PaStiX version 5.1 Quick Reference Guide

April 17, 2009

PaStiX Calls with 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`	Data structure used to keep informations for a step by step call. Should be given unallocated 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`	Integer parameter vector.
`dparm`	Double parameter vector.

In the current release, the matrix must be given in a Compress 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	=	$\{1, 2, 3, 4, 5, 6, 7, 8\}$

PaStiX Calls with distributed 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             mpi_comm
  pastix_int_t        n, rhs, ia(n), 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, mpi_comm, n, ia, ja, avals,
                     loc2glb, perm, invp, b, rhs, iparm, dparm)
```

Additional parameter :

`loc2glb`	Local to global column number correspondance.

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	=	$\{2, 4\}$

Floating and Integer parameters (dparm and iparm)

Keyword	Index	Definition	Default	IN/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	Analize time	-	OUT
DPARM_PRED_FACT_TIME	19	Predicted factorization time	-	OUT
DPARM_FACT_TIME	20	Factorization time	-	OUT
DPARM_SOLV_TIME	21	Solving time	-	OUT
DPARM_FACT_FLOPS	22	Factorization flops	-	OUT
DPARM_SOLV_FLOPS	23	Solving flops	-	OUT
DPARM_RAFF_TIME	24	Refinement time	-	OUT

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 Non Zeros in initial matrix	-	OUT
`IPARM_ALLOCATED_TERMS`	23	Number of Non Zeros in the factorized matrix	-	OUT
`IPARM_BASEVAL`	24	Baseval used for the matrix	0	IN
`IPARM_MIN_BLOCKSIZE`	25	Minimum blocksize	60	OUT
`IPARM_MAX_BLOCKSIZE`	26	Maximum blocksize	120	OUT
`IPARM_FACTORIZATION`	30	Factorization mode (see Factorization modes)	`API_FACT_LDLT`	IN
`IPARM_CPU_BY_NODE`	32	Number of CPU 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 thread 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 Check-points modes)	`API_IO_NO`	IN
`IPARM_RHS_MAKING`	38	Right-hand-side making (see Rhight-hand-side modes)	`API_RHS_1`	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 (Automatic Blocksize Splitting)	`API_NO`	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_THCOMM_DISABLED`	IN
`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_ERROR_NUMBER`	63	Return value	-	OUT

PaStiX API : Macros

PaStiX steps 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

Symetric modes (index `IPARM_SYM`)		
`API_SYM_YES`	0	Symmetric matrix
`API_SYM_NO`	1	Non Symmetric matrix

Factorization modes (index `IPARM_FACTORISATION_TYPE`)		
`API_FACT_LLT`	0	LL^t Factorization
`API_FACT_LDLT`	1	LDL^t Factorization
`API_FACT_LU`	2	LU Factorization

Verbose modes (index `IPARM_VERBOSE`)		
`API_VERBOSE_NOT`	0	No display
`API_VERBOSE_NO`	1	Some displays
`API_VERBOSE_YES`	2	Many displays

Check-points modes (index `IPARM_IO`)		
`API_IO_NO`	0	No output/input
`API_IO_LOAD`	1	Load data during preprocessing steps
`API_IO_SAVE`	2	Save data during preprocessing steps

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	$\forall i, X_i = i$

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

Comunication modes (index `IPARM_NB_THREAD_COMM`)		
`API_THCOMM_DISABLED`	0	No thread dedicated to communications
`API_THCOMM_ONE`	1	One thread dedicated to communications
`API_THCOMM_DEFINED`	2	Given by `IPARM_NB_THREAD_COMM`
`API_THCOMM_NBPROC`	3	One communication thread per computation thread

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 Text trace format
`API_TRACE_UNFORMATED`	3	Use 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 not needed anymore

Ordering modes (index `IPARM_ORDERING`)		
`API_ORDER_METIS`	1	Use Metis ordering
`API_ORDER_PERSONAL`	2	Use given permutation (resp. reverse permutation) array
`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

Getting local node informations

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

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

`pastix_data`	Data used for a step by step execution.

Return the node number in the new distribution computed by blend.
Needs blend to be runned with pastix_data before.

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

`pastix_data`	Data used for a step by step execution.
`nodelst`	An array where to write the list of local nodes/columns.

Fill in nodelst with the list of local nodes/columns.
Needs nodelst to be allocated with `nodenbr*sizeof(pastix_int_t)`, where nodenbr has been computed by `pastix_getLocalNodeNbr`.

Binding threads

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

`pastix_data`	Data structure used to keep informations between calls.
`thrdnbr`	Number of threads (should be the size of bindtab).
`bindtab`	Mapping vector for binding threads on processors.

Gives to PaStiX the mapping vector for binding threads on 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);
```

`pastix_comm`	PaStiX MPI communicator.
`verb`	Verbose mode (see Verbose modes).
`flagsym`	Indicates if the matrix is symetric (see Symetric 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 symetry 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 columns in `ja`.
`ja`	Row of each element.
`l2g`	Global number of each local column.

