<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper published in *Parallel Computing*. This paper has been peer-reviewed but does not include the final publisher proof-corrections or journal pagination.

Citation for the original published paper (version of record):

Karlsson, L., Kågström, B., Wadbro, E. (2013)

Fine-Grained Bulge-Chasing Kernels for Strongly Scalable Parallel QR Algorithms.

Parallel Computing

Access to the published version may require subscription.

N.B. When citing this work, cite the original published paper.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-79742>

Fine-Grained Bulge-Chasing Kernels for Strongly Scalable Parallel QR Algorithms

L. Karlsson^{a,*}, B. Kågström^a, E. Wadbro^a

^a*Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden*

Abstract

The bulge-chasing kernel in the small-bulge multi-shift QR algorithm for the non-symmetric dense eigenvalue problem becomes a sequential bottleneck when the QR algorithm is run in parallel on a multicore platform with shared memory. The duration of each kernel invocation is short, but the critical path of the QR algorithm contains a long sequence of calls to the bulge-chasing kernel. We study the problem of parallelizing the bulge-chasing kernel itself across a handful of processor cores in order to reduce the execution time of the critical path. We propose and evaluate a sequence of four algorithms with varying degrees of complexity and verify that a pipelined algorithm with a slowly shifting block column distribution of the Hessenberg matrix is superior. The load-balancing problem is non-trivial and computational experiments show that the load-balancing scheme has a large impact on the overall performance. We propose two heuristics for the load-balancing problem and also an effective optimization method based on local search. Numerical experiments show that speed-ups are obtained for problems as small as 40×40 on two different multicore architectures.

Keywords: fine-grained parallelism, scalability, load-balancing, load-balance optimization, auto-tuning

1. Introduction

The ratio between the problem size and the number of available processors greatly impacts the performance of a parallel algorithm. If this ratio is large, then the performance is often only limited by the number of processors; most processors are active most of the time. On the other hand, if the ratio is small, then the performance is often limited by the execution time of the critical path of the underlying task graph; enough processors are available to fully exploit the exposed parallelism. The performances of the computational kernels that make up the tasks on the critical path become important even if these kernels account for an asymptotically vanishing fraction of the total work. Hence, the performance on large problems can still be limited by the performance of many short-lived tasks performed by special-purpose computational kernels. Any speed-up, however small, obtained by optimizing and parallelizing these kernels translates directly into improved overall performance. In this paper, we develop fine-grained parallel algorithms for the bulge-chasing kernel appearing in the small-bulge multi-shift QR algorithm for the dense non-symmetric matrix eigenvalue problem [1, 2].

Following the success of the small-bulge multi-shift QR algorithm [1, 2] and its recent parallel implementation in the message passing paradigm [3, 4, 5], we are currently developing a strongly scalable shared-memory implementation of the QR algorithm. Typically, the most expensive part is the QR iterations, each of which introduces and chases a chain of 3×3 bulges from the top left to the bottom right corner of an upper Hessenberg matrix. Chasing a chain of k bulges through a Hessenberg matrix of size $n \times n$, where $k \ll n$, requires $\Theta(kn^2)$ floating point operations (flops) and the critical path accounts for $\Theta(k^2n)$ of these flops.

*Corresponding author.

Email addresses: larsk@cs.umu.se (L. Karlsson), bokg@cs.umu.se (B. Kågström), eddiew@cs.umu.se (E. Wadbro)

This means that the critical path can quickly become a sequential bottleneck in parallel implementations. State-of-the-art implementations of a QR iteration [1, 3] alternate between chasing the chain within a sub-matrix centered on the main diagonal—the so-called *computational window*—and updating the off-diagonal blocks using fast matrix–matrix multiplications. The computations within the computational window are performed by the *bulge-chasing kernel*, whose parallel implementation is the main subject of this paper. The bulge-chasing kernel accounts for a negligible fraction of the overall work and execution time in a sequential environment. However, since the off-diagonal updates are readily parallelizable and rely on the optimized level 3 BLAS `xGEMM`, the bulge-chasing kernel also becomes a sequential bottleneck in a parallel environment unless the problem is large enough. The size of the computational window determines the blocking factor and thereby the amount of memory traffic. The desired blocking factor has steadily increased to offset the growing gap between processor and memory speed. For problems larger than some cross-over point, the critical path no longer limits the performance. As a consequence of the blocking factor increasing over time, the cross-over point also drifts towards larger and larger problems. With several tens or hundreds of processors available on today’s and future multicore platforms, we therefore need to pay attention to the performance of the kernels on the critical path or otherwise risk underutilization of the system for a wide range of problem sizes.

Figure 1: Traces illustrating the potential benefits of parallelizing the bulge-chasing kernel (red). Left: The first of $p = 48$ threads is dedicated to the bulge-chasing kernels on the critical path, while the remaining $p - 1 = 47$ threads perform off-diagonal updates in the form of matrix–matrix multiplications (gray). Right: $q = 4$ threads process the bulge-chasing kernels in parallel. The remaining $p - q = 44$ threads perform the off-diagonal updates. Despite a 50% parallel efficiency for the parallelized kernel and an increased load on the remaining $p - q$ threads, the parallel execution time is still reduced by a factor of two. *Note: These traces illustrate our objective; actual traces are more complicated and would therefore obscure the main message.*

Figure 1 (left) illustrates how the sequence of invocations of the bulge-chasing kernel (red) becomes a sequential bottleneck when the off-diagonal updates are performed in parallel by the remaining $p - 1$ processors. The basic idea pursued in this paper is that by reserving $1 < q \ll p$ processors, instead of just one as in Figure 1 (left), for the bulge-chasing kernel, we can improve the overall performance by reducing the execution time of the critical path, see Figure 1 (right). Since $q \ll p$, the slow-down experienced for the off-diagonal updates, now performed by only $p - q$ processors, will be small in comparison to the gains made by parallelizing the bulge-chasing kernel. In this case, even if the kernel is parallelized with an efficiency of only 50% the overall performance doubles.

The time per invocation of the bulge-chasing kernel ranges from a few microseconds to a few milliseconds for typical problem sizes since the computational window is merely a small submatrix of a much larger matrix; a typical computational window does not contain more than a few hundred rows and columns. Efficient parallelization at such a fine granularity requires low-latency synchronization mechanisms. In particular, we cannot rely on standard synchronization primitives provided by the system, such as mutexes and conditional variables, due to the inherent overhead involved in transferring control to the operating system for the purpose of suspending and awaking threads. Instead, we resort to using atomic instructions to synchronize directly via the shared memory without involving system software. The main synchronization

problems are of the producer-consumer and barrier types and can be solved using standard techniques. One of the key challenges is to devise an effective load-balancing scheme for the bulge-chasing kernel. In general, sophisticated schemes tend to incur more overhead and hence might be inferior to simpler schemes for small problems. Since the problem sizes considered are typically small-sized, it is not at all obvious for what range of problem sizes, if any, a given algorithm is preferred. The cross-over points can, however, be readily found from measurements of actual implementations. We have therefore used the approach of developing a sequence of increasingly sophisticated algorithms and have used optimization to systematically determine the cross-over points. This approach also answers many questions of the type: How does the proposed algorithm(s) compare to simpler algorithms that have less overhead?

The rest of the paper is organized as follows. Section 2 provides some background for the algorithm used by the bulge-chasing kernel. In particular, Section 2.1 recalls the bulge-chasing mechanism [6, 7] and Section 2.2 recalls the standard technique for packing several small bulges in a tightly packed chain [1]. Readers already familiar with modern implementations of the QR algorithm can safely skip most of Section 2, except for Algorithm 0, which formally defines the bulge-chasing kernel as it is used in this paper. Section 3 presents our main contributions, namely a sequence of four parallel algorithms with an increasing level of sophistication with respect to their parallel decomposition and load-balancing schemes. The first three algorithms (Sections 3.1, 3.2, and 3.3) are similar and assign the tasks of performing the actual bulge-chasing steps to one thread. The fourth algorithm, presented in Section 3.4, treats the processors more uniformly and relies on pipelining to expose parallelism. A configurable partitioning of the matrix controls the distribution of work. Section 3.4.1 discusses the underlying task graph, Section 3.4.2 introduces the load-balancing problem and two heuristic solutions. Section 3.4.3 presents an optimization method based on local search that further tailors the load-balance parameters to a target system. Section 4 presents and analyzes some computational experiments and Section 5 concludes.

