

Homework 3

PANGBO

8.11

The source code is presented as [8_11.cpp](#).

To transpose a matrix, we first gather the matrix to a single processor, then scatter the matrix to all processors.

For processors other than the root processor, they send the data to the root processor, and then receive the data from the root processor.

```
MPI_Gatherv(buffer, buffer_size, MPI_INT, NULL,
            NULL, NULL, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatterv(NULL, NULL, NULL, MPI_INT, buffer,
            buffer_size, MPI_INT, 0, MPI_COMM_WORLD);
```

For the root processor, it need to calculate the displacement and block size for each processor, and then gather the data from all processors and scatter the data to all processors.

```
int *matrix_buffer = new int[N * N];
int *displs = new int[p];
int *receive_counts = new int[p];
for (int i = 0; i < p; i++) {
    displs[i] = BLOCK_LOW(i, p, N) * N;
    receive_counts[i] = BLOCK_SIZE(i, p, N) * N;
}
MPI_Gatherv(buffer, buffer_size, MPI_INT, matrix_buffer,
            receive_counts, displs, MPI_INT, 0, MPI_COMM_WORLD);
for (int i = 0; i < N; i++) {
    for (int j = i + 1; j < N; j++) {
        int temp = matrix_buffer[i * N + j];
        matrix_buffer[i * N + j] = matrix_buffer[j * N + i];
        matrix_buffer[j * N + i] = temp;
    }
}
```

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 MPI_Scatterv(matrix_buffer, receive_counts, displs, MPI_INT,
54             buffer, buffer_size, MPI_INT, 0, MPI_COMM_WORLD);
55
56 delete[] matrix_buffer;
57 delete[] displs;
58 delete[] receive_counts;
59

```

The snapshot of the result is shown in Figure 1. As we can see, the matrix is transposed correctly.

```

62 $ mpiexec -np 4 ./8_11 4
63 id = 0, buffer =
64 0 1 2 3
65 id = 1, buffer =
66 4 5 6 7
67 id = 2, buffer =
68 8 9 10 11
69 id = 3, buffer =
70 12 13 14 15
71 =====
72 id = 0, buffer =
73 0 4 8 12
74 id = 1, buffer =
75 1 5 9 13
76 id = 2, buffer =
77 2 6 10 14
78 id = 3, buffer =
79 3 7 11 15

```

Fig. 1. Snapshot of the result of 8.11

9.8

The source code is presented as 9_8.cpp.

For each worker, it will receive x from the manager, calculate $f(x)$, and send the result x, y back to the manager.

```

81 void worker()
82 {
83
84     double buffer[2];
85     while(true)
86     {
87         MPI_Recv(&buffer, 1, MPI_DOUBLE, 0, 0,
88                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
89         if (buffer[0] < 0) break;
90         buffer[1] = f(buffer[0]);
91         MPI_Send(&buffer, 2, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
92     }
93 }
94

```

For the manager, it will split the interval $[a, b]$ into p subintervals, and send these $p - 1$ split points to the workers. Then, it will receive the results from the workers and check for the new interval.

```

100 void manager(int p)
101 {
102
103     double low = 0.0;

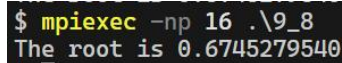
```

```

105  double high = 1.0;
106  while (high - low > 1e-11)
107  {
108
109      double interval = (high - low) / p;
110      for (int i = 1; i < p; ++i)
111      {
112          double x = low + i * interval;
113          MPI_Send(&x, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
114      }
115      for (int i = 1; i < p; ++i)
116      {
117          double buffer[2];
118          MPI_Recv(&buffer, 2, MPI_DOUBLE, i, 0,
119                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
120          if (buffer[1] > 0) high = MIN(high, buffer[0]);
121          else low = MAX(low, buffer[0]);
122      }
123  }
124  // terminate workers
125  for (int i = 1; i < p; ++i)
126  {
127      double x = -1;
128      MPI_Send(&x, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
129  }
130  printf("The root is %.10f\n", (low + high) / 2);
131 }

```

The snapshot of the result is shown in Figure 2.



