1. In given code “mpi-pi.c”, the jobs are distributed in cyclic manner. Modify and write the code that distributes the job using block partition, which assigns a set of consecutive jobs to each process. For example, if there are 40 jobs and 4 processes, distribute 10 jobs each process such as 1 to 10 to process 0, 11 to 20 to process 1, 21 to 30 to process 2, and 31 to 40 to process 3. For the simplicity, assume that user inputs the number of steps that is the multiple of the number of processes. For example, if the number of processes is 4, the number of steps should be multiple of 4 such as 16, 100, 200, 10000, etc. Your code should dynamically distribute the same amount of jobs over the number of available processes. Explain your job distribution method and it should be clearly documented in your code. (20 points)

Source Code:

**#include "mpi.h"**

**#include <stdio.h>**

**#include <math.h>**

**double f(a)**

**double a; {**

**return (4.0 / (1.0 + a\*a));**

**}**

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

**int i, n, myrank, nproc, done = 0;**

**double PI25DT = 3.141592653589793238462643;**

**double mypi, pi, h, sum, x;**

**int tasks = 0;**

**int startAt = 0;**

**int stopAt = 0;**

**MPI\_Init(&argc,&argv);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD,&nproc);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myrank);**

**n = 0;**

**while (!done)**

**{**

**if (myrank == 0)**

**{**

**printf("Enter the number of intervals: (0 quits)\n");**

**scanf("%d",&n);**

**}**

**MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); // printf("I am process**

**//%d and value n is %d\n", myid, n);**

**if (n == 0)**

**done = 1;**

**else**

**{**

**//get how many tasks each processors does**

**tasks = n / nproc;**

**//calculate where to