Group Final Project

2D Image Convolution

CP631 Parallel Programming

Professor: Pawel Pomorski

Aug 7th, 2022

GROUP 2

Barber Joshua, SID: 215854630

Choudhry, Khalil, SID: 225812340

Geng, Huayi(Justin), SID: 215836230

Introduction

Images on our computers are represented as a 2-dimensional distribution of small point values called pixels. It can be viewed as a function of two real variables, as f(x, y), where f is the Image at position amplitude (for example, brightness) (x,y). Digital image processing aims to enhance image quality before doing feature extraction and categorization. It is helpful in various related fields, including computer vision, medical imaging, meteorology, astronomy, and remote sensing. The fundamental issue is that it is typically time-consuming; parallel computing offers an effective and practical solution to this problem. Parallelizing improves the speed with which the image is created. This project examines the distinction between serial and parallel programming when it comes to image processing, and in particular, 2D convolution.

We felt that this was an ideal candidate for parallelization, as images are often represented as 2D matrices, and thus lend themselves to being divided into submatrices for distribution across multiple processes. Convolution has many uses. Different kernels are applied to an image to yield different results, such as for edge detection or image sharpening. Many users may be running edge detection on thousands of images. One could also imagine doing real-time convolution on video frames could benefit from parallelization.

Code Implementation

We chose to implement our solution with mpi4py which was mentioned in the course notes. We had previously emailed the professor to confirm this was allowed. The reason for this decision was so that we could easily leverage python image processing libraries, and to also ease development as python has some nice features such as managing memory automatically.

While setting up dependencies like mpi4py was easy on our local machines, it was more difficult when testing on the school/lab remote vpn machines. This is because we lacked root permissions and could not install rpm packages, causing "`pip install mpi4py`" to fail on our remote computers. A workaround was implemented that allowed us to install all dependencies and run the code on the remote machines. Please refer to attached readme with the submission. The readme also explains how to use and run the program.

Other dependencies such as "numpy", "opencv-python", "PySimpleGUI" could be installed via a simple pip install without error. Readme also explains how to run code and pass commandline args. We also built an (optional) simple python GUI. GUI has issues on remote so easier to run via commandline only.

Additionally, we made assumptions for simplicity. Namely that the number of processes will be evenly divisible against the image size (pixels), that our images were all squares (nxn), and for our cartesian topology, our num processes was a square (yields int square root).

Process Image with Serial Computing

The processor completes one task (an image operation) at a time during serial processing. The other tasks are then carried out sequentially after that. Each program that an operating system runs has several jobs to complete. All of these duties must be finished by the CPU, but it does so one at a time. While the processor does the current work, the other tasks are held in the queue. In other words, each image process operation is completed in order.

Below we have a pseudo-code representation of the 2D convolution. But first recall general form of convolution is given below, where g(x,y) is the output image, f(x,y) the input, ω is the kernel matrix. $-a \le dx \le a$, and $-b \le dy \le b$, is simply representation for all the indices in the kernel.

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x-dx, y-dy),$$

https://en.wikipedia.org/wiki/Kernel_(image_processing)

Our Pseudocode operation is given below:

```
convolve2DProcess(image, kernel):
    # kernel
    kernel <— apply cross correlation to our kernel
```

```
 # Gather Image and Kernel Shapes
  inputImageShapeY <— image.y
  inputImageShapeX <— image.x
  kernelShapeY <— kernel.y
  kernelShapeX <— kernel.x


  # create a fresh matrix with the deduced dimensions
  output = [newShape.x, newShape.y]

  # Iterate through whole image
  for y in image.y:
      # exit convolution if at the end of y direction
      if image.y - kernelShapeY is less than y :
          break
          for x in image.x:
              # exit convolution if at the end of x direction
              if image.shape[0] - kernelShapeX is less than x:
                  break
              try:
                      output[x, y] <— Image * kernel
#In actual code we implement calculation, but here we leave it simply as
          # convolution operator pseudocode
              except:
                  break


  return output
```

Process Image with Parallel Computing

The main idea behind parallel image processing is to divide the whole image into simple tasks and solve them concurrently, so that the total time can be divided between the total tasks (in the best case). As such we would divide the image matrix into smaller submatrices and send these “slices” to the various processes over MPI communication. Each individual process can then convolve2D on their own respective slice, and then communicate their output. The slices can be combined to yield a new image matrix for the output. The slices are run concurrently, whereas in serial they are run sequentially.

```
Main
   # Init MPI
  comm <— MPI.COMM_WORLD
  comm_rank <— comm.Get_rank()
  comm_size <— comm.Get_size()
  start <— set start time

   # Init kernel
  kernel <— np.array([3][3])

  # Data after processing
  finalData <— None

  if comm_rank is master:
     # Grayscale Image
```

```
        image  <—process image to grayscale
        finalData <— np.zeros((image.x, image.y))
        # Breakup image and send to processes
        slices = np.vsplit(image, comm_size)


        # Go through each process
        for i in range(1, comm_size):
            send slices[I] to each process
    else:
        received <— received from process 0
        outputSegment <— convolve2D(received, kernel, padding)
        send outputSegment to master process


    if comm_rank is master:
        outList <— collect all segments from each process
        output <— np.vstack(outList)
        # Write a new image
        cv2.imwrite('2DConvolved.jpg', output)
```

