# A Parallel Row Projection Solver for Large Sparse Linear Systems

M. D'Apuzzo and M. Lapegna

Dipartimento di Matematica e Applicazioni
Università di Napoli "Federico II"
Napoli, Italy, 80126

## Abstract

*In this paper we present a parallel iterative solver for large and sparse nonsymmetric linear systems. The solver is based on a row-projection algorithm, derived from the symmetrized block version of the Kaczmarz method with Conjugate Gradient acceleration. A comparison with some Krylov subspace methods shows the remarkable robustness of this algorithm when applied to systems with eingevalues arbitrarily distributed in the complex plane. The parallel version of the algorithm was developed for MIMD distributed memory machines and it is based on a row partitioning approach which allows to compute each iteration as a simultaneous set of independent least squares problems. Moreover, we propose a data distribution strategy leading to a scalable communication scheme. The algorithm has been tested both on a system Intel iPSC/860 and on the Intel Touchstone DELTA System, running the Intel NX message passing environment.*

## 1 Introduction

The solution of large and sparse linear systems is often the computational kernel of many scientific and engineering applications. Therefore, the research interest in this area has recently focused on the development of efficient parallel solvers. This requires both the investigation of methods with parallel properties and the study of suitable matrix block partitioning and data distribution strategies in order to obtain scalable parallel algorithms.
In this paper we describe and test a parallel *row-projection* algorithm for solving linear systems

$$Ax = b, \qquad (1)$$

where $A$ is a real nonsingular $N \times N$ matrix, on *distributed memory multiprocessors.* We assume that the rows of $A$ are partitioned into $L$ blocks

$$A = [A_1^T, A_2^T, ..., A_L^T]^T , \qquad (2)$$

and that the vector $b$ is partitioned conformally, $b = [b_{[1]}^T, b_{[2]}^T, ..., b_{[L]}^T]^T$. As the name suggests, a row-projection (RP) method is any iterative method which involves the projections of a vector onto the *range* of $A_t^T$, $\mathcal{R}(A_t^T)$, $t = 1, ..., L$. Contrary to other iterative solvers, RP methods do not place any restriction on the eigenvalue distribution of $A$ and can be also applied to nonsymmetric linear systems with indefinite symmetric parts. However, slow convergence has prevented their widespread use. Further, there has not been a suitable theory or method on how to select row partitionings for practical problems.
RP methods have been recently applied to nonselfadjoint elliptic PDE's in two and three dimensions, demonstrating both their numerical robustness and potential parallelism [1, 2, 9].
The RP algorithm examined in this paper is based on the general block version of the *Kaczmarz method.* Its iteration matrix is the product of orthogonal projectors. In particular, we consider symmetrizing the iteration matrix by following a forward sweep through the rows with a backward sweep, so that the *Conjugate Gradient acceleration* can be applied.
The computationally demanding aspect of the Kaczmarz method is the computation, in each iteration, of an orthogonal projection and hence the solution of a linear least squares problem. However, if it possible to permute the rows of the matrix $A$ so as to have separate subblocks in each larger block $A_t$, each projection can be computed as a set of independent linear least squares problems of smaller size. These independent subproblems can be solved simultaneously.
In Section 2 we outline the method and its convergence properties. Moreover, we show some results obtained by a comparison of the method with other existing

1066-6192/95 $4.00 © 1995 IEEE

ones, such as CG-like methods. In Section 3 we describe the parallel algorithm. In particular, we propose a data distribution strategy that allows to minimize the communication needed at the end of each iteration. Finally, in Section 4 we present a parallel evaluation of the algorithm by implemeting it on two distributed memory machines. All tests are performed on block tridiagonal systems.

## 2 Description of the method

Let the system (1) be partitioned as in (2). We suppose that all $A_t$ have equal size, that is $A_t \in \Re^{D \times N}$, with $D = N/L$. The row projection algorithm examined here derives from the block version of the Kaczmarz method [3, 5]:

$$\begin{aligned} &x^{(0)} \ \textit{arbitrary} \\ &x^{(k+1)} = x^{(k)} + A^T_{t(k)}\Omega_k(b_{[t(k)]} - A_{t(k)}x^{(k)}) \qquad (3) \\ &k = 0, 1, \ldots ; \qquad t(k) = k(\text{mod} L) + 1, \end{aligned}$$

where $\Omega_k$ is a $D \times D$ *relaxation matrix*. For the $L = N$ case the method (3) becomes the iterative scheme initially proposed by S. Kaczmarz in [8]. In [5] it is shown that a sufficient condition for the convergence of the sequence $\{x^{(k)}\}$ to the solution of (1) is:

$$\limsup_{k \to \infty} \|A^+_{t(k)}(I_M - A_{t(k)}A^T_{t(k)}\Omega_k)A_{t(k)}\| < 1 ,$$

where $I_D$ is the $D \times D$ identity matrix, $A^+_t$ denotes the Moore Penrose inverse of $A_t$ and $\|\cdot\|$ stands for the euclidean norm. A strong way of satisfying the above sufficient condition and of obtaining a stationary iterative procedure is by choosing $\Omega_k = (A_{t(k)}A^T_{t(k)})^{-1}$, $\forall k \geq 0$. With this choice, and by introducing an iteration parameter $\omega$, we have the following method

**BRPK** (**B**lock **R**ow **P**rojection **K**aczmarz) method

$$\begin{aligned} &x^{(0)} \ \textit{arbitrary} \\ &x^{(k+1)} = x^{(k)} + \omega A^+_{t(k)}(b_{[t(k)]} - A_{t(k)}x^{(k)}) \\ &k = 0, 1, \ldots ; \qquad t(k) = k(\text{mod} L) + 1 \end{aligned}$$

