Parallel Implementation of a Symmetric Eigensolver Based on the Yau and Lu Method *

Stéphane Domas[1], Françoise Tisseur[2],

[1] Laboratoire de l'Informatique du Parallélisme, URA 1398 du CNRS and INRIA Rhône-Alpes, 46 Allée d d'Italie, 69364 Lyon Cedex 07, France. `sdomas@lip.ens-lyon.fr`

[2] Equipe d'Analyse Numérique de St-Etienne,UMR 5585, 23, rue Paul Michelon, 42023 Saint-Etienne, France. `ftisseur@anumsun1.univ-st-etienne.fr`

Abstract. In this paper, we present preliminary results on a complete eigensolver based on the Yau and Lu method. We first give an overview of this invariant subspace decomposition method for dense symmetric matrices followed by numerical results and work in progress of a distributed-memory implementation. We expect that the algorithm's heavy reliance on matrix-matrix multiplication, coupled with FFT should yield a highly parallelizable algorithm. We present performance results for the dominant computation kernel on the Intel Paragon.

1 Introduction

As quantitative analysis becomes increasingly important in sciences and engineering, the need for faster methods to solve bigger and more realistic problem grows. Large order symmetric eigenvalues 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.

There are many algorithms for solving the symmetric eigenvalue problem [13]. Much recent work has been devoted on parallel solvers, both on traditional methods [8, 9, 10] and in the development of new methods [3, 7]. The traditional method for computing the eigensystem of a real dense symmetric matrix A consists in three steps [11]. First, A is reduced to tridiagonal form. Second, the eigenvalues and eigenvectors of the tridiagonal matrix are computed. Third, the eigenvectors are back transformed via the reduction transformation.

In this paper, we investigate the parallelization of a new eigensolver for real dense symmetric matrices. Our algorithm is based on a recent method attributed to Yau and Lu [16], which reduces the symmetric eigenvalue problem to a number of matrix multiplications. Yau and Lu's method involves approximating invariant subspaces of a special matrix using an FFT. The computation of the special

* This work is partly supported by the European project KIT 108 and Eureka Euro-TOPS project.

matrix and the vectors for the FFT is rich in matrix-matrix multiplications Matrix multiplications can be implemented efficiently on most high-performance machines, and is often available as an optimized implementation in a level 3 BLAS library [2, 4].

The rest of this paper will give an overview of the Yau and Lu's method. We will present the algorithm and valid it with numerical results. Then, we will investigate the parallelization of this new eigensolver and present performance results for the dominant computation kernel on the Intel Paragon.

2 Yau and Lu method

For computing invariant subspaces of a symmetric $n \times n$ matrix A with eigenvalues $\lambda_1, \ldots, \lambda_n$ and eigenvectors $x_1, \ldots, x_n$, Yau and Lu use a polynomial acceleration method.

Consider the unitary matrix $B = e^{iA}$ whose eigenvalues all lie on the unit circle. Note that A and B have the same invariant subspaces.

Let $P_N(z) = \sum_{j=0}^{N-1} \beta_j z^j$ be a polynomial of degree $N-1$ that has a peak at $z = 1$ and is close to zero on the unit circle away from a vicinity of $z = 1$. Such a polynomial exists and its coefficients $\beta_j, j = 0, N-1$ can be obtained by a recursive formula (see [16]).

Starting from an initial vector expanded in terms of the eigenvectors as $v_0 = \sum_{i=1}^{n} \alpha_i x_i$, we can define the function $u : [0, 2\pi) \to I\!R^n$ by

$$u(\lambda) = P_N(e^{-i\lambda} B) v_0 = \sum_{j=1}^{n} \alpha_j P_N(e^{i(\lambda_j - \lambda)}) x_j .$$

If λ is chosen close to a particular λ_k and the other eigenvalues of B are not close to λ then the coefficient of x_j will be small except when $j = k$. Thus, $u(\lambda)$ can be viewed as an approximation of the eigenvector of B associated with the eigenvalue $e^{i\lambda_k}$.

Setting $v_j = B^j v_0$, the function $u(\lambda)$ can be written as

$$u(\lambda) = P_N(e^{-i\lambda} B) v_0 = \sum_{j=0}^{N-1} \beta_j B^j v_0 e^{-ij\lambda} = \sum_{j=0}^{N-1} \beta_j v_j e^{-ij\lambda},$$

where $\beta_j v_j$ are the Fourier coefficients of u. Therefore, the FFT can be used to compute $u(\lambda)$ at many different values of λ simultaneously. Then, we need to select vectors $u(\lambda)$ that can be taken as eigenvectors, group them into p orthogonal clusters and add more vectors if necessary. These p clusters form an orthogonal basis $W = [W_1, \ldots, W_p]$ whose elements span invariant subspaces of A and hence, application of A to W decouples the spectrum :

$$W^T A W = \begin{pmatrix} A_1 & & 0 \\ & \ddots & \\ 0 & & A_p \end{pmatrix} .$$

So, the initial problem is reduced to a small symmetric matrix eigenvalue problem in each cluster. Note that the subproblems $A_1, \ldots, A_p$ can be solved totally independently. The algorithm is presented in next section.

