# Adaptive Parallel Householder Bidiagonalization[⋆]

Fangbin Liu[1] and Frank J. Seinstra[2]

[1] ISLA, Informatics Institute, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
fliu@science.uva.nl

[2] Department of Computer Science, VU University
De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands
fjseins@cs.vu.nl

**Abstract.** With the increasing use of large image and video archives and high-resolution multimedia data streams in many of today's research and application areas, there is a growing need for multimedia-oriented high-performance computing. As a consequence, a need for algorithms, methodologies, and tools that can serve as support in the (automatic) parallelization of multimedia applications is rapidly emerging.

This paper discusses the parallelization of Householder bidiagonalization, a matrix factorization method which is an integral part of full Singular Value Decomposition (SVD) — an important algorithm for many multimedia problems. Householder bidiagonalization is hard to parallelize efficiently because the total number of matrix elements taking part in the calculations reduces during runtime. To overcome the growing negative performance impact of load imbalances and overprovisioning of compute resources, we apply adaptive runtime techniques of *periodic matrix remapping* and *process reduction* for improved performance. Results show that our adaptive parallel execution approach provides a significant improvement in efficiency, even when applying a set of compute resources which is (initially) very large.

## 1 Introduction

The research area of Multimedia Content Analysis (MMCA) considers all aspects of the automatic extraction of knowledge from large multimedia archives and video streams. The MMCA domain is rapidly facing a computational problem of phenomenal proportions, as digital video produces high data rates, and multimedia archives steadily run into Petabytes of storage. As individual desktop computers can no longer satisfy the increasing computational demands, the use of compute clusters, grids, and cloud systems is rapidly becoming indispensible.

A common way to help researchers and developers in the MMCA domain in the development of high-performance applications is to provide a library of

⋆ This research is supported by the Netherlands Organization for Scientific Research (NWO) under NWO-GLANCE grant no. 643.000.602.

pre-parallelized building block operations that hide the complexities of parallelization behind a sequential programming interface. Over the past years we have developed one such library for *user transparent* parallel multimedia computing on compute clusters, called Parallel-Horus, which has been applied for implementation of a number of state-of-the-art MMCA applications [1].

While Parallel-Horus incorporates most algorithms commonly applied in the field, it is essential to expand the set of operations to further enhance the library's applicability. Important missing functionality is that of matrix factorization by way of Singular Value Decomposition (SVD). More specifically, in this paper we focus on the most interesting — and most computationally demanding — part of one approach to SVD factorization, i.e.: Householder bidiagonalization.

Parallel solutions to Singular Value Decomposition in general, and Householder bidiagonalization in particular, have been studied extensively in the literature [2], [3], [4], [5], [6]. In contrast to such earlier efforts, our main focus is on the integration of a parallel solution to Householder bidiagonalization behind a user transparent parallel programming interface. As such, the foremost research question underlying the work described in this paper is stated as follows: can we implement a parallel Householder bidiagonalization method based on the basic building block operations available in Parallel-Horus, such that it integrates effortlessly, yet efficiently, into our user transparent parallel library?

Householder bidiagonalization is hard to parallelize efficiently. This is because the size of the *working set* (i.e., the number of matrix elements taking part in the calculations) reduces over time. As a result, while it is often beneficial to use a large number of compute nodes in the early stages of the execution, the cost of parallelization and load imbalances eventually outweigh the cost of actual calculations, to the effect that obtained speedups (if at all) are generally low.

In the solution proposed in this paper we apply an approach called *periodic matrix remapping* for load balancing. Further, to optimize performance under the continuous reduction of the working set, we apply an approach called *process reduction* to gradually reduce the number of compute nodes at runtime. The decision making for process reduction is based on a performance model that continuously compares performance results obtained on the current set of nodes with estimations for a reduced number of nodes. Extensive evaluation of our solution to parallel Householder bidiagonalization shows that we obtain high speedups, even for a large number of compute nodes.

This paper is organized as follows. Section 2 introduces the Parallel-Horus multimedia computing library. Section 3 introduces the Householder bidiagonalization method. Section 4 discusses our adaptive parallel Householder bidiagonalization approach, and presents a simple performance model used for performance optimization. Subsequently, Section 5 gives an evaluation of the obtained performance and speedup results. Concluding remarks are given in Section 6.

## 2 Parallel-Horus

Parallel-Horus is a cluster programming framework that allows programmers to implement data parallel multimedia applications as fully sequential programs.

