A Parallel Performance Study of Jacobi-like Eigenvalue Solution

Makan Pourzandi *
Laboratoire de l'Informatique du Parallélisme,
Unité de Recherche Associée 1398 du CNRS
Ecole Normale Supérieure de Lyon,
69364 Lyon Cedex 07, France
Tel. (+33) 72 72 85 03 Fax (+33) 72 72 80 80
e-mail: mpourzan@lip.ens-lyon.fr

Bernard Tourancheau †‡
The University of Tennessee
Computer Science Department,
Knoxville, TN 37996-1301, USA
Tel (1) 615 974 8295, Fax (1) 615 974 8296,
e-mail: btouranc@cs.utk.edu

March 24, 1994

Abstract

In this report we focus on Jacobi like resolution of the eigen-problem for a real symmetric matrix from a parallel performance point of view: we try to optimize the algorithm working on the communication intensive part of the code. We discuss several parallel implementations and propose an implementation which overlaps the communications by the computations to reach a better efficiency. We show that the overlapping implementation can lead to significant improvements. We conclude by presenting our future work.

*This work was supported by MRE grant No. 974, the CNRS-NSF grant No. 950.22/ 07 and the research program C3.

†On leave from LIP, CNRS URA 1398, ENS Lyon, 4 allée d'Italie, 69364 Lyon Cedex 07, France.

‡This work was supported in part by the National Science Foundation under grant ASC-871728, the National Science Foundation Science and Technology Center Cooperative Agreement CCR-8809615, the DARPA and ARO under contract DAAL03-91-C-0047, PRC C^3, CNRS-NSF grant 950.223/07, Archipel SA and MRE under grant 974, and DRET.

1 Introduction

As quantitative analysis becomes increasingly important in sciences and engineering, the need grows for faster methods to solve large eigenvalue problems. Large eigenvalue problems occur in a wide variety of applications, including the dynamic analysis of large-scale structures such as aircraft and spacecraft, the prediction of structural responses in solid and soil mechanics, the study of solar convection, the modal analysis of electronic circuits, and the statistical analysis of data [TMLZ93]. Thus the need for faster methods to solve these large eigenvalue problems becomes very important.
The problem of finding the eigenvalues of a matrix can be stated as follows: Find the values λ that satisfy the equation: $Ax = \lambda x$ for a vector x, which is called an eigenvector and λ an eigenvalue.
In this report we focus on Jacobi like resolution of the eigen-problem for a real symmetric matrix from a parallel performance point of view: we try to optimize the algorithm working on the communication intensive part of the code. We discuss several parallel implementations [Ebe87, EP90, Fou89, LP89a] and propose an implementation which overlaps the communications by the computations to reach a better efficiency. We first present briefly our target machine and our analysis model in the sections 2 and 3. In Chapter 4, we present the sequential Jacobi like resolution [Jac46, Mod88, Wil65]. Afterwards, we discuss the parallel implementations (section 5). We discuss our implementation on the Intel machine iPSC/860 hypercube, using the Intel gossiping procedure. Then, we discuss the same implementation but using our hand coded gossiping algorithm, following the works of [Fra90, JH89] leading to a very efficient solution. Afterwards, we present the same algorithm with overlapping of the communications by the computations. We use for that, a general methodology, developed in [DT92, PT93] and a tuned implementation. In the next section, we compare the experimental results of all algorithms. We show that the overlapping implementation can lead to a 6% improvement of the execution timings and that represents a decrease of 35% of the total communication time. This is achieved on our target hardware which has only one asynchronous communication port. Regarding the communication strategy employed, we surely guess on even more improvement if the hardware is able to handle multi-ports asynchronous communications. We conclude by presenting our future work.

2 Target machine

The experiences were done on a 32 nodes iPSC/860 with a hypercube topology. Each node of the iPSC provides an i860 processor, a Direct Connect Module and 16 Megabytes of memory. The Direct Connect Module (DCM) does the internodes communications. Using the DCM, the communications are independent from the i860 and are routed in circuit switched mode [Int90, MMM91, SB77].

There are several technical reports concerning the iPSC/860 architecture and communication performances [Dun90, MM91]. One can easily refer to them to have more details.

The i860 is a 40 MHz RISC type processor with 8k bytes of cache memory. It uses two arithmetic units (adder and multiplier) and a graphic unit. These units could be used in pipelined and chained modes. It allows i860 to have peak performances of 80 Mflops (32 bits) or 60 Mflops (64 bits). Actually the performances are 11.5 Mflops with our present compilers (64 bits) for the average vector length of our experiments (512 words). The gap between the peak and sustain performance is principally due to memory delays (cache miss, page-translation-miss, DRAM access delays ...) [Dun90].

