pARMS: A Package for the Parallel Iterative Solution of General Large Sparse Linear System *
Reference Manual

November 10, 2010

1 Introduction

This reference manual is a supplementary material to the user's guide, and aims to introduce first-time or "basic" users to some of the features of the pARMS package. Please contact *dosei@cs.umn.edu* or *saad@cs.umn.edu* for more details.

This manual will mainly highlight functions users will need to call. Functions that are defined as part of the package, but used mainly internally will not be included here. The main areas covered are the *parms_mem.h* file reference, and the *parms_comm, parms_map, parms_mat, parms_pc, parms_operator, parms_solver* object structs.

In what follows, depending on the version of pARMS compiled (i.e. real (with -DDBL flag) or complex (with -DDBL_CMPLX flag)) the type FLOAT will either represent a double - for the real case, or a complex double - for the complex case. The type REAL represents a real floating point number.

2 Memory

2.1 parms_mem.h File Reference:

This file contains function definitions that allow for efficient memory control for the pARMS package.

2.2 Function Details

- **PARMS_ALLOC(n)**
  This is used for allocating n bytes, which is useful for reducing the number of calls to the malloc functions, by combining multiple malloc calls together.
- **PARMS_FREE(p)**
  Free the memory allocated for some object p. Here, p could be a simple vector array or a pARMS struct object.
- **PARMS_NEWARRAY(p, n)**
  Allocate memory for an array p, of size n bytes. Here, p could be defined as an array of integer or real entries for instance (eg. int *p)

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

- **PARMS_NEWARRAY0(p, n)**
  Allocate memory for an array p, of size n bytes and initialize entries to zero.
- **PARMS_NEW(p)**
  Allocate memory for a pARMS struct object p. p could be for instance, a matrix object - parms_mat.
- **PARMS_NEW0(p)**
  Allocate memory for a pARMS struct object p, and initialize to zero
- **PARMS_RESIZE(p, size)**
  Changes the size of an allocated memory referenced by p.

3 pARMS Communication Object

3.1 parms_comm.h File Reference:

The pARMS communication handler is used to facilitate the efficient computation of the matrix-vector product. Most of these functions will likely never be called directly by the user, but have been included here to reference purposes.

3.2 Function Details

- **int parms_CommCreate(parms_Comm *comm, MPI_Comm mpicomm)**
  Create a pARMS communication object.
- **int parms_CommDataBegin(parms_Comm comm, void *data, int offset)**
  Exchange data for the matrix-vector product. This function exchanges the interface variables. The parameter $offset$ indicates the distance between the start of the data to be exchanged, and the start of the local vector. This is useful for Schur complement based preconditioners since the data parameter is only the interface part of the local vector, rather than the entire local vector.
- **int parms_CommDataEnd(parms_Comm comm)**
  Wait for all messages after data is sent to a recieve buffer. After calling this function, you may use the data in the recieve buffer safely.
- **int parms_CommFree(parms_Comm *comm)**
  Free memory allocation for parms_Comm object.
- **int parms_CommGetNumRecv(parms_Comm comm)**
  Get the total number of variables recieved.
- **int parms_CommGetNumSend(parms_Comm comm)**
  Get the total number of variables sent.
- **int parms_CommGetRecvBuf(parms_Comm comm, FLOAT **rbuf)**
  Get the receive buffer.
- **int parms_CommView(parms_Comm comm, parms_Viewer v)**
  Dump the communication handler, *comm*, to a file prescribed by the pARMS viewer object v.

4 pARMS Map Object

4.1 parms_Map.h File Reference

The parms Map object represents how data are distributed across processors. It is the most important object in pARMS. All other types of objects are created based on it. Users must make sure the routines used for creating a parms Map object are consistent with the graph partitioning routines creating a parms Map object.