The Parallel-Horus framework consists of commonly used multimedia data types and associated operations, implemented in C++ and MPI. The library's API is made identical to that of an existing sequential library: Horus [7]. Similar to other frameworks [8], Horus recognizes that a small set of *algorithmic patterns* can be identified that covers the bulk of all multimedia-oriented functionality.

Horus includes patterns for all such functionality, including unary and binary pixel operations, global reduction, generalized convolution, iterative and recursive neighborhood operations, and geometric transformations. Current developments include patterns for operations on large datasets, as well as patterns on increasingly important data structures, such as feature vectors obtained from earlier calculations. For reasons of efficiency, all Parallel-Horus operations are capable of adapting to the performance characteristics of a parallel machine at hand, i.e. by being flexible in the partitioning of data structures. Moreover, it was realized that it is not sufficient to consider parallelization of library operations *in isolation*. Therefore, the library was extended with a run-time approach for communication minimization (called *lazy parallelization*) that automatically parallelizes a fully sequential program at runtime by inserting communication primitives and additional memory management operations whenever necessary.

Results for realistic multimedia applications have shown the feasibility of the Parallel-Horus approach, with data parallel performance (obtained on individual cluster systems) consistently being found to be optimal with respect to the abstraction level of message passing programs [1]. Notably, Parallel-Horus was applied in recent NIST TRECVID benchmark evaluations for content-based video retrieval, playing a crucial role in achieving top-ranking results in a field of strong international competitors [9], [10]. Moreover, recent extensions to Parallel-Horus, that allow for services-based multimedia Grid computing, have been applied successfully in large-scale distributed systems, involving hundreds of massively communicating compute resources covering our entire globe [1]. Real-time and off-line applications implemented with this extended system have resulted in a 'best technical demo award' at ACM Multimedia 2005 [11] and a 'most visionary research award' at AAAI 2007 [12]. Also, Parallel-Horus has been used in our prize-winning contribution to the First IEEE International Scalable Computing Challenge at CCGrid 2008 (Lyon, France) [13].

Clearly, Parallel-Horus is a system that serves well in bringing the benefits of high-performance computing to the multimedia community, but we are constantly working on further improvements, as exemplified in the following sections. Prototypical code (in C and MPI) of an earlier proof-of-concept implementation of Parallel-Horus, as well as of several example image processing applications, is available at: http://www.science.uva.nl/~fjseins/ParHorusCode/.

## 3 Singular Value Decomposition

In linear algebra, Singular Value Decomposition (SVD) is an important factorization method for rectangular real or complex matrices. The method is used in many application areas, including a.o. numerical weather prediction and multimedia content analysis. As stated in [14], a real $m \times n$ matrix $A$ with $m \geq n$ can be decomposed into three matrices:

$$A = U\Sigma V^T, U : m \times n, V : n \times n \tag{1}$$

where $U^T U = V^T V = V V^T = I_n$, matrix $\Sigma = diag(\sigma_1, \ldots, \sigma_n)$, and diagonal elements $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$ being the *singular values* of matrix $\Sigma$.

As shown in [15], the SVD factorization for a matrix $A$ can be performed in two steps. First, matrix $A$ is reduced into *upper bidiagonal form* by way of a series of *Householder transformations*. Second, the QR algorithm is performed to find the singular values of the upper bidiagonal matrix. These two phases combined properly produce the singular value decomposition of matrix $A$. In this paper, we will focus only on the first phase of the SVD factorization (i.e., the Householder bidiagonalization), as it is the most computationally demanding part of the calculation.

## 3.1 Householder Bidiagonalization

Householder bidiagonalization, or the reduction of input matrix $A$ into upper bidiagonal form, proceeds by alternately pre- and post-multiplying $A$ by so-called *Householder transformations*

$$P^{(k)} = I - 2x^{(k)}x^{(k)T}, \ with \ k = 1, 2, \ldots, n$$

and

$$Q^{(k)} = I - 2y^{(k)}y^{(k)T}, \ with \ k = 1, 2, \ldots, n-2$$

where $x^{(k)T}x^{(k)} = y^{(k)T}y^{(k)} = 1$, such that

$$P^{(n)} \ldots P^{(1)} A Q^{(1)} \ldots Q^{(n-2)} = \left\{ \begin{array}{ccccc} q_1 & e_2 & 0 & \ldots & 0 \\ 0 & q_2 & e_3 & & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ \vdots & & & \ddots & e_n \\ 0 & \ldots & \ldots & 0 & q_n \\ \vdots & & & & \vdots \\ 0 & \ldots & \ldots & \ldots & 0 \end{array} \right\} \tag{2}$$

Before discussing the parallelization of this Householder bidiagonalization in the next Section, we will first present a thorough overview of the data dependencies involved in the underlying algorithm.

