Introduction to PaStiX library

Introduction

The solver is currently evolving, and two versions are available. The first historical version is available on the Inria forge: `http://pastix.gforge.inria.fr`. This version will become deprecated to the benefit of the current version under development, and publicly available on `http://gitlab.inria.fr/solverstack/pastix`.
The main difference is the MPI support that is not yet integrated to the new PaStiX, but it includes all recent development on low-rank compression and runtime support for GPUs.
You can also access gitlab bug reports to ask questions or submit problems to the developer team.
PaStiX needs several libraries to be compiled and installed on your cluster:

- An implementation of the Message Passing Interface 2.0 standard like Mpich2 , OpenMPI , MVAPICH, ...
- An implementation of the Basic Linear Algebra Subroutines standard (BLAS) like OpenBLAS, ACML , MKL , ...
- A library to compute ordering in order to reduce fill-in and increase parallelism like Scotch , PT-Scotch or METIS . The user can also provide it's own ordering.
- The Hardware Locality library (HwLoc) is not required but highly recommended especially if PaStiX is compiled with threads support. It will give an uniform view of the architecture to the solver (Hide hyper-threading, linearize core numbering, ...).

Spack support is also provided to ease the compilation, and compile for you all the dependencies that you may require for PaStiX .

PaStiX 6.0

Compilation and installation

You might not want to do those steps, PaStiX has already been compiled for you.
Compilation and installation is now done through cmake, and requires you to have your environment correctly defined to make sure cmake will be able to find all required dependencies.

1. First, create a build directory: `mkdir build; cd build`. You can now configure the project: `cmake ..`, or `cmake src_dir` if you created the build directory somewhere else.
2. Some parameters can be modified by calling `ccmake .`
3. Compile with `make`, and eventually install with `make install`

Unfortunately, Spack support is not yet provided for this version, and will come soon with the alpha release.
To ease the tests for this training course, all libraries have been compiled and are available in `$NEWPASTIX_DIR` directory. PaStiX has been compiled with threads, StarPU, PaRSEC, MKL, Scotch , and METIS .

Basic testings

Be sure that you setup the correct environment and allocate some resources. Note that we need only one node for those experiments, so you can exit the previous allocation on multiple nodes, and do a new one with a single node.

```
source env-pastix.sh
qsub -I -l select=1:ncpus=16,walltime=01:00:00
```

To get the collection of examples provided by PaStiX (see `https://solverstack.gitlabpages.inria.fr/pastix/group__pastix__examples.html`): , make a copy of the files in your home directory

```
cp -r $NEWPASTIX_DIR/examples ~/newpastix
cd ~/newpastix
```

To run a simple example on a 2D Laplacian of size 1000, on one or more MPI processes:

```
./simple -lap 1000 -t 16
```

`-t 16` specifies that you want 16 threads.

To go further in the testings, a set of matrices is available in the `$MATRICES_DIR` directory. They use different storage formats. The examples can all read matrices in Harwell Boeing (`.rsa` or `.rua`), Matrix Market (`.mtx`), or IJV format (`.ijv`), just be careful to give the option that specifies the correct format for the file. You can also use a simple 2D Laplacian matrix generator to the test with the `-lap size` option. The `-h` option will give you all the options available in examples.

Choose ordering library: SCOTCH or METIS

It is possible to give to the PaStiX solver any pre-computed ordering by providing the permutation array of the unknowns. However, it is recommended to use the ordering step from the solver to reduce the fill-in of the matrix during the factorization and reduce the memory overhead of the solver. The graph partitioning library can be chosen at runtime if and only if the solver has been compiled with multiple choices. If it has been compiled with SCOTCH and METIS , both will be available at runtime.

Look the ordering impact on fill-in and factorization time.
Remark: with the examples `simple`, or `analyze`, you can switch from SCOTCH to METIS with `-ord metis` or `-ord scotch` option.

Low-rank compression

PaStiX 6.0 offers low-rank compression to either accelerate the factorization and solve steps, and/or reduce the memory peak of the solver. Three different schemes are possible through the parameter `iparm_compress_when`:

1. `PastixCompressNever` (0). This is the default behavior with no compression at all.
2. `PastixCompressBegin` (1). This is the behavior that will saves the most on the memory peak. However, it will slow down the factorization, and this phenomenon will increase when the block sizes are larger
3. `PastixCompressEnd` (2). This is the behavior that will help to save time.

Low -rank compression is also strongly dependent from other parameters:

1. `iparm_compress_method` the will define the family of kernels to use for the compression (SVD: `PastixCompressMethodSVD (0)`, or Rank-Revealing QR: `PastixCompressMethodRRQR (1)`)
2. `dparm_compress_tolerance` that defines the tolerance of the compression

3. `iparm_compress_min_width` that defines the minimal size of required to compress a supernode
4. `iparm_compress_min_height` that defines the minimal height required for an off-diagonal block to be compressed (when minimal width is respected)

Larger are the sizes of the blocks, more memory gain it will induce. However, with the Minimal Memory (Begin) scenario, it will also increase the complexity of the update kernels and slow down the computations. This is not the case for the Just-in-Time (End) scenario.
Look the low-rank compression impact on different matrices.

Runtime support

PaStiX 6.0 include support for internal schedulers, but also to use external runtime supports such as PaRSEC and StarPU to offer access to accelerators. You can test the runtime support by using the `-s` option which takes as value: 0 for sequential scheduling, 1 for internal static scheduling, 2 for PaRSEC, and 3 for StarPU.

Look the impact of using a runtime support on large and small matrices n the factorization time.

Step by step

many steps compose the PaStiX solver (see `https://solverstack.gitlabpages.inria.fr/pastix/group__pastix__users.html`):

1. Initialization of the solver which starts runtimes, and internal scheduler
2. Graph Ordering
3. Symbolic Factorization
4. analyze
5. Factorization
6. Solve
7. Refinement
8. Free data structure

Each of them can be called separately. This is important to simulation that factorizes the matrix once for multiple solves for examples. The factorization, solve and refinement steps can be called multiple times if your structure doesn't change or if only your right hand side changes at each iteration.

Look at the source code of the step-by-step to try different combinations.