Analysis of Different Partitioning Schemes for Parallel Gram-Schmidt Algorithms *

S. OLIVEIRA[‡], L. BORGES[†], M. HOLZRICHTER[†] and T. SOMA[‡]

[‡]*Dep. of Computer Science, The University of Iowa, Iowa City, IA 52246-1419.*
Corresponding author email: oliveira@cs.uiowa.edu

[†]*Dep. of Computer Science, Texas A&M University, College Station, TX 77843-3112.*

Abstract

In this paper we analyse implementations of parallel Gram-Schmidt orthogonalization algorithms. One of the first parallel orthogonalization of Gram-Schmidt was the row-wise partitioning of O'Leary and Whitman. In this paper we describe a pipelined implementation which uses column-wise partitioning schemes. Timing models for the column-wise parallel algorithms are derived. We compare our column-wise partitionings against the row-wise partitionings and validate our study with computational results. The pipelined orthogonalization algorithm is important because the timing analysis is independent of the architecture model. Threshold values of m_{max}, which is the number of rows where row partitioning becomes better than column partitioning are found theoretically and verified with our experiments.

Mathematics Subject Classification: 65F25; 65Y05
Key words and phrases: Gram-Schmidt orthogonalization, parallel algorithms.

1 Introduction

Modified Gram-Schmidt (MGS) orthogonalization is a basic algorithm which arises up repeatedly as part of diverse problems in Numerical Linear Algebra. The most common use of MGS is to create the QR factorization of matrix A. Given a finite set of n linearly independent vectors $\{a_1, a_2, \cdots, a_n\}$, where $a_i \in \mathbb{R}^m$, $i = 1, 2, \cdots, n$, the Gram-Schmidt algorithm constructs a sequence of orthonormal vectors $\{q_1, q_2, \cdots, q_n\}$ such that $\{q_1, q_2, \cdots, q_i\}$ forms an orthonormal basis of the space spanned by $\{a_1, a_2, \cdots, a_i\}$ for $i = 1, 2, \cdots, n$. In

*This research was supported by NSF grant ASC-9528912 and a Texas A&M University Interdisciplinary Initiative Award.

matrix form, consider an $m \times n$ matrix $A = [a_1\ a_2\ \cdots\ a_n]$. The Gram-Schmidt orthogonalization factors A into the product of matrices Q and R:

$$A = QR,$$

where $Q \in \mathbb{R}^{m \times n}$ has orthonormal columns and $R \in \mathbb{R}^{n \times n}$ is upper triangular. The QR factorization in turn can be used to solve least squares problems (alternatively Householder reflections or Givens rotations can be used to derive the QR factorization of A). The original procedure for the orthogonalization was introduced by Schmidt [15] to provide a simple and powerful theoretical tool for linear algebra analysis. This procedure is now known as the Classical Gram-Schmidt (CGS) algorithm. However, it turns out that this procedure is numerically unstable and the constructed matrix Q could be far from orthogonal. Various modifications of the Gram-Schmidt algorithm have been presented to overcome the instability of CGS. Mathematically all these variants are equivalent, i.e., they all produce the same set of orthonormal vectors. However, from the numerical point of view, they behave differently. A particular modification of Classical Gram-Schmidt is known as Modified Gram-Schmidt (MGS) algorithm and has two algorithmical versions: row-oriented and column-oriented [2] (row-oriented MGS produces the matrix R row by row, while column-oriented MGS constructs R column by column). In this paper we are concerned with parallel implementation and analysis of parallel algorithms for row-oriented MGS orthogonalization.

Many fields of science have researched the development of parallel MGS algorithms. Early high performance developments presented implementations for vector computers [4, 8] or implementations based on parallel languages (ACLAN processing language) [17, 20]. Depending on partitioning schemes, various parallel implementations can be developed. O'Leary and Whitman [13] discussed parallel implementation for row-oriented Gram-Schmidt orthogonalization with row-wise partitionings on distributed MIMD parallel machines [1]. In [3] we followed the row-oriented data distribution approach for the whole parallel eigenvalue algorithm (including the basis orthogonalization step). Other parallel orthogonalization implementations are available in software packages: ScaLAPACK, PARPACK [11], PESSL, PeIGS [7, 9]. Some of these packages perform QR factorizations based on Householder's reflections (ScaLAPACK). Other packages are strongly based on ScaLAPACK (PESSL from IBM, incorporating PBLAS and BLACS); PARPACK used MGS in its orthogonalization routine, but assumes that a processor can store all the vectors, and then performs internal orthogonalization (by copying the whole basis into each one of the processors). PeIGS incorporate some of the data partitioning and pipelined ideas similar to the ones presented in this paper, namely block-cyclic column partitioning. Due to the wide range of scientific applications which require the use of MGS, it is paramount to have a unified model to judge the above distinct

[1]In their paper, parallel implementation of the Householder algorithm was also investigated.

schemes. Meyer, Nioykindi and Berghe [12] presented a study to compare row and column-cyclic partitioning by analyzing the performance of both strategies when applied to square matrices. Similarly, the complexity analysis of [19] was solely for square matrices. However, many applications arise in the shape of general matrices. This shows the need for deeper comprehensive analyses incorporating various partitioning schemes for general rectangular matrices. The main aspect of this this paper is a presentation of a theoretical analysis of some of the possible partitioning schemes for MGS. The numerical results presented here were first obtained as part of a *Distributed Algorithms* course taught at Texas A&M University by the main author. Here we concentrate on the analysis and comparison of four types of column-wise partitionings and the first row-wise partitioning parallel implementation of O'Leary. Another important aspect of our column-wise algorithms is our theoretical analysis predicts architecture independent performance.

Since many new algorithms in numerical linear algebra use re-started approaches, i.e., keep the number of columns of A bounded and usually $n \leq m$, it seems at first that row-wise partitioning of the vector data may be the best possible choice. In this paper we show that even when $m > n$ column-wise partitioning may sometimes be still the best choice. Analytical models for parallel execution time required by these implementations are derived and compared with numerical results. Finally, threshold values of m, which is the number of rows where row-wise partitioning becomes better than column-wise partitioning, are obtained. The remainder of the paper is organized as follows. In Section 2 we introduce modified Gram-Schmidt orthogonalization and column-wise partitioning schemes, and in Section 3 we discuss our parallel implementations for the the column-wise partitioning algorithms presented. In Section 4 we describe the row-wise partitioning schemes and discuss their parallel implementation as in [13]. In Section 5 comparisons of column-wise and row-wise partitionings are studied. Finally in Section 6 we will present numerical results and in Section 7 we present our conclusions.

2 Column-wise Modified Gram-Schmidt Orthogonalization

Row-oriented MGS can be written as follows [2].
The row-oriented MGS algorithm (version 1):

for $i = 1$ *to* n
    $v_i = a_i$;
end;
for $i = 1$ *to* n
    $r_{ii} = ||v_i||_2$;
    $q_i = v_i / r_{ii}$;

```
7.    for j = i + 1 to n
8.       r_ij = q_i^T v_j;
9.       v_j = v_j - r_ij q_i;
10.   end;
11. end;
```

It is noted that if v_j is replaced with a_j in line 8, the algorithm is transformed into the row-oriented version of the classical Gram-Schmidt algorithm. Compared with CGS, MGS uses the most recently computed value v_j in the calculation of the coefficient r_{ij}. This modification makes the MGS algorithm more stable than the CGS algorithm in the sense that the orthogonality error ($\|Q^TQ - I\|$) is bounded by a modest constant times machine precision times the condition number of A [1]. The reader may refer to Björck's survey paper [2], where variants of the classical Gram-Schmidt algorithm are discussed.

Our pipelined parallel algorithm is based on the following key observation. Let $P_{\perp q}$ denote the projector onto the space orthogonal to a nonzero vector $q \in \mathrm{I\!R}^{\mathrm{m}}$. Then the sequential MGS can be written as in [18]

$$v_j = P_{\perp q_{j-1}} P_{\perp q_{j-2}} \cdots P_{\perp q_1} a_j, \qquad j = 2, 3, \cdots, n,$$

where

$$q_j = \frac{v_j}{||v_j||_2}, \qquad j = 1, 2, \cdots, n.$$

Let

$$v_j^{(i)} = P_{\perp q_{i-1}} P_{\perp q_{i-2}} \cdots P_{\perp q_1} a_j, \qquad 2 \le i \le j \le n.$$

With this notation, we have that

$$v_j^{(i)} = P_{\perp q_{i-1}} v_j^{(i-1)}, \qquad 2 \le i \le j \le n, \tag{1}$$

and the row-oriented MGS can be rewritten in the following form.
The row-oriented MGS algorithm (version 2) :

```
for j = 1 to n
   v_j = a_j;
end;
for i = 1 to n
   q_i = v_i/||v_i||_2;
   for j = i + 1 to n
      v_j = P_⊥q_i v_j;
   end;
end;
```

Equation (1) shows that computation of MGS depends on two parameters i and j, whose roles are symmetric. In other words, row-oriented MGS can be considered as a sequence of projections $P_{\perp q_i}, i = 1, 2, \cdots, n$ with each projection in turn applied to different vectors $v_j^{(i-1)}$ for $j \ge i$ immediately after q_i is

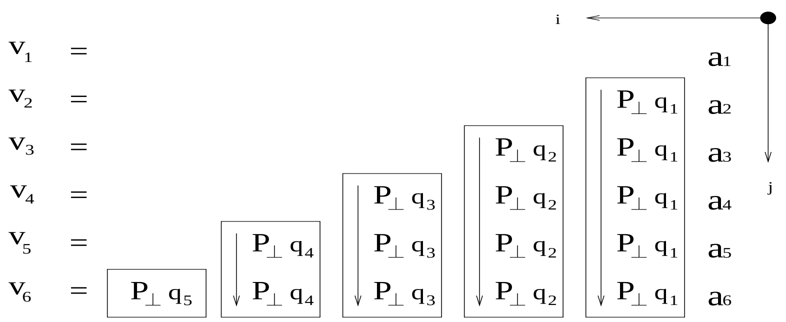


Figure 1: Pipelined MGS algorithm.

computed. Thus, if we consider these projections (with respect to i) as computational subtasks and different vectors applied (with respect to j) as different stages of a subtask, then we find that row-oriented MGS can be implemented in a pipelined fashion [5]. For a general description of pipelined algorithms refer to [10].

