INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
*Int. J. Numer. Meth. Engng* 2005; **63**:2069–2085
Published online 28 April 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/nme.1341

# Parallel eigenanalysis of multiaquifer systems

L. Bergamaschi[1], G. Pini[1] and F. Sartoretto[2,*,†]

[1]*Dipartimento di Metodi e Modelli Matematici, per le Scienze Applicate, Università degli Studi, Padova, Italy*
[2]*Dipartimento di Informatica, Università di Venezia, Italy*

SUMMARY

Finite element discretizations of flow problems involving multiaquifer systems deliver large, sparse, unstructured matrices, whose partial eigenanalysis is important for both solving the flow problem and analysing its main characteristics.

We studied and implemented an effective preconditioning of the Jacobi–Davidson algorithm by FSAI-type preconditioners.

We developed efficient parallelization strategies in order to solve very large problems, which could not fit into the storage available to a single processor.

We report our results about the solution of multiaquifer flow problems on an SP4 machine and a Linux Cluster. We analyse the sequential and parallel efficiency of our algorithm, also compared with standard packages. Questions regarding the parallel solution of finite element eigenproblems are addressed and discussed. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: multi-aquifer systems; finite elements; eigenpairs; parallel computations

## 1. INTRODUCTION

The equations governing transient 3D porous media flow are

$$\nabla \cdot (K \nabla \psi) = S \frac{\partial \psi}{\partial t} + \mathbf{f} \tag{1}$$

where $\psi$ is the pressure head, $K$ is the permeability tensor, $S$ is the elastic storage of the aquifer and $\mathbf{f}$ is a source or sink term. Dirichlet and Neumann boundary conditions must be given to identify a well-posed mathematical formulation of the flow problem. Integration of Equation (1) over a tetrahedral finite element (FE) grid [1, 2] yields the linear system of

*Correspondence to: F. Sartoretto, Dipartimento di Informatica, Università di Venezia, Italy.
†E-mail: sartoret@dmsa.unipd.it

Contract/grant sponsor: INDAM-GNCS

*Received 30 July 2004*
*Revised 24 January 2005*
*Accepted 26 January 2005*

differential equations

$$C\frac{\partial \boldsymbol{\psi}}{\partial t} + H\boldsymbol{\psi} + \mathbf{q} = 0 \tag{2}$$

where $H$ and $C$ are the symmetric, positive definite (SPD), stiffness and capacity matrices, respectively; $\mathbf{q}$ is a known vector. Integrating in time by finite differences, for instance using the Crank–Nicolson scheme, leads to a sequence of linear algebraic systems.

The solution of Equation (2) can also be computed by evaluating a number of the generalized eigenpairs

$$H\mathbf{u}_i = \lambda_i C\mathbf{u}_i, \quad i = 1, 2, \ldots, N \tag{3}$$

$\lambda_1 \leqslant \lambda_2 \leqslant \cdots \leqslant \lambda_N$ being the real positive eigenvalues and $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_N$ the corresponding eigenvectors. Assume that $\boldsymbol{\psi}(0) = \mathbf{0}$, $\mathbf{q}$ is time independent, and let

$$w_i = -\frac{\mathbf{u}_i^{\mathrm{T}}\mathbf{q}}{\lambda_i}(1 - \exp(-\lambda_i t)), \quad i = 1, 2, \ldots, N$$

The solution $\boldsymbol{\psi}(t)$ of (2) can be written as $\boldsymbol{\psi}(t) = U\mathbf{w}(t)$, where $U = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_N]$ is the modal matrix [3]. Only a number $q$ of the leftmost eigenpairs need to be evaluated, since the importance of the *flow modes*, in capturing the actual response, decreases from smaller (leftmost) eigenvalues to larger (rightmost) ones.

Using mass-lumping technique, the matrix $C$ is reduced to diagonal form, $\hat{C}$, hence $\hat{C}^{-1}$ is straightforwardly evaluated and the generalized problem (3) becomes computationally equivalent to the SPD classical eigenvalue problem

$$A\mathbf{u}_i = \lambda_i \mathbf{u}_i, \quad i = 1, 2, \ldots, N$$

In the sequel we refer to this reduced problem and the matrix $A = \hat{C}^{-1}H$ involved in it.

Our previously published results show that the Jacobi–Davidson (JD) method [4], preconditioned by AINV [5, 6] or FSAI-type [7] approximate inverses is an effective tool for the partial eigenanalysis of flow problems [8, 9]. For parallel preconditioner evaluations, we focused on FSAI, since the parallelization of AINV preconditioners proved more cumbersome (see e.g. Reference [10]).

This paper is organized as follows: Section 2 recalls some features of JD algorithm, and FSAI-class preconditioners. Section 3 sketches our parallelization strategy. Section 4 reports some considerations on the discretization of our problems. Section 5 displays and analyses distinguished numerical results. Section 6 summarizes our conclusions.

## 2. SEQUENTIAL PRECONDITIONED JD

We thoroughly experimented with JD algorithm in the partial eigensolution of our flow and transport problems [11, 12].

We found that unpreconditioned JD is unable to solve our large problems, while effective preconditioning by approximate inverse preconditioners, like those of FSAI type [7], allows for efficiently solving all our problems.

Tables I–III sketch the preconditioned JD algorithm after Reference [13].

Table I. JD algorithm (preconditioned Jacobi–Davidson) for computing $k_{\max}$ exterior eigenpairs.

```
begin
  Set $\mathbf{V}_{:,0}$ starting vector and $\theta$ reference value;
  Set $\tau$, $n_{it,\max}, m_{\min}, m_{\max}$;
  $\mathbf{t} := \mathbf{V}_{:,0}$,   $k := 0$,   $m := 0$,   $\mathbf{U} := [\,]$,   $n_{it} := 0$;
  while $(k < k_{\max})$ and $(n_{it} < n_{it,\max})$ do $n_{it} := n_{it} + 1$;
    for $i := 1, m$ do $\mathbf{t} := \mathbf{t} - (\mathbf{V}_{:,i}^{\mathrm{T}}\mathbf{t})\mathbf{V}_{:,i}$; endfor
    $m := m + 1$,   $\mathbf{V}_{:,m} = \mathbf{t}/\|\mathbf{t}\|_2$,   $\mathbf{V}_{:,m}^{A} = \mathbf{A}\mathbf{V}_{:,m}$;
    for $i := 1, m$ do $\mathbf{M}_{i,m} := \mathbf{V}_{:,i}^{\mathrm{T}}\mathbf{V}_{:,m}^{A}$; endfor
    Compute the eigenpairs $(\mu_i, \mathbf{s}_i)$ of the matrix $\mathbf{M}$, $\|\mathbf{s}_i\|_2 = 1$;
    Sort the pairs $(\mu_i, \mathbf{s}_i)$ such that $|\mu_i - \theta| \geqslant |\mu_{i-1} - \theta|$;
    $\mathbf{u} := \mathbf{V}\mathbf{s}_1$,   $\mathbf{u}^A := \mathbf{V}^A\mathbf{s}_1$,   $\mathbf{r} := \mathbf{u}^A - \mu_1\mathbf{u}$;
    {inner loop}
    {correction step}
  endwhile
  if $k < k_{\max}$
    then print 'less than $k_{\max}$ eigenpairs computed';
  endif
end
```