start and stop at**

**startAt = (myrank \* tasks) + 1;**

**stopAt = (myrank + 1) \* tasks;**

**h = 1.0 / (double) n;**

**sum = 0.0;**

**printf("This processor is doing %d jobs and starts at %d and ends at %d\n", tasks, startAt, stopAt );**

**for (i = startAt; i <= stopAt; i++){**

**printf("I am processor %d doing my job %d\n", myrank, i);**

**x = h \* ((double)i - 0.5);**

**sum += f(x);**

**}**

**mypi = h \* sum;**

**MPI\_Reduce(&mypi, &pi, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);**

**if (myrank == 0)**

**printf("\npi is approximately %.16f, Error is %.16f\n\n", pi, fabs(pi**

**- PI25DT));**

**}**

**}**

**MPI\_Finalize();**

**return 0;**

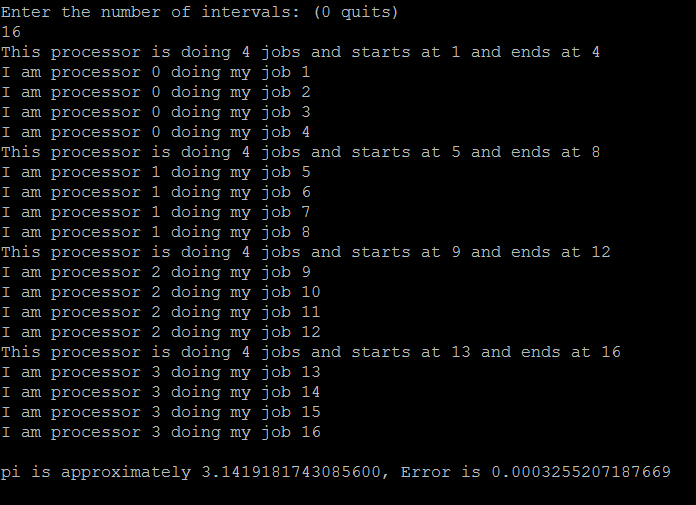
**}**

**Explanation:**

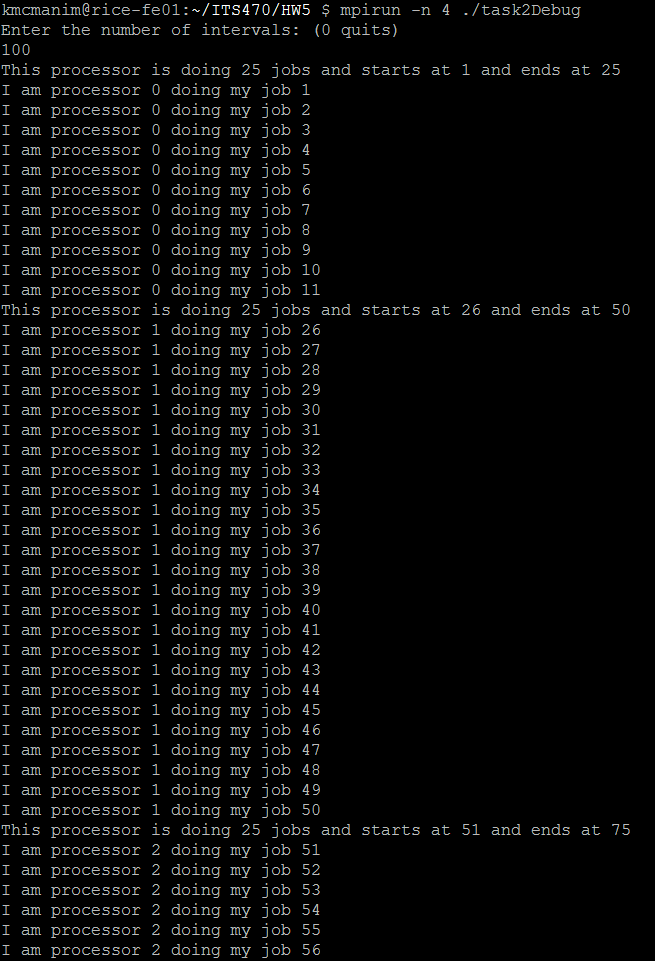
Now the above source code includes some debugging information to prove that it works correctly. It starts by calculating the amount of tasks that each processor should perform, it does this by taking the number of intervals “n” and then dividing it by the number of processors “nprocs”. After calculating the amount of tasks that each processor needs to run each processor then needs to calculate where to start and stop at. To do this I used a simple equation involving the myrank value which will be unique to each processor to calculate where the processor should start doing a loop and where to stop later on. I then modified the for loop so that it uses the start at and stop at values so that each processor only ran the block of intervals that it was meant to run. Inside the for loop I added which processor is doing what job to show what is going on and what processor is on what step.

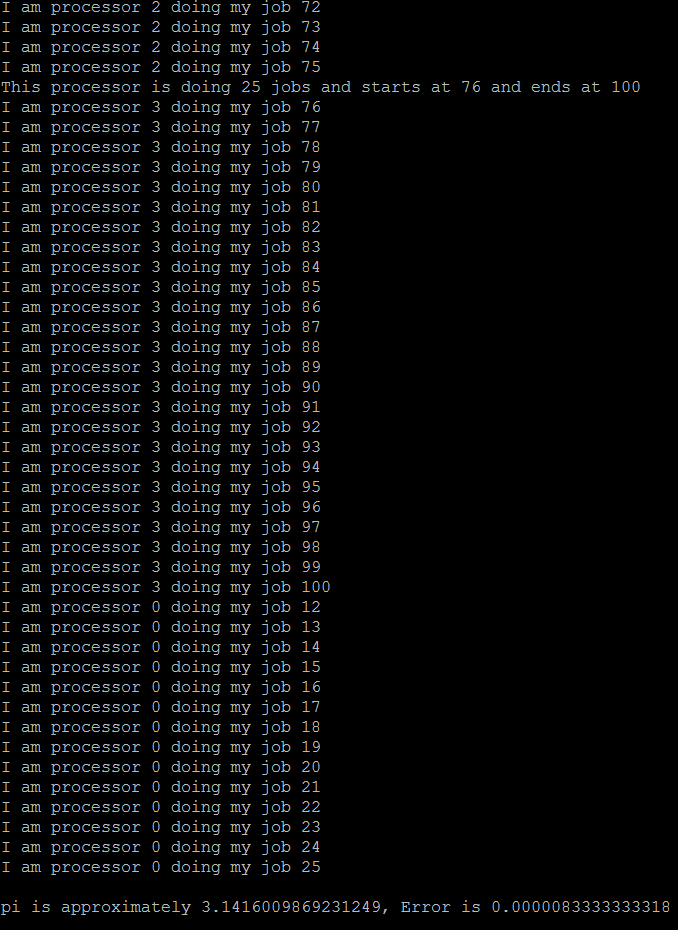
Here is output of it working using 4 processors and 3 different sizes for intervals:

For 16 intervals:

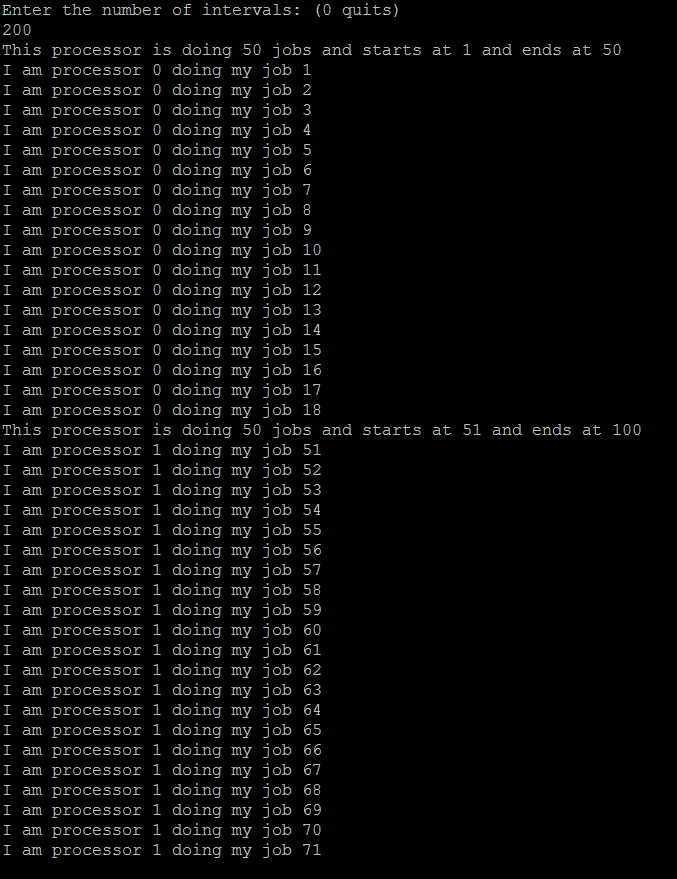


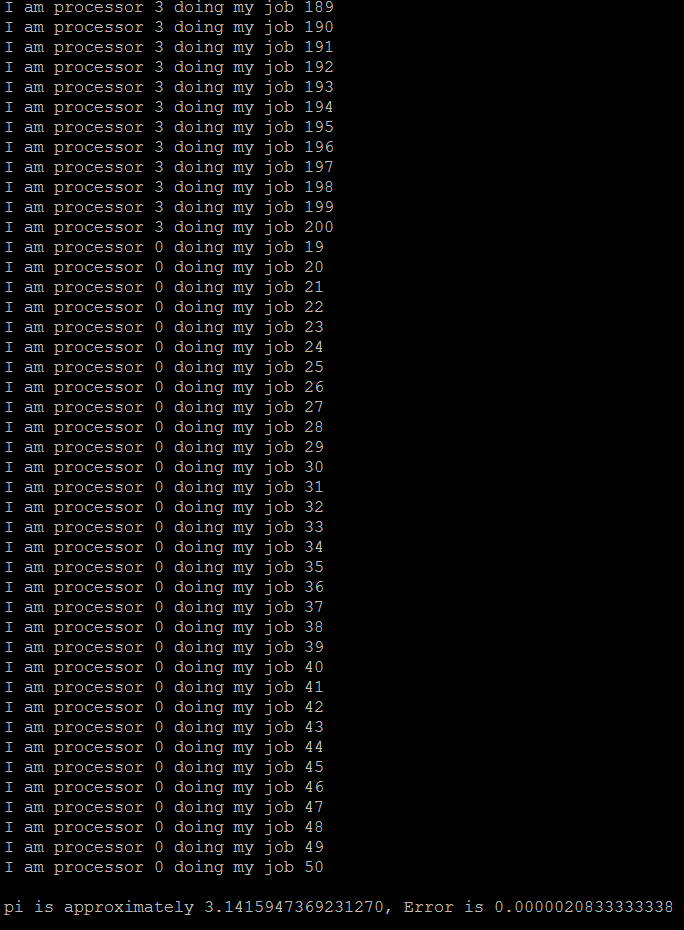
For 100 intervals:

and at the end:



For 200 intervals:

and at the end:



documentation in the code -5

1. Re-write pi estimation code using MPI\_Send and MPI\_Recv communication functions only. You can download sample code at course website called “mpi-pi.c”. Explain how you did in word here and it should be documented in your code. (Hint: Process 0 receives all partial local results and computes the estimating pi) (20 points)

Here is my source code:

**#include "mpi.h"**

**#include <stdio.h>**

**#include <math.h>**

**double f(a)**

**double a; {**

**return (4.0 / (1.0 + a\*a));**

**}**

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

**int i, n, myrank, nproc, done = 0;**

**double PI25DT = 3.141592653589793238462643;**

**double mypi, pi, h, sum, x, bufferzone;**

**MPI\_Init(&argc,&argv);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD,&nproc);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myrank);**

**n = 0;**

**while (!done)**

**{**

**if (myrank == 0)**

**{**

**printf("Enter the number of intervals: (0 quits)\n");**

**scanf("%d",&n);**

**}**

**//send n from 0 to all other processes**

**for(i = 1; i < nproc; i++){**

**if(myrank == 0)**

**MPI\_Send(&n, 1, MPI\_INT, i, i, MPI\_COMM\_WORLD);**

**}**

**//recieve n on all other processors from processor 0**

**if (myrank != 0)**

**MPI\_Recv(&n, 1, MPI\_INT, 0, myrank, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**printf("My rank is %d, and my value for n is %d\n", myrank, n);**

**//MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);**

**if (n == 0)**

**done = 1;**

**else**

**{**

**h = 1.0 / (double) n;**

**sum = 0.0;**

**for (i = myrank + 1; i <= n; i += nproc)**

**{**

**x = h \* ((double)i - 0.5);**

**sum += f(x);**

**}**

**mypi = h \* sum;**

**//printf("My rank is %d and mypi is %f and pi is %f\n", myrank, mypi, pi);**

**//MPI\_Reduce(&mypi, &pi, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);**

**//send mypi to processor 0 from all other processors**

**if (myrank != 0)**

**MPI\_Send(&mypi, 1, MPI\_DOUBLE, 0, nproc+myrank, MPI\_COMM\_WORLD);**

**if (myrank == 0){**

**//loop through and recieve from all other processors their mypi value and store it in a buffer, if not a different variable it deadlocks**

**for(i = 1; i < nproc; i++){**

**if(i == 1)**

**pi = mypi;**

**//printf("processor %d recieving on channel %d from processor %d \n", myrank, nproc+i, i);**

**MPI\_Recv(&bufferzone, 1, MPI\_DOUBLE, i, nproc+i, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**//printf("Adding %f to %f\n", bufferzone, pi);**