This observation may be better illustrated in Figure 1, where the orthogonalization of six vectors is considered. In this figure, the ith column projects all the current vectors v_j onto a subspace orthogonal to column q_i (each projection step corresponds to a different row of the ith column). Consequently, the row-oriented MGS algorithm can be pipelined. In the following sections, we shall describe in detail the pipelined parallel implementation for row-oriented MGS based on various column-wise striped partitioning schemes [14, 16]. Throughout our discussion, p denotes the number of processors involved in computation and label the processors by $P_1, P_2, \cdots, P_p$. We assume that $p \leq n$. For convenience, we call the computation represented by each column in Figure 1 as one round of projections, which, in other words, corresponds to one outer loop in the row-oriented MGS algorithm (version 2).

Our new implementations consider striped partitioning by columns, namely, matrix A is divided into groups of columns and each processor is assigned one such group. Column-wise partitioning schemes can be grouped into four types: column, block, cyclic or block-cyclic. For column partitioning each processor stores one column of the matrix; for block partitioning, each processor is assigned l consecutive columns; and for column-wise cyclic partitioning, the columns of the matrix are sequentially distributed among the processors in a wrap–around fashion (k blocks per processor). Obviously, if the number of columns equals to the number of processors, these three types of partitioning are identical. One may combine column-wise block partitioning and column-wise cyclic partitioning into column-wise block-cyclic partitioning, with which the matrix is striped into a number of blocks with each block consisting of a fixed number of consecutive columns (l) and these blocks are distributed among the processors in a wrap–around manner (k blocks per processor).

3 Column-wise Parallel Implementations

3.1 One-column partitioning

We first consider the simplest case, where $p = n$. That is, each processor contains only one column. We assume that each processor P_k stores column a_k, where $k = 1, 2, \cdots, n$. The pipelined algorithm proceeds as follows. Initially, processor P_1 computes vector q_1 and sends it to processor P_2. Once P_2 receives q_1, it forwards q_1 to P_3 and computes $v_2^{(2)} = P_{\perp q_1} v_2^{(1)} = P_{\perp q_1} a_2$. While P_3 forwards q_1 to its next neighboring processor P_4, P_2 can initialize the second round of projections by computing $q_2 = v_2^{(2)}/||v_2^{(2)}||_2$ and sending it to P_3, and so on. In general, during the k^{th} round of projections, processor P_k computes vector q_k and sends it to its next neighbor P_{k+1}. Whereas processor P_j, $j \geq k+1$, after receiving vector q_k from its previous neighbor P_{j-1}, forwards q_k to its next neighbor P_{j+1} and then computes $v_j^{(k+1)} = P_{\perp q_k} v_j^{(k)}$. Meanwhile, P_{k+1} can compute q_{k+1} immediately after it finishes the computation of $v_{k+1}^{(k+1)}$, and starts the $(k+1)^{th}$ round of projections by computing vector q_{k+1} and sending q_{k+1} to processor P_{k+2}. This pipelined algorithm is graphically illustrated in Figure 2, where q_i indicates the vector normalization (line 5 in version 2), $P_\perp q_i$ indicates a projection (line 7 in version 2).

To analyze the timing of this parallel algorithm we assume that communication is synchronous and takes time $t_s + mt_w$ to pass a message of size m words over a hop, where t_s is the startup time and t_w is the per-word transfer time. To simplify our analysis, we further assume that floating point operations, including addition, subtraction, multiplication, division, and square root, take unit time.

The parallel run time for a parallel algorithm can be derived by finding a *critical path* for the algorithm which must be performed sequentially. The critical path for the parallel algorithm shown in Figure 2 is indicated by the dashed line. Arrows indicate the communication where a orthonormalized column q_i is sent to the next processor.

For the computation time, we observe from Figure 2 that the critical path contains the computation for the vector normalization, which requires $3m$ time, and the projection, which takes $4m-1$ time, performed by each processor, except for the first processor, which only needs the vector normalization. Thus, the computation time is given by $T_{comp}^{one-column} = \sum_{i=1}^{n}(3m+4m-1)-(4m-1)$. As for the communication time, we notice that each processor passes two messages of size m to its next neighbor, taking time $2(t_s+mt_w)$, except for processor P_1, which sends only one message, and processor P_n, which does not forward any

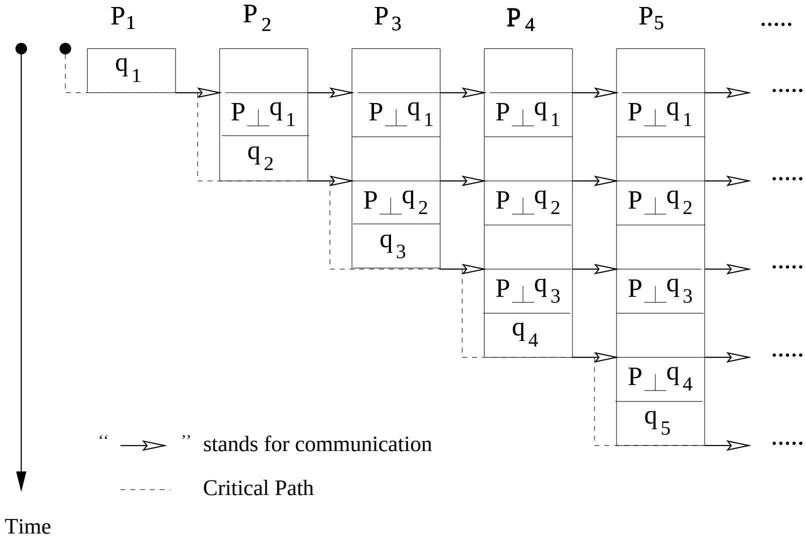


Figure 2: One column partitioning.

message in the pipeline chain. Thus, the communication time is $T_{comm}^{one-column} = (2n-3)(t_s + mt_w)$. Consequently, the parallel run time is given by

$$\begin{aligned} T_p^{one-column} &= \sum_{i=1}^{n}(3m + 4m - 1) - (4m-1) + (2n-3)(t_s + mt_w) \\ &= 7mn - 4m - n + 1 + (2n-3)(t_s + mt_w). \end{aligned} \quad (2)$$

It follows that the cost of this parallel algorithm is

$$\begin{aligned} C_p^{one-column} &= pT_p^{one-column} \\ &= 7mn^2 - 4nm - n^2 + n + n(2n-3)(t_s + mt_w). \end{aligned} \quad (3)$$

To analyze the performance of the parallel algorithm, we also need to find the sequential run time for the CGS algorithm. Since in each round of projections the time required for performing the vector normalization is $3m$ and the time for performing the orthogonalization is $4m - 1$, the total time for the sequential algorithm is

$$T_s = \sum_{i=1}^{n}\left[3m + \sum_{j=i+1}^{n}(4m-1)\right] = 2mn^2 + mn - \frac{1}{2}(n^2 - n). \quad (4)$$

Comparing (4) with (3), we observe that the cost of the parallel algorithm is higher than the sequential run time by a factor of $7/2$, even if the communication overhead is neglected. The overhead in the computation is mainly due to processor idling.

