Lab 1 MPI

Table of Contents

1. Ring Around the Rosie

1.1. Solution Approach

First it gets checked if the program is run with at least two processes. Process 0 initializes the token to -1 and doesn't receive a value. The last process sends the token back to 0 which receives the token and the ring gets closed. Each time the token gets sent or received a log message gets printed.

1.2. Source Code

Listing 1. ring_around_the_rosie.c

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

int main(int argc, char** argv) {
  MPI_Init(NULL, NULL);

  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  if(world_size < 2) {
    printf("These program requires at least 2 processes but only %d provided.\n"
,world_size);
    MPI_Abort( MPI_COMM_WORLD , -1);
  }
  printf("Current Process %d!\n",world_rank);

  int token;
  if (world_rank == 0) {
    token = -1;
  } else {
    printf("[DEBUG]: P%d - Receiving\n",world_rank);
    MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    printf("[DEBUG]: P%d - Received token %d from P%d\n", world_rank, token,
world_rank - 1);
  }

  printf("[DEBUG]: P%d - Sending token\n", world_rank);
  MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size, 0, MPI_COMM_WORLD);
  printf("[DEBUG]: P%d - Sending done.\n", world_rank);


  if (world_rank == 0) {
    MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("[DEBUG]: P%d - Received token %d from P%d\n", world_rank, token,
world_size - 1);
    printf("Ring closed\n");
  }

  MPI_Finalize();
}
```

1.3. Result: What is the output of the program? Submit the output as part of your protocol.

1.3.1. Compile and Run

`mpicc ring_around_the_rosie.c -o ring_around_the_rosie`

`mpirun -n 5 ring_around_the_rosie`

```
Current Process 1!
[DEBUG]: P1 - Receiving
Current Process 3!
[DEBUG]: P3 - Receiving
Current Process 4!
[DEBUG]: P4 - Receiving
Current Process 0!
Current Process 2!
[DEBUG]: P2 - Receiving
[DEBUG]: P0 - Sending token
[DEBUG]: P0 - Sending done.
[DEBUG]: P1 - Received token -1 from P0
[DEBUG]: P1 - Sending token
[DEBUG]: P1 - Sending done.
[DEBUG]: P2 - Received token -1 from P1
[DEBUG]: P2 - Sending token
[DEBUG]: P3 - Received token -1 from P2
[DEBUG]: P3 - Sending token
[DEBUG]: P2 - Sending done.
[DEBUG]: P4 - Received token -1 from P3
[DEBUG]: P4 - Sending token
[DEBUG]: P4 - Sending done.
[DEBUG]: P3 - Sending done.
[DEBUG]: P0 - Received token -1 from P4
Ring closed
```

1.4. Run the program multiple times: Does the message order change?

Yes the order does indeed change because processes are executed in parallel and you don’t know which one gets executed first.

```
Current Process 2!
Current Process 1!
[DEBUG]: P1 - Receiving
Current Process 3!
[DEBUG]: P3 - Receiving
Current Process 4!
[DEBUG]: P4 - Receiving
[DEBUG]: P1 - Received token -1 from P0
[DEBUG]: P1 - Sending token
[DEBUG]: P1 - Sending done.
Current Process 0!
[DEBUG]: P0 - Sending token
[DEBUG]: P0 - Sending done.
[DEBUG]: P2 - Receiving
[DEBUG]: P2 - Received token -1 from P1
[DEBUG]: P2 - Sending token
[DEBUG]: P4 - Received token -1 from P3
[DEBUG]: P4 - Sending token
[DEBUG]: P4 - Sending done.
[DEBUG]: P3 - Received token -1 from P2
[DEBUG]: P3 - Sending token
[DEBUG]: P3 - Sending done.
[DEBUG]: P2 - Sending done.
[DEBUG]: P0 - Received token -1 from P4
Ring closed
```

```
Current Process 0!
[DEBUG]: P0 - Sending token
[DEBUG]: P0 - Sending done.
Current Process 2!
[DEBUG]: P2 - Receiving
Current Process 4!
[DEBUG]: P4 - Receiving
[DEBUG]: P2 - Received token -1 from P1
[DEBUG]: P2 - Sending token
[DEBUG]: P2 - Sending done.
Current Process 1!
[DEBUG]: P1 - Receiving
[DEBUG]: P1 - Received token -1 from P0
[DEBUG]: P1 - Sending token
[DEBUG]: P1 - Sending done.
Current Process 3!
[DEBUG]: P3 - Receiving
[DEBUG]: P3 - Received token -1 from P2
[DEBUG]: P3 - Sending token
[DEBUG]: P3 - Sending done.
[DEBUG]: P4 - Received token -1 from P3
[DEBUG]: P4 - Sending token
[DEBUG]: P4 - Sending done.
```

