Evaluation of the Tunability of a New NUMA-Aware Hessenberg Reduction Algorithm*

by

Mahmoud Eljammaly, Lars Karlsson, and Bo Kågström

UMINF 16.21

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

*This version revised March 2017.

Evaluation of the Tunability of a New NUMA-Aware Hessenberg Reduction Algorithm*

Mahmoud Eljammaly
mjammaly@cs.umu.se

Lars Karlsson
larsk@cs.umu.se

Bo Kågström
bokg@cs.umu.se

Abstract

The reduction of a general dense and square matrix to Hessenberg form is a well known first step in many standard eigenvalue solvers. Although parallel algorithms exist, the Hessenberg reduction is still one of the bottlenecks in state-of-the-art software for the distributed QR algorithm. We propose a new NUMA-aware algorithm that fits the context of the QR algorithm and evaluate the tunability of its algorithmic parameters. The proposed algorithm can be faster than LAPACK and ScaLAPACK for small problem sizes. In addition, evaluating the algorithmic parameters shows that there is potential for auto-tuning some of the parameters.

Keywords: Hessenberg reduction, parallel cache assignment, NUMA-aware algorithm, shared-memory algorithm, tunable parameters, off-line tuning.

1 Introduction

The work presented is motivated by a bottleneck in a distributed parallel multi-shift QR algorithm for solving large-scale dense matrix eigenvalue problems [7]. On the critical path of the QR algorithm, we find an expensive procedure known as Aggressive Early Deflation (AED) [1]. The purpose of AED is two-fold. First, to detect and deflate converged eigenvalues. Second, to generate shifts for subsequent QR iterations. The AED procedure operates on a square sub-matrix located at the bottom right corner of the matrix. There are three main steps in AED. First, recursively apply the QR algorithm. Second, reorder the eigenvalues to detect converged eigenvalues. Third, to restore the sub-matrix to Hessenberg form. Future work will investigate the first two steps since the full potential of the present work is realized only when all three steps are parallelized in a similarly aggressive manner.

What distinguishes Hessenberg reduction in the particular context of AED compared to the general case? A key feature is that the problem size is relatively small. Since the computation sits on the critical path of the overall computation its performance is critical. The number of cores available is too large relative to the problem size for ordinary algorithms and implementations to perform well.

The work presented here uses only cores contained in one shared-memory node of a larger distributed system. This enables efficient use of fine-grained parallelization and auto-tuning

*NLAFET Working Note 8. Report UMINF 21.16, Dept. Computing Science, Umeå University, SE 901 87 Umeå, Sweden. Results were presented at the SIAM Conference on Parallel Processing for Scientific Computing, Paris, 2016.

techniques. The aim is to present a new implementation of multi-threaded Hessenberg reduction with a large degree of flexibility and locality of memory references. This means that the implementation has tunable parameters that can be tweaked to tune its performance. The goal is to distinguish those parameters that have an impact on the performance and need tuning from those that either have no impact or are trivial to set optimally.

A node in a distributed memory system commonly has a shared-memory architecture with Non-Uniform Memory Access (NUMA). Hessenberg reduction, as will be explained later, is a partially memory-bound computation with approximately 20% of the arithmetic operations accounted for by matrix–vector multiplications. High performance is obtained only when the cost of memory accesses is minimized. Therefore, our implementation makes use of the Parallel Cache Assignment (PCA) technique recently proposed by Castaldo and Whaley [2, 3, 4, 8] for one-sided factorizations and unblocked Hessenberg reduction. This technique leads to two benefits. First, the implementation becomes NUMA-aware in the sense that it leads to mostly local memory references. Second, the implementation can efficiently utilize the aggregate capacity of a parallel cache hierarchy if the problem is small enough to fit in cache.

The rest of the paper is organized as follows. Section 2 surveys published algorithms for Hessenberg reduction and recalls the PCA technique. Section 3 describes our multi-threaded implementation of blocked Hessenberg reduction using the PCA technique. Section 4 identifies algorithmic parameters. Section 5 summarizes experimental results showing measured performance and the impact of individual tuning of each parameter. Section 6 presents a simple off-line tuning mechanism and shows its impact on the new implementation. Section 7 concludes and highlights future work.

2 Background

2.1 Unblocked Hessenberg Reduction

Hessenberg reduction is an orthogonal similarity transformation that transforms a given square matrix A to an upper Hessenberg matrix $H = Q^T AQ$, where Q is an orthogonal matrix. The textbook algorithm [6] for Hessenberg reduction uses Householder reflections to zero out—*reduce*—the columns one by one from left to right. Specifically, let $A^{(0)} = A$ denote the initial state of the input. The first column of $A^{(0)}$ is reduced by a similarity transformation to obtain $A^{(1)}$. The second column of $A^{(1)}$ is reduced to obtain $A^{(2)}$. And so on for a total of $n-2$ iterations. The output $H = A^{(n-2)}$ is in upper Hessenberg form. In practice, the computation is organized in such a way that the sequence of matrices overwrite the initial matrix.

Consider an arbitrary iteration transforming $A^{(k-1)}$ into $A^{(k)}$. The step consists of constructing a Householder reflection $Q^{(k)} = I - \tau^{(k)} v^{(k)} {v^{(k)}}^T$ and applying it to $A^{(k-1)}$ by the similarity transformation $A^{(k)} \leftarrow {Q^{(k)}}^T A^{(k-1)} Q^{(k)}$. The reflection is constructed from the kth column such that the kth column of $A^{(k)}$ has only zeros below the first sub-diagonal. After $n-2$ iterations, the matrix A has been transformed into the matrix H by a similarity transformation of the form $H \leftarrow Q^T AQ$, where $Q = Q^{(1)} Q^{(2)} \cdots Q^{(n-2)}$.

2.2 Blocked Hessenberg Reduction

The textbook Hessenberg reduction algorithm described above involves mainly memory-bound matrix–vector multiplications hidden inside the structured application of a Householder reflection to both sides of A. With the advent of CPU caches, it was soon realized that a cache-blocked variant of the textbook algorithm that expresses most of the arithmetic in terms of compute-bound matrix–matrix multiplications would perform much better on machines with memory hierarchies [5]. This first cache-blocked variant expresses approximately 70% of the arithmetic in terms of matrix–matrix multiplications. An improvement of the algorithm was later developed that increased this ratio to 80% [11].

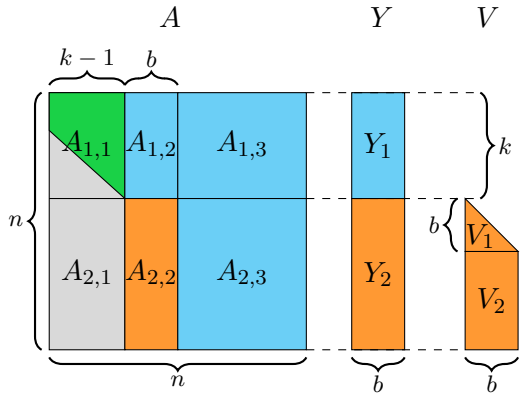

Figure 1: Partitioning of matrices A, Y, and V after reducing the first $k-1$ columns. Here, b is the panel width.

The blocked Hessenberg reduction algorithm (Algorithm 1) revolves around block iterations, each of which reduces a bunch of adjacent columns called a *panel*. After having reduced the first $k-1$ columns, the matrix A is partitioned as in Figure 1.

The purpose of a block iteration is to reduce the panel $A_{2,2}$ to upper triangular form by constructing and applying an orthogonal similarity transformation of the form

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

where $I - VTV^T$ is a compact WY representation [12] of the b Householder reflections that collectively reduce the panel. The algorithm is complicated by the fact that the reflections cannot be constructed without also partially applying them. In practice, therefore, the transformation is applied in the form

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

where the intermediate matrix Y is defined as

$$Y = AVT. \tag{2}$$

Both V and Y are partitioned alongside A as illustrated in Figure 1.