Table II. JD inner loop.

```
while $(\|\mathbf{r}\|_2 \leqslant \tau \cdot \mu_1)$ and $(k < k_{\max})$ do
  $k := k + 1$;   $\lambda_k := \mu_1$,   $\mathbf{U} := [\mathbf{U}|\mathbf{u}]$;
  print the new eigenpair $(\lambda_k, \mathbf{u})$;
  $n_{it} := 0$;
  if $k < k_{\max}$
    then $m := m - 1$,   $\mathbf{M} := 0$;
      for $i := 1, m$ do
        $\mathbf{V}_{:,i} := \mathbf{V}\mathbf{s}_{i+1}$,   $\mathbf{V}_{:,i}^{A} := \mathbf{V}^A\mathbf{s}_{i+1}$;
        $\mathbf{M}_{i,i} := \mu_{i+1}$,   $\mathbf{s}_i := \mathbf{e}_i$,   $\mu_i := \mu_{i+1}$;
      endfor
      $\mathbf{u} := \mathbf{V}_{:,1}$,   $\mathbf{r} := \mathbf{V}_{:,1}^{A} - \mu_1\mathbf{u}$;
  endif
endwhile
```

Table III. JD correction step. The matrix $\mathbf{K}$ is an approximate inverse preconditioning matrix.

```
if $k < k_{\max}$
  then
    if $m \geqslant m_{\max}$
      then $\mathbf{M} := 0$;
        for $i := 2, m_{\min}$ do
          $\mathbf{V}_{:,i} := \mathbf{V}\mathbf{s}_i$,   $\mathbf{V}_{:,i}^{A} := \mathbf{V}^A\mathbf{s}_i$,   $\mathbf{M}_{i,i} := \mu_i$;
        endfor
        $\mathbf{V}_{:,1} := \mathbf{u}$,   $\mathbf{V}_{:,1}^{A} := \mathbf{u}^A$,   $\mathbf{M}_{1,1} := \mu_1$,   $m := m_{\min}$;
    endif
    $\mu := \mu_1$,   $\mathbf{Q} := [\mathbf{U}|\mathbf{u}], \mathbf{P} = (\mathbf{I} - \mathbf{Q}\mathbf{Q}^{\mathrm{T}})$;
    Solve the problem $\mathbf{P}(\mathbf{A} - \mu\mathbf{I})\mathbf{P}\,\mathbf{t} = -\mathbf{r}$, for $\mathbf{t} \perp \mathrm{span}\{\mathbf{Q}\}$,
    using BiCGSTAB, preconditioned with the projected matrix $\mathbf{PKP}$;
endif
```

Note that in order to solve the symmetric correction equations we used BiCGSTAB [14], while other authors exploit conjugate gradients (CG). Indeed, preconditioned CG converges when solving JD correction equations, though they are not positive definite. However, we showed in Reference [15] that CG performance in our problems is not any time superior to BiCGSTAB.

## 2.1. The FSAI preconditioner

Let $\mathscr{P}(M)=\{(i,j) \text{ s.t. } M_{ij}\neq 0\}$ be the pattern of a matrix $M$, and let $M=M_{\mathrm{L}}+M_{\mathrm{D}}+M_{\mathrm{L}}^{\mathrm{T}}$ the classical splitting of $M$ into lower triangular ($M_{\mathrm{L}}$), diagonal ($M_{\mathrm{D}}$), and upper triangular parts. Let $A$ be a SPD matrix and let $A=LL^{\mathrm{T}}$ be its Cholesky factorization. The FSAI preconditioning method gives an approximate inverse of $A$ in the factorized form $K=G^{\mathrm{T}}G$, where $G$ is a sparse non-singular lower triangular matrix approximating $L^{-1}$. In order to compute $G$, one must first prescribe a sparsity pattern $W=\{(i,j) : 1\leqslant i<j\leqslant n\}$; a lower triangular matrix $\hat{G}$ is computed, by solving the equations

$$(\hat{G}A)_{ij}=\delta_{ij}, \quad (i,j)\in W$$

A common choice for the sparsity pattern is $W=\mathscr{P}(A_{\mathrm{L}})$. A slightly more sophisticated and more expensive choice relies upon setting $W=\mathscr{P}((A^k)_{\mathrm{L}})$, where $k$ is a small positive integer, say $k=2$ or $k=3$ (see Reference [16]). This variant is called *enlarged* FSAI. We found that $k=2$ is an effective choice for our eigenproblems. Let us label the ensuing preconditioner FSAI(A$^2$). In Reference [17] a simple approach, called *post filtration*, was proposed to improve the quality of *enlarged* FSAI preconditioners. This method is based on *a posteriori* sparsification of a given preconditioner, by dropping all the elements whose absolute value is below a given threshold, $\varepsilon$. The aim is to reduce the number of non-zero elements of the preconditioner factors, thus decreasing the arithmetic complexity of the iterative phase. Also, in a parallel environment, when performing products of the preconditioning matrix times a vector, one can achieve a substantial reduction in data exchange. To compute an efficient preconditioner, the drop tolerance parameter, $\varepsilon$, must be identified. We tested values ranging from 0 to 1. We found that, as a preconditioner in the solution of linear systems, the efficiency of FSAI(A$^2$) with post filtration does not heavily depend upon $\varepsilon$, as reported in Reference [18]. We found that $\varepsilon=0.1$ works well in our problems. In the sequel, the FSAI preconditioner with $W=\mathscr{P}(A_{\mathrm{L}})$ will be called simply 'FSAI1'. The *enlarged* FSAI with pattern $W=\mathscr{P}((A^2)_{\mathrm{L}})$ and post filtration with $\varepsilon=0.1$, will be called 'FSAI2'. The JD algorithm preconditioned with FSAI1 will be labelled 'JDF1', while 'JDF2' will denote JD preconditioned with FSAI2.

We detected a large number of cache misses, when performing sparse matrix–vector multiplications. To enhance the sequential performance, we implemented the pre-fetching strategy sketched in *Algorithm 2* after Reference [19]. Our implementation for the *local* matrix–vector product, supplemented with data pre-fetching instructions, is reported in Table IV. Data pre-fetching reduce data dependencies in the innermost loop iteration. All data are loaded from memory one loop iteration before it is actually needed, so that the processor can better overlap computation and memory transfer. While the algorithms in Reference [19] are written for matrices stored in symmetric sparse skyline (SSS) format, we exploited the compressed sparse row (CSR) format. Note that, albeit our matrices are symmetric ones, in order to enhance parallel efficiency, we stored them in full form, not only the upper (or lower) part.

Table IV. Our implementation of Algorithm 2 after Geus and Röllin.

```
      subroutine matvec(n, ia, ja, a, x, y)
c Matrix-vector product y = A x, with data pre-fetching.
c The matrix A is stored in CSR format.
c
      implicit none
      integer n, i, j, j1, k, k1, l
      integer ia(*), ja(*)
      real*8 a(*), x(*), y(*), xi, s, v, v1
c
      k = 1
      do i = 1, n
        xi = x(i) ! pre-fetch
        s = 0.
        k1 = ia(i+1)
        if (k .lt. k1) then
          j = ja(k) ! pre-fetch
          v = a(k) ! pre-fetch
          k = k+1
          do  while (k .lt. k1)
            j1 = ja(k) ! pre-fetch
            v1 = a(k) ! pre-fetch
             s = s + v * x(j)
             j = j1 ! pre-fetch
             v = v1 ! pre-fetch
             k = k + 1
          end do
          s = s + v * x(j)
        endif
        y(i) = s
      end do
      return
      end
```

## 3. ALGORITHM PARALLELIZATION

A data-splitting approach was exploited to perform in parallel the following tasks: The computation of the stiffness, $H$, and capacity, $C$, matrices; the evaluation of the FSAI preconditioners; the execution of the preconditioned JD algorithm.

