# ACSE-6: MPI Programming Assignment – Solving the Wave Equation

Github handle: acse-jy220
Date: 22/04/2021

# 1. Code structure

## 1.1 Simulation code

- **'read_parameters.h', 'read_parameters.cpp'**

A "read_parameters" class is defined in here, which reads the parameters defined in 'parameters.ini', you are free to adjust the parameters between simulation runs without recompilation.(don't forget to clear the outputs of last run using 'clear_output.sh'!)

- **'wave.h', 'wave.cpp'**

The main part of the my algorithm is located here (a 'wave_data' class and its memeber functions). after getting parameters from a 'read_parameters' class, I create and allocate the grids to each processor according to the number of processor given, then we set initial condition parallelly, build MPI communications, and run the simulation parallelly.

- **'main.cpp'**

The main function to run simulation is here. You could notice that I build some structures like communication parameters and Datatypes in one go, and then run the simulation based on this fixed structure.

## 1.2 Postprocessing code

- **'loaddata.py'**

It load parameters from the 'parameters.ini' as well as binary files in the "./output" folder.

- **'pic.py'**

It would generate a single shot of the wave simulation from a binary file.

- **'animation.py'**

It accesses all the binary files in the "./output" folder, and create an animation for the simulation, also based on the parameters in 'parameters.ini', the output video would have the same time scale as the real one.

## 1.3 other files

- **'parameters.ini'**

A config file which stores the parameters for the simulation, free to change.

- **'clear_output.sh'**

A script to delete all the binary files in "./output" folder.

# 2. Parallelling stategy (algorithms in 'wave.cpp')

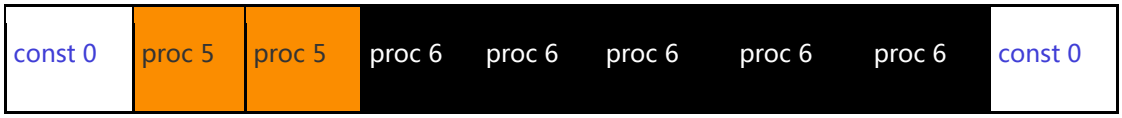
## 2.1 Allocating the grid to different processors

The grid allocation is done by allocate_grid() function in 'wave.cpp', for example, if a grid of size $n = i_{max} \times j_{max}$ is given, and we have $p$ processors, then the processor number $id$ would have $\frac{n - n \bmod p}{p} + \omega(id < n \bmod p)$ grids, where $\omega = 1$ if the statement inside the bracket is true, and 0 otherwise. And the start index for processor $id$ is $id \cdot \omega(id < n \bmod p) + n \bmod p \cdot \omega(id \geq n \bmod p) + id \cdot \frac{n - n \bmod p}{p}$, under which case, the grid would be devided into strip-like regions.

Because I use a row-major ordering deviding, there may be some problems that some processor (like processor 0 and p-1) have all of their responsiable grids located on the boundary (not very much interactive with others in the simulation). Here I come up with a stategy, if the boundary type is 'Neumann' or 'Dirchlet', I just simply trim the grid down 2 units and only allocate the interior to the processors, $i_{max} = i_{max} - 2, \quad j_{max} = j_{max} - 2$, also I create extra arrays to store the boundary values in the 'wave_data' class. Because the grids at the boundary only interacts with its nearest neighbour throughout the whole process, I just allocate them (those grid on the boundary) to the processor at which their nearest neighbour locates, which avoids extra communication.

See the following example, deviding a $7 \times 9$ grid to 6 processors (Neumann or Dirichlet boundary, the corner are constantly set as zero):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| const 0 | proc 1 | proc 1 | proc 1 | proc 1 | proc 1 | proc 1 | proc 2 | const 0 |
| proc 1 | proc 1 | proc 1 | proc 1 | proc 1 | proc 1 | proc 1 | proc 2 | proc 2 |
| proc 2 | proc 2 | proc 2 | proc 2 | proc 2 | proc 2 | proc 3 | proc 3 | proc 3 |
| proc 3 | proc 3 | proc 3 | proc 3 | proc 3 | proc 4 | proc 4 | proc 4 | proc 4 |
| proc 4 | proc 4 | proc 4 | proc 4 | proc 5 | proc 5 | proc 5 | proc 5 | proc 5 |
| proc 5 | proc 5 | proc 5 | proc 6 | proc 6 | proc 6 | proc 6 | proc 6 | proc 6 |




Of course, the interior grids and the boundary grids are stored in different arrays within one processor. The allocate_grid() is mainly just performing this task, and once the allocation is done, the build_OutputType() function would create MPI_Datatypes for output (use for MPI_File_write) based on this structure. In my opinion, the only inbalance workload allocation in this method is that, the processor having more grids at a boundary need to do more copying for Neumann_boundary condition every iteration, but I think a single copying operation at one boundary won't be a very big work load even for a 10000 * 10000 grid.

