Design of a library of parallel preconditioners

Tony Chan and Victor Eijkhout*

February 18, 1998

Abstract

We outline the design principles underlying the ParPre library of parallel preconditioners. ParPre is a message passing library of distributed preconditioners for linear systems, written using MPI and Petsc. It comprises Schwarz methods, Schur system domain decompositioning, various parallel incomplete factorisations, and multilevel methods.

1 Introduction

Many physical applications give rise to the solution of large, sparse, linear systems $Ax = b$. It has long been recognised that for the iterative solution of such systems, the choice of a high-quality preconditioner is crucial. While a large body of earlier literature exists for sequential preconditioners, in the context of parallel computing many questions are still open.

ParPre is a library for largely black-box generation of distributed memory parallel preconditioners, implemented using the MPI and Petsc libraries. In this paper, we describe the design issues resulting from our choice for black-box methods, and from the requirement of execution on message passing parallel computers.

Although preconditioners are typically used to solve linear systems deriving from partial differential equations, that is, systems from differential operators on some physical domain, we take a black-box approach and disregard all knowledge of the domain and the operator. The ParPre preconditioners are then completely derived from the linear system, and the connectivity information from the matrix graph.

There are two sides to this decision. On the one hand, with this approach, we can not expect to obtain the same performance as from preconditioners constructed with full knowledge of the continuous problem. On the other hand, a purely algebraic method will have a wider range of applicability, and we can

*University of California, Los Angeles 90095. This research was supported by the ARO under contract DAAL-03-9-C-0047 (Univ. of Tenn. subcontract ORA 4466.04, Amendment 1 and 2).

still expect to approach the performance of special purpose methods on systems that are actually from PDEs.

Constructing parallel preconditioners is different from constructing sequential ones, starting from the very choice of methods. Traditionally, the efficiency of preconditioners was measured in terms of the number of operations of the preconditioned iterative method, usually as a function of the mesh width parameter h. In the context of the parallel solution of PDEs, this measure is insufficient. Many classical methods such as SOR or ILU preconditioning are not very parallel in the natural matrix ordering, and orderings with more parallelism usually lead to slower converging iterative methods [4, 5, 7].

One then has the choice between taking these classical, typically purely algebraic, methods and finding some mean between increasing parallel and decreasing scalar performance, or taking a step back and considering methods that exhibit natural parallelism. In the ParPre package [6], we have largely taken the latter approach.

In ParPre, we are able to reproduce some of the classical, more sequential, preconditioners, but the emphasis of the package is on two classes of naturally parallel methods:

1. Domain decomposition methods, and
2. Multilevel methods.

The first class, comprising classical block methods, (overlapping) Schwarz methods, and Schur complement methods derives parallelism from the independence of (subsets of) subdomains. The two issues to consider there are the solution method on the subdomains, and any sequentiality between the subdomains.

The second class, consisting of multi-colour factorisation methods and various algebraic multigrid methods, derives its parallelism from the essentially local nature of most operations in multilevel methods. The main problem in these methods is the algorithm for generating, in parallel, successive grids in the absence of geometric information. As a byproduct, these methods are able to generate, algebraically, coarse grids for a given grid.

These two classes are described in detail in sections 2 and 3. A final section is devoted to some practical aspects of the ParPre packages.

2 Subdomain methods

Several succesful parallel preconditioners are based on a partitioning of the physical domain into (connected) subdomains.

For an elliptic partial differential equation, a subproblem on a connected subdomain is again an elliptic PDE, usually with different boundary conditions. Thus, splitting the physical domain, and correspondingly the matrix and vectors of the problem, leads to a parallel formulation with each process solving an elliptic problem.

Since we are working purely in an algebraic context, we do not partition the domain, but instead the matrix and the vectors. A subdomain is then simply a set of unknowns, and we hope that the numbering of the global problem is such that the subdomains will resemble physical subdomains if the problem derives from a PDE. Some remarks about this are in subsection 2.1.

After the partitioning in subdomains, we are faced with two choices:

1. The solution method on the subdomain;

2. The connection scheme between the subdomains.

These two issues are discussed in detail in subsections 2.2 and 2.4.

We will first devote some attention to the problem of the partitioning strategy.

2.1 Partitioning strategy

The way the problem variables are divided over processors has serious consequences for the performance of the iterative solution method. For instance, a preconditioner based on analytical properties of the problem, such as a Schur complement method, will not fare well if the subsets of variables do not correspond to (simply) connected subdomains.

