



<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper presented at *13th International Conference on Parallel Computing and Applied Mathematics, PPAM 2019, Bialystok, Poland, September 8-11, 2019*.

Citation for the original published paper:

Mylykoski, M., Kjelgaard Mikkelsen, C C. (2020)

Introduction to StarNEig: A Task-based Library for Solving Nonsymmetric Eigenvalue Problems

In: Roman Wyrzykowski and Boleslaw Szymanski (ed.), *Parallel Processing and Applied Mathematics: Revised Selected Papers, Part I* (pp. 70-81). Springer

Lecture Notes in Computer Science

https://doi.org/10.1007/978-3-030-43229-4_7

N.B. When citing this work, cite the original published paper.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-168419>

Introduction to StarNEig — A Task-based Library for Solving Nonsymmetric Eigenvalue Problems

Mirko Myllykoski^[0000–0002–3689–0899] and
Carl Christian Kjelgaard Mikkelsen^[0000–0002–9158–1941]

Department of Computing Science and HPC2N,
Umeå University, SE-901 87 Umeå, Sweden
{mirkom,spock}@cs.umu.se

Abstract. In this paper, we present the StarNEig library for solving dense nonsymmetric (generalized) eigenvalue problems. The library is built on top of the StarPU runtime system and targets both shared and distributed memory machines. Some components of the library support GPUs. The library is currently in an early beta state and only real arithmetic is supported. Support for complex data types is planned for a future release. This paper is aimed at potential users of the library. We describe the design choices and capabilities of the library, and contrast them to existing software such as ScaLAPACK. StarNEig implements a ScaLAPACK compatibility layer that should make it easy for new users to transition to StarNEig. We demonstrate the performance of the library with a small set of computational experiments.

Keywords: Eigenvalue problem · Task-based · Library.

1 Introduction

In this paper, we present the StarNEig library [5] for solving dense nonsymmetric (generalized) eigenvalue problems. StarNEig differs from the existing libraries such as LAPACK [1] and ScaLAPACK [3] in that it relies on a modern task-based approach (see, e.g., [21] and references therein). More specifically, StarNEig is built on top of the StarPU runtime system [6]. This allows StarNEig to target both shared memory and distributed memory machines. Furthermore,

Please cite this article as: M. Myllykoski, C.C. Kjelgaard Mikkelsen, Introduction to StarNEig — A Task-based Library for Solving Nonsymmetric Eigenvalue Problems, In *Parallel Processing and Applied Mathematics, 13th International Conference, PPAM 2019, Bialystok, Poland, September 8–11, 2019, Revised Selected Papers, Part I*, Lecture Notes in Computer Science, Vol. 12043, Wyrzykowski R., Boleslaw S. (eds), Springer Nature Switzerland AG, pp. PAGE–PAGE, 2020. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-43229-4_7.

some components of StarNEig support GPUs. The library is currently in an early beta state and under continuous development.

This paper targets potential users of the library. We hope that readers, who are already familiar with ScaLAPACK, will be able to decide if StarNEig is suitable for them. In particular, we want to communicate what type of changes are necessary to make their software work with StarNEig. We will explain, through an example, why the task-based approach can potentially lead to superior performance when compared to older, well-established, approaches. We also present a small sample of computational results which demonstrate the expected performance of StarNEig. We refer the reader to [20] for more comprehensive performance and accuracy evaluations.

The rest of this paper is organized as follows: Section 2 provides a brief introduction to the solution of dense nonsymmetric eigenvalue problems. Section 3 explains the task-based approach and Section 4 introduces the reader to some of the inner workings of StarNEig. Section 5 presents a small set of computational results and, finally, Section 6 concludes the paper.

