ETH zürich

High Performance Computing for Science and Engineering I

P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2018

Set 8 - MPI: Blocking Communication

Issued: November 16, 2018
Hand in (optional): November 26, 2018 23:59

Question 1: Implementing a distributed reduction

45 points total

In this question, you will use MPI to calculate the following sum:

$$x_{\text{tot}} = \sum_{n=1}^{N} n = 1 + 2 + 3 + \ldots + (N-1) + N \tag{1}$$

a) Fill in the missing part in the Makefile in order to compile the skeleton code with MPI support.

   Points: 2 for correct Makefile.

b) Validation of HPC code is an important subject. For example, there is an analytic formula for the above sum. Use this to check if your implementation is correct.
   To this end, implement the function `exact(N)`.

   Hint: A young C.F. Gauss found the formula in elementary school.

   Analytic solution:

$$x_{\text{tot}} = \sum_{n=1}^{N} n = \frac{N(N+1)}{2} \tag{2}$$

   This formula can be proofed by induction (not necessary to get points):

   $\underline{N=1:}$

$$x_{\text{tot}} = 1 = \frac{1(1+1)}{2} \tag{3}$$

$\underline{N-1 \to N}$:

$$\begin{aligned} x_{\text{tot}} &= \sum_{n=1}^{N} n \\ &= N + \sum_{n=1}^{N-1} n \\ &\overset{(*)}{=} N + \frac{(N-1)N}{2} \\ &= N + \frac{1}{2}\left(N^2 - N\right) \\ &= \frac{1}{2}\left(N^2 + N\right) \\ &= \frac{N(N+1)}{2} \end{aligned} \tag{4}$$

In (*), the induction hypothesis for $N-1$ was used.

Points:

- 2 points for correct formula
- 2 points for correct implementation

c) Initialize and finalize MPI by filling the corresponding gaps in the skeleton code.

See solution code.

Points:

- 1 point for calling `MPI_Reduce`
- 1 point for correct MPI datatype `MPI_LONG`
- 1 point for correct reduction operation `MPI_SUM`
- 1 point for sending all the data to rank 0
  (other reduction targets are also fine as long as in the end, only one rank has all the data and the receiving rank also prints the result)
- 1 point for using the correct communicator `MPI_COMM_WORLD`

d) Each rank performs only a part of the sum. Distribute the work load reasonably in order to guarantee load balancing. Each rank should calculate the subsum

$$\text{sum}_{\text{rank}} = N_{\text{start}} + (N_{\text{start}} + 1) + \ldots + N_{\text{end}}, \tag{5}$$

where N_{start} and N_{end} are the corresponding variables in the skeleton file.

See solution code.

Points:

- 2 points for correctly determining the start index
  Note: Starting at zero is also correct, but implies that rank 0 sums one element less than the other ranks if the other code in the solution is not changed accordingly (see solution code). However, in this exercise, this does not represent much of an imbalance.
- 2 points for correctly determining the end index of all ranks

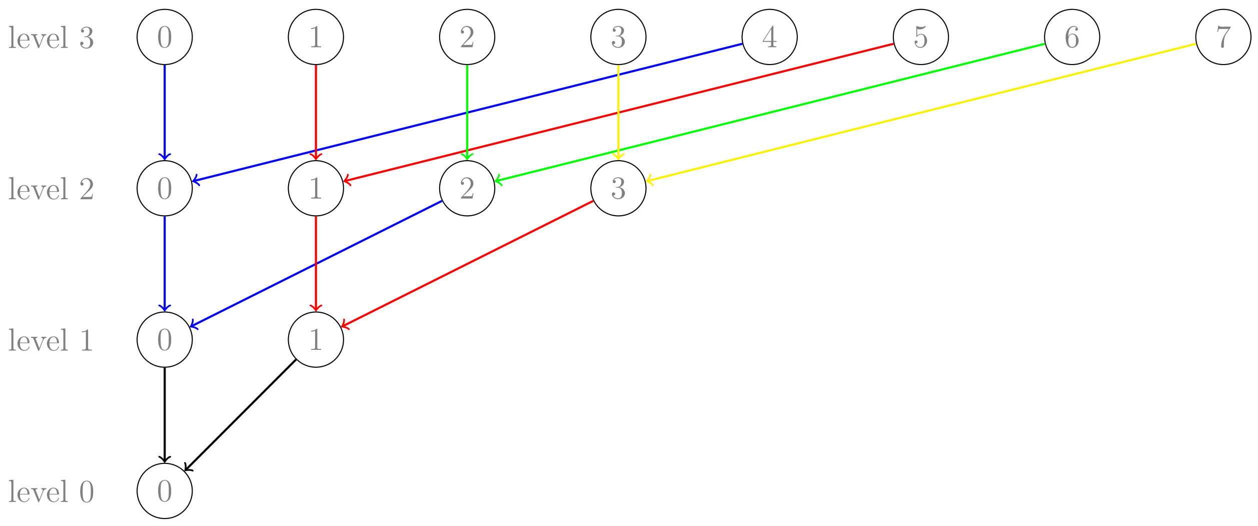

Figure 1: The communication pattern of a tree-like reduction. Each circle represents a rank, the number inside is the rank ID. Communication takes place along the arrows.

