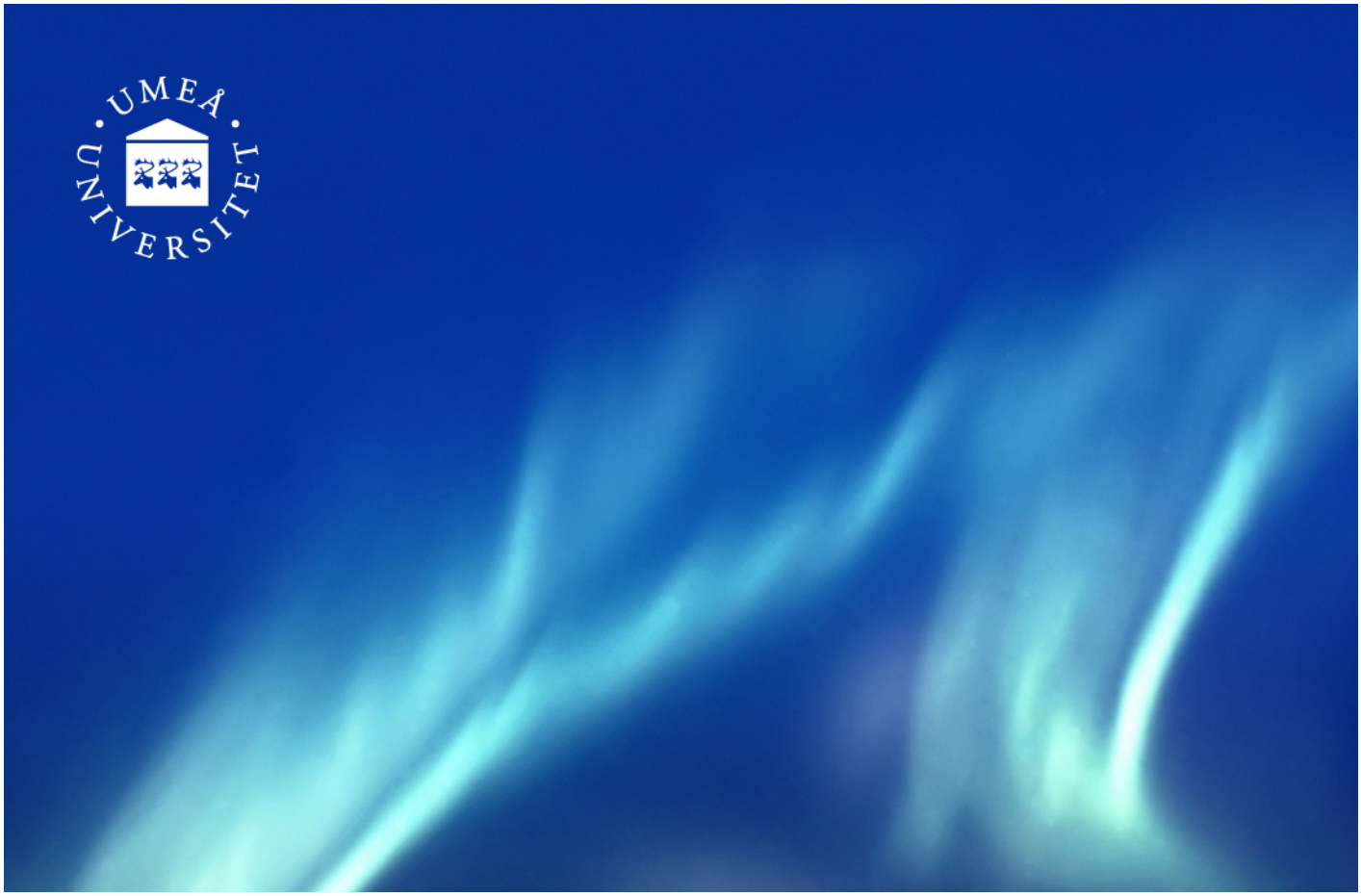

Efficient Reduction from Block Hessenberg Form to Hessenberg Form using Shared Memory

Lars Karlsson and Bo Kågström

UMINF 10.12

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
Sweden

Efficient Reduction from Block Hessenberg Form to Hessenberg Form using Shared Memory

Lars Karlsson[1] and Bo Kågström[1]

Department of Computing Science and HPC2N,
Umeå University, Umeå, Sweden
{larsk,bokg}@cs.umu.se

Abstract. A new cache-efficient algorithm for reduction from block Hessenberg form to Hessenberg form is presented and evaluated. The algorithm targets parallel computers with shared memory. One level of look-ahead in combination with a dynamic load-balancing scheme significantly reduces the idle time and allows the use of coarse-grained tasks. The coarse tasks lead to high-performance computations on each processor/core. Speedups close to 13 over the sequential unblocked algorithm have been observed on a dual quad-core machine using one thread per core.

