Parallelization of Cache Efficient BLAS 2.5 Operator GEMVT

by

MADHAN PREMKUMAR

Bachelor of Engineering
Computer Science and Engineering
Government College of Technology
Coimbatore, India
2000

A thesis submitted to
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science

in

Computer Science

Melbourne, Florida
May 2005
We the undersigned committee hereby approve the attached thesis

Parallelization of Cache Efficient BLAS 2.5 Operator GEMVT

by
Madhan Premkumar

Dr. Charles Fulton  
Professor, Mathematical Sciences  
Major Advisor

Dr. Gary Zarillo  
Professor, Oceanography

Dr. Jim Jones  
Associate Professor, Mathematical Sciences

Dr. William Shoaff  
Associate Professor and Head, Computer Sciences
Abstract

Title: Parallelization of Cache Efficient BLAS 2.5 Operator GEMVT
Author: Madhan Premkumar
Major Advisor: Charles Fulton, Ph.D.

In this thesis, two parallel versions of the cache efficient BLAS 2.5 GEMVT routine are presented. GEMVT is a routine that performs two coupled matrix-vector multiplications. The serial version of GEMVT is due to Gary W. Howell and was accepted into the BLAS 2000 standard by the BLAST Forum (Basic Linear Algebra Subroutines Technical Forum) in November 2000 [1]. The first cache efficient parallel algorithm makes use of the block cyclic data distribution and the second algorithm makes use of the column cyclic data distribution. Both the algorithms developed here are based on making two calls to BLAS (Basic Linear Algebra Subroutines) routine DGEMV (Double precision General Matrix Vector multiplication) from the ATLAS library.
## Contents

1 Introduction .......................... 1
   1.1 Cache Efficient Serial GEMVT .................. 3
   1.2 Process Grid ............................... 3
   1.3 Parallel BLAS Cache Efficient DGEMVT ......... 4
      1.3.1 Block cyclic data distribution ................. 4
      1.3.2 Column cyclic data distribution: ................. 6

2 Parallel Cache Efficient GEMVT using Block Cyclic data distribution .... 7
   2.1 Introduction ............................... 7
   2.2 Process Grid ............................... 7
   2.3 “Block Cyclic” data distribution using a $8 \times 8$ matrix .......... 8
      2.3.1 Data layout of $8 \times 8$ matrix on each process: ........... 10
      2.3.2 Data layout of $8 \times 1$ input vector Y on each process: .... 11
      2.3.3 Data layout of $1 \times 8$ input vector Z on each process: .... 13
   2.4 Computations and Communications ..................... 14
      2.4.1 Pseudo-code for a general matrix ................. 22
      2.4.2 Source code of the subroutine PDGEMVT .............. 24
   2.5 Driver Program ............................. 27
      2.5.1 Auxiliary Routines .......................... 27
      2.5.2 Driver Program ............................ 30
   2.6 Description of subroutines on each process ............ 38

3 Parallel Cache Efficient GEMVT using Column Cyclic data distribution .... 50
   3.1 Introduction ............................... 50
List of Tables

2.1 DESC Parameters ................................................................. 29
2.1 DESC Parameters (Continued) .................................................. 30
2.2 Input and output values to and from the subroutine NUMROC on each
process for a $8 \times 8$ matrix ......................................................... 33
2.3 DESC Parameters for the $8 \times 8$ matrix $A$, $8 \times 1$ vectors $Y$ and $W$, and
$1 \times 8$ vectors $Z$ and $X$ .......................................................... 34
2.4 Values of $IPA$, $IPY$, $IPZ$, $IPX$, $IPW$ and $IPL$ and values of $LLD_A$,
$LLD_Y$, $LLD_Z$, $LLD_X$, $LLD_W$ and $LLD_L$ on each process .... 36
3.1 Input and output values to and from the subroutine NUMROC on each
process for a $8 \times 8$ matrix .......................................................... 67
3.2 DESC Parameters for the $8 \times 8$ matrix $A$, $8 \times 1$ vectors $Y$ and $W$, and
$1 \times 8$ vectors $Z$ and $X$ .......................................................... 68
3.3 Values of $IPA$, $IPY$, $IPZ$, $IPX$, $IPW$ and $IPL$ and values of $LLD_A$,
$LLD_Y$, $LLD_Z$, $LLD_X$, $LLD_W$ and $LLD_L$ on each process .... 69
4.1 Average Timings of Cache-Efficient Block Cyclic PDGEMVT with dif-
f erent block sizes $k$ Compared with Average Timings of Non-Cache
Efficient PDGEMVT (Average of 3 runs) .......................................... 83
4.2 Percentage of $Bcache$ filled for different block sizes, $K$ ............... 84
4.3 Total MFlops of Cache-Efficient Block Cyclic PDGEMVT for different
values of $k$ and Total MFlops of Non-Cache Efficient PDGEMVT (Each
rate is the average of 3 runs). ...................................................... 85
4.4 Average Timings of Cache-Efficient Column Cyclic PDGEMVT with
different block sizes $k$ Compared with Average Timings of Non-Cache
Efficient PDGEMVT (Average of 3 runs) .......................................... 88
4.5 Percentage of $Bcache$ filled for different block size $k$ on each processor. 
\[ M = \sqrt{P \cdot 7200} \] .................................................. 89
4.5 Percentage of $Bcache$ filled for different block size $k$ on each processor. 
\[ (M = \sqrt{P \cdot 7200}) \text{ (Continued)} \] ........................................... 90
4.6 Total MFlops of Cache-Efficient Column Cyclic PDGEMVT and Total 
MFlops of Non-Cache Efficient PDGEMVT (Each rate is the average 
of 3 runs). ............................................................... 90
4.7 Theoretical Communication times for column cyclic PDGEMVT for 
matrices with $m = \sqrt{P \cdot 7200}$ rows on each processor. ............... 94
4.8 Experimental Communication times for column cyclic PDGEMVT for 
matrices with $m = \sqrt{P \cdot 7200}$ rows on each processor. ............... 95
4.9 Theoretical Communication times for block cyclic PDGEMVT for ma-
trices with $m = 7200$ rows on each processor. ................................. 97
4.9 Theoretical Communication times for block cyclic PDGEMVT for ma-
trices with $m = 7200$ rows on each processor (Continued). ............... 98
4.10 Experimental Communication times for block cyclic PDGEMVT for 
matrices with $m = 7200$ rows on each processor. ............................... 98
4.10 Experimental Communication times for block cyclic PDGEMVT for 
matrices with $m = 7200$ rows on each processor (Continued). ............ 99
4.11 Scaled Speedups for Block-cyclic and Column-cyclic Cache-Efficient 
GEMVT ........................................................................... 100
4.12 Scaled Speedups for Non-Cache Efficient GEMVT ............................ 100
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Six processes mapped onto a $2 \times 3$ process grid</td>
</tr>
<tr>
<td>2.1</td>
<td>Four processes mapped onto a $2 \times 2$ process grid</td>
</tr>
<tr>
<td>2.2</td>
<td>Process coordinates in a $2 \times 2$ process grid</td>
</tr>
<tr>
<td>2.3</td>
<td>$8 \times 8$ matrix partitioned into $2 \times 2$ blocks in block cyclic fashion</td>
</tr>
<tr>
<td>2.4</td>
<td>Process coordinates for $8 \times 8$ matrix, partitioned into $2 \times 2$ blocks</td>
</tr>
<tr>
<td>2.5</td>
<td>Input $Y$ vector partitioned into $2 \times 2$ blocks and the corresponding process coordinates for each block</td>
</tr>
<tr>
<td>2.6</td>
<td>Input $Z$ vector partitioned into $2 \times 2$ blocks and the corresponding process coordinates for each block</td>
</tr>
<tr>
<td>2.7</td>
<td>Column blocking with $k = 2$ on process $0 = (0,0)$</td>
</tr>
<tr>
<td>2.8</td>
<td>An $8 \times 8$ example matrix partitioned into $2 \times 2$ blocks in block cyclic fashion</td>
</tr>
<tr>
<td>2.9</td>
<td>An example input $Y$ vector partitioned using $2 \times 2$ blocks and the corresponding process coordinates for each block</td>
</tr>
<tr>
<td>2.10</td>
<td>An example input $Z$ vector partitioned using $2 \times 2$ block and the corresponding process coordinates for each block</td>
</tr>
<tr>
<td>3.1</td>
<td>Four processes mapped onto a $1 \times 2$ process grid</td>
</tr>
<tr>
<td>3.2</td>
<td>Process coordinates in a $1 \times 2$ process grid</td>
</tr>
<tr>
<td>3.3</td>
<td>$8 \times 8$ matrix partitioned into $8 \times 2$ blocks in column cyclic fashion</td>
</tr>
<tr>
<td>3.4</td>
<td>Process coordinates for $8 \times 8$ matrix, partitioned by $8 \times 2$ blocks</td>
</tr>
<tr>
<td>3.5</td>
<td>Input $Y$ vector distributed to process $0 = (0,0)$</td>
</tr>
<tr>
<td>3.6</td>
<td>Input $Z$ vector partitioned into $8 \times 2$ blocks and the corresponding process coordinates for each block</td>
</tr>
<tr>
<td>3.7</td>
<td>Column blocking with $k = 2$ on process $0 = (0,0)$</td>
</tr>
</tbody>
</table>
3.8 An 8 × 8 matrix partitioned into 8 × 2 blocks in column cyclic fashion. 72
3.9 A 8 × 1 input Y vector distributed to process 0 = (0,0) . . . . . . . . 73
3.10 An example input Z vector partitioned using 8 × 2 blocks and the corresponding process coordinates for each block . . . . . . . . . . . . 74
4.1 Total MFlops of Cache-Efficient Block Cyclic PDGEMVT for different values of k and Total MFlops of Non-Cache Efficient PDGEMVT . . . 86
4.2 Total MFlops of Cache-Efficient Column Cyclic PDGEMVT for different values of k and Total MFlops of Non-Cache Efficient PDGEMVT . 91
Acknowledgments

I would like to deeply thank my advisor Dr. Charles Fulton for his constant guidance, support and encouragement for my graduate study at Florida Institute of Technology. He has always provided valuable suggestions and areas of improvements to my research work and always guided me in the right direction.

I would also like to show my appreciation to Dr. Gary Howell for providing thoughtful ideas and valuable suggestions during the implementation and communication of the developed routines. I extend my appreciation for the helpful discussions and useful suggestions from Dr. Sven Hammarling. I am also grateful to all my committee members Dr. Gary Zarillo, Dr. William D. Shoaff and Dr. Jim Jones for their valuable suggestions and comments on my thesis. I would also thank Dale Means for helping me to resolve all the technical challenges I had with Beowulf machine during my research work.

I take this opportunity to deeply thank and love my family for their constant support and encouragement even though they were not present here. Without them, I would not have been in this place. I would like to thank my beloved mentors/friends Sumesh and Deepa for their moral support and believing me that I can succeed in anything I do. I would also like to thank my friend Tathagata Samanta for listening and encouraging my research work.

I acknowledge support from National Science Foundation research Grant EIA–0103642, “Cache-Efficient and Parallel Householder Bidiagonalization”, to Florida Institute of Technology, for support as a Graduate Research Assistant. The computations were performed on the National Science Foundation Terascale Computing System at the Pittsburgh Supercomputing Center. This was done under Grant #ASC50006P to Florida Institute of Technology for time on the Lemieux cluster.
Chapter 1

Introduction

In [1] the new BLAS 2.5 Operator \_GEMVT was introduced to accomplish the following two matrix-vector multiplications:

\[
\begin{align*}
  x &= \beta \ast A' \ast y + z \\
  w &= \alpha \ast A \ast x
\end{align*}
\]  

(1.1)
(1.2)

where \( A' \) denotes the transpose of \( A \). Here \( A \) is an arbitrary \( m \times n \) matrix.

For the serial computation of the above two matrix-vector multiplications it is easiest to make two successive calls to the BLAS 2 Operator \_GEMV which accomplishes the operation

\[
x = \alpha \ast A \ast y + \beta \ast x,
\]

or the similar operation in which \( A \) is replaced by \( A' \), the transpose of \( A \). Here \( A \) is an \( m \times n \) input matrix, \( y \) is an input \( m - vector \) and \( x \) is an input \( n - vector \), and the output overwrites \( x \).

That is, we utilize \_GEMV two times as follows:

\[
\begin{align*}
  x &= \beta \ast A' \ast y + z & (first \ \_GEMV \ call) \\
  w &= \alpha \ast A \ast x & (second \ \_GEMV \ call)
\end{align*}
\]

Here the first operation multiplies the \( n \times m \) transpose matrix \( A' \) by the \( m \)-vector \( y \), multiplies by the scalar \( \beta \) and then adds the \( n \)-vector \( z \). The second operation uses the output \( n \)-vector \( x \), multiplies on the left by the \( m \times n \) matrix \( A \), and then multiplies by the scalar \( \alpha \).

The BLAS 2.5 operator \_GEMVT has recently been utilized by G.Howell, et.al [5] to improve cache-efficiency in a new householder bidiagonalization algorithm. Sig-
significant speed-ups over the current LAPACK routine DGEBRD are obtained since
the two matrix-vector multiplications in (1.1) and (1.2) can be done simultaneously
by blocking the matrix so as to perform both operations in one pass of each block
through the $L_2$ cache. A description of the current LAPACK routine DGEBRD is
given in [8].

The present thesis deals with two parallel versions of cache efficient _GEMVT
and performance analysis of them. It is anticipated that this analysis will be helpful
in the development of a new parallel DGEBRD routine similar to PDGEBRD.

For general information on the LAPACK and ScaLAPACK projects we refer to
the LAPACK user’s guide [2], and the ScaLAPACK user’s guide [3].

For basic linear algebra routines the software package LAPACK was written for
shared memory machines to make efficient use of the hierarchical memory by maxi-
mizing data reuse, that is, by trying to avoid reloading cache memory too frequently.
This was done by casting linear algebra routines in a block oriented manner so as to
maximize the use of level 3 BLAS operations as much as possible.

Similarly, the software package ScaLAPACK (Scalable LAPACK) was written
to port LAPACK package to distributed memory parallel machines. This was done
with a view to maximizing scalability of the algorithms of the LAPACK library,
maintaining good load balancing and minimizing communication overhead.

Algorithms on distributed memory machines are regarded as “scalable” if they
continue to perform efficiently as the number of processors increases while keeping the
amount of work done on each processor constant (fixed “granularity”). These design
considerations resulted in the decision to employ the block cyclic data distribution
for a distributed matrix in all ScaLAPACK library routines. In Chapter 4, we discuss
the scalability of the new cache efficient parallel _GEMVT algorithm.

For further information on the basic structure and design issues in the ScaLA-
PACK library we refer to the LAPACK Working Notes [8], [9], [11], and [12].
1.1 Cache Efficient Serial GEMVT

The two matrix-vector multiplications described in the following section were implemented in [5] and [7] using a column blocking on the input matrix $A$:

\[
\begin{bmatrix}
A_1 & A_2 & \ldots & A_j
\end{bmatrix}
\]

The GEMVT is performed as follows:

For $i = 1$ to $j$

\[
x_i \leftarrow \alpha \ast A_i' \ast y_i + z_i \quad \text{(first GEMV call)}
\]

\[
w \leftarrow \alpha \ast A_i \ast x_i + w \quad \text{(second GEMV call)}
\]

End for

where $z$ and $y$ are the input vectors and $x$ and $w$ are the output vectors.

1.2 Process Grid

\[
\begin{array}{ccc}
0 & 1 & 2 \\
0 & 0 & 1 & 2 \\
1 & 3 & 4 & 5 \\
\end{array}
\]

Figure 1.1: Six processes mapped onto a $2 \times 3$ process grid

The $P$ processes of an abstract parallel computer are often represented as a one-dimensional linear array of processors labelled $0, 1, 2, \ldots, P - 1$. In our approach, we conveniently map this one-dimensional array of processes into a two-dimensional rectangular grid, or **process grid**. This grid will have $P_r$ process rows and $P_c$ process columns, where $P_r \ast P_c = P$. A process can now be referenced by its row and column.
coordinates, \((p_r, p_c)\), within the grid (where \(0 \leq p_r < P_r\), and \(0 \leq p_c < P_c\)). An example of such a mapping for \(P_r = 2\) and \(P_c = 3\) is shown in Figure 1.1.

The global input matrix and vector in all parallel algorithms of the ScaLAPACK library are mapped onto process grid to distribute the matrix and vector elements to different processes. In Figure 1.1, the processes are mapped to the process grid by using **row-major order**, that is, the numbering of the processes increases sequentially across each row. The BLACS routine BLACS_GRIDINIT is used in the driver program to perform this task of mapping the processes to the process grid.

### 1.3 Parallel BLAS Cache Efficient DGEMVT

Current computer architectures store data in a hierarchy of distances from the computational registers. Data in a small number of registers can be used for computations in the current clock cycle. Data in \(L_2\) cache memory is available in at most a few clock cycles. Accessing data in main memory requires several dozen clock cycles and is constrained by bus bandwidth. Matrices larger than a few hundred square are typically too large to fit in \(L_2\) cache memory and must be stored in main memory (RAM). In computers with cache architectures, halving data transfer substantially reduces computational time. Our new cache efficient parallel algorithms make use of this \(L_2\) cache memory to improve its performance. Two different parallel algorithms were implemented depending on the distribution of the input matrix and vector on different processes. These algorithms are discussed below for both the block cyclic and column cyclic data distribution.

#### 1.3.1 Block cyclic data distribution

Since the parallelization of \_GEMV in ScaLAPACK is done using the block cyclic data distribution, the parts of the distributed matrix \(A\) on each processor are multiplied on each processor by the appropriate parts of the distributed \(Y\) and \(X\) vectors that reside on each processor. The **BLAS (Basic Linear Algebra Subroutines)** routine \(P\_GEMV\) therefore does the following operation

\[ sub(X) = \alpha \ast sub(A) \ast sub(Y) + \beta \ast sub(X) \quad (1.3) \]
on each processor where sub(A), sub(Y) and sub(X) are the parts of the A matrix, Y and X vectors which reside on each processor. Here GEMV can also do the above operation with A\', the transpose of A, replacing A.

To accomplish the two matrix-vector multiplications in GEMVT in parallel we first distribute the A matrix to the processors using the block cyclic data distribution and then distribute the Y and X vectors to the processes using the block cyclic data distribution. Then we make call to a newly developed routine P.GEMVT which performs two GEMV calls on each column block of the distributed matrix on each process; here the ‘column blocks’ of sub(A) are generated by grouping k columns of sub(A) in each block, where k is chosen so that each column block fills the available L2 cache. This has the effect of halving data transfer between main memory and L2 cache on each process. If b is the total number of column blocks for sub(A) then ideally k \times b is equal to the total number of columns of sub(A).

The two calls to GEMV which make up the routine P.GEMVT can thus be described as follows:

1. for i = 1 to b, do in parallel on each matrix sub(A):
   
   (a) \( (sub(X))_i = \alpha \times (sub(A'))_i \times (sub(Y))_i + \beta \times (sub(Z))_i \) [1st GEMV call]

   (b) Sum the partial \( X_i \) vector among all processors in each column of the process grid to generate the complete \( X_i \) vectors.

   (c) \( sub(W) = \alpha \times (sub(A))_i \times (sub(X))_i + \beta \times sub(W) \) [2nd GEMV call]

   end for

2. Sum the partial W vector from all processes in each row of the process grid to generate final W vector

   Here the parts of the n-vector X which are computed by each process in the first GEMV is the partial result. So after the first GEMV call, the partial \( X_i \) vectors are added up among all the processes in each column of the process grid to generate the complete \( X_i \) vector. This vector is then used in the second GEMV call to generate the final W vector. In the next section we give some simple examples to illustrate
how the above described cache efficient _GEMVT code accomplishes these tasks in parallel.

1.3.2 Column cyclic data distribution:

In this version of the parallel code we distribute A Matrix, Y vector and X vector using the Column cyclic data distribution and then we make a call to a newly developed routine P_GEMVT which performs two _GEMV calls on each column block of the distributed matrix on each process; as before the ‘column blocks’ of sub(A) are generated by grouping $k$ columns of sub(A) in each block where $k$ is chosen so that each column block fills the available $L_2$ cache. Letting $b$ be the total number of column blocks for sub(A) we again assume $k \times b$ equals the total number of columns in sub(A).

The two calls to _GEMV which make up the routine P_GEMVT can thus be described as follows:

1. for $i = 1$ to $b$, do in parallel on each matrix sub(A):
   
   (a) $(sub(X))_i = \alpha * (sub(A))_i * Y + \beta * (sub(Z))_i$ \hspace{1cm} [1st _GEMV call]
   
   (b) $sub(W) = \alpha * (sub(A))_i * (sub(X))_i + \beta * sub(W)$ \hspace{1cm} [2nd _GEMV call]

   end for

2. Sum the partial W vectors from all processes in each row of the process grid to generate final W vector

Here the parts of the n-vector $X$ which are computed by each process in the first _GEMV call remain distributed as they are, so in column cyclic algorithm we need no communication to perform the second _GEMV call on that column block of distributed matrix A. Simple examples are provided to describe this cache efficient algorithm in the following section.
Chapter 2

Parallel Cache Efficient GEMVT using Block Cyclic data distribution

2.1 Introduction

This version of parallel GEMVT uses BLACS and BLAS routines to perform the GEVMT operations in parallel. The driver program pdgemvtdriver.F makes a call to a newly developed routine called PDGEMVT, which makes two calls to the BLAS routine DGEMV from the ATLAS library. The driver program uses the usual “Block Cyclic” data distribution to distribute the input matrix and input vector.

2.2 Process Grid

![Process Grid Diagram]

Figure 2.1: Four processes mapped onto a 2 × 2 process grid

In this version, a two-dimensional process grid is used to distribute the input matrix
and vector in block cyclic fashion. This grid will have $P_r$ process rows and $P_c$ process columns, where $P_r \times P_c = P$, the total number of processes. The $2 \times 2$ process grid ($P_r = 2$ and $P_c = 2$), shown in Figure 2.1, is used to illustrate the parallel GEMVT using a block cyclic data distribution of a $8 \times 8$ matrix in next section.

### 2.3 “Block Cyclic” data distribution using a $8 \times 8$ matrix

The block cyclic data distribution depends on the number of processes ($P_r \times P_c$) in the $P_r \times P_c$ process grid and the block size $MB \times NB$. The block cyclic data distribution using a $8 \times 8$ A matrix and Y and Z vectors of size 8 mapped onto a $2 \times 2$ process grid (shown in Figure 2.2) is illustrated in this section.

Let $MP$ and $NQ$ be the number of rows and columns of the distributed A matrix on each process. Thus, $MP \times NQ$ is the size of the matrix on each process. For a $8 \times 8$ matrix mapped onto a $2 \times 2$ process grid, $MP = 4$ and $NQ = 4$.

Let $A$ be a $8 \times 8$ matrix,

$$A = \begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\
a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\
a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\
a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\
a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\
a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\
a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\
a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88}
\end{pmatrix}.$$  

The coordinates of each process in the $2 \times 2$ process grid shown in Figure 2.2 are:

- Process 0: (0,0)
- Process 1: (0,1)
- Process 2: (1,0)
- Process 3: (1,1)
Let us partition the $8 \times 8$ matrix $A$ into $2 \times 2$ blocks, mapped onto a $2 \times 2$ process grid. The partitioned matrix is shown in the Figure 2.3

The mapping of the $2 \times 2$ blocks onto the process grid is “cyclic” in both the row and column direction, that is, every other $2 \times 2$ block across each row goes to successive processes in the process grid ($P_r$ alternating from 0 to 1) and every other $2 \times 2$ block across each column goes to successive processes in the process grid ($P_c$ alternating from 0 to 1). The process coordinates of the $8 \times 8$ matrix using $2 \times 2$ blocks in block cyclic data distribution is shown in the Figure 2.4

Thus:

- Process 0 gets the $2 \times 2$ matrices which arise from the intersections of the 1st and 3rd row-blocks with 1st and 3rd column-blocks.
\[ P_r \downarrow \quad P_c \Rightarrow \quad \begin{array}{cccc}
0 & 1 & 0 & 1 \\
\hline
(0,0) & (0,1) & (0,0) & (0,1) \\
(1,0) & (1,1) & (1,0) & (1,1) \\
\hline
(0,0) & (0,1) & (0,0) & (0,1) \\
(1,0) & (1,1) & (1,0) & (1,1) \\
\end{array} \]

Figure 2.4: Process coordinates for $8 \times 8$ matrix, partitioned into $2 \times 2$ blocks

- Process 1 gets the intersections of the $2^{nd}$ and $4^{th}$ row-blocks with the $1^{st}$ and $3^{rd}$ column-blocks.
- Process 2 gets the intersections of the $1^{st}$ and $3^{rd}$ row-blocks with the $2^{nd}$ and $4^{th}$ column-blocks.
- Process 3 gets the intersection of the $2^{nd}$ and $4^{th}$ row-blocks with the $2^{nd}$ and $4^{th}$ column-blocks.

Thus, the number of rows ($MP$) and number of columns ($NQ$) on each process are 4 and 4 respectively.

### 2.3.1 Data layout of $8 \times 8$ matrix on each process:

**Process 0:** $(0,0)$ in process grid

\[
A^{(0)} = \begin{bmatrix} a_{11} & a_{12} & a_{15} & a_{16} \\
a_{21} & a_{22} & a_{25} & a_{26} \\
a_{51} & a_{52} & a_{55} & a_{56} \\
a_{61} & a_{62} & a_{65} & a_{66} \end{bmatrix} \quad 4 \times 4
\]

$A^{(0)}$ is the $4 \times 4$ part of the $A$ matrix which is distributed to process $0 = (0,0)$.\]
Process 1: (0,1) in process grid

\[
A^{(1)} = \begin{bmatrix}
  a_{13} & a_{14} & a_{17} & a_{18} \\
  a_{23} & a_{24} & a_{27} & a_{28} \\
  a_{53} & a_{54} & a_{57} & a_{58} \\
  a_{63} & a_{64} & a_{67} & a_{68}
\end{bmatrix}
4 \times 4
\]

\(A^{(1)}\) is the 4 \times 4 part of the A matrix which is distributed to process 1 = (0,1).

Process 2: (1,0) in process grid

\[
A^{(2)} = \begin{bmatrix}
  a_{31} & a_{32} & a_{35} & a_{36} \\
  a_{41} & a_{42} & a_{45} & a_{46} \\
  a_{71} & a_{72} & a_{75} & a_{76} \\
  a_{81} & a_{82} & a_{85} & a_{86}
\end{bmatrix}
4 \times 4
\]

\(A^{(2)}\) is the 4 \times 4 part of the A matrix which is distributed to process 2 = (1,0).

Process 3: (1,1) in process grid

\[
A^{(3)} = \begin{bmatrix}
  a_{33} & a_{34} & a_{37} & a_{38} \\
  a_{43} & a_{44} & a_{47} & a_{48} \\
  a_{73} & a_{74} & a_{77} & a_{78} \\
  a_{83} & a_{84} & a_{87} & a_{88}
\end{bmatrix}
4 \times 4
\]

\(A^{(3)}\) is the 4 \times 4 part of the A matrix which is distributed to process 3 = (1,1).

2.3.2 Data layout of 8 \times 1 input vector Y on each process:

Let Y be a vector of size 8, partitioned into 2 \times 2 blocks, mapped onto a 2 \times 2 process grid. The Figure 2.5 shows the partitioned vector by 2 \times 2 blocks and the process coordinates in process grid for each block.

Since there is only one column, the Y vector is placed 2 components at a time on processes (0,0), (1,0), (0,0) and (1,0) alternately, that is in a row-cyclic manner.
\[
\begin{align*}
P_c & \Rightarrow 0 & P_c & \Rightarrow 0 \\
\begin{array}{c}
0 \\
1 \\
0 \\
1 \\
\end{array} & \begin{pmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
y_5 \\
y_6 \\
y_7 \\
y_8 \\
\end{pmatrix} & \begin{array}{c}
0 \\
1 \\
0 \\
1 \\
\end{array} & \begin{pmatrix}
(0,0) \\
(1,0) \\
(0,0) \\
(1,0) \\
\end{pmatrix}
\end{align*}
\]

Figure 2.5: Input \( Y \) vector partitioned into \( 2 \times 2 \) blocks and the corresponding process coordinates for each block

Thus:

- Process 0 gets the 1\(^{st}\) and 3\(^{rd}\) blocks.
- Process 1 doesn’t receive any block.
- Process 2 gets the 2\(^{nd}\) and 4\(^{th}\) blocks.
- Process 3 doesn’t receive any block.

**Process 0:** \((0,0)\) in process grid

\[
Y^{(0)} = \begin{bmatrix}
y_1 \\
y_2 \\
y_5 \\
y_6 \\
\end{bmatrix}_{4 \times 1}
\]

\(Y^{(0)}\) is a vector of size 4 which is distributed to process 0 = \((0,0)\).

