Parallel Krylov Subspace Basis Computation[*]

Roger B. SIDJE[†] AND Bernard PHILIPPE[†]

February 1994

Abstract

Numerical methods related on Krylov subspaces are widely used in large sparse numerical linear algebra. Vectors in these subspaces are manipulated through their representation onto orthonormal bases. Nowadays, on serial computers, the method of Arnoldi is considered as a reliable technique for constructing such bases. Unfortunately, this technique is rather inflexible to be efficiently implemented on parallel computers. In this report we examine several parallel and stable algorithms based on the idea of Reichel *et al.* [1, 2] which retrieve at their completion the same information as the sequential Arnoldi's method. We present timing results obtained from their implementations on the Intel Paragon distributed-memory multiprocessor machine.

Key words. Krylov subspaces, Leja points, parallel QR factorization, Intel Paragon.

AMS subject classifications. 65F50, 65F25, 65W05.

1 Introduction

Background and motivations. The question of obtaining with a stable procedure an orthonormal basis

$$V_m = [v_1, v_2, ..., v_m] \tag{1}$$

of the Krylov subspace

$$\mathbf{K}_m = \text{Span}\{v, Av, ..., A^{m-1}v\} \tag{2}$$

where $A \in \mathbb{R}^{N \times N}$ is sparse and $v \in \mathbb{R}^N$ (N is large and $m \ll N$) arises as a crucial problem in projection methods for approximating:

- solutions of linear systems for which the coefficient matrix is A (`FOM` and `GMRES` like procedures).
- eigenvalues and eigenvectors of A (Arnoldi procedure).
- functions of the matrix A [6, 9].

In presence of a multiprocessor environment, it becomes hardly difficult to efficiently implement the Arnoldi's algorithm which is the standard sequential tool for handling the problem.

Arnoldi basis. The method of Arnoldi is a stable procedure which is highly valued for the computation of V_m. It results from a recursive application of the Modified Gram-Schmidt (MGS) process on the columns of $[v_1, Av_1, ..., Av_{m-1}]$ and upon its completion, we have the following fundamental relation

$$AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^T \tag{3}$$

[*]This work is supported by a NSF-INRIA agreement.

[†]INRIA-IRISA, Campus de Beaulieu, Rennes, France (sidje@irisa.fr, philippe@irisa.fr).

or more condensely

$$AV_m = V_{m+1}\overline{H} \qquad (4)$$

where $V_j = [v_1, ..., v_j] \in \mathbb{R}^{N\times j}$ is an orthonormal basis of $\mathbf{K}_j$, $\overline{H} \in \mathbb{R}^{(m+1)\times m}$ is such that $H_m = \overline{H}(1:m, 1:m) = V_m^T A V_m$ is upper Hessenberg and $\overline{H}(m+1, 1:m) = (0, ..., 0, h_{m+1,m})^T$.

Unfortunately the underlying parallelism within this algorithm is limited only to matrix-vectors multiplies, dot-products and saxpys. For this reason, in the aim of increasing the parallelism, some authors prefer to use Classical Gram-Schmidt (CGS) on supercomputers. But since this version can sometimes be unstable, it is more cautious to apply twice over the orthogonalization process which now, causes a degradation of performances.

Scaled monomial basis. With $\sigma_0, ..., \sigma_{m-1}$ being scaling coefficients, we have

$$\mathbf{K}_m = \mathrm{Span}\{\sigma_0 v, \sigma_1 Av, ..., \sigma_{m-1}A^{m-1}v\}. \qquad (5)$$

Some authors proposed to adopt this representation and to carry out the QR decomposition of the columns appearing in (5) via Householder reflectors. Unfortunately this approach is inappropriated because, it is subject to severe rounding errors.

Newton basis at Leja points. Another approach recently introduced and, which seems to be more promising, comes from the relation

$$\mathbf{K}_m = \mathrm{Span}\{\sigma_0 v, \sigma_1(A - \lambda_1 I)v, ..., \sigma_{m-1}\prod_{\ell=1}^{m-1}(A - \lambda_\ell I)v\} \qquad (6)$$

where the stability can be significantly improved by a suitable choice of the $\sigma_0, ..., \sigma_{m-1}$ and the $\lambda_1, ..., \lambda_{m-1}$. The studies of Reichel *et al.* [1, 2, 8] show that a suitable set $\{\lambda_\ell\}_{\ell=1}^{m-1}$ which improved the conditioning of (6) are the $(m-1)$-st *Leja points* in the convex hull that encloses the spectrum of A. In particular the $\{\lambda_\ell\}_{\ell=1}^{m-1}$ can be the $(m-1)$-st eigenvalues of A numbered according to *Leja order*. Since in practice the spectrum of A is unknown, the considered eigenvalues are those of the upper Hessenberg matrix H_m obtained from a first step by Arnoldi algorithm. Moreover if the technique is used for eigenproblems, as soon as the Ritz values are obtained, they may be used as the new updated set of Leja points. But in order to avoid complex arithmetic, the Leja ordering must be somewhat modified. All the details are described in [1, 2].

Instead of focusing on a specific-purpose procedure that solves a particular problem as it is done in [1, 2] for the case of `GMRES`, we would like to develop here ready-to-use high-level parallel algorithms that retrieve the same information as the Arnoldi's method, i.e. the basis V_{m+1} and the matrix $\overline{H}$. Those information should be explicitly or implicitly given. This way of doing allows to properly design into parallel environments the wide family of scientific applications that make use of Krylov subspaces. Right now, it is worth mentioning that the problem of computing Krylov bases for the same matrix A and different starting vectors v's arises several times within an application but the approximated Leja points are computed once and we shall now admit that the set $\{\lambda_\ell\}_{\ell=1}^{m}$ is known.

Defining the $N \times j$ matrix[1]

$$\widehat{A}_j \equiv \left[\sigma_0 v, \sigma_1(A - \lambda_1 I)v, ..., \sigma_{j-1}\prod_{\ell=1}^{j-1}(A - \lambda_\ell I)v\right], \qquad (7)$$

[1]We adopt here notations which are sligthly different to those in [1, 2]: B_j is denoted by $\widehat{A}_j$ and $\widetilde{B}_j$ by $\widetilde{A}_j$.

we set v as the starting vector in [1, Algorithm 2.1] and we obtain real matrices $\widetilde{A}_j \in \mathbb{R}^{N\times j}$ and $\widetilde{D}_j = \mathrm{diag}(\tilde{d}_1, ..., \tilde{d}_j)$ satisfying

$$\widehat{A}_j = \widetilde{A}_j \widetilde{D}_j. \tag{8}$$

Therefore it turns out that if the QR decomposition of $\widetilde{A}_j$ is computed

$$\widetilde{A}_j = \widetilde{Q}_j \widetilde{R}_j \tag{9}$$

then, the desired orthonormal basis of $\mathbf{K}_j$ is simply

$$V_j = \widetilde{Q}_j. \tag{10}$$

How to recover $\overline{H}$? To be a general-purpose approach, we would like to end up with the unifying relation (4), i.e., to determine $\overline{H}$. It was shown in [1] that, going from the QR decomposition of $\widetilde{A}_{m+1} = \widetilde{Q}_{m+1}\widetilde{R}_{m+1}$, one can construct almost straightforwardly an upper Hessenberg matrix $\widehat{H} \in \mathbb{R}^{(m+1)\times m}$ such that

$$A\widehat{A}_m = \widetilde{Q}_{m+1}\widehat{H}. \tag{11}$$

Therefore we can state that

$$\overline{H} = \widehat{H}\left(\widetilde{R}_m \widetilde{D}_m\right)^{-1}. \tag{12}$$

