International Journal of Parallel Programming, Vol. 34, No. 6, December 2006 (© 2006)
DOI: 10.1007/s10766-006-0025-y

Distributed Jacobi Joint Diagonalization on Clusters of Personal Computers

Aleš Holobar,[1,2] Milan Ojsteršek,[1] and Damjan Zazula[1]

Received February 8, 2006; accepted October 10, 2006

A new algorithm is described for distributed joint diagonalization of real symmetric or complex Hermitian matrices. The approach, which is based on the Jacobi diagonalization, utilizes distribution of the computational power and memory space, minimizes the communication costs, and runs on clusters of personal computers. It further combines two-step load balancing algorithm with a standard Kalman filter to enable quick but low-cost adaptation to resource varying conditions. Theoretical analysis of its performance shows that the communication costs (when normalized by computational costs) decline linearly with the number and size of the diagonalized matrices. This is also confirmed by experimental results: the measured speedup ratio yields 42.2 when jointly diagonalizing 800 matrices of size 400×400 on a cluster of 50 personal computers.

KEY WORDS: joint diagonalization; distributed Jacobi; distributed load balancing; diffusion schemes; parallel computing.

1. INTRODUCTION

Joint diagonalization of Hermitian matrices (also known as approximate joint diagonalization or simultaneous diagonalization) consists of finding a unique transformation which simultaneously transforms the matrices into as close to diagonal as possible. This problem has been studied extensively

[1]Faculty of Electrical Engineering and Computer Science, Smetanova 17, 2000 Maribor, Slovenia
[2]To whom correspondence should be addressed. E-mail: ales.holobar@uni-mb.si

over the last decade and has provided solutions to a wide range of scientific and engineering problems. The joint parameter-estimation techniques which can be reduced to the problem of joint diagonalization cover, for example, the removal of artifacts and decomposition of compound signals in the area of biomedical signal processing, direction-of-arrival problem and crosstalk/interference elimination in the digital communication systems, identification of principal components in image restoration and understanding, separation and enhancement of audio and speech signals, and even theoretical investigation of the electronic structure of molecules, solids and liquids. Further insight on the applications of joint-diagonalization problem is given in Ref. 20.

A large variety of numerical methods for joint diagonalization have been developed utilizing Jacobi iterations,[4,9] theoretic matrix log-likelihood functions,[24] Gauss-Newton optimizations[21] and an alternating-directions algorithm.[26] They all consider the cases with only a few matrices to be diagonalized. On the other hand, applications of joint diagonalization to the problem of blind source separation (BSS) and identification of molecular structures attracted a lot of attention over the past decade, mainly due to their superior efficiency.[4,5,8,17,20] These techniques typically require a large number (at least several tens) of matrices to be processed, in order to suppress the negative influence of noise. Moreover, the matrices that enter joint diagonalization can be very large (of order of thousands), especially when separating convolutive mixtures of signals or when investigating the structures of large molecules. In a typical application of BSS techniques to the convolutive multiple-input–multiple-output (MIMO) mixing problem, for example, we deal with several hundreds of matrices consisting of several hundred rows/columns.[18,19] In all these cases, high-computational complexity of joint diagonalization becomes an important issue, as it is clear that the joint diagonalization soon meets the limitations of today's personal computers.

The problem of joint diagonalization is a natural extension of the symmetric eigenvalue problem.[15] The latter has been under investigation for more than a century, and many efficient solutions exist. From among them, the Jacobi method has maintained the best numerical stability[2,8] and superior accuracy,[12] but at the cost of a low convergence rate.[13] There are also many different parallel solutions to single-matrix diagonalization problem for various computational platforms.[3,6,11,15,22] Nevertheless, to our knowledge, only one description of a parallel algorithm exists for the joint diagonalization of several matrices. This approach, presented in Ref. 17, utilizes the ideas developed for the parallel diagonalization of a single matrix. As demonstrated in the sequel, it works well only for a small number of matrices being diagonalized on a multiprocessor (shared memory) system, where the communication costs are relatively small. When

jointly diagonalizing several tens of matrices on a distributed system these ideas lead to inefficient implementations with huge communication costs.

This paper introduces a novel approach to the distributed joint diagonalization of a large number of real symmetric or complex Hermitian matrices. The algorithm is based on the Jacobi diagonalization method[15] and can be run on the clusters of personal computers. It minimizes the communication costs, achieves a stable speedup factor, and a good load balance for all possible input matrices and numbers of processors used. The paper is organized as follows. Section 2 briefly reviews the Jacobi diagonalization of a single symmetric matrix, its parallel extension, and its extension to joint diagonalization. Our novel algorithm is introduced in Section 3, while Section 4 reveals its theoretical performance analysis. The experimental results obtained by the proposed method when applied to the clusters with different number of personal computers are presented in Section 5. We conclude the paper with a discussion in Section 6.

2. JOINT JACOBI DIAGONALIZATION

Define a set of K symmetric/Hermitian $N \times N$ matrices $\mathbf{A}_k$

$$\mathbf{\Theta} = \{\mathbf{A}_k;\ k = 1, ..., K\} \tag{1}$$

and suppose they all share a common eigen-structure, i.e. they can all be diagonalized by the same unitary matrix $\mathbf{U}$:

$$\mathbf{D}_k = \mathbf{U}^H \mathbf{A}_k \mathbf{U}, \tag{2}$$

where all $N \times N$ matrices $\mathbf{D}_k$ are diagonal and ^{H} denotes the complex conjugate transpose. The idea behind joint Jacobi method is to use successive orthogonal transformations $\mathbf{J}$ in order to reduce the norm of the off-diagonal elements[15]

$$off(\mathbf{\Theta}) = \sum_{k=1}^{K} \sqrt{\sum_{i=1}^{N} \sum_{j=1}^{N} \left| a_{ij}^{(k)} \right|^2}, \tag{3}$$

where $a_{ij}^{(k)}$ denotes the (i, j)th element of the matrix $\mathbf{A}_k$.Transformations $\mathbf{J}(p, q, c, s)$ are defined as Jacobi (Givens) rotation matrices equal to identity matrix but for the following entries:[15]

$$\begin{bmatrix} j_{pp} & j_{pq} \\ j_{qp} & j_{qq} \end{bmatrix} = \begin{bmatrix} c & \bar{s} \\ -s & \bar{c} \end{bmatrix}, \tag{4}$$

where p and q stand for row and column indices, respectively, (c, s) is a complex rotation cosine–sine pair with $|c|^2 + |s|^2 = 1$, and $\bar{c}$ and $\bar{s}$ denote the conjugate values of c and s, respectively.

For a given pair of indices (p, q), define a 3×3 matrix $\mathbf{G}$ as[10]

$$\mathbf{G}(p, q) = \text{real}\left(\sum_{k=1}^{K} \mathbf{h}(\mathbf{A}_k, p, q)^H \mathbf{h}(\mathbf{A}_k, p, q)\right), \tag{5}$$

where

$$\mathbf{h}(\mathbf{A}_k, p, q) = \left[a_{pp}^{(k)} - a_{qq}^{(k)}, a_{pq}^{(k)} + a_{qp}^{(k)}, i\left(a_{qp}^{(k)} - a_{pq}^{(k)}\right)\right]. \tag{6}$$

Then, under the constraint of $|c|^2 + |s|^2 = 1$, the value obtained by Eq. (3) is minimized when[10]

$$c = \sqrt{\frac{x + r}{2r}}, \qquad s = \frac{y - iz}{\sqrt{2r(x + r)}}, \qquad r = \sqrt{x^2 + y^2 + z^2}, \tag{7}$$

where $[x, y, z]^T$ is any eigenvector associated with the largest eigenvalue of $\mathbf{G}(p, q)$ and i stands for imaginary unit.

