Università della Svizzera italiana | **Institute of Computing CI**

High-Performance Computing Lab **Institute of Computing**

Student: Fabian Gobet Discussed with: Ekkehard Steinmacher, Henrique Gil

Solution for Project 5

1. Task 1 - Initialize and finalize MPI [5 Points]

This exercise is pretty straight forward. As in prior exercises we start by initializing MPI with the passed arguments and define the communication size and the relative rank.

```
...
MPI_Init(&argc, &argv);
int mpi_rank, mpi_size;
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
...
```

And at the end we terminate the MPI process with

```
...
MPI_Finalize();
return 0;
```

2. Task 2 - Create a Cartesian topology [10 Points]

To create a non-periodic cartesian topology, I make use of the MPI_Cart_create() function.

First the array with number of elements per dimension is initialized with zeros so that the function MPI_Dims_create() can initialize the correct format of sizes per dimension into the former array. A periods array is also initialized to zeros because we don't want cyclicity to happen in the borders.

Then, we initialize the cartesian topology communicator with MPI_Cart_create() with the former arrays and get the coordinates in the cartesian topology for the relative rank with MPI_Cart_coords().

Given all this process, we are in position to retrieve the respective neighbours in the cartesian topology according to the dimensions that we've established.

```
...
int dims[2] = { 0, 0 };
MPI_Dims_create(mpi_size, 2, dims);
ndomy = dims[0];
ndomx = dims[1];
int periods[2] = { 0, 0 };
MPI_Comm comm_cart;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);
domain.comm_cart = comm_cart;
MPI_Cart_coords(comm_cart, mpi_rank, 2, coords);
domy = coords[0]+1;
domx = coords[1]+1;
MPI_Cart_shift(comm_cart, 0, 1, &neighbour_south,&neighbour_north);
MPI_Cart_shift(comm_cart, 1, 1, &neighbour_west, &neighbour_east);
...
```

3. Task 3 - Change linear algebra functions [5 Points]

After each function has computed its part of the dot product, we want to accumulate the results of each process to all the others. This means that all processes will end up with the same result, which is the addition of all partial inner products. To achieve this, we can make use of the function MPI_Allreduce(), saving the results into a variable. Hence we have

```
double ss_dot(Field const& x, Field const& y)
{...
    double rcv = 0;
    MPI_Allreduce(&result, &rcv, 1, MPI_DOUBLE, MPI_SUM, data::domain.comm_cart);
    return rcv;
}
```

Analogous to the prior exercise for the dot product, we want to reduce in an addition all the partial results for the partial inner product and return the square root of it, which equates to the euclidean norm as pretended. Hence we have

```
double ss_norm2(Field const& x)
{...
    double rcv = 0;
    MPI_Allreduce(&result, &rcv, 1, MPI_DOUBLE, MPI_SUM, data::domain.comm_cart);
    return sqrt(rcv);
}
```

4. Task 4 - Exchange ghost cells [45 Points]

For this section of the exercise, we have to see if there is a neighbour on each of the possible directions and then asynchronously receive their data and set it to the respective buffer (according to the direction for which the neighbour is positioned), and copy the respective data into a buffer and asynchronously send it to that neighbour.

For the north and and south neighbours we'll be sending a data comprised on nx doubles, whereas for east and west neighbours we'll be sending a data comprised of ny doubles. Furthermore, the operator () from the Field class allows us to retrieve data on a grid-like manner without having the need to create a ghost data type, thus copying data to send into a buffer is straight forward.

The implementation idea is therefore non-blocking with P2P communication.

The code to achieve this for the south, east and west neighbours is

```
if(domain.neighbour_south>=0) {
    MPI_Irecv(&bndS[0], nx, MPI_DOUBLE, domain.neighbour_south, domain.neighbour_south,
        comm_cart, requests+num_requests); // 2
    num_requests++;
    for(int i=0; i<nx; i++)
        buffS[i] = U(i,0);
    MPI_Isend(&buffS[0], nx, MPI_DOUBLE, domain.neighbour_south, domain.rank,
        comm_cart, requests+num_requests);
    num_requests++;
}
if(domain.neighbour_east>=0) {
    MPI_Irecv(&bndE[0], ny, MPI_DOUBLE, domain.neighbour_east, domain.neighbour_east,
        comm_cart, requests+num_requests);
    num_requests++;
    for(int i=0; i<ny; i++)
        buffE[i] = U(nx-1,i);
    MPI_Isend(&buffE[0], ny, MPI_DOUBLE, domain.neighbour_east, domain.rank,
        comm_cart, requests+num_requests); // 5
    num_requests++;
}
if(domain.neighbour_west>=0) {
    MPI_Irecv(&bndW[0], ny, MPI_DOUBLE, domain.neighbour_west, domain.neighbour_west,
        comm_cart, requests+num_requests);
    num_requests++;
    for(int i=0; i<ny; i++)
        buffW[i] = U(0,i);
    MPI_Isend(&buffW[0], ny, MPI_DOUBLE, domain.neighbour_west, domain.rank,
        comm_cart, requests+num_requests); // 7
    num_requests++;
}
```

The biggest computational burden comes from computing the non-boundary inner grid points, in comparison with the inner boundary grid points. Because the former points have no dependence on the boundary points, we can perform this computation before all messages have been sent and received.

