Distributed memory parallelism with MPI

Xiaoyue Xiao (oa19132)

1. *Introduction*

The purpose of this assignment is to use MPI to parallelize the optimized stencil code on the previous assignment. Some important MPI communications first will be discussed. Next, this report will introduce how to decompose the original image along columns and rows respectively and how to distribute specific values to each process. Then, this article will describe the halo exchange which helps each process gets data from neighboring processes. The process of collecting data from all processes and timing also will be mentioned. Finally, an explanation of performance improvement and data analysis for numbers of cores will be shared.

2. *MPI communications*

MPI is a standard message passing interface that is enormously used on parallel computers [1]. MPI communications mainly include point-to-point and collective communications.

To make sure messages correctly sent and received, the blocking point-to-point communications are used in this assignment. MPI_Send call can transfer messages from one rank to another rank. MPI_Recv call can receive messages from other ranks and it blocks until the message has been received. MPI_Sendrecv call can send messages and receive messages at the same time.

In addition to point-to-point communications, one of the most important collective communications is also applied in this assignment which is MPI_reduce. The function of MPI_Reduce is to perform a binary operation on the data in every process sharing the same communicator [2].

3. *Domain decomposition*

In the beginning, the size of the input image is set to nx (the number of rows) * ny (the number of columns). This image should be surrounded by zero for the 5-point stencil function, which results to the new size of the image. The width of new image is ny+2 and the height is nx+2. After the code allocates an array with the size of y+2 * nx+2, The image is initialized to a checkerboard pattern by calling *init_image()* function.

After that, in order to use MPI communications, the *image* grid will be partitioned into N sub-grids according to the number of processes N. Because of the requirement of this assignment, two nodes should be used in the code and each node has 28 cores. Therefore, the number of total ranks is 56.

The grid will be equally divided into 56 parts. However, sometimes there is a remainder when the number of rows or columns over the number of processes. To solve this problem, the remainder will be added to the last rank.

3.1 Column decomposition

According to Figure 1, the input image is partitioned into N sub-grids by columns. Each rank has the same size except the last rank. The *calc_ncols_from_rank()* function will calculate the unique number of columns for each rank. Each rank will get a unique part of x-dimension and the full extent of y-dimension. The last rank has the most columns distributed because it will be added the extra remainder.

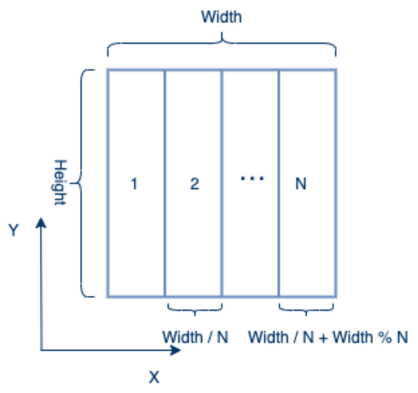

Figure 1: Decompose the image by columns.

The code will allocate an new array for each rank. The first column and the last column of the image are zero columns. After decomposing the image, the first column of the first rank and the last column of the last rank will be zero columns. In the next steps, all ranks have to do halo exchange. As a consequence, each rank should be added to a halo region. The first rank and the last rank have already been assigned a column of zeros, hence those two ranks just need to be added an extra column but other ranks need to be added two extra columns.

When the local grid (current rank) has been created, the core cells of the grid are set to the corresponding values of the initialized image, and halo cells are set to the zero value.

3.2 Row decomposition

The writer first tried to decompose the input image along columns. However, it is inconvenient to assign corresponding values to each rank because of its row-major data layout. Therefore, the idea that the input image is decomposed to N parts along rows is generated.

As can be seen in Figure 2, each rank has the same width and new height. The last rank has the most rows partitioned among ranks.

According to Table 1, The runtime of decomposing by rows is shorter than decomposing by columns because the former can use continuous assignments. In the following sections of this report, row decomposition will only be mentioned instead of column decomposition.

