On the Tunability of a New Hessenberg Reduction Algorithm Using Parallel Cache Assignment

Mahmoud Eljammaly(✉), Lars Karlsson, and Bo Kågström

Umeå University, 901 87 Umeå, Sweden
{mjammaly,larsk,bokg}@cs.umu.se

Abstract. The reduction of a general dense square matrix to Hessenberg form is a well known first step in many standard eigenvalue solvers. Although parallel algorithms exist, the Hessenberg reduction is one of the bottlenecks in AED, a main part in state-of-the-art software for the distributed multishift QR algorithm. We propose a new NUMA-aware algorithm that fits the context of the QR algorithm and evaluate the sensitivity of its algorithmic parameters. The proposed algorithm is faster than LAPACK for all problem sizes and faster than ScaLAPACK for the relatively small problem sizes typical for AED.

Keywords: Hessenberg reduction · Parallel cache assignment
NUMA-aware algorithm · Shared-memory · Tunable parameters
Off-line tuning

1 Introduction

This work is motivated by a bottleneck in the distributed parallel multi-shift QR algorithm for large-scale dense matrix eigenvalue problems [7]. On the critical path of the QR algorithm lies an expensive procedure called *Aggressive Early Deflation* (AED) [1,2]. The purpose of AED is to detect and deflate converged eigenvalues and to generate shifts for subsequent QR iterations. There are three main steps in AED: Schur decomposition, eigenvalue reordering, and Hessenberg reduction. This work focuses on the last step while future work will investigate the first two steps.

In the context of AED, Hessenberg reduction is applied to relatively small problems (matrices of order hundreds to thousands) and, since AED appears on the critical path of the QR algorithm, there are relatively many cores available for its execution. The distributed QR algorithm presented in [7] computes the AED using a subset of the processors. We propose to select one shared-memory node and use a shared-memory programming model (OpenMP) for the AED. The aim is to develop a new parallel Hessenberg reduction algorithm which outperforms the state-of-the-art algorithm for small problems by using fine-grained parallelization and tunable algorithmic parameters to make it more efficient and flexible. Tuning the algorithmic parameters of the new algorithm is not one of the main concerns in this paper. Rather, this work focuses on the tunability potential of the algorithmic parameters.

A shared-memory node within a distributed system commonly has a *Non-Uniform Memory Access* (NUMA) architecture. Since Hessenberg reduction is a memory-bound problem where matrix–vector multiplications typically account for most of the execution time, high performance is obtained when the cost of memory accesses is minimized. Therefore, our algorithm employs the *Parallel Cache Assignment* (PCA) technique proposed by Castaldo and Whaley [4,5,8]. This technique leads to two benefits. First, the algorithm becomes NUMA-aware. Second, the algorithm uses the aggregate cache capacity more effectively.

The rest of the paper is organized as follows. Section 2 reviews a blocked Hessenberg reduction algorithm and the PCA technique. Section 3 describes how we applied the PCA technique to the blocked algorithm. Section 4 evaluates the impact of tuning each parameter. Section 5 shows the new algorithm's performance after tuning and compares it with state-of-the-art implementations. Section 6 concludes and highlights future work.

2 Background

2.1 Blocked Hessenberg Reduction

In this section we review the basics of the state-of-the-art algorithm in [11] on which our algorithm is based. Hessenberg reduction transforms a given square matrix $A \in \mathbb{R}^{n\times n}$ to an upper Hessenberg matrix $H = Q^T AQ$, where Q is an orthogonal matrix. A series of Householder reflections applied to both sides of A are used to zero out—*reduce*—the columns one by one from left to right.

The algorithm revolves around block iterations, each of which reduces a block of adjacent columns called a *panel*. After reducing the first $k - 1$ columns, the matrix A is partitioned as in Fig. 1, where b is the *panel width*.

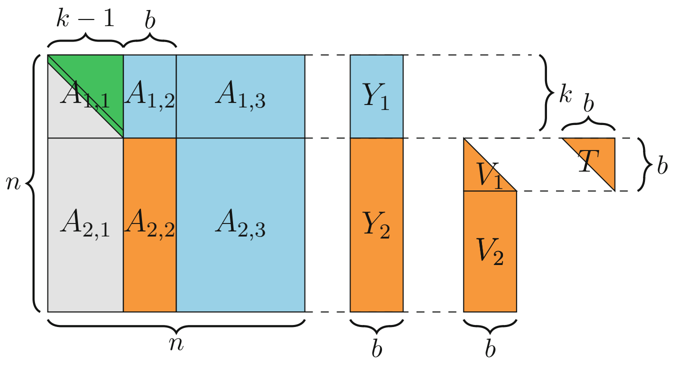

Fig. 1. Partitioning of A after reducing the first $k-1$ columns, and Y, V and T to be used for reducing $A_{2,2}$.

