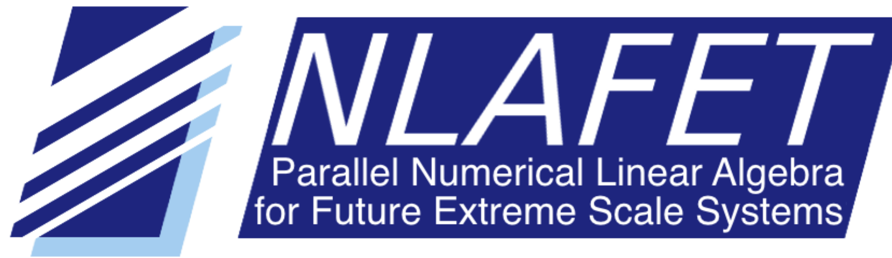


H2020–FETHPC–2014: GA 671633

D7.8

Release of the NLAFET library

April 2019

Document information

Scheduled delivery 2019-04-30
Actual delivery 2019-04-29
Version 2.0
Responsible partner UMU

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
20/02/2019	Bo Kågström	Draft	0.1	First layout of structure
27/03/2019	UMU: BK and MM	Draft	0.1	Draft of section 3: Dense eigenvalue problems—tools and solvers
03/04/2019	BK	Draft	0.1	Drafts and updates of contributions from INRIA (Sec 5), STFC (Sec 4) and UMU (Sec 3, Sec 6 + Intro etc)
23/04/2019	BK	Draft	0.2	Further updates of partner contributions
29/04/2019	Bo Kågström	Final	2.0	Revised for submission

Author(s)

- Bo Kågström, Mirko Myllykoski, Lars Karlsson, and Carl Christian Kjelgaard Mikkelsen, UMU.
- Sébastien Cayrols, Iain Duff, Florent Lopez, and Stojce Nakov, STFC.
- Srikara Pranesh, David Stevens and Jack Dongarra, UNIMAN.
- Simplice Donfack, Laura Grigori, and Olivier Tissot, INRIA.

Internal reviewers

- Authors from the list above have also reviewed parts of the different drafts.

Contributors

Besides the authors of this report, the following members of the NLAFET Team have made important contributions to the software presented and available at GitHub/NLAFET:

- Björn Adlerborn, Mahmoud Eljammaly, and Angelika Schwarz, UMU.

- Jonathan Hogg, STFC.
- Mawussi Zounon, NAG.
- Alan Ayala, INRIA.

We also recognize important contributions by the following collaboration partners.

- The Innovative Computing Laboratory (ICL), the University of Tennessee.
- Jim Demmel at the University of California Berkeley.

Copyright

This work is ©by the NLAFET Consortium, 2015–2019. Its duplication is allowed only for personal, educational, or research uses.

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

1 Introduction 4
1.1 NLAFET repositories on GitHub . . . 4

2 Dense factorizations and solvers 5
2.1 `BBLAS` . . . 5
2.1.1 Installation . . . 5
2.1.2 Testing . . . 6
2.2 `plasma and PlaStar` . . . 6

3 Dense eigenvalue problems—tools and solvers 7
3.1 MPI based eigenvalue solvers . . . 7
3.2 StarNEig library . . . 8
3.2.1 Documentation . . . 9
3.2.2 Installation . . . 9
3.2.3 Usage . . . 10
3.2.4 Distributed memory . . . 11
3.2.5 ScaLAPACK compatibility layer . . . 13
3.2.6 Computational interface functions . . . 13
3.2.7 Test program and an example . . . 15

4 Sparse direct factorizations and solvers 16
4.1 `SpLLT` . . . 16
4.2 `SyLVER` . . . 17
4.3 `ParSHUM` . . . 17
4.4 `BC` . . . 19

5 Communication optimal algorithms for iterative methods 19
5.1 preAlps library . . . 19

6 Cross-cutting tools 21
6.1 The experimental PCP runtime library . . . 21
6.2 One-sided factorizations with algorithm-based fault tolerance . . . 23

7 Summary and future contributions 23

List of Tables

1 NLAFET GitHub public repositories . . . 5
2 Current status of the StarNEig library for standard eigenvalue problems. . 9
3 Current status of the StarNEig library for generalized eigenvalue problems. 9

1 Introduction

The *Description of Action* (DoA) document states for deliverable D7.8:

> "*Release of the NLAFET library*
>
> First complete release of the NLAFET library and associated User's Guide."

This deliverable is in the context of Task 7.3 (Open source activities) and extends on the deliverable D7.5 *Beta release of the NLAFET library* with subtitle *Prototype software – Part 1.* Moreover, the deliverable D7.8 is based on the software, with some new contributions and further improvements, described in the following 13 deliverables:

- D2.4 Final prototype software for different versions of the BLAS (M36)
- D2.6 Prototype software for eigenvalue problem solvers (M30)
- D2.8 Bidiagonal factorization (M18)
- D2.9 Novel SVD algorithms (M42)
- D3.3 Software for symmetrically structured factorizations (M39)
- D3.5 Software for highly unsymmetric factorizations (M30)
- D3.7 Software for hybrid methods (M40)
- D4.1 Prototype software, phase 1 (M12)
- D4.3 Prototype software, phase 2 (M24)
- D4.5 Integration (M36)
- D5.2 Software integration (M30)
- D6.1 Prototypes for runtime systems exhibiting novel types of scheduling (M18)
- D6.7 Prototypes for tiled one-sided factorizations with algorithm-based fault tolerance (M42)

1.1 NLAFET repositories on GitHub

The codes developed and distributed in the context of the NLAFET project are hosted on the GitHub platform and available via the link `https://github.com/NLAFET/`. In Table 1, the public repositories that together constitute the NLAFET library software are listed. The description of these repositories is organized according to the following topics:

- Dense matrix factorizations and solvers (Section 2).
- Solvers and tools for standard and generalized dense eigenvalue problems (Section 3).
- Sparse direct factorizations and solvers (Section 4).
- Communication optimal algorithms for iterative methods (Section 5).
- Cross-cutting tools (Section 6).

Table 1: NLAFET GitHub public repositories

Repository name	WP	Brief description
BBLAS-ref	WP2	Batched BLAS reference implementation
plasma	WP2	Snapshot of PLASMA library
PlaStar	WP2	Some StarPU implementations
SEVP-PDHSEQR-Alg953	WP2	Standard eigenvalue problem solvers
GEVP-PDHGEQZ	WP2	Generalized eigenvalue problem solvers
StarNEig	WP2	StarPU nonsymmetric eigenvalue problem solvers
SpLLT	WP3	Sparse LL^T solver for $A = A^T$, positive definite
SyLVER	WP3	Symmetrically stuctured factorizations
ParSHUM	WP3	Sparse LU solver for highly unsymmetric matrices
BC	WP3	Hybrid solver based on block Cimmino for square and rectangular systems
preAlps	WP4	Preconditioned iterative methods and enlarged Krylov methods
PCP-runtime	WP6	Parallelizing the critical path
ABFT-factor	WP6	Tiled factorizations with ABFT

2 Dense factorizations and solvers

The NLAFET public repositories for dense factorizations and solvers are:

- `BBLAS`: Final batched BLAS reference implementation.
- `plasma`: Snapshot of PLASMA library routines for dense factorizations and solves.
- `PlaStar`: Some StarPU implementations of dense factorizations and solves.