2 Solution of Dense Nonsymmetric Eigenvalue Problems

Given a matrix $A \in \mathbb{R}^{n \times n}$, the standard eigenvalue problem consists of computing eigenvalues $\lambda_i \in \mathbb{C}$ and matching eigenvectors $x_i \in \mathbb{C}^n$, $x_i \neq 0$, such that

$$Ax_i = \lambda_i x_i. \quad (1)$$

Similarly, given matrices $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times n}$ the generalized eigenvalue problem for the matrix pair (A, B) consists of computing generalized eigenvalues $\lambda_i \in \mathbb{C}$ and matching generalized eigenvectors $x_i \in \mathbb{C}^n$, $x_i \neq 0$, such that

$$Ax_i = \lambda_i Bx_i. \quad (2)$$

If the matrices A and B are sparse, then the well-known SLEPc library [4] is one of the better tools for solving the eigenvalue problems (1) and (2). Similarly, if the matrices A and B are symmetric, then algorithms and software that take advantage of the symmetry are preferred (see, e.g., [2, 9, 14, 17, 18]). Otherwise, if the matrices are *dense* and *nonsymmetric*, then route of acquiring the (generalized) eigenvalues and the (generalized) eigenvectors usually includes the following three steps:

Hessenberg(-triangular) reduction: The matrix A or the matrix pair (A, B) is reduced to Hessenberg or Hessenberg-triangular form by an orthogonal similarity transformation

$$A = Q_1 H Q_1^T \quad \text{or} \quad (A, B) = Q_1 (H, R) Z_1^T, \quad (3)$$

where H is upper Hessenberg, R is an upper triangular, and Q_1 and Z_1 are orthogonal matrices.

Schur reduction: The Hessenberg matrix H or the Hessenberg-triangular matrix pair (H, R) is reduced to real Schur or generalized real Schur form by an orthogonal similarity transformation

$$H = Q_2 S Q_2^T \quad \text{or} \quad (H, R) = Q_2 (S, T) Z_2^T, \quad (4)$$

where S is upper quasi-triangular with 1×1 and 2×2 blocks on the diagonal, T is a upper triangular, and Q_2 and Z_2 are orthogonal matrices. The eigenvalues or generalized eigenvalues can be determined from the diagonal blocks of S or (S, T) . In particular, the 2×2 blocks on the diagonal of S correspond to the complex conjugate pairs of (generalized) eigenvalues.

Eigenvectors: Finally, we solve for vectors $y_i \in \mathbb{C}^n$ from

$$(S - \lambda_i I) y_i = 0 \quad \text{or} \quad (S - \lambda_i T) y_i = 0 \quad (5)$$

and backtransform to the original basis by

$$x_i = Q_1 Q_2 y_i \quad \text{or} \quad x_i = Z_1 Z_2 y_i. \quad (6)$$

Additionally, a fourth step can be performed to acquire an invariant subspace of A or (A, B) that is associated with a given subset of eigenvalues or a given subset of generalized eigenvalues:

Eigenvalue reordering: The real Schur form S or the generalized real Schur form (S, T) is reordered, such that a selected set of eigenvalues or generalized eigenvalues appears in the leading diagonal blocks of an updated real Schur form \hat{S} or an updated generalized real Schur form (\hat{S}, \hat{T}) , by an orthogonal similarity transformation

$$S = Q_3 \hat{S} Q_3^T \quad \text{or} \quad (S, T) = Q_3 (\hat{S}, \hat{T}) Z_3^T, \quad (7)$$

where Q_3 and Z_3 are orthogonal matrices.

See [11] for a detailed explanation of the underlying mathematical theory.

3 A Case for the Task-Based Approach

A task-based algorithm functions by cutting the computational work into self-contained tasks that all have a well defined set of inputs and outputs. In particular, StarNEig divides the matrices into (square) tiles and each task takes a set of tiles as its input and produces/modifies a set of tiles as its output. The main difference between tasks and regular function/subroutine calls is that a task-based algorithm does not call the associated computation kernels directly. Instead, the tasks are inserted into a runtime system that derives the task dependences and schedules the tasks to computational resources in a sequentially consistent order. The main benefit of this is that as long as the cutting is carefully done, the underlying parallelism is exposed automatically as the runtime system traverses the resulting task graph.