Keywords: Hessenberg reduction, block Hessenberg form, parallel algorithm, dynamic load-balancing, blocked algorithm, high performance

1 Introduction

We say that an $n \times n$ matrix A with zeroes below its r-th subdiagonal, i.e., $A(i,j) = 0$ if $i > j + r$, is in (upper) *Hessenberg form* if $r = 1$ and in (upper) *block Hessenberg form* if $r > 1$. The number of (possibly) nonzero subdiagonals is thus equal to r.

The *Hessenberg decomposition* $A = QHQ^T$ of a square matrix consists of an orthogonal matrix Q and a matrix H in upper Hessenberg form. The Hessenberg decomposition is a fundamental tool in numerical linear algebra and has many diverse applications. For example, a Schur decomposition is typically computed using the nonsymmetric QR algorithm with an initial reduction to Hessenberg form. Other applications include solving Sylvester-type matrix equations.

We focus in this paper on the special case when A is in block Hessenberg form. There are certainly applications where this case occurs naturally. However, our interest is primarily motivated by the fact that finding a Hessenberg decomposition of a matrix in block Hessenberg form is the second stage in a two-stage approach for Hessenberg reduction of general matrices. The first stage is reduction to block Hessenberg form and it can be implemented efficiently as described in, e.g., [2, 6].

In the following, we describe and evaluate a new high-performance algorithm that finds a Hessenberg decomposition of an $n \times n$ block Hessenberg matrix A with $1 < r \ll n$ nonzero subdiagonals.

Algorithm 1 (Unblocked) Given a block Hessenberg matrix $A \in \mathbb{R}^{n\times n}$ with r nonzero subdiagonals, the following algorithm overwrites A with $H = Q^T AQ$ where $H \in \mathbb{R}^{n\times n}$ is in upper Hessenberg form and $Q \in \mathbb{R}^{n\times n}$ is orthogonal.

```
1: for j = 1:n-2
2:    k_1 = 1 + ⌊(n-j-2)/r⌋
3:    for k = 0:k_1 - 1
4:       α_1 = j + kr + 1;    α_2 = min{α_1 + r - 1, n}
5:       β_1 = j + max{0, (k-1)r + 1};    β_2 = n
6:       γ_1 = 1;    γ_2 = min{j + (k+2)r, n}
7:       Reduce A(α_1:α_2, β_1) using a reflection Q_k^j
8:       A(α_1:α_2, β_1:β_2) = (Q_k^j)^T A(α_1:α_2, β_1:β_2)
9:       A(γ_1:γ_2, α_1:α_2) = A(γ_1:γ_2, α_1:α_2) Q_k^j
10:   end for
11: end for
```

2 Algorithms

Our blocked algorithm evolved from a known unblocked algorithm for symmetric band reduction [1] adapted to the nonsymmetric case. Since the understanding of the unblocked algorithm is crucial, we begin by describing it.

2.1 Unblocked algorithm

Algorithm 1 reduces the columns of A from left to right [1, 4, 5]. Consider the first iteration of the outer loop, i.e., $j = 1$. In the first iteration of the inner loop, $k = 0$ and a Householder reflection, Q_0^1, of order r is constructed on line 7. Lines 8–9 apply a similarity transformation that reduces the first column and also introduces an $r \times r$ bulge of fill-in elements in the strictly lower triangular part of $A(r+2:2r+1, 2:r+1)$.

The next iteration of the inner loop, i.e., $k = 1$, constructs and applies the reflection Q_1^1 of order r which reduces the first column of the bulge. This introduces another bulge r steps further down the diagonal. The subsequent iterations of the inner loop reduce the first column of each newly created bulge.