- **2 points for taking care of the case where N `mod size` $\neq 0$**

e) Finally, implement your own reduction. This can be done in a tree-like way as depicted in Fig. 1. Your task is to implement this scheme for the special case that the number of ranks is a power of 2, i. e.

$$|\text{ranks}| = 2^l, l \in \mathbb{N}_0 \tag{6}$$

See solution code.

Points:

- **20 points for performing the communication "in levels", as shown in Fig. 1.**
  - **7 points for correct separation between sending and receiving ranks in each level**
  - **7 points for correctly determining the communication partner in each level**
  - **4 points for correct usage of `MPI_Send` and `MPI_Recv`**
  - **2 points if only the receiving rank performs the reduction operation (addition)**
- **5 points for testing the implementation at least with 2, 4, 8, 16, 32 ranks.**

f) What is the advantage of this scheme compared to the naive reduction? Name 2 advantages and quickly justify your answer.
Hint: In the naive approach, every rank sends its elements directly to the master. The master then reduces all obtained elements by repeatedly applying the operation, in our case the sum.

- More ranks communicate with each other. Depending on the network topology, this means that there is potentially more bandwidth available.
- The reduction operation can be performed by many processes in parallel, leading to better load balance and faster execution of the reduction. In the naive version, every

process has to wait until the root of the reduction has performed the reduction operation on all items received by another rank. Also, only one process can send at each time, leaving the others idle and wasting an opportunity to do something useful.

Points:
1.5 points per each pair of advantage and discussion (3 points in total)
No points if discussion is missing!

Question 2: MPI Bug Hunt

16 points total

Find the bug(s) in the following MPI code snippets and find a way to fix the problem!

a)
```
1  const int N = 10000;
2  double* result = new double[N];
3  // do a very computationally expensive calculation
4  // ...
5
6  // write the result to a file
7  std::ofstream file("result.txt");
8
9  for(int i = 0; i <= N; ++i){
10     file << result[i] << std::endl;
11 }
12
13 delete[] result;
```

- There is a segfault hidden in line 9.
  It can be fixed by changing `i <= N` to `i < N`.
- All ranks write simultaneously to the same output file! This is a problem for many reasons:
  On one hand, it leads to an incorrect output file because of many concurrent writes that overwrite each other.
  On the other hand, the same work is done many times. One remedy would be to let only the root rank write the output. This can be implemented as shown in the code below.

```
const int N = 10000;
double* result = new double[N];
// do a very computationally expensive calculation
// ...

// write the result to a file
if(rank == 0){
    std::ofstream file("result.txt");

    for(int i = 0; i <= N; ++i){
        file << result[i] << std::endl;
    }
}

delete[] result;
```

Of course, this assumes that every rank has the same data. If that is not the case, one would have to send all the information to the root rank.
Later in the course, you will learn to to use MPI to perform I/O (input/output) operations involving many ranks in parallel.

Points:

- **1.25 points for each identified bug**
- **1.25 points per fixed bug**
- **-2.5 points if another bug is introduced through the "improvement"**
- **you cannot have less than zero points in this subquestion**

b)

```
// only 2 ranks: 0, 1
double important_value;

// obtain the important value
// ...

// exchange the value
if(rank == 0)
    MPI_Send(&important_value, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
else
    MPI_Send(&important_value, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD);

MPI_Recv(
    &important_value, 1, MPI_INT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE
);

// do other work
```

- The MPI type in the receive call (`MPI_INT`) does not match the MPI type that is sent (`MPI_DOUBLE`). Change it to `MPI_DOUBLE` to ensure defined behaviour.
- This code will deadlock: Both ranks send first and receive later. Neither of the two ranks can return from the send call because the call blocks until the sending is completed. But the sending cannot complete because the corresponding receive is not called yet. Deadlock!
  Possible solution:

```
// only 2 ranks: 0, 1
double important_value;

// obtain the important value
// ...

// exchange the value
if(rank == 0){
    MPI_Send(&important_value, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
    MPI_Recv(
        &important_value, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );
}else{
    MPI_Recv(
        &important_value, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );
    MPI_Send(&important_value, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD);
}
    // do other work
```

Points:

- 1.25 points for feach identified bug
- 1.25 points per fixed bug
- -2.5 points if another bug is introduced through the "improvement"
- you cannot have less than zero points in this subquestion

c) What is the output of the following program when run with 1 rank? What if there are 2 ranks? Will the program complete for any number of ranks?

```
MPI_Init(&argc, &argv);

int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int bval;
if (0 == rank)
{
    bval = rank;
    MPI_Bcast(&bval, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
else
{
    MPI_Status stat;
    MPI_Recv(&bval, 1, MPI_INT, 0, rank, MPI_COMM_WORLD, &stat);
}

cout << "[" << rank << "] " << bval << endl;

MPI_Finalize();
return 0;
```

For only 1 rank, everything is fine and the output of the program is:

```
[0] 0
```

With 2 ranks, however, there will be a deadlock:
Rank 0 arrives at the broadcast. This is a blocking *collective* operation, that means rank 0 waits until all other ranks in the communicator `MPI_COMM_WORLD` have reached this point and performed the collective operation. But rank 1 never arrives at this point! Instead, it gets stuck in the receive operation.

Points:

- **2 points for predicting correct behaviour with one rank**
- **2 point for predicting correct behaviour with two ranks**
- **2 points for justifying the behaviour of two ranks**