Initialize the matrix $\mathbf{U}$ equal to the $N \times N$ identity matrix ($\mathbf{U} = \mathbf{I}_{N \times N}$). The basic rotation of joint Jacobi diagonalization then comprises:[15]

1. choosing an index pair (p, q) where $1 \leqslant p < q \leqslant N$;
2. computing a complex cosine–sine pair (c, s) as suggested in (7);
3. replacing $\mathbf{A}_k$ with $\mathbf{B}_k = \mathbf{J}^H \mathbf{A}_k \mathbf{J}$ where $\mathbf{J} = \mathbf{J}(p, q, c, s)$;
4. replacing $\mathbf{U}$ with $\mathbf{U} = \mathbf{J}^H \mathbf{U}$.

Matrices $\mathbf{A}_k$ converge toward the diagonal matrix with each Jacobi step.[15] However, as only the pth and the qth columns and rows are altered in each step, several steps are required in order to generate all the possible (p, q) pairs, In the sequel, a cycle covering all possible (p, q) pairs, i.e. $N(N-1)/2$ Jacobi rotations that reduce all non-diagonal elements, will be referred to as a sweep.

3. DISTRIBUTED JOINT JACOBI DIAGONALIZATION

It is natural to implement joint Jacobi diagonalization as a distributed algorithm based on the ideas taken from the parallel Jacobi diagonalization of a single matrix. Each Jacobi transformation $\mathbf{J}(p, q, c, s)$ only changes the pth and qth row and column. By choosing pairs (p, q) wisely, several Jacobi rotations can be simultaneously applied to a set of matrices $\mathbf{A}_k$.

Recall, however, that all possible pairs of (p, q) should be generated in order to guarantee the convergence of Jacobi method. Orderings of indices which permit $N/2$ rotations to be carried out simultaneously in every step and walk through all possible combinations of indices are well known to the chess tournament players, and are commonly referred to as *tournament orderings*.[15] The first three steps of tournament ordering for $N = 8$ are exemplified in Eq. (8).

$$\begin{aligned} &\text{sweep step 1: } (1,2), (3,4), (5,6), (7,8), \\ &\text{sweep step 2: } (1,8), (2,3), (4,5), (6,7), \\ &\text{sweep step 3: } (1,7), (8,2), (3,4), (5,6). \end{aligned} \tag{8}$$

The aforementioned ideas for parallel joint diagonalization have already been outlined in Ref. 17. In the proposed solution, all the matrices $\mathbf{A}_k$ are thought of as having aligned rows and columns (they are placed one behind the other), whereas the index p is supposed to refer to the pth columns (rows) of all the matrices $\mathbf{A}_k$. Following the procedure for parallel Jacobi diagonalization of a single matrix,[15] the columns (rows) are distributed among the corresponding processing units (PUs). Each PU performs the Jacobi's basic step, sends the information about its rotations to all other PUs, receives the information about other rotations from all other PUs, and updates the columns of matrices $\mathbf{A}_k$. Afterwards, the indices are shifted according to the *tournament ordering*. Along with the indices, the corresponding columns of all the matrices $\mathbf{A}_k$ are also transferred. However, dealing with several hundred matrices, this transfer of columns creates heavy network traffic and slows down the computation, especially in heterogeneous distributed systems (see Section 4.1 for details).

The other possible way, as proposed in this paper, is to distribute the complete matrices among the PUs and let each PU diagonalize its own subset of matrices. In this case, no network transmission of matrix columns is required as each matrix subset is only assigned to a corresponding PU. However, to guarantee the generation of a universal transformation matrix $\mathbf{U}$ the rotations $\mathbf{J}(p, q, c, s)$ must be constructed globally, i.e. considering all the matrices $\mathbf{A}_k$. Rewrite (5) as

$$\begin{aligned} \mathbf{G}(p,q) = \text{real}&\left(\sum_{k=1}^{R_1(p,q)} \mathbf{h}(\mathbf{A}_k, p, q)^H \mathbf{h}(\mathbf{A}_k, p, q) \right) \\ &+\text{real}\left(\sum_{k=R_1(p,q)+1}^{R_1(p,q)+R_2(p,q)} \mathbf{h}(\mathbf{A}_k, p, q)^H \mathbf{h}(\mathbf{A}_k, p, q) \right) + \cdots + \end{aligned}$$

$$\text{real}\left(\sum_{k=K-R_Q(p,q)+1}^{K}\mathbf{h}(\mathbf{A}_k,\,p,\,q)^H\mathbf{h}(\mathbf{A}_k,\,p,\,q)\right)$$
$$=\mathbf{G}_1(p,q)+\mathbf{G}_2(p,q)+\cdots+\mathbf{G}_Q(p,q) \tag{9}$$

where Q stands for the number of PUs and each $R_i(p,q); i = 1,\ldots,Q$ with $R_1(p,q)+R_2(p,q)+\cdots+R_Q(p,q)=K$, denotes the number of matrices being diagonalized by the ith PU. Note that $R_i(p,q)$ may differ for different pairs of indices (p,q).This proves beneficial when it comes to load balancing, as described in Section 3.1.

The presented approach still utilizes the local operation property of Jacobi rotations. Following the *tournament ordering*, $N/2$ local matrices $\mathbf{G}_i(p,q)$ can be calculated on each PU prior to their global addition. However, the whole rotation cycle is now performed locally inside each PU (no column transfer among the PUs is required). Eigendecomposition of 3×3 $\mathbf{G}(p,q)$ matrix is a trivial task and can be performed by a single PU, as proposed in Fig. 1. Afterwards, the (c,s) pairs are propagated back to all PUs where local copies of the rotation matrix $\mathbf{J}(p,q,c,s)$ are stored. In this way, robustness to drop-outs of PUs is increased. The cosine–sine (c,s) pair computations can also be distributed among different PUs. In this case, all (p,q) index pairs must be first mapped onto Q predefined and globally known complementary sets that correspond to different PUs. Each PU then calculates global rotation angles (c,s) for its own set of (p,q) index pairs and dispatches them back to all other PUs.

3.1. Load Balancing

Running the described distributed algorithm on a cluster of personal computers with varying PU processing speeds and different connection capacities, it is essential to achieve efficient load balancing (LB) for all possible inputs and numbers of PUs used. Namely, additional waiting times arise as a consequence of the required global synchronizations (as described in Section 4.2). A smart dynamic distribution of matrices minimizes these waiting times.

