to a higher accuracy.

5. Experimental Setup

A number of experiments were run to isolate and quantify all relevant behaviors of the GPS(λ) and PGPS(λ) algorithms. The bulk of the experiments were designed to quantify the benefits of different enhancements, and across several different subsolvers. It examined 1) the folding / extracting technique; 2) the benefits of prioritization; and 3) the efficiency of the parallel version of the code.

Experiments were run across a wide variety of matrices, which are described in Figure 1. An effort was made to select matrices from different discplines, with different sizes,

Matrix name	Description	Size	NNZ	RHS	SPD
bcsstk08	BCS – TV studio	1,074	7,017	(made)	Yes
e20r0000	2D fluid flow in a driven cavity	4,241	131,556	MM	No
e40r5000	2D fluid flow in a driven cavity	17,281	553,956	MM	No
fidap003	Matrix from the FIDAP package	1,821	52,659	MM	No
fidap006	Matrix from the FIDAP package	1,651	49,479	MM	No
fidap010	Matrix from the FIDAP package	2,410	54,816	MM	No
fidap011	Matrix from the FIDAP package	16,614	1,091,362	MM	No
fidap014	Matrix from the FIDAP package	3,251	66,647	MM	No
fidap035	Matrix from the FIDAP package	19,716	218,308	MM	No
fidapm33	Matrix from the FIDAP package	2,353	23,765	MM	No
fs_760_3	Mixed kinetics diffusion problem	760	5,976	(made)	No
hp_400_5	Thermal convection; 5-pt stencil	160,000	798,400	(in-house)	No
mcar-160000	Reinforcement learning problem	160,000	637,831	(in-house)	No
nnc1374	Advanced gas-cooled nuclear reactor core	1,374	8,606	(made)	No
rdb2048l	2D reaction-diffusion model	2,048	12,032	(made)	No
sap-160000	Reinforcement learning problem	160,000	639,996	(in-house)	No
s3dkt3m2	FEA of cylindrical shell	90,449	1,921,955	(made)	Yes
sherman2	Thermal simulation with steam injection	1,080	23,094	MM	No
sherman3	IMPES simulation of a black oil model	5,005	20,033	MM	No

Figure 1. Descriptions of the matrices used.

and with different numerical properties. Most of the matrices used can be found at NIST's MatrixMarket. [1] Exceptions are labeled with an "in-house" label in the RHS column of Figure 1.

A fair comparison of GPS(λ) to other solvers is not easy. We selected unpreconditioned GMRES as the general baseline, because of its popularity and robustness across many different matrix types. As we will show, GPS(λ) often outperforms GMRES. On the one hand, it is not surprising that GPS(λ) outperforms an unpreconditioned solver, but on the other hand, GPS(λ) is not preconditioned either (this is because it is theoretically possible to develop a preconditioned version of GPS(λ) which operates on a preconditioned matrix, instead of directly on the matrix A). The most natural direct competitor is GMRES, preconditioned with an Additive Schwarz Method, and using a standard ILU subdomain preconditioner. With these issues in mind, most of the experiments were run using GMRES (labeled as "P-GMRES"), GMRES with an ASM+ILU preconditioner (labeled as "P-GMRES w/ASM+ILU"), GPS(λ) (labeled as "GPS-PQ" or "GPS-SEQ"), and GMRES with a standard MAT preconditioner (labeled as "P-GMRES w/MAT").

For most of the experiments, the parameters for GPS(λ) (such as number of partitions, maximum number of subdomain iterations, etc.) were aggressively hand-tuned to yield the best performance. When solving the problems, tolerances were selected primarily to elicit interesting behavior. For GPS(λ), when solving to a specified tolerance, subproblem tolerances were always set to be $\epsilon/\sqrt{|P|}$.

For all of the experiments the subsolvers incorporated into the GPS(λ) code were taken from the "Portable, Extensible Toolkit for Scientific Computation" (PETSc) library [1]. Experimentation with Sandia National Laboratory's "Aztec" package yielded worse performance, and did not have the preconditioners needed, so it was dropped.

To help isolate the benefits of the fold/extract approach from the benefits of prioritization, we will also reference an algorithm named "GPS-SEQ." This is a variant of the GPS(λ) algorithm which selects partitions sequentially, instead of in priority order. If clarification is needed, we will reference the prioritized version of GPS(λ) as "GPS-PQ."

