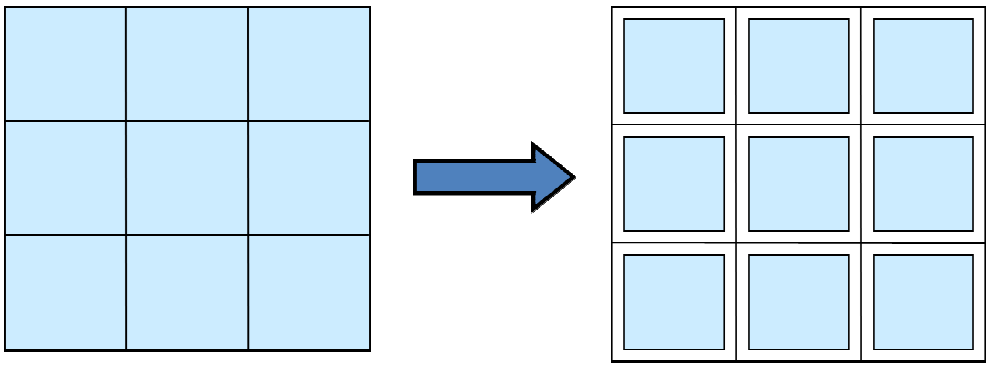
Report of Home Work 4:

first part, Ghost Region:



The Matrix is divided into several blocks in rows and columns. The dimension of the matrix is n and matrix is squared, the number of processors in row is named Pr and the number of processors in column is Pc. Then, the number of local columns and rows in each block could be specified. The file *myfile.dat* contains the matrix and *data.dat* contains the dimension, Pr, Pc, number of Local Row and number of Local Column, respectively. (The receptiveness of inserting data is also mentioned in the file)

One thing to be mentioned is the method I used to allocate memory for matrix A. As in MPI, the memory should be allocated in a contiguous way; I used a function to construct a contiguous piece of memory for 2 dimensional matrices:

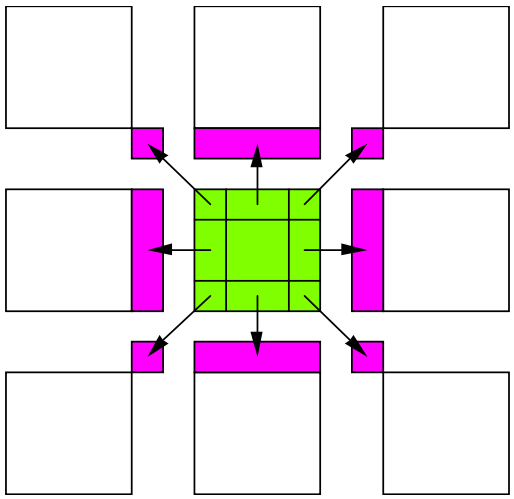
```
int MemAlloc2d(float ***array, int n, int m) {

    /*the goal is to  allocate the n*m contiguous items */
        float *p = (float *)malloc(n*m*sizeof(float));
        if (!p) return -1;

      /* allocate the row pointers into the memory */
        (*array) = (float **)malloc(n*sizeof(float*));
        if (!(*array)) {
           free(p);
           return -1;
                 }

        /* set up the pointers into the contiguous memory */
        int i;
         for (i=0; i<n; i++)
            (*array)[i] = &(p[i*m]);

        return 0;
}
```