**Process 1:** \((0,1)\) in process grid.

Nothing is distributed to process 1
Process 2: (1,0) in process grid

\[ Y^{(2)} = \begin{bmatrix} y_3 \\ y_4 \\ y_7 \\ y_8 \end{bmatrix} \]

\( Y^{(2)} \) is a vector of size 4 which is distributed to process 2 = (1,0).

Process 3: (1,1) in process grid.

Nothing is distributed to process 3.

2.3.3 Data layout of 1 × 8 input vector \( Z \) on each process:

Let \( Z \) be a row vector of size 8, partitioned into 2 × 2 blocks, mapped onto a 2 × 2 process grid. The Figure 2.6 shows the partitioned vector using 2 × 2 blocks and the process coordinates for each block.

Since there is only one row, the \( Z \) vector is placed 2 components at a time on processes (0,0), (0,1), (0,0) and (0,1) alternately, that is in a column-cyclic manner.

\[
P_r \downarrow \quad P_c \Rightarrow \begin{array}{cccc}
0 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
0 & (z_1, z_2) & (z_3, z_4) & (z_5, z_6) & (z_7, z_8) \\
\end{array}
\]

\[
P_r \downarrow \quad P_c \Rightarrow \begin{array}{cccc}
0 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
0 & (0,0) & (0,1) & (0,0) & (0,1) \\
\end{array}
\]

Figure 2.6: Input \( Z \) vector partitioned into 2 × 2 blocks and the corresponding process coordinates for each block

13
Thus:

- Process 0 gets the 1\textsuperscript{st} and 3\textsuperscript{rd} blocks.
- Process 1 gets the 2\textsuperscript{nd} and 4\textsuperscript{th} blocks.
- Process 2 doesn’t receive any block.
- Process 3 doesn’t receive any block.

**Process 0:** (0,0) in process grid

\[ Z^{(0)} = \begin{bmatrix} z_1 & z_2 & z_5 & z_6 \end{bmatrix} \times 4 \]

\(Z^{(0)}\) is a vector of size 4 which is distributed to process 0 = (0,0).

**Process 1:** (0,1) in process grid

\[ Z^{(1)} = \begin{bmatrix} z_3 & z_4 & z_7 & z_8 \end{bmatrix} \times 4 \]

\(Z^{(1)}\) is a vector of size 4 which is distributed to process 1 = (0,1).

**Process 2:** (1,0) in process grid

Nothing is distributed to process 2.

**Process 3:** (1,1) in process grid

Nothing is distributed to process 3.

### 2.4 Computations and Communications

The computation on each process and the communication required between processes to accomplish PDGEMVT for a 8\times8 matrix partitioned into 2\times2 blocks and mapped onto a 2\times2 process grid is described in this section.

The cache efficient PDGEMVT makes use of column blocking on the distributed part of the A matrix on each process to achieve cache efficiency. On each process, the idea is to implement two simultaneous matrix-vector multiplications in the same way it was done in serial on a single processor. See Howell, et.al.[5], and S.Malhotra
[6] for a detailed description. We have already seen that the size of the $A$ matrix $(MP \times NQ)$ distributed to each process is $4 \times 4$ for our $8 \times 8$ global matrix, so we want to implement cache-efficiency on column blocks of each of the $4 \times 4$ matrices on each process.

Let us assume, $k = 2$, is the number of columns in each column block. Let $b$ be total number of blocks and $b \times k = NQ$. In general, the block size $k$ on each process will be dependent on $NQ$. For the sake of the present discussion, however, we have $NQ = 4$ on every process and for simplicity we take $k = 2$ on each process. Hence $b = NQ/k = 2$ on each process. The idea, of course, is to select the block size $k$ such that $MP \times k$ matrix elements fit into the $L_2$ cache and generally fill the $L_2$ cache on each process; we can then expect that the $MP \times k$ matrix elements in one block will remain in $L_2$ cache as the $2$ matrix-vector multiplications are executed, where $MP$ is the number of rows on each process. The column blocking on process 0 is shown in Figure 2.7.

\[
A^{(0)} = \begin{pmatrix}
a_{11} & a_{12} & a_{15} & a_{16} \\
a_{21} & a_{22} & a_{25} & a_{26} \\
a_{51} & a_{52} & a_{55} & a_{56} \\
a_{61} & a_{62} & a_{65} & a_{66}
\end{pmatrix} \quad 4 \times 4
\]

Figure 2.7: Column blocking with $k = 2$ on process 0 = (0,0)

The block cyclic cache efficient algorithm PDGEMVT can be described as follows:

**Step 1:** Copy the $Z^{(p)}$ vector to $X^{(p)}$

\[
X^{(p)} \leftarrow Z^{(p)}
\]

For $i = 1$ to $b$, do in parallel ($b = NQ/k$)

**Step 2:** Generate partial $X_i^{(p)}$ vectors on each process:
\[ X_i^{(p)} = \alpha \ast (A_i^{(p)})^T \ast Y^{(p)} + X_i^{(p)} \]  
(first _GEMV call)

**Step 3:** Sum the partial \( X_i^{(p)} \) vectors generated on processes \( p = 0 \) and \( p = 2 \) and overwrite \( X_i^{(p)} \) with this sum on each of these processes. Similarly sum the partial \( X_i^{(p)} \) vectors for \( p = 1 \) and \( p = 3 \), and overwrite with the sum, i.e.,

\[ X_i^{(0)} = X_i^{(2)} = X_i^{(0)} + X_i^{(2)} \]
\[ X_i^{(1)} = X_i^{(3)} = X_i^{(1)} + X_i^{(3)} \]

**Step 4:** Make use of the (complete) \( X_i^{(p)} \) vectors to generate all \( MP \) components of the partial \( W^{(p)} \) vector on each process. The \( W^{(p)} \) vector is accumulated with the previous \( W^{(p)} \) vector from blocks 1, 2, ... \( i - 1 \):

\[ W^{(p)} = \alpha \ast A_i^{(p)} \ast X_i^{(p)} + \beta \ast W^{(p)} \]  
(second _GEMV call)

End for loop (Now, increment \( i \) and repeat steps 2 - 4).

**Step 5:** Sum the partial \( W^{(p)} \) vectors generated on processes \( p = 0 \) and \( p = 1 \) and overwrite \( W^{(p)} \) with this sum on each of these processes. Similarly, sum the partial \( W^{(p)} \) vectors for \( p = 2 \) and \( p = 3 \), and overwrite with the sum, i.e.,

\[ W^{(0)} = W^{(1)} = W^{(0)} + W^{(1)} \]
\[ W^{(2)} = W^{(3)} = W^{(2)} + W^{(3)} \]

The above pseudo-code can be generalized to any \( M \times N \) global matrix \( A \) with any \( MB \times NB \) choice of blocks and any process grid \( P_r \times P_c \), and, in general, \( b = b^{(p)} \) is different on each process, owing to different block sizes \( k = k_b \). We now describe in more detail the above algorithm for the \( 8 \times 8 \) global matrix.

**Step 0:** (Preliminary steps done in the driver for PDGEMVT):

(a) The output \( W^{(p)} \) vector is initialized to zero in all processes

\[ W^{(p)} = 0 \]
(b) The $Z^{(0)}$ and $Z^{(1)}$ vectors from process 0 and process 1 are not required to be distributed to other processes, process 2 and process 3, because the $Z$ vectors are added only once in the process 0 and process 1 in the first $GEMV$ call. Thus, $Z^{(2)} = 0$ and $Z^{(3)} = 0$.

(c) Send the $Y$ vector which resides on the processes 0 and 2 to processes 1 and 3, respectively:

- Process 0 sends $Y^{(0)} = \begin{bmatrix} y_1 \\ y_2 \\ y_5 \\ y_6 \end{bmatrix}$ to Process 1 to make $Y^{(1)}$
- Process 2 sends $Y^{(2)} = \begin{bmatrix} y_3 \\ y_4 \\ y_7 \\ y_8 \end{bmatrix}$ to Process 3 to make $Y^{(3)}$

**Step 1:** Copy the $Z^{(p)}$ vector to $X^{(p)}$

Process 0 Computes:

$$X^{(0)} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \\ x_5^{(0)} \\ x_6^{(0)} \end{bmatrix} = Z^{(0)} = \begin{bmatrix} z_1^{(0)} \\ z_2^{(0)} \\ z_5^{(0)} \\ z_6^{(0)} \end{bmatrix}$$

Process 1 Computes:

$$X^{(1)} = \begin{bmatrix} x_3^{(1)} \\ x_4^{(1)} \\ x_7^{(1)} \\ x_8^{(1)} \end{bmatrix} = Z^{(1)} = \begin{bmatrix} z_3^{(1)} \\ z_4^{(1)} \\ z_7^{(1)} \\ z_8^{(1)} \end{bmatrix}$$

For $i = 1$:

**Step 2** ($i=1$): Generation of partial $X_1$ vectors on each process.
Process 0 Computes:

\[ X^{(0)}_1 = \begin{bmatrix} x^{(0)}_1 \\ x^{(0)}_2 \end{bmatrix} = (A^{(0)}_1)^T \mathbf{Y}^{(0)} + X^{(0)}_1 \]

\[ = \begin{bmatrix} a_{11} & a_{21} & a_{51} & a_{61} \\ a_{12} & a_{22} & a_{52} & a_{62} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_5 \\ y_6 \end{bmatrix} + \begin{bmatrix} x^{(0)}_3 \\ x^{(0)}_4 \end{bmatrix} \]

Process 2 Computes:

\[ X^{(2)}_1 = \begin{bmatrix} x^{(2)}_1 \\ x^{(2)}_2 \end{bmatrix} = (A^{(2)}_1)^T \mathbf{Y}^{(2)} = \begin{bmatrix} a_{31} & a_{41} & a_{71} & a_{81} \\ a_{32} & a_{42} & a_{72} & a_{82} \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ y_7 \\ y_8 \end{bmatrix} \]

Process 1 Computes:

\[ X^{(1)}_1 = \begin{bmatrix} x^{(1)}_3 \\ x^{(1)}_4 \end{bmatrix} = (A^{(1)}_1)^T \mathbf{Y}^{(1)} + X^{(1)}_1 \]

\[ = \begin{bmatrix} a_{13} & a_{23} & a_{53} & a_{63} \\ a_{14} & a_{24} & a_{54} & a_{64} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_5 \\ y_6 \end{bmatrix} + \begin{bmatrix} x^{(1)}_3 \\ x^{(1)}_4 \end{bmatrix} \]

Process 3 Computes:

\[ X^{(3)}_1 = \begin{bmatrix} x^{(3)}_3 \\ x^{(3)}_4 \end{bmatrix} = (A^{(3)}_1)^T \mathbf{Y}^{(3)} = \begin{bmatrix} a_{33} & a_{43} & a_{73} & a_{83} \\ a_{34} & a_{44} & a_{74} & a_{84} \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ y_7 \\ y_8 \end{bmatrix} \]

**Step 3** (i=1): Communication between those processes in each column of the process grid.

Process 0 and 2 communicate to sum the \( X^{(0)}_1 \) and \( X^{(2)}_1 \) vectors and overwrite them with the sum on each process, and similarly process 1 and process 3 communicate to sum the \( X^{(1)}_1 \) and \( X^{(3)}_1 \) vectors and and overwrite them with the sum on each process.

\[ X^{(0)}_1 = X^{(2)}_1 = \begin{bmatrix} x^{(0)}_1 \\ x^{(0)}_2 \end{bmatrix} = X^{(0)}_1 + X^{(2)}_1 = \begin{bmatrix} x^{(0)}_3 \\ x^{(0)}_4 \end{bmatrix} + \begin{bmatrix} x^{(2)}_3 \\ x^{(2)}_4 \end{bmatrix} \]
\[ X_1^{(1)} = X_1^{(3)} = \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} = X_1^{(1)} + X_1^{(3)} = \begin{bmatrix} x_3^{(1)} \\ x_4^{(1)} \end{bmatrix} + \begin{bmatrix} x_3^{(3)} \\ x_4^{(3)} \end{bmatrix} \]

**Step 4** (i=1): Generation of partial W vectors on each process.

Process 0 Computes:

\[
W^{(0)} = \begin{bmatrix} w_1^{(0)} \\ w_2^{(0)} \\ w_5^{(0)} \\ w_6^{(0)} \end{bmatrix} = A_1^{(0)} X_1^{(0)} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{51} & a_{52} \\ a_{61} & a_{62} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}
\]

Process 2 Computes:

\[
W^{(2)} = \begin{bmatrix} w_3^{(2)} \\ w_4^{(2)} \\ w_7^{(2)} \\ w_8^{(2)} \end{bmatrix} = A_1^{(2)} X_1^{(2)} = \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \\ a_{71} & a_{72} \\ a_{81} & a_{82} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}
\]

Process 1 Computes:

\[
W^{(1)} = \begin{bmatrix} w_1^{(1)} \\ w_2^{(1)} \\ w_5^{(1)} \\ w_6^{(1)} \end{bmatrix} = A_1^{(1)} X_1^{(1)} = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \\ a_{53} & a_{54} \\ a_{63} & a_{64} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}
\]

Process 3 Computes:

\[
W^{(3)} = \begin{bmatrix} w_3^{(3)} \\ w_4^{(3)} \\ w_7^{(3)} \\ w_8^{(3)} \end{bmatrix} = A_1^{(3)} X_1^{(3)} = \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \\ a_{73} & a_{74} \\ a_{83} & a_{84} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}
\]

NOTE: Cache-efficiency is achieved for i=1 by virtue of the fact that \(A_1^{(p)}\) remains in the \(L_2\) cache (for each process \(p\)) from step 2 through 4; that is, both the first and second matrix-vector multiplications in step 2 and 4 access the \(A_1^{(p)}\) matrix in \(L_2\) cache, since it remains in \(L_2\) cache while the additions and communications of step 3 are performed.
For $i = 2$:

NOTE: Now the $A_1^{(p)}$ matrices are flushed from the $L_2$ cache, and step 2 brings the $A_2^{(p)}$ matrices into the $L_2$ cache on each process.

**Step 2** ($i=2$): Generation of partial $X_2$ vectors on each process.

Process 0 Computes:

$$X_2^{(0)} = \begin{bmatrix} x_5^{(0)} \\ x_6^{(0)} \end{bmatrix} = (A_2^{(0)})^T Y^{(0)} + X_2^{(0)}$$

$$= \begin{bmatrix} a_{15} & a_{25} & a_{55} & a_{65} \\ a_{16} & a_{26} & a_{56} & a_{66} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_5 \\ y_6 \end{bmatrix} + \begin{bmatrix} x_5^{(0)} \\ x_6^{(0)} \end{bmatrix}$$

Process 2 Computes:

$$X_2^{(2)} = \begin{bmatrix} x_5^{(2)} \\ x_6^{(2)} \end{bmatrix} = (A_2^{(2)})^T Y^{(2)} = \begin{bmatrix} a_{35} & a_{45} & a_{75} & a_{85} \\ a_{36} & a_{46} & a_{76} & a_{86} \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ y_7 \\ y_8 \end{bmatrix}$$

Process 1 Computes:

$$X_2^{(1)} = \begin{bmatrix} x_7^{(1)} \\ x_8^{(1)} \end{bmatrix} = (A_2^{(1)})^T Y^{(1)} + X_2^{(1)}$$

$$= \begin{bmatrix} a_{17} & a_{27} & a_{57} & a_{67} \\ a_{18} & a_{28} & a_{58} & a_{68} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_5 \\ y_6 \end{bmatrix} + \begin{bmatrix} x_7^{(1)} \\ x_8^{(1)} \end{bmatrix}$$

Process 3 Computes:

$$X_2^{(3)} = \begin{bmatrix} x_7^{(3)} \\ x_8^{(3)} \end{bmatrix} = (A_2^{(3)})^T Y^{(3)} = \begin{bmatrix} a_{37} & a_{47} & a_{77} & a_{87} \\ a_{38} & a_{48} & a_{78} & a_{88} \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ y_7 \\ y_8 \end{bmatrix}$$
Step 3 \((i=2)\): Communication between those processes in each column of the process grid.

Process 0 and 2 communicate to sum the \(X^{(0)}_2\) and \(X^{(2)}_2\) vectors and overwrite them with the sum on each process and similarly process 1 and process 3 communicate to sum the \(X^{(1)}_2\) and \(X^{(3)}_2\) vectors and overwrite them with the sum on each process.

\[
X^{(0)}_2 = X^{(2)}_2 = \begin{bmatrix} x_5 \\ x_6 \end{bmatrix} = X^{(0)}_2 + X^{(2)}_2 = \begin{bmatrix} x_5^{(0)} \\ x_6^{(0)} \end{bmatrix} + \begin{bmatrix} x_5^{(2)} \\ x_6^{(2)} \end{bmatrix}
\]

\[
X^{(1)}_2 = X^{(3)}_2 = \begin{bmatrix} x_7 \\ x_8 \end{bmatrix} = X^{(1)}_2 + X^{(3)}_2 = \begin{bmatrix} x_7^{(1)} \\ x_8^{(1)} \end{bmatrix} + \begin{bmatrix} x_7^{(3)} \\ x_8^{(3)} \end{bmatrix}
\]

Step 4 \((i=2)\): Generation of partial \(W\) vectors on each process.

Process 0 Computes:

\[
W^{(0)} = \begin{bmatrix} w^{(0)}_1 \\ w^{(0)}_2 \\ w^{(0)}_5 \end{bmatrix} = W^{(0)} + A^{(0)}_2 X^{(0)}_2 = \begin{bmatrix} w^{(0)}_1 \\ w^{(0)}_2 \\ w^{(0)}_5 \end{bmatrix} + \begin{bmatrix} a_{15} \\ a_{25} \\ a_{55} \end{bmatrix} \begin{bmatrix} x_5 \\ x_6 \end{bmatrix}
\]

Process 2 Computes:

\[
W^{(2)} = \begin{bmatrix} w^{(2)}_3 \\ w^{(2)}_4 \\ w^{(2)}_7 \\ w^{(2)}_8 \end{bmatrix} = W^{(2)} + A^{(2)}_2 X^{(2)}_2 = \begin{bmatrix} w^{(2)}_3 \\ w^{(2)}_4 \\ w^{(2)}_7 \\ w^{(2)}_8 \end{bmatrix} + \begin{bmatrix} a_{35} \\ a_{45} \\ a_{75} \\ a_{85} \end{bmatrix} \begin{bmatrix} x_5 \\ x_6 \end{bmatrix}
\]

Process 1 Computes:

\[
W^{(1)} = \begin{bmatrix} w^{(1)}_1 \\ w^{(1)}_2 \\ w^{(1)}_5 \end{bmatrix} = W^{(1)} + A^{(1)}_2 X^{(1)}_2 = \begin{bmatrix} w^{(1)}_1 \\ w^{(1)}_2 \\ w^{(1)}_5 \end{bmatrix} + \begin{bmatrix} a_{17} \\ a_{27} \\ a_{57} \end{bmatrix} \begin{bmatrix} x_7 \\ x_8 \end{bmatrix}
\]

Process 3 Computes:

\[
W^{(3)} = \begin{bmatrix} w^{(3)}_3 \\ w^{(3)}_4 \\ w^{(3)}_7 \\ w^{(3)}_8 \end{bmatrix} = W^{(3)} + A^{(3)}_2 X^{(3)}_2 = \begin{bmatrix} w^{(3)}_3 \\ w^{(3)}_4 \\ w^{(3)}_7 \\ w^{(3)}_8 \end{bmatrix} + \begin{bmatrix} a_{37} \\ a_{47} \\ a_{77} \\ a_{87} \end{bmatrix} \begin{bmatrix} x_7 \\ x_8 \end{bmatrix}
\]

21
Step 5: Communication between those processes in each row of the process grid, to generate the (complete) $W$ vectors on each process.

Process 0 and 1 communicate to sum $W^{(0)}$ and $W^{(2)}$ vectors and overwrite them with the sum on each process, and similarly process 1 and process 3 communicate to sum $W^{(1)}$ and $W^{(3)}$ vectors and overwrite them with the sum on each process.

$$W^{(0)} = W^{(1)} = \begin{bmatrix} w_1 \\ w_2 \\ w_5 \\ w_6 \end{bmatrix} = W^{(0)} + W^{(1)} = \begin{bmatrix} w_1^{(0)} \\ w_2^{(0)} \\ w_5^{(0)} \\ w_6^{(0)} \end{bmatrix} + \begin{bmatrix} w_1^{(1)} \\ w_2^{(1)} \\ w_5^{(1)} \\ w_6^{(1)} \end{bmatrix}$$

$$W^{(2)} = W^{(3)} = \begin{bmatrix} w_3 \\ w_4 \\ w_7 \\ w_8 \end{bmatrix} = W^{(2)} + W^{(3)} = \begin{bmatrix} w_3^{(2)} \\ w_4^{(2)} \\ w_7^{(2)} \\ w_8^{(2)} \end{bmatrix} + \begin{bmatrix} w_3^{(3)} \\ w_4^{(3)} \\ w_7^{(3)} \\ w_8^{(3)} \end{bmatrix}$$

Step 6: Generation of global $W$ vector. This is done in the driver program for PDGEMVT.

Process 2 sends its $W$ vector to process 0 to make the (complete) $W$ vector of size 8.

$$W = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \end{bmatrix}$$

2.4.1 Pseudo-code for a general matrix

Here we describe the generalization of the above pseudo code to an arbitrary $M \times N$ matrix, and write down the $F77$ statements from the actual code which implement each step of the pseudo code.
Let \( \text{sub}(A) \) be the part of \( A \) on process \( p \). Let the distributed matrix \( \text{sub}(A) \) on each process \( p \) be of size \( M \times N \). Let \( k \) be the number of columns of \( \text{sub}(A) \) for which the \( M \times k \) block fits into \( L_2 \) cache. Let \( b = N/k \)

**Step 1:** Load \( Z(p) \) into \( X(p) \).

For \( i = 1, b \), do in parallel (\( b = N/k \)):

**Step 2:** Generate partial \( X_i(p) \) vectors, by a call to DGEMV on each process:

\[
(sub(X))_i = \alpha * (sub(A))_i * sub(Y) + \beta * (sub(Z))_i
\]

CALL DGEMV('T', M, K, ALPHA, A((I-1)*K*M + 1), DESCA(LLD_),
+ Y, INCY, BETA, X((I-1)*K + 1), INCX)

**Step 3:** Communication: Sum the partial \( X_i(p) \) over each column of the process grid and overwrite \( X_i(p) \) with this sum. For this purpose, the BLACS routine DGSUM2D is used. Therefore, for each \( q \in \{ \text{column communicator} \} \), we perform the overwrite,

\[
(sub(X(q)))_i = \sum_{p \in \{ \text{col communicator} \}} (sub(X(p)))_i
\]

CALL DGSUM2D(ICTXT, 'COLUMN', ' ', K, 1, X((I-1)*K + 1),
+ DESCX(LLD_), -1, -1)

**NOTE:** Because the BLACS are written on top of MPI (Message Passing Interface), it is not necessary to initialize communicators which group the processors belonging to each column and each row of the process grid; these communicators are automatically created by the BLACS, and are accessed in the DGSUM2D calls by specifying 'Column' or 'Row' in the 2\(^{nd} \) argument.

**Step 4:** Generate the partial \( W(p) \) vectors by a second call to DGEMV on each process.

\[
\text{sub}(W) = \alpha * (sub(A))_i * (sub(X))_i + \beta * \text{sub}(W)
\]
CALL DGEMV('N', M, K, ALPHA, A((I-1)*K*M + 1), DESCA(LLD_), + X((I-1)*K + 1), INCX, BETA, W, INCW)

End For.

**Step 5:** Communication: Sum the partial $W^{(p)}$ vectors over each row of the process grid and overwrite $W^{(p)}$ with this sum to generate the (Complete) $W$ vector. The BLACS routine DGSUM2D is used for this purpose. for each $q \in \{\text{row communicator}\}$, we perform the overwrite,

$$
\text{sub}(W^{(q)}) = \sum_{p \in \{\text{row communicator}\}} \text{sub}(W^{(p)})
$$

CALL DGSUM2D(ICTX, 'ROW', ' ', M, 1, W, DESCW(LLD_), -1, -1)

### 2.4.2 Source code of the subroutine PDGEMVT

SUBROUTINE PDGEMVT ( M, N, ALPHA, A, IA, JA, DESCA, X, IX, JX, + DESCX, INCX, Y, IY, JY, DESCY, INCY, BETA, + W, IW, JW, DESCW, INCW, Z, IZ, JZ, DESCZ, + INCZ)

* -- Scalar Arguments --

INTEGER M, N, IA, JA, IX, JX, IY, JY, IW, JW, IZ, JZ
INTEGER INCX, INCY, INCW, INCZ
DOUBLE PRECISION ALPHA, BETA

* -- Array Arguments --

INTEGER DESCA(*), DESCX(*), DESCY(*)
INTEGER DESCW(*), DESCZ(*)
DOUBLE PRECISION A(*), X(*), Y(*), W(*), Z(*)

* -- Local Variables --
INTEGER INFO, ICTXT, IAM
INTEGER I, J, K, B
INTEGER START_A, START_Y, START_X, START_W
CHARACTER*1 ROW, COL, TOP
PARAMETER (ROW = 'R', COL = 'C', TOP = ' ')
INTEGER BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DT_,
         + LLD_, MB_, M_, NB_, N_, RSRC_
PARAMETER ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DT_ = 1,
            + CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5, NB_ = 6,
            + RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )

ICTXT = DESCA(CTXT_)
* -- Initializing the starting pointers.  
*  Generally all values will be 1 --
  START_A = (JA - 1) * DESCA(M_) + (IA - 1) + 1
  START_Y = (JY - 1) * DESCY(M_) + (IY - 1) + 1
  START_X = (JX - 1) * DESCX(M_) + (IX - 1) + 1
  START_W = (JW - 1) * DESCW(M_) + (IW - 1) + 1

* -- Copy the Z vector into the X Vector --
CALL DCOPY(N, Z, INCZ, X, INCX)
* -- Determine the block size 'k' for the local matrix ----
CALL BLOCKSIZE(M, K)
B = N/K

DO I = 1, B

* -- Perform 1st GEMV call, implementing Step 2 --
CALL DGEMV('T', M, K, ALPHA, A(START_A + ((I-1)*K*M)),
            + DESCA(LLD_), Y(START_Y), INCY, BETA,
            + X(START_X + (I-1)*K), INCX)
* -- Perform communication of X_i vectors, implementing Step 3 --
CALL DGSUM2D(ICTXT, COL, TOP, K, 1,
+ X(START_X + (I-1)*K), DESCX(LLD_), -1, -1)

* -- Perform 2nd GEMV call to accumulate the W-vectors,
* implementing Step 4 --
CALL DGEMV('N', M, K, ALPHA, A(START_A + ((I-1)*K*M)),
+ DESCA(LLD_), X(START_X + (I-1)*K), INCX,
+ BETA, W(START_W), INCW)
END DO
*
* -- CLEAN UP CODE --
*
J = (N/K) * K
IF (J.LT.N) THEN
  CALL DGEMV('T', M, N-J, ALPHA, A(START_A + (J*M)),
+ DESCA(LLD_), Y(START_Y), INCY, BETA,
+ X(START_X + J), INCX)
  CALL DGSUM2D(ICTXT, COL, TOP, N-J, 1, X(START_X + J),
+ DESCX(LLD_), -1, -1)
  CALL DGEMV('N', M, N-J, ALPHA, A(START_A + (J*M)),
+ DESCA(LLD_), X(START_X + J), INCX, BETA,
+ W(START_W), INCW)
END IF
*
* -- Sum the partial W vectors over each row of the process grid
* to generate the (complete) W vector. This implements Step 5 --
CALL DGSUM2D(ICTXT, ROW, TOP, M, 1, W(START_W), DESCW(LLD_),
+ -1, -1)
RETURN
END
**************************************************************************
END PDGEMVT**************************************************************************
The complete source code of the subroutine PDGEMVT is shown above. The last portion of the code is the clean up code. It is possible that the number of columns NQ of the distributed matrix on each process cannot be exactly divided by the block size k. This clean up code is therefore necessary to take care of the extra columns which are left over after partitioning the distributed matrix into column blocks of k columns each.

2.5 Driver Program

In this section, the parallel cache efficient GEMVT algorithm is described from the PDGEMVT driver program perspective on the 8 × 8 matrix whose data elements are block cyclicly distributed. Prior to our description of the algorithm, the ScaLAPACK auxiliary routines NUMROC and DESCINIT are described in Section 2.5.1.

2.5.1 Auxiliary Routines

Our parallel algorithm PDGEMVT uses some of the ScaLAPACK auxiliary routines like NUMROC and DESCINIT. These purpose of these routines and their inputs and outputs are explained in detail in this section, following the documentation in [3].

NUMROC

The ScaLAPACK tool function NUMROC (NUMber of Rows Or Columns) computes the local number of rows or columns of a block-cyclicly distributed matrix owned by the process indicated by IPROC.