**pi += bufferzone; }**

**printf("\npi is approximately %.16f, Error is %.16f\n\n", pi, fabs(pi**

**- PI25DT));**

**}**

**}**

**}**

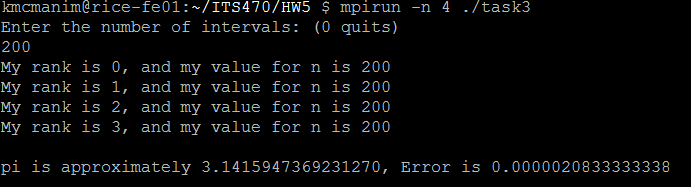
**MPI\_Finalize();**

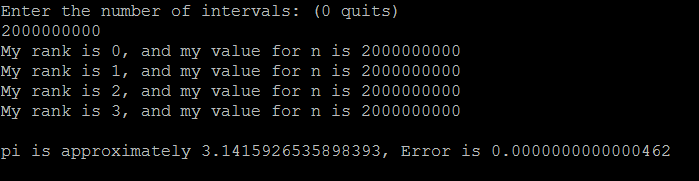
**return 0;**

**}**

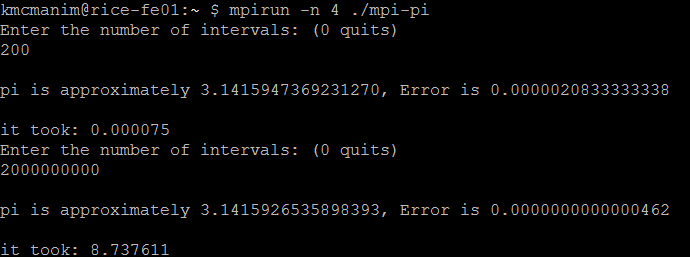
Now in my code I changed the broadcast to processor 0 sending on all other nodes the value of n and then I had all other processors receive what processor 0 sends. Then for reduce I had all processors but 0 send their mpi value to proc 0 using a channel different from the broadcast by nprocs length, I then had a loop in the rank 0 section which loops through all the other processors and if it is on the first processor to add processor 0’s value for mypi into the pi variable. I then after it added its own value for mypi into pi I had the processor 0 receive a new variable called bufferzone, because if it is the same variable the system deadlocks, with the mypi information sent from all the other processors. After receiving each other processors value it then adds it to pi, and then displays it after the loop.

Here some sample output that is the same pi value as if it was ran using broadcast and reduce:





Thes are the same results as question 4’s examples shown here:



1. Use given “mpi-pi.c” code, modify it to compute and print the elapsed time at process 0 using Wtime( ) function. Then, choose the large number of steps (i.e., n = 2,000,000,000) and run the program by use of 1, 4, 8, and 16 processes to find each corresponding elapsed time. Do this with the small number of steps (i.e., 200) to find the elapsed time at each number of processes. Plot both results to observe the relation between the number of processes and elapsed time at each number of steps. (40 points)

Here is my source code:

**#include "mpi.h"**

**#include <stdio.h>**

**#include <math.h>**

**double f(a)**

**double a;**

**{**

**return (4.0 / (1.0 + a\*a));**

**}**

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

**{**

**int i, n, myrank, nproc, done = 0;**

**double PI25DT = 3.141592653589793238462643;**

**double mypi, pi, h, sum, x, t1, t2;**

**MPI\_Init(&argc,&argv);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD,&nproc);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myrank);**