## 3.2 Data Dependencies

Generally speaking, Householder bidiagonalization takes place in $n$ iterations, where each iteration involves one pre-multiplication and one post-multiplication

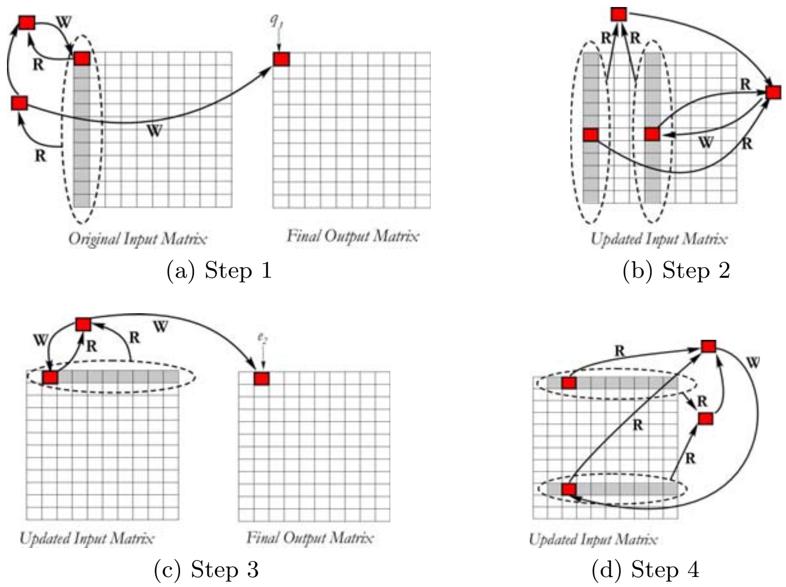


**Fig. 1.** Four steps in a single Householder iteration. R = read access; W = write access

resulting in a new (intermediate) $m \times n$ matrix $H^{(i)}$. In each iteration $i$ (with $0 \leq i \leq n$) all elements in column $i$ of $H^{(i)}$ are being set to 0, except for diagonal element $q_i$ and the upper diagonal element $e_i$ directly above it (if it exists). Similarly, all elements in row $i$ of $H^{(i)}$ are being set to 0 except for diagonal element $q_i$ and the upper diagonal element $e_{i+1}$ directly to its right (if it exists). Furthermore, all elements in the right-lower $(m-i) \times (n-i)$ submatrix of $H^{(i)}$ are being updated. In iteration $i+1$ this right-lower $(m-i) \times (n-i)$ submatrix of $H^{(i)}$ is being used as the basis for all calculations. The set of all elements in this $(m-i) \times (n-i)$ submatrix is what we refer to as the *working set* for iteration $i+1$.

In practice, a single iteration of Householder bidiagonalization consists of four major consecutive steps (see Figure 1). While all $i$ iterations are very similar, for ease of presentation we focus on the first iteration only (i.e.: $i = 1$). In the first of the four steps, the first diagonal element $q_1$ is calculated and stored in the final bidiagonal output matrix. As shown in Figure 1(a), the calculation of $q_1$ is dependent on all values in column 1. In addition, a new (temporary) value for $A[1,1]$ is calculated and stored in-place, on the basis of $q_1$ and $A[1,1]$ itself.

In the second step, all elements in all columns to the right of column 1 are being updated. As shown in Figure 1(b), each updated element $A[x,y]$ is dependent on all elements in column 1 as well as on all elements in column $y$. The update of each matrix element in this second step is again in-place.

Step 3 is rather similar to Step 1. In this step, the first upper diagonal element $e_2$ is calculated and stored in the final bidiagonal output matrix. As shown in Figure 1(c), the calculation of $e_2$ depends on all values in row 1, except the first.

As in the first step, a new (temporary) value for $A[1,2]$ is calculated and stored in-place, on the basis of $e_2$ and $A[1,2]$.

