Parallel Image Convolution
2017-2018

Νέμανια Νέντις
A.M. 1115201400124
Δημήτριος Γούναρης
A.M. 1115201500032

Contents

Intro

Compile & Run

Compile each program from its designated folder using:
`mpicc -o main main.c filter.h filter.c` for MPI
or
`mpicc -fopenmp -o ./main main.c filter.h filter.c` for OpenMP.

Afterwards you can run any of them as follows:
`mpirun -np ./main [processes-num] [isRGB] [path] [w] [h] [loops]`

processes-num, width, height, iterations: integer values.
isRGB: bool value. 1 for RGB, 0 for greyscale.
path: relative path to the image that is convoluted.
i.e.: `mpirun -np 16 ./main 0 ../waterfall_grey_1920_2520.raw 1920 2520 20`

Behaviour

• In case the width or the height of the image is not perfectly divisible by the sqrt of the requested processes-num or if the said sqrt is not an integer the program will terminate. This can be expanded if you choose a different parallelism policy.
• The process with rank 0 is always the one that performs check such as the previous one. Its also the one that calculates how the columns and rows will be split and broadcasts that info to other processes.
• A checksum is performed after ever convolution loop to ensure that the filter worked/was applied. In case it failed the program will print a message to stderr for each fail.
• The resulting (convoluted) image will be named blurred.raw and placed in the current directory.
• Max process passed time in seconds is printed to stdout at the end of the program as well as other useful info about the behavior such as idle time per process and partitioning info. That time represents only the time from the start of the first convolution loop until the end of the last one. This is the time referenced in the "Performance" section.

Design

We followed the Foster methodology which divides the design in four distinct stages of partitioning, communication, agglomeration and mapping. These are further discussed below under Partitioning-Communication sections.

Partitioning

This part is about partitioning the image into smaller segments and efficiently sharing them between processes (one segment per process).

The image will be divided into blocks and every processor is to be assigned one block of the image.

Right away you may notice that the convolution of a single pixel includes all of its neighboring pixels. So covoluting any of the border-pixels of a block will require their neighbors, which happen to belong to different processes. Therefore each process will need to communicate these border pixels with its neighbors.

So it becomes apparent that the cost of communication between processes is proportional to the perimeter of the said segment, so we should strive for our segments to have the smallest possible perimeter. Check out the OptimzeCommPerimeter() function for more detail.

The blocks dimensions depend on the number of processes and the size of the image (static mapping, these parameters are defined at the command line).

The partitioning may fail if we are not able to divide exactly the rows and columns between the processes. There are ways to solve this issue (i.e.:allow blocks to have irregular shapes) but that opens other questions and we want to focus on our main goal here which is the cost of communication and scalablitiy measurements.

The partitioning of an image can be seen in the following examples.

Note that it is very easy for each process to determine its neighbors only by knowing its rank and the way we divided the image.

Observation: The closer the $width/neight$ ratio is to 1 (square image) the better the communication overhead is in proportion to the image size.

Figure 1: Partition example for 4 processes. The image is divided into 4 smaller blocks.