Syntax

\[ \text{INTEGER FUNCTION NUMROC}(N, NB, IPROC, ISRCPROC, NPROCS) \]

On Entry

\begin{align*}
N & \quad (\text{global input}) \quad \text{INTEGER} \\
& \quad \text{Total number of columns (N) in the global matrix or total number of rows (M) in the global matrix.} \\
NB & \quad (\text{global input}) \quad \text{INTEGER} \\
& \quad \text{Number of columns per block (NB) or number of rows per block (MB).}
\end{align*}
IPROC  (local input) INTEGER
Column coordinate (MYCOL = q) of the process (p, q) whose
local number of columns is to be determined. Or,
Row coordinate (MYCOL = p) of the process (p, q) whose
local number of rows is to be determined.

ISRCPROC  (global input) INTEGER
The coordinate of the process that possesses the first
process row (or process column) of the distributed matrix.

NPROCS  (global input) INTEGER
The number of process rows (NPROW), or number of process
columns (NPCOL) in the process grid.

On Output

\[ X = \text{NUMROC}(N, NB, MYCOL, ISRCPROC, NPCOL) \]

where, \( X \) is the local number of columns of the \( M \times N \) global matrix \( A \) residing on
processes \((*,MYCOL)\) using an \((*) \times NPCOL\) process grid.

DESCINIT

Each global data object matrix or vector is described by an associated array descriptor. This array stores the information required to establish the mapping between
the global object element and its corresponding process and memory location. The
ScaLAPACK tool function DESCINIT initializes the array descriptor (DESC) with
9 values of which 8 are input arguments.

Syntax

\[
\text{SUBROUTINE DESCINIT}(\text{DESC}, M, N, MB, NB, RSRC, CSRC, ICTXT, MDLLDA, INFO)\]

On Entry

\[
\text{DESC}  \quad \text{(output) INTEGER array of dimension DLEN}.
A one-dimensional array containing the 9 integer arguments
DTYPE, M, N, MB, NB, RSRC, CSRC, ICTXT and LLDA. The
descriptor type, DTYPE, is hard coded as 1 in this subroutine.
\]

\[
M  \quad \text{(global input) INTEGER}
The number of rows in the distributed matrix. \( M \geq 0 \).\]
**N** (global input) INTEGER
The number of columns in the distributed matrix. \( N \geq 0. \)

**MB** (global input) INTEGER
The blocking factor used to distribute the rows of the matrix. \( MB \geq 1. \)

**NB** (global input) INTEGER
The blocking factor used to distribute the columns of the matrix. \( MB \geq 1. \)

**IRSRC** (global input) INTEGER
The process row over which the first row of the matrix is distributed. \( 0 < IRSRC < NPROW. \)

**ICSRC** (global input) INTEGER
The process column over which the first column of the matrix is distributed. \( 0 < ICSRC < NPCOL. \)

**ICTXT** (global input) INTEGER
The BLACS context handle, indicating the global context of the operation on the matrix. The context itself is global. (This is the integer which identifies the communicator that all the processes belong to.)

**LLD** (local input) INTEGER
The leading dimension of the local array storing the local blocks of the distributed matrix. \( LLD \geq \text{MAX}(1, \text{LOC}_r(A)). \)

**INFO** (output) INTEGER
- \( = 0: \) successful exit
- \( < 0: \) if \( INFO = -i, \) the \( i\)-th argument had an illegal value

Let A be the generic label for any 2-dimensional block cyclicly distributed matrix. Such a global matrix has an associated array descriptor DESCA. The details of the entries in this array is given in Table 2.1. In this table, the character \( \_ \) should be read as “of the global matrix A”.

<table>
<thead>
<tr>
<th>Index</th>
<th>Notation</th>
<th>Stored In</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DTYPE_A</td>
<td>DESCA(DTYPE(__))</td>
<td>The descriptor type. In this case ( DTYPE__ = 1. )</td>
</tr>
<tr>
<td>2</td>
<td>CTXT_A</td>
<td>DESCA(CTXT(__))</td>
<td>The BLACS context handle, indicating the BLACS process grid A is distributed over. The context itself is global, but the handle (the integer value) may vary.</td>
</tr>
</tbody>
</table>

Table 2.1: DESC Parameters
<table>
<thead>
<tr>
<th>Index</th>
<th>Notation</th>
<th>Stored In</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>M_A</td>
<td>DESCA(M_A)</td>
<td>The number of rows in the global array A.</td>
</tr>
<tr>
<td>4</td>
<td>N_A</td>
<td>DESCA(N_A)</td>
<td>The number of columns in the global array A.</td>
</tr>
<tr>
<td>5</td>
<td>MB_A</td>
<td>DESCA(MB_A)</td>
<td>The blocking factor used to distribute the rows of the array.</td>
</tr>
<tr>
<td>6</td>
<td>NB_A</td>
<td>DESCA(NB_A)</td>
<td>The blocking factor used to distribute the columns of the array.</td>
</tr>
<tr>
<td>7</td>
<td>RSRC_A</td>
<td>DESCA(RSRC_A)</td>
<td>The process row over which the first row of the array A is distributed.</td>
</tr>
<tr>
<td>8</td>
<td>CSRC_A</td>
<td>DESCA(CSRC_A)</td>
<td>The process column over which the first column of the array A is distributed.</td>
</tr>
<tr>
<td>9</td>
<td>LLD_A</td>
<td>DESCA(LLD_A)</td>
<td>The leading dimension of the local array.</td>
</tr>
</tbody>
</table>

Table 2.1: DESC Parameters (Continued)

2.5.2 Driver Program

The driver program for the parallel cache efficient PDGEMVT subroutine for $8 \times 8$ global matrix $A$ is described in this section.

1. Call subroutine `BLACS_PINFO(IAM, NPROCS)`\(^1\) to get values for $NPROCS$ and $IAM$ where $NPROCS$ is the total number of processes and $IAM$ is the rank of each process in the process grid.

2. Call subroutine `PDPBLASINFO(OUTFILE, ..., IAM, NPROCS)`\(^1\) to read the input file to get values for:

   \[
   \begin{align*}
   M &= \text{number of rows in global matrix } A \text{ and size of vector } Y \\
   N &= \text{number of columns in global matrix } A \text{ and size of vector } X \\
   V &= \text{number of columns in vector } Y \text{ which is always 1} \\
   MB &= \text{number of rows in the block}
   \end{align*}
   \]

\(^1\)Refer Appendix B for BLACS description.
\[ NB = \text{number of columns in the block} \]
\[ NPROW = \text{number of process rows} \]
\[ NPCOL = \text{number of process columns} \]
\[ \{ NPROW \times NPCOL \} = \text{total number of processes in the process grid.} \]

The following values are read from the input file for the $8 \times 8$ example matrix mapped onto a $2 \times 2$ process grid:

\begin{align*}
M &= 8 & NPROW &= 2 \\
N &= 8 & NPCOL &= 2 \\
V &= 1 \\
MB &= 2 \\
NB &= 2
\end{align*}

3. Call BLACS subroutines \texttt{BLACS\_GET} \footnote{Refer Appendix B for BLACS description.}, \texttt{BLACS\_GRIDINIT} and \texttt{BLACS\_GRIDINFO} \footnote{Refer Appendix B for BLACS description.} to get the global grid context, to initialize the Process Grid and to retrieve the process grid information respectively.

4. Call ScaLAPACK tool function \texttt{NUMROC} to get the exact number of rows and columns of that portion of the $A$ matrix distributed on process $P = (MYROW, MYCOL)$. Similarly, for the number of rows and columns of that portion of the $Y$ and $Z$ vectors distributed on each process.

\begin{align*}
MP &= \texttt{NUMROC}(M, MB, MYROW, 0, NPROW) \\
NQ &= \texttt{NUMROC}(N, NB, MYCOL, 0, NPCOL) \\
YQ &= \texttt{NUMROC}(V, NB, MYCOL, 0, NPCOL) \\
YP &= \begin{cases} 
MP & \text{if } YQ = 1 \\
0 & \text{if } YQ = 0 
\end{cases} \\
ZP &= \texttt{NUMROC}(V, MB, MYROW, 0, NPROW) \\
ZQ &= \begin{cases} 
NQ & \text{if } ZP = 1 \\
0 & \text{if } ZP = 0 
\end{cases}
\end{align*}
\[ M = \text{number of rows of the global matrix} \]
\[ N = \text{number of columns of the global matrix} \]
\[ V = 1 \text{ (because } Y \text{ is a vector)} \]
\[ MB = \text{number of rows in each } MB \times NB \text{ block} \]
\[ NB = \text{number of columns in each } MB \times NB \text{ block} \]
\[ MYROW = \text{row coordinate of process (MYROW, MYCOL)} \]
\[ MYCOL = \text{column coordinate of process (MYROW, MYCOL)} \]
\[ NPROW = \text{number of process rows in the process grid} \]
\[ NPCOL = \text{number of process columns in the process grid} \]

Therefore,

\[ MP = \text{number of rows in the part of matrix } A \text{ which is}
\text{distributed to processes(MYROW,J), } J = 0, 1, \ldots \text{ NPCOL-1}
\]
\[ = LOC_r(A) \]
\[ NQ = \text{number of columns in part of matrix } A \text{ which is}
\text{distributed to processes(I,MYCOL), } I = 0, 1, \ldots \text{ NPROW-1}
\]
\[ = LOC_c(A) \]
\[ YQ = \text{number of columns of vector } Y \text{ and } W \text{ which are}
\text{distributed to processes(I,MYROW), } I = 0, 1, \ldots \text{ NPROW-1} \]
\[ YP = \text{number of rows of vector } Y \text{ and } W \text{ which are}
\text{distributed to processes (MYROW,J), } J = 0, 1, \ldots \text{ NPCOL-1} \]
\[ ZP = \text{number of rows of vector } Z \text{ and } X \text{ which are}
\text{distributed to processes(I, MYCOL), } I = 0, 1, \ldots \text{ NPROW-1} \]
\[ ZQ = \text{number of columns of vector } Y \text{ and } W \text{ which are}
\text{distributed to processes (MYCOL, J), } J = 0, 1, \ldots \text{ NPCOL-1} \]

The input values to subroutine NUMROC and the output values \( MP \), \( NQ \), \( YQ \), \( YP \), \( ZP \) and \( ZQ \) from the subroutine on each process are listed in Table 2.2.

Therefore the number of rows and columns of the distributed matrix \( A \) on each process is \( MP \times NQ \)

- On process 0: \( 4 \times 4 \)
- On process 1: \( 4 \times 4 \)
- On process 2: \( 4 \times 4 \)
- On process 3: \( 4 \times 4 \)

The size of input vector \( Y \) is also determined on each process by making use of \( YP \) (size of the vector on each process) and \( YQ \) (number of columns in \( Y \) on each process (either 0 or 1)).

- On process 0: \( 4 \times 1 \)
Table 2.2: Input and output values to and from the subroutine NUMROC on each process for a $8 \times 8$ matrix

<table>
<thead>
<tr>
<th>Process#</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$InputValues$</td>
<td>[</td>
<td>[</td>
<td>[</td>
<td>[</td>
</tr>
<tr>
<td>$M$</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$N$</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$V$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$MB$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$NB$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$MYROW$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$MYCOL$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$NPROW$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$NPCOL$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$OutputValues$</td>
<td>[</td>
<td>[</td>
<td>[</td>
<td>[</td>
</tr>
<tr>
<td>$MP$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$NQ$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$YP$</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>$YQ$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$ZP$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$ZQ$</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

On process 1: \textit{Nothing stored}

On process 2: $4 \times 1$

On process 3: \textit{Nothing stored}

The size of input vector $Z$ is also determined on each process by making use of $ZP$ (size of the vector on each process) and $ZQ$ (number of rows in $Z$ on each process (either 0 or 1)).

On process 0: $1 \times 4$
On process 1: $1 \times 4$
On process 2: \textit{Nothing stored}
On process 3: \textit{Nothing stored}

5. Call ScaLAPACK tool routine DESCINIT to initialize the array descriptor (DESC_) for the matrix $A$ and input vectors $Y$ and $Z$, and output vectors $X$ and $W$. 

33
Let DESCA, DESCY, DESCZ, DESCX and DESCW be the array descriptors for matrix \(A\) and input vectors \(Y\) and \(Z\), and output vectors \(X\) and \(W\). Then DESCA(I), DESCY(I), DESCZ(I), DESCX(I) and DESCW(I) for \(I = 1, 9\) are arrays of integers which are initialized to the values shown in Table 2.3. These initializations are accomplished by the following subroutines calls:

\[
\text{CALL DESCINIT( DESCA, M, N, MB, NB, RSRC, CSRC, ICTXT, LLD\_A, } \\
\text{INFO)}
\]
\[
\text{CALL DESCINIT( DESCY, M, V, MB, NB, RSRC, CSRC, ICTXT, LLD\_Y, } \\
\text{INFO)}
\]
\[
\text{CALL DESCINIT( DESCW, M, V, MB, NB, RSRC, CSRC, ICTXT, LLD\_W, } \\
\text{INFO)}
\]
\[
\text{CALL DESCINIT( DESCZ, V, N, MB, NB, RSRC, CSRC, ICTXT, LLD\_Z, } \\
\text{INFO)}
\]
\[
\text{CALL DESCINIT( DESCX, V, N, MB, NB, RSRC, CSRC, ICTXT, LLD\_X, } \\
\text{INFO)}
\]

<table>
<thead>
<tr>
<th>DESC(I)</th>
<th>Parameter</th>
<th>DESCA</th>
<th>DESCY</th>
<th>DESCZ</th>
<th>DESCX</th>
<th>DESCW</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESC(1)</td>
<td>DTYPE</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DESC(2)</td>
<td>ICTXT</td>
<td>ICTXT</td>
<td>ICTXT</td>
<td>ICTXT</td>
<td>ICTXT</td>
<td>ICTXT</td>
</tr>
<tr>
<td>DESC(3)</td>
<td>M</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>DESC(4)</td>
<td>N</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>DESC(5)</td>
<td>MB</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DESC(6)</td>
<td>NB</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DESC(7)</td>
<td>RSRC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DESC(8)</td>
<td>CSRC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DESC(9)</td>
<td>LLD</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.3: DESC Parameters for the \(8 \times 8\) matrix \(A\), \(8 \times 1\) vectors \(Y\) and \(W\), and \(1 \times 8\) vectors \(Z\) and \(X\)
6. The driver program is executed by each process simultaneously and thus stores its part of the distributed matrix $A$ and the distributed vectors $Y$ and $Z$. A single vector $MEM$ is used to store the input matrix $A$, input vectors $Y$ and $Z$ and output vectors $X$ and $W$. We introduce the pointers (on each process) $IPA, IPY, IPZ, IPX$ and $IPW$ to point to the location in MEM where matrix $A$, vector $Y$, vector $Z$, vector $X$ and vector $W$ are stored, respectively.

Therefore, $MEM$ is a vector to store part of matrix $A$, vector $Y$, $Z$, $X$, and $W$ on each process and $IPA, IPY, IPZ, IPX$ and $IPW$ are the pointers having starting addresses of the parts of the matrix $A$, vector $Y$, vector $Z$, vector $X$, and vector $W$ on each process, respectively. We also define $IPL$ to be the ending address of $MEM$ array on each process.

The formulas for these pointers, on each process, are thus as follows:

\[
\begin{align*}
\text{IPA} & = 1 \\
\text{IPY} & = \text{IPA} + \text{DESCA} (L LD) \times NQ \\
\text{IPZ} & = \text{IPY} + \text{DESCY} (L LD) \\
\text{IPX} & = \text{IPZ} + \text{DESCZ} (L LD) \times NQ \\
\text{IPW} & = \text{IPX} + \text{DESCX} (L LD) \times NQ \\
\text{IPL} & = \text{IPW} + \text{DESCW} (L LD)
\end{align*}
\]

Where,

- $IPA$ is the starting address of matrix $A$ in $MEM$
- $IPY$ is the starting address of vector $Y$ in $MEM$
- $IPZ$ is the starting address of vector $Z$ in $MEM$
- $IPX$ is the starting address of vector $X$ in $MEM$
- $IPW$ is the starting address of vector $W$ in $MEM$
- $IPL$ is the ending address of $MEM$

The values of $IPA, IPY, IPZ, IPX, IPW$ and $IPL$ on each process for a $8 \times 8$ matrix and $8 \times 1$ input vector and the corresponding values of $L LD_A = \text{DESCA}(9), L LD_Y = \text{DESCY}(9), L LD_Z = \text{DESCZ}(9), L LD_X = \text{DESCX}(9), L LD_W = \text{DESCW}(9)$ are listed for each process in the Table 2.4.
7. Generate the distributed $A$ matrix and the distributed input vectors $Y$ and $Z$ on each process using the ScaLAPACK routine \textit{PDMATGEN}. The distribution of matrix $A$, vectors $Y$ and $Z$ on each process is explained in Section 2.3.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Process# & 0 & 1 & 2 & 3 \\
\hline
\textit{IPA} & 1 & 1 & 1 & 1 \\
\textit{IPY} & 17 & 17 & 17 & 17 \\
\textit{IPZ} & 21 & 21 & 21 & 21 \\
\textit{IPX} & 25 & 25 & 25 & 25 \\
\textit{IPW} & 29 & 29 & 29 & 29 \\
\textit{IPL} & 33 & 33 & 33 & 33 \\
\hline
\textit{LLD}_A & 4 & 4 & 4 & 4 \\
\textit{LLD}_Y & 4 & 0 & 4 & 0 \\
\textit{LLD}_Z & 1 & 0 & 1 & 0 \\
\textit{LLD}_X & 1 & 0 & 1 & 0 \\
\textit{LLD}_W & 4 & 0 & 4 & 0 \\
\hline
\end{tabular}
\caption{Values of $IPA$, $IPY$, $IPZ$, $IPX$, $IPW$ and $IPL$ and values of $LLD_A$, $LLD_Y$, $LLD_Z$, $LLD_X$, $LLD_W$ and $LLD_L$ on each process}
\end{table}

\begin{verbatim}
IASEED = 100
CALL PDMATGEN( ICTXT, 'No transpose', 'No transpose',
+ DESCA( M_ ), DESCA( N_ ), DESCA( MB_ ),
+ DESCA( NB_ ), MEM( IPA ), DESCA( LLD_ ),
+ DESCA( RSRC_ ), DESCA( CSRC_ ), IASEED,
+ 0, MP, 0, NQ, MYROW, MYCOL, NPROW, NPCOL )

IBSEED = 200
CALL PDMATGEN( ICTXT, 'No transpose', 'No transpose',
+ DESCY( M_ ), DESCY( N_ ), DESCY( MB_ ),
+ DESCY( NB_ ), MEM( IPY ), DESCY( LLD_ ),
+ DESCY( RSRC_ ), DESCY( CSRC_ ), IBSEED,
+ 0, MP, 0, NQ, MYROW, MYCOL, NPROW, NPCOL )

ICSEED = 300
CALL PDMATGEN( ICTXT, 'No transpose', 'No transpose',
+ DESCX( M_ ), DESCX( N_ ), DESCX( MB_ ),
\end{verbatim}
8. Send the $Y$ vector from processes 0 and 2 to all the other processes in each row communicator of the process grid. In our example, $Y^{(0)}$ is sent from process 0 to process 1 and stored as $Y^{(1)}$ and similarly $Y^{(2)}$ is sent from process 2 to process 3 and stored as $Y^{(3)}$.

```fortran
IF (MYCOL.EQ.0) THEN
   -- Sending the Y Vector --
   CALL DGEBS2D(ICTXT, 'R', ' ', MP, 1, MEM(IPY), DESCY(LLD_))
ELSE
   -- Receiving the Y Vector --
   CALL DGEBR2D(ICTXT, 'R', ' ', MP, 1, MEM(IPY), DESCY(LLD_),
               MYROW, 0)
ENDIF
```

9. We have distributed the matrix $A$ and distributed the input vectors $Y$ and $Z$ on each process. Now, call the cache efficient PDGEMVT routine on each process to generate the output vector $W$; internally PDGEMVT makes two calls to DGEMV using column blocking on the distributed matrix on each process. The object is to achieve cache-efficiency by selecting the block size $k$ so that the $MP \times k$ blocks effectively fill the available $L_2$ cache on each process.

```fortran
CALL PDGEMVT(MP, NQ, 1.d0, MEM(IPA), 1, 1, DESCA, MEM(IPX), 1,
              1, DESCX, 1, MEM(IPY), 1, 1, DESCY, 1, 1.d0,
              MEM(IPW), 1, 1, DESCW, 1, MEM(IPZ), 1, 1,
              DESCZ, 1)
```

10. Call subroutine `BLACS_GRIDEXIT(ICTX)` \(^1\) to release the process grid and and `BLACS_EXIT(0)` \(^1\) to indicate that all work using the BLACS has been completed; this call initiates the destruction of the BLACS internal structures and frees all memory.

\(^1\)Refer Appendix B for BLACS description.
2.6 Description of subroutines on each process

The action of all subroutines in our cache efficient PDGEMVT using the block cyclic data distribution is described in this section using the 8 × 8 example matrix.

Example: Let $A$ be the following 8 × 8 matrix, partitioned into 2 × 2 blocks, and mapped onto a 2 × 2 process grid.

$$
A = \begin{pmatrix}
11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\
21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\
31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\
41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\
51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \\
61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 \\
71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 \\
81 & 82 & 83 & 84 & 85 & 86 & 87 & 88 \\
\end{pmatrix}
$$

$P_r \downarrow$ $P_c \Rightarrow$ 0 1 0 1

\[\begin{array}{cccc}
0 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\
21 & 22 & 23 & 24 & 26 & 26 & 27 & 28 \\
31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\
41 & 42 & 43 & 44 & 46 & 46 & 47 & 48 \\
51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \\
61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 \\
71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 \\
81 & 82 & 83 & 84 & 85 & 86 & 87 & 88 \\
\end{array}\]

Figure 2.8: An 8 × 8 example matrix partitioned into 2 × 2 blocks in block cyclic fashion

The distribution of matrix $A$ on the 2 × 2 process grid is shown in Figure 2.8. The data layout of the 8 × 8 matrix on each of 4 processes is as follows:
Process 0: (0,0) in process grid

\[ A^{(0)} = \begin{bmatrix} 11 & 12 & 15 & 16 \\ 21 & 22 & 25 & 26 \\ 51 & 52 & 55 & 56 \\ 61 & 62 & 65 & 66 \end{bmatrix}_{4 \times 4} \]

\( A^{(0)} \) is the 4 \times 4 part of the A matrix which is distributed to process 0 = (0,0).

Process 1: (0,1) in process grid

\[ A^{(1)} = \begin{bmatrix} 13 & 14 & 17 & 18 \\ 23 & 24 & 27 & 28 \\ 53 & 54 & 57 & 58 \\ 63 & 64 & 67 & 68 \end{bmatrix}_{4 \times 4} \]

\( A^{(1)} \) is the 4 \times 4 part of the A matrix which is distributed to process 1 = (0,1).

Process 2: (1,0) in process grid

\[ A^{(2)} = \begin{bmatrix} 31 & 32 & 35 & 36 \\ 41 & 42 & 45 & 46 \\ 71 & 72 & 75 & 76 \\ 81 & 82 & 85 & 86 \end{bmatrix}_{4 \times 4} \]

\( A^{(2)} \) is the 4 \times 4 part of the A matrix which is distributed to process 2 = (1,0).

Process 3: (1,1) in process grid

\[ A^{(3)} = \begin{bmatrix} 33 & 34 & 37 & 38 \\ 43 & 44 & 47 & 48 \\ 73 & 74 & 77 & 78 \\ 83 & 84 & 87 & 88 \end{bmatrix}_{4 \times 4} \]

\( A^{(3)} \) is the 4 \times 4 part of the A matrix which is distributed to process 3 = (1,1).

Let \( Y \) be a vector of size 8, partitioned into 2 \times 2 blocks, and mapped onto a 2 \times 2 process grid. The Figure 2.9 shows the partitioned vector and the process coordinates in process grid for each block.
\[ P_c \Rightarrow 0 \quad P_c \Rightarrow 0 \]

\[
P_r \downarrow 0 \quad 0 \begin{pmatrix}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8
\end{pmatrix} \quad P_r \downarrow 0 \begin{pmatrix}
(0,0) \\
(1,0) \\
(0,0) \\
(1,0)
\end{pmatrix}
\]

Figure 2.9: An example input \( Y \) vector partitioned using \( 2 \times 2 \) blocks and the corresponding process coordinates for each block.

NOTE: Since \( Y \) is a single column vector, the \( 2 \times 2 \) blocks are \( 2 \times 1 \) blocks. So the \( Y \) vector is actually partitioned into \( 2 \times 1 \) blocks.

Therefore, the \( Y \) vector is placed 2 components at a time on processes \((0,0)\), \((1,0)\), \((0,0)\) and \((1,0)\) alternately, that is in a row-cyclic manner. The data layout of vector \( Y \) on each of the 4 processes is as follows:

**Process 0**: \((0,0)\) in process grid

\[
Y^{(0)} = \begin{bmatrix}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8
\end{bmatrix}_{4 \times 1}
\]

\(Y^{(0)}\) is a vector of size 4 which is distributed to process 0 = \((0,0)\)

**Process 1**: \((0,1)\) in process grid

Nothing is distributed to process 1

**Process 2**: \((1,0)\) in process grid

\[
Y^{(2)} = \begin{bmatrix}
3 \\
4 \\
5 \\
7 \\
8
\end{bmatrix}_{4 \times 1}
\]

\(Y^{(2)}\) is a vector of size 4 which is distributed to process 2 = \((1,0)\).
Process 3: (1,1) in process grid

Nothing is distributed to process 3.

Similarly, let \( Z \) be an input row vector of size 8, partitioned into \( 2 \times 2 \) blocks, and mapped onto a \( 2 \times 2 \) process grid. The Figure 2.10 shows the partitioned vector and the process coordinates in the process grid for each block.

\[
P_r \downarrow \quad P_c \Rightarrow \begin{array}{cccc}
0 & 1 & 0 & 1 \\
\end{array}
\]

\[
0 \begin{pmatrix}
(0,0) & (0,1) & (0,0) & (0,1)
\end{pmatrix}
\]

Figure 2.10: An example input \( Z \) vector partitioned using \( 2 \times 2 \) block and the corresponding process coordinates for each block

Since there is only one row, the \( Z \) vector is placed 2 components at a time on processes \((0,0), (0,1), (0,0) \) and \((0,1)\) alternatively, that is, in a column-cyclic manner. The data layout of vector \( Z \) on each of the 4 processes is as follows:

Process 0: \((0,0)\) in process grid

\[
Z^{(0)} = \begin{bmatrix}
1 & 2 & 5 & 6
\end{bmatrix}_{1 \times 4}
\]

\(Z^{(0)}\) is a vector of size 4 which is distributed to process \(0 = (0,0)\).

Process 1: \((0,1)\) in process grid

\[
Z^{(1)} = \begin{bmatrix}
3 & 4 & 7 & 8
\end{bmatrix}_{1 \times 4}
\]

\(Z^{(1)}\) is a vector of size 4 which is distributed to process \(1 = (0,1)\).
Process 2: (1,0) in process grid
Nothing is distributed to process 2.

Process 3: (1,1) in process grid
Nothing is distributed to process 3.

We now describe how our routine PDGEMVT can be expected to achieve cache-efficiency on each process of the process grid, using the above block-cyclic data layout for \( A, Y, Z \).

Let \( k = 2 \) be the number of columns in each column block of that part of the distributed matrix which resides on each process. Then the number of column blocks, \( b \), on each process is computed as

\[
b = \frac{NQ}{k} = \frac{4}{2} = 2.
\]

In the general situation of a large dense \( M \times N \) matrix, the idea is to select \( k \) so that the \( MP \times k \) matrix elements in each column block fill the available \( L_2 \) cache.

Step 0: (Preliminary steps done in the driver program for PDGEMVT)

(a) The output \( W^{(p)} \) vector is initialized to zero in all processes

\[
W^{(p)} = 0
\]

(b) Send the \( Y \) vector which resides on the processes 0 and 2 to processes 1 and 3:

Process 0 sends \( Y^{(0)} = \begin{bmatrix} 1 \\ 2 \\ 5 \\ 6 \end{bmatrix} \) to Process 1 to make \( Y^{(1)} \)

Process 2 sends \( Y^{(2)} = \begin{bmatrix} 3 \\ 4 \\ 7 \\ 8 \end{bmatrix} \) to Process 3 to make \( Y^{(3)} \)
Step 1: Copy the $Z^{(p)}$ vector to $X^{(p)}$

Process 0 Computes:

$$X^{(0)} = (Z^{(0)})^T = \begin{bmatrix} 1 \\ 2 \\ 5 \\ 6 \end{bmatrix}$$

Process 1 Computes:

$$X^{(1)} = (Z^{(1)})^T = \begin{bmatrix} 3 \\ 4 \\ 7 \\ 8 \end{bmatrix}$$

Next perform the \textit{cache-efficient} loop over $i=1,b$.

\textbf{For} $i = 1$:

Step 2 ($i=1$): Generation of partial $X_1$ vectors on each process.

