# Final Project Report

This project's target is to implement 2D convolution on parallel computers, using different parallelization techniques and models. In this project, we need to implement in four ways:

(a) Implement the 2D convolution using **SPMD model** and use **MPI send and receive** operations to perform communication. You need to run your program on 1, 2, 4, and 8 processors and provide speedups as well as computation and communication timings.

(b) Implement 2D convolution using **SPMD model** but use **MPI collective communication functions** wherever possible. You need to run your program on 1, 2, 4 and 8 processors and provide speedups as well as computation and communication timings.

(c) Implement 2D convolution model using S**PMD model using hybrid programming (MPIopenMP or MPI-Pthreads, Lecture 18).** You need to run your program on 1, 2, 4 and 8 processors, with 8 threads per processors. You need to provide speedups as well as computation and communication timings.

(d) Implement 2D convolution model using a **Task and Data Parallel Model**. You also need to show the use of communicators in MPI. Let's say we divide the P processors into four groups: P1, P2, P3, and P4. You will run Task 1 on P1 processors, Task 2 on P2 processors, Task 3 on P3 processors, and Task 4 on P4 processors. The Following figure illustrates this case. Report computation and communication results for P1=P2=P3=P4=2

But no matter in which problems, we should follow the 4 steps, or we can say four tasks.

**Task1: A=2D-FFT(Im1)**
**Task2:B=2D-FFT(Im2)**
**Task3:C=MM_Point(A,B)**
**Task4:Inverse-2DFFT(C)**

## 1. Code design

Before we start to implement using MPI, we should get the data from sample images. And the size of the images are both 512*512, so we can have image1[512][512] and image2[512][512].

### 1.1 MPI send and receive

In this part, the most important functions we use are MPI_Send and MPI_Recv. First, we let two images, image1 and image2, send each part according to other processors rank to them from processor0. And other processers wait and receive the data from processor 0

Processor 0 Sends Data

```
for(i = 1; i < size; i++){
      MPI_Send(&image1[i*(N/size)][0], N*N/size, mystruct, i, 0,
MPI_COMM_WORLD);
      MPI_Send(&image2[i*(N/size)][0], N*N/size, mystruct, i, 1,
MPI_COMM_WORLD);
}
```

Other processors receive data from processor0

```
for(i = 1; i < size; i++){
      MPI_Send(&image1[i*(N/size)][0], N*N/size, mystruct, i, 0,
MPI_COMM_WORLD);
      MPI_Send(&image2[i*(N/size)][0], N*N/size, mystruct, i, 1,
MPI_COMM_WORLD);
}
```

In this part of code, we can send data from processor0 to other processors. Here we should illustrate the parameter “mystruct”. We know MPI can send some basic data types directly, like int, float. But in this project we need to send complex number. Hence, we need to declare our own data type, mystruct.

Data Type: mystruct and myvector

```
    int blen[3] = {1,1,1};
    MPI_Aint indices[3];
    MPI_Datatype mystruct;
    MPI_Datatype myvector;
    MPI_Datatype old_types[3];

    old_types[0] = MPI_FLOAT;
    old_types[1] = MPI_FLOAT;
    old_types[2] = MPI_UB;

    indices[0] = 0;
    indices[1] = sizeof(float);
    indices[2] = sizeof(complex);

    MPI_Type_struct(3,blen,indices,old_types,&mystruct);
    MPI_Type_commit(&mystruct);
```

After we send the data, we just need to call the function, v_fft1d() and do 1D-FFT to each row. When finish 1D-FFT, we should transpose the whole image matrix, like temp[i][j] = image[i][j], image[j][i]=temp[i][j]. Then all processors send data back to processor0.

Other processors send back to processor0

```
MPI_Send(&image1[0][0],N*N/size,mystruct,0,rank,
            MPI_COMM_WORLD);
MPI_Send(&image2[0][0],N*N/size,mystruct,0,rank+N,
            MPI_COMM_WORLD);
```

Process0 reveives data from other processors

```
for(i=1; i<size; i++){

      MPI_Recv(&image1[i*N/size][0],N*N/size,mystruct,I
            ,i, MPI_COMM_WORLD,&status);

      MPI_Recv(&image2[i*N/size][0],N*N/size,mystruct,I
            ,i+N, MPI_COMM_WORLD,&status);
}
```

Now, we finish Row FFT, and the Col FFT is just the same. So we do not explain it more. After finish 2D-FFT to image1 and image2, we begin to multiply these two matrixes.

```
result[i][j].r = image1[i][j].r * image2[i][j].r;

result[i][j].i = image1[i][j].i * image2[i][j].i;
```

After point wise multiplication, we finish task3 and get a result matrix. The last step is to do inverse 2D-FFT to matrix result. The code is just like we do before.

**1.2 MPI scatter and gather**

Based on the previous design, we just need to know two functions: MPI_Scatter and MPI_Gather. Here I copy part of FFT to show how to use these two functions.

```
MPI_Scatter(&image1[0][0],N*(N/size),mystruct,&temp1[0][0],N*(N/size)
            ,mystruct,0,MPI_COMM_WORLD);

MPI_Scatter(&image2[0][0],N*(N/size),mystruct,&temp2[0][0],N*(N/size)
            ,mystruct,0,MPI_COMM_WORLD);

//1D-FFT
for(i = 0; i < N/size; i++){
    c_fft1d(&temp1[i][0],N,-1);
    c_fft1d(&temp2[i][0],N,-1);
}

MPI_Gather(&temp1[0][0],N*(N/size),mystruct,&image1[0][0],N*(N/size),
            mystruct,0,MPI_COMM_WORLD);

MPI_Gather(&temp2[0][0],N*(N/size),mystruct,&image2[0][0],N*(N/size),
            mystruct,0,MPI COMM WORLD);
```

From the code, we can see that MPI_Scatter and MPI_Gather are easier to use and we need to write less code to finish. The transposition of the images starts after Row FFT. And then we do scatter, 1D-FFT, and gather again. The code is almost the same and we do not explain more.

### 1.3 MPI-Pthreads

In this section, we use MPI and Pthreads to implement. Based on the MPI scatter and gather part, we just need to modify the code that execute in a processor and make them into different threads. The Pthreads has been used in homework 1.

After analysis the code in last section, we can parallel the 1D-FFT, transpose and multiplication. For example, let us see the difference between sequential and parallel.

Sequential

```
    for(i = 0; i < N/size; i++){
        c_fft1d(&temp1[i][0],N,-1);
        c_fft1d(&temp2[i][0],N,-1);
    }
```

Parallel in threads

```
    stripN = N / size / PR;
    long i;
    int rc;
    void* status;

    pthread_attr_t attr;
    pthread_t ffterid[PR];
    // set global thread attributes
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    if (sign == -1) {
        for (i = 0; i < PR; ++ i) {
            pthread_create(&ffterid[i], &attr, ffter, (void *)i);
        }
    }
    else {
        for (i = 0; i < PR; ++ i) {
            pthread_create(&ffterid[i], &attr, ffter_s, (void *)i);
        }
    }
```

### 1.4 Task and Data Parallel

Unlike the previous idea, in this implementation, we divide all processors into 4 groups. And each group finishes one of tasks. For example, we use 8 processors and make these 8 processors into group1(rank0, rank1), group2(rank2, rank3), group3(rank4, rank5), group4(rank6, rank7). The function, MPI_Comm_split, can help us divide all processors into groups.

```
color = rank/2;

MPI_Comm_split(MPI_COMM_WORLD,color,rank,&mycomm);
MPI_Comm_rank(mycomm,&rank_local);
MPI_Comm_size(mycomm,&size_local);
```

After split all processors, we have local rank and size under different color. Then we can reach any processor you want by the operation like code in the block.

```
if(color == 0){
    if(rank_local == 0){
        //do something
    }
}
```

Each group finishes one task. The relationship is:

Task1-------group1
Task2-------group2
Task3-------group3
Task4-------group4

And we should finish Task1 and Task2 firstly. Second, Image1 and image2 send to group3 to begin Task3 after 2D-FFT. Third, group3 finishes point wise multiplication. Last, Group3 send the data to group4 to finish inverse 2D-FFT and get the final results. The detail is almost like the previous problems.

