Prioritized Multiplicative Schwarz Procedures for Solutions to General 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.e du

***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.**

I. 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, and has motivated an effort to explore the benefits of AMS procedures as a *general* method.

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 therefore constructs a priority queue using a priority metric which is related to the residual of the variable and processes variables in priority order. 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 for parallelization 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 modified 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. In addition, no assumptions about the form of the matrix are made; for example, it is not assumed that the matrix has a diagonal structure (with any sized bandwidth). Instead of a traditional block decomposition formulation, 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 explicitly 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 a traditional AMS procedure, which is why we consider our algorithm a slight generalization of AMS procedures.

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.

The paper is organized as follows. Section II introduces the notation that will be used in the paper. Section III presents the serial GPS(λ) algorithm, while Section IV presents the parallel version. Section V discusses convergence and miscellaneous design details. Section VI presents our experimental setup, Section VII presents the results of our experiments, and Section VIII presents conclusions and future research.

II. Notation

Here, we briefly introduce the notation that will be used throughout the paper. We wish to solve the system of equations $Ax = b$, with $A \in \Re^{n \times n}$ and $x, b \in \Re^n$. Let λ be any linear system solver, such as GMRES, CGS, Gaussian elimination, etc. The GPS(λ) and PGPS(λ) algorithms invoke λ on several smaller systems of equations, which are defined in terms of partitions. Formally, let P be a partitioning of the variables in x. Each $p \in P$ is a set of indices, such that $i \in p$ means that variable x_i resides in partition p. Note that this is more general than AMS procedures, which usually stipulate that blocks be composed of contiguous variables. Sub-problems which λ solves are defined for each partition. Submatrices and subvectors will be subscripted with the corresponding partition, so that λ will be invoked to solve a system denoted by $A_p x_p = b_p$.

For PGPS(λ), we assign partitions to processors. These are termed "local" partitions, and they contain "local" variables. From the perspective of a processor, all other partitions and variables are "foreign."

```
Initialization
  1) Partition the variables
Repeat
  1) Select a subset of variables $p$ based on priority
  2) Solve the variables in $p$ using $\lambda$, while holding the
     rest of the variables in the system constant
  3) Recompute the residual of any variables that depend
     on variables in $p$, as well as the global error estimate
Until stopping criteria are met
```

Fig. 1. Basic steps in the GPS(λ) algorithm.

To avoid double-subscripting, we use array notation to denote specific elements in vectors and matrices. To reference a row vector corresponding to an entire row in a matrix, we use $A[i, *]$. $< a, b >$ denotes the dot-product of a and b. Unless otherwise specified, all vectors are column vectors.

We will use $r = Ax - b$ to be the global residual, and $r_p = A_p x_p - b_p$ to be residual of a subproblem. We will use the variable ξ_G to denote $\|r\|_2^2$. ξ_p denotes $\|r_p\|_2^2$. In the PGPS(λ) algorithm, ξ_{Pr} denotes the sum of all ξ_p for p local to processor Pr.

Let `VarDep`(p) be the *variable dependents* of a partition p, which is the set of all variables with an edge to any variable in p. Similarly, let `PartDep`(p) be the *partition dependents* of a partition p, which is the set of all partitions which contain at least one variable in `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. For PGPS(λ), let `LocalVarDep`(p) be 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) denotes 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. Informally, it is the set of all processors that own one variable which depends upon one variable in p.

III. The GPS(λ) Algorithm

Here, we present the serial version of the GPS(λ) algorithm. The next section discusses parallel extensions. We begin this section with an outline of the general principles of the algorithm, and then present the specific algorithm that results from our design decisions.

A. General Principles

An outline of the GPS(λ) algorithm is shown in Figure 1. The core ideas underlying the algorithm are quite simple: we solve for a subset of variables while holding the other variables in the system constant, and then update any quantities that depended upon variables in the subset. However, 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.1) 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 in the system constant. One way to accomplish this would be to derive modified versions of every algorithm that we wished to use. Practically, this is difficult, and requires a deep knowledge of the principles upon which each algorithm is based.

There is a much simpler way, however, which is to adopt a general method compatible with any algorithm. We term this a *fold/extract* approach. 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:

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

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 (Section VIII discusses some ideas for dealing with this problem in a principled manner).

More generally, assume that we are given an $n \times n$ matrix A and a partition p which contains a set of variable indices that we wish to solve for. Consider the equation for variable $i \notin p$:

$$\begin{aligned} \sum_{j}^{n} A[i,j]x[j] &= b[i] \\ \sum_{j \in p} A[i,j]x[j] + \sum_{j \notin p} A[i,j]x[j] &= b[i] \\ \sum_{j \in p} A[i,j]x[j] &= b[i] - \sum_{j \notin p} A[i,j]x[j] \\ \sum_{j \in p} A[i,j]x[j] &= b'[i] \end{aligned}$$

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. Specifically, we may 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

$$\begin{aligned} A' &= K_p A K_p^T \\ x' &= K_p x \\ b' &= K_p b \end{aligned}$$

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, and the corresponding entries in both both x' and b' will also be zero. 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:

$$\begin{aligned} A_p &= \texttt{compact}(A') \\ x_p &= \texttt{compact}(x') \\ b_p &= \texttt{compact}(b') \end{aligned}$$

This is the reason that we define the function `gv_to_lv()` function, which maps global variable indices to local variable indices.

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 adopt this notation for simplicity of exposition.

A.2) Partitioning: There are several principles governing the creation of partitions. We briefly note that 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, however. First, partitions must ensure that no sub-problem $A_p x_p = b_p$ is under constrained. This is equivalent to ensuring that every sub-matrix 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. Each variable must depend upon one other variable in the partition, and each variable must be depended upon by one other variable in the partition. We note that 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.

A.3) 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 VII. A second option is to process partitions in priority order, using a priority metric. As noted in the introduction, the principle is that variables with large residuals will tend to propagate that residual throughout the problem. The algorithm discussed in the next section is structured around the priority queue version of the code, although the extension to a sequential partition selection mechanism is trivial.

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. In [12], we report good results using a max-norm measure of the Bellman error for reinforcement learning problems. However, extending GPS(λ) to other priority metrics is left to future research.