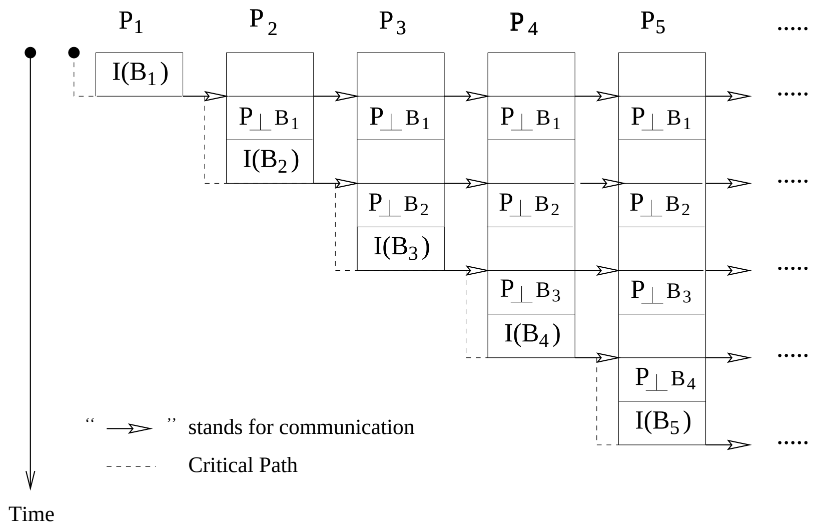


Figure 3: Block partitioning.

3.2 Block partitioning

The pipelined parallel algorithm presented in the previous subsection can easily be adapted to the block partitioning scheme, in which each processor is assigned a block consisting of $\ell = n/p$ consecutive columns of the matrix. Thus, instead of performing computation on one column as in the previous case, each processor will perform computation on one block, thereby producing less computational overhead. We may combine all orthonormal vectors from one block together to form one message (called a *block message*) so that the communication time is reduced.

The parallel algorithm for this case can be depicted by Figure 3, where an arrow indicates the communication of a block message B_i and $I(B_i)$ is understood as a set of orthonormal vectors obtained by applying sequential MGS orthogonalization to the block B_i of consecutive columns which are assigned to processor P_i. In the same way, $P \perp B_i$ represents a series of projections with each reference vector in a set of orthonormal vectors B_i passed by a previous processor. Block subscripts reflect the order in which blocks are stored in the original matrix. It can be shown that the time for the MGS orthogonalization of a block is given by $\sum_{i=1}^{\ell}\left[3m + \sum_{j=i+1}^{\ell}(4m-1)\right]$, whereas the time required for the projections is given by $\sum_{i=1}^{\ell}\sum_{j=1}^{\ell}(4m-1)$, where ℓ is the length of a block. In the context of block partitioning, we have $\ell = n/p$. Analogous to the previous case, processor P_1 performs one less projection operation than other processors do. Thus, the computation time is

$$\begin{aligned} T_{comp}^{block} &= p\{\sum_{i=1}^{\ell}[3m + \sum_{j=i+1}^{\ell}(4m-1)]\} + (p-1)\{\sum_{i=1}^{\ell}\sum_{j=1}^{\ell}(4m-1)\} \\ &= \frac{6mn^2}{p} + mn - \frac{3n^2}{2p} + \frac{1}{2}n - (4m-1)\frac{n^2}{p^2}. \end{aligned}$$

Since the size of a block message is $m\ell$, the communication time is

$$T_{comm}^{block} = (2p-3)(t_s + m\ell t_w) = (2p-3)(t_s + \frac{mn}{p}t_w) \ .$$

Thus the parallel run time for the block partitioning is

$$T_p^{block} = \frac{6mn^2}{p} + mn - \frac{3n^2}{2p} + \frac{1}{2}n - (4m-1)\frac{n^2}{p^2} + (2p-3)(t_s + \frac{mn}{p}t_w),$$

which is reduced to equation (2) if $p = n$. The cost for the parallel algorithm now becomes

$$C_p^{block} = 6mn^2 + mnp - \frac{3n^2}{2} + \frac{1}{2}pn - (4m-1)\frac{n^2}{p} + (2p-3)(pt_s + mnt_w) \ .$$

It is observed that for n sufficiently large, the parallel algorithm with the block partitioning is more efficient than the column partitioning case. This is because the communication in the block partitioning scheme takes less time. Nevertheless, we still have computation overhead in this algorithm due to processor idling, even if communication is ignored.

3.3 Cyclic partitioning

Cyclic partitioning has better load balance than block partitioning. In cyclic partitioning, processor P_i contains columns $a_i, a_{i+p}, \cdots, a_{i+n-p}$. The parallel algorithm for cyclic partitioning is illustrated in Figure 4 where processors are distributed in a wrap-around fashion.

Figure 4 shows the pipelined algorithm when three processors are used. Let k denote the number of columns assigned to each processor, called *the wrap–around number* ($k = n/p$). Each time a processor P_i performs the normalization on a column, say column v_j, it sends the resulting normal vector q_j to its next neighbor and then performs the corresponding projections of the remaining columns. Since the number of columns in each processor decreases from k to 0, we denote such projections as $P_{k-s}\perp q_j$ (meaning projection of the remaining $k-s$ columns on the processor onto the space orthogonal to q_j where s is the iteration or cycle number ($s = 1, \ldots, k$). Hence, the computation time is given

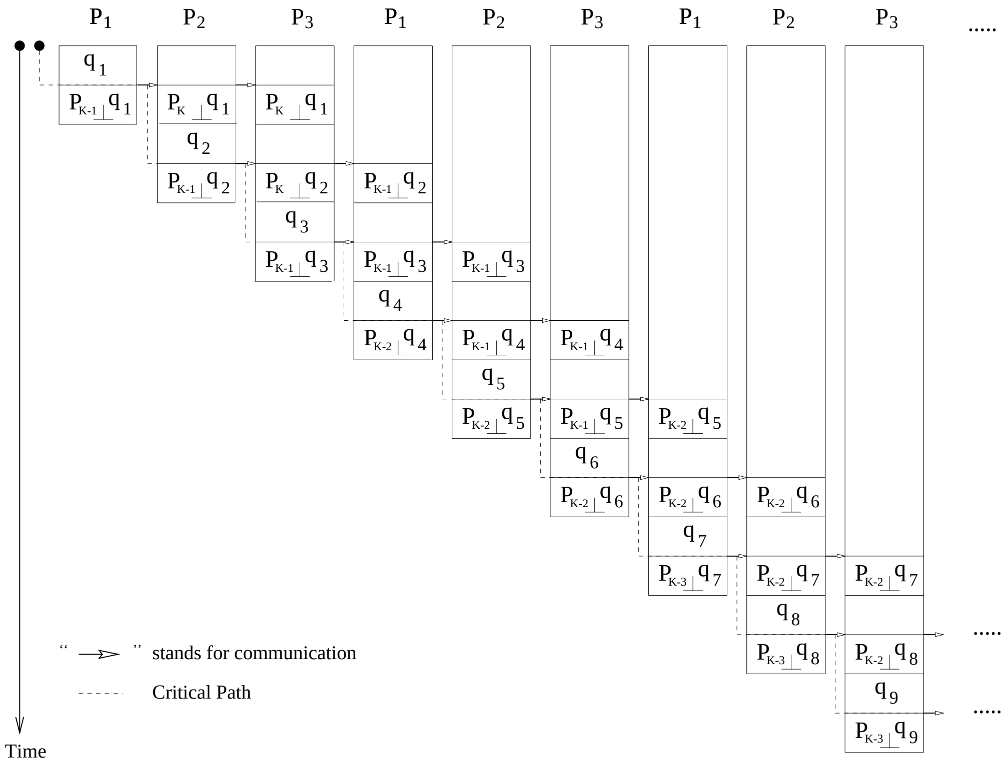


Figure 4: Cyclic partitioning: Case 1.

by

$$T_{comp}^{cyclic} = p\{\sum_{i=0}^{k-1}[3m + \sum_{j=1}^{k-i}(4m-1)]\} - (4m-1)k = \frac{2mn^2}{p} + 5mn - \frac{n^2}{2p} - \frac{1}{2}n - (4m-1)\frac{n}{p} \quad (5)$$

and the communication time is given by

$$T_{comm}^{cyclic} = (2kp-3)(t_s + mt_w) = (2n-3)(t_s + mt_w). \quad (6)$$

As a result, the parallel run time for cyclic partitioning is

$$T_p^{cyclic} = \tfrac{2mn^2}{p} + 5mn - \tfrac{n^2}{2p} - \tfrac{1}{2}n - (4m-1)\tfrac{n}{p} + (2n-3)(t_s + mt_w)\ , \quad (7)$$

which yields the cost

$$C_p^{cyclic} = 2mn^2 + 5mnp - \tfrac{n^2}{2} - \tfrac{1}{2}pn - (4m-1)n + p(2n-3)(t_s + mt_w). \quad (8)$$

Again, we see that equation (7) is reduced to (2) if $p = n$. However, from equation (8) we observe that unlike the previous two partitionings, cyclic partitioning does not have the computational overhead if n is sufficiently large with respect to p. On the other hand, cyclic partitioning has more communication overhead than block partitioning. This suggests that combining the block partitioning and the cyclic partitioning may give better performance as the resulting hybrid method will provide a tradeoff between computation and communication.