If we now consider a forward sweep through the blocks $A_t$ starting from $A_1$, we can write the BRPK method in the following classical form

$$x^{(k+1)} = B(\omega)x^{(k)} + R(\omega)b, \quad k \geq 0 \qquad (4)$$

where

$$B(\omega) = \textstyle\prod_{t=L}^{1} M_t \ , \quad M_t = I - \omega P_t \ , \quad P_t = A^+_t A_t$$

$$R(\omega) = \omega[T_1 A^+_1, \ldots, T_L A^+_L], \quad T_j = \textstyle\prod_{t=L}^{j+1} M_t, \quad T_L = I$$

We explicity observe that $P_t$ is the orthogonal projector onto $\mathcal{R}(A^T_t)$.
A fundamental result for the convergence of the iteration (4) is the following:

**Theorem 2.1** :
*The BRPK method converges to the solution of (1) if and only if* $0 < \omega < 2$.

The proof of this theorem is based on the classic theory of linear stationary iterative processes [4, 6].
The theoretical robustness of the BRPK method is remarkable and convergence is assured, under the hypothesis of the above theorem, even when $A$ is singular or rectangular. However, the convergence speed is determined by the spectral radius of $B(\omega)$, which in turn depends on the angles between the subspaces $\mathcal{N}(A_t)$, the nullspace of $A_t$. This angles can be small, with a correspondingly slow rate of convergence. For this reason we consider a symmetrization process for the matrix $B(\omega)$ by following a forward sweep through the blocks with a backward one. In this way the BRPK method becomes:

**SBRPK** (Symmetric BRPK) method

$$x^{(k+1)} = Q(\omega)x^{(k)} + T(\omega)b, \quad k \geq 0,$$

where

$$Q(\omega) = (I - \omega P_1) \cdots (I - \omega P_L)^2 (I - \omega P_{L-1}) \cdots (I - \omega P_1)$$

The $i^{th}$ block column of $T(\omega)$, $T^i(\omega)$, is given by:

$$T^i(\omega) = \prod_{t=1}^{i-1} M_t \left[ I + \prod_{t=i}^{L} M_t \prod_{t=L}^{i+1} M_t \right] (A^+_i)$$

where the first product is $I$ when $i = 1$ and the third product should be interpreted as $I$ when $i = L$.
For the SBRPK method the convergence theorem 2.1 still holds. Moreover, when $0 < \omega < 2$ the matrix $(I - Q(\omega))$ is positive definite and so it is possible to use the Conjugate Gradient (CG) method as acceleration procedure for the system

$$(I - Q(\omega))x = T(\omega)b \qquad (5)$$

One of the main implementation issue is the choice of $\omega$ in (5). In [9] it was shown that the "optimal" value for the relaxation parameter $\omega$ is 1 when the matrix $A$ is partitioned into two block rows, i.e., $L = 2$. Although this is no longer true for $L > 2$, the choice $\omega = 1$ is still a reasonable one for many reasons, as described in [1]. Therefore, for the remainder of the paper $\omega = 1$ will be used. This choice and the fact that $P_t$ is a projector lead to a simplification of the expressions of the matrix $Q = Q(1)$ and $T = T(1)$:

$$Q = (I - P_1)\cdots(I - P_L)(I - P_{L-1})\cdots(I - P_1)$$

$$T^i = \prod_{t=1}^{i-1} M_t \left[I + \prod_{t=i}^{L} M_t \prod_{t=L-1}^{i+1} M_t\right](A_i^+)$$

In conclusion, the final scheme of the algorithm we consider is the following:

**SBRPK with CG acceleration**

A. **Preprocessing**

```
$x^{(0)} = 0$
compute $r^{(0)} = Tb$
$v^{(0)} = r^{(0)}$;  $k = 0$
```

B. **Conjugate Gradient iterations**

```
$\boxed{y^{(k)} = (I - Q)v^{(k)}}$
$\alpha^{(k)} = (r^{(k)}, r^{(k)})/(y^{(k)}, v^{(k)})$
$x^{(k+1)} = x^{(k)} + \alpha^{(k)}v^{(k)}$
$r^{(k+1)} = r^{(k)} - \alpha^{(k)}y^{(k)}$
$\beta^{(k+1)} = (r^{(k+1)}, r^{(k+1)})/(r^{(k)}, r^{(k)})$
$v^{(k+1)} = r^{(k+1)} + \beta^{(k+1)}v^{(k)}$
```

C. **if the convergence test is not satisfied, set $k = k + 1$ and go to B.**

which will be called simply SBRPK in the sequel.