For the purposes of the algorithm, the priority queue must support three operations: first, the standard operations of `insert` and `pop`. Second, it must support a general `reprioritize` method, which must be callable for arbitrary partitions. This is necessary because the priority of a partition will often change when it is deep in the queue. In our implementation, `reprioritize` was implemented by extracting the partition from the queue, and reinserting it with the new priority measure.

A.4) Subproblem tolerance: The final issue involves determining the tolerance to which subproblems should be solved. There are two possible ways to decide when a subsolver should stop: first, it is possible to specify a maximum number of iterations (which we term "subdomain iterations"), or it is possible to specify a subproblem tolerance, or both.

Specifying a subproblem tolerance merits some discussion. Assume that the original specification required us to solve $Ax = b$ to tolerance of ϵ, using a 2-norm measure on the residual vector:

$$\left(\sum_{p\in P}\xi_p\right)^{1/2} < \epsilon$$

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

$$\begin{aligned}\left(\sum_{p\in P}\xi_p\right)^{1/2} &< \epsilon\\ (|P|\xi_p)^{1/2} &< \epsilon\\ \xi_p^{1/2} &< \frac{\epsilon}{|P|^{1/2}}\end{aligned}$$

This states that if the 2-norm of each partition is less than ϵ divided by the square root of the number of partitions, then the global 2-norm will be less than the global tolerance.

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.

B. 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.

B.1) Incremental error computations: Before discussing the algorithm in detail, a few notes related to stopping criteria are necessary. As previously noted, many different termination criteria could be used with the GPS(λ) algorithm. To facilitate the exposition of the algorithm, we have elected to present the most common, which is 2-norm of the residual vector (this is also the stopping criteria that was used in all of our experiments). The extension to other termination criteria is straightforward, but messy, and has been omitted.

An advantage of the 2-norm of the residual vector is that efficient incremental computations can be derived which aids in accelerating the algorithm. One of the advantages of the prioritization aspects of GPS(λ) is the fact that for some problems, variables do not always have to be processed; the incremental recomputation of global terms such as the 2-norm of the residual vector is compatible with this idea. The key observation is the fact that the 2-norm of a vector r is a sum of squares:

$$\|r\|_2 = \left(\sum_i^n r[i]^2\right)^{1/2}$$

If only one element in r_{old} has changed to yield r_{new} (say, element i), and given that $\|r_{old}\|_2$ has already been computed, the 2-norm of r_{new} may be computed incrementally:

$$\|r_{new}\|_2 := \left(\|r_{old}\|_2^2 - r_{old}[i]^2 + r_{new}[i]^2\right)^{1/2}$$

Practically, the squaring and square roots can be avoided if the 2-norm squared is compared to the tolerance squared.

We have employed this incremental technique for computing global error estimates, as well as maintaining error estimates for the sub-problem associated with each partition. Empirically, the technique is stable, but suffers from round-off errors. This can be avoided by periodically doing a full residual norm calculation.

Algorithm 1 GPS(λ)

```
Initialization
1: $r := b - Ax$
2: $\xi_G := \|r\|_2^2$
3: for all $p \in P$ 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:    $\xi_p := \|r_p\|_2^2$
8:    PQ.insert$(p)$
9: end for

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:    $x_p := \lambda(A_p, x_p, b_p)$
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:   $\xi_G := \xi_G - \xi_p$
11:   $\xi_p := \|r_p\|_2^2$
12:   $\xi_G := \xi_G + \xi_p$
13:
14:   // Update partition and variable residuals
15:   for all $i \in$ VarDep$(p)$ do
16:      $p' :=$ var_to_part$(i)$
17:      $\xi_{p'} := \xi_{p'} - r[i]^2$
18:      $\xi_G := \xi_G - r[i]^2$
19:      $r[i] := b[i] - < A[i, *]^T, x_i >$
20:      $\xi_{p'} := \xi_{p'} + r[i]^2$
21:      $\xi_G := \xi_G + r[i]^2$
22:   end for
23:
24:   // Reprioritize partitions
25:   for all $p' \in$ PartDep$(p)$ do
26:      PQ.reprioritize$(p')$
27:   end for
28: until $\xi_G < \epsilon^2$
```

B.2) Full algorithm: Algorithm 1 shows the full pseudocode for the serial GPS(λ) algorithm. The algorithm is divided into several basic pieces.

The `Initialization` section initializes the global error estimate, creates the submatrices needed for folding, and computes the residual and residual norm for each partition. The submatrix creation is detailed in Algorithm 2.

The `Main Loop` section is divided into four main parts. In lines 2-7, the algorithm selects the highest priority partition p. The relevant portions of the current x estimate are folded into the subproblem, and λ is invoked to solve it. We stipulate that λ solve the subproblem to a tolerance of $\epsilon/\sqrt{|P|}$. The results

Algorithm 2 create_submatrix(p, A)

```
1: $A_p = 0$
2: for all $p \in P$ do
3:    for all $i \in p$ do
4:       $i' :=$ gv_to_lv$(p, i)$
5:       for all $j \in A[i]$ do
6:          if $j \in p$ then
7:             $j' :=$ gv_to_lv$(p, j)$
8:             $A_p[i', j'] := A[i, j]$
9:          end if
10:      end for
11:   end for
12: end for
13: return $A_p$
```

Algorithm 3 fold(p, A, x, b)

```
1: for all $i \in p$ do
2:    $i' :=$ gv_to_lv$(p, i)$
3:    $b_p[i'] := b[i]$
4:    $x_p[i'] := x[i]$
5:    for all $j \in A[i]$ do
6:       if $j \notin p$ then
7:          $b_p[i'] := b_p[i'] - x[j]A[i, j]$
8:       end if
9:    end for
10: end for
11: return $x_p, b_p$
```

are then extracted back into the global r and x vectors. The folding and extracting subroutines are detailed in Algorithms 3 and 4, respectively.

Once x and r have changed, three important bookkeeping steps must be executed. First, in lines 9-12, the algorithm updates ξ_G. Second, in lines 14-22, the algorithm updates the residual of any variables that depended upon any variable in p. Naturally, as the residual of variables change, ξ_G must be updated, and the local partition error estimate ξ_p must be updated. Finally, in lines 24-28, each partition that depends upon variables in p is reprioritized.