4.2 Function Details

- **int parms_MapCreateFromDist(parms_Map *map, int *vtxdist, int *part, MPI_Comm mpicomm, int offset, int dof, VARSTYPE vtype)**
  Create a parms_Map object based on the output from a parallel graph partitioner. Here, we assume ParMetis is used.
  - **vtxdist:** An integer array of size $nproc + 1$, where $nproc$ is the number of processors. It indicates the range of vertices that are local to each processor (i.e. processor i stores vertices in the range of $(vtxdist[i], vtxdist[i+1])$.)
  - **part:** An integer array of size equal to the number of locally stored vertices. $part[j]$ indicates the processor id to which the vertex with local index j and global index $vtxdist[pid] + j$ belongs to (where pid is the processor id).
  - **offset:** This refers to the C or FORTRAN array start index - 1 for FORTRAN and 0 for C.
  - **dof:** This is the number of variables associated with each vertex.
  - **vtype:** This refers to the ordering of the variables. Assuming the variables u_i and v_i are associated with the vertex i, two styles of numbering could be used as follows:
    1. **INTERLACED:** Variables are ordered as $u_1, v_1, u_2, v_2, ...$
    2. **NONINTERLACED:** Variables are ordered as $u_1, u_2, u_3, ..., v_1, v_2, v_3, ...$
- **int parms_MapCreateFromGlobal(parms_Map *map, int gsize, int *npar, MPI_Comm mpicomm, int offset, int dof, VARSTYPE vtype)**
  Create a parms_Map object based on the partitioning results of a serial graph partitioner. Here we assume Metis is used.
  - **gsize:** The total number of vertices (or global size)
  - **npar:** An integer array of size gsize. Node i resides on processor $npar[i]$.
  - **offset:** This refers to the C or FORTRAN array start index - 1 for FORTRAN and 0 for C.
  - **dof:** This is the number of variables associated with each vertex.
  - **vtype:** This refers to the ordering of the variables. Assuming the variables u_i and v_i are associated with the vertex i, two styles of numbering could be used as follows:
    1. **INTERLACED:** Variables are ordered as $u_1, v_1, u_2, v_2, ...$
    2. **NONINTERLACED:** Variables are ordered as $u_1, u_2, u_3, ..., v_1, v_2, v_3, ...$
- **int parms_MapCreateFromLocal(parms_Map *map, int size, int offset)**
  Create a parms_Map object directly the on the local processor. This is ideal for applications where, say a grid is defined locally. Hence the local variables already reside on the local processor.
  - **size:** The total number of unknowns on the local processor

- **offset:** This refers to the C or FORTRAN array start index - 1 for FORTRAN and 0 for C.

- **int parms_MapCreateFromPetsc(parms_Map *map, int m, int M, MPI_Comm mpicomm)**
  Create a parms_Map object based on the default partitioning strategy in PETSc.
  - **m:** The local size of variables
  - **M:** The global size of variables.

- **int parms_MapCreateFromPtr(parms_Map *map, int gsize, int *nodes, int *p2nodes, MPI_Comm mpicomm, int dof, VARSTYPE vtype)**
  Create a parms_Map object based on the partitioning results of a general (or user defined) graph partitioning strategy such as DSE.
  - **gsize:** The total number of vertices (or global size)
  - **nodes:** An array of size gsize, containing a list of all vertices stored processor by processor.
  - **p2nodes:** An integer array of size $nproc+1$. If $k1 = p2nodes[i]$ and $k2 = p2nodes[i+1]$, then processor i contains the vertices in the range of $(nodes[k1], nodes[k2])$.
  - **dof:** This is the number of variables associated with each vertex.
  - **vtype:** This refers to the ordering of the variables. Assuming the variables u_i and v_i are associated with the vertex i, two styles of numbering could be used as follows:
    1. **INTERLACED:** Variables are ordered as $u_1, v_1, u_2, v_2, ...$
    2. **NONINTERLACED:** Variables are ordered as $u_1, u_2, u_3, ..., v_1, v_2, v_3, ...$

- **int parms_MapFree(parms_Map *map)**
  Free the pARMS map object, *map*.

- **int parms_MapGetGlobalSize(parms_Map map)**
  Get the global size of variables.

- **int parms_MapLocalSize(parms_Map map)**
  Get the local size of variables.

- **int parms_MapGetNumProcs(parms_Map map)**
  Get the number of processors.

- **int parms_MapGetPid(parms_Map map)**
  Get the processor id.

- **int parms_MapGetGlobalIndices(parms_Map map, int *lvars)**
  Get the global indices for the local variables residing on this processor. On return, the vector lvars contain the global indices of the variables on this processor.

