Exercise 3

To compile: unzip our uploaded code, and run `make` inside `code/`. The slurm scripts are stored inside `code/slurm/`.

3.1 Measure Latency

The MPI ping-pong implementation in `code/src/latency.cpp` is measuring the roundtrip latency for each ping pong exchange 100000 times. This results in the latencies observed in Figure 1. Visualized in red are the times for two ranks running on the same node, while the communication between two ranks on two different nodes is visualized in blue.

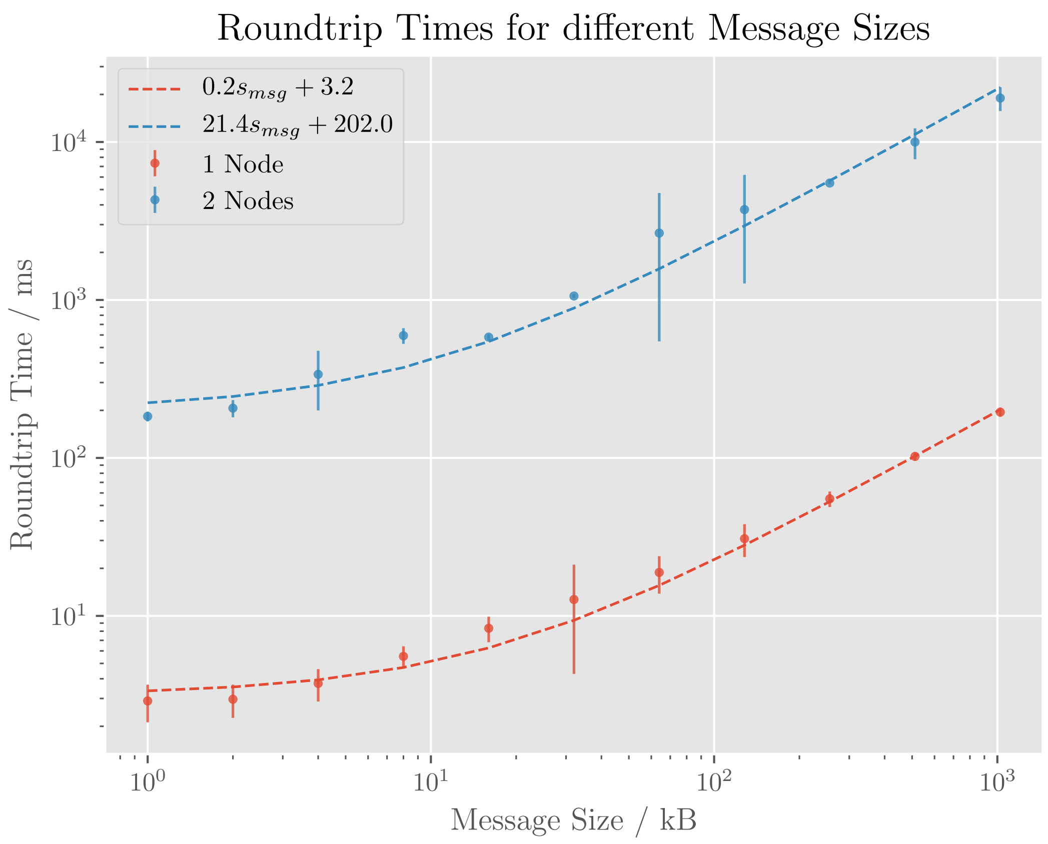

Figure 1: Roundtrip Latency over 100000 ping-pong exchanges

Both times the Latency roughly increases linearly with message size and for the communication between nodes a distinct offset and worse latency scaling can be observed

The scaling behaves worse by two magnitudes when comparing between node-to-node and inter-node communication.

3.2 Measure Bandwidth

The MPI flood-test implantation in `code/src/bandwidth.cpp` is measuring the bandwidth over an average of 1000 messages, repeating this measurement 100 times. This results in the bandwidths observed in Figure 2.

