pARMS: A Package for the Parallel Iterative Solution of General Large Sparse Linear Systems*
User's Guide

Yousef Saad† Masha Sosonkina‡

December 30, 2003

*This work was supported in part by NSF under grants NSF/ACI-0000443, NSF/ACI-0305120, and NSF/INT-0003274, and in part by the Minnesota Supercomputing Institute

†Department of Computer Science and Engineering, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455, saad@cs.umn.edu.

‡Ames Laboratory, Iowa State University, Ames, IA 50011, masha@scl.ameslab.gov.

Contents

1 Overview of pARMS

Viable solutions of modern computational problems that arise in science and engineering should efficiently utilize the combined power of multi-processor computer architectures and effective algorithms. For many large-scale applications, solving large sparse linear systems is the most intensive computational task. The important criteria for a suitable solver include numerical efficiency, robustness, and good parallel performance. The Parallel Algebraic Recursive Multilevel Solver (pARMS) [1] is a suite of distributed-memory iterative accelerators and preconditioners targeting the solution of general sparse linear systems. It adopts a general framework of distributed sparse matrices and relies on the solution of the resulting distributed Schur complement systems.

1.1 Distributed sparse linear systems

The framework of distributed linear systems [3, 4] provides an algebraic representation of the irregularly structured sparse linear systems arising in the Domain Decomposition methods. A typical distributed system arises, e.g., from a finite element discretization of a partial differential equation on a certain domain. To solve such systems on a distributed memory computer, it is common to partition the finite element mesh and assign a cluster of elements representing a physical sub-domain to one processor. Each processor then assembles only the local equations restricted to its assigned cluster of elements. In the case where the linear system is given algebraically, a graph containing vertices that correspond to the rows of the linear system can be partitioned. For both cases, the general assumption is that each processor holds a set of equations (rows of the global linear system) and he associated unknown variables.

Figure 1 shows a 'physical domain' viewpoint of a distributed sparse linear system, for which a vertex-based partitioning has been used without overlapping the unknowns. As is often done, we will distinguish between three types of variables: (1) Interior variables are those that are coupled only with local variables by the equations; (2) Inter-domain interface variables are those coupled with non-local (external) variables as well as local variables; and (3) External interface variables are those variables that belong to neighboring processors and are coupled with the above types of local variables. The local equations can be represented as shown in Figure 2. Note that these equations need not be contiguous in the original system. The matrix represented in Figure 2 can be viewed as a reordered version of the equations associated with a local numbering of the equations/unknowns pairs.

The rows of the matrix assigned to a certain processor have been split into two parts: a *local* matrix A_i which acts only on the local variables and an *interface* matrix X_i which acts only on the external interface variables. These external interface variables must be first received from neighboring processor(s) before a distributed matrix-vector product can be completed. Thus, each local vector of unknowns x_i $(i = 1, \ldots, p)$ is also split into two parts: the sub-vector u_i of interior variables followed by the sub-vector y_i of inter-domain interface variables. The right-hand side b_i is conformally split into the sub-vectors f_i and g_i,

$$x_i = \begin{pmatrix} u_i \\ y_i \end{pmatrix} \; ; \quad b_i = \begin{pmatrix} f_i \\ g_i \end{pmatrix} \; . \tag{1}$$

The local matrix A_i residing in processor i is block-partitioned according to this splitting, leading

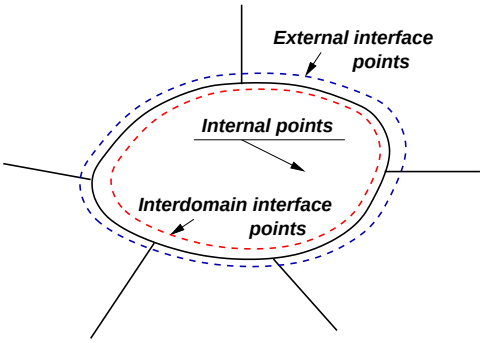


Figure 1: A local view of a distributed sparse linear system

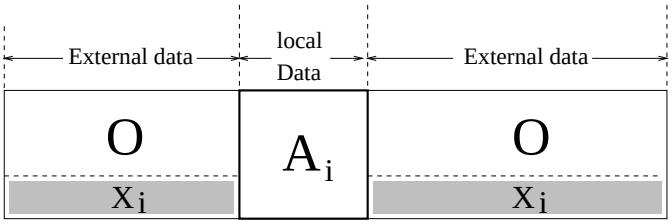


Figure 2: The corresponding local matrices A_i and X_i

to

$$A_i = \left(\begin{array}{c|c} B_i & F_i \\ \hline E_i & C_i \end{array} \right) . \tag{2}$$

With this, the equations assigned to processor i can be written as follows:

$$\begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix} \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix} . \tag{3}$$

