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

Postprint

This is the accepted version of a paper presented at *12th International Conference on Parallel Processing and Applied Mathematics (PPAM 2017)*.

Citation for the original published paper:

Mylykoski, M. (2018)

A Task-Based Algorithm for Reordering the Eigenvalues of a Matrix in Real Schur Form
In: Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, Konrad Karczewski (ed.),
Parallel Processing and Applied Mathematics: PPAM 2017 (pp. 207-216). Springer
Lecture Notes in Computer Science

https://doi.org/10.1007/978-3-319-78024-5_19

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-145987>

A Task-Based Algorithm for Reordering the Eigenvalues of a Matrix in Real Schur Form^{*}

Mirko Myllykoski

Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden
`mirko.myllykoski@umu.se`

Abstract. A task-based parallel algorithm for reordering the eigenvalues of a matrix in real Schur form is presented. The algorithm is realized on top of the StarPU runtime system. Only the aspects which are relevant for shared memory machines are discussed here, but the implementation can be configured to run on distributed memory machines as well. Various techniques to reduce the overhead and the core idle time are discussed. Computational experiments indicate that the new algorithm is between 1.5 and 6.6 times faster than a state of the art MPI-based implementation found in ScaLAPACK. With medium to large matrices, strong scaling efficiencies above 60% up to 28 CPU cores are reported. The overhead and the core idle time are shown to be negligible with the exception of the smallest matrices and highest core counts.

Keywords: eigenvalue reordering problem, task based programming, shared memory machines

1 Introduction

Every real matrix $A \in \mathbb{R}^{n \times n}$ has a real Schur decomposition $A = QSQ^T$ where S is a quasi triangular matrix with 1×1 and 2×2 blocks on its diagonal (i.e., S is a real Schur form of the matrix A) and $Q \in \mathbb{R}^{n \times n}$ is an orthogonal matrix, often referred to as the Schur basis. The 1×1 blocks on the diagonal of S are the real eigenvalues of A . Similarly, the complex conjugate pairs of eigenvalues of A appear as 2×2 blocks on the diagonal of S . These 1×1 and 2×2 blocks are thereafter referred to simply as blocks when it does not cause ambiguity. This paper deals with the numerical problem of forming an updated real Schur decomposition $A = \hat{Q}\hat{S}\hat{Q}^T$, where a user selected subset of eigenvalues of A appears in the leading diagonal blocks of the updated Schur form $\hat{S}$. In other words, the eigenvalues of a matrix in real Schur form are reordered.

^{*} Please cite this article as: M. Myllykoski, A Task-Based Algorithm for Reordering the Eigenvalues of a Matrix in Real Schur Form, In Parallel Processing and Applied Mathematics, 12th International Conference, PPAM 2017, Lublin, Poland, September 10-13, 2017, Revised Selected Papers, Part I, Lecture Notes in Computer Science, Vol. 10777, Wyrzykowski R., Dongarra J., Deelman E., Karczewski K. (eds), Springer International Publishing, pp. 207–216, 2018. The final authenticated version is available online at https://doi.org/10.1007/978-3-319-78024-5_19.

One of the simplest use cases for this reordering process occurs when the real Schur decomposition has already been formed (using the QR algorithm) and interest is focused on a m -dimensional invariant subspace $\mathcal{V}$ of A . In particular, the objective is to obtain an orthonormal basis for $\mathcal{V}$. This can be accomplished by reordering the eigenvalues that correspond to $\mathcal{V}$ to the upper left corner of the updated Schur form $\hat{S}$. Due to the properties of the real Schur decomposition, the first m columns of the updated Schur basis $\hat{Q}$ will then form an orthonormal basis for $\mathcal{V}$.

This paper presents a task-based parallel algorithm for reordering the eigenvalues of matrices in real Schur form. The algorithm is realized on top of the StarPU task-based runtime system [3]. Only the aspects relevant to the shared memory use case are covered in this paper but the implementation can be configured to run on distributed memory machines as well. The entire algorithm and its StarPU realization cannot be fully described here due to page constraints. Thus, this paper focuses mainly on highlighting the key features of the new algorithm. A more detailed description can be found in technical report [7].

The rest of the paper is organized as follows: Section 2 reviews the existing algorithms for the (standard) eigenvalue reordering problem. Section 3 describes the task-based algorithm and its StarPU realization. The computational results are presented in Sect. 4 and the final conclusions are given in Sect. 5.

