Parallel Reduction of a Block Hessenberg-Triangular Matrix Pair to Hessenberg-Triangular Form—Algorithm Design and Performance Results

Björn Adlerborn

Department of Computing Science and HPC2N
Umeå University
SE–901 87 Umeå, Sweden.
bjorn.adlerborn@softcenter.it

Report UMNAD xxx.04

November, 2004

Abstract

The design, implementation and performance of a parallel algorithm for reduction of a matrix pair in block upper Hessenberg-Triangular form (H_r, T) to upper Hessenberg-triangular form (H, T) is presented. This reduction is the second stage in a two-stage reduction of a regular matrix pair (A, B) to upper Hessenberg-Triangular from. The desired upper Hessenberg-triangular form is computed using two-sided Givens rotations. The parallel implementation is analyzed with regard to scalability properties and the selection of near to optimal algorithm parameters. Performance results for the ScaLAPACK-style implementation show that the parallel algorithm can be used to solve large scale problems effectively.

Contents

1 Introduction

"To speed things up" has been a goal for scientists and engineers for as long as we know. Producing faster cars, more food in less time etc. The same goes for computer science where the computing power of computers is roughly doubled from one year to another. However, we are still not able to solve all types of problems efficiently and effectively, that is solving them accurately and in reasonable time. Nevertheless, with the computer system evolution we can solve more and more complex problems.

Matrix computations are fundamental in most computational science and engineering problems. Examples include forecast modeling, earthquake simulators, real-time airline scheduling. In order to build application software for different types of applications, there is a great demand on library software for solving different types of linear algebra problems such as large linear systems of equations $Ax = b$, or different eigenvalue problems, the standard problem $Ax = \lambda x$, as well as the generalized eigenvalue problem $Ax = \lambda Bx$.

The importance of fast and accurate software for matrix operations is evident and explains why so much time is spent on research on this subject, both at universities and in the private sector. The research group on Parallel and Scientific Computing at Umeå University focuses on algorithms and applications for scalable high-performance computer systems. One of the ongoing research projects is design of algorithms and library software for various matrix computations. This Master Theses is an initiative from that group and deals with solving one special type of matrix problems namely computing eigenvalues and eigenvectors for a regular matrix pair (A, B). More specifically, this corresponds to solving the generalized eigenvalue problem $Ax = \lambda Bx$, where A and B are square real $n \times n$ matrices. If B is equal to the identity matrix, the problem reduces to the standard eigenvalue problem. If B is singular, the matrix pair (A, B) will have infinite eigenvalues besides possible finite eigenvalues.

To achieve this the matrix pair (A, B) is reduced to the so called generalized Schur form (S, T), where T is upper triangular and S is upper quasi-triangular, i.e., it may have 1×1 and 2×2 diagonal blocks corresponding to real and complex conjugate pairs of eigenvalues, respectively. The process of transforming (A, B) to (S, T) is performed in several stages. First, (A, B) is transformed to a block-Hessenberg-triangular form (H_r, T), where H_r is an upper block-Hessenberg matrix with r subdiagonals and T is upper triangular. The second stage reduces (H_r, T) to upper Hessenberg-triangular form (H, T), where now H is an upper Hessenberg matrix with one subdiagonal and T is upper triangular as before. The final third stage is to transform (H, T) to generalized Schur form (S, T), which is typically done by the QZ-algorithm. All stages are performed in terms of equivalence transformations on the matrix pair (A, B), i.e., orthogonal transformation matrices Q and Z are applied such that $(A, B) \leftarrow Q^T(A, B)Z = (Q^TAZ, Q^TBZ)$ for each of the three stages.

Much time has already been spent on this particular subject and the Umeå group has developed serial algorithms and implementations for the full reduction. In addition, a parallel implementation for the first stage of the three stages

has been implemented but no algorithm nor implementation existed for the second stage of this reduction. My thesis work was to find such an algorithm and develop software supporting concurrent processors.

The parallel algorithm and the results described here have been published in a number of articles by the Umeå group including [1, 2, 3]. The work on to this master thesis was mainly conducted during Winter 2000 and Spring 2001 but has not been properly documented until now.

Next, in Section 2, a brief introduction to parallel computing in general is presented. Section 3 gives a brief overview of the first two stages leading to the Hessenberg-triangular form and some previous work. In Sections 4 and 5, the assignment description and a discussion of its realization is presented. The results of the Master Thesis are presented in Section 6, including descriptions of data distribution, the parallel algorithm for reduction to Hessenberg-triangular form, and performance results on an IBM SP parallel computer system. Finally, in Section 7, some conclusions of the work are presented.

2 Some Background to Parallel Computing

2.1 Doing it Faster

Computers connected together in a network can be used to solve large scale computational problems faster. Some problems are so large that they can only be solved using several processors concurrently. Typically, the problem is split into smaller subproblems and each computer handles only a subproblem of the original much larger or more complex problem. Usually this will take less time than using only a single computer. Solving problems on more than one computer/cpu often involves some kind of communication between the computers (nodes), for example distributing the subproblems and reporting status and results of computations. If the communication gets heavy it will have impact on the time it takes to solve the problem and thus this way of solving problem is not always doable due to the different nature of problems, Some problems are compute-intensive others are communication-intensive, and many are a combination of the two extremes.