```
[DEBUG]: P0 - Received token -1 from P4
Ring closed
```

1.5. Explain the message order, and why it changes/does not change.

The order of the processes changes but the order of the transmission works stays the same. The order of sending and receving stays the same because the calls are blocking.

2. Counting Even Numbers

2.1. Solution Approach

Process 0 reads the data (Numbers and Size) and the other processes receive it and calculate the event number amount. After that they send the value to the process zero which calculates the sum and prints it after it receiving all even counts. Receiving and sending is blocking itself that's why in our case no barrier has to be used.

2.2. Source Code

Listing 2. count_even_numbers.c

```c
#include <mpi.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

void readData(char* fname, int lines,int** nums, int* lens) {
    int i = -1, j = 0;
    FILE* fp;
    char ch;
    int newLine=1;
    int val =0 ;

    const int DEBUG=0;

    if (fp = fopen(fname, "r")) {
        if(DEBUG) printf("Opened file\n");
        while (fscanf(fp, "%d%c", &val,&ch) != EOF) {
            if(DEBUG) printf("Read '%d' '%c'\n",val,ch);
            if(newLine==1) {
                i++;
                if(DEBUG) printf("New line found (I am now in line %d)...",i);
                nums[i]=(int*)malloc(sizeof(int)*val);
                lens[i]=val;
                if(DEBUG) printf("allocated array nums[%d] of size %d\n",i,val);
                j=0;
                newLine=0;
            } else {
                if(DEBUG) printf("Storing value %d at num[%d][%d]...",val,i,j);
                nums[i][j]=val;
                if(DEBUG) printf("done\n");
                if(ch=='\n') {
                    if(DEBUG) printf("Char was new line!\n\n");
                    newLine=1;
                }
                j++;
            }
        }
        fclose(fp);
    }
}

int countEvenNumbers(int* arrOfNumbers, int arrSize) {
  int count = 0;
  for (int i = 0; i < arrSize; i++) {
    if (arrOfNumbers[i] % 2 == 0) {
      count++;
    }
  }
  return count;
}

int main(int argc, char** argv) {
  const int FILE_LINES = 10;

    int evenCount = 0;

    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_size != FILE_LINES) {
        printf("This program requires exactly 10 processes (%d provided).\n",
world_size);
        MPI_Abort(MPI_COMM_WORLD, -1);
    }

    int** numbers = (int**)malloc(sizeof(int*) * FILE_LINES);
    int lines[FILE_LINES];

    if (world_rank == 0) {
        readData("input.txt", FILE_LINES, numbers, lines);

        for (int i = 1; i < world_size; i++) {
            MPI_Send(&lines[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            MPI_Send(numbers[i], lines[i], MPI_INT, i, 1, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(&lines[world_rank], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int* data = (int*)malloc(sizeof(int) * lines[world_rank]);

        MPI_Recv(data, lines[world_rank], MPI_INT, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
        evenCount = countEvenNumbers(data, lines[world_rank]);

        printf("P%d: Amount of even numbers is %d/%d\n", world_rank, evenCount, lines
[world_rank]);

        free(data);
    }

    // MPI_Barrier(MPI_COMM_WORLD);

    int totalEvenCount = 0;

    if (world_rank == 0) {
        totalEvenCount = countEvenNumbers(numbers[world_rank], lines[world_rank]);
        printf("P%d: Amount of even numbers is %d/%d\n", world_rank, totalEvenCount,
lines[world_rank]);
        for (int i = 1; i < world_size; i++) {
            MPI_Recv(&evenCount, 1, MPI_INT, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            totalEvenCount += evenCount;
        }

        printf("P%d: Total even count is %d\n", world_rank, totalEvenCount);
    } else {
        MPI_Send(&evenCount, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    }

    MPI_Finalize();

    return 0;
}
```

2.3. Result: Run the program with 10 processes (or rather, one process per line)

`mpirun -n 10 --oversubscribe count_even_numbers`

Info: oversubscribe was needed at least for my wsl instance

```
P2: Amount of even numbers is 3/3
P1: Amount of even numbers is 4/8
P4: Amount of even numbers is 5/5
P5: Amount of even numbers is 4/6
P3: Amount of even numbers is 4/8
P6: Amount of even numbers is 4/7
P0: Amount of even numbers is 3/7
P7: Amount of even numbers is 2/4
P8: Amount of even numbers is 2/4
P9: Amount of even numbers is 3/8
```

```
P0: Total even count is 34
```

2.4. Is the synchronization after counting the numbers necessary, or not? Explain your answer.

Syncronisation is in our case is not needed after counting because sending and receiving is blocking. That’s why MPI_Barrier(MPI_COMM_WORLD) is not necessary in this case.

2.5. Is there another (better?) way to distribute the array among processes? Explain your answer.

Yes there would be a better way with the MPI_Scatterv function. With the function a differing count of the data could be transmitted to each process.