2 Related Work

Most existing reordering algorithms are built around the swapping kernels developed by Bai and Demmel [4]. The kernels are best explained by concentrating on a real Schur form S that has only two diagonal blocks, that is,

$$S = \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix},$$

where S_{11} and S_{22} have size at most 2×2 . The swapping kernels can be used to construct an orthogonal matrix V such that

$$\hat{S} = V^T S V = \begin{bmatrix} \tilde{S}_{11} & \tilde{S}_{12} \\ 0 & \tilde{S}_{22} \end{bmatrix},$$

where $\tilde{S}_{11}$ has the same eigenvalues as S_{22} and $\tilde{S}_{22}$ has the same eigenvalues as S_{11} . Thus, the two blocks have been effectively swapped and $A = \hat{Q} \hat{S} \hat{Q}^T$, where $\hat{Q} = QV$. The idea can be generalized to larger real Schur forms by combining the swapping kernels with a bubble sort style sorting algorithm. More specifically, the selected diagonal blocks are moved, one at a time, to the upper left corner of the Schur form by repeatedly swapping adjacent blocks. The DTRSEN subroutine found in the LAPACK library [1] implements this algorithm in a robust manner.

The biggest drawback of the approach used in DTRSEN is that it is inherently memory-bound. In particular, the orthogonal updates that are applied after each swap are very thin (at most 4 rows/columns high/wide) which leads

Fig. 1. An illustration of how three selected diagonal 1×1 blocks are gathered to the upper left corner of the Schur form by chaining three diagonal windows together. These three overlapping windows form a window chain.

to a very low cache reuse. Fortunately, this situation can be improved by grouping the selected blocks and moving each group to the upper left corner of the Schur form in a blocked manner as done by Kressner [6]. For each group, a computational window is placed on the diagonal such that the selected block that is furthest down the diagonal is located flush against the bottom right corner of the window as shown in the leftmost illustration of Fig. 1. The swaps are initially performed only inside the diagonal window and the related orthogonal transformations are accumulated into a separate accumulator matrix. This leads to a much higher cache reuse as the diagonal window can be made small enough to fit inside CPU caches. All related orthogonal transformations are applied to the remaining parts of the Schur form S and the Schur basis Q by multiplying the relevant sections with the accumulator matrix. Furthermore, multiple overlapping diagonal windows can be chained together as shown in Fig. 1 and the complete reordering procedure can involve multiple *window chains*.

The blocked approach enables the development of efficient parallel algorithms. In particular, Granat, Kågström, and Kressner [5] presented a MPI-based parallel algorithm which is currently part of the ScaLAPACK library [2] as the PDTRSEN subroutine. The authors introduced the idea of processing several diagonal windows concurrently. The algorithm relies on broadcast messages (within the block row/column communicators) to communicate the accumulator matrices to the MPI processes that propagate the accumulated updates and performs two global synchronizations each time a set of diagonal windows is moved across the MPI process boundaries.

3 New Task-Based Algorithm

The main difference between the new approach presented in this paper and the existing algorithms is that it is expressed in the terms of the sequential task-flow (STF) model. This means that the various computational operations are encapsulated inside tasks and the tasks are inserted to the runtime system in a sequentially consistent order. This approach has two primary benefits: Firstly, proper use of the STF model exposes the underlying parallelism automatically. In particular, advanced runtime systems, such as StarPU, are able to determine when multiple diagonal windows can be processed in parallel. Secondly,

since the data dependency tracking is offloaded to the runtime system, many things that might have previously caused an algorithm designer to resort to using (global) synchronization (cf. global synchronization in PDTRSEN) are now handled automatically. This means that the runtime system can merge different computational stages together, thus potentially reducing the idle time that might otherwise occur between different stages of the algorithm due to imperfect load balancing.

The Schur form S and the Schur basis Q are partitioned into square tiles¹. This partitioning scheme is very natural for this type of two sided transformation algorithms as the Schur form S is updated from both the right and the left. Two dimensional tiling allows different sections of the Schur form to be operated on without introducing too many spurious data dependencies (i.e., data dependencies that are purely the result of the tile-based dependency tracking) between otherwise independent operations. StarPU considers each tile to be an independent unit of data and the data dependencies are automatically inferred from the corresponding data handles.