The second iteration of the outer loop, i.e., $j = 2$, reduces column two and the leftmost column of all bulges that appear. Note that the new bulges align with the partially reduced bulges from the previous iteration. The bulges thus move one step down the diagonal in each iteration of the outer loop.

Applying a Householder reflection of order r to a vector involves $4r$ flops. The flopcount of Algorithm 1 is thus $2n^3$ plus lower order terms. During one iteration of the outer loop, each entry in the matrix is involved in exactly 0, 1, or 2 reflections. This means that Algorithm 1 has a low arithmetic intensity (flops per main memory reference) and its performance is ultimately bounded by the memory bandwidth.

The reflections can be accumulated into a dense orthogonal matrix of order n for the cost of $2n^3$ additional flops. Efficient accumulation is accomplished using an alternative ordering of the reflections as described in [1].

2.2 Blocked algorithm

The key to increasing the arithmetic intensity of Algorithm 1, and thus creating a blocked algorithm, is to obtain reflections from multiple consecutive *sweeps*[1] and then apply them in a different order. The reflections are generated in the order of increasing j and increasing k for each j. A more efficient way to apply them, however, is in the order of decreasing k and increasing j for each k [1]. The reason for better efficiency is that the q reflections Q_k^j for $j = j_1 : j_1 + q - 1$ touch only $r + q - 1$ unique entries in each row/column of the matrix. Thus, the arithmetic intensity can be increased almost by a factor of q. The primary problem is how to use this trick when updating A itself and not only when accumulating the transformations. Below, we describe a solution to this problem.

Overview of the blocked algorithm. Fundamentally, the blocked algorithm consists of a sequence of iterations consisting of three main steps. In the first step, all reflections from q consecutive sweeps are generated while only necessary updates are applied. In the second/third step, the remaining updates from the right-hand/left-hand side are applied. Due to dependencies, these steps must be done in sequence. The first step is both time-consuming and sequential in nature so the three-step iteration can potentially cause large amounts of parallel overhead due to idle processors/cores. Therefore, we bisect the second and third steps to create a five-step iteration. This is enough to support one level of look-ahead, as we later show.

The purpose of each step is explained below.

1. Generate reflections from q consecutive sweeps (label: G).
2. Apply updates from the right-hand side to enable look-ahead (label: P(R)).
3. Apply updates from the left-hand side to enable look-ahead (label: P(L)).
4. Apply the remaining updates from the right-hand side (label: U(R)).
5. Apply the remaining updates from the left-hand side (label: U(L)).

Steps 4–5 are the most efficient in terms of memory traffic and parallel execution. Steps 2–3 are less efficient but still worthwhile to parallelize. Step 1 is mostly sequential and cannot make efficient use of the cache hierarchy.

Figure 1 shows an example where the entries touched by each step are distinguished. Note that the G, P(R), and P(L) steps combined touch entries which are close to the diagonal. The thickness of the band (*light and medium gray* in Figure 1) depends on the number of subdiagonals, r, and the number of consecutive sweeps, q, but is independent of the matrix size. The algorithm as a whole therefore performs $O(n^3)$ flops in a cache-efficient manner, i.e., during U(R) and U(L), and only $O(n^2)$ in more or less inefficient ways, i.e., during G, P(R), and P(L).

Generate reflections from multiple consecutive sweeps. The core of our algorithm is the G-step which is detailed in Algorithm 2. This algorithm gener-

[1] A *sweep* corresponds to the computations in one iteration of the outer loop in Algorithm 1.

