```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

typedef struct
{
  int      p;
  MPI_Comm comm;
  MPI_Comm row_comm;
  MPI_Comm col_comm;
  int      q;
  int      my_row;
  int      my_col;
  int      my_rank;
}
GRID_INFO_T;

main(int argc, char* argv[])
{
  int         rank;
  int         p;
  float       *block_A;
  float       *block_B;
  float       *mat_C;
  int         ma, na;
  int         mb, nb;
  int         i, j;
  GRID_INFO_T grid;
  MPI_Status  status;
  FILE         *fp;
  double      start1, start2, start3, start4;
  double      finish1, finish2, finish3, finish4;



  void Setup_grid(GRID_INFO_T*  grid);
  void Read_matrix(char* prompt, float block[], int m, int n, GRID_INFO_T* grid);
  float *parallel_Fox(float block_A[], float block_B[], int ma, int nb, int na,
int mb, GRID_INFO_T* grid);

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  Setup_grid(&grid);

  /* read dimension of matrix A*/
  if (rank == 0)
  {
  //printf("Enter the dimension m, n of the m x n matrix A:\n");
    scanf("%d %d", &ma, &na);
  }

  /*broadcast the demion of matrix A*/
  MPI_Bcast(&ma, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&na, 1, MPI_INT, 0, MPI_COMM_WORLD);

  block_A = (float*)calloc((na * ma / grid.p), sizeof(float));



  /* read matrix A*/
  start1 = MPI_Wtime();
  Read_matrix("The matrix A", block_A, ma, na, &grid);
  finish1 = MPI_Wtime();
```



```c
  /* read dimension of matrix B*/
  if (grid.my_rank == 0)
  {
    //printf("Enter the dimension m, n of the m x n matrix B:
\n");
    scanf("%d %d", &mb, &nb);
  }

  /*broadcast the demion of matrix B*/
  MPI_Bcast(&mb, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&nb, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if (na != mb)
  {
    printf("Matrices dimension do not match!\n");
    exit;
  }

  block_B = (float*)calloc(nb * mb / grid.p, sizeof(float));
  mat_C = (float*)malloc(ma * nb * sizeof(float));

  /* read matrix B*/
  start2 = MPI_Wtime();
  Read_matrix("The matrix B", block_B, mb, nb, &grid);
  finish2 = MPI_Wtime();

  /* FOX algorithem*/
  start3 = MPI_Wtime();
  mat_C = parallel_Fox(block_A, block_B, ma, nb, na, mb, &grid);
  finish3 = MPI_Wtime();

  /* save result in a file*/
  if (grid.my_rank == 0)
  {
    fp = fopen("C.txt", "a"); //create C.txt

    start4 = MPI_Wtime();
    fprintf(fp, "%d\n", ma);
    fprintf(fp, "%d\n", nb);

    for (i = 0; i < ma * nb; i++)
    {
      fprintf(fp, "%f\n", mat_C[i]);
    }
    finish4 = MPI_Wtime();

    fclose(fp);

    printf("Number of processes:      %d\n", grid.p);
    printf("Time elapsed with I/O:    %e\n", finish1 + finish2 + finish3 +
finish4 - start1 - start2 - start3 - start4);
    printf("Time elapsed without I/O: %e\n", finish3 - start3);
  }

  free(block_A);
  free(block_B);
  free(mat_C);

  MPI_Finalize();
}

/* read matrix*/
void Read_matrix(char* prompt, float block[], int m, int n, GRID_INFO_T* grid)
{
  float *temp;
  int m_bar;
```

```
  int n_bar;
  int i, j, k, h, x;
  int l;
  int q, dest;
  MPI_Status status;

  m_bar = m / grid->q;
  n_bar = n / grid->q;

  temp = (float*)malloc(m * n / grid->q * sizeof(float));

  /* this code use MPI_Scatter is shorter, but it needs to allocate entire
matrix,  which is memory cost */
  /* if (my_rank ==
0)

{
      for (i =0; i < q; i++) // row of processes
        for (j = 0; j < m; j++) // number of reading when read the part of matrix
for each row of porc.