Indeed by using (8–11) it comes,

$$V_{m+1}\widehat{H} = A\hat{A}_m = A\widehat{A}_m\widehat{D}_m = AV_m\widehat{R}_m\widehat{D}_m.$$

Step (12) is extremely sensitive and it explains why the set $\{\lambda_\ell\}_{\ell=1}^m$ must be chosen in a way such that the condition number of $\widehat{A}_m$ remains of reasonable magnitude. In the particular case of the `GMRES` implementation, $\overline{H}$ need not be explicitly form. Nonetheless in many other areas, $\overline{H}$ should be known. But undoubtely, the most consuming task of the entire computation is the QR decomposition of $\widetilde{A}_{m+1}$.

2 Parallelization

In fact the parallel implementation of the approach requires to parallelize:

(a) the operation $\sigma(A - \lambda I)v$ where $A \in \mathbb{R}^{N\times N}$ is *sparse* which involves a large amount of flops when N is very large.

(b) the QR factorization of $\widetilde{A}_{m+1}$ where $\widetilde{A}_{m+1} \in \mathbb{R}^{N\times(m+1)}$ is *dense* and $m \ll N$.

The point (a) is almost the same as the sparse matrix vector multiplication problem. We shall restrict our study in this report to point (b) and we shall describe parallel algorithms intended to distributed-memory multiprocessor supercomputers. We assume that the interconnexion processor network is physically or logically a bi-directional ring.

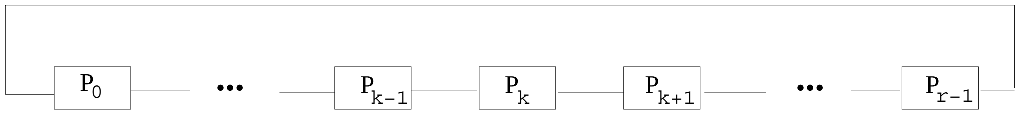


Figure 1: The interconnexion network is a bi-directional ring

Processors are numbered from 0 to $r-1$. We use three basic functions: `myid()` returns the number of the processor that issues the command. `myright()` and `myleft()` return the numbers of the right and the left neighbors respectively. We also make use of the fundamental message-communication functions `send()` and `receive()`. At this present level, there are important considerations to highlight.

- At the end of the algorithms, the factor $\widetilde{Q}_{m+1}$ which constitutes the basis we are looking for may not be explictly known and/or may be scattered across the network. Therefore we shall supply (parallel) procedures that compute the matrix vector product $z \leftarrow \widetilde{Q}_{m+1}y$ for any given vector $y \in \mathbb{R}^{m+1}$. This point constitutes the major difference with usual parallel least squares solvers in which a product of the form Q^Tb is perform while doing the factorization (see, e.g., [5, 10, 12] and references therein). Moreover it is unwise to form explictly Q systematically because the actual obtained Krylov basis is used once (i.e. there is only *one* product Qy) during the overall running of the application.
- The dimension m of the Krylov subspace is such that $N \gg m$ and even $\frac{N}{r} \gg m$.
- Although the algorithms are mathematically equivalent to Arnoldi's method, their computed values are not rigorously identical because their stability behaviors are different.

For the ease of the presentation, we shall denote $\widetilde{A}_{m+1}$ by A_m but it should be clear that we are really looking for the QR factorization of $\widetilde{A}_{m+1}$.

3 Parallel Algorithms of Reference

3.1 Modified Gram-Schmidt

```
ALGORITHM 3.1.1: MGSDEC
initialize $\left(N, m, N_{\mathrm{loc}}, A_{\mathrm{loc}}(1:N_{\mathrm{loc}},:)\right)$ ;
for $j := 1:m$ do
    for $i := 1:j-1$ do
        $s := A_{\mathrm{loc}}(:,i)^T A_{\mathrm{loc}}(:,j)$ ;
        $s :=$ GLOBALSUM($s$) ;
        $A_{\mathrm{loc}}(:,j) := A_{\mathrm{loc}}(:,j) - s * A_{\mathrm{loc}}(:,i)$ ;
        $R_{\mathrm{loc}}(i,j) := s$ ;
    endfor
    $s := A_{\mathrm{loc}}(:,j)^T A_{\mathrm{loc}}(:,j)$ ;
    $s :=$ GLOBALSUM($s$) ;   $s := \sqrt{s}$ ;
    $A_{\mathrm{loc}}(:,j) := s * A_{\mathrm{loc}}(:,j)$ ;
    $R_{\mathrm{loc}}(j,j) := s$ ;
endfor
```

```
ALGORITHM 3.1.2: MGSVEC
$z_{\mathrm{loc}}(1:N_{\mathrm{loc}}) := A_{\mathrm{loc}}(1:N_{\mathrm{loc}}, 1:m) * y$ ;
```

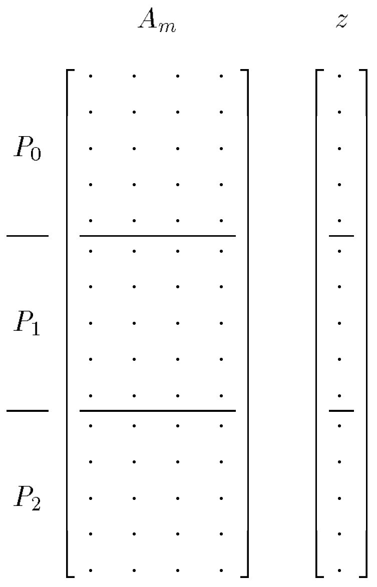


Figure 2: Distribution by contiguous rows.

Data distribution. Remember that r is the number of processors, the matrix A_m is partitionned into r portions of contiguous rows. If the euclidian division gives $N = \overline{N}r + \bar{r}$ then, in the kth processor ($0 \leq k \leq r-1$),

$$N_{\text{loc}} \leftarrow \begin{cases} \overline{N}+1 & \text{if } k \leq \bar{r}-1 \\ \overline{N} & \text{if } k > \bar{r}-1 \end{cases} \tag{13}$$

$$A_{\text{loc}}(1:N_{\text{loc}}, 1:m) \leftarrow \begin{cases} A_m(kN_{\text{loc}}+1:(k+1)N_{\text{loc}}, 1:m) & \text{if } k \leq \bar{r}-1 \\ A_m(\bar{r}+kN_{\text{loc}}+1:\bar{r}+(k+1)N_{\text{loc}}, 1:m) & \text{if } k > \bar{r}-1 \end{cases} \tag{14}$$

This distribution insures that $|N_{\text{loc}}^{(k)} - N_{\text{loc}}^{(k')}| \in \{0,1\}$ for two different processors k and k'. The factor R will be available into all processors upon completion. For `MGSVEC`, the vector y must be present into all processors and the resulting product $z = Qy$ is split according to the row distribution of A_m.

