Abstract

Parallel computing is nowadays essential to increase the performance of programs. We learned different strategies to increase the performance with multithreading, which relies on shared memory. In contrast, this report focuses on parallelism with distributed memory. Programming with distributed memory is hard since the programmer has to manage the communication between all processes. This report shows how I parallelize the given Mandelbrot program by using MPI with C++.

Compile and execution guidance

1. Run the make script to compile the sequential version and parallel version with the command
    - make
2. Submit script is available in the Makefile, however this script will submit all slurm job scripts (from 2 processes to 192 processes). This approach can corrupt the image file because many jobs will be executed at the same time and written in the same file. This approach is only relevant to see the execution time result! Run the command:
    - make submit
3. You can submit a single job script, which can provide a correct image with the command:
    - sbatch job_<number of processes>.sh
    - e.g. sbatch job_4.sh
    - info: job.sh will submit a job with 2 processes
4. Open the resulting image and .out file after the execution is finished.
5. You can also run the executables without script with the command:
    - Sequential version:
        - ./a3-sequential
    - MPI version:
        - mpirun -n <number of processes> ./a3-mpi

Introduction

This report shows how I parallelize a mandelbrot program by using MPI. The goal is to achieve a parallel version of the mandelbrot program with a much better performance than the sequential program. After executing the program, an image file will be generated, which is also helpful for programmers to see its correctness. To compare the difference between the sequential one and different parallel versions, the width variable is set to 6144 and the height variable is set to 4096, which means the generated image will have 25165824 pixels. The program includes 2 calculation phases: an image generation phase and a convolution phase. Both phases have to be parallelized. In order to achieve a good speedup, different MPI communication modes should be tested.

While implementing, I realized that MPI is more difficult than I thought. There were some problems I encountered, which will be also mentioned in this report. According to the assignment, I upload the best effort MPI code version and tested this one on ALMA(alma.par.univie.ac.at)[0] by using different number nodes and processes to measure the execution time and speedups.

Firstly, let's take a look at the performance of the sequential mandelbrot program with the height variable = 4096, width variable = 6144, nstep=18:

Sequential Version	Kernel size = 5	Kernel size = 7
Image generation time	93.316	86.8411
Convolution time :	129.431	325.688
Total execution time	222.747	412.5291

Figure 1: Execution time of sequential code

The program needs 93 seconds to finish the image generation and 129 seconds for the convolution calculations. These values are essential to calculate the speedups for the parallel versions.

Lastly, all parallel implementations have to generate the same image as the sequential one, from pixel to pixel. The following image should be generated:

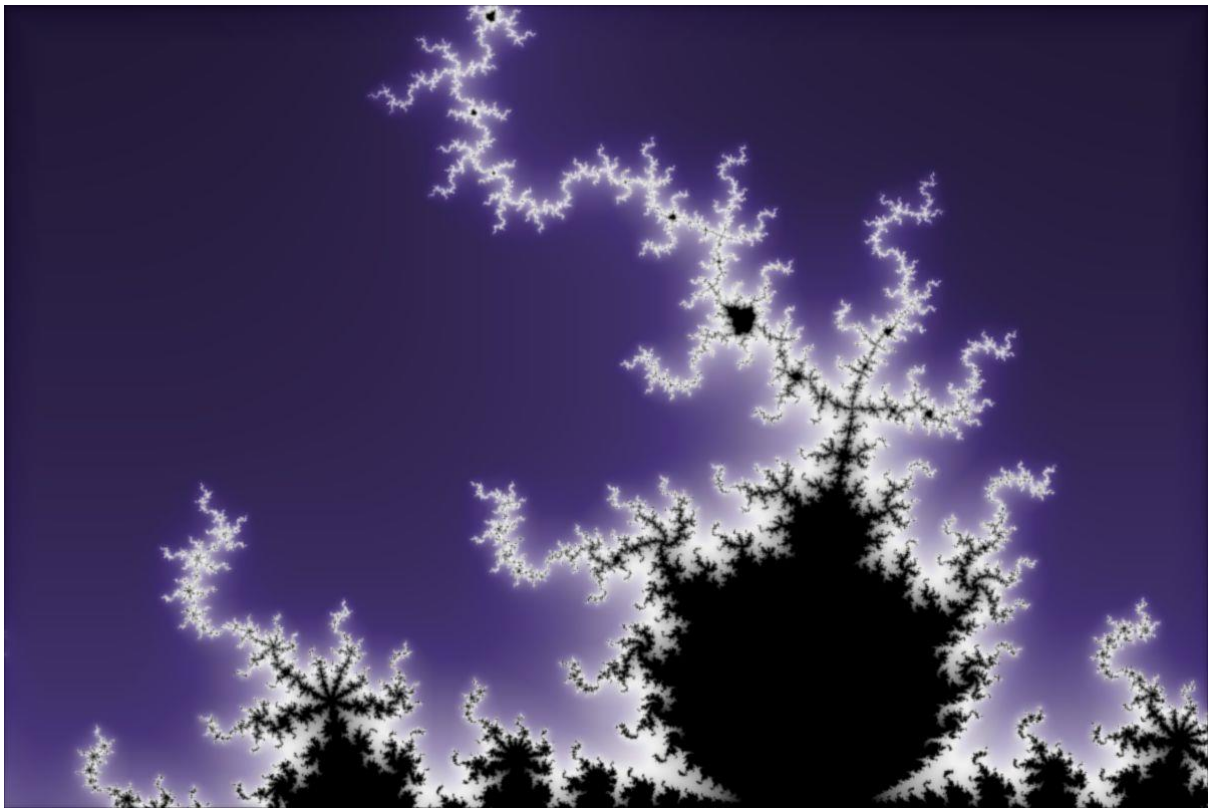

Figure 2: Image output

Image generation

The function “mandelbrot” has to be parallelized among all processes. Work distribution for all processes is essential for this task.

Work Distribution

I defined the variable “work_size”, which equals height divided by the numbers of processes. The variable “height_from” indicates from what height a process should begin and “height_to” indicates to what height a process should end. “height_from” equals to the rank of the process times the “work_size” and “height_to” equals to the rank+1 times the “work_size”. These variables are stored inside the image class.[Figure 3] Now each process has its own range to work for the mandelbrot. After each width iterations, each process will determine if the calculated value is an element of the mandelbrot set and then assign the pixel accordingly.

“pixels_inside” is a local and private variable for each process, and this variable indicates how many pixels are inside the image. Hence, this variable must be summed together at the end of the function. By using MPI_Reduce and MPI_SUM, we can easily get the sum of all private “pixels_inside” and return that value to the main.

However, this method can be optimized very well with load balancing, since many processes will be going idle, because some regions of the mandelbrot image require fewer calculations. I tried to load balance this function, however my implementation was either error prone or slower than the sequential one due to too many communications.

As we can see, the communication there is very crucial for the performance, less communication and blocking/synchronization operations between these processes is the key to reach a good performance.

```cpp
struct Image {
    std::vector<unsigned int> data;
    std::vector<unsigned int> buffer_top;
    std::vector<unsigned int> buffer_bottom;
    int height;
    int width;
    int channels;
    int half_kernel_size;
    int buffer_size;
    int height_from;
    int height_to;
    int work_size;
    int height_distance;
    Image();

    Image(int channels, int height, int width)
        : channels(channels), height(height), width(width), data(channels* height* width)
    {    }

    Image(int channels, int height, int width, int rank, int size)
        : channels(channels), height(height), width(width)
    {
        this->work_size = height / size;
        this->height_from = rank * work_size;
        this->height_to = (rank + 1) * work_size;
        this->height_distance = height_to - height_from;
        data.resize(channels * height_distance * width);
    }

    unsigned int& operator()(int c, int h, int w) {
        if (h < height_from)
        {
            return buffer_top[((-height_from + half_kernel_size + h) * width + w) * channels + c];
        }
        if (h >= height_to)
        {
            return buffer_bottom[((h - height_to) * width + w) * channels + c];
        }
        //ORIGINAL: (c*height + h)*width + w
        //CORRECT reordering for 1 process: [(h * width + w) * channels + c]
        return data[((h-height_from) * width + w) * channels + c];

    }

    void assign_kernel(int kernel_size)
    {
        this->half_kernel_size = kernel_size / 2;
        this->buffer_size = half_kernel_size * width * channels;
        if (height_from != 0)
        {
            buffer_top.resize(buffer_size);
        }
        if (height_to != height)
        {
            buffer_bottom.resize(buffer_size);
        }
    }
};
```