### *3.1. Parallel evaluation of $H$ and $C$*

Our code assumes that the 3D domain is discretized into a mesh counting $T$ tetrahedral elements and $N$ nodes [2]. Data splitting of the tetrahedral mesh is performed as follows. Assume that $P$ processors are engaged. Each processor, $p$, manages only those tetrahedrons whose node numbers, $i$, fall into the range $[N/P]\cdot(p-1)+1\leqslant i\leqslant[N/P]\cdot p$ (the operator $[\cdot]$ denotes the integer part of a real number). Since we numbered the nodes in a stratum-wise manner, each processor owns a subset of adjacent strata nodes.

The assembling procedure is performed independently on each processor, which owns merely $[N/P]$ rows of the global matrices.

### 3.2. FSAI parallelization

We implemented the parallel computation of FSAI1 and FSAI2 preconditioning factors. Though for a given matrix $A$ the computation of a generic FSAI preconditioning matrix, $G$, is an inherently parallel computation, there are some issues to be addressed, in order to attain an efficient implementation on a distributed memory computer. We used a *block row distribution* of matrices $A$ and $G$, i.e. complete rows were assigned to different processors. All our matrices are stored in static CSR format. In the SPD case any row, $i$, of matrix $G$ can be computed independently of each other, by solving a small SPD dense linear system, whose size, $n_i$, is equal to the number of non-zeros allowed in that row of $G$. Hence, the processor that computes row $i$ must access $n_i$ rows of $A$. A small subset of these rows must be received from the other processors. All the data exchanges among the processors are carried out before starting the computation of $G$, which proceeds afterward entirely in parallel.

Each processor computes its local rows of $G$ by solving, for any row, a (small) dense linear system. To form the dense sub-matrix the local part of $A$ and the previously received auxiliary non-local rows are needed.

The dense linear systems are solved using BLAS3 routines from LAPACK. Once $G$ is obtained, a parallel transposing routine provides every processor with its local part of $G^{\mathrm{T}}$.

### 3.3. Parallelization of JD

JD algorithm can be decomposed into a number of scalar products, `daxpy`-like linear combinations of vectors, $\alpha\mathbf{v} + \beta\mathbf{w}$, and matrix–vector (MV) products.

We focused on parallelizing these tasks, assuming that the code is to be run on a fully distributed *uniform memory access machine with identical, powerful processors*. This assumption is nowadays not usually true for parallel systems, but exploring and identifying architectural parameters in order to exploit the best efficiency on those different machine architectures that are available, is beyond our scope.

Scalar products, $\mathbf{v} \cdot \mathbf{w}$, were distributed among the $P$ processors by uniform block mapping.

We tailored the implementation of parallel matrix–vector products for application to sparse matrices, using a technique for minimizing data communication between processors [20]. In the greedy matrix–vector algorithm, each processor communicates with each other. Using our approach with our sparse FE matrices, usually each processor sends/receives data to/from at most 2 other processors, and when running on $P$ processors, the amount of data exchanged is far smaller than $[N/P]$.

## 4. DISCRETIZATION

Our test domain, which is the unit cube $[0, 1]^3$, is divided into $N_S$ aquifers (strata), whose top and bottom surfaces are discretized by a Friedrichs–Keller-type mesh [21] (squares divided by their left-to-right diagonals) counting $N_T$ triangles. By joining each surface node of each aquifer with the corresponding node on the bottom, the domain is divided into $N_T \times N_S$ triangular prisms. By dividing each prism into three tetrahedrons, we obtain the final grid, with $N$ nodes and $T = 3 \times N_T \times N_S$ tetrahedrons [2]. We used three meshes which exploit different numbers of strata and triangles. Table V shows the values of the discretization parameters.

Table V. Main characteristics of our test meshes and matrices.

| Problem | $N_T$ | $N_S$ | $N$ | $N_a$ | dd | HB |
|---|---|---|---|---|---|---|
| 1 | 10240 | 50 | 268515 | 3926823 | −2.67e-02 | 5265 |
| 2 | 10240 | 100 | 531765 | 7817373 | −1.33e-02 | 5265 |
| 3 | 40960 | 50 | 1059219 | 15605175 | −2.67e-02 | 20769 |
| 4 | 40960 | 100 | 2097669 | 31066125 | −1.33e-02 | 20769 |
| 5 | 163840 | 50 | 4027347 | 62216919 | −2.67e-02 | 82497 |
| 6 | 10240 | 50 | 268515 | 3926823 | 3.13e-09 | 5265 |
| 7 | 10240 | 100 | 531765 | 7817373 | 1.56e-09 | 5265 |

In Problems 1–5 the permeability, $K$, and the thickness, $l = 1/N_S$, are constant on each stratum. Two more problems were considered, i.e. Problems 6 and 7, sharing the same mesh structure as Problems 1 and 2, respectively, but featuring strata having unequal permeability and thickness, whose values were chosen in order to attain real-life-like problems. The minimum thickness was 5/10 000 of the domain height, the maximum one 1/20. Two permeability values, $K'$ and $K''$ were chosen, whose ratio is $K''/K' = 5 \times 10^{-5}$; the value $K'$ was set into the first couple of adjacent strata, then $K''$ in the following couple, and so on.

Dirichlet boundary conditions, $\psi = 0$, were set on the $x = 0$ face of the domain; Neumann flux $\partial\psi/\partial n = -1$ was set on the top face node with co-ordinates $x = 1$, $y = 0$. The remaining part of the boundary is impervious.

Each FE sparse matrix, $A$, arising in our eigenproblems, has $N_a$ active entries, shown in Table V. The table also reports the *diagonal dominance* parameter $dd = \max_i\{|A_{ii}| - \sum_{i \neq j} |A_{ij}|\}$. One can see that the matrices arising in Problems 1–5 are *not* diagonally dominant ones, while Problems 6 and 7 involve strictly diagonally dominant matrices. Table V shows the half-bandwidth $\text{HB} = \max\{|i - j| \text{ s.t. } A_{ij} \neq 0\}$ of our matrices. Compared to matrix size, small HB values are shown: Bandwidth reduction techniques should not reduce appreciably the HB value. The performance of matrix–vector products is not likely to be increased by renumbering, as shown in Reference [19]. Moreover, node rearranging produces a non-stratum-wise node numbering, which would reduce the efficiency of our data-splitting approach among the processors.

The patterns of our test matrices, due to our stratum-wise node numbering, are similar. Figure 1 sketches the pattern of the first 67 354 rows and columns of some matrices with size 268 515. The remaining part of the matrices has the same pattern. The matrices, $A$, arising in Problems 1 and 6, are considered, together with the pattern of the lower triangular part of $A^2$, which undergoes an appreciable fill-in, and has $\text{HB} = 10\,530$, twice the bandwidth of $A$. The FSAI2 lower triangular factor for Problem 6 is also shown. The FSAI2 factor has the same bandwidth as $A^2$, but a smaller number of active entries, due to post filtration.