2.2 Parallel Computers

The types of computer used when solving problems in parallel, that is using many CPUs concurrently, can be divided into two types: shared memory machines (SMM) and distributed memory machines (DMM). The differences lies in how the memory system is organized and how memory is used by the different CPUs. For SMM machines all CPUs share all available global memory, but which physically can be organized in different ways. In the DMM-case, each CPUs has its own memory which also is physically on the node. With these types of parallel machines different types of problems arises. For example, wfor a SMM system, the CPUs have to compete over the available resources.

One problem is how to ensure that only one CPU is using a part of the memory at any given time. This type of problem is often solved by using shared semaphores. One big advantage of using SMM is that communication is quite inexpensive, all communication is performed in memory, compared to communication over for example a regular TCP/IP based network or even a high-speed interconnect. The DMM does not have the memory synchronization problem as on SMM. However, since the CPUs are using their own memory, communication is required whenever more data is needed or results are to be shared across the CPUs. Even though networks are becoming faster and faster they can never compete with communication that is performed in memory. Despite the communication drawbacks on DMMs they are much more used than SMMs due to manufacturing costs. A simple and low cost DMM can easily be setup using a regular TCP/IP network and a couple of Linux machines connected to each other through a hub/switch. On the other hand a single computer handling a problem using threads can be seen as SMM. Today, we see an evolution of hybrid system with state-of-the-art processors of SMM type connected with a high-speed network, i.e., a DMM system on the global level.

2.3 Communication Fundamentals

As stated earlier communication is often required when solving problems in parallel and the communication has impact on the total execution time. There are two parameters to consider when designing a communication algorithm. The first is how many messages we need to send, the other is how big the messages are. The total time for communication can defined as follows:

$$\alpha \times NumMessages + \beta \times SumSizeMessages,$$

where α denotes the startup cost, called *node latency*, and β defines the per word cost (1 / β is the bandwidth). α and β varies between systems, but in general it is better to communicate larger chunks of data compared to communicating smaller packets but several times, i.e., typically the latency is dominating the communication overhead.

2.4 Memory Issues and Blocking

In order to write efficient algorithms, the memory architecture must also be considered. In most advanced computers systems, the memory is hierarchical were we in the bottom have the off-processor and shared memory and at the top the top level caches and registers. Typically, the amount of memory in top of the hierarchy is much less than the amount in the bottom. Since all types of calculations are performed at the top and the amount of memory there is limited, we need to move data around to be able to perform all needed calculations. In order to gain maximum performance, we need to maximize the data reuse, or data locality, which in turn minimizes the data movement within the memory hierarchy. There is though often a tradeoff between maximizing

the data reuse and maximizing the concurrency. Due to this some algorithms that works excellent on a single CPU do not perform well at all in an parallel environment.

A technique to increase data locality, or data reuse, is to reorganize the algorithm so it uses matrix-matrix operations instead of matrix-vector or vector-vector operations in the inner loops. This technique is called blocking and it has proven to be very successful, for example, in the development of the high-performance library LAPACK (Linear Algebra PACKage) [4]. LAPACK is a generic software library consisting of routines for linear algerba computations on SMM or single processor machines. LAPACK is based on another package called BLAS (Basic linear Algebra Subprograms) which in turn consists of optimized implementations of common linear algebra computations such as dot-products (level 1), matrix-vector operations (level 2) and matrix-matrix operations (level 3), e.g, matrix multiplications.

A subset of LAPACK exists also for DMM environments. The corresponding package is called ScaLAPACK [**?**] and is based on the packages PBLAS and BLACS. PBLAS consists of parallel level 1–3 BLAS routines, that is basic linear algebra operations combined with message passing. The message passing is performed by routines in BLACS (Basic Linear Algebra Communication Subprograms), which for example includes synchronous send/receive routines to communicate a matrix or submatrix from one process to another. BLACS is typically implemented on top of the MPI standard for message passing. The node computations are mainly done but calling LAPACK routines. ScaLAPACK is a scalable parallel library in the sense that it can effectively solve larger problems on several processors. Originally, ScaLAPACK was designed to give high efficiency on concurrent DMMs like the IBM SP series and the Cray T3 series.

3 Previous Work: The Two-Stage Algorithm

In the following section, a description of previous work, closely connected to this Master Thesis, by the Umeå group including the design of a blocked (LAPACK-style) two-stage variant of `DGGHRD` and a ScaLAPACK-style implementation of the first stage of the reduction [10, 9, 6, 8, 7].

The reduction a matrix pair (A, T) to (H, T) form is divided in to two separate stages.

Stage one reduces the matrix pair to a block upper Hessenberg-triangular form (H_r, T) form using Householder reflections and the compact WY representation [14] of the Householder matrices. The matrix H_r is upper r-Hessenberg with $r > 1$ subdiagonals and T is upper triangular. In the second stage of the reduction algorithm, all but one of the subdiagonals of the block Hessenberg A-part are set to zero while keeping T upper triangular. The annihilation of elements along the subdiagonals of H_r and fill-in elements in T are performed using Givens rotations.