In order to refine the granularity of LB, we should redistribute matrix columns rather than entire matrices. The matrices $\mathbf{A}_k$are first concatenated in a nose-to-tail circular order with the last column of the matrix $\mathbf{A}_k$ adjacent to the first column of the matrix $\mathbf{A}_{k+1}$(Fig. 2), while the last matrix $\mathbf{A}_K$ is made adjacent to the first matrix $\mathbf{A}_1$. The number of matrix columns P_i, processed by the ith PU, is no longer limited to an integer multiple of N, meaning that the columns of the same matrix may be diagonalized by two or even more neighbouring PUs. Referring to Fig. 2,

```
1:   until( the convergence condition is met )
2:       For all N/2 index pairs (p,q) calculate the local G_i(p,q) matrices.
3:       For all pairs of indices (p,q)
4:         If i=1
5:             Wait to receive all the local matrices G_i(p,q) and sum them up to
               form the global matrix G(p,q).
6:             Calculate the cosine-sine pairs (c,s) and send them to all other PUs
               (multicasting).
7:         else
8:             Send the local matrices G_i(p,q) to the PU 1 and wait to receive the
               cosine-sine pair (c,s).
9:         endIf
10:        Form the rotation matrix J(p,q,c,s) and update the matrices U and A_k:
           U=J(p,q,c,s)U;  A_k=J(p,q,c,s)^H A_k J(p,q,c,s).
11:      endFor
12:      Rebalance the load (if necessary).
13:      Generate new (p,q) pairs by shifting the indices according to tournament
         ordering
14:  endUntil
```

Fig. 1. Parallel Jacobi joint diagonalization algorithm for the ith PU; global synchronization (summation of global $\mathbf{G}(p,q)$ matrix) is described in lines 5–6. Line 12 corresponds to load balancing algorithm described in Fig. 3.

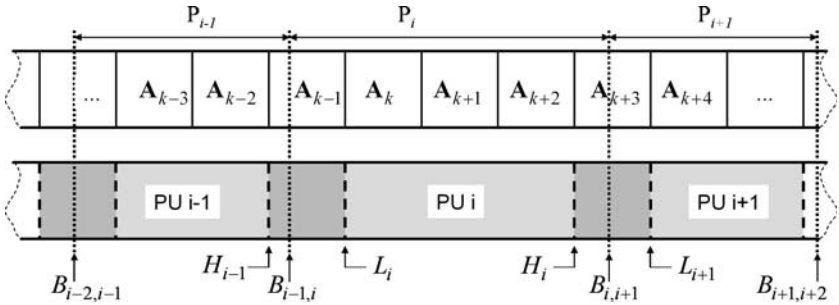

Fig. 2. Row-wise alignment of matrices (upper plot), and their column-wise distribution among different PUs (lower plot); P_i denotes the number of matrix columns being processed by the ith PU, L_i and H_i stand for lower and upper limits of memory space assigned to the local matrices of the ith PU, respectively, while $B_{i,i+1}$ denotes the boundary which separates the matrix columns processed by the ith PU form the columns processed by the $(i+1)$th PU. The last matrix $\mathbf{A}_K$ is supposed connected to the first matrix $\mathbf{A}_1$ (for clarity reasons only a portion of the circular data structure is depicted).

for example, the ith PU processes only approximately a half of the matrix $\mathbf{A}_{k+3}$, while the other half is processed by the $(i+1)$th PU. The exact choice of columns to be processed by each PU depends on the current values of (p,q) pairs and must be changed according to the tournament ordering (Fig. 1). As a result, a single pair of matrix columns must be interchanged after each sweep step between each pair of the PUs sharing

a common matrix. In order to minimize the communication costs, the number of shared matrices per PU should be kept minimal. On the other hand, all available network connections should be utilized to allow quick adaptation to load varying conditions. A good compromise is to allow each PU to share at maximum two matrices, i.e., to limit the exchange of shared matrix columns to a ring topology. This allows for rapid two-step adaptations to resource varying conditions. In the first LB run, all communication links are used to redistribute the local matrices (i.e. the matrices not being shared by different PUs), while the second LB run redistributes the columns of shared matrices across the virtually induced ring topology.

There are numerous ways how to determine the exact amount of data to be transferred between different PUs, in order to equalize their loads. However, taking into account the locality properties of the aforementioned matrix distribution, the diffusion-based nearest–neighbour algorithms[25] appear as the most appealing choice. Using the so called first-order diffusion scheme (FOS), for instance, the number of matrix columns to be transferred after the nth sweep step between the ith and jth PU can be calculated as:

$$\Delta_{i,j}^{(n)} = \sum_{l=1}^{L} \delta_{i,j}^{(n,l)} \tag{10}$$

with

$$\delta_{i,j}^{(n,l)} = \frac{m_{i,j}^{(n)}}{t_i^{(n)}} \left(t_j^{(n)} P_j^{(n,l)} - t_i^{(n)} P_i^{(n,l)} \right), \tag{11}$$

where $t_i^{(n)}$ stands for the effective processing time spent by the ith PU in the nth sweep step, L denotes the number of LB iterations, $P_i^{(n,l)} = P_i^{(n,l-1)} + \sum_j \delta_{i,j}^{(n,l-1)}$ with $P_i^{(n,0)} = P_i^{(n)}$ is the number of columns being processed by the ith PU after the lth iteration step (11), while $m_{i,j}^{(n)}$ stands for the (i, j)th element of generalized diffusion matrix $\mathbf{M}^{(n)}$.[14] For heterogeneous FOS schemes[25] already showed the optimal diffusion factors $m_{i,j}^{(n)}$ (in terms of convergence rates) can be calculated as:

$$m_{i,j}^{(n)} = \begin{cases} \phi_{i,j}^{(n)} \omega_{i,j}^{t_j^{(n)}}, & \text{if } i = j-1 \ \vee \ i = j+1, \\ 1 - \sum_k m_{k,j}^{(n)}, & \text{if } i = j, \\ 0, & \text{else}, \end{cases} \tag{12}$$

where $\omega_{i,j}$ stands for a weight associated to the communication link between ith and jth PU, and $\omega_{\min}$, $t_{\min}^{(n)}$ and $t_{\max}^{(n)}$ denote the minimal communication weight, and minimal and maximal processing times in the nth sweep step, respectively. The optimal convergence rate is achieved when $\phi_{i,j}^{(n)} = \min\left\{\frac{1}{t_i^{(n)}(\sum_k \omega_{i,k}+\varepsilon_0)}, \frac{1}{t_j^{(n)}(\sum_k \omega_{j,k}+\varepsilon_0)}\right\}$, where $\varepsilon_0 = e(C)\omega_{\min}\frac{t_{\max}^{(n)}}{t_{\min}^{(n)}}\sin^2(\frac{\pi}{2Q})$ and $e(C)$ stands for edge connectivity of the corresponding communication graph.[25] The main drawback of FOS schemes is their slow convergence. Typically, several hundreds iterations of (11) are needed to attain the optimal load distribution. More optimal second-order (SOS), and polynomial diffusion schemes, perform significantly faster.[14,23] However, they depend on a global knowledge of communication graph and require more sophisticated load movement algorithms (using higher-order diffusion schemes, the cases may appear in which a PU is selected to send out more load than it posses).