Even if the subsets of variables correspond to physical subdomains, there are issues such as load balancing, that influence performance.

However, in our case of a preconditioner library we need not pay explicit attention to these matters. Presumably, splitting and load balancing algorithms have done their work before the iterative method ever started. A preconditioner library, then, simply has to accept the ordering and splitting imposed by the surrounding code.

We will make the working assumption that the ordering as it is presented to the library is 'reasonable': subsets of variables are presumed to correspond to physical subdomains, and of approximately equal size. No redistribution will take place during preconditioner creation or application.

2.2 Subdomain solvers

Since each subdomain is handled by a single processor, we don't have to worry about parallelism in the subdomain solves. Any traditional, sequential, preconditioner can be used as a subdomain solver, including exact factorisations.

In ParPre, the code to specify a subdomain solver looks as follows:

```
PC the_pc; /* the parallel preconditioner */
{
  PC local_pc; /* a pointer to the local solver */
  PCParallelGetLocalPC(the_pc,&local_pc);
```

```
  /* example: set the local solve to full LU solve */
  PCSetType(local_pc,PCLU);
}
```

Detailed explanation of these calls can be found in [6].

The boundary for the subdomain solves is determined by the connection scheme of the subdomains. This will be discussed in section 2.4.

2.3 Subdomain sequences

Of the methods to follow, the Schur complement and Additive Schwarz methods process their subdomains in parallel. The Block SSOR and Multiplicative Schwarz methods process their subdomains in sequence. On the one hand, such sequentiality will make the methods less parallel, on the other hand, it is expected to reduce the number of iterations.

The amount of sequentiality is under user control through the following routine:

```
int PCParallelSubdomainPipelineSetType
     (PC pc,PipelineType pipe_type)
```

where `PipelineType` is one of the following:

PIPELINE_NONE Process all subdomains in parallel.

PIPELINE_SEQUENTIAL Process the subdomains in increasing numbering: $0, 1, \ldots$ This does not imply that the sequential time for the algorithm equals the number of processors P: for grid-connected processors we expect $P^{1/d}$ where d is the number of space dimensions of the problem.

PIPELINE_REDBLACK Process the subdomains in an odd-even manner.

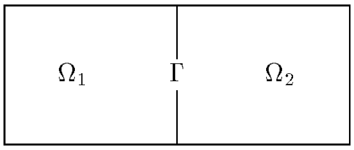
PIPELINE_MULTICOLOUR Apply a greedy colouring algorithm (this is redundantly done on each processor) to find a multi-colouring of the processor connectivity graph.

2.4 Subdomain connections

We distinguish three classes of methods, based on their different subdomain connection schemes, which correspond to the amount of overlap between the subdomains.

2.4.1 Schur complement system

If the subdomains are separated by an interface, the following matrix structure ensues. For the case of two subdomains Ω_1 and Ω_2, separated by a strip Γ, presumably of width $O(h)$:



we obtain a matrix structure:

$$\begin{pmatrix} A_{11} & \emptyset & A_{13} \\ \emptyset & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

Since a factorisation takes the form:

$$\begin{pmatrix} I & & \\ \emptyset & I & \\ A_{31}A_{11}^{-1} & A_{32}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & & \\ & A_{22} & \\ & & S \end{pmatrix} \begin{pmatrix} I & \emptyset & A_{11}^{-1}A_{13} \\ & I & A_{22}^{-1}A_{23} \\ & & I \end{pmatrix}$$

where $S = A_{33} - A_{31}A_{11}^{-1}A_{31} - A_{32}A_{22}^{-1}A_{23}$, we get the following solution scheme.

1. Solve all A_{ii} subdomain systems in parallel.
2. Use the A_{3i} blocks to establish boundary conditions for the S system.
3. Solve the S system.
4. Use the A_{i3} blocks to establish boundary conditions for the subdomain systems.
5. Solve the subdomain systems in parallel.

The interface system is again a distributed matrix, to be solved by any parallel method in ParPre (or in Petsc). We will add an option for the interface system to be collected and solved redundantly on each processor.

2.4.2 Additive Schwarz and Block Jacobi

If we define our method by solving the subdomains in parallel and constructing the solution by adding the subdomain solutions together, we obtain the Block Jacobi method if the union of subdomains is a disjoint splitting, or the Additive Schwarz method for a non-disjoint, overlapping, splitting.