Once the matrix is read by processor rank 0, it uses a data type to send each processor's portion of the matrix to them. I used MPI_Type_create_subarray() to construct the appropriate block for each one

the processors based on their rank. The key thing here is the starting point of Global Matrix at which we want to extract our (nLocRow)**x**(nLocCol) block. This is specified by *StartsAt* in the following piece of code.

```
/* Create a datatype to describe the subarrays of the global array */

   int Csize[2]    = {n, n};         /* global size */
   int blocksize[2] = {nLocRow, nLocCol};     /* local size */
   MPI_Datatype type;
   int startsAt[2]= {(myrank/Pc)*nLocRow,(myrank%Pc)*nLocCol};
   MPI_Type_create_subarray(2, Csize, blocksize, startsAt, MPI_ORDER_C,
       MPI_FLOAT, &type);
   MPI_Type_commit(&type);

   /*Setting the block for zero processor*/
   /*Sending the extracted block to other processors*/
   if (myrank==0) {
        for (i=0;i<nLocRow;i++){
            for (j=0;j<nLocCol;j++){
                blk[i][j]  =  C[i+(myrank/Pc)*nLocRow][j+(myrank%Pc)*nLocCol];
                }
            }
   for (i=1;i<Pr*Pc;i++)
        MPI_Send(&(C[(i/Pc)*nLocRow][(i%Pc)*nLocCol]), 1, type, i, 100,
       MPI_COMM_WORLD);

   }


   /*Recieving the corresponding blocks by all processors from root processor(0)*/
   if (myrank!=0){
        MPI_Recv(&(blk[0][0]), nLocRow*nLocCol, MPI_FLOAT , 0, 100,
       MPI_COMM_WORLD , &status);
   }
```

As they have received their blocks (named by &(blk[0][0])), they will decide based on their rank to extract the appropriate ghost regions for sending to neighbors. It could generally be 8 different types of Ghost Regions including 4 for corners. The problem is asking just for the 4 at each side.



In the code, the regions that are received by each block from their neighbors are named as their Ghost Regions. Specifically, Each one of the blocks (or processors) has LeG, RiG, UpG and BoG which are the abbreviations of Left Ghost Region, Right Ghost Region, Upper Ghost Region and Bottom Ghost Region, respectively.

Obviously, not all of the blocks have all the ghost regions. So, there are 4 conditions that are expressed by if statement to make sure that each processor is sending its meaningful ghost region and on the other side, is receiving appropriate one from neighbor (line 172-224).

I used MPI_Type_vector to build the proper datatypes. As, each one the processors should both send and receive at the same time, blocking send and receive would be trapped in deadlock. So, I used nonblocking send and blocking receive:

```
 /*to bottom side neighbor*/
   if ((myrank+Pc)<(Pr*Pc)){/*so the processor has bottom neighbor*/
       MPI_Type_vector(nLocCol,1,1,MPI_FLOAT,&BoGtype);
       MPI_Type_commit(&BoGtype);
       MPI_Isend(&(blk[nLocRow-1][0]), 1, BoGtype, myrank+Pc, 104,
              MPI_COMM_WORLD, &reqBo);
       MPI_Type_free(&BoGtype);
           }
/*from bottom neighbor*/
     if ((myrank+Pc)<(Pr*Pc)){/*so the processor has bottom neighbor*/
         MPI_Recv(&(BoG[0][0]), nLocCol, MPI_FLOAT, myrank+Pc, 103,
             MPI_COMM_WORLD,&statusBoG);
                      }
```

Here is an example:

A=

1.000000 2.000000 3.000000 4.000000 5.000000 6.000000

7.000000 3.000000 5.000000 8.000000 3.000000 2.000000

9.000000 1.000000 3.000000 5.000000 7.000000 2.000000

5.000000 3.000000 8.000000 2.000000 7.000000 9.000000

0.000000 9.000000 1.000000 6.000000 9.000000 3.000000

0.000000 9.000000 0.000000 1.000000 5.000000 8.000000

I set the input of *data.dat* as: 6 3 2 2 3 ==> for n,Pr,Pc,nLocRow,nLocCol

mpirun.openmpi -np 6 ./a.out

Matrix has been successfully read by process 0

Process 0's block

1.000000 2.000000 3.000000

7.000000 3.000000 5.000000

Process 1's block

4.000000 5.000000 6.000000

8.000000 3.000000 2.000000

Process 2's block

9.000000 1.000000 3.000000

5.000000 3.000000 8.000000

Process 3's block

5.000000 7.000000 2.000000

2.000000 7.000000 9.000000

Process 4's block

0.000000 9.000000 1.000000

0.000000 9.000000 0.000000

Process 5's block

6.000000 9.000000 3.000000

1.000000 5.000000 8.000000

I am block number 0 my Ghost Regions are:

RIGHT GHOST REGION(block.0):

4.000000

8.000000

BOTTOM GHOST REGION(block.0):

9.000000 1.000000 3.000000

I am block number 1 my Ghost Regions are:

LEFT GHOST REGION(block.1):

3.000000

5.000000

BOTTOM GHOST REGION(block.1):

5.000000 7.000000 2.000000

I am block number 2 my Ghost Regions are:

RIGHT GHOST REGION(block.2):

5.000000

2.000000

UPPER GHOST REGION(block.2):

7.000000 3.000000 5.000000

BOTTOM GHOST REGION(block.2):

0.000000 9.000000 1.000000

I am block number 3 my Ghost Regions are:

LEFT GHOST REGION(block.3):

3.000000

8.000000

UPPER GHOST REGION(block.3):

8.000000 3.000000 2.000000

BOTTOM GHOST REGION(block.3):

6.000000 9.000000 3.000000

I am block number 4 my Ghost Regions are:

RIGHT GHOST REGION(block.4):

6.000000

1.000000

UPPER GHOST REGION(block.4):

5.000000 3.000000 8.000000

I am block number 5 my Ghost Regions are:

LEFT GHOST REGION(block.5):

1.000000

0.000000

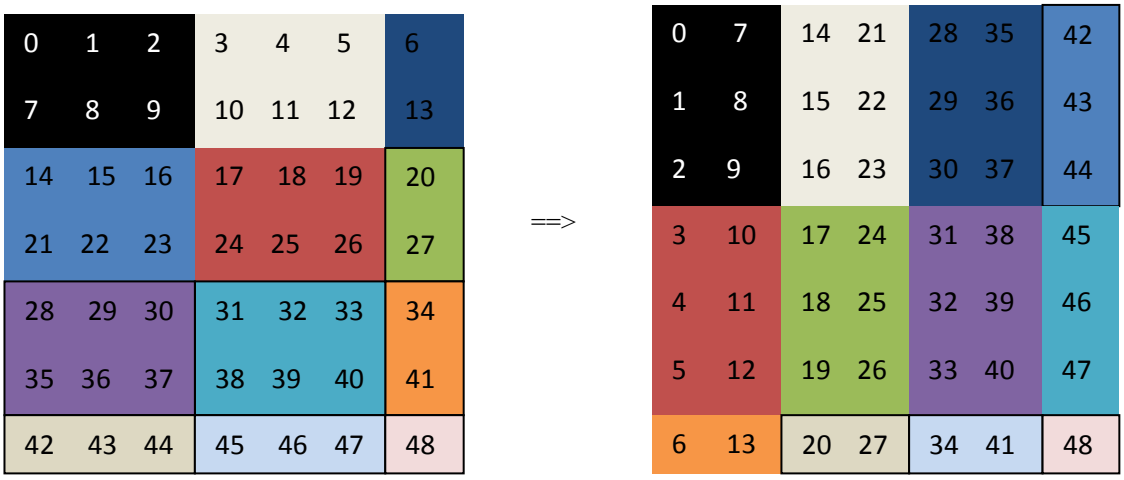
UPPER GHOST REGION(block.5):

2.000000 7.000000 9.000000

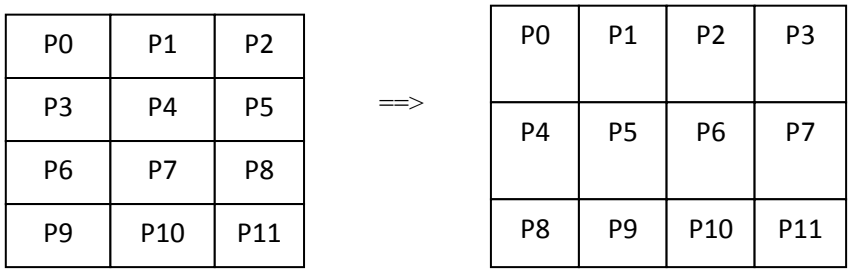
So, it seems that it gives the right answer. Here I assumed that the matrix is evenly divisible. For the next section, I extended the code to capture the cases in which it could be nondivisible.

