An Auto-Tuning Framework for a NUMA-Aware Hessenberg Reduction Algorithm

Mahmoud Eljammaly
Umeå University
Department of Computing Science
SE-901 87 Umeå, Sweden
mjammaly@cs.umu.se

Lars Karlsson
Umeå University
Department of Computing Science
SE-901 87 Umeå, Sweden
larsk@cs.umu.se

Bo Kågström
Umeå University
Department of Computing Science
and HPC2N
SE-901 87 Umeå, Sweden
bokg@cs.umu.se

ABSTRACT

The performance of a recently developed Hessenberg reduction algorithm greatly depends on the values chosen for its tunable parameters. The problem is hard to solve effectively with generic methods and tools. We describe a modular auto-tuning framework in which the underlying optimization algorithm is easy to substitute. The framework exposes sub-problems of standard auto-tuning type for which existing generic methods can be reused. The outputs of concurrently executing sub-tuners are assembled by the framework into a solution to the original problem. This paper presents work-in-progress.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; • **Software and its engineering** → *Software performance*; • **General and reference** → Performance;

KEYWORDS

Auto-tuning, Tuning framework, Binning, Search space decomposition, Multistage search, Hessenberg reduction, NUMA-aware

ACM Reference Format:
Mahmoud Eljammaly, Lars Karlsson, and Bo Kågström. 2018. An Auto-Tuning Framework for a NUMA-Aware Hessenberg Reduction Algorithm. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186304

1 INTRODUCTION

The motivation behind this work starts from the distributed parallel multishift QR algorithm [9], which is the key step in solving large dense unsymmetric eigenvalue problems. On the critical path of the distributed QR algorithm lies a costly process known as Aggressive Early Deflation (AED) [2, 3]. AED is composed of three major parts: Schur decomposition, eigenvalue reordering, and Hessenberg reduction. The AED process is currently a bottleneck in the distributed QR algorithm and we aim to accelerate it in the hopes of improving the performance and scalability of the QR algorithm. We recently developed a new NUMA-aware Hessenberg reduction algorithm [7] based on the Parallel Cache Assignment (PCA) technique [4, 5, 10]. The performance of the new algorithm depends greatly on the values chosen for its tunable parameters. Auto-tuning is required due to both the large number of parameters and the interactions between different parameters.

In this paper, we propose a modular auto-tuning framework that helps with the tuning process. In particular, the framework tries to search the huge search space efficiently by partitioning the parameters into subsets that are tuned independently, grouping similar sub-problems into the same bin and tune them as one, and searching in multiple stages (first coarsely and then finely). The framework by itself is not a complete solution. At the heart of the framework is a generic module for optimizing a sub-problem of standard type. The framework provides a clean interface to generic optimization methods and extends them into an auto-tuner for the complex and non-standard original problem. In addition, this makes it easy to experiment with different search algorithms.

The framework works as pre- and post-processing layers around the NUMA-aware algorithm. The interactions between the framework and the algorithm are as follows. The user provides to the framework an input matrix $A \in \mathbb{R}^{n \times n}$ and the number, p, of available cores. Based on n and p, the framework chooses specific values for all the algorithmic parameters of the Hessenberg algorithm. The framework then executes the algorithm on A with the specified parameters. The output matrices H and Q are returned to the user. Simultaneously, the Hessenberg algorithm feeds back internal time measurements to the framework for use in the tuning process.

2 NUMA-AWARE HESSENBERG REDUCTION

Hessenberg reduction is an orthogonal similarity transformation that maps a matrix $A \in \mathbb{R}^{n \times n}$ to an upper Hessenberg matrix $H = Q^T AQ$. The state-of-the-art algorithm [12] performs the reduction in a blocked manner. The matrix is reduced one block of columns (a *panel*) at a time from left to right. Each panel is reduced column-by-column using Householder reflectors. The reflectors are also applied to the rest of the matrix to update it. Most of the work associated with the updates are delayed. One iteration consists of two phases: a *reduction phase*, in which a panel is reduced, and an *update phase*, in which the delayed updates are fully applied, see [12] for details. Figure 1 shows the shapes of A and other matrices used in the reduction after the first k columns of A have been reduced. Here b refers to the width of the next panel.