Many of the matrices from the MatrixMarket have a RHS that is also downloadable, but some do not. For those that do not, one was manufactured by selecting a random x vector with each $x[i] \in (0, 1)$, then multiplying it through A to generate b. These are indicated in Figure 1 with "(made)" in the RHS column. Regardless of where the RHS came from, an initial guess of $x = 0$ was always used.

Both the GPS(λ) and PGPS(λ) depend on having a partition of the variables. Since almost all of the matrices are strongly diagonal, a naive partitioning was used. Each partition was assigned $n/|P|$ contiguous variables.

Serial results were obtained on a dual processor 2.4GHz P4 machine with 2G of RAM. Parallel results were obtained on a fully connected cluster of dual processor 2.4GHz P4 machines, each with 2G of RAM and equipped with

1 http://math.nist.gov/MatrixMarket/

IEEE COMPUTER SOCIETY

Myrinet interconnects. All code was written in C with MPI.

6. Results

The results of our experiments were very positive. We begin by presenting the general serial results, followed by the parallel scalability results. To be fair, we point out cases in which the algorithm fails, or yields worse performance than a baseline. The primary criterion measuring success is the wall-clock time needed to reduce the (unpreconditioned) 2-norm of the residual to a specific tolerance.

6.1. Serial results

Figure 2, top three rows, shows a representative sample of results for the GPS(λ) algorithm on a variety of matrices. The story is complicated: the various algorithms outperform each other on different problems, and even on the same problem, different algorithms may win or lose based on what tolerance was selected. For this reason, we opted to present the majority of results in graphical form. However, several general patterns are clearly evident, and the winner on any given matrix is usually clear in the limit.

There are three broad types of result:

- GPS(λ) dramatically improves performance (by up to several orders of magnitude) [bcsstk08, e20r0000, fidap006, fidap014, sap-160000, sherman2]
- GPS(λ) barely affects performance [e40r5000, fidap011, fs_760_3, mcar-160000]
- GPS(λ) dramatically worsens performance (sometimes making the problem impossible to solve) [fidap003, fidap010, nnc1374, fidap035, fidapm33, hp_400_5, s3dkt3m2, rdb2048l, sherman3]

The most exciting results in our experiments indicate that GPS(λ) yielded excellent performance on a wide variety of matrices. On the bcsstk08 and sap-160000 problems (see Figure 2), GPS(λ) solves it instantly while other algorithms gradually reduce the residual norm. For the e20r0000 (Figure 2) and fidap014 problems, GPS(λ) reliably reduces the residual norm better than all other solvers. On the fidap006 problem, GPS(λ) bottoms out at a residual norm of 0.118 after about 3 seconds, while P-GMRES achieves a residual norm of 0.135 after 40 seconds. Of course, not every positive result is as compelling: on the sherman2 matrix, for example, GPS(λ) can outperform P-GMRES by two orders of magnitude, but only with a small number of partitions.

Naturally, GPS(λ) is not a panacea. For some problems (such as e40r5000), no algorithm worked. On the fs_760_3 problem, GPS(λ) stabilizes at a residual norm lower than that of P-GMRES, but GPS-SEQ achieves the lowest residual. The fidap011 problem was an interesting case: P-GMRES and GPS-PQ flip-flopped twice, although GPS-PQ eventually won out. For the mcar-160000 problem, GPS-PQ almost instantly solves it, but the GPS-SEQ also solves it almost instantly, and to a better tolerance. This case illustrates two important points: first, since GPS-PQ and GPS-SEQ performed about the same, it appears that prioritization is not the defining performance factor, but the fact that a multiplicative Schwarz procedure was being used. Second, it is clear that for a problem like this, a hybrid algorithm, in which some iterations of GPS(λ) are run until stabilization, followed by some iterations of another solver, would probably best either algorithm alone.

We also observed situations where GPS(λ) degraded performance substantially. Often, it was the case that although GPS(λ) could reduce the tolerance reliably, other algorithms managed to solve a problem instantly. In other cases, GPS(λ) drives down residual norm down quickly and reliably, but another algorithm achieves a lower norm even more quickly. The fidap003, fidap010, nnc1374 and fidap035 problems all showed an interesting general pattern: P-GMRES was able to reduce the residual norm quickly, but GPS(λ) could not. If the number of partitions used was adjusted downward, however, GPS(λ) approached closer and closer to the performance of P-GMRES.

6.2. Parallel Results

Figure 2 (bottow two rows) show representative parallel results for PGPS(λ). The first two graphs show the wall-clock time of P-GMRES, P-GMRES/ASM+ILU, PGPS-PQ and PGPS-SEQ, while the second two graphs show the speedup of the same algorithms. The parallel results are presented for the s3dkt3m2 and hp_400_5 matrices.

