PaStiX version 5.2 Quick Reference Guide

November 22, 2012

Calling PaStiX with a global matrix

#include ``pastix.h''

```
void pastix ( pastix_data_t  ** pastix_data, MPI_Comm        pastix_comm,
              pastix_int_t      n,           pastix_int_t  * colptr,
              pastix_int_t    * row,         pastix_float_t * avals,
              pastix_int_t    * perm,        pastix_int_t  * invp,
              pastix_float_t  * b,           pastix_int_t    rhs,
              pastix_int_t    * iparm,       double        * dparm );
```

#include ``pastix_fortran.h''

```
pastix_data_ptr_t  :: pastix_data
integer            :: pastix_comm
pastix_int_t       :: n, rhs, ia(n), ja(nnz)
pastix_float_t     :: avals(nnz), b(n)
pastix_int_t       :: perm(n), invp(n), iparm(64)
real*8             :: dparm(64)

call pastix_fortran ( pastix_data, pastix_comm, n, ia, ja, avals,
                      perm, invp, b, rhs, iparm, dparm )
```

pastix_data	Area used to store information between calls. Should be given as NULL for first call.
pastix_comm	MPI communicator used to solve the system.
n	Matrix dimension.
nnz	Number of non-zeros.
colptr, row, avals	Matrix in CSC format (see example below).
perm	Permutation vector.
invp	Inverse permutation vector.
b	Right-hand side(s) and solution(s) as output.
rhs	Number of right-hand side(s).
iparm	Vector of integer parameters.
dparm	Vector of real parameters.

In the current release, the matrix must be given in Compressed Sparse Column format in Fortran numbering (starts from 1).

CSC matrix example : $\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 2 & 0 & 5 & 0 & 0 \\ 0 & 4 & 6 & 7 & 0 \\ 0 & 0 & 0 & 0 & 8 \end{pmatrix}$

colptr $= \{1, 3, 5, 7, 8, 9\}$
row $= \{1, 3, 2, 4, 3, 4, 4, 5\}$
avals $= \{1, 2, 3, 4, 5, 6, 7, 8\}$

Calling PaStiX with a local matrix

#include ``pastix.h''

```
void dpastix ( pastix_data_t  ** pastix_data, MPI_Comm        pastix_comm,
               pastix_int_t      n,           pastix_int_t  * colptr,
               pastix_int_t    * row,         pastix_float_t * avals,
               pastix_int_t    * loc2glb,
               pastix_int_t    * perm,        pastix_int_t  * invp,
               pastix_float_t  * b,           pastix_int_t    rhs,
               pastix_int_t    * iparm,       double        * dparm );
```

#include ``pastix_fortran.h''

```
pastix_data_ptr_t  :: pastix_data
integer            :: pastix_comm
pastix_int_t       :: n, rhs, ia(n+1), ja(nnz)
pastix_float_t     :: avals(nnz), b(n)
pastix_int_t       :: loc2glb(n), perm(n), invp(n), iparm(64)
real*8             :: dparm(64)

call dpastix_fortran ( pastix_data, pastix_comm, n, ia, ja, avals,
                       loc2glob perm, invp, b, rhs, iparm, dparm )
```

Additional parameter :

loc2glb	Local to global column number correspondance, all columns must be distributed once and loc2glob must be ordered increasingly.

The distribution of the CSC matrix is given through the loc2glb vector (see example below).

dCSC matrix example :

$$\begin{pmatrix} P_1 & P_2 & P_1 & P_2 & P_1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 2 & 0 & 5 & 0 & 0 \\ 0 & 4 & 6 & 7 & 0 \\ 0 & 0 & 0 & 0 & 8 \end{pmatrix}$$

On processor one :
colptr $= \{1, 3, 5, 6\}$
row $= \{1, 3, 3, 4, 5\}$
avals $= \{1, 2, 5, 6, 8\}$
loc2glb $= \{1, 3, 5\}$