```

$ mpiexec -np 16 ./9_8
The root is 0.6745279540

```

Fig. 2. Snapshot of the result of 9.8

11.4

(1) The computational time of each iteration is

$$\chi \frac{n}{p} \frac{n}{p} = \chi \frac{n^3}{p^2}$$

The communication time of each iteration is

$$\lambda + \frac{n \frac{n}{p}}{\beta} = \lambda + \frac{n^2}{p\beta}$$

When the communication time is less than the computational time, we have

$$\lambda + \frac{n^2}{p\beta} < \chi \frac{n^3}{p^2}$$

the solution is $n > 1072.2$, so n should be at least 1073.

(2) The computational time of each iteration is

$$\chi \frac{n^3}{\sqrt{p}} = \chi \frac{n^3}{p^{3/2}}$$

The communication time of each iteration is

$$2 \left(\lambda + \frac{\frac{n^2}{\sqrt{p}}}{\beta} \right) = 2\lambda + \frac{2n^2}{p\beta}$$

When the communication time is less than the computational time, we have

$$2\lambda + \frac{2n^2}{p\beta} < \chi \frac{n^3}{p^{3/2}}$$

the solution is $n > 544.1$, so n should be at least 545.

14.9

(a)

The algorithm is presented as Algorithm 1. We split the data into p partitions, and each processor sorts its own partition. Then, we merge the partitions in a binary tree fashion. The communication flow is shown in Figure 3.

Algorithm 1: Parallel Merge Sort

Input: Number of processors p , Rank of current processor $rank$

Output: Sorted array A

Receive own partition of data A ;

SORT(A);

$s \leftarrow 1$;

while $2^s \leq p$ **do**

if $rank \bmod 2^s = 0$ **then**

 Receive B from $rank + 2^{s-1}$;

$A \leftarrow \text{MERGE}(A, B)$;

$s \leftarrow s + 1$;

else

 Send A to $rank - 2^{s-1}$;

 Break;

end

end

if $rank = 0$ **then**

return A ;

end

(b)

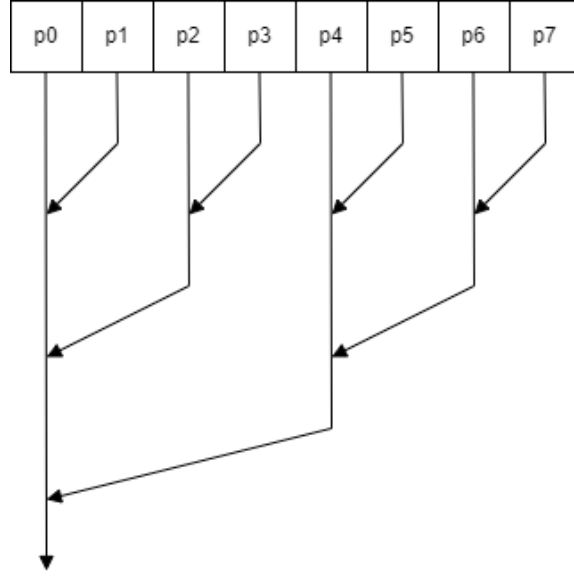


Fig. 3. Communication flow of parallel merge sort

Firstly, every processor needs to merge its own partition locally, the time complexity to sort n/p elements is $\Theta((n/p) \log(n/p))$. The total time complexity is $\Theta(n \log(n/p))$.

Then, denote $p = 2^k$, k merge is needed. For the i -th merge, there are 2^{k-i} working processors and each merges $\frac{2^i n}{p}$ elements. So the time complexity of inter-processor merging stage is

$$\begin{aligned} \sum_{i=1}^k 2^{k-i} \Theta\left(\frac{2^i n}{p}\right) &= \Theta\left(\sum_{i=1}^k n\right) \\ &= \Theta(nk) \\ &= \Theta(n \log p) \end{aligned}$$

The time complexity of this algorithm is $\Theta(n \log p) + \Theta(n \log(n/p)) = \Theta(n \log n)$.

(c)

If we only use one processor, the time complexity is $\Theta(n \log n)$.

If we use p processors, the processor 0 will do the most work, and the other processors will do less work. The computation time of processor 0 is $\Theta\left(\frac{n}{p} \log \frac{n}{p} + \sum_{i=1}^{\log p} \frac{2^i n}{p}\right) = \Theta\left(\frac{n}{p} \log \frac{n}{p} + n\right)$. The communication time of processor 0 is $\Theta\left(\sum_{i=1}^{\log p} \frac{2^{i-1} n}{p}\right) = \Theta(n)$.

Thus, $T(n, 1) = \Theta(n \log n)$, $T(n, p) = \Theta\left(\frac{n}{p} \log \frac{n}{p} + n\right)$, and $T_0(n, p) = p \Theta\left(\frac{n}{p} \log \frac{n}{p} + n\right) - \Theta(n \log n) = \Theta(np - n \log p)$. So, the isoefficiency function is given by

$$n \log n \geq C(np - n \log p) \Rightarrow n \geq \frac{e^{pC}}{p^C}$$

(d)

The source code is presented as [14_9.cpp](#).

In each iteration, if the processor already finished all its work, it will send its sorted subarray to another processor and break the loop.

```

int target_id = id >> aggregate_size << aggregate_size;
MPI_Send(&local_buffer_size, 1, MPI_INT, target_id, 0, MPI_COMM_WORLD);
MPI_Send(sorted_buffer, local_buffer_size,
         MPI_INT, target_id, 0, MPI_COMM_WORLD);
delete [] sorted_buffer;
break;

```

Otherwise, a processor will receive the sorted subarray from another processor and merge them.

```

int *local_part = sorted_buffer;
sorted_buffer = NULL;
int source_id = id + (0x1 << (aggregate_size - 1));
int receive_buffer_size;
MPI_Recv(&receive_buffer_size, 1, MPI_INT, source_id,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
receive_buffer = new int[receive_buffer_size];
MPI_Recv(receive_buffer, receive_buffer_size, MPI_INT,
        source_id, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
sorted_buffer = new int[local_buffer_size + receive_buffer_size];
merge(local_part, local_buffer_size, receive_buffer,
      receive_buffer_size, sorted_buffer);
delete [] local_part;
delete [] receive_buffer;
local_buffer_size += receive_buffer_size;
aggregate_size++;

```

The benchmark results are shown in Table 1. As we can see, for a small number of elements, the communication time is much larger than the computation time, so the parallel version is slower than the serial version. However, as the number of elements increases, the parallel version becomes faster than the serial version.

Table 1. Benchmark results for various combinations of p and n (in seconds)

$\begin{matrix} \backslash & p \\ n \end{matrix}$	1	2	4	8
100	0.000022	0.000100	0.000385	0.001196
10,000	0.001697	0.000865	0.001110	0.001705
1,000,000	0.186246	0.105908	0.058725	0.045571