Each block iteration consists of two distinct phases. In the first phase, the panel $A_{2,2}$ is reduced and fully updated. This gives rise to a set of Householder reflections, which are accumulated into a compact WY representation defined by V and T. The first phase also

Algorithm 1: Sequential blocked Hessenberg reduction [11]

```
   // Outer loop over panels
1  for k ← 1 : b : n − 2 do
       // Determine panel width
2      b̂ ← min{b, n − k − 1};
       // Initialize intermediate matrices
3      V ← 0_{n−k×0};
4      T ← 0_{0×0};
5      Y ← 0_{n×0};
       // Phase 1:  Inner loop
6      for j ← 1 : b̂ do
7          Partition A, V, and Y as in Figure 1 with b = j − 1;
           // Update column j of A22 from the right
8          A_{2,2}(:, j) ← A_{2,2}(:, j) − Y_2 V_2(1, :)^T;
           // Update column j of A22 from the left
9          A_{2,2}(:, j) ← A_{2,2}(:, j) − V T^T V^T A_{2,2}(:, j);
10         Construct a Householder reflection (v_j, τ_j) that reduces column j of A_{2,2};
           // Augment Y
11         Y ← [ Y_1                                  0
                 Y_2   τ_j A_{2,2:3}(:, j + 1 : n) v_j − Y_2 V^T v_j ];
           // Augment T
12         T ← [ T    −τ T V^T v_j
                 0          τ_j    ];
           // Augment V
13         V ← [ V    v_j ];
       // Phase 2:  Delayed updates
14     Partition A, V, and Y as in Figure 1 with b = b̂;
       // Update A_{2,3} from the right
15     A_{2,3} ← A_{2,3} − Y_2 V_2^T;
       // Update A_{2,3} from the left
16     A_{2,3} ← A_{2,3} − V T^T V^T A_{2,3};
       // Compute the top block of Y
17     Y_1 ← A_{1,2:3}(:, 2 : n) V T;
       // Update A_{1,2:3} from the right
18     A_{1,2:3}(:, 2 : n) ← A_{1,2:3}(:, 2 : n) − Y_1 V^T;
```

generates the lower block of Y (see Figure 1). The incremental construction of Y and T requires matrix–vector multiplications involving the blocks $A_{2,2}$ and $A_{2,3}$, but only $A_{2,2}$ is modified in the process. This is the most expensive part and which makes the first phase memory-bound.

In the second phase, the upper block of Y is computed from (2), and $A_{1,2}$, $A_{1,3}$ and $A_{2,3}$, are updated according to (1). The computations are in the form of matrix–matrix multiplications, which make the second phase compute-bound. We refer to the original sources [5, 11] for details omitted from this brief description.

Two stage algorithms for the Hessenberg reduction also exist [9]. These algorithms require large problems, a very unbalanced computer, and they are not NUMA-aware. Hence, they are not suitable to use in the context of AED.

2.3 PCA: Parallel Cache Assignment

Multicore processors and, more generally, shared-memory systems based on multicore processors, have parallel cache hierarchies. That is, several disjoint caches on the same level of the hierarchy, each one associated with a distinct subset of the cores. In such systems, the aggregate cache capacity—especially at higher cache levels—might be able to store all the data necessary to solve a problem of a significant size. Castaldo and Whaley recently proposed a technique called Parallel Cache Assignment (PCA) and applied it in a series of articles to the panel factorizations of one-sided factorizations [2, 4] as well as to the unblocked Hessenberg reduction algorithm [3]. They argued that PCA is able to—for sufficiently small problems at least—turn an otherwise memory-bound computation into a cache-bound or possibly even compute-bound computation by more effectively using the parallel caches to transform the majority of memory accesses into cache hits.

The main idea that defines PCA is to imagine sibling caches as though they are local memories in a distributed memory system and assign to each core a subset of the data. An interleaved distribution of the data over the NUMA nodes will reduce the bandwidth congestion. The main benefit is the effective use of the caches when the problem fits in cache.

There are two parts to PCA: one mandatory and one optional. The mandatory part of PCA is to partition the data and assign each part to a different thread and then distribute the work in accordance with the owner-computes rule. This results in natural locality of reference since each thread will repeatedly operate on its own subset of the data. The optional part is to explicitly copy the distributed data assigned to a thread into memory allocated local to that thread. This results in better utilization of caches and memory buses as well as fewer false sharing incidents. An implementation that fulfill both parts of PCA becomes NUMA-aware and can therefore benefit even for problems that do not fit in the cache.

How does PCA relate to LAPACK and ScaLAPACK? The difficulty of enforcing a data distribution between calls to the BLAS implies that the LAPACK library cannot fulfill even the mandatory part of PCA. The ScaLAPACK library, on the other hand, naturally enforces data distribution and the owner-computes rule and thus fulfills both parts of PCA. One point of view is to interpret PCA as an attempt to get some benefits of the ScaLAPACK approach without giving up the benefits of a multi-threaded approach.

3 Applying PCA to blocked Hessenberg reduction

Algorithm 1 consists of two nested loops (outer and inner loops). The inner loop implements the first phase while the remainder of the outer loop implements the second phase.

The parallelizations of the two phases of each outer loop iteration of Algorithm 1 are described separately in Sections 3.1 and 3.2. The first phase is memory-bound and the aim is to apply PCA to optimize the memory accesses. The second phase is compute-bound and the aim is to balance the load and avoid synchronization and communication.

3.1 Parallelization of the first phase

Since the first phase is memory-bound, the aim is to optimize the data locality using the PCA technique. The data and work will be distributed such that each thread primarily works on data that it owns. Partition A, V, and Y as in Figure 1. The first phase consists of five major steps for each column $\mathbf{a} = A_{2,2}(:, j)$ of the panel:

1. update $\mathbf{a}$ from the right (line 8),
2. update $\mathbf{a}$ from the left (line 9),
3. reduce $\mathbf{a}$ (line 10),
4. augment Y (line 11), and
5. augment T (line 12).

These steps depend on each other and must therefore be performed in order.

The optimal parallelization strategy depend on the context. Two strategies are considered: *full parallelization* and *partial parallelization*. In the full parallelization strategy, all steps that involve at least one matrix dimension larger than $\hat{b} = min\{b, n - k - 1\}$ are parallelized. This strategy exposes more parallelism at the cost of more overhead. Therefore, it may be appropriate for large problems. To avoid this extra synchronization overhead, the partial strategy only parallelizes the most expensive computational step, namely the matrix–vector multiplication in Step 4.

Below, we elaborate on each step.

Step 1. The column is updated from the right by the operation

$$\mathbf{a} \leftarrow \mathbf{a} - Y_2 V_2^T. \tag{3}$$

This is a tall-and-skinny matrix–vector multiplication and is parallelized in the full strategy by partitioning Y_2 into row blocks as illustrated in Figure 2.

Step 2. The column is updated from the left by the sequence of operations

$$\mathbf{w} \leftarrow V^T \mathbf{a}, \tag{4}$$

$$\mathbf{w} \leftarrow T^T \mathbf{w}, \tag{5}$$

$$\mathbf{a} \leftarrow \mathbf{a} - V\mathbf{w}. \tag{6}$$

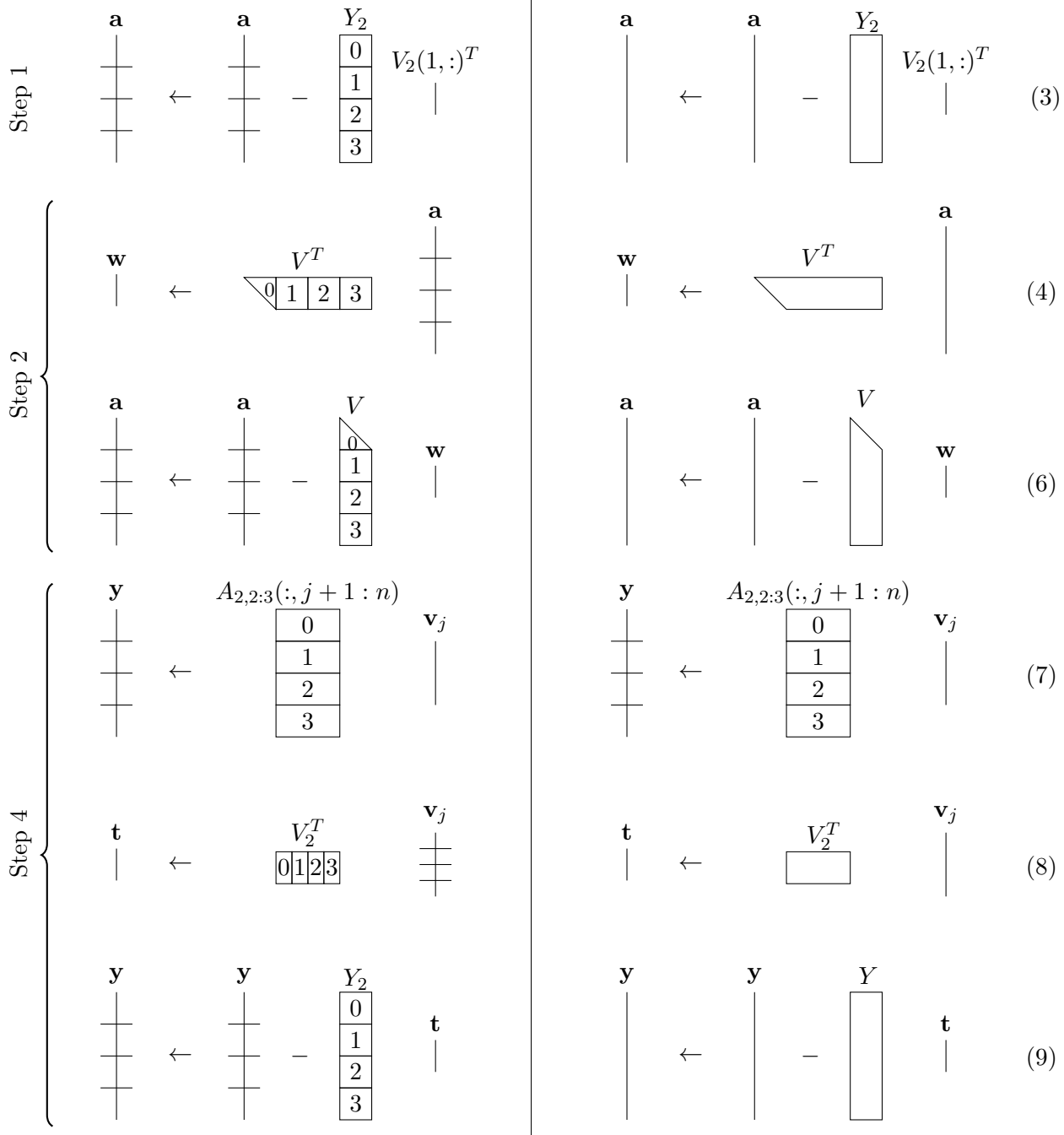

Figure 2: The shapes of matrices and vectors in the most expensive operations of the first phase of an outer loop iteration of Algorithm 1 for both the full and the partial parallelization strategies. Numbers inside the matrices show the distribution of work over threads. Numbers in the right-hand side margin refer to equations.