Once these points have been processed, we then proceed to process with inner-boundary grid points, which have a dependence on the boundary points which are exchanged in communications. Thus, it is crucial for each process to only proceed to this computation once all messages have been received. Therefore, in order to ensure that this is true, we can make use of the function MPI_Waitall(). This function takes in as argument the array with the requests regarding all the asynchronous communications and performs a blocking operations until all these request have been satisfied. In other words, the operation is blocking until all communications have been effective. The code to achieve this is the following

```
... // interior grid points processing
MPI_Waitall(num_requests, requests, statuses);
... // boundary points processing
```

5. Task 5 - Testing [20 Points]

In order to perform a scaling test I recorded all execution times for every combination of

- processes ∈ {1,2,4}
- threads ∈ {1,2,...,9,10}
- grid size ∈ {128,256,512,1024}

This equates to 120 data samples, so then it is necessary to choose a way of representing them. As such, I have chosen to compare each number of running threads along the grid sizes, for each of the number of processes, coming up with the following 3 plots

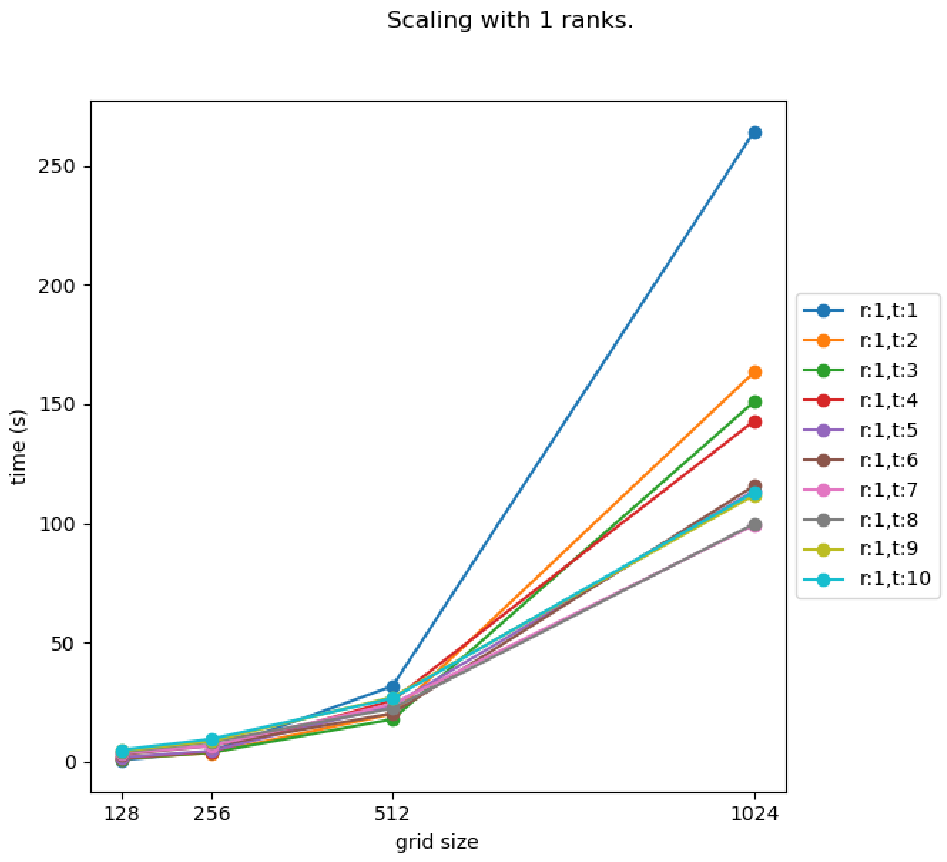