## 2. Result Analysis

### 2.1 MPI send and receive

| Processor Number | Total cost time(ms) | MPI cost time(ms) | Sp |
|---|---|---|---|
| 1 | 415.98 | 130.89 | N/A |
| 2 | 277.33 | 89.11 | 1.47 |
| 4 | 253.68 | 64.40 | 2.03 |
| 8 | 287.29 | 58.43 | 2.24 |

In the table, we can find that when the number of processors increases, the cost of MPI time is less. However, we cannot say that if more processors are used, the speed will be faster and faster. The communication among processors is also need time.

### 2.2 MPI scatter and gather

| Processor Number | Total cost time(ms) | MPI cost time(ms) | Sp |
|---|---|---|---|
| 1 | 339.43 | 134.92 | N/A |
| 2 | 288.53 | 84.68 | 1.59 |
| 4 | 266.90 | 64.71 | 2.08 |
| 8 | 295.90 | 53.98 | 2.50 |

In this part, I just change the send and receive methods into scatter and gather, and keep other code the same. But the result is almost the same. Because scatter and gather is based on send and recv method.

**2.3 MPI-Pthread**

| Processor Number | Total cost time(ms) | MPI cost time(ms) | Sp |
|---|---|---|---|
| 1 | 250.96 | 44.16 | N/A |
| 2 | 249.92 | 43.81 | 1.01 |
| 4 | 251.69 | 41.18 | 1.07 |
| 8 | 322.29 | 90.09 | 0.49 |

From the data in the table, we find that the speed does not become fast when the processors increase. The reason is that when process FFD and multiplication we use 8 threads in a processor and it speeds up the whole process. If increase the number of processors, we cost more time on the communication among processors and time on the creating and join thread in each processor. So in the table, we see the time becomes more using 8 processors.

**2.4 Task and Data Parallel**

| Total cost time(ms) | MPI cost time(ms) |
|---|---|
| 251.28 | 75.51 |
| 449.28 | 74.57 |
| 450.60 | 74.54 |
| 388.90 | 74.30 |

In this part, we assign different tasks to different groups of processors. And the processors are 8 in the demand.

**2.5 compare the result from 2.1~2.4 under 8 processors.**

| | Total cost time(ms) | MPI cost time(ms) |
|---|---|---|
| A(Send/Recv) | 287.29 | 58.43 |
| B(Scatter/Gather) | 295.90 | 53.98 |
| C(MPI-Pthreads) | 322.29 | 90.09 |
| D(Task&Data) | 388.90 | 74.30 |

From the comparison among these 4 implementation models, the faster is MPI collective communication functions (Scatter/Gather) under using 8 processors. When we use MPI-Pthreads in 8 processors, it becomes the slowest because the processors number is too much or we can say that the source file data is not large enough.

