CS-E4690 Programming Parallel Supercomputers
Sheet 3 Report

Nguyen Xuan Binh 887799

Part 2: A physical application case

Exercise instruction:

*Typical application cases of stencil computations are partial differential equations. A physical application case of passive scalar transport is described in detail in the document `phys_appl.md`. Your task is to complete the code template `advec_wave_2D_skel.c` to obtain a working 2D-solver for that problem using **one-sided** MPI communication.*

This requires
- defining a mapping of the $N$ MPI processes (ranks) to the $N$ equally-sized subdomains, into which the computational domain is decomposed
- figuring out the neighboring relationships of the MPI processes and implementing corresponding functions (see code template)
- establishing MPI windows
- choosing a scheme of non-blocking communication
- defining a convenient data type for MPI_Get or MPI_Put
- splitting the evaluation of the right-hand-side of the PDE to allow maximum concurrency with the communication
- verifying the obtained solution against the analytical one.

Establish the level of concurrency by timing runs with communication and time integration against runs with communication only and with time integration only (which would be physically incorrect, of course).
Write a short report, accounting on your design decisions and the results (solution verification and concurrency).

Dear professor, I have attempted this assignment with my best efforts, but it is much harder than my expectations. Therefore, I won't have time to do every bullet point down here, but at least I know that my implementation is close to the correct one, as there is something flawed in my code. I hope I can receive feedback on my assignment and thank you in advance.

Now, I will visit each point and list out the relevant code in my implementation

1st: defining a mapping of the N MPI processes (ranks) to the N equally-sized subdomains, into which the computational domain is decomposed

The relevant code of mine is

```
  MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   // ... (other code)

   nprocx = atoi(argv[1]); // process numbers in x directions
   nprocy = atoi(argv[2]);  // process numbers in y directions

   // ... (other code)

   int *proc_coords = find_proc_coords(rank, nprocx, nprocy);
   int ipx = proc_coords[0]; int ipy = proc_coords[1];

   // ... (other code)

   int domain_nx = atoi(argv[3]);  // Number of gridpoints in the x direction
   int domain_ny = atoi(argv[4]);  // Number of gridpoints in the y direction

   // ... (other code)

   int subdomain_nx = domain_nx / nprocx;   // subdomain x-size without halos
   int subdomain_ny = domain_ny / nprocy;   // subdomain y-size without halos
```

2nd: Figuring out the neighboring relationships of the MPI processes and implementing corresponding functions

```
// Create Cartesian communicator
MPI_Comm cart_comm; // Cartesian communicator
int dims[2] = {nprocx, nprocy}; // dimensions of the grid
int periods[2] = {1, 1}; // periodic boundary conditions in both dimensions
```

```
// Create a 2D Cartesian grid
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &cart_comm);

// Variables to store the Cartesian topology information
int coords[2];

// Get the Cartesian topology information for the current process
MPI_Cart_get(cart_comm, 2, dims, periods, coords);

// Now, coords[0] and coords[1] hold the x and y coordinates of the current process in the grid
int my_x_coord = coords[0];
int my_y_coord = coords[1];

// Determine the ranks of the neighboring processes
int north_neighbor_rank, south_neighbor_rank, east_neighbor_rank, west_neighbor_rank;

// MPI_Cart_shift virtually moves the cartesian topology of a communicator (created with
MPI_Cart_create)
// in the dimension specified. It permits to find the two processes that would respectively reach,
// and be reached by, the calling process with that shift. Shifting a cartesian topology by 1 unit
// (the displacement) in a dimension therefore allows a process to find its neighbours in that
dimension.
// In case no such neighbour exists, virtually located outside the boundaries of a non periodic
dimension
// for instance, MPI_PROC_NULL is given instead.

MPI_Cart_shift(cart_comm, 0, 1,
            &south_neighbor_rank, &north_neighbor_rank);

MPI_Cart_shift(cart_comm, 1, 1,
            &west_neighbor_rank, &east_neighbor_rank);
```

Due to the periodic nature of the field, the MPI_Cart_Shift has derived the neighbors as follows. You can verify this in my prog.out file

ipy / ipx	0	1	2
2	Rank 6	Rank 7	Rank 8
1	Rank 3	Rank 4	Rank 5
0	Rank 0	Rank 1	Rank 2