The operations in the reduction phase are mainly matrix–vector operations, which makes the whole phase memory-bound. The most expensive operation is a large matrix–vector multiplication involving $A_{2,2:3}$ during the computation of Y_2. To perform this multiplication efficiently, our NUMA-aware algorithm [7] uses the PCA technique [4, 5, 10].

The NUMA-aware algorithm provides two parallelization strategies for the reduction phase. In the *partial parallelization strategy*, multi-threading is used only for the most expensive multiplication while in the *full parallelization strategy* multi-threading is used for most of the operations.

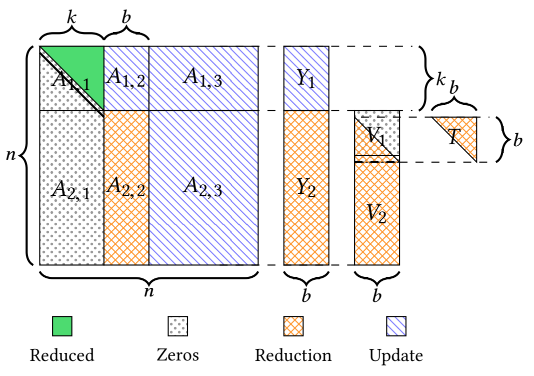

Figure 1: Partitioning of matrix A after reducing the first k columns, and Y, V and T will be used to reduce the panel A_{22}.

2.1 Algorithmic Parameters

There are four families of tunable parameters in the NUMA-aware algorithm (see Table 1). There is one instance of each parameter *per iteration* of the algorithm, which means that there are $4N$ parameters to tune if there are N iterations. A complicating factor is that N in turn depends on the values chosen for the panel width parameters (b). Since our particular context (as a part of AED) implies that n might be relatively small (hundreds to thousands) compared to p (available cores), it may turn out to be sub-optimal to use all available cores, especially towards the end of the computation. The parameters t_r and t_u therefore specify the number of threads/cores ($\leq p$) to use in the reduction and update phases, respectively.

Table 1: The four families of algorithmic parameters.

Parameter name	*Type*	*Domain*	*Phases*
Panel width (b)	Integer	$\{1, \ldots, n-k\}$	Both
Strategy (s)	Category	{Full, Partial}	Reduction
No. of threads (t_r)	Integer	$\{1, \ldots, p\}$	Reduction
No. of threads (t_u)	Integer	$\{1, \ldots, p\}$	Update

3 TECHNIQUES

At the heart of the framework is a *search module* (see Section 4.1 ahead), which abstracts any standard auto-tuning method behind a generic interface. The main aim of the framework is to extend the limited capability of the tuning algorithm within the search module into a complete auto-tuner for the NUMA-aware algorithm. The framework achieves this by employing three specific techniques described in this section.

3.1 Sub-Problem Decomposition

The algorithm consists of an outer loop with non-overlapping iterations, so it is reasonable to assume that parameters from different iterations are uncoupled. However, the four parameters within an iteration do strongly interact and must be tuned together. This leads to the thought of decomposing the problem of tuning all $4N$ parameters at once into tuning N independent sets of 4 parameters. Yet, since the number of iterations, N, depends on one of the parameter families (the panel width) this idea cannot be directly applied.

By analyzing the Hessenberg reduction algorithm and Figure 1 it becomes clear that the shape of A at the start of an iteration depends only on n and k. We associate a sub-problem with each pair (n, k). The sub-problem for (n, k) is defined as finding optimal parameter settings for the upcoming iteration. The objective function (for the sub-problem) is to maximize the performance

$$P = \frac{F_r + F_u}{T_r + T_u},$$

where F_r and F_u are the flop counts for the reduction and update phases, respectively, and T_r and T_u are the wall clock times. Since the total flop count for reducing a matrix is fixed, improving the performance of the sub-problems will also speed-up the overall problem, but not necessarily optimize its performance.