IV. 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.

To create PGPS(λ), we adopted an architecture with some asynchronous aspects as well as some synchronous aspects. The basic algorithm is shown in Figure 2, and 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.

Algorithm 4 `extract`(p, x, x_p, r, r_p)

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

A. *Variable Communication*

In the PGPS(λ) 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." Given this assignment, the PGPS(λ) algorithm embodies the same core concepts as the serial version: the local partition p with the highest priority is selected and solved. 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 now communicate the values in p to all other processors which depend upon p, and it must receive 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. This is exactly the set `ProcDep`(p) (as defined in Section II), and is typically very small in sparse matrices.

Communicating new values can be accomplished with a single, simple 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.

B. *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.

Initialization

1) Partition the variables
2) Assign partitions to processors

Repeat

1) Select a subset of local variables p
2) Solve the variables in p using λ, while holding the rest of the variables in the system constant
3) Recompute the residual of any local variables that depend on variables in p, as well as the global error estimate
4) Communicate the values of variables in p to other processors
5) Periodically synchronize error estimates

Until stopping criteria are met

Fig. 2. Basic steps in the PGPS(λ) algorithm.

C. *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.

D. *Full Algorithm*

Algorithm 5 shows the full PGPS(λ) algorithm. Note that processors now operate only on local partitions. The priority queue 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 error estimates and residuals are recomputed.

Lines 2-6 show pseudocode for the simple synchronous termination test. Lines 24-49 constitute the bulk of the difference between PGPS(λ) and GPS(λ), but is conceptually simple: each partition received from a foreign processor must be processed in the same way as a local partition.

V. 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; we leave much of it to future research. Here, we briefly consider convergence and miscellaneous design details.

Algorithm 5 PGPS(λ)

```
Initialization
 1: same as serial

Main Loop
 1: repeat
 2:   // Synchronously compute global error
 3:   if time to resync then
 4:     ensure all messages have been received
 5:     $\xi_G = \texttt{Reduce}(\xi_{Pr})$
 6:   end if
 7:
 8:   // Solve the highest priority partition
 9:   same as serial
10:
11:   // Update the global residual
12:   same as serial
13:
14:   // Update partition and variable residuals
15:   for all $i \in \texttt{LocalVarDep}(p)$ do
16:     same as serial
17:   end for
18:
19:   // Reprioritize partitions
20:   for all $p' \in \texttt{LocalPartDep}(p)$ do
21:     same as serial
22:   end for
23:
24:   // Send new partition values
25:   for all $i \in \texttt{ProcDep}(p)$ do
26:     $\texttt{Send}(i, p, x_p)$
27:   end for
28:
29:   // Receive new values
30:   for $i = 1$ to $nproc$ do
31:     if no messages waiting then
32:       break
33:     end if
34:     $(p', x_{p'}) = \texttt{Recv}()$
35:     // Update partition and variable residuals
36:     for all $i \in \texttt{LocalVarDep}(p')$ do
37:       $p'' := \texttt{var\_to\_part}(i)$
38:       $\xi_{p''} := \xi_{p''} - r[i]^2$
39:       $\xi_{Pr} := \xi_{Pr} - r[i]^2$
40:       $r[i] := b[i] - < A[i,*]^T, x_i >$
41:       $\xi_{p''} := \xi_{p''} + r[i]^2$
42:       $\xi_{Pr} := \xi_{Pr} + r[i]^2$
43:     end for
44:
45:     // Reprioritize partitions
46:     for all $p'' \in \texttt{LocalPartDep}(p')$ do
47:       $\texttt{PQ.reprioritize}(p'')$
48:     end for
49:   end for
50: until $\xi_G < \epsilon^2$
```

A. 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 SPD. Second, it seems that GPS(λ) should impose the additional constraints that asynchronous relaxations impose.

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, each of which was non-convergent. 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.

With a subsolver like Richardson iteration, it is possible to state more precisely the conditions under which GPS(λ) will converge. It is well known that chaotic linear relaxations of the form $x_{t+1} = f(x_t)$ converge when f satisfies the definitions of a contraction mapping [3]. Proofs of contraction have been constructed for several important cases, including linear relaxations. If iterations are of the form $x_{t+1} = Ax_t + b$, then the system is guaranteed to converge if $\rho(A) < 1$. However, Richardson iteration actually executes iterations of the form $x_{t+1} = x_t + r_t = x_t + b - Ax_t$, which is equivalent to $x_{t+1} = (-A + I)x_t + b$, implying that the algorithm will converge if $\rho(-A + I) < 1$. Since $\rho(A) < \|A\|$ for any matrix norm, and since the infinity norm of a matrix is simply the largest row-sum of a matrix, it is sometimes possible to scale A to make it a convergent matrix. Instead of solving $Ax = b$, we can solve $cAx = cb$, or even scale each row individually. However, as demonstrated for the case of Richardson iteration,

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

Fig. 3. Descriptions of the matrices used.

it is not A that must have $\rho < 1$, but rather $-A + I$. Scaling factors which ensure this condition can only be found if A is strongly diagonally dominant. Note that these are exactly the conditions described by [4], except that they did not have a corresponding derivation or compact notational description.

The rate of convergence of GPS(λ) in general is difficult to quantify. Since it is technically a chaotic relaxation algorithm, all known convergence proofs simply state that if all variables are relaxed an infinite number of times, the system will converge. No guaranteed rates are known, but empirically, we have observed excellent rates.

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

B. Miscellaneous Details

It is important to ensure that there is consistency between the priority metric and the specified subdomain tolerance. For example, if GPS(λ) uses the residual 2-norm with a given tolerance as the global stopping criteria, and a residual 2-norm stopping criteria for the subsolver, but a maxnorm priority metric, it is possible for situations to arise where all of the subproblems are below the specified tolerance (so that the subsolver doesn't do any more work), but the combination of all subproblems together is higher than the global tolerance (meaning that the GPS(λ) wrapper won't ever terminate).

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.

VI. Experimental Setup

A number of experiments were run to to isolate and quantify all relevant behaviors of the GPS(λ) and PGPS(λ) algorithms. Two major sets of experiments were run.