- **int *parms_MapFree(parms_Map map, int gindex)**
  Returns a pointer to the local index for a given global index *gindex*. If NULL, then the variable with global index *gindex* does not reside on the local processor.

- **int parms_MapView(parms_Map map, parms_Viewer v)**
  Dump the pARMS map object to a file prescribed by the pARMS viewer object v.

5 pARMS Mat Object

5.1 parms_mat.h File Reference:

The parms_mat object defines a serial or distributed matrix in CSR storage format. During the setup phase for the distributed matrix, the local sub-matrix is reordered so that interior (or independent) variables are labeled first, followed by the interface variables.

5.2 Function Details

- **int parms_MatCreate(parms_Mat *mat, parms_Map map)**
  Create a pARMS mat object based on data distribution layout map.
- **int parms_MatFree(parms_Mat *mat)**
  Free memory allocation for pARMS mat object.
- **int parms_MatGetDiag(parms_Mat mat, void **mat)**
  Get the diagonal part of the local matrix. On return, *mat* contains the diagonal part of the local matrix.
- **int parms_MatMVPY(parms_Mat mat, FLOAT alpha, FLOAT *x, FLOAT beta, FLOAT y, FLOAT, z)**
  Perform $z = alpha \times mat \times x + beta \times y$, where alpha and beta are scalars, and x, y and z are vectors.
- **int parms_MatSetCommType(parms_Mat mat, COMMTYPE ctype)**
  Set the communication style across processors. The communication styles associated with the variable *ctype* are:
  - **P2P:** point-to-point (data copied to or from auxilliary buffers).
  - **DERIVED:** derived datatype
- **int parms_MatSetValues(parms_Mat mat, int m, int *im, int *ia, int *ja, FLOAT *values, INSERTMODE mode)**
  Insert or add values to the parms_Mat object, *mat*. Here, we assume the entries are given in CSR format (ia, ja, values):
  - m**:** The number of rows to be inserted
  - **im:** An array of size m containing the global indices of the rows to be inserted
  - **ia:** An array pointer of size $m + 1$, pointing to the beginning of each row in array ja
  - **ja:** An array of global column indices
  - **values:** An array of values to be inserted
  - **mode:** The insert mode. One of two choices:
    - **INSERT:** Insert values into the parms_Mat object
    - **ADD:** Add values to the parms_Mat object
- **int parms_MatSetElementMatrix(parms_Mat mat, int m, int *im, int *ia, int *ja, FLOAT *values, INSERTMODE mode)**
  Insert or add values to the parms_Mat object, *mat*. This assumes the values are being set or added element-by-element. Hence this is suitable for applications where one loops over domain elements to set the matrix values (such as a finite element application). A call to parms_MatAssembleElementMatrix() is required to complete the matrix once all entries have been added. We assume the entries are given in CSR format (ia, ja, values):

- m**:** The number of rows to be inserted
- **im:** An array of size m containing the global indices of the rows to be inserted
- **ia:** An array pointer of size $m + 1$, pointing to the beginning of each row in array ja
- **ja:** An array of global column indices
- **values:** An array of values to be inserted
- **mode:** The insert mode. One of two choices:
  - **INSERT:** Insert values into the parms_Mat object
  - **ADD:** Add values to the parms_Mat object

- **int parms_MatAssembleElementMatrix(parms_Mat mat)**
  Assembles the (finite) element matrix and updates off-processor contributions.

- **int parms_MatResetRowValues(parms_Mat mat, int m, int *im, int *ia, int *ja, FLOAT *values)**
  Insert values to the parms_Mat object, *mat*. This assumes the matrix values for this row have already been set, and are to be replaced by new ones provided as input. Here, we assume the entries are given in CSR format (ia, ja, values):
  - m**:** The number of rows to be inserted
  - **im:** An array of size m containing the global indices of the rows to be inserted
  - **ia:** An array pointer of size $m + 1$, pointing to the beginning of each row in array ja
  - **ja:** An array of global column indices
  - **values:** An array of values to be inserted