## 2.1 Row partitioning strategies for parallelism

On each iteration the SBRPK method requires the computation of the product

$$y = (I - P_1)(I - P_2)\cdots(I - P_L)(I - P_{L-1})\cdots(I - P_1)v$$

and hence the solution of $2L - 1$ linear least squares problems of the form

$$w = \min_z \|u - A_t^T z\|_2. \qquad (6)$$

Consequently, it is important to select a row partitioning of $A$ in order to have least squares problems efficiently computable. The basic idea to achieve such a goal is to partition tha matrix $A$ so that each block $A_t$ consists of subblocks $C_{t,j}$ that are orthogonal to each other: then, the corresponding projection $P_t$ can be computed as a simultaneous set of smaller least squares subproblems. This subproblems are independent and can then be solved in parallel.
In particular, we consider the case when the matrix $A$ is a $d \times d$ block tridiagonal matrix, that is, when $A = tridiag[T_i, D_i, E_i]$ with $T_i$, $D_i$, $E_i \in \Re^{d\times d}$, $i = 1, ..., d$. For such a matrix a suitable row permutation allows a block partition with separate subblocks in each larger block $A_t$. To illustrate this idea, suppose $d = 12$ and multiply $A$ by a suitable permutation matrix in order to obtain the following three-blocks partition:

$$\hat{A} = \left[\begin{array}{cccccccccccc}
D_1 & E_1 & & & & & & & & & & \\
 & & T_4 & D_4 & E_4 & & & & & & & \\
 & & & & & T_7 & D_7 & E_7 & & & & \\
 & & & & & & & & T_{10} & D_{10} & E_{10} & \\
\hdashline
T_2 & D_2 & E_2 & & & & & & & & & \\
 & & & T_5 & D_5 & E_5 & & & & & & \\
 & & & & & & T_8 & D_8 & E_8 & & & \\
 & & & & & & & & & T_{11} & D_{11} & E_{11} \\
\hdashline
 & T_3 & D_3 & E_3 & & & & & & & & \\
 & & & & T_6 & D_6 & E_6 & & & & & \\
 & & & & & & & T_9 & D_9 & E_9 & & \\
 & & & & & & & & & & T_{12} & D_{12}
\end{array}\right] =$$

$$= [C_{1,1}{}^T, ..., C_{1,4}^T | C_{2,1}^T, ..., C_{2,4}^T | C_{3,1}^T, ..., C_{3,4}^T]^T =$$
$$= [A_1^T | A_2^T | A_3^T]^T \qquad (7)$$

From (7) we observe that $A_t$, $t = 1, 2, 3$, has 4 disjoint subblocks. Then, the least squares problem corresponding to $A_t$ can be decomposed into 4 smaller subproblems

$$w_i = \min_z \|u_{[i]} - C_{t,i}^T z\|_2, \quad i = 1, ..., 4$$

which are independent and can be solved in parallel.
We make the following two remarks. First, the row partitioning should be chosen not only to allow parallelism in the computations, but also to yield subproblems which can be easily solved, require at most $O(N)$ additional storage to be solved and are well conditioned. Moreover, another criterion for a row partitioning is that the spectrum of $Q$ is suitable for the CG method. Since it may proved that $1/L$ eigenvalues of $(I - Q)$ are exactly 1 [1], the latter goal can be achieved by keeping the number of blocks $L$ as small as possible.

Although these goals are conflicting, there are important classes of problems for which it is possible to flexibly satisfy all of the criteria previously described. Linear systems drawn for two dimensional elliptic partial differential equations are one such class [1, 2, 9]

The second remark is that (7) is not the only partition that gives disjoint blocks in each larger block $A_t$. For instance, we can consider a two-blocks partition,

$$A = [A_1^T, A_2^T]^T \ , \qquad (8)$$

by putting the row blocks $[E_i, D_i, T_i]$, $i = 1, 2, 5, 6, 9, 10$ into $A_1$ and the remaining row blocks into $A_2$. In this way, both blocks have 3 separate subblocks, each consisting of two row blocks:

$$A_1 = (1,2), (5,6), (9,10)$$
$$A_2 = (3,4), (7,8), (11,12) \ ,$$

where the parentheses indicate the separation between subblocks. Other examples of possible partition and a their comparison are shown in [2]. We remark that there is a corresponding loss of possible concurrency when the size of subproblems is increased. By comparing the partition (7) and (8), we observe that for the three-blocks partition the maximun concurrency in solving (6) is $d/3$, while for the partition (7) is $d/4$. In this paper, we shall assume that the SBRPK method is applied to matrices partitioned as in (7).

## 2.2 A comparison with other methods

In this subsection we present some results about a comparison of the SBRPK method with other iterative solvers. In particular we consider three CG-like methods from the NSPCG package [10]: the *Generalized Conjugate Residual* (GCR(k)), the *Generalized Minimum Residual* (GMR(k)) and the *Orthomin* (OMN(k)) [11, 12]. These methods are implemented in a truncated and/or restarted version; $k$ represents the number of search directions to be stored. The package NSPCG also provide an option for preconditioning. We analyze the ILU$(s,\alpha)$, MILU$(s,\alpha)$, and SSOR$(\alpha)$ preconditioners.

It well known that both OMN(k) and CGR(k) converge if the symmetric part of $A$ is positive definite. GMR(k) is known to converge for any nonsingular matrix if a sufficiently large value of $k$ is used; however, an increase in $k$ leads an increase in the memory requirements and in the computational work load. Finally, even with a reasonable value of $k$, the residual of GMR(k) can fail to decrease to zero.

We restrict out tests to the case when the matrix $A$ is obtained from the application of 5-point central differences operators to two-dimensional elliptic partial differential equations with Dirichlet boundary conditions on the unit square. An uniform grid of size $h = 1/(d+1)$ is used for both the $x$ and $y$ coordinates so that $A$ is of order $d^2$. Moreover, the matrix $A$ is block tridiagonal, with each diagonal block being a tridiagonal matrix of order $d$ and each off-diagonal blocks being a diagonal matrix. The solution is assumed known in order to compute the right-hand side function $f$. The test problem we consider are:

1 $\quad -u_{xx} - [(1+xy)u_y]_y - \beta[cos(x)u_x + (e^{-x} + x)u_y]$
$\quad +3u = f \ , \quad \beta = 10000$

2 $\quad -u_{xx} - u_{yy} - xu_x + 200yu_y - 300u = f$

3 $\quad -u_{xx} - u_{yy} + 1000e^{xy}(u_x - u_y) = f$

with the solution $u(x,y) = x + y$.