Figure 1: Running times: 1 process

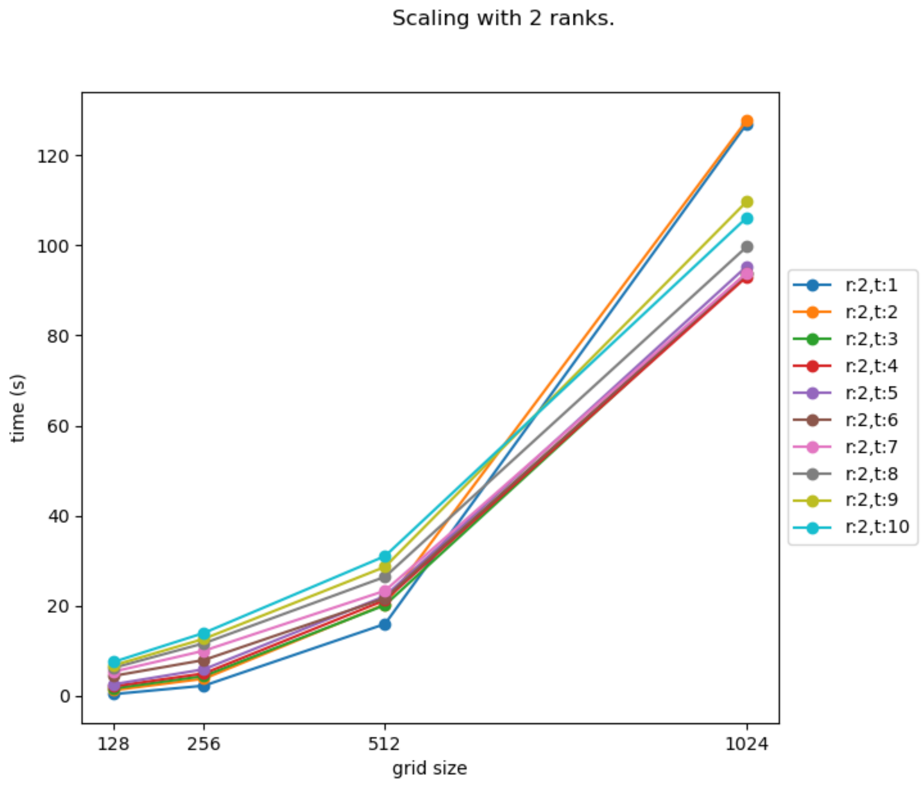

Figure 2: Running times: 2 processes

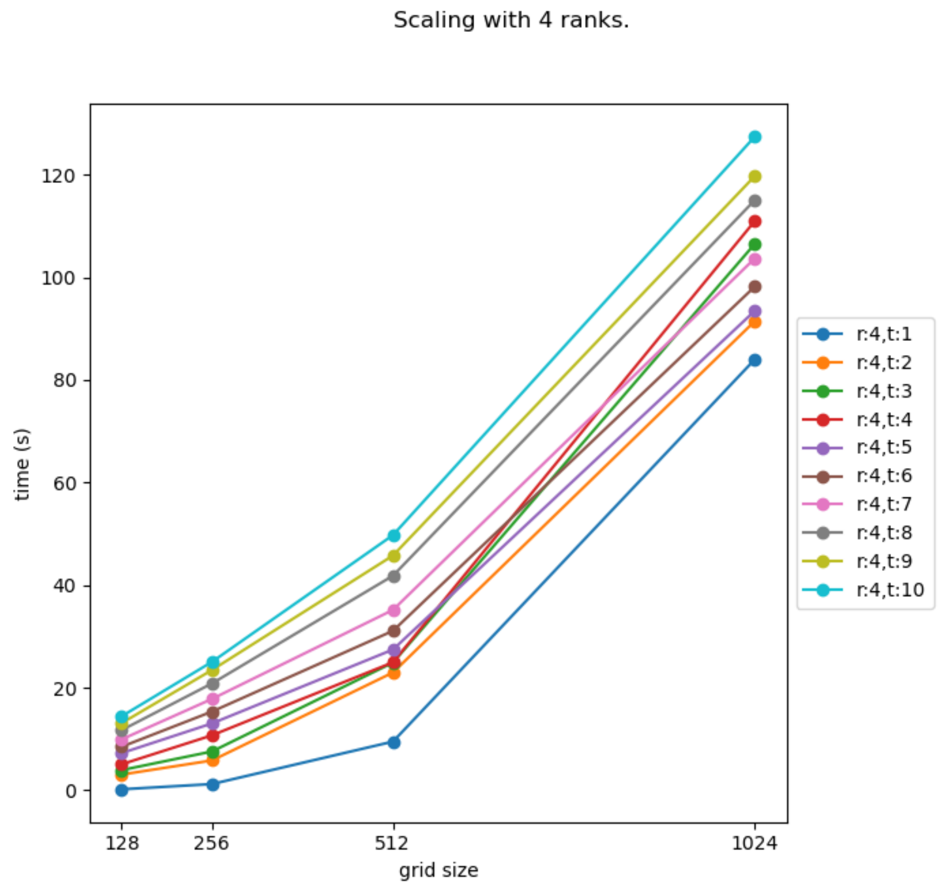

Figure 3: Running times: 4 processes

We can observe, comparing the several plots, that on a MacBook 16 Pro with 12 cores, from 1 to 4 processes the tendency is to regularize the running times in accordance to the number of threads. This happens because both MPI and OpenMP have their share of thread overhead when it comes to a a respective deployment. This is a strong indication that growing number of processes and number of threads for OpenMP isn't linear with the decrease of running times, as there may be a point for which overhead tends to affect performance to greater extent.

These plots give us an overall good idea of whats happening, so then one asks questions such as 'What is then the optimal number of processes and threads, in this setup, that provides the best performance?'.

We can derive this by analytically searching the data, thus coming up with he following table of best 2 for each grid size.

Threads	Ranks	Grid	Time
1	4	1024	84.0282
2	4	1024	91.4836
1	4	512	9.49185
1	2	512	15.90390
1	4	256	1.21170
1	2	256	2.19919
1	4	128	0.191735
1	2	128	0.318218

So then, we can conclude that overall 4 ranks and 1 threads only, on a MacBook 16 pro, seems to provide the best performance wise running times.