CPP ASS2 rapport

Mathias Kirkeng (11904836) Tim Müller (11774606)

November 2019

1 Introduction

To facilitate parallel computing, several libraries exist to facilitate running code in parallel. In the previous assignment, we already looked at Pthreads and OpenMP. This week, we will instead look and benchmark the Message Passing Interface, or MPI.

MPI is, in contradiction to OpenMP and Pthreads, a library that handles parallelizm by creating multiple instances of the same code and then running that code on multiple computers that are connected to each other. In addition, MPI can also simulate this behaviour on one computer by creating multiple processes. This means that, unlike the previous libraries, there is no shared memory and all communication has to happen through a network.

To benchmark MPI and learn more about it, this assignment is split into two tasks: parallelizing a wave simulation and by implementing a collective communication example, both with MPI.

The wave simulation that is to be paralellized uses a time- and space-discretized equation, namely:

$$A_{i,t+1} = 2 \times A_{i,t} - A_{i,t-1} + c \times (A_{i-1,t} - (2 \times A_{i,t} - A_{i+1,t})) \tag{1}$$

This can be seen as an array of numbers that need to be calculated each time step, which depend the result of the previous timestep. The amount of elements of this array is denoted by i_max while the number of timesteps to be computed is denoted by t_max. All implementations use three buffers of a single timestep. One buffer for $t-1$, one for t and one for $t+1$. After all numbers in $t+1$ are calculated, the buffers are switched such that t becomes $t-1$, $t+1$ becomes t and a new $t+1$ can be calculated.

The problems in parallelizing the wave simulation is that the timesteps are dependent on each other. More specifically, a given $A_{i,t+1}$ is dependend on $A_{i-1,t}$, $A_{i,t}$ and $A_{i+1,t}$. When dividing the work among, in the case of MPI, processes, this means that there is a small dependency between the edges of the processes' data. This region of dependency is called a *halo region* (Fig 1).

The collective communication makes use of MPI's point-to-point communication. In this particular example, we pretend that MPI processes are structured in a circle: if we have n processes, arbitrary process i can only communicate

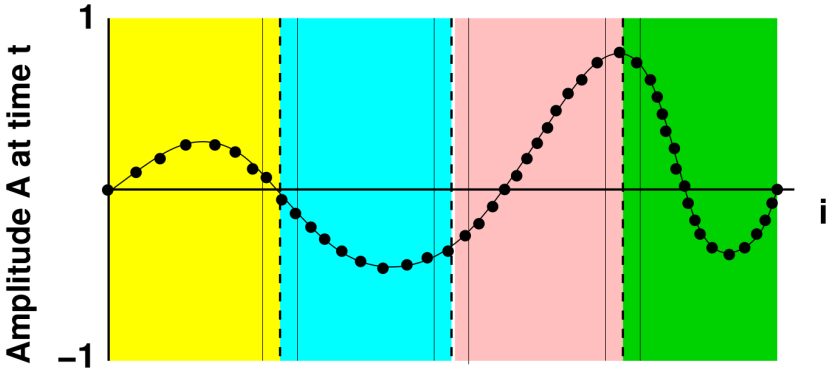

Figure 1: A visualisation of halo regions. Each colour represents the workload of a thread / process. The regions specified by the dotted lines represent the data points that introduce a dependency in between the processes. For the wave equation, this indicates that, for example, the yellow process requires the first element of the blue process. Similarly, the blue process requires the last element of the yellow process.

with processes $i-1$ and $i+1$. At the edges, the processes can communicate with the other edge. Using this structure, a broadcast function must be implemented that lets a message propagate through the network.

2 Implementation

On a high level the baseline sequential approach takes the following form:

---

let old **be** array of size i_max
let current **be** array of size i_max
let next **be** array of size i_max

for $t = 0$ **to** t_max:
    for $i = 0$ **to** $i\_max - 1$:
        calculate next(i)

    temp = old
    old = current
    current = next
    next = temp

In this case, *calculate_next(i)* is the computation for a new timestep which is shown in equation (1).

For this task, we focused on parallelizing the calculation of a single timestep. To this end, we developed two different implementations: *Blocking* and *Non-blocking*. Both operations are based on a the *MPI* message passing standard. The former uses blocking send and receive functions white the latter uses non-blocking send and receive fuctions. We didn't parallelize across the timesteps because they are all dependent on each other.