The first set of experiments was designed to quantify the relative benefits of the various enhancements we have discussed. Specifically, it sought to quantify: 1) the general benefits of the folding / extracting technique; 2) the general benefits of prioritization; and 3) the efficiency of the parallel version of the code. This set of experiments also attempted to quantify these benefits for several different subsolvers. To be fair, we also point out cases in which the algorithm fails, or yields worse performance than a baseline.

The second set of experiments was designed to quantify the performance of the algorithms as different parameters were tuned. Here, the experimental matrix explored 1) varying the number of sub-domain iterations; 2) varying the tolerance requested; 3) varying the number of partitions; and 4) varying subsolver-specific parameters (such as the restart parameter to GMRES).

Experiments were run across a wide variety of matrices, which are described in Figure 3. An effort was made to select matrices from different discplines, with different sizes, 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 3.

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

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) often 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, the bulk 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 variety of 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 3 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 in order to function. 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 Myrinet interconnects. All code was written in C, using MPI to create the parallel algorithms.

VII. Results

The results of our experiments were very positive. We begin by presenting the general serial results, followed by the parallel scalability results. We then present results which explore the performance of GPS(λ) for a number of different subsolvers. We also present some results exploring the many parameters involved, and finish with some miscellaneous results.

The primary criterion measuring success is the wall-clock time needed to reduce the 2-norm of the residual to a specific tolerance (note that this is not the preconditioned 2-norm, nor is it the approximate 2-norm as reported by GMRES). Some consideration was given to the fact that many times an algorithm reduces the residual extremely quickly and stabilizes it at a certain point. This was considered to be superior behavior to an algorithm which can eventually reduce the tolerance more, but only after a very long time.

A. Serial results

Figures 5, 6, 7, 8, 9 and 10 show 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 (sometimes by many orders of magnitude) [bcsstk08, e20r0000, fidap006, fidap014, sap-160000, sherman2]
- GPS(λ) slightly improves performance, yields the same performance, or slightly worsens 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 (Figure 9) and sap-160000 (Figure 5) problems, for instance, GPS(λ) solves it instantly while other algorithms gradually reduce the residual norm. For the e20r0000 (Figure 6) 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. 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.

GPS(λ) is not a panacea, of course. We tested many hard problems for which it did not seem to matter which algorithm was used. No algorithm seemed capable of solving the e40r5000 problem, for instance. 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 (Figure 7) problem was an interesting case: P-GMRES and GPS-PQ flip-flopped twice, although GPS-PQ eventually won out. For the mcar-160000 (Figure 8), GPS-PQ almost instantly solves it, but the GPS-SEQ also solves it almost instantly, and to a better tolerance. P-GMRES eventually drives the the residual norm lower than either, but only after about 32 seconds. 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 obvious from a graph like this that 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. Such an algorithm is left for future research.

We also observed situations where GPS(λ) degraded performance substantially. 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. Often, it was the case that although GPS(λ) could reduce the tolerance reliably, other algorithms managed to solve a problem instantly. This happened on the fidapm33 and rdb2048l problems (with P-GMRES as the winner). Often, it was the case that although GPS(λ) could reduce the tolerance reliably, other algorithms managed to solve a problem instantly. This happened on the fidapm33 and rdb2048l problems (with P-GMRES as the winner), and the sherman3 problem (with P-GMRES+ASM/ILU reducing to a residual norm of 1e-08 in about 11 seconds, and GPS-PQ reducing to a residual norm of 26 in 11 seconds). And sometimes, as on the s3dkt3m2 matrix (Figure 10) and the hp_400_5, GPS(λ) drives the residual norm down quickly and reliably, but another algorithm manages to achieve a lower norm even more quickly.

Overall, we consider the results very positive: the GPS(λ) algorithm was originally designed to solve reinforcement learning problems efficiently, so it is expected that it performs best on them. It is pleasantly surprising to see that it also performs so well on so many other types of problems, and to see that even in situations where GPS(λ) was not the absolute winner, it often produced respectable results.

B. Parallel Results

Figures 11, 12, 13 and 14 show parallel results of PGPS(λ). The first two show the wallclock time of P-GMRES, P-GMRES/ASM+ILU, PGPS-PQ and PGPS-SEQ, while the second two 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

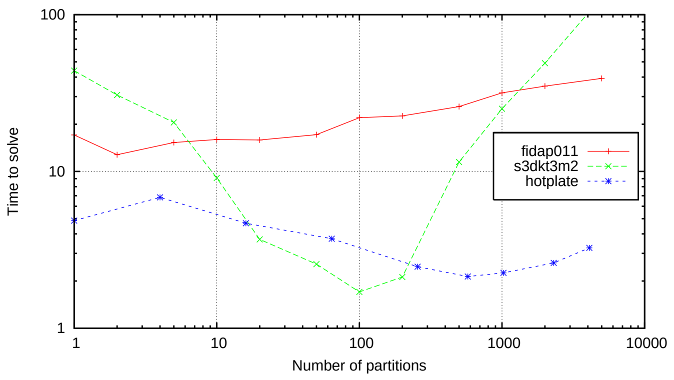

Fig. 4. Performance of PGPS(λ) vs. the number of partitions used.

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. Rather, each algorithm/problem combination should be viewed by itself. 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, which are summarized in the following table:

Algorithm	s3dkt3m2	hp_400_5
P-GMRES	2.4	0.4
P-GMRES/ASM+ILU	1.2e-8	4.6
GPS-PQ	1.0e-7	3.5
GPS-SEQ	0.09	13

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 prohibitive 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 a relatively un-optimized piece of prototype code [PGPS(λ)] exhibits similar scalability to a well-developed library relied on by many researchers [PETSc]), but also because it was easier to implement. The division of a problem into partitions is a natural domain decomposition, which is at once simpler than a functional decomposition and appears to incur less overhead.

C. 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). In fact, the divergence always seemed to be a function of the number of partitions used: if a particular number resulted in divergence, a slightly higher or lower number usually did not (we hasten to add that for the vast majority of problems, any number of partitions worked). Considering the highly asynchronous nature of the algorithm, and the discussion of convergence in Section V-A, this result is surprising. Certainly, it indicates that the algorithm is not limited to convergent matrices; this motivates future research into why convergence can be achieved for so many matrices under conditions that are very different than those specified in traditional chaotic convergence proofs.