3.1 Novelty in StarNEig

The first main source of novelty in StarNEig comes from the way in which the computational work is cut into tasks and how the task are inserted into the runtime system. The Hessenberg reduction, Schur reduction and eigenvalue reordering steps are based on two-sided transformation algorithms. These algorithms lead to task graphs that are significantly more complicated than the task graphs arising from one-sided transformation algorithm such as the LU factorization. Designing such a task graph, that also leads to high performance, is a non-trivial task as the left and right hand side updates can easily interfere with each other.

Furthermore, the Schur reduction and eigenvalue reordering steps apply a series of overlapping local transformations to the matrices. Due to this overlap, there cannot exist a clear one-to-one mapping between the tasks and the (output) tiles since the local transformations must at some point cross between two or more tiles. Instead, most tasks end up modifying several tiles and this can introduce spurious task dependences that limit the concurrency. By a spurious task dependency, we mean a dependency that is created when two (or more) tasks modify non-overlapping parts of the same tile (i.e., the tasks are independent from each other) but the runtime system interprets this as a task dependency.

The second main source of novelty in StarNEig is related to the eigenvector computation step. Here, the task dependences are comparatively simple but the computations must be protected against floating-point overflow. This is a nontrivial issue to address in a parallel setting as explained in [15, 16]. The implementation in StarNEig is both robust and tiled. The former means that the computed eigenvectors are always in the representable range of double precision floating-point numbers and the latter leads to level 3 BLAS performance. The same combination of robustness and performance does not exist neither in LAPACK nor ScaLAPACK since the corresponding routines in LAPACK are scalar codes and the corresponding ScaLAPACK routines are not robust.

3.2 Bulge Chasing and Eigenvalue Reordering

We will now use the Schur reduction and eigenvalue reordering steps to illustrate some benefits of the task-based approach. The modern approach for obtaining a Schur form S of A is to apply the multishift QR algorithm with Aggressive Early Deflation (AED) to the upper Hessenberg form H (see [7, 8, 13] and references therein). The algorithm is a sequence of steps of two types: AED and bulge chasing. The bulge chasing step creates a set of 3×3 bulges to the upper left corner of the matrix and the bulges are chased down the diagonal to complete one pipelined QR iteration. This is accomplished by applying sequences of overlapping 3×3 Householder reflectors to H . Similarly, the eigenvalue reordering step is based on applying sequences of overlapping Givens rotations and 3×3 Householder reflectors to S .

If the local transformations are applied one by one, then memory is accessed as shown in Fig. 1a. This is grossly inefficient for two reasons: i) the transformations are so localized that parallelizing them would not produce any significant

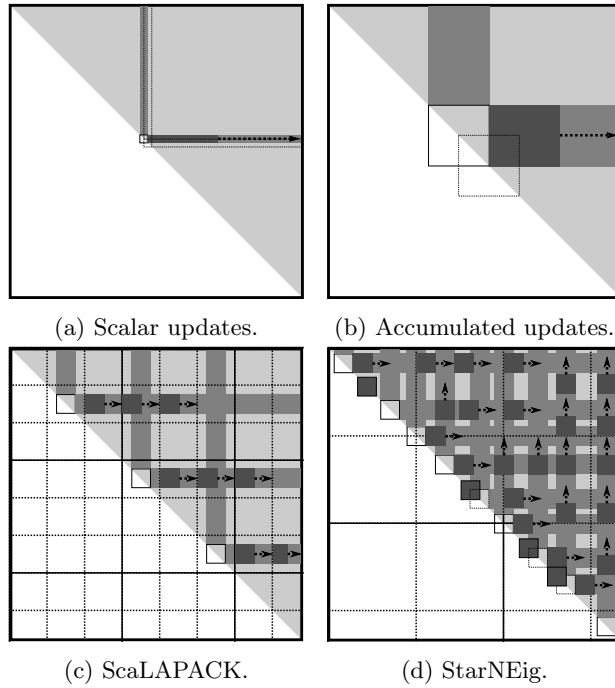


Fig. 1: Hypothetical snapshots taken during the computations. The currently active regions are highlighted with darker shade and the propagation directions of the off-diagonal updates are marked with arrows. In (a), the overlap between two overlapping transformations is highlighted with dashed lines. In (b), the overlap between two diagonal windows is highlighted with dashed lines. In (c) and (d), the dashed lines illustrate how the matrix is divided into distributed blocks and the solid lines illustrate the MPI process mesh.