          for(k = 0; k < n_bar; k++)
          {
            scanf("%f", &temp[m * n / p * (j % q) + j / q * n_bar + k + i * m_bar
* n_bar * q]); // rearrange the index
            // the size of temp equals to the size of the entire matrix
          }
    }
  MPI_Scatter(temp, m * n / p, MPI_FLOAT, block, m * n / p, MPI_FLOAT, 0,
MPI_COMM_WORLD);
  }*/


  /* this code reads 1/q of entries each time then send them to the corresponding
row of processes*/
  /* this code is longer, but needs less memory*/
  if (grid->my_rank == 0)
  {
    for (i =0; i < grid->q; i++) // row of processes
    {
      for (j = 0; j < m; j++) // number of reading when read the part of matrix
for each row of porc.
      {
        for(k = 0; k < n_bar; k++)
        {
          scanf("%f", &temp[m_bar * n_bar * (j % grid->q) + j / grid->q * n_bar +
k ]); // rearrange the index
        }
      }

      if (i == 0)
      {
        memcpy(block, temp, m_bar * n_bar * sizeof(float)); // keep for proc.0
itself

        for (l = 1; l < grid->q; l++) // send the rest to the rest 1st row
processes
        {
          dest = l;
          MPI_Send(temp + l * n_bar * m_bar, n_bar * m_bar, MPI_FLOAT, dest, 0 ,
grid->comm);
        }
      }

      else
      {
        for (l = 0; l < grid->q; l++) // send to other processes row wise
```



```
        {
          dest = l + i * grid->q;
          MPI_Send(temp + l * n_bar * m_bar, n_bar * m_bar, MPI_FLOAT, dest, 0 ,
grid->comm);
        }
      }
    }
  }

  else
  {
    MPI_Recv(block, n_bar * m_bar, MPI_FLOAT, 0, 0, grid->comm, &status);
  }

  free(temp);
}

/*local dot operation in the block*/
float *local_dot(float block_A[], float block_B[], int ma, int nb, int na,
GRID_INFO_T* grid)
{
  int i, j, k, h;
  int ma_bar, nb_bar, na_bar;
  float *res;

  res = (float*)calloc(ma * nb / grid->p, sizeof(float));
  ma_bar = ma / grid->q;
  nb_bar = nb / grid->q;
  na_bar = na / grid->q;

  for (j = 0; j < ma_bar; j++) // number of rows in block A
  {
    for (k = 0; k < nb_bar; k++) // number of columns in block B
    {
      for (h = 0; h < na_bar; h++)
      {
        res[j * nb_bar + k] = res[j * nb_bar + k] + block_A[h + j * na_bar] *
block_B[k + h * nb_bar];
      }
    }
  }

  return res;
}

/* circular shift in column communicators */
float column_circular_shift(float block_B[], int mb, int nb, GRID_INFO_T* grid)
{
  int dest;
  int source;
  int tag = 0;
  MPI_Status status;

  source = (grid->q + grid->my_row + 1) % grid->q;
  dest = (grid->q + grid->my_row - 1) % grid->q;

  MPI_Sendrecv_replace(block_B, mb * nb / grid->p, MPI_FLOAT, dest, tag, source,
tag, grid->col_comm, &status);
}

/* broadcast in row communicator */
float row_broadcast(float block_A[], float new_block[], int na, int ma, int step,
GRID_INFO_T* grid)
{
  int root;
  int  count;
```

```
  count = ma * na / grid->p;

  // determine which block should be broadcasted in step(0....grid.q-1) in each
row
  if (grid->my_rank == grid->my_row * grid->q + (grid->my_row + step) % grid->q)
  {
    memcpy(new_block, block_A, count * sizeof(float));
  }

  root = (grid->my_row + step % grid->q) % grid->q;
  MPI_Bcast(new_block, count, MPI_FLOAT, root, grid->row_comm);
}

/* parallel Fox algorithm */

float *parallel_Fox(float block_A[],
