CHAMELEON User's Guide

Software of MORSE project

A dense linear algebra software for heterogeneous architectures
Version 0.9.1

Inria
University of Tennessee
University of Colorado Denver
King Abdullah University of Science and Technology

Table of Contents

1 Introduction to CHAMELEON 1
1.1 MORSE project 1
1.1.1 MORSE Objectives 1
1.1.2 Research fields 1
1.1.2.1 Fine interaction between linear algebra and runtime systems 1
1.1.2.2 Runtime systems 2
1.1.2.3 Linear algebra 2
1.1.3 Research papers 2
1.2 CHAMELEON 2
1.2.1 CHAMELEON software 2
1.2.2 PLASMA's design principles 3
1.2.2.1 Tile Algorithms 3
1.2.2.2 Tile Data Layout 4
1.2.2.3 Dynamic Task Scheduling 5

2 Installing CHAMELEON 7
2.1 Downloading CHAMELEON 7
2.1.1 Getting Sources 7
2.1.2 Required dependencies 7
2.1.2.1 a BLAS implementation 7
2.1.2.2 CBLAS 7
2.1.2.3 a LAPACK implementation 7
2.1.2.4 LAPACKE 8
2.1.2.5 libtmg 8
2.1.2.6 QUARK 8
2.1.2.7 StarPU 8
2.1.2.8 hwloc 9
2.1.2.9 pthread 9
2.1.3 Optional dependencies 9
2.1.3.1 OpenMPI 9
2.1.3.2 Nvidia CUDA Toolkit 9
2.1.3.3 MAGMA 9
2.1.3.4 FxT 10
2.2 Build process of CHAMELEON 10
2.2.1 Setting up a build directory 10
2.2.2 Configuring the project with best efforts 10
2.2.3 Building 10
2.2.4 Tests 10
2.2.5 Installing 11

3 Configuring CHAMELEON 13

3.1 Compilation configuration 13
3.1.1 General CMake options 13
3.1.2 CHAMELEON options 14
3.2 Dependencies detection 16
3.3 Use FxT profiling through StarPU 17
3.4 Use simulation mode with StarPU-SimGrid 17

4 Using CHAMELEON 19

4.1 Using CHAMELEON executables 19
4.2 Linking an external application with CHAMELEON libraries 21
4.2.1 Static linking in C 21
4.2.2 Dynamic linking in C 21
4.2.3 Build a Fortran program with CHAMELEON 22
4.3 CHAMELEON API 22
4.3.1 Tutorial LAPACK to CHAMELEON 23
4.3.1.1 Step0 23
4.3.1.2 Step1 24
4.3.1.3 Step2 25
4.3.1.4 Step3 26
4.3.1.5 Step4 27
4.3.1.6 Step5 28
4.3.1.7 Step6 29
4.3.2 List of available routines 30
4.3.2.1 Auxiliary routines 30
4.3.2.2 Descriptor routines 30
4.3.2.3 Options routines 31
4.3.2.4 Sequences routines 31
4.3.2.5 Linear Algebra routines 31

1 Introduction to CHAMELEON

1.1 MORSE project

1.1.1 MORSE Objectives

When processor clock speeds flatlined in 2004, after more than fifteen years of exponential increases, the era of near automatic performance improvements that the HPC application community had previously enjoyed came to an abrupt end. To develop software that will perform well on petascale and exascale systems with thousands of nodes and millions of cores, the list of major challenges that must now be confronted is formidable: 1) dramatic escalation in the costs of intrasystem communication between processors and/or levels of memory hierarchy; 2) increased heterogeneity of the processing units (mixing CPUs, GPUs, etc. in varying and unexpected design combinations); 3) high levels of parallelism and more complex constraints means that cooperating processes must be dynamically and unpredictably scheduled for asynchronous execution; 4) software will not run at scale without much better resilience to faults and far more robustness; and 5) new levels of self-adaptivity will be required to enable software to modulate process speed in order to satisfy limited energy budgets. The MORSE associate team will tackle the first three challenges in a orchestrating work between research groups respectively specialized in sparse linear algebra, dense linear algebra and runtime systems. The overall objective is to develop robust linear algebra libraries relying on innovative runtime systems that can fully benefit from the potential of those future large-scale complex machines. Challenges 4) and 5) will also be investigated by the different teams in the context of other partnerships, but they will not be the main focus of the associate team as they are much more prospective.

1.1.2 Research fields

The overall goal of the MORSE associate team is to enable advanced numerical algorithms to be executed on a scalable unified runtime system for exploiting the full potential of future exascale machines. We expect advances in three directions based first on strong and closed interactions between the runtime and numerical linear algebra communities. This initial activity will then naturally expand to more focused but still joint research in both fields.

1.1.2.1 Fine interaction between linear algebra and runtime systems

On parallel machines, HPC applications need to take care of data movement and consistency, which can be either explicitly managed at the level of the application itself or delegated to a runtime system. We adopt the latter approach in order to better keep up with hardware trends whose complexity is growing exponentially. One major task in this project is to define a proper interface between HPC applications and runtime systems in order to maximize productivity and expressivity. As mentioned in the next section, a widely used approach consists in abstracting the application as a DAG that the runtime system is in charge of scheduling. Scheduling such a DAG over a set of heterogeneous processing units introduces a lot of new challenges, such as predicting accurately the execution time of each type of task over each kind of unit, minimizing data transfers between memory banks, performing data prefetching, etc. Expected advances: In a nutshell, a new runtime system API will be

designed to allow applications to provide scheduling hints to the runtime system and to get real-time feedback about the consequences of scheduling decisions.

1.1.2.2 Runtime systems

A runtime environment is an intermediate layer between the system and the application. It provides low-level functionality not provided by the system (such as scheduling or management of the heterogeneity) and high-level features (such as performance portability). In the framework of this proposal, we will work on the scalability of runtime environment. To achieve scalability it is required to avoid all centralization. Here, the main problem is the scheduling of the tasks. In many task-based runtime environments the scheduler is centralized and becomes a bottleneck as soon as too many cores are involved. It is therefore required to distribute the scheduling decision or to compute a data distribution that impose the mapping of task using, for instance the so-called "owner-compute" rule. Expected advances: We will design runtime systems that enable an efficient and scalable use of thousands of distributed multicore nodes enhanced with accelerators.

1.1.2.3 Linear algebra

Because of its central position in HPC and of the well understood structure of its algorithms, dense linear algebra has often pioneered new challenges that HPC had to face. Again, dense linear algebra has been in the vanguard of the new era of petascale computing with the design of new algorithms that can efficiently run on a multicore node with GPU accelerators. These algorithms are called "communication-avoiding" since they have been redesigned to limit the amount of communication between processing units (and between the different levels of memory hierarchy). They are expressed through Direct Acyclic Graphs (DAG) of fine-grained tasks that are dynamically scheduled. Expected advances: First, we plan to investigate the impact of these principles in the case of sparse applications (whose algorithms are slightly more complicated but often rely on dense kernels). Furthermore, both in the dense and sparse cases, the scalability on thousands of nodes is still limited; new numerical approaches need to be found. We will specifically design sparse hybrid direct/iterative methods that represent a promising approach.

1.1.3 Research papers

Research papers about MORSE can be found at

http://icl.cs.utk.edu/projectsdev/morse/pubs/index.html

1.2 CHAMELEON

1.2.1 CHAMELEON software

The main purpose is to address the performance shortcomings of the LAPACK and ScaLAPACK libraries on multicore processors and multi-socket systems of multicore processors and their inability to efficiently utilize accelerators such as Graphics Processing Units (GPUs).

CHAMELEON is a framework written in C which provides routines to solve dense general systems of linear equations, symmetric positive definite systems of linear equations and linear least squares problems, using LU, Cholesky, QR and LQ factorizations. Real arith-

metic and complex arithmetic are supported in both single precision and double precision. It supports Linux and Mac OS/X machines (only tested on Intel x86-64 architecture).

CHAMELEON is based on PLASMA source code but is not limited to shared-memory environment and can exploit multiple GPUs. CHAMELEON is interfaced in a generic way with both QUARK and StarPU runtime systems. This feature allows to analyze in a unified framework how sequential task-based algorithms behave regarding different runtime systems implementations. Using CHAMELEON with StarPU runtime system allows to exploit GPUs through kernels provided by cuBLAS and MAGMA and clusters of interconnected nodes with distributed memory (using MPI). Computation of very large systems with dense matrices on a cluster of nodes is still being experimented and stabilized. It is not expected to get stable performances with the current version using MPI.

1.2.2 PLASMA's design principles

CHAMELEON is originally based on PLASMA so that design principles are very similar. The content of this section Section 1.2.2 [PLASMA's design principles], page 3 has been copied from the '`Design principles`' section of the PLASMA User's Guide.

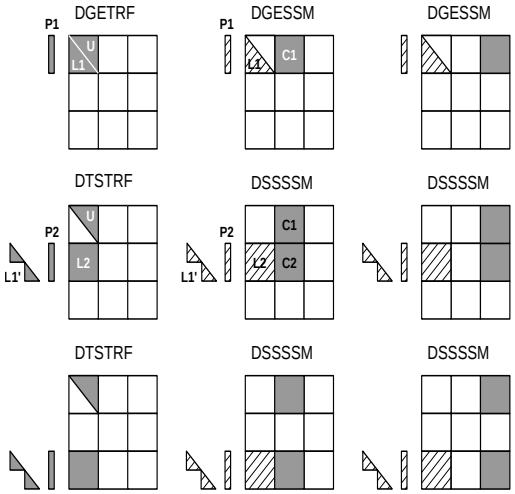
1.2.2.1 Tile Algorithms

Tile algorithms are based on the idea of processing the matrix by square tiles of relatively small size, such that a tile fits entirely in one of the cache levels associated with one core. This way a tile can be loaded to the cache and processed completely before being evicted back to the main memory. Of the three types of cache misses, *compulsory*, *capacity* and *conflict*, the use of tile algorithms minimizes the number of capacity misses, since each operation loads the amount of data that does not "overflow" the cache.

For some operations such as matrix multiplication and Cholesky factorization, translating the classic algorithm to the tile algorithm is trivial. In the case of matrix multiplication, the tile algorithm is simply a product of applying the technique of *loop tiling* to the canonical definition of three nested loops. It is very similar for the Cholesky factorization. The **left-looking** definition of Cholesky factorization from LAPACK is a loop with a sequence of calls to four routines: xSYRK (symmetric **rank-k** update), xPOTRF (Cholesky factorization of a small block on the diagonal), xGEMM (matrix multiplication) and xTRSM (triangular solve). If the xSYRK, xGEMM and xTRSM operations are expressed with the canonical definition of three nested loops and the technique of loop tiling is applied, the tile algorithm results. Since the algorithm is produced by simple reordering of operations, neither the number of operations nor numerical stability of the algorithm are affected.

The situation becomes slightly more complicated for LU and QR factorizations, where the classic algorithms factorize an entire panel of the matrix (a block of columns) at every step of the algorithm. One can observe, however, that the process of matrix factorization is synonymous with introducing zeros in approproate places and a tile algorithm can be fought of as one that zeroes one tile of the matrix at a time. This process is referred to as updating of a factorization or *incremental factorization*. The process is equivalent to factorizing the top tile of a panel, then placing the upper triangle of the result on top of the tile blow and factorizing again, then moving to the next tile and so on. Here, the tile LU and QR algorithms perform slightly more floating point operations and require slightly more memory for auxiliary data. Also, the tile LU factorization applies a different pivoting pattern and, as a result, is less numerically stable than classic LU with full pivoting. Numerical stability is

not an issue in case of the tile QR, which relies on orthogonal transformations (Householder reflections), which are numerically stable.