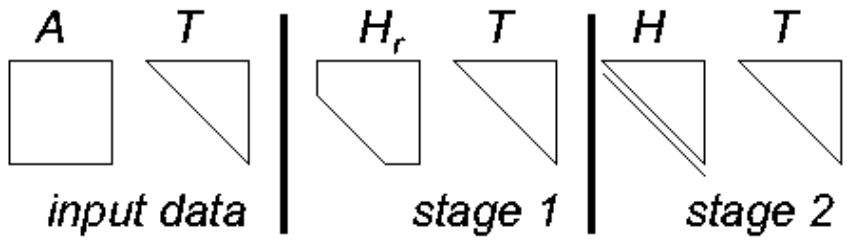

Figure 1: The two stages in the Hessenberg-triangular reduction algorithm.

3.1 Stage 1: Parallel Blocked Reduction to (H_r, T) Form

The parallel implementation of the first stage is presented and analyzed in [9, 6]. This includes a scalability analysis based on an hierarchical performance model [8] and real experiments. Here, we only review the parallel implementation of the blocked (H_r, T) reduction algorithm outlined in Figure 2, which is based on existing parallel operations in the ScaLAPACK library [5].

On entry to the `HrT` algorithm, $B \in R^{n \times n}$ is assumed to be in upper triangular form. If this is not the case, the ScaLAPACK routines `PDGEQRF` and `PDLARFB` are used to perform a QR factorization of B and to update the matrix A accordingly. On exit, A is upper $r-$Hessenberg, B is upper triangular, and Q, Z are the accumulated orthogonal transformation matrices such that $A = Q^H AZ$ and $B = Q^H BZ$.

The A-matrix is reduced by QR factorizations of rectangular $pr \times r$ blocks ($p \geq 2$) and B is restored by RQ factorizations of square $pr \times pr$ blocks, using the ScaLAPACK routines `PDGEQR2` and `PDGERQ2`, respectively. All updates are performed using a combination of the ScaLAPACK routines `PDLARFT` for extraction of the triangular matrix T and `PDLARFB` for application of the Householder transformations represented in compact WY form.

Since the fill-in overlap for consecutive iterations, we apply the RQ factorizations to blocks of size $(p-1)r \times pr$ in all iterations except the last one in each block column.

3.2 Stage 2: Unblocked and Blocked Reduction to Hessenberg-Triangular Form

The second stage is to annihilate the remaining $r-1$ subdiagonals of H_r to get an upper Hessenberg matrix H, while keeping the B-part upper triangular. This problem was solved by implementing a parallel variant of the blocked algorithm described in [10].

The operations to be performed for the (H_r, T) to (H, T) reduction are summarized in Figure 3. On entry to `HrT2HT`, $A, B \in \mathcal{R}^{n \times n}$, A is upper $r-$Hessenberg, and B is upper triangular. On exit, A is upper Hessenberg

```
function [A, B, Q, Z] = HrT (A, B, r, p)
  k = n/r;                    # blocks in the first block column.
  for j = 1:r:n-r
    k = max(k - 1, 2);        # blocks to reduce in current block column j.
    l = ceil((k-1)/(p-1));    # steps required for the reduction.
    i = n;
    for step = 1:l
      nb = min(p*r, i-j-r+1);
      Phase 1: Annihilation of p r × r blocks in block column j of A.
      [q, A(i-nb+1:i,j:j+r-1)] = qr(A(i-nb+1:i,j:j+r-1));
      A(i-nb+1:i,j+r:n) = q'*A(i-nb+1:i,j+r:n);
      B(i-nb+1:i,i-nb+1:n) = q'*B(i-nb+1:i,i-nb+1:n);
      Q(:,i-nb+1:i) = Q(:,i-nb+1:i)*q; Q = I_n initially.
      Phase 2: Restore B – annihilation of fill-in.
      [z, B(i-nb+1:i,i-nb+1:i)] = rq(B(i-nb+1:i,i-nb+1:i));
      A(1:n,i-nb+1:i) = A(1:n,i-nb+1:i)*z;
      B(1:i-nb,i-nb+1:i) = B(1:i-nb,i-nb+1:i)*z;
      Z(:,i-nb+1:i) = Z(:,i-nb+1:i)*z; Z = I_n initially.
      i = i - nb + r;         Pointer for next block annihilation.
    end
  end
```

Figure 2: Matlab-style algorithm for the blocked (H_r, T) reduction of (A, B) (Stage 1).

```
function [A,B] = HrT2HT(A, B, r)
  [m,n]=size(A)
  for k = 1:n-2
    for l = min(k+r-1, n-1):-1:k+1
      [c,s]=givens(A(l:l+1,k))
      A(l:l+1, k:n) = row_rot(A(l:l+1, k:n),c,s)
      for i = l:r:n-1
        B(i:i+1,i:n) = row_rot(B(i:i+1,i:n),c,s)
        [c,s]=givens(B(i+1,i:i+1))
        B(1:i+1, i:i+1) = col_rot(B(1:i+1, i:i+1),c,s)
        m = min(i+r+1,n)
        A(1:m,i:i+1) = col_rot(A(1:m,i:i+1),c,s)
        if (i+r+1 <= n)
          [c,s]=givens(A(i+r:i+r+1,i))
          A(i+r:i+r+1, i:n) = row_rot(A(i+r:i+r+1, i:n),c,s)
        end
      end
    end
```

Figure 3: Unblocked reduction to upper Hessenberg-triangular form (Stage 2).

and B is upper triangular.