On processor two :
colptr $= \{1, 3, 4\}$
row $= \{2, 4, 4\}$
avals $= \{3, 4, 7\}$
loc2glb $= \{2, 4\}$

Integer and real parameters (iparm and dparm)

Integer parameters and outputs.				
Keyword	Index	Definition	Default	IN/OUT
IPARM_MODIFY_PARAMETER	0	Indicate if parameters have been set by user	API_YES	IN
IPARM_START_TASK	1	Indicate the first step to execute (see PaStiX steps)	API_TASK_ORDERING	IN
IPARM_END_TASK	2	Indicate the last step to execute (see PaStiX steps)	API_TASK_CLEAN	IN
IPARM_VERBOSE	3	Verbose mode (see Verbose modes)	API_VERBOSE_NO	IN
IPARM_DOF_NBR	4	Degree of freedom per node	1	IN
IPARM_ITERMAX	5	Maximum iteration number for refinement	250	IN
IPARM_MATRIX_VERIFICATION	6	Check the input matrix	API_NO	IN
IPARM_ONLY_RAFF	8	Refinement only	API_NO	IN
IPARM_CSCD_CORRECT	9	Indicate if the cscd has been redistributed after blend	API_NO	IN
IPARM_NBITER	10	Number of iterations performed in refinement	-	OUT
IPARM_TRACEFMT	11	Trace format (see Trace modes)	API_TRACE_PICL	IN
IPARM_GRAPHDIST	12	Specify if the given graph is distributed or not	API_YES	IN
IPARM_AMALGAMATION_LEVEL	13	Amalgamation level	5	IN
IPARM_ORDERING	14	Choose ordering	API_ORDER_SCOTCH	IN
IPARM_DEFAULT_ORDERING	15	Use default ordering parameters with Scotch or Metis	API_YES	IN
IPARM_ORDERING_SWITCH_LEVEL	16	Ordering switch level (see Scotch User's Guide)	120	IN
IPARM_ORDERING_CMIN	17	Ordering cmin parameter (see Scotch User's Guide)	0	IN
IPARM_ORDERING_CMAX	18	Ordering cmax parameter (see Scotch User's Guide)	100000	IN
IPARM_ORDERING_FRAT	19	Ordering frat parameter (see Scotch User's Guide)	8	IN
IPARM_STATIC_PIVOTING	20	Static pivoting	-	OUT
IPARM_METIS_PFACTOR	21	Metis pfactor	0	IN
IPARM_NNZEROS	22	Number of nonzero entries in the factorized matrix	-	OUT
IPARM_ALLOCATED_TERMS	23	Maximum memory allocated for matrix terms	-	OUT
IPARM_BASEVAL	24	Baseval used for the matrix	0	IN
IPARM_MIN_BLOCKSIZE	25	Minimum block size	60	IN
IPARM_MAX_BLOCKSIZE	26	Maximum block size	120	IN
IPARM_SCHUR	27	Schur mode	API_NO	IN
IPARM_ISOLATE_ZEROS	28	Isolate null diagonal terms at the end of the matrix	API_NO	IN
IPARM_RHSD_CHECK	29	Set to API_NO to avoid RHS redistribution	API_YES	IN
IPARM_FACTORIZATION	30	Factorization mode (see Factorization modes)	API_FACT_LDLT	IN
IPARM_NNZEROS_BLOCK_LOCAL	31	Number of nonzero entries in the local block factorized matrix	-	OUT
IPARM_CPU_BY_NODE	32	Number of CPUs per SMP node	0	IN
IPARM_BINDTHRD	33	Thread binding mode (see Thread binding modes)	API_BIND_AUTO	IN
IPARM_THREAD_NBR	34	Number of threads per MPI process	1	IN
IPARM_LEVEL_OF_FILL	36	Level of fill for incomplete factorization	1	IN
IPARM_IO_STRATEGY	37	IO strategy (see Checkpoints modes)	API_IO_NO	IN
IPARM_RHS_MAKING	38	Right-hand-side making (see Right-hand-side modes)	API_RHS_B	IN
IPARM_REFINEMENT	39	Refinement type (see Refinement modes)	API_RAF_GMRES	IN
IPARM_SYM	40	Symmetric matrix mode (see Symmetric modes)	API_SYM_YES	IN
IPARM_INCOMPLETE	41	Incomplete factorization	API_NO	IN
IPARM_ABS	42	ABS level (Automatic Blocksize Splitting)	1	IN
IPARM_ESP	43	ESP (Enhanced Sparse Parallelism)	API_NO	IN
IPARM_GMRES_IM	44	GMRES restart parameter	25	IN
IPARM_FREE_CSCUSER	45	Free user CSC	API_CSC_PRESERVE	IN
IPARM_FREE_CSCPASTIX	46	Free internal CSC (Use only without call to Refin. step)	API_CSC_PRESERVE	IN
IPARM_OOC_LIMIT	47	Out of core memory limit (Mo)	2000	IN
IPARM_THREAD_COMM_MODE	51	Threaded communication mode (see Communication modes)	API_THREAD_MULT	IN