Schematic illustration of the tile LU factorization (kernel names for real arithmetics in double precision), courtesey of the PLASMA team.

1.2.2.2 Tile Data Layout

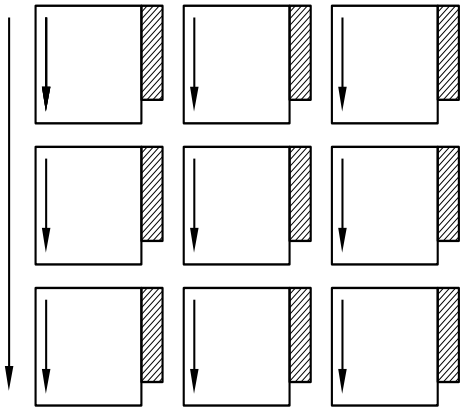
Tile layout is based on the idea of storing the matrix by square tiles of relatively small size, such that each tile occupies a continuous memory region. This way a tile can be loaded to the cache memory efficiently and the risk of evicting it from the cache memory before it is completely processed is minimized. Of the three types of cache misses, *compulsory*, *capacity* and *conflict*, the use of tile layout minimizes the number of conflict misses, since a continuous region of memory will completely fill out a **set-associative** cache memory before an eviction can happen. Also, from the standpoint of multithreaded execution, the probability of *false sharing* is minimized. It can only affect the cache lines containing the beginning and the ending of a tile.

In standard **cache-based** architecture, tiles continously laid out in memory maximize the profit from automatic prefetching. Tile layout is also beneficial in situations involving the use of accelerators, where explicit communication of tiles through DMA transfers is required, such as moving tiles between the system memory and the local store in Cell B. E. or moving tiles between the host memory and the device memory in GPUs. In most circumstances tile layout also minimizes the number of TLB misses and conflicts to memory banks or partitions. With the standard (**column-major**) layout, access to each column of a tile is much more likely to cause a conflict miss, a false sharing miss, a TLB miss or a bank or partition conflict. The use of the standard layout for dense matrix operations is a performance minefield. Although occasionally one can pass through it unscathed, the risk of hitting a spot deadly to performance is very high.

Another property of the layout utilized in PLASMA is that it is "flat", meaning that it does not involve a level of indirection. Each tile stores a small square submatrix of the main matrix in a **column-major** layout. In turn, the main matrix is an arrangement of tiles immediately following one another in a **column-major** layout. The offset of each

tile can be calculated through address arithmetics and does not involve pointer indirection. Alternatively, a matrix could be represented as an array of pointers to tiles, located anywhere in memory. Such layout would be a radical and unjustifiable departure from LAPACK and ScaLAPACK. Flat tile layout is a natural progression from LAPACK's **column-major** layout and ScaLAPACK's **block-cyclic** layout.

Another related property of PLASMA's tile layout is that it includes provisions for padding of tiles, i.e., the actual region of memory designated for a tile can be larger than the memory occupied by the actual data. This allows to force a certain alignment of tile boundaries, while using the flat organization described in the previous paragraph. The motivation is that, at the price of small memory overhead, alignment of tile boundaries may prove benefivial in multiple scenarios involving memory systems of standard multicore processors, as well as accelerators. The issues that come into play are, again, the use of TLBs and memory banks or partitions.



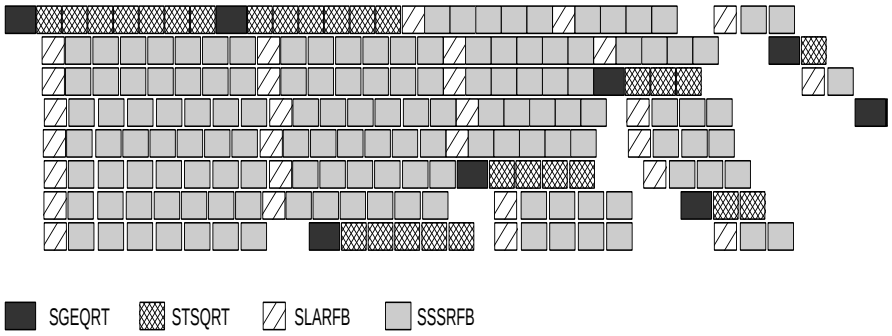
Schematic illustration of the tile layout with **column-major** order of tiles, **column-major** order of elements within tiles and (optional) padding for enforcing a certain alighment of tile bondaries, courtesey of the PLASMA team.

1.2.2.3 Dynamic Task Scheduling

Dynamic scheduling is the idea of assigning work to cores based on the availability of data for processing at any given point in time and is also referred to as *data-driven* scheduling. The concept is related closely to the idea of expressing computation through a task graph, often referred to as the DAG (*Direct Acyclic Graph*), and the flexibility exploring the DAG at runtime. Thus, to a large extent, dynamic scheduling is synonymous with *runtime scheduling*. An important concept here is the one of the *critical path*, which defines the upper bound on the achievable parallelism, and needs to be pursued at the maximum speed. This is in direct opposition to the *fork-and-join* or *data-parallel* programming models, where artificial synchronization points expose serial sections of the code, where multiple cores are idle, while sequential processing takes place. The use of dynamic scheduling introduces a **trade-off**, though. The more dynamic (flexible) scheduling is, the more centralized (and less scalable) the scheduling mechanism is. For that reason, currently PLASMA uses two

scheduling mechanisms, one which is fully dynamic and one where work is assigned statically and dependency checks are done at runtime.

The first scheduling mechanism relies on unfolding a *sliding window* of the task graph at runtime and scheduling work by resolving data hazards: *Read After Write˜(RAW)*, *Write After Read˜(WAR)* and *Write After Write˜(WAW)*, a technique analogous to instruction scheduling in superscalar processors. It also relies on *work-stealing* for balanding the load among all multiple cores. The second scheduling mechanism relies on statically designating a path through the execution space of the algorithm to each core and following a cycle: transition to a task, wait for its dependencies, execute it, update the overall progress. Task are identified by tuples and task transitions are done through locally evaluated formulas. Progress information can be centralized, replicated or distributed (currently centralized).



A trace of the tile QR factorization executing on eight cores without any global synchronization points (kernel names for real arithmetics in single precision), courtesey of the PLASMA team.

2 Installing CHAMELEON

CHAMELEON can be built and installed by the standard means of CMake (http://www.cmake.org/). General information about CMake, as well as installation binaries and CMake source code are available from http://www.cmake.org/cmake/resources/software.html. The following chapter is intended to briefly remind how these tools can be used to install CHAMELEON.

2.1 Downloading CHAMELEON

2.1.1 Getting Sources

The latest official release tarballs of CHAMELEON sources are available for download from chameleon-0.9.1.

2.1.2 Required dependencies

2.1.2.1 a BLAS implementation

BLAS (Basic Linear Algebra Subprograms), are a de facto standard for basic linear algebra operations such as vector and matrix multiplication. FORTRAN implementation of BLAS is available from Netlib. Also, C implementation of BLAS is included in GSL (GNU Scientific Library). Both these implementations are reference implementation of BLAS, are not optimized for modern processor architectures and provide an order of magnitude lower performance than optimized implementations. Highly optimized implementations of BLAS are available from many hardware vendors, such as Intel MKL and AMD ACML. Fast implementations are also available as academic packages, such as ATLAS and Goto BLAS. The standard interface to BLAS is the FORTRAN interface.

Caution about the compatibility: CHAMELEON has been mainly tested with the reference BLAS from NETLIB and the Intel MKL 11.1 from Intel distribution 2013_sp1.

2.1.2.2 CBLAS

CBLAS is a C language interface to BLAS. Most commercial and academic implementations of BLAS also provide CBLAS. Netlib provides a reference implementation of CBLAS on top of FORTRAN BLAS (Netlib CBLAS). Since GSL is implemented in C, it naturally provides CBLAS.

Caution about the compatibility: CHAMELEON has been mainly tested with the reference CBLAS from NETLIB and the Intel MKL 11.1 from Intel distribution 2013_sp1.

2.1.2.3 a LAPACK implementation

LAPACK (Linear Algebra PACKage) is a software library for numerical linear algebra, a successor of LINPACK and EISPACK and a predecessor of CHAMELEON. LAPACK provides routines for solving linear systems of equations, linear least square problems, eigenvalue problems and singular value problems. Most commercial and academic BLAS packages also provide some LAPACK routines.

Caution about the compatibility: CHAMELEON has been mainly tested with the reference LAPACK from NETLIB and the Intel MKL 11.1 from Intel distribution 2013_sp1.

2.1.2.4 LAPACKE

LAPACKE is a C language interface to LAPACK (or CLAPACK). It is produced by Intel in coordination with the LAPACK team and is available in source code from Netlib in its original version (Netlib LAPACKE) and from CHAMELEON website in an extended version (LAPACKE for CHAMELEON). In addition to implementing the C interface, LAPACKE also provides routines which automatically handle workspace allocation, making the use of LAPACK much more convenient.

Caution about the compatibility: CHAMELEON has been mainly tested with the reference LAPACKE from NETLIB. A stand-alone version of LAPACKE is required.

2.1.2.5 libtmg

libtmg is a component of the LAPACK library, containing routines for generation of input matrices for testing and timing of LAPACK. The testing and timing suites of LAPACK require libtmg, but not the library itself. Note that the LAPACK library can be built and used without libtmg.

Caution about the compatibility: CHAMELEON has been mainly tested with the reference TMG from NETLIB and the Intel MKL 11.1 from Intel distribution 2013_sp1.

2.1.2.6 QUARK

QUARK (QUeuing And Runtime for Kernels) provides a library that enables the dynamic execution of tasks with data dependencies in a multi-core, multi-socket, shared-memory environment. One of QUARK or StarPU Runtime systems has to be enabled in order to schedule tasks on the architecture. If QUARK is enabled then StarPU is disabled and conversely. Note StarPU is enabled by default. When CHAMELEON is linked with QUARK, it is not possible to exploit neither CUDA (for GPUs) nor MPI (distributed-memory environment). You can use StarPU to do so.

Caution about the compatibility: CHAMELEON has been mainly tested with the QUARK library from PLASMA release between versions 2.5.0 and 2.6.0.

2.1.2.7 StarPU

StarPU is a task programming library for hybrid architectures. StarPU handles run-time concerns such as:

- Task dependencies
- Optimized heterogeneous scheduling
- Optimized data transfers and replication between main memory and discrete memories
- Optimized cluster communications

StarPU can be used to benefit from GPUs and distributed-memory environment. One of QUARK or StarPU runtime system has to be enabled in order to schedule tasks on the architecture. If StarPU is enabled then QUARK is disabled and conversely. Note StarPU is enabled by default.

Caution about the compatibility: CHAMELEON has been mainly tested with StarPU-1.1 releases.

2.1.2.8 hwloc

hwloc (Portable Hardware Locality) is a software package for accessing the topology of a multicore system including components like: cores, sockets, caches and NUMA nodes. It allows to increase performance, and to perform some topology aware scheduling. `hwloc` is available in major distributions and for most OSes and can be downloaded from `http://www.open-mpi.org/software/hwloc`.

Caution about the compatibility: hwloc should be compatible with the version of StarPU used.

2.1.2.9 pthread

POSIX threads library is required to run CHAMELEON on Unix-like systems. It is a standard component of any such system.