2. Mathematical background

The small-bulge multi-shift QR algorithm is quite complicated and there are many trade-offs involved in its sequential and parallel implementations. In this paper, we only consider the bulge-chasing kernel, which is one of the sequential bottlenecks of the QR algorithm. This section provides a sufficiently detailed description of the bulge-chasing kernel to understand the rest of the paper. For the difficult trade-offs associated with the complete QR algorithm, we refer the reader to, e.g., [1, 2, 6, 3, 5].

2.1. The bulge-chasing mechanism

The bulge-chasing kernel operates on an upper Hessenberg matrix containing a chain of tightly packed bulges. An *upper Hessenberg matrix* $H \in \mathbb{R}^{n \times n}$ is upper triangular with an additional non-zero subdiagonal. We further assume that the matrix is *unreduced*, i.e., that all entries on the subdiagonal are non-zero. A *bulge* is a dense contiguous submatrix of size 3×3 centered on the first subdiagonal. For example, the matrix

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} & h_{15} & h_{16} \\ h_{21} & h_{22} & h_{23} & h_{24} & h_{25} & h_{26} \\ 0 & \boxed{h_{32} & h_{33} & h_{34}} & h_{35} & h_{36} \\ 0 & \textcolor{red}{h_{42}} & h_{43} & h_{44} & h_{45} & h_{46} \\ 0 & \textcolor{red}{h_{52}} & \textcolor{red}{h_{53}} & h_{54} & h_{55} & h_{56} \\ 0 & 0 & 0 & 0 & h_{65} & h_{66} \end{bmatrix}$$

is upper Hessenberg and has a bulge on the second column. The bulge can be moved or *chased* down the subdiagonal one column at a time by constructing a 3×3 Householder reflector $Q = I - \tau vv^T$ such that

$$Q^T \begin{bmatrix} h_{32} \\ \textcolor{red}{h_{42}} \\ \textcolor{red}{h_{52}} \end{bmatrix} = \begin{bmatrix} h_{32} \\ \textcolor{red}{h_{42}} \\ \textcolor{red}{h_{52}} \end{bmatrix} - \tau \begin{bmatrix} 1 \\ v_2 \\ v_3 \end{bmatrix} \begin{bmatrix} 1 \\ v_2 \\ v_3 \end{bmatrix}^T \begin{bmatrix} h_{32} \\ \textcolor{red}{h_{42}} \\ \textcolor{red}{h_{52}} \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ 0 \end{bmatrix}. \quad (1)$$

By applying Q^T to rows 3:5 and Q to columns 3:5 of H , we perform an orthogonal similarity transformation of H resulting in the updated matrix

$$\hat{H} = \begin{bmatrix} h_{11} & h_{12} & h'_{13} & h'_{14} & h'_{15} & h_{16} \\ h_{21} & h_{22} & h'_{23} & h'_{24} & h'_{25} & h_{26} \\ 0 & \alpha' & h''_{33} & h''_{34} & h''_{35} & h'_{36} \\ 0 & 0' & h''_{43} & h''_{44} & h''_{45} & h'_{46} \\ 0 & 0' & h''_{53} & h''_{54} & h''_{55} & h'_{56} \\ 0 & 0 & h'_{63} & h'_{64} & h'_{65} & h_{66} \end{bmatrix}.$$

Entries marked with one apostrophe have been affected by one of the updates (either from the left by Q^T or from the right by Q) and those marked with two have been affected by both updates. The application of a Householder reflection as hinted at in (1) requires merely 10 flops (per row/column) when implemented as follows: dot product with the Householder vector (4 flops; 2 multiplications and 2 additions), scalar multiplication (1 flop), and vector update or axpy (5 flops; 2 multiplications and 3 subtractions). Consequently, performing a bulge-chasing step requires no more than $10(n+5)$ flops, not counting the small number of operations required to construct the reflector itself.

2.2. Tightly packed chains of bulges

We can chase a single bulge $n-2$ steps from the top left to the bottom right corner. This requires reading and writing a matrix of size $n \times n$ while performing only $10(n+5)(n-2) = 10n^2 + \mathcal{O}(n)$ flops. The low computational intensity would limit the overall performance of the QR algorithm if we used the single-bulge approach.

Figure 2: The bulge-chasing kernel chases a chain of bulges from the top left corner to the bottom right corner of a Hessenberg matrix $H \in \mathbb{R}^{n \times n}$ and accumulates the transformations in an orthogonal matrix $U \in \mathbb{R}^{n \times n}$. Left: The location of the chain when entering the kernel. Right: The location of the chain when leaving the kernel.

Therefore, the small-bulge multi-shift QR algorithm chases not one but several bulges packed tightly together in a *chain of bulges*, as illustrated in Figure 2 (left). A chain containing n_b bulges can be chased $n - 3n_b - 1$ steps from the top left to the bottom right corner, as illustrated in Figure 2 (right). If we envision H as a small submatrix of a much larger matrix of size $N \times N$, then one invocation of the bulge-chasing kernel performs $n_b(n - 3n_b - 1)$ bulge-chasing steps and the off-diagonal updates that follow update roughly nN entries of the larger matrix. A good choice for the number of bulges maximizes the ratio of the number of bulge-chasing steps to communication volume, which leads to $n_b = \lfloor n/6 \rfloor$. In other words, the chain should cover half the computational window, as is illustrated in Figure 2.

We are now ready to formally define the bulge-chasing kernel. Given an upper Hessenberg matrix $H \in \mathbb{R}^{n \times n}$ with a chain of $n_b = \lfloor n/6 \rfloor$ tightly packed bulges in its top left corner, the bulge-chasing kernel

```

1: Initialize  $U \leftarrow I_n$ 
2: for  $s = n_b$  down to 1 do
3:   for  $k = 3s - 2$  to  $n - 3(n_b - s) - 4$  do
4:     Let  $x = H_{k+1:k+3, k}$  denote the first column of the bulge
5:     Construct a 3-by-3 reflector  $Q$  such that  $Q^T x = \alpha e_1$ 
6:     Update  $H_{k+1:k+3, k:n} \leftarrow Q^T H_{k+1:k+3, k:n}$ 
7:     Update  $H_{1:k+4, k+1:k+3} \leftarrow H_{1:k+4, k+1:k+3} Q$ 
8:     Update  $U_{:, k+1:k+3} \leftarrow U_{:, k+1:k+3} Q$ 
9:   end for
10: end for

```

Algorithm 0: Sequential bulge-chasing kernel.

chases the chain to the bottom right corner. The orthogonal transformations are accumulated into an orthogonal matrix $U \in \mathbb{R}^{n \times n}$ and H is overwritten by $U^T H U$. Algorithm 0 provides additional details. The loop on line 2 iterates over the n_b bulges starting with the right-most (5 in Figure 2) bulge. The bulges are numbered from left to right starting with bulge $s = 1$. The loop on line 3 iterates over the position, k , of the first column of bulge s . A reflector Q of order 3 that reduces the first column of bulge s is constructed on line 5. H is updated from the left on line 6 and from the right on line 7, in both cases exploiting the structure of H . The transformation is finally accumulated into U on line 8.

Figure 3: Illustration of the sparsity pattern of the orthogonal transformation matrix U for the case $n = 20$ ($n_b = 3$). Gray entries are potentially non-zero, while all other entries are structurally zero. The labels $s = \dots$ indicate during which iteration of the s -loop the non-zeros are introduced.

