Task-Based Parallel Algorithms for Eigenvalue Reordering of Matrices in Real Schur Forms[*]

Mirko Myllykoski
Carl Christian Kjelgaard Mikkelsen
Lars Karlsson
Bo Kågström

May 29, 2017

Abstract

We develop a task-based parallel algorithm for reordering eigenvalues of matrices in real Schur form. We describe how we implemented the algorithm using `StarPU` runtime system and report on experiments performed on a shared memory machine. Compared with `ScaLAPACK` we achieve average speedup of 3. We have strong and weak scaling efficiencies which are well above 50%. We are able to achieve more than 50% of the peak flop rate for all but the smallest matrices. The idle time and the overhead is negligible except for the smallest matrices. The next step is to reconfigure and further develop the code so that it can be applied to matrix pairs in generalized Schur forms and run efficiently on distributed memory machines.

[*]NLAFET Working Note 11. Report UMINF 17.11, Dept. Computing Science, Umeå University, SE 901 87 Umeå, Sweden. Results were presented at the SIAM Conference on Computational Science and Engineering, Atlanta, Georgia, USA, 2017.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Let $A \in \mathbb{R}^{n \times n}$ be a real matrix of dimension n. A non-zero vector x is an *eigenvector* of A if there exists an *eigenvalue* λ such that $Ax = \lambda x$. A subspace $\mathcal{V} \subset \mathbb{R}^{n \times n}$ is said to be *invariant* with respect to A if $v \in \mathcal{V} \Rightarrow Av \in \mathcal{V}$. If x is an eigenvector of A, then $\mathcal{V} = \text{span}\{x\}$ is an invariant subspace of A. In general, let $v_1, v_2, \ldots, v_m$ be a set of eigenvectors of A corresponding to the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_m$. Then $\mathcal{V} = \text{span}\{v_1, v_2, \ldots, v_m\}$ is an invariant vector space of A, since for any $v \in \mathcal{V}$ we have

$$Av = A\left(\sum_{i=1}^{m} \alpha_i v_i\right) = \sum_{i=1}^{m} \alpha_i A v_i = \sum_{i=1}^{m} \alpha_i \lambda_i v_i \in \mathcal{V}. \tag{1}$$

In many situations one wants to compute an orthonormal matrix V whose columns form an orthonormal basis for $\mathcal{V}$. This can be accomplished by first computing a *real Schur decomposition* $A = QSQ^T$ using the QR algorithm and then reordering the decomposition such that the eigenvalues associated with the desired invariant subspace appear in the leading diagonal blocks of the updated matrix S. The first m columns of the updated matrix Q will then form an orthonormal basis for $\mathcal{V}$. Above, S is a real upper quasi-triangular matrix with 1×1 and 2×2 blocks on the diagonal. In other words, S is a Schur form of the matrix A. The orthogonal matrix Q is the related Schur basis. Note that even if the entries of the matrix A are all real, some of the eigenvalues and eigenvectors may still be complex. In particular, the complex conjugate pairs of eigenvalues of A appear as 2×2 blocks on the diagonal of S. We will hereafter refer to the 1×1 and 2×2 blocks simply as *blocks* when it does not cause ambiguity.

For the generalized eigenvalue problem, the concept of a *pair of deflating subspaces* generalizes the notion of an invariant subspace. Orthonormal bases for deflating subspaces can be computed in a similar fashion by first decomposing the matrix pair into its generalized Schur form by applying the QZ algorithm and then reordering the selected (generalized) eigenvalues to the top left corner of the generalized Schur form. There are real and complex arithmetic analogues of both the Schur and generalized Schur forms and corresponding versions of the QR and QZ algorithms. For state-of-the-art MPI-based parallel algorithms for the multishift QR and QZ algorithms with aggressive early deflation and the computation of invariant or deflating subspaces, we refer to [2, 5].

Our long-term objective is to develop task-based parallel algorithms for the eigenvalue reordering stage. We aim to cover both the standard and generalized Schur forms, in both real and complex arithmetic, and for both shared and distributed memory. In other words, a total of 8 variants. In this report, we focus on standard Schur forms in real arithmetic. We develop a task-based parallel algorithm for shared memory systems and implement it on top of the `StarPU` runtime system [1]. The core components of the algorithm generalize to generalized Schur forms and complex arithmetic. In addition, the `StarPU` runtime system has features that make it easy to convert a shared memory realization of an algorithm to a distributed memory realization. Thus, extending the code to the other use cases is mostly a matter of reconfiguring the current realization correctly and will be discussed at a later date.

The rest of the paper is organized as follows. Section 2 reviews the existing state-of-the-art algorithms for the standard eigenvalue reordering problem. We discuss the fundamental kernels as well as both sequential and parallel algorithms. Fundamental aspects of the new task based parallel algorithm are described in Section 3. Technical implementation aspects of the new task based algorithm are addressed in Section 4. Results of experimental evaluations of the algorithm are described in Section 5 and Section 6 concludes with some related and future work.

2 State-of-the-art

In this section we briefly review the related work which is directly relevant for our current work. We begin by discussing the kernels which form the basis of the considered algorithms. Moreover, we discuss the sequential algorithm which is implemented in `LAPACK` as `DTRSEN`, Kressner's blocked improvement `BDTRSEN`, as well as, the parallel algorithm implemented in `ScaLAPACK` as `PDTRSEN`. We refer to the cited references and the references therein for a more thorough survey of the field.

2.1 Fundamental swapping kernels

The swapping kernels developed by Bai and Demmel [3] form the basis for most known algorithms for the eigenvalue reordering problem. Consider a matrix S in real Schur form which 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. Then the swapping kernels can be used to compute an orthogonal matrix V such that

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

where $\tilde{S}_{11}$ has the same eigenvalues as S_{22} and $\tilde{S}_{22}$ has the same eigenvalues as S_{11}. We say that the blocks/eigenvalues have been swapped.

The swapping kernels are based on the robust solution of a tiny Sylvester equation by solving the equivalent linear system using Gaussian elimination with complete pivoting and scaling to avoid overflow. Backward stability is guaranteed by cheaply monitoring the perturbations introduced by each swap and rejecting those swaps that lead to unacceptably large errors. The authors were unable to find or construct an example were the method failed, indicating that even though failures are a theoretical possibility they are extremely rare in practice. Kågström and Poromaa (1996) [6] have extended the work by Bai and Demmel to the generalized eigenvalue reordering problem. Their algorithm is based on the robust solution of a tiny generalized Sylvester equation and similarly guarantees backward stability.

2.2 Sequential algorithm implemented in DTRSEN

The sequential algorithm implemented in LAPACK as DTRSEN is based on the swapping kernels by Bai and Demmel [3]. The selected blocks are systematically reordered to the top left corner of the matrix by repeatedly swapping *adjacent* blocks. The algorithm scans the diagonal of S from the upper left to the lower right corner and moves any selected blocks to the top by a sequence of swaps. The orthogonal transformation constructed from a small submatrix enclosing a pair of adjacent blocks affects all entries above and to the right of the submatrix. In principle, it is possible to swap any pair of blocks, but unless the blocks are adjacent, the real Schur form is destroyed by fill-in. The biggest drawback of the approach used in DTRSEN is that it relies heavily on low-level BLAS operations and is therefore inherently memory-bound.

2.3 Blocked algorithm implemented in BDTRSEN

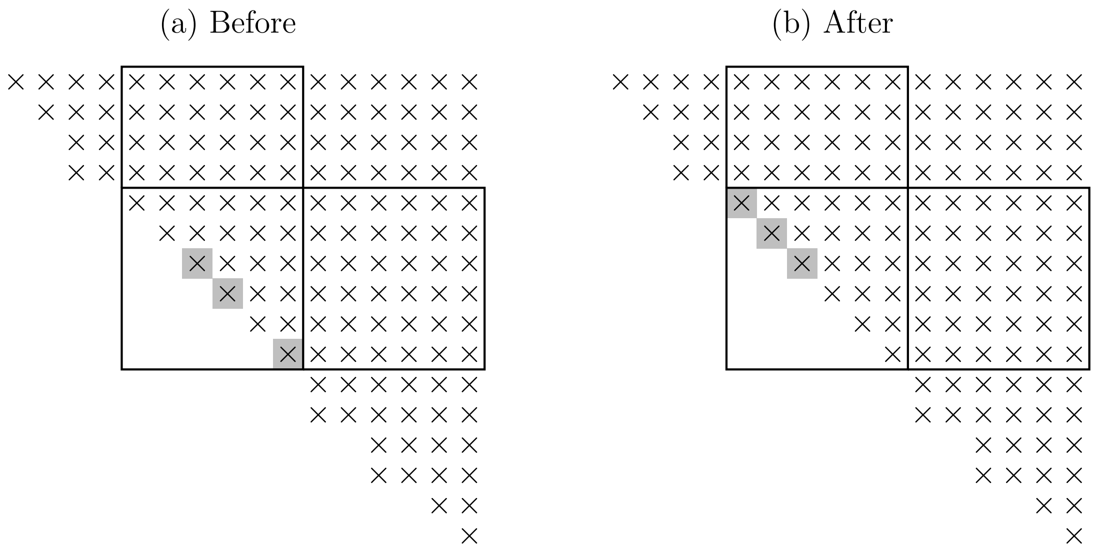


Figure 1: An example of a situation where three selected diagonal blocks (which all happen to be of size 1×1) are reordered in a blocked manner. The location of each selected block is highlighted before (on the left) and after (on the right) the reordering. A window of size 6×6 is placed on the diagonal and the selected blocks are reordered by a sequence of swaps (in this case, $2 + 2 + 3 = 7$ swaps). The off-diagonal submatrices that will be updated afterwards are also highlighted.

Kressner (2006) [7] improves upon the scalar DTRSEN algorithm by reorganizing the computations for improved cache reuse. More specifically, a group of diagonal blocks is reordered within a small window on the diagonal as shown in Figure 1. The diagonal window is placed on the diagonal such that the selected block furthest down the diagonal is located flush against the bottom right corner of the window. The key idea is to only reorder the diagonal window at this stage and accumulate the related similarity transformation into a separate accumulator matrix. After processing the window, the accumulator matrix, which is itself a similarity transformation, is then applied from the right to the submatrix above the diagonal window and from the left to the submatrix to the right of the diagonal window. The Schur form Q is also updated from the right. The off-diagonal updates can be performed using level 3 `BLAS` subroutines which leads to much higher arithmetic intensity. In addition, the diagonal window can be made small enough to fit into CPU caches. Alternatively, the similarity transformations can be applied in their original factored form.

2.4 Parallel algorithm implemented in `PDTRSEN`

Granat, Kågström, and Kressner (2009) [4] present a parallel algorithm for both the standard and generalized reordering problems. The algorithms are based on those in [7] and the software is expressed in the `ScaLAPACK` style, that is, the processes are arranged in a mesh and the matrices are distributed in a two-dimensional block-cyclic fashion. Several computational windows are introduced to increase the degree of concurrency. The algorithm broadcasts the accumulator matrices separately to those `MPI` nodes that perform the right updates and to those `MPI` nodes that perform the left updates. In addition, the algorithm performs two global synchronizations each time a set of diagonal windows is moved across `MPI` node boundaries. One of the downsides of this approach is that the broadcasts and, in particular, the global synchronizations form bottlenecks which limit the level of parallelism.

3 Fundamental aspects of the new task-based algorithm

The aim of this section is to sketch the fundamental aspects of the task-based parallel algorithm. Section 4 will delve into the many conceptual and technical complications that must be addressed when mapping the algorithm to the `StarPU` runtime system. We find it easier to comprehend the core of the algorithm when it is isolated from the practical complications in this fashion.

As with the blocked `BDTRSEN` algorithm (see Subsection 2.3), the elementary tool used here is the ability to process a small diagonal window, accumulate the related orthogonal transformation to an accumulator matrix (*local Q matrix*) and only later apply the related off-diagonal updates. Furthermore, multiple overlapping diagonal windows can be chained together as shown in Figure 2, which allows us to gather a set of selected blocks to a desired position on the diagonal. More specifically, the first window is placed such that the selected block that is furthest down the diagonal is located flush against the bottom right corner of the window. The remaining windows are placed such that the overlap between two windows is exactly big enough to accommodate all selected blocks that fall within the preceding windows. In this way, the windows can be processed in sequential order, starting from the bottom window, such that the reordering that takes place in one window always

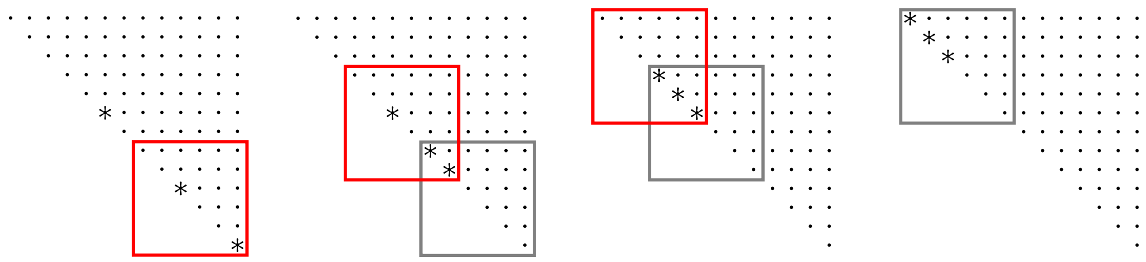


Figure 2: An example of how three diagonal windows are chained together in an overlapping manner so that three selected blocks are be gathered to the upper left corner of the matrix. Each subfigure corresponds to a diagonal window (highlighted with red color) and a set of related off-diagonal updates. The previous window is highlighted with gray color. Note how the overlap between two windows grows as more and more selected blocks are gathered from the diagonal.

moves the gathered blocks to the lower right corner of the next window. In the end, all selected blocks that fell within the combined computational area of the *window chain* are moved to the upper left corner of the topmost window.

