Homework 3 for Chapter 5-7

120033910005

April 16, 2021

Problem 5.9

For each process, we first calculate all the primes up to $\sqrt{n}$. And we allocate the each prime with the process in turn. For a process got a prime, it marks all mutiples of this prime as "not prime". In the end we use process 0 to calculate the number of all the primes up to n.

```
for (i = 2; i <= (int)(sqrt(limit)); ++i){
    if (is_prime[i])
        for(j = 2*i; j <= limit; j+=i)
            is_prime[j] = 0;
}

count = 0;
for (i = 2; i <= limit; ++i){
    if(is_prime[i]) {
        count++;
        if(count % p == id) {
            printf("process %d catches %d\n", id, i);
            for (j = 2*i; j <= n; j+=i)
                marked[j] = 1;
        }
    }
}
MPI_Reduce(marked, global_marked, n+1, MPI_CHAR, MPI_LOR, 0, MPI_COMM_WORLD);
// MPI_Bcast(&count, 1, MPI_INT, id, MPI_COMM_WORLD);
elapsed_time += MPI_Wtime();
if (!id) {
    count = 0;
    for (i = 2; i <= n; i++) {
        if (!global_marked[i]) count++;
    }
    printf ("%d primes are less than or equal to %d\n", count, n);
    printf ("Total elapsed time: %10.6f\n", elapsed_time);
}
```

And the result is shown below

```
chenzixuan@LAPTOP-947PNMTJ:/mnt/c/Users/ChenZixuan/Desktop/Parallel-Programming-Project/hw3$
mpirun -np 3 ./eratosthenes
 1000
168 primes are less than or equal to 1000
Total elapsed time:   0.000818
```

Problem 6.10

We define our own MPI_Bcast in the following rules. First if the current process is root process, it sends the buffer to all the other process using MPI_Send. If it is not, call MPI_Recv to receive new buffer information

from root process.

```
int my_MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
                 int root, MPI_Comm comm){
    int p, id;
    MPI_Status status;
    MPI_Comm_size (comm, &p);
    MPI_Comm_rank (comm, &id);
    if (id == root){
        for (int i = 0; i < p; ++i)
            if (i != id)
                MPI_Send(buffer, count, datatype, i, 0, comm);
    }
    else{
        MPI_Recv(buffer, count, datatype, root, 0, comm, &status);
    }
    return 0;
}
```

we test our algorithm by synchronizing certain amout of arrays. And the result is shown below

```
chenzixuan@LAPTOP-947PNMTJ:/mnt/c/Users/ChenZixuan/Desktop/Parallel-Programming-Project/hw3$
mpirun -np 3 ./my_MPI_Bcast 100000
Using my_MPI_Bcast: process 0 is initialized with value = 0
Using my_MPI_Bcast: process 1 is initialized with value = 1
Using my_MPI_Bcast: process 2 is initialized with value = 2
Using my_MPI_Bcast: process 1 updated with value = 0, time passed: 0.000708
Using MPI_Bcast: process 1 is re-initialized with value = 1
Using my_MPI_Bcast: process 0 updated with value = 0, time passed: 0.001026
Using my_MPI_Bcast: process 2 updated with value = 0, time passed: 0.001027
Using MPI_Bcast: process 2 is re-initialized with value = 2
Using MPI_Bcast: process 0 is re-initialized with value = 0
Using MPI_Bcast: process 2 updated with value = 0, time passed: 0.000343
Using MPI_Bcast: process 0 updated with value = 0, time passed: 0.000161
Using MPI_Bcast: process 1 updated with value = 0, time passed: 0.000724
```