**n = 0;**

**while (!done)**

**{**

**if (myrank == 0)**

**{**

**printf("Enter the number of intervals: (0 quits)\n");**

**scanf("%d",&n);**

**t1 = MPI\_Wtime();**

**}**

**MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);**

**// printf("I am process %d and value n is %d\n", myid, n);**

**if (n == 0)**

**done = 1;**

**else**

**{**

**h = 1.0 / (double) n;**

**sum = 0.0;**

**for (i = myrank + 1; i <= n; i += nproc)**

**{**

**x = h \* ((double)i - 0.5);**

**sum += f(x);**

**}**

**mypi = h \* sum;**

**MPI\_Reduce(&mypi, &pi, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);**

**if (myrank == 0){**

**t2 = MPI\_Wtime();**

**printf("\npi is approximately %.16f, Error is %.16f\n\n", pi, fabs(pi - PI25DT));**

**printf("it took: %f\n", t2 - t1);**

**}**

**}**

**}**

**MPI\_Finalize();**

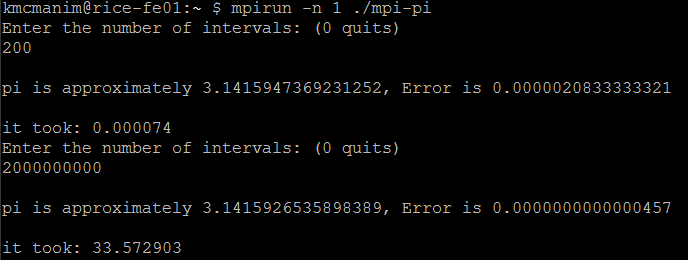
**return 0;**

**}**

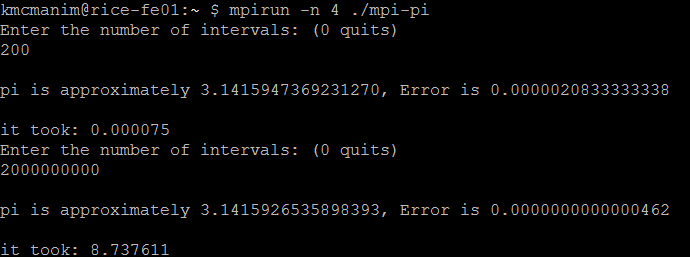
Now all I modified in the original code is that I added two double variables t1 and t2 which will be the different times for the calculation of time elapsed. Now when the user enters an interval t1 is set, then when the calculations are done t2 is set, and then a display after the calculation was added that shows t2 – t1 for the time in seconds for the calculation to complete.

Output:

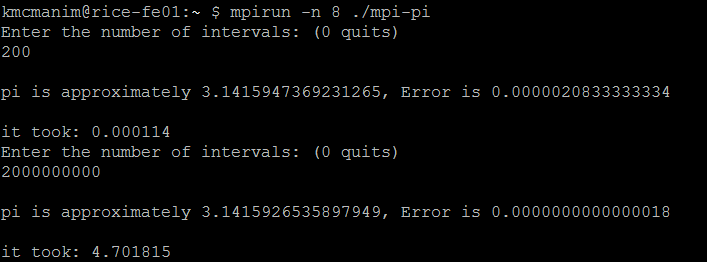
So using one processor and the intervals of 200 and 2000000000 you can see that for 200 it took .000074 seconds and 2 billion took 33.572903:



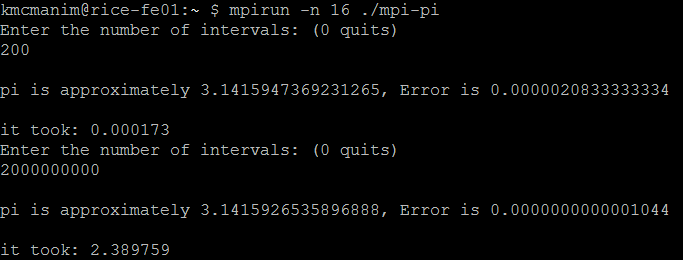
For four processors for 200 intervals it took 0.000075 and for 2 billion it took 8.737611:



For eight processors for 200 intervals it took 0.000114 and for 2 billion it took 4.701815:



For sixteen processors for 200 intervals it took 0.000173 and for 2 billion it took 2.389759 seconds:



So from what I can gather using a small amount of intervals has negative effects when using more and more processors, because with only one processor it was completed faster than 4, which was completed faster than 8, which was completed faster than 16 processors. But on the other hand, when using a large number of intervals there is a much shorter computation time the more processors you add. As I found when using 2 billion intervals, 16 nodes finished in just over 2 seconds, then 8 had 4.7 seconds, 4 had 8.7 seconds, and then only one node took 33.5 seconds to compute. This leads me to believe that when there is a small amount of data you want to use less processors so that less communication occurs which will shorten the amount of time to complete a task, but when using a large amount of data increasing the amount of processors will reduce the calculation time, most likely up to a certain point and then start to slowly increase the time from there.