Algorithm `HrT2HT` annihilates $A_{l+1,k}$ using a Givens rotation and applies the rotation to rows $l, l+1$ of A. To preserve the eigenvalues of (A, B) the rotation is applied to rows $l, l+1$ of B as well. This application introduces a non-zero element $B_{l+1,l}$. We zero this fill-in by a column rotation (applied from right), which in turn, when applied to A, introduces a new non-zero element $A_{l+r+1,l}$. The i-loop chases the unwanted non-zero elements down the $(r+1)$-th subdiagonal of A and the subdiagonal of B. To complete the reduction of column k this procedure is repeated $r-1$ times. Similar operations are applied to the remaining columns $(k+1, \ldots, n-2)$ to produce the desired (H, T) form.

In the blocked variant of the (H_r, T) to (H, T) reduction, the data locality in each sweep (one iteration of the k loop in `HrT2HT`) is improved as follows [10]:

1. All $r-1$ subdiagonal elements in column k are reduced before the chasing of unwanted non-zero fill-in elements starts.

2. A super-sweep that reduces m columns of A per iteration in the outer k loop is introduced.

3. The updates of A and B are restricted to r consecutive columns at a time. We store all rotations (in vectors) to enable delayed updates with respect to Givens rotations generated for previous updated columns in the current super-sweep.

These three items are important for reducing the data traffic in a single node memory hierarchy. However, as we will see, when moving to a (distributed) parallel environment the delayed updates (see item three above) cause most of the processors to be idle and only a few working at the same time. So, instead of delaying updates all rotation vectors are broadcasted row wise (or column wise) in the processor mesh after a complete reduction of a column in A (or row in B) has been made. Thereby, we enable other processors to participate in the computations as soon as possible.

4 Assignment Description

The purpose of this master thesis was to develop, implement, test and evaluate a parallel reduction algorithm for the second stage in the two-stage reduction of (A, T) to (H, T). The existing blocked implementation of the second stage was to be used as input. A natural part of the assignment was to get an understanding of the overall algorithm and specifically the different reduction techniques. Moreover studies of BLAS, PBLAS, LAPACK, BLACKS and ScaLAPACK was requried before I could start with the main work.

Output was expected to be a well tested and documented and an efficient and functional parallel implementation following the ScaLAPACK standard.

5 Realization

At first, an algorithm had to be developed. The raw algorithm was developed by using the existing blocked version of the second stage. One approach would have been to completely rewrite the second stage and do it parallel, another to take the existing code and parallelize it, line by line. The latter was chosen because the code was well organized and commented so it was quite easy to start.

The first month was spent reading about and testing existing LAPACK and ScaLAPACK routines. I had to understand what the reduction was all about, and get familiar with the LAPACK and ScaLAPACK routines to get it right from the beginning. I also produced an algorithm written in pseudo code which was discussed with my instructors before I began programming. The next 3 months I spent working on the code, and since all previous work was written in Fortran 77, I continued on that track. I started out with a version that had all required communication to share the problem across many processors. This version could not run on more than one CPU but gave me an opportunity to validate the code for computation. After that I added support for the grid configurations $1 \times P$, $P \times 1$ and $P \times P$.

When it comes to problems I did not, as I first feared, have any with the language or the environment, but when it came to debugging I ran into difficulties. The hardest thing was to debug the communication. I used small problems and low level tracing to file to figure out where things went wrong. Quiet a lot of

time was spent on this part. The first functional version was really inefficient and did not run well on practically any grid configuration but 1×1. We sat down and looked at the code and identified a couple of bottle-necks and changed to communication routines a bit. Instead of using the original approach where two or more processors shares data that is needed for a single computation and all data is sent to one processor that performs the calculation and sends the result back we send the required data to all involved processors and let them all do the calculation. As soon as one processor is ready with its computation it can go on with its next task and does not have to wait for the result from other processors. These changes gave the desired efficiency. Our goal was to achieve a speedup of at least $\sqrt{P}$ where P denotes the number of processors.

The remaining time was spent on testing and evaluation of the routine. Since this was the second stage out of two I had to test both routines at the same time to find the optimal parameters for a given problem size. As we will see in the Result section the second stage takes more time to complete than the first so the parameters which are the best for the second stage are the parameters which is best for complete reduction.

Testing was performed using Matlab where I compared the resulting matrices with what the equivalent routines gave in Matlab. I also computed the eigenvalues and compared with the correct ones, computed by an existing serial routine in LAPACK (dggev). In both tests I found that the difference was smaller than the machine precision (1E-16).

The evaluation was performed using different problem sizes and varying processor grids. I used up to 64 processors in a 8×8 grid and a problem size up to 8192×8192.

6 Results

6.1 Data Distribution

This parallel implementation of the two-stage reduction follows the ScaLAPACK software conventions [5]. The P processors (or virtual processes) are viewed as a rectangular processor grid $P_r \times P_c$, with $P_r \geq 1$ processor rows and $P_c \geq 1$ processor columns such that $P = P_r \cdot P_c$. The data layout of dense matrices on a rectangular grid is assumed to be done by the two-dimensional block-cyclic distribution scheme. The block size used in the parallel algorithms is $NB = r$, where r is the number of subdiagonals in the block Hessenberg-triangular reduction (Stage 1). This enables good load balancing by splitting the work reasonably evenly among the processors throughout the algorithms. Moreover, the block-cyclic data layout enables the use of blocked operations and the level 3 BLAS for high-performance computations on single processors.