Problem 6.13

We divide the cell state A into p pieces, with each piece A_i maintained by a process. In the actual algorithm, each process maintain a full copy of A and the counting matrix C. Each process is responsible for calculating the living cell count around each cell of A_i, and update new cell state of A_i.

```
int count_grid(const int * grid, int i, int m, int n)
{
    int count;
    int x = i/m, y = i%m;
    int xx, yy;
    int ii, jj;

    count = 0;
    for(ii = -1; ii <= 1; ++ii)
        for(jj = -1; jj <=1; ++jj){
            xx = x + ii;
            yy = y + jj;
            if(xx>=0 && xx<m && yy>=0 && yy<n)
                count += grid[xx*n+yy];
        }
    count -= grid[i];
```

```
    //printf("pos = %d, x = %d, y = %d, count = %d\n", i, x, y, count);
    return count;
}

int main (int argc, char *argv[]) {
    double elapsed_time; /* Parallel execution time */
    int i, j, m, n, p, k;
    int id; /* Process ID number */
    int* cell_state;
    int* count;
    MPI_Status status;
    int x, y;
    int low, high, size;
    int c;


    FILE* fp = fopen(FILE_NAME, "r");
    if (fp == NULL || fscanf(fp, "%d %d", &m, &n) == EOF)
        printf("ERROR: read from file %s.\n", FILE_NAME);
    if (m > 0 && n >0){
        cell_state = (int*)malloc(m*n*sizeof(int));
        count = (int*)malloc(m*n*sizeof(int));
    }
    else
        printf("ERROR: m = %d, n = %d\n", m, n);
    for(x = 0; x < m; ++x)
        for(y = 0; y < n; ++y)
            fscanf(fp, "%d", &cell_state[x*n+y]);
    fclose(fp);

    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 (p > m*n/2)
    {
        if(!id)
            printf("too many processes\n");
        MPI_Finalize();
        exit (1);
    }
    if (argc != 3)
    {
        if (!id)
        printf ("Command line: %s <m>\n", argv[0]);
        MPI_Finalize();
        exit (1);
    }
    j = atoi(argv[1]);
    k = atoi(argv[2]);

    low = BLOCK_LOW(id, p, m*n);
    high = BLOCK_HIGH(id, p, m*n);
    size = BLOCK_SIZE(id, p, m*n);
    // printf("low = %d, high = %d, size = %d\n", low, high, size);
```

```
    for(i = 0; i < j; ++i)
    {
        MPI_Bcast(cell_state, m*n, MPI_INT, 0, MPI_COMM_WORLD);
        if (!id && (i+1) % k == 0){
            printf("cell state at iteration %d:\n", i+1);
            for(x = 0; x < m; ++x){
                for(y = 0; y < n; ++y)
                    printf("%d ", cell_state[x*n+y]);
                printf("\n");
            }
        }
        for (c = low; c < high; ++c)
            count[c] = count_grid(cell_state, c, m, n);

        for (c = low; c < high; ++c){
            if(!cell_state[c] && count[c] == 3)
                cell_state[c] = 1;
            else if (cell_state[c] && (count[c] < 2 || count[c] > 3))
                cell_state[c] = 0;
        }
        if(id)
            MPI_Send(cell_state+low, size, MPI_INT, 0, 0, MPI_COMM_WORLD);
        else
            {
            for (int pid = 1; pid < p; ++pid)
                MPI_Recv(cell_state+BLOCK_LOW(pid, p, m*n),
                        BLOCK_SIZE(pid, p, m*n),
                        MPI_INT,
                        pid,
                        0,
                        MPI_COMM_WORLD,
                        &status);
            }
    }

    MPI_Finalize ();
    return 0;
}
```

The result is shown below:

```
\begin{verbatim}
chenzixuan@LAPTOP-947PNMTJ:/mnt/c/Users/ChenZixuan/Desktop/Parallel-Programming-Project/hw
mpirun -np 3 ./cellular_aut
omaton 4 1
cell state at iteration 0:
1 0 0 1 1
0 0 0 1 1
0 0 0 0 1
1 1 1 0 1
cell state at iteration 1:
0 0 0 1 0
0 0 0 0 1
1 0 0 1 0
1 0 1 0 0
cell state at iteration 2:
0 0 0 0 0
```

```
0 0 1 0 1
0 1 0 1 0
0 0 0 0 0
cell state at iteration 3:
0 0 0 0 0
0 1 1 0 0
1 1 0 0 0
0 0 0 0 0
\end{verbatim}
```

Problem 7.11

For Guatafson-Barsis's Law

$$\psi(n,p) \leq s + (1-s)p$$

Since $s = \frac{\sigma(n)}{\sigma(n)+\phi(n)/p}$ is an implicit function of p. It increases as p increases. Actually the overall speed up still converges.