2.1 `BBLAS`

The specifications for the level 1, 2 and 3 BLAS have been very successful in providing a standard for vector [26], matrix-vector [17], [16], and matrix-matrix [15] operations respectively. Vendors and other developers have provided highly efficient versions of the BLAS, and by using the standard interface have allowed software calling the BLAS to be portable.

With the need to solve larger and larger problems on todays high-performance computers, the methods used in a number of applications such as tensor contractions, finite element methods and direct linear equation solvers, require a large number of small vector or matrix operations to be performed in parallel. So a typical example might be to perform

$$C_i \leftarrow \alpha_i A_i B_i + \beta_i C_i, \quad i = 1, 2, 3, \ldots, k,$$

where k is large, but A_i, B_i and C_i are small matrices. A routine to perform such a sequence of operations is called a Batched Basic Linear Algebra Subprogram, or Batched BLAS, or BBLAS. In the `BBLAS` repository we provide a reference implementation of the BBLAS standard for the four standard precisions.

2.1.1 Installation

The following software must be installed on the target architecture:

- CMake 3.12 or latest.

- Intel Math Kernel Library (MKL), or Netlib BLAS or Netlib LAPACK.
- Doxygen for documentation.

After the configuration of make.inc, the software can be compiled using one of the following options

- `make [all]` – compiles lib test
- `make lib` – generates library files in lib/libbblas.{a,so} lib/libcore.{a,so}
- `make test` – generates tester files in test/test
- `make docs` – generates documentation docs/html
- `make generate` – generates routines of various precisions
- `make clean` – removes objects, libraries, and executables
- `make distclean` – removes above, Makefile.*.gen, and anything else that can be generated

2.1.2 Testing

At the end of the compilation, test routines (in four precisions [`c,d,s,z`]) will be available in the folder `BBLAS/test`. The main testing driver is the binary `./test`. It should be used with the kernel to test as the first parameter followed by the kernel arguments. For example `./test dgemm_batch` will run the double precision version `dgemm_batch` with default arguments. For help `./test -h` will display the list of kernels available for testing, while `./test dgemm_batch -h` will display all the possible more help and details on how to test `dgemm_batch`; this holds for all the kernels. We use random matrices for the tests. All the implementation and technical details can be found in the NLAFET deliverable report D7.6 *Batched BLAS Specification*.

2.2 `plasma` and `PlaStar`

The PLASMA numerical library (Parallel Linear Algebra Software for Multicore Architectures) is a dense linear algebra package tailored for multicore computing. It was developed as a response to the advent of multicore processors, as the standard state of the art solvers such as LAPACK [8] and ScaLAPACK [9], which proved to be inefficient on these new architectures. Initially PLASMA was developed based on the QUARK scheduler, along with Pthread-based routines. Since the OpenMP standard adopted the superscalar scheduling, PLASMA library has been ported into OpenMP. This work was carried out as a part of NLAFET project, and the software is available in the `plasma` repository. Installation and testing procedure is same as for the `BBLAS` software. However in addition to MKL, OpenMP is also required for installation. All the technical details can be found in the NLAFET deliverable report D2.1 *One-Sided Matrix Factorizations*.

In the `PlaStar` repository we provide LU, Cholesky, and QR factorization routines in the PLASMA library built on StarPU runtime system. We would like to emphasize that StarPU based PLASMA was purely exploratory, and was not intended to be a complete dense linear algebra software on its own.

3 Dense eigenvalue problems—tools and solvers

The NLAFET public repositories for the tools and solvers of dense standard and generalized eigenvalue problems are:

- `SEVP-PDHSEQR-Alg953`: Nonsymmetric standard eigenvalue problems—distributed memory package for computing a real standard Schur form $S = Q^T AQ$.
- `GEVP-PDHGEQZ`: Nonsymmetric generalized eigenvalue problems—distributed memory package for computing a real generalised Schur form $(S, T) = Q^T (A, B)Z$.
- `StarNEig`: Task-based library including a full suite of software for solving nonsymmetric standard ($Ax = \lambda x$) and generalized ($Ax = \lambda Bx$) eigenvalue problems, respectively.

Since, the task-based library `StarNEig` has not been presented in any earlier NLAFET deliverable, this motivates the longer description compared to other repositories.

The parallel distributed algorithms and software contributions for computing Schur forms of standard and generalized eigenvalue problems, and available in the first two listed repositories above, represent the front line of research using the common MPI based programming model. Within the NLAFET project, we provide new contributions that both improve on and extend the functionalities to be able to efficiently compute eigenvectors corresponding to an arbitrary user selection of eigenvalues. Moreover, in our effort of addressing extreme scale systems we are developing task-based counter parts of the functionalities already available in the `SEVP-PDHSEQR-Alg953` and `GEVP-PDHGEQZ` repositories. The main new contributions are available in the `StarNEig` library, which is built on top of the StarPU runtime system [1] and targets both shared memory and distributed HPC systems. Some components of the library also support GPUs.

The two first repositories are briefly presented in deliverable D7.5. For easy access and completeness some information is repeated below. The focus here is on an introductory presentation of the task based `StarNEig` library. For details we refer to the GitHub on-line documentations.

3.1 MPI based eigenvalue solvers

`SEVP-PDHSEQR-Alg953`: The repository contains a clone of ALGORITHM 953 of the *Collected Agorithms of ACM* authored by Robert Granat, Bo Kågström, Daniel Kressner and Meiyue Shao, which is a *state-of-the-art package of library routines* for computing a standard Schur decomposition $S = Q^T AQ$, where A is a general square matrix with real entries and Q is an orthogonal transformation matrix. The associated scientific paper is published by ACM Transactions of Mathematical Software [22].

The main routine `PDHGEQR()` is based on our parallel multishift QR algorithm with aggressive early deflation for computing the standard real Schur decomposition S. The real and complex conjugate pairs of eigenvalues appear as 1×1 and 2×2 blocks, respectively, along the diagonal of S and can be reordered in any order. Typically, this functionality is used to compute an orthogonal basis for an invariant subspace corresponding to a selected set of eigenvalues. The parallel algorithms and software are developed by the Umeå team in collaboration with EPFL. The software is MPI based, written in Fortran 90 for double precision real arithmetic, and targets distributed memory HPC systems.

The software is complemented by a PDHGEQR User's Guide, available in the repository, which includes sections on installation, compilation and usage of the parallel library routines. The Guide also contains instructions and scripts for tuning of parameters.

`GEVP-PDHGEQZ`: The repository contains a *state-of-the-art package of library routines* for computing a generalized Schur decomposition $(S, T) = Q^T(A, B)Z$, where A and B are general square matrices with real entries and Q and Z are orthogonal transformation matrices. The real and complex conjugate pairs of eigenvalues appear as 1×1 and 2×2 blocks, respectively, along the diagonals of (S, T) and can be reordered in any order. Typically, this functionality is used to compute orthogonal bases for a pair of deflating subspaces corresponding to a selected set of eigenvalues. The parallel algorithms and software are developed by the Umeå team in collaboration with EPFL (Daniel Kressner) and based on our scientific work published by SIAM Journal on Scientific Computing [2]. The software is MPI based, written in Fortran 90 for double precision real arithmetic, and targets distributed memory HPC systems.