Figure 3: struct image with work distribution

Convolution:

The “convolution_2d “function is responsible to calculate the result image. Our job is to parallelize this function. I also distributed the works with previous defined “work_size”, “from” and “to”. However, this is not enough to parallelize this function since each process requires additional image information from its neighbour to calculate the panels around the edge correctly. Without these values, the resulting image will be wrong. Hence, the information exchange is crucial in this part.

Since I distributed the works respectively with the height, each process except the very first and last process should send the top part of its image region to the upper or previous neighbour and bottom part of its image region to the lower or next neighbour. Also, each process except the very first and last process should receive two segments of image from its neighbours. Since the first and last processes only have one neighbour, they only receive one image segment from their neighbour and send one part of their image region. If we are dealing with a 5x5 kernel, the sent part will contain 2 rows, which has the size of 2 times the width and the color channels.

The true hard part is how to communicate among the processes. By performing blocking send and receive with MPI_Send and MPI_Receive, the image parts can be transmitted and received safely. I also tested many other combinations such as MPI_Rsend, MPI_Ssend etc., but as mentioned from the lecture, MPI_Rsend is not reliable, and we should use MPI_Send as the blocking send method. The fastest combination I tried with 16 processes is the MPI_Send and MPI_Irecv combination. Even though MPI_Isend is a non blocking version, it is still slightly slower than the MPI_Send method in this case. Thus, I chose MPI_Send and MPI_Irecv combination for my end version. [Figure 4]

```
void update(Image& src)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Request _request[2];
    bool up = rank != 0;
    bool down = rank != size - 1;


    //receive
    if (up){
        MPI_Irecv(&src.buffer_top[0], src.buffer_size, MPI_UNSIGNED, rank - 1, 0, MPI_COMM_WORLD, &_request[0]);
    }
    if (down){
        MPI_Irecv(&src.buffer_bottom[0], src.buffer_size, MPI_UNSIGNED, rank + 1, 1, MPI_COMM_WORLD, &_request[1]);
    }

    //send
    if (down)
    {
        int from = src.height_to - src.half_kernel_size;
        int send_index = ((from - src.height_from) * src.width) * src.channels;
        MPI_Send(&src.data[send_index], src.buffer_size, MPI_UNSIGNED, rank + 1, 0, MPI_COMM_WORLD);
    }
    if (up)
    {
        MPI_Send(&src.data[0], src.buffer_size, MPI_UNSIGNED, rank -1, 1, MPI_COMM_WORLD);
    }

    //update image data
    if (down)
    {
        MPI_Wait(&_request[1], MPI_STATUS_IGNORE);
    }
    if (up)
    {
        MPI_Wait(&_request[0], MPI_STATUS_IGNORE);
    }

}
```

Figure 4: Communication function "update" inside the function "convolution_2d" with MPI_Send and MPI_Irecv

Further improvement for memory layout

Derived data type

I have tried the derived datatype such as MPI_Type_vector or MPI_Type_hvector to further increase the performance. I spent a lot of the time to try the data structure for MPI_Type_vector out, but unfortunately I could not achieve a correct result with the derived datatype.

Problem

The biggest problem I encountered in this assignment is that I could not store a 2d array in my image struct/class. I wanted to declare a 2d array inside the image struct, however I could not do that, because the height and width variables are not global accessible, i.e., they are declared in main. I tried to create a pointer int**. which points on different rows int*. However, by doing that, I actually have to concatenate multiple rows before the communication between the processes takes place, if the kernel has a size bigger than 3x3. Concatenating multiple rows actually requires many copy and loop operations, which is harmful for the performance.