Description.

`MGSDEC` Evolves exactly as the standard sequential MGS. The jth column must be orthogonalized against the previous ones: $a_j \leftarrow a_j - (a_i^T a_j)a_i$, $i = 1, ..., j-1$. Since each column is split throughout the network, the dot-products are partial thus, before carrying saxpy, any processor has to gather the contributions from all the others. This is done by means of the system-supply function `GLOBALSUM()`. The same reasoning apply for the normalizing step.

`MGSVEC` At completion the factor Q is explicitly available but is shared in the same fashion that the original matrix A_m at the beginning. Hence the product $z = Qy$ is trivial.

Parallelism. The regularity and the simplicity of these algorithms are attractive compared to those presented in the following sections. But when the number of processors increases, one can rightly ask if multiple calls to `GLOBALSUM()` will result on a bottleneck slowing down drastically the method. We observe that this is not a crucial point. However this implementation of MGS didn't speed-up and, runs for about 1 to 9 times slowlier than other sophisticated methods.

For areas where the factor Q is explicitly needed, it will constitute the method of choice. It is very easy to implement and becomes quite competitive. This interesting facet is in accordance with what was already stated in [5].

3.2 Column-oriented Householder Decomposition

Data distribution. The matrix $A_m = [a_1, ..., a_m]$ is distributed by columns into a wrap-around fashion. If the euclidian division yields $m = \bar{m}r + \bar{r}$ then, in processor P_k, $A_{\text{loc}} = [a_1^{(k)}, ..., a_{m_{\text{loc}}}^{(k)}]$ with

$$m_{\text{loc}} \leftarrow \begin{cases} \bar{m}+1 & \text{if } k \leq \bar{r}-1 \\ \bar{m} & \text{if } k > \bar{r}-1 \end{cases}, \quad \begin{cases} a_j \in P_k & \Longleftrightarrow \quad k = (j-1)\%r \quad \Longleftrightarrow \quad (j-1-k)\%r = 0 \\ a_j = a_\ell^{(k)} & \Longleftrightarrow \quad j = (\ell-1)r + k + 1 \end{cases}$$

In our present implementation, the factor R will be available into all processors at the end of the decomposition. For the algorithm `COLVEC`, the vector y and the product Qy belong to a specific processor. It is the one in possession of the mth column.

```
Algorithm 3.2.1: COLDEC
initialize $\left(k = \texttt{myid}(), N, m, m_{\mathrm{loc}}, A_{\mathrm{loc}}(1:N, 1:r:(m_{\mathrm{loc}}-1)r+k+1)\right)$ ;
$jright$ := last column-index of myright()
$j := 1$ ;
$\ell := 1$ ;
while $\ell \le m_{\mathrm{loc}}$ do
    if $(j = (\ell - 1) * r + k + 1)$ then $\{a_j = a_\ell^{(k)}$ ........................$\}$
        create the j-th reflector and send it to myright()
        $\ell := \ell + 1$ ;
    else
        receive the j-th reflector from myleft()
        if its creator is not myright() and $(j < jright)$ send it to myright()
    endif
    update my local columns
    $j := j + 1$ ;
endwhile
```

```
Algorithm 3.2.2: COLVEC
$k :=$ myid() ;   $k_m := (m-1)\%r$ ;
$q := k - k_m$ ;   if $(q \le 0)$ $q := q + r$ ;
if $(k = k_m)$ $z := (y, 0, ..., 0)^T$ ;
$j := m$ ;
$\ell := m_{\mathrm{loc}}$ ;
while $(\ell > 0)$ do
    if $((j-1)\%r = k_m)$ then $\{a_j \in P_{k_m}$ .............................$\}$
        reflector := $A_{\mathrm{loc}}(:, \ell)$ ;
        if $(k = k_m)$ then
            apply the reflector on $z$
            $p := 1$ ;
        else
            $p := 0$ ;
        endif
        {re-fit if some processors have finished .......................}
        if $(\ell = 1$ and $k \le \bar{r} - 1)$ $q := q - (r - \bar{r})$ ;
    else
        $p := p + 1$ ;
        if $(k \ne k_m$ and $p \le q)$ send( reflector, myright() ) ;
        if $(k = k_m$ or $p < q)$ receive( reflector, myleft() ) ;
        if $(k = k_m)$ apply the reflector on $z$
    endif
    if $(p = q)$ $\ell := \ell - 1$ ;
    $j := j - 1$ ;
endwhile
```

Description.

`COLDEC` At the jth step, the Householder reflector is constructed by the processor in possession of the jth column. Then, this reflector is sent in transit within the ring in order to enable the other processors to update their columns. The conditional statement
«if its creator is not `myright()` **and** ($j < jright$) send it to `myright()`»
is crucial for a proper termination of the processes.

`COLVEC` The logic behind this algorithm is somewhat arduous but the principle is simple. At the end of `COLDEC` the lower trapezoidal part of A_m is overwritten by Householder vectors (see, e.g., [4, chap. 5]), hence the reflectors are distributed in the ring exactly like the matrix A_m at the beginning. So if k_m denotes the index of the processor in possession of the mth reflector, we assign to P_{k_m} the computation of

$$Q_m y = H_1 \cdots H_m \begin{bmatrix} I_m \\ 0 \end{bmatrix} y. \tag{15}$$

The way the proposed algorithm achieves this is better illustrated throughout an example. We consider a ring of $r = 3$ processors and a matrix of $m = 10$ columns, $A_m = [a_1, ..., a_{10}]$, $\bar{m} = 3$, $\bar{r} = 1$, $k_m = 0$ and $Q_m y = H_1 \cdots H_{10}[y, 0]^T$. We assume that P_{k_m} owns y. At the end of `COLDEC` we have the following distribution:

P_0	P_1	P_2
H_1		
H_4	H_2	H_3
H_7	H_5	H_6
H_{10}	H_8	H_9

The first thing we do is to *roll up* the ring in order to «see» P_{k_m} in the tail end. This is done by means of the (local) variable q which keeps trace of the new position of each processor. We obtain:

$q = 1$	$q = 2$	$q = 3$
P_1	P_2	P_0
		H_1
H_2	H_3	H_4
H_5	H_6	H_7
H_8	H_9	H_{10}

The problem now comes down to a production-line work. We must manage to let the reflectors reach processor P_{k_m} in the correct order. The crucial observation to make at this juncture is that, by unrolling the entire communication pattern, we came across a problem consisting of moving data along the rows of an upper triangular grid. To proceed further, we introduce some notations (Table 1).

command	action
Cj	Copy the jth reflector into the transit-buffer called `reflector` This corresponds to the statement «`reflector` := $A_{\text{loc}}(:,\ell)$»
Sj	Send the transit-buffer to `myright()` The jth reflector is sent
Rj	Receive into the transit-buffer from `myleft()` The jth reflector is received
Hj	Apply the Householder reflector in the transit-buffer on z The jth reflector was there

Table 1: Meaning of commands.