3 Numerical algorithm.

Consider a real symmetric matrix A. The following steps find the eigenvalues and eigenvectors of A to the desired precision.

1– **Scaling and Translation:** Compute upper and lower bounds of the spectrum of A and use these bounds to scale and translate the spectrum of A in $[0, 2\pi)$.
2– **Polynomial computation:** Let T_{N-1} be the Chebyshev polynomial of degree $N-1$. The degree N is chosen such that

$$\frac{1}{T_{N-1}\left((3-\cos\frac{\pi}{n})/(1+\cos\frac{\pi}{n})\right)} \leq \kappa,$$

where κ is a measure of the desired accuracy of the computed invariant subspace. Compute the coefficients $\beta_0^{(N-1)}, \ldots, \beta_{N-1}^{(N-1)}$ of the polynomial P_N by the recursive formulas :

$$\begin{aligned}
\beta_0^{(0)} &= 1 \ , \ \beta_0^{(1)} = b \ , \ \beta_1^{(1)} = a \\
\beta_0^{(k+1)} &= a\beta_1^{(k)} + 2b\beta_0^{(k)} - \beta_0^{(k-1)} \\
\beta_1^{(k+1)} &= a\left(2\beta_0^{(k)} + \beta_2^{(k)}\right) + 2b\beta_1^{(k)} - \beta_1^{(k-1)} \\
\beta_j^{(k+1)} &= a\left(\beta_{j-1}^{(k)} + \beta_{j+1}^{(k)}\right) + 2b\beta_j^{(k)} - \beta_j^{(k-1)} \text{ for } j > 1 \\
\beta_j^{(k+1)} &= 0 \text{ for } j > k+1.
\end{aligned}$$

where

$$a = \frac{2}{1+\cos(\pi/n)}, \quad b = \frac{1-\cos(\pi/n)}{1+\cos(\pi/n)}.$$

3– **Unitary matrix:** Compute matrices $\cos(\pi X)$ and $X^{-1}\sin(\pi X)$ where $X = \frac{A}{\pi} - I$ using the following Chebyshev expansion :

$$\begin{aligned}
\cos(\pi x) &\simeq c_0 + c_1T_2(x) + c_2T_4(x) + \cdots + c_5T_{10}(x) \\
&\quad + T_{10}\left(c_6T_2(x) + c_7T4(x) + \cdots + c_{10}T_{10}(x)\right) \\
\frac{\sin(\pi x)}{x} &\simeq s_0 + s_1T_2(x) + s_2T_4(x) + \cdots + s_5T_{10}(x) \\
&\quad + T_{10}\left(s_6T_2(x) + s_7T4(x) + \cdots + s_{10}T_{10}(x)\right),
\end{aligned}$$

where $c_0, \ldots, c_{10}, s_0, \ldots, s_{10}$ are Chebyshev coefficients.

4– **Computation of vectors** $v_j = e^{ijA}v_0, j = 0, 2N-1$ The real and imaginary part of v_1 are obtained with the approximation of $\cos(\pi X)$ and $X^{-1}\sin(\pi X)$:

$$v_1 = \cos(\pi X)v_0 + X^{-1}\sin(\pi X)(Xv_0).$$

The remaining vectors can be computed following these $M = (\log_2 N - 1)$ steps:

step 1 : $C_1 = \cos(A)$
$v_2 = 2C_1v_1 - v_0$

step 2 : $C_2 = 2C_1^2 - I$
$(v_3, v_4) = 2C_2(v_1, v_2) - \overline{(v_1, v_0)}$

step 3 : $C_3 = 2C_2^2 - I$
$(v_5, v_6, v_7, v_8) = 2C_3(v_1, v_2, v_3, v_4) - \overline{(v_3, v_2, v_1, v_0)}$

$\vdots$ $\vdots$

step M : $C_p = 2C_{p-1}^2 - I$
$(v_{\frac{N}{2}+1}, \ldots, v_N) = 2C_p(v_1, \ldots, v_{\frac{N}{2}}) - \overline{(v_{\frac{N}{2}-1}, \ldots, v_0)}.$

5– **Evaluation of** $u(\lambda)$**:** Via the FFT, compute the vectors

$$u_k = u(\lambda)_{\lambda=\frac{k\pi}{N}} = Re\sum_{j=0}^{N-1}\beta_j v_j e^{-i\frac{jk\pi}{N}}, \qquad \text{for } k = 0, \ldots, 2N-1.$$

6– **Selection and refinement:** Select the most useful vectors from the $2N$ vectors $u_0, \ldots, v_{2N-1}$. Group them into a number of orthogonal clusters, add more vectors if necessary and reduce the initial problem to a small symmetric matrix eigenvalue problem in each cluster.