Another improvement opportunity

I had another problem with the memory layout. The memory was contiguous with the function "(c*height+h)*width+w", which requires many operations through the for loop to extract or update the "halo" regions. It is much easier and less expensive if the memory layout is ordered by height because it allows us to send and receive the "halo" rows without any extraction processes.

Key Performance Increase for Convolution - Reordering the memory layout

During the implementation phase I realized that reordering the memory layout of the data variable in Image class is the key to accelerate the program. By changing `"data[(c*height+h)*width+w]"` to `"data[(h * width + w) * channels + c]"` and then adapt it for multiple processes: `data[((h-height_from) * width + w) * channels + c]` accelerates the convolutional part incredibly. The memory is then ordered by height like below:

h = 0	h = 1	….	….	h = height

This allows me to access each part of the "halo" regions very easily without any extraction and copy procedure. I can just send the top part of the respective image part with **MPI_Send(&data[0], <number of height required from other process>*width*channels, …..),** because the memory is ordered by height. [Figure 5]

```
unsigned int& operator()(int c, int h, int w) {
    if (h < height_from)
    {
        return buffer_top[((-height_from + half_kernel_size + h) * width + w) * channels + c];
    }
    if (h >= height_to)
    {
        return buffer_bottom[((h - height_to) * width + w) * channels + c];
    }
    //ORIGINAL: (c*height + h)*width + w
    //CORRECT reordering for 1 process: [(h * width + w) * channels + c]
    return data[((h-height_from) * width + w) * channels + c];

}
```

Figure 5: Reordering the memory layout of data

Save data to the disk

At the end, all image values calculated by the “mandelbrot()” and “convolution_2d” should be written to the disk correctly. A “mandelbrot.ppm” file must exist after the program finishes its execution and that file should deliver a correct result. Since each process has a part of the image, the values have to be written correctly and safely to the file.

Writing the image data

The root or first process creates the file and writes its own part of the image to the file. Other processes in contrast open the file and enter in the append mode, which allow the processes to write the data at the end of the file. After that each process checks whether it is his turn to write, and if it is, that process will append his part of image at the end of the file. Thus, all data from all processes can be written safely. In other words, this is a synchronous writing process, because if a process is writing, all others have to wait until it is their turn, and no other process is writing.[Figure 6]

```
int work_size = height / size;
int from = rank * work_size;
int to = (rank + 1) * work_size;
int currentRank = 0;
std::ofstream ofs;
while (currentRank != size)
{
    if (rank == currentRank)
    {
        if (rank == 0)
        {
            ofs.open("mandelbrot.ppm", std::ofstream::out);
            ofs << "P3" << std::endl;
            ofs << width << " " << height << std::endl;
            ofs << 255 << std::endl;
        }
        else {
            ofs.open("mandelbrot.ppm", std::ios::app);
        }
        for (int j = from; j < to; j++)
        {
            for (int i = 0; i < width; i++)
            {

                ofs << " " << filtered_image(0, j, i) << " " << filtered_image(1, j, i) << " " << filtered_image(2, j, i) << std::endl;
            }
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    currentRank++;
}
ofs.close();
```

Figure 6: Safely write the image to disc

Time measurements and speedups

According to http://www.par.univie.ac.at/teach/doc/alma.html [0] the cluster can support 32 processes on each node and there are six worker nodes available. Thus, the maximum runnable processes equal 32*6 = 192 processes. Hence, the best effort code version will be executed with 2, 4, 8, 16, 24, 32, 64, 96, 128 and 192 processes. However, there are only 16 cores per node, 32 processes are only available because of hyperthreading. In consequence, the execution time and speed up with processes above 96 might be inaccurate. The table below contains the execution time and speedup numbers respective to the number of processes.