Continued on next page ...

... continued from previous page				
Keyword	Index	Definition	Default	IN/OUT
IPARM_NB_THREAD_COMM	52	Number of thread(s) for communication	1	IN
IPARM_INERTIA	54	Return the inertia (symmetric matrix without pivoting)	-	OUT
IPARM_ESP_NBTASKS	55	Return the number of tasks generated by ESP	-	OUT
IPARM_ESP_THRESHOLD	56	Minimal block sizee to switch in ESP mode (128 * 128)	16384	IN
IPARM_DOF_COST	57	Degree of freedom for cost computation (If different from IPARM_DOF_NBR)	0	IN
IPARM_MURGE_REFINEMENT	58	Enable refinement in MURGE	API_YES	IN
IPARM_STARPU	59	Use StarPU runtime	API_NO	IN
IPARM_AUTOSPLIT_COMM	60	Automaticaly split communicator to have one MPI task by node	API_NO	IN
IPARM_PID	62	Pid of the first process (used for naming the log directory)	-1	OUT
IPARM_ERROR_NUMBER	63	Return value	-	OUT
IPARM_CUDA_NBR	64	Number of cuda devices	0	IN
IPARM_TRANSPOSE_SOLVE	65	Use transposed matrix during solve	API_NO	IN

Floating point parameters and outputs.				
Keyword	Index	Definition	Default	IN/OUT
DPARM_FILL_IN	1	Fill-in	-	OUT
DPARM_MEM_MAX	2	Maximum memory (-DMEMORY_USAGE)	-	OUT
DPARM_EPSILON_REFINEMENT	5	Epsilon for refinement	$1e^{-12}$	IN
DPARM_RELATIVE_ERROR	6	Relative backward error	-	OUT
DPARM_EPSILON_MAGN_CTRL	10	Epsilon for magnitude control	$1e^{-31}$	IN
DPARM_ANALYZE_TIME	18	Time for Analyse step (wallclock)	-	OUT
DPARM_PRED_FACT_TIME	19	Predicted factorization time	-	OUT
DPARM_FACT_TIME	20	Time for Numerical Factorization step (wallclock)	-	OUT
DPARM_SOLV_TIME	21	Time for Solve step (wallclock)	-	OUT
DPARM_FACT_FLOPS	22	Numerical Factorization flops (rate!)	-	OUT
DPARM_SOLV_FLOPS	23	Solve flops (rate!)	-	OUT
DPARM_RAFF_TIME	24	Time for Refinement step (wallclock)	-	OUT