The transformation matrix U starts out as the identity matrix (line 1) and ends up as a thickly banded matrix with additional structure within the band. Exploiting the structure of U can save up to 60% of the flops otherwise required to accumulate U . However, the details of the indexing are rather complicated and have been omitted from line 8 (and all subsequent algorithms) for brevity. More specifically, the range of rows of U to which a particular reflection must be applied is defined by maximum and minimum expressions whose terms are affine functions of the parameters (n and n_b) and the loop indices (s and k); these expressions can be derived from Figure 3, which illustrates the sparsity pattern of U and also indicates how the structure evolves through the iterations of the s -loop.

In summary, the cost of Algorithm 0 is roughly $10nn_b(n - 3n_b - 1)$ flops plus an additional 40% for the accumulation of U .

3. Parallel formulations of the bulge-chasing kernel

We now turn our attention to the development of four parallel formulations of the bulge-chasing kernel. The algorithms employ increasingly sophisticated parallel decomposition and load-balancing schemes. Our first algorithm, presented in Section 3.1, is perhaps the simplest possible and uses only two threads. One thread, $P1$, is assigned the task of processing H and the other thread, $P0$, assumes responsibility for U . Thread $P1$ receives a disproportionately large load and becomes the bottleneck. Our second algorithm, presented in Section 3.2, refines the first algorithm by adding a third thread, $P2$, to take on some of work associated with H . Unfortunately, $P1$ still receives a disproportionately large load and remains the bottleneck. In our third algorithm, presented in Section 3.3, a fourth thread, $P3$, is added to further assist the bottleneck $P1$. In this algorithm, the chain is chased in stages, which allows us to reduce the load on $P1$ using the same technique as in the second algorithm. Our fourth algorithm uses a completely different approach and relies on pipelining to expose parallelism. Thread $P0$ is still assigned sole responsibility for U and the remaining threads share the task of updating H . Each of the $p - 1$ remaining threads is assigned one block column or *panel* of H . However, the panels are not fixed and are adjusted before starting to chase each new bulge. These gradual adjustments are designed to account for the shifting load distribution between iterations/bulges. The fourth algorithm can in principle support a larger number of threads than the other algorithms. But for more than a handful of threads the updating of U will become a bottleneck and should be parallelized. Doing so would be trivial since U is always updated from the same side.

We can conceptually understand the bulge-chasing kernel as consisting of two parallel processes. The first process constructs Householder reflectors while the second process uses these reflectors to update H and U . Data dependencies make the first process inherently sequential. The second process, however, can be parallelized in many different ways. The two conceptual processes must be synchronized to ensure that the following conditions are satisfied:

1. a reflector must be constructed from up-to-date entries of H ,
2. an update must be performed after its associated reflector has been constructed, and
3. a newly generated reflector must not overwrite an existing reflector that is still in use.

The first two conditions are inherent to the problem, whereas the third condition is an artifact of the implementation and can be eliminated simply by not reusing storage for reflectors. We have designed our algorithms based on the following guidelines, which are adaptations of well-known general principles of parallel algorithm design:

Partition the columns but not the rows. The kernel inputs and outputs are stored in the column-major storage format with a large column stride (leading dimension) for H . Due to the context in which the kernel is called, we assume that the alignment of H and its column stride is arbitrary. Any partitioning of the rows of H will in general cut through cache lines and hence result in a substantial amount of false sharing. Partitioning the columns, on the other hand, would not result in the same negative effect since no pair of columns share the same cache line.

Communicate reflectors in batches. The reflector-constructing process could in principle notify the updating process as soon as a new reflector has been constructed. But doing so requires a large number of synchronizations/atomic operations and communication of matrix entries between threads. To reduce these and related overheads, one should construct reflectors in batches and notify the updating process after each batch.

Overlap the construction of reflectors with delayed updates. The updating process must wait for the construction of the first batch of reflectors. The reflector-constructing process does not, however, have to wait for all of the associated updates to complete before constructing the next batch of reflectors. To reduce the idle time, one should overlap the construction of reflectors with delayed updates.

3.1. First algorithm: Update H and U in separate threads

Algorithm 1 uses two threads and is based on the trivial observation that the accumulation of reflectors into U can proceed in parallel with the bulge-chasing in H . Only reflectors and no matrix entries need to be communicated. We assign the task of accumulating U to thread $P0$ and the task of chasing bulges and updating H to thread $P1$. Each bulge is chased completely from its initial to its final position and the associated reflectors are written into the shared buffer $Q[\text{buf}][*]$ (line 11). $P1$ goes ahead and updates H from the left (line 13) as well as from the right (line 14). When the bulge reaches its final position, $P1$ raises a flag (line 17) that notifies $P0$ of the availability of a new batch of reflectors. The notification is picked up by $P0$ (line 7), the reflector is retrieved from the shared buffer (line 12), and U is updated (line 15). When $P0$ is finished with the Q buffer, it lowers the corresponding flag (line 18). This notification is in turn picked up by $P1$ (line 6) before it proceeds to overwrite that buffer with a new batch of reflectors. The synchronization variables (flags) are initialized by $P1$ (line 1) and a barrier synchronizes the two threads upon entering (line 2) and leaving (line 21) the function.

```

1: P1: Initialize  $\text{flag}_i \leftarrow 0$  for all  $i = 1, \dots, n_{\text{buf}}$ 
2: Barrier
3: Initialize  $\text{buf} \leftarrow 1$ 
4: P0: Initialize  $U \leftarrow I_n$ 
5: for  $s = n_b$  down to 1 do
6:   P1: Wait until  $\text{flag}_{\text{buf}} = 0$ 
7:   P0: Wait until  $\text{flag}_{\text{buf}} = 1$ 
8:   for  $k = 3s - 2$  to  $n - 3(n_b - s) - 4$  do
9:     P1: Let  $x \leftarrow H_{k+1:k+3, k}$  denote the first column of the bulge
10:    P1: Construct a 3-by-3 reflector  $Q$  such that  $Q^T x = \alpha e_1$ 
11:    P1: Share  $Q$ :  $Q[\text{buf}][s] \leftarrow Q$ 
12:    P0: Receive  $Q$ :  $Q \leftarrow Q[\text{buf}][s]$ 
13:    P1:  $H_{k+1:k+3, k:n} \leftarrow Q^T H_{k+1:k+3, k:n}$ 
14:    P1:  $H_{1:k+4, k+1:k+3} \leftarrow H_{1:k+4, k+1:k+3} Q$ 
15:    P0:  $U_{:, k+1:k+3} \leftarrow U_{:, k+1:k+3} Q$ 
16:   end for
17:   P1:  $\text{flag}_{\text{buf}} \leftarrow 1$ 
18:   P0:  $\text{flag}_{\text{buf}} \leftarrow 0$ 
19:   Switch buffer:  $\text{buf} \leftarrow 1 + \text{buf} \bmod n_{\text{buf}}$ 
20: end for
21: Barrier

```

Algorithm 1: The first parallel bulge-chasing kernel.

Algorithm 1 (and later algorithms) should be interpreted as follows. The threads execute the statements independently and asynchronously with respect to the other threads. A line with the prefix “ Pk ” (in blue) should be executed only by thread Pk , a line with the prefix “! Pk ” (in blue) should be executed by all threads *except* Pk , and all lines without any prefix should be executed by all threads. Statements used for inter-thread synchronizations are shown in red.

The parameter n_{buf} controls the number of buffers and we must have $n_{\text{buf}} \geq 2$ for the algorithm to expose any parallelism at all. A larger value makes the algorithm more latency tolerant, but according to our experience there is little to gain in practice by using more than the minimum of two buffers.

3.2. Second algorithm: Parallelize one-sided updates of H

Algorithm 2 refines Algorithm 1 by adding a third thread to partially offload the bottleneck thread. This reduces the length of the critical path and potentially improves the overall performance as a result.