The term $E_{ij}y_j$ is the contribution to the local equations from the neighboring sub-domain number j and N_i is the set of sub-domains that are neighbors to sub-domain i. The sum of these contributions, seen on the left side of (3), is the result of multiplying the interface matrix X_i by the external interface variables. It is clear that the result of this product will affect only the inter-domain interface variables as is indicated by the zero in the upper part of the second term on the left-hand side of (3). For practical implementations, the sub-vectors of external interface variables are grouped into one vector called $y_{i,ext}$ and the notation

$$\sum_{j \in N_i} E_{ij} y_j \equiv X_i y_{i,ext}$$

will be used to denote the contributions from external variables to the local system (3). In effect, this represents a local ordering of external variables to write these contributions in a compact matrix form. With this notation, the left-hand side of (3) becomes

$$w_i = A_i x_i + X_i y_{i,ext} . \tag{4}$$

Note that w_i is also the local part of a global matrix-vector product Ax in which x is a distributed vector which has the local vector components x_i.

1.2 Solution methods in pARMS

Multi-level Schur complement techniques available in pARMS are based on techniques which exploit block independent sets, such as those described in [6] for the sequential ARMS preconditioner.

(a) (b)

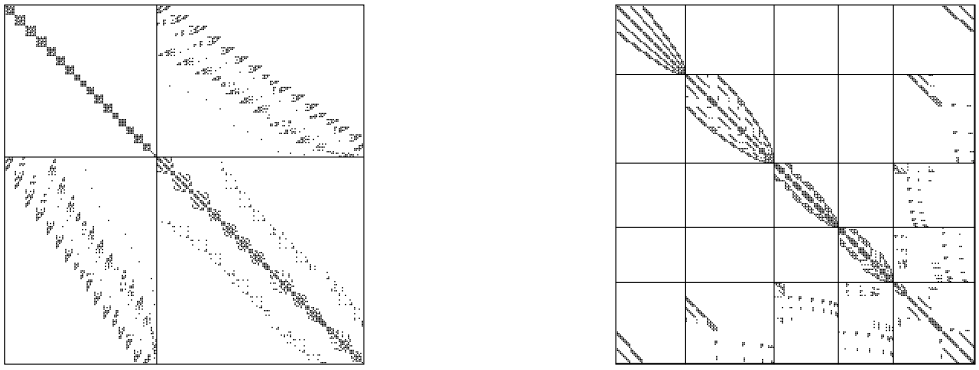


Figure 3: Group-independent set reorderings of a 9-point matrix: (a) Small groups (fine-grain), (b) large groups (coarse-grain)

1.2.1 ARMS as the building block for preconditioners in pARMS

The sequential version of ARMS is rooted in Multi-level ILU type techniques which exploit *independent sets*. An independent set is a set of unknowns that are not coupled to each other. Such orderings transform the original linear system into a system of the form

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \begin{pmatrix} u \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}, \tag{5}$$

where B is a diagonal matrix. The concept of independent sets has been generalized to *blocks* or *groups* [7]. A group-independent set is a collection of subsets (blocks) of unknowns such that there is no coupling between unknowns of any two different groups (blocks). Unknowns within the same group (block) may be coupled. When the unknowns of the group independent sets are labeled first, the matrix A will have the block structure (5) in which the B block is block diagonal instead of diagonal. An illustration of a matrix permuted with two different group-independent set orderings is given in Figure 3.

In ARMS, the following block factorization is computed 'approximately'

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} L & 0 \\ EU^{-1} & I \end{pmatrix} \times \begin{pmatrix} U & L^{-1}F \\ 0 & A_1 \end{pmatrix}, \tag{6}$$

where $LU \approx B$, and $A_1 \approx C - (EU^{-1})(L^{-1}F)$.

Conceptually, the ARMS factorization algorithm is quite simple to describe because of its recursive nature. In a nutshell, the ARMS procedure consists of essentially two steps: first, obtain a group-independent set and reorder the matrix in the form (5); second, obtain an ILU factorization $B \approx LU$ for B and approximations to the matrices $L^{-1}F$, EU^{-1}, and A_1. The process is repeated recursively on the matrix A_1, which may now be renamed A, until a selected number of levels is reached. At the last level, a simple ILUT factorization, possibly with pivoting, or an approximate inverse method can be applied. Figure 4 sketches the ARMS procedure, in which darker shaded areas represent Schur complements that are formed consecutively.