The panel $A_{2,2}$ (starting at the sub-diagonal) is reduced to upper triangular form by constructing and applying a transformation of the form

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

where $I - VTV^T$ is a compact WY representation [12] of the b Householder reflections that reduced the panel. In practice, the algorithm incrementally builds an intermediate matrix $Y = AVT$ to eliminate redundant computations in the updates from the right. The matrix Y is partitioned as in Fig. 1. Each block iteration consists of two phases. In the first phase, the panel $A_{2,2}$ is reduced and fully updated. This gives rise to a set of b Householder reflections, which are accumulated into a compact WY representation $I - VTV^T$. The first phase also incrementally computes $Y_2 \leftarrow A_{2,2:3}VT$. In the second phase, $Y_1 \leftarrow A_{1,2:3}VT$ is computed, and blocks $A_{1,2}$, $A_{1,3}$, and $A_{2,3}$ are updated according to

$$A \leftarrow (I - VTV^T)^T (A - YV^T), \tag{1}$$

where the dimensions of A, V, T and Y are derived from Fig. 1 according to which block is to be updated.

Other Variants of Hessenberg Reduction. A multi-stage Hessenberg reduction algorithm exists [9]. In this variant, some of the matrix-vector operations are substituted by matrix-matrix operations for the cost of performing more compute-bound computations overall. Applying PCA to this variant will be much less efficient since PCA is useful when we have repetitive memory-bound computations, as explained in Sect. 2.2.

2.2 PCA: Parallel Cache Assignment

Multicore shared-memory systems have parallel cache hierarchies with sibling caches on one or more levels. In such systems, the aggregate cache capacity might be able to persistently store the whole working set. To exploit this phenomenon, Castaldo and Whaley proposed the PCA technique and applied it to the panel factorizations of one-sided factorizations [5] as well as to the unblocked Hessenberg reduction algorithm [4]. They argued that PCA is able to turn memory-bound computations of small problems into cache-bound (or even compute-bound) computations by utilizing the parallel caches to transform the vast majority of memory accesses into local cache hits.

The main idea of PCA is to consider sibling caches as local memories in a distributed memory system and to assign to each core a subset of the data. Work is then assigned using the owner-computes rule. In addition, one may explicitly copy the data assigned to a specific core into a local memory to that core.

A pivotal aspect to benefit from using PCA is having a repeated memory-bound computation for the same memory region. Applying PCA allows fetching a large block of data from the main memory into several caches and use it repeatedly while still in the cache, which eliminates the slowdown penalty presented by repeatedly using the memory buses.

3 Hessenberg Reduction Using PCA

The proposed algorithm (Algorithm 1) is a parallel variant of [11] using PCA and aimed at small matrices. The algorithm consists of two nested loops. The inner loop, lines 7–24, implements the first phase while the remainder of the outer loop, lines 25–30, implements the second phase. In the following, we briefly describe the parallelization of each phase. For more details see the technical report [6].

3.1 Parallelization of the First Phase

The first phase is memory-bound due to the large matrix–vector multiplications on lines 17–18. The objective is to apply PCA to optimize the memory accesses. We partition A, V, and Y as illustrated in Fig. 1. This phase consists of four main steps for each column $\mathbf{a} = A_{2,2}(:, j)$ of the panel: update $\mathbf{a}$ from the right (lines 9–10), update $\mathbf{a}$ from the left (lines 11–15), reduce $\mathbf{a}$ (line 16), augment Y and T (lines 17–24). Two parallelization strategies are considered for this phase. In the *full strategy*, all multiplications except triangular matrix–vector are parallelized. In the *partial strategy*, only the most expensive computational step, lines 17–18, is parallelized. The full strategy exposes more parallelism at the cost of more overhead which makes it suitable only for sufficiently large problems.