This is a short-and-fat matrix–vector multiplication, followed by a small triangular matrix–vector multiplication, and finally a tall-and-skinny matrix–vector multiplication. In the full parallelization strategy, both $\mathbf{a}$ and V are partitioned into row blocks, see Figure 2. The computation of (5) is sequential.

Step 3. A Householder reflector $(\mathbf{v}_j, \tau_j)$ is constructed to reduce $\mathbf{a}$ below, but not including, element a_j. This involves taking the norm of and scaling a vector; both done sequentially.

Step 4. The matrix Y is augmented with another column through the operations

$$\mathbf{y} \leftarrow A_{2,2:3}(:, j+1:n)\mathbf{v}_j, \tag{7}$$

$$\mathbf{t} \leftarrow V_2^T \mathbf{v}_j, \tag{8}$$

$$\mathbf{y} \leftarrow \mathbf{y} - Y_2\mathbf{t}, \tag{9}$$

$$\mathbf{y} \leftarrow \tau_j \mathbf{y}, \tag{10}$$

$$Y \leftarrow \begin{bmatrix} Y_1 & 0 \\ Y_2 & \mathbf{y} \end{bmatrix}. \tag{11}$$

This step, in particular the matrix–vector multiplication in (7), typically accounts for the bulk of the memory references and arithmetic operations and is therefore a key to an effective parallelization strategy. In both parallelization strategies, the computation of (7) is parallelized by partitioning $\mathbf{y}$ and $A_{2,2:3}(:, j+1:n)$ into row blocks. This is the only operation parallelized in the partial strategy. The remaining operations are a short-and-fat matrix–vector multiplication, a tall-and-skinny matrix–vector multiplication, and finally a vector scaling. In the full parallelization strategy, the vectors $\mathbf{v}_j$ and $\mathbf{y}$ as well as the matrices V and Y_2 are partitioned into row blocks. See Figure 2 for illustrations.

Step 5. The matrix T is augmented with a new row and column through the operations

$$\mathbf{t} \leftarrow -\tau T\mathbf{t}, \tag{12}$$

$$T \leftarrow \begin{bmatrix} T & \mathbf{t} \\ 0 & \tau \end{bmatrix}. \tag{13}$$

This is a small triangular matrix–vector multiplication and is done sequentially. Note that $\mathbf{t}$ was computed in (8) during Step 4.

Application of parallel cache assignment. The bulk of the data accesses in the first phase are due to the block of A involved in (7), which makes that operation a prime target for the PCA technique. Prior to entering the first phase, the block $A_{2,2:3}(:, 2:n)$ is partitioned into block rows and each block is assigned to a thread. This is the mandatory part of PCA. Then, optionally, we copy the part into memory local to that thread. Inside the first phase and in each iteration, each thread computes its part of $\mathbf{y}$ involved in (7) using a local buffer, then the global $\mathbf{y}$ is updated.

3.2 Parallelization of the second phase

Consider an arbitrary iteration of the outer loop and partition the matrices A, V, and Y as in Figure 1. The second phase consists of four major steps:

1. update $A_{2,3}$ from the right (line 15),
2. update $A_{2,3}$ from the left (line 16),
3. compute Y_1 (line 17), and
4. update $A_{1,2:3}$ from the right (line 18).

The aim of this section is to describe an effective parallelization strategy for these steps. A good strategy has few synchronization points, good load balance, cache locality, and computationally efficient tasks. Figure 3 details the partitioning scheme of the matrices.

Step 1. The update of $A_{2,3}$ from the right is achieved by the matrix outer product

$$A_{2,3} \leftarrow A_{2,3} - Y_2 V_2^T. \tag{14}$$

Any data decomposition of $A_{2,3}$ leads to independent tasks and each element costs the same to update. In particular, one-dimensional block row/column decomposition or two-dimensional block decomposition are suitable candidates. As will be explained later, the block column distribution, as illustrated in Figure 3, is preferred.

Step 2. The update of $A_{2,3}$ from the left is achieved by the following sequence of operations:

$$W \leftarrow V^T A_{2,3}, \tag{15}$$
$$W \leftarrow T^T W, \tag{16}$$
$$A_{2,3} \leftarrow A_{2,3} - VW. \tag{17}$$

In (15), distributing W row-wise leads to a maximum of $\hat{b}$ tasks and each thread needs to access the entire $A_{2,3}$ block. On the other hand, distributing W column-wise leads in general to many more tasks and each thread only needs to access a part of $A_{2,3}$. Hence, a block column distribution of W, and consequently also of $A_{2,3}$, is preferred (see Figure 3). This choice implies that a matching distribution of $A_{2,3}$ in Step 1 should be used to avoid some of the synchronization between Steps 1 and 2. Applying the same distribution of W also to (16) and of $A_{2,3}$ also to (17) likewise avoids some synchronization. These distributions are illustrated in Figure 3.

Step 3. Computing the top block Y_1 of Y is achieved by the following sequence of operations:

$$Y_1 \leftarrow A_{1,2:3}(:, 2:n)V, \tag{18}$$
$$Y_1 \leftarrow Y_1 T. \tag{19}$$

The shape of Y_1 is tall and skinny, which favors a block row distribution as illustrated in Figure 3. This means that each thread only needs to access a part of $A_{1,2:3}(:, 2:n)$, while a block column distribution, on the other hand, would require each thread to access the entire block.