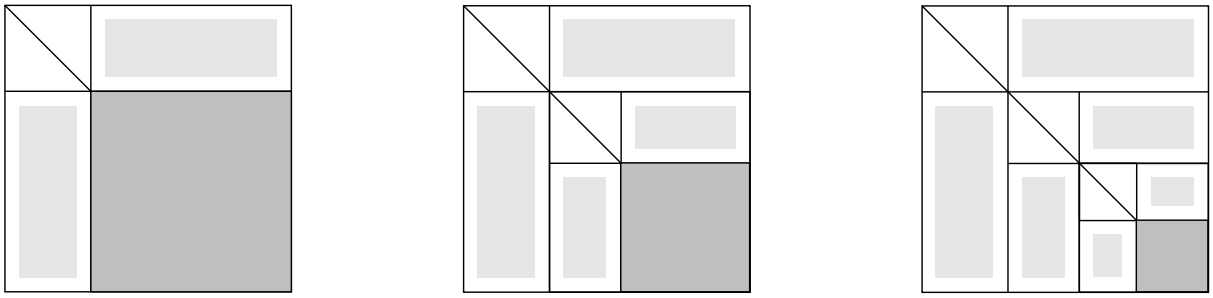


Figure 4: A step of the ARMS procedure to form consecutive Schur complements

The consecutive Schur complement matrices A_1 remain sparse but will get denser as the number of levels increases, so small elements are dropped in the block factorization to maintain sparsity. The matrices $EU^{-1}, L^{-1}F$ (or their approximations) need not be saved. They are needed only to compute an approximation to A_1. Once this is accomplished, they are discarded to save storage. Subsequent operations with $L^{-1}F$ and EU^{-1} are performed using U, L and the blocks E and F.

Recursive multilevel strategies of various types can be defined. Essentially, a preconditioning step amounts to an approximate solve with the factorization (5). Such a solve consists of three steps. The first step will solve with the first matrix on the right-hand side of (6). This will yield a new right-hand side for the second part of (5), which is $g_1 = g - EU^{-1}f$. Then the resulting Schur complement system $A_1 y = g_1$ is solved (iteratively). Finally, a back-substitution step is performed to obtain the variable $u = U^{-1}[f - L^{-1}Fy]$. It is not specified how the Schur complement system $A_1 y = g_1$ is to be solved – and this provides a source of many possible variations. Recursivity can clearly be exploited: if another level in the ARMS factorization is available, we can repeat the process and *descend* to the next level, solve (recursively), and *ascend* back to the current level. When the last level is reached, one can solve the system by an ILUT-preconditioned GMRES iteration [2] or a simple solve with the ILUT factors (without iteration).

Now let us apply a full-fledged ARMS procedure as a local solver in the construction of a global Schur complement preconditioner. For simplicity, consider a one-level ARMS acting locally in each subdomain in the pARMS setting. In Figure 5, solid thick lines determine the boundaries of subdomains. The local group-independent sets are constructed only on those local unknowns that are decoupled from external unknowns by producing local separators (dashed-dotted lines in Figure 5). The inter-domain interface unknowns (dashed lines in Figure 5) are *a priori* assigned to the Schur complement by the ARMS procedure. Thus in each subdomain, the local Schur complement matrix A_1, produced in the first level of ARMS reduction, will act on the inter-domain interface variables plus the separator variables, termed the *local-interface* variables. In other words, the Schur complement variables y_i, $i = 1, \ldots p$ (see (9)), are the union of the inter-domain interface variables and the variables separating the group-independent sets. We denote by *expanded Schur complement* the system involving the matrix A_1 that acts on the inter-domain and local interface unknowns. Upon solving the global system for all the variables y, the interior variables are obtained without communication by back substitution in ARMS within each processor.

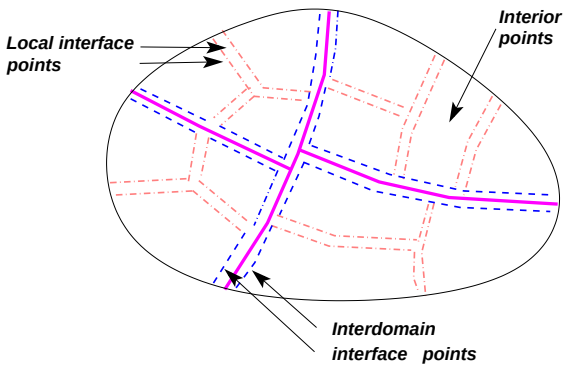


Figure 5: Classification of unknowns when ARMS is used as a local solver in the pARMS setting

1.2.2 Additive Schwarz preconditioning

When an additive Schwarz procedure is used as a preconditioner, the local residual vector is updated by a global iterative solver. The resulting local residual vector is then used as the local right-hand side. Of course, an exchange of data must precede the local residual vector update, such that each sub-domain receives from its neighbors the latest values of its external interface variables and sends to the neighbors the latest values of its inter-domain interface variables. Once the latest values of the external interface variables and the local right-hand side are available, the additive Schwarz procedure can be used to find a correction to the local solution. In simple terms, the Additive Schwarz preconditioners can be stated as follows:

ALGORITHM **1.1** *Additive Schwarz with minimum overlap*

1. *Update local residual* $r_i = (b - Ax)_i$.
2. *Solve* $A_i \delta_i = r_i$.
3. *Exchange* δ_i *among neighboring subdomains.*
4. *Update local solution* $x_i = x_i + \delta_i$.

This loop is executed on each processor simultaneously. Exchange of information takes place in Line 1, where the (global) residual is updated. Note that the residual is "updated" in that only the y-part of the right-hand side is changed. The local systems $A_i \delta_i = r_i$ can be solved in three ways: (1) by a (sparse) direct solver, (2) by using a standard preconditioned Krylov solver, or (3) by performing a forward-backward solution associated with an accurate ILU (e.g., ILUT) preconditioner. Experiments show that option (3) or option (2) with only a very small number of inner steps (e.g., 5) is quite effective.