The number of selected blocks that can be moved by a single window chain is limited by the window size. Thus, the complete reordering procedure usually involves multiple window chains that must be processed in a particular order. The selected blocks are divided into multiple disjoint subsets or *groups* containing some number of neighboring selected blocks (see Figure 3a). The first window chain is placed such that the first group falls within its combined computational area and topmost window is placed in the upper left corner of the matrix. The next window chain is placed such that the second group falls within its combined computational area. Moreover, its topmost window is placed such that its upper left corner is located one diagonal entry after the location where the last block of the first group gets moved (see Figures 3b and 3c). The same procedure is then repeated until all groups have been accounted for.

The main difference between the previous algorithms and our new approach is that we have expressed our algorithm in the terms of the sequential task-flow (STF) model roughly in the way shown in Algorithm 1. This means that we have encapsulated the various computational operations inside *tasks* that are created in a sequentially consistent order and all task dependencies can therefore be mechanically deduced by analyzing the data flow.

The main benefits of this new approach are:

- Proper use of the STF model exposes the underlying parallelism automatically. In particular, advanced runtime systems such as `StarPU` are able to automatically determine when multiple chains can be processed in parallel. This can increase the level of parallelism significantly.
- The approach does not require global synchronization (cf. global synchronization in `PDTRSEN`). Instead, synchronization is done automatically on much lower level using

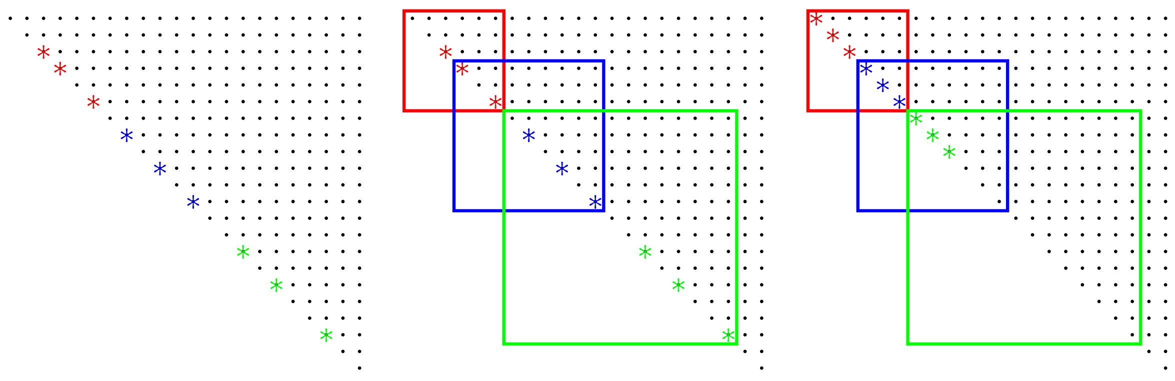

(a) Groups (b) Window chains (c) After reordering

Figure 3: Examples of (a) how the selected blocks are divided into three groups (with three 1×1 blocks in each) highlighted with colors red, blue and green; (b) how the groups are assigned to the window chains; and (c) how the matrix looks like after the window chains have been processed. Note that we have visualized each window chain as a square whose top left corner corresponds to the upper left corner of the topmost window in the window chain. Similarly, the bottom right corner of the square corresponds to the bottom right corner of the bottom window in the window chain.

information derived from the data flow. This means that different computational stages can be merged together, thus reducing the idle time that might otherwise occur between the stages.

- Different tasks can be given different priorities. This allows us to delay less important tasks and instead concentrate on the critical path.
- The same code can be configured for shared memory and distributed memory use cases. All necessary node to node communications can be derived from the data flow once the inter-node data distribution is known.
- In future, a support for accelerator devices such as GPUs can be integrated into our implementation with relative ease as only the window processing and update kernels need be implemented for each computing platform. The runtime system will handle all data transfers and make necessary scheduling decisions automatically[1].

4 Implementation of the new task based algorithm

4.1 Data partition and distribution

`StarPU` uses *data handles* to model the data flow between tasks. A data handle can

[1]The algorithm may have to specify some extra information about the various implementations so that `StarPU` can make informed decisions.

Algorithm 1: Task-based parallel eigenvalue reordering for standard Schur forms

```
Data: A real Schur decomposition $A = QSQ^T$ and a Boolean array that defines the
      selected subset of the blocks along the diagonal of $S$.
Result: An updated Schur decomposition $A = \tilde{Q}\tilde{S}\tilde{Q}^T$ for which the selected subset
        of the blocks are gathered in the top left corner of $\tilde{S}$.
1 Partition the selected blocks into $m$ groups of neighboring blocks (cf. Figure 3a);
2 for each group from the top down do
3     while some selected blocks in the current group are not in their final positions do
4         Place a window such that the block in the current group which is furthest
          down the diagonal is located flush against the bottom right corner of the
          window (cf. Figure 2);
5         Create a task for the processing of the window;
6         Create a set of independent tasks for the off-diagonal updates of $S$ from the
          left by partitioning the corresponding submatrix into blocks of columns and
          assign one task to each block;
7         Create a set of independent tasks for the off-diagonal updates of $S$ from the
          right by partitioning the corresponding submatrix into blocks of rows and
          assign one task to each block;
8         Create a set of independent tasks for the updates of $Q$ from the right by
          partitioning the corresponding submatrix into blocks of rows and assign one
          task to each block;
```

encapsulate any conceivable data type but the built-in *data interfaces* for scalars, vectors and matrices are adequate for many use cases. In our algorithm, the Schur form S and the Schur basis Q are partitioned into square *tiles* as shown in Figure 4, and each non-zero tile is registered with `StarPU` using a matrix interface. This means that `StarPU` considers each tile to be an independent unit of data that can be acted upon. In addition, the Boolean array that is used to indicate which diagonal blocks are selected is partitioned using the same scheme. This partitioning scheme simplifies the implementation but, more importantly, it also allows us to operate on different sections of the matrices without introducing too many data dependencies.

`StarPU-MPI` is an extension that integrates `StarPU` with the standardized `MPI` message-passing system. An algorithm may define all node to node communications explicitly by using the provided wrapper functions or the algorithm may simply define how the data is distributed among the nodes and let `StarPU` handle all necessary communication automatically. In the second approach, the algorithm must provide an unique *tag* and a *rank* (i.e. owner) for each data handle that is in some way involved in distributed computations.

At the moment, our algorithm uses the second approach. We define the data distribution by splicing adjacent tiles into *sections* such that tiles that belong to the same section have a common owner (rank). More specifically, a $n \times n$ matrix is divided into $s \times s$

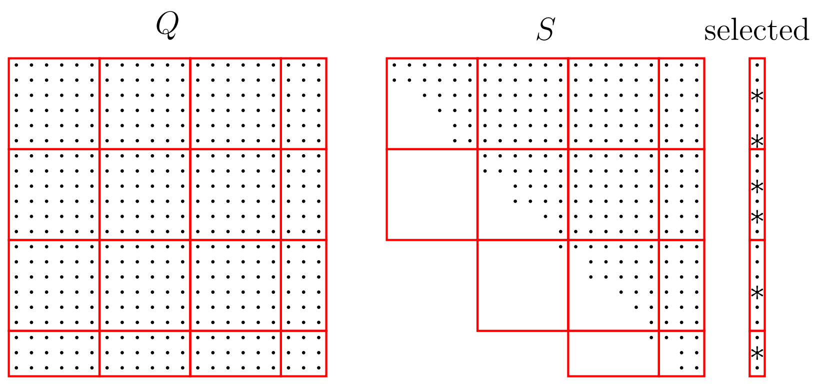


Figure 4: An example of how a Schur basis Q and the related Schur form S (both of size 21×21) are partitioned into square tiles of size 6×6. The Boolean array that defines the selected diagonal blocks is partitioned similarly. The left over tiles in the last tile row and the tile column are truncated to match the matrix dimensions. For technical reasons (that will be relaxed in the future), the tile size must be a multiple of 8.

sections which are then further divided into $t \times t$ tiles. Each section resides on some `MPI` node. We tried to reduce the overhead by making each `MPI` node register only those tiles that it actually needs. However, some tiles are needed by multiple `MPI` nodes. This is why we implemented a separate subsystem that will automatically register a "placeholder" data handle when a `MPI` node that does not own a tile requests a data handle to it. When this happens, the subsystem also communicates the correct tag and rank to `StarPU`.

4.2 Entry points

At the moment, the implementation has three entry points:

`reorder` entry point follows a traditional approach where the user specifies the problem through pointers, matrix dimensions and leading dimensions. All `StarPU` specific details are hidden from the user and the thread that enters the `reorder` function blocks until all tasks have been completed. However, the user can still influence `StarPU`'s behavior through optional arguments.

`submit_reorder` entry point assumes that the matrices and the Boolean array that defines the selected diagonal blocks are already partitioned in the correct way and registered with `StarPU`. The corresponding `StarPU` data handles are forwarded as arguments. The implementation extracts the block selection array and the locations of the 2×2 diagonal tiles from the input handles. The thread that enters the `submit_reorder` function blocks until all preceding dependencies related to the sub-diagonal and diagonal matrix S tiles have been resolved. The thread also blocks until all preceding dependencies related to the block selection array have been resolved. The thread returns from the function once all tasks have been inserted. This entry point allows different `StarPU` based functions to be merged together without introducing global synchronization points. The thread that enters the `submit_reorder` function will block but only a small subset of data handles are involved in the blocking operations.

`mpi_reorder` entry point follows an approach similar to the `reorder` entry point. The user specifies the data distribution and feeds the distributed data in the form of pointers, section dimensions and leading dimensions. All `StarPU` specific details are hidden from the user. The thread that enters the `mpi_reorder` function blocks until all local tasks and communications have been completed.

4.3 Task types

`StarPU` encapsulates various computational kernels inside objects called *codelets* and each task must have a codelet associated with it. Each codelet can have multiple implementations and `StarPU` is capable of automatically determining which implementation (and which computational resource) should be used in a given situation. This, however, requires that the algorithm specifies some extra information about the various implementations.

Our algorithm consists of three codelets:

`process_window` codelet performs the necessary transformations inside a diagonal computational window and accumulates the transformations into a local Q matrix. Depending on the window size, the codelet either reorders the window in a scalar manner similarly to `DTRSEN` or in a blocked manner similarly to `BDTRSEN`. Prior to this, the contents of the tiles that enclose the diagonal window are copied to a separate scratch buffer. The reordered window copied back to the tiles once the reordering has been completed.

`left_update` codelet performs a localized left update using a given local Q matrix. The tiles that enclose the computational area are copied to a separate scratch buffer and the actual update operation is performed using the `BLAS DGEMM` subroutine whose output is directed into a second scratch buffer. Finally, the result is copied from the second scratch buffer back to the original tiles.

`right_update` codelet performs a localized right update using a given local Q matrix. The functionality is analogous to the `left_update` tasks.

Currently, each one of the codelets only has a CPU implementation. The `right_update` codelet is the only one that is used to update both the Schur form S and the Schur basis Q. From now on, when we are talking about right updates or `right_update` tasks, we refer to those right updates or `right_update` tasks that modify the matrix S. In order to avoid ambiguity, we will explicitly mention when we are referring to right updates or `right_update` tasks that modify the Schur basis Q.

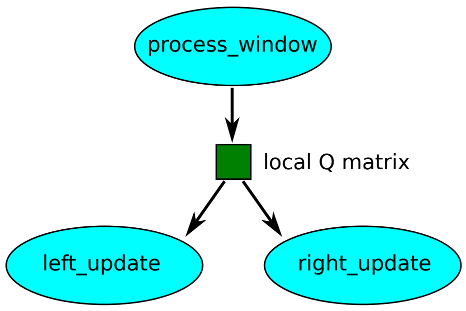


Figure 5: An illustration of how the local Q matrix flows from a `process_window` to the corresponding `left_update` and `right_update` tasks inducing implicit dependencies in its wake.

4.4 Dependencies

When an algorithm inserts a task, it must specify a codelet and a list of data handles. The same list of data handles acts as an argument list for the codelet when the task is later issued to a *worker* that executes the related codelet. `StarPU` uses this same argument list when it derives the data flow from one task to another. If two tasks are given the same data handle in their argument lists (i.e., they operate on the same tile), then depending on the order in which the tasks are inserted and various data access flags, an implicit data dependency may be induced between the tasks. For example, each inserted `process_window` task is given a handle to a newly created local Q matrix so that the task can write the accumulated transformation into the related memory buffer. The same handle is then fed to the corresponding `left_update` and `right_update` tasks. This induces implicit dependencies from the `process_window` task to the `left_update` and `right_update` tasks as shown in Figure 5. The algorithm may define a separate list of static arguments which is not taken into account in the data flow inferences. In addition, each task can be given a priority. The `default` priority is always 0. Minimum (`min`) priority is always smaller than or equal to `default` and maximum (`max`) priority is always larger that or equal to `default`. Larger value signals higher priority.