We also note that tuning the parameters of the algorithm is somewhat difficult. For example, Figure 4 shows performance of PGPS(λ) on three matrices, as a function of the number of partitions used. Unfortunately, each matrix shows a different optimal number. Currently, we can only offer two rules of thumb for selecting this number: the number of cross-partition dependencies must be minimized, and, the size of each partition should be kept to between 100 and 400.

VIII. 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.

There are many possibilities for future research. Perhaps one of the strongest directions for future research is to develop an explicitly preconditioned version of GPS(λ). Even though the subsolver may precondition a subproblem, GPS(λ) currently operates directly on A at a global level. Having such a preconditioned version would make the algorithm more directly comparable to other algorithms, and would almost certainly improve performance futher. There is also the strong possibility that GPS(λ) will accelerate multi-grid methods, by accelerating the solution to each level in the grid hierarchy. As noted, it is also theoretically possible to use different subsolvers on different subproblems, which could help solve problems that are (for example) “almost” semi-positive definite.

There are also many avenues for more theoretical research. The issue of convergence is still intriguing, and merits further study. It would be also be interesting to determine structural features of matrices that indicate when extreme benefits of applying GPS(λ) can be expected. Methods of auto-tuning parameters, or at least of selecting them in a principled way, would also be a boon. As mentioned several times, using the residual as a priority measure is a rough approximation of the true utility of solving partitions. More sophisticated estimates of this utility could potentially improve the algorithm further.

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.

References

[1] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong 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. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[3] Dimitri P. Bertsekas and John 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] Vijaykumar Gullapalli and Andrew G. Barto. Convergence of indirect adaptive asynchronous value iteration algorithms. In *Advances in Neural Information Processing Systems*, volume 6, pages 695–702, 1994.

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

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

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

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

[12] David Wingate and Kevin Seppi. P3VI: A partitioned, prioritized, parallel value iterator. In *International Conference on Machine Learning*, to appear 2004.

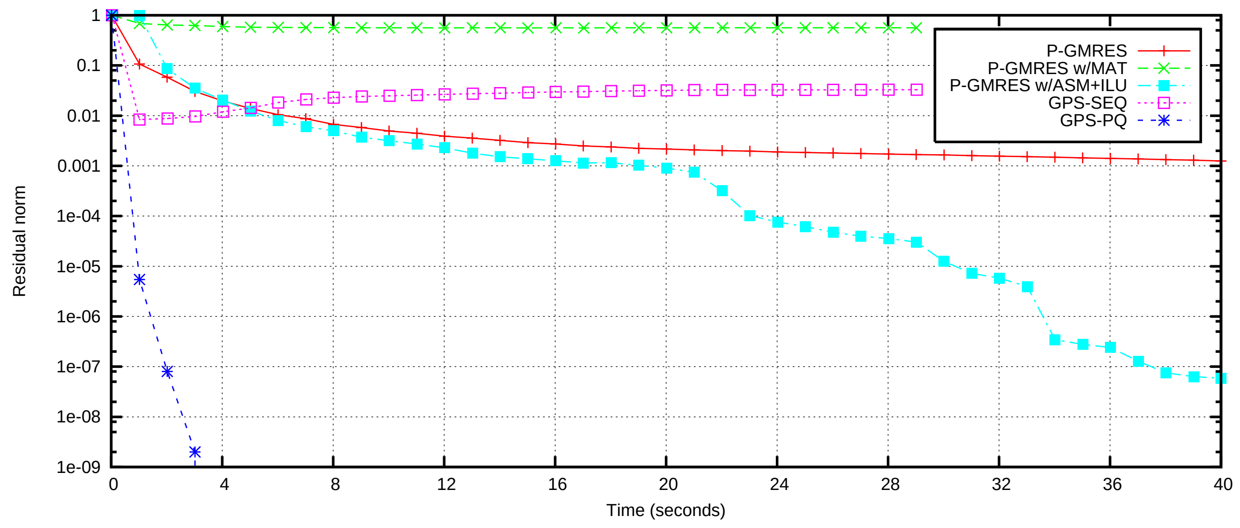

Fig. 5. Performance results on the SAP-160000 matrix.

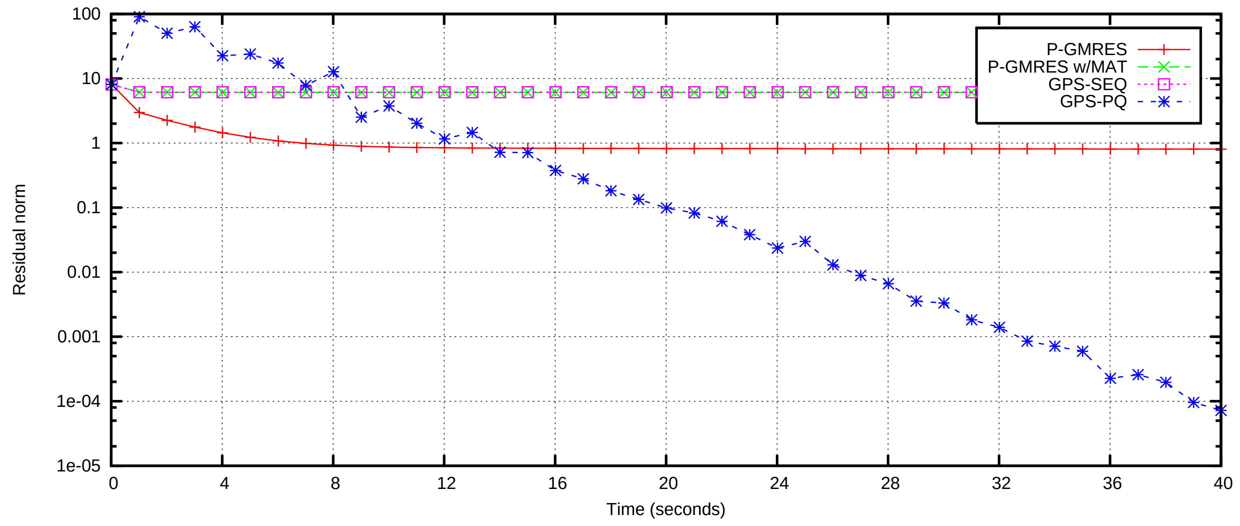

Fig. 6. Performance results on the e20r0000 matrix. P-GMRES w/ASM+ILU diverged on this matrix.