The main routine `PDHGEQZ()` is based on our parallel multishift QZ algorithm with aggressive early deflation for computing the generalized real Schur form of a matrix pair (A, B) in Hessenberg-triangular (HT) form [2]. A novel parallel algorithm for identifying and deflating infinite eigenvalues such that they do not inflict damage to other eigenvalues, due to round-off, is part of the software.

The software is complemented by a User's Guide, namely NLAFET Working Note WN-2 *PDGEQZ User Guide* [3]. The calling sequences for the main driver routines with input and output parameters are described in detail. In addition, a set of tunable parameters and their usage with default values are discussed. During the build process, internal tests are performed to ensure the software works as intended. Both sequential and parallel tests are performed, with validation of the computed results.

The package also includes our MPI-based implementation, `PDGGHRD()`, for the reduction of a real general matrix pair (A, B) to Hessenberg-triangular form (H, T), which is the first main step in the computation of a generalized Schur form (S, T) of (A, B). PDGGHRD is a one-stage distributed algorithm with wave-front scheduling. The static scheduler addresses the problem of underutilized processes caused by two-sided updates of matrix pairs based on sequences of rotations. For more details see the NLAFET Working Note WN-1 *Distributed One-Stage Hessenberg-Triangular Reduction with Wavefront Scheduling* [5] and the recent SIAM Journal of Scientific Computing article [4].

3.2 StarNEig library

The StarNEig library aims to provide a full suite of algorithms for solving *non-symmetric* standard and generalized eigenvalue problems. In practice a *matrix pencil* $A - \lambda B$ is treated as a *matrix pair* (A, B) and both concepts are used in the descriptions. As mentioned, the library is built on top of the *StarPU* runtime system and targets both shared memory and distributed memory machines. Some components of the library support GPUs. The four main components of the library are:

Hessenberg(-triangular) reduction: A dense matrix (or a dense matrix pair) is reduced to upper Hessenberg (or Hessenberg-triangular) form.

Schur reduction: An upper Hessenberg matrix (or a Hessenberg-triangular matrix pair) is reduced to (generalized) Schur form. The (generalized) eigenvalues can be determined from the diagonal blocks of the matrices in (generalized) Schur form.

Table 2: Current status of the StarNEig library for standard eigenvalue problems.

Component	Shared memory	Distributed memory	Accelerators (GPUs)
Hessenberg	Complete	ScaLAPACK wrapper	Single GPU
Schur	Complete	Experimental	Experimental
Reordering	Complete	Complete	Experimental
Eigenvectors	Complete	Integration ongoing	Not planned

Table 3: Current status of the StarNEig library for generalized eigenvalue problems.

Component	Shared memory	Distributed memory	Accelerators (GPUs)
Hessenberg	LAPACK wrapper	ScaLAPACK wrapper	Not planned
Schur	Complete	Experimental	Experimental
Reordering	Complete	Complete	Experimental
Eigenvectors	Complete	Integration ongoing	Not planned

Eigenvalue reordering: Reorders a user-selected set of (generalized) eigenvalues to the upper left corner of an updated (generalized) Schur form.

Eigenvectors: Computes (generalized) eigenvectors for a user-selected set of (generalized) eigenvalues.

The library is currently in a beta state and supports real arithmetic, which indeed is the most challenging case since matrices and matrix pairs with only real entries can have both real and complex conjugate pairs of eigenvalues and associated eigenvectors. In addition, some interface functions are implemented as LAPACK and ScaLAPACK wrappers. The overall status of the library is summarized in Tables 2 and 3. In NLAFET deliverable D2.7 *Eigenvalue solvers for nonsymmetric problems*, the StarNEig library routines are evaluated both with respect to performance (scalability) and accuracy and compared with corresponding state-of-the-art MPI-based routines.

3.2.1 Documentation

The User's Guide can be generated independently from the rest of the library. This assumes that the target HPC system has the following software dependencies installed:

- CMake 3.3 or newer, Doxygen, and Latex + pdflatex

It is recommended that a user builds the documentation in a separate build directory:

```
$ mkdir build_doc
$ cd build_doc/
$ cmake ../doc/
$ make
```

The HTML documentation is available in the `build_doc/html` directory. The PDF documentation, consisting of around 130 pages, is copied to `build_doc/starneig_manual.pdf`.

3.2.2 Installation

The library assumes that the machine has the following software dependences, including libraries and run-time systems, installed:

- Linux
- CMake 3.3 or newer
- Portable Hardware Locality (hwloc)
- StarPU 1.2 or 1.3
- BLAS
- LAPACK
- MPI (optional)
- CUDA (optional)
- ScaLAPACK (optional)
- pkg-config (test program and examples)
- MAGMA (test program, optional)

It is recommended that a user builds the library in a separate build directory:

```
$ mkdir build
$ cd build
```

The library is configured with the `cmake` command. In most cases, it is not necessary to give this command any additional arguments:

```
$ cmake ../
...
-- Configuring done
-- Generating done
-- Build files have been written to: /.../build
```

However, the library can be customized with various options. Please see the User's Guide for further information. The library (and other components) are compiled with the `make` command:

```
$ make
Scanning dependencies of target starneig
[  1%] Building C object src/CMakeFiles/starneig.dir/common/combined.c.o
[  2%] Building C object src/CMakeFiles/starneig.dir/common/common.c.o
...
```

The library and the related header files are installed by executing:

```
$ sudo make install
```

3.2.3 Usage

In order to use the interface function provided by the library, a user can simply include all header files as follows:

```
#include <starneig/starneig.h>
```

Each node must call the `starneig_node_init()` interface function to initialize the library and the `starneig_node_finalize()` interface function to shutdown the library:

```
starneig_node_init(cores, gpus, flags);

...

starneig_node_finalize();
```

The `starneig_node_init()` interface function initializes StarPU (and cuBLAS) and pauses all worker threads. The `cores` argument specifies the total number of used CPU cores. In distributed memory mode, one of these CPU cores is automatically allocated for the StarPU-MPI communication thread. The `gpus` argument specifies the total number of GPUs to be used. One or more CPU cores are automatically allocated for GPU devices.

Figure 1: Matrix divided into rectangular blocks of uniform size.

Figure 2: Example of a block mapping: the rank 0 owns blocks (0,1), (1,2), (1,5), (1,6), (2,6), (3,0) and (3,5).

The `flags` argument can provide additional configuration information. A node can also be configured with default values:

```
starneig_node_init(-1, -1, STARNEIG_DEFAULT);
```

This argument combination tells the library to use all available CPU cores and GPUs.