As with the blocked algorithm suggested by Kressner, the elementary tool used in the new algorithm is the ability to process a small diagonal window, accumulate the related orthogonal transformations and propagate the remaining updates in the accumulated form. The algorithm consists of three task types: The `process_window` tasks perform the necessary computations inside the diagonal windows. The `left_update` tasks propagate the accumulated left-hand side updates that relate to the diagonal windows. The `right_update` tasks propagate the accumulated right-hand side updates that relate to the diagonal windows.

All task types share certain features. The contents of the tiles that intersect task's computational window are copied to a separate scratch buffer and copied back once the computations have been completed. This isolates the computational kernels from the tile-based construction of the algorithm. The actual computations are performed by calling well established and vendor optimized subroutines from the LAPACK and BLAS libraries. In particular, the `left_update` and `right_update` tasks are implemented around the high performance BLAS 3 DGEMM subroutine.

Each diagonal window induces a set of left and right updates. In order to achieve a sufficient level of parallelism, the left and right updates are divided into update tasks such that the total number of update tasks induced by a `process_window` task is about twice the number of StarPU worker threads. More specifically, the algorithm forms a two-dimensional stencil (see Fig. 2) that separates the tiles into groups of adjacent tiles both vertically and horizontally. The number of groups in each direction is the same as the number of StarPU worker threads. This stencil cuts the right and left updates into tasks in vertical and horizontal directions, respectively. Moreover, as the edges of the stencil follows the boundaries of the underlying tiles, the approach does not induce vertical de-

¹ The word tile is used here to differentiate the diagonal 1×1 and 2×2 blocks from the data partitioning blocks/tiles.

Fig. 2. An illustration of how updates are divided into update tasks by using a 4×4 stencil. This corresponds to a case where StarPU has four worker threads. Each group contains two tiles in each direction. The right updates are divided into two **right_update** tasks and the left updates are divided into three **left_update** tasks.

dependencies between the **right_update** tasks or horizontal dependencies between the **left_update** tasks.

The placement of the windows on the diagonal is one of the deciding factors for the performance of the algorithm. In particular, it is important that the number of spurious data dependencies is kept at a minimum. The algorithm aims to accomplish this objective by selecting the groups of neighbouring diagonal blocks that are moved together such that all blocks in a group can be fitted inside a single tile when clustered together. That is, each group should contain strictly less than tile size eigenvalues. The reason why each group contain at most that many eigenvalues can be seen in Fig. 3. The algorithm tries to place the diagonal windows such that their upper left corners follow the boundaries of the underlying tiles. This, of course, is not always possible as shown in Fig. 3. The window size is not fixed, but grows as more and more (selected) blocks are gathered from the diagonal. However, the window size is explicitly limited to twice the size of the underlying tiles.

The tile size is a tunable parameter. If the tile size is too small in relation to the matrix dimension, then the total number of tiles is going to be large and the overhead resulting from internal data dependency tracking happening inside StarPU is likely to cause problems. In addition, as mentioned above, the window size is linked to the tile size which means that the tile size is also closely connected to task granularity. A task granularity which is too fine will also lead to excessive overhead. On the other hand, if the tile size is too large, then left and right updates cannot be divided into enough **left_update** and **right_update** tasks to saturate all StarPU worker threads. It is also preferable to keep the size of the diagonal window below the capacity of the L2 cache. Since the window size is connected to the tile size, a tile size which is too large is likely to reduce the performance of the **process_window** tasks.

Fig. 3. An illustration of how the windows are placed on the diagonal such that their upper left corners follow the boundaries of the underlying tiles. Note how the 2×2 block in the center of the matrix causes a slight deviation from this rule and how all selected blocks always end up inside the same tile.

Initial parameter sweeps (see Fig. 4) suggested that the optimal tile size depends linearly on the matrix dimension but only up to a certain point. This is consistent with the earlier observations. However, the increased overhead seemed to dominate over decreased cache reuse. That is, for larger matrices, the diagonal windows ended up becoming much larger than what can be fitted inside CPU's L2 cache. This led to a very poor performance in the `process_window` tasks but the overhead remained at manageable levels.

The problem was solved by modifying the `process_window` task's implementation such that windows that are bigger than a given threshold are reordered in a blocked manner. Parameters sweeps suggest that the threshold should be about 128 and the small window size about 64. This recursive blocking approach led to much better performance in the `process_window` tasks and it also improved the average performance by 25%. More importantly, as shown in Fig. 4, the optimal tile size now depends linearly on the matrix size. This means that the number tiles (and the number of tasks) is independent from the matrix size. The recursive blocking approach also opens the possibility to parallize the `process_window` task's implementation. Parallel `process_window` task is not implemented yet.