- **int parms_MatReset(parms_Mat mat, NNZSTRUCT nonzerostructure)**
  Reset the parms_Mat object, *mat*, to be re-used.
  - m**:** The number of rows to be inserted
  - **im:** An array of size m containing the global indices of the rows to be inserted
  - **ia:** An array pointer of size $m + 1$, pointing to the beginning of each row in array ja
  - **ja:** An array of global column indices
  - **values:** An array of values to be inserted
  - **nonzerostructure:** The nonzero pattern to use. One of two choices:
    - **DIFFERENT_NONZERO_STRUCTURE:** Resets the matrix object to receive data using a different nonzero structure from before
    - **SAME_NONZERO_STRUCTURE:** Resets the matrix object to receive data using the same nonzero structure as before

- **int parms_MatSetup(parms_Mat mat)**
  Setup the pARMS matrix object *mat*. This is the most important function for the parms_Mat object. It sets up the data structure needed for the distributed matrix-vector multiplication by dividing the local variables into two categories: interior, and interface variables.

- **int parms_MatVec(parms_Mat mat, FLOAT *x, FLOAT *y**
  Perform $y = mat \times x$, where x and y are vectors.

- **int parms_MatView(parms_Mat mat, parms_Viewer v)**
  Dump the parms_Matrix object to a file prescribed by the pARMS viewer object v.

6 pARMS Operator Object

6.1 parms_Operator.h File Reference:

The parms_Operator struct defines the different ilu-based local preconditioners included in the package (ILU0, ILU(k), ILUT, Symmetric ARMS, and ARMS-ddPQ (non-symmetric)). These functions are typically not called directly by the user, but could come in handy if one would like to define one's own operator.

6.2 Function Details

- **int parms_OperatorApply(parms_Operator M, FLOAT *y, FLOAT *x)**
  Perform $x = M^{-1}y$, where x and y are the solution and right-hand-side vectors respectively. Assuming $M = LU$, this function solves $LUx = y$.
- **int parms_OperatorAscend(parms_Operator M, FLOAT *y, FLOAT *x)**
  Assuming $M = LU$, this function performs the backward solve $Ux = y$, where x and y are the solution and right-hand-side vectors respectively.
- **int parms_OperatorCreate(parms_Operator *M)**
  Create an Operator object.
- **int parms_OperatorFree(parms_Operator *M)**
  Free memory for the parms_Operator object.
- **int parms_OperatorGetNnz(parms_Operator M, int *nnz_mat, int *nnz_pc)**
  Get number of non-zeros in the original matrix, *nnz_mat*, and the preconditioned matrix, *nnz_pc*. Note: *nnz_mat* and *nnz_pc* are actually pointers to the number of non-zeros in the original matrix and the preconditioned matrix respectively.
- **int parms_OperatorGetSchurPos(parms_Operator M)**
  Get the start position of the Schur complement in the local matrix.
- **int parms_OperatorInvS(parms_Operator M, FLOAT *y, FLOAT *x)**
  Perform the Schur complement solve. Assume
  $$M = \begin{pmatrix} L & 0 \\ EU^{-1} & I \end{pmatrix} \begin{pmatrix} U & L^{-1}F \\ 0 & S \end{pmatrix},$$
  then this function solves $Sx = y$, for some solution vector x, and rhs vector y.
- **int parms_OperatorLsol(parms_Operator M, FLOAT *y, FLOAT *x)**
  Assuming $M = LU$, this function performs the forward solve $Lx = y$, where x and y are the solution and right-hand-side vectors respectively.
- **int parms_OperatorView(parms_Operator M, parms_Viewer v)**
  Output the L and U parts of the parms_Operator to a file prescribed by the pARMS viewer object v.

7 pARMS Preconditioner Object

7.1 parms_pc.h File Reference:

Preconditioner related functions for the pARMS solver package. There are three main global preconditioners included in the package: Block Jacobi, Restricted Additive Schwarz, and Schur complement. Each of these global preconditioners utilize one of five different local preconditioners (operators) for the linear solve. They are: ILU0, ILU(k), ILUT, Standard (symmetric) ARMS, and Non-symmetric ARMS (ARMS-ddPQ).