In our case, the use of discrete load units prevents the presented diffusion algorithm to balance load completely. Limiting the first LB run to the exchange of local matrices only, the net transfer across each communication channel has to be rounded down to the nearest integer multiple of N^2 and is additionally limited by the number of local matrices on each PU:

$$\tilde{\Delta}_{i,j}^{(n)} = \min(H_i - L_i, \left\lfloor \Delta_{i,j}^{(n)} \right\rfloor_{N^2}) \tag{13}$$

where L_i and H_i stand for lower and upper limit of memory space assigned to the local matrices processed by the ith PU, respectively, and we assumed $\tilde{\Delta}_{i,j}^{(n)} > 0$, i.e. the load flows from the ith to the jth PU. Now, assume (10) converges to its final value. Then the largest possible difference $\left|\tilde{\Delta}_{i,j}^{(n)} - \Delta_{i,j}^{(n)}\right|$ between the demanded and net load transfer on channel (i, j) yields $2N^2$ matrix elements. Accumulating differences on all the channels, the load discrepancy for the ith PU yields $\left|\bar{P}_i^{(n)} - \tilde{P}_i^{(n)}\right| \leqslant (\upsilon_i + 1)N^2$, where $\bar{P}_i^{(n)}$ denotes the optimal workload distribution in the nth sweep step, $\tilde{P}_{i,1}^{(n)}$ stands for the number of columns left on the ith PU after the first LB run, while υ_i is the number of direct neighbours of the ith PU. Denoting by $\bar{\mathbf{p}}^{(n)} = \left[\bar{P}_1^{(n)}, \bar{P}_2^{(n)}, \ldots, \bar{P}_Q^{(n)}\right]$ the optimal load vector in the nth sweep step, and by $\mathbf{p}^{(n,l)} = \left[P_1^{(n,l)}, P_2^{(n,l)}, \ldots, P_Q^{(n,l)}\right]$ the actual load vector after the lth iteration step (11), one can prove the following inequality:

$$\left\| \mathbf{p}^{(n,l)} - \bar{\mathbf{p}}^{(n)} \right\|_\infty \leqslant \frac{\lambda_M^l \left(t_{\max}^{(n)} \right)^2}{t_{\min}^{(n)}} \left\| \mathbf{p}^{(n,0)} - \bar{\mathbf{p}}^{(n)} \right\|_\infty , \tag{14}$$

where $\left\| \mathbf{p}^{(n,0)} - \bar{\mathbf{p}}^{(n)} \right\|_\infty = \max_i \left(\left| P_i^{(n,0)} - \bar{P}_i^{(n)} \right| \right)$ stands for initial discrepancy of load vector $\mathbf{p}^{(n,0)}$, $\left\| \mathbf{p}^{(n,l)} - \bar{\mathbf{p}}^{(n)} \right\|_\infty = \max_i \left(\left| P_i^{(n,l)} - \bar{P}_i^{(n)} \right| \right)$ for the discrepancy of load vector $\mathbf{p}^{(n,l)}$, while λ_M denotes the convergence factor, i.e. the second largest eigenvalue (in absolute value) of diffusion matrix $\mathbf{M}^{(n)}$. Hence, the first LB run of (10) ends as soon as $\max_i \left(\left| P_i^{(n,l)} - \bar{P}_i^{(n)} \right| \right) < N^2$, i.e. after at maximum $\frac{1}{(\lambda_M - 1)} \ln\left(\frac{N^2 t_{\min}^{(n)}}{\left(t_{\max}^{(n)} \right)^2 \left\| \mathbf{p}^{(n,0)} - \bar{\mathbf{p}}^{(n)} \right\|_\infty} \right)$ iteration steps. In the second LB run, blocks of $2N$ matrix elements are transferred across the communication ring, reducing the differences in workload to $\max_i \left(\left| P_i^{(n,l)} - \bar{P}_i^{(n)} \right| \right) < 4N$. The first LB run guarantees $\left\| \mathbf{p}^{(n,0)} - \bar{\mathbf{p}}^{(n)} \right\|_\infty \leqslant (\upsilon_i + 1) N^2$. Therefore, the second LB completes after at maximum $\frac{1}{(\lambda_M - 1)} \ln\left(\frac{t_{\min}^{(n)}}{N \left(t_{\max}^{(n)} \right)^2} \right)$ steps.

Note, however, that the attainment of fair workload after each sweep step is not always globally optimal. In highly competitive environments, such as clusters of personal computers, for example, sweep processing times $t_i^{(n)}$ change rapidly, while the data transfer costs are relatively high and can easily outweigh the benefits of rebalancing. A perfect load balance after each sweep step is, hence, desired only in the presence of stable workloads, whereas in frequent-load-change scenarios a rapid but coarse reduction in load imbalance is preferred.

Modelling job arrivals at each PU as an independent identically distributed (i.i.d.) random process with i.i.d. random processing times, the ith component of optimal load balancing vector $\bar{\mathbf{p}}^{(n)}$ is proportional to $\frac{t_i^{(n)}}{\sum_j t_j^{(n)}}$. For a large Q, $\sum_{j=1}^{Q} t_j^{(n)}$ converges towards normal distribution and changes in a relative short time interval (with respect to $t_i^{(n)}$, which remains more or less constant between subsequent job arrivals on the ith PU). Fair load distribution $\bar{\mathbf{p}}^{(n)}$ can, hence, be modelled as a short-term stationary normally distributed random vector, making the vector version of Kalman filter the most favourable adaptive estimator of globally optimal load vector $\bar{\mathbf{p}}^{(n)}$. However, the vector version of Kalman filter requires the inversion of a $Q \times Q$ matrix after each sweep step and induces severe computational burden, especially for large Q. Suboptimal, but computationally appealing solution deploys a scalar version of Kalman filter to locally average the processing times $t_i^{(n)}$ before using them in the diffusion

1: Measure processing time $t_i^{(n)}$ needed to calculate $N/2$ local $\mathbf{G}_i(p,q)$ matrices, use it to update Kalman filter (15) and send it to the $(i\text{-}1)$-th and $(i+1)$-th PU.

2: Wait to receive processing times $\bar{t}_{i-1}^{(n)}$ and $\bar{t}_{i+1}^{(n)}$ from the $(i\text{-}1)$-th and $(i+1)$-th PU, respectively, and recalculate diffusion factors $m_{ij}^{(n)}$.

3: Use (13) to determine net transfer $\tilde{\Delta}_{i,j}^{(n)}$ across each channel, and redistribute the local matrices.

4: Set $\omega_{i,j}^{(n)} = 0\,;\ \forall j \neq i \pm 1$, recalculate diffusion factors $m_{i,j}^{(n)}$ and determine the amount of unbalanced workload $\Delta_{i,j}^{(n)}$.

5: Round down the net transfer in the second LB run ($\breve{\Delta}_{i-1,i}^{(n)} = \left\lfloor \Delta_{i-1,i}^{(n)} \right\rfloor_{2N}$ and $\breve{\Delta}_{i+1,i}^{(n)} = \left\lfloor \Delta_{i+1,i}^{(n)} \right\rfloor_{2N}$) and redistribute the columns of shared matrices.

Fig. 3. Pseudocode of two-step load balancing algorithm on the ith PU. The first step (line 3) utilizes all available network connections to redistribute the local matrices. The second step (lines 4 and 5) redistributes columns of remaining matrices across the virtually induced ring topology (Fig. 2).

scheme (Fig. 3). In such a case, Kalman filter acts solely as an adaptive exponentially weighted moving average estimator:

$$\begin{aligned} \bar{t}_i^{(n+1)} &= \bar{t}_i^{(n)} + c_i^{(n)}(t_i^{(n)} - \bar{t}_i^{(n)}), \\ c_i^{(n)} &= \left(e_i^{(n)} + \xi_i^{\text{model}}\right) / \left(e_i^{(n)} + \xi_i^{\text{model}} + \xi_i^{\text{measure}}\right), \end{aligned} \tag{15}$$

where $e_i^{(n+1)} = \left(1 - c_i^{(n)}\right) + (e_i^{(n)} + \xi_i^{\text{model}})$ with $e_i^{(0)} = 0$, and ξ_i^{model} and $\xi_i^{measure}$ stand for the model and measurement uncertainty, respectively. The use of the scalar Kalman filter (15) is further justified in Section 4.2.