Execution time of image generation and convolution, all measurements in seconds:

Execution time: height = 4096, width = 6144, Kernel size = 5, nsteps = 18			
Process	Image generation time	Convolution time	Total execution time
2	288.904	381.82	670.724
4	222.295	190.8	413.095
8	116.221	99.0022	215.2232
16	63.1977	53.4563	116.654
32	40.8215	27.0185	67.84
64	20.6493	14.04	34.6893
96	12.4269	9.24798	21.67488
128	15.7683	12.3454	28.1137
192	11.0529	8.09998	19.15288

Figure 7: Execution time respective to the number of processes

Speedup measurements of image generation and convolution, all speedups calculated with the table above and the sequential execution time with the formular S(n) = T(1)/T(n):

Execution time: height = 4096, width = 6144, Kernel size = 5, nsteps = 18,			
Process	Image generation speedup	Convolution speedup	Total speedup
2	0.323000028	0.338984338	0.332099343
4	0.419784521	0.678359539	0.539214951
8	0.802918578	1.307354786	1.034958127
16	1.476572723	2.421248758	1.909467314
32	2.285952256	4.79045839	3.283416863
64	4.519087814	9.218732194	6.421201927
96	7.509193765	13.99559688	10.2767351
128	5.917949303	10.48414794	7.923076649
192	8.442671154	15.97917526	11.62994808

Figure 8: Speedups respective to the number of processes

Plots of execution time developments of image generation and convolution. The x axis describes the number of processes, and the y axis is the execution time in seconds.

I created four plots to visualize the difference of execution time and speedup development between image generation and convolution.