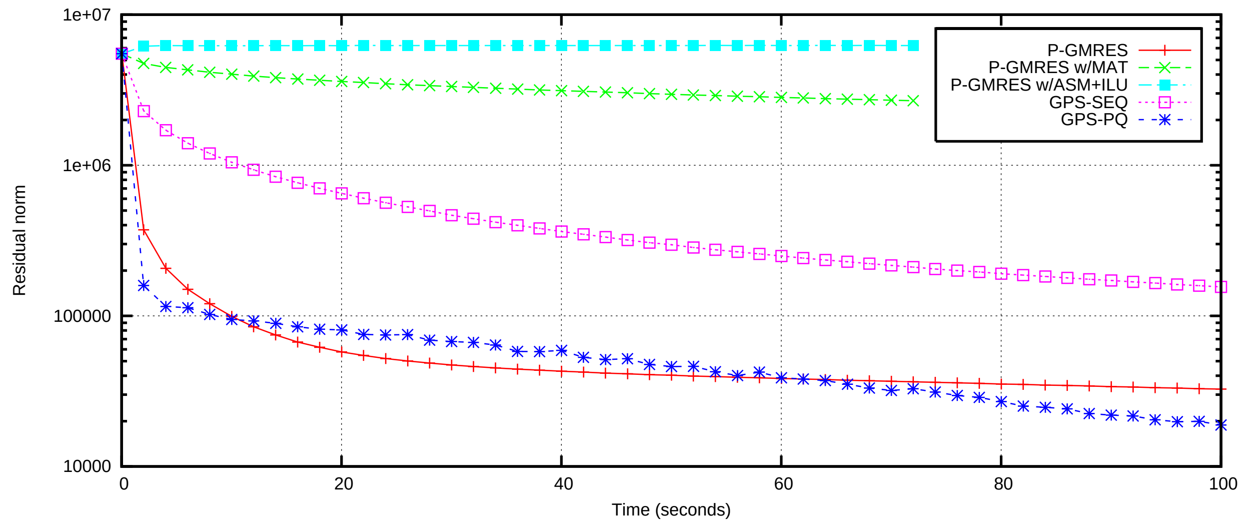

Fig. 7. Performance results on the fi dap011 matrix.

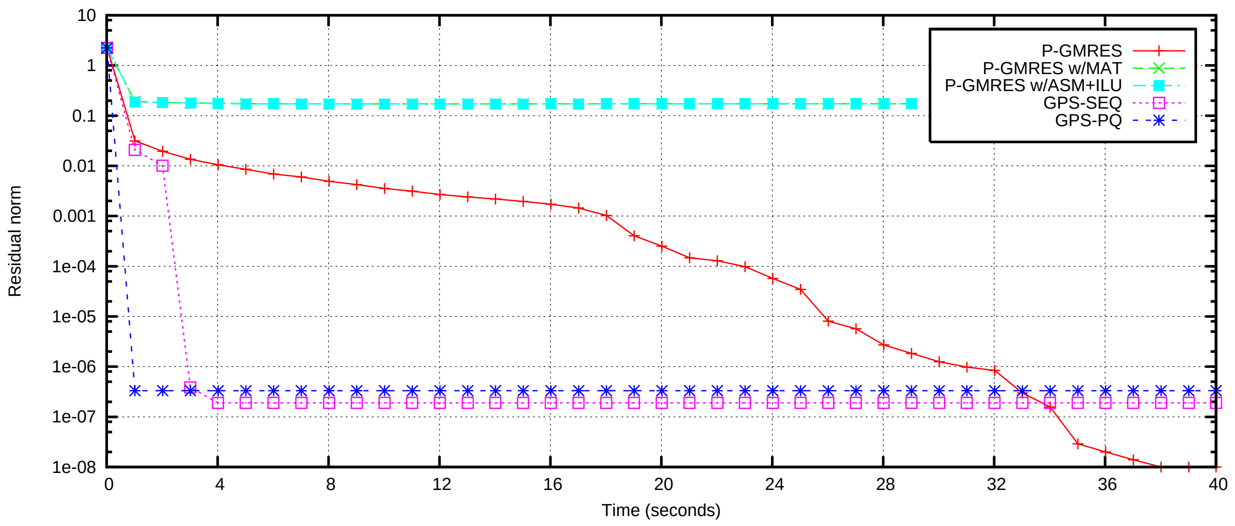

Fig. 8. Performance results on the MCAR-160000 matrix.

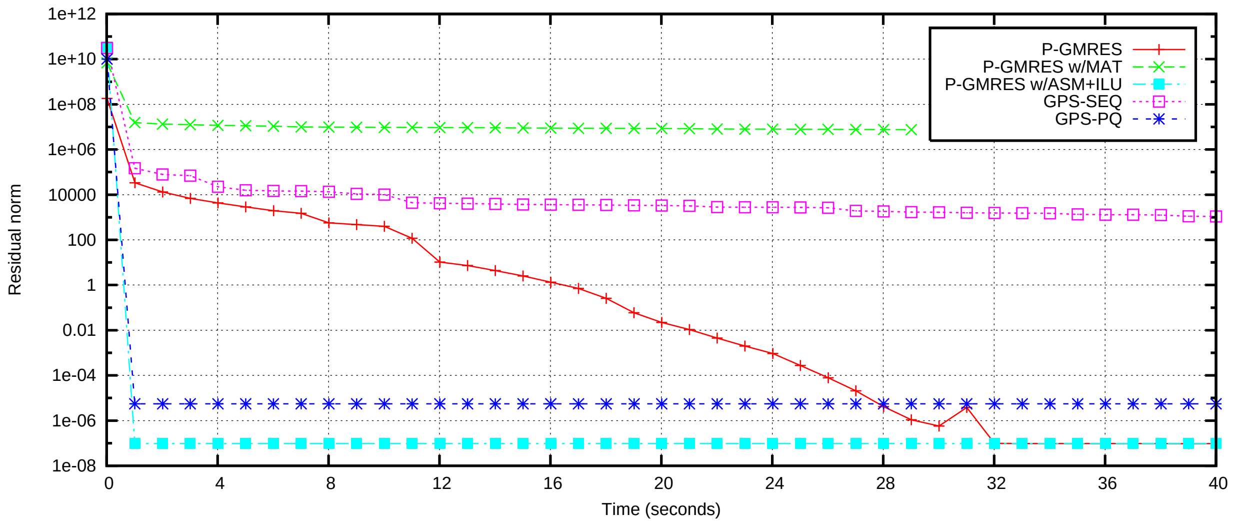

Fig. 9. Performance results on the bcsstk08 matrix.

