Homework 2 for Chapter 3-4

(120033910005)

March 2021

Problem 3.17

Partitioning

Suppose there are p processors. Let $p = 2^q$ The text string S is of length n. We divide S into p sub-strings. Each processor p_i perform single-core string matching algorithms on the p_ith sub-string.

Communication

When the pattern string matching with each sub-string. At each time step, They share the matching position in recursive form. If a processor found a matching sub-string, it send the start location to its adjacent node. If a processor didn't find a matching sub-string, it send NULL. For example, as is shown in Figure 1, there are 4 processors, where p_1 send information to p_2 and p_4 send information to p_3. p_3 send information to p_2.

For each time step, we just look at the 2^{q-1}th processor to determine if a matching substring is found. As soon as it found a match position. The program is terminated.

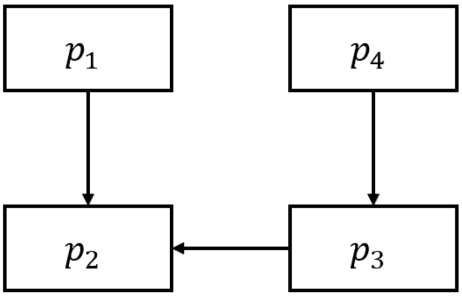

Figure 1: Communication

Problem 3.19

Partitioning

Suppose there are p processors. For each processor we just maintain two variables: the largest element L_1 and the second largest element L_2. Let $p = 2^q$. And we divide the input array into p sub-arrays. The p_ith processor is to find L_1 and L_2 of the pth sub-array.

Communication

Just follow ways of communication in Figure 1. When a processor receive information from other processors. It compare all the L_1 and L_2 and generate the new L_1 and L_2.

we just look at the 2^{q-1}th processor's L_2 and the result is determined.

Problem 4.2

add = 241, multiply = 128, maximum = 99, minimum = 13
bitwise_or = 127, bitwise_and = 0
logical_or = 1, logical_and = 1

Problem 4.8

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

#define MIN(a,b) ((a)<(b)?(a):(b))
#define BLOCK_LOW(id,p,n) ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n) - 1)
#define BLOCK_SIZE(id,p,n) (BLOCK_LOW((id)+1,p,n)-BLOCK_LOW(id,p,n))
#define BLOCK_OWNER(index,p,n) (((p)*((index)+1)-1)/(n))

int main (int argc, char *argv[]) {
    int count; /* Local prime count */
    int consecutive_count; /* Consecutive local prime count */
    double elapsed_time; /* Parallel execution time */
    int first; /* Index of first multiple */
    int global_count; /* Global prime count */
    int global_consecutive_count; /* Global consecutive prime count */
    int high_value; /* Highest value on this proc */
    int i;
    int id; /* Process ID number */
    int index; /* Index of current prime */
    int low_value; /* Lowest value on this proc */
    char *marked; /* Portion of 2,...,'n' */
    int n; /* Sieving from 2, ..., 'n' */
    int p; /* Number of processes */
    int proc0_size; /* Size of proc 0's subarray */
    int prime; /* Current prime */
    int size; /* Elements in 'marked' */

    MPI_Init (&argc, &argv);
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    if (argc != 2)
    {
        if (!id)
        printf ("Command line: %s <m>\n", argv[0]);
        MPI_Finalize();
        exit (1);
    }

    n = atoi(argv[1]);
    low_value = 2 + BLOCK_LOW(id,p,n-1);
    high_value = 2 + BLOCK_HIGH(id,p,n-1);
    size = BLOCK_SIZE(id, p, n-1);
```

```

    /* Bail out if all the primes used for sieving are
    not all held by process 0 */
    proc0_size = (n-1)/p;
    if ((2 + proc0_size) < (int) sqrt((double) n)) {
        if (!id)
            printf ("Too many processes\n");
        MPI_Finalize();
        exit (1);
    }

    marked = (char *) malloc (size + 2);
    if (marked == NULL) {
        printf ("Cannot allocate enough memory\n");
        MPI_Finalize();
        exit (1);
    }
    for (i = 0; i < size + 2; i++) marked[i] = 0;
    if (!id) index = 0;
    prime = 2;
    do
    {
        if (prime * prime > low_value)
            first = prime * prime - low_value;
        else {
            if (!(low_value % prime)) first = 0;
            else first = prime - (low_value % prime);
        }
        /* check primes */
        for (i = first; i < size + 2; i += prime) marked[i] = 1;
        if (!id) {
            while (marked[++index]);
            prime = index + 2;
        }
        MPI_Bcast (&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);