The actual data flow chart is much more complicated as shown in Figure 6. The `process_window` task accepts four data handles that together describe the diagonal window to be processed (highlighted in yellow). The `process_window` task processes the four tiles (the updated sections are highlighted in red) and outputs the local Q matrix (highlighted in dark green). Two of the modified tiles and the newly created local Q matrix are then fed to the `right_update` task together with four additional S matrix tiles. These four additional tiles describe the remaining part of the `right_update` task's computational area. The `right_update` task updates the six S matrix tiles (the updated sections are highlighted with in blue) and one of them is later fed to the `left_update` task. The `left_update` task also takes in one of the S matrix tiles processed by the `process_window` task, the local Q matrix, and six S matrix tiles describing the remaining part of the `left_update` task's

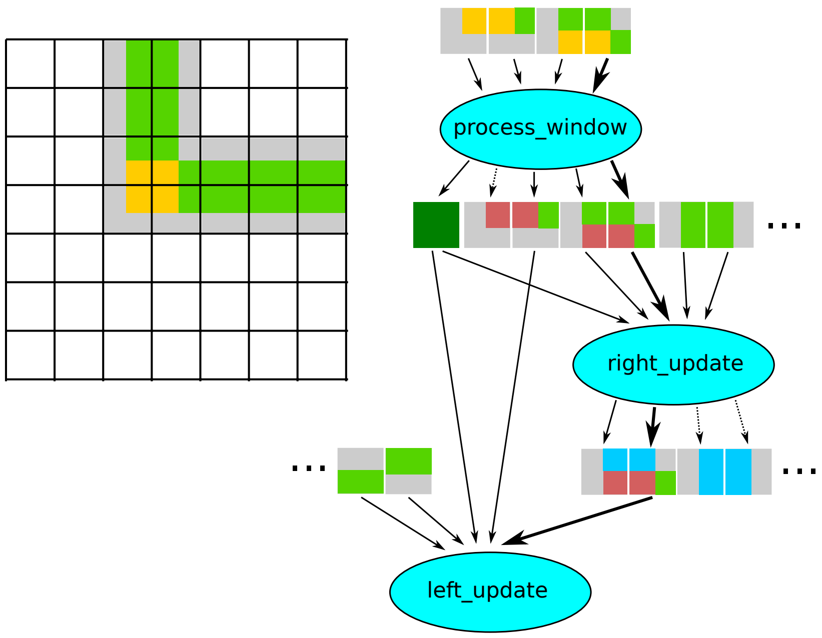


Figure 6: An example of how the S matrix data handles complicate the data flow chart.

computational area. The critical path is highlighted with thickened arrows.

It is vital to notice that one of the tiles introduces implicit dependency from the `right_update` task to the `left_update` task. These type of *spurious dependencies* are not easy to foresee in a general case and they may have a significant impact on performance if not handled correctly. In this particular case, many of these dependencies could be avoided by placing the diagonal windows such that their boundaries follow the boundaries of the underlying tiles. Thus, the tile that would otherwise induce a spurious dependency would no longer be shared between the `right_update` and the `left_update` tasks. If this type of window placement is not possible, then the adverse effects could at least be mitigated by splitting the updates into smaller update tasks such that length (time to execute) of the critical path (highlighted with thickened arrows in Figure 6) is reduced. For the sake of simplicity, we will ignore these type of dependencies in most the figures presented in the rest of this report.

Figure 7 shows a flow chart for a window chain that covers the whole diagonal. Note that we always draw the flow charts from the top down but the window chain itself is processed from bottom up. We have divided the updates into tasks such that each task does the minimal amount of work necessary to proceed to the next task in the graph. This is done to simplify the graph and in practice the task granularity can be much coarser. In addition, we have simplified the graph by leaving out the dependencies between the right and left updates as these dependencies are already force through the diagonal windows. Note that those tasks that the last window is dependent on are all located to the top right corner of the graph and are thus considered to be more important. In turn, all

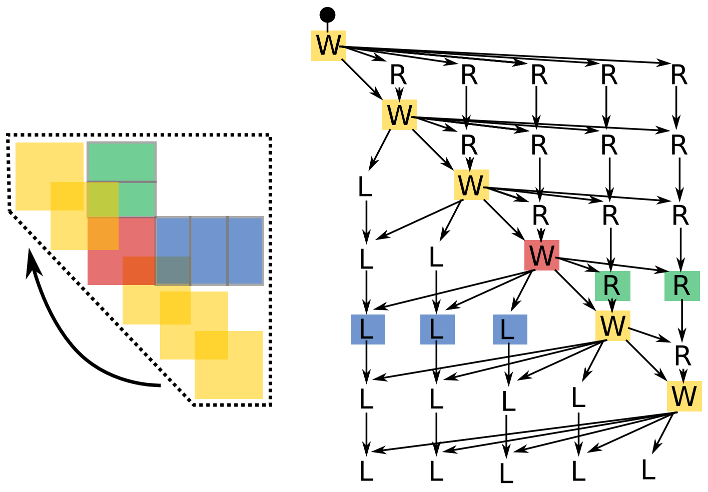


Figure 7: A data flow chart for a window chain that covers the whole diagonal. The diagonal windows are highlighted with yellow color. The left and right updates that correspond to the fourth window in the chain are highlighted with blue and green colors, respectively. The entry point to the data flow chart is marked with a black circle. The critical triangle is highlighted with dashed lines.

`left_update` tasks can be delayed until the very end because the diagonal windows do not depend on them. The section of the S matrix that is touched by the `process_window` and `right_update` tasks forms what we call a *critical triangle*. This critical triangle is a concept that we will now study in greater detail.

Figure 8 shows a similar data flow chart for a window chain that does not cover the whole diagonal. This figure introduces a new notation that will be used throughout the remaining part of this report. The left side of the figure shows the various regions of the S matrix that are being touched by the window chain (either by the `process_window` tasks or by the related update tasks). Note that the critical triangle is highlighted in dark red. The remaining left and right updates are highlighted with blue and green, respectively. The right side of Figure 8 shows a simplified data flow chart. The area highlighted in dark red corresponds to those tasks that the last window is dependent upon (cf. right side of Figure 7). As mentioned earlier, these `right_update` tasks are considered to be critical. The area highlighted in lighter red corresponds to those `left_update` tasks that operate on the critical triangle. The remaining `left_update` and `right_update` tasks are again highlighted in blue and green, respectively. Note the similarities between Figures 7 and 8.

The situation becomes more complicated when the reordering process involves multiple window chains and we want to increase the level of parallelism by processing these windows chains concurrently. Figure 9 shows the simplest case with two window chains. Note that the flow chart has two independent entry points highlighted with the black circles. It is clear that tasks operating on different critical triangles can be executed independently of each another. This, of course, is good for parallelism. However, some left updates from the upper windows chain overlap with some right updates from the lower window chain

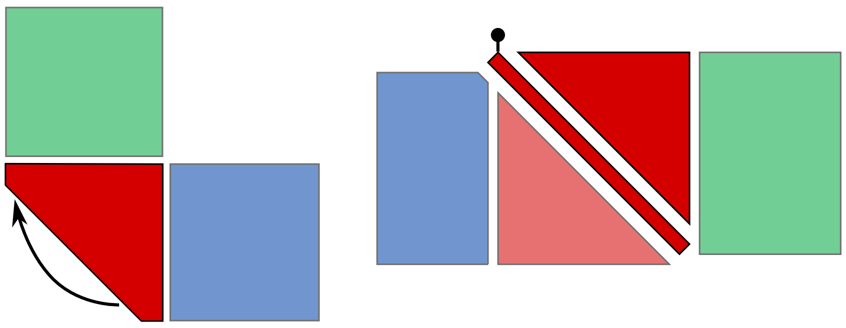


Figure 8: A simplified data flow chart for a window chain that does not cover the whole diagonal. Note that the entry point to the data flow chart is marked with a black circle.

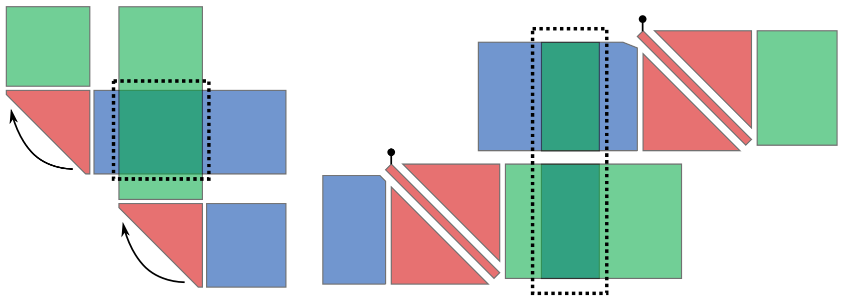


Figure 9: An illustration of how two non-overlapping window chains interact with each other.

as highlighted in the left side of Figure 9. This means that the order in which the task are inserted to `StarPU` becomes an important factor as it defines the order in which the overlapping area gets updated. In this situation, we can either apply the left updates from the upper window chain first and then apply the right updates from the lower window chain, or we can do the opposite. The important thing here is that the ordering stays consistent across all updates.

From the data flow perspective, this means that the corresponding data flow charts are not separate as drawn in right side of Figure 9. Instead, the areas highlighted with the dashed rectangle merge together. In this simple case, the merged sections are relatively small and the induced dependencies are highly localized. Thus, the situation does not necessarily require any sophisticated features from the runtime system. An algorithm could simply apply the right and left updates separately and synchronize after a set of independent updates had been made. For example, the parallel `PDTRSEN` (see Subsection 2.4) algorithm works roughly this way on individual window level.

The situation becomes even more complex when we extend our attention to cases where the critical triangles are allowed to overlap. Figure 10 shows an example of such a situation. We can distinguish six different cases where the computational areas of different task types

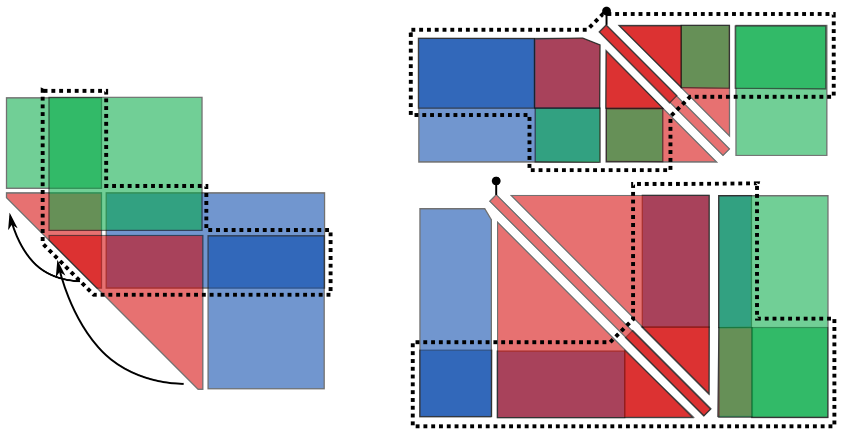


Figure 10: An illustration of how two overlapping window chains interact with each other.

from both window chains overlap. In particular, some tasks, which are related to the critical triangle of the lower windows chain, are being blocked by the left update tasks that are related to the upper window chain. This means that some left updates are actually more important that others. In turn, all (right) updates that are related to the upper window chain and operate sections above the lower window chain are not as important. The interactions between the data flow charts are again highlighted in the right side of Figure 10. Note how the two entry points are no longer independent of each other. We have now reached the point where the number of different cases is too large to handle by hand and more sophistication is expected from the runtime system. The question now becomes how to insert the tasks to `StarPU` in the most effective manner.

4.5 Task insertion

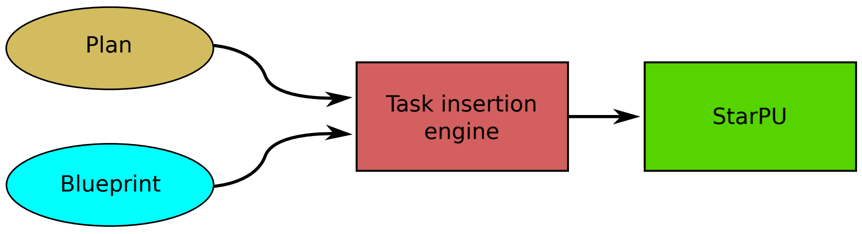


Figure 11: The three key components of our `StarPU` implementation: plan, blueprint and task insertion engine.

As pointed out in the previous subsection, the insertion of the tasks is not a trivial matter. For this reason, we adopted a modular approach where the process of task insertion is described in term of *plans*, *blueprints*, and a *task insertion engine* (see Figure 11):

Plan describes where the various diagonal windows are located and how they are connected to each others.

