Review: Exercise Sheet 08

HPCSE I
30.11.2018
Renato Bellotti

Question 1: Implement a distributed reduction

Question 1: Implementing a distributed reduction

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.

```
CXX=mpic++
CXXFLAGS+=-std=c++11 -Wall -Wpedantic -O3
```

Question 1: Implement a distributed reduction

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:

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

Question 1: Implement a distributed reduction

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

```
// Initialize MPI
MPI_Init(&argc, &argv);

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

```
// Finalize MPI
MPI_Finalize();
```

Question 1: Implement a distributed reduction

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

$$\text{sum}_{\text{rank}} = N_{\text{start}} + (N_{\text{start}} + 1) + \ldots + N_{\text{end}}, \tag{2}$$

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

Question 1: Implement a distributed reduction

```
// -------------------------
// Perform the local sum:
// -------------------------
long sum = 0;

// Determine work load per rank
long N_per_rank = N / size;

// Remark: Start at 1!!!
long N_start = rank * N_per_rank + 1;
long N_end = (rank+1) * N_per_rank;
// the last rank has to do some more additions if size does not divide N
if(rank == size-1){
    N_end += N % size;
}

// N_start + (N_start+1) + ... + (N_start+N_per_rank-1)
for(long i = N_start; i <= N_end; ++i){
    sum += i;
}
```

Question 1: Implement a distributed reduction

Using MPI blocking collectives:

```
void reduce_mpi(const int rank, long& sum){
    if(rank == 0){
        MPI_Reduce(MPI_IN_PLACE, &sum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
    }else{
        MPI_Reduce(&sum, &sum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
    }
}
```

Question 1: Implement a distributed reduction

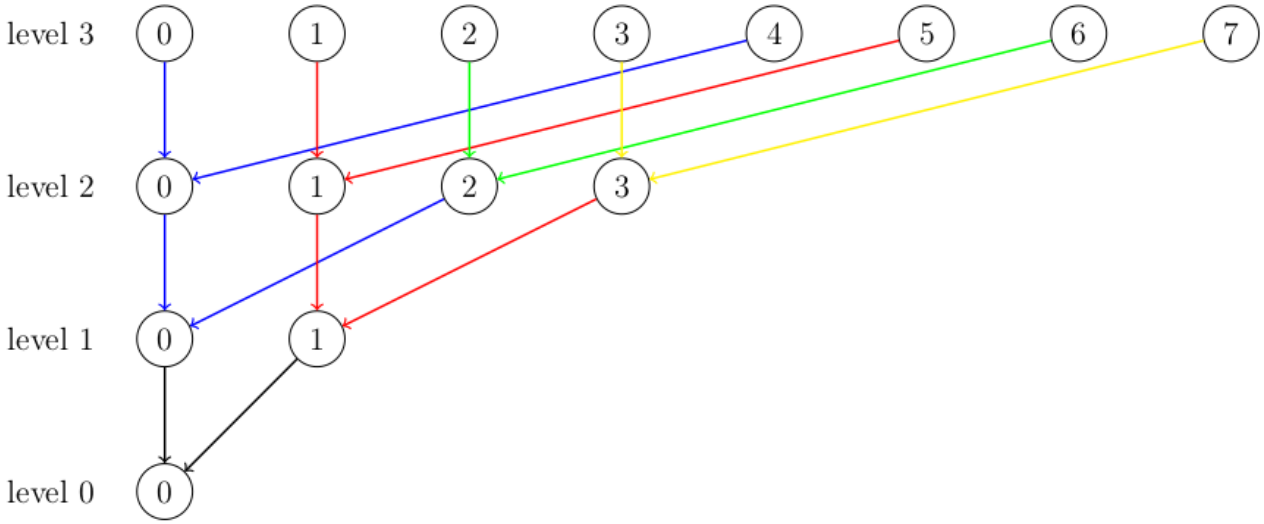
Be careful:

- **Collective operations:**
  Broadcast, gather, reduce, …
- **Point-to-point communication:**
  Send, Recv, …

Question 1: Implement a distributed reduction

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.

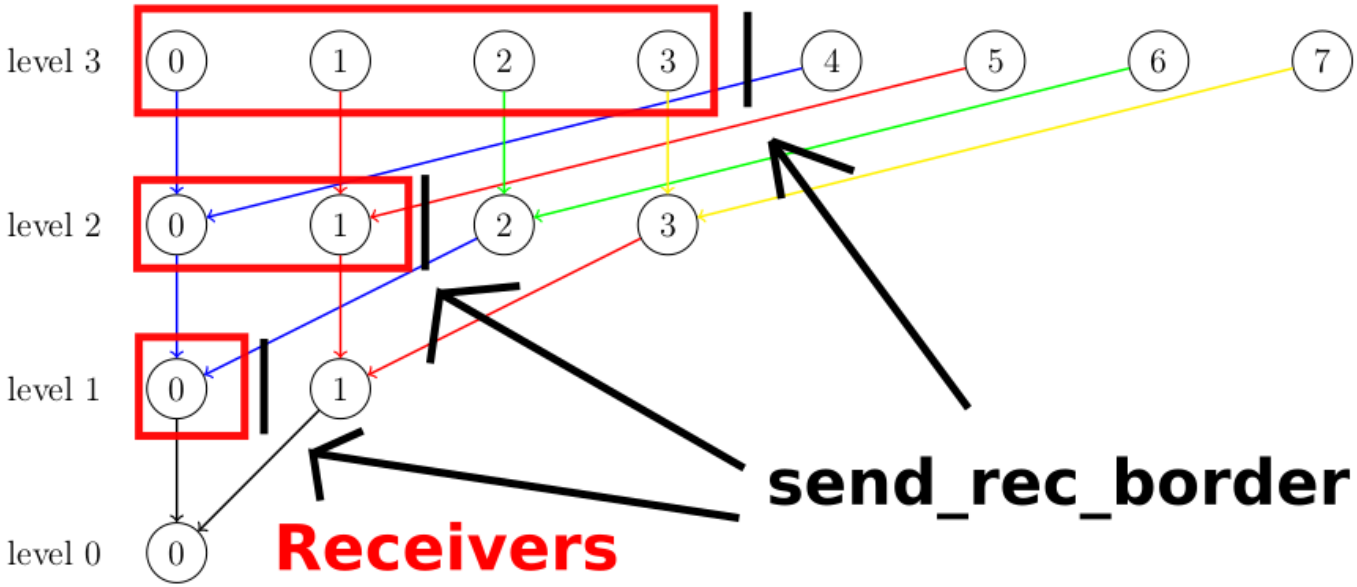
$$|\text{ranks}| = 2^l, l \in \mathbb{N}_0 \tag{3}$$



Question 1: Implement a distributed reduction

Idea:

In each level, the first half of the processes receive, the second half sends



Question 1: Implement a distributed reduction