Second part, Transpose of a matrix:

In this part, we do the same thing for making the blocks. The way I thought about it was that each processor has a location specified by (i,j). This (i,j) is equivalent to ((myrank/Pc),(myrank%Pc)). This processor has to send the transposed block to one the processor which is in a certain place and receive from another one which is in another location. Here is an example:



In this example, the matrix is a 7x7 matrix (a prime number). Then, the original dimension of the blocks is then 2x3 which is only disobeyed in right and bottom borders. The color here shows the owner of the block on the left hand side and the owner of transpose on the right hand side. If I show them like:



Then, it is apparent that P3 should send its transpose to P1 and receive the transpose of P9 and so on. Therefore, the formulation for source and destination would be:

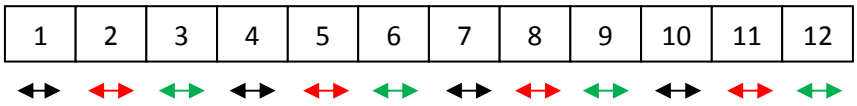
```
int dest = ((myrank%Pc))*Pr+(myrank/Pc);
int source = ((myrank%Pr))*Pc+(myrank/Pr);
```

As datatype, I used MPI_Type_vector() followed by MPI_Type_hvector() as follows:

```
MPI_Type_vector(nLocRow, 1, nLocCol, MPI_FLOAT, &col);
MPI_Type_hvector(nLocCol, 1,sizeof(float) , col, &transpose);
MPI_Type_commit(&transpose);

MPI_Sendrecv (&blk[0][0],1,transpose,dest,102,&blk_tr[0][0],
    nLocRow*nLocCol,MPI_FLOAT,source,102,MPI_COMM_WORLD,&tr_status);
```

This could be explained as follows:

1	2	3
4	5	6
7	8	9
10	11	12



In the bottom unrolled representation, each color belongs to one vector datatype. Then, 1 4 7 10 would be one datatype which has a distance of `sizeof(float)` with the neighbor datatype which is 2 5 8 11. Therefore, we will construct the transposed block from each block.

I also have used MPI_Sendrecv() for communication of processors in order to prevent getting stocked in deadlock.

For next step, one may gather the data in root processor (zero). Then first, I have to specify the displacement of each one of the processors in the final transposed matrix. For that purpose, I initially changed the units of displacement to be just 1 for one row of each block in transposed matrix using MPI_Type_create_resized():

```
int sizes[2]    = {Pc*nLocCol, Pr*nLocRow};       /* global size */
int subsizes[2] = {nLocCol,nLocRow};            /* local size */
int starts[2]   = {0,0};                      /* where this one starts */
MPI_Datatype typep, subarrtype;
MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                         MPI_FLOAT, &typep);
MPI_Type_create_resized(typep, 0, nLocRow*sizeof(float), &subarrtype);
MPI_Type_commit(&subarrtype);
```

Then, I wrote an algorithm to find the displacement (offset) for each one of the blocks and finally I used MPI_Gatherv() for gathering the transposed blocks of processors.

```
if (myrank == 0) {
       for (i=0; i<Pr*Pc; i++) sendcounts[i] = 1;
       int disp = 0;
       for (i=0; i<Pc; i++) {
           for (j=0; j<Pr; j++) {
               displs[i*Pr+j] = disp;
               disp += 1;
                              }
           disp += ((nLocCol)-1)*Pr;
                  }
            }


   MPI_Gatherv(&(blk_tr[0][0]), nLocRow*nLocCol,MPI_FLOAT,
              &(C_tr[0][0]), sendcounts, displs, subarrtype, 0, MPI_COMM_WORLD);
```

