AN ANALYSIS OF PARALLEL IMPLEMENTATIONS OF THE BLOCK-JACOBI ALGORITHM FOR COMPUTING THE SVD

PETER ARBENZ
Institute for Scientific Computing
Swiss Federal Institute of Technology (ETH)
8092 Zurich, Switzerland
arbenz@inf.ethz.ch
and
IVAN SLAPNIČAR[1]
Faculty of Electrical Engineering, Mechanical
Engineering and Naval Architecture
University of Split
R. Boškovića b.b.
58000 Split, Croatia
ivan.slapnicar@fesb.hr

Abstract

We analyze message passing communication model of distributed memory computers and show that, by using such model, we can implement one-sided block-Jacobi method for computing the singular value decomposition in a highly scalable manner.

Key words: distributed memory, singular values, block algorithms, Jacobi method

1 Introduction

In this paper we first analyze the requirements to a communication environment in order to make scalable implementations of algorithms in numerical linear algebra possible. As a typical example of such an algorithm, we then investigate the behavior of the block-Jacobi algorithm for computing the singular value decomposition (SVD) on parallel multicomputers [2], i.e. multiprocessor computers with distributed memory supporting the message passing programming model. We introduce the principal issues in Section 2 by means of the easily understood matrix-matrix multiplication. In Section 3 we apply the same consideration to the Jacobi algorithm. To verify the theory, numerical experiments have been performed on the Intel Paragon using the NX message passing interface [7]. In Section 4 we draw our conclusions.

2 Matrix multiplication

In this section we introduce the principal ideas by means of matrix-matrix multiplication. Let $A, B, C \in \mathbb{R}^{n\times n}$ be real n-by-n matrices. Given A and B, we want to compute the matrix product

$$C = A \cdot B$$

on a p processor multicomputer. We consider the processors, numbered from 0 to $p-1$, to be *logically* arranged in a ring. This means that each processor communicates exactly with two other processors. Processor i, say, can exchange messages with processor $(i-1) \bmod p$ and with $(i+1) \bmod p$. We call these neighbor processors the *previous* and *next* processors. For simplicity we assume that p divides n, $n = mp$.

We split the matrix A in the form

$$A = \begin{bmatrix} A_0 \\ \vdots \\ A_{p-1} \end{bmatrix}, \qquad A_i = [A_{i,0}, \ldots, A_{i,p-1}] \in \mathbb{R}^{m\times n}, \qquad A_{i,j} \in \mathbb{R}^{m\times m}, \qquad 0 \le i, j < p,$$

[1]Part of this work was done during this author's visit to the ETH Zurich. This author also acknowledges the Grant No. 1-01-252 from the Croatian Ministry of Science.

and likewise B and C. In our implementations we distribute the matrices in this block-row wise fashion. Initially, the ith block row of A, B, and C reside on processor i. With this storage scheme it is natural that processor i computes the block-row C_i according to the formula

$$C_i = A_i B = \sum_{j=0}^{p-1} A_{i,j} B_j, \qquad 0 \le i < p. \tag{1}$$

Notice that we could compute C_i without communication if we stored all of B on every processor. This storage scheme would however not be scalable. While A_i and C_i stay in the memory of the i-th processor, the rows of B move around the processor ring. A SPMD program is given in Algorithm 1.

Algorithm 1 (Synchronous matrix multiply) *Processor i computes C_i. After execution, the block-rows A_i, B_i, and C_i reside at their original places.*

$C_i \leftarrow 0_{m,n}$.
for $j = 0, 1, \ldots, p-1$
  $C_i \leftarrow C_i + A_{i,i-j \bmod p} B_{i-j \bmod p}$.
  Send $B_{i-j \bmod p}$ to next processor.
  Receive $B_{i-(j-1) \bmod p}$ from previous processor.
end for