We collect the values of the parameters for one sub-problem into a 4-tuple referred to as a *parameter tuple*. We arrange all the N parameter tuples as columns (from left to right) in a *parameter table*. The objective for the auto-tuner represented by the framework is to find a parameter table that minimizes the total execution time.

3.1.1 Concurrent Solution of Several Sub-Problems. The size of a parameter table depends on the number of iterations, which in turn depends on the chosen panel widths. For an input matrix of fixed size n, there are $n-2$ possible sub-problems (n, k) for $k = 0, 1, \ldots, n-3$. Any particular parameter table therefore consists of parameter tuples extracted from some subset of the sub-problems.

One execution of the Hessenberg algorithm uses N parameter tuples provided by the framework and in turn feeds back measurements used by the framework to make progress on N sub-problems. In other words, the framework concurrently solves several sub-problems. But note, however, that exactly *which subset* of the $n-2$ sub-problems are relevant for a given execution depends on the chosen panel widths. See Figure 2 for an illustration of the relationships between sub-problems, parameter tuples, and parameter tables. The framework logically keeps track of $n-2$ partially solved sub-problems and after each particular execution of the Hessenberg algorithm is able to make progress on some subset of them.

3.2 Binning Similar Sub-Problems

Two distinct sub-problems (n, k) and (n', k') are similar if $n \approx n'$ and $k \approx k'$ simply because the shapes of all operands are similar. What this means is that we could (with some loss of accuracy) treat

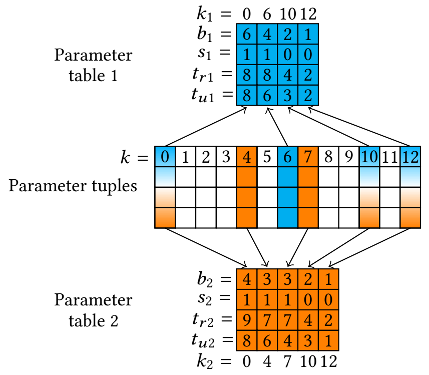

Figure 2: Two examples of parameter tables for $n = 15$.

the two as one single sub-problem. This has several benefits. First, it reduces the total number of sub-problems that need to be solved. Second, it allows the effort invested into making progress on one sub-problem to benefit also other (similar) sub-problems.

Specifically, we group adjacent sub-problems into bins and tune each bin as if it represents a single sub-problem. The bins are rectangular of size $\Delta_n \times \Delta_k$ as illustrated by the example in Figure 3 for $\Delta_n = 2$ and $\Delta_k = 3$. In particular, the sub-problems $(10, 4)$ and $(9, 6)$ belong to the same bin $(4, 2)$.

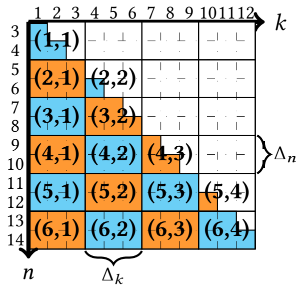

Figure 3: Binning of (12×12) space using bins of size (2×3).

3.3 Searching in Multiple Stages

Parameter tuples that yield good performance have a strong tendency (in this application) to cluster in one region of the search space. By performing the search in multiple stages, we can (potentially) more rapidly localize the search to this promising region. The idea is to start with a sparse but well distributed subset of the search space in the first stage. Once (near-)convergence is reached, the search space is made denser and also restricted to a region around the converged point in subsequent stages.

For example, consider the two-stage search in Figure 4 which involves only t_r and t_u for simplicity. The goal is to optimize within the domain $\{1, \ldots, 10\}$. In the first stage, we choose the sparse but well distributed sub-domain $\{1, 4, 7, 10\}$. Suppose the search in the first stage converges to the point $(t_r, t_u) = (4, 7)$. Then we include more points and restrict the search in the second stage to the sub-domain $\{2, 3, 4, 5, 6\}$ for t_r and $\{5, 6, 7, 8, 9\}$ for t_u.