We recall that our code treats *unstructured* matrices. It does not take advantage of any pattern regularity. Note that our sample FE matrices display band, unstructured patterns. The preconditioning factors display a larger bandwidth and a more irregular structure than the FE matrices.

Figure 2 shows the first 9 *relative eigenvalue gaps*, $g_i = 1 - \lambda_{i-1}/\lambda_i$, $i = 2, \ldots, 10$, which drive the convergence of eigenvalue algorithms [22] for the leftmost eigensolution. Note that all problems with constant permeability and layer thickness (Problems 1–5) display the same gaps, suggesting that *the coarseness of our uniform mesh discretizations does not affect the*

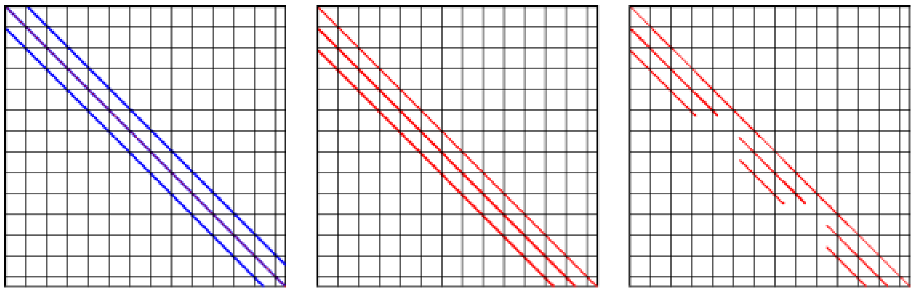

Figure 1. Patterns of the 67 354 size principal sub-matrices of 268 515 size matrices. A 5000 element wide square grid was superimposed. Left frame: pattern of the matrices, $A$, arising in Problems 1 and 6. Centre: the pattern of the lower triangular part of $A^2$. Right: the pattern of the FSAI2 lower triangular preconditioning factor for Problem 6.

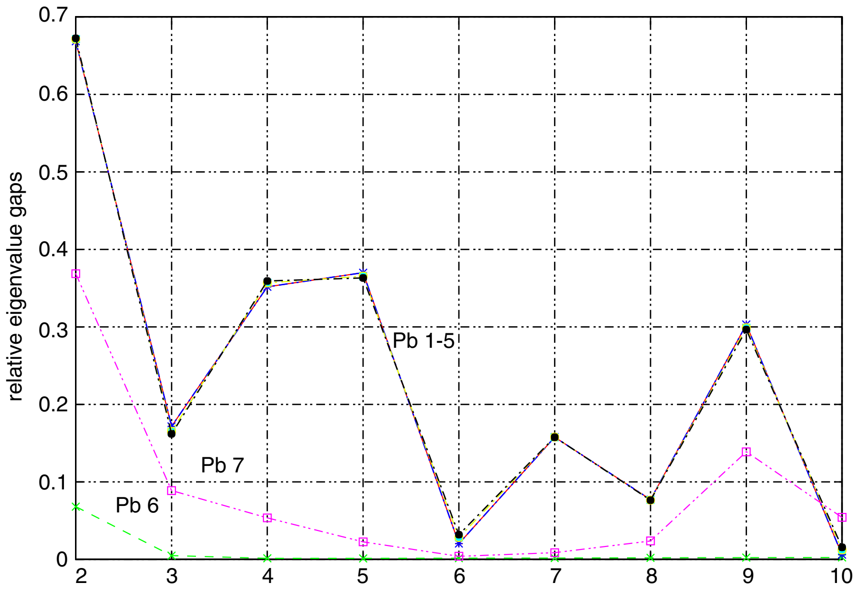

Figure 2. Relative eigenvalue gaps.

*numerical properties of the ensuing leftmost eigenvalue problem*. Heterogeneous problems 6 and 7 display worse (i.e. smaller) separation between eigenvalues, hence they are expected to require a higher computational cost to perform an accurate eigenanalysis.

## 5. NUMERICAL RESULTS

To achieve the highest portability and efficiency, we implemented our code using Fortran 77 and accomplishing parallel tasks by calls to standard MPI 1.0 library.

Our numerical tests were performed on both the IBM SP RS/6000 Power 4 Supercomputer (called SP4 in the sequel) and the IBM Linux Cluster 1350 (called CLX in the sequel), both located at the Supercomputing Centre CINECA, in Italy (`http://www.cineca.it`).

The SP4 system includes 512 POWER 4, 1.3 GHz, processors. The current configuration features 48 *virtual* nodes, 32 nodes with 8 processors and 16 GB RAM each, 14 nodes with 16 processors and 32 GB RAM and 2 nodes with 16 processors and 64 GB RAM each. The nodes are connected with 2 interfaces to a dual plane set of Colony switches.

The CLX cluster features 256 nodes, each one encompassing two Intel Xeon Pentium IV 3.055 GHz CPUs. Each processor has a 512 KB L2 cache, each node has a 2 GB DRAM. The processing elements are connected by a Myrinet-2000 network, consisting of 256 M3F-PCI64C adapters and 6 128-port switches, which guarantee *full bisectional bandwidth* functionality. The theoretical maximum bandwidth allowed is 200 MB/s.

## *5.1. Performance analysis*

We recorded the CPU times spent for evaluating the numerical part of our code, i.e. (a) the evaluation of the stiffness and capacity matrices (task Matr in Table VI), (b) the computation of the preconditioning matrices (tasks FSAI1, FSAI2), and (c) computation of $q=10$ leftmost eigenpairs via preconditioned JD (tasks JDF1, JDF2). From Table VI one can see that most of the computational effort is spent for performing the preconditioned JD algorithm.

The CPU time spent for each task changes appreciably from run to run, on both the SP4 and the CLX. We run *10 times* each job, activating in turn $P=1,2,4,8,16$ processors. Table VII summarizes the CPU seconds recorded through 10 runs, spent by JDF1 and JDF2, when solving Problem 2 on the SP4. Table VII reports the minimum time, $T_P^{(\min)}$, the maximum

Table VI. CPU seconds spent on the SP4 to perform each main task, and corresponding percentages, when $P=16$ processors were exploited.

| | Times | | | | Percentages | | |
|---|---|---|---|---|---|---|---|
| Problem | Matr | FSAI1 | JDF1 | Tot | Matr % | FSAI1 % | JDF1 % |
| 1 | 0.1 | 0.3 | 35.3 | 35.7 | 0.28 | 0.84 | 98.88 |
| 2 | 0.2 | 0.6 | 135.5 | 136.3 | 0.15 | 0.44 | 99.41 |
| 3 | 0.6 | 1.3 | 213.7 | 215.6 | 0.28 | 0.60 | 99.12 |
| 4 | 1.1 | 2.4 | 680.5 | 684.0 | 0.16 | 0.35 | 99.49 |
| 5 | 3.3 | 5.4 | 1593.0 | 1601.7 | 0.21 | 0.34 | 99.46 |
| 6 | 0.1 | 0.3 | 862.3 | 862.7 | 0.01 | 0.03 | 99.95 |
| 7 | 0.2 | 0.6 | 4812.0 | 4812.8 | 0.00 | 0.01 | 99.98 |
| Problem | Matr | FSAI2 | JDF2 | Tot | Matr % | FSAI2 % | JDF2 % |
| 1 | 0.1 | 1.4 | 26.2 | 27.7 | 0.36 | 5.05 | 94.58 |
| 2 | 0.2 | 2.7 | 89.4 | 92.3 | 0.22 | 2.93 | 96.86 |
| 3 | 0.6 | 5.6 | 191.4 | 197.6 | 0.30 | 2.83 | 96.86 |
| 4 | 1.1 | 11.1 | 504.0 | 516.2 | 0.21 | 2.15 | 97.64 |
| 5 | 3.3 | 23.0 | 1347.8 | 1374.1 | 0.24 | 1.67 | 98.09 |
| 6 | 0.1 | 1.4 | 174.1 | 175.6 | 0.06 | 0.80 | 99.15 |
| 7 | 0.2 | 2.8 | 412.2 | 415.2 | 0.05 | 0.67 | 99.28 |