These problems represent a variety of eigenvalue distributions for the matrix $A$. In particular, problems 1 and 3 both have eigenvalues in the right half plane, while the matrix of problem 2 has eigenvalues on both sides of the imaginary axis. Moreover, the symmetric part of $A$ is indefinite for all problems. These equations are also used in [2, 9], where a version of SBRPK based on a two-blocks partition is tested and compared with other methods.

We solve the least squares problems arising in the SBRPK method by using the Cholesky factorization on the normal equations. This choice follows from many considerations. First of all, since the least squares problems have to be solved many times, with different right-hand sides, one can perform the factorization once and use it for all successive iterations. Moreover, for the test problems used the matrix $CC^T$, for all the subblocks $C_{t,j}$, is pentadiagonal. Its Cholesky factor consists of 3 diagonals; hence, the Cholesky factors for all the subblocks can be stored using only 3 additional vectors of lenght $d^2$. The only drawbacks related to his approach are those usually associated with the normal equations. However, it can be show that the condition number of $C$ can never be worse than that of $A$. In particular, for all our test problems the subprolems are well-conditioned, as described in [2]. Therefore the choice of solving the normal equations is a reasonable one. We use the *DPBTRF* and *DPBTRS* routines from *Lapack*.

In all experiments we stop the iterations whenever

$$||r^{(k)}|| \ < \ 10^{-6} \ ||r^{(0)}||,$$

where $||r^{(k)}|| = ||b - Ax^{(k)}||$ is the residual of the original system. Moreover, we use $k = 3$ for all three

CG-like methods to match the three vectors of storage needed by the SBRPK method. For the preconditioners ILU($s, \alpha$), MILU($s, \alpha$), and SSOR($\alpha$) we consider $s = 0$. The parameter $s$ represents the fill-in level in the incomplete factorization. The parameter $\alpha$ governs the quantity added to the main diagonal elements during the factorization, and the relaxation parameter for the SSOR preconditioner. We use $\alpha = 0$ in the first case and 1 in the latter.

Finally the NSPCG package also allows to monitor the ratio $\|r^{(k)}\|/\|r^{(0)}\|$, which will be denoted as RES in the following, and the runs are automatically stopped if the decrease of this value is less than $10^{-5}$ after a prefixed number of iterations. This condition is labeled as a *RS* (residual stall) failure.

We carried out several numerical experiences on a *HP 900 series 700* workstation using *Fortran 77*. Here, just a summary of experiments for each test problem with $d = 36$ is shown in Tables 1-3. An upper limit of 1000 iterations is imposed on each run. The last column in each table is marked with an error code standing for a failure. If the preconditioner cannot be formed, an error code of *UP* (unstable preconditioner) is used. Experiments that exceed the maximum number of iterations are marked with *MI*. Finally, if RES increases very quickly for many iterations the experiment is stopped and marked with the label *DIV*.

The results obtained show that SBRPK succeedes in all cases, while for the test problem 2 every combination of method and preconditioner fails to converge. This confirms the fact that the SBRPK method has a robustness unmatched by the other solvers.

The unpreconditioned NSPCG methods converge only for the problem 3, but are slower than SBRPK and need more iterations. The preconditioned method, however, can outperform the SBRPK method when they work. This arises for test problem 1 where the three CG-like methods with the MILU preconditioner require only one iteration to converge. This shows that for such a problem MILU provides a highly accurate approximation of the matrix $A^{-1}$ so that only one iteration is needed to refine the solution. ILU and SSOR preconditioned methods never succeed, either due to a stalling residual or to unstability in forming the preconditioner.

We remark that, although a default value of $\alpha$ is used in our experiments, the NSPCG package allows the user to find suitable values that prevent the factorization failures. Moreover, if some knowledge of the system being solved is available, it can be more efficient to use adaptive procedures for finding an optimal acceleration parameter for the SSOR preconditioner.

Furthemore, the value of $k$ that we used may be too small, mainly for GMR(k). However, in [2, 9] it has been proved experimentally that increasing $k$ up to 20 does not greatly improve the performance of GMR(k) on these test problems.

In summary, the tests presented point out the following three considerations. The SBRPK method is more reliable than the other solvers, is generally faster than the unpreconditioned methods, and is slower than the preconditioned methods when they succeed.

| **Test problem 1** | | | | | |
|---|---|---|---|---|---|
| METH | PREC | ITER | TIME(s) | RES | FAIL |
| SBRPK | - | 221 | 1.55 | .97E-6 | |
| GCR(3) | - | 1000 | 1.07 | .90E-3 | *MI* |
| | ILU | | | | *DIV* |
| | MILU | 1 | .01 | .42E-6 | |
| | SSOR | | | | *UP* |
| GMR(3) | - | 914 | 1.10 | .99E-3 | *RS* |
| | ILU | | | | *DIV* |
| | MILU | 1 | .01 | .42E-6 | |
| | SSOR | | | | *UP* |
| OMN(3) | - | 1000 | 1.60 | .12E-4 | *MI* |
| | ILU | | | | *DIV* |
| | MILU | 1 | .02 | .42E-6 | |
| | SSOR | | | | *UP* |