The tasks can be inserted in many sequentially consistent orders. In a straightforward approach, the window chains (see Fig. 1) and the related update tasks are inserted in order starting from the window chain that contains the topmost diagonal window. This insertion order is simple to implement and guarantees that the updates are performed in the correct order. However, since the window chains overlap each other, a window chain may get blocked by the preceding window chains which may lead to excess idle time. In certain sense, the window chains can also be processed in a reverse order. An exact description can be found in [7]. Figure 5 shows a comparison between the straightforward insertion order and the reverse insertion order. The result shows that when a large percentage of the diagonal blocks is selected and the matrix size is small (i.e., it is likely that the window chains end up blocking each other and the tasks are

Fig. 4. Optimal tile size when 15% of the diagonal blocks are randomly selected and 28 StarPU worker threads are used. A similar pattern repeats regardless what percentage of the diagonal blocks is selected or how many worker threads are used.

very small), the reverse task insertion order can lead up to 55% improvement in performance.

4 Computational Experiments

All numerical experiments were performed on a system called Kebnekaise, which is located at the High Performance Computing Center North (HPC2N) at Umeå University. Each compute node in Kebnekaise contains two 14-core Intel Xeon E5-2690v4 CPUs organized into two NUMA islands. CPU's AVX base frequency is 2.1 GHz and the AVX boost frequency varies between 0.8 GHz and 1.4 GHz. All experiments were performed using double precision floating-point arithmetic. The application binary was linked against Intel Math Kernel Library.

An extensive set of test problems was generated for the computational experiments. Matrix dimension of the matrix A varied between 5000×5000 and 40000×40000 . In each matrices, half the eigenvalues were part of complex conjugate pairs and the corresponding 2×2 blocks were evenly distributed on the diagonal of the Schur form S . The probability that a given diagonal block gets selected was either 5%, 15%, 35% or 50%. That is, the selected diagonal blocks are uniformly distributed along the diagonal of the Schur form S and either 5%, 15%, 35% or 50% of the diagonal blocks are selected. A uniform random distribution makes the results easily replicable and the execution time coefficient of variation is most cases only a few percent. Due to page constraints, only a small subset of the results are shown here. Nevertheless, this reduced sample is a good representative of the overall results (see [7]).

Tables 1 and 2 show the median execution times for the StarPU realization of the new algorithm and the MPI-based PDTRSEN subroutine. The MPI-based subroutine was configured to use shared memory communication fabric. The

Fig. 5. Observed speedup when the reverse tasks insertion order is compared against the straightforward tasks insertion order.

Table 1. Execution times in seconds when 5% of the diagonal blocks are selected.

n	1 core	4 cores		9 cores		16 cores		25 cores	
	StarPU	MPI	StarPU	MPI	StarPU	MPI	StarPU	MPI	StarPU
5000	1.9	1.8	0.6	1.3	0.4	0.7	0.4	0.6	0.3
10000	13	12	3.8	6.6	2.0	5.7	1.3	5.1	1.0
20000	98	81	26	58	13	37	7.9	33	5.5
30000	280	265	78	160	37	109	22	97	15
40000	612	702	176	445	83	239	49	211	33

results show that the StarPU realization is between 1.5 and 6.6 times faster than the PDTRSEN subroutine. In general, the performance difference increases when the matrix dimension and/or the CPU core count are increased. In particular, the StarPU implementation appears to perform much better when 5% of diagonal blocks are selected (excluding the smallest matrix size).

Figure 6 shows the strong scalability results for the StarPU implementation when 5% and 35% of the diagonal blocks are selected. The parallel efficiency stays above 0.7 for the larger matrices ($30000 \leq n$) but drops all the way to 0.2 for smallest matrix when only 5% of the diagonal blocks are selected. The parallel efficiency improves when a larger percentage of diagonal blocks is selected. In particular, when 35% or more of the diagonal blocks are selected, the parallel efficiency stays above 0.7 for larger matrices ($20000 \leq n$) and above 0.45 for all matrices.