Rank 6, ipx = 0, ipy = 2 Rank 6 north_neighbor_rank: 0 Rank 6 south_neighbor_rank: 3 Rank 6 east_neighbor_rank: 7 Rank 6 west_neighbor_rank: 8	Rank 7, ipx = 1, ipy = 2 Rank 7 north_neighbor_rank: 1 Rank 7 south_neighbor_rank: 4 Rank 7 east_neighbor_rank: 8 Rank 7 west_neighbor_rank: 6	Rank 8, ipx = 2, ipy = 2 Rank 8 north_neighbor_rank: 2 Rank 8 south_neighbor_rank: 5 Rank 8 east_neighbor_rank: 6 Rank 8 west_neighbor_rank: 7
Rank 3, ipx = 0, ipy = 1 Rank 3 north_neighbor_rank: 6 Rank 3 south_neighbor_rank: 0 Rank 3 east_neighbor_rank: 4 Rank 3 west_neighbor_rank: 5	Rank 5, ipx = 2, ipy = 1 Rank 5 north_neighbor_rank: 8 Rank 5 south_neighbor_rank: 2 Rank 5 east_neighbor_rank: 3 Rank 5 west_neighbor_rank: 4	Rank 4, ipx = 1, ipy = 1 Rank 4 north_neighbor_rank: 7 Rank 4 south_neighbor_rank: 1 Rank 4 east_neighbor_rank: 5 Rank 4 west_neighbor_rank: 3
Rank 2, ipx = 2, ipy = 0 Rank 2 north_neighbor_rank: 5 Rank 2 south_neighbor_rank: 8 Rank 2 east_neighbor_rank: 0 Rank 2 west_neighbor_rank: 1	Rank 1, ipx = 1, ipy = 0 Rank 1 north_neighbor_rank: 4 Rank 1 south_neighbor_rank: 7 Rank 1 east_neighbor_rank: 2 Rank 1 west_neighbor_rank: 0	Rank 0, ipx = 0, ipy = 0 Rank 0 north_neighbor_rank: 3 Rank 0 south_neighbor_rank: 6 Rank 0 east_neighbor_rank: 1 Rank 0 west_neighbor_rank: 2

3rd: Establishing MPI windows

```
MPI_Win win; // MPI window object

MPI_Aint size = sizeof(float) * subdomain_mx * subdomain_my; // size of the MPI window = 4 *
52 * 52 = 10816 bytes
int disp_unit = sizeof(float);  // displacement unit = 4 bytes

// Create the MPI window
MPI_Win_create(data, size, disp_unit, MPI_INFO_NULL, MPI_COMM_WORLD, &win);

// MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
//               MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

4th: Choosing a scheme of non-blocking communication. Defining a convenient data type for MPI_Get or MPI_Put

```
// Define MPI datatypes for a column
MPI_Datatype column_type;

// Define MPI datatypes for a row
MPI_Datatype row_type;

// MPI vector type for fetching a column
MPI_Type_vector(subdomain_ny, 1, subdomain_mx, MPI_FLOAT, &column_type);
MPI_Type_commit(&column_type);

// MPI contiguous type for fetching a row
MPI_Type_contiguous(subdomain_nx, MPI_FLOAT, &row_type);
MPI_Type_commit(&row_type);
```

5th: Splitting the evaluation of the right-hand-side of the PDE to allow maximum concurrency with the communication

```
// Start computation that does not depend on the received halo data
// For example, compute RHS for the interior points of the subdomain
int interior_xrange[2] = {halo_width + 1, halo_width + subdomain_nx - 1};
int interior_yrange[2] = {halo_width + 1, halo_width + subdomain_ny - 1};
rhs(interior_xrange, interior_yrange, subdomain_my, data, d_data);

// End the RMA epoch to complete fetching operations
MPI_Win_fence(0, win);

// Now that halo data is available, compute RHS for the boundary points of the subdomain
int boundary_xrange[2] = {halo_width, halo_width + subdomain_nx};
int boundary_yrange[2] = {halo_width, halo_width + subdomain_ny};
rhs(boundary_xrange, boundary_yrange, subdomain_my, data, d_data);
```

6th: Verifying the obtained solution against the analytical one.

This is the part that I find confusing the most. Convergence depends on the domain_nx, domain_ny and also on the number of processes nprocs. I am using first order scheme, and none of my configurations manage to converge at all

This is my first configuration