The most time consuming part of the algorithm is the computation of the $2N$ vectors v_j. We need $\log_2 N - 1$ multiplications of real symmetric matrices and $\log_2 N - 1$ more multiplications between a symmetric matrix and a rectangular one. This part needs $(\log_2(N) - 1)n^3 + 4Nn^2$ floating point operations. The computation of e^{iA} to the desired accuracy requires 6 multiplications of real symmetric matrices for $C\cos\pi X$ and one more for $S = X^{-1}\sin\pi X$, that is, $7n^3$ operations. The step of computing the u_k is still efficient because of the FFT algorithm and can be done in $nN\log_2(N)$ operations. When necessary, the work for the supplementary vectors involves $\frac{8}{3}n^3$ more operations since we need to construct an orthogonal matrix by a QR factorization. The reduction W^TAW involves two matrix multiplications or $3n^3$ operations. Usually, the subproblems in each cluster only involve small matrices and the cost is negligible compared to the total work. So, the total number of operations is given by

$$(\frac{17}{3} + \log_2(N))n^3 + 4Nn^2 + O(n^2).$$

Since N is typically a small multiple of n, we see from this operation count that the sequential complexity of the Yau and Lu algorithm is considerably greater than the QR algorithm. However, note that steps 3,4 and 6 of the algorithm are all based on matrix-matrix multiplications and if we suppose that $N \simeq 8n$ then

$$\frac{\text{Total operations involving matrix multiplications}}{\text{Total operations}} \geq \frac{35 + 3\log_2(n)}{38 + 3\log_2(n)}.$$

So, for matrices of dimension between 500 and 1000, we find that matrix multiplications account for more than 90% of the total operation count. For larger dimensions of the matrix A, this percentage will of course increase. The efficiency of level 3 BLAS routines can justify the use of the extra multiplications. The Reuse-Ratio defined by the rapport between the number of flops and the size of memory reference bounds the performances. Its value is 2/3 for level 1 BLAS, 2 for level 2 BLAS and n/2 for level 3 BLAS. So, high level BLAS 3 improve performances.

4 Numerical results.

All the test results presented in this section where performed on a SUNsparc 512, MP. The arithmetic was IEEE standard double precision with a machine precision of $\varepsilon = 2^{-53} \simeq 2.22044 \times 10^{-16}$ and over/underflow threshold $10^{\pm 307}$.

We have tested our algorithm on a large set of test matrices using the LAPACK [1] test generation routine DLATMS. This routine constructs symmetric matrices of the form

$$A = U^T D U$$

where U is a random orthogonal matrix and $D = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ a diagonal matrix. We can define the elements of D and then simulate more or less critical situations that is, well separated spectrum, clusters of eigenvalues, ...

We quantified accuracy in the computed eigenvalues by computing the relative error

$$\max_{1 \leq i \leq n} \frac{|\lambda_i - \hat{\lambda}_i|}{|\lambda_{max}|}$$

where λ_i denotes an exact eigenvalue and $\hat{\lambda}_i$ the corresponding computed eigenvalue (see Fig.1).

Accuracy in the residuals for a given matrix A is quantified by computing the maximum normalized 2-norm residual

$$\max_i \frac{||A\hat{x}_i - \hat{\lambda}_i \hat{x}_i||_2}{||A||_F}, \qquad \text{with} \qquad ||\hat{x}_i||_2 = 1$$

where $\hat{x}_i$ is the computed eigenvector corresponding to the computed eigenvalue $\hat{\lambda}_i$ (see Fig.2).

We have computed (see Fig. 3) the departure from orthogonality given by

$$\max_{i,j} |(Q^T Q - I_n)_{ij}|$$

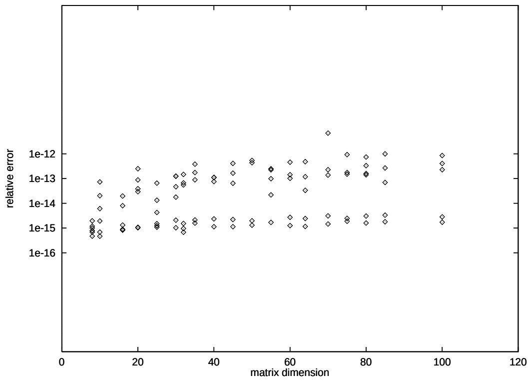

Fig. 1. Relative error on computed eigenvalues.