Line 13 of Algorithm 1 splits into the two lines 15 and 16 of Algorithm 2, which are executed by threads $P1$ and $P2$ in parallel. Figure 4 illustrates the parallel decomposition scheme. The figure shows


```

1: P1: Initialize  $\text{flags}_i \leftarrow 00_b$  for all  $i = 1, \dots, n_{\text{buf}}$ 
2: P2: Set  $\text{mask} \leftarrow 01_b$ 
3: P0: Set  $\text{mask} \leftarrow 10_b$ 
4: Barrier
5: Initialize  $\text{buf} \leftarrow 1$ 
6: P0: Initialize  $U \leftarrow I_n$ 
7: for  $s = n_b$  down to 1 do
8:   P1: Wait until  $\text{flags}_{\text{buf}} = 00_b$ 
9:   !P1: Wait until  $(\text{flags}_{\text{buf}} \text{ and } \text{mask}) \neq 0$ 
10:  for  $k = 3s - 2$  to  $n - 3(n_b - s) - 4$  do
11:    P1: Let  $x \leftarrow H_{k+1:k+3, k}$  denote the first column of the bulge
12:    P1: Construct a 3-by-3 reflector  $Q$  such that  $Q^T x = \alpha e_1$ 
13:    P1: Share  $Q$ :  $Q[\text{buf}][s] \leftarrow Q$ 
14:    !P1: Receive  $Q$ :  $Q \leftarrow Q[\text{buf}][s]$ 
15:    P1:  $H_{k+1:k+3, k:n-3(n_b-s)-1} \leftarrow Q^T H_{k+1:k+3, k:n-3(n_b-s)-1}$ 
16:    P2:  $H_{k+1:k+3, n-3(n_b-s):n} \leftarrow Q^T H_{k+1:k+3, n-3(n_b-s)-1:n}$ 
17:    P1:  $H_{1:k+4, k+1:k+3} \leftarrow H_{1:k+4, k+1:k+3} Q$ 
18:    P0:  $U_{:, k+1:k+3} \leftarrow U_{:, k+1:k+3} Q$ 
19:  end for
20:  P1:  $\text{flags}_{\text{buf}} \leftarrow 11_b$ 
21:  !P1:  $\text{flags}_{\text{buf}} \leftarrow \text{flags}_{\text{buf}} \text{ xor } \text{mask}$ 
22:  Switch buffer:  $\text{buf} \leftarrow 1 + \text{buf} \bmod n_{\text{buf}}$ 
23: end for
24: Barrier

```

Algorithm 2: The second parallel bulge-chasing kernel.

bulge $s = 2$ (red) being chased from its initial to its final position. During this process, all columns to the right of the vertical separator are updated only from the left and thus cannot affect the columns in the left block. More specifically, the gray entries in the right block are affected by updates from the left performed by $P2$. Meanwhile, thread $P1$ is responsible for updating entries in the left block. While this does improve the load balance compared to Algorithm 1, the load balance remains skewed and $P1$ is still a bottleneck. The balance gradually improves over time as the separator moves left by three columns after each completed bulge. When the last ($s = 1$) bulge is chased, the load is almost fully balanced. Further improvements therefore require additional reductions in the load assigned to $P1$.

The synchronization mechanism used in Algorithm 2 is a generalization of that in Algorithm 1. There are now two flags per buffer—one each for $P0$ and $P2$ —and as before a raised flag signals the availability of a batch of reflectors in the corresponding buffer. When a thread no longer needs the reflectors in a buffer, the corresponding flag is lowered (line 21). Thread $P1$ reuses the buffer only when both flags have been lowered, which is detected on line 20.

3.3. Third algorithm: Chase the chain in stages

Algorithm 3 is a further refinement of Algorithm 2. The underlying idea is to chase the chain in stages, and in each stage advance the chain k_{step} steps. A barrier is used to synchronize the threads between stages. Figure 5 illustrates the parallel decomposition. The task of updating H is now shared among three threads: $P1$, $P2$, and $P3$. Thread $P0$ is still solely responsible for updating U . The two vertical separators partition H into three panels. Panel $k \in \{1, 2, 3\}$, numbered left to right, is assigned to thread Pk . In each stage, the left-most separator moves left three columns after each completed bulge using the same mechanism as in Algorithm 2. The right-most separator remains fixed during a stage and moves only between stages when both separators are repositioned (and hence move right).

The updates of H from the left, which correspond to line 6 of Algorithm 0 and lines 15–16 of Algorithm 2, are further split into lines 21–23 of Algorithm 3 and executed by threads $P1$, $P2$, and $P3$ in parallel.

Figure 4: Illustration of the parallel decomposition used by Algorithm 2. When chasing the red bulge ($s = 2$) from its initial to its final position, all columns to the right of the vertical separator are updated only from the left. The two panels are assigned to different threads.

```

1: P1: Initialize  $\text{flags}_i \leftarrow 000_b$  for all  $i = 1, \dots, n_{\text{buf}}$ 
2: P2: Set  $\text{mask} \leftarrow 001_b$ 
3: P3: Set  $\text{mask} \leftarrow 010_b$ 
4: P0: Set  $\text{mask} \leftarrow 100_b$ 
5: Barrier
6: Initialize  $\text{buf} \leftarrow 1$ 
7: P0: Initialize  $U \leftarrow I_n$ 
8:  $k_{\text{max}} \leftarrow n - 3n_b - 1$ 
9:  $k_1 \leftarrow 1$ 
10: while  $k_1 \leq k_{\text{max}}$  do
11:    $\hat{k}_{\text{step}} \leftarrow \min\{k_{\text{step}}, k_{\text{max}} - k_1 + 1\}$ 
12:    $k_2 \leftarrow k_1 + \hat{k}_{\text{step}} - 1$ 
13:   for  $s = n_b$  down to 1 do
14:     P1: Wait until  $\text{flags}_{\text{buf}} = 000_b$ 
15:     !P1: Wait until  $(\text{flags}_{\text{buf}} \text{ and } \text{mask}) \neq 0$ 
16:     for  $k = k_1 + 3(s - 1)$  to  $k_2 + 3(s - 1)$  do
17:       P1: Let  $x \leftarrow H_{k+1:k+3, k}$  denote the first column of the bulge
18:       P1: Construct a 3-by-3 reflector  $Q$  such that  $Q^T x = \alpha e_1$ 
19:       P1: Share  $Q$ :  $Q[\text{buf}][s] \leftarrow Q$ 
20:       !P1: Receive  $Q$ :  $Q \leftarrow Q[\text{buf}][s]$ 
21:       P1:  $H_{k+1:k+3, k:k_2+3s} \leftarrow Q^T H_{k+1:k+3, k:k_2+3s}$ 
22:       P2:  $H_{k+1:k+3, k_2+3s+1:k_2+3n_b} \leftarrow Q^T H_{k+1:k+3, k_2+3s+1:k_2+3n_b}$ 
23:       P3:  $H_{k+1:k+3, k_2+3n_b:n} \leftarrow Q^T H_{k+1:k+3, k_2+3n_b:n}$ 
24:       P1:  $H_{1:k+4, k+1:k+3} \leftarrow H_{1:k+4, k+1:k+3} Q$ 
25:       P0:  $U_{:, k+1:k+3} \leftarrow U_{:, k+1:k+3} Q$ 
26:     end for
27:     P1:  $\text{flags}_{\text{buf}} \leftarrow 111_b$ 
28:     !P1:  $\text{flags}_{\text{buf}} \leftarrow \text{flags}_{\text{buf}} \text{ xor } \text{mask}$ 
29:     Switch buffer:  $\text{buf} \leftarrow 1 + \text{buf} \bmod n_{\text{buf}}$ 
30:   end for
31:   Barrier
32:    $k_1 \leftarrow k_2 + 1$ 
33: end while

```

Algorithm 3: The third parallel bulge-chasing kernel.

Figure 5: Illustration of the parallel decomposition used by Algorithm 3. Two vertical separators partition the matrix into three panels assigned to three different threads. The left-most separator moves left by three columns after each completed chasing of a bulge. The right-most separator moves right by k_{step} steps between stages and the left-most separator is repositioned as well.