2.1.3 Optional dependencies

2.1.3.1 OpenMPI

OpenMPI is an open source Message Passing Interface implementation for execution on multiple nodes with distributed-memory environment. MPI can be enabled only if the runtime system chosen is StarPU (default). To use MPI through StarPU, it is necessary to compile StarPU with MPI enabled.

Caution about the compatibility: CHAMELEON has been mainly tested with OpenMPI releases from versions 1.4 to 1.6.

2.1.3.2 Nvidia CUDA Toolkit

Nvidia CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications. CHAMELEON can use a set of low level optimized kernels coming from cuBLAS to accelerate computations on GPUs. The cuBLAS library is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the Nvidia CUDA runtime. cuBLAS is normaly distributed with Nvidia CUDA Toolkit. CUDA/cuBLAS can be enabled in CHAMELEON only if the runtime system chosen is StarPU (default). To use CUDA through StarPU, it is necessary to compile StarPU with CUDA enabled.

Caution about the compatibility: CHAMELEON has been mainly tested with CUDA releases from versions 4 to 6. MAGMA library must be compatible with CUDA.

2.1.3.3 MAGMA

MAGMA project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems. CHAMELEON can use a set of high level MAGMA routines to accelerate computations on GPUs. To fully benefit from GPUs, the user should enable MAGMA in addition to CUDA/cuBLAS.

Caution about the compatibility: CHAMELEON has been mainly tested with MAGMA releases from versions 1.4 to 1.6. MAGMA library must be compatible with CUDA. MAGMA library should be built with sequential versions of BLAS/LAPACK. We should not get some MAGMA link flags embarking multithreaded BLAS/LAPACK because it

could affect permformances (take care about the MAGMA link flag `-lmkl_intel_thread` for example that we could heritate from the pkg-config file `magma.pc`).

2.1.3.4 FxT

FxT stands for both FKT (Fast Kernel Tracing) and FUT (Fast User Tracing). This library provides efficient support for recording traces. CHAMELEON can trace kernels execution on the different workers and produce .paje files if FxT is enabled. FxT can only be used through StarPU and StarPU must be compiled with FxT enabled, see how to use this feature here Section 3.3 [Use FxT profiling through StarPU], page 17.

Caution about the compatibility: FxT should be compatible with the version of StarPU used.

2.2 Build process of CHAMELEON

2.2.1 Setting up a build directory

The CHAMELEON build process requires CMake version 2.8.0 or higher and working C and Fortran compilers. Compilation and link with CHAMELEON libraries have been tested with **gcc/gfortran 4.8.1** and **icc/ifort 14.0.2**. On Unix-like operating systems, it also requires Make. The CHAMELEON project can not be configured for an in-source build. You will get an error message if you try to compile in-source. Please clean the root of your project by deleting the generated `CMakeCache.txt` file (and other CMake generated files).

```
mkdir build
cd build
```

You can create a build directory from any location you would like. It can be a sub-directory of the CHAMELEON base source directory or anywhere else.

2.2.2 Configuring the project with best efforts

```
cmake <path to SOURCE_DIR> -DOPTION1= -DOPTION2= ...
```

`<path to SOURCE_DIR>` represents the root of CHAMELEON project where stands the main (parent) `CMakeLists.txt` file. Details about options that are useful to give to `cmake <path to SOURCE_DIR>` are given in Section 3.1 [Compilation configuration], page 13.

2.2.3 Building

```
make [-j[ncores]]
```

do not hesitate to use `-j[ncores]` option to speedup the compilation

2.2.4 Tests

In order to make sure that CHAMELEON is working properly on the system, it is also possible to run a test suite.

```
make check
```

or

```
ctest
```

2.2.5 Installing

In order to install CHAMELEON at the location that was specified during configuration:

```
make install
```

do not forget to specify the install directory with `-DCMAKE_INSTALL_PREFIX` at cmake configure

```
cmake <path to SOURCE_DIR> -DCMAKE_INSTALL_PREFIX=<path to INSTALL_DIR>
```

Note that the install process is optional. You are free to use CHAMELEON binaries compiled in the build directory.

3 Configuring CHAMELEON

3.1 Compilation configuration

The following arguments can be given to the `cmake <path to source directory>` script.

In this chapter, the following convention is used:

- `path` is a path in your filesystem,
- `var` is a string and the correct value or an example will be given,
- `trigger` is an CMake option and the correct value is `ON` or `OFF`.

Using CMake there are several ways to give options:

1. directly as CMake command line arguments
2. invoque `cmake <path to source directory>` once and then use `ccmake <path to source directory>` to edit options through a minimalist gui (required '`cmake-curses-gui`' installed on a Linux system)
3. invoque `cmake-gui` command and fill information about the location of the sources and where to build the project, then you have access to options through a user-friendly Qt interface (required '`cmake-qt-gui`' installed on a Linux system)

Example of configuration using the command line

```
cmake ~/chameleon/ -DCMAKE_BUILD_TYPE=Debug          \
                   -DCMAKE_INSTALL_PREFIX=~/install  \
                   -DCHAMELEON_USE_CUDA=ON           \
                   -DCHAMELEON_USE_MAGMA=ON          \
                   -DCHAMELEON_USE_MPI=ON            \
                   -DBLA_VENDOR=Intel10_64lp         \
                   -DSTARPU_DIR=~/install/starpu-1.1 \
                   -DCHAMELEON_USE_FXT=ON
```

You can get the full list of options with `-L[A][H]` options of `cmake` command:

```
cmake -LH <path to source directory>
```

3.1.1 General CMake options

`-DCMAKE_INSTALL_PREFIX=path (default:path=/usr/local)`

Install directory used by `make install` where some headers and libraries will be copied. Permissions have to be granted to write onto `path` during `make install` step.

`-DCMAKE_BUILD_TYPE=var (default: Release)`

Define the build type and the compiler optimization level. The possible values for `var` are:

`empty`

`Debug`

`Release`

`RelWithDebInfo`
`MinSizeRel`

`-DBUILD_SHARED_LIBS=trigger (default:OFF)`
: Indicate wether or not CMake has to build CHAMELEON static (`OFF`) or shared (`ON`) libraries.

3.1.2 CHAMELEON options

List of CHAMELEON options that can be enabled/disabled (value=`ON` or `OFF`):

`-DCHAMELEON_SCHED_STARPU=trigger (default: ON)`
: to link with StarPU library (runtime system)

`-DCHAMELEON_SCHED_QUARK=trigger (default: OFF)`
: to link with QUARK library (runtime system)

`-DCHAMELEON_USE_CUDA=trigger (default: OFF)`
: to link with CUDA runtime (implementation paradigm for accelerated codes on GPUs) and cuBLAS library (optimized BLAS kernels on GPUs), can only be used with StarPU

`-DCHAMELEON_USE_MAGMA=trigger (default: OFF)`
: to link with MAGMA library (kernels on GPUs, higher level than cuBLAS), can only be used with StarPU

`-DCHAMELEON_USE_MPI=trigger (default: OFF)`
: to link with MPI library (message passing implementation for use of multiple nodes with distributed memory), can only be used with StarPU

`-DCHAMELEON_USE_FXT=trigger (default: OFF)`
: to link with FxT library (trace execution of kernels on workers), can only be used with StarPU

`-DCHAMELEON_SIMULATION=trigger (default: OFF)`
: to enable simulation mode, means CHAMELEON will not really execute tasks, see details in section Section 3.4 [Use simulation mode with StarPU-SimGrid], page 17. This option must be used with StarPU compiled with SimGrid allowing to guess the execution time on any architecture. This feature should be used to make experiments on the scheduler behaviors and performances not to produce solutions of linear systems.

`-DCHAMELEON_ENABLE_DOCS=trigger (default: ON)`
: to control build of the documentation contained in `docs/` sub-directory

`-DCHAMELEON_ENABLE_EXAMPLE=trigger (default: ON)`
: to control build of the examples executables (API usage) contained in `example/` sub-directory

`-DCHAMELEON_ENABLE_TESTING=trigger (default: ON)`
: to control build of testing executables (numerical check) contained in `testing/` sub-directory

`-DCHAMELEON_ENABLE_TIMING=trigger (default: ON)`
: to control build of timing executables (performances check) contained in `timing/` sub-directory

`-DCHAMELEON_PREC_S=trigger (default: ON)`
: to enable the support of simple arithmetic precision (float in C)

`-DCHAMELEON_PREC_D=trigger (default: ON)`
: to enable the support of double arithmetic precision (double in C)

`-DCHAMELEON_PREC_C=trigger (default: ON)`
: to enable the support of complex arithmetic precision (complex in C)

`-DCHAMELEON_PREC_Z=trigger (default: ON)`
: to enable the support of double complex arithmetic precision (double complex in C)

`-DBLAS_VERBOSE=trigger (default: OFF)`
: to make BLAS library discovery verbose

`-DLAPACK_VERBOSE=trigger (default: OFF)`
: to make LAPACK library discovery verbose (automatically enabled if `BLAS_VERBOSE=ON`)

List of CHAMELEON options that needs a specific value:

`-DBLA_VENDOR=var (default: empty)`
: The possible values for `var` are:

  `empty`

  `all`

  `Intel10_64lp`
  `Intel10_64lp_seq`
  `ACML`

  `Apple`

  `Generic`

  `...`

  to force CMake to find a specific BLAS library, see the full list of BLA_VENDOR in `FindBLAS.cmake` in `cmake_modules/morse/find`. By default `BLA_VENDOR` is empty so that CMake tries to detect all possible BLAS vendor with a preference for Intel MKL.

List of CHAMELEON options which requires to give a path:

`-DLIBNAME_DIR=path (default: empty)`
: root directory of the LIBNAME library installation

`-DLIBNAME_INCDIR=path (default: empty)`
: directory of the LIBNAME library headers installation

`-DLIBNAME_LIBDIR=path (default: empty)`
: directory of the LIBNAME libraries (.so, .a, .dylib, etc) installation

LIBNAME can be one of the following: BLAS - CBLAS - FXT - HWLOC - LAPACK - LAPACKE - MAGMA - QUARK - STARPU - TMG. See paragraph about Section 3.2 [Dependencies detection], page 16 for details.

Libraries detected with an official CMake module (see module files in `CMAKE_ROOT/Modules/`):

- CUDA
- MPI
- Threads

Libraries detected with CHAMELEON cmake modules (see module files in `cmake_modules/morse/find/` directory of CHAMELEON sources):

- BLAS
- CBLAS
- FXT
- HWLOC
- LAPACK
- LAPACKE
- MAGMA
- QUARK
- STARPU
- TMG

3.2 Dependencies detection

You have different choices to detect dependencies on your system, either by setting some environment variables containing paths to the libs and headers or by specifying them directly at cmake configure. Different cases :

1. detection of dependencies through environment variables:
   - `LD_LIBRARY_PATH` environment variable should contain the list of paths where to find the libraries:
     ```
     export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:path/to/your/libs
     ```
   - `INCLUDE` environment variable should contain the list of paths where to find the header files of libraries
     ```
     export INCLUDE=$INCLUDE:path/to/your/headers
     ```
2. detection with user's given paths:
   - you can specify the path at cmake configure by invoking
     ```
     cmake <path to SOURCE_DIR> -DLIBNAME_DIR=path/to/your/lib
     ```
     where LIB stands for the name of the lib to look for, example
     ```
     cmake <path to SOURCE_DIR> -DSTARPU_DIR=path/to/starpudir \
                                -DCBLAS_DIR= ...
     ```
   - it is also possible to specify headers and library directories separately, example

```
cmake <path to SOURCE_DIR>                                  \
-DSTARPU_INCDIR=path/to/libstarpu/include/starpu/1.1 \
-DSTARPU_LIBDIR=path/to/libstarpu/lib
```

- Note BLAS and LAPACK detection can be tedious so that we provide a verbose mode. Use `-DBLAS_VERBOSE=ON` or `-DLAPACK_VERBOSE=ON` to enable it.

