

# Parallelizing Convolutions using MPI

**ME766 - High Performance Scientific Computing**

## **A Course Project**

*Submitted by,*

Vivek Revi (193100076)  
Saurabh Mandoakar (193100081)  
Sudip Walter Thomas (193100063)  
Komal Agnihothri (193230001)



**Indian Institute of Technology, Bombay**

**May 2021**

# 1.0 Introduction

In image processing, convolution algorithms play an important role. It is the process of adding each element of the image to its local neighbourhood weighted by a kernel. Depending upon the kernel, the image can be transformed to Blurred, Sharpened or to extract important features such as edges and curves. This becomes very useful in machine learning algorithms used to detect objects from image using high level features of the image extracted using convolution algorithms. Due to the nature of this algorithm which involves a large number of repeated matrix operations, makes it ideal for parallelization. In this project we try to make use of Message Passing Interface (MPI) for parallelizing convolutional algorithms and then analyse the effect of varying image sizes and number of processes on the speed of execution.

## 1.1 Gaussian Blur using Convolution

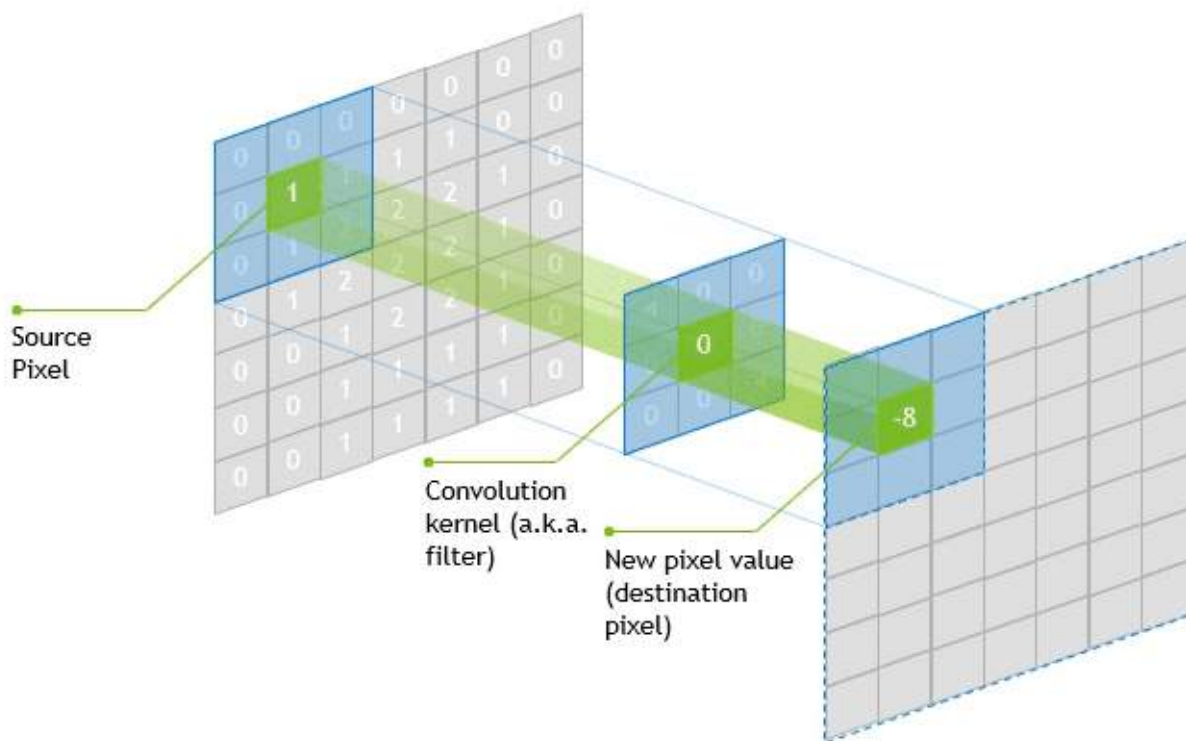


Figure 1.1.1: Schematic diagram of convolution operation.