To apply PCA, before each first phase the data are assigned to threads where each thread mainly works on data it owns. The matrix–vector multiplications in this phase involve mostly tall–and–skinny or short–and–fat matrices. For efficient parallelization in the full strategy, the matrices are partitioned along their longest dimension into p_1 parts assigned to p_1 threads. To parallelize the costly step in lines 17–18, $A_{2,2:3}$ is first partitioned into p_1 block rows then each thread *explicitly copies* its assigned block into local memory, (line 6). Having the assigned data from this block in a buffer local to the thread will reduce the amount of remote memory accesses, cache conflicts and false sharing incidents, which make the algorithm NUMA-aware. So even if the data did not fit into the cache, the algorithm will still benefit from the data locality. In general, all matrices are distributed among the threads in a round-robin fashion based on memory-pages.

3.2 Parallelization of the Second Phase

The second phase is compute-bound and mainly involves matrix–matrix multiplications. The objective is to balance the workload and avoid synchronization as much as possible. There are four main steps: updating $A_{2,3}$ from the right (lines 26–27), updating $A_{2,3}$ from the left (line 28), computing Y_1 (line 29), and updating $A_{1,2:3}$ (line 30). With conforming block partitions of the columns of $A_{2,3}$ and V_2^T, and of the block rows of $A_{1,2:3}$ and Y_1 (line 25) the computation can be performed without any synchronization.

3.3 Algorithmic Parameters

There are four primary algorithmic parameters: the panel width, the parallelization strategy, and the thread counts for both phases. The panel width b can be set

Algorithm 1. Parallel blocked Hessenberg reduction using PCA.

```
for $k \leftarrow 1 : b : n-2$ do // Outer loop over panels
    $V \leftarrow 0_{n-k\times 0}$, $T \leftarrow 0_{0\times 0}$, $Y \leftarrow 0_{n\times 0}$// Initialize intermediate matrices
    if $s = full$ then $\hat{p} \leftarrow p_1$ else $\hat{p} \leftarrow 1$// Select strategy
    Partition $A$, $V$, and $Y$ as in Fig. 1
    Partition $A_{2,2:3}$ into $p_1$ row blocks $A_{2,2:3}^{(i)}$ for $i = 1 \dots p_1$
    Thread $i$ copies $A_{2,2:3}^{(i)}$ to local memory
    // First Phase
    for $j \leftarrow 1 : \min\{b, n-k-1\}$ do
        Partition $A_{2,2}(:,j), V, V_2, \mathbf{v}_j, Y_2$ and $\mathbf{y}_j$ into $\hat{p}$ row blocks
        $A_{2,2}^{(i)}(:,j), V^{(i)}, V_2^{(i)}, \mathbf{v}_j^{(i)}, Y_2^{(i)}$ and $\mathbf{y}_j^{(i)}$ for $i = 1 \dots \hat{p}$
        // Update column $j$ of $A_{22}$ from both sides
        parfor $i \leftarrow 1 : \hat{p}$ do
            $A_{2,2}^{(i)}(:,j) \leftarrow A_{2,2}^{(i)}(:,j) - Y_2^{(i)} V_2(1,:)^T$
            $\mathbf{w}^{(i)} \leftarrow V^{i^T} A_{2,2}^{(i)}(:,j)$
        $\mathbf{w} \leftarrow \mathbf{w}^{(1)} + \cdots + \mathbf{w}^{(\hat{p})}$
        $\mathbf{w} \leftarrow T^T \mathbf{w}$
        parfor $i \leftarrow 1 : \hat{p}$ do
            $A_{2,2}^{(i)}(:,j) \leftarrow A_{2,2}^{(i)}(:,j) - V^{(i)} \mathbf{w}$
        Construct a Householder reflection $(\mathbf{v}_j, \tau_j)$ that reduces $A_{2,2}(j+1:n, j)$
        // Augment $Y$, $T$, and $V$
        parfor $i \leftarrow 1 : p_1$ do
            $\mathbf{y}^{(i)} \leftarrow A_{2,2:3}^{(i)}(:, j+1:n)\mathbf{v}_j$
        parfor $i \leftarrow 1 : \hat{p}$ do
            $\mathbf{t}^{(i)} \leftarrow V_2^{(i)^T} \mathbf{v}_j^{(i)}$
        $\mathbf{t} \leftarrow \mathbf{t}^{(1)} + \cdots + \mathbf{t}^{(\hat{p})}$
        parfor $i \leftarrow 1 : \hat{p}$ do
            $\mathbf{y}^{(i)} \leftarrow \tau \mathbf{y}^{(i)} - Y_2^{(i)} \mathbf{t}$
        $Y \leftarrow \begin{bmatrix} Y_1 & 0 \\ Y_2 & \mathbf{y} \end{bmatrix}$, $T \leftarrow \begin{bmatrix} T & -\tau_j T \mathbf{t} \\ 0 & \tau_j \end{bmatrix}$, $V \leftarrow \begin{bmatrix} V & \mathbf{v}_j \end{bmatrix}$
    // Second Phase
    Partition $A_{2,3}$ into $p_2$ column blocks $A_{2,3}^{(i)}$ and $A_{1,2:3}(:,2:n), Y_1$ and $V_2$ into
    $p_2$ row blocks $A_{1,2:3}^{(i)}(:,2:n), Y1^{(i)}$ and $V_2^{(i)}$ for $i = 1 \dots p_2$
    parfor $i \leftarrow 1 : p_2$ do
        // Update $A_{2,3}$ from the right
        $A_{2,3}^{(i)} \leftarrow A_{2,3}^{(i)} - Y_2 V_2^{(i)^T}$
        // Update $A_{2,3}$ from the left
        $A_{2,3}^{(i)} \leftarrow A_{2,3}^{(i)} - V T^T V^T A_{2,3}^{(i)}$
        // Compute the top block of $Y$
        $Y_1^{(i)} \leftarrow A_{1,2:3}^{(i)}(:,2:n) V T$
        // Update $A_{1,2:3}$ from the right
        $A_{1,2:3}^{(i)}(:,2:n) \leftarrow A_{1,2:3}^{(i)}(:,2:n) - Y_1^{(i)} V^T$
```