Appendix

1. MPI Send/Recv: solutionA.c

```
/**
    Implement the 2D convolution using SPMD model and use MPI send and
receive
    operations to perform communication. You need to run your program
on 1, 2, 4, and 8
    processors and provide speedups as well as computation and
communication timings.
*/

#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

typedef struct {float r; float i;} complex;
static complex ctmp;

//size of the image
#define N 512
//file
FILE* fp;
//matrix
complex image1[N][N];
complex image2[N][N];
complex result[N][N];
complex temp1[N][N];
complex temp2[N][N];

#define C_SWAP(a,b) {ctmp=(a);(a)=(b);(b)=ctmp;}

void c_fft1d(complex *r, int      n, int      isign)
{
   int     m,i,i1,j,k,i2,l,l1,l2;
   float   c1,c2,z;
   complex t, u;

   if (isign == 0) return;

   /* Do the bit reversal */
   i2 = n >> 1;
   j = 0;
   for (i=0;i<n-1;i++) {
      if (i < j)
         C_SWAP(r[i], r[j]);
      k = i2;
      while (k <= j) {
         j -= k;
         k >>= 1;
      }
      j += k;
   }
```

```
    /* m = (int) log2((double)n); */
    for (i=n,m=0; i>1; m++,i/=2);

    /* Compute the FFT */
    c1 = -1.0;
    c2 =  0.0;
    l2 =  1;
    for (l=0;l<m;l++) {
       l1   = l2;
       l2 <<= 1;
       u.r = 1.0;
       u.i = 0.0;
       for (j=0;j<l1;j++) {
          for (i=j;i<n;i+=l2) {
             i1 = i + l1;

             /* t = u * r[i1] */
             t.r = u.r * r[i1].r - u.i * r[i1].i;
             t.i = u.r * r[i1].i + u.i * r[i1].r;

             /* r[i1] = r[i] - t */
             r[i1].r = r[i].r - t.r;
             r[i1].i = r[i].i - t.i;

             /* r[i] = r[i] + t */
             r[i].r += t.r;
             r[i].i += t.i;
          }
          z =  u.r * c1 - u.i * c2;

          u.i = u.r * c2 + u.i * c1;
          u.r = z;
       }
       c2 = sqrt((1.0 - c1) / 2.0);
       if (isign == -1) /* FWD FFT */
          c2 = -c2;
       c1 = sqrt((1.0 + c1) / 2.0);
    }

    /* Scaling for inverse transform */
    if (isign == 1) {       /* IFFT*/
       for (i=0;i<n;i++) {
          r[i].r /= n;
          r[i].i /= n;
       }
    }
}

void read(){
    if((fp = fopen("1_im1","r")) == NULL){
        printf("Cannot find the goal file.\n");
        exit(0);
    }
    int i,j;
    for(i = 0;i < N; i++){
        for(j = 0;j < N; j++){
```

```
            fscanf(fp,"%g",&image1[i][j].r);
            image1[i][j].i = 0;
        }
    }

    fclose(fp);

    if((fp = fopen("1_im2","r")) == NULL){
        printf("Cannot find the goal file.\n");
        exit(0);
    }
    for(i = 0;i < N; i++){
        for(j = 0;j < N; j++){
            fscanf(fp,"%g",&image2[i][j].r);
            image2[i][j].i = 0;
        }
    }
    fclose(fp);
    printf("Finish reading data.\n");
}

void write(){
    if((fp = fopen("my_out_1","w+")) == NULL){
        printf("Cannot find the goal file.\n");
        exit(0);
    }
    int i,j;
    for(i = 0;i < N; i++){
        for(j = 0;j < N; j++){
            fprintf(fp,"%6.2g",result[i][j].r);
        }
        fprintf(fp,"\n");
    }

    fclose(fp);
    printf("Finish writing data.\n");
}

int main(int argc, char **argv){
    int rank;
    int size;
    int i, j;
    double start_time, end_time;
    double mpi_start_time, mpi_end_time;


    /* Initial */
    MPI_Init(&argc, &argv);

    start_time = MPI_Wtime();

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //make own type of complex
    int blen[3] = {1,1,1};
    MPI_Aint indices[3];
```

```
    MPI_Datatype mystruct;
    MPI_Datatype myvector;
    MPI_Datatype old_types[3];

    old_types[0] = MPI_FLOAT;
    old_types[1] = MPI_FLOAT;
    old_types[2] = MPI_UB;

    indices[0] = 0;
    indices[1] = sizeof(float);
    indices[2] = sizeof(complex);

    MPI_Type_struct(3,blen,indices,old_types,&mystruct);
    MPI_Type_commit(&mystruct);
    MPI_Status status;

    /* read data file */
    if(rank == 0){
        //read file
        read();
    }

    MPI_Barrier(MPI_COMM_WORLD);

    mpi_start_time = MPI_Wtime();
    /* start 2D-FFT */
    //Row FFT
    if(rank == 0){

        //send data to other processes
        for(i = 1; i < size; i++){
            MPI_Send(&image1[i*(N/size)][0], N*N/size, mystruct, i, 0,
MPI_COMM_WORLD);
            MPI_Send(&image2[i*(N/size)][0], N*N/size, mystruct, i, 1,
MPI_COMM_WORLD);
        }

        //1D-fft
        for(i= 0; i < N/size; i++){
            c_fft1d(&image1[i][0], N, -1);
            c_fft1d(&image2[i][0], N, -1);
        }

        for(i=1; i<size; i++){
            MPI_Recv(&image1[i*N/size][0],N*N/size,mystruct,i,i,
MPI_COMM_WORLD,&status);
            MPI_Recv(&image2[i*N/size][0],N*N/size,mystruct,i,i+N,
MPI_COMM_WORLD,&status);
        }


        //printf("Processor 0 recieve all data. Row FFT Finish.\n");

    }else{
        MPI_Recv(&image1[0][0],N*N/size,mystruct,0,0,
MPI_COMM_WORLD,&status);
```

```
        MPI_Recv(&image2[0][0],N*N/size,mystruct,0,1,
MPI_COMM_WORLD,&status);

        //1D-fft
        for(i= 0; i < N/size; i++){
            c_fft1d(&image1[i][0], N, -1);
            c_fft1d(&image2[i][0], N, -1);
        }

        MPI_Send(&image1[0][0],N*N/size,mystruct,0,rank,
MPI_COMM_WORLD);
        MPI_Send(&image2[0][0],N*N/size,mystruct,0,rank+N,
MPI_COMM_WORLD);
    }//Row FFT finish

    if(rank == 0){
        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                temp1[i][j] = image1[i][j];
                temp2[i][j] = image2[i][j];