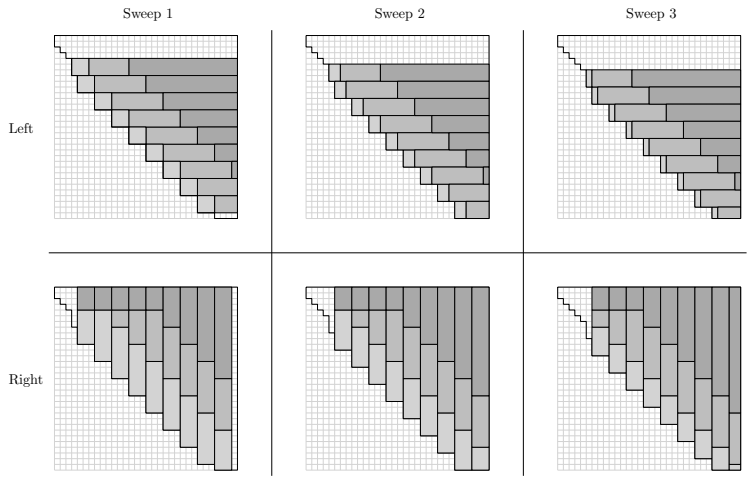

Fig. 1. Entries touched in each of the five steps: G (*light gray*), P(R) (*medium gray, bottom*), P(L) (*medium gray, top*), U(R) (*dark gray, bottom*), and U(L) (*dark gray, top*). Transformations from the left-hand side (*top*) are shown separately from the transformations from the right-hand side (*bottom*). The $q = 3$ sweeps are shown separately from left to right.

ates $Q_\star^j$ for q consecutive sweeps starting at column $j = j_1$. Only a few entries near the main diagonal are updated in the process (*light gray* in Figure 1).

The structure of Algorithm 2 is as follows. The outer j-loop steps through the q sweeps starting at column j_1. The k-loop steps through the k_1 reflections contained in this sweep. First, the column that is about to be reduced is brought up-to-date by applying delayed updates from the left (if any) in the loop that starts on line 6. The actual reduction is performed on line 13. Some of the updates from the right-hand side gets applied on line 16.

Apply updates efficiently. Algorithm 3 applies the updates that were not performed in the G-step. The algorithm implements all four of the steps P(R), P(L), U(R), and U(L) since they are similar. The arithmetic intensity is increased by using the reordering trick from [1].

To facilitate parallel execution, Algorithm 3 restricts the updates from the left-hand side to the column range $c_1 : c_2$ and the updates from the right-hand side to the row range $r_1 : r_2$. By partitioning the range $1 : n$ into p disjoint ranges, the same variant of Algorithm 3 can be concurrently executed by p threads without any need for synchronization. The arbitrary partitioning can be exploited in order to balance the load.

Algorithm 2 (Generate) Given a block Hessenberg matrix $A \in \mathbb{R}^{n\times n}$ with r nonzero subdiagonals which is already in upper Hessenberg form in columns $1:j_1-1$, the following algorithm generates $Q_\star^j$ for $j = j_1:j_1+q-1$. The matrix A is partially overwritten by the similarity transformation implied by the q sweeps. The remaining updates can be applied using Algorithm 3.

for $j = j_1:j_1+q-1$
  $k_1 = 2 + \left\lfloor \frac{n-j-1}{r} \right\rfloor$
  for $k = 0:k_1-1$
    $\alpha_1 = j + kr + 1; \quad \alpha_2 = \min\{\alpha_1 + r - 1, n\}$
    $\beta = j + \max\{0, (k-1)r + 1\}$
    for $\hat{j} = j_1:j-1$
      $\hat{\alpha}_1 = \hat{j} + kr + 1; \quad \hat{\alpha}_2 = \min\{\hat{\alpha}_1 + r - 1, n\}$
      if $\hat{\alpha}_2 - \hat{\alpha}_1 + 1 \geq 2$ **then**
        $A(\hat{\alpha}_1:\hat{\alpha}_2, \beta) = (Q_k^{\hat{j}})^T A(\hat{\alpha}_1:\hat{\alpha}_2, \beta)$
      end if
    end for
    if $\alpha_2 - \alpha_1 + 1 \geq 2$ **then**
      Reduce $A(\alpha_1:\alpha_2, \beta)$ using a reflection Q_k^j
      $A(\alpha_1:\alpha_2, \beta) = (Q_k^j)^T A(\alpha_1:\alpha_2, \beta)$
      $\gamma_1 = j_1 + 1 + \max\{0, (k + j - j_1 - q + 2)r\}; \quad \gamma_2 = \min\{j + (k+2)r, n\}$
      $A(\gamma_1:\gamma_2, \alpha_1:\alpha_2) = A(\gamma_1:\gamma_2, \alpha_1:\alpha_2)Q_k^j$
    end if
  end for
end for

3 Parallel Issues

To develop a shared-memory implementation of the new algorithm we have to decide how to decompose the five steps into independent tasks, how to map the tasks to threads, and how to synchronize the threads to honor the dependencies. Moreover, the load must be balanced to achieve high parallel efficiency.

