Parallelizing a numerical 2D wave equation solver with MPI

IN3200/IN4200 obligatory assignment 2, Spring 2023

Note: This is one of the two obligatory assignments that both need to be approved before a student is allowed to take the final exam. (The grade of the final exam is otherwise *not* related to these two assignments.)

Note: Discussions between the students are allowed, but each student should write her/his own implementations. The details about how to submit the assignment can be found at the end of this document.

1 Objectives

This obligatory assignment has the following objectives:

- The student is given a hands-on exercise of parallelizing a numerical code based on a division of the data and computation, aiming for execution on distributed-memory parallel computers.
- The student is asked to implement the parallelization and the necessary inter-process communication using appropriate MPI function calls.
- The student is given a hands-on exercise of compiling and running the MPI-parallelized code.

2 The 2D wave equation solver

The numreical recipe for the serial solver of the 2D wave equation was given in detail for the first obligatory assignment. (Please refer to the text of the first assignment.) The same numerical recipe should be used for this second assignment.

3 The assignment

3.1 Division of work and data

The computational work is to be divided between P processes (where the value of P should not be hardcoded, but prescribed at runtime). For simplicity, the entire 2D uniform $n_y \times n_x$ grid is to be decomposed as horizontal "stripes". That is, each process is responsible for computing a contiguous block of rows. The total number of rows, n_y, may not be divisible by P, but the work division should be as even as possible. No process should create arrays that cover the entire $n_y \times n_x$ grid. Instead, each process works on a subgrid of $n_y^{\text{my}} \times n_x$ points. (Note: The value of n_y^{my} may differ from process to process, when n_y is not divisible by P.)

It is suggested that the MPI process with rank 0 is responsible for the bottom stripe, whereas the MPI process with rank $P-1$ is responsible for the top stripe. For the ease of implementation of the necessary communication, it is suggested that processes 0 and $P-1$ each is extended with one layer of ghost points, whereas the other processes each is extended with two layers of ghost points (for the two neighbors from below and above).

3.2 Programming

The student is required to implement the following four functions. Each function should be placed inside a separate file, using the same file name as the function name. The syntax of these functions should be as follows:

```
void communicate_above_below (int my_rank, int P, int nx, int my_ny, double **my_u);

void subg_first_time_step (int my_rank, int P, int nx, int my_ny, double dx, double dy, double dt,
                           double **my_u, double **my_u_prev);

void subg_one_fast_time_step (int my_rank, int P, int nx, int my_ny, double dx, double dy, double dt,
                              double **my_u_new, double **my_u, double **my_u_prev);

double all_compute_numerical_error (int my_rank, int my_offset, int P, int nx, int my_ny,
                                    double dx, double dy, double t_value, double **my_u);
```

Function `communicate_above_below`. It should let each process receive from its two neighbors from above and below the values needed on the ghost layers of points. In return, each process should send values to the neighbors for their corresponding ghost ponts. (Note: Processes 0 and $P-1$ each has only one neighbor.)

Function `subg_first_time_step`. It should let each process compute the first time step (see assignment one), restricted to its subgrid.

Function `subg_one_fast_time_step`. It should let each process compute a subsequent time step (see assignment one), restricted to its subgrid.

Function `all_compute_numerical_error`. This function should returns on all the processes the computed numerical error that is the same as the serial version `compute_numerical_error` (see assignment one) would have returned on a process that has the entire solution. (Please see the skeleton of the parallel main program below for the meaning of `my_offset`.)

Moreover, each student should implement an MPI-parallelized main function (in a file named `mpi_main.c`) with the following skeleton:

```
int main (int nargs, char **args)
{
  int nx = 1001, ny = 1001; double T = 2.0; // default values
  int i,j;
  double dx, dy, dt, t;
  double **my_u, **my_u_new, **my_u_prev, **my_tmp_ptr;

  int my_rank, my_offset, P, has_neigh_below, has_neigh_above;

  MPI_Init (&nargs, &args);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &P);

  if (my_rank==0) {
    if (nargs>1)  // if a new value of nx is provided on the command line
      nx = ny = atoi(args[1]);
    if (nargs>2)  // if a new value of T is provided on the command line
      T = atof(args[2]);
  }

  // let process 0 broadcast values of nx, ny and T to all other processes
```

```
  // ....

  has_neigh_below = (my_rank>0) ? 1 : 0;
  has_neigh_above = (my_rank<P-1) ? 1 : 0;
  my_offset = (my_rank*ny)/P;
  my_ny = ((my_rank+1)*ny)/P - my_offset;

  allocate_2D_array (&my_u, nx, (my_ny + has_neigh_below + has_neigh_above));
  allocate_2D_array (&my_u_new, nx, (my_ny + has_neigh_below + has_neigh_above));
  allocate_2D_array (&my_u_prev, nx, (my_ny + has_neigh_below + has_neigh_above));

  dx = 1.0/(nx-1); dy = 1.0/(ny-1);
  dt = 2.0*dx;  // maximum value allowed for the time step size

  // start timing
  // ...

  // set initial condition
  for (i=0; i<my_ny; i++)
    for (j=0; j<nx; j++)
      my_u_prev[i+has_neigh_below][j] = cos(2.0*Pi*j*dx)*cos(2.0*Pi*(i+my_offset)*dy);

  communicate_above_below (my_rank, P, nx, my_ny, my_u_prev);
  subg_first_time_step (my_rank, P, nx, my_ny, dx, dy, dt, my_u, my_u_prev);

  // compute the remaining time steps
  t = dt;
  while (t<T) {
    t += dt;
    communicate_above_below (my_rank, P, nx, my_ny, my_u);
    subg_one_fast_time_step (my_rank, P, nx, my_ny, dx, dy, dt, my_u_new, my_u, my_u_prev);
    /* pointer swaps */
    // ....
  }

  printf("my_rank=%d, nx=%d, my_ny=%d, T=%g, dt=%g, error=%e\n",my_rank,nx,my_ny,t,dt,
         all_compute_numerical_error(my_rank,my_offset,P,nx,my_ny,dx,dy,t,my_u));

  // stop timing
  // ...

  // deallocate arrays my_u_new, my_u, my_u_prev
  // ....

  MPI_Finalize ();

  return 0;
}
```

4 Submission

Each student is required to submit, via Devilry, a zip file or a tarball of a folder named `oblig2_src_xxx` where `xxx` is your UiO username. (**Note: Don't reveal your candidate number.**) The folder of `oblig2_src_xxx` should contain two sub-folders: `serial_part` and `mpi_part`, where the former can contain the source code of `allocate_2D_array` and `deallocate_2D_array` that were implemented for

the first assignment. The sub-folder of `mpi_part` should contain the source code of the above four functions (one per file) and `mpi_main.c`, possibly other files. In addition, the sub-folder of `mpi_part` should also contain a `README.txt` or `README.md` file that describes how the parallel compilation and execution can be done. There is no need to write a report.

The submission deadline is Monday May 1st, 23:59.

Hint: In case you don't have a proper computer to develop and test your implementations, please use the "Fox" system that is provided on EduCloud at UiO. (Please read the "Advice" webpage on the IN3200 semester website.)