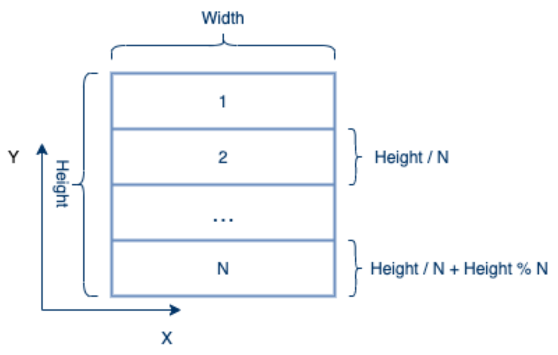

Figure 2: Decompose the image by rows.

Table 1: The runtime of decomposing by rows is shorter.

Method / Image size	By columns	By rows
1024 * 1024	0.039379 s	0.028604 s
4096 * 4096	0.229268 s	0.180593 s
8000 * 8000	1.320589 s	1.241563 s

4. *Halo exchange*

The original image is divided into N parts and N processes do the same things at the same time. For the 5-point stencil, the value in each cell of the grid is updated to the average value of the neighboring North, South, East and West cells. There are a large number of cells requiring data from adjacent cells from other processes.

In order to read the values of neighbors on different processes, halo regions are added to each processor boundaries and halo regions are populated with a copy of data from adjacent processes. According to the Figure 3, the first rank just needs to be added an extra row of cells under the local grid and the last rank needs to be added an extra row over the grid, but other ranks have to be appended to one halo region above and below the grid respectively. This is because the first and the last rank have been distributed one row of zeros.

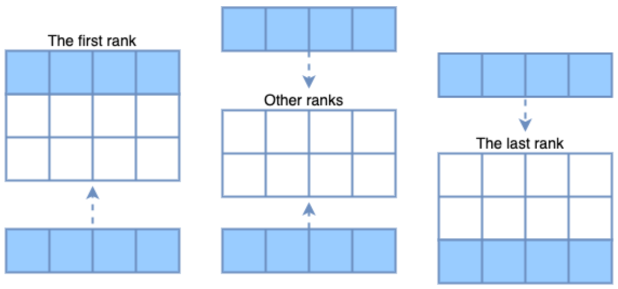

Figure 3: Append halo regions to each rank.

After adding the halo regions for all processors, the data from neighboring processes will be sent and received by calling MPI_Sendrecv function. As can be seen in Figure 4, the code packs data into an array and then send it to those adjoining processes, other processes are required to unpack the received buffer. Each rank needs to send messages to the upper rank as well as the lower rank and receive messages from them. Therefore, halo exchange should first send messages to the upper grid and receive from the lower grid, then send messages to the lower grid and receive from the upper grid.

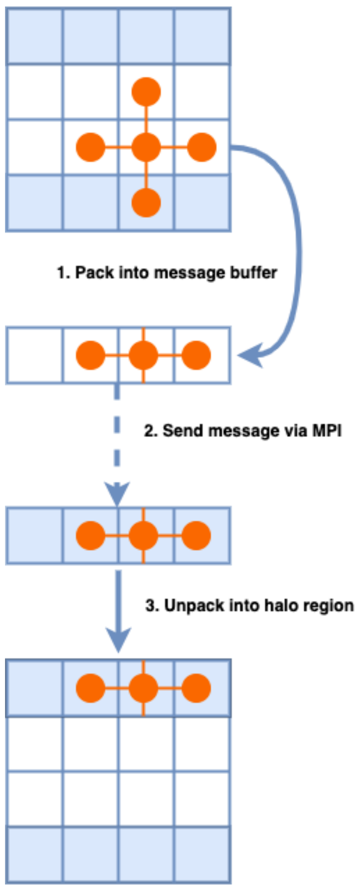

Figure 4: Halo exchange in MPI.