Notice, the StarPU performance models must be calibrated before the software can function efficiently on heterogeneous platforms (CPUs + GPUs). The calibration is triggered automatically if the models are not calibrated well enough for a given problem size. This may impact the execution time negatively during the first run. Please, see the StarPU handbook for further information: `http://starpu.gforge.inria.fr/doc/html/Scheduling.html`

3.2.4 Distributed memory

The `STARNEIG_HINT_DM` initialization flag tells the library to configure itself for distributed memory computation. The flag is indented to only be a hint and the library will automatically reconfigure itself for the correct computation mode. A user is allowed to mix shared memory and distributed memory functions without reninitializing the library. The library is intended to be run in a *hybrid configuration* (each MPI rank is mapped to several CPU cores). Failing to do so leads to CPU core oversubscription. It is generally a good idea to map each MPI rank to a single node or a NUMA island / CPU socket. The library assumes that the MPI library is already initialized when the `starneig_node_init()` interface function is called with the `STARNEIG_HINT_DM flag` or when the library reconfigures itself for distributed memory after a user has called a distributed memory interface function.

Distributed matrices are represented using two opaque objects:

- Data distribution
- Distributed matrix

Each matrix is partitioned into rectangular blocks of uniform size (excluding the last block row and column) as illustrated in Figure 1. The blocks are indexed using a two-dimensional index space. A data distribution encapsulates an arbitrary mapping from this two-dimensional block index space to the one-dimensional MPI rank space as illustrated

in Figure 2. Naturally, a data distribution can describe a two-dimensional block cyclic distribution which is commonly used in ScaLAPACK subroutines.
A data distribution can be created using one of the following interface functions:

- `starneig_distr_init()` creates a default distribution (row-major ordered two-dimensional block cyclic distribution with a squarish process mesh).
- `starneig_distr_init_mesh()` creates a row-major or column-major ordered two-dimensional block cyclic distribution with desired number of rows and columns in the process mesh.
- `starneig_distr_init_func()` creates an arbitrary distribution defined by a function.

For example,

```
starneig_distr_t distr =
    starneig_distr_init_mesh(4, 6, STARNEIG_ORDER_DEFAULT);
```

creates a two-dimensional block cyclic distribution with 4 rows and 6 columns in the process mesh. Alternatively, a user can create an equivalent data distribution using the `starneig_distr_init_func()` interface function:

```
struct block_cyclic_arg {
    int rows;
    int cols;
};

int block_cyclic_func(int i, int j, void *arg)
{
    struct block_cyclic_arg *mesh = arg;
    return (i % mesh->rows) * mesh->cols + j % mesh->cols;
}

void func(...)
{
    ...

    struct block_cyclic_arg arg = { .rows = 4, .cols = 6 };
    starneig_distr_t distr =
        starneig_distr_init_func(&block_cyclic_func, &arg, sizeof(arg));

    ...
}
```

A distributed matrix is created using the `starneig_distr_matrix_create()` interface function. The function call will automatically allocate the required local resources. For example,

```
starneig_distr_t distr =
    starneig_distr_init_mesh(4, 6, STARNEIG_ORDER_DEFAULT);
starneig_distr_matrix_t dA = starneig_distr_matrix_create(
    m, n, bm, bn, STARNEIG_REAL_DOUBLE, distr);
```

creates an $m \times n$ double-precision real matrix that is distributed in a two-dimensional block cyclic fashion in $bm \times bn$ blocks. Or,

```
starneig_distr_matrix_t dB =
    starneig_distr_matrix_create(n, n, -1, -1, STARNEIG_REAL_DOUBLE, NULL);
```

creates an $n \times n$ double-precision real matrix with a default data distribution (`NULL` argument) and a default block size (`-1, -1`).

3.2.5 ScaLAPACK compatibility layer

The library provides a ScaLAPACK compatibility layer. This allows a user to easily integrate our library with their existing ScaLAPACK compatible software. A two-dimensional block cyclic data distribution can be converted to a BLACS context and vice versa. Similarly, a distributed matrix that uses a two-dimensional block cyclic data distribution can be converted to a BLACS descriptor (and a local buffer) and vice versa. The conversion is performed in-place and a user is allowed to mix `StarNEig` interface functions with ScaLAPACK style subroutines/functions without reconversion.

For example,

```
starneig_distr_matrix_t dA = starneig_distr_matrix_create(...);

...

// convert the data distribution to a BLACS context
starneig_distr_t distr = starneig_distr_matrix_get_distr(A);
starneig_blacs_context_t context = starneig_distr_to_blacs_context(distr);

// convert the distributed matrix to a BLACS descriptor and a local buffer
starneig_blacs_descr_t descr_a;
double *local_a;
starneig_distr_matrix_to_blacs_descr(
    dA, context, &descr_a, (void **)&local_a);

...

// a ScaLAPACK subroutine for reducing general distributed matrix to upper
// Hessenberg form
extern void pdgehrd_(int const *, int const *, int const *, double *,
    int const *, int const *, starneig_blacs_descr_t const *, double *,
    double *, int const *, int *);

pdgehrd_(&n, &ilo, &ihi, local_a, &ia, &ja, &descr_a, tau, ...);
```

converts a distributed matrix `dA` to a BLACS descriptor `descr_a` and a local pointer `local_a`. The descriptor and the local array are then fed to a ScaLAPACK subroutine.

3.2.6 Computational interface functions

The interface functions for the main components of the standard eigenvalue problem $Ax = \lambda x$, targeting shared memory (`SM`) and distributed memory (`DM`), are briefly described below.

Hessenberg reduction: Given a general matrix A, the interface functions

- `starneig_SEP_SM_Hessenberg()` and
- `starneig_SEP_DM_Hessenberg()`

compute a Hessenberg decomposition $A = UHU^T$, where H is upper Hessenberg and U is orthogonal. On exit, A is overwritten by H and Q (which is an orthogonal matrix on entry) is overwritten by $Q \leftarrow QU$.

Schur reduction: Given a Hessenberg decomposition $A = QHQ^T$, of a general matrix A, the interface functions

- `starneig_SEP_SM_Schur()` and

- `starneig_SEP_DM_Schur()`

compute a Schur decomposition $A = Q(USU^T)Q^T$, where S is upper quasi-triangular with 1×1 and 2×2 blocks on the block diagonal (Schur matrix) and U is orthogonal. On exit, H is overwritten by S and Q is overwritten by $Q \leftarrow QU$.

Eigenvalue reordering: Given a Schur decomposition $A = QSQ^T$ of a general matrix A and a user selection of eigenvalues, the interface functions

- `starneig_SEP_SM_ReorderSchur()` and
- `starneig_SEP_DM_ReorderSchur()`

attempt to reorder the selected eigenvalues to the top left corner of an updated Schur matrix $\hat{S}$ by an orthogonal similarity transformation $A = Q(U\hat{S}U^T)Q^T$. On exit, S is overwritten by $\hat{S}$ and Q is overwritten by $Q \leftarrow QU$.

Combined reduction and reordering: Given a general matrix A, the interface functions

- `starneig_SEP_SM_Reduce()` and
- `starneig_SEP_DM_Reduce()`

compute a (reordered) Schur decomposition $A = USU^T$, where S is upper quasi-triangular with 1×1 and 2×2 blocks on the block diagonal (Schur matrix) and U is orthogonal. Optionally, the interface functions attempt to reorder selected eigenvalues to the top left corner of the Schur matrix S.