1.2.3 Schur Complement techniques

Schur complement techniques refer to methods which iterate on the inter-domain interface unknowns only, implicitly using interior unknowns as intermediate variables. Schur complement systems are derived by eliminating the variables u_i from (3). Extracting from the first equation $u_i = B_i^{-1}(f_i - F_i y_i)$ yields, upon substitution in the second equation,

$$S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i - E_i B_i^{-1} f_i \equiv g_i', \tag{7}$$

where S_i is the "local" Schur complement

$$S_i = C_i - E_i B_i^{-1} F_i. \tag{8}$$

The equations (7) for all sub-domains i ($i = 1, \ldots, p$) constitute a global system of equations involving only the inter-domain interface unknown vectors y_i. This global reduced system has a natural block structure related to the inter-domain interface points in each sub-domain:

$$\begin{pmatrix} S_1 & E_{12} & \ldots & E_{1p} \\ E_{21} & S_2 & \ldots & E_{2p} \\ \vdots & & \ddots & \vdots \\ E_{p1} & E_{p-1,2} & \ldots & S_p \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} = \begin{pmatrix} g'_1 \\ g'_2 \\ \vdots \\ g'_p \end{pmatrix}. \tag{9}$$

The diagonal blocks in this system, the matrices S_i, are dense in general. The off-diagonal blocks E_{ij}, which are identical with those involved in (3), are sparse.

The system (9) can be written as $Sy = g'$, where y consists of all inter-domain interface variables $y_1, y_2, \ldots, y_p$ stacked into a long vector. The matrix S is the "global" Schur complement matrix. One can exploit approximate solvers for the reduced system (9) to develop preconditioners for the original (global) distributed system. Once the global Schur complement system (9) is (approximately) solved, *each* processor will compute the u-part of the solution vector (see (1)) by solving the system $B_i u_i = f_i - E_i y_i$ obtained by substitution from (3). In summary, a Schur complement iteration may be expressed by the following algorithm:

ALGORITHM **1.2** *Schur Complement Iteration*

1. *Forward: compute local right-hand sides* $g'_i = g_i - E_i B_i^{-1} f_i$.
2. *Solve global Schur complement system* $Sy = g'$.
3. *Backward: substitute to obtain* u_i, *i.e., solve* $B_i u_i = f_i - E_i y_i$.

For convenience, (7) is rewritten as a preconditioned system with the diagonal blocks:

$$y_i + S_i^{-1} \sum_{j \in N_i} E_{ij} y_j = S_i^{-1} \left[g_i - E_i B_i^{-1} f_i \right]. \tag{10}$$

This can be viewed as a block-Jacobi preconditioned version of the Schur complement system (9). The system (10), which couples local and external unknowns, can be solved by a GMRES-like accelerator, requiring a solve with S_i at each step (see, e.g., [5]).

2 pARMS components

2.1 List of available preconditioners

1. Additive Schwarz type: **add_ilu0, add_ilut, add_iluk, add_arms**;
2. Schur complement-based: **lsch_ilu0, lsch_ilut, lsch_iluk, lsch_arms, rsch_ilu0, rsch_ilut, rsch_iluk, rsch_arms**;
3. Schur complement-based with enhancements: **sch_gilu0, sch_sgs**.

In this list,

add_X indicates additive Schwartz preconditioner with local X preconditioner, where X stands for ILU0, ILUT, ILUK, or ARMS.

lsch_X indicates left Schur complement preconditioner with X as local preconditioner.

rsch_X indicates right Schur complement preconditioner with X as local preconditioner.

sch_gilu0 stands for distributed ILU0 preconditioner on interface nodes.

sch_sgs stands for distributed Gauss-Seidel preconditioner on interface nodes.

2.2 List of available accelerators

Three distributed accelerators (see [2]) are currently available:

1. **fgmresd** is a distributed version of flexible GMRES, which permits iterations in preconditioner application;
2. **dgmresd** is a distributed version of deflated GMRES, which uses eigenvalue deflation;
3. **bcgstabd** is a distributed version of bi-CG stabilized.

Important note. When you use deflated GMRES or BCGSTABD, you cannot use iterations in preconditioner application (i.e., in the inner solve), because this requires flexible preconditioner.

2.3 Functions used to construct distributed matrix

The following functions need to be called for constructing a preconditioner.

1. **CreateMat** allocates memory for the Distributed Matrix;
2. **bdry** constructs arrays of local interface variables;
3. **setup** permutes the local matrix, such that rows corresponding to the interior nodes precede the rows of the interface nodes;
4. **CreatePrec** constructs a preconditioners from the list above.