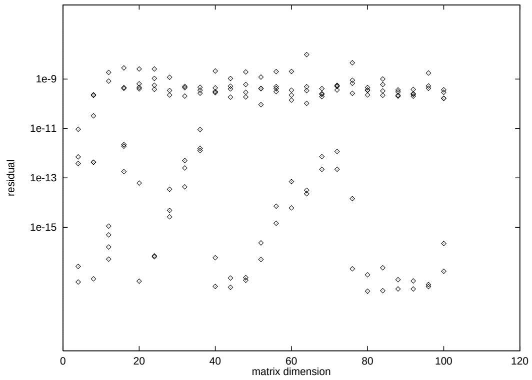

Fig. 2. Residuals for dense symmetric matrices.

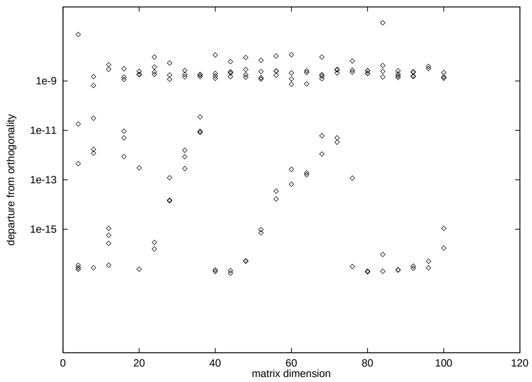

Fig. 3. Departure from orthogonality for dense symmetric matrices.

where Q is the matrix of eigenvectors.

The accuracy of invariant subspaces is controlled in step 2 of the algorithm. As we have chosen N such that $\kappa \leq 10^{-9}$ we expect at most nine corrects significant digits for the eigenvectors. If we choose N such that $\kappa \leq 10^{-16}$, we increase the computational cost but obtain better accuracy (see Fig. 4 and Fig. 5).

5 Parallel implementation.

There are two forms of parallelism in the Yau and Lu method. The first one corresponds to data parallelism with the heavy reliance on matrix-matrix multiplications. The second one is a kind of functional parallelism with the reduction of the initial problem to a number of small symmetric eigenvalue problems that can be solved totally independently on each processor. That is why we say that Yau and Lu method yields to a highly parallelizable algorithm.

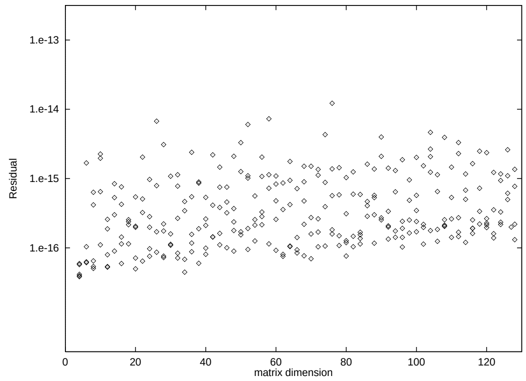

Fig. 4. Residuals on computed eigenvalues for dense symmetric random matrices.

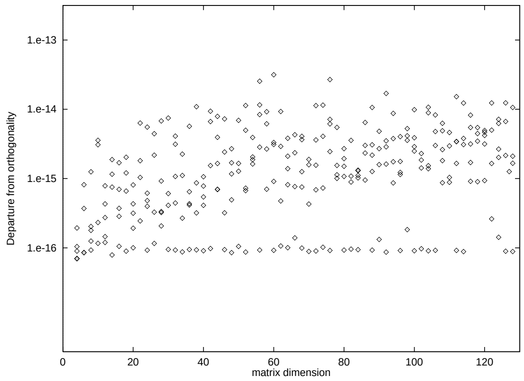

Fig. 5. Departure from orthogonality for dense symmetric random matrices.