During the round-trip of the B_i each processor computes its part of C. C_i is computed in p steps according to (1). Each step costs $2nm^2$ floating point operations. After the update of C_i, the local block of B is forwarded to the next processor and replaced by the block of the previous processor. In order to avoid deadlocks a message passing buffer of at least size $2nm$ doubles, i.e. twice the message length, has to be provided. We model the cost of the transmission of a message of length n doubles (8 bytes) by $\sigma + \tau n$. σ denotes the startup time in μsec while $1/\tau$ is the bandwidth in doubles per μsec. If the execution of one flop takes $\varphi(n, m)$ μsec, then the total cost of Algorithm 1 is

$$C_1(n, p) = p\,(nm(2m-1)\varphi(n, m) + 2\sigma + 2\tau nm)\,. \tag{2}$$

Speedup and efficiency are given by [5, §3.1]

$$S_1(n, p) = C_1(n, 1)/C_1(n, p), \qquad E_1(n, p) = S_1(n, p)/p. \tag{3}$$

On the Intel Paragon, $\sigma = 65\mu$sec and $\tau = 1/8\mu$sec, corresponding to a bandwidth of 65Mbyte/s = 65byte/μsec. Due to the complicated pipeline architecture in a RISC processor it is difficult to assign a value to φ. The Paragon, e.g., has a peak performance of 50Mflop/s [4]. The measured performances ranges from 10Mflop/s for the Linpack benchmark [4] up to 46Mflop/s for the BLAS-3 routine `dgemm` which incorporates matrix multiplication. Since we use `dgemm` as a major computational routine to perform a $(n \times m) \times (m \times m)$ multiplication, we have measured its performance and defined φ as the function of the matrix sizes. For $n = 512$ the Mflop/s rates are given in Table 1. The Mflop/s rates for larger values of n are similar. The values of $\varphi(n, m)$ are the inverses of the corresponding Mflop/s rates.

Table 1: Mflop/s rate for `dgemm` for $n = 512$

$m \equiv n/p$	1	2	4	8	16	32	64	128	256	512
Mflop/s	11.7	14.8	26.6	34.5	40.5	43.0	44.7	45.3	45.8	45.7

Modern massive parallel computers have hardware to relieve the CPU from dealing with the overhead involved with message passing. The Intel Paragon, e.g., has a second Intel i860 RISC processor called message processor for that purpose. On such computers it is possible to *overlap* computation and communication by issuing *asynchronous* (non-blocking) send and receive primitives. A call to an asynchronous send or receive function initiates some action of the message processor and then returns immediately. So the compute processor takes only the startup time; the actual transmission is handled by the message processor. In the asynchronous version of Algorithm 1, B is being prepared for communication at the same time the compute processor updates C. In order not to send too early, processors are synchronized before the matrices are communicated which requires two additional messages (of zero length). This enables the message processor to store the arriving matrix directly in user space, thus avoiding a time consuming intermediate buffering.

With the notations as in (2) the total cost of the asynchronous algorithm becomes

$$C_2 = p \max\{nm(2m-1)\varphi(n,m) + 4\sigma, 2\tau nm\}. \tag{4}$$

In Figure 1 theoretical and measured speedup and efficiency of Algorithm 1 and its asynchronous version are depicted. The *theoretical* considerations show a big advantage of asynchronous over

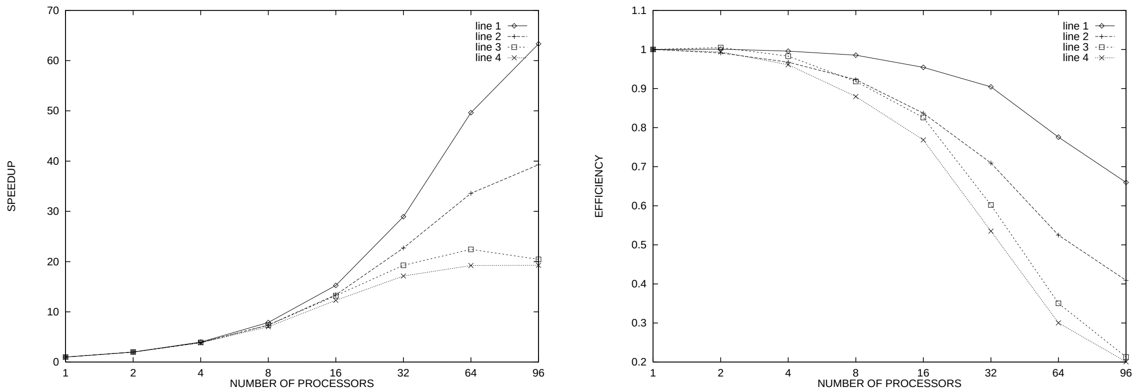

Figure 1: Speedup and efficiency of theoretical asynchronous (line 1) and synchronous (line 2), and measured asynchronous (line 3) and synchronous (line 4) for $n = 576$.

synchronous version of Algorithm 1 in the range of a few to a medium number of processors. In this range, communication hiding makes possible close to optimal speedups. The theoretical gain in speedup and efficiency through communication hiding can be up to 2. The gain is highest if computation and communication last equally long [8]. For high numbers of processors the communication overhead starts to dominate and the speedup curve starts to decrease; the problem is then getting too small for the number of processors used.