Here is an example:

C=

```
     0.399020  0.893633  0.877799  0.984776  0.749018  0.788113  0.939410
     0.047401  0.054792  0.582433  0.715678  0.503888  0.780296  0.980904
     0.342374  0.303661  0.070684  0.838970  0.646810  0.668512  0.286620
     0.735966  0.046192  0.922745  0.433261  0.307746  0.133504  0.800820
     0.794682  0.195477  0.800372  0.470625  0.138725  0.021556  0.896111
     0.544906  0.720166  0.285947  0.560713  0.475573  0.559841  0.597527
     0.686223  0.721753  0.543663  0.269092  0.362459  0.300819  0.884017
```

in *myfile.dat* 7 4 3 2 3 as Pr, Pc, nLocRow, nLocCol.

```
Matrix has been successfully read by process 0
Process 0's block
0.399020        0.893633        0.877799
0.047401        0.054792        0.582433
Process 1's block
0.984776        0.749018        0.788113
0.715678        0.503888        0.780296
Process 2's block
0.939410
0.980904
Process 3's block
0.342374        0.303661        0.070684
0.735966        0.046192        0.922745
Process 4's block
0.838970        0.646810        0.668512
0.433261        0.307746        0.133504
Process 5's block
0.286620
0.800820
```

```
Process 6's block
0.794682        0.195477        0.800372
0.544906        0.720166        0.285947

Process 7's block
0.470625        0.138725        0.021556
0.560713        0.475573        0.559841
Process 8's block
0.896111
0.597527
Process 9's block
0.686223        0.721753        0.543663
Process 10's block
0.269092        0.362459        0.300819
Process 11's block
0.884017


Process 0's transposed block
0.399020        0.047401
0.893633        0.054792
0.877799        0.582433
Process 1's transposed block
0.342374        0.735966
0.303661        0.046192
0.070684        0.922745
Process 2's transposed block
0.794682        0.544906
0.195477        0.720166
0.800372        0.285947
Process 3's transposed block
0.686223
0.721753
0.543663
Process 4's transposed block
0.984776        0.715678
0.749018        0.503888
0.788113        0.780296
Process 5's transposed block
0.838970        0.433261
0.646810        0.307746
0.668512        0.133504
Process 6's transposed block
0.470625        0.560713
0.138725        0.475573
0.021556        0.559841
Process 7's transposed block
0.269092
0.362459
0.300819
Process 8's transposed block
0.939410        0.980904
Process 9's transposed block
0.286620        0.800820
Process 10's transposed block
0.896111        0.597527
Process 11's transposed block
```

0.884017

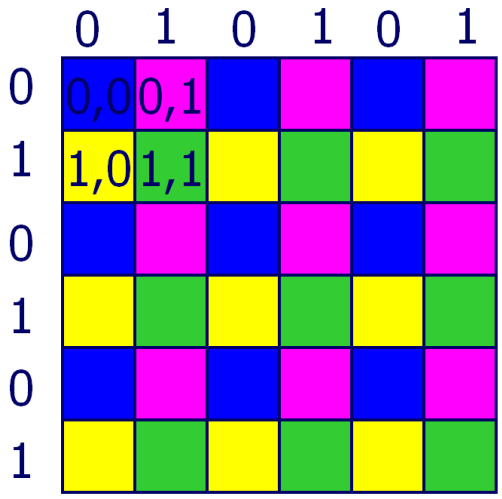
Transposed Matrix:

0.399020	0.047401	0.342374	0.735966	0.794682	0.544906	0.686223
0.893633	0.054792	0.303661	0.046192	0.195477	0.720166	0.721753
0.877799	0.582433	0.070684	0.922745	0.800372	0.285947	0.543663
0.984776	0.715678	0.838970	0.433261	0.470625	0.560713	0.269092
0.749018	0.503888	0.646810	0.307746	0.138725	0.475573	0.362459
0.788113	0.780296	0.668512	0.133504	0.021556	0.559841	0.300819
0.939410	0.980904	0.286620	0.800820	0.896111	0.597527	0.884017

