דוגמאות לקוד MPI מהמודל:

תוכן עניינים

2..... Hello

3..... Send-receive

4..... Get count + probe

5..... Sendreceive

6..... Broadcast

7.....Gather

8..... Scatter

9..... ANY

10..... MPI_Pack

11.....Barrier

12.....Cart create

13.....Type create

14..... Group

15..... Split

Hello

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

int main(int argc, char* argv[]){
	int  my_rank; /* rank of process */
	int  p;       /* number of processes */
	int source;   /* rank of sender */
	int dest;     /* rank of receiver */
	int tag=0;    /* tag for messages */
	char message[100];        /* storage for message */
	MPI_Status status ;   /* return status for receive */

	/* start up MPI */

	MPI_Init(&argc, &argv);

	/* find out process rank */
	MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

	/* find out number of processes */
	MPI_Comm_size(MPI_COMM_WORLD, &p);


	if (my_rank !=0){
		/* create message */
		sprintf(message, "Hello MPI World from process %d!", my_rank);
		dest = 0;
		/* use strlen+1 so that '\0' get transmitted */
		MPI_Send(message, strlen(message)+1, MPI_CHAR,
		   dest, tag, MPI_COMM_WORLD);
	}
	else {
		printf("Hello MPI World From process 0: Num processes: %d\n",p);
		for (source = 1; source < p; source++) {
			MPI_Recv(message, 100, MPI_CHAR, source, tag,
				MPI_COMM_WORLD, &status);
			printf("%s\n",message);
		}
	}
	/* shut down MPI */
	MPI_Finalize();


	return 0;
	}
```

Send-receive

```c
#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[]){
	int x, y;
	int  my_rank; /* rank of process */
	int  p;       /* number of processes */
	MPI_Status status ;   /* return status for receive */

	/* start up MPI */
	MPI_Init(&argc, &argv);

	/* find out process rank */
	MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

	/* find out number of processes */
	MPI_Comm_size(MPI_COMM_WORLD, &p);


	if (my_rank == 0){
		x = 7;
		MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
		MPI_Recv(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
		printf("The new value of x = %d\n", x);
	}
	else{
		MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
		y = y*3;
		MPI_Send(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
	}
	/* shut down MPI */
	MPI_Finalize();


	return 0;
}
```

Get count + probe

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
int main(int argc,char **argv)
{
    int myid, numprocs;
    MPI_Status status;
    int count;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);


    if (myid == 0) {
       int data[3] = {200, 300, 400};
       MPI_Send(data, 3, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else {
       int *msg, j;
       MPI_Probe(0 ,0, MPI_COMM_WORLD, &status);

       MPI_Get_count(&status, MPI_INT, &count);
       printf("getting count = %d\n",count);

       msg = (int*) malloc(count*sizeof(int));

       MPI_Recv(msg, count,MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

       for(j=0; j < count;  j++)
              printf("%d ",msg[j]);
       printf("\n");
    }
    MPI_Finalize();
}
```

Sendreceive

```
#include "mpi.h"
#include <stdio.h>

 int main(int argc, char *argv[])
 {
     int myid, numprocs, left, right;
     int buffer[10], buffer2[10];
       int x, y;
       int dest;
       int errorCode = 999;

     MPI_Status status;

     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &myid);

      // Check that there is exactly two proceeses
      if (numprocs != 2)
             MPI_Abort(MPI_COMM_WORLD, errorCode);

      // Define the rank of destination processes
       if (myid == 0)
             dest = 1;
       else
             dest = 0;

        // Send the its rank to the destination
        x = myid;

     MPI_Sendrecv(&x, 1, MPI_INT, dest, 0, &y, 1, MPI_INT, dest, 0,
      MPI_COMM_WORLD, &status);


       printf("myid = %d, x = %d,  y = %d\n", myid, x,  y);

     MPI_Finalize();
     return 0;
}
```

Broadcast

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

#define SIZE 10

 int main(int argc, char *argv[])
 {
     int myid, numprocs;
       char buffer[SIZE] = {'\0'};

       int errorCode = MPI_ERR_COMM;