Formally, let $\{\Omega_i\}_i$ be a partitioning of the domain, and let I_i be the index set of Ω_i, with size N_i. Let R_i be the restriction operator

$$R_i \colon \mathbf{R}^N \to \mathbf{R}^{N_i}, \qquad (R_i x)(j) = x_{I_i(j)},$$

then the subdomain coefficient matrices are given by

$$M_i = R_i A R_i^t,$$

and the preconditioner is multiplicatively defined as

$$M^{-1} = \sum_i M_i^{-1}.$$

2.4.3 Multiplicative Schwarz and Generalised Block SSOR

Solving subdomains in sequence, and using boundary conditions established by earlier subdomain solves in the sequence, leads to a block version of the Generalised SSOR [1] method for a disjoint splitting in subdomains, and the Multiplicative Schwarz method for an overlapping splitting.

For the case of p subdomains, in the n-th iteration we compute partial iterates:

$$k = 1, \ldots, p\colon x_{n+k/p} = x_{n+(k-1)/p} - M_k^{-1} r_{n+(k-1)/p}.$$

Defining splittings as $A = M_i - N_i$, we find for the total preconditioner:

$$M^{-1} = M_p^{-1}[I + N_p M_{p-1}^{-1}[\cdots[I + N_2 M_1^{-1}]\cdots]].$$

Defining the complementary restriction operator $\bar{R}_i$ to $\bar{I}_i$, and defining off-diagonal blocks as $B_i = R_i A \bar{R}_i^t$, we can give the following algorithmic formulation for solving $My = f$:

- Set $y \leftarrow 0$.
- For $i = 1, \ldots, p$:
  1. Let $g = B_i y$; this corresponds to evaluating the boundary conditions around domain Ω_i, and has to be done in practice only for the lower numbered subdomains surrounding Ω_i, in particular for $i = 1$ this step can be omitted.
  2. Solve $y_i = M_i^{-1}(f - g)$.
  3. Update $y|_{I_i} \leftarrow y_i$.
- For symmetry, execute the above loop again, but in reverse order, that is: for $i = p, \ldots, 1$.

If the above solution scheme is applied to the original matrix, that is, $M_i = A_{ii}$, this is indeed a block SSOR method. By deriving the M_i blocks from a block factorisation we get the Generalised Block SSOR method, which potentially can be an exact factorisation. The user can switch the global factorisation on or off by the following routines:

```
int PCGenBlockSSORSetGlobalFactorisation(PC pc);
int PCGenBlockSSORSetNoGlobalFactorisation(PC pc);
```

where the default is to have no global factorisation.

The generalised block SSOR method can be made to reduce to a point ILU factorisation, by choosing the subdomain solver to be an ILU method. The block method is then equivalent to a point factorisation on the ordering where first the variables on one processor are numbered, then the ones on the next, et cetera.

On the other hand, if the matrix is block tri-diagonal and the subdomain method is taken to be an exact LU factorisation, the block SSOR method becomes an exact factorisation.

3 Multilevel methods

The class of multilevel methods in ParPre comprises several methods named 'algebraic multigrid' by their respective authors [2, 10]. It also contains a few methods that are more properly described as incomplete factorisation methods [8].

We formally define levels by: the sets L_i for $i = 1, \ldots, \ell$ are called levels in the domain if they form a disjoint partitioning of the domain. If the domain is also partitioned over processor index sets I_i for $i = 1, \ldots, p$, we assume that the level sets and the index sets satisfy

$$L_i \cap I_j \neq 0 \quad \text{for all } i,\, j.$$

(In practice we allow zero intersections for a small number of index pairs.) This means that every processor will have variables in (almost) each of the levels, so if the levels can be processed in parallel, we have a fully parallel method.

3.1 Generation of levels

In generating the levels, we will use the concept of independent sets in the matrix graph. A set of vertices S in a matrix graph G (with vertices V and edges E) is called independent, if for all $i, j \in S$ the edge $(i, j) \notin E$. Given a graph G and an independent set $S \subset V$, we then have the choice of defining a level set L:

- Let $L \equiv S$, or
- let $L \equiv V - S$.

Either choice can lead to a valid method. The first choice is typical of multicolour incomplete factorisations, the second is usually taken in multigrid-type methods.