Unlike the previous algorithms, thread $P1$ is no longer necessarily a bottleneck. A more sophisticated load-balancing scheme might therefore improve the performance slightly. However, thread $P1$ will still receive a disproportionately large total load.

3.4. Fourth algorithm: Utilize pipelining

It is difficult to incrementally improve the previous algorithms any further without violating the first guideline of Section 3, which states that we should not partition the rows of H since that would lead to substantial amounts of false sharing. Therefore, we apply a fundamentally different approach in Algorithm 4 and pipelining is used to expose parallelism. The tasks are more equally shared among the threads. In the previous algorithms, only $P1$ actually chased any bulges, but in Algorithm 4 all threads (except $P0$, which is still dedicated to U) contribute, enabling the effective use of more than four threads.

Figure 6 illustrates the parallel decomposition used by Algorithm 4 for $n = 49$, $n_b = 6$, and $p = 4$. The p separators partition the columns between the two outer-most separators into $p - 1$ panels. The panels are numbered $1, 2, \dots, p - 1$ from left to right and panel k is assigned to thread Pk . Three of the six bulges have not yet been chased (blue, top left), two bulges have already reached their final positions (blue, bottom right), and one bulge is currently being chased (red, middle). The currently chased bulge is illustrated in three of its key positions relative to the second panel.

In Figure 6, thread $P2$ assumes (from $P1$) the responsibility of advancing a bulge once it arrives at Position 1, and hands over the responsibility (to $P3$) after the bulge arrives at Position 3. When the bulge is chased across the border from Position 1 to Position 2, the updates from the right will engage some entries in both of the neighbouring panels. Thread $P2$ is responsible also for these cross-border updates. One can chase a bulge from Position 2 to Position 3 and apply the subsequent updates of the second panel without further inter-thread communication. At some point, thread $P1$ must wait for $P2$ to complete the cross-border updates in order to avoid race conditions in its own panel. To minimize the idle time caused by this synchronization point, we first chase the bulge to Position 2 and apply the cross-border updates, then notify the neighbouring thread, and eventually continue chasing to Position 3.

We are now ready to describe Algorithm 4 in more detail. The $p \geq 2$ threads are identified by their **rank** $\in \{0, 1, \dots, p - 1\}$. As always, the thread with **rank** $= 0$ is responsible for updating U , and its operations are described on lines 6–8 and 12–15. The bulges are processed in order by the s -loop starting on line 9. An iteration of this loop chases bulge s from its initial position $(3s - 2)$ to its final position $(n - 3(n_b - s) - 3)$. During this process, the bulge passes through some or all of the $p - 1$ panels and the task of chasing the bulge is passed from thread to thread. Parallelism is extracted both by performing multiple updates from the left simultaneously and by pipelining multiple bulges.

The threads are synchronized through the use of n_b integer variables **bpos** $[s]$, where $s \in \{1, 2, \dots, n_b\}$, and $p - 2$ flags **flag** $[b]$ for $b \in \{1, 2, \dots, p - 2\}$. The value of **bpos** $[s]$ tells which thread is currently responsible

```

1: if rank = 1 then
2:   Initialize  $\text{bpos}[s] \leftarrow 1$  for all  $s = 1, \dots, nb$ 
3:   Initialize  $\text{flag}[b] \leftarrow 0$  for all  $b = 1, \dots, p - 2$ 
4: end if
5: Barrier
6: if rank = 0 then
7:   Initialize  $U \leftarrow I_n$ 
8: end if
9: for  $s = n_b$  down to 1 do
10:  Balance the work by (re-)positioning the separators (see Section 3.4.2)
11:  if rank = 0 then
12:    for each panel  $b = 1, 2, \dots, p - 1$  do
13:      Wait until  $\text{bpos}[s] > b$ 
14:      Update  $U$  using the reflectors obtained when chasing bulge  $s$  through panel  $b$ 
15:    end for
16:  else
17:    if rank  $\neq p - 1$  then
18:      Wait until  $\text{flag}[\text{rank}] = 0$ 
19:    end if
20:    for each panel  $b = 1, 2, \dots, \text{rank}$  do
21:      if  $b < \text{rank}$  then
22:        Wait until  $\text{bpos}[s] > b$ 
23:        Update panel rank of  $H$  from the left using the reflectors obtained when chasing bulge  $s$  through panel  $b$ 
24:      else if  $b = \text{rank}$  then
25:        if  $b \neq 1$  then
26:          Wait until  $\text{flag}[b - 1] = 1$ 
27:          Chase bulge  $s$  across the border between panel  $b - 1$  and panel  $b$ . Limit the updates of  $H$  from the left to these panels but fully update  $H$  from the right.
28:           $\text{flag}[b - 1] \leftarrow 0$ 
29:        end if
30:        Chase bulge  $s$  within panel  $b$  either until it reaches its final position or the right-most separator of the panel. Limit the updates of  $H$  from the left to this panel. Apply a minimum of updates to  $H$  from the right.
31:         $\text{bpos}[s] \leftarrow \text{bpos}[s] + 1$ 
32:        Apply the remaining updates of  $H$  from the right that were delayed by the previous step
33:        if rank  $\neq p - 1$  then
34:           $\text{flag}[b] \leftarrow 1$ 
35:        end if
36:      end if
37:    end for
38:  end if
39: end for
40: Barrier

```

Algorithm 4: The fourth parallel bulge-chasing kernel.

Figure 6: Illustration of the parallel decomposition used by Algorithm 4 for $p = 4$ threads. The p separators partition the submatrix between the two outer-most separators into $p - 1$ panels, each of which is handled by a dedicated thread. Of the six bulges, five are stationary (blue) and one is currently being chased (red). The chased bulge is shown in three key positions relative to the second panel: Next to the outside of the left separator (top), next to the inside of the left separator (middle), and next to the inside of the right separator (bottom).

for the chasing of bulge s . Initially, $\mathbf{bpos}[s] = 1$ for all s (see Section 3.4.2 for details). Updates to the cross-border regions need to be synchronized to avoid race conditions on the column(s) immediately to the left of a separator. Recall that these columns are owned by $\mathbf{rank} = k$ but the cross-border chases and updates are performed by $\mathbf{rank} = k + 1$. This synchronization problem is solved by raising and lowering the flags in the following way. If $\mathbf{flag}[b]$ is lowered, then panel b can only be accessed by $\mathbf{rank} = b$. On the other hand, if $\mathbf{flag}[b]$ is raised, then panel b can only be accessed by $\mathbf{rank} = b + 1$, i.e., the thread to the right of the separator.

Let us now briefly explain the purpose of each line of Algorithm 4. The s -loop starting on line 9 iterates over the bulges in order. On line 18, each thread (except the owner of the right-most panel) waits until it receives control over its own panel.

The loop starting on line 20 tracks bulge s as it moves through the panels from left to right, and each thread performs the updates corresponding to its own panel. A bulge no longer affects a panel once it has crossed the panel. For this reason, the loop on line 20 ends at iteration $b = \mathbf{rank}$.

For all iterations $b < \mathbf{rank}$ of the loop on line 20, panel $\mathbf{rank}$ is only affected by updates from the left using reflectors produced by other threads. Each thread therefore waits until the bulge has passed panel b (line 22), and then applies the resulting left updates to its own panel (line 23). In the last iteration ($b = \mathbf{rank}$) of the loop, the bulge is chased through the thread's own panel.

Recall that a thread receives control over a bulge when it is positioned next to its panel's left separator. However, before it is safe to continue chasing the bulge, the thread needs to receive control over the panel to its left (line 26) in order to perform the cross-border updates. The bulge is then chased across the separator from Position 1 to Position 2 (line 27, see also Figure 6). The border region is immediately updated and the panel to the left is returned to its owner by lowering the corresponding flag (line 28). Now the bulge is chased through the thread's own block without further communication (line 30). In this step of the algorithm, only a minimum of updates are applied from the right and the rest are delayed. The availability of the new batch of reflectors is broadcast to the waiting threads by incrementing the $\mathbf{bpos}[s]$ synchronization variable (line 31). The delayed updates from the right are now applied (line 32), and control over the thread's panel is given to the thread on its right by raising the corresponding flag (line 34).