In this project we are implementing a 3x3 convolution kernel for gaussian blur on an input image. Gaussian blur reduces noise and details of the input image producing an output image similar to that seen through a translucent screen. Mathematically,

applying a Gaussian blur to an image is the same as convolving the image with a Gaussian function. The kernel used for gaussian blur is shown below.

Kernel used for Gaussian Blur:

$$\frac{1}{16}$$

1	2	1
2	4	2
1	2	1

The above 3x3 kernel is convolved across the entire pixel height and width of the image to transform it into a blurred output.

## 1.2 Message Passing Interface (MPI)

MPI is a library which helps us to do problems in parallel on a distributed memory machine using message passing to communicate between processes. The communication may be via a dedicated MPP torus network or even an office LAN, for MPI programming it does not matter. The basic functionalities MPI provides are,

1. Start Process
2. Send Messages
3. Recieve Messages
4. Synchronize

Even though there are over 125 functionalities MPI offers for specific tasks, the above routines are used in a vast majority of cases. In this project we are coding the convolution algorithm in MPI to parallelize the program using PARAM Sanganak HPC cluster.

## 1.3 PARAM Sanganak Specifications

PARAM Sanganak is a supercomputing facility at IIT Kanpur. It is based on a heterogeneous and hybrid configuration of Intel Xeon Cascade lake processors, and NVIDIA Tesla V100. The system was designed and implemented by HPC Technologies team, Centre for Development of Advanced Computing (C-DAC). It consists of 2 Master nodes, 8 Login nodes, 10 Service/Management nodes and 312 (CPU+GPU) nodes with total peak computing capacity of 1.66 (CPU+GPU) PFLOPS performance. PARAM Sanganak systems are based on Intel Xeon Platinum 8268, NVIDIA Tesla V100 with total peak performance of 1.6 PFLOPS. The cluster consists of compute nodes

connected with Mellanox (HDR) InfiniBand interconnect network. The system uses the Lustre parallel file system.

Total number of nodes: 332 (20 + 312)

- Service nodes: 20\*\* (Master+ Login+ Service+ Management Nodes)
- CPU only nodes: 150
- GPU ready nodes: 64
- GPU nodes: 20
- High Memory nodes: 78

## 2.0 Working of Code

The operations involved in the execution of this algorithm is as follows,

- Image is zero padded in order to maintain the resolution of image post convolution.
- This image is partitioned into equal sections based on the image size and number of processes. (2D Domain Decomposition)
- At the beginning of an iteration boundary data is being shared with each of the neighbouring processes.
- Interior data points depend only on data residing same process (local data, not shared)
- Boundary data is communicated across processes, and they are updated.
- The above operations are repeated in each of the iterations.
- Once all convolutions are done, we combine the individual partitions to obtain the output image.

## 3.0 Results and Discussions

The above code was executed on the PARAM Sanganak supercomputer. The obtained output image was saved and the time taken for execution was tabulated. Shown below are the results after code execution.

### 3.1 Input and Output of Program

Both RGB and Grayscale images of various sizes were given as input to the program and corresponding results were noted. The image sizes used as input to the convolution algorithm were as follows,

- 1920 x 630
- 1920 x 1260

- 1920 x 2520
- 1920 x 5040

An example of Input and output of RGB 1290 x 1260 image from is shown below,

**Input: RGB 1920 x 1260**



**Output: Gaussian Blurred RGB 1920 x 1260 (50 iterations)**



### 3.2 Performance Charts

The time taken for execution of each image size including RGB and Grayscale variants are noted. The performance was also measured for a varying number of processes. Here we are keeping the number of convolution iterations over the input image as a constant at 50 iterations. We are evaluating the time taken for code execution for a number of processes varying from 1 to 20. For this we make use of 2 compute nodes each with a maximum task per node as 10. The results obtained are shown below.