The following routines are used in ParPre to specify these choices:

```
int AMLSetCoarseGridDependent(PC pc)
int AMLSetCoarseGridIndependent(PC pc)
```

Traditional methods for identifying independent sets are typically based on so-called 'greedy' numbering schemes. In a parallel context, these schemes may potentially have low parallelism, or lead to large numbers of synchronisation points between the processors. Therefore we use a parallel colouring heuristic [9] (we will interchangeably use the terms 'colour' and 'independent set'):

- Let each node x be assigned a random number $\alpha(x)$.
- Identify the set S of all nodes such that they have a higher random number than their neighbours:

$$S = \{x: (x, y) \in E \rightarrow \alpha(x) > \alpha(y)\}.$$

It is clear that the set S derived by this algorithm is an independent set. To construct it in parallel, each node needs to acquire knowledge of its neighbours; this can be done in one parallel exchange operation. With processors owning subsets of variables, each processor needs to exchange data only with processors having physically surrounding subsets of variables.

As in [10], we base the level sets not on the full matrix graph, but rather on the graph obtained by ignoring the weak connections. The exact definition of strong and weak connections is a heuristic, the fine-tuning of which can greatly influence the performance of the resulting method.

3.2 Multilevel factorisation methods

The above parallel colouring heuristic was intended to be used in an $ILU(0)$ type factorisation [8]. The independent sets are found by the following iteration:

- Let $G(A) = \langle V, E\rangle$ be the matrix graph, and set $V_1 \leftarrow V$, $E_1 \leftarrow E$.
- Iterate for $i = 1, \ldots$ while $V_i \neq \emptyset$:
  1. Let S_i be an independent set in $\langle V_i, E_i\rangle$.
  2. Set $V_{i+1} = V_i - S$, $E_{i+1} = E_i - \{(x, y): x \in S_i \text{ or } y \in S_i\}$.
  3. Set $L_i = S_i$.

The factorisation, eliminating the variables in each set L_i in sequence, can then be performed in parallel, taking one synchronisation step per set L_i.

We can define a parallel factorisation with fill-in by modifying the above algorithm:

- Initialise as above, but additionally define $A_1 \leftarrow A$.
- Iterate for $i = 1, \ldots$ while $V_i \neq \emptyset$:

1. Let S_i be an independent set in $\langle V_i, E_i \rangle$.
2. Let $L_i = S_i$ or $L_i = \bar{S}_i$.
3. Let A_{i+1} be (a sparse approximation to) the Schur complement from eliminating the set L_i in A_i.
4. Let $G(A_{i+1}) = \langle V_{i+1}, E_{i+1} \rangle$.

In this case, the numbering and factorisation phases are no longer performed separately.

In ParPre, the following routine is available

```
int AMLSetFillMethod(PC pc,AMLFillMethod fill_method)
```

where the method parameter is one of the following:

AMLFillNone no fill-in, this results in an SSOR type factorisation. Basically, the factorisation is only concerned with identifying the level sets.

AMLFillDiag allow fill-in on the diagonal. This gives an $ILU(0)$-type factorisation.

AMLFillStrong allow large fill-in elements. This is in effect a drop tolerance method.

AMLFillFull we accept the full fill matrix.

3.3 Two-level structure

After identifying an independent set in the matrix graph, we can partition the matrix accordingly

$$A^h = \begin{pmatrix} D & C^t \\ C & E \end{pmatrix}$$

with the independent set either corresponding to the variables in the first or second block row.

Incomplete factorisations can be formulated in this 2×2 framework as:

$$M^h_{ILU} = \begin{pmatrix} I & 0 \\ CD^{-1} & I \end{pmatrix} \begin{pmatrix} D & \\ & M^H_{ILU} \end{pmatrix} \begin{pmatrix} I & D^{-1}C^t \\ 0 & I \end{pmatrix}$$

or, for the multiplicatively defined inverse:

$$M^{h^{-1}}_{ILU} = \begin{pmatrix} I & 0 \\ -CD^{-1} & I \end{pmatrix} \begin{pmatrix} D^{-1} & \\ & M^{H^{-1}}_{ILU} \end{pmatrix} \begin{pmatrix} I & -D^{-1}C^t \\ 0 & I \end{pmatrix}. \tag{1}$$

ParPre unifies incomplete factorisations and multigrid-type methods by noticing that, traditionally, multigrid methods are defined in a very similar way. The two-level structure of multigrid-type methods can be written as: Multigrid type:

$$M^{h^{-1}}_{MG} = \begin{pmatrix} I & 0 \\ -CD^{-1} & I \end{pmatrix} \begin{pmatrix} \emptyset & \\ & M^{H^{-1}}_{MG} \end{pmatrix} \begin{pmatrix} I & -D^{-1}C^t \\ 0 & I \end{pmatrix} \tag{2}$$

Additionally, smoothers are needed.