An immediate improvement can be seen by looking at Figure 4. Note that for processors $P_2, \ldots, P_p$ the only projection needed before the vector normalization q_i is the projection of the vector q_{i-1} just received against column v_i. Therefore, for P_2 only orthogonalization of v_2 in relation to q_1 needs to be performed before the normalization step which generates q_2, but the orthonormalization of q_5, q_8, $\ldots$, etc. in relation to q_1 can be delayed until after q_2 is passed to P_3. The steps $P_{k-s} \perp q_{i-1}$ and q_i performed in a processor can be split into three steps: compute the orthogonalization $P_{q_i} \perp q_{i-1}$ (meaning projection of q_i into the space orthogonal to q_{i-1}), normalize q_i (which is sent to the next processor), and perform the remaining orthogonalizations $P_{k-(s+1)} \perp q_{i-1}$. This reduces idling time of the following processor. This principle may also be applied to other orthonormalizations which occur before the normalization step q_i. These are illustrated as computations of the form $P_{k-s} \perp q_j$, $j < i$, that are shown before the q_i step in Figure 4. They can be split into two components: the first orthogonalization $P_{q_i} \perp q_j$ is computed before the normalization q_i, and the remaining orthogonalizations $P_{k-(s+1)} \perp q_j$ after the normalization q_i. The critical path for

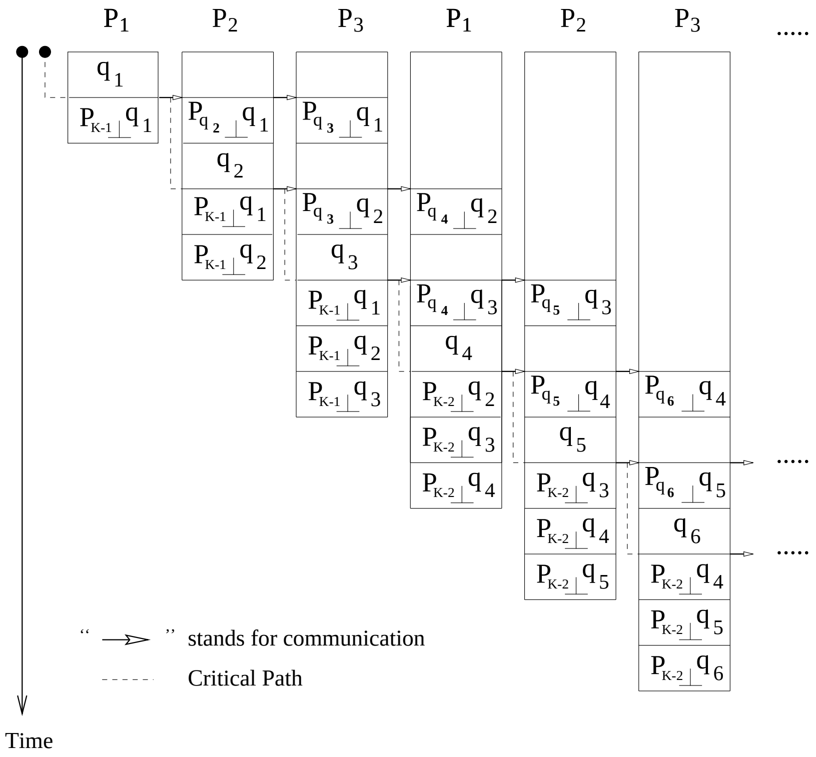


Figure 5: Cyclic partitioning: Case 2.

this algorithm (referred henceforth as case 2) is shown in Figure 5. Nevertheless, exact timings for this alternative algorithm are hard to derive theoretically. In fact, we were only able to derive theoretical timing bounds for the approach shown in Figure 5 [16]. Also, from the numerical results in Section 6, we can see that the approach of case 2 causes little improvement, the reason being that the delayed projections may cause problems of synchronization every time the pipelined algorithm gets around to re-using the processors again.

3.4 Block-cyclic partitioning

Our last study is for the pipelined parallel implementation using block-cyclic partitioning, which is a combination of block and cyclic partitionings. More precisely, in Figure 6 we show that normalization and projection are performed on a block basis (as in the block partitioning), and processors are distributed in a wrap-around manner (as in cyclic partitioning). We denote B_j^i as the j block stored on processor P_i. Recall ℓ is the length of a block and k be the wrap–around number, thus $k\ell p = n$. The computation time can be obtained

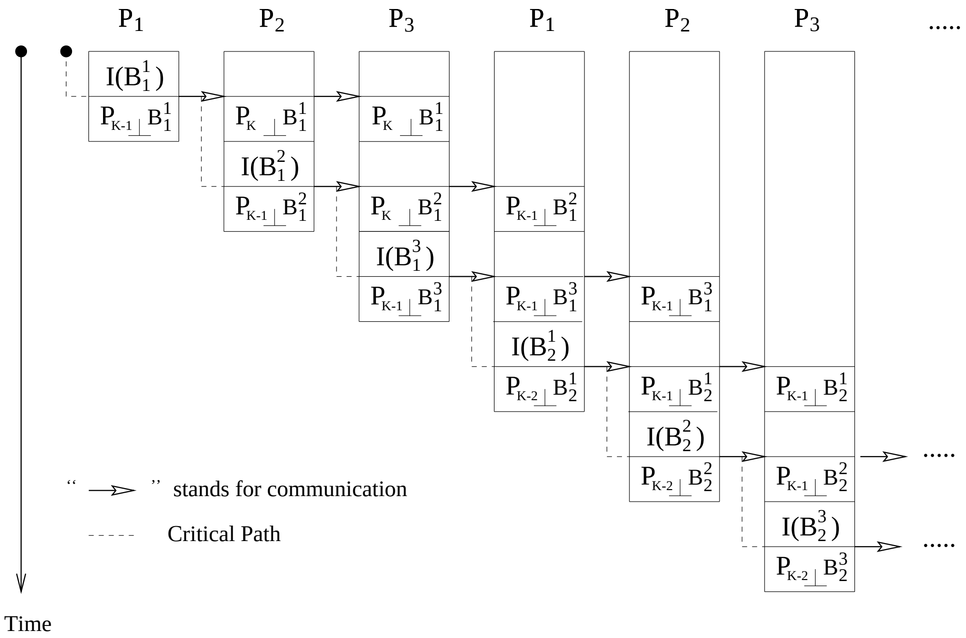


Figure 6: Block-cyclic partitioning: Case 1.

by replacing the first terms $3m$ and $4m-1$ in equation (5) with $\sum_{r=1}^{\ell}[3m+\sum_{s=r+1}^{\ell}(4m-1)]$ and $\sum_{r=1}^{\ell}\sum_{s=1}^{\ell}(4m-1)$, respectively. Thus,

$$
\begin{aligned}
T_{comp}^{block-cyclic} &= p\sum_{i=0}^{k-1}\{\sum_{r=1}^{\ell}[3m+\sum_{s=r+1}^{\ell}(4m-1)]+\sum_{j=1}^{k-i}\sum_{r=1}^{\ell}\sum_{s=1}^{\ell}(4m-1)\}-k\sum_{r=1}^{\ell}\sum_{s=1}^{\ell}(4m-1) \\
&= \frac{2mn^2}{p}+4mn\ell-\frac{n^2}{2p}+mn-n\ell+\frac{n}{2}-(4m-1)\frac{n}{p}\ell.
\end{aligned}
$$

Similarly, replacing the term m in equation (6) with $m\ell$ we can obtain the communication time

$$T_{comm}^{block-cyclic}=(2kp-3)(t_s+m\ell t_w)=(2\frac{n}{\ell}-3)(t_s+m\ell t_w).$$

As a result, the parallel run time is given by

$$
\begin{aligned}
T_p^{block-cyclic} = &\tfrac{2mn^2}{p}+4mn\ell-\tfrac{n^2}{2p}+mn-n\ell+\tfrac{n}{2}-(4m-1)\tfrac{n}{p}\ell \\
&+(2\tfrac{n}{\ell}-3)(t_s+m\ell t_w).
\end{aligned}
\tag{9}
$$

It is easily seen that if $\ell=n/p$ (thus $k=1$), which corresponds to block partitioning, then we have $T_{comp}^{block-cyclic}=T_{comp}^{block}$, and if $\ell=1$ (thus $k=n/p$), which is case 1 for cyclic partitioning, then we obtain that $T_{comp}^{block-cyclic} = T_{comp}^{cyclic}$. The cost is

$$C_p^{block-cyclic} = 2mn^2 + 4mnp\ell - \tfrac{n^2}{2} + mnp - np\ell + \tfrac{np}{2} - (4m-1)n\ell \\ + p\,(2\tfrac{n}{\ell} - 3)(t_s + m\ell t_w)\ .$$

It seems that block-cyclic partitioning achieves the low computational overhead of cyclic partitioning with less communication time. One can directly use equations (7) and (9) to compare the performance of both partitioning schemes:

$$T_p^{block-cyclic} - T_p^{cyclic} = (\ell - 1)\left((1 - \frac{1}{p})\,4mn - (1 - \frac{1}{p})n - \frac{2n}{\ell}t_s - 3mt_w\right)$$

which shows that for large values of n (consequently large values of m) the leading term $(1 - 1/p)\,4mn$ is always positive and the other terms in the second parentheses are relatively small. In other words, for large m and n cyclic will perform better than or equal to block-cyclic unless $\ell = 1$, in which case the two are the same.