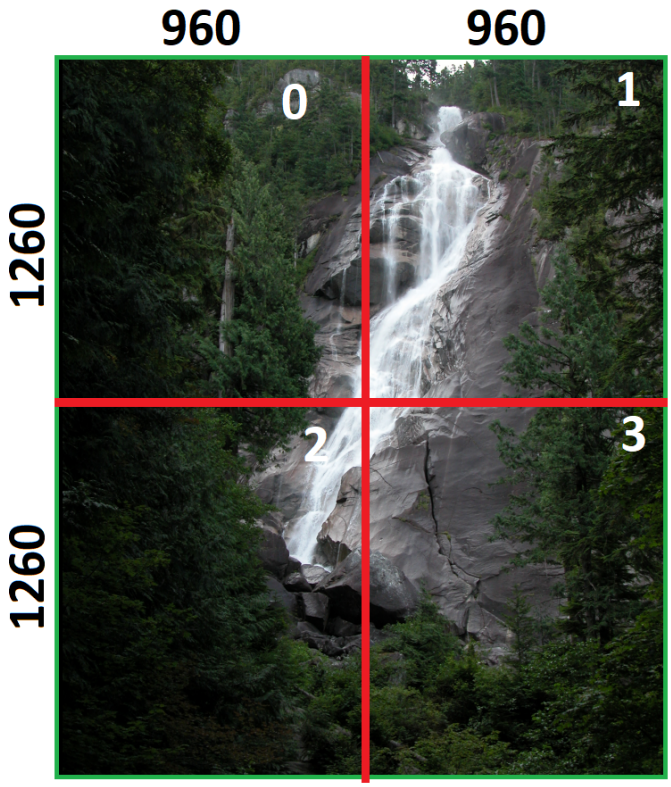
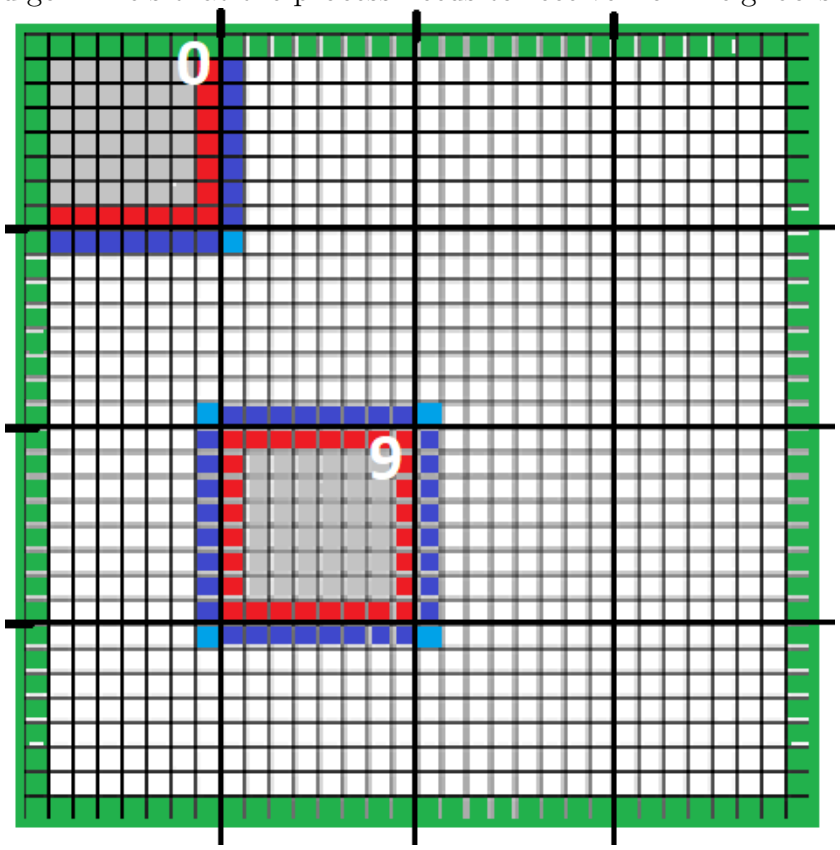


Figure 2: Demonstration for 16 processes. Border-Pixels of processes with rank 0 and 9 are colored. Red: pixels dependent on neighbors. Gray: Inner pixels(independent). Green: Edge pixels that don't have neighbors. Blue,Indigo: Pixels that the process needs to receive from neighbors.



note: Pixels that are at the border of the original image(green) do not have neighbors. These missing neighbors will be replaced with 0 since we init our arrays with calloc(), feel free to change this part to any constant of your liking.

Communication

For communication between processes used non-blocking communication (MPI_Isend(), MPI_Irecv()).

At the start of every loop each process will send the pixels that their neighbors need and request the pixels that it need from its neighbors. If we use non-blocking communication we can use the meantime to convolute the internal pixels that don't depend on other processes and that way save time, instead of blocking communication for every send/receive.

However, we still have to MPI_Wait() before covoluting the border-pixels AND before finishing a loop we have to make sure that every process has sent its border-pixels to the neighbors.

More on non-blocking communication:
For Non-Blocking Communication, the application creates a request for communication for send and / or receive and gets back a handle and then terminates. That's all that is needed to guarantee that the process is executed. I.e the MPI library is notified that the operation has to be executed.
For the sender side, this allows overlapping computation with communication.
For the receiver side, this allows overlapping a part of the communication overhead , i.e copying the message directly into the address space of the receiving side in the application.

Data types

For the simplicity and safety of communication we use one-dimensional arrays along with MPI's contiguous data types.

RGB_PIXEL: 3bytes, used to represent one red-green-blue pixel.
rowGrey/rowRGB: used to represent a full row of pixels.
colGrey/colRGB: used to represent a column of pixels.