Img/Processes	1log1	2log1	2log2	2log4	2log8	2log10
grey_1920x630	2.711	1.361	0.756	0.398	0.232	0.198
grey_1920x1260	5.409	2.707	1.384	0.758	0.395	0.335
grey_1920x2520	10.873	5.444	2.726	1.422	0.746	0.618
grey_1920x5040	21.764	10.879	6.284	2.772	1.621	1.229
rgb_1920x630	7.104	3.561	1.846	0.954	0.502	0.421
rgb_1920x1260	14.209	7.116	3.611	1.846	0.958	0.778
rgb_1920x2520	28.454	14.235	7.172	3.652	1.968	1.524
rgb_1920x5040	56.893	28.431	14.276	7.201	3.746	2.904

Table 3.2.1: Timing chart for code execution. All times are given in seconds.

The above table shows times taken in seconds for code execution in each case of image type, image size and number of processes. The greyscale images having only one channel takes less time than RGB images having three channels. Here, the number of processes represented by “xlogy” indicates:

- x = Number of nodes.
- y = Number of tasks per node.

The total number of parallel processes is given by  $x*y$ .

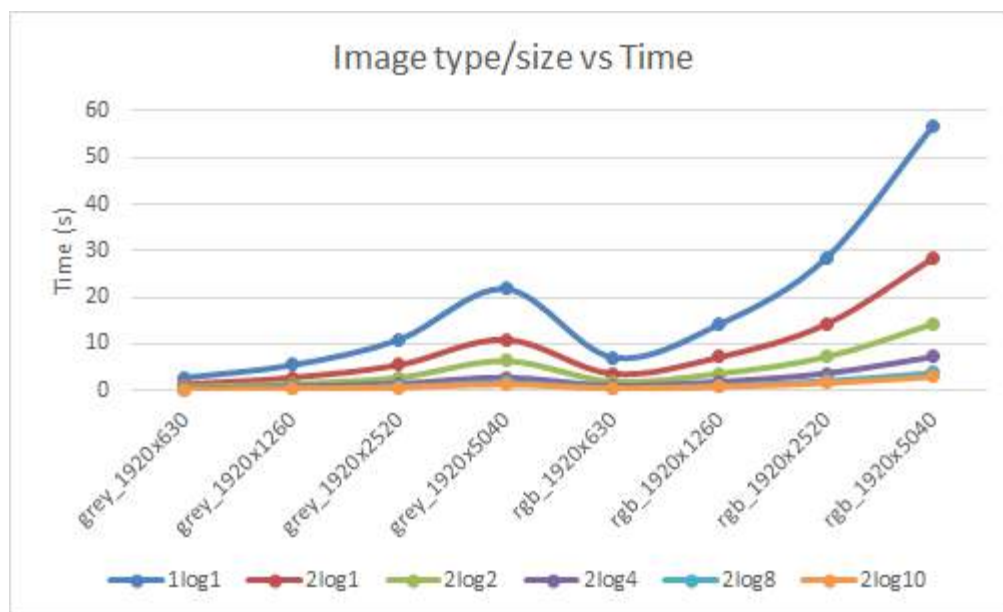


Figure 3.2.1: Line plot of time taken with respect to image type and size. The line color represents various number of processes.