3 Analysis Model

Let τ_a be the time to perform a floating point operation (double precision, addition or multiplication). The time to communicate between 2 neighbor nodes is modelized by $\alpha + L * \beta$ where α is the startup time and β is the time to transmit a word. For the iPSC/860 α is 136 μs for long messages (larger than 100 bytes) and 75 μs for short ones. β is 3,2 μs for a word of 4 bytes [DS86, Dun90].

We define the speedup as $sp = \frac{T_1}{T_p}$ and the efficiency as $e = \frac{T_1}{pT_p}$, where T_i is the execution time for the algorithm using i processors and T_1 is the execution time for the best sequential algorithm.

4 Jacobi Algorithm

In this subsection we define our notation for the Jacobi method for symmetric matrix diagonalization. For an $n \times n$ real symmetric matrix A with elements a_{pq} the Jacobi method [GL90] systematically reduces the norm of the non-diagonal elements:

$$off(A) = \sum_{p} \sum_{q \neq p} a_{pq}^2 \tag{1}$$

by a sequence of plane (Jacobi) rotations. We call a Jacobi sweep, every $\frac{n(n-1)}{2}$ plane rotations reducing all non diagonal elements.

$$A^{(1)} = A,\ A^{(k+1)} = J^{(k)} A^{(k)} J^{(k)T},\ k = 1, 2, 3 \ldots$$

Where the matrix $J^{(k)}$ is block-diagonal such that

$$J^{(k)}_{(i,j)} = \delta_{ij}\ (i, j \neq p, q), \quad \text{with } \delta_{ij} = 1 \text{ if } i = j, = 0 \text{ otherwise}$$

$$J^{(k)}_{(p,q)} = -J^{(k)}_{(q,p)} = s \quad J^{(k)}_{(p,p)} = J^{(k)}_{(q,q)} = c$$

where $s = sin(\Theta_{pq}^{(k)})$ and $c = cos(\Theta_{pq}^{(k)})$. The angle $\Theta_{pq}^{(k)}$ is chosen such that a_{pq}^{k+1} is annihilated.

The Forbenius norm, i.e. the sum of the squares of the matrix elements, is invariant under orthogonal transformations, hence, we have:

$$off(A^{(k+1)}) = off(A^{(k)}) - 2\left(a_{pq}^{(k)}\right)^2.$$

Thus a sequence of matrices $A^{(k)}$ is produced such that $\lim_{k\to\infty} A^{(k)} = D$, a diagonal matrix consisting of the eigenvalues of A, and

$$\lim_{k\to\infty} J^{(1)T} J^{(2)T} J^{(3)T} \ldots J^{(k)T} = V,$$

is a matrix consisting of the eigenvectors of A [GL90, TY91]. We skip the annihilation of $a_{pq}^{(k)}$ when $a_{pq}^{(k)} < \varepsilon$, because the reduction in off(A) is not worth the cost. This leads to the algorithm which is called the Jacobi threshold method. To save computing time, one chooses a definite order for the rotations. One can use for instance the raw- (or column-) cyclic method [Mod88, Wil65]. In the raw-cyclic scheme, we simply pick (p, q) in raw-by-raw fashion. For instance in the case $n = 4$ the following rotations

$$(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)$$

are made in a complete sweep [GL90, TY91]. We present the corresponding sequential algorithm on Figure 1.

While Iterations $<$ NBMAXITER **and** off(A) $> \varepsilon$ **do**
  Choose indices p and q with $1 \leq p < q \leq n$
  if $(a_{pq} > \varepsilon)$ **then** Execute Jacobi rotation(p, q)
  Update columns p and q
  Copy columns p and q on rows p and q
  Iterations = Iterations + 1
endwhile

Figure 1: General form of a sequential Jacobi algorithm for a $n \times n$ matrix

5 Parallel Jacobi Algorithm

5.1 Parallel Jacobi rotations

From the previous subsection we note that a Jacobi rotation affects only the elements in the p, q columns and rows for annihilating the element (p, q) of

A. Furthermore one can easily prove that $J_{pq}J_{p'q'} = J_{p'q'}J_{pq}$ if p, q, p', q' are all distinct. These features of the Jacobi method make it possible to annihilate more than one element at a time. Since each rotation affects two columns and rows, the maximum number of the rotations which can be performed simultaneously is $\frac{n}{2}$. Our parallel Jacobi method consists in doing concurrent Jacobi rotations at each of the processors of our computing system.