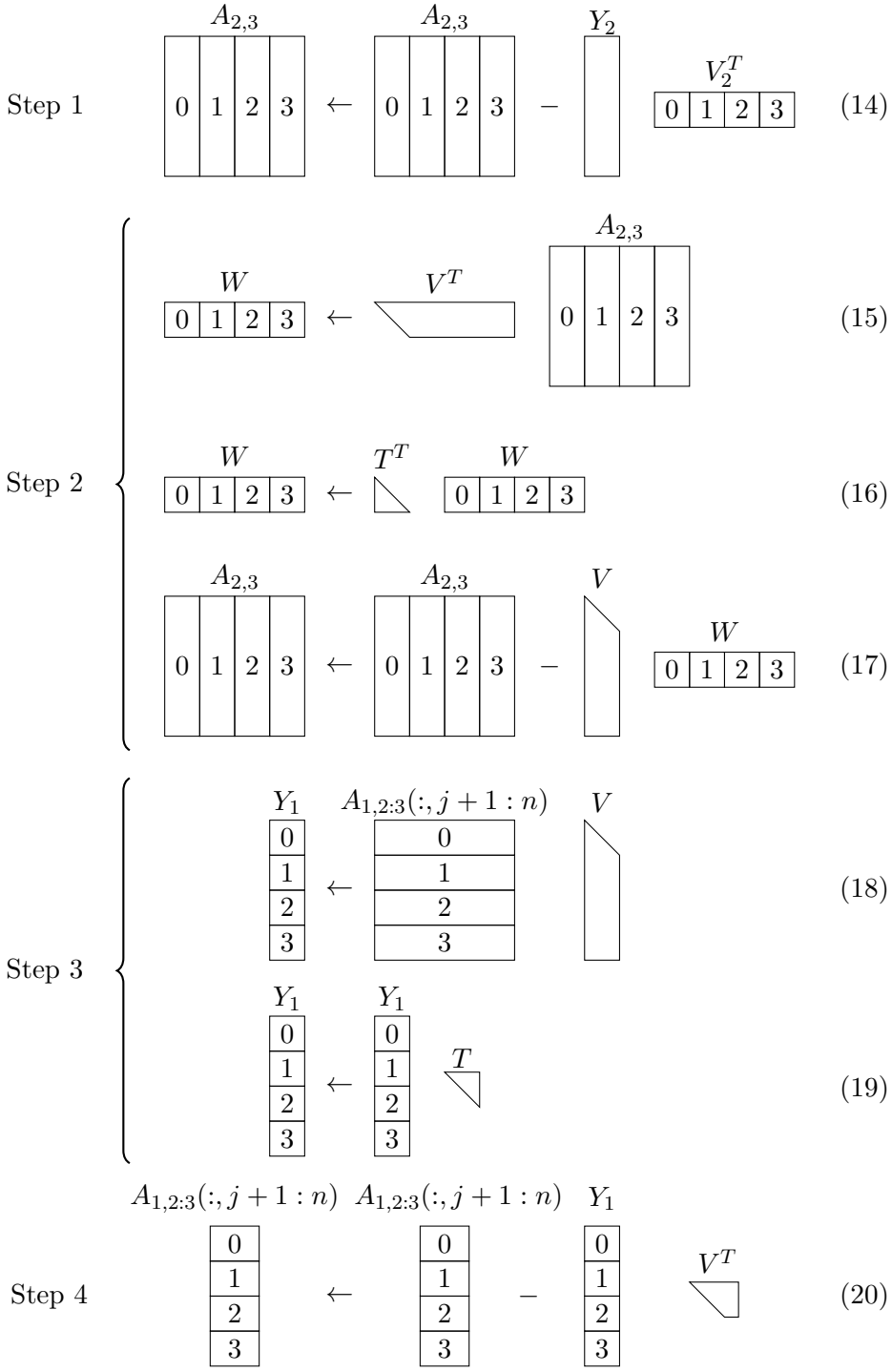

Figure 3: Illustration of the decomposition of the major steps involved in the second phase of an outer loop iteration of Algorithm 1. Numbers inside the matrices show the distribution of work over threads. Numbers in the right-hand side margin refer to equations.

Step 4. The update of $A_{1,2:3}(:, 2:n)$ from the right is achieved by the matrix outer product

$$A_{1,2:3}(:, 2:n) \leftarrow A_{1,2:3}(:, 2:n) - Y_1 V^T. \tag{20}$$

The shape of the output gradually changes from short-and-fat to tall-and-skinny as the algorithm progresses. The preferred distribution scheme is different in these two extremes. After experiments we found that block row distribution is preferred as illustrated in Figure 3.

3.3 Synchronization

The parallelization of the first phase is comparatively fine-grained even for the partial parallelization strategy. Barrier synchronization based on operating system synchronization primitives can incur a latency cost per synchronization measured in milliseconds. This is much too expensive for the relatively small problems we are targeting. Hence, we rely instead on synchronization based on shared memory and spin loops (busy-waiting; repeatedly check to see if a condition is true). The barrier primitive we use is based on a binary tree algorithm [10, Section 3] and uses only shared memory reads and writes.

4 Algorithmic Parameters

For sufficiently large problems, the key to performance is to use a panel width large enough to obtain good performance on the matrix–matrix multiplications in the second phase and to ensure that the large matrix–vector multiplication in (7) in the first phase becomes efficient. The other aspects of the computation can be more or less neglected. But in our context (AED in the multi-shift QR algorithm) we are targeting comparatively small problems solved using relatively many threads. More aspects of the computation become important since asymptotic arguments such as the one above no longer apply.

The key to portable performance is to make the implementation flexible by identifying many algorithmic parameters that can be used to tweak the performance. The aim of this section is to describe various algorithmic parameters in the blocked Hessenberg reduction algorithm with PCA, described in Section 3.

4.1 Parallelization strategy for the first phase

The first phase can be parallelized using either the full or the partial parallelization strategy, as elaborated in Section 3.1. The parallelization strategy can be set for each iteration of the outer loop independently.

4.2 Copying strategy for the PCA technique

Explicitly copying the data to a locally allocated memory area better exploits the NUMA architecture but the copying itself adds overhead. Whether to explicitly copy or not is an algorithmic parameter that can be set once per iteration of the outer loop.

4.3 Thread count

Using all available threads (cores) in the first phase may be sub-optimal, especially in the last iterations when the submatrix operations are relatively small. The thread count can in

principle be allowed to vary per iteration of the inner loop, but due to the use of parallel cache assignment it seems more appropriate to limit changes in the thread count to once per iteration of the outer loop.

Similarly, the computations in the second phase may also sometimes run faster by using fewer threads than the number of cores. The parallel overhead of the synchronization and communication may outweigh the benefits of using more cores. The number of threads to use can be set independently in each iteration of the outer loop.

With the possibility to set the number of threads independently for the two phases, we have four strategies to differentiate between:

S1: vary in both phases but keep the counts equal,

S2: vary only in the first phase and use the maximum in the second phase,

S3: vary only in the second phase and use the maximum in the first phase,

S4: vary in both phases independently.

The choice of strategy is an algorithmic parameter. There is also the related question of how to assign the threads to cores when the thread count is not equal to the core count. The thread affinity scheme is yet another algorithmic parameter and we consider two options. In the packed scheme, the threads are packed into as few NUMA domains as possible. In the distributed scheme, the threads are instead spread out over the NUMA domains as evenly as possible.

4.4 Panel width

The panel width b in Algorithm 1 can be set to any value in the range $1 \dots n-2$. When $b = 1$, the algorithm reduces to an unblocked algorithm. The overhead of using the compact WY representation grows like nb^2, which means that very large values of b are also inappropriate. Allowing the panel width to be set per execution is more flexible than fixing the panel width once and for all.

The panel width as used in Algorithm 1 is constant for the whole execution. Another possibility is to set the panel width independently for each iteration of the outer loop. Such a variable panel width adds even more flexibility.

5 Computational experiments

The aim of this section is both to compare the performance of the implementation relative to established implementations and to evaluate if the tunable parameters identified in Section 4 have any significant impact on performance. The experiments were performed on one node of the distributed memory system Abisko at the High-Performance Computing Center North (HPC2N) at Umeå University. During the course of the experiments, no other jobs were running on the same node. One node consists of four AMD Opteron 6238 processors with a total of 48 cores. Each processor contains two chips with six cores each. Each chip has its own memory controller, which means that the node has eight NUMA domains.

To match the architecture, the experiments were performed using $p \in \{6, 12, \dots, 48\}$ threads. Unless otherwise stated, successive threads were bound to cores in the same domain. Specifically, when using $p = 6$ threads, cores in the same domain (chip) are used even though

all 48 cores of the node have been reserved. In addition, memory is only allocated from domains to which a thread is also bound.

The PathScale (5.0.0) compiler is used together with the following libraries: OpenMPI (1.8.1), OpenBLAS (0.2.13), LAPACK (3.5.0), and ScaLAPACK (2.0.2). The default parameter values specified in Table 1 were used in the experiments unless otherwise stated. It is worth mentioning that the allocation of the local buffers used in the copying strategy is made once before the timing of the computation. This gives a fair comparison with LAPACK and ScaLAPACK.

Table 1: Default algorithmic parameter values.

Parameter	*Default*
Panel width	50
Number of threads	Maximum
Parallelization strategy	Partial parallelization
Copying strategy	Copy
Affinity scheme	Packed
Grid shape (ScaLAPACK)	Square if possible. Otherwise more columns.
Block size (ScaLAPACK)	50×50

All data points reported in this section is the median of 100 trials, unless otherwise stated. The Hessenberg reduction cost roughly $\frac{10}{3}n^3$ flops [6], which means that the performance in GFLOPS is calculated as:

$$(\text{Performance}) = \frac{\frac{10}{3}n^3}{(\text{Execution time in seconds}) * 10^9}. \tag{21}$$

5.1 Performance comparisons

The purpose of this section is to show the performance of the PCA variant and compare it with both LAPACK and ScaLAPACK over a range of problem sizes.

5.1.1 The performance of the PCA variant

Figure 4 shows the performance of the PCA variant on different numbers of cores. Different matrices of size $n \times n$ are tested where $n \in \{100, 300, \ldots, 3900\}$. Each graph has a point where the performance saturates. For small problems, using the maximum number of cores is not optimal.

5.1.2 Comparing the PCA variant with LAPACK

Figure 5 compares the PCA variant with the `DGEHRD` routine in LAPACK. The PCA variant is faster than `DGEHRD` for all cases except some problems when six cores are used. When only six cores are used there is no NUMA effect (all cores belong to the same domain) and hence the two implementations are virtually equivalent. Using more than six cores makes the NUMA effect noticeable. Increasing the number of cores also increases the speed-up of the

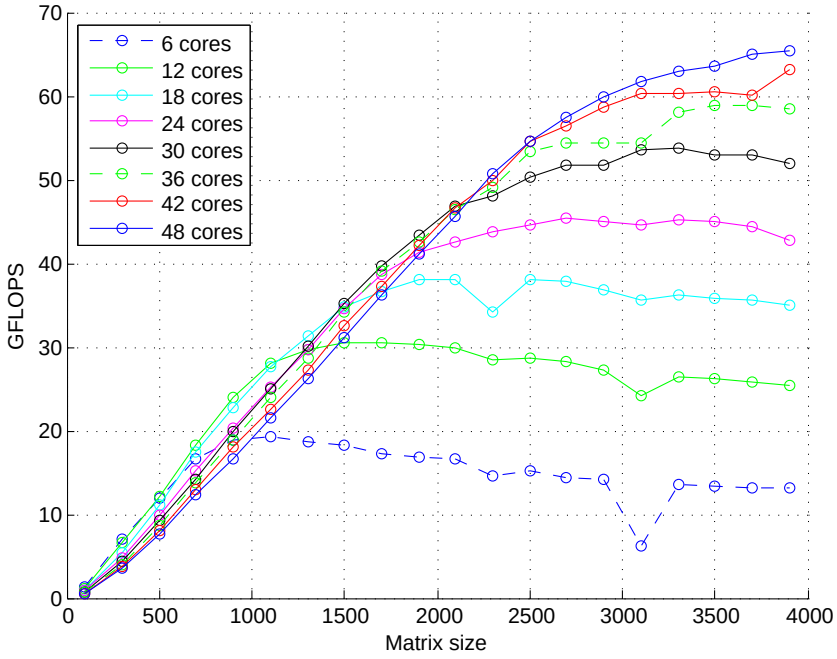

Figure 4: Performance of the PCA variant.