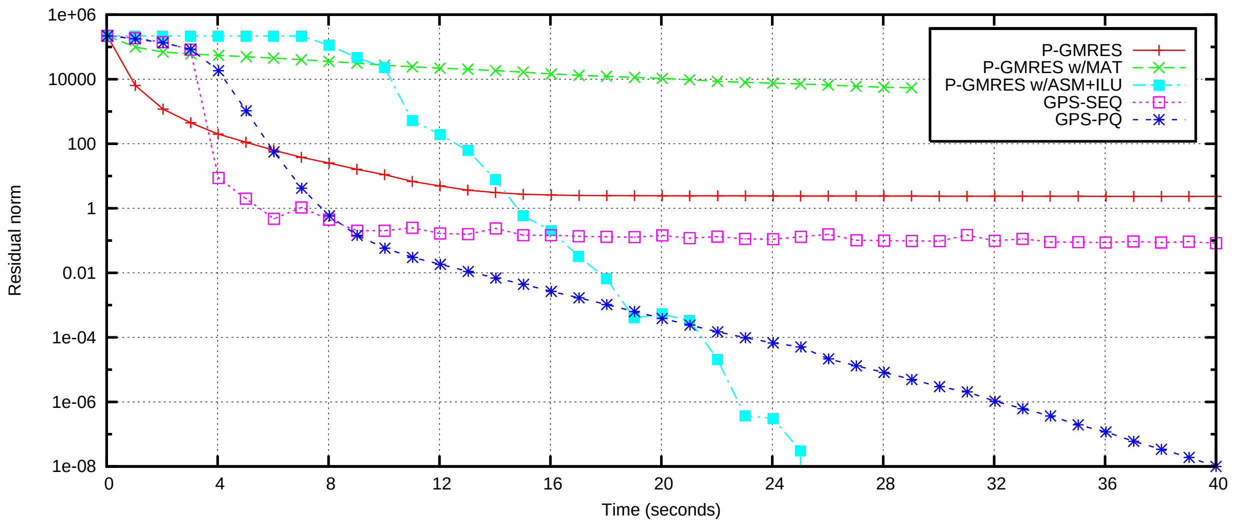

Fig. 10. Performance results on the s3dkt3m2 matrix.

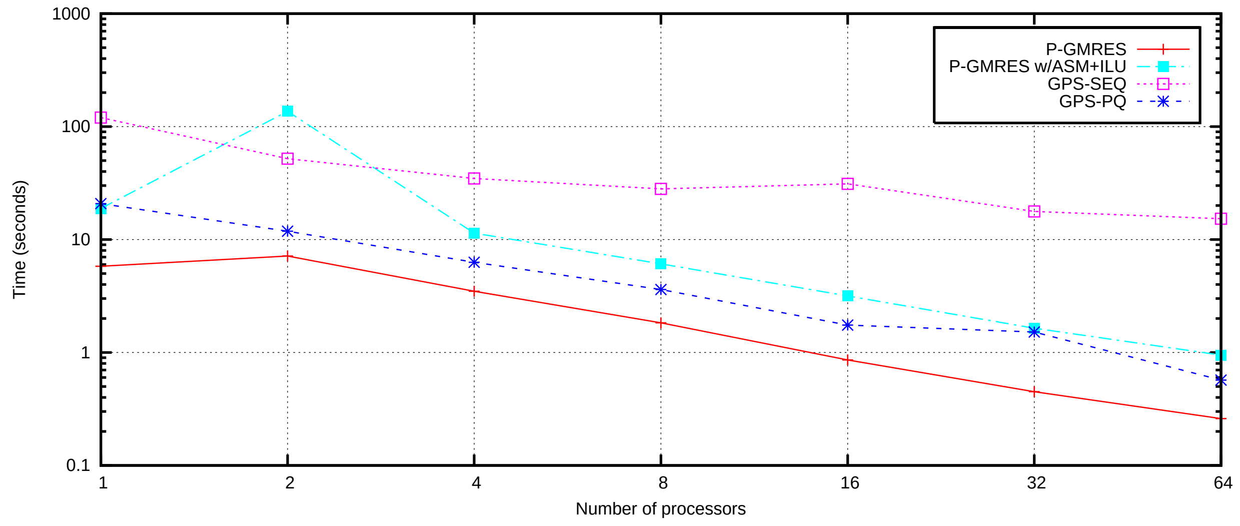

Fig. 11. Parallel performance results on the s3dkt3m2 matrix. As noted in the text, different algorithms should not be compared directly to each other; rather, the graph is meant to show only the scalability of each algorithm individually.