3.3 Use FxT profiling through StarPU

StarPU can generate its own trace log files by compiling it with the `--with-fxt` option at the configure step (you can have to specify the directory where you installed FxT by giving `--with-fxt=...` instead of `--with-fxt` alone). By doing so, traces are generated after each execution of a program which uses StarPU in the directory pointed by the `STARPU_FXT_PREFIX` environment variable. Example:

```
export STARPU_FXT_PREFIX=/home/yourname/fxt_files/
```

When executing a `./timing/...` CHAMELEON program, if it has been enabled (StarPU compiled with FxT and `-DCHAMELEON_USE_FXT=ON`), you can give the option `--trace` to tell the program to generate trace log files.

Finally, to generate the trace file which can be opened with Vite program, you have to use the `starpu_fxt_tool` executable of StarPU. This tool should be in `path/to/your/install/starpu/bin`. You can use it to generate the trace file like this:

- `path/to/your/install/starpu/bin/starpu_fxt_tool -i prof_filename`

  There is one file per mpi processus (prof_filename_0, prof_filename_1 ...). To generate a trace of mpi programs you can call it like this:
- `path/to/your/install/starpu/bin/starpu_fxt_tool -i prof_filename*`

  The trace file will be named paje.trace (use -o option to specify an output name).

3.4 Use simulation mode with StarPU-SimGrid

Simulation mode can be enabled by setting the cmake option `-DCHAMELEON_SIMULATION=ON`. This mode allows you to simulate execution of algorithms with StarPU compiled with SimGrid. To do so, we provide some perfmodels in the `simucore/perfmodels/` directory of CHAMELEON sources. To use these perfmodels, please set the following

- `STARPU_HOME` environment variable to:

  ```
  <path to SOURCE_DIR>/simucore/perfmodels
  ```

- `STARPU_HOSTNAME` environment variable to the name of the machine to simulate. For example, on our platform (PlaFRIM) with GPUs at Inria Bordeaux

  ```
  STARPU_HOSTNAME=mirage
  ```

  Note that only POTRF kernels with block sizes of 320 or 960 (simple and double precision) on mirage machine are available for now. Database of models is subject to change, it should be enrich in a near future.

4 Using CHAMELEON

4.1 Using CHAMELEON executables

CHAMELEON provides several test executables that are compiled and link with CHAMELEON stack of dependencies. Instructions about the arguments to give to executables are accessible thanks to the option `-[-]help` or `-[-]h`. This set of binaries are separated into three categories and can be found in three different directories:

- example

  contains examples of API usage and more specifically the sub-directory lapack_to_morse/ provides a tutorial that explain how to use CHAMELEON functionalities starting from a full LAPACK code, see Section 4.3.1 [Tutorial LAPACK to CHAMELEON], page 23

- testing

  contains testing drivers to check numerical correctness of CHAMELEON linear algebra routines with a wide range of parameters

  ```
  ./testing/stesting 4 1 LANGE 600 100 700
  ```

  Two first arguments are the number of cores and gpus to use. The third one is the name of the algorithm to test. The other arguments depend on the algorithm, here it lies for the number of rows, columns and leading dimension of the problem.

  Name of algorithms available for testing are:

  - LANGE: norms of matrices Infinite, One, Max, Frobenius
  - GEMM: general matrix-matrix multiply
  - HEMM: hermitian matrix-matrix multiply
  - HERK: hermitian matrix-matrix rank k update
  - HER2K: hermitian matrix-matrix rank 2k update
  - SYMM: symmetric matrix-matrix multiply
  - SYRK: symmetric matrix-matrix rank k update
  - SYR2K: symmetric matrix-matrix rank 2k update
  - PEMV: matrix-vector multiply with pentadiagonal matrix
  - TRMM: triangular matrix-matrix multiply
  - TRSM: triangular solve, multiple rhs
  - POSV: solve linear systems with symmetric positive-definite matrix
  - GESV_INCPIV: solve linear systems with general matrix
  - GELS: linear least squares with general matrix

- timing

  contains timing drivers to assess performances of CHAMELEON routines. There are two sets of executables, those who do not use the tile interface and those who do (with _tile in the name of the executable). Executables without tile interface allocates data following LAPACK conventions and these data can be given as arguments to

CHAMELEON routines as you would do with LAPACK. Executables with tile interface generate directly the data in the format CHAMELEON tile algorithms used to submit tasks to the runtime system. Executables with tile interface should be more performant because no data copy from LAPACK matrix layout to tile matrix layout are necessary. Calling example:

```
./timing/time_dpotrf --n_range=1000:10000:1000 --nb=320
                     --threads=9 --gpus=3
                     --nowarmup
```

List of main options that can be used in timing:

- `--help`: show usage
- `--threads`: Number of CPU workers (default: `_SC_NPROCESSORS_ONLN`)
- `--gpus`: number of GPU workers (default: `0`)
- `--n_range=R`: range of N values, with `R=Start:Stop:Step` (default: `500:5000:500`)
- `--m=X`: dimension (M) of the matrices (default: `N`)
- `--k=X`: dimension (K) of the matrices (default: `1`), useful for GEMM algorithm (k is the shared dimension and must be defined >1 to consider matrices and not vectors)
- `--nrhs=X`: number of right-hand size (default: `1`)
- `--nb=X`: block/tile size. (default: `128`)
- `--ib=X`: inner-blocking/IB size. (default: `32`)
- `--niter=X`: number of iterations performed for each test (default: `1`)
- `--rhblk=X`: if X > 0, enable Householder mode for QR and LQ factorization. X is the size of each subdomain (default: `0`)
- `--[no]check`: check result (default: `nocheck`)
- `--[no]profile`: print profiling informations (default: `noprofile`)
- `--[no]trace`: enable/disable trace generation (default: `notrace`)
- `--[no]dag`: enable/disable DAG generation (default: `nodag`)
- `--[no]inv`: check on inverse (default: `noinv`)
- `--nocpu`: all GPU kernels are exclusively executed on GPUs (default: `0`)

List of timing algorithms available:

- LANGE: norms of matrices
- GEMM: general matrix-matrix multiply
- TRSM: triangular solve
- POTRF: Cholesky factorization with a symmetric positive-definite matrix
- POSV: solve linear systems with symmetric positive-definite matrix
- GETRF_NOPIV: LU factorization of a general matrix using the tile LU algorithm without row pivoting
- GESV_NOPIV: solve linear system for a general matrix using the tile LU algorithm without row pivoting

- GETRF_INCPIV: LU factorization of a general matrix using the tile LU algorithm with partial tile pivoting with row interchanges
- GESV_INCPIV: solve linear system for a general matrix using the tile LU algorithm with partial tile pivoting with row interchanges matrix
- GEQRF: QR factorization of a general matrix
- GELS: solves overdetermined or underdetermined linear systems involving a general matrix using the QR or the LQ factorization

4.2 Linking an external application with CHAMELEON libraries

Compilation and link with CHAMELEON libraries have been tested with **gcc/gfortran 4.8.1** and **icc/ifort 14.0.2**.

4.2.1 Static linking in C

Lets imagine you have a file main.c that you want to link with CHAMELEON static libraries. Lets consider `/home/yourname/install/chameleon` is the install directory of CHAMELEON containing sub-directories `include/` and `lib/`. Here could be your compilation command with gcc compiler:

```
gcc -I/home/yourname/install/chameleon/include -o main.o -c main.c
```

Now if you want to link your application with CHAMELEON static libraries, you could do:

```
gcc main.o -o main                                          \
/home/yourname/install/chameleon/lib/libchameleon.a         \
/home/yourname/install/chameleon/lib/libchameleon_starpu.a \
/home/yourname/install/chameleon/lib/libcoreblas.a          \
-lstarpu-1.1 -Wl,--no-as-needed -lmkl_intel_lp64            \
-lmkl_sequential -lmkl_core -lpthread -lm -lrt
```

As you can see in this example, we also link with some dynamic libraries `starpu-1.1`, `Intel MKL` libraries (for BLAS/LAPACK/CBLAS/LAPACKE), `pthread`, `m` (math) and `rt`. These libraries will depend on the configuration of your CHAMELEON build. You can find these dependencies in .pc files we generate during compilation and that are installed in the sub-directory `lib/pkgconfig` of your CHAMELEON install directory. Note also that you could need to specify where to find these libraries with `-L` option of your compiler/linker.

Before to run your program, make sure that all shared libraries paths your executable depends on are known. Enter `ldd main` to check. If some shared libraries paths are missing append them in the `LD_LIBRARY_PATH` (for Linux systems) environment variable (`DYLD_LIBRARY_PATH` on Mac, `LIB` on Windows).

4.2.2 Dynamic linking in C

For dynamic linking (need to build CHAMELEON with CMake option `BUILD_SHARED_LIBS=ON`) it is similar to static compilation/link but instead of specifying path to your static libraries you indicate the path to dynamic libraries with `-L` option and you give the name of libraries with `-l` option like this:

```
gcc main.o -o main                                  \
```

```
-L/home/yourname/install/chameleon/lib           \
-lchameleon -lchameleon_starpu -lcoreblas        \
-lstarpu-1.1 -Wl,--no-as-needed -lmkl_intel_lp64 \
-lmkl_sequential -lmkl_core -lpthread -lm -lrt
```

Note that an update of your environment variable `LD_LIBRARY_PATH` (`DYLD_LIBRARY_PATH` on Mac, `LIB` on Windows) with the path of the libraries could be required before executing, example:

```
export LD_LIBRARY_PATH=path/to/libs:path/to/chameleon/lib
```

4.2.3 Build a Fortran program with CHAMELEON

CHAMELEON provides a Fortran interface to user functions. Example:

```
call morse_version(major, minor, patch) !or
call MORSE_VERSION(major, minor, patch)
```

Build and link are very similar to the C case.

Compilation example:

```
gfortran -o main.o -c main.c
```

Static linking example:

```
gfortran main.o -o main                                    \
/home/yourname/install/chameleon/lib/libchameleon.a        \
/home/yourname/install/chameleon/lib/libchameleon_starpu.a \
/home/yourname/install/chameleon/lib/libcoreblas.a         \
-lstarpu-1.1 -Wl,--no-as-needed -lmkl_intel_lp64           \
-lmkl_sequential -lmkl_core -lpthread -lm -lrt
```

Dynamic linking example:

```
gfortran main.o -o main                          \
-L/home/yourname/install/chameleon/lib           \
-lchameleon -lchameleon_starpu -lcoreblas        \
-lstarpu-1.1 -Wl,--no-as-needed -lmkl_intel_lp64 \
-lmkl_sequential -lmkl_core -lpthread -lm -lrt
```

4.3 CHAMELEON API

CHAMELEON provides routines to solve dense general systems of linear equations, symmetric positive definite systems of linear equations and linear least squares problems, using LU, Cholesky, QR and LQ factorizations. Real arithmetic and complex arithmetic are supported in both single precision and double precision. Routines that compute linear algebra are of the folowing form:

```
MORSE_name[_Tile[_Async]]
```

- all user routines are prefixed with `MORSE`
- `name` follows BLAS/LAPACK naming scheme for algorithms (*e.g.* sgemm for general matrix-matrix multiply simple precision)
- CHAMELEON provides three interface levels

- `MORSE_name`: simplest interface, very close to CBLAS and LAPACKE, matrices are given following the LAPACK data layout (1-D array column-major). It involves copy of data from LAPACK layout to tile layout and conversely (to update LAPACK data), see Section 4.3.1.2 [Step1], page 24.
- `MORSE_name_Tile`: the tile interface avoid copies between LAPACK and tile layouts. It is the standard interface of CHAMELEON and it should achieved better performance than the previous simplest interface. The data are given through a specific structure called a descriptor, see Section 4.3.1.3 [Step2], page 25.
- `MORSE_name_Tile_Async`: similar to the tile interface, it avoids synchonization barrier normally called between `Tile` routines. At the end of an `Async` function, completion of tasks is not guarentee and data are not necessarily up-to-date. To ensure that tasks have been all executed a synchronization function has to be called after the sequence of `Async` functions, see Section 4.3.1.5 [Step4], page 27.

