HPC4M - ASSIGNMENT 2

KAROLINA BENKOVA

Exercise 3

The task in this exercise is to use the halo-swapping technique to solve the 1D heat equation in parallel. We will employ point-to-point MPI functions `MPI_Ssend` and `MPI_Recv`.

1. Set-up for running the job on Cirrus

The number of processes we are going to use as an example is going to vary based on the formula $M + 1 = \text{size} \cdot (J - 2) + 2$, where `size` stands for number of processes, J is the number of grid points given to each process and M is the total number intervals (hence we have $M+1$ datapoints). This desired number of processes can be adjusted in file `run_job.slurm` in `tasks-per-node`. The output file is stored in the file `out`.

2. Discretisation of the heat equation

In this task we parallelise the following heat equation

$$u_t = u_{xx} \quad \text{for } x \in (0,1), t \in (0,T),$$

subject to an initial condition

$$u(x, t = 0) = \sin(2\pi x) + 2\sin(5\pi x) + 3\sin(20\pi x),$$

and homogeneous Dirichlet boundary conditions

$$u(x = 0, t) = u(x = 1, t) = 0.$$

To discretise the equation we apply the method of lines in space and forward Euler scheme in time which leads to

$$\frac{U_m^{n+1} - U_m^n}{\Delta t} = \frac{U_{m-1}^n - 2U_m^n + U_{m+1}^n}{(\Delta x)^2} \quad \text{for } m \in 1, ..., M-1, n \in 0, ..., N-1,$$

with the discretised initial and boundary conditions

$$\begin{aligned} U_m^0 &= \sin(2\pi x_m) + 2\sin(5\pi x_m) + 3\sin(20\pi x_m) \quad \text{for } m \in 1, ..., M-1 \\ U_0^n &= U_m^n = 0 \quad \text{for } n \in 0, ..., N, \end{aligned}$$

where U_m^n is a discrete approximation for $U(x = x_m = m\Delta x, t = t_n = n\Delta t)$, with $\Delta X = 1/M$ and $\Delta t = T/N$ for $M, N \in \mathbb{Z}^+$. We note that in order to update the point at step $n + 1$, three values at the previous step need to be used $(U_{m+1}^n, U_M^n, U_{m+1}^n)$.

3. Parallel code

The problem outlined in the previous section is to be parallelised by splitting the physical space into subintervals, assigning each subinterval to a separate process. The subintervals will be overlapped by two points as we don't have information about their end points unless they are the boundary points for which we have defined boundary conditions. The overlapping will enable us to use the halo-swapping technique. Each subinterval will update all their middle points (i.e. all points except the endpoints), and the missing information about the endpoints is going to be obtained from the neighbouring intervals using send and receive functions in MPI.

In the script `heat_par.cpp`, we start with setting up the initial condition in a $M+1$-dimensional vector. We consider it easier to do it this way instead of in the J-dimensional subvectors as the initial values depend on $m \in \{0, ..., M\}$. In this vector we also include the Dirichlet BCs at time 0 (or t_n). Next, we broadcast the initial condition vector to all the other processes involved in the parallel computation using the `MPI_Bcast` function. The vector is split into as many sub-vectors (subintervals) as the number of processes, each of size J. The Dirichlet BCs at time $n+1$ are defined in the second row of the vector for the first and last process as these processes involve the boundaries of the domain.

In the numerical scheme itself, next step is calculated for all values in the vector except the first and the last place in each of them - the values for these places are obtained by halo swapping, i.e. the processes exchange their values since the subvectors are overlapping by 2 places. For all processes but the first (rank r), the 2nd value in the new vector is sent to the previous process (rank $r-1$) where it is placed (received) into its final (Jth) place. Similarly, all processes but the last one send their pen-ultimate value to the next process (rank $r+1$), where it is received into its first place.

After reaching the final time, the first $J-2$ elements of each vector are placed into vector c since the last two values overlap and we don't want the values to be repeated (i.e. we cannot just stick the subvectors together). All the vectors c are gathered using `MPI_Gather`. However, we're still missing the final two values from the solution vector from the last process so these are sent to the final vector separately. At the end, the exact and numerical values are printed along with absolute errors and the numerical solution obtained by serial computation. We can notice that the errors are very small (of order 10^{-6}) and that the parallel and serial solutions match exactly. We also print computational time for each rank starting at the beginning of the numerical scheme and finishing after completing the whole vector for the numerical solution.

The script is compiled on a login node of Cirrus using the script `compile_now` using the Intel compiler.

4. Implementation of the MPI functions

First, we used the function `MPI_Bcast` to broadcast the initial condition vector to all processes

```
MPI_Bcast(IC, M+1, MPI_DOUBLE, 0, comm),
```

where IC is $M+1$-dimensional data copied from the memory of the root process to the memory of other processes in the communicator. The datatype of most vectors is double, so we used `MPI_DOUBLE`.

During halo swapping we used the MPI function for synchronous sending (e.g. this one is for sending the second value to the previous process)

```
MPI_Ssend(&U[1][1], 1, MPI_LONG_DOUBLE, rank-1, 0, comm),
```

and for receiving we used

`MPI_Recv(&U[1][J-1], 1, MPI_LONG_DOUBLE, rank+1, 0, comm, MPI_STATUS_IGNORE),`
(here the value was received from the next process and placed on the last place in the vector). Right before halo swapping and on other critical places we also used `MPI_Barrier(comm)` to avoid confusion in the values and allow for the processes to be in sync.

At the end, the vectors c were gathered using

```
MPI_Gather(&c, J-2, MPI_DOUBLE, Unum, J-2, MPI_DOUBLE, 0, comm);
```

where the $J-2$-dimensional vector c is collected from all the processes and reassembled in the root process into the vector for the final numerical solution `Unum`.

5. Computation time for different number of processes

We experimented with varying the number of processes and number of grid points given to each of them with total $M = 101$, i.e. 102 datapoints.

Processes	5	10	20	25
J	22	12	7	6
Time	0.06544	0.14209	0.31698	0.44642

Intuitively we might except that the calculation would take less time with a larger number of processes, however, we find the opposite to be true. The cause of this may be the fact that the whole computation was not completely parallel due to the swapping happening and due to using the `MPI_Barrier` function which stops all the processes until all of them finish, i.e. the increased time might be caused by waiting for all the processes to finish.