## 2.2 Storing time steps

Something to note here is that I only save three time steps, namely 'u', 'new_u' and 'old_u' through the simulation. In addition, instead of doing pointer swapping, I just keep them fixed. From iteration 0 (initial state), I just set 'old_u' = 'u', and update 'new_u' depend on those two at iteration 1, then I update 'old_u' with 'new_u' being the current state and 'u' to be the state before at iteration 2, after that, I come back to update 'u' with 'old_u' being the current state and 'u' to be the state before at iteration 3, and so_on. Thus, the newest valid data is stored in:

$u \rightarrow new\ u \rightarrow old\ u \rightarrow u \rightarrow \cdots$, based on this storing scheme, the MPI_grid_to_file() function in 'wave.cpp' would output values of one of these three arrays based on the current iteration number. It is simply calculating the remainder of the iter number w.r.t 3 to decide which array to print. Also, the set_Neumman_boundary() functions are set for these three arrays separately.

## 2.3 Find neighbouring processors/isolated/non-siolated indexes

To build communications, first I created 12 double pointers "left_1", "right_1","up_1", "down_1", "left_2"...etc, they stores the addresses of the 4 neighbours of a certain interior grid that the node is responsiable, the number 1, 2, 3 just refers to getting address from 'u', 'new_u' or 'old_u'. Then I implement the find_neighbour_for_other_processor()
function to find the neighbours of the grids inside one processor, and if that neighbour (for instance right neighbour) is stored within the current processor (whether it's stored in the array for iterior points or the arrays for boudary values), I directly set the "right_1", "right_2", "right_3" pointers to the array with corresponding indexes. If its neighbour is allocated to other processor, I append this index to the vector container index_left_for_processor[another_proc].

Another job done by the find_neighbour_for_other_processor() function is that it justifies whether a interior grid is "isolated" or not. That is, if a grid has one of its 4 neighbour is in other processor, then it's "not isolated", otherwise, it is "isolated".

## 2.4 Build MPI_Datatypes for communication

Having got those information, we could build MPI_DataTypes for communication in the

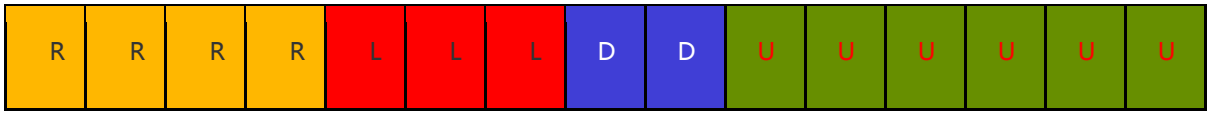
build_communication_type() function. First, we create 3 double MPI_DataType pointers named "Send_to_neighbour", "Send_to_neighbour2", "Send_to_neighbour3" with length $p$ for three time step arrays 'u', 'new_u' and 'old_u'.

We loop over the processor numbers, first do a calculation that send_length[i] = index_right_for_processor[i].size() + index_left_for_processor[i].size() \
+ index_down_for_processor[i].size() + index_up_for_processor[i].size();

if the result is zero, that means the current process $id$ will have no communication with processor $i$, we just skip it. If send_length[i] is not zero, we will go over those four vector containers, in the squence of index_right_for_processor[i], index_left_for_processor[i], index_down_for_processor[i], index_up_for_processor[i] and record the corresponding MPI_Aints for indexes with in these containers to build the MPI_DataTypes "Send_to_neighbour[i]", "Send_to_neighbour2[i]", "Send_to_neighbour3[i]". At the same time, we can configure a receive array for communication bewteen current process $id$ and processor $i$, such that receive_from_neighbour[i] = new double[send_length[i]], the same length as our addresses to sent to processor $i$ actually. But very important to note here, we need to have a subtle swap of places of indexes when setting the receive array.

What will be the array sent from processor $i$ to process $id$ (by symmetry)?

It looks like:




the yellow is the right neighbours for indexes in the container index_left_for_processor[i], the red is the the left neighbours for indexes in the container index_right_for_processor[i], the blue is the down neighbours for indexes in the container index_up_for_processor[i] and the green is up neighbours for indexes in the container index_down_for_processor[i]. The reason is when we configure MPI_Datatypes we have set that the sequence of the sending array would always be

$$right\ neighbours|left\ neighbours|down\ neighbours|up\ neighbours$$

for the recieving processor, but also to note with in our node, actually it's the indexes in index_left_for_processor[i] who receive their right neighbours from processor $i$, so a index swapping is needed here. We need to set the left neighbours for the indexes in container index_right_for_processor[i] with a displacement etc. In my code this looks like:

left_1[index_right_for_processor[i][len_vec]] = &receive_from_neighbour[i][com_len + index_left_for_processor[i].size()] and right_1[index_left_for_processor[i][len_vec]] = &receive_from_neighbour[i][com_len - index_right_for_processor[i].size()]

After we configure the send datatypes and receive arrays, we could then build non-blocking communications based on them.