Moreover two commands which are executed one after the other are written on the same line and are separated by a semicolon. For instance S6;R5 means perform in sequence S6 and R5.

head ⟶ tail

ℓ	P_1	ℓ	P_2	ℓ	P_0
				1	H_1
1	H_2	1	H_3	2	H_4
2	H_5	2	H_6	3	H_7
3	H_8	3	H_9	4	H_{10}

	j	p	q: 1	2	3
	10	0	C8	C9	C10; H10
slice 1	9	1	S8	S9, R8	R9; H9
	8	2	•	S8	R8; H8
	7	0	C5	C6	C7; H7
slice 2	6	1	S5	S6; R5	R6; H6
	5	2	•	S5	R5; H5
	4	0	C2	C3	C4; H4
slice 3	3	1	S2	S3; R2	R3; H3
	2	2	•	S2	R2; H2
slice 4	1	0			C1; H1

Figure 3: `COLVEC`. Time-space diagram.

The diagram in Figure 3 represents the entire set of commands involved in our example. One by one, the reflectors are brought to P_{k_m}. The steps can be linked together into slices. Specifically, a slice is the set of steps necessary to bring a reflector from head to tail. Therefore the size of the slice is simply the number of processors. In a slice, for reasons of consistency, a processor becomes idle just after sending the reflector coming from the head. In the algorithm, this is detected when $p = q$ and therefore by decreasing their ℓ, the processors insure that they will copy in their transit-buffer the appropriated reflector for the continuation.

At the jth step, the behavior of a processor other than P_{k_m} can be summarized by: «if the jth reflector belongs to P_{k_m} then I copy my current reflector into my transit-buffer otherwise I send my transit-buffer to right; if there is a reflector coming from left then I receive it.» The behavior of P_{k_m} is particular since it does not send any message but it must apply all the reflectors.

There remains the question of consistent termination. We distinguish two cases. When $\bar{r} \neq 0$, no all processors will be involved in the very last slice so a re-fitting must be done. When $\bar{r} = 0$, no problem occurs.

Parallelism. Although these algorithms may be suitable for some cases [7], we do not achieve interesting speed-ups in our context and even sometimes no speed-up at all. This comes from the fact that, when N is very large and $m \ll N$ there are only a few columns per processor so that the Householder vectors which are in transit across the ring are very long but are not used intensively. Performances are valuable only for moderate N when we increase sufficiently m.

Since $\frac{N}{r} \gg m$, simpler versions based on row partitionning as in `MGSDEC` could have been considered. There, processor P_0 would hold the factor R. The Householder vectors would then be split according to the row distribution of A_m so that their use at each stage, would necessitate global operations involving the contribution of the entire set of processors. However since the QR decomposition via MGS is cheaper than via Householder reflectors, we do not expect to go, with these row oriented versions, as fast as the former presented MGS implementations do.

4 Parallel Hybrid Algorithms

The algorithms we examine here avoid global operations and relies upon two basic tasks:

- Elimination through Householder elementary transformations
- Elimination through Givens rotations.

We consider two algorithms. In the first one (§ 4.1) we allow the basic tasks to overlap whereas in the second one (§ 4.2) the code-sections in which these tasks are involved are completely broken up.

4.1 Row-partitionning Column-oriented Decomposition

```
ALGORITHM 4.1.1: ROCDEC
initialize $\left(N, m, N_{\rm loc}, A_{\rm loc}(1:N_{\rm loc},:)\right)$;
for $j := 1 : m$ do
    create my reflector[j] and update my local columns
    if myid() = 0 then
        send row $A_{\rm loc}(j, j:m)$ to myright()
    else
        receive row(j:m) from myleft()
        create my rotation[1,j] to annihilate $A_{\rm loc}(1, j)$
        if myid() $\neq r-1$ update and send row(j:m) to myright()
    endif
endfor
```

```
ALGORITHM 4.1.2: ROCVEC
$k :=$ myid();
$z_{\text{loc}} := 0$;
{apply orthogonal transformations in reverse order ...........}
for $j := m : 1$ step $-1$ do
    if $(k = 0)$ then
        receive $z_{\text{loc}}(j)$ from myright()
        $i := j$;
    else
        if $(k = r - 1)$ then
            apply my rotation$[1, j]$ on $[y(j), z_{\text{loc}}(1)]$
            send the updated $y(j)$ to myleft()
        else
            receive $yj$ from myright()
            apply my rotation$[1, j]$ on $[yj, z_{\text{loc}}(1)]$
            send the updated $yj$ to myleft()
        endif
        $i := 1$;
    endif
    apply my reflector$[j]$ on $z_{\text{loc}}(i : N_{\text{loc}})$
endfor
```

Data distribution. Contiguous rows as in MGSDEC (§ 3.1, Figure 2). The factor R goes to P_{r-1} upon completion. For ROCVEC which computes the product $z = Qy$, the vector $y \in \mathbb{R}^m$ must be at the beginning in processor P_{r-1} but the resulting vector $z \in \mathbb{R}^N$ is split at completion.

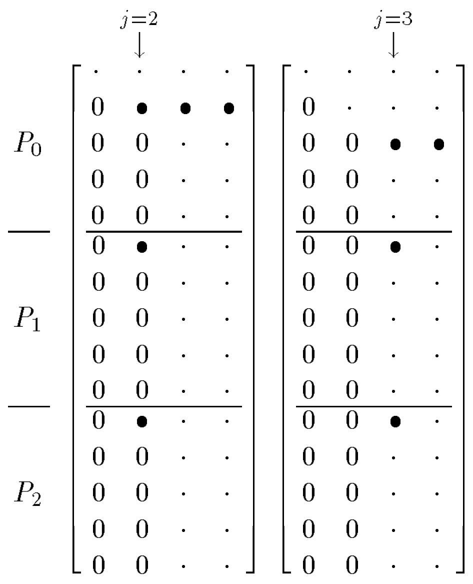


Figure 4: ROCDEC. Factorization in progress.

Description.

ROCDEC At the jth step, processeur P_0 applies the Householder reflector that annihilates on its

jth column the elements below its main diagonal whereas any other processor applies the reflector that annihilates on its jth column the elements below its first row (this is illustrated is Figure 4 when $j = 2$). Then the non null portion of the jth row of processor P_0 is passed round the ring and on the way, in each processor, this row is used to apply the Givens rotation that zero-out the unique element of the column which was not eliminated by the Householder transformation.