The fourth step is very similar to Step 2. In this final step, all elements in all rows below row 1 are being updated, except for the first element in each row. Figure 1(d) shows that each updated element $A[x,y]$ is dependent on all elements in row 1 as well as on all elements in row $x$. As before, the update of each matrix element is an in-place operation.

The completion of these four calculation steps also completes one iteration of the Householder bidiagonalization. As stated above, the right-lower sub-diagonal matrix of the matrix that results after Step 4 constitutes the working set for the next iteration. In practice this means that each subsequent iteration uses an smaller matrix as input: i.e. the output matrix of Step 4 excluding its first row and its first column. As discussed in the following Section, this continuous reduction of the working set has important consequences for parallel execution.

## 4 Parallelization

In the parallelization of the Householder bidiagonalization, two strategic choices must be made: the parallel execution of a single Householder iteration, and the parallel execution of all iterations in turn. Below, we will first present our considerations with respect to the parallelization of a single iteration. This is followed by a discussion of the parallel execution of the full Householder bidiagonalization.

### 4.1 Parallelizing a Single Iteration

In this section we focus on the parallelization of a single iteration of the algorithm as presented in Figure 1, as well as on the integration with the Parallel-Horus library. For reasons beyond the scope of this paper, Parallel-Horus only offers implementations on the basis of the paradigm of *data parallelism*. As a consequence, we restrict our considerations to data parallel solutions only.

Parallel-Horus offers a rich set of data partitioning routines, which allow data structures to be split up in a multitude of ways. The most commonly used data partitioning routines provide splitting and scattering of data structures in a horizontal (or row-wise), vertical (or column-wise), and hybrid (or block-wise) manner. Each of these data partitioning routines ensures that the workload is spread in a balanced way: each compute node in the system will get an (approximately) equal subsection of the entire data structure.

In principle, each of these three data partitioning strategies is a good candidate for parallelization of a single iteration of the Householder bidiagonalization. When considering the data dependencies discussed in the previous section, however, there are several issues that are of importance. In case of column-wise (vertical) partitioning of the data structures involved in the Householder algorithm, we see that Step 1 can be executed without problems. In fact, only one of the nodes has work to do — and it can do so immediately because all data dependencies are resolved locally (i.e. without the need for inter-node communication). Step 2 is only marginally more problematic. The updating of the matrix

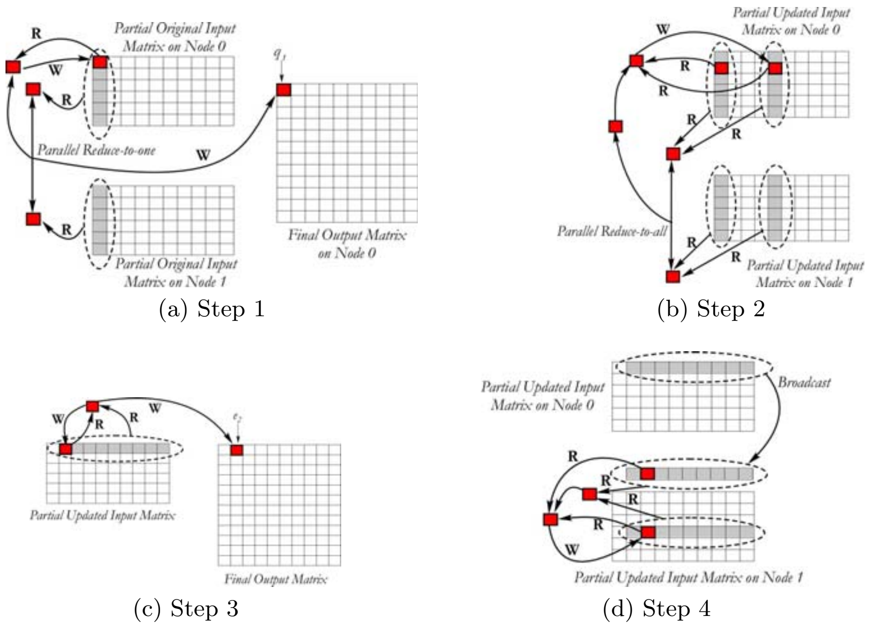

R
W
R
Partial Original Input Matrix on Node 0
Parallel Reduce-to-one
W
$q_1$
R
Partial Original Input Matrix on Node 1
Final Output Matrix on Node 0
(a) Step 1
W
Partial Updated Input Matrix on Node 0
R
R
R
R
Parallel Reduce-to-all
R
R
Partial Updated Input Matrix on Node 1
(b) Step 2
W
R
R
W
$e_1$
Partial Updated Input Matrix
Final Output Matrix
(c) Step 3
Partial Updated Input Matrix on Node 0
Broadcast
R
R
R
R
W
Partial Updated Input Matrix on Node 1
(d) Step 4

