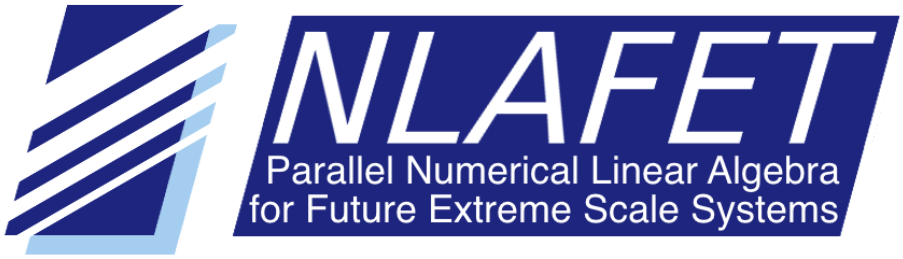


H2020–FETHPC–2014: GA 671633

D6.3

Evaluation of software prototypes

April 2019

Document information

Scheduled delivery	2019-04-30
Actual delivery	2019-04-29
Version	1.0
Responsible partner	UMU

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2019-03-20	Mahmoud, Lars	Draft	0.1	First draft
2019-04-25	Mahmoud, Lars	Final	1.0	Final revision with respect to comments from reviewers

Author(s)

Lars Karlsson (UMU)
Mahmoud Eljammaly (UMU)

Internal reviewers

Sebastien Cayrols (STFC)
Stojce Nakov (STFC)

Contributors

Bo Kågström (UMU)

Copyright

Acknowledgements

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

Table of Contents

1 Introduction **3**
1.1 Some assumptions and terminology 3
1.2 Two scheduling approaches 3
1.3 Why consider parallel critical path scheduling? 4

2 Parallel Critical Path Scheduling using StarPU **6**

3 Structured QR Factorization **6**
3.1 An Optimization Problem 7
3.2 The Structured QR Factorization Kernel 8
3.2.1 The Update Routine 9
3.2.2 The Factor Routine 11
3.3 The Parallel Formulation 11

4 Experimental Analysis **12**
4.1 Tools for Data Analysis 12
4.2 Traces 13
4.3 Pace as a Function of the Block Size 15
4.4 Pace Comparison 16

5 Discussion **16**

1 Introduction

The *Description of Action* (DoA) states for deliverable D6.3:

> "*D6.3: Evaluation of software prototypes*
>
> Report evaluating the prototype software developed in task 6.1 with regard to overall system performance on a selection of linear algebra algorithms."

This deliverable is in the context of Task 6.1 titled *Scheduling and Runtime Systems*. We report on an extension of the work first reported in [7] where we presented a prototypical runtime system optimized for testing the idea of parallelizing the critical path when scheduling a task graph on a multicore system. Here we incorporate the idea into a more feature-complete runtime system called *StarPU* and evaluate it on an important kernel in our recently developed non-symmetric generalized eigenvalue solver. We conclude with some general guidelines to practitioners who want to predict the impact of the idea before investing into the development necessary for its application.

1.1 Some assumptions and terminology

We assume a homogeneous multicore system with p cores and shared memory. We use $G = (V, E)$ to denote a *task graph*, i.e., a directed acyclic graph in which the vertices V represent tasks and the edges E represent precedence constraints.

Take note that we (re)define concepts such as execution time, cost, path length, critical paths, etc. relative to a particular execution of the task graph. The *execution time* of a task graph is defined as the duration from the start of the first task to the completion of the last task. The *cost* of a task graph is defined as the product of the execution time and the number of cores used. The *task time* of $v \in V$, denoted by $\omega(v)$, is defined as the duration from start to completion of v. The *task cost* of $v \in V$, denoted by $c(v)$, is defined as the product of the task time and the number of cores used. In particular, $c(v) = \omega(v)$ for a sequential task v and $c(v) = q\omega(v)$ for a parallel task v using q cores. The *total task cost* of a task graph is defined as the sum of all task costs. The *path length* of a path $v_1, v_2, \ldots, v_n$, where $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, n-1$, is defined as the sum of the task times along the path. Similarly, the *path cost* of a path is defined as the sum of the task costs along the path. A *critical path* is defined as a path with maximal path length.

1.2 Two scheduling approaches

A straightforward approach to execute a task graph on a multicore system is to execute each task sequentially on one core without preemption[1]. We refer to this approach as *regular scheduling*.

Definition 1 (Regular scheduling)**.** *A task-based runtime system for a multicore system with p cores uses* regular scheduling *if it schedules the tasks in a task graph $G = (V, E)$ according to the following principles:*

1. *If $(v_1, v_2) \in E$ then v_2 will not start until v_1 has finished.*

[1] With preemption we mean preemption of an executing task performed by the runtime system. We do not preclude the operating system from preempting threads the runtime system uses to execute tasks.

2. *Once a task has started it runs to completion without preemption.*

3. *Each task is sequential and can be scheduled on any core.*

We contrast regular scheduling with *parallel critical path scheduling*, in which both the set of tasks and the set of cores are partitioned into two subsets. A specific path in the graph (ideally a critical path) is selected and the tasks on this path are executed in parallel (one at a time since they lie on a path) on a set of cores which has been reserved for this purpose. The remaining tasks are executed sequentially on the other set of cores. These tasks are executed precisely as for regular scheduling.

Definition 2 (Parallel critical path scheduling)**.** *A task-based runtime system for a multicore system with p cores and a reservation of $0 \leq q < p$ cores uses* parallel critical path scheduling *if it schedules the tasks in a task graph $G = (V, E)$ according to the following principles:*

1. *If $(v_1, v_2) \in E$ then v_2 will not start until v_1 has finished.*

2. *Once a task has started it runs to completion without preemption.*

3. *The tasks are partitioned as $V = V_{\text{cp}} \cup V_{\text{nc}}$ s.t. all tasks in V_{cp} lie on the same path.*

4. *The cores are partitioned as $C = C_{\text{res}} \cup C_{\text{other}}$ s.t. $|C_{\text{res}}| = q$.*

5. *If $q > 0$, each task in V_{cp} runs in parallel on all cores in C_{res}.*

6. *If $q = 0$, each task in V_{cp} is sequential and runs on a single core in C_{other}.*

7. *Each task in V_{nc} is sequential and can be scheduled only on cores in C_{other}.*

The tasks in V_{cp} are referred to as the critical tasks*, and those in V_{nc} are referred to as the* non-critical tasks*. The cores in C_{res} are referred to as the* reserved cores*. Notice that for $q = 0$ this definition coincides with that of regular scheduling.*

1.3 Why consider parallel critical path scheduling?

Consider the execution of a task graph $G = (V, E)$ on p cores using regular scheduling. Let TIME denote the resulting execution time and let CP denote the length of a critical path. Trivially, we have CP $\leq$ TIME. If the ratio CP/TIME is close to one, then it is effectively the critical paths which are limiting the execution time. Increasing the number of cores is unlikely to yield any significant time savings since the additional cores can have no significant impact on CP under regular scheduling.

Parallel critical path scheduling is attractive precisely when—and only when—the ratio CP/TIME is close to one. By accelerating the tasks along a path, one hopes to reduce CP as well as TIME. However, if CP/TIME is *not* close to one, then a reduction in CP is unlikely to trigger a reduction in TIME simply because the execution time is likely bounded by other factors than the length of a critical path.