Check the graph symetry.

Correcting the symetry 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 columns in `ja` and `a`.
`ja`	Row of each element.
`a`	Values 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.

Symetrize 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`	First cscd size.
`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).
`l2g`	Local to global column numbers for first cscd.
`addn`	CSCD to add size.
`addia`	CSCD to add starting index of each column in `addja` and `adda`.
`addja`	Row of each element in second CSCD.
`adda`	Value of each cscd in second CSCD (can be NULL -¿ add 0).
`addl2g`	Local to global column numbers for second cscd.
`newn`	New cscd size (same as first).
`newia`	CSCD to add starting index of each column in `newja` and `newwa`.
`newja`	Row of each element in third CSCD.
`newa`	Value of each cscd in third CSCD.
`malloc_flag`	Flag to indicate if function call is intern to pastix or extern.
`OP`	Operation to manage common CSCD coefficients.

Add the second CSCD to the first CSCD, result is stored in the third CSCD (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   **linvp,
                  pastix_int_t    **loc2glob,
                  pastix_int_t (*dispatch_function)(pastix_int_t ,
                                                    pastix_int_t ,
                                                    MPI_Comm),
                  MPI_Comm pastix_comm);
```

`gN`	Global number of columns.
`gcolptr`	Global starting index of each column in grows ans gavals.
`grows`	Global rows of each element.
`gavals`	Global values of each element.
`gperm`	Global permutation tabular.
`ginvp`	Global reverse permutation tabular.
`lN`	Local number of columns (output).
`lcolptr`	Starting index of each local column (output).
`lrowptr`	Row number of each local element (output).
`lavals`	Values of each local element (output).
`lrhs`	Local part of the right hand side (output).
`lperm`	Local part of the permutation tabular (output).
`loc2glob`	Global numbers of local columns (before permutation).
`dispatch_function`	Function giving owner of each column.
`pastix_comm`	PaStiX MPI communicator.

Distribute a CSC into a CSCD.

CSC distributing rules

```
pastix_int_t csc_simple_distribution(pastix_int_t column,
                                     pastix_int_t columnnbr,
                                     MPI_Comm     pastix_comm)
```

`column`	Column number to distribute.
`columnnbr`	Number of colmuns.
`pastix_comm`	PaStiX MPI communicator.

Distribute the CSC. First columns are on first processor and so on.