MORSE routine calls have to be precede from

```
MORSE_Init( NCPU, NGPU );
```

to initialize MORSE and the runtime system and followed by

```
MORSE_Finalize();
```

to free some data and finalize the runtime and/or MPI.

4.3.1 Tutorial LAPACK to CHAMELEON

This tutorial is dedicated to the API usage of CHAMELEON. The idea is to start from a simple code and step by step explain how to use CHAMELEON routines. The first step is a full BLAS/LAPACK code without dependencies to CHAMELEON, a code that most users should easily understand. Then, the different interfaces CHAMELEON provides are exposed, from the simplest API (step1) to more complicated ones (until step4). The way some important parameters are set is discussed in step5. Finally step6 is an example about distributed computation with MPI.

Source files can be found in the `example/lapack_to_morse/` directory. If CMake option `CHAMELEON_ENABLE_EXAMPLE` is `ON` then source files are compiled with the project libraries. The arithmetic precision is `double`. To execute a step '`X`', enter the following command:

```
./step'X' --option1 --option2 ...
```

Instructions about the arguments to give to executables are accessible thanks to the option `-[-]help` or `-[-]h`. Note there exist default values for options.

For all steps, the program solves a linear system $Ax = B$ The matrix values are randomly generated but ensure that matrix A is symmetric positive definite so that A can be factorized in a LL^T form using the Cholesky factorization.

Lets comment the different steps of the tutorial

4.3.1.1 Step0

The C interface of BLAS and LAPACK, that is, CBLAS and LAPACKE, are used to solve the system. The size of the system (matrix) and the number of right hand-sides can be given as arguments to the executable (be careful not to give huge numbers if you do not have an infinite amount of RAM!). As for every step, the correctness of the solution is checked by calculating the norm $||Ax - B||/(||A||||x|| + ||B||)$. The time spent in factorization+solve is

recorded and, because we know exactly the number of operations of these algorithms, we deduce the number of operations that have been processed per second (in GFlops/s). The important part of the code that solves the problem is:

```
/* Cholesky factorization:
 * A is replaced by its factorization L or L^T depending on uplo */
LAPACKE_dpotrf( LAPACK_COL_MAJOR, 'U', N, A, N );
/* Solve:
 * B is stored in X on entry, X contains the result on exit.
 * Forward ...
 */
cblas_dtrsm(
    CblasColMajor,
    CblasLeft,
    CblasUpper,
    CblasConjTrans,
    CblasNonUnit,
    N, NRHS, 1.0, A, N, X, N);
/* ... and back substitution */
cblas_dtrsm(
    CblasColMajor,
    CblasLeft,
    CblasUpper,
    CblasNoTrans,
    CblasNonUnit,
    N, NRHS, 1.0, A, N, X, N);
```

4.3.1.2 Step1

It introduces the simplest CHAMELEON interface which is equivalent to CBLAS/LAPACKE. The code is very similar to step0 but instead of calling CBLAS/LAPACKE functions, we call CHAMELEON equivalent functions. The solving code becomes:

```
/* Factorization: */
MORSE_dpotrf( UPLO, N, A, N );
/* Solve: */
MORSE_dpotrs(UPLO, N, NRHS, A, N, X, N);
```

The API is almost the same so that it is easy to use for beginners. It is important to keep in mind that before any call to MORSE routines, `MORSE_Init` has to be invoked to initialize MORSE and the runtime system. Example:

```
MORSE_Init( NCPU, NGPU );
```

After all MORSE calls have been done, a call to `MORSE_Finalize` is required to free some data and finalize the runtime and/or MPI.

```
MORSE_Finalize();
```

We use MORSE routines with the LAPACK interface which means the routines accepts the same matrix format as LAPACK (1-D array column-major). Note that we copy the

matrix to get it in our own tile structures, see details about this format here Section 1.2.2.2 [Tile Data Layout], page 4. This means you can get an overhead coming from copies.

4.3.1.3 Step2

This program is a copy of step1 but instead of using the LAPACK interface which leads to copy LAPACK matrices inside MORSE routines we use the tile interface. We will still use standard format of matrix but we will see how to give this matrix to create a MORSE descriptor, a structure wrapping data on which we want to apply sequential task-based algorithms. The solving code becomes:

```
/* Factorization: */
MORSE_dpotrf_Tile( UPLO, descA );
/* Solve: */
MORSE_dpotrs_Tile( UPLO, descA, descX );
```

To use the tile interface, a specific structure `MORSE_desc_t` must be created. This can be achieved from different ways.

1. Use the existing function `MORSE_Desc_Create`: means the matrix data are considered contiguous in memory as it is considered in PLASMA (Section 1.2.2.2 [Tile Data Layout], page 4).
2. Use the existing function `MORSE_Desc_Create_User`: it is more flexible than `Desc_Create` because you can give your own way to access to tile data so that your tiles can be allocated wherever you want in memory, see next paragraph Section 4.3.1.4 [Step3], page 26.
3. Create you own function to fill the descriptor. If you understand well the meaning of each item of `MORSE_desc_t`, you should be able to fill correctly the structure (good luck).

In Step2, we use the first way to create the descriptor:

```
MORSE_Desc_Create(&descA, NULL, MorseRealDouble,
                  NB, NB, NB*NB, N, N,
                  0, 0, N, N,
                  1, 1);
```

- `descA` is the descriptor to create.
- The second argument is a pointer to existing data. The existing data must follow LAPACK/PLASMA matrix layout Section 1.2.2.2 [Tile Data Layout], page 4 (1-D array column-major) if `MORSE_Desc_Create` is used to create the descriptor. The `MORSE_Desc_Create_User` function can be used if you have data organized differently. This is discussed in the next paragraph Section 4.3.1.4 [Step3], page 26. Giving a `NULL` pointer means you let the function allocate memory space. This requires to copy your data in the memory allocated by the `Desc_Create`. This can be done with

  ```
  MORSE_Lapack_to_Tile(A, N, descA);
  ```

- Third argument of `Desc_Create` is the datatype (used for memory allocation).
- Fourth argument until sixth argument stand for respectively, the number of rows (`NB`), columns (`NB`) in each tile, the total number of values in a tile (`NB*NB`), the number of rows (`N`), colmumns (`N`) in the entire matrix.

- Seventh argument until ninth argument stand for respectively, the beginning row (`0`), column (`0`) indexes of the submatrix and the number of rows (`N`), columns (`N`) in the submatrix. These arguments are specific and used in precise cases. If you do not consider submatrices, just use `0, 0, NROWS, NCOLS`.
- Two last arguments are the parameter of the 2-D block-cyclic distribution grid, see ScaLAPACK. To be able to use other data distribution over the nodes, `MORSE_Desc_Create_User` function should be used.

4.3.1.4 Step3

This program makes use of the same interface than Step2 (tile interface) but does not allocate LAPACK matrices anymore so that no copy between LAPACK matrix layout and tile matrix layout are necessary to call MORSE routines. To generate random right hand-sides you can use:

```
/* Allocate memory and initialize descriptor B */
MORSE_Desc_Create(&descB,  NULL, MorseRealDouble,
                  NB, NB,  NB*NB, N, NRHS,
                  0, 0, N, NRHS, 1, 1);
/* generate RHS with random values */
MORSE_dplrnt_Tile( descB, 5673 );
```

The other important point is that is it possible to create a descriptor, the necessary structure to call MORSE efficiently, by giving your own pointer to tiles if your matrix is not organized as a 1-D array column-major. This can be achieved with the `MORSE_Desc_Create_User` routine. Here is an example:

```
MORSE_Desc_Create_User(&descA, matA, MorseRealDouble,
                       NB, NB, NB*NB, N, N,
                       0, 0, N, N, 1, 1,
                       user_getaddr_arrayofpointers,
                       user_getblkldd_arrayofpointers,
                       user_getrankof_zero);
```

Firsts arguments are the same than `MORSE_Desc_Create` routine. Following arguments allows you to give pointer to functions that manage the access to tiles from the structure given as second argument. Here for example, `matA` is an array containing addresses to tiles, see the function `allocate_tile_matrix` defined in `step3.h`. The three functions you have to define for `Desc_Create_User` are:

- a function that returns address of tile $A(m, n)$, m and n standing for the indexes of the tile in the global matrix. Lets consider a matrix $4x4$ with tile size $2x2$, the matrix contains four tiles of indexes: $A(m = 0, n = 0)$, $A(m = 0, n = 1)$, $A(m = 1, n = 0)$, $A(m = 1, n = 1)$
- a function that returns the leading dimension of tile $A(m, *)$
- a function that returns MPI rank of tile $A(m, n)$

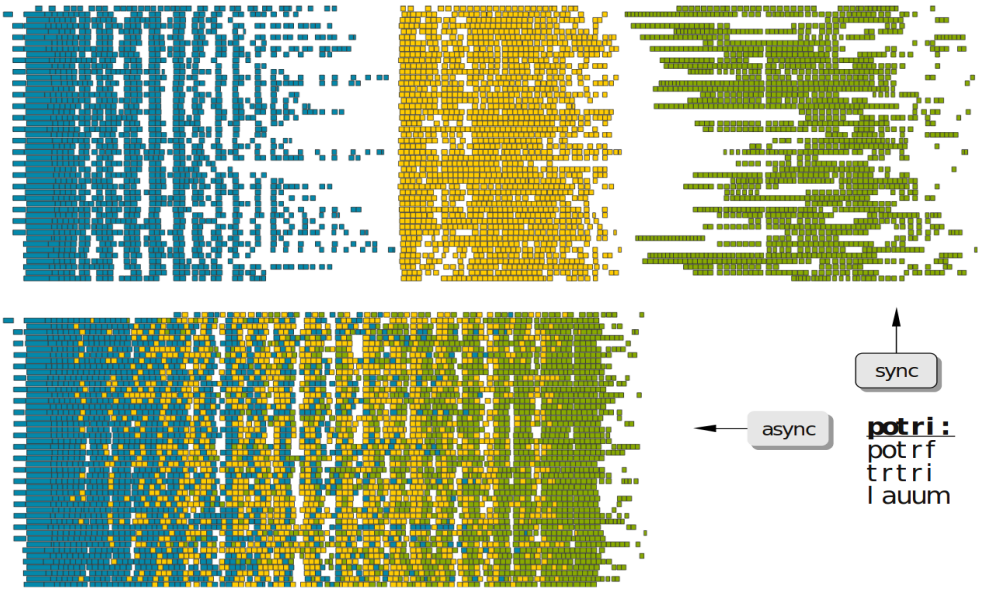
Examples for these functions are vizible in `step3.h`. Note that the way we define these functions is related to the tile matrix format and to the data distribution considered. This example should not be used with MPI since all tiles are affected to processus `0`, which means a large amount of data will be potentially transfered between nodes.

