Parallel Solution of Sparse Linear Systems Arising in Advection–Diffusion Problems

Luca Bergamaschi[1], Giorgio Pini[1], and Flavio Sartoretto[2]

[1] Dipartimento di Metodi e Modelli Matematici per le Scienze Applicate
Universita' degli Studi, Via Belzoni 7, 35131 Padova, Italy
{berga,pini}@dmsa.unipd.it
[2] Dipartimento di Informatica, Universitá di Venezia
Via Torino 155, 30173 Mestre VE
sartoret@dsi.unive.it

Abstract. Flow problems permeate hydraulic engineering. In order to solve real–life problems, parallel solutions must be engaged, for attaining large storage amounts and small wall–clock time. In this communication, we discuss valuable key points which allow for the efficient, parallel solution of our large, sparse linear systems, arising from the discretization of advection–diffusion problems. We show that data pre-fetching is an effective technique to improve the efficiency of the sparse matrix–vector product, a time consuming kernel of iterative solvers, which are the best choice for our problems. Preconditioning is another key topic for the efficient solution of large, sparse, ill–conditioned systems. Up to now, no extensive theory for choosing the best preconditioner is available, thus ad–hoc recipes and sound based experience is mandatory. We compare many preconditioners in order to show their efficiency and allowing a good choice when attacking problems like ours.

1 Introduction

The advection–diffusion equations are [1]

$$\frac{\partial u}{\partial t} = \nabla \cdot (\mathbf{K}\nabla u - \mathbf{v}u) + \mathbf{f}, \tag{1}$$

where u is the unknown function, $\mathbf{K}$ is the diffusion tensor, $\mathbf{v}$ is a given velocity, 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. Finite Element (FE) integration in space over a 3D FE N–node grid is performed. Further integration in time by finite difference methods is performed by Crank-Nicolson scheme when $\mathbf{v} = 0$ [1], implicit Euler otherwise. One obtains a sequence of $N \times N$ linear algebraic systems, $Ax = b$. Classical FE methods yield large, sparse linear systems. When the flow velocity is to be accurately computed, the Mixed Hybrid Finite Element (MHFE) method are exploited. MHFE provides simultaneous solution of fluid pressure and velocity. In our framework, piecewise-constant pressure is considered, while velocities are approximated using the lowest order Raviart-Thomas elements [2]. MHFE requires the solution

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 804–814, 2005.

of sparse linear systems, which for a given mesh are 7∼8 times larger than FE ones.

Iterative methods are the best choice for solving our test problems, provided efficient preconditioners are available. When $\mathbf{v} = 0$, Symmetric Positive Definite (SPD) matrices are obtained; unsymmetric ones otherwise. For SPD matrices, the best available iterative algorithm is Preconditioned Conjugate Gradient (PCG), while for general, unsymmetric matrices no *best* iterative algorithm is available. On the ground of our experience, we selected the BiCGSTAB algorithm [3], which displays robustness and efficiency when attacking our problems.

A core, time consuming, sub–task inside all iterative methods is the matrix–vector product. We tested Algorithm 2 after Geus and Rollin [4], which attempts to enhance cache usage by data pre-fetching (DP) techniques. Table 1 shows our implementation of the algorithm.

```
       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(*), s, v, v1
c
       k = 1
       do i = 1, n
          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
```

Fig. 1. Our implementation of Algorithm 2 after Geus and Röllin.

Preconditioning is another key issue for the efficient solution of large linear systems. We exploited classical Jacobi, which does not improve convergence very much, but is both not storage consuming and easily efficiently parallelizable; we also tested the more powerful FSAI preconditioners [5], which we computed by our efficient, parallel implementation. For a typical range of Peclet number values, the quality of ILU(0) and ILUT [6] preconditioners is analyzed in [7]. The results can be easily extended to FSAI and pARMS type preconditioners.

At present, no general rules to identify the best solver for either diffusion or advection dominated problems are available. This is another motivation to our study, which is aimed to suggest good solution strategies for several situations.