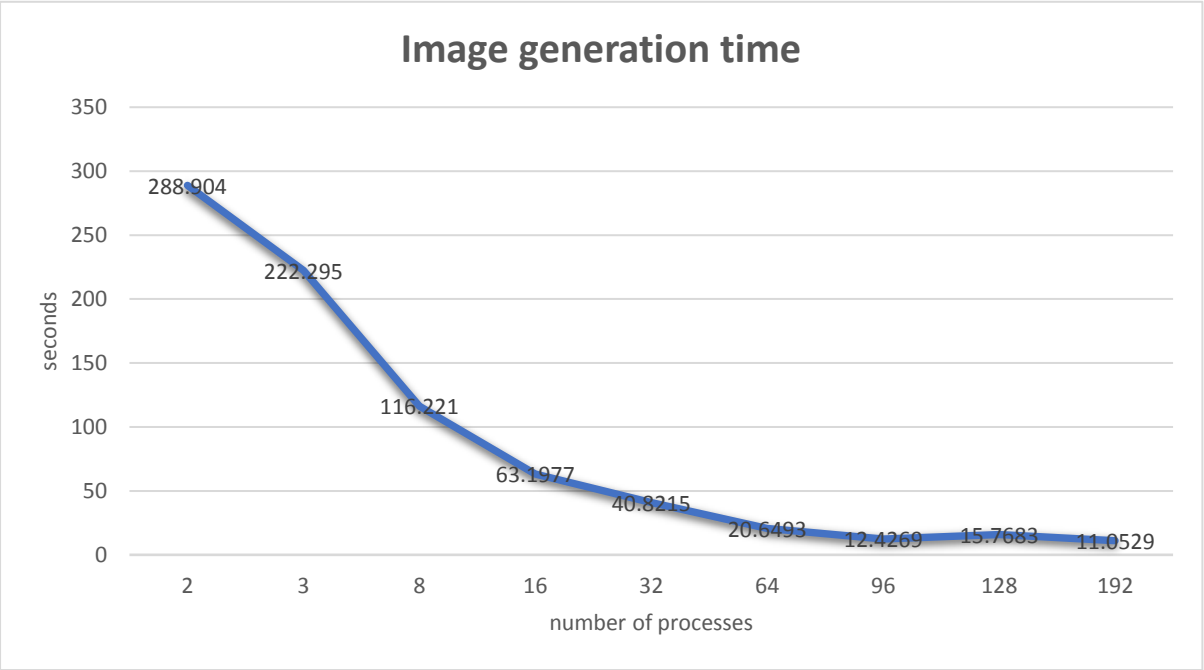

Figure 9: Image generation execution time development

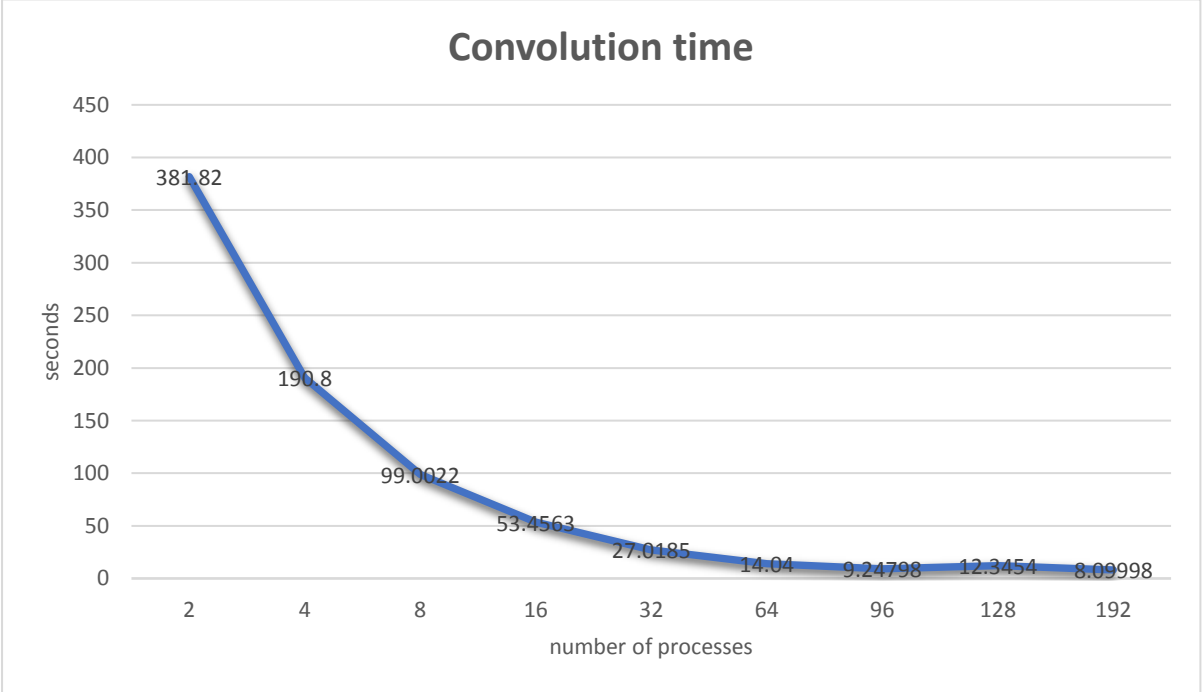

Figure 10: Convolution execution time development

Plots of speedup developments of image generation and convolution. The x axis describes the number of processes, and the y axis is the speedup number