Eigenvectors: Given a Schur decomposition $A = QSQ^T$ of a general matrix A and a (user) selection of eigenvalues, the interface functions

- `starneig_SEP_SM_Eigenvectors()` and
- `starneig_SEP_DM_Eigenvectors()`

compute and return an eigenvector for each of the selected eigenvalues.

The library provides a similar set of interface functions for the generalized eigenvalue problem, both for shared and distributed memory, respectively:

- `starneig_GEP_SM_HessenbergTriangular()` and
- `starneig_GEP_DM_HessenbergTriangular()`,
- `starneig_GEP_SM_Schur()` and
- `starneig_GEP_DM_Schur()`,
- `starneig_GEP_SM_Reduce()` and
- `starneig_GEP_DM_Reduce()`,
- `starneig_GEP_SM_Eigenvectors()` and
- `starneig_GEP_DM_Eigenvectors()`

One main difference between the two cases is that for the standard eigenvalue problem the reduction operations in the main components are by *similarity transformations* of type $Q^T AQ$, where Q is an orthonormal matrix, while for the generalized eigenvalue problem the corresponding reduction operations work on matrix pairs (A, B) via *equivalence transformations* of type $Q^T(A, B)Z$, where both Q and Z are orthonormal. In both cases, the matrices are modified both (block) rowwise and (block) columnwise which lead to complex data dependencies and challenges in the design of task based algorithms aiming at extreme scale.

3.2.7 Test program and an example

The `StarNEig` test program

```
1 $ ./starneig-test (options)
```

provides a unified interface to test all software components. Most command line arguments have default values and in most cases it is not necessary to set more than a few. The overall design of the test program is modular. Each experiment module is built on *initializers*, *solvers* and *hooks*. Each experiment module contains several of each allowing a user to initialize the data in various ways and compare different solvers with each other. Each of these building blocks can be configured with various parameters. However, in most cases only the problem dimension `-n (num)` needs to be specified. Hooks are used to test and validate the output of the software components. For details see the Test program and Examples sections in the User's Guide.

Using a shared memory HPC system, below is a slightly stripped-down example of how to call StarNEig routines for solving a generalized eigenvalue problem. Outgoing from a matrix pencil $A - \lambda B$, the entire chain of the main component algorithms for matrix pairs (A, B) is used to calculate all eigenvalues with positive real part and associated eigenvectors.

```
1 // a predicate function that selects all finite eigenvalues that have
2 // a positive real part
3 static int predicate(double real, double imag, double beta, void *arg)
4 {
5     return beta != 0.0 && 0.0 < real;
6 }
7
8 void main()
9 {
10     double *A, *B, *Q, *Z, *X;
11     int ldA, ldB, ldQ, ldZ, ldX;
12
13     ...
14
15     // Initialize the StarNEig library using a default number of CPU
16     // cores and GPUs. The STARNEIG_HINT_SM flag indicates that the
17     // library should initialize itself for shared memory computations
18     // and the STARNEIG_AWAKE_WORKERS indicates that the library should
19     // keep StarPU worker threads awake between interface function calls.
20
21     starneig_node_init(-1, -1, STARNEIG_HINT_SM | STARNEIG_AWAKE_WORKERS);
22
23     // reduce the dense-dense matrix pair (A,B) to Hessenberg-triangular
24     // form
25
26     starneig_GEP_SM_HessenbergTriangular(
```

```
27          n, A, ldA, B, ldB, Q, ldQ, Z, ldZ);
28
29      // reduce the Hessenberg-triangular matrix pair (A,B) to generalized
30      // Schur form
31
32      starneig_GEP_SM_Schur(
33          n, A, ldA, B, ldB, Q, ldQ, Z, ldZ, real, imag, beta);
34
35      // selects all finite eigenvalues that have a positive real part
36
37      int num_selected;
38      starneig_GEP_SM_Select(
39          n, A, ldA, B, ldB, &predicate, NULL, select, &num_selected);
40
41      // compute eigenvectors corresponding to the selected set of
     eigenvalues
42
43      starneig_GEP_SM_Eigenvectors(
44          n, select, A, ldA, B, ldB, Q, ldQ, X, ldX);
45
46      // de-initialize the StarNEig library
47
48      starneig_node_finalize();
49  }
```

More detailed examples including validation via accuracy residual tests are presented in the User's Guide.

4 Sparse direct factorizations and solvers

We gave details of the library software for sparse direct factorizations and software that were developed by 30 April 2017 in Deliverable D7.5. This work is part of Task 3.2 of workpackage WP 3.

The NLAFET public repositories for sparse direct factorizations and solvers are:

- `SpLLT`: Sparse LL^T solver for $A = A^T$, where A is positive definite.
- `SyLVER`: Symmetrically structured factorizations.
- `ParSHUM`: Sparse LU solver for highly unsymmetric matrices.
- `BC`: Hybrid solver based on block Cimmino for square and rectangular systems.

We now discuss the software in these four repositories using the earlier deliverable as our baseline.

4.1 `SpLLT`

As discussed in Deliverable D7.5, the supernodal sparse Cholesky solver `SpLLT`, is implemented as a task-based algorithm and has STF versions using OpenMP and StarPU and a PTG version using ParSec. The use of runtime systems means that the code is easier to port between different architectures and we showed that there was very little overhead to using the runtime systems. Indeed on large problems our code outperformed similar state-of-the-art solvers that just used OpenMP. Since then there have been very significant enhancements to the `SpLLT` solver.

A major improvement was to develop a blocked parallel implementation of the routines that use the Cholesky factorization to solve the sparse symmetric positive definite system. This work was presented in NLAFET Working Note WN-20 *Parallelization of the solve phase in a task-based Cholesky solver using a sequential task flow model* [13]. This is particularly important when solutions of several different equations are required, for example in optimization. This is also true if the direct solver is used in an iterative method, and we show an example of this in Section 4.4. This solve code was also upgraded for a block of right-hand sides and got a significant performance enhancement by vectorizing across the right-hand sides.

4.2 `SyLVER`

The SyLVER repository contains codes for the factorization of sparse symmetrically structured matrices and the solution of equations with these as a coefficient matrix.

The codes are task-based and use the multifrontal approach that is based upon an elimination tree representation of the sparse factorization. The matrix is ordered using a standard sparsity reducing ordering such as nested dissection that is performed on the structure of the matrix $|A| + |A|^T$ and this ordering is used to build the tree where *parallelism* can be exploited at two levels. *At the tree level*, when operations on different branches of the tree can proceed in parallel and, *at the node level*, when dense kernels are used. Since the tree is based on the symmetrized pattern of A, it is specifically designed for symmetrically structured matrices although any unsymmetric matrix could be factorized by including explicit zeros so that the pattern is symmetric.

If the matrix is symmetric positive definite then numerical pivoting is not required and the code provides an alternative to the supernodal code described in Section 4.1. The two approaches have their own strengths and weaknesses and one might perform better than the other depending on the structure of the elimination tree. Our experience is that the multifrontal approach is both easier to implement and more suited for a GPU implementation but requires more memory. For background information on these approaches to the direct solution of sparse equations, we refer the reader to the book [19].