Process 0 Computes $X_1^{(0)}$:

$$X_1^{(0)} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \end{bmatrix} = (A_1^{(0)})^T \begin{bmatrix} Y^{(0)} \\ X_1^{(0)} \end{bmatrix}$$

$$= \begin{bmatrix} 11 & 21 & 51 & 61 \\ 12 & 22 & 52 & 62 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 675 \\ 690 \end{bmatrix}$$

Process 2 Computes $X_1^{(2)}$:

$$X_1^{(2)} = \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} = (A_1^{(2)})^T \begin{bmatrix} Y^{(2)} \end{bmatrix}$$

$$= \begin{bmatrix} 31 & 41 & 71 & 81 \\ 32 & 42 & 72 & 82 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 1402 \\ 1424 \end{bmatrix}$$
Process 1 Computes $X_1^{(1)}$:

$$X_1^{(1)} = \begin{bmatrix} x_3^{(1)} \\ x_4^{(1)} \end{bmatrix} = (A_1^{(1)})^T Y^{(1)} + X_1^{(1)}$$

$$= \begin{bmatrix} 13 & 23 & 53 & 63 \\ 14 & 24 & 54 & 64 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 705 \\ 720 \end{bmatrix}$$

Process 3 Computes $X_1^{(3)}$:

$$X_1^{(3)} = \begin{bmatrix} x_3^{(3)} \\ x_4^{(3)} \end{bmatrix} = (A_1^{(3)})^T Y^{(3)}$$

$$= \begin{bmatrix} 33 & 43 & 73 & 83 \\ 34 & 44 & 74 & 84 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 1446 \\ 1468 \end{bmatrix}$$

**Step 3** (i=1): Communication between processes in each column communicator.

Process 0 and 2 sums the $X_1$ vectors and overwrites $X_1$ on each of these processes.

$$X_1^{(0)} = X_1^{(2)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = X_1^{(0)} + X_1^{(2)}$$

$$= \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \end{bmatrix} + \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} = \begin{bmatrix} 675 \\ 690 \end{bmatrix} + \begin{bmatrix} 1402 \\ 1424 \end{bmatrix} = \begin{bmatrix} 2077 \\ 2114 \end{bmatrix}$$

Process 1 and 3 sums the $X_1$ vectors and overwrites $X_1$ on each of these processes.

$$X_1^{(1)} = X_1^{(3)} = \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} = X_1^{(1)} + X_1^{(3)}$$

$$= \begin{bmatrix} x_3^{(1)} \\ x_4^{(1)} \end{bmatrix} + \begin{bmatrix} x_3^{(3)} \\ x_4^{(3)} \end{bmatrix} = \begin{bmatrix} 705 \\ 720 \end{bmatrix} + \begin{bmatrix} 1446 \\ 1468 \end{bmatrix} = \begin{bmatrix} 2151 \\ 2188 \end{bmatrix}$$
**Step 4** (i=1): Generation of partial $W$ vectors on each process.

**Process 0 Computes $W^{(0)}$:**

$$W^{(0)} = \begin{bmatrix} w^{(0)}_1 \\ w^{(0)}_2 \\ w^{(0)}_5 \\ w^{(0)}_6 \end{bmatrix} = A^{(0)} X^{(0)} = \begin{bmatrix} 11 & 12 \\ 21 & 22 \\ 51 & 52 \\ 61 & 62 \end{bmatrix} \begin{bmatrix} 2077 \\ 2114 \end{bmatrix} = \begin{bmatrix} 48215 \\ 90125 \\ 215855 \\ 257765 \end{bmatrix}$$

**Process 2 Computes $W^{(2)}$:**

$$W^{(2)} = \begin{bmatrix} w^{(2)}_3 \\ w^{(2)}_4 \\ w^{(2)}_7 \\ w^{(2)}_8 \end{bmatrix} = A^{(2)} X^{(2)} = \begin{bmatrix} 31 & 32 \\ 41 & 42 \\ 71 & 72 \\ 81 & 82 \end{bmatrix} \begin{bmatrix} 2077 \\ 2114 \end{bmatrix} = \begin{bmatrix} 132035 \\ 173945 \\ 299675 \\ 341585 \end{bmatrix}$$

**Process 1 Computes $W^{(1)}$:**

$$W^{(1)} = \begin{bmatrix} w^{(1)}_1 \\ w^{(1)}_2 \\ w^{(1)}_5 \\ w^{(1)}_6 \end{bmatrix} = A^{(1)} X^{(1)} = \begin{bmatrix} 13 & 14 \\ 23 & 24 \\ 53 & 54 \\ 63 & 64 \end{bmatrix} \begin{bmatrix} 2151 \\ 2188 \end{bmatrix} = \begin{bmatrix} 58595 \\ 101885 \\ 232155 \\ 275545 \end{bmatrix}$$

**Process 3 Computes $W^{(3)}$:**

$$W^{(3)} = \begin{bmatrix} w^{(3)}_3 \\ w^{(3)}_4 \\ w^{(3)}_7 \\ w^{(3)}_8 \end{bmatrix} = A^{(3)} X^{(3)} = \begin{bmatrix} 33 & 34 \\ 43 & 44 \\ 73 & 74 \\ 83 & 84 \end{bmatrix} \begin{bmatrix} 2151 \\ 2188 \end{bmatrix} = \begin{bmatrix} 145375 \\ 188765 \\ 318935 \\ 362325 \end{bmatrix}$$

**For i = 2:**

Flush the $L_2$ cache of $A^{(p)}_1, p=0,1,2,3$, and reload with $A^{(p)}_2, p=0,1,2,3$ for the next part of the two matrix-vector multiplications, $X = A^T * Y + Z$ and $W = A * X$.

**Step 2** (i=2): Generation of partial $X_2$ vectors on each process.

**Process 0 Computes $X_2^{(0)}$:**

$$X_2^{(0)} = \begin{bmatrix} x^{(0)}_5 \\ x^{(0)}_6 \end{bmatrix} = (A^{(0)}_2)^T Y^{(0)} + X_2^{(0)}$$
\[
\begin{bmatrix}
15 & 25 & 55 & 65 \\
16 & 26 & 56 & 66
\end{bmatrix}
+ \begin{bmatrix}
1 \\
2 \\
5 \\
6
\end{bmatrix} = \begin{bmatrix}
735 \\
750
\end{bmatrix}
\]

Process 2 Computes \(X_2^{(2)}\):
\[
X_2^{(2)} = \begin{bmatrix}
x_5^{(2)} \\
x_6^{(2)}
\end{bmatrix} = (A_2^{(2)})^T \ Y^{(2)}
\]
\[
= \begin{bmatrix}
35 & 45 & 75 & 85 \\
36 & 46 & 76 & 86
\end{bmatrix}
\begin{bmatrix}
3 \\
4 \\
7 \\
8
\end{bmatrix} = \begin{bmatrix}
1490 \\
1512
\end{bmatrix}
\]

Process 1 Computes \(X_2^{(1)}\):
\[
X_2^{(1)} = \begin{bmatrix}
x_7^{(1)} \\
x_8^{(1)}
\end{bmatrix} = (A_2^{(1)})^T \ Y^{(1)} + X_2^{(1)}
\]
\[
= \begin{bmatrix}
17 & 27 & 57 & 67 \\
18 & 28 & 58 & 68
\end{bmatrix}
\begin{bmatrix}
1 \\
2 \\
5 \\
6
\end{bmatrix} + \begin{bmatrix}
7 \\
8
\end{bmatrix} = \begin{bmatrix}
765 \\
780
\end{bmatrix}
\]

Process 3 Computes \(X_2^{(3)}\):
\[
X_2^{(3)} = \begin{bmatrix}
x_7^{(3)} \\
x_8^{(3)}
\end{bmatrix} = (A_2^{(3)})^T \ Y^{(3)}
\]
\[
= \begin{bmatrix}
37 & 47 & 77 & 87 \\
38 & 48 & 78 & 88
\end{bmatrix}
\begin{bmatrix}
3 \\
4 \\
7 \\
8
\end{bmatrix} = \begin{bmatrix}
1534 \\
1556
\end{bmatrix}
\]

Step 3 (i=2): Communication between processes in each column communicator.

Process 0 and 2 sums the \(X_2\) vectors and overwrites \(X_2\) on each of these processes.
\[
X_2^{(0)} = X_2^{(2)} = \begin{bmatrix}
x_5 \\
x_6
\end{bmatrix} = X_2^{(0)} + X_2^{(2)}
\]

46
\[
\begin{bmatrix}
x_5^{(0)} \\
x_6^{(0)}
\end{bmatrix} +
\begin{bmatrix}
x_5^{(2)} \\
x_6^{(2)}
\end{bmatrix}
= 
\begin{bmatrix}
735 \\
750
\end{bmatrix} +
\begin{bmatrix}
1490 \\
1512
\end{bmatrix}
= 
\begin{bmatrix}
2225 \\
2262
\end{bmatrix}
\]

Process 1 and 3 sums the \( X_2 \) vectors and overwrites \( X_2 \) on each of these processes.

\[
X_2^{(1)} = X_2^{(3)} = \begin{bmatrix} x_7 \\ x_8 \end{bmatrix} = X_2^{(1)} + X_2^{(3)}
= \begin{bmatrix} x_7^{(1)} \\ x_8^{(1)} \end{bmatrix} +
\begin{bmatrix} x_7^{(3)} \\ x_8^{(3)} \end{bmatrix}
= \begin{bmatrix} 765 \\ 780 \end{bmatrix} +
\begin{bmatrix} 1534 \\ 1556 \end{bmatrix}
= \begin{bmatrix} 2299 \\ 2336 \end{bmatrix}
\]

**Step 4** \((i=2)\): Generation of partial \( W \) vectors on each process.

**Process 0 Computes \( W^{(0)} \):**

\[
W^{(0)} = \begin{bmatrix}
w_1^{(0)} \\
w_2^{(0)} \\
w_5^{(0)} \\
w_6^{(0)}
\end{bmatrix}
= W^{(0)} + A_2^{(0)} X_2^{(0)}
= \begin{bmatrix}
48215 \\
90125 \\
215855 \\
257765
\end{bmatrix} +
\begin{bmatrix}
15 \\
25 \\
55 \\
65
\end{bmatrix} \begin{bmatrix}
2225 \\
2262
\end{bmatrix}
= \begin{bmatrix}
117782 \\
204562 \\
464902 \\
551682
\end{bmatrix}
\]

**Process 2 Computes \( W^{(2)} \):**

\[
W^{(2)} = \begin{bmatrix}
w_3^{(2)} \\
w_4^{(2)} \\
w_7^{(2)} \\
w_8^{(2)}
\end{bmatrix}
= W^{(2)} + A_2^{(2)} X_2^{(2)}
= \begin{bmatrix}
132035 \\
173945 \\
299675 \\
341585
\end{bmatrix} +
\begin{bmatrix}
35 \\
45 \\
75 \\
85
\end{bmatrix} \begin{bmatrix}
2225 \\
2262
\end{bmatrix}
= \begin{bmatrix}
291342 \\
378122 \\
638462 \\
725242
\end{bmatrix}
\]

47
Process 1 Computes $W^{(1)}$:

$$W^{(1)} = \begin{bmatrix} w_{1}^{(1)} \\ w_{2}^{(1)} \\ w_{5}^{(1)} \\ w_{6}^{(1)} \end{bmatrix} = W^{(1)} + A_{2}^{(1)} X_{2}^{(1)}$$

$$= \begin{bmatrix} 58595 \\ 101985 \\ 232155 \\ 275545 \end{bmatrix} + \begin{bmatrix} 17 & 18 \\ 27 & 28 \\ 57 & 58 \\ 67 & 68 \end{bmatrix} \begin{bmatrix} 2299 \\ 2336 \end{bmatrix} = \begin{bmatrix} 139726 \\ 229466 \\ 498686 \\ 588426 \end{bmatrix}$$

Process 3 Computes $W^{(3)}$:

$$W^{(3)} = \begin{bmatrix} w_{3}^{(3)} \\ w_{4}^{(3)} \\ w_{7}^{(3)} \\ w_{8}^{(3)} \end{bmatrix} = W^{(3)} + A_{2}^{(3)} X_{2}^{(3)}$$

$$= \begin{bmatrix} 145375 \\ 188765 \\ 318935 \\ 362325 \end{bmatrix} + \begin{bmatrix} 37 & 38 \\ 47 & 48 \\ 77 & 78 \\ 87 & 88 \end{bmatrix} \begin{bmatrix} 2299 \\ 2336 \end{bmatrix} = \begin{bmatrix} 319206 \\ 408946 \\ 678166 \\ 767906 \end{bmatrix}$$

Step 5: Communication between processes in each row communicator, to generate the (complete) $W$ vectors on each process.

Process 0 and 1 sums its $W$ vector and overwrites $W$ on each of these processes.

$$W^{(0)} = W^{(1)} = \begin{bmatrix} w_{1} \\ w_{2} \\ w_{5} \\ w_{6} \end{bmatrix} = W^{(0)} + W^{(1)}$$

$$= \begin{bmatrix} w_{1}^{(0)} \\ w_{2}^{(0)} \\ w_{5}^{(0)} \\ w_{6}^{(0)} \end{bmatrix} + \begin{bmatrix} w_{1}^{(1)} \\ w_{2}^{(1)} \\ w_{5}^{(1)} \\ w_{6}^{(1)} \end{bmatrix} = \begin{bmatrix} 117782 \\ 204562 \\ 464902 \\ 551682 \end{bmatrix} + \begin{bmatrix} 139726 \\ 229466 \\ 498686 \\ 588426 \end{bmatrix} = \begin{bmatrix} 257508 \\ 434028 \\ 963588 \\ 1140108 \end{bmatrix}$$

48
Process 2 and 3 sums its $W$ vector and overwrites $W$ on each of these processes.

$$W^{(2)} = W^{(3)} = \begin{bmatrix} w_3 \\ w_4 \\ w_7 \\ w_8 \end{bmatrix} = W^{(2)} + W^{(3)}$$

$$= \begin{bmatrix} w_3^{(2)} \\ w_4^{(2)} \\ w_7^{(2)} \\ w_8^{(2)} \end{bmatrix} + \begin{bmatrix} w_3^{(3)} \\ w_4^{(3)} \\ w_7^{(3)} \\ w_8^{(3)} \end{bmatrix} = \begin{bmatrix} 291342 \\ 378122 \\ 638462 \\ 725242 \end{bmatrix} + \begin{bmatrix} 319206 \\ 408946 \\ 678166 \\ 767906 \end{bmatrix} = \begin{bmatrix} 610548 \\ 787068 \\ 1316628 \\ 1493148 \end{bmatrix}$$

With the completion of Step 5, the action of subroutine PDGEMVT is complete. Four of the components of the $W$ vector reside on Process 0 and 1, and the remaining four components reside on Process 2 and 3. All components are complete, however, so generation of the global $W$ vector on one process only requires one additional communication after the call to PDGEMVT.

**Step 6: Generation of global $W$ vector.** This is done in the driver program for PDGEMVT, as the last task of the two matrix-vector multiplications.

Process 2 sends its $W$ vector to process 0 to make the (complete) $W$ vector of size 8.

$$W = \begin{bmatrix} 257508 \\ 434028 \\ 610548 \\ 787068 \\ 963588 \\ 1140108 \\ 1316628 \\ 1493148 \end{bmatrix}$$
Chapter 3

Parallel Cache Efficient GEMVT using Column Cyclic data distribution

3.1 Introduction

This version of parallel GEMVT uses BLACS and BLAS routines to perform the GEMVT operations in parallel in a manner similar to the block cyclic cache efficient PDGEMVT of Chapter 2. The driver program pdgemvtdriver.F makes a call to a newly developed routine PDGEMVT in the file pdgemvt.c.f, which makes two calls to the BLAS routine DGEMV from the ATLAS library. The driver program uses the “Column Cyclic” data distribution to distribute the input matrix and input vector in order to perform PDGEMVT.

3.2 Process Grid

\[
\begin{array}{c|c}
0 & 1 \\
\hline
0 & 0 & 1 \\
\end{array}
\]

Figure 3.1: Four processes mapped onto a $1 \times 2$ process grid

In this version, one-dimensional process grid is used to distribute the input matrix and vector in column cyclic fashion. This grid will have 1 process row and $P$ process
columns, where $P$ is the total number of processes. A $1 \times 2$ process grid ($P = 2$), shown in Figure 3.1, is used to illustrate the parallel GEMVT using column cyclic data distribution of a $8 \times 8$ matrix in next section.

### 3.3 “Column Cyclic” data distribution using a $8 \times 8$ matrix

The column cyclic data distribution depends on the total number of processes ($P$) in $1 \times P$ process grid and the column block parameter $NB$. Since the number of rows in the process grid is 1, the row block parameter $MB$ will be $M$, where $M \times N$ is the global matrix size. The column cyclic data distribution using a $8 \times 8$ $A$ matrix and $Z$ and $Y$ vectors of size 8 mapped onto $1 \times 2$ process grid (shown in Figure 3.1) is illustrated in this section.

Let $MP$ and $NQ$ be the number of rows and columns of the distributed $A$ matrix on each process. Thus, $MP \times NQ$ is the size of the matrix on each process. For $8 \times 8$ matrix mapped onto $1 \times 2$ process grid, $MP = M = 8$ and $NQ = 4$.

Let $A$ be a $8 \times 8$ matrix,

$$A = \begin{pmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\
  a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\
  a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\
  a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\
  a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\
  a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\
  a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\
  a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88}
\end{pmatrix}.$$

The coordinates of each process in the $1 \times 2$ process grid shown in Figure 3.2 are:

- Process 0: (0,0)
- Process 1: (0,1)

Let us partition the $8 \times 8$ matrix $A$ into $8 \times 2$ blocks, mapped onto a $1 \times 2$ process grid. The partitioned matrix is shown in the Figure 3.3
Figure 3.2: Process coordinates in a $1 \times 2$ process grid

\[ P_c \Rightarrow \begin{array}{cccc}
0 & 1 & 0 & 1 \\
0 & (0,0) & (0,1)
\end{array} \]

Figure 3.3: $8 \times 8$ matrix partitioned into $8 \times 2$ blocks in column cyclic fashion

The mapping of the $8 \times 2$ blocks onto the process grid is “column cyclic” in column direction, that is, every other 2 column blocks across each column goes to successive processes in the process grid ($P_c$ alternating from 0 to 1). The process coordinates for the $8 \times 8$ matrix partitioned by $8 \times 2$ blocks using the column cyclic data distribution is shown in the Figure 3.4

Therefore,

- Process 0 gets the $1^{st}$ and $3^{rd}$ column-blocks.

\[ P_r \downarrow P_c \Rightarrow \begin{array}{cccc}
0 & 1 & 0 & 1 \\
0 & (0,0) & (0,1) & (0,0) & (0,1)
\end{array} \]

Figure 3.4: Process coordinates for $8 \times 8$ matrix, partitioned by $8 \times 2$ blocks

52
• Process 1 gets the 2\textsuperscript{nd} and 4\textsuperscript{th} column-blocks.

Thus, the number of rows ($MP$) and number of columns ($NQ$) on each process are 8 and 4 respectively.

### 3.3.1 Data layout of $8 \times 8$ matrix on each process:

**Process 0:** $(0,0)$ in process grid

$$A^{(0)} = \begin{bmatrix}
    a_{11} & a_{12} & a_{15} & a_{16} \\
    a_{21} & a_{22} & a_{25} & a_{26} \\
    a_{31} & a_{32} & a_{35} & a_{36} \\
    a_{41} & a_{42} & a_{45} & a_{46} \\
    a_{51} & a_{52} & a_{55} & a_{56} \\
    a_{61} & a_{62} & a_{65} & a_{66} \\
    a_{71} & a_{72} & a_{75} & a_{76} \\
    a_{81} & a_{82} & a_{85} & a_{86}
\end{bmatrix}_{8 \times 4}$$

$A^{(0)}$ is the $8 \times 4$ part of the A matrix which is distributed to process $0 = (0,0)$.

**Process 1:** $(0,1)$ in process grid

$$A^{(1)} = \begin{bmatrix}
    a_{13} & a_{14} & a_{17} & a_{18} \\
    a_{23} & a_{24} & a_{27} & a_{28} \\
    a_{33} & a_{34} & a_{37} & a_{38} \\
    a_{43} & a_{44} & a_{47} & a_{48} \\
    a_{53} & a_{54} & a_{57} & a_{58} \\
    a_{63} & a_{64} & a_{67} & a_{68} \\
    a_{73} & a_{74} & a_{77} & a_{78} \\
    a_{83} & a_{84} & a_{87} & a_{88}
\end{bmatrix}_{8 \times 4}$$

$A^{(1)}$ is the $8 \times 4$ part of the A matrix which is distributed to process $1 = (0,1)$. 
3.3.2 Data layout of $8 \times 1$ input vector on each process:

Let $Y$ be the vector of size 8. Since there is only one column and the row block size is 8, the complete vector is placed in process $0 = (0,0)$. The distribution of the input $Y$ vector is shown in Figure 3.5.

Thus:

- Process 0 gets the 1st column-block.
- Process 1 doesn’t receive any column block.

**Process 0: (0,0) in process grid**

$$Y^{(0)} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix}$$

$Y^{(0)}$ is a vector of size 8 which is distributed to process $0 = (0,0)$.

**Process 0: (0,1) in process grid**

Nothing is distributed to process 1.
3.3.3 Data layout of $1 \times 8$ input vector $Z$ on each process:

Let $Z$ be a row vector of size 8, partitioned into $8 \times 2$ blocks, mapped onto a $1 \times 2$ process grid. The Figure 3.10 shows the partitioned vector using $8 \times 2$ blocks and the process coordinates for each block.

Since there is only one row, the $Z$ vector is placed 2 components at a time on processes $(0,0)$, $(0,1)$, $(0,0)$ and $(0,1)$ alternately, that is in a column-cyclic manner.

\[
P_r \downarrow \quad P_c \Rightarrow \begin{array}{cccc}
0 & 1 & 0 & 1 \\
\end{array}
\]

\[
0 \quad \begin{pmatrix}
(0,0) & (0,1) & (0,0) & (0,1)
\end{pmatrix}
\]

Figure 3.6: Input $Z$ vector partitioned into $8 \times 2$ blocks and the corresponding process coordinates for each block.

Thus:

- Process 0 gets the 1st and 3rd blocks.
- Process 1 gets the 2nd and 4th blocks.

**Process 0:** $(0,0)$ in process grid

\[
Z^{(0)} = \begin{bmatrix}
z_1 & z_2 & z_5 & z_6
\end{bmatrix}_{1 \times 4}
\]

$Z^{(0)}$ is a vector of size 4 which is distributed to process $0 = (0,0)$. 

55
Process 1: (0,1) in process grid

\[ Z^{(1)} = \begin{bmatrix} z_3 & z_4 & z_7 & z_8 \end{bmatrix} 1 \times 4 \]

\( Z^{(1)} \) is a vector of size 4 which is distributed to process 1 = (0,1).

### 3.4 Computations and Communications

The computations on each process and the communication required between processes to accomplish PDGEMVT for a \(8 \times 8\) matrix partitioned into \(8 \times 2\) blocks and mapped onto \(1 \times 2\) process grid is described in this section.

The cache efficient PDGEMVT makes use of column blocking on the distributed matrix \(A\) on each process to achieve cache efficiency. On each process, the idea is to implement two simultaneous matrix-vector multiplications in the same way it was done in serial on a single processor. We have already seen that the size of the \(A\) matrix \((MP \times NQ)\) distributed to each process is \(8 \times 4\) for our \(8 \times 8\) global matrix, so we want to implement cache-efficiency on column blocks of each of the \(8 \times 4\) matrices on each process.

Let us assume, \(k = 2\), is the number of columns in each column block. Let \(b\) be total number of blocks and \(b \times k = NQ\). In general, the block size \(k\) will be dependent on \(NQ\). Since \(NQ = 4\) and \(k = 2\), \(b = NQ/k = 2\) on each process, where \(NQ\) is the
number of columns on each process. The idea, of course, is to select the block size \( k \) such that \( M \times k \) matrix elements fit into the \( L_2 \) cache and generally fill the \( L_2 \) cache on each process; we can then expect that the \( M \times k \) matrix elements in one block will remain in \( L_2 \) cache as the 2 matrix-vector multiplications are executed, where \( M \) is the number of rows on each process. The column blocking on process 0 for the column cyclic distributed matrix is shown in Figure 3.7.

The column cyclic cache efficient PDGEMVT algorithm can be described as:

**Step 1:** Copy the \( Z^{(p)} \) vector to \( X^{(p)} \)

\[
X^{(p)} \leftarrow Z^{(p)}
\]

For \( i = 1 \) to \( b \), do in parallel (\( b = NQ/k \))

**Step 2:** Generate the (complete) \( X^{(p)}_i \) vectors on each process:

\[
X^{(p)}_i = \alpha \times (A_i^{(p)})^T \times Y^{(p)} + X^{(p)}_i \quad \text{(first GEMV call)}
\]

**NOTE:** The usual step 3 from the Block Cyclic code is omitted because all components of \( Y \) vector are available on each process so that the \( X^{(p)}_i \) vectors are complete on each process. Hence this step is not needed in parallel implementation using the column cyclic data distribution.

**Step 3:** Make use of the (complete) \( X^{(p)}_i \) vectors to generate all \( M \) components of the partial \( W^{(p)} \) vector on each process. The \( W^{(p)} \) vector is accumulated with the previous \( W^{(p)} \) vector from blocks 1, 2, ...\( i - 1 \):

\[
W^{(p)} = \alpha \times A_i^{(p)} \times X^{(p)}_i + \beta \times W^{(p)} \quad \text{(second GEMV call)}
\]

End for loop (Now, increment \( i \) and repeat steps 2 - 3).

**Step 4:** Sum the partial \( W^{(p)} \) vectors generated on all processes \( p \) and overwrite \( W^{(p)} \) with this sum on each of these processes.
The above pseudo-code can be generalized to any $M \times N$ global matrix $A$ with any $M \times NB$ choice of blocks and any process grid $1 \times P_c$, and, in general, $b = b^{(p)}$ is different on each process, owing to different block sizes $k = k_b$. We now describe in more detail the above algorithm for the $8 \times 8$ global matrix.

**Step 0:** (Preliminary steps done in the driver for PDGEMVT):

(a) The output $W^{(p)}$ vector is initialized to zero in all processes

\[ W^{(p)} = 0 \]

(b) Send the $Y$ vector which resides on the processes 0 to processes 1.

Process 0 sends $Y^{(0)} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix}$ to Process 1 to make $Y^{(1)}$

**Step 1:** Copy the $Z^{(p)}$ vector to $X^{(p)}$

Process 0 Computes:

\[
X^{(0)} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \\ x_3^{(0)} \\ x_4^{(0)} \\ x_5^{(0)} \\ x_6^{(0)} \end{bmatrix} = Z^{(0)} = \begin{bmatrix} z_1^{(0)} \\ z_2^{(0)} \\ z_3^{(0)} \\ z_4^{(0)} \\ z_5^{(0)} \end{bmatrix}
\]

Process 1 Computes:

\[
X^{(1)} = \begin{bmatrix} x_3^{(1)} \\ x_4^{(1)} \\ x_5^{(1)} \\ x_7^{(1)} \\ x_8^{(1)} \end{bmatrix} = Z^{(1)} = \begin{bmatrix} z_3^{(1)} \\ z_4^{(1)} \\ z_5^{(1)} \\ z_7^{(1)} \\ z_8^{(1)} \end{bmatrix}
\]
For $i = 1$:

**Step 2** ($i=1$): Generation of complete $X_1$ vectors on each process.

**Process 0 Computes:**

$$X^{(0)}_1 = \begin{bmatrix} x^{(0)}_1 \\ x^{(0)}_2 \end{bmatrix} = (A^{(0)}_1)^T Y^{(0)} + X^{(0)}_1$$

$$= \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} & a_{51} & a_{61} & a_{71} & a_{81} \\ a_{12} & a_{22} & a_{32} & a_{42} & a_{52} & a_{62} & a_{72} & a_{82} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} + \begin{bmatrix} x^{(0)}_1 \\ x^{(0)}_2 \end{bmatrix}$$

**Process 1 Computes:**

$$X^{(1)}_1 = \begin{bmatrix} x^{(1)}_3 \\ x^{(1)}_4 \end{bmatrix} = (A^{(1)}_1)^T Y^{(1)} + X^{(1)}_1$$

$$= \begin{bmatrix} a_{13} & a_{23} & a_{33} & a_{43} & a_{53} & a_{63} & a_{73} & a_{83} \\ a_{14} & a_{24} & a_{34} & a_{44} & a_{54} & a_{64} & a_{74} & a_{84} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} + \begin{bmatrix} x^{(1)}_3 \\ x^{(1)}_4 \end{bmatrix}$$

**Step 3** ($i=1$): Generation of partial $W$ vectors on each process.

**Process 0 Computes:**

$$W^{(0)} = \begin{bmatrix} w^{(0)}_1 \\ w^{(0)}_2 \\ w^{(0)}_3 \\ w^{(0)}_4 \\ w^{(0)}_5 \\ w^{(0)}_6 \\ w^{(0)}_7 \\ w^{(0)}_8 \end{bmatrix} = A^{(0)}_1 X^{(0)}_1 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \\ a_{51} & a_{52} \\ a_{61} & a_{62} \\ a_{71} & a_{72} \\ a_{81} & a_{82} \end{bmatrix} \begin{bmatrix} x^{(0)}_1 \\ x^{(0)}_2 \end{bmatrix}$$
Process 1 Computes:

\[ W^{(1)} = \begin{bmatrix} w_1^{(1)} \\ w_2^{(1)} \\ w_3^{(1)} \\ w_4^{(1)} \\ w_5^{(1)} \\ w_6^{(1)} \\ w_7^{(1)} \\ w_8^{(1)} \end{bmatrix} = A^{(1)} \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \\ x_4^{(1)} \end{bmatrix} = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \\ a_{33} & a_{34} \\ a_{43} & a_{44} \\ a_{53} & a_{54} \\ a_{63} & a_{64} \\ a_{73} & a_{74} \\ a_{83} & a_{84} \end{bmatrix} \begin{bmatrix} x_3^{(1)} \\ x_4^{(1)} \end{bmatrix} \]

For \( i = 2 \):

NOTE: now the \( A^{(p)}_1 \) matrices are flushed from the \( L_2 \) cache, and Step 2 brings the \( A^{(p)}_2 \) matrices into the \( L_2 \) cache on each process.