```
#SBATCH --ntasks-per-node=4
# 5 arguments: nprocx, nprox, domain_nx, domain_ny, iterations
time srun advec_wave_2D_skel 2 2 4 4 10
```

```
Iteration 0 Global total error: 2.838478

Iteration 1 Global total error: 2.279956

Iteration 2 Global total error: 2.504887

Iteration 3 Global total error: 2.654073

Iteration 4 Global total error: 2.700539

Iteration 5 Global total error: 2.923237

Iteration 6 Global total error: 3.007751

Iteration 7 Global total error: 3.049272

Iteration 8 Global total error: 3.189360

Iteration 9 Global total error: 3.471519
```

```
#SBATCH --ntasks-per-node=9
# 5 arguments: nprocx, nprox, domain_nx, domain_ny, iterations
time srun advec_wave_2D_skel 3 3 6 6 10
```

```
Iteration 0 Global total error: 5.725004

Iteration 1 Global total error: 5.328478

Iteration 2 Global total error: 5.351368

Iteration 3 Global total error: 5.315886
Iteration 4 Global total error: 5.467592

Iteration 5 Global total error: 6.112571

Iteration 6 Global total error: 5.820673

Iteration 7 Global total error: 5.989224

Iteration 8 Global total error: 6.767714

Iteration 9 Global total error: 6.519992
```

However, there can be some values for domain_nx and domain_ny, the divergence becomes so fast, pushing the global total error to infinity.

```
#SBATCH --ntasks-per-node=4
# 5 arguments: nprocx, nprox, domain_nx, domain_ny, iterations
time srun advec_wave_2D_skel 2 2 50 50 10
```

```
Iteration 0 Global total error: nan

Iteration 1 Global total error: nan

….
```

Therefore, I can conclude that my implementation is still flawed somewhere, but I no longer have time to debug.

7th: Establish the level of concurrency by timing runs with communication and time integration against runs with communication only and with time integration only

To carry out this analysis, we would run program in three different configurations:

First, with both computation and communication.
Second, with only communication.
Third, with only computation.

However, due to time limitations, I can only conduct the first one. Here is the result of my timing for RHS computation and communication using MPI_Get

```
Your configurations

nprocx = 2, nprocy = 2
Rank 0, ipx = 0, ipy = 0
domain_nx = 5000, domain_ny = 5000
subdomain_nx = 2500, subdomain_ny = 2500
subdomain_mx = 2502, subdomain_my = 2502
dx = 0.001257, dy = 0.001257
dt= 0.000377


Iteration 0
Global total error: 3.299872
Total communication time: 0.242061 seconds
Total computation time: 0.000000 seconds

Iteration 1
Global total error: 4.569288
Total communication time: 0.181384 seconds
Total computation time: 0.500000 seconds

Iteration 2
Global total error: 5.818986
Total communication time: 0.160422 seconds
Total computation time: 0.500000 seconds

Iteration 3
Global total error: 7.269196
Total communication time: 0.145867 seconds
Total computation time: 0.500000 seconds

Iteration 4
Global total error: 9.499763
Total communication time: 0.220601 seconds
Total computation time: 0.500000 seconds

Iteration 5
Global total error: 12.446865
Total communication time: 0.182978 seconds
Total computation time: 0.500000 seconds

Iteration 6
Global total error: 15.830035
Total communication time: 0.216672 seconds
Total computation time: 0.500000 seconds

Iteration 7
Global total error: 20.648422
Total communication time: 0.168804 seconds
Total computation time: 0.000000 seconds

Iteration 8
Global total error: 26.762169
Total communication time: 0.229327 seconds
Total computation time: 1.000000 seconds

Iteration 9
Global total error: 34.815811
Total communication time: 0.155771 seconds
Total computation time: 0.500000 seconds
Rank 2, ipx = 0, ipy = 1
Rank 3, ipx = 1, ipy = 1
Rank 1, ipx = 1, ipy = 0
```

I look forward to receiving feedback from you. I really appreciate if my efforts are worthy of significance in your grading scheme.