2 The FSAI Preconditioner

Given a SPD matrix A, let $A = L_A L_A^T$ be its Cholesky factorization. The FSAI method computes an approximate inverse of A in the factorized form $H = G_L^T G_L$, where G_L is a sparse nonsingular lower triangular matrix approximating L_A^{-1}. To attain G_L, one must first prescribe a sparsity pattern $S_L \subseteq \{(i,j) : 1 \leq i \neq j \leq N\}$, such that $\{(i,j) : i < j\} \subseteq S_L$. A lower triangular matrix $\hat{G}_L$ is computed by solving the equations

$$(\hat{G}_L A)_{ij} = \delta_{ij}, \qquad (i,j) \notin S_L. \tag{2}$$

The diagonal entries of $\hat{G}_L$ are all positive. Defining $D = [\mathrm{diag}(\hat{G}_L)]^{-1/2}$ and setting $G_L = D\hat{G}_L$, the preconditioned matrix $G_L A G_L^T$ is SPD and has diagonal entries all equal to 1. A common choice for the sparsity pattern is to allow non zeros in G_L only in positions corresponding to non zeros in the lower triangular part of A^k, where k is a small positive integer, e.g., $k = 1, 2, 3$; see [8]. The extension of FSAI to the non symmetric case is straightforward; however the resolvability of the local linear systems and the non singularity of the approximate inverse is only guaranteed if all the principal sub-matrix of A are non singular (which holds true, for instance, when $A + A^T$ is SPD).

While the approximate inverses corresponding to A^k, $k > 1$, are often better than the one corresponding to $k = 1$, they may be too expensive to compute and apply. In [9] a simple approach, called *post-filtration*, was proposed to improve the quality of FSAI preconditioners in the SPD case. The method is based on a posteriori sparsification, by using a drop–tolerance parameter. We found that the quality of the preconditioner does not heavily depend upon its value, which ranges in the interval $[0, 1]$. The aim is to reduce the number of nonzero elements of the preconditioning factors, in order to decrease the arithmetic complexity of the iteration phase. In a parallel environment, a substantial reduction of the communication complexity of the preconditioner-by-vector multiplication can be achieved.

In the non symmetric case both preconditioner factors, G_L and G_U, must be sparsified. Non symmetric matrices with a symmetric nonzero pattern are considered, i.e. $S_L = S_U^T$ is assumed, and a symmetric filtration of factors G_L and G_U is performed.

3 Parallel Implementation

Our parallel implementation of the algorithms rely upon a data splitting approach, designed for sparse FE/MHFE matrix–vector (MV) products. The code is written in FORTRAN 90 and exploits MPI 1.0 calls for exchanging data among the processors. All our matrices are statically stored into CSR formatted data structures.

BiCGSTAB and PCG algorithms can be decomposed into a number of scalar products, `daxpy`-like linear combinations of vectors, $\alpha\mathbf{v}+\beta\mathbf{w}$, and MV products.

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

Concerning matrix splitting, note that uniform block mappings, like those exploited in High Performance Fortran `cyclic` directive, are not suitable for our sparse matrices. We splitted our matrices by a uniform, row–wise block mapping. Such distribution is ideal for our problems, since it allows for performing a piece of MV product on each processor. Moreover, our sparse matrices have quite the same number of non–zero entries per row, hence blocks consisting of the same number of rows consist of quite the same amount of bytes. We exploited blocks of *contiguous* rows. Non–contiguous row distributions yield more complex algorithms, which moreover do not perform well on (old) machines where the communication time changes with the relative position of processors in the communication net. We improved MV evaluation by using a technique for minimizing data communication between processors [10]. In the greedy matrix-vector algorithm, each processor communicates with each other. Using our approach with our sparse matrices, usually each processor sends/receives data to/from at most 2 other processors. Moreover, when running on P processors, the amount of data exchanged, when dealing with a matrix featuring M non–zero entries, is far smaller than $[M/P]$.

3.1 Parallel Implementation of FSAI

We implemented the FSAI preconditioner computation, both for SPD, and non symmetric matrices. Our code allows for the specification of either A or A^2 sparsity patterns. We used a block row distribution of matrices A, G_L (and also G_U in the non symmetric case). Complete rows are assigned to different processors.

Let n_i be the number of non zeros allowed in the i-th row of G_L. In the SPD case, any row i of the G_L matrix can be computed independently of each other, by solving a small SPD dense linear system of size n_i. To attain parallelism, the processor that computes row i must access n_i rows of A. Since the number of non local rows needed by each processor is relatively small, we temporarily replicate the non local rows on auxiliary data structures. The dense factorizations needed are carried out using BLAS3 routines from LAPACK. Once G_L is obtained, a parallel transposition routine provides to every processor the eligible part of G_L^T.