4. THEORETICAL PERFORMANCE EVALUATION

In this section, we compare the theoretical performance of the proposed algorithm to the performances of parallel algorithm described in Ref. 17 and sequential Jacobi joint diagonalization algorithm. All algorithms share the same number of required sweeps. Generally speaking, no proof of their global convergence rate can be made (there may even be no transformation $\mathbf{U}$ which simultaneously diagonalizes the whole set of matrices $\mathbf{A}_k$). However, under the assumption that all diagonalized matrices $\mathbf{A}_k$ share a common eigen-structure, the quadratic local convergence of sequential Jacobi algorithm has been proven.[7]

In the sequel, our discussion is limited to a single sweep, while the basic floating point operation will be referred to as a flop. No distinction between the real and complex flops will be made. Note, however, that the flops are implicitly supposed to be complex whenever the Hermitian matrices are diagonalized.

4.1. Communication Complexity and Comparison to Parallel Algorithm

The transfer of symmetric 3×3 $\mathbf{G}_i(p, q)$ matrices in (9) represents no serious network load. Assuming that floating-point data type occupies b_r bytes, a total amount of $6b_rQ$ bytes for each global addition, i.e. each Jacobi rotation, is transferred over the network. Exchange of the shared matrix columns requires at maximum $4QNb_r$ bytes per sweep. Finally, rebalancing of the PUs' workload takes additional $2b_rQ$ bytes per sweep step for dispatching processing times $t_i^{(k)}$, $4LQb_r$ for mutual exchange of $P_i^{(n,l)}$ and b_rN bytes per transferred matrix column. Having $N-1$ steps per sweep, the total communication costs yield:

$$C_{\text{distributed}}^{\text{sweep}} = \sum_{i=1}^{Q} [N(3N+1)b_r\varepsilon_i + 2(N-1)\sigma_i] + C_{\text{loadbalancing}}^{\text{sweep}}, \tag{16}$$

where

$$\begin{aligned} C_{\text{loadbalancing}}^{\text{sweep}} = \sum_{i=1}^{Q} \Big[& 2(N-1)(2L+1)b_r\varepsilon_i + S_iN^2b_r\varepsilon_i \\ & +3(N-1)(2L+1)\sigma_i \Big], \end{aligned} \tag{17}$$

stands for communication costs due to load balancing, ε_i is the average time required to transfer 1 byte from the ith PU to its neighbours, S_i is the number of local matrices per sweep that are transferred between the ith PU and its neighbours due to the first LB run, and σ_i denotes the start-up time required to initiate the communication activity.

According to (16), the amount of transferred data is independent of the number of matrices $\mathbf{A}_k$, but increases with their sizes and the number of PUs. On the other hand, using conventional column (row) shifting as proposed in Ref. 17, the amount of $2(Q-1)KNb_r + Nb_r$ bytes per sweep step has to be transferred (without the LB costs). The total communication costs of such an approach yield:

$$C_{\text{parallel}}^{\text{sweep}} = \sum_{i=1}^{Q} [N(N-1)b_r\varepsilon_i + a(i)KN(N-1)b_r\varepsilon_i + a(i)(N-1)\sigma_i] + C_{\text{loadbalancing}}^{\text{sweep}}, \tag{18}$$

where $a(i) = \begin{cases} 1 \; if \; i = 0, Q \\ 2 \;\; \text{otherwise} \end{cases}$

Note that no load balancing algorithm was proposed in Ref. 17, hence, only a general denotation of load balancing costs is used in (18).

Comparison between the costs (16) and (18) for a different number of $N \times N$ matrices, K, on a different number of PUs, Q, is further shown by Fig. 4. For the sake of simplicity, all the PUs were assumed to be of the same computational power, i.e.$\forall i$, $\gamma_i = 4 \cdot 10^{-9}$, where γ_i stands for the time required to execute one flop on the ith PU, while a 100 Mbps network was simulated, which results in $\varepsilon_i = 10^{-7}$and in time to initiate the network transfer $\sigma_i = 10^{-4}$. With no LB algorithm proposed in Ref. 17, no costs due to load balancing were taken into account. Figure 4 should, hence, serve only for a comparison of two different matrix distribution strategies.

4.2. Computational Complexity and Comparison to Sequential Joint Diagonalization

The main computational cost in the sequential Jacobi procedure is due to the update of the diagonalized matrices. In each Jacobi step we compute the matrix $\mathbf{G}(p, q)$, which requires $12K$ flops, and update the matrices $\mathbf{A}_k$, which takes additional $12KN$ flops. Finally, updating the matrix $\mathbf{U}$ takes $6N$ flops per Jacobi step. Performing $N(N-1)/2$ rotations per sweep, the computational complexity of the sequential Jacobi procedure on only one, for example the ith PU, yields

$$T_{\text{sequential}}^{\text{sweep}}(N, K) = \gamma_i \left[6KN(N-1) + 6KN^2(\mathbf{N}-1) \; + 3N^2(\mathbf{N}-1) + \frac{N(N-1)}{2}\beta \right], \tag{19}$$

where β denotes the computational complexity (in flops) of 3×3 eigen-decomposition. The exact choice of eigen-decomposition algorithm depends on the numerical library used, hence, only a general denotation of its complexity is used in this paper. In either case this adds very little to the computational complexity (in the case of symmetric QR algorithm, for instance, β yields $36^{(15)}$).

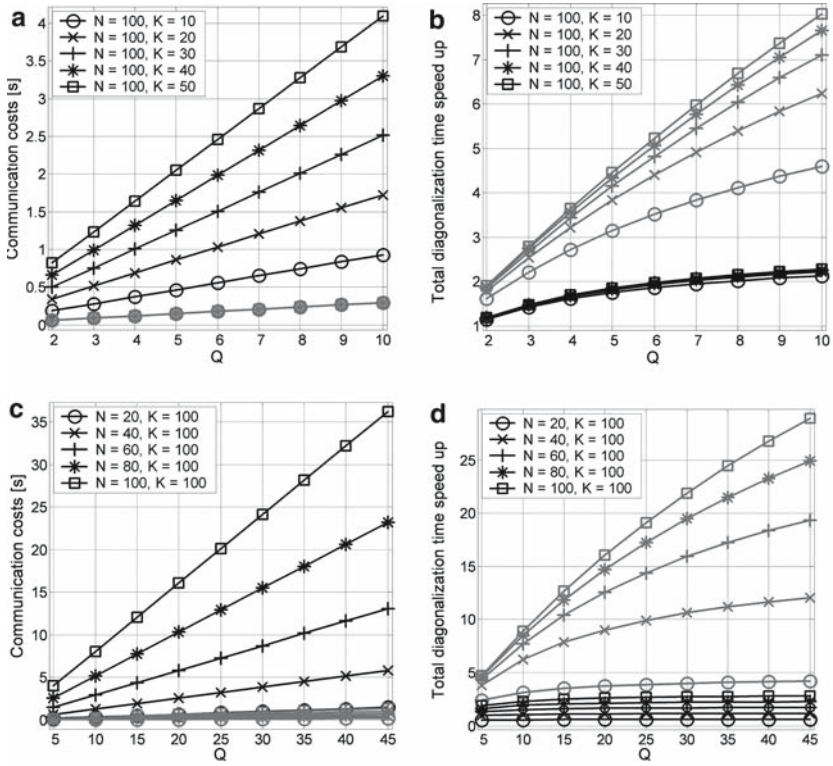

Fig. 4. Theoretical comparison of communication costs of joint diagonalization for our algorithm (grey lines) and the algorithm proposed in Ref. 17 (black lines) versus the number of processing units (Q) joint diagonalizing: (a) different number (K) of 100×100 matrices, (c) 100 matrices of different sizes ($N \times N$). The corresponding total diagonalization time speedups are depicted in subplots (b) and (d). All PUs were assumed to be of the same computational power, $\forall i, \gamma_i = 4 \cdot 10^{-9}$ (250 Mflops), while a 100 Mbps network was simulated ($\varepsilon_i = 10^{-7}, \sigma_i = 10^{-4}$). Costs of load balancing are not depicted.