Figure 1 helps to explain when and why parallel critical path scheduling can and cannot execute a task graph faster than regular scheduling. The graph in Figure 1a has four identical and independent tasks. Assuming that $p \geq 4$, the execution time will be bounded below by the maximum task time. Parallel critical path scheduling will select one task (path) and parallelize it. The corresponding task time hopefully decreases, but

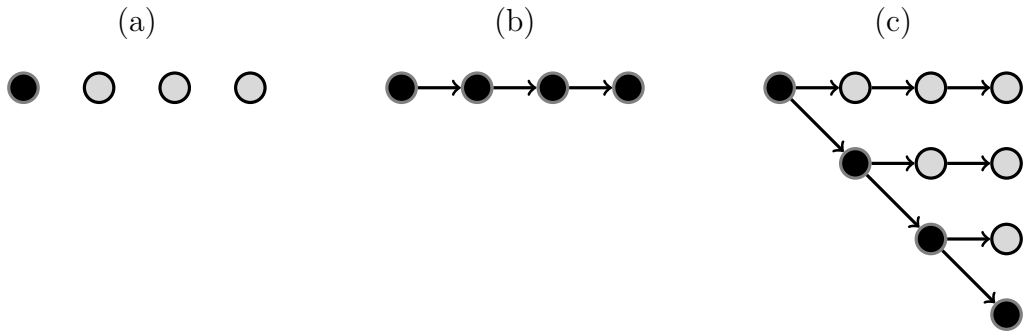


Figure 1: Illustration of task graphs for which parallel critical path scheduling is (a) useless, (b) ideal, and (c) attractive. The selected path (V_{cp} in Definition 2) are shown in black.

the task time of every other task is unaffected. The lower bound on the execution time therefore remains. Hence, parallel critical path scheduling is useless for perfectly parallel computations.[2]

The graph depicted in Figure 1b is the polar opposite of the previous example. All tasks lie on the same path and are therefore completely serialized. Any reduction in the critical path length triggered by parallel critical path scheduling directly translates into an equal reduction in the execution time. This is the ideal case but also the least realistic.

The graph in Figure 1c is more realistic and the effect of parallel critical path scheduling is somewhere in between the two extremes. More precisely, suppose that $p \geq 4$ and $\omega(v) = 1$ for all tasks $v \in V$. Then there are four critical paths, each with a path length of 4. Since $p \geq 4$, it is possible to schedule the tasks using regular scheduling such that the execution time is 4. Suppose that we switch to parallel critical path scheduling with a reservation of q cores and that this reduces the task time of the critical tasks from 1 to $s \in (0, 1]$. The lengths of the four maximal paths are then given by

$$\ell_i = is + (4 - i), \quad i = 1, 2, 3, 4.$$

If we further assume $p - q \geq 3$, then it is possible to schedule the tasks such that the execution time is equal to the length of a critical path ($\ell_1 = s + 3$). The speedup one can achieve by switching from regular scheduling to parallel critical path scheduling is thus bounded above by

$$\text{SPEEDUP} = \frac{4}{s+3} < 4/3.$$

Note that we selected one of the four paths which were critical using regular scheduling, but as soon as we see any acceleration ($s < 1$) it is no longer a critical path. In fact, it becomes the *shortest* path with four tasks. Nevertheless, we see a continuous increase in the speedup when we accelerate the selected path.

[2] Parallelizing the tasks even in an otherwise perfectly parallel computation actually makes sense if the parallelization of tasks achieve super-linear speedup; see [6] for a thorough discussion and numerical experiments. As a simplified example, suppose there are n identical and independent tasks that each take time T_q to execute on q cores. The computational cost is given by nqT_q. Since there is perfect parallelism we may reasonably assume there to be little overhead besides computation and overhead internal to the tasks. Thus the execution time on p cores with each task parallelized over q cores is given by $T_{q,p} \approx nqT_q/p$. If parallelizing the tasks gives a performance boost, then we have $T_{q,p} < T_{1,p}$ for $q > 1$, which implies $T_1/T_q > q$, i.e., the tasks experienced super-linear speedup.

2 Parallel Critical Path Scheduling using StarPU

In this section, we describe how we have implemented parallel critical path scheduling using StarPU [1]. In StarPU, every task is associated with a specific *codelet*. A codelet encapsulates one or more alternative implementations (say for a CPU or a GPU) of some type of task. A task provides the input/output data structures to the task's associated codelet. Each task is submitted to a *scheduling context* which schedules its execution on the required compute resources. By default, there is one scheduling context which includes all compute resources, but other contexts can be created. Each scheduling context has its own scheduling policy and uses a set of workers (threads) to execute the submitted tasks. The set of workers could be dedicated solely for a specific scheduling context or dynamically move between scheduling contexts based on the needs; see [2] for details.

To use StarPU with regular scheduling, it is sufficient to construct codelet(s) with a single sequential CPU implementation. But to use parallel critical path scheduling we need to do some extra work in addition to what is required for regular scheduling:

- Firstly, we must select a specific path in the task graph, ideally but not necessarily one that is likely to be a critical path using regular scheduling. In other words, we need to specify V_{cp} in Definition 2. The path is implicitly specified by telling StarPU when the task is inserted if it is a member of V_{cp} or not.

- Secondly, for each codelet associated with a task in V_{cp}, we must provide a parallel implementation or use a different codelet with a parallel implementation.

- Finally, we must choose the number of reserved cores q. This must be done at the beginning of the computation.

StarPU's concept of scheduling contexts is used to implement the two subsets of cores required for parallel critical path scheduling. The reserved cores belong to one context and the other cores belong to another context.

scheduling contexts

3 Structured QR Factorization

We applied parallel critical path scheduling using StarPU to a kernel in the Hessenberg-triangular reduction algorithm presented in [5]. The overall performance of this algorithm greatly depends on the performance of a kernel which we are about to describe. A detailed understanding of the full algorithm in [5] is not necessary to follow the discussion below.

A matrix pair (A, B) with $A, B \in \mathbb{R}^{n\times n}$ is reduced by the algorithm presented in [5] to Hessenberg-triangular form by an orthogonal equivalence transformation $(A, B) \mapsto (Q^T AZ, Q^T BZ)$ carried out over $n-2$ iterations. Each iteration reduces one column of A and gives rise to a pair of Householder reflectors: one acting from the left and one acting from the right. Periodically, after b pairs of reflectors have been accumulated, the iterations are paused and the reflectors are "absorbed" into the matrix pair. The iterations then resume with an emptied set of reflectors.

The kernel we describe below is one of the most computationally expensive parts of this "absorption" process (see [5] for details). We will see that the number of accumulated reflectors, b, is a parameter of the input together with the dimension of the matrix, n (see Section 3.2 below). For this reason, we cannot directly compare the execution times for

different b. However, we can meaningfully compare the *pace* (time per operation) defined as $P = T/b$, where T is the execution time.