PaStiX API : Macros

PaStiX step modes (index IPARM_START_TASK and IPARM_END_TASK)		
API_TASK_INIT	0	Set default parameters
API_TASK_ORDERING	1	Ordering
API_TASK_SYMBFACT	2	Symbolic factorization
API_TASK_ANALYSE	3	Tasks mapping and scheduling
API_TASK_NUMFACT	4	Numerical factorization
API_TASK_SOLVE	5	Numerical solve
API_TASK_REFINE	6	Numerical refinement
API_TASK_CLEAN	7	Clean

Boolean modes (All boolean except IPARM_SYM)		
API_NO	0	No
API_YES	1	Yes

Symmetric modes (index IPARM_SYM)		
API_SYM_YES	0	Symmetric matrix
API_SYM_NO	1	Nonsymmetric matrix
API_SYM_HER	2	Hermitian

Factorization modes (index IPARM_FACTORISATION)		
API_FACT_LLT	0	LL^t Factorization
API_FACT_LDLT	1	LDL^t Factorization
API_FACT_LU	2	LU Factorization
API_FACT_LDLH	3	LDL^h hermitian factorization

Verbose modes (index IPARM_VERBOSE)		
API_VERBOSE_NOT	0	Silent mode, no messages
API_VERBOSE_NO	1	Some messages
API_VERBOSE_YES	2	Many messages
API_VERBOSE_CHATTERBOX	3	Like a gossip
API_VERBOSE_UNBEARABLE	4	Really talking too much...

Check-points modes (index IPARM_IO)		
API_IO_NO	0	No output or input
API_IO_LOAD	1	Load ordering during ordering step and symbol matrix instead of symbolic factorisation.
API_IO_SAVE	2	Save ordering during ordering step and symbol matrix instead of symbolic factorisation.
API_IO_LOAD_GRAPH	4	Load graph during ordering step.
API_IO_SAVE_GRAPH	8	Save graph during ordering step.
API_IO_LOAD_CSC	16	Load CSC(d) during ordering step.
API_IO_SAVE_CSC	32	Save CSC(d) during ordering step.

Right-hand-side modes (index IPARM_RHS)		
API_RHS_B	0	User's right hand side
API_RHS_1	1	$\forall i, X_i = 1$
API_RHS_I	2	$\forall i, X_i = i$
API_RHS_0	3	

Refinement modes (index IPARM_REFINEMENT)		
API_RAF_GMRES	0	GMRES
API_RAF_PIVOT	1	Iterative Refinement (only for LU factorization)
API_RAF_GRAD	1	Conjugate Gradient (LL^t or LDL^t factorization)

Comunication modes (index IPARM_NB_THREAD_COMM)		
API_THREAD_MULTIPLE	1	All threads communicate.
API_THREAD_FUNNELED	2	One thread perform all the MPI Calls.
API_THREAD_COMM_ONE	4	One dedicated communication thread will receive messages.
API_THREAD_COMM_DEFINED	8	Then number of threads receiving the messages is given by IPARM_NB_THREAD_COMM.
API_THREAD_COMM_NBPROC	16	One communication thread per computation thread will receive messages.

Trace modes (index IPARM_TRACEFMT)		
API_TRACE_PICL	0	Use PICL trace format
API_TRACE_PAJE	1	Use Paje trace format
API_TRACE_HUMREAD	2	Use human-readable text trace format
API_TRACE_UNFORMATED	3	Unformated trace format

CSC Management modes (index IPARM_FREE_CSCUSER and IPARM_FREE_CSCPASTIX)		
API_CSC_PRESERVE	0	Do not free the CSC

Ordering modes (index IPARM_ORDERING)		
API_ORDER_SCOTCH	0	Use Scotch ordering

CSC Management modes (index IPARM_FREE_CSCUSER and IPARM_FREE_CSCPASTIX)		
API_CSC_FREE	1	Free the CSC when no longer needed

Ordering modes (index IPARM_ORDERING)		
API_ORDER_METIS	1	Use Metis ordering
API_ORDER_PERSONAL	2	Apply user's permutation
API_ORDER_LOAD	3	Load ordering from disk