6.2 Parallel Reduction to Hessenberg-Triangular Form

In this section a description of the algorithm of the parallel blocked variant of `HrT2HT` is given.

When reducing the k-th column of A, the sub matrix pair $(A_{:,k+1:n}, B_{:,k+1:n})$ is partitioned in $s = \lceil (n-k)/r \rceil$ column blocks of size $n \times r$ (the last one of size $n \times \mathrm{mod}((n-k), r)$ when r is not a factor of $n-k$). Each block column pair i is further divided into square $r \times r$ upper triangular blocks denoted $A_i^{(t)}$, $B_i^{(t)}$, and rectangular blocks denoted $A_i^{(r)}$ and $B_i^{(r)}$:

$$\begin{bmatrix} A_i^{(r)} \\ A_i^{(t)} \\ 0 \end{bmatrix}, \qquad \begin{bmatrix} B_i^{(r)} \\ B_i^{(t)} \\ 0 \end{bmatrix}, \tag{1}$$

where $A_i^{(r)}$ is of size $(i \cdot r + k) \times r$, and $B_i^{(r)}$ is $((i-1) \cdot r + k) \times r$. Notice that the zero-block is not present in block column $s-1$ of $A_{:,k+1:n}$ and in block column s of $B_{:,k+1:n}$. Moreover, the last block column of $A_{:,k+1:n}$ has neither a zero nor a triangular block, i.e., it consists of $A_s^{(r)}$ only. Also remark that $A_{s-1}^{(t)}$ is upper trapezoidal when it has fewer than r rows.

This block partitioning is illustrated in Figure 4 when the first column ($k = 1$) of a matrix pair of size 12×12 is reduced, where A has $r = 4$ sub diagonals. The blocks labeled 5 and 9 are $A_1^{(t)}$ and $A_2^{(t)}$, and the blocks labeled 4, 8, and 12 correspond to $A_1^{(r)}$, $A_2^{(r)}$ and $A_3^{(r)}$, respectively. Similarly, the diagonal blocks 2, 6 and 10 are $B_1^{(t)}$, $B_2^{(t)}$, and $B_3^{(t)}$, and finally, the blocks 3, 7, and 11 are $B_1^{(r)}$, $B_2^{(r)}$, and $B_3^{(r)}$, respectively.

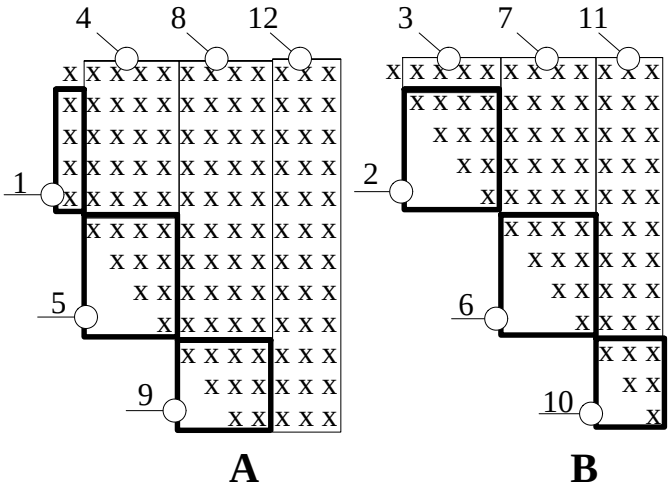

Figure 4: Block partitioning and reference pattern in blocked (H_r, T) to (H, T) reduction (Stage 2).

row_i is defined as the set of all row eliminations (rotations) required to annihilate $r-1$ elements of A, and similarly col_i is the set of all column rotations needed to annihilate $r-1$ elements of B. The row_1-set reduces $r-1$ sub diagonal elements of the k-th column of A, while row_i for $i \geq 2$ annihilates fill-in introduced in the sub diagonal of $A_{i-1}^{(t)}$. The sets col_i zero fill-in introduced in the sub diagonals of $B_i^{(t)}$. By $row_{1:i}$ and $col_{1:i}$ we denote the row and column sets 1 to i, respectively. Notice that when $A_i^{(t)}$ and $B_i^{(t)}$ have $r' < r$ rows, we annihilate $r'-1$ elements only and the associated rotation sets contain $r'-1$ rotations.

The labeling of the blocks in Figure 4 follows the blocked spiral reference pattern of a matrix pair (A, B) in the blocked implementation [10]. The access pattern of the blocks are different in our parallel implementation, which is obvious from the algorithm description below. The label sets within brackets show which blocks in Figure 4 that are referenced in the operations for each block iteration i.

A *sweep* reducing column k of A proceeds as follows:
Reduce:
The set row_1 is generated ([1]) and broadcasted along the current processor row. The set is then applied to $B_1^{(t)}$ ([2]), $B_{2:s}^{(r)}$ ([7, 11]), $A_{1:s}^{(r)}$ ([4, 8, 12]). By generating col_1 the resulting fill-in is annihilated in $B_1^{(t)}$ ([2]). col_1 is broadcasted along current processor column and $B_1^{(r)}$ is updated with respect to col_1 ([3]).

