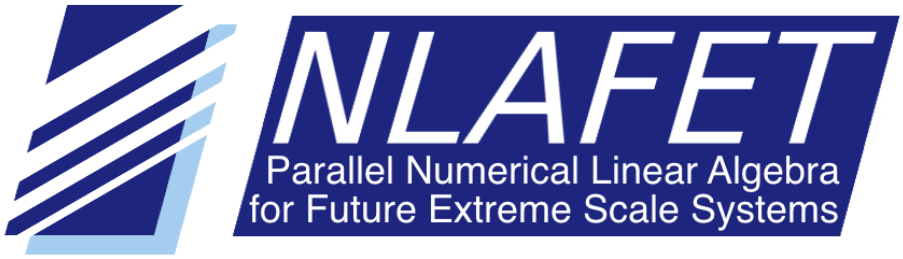


H2020–FETHPC–2014: GA 671633

D6.5

Evaluation of auto-tuning techniques

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
2019-04-01	Lars, Mahmoud, Mirko	Draft	0.1	First draft
2019-04-25	Lars, Mahmoud, Mirko	Final	1.0	Final revision with respect to comments from reviewers

Author(s)

Lars Karlsson (UMU)
Mahmoud Eljammaly (UMU)
Mirko Myllykoski (UMU)

Internal reviewers

Jan Papez (INRIA)
Srikara Pranesh (UNIMAN)
Pierre Blanchard (UNIMAN)

Contributors

Bo Kågström (UMU)
Angelika Schwarz (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

List of Figures

1 Introduction

The *Description of Action* (DoA) states for deliverable D6.5:

> "*D6.5: Evaluation of auto-tuning techniques*
>
> Report on the effect of applying the novel scheduling and auto-tuning prototypes to various linear algebra problems."

This deliverable is in the context of Task 6.1 (Scheduling and Runtime Systems).

Consider the perspective of a user of a parallel numerical linear algebra library developed by a third party. The performance of a library routine is determined by the interactions of a multitude of factors, including but not limited to

- the characteristics of the machine,
- the number and configuration of the nodes and cores,
- the characteristics of the problem instances.

The user expects high performance from the library routine, but the library developers had no knowledge of any of the above when they developed and tuned their code. The developers are faced with the challenge of developing software which performs well on unknown hardware, using unknown configurations, when applied to unknown problem instances.

A common way to tackle the challenge is by making the internals of the routine configurable from the outside. A number of *algorithmic parameters* (e.g., tile sizes) are introduced to control some of the internals of the routine. The parameters are then configured externally, either manually by the user or in some automated fashion using offline or online auto-tuning tools. The LAPACK library is one example of a configurable library. Many of the routines in LAPACK are configurable and the parameter values are determined by the `ILAENV` auxiliary function. The library itself does not include an auto-tuning facility, but in principle a third-party tool can be hooked up to provide such functionality [7]. A user of LAPACK can also choose to manually tweak this function to optimize the configuration for the user's particular environment and types of problem instances.

A different approach is to make the routine adapt itself at run-time. We here refer to this as a *self-adaptive routine*.

Definition 1 (Self-adaptive routine)**.** *A* self-adaptive routine *attempts to continuously optimize its own performance by making its behavior dependent on measurements of durations and/or on the ordering of events during productive runs of the routine. The measurements may stem from the current run and/or previous runs.*

Integrating a regular offline auto-tuning technique into a routine will make it self-adaptive. But the definition above allows for adaptive mechanisms which are not so easily expressed in terms of optimization of parameters. Dynamic scheduling of tasks is an excellent example of a self-adaptive mechanism. The order in which tasks are scheduled and the assignment of tasks to threads is indirectly determined by the ordering of task-completion events in the present run.

In this report, we share some of our experiences trying to add various self-adaptive mechanisms to routines used in our non-symmetric eigenproblem solvers.

- *Case I: Hessenberg reduction*
  The performance of Hessenberg reduction is dependent on a parameter which sets a certain block size in the algorithm. We can argue qualitative that an appropriate configuration of this parameter should neither be too small nor too large. But precisely where the range of good values are depends on the system and its configuration as well as the size of the problem. In Section 2, we present a self-adaptive mechanism which is capable of finding good values *without prior training* with just a fraction of a *single* run.

- *Case II: Eigenvector computation*
  The performance of eigenvector computations given a Schur form is dependent on the size of a tile. From qualitative arguments we can reason that there is a trade-off between parallelism, computational efficiency, and idling overhead. Again, where precisely the range of good tile sizes is depends on the system and the problem characteristics. In Section 3, we present a self-adaptive mechanism in which the routine adapts its choice of tile size from one run to the next. The cost of the task and of the idling overhead are modeled with the aim of capturing the essence of how they depend on the tile size. The models are fitted to measurements made during previous runs and drive the adjustment of the tile size prior to the next run.

- *Case III: Schur reduction*
  In reduction to Schur form, one of the subroutines can become a sequential bottleneck. The negative impact of this can be reduced by parallelizing the subroutine. However, parallelization ought to be applied only when it is necessary or else the additional overhead will only compound the problem. In Section 4, we present a self-adaptive mechanism which uses run-time modeling of task completion rates and task execution times to make dynamic decisions on when to parallelize the problematic subroutine.

These case studies have some things in common. The overheads of the self-adaptive mechanisms in terms of both space and time are negligible to the point of being trivial. The mechanisms also rapidly find a range of good values (cases I and II) or learn to make informed decisions (case III).

2 Case I: Hessenberg Reduction

A general dense matrix $A \in \mathbb{R}^{n \times n}$ can be reduced to upper Hessenberg form $H = Q^T AQ$ using the cache-blocked algorithm first described in [3] and later slightly improved in [4]. In this algorithm, the reduction is carried out by constructing and applying a total of $n-2$ Householder reflections. Cache-efficiency is obtained by expressing most of the arithmetic (around 80% of the flops) in terms of level 3 BLAS (mostly `GEMM`, general matrix times general matrix) operations by using compact WY representations of products of reflectors. Almost all of the remaining 20% of the flops are accounted for by $n-2$ large matrix-vector multiplications (level 2 BLAS and hence memory-bound). The matrix-vector operations dominate the execution time even though they account for a minority of the flops. One can observe in practice that more than 80% of the execution time is due to matrix-vector multiplications [6].

For our present purpose, we need only a high-level understanding of the structure of the algorithm. The leading $n-2$ columns are partitioned into N column blocks of

width $b_i \geq 1$, $i = 1, 2, \ldots, N$, such that $\sum_{i=1}^{N} b_i = n - 2$. The column blocks are reduced to the correct shape one by one from left to right. The width of a block is crucial for performance. If we set it too small then the cache reuse will not be enough to make the level 3 BLAS operations (mostly `GEMM`s) efficient. But if we set it too large then the lower-order arithmetic overheads start to become significant.

The block reduction is performed using multi-threaded kernels. OpenMP is used as a parallel programming model.

2.1 Self-Adaptive Mechanisms

We use an implementation of the algorithm which has the ability to dynamically choose b_{i+1} in the time interval between iterations i and $i + 1$. We present two self-adaptive mechanisms which adapt the block size between iterations in one run. These are based on a few general observations.

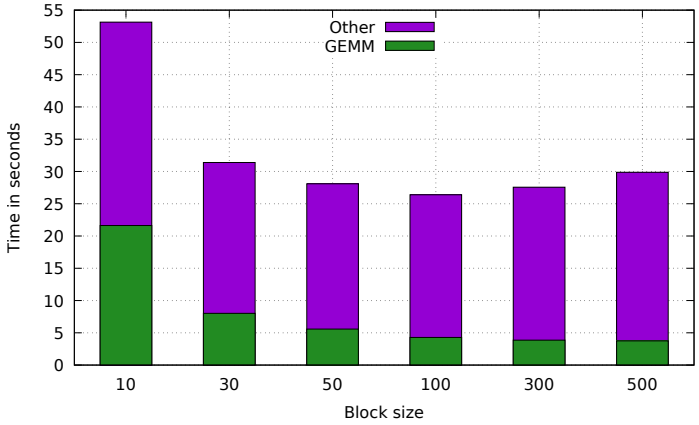


Figure 1: Illustration of the portion of the total execution time for Hessenberg reduction attributable to `GEMM` operations and other operations, respectively. The matrix dimension is $n = 10000$.

The first observation. If we choose all block sizes b_i to be small but still large enough to yield an acceptable `GEMM` performance, then the resulting performance tends to be acceptable as well. Figure 1 provides support for this observation. We see the portion of the execution time which is attributable to `GEMM` operations and other operations, respectively, for a variety of block sizes. The `GEMM` portion decreases as the block size increases but plateaus for $b \gtrapprox 100$. This is in the same vicinity where the total execution time is at a minimum. Increasing the block size even further increases the overhead caused by other operations and the total execution time increases. As a rule of thumb: the block size should be set large enough to give good `GEMM` performance but small enough to not suffer from overheads in the other operations.