speedup and ii) the matrix elements are touched only once thus leading to very low arithmetic intensity. The modern approach groups together a set of local transformation and initially applies them to a relatively small diagonal window as shown in Fig. 1b. The local transformations are accumulated into an accumulator matrix and later propagated as level 3 BLAS operations acting on the off-diagonal sections of the matrix. This leads to much higher arithmetic intensity and enables proper parallel implementations as *multiple* diagonal windows can be processed simultaneously.

In particular, the Schur reduction and eigenvalue reordering steps are implemented in ScaLAPACK as PDHSEQR [13] and PDTRSEN [12] subroutines, respectively. Following the ScaLAPACK convention, the matrices are distributed in two-dimensional block cyclic fashion [10]. The resulting memory access pattern is illustrated in Fig. 1c for a 3×3 MPI process mesh. In this example, three diagonal windows can be processed simultaneously. The related level 3

BLAS updates require careful coordination since the left and right hand side updates must be performed in a sequentially consistent order. In practice, this means (global or broadcast) synchronization after each set of updates have been applied.

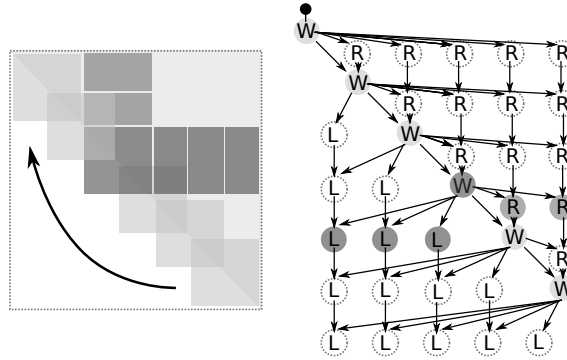


Fig. 2: A hypothetical task graph arising from a situation where a Schur form is reordered with a single chain of overlapping diagonal windows. We have simplified the graph by omitting dependences between the right (R) and left (L) update tasks as these dependences are enforced through the window tasks (W).

In a task-based approach, this can be done using the following task types:

- Window task** applies a set of local transformations inside the diagonal window. Takes the intersecting tiles as input, and produces updated tiles and an accumulator matrix as output.
- Right update task** applies accumulated right-hand side updates using level 3 BLAS operations. Takes the intersecting tiles and an accumulator matrix as input, and produces updated tiles as output.
- Left update task** applies accumulated left-hand side updates using level 3 BLAS operations. Takes the intersecting tiles and an accumulator matrix as input, and produces updated tiles as output.

The tasks are inserted into the runtime system in a sequentially consistent order and each chain of overlapping diagonal windows leads to a task graph like the one shown in Fig. 2. Note that real live task graphs are significantly more complex than shown here, but also enclose more opportunities for parallelism. It is also critical to realize that the runtime system *guarantees* that the tasks are executed in a sequentially consistent order. In particular, there is no need for synchronization and different stages are allowed to overlap and merge together as illustrated in Fig. 1d. This can lead to a much higher concurrency since idle time can be reduced by delaying low priority tasks until computational resources start becoming idle. The AED step in the QR algorithm can also be overlapped with the bulge chasing steps and this improves the concurrency significantly. Other

Table 1: Current status of the StarNEig library.

Step	Shared memory	Distr. memory	GPUs (CUDA)
Hessenberg	Complete	ScaLAPACK	Single GPU
Schur	Complete	Complete	Experimental
Reordering	Complete	Complete	Experimental
Eigenvectors	Complete	In progress	—
Hessenberg-triangular	LAPACK	ScaLAPACK	—
Generalized Schur	Complete	Complete	Experimental
Generalized reordering	Complete	Complete	Experimental
Generalized eigenvectors	Complete	In progress	—

benefits of the task-based approach include, for example, better load balancing, task priorities, accelerators support and implicit MPI communication. See [19, 20] for further information.