     MPI_Status status;

     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &myid);

       // Exit if launced only one process
       if (numprocs == 1)
             MPI_Abort(MPI_COMM_WORLD, errorCode);

     // Initialize the send buffer in root process
       if (myid == 0)
             strcpy(buffer, "afeka");
       printf("before bcast myid = %d, buffer = %s\n", myid, buffer);

       // Perform the broadcast
       MPI_Bcast(buffer, SIZE, MPI_CHAR, 0, MPI_COMM_WORLD);
       printf("after bcast myid = %d, buffer = %s\n", myid, buffer);


     MPI_Finalize();
    return 0;
 }
```

Gather

```c
#include "mpi.h"
#include <stdio.h>

#define MAX_PROCESSES 10
#define SIZE  4

int main(int argc, char **argv)
{
	int rank, size;
	int data[2];

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);

	if (size != 4) {
		printf("launch 4 processes only\n");
		MPI_Abort(MPI_COMM_WORLD, 0);
	}

	// Local data to be gathered
	data[0] = rank;
	data[1] = rank + 1;

	if (rank != 0) {
		/* Scatter the big table to everybody's little table */
		MPI_Gather(data, 2, MPI_INT, NULL, 0, MPI_INT, 0, MPI_COMM_WORLD);
	} else {
		int result[2*SIZE];
		MPI_Gather(data, 2, MPI_INT, result, 2, MPI_INT, 0, MPI_COMM_WORLD);
		for (int i = 0;   i < SIZE;   i++)
			printf("%d %d\n", result[2*i], result[2*i + 1]);
	}

	MPI_Finalize();
	return 0;
}
```

Scatter

```c
#include "mpi.h"
#include <stdio.h>


int main(int argc, char **argv)
{
	int rank, size;
	int data[2];

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);

	if (size != 4) {
		printf("launch 4 processes only\n");
		MPI_Abort(MPI_COMM_WORLD, 0);
	}

	if (rank == 0) {
		int original[] = {1, 2, 3, 4, 5, 6, 7, 8};
		// Scatter the array - 2 integers for each process
		MPI_Scatter(original, 2, MPI_INT, data, 2, MPI_INT, 0, MPI_COMM_WORLD);
	} else
		MPI_Scatter(NULL, 0, MPI_INT, data, 2, MPI_INT, 0, MPI_COMM_WORLD);

    printf("rank = %d data = [%d, %d]\n", rank, data[0], data[1]);


	MPI_Finalize();
	return 0;
}
```

ANY

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

#define MAX   10
int main(int argc,char **argv)
{
    int myid, numprocs;
    MPI_Status status;
 int data[MAX];
    int count;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);


    if (myid == 0) {
 MPI_Recv(data, MAX, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
 MPI_Get_count(&status, MPI_INT, &count);
 printf("count = %d, tag = %d, source = %d\n", count, status.MPI_TAG, status.MPI_SOURCE);
    }
 else {
 int data[3] = { 3, 5, 7 };
 MPI_Send(data, 3, MPI_INT, 0, 45, MPI_COMM_WORLD);
 }
    MPI_Finalize();
}
```