The second observation. In the ith iteration, the `GEMM` performance tends to increase and eventually plateau when viewed as a function of the block size b_i. Figure 2 provides support for this observation. We see the `GEMM` performance in the ith iteration as a function of the block size b_i for $i = 1, 2, 4, 8, 16$. (The preceding block sizes b_j for $1 < j < i$ were set to $b = 10$, any other value will do as long as at iteration i we have at least b_i columns to reduce.). In all cases, the performance increases roughly monotonically and eventually plateaus. We take advantage of the general shape of the `GEMM` performance

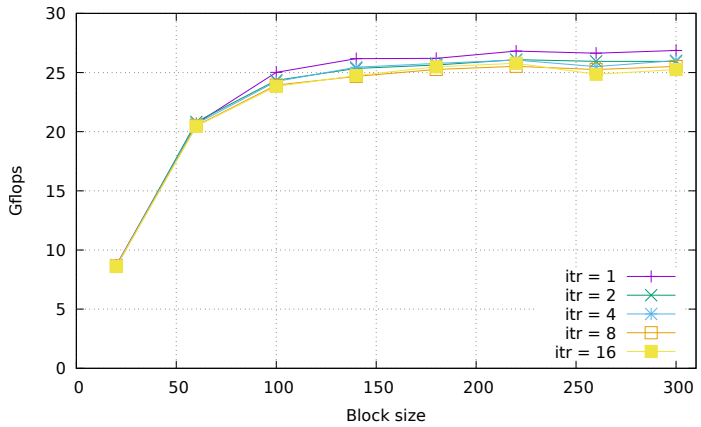


Figure 2: Illustration of the GEMM performance in the ith iteration of Hessenberg reduction as a function of the block size. The matrix dimension is $n = 10000$.

in our self-adaptive mechanisms. Specifically the facts that it increases and eventually plateaus.

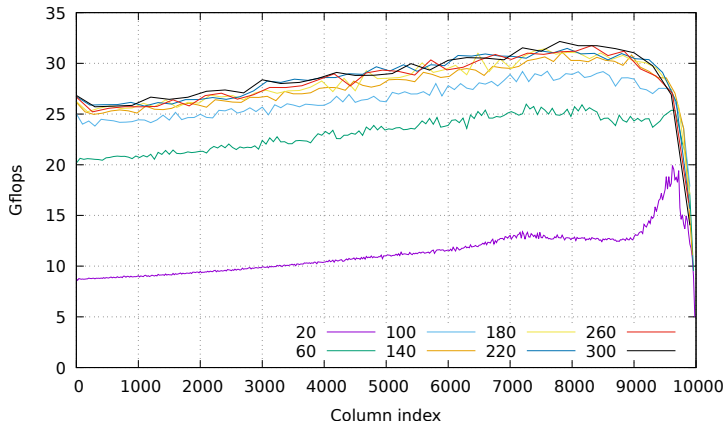


Figure 3: Illustration of how the GEMM performance per iteration varies from one iteration to the next for a variety of block sizes. The x value ("column index") is the first column index of the unreduced part at the start of the iteration. The matrix dimension is $n = 10000$.

The third observation. There is a strong correlation between how the GEMM performs in one iteration and how it performs in the next, particularly in the first half of the iterations. Figure 3 supports this observation by showing how the GEMM performance per iteration varies from start to finish for a variety of block sizes. We see that the performance is relatively constant in the first half with a slight upward drift. We use this observation in our self-adaptive mechanisms to deduce an appropriate value for b_{i+1} based on measurements obtained in the previous iteration using b_i. This gives an ability to rapidly improve the block size from one *iteration* to the next rather than from one *run* to the next.

The rest of this section describes the two self-adaptive mechanisms and Section 2.2 presents some experimental result.

2.1.1 Mechanism A

Mechanism A initializes b_1 randomly and selects block size b_{i+1} in the time interval between iterations i and $i+1$. The algorithm is essentially a finite state machine with three states (see Figure 4) and transitions are triggered by comparing measurements of `GEMM` performance in the two immediately preceding iterations. The adaptivity is disabled after half the matrix has been processed, in part motivated by the fact that the third observation (recall Section 2.1) breaks down close to the end.

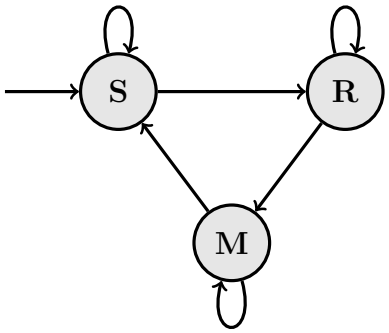


Figure 4: Self-adaptive mechanism A as a finite state machine.

The search state. The S-state (the "search state") is the initial state. While in this state the algorithm rapidly increases the block size in an attempt to find the first instance of a block size which provides acceptable `GEMM` performance. The real-valued parameter $\alpha > 1$ controls the rate at which the block size is increased. Consider the transition from iteration i to $i+1$. Let g_i denote the observed (measured) `GEMM` performance in the ith iteration. If $i = 1$ we have only one prior measurement, so we unconditionally set $b_2 = \alpha b_1$ to get a second measurement. Otherwise ($i \geq 2$) we have measurements of both g_{i-1} and g_i and we begin to estimate the rate of change which we define as

$$\eta_i = \left| \frac{g_i - g_{i-1}}{b_i - b_{i-1}} \right|. \tag{1}$$

If $\eta_i < \theta_s$, where θ_s is a real-valued positive parameter close to zero, this indicates a small (positive or negative) rate of change from b_{i-1} to b_i. We interpret this as an indication that the two points are in the plateau region of the `GEMM` performance profile (recall Figure 2 and the second observation). In this way, the parameter θ_s indirectly defines the "plateau"; the smaller the value of θ_s the more flat the curve must be. The algorithm transits to the R-state if $\eta_i < \theta_s$ and sets b_{i+1} as if the machine transitioned from the R-state back to the R-state. Otherwise it sets $b_{i+1} = \alpha b_i$ and remains in the S-state.

The refinement state. The R-state (the "refinement state") aims to slowly reduce the block size from a known acceptable value to a smaller value which is also acceptable. A real-valued parameter $r > 1$ controls the rate of the decrease, which is multiplicative with scaling factor

$$\beta = \alpha^{-1/r} \in (0, 1).$$

The rate of change in the performance is calculated using (1). The algorithm transits to the M-state if $\eta_i > \theta_r$, where θ_r is a real-valued positive parameter close to zero, and sets

$b_{i+1} = b_{i-1}$. Otherwise it sets $b_{i+1} = \beta b_i$ and remains in the R-state. Noisy measurements can easily trigger the condition $\eta_i > \theta_r$ prematurely. To reduce the sensitivity to noise we actually require the condition to be triggered *twice in succession* before making a transition. When the condition is triggered for the first time the algorithm will change neither the state nor the block size before the next iteration.

The monitoring state. The M-state (the "monitoring state") aims to keep the block size fixed as long as there are no reasons to believe that changing the block size can improve performance. The block size b_{i+1} is set to b_i and the expected performance, g_{exp}, is estimated by an exponential moving average to account for the upward drift (see Figure 3 and the third observation). As long as the relative performance difference is small, i.e.,

$$\left| \frac{g_i - g_{\text{exp}}}{g_{\text{exp}}} \right| < \epsilon, \tag{2}$$

for a small and positive real-valued parameter ϵ, the algorithm remains in the M-state. Noisy measurements are handled similarly to the R-state. Three consecutive iterations must violate (2) before the algorithm transits to the S-state.

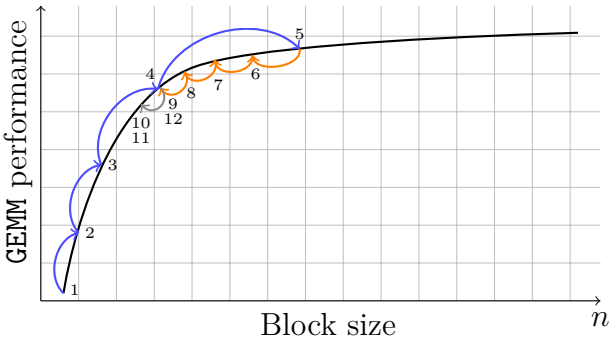


Figure 5: Sketch of how mechanism A explores an idealized `GEMM` performance profile. The block size is increased in the S-state (blue), decreased in the R-state (orange), and transits to the M-state after detecting a significant and robust drop in performance (gray).

Figure 5 sketches the behavior of mechanism A. The mechanism starts in the S-state using a randomly selected value for the first block size. In this state, the mechanism multiplicatively increases the block size (blue arrows above the performance profile) until reaching the plateau at iteration 5. The mechanism then transits to the R-state, where it multiplicatively decreases the block size using short jumps (orange arrows under the performance profile). At the 10^{th} and 11^{th} iterations (gray arrow under the performance profile), the mechanism detects that further reduction of the block size degrades the performance rapidly, so it returns (in the 12^{th} iteration) to the previous block size from iteration 9 and transits to the M-state.

2.1.2 Mechanism B

Mechanism B is similar to A. We found that mechanism A would get stuck at inappropriately small block sizes or use way too large ones after being fooled by an unfortunate sequence of noisy measurements. The vulnerable parts of mechanism A are the state transits. Requiring the triggering condition to be violated two or three times in succession

helps alleviate the problem to some extent but not completely. This led us to consider an alternative mechanism which cannot get stuck and does not explode as readily.

The biggest difference between the two mechanisms is that mechanism B lacks states. The choice of b_{i+1} is made on the basis of the (signed) rate of change in performance from iteration $i-1$ to i, defined as

$$\eta_i = \frac{g_i - g_{i-1}}{b_i - b_{i-1}}.$$

The algorithm starts with a randomly selected b_1 and sets $b_2 = \alpha b_1$, where $\alpha > 1$ is a real-valued parameter controlling the rate of increase. After iteration $i \geq 2$, the algorithm checks the condition

$$\frac{g_i + \eta_i b_i}{g_i} \geq \theta, \tag{3}$$

where $\theta > 1$ is a real-valued parameter close to one.[1] If the condition is satisfied, then the block size is increased to $b_{i+1} = \alpha b_i$. Otherwise the block size is reduced, since a low rate of change is indicative of reaching the plateau (see Figure 5).