Step 2 (i=2): Generation of complete \( X_2 \) vectors on each process.

Process 0 Computes:

\[ X_2^{(0)} = \begin{bmatrix} x_5^{(0)} \\ x_6^{(0)} \end{bmatrix} = (A^{(0)}_2)^T Y^{(0)} + X_2^{(0)} \]

\[ = \begin{bmatrix} a_{15} & a_{25} & a_{35} & a_{45} & a_{55} & a_{65} & a_{75} & a_{85} \\ a_{16} & a_{26} & a_{36} & a_{46} & a_{56} & a_{66} & a_{76} & a_{86} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} + \begin{bmatrix} x_5^{(0)} \\ x_6^{(0)} \end{bmatrix} \]

Process 1 Computes:

\[ X_2^{(1)} = \begin{bmatrix} x_7^{(1)} \\ x_8^{(1)} \end{bmatrix} = (A^{(1)}_2)^T Y^{(1)} + X_2^{(1)} \]
\[
\begin{bmatrix}
  a_{17} & a_{27} & a_{37} & a_{47} & a_{57} & a_{67} & a_{77} & a_{87} \\
  a_{18} & a_{28} & a_{38} & a_{48} & a_{58} & a_{68} & a_{78} & a_{88}
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
y_5 \\
y_6 \\
y_7 \\
y_8
\end{bmatrix}
+ \begin{bmatrix}
x_7^{(0)} \\
x_8^{(0)}
\end{bmatrix}
\]

Step 3 (i=2): Generation of partial \( W \) vectors on each process.

Process 0 Computes:

\[
W^{(0)} = \begin{bmatrix}
w_1^{(0)} \\
w_2^{(0)} \\
w_3^{(0)} \\
w_4^{(0)} \\
w_5^{(0)} \\
w_6^{(0)} \\
w_7^{(0)} \\
w_8^{(0)}
\end{bmatrix} = W^{(0)} + A_2^{(0)} X_2^{(0)} = \begin{bmatrix}
w_1^{(0)} \\
w_2^{(0)} \\
w_3^{(0)} \\
w_4^{(0)} \\
w_5^{(0)} \\
w_6^{(0)} \\
w_7^{(0)} \\
w_8^{(0)}
\end{bmatrix} + \begin{bmatrix}
a_{15} & a_{16} \\
a_{25} & a_{26} \\
a_{35} & a_{36} \\
a_{45} & a_{46} \\
a_{55} & a_{56} \\
a_{65} & a_{66} \\
a_{75} & a_{76} \\
a_{85} & a_{86}
\end{bmatrix}
\begin{bmatrix}
x_3^{(0)} \\
x_4^{(0)}
\end{bmatrix}
\]

Process 1 Computes:

\[
W^{(1)} = \begin{bmatrix}
w_1^{(1)} \\
w_2^{(1)} \\
w_3^{(1)} \\
w_4^{(1)} \\
w_5^{(1)} \\
w_6^{(1)} \\
w_7^{(1)} \\
w_8^{(1)}
\end{bmatrix} = W^{(1)} + A_2^{(1)} X_2^{(1)} = \begin{bmatrix}
w_1^{(1)} \\
w_2^{(1)} \\
w_3^{(1)} \\
w_4^{(1)} \\
w_5^{(1)} \\
w_6^{(1)} \\
w_7^{(1)} \\
w_8^{(1)}
\end{bmatrix} + \begin{bmatrix}
a_{17} & a_{18} \\
a_{27} & a_{28} \\
a_{37} & a_{38} \\
a_{47} & a_{48} \\
a_{57} & a_{58} \\
a_{67} & a_{68} \\
a_{77} & a_{78} \\
a_{87} & a_{88}
\end{bmatrix}
\begin{bmatrix}
x_3^{(1)} \\
x_4^{(1)}
\end{bmatrix}
\]

Step 4: Communication between all processes in the process grid, to generate the (complete) \( W \) vectors on each process.

Process 0 and 1 communicate to sum \( W^{(0)} \) and \( W^{(1)} \) vectors, generating the complete
W vector, and overwriting $W^{(0)}$ and $W^{(1)}$ by $W$.

\[
W^{(0)} = W^{(1)} = \begin{bmatrix}
  w_1 \\
  w_2 \\
  w_3 \\
  w_4 \\
  w_5 \\
  w_6 \\
  w_7 \\
  w_8 
\end{bmatrix} = W^{(0)} + W^{(1)} = \begin{bmatrix}
  \alpha w_1^{(0)} + \beta w_1^{(1)} \\
  \alpha w_2^{(0)} + \beta w_2^{(1)} \\
  \alpha w_3^{(0)} + \beta w_3^{(1)} \\
  \alpha w_4^{(0)} + \beta w_4^{(1)} \\
  \alpha w_5^{(0)} + \beta w_5^{(1)} \\
  \alpha w_6^{(0)} + \beta w_6^{(1)} \\
  \alpha w_7^{(0)} + \beta w_7^{(1)} \\
  \alpha w_8^{(0)} + \beta w_8^{(1)} 
\end{bmatrix} + \begin{bmatrix}
  \beta w_1^{(0)} + \alpha w_1^{(1)} \\
  \beta w_2^{(0)} + \alpha w_2^{(1)} \\
  \beta w_3^{(0)} + \alpha w_3^{(1)} \\
  \beta w_4^{(0)} + \alpha w_4^{(1)} \\
  \beta w_5^{(0)} + \alpha w_5^{(1)} \\
  \beta w_6^{(0)} + \alpha w_6^{(1)} \\
  \beta w_7^{(0)} + \alpha w_7^{(1)} \\
  \beta w_8^{(0)} + \alpha w_8^{(1)} 
\end{bmatrix}
\]

### 3.4.1 Pseudo-code for a general matrix

Here we describe the generalization of the above pseudo code to an arbitrary $M \times N$ matrix, and write down the F77 statements from the actual code which implement each step of the pseudo code.

Let $\text{sub}(A)$ be the part of $A$ on process $p$. Let the distributed matrix $\text{sub}(A)$ on each process $p$ be of size $M \times N$. Let $k$ be the number of columns of $\text{sub}(A)$ for which the $M \times k$ block fits into $L_2$ cache. Let $b = N/k$.

**Step 1:** Load $Z^{(p)}$ into $X^{(p)}$.

For $i = 1, b$, do in parallel ($b = N/k$)

**Step 2:** Generate complete $X^{(p)}_i$ vectors, by a call to DGEMV on each process.

\[
(\text{sub}(X))_i = \alpha \ast (\text{sub}(A))_i \ast Y + \beta \ast (\text{sub}(Z))_i
\]

\[
\begin{aligned}
\text{CALL DGEMV}(\text{’T’}, M, K, \text{ALPHA}, A((I-1)\ast K \ast M + 1), \text{DESCA(LLD_)}, \\
+ Y, \text{INCY}, \text{BETA}, X((I-1)\ast K + 1), \text{INCX})
\end{aligned}
\]

**Step 3:** Generate the partial $W^{(p)}$ vectors by a second call to DGEMV on each process.

\[
\text{sub}(W) = \alpha \ast (\text{sub}(A))_i \ast (\text{sub}(X))_i + \beta \ast \text{sub}(W)
\]
CALL DGEMV(‘N’, M, K, ALPHA, A((I-1)*K*M + 1), DESCA(LLD_),
+ X((I-1)*K + 1), INCX, BETA, W, INCW)

End For.

**Step 4:** Communication: Sum the partial $W^{(p)}$ vectors among all processes and overwrite $W^{(p)}$ with this sum to generate the (Complete) $W$ vector. The BLACS routine DGSUM2D is used for this purpose. Thus:

$$W = sub(W) = \sum_{all\ processes} sub(W^{(p)})$$

CALL DGSUM2D(ICTXT, ‘ROW’, ‘ ’, M, 1, W, DESCW(LLD_), -1, -1)

### 3.4.2 Source code of the subroutine PDGEMVT

```fortran
SUBROUTINE PDGEMVT ( M, N, ALPHA, A, IA, JA, DESCA, X, IX, JX,
+ DESCX, INCX, Y, IY, JY, DESCY, INCY, BETA,
+ W, IW, JW, DESCW, INCW, Z, IZ, JZ, DESCZ,
+ INCZ)
```

* -- Scalar Arguments --

```
INTEGER M, N, IA, JA, IX, JX, IY, JY, IW, JW, IZ, JZ
INTEGER INCX, INCY, INCW, INCZ
DOUBLE PRECISION ALPHA, BETA
```

* -- Array Arguments --

```
INTEGER DESCA(*), DESCX(*), DESCY(*)
INTEGER DESCW(*), DESCZ(*)
DOUBLE PRECISION A(*), X(*), Y(*), W(*), Z(*)
```

* -- Local Variables --

```
INTEGER INFO, ICTXT, IAM
```
63
INTEGER I, J, K, B
INTEGER START_A, START_Y, START_X, START_W
CHARACTER*1 ROW, COL, TOP
PARAMETER (ROW = 'R', COL = 'C', TOP = ' ')
INTEGER BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DT_,
+ LLD_, MB_, M_, NB_, N_, RSRC_
PARAMETER ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DT_ = 1,
+ CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5, NB_ = 6,
+ RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )

ICTXT = DESCA(CTXT_)
* -- Initializing the starting pointers.
* Generally all values will be 1 --
START_A = (JA - 1) * DESCA(M_) + (IA - 1) + 1
START_Y = (JY - 1) * DESCY(M_) + (IY - 1) + 1
START_X = (JX - 1) * DESCX(M_) + (IX - 1) + 1
START_W = (JW - 1) * DESCW(M_) + (IW - 1) + 1

* -- Copy the Z vector into the X Vector --
CALL DCOPY(N, Z, INCZ, X, INCX)

* -- Determine the block size for the local matrix ----- 
CALL BLOCKSIZE(M, K)
B = N/K

DO I = 1, B

* -- Perform 1st GEMV call, implementing Step 2 -- 
CALL DGEMV('T', M, K, ALPHA, A(START_A + ((I-1)*K*M)),
+ DESCA(LLD_), Y(START_Y), INCY, BETA,
+ X(START_X + (I-1)*K), INCX)
The complete source code of the subroutine PDGEMVT is shown above. The last portion of the code is the clean up code. It is possible that the number of columns NQ of the distributed matrix on each processes cannot be exactly divided by the block size k. This clean up code is therefore necessary to take care of the extra columns which are left over after partitioning the distributed matrix into blocks of k columns each.
3.5 Driver Program

In this section, the parallel cache efficient GEMVT algorithm is described from the PDGEMVT driver program perspective on the $8 \times 8$ matrix whose data elements are column cyclicly distributed.

3.5.1 Auxiliary Routines

The ScaLAPACK auxiliary routines NUMROC and DESCINIT are used in this algorithm in the same way as the previous algorithm. The details of these routines are given in Section 2.5.1.

3.5.2 Driver Program

The driver program for the parallel cache efficient PDGEMVT subroutine for a $8 \times 8$ global matrix $A$ is described in this section.

1. Call subroutine $\text{BLACS\_PINFO}(IAM, NPROCS)^1$ to get values for $NPROCS$ and $IAM$ where $NPROCS$ is the total number of processes and $IAM$ is the rank of each process in the process grid.

2. Call subroutine $\text{PDPBLASINFO}(OUTFILE, ..., IAM, NPROCS)^1$ to read the input file. Details about the various parameters are explained in Section 2.5.2.

The following values are read from the input file for the $8 \times 8$ example matrix mapped onto a $1 \times 2$ process grid:

- $M = 8$  \hspace{1cm}  $NPROW = 1$
- $N = 8$  \hspace{1cm}  $NPCOL = 2$
- $V = 1$
- $MB = 8$
- $NB = 2$

3. Call BLACS subroutines $\text{BLACS\_GET}^1$, $\text{BLACS\_GRIDINIT}^1$ and $\text{BLACS\_GRIDINFO}^1$ to get the global grid context, to initialize the Process Grid and to retrieve the process grid information respectively.

---

1Refer Appendix B for BLACS description.
4. Call ScALAPACK tool function NUMROC to get the exact number of rows and columns of that part of the $A$ matrix distributed on each process. The NUMROC routine call is same as described for block cyclic data distribution in Section 2.5.2.

The input values to subroutine NUMROC and the output values $MP$, $NQ$, $YP$, $ZP$ and $ZQ$ from the subroutine on each process are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Process#</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InputValues</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$N$</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$V$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$MB$</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$NB$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$MYROW$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$MYCOL$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$NPROW$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$NPCOL$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>OutputValues</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MP$</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$NQ$</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$YP$</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>$YQ$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$ZP$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$ZQ$</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.1: Input and output values to and from the subroutine NUMROC on each process for a $8 \times 8$ matrix

Therefore the number of rows and columns of the distributed matrix $A$ on each process is $MP \times NQ = M \times NQ$.

- On process 0: $8 \times 4$
- On process 1: $8 \times 4$

The size of input vector $Y$ is also determined on each process by making use of $VP$ (size of vector on each process) and $VQ$ (number of columns in $Y$ on each process (either 0 or 1).
On process 0: \(8 \times 1\)

On process 1: \textit{Nothing stored.}

The size of input vector \(Z\) is also determined on each process by making use of \(ZP\) (size of the vector on each process) and \(ZQ\) (number of rows in \(Z\) on each process (either 0 or 1)).

On process 0: \(1 \times 4\)

On process 1: \(1 \times 4\)

5. Call ScaLAPACK tool routine DESCINIT to initialize the array descriptors (\(\text{DESC}_i\)) for the matrix \(A\) and input vectors \(Y\) and \(Z\), and output vectors \(X\) and \(W\).

Let \(\text{DESCA}, \text{DESCY}, \text{DESCZ}, \text{DESCX}\) and \(\text{DESCW}\) be the array descriptor for matrix \(A\) and input vectors \(Y\) and \(Z\), and output vectors \(X\) and \(W\). The DESCINIT routine call is the same as described for block cyclic data distribution in Section 2.5.2. Then \(\text{DESCA}(I), \text{DESCY}(I), \text{DESCZ}(I), \text{DESCX}(I)\) and \(\text{DESCW}(I)\) for \(I = 1, 9\) are arrays of integers which are initialized to the values shown in Table 3.2

<table>
<thead>
<tr>
<th>(\text{DESC}_i(I))</th>
<th>Parameter</th>
<th>(\text{DESCA})</th>
<th>(\text{DESCY})</th>
<th>(\text{DESCZ})</th>
<th>(\text{DESCX})</th>
<th>(\text{DESCW})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{DESC}_1(1))</td>
<td>(\text{DTYPE})</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(\text{DESC}_2(2))</td>
<td>ICTXT</td>
<td>ICTXT</td>
<td>ICTXT</td>
<td>ICTXT</td>
<td>ICTXT</td>
<td>ICTXT</td>
</tr>
<tr>
<td>(\text{DESC}_3(3))</td>
<td>(M)</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>(\text{DESC}_4(4))</td>
<td>(N)</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>(\text{DESC}_5(5))</td>
<td>(MB)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(\text{DESC}_6(6))</td>
<td>(NB)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(\text{DESC}_7(7))</td>
<td>RSRC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(\text{DESC}_8(8))</td>
<td>CSRC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(\text{DESC}_9(9))</td>
<td>LLD</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.2: DESC Parameters for the \(8 \times 8\) matrix \(A\), \(8 \times 1\) vectors \(Y\) and \(W\), and \(1 \times 8\) vectors \(Z\) and \(X\)

6. The driver program is executed by each process simultaneously and thus stores it’s part of the distributed matrix \(A\) and the distributed vectors \(Y\) and \(Z\). A single vector \(\text{MEM}\) is used on each process to store the input matrix \(A\), input
vectors Y and Z and output vectors X and W. We introduce the pointers (on each process) \textit{IPA}, \textit{IPY}, \textit{IPZ}, \textit{IPX} and \textit{IPW} to point to the location in \textit{MEM} where the parts of the matrix \textit{A}, vector \textit{Y}, vector \textit{Z}, vector \textit{X} and vector \textit{W} are stored on each process, respectively.

Therefore, \textit{MEM} is a vector to store part of matrix \textit{A}, vector \textit{Y}, \textit{Z}, \textit{X}, and \textit{W} on each process and \textit{IPA}, \textit{IPY}, \textit{IPZ}, \textit{IPX} and \textit{IPW} are the pointers having starting addresses of the parts of matrix \textit{A}, vector \textit{Y}, vector \textit{Z}, vector \textit{X}, and vector \textit{W} on each process, respectively. We also define \textit{IPL} to be the ending address of \textit{MEM} array.

The formulas for these pointers, on each process, are thus as follows:

\begin{align*}
\textit{IPA} &= 1 \\
\textit{IPY} &= \textit{IPA} + \text{DESCA}(\textit{LLD}_A) \times \textit{NQ} \\
\textit{IPZ} &= \textit{IPY} + \text{DESCY}(\textit{LLD}_A) \\
\textit{IPX} &= \textit{IPZ} + \text{DESCZ}(\textit{LLD}_A) \times \textit{NQ} \\
\textit{IPW} &= \textit{IPX} + \text{DESCX}(\textit{LLD}_A) \times \textit{NQ} \\
\textit{IPL} &= \textit{IPW} + \text{DESCW}(\textit{LLD}_A)
\end{align*}

\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
Process\# & 0 & 1 \\
\hline
\textit{IPA} & 1 & 1 \\
\textit{IPY} & 33 & 33 \\
\textit{IPZ} & 41 & 41 \\
\textit{IPX} & 45 & 45 \\
\textit{IPW} & 49 & 49 \\
\textit{IPL} & 57 & 57 \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
\textit{LLD}_A & 8 & 8 \\
\textit{LLD}_Y & 8 & 8 \\
\textit{LLD}_Z & 4 & 4 \\
\textit{LLD}_X & 4 & 4 \\
\textit{LLD}_W & 8 & 8 \\
\hline
\end{tabular}
\end{table}

Table 3.3: Values of \textit{IPA}, \textit{IPY}, \textit{IPZ}, \textit{IPX}, \textit{IPW} and \textit{IPL} and values of \textit{LLD}_A, \textit{LLD}_Y, \textit{LLD}_Z, \textit{LLD}_X, \textit{LLD}_W and \textit{LLD}_L on each process
The values of $IPA$, $IPY$, $IPZ$, $IPX$, $IPW$ and $IPL$ on each process for a $8 \times 8$ matrix and $8 \times 1$ input vector and the corresponding values of $LLD_A = DESC_A(9)$, $LLD_Y = DESC_Y(9)$, $LLD_Z = DESC_Z(9)$, $LLD_X = DESC_X(9)$, $LLD_W = DESC_W(9)$ are listed for each process in the Table 3.3

7. Generate the distributed $A$ matrix and the distributed input vectors $Y$ and $Z$ on each process using the ScaLAPCK routine $PDMATGEN$. The distribution of matrix $A$, vectors $Y$ and $Z$ on each process is given in Section 3.3.

$IASEED = 100$

CALL PDMATGEN( ICTXT, 'No transpose', 'No transpose',
+ DESC_A( M_ ), DESC_A( N_ ), DESC_A( MB_ ),
+ DESC_A( NB_ ), MEM( IPA ), DESC_A( LLD_ ),
+ DESC_A( RSRC_ ), DESC_A( CSRC_ ), IASEED,
+ 0, MP, 0, NQ, MYROW, MYCOL, NPROW, NPCOL )

$IBSEED = 200$

CALL PDMATGEN( ICTXT, 'No transpose', 'No transpose',
+ DESC_Y( M_ ), DESC_Y( N_ ), DESC_Y( MB_ ),
+ DESC_Y( NB_ ), MEM( IPY ), DESC_Y( LLD_ ),
+ DESC_Y( RSRC_ ), DESC_Y( CSRC_ ), IBSEED,
+ 0, MP, 0, VQ, MYROW, MYCOL, NPROW, NPCOL )

$ICSEED = 300$

CALL PDMATGEN( ICTXT, 'No transpose', 'No transpose',
+ DESC_X( M_ ), DESC_X( N_ ), DESC_X( MB_ ),
+ DESC_X( NB_ ), MEM( IPX ), DESC_X( LLD_ ),
+ DESC_X( RSRC_ ), DESC_X( CSRC_ ), ICSEED,
+ 0, MP, 0, VQ, MYROW, MYCOL, NPROW, NPCOL )

8. Send the $Y$ vector from process 0 to all processes in the process grid. In our example, $Y^{(0)}$ is sent from process 0 to process 1 and stored in $Y^{(1)}$

IF (MYCOL.EQ.0) THEN
  * -- Sending the Y Vector --
  CALL DGEBS2D(ICTXT, 'R', ',', MP, 1, MEM(IPY), DESC_Y(LLD_))
ELSE
*    -- Receiving the Y Vector --
    CALL DGEBR2D(ICTXT, 'R', ' ', MP, 1, MEM(IPY), DESCY(LLD_),
             +       MYROW, 0)
ENDIF

9. We have distributed the matrix \( A \) and distributed the input vectors \( Y \) and \( Z \) on each process. Now, call the cache efficient PDGEMVT routine on each process to generate the output vector \( W \); internally PDGEMVT makes two calls to \( DGEMV \) using column blocking on the distributed matrix on each process. The object is to achieve cache-efficiency by selecting the block size \( k \) so that the \( MP \times k \) blocks effectively fill the available \( L_2 \) cache on each process.

\[
\text{CALL PDGEMVT}(MP, NQ, 1.d0, MEM(IPA), 1, 1, DESCA, MEM(IPX), 1,
             +       1, DESCX, 1, MEM(IPY), 1, 1, DESCY, 1, 1.d0,
             +       MEM(IPW), 1, 1, DESCW, 1, MEM(IPZ), 1, 1,
             +       DESCZ, 1)
\]

10. Call subroutine \( BLACS\_GRIDEXIT(ICTXT) \) \(^1\) to release the process grid and \( BLACS\_EXIT(0) \) \(^1\) to indicate that all work using the BLACS has been completed; this call initiates the destruction of the BLACS internal structures and frees all memory.

### 3.6 Description of subroutines on each process

The action of all subroutines in our cache efficient PDGEMVT using the column cyclic data distribution is described in this section using the \( 8 \times 8 \) example matrix.

Example: Let \( A \) be the following \( 8 \times 8 \) matrix, partitioned into \( 8 \times 2 \) column blocks, and mapped onto a \( 1 \times 2 \) process grid.

\(^1\)Refer Appendix B for BLACS description.
\[
A = \begin{pmatrix}
11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\
21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\
31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\
41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\
51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \\
61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 \\
71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 \\
81 & 82 & 83 & 84 & 85 & 86 & 87 & 88
\end{pmatrix}
\]

\[
P_c \Rightarrow \begin{pmatrix} 0 & 1 & 0 & 1 \end{pmatrix}
\]

\[
P_r \downarrow \begin{pmatrix}
11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\
21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\
31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\
41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\
51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \\
61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 \\
71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 \\
81 & 82 & 83 & 84 & 85 & 86 & 87 & 88
\end{pmatrix}
\]

Figure 3.8: An 8 × 8 matrix partitioned into 8 × 2 blocks in column cyclic fashion.

The distribution of matrix $A$ on the 1 × 2 process grid is shown in Figure 3.8. The data layout of the 8 × 8 matrix on each of 2 processes is as follows:

**Process 0:** (0,0) in process grid

\[
A^{(0)} = \begin{pmatrix}
11 & 12 & 15 & 16 \\
21 & 22 & 25 & 26 \\
31 & 32 & 35 & 36 \\
41 & 42 & 45 & 46 \\
51 & 52 & 55 & 56 \\
61 & 62 & 65 & 66 \\
71 & 72 & 75 & 76 \\
81 & 82 & 85 & 86
\end{pmatrix} 8 \times 4
\]

$A^{(0)}$ is the 8 × 4 part of the A matrix which is distributed to process 0 = (0,0).
**Process 1:** (0,1) in process grid

\[
A^{(1)} = \begin{bmatrix}
13 & 14 & 17 & 18 \\
23 & 24 & 27 & 28 \\
33 & 34 & 37 & 38 \\
43 & 44 & 47 & 48 \\
53 & 54 & 57 & 58 \\
63 & 64 & 67 & 68 \\
73 & 74 & 77 & 78 \\
83 & 84 & 87 & 88
\end{bmatrix} \quad 8 \times 4
\]

\(A^{(1)}\) is the \(8 \times 4\) part of the A matrix which is distributed to process 1 = (0,1).

Let \(Y\) be an input vector of size 8, partitioned into \(8 \times 2\) blocks, and mapped onto a \(1 \times 2\) process grid. The Figure 3.9 shows the partitioned vector and the process coordinates in process grid for each block.

\[
\begin{align*}
P_c &\Rightarrow 0 & P_c &\Rightarrow 0 \\
\downarrow P_r & & \downarrow P_r \\
\begin{pmatrix}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8
\end{pmatrix} & & \begin{pmatrix}
\end{pmatrix}
\end{align*}
\]

Figure 3.9: A \(8 \times 1\) input \(Y\) vector distributed to process 0 = (0,0)

**NOTE:** Since there is only one column and the block size is 8, the complete vector is placed in process 0 = (0,0).

The data layout of vector \(Y\) on each of the 2 processes is as follows:
Process 0: (0,0) in process grid

\[ Y^{(0)} = \begin{bmatrix}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8
\end{bmatrix} \]

\( Y^{(0)} \) is a vector of size 8 which is distributed to process \( 0 = (0,0) \).

Process 1: (0,1) in process grid

Nothing is distributed to process 1

Similarly, let \( Z \) be an input row vector of size 8, partitioned using \( 8 \times 2 \) blocks, and mapped onto a \( 1 \times 2 \) process grid. The Figure 3.10 shows the partitioned vector and the process coordinates for each block.

\[ P_r \downarrow P_c \Rightarrow 0 \quad 1 \quad 0 \quad 1 \\
0 \begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8
\end{array}
\]

\[ P_r \downarrow P_c \Rightarrow 0 \quad 1 \quad 0 \quad 1 \\
0 \begin{array}{cccc}
(0,0) & (0,1) & (0,0) & (0,1)
\end{array}
\]

Figure 3.10: An example input \( Z \) vector partitioned using \( 8 \times 2 \) blocks and the corresponding process coordinates for each block

Since there is only one row, the \( Z \) vector is placed 2 components at a time on processes \((0,0), (0,1), (0,0) \) and \((0,1)\) alternately, that is, in a column-cyclic manner. The data layout of vector \( Z \) on each of the 2 processes is as follows:
Process 0: (0,0) in process grid

\[ Z^{(0)} = \begin{bmatrix} 1 & 2 & 5 & 6 \end{bmatrix}_{1 \times 4} \]

\( Z^{(0)} \) is a vector of size 4 which is distributed to process 0 = (0,0).

Process 1: (0,1) in process grid

\[ Z^{(1)} = \begin{bmatrix} 3 & 4 & 7 & 8 \end{bmatrix}_{1 \times 4} \]

\( Z^{(1)} \) is a vector of size 4 which is distributed to process 1 = (0,1).

We now describe how our routine PDGEMVT can be expected to achieve cache-efficiency on each process of the process grid, using the above column-cyclic data layout for \( A, Y, Z \).

Let \( k = 2 \) be the number of columns in each column block of that part of the distributed matrix which resides on each process. Then the number of column blocks, \( b \), on each process is computed as

\[ b = NQ/k = 4/2 = 2 \]

In the general situation of a large dense matrix \( M \times N \) matrix, the idea is to select \( k \) so that the \( M \times k \) matrix elements in each column block fill the available \( L_2 \) cache.

Step 0: (Preliminary steps done in the driver for PDGEMVT)

(a) The output \( W^{(p)} \) vector is initialized to zero in all processes

\[ W^{(p)} = 0 \]

(b) Send the \( Y \) vector which resides on the process 0 to process 1:

Process 0 sends \( Y^{(0)} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} \) to Process 1 to make \( Y^{(1)} \)
**Step 1:** Copy the $Z^{(p)}$ vector to $X^{(p)}$

Process 0 Computes:

$$X^{(0)} = (Z^{(0)})^T = \begin{bmatrix} 1 \\ 2 \\ 5 \\ 6 \end{bmatrix}$$

Process 1 Computes:

$$X^{(1)} = (Z^{(1)})^T = \begin{bmatrix} 3 \\ 4 \\ 7 \\ 8 \end{bmatrix}$$

Next perform the *cache-efficient* loop over $i=1,b$.