    } while (prime * prime <= n);
    count = 0;
    consecutive_count = 0;
    for (i = 0; i < size; i++)
    {
        if (!marked[i]) count++;
        if (!marked[i] && !marked[i+2]) consecutive_count++;
    }


    MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM, 0,
    MPI_COMM_WORLD);
    MPI_Reduce (&consecutive_count, &global_consecutive_count, 1,
    MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    elapsed_time += MPI_Wtime();
    if (!id) {
        printf ("%d primes are less than or equal to %d\n",
        global_count, n);
        printf ("among which, %d primes are consecutive\n",
        global_consecutive_count);
```

```
        printf ("Total elapsed time: %10.6f\n", elapsed_time);
    }
    MPI_Finalize ();
    return 0;
}
```

The answer is

```
78498 primes are less than or equal to 1000000
among which, 8169 primes are consecutive
Total elapsed time:   0.001776
```

Problem 4.11

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

/* This program computes pi using the rectangle rule. */
#define INTERVALS 1000000


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

int main (int argc, char *argv[])
{
    double area; /* Area under curve */
    double global_area; /* Global area under curve */
    double ysum; /* Sum of rectangle heights */
    double xi; /* Midpoint of interval */
    int i;
    int id; /* Process ID number */
    int p; /* Number of processes */
    int low_value; /* Lowest value on this proc */
    int high_value; /* Highest value on this proc */
    int size; /* Elements in 'marked' */
    double elapsed_time; /* Parallel execution time */

    MPI_Init (&argc, &argv);
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    low_value = BLOCK_LOW(id,p,INTERVALS);
    high_value = BLOCK_HIGH(id,p,INTERVALS);
    size = BLOCK_SIZE(id,p,INTERVALS);
    ysum = 0.0;
    for (i = low_value; i <= high_value; i++) {
        xi = (1.0/INTERVALS)*(i+0.5);
        ysum += 4.0/(1.0+xi*xi);
    }
```

```
    area = ysum * (1.0 / INTERVALS);

    // printf("process No.%d, low_value=%d, high_value=%d, size=%d,
    // area = %10.6f\n",
    //         id, low_value, high_value, size, area);
    MPI_Reduce (&area, &global_area, 1, MPI_DOUBLE, MPI_SUM, 0,
   MPI_COMM_WORLD);
    elapsed_time += MPI_Wtime();
    if (!id)
    {
        printf ("Area is %13.11f\n", global_area);
        printf ("Total elapsed time: %10.6f\n", elapsed_time);
    }
    MPI_Finalize();
    return 0;
}
```

Benchmarking is shown in Figure 2

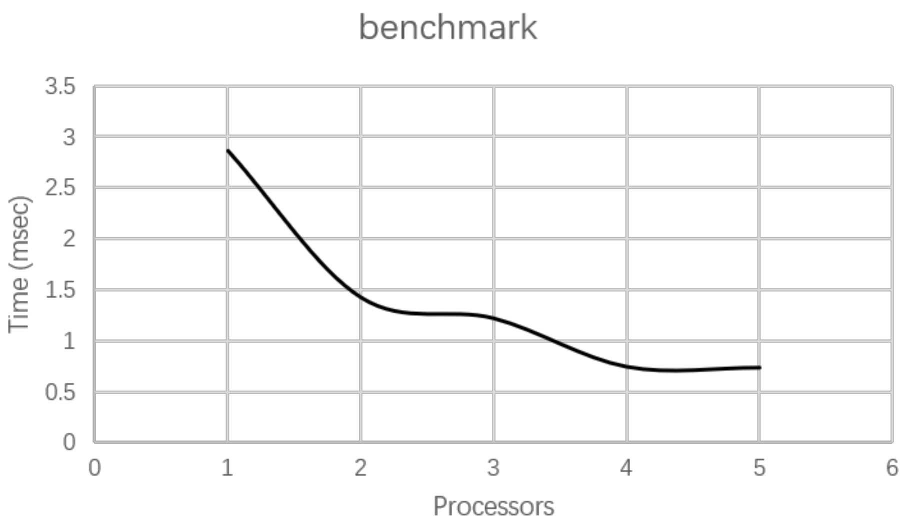

Figure 2: Benchmarking