The scheme we use is actually slightly more elaborate than simply a multiplicative decrease. Using only multiplicative decreases we discovered a phenomenon where the algorithm would be drawn to very large block sizes due to a sequence of noisy measurements. The algorithm keeps a partial history of the ℓ previous measurements/iterations. When the condition (3) is not satisfied the algorithm looks through its history and picks the smallest block size b_s smaller than b_i whose performance g_s satisfies $g_s > \gamma \cdot \max\{g_{i-1}, g_i\}$ where $\gamma < 1$ is a real-valued parameter close to one. If such a block size exists, then the algorithm decreases the block size to $b_{i+1} = b_s$. Otherwise it decreases the block size to $b_{i+1} = \alpha^{-1} b_i$.

2.2 Experiments

We experimentally compared the two mechanisms in Sections 2.1.1 and 2.1.2. We used one compute node (in exclusive mode) of the Kebnekaise system at HPC2N, Umeå University. A node contains 28 Intel Xeon E5-2690v4 cores with a total of 128 GB of main memory. We used one thread per core (and all 28 cores) unless otherwise mentioned.

For mechanism A we set the parameters to $\alpha = 1.5$, $\theta_{\mathrm{s}} = 0.05$, $\theta_{\mathrm{r}} = 0.035$, $r = 1.40942084$, and $\epsilon = 0.02$. For mechanism B we set the parameters to $\alpha = 1.2$, $\theta = 1.1$, and $\gamma = 0.95$.

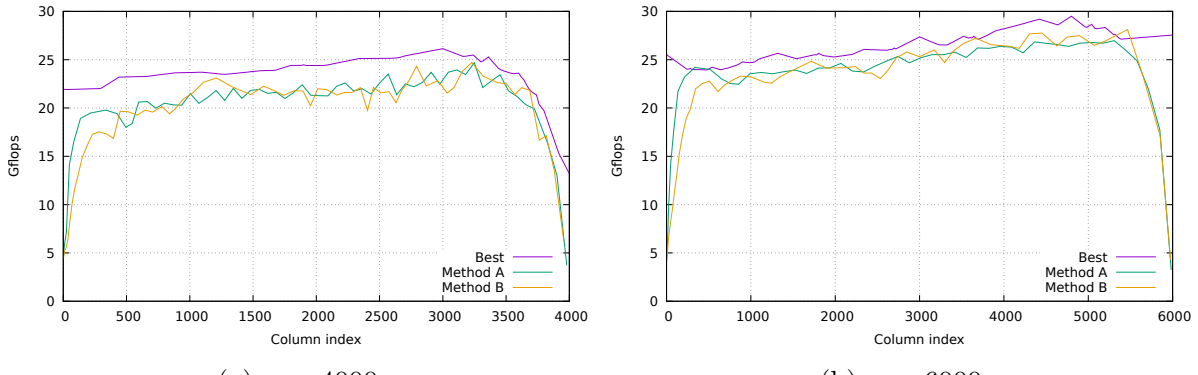
Figure 6 shows the `GEMM` performance per iteration for mechanisms A and B (starting at $b_1 = 10$) compared to the best observed performance for the fixed block sizes

$$b \in \{20, 60, 100, 140, 180, 220, 260, 300\}.$$

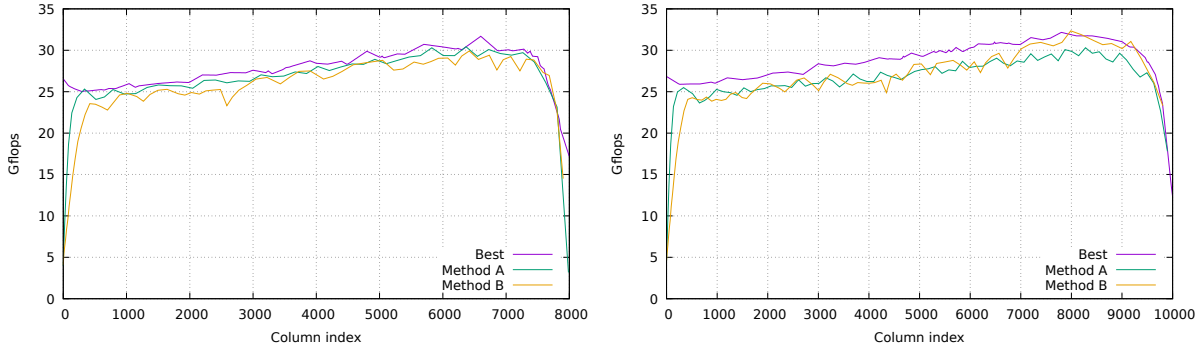
Both mechanisms are able to quickly (after just a few iterations in *one* run) locate and stay at a range of block sizes which give acceptable (close to the best observed) performance. The block sizes corresponding to the data in Figure 6 are shown in Figure 7.

Figure 8 shows the ratio of total execution time using mechanisms A and B to the best observed total execution time for the same experiment in Figures 6 and 7. Both mechanisms in a single run reached total execution time very close (less than 10% slower) to the best observed time using fixed block sizes.

[1] The left-hand side is an estimate of the relative improvement in performance obtained by doubling the block size. The estimate is based on a linear extrapolation using the current performance and the estimated rate of the change.

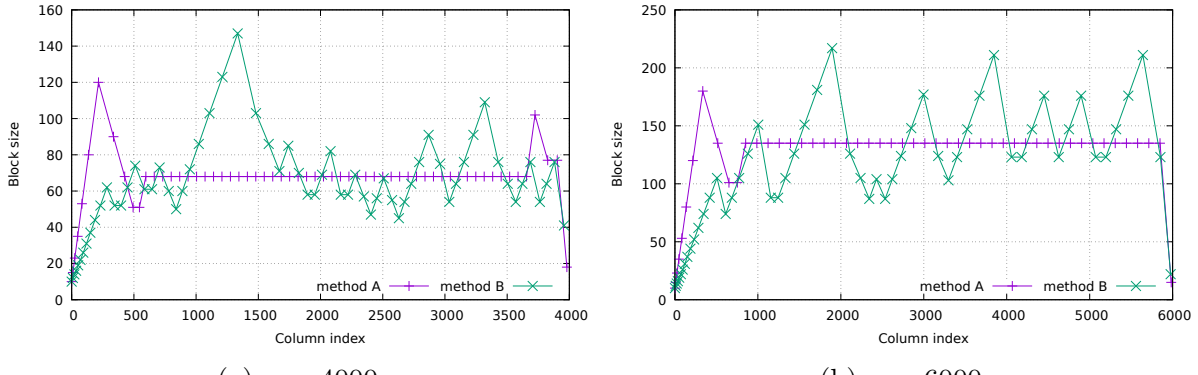


(a) $n = 4000$ (b) $n = 6000$

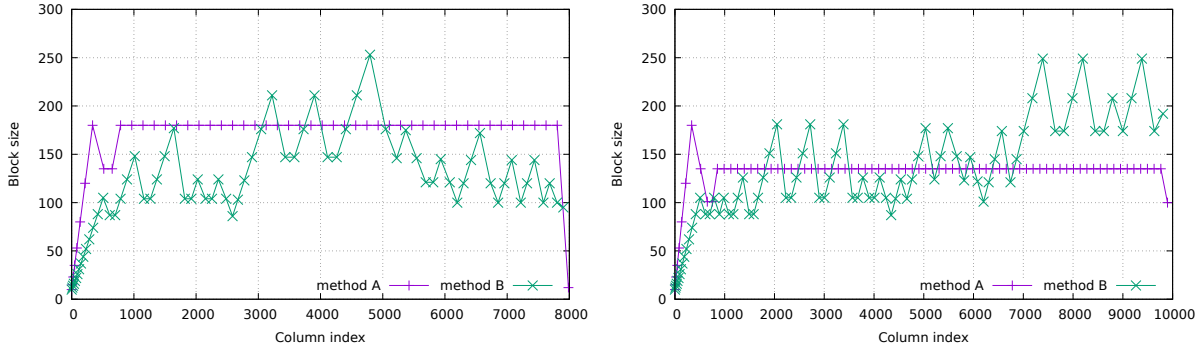


(c) $n = 8000$ (d) $n = 10000$

Figure 6: The `GEMM` performance per iteration for mechanisms A and B compared to runs using various fixed block sizes.



(a) $n = 4000$ (b) $n = 6000$



(c) $n = 8000$ (d) $n = 10000$

Figure 7: The block sizes used by mechanisms A and B for the data in Figure 6.

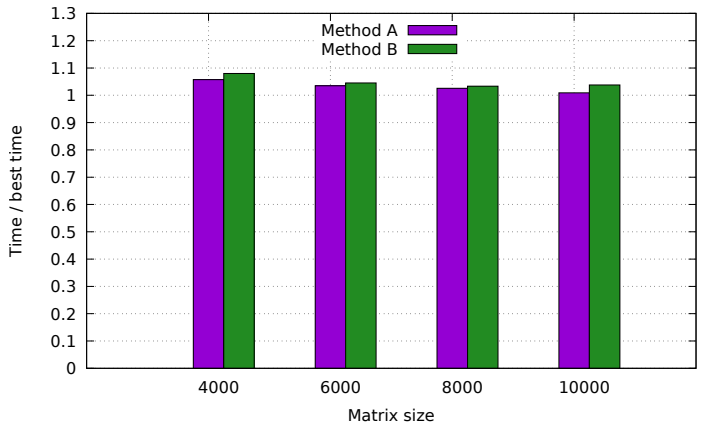


Figure 8: Ratio of total execution time using mechanisms A and B to the best total execution time for different problem sizes.