However, for matrices that are not positive definite, we require numerical pivoting to ensure numerical stability. This is done during the numerical factorization and requires dense kernels to be coded to accommodate this. The main algorithmic work, in Task 3.2, has been on designing kernels that maintain high levels of parallelism while maintaining numerical stability. This work is described in NLAFET Working Note WN-21 *A new sparse* LDL^T *solver using A Posteriori Threshold Pivoting* [18] and uses 2-D blocking, speculative execution, and a fail-in-place approach to enable most of the computation to proceed in parallel.

Details for the installation of the codes in SyLVER are presented in the Deliverable D3.3.

4.3 `ParSHUM`

The `ParSHUM` repository contains codes for the solution of matrices that are highly unsymmetric. We define a highly unsymmetric matrix as a matrix whose structure is not well approximated by the structure of $|A| + |A|^T$. Various authors have defined a measure of the asymmetry of a matrix and here we use that defined in [21] which is the proportion of off-diagonal entries for which there is a corresponding entry in the transpose, viz.

$$si(A) = \frac{number_{i \neq j}\{a_{ij} * a_{ji} \neq 0\}}{nz\{A\}},$$

where si is called the symmetry index and $nz\{A\}$ is the number of off-diagonal entries in the matrix A. A symmetric matrix will thus have a symmetry index of 1.0. Matrices with symmetry indices of less than 0.9 can be considered highly unsymmetric and these are the main target of the `ParSHUM` package developed within Task 3.3.

One of the main algorithmic challenges in this work was to design parallel algorithms for implementing a right-looking factorization, which is described in NLAFET Working Note WN-22 *Design and implementation of a parallel Markowitz threshold algorithm* [14]. The code in the repository based on this algorithm outperforms all other codes for this class of matrices on multicore CPUs.

In order to solve such systems on distributed memory machines, we first perform a block partitioning of the matrix to a form shown in Figure 3. The blocks on the

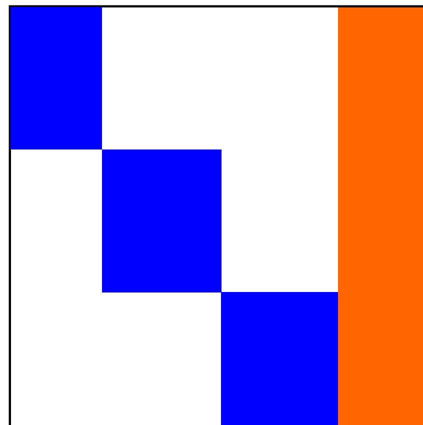
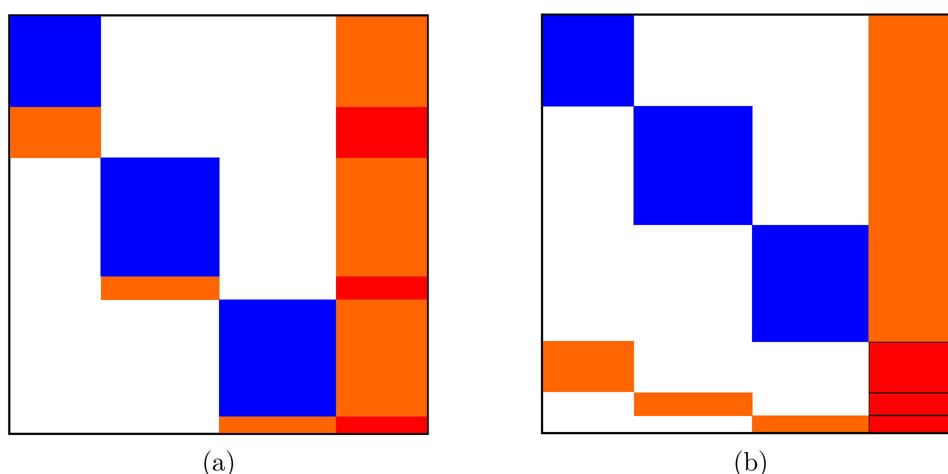


Figure 3: A singly bordered block diagonal form.

diagonal are distributed and the core ParSHUM multicore code is used to factorize these rectangular blocks as we show in Figure 4a. When all blocks are factorized, the remaining Schur complement shown red in Figure 4b is factorized using a parallel dense factorization.



(a) (b)

Figure 4: Factorization of SBBD form.

The performance of this approach depends on having a small border and in our extensive numerical experiments on this we have shown that for highly symmetric matrices particularly from applications where the matrix is very sparse this border is indeed small.

It increases when we increase the number of partitions but still tends to remain very small relative to the size of the original matrix.

We use Zoltan [10] to effect the partitioning and `ParSHUM` to factorize the blocks on the diagonal. Thus we combine distributed memory parallelism (using MPI) with shared memory parallelism (using OpenMP).

We have used this code when solving systems from Power System applications and have described this work in Deliverable D5.3.

Details for the installation of the codes in ParSHUM are presented in the Deliverable D3.6.

4.4 BC

The codes in the `BC` repository implement a hybrid approach to solving sparse equations. They can solve sparse systems with unsymmetric or even rectangular matrices using both distributed memory and shared memory parallelism. The codes are based on the `ABCD` code[1] [20] and much of this work in Task 3.4 has been done in collaboration with our colleagues in Toulouse, France. We described this work in Deliverable D3.7 submitted in February 2019 (M40).

In practice, the code detects if the matrix has more rows than columns. If it does, the matrix is partitioned into block columns and distributed. Each block of columns forms a subsystem that is then used to create an augmented system which is solved by using `SyLVER` (see Section 4.2). The local solutions are concatenated into a vector that is used in an CG-like iterative method. Otherwise, when the number of rows is not greater than the number of columns, we use a block row partitioning so that the local solution of the augmented systems is now summed rather than concatenated. We give details of these two algorithms in Deliverable D3.7.

The performance of the code relies on the parallel implementation of CG and the efficiency of the `SyLVER` solver, but also on a good partitioning of the matrix. The convergence of `BC` is improved by using a numerically aware partitioner [29]. Experimental results have shown that, depending on the problem, this partitioner is able to reduce the number of iterations by a factor of up to 5 over a partitioner that does not respect numerical values. This numerically aware partitioning is the default in our package.

Details for the installation of the codes in BC are presented in the Deliverable D3.7.

5 Communication optimal algorithms for iterative methods

In Deliverables 4.3 and 4.4 we have introduced a preconditioned Krylov subspace solver which aims to reduce communication when solving large sparse linear systems of equations.

The NLAFET public repository for communication optimal algorithms for iterative methods is:

- `preAlps`: Preconditioned iterative methods and enlarged Krylov methods.

5.1 preAlps library

In Deliverable 4.5 we have described the `preAlps` library which integrates a highly parallel and efficient implementation of enlarged CG Krylov subspace methods and the multilevel LORASC preconditioner. Their performance has been assessed in Deliverables 4.4, 4.5, and 5.3.