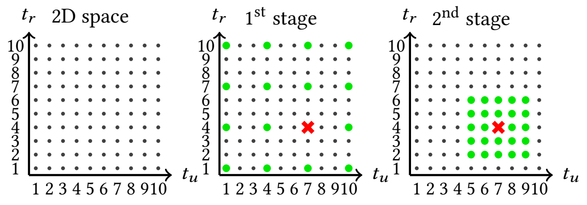

Figure 4: Two-stage search for 2D parameter space.

4 THE FRAMEWORK'S ARCHITECTURE

The framework consists of three modules described in this section.

4.1 The Search Module

The purpose of the Search Module is to encapsulate some standard auto-tuning method behind an abstract interface. The framework does not provide any implementation of this module by itself.

The Search Module has two primary functions: choose a parameter tuple for a given sub-problem and advance the search for a given sub-problem by one step in response to feedback. The module implementation itself is supposed to be state-less. A search state is instead encapsulated by the implementation into an opaque object[1] that is externally managed by the framework (see Sections 4.2 and 4.3 ahead). Since the specifics of what constitutes a "search state" depends entirely on the implementation of the Search Module, the framework views these objects as binary blobs[2].

The Search Module exposes the following interface:

- Create–State: Creates a new state.
- Select–Parameters: Chooses a parameter tuple.
- Receive–Feedback: Feeds back time measurements.
- Update–State: Performs a search step using feedback.
- Check–Convergence: Check if the search has converged.

4.2 The Management Module

The Management Module provides the glue that binds all the other modules together with the user input/output and the Hessenberg algorithm. The core functions of the module are as follows:

Construct parameter table. Starting from $k = 0$ and repeatedly calling the Select–Parameters function of the Search Module, a complete parameter table can be constructed. The search state to use is either fetched from the Database Module or initialized using Create–State. Binning is applied before looking up a search state. The process of constructing a parameter table also implicitly selects the subset of *active sub-problems*, i.e., sub-problems used in the next execution. So *before* calling Select–Parameters, the function Update–State is called on to make one step in the optimization algorithm. Furthermore, if the Check–Convergence function signals convergence, then the state is re-initialized with the search space used in the next stage of the multi-stage search.

Run the Hessenberg algorithm. The parameter table is passed alongside the other inputs to the Hessenberg algorithm. The computed matrices are output to the user.

[1]An object whose content and structure are not concretely known.
[2]Collection of data stored in binary as a single entry.

Feed back measurements. The internal time measurements from each iteration are fed back to the active sub-problem search states using the Receive–Feedback function.

4.3 The Database Module

The Database Module stores binary blobs representing the search states. The search states are indexed by bin coordinates.

5 EXPERIMENTAL RESULTS

The framework is dependent on the implementation of the Search Module. In order to test the framework we implemented the Search Module using the *Nelder-Mead* algorithm [11], with $\alpha = 1$, $\beta = \frac{1}{2}$, $\gamma = 2$. This is neither the best nor the worst choice of algorithm. Ultimately the choice is not so important since the aim of this section is to show that the framework is able to make gradual improvements of the overall performance even though the actual optimization is only performed on small sub-problems. What the most effective implementation of the Search Module looks like is an open problem and something we do not contemplate in this paper.

We used bins of size 10×10 and multi-stage search spaces as defined by Table 2, where b', t'_r, t'_u refer to the best values found in the first stage. The vertices are mapped to the nearest integer point. Figure 5 shows the execution times (dots) of 500 executions (no repetition) for a matrix of order $n = 1000$. The curve shows a moving average of 50 consecutive measurements. The results indicate that in general the performance is improving over time. We hope by using the best choice of algorithm to reach results comparable to [7]. The experiments were performed on one node of Abisko at HPC2N.

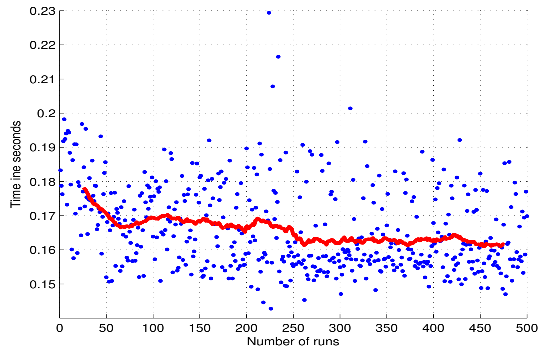

Figure 5: Execution time of 500 runs of the new algorithm for $n = 1000$ using the framework. The red curve represents the moving average for a window of size 50.