PCA variant over `DGEHRD`. Interestingly, increasing the problem size decreases the speed-up when more than 30 cores are used. One reason could be that the PCA variant is able to use the parallel caches more effectively and that this effect is greater than the NUMA effect that dominates for larger problems.

5.1.3 Comparing the PCA variant with ScaLAPACK

Figure 6 compares the PCA variant with the `PDGEHRD` routine in ScaLAPACK. The PCA variant performs better for small problems, but the opposite is true for large problems solved using many cores. We speculate that this is due to the PCA variant with its fast synchronization and communication is able to better parallelize tiny problems, whereas `PDGEHRD` better exploits the NUMA architecture since it is NUMA-aware in both phases and not only in the first phase as is the case for the PCA variant.

5.2 Evaluation of the tuning potential

Before researching run-time auto-tuning techniques, it is useful to determine if and if so how much can be gained by tuning each parameter separately. This will help discriminate between parameters that have little or no impact from those that have a large impact.

In this section, we systematically evaluate the tuning potential of each individual parameter while keeping all others at their default setting. The aim is to determine for each parameter whether (a) there exists a case for which tuning the parameter has a significant effect or (b) the parameter is likely to have no or little effect in all cases. Note that there is fundamental difference between the two cases. The first case is akin to an existence proof and thus merely requires the discovery of an example. The second case, however, is akin to

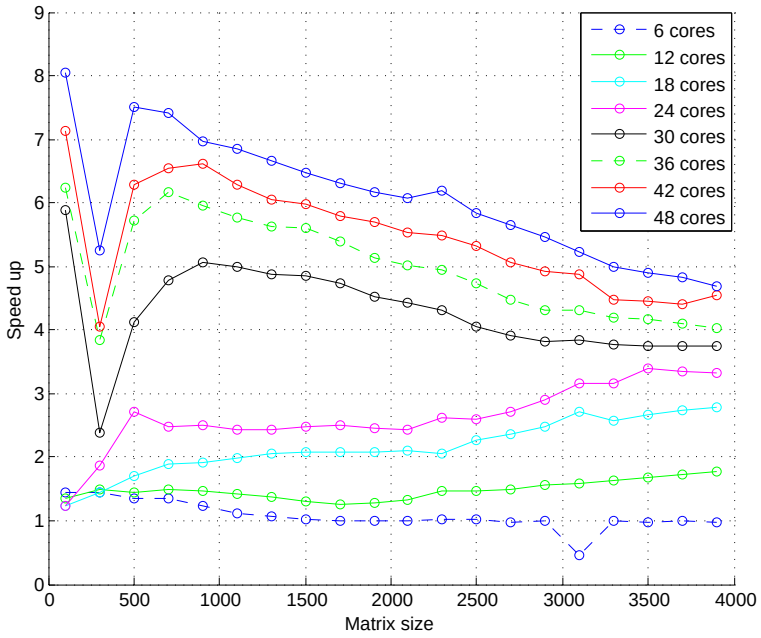

Figure 5: Comparison between the PCA variant and `DGEHRD` in LAPACK.

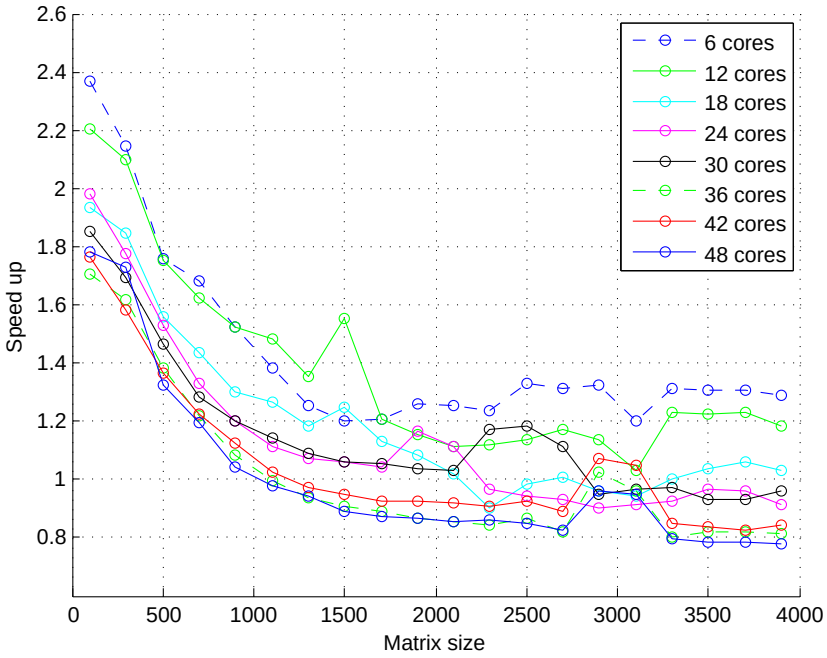

Figure 6: Comparison between the PCA variant and `PDGEHRD` in ScaLAPACK.

showing non-existence of an example and is therefore intractable; we can hope only to make a plausible case.

5.2.1 Tuning potential for the first phase parallelization strategy

As mentioned in Section 3.1, there are two strategies to parallelize the first phase. Recall that the parameter can be set independently for each iteration of the outer loop. Partial parallelization is expected to be faster for small matrices due to its lower parallelization overhead. Conversely, the full parallelization is expected to be faster for large matrices due to its larger degree of parallelism. The "size of the problem" here refers to the size of the block that is accessed by the first phase, which shrinks from one iteration to the next.

Since the parallelization overhead is a key argument in the above reasoning, an example is likely to be found by maximizing the number of threads. In addition, it is important to vary the problem size sufficiently to capture both "small" and "large" problems. This leads us to run an experiment with $p = 48$ and $n = 4000$.

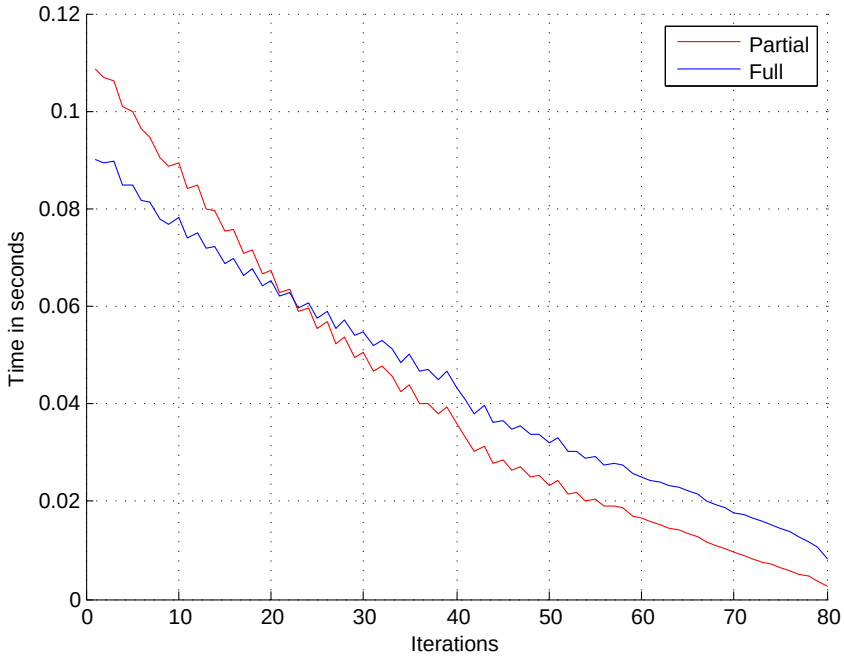

Figure 7: Comparison of the full and partial parallelization strategies.

Figure 7 shows the execution times per iteration of the outer loop for both of the parallelization strategies. Note that for the first 20 or so iterations the full parallelization strategy is faster, while the opposite is true for the remaining iterations. We conclude that there is a good potential for tuning the parallelization strategy parameter.