MPI_Pack

```
#include "mpi.h"
#include <stdio.h>

#define BUFFER_SIZE 100
int main(int argc, char *argv[])
 {
     int rank, numprocs;
     int i;
        float f[3];
     char c;
        char buffer[BUFFER_SIZE];
        int position;
     MPI_Status status;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

     // Check that there is exactly two proceeses
     if (numprocs != 2)
       MPI_Abort(MPI_COMM_WORLD, 0);

     MPI_Comm_rank(MPI_COMM_WORLD, &rank);

     if (rank == 0) {
       // Define the data
        i = 123;
       f[0] = 200.2f;
       f[1] = 201.3f;
       f[2] = 202.4f;
       c = '#';

       // Start packing from the very beginning of the buffer
       position = 0;
        MPI_Pack(&i, 1, MPI_INT, buffer,  BUFFER_SIZE, &position, MPI_COMM_WORLD);
        MPI_Pack(f, 3, MPI_FLOAT, buffer, BUFFER_SIZE, &position, MPI_COMM_WORLD);
       MPI_Pack(&c, 1, MPI_CHAR, buffer, BUFFER_SIZE, &position, MPI_COMM_WORLD);

       // Send the packed message
        MPI_Send(buffer, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
       printf("rank %d sends %d %5.1f %5.1f %5.1f %c\n", rank, i, f[0], f[1], f[2], c);
      }
     if (rank == 1) {

       // Recieve the packed message
       MPI_Recv(buffer, BUFFER_SIZE, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);

       // Start unpacking to the very beginning of the buffer
       position = 0;
       MPI_Unpack(buffer, BUFFER_SIZE, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, BUFFER_SIZE, &position, f,  3, MPI_FLOAT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, BUFFER_SIZE, &position, &c, 1, MPI_CHAR, MPI_COMM_WORLD);
       printf("rank %d recieved %d %5.1f %5.1f %5.1f %c\n", rank, i, f[0], f[1], f[2], c);
     }

    MPI_Finalize();
     return 0;
 }
```

Barrier

```
#include "mpi.h"
#include <stdio.h>


void heavyTask(double sec) {
       double t = MPI_Wtime();

       // Loop till sec seconds will pass
    while (MPI_Wtime() - t < sec);
}
int main(int argc, char *argv[])
 {
     int rank, numprocs;
     double t1, t2;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        // Check that there is exactly two proceeses
        if (numprocs < 2)
               MPI_Abort(MPI_COMM_WORLD, 0);


        t1 = MPI_Wtime();
     if (rank == 1)
               heavyTask(2);

        MPI_Barrier(MPI_COMM_WORLD);
        t2 = MPI_Wtime();

     printf("myrank = %d time = %e\n", rank, t2-t1);
     MPI_Finalize();
     return 0;
                                                                        }
```

Cart create

```
#include<mpi.h>
#include<stdio.h>
#include <stdlib.h>
#define ROWS  4
#define COLUMNS 3
int main(int argc, char *argv[]) {
    int rank, size;
     MPI_Comm comm;
     int dim[2], period[2], reorder;
     int coord[2], id;
    int row, column;
    int source, dest;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);

     if (size != ROWS*COLUMNS || argc != 3) {
         printf("Please run with 12 processes and two parametrs - row and column.\n");fflush(stdout);
         MPI_Abort(MPI_COMM_WORLD, 1);
     }
    // Parameters from command line
    column = atoi(argv[1]);
    row = atoi(argv[2]);
    if (row > ROWS - 1 || column > COLUMNS - 1)  {
    printf("Please enter row < %d and column < %d\n", ROWS, COLUMNS); fflush(stdout);
         MPI_Abort(MPI_COMM_WORLD, 1);
     }
    // A two-dimensional cylindr of 12 processes in a 4x3 grid //
     dim[0] = COLUMNS;
    dim[1] = ROWS;
     period[0] = 0;
    period[1] = 0;
     reorder = 1;
     MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm);

    // Each process displays its rank and cartesian coordinates
     MPI_Cart_coords(comm, rank, 2, coord);
     printf("Rank %d coordinates are %d %d\n", rank, coord[0], coord[1]);fflush(stdout);

     if(rank==0) {
         coord[0] = column;
    coord[1] = row;
         MPI_Cart_rank(comm, coord, &id);
         printf("The processor at position (%d, %d) has rank %d\n", coord[0], coord[1], id);fflush(stdout)
     }
    if (rank == 5) {
       MPI_Cart_shift( comm, 0, 1, &source, &dest );
       printf("Rank = %d Source = %d  Destination %d\n", rank, source, dest); fflush(stdout);
    }
     MPI_Finalize();
      return 0;
}
```