Figure 7 shows how the execution time is divided among the overhead, the idle time and the three computational tasks when 15% of the diagonal blocks are selected. The StarPU startup and shutdown times are included in the reported overhead. In general, the overhead and the idle time are negligible when $20000 \leq n$. Furthermore, when 35% or more of the diagonal blocks are selected, the overhead and the idle time are negligible when $10000 \leq n$. However, the overhead and the idle time constitute over 60% of the total execution time in a case where

Table 2. Execution times in seconds when 35% of the diagonal blocks are selected.

n	1 core	4 cores		9 cores		16 cores		25 cores	
	StarPU	MPI	StarPU	MPI	StarPU	MPI	StarPU	MPI	StarPU
5000	7.6	4.2	2.3	3.2	1.2	2.2	0.8	2.5	0.6
10000	52	28	15	20	7.4	14	4.4	12	3.1
20000	369	191	102	145	49	90	30	83	19
30000	1744	620	328	430	161	284	94	245	60
40000	2750	1510	770	1063	373	606	222	537	147

Fig. 6. Strong scalability of the StarPU implementation when 5% (on the left) and 35% (on the right) of the diagonal blocks are selected.

the matrix is size is 5000, only 5% of the diagonal blocks are selected and the number of StarPU worker threads is 28.

5 Conclusions and Future Work

This paper presented a task-based parallel algorithm for reordering the eigenvalues of matrices in real Schur form. The recursive blocking technique was shown to lead on average 25% performance improvement. Moreover, the parameter sweeps showed that the optimal tile size can be made to depend linearly on the matrix size which means that the number tiles (and the number of tasks) is independent from the matrix size. An optimized task insertion order was shown to lead up to 55% performance improvement. The presented computational results illustrate the power of the task based approach. In particular, the StarPU realization of the new algorithm was shown to be between 1.5 and 6.6 times faster than the state of the art MPI-based implementation (PDTRSEN) found in ScaLAPACK. With medium to large matrices, strong scaling efficiencies well above 60% up to 28 CPU cores were reported. The overhead and the core idle time were shown to be negligible except for the smallest matrices.

The StarPU implementation can be configured to run on distributed memory machines. It suffices to provide StarPU with the distribution of the tiles across

Fig. 7. Overhead, idle time and actual computation as relative fractions of the execution time when 15% of the diagonal blocks are selected.

the MPI processes. Once this information is known, StarPU is able to perform all necessary process to process commutations automatically. However, additional work needs to be invested in making the implementation run efficiently on distributed memory machines. The core features of the algorithm generalize to generalized and complex Schur forms. The generalized case is already implemented. CUDA support is currently in development.

Acknowledgements

This work is part of a project that has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 671633 (NLAFET). Support has also been received from eSCIENCE, a collaborative e-Science program funded by the Swedish Government via the Swedish Research Council (VR). The author would like to extend his gratitude to Asst. Prof. Lars Karlsson and Dr. Carl Christian Kjelgaard Mikkelsen for their valuable comments and suggestions. Moreover, the author would like to thank Lic. Björn Adlerborn, Prof. Daniel Kressner (EPFL) and Prof. Bo Kågström, who is coordinator and scientific director of the NLAFET project, as well as the StarPU development team for answering various question on StarPU. Finally, the author thanks the anonymous reviewers for their valuable feedback.

References

1. LAPACK — Linear Algebra PACKage, see <http://www.netlib.org/lapack/>.
2. ScaLAPACK — Scalable Linear Algebra PACKage, see <http://www.netlib.org/scalapack/>.
3. StarPU — A Unified Runtime System for Heterogeneous Multicore Architectures, see <http://starpup.gforge.inria.fr/>.

4. Bai, Z., Demmel, J.W.: On swapping diagonal blocks in real Schur form. *Linear Algebra Appl.* 186, 73–95 (1993), DOI: 10.1016/0024-3795(93)90286-W.
5. Granat, R., Kågström, B., Kressner, D.: Parallel eigenvalue reordering in real Schur forms. *Concurrency and Computation: Practice and Experience* 21(9), 1225–1250 (2009), DOI: 10.1002/cpe.1386.
6. Kressner, D.: Block Algorithms for Reordering Standard and Generalized Schur Forms. *ACM Transactions on Mathematical Software* 32(4), 521–532 (December 2006), DOI: 10.1145/1186785.1186787.
7. Myllykoski, M., Kjelgaard Mikkelsen, C.C., Karlsson, L., Kågström, B.: Task-Based Parallel Algorithms for Eigenvalue Reordering of Matrices in Real Schur Form. *NLAFET Working Note* WN-11 (April, 2017), also as Report UMINF 17.11, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.