Definition 3 (Pace)**.** *Let T denote the execution time of the kernel with a block size of b. Then the* pace *of the kernel is given by $P = T/b$. The smaller the number, the faster the pace. This is analogous to the notion of pace in the context of running: a runner with a pace of 5 min/km runs faster than one with a pace of 6 min/km.*

The pace tells us how long it takes to process one reflector pair. Since we have a total of $n - 2$ reflector pairs to process, the pace also tells us how fast we are moving towards completion of the full algorithm. In particular, a computation which proceeds at a *fast pace*, i.e., $P = T/b$ is a small number, makes rapid progress towards the final result. For example, suppose the block size b_1 yields a time of T_1 while $b_2 = 2b_1$ yields a time in the range $T_1 < T_2 < 2T_1$, then the paces P_1 and P_2 satisfy the inequalities

$$\frac{1}{2}P_1 < P_2 < P_1.$$

In other words, the larger block size b_2 gives a faster pace than the smaller block size b_1 and should therefore be preferred even though the execution time is longer. In the analogy of running, a runner doing 1 km in 10 min is clearly slower than a runner doing 2 km in 15 min even though she completes her distance faster.

3.1 An Optimization Problem

Thus far we have considered only a single task graph. In the context of Hessenberg-triangular reduction we are actually interested in an optimization problem over a finite set of related task graphs. Fix the matrix dimension n. For each block size b there is an associated task graph G_b. We seek to find a block size which yields the fastest pace.

We are free to choose, for any given block size b, either regular scheduling or parallel critical path scheduling with some reservation q. Let $P_{\text{reg}}(b)$ denote the pace we get for block size b with regular scheduling. The fastest pace we can achieve with regular scheduling is given by

$$P_{\text{reg}}^{\text{opt}} = \min_b P_{\text{reg}}(b).$$

Similarly, let $P_{\text{pcp}}(b, q)$ denote the pace we get for block size b with parallel critical path scheduling with a reservation of q cores. The fastest past we can achieve with parallel critical path scheduling is given by

$$P_{\text{pcp}}^{\text{opt}} = \min_{b,q} P_{\text{pcp}}(b, q).$$

Can parallel critical path scheduling achieve a faster pace than regular scheduling? In other words, is the speedup ratio

$$\text{SPEEDUP} = \frac{P_{\text{reg}}^{\text{opt}}}{P_{\text{pcp}}^{\text{opt}}} \tag{1}$$

greater than one? This is the question we seek to understand in this report.

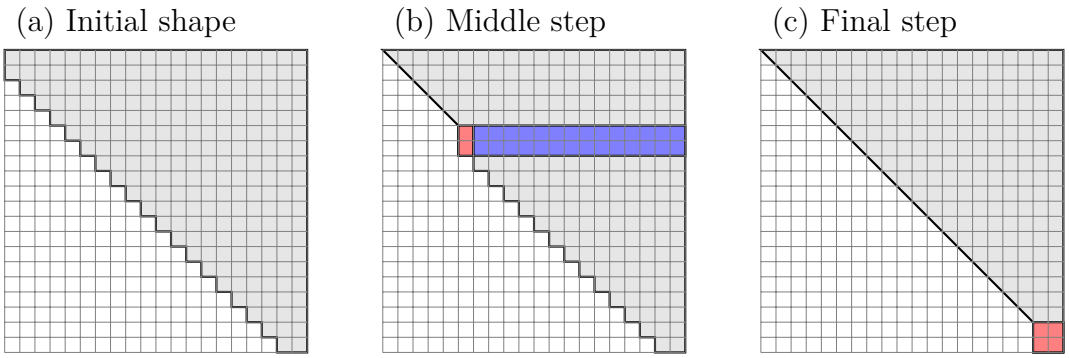


Figure 2: (a) Matrix A in upper block Hessenberg form. (b) A step in the middle of Algorithm 1. (c) The final step of Algorithm 1.

3.2 The Structured QR Factorization Kernel

The kernel we are considering is illustrated in Figure 2 and we refer to it as a *structured QR factorization* for reasons that will become clear in a moment. The input is a matrix A of size $n \times n$ in block upper Hessenberg form with blocks of size $b \times b$ (see Figure 2a). Every grid cell in Figure 2 represents one such $b \times b$ block. The aim is to reduce A to upper triangular form by orthogonal transformations from the left, i.e., to *implicitly* construct an orthogonal matrix Q such that $R = Q^T A$ is upper triangular.

Algorithm 1: StructuredQR(A)

1 **for** $j \leftarrow 1, 2, \ldots, N-2$ **do**
  `// Reduce a` $2b \times b$ `block near the diagonal (red)`
2 $W, Y, A_{j:j+1,j} \leftarrow$ FACTOR($A_{j:j+1,j}$);
  `// Update an off-diagonal horizontal slab (blue)`
3 $A_{j:j+1,j+1:N} \leftarrow$ UPDATE($W, Y, A_{j:j+1,j+1:N}$);
`// Reduce the final` $2b \times 2b$ `block (red)`
4 $W, Y, A_{N-1:N,N-1:N} \leftarrow$ FACTOR($A_{N-1:N,N-1:N}$);

Algorithm 1 is straightforward. The input matrix A is overwritten by the upper triangular matrix $R = Q^T A$. The matrix Q is not computed explicitly. The number of block columns is denoted by $N = \lceil n/b \rceil$. The matrix is reduced one block column at a time from left to right using $N-2$ identical iterations and one special case at the end to take care of the last two block columns. In each iteration, a $2b \times b$ block (red in Figure 2b) is reduced to upper triangular form by a QR factorization. The columns to the right (blue in Figure 2b) are subsequently updated by a multiplicative update from the left with the previously computed orthogonal factor. In the end, the two last block columns are reduced (red in Figure 2c) to triangular form by a QR factorization without any subsequent updates.

A product of k reflectors

$$H_1 H_2 \cdots H_k = \prod_{i=1}^{k} (I - \tau_i v_i v_i^T), \quad H_i = I - \tau_i v_i v_i^T$$

is represented in regular WY form [4]

$$H_1 H_2 \cdots H_k = I - W Y^T, \quad W, Y \in \mathbb{R}^{n \times k}$$

for the sake of computational efficiency[3]. Note that a reflector $I - \tau vv^T$ is essentially already in WY form simply by taking $W = v$ and $Y = \tau v$.

The product of two WY forms can be expressed in WY form as well. Let $I - W_1 Y_1^T$ and $I - W_2 Y_2^T$ be two WY forms representing the products of $k_1 \geq 1$ and $k_2 \geq 1$ reflectors, respectively. Then their product can again be expressed in WY form as seen by

$$(I - W_1 Y_1^T)(I - W_2 Y_2^T) = I - \underbrace{\begin{bmatrix} W_1 & W_2 \end{bmatrix}}_{W} \underbrace{\begin{bmatrix} (I - W_2 Y_2^T)^T Y_1 & Y_2 \end{bmatrix}^T}_{Y^T} = I - WY^T, \qquad (2)$$

where W and Y have $k_1 + k_2$ columns each.

3.2.1 The Update Routine

The UPDATE routine (line 3 in Algorithm 1) applies an in-place update of the form

$$A \leftarrow (I - WY^T)^T A,$$

where A is a general matrix of size $m \times \ell$ and $I - WY^T$ is a WY form with $k \leq m/2$ reflectors. The matrices W and Y, produced by the FACTOR routine defined below, are lower trapezoidal. This shape is exploited by partitioning both W and Y into a lower triangular submatrix of size $k \times k$ on top of a dense submatrix of size $(m - k) \times k$. Algorithm 2 provides details. The computation is dominated by calls to the `TRMM`, triangular matrix times general matrix, and `GEMM`, general matrix times general matrix, level 3 BLAS routines, which can execute with high performance due to cache reuse.