In the non symmetric case, recall that we assume a symmetric non zero pattern for matrix A, i.e. we ideally set $S_L = S_U^T$. The preconditioner factor G_L is computed as described before, while G_U is computed by columns. Hence, no

additional row exchange is needed with respect to the SPD case. Every processor performs a fully parallel computation both of a set of rows into G_L, and of a set of columns into G_U.

4 pARMS

The parallel Algebraic Recursive Multilevel Solvers (pARMS) package [11, 12] is an interesting effort in devising distributed preconditioners for iterative solvers. It works in the framework of distributed linear systems, which provides an algebraic representation for the parallel solution of linear systems, $Ax = b$, arising in Domain Decomposition Methods. The coefficient matrix A is split among the available processors. A local $N_i \times N_i$ matrix, A_i, and an interface matrix, X_i are assigned to the i-th processor. Each local vector of unknowns, x_i, is split into a sub–vector u_i of interior variable contributions, and a sub-vector y_i of inter–domain interface variables. Analogously, each local right–hand side vector, b_i, is chopped into f_i and g_i contributions. The equations assigned to processor i can be written

$$\begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix} \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix}. \tag{3}$$

where the matrices B_i, F_i, E_i, C_i compose a block–splitting of A_i. Additive Schwarz techniques (with or without overlapping), can be exploited, as well as Schur complement–type ones. It is well known that scalability and robustness of Additive Schwarz can be very poor [12]. We found that Schur techniques are better suited to our problems. These latter techniques rely upon Schur complement systems. They are derived by eliminating the variables u_i in equation (3), using $u_i = B_i^{-1}(f_i - F_i y_i)$. By substitution in the second equation, one gets

$$S_i y_i = \sum_{j \in N_i} E_{ij} y_j = g_i - E_i B_i^{-1} f_i = g_i', \tag{4}$$

where S_i is the *local* Schur complement

$$S_i = C_i - E_i B_i^{-1} F_i.$$

Assembling equations (4) over all processors, the *global Schur complement system*

$$Sy = \begin{pmatrix} S_1 & E_{1,2} & \dots & E_{1,p} \\ E_{2,1} & S_2 & \dots & E_{2,p} \\ \vdots & \vdots & \dots & \vdots \\ E_{p,1} & E_{p-1,2} & \dots & S_p \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} = \begin{pmatrix} g_1' \\ g_2' \\ \vdots \\ g_p' \end{pmatrix}$$

is obtained. The matrix S is the *global* Schur complement. Once the system is approximately solved, each processor works out the system $B_i u_i = f_i - E_i y_i$, hence attaining an approximated solution x of the original problem. By extensive

testing, we found that for our problems the `lsch_ilut` (left–Schur complement) algorithm performs better w.r.t. the Additive Schwarz preconditioners, right Schur complement preconditioners, and Gauss–Seidel preconditioners, which are described in [12]. The `lsch_ilut` approach solves the global Schur complement system via Block–Jacobi preconditioning. Note that pARMS package suffers from the need of assessing a large number of parameters (about 15), corresponding to the high number of possibilities that can be exploited.

pARMS algorithms are intrinsically parallel. Note that the number of iterations performed heavily depend upon the number of engaged processors, hence pARMS parallel performance analysis cannot naively rely upon classical parameters, like speedup.

5 Numerical Results

Table 1 shows the main characteristics of our test problems.

Table 1. Main characteristics of our test matrices. N=size, n_z=number of non–zero elements, dd= "is it a diagonal dominant matrix?"; HB=half–bandwidth; type = Matrix type: spd= symmetric positive definite, problem = problem type, algorithm = discretization algorithm, uns= unsymmetrical matrix. Solvers: P=PCG, B=BiCGSTAB.