3.4.1. The underlying task graph

Algorithm 4 is effectively an encoding of a particular static schedule of an underlying task graph. The understanding of the algorithm can be enhanced by studying this implicit task graph. The tasks are defined as follows.¹ $C_b(s, b)$ denotes the task of chasing bulge s from Position 1 to Position 2 across the separator between panel $b - 1$ and panel b . This task is performed on line 27 by thread b . $C_i(s, b)$ denotes the task of chasing bulge s from Position 2 to Position 3 in panel b and is performed on line 30 by thread b . $L(s, b', b)$ denotes the task of applying to panel b the left updates obtained by chasing bulge s through panel b' and is performed on line 23 by thread b . $R_b(s, b)$ denotes the task of applying the right updates resulting from task $C_b(s, b)$ and is performed on line 27 by thread b . Finally, $R_i(s, b)$ denotes the task of applying the right updates resulting from task $C_i(s, b)$ and is performed on line 32 by thread b .

Figure 7 illustrates the tasks and dependencies associated with the first two bulges when $p = 4$. The tasks have been partitioned into three groups separated by vertical dashed lines. All tasks within a group are mapped to the same thread. The numbers shown next to the tasks indicate the order (i.e., the *schedule*) in which the tasks are executed by Algorithm 4. The black arrows indicate intra-thread dependencies, which, due to the static schedule, are implicitly satisfied. The red arrows indicate inter-thread dependencies and therefore require explicit synchronizations. The label associated with each red arrow gives the condition that must be satisfied before the dependent thread continues.

The excerpted task graph in Figure 7 explains how parallelism is extracted by Algorithm 4. The tasks mapped to any particular thread almost form a sequence, except for the possibility of executing $R_b(s, b)$ and $C_i(s, b)$ in parallel. Another observation we can make from Figure 7 is that simultaneously as thread b is executing tasks $C_b(s, b)$ and $R_b(s, b)$, thread $b - 1$ has no choice but to idle. These observations suggest that we should assign task $R_b(s, b)$ to thread $b - 1$ instead of to thread b . On the other hand, this would require an additional inter-thread synchronization point.

3.4.2. Load-balancing scheme and heuristics

The work distribution of Algorithm 4 can be configured by changing the positions chosen for the separators on line 10 (see also Figure 6). Note that the separators are repositioned for each and every bulge. The two outer-most separators have predefined positions, but the $p - 2$ interior separators can (and should) be chosen to maximize performance. In this section, we specify the constraints which restrict the placement of separators and present two load-balancing heuristics. Before we continue, a small remark on terminology: When we say that a separator is in position k we mean that it is located between columns $k - 1$ and k .

Bulge s does not affect columns strictly to the left of column $3s - 2$. We therefore position the left-most separator there. The right-most separator never moves and is always in position $n + 1$. More specifically, we introduce the pn_b load-balance parameters $\alpha_{k,s}$ for $k \in \{1, 2, \dots, p\}$ and $s \in \{1, 2, \dots, n_b\}$ as entries in a table $\mathcal{A} \in \mathbb{N}^{p \times n_b}$. We let $\alpha_{k,s}$ denote the position of the k -th separator (from left to right) during the processing of bulge s . With this notation, the prescribed positions of the outer-most separators translate into $\alpha_{1,s} = 3s - 2$ and $\alpha_{p,s} = n + 1$, for all s . The following constraints restrict the choices for the $(p - 2)n_b$ interior separators:

Minimum panel width Each panel must have room for at least one 3×3 bulge, i.e., the minimum panel width is three columns. Formally, we must have $\alpha_{k+1,s} - \alpha_{k,s} \geq 3$ for all s and $k \neq p$.

Separators move left If a separator moved right from one bulge to the next (i.e., $\alpha_{k,s} > \alpha_{k,s+1}$), then the new panel will overlap the old panel on its right and require additional synchronization points.

We therefore restrict the separators to only move left. Formally, we must have $\alpha_{k,s+1} - \alpha_{k,s} \geq 0$ for all k and $s \neq n_b$.

When balancing the load by partitioning the matrix, we need to understand how the flops are distributed across the columns. The bulge-chasing kernel has a very non-uniform distribution of flops, as illustrated in Figure 8 (right) for the case $n = 150$. This uneven distribution is in part due to the Hessenberg structure

¹We ignore the tasks associated with updating U .

Figure 7: Excerpt from the task graph underlying Algorithm 4 for $p = 4$. Shown are all tasks and dependencies associated with the first two bulges ($s = n_b$ and $s = n_b - 1$).

Figure 8: Illustration of how the flops are distributed across the columns of H when $n = 150$. Left: The flop count per column per bulge. Right: The total flop count per column, i.e., the sum of the graphs on the left.

of H and in part due to the initial and final positions being different for each bulge. For example, all bulges pass through column $n/2$ whereas no bulge passes through column n . Moreover, the first columns are relatively short and are hence affected by fewer flops during an update from the right compared to the much longer columns close to the middle. Since we can and want to adjust the separators on a per-bulge basis, we decompose the total flop count per column into its per-bulge contributions, which we illustrate in Figure 8 (left).

Figure 9: Left: The load distribution of Algorithm 4 for $n = 150$, $p = 4$, and uniform block sizes. Right: The load distribution when the load has been approximately balanced per bulge.

Our first, and simplest, load-balancing heuristic chooses the separators such that the panel widths are nearly uniform for each bulge. Formally, we define the separator positions

$$\alpha_{k+1,s} - \alpha_{k,s} \approx \frac{\alpha_{p,s} - \alpha_{1,s}}{p-1}$$

for all s and $k \in \{2, 3, \dots, p-1\}$. We refer to this strategy as the SAME SIZE heuristic. Considering Figure 8 (left), we see that the load will be poorly balanced by this strategy, as is illustrated in Figure 9 (left) for $n = 150$ and $p = 4$. We observe that threads $P2$ and $P3$ receives a much larger load than $P1$.

Our second load-balancing heuristic, which we refer to as the SAMELOAD heuristic, tries to improve performance by approximately balancing the load on a per-bulge basis. From Figure 8 (left) we can construct a model

$$f : \{1, 2, \dots, n_b\} \times \{1, 2, \dots, n\} \rightarrow \mathbb{R}$$

that maps a bulge, s , and column, x , to the number of flops, $f(s, x)$, associated with bulge s that we attribute to column x . Using the model, we find separator positions $\alpha_{k,s}$ for all s and $k \in \{2, 3, \dots, p-1\}$ such that the flops are nearly uniformly distributed across the panels. Formally, we position the separators such that

$$\sum_{x=\alpha_{k,s}}^{\alpha_{k+1,s}-1} f(s, x) \approx \frac{1}{p-1} \sum_{x=1}^n f(s, x).$$

Due to the presence of cross-border updates and the fact that we cannot distribute fractional columns, this scheme only approximately balances the load. Figure 9 (right) gives an example of the actual flop distribution across threads. A direct comparison with Figure 9 (left) shows a much more even flop distribution.

3.4.3. Optimization of the load-balance parameters

It is important to remember that attempting to balance the load, either globally or per bulge, is merely a heuristic for the actual goal of maximizing performance. One expects that a balanced load also results in good performance, but other factors, such as memory operations and synchronizations, also influence the performance. All legal settings of the load-balance parameters $\mathcal{A} = (\alpha_{k,s})$ define a large finite set of work distributions. We will next present an effective optimization method for the problem of finding the optimal configuration.