Thread-binding modes (index IPARM_BINTHRD)		
API_BIND_NO	0	Do not bind thread
API_BIND_AUTO	1	Default binding
API_BIND_TAB	2	Use vector given by pastix_setBind

Ordering modes (index IPARM_ORDERING)		
API_REALSINGLE	0	Use Scotch ordering
API_REALDOUBLE	1	Use Metis ordering
API_COMPLEXSINGLE	2	Apply user's permutation
API_COMPLEXDOUBLE	3	Load ordering from disk

Getting local node information

These functions are called when PaStiX is used with a distributed matrix.

```
pastix_int_t pastix_getLocalNodeNbr ( pastix_data_t ** pastix_data );
```

`pastix_data`	Area used to store information between calls.

Return the node number in the new distribution computed by the analyze step (Analyze step must have already been executed).

```
int pastix_getLocalNodeLst ( pastix_data_t ** pastix_data,
                             pastix_int_t   *  nodelst );
```

`pastix_data`	Area used to store information between calls.
`nodelst`	Array to receive the list of local nodes.

Fill `nodelst` with the list of local nodes (`nodelst` must be at least `nodenbr*sizeof(pastix_int_t)`, where `nodenbr` is obtained from `pastix_getLocalNodeNbr`).

Binding threads

```
void pastix_setBind ( pastix_data_t ** pastix_data, int thrdnbr,
                      int           *  bindtab );
```

`pastix_data`	Area used to store information between calls.
`thrdnbr`	Number of threads (== length of `bindtab`).
`bindtab`	List of processors for threads to be binded on.

Assign threads to processors.

Checking the CSC

```
void pastix_checkMatrix ( MPI_Comm        pastix_comm, int               verb,
                          int             flagsym,     int               flagcor,
                          pastix_int_t    n,           pastix_int_t   ** colptr,
                          pastix_int_t ** row,         pastix_float_t ** avals,
                          pastix_int_t ** loc2glob );  int               dof
```

`pastix_comm`	PaStiX MPI communicator.
`verb`	Verbose mode (see Verbose modes).
`flagsym`	Indicates if the matrix is symmetric (see Symmetric modes).
`flagcor`	Indicates if the matrix can be reallocated (see Boolean modes).
`n`	Matrix dimension.
`colptr, row, avals`	Matrix in CSC format.
`loc2glb`	Local to global column number correspondance.

Check and correct the user matrix in CSC format.

Checking the symmetry of a CSCD

```
int cscd_checksym ( pastix_int_t   n,    pastix_int_t * ia,
                    pastix_int_t * ja,   pastix_int_t * l2g,
                    MPI_Comm       comm );
```

`n`	Number of local columns.
`ia`	Starting index of each column in `ja`.
`ja`	Row of each element.
`l2g`	Global column numbers of local columns.

Check the graph symmetry.

Correcting the symmetry of a CSCD

```
int cscd_symgraph ( pastix_int_t    n,     pastix_int_t    *  ia,
                    pastix_int_t *  ja,    pastix_float_t  *  a,
                    pastix_int_t *  newn,  pastix_int_t    ** newia,
                    pastix_int_t ** newja, pastix_float_t  ** newa,
                    pastix_int_t *  l2g,   MPI_Comm           comm,
```

`n`	Number of local columns.
`ia`	Starting index of each column in `ja` and `a`.
`ja`	Row of each element.
`a`	Value of each element.
`newn`	New number of local columns.
`newia`	Starting index of each columns in `newja` and `newa`.
`newja`	Row of each element.
`newa`	Values of each element.
`l2g`	Global number of each local column.
`comm`	MPI communicator.

Symmetrize the graph.

