Homework 2

Xinyuan Miao
020033910009
mxinyuan@sjtu.edu.cn

Exercise 3.17

Since we are only interested in finding the first occurrences of a particular substring (pattern string), it's not appropriate to divide the text into segments with the number equal to the number of the available processors. Alternatively, processors can fetch task in an interleaved fashion. Considering the possible case that the substring occur across to segments, we need to make sure that adjacent tasks should overlap to a certain length(at least $m-1$, m is the **length of the substring**).
For instance, if we have 2 processors, the processor-0 searches for the occurrence of the substring in position $[1, 2m-1]$ of the text, the processor-1 takes charge for $[m+1, 3m-1]$ and the second task for the processor-0 is to search for the matching in the interval $[2m+1, 4m-1]$...

Exercise 3.19

Given p processors, the list can be first divided into p parts of roughly equal size. Each processor process one primitive tasks that is to scan the part allocated to get and keep the largest 2 keys. In the communication stage, each processor sends the local results to other processors and receive the results from others as well. After checking the results both from the local and other processors, all the processors can get the global results, say the second-largest key in the whole list.

Exercise 4.2

1. add: 241
2. multiply: For the 8-bits unsigned decimal values, the result is 128 with overflow.
3. maximum: 99
4. minimum: 13
5. bitwise *or*: 127
6. bitwise *and*: 0
7. logical *or*: 1
8. logical *and*: 1

Exercise 4.8

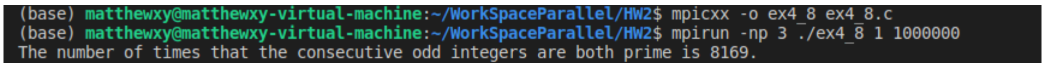
Main Idea

Intuitively we can split the list of number evenly to form multiple primitive tasks according to the number of the processors and do allocation. Each processor takes charge of one task to search for the number of times that the consecutive odd integers are both prime. Note that it is possible that one pair of consecutive odd integers crosses the task. Therefore, if one processor finds that the last odd integer of its range is prime, the processor should look one more step to check if there's another pair satisfying the condition.

Critical Code

```

#include <stdio.h>
#include <math.h>
#include <mpi.h>
#define TRUE 1
#define FALSE 0
#define BLOCK_LOW(id, p, n) ((id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1, p, n)-1)
int main(int argc, char* argv[]){
    int id;
    int p;
    int lb, rb; // left boundary and right boundary of the global range
    int low_value, high_value; // for local
    int local_count, global_count = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if (argc != 3){
        if(!id) printf("To ensure you have input valid interval!");
        MPI_Finalize();
        exit(1);
    }

    int process(int, int, int);
    int isPrime(int);

    lb = atoi(argv[1]);
    rb = atoi(argv[2]);
    /* allocate primitve task*/
    low_value = lb + BLOCK_LOW(id, p, rb-lb+1);
    high_value = lb + BLOCK_HIGH(id, p, rb-lb+1);

    local_count = process(low_value, high_value, rb);
    MPI_Reduce(&local_count, &global_count, 1,
               MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (!id){
        printf("The number of times that the consecutive odd integers are both prime is
            %d.\n", global_count);
    }
    fflush(stdout);
    MPI_Finalize();
    return 0;
}
int process(int low_value, int high_value, int rb){
    /* start from an odd number */
    int i = ((low_value%2)==1) ? low_value : (low_value+1);
    int count = 0;
    int preIsPrime = FALSE;

    while(i<=high_value){
        if (!isPrime(i)){
            preIsPrime = FALSE;
            i+=2;
            continue;
        }
        if (preIsPrime) count++;
        else preIsPrime = TRUE;
        i+=2;
    }
    if(i<=rb && isPrime(i) && preIsPrime)count++;
    return count;
}
```

Results



```
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpicxx -o ex4_8 ex4_8.c
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpirun -np 3 ./ex4_8 1 1000000
The number of times that the consecutive odd integers are both prime is 8169.
```

Figure 1: Execution result

Exercise 4.11

Main Idea

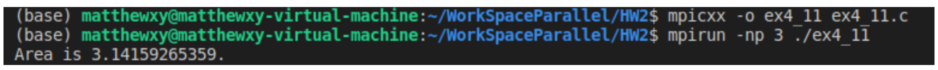
Given the rectangle rule to approximate the area under a curve, it is quite simple to assign tasks to multiple processors do the computation. Each processor computes the sum of the area of the rectangles in its range. After all this have been done, we just sum up the results from each processor and get the final reuslts.