When running a distributed version of joint Jacobi diagonalization on clusters of personal computers, the additional waiting times w_i arise as a consequence of the required global synchronizations. Having $N - 1$ global synchronizations per sweep the computational costs of the ith PU per sweep reach

$$T_i^{\text{sweep}}(N, K) = \gamma_i \Bigg[3P_i(N-1)(2N+1) + \frac{N(N-1)(\beta + 3N)}{2} + (26{+}10L)(N-1) \Bigg] + w_i, \tag{20}$$

where L denotes the number of LB iterations in each LB run and

$$w_i = \sum_{n=1}^{N-1} \left(\max_j (t_j^{(n)}) - t_i^{(n)} \right), \tag{21}$$

denotes the waiting time of ith PU per sweep.

Define now the ith optimal speedup factor as a ratio between the joint computational power of all PUs and the computational power of the ith PU:

$$F_i^{\text{opt}} = \gamma_i \sum_{j=1}^{Q} \frac{1}{\gamma_j}. \tag{22}$$

The computational complexity of the distributed Jacobi algorithm, measured on ith PU, approximates:

$$T_{\text{distributed}}^{\text{sweep}} \approx \left[\frac{\gamma_i \left(6KN(N-1) + 6KN^2(\mathbf{N}-1) + N(N-1)\left(3N + \frac{\beta}{2}\right) + (26+10L)(N-1) \right)}{F_i^{\text{opt}}} \right] + w_i. \tag{23}$$

When $K \gg 1$ and $N \gg 1$ the achieved speedup factor (a ratio between the total processing time of sequential and distributed Jacobi algorithm) measured on the ith PU yields

$$\begin{aligned} F_i^{\text{distributed}} &= \left(\frac{T_{\text{distributed}}^{\text{sweep}} + C_{\text{distributed}}^{\text{sweep}}}{T_{\text{sequential}}^{\text{sweep}}} \right)^{-1} \\ &\approx \left(\frac{1}{F_i^{\text{opt}}} + \frac{w_i}{T_{\text{sequential}}^{\text{sweep}}} + \frac{26+10L}{(6K+3)N^2 + 6KN} + \frac{b_r \sum_{j=1}^{Q} (S_j + 3 + \frac{4L}{N})\varepsilon_j}{6KN\gamma_i} + \frac{(5+6L)\sum_{j=1}^{Q} \sigma_j}{6KN^2\gamma_i} \right)^{-1}, \end{aligned} \tag{24}$$

which comprises

$$\frac{T_{\text{distributed}}^{\text{sweep}}}{T_{\text{sequential}}^{\text{sweep}}} \approx \frac{1}{F_i^{\text{opt}}} + \frac{w_i}{T_{\text{sequential}}^{\text{sweep}}} + \frac{26 + 10L}{(6K+3)N^2 + (6K + \beta/2)N} \tag{25}$$

and

$$\frac{C^{\text{sweep}}_{\text{distributed}}}{T^{\text{sweep}}_{\text{sequential}}} = \frac{N^2 b_r \sum_{j=1}^{Q} S_j \varepsilon_j + [3N(N-1)+(4L+2)(N+1)] b_r \sum_{j=1}^{Q} \varepsilon_j + (N-1)(5+6L) \sum_{j=1}^{Q} \sigma_j}{6KN^2\gamma_1 \left[N - \frac{1}{N} + \frac{\beta}{12K} - \frac{\beta}{12KN} \right]}. \tag{26}$$

Equation (24) clearly reveals that the proposed algorithm converges to the optimal speedup factor as the number and sizes of the diagonalizing matrices increase. The importance of smart load balancing strategy is also clarified, revealing the tradeoff between the waiting times w_i and load balancing costs. In order to further illustrate this tradeoff, the proposed LB algorithm was applied to simulated clusters of 25 and 50 PUs, arranged in 2D mesh (with fixed dimension set to 5) and ring topology. Speeds of PUs were considered inversely proportional to the number of currently executed jobs. Job arrivals were modelled as independent Poisson random processes with exponentially distributed job processing times. Different levels of load variability at the same average PU load were simulated by setting the mean job inter-arrival time and job processing time to 0.5, 1, 5 10 and 25 s, respectively. The results, averaged over 10 simulation runs, are depicted in Fig. 5. In each simulation run, communication weights ω_{ij} were randomly chosen from the interval [1, 10]. The model uncertainty ξ_i^{model} in (15) was set to 0.01, while the variance of local sweep processing time $t_i^{(n)}$ was used as an estimate of the measurement uncertainty ξ_i^{measure}.

5. EXPERIMENTAL RESULTS

The presented distributed joint Jacobi diagonalization algorithm was implemented in Microsoft Visual C++. Communication among different PUs was carried out using the MPICH implementation[1] of the Message Passing Interface, while the VectorSpace C++ Library was used for matrix computations (the library is accessible online through the URL at http://www.vector-space.com). The tests were conducted on one single personal computer, and clusters of 10, 20, 30, 40 and 50 personal computers. All cluster nodes ran the Windows XP operating system and were connected to 100 Mbps Ethernet. Technical details about PUs constituting the clusters are presented in Table I. The average time σ_i to initiate the communication activity among two nodes was measured using the mpptest benchmark[16] and yielded 100.4 μs, while the average times to transfer 32, 128 and 512 bytes of data were estimated to 3.2, 12.5 and 46.2 μs, respectively. No special attention was paid to avoid competition for resources