to any value in the range $1, \dots, n-2$. The first phase can be parallelized using either the full or the partial parallelization strategy, as described in Sect. 3.1. The strategy $s \in \{\text{full, partial}\}$ can be set independently for each iteration of the outer loop. Using all available cores can potentially hurt the performance, especially near the end where the operations are small-sized. The synchronization overhead and cache interference may outweigh the benefits of using more cores. Therefore, the number of threads to use in each phase (p_1 and p_2) are tunable parameters that can be set independently in each outer loop iteration. If the thread count is less than the number of available cores, then threads are assigned to as few NUMA domains as possible to maximize memory throughput.

4 Evaluation of the Tuning Potential

This section evaluates the tuning potential of each algorithmic parameter while keeping all the others at their default setting.

The experiments were performed on the Abisko system at HPC2N, Umeå University. During the experiments, no other jobs were running on the same node. One node consists of four AMD Opteron 6238 processors each containing two chips with six cores each for a total of 48 cores. Each chip has its own memory controller, which means that the node has eight NUMA domains. The PathScale (5.0.0) compiler is used together with the following libraries: OpenMPI (1.8.1), OpenBLAS (0.2.13), LAPACK (3.5.0), and ScaLAPACK (2.0.2). The default parameter values in Table 1 were used in the experiments unless otherwise stated. All reported data points is the median of 100 trials, unless otherwise stated.

Tuning Potential for the Panel Width. The panel width plays a key role in shaping the performance since it determines the distribution of work. To find how b depends on the problem size we used $n \in \{500, 1000, \dots, 4000\}$. Figure 2 shows the execution time of the new algorithm for different problem sizes and panel widths. The stars correspond to the best b found for each problem size. The algorithm execution time is sensitive to the choice of b which means b need tuning.

Tuning Potential for the Parallelization Strategy. The *partial strategy* is expected to be faster for small panels due to its lower parallelization overhead, while the *full strategy* is expected to be faster for large panels due to its higher degree of parallelism. Figure 3 shows the execution times per iteration of the outer loop for both strategies for $p = 48$ and $n = 4000$. For the first 20 or so iterations, the *full strategy* is faster, while the opposite is true for the remaining iterations. Hence, s needs tuning to find which strategy to use for each iteration of a reduction. For a smaller n and the same fixed parameters, the resulting figure is a subset of Fig. 3, e.g., for $n = 2000$, the resulting figure consists of iterations 40 to 80 of Fig. 3.

Table 1. Default values for the algorithmic parameters.