3.1 Task decomposition and dependencies

Figure 2 illustrates all direct dependencies between the steps of four consecutive iterations. The steps within one iteration are layed out horizontally. Note in particular that the U(R)-step is not dependent on the P(L)-step and that the G-step of the next iteration can begin as soon as the P(L)-step finishes. This latter fact is what makes look-ahead possible.

The polygons in Figure 2 must execute sequentially even though some steps may overlap within each polygon. Thus it is appropriate to implement the computation as a loop where each iteration corresponds to a polygon in Figure 2. The *prologue* and *epilogue* polygons correspond to the computations before and after the look-ahead loop, respectively. The steps PU(R) and PU(L) in the epilogue represent the merging of P(R) with U(R) and of P(L) with U(L), respectively. This is a more efficient way to apply the updates when no look-ahead is desired.

Algorithm 3 (Update) Given a block Hessenberg matrix $A \in \mathbb{R}^{n\times n}$ with r nonzero subdiagonals, the following algorithm applies $Q_\star^j$ for $j = j_1 : j_1 + q - 1$ from both sides. This algorithm consists of four variants (P(R), P(L), U(R), and U(L)). Together they complete the updates which were not performed by Algorithm 2. The range of rows/columns updated is controlled by $r_1 : r_2$ and $c_1 : c_2$, respectively

$k_1 = 1 + \left\lfloor \frac{n-j_1-2}{r} \right\rfloor$
for $k = k_1 - 1 : -1 : 0$
  for $j = j_1 : j_1 + q - 1$
    $\alpha_1 = j + kr + 1; \quad \alpha_2 = \min\{\alpha_1 + r - 1, n\}$
    if $\alpha_2 - \alpha_1 + 1 \geq 2$ **then**
      if variant is P(R) or U(R) **then**
        if variant is P(R) **then**
          $\gamma_1 = \max\{r_1, j_1 + (k + j - j_1 - 2q + 2)r + 1\}$
          $\gamma_2 = \min\{r_2, j_1 + \max\{0, (k + j - j_1 - q + 2)r\}\}$
        else if variant is U(R) **then**
          $\gamma_1 = \max\{r_1, 1\}$
          $\gamma_2 = \min\{r_2, j_1 + (k + j - j_1 - 2q + 2)r\}$
        end if
        $A(\gamma_1 : \gamma_2, \alpha_1 : \alpha_2) = A(\gamma_1 : \gamma_2, \alpha_1 : \alpha_2)Q_k^j$
      else if variant is P(L) or U(L) **then**
        if variant is P(L) **then**
          $\beta_1 = \max\{c_1, j_1 + q, j_1 + q + (k - 1)r + 1\}$
          $\beta_2 = \min\{c_2, j_1 + q + (k + q - 1)r\}$
        else if variant is U(L) **then**
          $\beta_1 = \max\{c_1, j_1 + q + (k + q - 1)r + 1\}$
          $\beta_2 = \min\{c_2, n\}$
        end if
        $A(\alpha_1 : \alpha_2, \beta_1 : \beta_2) = (Q_k^j)^T A(\alpha_1 : \alpha_2, \beta_1 : \beta_2)$
      end if
    end if
  end for
end for

3.2 Parallel execution

Following the dependencies in Figure 2, we see that step P(R) must be completed before either of the steps P(L) or U(R) may start. This implies a barrier-style synchronization. Next, we could potentially do both P(L) and U(R) concurrently. However, heuristically we would like to start the G-step as soon as possible. Thus we do the P(L)-step before doing the steps G and U(R) in parallel. Note that the G-step must not start until P(L) has completed, but U(R) can start at any time. Thus, the synchronization implied at this point is weaker than a barrier. Proceeding to the U(L)-step we see that both U(R) and P(L) must complete, so again we have a barrier-style synchronization. At the end of the iteration there is a third and final barrier since all steps must complete before the next iteration.