Table 1

| **Test problem 2** | | | | | |
|---|---|---|---|---|---|
| METH | PREC | ITER | TIME(s) | RES | FAIL |
| SBRPK | - | 234 | 1.66 | .94E-6 | |
| GCR(3) | - | 1000 | 1.07 | .26E+0 | *MI* |
| | ILU | 1000 | 1.99 | 26E-5 | *MI* |
| | MILU | 661 | 1.45 | .76E-1 | *RS* |
| | SSOR | 1000 | 2.19 | .72E-1 | *MI* |
| GMR(3) | - | 200 | .25 | .29E+0 | *RS* |
| | ILU | 1000 | 2.06 | .75E-5 | *MI* |
| | MILU | 50 | .14 | .43E-1 | *RS* |
| | SSOR | 86 | .26 | .75E-1 | *RS* |
| OMN(3) | - | 44 | .16 | .31E+0 | *RS* |
| | ILU | 77 | .23 | .12E+0 | *RS* |
| | MILU | 220 | .60 | .14E+0 | *RS* |
| | SSOR | 47 | .12 | .95E-1 | *RS* |

Table 2

| Test problem 3 | | | | | |
|---|---|---|---|---|---|
| METH | PREC | ITER | TIME(s) | RES | FAIL |
| SBRPK | - | 96 | .74 | .95E-6 | |
| GCR(3) | - | 607 | .70 | .95E-6 | |
| | ILU | 1000 | 2.01 | .31E-2 | *MI* |
| | MILU | 1000 | 2.22 | .23E-2 | *RS* |
| | SSOR | | | | *UP* |
| GMR(3) | - | 596 | .72 | .99E-6 | |
| | ILU | 47 | .12 | .52E-2 | *RS* |
| | MILU | 1000 | 2.32 | .22E-3 | *MI* |
| | SSOR | | | | *UP* |
| OMN(3) | - | 558 | .99 | .99E-6 | |
| | ILU | 47 | .17 | .11E-1 | *RS* |
| | MILU | 1000 | 2.54 | .67E-2 | *MI* |
| | SSOR | | | | *UP* |

Table 3

## 3 The parallel algorithm

### 3.1 Parallel developing environment

In designing the parallel version of the SBRPK method, we assume a distributed memory message passing computing environment consisting of $P$ nodes logically organized as a ring and numbered from 0 to $P-1$. Each node is formed by a CPU and a local memory. Moreover, a communication network among the nodes allows each of them to perform both broadcast and one-to-one send/receive operations. We develop our algorithm making use of a common concept for this computing environment [7]; that is, the concurrent algorithm is a set of asyncronous processes, performing the same task on different data and syncronizing their activities by communicating messages. In particular, we assume that there is a one-to-one correspondence between processes and nodes.

### 3.2 Data distribution strategy

As we have already pointed out in Section 2, the basic idea to obtain a parallel implementation of the SBRPK method is to distribute the subblocks $C_{t,j}$ over the $P$ processes so that each of them solves one or more subproblems of the form

$$w = \min_z \|u - C^T z\|_2$$

corresponding to every one of the $2L-1$ projections related to a single iteration:

$$y^i = (I - P_i)y^{i-1} \quad i = 1, \dots, L$$

$$y^j = (I - P_j)y^{j+1} \quad j = L-1, \dots, 1$$

Hence, for a fixed projection $P_t$, each node computes a part of the vector $y^t$. Once this concurrent computation of the generic projection is terminated, there must be a communication phase in which the parts of the vector $y^t$ computed by all nodes are suitably exchanged in order to begin the computation of the next projection.
Starting from this idea, our aim is to select a data distribution strategy in order to

- have a good load balancing
- reduce as much as possible the amount of information that each process needs to exchange with any other processes.

With this two objectives in mind, we consider the following strategy. Suppose $d = 12$ and $P = 4$. Then, distribute the row blocks of the matrix $A$ as follows:

$$C_{1,i},\ C_{2,i},\ C_{3,i} \longrightarrow proc.\, i-1\ , \quad i = 1, \dots, 4.$$

This is one of the most natural way to distribute data among processes. In our case, the row blocks of $A$ are logically grouped in 4 groups of 3 contiguous row blocks and then each group is allocated to a particular process. Moreover, the use of this data distribution implicitly leads to the three-blocks partition. The collection of all subblocks assigned to a process forms a local matrix partitioned in 3 blocks. Obviously, each local block consists of only one subblock. The described subblock distribution can be represented as a map from a global index set to a local one in each process:

$$\lambda(t,j) = (p := j-1, \hat{t} := t, s := 1)$$

The function $\lambda$ maps the global subblock index pair $(t,j)$ into three indices $(p, \hat{t}, s)$, where $p$ is the process holding the block $C_{t,j}$, and $(\hat{t}, s)$ is the local index pair of the subblock into the process $p$.
We now focus our attention on the computation of $y = (I-Q)z$ at the generic iteration. The Figure 1 shows the parts of vectors $y^i$ computed by each process and the related communication scheme. The first computational step consists of computing the $d^2$-dimensional vector $y^1 = (I - P_1)y^0$. According to the described data distribution, the process 0 computes the first $2*d$ components of $y^1$, the process 1

the next $3 * d$ and so on for the other processes. After this concurrent computation, the process $i$ sends the first $d$ components of its local vector to process $i-1$, i=2,3,4. This communication phase can be realized efficiently in two steps, in each of them pairs of processes communicate in parallel. For the example under consideration, at the first step we have the communicating pairs $(0 \leftarrow 1)$, $(2 \leftarrow 3)$, while in the second step we have the communicating pair $(1 \leftarrow 2)$. For the other projections, we have an analogous computational and communication scheme. The only variation arises when the projections go back through the blocks. In this case, the process $i$ sends a suitable group of $d$ components of its vector to process $i+1$, $i = 1, 2, 3$. The advantages of this data distribution strategy are:

- each process has to communicate only with two processes, which are the same for each projection;
- the vector to be sent/received has lenght $d$.