3 Template drivers in pARMS

There are a few template programs that may be compiled and run to solve a distributed linear system. These templates are provided in the TESTS/GENERAL and TESTS/GRIDS directories, which are used to solve, respectively, (i) a general sparse linear system stored in the Harwell-Boeing sparse matrix (or user-defined) format as a single file and (ii) a model partial-differential equation generated in distributed manner, such that processor generates its local points only. All the template drivers use preconditioned FGMRES.

3.1 TESTS/GENERAL directory

dd-HB-allread.c reads the matrix simultaneously by all the processors. Then each processor calls the partitioner. After and the submatrices assigned to a given processor are extracted, the global matrix is then discarded. In this driver, the right-hand side is either set-up artificially to be A*ones(1:n) or the driver dd-HB.c uses the right-hand side provided.

dd-HB.c is a simplified version of dd-HB-allread.c which does not read parameters from an input file but uses a function to set default values. It also writes information on standard output instead of a file. This driver outputs less information on performance, but this is the simplest driver to run and is *recommended as a starting point.*

dd-HB-bcast.c reads the matrix a single processor. After being partitioned, its rows are distributed to other processors according to a global mapping.

As a partitioner in dd-HB-allread, dd-HB, and dd-HB-bcast, we use "DSE" (Distributed Site Expansion). This is a simple partitioner based on the level-set expansion and on finding partition centers as the starting points.

We also provide drivers **dd-HB-metis.c** and **dd-HB-parmetis.c** to use with the sequential Metis and parallel ParMetis partitioners, respectively, developed by Kumar and Karypis (see http://www.cs.umn.edu/~karypis). Except for partitioners, dd-HB-metis.c and dd-HB-parmetis.c have the functionality similar to dd-HB-allread.c, when artificial right-hand side is used.

Important note. Neither Metis nor ParMetis are distributed with pARMS. To run the template drivers which require them, you need to install these libraries if they do not exist already in your system. You then need to add the proper links to makefile.in (or in configure.in prior to running the configure script).

3.2 Input parameters

All test drivers except dd-HB.c will read test parameters from a file. The sample input file called "inputs" in TESTS/GENERAL is read by default (i.e., when the executables are invoked with no arguments on the command line). Another file can be specified for the inputs by entering it as an argument at the command line

For example:

`%mpirun -np <proc_number> dd-HB-metis.ex` The file called "inputs" will be read by the executable.

`%mpirun -np <proc_number> dd-HB-metis.ex my_inputs` The file called "my_inputs" will be read by the executable. The driver dd-HB.c does not read info from an input file. Instead these are entered by the function set_def_params in `setpar.c`.

Consider a description of the input parameters (see also file "inputs") in the order required by the template drivers.

1. *filename* (character) is a full path to the location of matrix file.
2. *iov* (boolean) is an indicator of overlap: 1 = yes 1-level overlap, 0 = no overlap.
3. *scale* (boolean) is an indicator: 1 = scale the system by 2-norm rows then columns, 0 = do not scale.

4. *usmy* (boolean) is an indicator: 1 = symmertize the matrix pattern (nonzero places), 0 = do not perform symmetrizing.

5. *prec_meth* (character) is a one of the names from the list of existing preconditioners in pARMS.

6. *eps* (double) is the tolerance for the inner iteration of GMRES within a preconditioner. Should be rather low for faster preconditioner application.

7. *eps1* (double) is the tolerance for outer iteration.

8. *nlev* (integer) is the number of levels for the application of the *sequential ARMS preconditioner locally*. The larger it is, the smaller is the local last reduced system at expense of the increased local ARMS solution time. Note that this system cannot be smaller than the size of the inter-domain interface. Good *nlev* values, for example, are 3–6.

9. *bsize* (integer) is the block size for block independent sets to be found by the *sequential ARMS preconditioner locally*. The larger it is, the more storage is needed locally and the fewer `nlev` levels may be found in a matrix. Good *bsize* values , for example, are 10–100.

10. *tolind* (double) is the tolerance used by the *sequential ARMS preconditioner locally* to determine block independent sets. Only rows that are more diagonally dominant than a normalized *tolind* may be included in a block independent set. The larger *tolind* is the fewer block independent sets may be found and thus the larger is the local Schur Complement system. The values of *tolind* typically range from 0.0 to 1 with a good starting value being 0.1. Note that *tolind* is the important parameter the quality of the preconditioner and should be experimented with, if the desired accuracy of the solution cannot be reached with the default values of the "inputs" file.

11. *im* (integer) is the Krylov subspace size for outer iteration.

12. *maxits* (integer) is the number of the outer FGMRES iteration.

13. *kim* (integer) is the Krylov subspace size for inner iteration of the preconditioner application. This parameter should not be large, equal, for example, to 5, to make a preconditioner inexpensive.

14. *itsgmr* (integer) is the number of the inner GMRES iterations used within the preconditioner. This parameter should not be large to make a preconditioner inexpensive.

15. *lfil0* (integer) is the fill-in for the B-, E-, and F-parts of ARMS, or for ILUK, ILUT.

16. *lfil4* (integer) is the fill-in for the sparsity of the successive Schur Complements in ARMS, or unused otherwise.