Code Outline

Check Input parameters*
Find optimum block dimensions*
Broadcast dimensions to other processes*
Init arrays and read from the image
Find your neighbors
—Barrier—(wait for all processses to catch up before starting the clock)
Start clock
Loop:
    Send and Request borders from neighbors (non-blocking)
    Convolute inner pixels
    Get borders and convolute border-pixels
    Checksum
    Make sure you sent all your borders to neighbors
Stop clock
Write the result to blurred.raw
Print idle times and max process time to stdout
Cleanup and terminate

    *(only process with rank 0 does these)

Performance

Some sections are missing since we had technical issues on Sunday after 17:00 with the cluster (our programs would not terminate for any input. While 20mins before that they would termiante (i.e.: 0.79sec) for the same input). The mpi vs hybrid comparison was done on a local machine because of said technical difficulties.

MPI

Grayscale

num processes\image dimesions	480x630	960x1260	1920x2520	1920x5040	1920x10080
1	0.27	1.04	3.95		
4	0.07	0.21	1.79		
9	0.06	0.23	1.04		
16	$0.03_{(15)}$	0.07	0.50		
25	0.12	0.11	0.26		
36	0.09	0.09	0.20		

Red-Green-Blue

num processes\image dimesions	480x630	960x1260	1920x2520	1920x5040	1920x10080
1	0.75	6.43	11.83		
4	0.13	0.85	4.43		
9	0.08	0.22	1.73		
16	$0.04_{(15)}$	0.14	0.83		
25	0.15	0.17	0.51		
36	0.23	0.13	0.25		

Speedup

According to Amdahl's Law our code which comprises 90% parallelisable computations should at best achieve x10 speedup. Below are the graphs where green=480x630 red=960x1260 indigo=1920x2520.

Figure 3: MPI greyscale speedups.

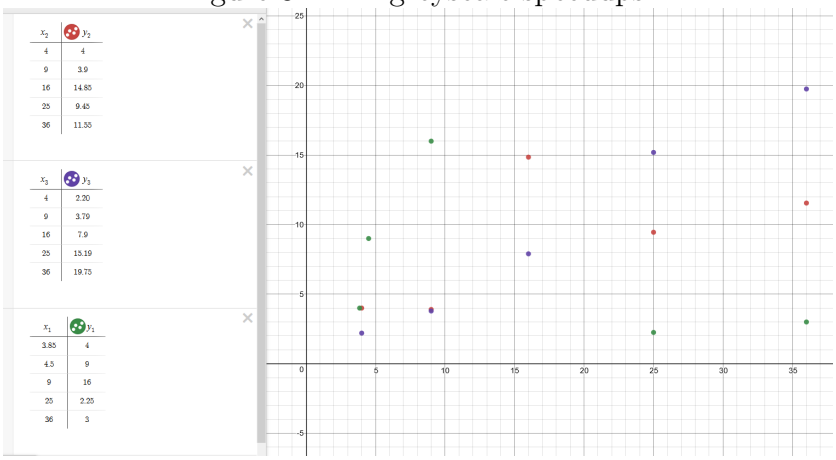
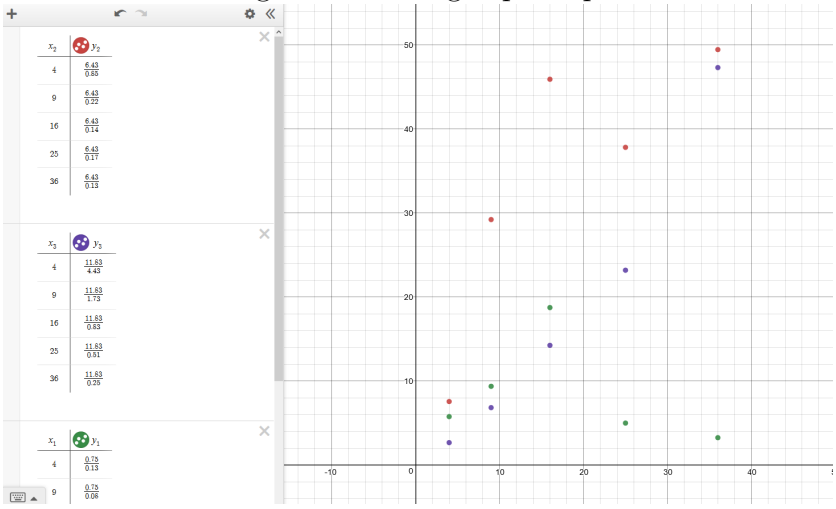


Figure 4: MPI rgb speedup.



Hybrid MPI+OpenMP

We used OpenMP parallelism in the loops where the filter is applied by collapsing them and sharing the load between threads equally. There was no use of creating threads in the main convolution loop since the communication overhead is not that high and the thread overhead would not be beneficial.