Algorithm 2: $A = \text{UPDATE}(W, Y, A)$

1 Partition A, W, and Y as in

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, \quad W = \begin{bmatrix} W_1 \\ W_2 \end{bmatrix}, \quad Y = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix}$$

such that A_1, Y_1, W_1 have k rows;
2 $Z \leftarrow A_1$; `// Copy`
3 $Z \leftarrow W_1^T Z$; `// TRMM`
4 $Z \leftarrow Z + W_2^T A_2$; `// GEMM`
5 $A_2 \leftarrow A_2 - Y_2 Z$; `// GEMM`
6 $Z \leftarrow Y_1 Z$; `// TRMM`
7 $A_1 \leftarrow A_1 - Z$; **return** A;

We need both sequential and parallel implementations of the UPDATE routine. We have developed two variations of the parallel implementation. In the *column-partitioned variant* we uniformly partition the columns of A into q blocks and assign one block to each of the q reserved cores. This perfectly parallel scheme is well suited for performing update tasks along the selected path (see Section 3.3 below). Algorithm 3 shows the column-partitioned variant of the parallel update.

The *row-partitioned variant* is suitable when the number of columns is small, which is the case when updates are performed inside the FACTOR routine. We uniformly partition the rows of A into q blocks and assign one block to each core. Communication between cores is necessary when computing $W^T A$ in parallel. Algorithm 4 shows the row-partitioned variant of the parallel update.

[3]Regular WY has a significantly lower arithmetic overhead than compact WY when the size of the reflectors is not much greater then the number of reflectors.

Algorithm 3: $A = \text{UPDATE}(W, Y, A)$ *column-partitioned*

Partition A as $A = \begin{bmatrix} A_1 & A_2 & \cdots & A_q \end{bmatrix}$;
foreach $i \in \{1, 2, \ldots, q\}$ **do** `// in parallel`

  Partition A_i, W, and Y as in

$$A_i = \begin{bmatrix} A_{1i} \\ A_{2i} \end{bmatrix}, \quad W = \begin{bmatrix} W_1 \\ W_2 \end{bmatrix}, \quad Y = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix}$$

  such that A_{1i}, Y_1, W_1 have k rows;
  $Z \leftarrow A_{1i}$; `// Copy`
  $Z \leftarrow W_1^T Z$; `// TRMM`
  $Z \leftarrow Z + W_2^T A_{2i}$; `// GEMM`
  $A_{2i} \leftarrow A_{2i} - Y_2 Z$; `// GEMM`
  $Z \leftarrow Y_1 Z$; `// TRMM`
  $A_{1i} \leftarrow A_{1i} - Z$;
return A;

Algorithm 4: $A = \text{UPDATE}(W, Y, A)$ *row-partitioned*

Partition A, W, and Y as in

$$A_i = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_q \end{bmatrix}, \quad W = \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_q \end{bmatrix}, \quad Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_q \end{bmatrix}$$

  such that A_1, Y_1, W_1 have k rows;
foreach $i \in \{1, 2, \ldots, q\}$ **do** `// in parallel`

  if $i = 1$ **then**
    $Z_1 \leftarrow A_1$; `// Copy`
    $Z_1 \leftarrow W_1^T Z_1$; `// TRMM`
  else
    $Z_i \leftarrow Z_i + W_i^T A_i$; `// GEMM`
  BARRIER(); `// synchronize`
  $Z_i \leftarrow Z_1 + Z_2 + \cdots + Z_q$; `// ReduceAll`
  if $i = 1$ **then**
    $Z_1 \leftarrow Y_1 Z_1$; `// TRMM`
    $A_1 \leftarrow A_1 - Z_1$;
  else
    $A_i \leftarrow A_i - Y_i Z$; `// GEMM`
return A;

3.2.2 The Factor Routine

The FACTOR routine (lines 2 and 4 in Algorithm 1) computes a QR factorization

$$A = QR = (I - WY^T)R$$

of a matrix A of size $m \times \ell$ where $m \geq \ell$ and in the process overwrites A by the upper trapezoidal or triangular matrix R.

Algorithm 5: $W, Y, R = \text{FACTOR}(A)$

```
1  Let m × ℓ be the size of A (assuming m ≥ ℓ);
2  if m = ℓ = 1 then
3      return ([], [], A);
4  k ← min{m − 1, ℓ};
5  if ℓ = 1 then
       // A is a vector
6      Reduce (the vector) A by a reflector I − τvv^T;
7      return (v, τv, A);
8  else
       // Split the columns in two halves
9      ℓ1 ← ⌊ℓ/2⌋;
10     ℓ2 ← ℓ − ℓ1;
       // Factor the left half (recursively)
11     W1, Y1, A1:m,1:ℓ1 ← FACTOR(A1:m,1:ℓ1);
       // Update the right half
12     A1:m,ℓ1+1:ℓ ← UPDATE(W1, Y1, A1:m,ℓ1+1:ℓ);
       // Factor the right half (recursively)
13     W2, Y2, Aℓ1+1:m,ℓ1+1:ℓ ← Factor(Aℓ1+1:m,ℓ1+1:ℓ);
14     Pad the tops of W2 and Y2 with ℓ1 rows of zeros;
       // Construct the product of the two WY forms
15     (Y1)ℓ1+1:m,1:ℓ1 ← UPDATE(W2, Y2, (Y1)ℓ1+1:m,1:ℓ1);
16     return ([W1  W2], [Y1  Y2], A);
```

Algorithm 5 recursively over the columns factors A such that the triangular factor R overwrites A. Both W and Y will be lower trapezoidal/triangular, i.e., they will be zero above the main diagonal.

The FACTOR routine is parallelized by partitioning the rows of A into q blocks. The construction of reflectors is serialized. The bulk of the work is in calls to the UPDATE routine for which we use the row-partitioned variant (see Section 3.2.1).

3.3 The Parallel Formulation

The structured QR factorization (Algorithm 1) is straightforward to formulate in terms of a task graph. Calls to the FACTOR routine translate into a task of "factor" type, each factoring a $2b \times b$ or $2b \times 2b$ block. Calls to the UPDATE routine translate into a set of independent tasks of "update" type, each updating a $2b \times b$ block. There are data dependencies from a FACTOR task to every UPDATE task within the same iteration. There are also data dependencies from an UPDATE task in one iteration to the task (either an UPDATE or a FACTOR) in the next iteration that affects the same block column. Figure 3 illustrates the task graph for $N = n/b = 6$. The highlighted path (thick red edges) is the one that we select for parallel critical path scheduling.

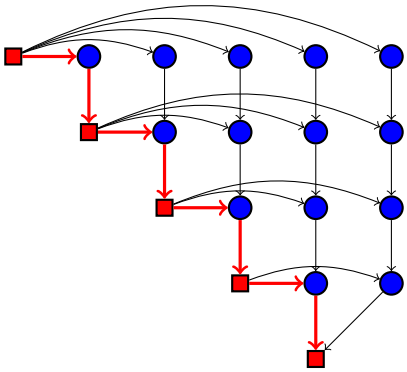


Figure 3: Illustration of the task graph for the structured QR factorization when $n/b = 6$.