Moreover, this strategy can be easily generalized. Let $NS$ the number of subblocks in each block $A_t$ and suppose that $LNS = NS/P$ is an integer. Then, for each block $A_t$, the first $LNS$ subblocks are assigned to process 0, the second $LNS$ subblocks to process 1 and so on. The related distribution function is:

$$\lambda(t,j) = (p := \frac{j-1}{LNS}, \hat{t} := i, s := j - (p \cdot LNS)) .$$

The use of this strategy leads to a communication scheme having the same features as that for $d = 12$ and $P = 4$. Therefore, since the communication is independent of the size of problem and the number of processes used, the main goal to obtain a scalable parallel algorithm is achieved.

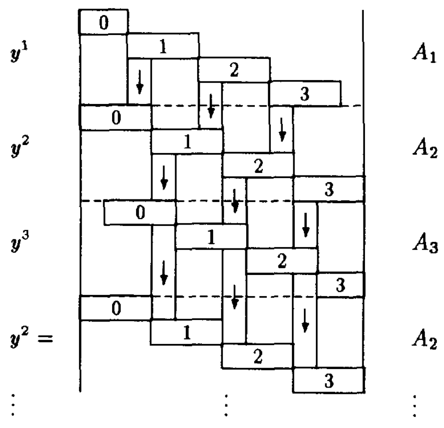

Fig. 1
Communication scheme for $d = 12$ and $P = 4$.

## 3.3 Algorithm framework

Starting from the above discussion, we now illustrate in detail the concurrent algorithm framework for the generic process $p$. In particular we focus our attention on a single step B of the SBRPK method. As in previuos section, we assume that $LNS = NS/P$ is an integer; that is, each process has the same number of subblocks in each local block $A_t$. Moreover, we assume that the least squares problems are solved by the Cholesky algorithm on the normal equation and we denote with $P_{t,s}$ the product $C_{t,s}^T(R_{t,s}R_{t,s}^T)^{-1}C_{t,s}$ where $R_{t,s}$ is the Cholesky factor of the product $C_{t,s}C_{t,s}^T$. We also assume that each process has already computed the Cholesky factors of its local subblocks. To describe the communication in the concurrent algorithm we make use of the following common notations:

- *send(buf,proc)*
  the message stored in *buf* is sent to the process *proc*
- *receive(buf,proc)*
  a message is received from process *proc* and stored in *buf*
- *globalsum(buf,nproc)*
  data stored in buffer are individually added across *nproc* processes. The final sum will overwrite *buf* in each process.

The concurrent algorithm is displayed in Figure 2. The index set $\mathcal{S}_t$ keeps track of the subblocks in the local block $A_t$. The arrays $y^{old}$ and $y^{new}$ are the vector $y$ to be computed and the vector computed to the previous iteration, respectively. The array $x$ stores the vector solution. Finally the array *res* represents the pseudo-residual at the current iteration. The algorithm starts with the computation of $y = (I - Q)v$. This is done by two iterative cycles, the first one driving the forward sweep through the blocks, the other one the back sweep. Communication occurring between two projections is realized by a pair of send/receive operations according to the described communication scheme. The second phase of the algorithm consists of updating the vectors $y$, $x$ and the residual. We observe the presence of two others communication operations. More precisely, since each process holds a part of this vectors, it can computes a partial sum of the scalar products $\alpha$ and *ron*. However, these products are to be held entirely by all processes in order to update the mentioned vectors. Then it is necessary to perform a global sum of these quantities over all processes. Moreover, we point out that each process independently computes a different part of the vector solution $x$.

```
$S_t := \{s : 1 \le s \le LNS\}, \quad 1 \le t \le L$ ;

for $t = 1, L$ do begin
  $init := (i-1)d$ ; $start := init$
  for all $s \in S$ do begin
    $start := start + 3(s-1)d$ ; $end := start + 3d$
    $y^{new}[start+1 : end] := (I - P_{t,s})y^{new}[start+1 : end]$
  endfor
  if $t < L$ then
    send($y^{new}[init+1 : init+d], p-1$)
    receive($y^{new}[end+1 : end+d], p+1$)
  else
    receive($y^{new}[init-d+1 : init], p-1$)
    send($y^{new}[end-d+1 : end], p+1$)
  endif
endfor
for $t = L-1, 1, -1$ do begin
  $init := (i-1)d$ ; $start := init$
  for all $s \in S$ do begin
    $start := start + 3(s-1)d$ ; $end := start + 3d$
    $y^{new}[start+1 : end] := (I - P_{t,s})y^{new}[start+1 : end]$
  endfor
  if $t > 1$ then
    receive($y^{new}[init-d+1 : init], p-1$)
    send($y^{new}[end-d+1 : end], p+1$)
  endif
endfor
$i1 = 1$ ; $i2 = LNS * 3d$
$y^{new}[i1 : i2] := y^{old}[i1 : i2] - y^{new}[i1 : i2]$
$\alpha := (y^{new}[i1 : i2], y^{old}[i1 : i2])$
globalsum($\alpha, P$)
$\alpha := ro/\alpha$
$x[i1 : i2] := x[i1 : i2] + \alpha * y^{old}[i1 : i2]$
$res[i1 : i2] := res[i1 : i2] - \alpha * y^{new}[i1 : i2]$
$ron := (res[i1 : i2], res[i1 : i2])$
globalsum($ron, P$)
$\beta := ron/ro$
$y^{new}[i1 : i2] := res[i1 : i2] + \beta * y^{old}[i1 : i2]$
$y^{old}[i1 : i2] := y^{new}[i1 : i2]$
$ro := ron$
```