5.1 Parallel Yau and Lu algorithm.

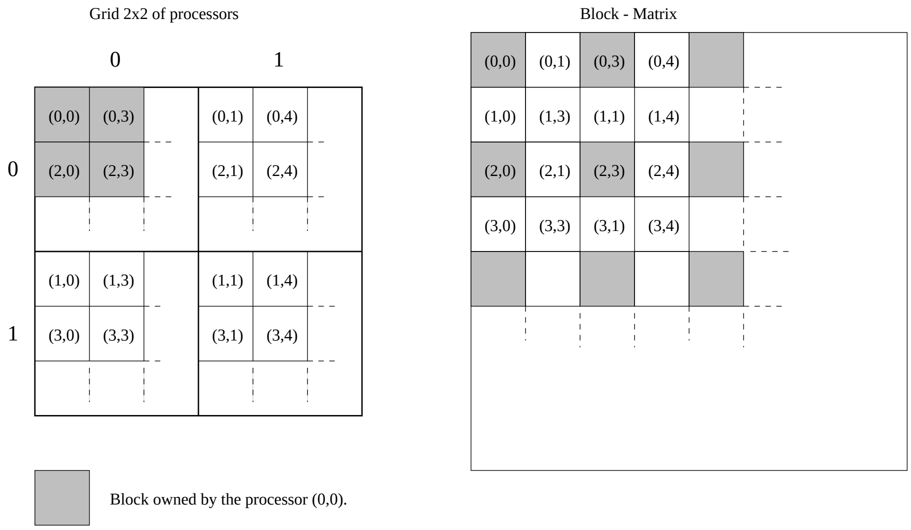

Fig. 6. Block cyclic distribution on a 2x2 processor grid.

For an estimation of the upper and lower bound of the spectrum of A we use a method developed by Rojo and Soto [15]. This method is based on matrix multiplications but does not increase the total number of operations since the computed matrix-matrix product is reused for the construction of C and S.

We prefer to focus our parallelization on the most cost effective part of the algorithm, that is the construction of the matrices C, S and the Fourier's coefficients. In order to ensure a good load balancing, performance and scalability of our code, we use a 2-dimensional block cyclic distribution on a $P \times Q$ processor grid (see Fig.6). This kind of distribution encompasses a large number (but not all) data distribution schemes.

```
/*Construction of T_1, T_2, T_4, T_8, T_10.                          */
T_2 ← 2A * A − I                         /* Parallel level 3 BLAS */
FOR i=4 TO 10 STEP 2
        T_i ← 2 T_{i−2} * T_2 − T_{i−4}    /* Parallel level 3 BLAS */
ENDFOR
/*Chebyshev expansion of C and S                                     */
C ← c_0 I
S ← s_0 I
FOR i=1 TO 5
        C ← c_i T_{2i} + C
        S ← s_i T_{2i} + S
        CC ← c_{i+5} T_{2i} + CC
        SS ← s_{i+5} T_{2i} + SS
ENDFOR
C := −T_10 * CC + C                      /* Parallel level 3 BLAS */
S := −T_10 * SS + S                      /* Parallel level 3 BLAS */
```

Table 1. Algorithm for computing C and S.

In Tab.1, we present the parallel algorithm for the computation of the matrices $C = \cos \pi X$ and $S = X^{-1} \sin \pi X$ which defined the unitary matrix e^{iA}. It is only based on 7 calls of parallel level 3 BLAS.

The computation of Fourier's vectors is the most cost effective part in computation and communication times. At each step $k, k = 1, \ldots, \log_2(N/2)$, the following matrix-matrix multiplication is performed:

$$C_k = 2C_{k-1} * C_{k-1},$$
$$V_k = 2C_k * U_k - W_k,$$

where

$$C_0 - \cos \pi X, W_0 = v_0 \text{ and } U_0 = v_1$$
$$W_k = [\mathrm{perm}(\overline{V_{k-1}}),\ W_{k-1}], \quad U_k = [U_{k-1}, V_{k-1}].$$

The three matrices V_k, U_k, W_k are rectangular ones and their sizes increase from iteration to iteration. The rectangular matrix W_k depends on a permutation of V_k's columns and communications between processor columns are necessary for its construction. For a good load balancing of the computation, we impose the matrices V_k, U_k, W_k to have a column block size partitioning equal to 2. The distribution we use for the computed Fourier's coefficients v_j avoid the necessary communications for the bit-reversal that is the first step of the FFT.We present the parallel algorithm in Tab.2.

For the parallel 1-D FFT [6], we use a communication computation overlap algorithm.

/* Initialization	*/
$W \leftarrow v_0$	
$U \leftarrow Cv_0 + iSv_0$	/* Parallel level 2 BLAS */
$V \leftarrow 2C * U - W$	/* Parallel level 3 BLAS */
/* Main loop	*/
FOR $i = 1$ TO $\log_2(N/2)$ DO	
$\quad C \leftarrow 2C * C - I$	/* Parallel level 3 BLAS */
$\quad W \leftarrow [\mathrm{perm}(\overline{V}), W]$	/* update of W, send/recv between processor col.*/
$\quad U \leftarrow [U, V]$	/* update of W
$\quad V \leftarrow 2C * U - W$	/* Parallel level 3 BLAS */
ENDFOR	

Table 2. Algorithm for computing v_j for $j = 0, \ldots, N-1$.