Here, we are benchmarking two distinct parallel implementations: the P-GMRES and P-GMRES/ASM+ILU algorithms are the standard PETSc implementations, using their MPI-based parallelization. Recall that although PGPS(λ) uses PETSc as a subsolver, it is not a *parallel* subsolver. The parallel code for PGPS(λ) was developed in-house.

The interpretation of these results requires some care. Specifically, the results of different algorithms should not be compared to each other. This is because the point of these figures is to show the parallel scalability and speedup of each algorithm on two different matrices. To demonstrate this in a clear way, we used different tolerances for different algorithms, meaning that each algorithm is effectively solving a different problem.

The results demonstrate excellent overall scalability. PGPS(λ) consistently scales almost linearly with the number of processors, and although some parallel overhead was seen, it did not appear to be as pronounced as some of the overhead that the PETSc library incurred. This is also an important validation of our assertion that cross-processor communication overhead is not pro-

hibitive in PGPS(λ), even with the quasi-synchronous architecture we adopted.

The speedup graphs indicate that, for both PGPS(λ) and PETSc, there is room for improvement. Neither package bests the other consistently, although GPS-SEQ always shows the best speedup. Of course, experiments on more than two matrices would need to be run to truly determine how the packages compare.

The competition with PETSc is significant not only because PGPS(λ) appears to scale better (demonstrating that an un-optimized prototype code is similar to a well-developed library relied on by many researchers), but also because it was easier to implement.

6.3. Miscellaneous Results

One meta result that does not fall neatly into any category is the fact that throughout our experimentation, very few cases of divergence were observed while using GPS(λ), assuming that the preconditioner suceeded on the subproblems. Considering the highly asynchronous nature of the algorithm, and the discussion of convergence in Section 4.1, this result is surprising, and merits further study. Finally, we note that tuning the parameters of the algorithm is difficult. For example, performance varies significantly as function of the number of partitions used, which is another issue for future research.

7. Conclusions and Future Research

Perhaps the best way to describe our final conclusions is this: initial experimentation using GPS(λ) and PGPS(λ) generated many promising results, and indicates significant potential for the idea of prioritization. Empirically, we have shown that even by themselves, the AMS procedures embodied in GPS(λ) and PGPS(λ) can provide dramatic performance benefits for many problems, regardless of the origin of the underlying problem. The addition of prioritization to these procedures can often further improve performance. Of course, there is no reason to suppose that prioritizing subdomains in an AMS procedure is the only way that prioritization can be integrated into an algorithm. Studying other ways to accomplish this (perhaps by deriving directly prioritized versions of algorithms) is a significant direction for future research. We have also shown that GPS(λ) is effectively parallelizable, allowing many different subsolvers to be used in a general parallel framework.

The GPS(λ) algorithm is not perfect, of course. Use of the GPS(λ) sometimes worsened performance over baseline algorithms, and sometimes lead to divergence (especially if the partitioning was not done carefully). Perhaps this result simply means that GPS(λ) fits squarely with other linear system solvers: no single algorithm always outperforms every other algorithm, and very few algorithms are guaranteed to converge on every problem.

Overall, our results indicate that AMS procedures can be effectively used to accelerate many problems. This is true even in the serial case, and even for matrices which came from an unknown underlying problem. Perhaps more generally, the results indicate that there is still room for improvements to the current generation of best solvers. This motivates continued research into advances such as prioritization with partitioning, in an effort to one day find the optimal way of solving large sparse linear systems.

Acknowledgements

David Wingate is supported under a National Science Foundation Graduate Research Fellowship.

References

[1] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[4] D. Chazan and W. Miranker. Chaotic relaxation. In *Linear Algebra and its Applications*, volume 2, pages 199–222, 1969.

[5] V. Gullapalli and A. G. Barto. Convergence of indirect adaptive asynchronous value iteration algorithms. In *Advances in Neural Information Processing Systems*, volume 6, pages 695–702, 1994.

[6] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*. Springer-Verlag, New York, 1994.

[7] L. A. Hageman and D. M. Young. *Applied Iterative Methods*. Academic Press, New York, NY, 1981.

[8] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[9] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

[10] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, 1996.

[11] H. A. Schwarz. *Gesammelte Mathematische Abhandlungen*, volume 2. Springer-Verlag, 1890.