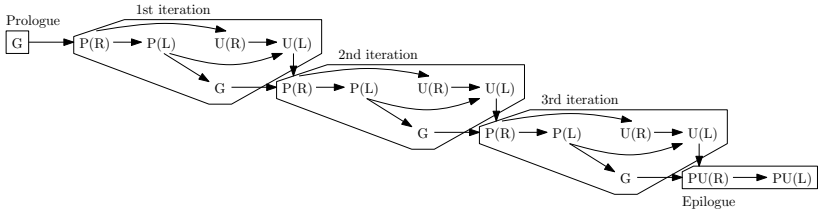

Fig. 2. Dependence graph for a problem consisting of four iterations (separated vertically). The steps of one iteration are layed out horizontally. Polygons enclose the steps within the prologue, epilogue, and looka-head iterations, respectively.

The large number of inherent barrier-style synchronization points, namely $O(n/q)$, are a cause for concern if we want to use dynamic scheduling of fine-grained tasks. The two primary reasons are that (i) fine-grained tasks cannot be executed as efficiently and cause more overhead than large-grained tasks, and (ii) the barriers introduce overhead due to idle processors/cores. We therefore adopted a model-driven dynamic load-balancing scheme capable of supporting coarse-grained tasks.

Task mapping. As the G-step is difficult to parallelize, we have chosen to keep it sequential. The other steps can be parallelized simply by partitioning the rows/columns.

The main problem is how to map tasks to threads in a way that minimizes the idle time caused by the implicit barriers. Thread p_0 could potentially participate in the computation of both $U(R)$ and $U(L)$, only $U(L)$, or none of them. The correct choice depends on how long it takes to complete the G-step. Figure 3 shows three hypothetical scenarios. In Figure 3(a), the G-step is the limiting

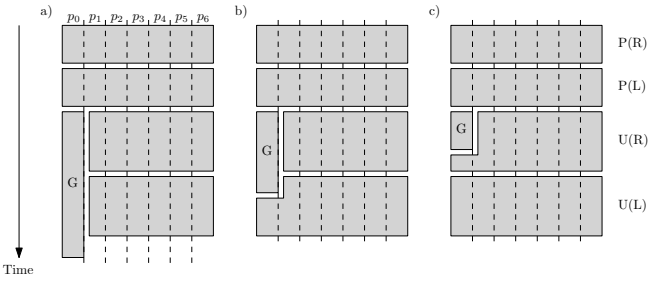

Fig. 3. Three scenarios for parallel execution of one look-ahead iteration.

factor and p_0 should not take part in the updates. In Figure 3(b), the G-step finishes half-way into step U(L) and a (small) piece of the U(L)-step should thus be assigned to thread p_0. In Figure 3(c), the G-step finishes before the U(R)-step

and now p_0 should be assigned a (small) piece of U(R) as well as a (large) piece of U(L).

Dynamic load balancing. The problem at hand is the following. Given p threads and a step (P(R), P(L), U(R), or U(L)), find a row/column partitioning corresponding to p tasks mapped to p threads such that if thread i starts executing at time a_i then all threads finish their respective task at the same time. By using an appropriate model to simplify the problem, we solve it exactly. Under reasonable assumptions, the obtained solution is an approximate solution to the original problem.

Let $f(x)$ be a function defined at the integer values $x = 1:n$ that gives the number of flops applied to the entries on row/column x by a particular step. The total number of flops is therefore $W = \sum_{x=1}^{n} f(x)$.

Assume for a moment that we know that thread i will execute its next task with an average speed of s_i flops per second. If this task requires ω_i flops and the thread starts executing it at time a_i, then it finishes at time $t_i = a_i + \omega_i s_i$.

The optimal execution time, $t_{\min}$, is obtained by solving $\int_0^x s(t)\ dt = W$ for x where $s(t)$ is the sum of the speeds s_i for all i such that $a_i \leq t$. The corresponding task sizes can be computed from $\omega_i = s_i \cdot \max\{0, t_{\min} - a_i\}$. Finally, a partitioning of the range $1:n$ is constructed from the task sizes by solving $\sum_{x=1}^{x_i} f(x) = \sum_{j=0}^{i-1} \omega_j$ for x_i where $i = 1:p-1$. The range of rows/columns assigned to thread i is $x_i : x_{i+1} - 1$ where we use the convention that $x_0 = 1$ and $x_p = n + 1$.