Third part, pdgemm:

I think about this problem in three steps:

1. Is to generate the distributed array. (As scaLAPACK format)

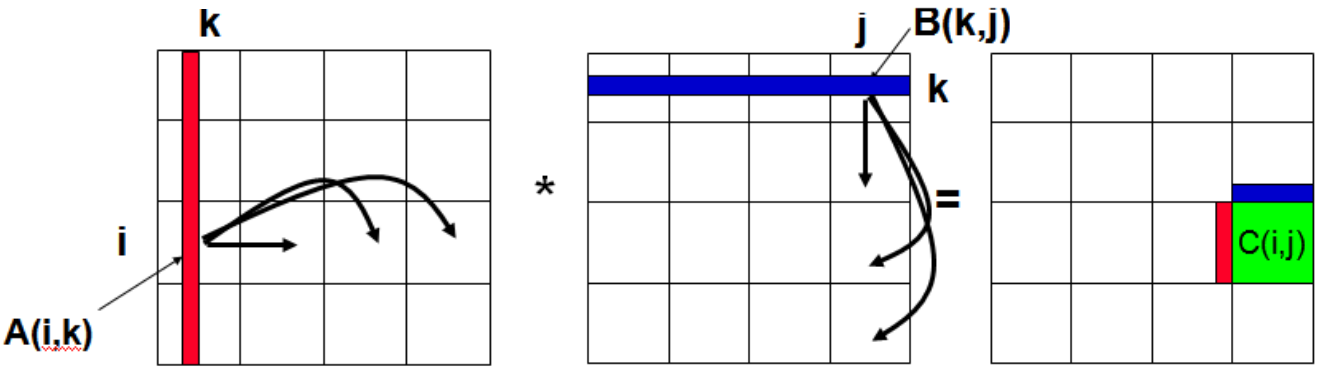


Again one can use vector followed by hvector to construct the blocks:

```
MPI_Datatype type,type2;
MPI_Status status;

MPI_Type_vector(kk*n/(Pc*kk), kk, kk*Pc, MPI_FLOAT, &type);
MPI_Type_hvector(n/(Pr*kk), 1,kk*Pr*n*sizeof(float) , type, &type2);
MPI_Type_commit(&type2);
```

where kk is the size of one block in each processor (cyclic(kk)/cyclic(kk)).

2. Then, we need to know the algorithm of SUMMA or PUMMA. Searching through the web, I found this useful slide for explaining the algorithm:



- i, j represent all rows, columns owned by a processor
- k is a block of kk ≥ 1 rows or columns
- $C(i,j) = C(i,j) + \sum A(i,k) * B(k,j)$
- Assume a P_r by P_c processor grid ($P_r = P_c = 4$ above)
    - Need not be square