**Fig. 2.** A single parallel Householder iteration by way of horizontal data partitioning

elements in this step requires each node to have access to the first matrix column. Hence, this column would have to be provided (broadcast) to all compute nodes in the system. Steps 3 and 4 are more complicated, as in both steps many of the data dependencies are not local. As a result, inter-node communication (by way of parallel reduction) would be needed to resolve these dependencies.

Clearly, a similar line of reasoning holds for row-wise partitioning. Cyclic block-wise partitioning, on the other hand, would cause non-local data dependencies to exist in all four steps of the algorithm, so we do not consider this approach any further. One further approach, which would overcome much of the non-local data dependencies as existent in both vertical and horizontal partitioning, is to use vertical partitioning in the first two steps, and horizontal partitioning in the last two. This, however, would require a complete matrix re-distribution — causing significant communication overhead — in the heart of the parallel algorithm. Initial experiments with this parallelization approach applied to various numbers of nodes in a real cluster system indeed showed an increase in the execution time of a factor of two or more, when compared to the approach based on horizontal partitioning. As a result, we also do not consider this two-step vertical-horizontal partitioning approach further.

Finally, a comparison of results obtained for horizontal versus vertical partitioning — which theoretically should give identical results — showed that horizontal partitioning is always significantly faster (on average about 20%). This is explained by the fact that, in the case of horizontal partitioning, most of the data structures to be communicated are stored contiguously in memory. In contrast, vertical partitioning causes data structures to be stored non-contiguously, a problem which is discussed extensively in [16]. As a result, horizontal partitioning is our parallelization strategy of choice for each single iteration of the Householder bidiagonalization (see Figure 2).

## 4.2 Executing Multiple Iterations

In this section we build further on the parallelization approach presented above, and discuss the parallelization of consecutive Householder iterations. As stated in Section 3, one important problem of our algorithm is that the working set (i.e.: the total number of matrix elements taking part in the calculations) reduces over time. Specifically, in comparison to the matrix used in iteration $i$, the matrix used in iteration $i + 1$ has lost one row and one column.

The continuous reduction of the size of the working set has two important consequences for our parallelization strategy. First, if we would apply the original horizontal partitioning for all iterations without change, the parallel execution would face a gradually increasing *load imbalance*. This is because the loss of matrix elements is not evenly distributed over all compute nodes. The node that has been provided with the uppermost part of the matrix will be the first to gradually loose all its data *before* any of the other nodes looses even a single matrix row (note that, at the same time, all partial matrices on all nodes do loose columns in an evenly distributed manner). When the algorithm would progress in this way, eventually only a single node would be calculating on the remaining data — with all other nodes wasting idle cycles.

A second problem appears when we assume that we would be able to implement a mechanism that overcomes load imbalances. With such a mechanism the algorithm would run at a maximum level of parallelism: all nodes would be kept busy until the final phase of the calculations. While generally beneficial, such a mechanism would still not ensure maximum performance. This is because the amount of calculations to be performed in each iteration decreases at a faster rate than the amount of communication required to resolve all data dependencies. In other words, the computation versus communication ratio changes over time: in the initial stages of the calculations this ratio is generally high, but it gradually decreases while the algorithm progresses. Eventually, the time spend on communication can (en will) overtake the time spend on actual calculations. Stated differently, the parallel algorithm will suffer from a gradual *overprovisioning of compute resources*.

### Adaptive Parallelization

To overcome the first problem of load imbalances we apply a method called *periodic matrix remapping*. In general, this means that for a parallel system consisting of $n$ nodes the working set is repartitioned after each $n$ iterations of the Householder algorithm. After the repartitioning has taken place, the working set is again evenly distributed over all nodes in the system. In practice, we have implemented the repartitioning by way of *upward matrix row-shifting*, meaning that each compute node communicates a number of rows to its upper neighbor and receives a (smaller) number of rows from its lower neighbor.