                    float block_B[],
                    int ma,
                    int nb,
                    int na,
                    int mb,
                    GRID_INFO_T* grid)
{
  int i, j, k,h,max_step;
  float *res;
  float *new_block;
  float *pieces_res;
  float *mat_C;

  float *local_dot(float block_A[], float block_B[], int ma, int nb, int na,
GRID_INFO_T* grid);
  float column_circular_shift(float block_B[], int mb, int nb, GRID_INFO_T* grid);
  float row_broadcast(float block_A[], float new_block[], int na, int ma, int
step, GRID_INFO_T* grid);
  void rearrange_result(float *block_C, float *res, int ma, int nb, GRID_INFO_T*
grid);

  max_step = grid->q;
  res = (float*)calloc(ma * nb / grid->p, sizeof(float));
  pieces_res = (float*)malloc(ma * nb / grid->p * sizeof(float));
  new_block = (float*)calloc(na * ma / grid->p, sizeof(float));
  mat_C = (float*)malloc(ma * nb * sizeof(float));

  for (i = 0; i < max_step; i++)
  {
    row_broadcast(block_A, new_block, na, ma, i, grid);
    pieces_res = local_dot(new_block, block_B, ma, nb, na, grid);
    column_circular_shift(block_B, mb, nb, grid);

    for (j = 0; j < ma * nb / grid->p; j++)
    {
      res[j] = res[j] + pieces_res[j]; // update local result
    }
  }

  rearrange_result(res, mat_C, ma, nb, grid); // arrange entries to a nomal order

  free(res);
  free(pieces_res);
  free(new_block);

  return mat_C;
}

void Setup_grid( GRID_INFO_T*  grid  /* out */)
{
```



```
  int old_rank;
  int dimensions[2];
  int wrap_around[2];
  int coordinates[2];
  int free_coords[2];

  /* Set up Global Grid Information */
  MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
  MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

  /* We assume p is a perfect square */
  grid->q = (int) sqrt((double) grid->p);
  dimensions[0] = dimensions[1] = grid->q;

  /* We want a circular shift in second dimension.  */
  /* Don't care about first                         */
  wrap_around[0] = wrap_around[1] = 1;
  MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
                  wrap_around, 1, &(grid->comm));
  MPI_Comm_rank(grid->comm, &(grid->my_rank));
  MPI_Cart_coords(grid->comm, grid->my_rank, 2,
                  coordinates);
  grid->my_row = coordinates[0];
  grid->my_col = coordinates[1];

  /* Set up row communicators */
  free_coords[0] = 0;
  free_coords[1] = 1;
  MPI_Cart_sub(grid->comm, free_coords,
               &(grid->row_comm));

  /* Set up column communicators */
  free_coords[0] = 1;
  free_coords[1] = 0;
  MPI_Cart_sub(grid->comm, free_coords,
               &(grid->col_comm));
} /* Setup_grid */

void rearrange_result(float *block_C, float *res, int ma, int nb, GRID_INFO_T*
grid)
{
  float *buff;
  int i, j, k;
  int nb_bar, ma_bar;

  ma_bar = ma / grid->q;
  nb_bar = nb / grid->q;
  buff = (float*)malloc(ma_bar * nb * sizeof(float));

  for (i = 0; i < ma_bar; i++) // rearrange the entries in each row of grid
processes
  {
    MPI_Gather(block_C + i * nb_bar, nb_bar, MPI_FLOAT, buff + i * nb, nb_bar,
MPI_FLOAT, 0, grid->row_comm);
  }

  // gather the ordered entries from first column to the grid prosses in proc.0
  MPI_Gather(buff, ma_bar * nb, MPI_FLOAT, res, ma_bar * nb, MPI_FLOAT, 0, grid-
>col_comm);

  free(buff);
}
```