2.1 Blocking

The blocking MPI implementation starts with initialising and MPI world. Each process then gets its own rank and uses it to calculate which part of the timestep it needs to calculate. This work is divided evenly across the processes. In the case that there is remaining work, the last process will be assigned to do this. After this all processes but the first will send their part of the halo region to the left and listen for a message from their neighbour to the left. Then all processes but the last do the same thing but to the right. When each process has all necessary elements it does all its computations. Then all arrays are swapped and the loop continues to the next timestep. Once a process is done with all timesteps it sends its results to the main process(with rank 0) and exits. The main process receives these results into its own array from all processes and returns the array once this is done.

The pseudocode for this implementation looks like this:

```
prank = get_rank()
calculate own start_i and stop_i

for t = 0 to t_max:
    if (prank != 0):
        blocking_send(to=prank-1, buffer=current_array[start_i])
        blocking_recv(from=prank-1, buffer=current_array[start_i-1])

    if (prank != n_processes-1):
        blocking_send(to=prank+1, buffer=current_array[stop_i-1])
        blocking_recv(from=prank+1, buffer=current_array[stop_i])

    for i = start_i to stop_i:
        // calculate calculates an element according to the wave equation
        next_array[i] = calculate(i)

    // swaps the old, current and next buffers
    swap_buffers()

if (prank == 0):
    for i = 0 to n_processes:
        process_start_i = calculate_start_i(i)
        blocking_recv(from=i, buffer=current_array[process_start_i])
```

```
else:
    blocking_send(to=0, buffer=current_array[start_i-1])
    exit()
```

The main advantage over the sequential version is that the work is divided over multiple processes which can execute the computation in parallel. Because of this we expect significant speedup versus the sequential implementation with with more processes.

In this version, both the send and receive are blocking. While we didn't implement it, it is worth considering an implementation with a non-blocking send and a blocking receive. This code can be very similar to the blocking implementation, in the sense that we can fire a send at the start and wait for it to complete at the end of a timestep-iteration. However, this implementation offers no performance speedup, as the blocking receive immediately follows the send. Since the actual sending isn't fast, this means that the receive returns no earlier than before. Therefore, to achieve speedup, we have to migrate the receive to after the bulk computation. This looks as follows:

```
prank = get_rank()
calculate own start_i and stop_i

for t = 0 to t_max:
    if (prank != 0):
        non-blocking_send(to=prank-1,
                          buffer=current_array[start_i],
                          handle=send_left)

    if (prank != n_processes-1):
        non-blocking_send(to=prank+1,
                          buffer=current_array[stop_i-1],
                          handle=send_right)

    for i = start_i + 1 to stop_i - 1:
        // calculate calculates an element according to the wave equation
        next_array[i] = calculate(i)

    // receive the halo regions
    if (prank != 0):
        wait(send_left)
        blocking_recv(from=prank-1, buffer=current_array[start_i-1])
    if (prank != n_processes - 1):
        wait(send_right)
        blocking_recv(from=prank+1, buffer=current_array[stop_i])

    // compute the elements from the halo region
```

```
        next_array[start_i] = calculate(start_i)
        next_array[stop_i-1] = calculate(stop_i-1)

        // swaps the old, current and next buffers
        swap_buffers()

    if (prank == 0):
        for i = 0 to n_processes:
            process_start_i = calculate_start_i(i)
            blocking_recv(from=i, buffer=current_array[process_start_i])
    else:
        blocking_send(to=0, buffer=current_array[start_i-1])
        exit()
```

Now, the program does useful work while waiting for the send() to complete.

2.2 Non-blocking

The non-blocking implementation makes use of non-blocking send and receives. The main difference in this one is that the computation is done for all elements except the first and last(the halo regions). This is because these computations can be done without any information from the neighbours and as such can be done while waiting for the sends and receives to complete. After the computation all processes but the first wait for the left send and receives to complete and then calculate the first element of the timestep. Then all processes but the last wait for the right send and receives to complete and then calculate the last element of the timestep. The rest works just as the blocking implementation.

The pseudocode for this implementation looks like this:

```
prank = get_rank()
calculate own start_i and stop_i

for t = 0 to t_max:
    if (prank != 0):
        non-blocking_send(to=prank-1,
                          buffer=current_array[start_i],
                          handle=send_left)
        non-blocking_recv(from=prank-1,
                          buffer=current_array[start_i-1],
                          handle=recv_left)

    if (prank != n_processes-1):
        non-blocking_send(to=prank+1,
                          buffer=current_array[stop_i-1],
                          handle=send_right)
        non-blocking_recv(from=prank+1,
```

```
                                buffer=current_array[stop_i],
                                handle=recv_right)

        for i = start_i + 1 to stop_i − 1:
            // calculate calculates an element according to the wave equation
            next_array[i] = calculate(i)

        if (prank != 0):
            wait(send_left)
            wait(recv_left)

        // swaps the old, current and next buffers
        swap_buffers()

    if (prank == 0):
        for i = 0 to n_processes:
            process_start_i = calculate_start_i(i)
            blocking_recv(from=i, buffer=current_array[process_start_i])
    else:
        blocking_send(to=0, buffer=current_array[start_i-1])
        exit()
```

The advantage of this implementation over the blocking implementation is that communication and computation are overlapped. This way the process doesn't first wait for the communication to complete while doing nothing. However we expect the communication to not take that long. Because of this we expect the speedup over the blocking implementation to be minor.

