Petr Kotas; Vít Vondrák; Pavel Praks

Parallel SVD computation

In: Jan Chleboun and Petr Přikryl and Karel Segeth and Jakub Šístek (eds.): Programs and Algorithms of Numerical Mathematics, Proceedings of Seminar. Dolní Maxov, June 6-11, 2010. Institute of Mathematics AS CR, Prague, 2010. pp. 113–118.

Persistent URL: http://dml.cz/dmlcz/702748

PARALLEL SVD COMPUTATION*

Petr Kotas, Vít Vondrák, Pavel Praks

1 Introduction

The aim of this paper is to present experiments with parallel implementation of large scale singular value decomposition (SVD). The SVD has remarkable properties and it is widely used as a tool in matrix computations. However, there are problems with enormous computational demands of SVD. Recently there are many new SVD applications in the computational science. Just for an illustration we mention eigenfaces [5], which is probably one of the earliest computationally demanding application of the eigenvalue analysis applied to large data sets. Another widely used application is the Latent Semantic Indexing (LSI) [6]. LSI is used in data-mining and information retrieval communities for reducing dimension of a problem and for uncovering so called latent semantic, which is hidden in analyzed data.

In this paper we present a parallel implementation of bidiagonalization routine. Our main goal is to solve SVD for large matrices which cannot fit into the memory of standard PC and speed-up current algorithms porting them on massively parallel computers. We have implemented our version of parallel SVD algorithm in C++ programming language using Message Passing Interface (MPI). This allows us to utilize distributed resources and to load even huge data directly to the computer memory. Although other parallel implementations exist, many of them utilize multicore architectures to gain more speed-up with the same amount of local memory [8].

This paper is organized as follows: In Section 2 we present our parallel implementation of the bidiagonalization algorithm. Furthermore, in Section 3 we present efficiency of our algorithm on numerical experiments. Final comments and conclusions are presented in Section 4.

2 Computing SVD

The SVD computation consists of three consecutive steps: (i) bidiagonalization, (ii) computation of singular values and vectors, (iii) post-multiplication of results from previous two steps. In preprocessing stage the Householder bidiagonalization is used. This method utilizes the Householder reflection

$$H = I - 2vv^*, \tag{1}$$

*This work was supported by grant #GD103/09/H078 of Grant Agency of the Czech Republic and grant #SP/2010173 of Students Grant Competition VSB-Technical University of Ostrava.

Algorithm 1 Parallel bidiagonalization

1: **Input:**A distributed to all nodes
2: **Output:**B, bidiagonal matrix
3: $[m, n] \leftarrow$ `size`(A)
4: **for** $k = 1$ **to** $min(m, n)$ **do**
5: activeColumn $\leftarrow$ `allGather`$(A_{loc}(:, k))$
6: $v \leftarrow$ `householder`(activeColumn)
7: **for** $j = 1$ **to** n **do**
8: $\gamma_{loc}(j) = v^T A_{loc}(:, j)$
9: **end for**
10: $\gamma \leftarrow$ `allReduce`(γ_{loc})
11: **for** $j = k$ **to** n **do**
12: $A_{loc}(:, j) \leftarrow A_{loc}(:, j) - 2\frac{\gamma(j)}{v^T v}v$
13: **end for**
14: **if** $k < (n - 2)$ **then**
15: **if** node has $A_{loc}(k, :)$ **then**
16: `broadcast`$(A_{loc}(k, :))$
17: activeRow $\leftarrow A_{loc}(k, :)$
18: **else**
19: activeRow $\leftarrow$`receive`$(A_{loc}(k, :))$
20: **end if**
21: $v \leftarrow$`householder`(activeRow)
22: **for** $i = k$ **to** m **do**
23: $\gamma \leftarrow A_{loc}(i, :)v$
24: $A_{loc}(i, :) \leftarrow A_{loc}(i, :) - 2\frac{\gamma}{v^T v}v$
25: **end for**
26: **end if**
27: **end for**
28: $B \leftarrow A$

where $v = x \pm \|x\|_2 e, v \in \mathbb{R}^n$ is the Householder vector. For further details on bidiagonalization see [1]. In Algorithm 1 we propose our parallel version of basic bidiagonalization routine defined in [1]. This is optimized version without implicit accumulation of orthogonal transformation matrices. Function `householder` in Algorithm 1 denotes the standard Householder reflector as in [1]. The singular values of bidiagonal matrix computed by Algorithm 1 are the same as the singular values of the original matrix A. This is obvious fact since the Householder reflection preserves the orthogonality among singular vectors, as has been proved in [1].