#	N	n_z	dd	HB	type	problem	algorithm	solver
1	268,515	3,926,823	Y	5265	spd	diffusion	FE	P
2	390,160	2,694,160	N	18062	spd	diffusion	MHFE	P
3	531,765	7,817,373	Y	5265	spd	diffusion	FE	P
4	1,059,219	15,605,175	N	20769	spd	diffusion	FE	P
5	1,317,141	19,458,621	N	13041	uns	adv-diff	FE	B
6	2,097,669	31,066,125	N	20769	spd	diffusion	FE	P
7	2,635,731	38,927,991	N	51681	uns	adv-diff	FE	B
8	3,096,640	21,528,640	N	71966	spd	diffusion	MHFE	P

We stopped the iterations when the euclidean norm of the residual, $r_k = b - Ax_k$, satisfies $\|r_k\| \simeq 10^{-12}$.

We performed our runs on the IBM SP4 system and the IBM Linux Cluster 1350 machine, both located at CINECA supercomputing center, Italy.

The SP4 machine features 16 nodes, each one including 32 POWER 4, 1300 MHz processors. Each node is equipped with a 64 GB memory, one only node featuring a 128 GB core memory. The nodes are connected with 2 interfaces to a dual plane set of Colony switches.

The IBM Linux Cluster 1350, CLX for short, is a 256 node machine. Each node encloses a 2GB DRAM (32 nodes have a 4GB DRAM), and two Intel Xeon Pentium IV 3.055 GHz processors (10 I/O nodes feature 2.8 GHz processors). Each processor is equipped with a 512Kb L2 Cache. Disk space is 5.5 TB. The internal network is a *Myrinet IPC* one.

Table 2. Wall–clock seconds spent on the SP4, to solve our test problems up to $\|r_k\| \simeq 10^{-12}$ accuracy. D = PCG + diagonal preconditioning; F = PCG + FSAI(A) preconditioning; F1 = PCG + FSAI(A^2) preconditioning; D*, F*, F1* are D, F, F1, respectively, where *no data prefetching* inside matrix–vector products was exploited.

N	alg	T_1	T_2	T_4	T_8	T_{16}	T_{32}	S_2	S_4	S_8	S_{16}	S_{32}
2,097,669	D*	2611.0	1391.2	834.0	534.9	287.4	151.8	1.88	3.13	4.88	9.08	17.20
	F*	1976.0	1111.3	622.4	408.2	205.2	110.2	1.78	3.17	4.84	9.63	17.93
	F1*	1548.4	858.6	470.6	347.2	176.3	106.8	1.80	3.29	4.46	8.78	14.50
	averages	2045.1	1120.4	642.3	430.1	223.0	122.9	1.82	3.20	4.73	9.17	16.54
2,097,669	D	1822.7	985.5	555.1	375.1	201.6	102.4	1.85	3.28	4.86	9.04	17.80
	F	1236.6	655.6	369.6	243.3	144.2	91.2	1.89	3.35	5.08	8.58	13.56
	F1	1067.7	570.5	315.0	211.2	118.6	79.6	1.87	3.39	5.06	9.00	13.41
	averages	1375.7	737.2	413.2	276.5	154.8	91.1	1.87	3.34	5.00	8.87	14.92

Table 2 compares, on an appropriate test matrix, the performance on the SP4 of our PCG code either with or without, data pre-fetching (DP) by Geus & Rollin. The value T_p is the wall–clock seconds spent to solve a problem; $S_p = T_1/T_p$ is the classical speedup value. One can see that appreciably less wall–clock seconds are spent to solve our test problems when the DP technique is exploited. The average time over all tests goes down from 764.0 seconds when no DP is used, to as less as 508.1, which is only 67% of the former time, when DP is exploited. On the other hand, the speedup values are quite similar; their average values over all tests are 7.09 for no DP, vs 6.80 with DP. Concerning the assessment of the parameters in pARMS, we extensively engaged the package on our problems. We tested $s = 30, 50, 80, 100$ Krylov subspace sizes. We found that a good choice is using flexible GMRES (FGMRES) [13], together with $s = 100$. The `lsch_ilut` preconditioner with overlapping was enrolled. The fill-in parameter was set to `lfill`=60 for all the recursion levels, and the dropping tolerance was `tol`=10^{-4}. The group independent set size was set to 5000, while the maximum number of internal iterations was 5. These values are also suggested in [11].

Table 3 shows the wall-clock times and *relative* speedup values, $S_p^{(r)} = T_{p/2}/T_p$, $p = 2, 4, 8, 16, 32$, recorded on the SP4 when solving our test problems (all obtained by exploiting DP technique). Note that the I/O time needed for data input is not considered. The time for printing output results is negligible.