Accelerating the selected path infinitely will reduce the critical path length: the question is by how much? Let ω_{fac} and ω_{upd} denote the time per FACTOR and UPDATE task, respectively. If we let the time for each critical task go to zero, then the longest path in the graph will shrink from

$$(N-1)\omega_{\text{fac}} + (N-2)\omega_{\text{upd}} \qquad \text{to} \qquad (N-2)\omega_{\text{upd}}.$$

(The path corresponding to the shorter length includes the right-most column of tasks in the figure.) The relative change is given by

$$\frac{(N-1)\omega_{\text{fac}} + (N-2)\omega_{\text{upd}}}{(N-2)\omega_{\text{upd}}} = 1 + \frac{N-1}{N-2}\frac{\omega_{\text{fac}}}{\omega_{\text{upd}}} \approx 1 + \frac{\omega_{\text{fac}}}{\omega_{\text{upd}}}.$$

Clearly, even with an infinite acceleration of the selected path, the critical path length drops by a relatively modest factor.

4 Experimental Analysis

We used StarPU version 1.2.2. Unless otherwise noted, we used the `peager` scheduler[4] for the critical context and the `prio` scheduler for the non-critical context[5]. Priorities were computed at task insertion time using a formula derived from a critical path heuristic, i.e., a task is given a priority which is proportional to the number of tasks in the longest path that originates from the task. All experiments were performed on one compute node (in exclusive mode) of the Kebnekaise system [3] at HPC2N, Umeå University. One node consists of 28 Intel Xeon E5-2690v4 cores (spread over 2 sockets) with 128 GB RAM.

4.1 Tools for Data Analysis

We present analyses with the aim of gaining insight into the two scheduling approaches and how they interact with the pace optimization problem defined in Section 3.1.

Let G_b denote the task graph associated with block size b. Let $T_{\text{reg}}(b)$ and $P_{\text{reg}}(b) = T_{\text{reg}}(b)/b$ denote the time and pace using regular scheduling, respectively. Similarly, let

[4]StarPU version 1.2.2 supports two *experimental* parallel schedulers and `peager` was the only one working with our implementation.

[5]The global view of the task graph encoded in the priorities used by the `prio` scheduler often leads to schedules with less idle time compared to other schedulers.

$T_{\text{pcp}}(b,q)$ and $P_{\text{pcp}}(b,q) = T_{\text{pcp}}(b,q)/b$ denote the time and pace using parallel critical path scheduling, respectively.

The length of any path in G_b is a lower bound on the execution time and hence the pace. This applies in particular to the path, V_{cp}, we select for parallel critical path scheduling. Formally, we have

$$P_{\text{pcp}}(b,q) \geq \frac{1}{b} \sum_{v \in V_{\text{cp}}} \omega(v). \tag{3}$$

We refer to this as the *critical path bound*. This bound applies also to regular scheduling.

A second limiting factor is the amount of parallel resources available to execute the non-critical tasks. More precisely, the pace is bounded below by

$$P_{\text{pcp}}(b,q) \geq \frac{1}{b} \cdot \frac{1}{p-q} \cdot \sum_{v \in V_{\text{nc}}} \omega(v). \tag{4}$$

We refer to this as the *non-critical cost bound*. A similar bound applies also to regular scheduling if we take $q = 0$ and sum over all tasks.

The non-critical cost bound is somewhat unrealistic as close to perfect load balancing is not achievable in practice when n/b is small relative to $p-q$. The structure of the task graph (see Figure 3) is such that towards the end of the computation, fewer and fewer tasks can be ready at the same time. Specifically, after having exactly $p-q$ ready tasks the cores become idle one by one at the start of each factor task. The number of cores which are idle at the start of each factor task sum to

$$\sum_{i=1}^{p-q-1} i = \frac{(p-q)(p-q-1)}{2}.$$

The time that each core is going to be idle between two successive factor tasks is estimated to be the average time for a critical factor task and a critical update task. Based on this, we estimate the *inherent idling cost* by

$$\frac{(p-q)(p-q-1)}{2} \cdot (\omega_{\text{fac}} + \omega_{\text{upd}}), \tag{5}$$

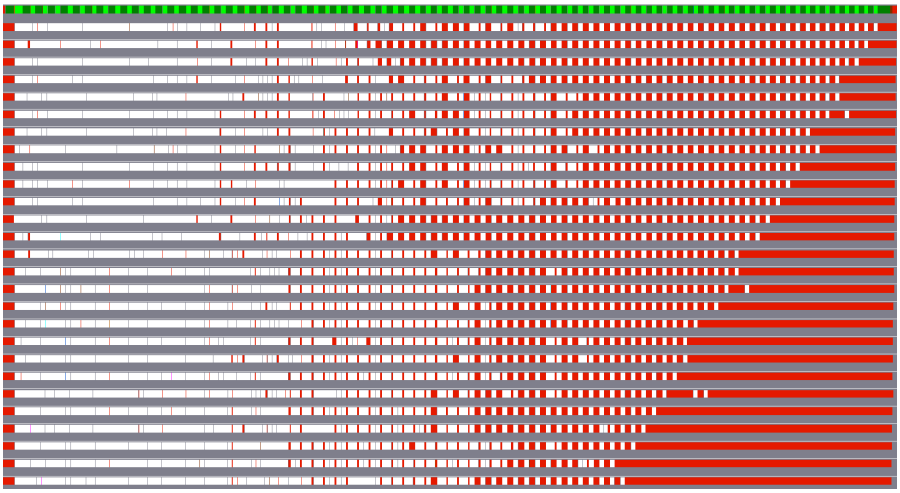
where ω_{fac} is the average time of a (critical) factor task and ω_{upd} is the average time of a (critical) update task. Using this estimate to augment Eq. (4) we obtain the more realistic approximate lower bound

$$P_{\text{pcp}}(b,q) \gtrapprox \frac{1}{b} \left(\frac{1}{p-q} \cdot \sum_{v \in V_{\text{nc}}} \omega(v) + \frac{p-q-1}{2} \cdot (\omega_{\text{fac}} + \omega_{\text{upd}}) \right). \tag{6}$$

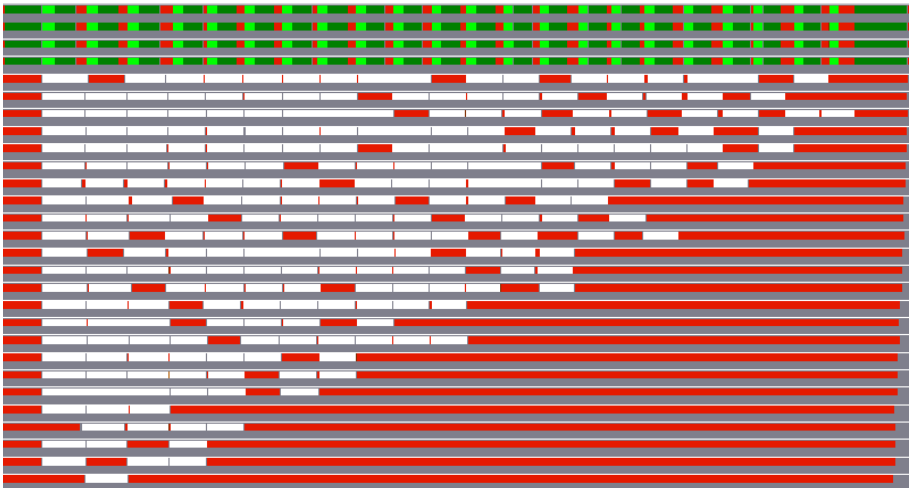
4.2 Traces

Figure 4a shows the trace for one example where parallel critical scheduling would be pointless. The sheer volume of non-critical tasks, rather than the length of the critical path, is limiting the execution time. In general, when n/b is sufficiently large relative to $p-q$ and $q = 1$, the non-critical work will dominate. Note how the non-reserved cores are busy with little idling. The reserved core (top row) runs out of work approximately one quarter into the execution. There it starts to idle while waiting for the non-reserved