[1] `http://abcd.enseeiht.fr/`

The library depends on a few external libraries that need to be installed and linked with `preAlps` before it is ready to use:

- BLAS and LAPACK [7]: BLAS is a standard library for performing basic vector and matrix operations. LAPACK is a standard software for numerical linear algebra. Although any library providing BLAS and LAPACK can be used, we recommend MKL [30].
- METIS[24] and ParMETIS [25]: sequential and parallel graph partitioning tools. METIS is required in order to use ECG, while ParMETIS is required in order to use LORASC. We recommend to install ParMETIS as it already contains all METIS routines. `preAlps` was tested with METIS 5.1.0 and ParMETIS 4.0.3. These partitioning tools can be downloaded from `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview` and `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.
- PARPACK [27]: a parallel library used to solve eigenvalue problems. PARPACK is required in order to use LORASC. A latest version can be downloaded from `http://www.caam.rice.edu/software/ARPACK/download.html` . At the moment, only PARPACK is supported in `preAlps`, but we plan to use other eigenvalue solvers.
- MUMPS [6]: a distributed parallel sparse direct solver. MUMPS is required in order to use LORASC. `preAlps` was tested with MUMPS 5.1.2. It can be downloaded from `http://mumps.enseeiht.fr/`
- PARDISO: a sequential and multithreaded sparse direct solver. This library is optional if MKL is already provided. If MKL is not provided, PARDISO from `http://pardiso-project.org/` should be installed.

The complete installation of `preAlps` is described in Deliverable 4.5 and in the README file of `preAlps` repository. The directory *example* of `preAlps` contains several standalone examples for testing ECG and LORASC. The *-h* option details the usage of these example codes.

For completeness we recall here the different routines available. Our implementation of ECG is based on Reverse Communication Interface [23] and written in C and MPI. Following this scheme we provide four routines:

- `preAlps_ECGInitialize(ECG_t* ecg, double* rhs, int* rci_request)`: initialize the underlying structure,
- `preAlps_ECGIterate(ECG_t* ecg, int* rci_request)`: used within a loop to perform the iterations of the method,
- `preAlps_ECGStoppingCriterion(ECG_t* ecg, int* stop)`: compute and check the convergence of the method,
- `preAlps_ECGFinalize(ECG_t* ecg, double* solution)`: retrieve the solution and free the memory.

LORASC preconditioner can be built separately and used in any sparse iterative solver. The implemented routines are briefly described below:

ALGORITHM 1: Tiled algorithm for solving $Ax = b$.

for $j = N, N-1, \dots, 1$ **do**
  $x_j \leftarrow A_{jj}^{-1} b_j$;
  for $i = 1, 2, \dots, j-1$ **do**
    $b_i \leftarrow b_i - A_{ij} x_j$
  end
end

- `preAlps_LorascAlloc (preAlps_Lorasc_t **lorasc)` : creates an object of the type `preAlps_Lorasc_t`. The resulting object can be used by the end-user to replace the default parameters such as the `deflation_tolerance`.
- `preAlps_LorascBuild (preAlps_Lorasc_t *lorasc, CPLM_Mat_CSR_t *A, CPLM_Mat_CSR_t *locAP, MPI_Comm comm)` : constructs LORASC preconditioner from an input matrix A and stores all the required internal workspace in the object `lorasc`. First, it partitions and permutes the matrix A into a block arrow structure, then distributes it to each processor. After this distribution, each processor stores in the output matrix $locAP$ its block from a 1-D block row distribution of the permuted matrix A. Finally it constructs the preconditioner itself.
- `preAlps_LorascApply (preAlps_Lorasc_t *lorasc, double *x, double *y)` : applies LORASC preconditioner on a vector x and returns the result in the vector y.
- `preAlps_LorascApplyMat (preAlps_Lorasc_t *lorasc, CPLM_Mat_Dense_t *X, CPLM_Mat_Dense_t *Y)` : applies LORASC preconditioner on a dense matrix X, and returns the result in a dense matrix Y. This routine does the same computation as `preAlps_LorascApply` routine with the difference that it applies the preconditioner on a dense matrix.
- `preAlps_LorascDestroy (preAlps_Lorasc_t **lorasc)` : frees the internal memory allocated by LORASC preconditioner and destroys `lorasc` object.

6 Cross-cutting tools

The NLAFET public repositories for cross-cutting topics are:

- `PCP-runtime`: Parallelizing the critical path.
- `ABFT-factor`: Tiled factorizations with ABFT.

6.1 The experimental PCP runtime library

Many of the task-based parallel algorithms explored in the NLAFET project have task graphs which depend on one or more parameters, e.g., tile sizes and algorithmic block sizes. Consider for example a tiled algorithm (Algorithm 1) for solving a triangular linear system of equations

$$\begin{bmatrix} A_{11} & \cdots & A_{1N} \\ & \ddots & \vdots \\ & & A_{NN} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix}.$$

By creating a task for each tiled solve $A_{jj}^{-1}b_j$ and each tiled update $b_i - A_{ij}x_j$, one obtains a task graph G_N with $N(N+1)/2$ tasks. The total amount of work/arithmetic is independent of N, which implies that N determines the task granularity. Large N results in many small tasks and a high degree of concurrency. Small N results in a few large tasks and barely any concurrency.

Task-based implementations of parallel algorithms commonly work under the assumption that each task runs sequentially on one core, which is here referred to as *regular scheduling*. Suppose regular scheduling is used when solving a triangular system of size $n \times n$. The performance can be optimized for a particular machine and problem size by tuning N. The execution time, as a function of N, tends to decrease before eventually increasing. The initial decrease stems from an increasing degree of concurrency, and the eventual increase is the result of a decrease in task efficiency and increase in scheduling overhead.

If the degree of concurrency is too low, then the execution time will be determined by the execution time of the critical path. In such situations, increasing the number of cores will not help. However, some improvement might be possible by running some or all tasks in parallel instead of sequentially. This is clearly the case for some fixed values of N, but are improvements possible also for optimal[2] values of N?

Within the NLAFET project, the idea of parallelizing (only) the tasks on the critical path, here referred to as *parallel critical path scheduling*, has been extensively investigated and reported in the NLAFET deliverable reports D6.1 *Prototypes for runtime systems exhibiting novel types of scheduling* and D6.3 *Evaluation of software prototypes*. To use parallel critical path scheduling, the user selects a path in the graph, ideally but not necessarily a path which is critical under regular scheduling. Then the user supplies parallel implementations of all types of tasks on the selected path. Finally, the user sets the number of cores q to use for the parallel execution of critical tasks.

A runtime system supporting parallel critical path scheduling behaves as follows. Before execution begins, q cores are set aside for the critical tasks and the remaining $p - q$ cores are reserved for the non-critical tasks. (Note that regular scheduling essentially corresponds to $q = 0$.)

Since parallel critical path scheduling generalizes regular scheduling, one would expect that by optimizing both N and q one can achieve comparable or superior performance compared to regular scheduling. In order to properly test the idea, an experimental task-based runtime system (called `PCP-runtime`) was developed. The objective was to enable a fair comparison of parallel critical path scheduling with regular scheduling. At the time when D6.1 was delivered, no runtime systems were optimized for parallel critical path scheduling. The source code for the experimental runtime system, including a couple of examples of its use, is available for download in the `PCP-runtime` repository[3].