As in the previous section, the principle for case 2 of cyclic partitioning can also be applied here. Projections which are not against the columns to be passed can be removed from the critical path. Figure 7 illustrates the parallel algorithm for case 2 block-cyclic partitioning. The first idle time for each processor can be reduced by separating an orthogonalization $P_{k-s} \perp B_j^{i-1}$ (before the internal orthonormalization $I(B_j^i)$) into two components. Before computing $I(B_j^i)$ we only perform the projection $P_{B_j^i} \perp B_j^{i-1}$, which orthogonalizes block B_j^i against B_j^{i-1}. After sending block B_j^i we perform the remaining projections $P_{k-(s+1)} \perp B_j^{i-1}$.

4 Row-wise Partitioning for Parallel Gram-Schmidt

Now, we focus on the parallel algorithm which partitions the matrix into blocks of rows [13]. We assume that we have p processors, each containing the matrix elements for b rows of the matrix. For simplicity we will assume that p divides m, so that $b = m/p$.

In the parallel algorithm, the norm of each vector, and its inner product with subsequent vectors, must be computed through a process of locally accumulating the partial inner products for the blocks, passing the inner products across processors, making the inner products known globally, and then doing the local modification of the vector blocks.

From the point of view of each individual processor, the ith step of the algorithm ($i = 1, ..., n$) is as follows:

DOT^i: Compute the dot products of the local block of ith column with columns $i, .., n$. This consists of $n - i + 1$ dot products of length b.

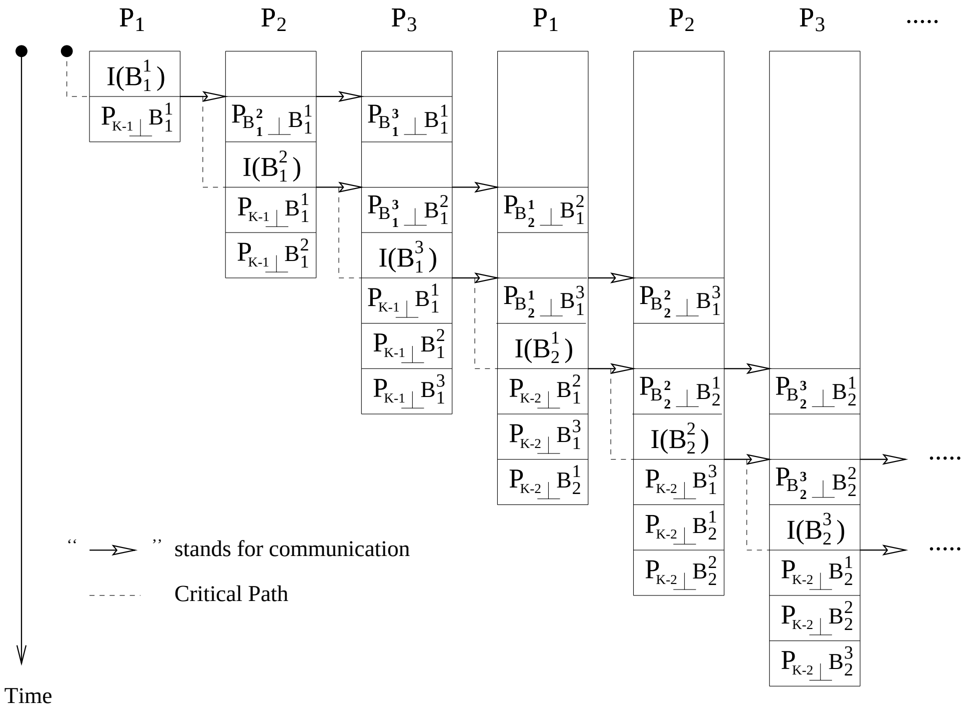


Figure 7: Block-cyclic partitioning: Case 2.

ACC^i: Wait for the $n-i+1$ dot products from a preceding processor, if there is one, and add to the local dot product accumulator.

$PASS^i$: Send the revised $n-i+1$ dot products to a successor processor, if there is one.

SCL^i: The last processor takes a square root to get the norm of the ith column, and scales the other dot products by that norm. This takes $n-i+2$ operations.

$BRDCAST^i$: Wait for the $n-i+1$ accumulated dot products from a preceding processor, if there is one, and pass them on to a successor processor, if there is one.

$UPDT^i$: Scale the local block of column i by the norm of the ith vector, and update the local block of column $i+1$ through n by subtracting from them multiples of the local block of column i to make those columns orthogonal to column i. This takes $b+2b\,(n-i)$ operations.

One can find the critical path through the algorithm which must be performed sequentially, and add the communication times. As an example, the critical path for four processors would be as indicated in Figure 8 where the processor network is a ring.

Notice that row-partitioning is architecture dependent: Both broadcasting and accumulation operations will have different performance when implemented either on a ring or a hypercube architecture. From now on we will call row partitioning implemented on a ring architecture, *row-ring* partitioning. The parallel timing for a ring architecture is

$$\begin{aligned}
T_p^{row-ring} &= \textstyle\sum_{i=1}^{n} \{DOT^i + (p-1)ACC^i + (p-1)PASS^i + SCL^i \\
&\quad + 2\,BRDCAST^i + UPDT^i\} \\
&\quad + (p-3)BRDCAST^n \\
&= 2\,\frac{m\,n^2}{p} + \frac{1}{2}\,p\,n^2 + \frac{n\,m}{p} - \frac{1}{2}\,n^2 + \frac{1}{2}\,p\,n + \frac{1}{2}\,n \\
&\quad + (\,p\,n + n + p - 3\,)\,t_s + \left(\frac{1}{2}\,p\,n^2 + \frac{1}{2}\,n^2 + \frac{1}{2}\,p\,n + \frac{1}{2}\,n + p - 3\right)\,t_w\ .
\end{aligned} \tag{10}$$

Better performance can be obtained by adopting a hypercube or any other architecture that accomplishes broadcasting and accumulation in $\log_2 p$ steps. Since in these kind of architecture the broadcast communication is a true broadcast and not merely neighbor to neighbor passaging of data, we will refer to these as the *row-brdc* implementation. In other words, the row-brdc abbreviation represents row partitioning implemented on a hypercube. The parallel timing for these architectures is given by

$$\begin{aligned}
T_p^{row-brdc} &= \sum_{i=1}^{n} \{DOT^i + (\log_2 p - 1)\,ACC^i + \log_2 p\,PASS^i + SCL^i \\
&\quad + (\log_2 p)\,BRDCAST^i + UPDT^i\}
\end{aligned}$$

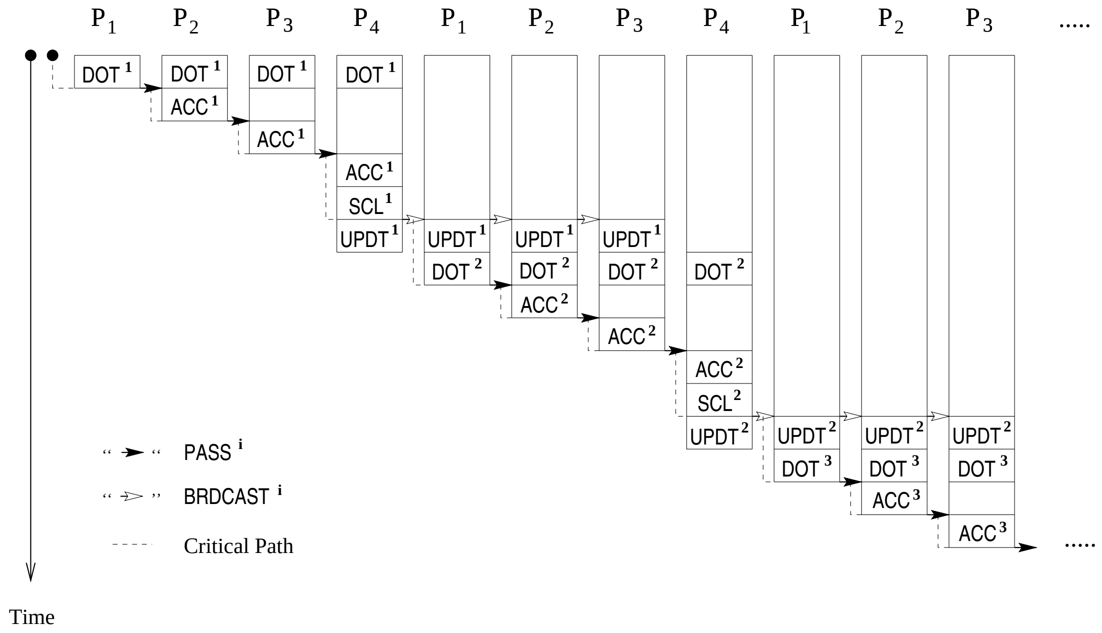


Figure 8: Row partitioning: Ring.

$$= \quad 2\,\frac{m\,n^2}{p} + \frac{n^2\,\log_2\,p}{2} + \frac{n\,m}{p} - \frac{1}{2}\,n^2 + \frac{1}{2}\,n\,\log_2\,p + \frac{1}{2}\,n$$
$$+\,2\,n\,(\log_2 p)\;\,t_s + n\,(n+1)\,(\log_2\,p)\;\,t_w\;. \qquad (11)$$

Row partitioning was designed particularly to deal with problems where the number of rows m is much larger than the number of columns n [13]. In Section 5 we compare row partitioning against column partitioning for various values of n and m.