Table VII. Problem 2. Minimum, maximum, and average CPU seconds spent on the SP4, counting 10 runs for each value of $P$. The ensuing speedup values and their maximum variations are also shown.

| $P$ | $T_P^{(\min)}$ | $T_P^{(\max)}$ | $T_P^{(\mu)}$ | $V_P^{(T)}(\%)$ | $S_P^{(\min)}$ | $S_P^{(\max)}$ | $V_P^{(S)}(\%)$ |
|---|---|---|---|---|---|---|---|
| *FSAI1* | | | | | | | |
| 1 | 4.2 | 4.4 | 4.3 | 4.76 | — | — | — |
| 2 | 2.3 | 2.4 | 2.4 | 4.35 | 1.75 | 1.91 | 9.32 |
| 4 | 1.3 | 1.4 | 1.3 | 7.69 | 3.00 | 3.38 | 12.82 |
| 8 | 0.8 | 0.8 | 0.8 | 0 | 5.25 | 5.50 | 4.76 |
| 16 | 0.6 | 0.6 | 0.6 | 0 | 7.00 | 7.33 | 4.76 |
| *FSAI2* | | | | | | | |
| 1 | 34.1 | 35.5 | 34.8 | 4.11 | — | — | — |
| 2 | 17.3 | 17.4 | 17.4 | 0.58 | 1.96 | 2.05 | 4.71 |
| 4 | 8.9 | 9.2 | 9.1 | 3.37 | 3.71 | 3.99 | 7.61 |
| 8 | 4.7 | 4.8 | 4.8 | 2.13 | 7.10 | 7.55 | 6.32 |
| 16 | 2.7 | 2.8 | 2.7 | 3.70 | 12.18 | 13.15 | 7.96 |
| *JDF1* | | | | | | | |
| 1 | 959.6 | 1002.2 | 979.7 | 4.44 | — | — | — |
| 2 | 545.9 | 558.8 | 553.5 | 2.36 | 1.72 | 1.84 | 6.91 |
| 4 | 334.7 | 359.0 | 348.3 | 7.26 | 2.67 | 2.99 | 12.02 |
| 8 | 228.9 | 241.3 | 233.6 | 5.42 | 3.98 | 4.38 | 10.10 |
| 16 | 135.5 | 157.3 | 143.5 | 16.09 | 6.10 | 7.40 | 21.24 |
| *JDF2* | | | | | | | |
| 1 | 674.4 | 739.7 | 693.4 | 9.68 | — | — | — |
| 2 | 351.5 | 414.4 | 366.5 | 17.89 | 1.63 | 2.10 | 29.31 |
| 4 | 232.3 | 278.4 | 244.5 | 19.85 | 2.42 | 3.18 | 31.45 |
| 8 | 165.9 | 191.9 | 173.8 | 15.67 | 3.51 | 4.46 | 26.87 |
| 16 | 88.7 | 93.4 | 89.3 | 5.30 | 7.22 | 8.34 | 15.49 |

one, $T_P^{(\max)}$, the average one, $T_P^{(\mu)}$, and the maximum variation $V_P^{(T)} = T_P^{(\max)}/T_P^{(\min)} - 1$. Moreover, the minimum speedup is shown, together with the maximum one, and the *speedup variation*, defined as

$$S_P^{(\min)} = \frac{T_1^{(\min)}}{T_P^{(\max)}}, \quad S_P^{(\max)} = \frac{T_1^{(\max)}}{T_P^{(\min)}}, \quad V_P^{(S)} = \frac{S_P^{(\max)}}{S_P^{(\min)}} - 1$$

One can see from Table VII that the speedup of JDF2, as an example, can change more than 30%, which tells us that the speedup has a *qualitative* meaning, rather than a *quantitative* one.

In the sequel we report the *minimum* CPU time recorded, assuming it is the ideal time which would be reported if the machine loads merely our job (users are not allowed to run dedicated jobs).

*5.1.1. FE matrices evaluation.* One can see from Table VI that our code evaluates the stiffness and capacity matrices by a *negligible* fraction of the overall resolution time. A similar fraction

Table VIII. CPU seconds for building the preconditioning factors on the SP4.

| Problem | $T_1$ | $T_2$ | $T_4$ | $T_8$ | $T_{16}$ |
|---|---|---|---|---|---|
| *FSAI1* | | | | | |
| 1 | 2.2 | 1.2 | 0.7 | 0.4 | 0.3 |
| 2 | 4.2 | 2.3 | 1.3 | 0.8 | 0.6 |
| 3 | 8.9 | 4.9 | 2.7 | 1.6 | 1.3 |
| 4 | * | 9.5 | 5.3 | 3.2 | 2.4 |
| 5 | * | * | * | 6.9 | 5.4 |
| *FSAI2* | | | | | |
| 1 | 17.1 | 8.8 | 4.6 | 2.4 | 1.4 |
| 2 | 34.1 | 17.3 | 8.9 | 4.7 | 2.7 |
| 3 | 69.2 | 35.6 | 18.6 | 9.9 | 5.6 |
| 4 | * | 70.4 | 36.3 | 19.4 | 11.1 |
| 5 | * | * | * | 41.5 | 23.0 |

is spent when $P<16$ processors are used. These results testify the efficiency of the matrix–evaluation portion of our code.

*5.1.2. Preconditioner computations.* Table VIII shows the CPU seconds spent on the SP4 to compute FSAI preconditioning factors by parallel runs. Problems 6 and 7 are not shown, since they require practically the same time as 1 and 2, respectively. Note that we exploited 8-processor virtual nodes, where *each* processor shares (in principle) the whole 16 GB available storage. Such amount of memory is larger than that required by our largest problem. However, the SP4 scheduler practically prevents from running jobs requiring more than 3 GB/processor, in order to avoid too large memory conflict rates. This limitation prevented us from running our largest problems, unless we split the data between a suitable number of processors. In our tables, a result which could not be obtained due to memory limitations is replaced by an asterisk. The quantities which cannot be computed due to lacking results, are replaced by a slash. As an example, inspecting Table VIII one can see that Problems 4 and 5 could not be run on one processor; Problem 5 could not be run even on $P=1,2,4$ processors.

Concerning the evaluation of the preconditioning matrices, inspecting Table VIII, one can see that computing FSAI2 requires more than 4 times the CPU seconds required for FSAI1. In any case Table VI shows that evaluating both preconditioning factors require a very small fraction of the time spent by the JD algorithm.