Blueprint describes the order in which the tasks are to be inserted. Each blueprint is implemented as an array of C enumerators / integers.

Task insertion engine interprets the plan and inserts the tasks to `StarPU` by following the provided blueprint.

This modular design allow us to experiment with various ideas and answer questions such as:

- In what order should we process the window chain? We could start from the topmost window chains as described in Algorithm 1 or we could start from the bottom window chain. The second approach might have some benefits as the bottom window chain is likely to be the longest one and we may want to start processing it as soon as possible.
- Should we delay some task? It might be a good idea to insert all high priority tasks first and only later insert less important tasks. On the other hand, the number of critical tasks that can be executed concurrently is not necessary large enough to saturate all workers. This might cause a situation where many workers are idling at the beginning.
- Should we pay attention to the critical path and the critical triangles?
- How should we assign priorities to tasks?

4.5.1 Planning phase

This section describes how our algorithm forms a plan that is later used to decide how the tasks are inserted to `StarPU`. The following four terms are used throughout this report:

Window describes a diagonal reordering window. Each window has an index number, a position and a size.

Window chain consists of multiple overlapping windows that are intended to be processed in a particular order. The related windows are stored in a doubly linked list.

Chain list consists of multiple window chains. The window chains can be processed in a particular order or independently of each other as long as windows that belong to different chains do not overlap. The window chains are stored in a doubly linked list.

Plan consists of multiple window lists that must be processed in a particular order. The chain lists are stored in a doubly linked list.

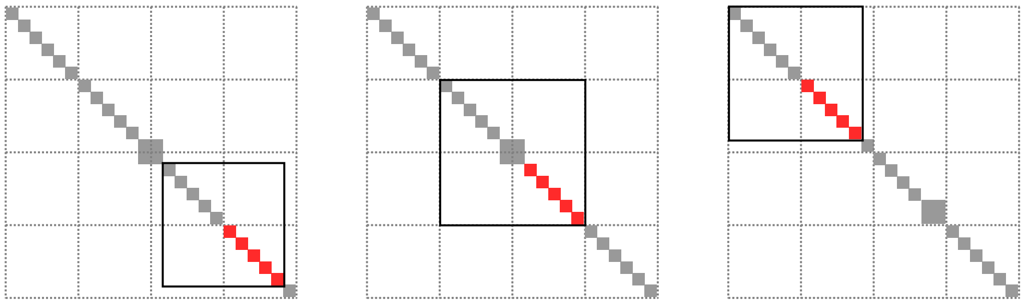


Figure 12: An example of how the windows are placed on the diagonal such that their upper left corners respect the boundaries of the underlying tiles. Note how the 2×2 block causes us to deviate from this rule and how all selected blocks always end up in the same tile.

Presently, our implementation supports two different ways of forming a plan:

Single-part plan contains single chain lists that is formed similarly as described in Section 3. The groups are selected such that all blocks in a group can be fitted inside a single tile when clustered together. For example, if the tile size is 64×64, then each group contains at most 63 eigenvalues[2]. We always try to place the windows such that their upper left corners follow the boundaries of the underlying tiles as shown in Figure 12. Therefore, the window size is not fixed, but grows as the window chain collects more and more (selected) blocks from the diagonal. The window size is explicitly limited to twice the size of the underlying tiles. This means that the top left corners of the windows that belong to the same window chain are always separated by a single tile. If necessary, each window is re-sized to avoid splitting any 2×2 blocks. However, the re-sizing is done in such a way that the number of tiles involved with each `process_window` task is at most four and all selected blocks always end up in the same tile. Thus, each diagonal tile is touched at most twice by each window chain. All this is done to reduce the number of spurious dependencies that might be otherwise induced if the windows were placed more freely. These choices also reduce the amount of data that needs to be transferred across the `MPI` node boundaries.

Multi-part plan is derived from a single-part plan by splitting the associated chain list into multiple chains lists as illustrated in Figure 13. The process is described in Algorithm 2. Note that the chains that belong to the same chain list in Figure 13 are independent from each other, that is, their critical triangles do not overlap. Thus, we have effectively transformed a situation of the type shown in Figure 10, into a situation of the type shown in Figure 9. However, the chain lists must be processed

[2]Each group should contain strictly less than tile size eigenvalues. See Figure 12. If the group contained six selected blocks instead of the five shown in the figure, then the 2×2 block in the center of the matrix would cause a problem.

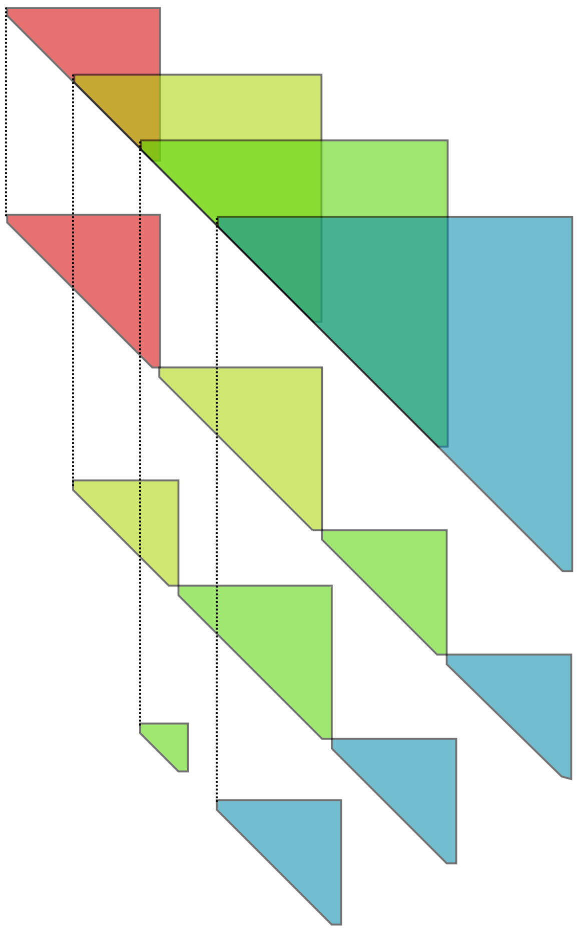


Figure 13: An illustration of how a single chain list containing four overlapping chains is divided into three non-overlapping chains lists.

Algorithm 2: Formation of a multi-part plan

```
Data: A real Schur form $S$ and a Boolean array that defines the selected subset of
      the blocks along the diagonal of $S$.
Result: A multi-part plan plan for $S$ and the given Boolean array.
1  Form a template chain list (a single-part plan) using $S$ and the given Boolean array;
2  Form an empty plan plan;
3  while the template chain list is not empty do
4      Set end := top left corner of the matrix - 1;
5      Form an empty chain list list;
6      for window chains in the template chain list from the top down do
7          Form a empty window chain chain;
8          Remove those windows from the window chain whose top left corners are
           located below end and add them to chain;
9          if chain is not empty then
10             Add chain to list;
11             Set end := bottom right corner of the bottom window in chain;
12         if the window chain is empty then
13             Remove the window chain from the template chain list;
14     if list is not empty then
15         Add list to plan;
```

in the order they were added to the multi-part plan in step 15 of Algorithm 2. Step 8 of Algorithm 2 is in reality slightly more complicated as the algorithm tries to avoid situations where two window chains in a same chain list would touch a common tile. A shared tile would induce additional spurious dependencies.

A multi-part plan is not suitable for all situations. For example, if a large proportion of the diagonal blocks are selected, then each chain list would contain no more than a few window chains. The same thing happens when the selected blocks are clustered together. In an extreme case, each chain list would contain only one window chain and each window chain would contain only one window. That is, the resulting multi-part plan would be, in essence, equivalent with the original single-part plan.

4.5.2 Blueprints

This section describes a few of the various blueprints that are currently implemented.

One-pass forward blueprint leads to Algorithm 1 when the blueprint combined with a single-part plan. In this configuration, our algorithm relies entirely on `StarPU` to deduce all data dependencies. The main purpose of this blueprint is to serve as a comparison for more advanced blueprints. The `process_window` tasks are always

Algorithm 3: One-pass forward blueprint

```
1 for chain lists in the plan do
2     for window chains in the chain list from the top down do
3         for windows in the window chain from the bottom up do
4             Insert the process_window task with priority max;
5             Insert the corresponding right_update tasks with the priority
              $\max($default,max$ - 1)$;
6             Insert the corresponding left_update tasks with the priority default;
7             Insert the corresponding right_update tasks that update the Schur basis
              $Q$ with the priority min;
```

inserted with the highest priority because they form a part of the algorithm's critical path. The `right_update` tasks are given the second highest priority, because as shown in Figure 7, the last window in the window chain depends upon some of these right updates[3]. The `left_update` tasks are given higher priority than those `right_update` tasks that update the Schur form Q as some left updates may prevent a window chain below the current window chain from proceeding upwards. See Algorithm 3 for more detailed description. Note that the blueprint does not have any input data or any result. This is because each blueprint is simply a data structure that gets interpreted by the tasks insertion engine.

Algorithm 4: Two-pass backward blueprint

```
1 for chain lists in the plan do
2     for window chains in the chain list from the bottom up do
3         for windows in the window chain from the bottom up do
4             Insert the process_window task with priority max;
5             Insert the corresponding right_update tasks with the priority
              $\max($default,max$ - 1)$;
6     for window chains in the chain list from the bottom up do
7         for windows in the window chain from the bottom up do
8             Insert the corresponding left_update tasks with the priority default;
9             Insert the corresponding right_update tasks that update the Schur basis
              $Q$ with the priority min;
```

Two-pass backward blueprint differs from the one-pass forward blueprint in two important ways: the window chains are processed in a reverse order and the window

[3]The last window depends upon those right updates that operate on the critical triangle. Other right updates are less important.

chains are processed in two separate phases. Both of these features require a multi-part plan. The main purpose of this blueprint is to investigate the idea of inserting the window chains in reverse order. See Algorithm 4 for more detailed description.

Algorithm 5: One-pass forward chained blueprint

```
1 for chain lists in the plan do
2     for window chains in the chain list from the top down do
3         Insert those process_window and right_update tasks that modify the
          window chain's critical triangle;
4         Insert all corresponding left_update tasks;
5         Insert the remaining (low priority) right_update tasks;
6         Insert all right_update tasks that update the Schur basis $Q$;
```

One-pass forward chained blueprint uses a completely different approach where the critical triangle is inserted first and then followed by the related `left_update` tasks. The exact details of how the tasks related to the critical triangle are inserted are given in the next subsection. See Algorithm 5 for more detailed description.

Algorithm 6: One-pass backward chained blueprint

```
1 for chain lists in the plan do
2     for window chains in the chain list from the bottom up do
3         Insert those process_window and right_update tasks that modify the
          window chain's critical triangle;
4         Insert the remaining right_update tasks;
5         Insert all corresponding left_update tasks;
6         Insert all right_update tasks that update the Schur basis $Q$;
```

One-pass backward chained blueprint differs from the one-pass forward chained blueprint in that the window chains are processed in a reverse order. Thus, it is very similar to the two-pass backward blueprint and requires a multi-part plan. See Algorithm 6 for more detailed description.

4.5.3 Task insertion engine

The blueprints described in the previous subsection left open the question of how the tasks are actually inserted into `StarPU`. This subsection aims to answer this question.

The way how the left and right updates are divided into corresponding update tasks is one of the deciding factor for the performance of the algorithm. We want to achieve two objectives. Firstly, we want to guarantee that each `process_window` task induces enough update tasks to saturate all workers threads. Secondly, we want to avoid introducing

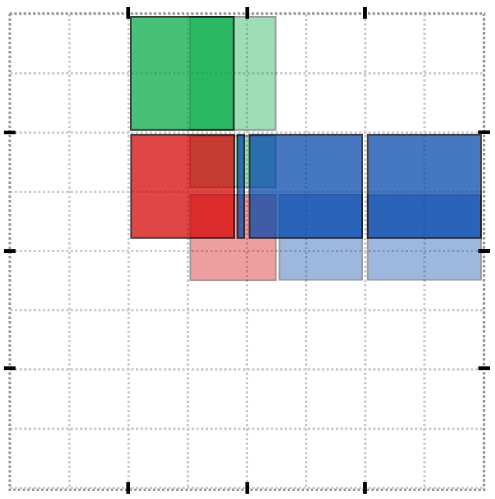


Figure 14: An illustration of how updates are cut into tasks by using a 4×4 stencil. The stencil points where the cutting is performed are marked along the edges of the matrix.

spurious data dependencies. We achieve these objectives by forming a two-dimensional stencil that divides the matrices both vertically and horizontally into P sections such that the section size is a multiple of the tile size. Above, P is the number of worker threads. This stencil cuts the right and left updates into tasks in vertical and horizontal directions, respectively, as shown in Figure 14 with $P = 4$. Note how there are no vertical dependencies between `right_update` tasks and no horizontal dependencies between `left_update` tasks. If the implementation is being executed in a distributed memory environment, then special care is taken to make sure that the stencil follows the boundaries of the underlying sections (see Subsection 4.1).