4 StarNEig Library

StarNEig is a C-library that runs on top of the StarPU task-based runtime system. StarPU handles low-level operations such as heterogeneous scheduling; data transfers and replication between various memory spaces; and MPI communication between compute nodes. In particular, StarPU is responsible for managing the various computational resources such as CPU cores and GPUs. The support for GPUs and distributed memory were the main reasons why StarPU was chosen as the runtime system.

StarPU manages a set of worker threads; usually one thread per computational resource. In addition, one thread is responsible for inserting the tasks into StarPU and tracking the state of the machine. If necessary, one additional thread is allocated for MPI communication. For these reasons, StarNEig must be used in a *one process per node* (1ppn) configuration, i.e., several CPU cores should be allocated for each MPI process (a node can be a full node, a NUMA island or some other reasonably large collection of CPU cores).

The current status of StarNEig is summarized in Table 1. The library is currently in an early beta state. The *Experimental* status indicates that the software component has not been tested as extensively as those software components that are considered *Complete*. In particular, the GPU functionality requires some additional involvement from the user (performance model calibration). At the time of writing this paper, only real arithmetic is supported and certain interface functions are implemented as LAPACK and ScaLAPACK wrapper functions. However, we emphasize that StarNEig supports real valued matrices that have complex eigenvalues and eigenvectors. Additional distributed memory functionality and support for complex data types are planned for a future release.

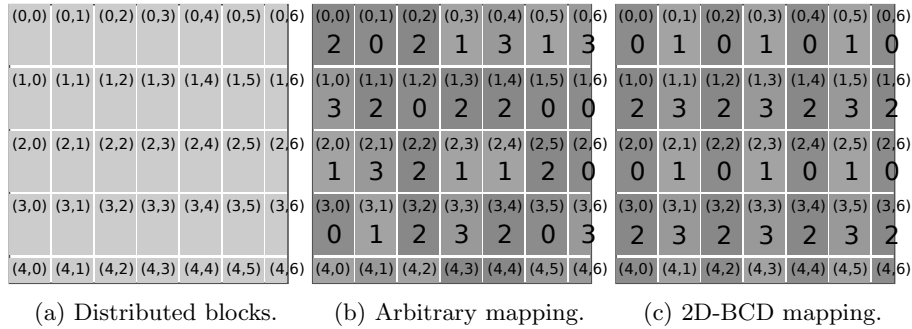


Fig. 3: Examples of various data distributions supported by StarNEig, including two-dimensional block cyclic distribution (2D-BCD).

4.1 Distributed Memory

StarNEig distributes the matrices in rectangular blocks of a uniform size (excluding the last block row and column) as illustrated in Fig. 3a. The data distribution, i.e., the mapping from the distributed blocks to the MPI process rank space, can be arbitrary as illustrated in Fig. 3b. A user has three options:

1. Use the default data distribution. This is recommended for most users and leads to reasonable performance in most situations.
2. Use a two-dimensional block cyclic distribution (see Fig. 3c). In this case, the user may select the MPI process mesh dimensions and the rank ordering.
3. Define a data distribution function $d : \mathbb{Z}^+ \times \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ that maps the block row and column indices to the MPI rank space. For example, in Fig. 3b, the rank 0 owns the blocks (0,1), (1,2), (1,5), (1,6), (2,6), (3,0) and (3,5).

The library implements distribution agnostic copy, scatter and gather operations.

Users who are familiar with ScaLAPACK are likely accustomed to using relatively small distributed block sizes (between 64–256). In contrast, StarNEig functions optimally only if the distributed blocks are relatively large (at least 1000 but preferably must larger). This is due to the fact that StarNEig further divides the distributed blocks into tiles and a tiny tile size leads to excessive task scheduling overhead because the tile size is closely connected to the task granularity. Furthermore, as mentioned in the preceding section, StarNEig should be used in 1ppn configuration as opposed to a *one process per core* (1ppc) configuration which is common with ScaLAPACK.