(a) $n = 40000$, $b = 185$, $q = 1$, execution time: 1908.057 ms



(b) $n = 40000$, $b = 494$, $q = 1$, execution time: 6033.201 ms



(c) $n = 20000$, $b = 834$, $q = 4$, execution time: 3182.128 ms

Figure 4: Traces illustrating a variety of phenomena observed using parallel critical path scheduling. Each row is one core and time flows left to right. The reserved cores are on the first q rows. Green is a critical task, white is a non-critical task, red is idling, and gray is the background.

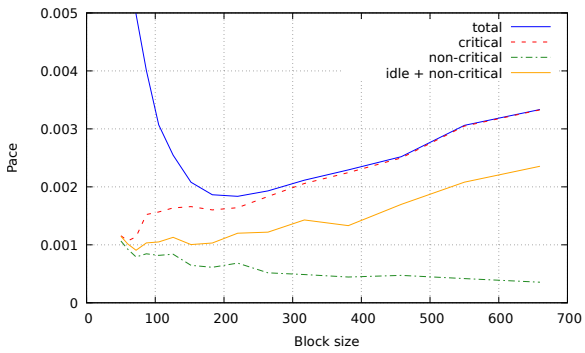
cores to complete the task(s) which are stalling the next critical task. The execution time is close to the non-critical cost bound Eq. (6).

Figure 4b shows the trace for a contrasting example where parallel critical path scheduling actually could help. Here $q = 1$ but the ratio n/b is now small relative to $p - q$. Note how the reserved core (top row) is executing the critical tasks without any interruptions. The non-reserved cores, on the other handle, are idling quite a bit towards the end, due to the task graph's inherent idling (recall Section 4.1). The execution time is close to the critical path bound Eq. (3).

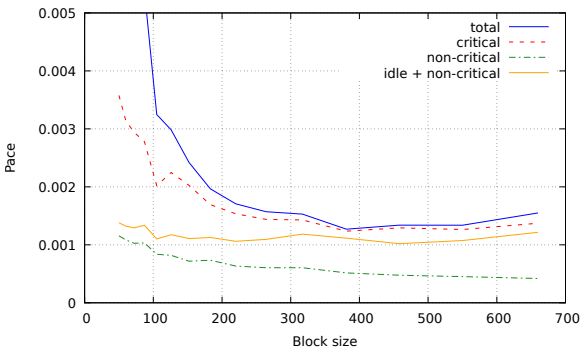
Figure 4c shows the trace for an example where parallel critical path scheduling has been applied and significantly reduced the execution time. The ratio n/b is even smaller relative to $p - q$ and now $q = 4$. Note that there are periods of idling between pairs of critical tasks despite a lack of non-critical tasks. The reason for this is that the selected path has been accelerated to such an extent that it is not a critical path. However, recall from the toy example in Figure 1c in Section 1.3 that a reduction in the execution time by increasing q is still possible.

4.3 Pace as a Function of the Block Size

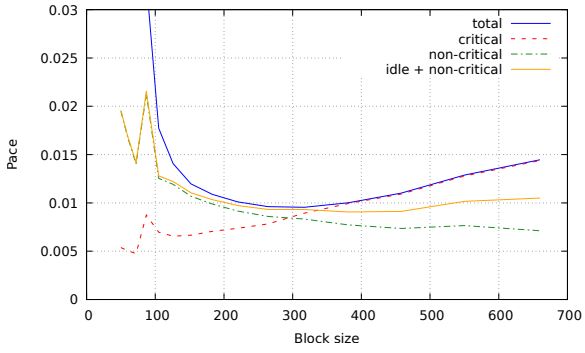
Figure 5 plots the pace as a function of the block size for a couple of values of n and q and aims to illustrate some qualitatively different cases. For block sizes $b \lessapprox 100$, the overhead in the runtime system is too severe to be of practical use[6].



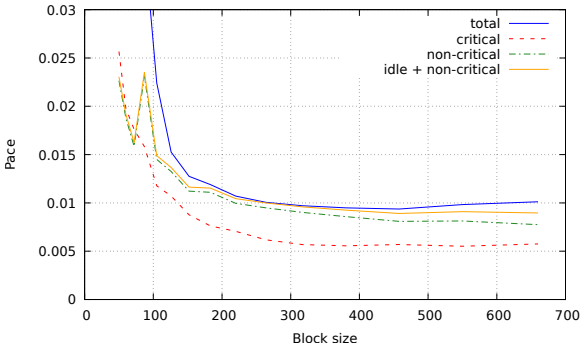
(a) $n = 10000$, $q = 1$



(b) $n = 10000$, $q = 4$



(c) $n = 40000$, $q = 1$



(d) $n = 40000$, $q = 4$

Figure 5: The pace as a function of the block size compared to the bounds from Section 4.1.

[6]We can tell this by observing that the execution time is far above the nearest bound.

In Figure 5a, the pace is limited by the critical path for $b \gtrapprox 200$. For smaller block sizes the runtime overhead becomes significant. Parallelizing the critical path improves the pace for $b \gtrapprox 200$, as seen by comparing with Figure 5b. Note also that the inherent idling cost dramatically decreases as a result of accelerating the critical path.

In Figure 5c, block sizes in the range $150 \lessapprox b \lessapprox 300$ are large enough to not be impacted by overhead but also small enough to make the non-critical cost bound Eq. (4) active. The cost is dominated by the non-critical tasks and the critical path is not a limiting factor here. Parallel critical path scheduling would thus be useless in this range. For $b \gtrapprox 300$, the critical path starts to dominate the cost and the bound Eq. (3) is active. Parallelizing the critical path has a significant impact in this range, as is evident in Figure 5d where $q = 4$. The non-critical cost bound Eq. (6) has become active.

4.4 Pace Comparison

The central question is how regular scheduling compares with parallel critical path scheduling under optimal conditions, meaning optimally chosen block sizes and number of reserved cores (recall Section 3.1). Figure 6 shows how the two perform as a function of the block size b when q is chosen optimally. The initial downward slope is due to the lessening impact of the runtime system overhead. The curve for regular scheduling eventually increases due to the critical path starting to dominate. The curve for parallel critical path scheduling stays more flat since the acceleration can counter the increased work on the critical path.