For the last part of the algorithm, each processor selects the most useful computed vectors u_k and forms a basis. When the number of selected vectors is less than n, we use a parallel QR factorization in order to complete the basis. after projection onto this basis, each processor solves its own small symmetric eigenvalue problem using a standard symmetric eigensolver (for example, DSYEV from LAPACK [1]).

5.2 Symmetric matrix-matrix product.

In Section 3, we have shown that the computational cost of the algorithm is dominated by dense matrix-matrix multiplications. Thus, the performance of this algorithm will depend heavily on the matrix multiplication code. We need two different types of matrix-matrix products. The first one is the product between symmetric and rectangular matrices and the second one is product between two symmetric matrices which commute. So the result will be a symmetric matrix. We want to develop a double precision distributed matrix code for the symmetric matrix product that take into account the special properties of these matrices (which is not currently available in ScaLAPACK [5]).

The algorithm below is based on an idea presented by Snyder in [12]. It uses a block scattered distribution of the matrices. The whole matrices are distributed and not only the upper or lower triangular part. The symbols are the followings. M is the matrix size, distributed on a $P \times Q$ grid of processors. There are $N_b \times N_b$ blocks, and each processor has $P_b \times P_b$ blocks.

$N_b = \lceil \frac{M}{B_s} \rceil$
$P_b = \lceil \frac{N_b}{P} \rceil$
/* Computation of the diagonal blocks of C */
FOR $i = 0$ TO N_b DO
$cur\_row = mod(i, P)$
$cur\_col = mod(i, Q)$
$b_r = \lceil \frac{i}{P} \rceil$
$b_c = \lceil \frac{i}{Q} \rceil$
IF ($my\_col = cur\_col$) THEN
computes the diagonal blocks of C :
DSCMM $\Rightarrow \alpha A^T_{.,b_c} \times B_{.,b_c} + \beta C_{b_r,b_c} \rightarrow C_{b_r,b_c}$
global sum of C_{b_r,b_c} :
DGSUM2D $\Rightarrow$ the result is left on proc. (cur_row, cur_col)
ENDIF
ENDFOR
/* Computation of the blocks of the upper triangular part of C */
FOR $i = 0$ TO $N_b - 1$ DO
$cur\_row = mod(i, P)$
$cur\_col = mod(i, Q)$
$b_r = \lceil \frac{i}{P} \rceil$
$b_c = \lceil \frac{i}{Q} \rceil$
IF ($my\_col = cur\_col$) THEN
DGEBS2D $\Rightarrow$ broadcasts $A_{.,b_c}$ to all processors of the cur_row row
computes the i^{th} block row of the upper triang. part of C :
DGEMM $\Rightarrow \alpha A^T_{.,b_c} \times B_{.,(b_c+1,\ldots,P_b)} + \beta C_{b_r,(b_c+1,\ldots,P_b)} \rightarrow C_{b_r,(b_c+1,\ldots,P_b)}$
global sum of $C_{b_r,(b_c+1,\ldots,P_b)}$:
DGSUM2D $\Rightarrow$ the result is left on proc. (cur_row, my_col)
ELSE
DGBR2D $\Rightarrow$ receives $A_{.,b_c}$
computes the i^{th} block row of the upper triang. part of C :
DGEMM $\Rightarrow \alpha A^T_{.,b_c} \times B_{.,(b_c,\ldots,P_b)} + \beta C_{b_r,(b_c,\ldots,P_b)} \rightarrow C_{b_r,(b_c,\ldots,P_b)}$
global sum of $C_{b_r,(b_c,\ldots,P_b)}$:
DGSUM2D $\Rightarrow$ the result is left on proc. (cur_row, my_col)
ENDIF
ENDFOR
/* transpose and copy the upper triangular part of C in the lower part.*/

Table 3. Symmetric matrix multiplication algorithm.

In the first phase, the diagonal blocks of the matrix C are computed. Only the upper triangular part of each block is computed. For this, a FORTRAN subroutine (`SCMM`) has been developed since no level 3 BLAS subroutine exists to achieve such a computation. On the Intel Paragon, this compiled subroutine is as efficient as the optimized level 3 BLAS subroutine.

Since the matrices are symmetric, each diagonal block is the product of two block columns that are distributed on the same column of processors. But after the multiplication, each processor has a part of the result. Then, a global sum on this column of processor gives the total result. For example, on a 2×3 grid, the C_{11} block is the product of column $A_{.1}$ by the column $B.1$ and these two columns are distributed on processors 1 and 4. The result of the global sum is left on processor 4 which owns the C_{11} block.