Chase:

for $i = 1 : s$
  Apply col_i to $A_i^{(r)}$. ([4], [8], [12])
  if $i < s$
    Apply col_i to $A_i^{(t)}$. ([5], [9])
    Zero fill-in of $A_i^{(t)}$, i.e., generate row_{i+1}. ([5], [9])
    Broadcast row_{i+1} along the current processor row.
    Apply row_{i+1} to $A_{i+1:s}^{(r)}$. ([8, 12], [12])
    Apply row_{i+1} to $B_{i+1}^{(t)}$, $B_{i+2:s}^{(r)}$. ([6, 11], [10])
    Zero fill-in of $B_{i+1}^{(t)}$, i.e., generate col_{i+1}. ([6], [10])
    Broadcast col_{i+1} along the current processor column.
    Apply col_{i+1} to $B_{i+1}^{(r)}$. ([7], [11])
end

In case of empty row or column rotation sets, no action is taken in the updates.

As in the blocked implementation [10], the above described procedure is extended to allow m columns to be reduced and chased in each sweep (called *super-sweep*). To distinguish *row*-sets belonging to different reduced columns of A, we use a superscript $j = 1, \ldots, m$. For example, row_1^1, row_1^2, ..., row_1^m

denote the first *row*-set of each of the m columns reduced in a super-sweep. Column sets belonging to a super-sweep are denoted analogously.

In the reduce-part of a super-sweep, the sets $row^i_{1:m-i+1}$ and $col^i_{1:m-i}$ ($i = 1, \ldots, m$) are generated. The chase-part of the super-sweep iteratively advances the sweeps one block column ahead in a pipelined fashion, starting with the leading block.

To find the optimal value of m several parameters must be considered, including the number of non-zero sub diagonals r ($= NB$), the matrix size N, processor grid configuration and the memory hierarchy of the processors.

Typically, the annihilation of elements and the resulting fill-in require cooperation between four processors since the current window (a virtual block of size $NB \times NB$) most of the time is spread among different processors. No more than four processors can share a virtual block since the cyclic distribution is done with a blocking factor of NB. The sharing of blocks means that boarder elements have to be exchanged for the application of *row*- and *col*-sets and the calculation of new *row*- and *col*-sets for reduction of the fill-in since the update/reduction always operates on two consecutive rows or columns.

In Figure 5, we illustrate how a row rotation in B and reduction of the resulting fill-in is done on a $NB \times NB$ block ($NB = 6$), which is shared by four processors. The procedure for a column rotation is similar. A rowwise broadcast is performed before application of the row rotations begins, that is, before step 1, and during steps 5–7 a columnwise broadcast is performed. This is done to ensure that all processors that need rotation values have them before the associated updates start.

6.3 Performance Results

The measured performance results of `AT2HrT` and `HrT2HT` are obtained by using up to 64 Thin Nodes (P2SC, 120 MHz) on the IBM Scalable POWER Parallel System at High Performance Center North (HPC2N).

P is varied between 2 and 64 in multiples of 2 and for a fixed P we investigate different grid configurations. For each processor grid we vary the block size NB to find a near to optimal value, which provides the best performance of the parallel algorithm. These best NB values together with the performance measured in Million floating point operations per second (*Mflops/s*) and the scaled (constant memory usage) speedup (S_P) are listed in columns 4–6 in Table 1 for the parallel Stage 1 reduction. S_P is computed as the ratio between the performance (measured in Mflops/s) obtained on P processors and one processor, respectively. The S_P values shown are rounded to one decimal's accuracy.

Similar results for the parallel Stage 2 reduction are listed in columns 7–9 in Table 1. The results for the complete parallel two-stage Hessenberg-triangular reduction are displayed in columns 4–6 of Table 2. In Stage 1, the number of blocks annihilated in each blocked QR factorization is chosen as $p = \max(2, P_r)$. In Stage 2, the number of columns in a supersweep is kept fixed ($m = 2$). We

Configuration			Stage 1			Stage 2		
N	P_r	P_c	NB	*Mflops/s*	S_P	NB	*Mflops/s*	S_P
1024	1	1	160	276	1.0	64	89	1.0
1448	2	1	150	479	1.7	150	184	2.1
1448	1	2	150	446	1.6	200	163	1.8
2048	2	2	160	734	2.7	180	244	2.7
2048	4	1	170	734	2.7	170	236	2.7
2048	1	4	160	473	1.7	180	40	0.4
2816	2	4	180	910	3.3	180	321	3.6
2816	4	2	140	1189	4.3	180	380	4.3
4096	4	4	180	1791	6.5	200	573	6.4
4096	8	2	170	1588	5.8	200	546	6.1
4096	2	8	200	1076	3.9	200	394	4.4
5792	16	2	180	1775	6.4	200	613	6.9
5792	2	16	200	1186	4.3	200	453	5.1
5792	8	4	180	2573	9.3	180	752	8.4
5792	4	8	180	2041	7.4	200	707	7.9
8192	8	8	170	3581	13.0	200	989	11.1
8192	16	4	150	3146	11.4	200	922	10.4
8192	4	16	180	2095	7.6	200	716	8.0

Table 1: Performance results for Stages 1 and 2 on 1, 2, 4, 8, 16, 32, and 64 IBM SP Thin Nodes (120 MHz).