[12] D. Wingate and K. Seppi. P3VI: A partitioned, prioritized, parallel value iterator. In *Proceedings of the Twenty-First International Conference on Machine Learning*, pages 863–870, 2004.

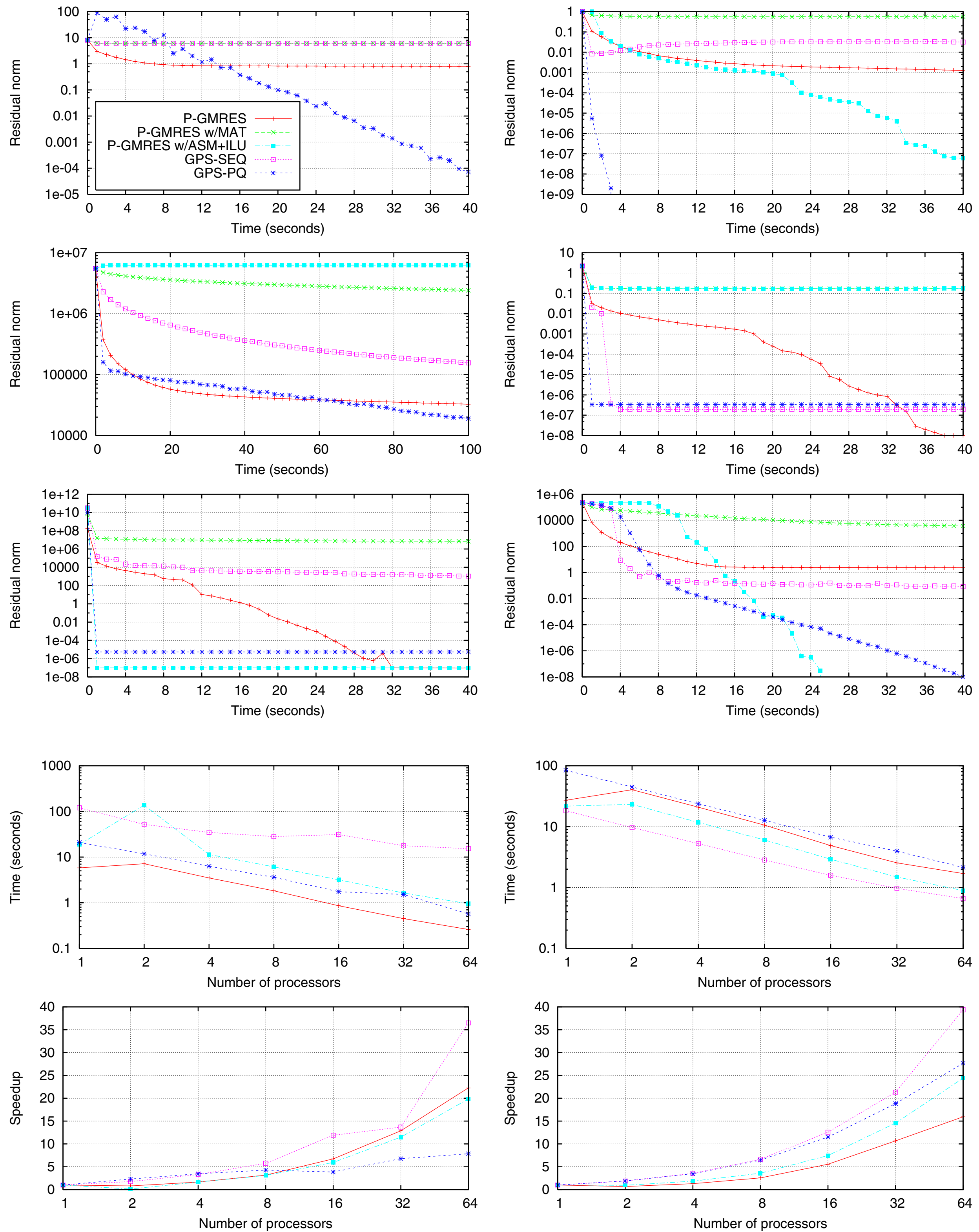

Figure 2. Performance results (top three rows). The matrices shown are (left to right, top to bottom) e20r0000, SAP-160000, fidap011, MCAR-160000, bcsstk08, and s3dkt3m2. Parallel performance results (fourth row) and speedup results (bottom row) for the s3dkt3m2 (left) and hp_400_5 (right) matrices. The same legend (shown in the upper-left graph) is used for all graphs.

IEEE COMPUTER SOCIETY