            }
        }

        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                image1[j][i] = temp1[i][j];
                image2[j][i] = temp2[i][j];
            }
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);

    //Col FFT
    if(rank == 0){
        //send data to other processes
        for(i = 1; i < size; i++){
            MPI_Send(&image1[i*(N/size)][0], N*N/size, mystruct, i, 0,
MPI_COMM_WORLD);
            MPI_Send(&image2[i*(N/size)][0], N*N/size, mystruct, i, 1,
MPI_COMM_WORLD);
        }

        //1D-fft
        for(i= 0; i < N/size; i++){
            c_fft1d(&image1[i][0], N, -1);
            c_fft1d(&image2[i][0], N, -1);
        }

        //send back
        for(i = 1;i < size; i++){
            MPI_Recv(&image1[i*N/size][0],N*N/size,mystruct,i,i,
MPI_COMM_WORLD,&status);
            MPI_Recv(&image2[i*N/size][0],N*N/size,mystruct,i,i+N,
MPI_COMM_WORLD,&status);
        }
```

```
        //printf("Processor 0 recieve all data. Col FFT Finish.\n");

    }else{
        MPI_Recv(&image1[0][0],N*N/size,mystruct,0,0,
MPI_COMM_WORLD,&status);
        MPI_Recv(&image2[0][0],N*N/size,mystruct,0,1,
MPI_COMM_WORLD,&status);

        for(i = 0; i < N/size; i++)
        {
            c_fft1d(&image1[i][0],N, -1);
            c_fft1d(&image2[i][0],N, -1);
        }

        MPI_Send(&image1[0][0],N*N/size,mystruct,0,rank,
MPI_COMM_WORLD);
        MPI_Send(&image2[0][0],N*N/size,mystruct,0,rank+N,
MPI_COMM_WORLD);
    }//Col FFT finish
    /* end of 2D-FFT*/

    MPI_Barrier(MPI_COMM_WORLD);


    /* Point-Wise Multiplication */
    if(rank == 0){
        //send data
        for(i = 1; i < size; i++){
            MPI_Send(&image1[i*(N/size)][0], N*N/size, mystruct, i, 0,
MPI_COMM_WORLD);
            MPI_Send(&image2[i*(N/size)][0], N*N/size, mystruct, i, 1,
MPI_COMM_WORLD);
        }

        for(i = 0; i < N/size; i++){
            for(j = 0; j < N; j++){
                result[i][j].r = image1[i][j].r * image2[i][j].r;
                result[i][j].i = image1[i][j].i * image2[i][j].i;
            }
        }

        for(i = 1; i < size; i++){
            MPI_Recv(&result[i*N/size][0], N*N/size, mystruct, i, i,
MPI_COMM_WORLD, &status);
        }

        //printf("Finish Point-Wise Multiplication \n");

    }else{
        //receive data from processor 0
        MPI_Recv(&image1[0][0],N*N/size,mystruct,0,0,
MPI_COMM_WORLD,&status);
        MPI_Recv(&image2[0][0],N*N/size,mystruct,0,1,
MPI_COMM_WORLD,&status);

        for(i = 0; i < N/size; i++){
            for(j = 0; j < N; j++){
```

```
                result[i][j].r = image1[i][j].r * image2[i][j].r;
                result[i][j].i = image1[i][j].i * image2[i][j].i;
            }
        }

        MPI_Send(&result[0][0],N*N/size,mystruct,0,rank,
MPI_COMM_WORLD);
    }//point-wise multiplication end

    MPI_Barrier(MPI_COMM_WORLD);

    /* start inverse 2D-FFT */
    //Row inverse-FFT
    if(rank == 0){
        //send data to other processes
        for(i = 1; i < size; i++){
            MPI_Send(&result[i*(N/size)][0], N*N/size, mystruct, i, 0,
MPI_COMM_WORLD);
        }

        //1D-fft
        for(i= 0; i < N/size; i++){
            c_fft1d(&result[i][0], N, 1);
        }

        //recieve data from other processes
        for(i=1; i<size; i++){
            MPI_Recv(&result[i*N/size][0],N*N/size,mystruct,i,i,
MPI_COMM_WORLD,&status);
        }

        //printf("Processor 0 recieve all data. Row inverse-FFT
Finish.\n");

    }else{
        MPI_Recv(&result[0][0],N*N/size,mystruct,0,0,
MPI_COMM_WORLD,&status);

        //1D-fft
        for(i= 0; i < N/size; i++){
            c_fft1d(&result[i][0], N, 1);
        }

        MPI_Send(&result[0][0],N*N/size,mystruct,0,rank,
MPI_COMM_WORLD);
    }//Row inverse-FFT end

    if(rank == 0){
        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                temp1[i][j] = result[i][j];
            }
        }

        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                result[j][i] = temp1[i][j];
```

```
            }
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);

    //Col inverse-FFT
    if(rank == 0){
        //send data to other processes
        for(i = 1; i < size; i++){
            MPI_Send(&result[i*(N/size)][0], N*N/size, mystruct, i, 0,
MPI_COMM_WORLD);
        }

        //1D-fft
        for(i= 0; i < N/size; i++){
            c_fft1d(&result[i][0], N, 1);
        }

        //send back
        for(i = 1;i < size; i++){
            MPI_Recv(&result[i*N/size][0],N*N/size,mystruct,i,i,
MPI_COMM_WORLD,&status);
        }

        //printf("Processor 0 recieve all data. Col inverse-FFT
Finish.\n");

    }else{
        MPI_Recv(&result[0][0],N*N/size,mystruct,0,0,
MPI_COMM_WORLD,&status);
        for(i = 0; i < N/size; i++)
        {
            c_fft1d(&result[i][0],N, 1);
        }

        MPI_Send(&result[0][0],N*N/size,mystruct,0,rank,
MPI_COMM_WORLD);
    }//Col FFT finish
    /* end of 2D-inverse-FFT*/

    MPI_Barrier(MPI_COMM_WORLD);

    if(rank == 0){
        mpi_end_time = MPI_Wtime();

        write();

        end_time = MPI_Wtime();

        printf("MPI Cost time: %f (ms) \n", (mpi_end_time-
mpi_start_time)*1000);
        printf("Total time: %f (ms) \n", (end_time-start_time)*1000);

        printf("Finish All!!!\n");
    }
```

```
    MPI_Type_free(&mystruct);
    //MPI_Type_free(&myvector);
    MPI_Finalize();
    exit(0);

}
```