The smallest matrices (problems 1 and 2) were solved on up to 16 processors. A larger number of processors would assign a too small data set to each one. To solve the larger problems, up to 32 processors were engaged. Inspecting Table 3 one can see that FSAI(A) allows for a slight decrease of the computing time, over Jacobi, while FSAI(A^2) with drop–tolerance value 0.1 provides appreciable enhancements over FSAI(A) and Jacobi.

One can see that pARMS in the smaller problems (1–6) is usually more time consuming than the other methods, while it is comparably expensive in the larger problem 7. In spite of the fact that we made extensive parameter space analysis, we could not attain pARMS convergence on problem 8. It is well known that pARMS suffer from high changes in the iteration number, depend-

Table 3. Analogous to the previous Table. Du = BiCGSTAB + Jacobi; Fu = BiCGSTAB + FSAI(A); F1u = BiCGSTAB + FSAI(A^2); pA = pARMS. Legend for the symbols: "*" = no convergence attained; "-" = value not computed, "/" = value not computable.

#	N	alg	T_1	T_2	T_4	T_8	T_{16}	T_{32}	$S_2^{(r)}$	$S_4^{(r)}$	$S_8^{(r)}$	$S_{16}^{(r)}$	$S_{32}^{(r)}$
1	268,515	D	448.0	230.6	136.5	84.5	52.2	-	1.94	1.69	1.62	1.62	/
		F	382.9	198.4	116.7	79.8	49.1	-	1.93	1.70	1.46	1.63	/
		F1	86.2	45.3	25.6	16.1	9.3	-	1.90	1.77	1.59	1.73	/
		pA	131.3	91.9	86.8	62.2	50.7	-	1.43	1.06	1.40	1.23	/
2	390,160	D	201.2	105.8	62.5	42.9	24.3	-	1.90	1.69	1.46	1.77	/
		F	183.2	98.1	54.8	37.6	21.3	-	1.87	1.79	1.46	1.77	/
		F1	140.1	73.4	42.3	28.4	15.0	-	1.91	1.74	1.49	1.89	/
		pA	*	235.8	189.8	217.9	*	-	/	1.24	0.87	/	/
3	531,765	D	1519.3	824.9	459.1	299.7	138.8	88.3	1.84	1.80	1.53	2.16	1.57
		F	1542.6	867.9	467.5	323.1	150.4	98.7	1.78	1.86	1.45	2.15	1.52
		F1	236.0	126.7	69.9	45.3	21.6	13.5	1.86	1.81	1.54	2.10	1.60
		pA	554.3	169.2	140.3	183.9	136.2	131.7	3.28	1.21	0.76	1.35	1.03
4	1,059,219	D	2522.7	1344.6	778.2	549.2	270.4	140.3	1.88	1.73	1.42	2.03	1.93
		F	2044.1	1050.5	598.9	479.4	253.7	136.1	1.95	1.75	1.25	1.89	1.86
		F1	770.9	412.3	229.1	174.4	93.6	52.3	1.87	1.80	1.31	1.86	1.79
		pA	*	351.8	245.0	216.8	176.3	92.3	/	1.44	1.13	1.23	1.91
5	1,317,141	Du	666.6	369.1	192.0	132.7	70.8	46.5	1.81	1.92	1.45	1.87	1.52
		Fu	527.6	280.3	163.4	113.2	64.9	45.3	1.88	1.72	1.44	1.74	1.43
		F1u	*	*	170.1	102.6	62.0	41.3	/	/	1.66	1.65	1.50
		pA	*	*	*	202.8	133.6	79.7	/	/	/	1.52	1.68
6	2,097,669	D	1822.7	985.5	555.1	375.1	201.6	102.4	1.85	1.78	1.48	1.86	1.97
		F	1236.6	655.6	369.6	243.3	144.2	91.2	1.89	1.77	1.52	1.69	1.58
		F1	1067.7	570.5	315.0	211.2	118.6	79.6	1.87	1.81	1.49	1.78	1.49
		pA	*	*	*	652.7	376.2	226.5	/	/	/	1.73	1.66
7	2,635,731	Du	*	*	768.9	546.6	380.2	295.2	/	/	1.41	1.44	1.29
		Fu	*	*	*	536.0	321.6	271.4	/	/	/	1.67	1.18
		F1u	*	*	*	419.3	254.1	212.4	/	/	/	1.65	1.20
		pA	*	*	*	419.7	295.3	191.2	/	/	/	1.42	1.54
8	3,096,640	D	6202.0	3423.4	1949.1	1612.2	714.7	350.7	1.81	1.76	1.21	2.26	2.04
		F	4950.2	2656.0	1506.4	1132.2	574.6	307.2	1.86	1.76	1.33	1.97	1.87
		F1	3801.6	2008.9	1176.0	924.3	428.7	242.8	1.89	1.71	1.27	2.16	1.77
		pA	*	*	*	*	*	*	/	/	/	/	/
		averages	1410.8	715.7	418.0	337.6	186.8	145.1	1.92	1.68	1.38	1.76	1.61