Practical timings for larger numbers of processors are worse than predicted. The overhead due to message passing is larger than our simple timing model suggests. We suspect that the NX message passing interface, in particular the asynchronous send/receive primitives, are not optimally implemented. Furthermore, the synchronization of the memory accesses of both compute and message processor working at full speed may cause problems.

The timings are better for larger dimensions, as we shall see in next section. For orientation, the 576×576 matrix multiplication takes 8.3 seconds on one processor, and 0.677\0.630 seconds on 16 processors with the synchronous\asynchronous algorithm.

3 The Jacobi algorithm

The SVD of a non-singular $n \times n$ matrix A is written as

$$A = U\Sigma V^T, \tag{5}$$

where U and V are orthogonal matrices, and Σ is a diagonal matrix whose diagonal entries are the singular values of A. One-sided (implicit) Jacobi method [3] computes the singular values, and the corresponding left singular vectors (the matrix U), by forming a sequence of matrices

$$A_0 = A, \qquad A_{k+1} = A_k R_k, \tag{6}$$

where R_k is a plane rotation, i.e. the identity matrix except for the elements $[R_k]_{ii} = [R_k]_{jj} = \cos\phi$ and $[R_k]_{ij} = -[R_k]_{ji} = \sin\phi$, where $i < j$. The angle ϕ is chosen so that the element (i, j) of the implicitly defined Gram matrix $G_{k+1} \equiv A_{k+1}^T A_{k+1} = R_k^T A_k^T A_k R_k$ is annihilated,

$$[G_{k+1}]_{ij} = [G_{k+1}]_{ji} = 0.$$

To determine ϕ we need to know $[G_k]_{ii}$, $[G_k]_{jj}$, and $[G_k]_{ij} = [A_k]_{:i}^T [A_k]_{:j}$. The first two quantities are kept and updated in a separate vector, so we just need to form one scalar product. The sequence (6) converges to the matrix $\bar{A}$ with orthogonal columns. The norms of the columns of $\bar{A}$ are the singular values of A, and the normalized columns of $\bar{A}$ are the corresponding left singular vectors,

$$\Sigma_{ii} = \|\bar{A}_{:i}\|_2, \qquad U = \bar{A}[\mathrm{diag}(\|\bar{A}_{:i}\|_2)]^{-1} = \bar{A}\Sigma^{-1}.$$

Thus, the initial matrix is overwritten by the singular vectors, which halves the necessary storage. We use the relative stopping criterion [3]: the rotation is performed only if

$$|[G_k]_{ij}| > tol \cdot \sqrt{[G_k]_{ii}}\sqrt{[G_k]_{jj}}. \tag{7}$$

where tol is a user parameter, usually $tol = n\epsilon$, and ϵ is the machine precision. The algorithm stops after $n(n-1)/2$ successive rotations (one sweep) have been skipped. This criterion implies the high relative accuracy of the method (6) [3]. The algorithm for an $m \times n$, $m \geq n$ matrix is similar, but in this case only the first n columns of U are computed. If A is singular, then the algorithm computes only those columns of U which correspond to non-zero singular values.

Since the rotations on non-overlapping index pairs are completely independent, they can be applied simultaneously. This makes the one-sided Jacobi method ideal for parallel computation. Modern processors with pipelining features prefer block algorithms where the major computational effort is performed with matrix multiplications (BLAS-3). In order to exploit this property, we use a block-Jacobi method similar to the one used in [1] for shared memory machines: we partition A into column blocks

$$A = \begin{bmatrix} A^{(1)} & A^{(2)} & \cdots & A^{(2p)} \end{bmatrix} \tag{8}$$

and assign two consecutive blocks to each processor. Assume for simplicity that all blocks have the same number of columns m, $n = 2mp$. Denote the two blocks held by some processor by A_L and A_R. The idea is that each processor forms the Gram matrix of its block columns,

$$G \equiv \begin{bmatrix} G_{11} & G_{12} \\ G_{12}^T & G_{22} \end{bmatrix} = \begin{bmatrix} A_L^T A_L & A_L^T A_R \\ A_R^T A_L & A_R^T A_R \end{bmatrix},$$