7.2 Function Details

- **int parms_PCCreate(parms_PC *pc, parms_Mat mat)**
  Create a preconditioner object based on the matrix *mat*
- **int parms_PCCreateAbstract(parms_PC *pc)**
  Create an abstract preconditioner object. This may be useful for creating one's own global preconditioner
- **int parms_PCFree(parms_PC *pc)**
  Free memory for the preconditioner object
- **int parms_PCGetName(parms_PC pc, char **name)**
  Return the name of the preconditioner
- **int parms_PCGetRatio(parms_PC pc, double *ratio)**
  Get the ratio of the number of non-zero entries of the preconditioner, to that of the original matrix. On return, *ratio* points to this value.
- **int parms_PCILUGetName(parms_PC pc, char **iluname)**
  Return the name of the local ilu-based preconditioner
- **int parms_PCSetBsize(parms_PC pc, int bsize)**
  Set the size of the B block for the ARMS reordering of the matrix

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix}.$$

- **int parms_PCSetFill(parms_PC pc, int *fill)**
  Set the fill-in parameter for ILUT, ILUK and ARMS. Here, $fill$ is an array of size 7. Let

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} L_B & 0 \\ EU_B^{-1} & I \end{pmatrix} \begin{pmatrix} U_B & L_B^{-1}F \\ 0 & S \end{pmatrix} = M,$$

  then during the factorization, the entries in $fill$ are used as follows:
  - fill[0] - amount of fill-in kept in L_B
  - fill[1] - amount of fill-in kept in U_B
  - fill[2] - amount of fill-in kept in EU_B^{-1}
  - fill[3] - amount of fill-in kept in $L_B^{-1}F$
  - fill[4] - amount of fill-in kept in the formation of S
  - fill[5] - amount of fill-in kept in the L part of the factorization of S, L_S
  - fill[6] - amount of fill-in kept in the U part of the factorization of S, U_S

  If the local preconditioner is ILUT or ILUK, then the level of fill parameter for these preconditioners will take the value of $fill$[0].
- **int parms_PCILUSetType(parms_PC pc, PCILUTYPE pctype)**
  Set the local preconditioner type:
  - **PCILU0:** ILU0 preconditioner
  - **PCILUK:** ILU(k) preconditioner
  - **PCILUT:** ILUT preconditioner
  - **PCARMS:** ARMS preconditioner

For the ARMS preconditioner, one can use either the symmetric ARMS or the non-symmetric variant (ARMS-ddPQ) by setting the permutation type. See parms_PCSetPermType(...).

- **int parms_PCSetInnerEps(parms_PC pc, REAL eps)**
  Set the convergence tolerance, *eps*, for the inner GMRES solve, when the Schur (global) preconditioner is used.

- **int parms_PCSetInnerKSize(parms_PC pc, int im)**
  Set the restart size, *im*, for the inner GMRES solve, when the Schur (global) preconditioner is used.

- **int parms_PCSetInnerMaxits(parms_PC pc, int imax)**
  Set the maximum iteration count, *imax*, for the inner GMRES solve, when the Schur (global) preconditioner is used.

- **int parms_PCSetNlevels(parms_PC pc, int nlevel)**
  Set the number of levels for ILUK and ARMS.

- **int parms_PCSetOP(parms_PC pc, parms_Mat mat)**
  Set the matrix to be used to create the preconditioning matrix. The preconditioner will be built from the matrix *mat*.

- **int parms_PCSetParams(parms_PC pc, int nflags, char **params)**
  Set the parameters for the preconditioner object. Ideal for setting parameters collectively. Supported parameters are:

  - **tol:** the drop tolerance
  - **fil:** the fill-in
  - **nlev:** the number of levels
  - **bsize:** the block size for finding independent sets in ARMS
  - **tolind:** the drop tolerance for finding independent sets
  - **iksize:** the restart size for the inner GMRES
  - **imax:** the number of iterations for the inner GMRES

  The argument $nflags$ is the number of parameters to be set. The argument *params* is a pointer to an array pointer of characters containing the parameter name, followed by the value to be set. Thus, say we wish to set the restart dimension for the inner GMRES to 10, and the number of iterations for the inner GMRES to 20, then *params* could take the form: *char *params = {'iksize','10','imax','20'}*. We can then call parms_PCSetParams(pc, 2, &params) to set these parameter values for the preconditioner. Alternatively, we could also call the parms_PCSetInnerKSize(...) and the parms_PCSetInnerMaxits(...) to set these parameters individually.