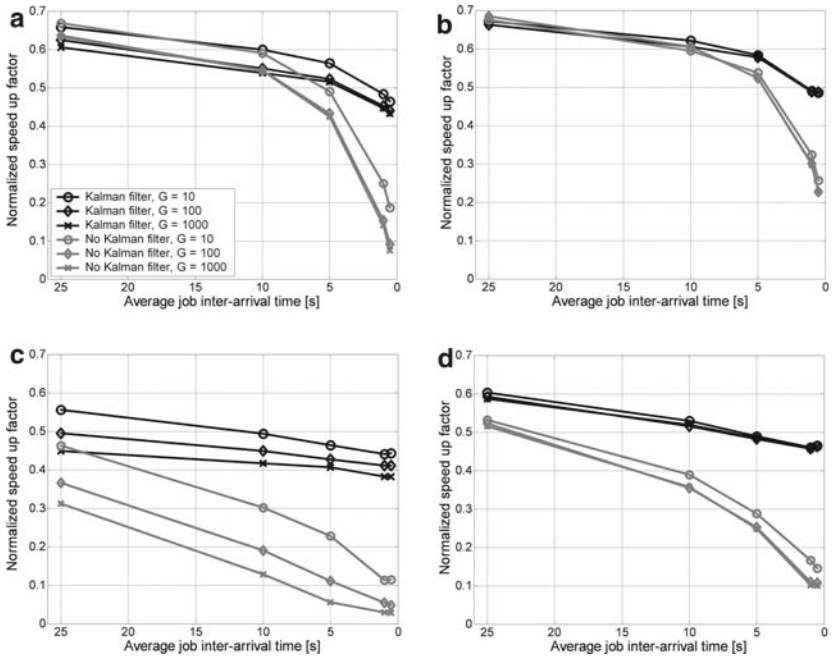

Fig. 5. Total diagonalization time speedup depending on the load variability (average job inter-arrival time on all PUs) and load balancing strategy (number of load balancing iterations G) used when joint diagonalizing 200 of 200×200 matrices for: (a) ring topology with 25 PUs, (b) 2D mesh topology with 25 PUs, (c) ring topology with 50 PUs, and (b) 2D mesh topology with 50 PUs. The results are averaged over 10 simulation runs and normalized by the sum of available computation power on all PUs in each simulation run. In all simulation runs, the average PU load was kept constant by setting average job processing time equal to the average job inter-arrival time.

with other applications. However, the fair initial load distribution was calculated as the first step in all experiments. First, the computing power of each node was determined experimentally by measuring the time spent on multiplying 1000 random matrices of size 100×100 on each node (Table I). Second, the matrices were distributed proportionally to the estimated computing power of each node.

In the first experiment, the time required to diagonalize the same set of matrices on a different number of PUs was determined. Groups of 200, 400, 600 and 800 random symmetric matrices of size 400×400 were generated. No special care about the eigen-structure of the generated matrices was taken (note that the joint diagonalization procedure does not rely on the assumption that all the matrices share the same eigen-structure[8,10]).

Table I. Declared characteristics and measured computational rates (in Mflops per second) of PUs constituting test clusters of 1–50 personal computers (PCs). All PCs ran the Windows XP operating system and were connected with a 100 Mbps network. The measured computational rate of each PU was calculated as a reciprocal value of the time needed to multiply 1000 random matrices of size 100×100.

Sequential number	Total memory	CPU, declared speed	Computational power (in Mflops per sec.)
1	1.5 GB	Intel Pentium 4, 2.0 GHz	283.2
2	500 MB	Intel Pentium 4, 1.8 GHz	259.5
3–15	256 MB	Intel Pentium 4, 1.6 GHz	231.5
16–26	256 MB	AMD Athlon 2000+, 1.6 GHz	135.8
27–39	256 MB	Intel Celeron 4, 1.7 GHz	250.0
40	512 MB	AMD Athlon 2600+, 2.1 GHz	139.6
41–50	256 MB	Intel Celeron 4, 1.7 GHz	252.7

The generalized LB diffusion matrix as defined in (12) was used. The number of LB iterations in each LB run was set to $G = 10$. The measured sweep processing times were averaged using scalar Kalman filter with the model uncertainty $\xi_i^{\text{model}} = 0.01$ and measurement uncertainty ξ_i^{measure} set equal to the variance of the local processing time $t_i^{(n)}$.

All the performance indices were averaged over 10 experimental sessions with randomly allocated cluster nodes and new random matrices generated in each session. The total execution time and the time spent for communication between the PUs were measured. Figure 6(a) depicts the average speed-up factor (the ratio between the time required to diagonalize a particular group of matrices on a single PU and the time required to diagonalize it on several PUs) achieved by a distributed implementation of the joint diagonalization depending on the number of PUs. For the sake of clarity, the computational powers of PUs (Table I) constituting the cluster were equalized and proportionally scaled processing times are displayed. Note also the times required to diagonalize different number of matrices on a single PU are all normalized to 1. The time spent for communication among PUs during the same experiment is illustrated in Figure 6(b). To clarify the communication costs, the values are divided by the corresponding total diagonalization time on a single PU (PU 1 in Table I).

In the second experiment, the performance of distributed joint diagonalization was measured versus the sizes of matrices. Assuming the size of matrices varies instead of their number, the same protocol was followed as in the first experiment. First, groups of 400 symmetric matrices of sizes 100×100, 200×200, 400×400 and 600×600 were randomly generated.

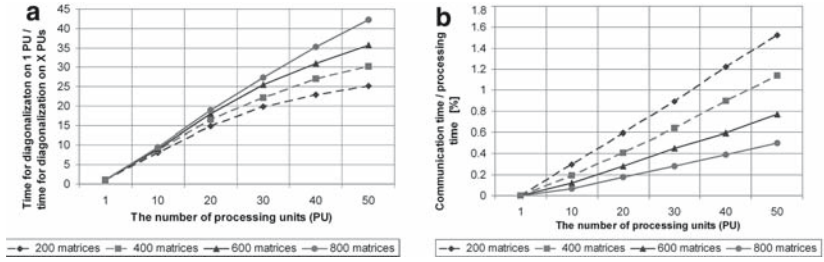

Fig. 6. Performance of the proposed distributed algorithm used for joint diagonalisation of a different number of 400×400 matrices versus the number of processing units: (a) total diagonalization time speedup, (b) the ratio between the time spent for communication among the PUs and the total diagonalization time. The processing times are scaled in proportion to the measured computational powers of PUs (Table I), while the times required to diagonalize a different number of matrices on a single PU are all normalized to 1.

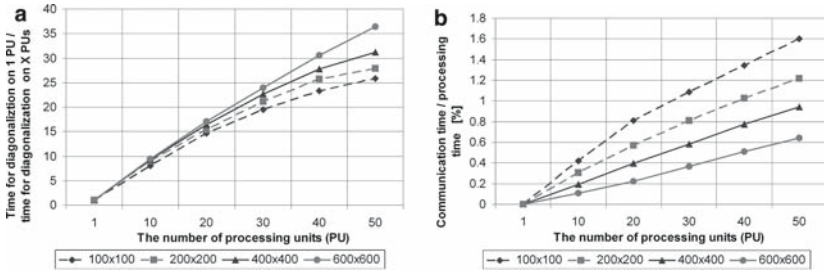

Fig. 7. Performance of the proposed distributed algorithm depending on the number of PUs when joint diagonalizing 400 matrices of different sizes: (a) total diagonalization time speedup, (b) the ratio between the time spent for communication among the PUs and the total diagonalization time. The processing times are proportionally scaled according to the measured computational powers of PUs (Table I), while the times required to diagonalize different number of matrices on a single PU are all normalized to 1.

The total execution time and the time spent on communication between the PUs were averaged over 10 experimental sessions, while, analogously to the first experiment, new random matrices were generated in each session. The results are depicted in Fig. 7.