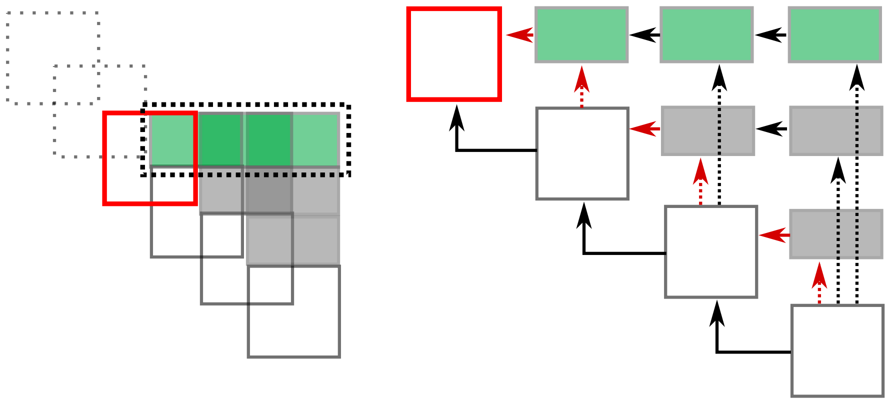


Figure 15: An illustration of how `process_window` and `right_update` tasks that modify the window chain's critical triangle are inserted.

Right updates that modify the window chain's critical triangle in the chained blueprints are treated in a slightly different manner as shown in Figure 15. Diagonal windows are processed in order starting from the window in the bottom right corner The current diagonal window is highlighted with red boundary. The rectangles highlighted with green are the right updates which the current diagonal window depends upon. The preceding right updates (highlighted with gray) have already been inserted. The task insertion engine inserts the corresponding `right_update` tasks in order from right to left followed by the `process_window` task. The `process_window` tasks are inserted with priority `max` and `right_update` tasks with priority max(`default`, `max` − 1). The computational areas of the `right_update` tasks are always "rounded" upwards such that their upper edges follow the boundaries of the underlying tiles.

Our objective is to minimize the length of the critical path. What we mean by this can be better understood by investigating the right side of Figure 15. The dotted arrows show the dependencies between the preceding diagonal windows and the current right updates. The solid arrows show the remaining dependencies between the right updates and the diagonal windows. Since the `process_window` tasks are generally more expensive that equally sized update tasks, the critical path is likely to be the one highlighted with red arrows. We have divided the right updates into tasks such that each task does the minimal amount of work necessary to proceed to the next task in the graph in an attempt to reduce the length of the critical path.

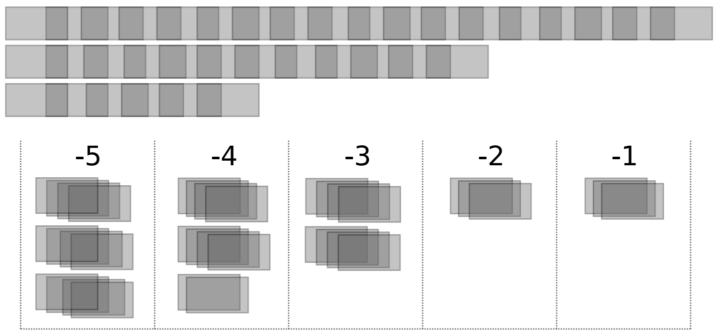


Figure 16: An illustration of how low priority update tasks are divided into priority groups. In this example, we are using the priority range $[-5, 0[$. The longest update chain (on the top) determines the number of update tasks assigned to each priority group. The groups are filled in order, starting from the lower priority.

As the one-pass forward chained blueprint (see Algorithm 5) is designed to be used with a single-part plan, all `right_update` tasks that operate sections above the critical triangle are considered to be low priority tasks. The same applies to `right_update` tasks that operate the Schur form Q in the other chained blueprints. However, simply assigning priority `min` to these tasks can cause problems because when a window chain is inserted to `StarPU`, the related update tasks form chains where each update is always dependent on the previous update. These update chains can have varying lengths which may cause load balancing issues. We have reserved the priority range [`min`, `default`[for these low priority tasks. Our algorithm locates the longest update chain in the plan and uses its length as a basis for dividing all low priority update tasks into priority groups as shown in Figure 16. The chain length information is preserved when a single-part plan is transformed into a multi-part plan. This priority assignment scheme aims to force StarPU to process the update chains at an even rate.

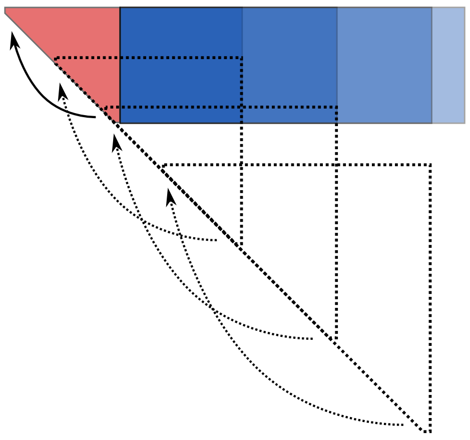


Figure 17: An illustration of how the left update tasks are divided into priority groups. Darker color signals higher priority.

As noted in Section 4.4, it is possible that some `left_update` tasks may prevent other window chains from proceeding. Figure 17 shows an example where `left_update` tasks, which correspond to the topmost window chain, could potentially prevent the two window chain below it from proceeding. We are using the priority range $[\texttt{default}, \max(\texttt{default}, \texttt{max}-1)[$ for assigning priorities for `left_update` tasks based on their importance when dealing with chained blueprints. For each window chain in a chain list, our algorithm looks downwards in the chain list and goes through all window chains below the current window chain. The right side edges of the critical triangles of these window chains are used as a stencil for grouping the `left_update` into priority groups. The priorities are assigned in descending order from left to right.

The `right_update` tasks in the one-pass backward chained blueprint (see Algorithm 6) are processed similarly. We are using the priority range $[\texttt{default}, \max(\texttt{default}, \texttt{max}-1)[$ for assigning priorities for `right_update` tasks. For each window chain in a chain list, our algorithm looks upward in the chain list and goes through all window chains above the current window chain. The upper edges of critical triangles of these window chains are used as a stencil for separating the `right_update` into priority groups. The priorities are assigned in descending order from the bottom up. The process is visualized in Figure 18.

4.6 Tunable parameters

The implementation has several tunable parameters:

- Plan and blueprint (more blueprint can be added with minimal effort)

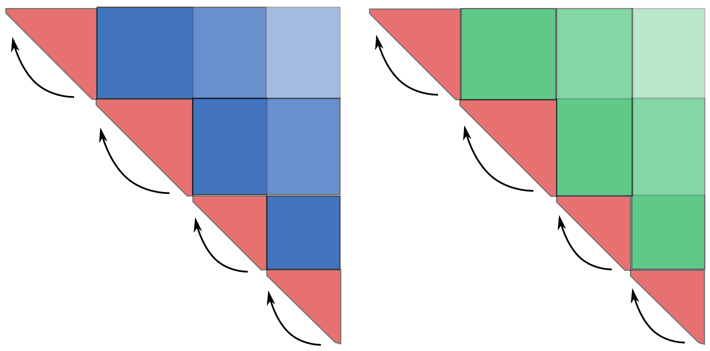


Figure 18: An illustration of how left (on the left) and the right (on the left) update tasks are divided into priority groups. Darker color signals higher priority.

- Tile size
- Section size (`MPI` specific parameter)
- Window size (can be set explicitly)
- Group size (can be set explicitly)

In addition, the following parameters are currently hard-coded but could be exposed to the user in future:

- Size of the stencil discussed in Subsection 4.5.3 (see Figure 14)
- Window size used inside the `process_window` codelet (hard-coded to 128×128)
- Size of the largest window to be reordered in scalar manner (hard-coded to 256×256)

If the window size is defined explicitly, then the implementation does not take into account the underlying tile structure when it is placing the diagonal windows. We have not found any evidence suggesting that the window size should be defined explicitly. In addition, numerical experiments have shown that the number of selected eigenvalues in a group should be half of the window size. Thus, these two parameters should be left to their default values in most use cases.

5 Computational experiments

In this section we present computational results pertaining to reordering eigenvalues of matrices in real Schur form using a shared memory machine and double precision arithmetic. Our implementation can be configured to run on a distributed memory machine using `StarPU`, but while the efficiency has improved recently, it is still quite low. In addition, we are currently working on fixing certain technical problems related to processing larger matrices.

5.1 Computer system

All experiments where executed on a system called `Kebnekaise`, which is located in the High Performance Computing Center North (HPC2N) at Umeå University[4]. Each compute node contains 28 Intel Xeon E5-2690v4 cores organized into 2 NUMA islands with 14 cores in each. The nodes are connected with a FDR Infiniband Network. Each CPU core has 32 KiB L1 data cache, 32 KiB L1 instruction cache and 256 KiB L2 cache. Moreover, for every NUMA island there is 35 MiB of shared L3 cache. The total amount of RAM per compute node is 128 GiB. A schematic representation of a compute node is as seen in Figure 19.

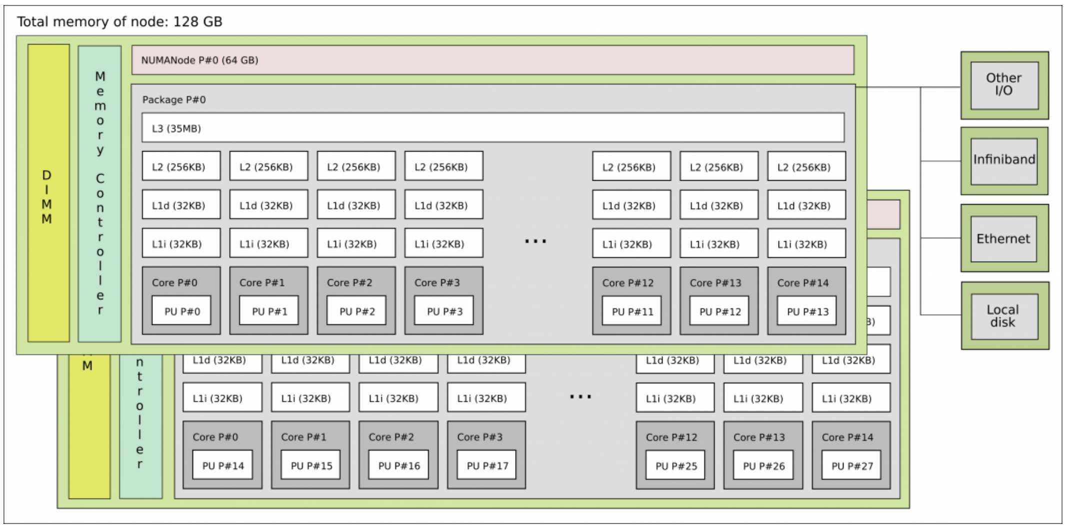


Figure 19: A schematic representation of a compute node on the Kebnekaise system.

5.2 Test matrices

We designed a matrix generator which can build a complete experiment from a single random seed and a small set of parameters (n, k, p). Here n is the dimension of the matrix A, k is the number of diagonal 2×2 blocks and p is the probability that the user selects a given diagonal block. The generator is available from the authors on demand.

5.2.1 Construction of the Schur basis Q

The orthogonal matrix Q is generated as a single Householder reflector $I - vv^T$. The vector v is generated from the seed.

[4]See `https://www.hpc2n.umu.se/resources/hardware/kebnekaise`.

5.2.2 Construction of the Schur form S

The main problems are to choose a location for each 2×2 block along the diagonal of S and to assign values to the nonzero entries of S. The locations of the 2×2 blocks are determined as follows. The $m = n - 2k$ real values must be interleaved with the 2×2 blocks. It follows, that the 1×1 blocks will form at most $k + 1$ groups. By choosing m numbers from the set $\{0, 1, 2, \ldots, k\}$ we can assign each 1×1 to exactly one group. Afterwards, we can count the number of 1×1 block in each group. These counts uniquely determine the location of all the 2×2 blocks.

The complex eigenvalues are chosen at random from a discrete grid of points in the upper half of the complex plane. This ensures all diagonal blocks can be swapped without difficulty, because the tiny Sylvester equations in the swapping kernels are well conditioned. The off diagonal entries of S are chosen at random, uniformly distributed in the interval $[0, 1)$.

5.2.3 Diagonal block selection process

There are k pairs of complex conjugate eigenvalues (one pair per 2×2 block) and $m = n-2k$ real eigenvalues (1×1 blocks). The user chooses each of the $n - k$ diagonal blocks with probability p. On average, the user chooses $2kp$ blocks of size 2×2 blocks and mp blocks of size 1×1 blocks. Therefore, the total number of selected eigenvalues is on average $2kp + mp = np$.

5.3 Experimental methodology

Each experiment was repeated several times. The runtimes were saved and the median was computed. Compared with the average, the median is much less sensitive to the effect of outliers. Problems were generated from a random seed using the parameters listed below:

1. Matrix dimension $n \in \{10000, 20000, 30000, 40000\}$.
2. Number of 2×2 diagonal blocks $k = n/4$, that is, half of the eigenvalues belong to complex conjugate pairs.
3. The user's chance of choosing a specific diagonal block $p \in \{0.05, 0.15, 0.35, 0.50\}$.

Parallel experiments with P cores were always executed using cores 0 through $P - 1$.