- **int parms_PCSetPermScalOptions(parms_PC pc, int *meth, int flag)**
  Set the permutation and scaling options for the interlevel blocks (when the ARMS local preconditioner is used). The argument *meth* is an array pointer of size 4, which takes values of 0 or 1, with the following meaning:

  - **meth[0]:** non-symmetric row permutation - (1 = *yes*, and 0 = *no*)
  - **meth[1]:** column permutations (eg. ILUTP instead of ILUT) - (1 = *yes*, and 0 = *no*)
  - **meth[2]:** diagonal row scaling - (1 = *yes*, and 0 = *no*)
  - **meth[3]:** diagonal column scaling - (1 = *yes*, and 0 = *no*)

The argument $flag$ indicates the level for which to set these parameters: $flag = 0$ indicates that these options will be set for the last level block only; $flag = 1$ indicates that they will be set for the intermediate level blocks only.

NOTE: Currently $meth[1]$ is only used at the last level block, for the $flag = 1$, the value of $meth[1]$ does not matter.

- **int parms_PCSetPermType(parms_PC pc, int type)**
  Set the type of permutation for ARMS: $type = 0$ implies standard symmetric ARMS, and $type = 1$ implies non-symmetric ARMS (ARMS-ddPQ).

- **int parms_PCSetTol(parms_PC pc, double *tol)**
  Set the drop tolerance for ILUT and ARMS. Here, *tol* is an array of size 7. Let

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} L_B & 0 \\ EU_B^{-1} & I \end{pmatrix} \begin{pmatrix} U_B & L_B^{-1}F \\ 0 & S \end{pmatrix} = M,$$

  then during the factorization, the entries in *tol* are used as follows:

  - tol[0] - threshold for dropping in L_B
  - tol[1] - threshold for dropping in U_B
  - tol[2] - threshold for dropping in EU_B^{-1}
  - tol[3] - threshold for dropping in $L_B^{-1}F$
  - tol[4] - threshold for dropping in the formation of S
  - tol[5] - threshold for dropping in the L part of the factorization of S, L_S
  - tol[6] - threshold for dropping in the U part of the factorization of S, U_S

  If the local preconditioner is ILUT, then the drop tolerance parameter for this preconditioner will take the value of $tol[0]$.

- **int parms_PCSetTolInd(parms_PC pc, REAL tolind)**
  Set the tolerance for finding independent sets (for ARMS).

- **int parms_PCSetType(parms_PC pc, PCTYPE pctype)**
  Set the (global) preconditioner type, *pctype*. Currently supported preconditioners are: **PCBJ** - block Jacobi; **PCRAS** - restricted additive Schwarz; and **PCSCHUR** - Schur complement.

- **int parms_PCSetup(parms_PC pc)**
  Setup the preconditioner - create the preconditioning matrix.

- **int parms_PCSolve(parms_PC pc, FLOAT *y, FLOAT *x)**
  Perform the (global) preconditioner solve $Mx = y$, for some preconditioner M, solution vector x, and rhs vector y.

- **int parms_PCView(parms_PC pc, parms_Viewer v)**
  Dump the preconditioner object to a file prescribed by the pARMS viewer object v.

8 pARMS Solver Object

8.1 parms_Solver.h File Reference:

Functions related to the Krylov subspace methods. Currently, only GMRES and FGMRES (default) are supported.