Figure 7 illustrates the speedup ratio for each block size, defined as

$$\text{SPEEDUP} = \frac{P_{\text{reg}}(b)}{\min\limits_{q \in \{0,1,\ldots,4\}} P_{\text{pcp}}(b,q)}. \tag{7}$$

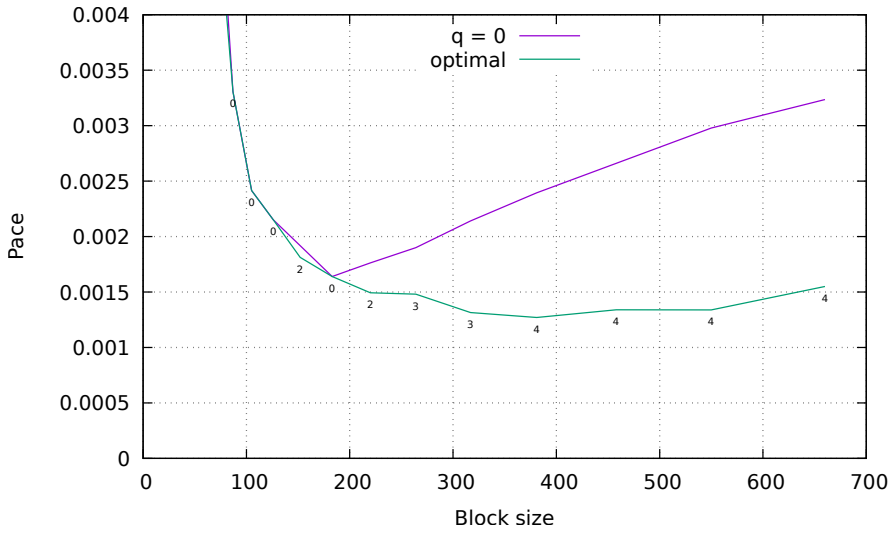
This plot shows that speedup can be observed for sufficiently large block sizes. For large problems (e.g., $n = 40000$), a larger block size is required before any speedup can be observed.

The comparisons above use the same block size for both approaches. This is illustrative but not a proper comparison since we can choose the block size independently for each. A more appropriate metric is the ratio Eq. (1) from Section 3.1. The observed speedups for various matrix sizes is summarized in Figure 8. We conclude that parallel critical path scheduling provides the most speedup for medium-sized problems, and the optimal block size is larger than for regular scheduling.

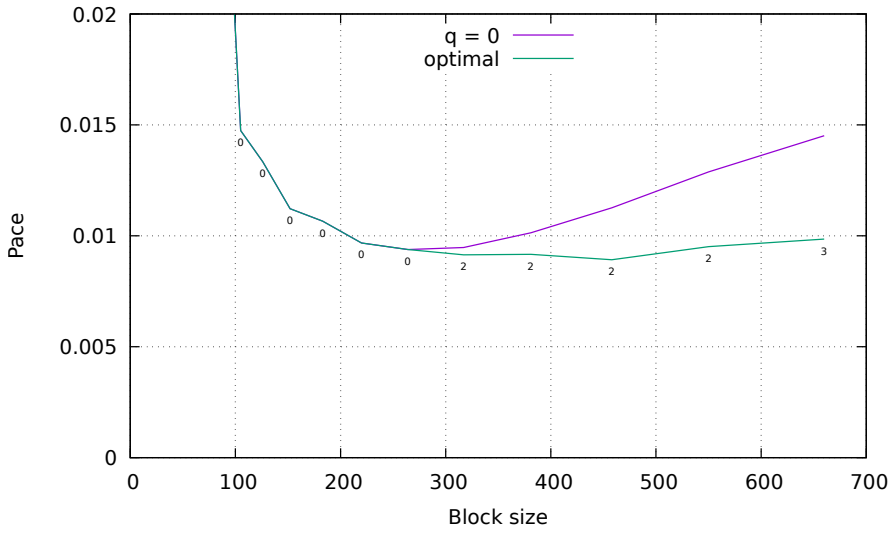
5 Discussion

The results and analyses in Section 4 show that parallel critical path scheduling can improve the pace of the kernel when parameters are chosen optimally. A speedup of up to 1.4 has been observed for medium-sized matrices.

However, using parallel critical path scheduling is not without cost. Parallel implementations must be developed for the task types on the selected path, and a strategy for optimizing the new parameter q must be devised. An important engineering question is thus: How can I determine ahead of time how much, if any, speedup I can hope to gain from applying parallel critical path scheduling?



(a) $n = 10000$.



(b) $n = 40000$.

Figure 6: Comparison of regular scheduling (labeled "$q = 0$") with parallel critical path scheduling with optimally chosen $q \in \{0, 1, 2, 3, 4\}$. The numbers in the plot indicate the optimal value of q for the associated block size.

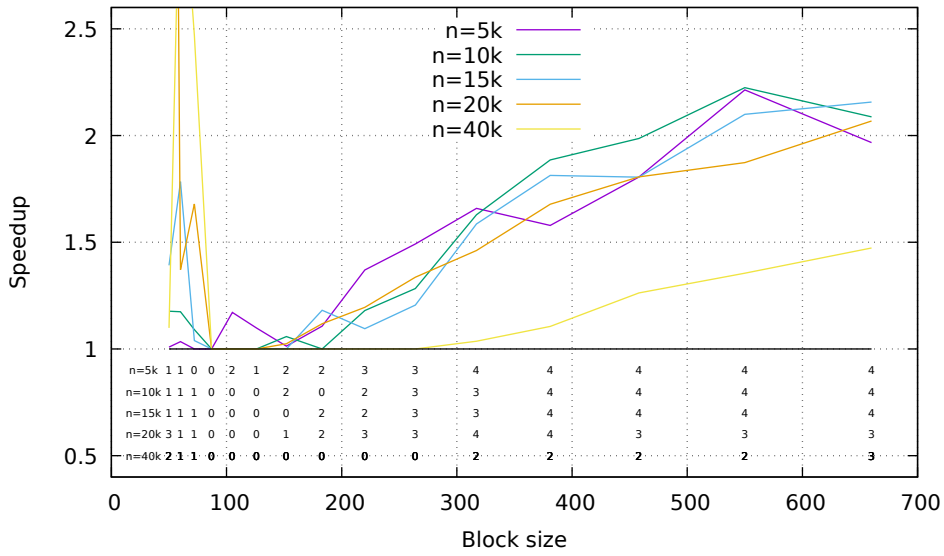


Figure 7: The speedup of parallel critical path scheduling over regular scheduling on a block size by block size basis. The five rows of numbers indicate the optimal value of q for the corresponding block size and matrix size.