**For** $i = 1$:

**Step 2** ($i=1$): Generation of complete $X_1$ vectors on each process.

Process 0 Computes:

$$X_1^{(0)} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \end{bmatrix} = (A_1^{(0)})^T \ Y^{(0)} + X_1^{(0)}$$

$$= \begin{bmatrix} 11 & 21 & 31 & 41 & 51 & 61 & 71 & 81 \\ 12 & 22 & 32 & 42 & 52 & 62 & 72 & 82 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2077 \\ 2114 \end{bmatrix}$$

Process 1 Computes:

$$X_1^{(1)} = \begin{bmatrix} x_3^{(1)} \\ x_4^{(1)} \end{bmatrix} = (A_1^{(1)})^T \ Y^{(1)} + X_1^{(1)}$$
\[
\begin{bmatrix}
13 & 23 & 33 & 43 & 53 & 63 & 73 & 83 \\
14 & 24 & 34 & 44 & 54 & 64 & 74 & 84
\end{bmatrix}
+ \begin{bmatrix}
3 \\
4
\end{bmatrix} = \begin{bmatrix}
2151 \\
2188
\end{bmatrix}
\]

**Step 3 (i=1):** Generation of partial \( W \) vectors on each process.

Process 0 Computes:

\[
W^{(0)} = \begin{bmatrix}
w_1^{(0)} \\
w_2^{(0)} \\
w_3^{(0)} \\
w_4^{(0)} \\
w_5^{(0)} \\
w_6^{(0)} \\
w_7^{(0)} \\
w_8^{(0)}
\end{bmatrix} = A_1^{(0)} X_1^{(0)} = \begin{bmatrix}
11 & 12 \\
21 & 22 \\
31 & 32 \\
41 & 42 \\
51 & 52 \\
61 & 62 \\
71 & 72 \\
81 & 82
\end{bmatrix} = \begin{bmatrix}
48215 \\
90125 \\
132035 \\
173945 \\
215855 \\
257765 \\
299675 \\
341585
\end{bmatrix}
\]

Process 1 Computes:

\[
W^{(1)} = \begin{bmatrix}
w_1^{(1)} \\
w_2^{(1)} \\
w_3^{(1)} \\
w_4^{(1)} \\
w_5^{(1)} \\
w_6^{(1)} \\
w_7^{(1)} \\
w_8^{(1)}
\end{bmatrix} = A_1^{(1)} X_1^{(1)} = \begin{bmatrix}
13 & 14 \\
23 & 24 \\
33 & 34 \\
43 & 44 \\
53 & 54 \\
63 & 64 \\
73 & 74 \\
83 & 84
\end{bmatrix} = \begin{bmatrix}
58595 \\
101985 \\
145375 \\
188765 \\
232155 \\
275545 \\
318935 \\
362325
\end{bmatrix}
\]

For \( i = 2 \):

Flush the \( L_2 \) cache of \( A_1^{(p)} \), \( p=0,1 \), and reload with \( A_2^{(p)} \), \( p=0,1 \) for the next part of the two matrix-vector multiplications, \( X = A^T \ast Y + Z \) and \( W = A \ast X \).
Step 2 (i=2): Generation of complete $X_2$ vectors on each process.

Process 0 Computes:

\[
X_2^{(0)} = \begin{bmatrix} x_5^{(0)} \\ x_6^{(0)} \end{bmatrix} = (A_2^{(0)})^T Y^{(0)} + X_2^{(0)}
\]

\[
= \begin{bmatrix} 15 & 25 & 35 & 45 & 55 & 65 & 75 & 85 \\ 16 & 26 & 36 & 46 & 56 & 66 & 76 & 86 \end{bmatrix} + \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 2225 \\ 2262 \end{bmatrix}
\]

Process 1 Computes:

\[
X_2^{(1)} = \begin{bmatrix} x_7^{(1)} \\ x_8^{(1)} \end{bmatrix} = (A_2^{(1)})^T Y^{(1)} + X_2^{(1)}
\]

\[
= \begin{bmatrix} 17 & 27 & 37 & 47 & 57 & 67 & 77 & 87 \\ 18 & 28 & 38 & 48 & 58 & 68 & 78 & 88 \end{bmatrix} + \begin{bmatrix} 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 2299 \\ 2336 \end{bmatrix}
\]

Step 4 (i=2): Generation of partial $W$ vectors on each process.

Process 0 Computes:

\[
W^{(0)} = \begin{bmatrix} w_1^{(0)} \\ w_2^{(0)} \\ w_3^{(0)} \\ w_4^{(0)} \\ w_5^{(0)} \\ w_6^{(0)} \\ w_7^{(0)} \\ w_8^{(0)} \end{bmatrix} = W^{(0)} + A_2^{(0)} X_2^{(0)}
\]
Process 1 Computes:

\[
\begin{pmatrix}
48215 \\
90125 \\
132035 \\
173945 \\
215855 \\
257765 \\
299675 \\
341585 \\
\end{pmatrix} + \begin{pmatrix}
15 & 16 \\
25 & 26 \\
35 & 36 \\
45 & 46 \\
55 & 56 \\
65 & 66 \\
75 & 76 \\
85 & 86 \\
\end{pmatrix} \begin{pmatrix}
2225 \\
\end{pmatrix} = \begin{pmatrix}
117782 \\
204562 \\
291342 \\
378122 \\
464902 \\
551682 \\
638462 \\
725242 \\
\end{pmatrix}
\]

\[
W^{(1)} = \begin{pmatrix}
w^{(1)}_1 \\
w^{(1)}_2 \\
w^{(1)}_3 \\
w^{(1)}_4 \\
w^{(1)}_5 \\
w^{(1)}_6 \\
w^{(1)}_7 \\
w^{(1)}_8 \\
\end{pmatrix} = W^{(1)} + A^{(1)}_2 X^{(1)}_2
\]

\[
\begin{pmatrix}
58595 \\
101985 \\
145375 \\
188765 \\
232155 \\
275545 \\
318935 \\
362325 \\
\end{pmatrix} + \begin{pmatrix}
17 & 18 \\
27 & 28 \\
37 & 38 \\
47 & 48 \\
57 & 58 \\
67 & 68 \\
77 & 78 \\
87 & 88 \\
\end{pmatrix} \begin{pmatrix}
2299 \\
\end{pmatrix} = \begin{pmatrix}
139726 \\
229466 \\
319206 \\
408946 \\
498686 \\
588426 \\
678166 \\
767906 \\
\end{pmatrix}
\]

**Step 5:** Communication between all processes in the process grid, to generate the complete \( W \) vectors on each process.

Process 0 and 1 sums its \( W \) vector and overwrites \( W \) on each of these processes to generate the complete \( W \) vector.
\[
W = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \end{bmatrix} = W^{(0)} + W^{(1)} = \begin{bmatrix} w_1^{(0)} \\ w_2^{(0)} \\ w_3^{(0)} \\ w_4^{(0)} \\ w_5^{(0)} \\ w_6^{(0)} \\ w_7^{(0)} \\ w_8^{(0)} \end{bmatrix} + \begin{bmatrix} w_1^{(1)} \\ w_2^{(1)} \\ w_3^{(1)} \\ w_4^{(1)} \\ w_5^{(1)} \\ w_6^{(1)} \\ w_7^{(1)} \\ w_8^{(1)} \end{bmatrix}
\]

\[
= \begin{bmatrix} 117782 \\ 204562 \\ 291342 \\ 378122 \\ 464902 \\ 551682 \\ 638462 \\ 725242 \end{bmatrix} + \begin{bmatrix} 139726 \\ 229466 \\ 319206 \\ 408946 \\ 498686 \\ 588426 \\ 678166 \\ 767906 \end{bmatrix} = \begin{bmatrix} 257508 \\ 434028 \\ 610548 \\ 787068 \\ 963588 \\ 1140108 \\ 1316628 \\ 1493148 \end{bmatrix}
\]
Chapter 4

Timing and Performance Analysis for Cache-efficient Parallel GEMVT Codes

4.1 Introduction

The timing and performance analysis of our cache-efficient parallel GEMVT codes were done on the Lemieux\(^1\) parallel cluster. The Lemieux cluster at Pittsburgh Super Computing center consists of 750 Compaq Alphaserver ES45 nodes and two separate front end nodes. Each computational node contains four 1-GHz Alpha EV6.8CB processors and runs the Tru64 Unix operating system. A Quadrics interconnection network connects the nodes. The Alphaserver ES45 has a second level cache. This secondary cache, or what we have hereafter called \(L_2\) cache is the \textit{Bcache} \citep{15}, which has 8 MB of memory.

Each node is a 4 Shared Memory Processor (SMP), with 4 GB of main memory and 8 MB of \textit{Bcache}. In order to avoid any data distribution of the part of \(A\) allocated to one node to the main memory of 4 different processors, we choose to use only one processor per node. This ensures that the part of \(A\) allocated to each node by the block cyclic data distribution will reside on the RAM of only one processor, and that the reading of \(A\) from RAM to \textit{Bcache} on this processor will achieve cache efficiency for the block sizes we elect to use for our experiments.

\(^1\)http://www.psc.edu/machines/tcs/lemieux.html
4.2 Performance Analysis for the Block-Cyclic Cache-efficient Parallel GEMVT code and Comparisons with a non-cache efficient GEMVT code

Following a suggestion of Cleve Moler[14] we measure the parallel efficiency of our parallel cache-efficient GEMVT code by looking at the scaled speedups as the total number of computations and data is increased in a manner proportional to the number of processors. The idea is to keep the amount of work done on each processor constant as the number of processors is increased so as to measure the degradation of times and MFlop rates due to communication overhead as the number of processors is increased. However, we also wish to establish a comparison of our cache-efficient parallel GEMVT with the reference code which makes two straight calls to PDGEMV from the PBLAS library. This reference code is the PBLAS block cyclic implementation described in [7] and [6]; here no attempt was made to achieve cache-efficiency on each node. We find that both this reference code and the block-cyclic version of our new cache-efficient parallel GEMVT subroutine, PDGEMVT, are perfectly scalable. However, our new PDGEMVT code generally runs faster than the non-cache efficient reference code, due to faster execution of the two coupled matrix-vector multiplications which are accomplished by column blocking of $sub(A)$ on each processor, where $sub(A)$ is the part of $A$ allocated to each processor under the block cyclic data distribution.

In addition, we implemented a cache-efficient parallel GEMVT which was based on a column-cyclic data distribution, using column blocking of $sub(A)$ on each processor. This code, however, did not perform as well as the cache-efficient parallel GEMVT based on the block-cyclic data distribution. On the other hand, the column-cyclic cache-efficient code was competitive with the reference code up to 25 processors, after which the reference code performed better.

For square matrices having $7200 \times 7200$ sub-matrices on each processor, i.e, 51.84 million matrix elements per CPU, the block cyclic cache-efficient GEMVT was run using a square process grid, yielding timing numbers and MFlop rates for 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 processors. To determine the influence of the column blocking of $sub(A)$ on each processor, this was done with column block sizes of $k = 12, 84,$
### Table 4.1: Average Timings of Cache-Efficient Block Cyclic PDGEMVT with different block sizes k Compared with Average Timings of Non-Cache Efficient PDGEMVT (Average of 3 runs)

<table>
<thead>
<tr>
<th>#Processor</th>
<th>Comp Time</th>
<th>Wall Time</th>
<th>Comm Time</th>
<th>Wall Time</th>
<th>Comm Time</th>
<th>Wall Time</th>
<th>Wall Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>k = 164</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.4610326</td>
<td>0.4621445</td>
<td>0.0011119</td>
<td>0.4421574</td>
<td>0.435661</td>
<td>0.0014087</td>
<td>0.504412</td>
</tr>
<tr>
<td>4</td>
<td>0.4610326</td>
<td>0.4718945</td>
<td>0.0108619</td>
<td>0.4421574</td>
<td>0.4556027</td>
<td>0.0134453</td>
<td>0.516439</td>
</tr>
<tr>
<td>9</td>
<td>0.4610326</td>
<td>0.4694585</td>
<td>0.0084259</td>
<td>0.4421574</td>
<td>0.4507019</td>
<td>0.0085445</td>
<td>0.517723</td>
</tr>
<tr>
<td>16</td>
<td>0.4610326</td>
<td>0.4781023</td>
<td>0.0170697</td>
<td>0.4421574</td>
<td>0.4624316</td>
<td>0.0202742</td>
<td>0.519215</td>
</tr>
<tr>
<td>25</td>
<td>0.4610326</td>
<td>0.4806795</td>
<td>0.0196469</td>
<td>0.4421574</td>
<td>0.4657354</td>
<td>0.023578</td>
<td>0.518329</td>
</tr>
<tr>
<td>36</td>
<td>0.4610326</td>
<td>0.4848142</td>
<td>0.0237816</td>
<td>0.4421574</td>
<td>0.4723376</td>
<td>0.0301802</td>
<td>0.518578</td>
</tr>
<tr>
<td>49</td>
<td>0.4610326</td>
<td>0.48822</td>
<td>0.0271874</td>
<td>0.4421574</td>
<td>0.4768889</td>
<td>0.0347315</td>
<td>0.520744</td>
</tr>
<tr>
<td>64</td>
<td>0.4610326</td>
<td>0.4894023</td>
<td>0.0283697</td>
<td>0.4421574</td>
<td>0.4775189</td>
<td>0.0353615</td>
<td>0.520180</td>
</tr>
<tr>
<td>81</td>
<td>0.4610326</td>
<td>0.4870553</td>
<td>0.0260227</td>
<td>0.4421574</td>
<td>0.4754393</td>
<td>0.0332819</td>
<td>0.524398</td>
</tr>
<tr>
<td>100</td>
<td>0.4610326</td>
<td>0.484253</td>
<td>0.0263927</td>
<td>0.4421574</td>
<td>0.4752659</td>
<td>0.0331085</td>
<td>0.521564</td>
</tr>
<tr>
<td>k = 132</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.4426123</td>
<td>0.443942</td>
<td>0.0033297</td>
<td>0.4432783</td>
<td>0.4448776</td>
<td>0.0015993</td>
<td>0.504412</td>
</tr>
<tr>
<td>4</td>
<td>0.4426123</td>
<td>0.454321</td>
<td>0.0117087</td>
<td>0.4432783</td>
<td>0.4541273</td>
<td>0.010849</td>
<td>0.516439</td>
</tr>
<tr>
<td>9</td>
<td>0.4426123</td>
<td>0.451799</td>
<td>0.0091076</td>
<td>0.4432783</td>
<td>0.4539631</td>
<td>0.0106848</td>
<td>0.517723</td>
</tr>
<tr>
<td>16</td>
<td>0.4426123</td>
<td>0.4601394</td>
<td>0.0175271</td>
<td>0.4432783</td>
<td>0.4593977</td>
<td>0.0161194</td>
<td>0.519215</td>
</tr>
<tr>
<td>25</td>
<td>0.4426123</td>
<td>0.4627634</td>
<td>0.0201511</td>
<td>0.4432783</td>
<td>0.4617438</td>
<td>0.0184655</td>
<td>0.518329</td>
</tr>
<tr>
<td>36</td>
<td>0.4426123</td>
<td>0.4674936</td>
<td>0.0248813</td>
<td>0.4432783</td>
<td>0.4667102</td>
<td>0.0234319</td>
<td>0.518578</td>
</tr>
<tr>
<td>49</td>
<td>0.4426123</td>
<td>0.4718011</td>
<td>0.0291888</td>
<td>0.4432783</td>
<td>0.4691627</td>
<td>0.0258844</td>
<td>0.520744</td>
</tr>
<tr>
<td>64</td>
<td>0.4426123</td>
<td>0.4726553</td>
<td>0.030043</td>
<td>0.4432783</td>
<td>0.4684923</td>
<td>0.025214</td>
<td>0.520180</td>
</tr>
<tr>
<td>81</td>
<td>0.4426123</td>
<td>0.4697844</td>
<td>0.0271721</td>
<td>0.4432783</td>
<td>0.4673136</td>
<td>0.0240353</td>
<td>0.524398</td>
</tr>
<tr>
<td>100</td>
<td>0.4426123</td>
<td>0.4708126</td>
<td>0.0282003</td>
<td>0.4432783</td>
<td>0.4673177</td>
<td>0.0240934</td>
<td>0.521564</td>
</tr>
</tbody>
</table>

| k = 84     |           |           |           |           |           |           |           |
| 1          | 0.4426123 | 0.443942  | 0.0033297 | 0.4432783 | 0.4448776 | 0.0015993 | 0.504412  |
| 4          | 0.4426123 | 0.454321  | 0.0117087 | 0.4432783 | 0.4541273 | 0.010849  | 0.516439  |
| 9          | 0.4426123 | 0.451799  | 0.0091076 | 0.4432783 | 0.4539631 | 0.0106848 | 0.517723  |
| 16         | 0.4426123 | 0.4601394 | 0.0175271 | 0.4432783 | 0.4593977 | 0.0161194 | 0.519215  |
| 25         | 0.4426123 | 0.4627634 | 0.0201511 | 0.4432783 | 0.4617438 | 0.0184655 | 0.518329  |
| 36         | 0.4426123 | 0.4674936 | 0.0248813 | 0.4432783 | 0.4667102 | 0.0234319 | 0.518578  |
| 49         | 0.4426123 | 0.4718011 | 0.0291888 | 0.4432783 | 0.4691627 | 0.0258844 | 0.520744  |
| 64         | 0.4426123 | 0.4726553 | 0.030043  | 0.4432783 | 0.4684923 | 0.025214  | 0.520180  |
| 81         | 0.4426123 | 0.4697844 | 0.0271721 | 0.4432783 | 0.4673136 | 0.0240353 | 0.524398  |
| 100        | 0.4426123 | 0.4708126 | 0.0282003 | 0.4432783 | 0.4673177 | 0.0240934 | 0.521564  |
108, 132 and 164. The computation times, wall times and communication times for these runs are displayed in Table 4.1, along with the data for the non-cache efficient reference code. Here all the timing numbers are the average of 3 runs on the Lemieux cluster. The times for computation were taken as the actual time using MPI_WTime when running on one processor with all calls to DGSUM2D commented. Since this time would be the same on all processors as the number of processors is increased, this number was used for all values of \( NPROCS \). For each \( k \) value the average of 3 runs was taken to compute this computation time.

It will also be observed that the cache-efficiency is not affected very much by the choice of column blocking parameter \( k \) on \( \text{sub}(A) \) on each processor. In Table 4.5 the percentage of \( Bcache \) filled by the \( 7200 \times k \) column blocks of the \( 7200 \times 7200 \text{sub}(A) \) matrix is given for \( k = 12, 84, 108, 132 \) and 164.

<table>
<thead>
<tr>
<th>Block Size ( k )</th>
<th>Memory Size in MBytes</th>
<th>% of ( Bcache ) filled on each processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>164</td>
<td>9.00</td>
<td>112.61</td>
</tr>
<tr>
<td>132</td>
<td>7.25</td>
<td>90.63</td>
</tr>
<tr>
<td>108</td>
<td>5.93</td>
<td>74.16</td>
</tr>
<tr>
<td>84</td>
<td>4.61</td>
<td>57.68</td>
</tr>
<tr>
<td>12</td>
<td>0.66</td>
<td>8.24</td>
</tr>
</tbody>
</table>

Table 4.2: Percentage of \( Bcache \) filled for different block sizes, \( K \)

Generally, the smallest times occur for \( k=84 \) (57.68\% of \( Bcache \)) or \( k=108 \) (74.16\% of \( Bcache \)), while for \( k=164 \) (112.61\% of \( Bcache \)) and \( k=12 \) (8.24\% of \( Bcache \)) we notice longer times. For \( k=164 \), this is consistent with the fact that there must be some cache-misses in the reading of \( 7200 \times 164 \) sub-matrices from RAM to \( Bcache \). For \( k=12 \), not enough of the available \( Bcache \) is utilized.

However, for values of \( 84 \leq k \leq 132 \), it does not appear that the choice of block size plays a significant role in achieving high cache-efficiency.
The MFlop rates were computed for the values of \( k = 12, 84, 108, 132 \) and 164 in Table 4.3 using the formula,

\[
\text{MFlops} = 4 \times (7200)^2 / \text{wall time}
\]

where, \( \text{wall time} \) is measured by MPI_WTime.

<table>
<thead>
<tr>
<th>#P</th>
<th>164</th>
<th>132</th>
<th>108</th>
<th>84</th>
<th>12</th>
<th>Cache-Efficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>448.69</td>
<td>467.48</td>
<td>467.09</td>
<td>466.11</td>
<td>462.77</td>
<td>411.09</td>
</tr>
<tr>
<td>4</td>
<td>1757.78</td>
<td>1820.78</td>
<td>1825.78</td>
<td>1826.50</td>
<td>1750.32</td>
<td>1606.08</td>
</tr>
<tr>
<td>9</td>
<td>3975.31</td>
<td>4140.74</td>
<td>4131.41</td>
<td>4111.00</td>
<td>3880.32</td>
<td>3604.71</td>
</tr>
<tr>
<td>16</td>
<td>6940.69</td>
<td>7177.17</td>
<td>7211.44</td>
<td>7222.42</td>
<td>6772.44</td>
<td>6390.10</td>
</tr>
<tr>
<td>25</td>
<td>10785.90</td>
<td>11132.68</td>
<td>11203.15</td>
<td>11227.36</td>
<td>10376.72</td>
<td>10001.40</td>
</tr>
<tr>
<td>36</td>
<td>15397.82</td>
<td>15804.84</td>
<td>15968.23</td>
<td>15995.59</td>
<td>14744.88</td>
<td>14395.06</td>
</tr>
<tr>
<td>49</td>
<td>20814.25</td>
<td>21311.24</td>
<td>21538.76</td>
<td>21658.04</td>
<td>19739.39</td>
<td>19511.83</td>
</tr>
<tr>
<td>64</td>
<td>27118.46</td>
<td>27796.09</td>
<td>28081.23</td>
<td>28329.13</td>
<td>26274.98</td>
<td>25512.54</td>
</tr>
<tr>
<td>81</td>
<td>34485.29</td>
<td>35327.92</td>
<td>35753.42</td>
<td>35942.31</td>
<td>33175.95</td>
<td>32030.74</td>
</tr>
<tr>
<td>100</td>
<td>42542.35</td>
<td>43631.19</td>
<td>44043.93</td>
<td>44368.16</td>
<td>40550.05</td>
<td>39757.59</td>
</tr>
</tbody>
</table>

Table 4.3: Total MFlops of Cache-Efficient Block Cyclic PDGEMVT for different values of \( k \) and Total MFlops of Non-Cache Efficient PDGEMVT (Each rate is the average of 3 runs).

The numbers in Table 4.3 are also graphed in Figure 4.1. These plots are sometimes called isogranularity curves [12]. The scalability of a parallel algorithm degrades to the extent that the isogranularity curve lies below a straight line. Thus we see that \( k = 84 \) gives the most perfect scalability for the block-cyclic cache-efficient parallel GEMVT. Of interest, is that the reference code, while also a straight line on this plot, lies below all the curves for the cache-efficient parallel GEMVT for the values of \( k = 12, 84, 108, 132 \) and 164.
Figure 4.1: Total MFlops of Cache-Efficient Block Cyclic PDGEMVT for different values of \( k \) and Total MFlops of Non-Cache Efficient PDGEMVT
4.3 Performance Analysis for the Column-Cyclic Cache-efficient Parallel GEMVT code and Comparisons with a non-cache efficient GEMVT code

In this section, the Cache-Efficient Column cyclic parallel GEMVT is analyzed for performance and compared with the non-cache efficient reference code.

The global matrix size when \( P \) processors are utilized is taken as \( M \times N = \sqrt{P} \times 7200 \times \sqrt{P} \times 7200 \). To achieve a direct comparison with the block-cyclic code, we partition \( N = \sqrt{P} \times 7200 \) columns equally among \( P \) processors, each having \( n = (\sqrt{P} \times 7200)/P \) columns. Then each \( \text{sub}(A) \) on each processor is of dimension \( m \times n = \sqrt{P} \times 7200 \times 7200/\sqrt{P} \) and therefore has \((7200)^2 = 51.84 \) million matrix elements, the same as we used in our timing analysis for the block-cyclic cache-efficient parallel GEMVT.

The column cyclic cache-efficient GEMVT was run using a process grid \( 1 \times P \), yielding timing numbers and MFlop rates for 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 processors. To determine the influence of the column blocking of \( \text{sub}(A) \) on each processor, this was done by determining various block sizes \( k \), such that, \( m \times k \) fills 112.61%, 90.63%, 74.16%, 57.68% and 8.24% of \( B_{cache} \) (same amount of \( B_{cache} \) is filled in block cyclic cache-efficient parallel GEMVT with \( k = 164, 132, 108, 84, 12 \)). To run our GEMVT on 4 processors using column cyclic data distribution, each processor will have \( \sqrt{P} \cdot 7200 = 2 \cdot 7200 = 14400 \) rows and \( 7200/\sqrt{P} = 7200/2 = 3600 \) columns. And the various column block sizes of \( k = 80, 64, 52, 40, 4 \) fill 109.86%, 87.89%, 71.41%, 54.93% and 5.49% of \( B_{cache} \) respectively. For \( P = 9, 16, 25, 36, 49, 64, 81, 100 \) the choices of block sizes which fill about the same percentage of the \( B_{cache} \) are given in Table 4.5.