5.4 Accuracy

Result are worthless if they are not accurate. With respect to all `StarPU` experiments reported in this paper the following statements hold true:

1. For each eigenvalue $\hat{\lambda}$ in the updated Schur form $\hat{S}$, we have the relative error bound

$$\frac{|\lambda - \hat{\lambda}|}{|\lambda|} \lesssim 900u. \qquad (2)$$

where λ is the original value of the eigenvalue.

2. For each reordered Schur decomposition $\hat{A} = \hat{Q}\hat{S}\hat{Q}^T$ we have a relative backward error bound
$$\frac{\|A - \hat{A}\|_F}{\|A\|_F} \lesssim 190u. \tag{3}$$
where $A = QSQ^T$ is the original matrix.

3. For each new Schur basis $\hat{Q}$, we have
$$\frac{\|\hat{Q}^T\hat{Q} - I\|_F}{\|I\|_F} \lesssim 315u, \tag{4}$$
which shows that the orthogonality is very nearly preserved.

Above, $u = 2^{-52}$ is the machine epsilon. In exact arithmetic, the errors should be zero. The small values that we have obtained merely serve to illustrate the our `StarPU` implementation does not contain any obvious errors and that our test problems are well conditioned.

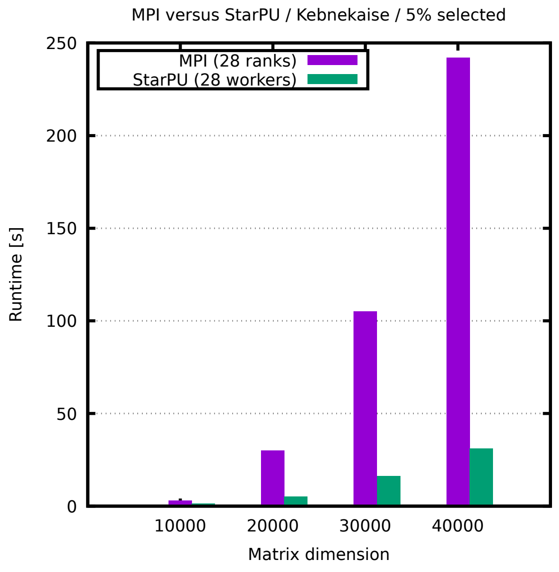
5.5 Time to solve

We report on the speed of our `StarPU` implementation relative to the existing `ScaLAPACK` routine `PDTRSEN`. We compare the case of 28 `MPI` ranks to the case of 28 `StarPU` workers, i.e. full utilization of a single node on `Kebnekaise`. The results are illustrated in Figure 20. Across all performed experiments, the `StarPU` implementation is between 1.3 and 7.8 times faster that the `PDTRSEN` subroutine. On average, the `StarPU` implementation is 2.9 times faster. The performance difference increases when the matrix dimension and/or the CPU core count are increased. In addition, the StarPU implementation appears to perform much better when a fewer number of diagonal blocks are selected.

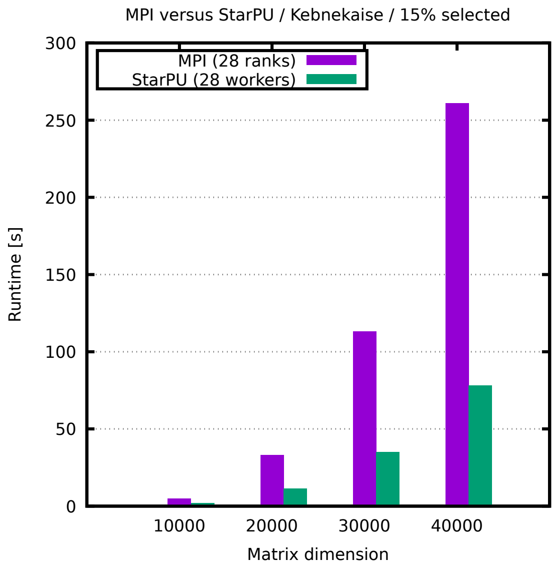
5.6 Sequential execution

We report on the time T_s to solve problems using single-threaded code and compare our code to the sequential blocked code BDTRSEN. This is an important measurement, because it allows us to determine the best sequential code and establish a baseline against which all parallel speedups can be computed. Our results are shown in Figure 21. We note the paradoxical result that our `StarPU` implementation which uses `BDTRSEN` as a kernel, is somewhat faster than `BDTRSEN` running alone. It remains an open problem to fully explain this situation, but the fact that the window size can be chosen more freely in our implementations is probably a major factor[5]. This allows us to have larger update tasks which causes fewer and larger DGEMM operations, hence a higher flop rate.

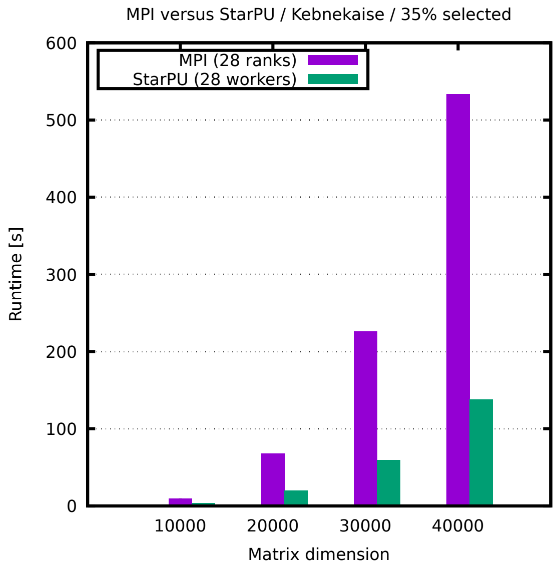
[5]See Subsection 5.10 for a more in depth explanation.



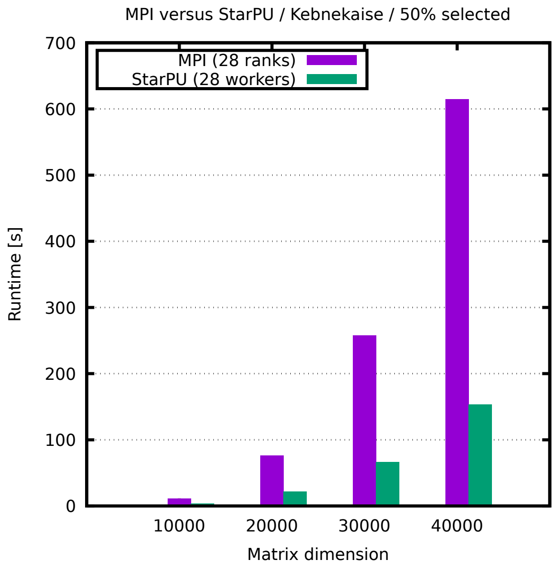
(a) 5% selected



(b) 15% selected

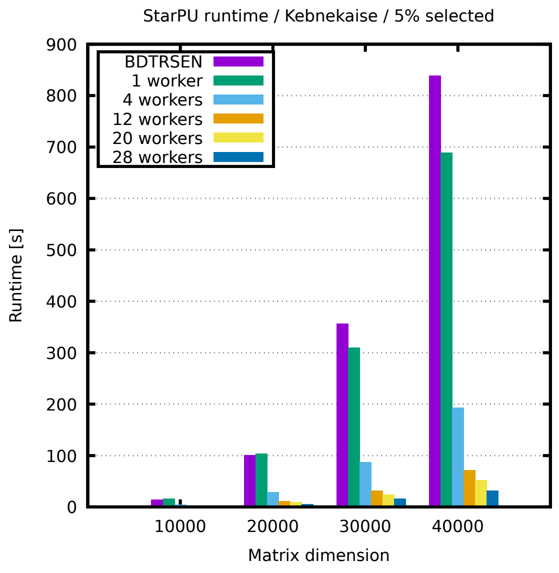


(c) 35% selected

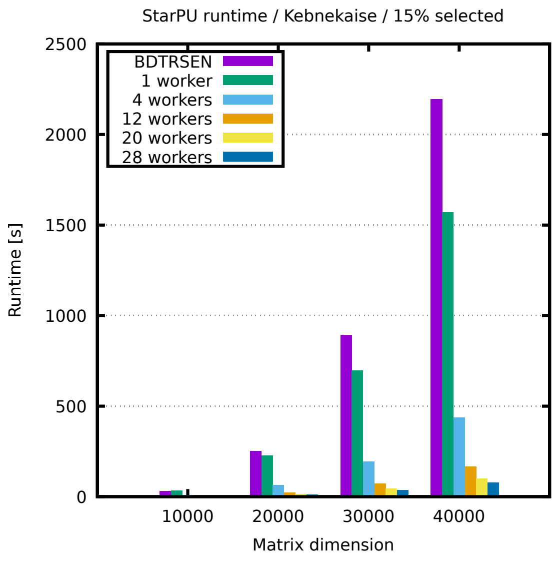


(d) 50% selected

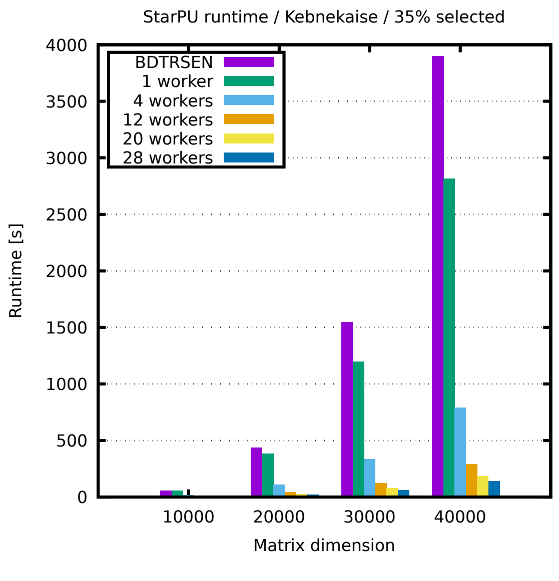
Figure 20: Comparison of the runtime for `PDTRSEN` from `ScaLAPACK` and our new `StarPU` implementation. The number of selected diagonal blocks relative to the matrix size runs through the values 5, 15, 35, and 50 percent. Observe that the scale on the y-axis varies between sub-figures.



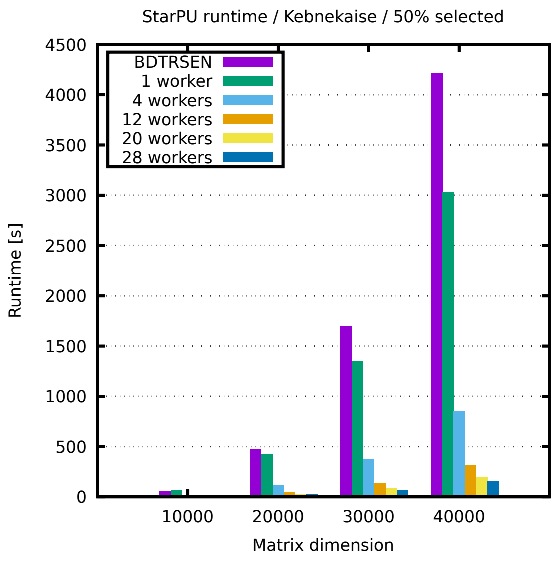
(a) 5% selected



(b) 15% selected



(c) 35% selected



(d) 50% selected

Figure 21: The runtime of the sequential code `BDTRSEN` and our `StarPU` implementation. The number of selected diagonal blocks relative to the matrix size runs through the values 5, 15, 35, and 50 percent. Observe that the scale on the y-axis varies between sub-figures.

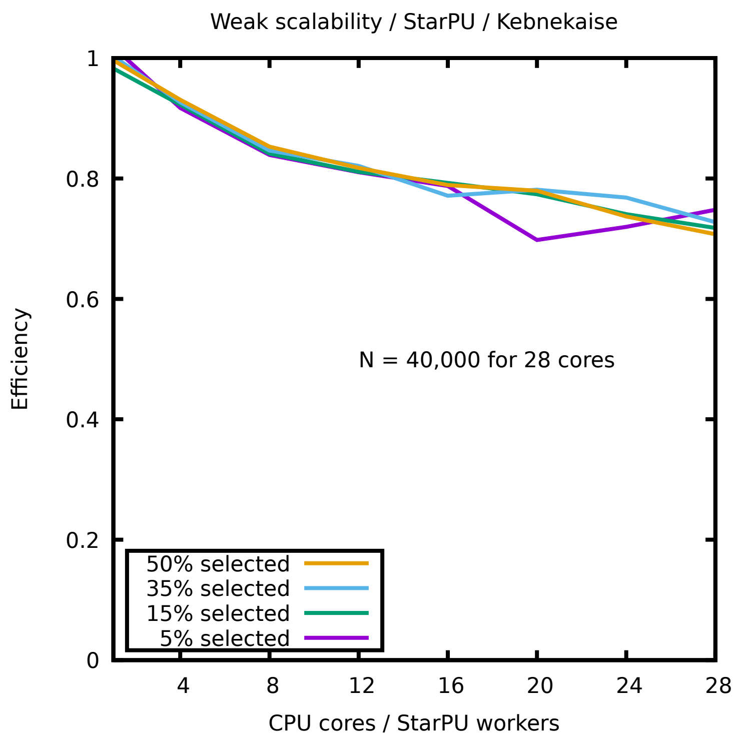


Figure 22: Weak scalability of our `StarPU` implementation.