8.2 Function Details

- **int parms_SolverApply(parms_Solver solver, FLOAT *x, FLOAT *y)**
  Solve the equation $Ax = y$, for some matrix A, solution vector x, and rhs vector y.
- **int parms_SolverCreate(parms_Solver *solver, parms_Mat mat, parms_PC pc)**
  Create a pARMS solver object.
- **int parms_SolverFree(parms_Solver *solver)**
  Free memory allocation for the pARMS solver object.
- **int parms_SolverGetIts(parms_Solver solver)**
  Get the iteration count.
- **int parms_SolverGetMat(parms_Solver solver, parms_Mat *mat)**
  Get the matrix of the linear system. On return, a pointer to the matrix *mat* is returned.
- **int parms_SolverGetPC(parms_Solver solver, parms_PC *pc)**
  Get the preconditioning matrix. On return, a pointer to the preconditioner *pc* is returned.
- **int parms_SolverGetResidual(parms_Solver solver, FLOAT *y, FLOAT *x, FLOAT *rvec)** Get the residual vector. Here, x is the solution vector, y is the right hand side vector, and *rvec* is the computed residual vector to be returned.
- **int parms_SolverGetResidualNorm(parms_Solver solver, FLOAT *y, FLOAT *x, REAL *rnorm)** Get the 2-norm of the residual. Here, x is the solution vector, y is the right hand side vector, and *rnorm* is a pointer to the computed residual norm to be returned. Note that we assume that the residual has not been pre-computed. If the residual vector is already known, then we may simply call parms_VecGetNorm2(...) to compute the norm (or use the BLAS routines).
- **int parms_SolverSetParam(parms_Solver solver, PARAMTYPE ptype, char *param)**
  Set a parameter for the solver. The argument *ptype* takes one of the following values:
  - **MAXITS:** maximum iteration count
  - **KSIZE:** restart dimension for GMRES
  - **DTOL:** convergence tolerance

  The argument *param* is a pointer to the value to be set.
- **int parms_SolverSetType(parms_Solver solver, SOLVERTYPE stype)**
  Set the type of Krylov solver to use. *stype* takes the value SOLGMRES for GMRES (left preconditioned), and SOLFGMRES for the default (right preconditioned) FGMRES.
- **int parms_SolverView(parms_Solver solver, parms_Viewer v)**
  Dump the solver object to a file prescribed by the pARMS viewer object v.

9 parms_sys.h File Reference

This file contains the macros and typedefs needed by all other header files. The enumerations below define some typdefs and the choices they take:

- enum **VARSTYPE** {INTERLACED, NONINTERLACED}
- enum **INSERTMODE** {INSERT, ADD}
- enum **COMMTYPE** {P2P, DERIVED}

- enum **SOLVERTYPE** {SOLFGMRES, SOLGMRES}
- enum **PARAMTYPE** {MAXITS, KSIZE, DTOL}
- enum **MATTYPE** {MAT_NULL=-1, MAT_VCSR=0, MAT_CSR=1}
- enum **PCTYPE** {PCBJ, PCSCHUR, PCRAS}
- enum **PCILUTYPE** {PCILU0, PCILUK, PCILUT, PCARMS}

10 pARMS Timer Object

10.1 parms_Timer.h File Reference

Functions related to the parms_Timer object.

10.2 Function Details

- **void parms_TimerCreate(parms_Timer *tt)**
  Create a timer object
- **int parms_TimerFree(parms_Timer *tt)**
  Free memory for parms_Timer object
- **double parms_TimerGet(parms_Timer tt)**
  Return the elapsed wall clock time in seconds, since the last call to parms_TimerReset, parms_TimerResetDelay, or parms_TimerRestart.
- **int parms_TimerPause(parms_Timer tt)**
  Pause the parms_timer object.
- **int parms_TimerReset(parms_Timer tt)**
  Reset parms_Timer object.
- **int parms_TimerResetDelay(parms_Timer tt, double delay)**
  Reset the elapsed time of the parms_Timer object, *tt*, to *delay*.
- **int parms_TimerRestart(parms_Timer tt)**
  Restart the timer.
- **int parms_TimerView(parms_Timer tt, parms_Viewer v)**
  Dump the parms_Timer object to a file prescribed by the pARMS viewer object v.

11 pARMS Vectors

Vectors in pARMS are defined by the standard C array pointers. This was done to simplify the code, and also allow for easy portability to application programs or codes. Nonetheless, some vector functions have been defined as part of this package to facilitate easy coding with the package. Some of these functions have blas versions, and these versions should be used whenever the user has blas-lapack installed (See README file).

11.1 Function Details

- **int parms_VecAXPY(FLOAT *vec, FLOAT *x, FLOAT alpha, parms_Map map)**
  Perform $vec = alpha \times x + vec$, where *alpha* is a scalar, and *vec* and x are vectors. (NOTE: pARMS will use daxpy (or zaxpy for complex arithmetic) if blas is installed).

- **int parms_VecAYPX(FLOAT *vec, FLOAT *x, FLOAT alpha, parms_Map map)**
  Perform $vec = alpha \times vec + x$, where *alpha* is a scalar, and *vec* and x are vectors.