3 Case II: Eigenvector Computation

Consider the task of computing eigenvectors for the last m eigenvalues in a given real Schur form $S = Q^T AQ$ of size $n \times n$ (a special case of [5]) on a shared-memory system of p cores. The input consists of S and m and the output is a matrix of size $n \times m$ whose columns are eigenvectors of S associated with the last m eigenvalues on the diagonal of S. The algorithm internally partitions S into tiles of size $b \times b$. The tile size is a tunable parameter set to maximize performance. However, unlike the previous case in Section 2, the tile size cannot be modified during a run. The routine must therefore adapt itself from one run to the next.

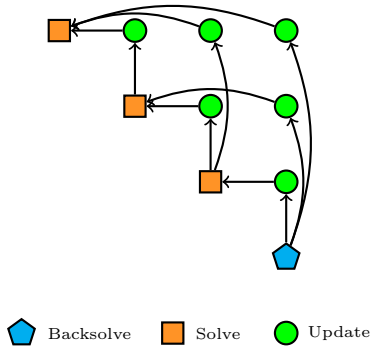


Figure 9: Illustration of the task graph for eigenvector computation when $n/b = 4$.

The algorithm for computing eigenvectors can be expressed as a task graph G_b which is dependent on b. Figure 9 illustrates the shape of G_b for $n/b = 4$. There are three type of tasks: *backsolve* (blue pentagon), *solve* (orange square), and *update* (green circle). There is one task per $b \times b$ tile in the upper triangular part of S. Thus, a small b implies a large number of tasks and vice versa. The total arithmetic cost is essentially independent of b, so if G_b has many tasks they are necessarily also fine-grained. A large tile size conversely implies a graph with only a few coarse-grained tasks and the resulting low degree of concurrency will give rise to a significant idling cost. Optimizing the performance therefore boils down to finding a balance between efficient tasks on the one hand and low idling cost on the other.

3.1 Self-Adaptive Mechanism

The self-adaptive mechanism we describe below constructs simple models of the task execution times from which we derive models of two prominent components of the parallel cost: the total task cost and the idling cost. The models are adapted after each run based on measurements recorded in all previous runs. The aim of the modeling is to capture the long-range behavior (e.g., decreasing and then increasing) rather than the short-range variations (e.g., high-frequency oscillations).

The execution times of tasks of each type are modeled as second-degree polynomial functions of b. Specifically,

$$\omega_{\mathrm{bs}}(b;\alpha_0,\alpha_1,\alpha_2) = \alpha_2 b^2 + \alpha_1 b + \alpha_0 \qquad \text{models backsolve tasks,} \tag{4}$$

$$\omega_{\mathrm{s}}(b;\beta_0,\beta_1,\beta_2) = \beta_2 b^2 + \beta_1 b + \beta_0 \qquad \text{models solve tasks, and} \tag{5}$$

$$\omega_{\mathrm{u}}(b;\gamma_0,\gamma_1,\gamma_2) = \gamma_2 b^2 + \gamma_1 b + \gamma_0 \qquad \text{models update tasks.} \tag{6}$$

There is only one backsolve task, so its contribution to the cost is negligible. Similarly, there are only $\mathcal{O}(N)$ solve tasks but as many as $\mathcal{O}(N^2)$ update tasks, where $N = n/b$. We therefore ignore also the cost of the solve tasks. We end up modeling the total task cost by

$$C_{\mathrm{task}}(b;\gamma_0,\gamma_1,\gamma_2) = \underbrace{\frac{N^2-N}{2}}_{\#\text{ of tasks}} \cdot \underbrace{\omega_{\mathrm{u}}(b;\gamma_0,\gamma_1,\gamma_2)}_{\text{time per task}}. \tag{7}$$

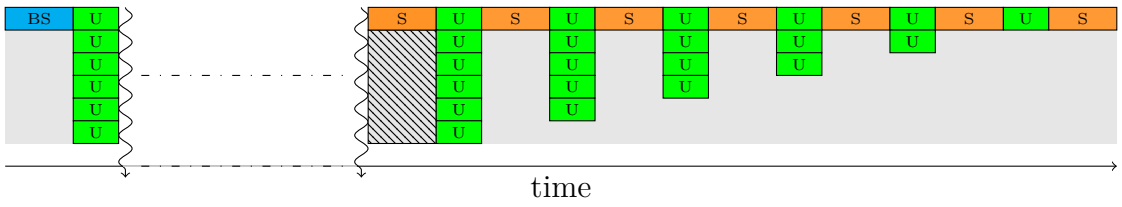


Figure 10: Sketch illustrating idling at the beginning and end of eigenvector computation. The letters identify the task types: backsolve (BS), solve (S), and update (U).

The task graph does not allow for any schedule with zero idling cost. Initially there is merely one ready task (the backsolve). During its execution, one core will be busy and the remaining $p-1$ cores will idle. Suppose $N > p$. After completing the backsolve, $N-1 \geq p$ update tasks become ready (see Figure 9). This is sufficient to make all cores busy immediately after the completion of the backsolve task. This initial phase of the computation is sketched on the left side of Figure 10.

The path in the graph with the most tasks is of the form

$$\mathrm{BS}(\rightarrow \mathrm{U} \rightarrow \mathrm{S})^{N-1},$$

where $(x)^y$ is interpreted as "x repeated y times". Hence, the N backsolve or solve tasks are indirectly serialized. Immediately after completion of the kth backsolve or solve task, $k = 1, 2, \ldots, N$, the number of ready tasks is bounded above by

$$k(N-k). \tag{8}$$

(In practice, far fewer tasks tend to be ready at this point.) Consider the situation after completing the kth backsolve or solve task where $k = N - p$. According to (8) there are

at most $p(N-p)$ ready tasks (all of them are update tasks), of which exactly p tasks were unlocked by the kth backsolve or solve task. Suppose the p fresh ready tasks are the *only* ready tasks at this point and that all cores are presently idle (which is likely the case when the tile size is large). What remains of the task graph can then be scheduled optimally as illustrated on the right side of Figure 10. By calculating the area corresponding to idling we obtain the following model for the idling cost:

$$C_{\text{idle}}(b;\beta_0,\beta_1,\beta_2,\gamma_0,\gamma_1,\gamma_2) = p(p-1)\cdot\omega_{\text{s}}(b;\beta_0,\beta_1,\beta_2) + \frac{p(p-1)}{2}\cdot\omega_{\text{u}}(b;\gamma_0,\gamma_1,\gamma_2). \quad (9)$$

We stress that this is by no means an *accurate* model of the true idling cost. The model does, however, capture that idling cost grows with b and that the amount of idling is closely related to the execution times for certain types of tasks.

The total cost is modeled as the sum of the task cost (7) and the idling cost (9):

$$C(b) = C_{\text{task}}(b) + C_{\text{idle}}(b). \quad (10)$$

The self-adaptive mechanism can be summarized as follows:

1. Sample a few tile sizes until enough data is available to bootstrap the models.
2. Update the models by fitting them to the available data.
3. Choose a tile size that minimizes Eq. (10) [2] for the next run.

These steps are described in more detail below (in the order 2, 3, 1).

3.1.1 Model fitting

The solve and update tasks are such that their *pace* (time per elementary operation) tends to monotonically decrease with b. In other words, these tasks tend to become more efficient when we increase the tile size. In particular, the pace for an update task is

$$P_{\text{u}}(b;\gamma_0,\gamma_1,\gamma_2) = \frac{\omega_{\text{u}}(b;\gamma_0,\gamma_1,\gamma_2)}{b^2} = \gamma_2 + \frac{\gamma_1}{b} + \frac{\gamma_0}{b^2}.$$

(The pace for a solve task is defined analogously.) Non-negativity constraints on γ_i ensure that P_{u} is a non-increasing function with γ_2 as the limit when $b \to \infty$.

The task models are re-fitted after each run in order to make use of the new measurements. Let $s_k = (b_k, t_k)$, $k = 1, 2, \ldots, s$, denote the sequence of observations made thus far. For the kth run, b_k denotes the tile size used and t_k denotes the mean of the observed update task execution times. Using linear least squares regression, we determine the parameters γ_i which minimizes the model error

$$E(\gamma_0,\gamma_1,\gamma_2) = \sum_{k=1}^{s}\left(\frac{t_k}{b_k^2} - P_{\text{u}}(b_k;\gamma_0,\gamma_1,\gamma_2)\right)^2$$

subject to non-negativity constraints on γ_i. (An analogous procedure is used to fit the model ω_{s}.)

[2] Minimizing execution time is equivalent to minimizing cost when the number of cores is constant.

Since we have assumed the pace to be non-increasing, we can filter out outliers by fitting the model only to points on the lower convex hull of the point set

$$\{(b_k, t_k/b_k^2)\}_{k=1}^{s}.$$

The set of observations grows over time. Therefore, any point found to lie above the lower convex hull will remain above the convex hull forever. We can save space and time by simply discarding all such points. If the same tile size is sampled several times, then only the measurements corresponding to the run with the shortest execution time are kept[3].

3.1.2 Tile size selection

Refinements of the task models propagate to the task cost model (7), the idling cost model (9), and ultimately to the full cost model (10). The tile size to use in the next run is selected as one that minimizes the cost model $C(b)$ rounded to the nearest integer.