Fig. 2
Concurrent SBRPK algorithm framework

## 4 Parallel efficiency evaluation

### 4.1 Testing environment

To verify the parallel efficiency of our concurrent algorithm we tested it on two MIMD distributed memory machines: an *Intel iPSC/860* at the Department of Mathematics and Applications of the University of Naples and the *Intel Touchstone DELTA System* at the California Institute of Technology (*Caltech*).
The Intel iPSC/860 [13] is an hypercube multiprocessors with 16 nodes based on the i860 microprocessor . Each node has 16 Mbytes of local memory and is rated to a peak performance of 60 Mflops in double precision arithmetic. A direct-connect communication system provides the data pathway between all the nodes of the machine.
The Intel Touchstone DELTA System is a distributed memory machine with 512 computing nodes based on the i860 microprocessor and connected by a 2D-mesh. Each node has the same features as the Intel iPSC/860 nodes.
The parallel SBRPK algorithm is implemented in Fortran 77 using the Intel *NX* communication system which provides all the basic facilities to run parallel applications [13]. We use the SPMD parallel programming model in which a single instance of the same program runs on each node. To get timing we use the *second()* routine of the Fortran library.

### 4.2 Experimental Results

The results we show refer to tests performed on the block tridiagonal matrix arising from the test problem 1, with $d$ ranging from 48 to 288. To evaluate the parallel performance of the algorithm we use the following classical parameter

$$Efficiency \quad : \qquad E_p(d) = \frac{T_1(d)}{pT_p(d)}$$

where $T_i(d)$ is the elapsed execution time on $i$ nodes for a $d$ size problem.
The efficiency values realized on the Intel iPSC/860, with $P = 4, 8$ and on the DELTA System, with $P = 8, 16, 32$, for $d = 48, 96, 144, 192, 240, 288$, are plotted in Figure 3 and 4 respectively. We observe that this values never drop below 0.5. Moreover, high efficiency values (greater than 0.8) are obtained for $d \ge 192$ whatever number of nodes used. The only exception is represented by the $P = 32$ case. We observe an efficiency drop for $d = 244$. This is due to the fact that for $d = 244$ the number of subblocks

in each $A_t$ is not a multiple of the number of nodes; consequently, the data distribution in our algorithm doesn't allow to obtain a good load balancing while performing all the projections. However, the efficiency value obtained can be still considered a good one and we can state that our data distribution strategy allows a good parallel efficiency even if the number of nodes increases for medium size problem.
For a better comprehension of how the algorithm can reach a good efficiency, we also report in Table 4 the values of the total execution time and of the communication time, in seconds, on the Delta System for the selected problem size $d$. Looking at the $T_{com}$ columns in the table, we note that the communication time increases very slowly when the problem size grows. An analogous behavior may be observed for the communication time for a fixed problem size variyng the number of nodes; that is, this values are almost the same. We again remark that in the $P = 32$ case this is no longer true only for $d = 144, 240$ for the reasons previously described. In summary, the results obtained shows that the communication overhead is independent of both the problem size and the number of nodes used, confirming our theoretically consideration about the scalability of the algorithm.

## 4.3 Conclusions and perspectives

In this paper we presented a parallel iterative algorithm for the solution of large sparse nonsymmetric systems of linear equations. The algorithm is based on a row-projection method deriving from the block Kaczmarz procedure. This method transforms a nonsymmetric linear system with arbitrary eingenvalues distribution into a symmetric one with eigenvalues restricted to $[0, 1)$.
Numerical experiments on block tridiagonal systems showed that the SBRPK method has a robustness unmatched by other iterative nonsymmetric solvers, according with the results presented in other papers related to this method. Moreover, a simple permutation of the rows of the matrix made the SBRPK method suitable for implementation on multiprocessors. In particular, we proposed a parallel version of the method for MIMD distributed memory machines. The results obtained by implementing the parallel algorithm on two of these machines showed that a suitable data distribution strategy leads to a communication overhead that doesn't depend on the number of nodes used. This, in turn, allows to obtain good performances.

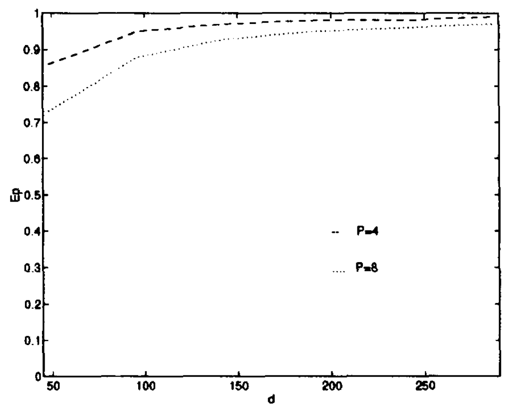

Fig. 3
Efficiency values of the concurrent SBRPK algorithm on the Intel iPSC/860

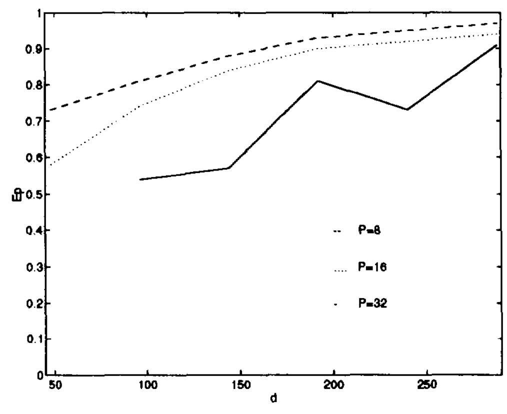

Fig. 4
Efficiency values of the concurrent SBRPK algorithm on the Intel Touchstone Delta System