The second phase is hardly different from the first. It computes the remaining blocks of the upper triangular part of C. Therefore, the A and B block columns to multiply are not always on the same processors. For example, on a 2×3 grid, C_{13} is the product of $A_{.1}$ which is distributed on processors 1 and 4, and $B_{.3}$ which is on 0 and 3. Consequently, the $A_{.i}$ block column has to be broadcast and multiplied by $B_{.,(i,\dots,N_b)}$ to compute $C_{i,(i,\dots,N_b)}$. As in the first part, each processor has a partial result after the multiplication and a global sum is needed.

The last step consits to transpose the strictly upper triangular part of C in order to obtain the full matrix.

The Fig. 7 shows a comparison between the `PDGEMM` routine that computes a full matrix multiplication and our routine. In solid line, this is the time in seconds taken by `PDGEMM` for different matrix sizes. In dashed line, this is the same for our routine. In dotted line, this is the division of the two times. We can see that our code is very efficient for large sizes. The ratio between a complete and a symmetric product can even reach 2. With smaller matrix sizes, the gain decreases. A preliminary theoretical analysis shows that a ratio of 2 is not possible for small matrix sizes. This is due to the block scattered distribution. Each processor has not exactly the half of the computation to achieve a symmetric product. But this ratio is above 1.5 most of the time whatever the matrix size.

5.3 Implementation on the Intel Paragon.

All the tests have been done on an Intel Paragon with 30 nodes. Each node is composed of two i860, one for the computations and one for the communications. The nodes are connected by a bidirectional 2d-torus that allows a sustained bandwidth of 69 Mbytes and a latency of 60μs.

Preliminary results on the Paragon have been obtained (see Fig. 8 and Fig. 9). Measures concern the main computational kernel of the code with the matrix-matrix product **pdgemm** of ScaLAPACK.

Because of the large amount of memory needed for the algorithm, the maximum problem size is 256 on one processor and 512 on 4. Even if we have not yet incorporated our symmetric matrix-matrix product in the code, we obtain an efficiency close to 1 and speed-up close to 4 for middle problem size (256) on 4 processors. For smaller problem sizes (50-100), the speed-up stays above 2.

Fig. 10 shows a comparison of the execution times between the ScaLAPACK routine PDSYEVX and the main computational kernel of our code. The routine PDSYEVX [8] is based on a bisection method followed by inverse iterations. We

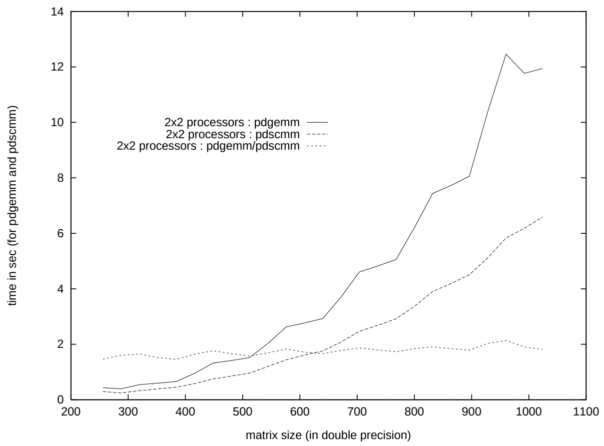

Fig. 7. Comparison between PDGEMM and symmetric matrix multiplication performances.

conclude that our code may be competitive with the bisection method for the computation of all the eigenvalues and all the eigenvectors.

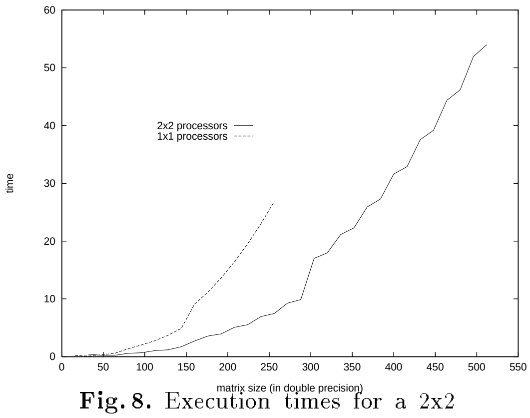

Fig. 8. Execution times for a 2x2 grid according to the size of the matrix on Paragon.

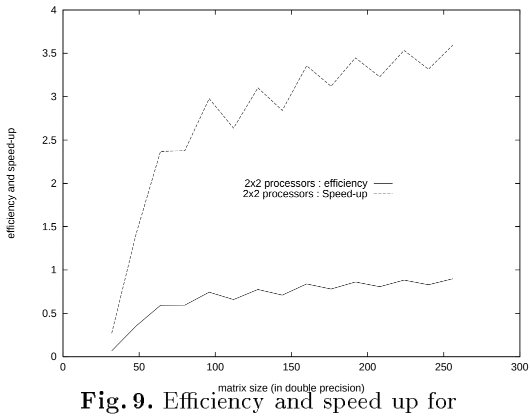

Fig. 9. Efficiency and speed up for a 2x2 grid according to the size of the matrix on Paragon.