Grayscale

Hybrid MPI-OpenMP:

num processes\image dimesions	480x630	960x1260	1920x2520	1920x5040	1920x10080
1	0.22	0.69	2.74	4.98	11.47
4	3.06	3.04	4.25	5.21	7.53
9	10.18	10.43	11.28	12.07	8.39
16	$18.09_{(15)}$	19.09	19.26	15.68	17.31
25	30.85	29.73	30.35	32.35	$22.61_{(24)}$
36	44.78	44.03	44.26	44.21	45.60

vs MPI:

num processes\image dimesions	480x630	960x1260	1920x2520	1920x5040	1920x10080
1	0.28	1.04	4.10	8.34	18.10
4	0.10	0.32	1.01	1.90	4.42
9	0.30	0.33	0.99	1.93	4.67
16	$0.51_{(15)}$	0.53	1.22	2.20	4.09
25	0.56	0.59	1.41	2.35	$4.35_{(24)}$
36	0.98	0.98	1.20	2.60	4.52

Red-Green-Blue

Hybrid MPI-OpenMP:

num processes\image dimesions	480x630	960x1260	1920x2520	1920x5040	1920x10080
1	0.29	1.09	3.99		
4	3.45	3.62	6.46		
9	10.01	10.80	13.11		
16	$17.68_{(15)}$	20.11	21.26		
25	30.71	30.27	32.39		
36	43.78	43.53	46.17		

vs MPI:

num processes\image dimesions	480x630	960x1260	1920x2520	1920x5040	1920x10080
1	0.83	3.36	12.89	25.37	50.90
4	0.20	0.54	3.42	6.76	13.67
9	0.27	0.59	2.65	4.29	13.59
16	$0.54_{(15)}$	2.07	6.85	6.85	14.18
25	0.62	0.95	2.68	5.28	$13.56_{(24)}$
36	1.01	0.89	2.53	4.81	10.38

Comparison

It seems that the Hybrid program is more inefficient. The overall speedup and efficiency curves follow the same patterns but almost always the hybrid one is lagging behind the MPI one. We suppose that is mainly due to the interlocking mechanisms threads depend on in order to share memory. Besides for very small examples the overhead of creating a thread is not beneficial at all.

Scalability

Granted that our main task is tackling the communication overhead then scaling our processes will (in theory) be:

Best Case

Best case scenario is that our image is a perfect square and we go on scaling our processes x4 (to create 4 more perfect square out of each one.

Lets do some simple math:

comm-overhead $= p * 4a$,

where a is the side of our square and p is the number of processes.

Therefore if we quadruple the number of our processes:

comm-overhead $= (4p) * 4\frac{a}{2} = 8p * a$

Notice that for the same problem size our communication overhead doubled when our processes quadrupled. This is pretty good but don't forget that this is the best case scenario and that in reality this stops being true when our blocks get too small and the overhead of parallelism does not benefit us anymore.

Worst/Average Case

Following the same logic, we assume the case where we lets say double p but our block is skewed and has sides a, b (lets assume $a > b$, generality unaffected).

Now: comm-overhead $= p * (2a + 2b)$

If we double p the block will split efficiently so that the new dimensions will be $\frac{a}{2}, b$:

comm-overhead $= 2p * (2 * \frac{a}{2} + 2b) = p * (2a + 2b) + 2p * b$

The overhead has increased by $2p * b$!

Conclusions and Possible Improvements

Partitioning

We noticed that most of the time the process that handles the "middle" blocks (surrounded by other block on all four sides) are the ones to finish last, which leaves the other processes idle. This is of course because these blocks have to communicate on all four sides while the ones on the border don't need to do so.

We could try a different partitioning strategy where we could make these border bigger and the middle ones smaller in order to balance the overall communication overhead per process.

Hardware Dependency

When measuring performance on our local machine (4 cores) we noticed an interesting pattern. The times began to rise again after the number of processes surpassed the number of cores, while when testing in the cluster which had hundreds of cores the times kept dropping all the way till 36 processes. That can be blamed on the OS scheduling but in general, even for big clusters of machines you should try and balance out the ratio core/process to 1 to avoid the scheduling overhead.

In General

For large problems there is no doubt room for improvement by using parallelism, especially since Moore's Law became obsolete. But we should be very careful when designing our code and follow established and tested practices that will set us on the right course.

But its equally important to take measures that depend on your situation (i.e.:hardware architecture). That we found is best acomplished by testing and comparing times, extracting the metrics and comparing them to the theory. No doubt will reality be far from theory by asking yourself "why so?" will help you really improve your solution and find things you missed out on, as well as help you learn more.