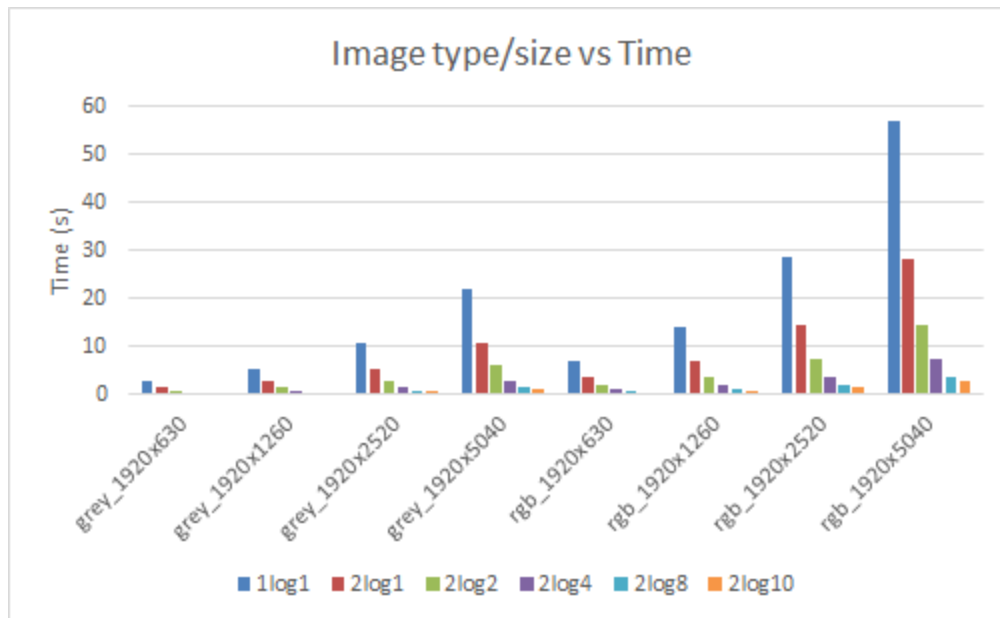


Figure 3.2.2: Histogram plot of time taken with respect to image type and size. The bar color represents various number of processes.

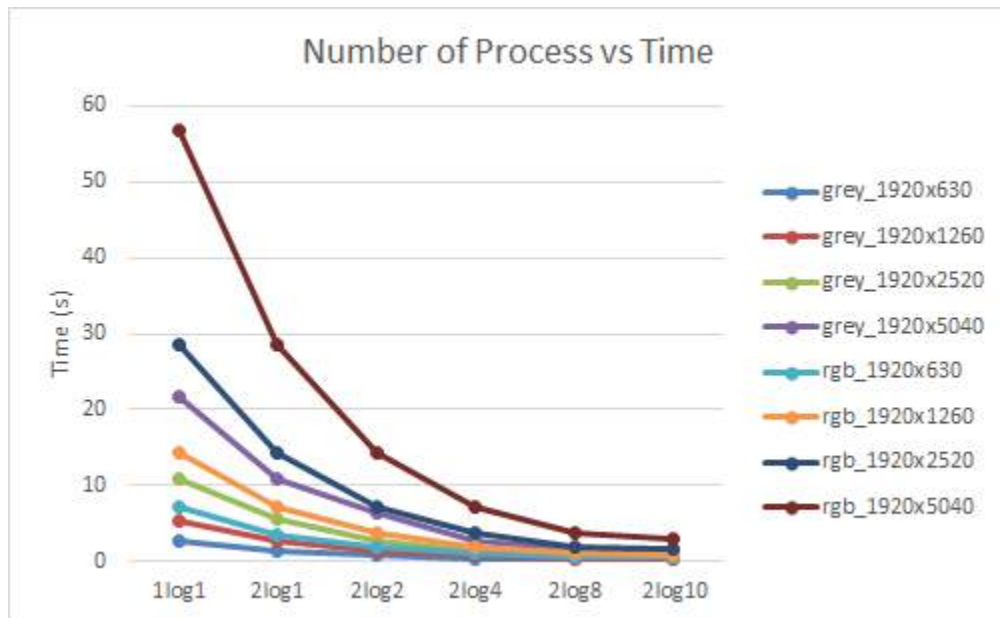


Figure 3.2.3: Line plot of time taken with respect to number of processes. The line color represents type and size of images.