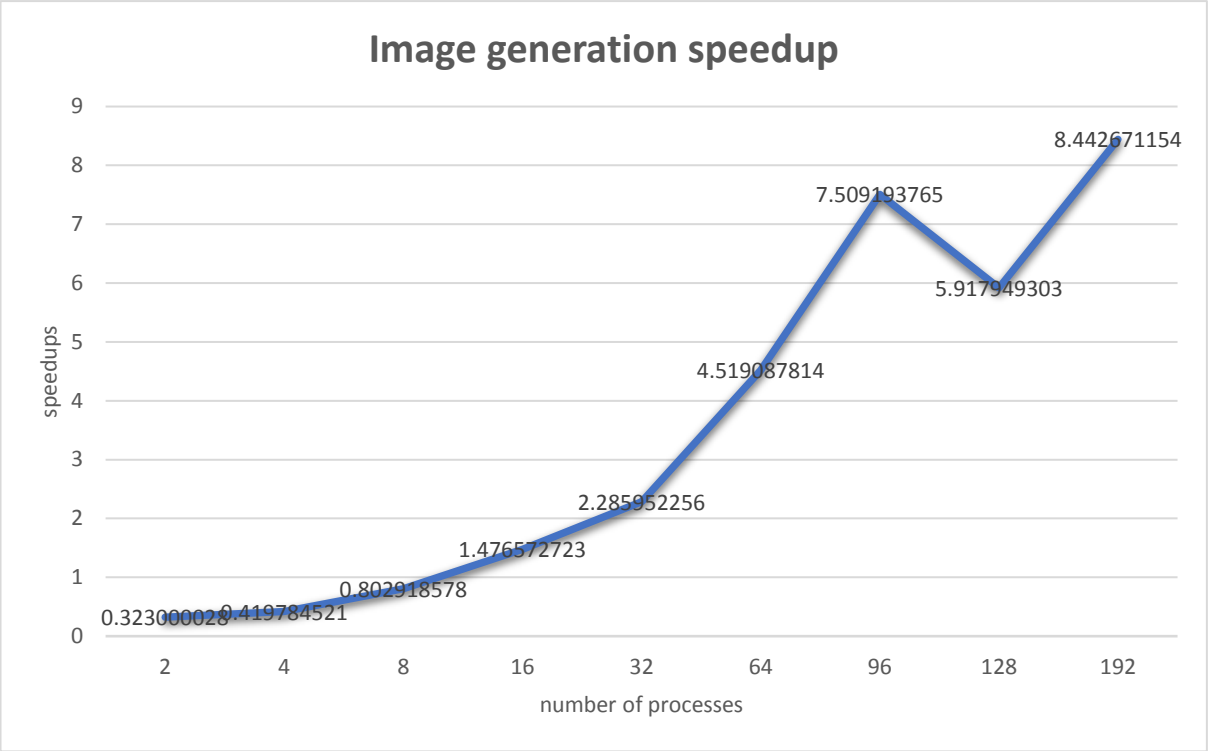

Figure 11: image generation speedup development

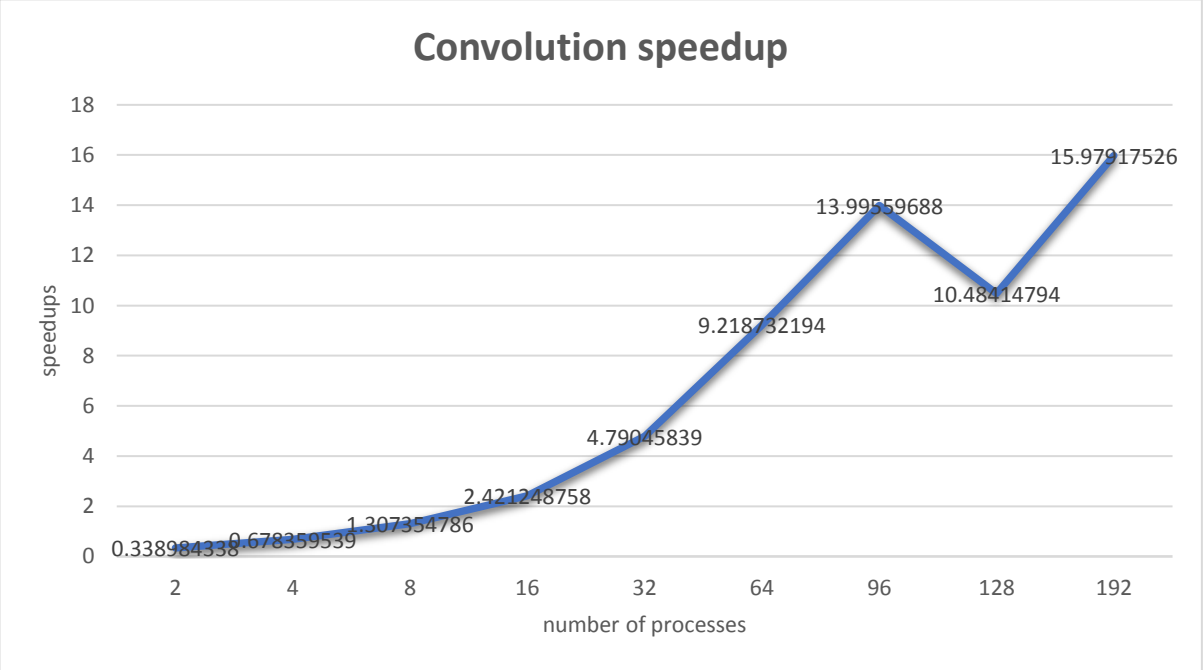

Figure 12: Convolution speedup development

I tested with the code with different kernel sizes, the table below illustrate the execution time and speedup with a kernel size = 7.

Execution time: height = 4096, width = 6144, Kernel size = 7, nsteps = 18,			
Process	Image generation time	Convolution time	Total execution time
2	288.09	722.643	1010.733
4	222.657	361.695	584.352
8	117.532	187.406	304.938
16	65.5132	100.703	166.2162
32	40.8376	51.0484	91.886
64	20.5961	26.5759	47.172
96	12.4296	17.4656	29.8952
128	16.428	22.8258	39.2538
192	10.9336	15.3619	26.2955

Figure 13: Execution time respective to the number of processes

Execution time: height = 4096, width = 6144, Kernel size = 7, nsteps = 18,			
Process	Image generation speedup	Convolution speedup	Total speedup
2	0.301437398	0.450690036	0.408148443
4	0.390021872	0.900449274	0.705959935
8	0.738871967	1.737873921	1.352829428
16	1.325551187	3.234143968	2.481882632
32	2.126498619	6.379984485	4.489575126
64	4.216385626	12.255013	8.745211142
96	6.986636738	18.64739831	13.79917512
128	5.286163867	14.26841556	10.50927808
192	7.942589815	21.20102331	15.6882014

Figure 14: Speedups respective to the number of processes

Documentation of performance

The image generation time can reach a max speedup around 7.5 to 8.5. As we can see, the image generation speedup increases slower than convolution. The reason is that many processes could finish their work very quickly and some of them have much more work to do. Consequently, root process has to wait until all other processes have finished their work for the "reducing". Hence, the function can be improved furthermore by applying load balancing correctly, which will be quite complicated, because it is very difficult to distribute the works dynamically and efficiently via message passing.