Critical Code

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#define INTERVALS 1000000
```

```c
#define BLOCK_LOW(id, p, n) ((id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1, p, n)-1)

int main(int argc, char* argv[]){
    int id;
    int p;
    int low_value, high_value;
    double local_area, global_area = 0;
    double elapsed_time = 0.0;
    MPI_Init(&argc, &argv);
    MPI_Barrier (MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);

    double process(int, int, int);

    /* allocate primitve task*/
    low_value = BLOCK_LOW(id, p, INTERVALS);
    high_value = BLOCK_HIGH(id, p, INTERVALS);

    local_area = process(low_value, high_value, INTERVALS);
    MPI_Reduce(&local_area, &global_area, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    elapsed_time += MPI_Wtime();
    if (!id){
        printf("Area is %13.11lf.\nWith the help of %d processors, the total computation
            takes %.6lf seconds\n", global_area, p, elapsed_time);
    }
    fflush(stdout);
    MPI_Finalize();
    return 0;
}

double process(int low_value, int high_value, int total){
    double ysum = 0.0;
    double xi;
    int i;

    for(i=low_value; i<=high_value; i++){
        xi = (1.0/total)*(i+0.5);
        ysum+=4.0/(1.0+xi*xi);
    }
    return ysum/total;
}
```

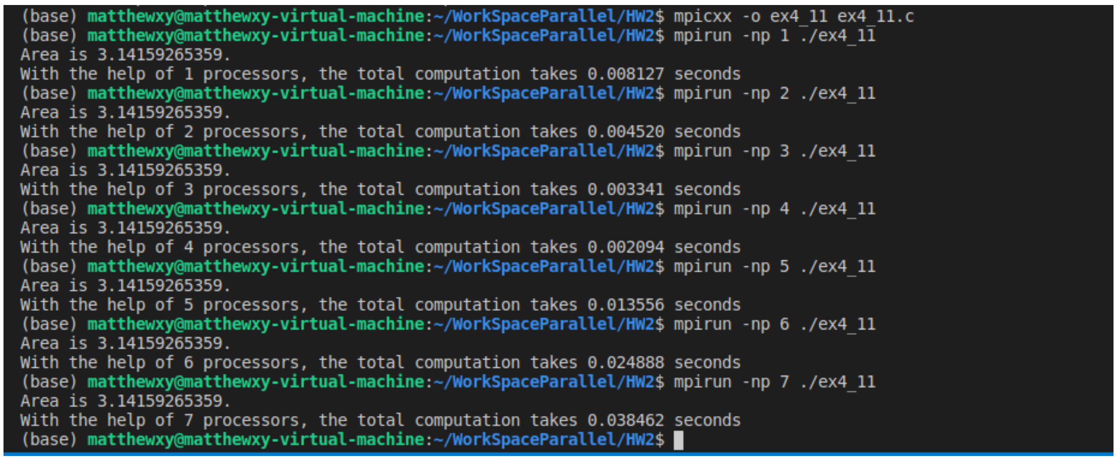
Results



```
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpicxx -o ex4_11 ex4_11.c
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpirun -np 3 ./ex4_11
Area is 3.14159265359.
```

Figure 2: Execution result

We also benchmark the program on various on various number of processors: From the Figure. 3,



```
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpicxx -o ex4_11 ex4_11.c
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpirun -np 1 ./ex4_11
Area is 3.14159265359.
With the help of 1 processors, the total computation takes 0.008127 seconds
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpirun -np 2 ./ex4_11
Area is 3.14159265359.
With the help of 2 processors, the total computation takes 0.004520 seconds
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpirun -np 3 ./ex4_11
Area is 3.14159265359.
With the help of 3 processors, the total computation takes 0.003341 seconds
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpirun -np 4 ./ex4_11
Area is 3.14159265359.
With the help of 4 processors, the total computation takes 0.002094 seconds
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpirun -np 5 ./ex4_11
Area is 3.14159265359.
With the help of 5 processors, the total computation takes 0.013556 seconds
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpirun -np 6 ./ex4_11
Area is 3.14159265359.
With the help of 6 processors, the total computation takes 0.024888 seconds
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$ mpirun -np 7 ./ex4_11
Area is 3.14159265359.
With the help of 7 processors, the total computation takes 0.038462 seconds
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW2$
```

Figure 3: Execution result

we can find that as the number of processors increases, computation time decreases first. However, after the number of processors is added up to 4, more processors did not bring more reduction to the time consumption but increased the computation time. I guess the reason is that the virtual machine was allocated only 4 cores, so when we 'force' the program to be executed in more than 4 threads, it actually cannot really do complete parallelism.