4.3.1.5 Step4

This program is a copy of step2 but instead of using the tile interface, it uses the tile async interface. The goal is to exhibit the runtime synchronization barriers. Keep in mind that when the tile interface is called, like `MORSE_dpotrf_Tile`, a synchronization function, waiting for the actual execution and termination of all tasks, is called to ensure the proper completion of the algorithm (i.e. data are up-to-date). The code shows how to exploit the async interface to pipeline subsequent algorithms so that less synchronisations are done. The code becomes:

```
/* Morse structure containing parameters and a structure to interact with
 * the Runtime system */
MORSE_context_t *morse;
/* MORSE sequence uniquely identifies a set of asynchronous function calls
 * sharing common exception handling */
MORSE_sequence_t *sequence = NULL;
/* MORSE request uniquely identifies each asynchronous function call */
MORSE_request_t request = MORSE_REQUEST_INITIALIZER;
int status;

...

morse_sequence_create(morse, &sequence);

/* Factorization: */
MORSE_dpotrf_Tile_Async( UPLO, descA, sequence, &request );

/* Solve: */
MORSE_dpotrs_Tile_Async( UPLO, descA, descX, sequence, &request);

/* Synchronization barrier (the runtime ensures that all submitted tasks
 * have been terminated */
RUNTIME_barrier(morse);
/* Ensure that all data processed on the gpus we are depending on are back
 * in main memory */
RUNTIME_desc_getoncpu(descA);
RUNTIME_desc_getoncpu(descX);

status = sequence->status;
```

Here the sequence of `dpotrf` and `dpotrs` algorithms is processed without synchronization so that some tasks of `dpotrf` and `dpotrs` can be concurently executed which could increase performances. The async interface is very similar to the tile one. It is only necessary to give two new objects `MORSE_sequence_t` and `MORSE_request_t` used to handle asynchronous function calls.



POTRI (POTRF, TRTRI, LAUUM) algorithm with and without synchronization barriers, courtesey of the PLASMA team.

4.3.1.6 Step5

Step5 shows how to set some important parameters. This program is a copy of Step4 but some additional parameters are given by the user. The parameters that can be set are:

- number of Threads
- number of GPUs

  The number of workers can be given as argument to the executable with `--threads=` and `--gpus=` options. It is important to notice that we assign one thread per gpu to optimize data transfer between main memory and devices memory. The number of workers of each type `CPU` and `CUDA` must be given at `MORSE_Init`.

```
if ( iparam[IPARAM_THRDNBR] == -1 ) {
    get_thread_count( &(iparam[IPARAM_THRDNBR]) );
    /* reserve one thread par cuda device to optimize memory transfers */
    iparam[IPARAM_THRDNBR] -= iparam[IPARAM_NCUDAS];
}
NCPU = iparam[IPARAM_THRDNBR];
NGPU = iparam[IPARAM_NCUDAS];

/* initialize MORSE with main parameters */
MORSE_Init( NCPU, NGPU );
```

- matrix size
- number of right-hand sides
- block (tile) size

  The problem size is given with `--n=` and `--nrhs=` options. The tile size is given with option `--nb=`. These parameters are required to create descriptors. The size tile `NB` is a key parameter to get performances since it defines the granularity of tasks. If `NB`

is too large compared to N, there are few tasks to schedule. If the number of workers is large this leads to limit parallelism. On the contrary, if NB is too small (*i.e.* many small tasks), workers could not be correctly fed and the runtime systems operations could represent a substantial overhead. A trade-off has to be found depending on many parameters: problem size, algorithm (drive data dependencies), architecture (number of workers, workers speed, workers uniformity, memory bus speed). By default it is set to 128. Do not hesitate to play with this parameter and compare performances on your machine.

- inner-blocking size

  The inner-blocking size is given with option --ib=. This parameter is used by kernels (optimized algorithms applied on tiles) to perform subsequent operations with data block-size that fits the cache of workers. Parameters NB and IB can be given with MORSE_Set function:

```
MORSE_Set(MORSE_TILE_SIZE,        iparam[IPARAM_NB] );
MORSE_Set(MORSE_INNER_BLOCK_SIZE, iparam[IPARAM_IB] );
```

4.3.1.7 Step6

This program is a copy of Step5 with some additional parameters to be set for the data distribution. To use this program properly MORSE must use StarPU Runtime system and MPI option must be activated at configure. The data distribution used here is 2-D block-cyclic, see for example ScaLAPACK for explanation. The user can enter the parameters of the distribution grid at execution with --p= option. Example using OpenMPI on four nodes with one process per node:

```
mpirun -np 4 ./step6 --n=10000 --nb=320 --ib=64 \
                     --threads=8 --gpus=2 --p=2
```

In this program we use the tile data layout from PLASMA so that the call

```
MORSE_Desc_Create_User(&descA, NULL, MorseRealDouble,
                       NB, NB, NB*NB, N, N,
                       0, 0, N, N,
                       GRID_P, GRID_Q,
                       morse_getaddr_ccrb,
                       morse_getblkldd_ccrb,
                       morse_getrankof_2d);
```

is equivalent to the following call

```
MORSE_Desc_Create(&descA, NULL, MorseRealDouble,
                  NB, NB, NB*NB, N, N,
                  0, 0, N, N,
                  GRID_P, GRID_Q);
```

functions morse_getaddr_ccrb, morse_getblkldd_ccrb, morse_getrankof_2d being used in Desc_Create. It is interesting to notice that the code is almost the same as Step5. The only additional information to give is the way tiles are distributed through the third function given to MORSE_Desc_Create_User. Here, because we have made experiments only with a 2-D block-cyclic distribution, we have parameters P and Q in the interface of Desc_Create but they have sense only for 2-D block-cyclic distribution and then using

`morse_getrankof_2d` function. Of course it could be used with other distributions, being no more the parameters of a 2-D block-cyclic grid but of another distribution.

4.3.2 List of available routines

4.3.2.1 Auxiliary routines

Reports MORSE version number.

```
int MORSE_Version        (int *ver_major, int *ver_minor, int *ver_micro);
```

Initialize MORSE: initialize some parameters, initialize the runtime and/or MPI.

```
int MORSE_Init           (int nworkers, int ncudas);
```

Finalyze MORSE: free some data and finalize the runtime and/or MPI.

```
int MORSE_Finalize       (void);
```

Return the MPI rank of the calling process.

```
int MORSE_My_Mpi_Rank    (void);
```

Suspend MORSE runtime to poll for new tasks, to avoid useless CPU consumption when no tasks have to be executed by MORSE runtime system.

```
int MORSE_Pause          (void);
```

Symmetrical call to MORSE_Pause, used to resume the workers polling for new tasks.

```
int MORSE_Resume         (void);
```

Conversion from LAPACK layout to tile layout.

```
int MORSE_Lapack_to_Tile (void *Af77, int LDA, MORSE_desc_t *A);
```

Conversion from tile layout to LAPACK layout.

```
int MORSE_Tile_to_Lapack (MORSE_desc_t *A, void *Af77, int LDA);
```

4.3.2.2 Descriptor routines

Create matrix descriptor, internal function.

```
int MORSE_Desc_Create  (MORSE_desc_t **desc, void *mat, MORSE_enum dtyp,
                        int mb, int nb, int bsiz, int lm, int ln,
                        int i, int j, int m, int n, int p, int q);
```

Create matrix descriptor, user function.

```
int MORSE_Desc_Create_User(MORSE_desc_t **desc, void *mat, MORSE_enum dtyp,
                           int mb, int nb, int bsiz, int lm, int ln,
                           int i, int j, int m, int n, int p, int q,
                           void* (*get_blkaddr)( const MORSE_desc_t*, int, int),
                           int (*get_blkldd)( const MORSE_desc_t*, int ),
                           int (*get_rankof)( const MORSE_desc_t*, int, int ));
```

Destroys matrix descriptor.

```
int MORSE_Desc_Destroy (MORSE_desc_t **desc);
```

Ensure that all data are up-to-date in main memory (even if some tasks have been processed on GPUs)

```
int MORSE_Desc_Getoncpu(MORSE_desc_t  *desc);
```

4.3.2.3 Options routines

Enable MORSE feature.

```
int MORSE_Enable  (MORSE_enum option);
```

Feature to be enabled:

- `MORSE_WARNINGS`: printing of warning messages,
- `MORSE_ERRORS`: printing of error messages,
- `MORSE_AUTOTUNING`: autotuning for tile size and inner block size,
- `MORSE_PROFILING_MODE`: activate kernels profiling.

Disable MORSE feature.

```
int MORSE_Disable (MORSE_enum option);
```

Symmetric to `MORSE_Enable`.

Set MORSE parameter.

```
int MORSE_Set     (MORSE_enum param, int  value);
```

Parameters to be set:

- `MORSE_TILE_SIZE`: size matrix tile,
- `MORSE_INNER_BLOCK_SIZE`: size of tile inner block,
- `MORSE_HOUSEHOLDER_MODE`: type of householder trees (FLAT or TREE),
- `MORSE_HOUSEHOLDER_SIZE`: size of the groups in householder trees,
- `MORSE_TRANSLATION_MODE`: related to the `MORSE_Lapack_to_Tile`, see `ztile.c`.

Get value of MORSE parameter.

```
int MORSE_Get     (MORSE_enum param, int *value);
```

4.3.2.4 Sequences routines

Create a sequence.

```
int MORSE_Sequence_Create  (MORSE_sequence_t **sequence);
```

Destroy a sequence.

```
int MORSE_Sequence_Destroy (MORSE_sequence_t *sequence);
```

Wait for the completion of a sequence.

```
int MORSE_Sequence_Wait    (MORSE_sequence_t *sequence);
```