2. MPI collective communication function: solutionB.c

```
/**
    Implement 2D convolution using SPMD model but use MPI collective
communication
    functions wherever possible. You need to run your program on 1, 2,
4 and 8 processors and
    provide speedups as well as computation and communication timings.
*/

#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

typedef struct {float r; float i;} complex;
static complex ctmp;

//size of the image
#define N 512
//file
FILE* fp;
//matrix
complex image1[N][N];
complex image2[N][N];
complex result[N][N];
complex temp1[N][N];
complex temp2[N][N];

#define C_SWAP(a,b) {ctmp=(a);(a)=(b);(b)=ctmp;}

void c_fft1d(complex *r, int      n, int      isign)
{
   int     m,i,i1,j,k,i2,l,l1,l2;
   float   c1,c2,z;
   complex t, u;

   if (isign == 0) return;

   /* Do the bit reversal */
   i2 = n >> 1;
   j = 0;
   for (i=0;i<n-1;i++) {
      if (i < j)
         C_SWAP(r[i], r[j]);
      k = i2;
      while (k <= j) {
```

```
            j -= k;
            k >>= 1;
         }
         j += k;
      }

      /* m = (int) log2((double)n); */
      for (i=n,m=0; i>1; m++,i/=2);

      /* Compute the FFT */
      c1 = -1.0;
      c2 =  0.0;
      l2 =  1;
      for (l=0;l<m;l++) {
         l1   = l2;
         l2 <<= 1;
         u.r = 1.0;
         u.i = 0.0;
         for (j=0;j<l1;j++) {
            for (i=j;i<n;i+=l2) {
               i1 = i + l1;

               /* t = u * r[i1] */
               t.r = u.r * r[i1].r - u.i * r[i1].i;
               t.i = u.r * r[i1].i + u.i * r[i1].r;

               /* r[i1] = r[i] - t */
               r[i1].r = r[i].r - t.r;
               r[i1].i = r[i].i - t.i;

               /* r[i] = r[i] + t */
               r[i].r += t.r;
               r[i].i += t.i;
            }
            z =  u.r * c1 - u.i * c2;

            u.i = u.r * c2 + u.i * c1;
            u.r = z;
         }
         c2 = sqrt((1.0 - c1) / 2.0);
         if (isign == -1) /* FWD FFT */
            c2 = -c2;
         c1 = sqrt((1.0 + c1) / 2.0);
      }

      /* Scaling for inverse transform */
      if (isign == 1) {       /* IFFT*/
         for (i=0;i<n;i++) {
            r[i].r /= n;
            r[i].i /= n;
         }
      }
   }

   void read(){
       if((fp = fopen("1_im1","r")) == NULL){
           printf("Cannot find the goal file.\n");
```

```
        exit(0);
    }
    int i,j;
    for(i = 0;i < N; i++){
        for(j = 0;j < N; j++){
            fscanf(fp,"%g",&image1[i][j].r);
            image1[i][j].i = 0;
        }
    }

    fclose(fp);

    if((fp = fopen("1_im2","r")) == NULL){
        printf("Cannot find the goal file.\n");
        exit(0);
    }
    for(i = 0;i < N; i++){
        for(j = 0;j < N; j++){
            fscanf(fp,"%g",&image2[i][j].r);
            image2[i][j].i = 0;
        }
    }
    fclose(fp);
    printf("Finish reading data.\n");
}

void write(){
    if((fp = fopen("my_out_1","w+")) == NULL){
        printf("Cannot find the goal file.\n");
        exit(0);
    }
    int i,j;
    for(i = 0;i < N; i++){
        for(j = 0;j < N; j++){
            fprintf(fp,"%6.2g",result[i][j].r);
        }
        fprintf(fp,"\n");
    }

    fclose(fp);
    printf("Finish writing data.\n");
}

int main(int argc, char **argv){
    int rank;
    int size;
    int i, j;
    double start_time, end_time;
    double mpi_start_time, mpi_end_time;


    /* Initial */
    MPI_Init(&argc, &argv);

    start_time = MPI_Wtime();

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //make own type of complex
    int blen[3] = {1,1,1};
    MPI_Aint indices[3];
    MPI_Datatype mystruct;
    //MPI_Datatype myvector;
    MPI_Datatype old_types[3];

    old_types[0] = MPI_FLOAT;
    old_types[1] = MPI_FLOAT;
    old_types[2] = MPI_UB;

    indices[0] = 0;
    indices[1] = sizeof(float);
    indices[2] = sizeof(complex);

    MPI_Type_struct(3,blen,indices,old_types,&mystruct);
    MPI_Type_commit(&mystruct);
    //MPI_Type_vector(N,N/size,N,mystruct,&myvector);
    //MPI_Type_commit(&myvector);
    MPI_Status status;

    /* read data file */
    if(rank == 0){
        //read file
        read();
    }

    MPI_Barrier(MPI_COMM_WORLD);

    mpi_start_time = MPI_Wtime();
    /* start 2D-FFT */
    //Row FFT

MPI_Scatter(&image1[0][0],N*(N/size),mystruct,&temp1[0][0],N*(N/size),m
ystruct,0,MPI_COMM_WORLD);

MPI_Scatter(&image2[0][0],N*(N/size),mystruct,&temp2[0][0],N*(N/size),m
ystruct,0,MPI_COMM_WORLD);

    for(i = 0; i < N/size; i++){
        c_fft1d(&temp1[i][0],N,-1);
        c_fft1d(&temp2[i][0],N,-1);
    }


MPI_Gather(&temp1[0][0],N*(N/size),mystruct,&image1[0][0],N*(N/size),my
struct,0,MPI_COMM_WORLD);

MPI_Gather(&temp2[0][0],N*(N/size),mystruct,&image2[0][0],N*(N/size),my
struct,0,MPI_COMM_WORLD);

    //Transpose in processor 0
    if(rank == 0){
        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
```

```
                temp1[i][j] = image1[i][j];
                temp2[i][j] = image2[i][j];
            }
        }

        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                image1[j][i] = temp1[i][j];
                image2[j][i] = temp2[i][j];
            }
        }
    }

    //Col FFT

MPI_Scatter(&image1[0][0],N*(N/size),mystruct,&temp1[0][0],N*(N/size),m
ystruct,0,MPI_COMM_WORLD);

MPI_Scatter(&image2[0][0],N*(N/size),mystruct,&temp2[0][0],N*(N/size),m
ystruct,0,MPI_COMM_WORLD);

    for(i = 0; i < N/size; i++){
        c_fft1d(&temp1[i][0],N,-1);
        c_fft1d(&temp2[i][0],N,-1);
    }


MPI_Gather(&temp1[0][0],N*(N/size),mystruct,&image1[0][0],N*(N/size),my
struct,0,MPI_COMM_WORLD);

MPI_Gather(&temp2[0][0],N*(N/size),mystruct,&image2[0][0],N*(N/size),my
struct,0,MPI_COMM_WORLD);
    /* end of 2D-FFT*/


    /* Point-Wise Multiplication */

MPI_Scatter(&image1[0][0],N*(N/size),mystruct,&temp1[0][0],N*(N/size),m
ystruct,0,MPI_COMM_WORLD);

MPI_Scatter(&image2[0][0],N*(N/size),mystruct,&temp2[0][0],N*(N/size),m
ystruct,0,MPI_COMM_WORLD);

    for( i = 0; i < N/size; i++){
        for(j = 0; j < N; j++){
            temp1[i][j].r = temp1[i][j].r * temp2[i][j].r;
            temp1[i][j].i = temp1[i][j].i * temp2[i][j].i;
        }
    }


MPI_Gather(&temp1[0][0],N*(N/size),mystruct,&result[0][0],N*(N/size),my
struct,0,MPI_COMM_WORLD);
    /* point wise multiplication end */


    /* start inverse 2D-FFT */
```

```
MPI_Scatter(&result[0][0],N*(N/size),mystruct,&temp1[0][0],N*(N/size),m
ystruct,0,MPI_COMM_WORLD);

    for(i = 0; i < N/size; i++){
        c_fft1d(&temp1[i][0],N,1);
    }


MPI_Gather(&temp1[0][0],N*(N/size),mystruct,&result[0][0],N*(N/size),my
struct,0,MPI_COMM_WORLD);

    if(rank == 0){
        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                temp1[i][j] = result[i][j];
            }
        }

        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                result[j][i] = temp1[i][j];
            }
        }
    }


MPI_Scatter(&result[0][0],N*(N/size),mystruct,&temp1[0][0],N*(N/size),m
ystruct,0,MPI_COMM_WORLD);

    for(i = 0; i < N/size; i++){
        c_fft1d(&temp1[i][0],N,1);
    }


MPI_Gather(&temp1[0][0],N*(N/size),mystruct,&result[0][0],N*(N/size),my
struct,0,MPI_COMM_WORLD);
    /* end of 2D-inverse-FFT*/

    if(rank == 0){
        mpi_end_time = MPI_Wtime();

        write();

        end_time = MPI_Wtime();

        printf("MPI Cost time: %f (ms) \n", (mpi_end_time-
mpi_start_time)*1000);
        printf("Total time: %f (ms) \n", (end_time-start_time)*1000);

        printf("Finish All!!!\n");
    }
    MPI_Type_free(&mystruct);
    //MPI_Type_free(&myvector);
    MPI_Finalize();
    exit(0);
```

```
}
```