Here is a chart with the data I gathered:

Here is them individually based:

As the charts show there is a positive relationship with a small set of data in relation to computation time vs number of processors where comp time goes up as processors do, but an inverse relationship when using a large set of data, where as more processors are added the time goes down, up to a certain point I’m sure.

1. Following information is given to create the graph and use them to draw the graph topology.

number of node = 4, index[4] = {2, 4, 6, 8}, edges[8] = {2, 3, 2, 3 , 0, 1, 0, 1}. (10 points)

Given information, following table can be found.

|  |  |  |  |
| --- | --- | --- | --- |
| Node | Neighbors | Index | Edges |
| 0 | 2 | 2 | 2, 3 |
| 1 | 2 | 4 | 2, 3 |
| 2 | 2 | 6 | 0, 1 |
| 3 | 2 | 8 | 0, 1 |

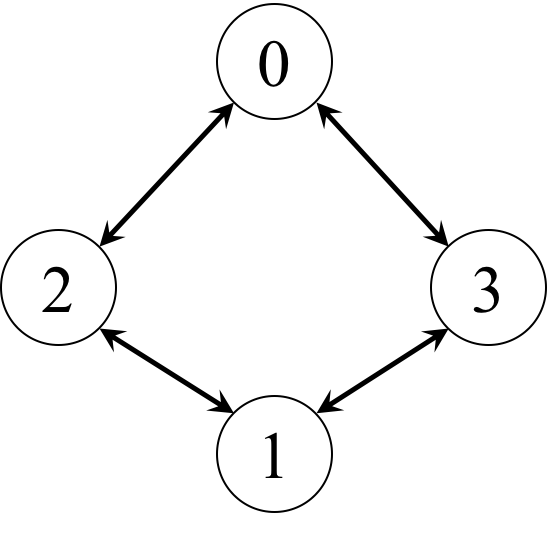
Node 0 has two neighbors and they are node 2 and 3.

Node 1 has two neighbors and they are node 2 and 3.

Node 2 has two neighbors and they are node 0 and 1.

Node 3 has two neighbors and they are node 0 and 1.

Therefore, the graph can be found as



1. Write the code using MPI\_Comm\_split routine to divide a group and create communicator such that:

all processes with even ranks are in one group

all processes with odd ranks are in the other group

maintain the reverse order by rank

Your output should display each process’s group number, rank from original communicator, new rank from new communicator. (10 points)

The sample code is shown below.

#include <mpi.h>

#include <stdio.h>

#include <unistd.h>

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

{

        int myid, nproc, new\_myid, new\_proc;

        int icolor, ikey;

        MPI\_Comm newcomm;

        MPI\_Init(&argc, &argv);

        MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);

        MPI\_Comm\_size(MPI\_COMM\_WORLD, &nproc);

        if (myid % 2 == 0)

        {

                icolor = 1;

                ikey = nproc - myid;

        } else {

                icolor = 2;

                ikey = nproc - myid;

        }

        MPI\_Comm\_split(MPI\_COMM\_WORLD, icolor, ikey, &newcomm);

        MPI\_Comm\_rank(newcomm, &new\_myid);

        MPI\_Comm\_size(newcomm, &new\_proc);

        printf("Group: %d    Original Rank: %d    New rank: %d\n", icolor, myid, new\_myid);

        MPI\_Finalize();

        return 0;

}

Sample output with 10 processes should look like

kim1347@rice-fe03:**~/its470/test/group** $ mpirun -n 10 ./mpi-split1

Group: 1    Original Rank: 0    New rank: 4

Group: 2    Original Rank: 1    New rank: 4

Group: 1    Original Rank: 2    New rank: 3

Group: 2    Original Rank: 3    New rank: 3

Group: 1    Original Rank: 4    New rank: 2

Group: 2    Original Rank: 5    New rank: 2

Group: 1    Original Rank: 6    New rank: 1

Group: 2    Original Rank: 7    New rank: 1

Group: 1    Original Rank: 8    New rank: 0

Group: 2    Original Rank: 9    New rank: 0