`ROCVEC` We explain the principle of this algorithm by using the matrix formulation of the problem. At the end of `ROCDEC` we can write

$$\begin{aligned} z &= Q_m y \\ &= H_1 G_1 \cdots H_m G_m \begin{bmatrix} y \\ 0 \end{bmatrix} \end{aligned}$$

where

$$z = \left[\begin{array}{c} z^{(0)} \\ \hline z^{(1)} \\ \hline \vdots \\ \hline z^{(r-1)} \end{array}\right], \quad H_j = \begin{bmatrix} H_j^{(0)} & & & \\ & H_j^{(1)} & & \\ & & \ddots & \\ & & & H_j^{(r-1)} \end{bmatrix}, \quad G_j = G_{1j}^{(1)} \cdots G_{1j}^{(r-1)}.$$

$H_j^{(k)}$ is the Householder transformation constructed by P_k for column j and $G_{1j}^{(k)}$ is its Givens rotation used to annihilate element $(1, j)$. To get rid of size mismatch we presently assume that the rotations are augmented with the identity matrix and are of consistent sizes. If at step j the current value of z consists already of the previous accumulated products, the current computation $z \leftarrow G_j z$ is done as follow. Processor P_{r-1} starts by applying its rotation and this affects only the component $z^{(0)}(j)$ and $z^{(r-1)}(1)$. An interesting simplification arises here because the jth component of z satisfies at this stage[2] $z^{(0)}(j) \equiv z(j) = y(j)$. Then the updated component $y(j)$ is passed round the ring and this allows the other processors to apply their rotation. When this component reached P_0, its value is $[G_j z]_j$ and is then stored in the suitable entry. Finally the operation $z \leftarrow H_j z$ is complete when each processor applies its own reflector.

Parallelism. The application of Householder reflectors is a completely independent stage. However to annihilate the remaining non zero elements via Givens rotations, it is necessary to transport a row portion along r processors. But as soon as that portion is passed through a processor, the latter applies its next reflector so that there is an overlapping of its computations and the transition of that row portion in the other processors. Since there will be a total of m transfers of this sort during the whole factorization, communication overheads will be masked if $m \gg r$. To summarize all, we expect an interesting speed-up when $\frac{N}{r} \gg m$ and $m \gg r$.

[2]that is why the vector y is into P_{r-1} at the beginning!

4.2 Row-partitionning Diagonal-oriented Decomposition

```
ALGORITHM 4.2.1: RODDEC
initialize $\left(k = \texttt{myid}(), N, m, N_{\text{loc}}, A_{\text{loc}}(1 : N_{\text{loc}}, :)\right)$ ;
{local QR factorization ..................................}
$[Q_{\text{loc}}, R_{\text{loc}}] := \texttt{QR}(A_{\text{loc}})$ ;
{annihilate undesirable $R_{\text{loc}}$ diagonal by diagonal ............}
for $d := 1 : m$ do
    if $(k = 0)$ then
        send row $R_{\text{loc}}(d, d : m)$ to myright()
    else
        receive row(d:m) from myleft()
        create rotation to annihilate my element $R_{\text{loc}}(1, d)$
        if $(k \neq r - 1)$ send the updated row(d:m) to myright()
        create rotations to annihilate my d-th diagonal of $R_{\text{loc}}$
    endif
endfor
```

```
ALGORITHM 4.2.2: RODVEC
{apply Givens rotations in reverse order .....................}
for $d := m : 1$ step $-1$ do
    if (k=0) then
        receive $z_{\text{loc}}(d)$ from myright()
    else
        for $j := m : d + 1$ step $-1$ do
            $i := j - d + 1$ ;
            apply rotation$[i - 1, i]$ on $[z_{\text{loc}}(i - 1), z_{\text{loc}}(i)]$
        endfor
        if $(k = r - 1)$ then
            apply rotation$[1, d]$ on $[y(d), z_{\text{loc}}(1)]$
            send the updated $y(d)$ to myleft()
        else
            receive $yd$ from myright()
            apply rotation$[1, d]$ on $[yd, z_{\text{loc}}(1)]$
            send the updated $yd$ to myleft()
        endif
    endif
endfor
{apply Householder reflectors in reverse order ...............}
for $j := m : 1$ step $-1$ do
    apply reflector$[j]$ on $z_{\text{loc}}(1 : N_{\text{loc}})$
endfor
```

Data distribution. Contiguous rows as in MGSDEC (§ 3.1, Figure 2). The factor R goes to P_{r-1}. To compute the product $z = Qy$ the vector $y \in \mathbb{R}^m$ must be at the beginning in processor P_{r-1}.

The resulting vector $z \in \mathbb{R}^N$ is split at completion.

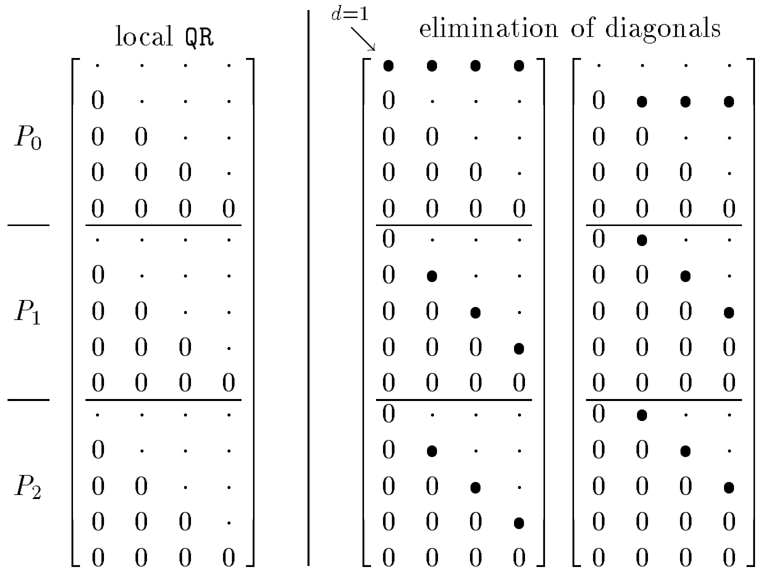


Figure 5: `RODDEC`. Factorization in progress.

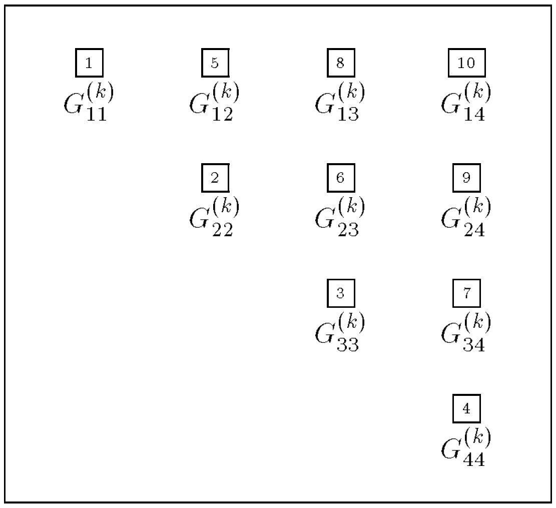


Figure 6: `RODDEC`. Rotations are applied in the order $\searrow\searrow$.