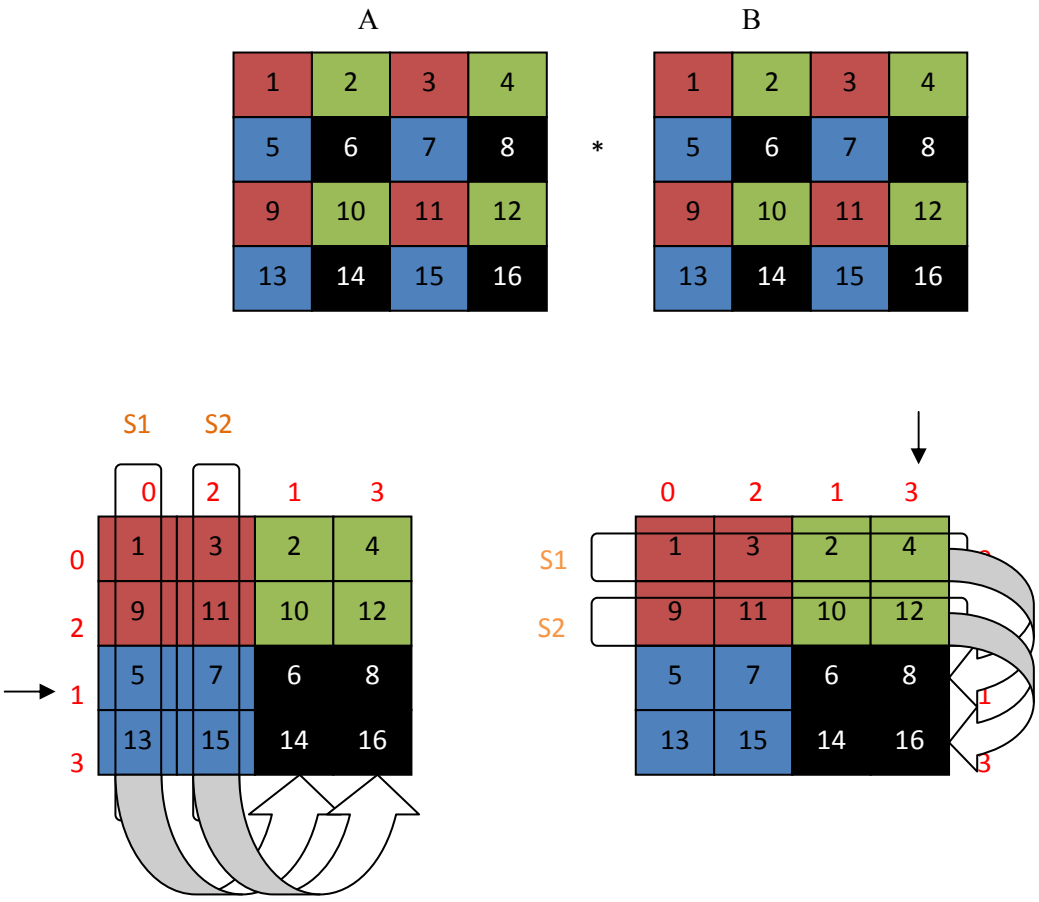
Therefore:

```
for k=0 to n/k-1 {
     for all i = 1 to pr   … in parallel{
           owner of A(i,k) broadcasts it to whole processor row
     }
     for all j = 1 to pc  … in parallel{
            owner of B(k,j) broadcasts it to whole processor column
     }

     Receive A(i,k) into Acol
     Receive B(k,j) into Brow
     C_myproc  = C_myproc  + Acol * Brow

}
```

2.1. In this sub stage, we specify the appropriate communicators for sending the blocks. Therefore for the example above, we need four communicator for A (i.e. 0 1 2 3 in commrow1 and 4 5 6 7 in commrow2 and ...) and four communicator for B (i.e. 0 4 8 12 in commcol1 and 1 5 9 13 in commcol2).

However, as it was mentioned in class, they could be obtained by just one call for each one of them. So two calls for constructing 8 different communicators (using module and /).

Then, Let's say we are solving this example:



And we want to obtain C(1,3)

This is actually located at processor black. So, the required components of the matrix should be sent to this processor. We need 5 and 7 in A and processor blue to be sent to black. At the same time 4 and 12 in B would be sent from processor green to black. So for C(1,3), we had already 6 8 in row (1) of processor black and 8 16 in column (3) of that processor. Multiplying the row and column would result in 6*8+8*16 which should be added to multiplication of the received row from left 5 7 and column 4 12; 5*4+7*12. Then we would get the C(1,3) as: C(1,3) = 5*4+7*12+6*8+8*16

And the nice thing is the fact that by this method of broadcasting, now we are able to calculate all the other components of C in this processor. like C(3,1)

C(3,1)=14*6+16*14 (which were already in that processor) + 13*2+15*10

If we had more processors in column or row, then we need to broadcast their contribution in the same way I explained. which results in the Ring Broadcasting method.