2.3 Ring broadcast

For this part we first implemented a simple MPI broadcast and a MPI broadcast which is optimized for a ring topology. The simple version is called by both the sending and receiving nodes. The process starts by getting its rank. It then compares its rank to the root to see whether it is the sending node. If it is, simply iterates over the ranks and sends each process a message. If it is not, it receives a message from the root.

The optimized version is also called by both the sending and receiving process. The process then uses its rank to determine whether it is the root of a receiver. If it's the root it sends the message to both it's neighbours. It calculates it's neighbours rank with the formula $(rank \pm 1)\ mod\ n\_processes$. This ensures that the rank circles back around if it works out to a negative number or a number bigger than $n\_processes$. If the process is a receiver it calculates the distance between itself and the root and uses this to determine from which side the message will arrive. If distance is shorter on the right side, it will listen to its right neighbour and, once it receives a message, sends the message to the

left neighbour. The left side works the other way around. On a higher level this means that the root node will send a message to both its neighbours and the message will propagate on both sides through the processes until it reaches the opposite side.

The pseudocode for the optimized implementation is as follows:

```
let root be the rank of the initiating process
let message be the message to be broadcasted
prank = get_rank()

next_neighbour = (prank + 1) mod n_processes
previous_neighbour = (prank - 1) mod n_processes

if (prank == root):
    send(to=next_neighbour, buffer=message)
    send(to=prev_neighbour, buffer=message)
else:
    // calculates distance to root via left neighbour
    distL = calculate_dist(root, prank, left)
    // calculates distance to root via right neighbour
    distR = calculate_dist(root, prank, right)

    if (distL < distR):
        recv(from=next_neighbour, buffer=message)
        send(to=prev_neighbour, buffer=message)
    else:
        recv(from=prev_neighbour, buffer=message)
        send(to=next_neighbour, buffer=message)
```

For n processes the broadcast requires n atomic communication events. However because these messages propagate in both directions through the ring network every two communication events happen in parallel. Because of this we expect a speedup of two versus the simple broadcast implementation.

2.4 Pthreads and OpenMP

We also chose two different implementations of multithreading to compare the MPI implementations against.

The first, Keep-alive, is based on pthreads and aims to divide the work evenly accross all threads, giving the last thread the remaining work. The main thread coordinates all threads, keeping them alive and making sure they wait for all other threads to be finished with the current timestep before moving on.

The second implementation is based on OpenMP and adds a OMP parallel pragma over the inner loop using the *for* work sharing contract.

3 Results

Below are the results of this research. These are only the results from the wave simulation, as there is no need for performance metrics from the collective communication.

3.1 Implementation Results

To research the performance of each implementation, each of them was run on the DAS4 with different parameters. The two parameters that we varied were the number of nodes number of cores and the number of amplitude points (APs). The number of nodes cores used are: 1 node 8 cores, 8 nodes 1 core and 8 nodes 8 cores. The number of amplitudes used are: 10^3, 10^4, 10^5, 10^6 and 10^7.

For the amplitude points, we also varied the number of timesteps to keep the total number of elements constant. For 10^3 APs, we used 10^7 timesteps. Similarly, we used 10^6 timesteps for 10^4 APs, etc.

The sequential implementation is largely consistent regarding runtimes. In general, a higher number of APs results in marginally larger runtimes (figure 2).

The blocking implementation has the highest runtimes for 10^3 amplitude points, and seems to produce the lowest runtimes for 10^6 (figure 3).

The non-blocking implementation is, too, the slowest for 10^3 and the fastest for 10^7 (figure 4).

The average runtimes for different system configurations are shown in figure 5. In general, the sequential shows larger runtimes than the blocking and non-blocking implementations.

Finally, the results compared to the results from last week are shown in figure 6. Both Destroyed and Keep-Alive were slower than Sequential, Blocking and Non-Blocking for 6 threads and 8 nodes, 1 core for MPI.

4 Discussion

In this section, we discuss the results gained from the research into the wave equation.