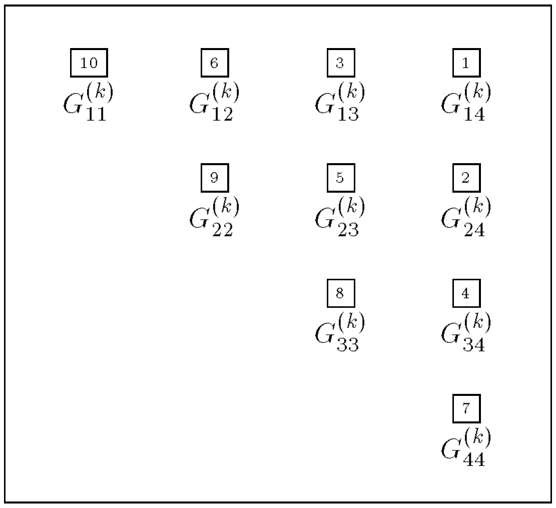


Figure 7: `RODVEC`. Rotations are applied in the order $\nwarrow\nwarrow$.

Description.

`RODDEC` The processors begin by carrying out *completely* in parallel the QR decomposition of their local blocks. After that, it is necessary to zero-out the factors R belonging to processors different from P_0. This is done diagonal by diagonal. At the dth step, the non null portion of the dth row of processor P_0 is passed round the ring and in each processor P_k, this portion is used to apply the Givens rotation $G_{1,d}^{(k)}$, necessary to eliminate the first non null element of its first row. Then, by applying consecutively a sequence of *internal* rotations, $G_{2,d+1}^{(k)}, ..., G_{d,m}^{(k)}$, each defined by the two appropriated local rows, they set to zero the remaining part of their dth diagonal.

`RODVEC` As in `ROCVEC` this algorithm is better understood through a matrix formulation. Let us assume that in entrance of step d, the current value of $z = [z^{(0)}, ..., z^{(r-1)}]^T$ comes from the previous accumulated rotations and is correct. Then each processor P_k except P_0 applies independently its *internal* sequence of rotations (in reverse order of their generations)

$$z^{(k)} \leftarrow G^{(k)}_{2,d+1} \cdots G^{(k)}_{d,m} z^{(k)}.$$

To carry out the remaining rotation $G^{(k)}_{1,d}$ which involves the contribution of the right neighbor except for P_{r-1} since[3] $z^{(0)}(d) \equiv z(d) = y(d)$, the updated value of $y(d)$ is passed round the ring from processor P_{r-1} to P_0. When it reached P_0 it has the actual value of $z(d)$ and is then stored in the suitable entry. Once all the rotations have been applied, the independent Householder transformations used in the local blocks are on their turn applied.

Parallelism. During the individual QR factorization, we achieve the perfect parallelism. But after that, the whole factor R of P_0 must be transferred (and updated) around the ring to enable the cancellation of $\frac{m(m+1)}{2}$ elements per any other processor. Since there are r processors to cross, this strategy will be efficient if $m \gg r$ otherwise, the amount of communications will not be sufficiently covered by the computations.

5 Additional Remarks

We can state here some observations which should have been eclipsed by the technicalities if they were mentioned earlier in the previous sections.

- At the end of all theses algorithms (except `MGSDEC` and `MGSVEC`), there has been roughly m revolutions of messages around the ring. For algorithm `COLDEC` the revolutions involve messages of length N whereas in `ROCDEC` and `RODDEC`, they have an average length of $\frac{m+1}{2}$. In their associated joint algorithms for the product Qy, the messages are of length N for `COLVEC` but are simply scalars for `ROCVEC` and `RODVEC`.

- There is an underlying pipeline chain. However this chain effect disappears within algorithms `ROCDEC` and `RODDEC` if one considers a programming scheme of the style:

```
        • • •
        if (myid() = 0) then
            send $A_{\text{loc}}(d, d : m)$ to myright()
$\ominus \longrightarrow$ receive $A_{\text{loc}}(d, d : m)$ from myleft()
        else
            receive row(d:m) from myleft()
            • • •
            send the updated row(d:m) to myright()
            • • •
        endif
```

The processor P_0 will have to wait until the end of the revolution and as a consequence of the interruption of the production-line work, all the other intermediate processors will also have to wait. We have avoided this side-effect in our implementations, and that is why the factor R is in the last processor P_{r-1} at the termination of the processes.

[3]that is also why, as in `ROCVEC`, the vector y must belong to P_{r-1} at the beginning.

- Practically, the extra storage which is used in the real implementations is negligible. The so-called «essential parts» of Householder vectors are in the lower trapezoidal part of A_m whereas the Givens rotations are in the place of the elements to zero as it is preconised by Stewart's technique [11].

- The factors Q_m and R_m as obtained, may have elements of different signs from a method to another. To recover the unique QR decomposition, i.e. the one in which the diagonal of R is positive, it is necessary to (implicitly) apply the transformation $A_m = (Q_m D_m)(D_m R_m)$ where $D_m = \text{diag}(\text{sign}(r_{11}), ..., \text{sign}(r_{mm}))$.

- When building up A_m by formulas (7) and (8) before the QR factorization step, the eventuality that the columns may be linearly dependent is not considered. However this situation, which is known as a *breakdown*, must be reported. Therefore if the case of linear dependence of columns is detected within the QR step, an ad-hoc action should be issued.

6 Numerical Tests

6.1 The Intel Paragon

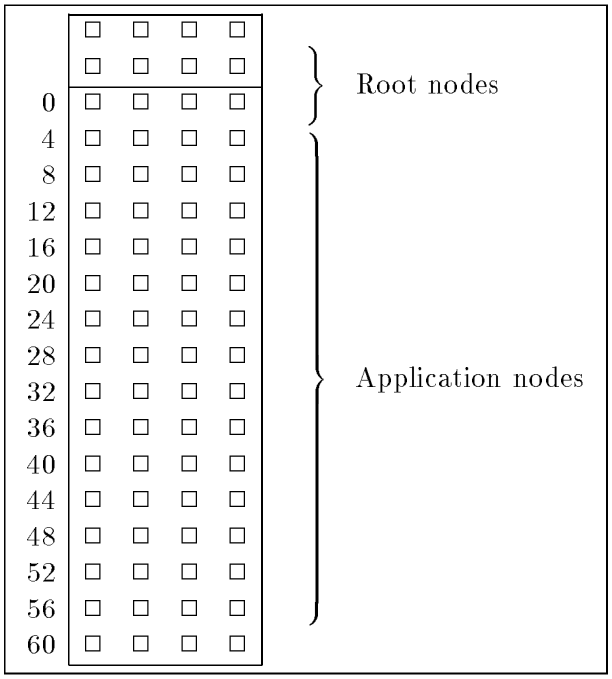


Figure 8: Network topology.

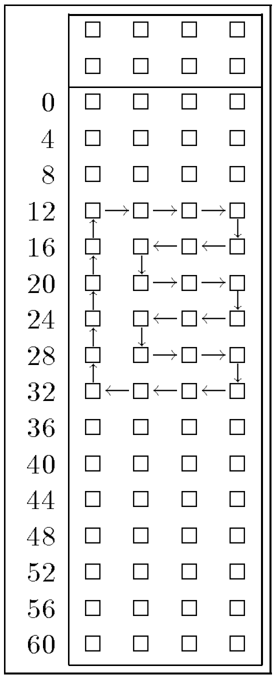


Figure 9: Mapping a ring on the grid.