## 2.5 Set up non-blocking communications and do iterations

The functions we stated before can all be viewed as configuration functions, as they are only executed

once before the first iteration taken place.

Next I will introduce functions involved in each iteration, they are do_communication()

and do_iteration().

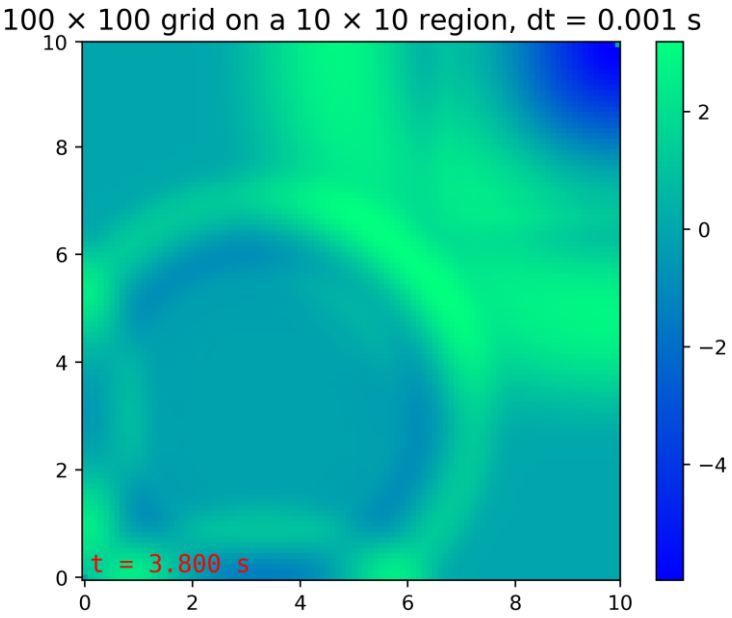
For do_communication() function, we would set up at most $2 \cdot (p-1)$ non-blocking communications for processor, but to note as we have calculated a vector send_length before where if send_length[i] = 0 indicates there will no communications happen bewteen current processor with that processor $i$, thus we can add a judging statement to just skip this, we could use a 'cnt' counter to record the total communication numbers needed. So actually much less than $2 \cdot (p-1)$ communications would be set for each processor at every iteration.

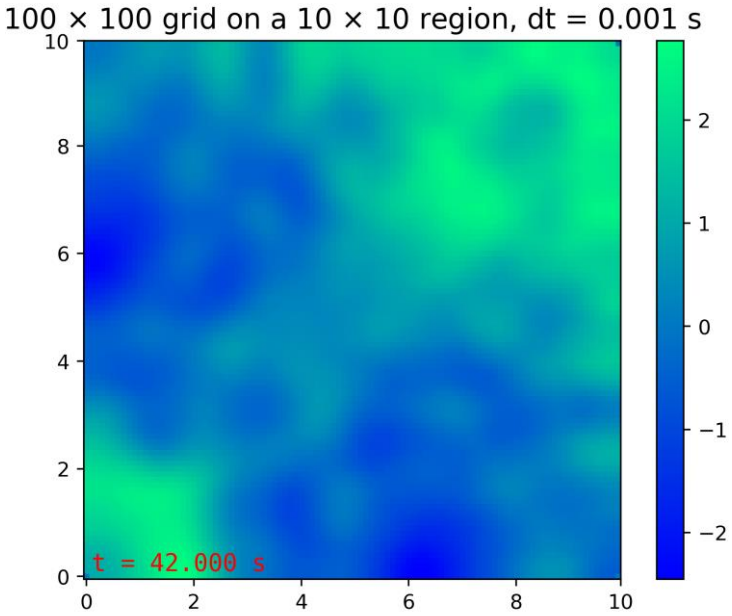
In do_iteration() function, first we calculate the remainder of the iteration number w.r.t to 3 to decide which one of 'u', 'new_u' and 'old_u' need to be update at this iteration.

Then remember we have find out the **isolated** grids at each processor, which means all of its neighbours are stored somewhere within this processor, so we could update them first without waiting the non-blocking communications to finish. Then we should do MPI_Waitall(cnt, requests, MPI_STATUS_IGNORE) to ensure our receive_from_neighbour array has received all the neighbour values needed for **non-isolated** grids and until then can we update those **non-isolated** grids.

# 3. Outputs

For detail process about how to execute the program and generate a output picture/animation for the simulation, please refer to the guide in 'ReadMe.md'.
Also three example videos are placed in the "./anime" folder.






