

Homework 2

PANGBO

3.18

Assume there are p processors.

- (1) Divide array to p subarrays:
 - Let $r = n \bmod p$
 - 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 largest 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Association for Computing Machinery.

Manuscript submitted to ACM

Manuscript submitted to ACM

```

53     int total_count;
54     MPI_Reduce(&count, &total_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
55
56
57     if (rank == 0) {
58         printf("Total_count: %d\n", total_count);
59     }
60 }

```

The snapshot of the output is shown in Figure 1.

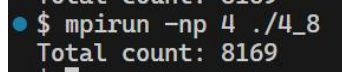


Fig. 1. Snapshot of the output of 4.8

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} .

```

74     sqrt_n = sqrt(n);
75     size = sqrt_n - 1;
76     marked = new char[size / 2];
77     prime = 3;
78     do
79     {
80
81         first = prime * prime - 3;
82
83
84         for (i = first; i < size; i += prime*2)
85             marked[i/2] = 1;
86
87
88         while (marked[++index]);
89         prime = index * 2 + 3;
90
91     } while (prime * prime <= sqrt_n);
92
93
94

```

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

```

97     while ((prime = base_primes[j]) != -1)
98     {
99
100         if (prime * prime > low_value)
101         {
102             first = prime * prime - low_value;
103

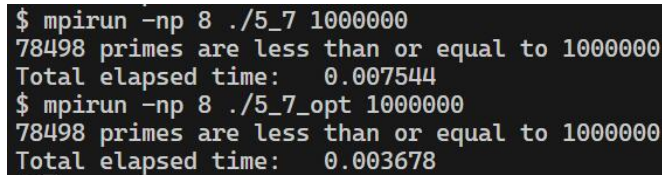
```

```

105     }
106     else
107     {
108         if (!(low_value % prime))
109             first = 0;
110         else
111             first = prime - (low_value % prime);
112         if ((low_value + first) % 2 == 0) first += prime;
113     }
114     for (i = first; i < size; i += prime*2){
115         marked[i/2] = 1;
116     }
117     j++;
118 }
119
120
121
122
123

```

The snapshot of the output is shown in Figure 2.



```

$ mpirun -np 8 ./5_7 1000000
78498 primes are less than or equal to 1000000
Total elapsed time: 0.007544
$ mpirun -np 8 ./5_7_opt 1000000
78498 primes are less than or equal to 1000000
Total elapsed time: 0.003678

```

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.

```

144 void reduce(const int *sendbuf, int *recvbuf, int count, int root) {
145     int rank, size;
146     int *tempbuf = new int[count];
147     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
148     MPI_Comm_size(MPI_COMM_WORLD, &size);
149     if(rank == root) {
150         for (int i = 0; i < count; i++) {
151             recvbuf[i] = 0;
152         }
153     }
154 }
155
156

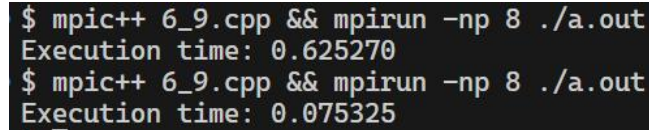
```

```

157     for(int i = 0; i < size; i++) {
158         if(i != root) {
159             MPI_Recv(tempbuf, count, MPI_INT, i,
160                     0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
161             for (int j = 0; j < count; j++) {
162                 recvbuf[j] += tempbuf[j];
163             }
164         } else {
165             for (int j = 0; j < count; j++) {
166                 recvbuf[j] += sendbuf[j];
167             }
168         }
169     }
170     MPI_Send(sendbuf, count, MPI_INT, root, 0, MPI_COMM_WORLD);
171 }
172 delete[] tempbuf;
173 }
174 }
175 }
176 }
177 }
178 }
179 }
180 }
181 }

```

The snapshot of the output is shown in Figure 3.



```

$ mpic++ 6_9.cpp && mpirun -np 8 ./a.out
Execution time: 0.625270
$ mpic++ 6_9.cpp && mpirun -np 8 ./a.out
Execution time: 0.075325

```

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.