We resume our parallel Jacobi algorithm for each processor in Figure 2.

As noted in [Sam71], all the processors in a parallel machine can and should do their own Jacobi rotations at the same time. In concurrent rotations the transformations are done on the original columns. Each rotation (p, q) affects the columns and rows p, q. One must therefore correct for the elements in the rows p, q on the other processors. Because of the commutativity of Jacobi rotations mentioned above, the corrections may be done after each set of concurrent rotations. We just have to store the parameters and the indices of the rotations done to update the data located in the other processors at the end of a sweep.

In our parallel implementation, we store the matrix by entire columns distributed in each processors. It is obvious that we can gain half of the memory space by storing only half of the symmetric matrix. But this would cause an extra amount of communications at each column update to find out the necessary elements distributed in other processors. With our parallel implementation, there is two times more update computations and memory use compared to the sequential implementation. But we do not have any constraints for memory size with the range of matrix treated. Furthermore, as a computation operation is far more cheaper than a communication operation, we prefer the increase in the computation operations than an increase in communication operations. This choice is valid for small size problems ($N < 512$) where the $\theta(N^2)$ communications cost the same as the $\theta(N^3)$ computations.

While Iterations $<$ NBMAXITER **and** off(A) $> \varepsilon$ **do**
  for stage = 1 **to** n **do**
    for i = 1 **to** $\frac{n}{2P}$ **do**
      if $(a_{p_i q_i} > \varepsilon)$ **then** Execute Jacobi rotation(i)
    endfor
  Communicate the $\frac{n}{2P}$ rotation parameters
  Update columns
  Shuffle columns
  endfor
  Iterations = Iterations + 1
endwhile

Figure 2: Parallel Jacobi algorithm for each processor.

5.2 Shuffling of matrix columns

In this subsection, we show how to shuffle the matrix columns during each sweep in order to complete it. We have seen in the previous subsection that it is necessary to annihilate every off diagonal element of the matrix. This requires to do the rotations between all possible pairs of columns in the matrix. Thus we have to permute the various columns so that with a complete sequence of shuffling of columns and concurrent rotations, we complete a full Jacobi sweep.

We give a definition of the parallel ordering:

$$(i_1, j_1), (i_2, j_2), \ldots, (i_q, j_q) \ with\, q = \frac{n(n-1)}{2}$$

is a parallel ordering of the set $\{(i,j)|1 \leq i \leq j \leq n\}$ if for $s = 1 : n-1$ the rotation set

$$Rot(s) = \{(i_r, j_r)|r = 1 + n(s-1)/2 : ns/2\}$$

consists of non conflicting rotations [GL90].

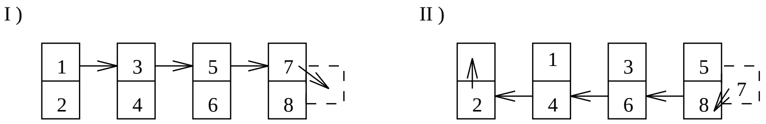

Figure 3: Communication schema for caterpillar-track parallel ordering

Most of the empirical results for parallel Jacobi-type algorithms that are found in the literature use odd-even ordering. Convergence has been proved on some cases, but, only for odd-even orderings or the orderings equivalent to odd-even one. The use of various orderings does make a difference in convergence rates, which in some instances is quite striking [ME93]. But, quadric convergence is always observed in real symmetric cases. Convergence for symmetric matrices has been proven by Forsythe and Henrici [FH60] for column-cycling ordering. Luk and Park [LP89b] proved the convergence for odd-even Jacobi sets by proving it is equivalent to column-cycling orderings. The odd-even ordering, however, is not optimal for parallel computation in that it completes a sweep in n sweeps instead of $(n-1)$.

We use the caterpillar-track ordering [Ebe86, EP90] which which is identical to the odd-even ordering [LP89a]. In Figure 4, we show this parallel ordering for $n = 8$. We embed a ring in the hypercube and do the communications through this ring. At each stage k of the ordering, a processor p, according to k, sends a column to a neighbor ($p+1$ or $p-1$ whether k is even or odd) and receives a column from the another neighbor [EP90]. The communication schema is shown in Figure 3. Each block represents a processor. The numbers in each block are the column numbers housing in this processor and the arrows indicate the communication at each stage. We only consider the case with an even number of matrix columns. The case with odd number of columns follows trivially by adding a dummy column to obtain the even case. Thus we need n stages to complete the sweep because half of the stages are performing $p-1$ rotations.

stage 1	(1,2)(3,4)(5,6)(7,8)
stage 2	2(1,4)(3,6)(5,8)7
stage 3	(2,4)(1,6)(3,8)(5,7)
stage 4	4(2,6)(1,8)(3,7)5
stage 5	(4,6)(2,8)(1,7)(3,5)
stage 6	6(4,8)(2,7)(1,5)3
stage 7	(6,8)(4,7)(2,5)(1,3)
stage 8	8(6,7)(4,5)(2,3)1

Figure 4: One sweep of caterpillar-track ordering for $n = 8$.