then applies one step of the standard two-sided Jacobi method [9] to G accumulating the rotations in the matrix R, and, finally multiplies $\begin{bmatrix} A_L & A_R \end{bmatrix} R$. The last step contains the major part of the computation and can in this manner be performed by the BLAS-3 routine `dgemm`. After computation, the blocks A_R circulate along the processor ring according to the caterpillar parallel strategy [6], so that each block of the global Gram matrix $A^T A$ is accessed exactly once in each sweep. This strategy also restores the original layout (8) after every 2nd sweep ($2p$ stages). The algorithm stops when all processors performed no rotations in two-sided Jacobi for more than one sweep. Note that only upper triangles of G_{11} and G_{22} are needed, since we can apply two-sided Jacobi only on the upper triangle of the symmetric G. The algorithm is given in Algorithm 2.

Since the stopping criterion (7) is at the same time a threshold, Algorithm 2 always converges (see [9, p. 278]). The non-blocked algorithm (6) computes the singular values with high relative accuracy [3]. Numerical tests show that Algorithm 2 computes the singular values with the same high relative accuracy, although we have not yet a formal proof.

Let us compute the cost. For larger dimensions ($n = 1000, 2000$), the algorithm takes about 8 double sweeps to converge. The non-blocked algorithm requires $8n(n-1)(2n-1+6n) \approx 64n^3$ operations. However, this algorithm uses only scalar products and $(m \times 2) \times (2 \times 2)$ multiplications, so from Table 1 we see that the utilization of processors would be rather poor. The cost for Algorithm 2 with synchronous communication is

$$\begin{aligned} C_3(n,p) \;=\; & (2m^2+m)(2n-1)\varphi(n,m) + 8 \cdot 4p\left[6(2m)^3\varphi(m,2) + n(2m)(4m-1)\varphi(n,2m)\right. \\ & \left.+m^2(2n-1)\varphi(n,m) + 4\sigma + 2\tau(m(m+1)/2) + 2\tau nm\right]. \end{aligned} \tag{9}$$

Algorithm 2 (Block-Jacobi method) *Processor i executes the following program:*

```
G_11 ← A_L^T A_L, G_22 ← A_R^T A_R, G_12 ← A_L^T A_R.
repeat until convergence
  for j = 1, ..., 4p
    if (j − 1) mod 2p + 1 ≠ 2p − 2(i − 1) then
      apply one sweep of Jacobi to G and compute R.
      [ A_L  A_R ] ← [ A_L  A_R ] R.
      Send A_R and G_22 to next processor.
      Receive A_R and G_22 from previous processor.
      G_12 ← A_L^T A_R.
    else
      Swap A_L with A_R, and G_11 with G_22.
      Send A_R and G_22 to next processor.
      Receive A_R and G_22 from previous processor.
      G_12 ← A_L^T A_R.
    end if
  end for
end repeat
```

The speedup, $S_3(n,p)$, and efficiency, $E_3(n,p)$, are obtained according to (3). The cost of the asynchronous version is obtained similarly as in (4).

In Figure 2 theoretical and measured speedup and efficiency of Algorithm 2 and its asynchronous version are depicted. For orientation, the asynchronous version takes 630, 342, and 300 seconds

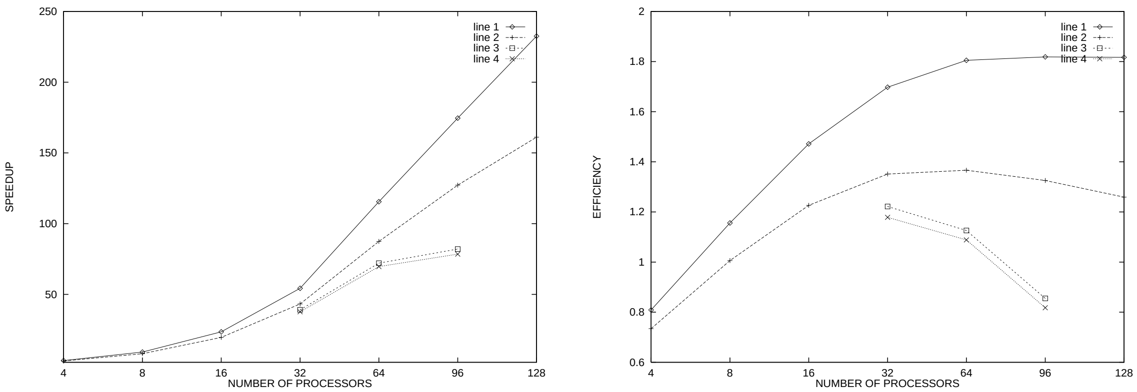

Figure 2: Speedup and efficiency of theoretical asynchronous (line 1) and synchronous (line 2), and measured asynchronous (line 3) and synchronous (line 4) for $n = 2048$.

for $p = 32, 64, 96$, respectively.