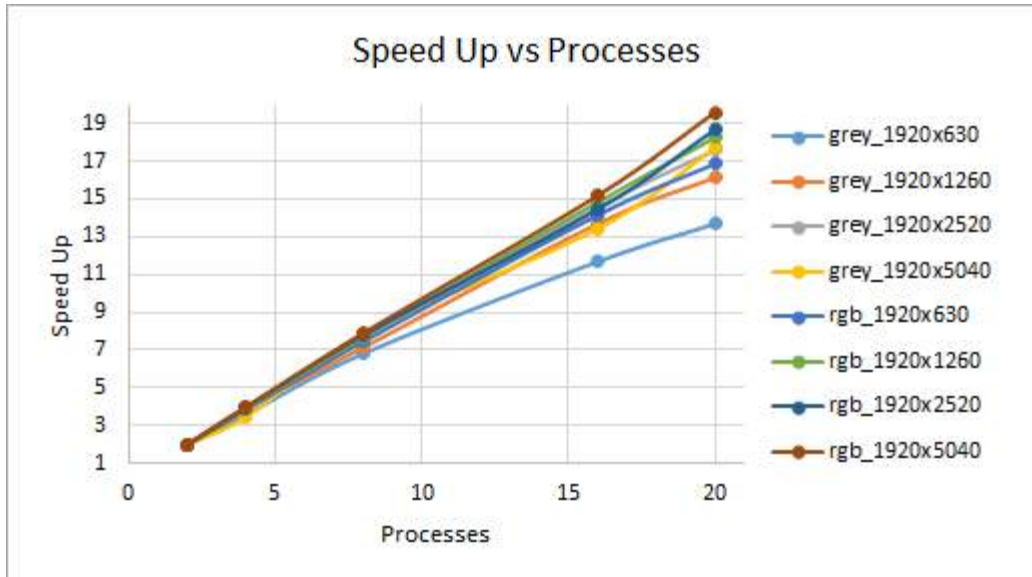


Figure 3.2.4:  $\text{Speed Up} = T_{\text{serial}}/T_{\text{parallel}}$

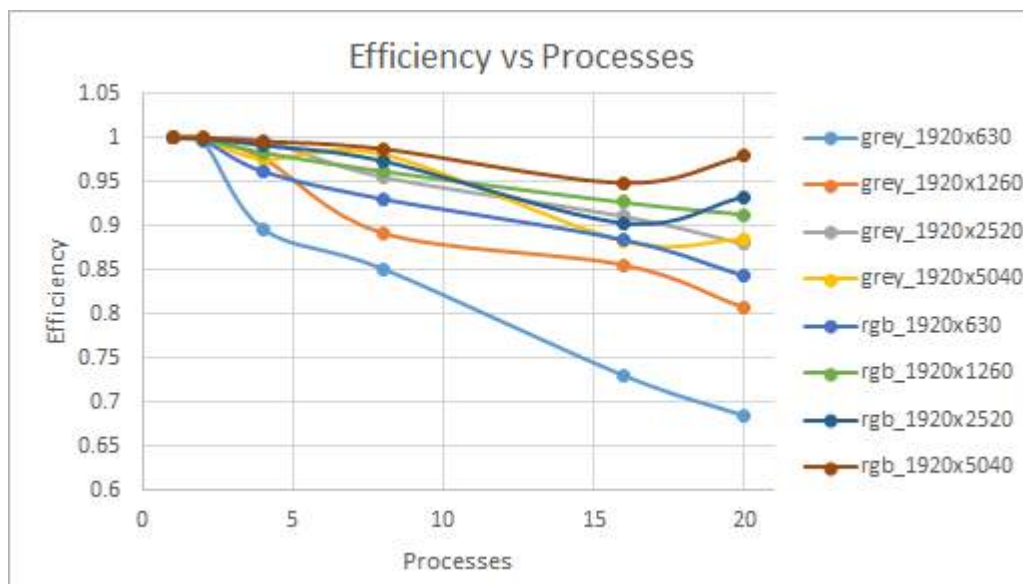


Figure 3.2.5:  $\text{Efficiency} = \text{Speed Up}/\text{No. of processes}$

### 3.3 Effect of Number of Iterations

A single convolution through the entire image using the gaussian filter does not result in a significantly noticeable blur. This is why we decided to do multiple iterations, one on top of another in order to get a significant amount of blur on the output image. The number of iterations performed has also got an effect on the time taken for



execution. The results below show this effect for an RGB image of size 1920x1260 with number of processes 20 for all the cases.