6 conclusion

We studied a new approach to compute the eigenvalues and eigenvectors of a real symmetric matrix. The algorithm parallelized in this paper is not efficient on a

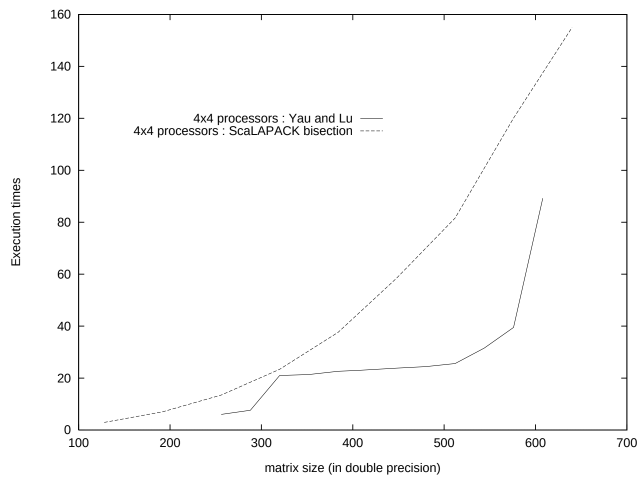

Fig. 10. Execution times for PDSYEVX and Yau & Lu on a Paragon.

sequential machine but can fully take advantage of parallel machines because of less data dependences and the use of matrix products as the most important computed kernel. We obtain good performances for our code concerning efficiency and execution time. Theoritical study [14] showed that our code should scale nicely on parallel machines with a very large number of processors. Of course, the experiments carried out with a grid of 2×2 processors are not conclusive in this respect. That why we want to test our code on large size problems. Future target machine are the SP2 and the T3D.

Acknowledgment I would like to thank the referees and F. Desprez for their helpful comments and suggestions.

References

1. E. Anderson, Z. Bai, C. H. Bischof, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide, Release 2.0.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 1995.
2. Christian Bischof, William George, Steven Huss-Lederman, Xiaobai Sun, Anna Tsao, and Thomas Turnbull. SYISDA users' guide, Version 2.0. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA, 1995.
3. Christian Bischof, Steven Huss-Lederman, Xiaobai Sun, Anna Tsao, and Thomas Turnbull. Parallel performance of a symmetric eigensolver based on the invariant subspace decomposition approach. In *proceedings of Scalable High Performance Computing Conference'94, Knoxville, Tennessee*, pages 32–39, May 1994. (also PRISM Working Note #15.

4. Jaeyoung Choi, James Demmel, I. Dhillon, Jack J. Dongarra, Susan Ostrouchov, Antoine P. Petitet, k. Stanley, David W. Walker, and R. Clint Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performances. LAPACK Working Note 95, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 1995.
5. Jaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. Technical Report CS-92-181, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, November 1992. LAPACK Working Note 55.
6. C.W. Cooley and J.W. Tuckey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, 1965.
7. J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.
8. James W. Demmel and K. Stanley. The performance of finding eigenvalues and eigenvectors of dense symmetric matrices on distributed memory computers. Technical Report CS-94-254, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, September 1994. LAPACK Working Note 86.
9. J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. Stat. Comput.*, 8:139–154, 1987.
10. D. Giménez, R. van de Geijn, V. Hernández, and A. M. Vidal. Exploiting the symmetry on the jacobi method on a mesh of processors. In *4th EUROMICRO Worshop on Parallel and Distributed Processing, Braga, Portugal*, 1996.
11. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, USA, second edition, 1989.
12. C. Lin and L.Snyder. A matrix product algorithm and its comparative performance on hypercubes. In *Scalable High Performance Computing Conference SHPCC92*, pages 190–193. IEEE Computer Society, 1992.
13. Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1980.
14. Makan Pourzandi and Françoise Tisseur. Parallèlisation d'une nouvelle méthode de recherche de valeurs propres pour des matrices réelles symétriques. Report TR94-37, LIP, ENS Lyon, 1994.
15. Oscar Rojo and Ricardo L. Soto. A decreasing sequence of eigenvalue localization regions. *Linear Algebra and Appl.*, 196:71–84, 1994.
16. Shing-Tung Yau and Ya Yan Lu. Reducing the symmetric matrix eigenvalue problem to matrix multiplications. *SIAM J. Sci. Comput.*, 14(1):121–136, 1993.