5 Comparison Study

The row partitioning scheme aims for a well balanced workload by partitioning the matrix into blocks of rows so that each inner product computation is divided among all processors. However, such data distribution requires $\mathcal{O}(n\,p)$ messages on the critical path for a ring architecture and $\mathcal{O}(n\,\log_2 p)$ messages for a hypercube. On the other hand, the column-cyclic partitioning scheme presents only $\mathcal{O}(n)$ messages, but a given processor may be idle waiting for orthonormalized data (column or block of columns) from its previous neighbor. These contrasting characteristics create a tradeoff when making a choice between row or column partitioning. Intuitively speaking, column partitioning will perform better than row partitioning on square matrices because of the smaller amount of communication. As the number of rows m increases, the inner product computation turns out to be the predominant factor in the total timing, which makes row partitioning more attractive. Figure 9 shows two examples for the behavior of cyclic and row-brdc partitioning when $m = n$ and $m > n$. Each code was run on four processors, and time-stamps were obtained for each main component of the algorithms. Timings were normalized so that all four figures have the same size. Figures (a) and (b) present timings for an 8×8 square matrix, while figures (c) and (d) address a 320×8 matrix. Figures (a) and (c) present results for cyclic partitioning, and figures (b) and (d) are related to row-brdc partitioning. Colors purple and green are related with communication timings, and the remaining colors refer to computational timings. We also show the critical path (dotted lines) on (a) and (b). Cyclic partitioning was described by having four components: *Normalization* (red) which corresponds to the normalization of q_i, *Project* (yellow) representing projections $P \perp q_i$, *Receive* (purple) measuring the time from the point when a processor is ready to receive an orthonormalized vector q_i until the point when the communication is completed, and *Send* (green) indicating that a processor is sending a orthonormalized vector to its neighbor. Thus, we can observe the idle time for each processor on a cyclic partitioning by looking at the purple bars in figures (a) and (c): The ratio between the idle timing and the total timing remains constant as the number of rows increases. This reflects the fact that cyclic partitioning is dominated by computational work. When illustrating row-brdc, ACC and $PASS$ are considered as part of the same step, since they are implemented this way in a hypercube architecture. For row-brdc, communication timing is related to ACC, $PASS$, and $BRDCAST$. Figure (b) exemplifies how much row partitioning is communication dominated: each processor performs more communication (purple and green) than their counterparts in cyclic partitioning (a). Thus, the larger number of messages shared in the row partitioning dominates the total timing for orthogonalizing square matrices, as we can see in figure (b). As the number of rows increases, the well balanced workload and the constant value for message size allows the computational work to dominate the overall timing. In this section we compare the theoretical performance of column-cyclic partitioning against row partitioning and in Section 6 we present numerical results which validate our analysis.

One can compare the timings for column-cyclic partitioning and row-ring partitioning by taking the difference $T_p^{cyclic} - T_p^{row-ring}$ and using $k = n/p$. In such way, a negative value will indicate a better performance for the column-cyclic partitioning. Equations (7) and (10) lead to the estimate

$$\begin{aligned} T_p^{cyclic} - T_p^{row-ring} &= mn\left(5 - \frac{1}{p}\right) - \frac{1}{2}\frac{n^2\,(p^2 - p + 1)}{p} - \frac{1}{2}n\,(p+2) - \frac{n}{p}\,(4m - 1) \\ &\quad - (pn - n + p)\,t_s + \left(2mn - \frac{1}{2}n\,(n+1)\,(p+1) - 3m - p + 3\right)\,t_w \;. \end{aligned}$$

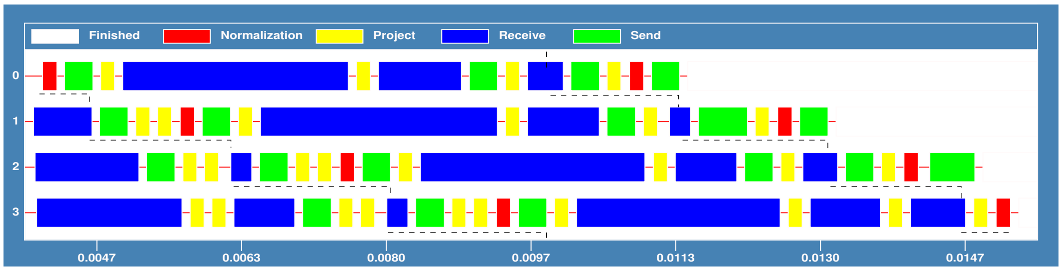
Orthonormalization of square matrices is an important application for the MGS method. The above estimate can be rewritten for square matrices by taking $m = n$, which gives

$$\begin{aligned} T_p^{cyclic} - T_p^{row-ring} &= n^2\,(\frac{-p^2 + 11p - 11}{2p}) - n\,(\frac{p}{2} + 1 - \frac{1}{p}) \\ &\quad - (pn - n + p)\,t_s - \left((p-3)\,(\frac{n^2}{2} + 1) + \frac{1}{2}n\,(p+7)\right)\,t_w \;. \end{aligned}$$

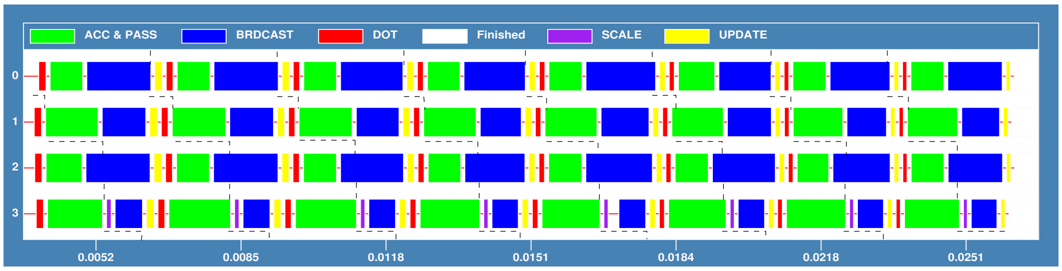
On the above equation, the terms in t_s and t_w have negative coefficients for $p \geq 3$. For the computational time, the sign of the leading term is defined by the polynomial $-p^2 + 11p - 11$. Such polynomial has roots $(\frac{1}{2}(11 - \sqrt{77}),\ \frac{1}{2}(11 + \sqrt{77})) \approx (1.11,\ 9.89)$ which guarantees that column-cyclic partitioning is faster than row-ring implementation if $p \geq 10$.

The same kind of analysis can be used to compare column-cyclic partitioning against row-brdc partitioning. Based on equations (7) and (11) we obtain

$$\begin{aligned} &T_p^{cyclic} - T_p^{row-brdc} = \\ &\quad mn\left(5 - \frac{1}{p}\right) - \frac{1}{2}n^2\left(\frac{1}{p} - 1 + \log_2 p\right) - n\left(\frac{1}{2}\log_2 p + 1\right) - \frac{n}{p}\,(4m - 1) \\ &\quad - (2n\,(\log_2 p - 1) + 3)\;t_s - (n\,((n+1)\log_2 p - 2m) + 3m)\;t_w \;. \end{aligned} \tag{12}$$

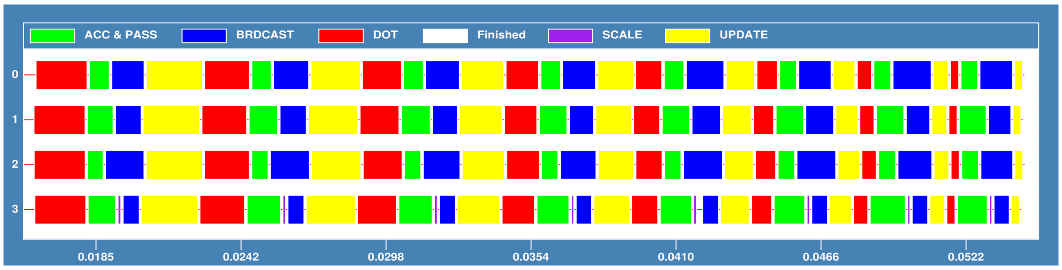


(a) Cyclic partitioning applied to an 8x8 matrix.



(b) Row-brdc partitioning applied to an 8x8 matrix.

(c) Cyclic partitioning applied to a 320x8 matrix.



(d) Row-brdc partitioning applied to a 320x8 matrix.

Figure 9: Time stamps for cyclic and row-brdc partitioning when using 4 processors.