Configuration			Stage 1+2			Ratios	
N	P_r	P_c	NB	*Mflops/s*	S_P	F	T
1024	1	1	64	153	1.0	0.7	1.9
1448	2	1	150	292	1.9	0.7	1.9
1448	1	2	170	253	1.7	0.8	2.1
2048	2	2	180	414	2.7	0.7	2.1
2048	4	1	170	407	2.7	0.7	2.2
2048	1	4	150	90	0.6	0.7	8.1
2816	2	4	180	550	3.6	0.7	1.9
2816	4	2	180	673	4.4	0.7	2.1
4096	4	4	200	1034	6.8	0.6	2.0
4096	8	2	170	968	6.3	0.6	1.8
4096	2	8	200	679	4.4	0.6	1.7
5792	16	2	180	1092	7.1	0.6	1.6
5792	2	16	200	781	5.1	0.6	1.6
5792	8	4	180	1430	9.3	0.6	2.1
5792	4	8	200	1260	8.2	0.6	1.8
8192	8	8	200	1919	12.5	0.6	2.3
8192	16	4	170	1736	11.3	0.6	2.1
8192	4	16	180	1269	8.3	0.6	1.8

Table 2: Performance results for the complete two-stage reduction on 1, 2, 4, 8, 16, 32, and 64 IBM SP Thin Nodes (120 MHz).

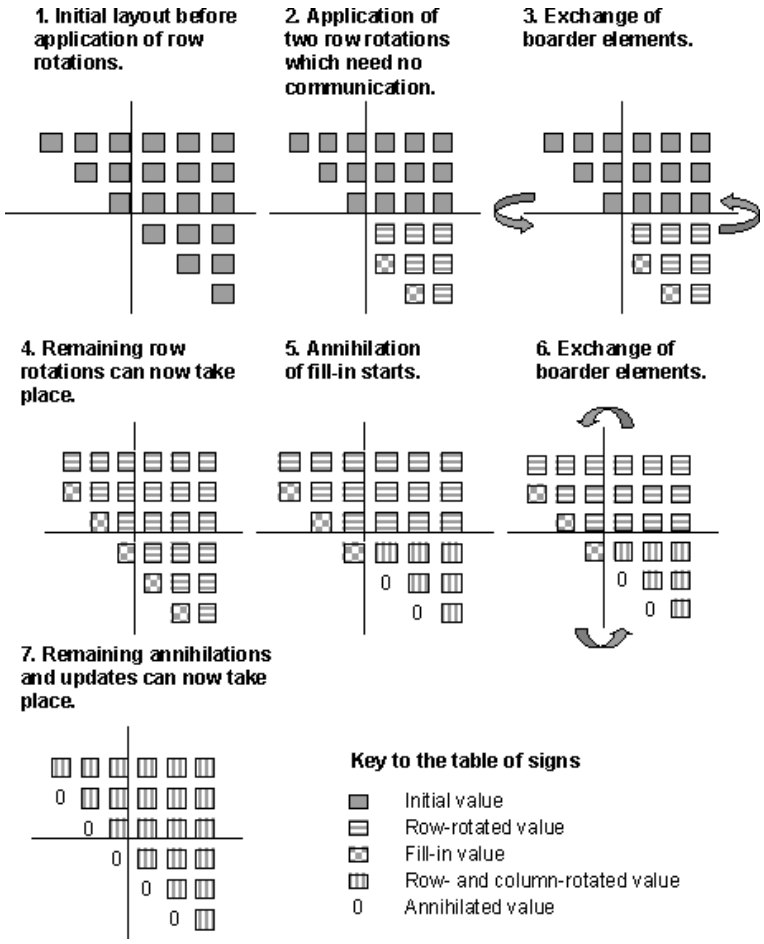

Figure 5: Application of row rotation and annihilation of fill-in across processor boarders (Stage 2).

have performed testing with larger values of m, but without observing any substantial performance improvements. Some results are shown in Table 3.

The last two columns in Table 2 show the ratios between the number of floating point instructions (*flops*) in Stage 2 and Stage 1 (F) and their corresponding execution time ratios (T). The number of flops in the two stages are determined by N, NB and p (p is only used in Stage 1).

Configuration				Stage 1+2		
N	P_r	P_c	m	NB	*Mflops/s*	S_P
2816	4	2	1	180	656	4.3
2816	4	2	2	180	673	4.4
2816	4	2	3	180	679	4.4
2816	4	2	4	180	661	4.4
2816	4	2	5	180	671	4.4

Table 3: Performance results using different values of m in Stage 2 on 8 IBM SP Thin Nodes (120 MHz).

7 Conclusions

7.1 Selection of Algorithm and Machine Parameters

Although Stage 2 involves much less flops ($0.6 \leq F \leq 0.8$), the best execution time for Stage 2 is roughly twice as long as for the corresponding Stage 1 reduction. This can be explained by the implicit nature of the Stage 2 parallel reduction. It has a more fine-grained and costly communication compared to Stage 1, which is ruled by the data dependencies of Stage 2. Moreover, mainly level 3 operations are performed in Stage 1, while there are lower level operations in Stage 2 (mostly level 1–2 and some level 2.5, that is a mix between level 2 and 3).