4.3.2.5 Linear Algebra routines

Routines computing linear algebra of the form `MORSE_name[_Tile[_Async]]` (`name` follows LAPACK naming scheme, see http://www.netlib.org/lapack/lug/node24.html availables:

```
/** ****************************************************
 *  Declarations of computational functions (LAPACK layout)
 **/

int MORSE_zgelqf(int M, int N, MORSE_Complex64_t *A, int LDA,
                 MORSE_desc_t *descT);
```

```
int MORSE_zgelqs(int M, int N, int NRHS, MORSE_Complex64_t *A, int LDA,
                 MORSE_desc_t *descT, MORSE_Complex64_t *B, int LDB);

int MORSE_zgels(MORSE_enum trans, int M, int N, int NRHS,
                MORSE_Complex64_t *A, int LDA, MORSE_desc_t *descT,
                MORSE_Complex64_t *B, int LDB);

int MORSE_zgemm(MORSE_enum transA, MORSE_enum transB, int M, int N, int K,
                MORSE_Complex64_t alpha, MORSE_Complex64_t *A, int LDA,
                MORSE_Complex64_t *B, int LDB, MORSE_Complex64_t beta,
                MORSE_Complex64_t *C, int LDC);

int MORSE_zgeqrf(int M, int N, MORSE_Complex64_t *A, int LDA,
                 MORSE_desc_t *descT);

int MORSE_zgeqrs(int M, int N, int NRHS, MORSE_Complex64_t *A, int LDA,
                 MORSE_desc_t *descT, MORSE_Complex64_t *B, int LDB);

int MORSE_zgesv_incpiv(int N, int NRHS, MORSE_Complex64_t *A, int LDA,
                       MORSE_desc_t *descL, int *IPIV,
                       MORSE_Complex64_t *B, int LDB);

int MORSE_zgesv_nopiv(int N, int NRHS, MORSE_Complex64_t *A, int LDA,
                      MORSE_Complex64_t *B, int LDB);

int MORSE_zgetrf_incpiv(int M, int N, MORSE_Complex64_t *A, int LDA,
                        MORSE_desc_t *descL, int *IPIV);

int MORSE_zgetrf_nopiv(int M, int N, MORSE_Complex64_t *A, int LDA);

int MORSE_zgetrs_incpiv(MORSE_enum trans, int N, int NRHS,
                        MORSE_Complex64_t *A, int LDA,
                        MORSE_desc_t *descL, int *IPIV,
                        MORSE_Complex64_t *B, int LDB);

int MORSE_zgetrs_nopiv(MORSE_enum trans, int N, int NRHS,
                       MORSE_Complex64_t *A, int LDA,
                       MORSE_Complex64_t *B, int LDB);

#ifdef COMPLEX
int MORSE_zhemm(MORSE_enum side, MORSE_enum uplo, int M, int N,
                MORSE_Complex64_t alpha, MORSE_Complex64_t *A, int LDA,
                MORSE_Complex64_t *B, int LDB, MORSE_Complex64_t beta,
                MORSE_Complex64_t *C, int LDC);

int MORSE_zherk(MORSE_enum uplo, MORSE_enum trans, int N, int K,
```

```
                double alpha, MORSE_Complex64_t *A, int LDA,
                double beta, MORSE_Complex64_t *C, int LDC);

int MORSE_zher2k(MORSE_enum uplo, MORSE_enum trans, int N, int K,
                 MORSE_Complex64_t alpha, MORSE_Complex64_t *A, int LDA,
                 MORSE_Complex64_t *B, int LDB, double beta,
                 MORSE_Complex64_t *C, int LDC);
#endif

int MORSE_zlacpy(MORSE_enum uplo, int M, int N,
                 MORSE_Complex64_t *A, int LDA,
                 MORSE_Complex64_t *B, int LDB);

double MORSE_zlange(MORSE_enum norm, int M, int N,
                    MORSE_Complex64_t *A, int LDA);

#ifdef COMPLEX
double MORSE_zlanhe(MORSE_enum norm, MORSE_enum uplo, int N,
                    MORSE_Complex64_t *A, int LDA);
#endif

double MORSE_zlansy(MORSE_enum norm, MORSE_enum uplo, int N,
                    MORSE_Complex64_t *A, int LDA);

double MORSE_zlantr(MORSE_enum norm, MORSE_enum uplo, MORSE_enum diag,
                    int M, int N, MORSE_Complex64_t *A, int LDA);

int MORSE_zlaset(MORSE_enum uplo, int M, int N, MORSE_Complex64_t alpha,
                 MORSE_Complex64_t beta, MORSE_Complex64_t *A, int LDA);

int MORSE_zlauum(MORSE_enum uplo, int N, MORSE_Complex64_t *A, int LDA);

#ifdef COMPLEX
int MORSE_zplghe( double bump, int N, MORSE_Complex64_t *A, int LDA,
                  unsigned long long int seed );
#endif

int MORSE_zplgsy( MORSE_Complex64_t bump, int N,
                  MORSE_Complex64_t *A, int LDA,
                  unsigned long long int seed );

int MORSE_zplrnt( int M, int N, MORSE_Complex64_t *A, int LDA,
                  unsigned long long int seed );

int MORSE_zposv(MORSE_enum uplo, int N, int NRHS,
                MORSE_Complex64_t *A, int LDA,
                MORSE_Complex64_t *B, int LDB);
```

```
int MORSE_zpotrf(MORSE_enum uplo, int N, MORSE_Complex64_t *A, int LDA);

int MORSE_zsytrf(MORSE_enum uplo, int N, MORSE_Complex64_t *A, int LDA);

int MORSE_zpotri(MORSE_enum uplo, int N, MORSE_Complex64_t *A, int LDA);

int MORSE_zpotrs(MORSE_enum uplo, int N, int NRHS,
                 MORSE_Complex64_t *A, int LDA,
                 MORSE_Complex64_t *B, int LDB);

#if defined (PRECISION_c) || defined(PRECISION_z)
int MORSE_zsytrs(MORSE_enum uplo, int N, int NRHS,
                 MORSE_Complex64_t *A, int LDA,
                 MORSE_Complex64_t *B, int LDB);
#endif

int MORSE_zsymm(MORSE_enum side, MORSE_enum uplo, int M, int N,
                MORSE_Complex64_t alpha, MORSE_Complex64_t *A, int LDA,
                MORSE_Complex64_t *B, int LDB, MORSE_Complex64_t beta,
                MORSE_Complex64_t *C, int LDC);

int MORSE_zsyrk(MORSE_enum uplo, MORSE_enum trans, int N, int K,
                MORSE_Complex64_t alpha, MORSE_Complex64_t *A, int LDA,
                MORSE_Complex64_t beta, MORSE_Complex64_t *C, int LDC);

int MORSE_zsyr2k(MORSE_enum uplo, MORSE_enum trans, int N, int K,
                 MORSE_Complex64_t alpha, MORSE_Complex64_t *A, int LDA,
                 MORSE_Complex64_t *B, int LDB, MORSE_Complex64_t beta,
                 MORSE_Complex64_t *C, int LDC);

int MORSE_ztrmm(MORSE_enum side, MORSE_enum uplo,
                MORSE_enum transA, MORSE_enum diag,
                int N, int NRHS,
                MORSE_Complex64_t alpha, MORSE_Complex64_t *A, int LDA,
                MORSE_Complex64_t *B, int LDB);

int MORSE_ztrsm(MORSE_enum side, MORSE_enum uplo,
                MORSE_enum transA, MORSE_enum diag,
                int N, int NRHS,
                MORSE_Complex64_t alpha, MORSE_Complex64_t *A, int LDA,
                MORSE_Complex64_t *B, int LDB);

int MORSE_ztrsmpl(int N, int NRHS, MORSE_Complex64_t *A, int LDA,
                  MORSE_desc_t *descL, int *IPIV,
                  MORSE_Complex64_t *B, int LDB);
```

```
int MORSE_ztrsmrv(MORSE_enum side, MORSE_enum uplo,
                  MORSE_enum transA, MORSE_enum diag,
                  int N, int NRHS,
                  MORSE_Complex64_t alpha, MORSE_Complex64_t *A, int LDA,
                  MORSE_Complex64_t *B, int LDB);

int MORSE_ztrtri(MORSE_enum uplo, MORSE_enum diag, int N,
                 MORSE_Complex64_t *A, int LDA);

int MORSE_zunglq(int M, int N, int K, MORSE_Complex64_t *A, int LDA,
                 MORSE_desc_t *descT, MORSE_Complex64_t *B, int LDB);

int MORSE_zungqr(int M, int N, int K, MORSE_Complex64_t *A, int LDA,
                 MORSE_desc_t *descT, MORSE_Complex64_t *B, int LDB);

int MORSE_zunmlq(MORSE_enum side, MORSE_enum trans, int M, int N, int K,
                 MORSE_Complex64_t *A, int LDA,
                 MORSE_desc_t *descT,
                 MORSE_Complex64_t *B, int LDB);

int MORSE_zunmqr(MORSE_enum side, MORSE_enum trans, int M, int N, int K,
                 MORSE_Complex64_t *A, int LDA, MORSE_desc_t *descT,
                 MORSE_Complex64_t *B, int LDB);

/** ****************************************************
 *  Declarations of computational functions (tile layout)
 **/

int MORSE_zgelqf_Tile(MORSE_desc_t *A, MORSE_desc_t *T);

int MORSE_zgelqs_Tile(MORSE_desc_t *A, MORSE_desc_t *T, MORSE_desc_t *B);

int MORSE_zgels_Tile(MORSE_enum trans, MORSE_desc_t *A, MORSE_desc_t *T,
                     MORSE_desc_t *B);

int MORSE_zgemm_Tile(MORSE_enum transA, MORSE_enum transB,
                     MORSE_Complex64_t alpha, MORSE_desc_t *A,
                     MORSE_desc_t *B, MORSE_Complex64_t beta,
                     MORSE_desc_t *C);

int MORSE_zgeqrf_Tile(MORSE_desc_t *A, MORSE_desc_t *T);

int MORSE_zgeqrs_Tile(MORSE_desc_t *A, MORSE_desc_t *T, MORSE_desc_t *B);

int MORSE_zgesv_incpiv_Tile(MORSE_desc_t *A, MORSE_desc_t *L, int *IPIV,
                            MORSE_desc_t *B);
```

```
int MORSE_zgesv_nopiv_Tile(MORSE_desc_t *A, MORSE_desc_t *B);

int MORSE_zgetrf_incpiv_Tile(MORSE_desc_t *A, MORSE_desc_t *L, int *IPIV);

int MORSE_zgetrf_nopiv_Tile(MORSE_desc_t *A);

int MORSE_zgetrs_incpiv_Tile(MORSE_desc_t *A, MORSE_desc_t *L, int *IPIV,
                             MORSE_desc_t *B);

int MORSE_zgetrs_nopiv_Tile(MORSE_desc_t *A, MORSE_desc_t *B);

#ifdef COMPLEX
int MORSE_zhemm_Tile(MORSE_enum side, MORSE_enum uplo,
                     MORSE_Complex64_t alpha, MORSE_desc_t *A,
                     MORSE_desc_t *B, MORSE_Complex64_t beta,
                     MORSE_desc_t *C);

int MORSE_zherk_Tile(MORSE_enum uplo, MORSE_enum trans,
                     double alpha, MORSE_desc_t *A,
                     double beta, MORSE_desc_t *C);

int MORSE_zher2k_Tile(MORSE_enum uplo, MORSE_enum trans,
                      MORSE_Complex64_t alpha, MORSE_desc_t *A,
                      MORSE_desc_t *B, double beta, MORSE_desc_t *C);
#endif

int MORSE_zlacpy_Tile(MORSE_enum uplo, MORSE_desc_t *A, MORSE_desc_t *B);

double MORSE_zlange_Tile(MORSE_enum norm, MORSE_desc_t *A);

#ifdef COMPLEX
double MORSE_zlanhe_Tile(MORSE_enum norm, MORSE_enum uplo, MORSE_desc_t *A);
#endif

double MORSE_zlansy_Tile(MORSE_enum norm, MORSE_enum uplo, MORSE_desc_t *A);

double MORSE_zlantr_Tile(MORSE_enum norm, MORSE_enum uplo,
                         MORSE_enum diag, MORSE_desc_t *A);

int MORSE_zlaset_Tile(MORSE_enum uplo, MORSE_Complex64_t alpha,
                      MORSE_Complex64_t beta, MORSE_desc_t *A);

int MORSE_zlauum_Tile(MORSE_enum uplo, MORSE_desc_t *A);

#ifdef COMPLEX
int MORSE_zplghe_Tile(double bump, MORSE_desc_t *A,
                      unsigned long long int seed);
```

```
#endif

int MORSE_zplgsy_Tile(MORSE_Complex64_t bump, MORSE_desc_t *A,
                      unsigned long long int seed );

int MORSE_zplrnt_Tile(MORSE_desc_t *A, unsigned long long int seed );

int MORSE_zposv_Tile(MORSE_enum uplo, MORSE_desc_t *A, MORSE_desc_t *B);

int MORSE_zpotrf_Tile(MORSE_enum uplo, MORSE_desc_t *A);

int MORSE_zsytrf_Tile(MORSE_enum uplo, MORSE_desc_t *A);

int MORSE_zpotri_Tile(MORSE_enum uplo, MORSE_desc_t *A);

int MORSE_zpotrs_Tile(MORSE_enum uplo, MORSE_desc_t *A, MORSE_desc_t *B);

#if defined (PRECISION_c) || defined(PRECISION_z)
int MORSE_zsytrs_Tile(MORSE_enum uplo, MORSE_desc_t *A, MORSE_desc_t *B);
#endif

int MORSE_zsymm_Tile(MORSE_enum side, MORSE_enum uplo,
                     MORSE_Complex64_t alpha, MORSE_desc_t *A,
                     MORSE_desc_t *B, MORSE_Complex64_t beta,
                     MORSE_desc_t *C);

int MORSE_zsyrk_Tile(MORSE_enum uplo, MORSE_enum trans,