17. *lfil5* (integer) is the fill-in for the L and U parts of the *local* last reduced system.

18. *droptol0* (double) is the drop tolerance corresponding to *lfil0*.

19. *droptol4* (double) is the drop tolerance corresponding to *lfil4*.

20. *droptol5* (double) is the drop tolerance corresponding to *lfil5*.

21. *multicol* (boolean) is used by sch_sgs and sch_gilu0 preconditioners to indicated whether to use multicolor ordering (1) or global number ordering (0).

Note on the inputs parameters. Most drivers use the `userread` Fortran subroutine in uread.f. `userread` looks at the first line of "inputs". If the first character is not "1", then it expects that line to be the path of a Harwell Boeing matrix. Otherwise it assumes that the rest of the line is the filename of a file containing a pathname to a matrix that is read by userread using any format. The user may substitute the uread file with her/his own routine and this will allow to read matrices in any format.

All the rsch_X and lsch_X preconditioners require inner iterations, so *kim, itsgmr, eps* should be greater than zero for these preconditioners on the input.

3.3 TESTS/GRIDS directory

GRIDS contains test drivers related to simple 2-D and 3-D centered finite difference elliptic PDEs on rectangular domains. The drivers call subroutines included in the library to generate a linear systems associated with the 5-point (or 7-point) discretization of a simple elliptic operator on a rectangular region. Each node generates its own part of the 5/7-point matrix using a simple regular partitioning. The makefile provided will make the executable by invoking its name.

grid.ex is a driver (dd-grid.c) for a vertex-based partitioned problem only; FGMRES is used as a solver

grid-simple.ex This is a simplified version (dd-grid-simple.c) of grid.ex which does not read parameters from an input file but uses a function to set default values. It also writes information on standard output instead of a file (less information is provided on performance, but this is the simplest driver to run and is recommended as a starting point.) The test driver is to be run on 4 processors with the default mesh size of 30 per processor and the default preconditioned `add_ilut`. For the rest of default settings, see function `set_def_params()`.

grid-edge.ex is a driver (dd-grid-edge.c) for an edge-based partitioned problem only; FGMRES is used as a solver.

grid-solver.ex is a driver (dd-grid-solver.c) to test multiple accelerators and partitionings. FGMRES, deflated GMRES, and BCGSTAB are the available accelerators for the outer iterations. You may use one of them as a solver by changing the definition of SOLVER in dd-grid-solver.c to fgmresd, dgmresd, and bcgstabd, respectively. In addition, this driver supports vertex-based as well as edge-based partitioning. [In edge-based partitioning, at least one line overlaps between two neighboring domains]

3.4 Auxiliary files in the TESTS/GRIDS directory

fdmat.f is a suite or routines adapted from SPARSKIT for generating 5-point and 7-point finite difference matrices

functs.f contains functions for defining the PDE being solved on the unit square. It is needed by fdmat.f;

setpar.c reads input parameters for the run from a file also sets default parameters in function `set_def_params()` if input is not read from a file;

inputs is a sample input file for various parameters.

3.5 Input parameters

The parameters *prec_meth, eps, eps1, nlev, bsize, tolind, im, maxits, kim, itsgmr, lfil0, lfil4, lfil5, droptol0, droptol4, droptol5, multicol* are the same as for the drivers in the TESTS/GENERAL directory.

mprocx (integer) is the number of processors in the x direction of the processor grid.

mprocy (integer) is the number of processors in the y direction of the processor grid.

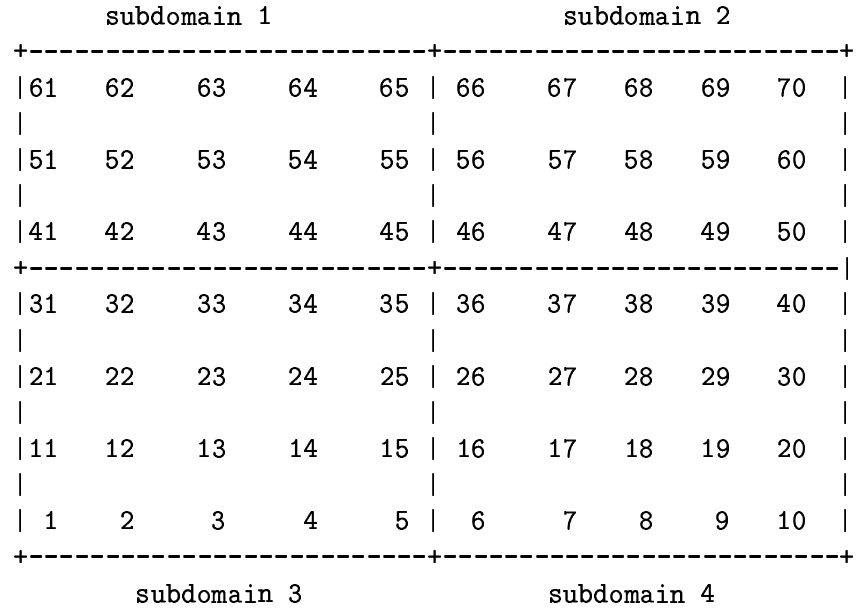
nmesh (integer) is the number of mesh points *per processor*.