The accuracy of the scheme clearly depends on how well we can guess the future speeds s_i. A good predictor in this case comes from the previously observed speeds. Specifically, we measure the actual execution time $\hat{t}_i$ and use the guess $s_i = \omega_i / \hat{t}_i$ for the next iteration. We expect this to work reasonable well because there are strong similarities between the load-balancing problems in consecutive iterations.

4 Experiments

Tests were run on the Akka system at HPC2N. Akka is a cluster with dual Intel Xeon L5420 nodes (4 cores per socket) with a double precision theoretical peak performance of 80 Gflop/s (10 Gflop/s per core and 8 cores in total).

We compare four different implementations to give an idea of the impact of various aspects of our implementation. The first is an implementation of Algorithm 1 which we labeled Unblocked. The second, labeled Basic, is a variant of our blocked algorithm without both look-ahead and adaptive modeling of the speeds. The third, labeled Basic + Adapt, includes adaptive modeling. The fourth implementation, labeled Look-ahead + Adapt, includes both look-ahead and adaptive modeling.

The unblocked implementation is sequential while the others are parallel. All parallel executions use one thread per core (8 threads in total). We performed each experiment twice and selected the shortest execution time.

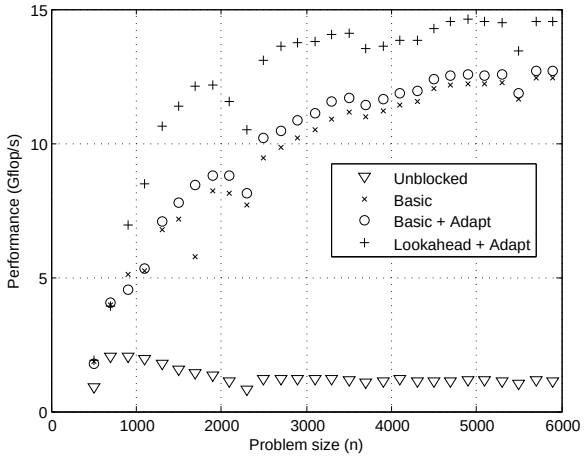

Fig. 4. Performance ($r = 12$, $q = 16$).

The performance, which we calculate as $2n^3/t$ where t is the execution time, is illustrated in Figure 4. As expected, the unblocked implementation is quite slow since not only is it sequential but it also causes a lot of memory traffic. The most advanced implementation, Look-ahead + Adapt, comes out on top and peaks close to 15 Gflop/s (19% of the theoretical peak). The corresponding speedups over Unblocked is shown in Figure 5. The look-ahead technique adds a significant performance boost (15–50%) as shown in Figure 6.

The idle time gets substantially reduced by the addition of adaptation and the look-ahead implementation incurs relatively small amounts of idle time. Figure 7 illustrates this by showing the measured idle time per iteration on an $n = 3000$, $r = 12$, $q = 16$ problem. The improvements obtained when going from Basic to Basic + Adapt are obious. However, it is not possible to isolate the effect of the adaptive modeling in the look-ahead implementation since the adaptation is an integral part of the look-ahead approach (see Figure 3).

5 Conclusion

We have presented a new blocked high-performance shared-memory implementation of a Householder-based algorithm for reduction from block Hessenberg form to Hessenberg form. The implementation delays updates and applies them cache-efficiently in parallel. One level of look-ahead in conjunction with an adaptive coarse-grained load-balancing scheme significantly improves the performance. A performance of 15 Gflop/s (19% of the theoretical peak) has been observed on a dual quad-core machine. This corresponds to a speedup close to 13 over a sequential unblocked algorithm.

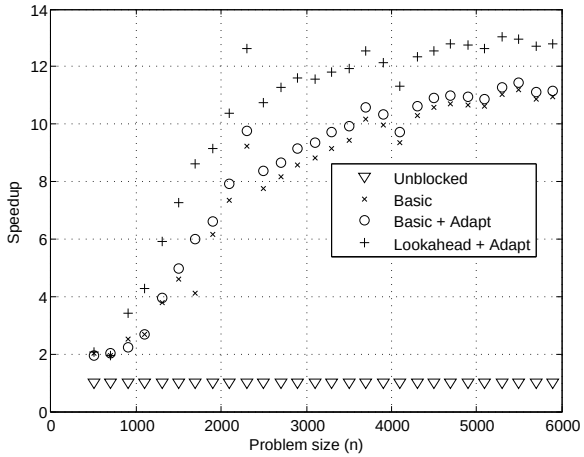

Fig. 5. Speedup over Unblocked ($r = 12$, $q = 16$).

