High Performance Computing, Assignment 2

UID: 1690550

March 15, 2017

1 Initial readings, before any changes were made

After a clean make and running the code for the first time, we get the following results from gprof:

function	1
poisson	23.14
compute_tentative_velocity	1.10
compute_rhs	0.12
apply_boundary_conditions	0.02
set_timestep_interval	0.02
update_velocity	0.02

We can see that the poisson function is by far the most expensive one, and so we shall focus on it first.

2 MPI approach

2.1 Domain Decomposition

We have the option to decompose the region in 1 or 2 directions. After some trial and error, we find that attempting to split the region both horizontally and vertically is not very helpful. It creates a lot of problems for communication between processes later, and generally we do not get much improvement this way. Additionally we can observe that the data is contiguous in columns, but not in rows.

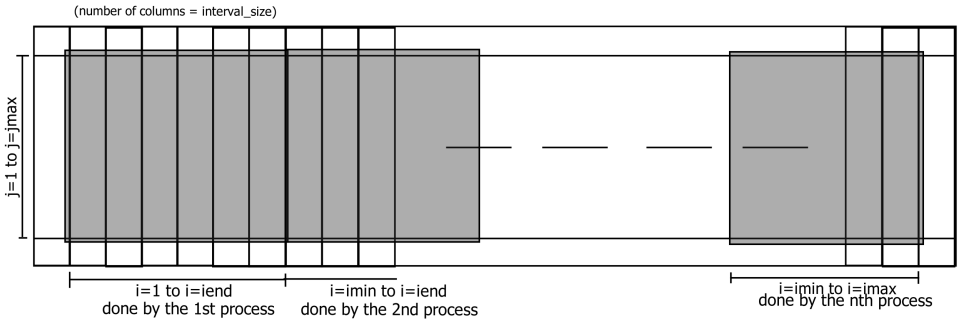

Figure 1: Domain decomposition

Therefore we split the region into vertical chunks, one for each process, as shown in Fig. 1. It is straightforward to calculate the number of chunks necessary. We can find the number of processes using:

```
int nprocs = 0
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

We put that right after initializing MPI and working out number of processes and assigning task ids, just above the main loop. For ease of calculations, we allow only a number of processes by which the size of the array is divisible. Otherwise we would get uneven chunks.

Each process is assigned a chunk consisting of $interval\_size = \frac{imax}{nprocs}$ columns to balance the load.

We had to define new variables, imin and iend to denote the first and the last column of a chunk respectively. They were added as parameters to each function in 'simulation.c' (and its header).

2.2 Passing boundaries

The processes have to exchange boundaries with their neighbours. It is done by passing the column imin to the neighbour process on the left, and iend to the neighbour on the right. Of course the 1st and last process have only a single neighbour, so to exchange boundaries we use the following:

```
if (proc != nprocs -1) {
    MPI_Send(&p[iend], jmax+2, MPI_FLOAT, proc+1, 0, MPI_COMM_WORLD);
    MPI_Recv(&p[iend+1], jmax+2, MPI_FLOAT, proc+1, 1, MPI_COMM_WORLD);
}
```

Similarly for a process to the left, if the process isn't 0, we send column imin, and receive imin-1. This is pictured in Fig. 2.

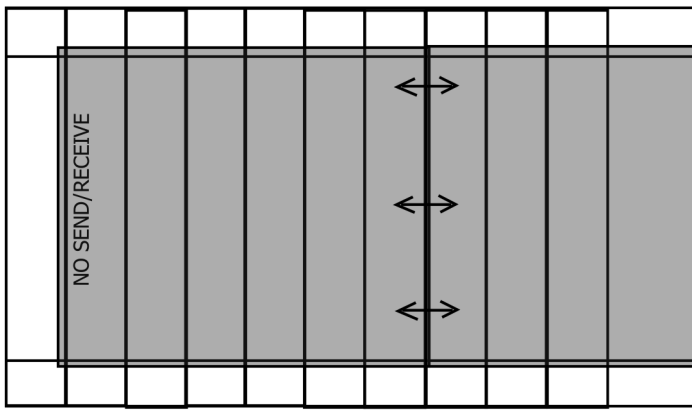

Figure 2: Boundary sharing

2.3 Reductions

There are a few places where reductions need to take place, and they are easy to miss. In each case we create a copy of the variable (naming convention: var_global), to reduce to.

- In poisson() function: p0 is the sum of squares of the values in p. We need MPI_SUM.
- In poisson() function: *res is a more complicated function, so we create a temporary variable and a global version of the latter. Later we can calculate *res using temp_global
- In set_timestamp_interval() function, both umax and vmax can be reduced with MPI_MAX.

2.4 Gathering the data

After the main loop, we need to gather matrices p, u, and v, which are then printed to a file by the MASTER process. It is most efficient to do it in place. We also put an `MPI_Barrier` before gathering, to ensure that the processes have finished their calculations.

We elected to use `MPI_Gatherv`. It requires defining our data pattern. Since we want to gather it on the MASTER process (process 0), we stop it from sending anything to itself, by defining first data chunk to be of size 0 and the following ones of (number of columns in each chunk) $\times$ (jmax+2). The displacement in each case is just the size of the chunk after the previous one.

To avoid creating a copy of the array, which would be slower and memory inefficient, we use `MPI_IN_PLACE` as the parameter for the receiving buffer.

Other than the main loop, there is one more place where gathering data has to happen - at the end of update_velocity() function, output of which is used to apply boundary conditions.

Before each MPI_Gatherv statement, we put a barrier to make sure that all processes have finished their calculations by then.

3 Using OpenMP

First we observe that in the `simulation.c` there are two for loops iterating over the region: one is calculating the u values, the other is calculating the values of v.