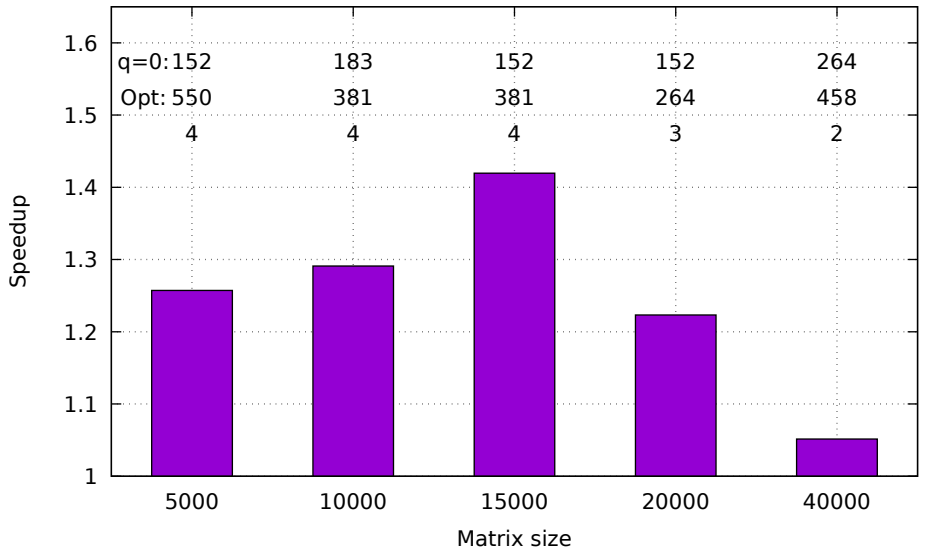


Figure 8: Speedup calculated according to Eq. (1) for a variety of matrix sizes. The numbers in the rows on top show the optimal values of b for each approach (first two rows) and the optimal values of q (third row) for parallel critical path scheduling.

We sketch, by means of an example, a methodology that aims to estimate an answer to this question. For the kernel described in Section 3, the following procedure can be used *a priori* to estimate the maximum speedup achievable through parallel critical path scheduling.

1. Select a path in the task graph which is thought to be a critical path.

2. Using regular scheduling, run a parameter sweep across a sufficiently wide range of block sizes $b = b_1, b_2, \ldots, b_m$ to capture the characteristic "bowl shape" of the pace curve (see Figure 5). For each block size b_i, record:

   - the total execution time T_i,
   - the total critical task cost C_i^{cp},
   - the total non-critical task cost C_i^{nc},
   - the average execution time of a factor task ω_i^{fac}, and
   - the average execution time of a critical update task ω_i^{upd}.

3. Based on these measurements we can speculate how the kernel will perform under parallel critical path scheduling. According to Eq. (3) and Eq. (6), both the length of the selected path and the non-critical task cost impose bounds which depend on q. Taking the maximum of these two bounds and *generously* assuming linear speedup for the critical tasks, we obtain the following bound:

$$P_{\text{pcp}}(b_i, q) \geq \frac{1}{b_i} \cdot \max\left\{\left(\frac{C_i^{\text{nc}} + C_i^{\text{id}}}{p-q}\right), \frac{C_i^{\text{cp}}}{q}\right\}, \tag{8}$$

   where the inherent idling cost according to Eq. (5) (and linear speedup of tasks) is estimated by

$$C_i^{\text{id}} = \frac{(p-q)(p-q-1)}{2} \cdot \frac{\omega_i^{\text{fac}} + \omega_i^{\text{upd}}}{q}. \tag{9}$$

4. The observed pace for regular scheduling is T_i/b_i, which includes overheads. The bound Eq. (8), on the other hand, does not. To compensate for this we estimate the amount of overhead and add it to the estimate for parallel critical path scheduling. Specifically, we estimate the overhead by

$$T_i^{\text{oh}} = T_i - \max\left\{C_i^{\text{cp}}, (C_i^{\text{nc}} + C_i^{\text{cp}})/p\right\}, \tag{10}$$

   which is essentially the difference between the observed time T_i and the closest bound. We then estimate the maximum speedup that we can hope for using parallel critical path scheduling by

$$S_{\max} = \frac{\min_i T_i/b_i}{\min_{i,q} \frac{1}{b_i}\left(\max\left\{(C_i^{\text{nc}} + C_i^{\text{id}})/(p-q), C_i^{\text{cp}}/q\right\} + T_i^{\text{oh}}\right)}. \tag{11}$$

Figure 9 compares the speedup estimates calculated via Eq. (11) with the actual speedups reported in Figure 8. The corresponding optimal values of b and q for each matrix size are reported in Table 1. Clearly, the speedup estimate is optimistic: in all cases the speedup is overestimated. For small matrices and after examining the data,

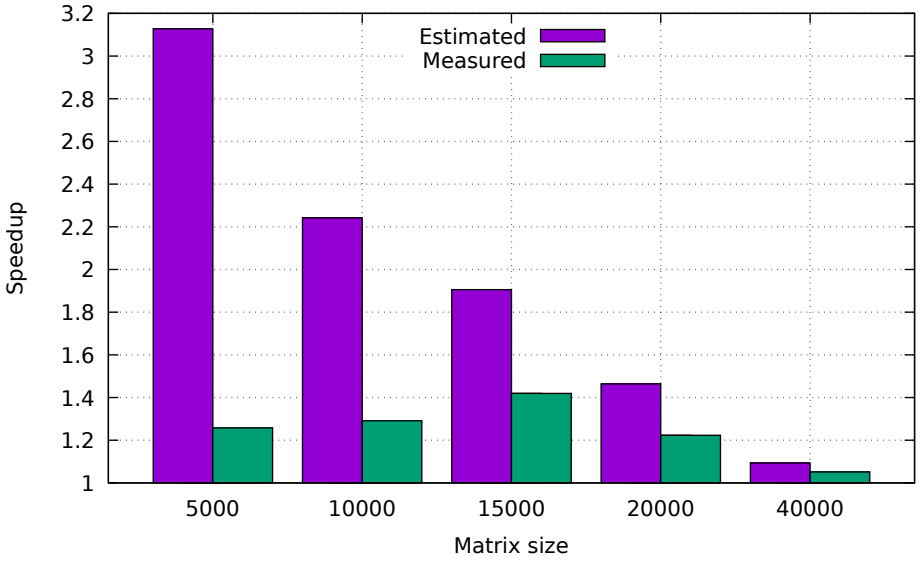


Figure 9: Speedup estimates using Eq. (11) compared to the measurements in Figure 8.

Table 1: The optimal values of b and q associated with the data in Figure 9.

n	5000		10000		15000		20000		40000	
	b	q	b	q	b	q	b	q	b	q
Regular	152	0	183	0	152	0	152	0	264	0
Estimated	152	4	220	4	458	4	458	4	458	2
Measured	550	4	381	4	381	4	264	3	458	2

the large gap can be explained by the unrealistic assumption of linear speedup of the critical tasks. This suggests that if additional information about the parallel speedup of the critical tasks is available, then the accuracy of the speedup estimates can likely be greatly improved.

Acknowledgements

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

References

[1] StarPU, starpu.gforge.inria.fr.

[2] StarPU Handbook, http://starpu.gforge.inria.fr/files/starpu-1.2.2/html/.

[3] https://www.hpc2n.umu.se/resources/hardware/kebnekaise.

[4] Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1987.

[5] Z. Bujanović, L. Karlsson, and D. Kressner. A Householder-based algorithm for Hessenberg-triangular reduction. *SIAM Journal on Matrix Analysis and Applications*, 39(3):1270–1294, 2018.

[6] L. Karlsson, C.C.K. Mikkelsen, and B. Kågström. Improving perfect parallelism. In *International Conference on Parallel Processing and Applied Mathematics*, pages 76–85. Springer, 2013.

[7] L. Karlsson, A. Schwarz, and C.C.K. Mikkelsen. Prototypes for runtime systems exhibiting novel types of scheduling. *NLAFET Public Deliverable D6.1*, 2017.