Figure 3 shows the speedup ranges raised in Problem 2 when computing FSAI1 and FSAI2. The *minimum* FSAI1 speedup values over 10 runs were linearly interpolated; the same interpolation was performed for the *maximum* FSAI1 speedup values; the area encompassed by the two piece-wise linear curves was filled. The same operations were performed for FSAI2. Figure 3 shows that the parallel degree of the FSAI1 construction phase is smaller. Indeed, the amount of data to be exchanged among the processors remains roughly constant when the number of processors changes, due to the band-like structure of the matrices involved (cf. Figure 1). On the other hand, the complexity of the local computation decreases. This behaviour obviously worsens the parallel performance. The situation changes when the pattern of $A^2$ is used, since it displays an increase in the bandwidth w.r.t. $A$ (analogous to that one shown for Problem 6 in Figure 1), hence the speedup is larger when evaluating FSAI2.

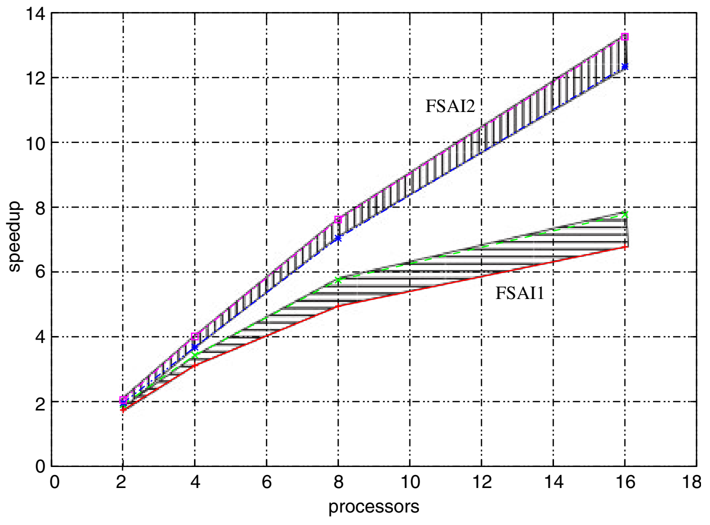

Figure 3. Problem 2. Speedup ranges on the SP4, recorded when evaluating FSAI1, FSAI2.

Table IX. SP4 CPU seconds spent by preconditioned JD for computing $q = 10$ eigenpairs.

| Problem | $T_1$ | $T_2$ | $T_4$ | $T_8$ | $T_{16}$ |
|---|---|---|---|---|---|
| *JDF1* | | | | | |
| 1 | 246.8 | 138.2 | 85.7 | 62.3 | 35.3 |
| 2 | 959.6 | 545.9 | 334.7 | 228.9 | 135.5 |
| 3 | 1363.3 | 765.9 | 433.5 | 347.8 | 213.7 |
| 4 | * | 3200.5 | 1765.6 | 1069.0 | 680.5 |
| 5 | * | * | * | 2011.8 | 1593.0 |
| 6 | 7042.2 | 2660.1 | 1801.4 | 1539.3 | 862.3 |
| 7 | 38027.1 | 22494.5 | 10858.9 | 8561.6 | 4812.0 |
| *JDF2* | | | | | |
| 1 | 205.0 | 118.6 | 65.3 | 51.2 | 26.2 |
| 2 | 674.4 | 351.5 | 232.3 | 165.9 | 88.7 |
| 3 | 1525.1 | 763.7 | 435.5 | 330.8 | 191.4 |
| 4 | * | 2470.3 | 1103.6 | 781.1 | 504.0 |
| 5 | * | * | * | 1957.5 | 1347.8 |
| 6 | 1238.4 | 679.4 | 421.3 | 299.8 | 174.1 |
| 7 | 3590.6 | 1863.4 | 1010.2 | 762.5 | 412.2 |

The overall speedup behaviour is typical of linear algebraic computations.

*5.1.3. Eigenpairs.* The leftmost $q = 10$ eigenpairs were computed, up to tolerance $\tau = 10^{-3}$ in the residual norm.

Table IX shows the CPU seconds spent to compute $q = 10$ leftmost eigenpairs, by either JDF1 or JDF2. None of our test problems can be solved by unpreconditioned JD, while JDF1

Table X. Ratios $\rho_P$ between CPU times for performing JDF1 without/with pre-fetching.

| Problem | $\rho_1$ | $\rho_2$ | $\rho_4$ | $\rho_8$ | $\rho_{16}$ |
|---|---|---|---|---|---|
| 1 | 1.37 | 1.35 | 1.39 | 1.36 | 1.24 |
| 2 | 1.37 | 1.35 | 1.42 | 1.39 | 1.26 |
| 3 | 1.49 | 1.41 | 1.44 | 1.45 | 1.27 |
| 4 | / | 1.23 | 1.38 | 1.43 | 1.24 |
| 5 | / | / | / | 1.31 | 1.11 |
| 6 | 1.28 | 1.50 | 1.36 | 1.30 | 1.18 |
| 7 | 1.37 | 1.24 | 1.41 | 1.35 | 1.23 |
| Average | 1.38 | 1.35 | 1.40 | 1.37 | 1.22 |

Table XI. Average number of iterations spent on the SP4 by JDF1 and JDF2. The maximum differences, $d_i$, are also shown.

| Problem | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| *JDF1* | | | | | | | |
| Average | 1123.2 | 1908.4 | 1289.6 | 2106.0 | 1820.0 | 26878.8 | 70486.0 |
| $d_i$ (%) | 2.8 | 4.9 | 3.2 | 12.3 | 0.0 | 19.0 | 9.7 |
| *JDF2* | | | | | | | |
| Average | 993.2 | 1534.0 | 1102.4 | 1608.0 | 1404.0 | 6624.8 | 6708.0 |
| $d_i$ (%) | 5.8 | 6.8 | 3.3 | 16.4 | 3.7 | 5.2 | 4.7 |

and JDF2 allow for efficiently computing the solutions. The CPU seconds spent by JDF2 are smaller than those for JDF1, except for Problem 3, $P=1,4$. Summarizing, JDF2 seems preferable over JDF1.

Let $T_P^{(n)}$ be the CPU times which was spent on the SP4 for performing JDF1 on $P$ processors, when *no* pre-fetching is exploited in matrix–vector products. Let $T_P^{(f)}$ be the time spent when pre-fetching *is* exploited (these latter CPU times are shown in Table IX). Table X shows the ratios $\rho_P = T_P^{(n)}/T_P^{(f)}$, raised when solving Problems 1–5. Note that everywhere $\rho_P > 1$, hence pre-fetching is indeed *effective* for lowering JDF1 CPU time. Analogous results hold for all our test problems, dealing with both JDF1 and JDF2 computations. On the other hand, we found that the speedup recorded when pre-fetching is *not* exploited is *in the same range* as the speedup recorded when pre-fetching *is* exploited. Hence pre-fetching is indeed useful for enhancing the sequential performance of our algorithm, without lowering the parallel performance. This is an interesting result, recalling also that we exploit CSR matrix store format, instead of SSS one, used in Reference [19]. In the sequel we report the CPU times spent when *using* the pre-fetching strategy in our matrix–vector code.