The computation time, wall time and the communication times for these runs are displayed in Table 4.4, along with the data for the non-cache efficient reference code. Here all the timing numbers are the average of 3 runs on the Lemieux cluster. For the column-cyclic code there is only one call to DGSUM2D to sum the \( w \) vectors at the end of the code. The computation times listed in Table 4.4 were obtained by measuring the run time of the code with the one call to DGSUM2D commented.
### Table 4.4: Average Timings of Cache-Efficient Column Cyclic PDGEMVT with different block sizes k Compared with Average Timings of Non-Cache Efficient PDGEMVT (Average of 3 runs)

<table>
<thead>
<tr>
<th>NPROCS = #Processor</th>
<th>Comp Time</th>
<th>Wall Time</th>
<th>Comm Time</th>
<th>Comp Time</th>
<th>Wall Time</th>
<th>Comm Time</th>
<th>Wall Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.4616754</td>
<td>0.4614340</td>
<td>0.0002144</td>
<td>0.4425911</td>
<td>0.4424418</td>
<td>0.0001493</td>
<td>0.504412</td>
</tr>
<tr>
<td>4</td>
<td>0.4641732</td>
<td>0.4689519</td>
<td>0.0047788</td>
<td>0.4492539</td>
<td>0.4533567</td>
<td>0.0041028</td>
<td>0.516439</td>
</tr>
<tr>
<td>9</td>
<td>0.4686982</td>
<td>0.4803280</td>
<td>0.0116298</td>
<td>0.4583117</td>
<td>0.4702142</td>
<td>0.0119025</td>
<td>0.517723</td>
</tr>
<tr>
<td>16</td>
<td>0.4787932</td>
<td>0.4986204</td>
<td>0.0198271</td>
<td>0.4674879</td>
<td>0.4877448</td>
<td>0.0202569</td>
<td>0.519215</td>
</tr>
<tr>
<td>25</td>
<td>0.4850572</td>
<td>0.5121060</td>
<td>0.0270488</td>
<td>0.4749102</td>
<td>0.5037310</td>
<td>0.028208</td>
<td>0.518329</td>
</tr>
<tr>
<td>36</td>
<td>0.4758658</td>
<td>0.5111428</td>
<td>0.0352770</td>
<td>0.4800936</td>
<td>0.5157575</td>
<td>0.0356639</td>
<td>0.518578</td>
</tr>
<tr>
<td>49</td>
<td>0.4864380</td>
<td>0.5378897</td>
<td>0.0514518</td>
<td>0.4925941</td>
<td>0.5374122</td>
<td>0.0448180</td>
<td>0.520744</td>
</tr>
<tr>
<td>64</td>
<td>0.5389879</td>
<td>0.5885833</td>
<td>0.0495954</td>
<td>0.5374437</td>
<td>0.5854449</td>
<td>0.0480012</td>
<td>0.520180</td>
</tr>
<tr>
<td>81</td>
<td>0.5028612</td>
<td>0.5755114</td>
<td>0.0726502</td>
<td>0.5112455</td>
<td>0.5690284</td>
<td>0.057829</td>
<td>0.524398</td>
</tr>
<tr>
<td>100</td>
<td>0.5183743</td>
<td>0.5860430</td>
<td>0.0676687</td>
<td>0.5132882</td>
<td>0.5831511</td>
<td>0.0698629</td>
<td>0.521564</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NPROCS = #Processor</th>
<th>Comp Time</th>
<th>Wall Time</th>
<th>Comm Time</th>
<th>Comp Time</th>
<th>Wall Time</th>
<th>Comm Time</th>
<th>Wall Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.4430053</td>
<td>0.4426781</td>
<td>0.0003272</td>
<td>0.4433084</td>
<td>0.4440945</td>
<td>0.0002139</td>
<td>0.504412</td>
</tr>
<tr>
<td>4</td>
<td>0.4485544</td>
<td>0.4526586</td>
<td>0.0014104</td>
<td>0.4509312</td>
<td>0.4541313</td>
<td>0.0032001</td>
<td>0.516439</td>
</tr>
<tr>
<td>9</td>
<td>0.460518</td>
<td>0.4710953</td>
<td>0.0110435</td>
<td>0.4638233</td>
<td>0.4737333</td>
<td>0.0099100</td>
<td>0.517723</td>
</tr>
<tr>
<td>16</td>
<td>0.4712159</td>
<td>0.4887532</td>
<td>0.0175373</td>
<td>0.4752375</td>
<td>0.4911718</td>
<td>0.0159343</td>
<td>0.519215</td>
</tr>
<tr>
<td>25</td>
<td>0.4778094</td>
<td>0.5031283</td>
<td>0.0253189</td>
<td>0.4845732</td>
<td>0.5037952</td>
<td>0.0192220</td>
<td>0.518329</td>
</tr>
<tr>
<td>36</td>
<td>0.4855518</td>
<td>0.5162380</td>
<td>0.0306862</td>
<td>0.4943959</td>
<td>0.5223770</td>
<td>0.0279411</td>
<td>0.518578</td>
</tr>
<tr>
<td>49</td>
<td>0.5013212</td>
<td>0.5391394</td>
<td>0.0378182</td>
<td>0.5013231</td>
<td>0.5387251</td>
<td>0.0374020</td>
<td>0.520744</td>
</tr>
<tr>
<td>64</td>
<td>0.5422593</td>
<td>0.5882786</td>
<td>0.0460193</td>
<td>0.5482821</td>
<td>0.5932134</td>
<td>0.0449313</td>
<td>0.520180</td>
</tr>
<tr>
<td>81</td>
<td>0.5112591</td>
<td>0.5690735</td>
<td>0.0578144</td>
<td>0.5300112</td>
<td>0.5811505</td>
<td>0.051392</td>
<td>0.524398</td>
</tr>
<tr>
<td>100</td>
<td>0.5336889</td>
<td>0.5968023</td>
<td>0.063134</td>
<td>0.5336782</td>
<td>0.5978030</td>
<td>0.0641248</td>
<td>0.521564</td>
</tr>
<tr>
<td># Processor</td>
<td>Block Size k</td>
<td>% of Bcache filled on each processor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>109.86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>87.89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>71.41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>54.93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5.49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>107.12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>90.64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>36</td>
<td>74.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>57.68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>109.86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>87.89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>24</td>
<td>65.92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>54.93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5.49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>109.86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>82.40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>20</td>
<td>68.66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>54.93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6.87</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>98.88</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>82.40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>16</td>
<td>65.92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>49.44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>96.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>76.90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>12</td>
<td>57.68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>57.68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9.61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Percentage of Bcache filled for different block size k on each processor. ($M = \sqrt{P} \cdot 7200$)
Table 4.5: Percentage of \textit{Bcache} filled for different block size $k$ on each processor. ($M = \sqrt{P} \cdot 7200$) (Continued)

<table>
<thead>
<tr>
<th># Processor</th>
<th>Block Size ( k )</th>
<th>% of \textit{Bcache} filled on each processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>109.86</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>87.89</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>12</td>
<td>65.92</td>
</tr>
<tr>
<td>8</td>
<td>43.95</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10.99</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>98.88</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>12</td>
<td>74.16</td>
</tr>
<tr>
<td>8</td>
<td>49.44</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12.36</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>109.86</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>12</td>
<td>82.40</td>
</tr>
<tr>
<td>8</td>
<td>54.93</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13.73</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Total MFlops of Cache-Efficient Column Cyclic PDGEMVT and Total MFlops of Non-Cache Efficient PDGEMVT (Each rate is the average of 3 runs).

<table>
<thead>
<tr>
<th>#P</th>
<th>% of \textit{Bcache} Filled</th>
<th>Non Cache-Efficient</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>112.61%</td>
<td>90.63%</td>
</tr>
<tr>
<td>1</td>
<td>448.69</td>
<td>467.48</td>
</tr>
<tr>
<td>4</td>
<td>1768.73</td>
<td>1829.58</td>
</tr>
<tr>
<td>9</td>
<td>3885.66</td>
<td>3969.28</td>
</tr>
<tr>
<td>16</td>
<td>6655.07</td>
<td>6804.33</td>
</tr>
<tr>
<td>25</td>
<td>10123.06</td>
<td>10291.41</td>
</tr>
<tr>
<td>36</td>
<td>14606.12</td>
<td>14475.93</td>
</tr>
<tr>
<td>64</td>
<td>22550.49</td>
<td>22670.62</td>
</tr>
<tr>
<td>81</td>
<td>29185.14</td>
<td>29517.91</td>
</tr>
<tr>
<td>100</td>
<td>35384.50</td>
<td>35560.04</td>
</tr>
</tbody>
</table>

For the purpose of comparison with the non-cache efficient reference code, we list in Table 4.6 the total MFlops of the cache-efficient column-cyclic PDGEMVT code and the total MFlops of the non-cache efficient code. Here the values of block
Figure 4.2: Total MFlops of Cache-Efficient Column Cyclic PDGEMVT for different values of k and Total MFlops of Non-Cache Efficient PDGEMVT
size \( k \) for \( P \) processors is different for each \( P \) but chosen from Table 4.5 so that the percentage of \( Bcache \) filled is closest to 112.61\%, 90.63\%, 74.16\%, 57.68\% and 8.24\% for each \( P \).

It will be observed that the best cache-efficiency is achieved when about 90.63\% of \( Bcache \) is filled with matrix elements. However, our column-cyclic GEMVT did not perform as well as the cache-efficient block cyclic GEMVT. On the other hand, it performed better than the reference code up to 25 processors, after which the reference code was better. The MFlop rates were computed in Table 4.6 using the formula,

\[
MFlops = \frac{4 \times m \times n}{\text{wall time}}
\]

where, \( \text{wall time} \) is measured by MPI\_WTime.

The numbers in Table 4.6 are also graphed in Figure 4.2. The column cyclic cache-efficient GEMVT is not perfectly scalable like the block cyclic cache-efficient GEMVT, and the degradation from perfect scalability is quite evident after 25 processors, where the curves are no longer straight lines.

### 4.4 Communication Analysis for Column Cyclic PDGEMVT code

In this section, we discuss a simple model for the cost of the communication required in the column-cyclic cache-efficient parallel GEMVT code. The only communication in this code is the call to DGSUM2D after the contributions to the final \( W^{(p)} \) vector are computed on each processor \( p \). This BLACS routine makes a call to the MPI\_AllReduce routine. Here we shall assume that the DGSUM2D call is equivalent to implementing an ordinary MPI\_AllReduce. Since the final \( W^{(p)} \) vectors available on each processor \( p \) after the completion of the two matrix vector multiplications have the same length which is equal to the number of rows of \( \text{sub}(A) \), i.e., \( m = \sqrt{P} \cdot 7200 \), the MPI\_AllReduce call consists of performing the sum

\[
W = \sum_{i=1}^{i=p} W^{(i)}
\]
where $W^{(i)}$ is a vector of length $m$ on $i^{th}$ processor. Since all processors must be involved in this summation operation and since $W$ vector must be deposited on each processor after completion, the usual model requires one fan-in and one fan-out each involving $\log_2(P)$ stages, to accomplish the above sum and place $W$ on all processors.

For any parallel code, any communication operation requires:

(i) the latency time for starting the message, $T_{S_{\text{node}}}$ and

(ii) the time to get the message passed, $T_{M_{\text{node}}}$

Therefore the total communication time, $T_{\text{comm}}$, is measured as:

$$T_{\text{comm}} = T_{S_{\text{node}}} + T_{M_{\text{node}}}$$

If the bandwidth of the Quadrics interconnection network on the Lemieux is $BW = 250 \text{ MB/Sec}$ (Megabytes/Sec), then the time required for moving 1 byte from any processor to any other processor is $\frac{1}{BW} = \frac{1}{25} \cdot 10^{-8}$ seconds. In general when $P$ is not a power of 2, we should round $\log_2(P)$ up to the next integer to allow for the time of one additional stage. So if,

$$\lceil \log_2(P) \rceil = \text{ceil} (\log_2(p))$$

the formula for the time for a fan-out plus a fan-in is

$$2 \cdot \lceil \log_2(P) \rceil \cdot \frac{8m}{BW} \text{ seconds},$$

since there are a total of $2 \cdot \lceil \log_2(P) \rceil$ stages. In addition, the latency time (time to start up a message from one processor to another) must be added for each stage required to be executed. If $T_{S_{\text{node}}}$ is this latency time, then we have a total latency for all the stages for a fan-in and fan-out of $2 \cdot \lceil \log_2(P) \rceil \cdot T_{S_{\text{node}}} \text{ seconds}$.

Accordingly, this model for communication time gives the total time for the DGSUM2D call as:

$$T_{\text{comm}} = 2 \cdot \lceil \log_2(P) \rceil \cdot (T_{S_{\text{node}}} + \frac{8m}{BW})$$

Here we computed $T_{S_{\text{node}}}$ by running a single call to MPI_AllReduce with a vector of length 2. Then $\frac{16}{BW}$ is negligible, so $T_{S_{\text{node}}}$ can be computed by dividing the time for MPI_AllReduce by $2 \cdot \lceil \log_2(P) \rceil$, i.e.,

$$T_{S_{\text{node}}} = \frac{T_{\text{comm}}}{2 \cdot \lceil \log_2(P) \rceil}.$$
\[ m = \sqrt{P} \cdot 7200 \]
\[ BW = 250 \text{ MB/Sec} \]
\[ TS_{\text{node}} = 6.34137 \times 10^{-6} \text{ Sec} \]

<table>
<thead>
<tr>
<th>#P</th>
<th>\log_2(P)</th>
<th>\lceil \log_2(P) \rceil</th>
<th>TM_{\text{node}} = 8 \cdot m / BW</th>
<th>T_{\text{comm}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.00</td>
<td>2</td>
<td>0.0004608</td>
<td>0.0018686</td>
</tr>
<tr>
<td>9</td>
<td>3.17</td>
<td>4</td>
<td>0.0006912</td>
<td>0.0055803</td>
</tr>
<tr>
<td>16</td>
<td>4.00</td>
<td>4</td>
<td>0.0009216</td>
<td>0.0074235</td>
</tr>
<tr>
<td>25</td>
<td>4.64</td>
<td>5</td>
<td>0.0011520</td>
<td>0.0115834</td>
</tr>
<tr>
<td>36</td>
<td>5.17</td>
<td>6</td>
<td>0.0013824</td>
<td>0.0166649</td>
</tr>
<tr>
<td>49</td>
<td>5.61</td>
<td>6</td>
<td>0.0016128</td>
<td>0.0194297</td>
</tr>
<tr>
<td>64</td>
<td>6.00</td>
<td>6</td>
<td>0.0018432</td>
<td>0.0221945</td>
</tr>
<tr>
<td>81</td>
<td>6.34</td>
<td>7</td>
<td>0.0020736</td>
<td>0.0291192</td>
</tr>
<tr>
<td>100</td>
<td>6.64</td>
<td>7</td>
<td>0.0023040</td>
<td>0.0323448</td>
</tr>
</tbody>
</table>

Table 4.7: Theoretical Communication times for column cyclic PDGEMVT for matrices with \( m = \sqrt{P} \cdot 7200 \) rows on each processor.

Here we take \( TS_{\text{node}} \) to be the average value obtained for these runs over \( P = 2, 4, 8, 16, 32, 64 \), which came to \( TS_{\text{node}} = 6.34137 \times 10^{-6} \) sec.

In Table 4.7, we have logged the values of the above theoretical communication times, \( T_{\text{comm}} \), as a function of \( P \). For comparison we computed experimentally observed communication times in two ways as follows:

(i) MPI_WTime (wall time) were computed for the whole code and then with the DGSUM2D call commented, for the same set of \( k \) values used in Table 4.5. The experimentally determined communication was then computed by taking the difference of these times (\( T_{\text{comm}}^1 \)).

(ii) MPI_WTime were computed for the code which was commented for all computation statements and ran only the DGSUM2D call (\( T_{\text{comm}}^2 \)).

In Table 4.8, the experimental communication times are logged for several \( k \) values, and compared with the theoretical communication times from Table 4.7.
<table>
<thead>
<tr>
<th>#P</th>
<th>k</th>
<th>% of Bcache Filled</th>
<th>Total Time</th>
<th>Comp Time</th>
<th>$T^{1}_{comm}$</th>
<th>$T^{2}_{comm}$</th>
<th>$T^{Th}_{comm}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>80</td>
<td>109.86</td>
<td>0.468951</td>
<td>0.464173</td>
<td>0.004778</td>
<td>0.003642</td>
<td>0.001869</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>87.89</td>
<td>0.453356</td>
<td>0.449253</td>
<td>0.004102</td>
<td>0.004010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>71.41</td>
<td>0.452658</td>
<td>0.448554</td>
<td>0.004104</td>
<td>0.003610</td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>54.93</td>
<td>0.454131</td>
<td>0.450931</td>
<td>0.003200</td>
<td>0.003120</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>52</td>
<td>107.12</td>
<td>0.480328</td>
<td>0.468698</td>
<td>0.011629</td>
<td>0.011301</td>
<td>0.005580</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>90.64</td>
<td>0.470214</td>
<td>0.458311</td>
<td>0.011902</td>
<td>0.011391</td>
<td></td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>74.16</td>
<td>0.471095</td>
<td>0.460051</td>
<td>0.011043</td>
<td>0.010490</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>57.68</td>
<td>0.473733</td>
<td>0.463823</td>
<td>0.009910</td>
<td>0.009226</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>40</td>
<td>109.86</td>
<td>0.498620</td>
<td>0.478793</td>
<td>0.019827</td>
<td>0.019952</td>
<td>0.007423</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>87.89</td>
<td>0.487744</td>
<td>0.467487</td>
<td>0.020256</td>
<td>0.021016</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>69.92</td>
<td>0.488753</td>
<td>0.471215</td>
<td>0.017537</td>
<td>0.018002</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>54.93</td>
<td>0.491171</td>
<td>0.475237</td>
<td>0.015934</td>
<td>0.016287</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>32</td>
<td>109.86</td>
<td>0.512106</td>
<td>0.485057</td>
<td>0.027048</td>
<td>0.027631</td>
<td>0.011583</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>82.40</td>
<td>0.503731</td>
<td>0.474910</td>
<td>0.028820</td>
<td>0.029491</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>68.66</td>
<td>0.503128</td>
<td>0.477809</td>
<td>0.025318</td>
<td>0.025709</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>54.93</td>
<td>0.503795</td>
<td>0.484573</td>
<td>0.027941</td>
<td>0.026961</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>98.88</td>
<td>0.511142</td>
<td>0.475865</td>
<td>0.035277</td>
<td>0.035680</td>
<td>0.016666</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>82.40</td>
<td>0.515757</td>
<td>0.480093</td>
<td>0.035663</td>
<td>0.035807</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>65.92</td>
<td>0.516238</td>
<td>0.485551</td>
<td>0.030686</td>
<td>0.030853</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>49.44</td>
<td>0.522337</td>
<td>0.494395</td>
<td>0.027941</td>
<td>0.026961</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>20</td>
<td>96.13</td>
<td>0.537889</td>
<td>0.486438</td>
<td>0.051451</td>
<td>0.050555</td>
<td>0.019430</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>76.90</td>
<td>0.537412</td>
<td>0.492594</td>
<td>0.044818</td>
<td>0.043803</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>57.68</td>
<td>0.539139</td>
<td>0.501321</td>
<td>0.037818</td>
<td>0.036427</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>38.45</td>
<td>0.538725</td>
<td>0.501323</td>
<td>0.037402</td>
<td>0.036515</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>20</td>
<td>109.86</td>
<td>0.588583</td>
<td>0.538987</td>
<td>0.049959</td>
<td>0.049629</td>
<td>0.022194</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>87.89</td>
<td>0.585444</td>
<td>0.537443</td>
<td>0.048001</td>
<td>0.048092</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>65.92</td>
<td>0.588278</td>
<td>0.542259</td>
<td>0.046019</td>
<td>0.045726</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>43.95</td>
<td>0.593213</td>
<td>0.548282</td>
<td>0.044931</td>
<td>0.045362</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>16</td>
<td>98.88</td>
<td>0.575511</td>
<td>0.502861</td>
<td>0.072650</td>
<td>0.072946</td>
<td>0.029119</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>74.16</td>
<td>0.569028</td>
<td>0.511245</td>
<td>0.057782</td>
<td>0.058142</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>49.44</td>
<td>0.569073</td>
<td>0.511259</td>
<td>0.057814</td>
<td>0.058133</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>24.72</td>
<td>0.581150</td>
<td>0.530011</td>
<td>0.051139</td>
<td>0.051211</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>16</td>
<td>109.86</td>
<td>0.586043</td>
<td>0.518374</td>
<td>0.067668</td>
<td>0.068082</td>
<td>0.032345</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>82.40</td>
<td>0.583151</td>
<td>0.513288</td>
<td>0.069862</td>
<td>0.070530</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>54.93</td>
<td>0.596802</td>
<td>0.533688</td>
<td>0.063113</td>
<td>0.062934</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>27.47</td>
<td>0.597803</td>
<td>0.533678</td>
<td>0.064124</td>
<td>0.061624</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.8: Experimental Communication times for column cyclic PDGEMVT for matrices with $m = \sqrt{P} \cdot 7200$ rows on each processor.
4.5 Communication Analysis for Block Cyclic PDGEMVT code

In this section we discuss a simple model for the cost of the communication required in the block-cyclic cache-efficient parallel GEMVT code. Here we have column communicators executing DGSUM2D inside the loop over \( i = 1, b \) to sum the \( X \) vectors of length \( k \), the block size, in the pseudo-code discussed in Section 3.4.1. The number of processors involved in each column communicators is \( \sqrt{P} \). Moreover, there are \( b \) DGSUM2D calls over these \( \sqrt{P} \) processors. The communication time to sum \( X \) vector is measured as

\[
b \cdot 2 \cdot \lceil \log_2(\sqrt{P}) \rceil \cdot (T_{node} + \frac{8 \cdot k}{BW}) \text{ seconds.}
\]

And the communication time to sum the \( W \) vectors of length \( m \), among \( \sqrt{P} \) processors is measured as:

\[
2 \cdot \lceil \log_2(\sqrt{P}) \rceil \cdot (T_{node} + \frac{8 \cdot m}{BW}) \text{ seconds.}
\]

The sum of these values gives the total communication time for the block-cyclic cache-efficient PDGEMVT code. Therefore, \( T_{comm} \) should be measured as

\[
T_{comm} = b \cdot 2 \cdot \lceil \log_2(\sqrt{P}) \rceil \cdot (T_{node} + \frac{8 \cdot k}{BW}) + 2 \cdot \lceil \log_2(\sqrt{P}) \rceil \cdot (T_{node} + \frac{8 \cdot m}{BW})
\]

\[
= 2 \cdot \lceil \log_2(\sqrt{P}) \rceil \cdot \{b \cdot (T_{node} + \frac{8 \cdot k}{BW}) + (T_{node} + \frac{8 \cdot m}{BW})\}
\]

In Table 4.9, we have logged the values of the above theoretical total communication times, \( T_{comm} \), as a function of \( P \). For comparison we computed experimentally observed communication times in two ways as follows:

(i) MPI_WTimes (wall time) were computed for the whole code. The experimentally determined communication time \( (T_{comm}^1) \) was then computed by subtracting the time it take to run the 7200 \( \times \) 7200 matrix on one processor with the DGSUM2D calls commented.

(ii) MPI_WTime were computed for the code which was commented for all computation statements and ran only the DGSUM2D calls \( (T_{comm}^2) \).
\[ m = 7200 \]
\[ BW = 250 \text{ MB/Sec} \]
\[ TS_{\text{node}} = 6.34137 \times 10^{-6} \text{ Sec} \]
\[ T_1 = b \cdot (TS_{\text{node}} + 8 \cdot k/BW) \text{ Sec} \]
\[ T_2 = TS_{\text{node}} + 8 \cdot m/BW = 0.0002943 \text{ Sec} \]

<table>
<thead>
<tr>
<th>#P</th>
<th>(2 \cdot \log_2(\sqrt{P}))</th>
<th>(k)</th>
<th>(b = \lceil m/k \rceil)</th>
<th>(T_1)</th>
<th>(T_{\text{comm}} = T_1 + T_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>164</td>
<td>44</td>
<td>0.0005677</td>
<td>0.0017240</td>
</tr>
<tr>
<td></td>
<td></td>
<td>132</td>
<td>55</td>
<td>0.0006392</td>
<td>0.0018670</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108</td>
<td>67</td>
<td>0.0007143</td>
<td>0.0020173</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84</td>
<td>86</td>
<td>0.0008343</td>
<td>0.0022573</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>164</td>
<td>44</td>
<td>0.0005677</td>
<td>0.0034480</td>
</tr>
<tr>
<td></td>
<td></td>
<td>132</td>
<td>55</td>
<td>0.0006392</td>
<td>0.0037341</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108</td>
<td>67</td>
<td>0.0007143</td>
<td>0.0040346</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84</td>
<td>86</td>
<td>0.0008343</td>
<td>0.0045146</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>164</td>
<td>44</td>
<td>0.0005677</td>
<td>0.0034480</td>
</tr>
<tr>
<td></td>
<td></td>
<td>132</td>
<td>55</td>
<td>0.0006392</td>
<td>0.0037341</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108</td>
<td>67</td>
<td>0.0007143</td>
<td>0.0040346</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84</td>
<td>86</td>
<td>0.0008343</td>
<td>0.0045146</td>
</tr>
<tr>
<td>25</td>
<td>6</td>
<td>164</td>
<td>44</td>
<td>0.0005677</td>
<td>0.0051720</td>
</tr>
<tr>
<td></td>
<td></td>
<td>132</td>
<td>55</td>
<td>0.0006392</td>
<td>0.0056011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108</td>
<td>67</td>
<td>0.0007143</td>
<td>0.0060519</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84</td>
<td>86</td>
<td>0.0008343</td>
<td>0.0067720</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>164</td>
<td>44</td>
<td>0.0005677</td>
<td>0.0051720</td>
</tr>
<tr>
<td></td>
<td></td>
<td>132</td>
<td>55</td>
<td>0.0006392</td>
<td>0.0056011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108</td>
<td>67</td>
<td>0.0007143</td>
<td>0.0060519</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84</td>
<td>86</td>
<td>0.0008343</td>
<td>0.0067720</td>
</tr>
<tr>
<td>49</td>
<td>6</td>
<td>164</td>
<td>44</td>
<td>0.0005677</td>
<td>0.0051720</td>
</tr>
<tr>
<td></td>
<td></td>
<td>132</td>
<td>55</td>
<td>0.0006392</td>
<td>0.0056011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108</td>
<td>67</td>
<td>0.0007143</td>
<td>0.0060519</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84</td>
<td>86</td>
<td>0.0008343</td>
<td>0.0067720</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>164</td>
<td>44</td>
<td>0.0005677</td>
<td>0.0051720</td>
</tr>
<tr>
<td></td>
<td></td>
<td>132</td>
<td>55</td>
<td>0.0006392</td>
<td>0.0056011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108</td>
<td>67</td>
<td>0.0007143</td>
<td>0.0060519</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84</td>
<td>86</td>
<td>0.0008343</td>
<td>0.0067720</td>
</tr>
</tbody>
</table>

Table 4.9: Theoretical Communication times for block cyclic PDGEMVT for matrices with \(m = 7200\) rows on each processor.
\[ P \cdot \lceil \log_2(\sqrt{P}) \rceil \] 

\[ k \quad b = \lceil m/k \rceil \quad T_1 \quad T_{comm} = T_1 + T_2 \]

<table>
<thead>
<tr>
<th>( #P )</th>
<th>( 2 \cdot \lceil \log_2(\sqrt{P}) \rceil )</th>
<th>( k )</th>
<th>( b = \lceil m/k \rceil )</th>
<th>( T_1 )</th>
<th>( T_{comm} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>81</td>
<td>8</td>
<td>164</td>
<td>44</td>
<td>0.0005677</td>
<td>0.0068960</td>
</tr>
<tr>
<td></td>
<td></td>
<td>132</td>
<td>55</td>
<td>0.006392</td>
<td>0.0074681</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108</td>
<td>67</td>
<td>0.007143</td>
<td>0.0080692</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84</td>
<td>86</td>
<td>0.008343</td>
<td>0.0090293</td>
</tr>
<tr>
<td>100</td>
<td>8</td>
<td>164</td>
<td>44</td>
<td>0.0005677</td>
<td>0.0068960</td>
</tr>
<tr>
<td></td>
<td></td>
<td>132</td>
<td>55</td>
<td>0.006392</td>
<td>0.0074681</td>
</tr>
<tr>
<td></td>
<td></td>
<td>108</td>
<td>67</td>
<td>0.007143</td>
<td>0.0080692</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84</td>
<td>86</td>
<td>0.008343</td>
<td>0.0090293</td>
</tr>
</tbody>
</table>

Table 4.9: Theoretical Communication times for block cyclic PDGEMVT for matrices with \( m = 7200 \) rows on each processor (Continued).

In Table 4.10, the experimental communication times are logged for several \( k \) values, and compared with the theoretical communication times from Table 4.9.