| | number of nodes | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 8 | | | 16 | | |
| $d$ | $T_{tot}$ | $T_{tot}$ | $T_{com}$ | $E_p$ | $T_{tot}$ | $T_{com}$ | $E_p$ |
| 48 | 21 | 3.7 | 0.9 | 0.73 | 2.3 | 0.9 | 0.58 |
| 96 | 77 | 12 | 1.1 | 0.81 | 6.5 | 1.0 | 0.74 |
| 144 | 178 | 25 | 1.3 | 0.88 | 13 | 1.2 | 0.84 |
| 192 | 323 | 43 | 1.5 | 0.93 | 22 | 1.5 | 0.90 |
| 240 | 515 | 67 | 1.8 | 0.95 | 35 | 1.6 | 0.92 |
| 288 | 747 | 96 | 1.8 | 0.97 | 49 | 1.9 | 0.94 |

Table 4
Efficiency, total execution time and communication time values of the concurrent SBRPK algorithm on the Intel Touchstone Delta System for $P = 1, 8, 16$

|  | number of nodes |  |  |  |
|---|---|---|---|---|
|  | 1 | 32 |  |  |
| $d$ | $T_{tot}$ | $T_{tot}$ | $T_{com}$ | $E_p$ |
| 48 | 21 | - | - | - |
| 96 | 77 | 4.4 | 1.5 | 0.54 |
| 144 | 178 | 9.7 | 5.5 | 0.57 |
| 192 | 323 | 12 | 1.7 | 0.81 |
| 240 | 515 | 22 | 8.2 | 0.73 |
| 288 | 747 | 26 | 2.0 | 0.91 |

Table 5
Efficiency, total execution time and communication time values of the concurrent SBRPK algorithm on the Intel Touchstone Delta System for $P = 1, 32$

The main issue related to the data distribution used is a poor load balancing when the number of subblocks in each larger blocks is not a multiple of the number of nodes. Since our goal is to realize an efficient black-box parallel row-projection software, future work needs to develop suitable strategies for a better load balancing in the case previously mentioned.
Finally, the SBRPK method can be efficiently applied to any system that can be reordered into a banded system. Hence, it may be interesting to study row partitionings for general sparse systems leading to matrix suitable for the row projection method.

## Acknowledgements

This work was supported by the CNR (Special Project "Informatic Systems and Parallel Computing") under grant n. 93.01615.PF69 and it was performed in part using the Intel Touchstone DELTA System operated by Caltech on behalf of Concurrent Supercomputing Consortium. Access to this facility has provided by Caltech.

## References

[1] R. Bramley, A. Sameh, " Row Projection Methods for Large Nonsymmetric Linear Systems", *SIAM J. Sci. Stat. Comput.*, Vol. 13 (1), pp. 168-193, 1992.

[2] R. Bramley, A. Sameh, "A Robust Parallel Solver for Block Tridiagonal Systems",, *CSRD, Univ. Illinois - Urbana*, Tech. Rep. 806, 1988.

[3] Y. Censor, "Parallel Application of Block-Iterative Methods in Medical Imaging and Radiation Therapy", *Math. Programming*, Vol. 42, pp.307-325, 1988.

[4] M. D'Apuzzo, *Calcolo Parallelo: Metodi Row-Action per la Risoluzione di Sistemi di Equazioni Lineari*, Ph.D. thesis, University of Naples, 1991.

[5] P.P.B Eggermont, G.T. Herman, A. Lent, "Iterative Algorithms for Large Partitioned Linear System, with Applications to Image Reconstruction", *Linear Algebra and its Applications*, Vol. 40, pp.37-67, 1981.

[6] T. Elfving, "Block-Iterative Methods for Consistent and Inconsistent Linear Equations", *Numer. Math.*, Vol. 35, pp.1-12, 1980.

[7] G.C. Fox et al., *Solving Problems on Concurrent Processors*, Prentice-Hall, 1988.

[8] S. Kaczmarz, "Angenäherte Auflösing von Systemen Linearer Gleichungen", *Bull. Internat. Acad. Polon. Sci. Cl. A*, pp.335-357, 1937.

[9] C. Kamath, A. Sameh, "A projection Method for Solving Nonsymmetric Linear Systems on Multiprocessors", *Parallel Computing*, Vol. 9, pp. 291-312, 1988/89.

[10] T.C. Oppe, W.D. Joubert, and D.R. Kincaid, "NSPCG User's Guide, version 1.0: A package for Solving Large Linear Systems by Various Iterative Methods", *Center for Numerical Analysis, University of Texas at Austin, Austin, TX*, Tech. Rep. CNA 216, 1988.

[11] Y. Saad, M. Schultz, "Conjugate Gradient-Like Algorithms for Solving Nonsymmetric Linear Systems" *Mathematics of Computations*, Vol. 44, pp. 417-424, 1985.

[12] Y. Saad, M. Schultz, "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems", *SIAM J. Sci. Statist. Comput.*, Vol. 7, pp. 856-869, 1986.

[13] Intel Corporation, *iPSC/860 Manuals*, Intel, 1991.