Thus the total number of points is as follows: $nmesh \times mprocx \times mprocy$. To preserve the aspect ratio, it is advisable to use $mprocx = mprocy$. The number of processors $mprocz$ in the z direction is given as $mprocz = numproc/(mprocx \times mprocy)$.

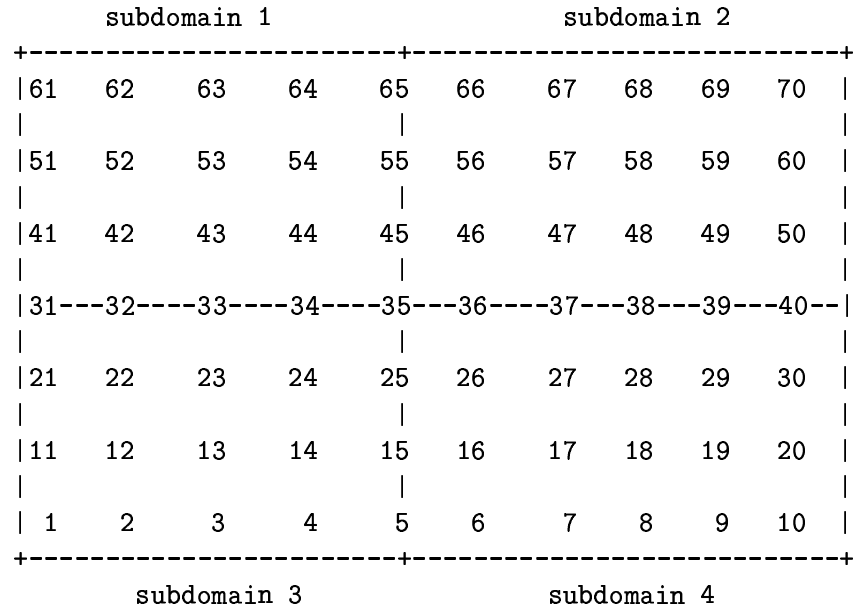
Important note. The input for the GRIDS template drivers specifies the problem size in *each* processor rather than the problem size of the whole problem in all the processors. Thus the execution time may *increase* with the number of processors since the total size of the problem also increases. In a perfectly scalable case, the execution time should be constant.

3.6 Vertex-based and edge-based partitioning

VERTEX based:



EDGE based [element-based or cell-based actually]:



Cells (elements or volumes) are assigned to processors, such that there is always at least one shared line of nodes between neighboring subdomains. For example, nodes 66, 55, 45, 35 are shared by subdomain 1 and subdomain 2.

4 Installation of pARMS

The code has been tested on the IBM SP, SGI Origin3800, and Linux PC and Sun clusters using MPICH at the Minnesota Supercomputing Institute, University of Minnesota and elsewhere. To run the test examples on other machines, it should be necessary only to edit makefile.in and compile. A configure script is also provided which will automatically create the file makefile.in for certain common platforms. You need to first provide some paths in the file configure.in and then type "configure". This will generate the file "makefile.in" which contains the machine specific directives, such as the names of the C and FORTRAN compilers and the path names for the MPI and BLAS libraries. Once the makefile.in file is ready the command makefile will make the library libparms.a in LIB. You may have to alter makefile.in in some cases but you need not change anything to the file "makefile".

There are three makefiles. The makefile in the top directory creates the library LIB/libparms.a. Makefiles in the directories TESTS/GENERAL and TESTS/GRIDS are for running the sample template drivers therein. Instructions on how to make the library libparms.a and sample drivers are given below.

If configure does not work for you (because it does not recognize the system) you can alter the makefile.in directly. See the configure file for more details. After modifying makefile.in, typing 'make' creates the library. 'make tests' will make the library libparms.a AND sample template drivers. However, it is recommended to just make the pARMS library and then go to TESTS/GENERAL (general sparse problems) or TESTS/GRIDS (finite difference problems) to run some or all of the sample template problems provided.

4.1 How to run the template drivers

Once you have the driver **driver.ex**, where driver is one of grid, grid-edge, grid-solver, you can execute driver.ex as follows: For example, on the IBM SP, type

```
%poe driver.ex -hostfile host.file -procs 4
```

to execute on 4 processors. In this case, the driver will use the input file "inputs" given above and will write the results on standard output. You can also specify different input and output files:

```
%driver.ex [inp [out]]
```

For example on the IBM SP:

```
%poe driver.ex inp out -hostfile host.file -procs 4
```

uses the file `inp` as input file and `out` for output.
Note that the driver **grid-simple.ex** will not use an input file to read the test parameters. Also results are printed to standard output. For example:

```
%mpirun grid-simple.ex -np 4
```

will execute grid-simple.ex on 4 processors.