3.1.3 Initialization

With three-parameter models we need to sample at least three tile sizes before we have enough data to create a first fit. These initial samples are randomly chosen from the set

$$\{k \in \mathbb{N} : m \leq k \leq n/p\},$$

where m is the smallest possible tile size and n/p is the largest tile size which affords at least p tiles in either dimension.

3.2 Experiments

We experimentally evaluated the self-adaptive mechanism on one compute node (in exclusive mode) of the Kebnekaise system (refer back to Section 2.2 for details). We used OpenMP tasks as a runtime system. OpenMP performs task-based parallelization by scheduling ready tasks to available cores. We compare two variants of the code: one that uses the self-adapting mechanism (the *adaptive* variant) and one that does not (the *regular* variant). All kernels in the code are sequential, i.e., performed using a single thread. We assigned one thread to each core.

To enable vectorization in the code, we round b to the nearest multiple of 4. We also never allow tile sizes such that $b < m$ because it would give rise to a differently shaped task graph. We used $m = 32$ for all experiments. Furthermore, to ensure that all tiles are of size $b \times b$, we round n up to the nearest multiple of b. We noticed that the first run tended to be slower than subsequent runs. We therefore repeated each run twice and recorded measurements only for the fastest of the two.

Figure 11 relates measurement-based costs with model-based costs for $n = 30000$ (rounded up to the nearest multiple of b). The regular variant was run over a sweep of tile sizes. For each run, we recorded the total cost (blue solid line), the total task cost (green solid line) and the mean task execution times per task type. The idling cost cannot be measured directly (cannot separate idling cost from overhead). We instead estimate the idling cost for each tile size using the formula (9) with the models ω_s and ω_u replaced with the corresponding *measured* mean task execution times.

[3]Total execution time is used as a selection criteria to keep the measurements of a single run coherent.

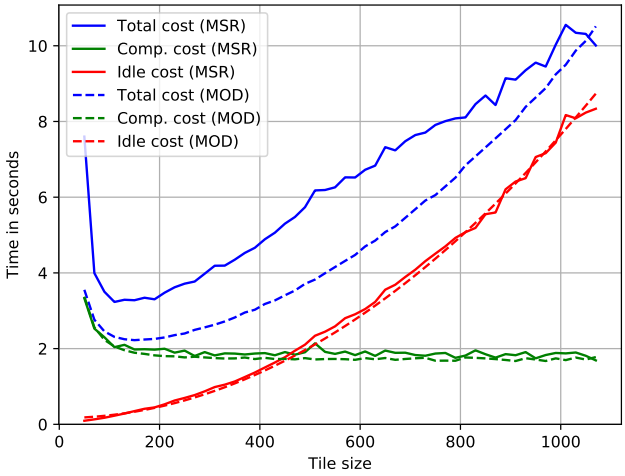
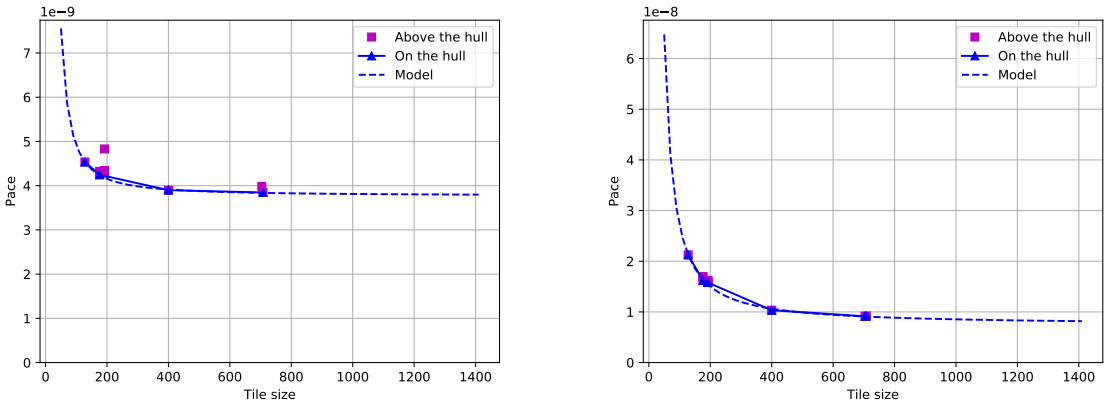


Figure 11: Comparison between measurement-based (MSR) costs and model-based (MOD) costs for $n = 30000$ (rounded up to the nearest multiple of b).

The adaptive variant was run a total of $3 + 10$ times. The final models of task cost, idling cost, and total cost are shown as dashed lines in Figure 11. The task cost model is able to capture the long-range behavior fairly well. The small variations observed in the data are effectively filtered out by the model. The full cost model has a large absolute error, mostly because it does not account for all components of the cost. On the other hand, the model captures the overall shape of the total cost and most importantly it accurately locates the good tile sizes.

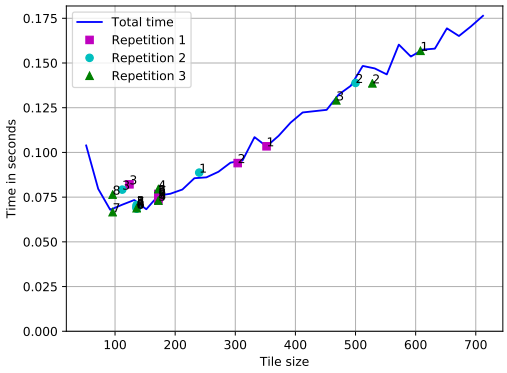
Figure 12 shows all observed points $(b_k, t_k/b_k^2)$ for the solve and update tasks in the adaptive variant, the lower convex hulls, and the corresponding models (5) and (6) (converted from time to pace). The matrix sizes were $n = 40000$ rounded up to the nearest multiple of b.



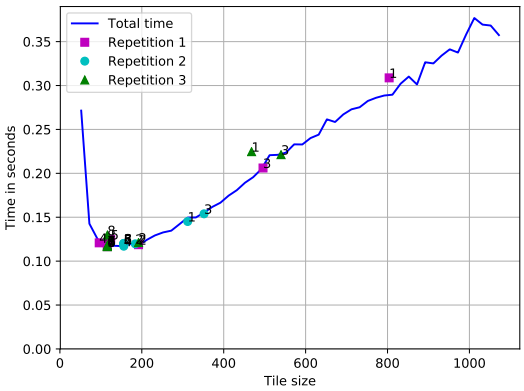
(a) Update tasks.

(b) Solve tasks.

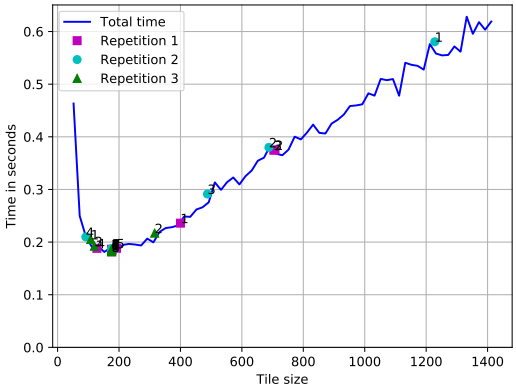
Figure 12: The observed and modeled paces for the update and solve tasks for matrices of size $n = 40000$ rounded up to the nearest multiple of b. The observations $(b_k, t_k/b_k^2)$ are shown together with the lower convex hulls and the corresponding task execution time model (converted from time to pace).



(a) $n = 20000$



(b) $n = 30000$



(c) $n = 40000$

Figure 13: Results of the self-adaptive mechanisms' search for an appropriate tile size. Each figure shows a different matrix size (rounded up to the nearest multiple of b) and contains three repetitions (different initialization points). There are a total of $3+5$ points per repetition.

Figure 13 shows how the adaptive variant searches for an appropriate tile size. For each matrix size, the progression of the search for three repetitions (with different initial points) are shown. The total execution time of the regular variant is included for comparison (solid line). The mechanism very quickly (immediately after the random initialization) locates and then remains at acceptable tile sizes.

4 Case III: Schur Reduction

All eigenvalues of a matrix $A \in \mathbb{R}^{n \times n}$ can be computed by a reduction to real Schur form via an intermediate Hessenberg form. Specifically, one first reduces A to upper Hessenberg form $H = Q_1^T A Q_1$ by an orthogonal similarity transformation. Then one applies the small-bulge multi-shift QR algorithm with aggressive early deflation (AED) [1, 2] to H to obtain a real Schur form $S = Q_2^T H Q_2$ with 1×1 and 2×2 blocks on the main diagonal. The QR algorithm with AED has two computationally expensive components: aggressive early deflation and bulge chasing.

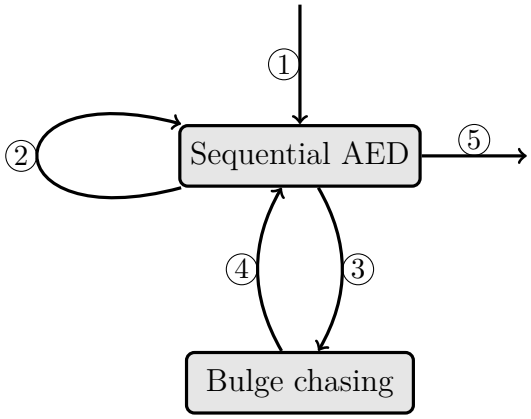


Figure 14: Simplified control-flow graph for the QR algorithm with AED. 1) The algorithm starts with AED. 2) AED can be followed by another AED, typically when many eigenvalues were deflated. 3) AED generates shifts for a subsequent bulge chasing. 4) Bulge chasing is followed by AED. 5) The algorithm terminates after AED deflates all remaining eigenvalues.