ing upon the number of processors. Note that in many problems pARMS could not be run on 1 or 2 processors, due to lack of core memory. When the relative speedup can be measured, it displays large oscillations, and questionable values (e.g. $S_2^{(r)} = 3.28$ for N=531,765). From this point of view, PCG and BiCGSTAB are more robust on this kind of problems. The average standard deviation in the number of iterations, counting all our PCG and BiCGSTAB tests, is 12.9, while for pARMS is 90.3. We feel that solving even larger problems on

Table 4. Analogous to Table 3. Problem 3, N=531,765, results obtained on the CLX system.

N	alg	T_1	T_2	T_4	T_8	T_{16}	T_{32}	$S_2^{(r)}$	$S_4^{(r)}$	$S_8^{(r)}$	$S_{16}^{(r)}$	$S_{32}^{(r)}$
531,765	D	1726.7	908.7	473.3	250.7	133.2	74.2	1.90	1.92	1.89	1.88	1.80
	F	1560.3	821.2	437.4	237.7	139.6	83.3	1.90	1.88	1.84	1.70	1.68
	F1	287.9	163.3	90.9	51.1	28.7	16.4	1.76	1.80	1.78	1.78	1.75
	pA	521.3	141.6	114.6	166.4	92.3	80.7	3.68	1.24	0.69	1.80	1.14

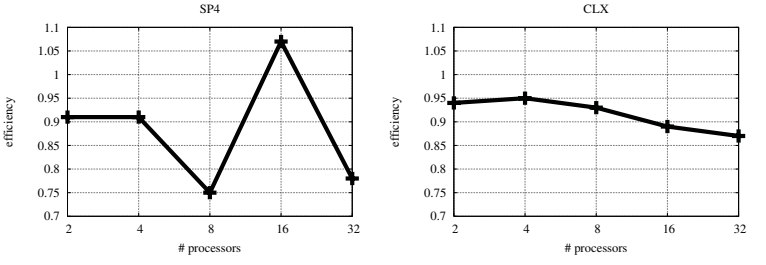

Fig. 2. Problem 3, N=531,765. Average relative efficiency on D, F, and F1 methods, obtained when running on the SP4 (left frame), and CLX (right frame).

a larger number of processors, pARMS could perform better. Figure 2 shows the average *relative* efficiency, $E_p^{(r)} = S_p^{(r)}/2$, on problem 3, recorded either on the SP4 (left frame) or on the CLX (right frame). The average was performed on D, F, and F1 algorithms; the pA technique was not considered: recall that the cost of the algorithm heavily depends upon the number of running processors, hence plain efficiency is meaningless. Usually, the relative efficiency on the SP4 is quasi optimal for $p = 2$ processors, good for 4, worsens when doubling to 8 processors, acceptable on more than 8. Such a behavior is typical for our problems, when running on CINECA's machine. The interconnecting network is not so fast as to allow high speedup values on a large number of processors (see also [4, 14]). The performance degradation when going from 4 to 8 processors occurred in all our parallel experiences on this machine, due to hard/soft processor aggregation into virtual/physical nodes. Since 8 processors share the same node core memory, when all 8 are engaged on unstructured matrix computations, many memory conflicts are raised. For comparison, Table 4 shows the time and speedup recorded on the CLX system, for problem 3 (N=531,765). Comparing with the results on the SP4 after Table 3, one can see that the performance is usually better on the CLX. A slight performance decrease is recorded when going from 1 to 2 processors. Recall that a CLX node encompasses two processors, sharing the node core memory. One can see that on the CLX the efficiency behavior matches the parallel expert feeling. This result confirms that the disturbing low performance on the SP4, when running on 8 processors, is due to the machine architecture, rather than to our algorithm, which performs well on other architectures, like CLX.