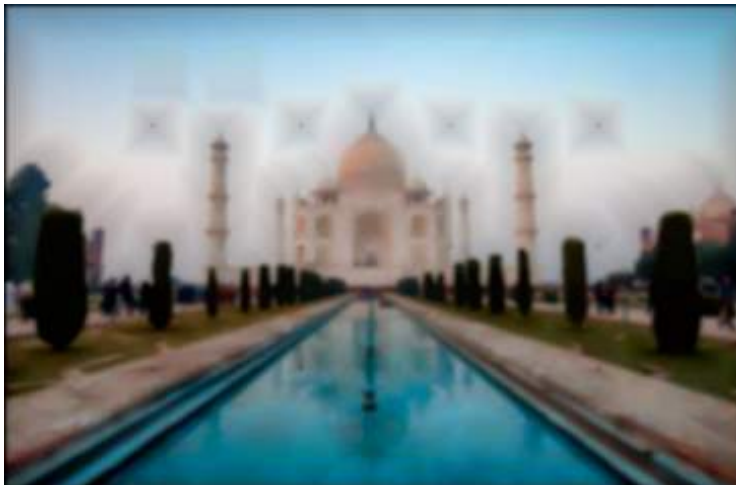
**Iteration = 25, Execution Time = 0.410s**



**Iteration = 50, Execution Time = 0.753s**



**Iteration = 100, Execution Time = 1.494s**


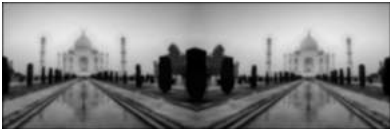






## 4.0 Observations & Conclusions

From the table and figure given in above Section 3.2, we can observe the following:

- The time taken for code execution is less for Grayscale compared to an equivalent sized RGB image. A time difference of nearly 3 times is observed between Grayscale and RGB. This is due to the presence of 3 color channels in RGB, as opposed to 1 in Grayscale. The convolution operation takes place in each of the channels separately hence the time difference.
- From Figure 3.2.2, we can observe the time taken for execution halving as we double the number of parallel processes. These results are in compliance with the known relationship between the number of processes and time for execution.
- The image sizes ( $w \times h$ ) are chosen in such a way that the width ( $w=1920$ ) remains constant but the height doubles from case to case ( $h$ ,  $2h$ ,  $3h$  and  $4h$ ). This corresponds to a doubling of time.
- In figure 3.2.4 we see the speed up for each process and observe that the speeding up follows a more linear trend. For `rgb_1920x5040` we see the maximum speed up as for serial processing this image is too hard to handle but that is where MPI parallelization is working its best too.
- Figure 3.2.5 is an efficiency chart, where `rgb_1920x5040` maintains an almost constant efficiency while the least complicated image loses its speedup/process as the number of processes increases. This shows how for simpler codes parallelization can perform poorly but for more complicated one perform excellently. Also this happens as more number of processes means more sharing of information
- Section 3.3 shows us the effect of the number of iterations against the time taken for code execution. We can see the time taken doubling as the number of iterations is doubled.

# 5.0 All Inputs and Outputs

Image	Input	Output
grey_1920x630		
grey_1920x1260		
grey_1920x2520		

grey\_1920x5040



rgb\_1920x630



rgb\_1920x1260



rgb\_1920x2520



rgb\_1920x5040