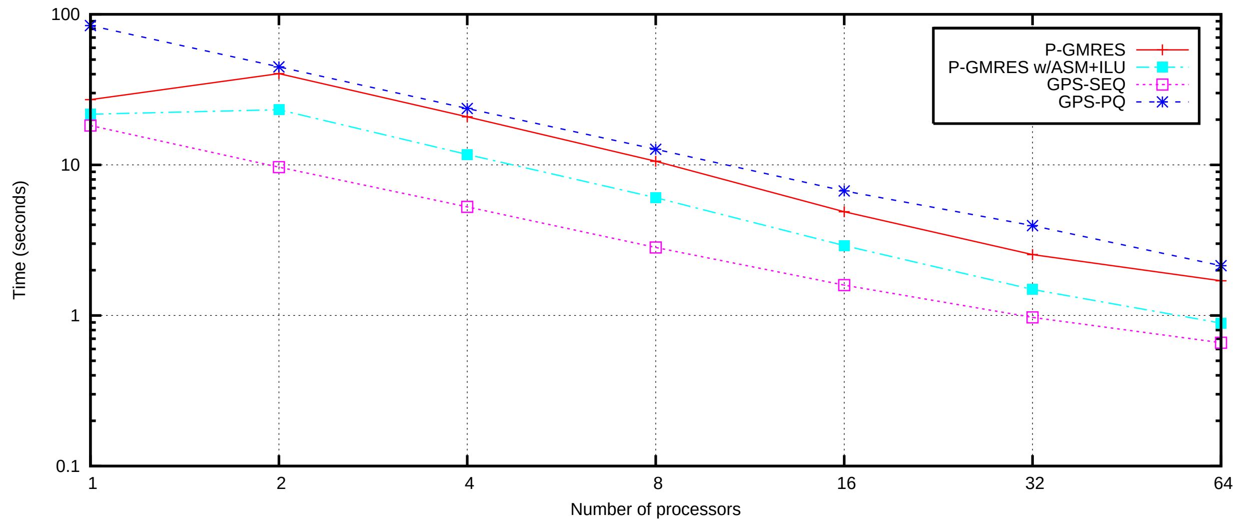

Fig. 12. Parallel performance results on the hp_400_5 matrix. As noted in the text, different algorithms should not be compared directly to each other; rather, the graph is meant to show only the scalability of each algorithm individually.

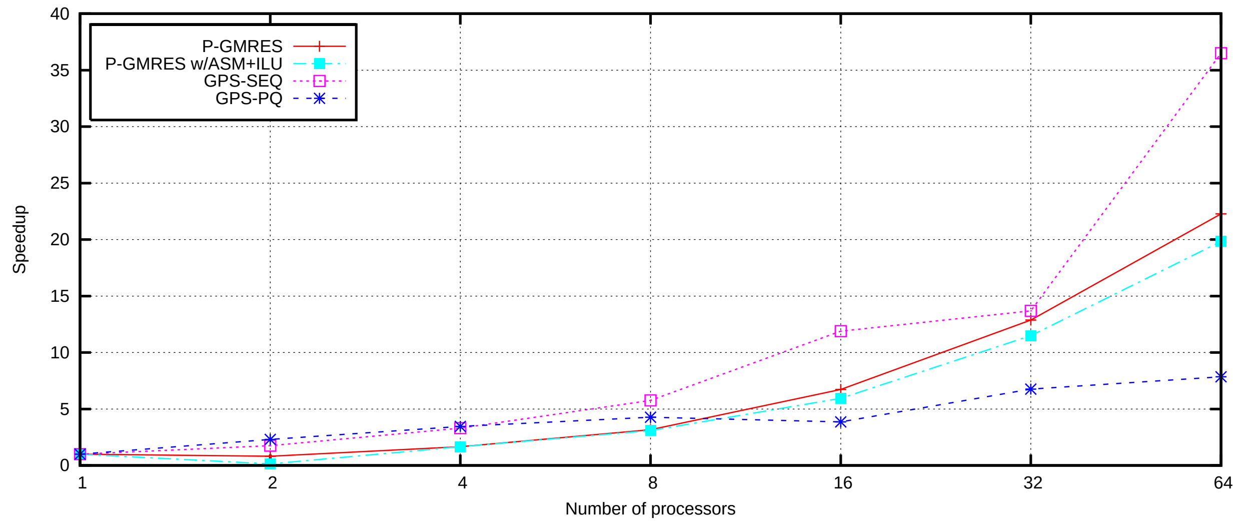

Fig. 13. Parallel speedup results on the s3dkt3m2 matrix. As noted in the text, different algorithms should not be compared directly to each other; rather, the graph is meant to show only the scalability of each algorithm individually.

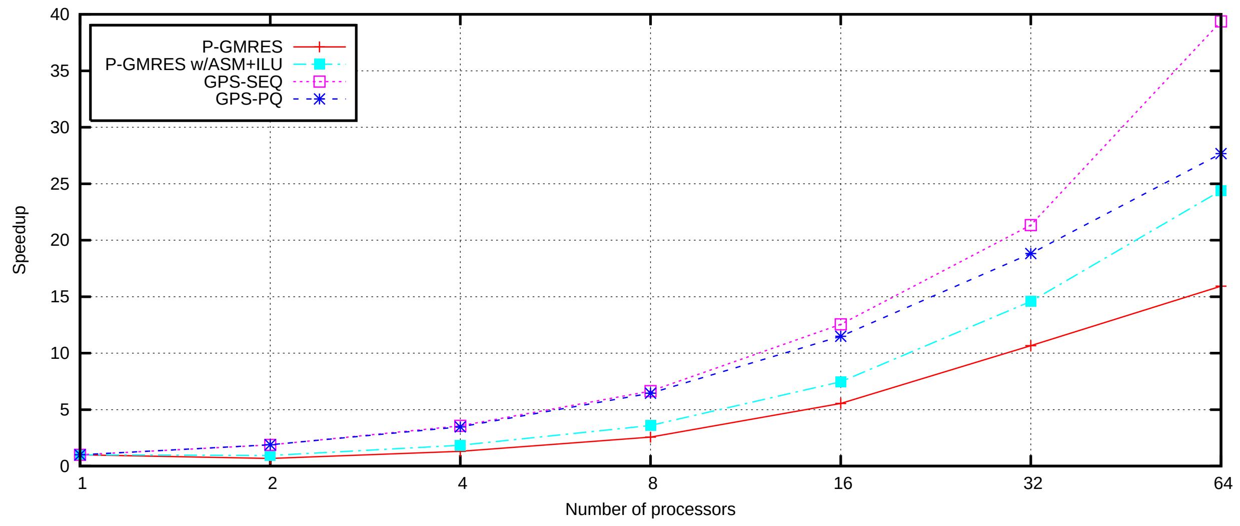

Fig. 14. Parallel speedup results on the hp_400_5 matrix. As noted in the text, different algorithms should not be compared directly to each other; rather, the graph is meant to show only the scalability of each algorithm individually.