Figure 14 shows a simplified control-flow graph for the QR algorithm with AED. The algorithm starts with AED, which may detect and deflate zero or more converged eigenvalues. The algorithm terminates if the AED deflates the problem down to nothing. Otherwise the algorithm chooses to either perform another AED or one round of bulge chasing. (This decision is typically made on the basis of how many eigenvalues were deflated.) Bulge chasing is always followed by AED.

Figure 15 illustrates a typical iteration of the QR algorithm with AED. AED is performed on a trailing principal submatrix referred to as the *AED window*. The submatrix inside the window is reduced to Schur form by a recursive application of the QR algorithm with AED. When the corresponding transformation is applied to the whole matrix, a spike emerges immediately to the left of the AED window (by engaging a non-zero subdiagonal entry from the left). If the last element of the spike is sufficiently small, then it can be truncated to zero, which deflates the problem into a smaller one. Otherwise, the corresponding eigenvalue candidate is not considered converged and is moved out of the way.

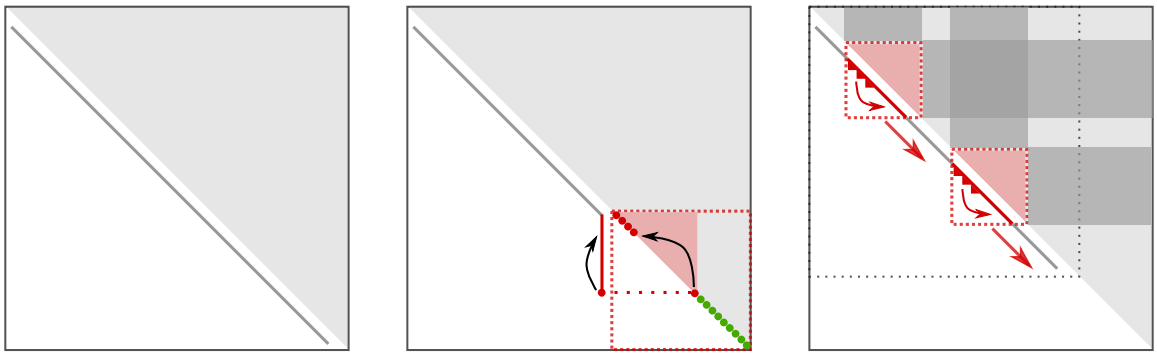


Figure 15: Illustration of one iteration of the QR algorithm with AED. Left: The Hessenberg matrix H. Middle: The AED window and its associated spike. Converged eigenvalues shown in green and failed eigenvalue candidates (potential shifts) shown in red. Right: Bulge chasing of six bulges spread over two bulge-chasing windows.

After systematically checking and reordering all eigenvalue candidates in the AED window, we will have identified zero or more converged eigenvalues (green) and zero or more potential shifts for bulge chasing (red). The spike is finally eliminated by a Hessenberg reduction.

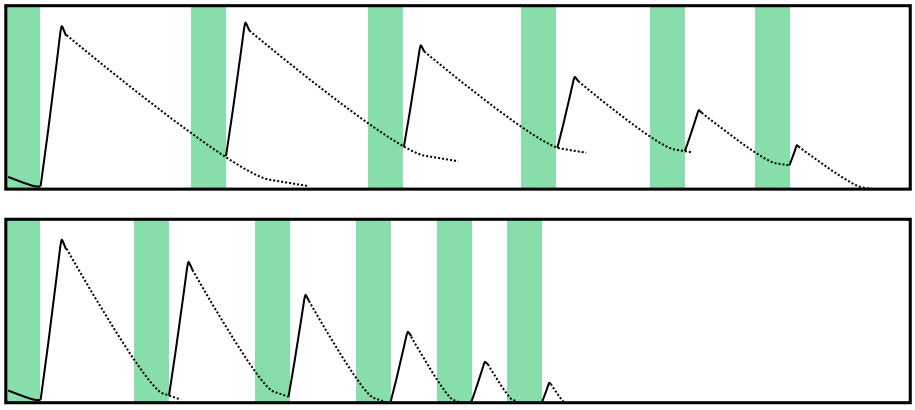
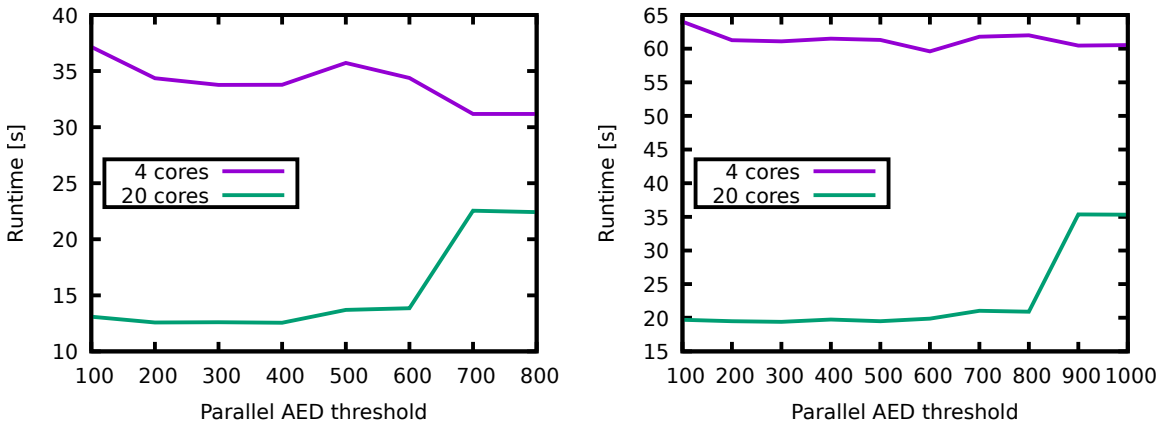


Figure 16: Illustration of why AED is sometimes a bottleneck (bottom) and other times not a bottleneck (top) in a task-based implementation of the QR algorithm. Time flows along the horizontal axes and the number of tasks known to the runtime system is on the vertical axes. The time intervals during which a (sequential) AED is performed are shaded green. Top: There are enough tasks available during all AEDs to keep the other cores busy. Bottom: The number of available tasks falls below the level required to keep all cores busy.

If performed sequentially, AED can quickly limit the strong scalability of the algorithm as illustrated in Figure 16. Each AED is followed by a sudden spike in the number of tasks (vertical axes) due to the rapid insertion of bulge-chasing tasks. The next AED transitively depends only on a subset of the bulge-chasing tasks. Therefore, an AED task can run on one core while the rest of the cores are still busy with bulge-chasing tasks. If no core runs out of (bulge-chasing) tasks during AED, then AED is not hampering the performance (at least not locally). This case is sketched in Figure 16 (top). Otherwise, one ore more cores will become idle due to a lack of bulge-chasing tasks during AED; see Figure 16 (bottom).



(a) $n = 8000$ (b) $n = 10000$

Figure 17: The runtime of the QR algorithm as a function of the parallel AED threshold.

Note that AED can itself be parallelized. But when does it make sense to run AED in parallel? The answer is simple if we do *not* allow bulge chasing and AED to overlap. In this case, *any* speedup of AED directly translates into speedup of the whole algorithm, so AED should run in parallel if and only if it can be done with some speedup. The question gets more complicated as soon as we allow overlap. A naive implementation would impose a *parallel AED threshold* that defines when an AED is large enough to perform in parallel. Only AEDs with a window size above the threshold would be processed in parallel. As shown in Figure 17, the tricky part is to find a single threshold that would work with a large range of matrix sizes and CPU core counts. The optimal threshold is different for four cores compared to 20 cores. In addition, the behavior changes with the problem size.

In order to understand what is happening, suppose the overlap is complete, i.e., no idling during AED. In this situation, running AED in parallel would likely be unwise since it will add overhead to the total cost without any gain (at least not locally). On the other hand, if there is zero overlap or only a partial overlap, then running AED in parallel would likely improve performance (at least locally) by reducing the idling overhead. We present in Section 4.2 and evaluate in Section 4.3 a self-adaptive mechanism which dynamically decides when to run AED in parallel by trying to answer the question: "Will the overlap of the next AED be partial?"