Adding a CSCD into an other one

```
int cscd_addlocal ( pastix_int_t         n,      pastix_int_t   *  ia,
                    pastix_int_t       * ja,     pastix_float_t *  a,
                    pastix_int_t       * l2g,    pastix_int_t      addn,
                    pastix_int_t       * addia,  pastix_int_t   *  addja,
                    pastix_float_t     * adda,   pastix_int_t   *  addl2g,
                    pastix_int_t       * newn,   pastix_int_t   ** newia,
                    pastix_int_t      ** newja,  pastix_float_t ** newa
                    CSCD_OPERATIONS_t    OP );
```

n	Size of first CSCD matrix (same as newn).
ia	Column starting positions in first CSCD matrix.
ja	Rows in first CSCD matrix.
a	Values in first CSCD matrix (can be NULL).
l2g	Global column number map for first CSCD matrix.
addn	Size of the second CSCD matrix (to be added to base).
addia	Column starting positions in second CSCD matrix.
addja	Rows in second CSCD matrix.
adda	Values in second CSCD (can be NULL → add ø).
addl2g	Global column number map for second CSCD matrix.
newn	Size of output CSCD matrix (same as n).
newia	Column starting positions in output CSCD matrix.
newja	Rows in output CSCD matrix.
newa	Values in outpur CSCD matrix.
malloc_flag	Flag: Function call is internal to PaStiX.
OP	Specifies treatment of overlapping CSCD elements.

Adds CSCD matrix adda to a, producing newa (allocated in the function).
The operation OP can be : CSCD_ADD, CSCD_KEEP, CSCD_MAX, CSCD_MIN, and CSCD_OVW (overwrite).

Building a CSCD from a CSC

```
void csc_dispatch ( pastix_int_t         gN,         pastix_int_t   *  gcolptr,
                    pastix_int_t       * grow,       pastix_float_t *  gavals,
                    pastix_float_t     * grhs,       pastix_int_t   *  gperm,
                    pastix_int_t       * ginvp,
                    pastix_int_t       * lN,         pastix_int_t   ** lcolptr,
                    pastix_int_t      ** lrow,       pastix_float_t ** lavals,
                    pastix_float_t    ** lrhs,       pastix_int_t   ** lperm,
                    pastix_int_t      ** loc2glob,   int               dispatch,
                    MPI_Comm             pastix_comm );
```

gN	Global CSC matrix number of columns.
gcolptr, grows, gavals	Global CSC matrix
gperm	Permutation table for global CSC matrix.
ginvp	Inverse permutation table for global CSC matrix.
lN	Local number of columns (output).
lcolptr, lrows, lavals	Local CSCD matrix (output).
lrhs	Local part of the right hand side (output).
lperm	Local part of the permutation table (output).
loc2glob	Global numbers of local columns (before permutation).
dispatch	Dispatching mode: CSC_DISP_SIMPLE Cut in n_{proc} parts of consecutive columns CSC_DISP_CYCLIC Use a cyclic distribution.
pastix_comm	PaStiX MPI communicator.

Distribute a CSC into a CSCD.

Redistributing a CSCd

```
int cscd_redispatch ( pastix_int_t      n,     pastix_int_t   *  ia,
                      pastix_int_t    * ja,    pastix_float_t *  a,
                      pastix_float_t  * rhs,   pastix_int_t   *  l2g,
                      pastix_int_t      dn,    pastix_int_t   ** dia,
                      pastix_int_t   ** dja,   pastix_float_t ** da,
                      pastix_float_t ** drhs,  pastix_int_t   *  dl2g,
                      MPI_Comm          comm);
```

n	Number of local columns
ia	First cscd starting index of each column in ja and a
ja	Row of each element in first CSCD
a	Value of each CSCD in first CSCD (can be NULL)
rhs	Right-hand-side member corresponding to the first CSCD (can be NULL)
l2g	Local to global column numbers for first CSCD
dn	Number of local columns
dia	New CSCD starting index of each column in ja and a
dja	Row of each element in new CSCD
da	Value of each CSCD in new CSCD
rhs	Right-hand-side member corresponding to the new CSCD
dl2g	Local to global column numbers for new CSCD
comm	MPI communicator

Redistribute the first cscd, distributed with l2g local to global array, into a new one using dl2g as local to global array.