- **int parms_VecDOT(FLOAT *vec, FLOAT *x, FLOAT value, parms_Map map)**
  Perform the inner product of two vectors. On return, *value* contains the result. (NOTE: pARMS will use ddot (zdotu for complex arithmetic) if blas is installed).

- **int parms_VecDOTC(FLOAT *vec, FLOAT *x, REAL value, parms_Map map)**
  Perform the inner product of two (complex-valued) vectors. On return, *value* contains the result of the dot product between *vec* and the complex conjugate of x. (NOTE: pARMS will use zdotc if blas is installed).

- **int parms_VecGetNorm2(FLOAT *vec, REAL *value, parms_Map map)**
  Return a pointer to the 2-norm of the vector *vec*.

- **int parms_VecGather(FLOAT *self, FLOAT *ga, parms_Map map)**
  Gather a distributed vector, *vec*, into a global array, *ga*.

- **int parms_VecInvPermAux(FLOAT *vec, FLOAT *aux, parms_Map map)**
  Perform the inverse permutation on the vector *vec*. On return, *aux* contains the (inversely) permuted vector.

- **int parms_VecPermAux(FLOAT *vec, FLOAT *aux, parms_Map map)**
  Permute the vector *vec*, into *aux*. On return, *aux* contains the permuted vector.

- **int parms_VecScale(FLOAT *vec, FLOAT alpha, parms_Map map)**
  Scale the vector *vec* by the scalar *alpha*.

- **int parms_VecSetValues(FLOAT *vec, int m, int *im, FLOAT *values, INSERTMODE mode, parms_Map map)**
  Insert values into the vector *vec*. Here, m is the number of variables to be inserted; *im* is an array of global variable indices; *value* is an array of the values to be inserted; and *mode* is the insert mode: **ADD** will add values to the vector entries, and **INSERT** will insert the values into the vector.

- **int parms_VecSetElementVector(FLOAT *vec, int m, int *im, FLOAT *values, INSERTMODE mode, parms_Map map)**
  Insert values into the vector *vec*. This assumes the vector values are being set element-by-element. A call to parms_VecAssembleElementVector() is required to complete the vector once all entries have been added. Here, m is the number of variables to be inserted; *im* is an array of global variable indices; *value* is an array of the values to be inserted; and *mode* is the insert mode: **ADD** will add values to the vector entries, and **INSERT** will insert the values into the vector.

- **int parms_VecAssembleElementVector(FLOAT *vec, int m, int *im, FLOAT *values, INSERTMODE mode, parms_Map map)**
  Completes setting up values for the distributed vector *vec*. A preceding call to parms_VecSetElementVector() should have been made prior to calling this function.

12 pARMS Viewer Object

12.1 parms_Viewer File Reference:

Functions related to parms_Viewer object.

12.2 Function Details

- **int parms_ViewerCreate(parms_Viewer *v, char *filename)**
  Create a parms_Viewer object. If PARMS_COUT and PARMS_CERR are input as filename, they stand for standard output and standard error, respectively. Otherwise, each processor creates a file "fnameID.dat", where ID is the ID of processor.
- **int parms_ViewerFree(parms_Viewer *v)**
  Free the memory for the parms_Viewer object.
- **int parms_ViewerGetName(parms_Viewer v, char **fname)**
  Retrieve the filename.
- **int parms_ViewerGetFP(parms_Viewer v, char **fp)**
  Get a pointer to the file pointer.
- **int parms_ViewerStoreFP(parms_Viewer v, FILE *fp)**
  Store fp to the parms_Viewer object.

13 FORTRAN

Fortran wrappers have been provided for each of the above pARMS objects. The calling sequence is the same as above, except with the additional standard fortran argument - $ierr$. This is typically the last argument in each of the wrapper functions except for the parms_Map object functions : parms_mapgetglobalsize(parms_Map *self, int *gsize); parms_mapgetlocalsize(parms_Map *self, int *lsize); parms_mapgetnumprocs(parms_Map *self, int *numpro); parms_mapgetglobaltolocal(parms_Map *self, int *gindex, int *lindex) and parms_mapgetpid(parms_Map *self, int *pid), where the ierr takes on the value returned by the function - (in this case gsize, lsize, numpro, lindex, and pid respectively).