```
pastix_int_t csc_cyclic_distribution(pastix_int_t column,
                                     pastix_int_t columnnbr,
                                     MPI_Comm     pastix_comm)
```

`column`	Column number to distribute.
`columnnbr`	Number of colmuns.
`pastix_comm`	PaStiX MPI communicator.

Cyclicaly distribute the CSC.

How-to compile PaStiX

Requirements

The PaStiX team recommands you to get 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 clean`	clean to rebuild the library
`make expor`	compile the PaStiX library
`make debug`	compile the PaStiX library in debug mode
`make install`	install the PaStiX library ('`make expor`' or '`debug`' required)
`make test`	compile the C driver ('`make install`' required)
`make testf`	compile the Fortran driver ('`make install`' required)
`make examples`	compile the PaStiX examples ('`make install`' required)

`make murge_examples`	compile MURGE examples (only available in distributed mode -DDISTRIBUTED, '`make install`' required)

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`	Compilation without MPI support
`-DFORCE_NOSMP`	Compilation without Thread support

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

Solver options - *See $PASTIX_HOME/sopalin/src/sopalin_define.h*	
`-DNUMA_ALLOC`	Allocation of the coefficient vector locally on each thread.
`-DNO_MPI_TYPE`	Avoids MPI types usage by copying into communication buffers.
`-DTEST_IRECV`	Non blocking receptions
`-DTHREAD_COMM`	Reception on dedicated threads (persistent communications).
`-DPASTIX_FUNNELED`	Main thread makes all communications.

Statistics and Debug options - *See $PASTIX_HOME/sopalin/src/sopalin_define.h*	
`-DMEMORY_USAGE`	Shows memory allocations (could slow down execution)
`-DSTATS_SOPALIN`	Shows parallelization memory overhead
`-DVERIF_MPI`	Checks MPI Communication successful
`-DFLAG_ASSERT`	Adds some checks during factorization

Check points 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[37]` to `API_IO_SAVE` and call step 1 (Ordering) and 2 (Symbolic Factorization) of Pastix. It will generate two files, `ordergen` and `symbolgen` in the working directory.
Copy (or move, or link) `ordergen` and `symbolgen` to `ordername` and `symbolname`.
Set `iparm[37]` 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 developpement.
To use it, you must get the corresponding PaStiX development branch and compile it with the flag `-DOOC`.
To use OOC with contribution buffer, with MPI, set `-DOOC_FTGT` instead.
Then you have to set iparm[47] to the memory limit size and iparm[48] to the number of OOC thread(s) per computing thread (0 : the computing thread will read and write, very long).

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

Bubbles in PaStiX

It is possible to use Marcel thread library instead of `POSIX` threads.

Solver scheduling strategy - *Static scheduling used by default*	
`-DPASTIX_BUBBLE`	Dynamic scheduling
`-DPASTIX_USE_BUBBLE`	Dynamic scheduling with Marcel's bubble scheduler

Compile with `` `pm2-config --cflags` -DMARCEL `` and link with `` `pm2-config --libs` ``.

PaStiX launching scripts

Introduction

A set of perl scripts allow you to compile and execute PaStiX.
Those scripts are used for our tests on our machines but it can be adapted to other machines.

How to use it

First edit `Scripts/Modules/Common.pm` and copy the text in top between "`## Options Par defaut si Conf.pm n'existe pas`" and "`## Options Communes`" to a new file : "`Scripts/Modules/Conf.pm`".
In `Conf.pm` change the options as you want.

Compilation

The strings of the array `liste_executables` will determine the compilation options.

Option	Tag
-DFORCE_LONG	_long
-DFORCE_DOUBLE	_double
-DFORCE_COMPLEX	_complex
-DFORCE_NOMPI	_nompi
-DFORCE_NOSMP	_nosmp
-DMETIS	_metis
-DNUMA_ALLOC	_Numa
-DNO_MPI_TYPE	_notypes
-DTEST_IRECV	_IRecv
-DTHREAD_COMM	_ThComm
-DPASTIX_FUNNELED	_Funneled
Marcel Usage	_Marcel or _UseBubble
Marcel Flavor	_FL*flavorname*FL
-DPASTIX_BUBBLE	Bubble
-DPASTIX_USE_BUBBLE	_UseBubble
-DMEMORY_USAGE	_mem
-DSTATS_SOPALIN	_ovh
-DTRACE_SOPALIN	_trace
-DONLY_LOAD_SYMBMTX	_symbmtx
-DNOSMP_RAFF	_noSMPRaff
-DCOMPACT_SMX	_compact

Using the executable name, the config.in will be created from `/template/config.tpl`.
All `_KEYWORD_` in file config.tpl will be replaced by the value of `$param_exec{_KEYWORD_}` defined in `Scripts/Modules/Compilation.pl`.

Then you can run the script `Pastix.pl` to compile (`-c`).
If you have issues compiling you can edit the `Modules/Compilation.pl` file.
There you can add specific compilation options for your machine after the line :
"`# Adaptation des options de compilation  chaque machine`".

Creation of the directories for the execution

Before the execution, you have to run `Pastix -f`.
All files will be created into the results path specified in `Conf.pm`
The file tree follows this model :
`/executableName/MatriceName/AAP_BBT_levelCC_amalgEE/YYYYMMDD-HHMM/`, where `AA` is the number of MPI process, `BB` the number of thread, `CC` the level of fill, `EE` the level of amalgamation, and `YYYYMMDD-HHMM` the date and hour.

Execution

For execution, you have to run `Pastix -e`.
You may have to modify the function ExecuteJobs from `Pastix.pl`, probably after the line "`# Execution du job`" .
You may to have to add your machine to one of the curent `if` or `elif` or to had your proper one and maybe a new tpl to execute your job if your batch scheduler command is different from llsubmit, bsub or oarsub.

Browse results

To see your results, just run `GetAllRes.pl` (use `-h` to get available options).
If you have a mysql Database, you can also add results to the Database using `fillDB.pl`, for this ask for our DB.sql file and our (under developpement) results browsing website.