5.2.2 Tuning potential for the first phase copying strategy

Recall from Section 2.3 that the PCA technique consists of a mandatory part (data decomposition) and an optional part (data distribution over NUMA domains). Evaluating the tuning potential of the optional part is the aim of this section. Explicitly copying the data transforms

the algorithm from NUMA-oblivious to NUMA-aware. When the block accessed by the first phase is large it will overflow the cache and emphasize the difference in performance between NUMA-aware and NUMA-oblivious implementations.

Since the NUMA effect is a key argument in the above reasoning, we are likely to find an example where the parameter has an impact by keeping the problem large and maximizing the number of NUMA domains. This leads us to run an experiment with $p = 48$ and $n = 4000$.

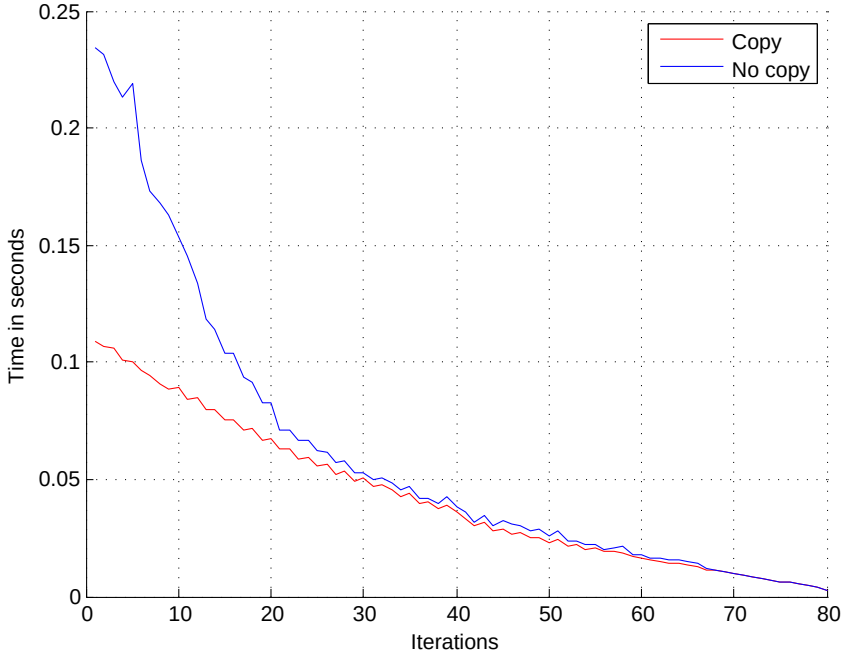

Figure 8: Comparison of the copying strategies.

Figure 8 compares the two copying strategies by showing the execution times per iteration of the outer loop. In the first 20 iterations or so, explicitly copying the data gives, as expected, much better performance. Towards the end, however, the difference between the two strategies is negligible. We conclude that copying could be enabled by default and never tuned without loosing significant performance. In other words, this parameter is not a good target for tuning.

5.2.3 Tuning potential for the thread count

As mentioned in Section 4, the number of threads used in the two phases can potentially affect the performance. In addition, the thread affinity plays a role when not all cores are used. Varying the thread count affects both the cache behavior and the parallel overhead. Following a similar line of reasoning as before, the problem(s) considered should be large enough and use many NUMA domains. Unlike previously studied parameters, however, the size of the problem (for the second phase) also includes the size of the original matrix. Hence, two experiments are considered: one small ($n = 1000$) and one large ($n = 4000$).

Also unlike previous parameters, sweeping over the range is not feasible since the parameter is actually a family of related parameters (one per iteration). Instead of sweeping over the parameter settings, we find a single specific parameter configuration by solving a simplified optimization problem with data obtained from a limited set of measurements. The underlying

assumption is that the configuration so obtained is a reasonable approximation of an optimal configuration. The actual performance using the derived parameter configuration is measured and used to judge the tuning potential of the thread count parameter.

The problem of finding optimal thread count settings becomes much simpler by assuming that the optimal number of threads to use in a particular iteration and phase does not depend on any of the other thread counts. To find the optimal configuration it suffices to know the execution time of each iteration and phase for each thread count. These data can be rapidly obtained by repeating the same execution with different fixed thread counts. Measuring the time required for each phase and iteration results in two tables: T_1 for the first phase and T_2 for the second phase (not explicitly showed). The rows correspond to thread counts and the columns correspond to iterations. Each cell thus contains the execution time for the corresponding thread count and iteration. Using these tables, the optimal thread count for a particular phase and iteration is determined by searching for the table row that contains the minimum of the column that corresponds to the iteration. The difference between the four strategies (see Section 4.3) is which table to consult. Specifically, strategy S1 consults the table $T_1 + T_2$, strategy S2 the table T_1, strategy S3 the table T_2, and strategy S4 consults both tables independently.

This exercise yields thread counts that should produce optimal results under the given assumption. However, in practice the assumption does not hold exactly. The choices are not truly independent. Therefore, another set of experiments (one per strategy) is performed where the thread counts vary as previously determined. This whole process is performed once for each of the two thread affinity schemes. The thread counts are chosen from the set $\{6, 12, \ldots, 48\}$.

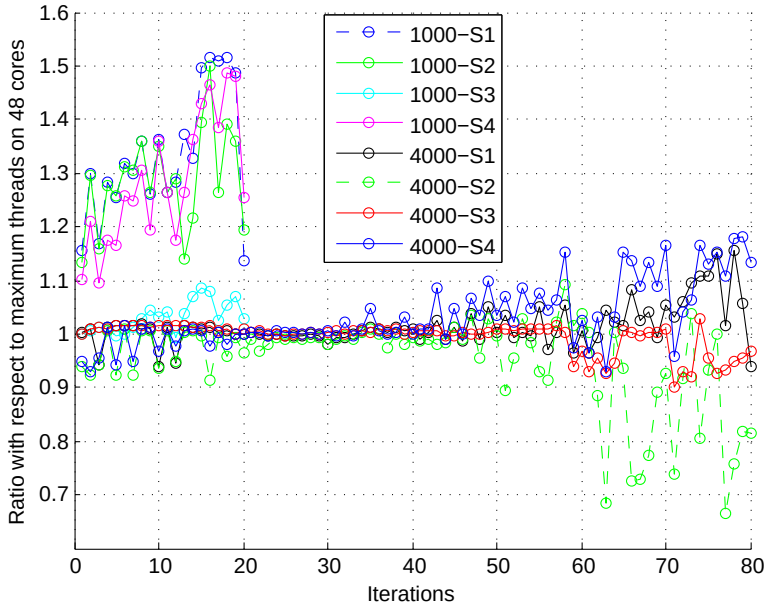

Figure 9: Comparison of the four strategies for varying the thread count using the packed affinity scheme.

Figures 9 and 10 compare the four strategies and both affinity schemes on all 48 cores. Any strategy gives a speed-up for the small problem. For the large problem, on the other

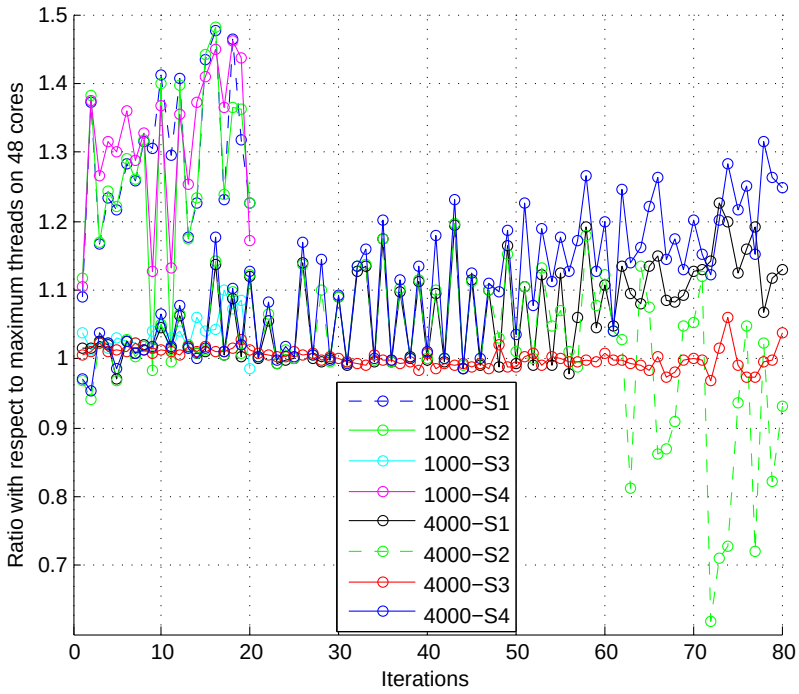

Figure 10: Comparison of the four strategies for varying the thread count using the distributed scheme.

hand, the performance is roughly unaffected for the first iterations and a speed-up is observed only in the last iterations for some of the strategies. In particular, strategy S2 (varying only in the first phase) does not give any speed-up (it actually slows down). Generally, the best performance for both affinity schemes are obtained by varying the thread count in both phases independently. This is natural since it is the most general approach and includes all others as special cases. In conclusion, the thread count for each iteration and phase is a target for tuning.