5.3 Gossiping

We define the gossiping[1] as the communication procedure which sends from each processor a distinct message of length L to every other processor and receives messages of length L respectively from all the other processors.

At each sweep we execute at most $\frac{n}{2}$ rotations. Each processor executes $\frac{n}{2P}$ or $\frac{n}{2P} - 1$ rotations at every stage (each processor owns $\frac{n}{P}$ or $\frac{n}{P} - 1$ columns and there are two columns necessary to execute a rotation). Before beginning the following sweep, we have to update the matrix with the $\frac{n}{2} - \frac{n}{2P}$ rotations executed in the other processors.

Each processor sends the informations about its rotations to all other processors, receives the informations about other rotations from all the other processors and updates its columns. Hence, at each sweep we have to do a all-to-all communication procedure. This is the communication procedure with the biggest cost in our algorithm. In the next subsections, we show how we decrease this communication cost by overlapping the gossiping communications with computations.

5.4 Parallel version using Intel gossiping procedure

Figure 2 shows the algorithm on a hypercube network. This schema is the same for all the studied versions. The difference between these versions is essentially the gossiping communication procedure. Henceforth, we will discuss only these differences in the following subsections.

[1]The *gossiping* communication procedure is also referred as *all to all* or *total exchange*.

In the first version we use the Intel gossiping procedure (*gcolx*) for gossiping[2]. We obtain very good performances. We believe that it is due to the efficient use of machine low level characteristics.

Hereafter, we use this version as a reference to show the gain provided by the overlapped communication procedures.

5.5 Parallel version with hand coded gossiping procedure

Our motivation was a hand coded gossiping procedure which would be more efficient than the vendor gossiping procedure and that could be overlapped with computation. We follow the works of [Fra90, JH89] for the implementation of our gossiping procedure on a hypercube network.

me = $mynode()$
for $i = 0$ **to** d **do**
  Destination = me xor 2^i
  Exchange message of length $L * 2^i$ with the Destination processor
endfor

Figure 5: Gossiping algorithm for a message of length L on a hypercube network of dimension d.

The gossiping procedure for hypercubes is described in Figure 5. Its cost for messages of length L through a $d = log_2(P)$ dimension hypercube, takes into account that the length of the messages exchanged double at each step:

$$T_{comm} = d\alpha + \sum_{i=0}^{d-1} 2^i L\beta = d\alpha + (2^d - 1)L\beta = log_2(P)\alpha + (P-1)L\beta$$

5.6 Parallel version with overlapped gossiping procedure

In this subsection we describe how the execution time decreases while we overlap the communications by the computations.

In Figure 6, we show the principles of our overlapped gossiping communication procedure. We remark that this procedure can be used for other applications requiring a gossiping as a communication procedure during their execution. The user just has to change the *Update* procedure by any computation procedure to take advantage of the pipeline overlap effect.

This procedure does not have any consequences on the convergence speed of our parallel algorithm, because, there is no change on the computation sequence

[2] We use the name *gossiping* for the Intel procedure (*gcolx*) because it has the same functionality. This name is not used in the Intel documentations.

```
me = mynode()
Destination = me xor 2^0
Exchange message of length L with the Destination processor
for i = 1 to d − 1 do
  Destination = me xor 2^i
  Do Parallel
   Exchange asynchronously a message of length L ∗ 2^i
     with processor Destination
   Update columns for the message received at step i − 1
  enddo
endfor
Update columns for the message received at step d − 1
```

Figure 6: Gossiping procedure with overlapping of some Jacobi update computations.

nor on the ordering but only a decrease of the communication overhead at each sweep.

6 Theoretical study of the complexities

In this subsection we study the complexity of our algorithm. Remark that if we use the Intel gossiping procedure for reference version in our experiments, we do not know the algorithm that is used by Intel in its gcolx communication procedure, hence, it is impossible for us to give its communication complexity.

Notice also, that the computation part is the same for all the parallel versions implemented, so the computation complexity study is valid for all the parallel versions described.