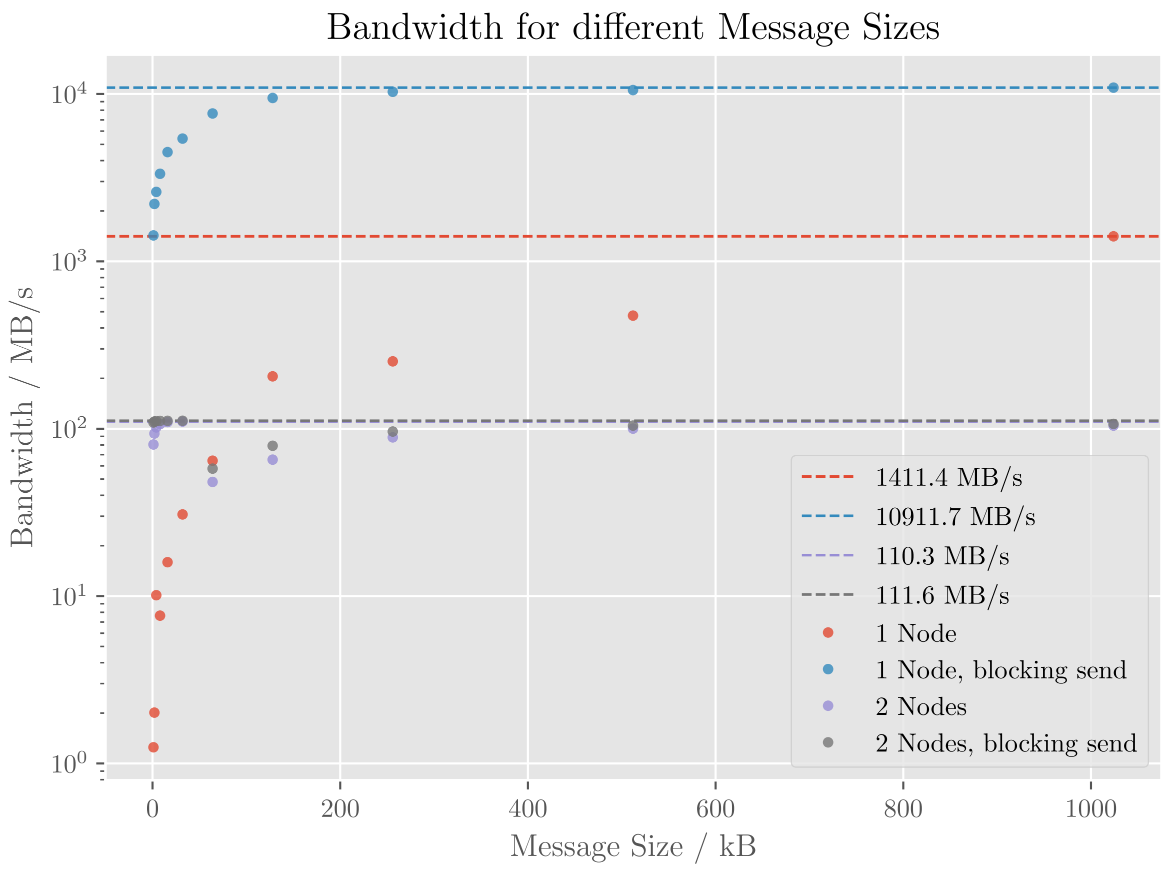

Figure 2: Bandwidth over 100000 flood-messages

Here a distinct increase in bandwidth is visible when communicating between two nodes. Additionally the communication between two nodes seems to be capped at ≈111 MB/s, in which case the interconnect may be the reason (1 Gb/s minus additional overhead additional overhead). **No distinct difference between blocking and non-blocking send** is observable when communication between nodes.

For the communication within one node, a distinct difference between blocking and non-blocking send was observable, here the **blocking send bandwidth was around** 30 % **higher**
.

3.3 Matrix multiply — sequential

The naïve implementation uses three nested for loops to perform the matrix multiply.

```
for (size_t i = 0; i < dim; ++i) {
        for (size_t j = 0; j < dim; ++j) {
                for (size_t k = 0; k < dim; ++k) {
                        C[i * dim + j] +=
                            A[i * dim + k] * B[k * dim + j];
                }
        }
}
```

On the *creek* cluster a performance of 0.104 162 GFLOP/s is achieved. This deviates a lot from the theoretical peak performance of the XEON E5. There are two main problems with the implementation:

- Cache utilization: The access pattern used for the naïve version is strided accesses to `B` are always cache misses.
- Hardware utilization: The processor used has 4 cores. The naïve Code only uses one thread.

To combat these effects the main loops were broken down into smaller blocks in order to gain better data locality. Instead of traversing the whole matrix progress is made block wise in each dimension. We thereby achieve both temporal and data locality, because more values are reused (still in cache) and the stride becomes limited to cachable blocks. The innermost loops are parallelized using *OpenMP*.

```
for (size_t ii = 0; ii < dim; ii += BLOCK_SIZE) {
    for (size_t jj = 0; jj < dim; jj += BLOCK_SIZE) {
        for (size_t kk = 0; kk < dim; kk += BLOCK_SIZE) {
#pragma omp parallel for collapse(3)
            for (size_t i = ii; i < std::min(ii + BLOCK_SIZE, dim); ++i) {
                for (size_t j = jj; j < std::min(jj + BLOCK_SIZE, dim); ++j) {
                    for (size_t k = kk; k < std::min(kk + BLOCK_SIZE, dim); ++k) {
                        C[i * dim + j] += A[i * dim + k] * B[k * dim + j];
                    }
                }
            }
        }
    }
}
```

Using this Code and a block size of 10 MB / 8 B/Element we achieve 0.223 383 GFLOP/s ≈ twice the performance. This is still pretty slow! On my local machine the speedup is a lot more significant (≈ factor 10).