The processor grid configuration affects the data distribution and thereby the communication overhead and the execution rate of the parallel algorithms. The results in Table 1 show that choosing $P_r = P_c$ (when possible) gives the best performance. Otherwise, the best results are obtained for $P_r > P_c$, with $P_c > 1$ as large as possible. For a given configuration (N, $P_r \times P_c$), the block size NB giving the best performance of the combined Stage 1+2 algorithm is, as we expected, in between the block sizes for the parallel Stage 1 and Stage 2 algorithms. Typically, Stage 2 and Stage 1+2 have the same "best" NB values, which also show the impact of Stage 2 to the overall performance of the two-stage algorithm.

7.2 Scalability Analysis

As we have seen in Table 1 the second stage is a scalable routine. The speedup is roughly $\sqrt{P}$ which is what can be expected. The speedup increases with the problem size N and number of processors, P.

All in all this tells us that we can solve large problems and the computational experiments have shown that we can solve them both efficiently and accurately.

7.3 Limitations

Since support for the accumulation of Q and Z was added quite late not much time was spent testing this part. Otherwise there are no known limits as of today.

7.4 Improvements and Future Work

Future work includes the design of a performance model of the parallel two-stage algorithm that can be used for automatic selection of algorithm-architecture parameters, e.g., block and grid sizes.

8 Acknowledgements

In different and many ways my supervisors Bo Kågström and Krister Dackland at Umeå University have contributed to this Master Thesis, both in the algorithm developing and the final writing. I would also like to thank the High Performance Computing Center North (HPC2N) for access to the IBM SP system Knut and the HPC2N staff for their excellent user support.

References

[1] B. Adlerborn, K. Dackland, and B. Kågström. Parallel Two-Stage Reduction of a Regular Matrix Pair to Hessenberg-Triangular Form And Generalized Schur Forms, In J. Fagerholm et.al. (eds.), *Applied Parallel Computing. Advanced scientific computing.* Springer-Verlag, Lecture Notes in Computer Science, pp 319–328, 2002.

[2] B. Adlerborn, K. Dackland, and B. Kågström. A Parallel Two-Stage Algorithm for Reduction of a Regular Matrix Pair to Hessenberg-Triangular Form, Report UMINF-02.04, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, 2002.

[3] B. Adlerborn, K. Dackland, and B. Kågström. Parallel Two-Stage Reduction of a Regular Matrix Pair to Hessenberg-Triangular Form. In T. Sørevik et al (eds), *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*. Springer-Verlag, Lecture Notes in Computer Science. Vol. 1947, pp 92–102, 2001.

[4] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen. *LAPACK Users' Guide*, Third Edition. SIAM Publications, Philadelphia, 1999.

[5] S. Blackford, J. Choi, A. Clearly, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petit, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide.* SIAM Publications, Philadelphia, 1997.

[6] K. Dackland. Parallel Reduction of a Regular Matrix Pair to Block Hessenberg-Triangular Form - Algorithm Design and Performance Modeling. Report UMINF-98.09, Department of Computing Science, Umeå University, S-901 87 Umeå, 1998.

[7] K. Dackland and B. Kågström. Reduction of a Regular Matrix Pair (A, B) to Block Hessenberg-Triangular Form. In Dongarra et.al., editors, *Applied Parallel Computing: Computations in Physics, Chemistry and Engineering Science, PARA95*, Lecture Notes in Computer Science, Springer, Vol. 1041, pages 125–133, 1995.

[8] K. Dackland and B. Kågström. An Hierarchical Approach for Performance Analysis of ScaLAPACK-based Routines Using the Distributed Linear Algebra Machine. In Wasniewski et.al., editors, *Applied Parallel Computing in Industrial Computation and Optimization, PARA96*, Lecture Notes in Computer Science, Springer, Vol. 1184, pages 187–195, 1996.

[9] K. Dackland and B. Kågström. A ScaLAPACK-Style Algorithm for Reducing a Regular Matrix Pair to Block Hessenberg-Triangular Form. In Kågström et.al., editors, *Applied Parallel Computing: Large Scale Scientific and Industrial Problems, PARA98*, Lecture Notes in Computer Science, Springer, Vol. 1541, pages 95–103, 1998.

[10] K. Dackland and B. Kågström. Blocked Algorithms and Software for Reduction of a Regular Matrix Pair to Generalized Schur Form. *ACM Trans. Math. Software*, Vol. 25, No. 4, 425–454, 1999.

[11] W. Enright and S. Serbin. A Note on the Efficient Solution of Matrix Pencil Systems. *BIT* 18, 276–281, 1978.

[12] G. H. Golub and C. F. Van Loan. *Matrix Computations*, Second Edition. The John Hopkins University Press, Baltimore, Maryland, 1989.

[13] C. B. Moler and G. W. Stewart. An Algorithm for Generalized Matrix Eigenvalue Problems. *SIAM J. Num. Anal.*, 10:241–256, 1973.

[14] R. Schreiber and C. Van Loan. A Storage Efficient WY Representation for Products of Householder Transformations. *SIAM J. Sci. and Stat. Comp.*, 10:53-57, 1989.