4.1 A Simple Model of Overlap

A simple model helps us gain insight into how the critical path and the amount of overlap combine to limit the potential speedup. Consider a fixed task graph for the QR algorithm with AED. Let C denote the total computational cost of this graph. Some portion, αC where $\alpha \in (0, 1]$, of the cost is on the critical path. Assuming that all tasks run sequentially, then the parallel execution time is bounded below by

$$T_p \geq \alpha C. \tag{11}$$

Consider the union of all time intervals in which a critical task is being executed. At any moment there are $p - 1$ cores who are not executing a critical task. The corresponding cost is $\alpha(p-1)C$. Some portion, $\alpha\beta(p-1)C$ where $\beta \in [0, 1]$, of this cost is accounted for by computation. The rest, $\alpha(1 - \beta)(p - 1)C$, is idling cost. The parallel execution time

is therefore bounded below also by

$$T_p \geq (1 + \alpha(1-\beta)(p-1)) \cdot \frac{C}{p}. \tag{12}$$

Combining (11) and (12) we get

$$T_p \geq \underbrace{\max\{\alpha p, 1 + \alpha(1-\beta)(p-1)\}}_{\gamma_{\alpha,\beta,p}} \cdot \frac{C}{p} = \gamma_{\alpha,\beta,p} \cdot \frac{C}{p}. \tag{13}$$

Note that C/p is the ideal execution time corresponding to zero overhead and $\gamma_{\alpha,\beta,p} \geq 1$ measures how far from the ideal case the lower bounds are. As a consequence of (13), the parallel speedup will be bounded above by

$$S_p = \frac{C}{T_p} \leq \frac{p}{\gamma_{\alpha,\beta,p}}. \tag{14}$$

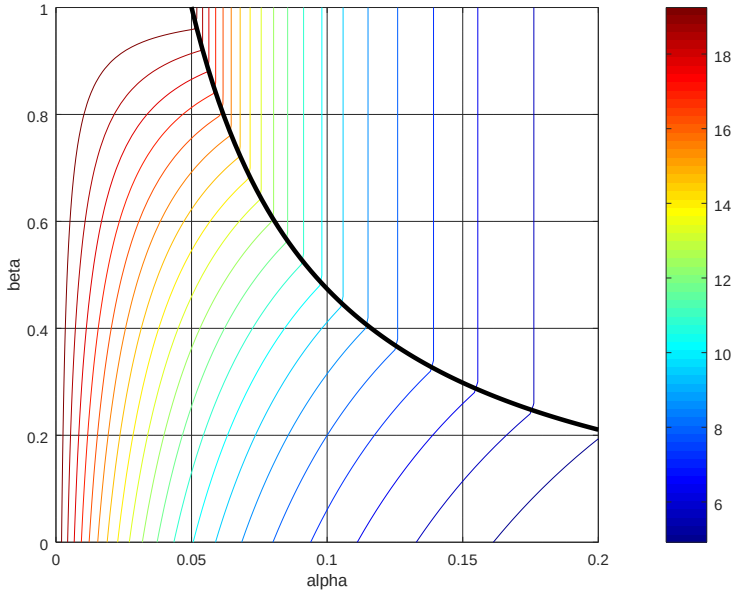


Figure 18: Contours of $p/\gamma_{\alpha,\beta,p}$ from the speedup bound (14) for $p = 20$. The thick black curve demarcates the boundary between the two cases generated by the max in (13).

Figure 18 illustrates the bound (14) as a function of α and β for $p = 20$. The thick black curve demarcates the boundary between the two cases of the max in (13). The critical path is the limiting factor above the curve, and computational cost and lack of overlap are the limiting factors below.

Suppose AED and bulge chasing do not overlap at all. This situation corresponds to β being small since AED typically accounts for the majority of the cost of the critical path. According to Figure 18, the speedup will be sensitive to the relative cost of the critical path (represented by α).

4.2 Self-Adaptive Mechanism

Since the bulge-chasing tasks are similar to one another we expect them to have roughly the same execution times. As a consequence, the task pool will deplete at a rate proportional to the number of tasks executed at the same time (which is p or less). As long

as the number of ready tasks is greater than or equal to p, the size of the task pool will decrease at a constant rate. When the task pool begins to dry up, fewer and fewer tasks will be able to run concurrently and as a result the size of the task pool will start to decrease at a slower rate.

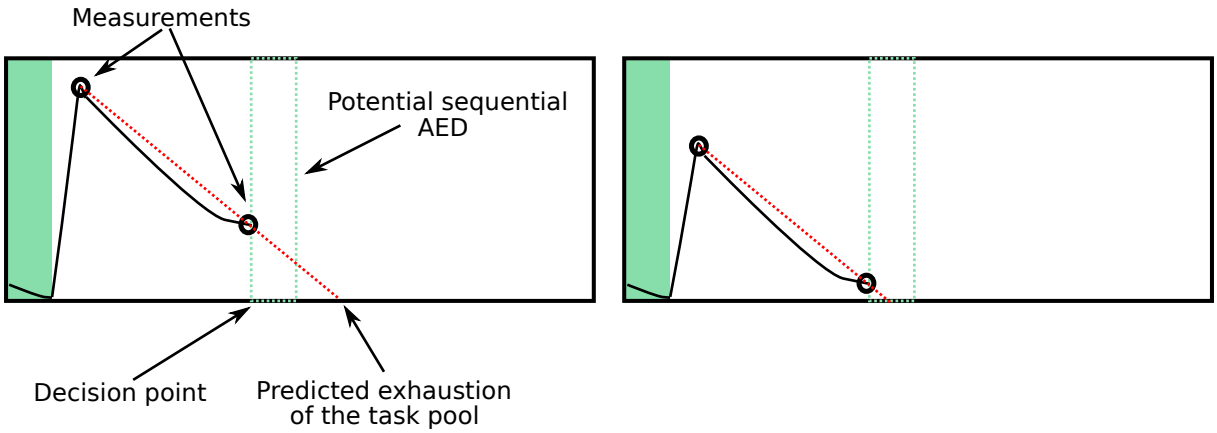


Figure 19: An illustration of how the performance models can be used to make adaptive task insertion decisions. The vertical axis shows the total number of inserted task and the horizontal axis shows the elapsed real time. An already executed sequential AED is illustrated with green shading. The red dashes show the predicted slope of the graph. The left illustration models a situation where the sequential AED is predicted to finish before the task pool has been exhausted. The right illustration models a situation where the sequential AED is predicted to finish after the task pool has been exhausted.

Our self-adaptive mechanism makes good use of the observation that the number of tasks in the task pool as a function of time can be well approximated by a linear model. Due to this we only need two adequately spaced samples to accurately estimate the parameters of the linear model. The idea is illustrated in Figure 19. We start by recording the number of tasks (and the time) immediately after inserting all the bulge-chasing tasks. We then again record the number of tasks immediately before starting an AED. We draw a line through these two points and extrapolate to find the point in time, t_{dep}, when the task pool is expected to be depleted. Let t_{aed} denote the point in time when the next AED is estimated to complete if it is executed sequentially. We use StarPU's performance modeling capability to estimate the execution time of a sequential AED based on a performance model that StarPU calibrates at run-time. The model for the AED tasks is of the form an^b, where a and b are parameters estimated by StarPU and n is the size of the AED window. The self-adaptive mechanism dynamically decides to run the next AED in parallel if $t_{\text{aed}} < t_{\text{dep}}$. Otherwise it will run sequentially.

This adaptive approach is indented to be used in situations where it is unclear whether parallel AED is beneficial. In its current form, the implementation will perform a sequential AED if the performance model is not calibrate well enough for a given AED size. The acquired data point is then used to calibrate the model. If the AED window size is 1000, then parallel AED is likely to be faster but the first few sequential AEDs are not going to matter that much in the long run since the performance model will gradually calibrate itself. However, a few sequential AEDs of the size 5000 are virtually guaranteed to destroy the performance. Furthermore, if the AED window is very small, then the StarPU overhead is going to dominate the execution time on any machine. Thus, the self-adaptive mechanism is only activated when the AED window size w is within a cer-

tain interval $w_{\text{lo}} \leq w \leq w_{\text{up}}$ defined by a *lower parallel AED threshold* w_{lo} and an *upper parallel AED threshold* w_{up}. AEDs with a window size smaller than w_{lo} are processed sequentially. Those whose window size is larger than w_{up} are processed in parallel. Note that since AED is recursive (by invoking the QR algorithm with AED), the self-adaptive mechanism can be used inside a recursive call even for large problems.