6.1 Serial algorithm

There is a part of computation independent from n concerning the computation of rotation angle Θ_{pq} and the update of the elements a_{pq}, a_{pp} and a_{qq} at each rotation. As in [GL90], we assume this amount of computation is constant, let it be C^{te} flops (for our implementation $C^{te} \approx 53$ flops). As we mentioned in subsection 4, when executing the rotation (p, q), only rows and columns p and q are altered. Then the update $A = J_{(p,q)} A J^T_{(p,q)}$ can be implemented in $6n$ flops if the symmetry is exploited for every rotation. There are $\sum_{i=1}^{n} \sum_{j \geq i}^{n} j = \frac{n(n-1)}{2}$ rotations in each sweep to annihilate all non-diagonal elements. In our algorithm (see Figure 2), we define the $off(A)$ to find out the convergence of the algorithm.

It costs about $2n$ flops. As we see on Figure 2, we execute the rotation only when a_{pq} is greater than ε. This is not always the case, specially for the last sweeps. Therefore the maximum computation complexity for each sweep k is:

$$\begin{aligned} max(T^k_{comp}) &= \frac{n(n-1)}{2}(6n + C^{te}) + 2n \\ &= 3n^3 + n^2(\frac{C^{te}}{2} - 3) + n(2 - \frac{C^{te}}{2}) \end{aligned}$$

There is no rigorous theory that enables one to predict the number of sweeps [GL90]. But Brent and Luk have argued heuristically that the number of sweeps is proportional to $log_2(n)$. Therefore the total computation complexity is $O(log_2(n)n^3)$ flops.

6.2 Parallel algorithm

6.2.1 Computation Complexity

We remind that the computation complexity for the sequential version is the half of the parallel version (see subsection 5.1), because the sequential version profits of the matrix symmetry which is not the case for the parallel versions. The computation load is the same for each processor because at the end each one will have done the same number of updates. Therefore the maximum computation complexity, for each sweep k and for each one of the p processors is:

$$max(T^k_{comp}) = 6\frac{n^3}{P} + \frac{n^2}{P}(C^{te} - 6) + \frac{n}{P}(4 - C^{te})$$

Hence the total execution time for parallel algorithm is $O(\frac{log_2(n)n^3}{P})$ flops.

6.2.2 Communication Complexity

As one can see in Figure 2, there are two main parts needing communications with other processors at each step. First, when we shuffle columns across the processors and second when we gossip the informations concerning the rotations through all processors. Then the communication time is $T_{comm} = T_{shuffle} + T_{gossip}$. In the shuffle case, each processor communicates only with its neighbors (see subsection 5.2). It sends a column to the next neighbor on the ring and receives asynchronously another column from the preceding neighbor on the ring. So the communication time $t_{shuffle}$ for each step is the time to send a column to the next neighbor (the overcost of the asynchronous receive during the send is neglected). Then, for each step, $t_{shuffle} = \alpha + n\beta$ (see subsection 3). There are n steps at each sweep k, so $T^k_{shuffle} = \alpha n + n^2\beta$. The other part requiring communication is the gossiping of the rotation parameters through the hypercube.

With our hand coded algorithm for gossiping on a hypercube, the cost for a message of length L is $log_2(P)\alpha + (p-1)L\beta$ (see subsection 5.5). In our Jacobi algorithm, the length of the message to gossip is $\frac{2n}{p}$ which leads to $t_{gossip} = log_2(P)\alpha + 2n\frac{P-1}{P}\beta$ for each step. There are n steps at each sweep k, so $T^k_{gossip} = log_2(P)n\alpha + 2n^2\frac{P-1}{P}\beta$.

Then the total communication time for each sweep k is:

$$T^k_{comm} = n\alpha + n^2\beta + log_2(p)n\alpha + 2n^2\frac{p-1}{p}\beta$$

which leads to:

$$T^k_{comm} \quad = \quad n\alpha(1 + log_2(p)) + n^2\left(1 + 2\frac{p-1}{p}\right)\beta$$

As we have seen in subsection 6.1, there are heuristically $log_2(n)$ sweeps before convergence. So the total communication complexity for the parallel version without overlapping is $T_{comm} = O(3n^2 log_2(n)\beta + n log_2(n)(1 + log_2(p))\alpha)$.

The shuffle time $T_{shuffle}$ stays the same in both versions, overlapped and no overlapped. But the gossiping time T_{gossip} decreases dramatically in the overlapped version. Roughly, we have $T^{over}_{comm} = T_{shuffle}$ and $T^{noover}_{comm} = T_{shuffle} + T_{gossip}$.