Each iteration should update all the cells which include core cells and halo cells. The core cells of each domain will be updated by calling *stencil* function while the halo cells need to be updated by doing halo exchange, which makes each iteration becomes two steps. The first step is to exchange halo regions and the second step is to call the *stencil* function for computing the new values of core cells.

5. Data collection and timing

When all processes finish calculations, the generated grids should be collected from each process. As can be seen in Figure 5, the first rank will first replace the corresponding values of the image array with the calculated values of the local grid. Next, the second rank receives the modified image array and replaces the values of its unique rows. Each rank will do the same thing until the last rank. The last rank will modify the image array and then send the image array to the first rank. The first rank receives this array and then calls *output_image* function for outputting the new image.

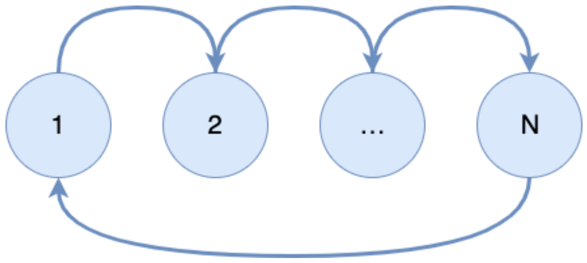

Figure 5: Collect data from the first rank to the last.

The timer only times the halo exchange and main compute loop of the code. To print the correct runtime, MPI_Reduce function is used to obtain the minimum start time and the maximum end time among all processes. The printed runtime is the difference between the minimum start time and the maximum end time.

6. Explanations and data analysis

Table 2 demonstrates that distributed memory parallelism significantly improves performance. Serial stencil code has to compute the whole image while the optimized method partitions the input image into 56 small images which can be computed at the same time via MPI which saves a lot of time. According to Figure 6, take 8000*8000 input image as an example, it is clear that there is a non-linear relationship between the number of cores and the runtime. With the increase in the number of cores, the influence on runtime becomes less obvious.

Table 2: Optimizations significantly improve performance.

	Before optimization	**After optimization**
1024 * 1024	0.110686 s	0.028604 s
4096 * 4096	2.878536 s	0.180593 s
8000 * 8000	10.348350 s	1.241563 s

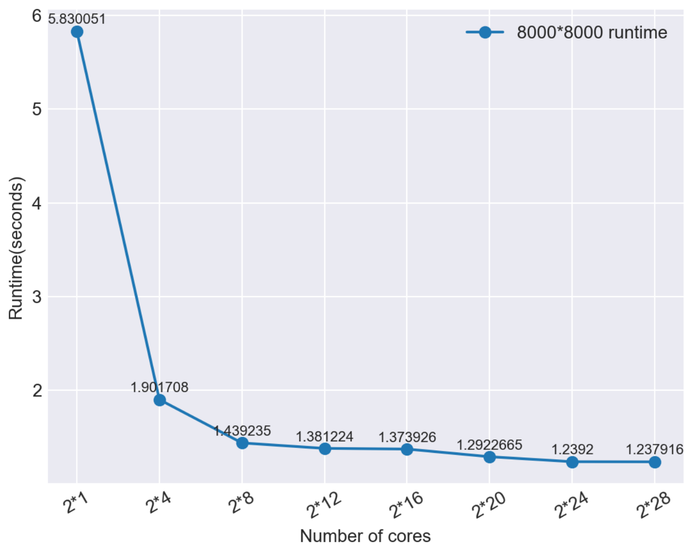

Figure 6: The relationship between the number of cores and runtime.

Reference

[1] Walker D W, Dongarra J J. MPI: a standard message passing interface[J]. Supercomputer, 1996, 12: 56-68.

[2] Makpaisit P, Ichikawa K, Uthayopas P, et al. MPI_Reduce algorithm for OpenFlow-enabled network[C]//2015 15th International Symposium on Communications and Information Technologies (ISCIT). IEEE, 2015: 261-264.