4.2 ScaLAPACK Compatibility

StarNEig is fully compatible with ScaLAPACK and provides a ScaLAPACK compatibility layer that encapsulates BLACS contexts and descriptors [10] inside transparent objects, and implements a set of bidirectional conversion functions. The conversions are performed in-place and do not modify any of the

underlying data structures. Thus, users can mix StarNEig interface functions with ScaLAPACK subroutines without intermediate conversions. The use of the ScaLAPACK compatibility layer requires the use of either the default data distribution or the two-dimensional block cyclic data distribution.

5 Performance Evaluation

Table 2: A run time comparison between ScaLAPACK and StarNEig.

n	CPU cores		Schur reduction (<i>secs</i>)		Eigenvalue reordering (<i>secs</i>)	
	ScaLAPACK	StarNEig	PDHSEQR	StarNEig	PDTRSEN	StarNEig
10 000	36	28	38	18	12	3
20 000	36	28	158	85	72	25
40 000	36	28	708	431	512	180
60 000	121	112	992	563	669	168
80 000	121	112	1667	904	1709	391
100 000	121	112	3319	1168	3285	737
120 000	256	252	3268	1111	2902	581

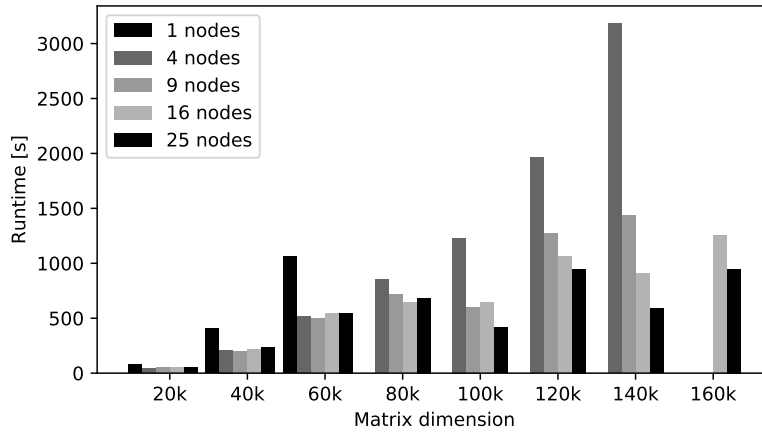


Fig. 4: Distributed memory scalability of StarNEig when computing a Schur form. Each node contains 28 CPU cores.

Computational experiments were performed on the Kebnekaise system, located at the High Performance Computing Center North (HPC2N), Umeå University. Each regular compute node contains 28 Intel Xeon E5-2690v4 cores (2

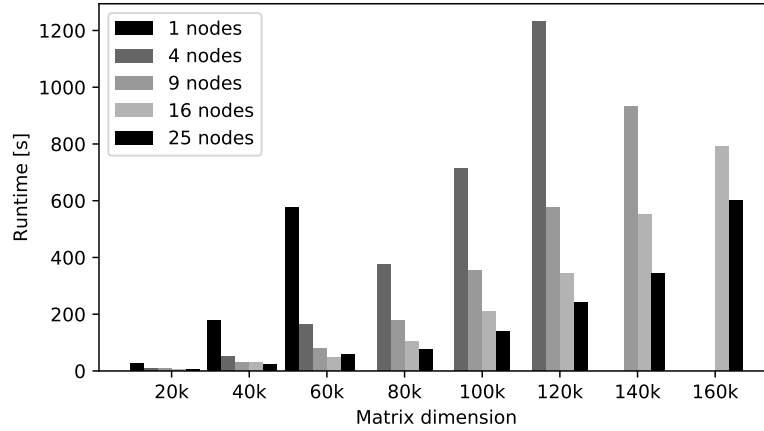


Fig. 5: Distributed memory scalability of StarNEig when reordering a Schur form. Each node contains 28 CPU cores.