Note that using JD algorithm, we faced the well-known phenomenon that appreciable changes in the number of iterations can be observed when one changes the number of engaged processors, since the evaluation sequence of floating point operations changes. Table XI shows the

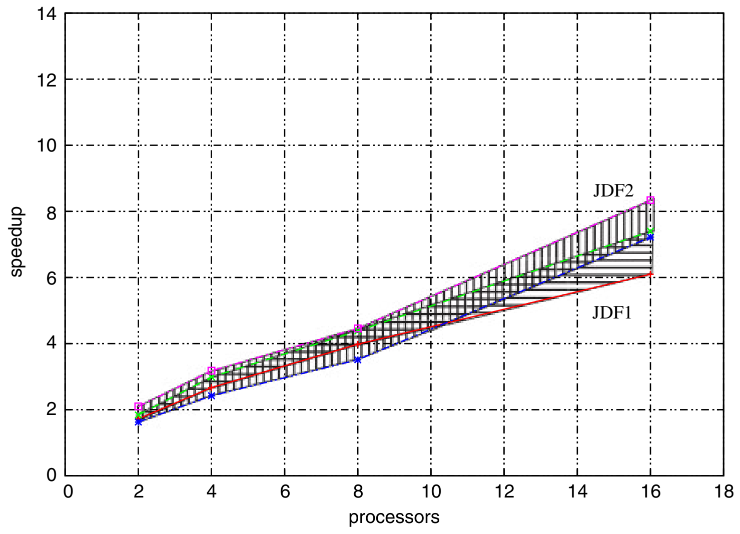

Figure 4. Problem 2. Speedup ranges on the SP4, recorded when running JDF1 or JDF2 (same scales as in Figure 3).

Table XII. Same as Table IX, for JDF2, running on the CLX system.

| Problem | $T_1$ | $T_2$ | $T_4$ | $T_8$ | $T_{16}$ |
|---|---|---|---|---|---|
| 1 | 212.1 | 113.2 | 66.3 | 37.4 | 25.1 |
| 2 | 537.0 | 293.9 | 147.0 | 88.0 | 56.0 |
| 3 | * | 594.6 | 322.8 | 183.4 | 109.4 |
| 6 | 1304.2 | 711.2 | 384.6 | 250.5 | 135.3 |
| 7 | 2528.5 | 1303.2 | 718.5 | 415.0 | 242.8 |

average JD iterations performed. The maximum differences

$$d_i = \max_{P=1,2,4,8,16} |I_{i,P} - \mu_i|/\mu_i, \quad i = 1, \ldots, 7$$

are also shown, being $I_{i,P}$ the number of preconditioned JD iterations performed to solve the $i$th problem by a $P$ processor run, while $\mu_i$, $i = 1, \ldots, 7$, is the average value over the set $\{I_{i,P},\ P = 1, 2, 4, 8, 16\}$. Such a behaviour affects the speedup values of preconditioned JD, but we found that its parallel performance is qualitatively unaffected.

Figure 4 is analogous to Figure 3, and it compares the speedup raised in Problem 2 when performing JDF1 and JDF2. One can see that a smaller speedup is raised w.r.t. FSAI computations, due to larger amounts of data exchanged. Nevertheless, a speedup 8 for $P = 16$ can be rated satisfactory in large flow problems.

To further analyse the efficiency of our code, we solved our problems on the CLX machine. Table XII shows the CPU seconds spent on the CLX system for performing JDF2 in Problems 1–3, 6 and 7. We were unable to run Problems 4 and 5, due to lack of core memory (2 GB/node is not enough). Note that when $P > 1$ *only one processor* per physical 2-processor node was exploited, to avoid memory conflicts which highly reduce the parallel performance.

Table XIII. Same as Table IX, for pARPACK on SP4. The number of performed iterations, $I$, is also shown, together with the ratio, $R_{16} = T_{16}^{(\mathrm{ARP})}/T_{16}^{(\mathrm{JDF})}$, between the CPU seconds spent, when $P = 16$, by pARPACK core, and the time spent by JDF1 or JDF2.

| Problem | I | $T_1$ | $T_2$ | $T_4$ | $T_8$ | $T_{16}$ | $R_{16}$ |
|---|---|---|---|---|---|---|---|
| *FSAI1* | | | | | | | |
| 1 | 35 | 1060.3 | 531.7 | 334.1 | 197.6 | 103.1 | 2.9 |
| 2 | 35 | 4543.6 | 2372.2 | 1341.4 | 874.0 | 408.9 | 3.0 |
| 3 | 30 | 5462.4 | 2955.3 | 1611.9 | 1160.9 | 633.3 | 3.0 |
| 4 | 35 | * | 10686.3 | 5935.2 | 4840.8 | 2712.1 | 4.0 |
| 5 | 35 | * | * | * | 8140.1 | 5127.8 | 3.2 |
| 6 | 192 | * | 23165.9 | 15693.5 | 8829.7 | 4458.7 | 5.2 |
| 7 | 63 | * | 36909.5 | 21785.6 | 13654.5 | 6045.4 | 1.3 |
| *FSAI2* | | | | | | | |
| 1 | 35 | 675.5 | 374.1 | 217.4 | 136.4 | 77.7 | 3.0 |
| 2 | 35 | 2624.8 | 1421.2 | 790.2 | 511.9 | 268.4 | 3.0 |
| 3 | 30 | 4643.9 | 2510.4 | 1438.6 | 960.7 | 548.4 | 2.9 |
| 4 | 35 | * | 7679.3 | 4380.1 | 3136.9 | 1936.7 | 3.8 |
| 5 | 35 | * | * | * | 7376.7 | 4518.8 | 3.4 |
| 6 | 212 | * | 6070.7 | 3323.3 | 2480.5 | 1286.0 | 7.4 |
| 7 | 75 | * | 7278.3 | 4330.6 | 2654.3 | 1234.3 | 3.0 |

It is interesting to note that the wall-clock time spent on the CLX is usually smaller than that on the SP4 machine, which is much more expensive than the CLX. Note, however, that much larger amounts of I/O time can be spent on the CLX, amounts which subsume too odd variations from run to run to be worth analysing.

From Table XII, one can argue that the speedup values on the CLX are larger than on the SP4, which is not surprising. One can compute that for each problem, $S_{16}$ is in the range 8–10. Indeed, the single processor peak performance on the CLX is worse than on the SP4, while the interconnection network performance seems comparably larger on the CLX.

*5.1.4. Comparison with pARPACK.* In order to confirm the efficiency of our preconditioned JD code, we also exploited the well-known pARPACK package [23, 24].

As suggested e.g. in Reference [13], we assumed that the dimension of the Ritz space is $s = 20$, i.e. twice the number, $q = 10$, of required eigenvalues. When solving Problems 1 to 5, we set the tolerance in the solution of linear systems to $\tau_g = 10^{-5}$, while we set $\tau_e = 10^{-4}$ the tolerance on the eigenvalue norms. By these settings, the final, average relative residual norm is $r \simeq 10^{-3}$, which well compares with that one obtained by preconditioned JD algorithm. Note, however, that in order to achieve a residual norm $r \simeq 10^{-3}$ when solving Problems 6 and 7 by JDF2, we had to set $\tau_g = 10^{-7}$, $\tau_e = 10^{-6}$. Achieving a comparable accuracy using JDF1, would require much smaller tolerances and unacceptably larger CPU times. Hence, also when solving Problems 6 and 7 by JDF1, we set $\tau_g = 10^{-7}$, $\tau_e = 10^{-6}$, thus obtaining a larger final residual $r \simeq 10^{-1}$.