Parameter	Default
Panel width	$b = 50$
Thread count	$p_1 = p_2 = p$
Parallelization strategy	$s =$ partial

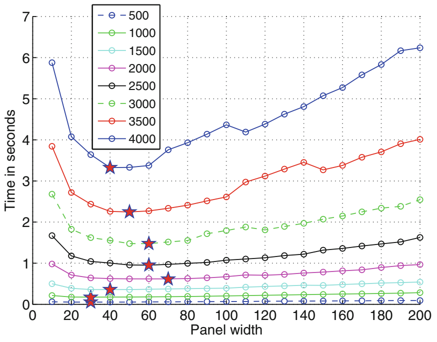

Fig. 2. Effect of the panel width on the execution time for $p = 48$ and $n \in \{500, 1000, \ldots, 4000\}$ with all other parameters as in Table 1. The stars represent the best b for each n.

Tuning Potential for the Thread Counts. The number of threads used in each phase affects the performance since it affects both the cache access patterns and the parallel overhead. To find the optimal configuration it suffices to know the execution time of each of the two phases in every iteration for each thread count since the phases do not overlap. These data can be obtained by repeating the same execution with different fixed thread counts. The time measurements are collected in two tables: T_1 for the first phase and T_2 for the second phase (not explicitly showed). One row per thread count and one column per iteration. To find the optimal thread count for a particular phase and iteration, one scans the corresponding column of the appropriate table and selects the thread count (row) with the smallest entry. Figure 4 compares the effect of varying the thread counts as opposed to always using the maximum number (48). The result shows that varying the thread counts is better, which means we need to tune the thread counts for each phase and iteration.

More Evaluation Results. A more thorough evaluation is discussed in the technical report [6]. Specifically, the report includes an evaluation of varying the panel width at each iteration of the reduction. The results show that the gain is insignificant compared to varying the panel width once per reduction. The evaluation of either performing the explicit data redistribution (copying to local buffers) or not is also included. The results show that it is always useful to redistribute the data. In addition, more cases for evaluating the effect of varying the thread counts are considered. The cases include experimenting with varying either p_1 or p_2 while fixing the other to the max, varying both but keeping $p_1 = p_2$, testing for a different problem size ($n = 4000$), and distributing the threads to the cores in two scheme: packed and round-robin. The general conclusion of all these cases is that p_1 and p_2 need to be tuned independently.

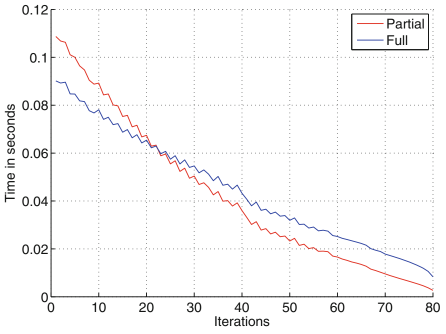

Fig. 3. Comparison of the full and partial strategies for $p = 48$ and $n = 4000$ with all other parameters as in Table 1.

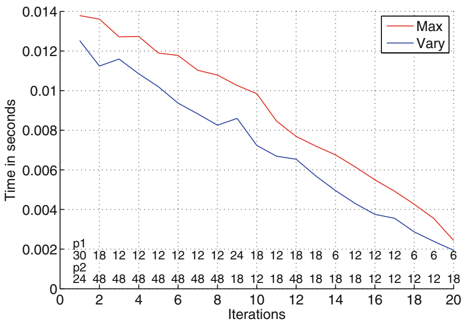

Fig. 4. Comparison of varying the thread counts and using maximum number of cores (48) for $n = 1000$ with all other parameters as in Table 1. The numbers at the bottom of the figure are the thread counts used in each iteration for each phase.

5 Performance Comparisons

This section illustrates the performance of the new parallel algorithm after tuning and compares it with LAPACK and ScaLAPACK over a range of problem sizes.

Off-Line Auto-tuning. To tune the parameters we used several rounds of *univariate search.* Our objective is not to come up with the best off-line auto-tuning mechanism but rather to get a rough idea how the new algorithm performs after tuning. Univariate search works by optimizing one variable at a time, in this case through exhaustive search, while fixing the other variables. The parameters are tuned separately for each problem size and number of cores.

Hessenberg reduction with and without PCA. Figure 5 shows the speed up of the Hessenberg algorithm with PCA against without PCA. The LAPACK routine `DGEHRD` was used as the variant without PCA since it is the closest in its implementation to the new algorithm. The comparison made for square matrices of size $n \in \{100, 300, \ldots, 3900\}$ using $p \in \{6, 12, \ldots, 48\}$. To have a fair comparison, the parameters of the PCA variant are fixed to the default values in Table 1. The results show that for most cases the PCA variant is faster.