To overcome the additional problem of the overprovisioning of resources, we have extended our periodic matrix remapping approach with an approach called *process reduction*. Essentially, this approach causes a gradual decrease in the number of compute nodes applied in the calculations, to the effect that the final phase of the algorithm may (and generally will) use just a single compute node — with all remaining nodes being available for other tasks.

Clearly, for optimal performance one needs to carefully select the moment and frequency at which process reductions take place. Reducing the number of nodes too early leads to a (temporary) underprovisioning of compute resources. Doing it too late or too infrequently may cause the cost of the communication performed by the algorithm to outweigh the cost of computation for too long. Reducing the number of nodes too often also will downgrade the performance of the algorithm, due to the inherent communication needed for the process reductions themselves.

## Performance Estimation

For performance optimization we apply a performance model that can estimate the parallel performance for a single iteration of the Householder algorithm on any given number of nodes. The results of the model are used to steer the decision making for the execution of process reduction.

The applied performance model is based on our earlier work on the estimation of the computation costs and the communication costs of data parallel image and video applications [16], [17]. A full discussion of the underlying modeling techniques is far beyond the scope of this paper. At a high level of abstraction, however, we can state that the execution time $T_{seqHH}$ of the sequential computations executed in a single Householder iteration performed on a $m \times n$ input matrix can be approximated by

$$T_{seqHH} = (2m - 1) \times (n - 1) \times C_{update}$$

where $C_{update}$ is the cost of updating a single matrix element, as performed in Steps 2 and 4 of our sequential algorithm (see Figure 1). The (static) benchmarking process for obtaining an accurate value for this model parameter is explained in [17]. Note that we have abstracted away all other parts of the sequential calculations, which is acceptable for our purposes.

The execution time $T_{parHH}$ for parallel execution of the computations taking place in our algorithm is simply obtained by

$$T_{parHH} = \frac{T_{seqHH}}{nrCPUs}.$$

In addition, the execution time $T_{comm}$ of the communication steps executed in a single Householder iteration can be approximated by

$$T_{comm} = (2n - 1) \times C_{bcast}$$

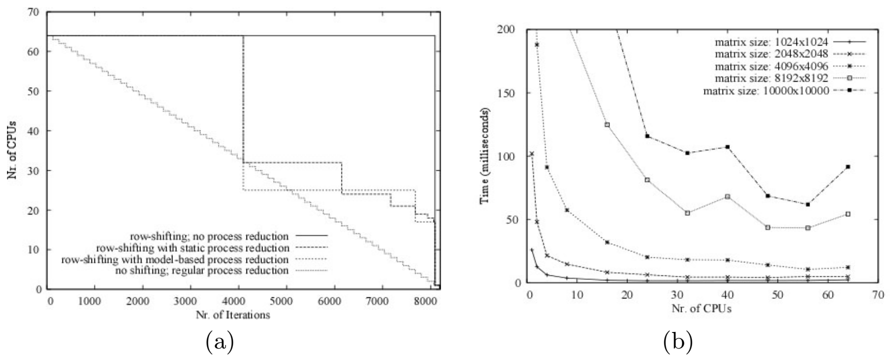


**Fig. 3.** (a) Four adaptive parallel execution strategies; (b) execution times for a single Householder iteration using matrices ranging from $1024^2$ to $10000^2$

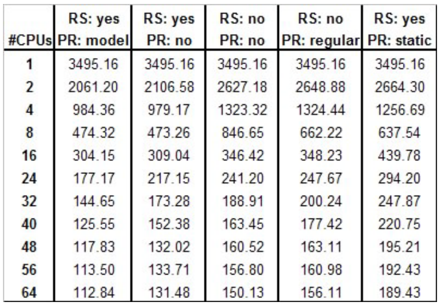
where $C_{bcast}$ is the cost for performing a broadcast of a 8-byte double value in a parallel system consisting of $nrCPUs$ nodes. A description of the modeling of such a broadcast operation, and the required benchmarking process, is given in [16]. Note that, again, we have abstracted away several communication steps, including a parallel reduce-to-one operation.

We have evaluated our model by comparing measurement results obtained for a single Householder iteration with our model predictions. At all times, our predictions where within 3 to 5% of the measurements, a result which is entirely in line with our earlier results presented in [16], [17].

## 5 Evaluation

We have tested our implementations on one of the clusters part of the 5-cluster Distributed ASCI Supercomputer 3 (DAS-3) installed in The Netherlands. The cluster, located at VU University (Amsterdam), consists of 85 dual-CPU/dual-core 2.4 GHz AMD Opteron DP 280 compute nodes (each having 4 GB of memory), all of which are linked via a high speed Myri-10G interconnect.