3. MPI-Pthreads: solutionC.c

```
/**
Implement 2D convolution model using SPMD model using hybrid
programming (MPIopenMP
or MPI-Pthreads, Lecture 18). You need to run your program on 1, 2, 4
and 8
processors, with 8 threads per processors. You need to provide speedups
as well as
computation and communication timings.
*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>

#include <mpi.h>
#include <pthread.h>

#define C_SWAP(a,b) { ctmp = (a); (a) = (b); (b) = ctmp; }
//size of the image
#define     N       512
#define     PR      8

/* struct complex_s {
    float r;
    float i;
};

typedef struct complex_s complex;
static complex ctmp; */

typedef struct {float r; float i;} complex;
static complex ctmp;

//matrix
static complex image1[N][N];
static complex image2[N][N];
static complex result[N][N];
static complex temp1[N][N];
static complex temp2[N][N];

static int stripN;

void c_fft1d(complex* r, int n, int isign)
{
    int     m, i, i1, j, k, i2, l, l1, l2;
    float   c1, c2, z;
    complex t, u;

    if (isign == 0) return;
```

```
/* Do the bit reversal */
i2 = n >> 1;
j = 0;
for (i = 0; i < n - 1; i ++) {
    if (i < j)
        C_SWAP(r[i], r[j]);
    k = i2;
    while (k <= j) {
        j -= k;
        k >>= 1;
    }
    j += k;
}

/* m = (int) log2((double)n); */
for (i = n, m = 0; i > 1; m ++, i /= 2);

/* Compute the FFT */
c1 = -1.0;
c2 =  0.0;
l2 =  1;
for (l = 0; l < m; l ++) {
    l1  = l2;
    l2 <<= 1;
    u.r = 1.0;
    u.i = 0.0;
    for (j = 0; j < l1; j ++) {
        for (i = j; i < n; i += l2) {
            i1 = i + l1;

            /* t = u * r[i1] */
            t.r = u.r * r[i1].r - u.i * r[i1].i;
            t.i = u.r * r[i1].i + u.i * r[i1].r;

            /* r[i1] = r[i] - t */
            r[i1].r = r[i].r - t.r;
            r[i1].i = r[i].i - t.i;

            /* r[i] = r[i] + t */
            r[i].r += t.r;
            r[i].i += t.i;
        }
        z =  u.r * c1 - u.i * c2;

        u.i = u.r * c2 + u.i * c1;
        u.r = z;
    }
    c2 = sqrt((1.0 - c1) / 2.0);
    if (isign == -1) /* FWD FFT */
        c2 = -c2;
    c1 = sqrt((1.0 + c1) / 2.0);
}

/* Scaling for inverse transform */
if (isign == 1) {       /* IFFT*/
    for (i = 0; i < n;i ++) {
        r[i].r /= n;
```

```
            r[i].i /= n;
        }
    }
}

void read()
{
    FILE* fp = NULL;

    if ((fp = fopen("1_im1", "r")) == NULL) {
        printf("Cannot find the goal file.\n");
        exit(0);
    }

    int i, j;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            fscanf(fp, "%g", &image1[i][j].r);
            image1[i][j].i = 0;
        }
    }

    fclose(fp);

    if ((fp = fopen("1_im2", "r")) == NULL) {
        printf("Cannot find the goal file.\n");
        exit(0);
    }

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            fscanf(fp, "%g", &image2[i][j].r);
            image2[i][j].i = 0;
        }
    }

    fclose(fp);
    printf("Finish reading data.\n");
}

void write()
{
    FILE* fp = NULL;

    if ((fp = fopen("my_out_1", "w+")) == NULL) {
        printf("Cannot find the goal file.\n");
        exit(0);
    }

    int i, j;

    for (i = 0; i < N; i ++) {
        for (j = 0; j < N; j ++) {
            fprintf(fp, "%6.2g", result[i][j].r);
        }
        fprintf(fp, "\n");
```

```
    }

    fclose(fp);
    printf("Finish writing data.\n");
}

void* ffter(void* pId)
{
    int id = (int)pId;
    int s, t;

    s = id * stripN;
    t = s + stripN;

    for (; s < t; ++ s) {
        c_fft1d(&temp1[s][0], N, -1);
        c_fft1d(&temp2[s][0], N, -1);
    }

    return 0;
}

void* ffter_s(void* pId)
{
    int id = (int)pId;
    int s, t;

    s = id * stripN;
    t = s + stripN;

    for (; s < t; ++ s) {
        c_fft1d(&temp1[s][0], N, 1);
    }

    return 0;
}

void fft(int size, int sign)
{
    stripN = N / size / PR;
    long i;
    int rc;
    void* status;

    pthread_attr_t attr;
    pthread_t ffterid[PR];

    // set global thread attributes
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    if (sign == -1) {
        for (i = 0; i < PR; ++ i) {
            pthread_create(&ffterid[i], &attr, ffter, (void *)i);
        }
    }
    else {
```

```
        for (i = 0; i < PR; ++ i) {
            pthread_create(&ffterid[i], &attr, ffter_s, (void *)i);
        }
    }

    pthread_attr_destroy(&attr);
    for (i = 0; i < PR; ++ i) {
        rc = pthread_join(ffterid[i], &status);
        if (rc) {
            printf("ERROR; return code from pthread_join() is %d\n",
rc);
            exit(-1);
        }
#ifdef DEBUG
        printf("FFT: completed join with thread %ld having a status
of %ld\n", i, (long)status);
#endif
    }


}

void flip(int row)
{
    complex ctmp1, ctmp2;
    int counter;

    for (counter = row + 1; counter < N; ++ counter) {
        ctmp1 = image1[counter][row];
        ctmp2 = image2[counter][row];
        image1[counter][row] = image1[row][counter];
        image2[counter][row] = image2[row][counter];
        image1[row][counter] = ctmp1;
        image2[row][counter] = ctmp2;
    }
}

void flip_s(int row)
{
    complex ctmp_s;
    int counter;

    for (counter = row + 1; counter < N; ++ counter) {
        ctmp_s = result[counter][row];
        result[counter][row] = result[row][counter];
        result[row][counter] = ctmp_s;
    }
}

void* flipper(void *arg)
{

    int id =(int) arg;
    int s, t;

    s = id * stripN;
    t = s + stripN;
```

```
    for (; s < t; ++ s) {
        flip(s);
    }

    return 0;
}

void* flipper_s(void *arg)
{
    int id =(int) arg;
    int s, t;

    s = id * stripN;
    t = s + stripN;

    for (; s < t; ++ s) {
        flip_s(s);
    }

    return 0;
}

void transpose(int sign)
{
    stripN = N / PR;
    long i;
    int rc;
    void* status;

    pthread_attr_t attr;
    pthread_t flipperid[PR];

    // set global thread attributes
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    // create flippers
    if(sign == 1) {
        for (i = 0; i < PR; ++ i) {
            pthread_create(&flipperid[i], &attr, flipper, (void *)i);
        }
    }
    else {
        for (i = 0; i < PR; ++ i) {
            pthread_create(&flipperid[i], &attr, flipper_s, (void *)i);
        }
    }

    pthread_attr_destroy(&attr);
    for (i = 0; i < PR; ++ i) {
        rc = pthread_join(flipperid[i], &status);
        if (rc) {