5.2.4 Panel width

The width of the panel (b) plays a key role in shaping the performance since it determines the distribution of work over different types of operations. The panel width can either be kept fixed for the whole execution or be varied from one iteration to the next.

We first evaluate the tuning potential of keeping the panel width fixed. To find if panel width depends on the problem size we used $n = \{500, 1000, \ldots, 4000\}$ and to test the distribution effect we used $p = 48$.

Figure 11 shows the performance of the PCA variant for different problem sizes and panel widths. The performance is sensitive to the choice of panel width and more so when the problem is large. We conclude that the panel width is a prime target for tuning.

We next evaluate the tuning potential of the more general strategy where the panel width is allowed to vary from one iteration to the next. From Figure 11 one can observe that the panel width impacts performance and, moreover, the optimal width depends on the problem size. Considering that the performance of one execution is determined by the performance of

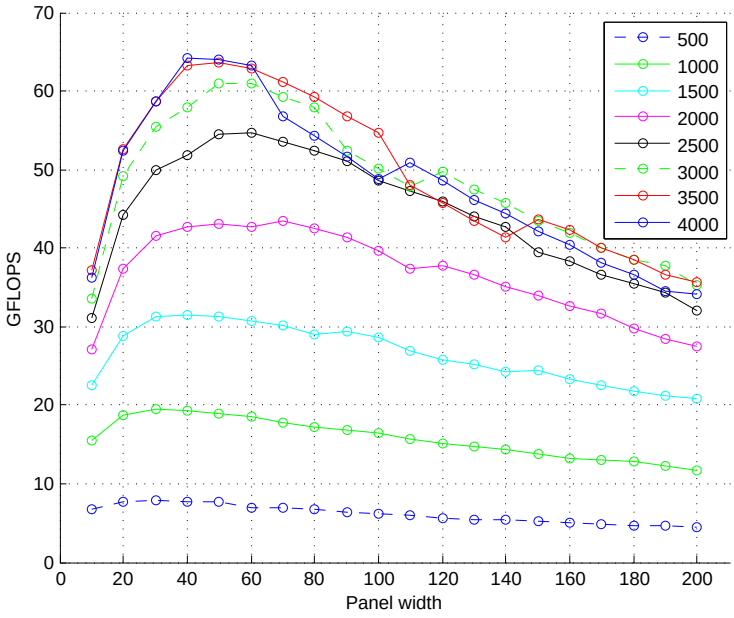

Figure 11: Illustration of the performance of the PCA variant on different problem sizes for different fixed panel widths.

each iteration of the outer loop, it is reasonable to suspect that the optimal panel width per iteration is not constant since the size of the block involved in an iteration changes over time.

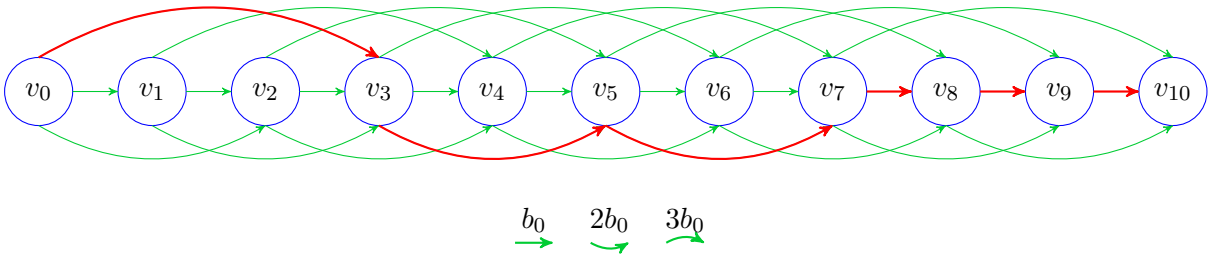

Figure 12: Illustration of a directed graph used to determine the optimal panel width configuration when the panel width is allowed to vary from one iteration to the next.

There is a huge number of possible configurations when the panel width can vary between iterations. Too many to exhaustively search or even to obtain sufficiently dense samples. Therefore, we take an alternate approach of trying to come up with an educated guess of a good configuration. Specifically, we simplify the problem by assuming that the execution time is the sum of smaller parts. Each part represents an outer loop iteration. The simplified problem can be expressed as a graph problem in the following way. For a problem of size n, construct a directed graph $G = (V, E)$ where vertex $v_k \in \{v_0, v_1, \ldots, v_{n-2}\}$ represents the state where k columns have been reduced. Clearly, the initial state is v_0 and the final state is v_{n-2}. An edge from v_k to v_{k+b} captures the act of performing an outer loop iteration with panel width b.

Under the given assumption, each edge has a well-defined weight given by the execution

time of the corresponding iteration, and the edges along the shortest path from v_0 to v_{n-2} define the optimal parameter configuration.

The question that remains to be answered is how to measure the weight of each edge. For large n, the number of edges is very large. Instead of considering all edges, we restrict ourselves to consider only edges of the form $v_k \rightarrow v_{k+b}$ where $b \in \{b_0, 2b_0, \ldots, mb_0\}$ for some constants b_0 and m. Figure 12 gives an example where $b_0 = 1$ and $m = 3$.

For each problem size we used in the fixed panel width experiment we constructed a directed graph with $b_0 = 10$ and $m = 10$. It is worth mentioning that the data points gathered to create the graphs are the median of 20 trials.

The impact of varying the panel width compared to using the optimal setting from Figure 11 is not significant, in all cases less than 4%. We conclude that a fixed panel width is sufficiently flexible.

6 Off-line auto-tuning mechanism

This section presents a simple off-line automatic tuning mechanism and compares the performance of the new algorithm after being tuned against other established implementations. The goal is not to come up with the best off-line auto-tuning mechanism but rather to have a rough idea how the algorithm performs after some tuning.

We conclude from Section 5.2 that the parameters in Table 2 are targets for tuning. The panel width needs to be tuned *once per reduction* while the other parameters need to be tuned for *each iteration*. This means the algorithm does not have only four parameters to tune. Instead, it has four *types* of parameters. In a given reduction, there is one global parameter of type panel width and three local parameters, one from each of the other three types, for each iteration.

But how many different iterations are there for a given problem? We consider two iterations different if they have different amount of computation or memory access patterns. The amount of computation and memory access patterns in an iteration depend on the size of the blocks in Figure 1. Three factors define the blocks sizes: the problem size n, the panel width b and the width of the trailing matrix (the unreduced part of the matrix) l. Since n is fixed during the reduction we identify an iteration using the pair (b, l). In general, the set of all possible iterations for a problem of size n can be defined as the set which contains all pairs (b_i, l_j), where $1 \leq i \leq n-2$ and $1 \leq j \leq i$.

Table 2: List of the parameters to be tuned.

Nr.	*Parameter*
1	Panel width
2	Parallelization strategy
3	Number of threads in the first phase
4	Number of threads in the second phase

To tune the parameters in a given reduction we used several rounds of *univariate search* (also known as orthogonal search). Univariate search works by optimizing one variable at a time, in this case through exhaustive search, while fixing the other variables. We used

univariate search by optimizing one type of parameters at a time.

The tuning round starts by tuning the panel width, using exhaustive search as mentioned. The panel width will be fixed to the size that gives the highest performance. By fixing the panel width, say to $\hat{b}$, we end up using a subset of all possible iterations, the ones associated with panel width $\hat{b}$. After that, we tune one type of parameters at a time. Exhaustive search is used to tune all the parameters from one type together at the same time. This is done by picking one value from the set of possible values for this type of parameters and test all the iterations simultaneously. Repeat that for all the values in the set. Then, compare the obtained results for each parameter independently and use the one that gives the best iteration performance. For example, assume we are tuning the number of threads in the first phase for a reduction of three iterations, where we can use any value from $\{1, 2, 3, 4, 5\}$ as a number of threads in each iteration. We start the tuning by testing all iterations at the same time using one thread. Then we repeat the test but this time we use two threads in each iteration. We repeat the test again for three, four and five threads. Now we have five results for each parameter using five different number of threads. To find the best number of threads to use in each iteration we compare the five results and choose the one with the highest performance. This could be for example, use two threads in the first iteration, four threads in the second iteration and 3 threads in the third iteration.

For parameters of type parallelization strategy, we do not change the strategy in each iteration. We assume that the full parallelization strategy is the default strategy. Then, at some iteration i, when there is not enough parallelism to use the full parallelization strategy, we switch to the partial parallelization strategy. To find this iteration i, we start by testing the full parallelization strategy for all iterations. We measure the performance at each iteration in the test. Then, we repeat the test but using the partial parallelization strategy for all iterations. And we also measure the performance at each iteration. Finally, we compare the two collected performances of each iteration with an ascending order starting from the first iteration. The switching iteration i is the first iteration where the performance of using the partial parallelization strategy is better than the performance of the full parallelization strategy. Notice that the panel width is fixed at this point so what we are searching for actually is the associated trailing matrix width after which there will not be enough parallelism to use the full parallelization strategy. This method is not the most robust way to do the tuning but it is good enough for our purpose.