The user can switch between using the D block or $\emptyset$ in the $(1,1)$ location by the following call:

```
int AMLSetSolutionScheme(PC pc,AMLSolveScheme scheme)
```

where the scheme parameter is one of the following:

AMLSolveILU for a Gaussian factorisation solve.

AMLSolveMG for a zero $(1,1)$ block, and with pre and post smoothers added.

3.4 Generalised W-cycle

ParPre has generalised W-cycle preconditioners, which are derived from the above V-cycle ones by replacing the $M^{H^{-1}}$ block in equations (1) and (2) by

$$Z^{-1} = (I - P(M^{H^{-1}}A^H))A^{H^{-1}} \tag{3}$$

where M stands for either M_{ILU} or M_{MG} and where P is a low degree polynomial satisfying $P(x) \in (0,1)$ for x in the domain of definition. Details on this polynomial formalism can be found in [2, 3].

4 Practical aspects

The ParPre package uses the Petsc library, and MPI. Since data structures are identical to (or inspired by) Petsc structures, ParPre is pretty much seamlessly integrated with Petsc.

In order to use ParPre with an arbitrary code, we have to focus on two issues: preconditioner creation and application. We will only discuss these issues in global terms; details can be found in [6].

4.1 Preconditioner creation

The basis for forming a preconditioner is a coefficient matrix. Given that ParPre is based on Petsc, it will not be a surprise that the matrix has to be in Petsc storage format. Unless one already uses Petsc, this will lead to approximately a doubling of the storage requirement, at least during the preconditioner setup. After that, the Petsc format matrix can be released again.

4.2 Preconditioner application

To apply a ParPre preconditioner, both the input and the output vectors need to be in Petsc format. However, unlike in the above case of the preconditioner creation, this does not impose any undue demands.

Since Petsc vectors store the vector values as contiguous arrays of values, a Petsc vector can be formed at essentially no cost from any pointer to such an array. This means that in the iterative method loop converting vectors to Petsc format is basically for free.

5 Conclusion

We have outlined the design principles of ParPre, a fully algebraic package for parallel preconditioners. The methods in ParPre are based on methods that are known to work for PDE problems, but the implementation here needs only the coefficient matrix as input.

The methods fall into two classes: subdomain methods and multilevel methods. Both are implemented largely as black-box routines, but the user can set a number of parameters, such as the sequential method used to solve subdomain systems.

References

[1] O. Axelsson. A generalized SSOR method. *BIT*, 12:443–467, 1972.

[2] O. Axelsson and P. Vassilevski. Algebraic multilevel preconditioning methods, I. *Numer. Math*, 56:157–177, 1989.

[3] Owe Axelsson and Victor Eijkhout. The nested recursive two-level factorization method for nine-point difference matrices. *SIAM J. Sci. Stat. Comput.*, 12:1373–1400, 1991.

[4] I.S. Duff and G.A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT*, 29:635–657, 1989.

[5] Victor Eijkhout. Analysis of parallel incomplete point factorizations. *Lin. Alg. Appl.*, 154–156:723–740, 1991.

[6] Victor Eijkhout and Tony Chan. ParPre: A parallel preconditioners package, reference manual for version 2.0.17. Technical Report CAM Report 97-24, UCLA, 1997.

[7] Louis A. Hageman and David M. Young. *Applied Iterative Methods*. Academic Press, New York, 1981.

[8] M.T. Jones and P.E. Plassmann. Parallel solution of unstructed, sparse systems of linear equations. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Proceedings of the Sixth SIAM conference on Parallel Processing for Scientific Computing*, pages 471–475, Philadelphia. SIAM.

[9] M.T. Jones and P.E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Stat. Comput.*, 14, 1993.

[10] J.W. Ruge and K. Stüben. Algebraic multigrid. In Stephen F. McCormick, editor, *Multigrid Methods*. SIAM, 1987. chapter 4.