Homework 2

PANGBO

3.18

Assume there are p processors.

(1) Divide array to p subarrays:
- Let $r = n \mod p$
- First r subarrays have $\lceil n/p \rceil$ elements
- The rest $p - r$ subarrays have $\lfloor n/p \rfloor$ elements

(2) Each processor i scans its subarray, record the larget element m_i and the second largest element m_i'.

(3) Agglomerate m_i and m_i' from all processors to processor 0, then we get a array of $2p$ elements.

(4) Scan the new array to get the second largest element.

4.8

The source code is presented as 4_8.cpp.

We first split N into p intervals, each processor count primes in its interval.

```
int start = rank * (1000000 / size) + 1;
int end = (rank + 1) * (1000000 / size) + 1;

if (rank == size - 1) {
    end = 1000000;
}

for (int i = start; i < end; i += 2) {
    if (isPrime(i)) {
        if (isPrime(i + 2)) count++;
        else i += 2;
    }
}
```

Then we use MPI_Reduce to sum up the number of primes in each interval.

Author's address: pangbo.

Manuscript submitted to ACM

```
int total_count;
MPI_Reduce(&count, &total_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
    printf("Total count: %d\n", total_count);
}
```

The snapshot of the output is shown in Figure 1.

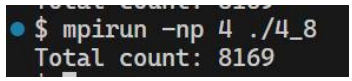

Fig. 1. Snapshot of the output of 4.8

5.7

The original source code is presented as 5_7.cpp, and the modified source code is presented as 5_7_opt.cpp.

Firstly, every processor will find all the prime numbers between 3 and $\sqrt{N}$.

```
sqrt_n = sqrt(n);
size = sqrt_n - 1;
marked = new char[size / 2];
prime = 3;
do
{
    first = prime * prime - 3;

    for (i = first; i < size; i += prime*2)
        marked[i/2] = 1;

    while (marked[++index]);
    prime = index *2 + 3;

} while (prime * prime <= sqrt_n);
```

Then those primes are used to mark the table. Moreover, to reduce memory comsumption, we only store odd numbers in the table. That is, when we want to mark x, we actually mark $\lfloor x/2 \rfloor$.

```
while ((prime = base_primes[j]) != -1)
{
    if (prime * prime > low_value)
    {
        first = prime * prime - low_value;
```

```
        }
        else
        {
            if (!(low_value % prime))
                first = 0;
            else
                first = prime - (low_value % prime);
            if ((low_value + first) % 2 == 0) first += prime;
        }
        for (i = first; i < size; i += prime*2){
            marked[i/2] = 1;
        }
        j++;
    }
```

The snapshot of the output is shown in Figure 2.

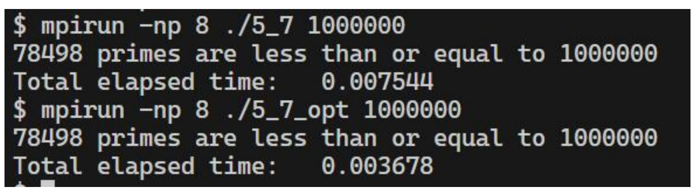

Fig. 2. Snapshot of the output of 5.7

After optimization, the time consumption is reduced by about 50%.

6.9

The source code is presented as 6_9.cpp.

Instead of using ‘MPT_Reduce’, we will use MPI_Recv and MPI_Send to implement the reduce function. All processors will send their data to the root processor, and the root processor will sum up all the data.

```
    void reduce(const int *sendbuf, int *recvbuf, int count, int root) {
        int rank, size;
        int *tempbuf = new int[count];
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if(rank == root) {
            for (int i = 0; i < count; i++) {
                recvbuf[i] = 0;
            }
```

```
            for(int i = 0; i < size; i++) {
                if(i != root) {
                    MPI_Recv(tempbuf, count, MPI_INT, i,
                        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    for (int j = 0; j < count; j++) {
                        recvbuf[j] += tempbuf[j];
                    }
                } else {
                    for (int j = 0; j < count; j++) {
                        recvbuf[j] += sendbuf[j];
                    }
                }
            }
        } else {
            MPI_Send(sendbuf, count, MPI_INT, root, 0, MPI_COMM_WORLD);
        }
        delete[] tempbuf;
    }
```

The snapshot of the output is shown in Figure 3.

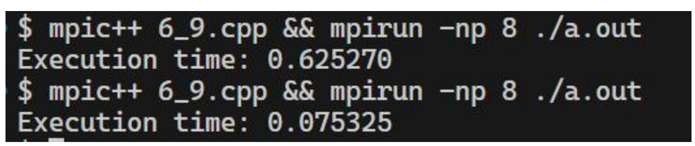

Fig. 3. Snapshot of the output of 6.9

‘MPT_Reduce’ is used in the first run, and my own implementation is used in the second run. My implementation is faster than ‘MPT_Reduce’, I guess it is because ‘MPT_Reduce’ is implemented in a more general way, and my implementation is more specific.

7.5

With Amdahl’s Law

$$10 = \frac{1}{(1 - 0.94) + \frac{0.94}{N}}$$
$$N = 23.5$$

So at least 24 processors are needed to achieve a speedup of 10.

7.8

The fraction of time spent in the parallel computation performing inherently sequential operations:

$$s = \frac{9}{242} = 0.037$$

Number of processors p is 16.

$$\Psi = p + s(1 - p) = 15.442$$

So the scaled speedup is 15.442.