In our evaluation we present results for five different parallel execution strategies. The first strategy only applies horizontal data partitioning, but does not apply adaptive parallel execution. Essentially, this strategy serves as the basis for our comparison. In the second strategy, we apply upward matrix row-shifting for load balancing, but we do not apply process reduction. In the third approach, we perform process reduction at regular intervals (i.e.: whenever a node is found to have an empty working set), without performing load balancing. In essence, these latter two approaches constitute two extreme ends in the range of adaptive parallel execution strategies. Our two remaining strategies do apply load balancing by way of row-shifting as well as process reduction. The two strategies differ in that the first applies process reductions at fixed (pre-selected) instances, while the second approach makes all decisions regarding process reductions on the basis of our performance model. Figure 3(a) presents the gradual reductions taking place for each of the four *adaptive* strategies.



| #CPUs | RS: yes PR: model | RS: yes PR: no | RS: no PR: no | RS: no PR: regular | RS: yes PR: static |
|---|---|---|---|---|---|
| 1 | 3495.16 | 3495.16 | 3495.16 | 3495.16 | 3495.16 |
| 2 | 2061.20 | 2106.58 | 2627.18 | 2648.88 | 2664.30 |
| 4 | 984.36 | 979.17 | 1323.32 | 1324.44 | 1256.69 |
| 8 | 474.32 | 473.26 | 846.65 | 662.22 | 637.54 |
| 16 | 304.15 | 309.04 | 346.42 | 348.23 | 439.78 |
| 24 | 177.17 | 217.15 | 241.20 | 247.67 | 294.20 |
| 32 | 144.65 | 173.28 | 188.91 | 200.24 | 247.87 |
| 40 | 125.55 | 152.38 | 163.45 | 177.42 | 220.75 |
| 48 | 117.83 | 132.02 | 160.52 | 163.11 | 195.21 |
| 56 | 113.50 | 133.71 | 156.80 | 160.98 | 192.43 |
| 64 | 112.84 | 131.48 | 150.13 | 156.11 | 189.43 |

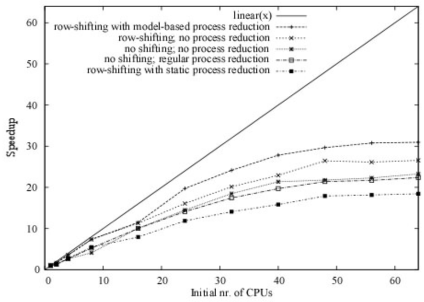


**Fig. 4.** Performance (left) and speedup (right) results for our five parallel execution strategies using a matrix of $8192^2$. RS = row-shifting; PR = process reduction.

Before presenting results for complete runs of the algorithm, we will first briefly evaluate the data parallel performance of a single Householder iteration. We have measured the execution time for a single iteration applied to a wide range of input matrix sizes on a varying number of nodes. As expected, and as shown in Figure 3(b), for fastest execution the number of nodes to be used is dependent on the size of the input matrix: for smaller matrices it is beneficial to use a smaller number of compute nodes. From this we conclude that, with a gradually decreasing working set, process reduction should be beneficial indeed.

Performance in seconds and speedup characteristics for our five parallel execution strategies are given in Figure 4. As can be seen, our implementation based on performance modeling — applied to a matrix of size $8192^2$ — provides highest performance. All other strategies lag behind, with the implementation based on static (fixed) process reduction being the slowest. Interestingly, the non-adaptive parallel implementation still is the third fastest — even though it is significantly slower than our model-based approach. From this, we conclude that the combination of periodic matrix remapping and process reduction indeed can give improved performance, but only if the decision making process underlying these approaches cleverly incorporates the runtime characteristics of the algorithm on the specific parallel machine at hand. When periodic matrix remapping and process reduction are applied without care, the extra communication steps induced by these optimization strategies may prove too expensive.