Again, taking $m = n$, we have

$$T_p^{cyclic} - T_p^{row-brdc} = n^2 \left(\frac{(11 - \log_2 p)\, p - 11}{2\,p} \right) - n \left(\frac{\log_2 p}{2} + 1 - \frac{1}{p} \right)$$
$$- (2\,n\,(\log_2 p - 1) + 3)\; t_s - (n\,((n+1)\, log_2\, p - 2\,n) + 3\,n)\; t_w\ .$$

The terms in t_s and t_w are negative when $p \geq 4$. The sign of the leading term of the computational timing is defined by the function

$$(11 - \log_2 p)p - 11\ ,$$

which is negative for $p > 2050$, implying column-cyclic is faster than row-brdc for a large number of processors.

A better estimate can be derived from the fact that typically the time t_w spent to send or receive a floating point number is much larger than the time to perform a floating point operation: Assuming t_w equals to one flop, one can group the leading terms in computational timing and t_w as

$$\frac{(11 - \log_2 p)\,p - 11}{2p} - (\log_2 p - 2) = \frac{(15 - 3\log_2 p)\,p - 11}{2p}\ .$$

Thus, column-cyclic partitioning is faster than row-brdc partitioning when $p > 30$. Numerical comparisons for square matrices are presented in the next section.

A more general result corresponds to deriving a theoretical threshold value for the number of rows m_{max} where row partitioning in a hypercube architecture (row-brdc) becomes better than column-cyclic partitioning. Fixing the set of values n, p, t_s and t_w, one can determine the associated value m_{max} which results $T_p^{cyclic}(m_{max}) - T_p^{row-brdc}(m_{max}) = 0$. From equation (12) we obtain the upper-bound

$$m_{max} = \left(\frac{n\,(5\,p - 5)}{p} + (2\,n - 3)\, t_w \right)^{-1} \left\{ n \left(\frac{n\,(1 - p + p\,\log_2 p)}{2\,p} + \frac{\log_2 p}{2} + \frac{p-1}{p} \right) \right.$$
$$\left. + (2\,n\,(\log_2 p - 1) + 3)\; t_s + n\,(n+1)\,\log_2 p\, t_w \right\}\ . \tag{13}$$

which guarantees that column-cyclic partitioning will be faster than row-brdc for $m \leq m_{max}$. The above equation is asymptotically similar to a linear equation in n. In fact, grouping only the terms in n on equation (13) we obtain the linearized version

$$m_{max} \simeq (5 + 2\,t_w)^{-1} \left\{ n \left(\frac{1}{2}\, \frac{(1 + p\,\log_2 p - p)}{p} + \log_2 p\, t_w \right) \right.$$
$$\left. + \frac{1}{2}\, \log_2 p + \frac{p-1}{p} + 2\,\log_2 p\, t_s + \log_2 p\, t_w \right\}\ . \tag{14}$$

Our analysis indicates that the choice of partitioning scheme must be based on the ratio between number of columns and number of rows. We do not present 2-D partitioning schemes, whose theoretical analysis can be considered as a combination of *row-wise* partitioning and *column-wise* partitioning.

6 Numerical Results

The parallel codes were implemented using MPI [6] as the communication library and the computational results were obtained on the Paragon and nCUBE computers. The data structures used is an important issue to achieve good performance. More specifically, column-wise storage provides a better communication scheme for column partitioning: either a single column or a block of consecutive columns can be addressed by pointing its first element. Thus, no intermediate buffering operations are required when passing vectors across processors. In addition, we have adopted cache optimization on our implementations of row and column partitionings on the Paragon.

Note that when using column partitionings the theoretical estimates are independent of the computer architecture.[2] The MGS algorithm was run on 4, 8, 12, 16, and 24 processors to orthonormalize a matrix with $n = 2000$ columns and $m = 1056$ rows. Figure 10 compares the observed running timings against their estimates.

Three cases are shown: block partitioning (B), cyclic partitioning (C), block-cyclic partitioning (BC) with k kept equal to 2, and block-cyclic partitioning (BC) with ℓ kept equal to 2. As previously observed in the theoretical comparisons between these cases, block partitioning achieves the longest timings, while cyclic partitioning shows the best performance. It can also be demonstrated that block-cyclic partitioning approaches the minimum timings of cyclic partitioning as $\ell \to 1$. Experiments showed that there is basically no improvement by implementing case 2 for cyclic and block-cyclic partitioning, over the implementation of case 1.

Figure 11 contains plots for actual and estimated timings for row-wise partitioning. A mesh of 24 processors on Paragon was employed to orthonormalize 2000 columns with a increasing number of rows ranging from 2040 to 7200. As it was expected, the row-brdc implementation produces smaller running timings due to the fact that broadcasting and accumulation are performed in $\log_2 p$ steps.

A comparison study between row and column partitioning was developed in Section 5. Our analytical models indicate that column-cyclic partitioning is faster than row-wise partitioning for square matrices. Figure 12 shows timings obtained on the nCUBE. In Figure 12 the number of columns increases from 64 to 1024 and the data is distributed among 32 processors. Figure 13 shows

[2]The reason is because MGS parallel algorithms based on column partitionings do not rely on broadcast communication but only neighbor to neighbor communication.

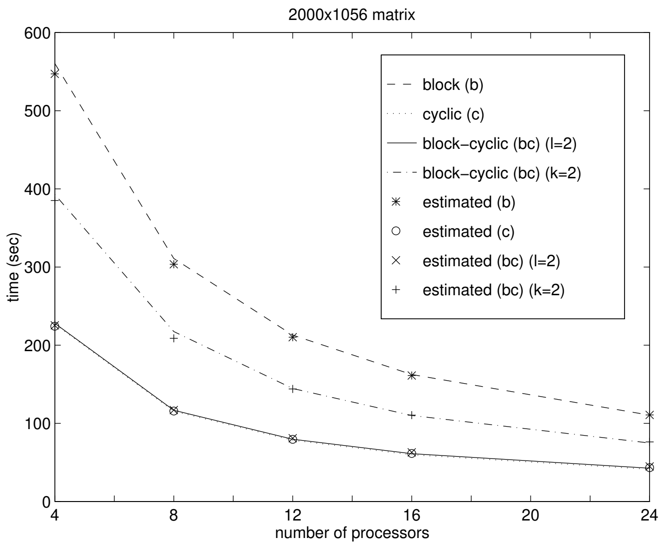


Figure 10: Estimated timings and actual running timings for block, cyclic, and block-cyclic partitionings on the Paragon.

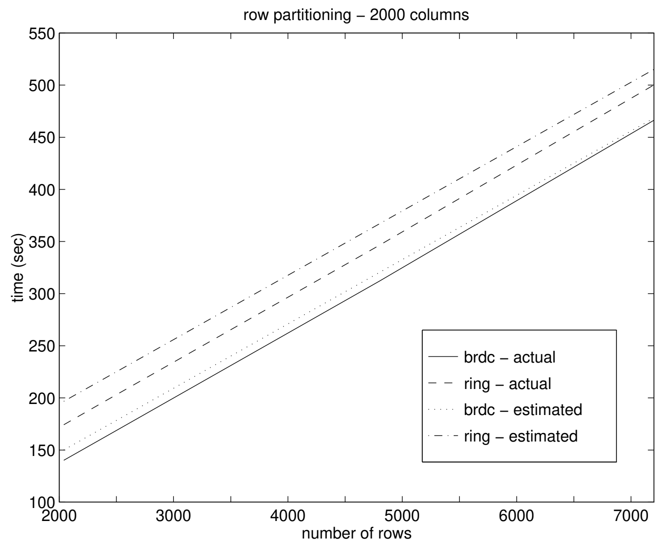


Figure 11: Timing models against actual running timings for row-ring and row-brdc partitionings using 24 processors on the Paragon.

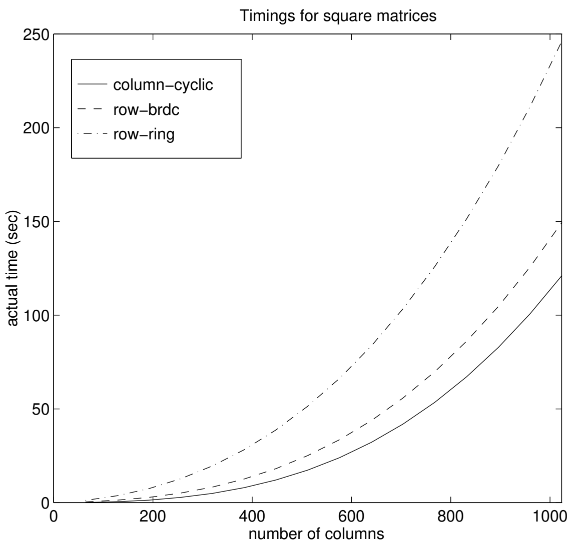


Figure 12: Timings comparing the best column partitioning (cyclic) against the row partitioning algorithms for square matrices on a 32-node hypercube.

similar results on the Paragon. Here the number of columns increases from 2520 to 4020 and the data is distributed among 60 processors. Since the number of processors is larger than 30, cyclic partitioning performs better than row-ring partitioning and (slightly) better than row-brdc partitioning, as predicted by our analysis. The differences between the curves in Figure 13 would be even more accentuated if more processors were available.