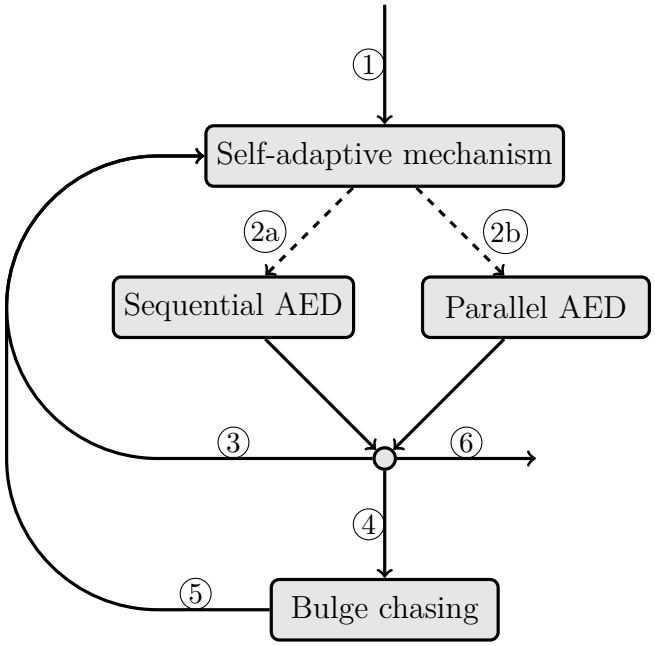


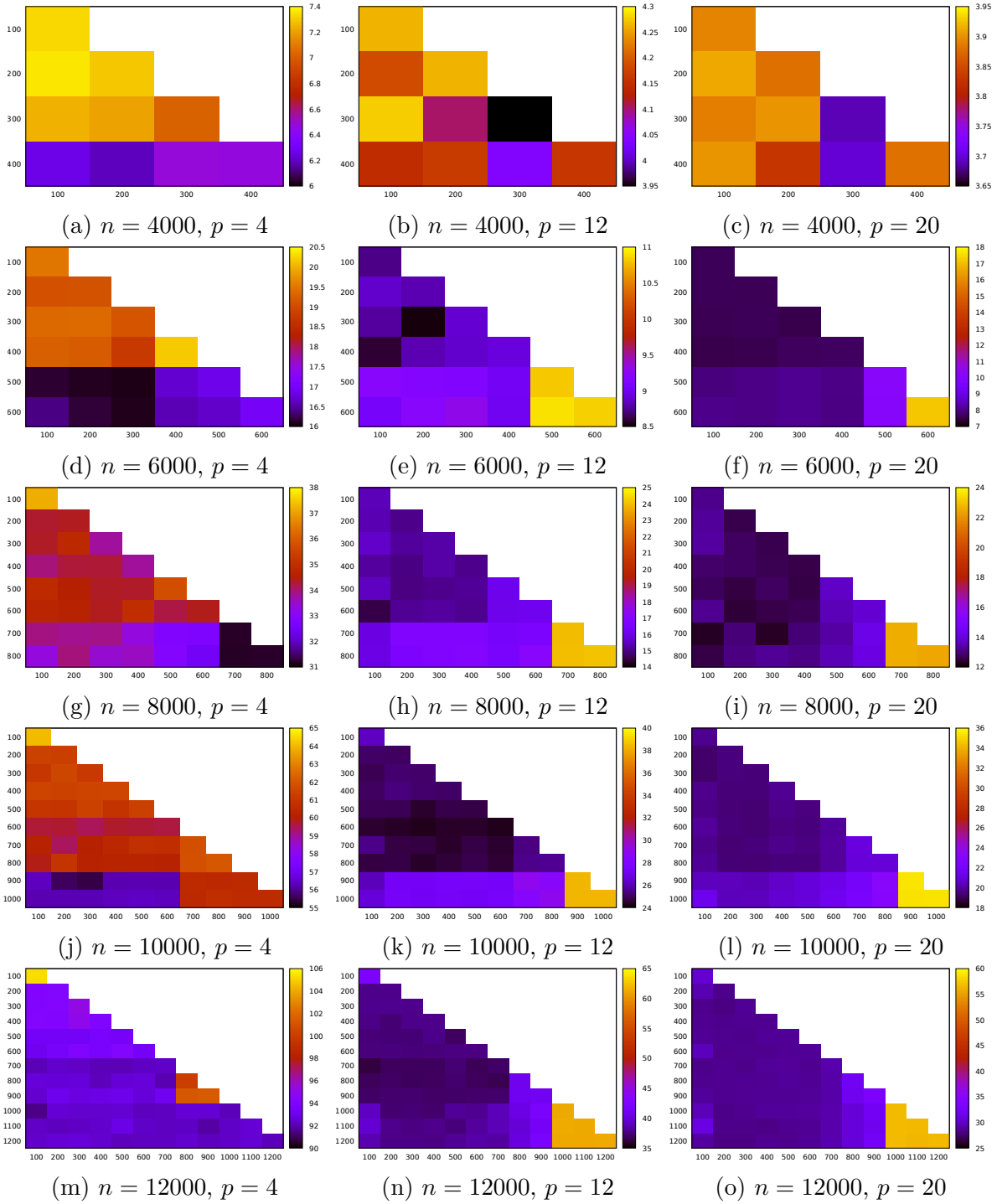
Figure 20: Simplified control-flow graph for the QR algorithm with a self-adaptive mechanism which decides when to use sequential or parallel AED. 1) The algorithm starts with AED. 2) A decision is made to use either sequential or parallel AED. 3) AED can be followed by AED. 4) AED can generate shifts to a subsequent bulge chasing. 5) Bulge chasing is followed by AED. 6) The algorithm terminates after AED deflates all remaining eigenvalues.

Figure 20 shows a simplified control-flow graph for the QR algorithm with AED and our self-adaptive mechanism. The self-adaptive mechanism chooses between sequential (2a) and parallel (2b) AED. Regardless of the choice, AED is followed by another AED, by bulge chasing, or by termination.

4.3 Experiments

We evaluated the self-adaptive mechanism through experiments whose results are summarized in Figure 21. Each of the 15 heat maps shows the runtime of the QR algorithm as a function of the lower/upper parallel AED thresholds w_{lo} and w_{up}. Recall that the self-adaptive mechanism is applied only for AEDs with a window size between these thresholds. The diagonals correspond to fixed thresholds with the self-adaptive mechanism turned off (see also Figure 17).

We draw two conclusions from this data. Firstly, if we use a fixed threshold (which corresponds to the diagonals in the heat maps), then the optimal value of the threshold depends on both the problem size and the core count. Secondly, optimal or close to optimal runtimes are obtained using the self-adaptive mechanism with a lower threshold of $w_{\text{lo}} = 300$ and an upper threshold which is effective "infinite". It should be noted that



(a) $n = 4000$, $p = 4$ (b) $n = 4000$, $p = 12$ (c) $n = 4000$, $p = 20$

(d) $n = 6000$, $p = 4$ (e) $n = 6000$, $p = 12$ (f) $n = 6000$, $p = 20$

(g) $n = 8000$, $p = 4$ (h) $n = 8000$, $p = 12$ (i) $n = 8000$, $p = 20$

(j) $n = 10000$, $p = 4$ (k) $n = 10000$, $p = 12$ (l) $n = 10000$, $p = 20$

(m) $n = 12000$, $p = 4$ (n) $n = 12000$, $p = 12$ (o) $n = 12000$, $p = 20$

Figure 21: The runtime of the QR algorithm as a function of the lower (horizontal axes) and upper (vertical axes) parallel AED thresholds. The color indicates the runtime in seconds. (Notice that the scales differ between panels.)

these experiments were performed under conditions where the performance models were given best possible change to calibrate themselves properly and the predictions could thus be expected to be quite accurate. If the performance models are not as well calibrated, then unsuccessful predictions are more likely to happen and this could be very costly with larger AED windows. In addition, the sequential implementation does not perform well with larger AED windows due to cache reuse related reasons. Thus, the upper threshold should still be imposed in practise.

5 Conclusion

We explored a variety of mechanisms that make parallel numerical linear algebra routines *self-adaptive* in the sense that they dynamically optimize their own performance during one run and/or from one run to the next. The advantages from a user's perspective are clear: less need to manually configure the software and less need to allocate computational resources to just-in-case offline tuning.

We presented three case studies from various components of our non-symmetric eigenvalue problem solvers. In Case I we used a rule of thumb together with a general observation about the shape of the performance profile of `GEMM` to derive a self-adaptive mechanism. Experimental results showed that this mechanism was capable of locating, within a fraction of a single run, a range of decent values for an important algorithmic block size parameter.

In Case II we constructed a self-adaptive mechanism which addresses a delicate trade-off between computational efficiency and idling overhead by crudely modeling two important components of the parallel cost. Even though the resulting model does not accurately estimate the true cost, it is nevertheless good enough to accurately and robustly locate a range of tile sizes with decent overall performance. In this case, the adaptation was from one run to the next. Experimental results showed that good tile sizes could be found after just a handful of runs.

In Case III we added self-adaptivity by making a previously static decision dynamic. A certain task was at risk of becoming a sequential bottleneck unless it was parallelized. We modeled the rate at which tasks were executed and estimated the execution time of an upcoming task to dynamically decide when to parallelize a critical task. Experimental results show that the self-adaptive mechanism in many cases yields optimal or close to optimal runtimes, thus removing the need for choosing a single threshold. In addition, even in the worst cases the adaptive approach does not ruin the performance.

At first glance, it may appear as if we have merely replaced the problem of setting one set of parameters with the problem of setting a different set of parameters. But this ignores a subtle but very important difference in the character of these parameters. The original parameters determine the performance which we are trying to optimize. The new parameters, however, influence some aspects of the adaptive mechanism. For example, how fast it adapts. Concrete values must be chosen for the new parameters, but there is no need to *optimize* them. Therein lies the primary distinction. Regardless of how the new parameters are set, we will end up with an adaptive scheme. We could have eliminated the new parameters altogether by simply picking some concrete values, but we chose instead to make the design choices explicit.

In conclusion, there appears to be many diverse and effective low-cost mechanisms for adding self-adaptivity to parallel numerical linear algebra routines. The final performance may not be as good as what can be achieved through more rigorous approaches such as

offline tuning. But considering the robustness, ease-of-use, and negligible run-time overhead that appear to be within reach with these types of approaches, getting only decent instead of near-optimal performance might be a fair price to pay. Despite the potential of self-adaptive mechanisms demonstrated in this report, it is hard to find literature on the subject.

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] K. Braman, R. Byers, and R. Mathias. The Multishift QR Algorithm. Part I: Maintaining Well-Focused Shifts and Level 3 Performance. *SIAM J. Matrix Anal. Appl.*, 23(4):929–947, 2002.

[2] K. Braman, R. Byers, and R. Mathias. The Multishift QR Algorithm. Part II: Aggressive Early Deflation. *SIAM J. Matrix Anal. Appl.*, 23(4):948–973, 2002.

[3] J. Dongarra, D. Sorensen, and S. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215–227, 1989.

[4] G. Quintana-Ortí and R. van de Geijn. Improving the performance of reduction to hessenberg form. *ACM Transactions on Mathematical Software*, 32(2):180–194, 2006.

[5] A. Schwarz and L. Karlsson. Scalable Eigenvector Computation for the Non-Symmetric Eigenvalue Problem. Submitted (under review), 2019.

[6] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal,and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 36(12):645–654, 2010.

[7] R. C. Whaley. Empirically tuning LAPACK's blocking factor for increased performance. In *2008 International Multiconference on Computer Science and Information Technology*, pages 303–310. IEEE, 2008.