The pictures above show intermediate shots for the intial condition being two splashs of radius 1.12381 and 3.2313, locating at (3.12312, 3.31312) and (9.2313, 9.12312) respectively, with Neumann boundary condition.

# 4. Performance analysis

Time used for 100 * 100 grid with dt = 0.001s , t_end = 10/50/100/500 s (10000/50000/... iterations) **Neumann** boundary condition: (in seconds, omiting time for outputs)

| iterations\num.proc | serial | 2 | 4 | 6 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| 10000 | 11.778 | 10.838 | 3.123 | 2.3723 | 1.641 | 0.8045 | 0.5104 | 0.4021 |
| 50000 | 28.536 | 21.103 | 7.9159 | 5.3547 | 4.1706 | 3.6836 | 2.2597 | 1.7888 |
| 100000 | N/A | 48.308 | 16.447 | 11.298 | 8.78 | 7.7244 | 4.8713 | 3.7978 |
| 500000 | N/A | 253.79 | 80.894 | 56.583 | 46.37 | 40.647 | 24.12 | 19.40 |

We could see that for a fixed number of procssor, the total time taken for the simulation just increases

almost linearly with the total iterations, which also proves that the initial configurations of our algorithm does not have a great contribution in the overall simulation.

1)We extend to grid size to 1000 * 1000 grid with same dt , also with Neumann boundary condition:

| iterations\num.proc | serial | 2 | 4 | 6 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| 10000 | > 6min | > 6min | > 6min | 311.28 | 200.49 | 136.02 | 75.176 | 48.644 |

We could see as the grid size increases by 10 * 10 = 100 times, the execution time for same iterations also increase by about 100 times, which leads by the "data paralleling" scheme.

2) Let's see how the shape influence our performance, like for thin or fat rectangles, first we change to grid to a 'thin' grid of 10000 * 100 (dt change to 1/10, but t_end also to 1/10 to keep a same iteration number):

| iterations\num.proc | serial | 2 | 4 | 6 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| 10000 | > 5min | > 5min | > 5min | 229.51 | 154.97 | 85.347 | 47.182 | 29.962 |

and 'fat' grid of 100 * 10000:

| iterations\num.proc | serial | 2 | 4 | 6 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| 10000 | > 5min | > 5min | > 5min | > 5min | 216.26 | 155.79 | 86.262 | 54.907 |

From the above two tables we could observe that our implementation could better the thin rectangles, the reason for this should be the allocation we defined in allocate_grid() function in 'wave.cpp' is deviding a square grid with row-major ordering, which cause some cores have too many boundary values to hold, and copying (Neumann boundary). An improvement to this could be smartly choose row-major ordering or column-major ordering when detecting the input imax and jmax (row-order for 'thin', column-order for 'fat'). However, this implementation would also influence many functions involved in

the later process and other extra work need to be done.

3) Now let we come back to the 1000 * 1000 square grid, but swap the boundary condition to

**Dirichlet** condition:

| iterations\num.proc | serial | 2 | 4 | 6 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| 10000 | > 5min | > 5min | > 5min | 177.76 | 152.56 | 105.24 | 67.552 | 44.543 |

We could see that **Dirichlet** condition is fairly 'cheaper' in computational cost than **Neumann** boundary condition, that's mainly because the **Neumann** boundary condition need to update boundary values every iteration, while in Dirichlet condition we could fix the boundary value with constant zeros.

**periodic** condition:

| iterations\num.proc | serial | 2 | 4 | 6 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| 10000 | > 5min | > 5min | 217.52 | 151.51 | 118.68 | 93.948 | 51.939 | 28.283 |

We see under same parameters, our program deal with the periodic boundary faster than the other two boundary types. This may because, in **periodic** condition, the number of grids allocate to each processor is the most balance (every processor is responsible for almost equal number of grids, and do about equal number of work).

However, for **Dirichlet** boundary or **Neumann** boundary, that's not the case. Recall my strategy of deviding grids, some nodes take response of most grids at the boundary. (like processor 0 in row-major ordering). This would cause those processor doing more work every iter under **Neumann** condition (copying) and doing less work under **Dirichlet** boundary condition (discard constant zeros).

## Some conclusion

1. The algorithm designed in this project has a 'configure once, run on this cofiguration all through" feature, and by experiment, the initial configurations does not have a great contribution in the time

cost.

That's to say, in a long run simulation, the time cost would just be almost proportional to the total time period you are simulating.

2. Because of the design of my allocation scheme, the program has better performance with 'thin' grids (whose jmax is larger then imax) than square or 'fat' grids (whose imax is larger then jmax) and has better performance with periodic boundary condition than Dirichlet boundary condition or Neumann boundary condition.