Performance of The New Algorithm. To measure the new algorithm performance, tests are run on square matrices of size $n \in \{100, 300, \ldots, 3900\}$ using $p \in \{6, 12, \ldots, 48\}$ threads with 15 rounds of tuning. Figure 6 shows the performance measured in GFLOPS of the new algorithm after tuning on different numbers of cores. It is inconvenient to present all the parameter values in all tests since there are thousands of them. The results show that for small problems ($n \lesssim 2000$), it is not optimal to use the maximum number of cores (48).

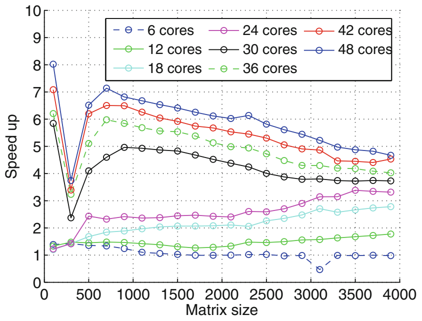

Fig. 5. Speed up comparison between the Hessenberg reduction algorithm with PCA, using the default parameters in Table 1, and without PCA.

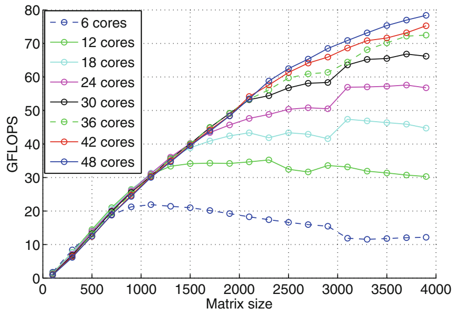

Fig. 6. Performance of the new algorithm using 1–8 NUMA domains.

Comparison with LAPACK and ScaLAPACK. Figure 7 shows the speed up of the new algorithm after tuning against the DGEHRD routine from LAPACK and the PDGEHRD routine from ScaLAPACK. The three routines are run using $p \in \{6, 12, 18, \cdots, 48\}$ threads for each problem of size $n \in \{100, 300, \cdots, 3900\}$. The numbers in the figure indicate for each implementation which p gives the best performance for each n. The comparison for each n is then made between the best case of the three implementations. Table 2 shows the values of b and s which are used in the new algorithm for each best case. For $n \geq 3100$, the *full strategy* is used for the first few iterations then the *partial strategy* is used. It is inconvenient

Table 2. The panel widths and strategies of the new algorithm after tuning for the cases used in the comparison in Fig. 7.

n	b	s	n	b	s
100	30	Partial	2100	60	Partial
300	30	Partial	2300	60	Partial
500	30	Partial	2500	60	Partial
700	30	Partial	2700	50	Partial
900	40	Partial	2900	60	Partial
1100	40	Partial	3100	60	Full until 4
1300	40	Partial	3300	60	Full until 7
1500	40	Partial	3500	60	Full until 11
1700	50	Partial	3700	60	Full until 14
1900	60	Partial	3900	60	Full until 19

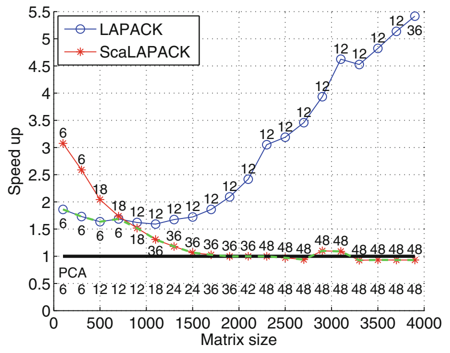

Fig. 7. Best case speed up comparison between our new algorithm after tuning and its counterparts in LAPACK and ScaLAPACK (block size 50 × 50). The numbers in the figure show the value of p which gives the best performance for each n.

to present the values of p_1 and p_2 for each case. Instead, we summarize how they change during the reduction. Generally, any reduction starts with $p_1 = p_2 = p$, then p_1 gradually decreases until it reaches the minimum number of threads (6), while p_2 decreases but less gradually and does not necessarily reaches the minimum. The results show that the new algorithm outperforms LAPACK for all the tested problems while it outperforms ScaLAPACK only for small problems ($n \lesssim 1500$), a possible reason is that ScaLAPACK might be using local memory access for both phases.