As reported in the M42 deliverable D6.3, parallel critical path scheduling can now be implemented (albeit not with minimal overhead) on top of StarPU. No modification of the StarPU source code is necessary. New results presented in D6.3 suggest that the performance impact of parallel critical path scheduling is modest. The main benefit appears to be the possibility to avoid using very small tiles — tiles so small that the overhead in the runtime system (StarPU in this case) ruins the performance.

[2]The optimal value of N using regular scheduling is likely to be different from the optimal value using some other form of scheduling.

[3]`https://github.com/NLAFET/PCP-runtime`

6.2 One-sided factorizations with algorithm-based fault tolerance

The `ABFT-factor` repository contains prototype software for tiled one-sided factorisations with algorithm-based fault tolerance. Fault tolerance methods aim to provide a mechanism for detecting and correcting errors, such as memory bit-flips, data corruption, or intermittent hardware faults that can be expected to occur during large-scale linear algebra computations [28]. ABFT implementations seek to exploit algorithmic structure in order to minimise the computational cost of detecting and correcting such errors within parallel systems. The increasing parallelism of modern HPC architectures inevitably leads to an increase in the frequency of such errors [11], and therefore places increased importance on their efficient detection and correction. The ABFT prototype software contained within the repository uses the PaRSEC framework [12], which provides a task-based environment on which to run shared- or distributed-memory computations. The ABFT implementation is demonstrated using Cholesky factorisation, with a dual-checksum approach, allowing the location of faults to be identified and corrected efficiently. Details on the implementation used, and operation of the software, can be found in the associated report for deliverable D6.7. Performance tests demonstrate a low overhead in computational cost and memory utilisation for the inclusion of ABFT, strengthening the case for its use to allow high-performance error mitigation within dense linear algebra routines.

7 Summary and future contributions

We have presented the current 13 public repositories which together make up the public release of the NLAFET library. In addition to source codes, each repository include documentation that describes installation, usage, and testing of the main software components. In most cases the documentation is provided in Users' Guides generated via the Doxygen system.

In comparisons with existing state-of-the-art library software, NLAFET shows outstanding results in terms of performance, scalability and accuracy. Such comparisons and results concerning all NLAFET repositories are presented in several NLAFET deliverable reports and NLAFET Working Notes, all available via the NLAFET web-site[4]. For example, the M42 NLAFET deliverable report D5.3 *Validation and evaluation* describes efforts on validation and integration of NLAFET library components in some challenging applications.

Altogether, the software components of the NLAFET library release for solving fundamental and important numerical linear algebra problems provide novel task-based algorithms using various programming environments (MPI, OpenMP, ParSEC and StarPU) and contribute to the development of parallel numerical linear algebra for future extreme scale systems. For the future, we plan to update and integrate new progress concerning software components in the NLAFET library.

Acknowledgements

We thank the High Performance Computing Center North (HPC2N) at Umeå University, which is part of the Swedish National Infrastructure for Computing (SNIC), for providing computational resources and valuable support.

[4] `http://www.nlafet.eu/public-deliverables/` and `http://www.nlafet.eu/working-notes/`

References

[1] StarPU — A Unified Runtime System for Heterogeneous Multicore Architectures. http://starpu.gforge.inria.fr/.

[2] B. Adlerborn, B. Kågström, and D. Kressner. A Parallel QZ Algorithm for Distributed Memory HPC Systems. *SIAM J. Sci. Comput.*, 36(5):C480–C503, 2014.

[3] Björn Adlerborn, Bo Kågström, and Daniel Kressner. PDHGEQZ User Guide. *NLAFET Working Note* WN-2, May, 2016. Also as Report UMINF 15.12, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.

[4] Björn Adlerborn, Lars Karlsson, and Bo Kågström. Distributed one-stage hessenberg-triangular reduction with wavefront scheduling. *SIAM J. Scientific Computing*, 40(2), 2018.

[5] Björn Adlerborn, Lars Karlsson, and Bo Kågström. Distributed One-Stage Hessenberg-Triangular Reduction with Wavefront Scheduling. *NLAFET Working Note* WN-1, May, 2016. Also as Report UMINF 16.10, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.

[6] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23(1):15–41, 2001.

[7] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*, volume 9. SIAM, 1999.

[8] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and D Sorensen. *LAPACK Users' guide*, volume 9. SIAM, 1999.

[9] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users' guide*, volume 4. SIAM, 1997.

[10] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Courtenay Vaughan, Umit Catalyurek, Doruk Bozdag, William Mitchell, and James Teresco. *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4748W http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.

[11] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience*, 26(17):2772–2791, 2014.

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[13] Sébastien Cayrols, Iain Duff, and Florent Lopez. Parallelization of the solve phase in a task-basedCholesky solver using a sequential task flow model. *NLAFET Working Note* WN-20, October, 2018. Also as Technical Report RAL-TR-2018-008, Science & Technology Facilities Council, UK.

[14] Timothy Davis, Iain S. Duff, and Stojce Nakov. Design and implementation of a parallel Markowitz threshold algorithm. *NLAFET Working Note* WN-22, February, 2019. Also as Technical Report RAL-TR-2019-003, Science & Technology Facilities Council, UK.

[15] Jack J Dongarra, Jermey Du Cruz, Sven Hammarling, and Iain S Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):18–28, 1990.

[16] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J Hanson. Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):18–32, 1988.

[17] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):1–17, 1988.

[18] Iain Duff, Jonathan Hogg, and Florent Lopez. A new sparse symmetric indefinite solver using A Posteriori Threshold Pivoting. *NLAFET Working Note* WN-21, December, 2018. Also as Technical Report RAL-TR-2018-012, Science & Technology Facilities Council, UK.

[19] Iain S. Duff, Albert M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices. Second Edition.* Oxford University Press, Oxford, England, 2017.

[20] Iain S. Duff, Ronan Guivarch, Daniel Ruiz, and Mohamed Zenadi. The augmented block Cimmino distributed method. *SIAM J. Scientific Computing*, 37(3):A1248–A1269, 2015.

[21] A. M. Erisman, R. G. Grimes, J. G. Lewis, W. G. Poole Jr., and H. D. Simon. Evaluation of orderings for unsymmetric sparse matrices. *SIAM Journal on Scientific and Statistical Computing*, 7:600–624, 1987.

[22] R. Granat, B. Kågström, D. Kressner, and M. Shao. ALGORITHM 953: Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation. *ACM Trans. Math. Software*, 41(4):Article 29:1–23, 2015.

[23] Dongarra J., Eijkhout V., and Kalhan A. Reverse communication interface for linear algebra templates for iterative methods. *UT, CS-95-291, May*, 1995.

[24] George Karypis and Vipin Kumar. METIS –unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[25] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[26] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. 1977.

[27] R. B. Lehoucq, D. C. Sorensen, and C. Yang. Arpack User's Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods, 1997.

[28] Bianca Schroeder and Garth A. Gibson. A Large-Scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–351, 2010.

[29] F. S. Torun, M. Manguoglu, and C. Aykanat. A novel partitioning method for accelerating the block Cimmino algorithm. *SIAM J. Scientific Computing*, 40(6):C827–C850, 2018.

[30] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.