            printf("ERROR; return code from pthread_join() is %d\n",
rc);
            exit(-1);
        }
```

```
#ifdef DEBUG
        printf("TRANSPOSE: completed join with thread %ld having a
status of %ld\n", i, (long)status);
#endif
    }
}

void multiply(int row)
{
    int i, j = 0;
    for(i = row; j < N; ++ j) {
        temp1[i][j].r = temp1[i][j].r * temp2[i][j].r;
        temp1[i][j].i = temp1[i][j].i * temp2[i][j].i;
    }
}

void* multiplier(void* arg)
{
    int id =(int) arg;
    int s, t;

    s = id * stripN;
    t = s + stripN;

    for (; s < t; ++ s) {
        multiply(s);
    }

    return 0;
}

void mul(int size)
{
    stripN = N / size / PR;
    long i;
    int rc;
    void* status;

    pthread_attr_t attr;
    pthread_t multiplierid[PR];

    // set global thread attributes
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    // create flippers
    for (i = 0; i < PR; ++ i) {
        pthread_create(&multiplierid[i], &attr, multiplier, (void *)i);
    }

    pthread_attr_destroy(&attr);
    for (i = 0; i < PR; ++ i) {
        rc = pthread_join(multiplierid[i], &status);
        if (rc) {
            printf("ERROR; return code from pthread_join() is %d\n",
rc);
            exit(-1);
```

```
        }
#ifdef DEBUG
        printf("MUL: completed join with thread %ld having a status
of %ld\n", i, (long)status);
#endif
    }
}

int main(int argc, char* argv[])
{
    int rank = 0;
    int size = 0;
    int i = 0, j = 0;
    double start_time = 0, end_time = 0;
    double mpi_start_time = 0, mpi_end_time = 0;

    /* Initial */
    MPI_Init(&argc, &argv);

    start_time = MPI_Wtime();

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //make own type of complex
    int blen[3] = { 1, 1, 1 };
    MPI_Aint indices[3];
    MPI_Datatype mystruct;
    MPI_Datatype old_types[3];

    old_types[0] = MPI_FLOAT;
    old_types[1] = MPI_FLOAT;
    old_types[2] = MPI_UB;

    indices[0] = 0;
    indices[1] = sizeof(float);
    indices[2] = sizeof(complex);

    MPI_Type_struct(3, blen, indices, old_types, &mystruct);
    MPI_Type_commit(&mystruct);

    if(rank == 0) {
        //read file
        read();
    }
    mpi_start_time = MPI_Wtime();

    /* start 2D-FFT */
    //Row FFT
    MPI_Scatter(&image1[0][0], N * (N / size), mystruct, &temp1[0][0],
N * (N / size), mystruct, 0, MPI_COMM_WORLD);
    MPI_Scatter(&image2[0][0], N * (N / size), mystruct, &temp2[0][0],
N * (N / size), mystruct, 0 ,MPI_COMM_WORLD);

    // pthread
    fft(size, -1);
```

```
   /*  for(i = 0; i < N/size; i++){
        c_fft1d(&temp1[i][0],N,-1);
        c_fft1d(&temp2[i][0],N,-1);
    } */


    MPI_Gather(&temp1[0][0], N * (N / size), mystruct, &image1[0][0], N
* (N / size), mystruct, 0, MPI_COMM_WORLD);
    MPI_Gather(&temp2[0][0], N * (N / size), mystruct, &image2[0][0], N
* (N / size), mystruct, 0, MPI_COMM_WORLD);

    //int j = 0;
    for (i = 0; i < N; ++ i) {
        for (j = 0; j < N; ++ j) {
            temp1[i][j].r = 0; temp1[i][j].i = 0;
            temp2[i][j].r = 0; temp2[i][j].i = 0;
        }
    }

    //Transpose in processor 0
    if(rank == 0) {
        /* for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                temp1[i][j] = image1[i][j];
                temp2[i][j] = image2[i][j];
            }
        }

        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                image1[j][i] = temp1[i][j];
                image2[j][i] = temp2[i][j];
            }
        } */
        transpose(1);
    }

    //Col FFT
    MPI_Scatter(&image1[0][0], N * (N / size), mystruct, &temp1[0][0],
N * (N / size), mystruct, 0, MPI_COMM_WORLD);
    MPI_Scatter(&image2[0][0], N * (N / size), mystruct, &temp2[0][0],
N * (N / size), mystruct, 0, MPI_COMM_WORLD);

    // pthread
    fft(size, -1);
    /* for(i = 0; i < N/size; i++){
        c_fft1d(&temp1[i][0],N,-1);
        c_fft1d(&temp2[i][0],N,-1);
    } */

    MPI_Gather(&temp1[0][0], N * (N / size), mystruct, &image1[0][0], N
* (N / size), mystruct, 0, MPI_COMM_WORLD);
    MPI_Gather(&temp2[0][0], N * (N / size), mystruct, &image2[0][0], N
* (N / size), mystruct, 0, MPI_COMM_WORLD);
    /* end of 2D-FFT*/
```

```
    /* Point-Wise Multiplication */
    MPI_Scatter(&image1[0][0], N * (N / size), mystruct, &temp1[0][0],
N * (N / size), mystruct, 0, MPI_COMM_WORLD);
    MPI_Scatter(&image2[0][0], N * (N / size), mystruct, &temp2[0][0],
N * (N / size), mystruct, 0, MPI_COMM_WORLD);

    // pthread mul
    mul(size);
    /* for( i = 0; i < N/size; i++){
        for(j = 0; j < N; j++){
            temp1[i][j].r = temp1[i][j].r * temp2[i][j].r;
            temp1[i][j].i = temp1[i][j].i * temp2[i][j].i;
        }
    } */

    MPI_Gather(&temp1[0][0], N * (N / size), mystruct, &result[0][0], N
* (N / size), mystruct, 0, MPI_COMM_WORLD);
    /* point wise multiplication end */


    /* start inverse 2D-FFT */
    MPI_Scatter(&result[0][0], N * (N / size), mystruct, &temp1[0][0],
N * (N / size), mystruct, 0, MPI_COMM_WORLD);

    // pthread iFFT per r
    fft(size, 1);
    /* for(i = 0; i < N/size; i++){
        c_fft1d(&temp1[i][0],N,1);
    } */

    MPI_Gather(&temp1[0][0], N * (N / size), mystruct, &result[0][0], N
* (N / size), mystruct, 0, MPI_COMM_WORLD);

    // pthread transpose
    if(rank == 0) {
        /* for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                temp1[i][j] = result[i][j];
            }
        }

        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                result[j][i] = temp1[i][j];
            }
        } */
        transpose(2);
    }

    MPI_Scatter(&result[0][0], N * (N / size), mystruct, &temp1[0][0],
N * (N / size), mystruct, 0, MPI_COMM_WORLD);

    // pthread iFFT per r
    fft(size, 1);
    /* for(i = 0; i < N/size; i++){
        c_fft1d(&temp1[i][0],N,1);
    } */
```

```
    MPI_Gather(&temp1[0][0], N * (N / size), mystruct, &result[0][0], N
* (N / size), mystruct, 0, MPI_COMM_WORLD);
    /* end of 2D-inverse-FFT*/

    if(rank == 0) {
        mpi_end_time = MPI_Wtime();

        write();

        end_time = MPI_Wtime();

        printf("MPI Cost time: %f (ms) \n", (mpi_end_time-
mpi_start_time) * 1000);
        printf("Total time: %f (ms) \n", (end_time-start_time) * 1000);

        printf("Finish All!!!\n");
    }

    MPI_Type_free(&mystruct);
    MPI_Finalize();

    return 0;
}
```