4.1 MPI Comparison

For all AP$> 10^3$, the sequential performed worse on the Blocking and Non-Blocking implementations. The Non-Blocking was even faster for AP$= 10^3$. The fact that Blocking is slower for this value is probably because of the gain from the parallelism is outweighed by the size of the overhead from the network operations. It is in accordance to our hypothesis.

The Non-Blocking implementation is also slightly faster than the Blocking one. The difference is only so small because the overhead from the network is not

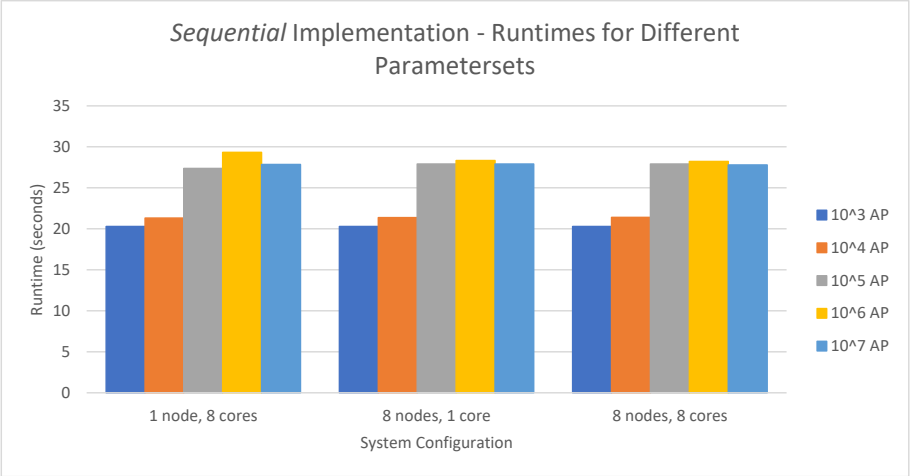

Figure 2: The runtimes for the sequential implementation. The different bars per system configuration denotes the number of amplitude points the simulation was run with.

as big as we might like. This is in accordance with our expectations. However, what is a little surprising is that Non-Blocking performs better in comparison to Blocking for smaller numbers of APs. This is because there are more timesteps when a smaller number of APs are chosen, resulting in more network operations. This in turn increases the effect of the non-blocking networking, which makes the speed difference higher.

4.2 Threading vs MPI

As can be seen in the results, the all MPI implementations perform significantly better than the threading implementations. Though these results may be skewed do to the outliers at smaller amplitude points in the threading implementations. Apparently the overhead creating and coordinating threads is more than the overhead of sending messages between processes and between nodes.

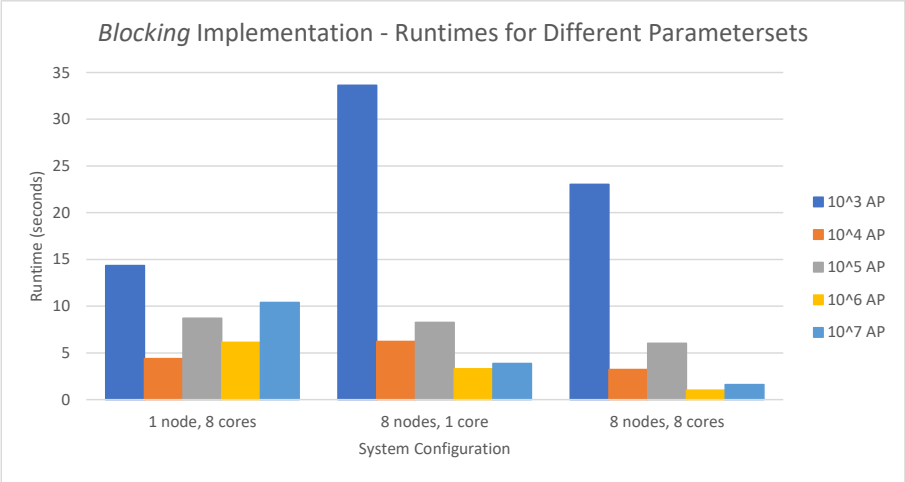

Figure 3: The runtimes for the blocking implementation. The different bars per system configuration denotes the number of amplitude points the simulation was run with.