Summarizing, the parallel performance on the SP4 is more erratic than on the CLX, but note that the largest problems cannot run on a 2GB CLX node, unless a suitably large number of processors is engaged.

The parallel degrees obtained on the SP4 are compatible with the exploited machine, in accordance with the degrees shown e.g. in [4].

6 Conclusions

Summarizing, the PCG algorithm for SPD problems and BiCGSTAB for unsymmetric ones, equipped with FSAI(A^2) preconditioning, prove to be the best parallel solvers for our problems, on our tests.

Data pre-fetching allows for appreciably improving the efficiency of our sparse matrix–vector products.

The parallel efficiency of our code on the SP4 can be rated satisfactory. Our results provide a guideline for the parallel performance that one can expect when running FE codes. Parallel performance losses can be recorded running on 8 processors, due to the complex, highly non uniform, SP4 architecture. This problem does not occur on the CLX, where typical parallel performance results are achieved.

Acknowledgments

This work has been supported by the italian MIUR project *Numerical models for multi-phase flow and deformation in porous media*.

References

1. Gambolati, G., Pini, G., Tucciarelli, T.: A 3-D finite element conjugate gradient model of subsurface flow with automatic mesh generation. Adv. Water Resources **3** (1986) 34–41
2. Brezzi, F., Fortin, M.: Mixed and Hybrid Finite Element Methods. Springer-Verlag, Berlin (1991)
3. van der Vorst, H.A.: Bi-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **13** (1992) 631–644
4. Geus, R., Röllin, S.: Towards a fast parallel sparse symmetric matrix-vector multiplication. Parallel Computing **27** (2001) 883–896
5. Yu. Kolotilina, L., Yu. Yeremin, A.: Factorized sparse approximate inverse preconditionings I. Theory. SIAM J. Matrix Anal. Appl. **14** (1993) 45–58
6. Saad, Y.: ILUT: A dual threshold incomplete lu factorization. Numer. Linear Alg. Appl. **1** (1994) 387–402
7. Pini, G., Putti, M.: Krylov methods in the finite element solution of groundwater transport problems. In Peters, A., Wittum, G., Herrling, B., Meissner, U., Brebbia, C.A., Gray, W.G., Pinder, G.F., eds.: Computational Methods in Water Resources X, Volume 1, Dordrecht, Holland, Kluwer Academic (1994) 1431–1438

8. Kaporin, I.E.: New convergence results and preconditioning strategies for the conjugate gradient method. Numer. Linear Alg. Appl. **1** (1994) 179–210
9. Yu. Kolotilina, L., Nikishin, A.A., Yu. Yeremin, A.: Factorized sparse approximate inverse preconditionings IV. Simple approaches to rising efficiency. Numer. Linear Alg. Appl. **6** (1999) 515–531
10. Bergamaschi, L., Putti, M.: Efficient parallelization of preconditioned conjugate gradient schemes for matrices arising from discretizations of diffusion equations. In: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing. (March, 1999) (CD–ROM).
11. Li, Z., Saad, Y., Sosonkina, M.: pARMS: a parallel version of the algebraic recursive multilevel solver. Numer. Linear Alg. Appl. **10** (2003) 485–509
12. Saad, Y., Sosonkina, M.: pARMS: a package for solving general sparse linear systems of equations. In Wyrzykowski, R., Dongarra, J., Paprzycki, M., Wasniewski, J., eds.: Parallel Processing and Applied Mathematics. Volume 2328 of Lecture Notes in Computer Science., Berlin, Springer-Verlag (2002) 446–457
13. Saad, Y.: Iterative Methods for Sparse Linear Systems. Second edition. SIAM, Philadelphia, PA (2003)
14. Bergamaschi, L., Pini, G., Sartoretto, F.: Computational experience with sequential and parallel preconditioned Jacobi Davidson for large sparse symmetric matrices. J. Comput. Phys. **188** (2003) 318–331