Type create

```
#include "mpi.h"
#include <stdio.h>

 struct Particle
 {
     float x;
        float y;
        int color;
 };

 int main(int argc, char *argv[])
 {
     struct Particle particle;
     int myrank, size;
     MPI_Status status;
     MPI_Datatype ParticleMPIType;
     MPI_Datatype type[3] = { MPI_FLOAT, MPI_FLOAT, MPI_INT };
     int blocklen[3] = { 1, 1, 1 };
     MPI_Aint disp[3];

     MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (size != 2) {
         printf("Please run with 2 processes.\n");fflush(stdout);
         MPI_Abort(MPI_COMM_WORLD, 1);
     }


        // Create MPI user data type for partical
     disp[0] = (char *) &particle.x -     (char *) &particle;
     disp[1] = (char *) &particle.y -     (char *) &particle;
     disp[2] = (char *) &particle.color - (char *) &particle;
     MPI_Type_create_struct(3, blocklen, disp, type, &ParticleMPIType);
     MPI_Type_commit(&ParticleMPIType);


    // Send and recieve one struct of the ParticleMPIType type
    if (myrank == 0) {
               particle.x = 1.1;
               particle.y = 2.2;
               particle.color = 3;
               printf("myrank = %d x = %e  y = %e  color = %d\n", myrank, particle.x, particle.y,
particle.color);
         MPI_Send(&particle, 1, ParticleMPIType, 1, 0, MPI_COMM_WORLD);
     }
     else  if (myrank == 1) {
               struct Particle part;
         MPI_Recv(&part, 1, ParticleMPIType, 0, 0, MPI_COMM_WORLD, &status);
               printf("myrank = %d x = %e  y = %e  color = %d\n", myrank, part.x, part.y, part.color);
     }
     MPI_Finalize();
     return 0;
 }
```

Group

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MESSAGE_SIZE 6
int main(int argc, char *argv[])
{
	MPI_Group newGroup, worldGroup;
	int ranks[16] = { 2, 4, 7 }, size, newSize, newRank, rank;
	char msg[MESSAGE_SIZE] = { '\0' };
	MPI_Comm newComm;
	MPI_Status status;

	MPI_Init(0, 0);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);

	if (size < 8) {
		printf("Test requires more than 8 processors\n");
		MPI_Abort(MPI_COMM_WORLD, __LINE__);
	}
	// get the World group
	MPI_Comm_group(MPI_COMM_WORLD, &worldGroup);

	// Create a new group into the World Group
	MPI_Group_incl(worldGroup, 3, ranks, &newGroup);

	// Check the size and ranks in the resulting group
	MPI_Group_size(newGroup, &newSize);
	MPI_Group_rank(newGroup, &newRank);

	printf("rank %d has newRank %d in newGroup, newSize is %d \n", rank, newRank, newSize);

	// Get communicator of the new group
	MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);

	// Broadcast in for new group only
	if (newRank == 0)
		strcpy(msg, "Hello");

	if (rank == ranks[0] || rank == ranks[1] || rank == ranks[2])
		MPI_Bcast(msg, 6, MPI_CHAR, 0, newComm);
	printf("rank = %d   after broadcast: msg = %s\n", rank, msg);

	if (newRank == 1) {
		int value = 123;
		MPI_Send(&value, 1, MPI_INT, 0, 0, newComm);
	}
	else if (newRank == 0){
		int data;
		MPI_Recv(&data, 1, MPI_INT, 1, 0, newComm, &status);
		printf("newRank 1 received %d from newRank 0\n", data);
	}
	MPI_Group_free(&newGroup);
	MPI_Finalize();
	return 0;
}
```

Split

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
 {


      int myid, numprocs;
   int color, newRank;
   MPI_Comm newComm;
   MPI_Init(&argc, &argv);

   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD ,&myid);

   color = myid % 2;

   MPI_Comm_split(MPI_COMM_WORLD, color, 0, &newComm);
   MPI_Comm_rank( newComm, &newRank);
   printf("myid = %d, color = %d     newRank = %d\n", myid, color, newRank);

   MPI_Finalize();

 }


      }
```