Table 2: The search spaces used in multi-stage search.

Parameter symbol	*1st stage domain*	*2nd stage domain*
b	$10 : 10 : 100$	$b' - 9 : b' + 10$
s	{Full, Partial}	{Full, Partial}
t_r	$6 : 6 : 48$	$t'_r - 5 : t'_r + 6$
t_u	$6 : 6 : 48$	$t'_u - 5 : t'_u + 6$

6 SUMMARY

In this paper we propose a modular auto-tuning framework for a recently developed Hessenberg reduction algorithm. A brief description of the new algorithm and its parameters are presented. The algorithm's parameters interact with each other and span a huge search space which makes using generic tuning methods and tools like [1, 6, 8] not directly applicable.

The framework applies search space decomposition, binning, and multi-stage search to accelerate the search. It defines an abstract module with clear interface which can encapsulate any standard optimization methods or generic tuning tools. including [1, 6, 8]. This allows the experimentation with different tuning algorithms.

For testing the framework's ability to improve the overall performance of the new Hessenberg reduction algorithm, we used the Nelder-Mead algorithm in the search module. The results show that the performance is gradually improving over time.

Future work includes experimenting with both generic and specialized tuning algorithms in the search module and apply the idea underlying the framework to other similar algorithms.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments. We thank the High Performance Computing Center North (HPC2N) at Umeå University for providing computational resources and valuable support during test and performance runs. Financial support has been received from the European Unions Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633, and by eSSENCE, a strategic collaborative e-Science programme funded by the Swedish Government via VR.

REFERENCES

[1] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*. Edmonton, Canada.
[2] K. Braman, R. Byers, and R. Mathias. 2002. The Multishift QR Algorithm. Part I: Maintaining Well-Focused Shifts and Level 3 Performance. *SIMAX* 23, 4 (2002), 929–947.
[3] K. Braman, R. Byers, and R. Mathias. 2002. The Multishift QR Algorithm. Part II: Aggressive Early Deflation. *SIMAX* 23, 4 (2002), 948–973.
[4] A. Castaldo and R. Whaley. 2011. Achieving Scalable Parallelization for the Hessenberg Factorization. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on.* IEEE, 65–73.
[5] A. Castaldo, R. Whaley, and S. Samuel. 2013. Scaling LAPACK Panel Operations Using Parallel Cache Assignment. *ACM Trans. Math. Software* 39, 4 (2013).
[6] C. Ţăpuş, I. Chung, and J. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02).* IEEE Computer Society Press, 1–11.
[7] M. Eljammaly, L. Karlsson, and B. Kågström. December, 2016. *Evaluation of the Tunability of a New NUMA-Aware Hessenberg Reduction Algorithm. NLAFET Working Note 8.* Also as Report UMINF 16.22, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.
[8] M. Gerndt, S. Benkner, E. César, C. Navarrete, E. Bajrovic, J. Dokulil, C. Guillén, R. Mijakovic, and A. Sikora. 2017. A multi-aspect online tuning framework for HPC applications. *Software Quality Journal* (16 May 2017).
[9] R. Granat, B. Kågström, D. Kressner, and M. Shao. 2015. Algorithm 953: Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation. *ACM Trans. Math. Softw.* 41, 4, Article 29 (Oct. 2015), 29:1–29:23 pages.
[10] M. Hasan and R. Whaley. 2014. Effectively Exploiting Parallel Scale for all Problem Sizes in LU Factorization. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International.* IEEE, 1039–1048.
[11] J. A. Nelder and R. Mead. 1965. A simplex method for function minimization. *The computer journal* 7, 4 (1965), 308–313.
[12] G. Quintana-Ortí and R. van de Geijn. 2006. Improving the performance of reduction to Hessenberg form. *ACM Trans. Math. Software* 32, 2 (2006), 180–194.