The Paragon on which we carry out our experiments is a distributed-memory meshgrid of size $16 \times 4 = 64$ nodes. Each node consists in 16MB of RAM and two i860 64-bit microprocessor chips. One chip among the two is dedicated to communications but is not operating yet, at this date, on the actual Paragon XP/S i860 model. At this date also, a large part of the RAM is occupied by the system so that the main memory remaining for application purposes (code, data, message-buffers) is about 7MB only. Since in this 64-Node model (as well as in the others), 8 nodes are reserved for the system management (the *root partition*), we are able to use up to 56 processors[4]. A ring is obtained by a logical mapping. However there is no predefined functions

[4]This is true on the present releases but may not be true in the future versions where some nodes in the root

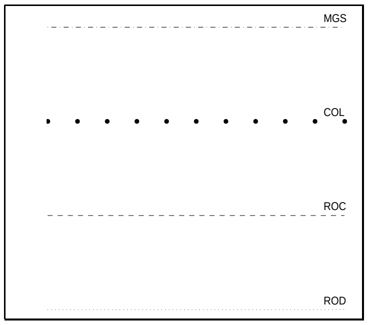
for such a mapping and the user must simulate the ring by himself. In our mapping we have insured that the right and the left neighbors of a processor are also physically its neighbors (Figure 9). The computations are in double precision and we make use of the scientific library BLAS available in its optimized version for the machine. We intensively use the vector-vector operations `DDOT`, `DAXPY` and `DNRM2`.

6.2 Timing results

In this section, for several values of N, we present the time spent in running the Paragon Fortran code corresponding to the algorithms. The values indicated on the tables represent the time spent in the obtention of the decomposition augmented by the time spent in computing one product Qy. We use the following abbreviations:

`MGS`: `MGSDEC` follows by `MGSVEC`
`COL`: `COLDEC` follows by `COLVEC`
`ROC`: `ROCDEC` follows by `ROCVEC`
`ROD`: `RODDEC` follows by `RODVEC`.

Legend.
The graphic on the right represents the drawing symbols adopted in our curves.
The plain line will always represent the perfect speedup.



During our experiments, we have to face lack of memory when attempting to run very large problems on few nodes. As a consequence, the times obtained were perturbed by memory swaps. That is why we decide to consider speed-ups with respect to the number of nodes in which little memory misses occur. More specifically, if n is the smallest number of nodes for which the main matrix $A_m \in \mathbb{R}^{N \times m}$ fits in the distributed-memory without performance overheads, the speed-up with respect to (w.r.t) n processors is $T(r)/T(n)$. Those numbers are indicated by arrows on the tables. Our timing does not take into consideration the time used to load the matrix. In fact, we use dynamic space allocation to reserve exactly the sufficient memory storage needed within each processor for the problem in hand. Then the data were randomly generated.

partition could be used like any other node of the application partition. The system is evolving quite rapidly so that all the information mentioned here could be out-of-date at the time you read this.

Example 6.1

$N = 10{,}000$, $m = 40$				
r	MGS	COL	ROC	ROD
1	0.16E+01	0.18E+01	0.18E+01	0.18E+01
2	0.13E+01	0.17E+01	0.10E+01	0.10E+01
4	0.12E+01	0.17E+01	0.61E+00	0.63E+00
8	0.15E+01	0.19E+01	0.32E+00	0.33E+00
16	0.16E+01	0.22E+01	0.25E+00	0.27E+00
24	0.17E+01	0.25E+01	0.23E+00	0.24E+00
32	0.18E+01	0.26E+01	0.22E+00	0.23E+00
56	0.20E+01		0.22E+00	0.22E+00

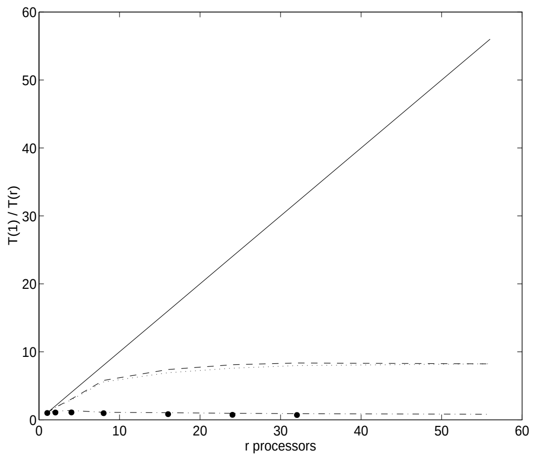


Figure 10: Time in seconds and speed-up w.r.t 1 processor.

This is a problem of moderate size. MGS and COL are the more slowest methods. In addition their speed-ups are very poor. ROC and ROD perform slightly better. They achieve a speed-up of about 8 with 24 processors but they didn't go further with more processors.

Example 6.2

$N = 50{,}000$, $m = 40$				
r	MGS	COL	ROC	ROD
1				
2				
4	0.29E+01	0.79E+01	0.24E+01	0.24E+01
8	0.22E+01	0.80E+01	0.13E+01	0.13E+01
16	0.20E+01	0.93E+01	0.80E+00	0.80E+00
24	0.24E+01	0.10E+02	0.53E+00	0.53E+00
32	0.23E+01	0.12E+02	0.42E+00	0.43E+00
56	0.23E+01		0.34E+00	0.34E+00

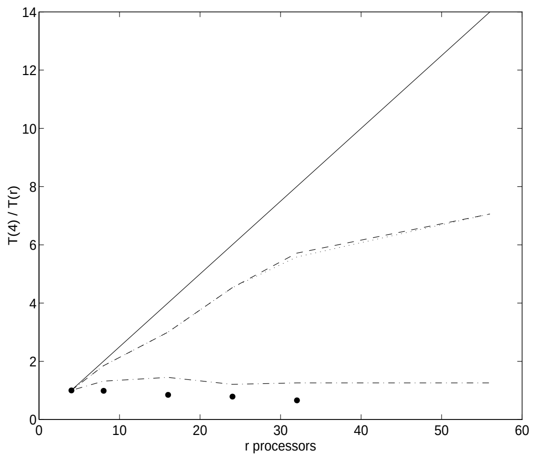


Figure 11: Time in seconds and speed-up w.r.t 4 processors.

Here once again MGS and COL are the slowest. No speedup is achieved for MGS while COL becomes more and more poorer. The behaviors of ROC and ROD are almost identical. After achieving a speedup of about 5.6 for 32 processors, they are about to stagnate. This problem comes from the fact that from there, m is no more large when compared to r. Hence this observation is in accordance to what we were expecting in advance (see the discussions about their potentiel parallelism).