5.7 Scalability

The scalability of a program measures its response to an increase in the number of processors. In the context of high performance computing we are interested in weak and strong scalability. In both case we report on the parallel efficiency ρ given by

$$\rho = \frac{T_s}{PT_P}, \tag{5}$$

where T_s is the serial execution time and T_P is the parallel execution time using P cores. We devote one section to each concept.

5.7.1 Weak scalability

Weak scalability refers to the situation where the problem size per processor is constant as the number of processors is increased. Here we report on the weak scalability efficiency obtained by scaling our largest problem ($n = 40,000$) from $P = 28$ down to $P = 1$ cores. Our results are plotted in Figure 22. We are pleased to report that the efficiencies are well above 60%. Except for some minor bumps the curves are monotone decreasing exactly as one would expect.

5.7.2 Strong scalability

Strong scalability refers to the situation where the problem size is constant as the number of processors is increased. Here we report on strong scalability efficiency. Our results are given in Figure 23. Ideally, one would like to obtain curves which are monotone and slowly decreasing. This is very nearly true for our experiments with a noteworthy exception being represented by the cases of 5% and 15% percent selected diagonal blocks. In general, shorter runs are more sensitive to disruptions beyond our control, i.e. intervention by the operating system, so we are not surprised to record bumps when the computational load is rather light.

5.8 Flop-rate

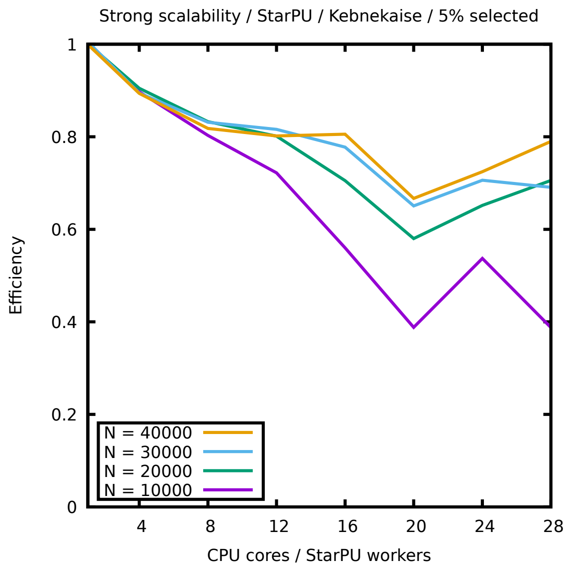
The flop rate should be measured relative to the theoretical peak flop rate which depends on the clock frequency, the core count, the width of the SIMD registers and the number of instructions a core can retire per cycle. The clock frequency depends on the number of active cores per NUMA island. On `Kebnekaise` the AVX base frequency is 2.1 GHz and the boost frequencies are given in Table 1. Consider a single NUMA island. If only a single core is active, then the boost is $14 \cdot 100$ MHz $= 1.4$ GHz and the clock frequency is therefore 3.5 GHz. If all 14 cores are active, then the boost is only $8 \cdot 100$ MHz and the clock frequency is 2.9 GHz for each core. The cores support the AVX2 instruction set and have 256 bit vector registers. This corresponds to a vector length of 4 double precision numbers. The cores can retire 2 fused multiply–add instructions per cycle. The theoretical flop rate is therefore 16 flops per cycle for each core (two flops per instruction). The peak flop rate for a full node (2 NUMA islands) is therefore

$$\texttt{28} \cdot \texttt{16 flops/cycle} \cdot \texttt{2.9 gigacycles/s = 1299 Gflops/s}.$$

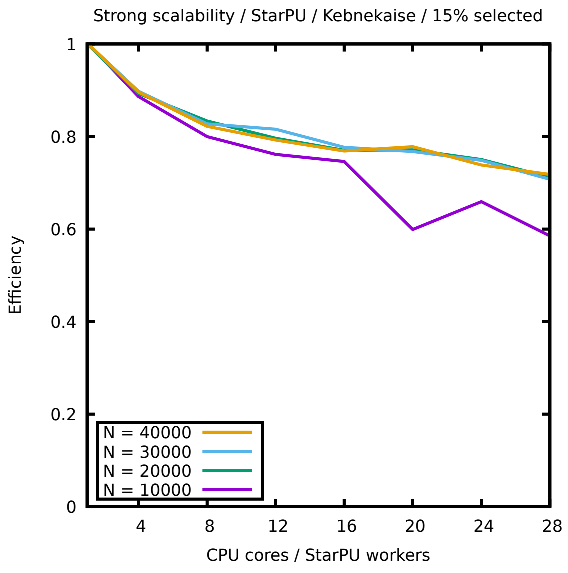
active cores	1	2	3	4	5	6	7	8	9	10	11	12	13	14
boost	14	14	12	11	10	9	8	8	8	8	8	8	8	8

Table 1: AVX2 boost table for compute nodes on `Kebnekaise`. The boost is given in multiples of 100 MHz. The boost is a decreasing function of the number of active cores, reflecting the difficulty of providing adequate power and cooling when the node is run at full capacity.

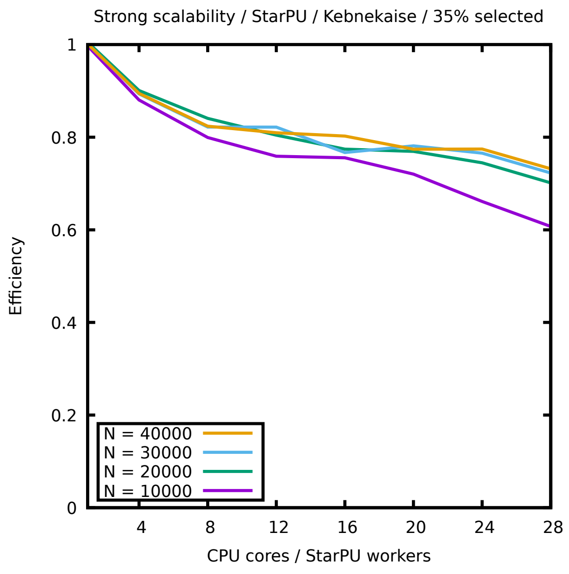
Our results are presented in Figure 24. The curves represent *lower* bounds on our relative flop rate, as we have only included the flops performed during the row and column operations, i.e. matrix-matrix multiplications. The flops performed within each window are more difficult to count and have not been included. We are pleased to note that the lower bound for the relative flop rate is well above 50% for all but the smallest matrices and the lowest number of selected diagonal blocks. Ideally, one would expect to obtain curves which are monotone decreasing. In practice, the occasional bumps are unavoidable



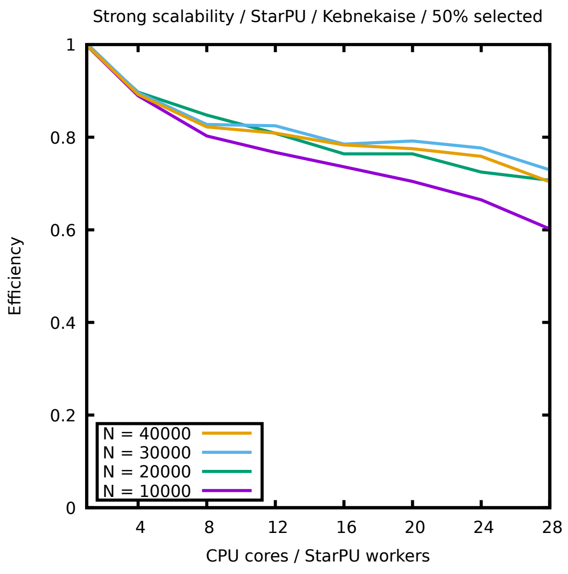
(a) 5% selected



(b) 15% selected



(c) 35% selected



(d) 50% selected

Figure 23: The strong scalability efficiency of our `StarPU` implementation. The number of selected diagonal blocks relative to the matrix size runs through the values 5, 15, 35, and 50 percent.

and not necessarily reproducible from one day to the next. In particular, it remains an open problem to explain the bumps which are evident in Figure 24a and Figure 24b.

5.9 Idle time and overhead

Worker threads executing code under a runtime system such as `StarPU` are either busy doing useful work, waiting for tasks to become available (idle time) or running the system itself (parallel overhead). Ideally, we want all cores to be fully engaged with the main calculation at all times, but this is of course not possible in general. `StarPU` has the ability to record the time spent in each activity. In addition, we included `StarPU` startup and shutdown times to the reported overhead. Here we report on the idle time and the overhead as a fraction of the total execution time. Our results are presented in Figure 25. In general, the impression is favorable and the workers are almost always moving the computation along, especially when the problem size is large and a large fraction of diagonal blocks has been selected. If we omit the smallest problem size and the case of 5% selected diagonal blocks, then all workers are devoting more than 95% of their time to advancing the computation. There is significant amount of time lost to idling when only 5% are selected and the worker count is high. This is hardly surprising as we are trying to schedule a small load across a large number of workers.

5.10 Tunability

Based on our experiences, the tile size appears to be the most important parameter. It has implications both for the data layout and the dependence tracking in `StarPU`. In particular, if the window and group sizes are left undefined, then the tile size defines the task granularity. Preliminary parameter value sweeps suggest that the optimal tile size depends linearly on the matrix dimension. Thus, we fitted a linear regression model to our sweep data. The resulting decision function is

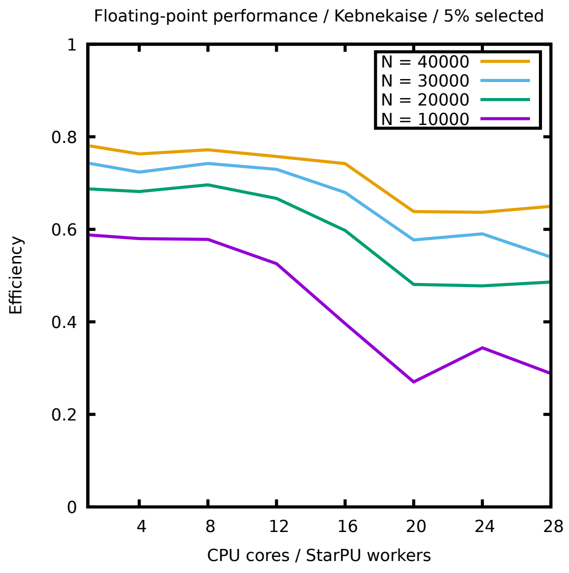
$$\tilde{b}(n) = \left\lceil \frac{14}{625} n + \frac{184}{5} \right\rceil_8, \tag{6}$$

where $\lceil \cdot \rceil_8$ rounds upwards to the next multiple of 8. This function gives a near optimal tile size for a given matrix dimension n. In practice, we ended limiting the tile size to

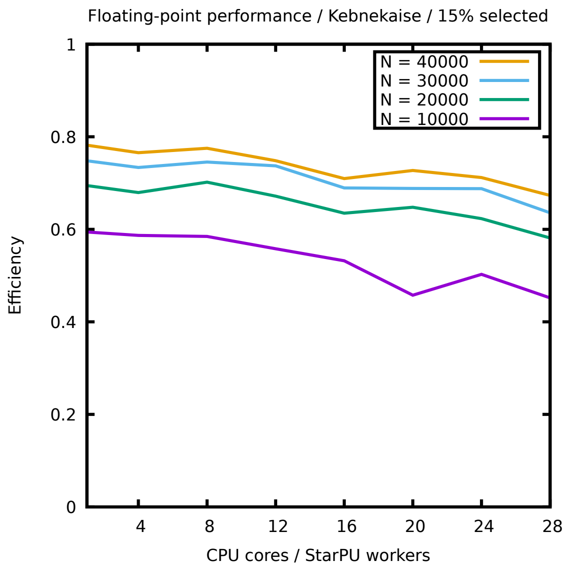
$$b(n,P) = \max\left(64, \min\left(\tilde{b}(n), \left\lceil \frac{n}{2P} \right\rceil_8\right)\right), \tag{7}$$

where P is the worker count. The upper limit guarantees that each diagonal window induces enough update tasks to saturate all workers.