The superlinear speedups seen in Figure 2 can be explained as follows: the serial complexity of Algorithm 2, i.e. the cost of operations that each processor performs in each stage and also in each sweep, decreases more than linearly as p increases. More precisely, by inserting $m = n/(2p)$ into the part of (9) associated with the iterative part of Algorithm 2, we see that the operation count is $32n^3(6/(p^2)+2.5/p)$. The quantities $\varphi(n,2m)$ and $\varphi(n,m)$ decrease by at most the factor 2 as p increases (see Table 1), which altogether results in the superlinear decrease of the cost. This confirms the statement that the block-Jacobi method is ideally suited for parallel computation. When the number of processors further increases, the communication cost starts to dominate, and the efficiency decreases again.

We see that the block-Jacobi method has even better theoretical properties than matrix multiplication, since it allows speedups close to optimal. Also, the synchronous and asynchronous version differ theoretically as much as in matrix multiplication. Our implementations on Intel Paragon behave similarly as predicted for $p = 32, 64$; the difference is due to the fact that actual implementations have some extra operations not counted in (9), like computing the Jacobi rotation parameters. Such scalar operations are relatively slow on RISC architectures. Further, note that, as in matrix multiplication, the synchronous and asynchronous implementations differ only by little. Here send and receive also need some extra copying to/from buffer, and the memory copy takes approximately the same amount of time as inter-processor communication (without the startup time). For $p = 96$ our implementations show a relative slowdown, which is due to the reasons explained in Section 2.

Algorithm 2 can also be applied to compute the spectral decomposition of a symmetric positive definite matrix – eigenvalues are squared singular values, and eigenvectors are left singular vectors. If the matrix is indefinite, then we get absolute values of the eigenvalues and the corresponding eigenvectors, and a small extra effort is needed to determine the signs of the eigenvalues.

4 Conclusions

We have shown by a simple model for synchronous and asynchronous message passing that the overlapping of communication and computation can help to increase the scalability of algorithms. The concrete applications presented are matrix multiplication and one-sided block-Jacobi algorithm. Although the measurements we executed on the Intel Paragon did not precisely reflect the theoretical considerations, they confirm the fact that considerable speedup and efficiency increases can be obtained by communication latency hiding. This technique makes it possible to increase the efficiency with which an application runs on a certain number of processors, or, exploit a higher number of processors given a desired efficiency. The block-Jacobi method attains nearly optimal speedups, which makes it very suitable for parallel computations, even though the method, as implemented, makes no great use of communication latency hiding.

References

[1] P. Arbenz and M. Oettli (1992), Block implementations of the symmetric QR and Jacobi algorithms, Institute for Scientific Computing Technical Report 178, ETH, Zürich.

[2] G. Bell (1992), "Ultracomputers: A teraflop before its time", *Communications of the ACM*, Vol. 35, pp. 27–47.

[3] J. W. Demmel and K. Veselić (1992), "Jacobi's method is more accurate than QR", SIAM J. Matrix Anal. Appl., Vol. 13, pp. 1204–1243.

[4] J. J. Dongarra (1995), Performance of various computers using standard linear equation software, Tech. Rep. CS-89-85, Univ. of Tennessee, Computer Science Dept., Knoxville, TN.

[5] K. Hwang (1993), Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw–Hill, New York.

[6] F. T. Luk and H. Park (1989), "On parallel Jacobi orderings", SIAM J. Sci. Statist. Comput., Vol. 10, pp. 18–26.

[7] P. Pierce (1994), "The NX message passing system", Parallel Computing, Vol. 20, pp. 463–480.

[8] V. Strumpen and T. L. Casavant (1994), "Exploiting communication latency hiding for parallel network computing: Model and analysis", Proc. of the 1994 Int. Conf. on Parallel and Distributed Systems, Los Alamitos, CA, IEEE Computer Society Press, pp. 622–627.

[9] J. H. Wilkinson (1965), The Algebraic Eigenvalue Problem, Oxford University Press.