With respect to the speedup graph of Figure 4 we need to state that the three graphs for runs including process reduction are given for the *initial* number of compute nodes, without identifying subsequent process reductions. This means that one can not simply calculate efficiency figures by dividing the obtained speedup by the indicated number of CPUs. Such a figure would merely provide a lower bound on the obtained efficiency. One way to solve this problem is to normalize our results for the actual number of nodes used in each execution phase. Our results show that we obtain a normalize efficiency for a 64-node run of 52.4% for our model-based solution, while the non-adaptive execution strategy obtains only 36.4%. Hence, our model-based approach provides an efficiency improvement of more than 44%. Moreover, the power of our model-based approach is shown by the fact that, for a smaller number of initial nodes (in this case up to 24 nodes), close-to-linear speedup is obtained. Our results for different matrix sizes, and for different sizes of the parallel system, give a similar picture.

## 6 Conclusions

In this paper we have presented an adaptive parallel execution strategy for Householder bidiagonalization — an important algorithm in (a.o.) multimedia content analysis. The algorithm was implemented by using (as much as possible) the basic building blocks for data parallel processing available in the Parallel-Horus parallel multimedia computing library.

The Householder algorithm is hard to parallelize efficiently, as it suffers from a gradual decrease in the number of matrix elements used in the calculations. To overcome the negative performance impact of load imbalances and overprovisioning of compute resources, we have applied adaptive runtime techniques of *periodic matrix remapping* and *process reduction* for improved performance. Our results show that the combination of periodic matrix remapping and process reduction can improve performance, but only if the runtime characteristics of the algorithm are taken into account. We have shown that our model-based adaptive parallel execution approach can improve the obtained efficiency of a non-adaptive execution strategy by 44% or more. Moreover, in contrast to other strategies, our model-based approach obtains close-to-linear speedups when excessive overprovisioning of initial compute nodes is avoided.

## References

1. Seinstra, F., et al.: High-Performance Distributed Video Content Analysis with Parallel-Horus. IEEE Multimedia 14(4), 64–75 (2007)
2. Chu, E., George, J.: Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor. Parallel Computing 5(1), 65–74 (1987)
3. Dongarra, J., Sorensen, D.: A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem. SIAM J. Sci. Stat. Comput. 8(2), 139–154 (1987)
4. Egecioglu, O., et al.: Givens and Householder Reductions for Linear Least Squares on a Cluster of Workstations. In: Proc. HiPC 1995, December 1995, pp. 734–739 (1995)
5. Groser, B., Lang, B.: Efficient Parallel Reduction to Bidiagonal Form. Parallel Computing 25(8), 969–986 (1999)
6. Rabani, E., Toledo, S.: Out-of-core SVD and QR Decompositions. In: Proceedings of PPSC 2001, Portsmouth, USA (March 2001)
7. Koelma, D., et al.: Horus C++ Reference, v. 1.1. Technical report, University of Amsterdam, The Netherlands (January 2002)
8. Morrow, P., et al.: Efficient Implementation of a Portable Parallel Programming Model for Image Processing. Concur. Pract. Exp. 11, 671–685 (1999)
9. Seinstra, F., et al.: User Transparent Parallel Processing of the 2004 NIST TRECVID Data Set. In: Proceedings of IPDPS 2005, Denver, USA (April 2005)

10. Snoek, C., et al.: The Semantic Pathfinder: Using an Authoring Metaphor for Generic Multimedia Indexing. IEEE Trans. Pat. Anal. Mach. Intel. 28(10), 1678–1689 (2006)
11. Snoek, C., et al.: MediaMill: Exploring News Video Archives based on Learned Semantics. In: ACM Multimedia 2005, Singapore (November 2005)
12. Geusebroek, J., Seinstra, F.: Color-Based Object Recognition by a Grid-Connected Robot Dog. In: AAAI 2007 - Video Competition, Vancouver, Canada (July 2007)
13. Seinstra, F., et al.: Scalable Wall-Socket Multimedia Grid Computing, First IEEE International Scalable Computing Challenge, First Prize Winner (May 2008)
14. Golub, G., Reinsch, C.: Singular Value Decomposition and Least Squares Solutions. Numerische Mathematik 14(5), 403–420 (1970)
15. Evans, D., Gusev, M.: Systolic SVD and QR Decomposition by Householder Reflections. Int. J. Comp. Math. 79(4), 417–439 (2002)
16. Seinstra, F., et al.: Finite State Machine-Based Optimization of Data Parallel Regular Domain Problems Applied in Low-Level Image Processing. IEEE Transactions on Parallel and Distributed Systems 15(10), 865–877 (2004)
17. Seinstra, F., et al.: A Software Architecture for User Transparent Parallel Image Processing. Parallel Computing 28(7-8), 967–993 (2002)