4. Task and Data Parallel: solutionD.c

```
/**
    Implement 2D convolution model using a Task and Data Parallel Model.
You also need to
    show the use of communicators in MPI. Let's say we divide the P
processors into four groups:
    P1, P2, P3, and P4. You will run Task 1 on P1 processors, Task 2 on
P2 processors, Task 3
    on P3 processors, and Task 4 on P4 processors. The Following figure
illustrates this case.
    Report computation and communication results for P1=P2=P3=P4=2.
*/

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <assert.h>
#include <math.h>

typedef struct {float r; float i;} complex;
static complex ctmp;

//size of the image
#define N 512
//file
FILE* fp;
//matrix
complex image1[N][N];
complex image2[N][N];
complex result[N][N];
```

```
complex temp1[N][N];
complex temp2[N][N];

#define C_SWAP(a,b) {ctmp=(a);(a)=(b);(b)=ctmp;}

void c_fft1d(complex *r, int      n, int      isign)
{
   int     m,i,i1,j,k,i2,l,l1,l2;
   float   c1,c2,z;
   complex t, u;

   if (isign == 0) return;

   /* Do the bit reversal */
   i2 = n >> 1;
   j = 0;
   for (i=0;i<n-1;i++) {
      if (i < j)
         C_SWAP(r[i], r[j]);
      k = i2;
      while (k <= j) {
         j -= k;
         k >>= 1;
      }
      j += k;
   }

   /* m = (int) log2((double)n); */
   for (i=n,m=0; i>1; m++,i/=2);

   /* Compute the FFT */
   c1 = -1.0;
   c2 =  0.0;
   l2 =  1;
   for (l=0;l<m;l++) {
      l1   = l2;
      l2 <<= 1;
      u.r = 1.0;
      u.i = 0.0;
      for (j=0;j<l1;j++) {
         for (i=j;i<n;i+=l2) {
            i1 = i + l1;

            /* t = u * r[i1] */
            t.r = u.r * r[i1].r - u.i * r[i1].i;
            t.i = u.r * r[i1].i + u.i * r[i1].r;

            /* r[i1] = r[i] - t */
            r[i1].r = r[i].r - t.r;
            r[i1].i = r[i].i - t.i;

            /* r[i] = r[i] + t */
            r[i].r += t.r;
            r[i].i += t.i;
         }
         z =  u.r * c1 - u.i * c2;
```

```
            u.i = u.r * c2 + u.i * c1;
            u.r = z;
         }
         c2 = sqrt((1.0 - c1) / 2.0);
         if (isign == -1) /* FWD FFT */
            c2 = -c2;
         c1 = sqrt((1.0 + c1) / 2.0);
      }

      /* Scaling for inverse transform */
      if (isign == 1) {       /* IFFT*/
         for (i=0;i<n;i++) {
            r[i].r /= n;
            r[i].i /= n;
         }
      }
}

void read_image1(){
    if((fp = fopen("1_im1","r")) == NULL){
        printf("Cannot find the goal file.\n");
        exit(0);
    }
    int i,j;
    for(i = 0;i < N; i++){
        for(j = 0;j < N; j++){
            fscanf(fp,"%g",&image1[i][j].r);
            image1[i][j].i = 0;
        }
    }

    fclose(fp);
    printf("Finish reading data from image1.\n");
}

void read_image2(){
    if((fp = fopen("1_im2","r")) == NULL){
        printf("Cannot find the goal file.\n");
        exit(0);
    }
    int i,j;
    for(i = 0;i < N; i++){
        for(j = 0;j < N; j++){
            fscanf(fp,"%g",&image2[i][j].r);
            image2[i][j].i = 0;
        }
    }
    fclose(fp);
    printf("Finish reading data from image2.\n");
}

void write(){
    if((fp = fopen("my_out_1","w+")) == NULL){
        printf("Cannot find the goal file.\n");
        exit(0);
    }
    int i,j;
```

```
    for(i = 0;i < N; i++){
        for(j = 0;j < N; j++){
            fprintf(fp,"%6.2g",result[i][j].r);
        }
        fprintf(fp,"\n");
    }

    fclose(fp);
    printf("Finish writing data.\n");
}

int main(int argc, char **argv){
    int rank, size; //gobal rank and size
    int rank_local, size_local; //local rank and size
    int color; //control of subset assignment (nonnegative integer).
               //Processes with the same color are in the same new
communicator
    int i, j;
    double start_time, end_time;
    double mpi_start_time, mpi_end_time;
    MPI_Comm mycomm;
    MPI_Status status;

    /* Initial */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    start_time = MPI_Wtime();
    /*
     * (rank 0,1),(rank 2,3), (rank 4,5) (rank 6,7)
     *  color=0    color=1     color=2    color=3
     * split 8 processes into 4 group
     */
    color = rank/2;
    MPI_Comm_split(MPI_COMM_WORLD,color,rank,&mycomm);
    MPI_Comm_rank(mycomm,&rank_local);
    MPI_Comm_size(mycomm,&size_local);

    //make own type of complex
    int blen[3] = {1,1,1};
    MPI_Aint indices[3];
    MPI_Datatype mystruct;
    MPI_Datatype myvector;
    MPI_Datatype old_types[3];

    old_types[0] = MPI_FLOAT;
    old_types[1] = MPI_FLOAT;
    old_types[2] = MPI_UB;

    indices[0] = 0;
    indices[1] = sizeof(float);
    indices[2] = sizeof(complex);

    MPI_Type_struct(3,blen,indices,old_types,&mystruct);
    MPI_Type_commit(&mystruct);
    //MPI_Type_vector(N,N/size,N,mystruct,&myvector);
    //MPI_Type_commit(&myvector);
```

```
    /* read file, rank0 and rank2 read image1 and image2, respectively
*/
    if(color == 0){
        //rank0 read image1
        if(rank_local == 0){
            read_image1();
        }
    }

    if(color == 1){
        if(rank_local == 0){
            read_image2();
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    mpi_start_time = MPI_Wtime();

    /* ******************
    *        Task 1      *
    *    2D-FFT Image1   *
    * ******************/
    if(color == 0){
        /* rank0 and rank1 finish 2D-FFT of image1 */
        //Row FFT of image1

MPI_Scatter(&image1[0][0],N*N/size_local,mystruct,&temp1[0][0],N*N/size
_local,mystruct,0,mycomm);
        for(i = 0; i < N/size_local;i++){
            c_fft1d(&temp1[i][0],N, -1);
        }

MPI_Gather(&temp1[0][0],N*N/size_local,mystruct,&image1[0][0],N*N/size_
local,mystruct,0,mycomm);

        //Transpose of image1
        if(rank_local == 0){
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    temp1[i][j] = image1[i][j];
                }
            }
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    image1[j][i] = temp1[i][j];
                }
            }
        }

        //Col FFT of image1

MPI_Scatter(&image1[0][0],N*N/size_local,mystruct,&temp1[0][0],N*N/size
_local,mystruct,0,mycomm);
        for(i = 0; i < N/size_local;i++){
            c_fft1d(&temp1[i][0],N, -1);
        }
```

```
MPI_Gather(&temp1[0][0],N*N/size_local,mystruct,&image1[0][0],N*N/size_
local,mystruct,0,mycomm);

        if(rank_local == 0){
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    temp1[i][j] = image1[i][j];
                }
            }
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    image1[j][i] = temp1[i][j];
                }
            }
        }

        //printf("Task1 finish!! \n");
    }

    /* ******************
    *        Task 2      *
    *    2D-FFT Image2   *
    * ******************/
    if(color == 1){
        /* rank2 and rank3 finish 2D-FFT of image2 */
        //Row FFT of image2

MPI_Scatter(&image2[0][0],N*N/size_local,mystruct,&temp2[0][0],N*N/size
_local,mystruct,0,mycomm);
        for(i = 0; i < N/size_local;i++){
            c_fft1d(&temp2[i][0],N, -1);
        }

MPI_Gather(&temp2[0][0],N*N/size_local,mystruct,&image2[0][0],N*N/size_
local,mystruct,0,mycomm);

        //Transpose of image2
        if(rank_local == 0){
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    temp2[i][j] = image2[i][j];
                }
            }
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    image2[j][i] = temp2[i][j];
                }
            }
        }

        //Col FFT of image2

MPI_Scatter(&image2[0][0],N*N/size_local,mystruct,&temp2[0][0],N*N/size
_local,mystruct,0,mycomm);
        for(i = 0; i < N/size_local;i++){
            c_fft1d(&temp2[i][0],N, -1);
```

```
        }

MPI_Gather(&temp2[0][0],N*N/size_local,mystruct,&image2[0][0],N*N/size_
local,mystruct,0,mycomm);

        if(rank_local == 0){
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    temp2[i][j] = image2[i][j];
                }
            }
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    image2[j][i] = temp2[i][j];
                }
            }
        }

        //printf("Task2 finish!! \n");

    }

    //continue until task1 and task2 are finished
    //MPI_Barrier(MPI_COMM_WORLD);

    /* ******************
    *        Task 3      *
    *   Point-Wise Multi *
    * ******************/
    //before do point-wise multiplication, send data to rank4
    if(color == 0){
        if(rank_local == 0){
            MPI_Send(&image1[0][0],N*N,mystruct,4,0,MPI_COMM_WORLD);
        }
    }

    if(color == 1){
        if(rank_local == 0){
            MPI_Send(&image2[0][0],N*N,mystruct,4,0,MPI_COMM_WORLD);
        }
    }

    // start point-wise multiplication
    if(color == 2){
        if(rank_local == 0){

MPI_Recv(&image1[0][0],N*N,mystruct,0,0,MPI_COMM_WORLD,&status);

MPI_Recv(&image2[0][0],N*N,mystruct,2,0,MPI_COMM_WORLD,&status);
        }


MPI_Scatter(&image1[0][0],N*N/size_local,mystruct,&temp1[0][0],N*N/size
_local,mystruct,0,mycomm);

MPI_Scatter(&image2[0][0],N*N/size_local,mystruct,&temp2[0][0],N*N/size
_local,mystruct,0,mycomm);
```

```
        for(i = 0; i < N/size_local; i++){
            for(j = 0; j < N; j++){
                temp1[i][j].r = temp1[i][j].r * temp2[i][j].r;
                temp1[i][j].i = temp1[i][j].i * temp2[i][j].i;
            }
        }

MPI_Gather(&temp1[0][0],N*N/size_local,mystruct,&result[0][0],N*N/size_
local,mystruct,0,mycomm);
        //printf("Task3 finish!!! \n");
        if(rank_local==0){
            MPI_Send(&result[0][0],N*N,mystruct,6,0,MPI_COMM_WORLD);
        }
    }

    /* ******************
    *        Task 4      *
    *   Inverse 2D-FFT   *
    * ******************/

    //before inverse FFT, receive the data from rank4
    if(color == 3){
        if(rank_local == 0){

MPI_Recv(&result[0][0],N*N,mystruct,4,0,MPI_COMM_WORLD,&status);
        }

        /* rank6 and rank7 finish inverse 2D-FFT of result */

MPI_Scatter(&result[0][0],N*N/size_local,mystruct,&temp1[0][0],N*N/size
_local,mystruct,0,mycomm);
        for(i = 0; i < N/size_local;i++){
            c_fft1d(&temp1[i][0],N, 1);
        }

MPI_Gather(&temp1[0][0],N*N/size_local,mystruct,&result[0][0],N*N/size_
local,mystruct,0,mycomm);

        //Transpose of result
        if(rank_local == 0){
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    temp1[i][j] = result[i][j];
                }
            }
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    result[j][i] = temp1[i][j];
                }
            }
        }


MPI_Scatter(&result[0][0],N*N/size_local,mystruct,&temp1[0][0],N*N/size
_local,mystruct,0,mycomm);
        for(i = 0; i < N/size_local;i++){
            c_fft1d(&temp1[i][0],N, 1);
```

```
        }

MPI_Gather(&temp1[0][0],N*N/size_local,mystruct,&result[0][0],N*N/size_
local,mystruct,0,mycomm);

        if(rank_local == 0){
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    temp1[i][j] = result[i][j];
                }
            }
            for(i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                    result[j][i] = temp1[i][j];
                }
            }

            mpi_end_time = MPI_Wtime();
        }

        //printf("Task4 finish!! \n");

        if(rank_local == 0){

            write();
            end_time = MPI_Wtime();

            printf("MPI Cost time: %f (ms) \n", (mpi_end_time-
mpi_start_time)*1000);
            printf("Total time: %f (ms) \n", (end_time-
start_time)*1000);
        }
    }

    MPI_Type_free(&mystruct);
    MPI_Finalize();
    exit(0);
}
```