PaStiX API : Murge Interface

Description

Murge is a common interface definition to multiple solver. It has been initiated by HIPS and PaStiX solvers developpers in january 2009.

A documentation about this new interface can be found at `http://murge.gforge.inria.fr/`.

Few function were added specificaly to PaStiX implementation of murge.

PaStiX specific implementation: Analyze step

INTS `MURGE_Analyze` (**INTS** `id`);

`id`	Solver instance identification number.

Perform matrix analyze:

- Compute a new ordering of the unknows
- Compute the symbolic factorisation of the matrix
- Distribute column blocks and computation on processors

This function is not needed to use Murge interface, it only forces analyze step when user wants.

If this function is not used, analyze step will be performed when getting new distribution from MURGE, or filling the matrix.

PaStiX specific implementation: Factorization step

INTS `MURGE_Factorize` (**INTS** `id`);

`id`	Solver instance identification number.

Perform matrix factorization.

This function is not needed to use Murge interface, it only forces factorization when user wants.

If this function is not used, factorization will be performed with solve, when getting solution from MURGE.

PaStiX specific implementation: Assembly sequences

INTS `MURGE_AssemblySetSequence` (**INTS** `id` , **INTL** `coefnbr`, **INTS** * `ROWs`, **INTS** * `COLs`, **INTS** `op`, **INTS** `op2`, **INTS** `mode`, **INTS** `nodes`, **INTS** * `id_seq`);

`id`	Solver instance identification number.
`coefnbr`	Number of local entries in the sequence.
`ROWs`	List of rows of the sequence.
`COLs`	List of columns of the sequence.
`op`	Operation to perform for coefficient which appear several tim (see `MURGE_ASSEMBLY_OP`).
`op2`	Operation to perform when a coefficient is set by two different processors (see `MURGE_ASSEMBLY_OP`).
`mode`	Indicates if user ensure he will respect solvers distribution (see `MURGE_ASSEMBLY_MODE`).
`nodes`	Indicate if entries are given one by one or by node : 0 : entries are entered value by value, 1 : entries are entries node by node.
`id_seq`	Sequence ID.

Create a sequence of entries to build a matrix and store it for being reused.

INTS `MURGE_AssemblyUseSequence` (**INTS** `id` , **INTS** `id_seq`, **COEF** * `values`);

`id`	Solver instance identification number.
`id_seq`	Sequence ID.
`values`	Values to insert in the matrix.

Assembly the matrix using a stored sequence.

INTS `MURGE_AssemblyDeleteSequence` (**INTS** `id` , **INTS** `id_seq`);

`id`	Solver instance identification number.
`id_seq`	Sequence ID.

Destroy an assembly sequence.

How-to compile PaStiX

Requirements

The PaStiX team recommends that you get the Scotch (`http://gforge.inria.fr/projects/scotch/`) and compile it.
Then go into PaStiX directory. Select the config file corresponding to your machine in `${PASTIX_DIR}/config/` and copy it to `${PASTIX_DIR}/config.in`.
Now edit this file, select the options you want, and set the correct path for `${SCOTCH_HOME}`.
If you want to use Metis, you also have to compile it and edit the path in `config.in`.