Example 6.3

$N = 100{,}000,\ \ m = 40$				
r	MGS	COL	ROC	ROD
1				
2				
4	0.84E+01	0.15E+03	0.34E+02	0.67E+01
→ 8	0.33E+01	0.16E+02	0.24E+01	0.25E+01
16	0.26E+01	0.18E+02	0.14E+01	0.14E+01
24	0.25E+01	0.20E+02	0.10E+01	0.10E+01
32	0.24E+01	0.22E+02	0.84E+00	0.84E+00
56	0.27E+01		0.51E+00	0.50E+00

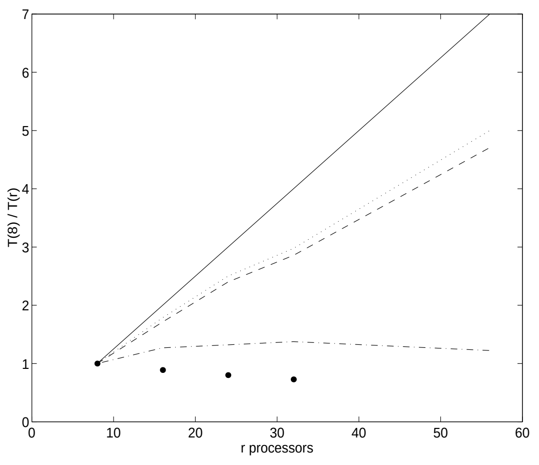


Figure 12: Time in seconds and speed-up w.r.t 8 processors.

With this large problem we can definitively conclude that `MGS` and `COL` are not competitive in our context. However in contrast, `ROC` are `ROD` appear to be nicely scalable.

In the following series of examples, we present the time behavior of the algorithms when we fix a constant load for each processor whatever is the total number of processors. If for reference, we consider that the complexity of these algorithms is about $\mathcal{O}(2Nm^2)$, we can clearly see with the very last example that we cross a rate performance of 779 Mflops for 56 nodes, i.e. 13.9 Mflops per processor.

Example 6.4

$\frac{N}{r} = 100,\ \ m = 40$				
r	MGS	COL	ROC	ROD
1	0.50E-01	0.53E-01	0.56E-01	0.48E-01
2	0.44E+00	0.11E+00	0.10E+00	0.12E+00
4	0.77E+00	0.21E+00	0.14E+00	0.15E+00
8	0.11E+01	0.43E+00	0.15E+00	0.17E+00
16	0.14E+01	0.77E+00	0.17E+00	0.18E+00
24	0.16E+01	0.10E+01	0.18E+00	0.18E+00
32	0.17E+01	0.12E+01	0.18E+00	0.18E+00
56	0.19E+01		0.20E+00	0.19E+00

Example 6.5

$\frac{N}{r} = 500,\ \ m = 40$				
r	MGS	COL	ROC	ROD
1	0.14E+00	0.11E+00	0.11E+00	0.10E+00
2	0.57E+00	0.23E+00	0.16E+00	0.17E+00
4	0.91E+00	0.47E+00	0.19E+00	0.21E+00
8	0.13E+01	0.95E+00	0.21E+00	0.22E+00
16	0.15E+01	0.19E+01	0.23E+00	0.23E+00
24	0.18E+01	0.30E+01	0.24E+00	0.24E+00
32	0.19E+01	0.39E+01	0.25E+00	0.25E+00
56	0.20E+01		0.23E+00	0.24E+00

Example 6.6

$\frac{N}{r} = 1000, \ m = 40$				
r	MGS	COL	ROC	ROD
1	0.27E+00	0.17E+00	0.18E+00	0.17E+00
2	0.74E+00	0.41E+00	0.23E+00	0.24E+00
4	0.11E+01	0.81E+00	0.26E+00	0.28E+00
8	0.14E+01	0.16E+01	0.28E+00	0.30E+00
16	0.18E+01	0.33E+01	0.31E+00	0.30E+00
24	0.19E+01	0.53E+01	0.32E+00	0.32E+00
32	0.21E+01	0.77E+01	0.33E+00	0.32E+00
56	0.23E+01		0.36E+00	0.35E+00

Example 6.7

$\frac{N}{r} = 10{,}000, \ m = 40$				
r	MGS	COL	ROC	ROD
1	0.16E+01	0.18E+01	0.18E+01	0.18E+01
2	0.21E+01	0.33E+01	0.19E+01	0.19E+01
4	0.25E+01	0.11E+02	0.20E+01	0.20E+01
8	0.28E+01	0.24E+02	0.20E+01	0.20E+01
16	0.32E+01	0.54E+02	0.21E+01	0.21E+01
24	0.34E+01	0.13E+03	0.21E+01	0.21E+01
32	0.36E+01	∞	0.22E+01	0.22E+01
56	0.40E+01		0.24E+01	0.23E+01

The results show that algorithms `ROC` and `ROD` are much more scalable that the two others. Actually the global sum function involved by MGS prevents a valuable efficiency as long as the local vector length in any processor is shorter than a few thousands.

7 Conclusion

We have described parallel algorithms for which to construct orthonormal bases of Krylov subspaces. Algorithms to access the computed bases thanks to their multiplication by a vector were also presented. All these algorithms have been designed in the context of a distributed-memory multiprocessor environment. Several tests have been carried out on a 64-Node Intel Paragon XP/S i860 and it appears that, as far as parallelism is concerned, this way of constructing Krylov bases is suitable. Alternative versions for shared-memory machines can be directly designed. In fact, according to the technological details of a given architecture one can take into account several considerations during their implementation as done in [3].

Acknowledgement. We thank Y. Jegou and H. Leroy at IRISA/INRIA for answering technical questions on the Paragon.

References

[1] Z. Bai, D. Hu, and L. Reichel. An implementation of the GMRES method using QR factorisation. In J. Dongarra, K. Kenedy, P. Messina, D. C. Sorensen, and R. G. Voigt, editors, *Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Philadephia, 1992.

[2] D. Calvetti, J. Petersen, and L. Reichel. A parallel implementation of the GMRES method. *Numer. Lin. Algebra*, to appear.

[3] J. J. Dongarra, A. H. Sameh, and D. C. Sorensen. Implementation of some concurrent algorithms for matrix factorization. *Parallel Computing*, 3:25–34, 1986.

[4] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, second edition, 1989.

[5] D. P. O'Leary and Peter Whitman. Parallel QR factorization by Householder and modified Gram-Schmidt algorithms. *Parallel Computing*, 16:99–112, 1990.

[6] B. Philippe and R. B. Sidje. *Transient Solutions of Markov Processes by Krylov Subspaces.* Technical Report 1989, INRIA, August 1993. in revision for SIAM J. Sci. Comput.

[7] A. Prothen and P. Raghavan. Distributed orthogonal factorization: Givens and Householder algorithms. *SIAM J. Sci. Statist. Comput.*, 10:1113–1135, 1989.

[8] L. Reichel. Newton interpolation at Leja points. *BIT*, 30:332–346, 1990.

[9] Y. Saad. Analysis of some Krylov subspace approximations to the matrix exponential operator. *SIAM J. Numer. Anal.*, 29(1):208–227, February 1992.

[10] A. H. Sameh. Solving the linear least squares problem on a linear array of processors. In L. Synder, editor, *Algorithmically Specialized Computers*, pages 191–200, Academic Press, NY, 1985.

[11] G. W. Stewart. The economical storage of plane rotations. *Numer. Math.*, 25:137–138, 1976.

[12] J. Zhu. QR factorization for the regularized least squares problem on hypercubes. *Parallel Computing*, 19:939–948, 1993.