Formally, let

$$F = \left\{ \mathcal{A} \in \mathbb{N}^{p \times n_b} \left| \begin{array}{l} \alpha_{1,s} = 3s - 2 \text{ and } \alpha_{p,s} = n + 1 \\ \alpha_{k+1,s} - \alpha_{k,s} \geq 3 \text{ for all } s \text{ and } k \neq p \\ \alpha_{k,s+1} - \alpha_{k,s} \geq 0 \text{ for all } k \text{ and } s \neq n_b \end{array} \right. \right\}$$

be the set of feasible configurations. Denote by $J(\mathcal{A})$, where $\mathcal{A} \in F$, the maximum theoretically attainable performance of Algorithm 4 using configuration $\mathcal{A}$ on a particular machine. Here, we are interested in finding the configuration that maximizes the performance, that is, we want to find $\mathcal{A}^* = \arg \max_{\mathcal{A} \in F} J(\mathcal{A})$. Unfortunately, we cannot directly evaluate $J(\mathcal{A})$ due to inevitable noise in the time measurements. The performance $\hat{J}(\mathcal{A})$ of an actual experiment will therefore be less than or equal to $J(\mathcal{A})$. More specifically, we approximate $J(\mathcal{A})$ by evaluating $\hat{J}(\mathcal{A}) = J(\mathcal{A}) + \eta$, where $\eta \leq 0$ represents the noise. A straightforward way of improving the approximation of $J(\mathcal{A})$ is to select the maximum, $\hat{J}_M(\mathcal{A})$, from a set of M repeated experiments. The function $\hat{J}_M$ is a sum of a deterministic function J and a random variable η_M , where the expected value of $\eta_M \rightarrow 0$ almost surely as $M \rightarrow \infty$.

The set of feasible configurations F is typically too large for an exhaustive search to be practical and parallel search cannot be used since the objective function is tied to a particular machine. We instead propose an effective optimization method based on local search. By using the two heuristics defined in Section 3.4.2 as initial guesses, we can both expect to find a better work distribution and simultaneously estimate the quality of the heuristics.

For a feasible configuration $\mathcal{A} \in F$, we let $N(\mathcal{A}) \subset F$ denote a *neighborhood* of $\mathcal{A}$. If $\hat{J}_M(\mathcal{A}_M^*) \geq \hat{J}_M(\mathcal{A})$ for all $\mathcal{A} \in N(\mathcal{A}_M^*)$, then we say that $\mathcal{A}_M^*$ is locally M -optimal. Our local search optimization algorithm starts from an initial configuration $\mathcal{A}^{(0)} \in F$. In each step $k \geq 1$ of the algorithm, a new configuration $\mathcal{A}^{(k)} \in N(\mathcal{A}^{(k-1)})$ is chosen such that $\hat{J}_M(\mathcal{A}^{(k)})$ is maximal among all configurations in the neighborhood. If no improved configuration can be found, then configuration $\mathcal{A}^{(k-1)}$ is returned as a locally M -optimal configuration.

The definition of the neighborhood greatly impacts the speed and effectiveness of the optimization method. The neighborhood needs to be large enough to contain new configurations with sufficiently improved performance yet small enough to make an exhaustive search of the neighborhood tractable. In our case, a natural definition of a neighborhood includes all feasible configurations obtained by adding $\delta \in \{-2, -1, 1, 2\}$

to one of the $\alpha_{k,s}$ while keeping all other parameters constant. Unfortunately, such small adjustments are likely to result in very small changes in the performance. We therefore enrich the neighborhood by including a hierarchy of adjustments made to blocks of parameters. More specifically, we add (the same) δ to the β variables $\alpha_{k:k+\beta-1,s}$, where β is the block size, and let the neighborhood contain the configurations obtained for block sizes of the form $2^i \leq n_b/2$. Adjustments of this type represent shifts in the load from one thread to another and the inclusion of block adjustments makes it more likely that the neighborhood contains a sufficiently improved configuration.

4. Computational experiments

We have tested our implementations on a couple of different systems. System A has an Intel Core i7-2600 processor with a 3.4 GHz nominal clock frequency, 4 cores, 32 KB L1 (private), 256 KB L2 (private), and 8 MB L3 (shared) caches. The theoretical peak performance is 27.2 Gflop/s (DP) per core (108.8 Gflop/s total). We compiled the code using version 4.6.3 of the GCC C-compiler with the flags `-std=c99 -march=native -O3`. System B has an AMD Opteron 6238 processor with a 2.6 GHz nominal clock frequency, 12 cores (2 NUMA domains), 16 KB L1 (private), 2 MB L2 (shared by two), and 6 MB L3 (shared by six) caches. On this processor, pairs of adjacent cores dynamically share the same floating point unit. The theoretical peak performance is 10.4 Gflop/s (DP) per core assuming access only to half of the shared FPU (124.8 Gflop/s total). We compiled the code using version 4.4.3 of the GCC C-compiler with the flags `-std=c99 -march=native -O3`.

<pre> 1: Allocate memory for H and U of the given size $n \times n$ and with a given column stride for H. 2: for each sample do 3: Initialize the input matrix H (identically for each sample). 4: Copy H for later use in numerical verification. 5: Flush the master thread's cache by writing to a large and consecutive region of memory. 6: Start the timer 7: Write the bulge-chasing kernel parameters to shared memory. 8: (All) Barrier 9: (All) Read the kernel parameters from shared memory. 10: (All) Call the parallel bulge-chasing kernel. 11: (All) Barrier 12: Stop the timer 13: Record the elapsed time. 14: Numerically verify the correctness of the output. 15: end for </pre>

Algorithm 5: Method used for measuring kernel execution times

Since each kernel invocation lasts for no longer than a few milliseconds and the problems are so small that they simultaneously fit in several levels of the cache hierarchy, the details of the timing methodology become important for correctly interpreting the results. Our methodology is outlined in Algorithm 5. Each statement is performed by the master thread and only statements prefixed by “(All)” are executed by all threads in parallel. A couple of details are worth mentioning explicitly. The same memory region and the same inputs are used for each sample. The master thread flushes its cache hierarchy between samples by writing to a large and consecutive region of memory. The timings are obtained by reading the Time Stamp Counter (TSC) on x86 architectures. To obtain reliable measurements, we have followed the recommended instruction sequence described in [8]. Briefly, the timer start is implemented by a serializing `cpuid` instruction followed by an `rdtsc` instruction to read TSC, and the timer stop is implemented by the `rdtscp` instruction (note the suffix p) to read the TSC followed by a serializing `cpuid` instruction. On all tested systems, the TSC is incremented at a constant rate irrespective of the current frequencies of the cores, i.e., the measured tick count is proportional to the wall clock time.

4.1. Remarks

We remark that the two systems dynamically alter their core operating frequencies. Given that we performed the test cases back-to-back it is likely that the core frequencies were held near the maximum at all times, but we did not have the means to control nor monitor this explicitly.

Since the problems we are interested in are so small, even the slightest overhead will likely spoil the performance. In particular, we cannot afford to call any LAPACK routines or BLAS. The LAPACK routine `xLARFG` robustly constructs a Householder reflector that reduces a given vector by carefully avoiding unnecessary overflows and underflows. But since `xLARFG` resides in an external library and calls several auxiliary routines in both LAPACK and BLAS, the overhead of these calls will drastically increase the execution time. Instead, we use a much faster but less robust implementation that the compiler can inline.

The parallel algorithms presented in this paper can be trivially applied to the recently proposed optimal bulge-packing scheme [9], an alternative to the standard tight packing scheme. In the following, we present results only for the tight packing scheme.

In Algorithm 3, we can choose the number of stages and the number of steps to advance the chain in each stage. We present results only for two stages with half the bulge-chasing steps performed in each stage.

4.2. Performance evaluation and cross-over points

Figure 10: Performance comparison on System A. Algorithm 0 uses one thread, Algorithm 1 uses two threads, Algorithm 2 uses three threads, and the remaining algorithms use four threads each. Algorithm 4 is shown with four different load-balance parameter settings: SAME SIZE (label: S), SAME LOAD (label: L), optimized from SAME SIZE (label: OS), and optimized from SAME LOAD (label: OL).