5 Version and contact information

Current version is 0.2. The list of known bugs:

1. rsch_arms preconditioner is not stable on Linux clusters.

Please submit bug reports to ***saad@cs.umn.edu*** or to ***masha@cs.umn.edu*** by providing an exact error message, stating the computing platform, and briefly describing the problem. Future updates of pARMS will be posted in **www.cs.umn.edu/~saad/software/PARMS**.

References

[1] Z. Li, Y. Saad, and M. Sosonkina, *pARMS: A parallel version of the algebraic recursive multilevel solver*, Numerical Linear Algebra with Applications, 10 (2003), pp. 485–509.

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2nd ed., 2003.

[3] Y. Saad and A. Malevsky, *PSPARSLIB: A portable library of distributed memory sparse iterative solvers*, in Proceedings of Parallel Computing Technologies (PaCT-95), 3-rd international conference, St. Petersburg, Russia, Sept. 1995, V. E. M. et al., ed., 1995.

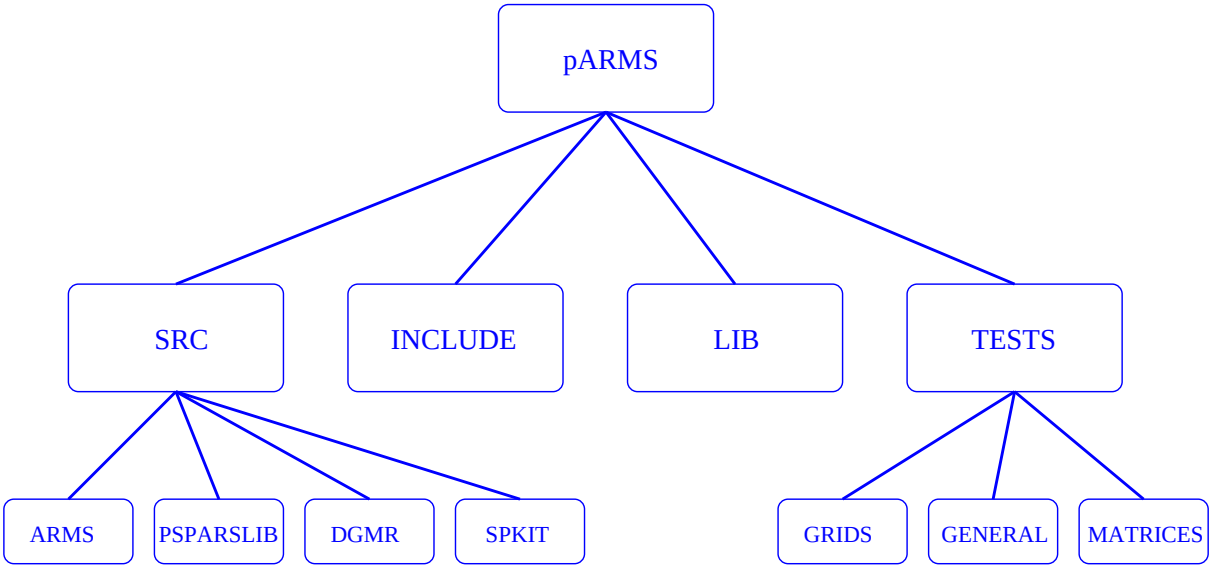
[4] Y. Saad and M. Sosonkina, *Solution of distributed sparse linear systems using PSPARSLIB*, in Applied Parallel Computing, PARA'98, B. Kågström et al., eds., Lecture Notes in Computer Science, No. 1541, Berlin, 1998, Springer-Verlag, pp. 503–509.

[5] Y. Saad and M. Sosonkina, *Distributed Schur Complement techniques for general sparse linear systems*, SIAM J. Scientific Computing, 21 (1999), pp. 1337–1356.

[6] Y. Saad and B. Suchomel, *ARMS: An algebraic recursive multilevel solver for general sparse linear systems*, Tech. Rep. umsi-99-107, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1999.

[7] Y. Saad and J. Zhang, *BILUM: Block versions of multi-elimination and multi-level ilu preconditioner for general sparse linear systems*, SIAM Journal on Scientific Computing, 21 (2000).

6 Appendix A: pARMS directory structure



SRC/	the library source files
ARMS	sequential ARMS
PSPARSLIB	files containing parallel implementations
DGMR	files needed by deflated GMRES
SPKIT	sequential FORTRAN and C sparse matrix routines
TESTS/	driver programs, input files
GRIDS/	template drivers for regularly structured programs
extras	sample input files and batch scripts for GRIDS
GENERAL/	test drivers for general problems
extras	sample input files and batch scripts for GENERAL
MATRICES	sample problems in the Harwell-Boeing format
INCLUDE	header files
LIB	directory in which the pARMS library is stored
COPYRIGHT	copyright file
GNU	GNU licensing agreement
README	information file
configure	a script to automatically modify makefile.in
configure.in	a template for the configure script
makefile.in	a part of the makefile that may be modified
makefile	complete makefile (does not need modification)