7 Experimental results

In this subsection, we comment our experimental results that have been done on an iPSC/860 hypercube. We use the level 1 BLAS subroutines [DCHH88] to update the columns on sequential and parallel versions. In general, the performances of these subroutines increase with the size of the data treated. This implies some performance gain for longer vectors.

In Figure 7, we present the speedup for all versions. Remark the the computation time for the sequential version is the half of the parallel version but as the computation routines perform better on longer vectors, the computation efficiency is better in the parallel case where the vector are two times longer. Thus, speedup results better than $P/2$ are obtained before the full speed is attained by the sequential version and then they decrease to the $P/2$ asymptote.

As we see in Figure 9, the efficiency is always better for the overlap version. Since the computation complexity is the same in all cases, the explanation is the gain on the communication time, realized by the overlapping the communications by the computations. The efficiency is a decreasing function of the number of processors because the gossiping communication time increases

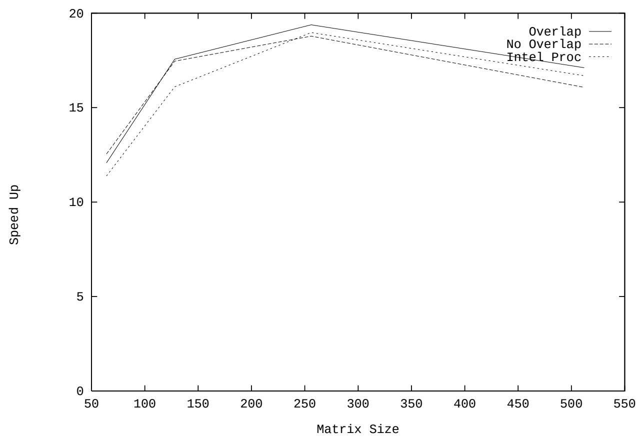

Figure 7: Speed up for the overlap, the no overlap and the Intel versions of the Jacobi procedure according to the matrix size, using 32 processors.

when the number of processors grows (see subsection 5.5). Notice that the difference between overlapped and the non-overlapped version also increases with the number of processor.

In Figure 10, we present the percentage of the gain for the overlap version compared to the no overlap version. We obtain this percentage with the ratio $\frac{T_{Noover}-T_{Over}}{T_{Noover}}$, where T_{Over} and T_{Noover} are respectively the total execution time on the overlap and the no overlap versions. When the size of the matrix is small (< 120) the number of columns per processor is too small to allow the overlapping of the communications by the computations. In this case, the overhead that the overlapping induces results in a longer execution time for the overlap version compare to the no overlap version.

The percentage of the gain increases with the matrix size. There are several reasons for this. The first one is that the performances of the level 1 BLAS subroutines increase with the matrix size. As T_{comp} is decreasing, the ratio $\frac{T_{comm}}{T_{comp}}$ augments. This leads to a relative greater importance of the overlapped part of T_{comm} versus the total execution time. The second one is explained in subsection 6.1. After several sweeps the number of non diagonal elements greater than the threshold decreases and becomes very small when the convergence approaches. Thus the computation time diminishes while the communication time remain the same, so this results in another increase of the ratio $\frac{T_{comm}}{T_{comp}}$.

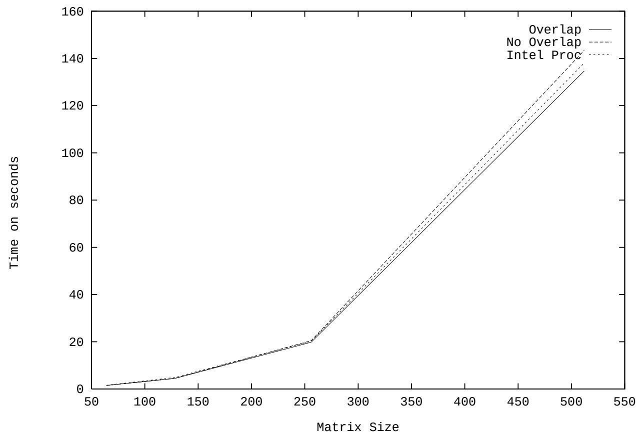

Figure 8: Execution times in seconds for the overlap, the no overlap and the Intel versions of the Jacobi procedure according to the matrix size, using 32 processors.

8 Conclusion and future works

We presented the overlap of the gossiping communications by the computations in the Jacobi algorithm. The cost of this type of global communication scheme increases with the matrix size and the number of processors and is thus very important to overlap from the scalability point of view.