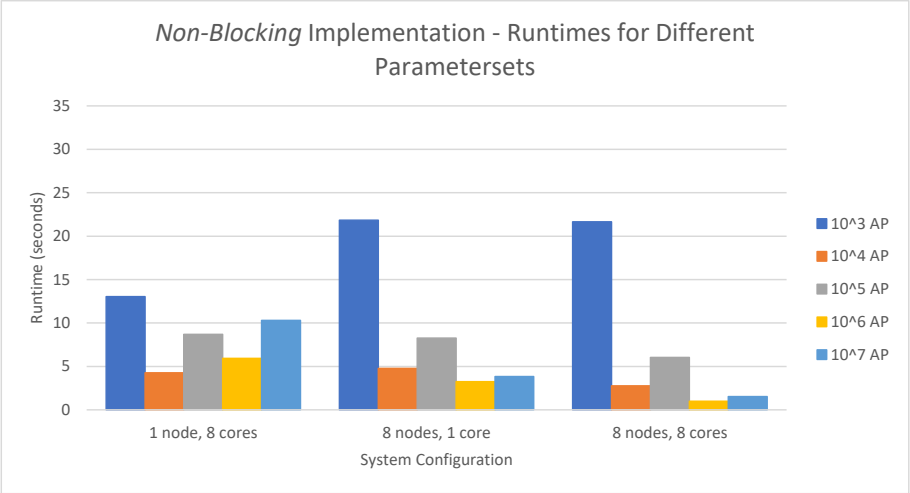

Figure 4: The runtimes for the non-blocking implementations. The different bars per system configuration denotes the number of amplitude points the simulation was run with.

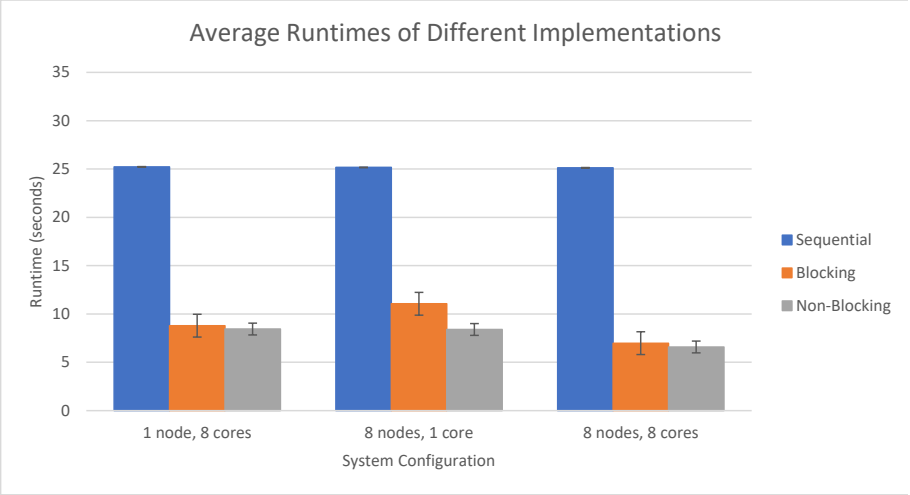

Figure 5: The average runtimes of the various implementations compared for each system configuration.

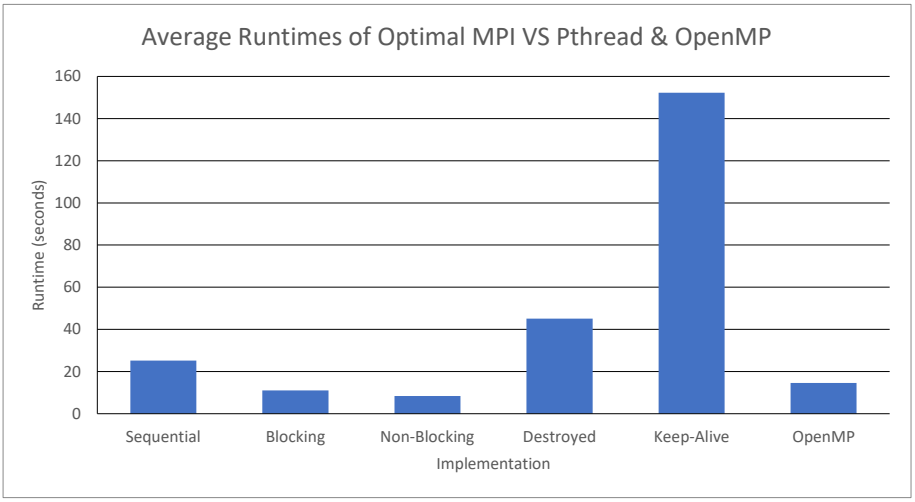

Figure 6: The average runtimes for MPI Pthread / OpenMP implementations compared. For MPI, a system configuration of 8 nodes and 1 cores was used. For Pthread OpenMP, 6 threads were used.