High-Performance Computing Lab for CSE **2021**

Student: Sina Klampt

Solution for Project 6

Due date: May 20, 2021, 12pm (midnight)

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 35 points]

1.1. Initialize and finalize MPI [5 Points]

Since this part was pretty straight forward, I just followed the instructions given. I added a few lines to initialize the MPI, get the current rank and number of ranks and finalize the MPI.

1.2. Create a Cartesian topology [5 Points]

I used the MPI_Dims_create function to determine the number of sub domains in the x- and y-direction. With the MPI_Cart_create function I created a Cartesian communicator. Then using the MPI_Cart_coords function I got the coordinates of a given rank in the Cartesian topology. Finally, with the MPI_Cart_shift function, I determined the neighbours of a given rank.

1.3. Extend the linear algebra functions [5 Points]

In this sub task we had to implement two functions, the dot product and the hpc norm. For the dot product I used the MPI_Allreduce function to share my locally computed dot products between the different ranks. I used the same method to implement the norm. We need a vector distributed over all ranks in these functions because they depend on all entries and need therefore need to communicate. In the other functions, we only have local operations.

1.4. Exchange ghost cells [10 Points]

In this task we needed to consider periodic boundary conditions, since otherwise we would not know how many neighbours one rank has. I therefore created an array of requests with the maximal length and filled it with the number of requests I needed. Then I loaded the correct values into the buffer and sent them afterwards. I counted the number of requests and used the MPI_Waitall function to wait for the communication to finish.

1.5. Scaling experiments [10 Points]

I assumed that it would be really hard to get data for more than 24 threads as we have seen in previous exercises. I tried using the XeonGold 6150 processor using 1-24 threads. However there have been a lot of errors when running the code on Euler and I could not figure out the mistakes in my code as time was running out. Therefore, I was not able to reproduce any data to plot unfortunately.

I would expect that it would scale well especially for bigger tasks since we can split up the work amongst the processors. I also think that the performance increases with the number of threads, like in other tasks.

2. Python for High-Performance Computing (HPC) [in total 50 points]

2.1. Sum of ranks: MPI collectives [5 Points]

In this sub task we had to implement the sum of all ranks with two different approaches.

For the "pickle-based" communication of generic Python objects, I called "send" with the send-buffer and the destination and "recv" with the source.

For the direct array data communication I called on the one hand "Send" along with the array, the MPI types in the array, the destination and a tag and on the other hand "Recv" with the same arguments except changing the destination to the source. I used non-blocking communication for both methods.

2.2. Domain decomposition: Create a Cartesian topology [5 Points]

For this sub task I used the Compute_dims, Create_cart, Get_coords and Shift function to compute the 2d periodic process distribution and create a Cartesian topology. In the end I printed the output according to the instructions given in the exercise. All of the above mentioned functions work similar to the equivalent functions in C++.

2.3. Exchange rank with neighbours [5 Points]

We had to combine the previous sub tasks to solve this sub task. I used non-blocking communication again and send to one neighbour (e.g. east) while receiving from another (west). In order for the sending to be in parallel, I wait at the end instead of waiting after every send-receive. At the end I compared the results to the results of the Shift function to make sure they were the same.

2.4. Parallel space solution of a nonlinear PDE using Python and MPI

2.4.1. Change linear algebra functions [5 Points]

I implemented the dot product the same way as in the C++ version. The only part that is different are the numpy arrays. For the norm I just took the square root of the result after calling the hpc_dot function.

2.4.2. Exchange ghost cells [5 Points]

I filled the buffers when necessary using -1 to access the last row or column. I set the requests as member variables because they were also needed in the exchange_waitall function. I again used non-blocking send and receives if the neighbour exists and wait for the requests.

2.4.3. Scaling experiments [5 Points]

I tried using the same scaling as in the C++ exercise and ran my code on the same processor, XeonGold 6150. However I got the following error which I could not find a solution for:

It looks like opal_init failed for some reason; your parallel process is likely to abort. There are many reasons that a parallel process can fail during opal_init; some of which are due to configuration or environment problems. This failure appears to be an internal failure.

2.5. A self-scheduling example: Parallel Mandelbrot [20 Points]

I implemented this task using pickle-based communication. The idea is that the the manager sends work to all the workers that are initialized in an infinite loop. After receiving a finished task, the manager then reassigns a task to the worker who has just finished a job. This is repeated until no tasks are left to be done and assigned. The manager waits for the remaining workers to finish and tells them that there are no more tasks.

As stated in the exercise, I added a scaling study using different number of workers splitting the workload once into 50 and once into 100 tasks. You can see my results in the following plot (Figure 1):

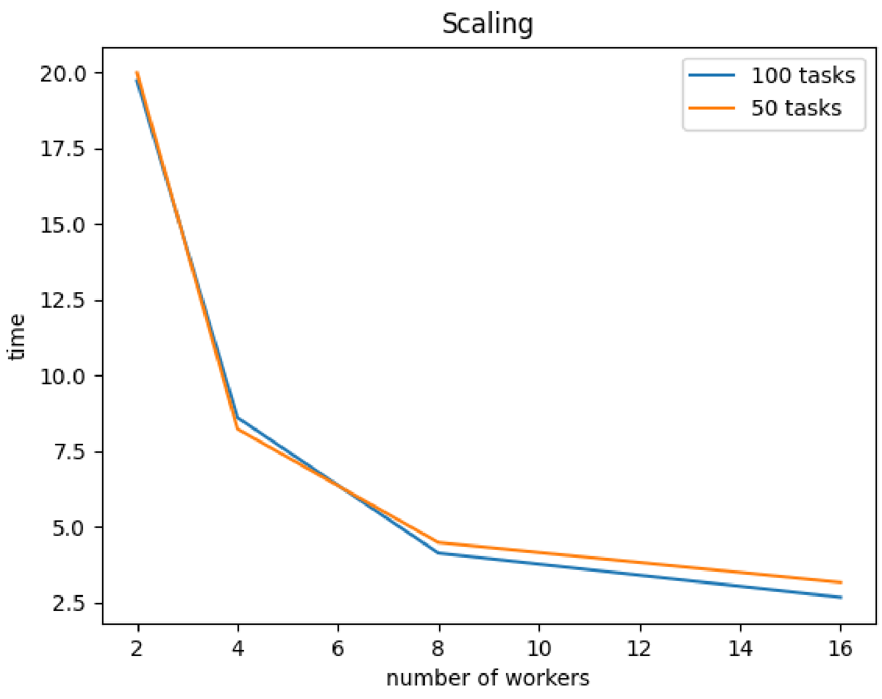

Figure 1: Scaling study using 2, 4, 8, 16 workers over time