We have evaluated the performance and parallel speed-up of all algorithms on the systems described above. Figure 10 (left) shows the performance in terms of billions of flops per second (Gflop/s) and Figure 10 (right) shows the corresponding parallel speed-up relative to the sequential Algorithm 0. Since a typical input to the bulge-chasing kernel is a submatrix of a much larger matrix, we have set the column stride of H to 10000 in all our experiments. All results were obtained using 1000 samples for each data point and the graphs show the maximum attained performance. As is common in comparing different algorithms, the performance numbers are based on the theoretical number of flops. Hence, a performance ratio corresponds exactly to an execution time ratio. We can observe that the parallel algorithms perform reliably better than the sequential algorithm on System A for $40 \leq n \leq 200$. Figure 11 (left) shows the analogous speed-up curves on System B. Here, the speed-ups are slightly smaller. Moreover, they do not increase much for $n \geq 100$ and for Algorithms 1 and 2 they even decrease slightly.

Figure 11 (right) illustrates the scalability of Algorithm 4 for $p = 2, 3, \dots, 6$. Note that there is virtually no improvement from $p = 5$ to $p = 6$, which we attribute (by examining traces) to P_0 (the thread updating U) becoming a bottleneck.

Figure 11: Performance comparison on System B. Left: Speed-up relative to Algorithm 0. Algorithm 4 is shown with two different load-balance parameter settings: SAMESIZE (label: S) and SAMELOAD (label: L). Right: Performance of Algorithm 4 with SAMELOAD for $p \in \{2, 3, 4, 5, 6\}$. Examination of traces show that the thread responsible for accumulating U is the bottleneck for $p \geq 6$.

4.3. Optimization and the effectiveness of the heuristics

Figure 12: Illustration of the load-balance parameters $\alpha_{k,s}$ chosen by the SAMESIZE heuristic (dotted), the SAMELOAD heuristic (dashed), and optimized with the SAMELOAD heuristic as the initial guess (solid).

The SAMESIZE and SAMELOAD heuristics are oblivious of the target system, whereas the optimization method presented in Section 3.4.3 tunes the parameters to match the system. Figure 12 compares the parameter settings of the two heuristics with the optimized parameters on System A for $n = 100$ and $p = 4$. The values of the parameters $\alpha_{k,*}$ for the SAMESIZE heuristic are illustrated using four dotted curves. Since this heuristic chooses uniform panel widths for each bulge, these curves are in fact straight lines. The parameters chosen by the SAMELOAD heuristic are shown using dashed curves. (Only the free parameters, that is $\alpha_{2,*}$ and $\alpha_{3,*}$, are shown.) The optimized parameters, with the SAMELOAD heuristic as the initial guess, are shown using solid curves. Figure 12 (right) shows the corresponding work distribution. We observe that the optimizer has reduced the load on $P1$ and increased it on $P3$ in the second half of the computation (low-numbered bulges).

We ran the optimization method on System A with $M = 100$ for all $n \in \{10, 20, \dots, 200\}$ using both

heuristics as initial guesses. The resulting performances and speed-ups are shown in Figure 10. All Algorithm 4 variants show similar performance for small matrices ($n \leq 50$). The SAMELOAD heuristic gradually outperforms the SAMESIZE heuristic as the size of the matrix increases. Running the optimizer with either the SAMESIZE heuristic or the SAMELOAD heuristic as the initial guesses yields different placements of the separators $\alpha_{k,s}$ but with similar performances. (Starting from the SAMELOAD heuristic results in a similar, but slightly better, performance.) For $n \geq 100$, the average improvement is approximately 5% and 15% compared to the SAMELOAD and SAMESIZE heuristics, respectively.

Figure 13: Actual traces of Algorithm 4 on System A with $n = 100$ and four threads for (a) the SAMESIZE heuristic, (b) the SAMELOAD heuristic, and (c) optimized from the SAMELOAD heuristic.

To get an even better understanding of the differences between the heuristics and the optimized settings we show three actual traces in Figure 13. The traces have the same scale in the sense that one centimeter in the horizontal direction corresponds to the same amount of time in all traces. We observe a fair amount of idle time in all traces, although slightly less in the optimized trace. In the optimized trace, there is one thread (the second row) that is making progress most of the time, which is not the case for the other traces.

5. Conclusion

We presented four fine-grained parallel algorithms for the bulge-chasing kernel appearing in the small-bulge multi-shift QR algorithm for the non-symmetric dense eigenvalue problem. It was necessary to eliminate library function calls and use inexpensive atomic operations instead of standard synchronization mechanisms such as locks and conditional variables.

Our experiments suggest that the parallel algorithm based on pipelining of the bulges in conjunction with a highly configurable moving block column partitioning of the Hessenberg matrix is superior. We proposed two heuristics for balancing the load. We further developed an effective optimization method based on local search that adapts the load-balance parameters to a particular system. Additionally, the experiments suggest that the SAMELOAD heuristic is near-optimal in the sense that the optimized parameter settings from two different initial guesses yield comparable performances. Still, the performance obtained by using the optimized parameter settings is 5% better on average.

Our parallel kernel can be used to reduce the execution time of the critical path of the QR algorithm and thereby improve its strong scalability. Alternatively, the increased performance can be traded for an increase in the size of the computational window and thereby indirectly improve the performance by reducing the

amount of memory traffic in other parts of the QR algorithm. Similar trade-offs involving memory traffic, computational kernel performance, and the length of the critical path have been previously explored in the context of other linear algebra operations, see for example [10, 11].

Acknowledgments

Financial support has been provided by the Swedish Research Council (grant VR A0581501), UMIT Research Lab via Balticgruppen, and eSSENCE, a strategic collaborative eScience programme funded by the Swedish Research Council. This research was conducted using the resources of High Performance Computing Center North (HPC2N).

References

- [1] K. Braman, R. Byers, R. Mathias, The multishift QR algorithm. I. Maintaining well-focused shifts and level 3 performance, *SIAM J. Matrix Anal. Appl.* 23 (2002) 929–947.
- [2] K. Braman, R. Byers, R. Mathias, The multishift QR algorithm. II. Aggressive early deflation, *SIAM J. Matrix Anal. Appl.* 23 (2002) 948–973.
- [3] R. Granat, B. Kågström, D. Kressner, A novel parallel QR algorithm for hybrid distributed memory HPC systems, *SIAM J. Sci. Comput.* 32 (2010) 2345–2378.
- [4] B. Kågström, D. Kressner, M. Shao, On aggressive early deflation in parallel variants of the QR algorithm, in: *PARA 2010, Applied Parallel and Scientific Computing*, volume 7134 of *LNCS*, Springer, 2012, pp. 1–10.
- [5] R. Granat, B. Kågström, D. Kressner, M. Shao, Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation, Technical Report UMINF-12.06, Dept. of Computing Science, Umeå University, Sweden, 2012.
- [6] G. H. Golub, C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, third edition, 1996.
- [7] D. S. Watkins, The transmission of shifts and shift blurring in the QR algorithm, *Linear Algebra and its Applications* 241/243 (1996) 877–896.
- [8] G. Paoloni, How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures, Technical Report, Intel Corporation, 2010.
- [9] L. Karlsson, D. Kressner, B. Lang, Optimally packed chains of bulges in multishift QR algorithms, *AMC TOMS* (2013 (in press)). Preprint available as LAPACK Working Note #271.
- [10] E. Agullo, J. Dongarra, R. Nath, S. Tomov, A fully empirical autotuned dense QR factorization for multicore architectures, in: *Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, Springer Berlin / Heidelberg, Bordeaux, France, 2011, pp. 194–205.
- [11] H. Bouwmeester, M. Jacquelin, J. Langou, Y. Robert, Tiled QR factorization algorithms, in: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, ACM, 2011, pp. 7:1–7:11.