We perform several rounds of the univariate search to refine the tuning. The idea is to use the tuned parameters in the *last* round as a starting point in the next round. So for example when we tune the panel width in the new round we tune it using number of threads and parallelization strategies from the last round. This way we accumulate the gained knowledge and pass it from one round to the next.

As mentioned before, the new round starts by tuning the panel width. Changing the panel width will result in using a different subset of iterations compared to the last run. So we need a way to initialize the parameters of the new round iterations using the parameters of the previous round iterations. We use a mapping function that maps each iteration in the new round to an iteration in the previous round that is close to it in amount of computation and memory access pattern.

The mapping function uses the starting column of the iterations to perform the mapping. In a reduction of a matrix of order n where b is the panel width, iteration $k \in \{1, 2, \ldots, \lceil (n - 2)/b \rceil\}$ starts at column $(k - 1)b$. The mapping is done such that the absolute difference between the starting columns of the two mapped iterations is minimized. This method ensures

that each new iteration is mapped to its most similar iteration from the previous round (in amount of computation and memory access pattern). For example, let b_1 be the panel width used in Round 1 (the last round) and b_2 be the panel width used in Round 2 (the new round). For each iteration $k_2 \in \{1, 2, \ldots, \lceil (n-2)/b_2 \rceil\}$ in Round 2 we map it to an iteration $k_1 \in \{1, 2, \ldots, \lceil (n-2)/b_1 \rceil\}$ from Round 1 that gives the minimum value of:

$$|(k_2 - 1)b_2 - (k_1 - 1)b_1|. \tag{22}$$

Figure 13 shows the mapping between two rounds where we use $b_1 = 10$ in Round 1 and $b_2 = 7$ in Round 2.

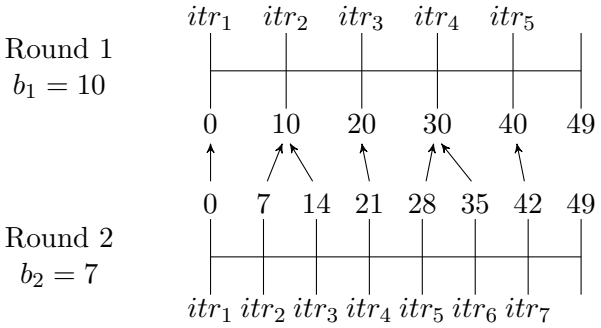

Figure 13: Illustration of mapping the iterations of two different rounds

We performed 15 rounds of tuning on matrices of different sizes using different number of allowed threads. Table 3 shows the change in the performance of the new algorithm for six random cases during the first three rounds of tuning. The performance after tuning each parameter type, following the order in Table 2, is presented. The results show that the performance could decrease after tuning one type of parameters or even after a complete round. But in general the performance improves with more tuning rounds. Figure 14 shows the speed up of the best solution found for the new algorithm after tuning against the best solution found for the `DGEHRD` function from LAPACK and the `PDGEHRD` function from ScaLAPACK. In selecting the best performance after tuning we considered the performance at the end of each tuning round only. The results show that the new algorithm outperforms LAPACK for all the tested problems while it outperforms ScaLAPACK for the small size problems up to order 1500.

The best way to judge the efficiency of a tuning algorithm is to compare its results against the best found after exhaustive search. Unfortunately, exhaustive search is not feasible in our case, which means we have no way of assessing the effectiveness of the tuning mechanism.

7 Conclusion

In this paper we presented a new parallel implementation of the blocked Hessenberg reduction. The proposed algorithm is aimed to speedup the costly AED procedure which is in the critical path of the distributed parallel multi-shift QR algorithm. The proposed implementation is characterized by having a high degree of flexibility and a high level of locality of memory references. The proposed solution outperforms LAPACK's routine `DGEHRD` for most cases and ScaLAPACK's routine `PDGEHRD` for small problem sizes.

Table 3: First three rounds of tuning different problems

		GFLOPS					
Matrix size		$n = 500$	$n = 1100$	$n = 1300$	$n = 2300$	$n = 3100$	$n = 3700$
# cores		$p = 24$	$p = 36$	$p = 30$	$p = 48$	$p = 18$	$p = 42$
w/o tuning		10.7236	31.1575	33.4703	56.3084	38.6603	73.7369
Round 1	param 1	11.3570	31.2902	34.0326	55.7050	39.3838	75.7032
	param 2	11.4936	31.1101	33.9934	55.5832	44.9821	77.3813
	param 3	11.7467	31.3054	33.3981	55.2197	45.1992	78.0871
	param 4	13.5506	32.5398	35.3458	57.3648	44.3091	78.1880
Round 2	param 1	13.5479	32.4150	35.3175	58.3909	46.8890	82.3765
	param 2	13.3568	32.0874	32.6908	51.1839	46.6522	82.9368
	param 3	13.6299	32.5625	33.4370	51.8943	46.5636	82.7718
	param 4	13.7256	33.0276	36.0002	57.8282	46.0662	83.4532
Round 3	param 1	14.0032	33.0210	35.9560	57.0685	46.8001	85.8693
	param 2	13.5132	31.7747	33.1033	54.4554	46.5290	75.7112
	param 3	13.6769	31.2970	33.2005	55.2553	46.7058	75.9872
	param 4	13.9985	32.6766	35.5517	57.2649	45.9611	82.6212

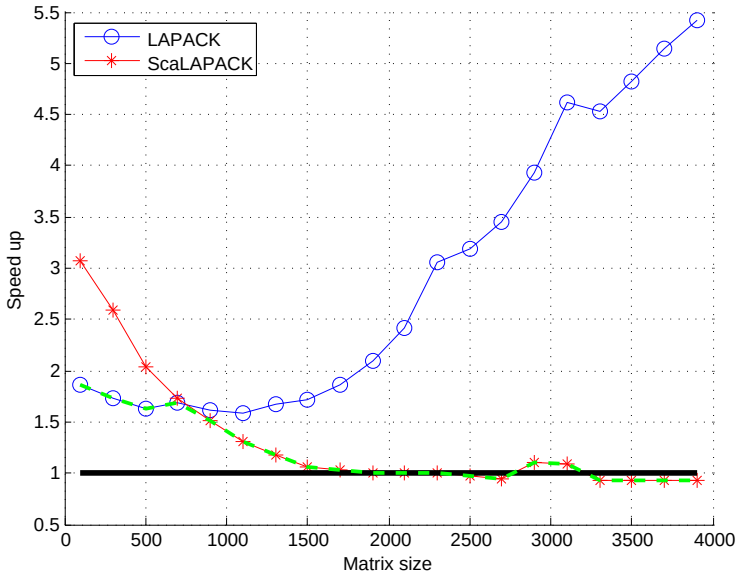

Figure 14: Comparison between the tuned PCA variant and similar established implementations from LAPACK and ScaLAPACK

Various algorithmic parameters of the proposed algorithm were evaluated to find which ones have high impact on performance. It is found that the parallelization strategy for the first phase, the thread count in each phase, and the panel width have significant impact on performance. On the other hand, the copying strategy for the first phase can easily be set optimally without tuning. In addition, the impact of varying the panel width in each outer loop iteration was found to be insignificant and thus using fixed panel width per execution is enough. More over, an off-line auto-tuning mechanism was presented to see the impact of auto-tuning on the proposed algorithm.

Future work includes designing an on-line (dynamic) auto-tuning mechanism for tuning the most significant parameters identified in this paper. The aim is to have a mechanism that gives the algorithm the highest degree of flexibility with respect to various architecture characteristics.

Acknowledgements

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

References

[1] K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm. Part II: Aggressive early deflation. *SIAM Journal on Matrix Analysis and Applications*, 23(4):948–973, 2002.

[2] A. M. Castaldo and R. C. Whaley. Scaling LAPACK panel operations using parallel cache assignment. In *ACM SIGPLAN Notices*, volume 45, pages 223–232. ACM, 2010.

[3] A. M. Castaldo and R. C. Whaley. Achieving scalable parallelization for the Hessenberg factorization. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 65–73. IEEE, 2011.

[4] A. M. Castaldo, R. C. Whaley, and S. Samuel. Scaling LAPACK panel operations using parallel cache assignment. *ACM Transactions on Mathematical Software*, 39(4), 2013.

[5] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215–227, 1989.

[6] G. H. Golub and C. F. Van Loan. *Matrix Computations*, volume 3. JHU Press, 2012.

[7] R. Granat, B. Kågström, D. Kressner, and 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, SE-901 87, 2012.

[8] M. R. Hasan and R. C. Whaley. Effectively exploiting parallel scale for all problem sizes in LU factorization. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1039–1048. IEEE, 2014.

[9] L. Karlsson and B. Kågström. Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures. *Parallel Computing*, 37(12):771 – 782, 2011. 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10).

[10] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[11] G. Quintana-Ortí and R. van de Geijn. Improving the performance of reduction to hessenberg form. *ACM Transactions on Mathematical Software*, 32(2):180–194, 2006.

[12] R. S. Schreiber and C. Van Loan. A storage efficient WY representation for products of Householder transformations. Technical report, Cornell University, 1987.