NUMA islands) and 128 GB memory. The nodes are connected with FDR Infiniband. Each GPU compute node contains 28 Intel Xeon Gold 6132 cores (2 NUMA islands), 192 GB memory and two NVidia Tesla V100 GPUs.

The software was compiled with GCC 7.3.0 and linked to OpenMPI 3.1.3, OpenBLAS 0.3.2, ScaLAPACK 2.0.2, CUDA 9.2.88, and StarPU 1.2.8. All experiments were performed using a square MPI process grid. We always map each StarNEig process to a full node (28 cores) and each ScaLAPACK process to a single CPU core. The number of CPU cores in each ScaLAPACK experiment is always equal or larger than the number of CPU cores in the corresponding StarNEig experiment. The upper Hessenberg matrices for the Schur reduction experiments were computed from random matrices (entries uniformly distributed over the interval $[-1, 1]$).

Table 2 shows a comparison between ScaLAPACK and StarNEig¹. We note that StarNEig is between 1.6 and 2.9 times faster than PDHSEQR and between 2.8 and 5.0 times faster than PDTRSEN. Figures 4 and 5 give some idea of how well the library is expected to scale in distributed memory. We note that StarNEig scales reasonably when computing the Schur form and almost linearly when reordering the Schur form. The iterative nature of the QR algorithm makes the Schur reduction results less predictable as different matrices require different number of bulge chasing steps. Figure 6 demonstrates that StarNEig can indeed take advantage of the available GPUs. See [20] for more comprehensive comparisons.

¹ StarNEig was compared against an updated version of PDHSEQR; see [13].

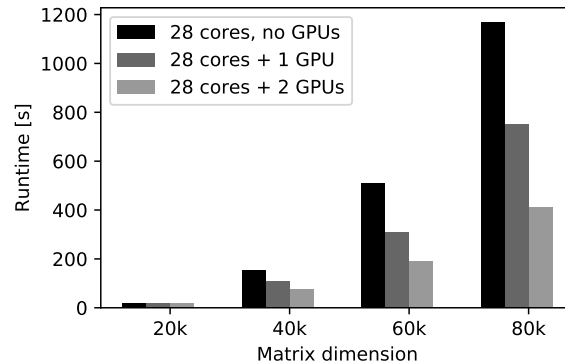


Fig. 6: GPU performance of StarNEig when reordering a Schur form. Each socket (14 cores) is connected to one NVidia Tesla V100 GPU.

6 Summary

This paper presented a new library called StarNEig. The paper is aimed for potential users of the library. Various design choices were explained and contrasted to existing software. In particular, users who are already familiar with ScaLAPACK should know following:

- StarNEig expect that the matrices are distributed in relatively large blocks compared to ScaLAPACK.
- StarNEig should be used in a *one process per node* (1ppn) configuration as opposed to a *one process per core* (1ppc) configuration which is common with ScaLAPACK.
- StarNEig implements a ScaLAPACK compatibility layer.

The presented distributed memory results indicate that the library is highly competitive with ScaLAPACK.

Future work with StarNEig includes the implementation and integration of the missing software components. Support for complex valued matrices is also planned. The GPU support, and the multi-GPU support in particular, are still under active development. The authors hope to start a discussion which would help guide and prioritize the future development of the library.

Acknowledgements

StarNEig has been developed by the authors, Angelika Schwarz (who has written the standard eigenvector solver), Lars Karlsson, and Bo Kågström. This work is part of a project (NLAFET) that has received funding from the European

Union’s Horizon 2020 research and innovation programme under grant agreement No 671633. This work was supported by the Swedish strategic research programme eSENCE. We thank the High Performance Computing Center North (HPC2N) at Umeå University for providing computational resources and valuable support during test and performance runs. Finally, the author thanks the anonymous reviewers for their valuable feedback.