Compilation

Makefile tags (from the root directory)	
`make help`	print this help
`make all`	build PaStiX library
`make debug`	build PaStiX library in debug mode
`make drivers`	build matrix drivers library
`make debug drivers`	build matrix drivers library in debug mode
`make examples`	build examples (will run `'make all'` and `'drivers'` if required)
`make murge`	build Murge examples
`make python`	Build python wrapper and run an example
`make clean`	remove all binaries and objects directories
`make cleanall`	remove all binaries, objects and dependencies directories

Compilation options (`config.in`)

General options	
`-DDISTRIBUTED`	Enable distributed mode `dpastix` (PT-Scotch required)
`-DFORCE_LONG`	Use long integers
`-DFORCE_DOUBLE`	Use double floating coefficients
`-DFORCE_COMPLEX`	Use complex coefficients
`-DFORCE_NOMPI`	Compile without MPI support
`-DFORCE_NOSMP`	Compile without Thread support

Preprocessing options	
`-DMETIS`	Use Metis ordering library (needs `-L${METIS_HOME} -lmetis`)
`-DWITH_SCOTCH`	Activate Scotch ordering library

Solver options - *See $PASTIX_HOME/sopalin/src/sopalin_define.h*	
`-DNUMA_ALLOC`	Allocate the coefficient vector locally on each thread.
`-DNO_MPI_TYPE`	Copy into communication buffers to avoid using MPI types.
`-DTEST_IRECV`	Use nonblocking receives
`-DTHREAD_COMM`	Receive on dedicated threads (persistent communications).
`-DPASTIX_FUNNELED`	Use main thread for all communications.

Statistics and Debug options - *See $PASTIX_HOME/sopalin/src/sopalin_define.h*	
`-DMEMORY_USAGE`	Show memory allocations (may slow down execution)
`-DSTATS_SOPALIN`	Show parallelization memory overhead
`-DVERIF_MPI`	Check MPI Communications for success
`-DFLAG_ASSERT`	Adds some checks during factorization

Checkpoints in PaStiX

You can save ordering and solver structures on disk to start directly from step 3 (Tasks Mapping and Scheduling) when launching PaStiX again.
Set `iparm[IPARM_IO_STRATEGY]` to `API_IO_SAVE` and call step 1 (Ordering) and 2 (Symbolic Factorization). This will create two files, `ordergen` and `symbolgen` in the working directory.
Copy (or move, or link) `ordergen` and `symbolgen` to `ordername` and `symbolname`.
Set `iparm[IPARM_IO_STRATEGY]` to `API_IO_LOAD` and then call PaStiX again from step 3.

Out-Of-Core in PaStiX

An out-of-core version of PaStiX is under development.
You will then have to set `iparm[IPARM_OOC_LIMIT]` to fix the memory limit size.

OOC compilation options	
`-DOOC`	Simple OOC without contribution buffer management
`-DOOC_FTGT`	OOC with contribution buffer management
`-DOOC_CLOCK`	Compute time spent waiting for data to be loaded

Dynamic Scheduling in PaStiX

Solver scheduling strategy - *Static scheduling used by default*	
`-DPASTIX_DYNSCHED`	Dynamic scheduling

Using StarPU in PaStiX

Using `StarPU` in PaStiX	
`-DWITH_STARPU`	Enable `StarPU`, needs `IPARM_STARPU` to be set to `API_YES`
`-DFORCE_NO_CUDA`	Disable `CUDA` kernels (only LL^t and LU `GEMM` provided)

Splitting communicators in PaStiX

One can run PaStiX on a communicator and get sequential and `MPI+Threads` parts runned on one MPI task per node and one thread by processor, `MPI` only parts runned on the whole communicator using `IPARM_AUTOSPLIT_COMM`.

Options linked to `IPARM_AUTOSPLIT_COMM`	
`-DWITH_SEM_BARRIER`	Semaphore barrier on idle `MPI` entity (less CPU consuming)

Multiple Arithmetic in PaStiX

default	simple	double	simple complex	double complex
`pastix`	`s_pastix`	`d_pastix`	`c_pastix`	`z_pastix`
`dpastix`	`s_dpastix`	`d_dpastix`	`c_dpastix`	`z_dpastix`
`<function>`	`s_<function>`	`d_<function>`	`c_<function>`	`z_<function>`