                     MORSE_Complex64_t alpha, MORSE_desc_t *A,
                     MORSE_Complex64_t beta, MORSE_desc_t *C);

int MORSE_zsyr2k_Tile(MORSE_enum uplo, MORSE_enum trans,
                      MORSE_Complex64_t alpha, MORSE_desc_t *A,
                      MORSE_desc_t *B, MORSE_Complex64_t beta,
                      MORSE_desc_t *C);

int MORSE_ztrmm_Tile(MORSE_enum side, MORSE_enum uplo,
                     MORSE_enum transA, MORSE_enum diag,
                     MORSE_Complex64_t alpha, MORSE_desc_t *A,
                     MORSE_desc_t *B);

int MORSE_ztrsm_Tile(MORSE_enum side, MORSE_enum uplo,
                     MORSE_enum transA, MORSE_enum diag,
                     MORSE_Complex64_t alpha, MORSE_desc_t *A,
                     MORSE_desc_t *B);

int MORSE_ztrsmpl_Tile(MORSE_desc_t *A, MORSE_desc_t *L,
                       int *IPIV, MORSE_desc_t *B);
```

```
int MORSE_ztrsmrv_Tile(MORSE_enum side, MORSE_enum uplo,
                       MORSE_enum transA, MORSE_enum diag,
                       MORSE_Complex64_t alpha, MORSE_desc_t *A,
                       MORSE_desc_t *B);

int MORSE_ztrtri_Tile(MORSE_enum uplo, MORSE_enum diag, MORSE_desc_t *A);

int MORSE_zunglq_Tile(MORSE_desc_t *A, MORSE_desc_t *T, MORSE_desc_t *B);

int MORSE_zungqr_Tile(MORSE_desc_t *A, MORSE_desc_t *T, MORSE_desc_t *B);

int MORSE_zunmlq_Tile(MORSE_enum side, MORSE_enum trans, MORSE_desc_t *A,
                      MORSE_desc_t *T, MORSE_desc_t *B);

int MORSE_zunmqr_Tile(MORSE_enum side, MORSE_enum trans, MORSE_desc_t *A,
                      MORSE_desc_t *T, MORSE_desc_t *B);

/** ****************************************
 *  Declarations of computational functions
 *  (tile layout, asynchronous execution)
 **/

int MORSE_zgelqf_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *T,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zgelqs_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *T,
                            MORSE_desc_t *B,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zgels_Tile_Async(MORSE_enum trans, MORSE_desc_t *A,
                           MORSE_desc_t *T, MORSE_desc_t *B,
                           MORSE_sequence_t *sequence,
                           MORSE_request_t *request);

int MORSE_zgemm_Tile_Async(MORSE_enum transA, MORSE_enum transB,
                           MORSE_Complex64_t alpha, MORSE_desc_t *A,
                           MORSE_desc_t *B, MORSE_Complex64_t beta,
                           MORSE_desc_t *C, MORSE_sequence_t *sequence,
                           MORSE_request_t *request);

int MORSE_zgeqrf_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *T,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request)
```

```
int MORSE_zgeqrs_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *T,
                            MORSE_desc_t *B,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zgesv_incpiv_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *L,
                                  int *IPIV, MORSE_desc_t *B,
                                  MORSE_sequence_t *sequence,
                                  MORSE_request_t *request);

int MORSE_zgesv_nopiv_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *B,
                                 MORSE_sequence_t *sequence,
                                 MORSE_request_t *request);

int MORSE_zgetrf_incpiv_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *L,
                                   int *IPIV, MORSE_sequence_t *sequence,
                                   MORSE_request_t *request);

int MORSE_zgetrf_nopiv_Tile_Async(MORSE_desc_t *A,
                                  MORSE_sequence_t *sequence,
                                  MORSE_request_t *request);

int MORSE_zgetrs_incpiv_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *L,
                                   int *IPIV, MORSE_desc_t *B,
                                   MORSE_sequence_t *sequence,
                                   MORSE_request_t *request);

int MORSE_zgetrs_nopiv_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *B,
                                  MORSE_sequence_t *sequence,
                                  MORSE_request_t *request);

#ifdef COMPLEX
int MORSE_zhemm_Tile_Async(MORSE_enum side, MORSE_enum uplo,
                           MORSE_Complex64_t alpha, MORSE_desc_t *A,
                           MORSE_desc_t *B, MORSE_Complex64_t beta,
                           MORSE_desc_t *C, MORSE_sequence_t *sequence,
                           MORSE_request_t *request);

int MORSE_zherk_Tile_Async(MORSE_enum uplo, MORSE_enum trans,
                           double alpha, MORSE_desc_t *A,
                           double beta, MORSE_desc_t *C,
                           MORSE_sequence_t *sequence,
                           MORSE_request_t *request);

int MORSE_zher2k_Tile_Async(MORSE_enum uplo, MORSE_enum trans,
                            MORSE_Complex64_t alpha, MORSE_desc_t *A,
                            MORSE_desc_t *B, double beta, MORSE_desc_t *C,
```

```
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);
#endif

int MORSE_zlacpy_Tile_Async(MORSE_enum uplo, MORSE_desc_t *A,
                            MORSE_desc_t *B, MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zlange_Tile_Async(MORSE_enum norm, MORSE_desc_t *A, double *value,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

#ifdef COMPLEX
int MORSE_zlanhe_Tile_Async(MORSE_enum norm, MORSE_enum uplo,
                            MORSE_desc_t *A, double *value,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);
#endif

int MORSE_zlansy_Tile_Async(MORSE_enum norm, MORSE_enum uplo,
                            MORSE_desc_t *A, double *value,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zlantr_Tile_Async(MORSE_enum norm, MORSE_enum uplo,
                            MORSE_enum diag, MORSE_desc_t *A, double *value,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zlaset_Tile_Async(MORSE_enum uplo, MORSE_Complex64_t alpha,
                            MORSE_Complex64_t beta, MORSE_desc_t *A,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zlauum_Tile_Async(MORSE_enum uplo, MORSE_desc_t *A,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

#ifdef COMPLEX
int MORSE_zplghe_Tile_Async(double bump, MORSE_desc_t *A,
                            unsigned long long int seed,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request );
#endif

int MORSE_zplgsy_Tile_Async(MORSE_Complex64_t bump, MORSE_desc_t *A,
                            unsigned long long int seed,
```

```
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request );

int MORSE_zplrnt_Tile_Async(MORSE_desc_t *A, unsigned long long int seed,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request );

int MORSE_zposv_Tile_Async(MORSE_enum uplo, MORSE_desc_t *A,
                           MORSE_desc_t *B,
                           MORSE_sequence_t *sequence,
                           MORSE_request_t *request);

int MORSE_zpotrf_Tile_Async(MORSE_enum uplo, MORSE_desc_t *A,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zsytrf_Tile_Async(MORSE_enum uplo, MORSE_desc_t *A,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zpotri_Tile_Async(MORSE_enum uplo, MORSE_desc_t *A,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zpotrs_Tile_Async(MORSE_enum uplo, MORSE_desc_t *A,
                            MORSE_desc_t *B, MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

#if defined (PRECISION_c) || defined(PRECISION_z)
int MORSE_zsytrs_Tile_Async(MORSE_enum uplo, MORSE_desc_t *A,
                            MORSE_desc_t *B,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);
#endif

int MORSE_zsymm_Tile_Async(MORSE_enum side, MORSE_enum uplo,
                           MORSE_Complex64_t alpha, MORSE_desc_t *A,
                           MORSE_desc_t *B, MORSE_Complex64_t beta,
                           MORSE_desc_t *C, MORSE_sequence_t *sequence,
                           MORSE_request_t *request);

int MORSE_zsyrk_Tile_Async(MORSE_enum uplo, MORSE_enum trans,
                           MORSE_Complex64_t alpha, MORSE_desc_t *A,
                           MORSE_Complex64_t beta, MORSE_desc_t *C,
                           MORSE_sequence_t *sequence,
                           MORSE_request_t *request);
```

```
int MORSE_zsyr2k_Tile_Async(MORSE_enum uplo, MORSE_enum trans,
                            MORSE_Complex64_t alpha, MORSE_desc_t *A,
                            MORSE_desc_t *B, MORSE_Complex64_t beta,
                            MORSE_desc_t *C, MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_ztrmm_Tile_Async(MORSE_enum side, MORSE_enum uplo,
                           MORSE_enum transA, MORSE_enum diag,
                           MORSE_Complex64_t alpha, MORSE_desc_t *A,
                           MORSE_desc_t *B, MORSE_sequence_t *sequence,
                           MORSE_request_t *request);

int MORSE_ztrsm_Tile_Async(MORSE_enum side, MORSE_enum uplo,
                           MORSE_enum transA, MORSE_enum diag,
                           MORSE_Complex64_t alpha, MORSE_desc_t *A,
                           MORSE_desc_t *B, MORSE_sequence_t *sequence,
                           MORSE_request_t *request);

int MORSE_ztrsmpl_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *L, int *IPIV,
                             MORSE_desc_t *B, MORSE_sequence_t *sequence,
                             MORSE_request_t *request);

int MORSE_ztrsmrv_Tile_Async(MORSE_enum side, MORSE_enum uplo,
                             MORSE_enum transA, MORSE_enum diag,
                             MORSE_Complex64_t alpha, MORSE_desc_t *A,
                             MORSE_desc_t *B, MORSE_sequence_t *sequence,
                             MORSE_request_t *request);

int MORSE_ztrtri_Tile_Async(MORSE_enum uplo, MORSE_enum diag,
                            MORSE_desc_t *A,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zunglq_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *T,
                            MORSE_desc_t *B,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zungqr_Tile_Async(MORSE_desc_t *A, MORSE_desc_t *T,
                            MORSE_desc_t *B,
                            MORSE_sequence_t *sequence,
                            MORSE_request_t *request);

int MORSE_zunmlq_Tile_Async(MORSE_enum side, MORSE_enum trans,
                            MORSE_desc_t *A, MORSE_desc_t *T,
                            MORSE_desc_t *B, MORSE_sequence_t *sequence,
                            MORSE_request_t *request);
```

```
int MORSE_zunmqr_Tile_Async(MORSE_enum side, MORSE_enum trans,
                            MORSE_desc_t *A, MORSE_desc_t *T,
                            MORSE_desc_t *B, MORSE_sequence_t *sequence,
                            MORSE_request_t *request);
```