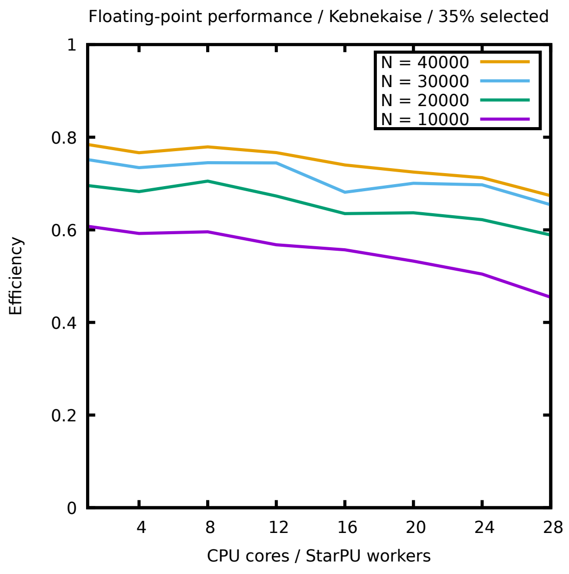
These preliminary parameter value sweeps also motivated us to experiment with the idea of reordering the diagonal windows in blocked manner. In the beginning, we noticed that if the diagonal windows are reordered in a scalar manner, then the connection between the matrix dimension and the optimal tile size is not linear. Instead, the optimal tile size appeared to have an upper limit. This would have limited us to using smaller tile size



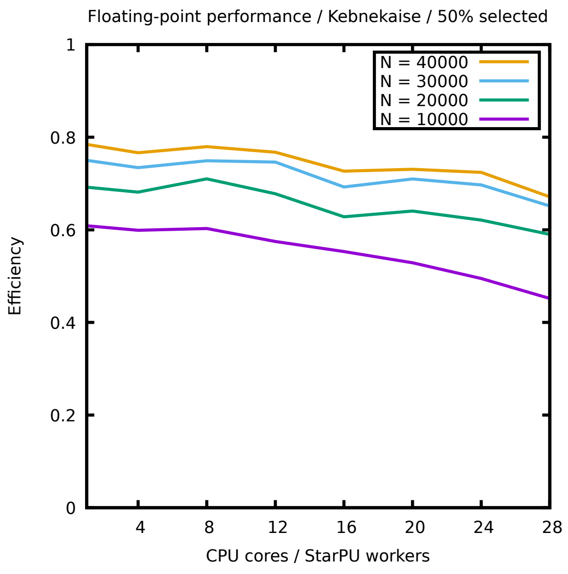
(a) 5% selected



(b) 15% selected

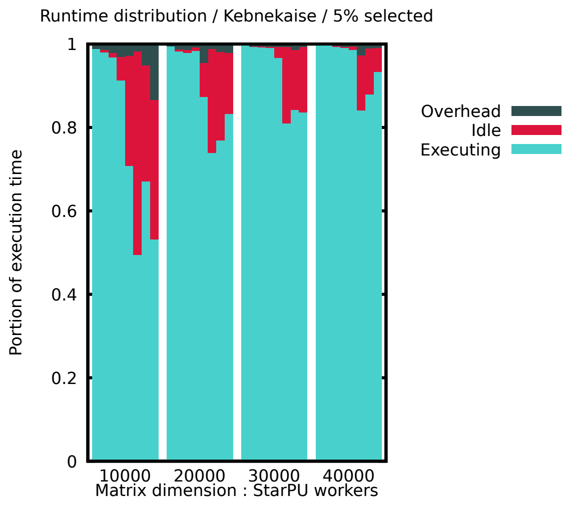


(c) 35% selected

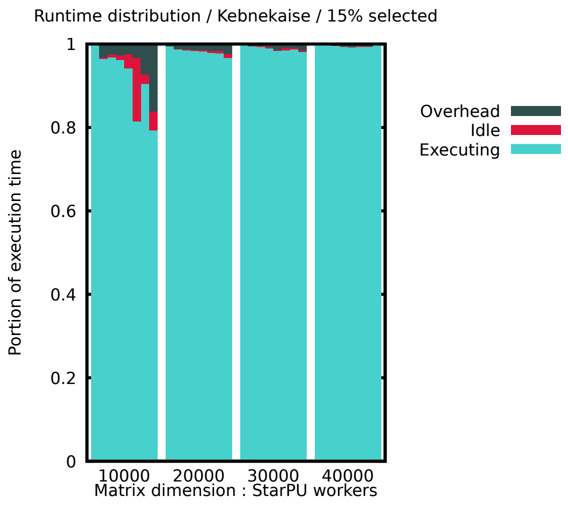


(d) 50% selected

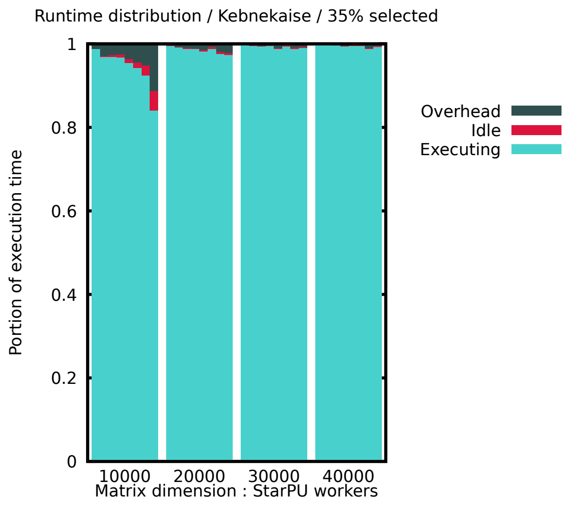
Figure 24: A lower bound for the flop rate of our `StarPU` implementation relative to the peak flop rate for the specific selection of cores. The number of selected diagonal blocks relative to the matrix size runs through the values 5, 15, 35, and 50 percent.



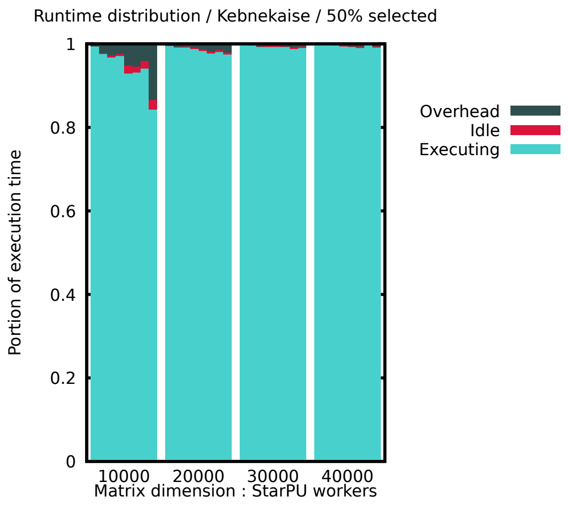
(a) 5% selected



(b) 15% selected



(c) 35% selected



(d) 50% selected

Figure 25: The idle time and the overhead of our `StarPU` implementation relative to the total runtime. Each sub-figure corresponds to a specific fraction of the diagonal blocks, specifically, 5, 15, 35, and 50 percent of the total. Within each figure, the results are grouped by matrix dimension. Within each group, the `StarPU` worker count runs through the numbers 1, 4, 8, 12, 16, 20, 24, and 28 as we move left to right.

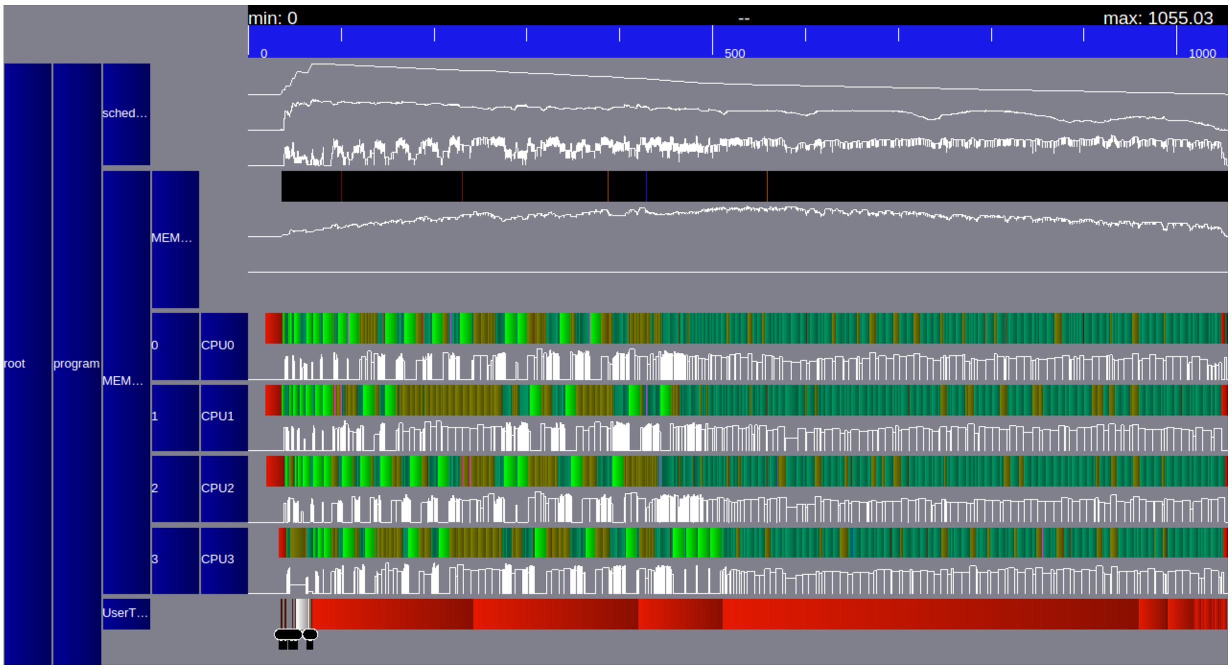


Figure 26: An example of a trace plot generated by the StarPU. The matrix dimension is 4000×4000 and 15% of the diagonal blocks are selected. From top down: total number of inserted tasks that are not yet issued to the worker threads, number of tasks that are ready to be issued to the worker threads, total flop rate, amount of allocated memory, statuses of the four worker threads (active task and flop rate), and status of the main thread. The red color corresponds to idle time, light green color to `process_window` tasks, teal (bluish green) color to all `right_update` tasks and moss green (yellowish green) color to `left_update` tasks. The horizontal axis shows the wall-time in milliseconds.

which would lead to larger overhead. Actually, the benefits of the blocked approach are more extensive than we expected. There are two reasons for this: Firstly, larger tile size leads to larger update tasks which further leads to higher flop rates in the related `DGEMM` kernels. Secondly, we will actually benefit from the blocked approach even when the diagonal windows are relatively small. This is because we have multiple worker threads running in parallel and the L3 cache, from which the scalar algorithm would otherwise benefit from, is shared among multiple cores. Thus, although a simpler scalar algorithm would usually outperform a more complicated blocked algorithm when executed alone in a vacuum, the blocked algorithm will actually outperform it in a multi-threaded situation.

The computational experiments were performed using the one-pass forward chained blueprint (see Algorithm 5). Investigation into various blueprints is still undergoing but a trace plot shown in Figure 26 gives some reasons to think that that the blueprint actually works as intended. In particular, note how the `process_window` tasks are completed at the very beginning while low priority `right_update` tasks are delayed to the very end. In addition, multiple `process_window` tasks are being executed concurrently. Since the reordering process involves multiple window chains in this particular case, some `left_update` tasks are given higher priority. This can also be seen in the plot.

6 Conclusion and future work

Above all, our work illustrates the power of task based approach. For the problem of reordering the eigenvalues of a matrix in real Schur form we are able to achieve a significant reduction of the time to solution compared with `PDTRSEN` in `ScaLAPACK` when running on a single node (28 cores) of the `Kebnekaise` system. More importantly, we have been able to make excellent use of the available resources. We have recorded strong and weak parallel scaling efficiencies well above 50% and routinely achieve more than 50% of the peak flop rate.

We are currently working to reconfigure and further develop the code to run properly on a distributed memory machine under `StarPU`. We are working with the team developing `StarPU` to resolve related issues. The next step is to extend our algorithm and its `StarPU` realization the generalized eigenvalue problem for real matrices A and B. Here we will be able to capitalize on the work that we have already completed as the underlying principles are the same. We anticipate even fewer problems when addressing the standard and the generalized eigenvalue problem for complex matrices, since there are no 2×2 blocks which require special attention.

It has proven difficult in the extreme to tune our subroutines automatically. Obtaining reliable runtimes for a subroutine with a very short runtime is nearly impossible due to the non-deterministic properties of modern computers. Our efforts to simulate our subroutines have regrettably resulted in limited success so far. Obviously, we will continue to investigate this matter.

Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671633. Support has also been received from eSSENCE, a collaborative e-Science programme funded by the Swedish Government via the Swedish Research Council (VR). All experiments have been conducted using the computing resources available at the High Performance Computing Center North (HPC2N) at Umeå University. We would like to thank the `StarPU` development team of responding with speed and accuracy for our comments, questions and suggestions. Moreover, we thank Prof. Daniel Kressner (EPFL) for allowing us to use the sequential blocked `BDTRSEN` code as a kernel and Lic. Björn Adlerborn (Umeå University) for providing the data that was used in the comparison against the PDTRSEN algorithm.

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] Z. Bai and J. W. Demmel. On swapping diagonal blocks in real Schur form. *Linear Algebra Appl.*, 186:73–95, 1993.

[4] R. Granat, B. Kågström, and D. Kressner. Parallel eigenvalue reordering in real Schur forms. *Concurrency and Computation: Practice and Experience*, 21(9):1225–1250, 2009.

[5] 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.

[6] B. Kågström and P. Poromaa. Computing eigenspaces with specified eigenvalues of a regular matrix pair (A, B) and condition estimation: theory, algorithms and software. *Numer. Algorithms*, 12(3-4):369–407, 1996.

[7] D. Kressner. Block Algorithms for Reordering Standard and Generalized Schur Forms. *ACM Transactions on Mathematical Software*, 32(4):521–532, December 2006.