Table XIII shows the CPU seconds spent for running the *numerical core* of pARPACK code (the computation of the preconditioning matrix is not counted) on the SP4. Comparing Tables IX and XIII one can see that the CPU time spent on the SP4 by pARPACK core is larger than that spent by JDF1 and JDF2. Moreover, recall that when solving Problems 6 and 7 pARPACK is *less accurate than JD*, as remarked previously. Table XIII reports also the ratios $R_{16} = T_{16}^{(\mathrm{ARP})}/T_{16}^{(\mathrm{JDF})}$, concerning $P = 16$ parallel runs, where $T_{16}^{(\mathrm{ARP})}$ is the time spent by pARPACK core, while $T_{16}^{(\mathrm{JDF})}$ is the time spent by either JDF1, when pARPACK is preconditioned by FSAI1, or JDF2, when FSAI2 is the preconditioner. For $P = 16$ these ratios show e.g. that the CPU time spent for performing pARPACK core, preconditioned by FSAI2, is at least 2.9 times that consumed for running JDF2. We usually found that $R_P > 1$.

We equipped the pARPACK package with the same matrix–vector multiplication code which we used for our JD solver, supplemented with the pre-fetching strategy by Geus and Röllin. Since matrix–vector multiplication is one of the most time consuming blocks inside pARPACK, it is not surprising that its parallel efficiency proved similar to that one of our JD code, as one can argue by comparing Tables IX and XIII.

Incidentally, note that, unlike JD, the number of pARPACK iterations, reported in Table XIII, *does not change* with the number of running processors, $P$.

## 6. CONCLUSIONS

The following points deserve mention:

- Performing parallel eigenanalysis of large, sparse, unstructured, symmetric matrices, arising from FE integration of large multiaquifer flow problems can be effectively done by JD algorithm, using FSAI-class preconditioners.
- We recommend using pre-fetching strategies when performing time consuming sparse matrix–vector products. Pre-fetching allows for a valuable reduction in the CPU wall-clock time.
- The fluctuations in the CPU time occurring from run to run, make it unpractical to accurately measure the parallel efficiency of numerical algorithms on CINECA's SP4 and CLX machines. Qualitative measures must be devised.
- The efficiency of JDF1 and JDF2, together with the quite inexpensive cost of evaluating the preconditioning factors, show that JDF1 and JDF2 are effective techniques for solving our very large, sparse problems. JDF2 proved generally more efficient than JDF1.
- The pARPACK package requires larger CPU times than JDF2, when attacking our symmetric problems. Moreover, when solving Problems 6 and 7, pARPACK provides a less accurate solution. These results confirm that exploiting our accurately tuned numerical code, which takes advantage of the characteristics of our problems, delivers a more efficient solver than using a standard, multi–purpose package.
- Compared to the SP4, the wall-clock time spent for performing JDF2 on the CLX machine proved usually smaller, and the speedup was larger (note that the I/O time can be much larger on the CLX).

ACKNOWLEDGEMENTS

This work was funded by INDAM-GNCS 2004 Project *Metodi e Algoritmi Numerici per Problemi Differenziali con Applicazioni alla Modellistica Ambientale*. Accounting units were granted by Padua and Venice Universities, as members of CINECA Consortium.

## REFERENCES

1. Zienkiewicz OC. *The Finite Element Method*. McGraw Hill: New York, 1986.
2. Gambolati G, Pini G, Tucciarelli T. A 3-D finite element conjugate gradient model of subsurface flow with automatic mesh generation. *Advances in Water Resources* 1986; **3**:34–41.
3. Gambolati G. On time integration of groundwater flow equations by spectral methods. *Water Resources Research* 1993; **29**(4):1257–1267.
4. Sleijpen GLG, van der Vorst HA. A Jacobi–Davidson method for linear eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications* 1996; **17**(2):401–425.
5. Benzi M, Meyer CD, Tůma M. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing* 1996; **17**(5):1135–1149.
6. Benzi M, Cullum JK, Tůma M. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing* 2000; **22**(4):1318–1332.
7. Kolotilina LY, Yeremin AY. Factorized sparse approximate inverse preconditionings I. Theory. *SIAM Journal on Matrix Analysis and Applications* 1993; **14**:45–58.
8. Bergamaschi L, Putti M. Numerical comparison of iterative eigensolvers for large sparse symmetric matrices. *Computer Methods in Applied Mechanics and Engineering* 2002; **191**(45):5233–5247.
9. Bergamaschi L, Martìnez A, Pini G, Sartoretto F. Eigenanalysis of finite element 3d flow models by parallel Jacobi Davidson. In *Euro PVM/MPI 2003 LNCS 2840*, Heidelberg, Dongarra J, Laforenza F, Orlando S (eds). Springer: Berlin, 565–569.
10. Benzi M, Marin J, Tůma M. A two-level parallel preconditioner based on sparse approximate inverses. In *Iterative Methods in Scientific Computation IV*, Kincaid DR, Elster AC (eds), IMACS Series in Computational and Applied Mathematics, vol. 5, 167–178, New Brunswick, New Jersey, U.S.A., 1999. International Association for Mathematics and Computers in Simulation.
11. Bergamaschi L, Pini G, Sartoretto F. Parallel preconditioning of a sparse eigensolver. *Parallel Computing* 2001; **27**:963–976.
12. Bergamaschi L, Pini G, Sartoretto F. Approximate inverse preconditioning in the parallel solution of sparse eigenproblems. *Numerical Linear Algebra with Applications* 2000; **7**(3):99–116.
13. Bai Z, Demmel J, Dongarra J, Ruhe A, van der Vorst H. *Templates for the Solution of Algebraic Eigenvalue Problems*: *A Practical Guide*. SIAM: Philadelphia, PA, 2000.
14. van der Vorst HA. Bi-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing* 1992; **13**:631–644.
15. Bergamaschi L, Pini G, Sartoretto F. Computational experience with sequential and parallel preconditioned Jacobi Davidson for large sparse symmetric matrices. *Journal of Computational Physics* 2003; **188**(1): 318–331.
16. Kaporin IE. New convergence results and preconditioning strategies for the conjugate gradient method. *Numerical Linear Algebra with Applications* 1994; **1**:179–210.
17. Kolotilina LY, Nikishin AA, Yeremin AY. Factorized sparse approximate inverse preconditioning IV. Simple approaches to rising efficiency. *Numerical Linear Algebra with Applications* 1999; **6**:515–531.
18. Bergamaschi L, Martínez A. Parallel acceleration of Krylov solvers by factorized approximate inverse preconditioners. In *LNCS 3402*, Daydè M *et al*. (eds). Springer-Verlag: Heidelberg, 2005; 621–634.
19. Geus R, Röllin S. Towards a fast parallel sparse symmetric matrix-vector multiplication. *Parallel Computing* 2001; **27**:883–896.
20. Bergamaschi L, Putti M. Efficient parallelization of preconditioned conjugate gradient schemes for matrices arising from discretizations of diffusion equations. *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999 (CD–ROM).
21. Knaber P, Angermann L. *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*. Springer: New York, 2003.
22. Parlett BN. *The Symmetric Eigenvalue Problem*. Prentice-Hall: Englewood Cliffs, NJ, U.S.A., 1980.
23. Lehoucq RB, Sorensen DC, Yang C. *ARPACK Users Guide. Solution of Large Scale Eigenvalue Problem with Implicit Restarted Arnoldi Methods*. SIAM: Philadelphia, PA, 1998.
24. Maschhoff KJ, Sorensen DC. A portable implementation of ARPACK for distributed memory parallel architectures. *Proceedings of the Copper Mountain Conference on Iterative Methods*, vol. 1. April 1996; 9–16.