In the second step a diagonalization of bidiagonal matrix B computed in first step is performed. The resulting diagonal matrix consists only from singular values of the bidiagonal matrix B. Our diagonalizatin routine uses sequential implicit QR algorithm as it is described in [2], and can be theoretically implemented for

massively parallel computers. At this time, we use the LAPACK[1] sequential function _BDSDC, which computes singular values of real bidiagonal matrix B. Therefore, the diagonalization part represents the bottleneck of our algorithm.

In the third step of SVD, the Householder matrices (U_H and V_H), the matrix of singular values (Σ) as well as singular vectors of the bidiagonal matrix (U_B and V_B) are assembled. To complete the whole decomposition, one only needs to multiply

$$\begin{aligned} U &= U_H \cdot U_B, \\ V &= V_B \cdot V_H. \end{aligned}$$

After this final step the full SVD decomposition of an arbitrary real matrix A is obtained.

3 Numerical experiments

The overall execution time of T_p Algorithm 1 is given by the following equation

$$T_p = t_c \frac{n^3}{3p} + 2t_s n + t_w \frac{n^2}{\sqrt{p}}, \tag{2}$$

where t_c is time needed for computing one FLOP[2], t_s and t_w denote both send and wait latencies of MPI, $n = \max(rows, cols)$ is dimension of the matrix A and p is the number of processors.

All experiments presented in this section are computed on cluster Teri with hardware configuration: 32x Intel Xeon QuadCore 2.5 GHz, 18 GB RAM, 4XDDR IB Mezzanine HCA 20Gb/s FullDuplex (per node). All experiments were performed on dense random matrices with sizes: 8×8, 16×16, 32×32, ..., 4096×4096.

Figure 1 compares theoretical time computed by (2) with real execution times of our implementation of Algorithm 1. Times t_s, t_w and t_c are estimated from measurements performed on cluster Teri. However, latency times t_s and t_w are heavily dependent on current load of computational cluster, which means execution time T_p could vary. Significant increase of execution times for 64 and 128 processors is caused by raising rate of MPI communication. This problem could be solved with more suitable decomposition scheme.

In order to measure the performance of our implementation of bidiagonalization with fixed number of processors, we carried out several series of tests with varying size of the problem. Figure 2 shows total bidiagonalization time (including MPI communication) and MPI communication itself.

Similar tests were run to show behavior of our algorithm with increasing number of processors.

[1] LAPACK - Linear Algebra PACKage http://www.netlib.org/lapack/

[2] FLOP is abbreviation for floating point operation.

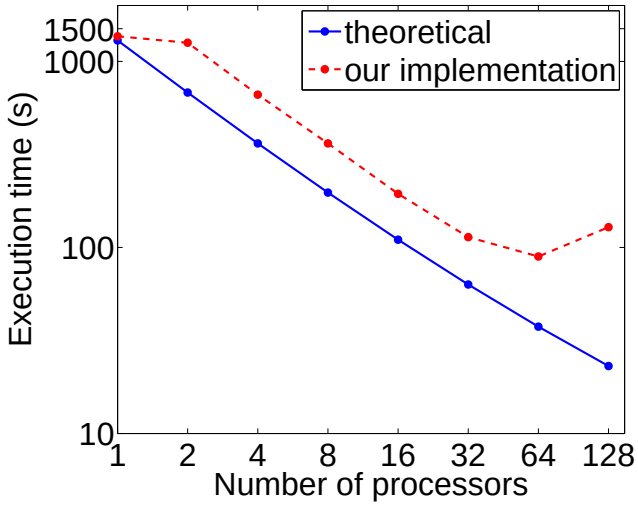
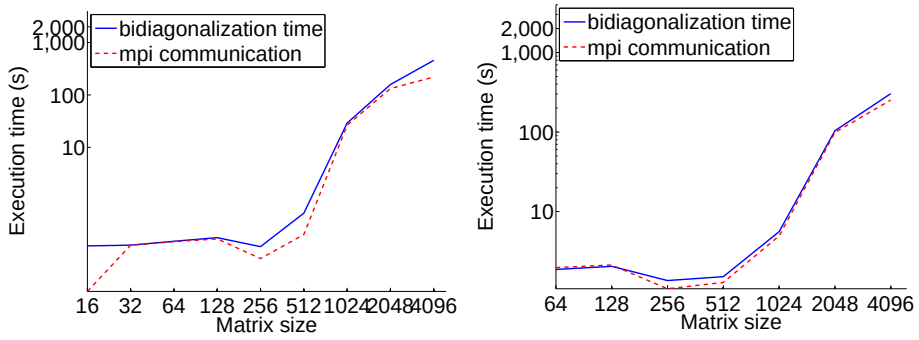