References

1. LAPACK—Linear Algebra PACKage, <http://www.netlib.org/lapack>
2. PLASMA—Parallel Linear Algebra Software for Multicore Architectures, <http://icl.cs.utk.edu/plasma/software>
3. ScaLAPACK—Scalable Linear Algebra PACKage, <http://www.netlib.org/scalapack>
4. SLEPc — The Scalable Library for Eigenvalue Problem Computations, <http://slepc.upv.es>
5. StarNEig — A task-based library for solving nonsymmetric eigenvalue problems, <https://github.com/NLAFET/StarNEig>
6. StarPU — A unified runtime system for heterogeneous multicore architectures, <http://starpu.gforge.inria.fr>
7. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. I. Maintaining well-focused shifts and level 3 performance. *SIAM J. Matrix Anal. Appl.* **23**(4), 929–947 (2002). <https://doi.org/10.1137/S0895479801384573>
8. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. II. Aggressive early deflation. *SIAM J. Matrix Anal. Appl.* **23**(4), 948–973 (2002). <https://doi.org/10.1137/S0895479801384585>
9. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* **35**(1), 38–53 (2009). <https://doi.org/10.1016/j.parco.2008.10.002>
10. Dongarra, J., Whaley, R.C.: *A User’s Guide to the BLACS (1997)*, LAWN 94
11. Golub, G.H., Van Loan, C.F.: *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 4th edn. (2012)
12. Granat, R., Kågström, B., Kressner, D.: Parallel eigenvalue reordering in real Schur forms. *Concurrency and Computation: Practice and Experience* **21**(9), 1225–1250 (2009), <http://dx.doi.org/10.1002/cpe.1386>
13. Granat, R., Kågström, B., Kressner, D., Shao, M.: ALGORITHM 953: Parallel library software for the multishift QR algorithm with aggressive early deflation. *ACM Trans. Math. Software* **41**(4), 1–23 (2015). <https://doi.org/10.1145/2699471>
14. Imachi, H., Hoshi, T.: Hybrid numerical solvers for massively parallel eigenvalue computations and their benchmark with electronic structure calculations. *Journal of Information Processing* **24**(1), 164–172 (2016). <https://doi.org/10.2197/ipsjip.24.164>
15. Kjelgaard Mikkelsen, C.C., Myllykoski, M.: Parallel robust computation of generalized eigenvectors of matrix pencils in real Schur form. Accepted to PPAM 2019
16. Kjelgaard Mikkelsen, C.C., Schwarz, A.B., Karlsson, L.: Parallel robust solution of triangular linear systems. *Concurrency and Computation: Practice and Experience* **0**(0), 1–19 (2018). <https://doi.org/10.1002/cpe.5064>

17. Luszczek, P., Ltaief, H., Dongarra, J.: Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In: *Parallel & Distributed Processing Symposium (IPDPS)*, 2011 IEEE International. pp. 944–955. IEEE (2011). <https://doi.org/10.1109/IPDPS.2011.91>
18. Marek, A., Blum, V., Johanni, R., Havu, V., Lang, B., Auckenthaler, T., Heinecke, A., Bungartz, H.J., Lederer, H.: The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter* **26**(21), 201–213 (2014). <https://doi.org/10.1088/0953-8984/26/21/213201>
19. Myllykoski, M.: A task-based algorithm for reordering the eigenvalues of a matrix in real Schur form. In: *Parallel Processing and Applied Mathematics, PPAM 2017*. LNCS, vol. 10777, pp. 207–216. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-78024-5_19
20. Myllykoski, M., Kjelgaard Mikkelsen, C.C., Schwarz, A., Kågström, B.: D2.7: Eigenvalue solvers for nonsymmetric problems. Tech. rep., Umeå University (2019), <http://www.nlafet.eu/wp-content/uploads/2019/04/D2.7-EVP-solvers-evaluation-final.pdf>
21. Thibault, S.: On runtime systems for task-based programming on heterogeneous platforms (2018), Habilitation à diriger des recherches, Université de Bordeaux