Comparison with Other Libraries. There are other libraries for numerical linear algebra than LAPACK and ScaLAPACK. The latest release (2.8) of the PLASMA [3] library does not support Hessenberg reduction, while MAGMA [13] uses GPU which is not our focus. On the other hand, libFLAME [14] uses the LAPACK routine for a counterpart implementation, while the implementation from Elemental library [10] produces comparable results to ScaLAPACK in the best case speed up comparison.

6 Conclusion

We presented a new parallel algorithm for Hessenberg reduction which applies the PCA technique to an existing algorithm. The algorithm is aimed to speed up the costly AED procedure which lies on the critical path of the distributed parallel multi-shift QR algorithm [7]. The proposed algorithm has a high degree of flexibility (due to tens or hundreds of tunable parameters) and memory locality (due to the application of PCA). The impact of various algorithmic parameters of the new algorithm were evaluated. The panel width, the parallelization strategy and the thread counts found to have a significant impact on the algorithm performance and though they need tuning. A basic off-line auto-tuning using univariate search is used to tune the parameters. The proposed solution with tuning outperforms LAPACK's routine `DGEHRD` for all cases and ScaLAPACK's routine `PDGEHRD` for small problem sizes.

Future work includes designing an on-line auto-tuning mechanism. The aim is to obtain an implementation that continuously improves itself the more it is being used. A major challenge is how to effectively handle the per-iteration parameters (thread count and parallelization strategy) as well as how to share information across nearby problem sizes.

Acknowledgements. 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. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. Part I: maintaining well-focused shifts and level 3 performance. SIMAX **23**(4), 929–947 (2002). https://doi.org/10.1137/S0895479801384573
2. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. Part II: aggressive early deflation. SIMAX **23**(4), 948–973 (2002). https://doi.org/10.1137/S0895479801384585
3. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009). https://doi.org/10.1016/j.parco.2008.10.002
4. Castaldo, A., Whaley, R.C.: Achieving scalable parallelization for the Hessenberg factorization. In: 2011 IEEE International Conference on Cluster Computing (CLUSTER), pp. 65–73. IEEE (2011). https://doi.org/10.1109/CLUSTER.2011.16
5. Castaldo, A., Whaley, R.C., Samuel, S.: Scaling LAPACK panel operations using parallel cache assignment. ACM TOMS **39**(4), 22 (2013). https://doi.org/10.1145/2491491.2491493
6. Eljammaly, M., Karlsson, L., Kågström, B.: Evaluation of the tunability of a new NUMA-aware Hessenberg reduction algorithm. NLAFET Working Note 8, December 2016. Also as Report UMINF 16.22, Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden
7. Granat, R., Kågström, B., Kressner, D., Shao, M.: Algorithm 953: parallel library software for the multishift QR algorithm with aggressive early deflation. ACM Trans. Math. Softw. **41**(4), 1–23 (2015). https://doi.org/10.1145/2699471. Article no. 29
8. Hasan, M.R., Whaley, R.C.: Effectively exploiting parallel scale for all problem sizes in LU factorization. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1039–1048. IEEE (2014). https://doi.org/10.1109/IPDPS.2014.109
9. Karlsson, L., Kågström, B.: Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures. Parallel Comput. **37**(12), 771–782 (2011). https://doi.org/10.1016/j.parco.2011.05.001. 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA 2010)
10. Poulson, J., Marker, B., van de Geijn, R.A., Hammond, J.R., Romero, N.A.: Elemental: a new framework for distributed memory dense matrix computations. ACM Trans. Math. Softw. **39**(2), 13:1–13:24 (2013). https://doi.org/10.1145/2427023.2427030
11. Quintana-Ortí, G., van de Geijn, R.: Improving the performance of reduction to Hessenberg form. ACM TOMS **32**(2), 180–194 (2006). https://doi.org/10.1145/1141885.1141887
12. Schreiber, R., Loan, C.V.: A storage efficient WY representation for products of Householder transformations. Technical report, no. 1 (1989). https://doi.org/10.1137/0910005
13. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput. **36**(5–6), 232–240 (2010). https://doi.org/10.1016/j.parco.2009.12.005
14. Zee, F.G.V., Chan, E., van de Geijn, R.A., Quintana-Ortí, E.S., Quintana-Ortí, G.: The libflame library for dense matrix computations. Comput. Sci. Eng. **11**(6), 56–63 (2009). https://doi.org/10.1109/MCSE.2009.207