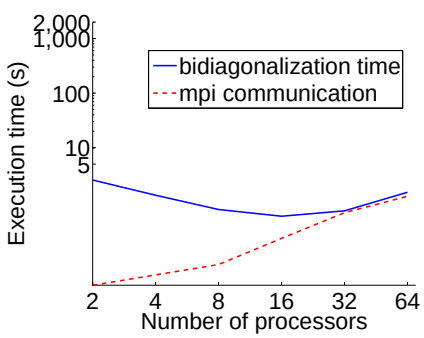
Fig. 1: *Bidiagonalization of the 4096 × 4096 matrix.*



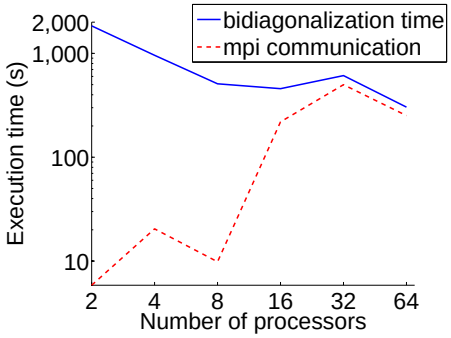
(a) Processors = 32

(b) Processors = 64

Fig. 2: *Fixed number of processors.*



(a) Matrix size = 512



(b) Matrix size = 4096

Fig. 3: *Fixed matrix size.*

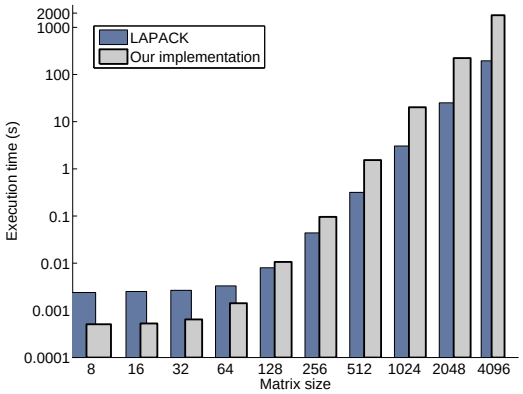


Fig. 4: *Sequential version of bidiagonalization compared to* `LAPACK`.

Figure 4 plots the execution times taken by bidiagonalization routine DGBERD from `LAPACK/ATLAS` library and our sequential version of Algorithm 1, respectively. We can see that both implementations have similar time evolution. But finally, the `LAPACK` sequential implementation seems to be faster because it uses optimized `BLAS` libraries.

The main advantage of our algorithm is its ability to process even large-scale data. We tried to decompose some very large problems. The largest matrix decomposed by our algorithm had dimension 32768×32768, which required $8.1GB$ of memory. We used 32 processors and our algorithm had been running for 32.32 hours. The MPI communication required 1.79 hours.

4 Comments and conclusions

The advantage of our implementation is effective handling of large dense problems. On the other hand, it seems that our algorithm is less effective in term of parallel scalability for more than 32 processors. This problem could be solved by more sophisticated decomposition scheme, which is left for further research. Further, improvements could be done utilizing the parallel implementation of the diagonalization routine and by using both MPI and OpenMP libraries. These improvements could lead to a significant speed-up, especially for large tasks running on large numbers of processors.

References

[1] Golub, G.H. and Van Loan, Ch.F.: *Matrix computations*. The Johns Hopkins University Press; 3rd edition, 1998.

[2] Kotas, P.: *Efficient implementation of SVD and its application to biometric data processing*. VŠB - Diploma thesis, 2009.

[3] Gu, M. and Eisenstat, S.C.: A divide-and-conquer algorithm for the bidiagonal SVD. SIAM J. Mat. Anal. Appl. **16** (1995), 79–92.

[4] Jessup, E. and Sorensen, D.: A parallel algorithm for computing the singular value decomposition of a matrix. Mathematics and Computer Science Division Report ANL/MCS-TM-102, Argonne National Laboratory, Argonne, IL, December 1987.

[5] Turk, M. and Pentland, A.: Face recognition using eigenfaces. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, pp. 586–591, 1991.

[6] Letsche, T.A. and Berry, M.W.: Large-scale information retrieval with latent semantic indexing. Information Sciences **100** 1-4 (August 1997), 105–137.

[7] Larsen, R.M.: Lanczos bidiagonalization with partial reorthogonalization. Part of documentation to software package PROPACK, 1998.

[8] Ltaief, H., Kurzak, J., and Dongarra, J.: Parallel two-sided matrix reduction to band bidiagonal form on multicore architectures. IEEE Transactions on Parallel and Distributed Systems **99** (2009), 417–423.