Università della Svizzera italiana | Institute of Computing CI

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

Student: Kaan Sen Discussed with: Lorenzo Varese

Solution for Project 5

This project will introduce you to a parallel space solution of a nonlinear PDE using MPI.

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

In the file `main.cpp`, MPI is initialized, the current rank and the number of ranks is obtained, and MPI is finalized the following way:

```
int thread_level;
MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &thread_level);
...
```

To ensure memory allocation we add this code to our program :

```
    MPI_Comm_free(&domain.comm_cart);
    MPI_Finalize();
```

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

In the file `data.cpp`, the number of partitions in each dimension is calculated:

```
   // determine the number of subdomains in the x and y dimensions
int dims[2] = { 0, 0 };
...
// create a 2D non-periodic cartesian topology
int periods[2] = { 0, 0 };
MPI_Comm comm_cart;
```

for detailed version code is available
A communicator with cartesian topology and the coordinates of the current rank in the domain grid are created:

```
MPI_cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);
domain.comm_cart = comm_cart;

int coords[2];
MPI_cart_coords(comm_cart, mpi_rank, 2, coords);
domy = coords[0]+1;
domx = coords[1]+1;
```

And finally, neighbors of the current rank: east, west, north, and south directions are identified:

```
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]

In the file linalg.cpp, a collective operation to compute the dot product is added as follows:

```
double final_result=0;
MPI_Allreduce(&result, &final_result, 1, MPI_DOUBLE, MPI_SUM, data::domain.comm_cart);
return result;
```

A collective operation to compute the norm is available in the code.

These are the only functions that need to be communicated by all the ranks, since these two functions are the only ones returning values, and the values returned should be in sync in order for the results of the communication to be successful.

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

In the file operators.cpp, point-to-point communication for all neighbors in all directions is added. For the receive and send operations, non-blocking communication is used. For the south neighbor:

```
if(domain.neighbour_south>=0) {
    //TODO
    //Receiving buffer
    MPI_Irecv(&bndS[0], nx, MPI_DOUBLE, domain.neighbour_south, domain.neighbour_south,
        comm_cart, requests+num_requests);
    num_requests++;

    //packing buffer
    for(int i=0; i<nx; i++)
        buffS[i] = U(i,0);

    //Sending buffer
    MPI_Isend(&buffS[0], nx, MPI_DOUBLE, domain.neighbour_south, domain.rank,
        comm_cart, requests+num_requests);
    num_requests++;
}
```

for east neighbor, then for West the similar strategy.
In the remaining of the code, the following call is used to wait for the async transfers to finish before the boundary operations:

```
//Use MPI_Waitall for this
MPI_Waitall(num_requests, requests, statuses);
```

The program output with a grid size of 128 × 128 is given below. The resulting figure is provided in the figure 1.

5. Task 5 - Testing [20 Points]

The program is executed for p = 1, 2, ..., 10, where p is the number of threads. Different amounts of MPI ranks are used. N = 1, 2, 4, where N is the number of ranks. Various combinations of grid size, number of ranks, and number of threads are experimented with. Where s is the grid size, the benchmarks used are: s = 128×128 s = 256×256 s = 512×512 The execution times of running the program for different grid sizes, with various number of threads and MPI ranks can be found below:

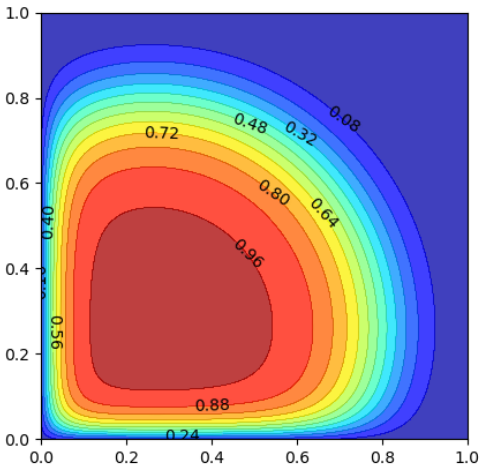
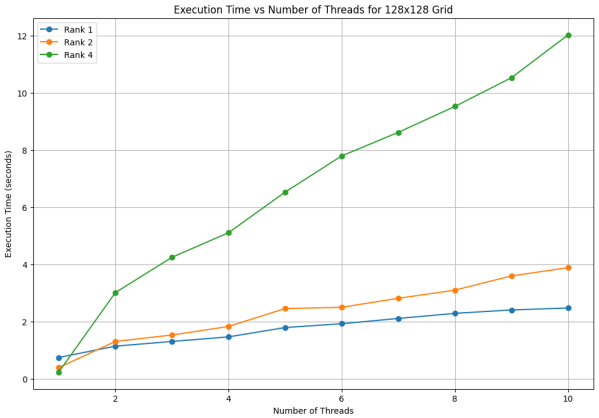
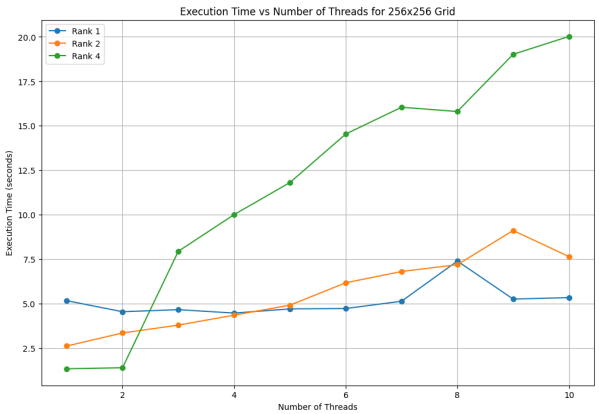


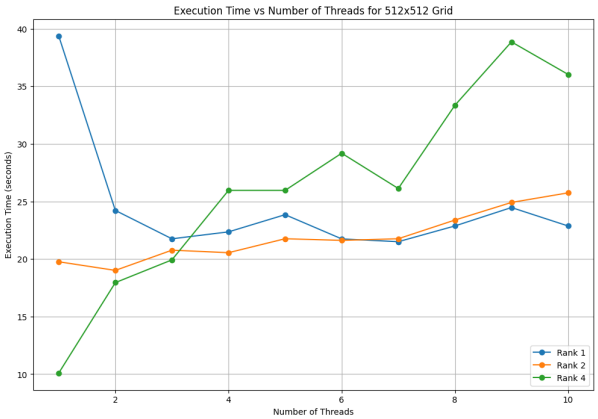
Figure 1: Resulting figure of the program



(a) Execution time for Grid 128x128



(b) Execution time for Grid 256x256



(c) Execution time for 512x512

Figure 2: Execution times for different Grids

For the 128x128 Grid:

- Rank 4's execution time increases significantly with more threads, which is similar to the 256x256 grid case and suggests that the overhead of threading is significant relative to the computation being done.

- Rank 1 and Rank 2 show improved execution time with more threads, with Rank 1, in particular, showing a steady improvement. This could be because the smaller grid size allows for more efficient data handling and less overhead.

For the 256x256 Grid:

- Rank 2 and 4 shows an increase in execution time as the number of threads increases. This is counterintuitive because we typically expect that adding more threads would decrease execution time due to parallelism. However, on multicore systems like the M1, this could indicate that the overhead of managing more threads outweighs the performance gains from parallelism, especially if the workload per thread is not enough to justify the parallel overhead.
- Ranks 1 show an initial decrease in execution time as threads increase up to 4, and then it levels off or increases slightly. This suggests that there is a sweet spot where the overhead and performance gains balance out.

For the 512x512 Grid:

- Rank 1 again shows a drastic reduction in execution time as threads increase initially, but then it fluctuates. The fluctuations could be due to inconsistent thread management or varying levels of cache efficiency.
- Rank 2 and Rank 4 display a more consistent decrease in execution time as threads increase, with Rank 2 seeming to benefit more from additional threads than Rank 4. This could mean that the workload is distributed more efficiently across threads in Rank 2.