<table>
<thead>
<tr>
<th>#P</th>
<th>( k )</th>
<th>Total Time</th>
<th>Comp Time</th>
<th>( T_{1\text{comm}} )</th>
<th>( T_{2\text{comm}} )</th>
<th>( T_{Th\text{comm}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>164</td>
<td>0.4621445</td>
<td>0.4610326</td>
<td>0.0011118</td>
<td>0.0002011</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4435661</td>
<td>0.4421574</td>
<td>0.0014086</td>
<td>0.0001824</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.443942</td>
<td>0.4426123</td>
<td>0.0013297</td>
<td>0.0001820</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4448776</td>
<td>0.4432783</td>
<td>0.0015993</td>
<td>0.0001937</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>164</td>
<td>0.4718945</td>
<td>0.4610326</td>
<td>0.0108619</td>
<td>0.0032194</td>
<td>0.0017240</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4556027</td>
<td>0.4421574</td>
<td>0.0134453</td>
<td>0.0025464</td>
<td>0.0018670</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.4543210</td>
<td>0.4426123</td>
<td>0.0117087</td>
<td>0.0028702</td>
<td>0.0020173</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4541273</td>
<td>0.4432783</td>
<td>0.010849</td>
<td>0.0031149</td>
<td>0.0022573</td>
</tr>
</tbody>
</table>

Table 4.10: Experimental Communication times for block cyclic PDGEMVT for matrices with \( m = 7200 \) rows on each processor.
<table>
<thead>
<tr>
<th>#P</th>
<th>k</th>
<th>Total Time</th>
<th>Comp Time</th>
<th>(T_{\text{comm}}^1)</th>
<th>(T_{\text{comm}}^2)</th>
<th>(T_{\text{Th}}^\text{comm})</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>164</td>
<td>0.4694585</td>
<td>0.4610326</td>
<td>0.0084259</td>
<td>0.0039917</td>
<td>0.0034480</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4507019</td>
<td>0.4421574</td>
<td>0.0085445</td>
<td>0.0037123</td>
<td>0.0037341</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.4517199</td>
<td>0.4426123</td>
<td>0.0091076</td>
<td>0.0040639</td>
<td>0.0040346</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4539631</td>
<td>0.4432783</td>
<td>0.0106848</td>
<td>0.0045735</td>
<td>0.0045146</td>
</tr>
<tr>
<td>16</td>
<td>164</td>
<td>0.4781023</td>
<td>0.4610326</td>
<td>0.0170697</td>
<td>0.0057295</td>
<td>0.0034480</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4624316</td>
<td>0.4421574</td>
<td>0.0202742</td>
<td>0.0052784</td>
<td>0.0037341</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.4617438</td>
<td>0.4432783</td>
<td>0.0215111</td>
<td>0.0055988</td>
<td>0.0037046</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4617438</td>
<td>0.4432783</td>
<td>0.0184655</td>
<td>0.0074528</td>
<td>0.0067720</td>
</tr>
<tr>
<td>25</td>
<td>164</td>
<td>0.4806795</td>
<td>0.4610326</td>
<td>0.0196469</td>
<td>0.0067032</td>
<td>0.0051720</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4657354</td>
<td>0.4421574</td>
<td>0.023578</td>
<td>0.0061418</td>
<td>0.0056011</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.4627634</td>
<td>0.4426123</td>
<td>0.0201511</td>
<td>0.0065204</td>
<td>0.0060519</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4617438</td>
<td>0.4432783</td>
<td>0.0234319</td>
<td>0.0080821</td>
<td>0.0067720</td>
</tr>
<tr>
<td>36</td>
<td>164</td>
<td>0.4882200</td>
<td>0.4610326</td>
<td>0.0271874</td>
<td>0.0094553</td>
<td>0.0051720</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4768889</td>
<td>0.4421574</td>
<td>0.0347315</td>
<td>0.0091695</td>
<td>0.0056011</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.4617438</td>
<td>0.4432783</td>
<td>0.0258844</td>
<td>0.0112467</td>
<td>0.0067720</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4617438</td>
<td>0.4432783</td>
<td>0.0234319</td>
<td>0.0080821</td>
<td>0.0067720</td>
</tr>
<tr>
<td>49</td>
<td>164</td>
<td>0.4840233</td>
<td>0.4610326</td>
<td>0.0283697</td>
<td>0.0076737</td>
<td>0.0051720</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4775189</td>
<td>0.4421574</td>
<td>0.0353615</td>
<td>0.0091695</td>
<td>0.0056011</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.462553</td>
<td>0.4426123</td>
<td>0.030043</td>
<td>0.0085595</td>
<td>0.0060519</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4684923</td>
<td>0.4432783</td>
<td>0.025214</td>
<td>0.0090533</td>
<td>0.0067720</td>
</tr>
<tr>
<td>64</td>
<td>164</td>
<td>0.4890423</td>
<td>0.4610326</td>
<td>0.0283697</td>
<td>0.0076737</td>
<td>0.0051720</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4775189</td>
<td>0.4421574</td>
<td>0.0353615</td>
<td>0.0091695</td>
<td>0.0056011</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.462553</td>
<td>0.4426123</td>
<td>0.030043</td>
<td>0.0085595</td>
<td>0.0060519</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4684923</td>
<td>0.4432783</td>
<td>0.025214</td>
<td>0.0090533</td>
<td>0.0067720</td>
</tr>
<tr>
<td>81</td>
<td>164</td>
<td>0.4870533</td>
<td>0.4610326</td>
<td>0.0260227</td>
<td>0.0095627</td>
<td>0.0068960</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4754393</td>
<td>0.4421574</td>
<td>0.0332819</td>
<td>0.0072480</td>
<td>0.0074681</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.4697844</td>
<td>0.4426123</td>
<td>0.0271721</td>
<td>0.0076768</td>
<td>0.0080692</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4673136</td>
<td>0.4432783</td>
<td>0.0240353</td>
<td>0.0085860</td>
<td>0.0090293</td>
</tr>
<tr>
<td>100</td>
<td>164</td>
<td>0.4874253</td>
<td>0.4610326</td>
<td>0.0263927</td>
<td>0.0082369</td>
<td>0.0068960</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>0.4752659</td>
<td>0.4421574</td>
<td>0.0331085</td>
<td>0.0072831</td>
<td>0.0074681</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0.4708126</td>
<td>0.4426123</td>
<td>0.0282003</td>
<td>0.0076824</td>
<td>0.0080692</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0.4673136</td>
<td>0.4432783</td>
<td>0.0240934</td>
<td>0.0084667</td>
<td>0.0090293</td>
</tr>
</tbody>
</table>

Table 4.10: Experimental Communication times for block cyclic PDGEMVT for matrices with \( m = 7200 \) rows on each processor (Continued).
4.6 Scaled Speedups

Parallel Speed-Up is defined (see [11], [12]) as

\[
S_p = \frac{\text{Execution time for matrix size } \sqrt{P} \cdot g \times \sqrt{P} \cdot g \text{ on one processor}}{\text{Parallel Execution time on } P \text{ processors, each with } g \times g \text{ matrices}}
\]  

(4.1)

And parallel efficiency is defined as

\[
E_p = \frac{S_p}{P}
\]  

(4.2)

With \( g = 7200 \) and \( P = 4, 9 \) the matrices of size \( \sqrt{P} \cdot 7200 \times \sqrt{P} \cdot 7200 \) fit in the 4 GB of RAM on one processor on the Lemieux cluster. The timing numbers for the cache-efficient block-cyclic and column-cyclic GEMVT for the global matrix sizes 14400 × 14400 and 21600 × 21600 on a single processor is shown in Table 4.11. And also the timing numbers for the non-cache efficient GEMVT is shown in Table 4.12.

<table>
<thead>
<tr>
<th>( P )</th>
<th>( m )</th>
<th>( n )</th>
<th>( k )</th>
<th>Wall Time</th>
<th>( S_p )</th>
<th>( E_p )</th>
<th>Ideal ( S_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14400</td>
<td>14400</td>
<td>40</td>
<td>1.71893</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>B-Cyclic</td>
<td>4</td>
<td>7200</td>
<td>7200</td>
<td>84</td>
<td>0.45413</td>
<td>3.78511</td>
<td>0.94628</td>
</tr>
<tr>
<td>C-Cyclic</td>
<td>4</td>
<td>14400</td>
<td>3600</td>
<td>40</td>
<td>0.45413</td>
<td>3.78511</td>
<td>0.94628</td>
</tr>
<tr>
<td>1</td>
<td>21600</td>
<td>21600</td>
<td>44</td>
<td>4.06361</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>B-Cyclic</td>
<td>9</td>
<td>7200</td>
<td>7200</td>
<td>84</td>
<td>0.45396</td>
<td>8.95147</td>
<td>0.99461</td>
</tr>
<tr>
<td>C-Cyclic</td>
<td>9</td>
<td>21600</td>
<td>2400</td>
<td>44</td>
<td>0.47373</td>
<td>8.57790</td>
<td>0.95310</td>
</tr>
</tbody>
</table>

Table 4.11: Scaled Speedups for Block-cyclic and Column-cyclic Cache-Efficient GEMVT

<table>
<thead>
<tr>
<th>( P )</th>
<th>( m )</th>
<th>( n )</th>
<th>Wall Time</th>
<th>( S_p )</th>
<th>( E_p )</th>
<th>Ideal ( S_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14400</td>
<td>14000</td>
<td>2.03139</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>4</td>
<td>7200</td>
<td>7200</td>
<td>0.51644</td>
<td>3.93345</td>
<td>0.98336</td>
<td>4.00000</td>
</tr>
<tr>
<td>1</td>
<td>21600</td>
<td>21600</td>
<td>4.60783</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>9</td>
<td>7200</td>
<td>7200</td>
<td>0.51772</td>
<td>8.90018</td>
<td>0.98891</td>
<td>9.00000</td>
</tr>
</tbody>
</table>

Table 4.12: Scaled Speedups for Non-Cache Efficient GEMVT
4.7 Conclusions

The new cache-efficient block-cyclic code performed on the Lemieux cluster significantly better than the reference code which makes two straight calls to PDGEMV with no blocking to achieve cache-efficiency on the nodes. From Table 4.1 we observe that with each processor operating on a $7200 \times 7200$ sub($A$) matrix, the new code ran in about 91% of the time required for the reference code when the $Bcache$ was filled to 90.63% ($k = 132$) of the 8 MB, and in about 89% of the time when the $Bcache$ was filled to only 51.68% ($k = 84$) of the 8 MB. This was the best range of percentage of $Bcache$ filled, and there was significant drop off when the $Bcache$ was filled only 8.24% or to 112.6%.

The block-cyclic code was seen to be quite scalable as evidenced by the straight line graphs in Figure 4.1 for the MFlop rates as a function of the number of processors. The calculation of theoretical communication times for the block-cyclic code in Table 4.10 showed that this time increased in small jumps, but very gradually as the number of processors increased. For $k = 84$ we have 0.00451 secs when $NPROCS = 9, 16$, 0.00677 secs when $NPROCS = 25, 36, 49, 64$, and 0.00903 secs when $NPROCS = 81, 100$. This theoretical calculation is consistent with the fact that the block-cyclic code is very scalable. However, it was found difficult to separate the computation time and communication time by actual measurements on the Lemieux cluster.

The assumption that the computation time on each processor, as $NPROCS$ was increased, was the same as the time utilized by one processor executing the whole code with the communication calls commented, gave rise in Table 4.1 and Table 4.10 to significantly larger communication times than the theory predicted. For $k = 84$, for example, the observed communication times $T_{comm}^1$ were about 2-3 times the theoretical communication times for $NPROCS$ running from 16 to 100. We have no explanation for this discrepancy. Either the communication model is not accurate, or, more likely, the assumption that the computation time on one processor is OK to use when $NPROCS > 1$ is not valid. On the other hand, the measurement $T_{comm}^2$ which is only the time for all the DGSUM2D calls is seen to be surprisingly close to the theoretical communication time.
The cache-efficient column-cyclic code performed on the Lemieux cluster better than the reference code up to 25 processors, but then ran slower than the reference code for 36 - 100 processors. The code is not scalable in the range of 36 - 100 processors, as evidenced by the graphs of MFlops rates versus \( \text{NPROCS} \) in Figure 4.2. This lack of scalability can be attributed to fairly rapid increase in the communication times as \( \text{NPROCS} \) is increased. The theoretical communication times show a steady increase in Table 4.7. In this case all of the communication cost is in the final DGSUM2D call at the end of the code, and the computation time could therefore be more readily approximated by measuring the runs of the code with the one DGSUM2D call commented. The computation times in Table 4.8 go up slightly as \( \text{NPROCS} \) increases, and the resulting communication times \( T_{\text{comm}}^1 \) also slowly increase. However, it is noteworthy that these measured communication times are on the same order of magnitude as the theoretical communication times, and always slightly larger. We also note that the theoretical communication times for the column-cyclic code are about 10 times larger than for the block-cyclic code for \( \text{NPROCS} \geq 36 \). The reason is that the \( \frac{8m}{\text{BW}} \) in the \( T_{\text{comm}} \) formula is much larger for the long vectors in the column-cyclic implementation.

Because it is more scalable our cache-efficient block-cyclic code is to be preferred over the column-cyclic code for use in other parallel algorithms which require a parallel GEMVT code.
Appendix A

A.1 Driver Program for PDGEMVT

PROGRAM PDGEMTVDRIVER
*
.. Parameters ..
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER DBLESZ, MEMSIZ, TOTMEM
PARAMETER ( DBLESZ = 8, TOTMEM = 500000000,
$               MEMSIZ = TOTMEM / DBLESZ )
INTEGER BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DT_,
$               LLD_, MB_, M_, NB_, N_, RSRC_
PARAMETER ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DT_ = 1,
$               CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5,
$               NB_ = 6, RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )
*
.. Local Scalars ..
CHARACTER*80 OUTFILE
INTEGER IAM, IASEED, IBSEED, ICSEED, ICTXT, INFO,
$               IPA, IPB, IPC, IPW, IPY, IPL, N, KP, NQ, M,
$               MP, MYCOL, MYROW, V, MB, NB, NOUT, NPCOL,
$               NPROCS, NPROW, WORKSIZ, IPX, IPZ, YP, YQ,
$               ZP, ZQ
*
.. Local Arrays ..
INTEGER DESCA(DLEN_), DESCY(DLEN_)
INTEGER DESCX(DLEN_), DESCZ(DLEN_), DESCW(DLEN_)
DOUBLE PRECISION MEM( MEMSIZ)
INTEGER I, J, K, ierr
REAL E_Avg_Time, E_Avg_Mflop
REAL E_Max_Time, E_Min_Mflop
REAL W_Avg_Time, W_Avg_Mflop
REAL W_Max_Time, W_Min_Mflop
REAL e_mflops, w_mflops
REAL cpu1, user1, sys1
REAL cpu2, user2, sys2
REAL cputime, usertime, systime, w_time
REAL start, finish
* .. External Subroutines ..
  EXTERNAL  BLACS_EXIT, BLACS_GET, BLACS_GRIDEXIT,
             $ BLACS_GRIDINFO, BLACS_GRIDINIT,
             $ BLACS_PINFO, DESCINIT, PDMATGEN,
             $ PDPBLASINFO, PDGEMVT
* ..
* .. External Functions ..
  EXTERNAL  NUMROC
* ..
* .. Intrinsic Functions ..
  INTRINSIC  DBLE, MAX
* ..
* .. Executable Statements ..
*
  CALL BLACS_PINFO( IAM, NPROCS )
  CALL PDPBLASINFO( OUTFILE, NOUT, M, N, V, NB, NPROW, NPCOL,
                   $ MEM, IAM, NPROCS )

#ifdef BLOCK
* .. Block Cyclic data distribution
  MB = NB
#else
* .. Column Cyclic data distribution
  MB = M
#endif

#ifdef INPUT
  IF (IAM.EQ.0) THEN
    WRITE(*,*) 'Input arguments on processor ---> ', IAM
    WRITE(*,*) '*******************************'
    WRITE(*,*) 'M = ', M
    WRITE(*,*) '*******************************'
  END IF
#endif

* Define process grid
*
  CALL BLACS_GET( -1, 0, ICTXT )
  CALL BLACS_GRIDINIT( ICTXT, 'Row-major', NPROW, NPCOL )
  CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
* Go to bottom of process grid loop if this case doesn’t
* use my process
  IF( MYROW.GE.NPROW .OR. MYCOL.GE.NPCOL )
    GO TO 20
* ScaLAPACK tool function NUMROC returns the exact number of
* Rows and Columns on each processor.

104
MP = NUMROC( M, MB, MYROW, 0, NPROW )
NQ = NUMROC( N, NB, MYCOL, 0, NPCOL )
YQ = NUMROC( V, NB, MYCOL, 0, NPCOL )
if (YQ.eq.0) then
  YP = 0
else
  YP = MP
end if
ZP = NUMROC( V, MB, MYROW, 0, NPROW )
if (ZP.eq.0) then
  ZQ = 0
else
  ZQ = NQ
end if

#ifdef FINER
  if (iam.eq.0) then
    WRITE(*,*) 'No. of rows and cols on processor --> ', IAM
    WRITE(*,*) '********************************************
    WRITE(*,*) 'MP (rows in Matrix A ) = ', MP
    WRITE(*,*) 'NQ (cols in Matrix A ) = ', NQ
    WRITE(*,*) 'YP (rows in vector Y,and W (size)) = ', YP
    WRITE(*,*) 'YQ (cols in vector Y,and W (0 or 1)) = ', YQ
    WRITE(*,*) 'ZP (rows in vector Z,and X (size)) = ', ZP
    WRITE(*,*) 'ZQ (cols in vector Z,and X (0 or 1)) = ', ZQ
    WRITE(*,*) '********************************************
  end if
#endif

* Initialize the array descriptor for the matrix A, VECTORs
  Y, X, Z and W
*
  CALL DESCINIT( DESCA, M, N, MB, NB, 0, 0, ICTXT, MAX( 1, MP ), INFO )
  CALL DESCINIT( DESCY, M, 1, MB, NB, 0, 0, ICTXT, MAX( 1, MP ), INFO )
  CALL DESCINIT( DESCX, 1, N, MB, NB, 0, 0, ICTXT, 1, INFO )
  CALL DESCINIT( DESCZ, 1, N, MB, NB, 0, 0, ICTXT, 1, INFO )
  CALL DESCINIT( DESCW, M, 1, MB, NB, 0, 0, ICTXT, MAX( 1, MP ), INFO )

  Assign pointers into MEM for SCALAPACK arrays, A is allocated
  starting at position MEM( 1 )
  *
  IPA = 1
  IPY = IPA + DESCA( LLD_ )*NQ
  IPZ = IPY + DESCY( LLD_ )

105
IPX = IPZ + DESCZ( LLD_ )*NQ
IPW = IPX + DESCX( LLD_ )*NQ
IPL = IPW + DESCW( LLD_ )

#ifdef FINER
IF (IAM.eq.0) THEN
WRITE(*,*) 'Size of vectors on processor ', IAM
WRITE(*,*) '***********************************'
WRITE(*,*) 'IPA = ', IPA
WRITE(*,*) 'IPY = ', IPY
WRITE(*,*) 'IPZ = ', IPZ
WRITE(*,*) 'IPX = ', IPX
WRITE(*,*) 'IPW = ', IPW
WRITE(*,*) 'IPL = ', IPL
WRITE(*,*) '***********************************'
ENDIF
#endif

WORKSIZ = MB
*
* Check for adequate memory for problem size
*
INFO = 0
IF( IPL+WORKSIZ.GT.MEMSIZ ) THEN
IF( IAM.EQ.0 )
WRITE( NOUT, FMT = 9998 ) ( IPX+WORKSIZ )*DBLESZ
INFO = 1
END IF
*
* Check all processes for an error
*
CALL IGSUM2D( ICTXT, 'All', ' ', 1, 1, INFO, 1, -1, 0 )
IF( INFO.GT.0 ) THEN
IF( IAM.EQ.0 )
WRITE( NOUT, FMT = 9999 ) 'MEMORY'
GO TO 10
END IF
** ---- Generate random matrix A, vector Y and vector Z ----
IASEED = 100
CALL PDMATGEN( ICTXT, 'No transpose', 'No transpose',
$ DESCA( M_ ), DESCA( N_ ), DESCA( MB_ ),
$ DESCA( NB_ ), MEM( IPA ), DESCA( LLD_ ),
$ DESCA( RSRC_ ), DESCA( CSRC_ ), IASEED, 0,
$ MP, 0, NQ, MYROW, MYCOL, NPROW, NPCOL )
IBSEED = 200
CALL PDMATGEN( ICTXT, 'No transpose', 'No transpose',
$ DESCY( M_ ), DESCY( N_ ), DESCY( MB_ ),
$ DESCY( NB_ ), MEM( IPY ), DESCY( LLD_ ),
$ DESCY( RSRC_ ), DESCY( CSRC_ ), IBSEED, 0,
$ YP, 0, YQ, MYROW, MYCOL, NPROW, NPCOL )
ICSEED = 300
CALL PDATGEN( ICTXT, 'No transpose', 'No transpose', $ DESCZ( M_ ), DESCZ( N_ ), DESCZ( MB_ ), $ DESCZ( NB_ ), MEM( IPZ ), DESCZ( LLD_ ), $ DESCZ( RSRC_ ), DESCZ( CSRC_ ), ICSEED, 0, $ ZP, 0, ZQ, MYROW, MYCOL, NPROW, NPCOL )

** -- Send the Y vectors to other processes in each row of the row grid
IF (MYCOL.EQ.0) THEN
   CALL DGEBS2D(ICTXT, 'R', ' ', MP, 1, MEM(IPY), $ DESCY(LLD_))
ELSE
   CALL DGEBR2D(ICTXT, 'R', ' ', MP, 1, MEM(IPY), $ DESCY(LLD_), MYROW, 0)
ENDIF

#ifdef TIMING
CALL BLOCKSIZE(MP, K)
IF (IAM.EQ.0) THEN
   WRITE(*,*) 'K = ', K
END IF
#endif

****************************************************************
* Call PDGEMVT to perform operation X = Z + A' * Y             *
* W = W + A * X                                             *
****************************************************************

start = MPI_Wtime()
call cpu_second(cpu1,user1,sys1)

CALL PDGEMVT(MP, NQ, 1.d0, MEM(IPA), 1, 1, DESCA, MEM(IPX), $ 1, 1, DESCX, 1, MEM(IPY), 1, 1, DESCY, 1, 1.d0, $ MEM(IPW), 1, 1, DESCW, 1, MEM(IPZ), 1, 1, $ DESCZ, 1)
call cpu_second(cpu2,user2,sys2)
finish = MPI_Wtime()

# ifndef FINAL
if (IAM.EQ.0 .OR. MOD(IAM, NPCOL).EQ.0) then
   write(*,*), "Final value of W vector: ..Iam = ", IAM
   write(*,*), (mem(I), I = IPW,IPL-1)
endif
#endif

#ifdef TIMING
w_time = finish - start
#endif
cputime = (cpu2 - cpu1)
usertime = (user2 - user1)
systime = (sys2 - sys1)

e_mflops = 4.e0*FLOAT(m)*FLOAT(n)/(cputime*1.e6)
w_mflops = 4.e0*FLOAT(m)*FLOAT(n)/(w_time*1.e6)

call MPI_REDUCE(cputime, E_Max_Time, 1, MPI_REAL,
                 $  MPI_MAX, 0, MPI_COMM_WORLD, ierr)
call MPI_REDUCE(cputime, E_Avg_Time, 1, MPI_REAL,
                 $  MPI_SUM, 0, MPI_COMM_WORLD, ierr)
call MPI_REDUCE(e_mflops, E_Min_Mflop, 1, MPI_REAL,
                 $  MPI_MIN, 0, MPI_COMM_WORLD, ierr)
call MPI_REDUCE(e_mflops, E_Avg_Mflop, 1, MPI_REAL,
                 $  MPI_SUM, 0, MPI_COMM_WORLD, ierr)

call MPI_REDUCE(w_time, W_Max_Time, 1, MPI_REAL,
                 $  MPI_MAX, 0, MPI_COMM_WORLD, ierr)
call MPI_REDUCE(w_time, W_Avg_Time, 1, MPI_REAL,
                 $  MPI_SUM, 0, MPI_COMM_WORLD, ierr)
call MPI_REDUCE(w_mflops, W_Min_Mflop, 1, MPI_REAL,
                 $  MPI_MIN, 0, MPI_COMM_WORLD, ierr)
call MPI_REDUCE(w_mflops, W_Avg_Mflop, 1, MPI_REAL,
                 $  MPI_SUM, 0, MPI_COMM_WORLD, ierr)

print*, '----------------------------------------------'
print*, 'IAM = ', IAM , ' cputime = ', cputime
print*, 'IAM = ', IAM , ' usertime = ', usertime
print*, 'IAM = ', IAM , ' systime = ', systime
print*, 'IAM = ', IAM , ' w_time= ', w_time

if (IAM.eq.0) then
  print*, '----------------------------------------------'
  print*, 'E_TIME: '
  print*, 'Average E_Time = ', E_Avg_Time/NPROCS
  print*, 'Max E_Time = ', E_Max_Time
end if
if (IAM.eq.0) then
  print*, 'WTIME: '
  print*, 'Average W_Time = ', W_Avg_Time/NPROCS
  print*, 'Max W_Time = ', W_Max_Time
end if

#endif

10 CONTINUE
20 CONTINUE

CLOSE ( NOUT )

* BLACS Grid Exit
CALL BLACS_GRIDEXIT( ICTXT )
CALL BLACS_EXIT( 0 )

* 9999 FORMAT( 'Bad ', A6, ' parameters: going on to next test case.' )
9998 FORMAT( 'Unable to perform ', A, ': need TOTMEM of at least', $ I11 )

STOP

* End of PGEMVTDRIVER

* END
Appendix B

B.1 BLACS Description

The following information is taken from Dongarra and Whaley [10] and information from the source code for the BLACS.

**BLACS_PININFO (MYPNUM, NPROCS)**

MYPNUM (output) INTEGER
An integer between 0 and (NPROCS - 1) which uniquely identifies each process.

NPROCS (output) INTEGER
The number of processes available for BLACS use.

This routine is used when some initial system information is required before the BLACS are set up. NPROCS is the actual number of processes available for use (i.e. NPROWS × NPCOLS ≤ NPROCS).

**BLACS_GRIDINIT (ICONTXT, ORDER, NPROW, NPCOL)**

This routine creates a simple NPROW × NPCOL process grid. This process grid will use the first NPROW * NPCOL processes, and assign them to the grid in a row or column-major natural ordering. If these process-to-grid mappings are unacceptable, BLACS_GRIDINIT’s more complex companion routine BLACS_GRIDMAP must be called instead.

ICONTXT (input/output) INTEGER
On input, an integer handle indicating the system context to be used in creating the BLACS context. The user may obtain a default system context via a call to BLACS GET.
On output, the integer handle to the created BLACS context.
ORDER (input) CHARACTER*1
Indicates how to map processes to BLACS grid. Choices are:
‘R’ : Use row-major natural ordering.
‘C’ : Use column-major natural ordering.
ELSE : Use row-major natural ordering.

NPROW (input) INTEGER
Indicates how many process rows the process grid should contain.

NPCOL (input) INTEGER
Indicates how many process columns the process grid should contain.

All BLACS codes must call this routine, or its companion routine BLACS_GRIDMAP. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system will map into the native machine’s process numbering system. Each BLACS grid is contained in a context (its own message passing universe), so that it does not interfere with distributed operations which occur within other grids/contexts. These grid creation routines may be called repeatedly in order to define additional contexts/grids.

The creation of a grid requires input from all processes which are defined to be in it. It is therefore a globally-blocking (sometimes called synchronous) operation which means that processes belonging to more than one grid will have to agree on which grid formation will be serviced first.

These grid creation routines set up various internals for the BLACS, and so one of them must be called before any calls are made to the non-initialization BLACS.

BLACS_GRIDINFO(ICONTXT, NPROW, NPCOL MYPROW, MYPCOL)

This routine returns information about the process grid. It returns total number of process rows, total number of process columns, process row of the calling process and process column of the calling process.

ICONTXT (input) INTEGER
Integer handle indicating the BLACS context to be queried.
NPROW (output) INTEGER
On output, the number of process rows in ICONTXT’s process grid.

NPCOL (output) INTEGER
On output, the number of process columns in ICONTXT’s process grid.

MYPROW (output) INTEGER
On output, the calling process’s row coordinate in the process grid.

MYP_COL (output) INTEGER
On output, the calling process’s column coordinate in the process grid.

Returns information about the process grid contained in the context whose handle is ICONTXT. If the context handle is invalid, all quantities are returned as -1.

**BLACS_GET(ICONTXT, WHAT, VAL)**

This routine returns the values the BLACS are using for internal defaults. Some values are tied to a BLACS context, and some are more general. The most common use is in retrieving a default system context for input into BLACS_GRIDINIT or BLACS_GRIDMAP. Some systems, such as MPI, supply their own version of context (in MPI, this corresponds to a communicator). For those users who mix system code with BLACS code, we therefore need to be able to form a BLACS context in reference to a system context. Thus, the grid creation routines take a system context as input. If you wish to have strictly portable code, you may use BLACS_GET to retrieve a default system context which will include all available processes.

Also not tied to a particular BLACS context are the message ID range and the debug level the BLACS were compiled with. For these three values of WHAT, the parameter ICONTXT is not referenced.

The other choices of WHAT are all tied to a particular BLACS context, so the parameter ICONTXT must be a valid BLACS context handle.

**ICONTXT** (input) INTEGER
On WHATs that are tied to a particular context, this is the integer handle indicating the BLACS context to query. Otherwise, it is ignored.
**WHAT** (input) INTEGER
What BLACS internal information should be returned in VAL.
Present options are:

<table>
<thead>
<tr>
<th>WHAT</th>
<th>Returned in VAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Handle indicating default system context, calls Cssys2blacs_handle(MPI_COMM_WORLD) for C-Interface and returns F77_MPI_COMM_WORLD for F77 interface.</td>
</tr>
<tr>
<td>1</td>
<td>The BLACS message ID range, calls MPI_Attr_get().</td>
</tr>
<tr>
<td>2</td>
<td>The BLACS debug level, returns BlacsDebugLvl.</td>
</tr>
</tbody>
</table>
| 10   | Handle indicating the system context used to define the BLACS context whose handle is ICONTEXT, i) Calls MGetConTxt() which returns ctxt
|     | ii) If C-Interface
|     | a) BL_MPI_F77_to_c_trans_comm();
|     | b) returns VAL = Cssys2blacs_handle();
|     | If F77-Interface
|     | a) BL_MPI_C_to_f77_trans_comm();
|     | b) returns VAL = *ctxt->F77_comm; |
| 11   | Number of rings multiring broadcast topology is presently using, calls MGetConTxt() |
| 12   | Number of branches general tree broadcast topology is presently using, calls MGetConTxt() |
| 13   | Number of rings multiring combine topology is presently using, calls MGetConTxt() |
| 14   | Number of branches general tree combine topology is presently using, calls MGetConTxt() |
| 15   | If topologies are being forced to be repeatable, a non-zero is returned. If repeatability is not being enforced, zero is returned. Calls MGetConTxt() |
| 16   | If topologies are being forced to be heterogeneous coherent, a non-zero is returned. If heterogeneous coherence is not being enforced, zero is returned. Calls MGetConTxt() |

**VAL** (output) INTEGER ARRAY of variable dimension
The value to which the BLACS internal is presently set.
The dimension of VAL is (2) if the message ID range is being returned. For all other queries it is (1).
BLACS_EXIT(CONTINUE )

CONTINUE  (input) INTEGER

If CONTINUE is non-zero, it is assumed that the user will continue using the machine after the BLACS are done. Otherwise, it is assumed that no message passing will be done after the BLACS_EXIT call.

This routine should be called when a process has finished all use of the BLACS. It frees all BLACS contexts and releases all memory the BLACS have allocated. CONTINUE indicates whether the user will be using the underlying communication platform after the BLACS are finished. This information is most important for the PVM BLACS. If CONTINUE is set to 0, then pvm_exit will be called; otherwise, it will not. If the user sets CONTINUE not equal to 0, he is indicating that he will be calling explicit PVM send/recvs after the BLACS are done, so that the process cannot tell the virtual machine that it is done. It then becomes the user’s responsibility to make sure his code calls pvm_exit. PVM users should either call BLACS_EXIT or explicitly call pvm_exit to avoid PVM problems.
Bibliography