We showed that the overlapping implementation can lead to non-negligible improvements. Moreover, this is achieved on our target hardware which has only one asynchronous communication port. Regarding the communication strategy employed, we guess on better improvements if the hardware were able to handle multi-ports asynchronous communications.

We intend to overlap the shuffle communications and try to take advantage of the matrix symmetry to reduce the computation time. Finally, we are working on a more general version of our gossiping procedure to include it in the LOCCS library [DT92].

We run several tests on the Paragon machine. These first experiments are encouraging but we will have to redesign our hand coded gossiping procedure to squeeze the most out of the grid topology.

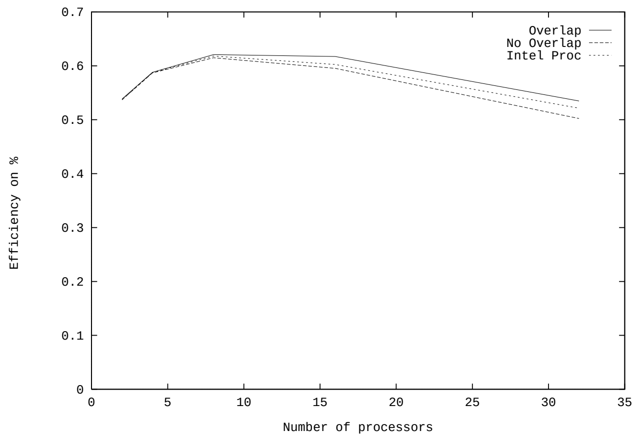

Figure 9: Efficiency for the overlap, the no overlap and the Intel versions of the Jacobi procedure according to the number of processors on a 512×512 matrix.

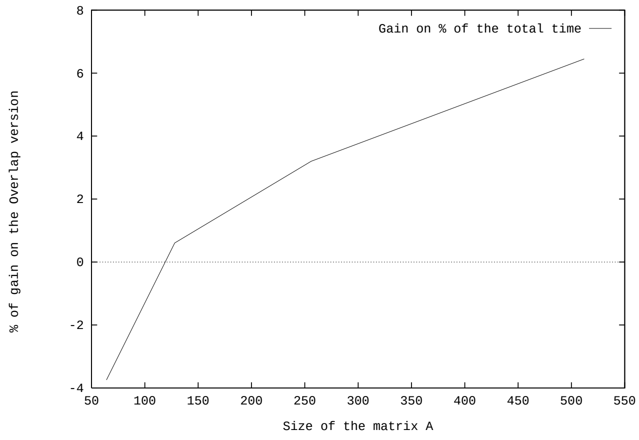

Figure 10: Percentage of the gain on the Overlap version compared to the No overlap version of the Jacobi procedure, using $p = 32$ and according to the matrix size.

References

[DCHH88] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subroutines. *ACM Transaction on Mathematical Software*, 1(14):1–17, March 1988.

[DS86] J. Dongarra and DC. Sorensen. Linear algebra on high performance computers. *Parallel Computing*, 85:221–236, 1986.

[DT92] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. Technical Report 92-44, Laboratoire d'Informatique du Parallélisme-ENSL, December 1992.

[Dun90] T.H Dunigan. Performance of the intel iPSC/860. Technical Report TM-11491, Oak Ridge National Laboratory, June 90.

[Ebe86] P. J. Eberlein. Comments on some parallel Jacobi orderings. Technical Report 86-16, Dept. Comp. Sci., State University of New York at Buffalo, 1986.

[Ebe87] P. J. Eberlein. On one-sided Jacobi methods for parallel computation. *SIAM J. ALG. DISC. METH.*, 8(4):790–796, October 1987.

[EP90] P. J. Eberlein and H. Park. Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory arrchitectures. *Journal of Parallel and Distributed Computing*, (8):358–366, 1990.

[FH60] G. E. Forsythe and P. Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Trans. Amer. Math. Soc.*, 94:1–23, 1960.

[Fou89] David E. Foulser. A blocked Jacobi method for the symmetric eigenproblem. Technical Report RR-680, Dept. of Computer Science, Yale University, February 1989.

[Fra90] P. Fraigniaud. *Communications intensives dans les architectures à mémoires distribuées et Algorithme parallèle pour la recherche de racines de pôlynomes*. PhD thesis, Ecole Normale Superieure de Lyon, December 1990.

[GL90] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins University Press, 1990. 2nd edition.

[Int90] Intel Corporation. *iPSC/860 User's Guide*, June 1990.

[Jac46] C.G.J. Jacobi. *Uber ein leichtes Verfahren*. 1846.

[JH89] S. Johnsson and C. T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comp.*, 38(9):1249–1268, 1989.