Matrices which are orthogonalized for re-started algorithms usually have $m > n$. A formula for threshold values m_{max} as a function of the number of columns n was derived in its full estimate (13) and its linearized version (14). Figure 14 presents both full and linearized estimates for matrices with number of columns n ranging from 32 to 640. The maximum number of rows m_{max} works as a safe upper-bound to guarantee that column-cyclic partitioning will perform faster than row-brdc partitioning. That is, any (n, m) pair that belongs to the region under the curve indicates that column-cyclic partitioning will be the best choice for an $m \times n$ matrix. Indeed, Figure 15 compares the actual timings for both partitioning schemes against the estimated timing for the threshold value m_{max}. Matrix sizes were based on Figure 14 so that given a number of columns n, the associated number of rows is m_{max}. As we can observe the theoretical estimates for the threshold provides a safe boundary where column-cyclic partitioning delivers a better performance than row-bdrc partitioning algorithm.

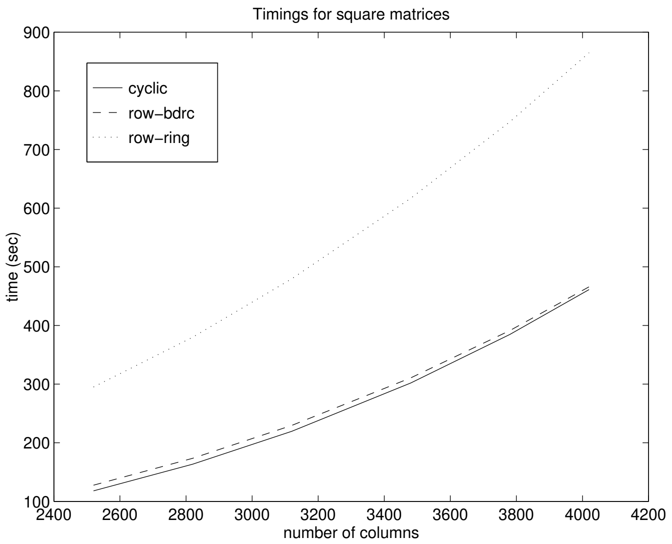


Figure 13: Timings comparing the best column partitioning (cyclic) against the row partitioning algorithms for square matrices when using 60 processors on the Paragon.

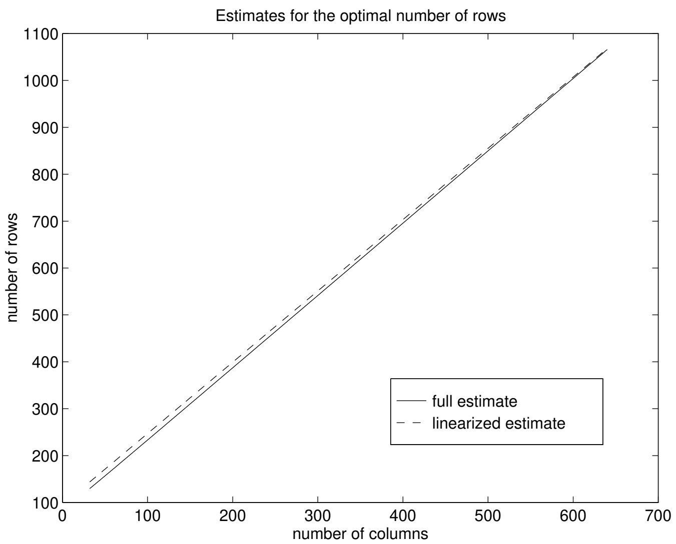


Figure 14: Threshold values m_{max} when using 32 nodes on a hypercube architecture.

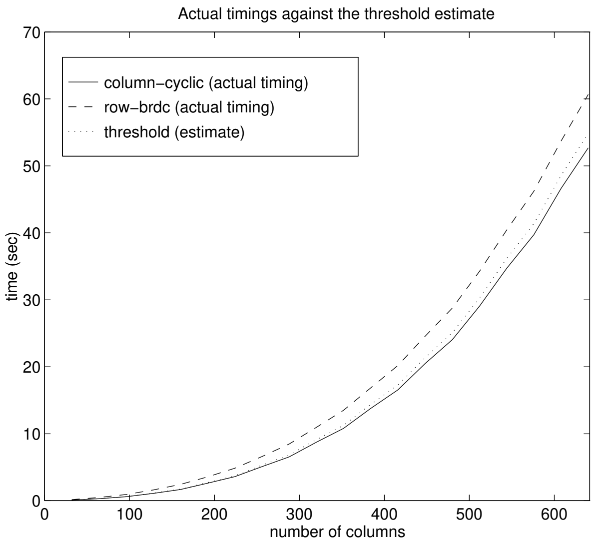


Figure 15: Column-cyclic partitioning and row-brdc partitioning compared against the estimated threshold timings for $p = 32$.

7 Conclusions

In this paper partitioning schemes for parallel pipelined MGS algorithms were analysed. We compared four kinds of column-wise partitionings: single column, block, cyclic and block-cyclic and concluded, theoretically and numerically that cyclic partitioning achieves the best performance between the four column-wise data partitionings studied. When compared to the row-wise partitioning of [13], column-wise partitioning is faster than the row-ring implementation when $p \geq 10$, and guaranteed to be faster than the row-brdc implementation when $p \geq 2050$. Furthermore, threshold values of m for which the row partitioning scheme of [13] becomes better than the cyclic partitioning are derived for the row-brdc implementation. The timings obtained from our numerical implementations corroborate our theoretical conclusions.

Acknowledgement

I would like to thank the referees whose comments improved the presentation of this research.

References

[1] Å. Björck. Solving linear least squares problems by Gram-Schmidt orthogonalization. *BIT*, (7):1–21, 1967.

[2] Å. Björck. Numerics of Gram-Schmidt orthogonalization. *Linear Algebra Appl.*, (197 & 198):297–316, 1994.

[3] L. Borges and S. Oliveira. A parallel Davidson-type algorithm for several eigenvalues. *Journal of Computational Physics*, (144):763–770, August 1998.

[4] A. Chronopoulos. Parallel iterative S-step methods for unsymmetric linear systems. *Parallel Comput.*, 22(5):623–641, 1996.

[5] Y. Deng. Some applications of pipelining techniques in parallel scientific computing. Master's thesis, Texas A&M University, 1996.

[6] J. J. Dongarra, S. W. Otto, and D. Walker M. Snir. A message passing standard for MPP and workstations. *Comm. ACM*, 39:84–90, 1996.

[7] G. I. Fann and R. J. Littlefield. Parallel inverse iteration with reorthogonalization. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume 1, pages 409–413, March 1993.

[8] W. Hoffmann. Iterative algorithms for Gram-Schmidt orthogonalization. *Computing*, 41(4):335–348, 1989.

[9] G. Fann I. Dhillon and B. Parllet. Application of a new algorithm for the symmetric eigenproblem to computational quantum chemistry. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. Society for Computer Simulation, March 1997. CD-ROM.

[10] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings, New York, 1994.

[11] K. Maschhoff and D. Sorensen. A portable implementation of ARPACK for distributed memory parallel architectures. In *Proceedings of Copper Mountain Conference on Iterative Methods*, April 9–13 1996.

[12] H. De Meyer, T. Niyokindi, and G. Vanden Berghe. The implementation of parallel Gram-Schmidt orthogonalisation algorithms on a ring of transputers. *Comput. Math. Applic.*, 25(1):65–72, 1993.

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

[14] S. Oliveira and T. Soma. New partitioning schemes for parallel modified Gram-Schmidt orthogonalization. In Feipei Lai, Bruce Maggs, and Frank Hsu, editors, *Proceedings of the 1997 International Symposium on Parallel Architectures, algorithms and Networks (ISPAN97)*, pages 233–239. IEEE Computer Society, December 1997.

[15] E. Schmidt. Über die Auflösung linearer Gleichungen mit unendlich vielen Unbekannten. *Rend. Circ. Mat. Palermo*, (25):53–77, 1908.

[16] T. Soma. Parallel implementations of modified Gram-Schmidt orthogonalization. Master's thesis, Texas A&M University, 1998.

[17] S. Thomas and R. Zahar. Efficient orthogonalization in the M-norm. In *Proceedings of the Twentieth Manitoba Conference on Numerical Mathematics and Computing*, volume 80 of *Congr. Numer.*, pages 23–32, 1991.

[18] L. N. Trefethen and D. Bau, III. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

[19] L. Waring and M. Clint. Parallel Gram-Schmidt orthonormalisation on a network of transputers. *Parallel Comput.*, 17(9):1043–1050, 1991.

[20] E. Zapata, J. Lamas, F. Rivera, and O. Plata. Modified Gram-Schmidt QR factorization on hypercube SIMD computers. *J. Parallel Distrib. Comput.*, 12(1):60–69, 1991.