5.1 Related work

A different approach to increase the arithmetic intensity of the reduction from block Hessenberg form to Hessenberg form is presented in [3]. A sliding computational window is employed to obtain a set of reflections while at the same time delaying most of the updates. The off-diagonal blocks are then updated efficiently. However, the focus in [3] is on algorithms which are theoretically I/O-efficient and it is also primarily concerned with efficiency in the asymptotic sense. Specifically, the proposed blocking appears to be impractical for matrices that fit entirely in main memory since the computational window would have to be relatively large in order to obtain even modest improvements to the arithmetic intensity.

Acknowledgements. We would like to thank Daniel Kressner for his support and motivating discussions. This research was conducted using the resources of High Performance Computing Center North (HPC2N) and was supported by the *Swedish Research Council* under grant VR7062571 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.

References

1. Bischof, C.H., Lang, B., Sun, X.: A framework for symmetric band reduction. ACM Trans. Math. Software 26(4), 581–601 (2000)
2. Ltaief, H., Kurzak, J., Dongarra, J.J., Badia, R.M.: Scheduling two-sided transformations using tile algorithms on multicore architectures. Scientific Programming 18(1), 35–50 (2010)

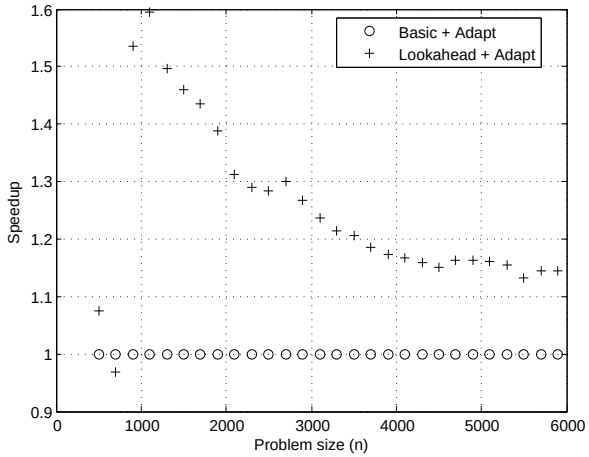

Fig. 6. Speedup over Basic+Adapt ($r = 12$, $q = 16$).

3. Mohanty, S.: I/O Efficient Algorithms for Matrix Computations. Ph.D. thesis, Indian Institute of Technology Guwahati (2009)
4. Murata, K., Horikoshi, K.: A new method for the tridiagonalization of the symmetric band matrix. Information Processing in Japan 15, 108–112 (1975)
5. Rutishauser, H.: On Jacobi rotation patterns. In: Proc. Sympos. Appl. Math. vol. XV, pp. 219–239. Amer. Math. Soc., Providence, R.I. (1963)
6. Tomov, S., Dongarra, J.J.: Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. Tech. Rep. UT-CS-09-642, University of Tennessee Computer Science (May 2009), also as LAPACK Working Note 219

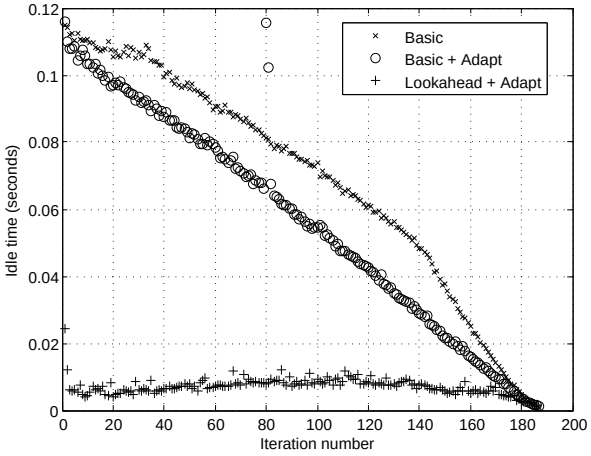

Fig. 7. Idle time per iteration ($n = 3000$, $r = 12$, $q = 16$).