6. CONCLUSION

The presented approach for distributed joint Jacobi diagonalization proves to be very efficient. Analytically assessed performance reveals its

asymptotically optimal speedup factor and, at the same time, demonstrates its high scalability (no restriction is put on the number of processors and no zero-padding[11] of the matrices $\mathbf{A}_k$ is required). Communication of local $\mathbf{G}_i(p, q)$ matrices among the PUs proves to be a minor cost, especially when a large number of matrices is diagonalized. Both the theoretical and experimental results prove the relative contribution of communication activity to the overall costs is proportional to the number of nodes, but decreases linearly with both the number and the sizes of matrices. Diagonalizing 800 matrices of size 400×400 on 50 PUs, the speed-up factor yields 42.2 and improves with the number of matrices (Fig. 6). Diagonalizing 400 matrices of size 600×600 on 50 PUs the speed-up ratio drops to 36.1 (Fig. 7) but still increases with the size of matrices. The processing time by far exceeds the communication costs. Although a slight increase is noticed when adding new PUs, the relative share of communication costs reduces strongly with both the number and the sizes of matrices.

Diagonalization of small number of matrices appeared to be less efficient. The reason can be sought in local update of unitary matrix $\mathbf{U}$. In order to increase the robustness of distributed joint diagonalization, local copy of entire matrix $\mathbf{U}$ is stored and updated by each PU. This adds very little to computational costs when a large number of matrices is being diagonalized, but gets significant in the case of small number of matrices. An alternative solution would be to distribute the task of updating the matrix $\mathbf{U}$ among all PUs. In such a case each PUs updates only a portion of matrix $\mathbf{U}$ (to guarantee fair workload distribution, the exact number of columns of $\mathbf{U}$ to be updated by the ith PU must be made proportional to the number of processed matrix columns P_i). This speeds up the diagonalization but makes it vulnerable to the drops-outs of the cluster nodes.

Finally, adaptive diffusion-based load balancing strategy introduced in this paper relies on the experimentally determined computing power of cluster nodes and exhibits very low and adaptable communication costs. Each PU communicates only with its direct neighbors, commuting information about its sweep processing time $t_i^{(n)}$. Under steady conditions (no multitasking on PUs), optimal load balance is achieved after only a few initial sweep steps, while in dynamic conditions, an adaptive averaging of the measured processing times using the standard Kalman filter limits the load balancing costs. Although heuristic and suboptimal, scalar Kalamn filter proved to be an efficient and computationally appealing solution (Fig. 5). Moreover, globally optimal load balance was achieved in our studies only after approximately 10 iterations per LB run, proving that slow convergence of the FOS schemes is of minor concern in our case.

Theoretical explanation of this phenomenon is beyond the scope of this paper and will be investigated in the future work.

ACKNOWLEDGEMENT

This work was supported by the Slovenian Ministry of Higher Education, Science, and Technology (Programme Funding P2-0041).

REFERENCES

1. Argonne National Laboratory, MPICH – A Portable Implementation of MPI, http://www-unix.mcs.anl.gov/mpi/mpich/html, (2006).
2. J. L. Barlow and I. Slapniéar, Optimal perturbation bounds for the Hermitian Eigenvalue Problem, *Linear Algebra Appl.*, **309**:19–43 (1993).
3. M. Bečka, G. Oška, and M. Vajteršic, Dynamic Ordering for Parallel Block-Jacobi SVD Algorithm, *Parall. Comput.* **28**:243–262 (2002).
4. A. Belouchrani and K. M. Amin, Blind Source Separation Based on Time-Frequency Signal Representation, *IEEE Trans. Signal Process.* **46**:2888–2898 (1998).
5. B. Boashash, *Time-Frequency Signal Analysis and Processing,* Prentice Hall PTR, Englewood Cliffs, NJ (2001).
6. R. P. Brent and F. T. Luk, The Solution of Singular-Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays, *SIAM J. Sci. Stat. Comput.* **6**:69–84 (1985).
7. A. Bunse-Gerstner, R. Byers, and V. Mehrmann, Numerical Methods for Simultaneous Diagonalization, *SIAM J. Matrix Anal. Appl.* **14**:927–949 (1993).
8. J. F. Cardoso, Perturbation of Joint Diagonalizers, Technical. report 94D023, Signal Department Telecom Paris, (1994).
9. J. F. Cardoso and A. Souloumiac, Blind Beamforming for Non-Gaussian Signals, *IEE Proc. F* **6**:362–370 (1993).
10. J. F. Cardoso and A. Souloumiac, Jacobi Angles for Simultaneous Diagonalization, *SIAM J. Mater Anal. Appl.* **17**:161–164 (1996).
11. E. M. Daoudi and A. Lakhouaja, Exploiting the Symmetry in the Parallelization of the Jacobi Method, *Parall. Comput.* **23**:137–151 (1997).
12. J. W. Demmel and K. Veselic, Jacobi Method is More Accurate Than QR, Technical report 468, Department of Computer Science, Courant Institute of Mathematical Science, New York University (1989).
13. J. W. Demmel, *Trading Off Parallelism and Numerical Stability,* Technical report CRPC-TR92422, Center for research on Parallel Computation, Rice University, Huston (1992).
14. R. Diekmann, A. Frommer, and B. Monien, Efficient Schemes for Nearest Neighbor Load Balancing, *Parall. Comput.* 789–812 (1999).
15. G. H. Golub and C. F. Van Loan, *Matrix Computation*, 3rd Ed., The Johns Hopkins University Press, Baltimore (1996).
16. W. Gropp and E. Lusk, Reproducible Measurements of MPI Performance Characteristics, in *Proc. 6th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Barcelona, Spain pp. 26–29 (1999).
17. F. Gygi, J. L. Fattebert, and E. Schwegler, Computation of Maximally Localized Wannier Functions Using a Simultaneous Diagonalization Algorithm, *Comput. Phys. Commun.* **155**:1–6 (2003).

18. A. Holobar, C. Fevotte, C. Doncarli, and D. Zazula, Single Autoterm Selection for Blind Source Separation in Time-Frequency Plane, in Proc. EUSIPCO'02, Toulouse, France (2002).
19. A. Holobar and D. Zazula, A New Approach for Blind Source Separation of Convolutive Mixtures of Pulse Trains", in *Proc. BSI'02*, Como, Italy pp. 163–166 (2002).
20. A. Hyvarinen, J. Karhunen, and E. Oja, *Independent Component Analysis*, Wiley, New York (2001).
21. M. Joho and K. Rahbar, Joint Diagonalization of Correlation Matrices by Using Newton Methods With Application to Blind Signal Separation, in *Proc. SAM*, Rosslyn pp. 403–407 (2002).
22. T. Katagiri and Y. Kanada, An Efficient Implementation of Parallel Eigenvalue Computation for Massively Parallel Processing, *Parall. Comput.* **27**:1831–1845 (2001).
23. T. Lücking, B. Monien, and M. Rode, On the Problem of Scheduling Flows on Distributed Networks, in Proc. MFCS 2002, Vol. 2420, Warsaw, Poland, pp. 495–505 (2002).
24. D. T. Pham, Joint Approximate Diagonalization of Positive Definite Hermitian Matrices, *SIAM J. Matrix Anal. Appl.* **22**:1136–1152 (2001).
25. T. Rotaru and H-H. Nägeli, Dynamic Load Balancing by Diffusion in Heterogeneous Systems, *J. Parallel Distrib. Comput.* **64**:481–497 (2004).
26. A. Yeredor, Non-Orthogonal Joint Diagonalization in the Least Squares Sense With Application in Blind Source Separation, *IEEE Trans. Signal Process.* **50**:1545–1553 (2002).