```
// PRE: size is a power of 2 for simplicity
void reduce_manual(int rank, int size, long& sum){
    const int TAG = 1337;

    for(int send_rec_border = size/2; send_rec_border >= 1; send_rec_border/=2)
    {
        if(rank < send_rec_border)
        {
            long buffer;
            MPI_Recv(&buffer, 1, MPI_LONG, rank+send_rec_border, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            sum += buffer;
        }else{
            MPI_Send(&sum, 1, MPI_LONG, rank-send_rec_border, TAG, MPI_COMM_WORLD);
        }
    }
}
```

Question 1: Implement a distributed reduction

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:**
  Potentially higher bandwidth (but NOT less communication!)
- **Reduction operation is performed in parallel:**
  Better load balancing & faster execution because of less time wasted waiting for communication
  Naive: **O(n)**
  Tree-based: **O(log n)**

Question 1: General remark I

- **The basis is not needed in asymptotic complexities:**
  O(log n) instead of O(log_2 n)

- **Don't forget to run your MPI code with *mpiexec*!**

Question 1: General remark II

```
void reduce_mpi(const int rank, long& sum){
    // TODO e): Perform the reduction using blocking collectives.
    long temporal_sum = sum;
    MPI_Reduce(&temporal_sum, &sum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

What is wrong here?

- The program expects that the reduced value is written to *sum*.
- This does not happen!
- **Reason:**
  temporal_sum is assigned the value of *sum* at that time.
  When *temporal_sum* changes later, the change is not visible!
  → No return value!
- **Fix:** Change *temporal_sum* to long**&** (or leave it away entirely)

Question 2: MPI Bug Hunt

- **In future:**
  When we talk about MPI programs, assume that the code is run by multiple processes.
- **In exam: Stated explicitly.**
- **Sorry for the confusion!**

Question 2.a): MPI Bug Hunt

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

// write the result to a file
std::ofstream file("result.txt");

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

delete[] result;
```

Segmentation fault!

All ranks write to same file!

Fix: < instead of <=

- Use MPI I/O
- Alternative:
  send all the data to root
  (might be inefficient)

Question 2.a): MPI Bug Hunt - Remark

- **Some people said to parallelize the «computationally expensive calculation»**
- **This might already be done, we have not said anything about that section (only that there is no error in there)**

Question 2.b): MPI Bug Hunt

```
// only two 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
```

Must be MPI_DOUBLE

Deadlock:

Both processes block because they want to send, but nobody can receive

Question 2.b): MPI Bug Hunt

Possible solution:

```
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);
}
```

Question 2.c): MPI Bug Hunt

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

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

int bval;
if(rank == 0)
{
    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;
```

Output with 1 process?

[0] 0

Output with 2 processes?

Deadlock!

Blocking collectives must be called by all processes!