[LP89a] F. T. Luk and H. Park. On parallel Jacobi orderings. *SIAM J. SCI. STAT. COMPUT.*, 10(1):18–26, January 1989.

[LP89b] F. T. Luk and H. Park. A proof of convergence for two parallel Jacobi SVD algorithms. *IEEE Trans. Comput.*, 38(6):806–811, June 1989.

[ME93] M. Mantharam and P. J. Eberlein. New Jacobi-sets for parallel computations. *Parallel Computing*, 19:437–454, 1993.

[MM91] C.L. McCreary and M.E Mcradle. Modeling communication delay on the iPSC/2 and iPSC/860 hypercubes. Technical Report CSE-91-12, Aubran University, September 1991.

[MMM91] C.L. McCreary, M.E. Mcardle, and J.D. McCreary. Broadcast communication delay metric for iPSC/2 and iPSC/860 hypercubes. July 1991.

[Mod88] J. J. Modi. *Parallel Algorithms and Matrix Computation*. OXFORD: CLARENDON Press, 1988.

[PT93] M. Pourzandi and B. Tourancheau. Overlapping in Gaussian elimination on iPSC/860. In M. A. Yaghoubi, editor, *Proceeding of International Congress on Computational Methods in Engineering*, volume 4, pages 183–193, University of Shiraz, Iran, May 1993.

[Sam71] A. Sameh. On Jacobi and Jacobi-like algorithms for a parallel computer. *Math. Comp.*, (25):579–590, 1971.

[SB77] H. Sulivan and T.R. Bashkow. A large scale, homogeneous, fully distributed parallel machine. In *Proceeding of the fourth Symposium on computer architectures*, pages 105–117, 1977.

[TMLZ93] C. Trefftz, P. K. McKinley, T. Y. Li, and Z. Zeng. A scalable eigenvalue solver for symmetric tridiagonal matrices. In R. Sincovec, D. E. Keyes, M. R. Leuze, L. R. Petzold, and D. A. Reed, editors, *Proceedings of the sixth SIAM Conference on Parallel Processing*, volume 2, pages 602–609, 1993.

[TY91] P. Tervola and W. Yeung. Parallel Jacobi algorithm for matrix diagonalization on transputer networks. *Parallel Computing*, (17):155–163, 1991.

[Wil65] J. H. Wilkinson. *The Algebric Eigenvalue Problem*. OXFORD: CLARENDON Press, 1965.

9 Appendix : Impact of the threshold

In this appendix, we discuss the decrease in the number of rotations computed during each iteration of the Jacobi method. Notice that there are little theoretical results concerning the number of rotations at each iteration before convergence with the threshold Jacobi method. We try to give some hints to gain time on the gossip procedure.

We treat only the dense matrices which elements are randomly chosed and so they have not any particular structure. Hence, our experiences are not representative of the class of dense matrices. However, we found out that the number of rotations decrease dramatically in the last iterations before convergence.

The data gossiped at each step concern only the rotations angles. One can use the decrease on the number of rotations to diminish the amount of data gossiped. Remark this is very difficult to predicate the number of elements superior than the threshold which will be rotated and the processors holding them. This implies practically a stage of pre-treatment in the beginning of the gossip procedure to inform each processor about the amount of data to be received (it consists mainly of a gossip-type communication procedure). The problem now is to know if the gain on the gossip procedure justify the cost of the pre-treatment ?

In Figure 11, we show the number of rotations per iteration on a 256×256 test matrix. As one can remark the number of rotations decreases dramatically only in the last iterations. We conclude that it is not necessary to pre-treat the gossip procedure in the first iterations. Unfortunatly, in our knowledge, there is no theoretical results to indicate exactly how many iterations have to be computed before convergence. Therefore, it is very difficult to determine which iterations have to be pre-treated. Empirically, with our test matrices (random, of order $64 \leq n \leq 512$), the pre-treatment of the gossip procedure is worthwhile after the $(log_2(n) - 2)^{th}$ iteration.

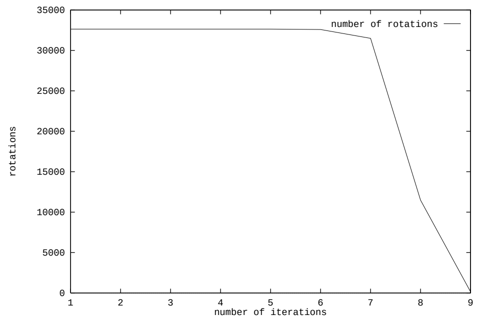

Figure 11: Number of rotations according to the number of iterations.