In contrast, the convolution time can reach a max speedup of almost 16 for a kernel size of 5 and more than 21 for a kernel size of 7 with 192 processes. The reason is that processes only have to exchange their "halo" part of the image. However, the exchange must efficiently take place. Every additional operation could lead to heavily performance decrease, because "convolution_2d" function requires many information exchanges between the neighbour processes. Thus, all processes are dependent from its neighbour, which makes them have to wait on each other. Thus, if the communication overhead is minimized, the program can be accelerated incredibly. The key performance increase is the memory reordering for the "data" variable inside the Image class. Without the reordering, the performance will be poorly, because without reordering the communication overhead of data extraction is very heavy.

Discussion and conclusion

The performance for distributed memory is very different to shared memory. In multithreading, threads have to deal with data dependency and solve it by using e.g. synchronization. Processes of message passing interface also require many communications to achieve parallelization for some part of the program. And sometimes a process has to wait data from another process, which requires a lot of time. In the "image generation" part many processes go idle quickly. The "convolution_2d" part shows us the importance of memory layout. Hence, reducing and avoiding the communication between processes and operations during communication phase is the key to achieve a good performance for distributed memory model.

The computation of parallelization for distributed memory is significantly harder than multithreading and OpenMP. The reason is that it is hard and complicated to exchange the messages. In addition, the MPI data transmission has many requirements, such as data type of the sending and receiving data.

References:

[0] ALMA, alma.par.univie.ac.at

[1] Alma cluster user documentation, http://www.par.univie.ac.at/teach/doc/alma.html