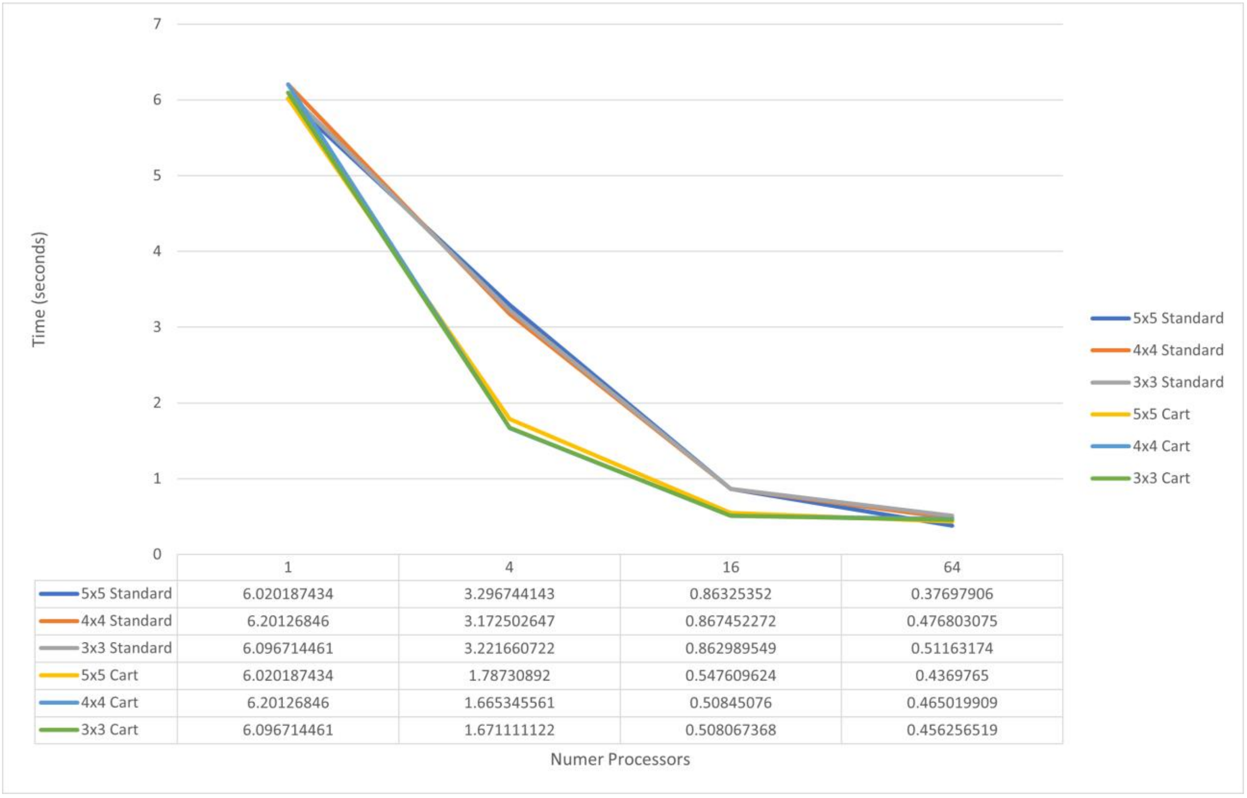
Where convolve2D is the same as it is in serial computing.

We later realized that this could be optimized with cartesian topologies. We added another implementation in our code. With some modifications we made two changes. First instead of purely horizontal splices that we obtained (via vertically splitting the matrix), we had a further round of splitting (hsplit) on each of these slices. The result is we not only had rows, but columns and a grid of submatrices. This structure was easy to use by creating a cartesian topology and mapping the co-ordinates to this structure. The dimensions of the topology were sqrt(comm_size) * sqrt(comm_size). Thus we can access the appropriate submatrix on each process using code such as this example below:

```
cartesian2d = comm.Create_cart(dims = [sq_dim, sq_dim] ,periods = None ,reorder=False)
coord2d = cartesian2d.Get_coords(comm_rank)
slice = slicesXY[coord2d[0]][coord2d[1]]
outputSegment = convolve2D(slice, kernelArray)
```

SlicesXY is created by taking our Slices from the previous version, and hspliting each slice. Additionally, we no longer splice by comm_size, but the sqrt of comm_size. Additionally, each process has an identical SlicesXY. After creating SlicesXY on process 0, we broadcast it to all processes.

Comparison between Serial and Parallel Computing



	1	4	16	64
5x5 Standard	6.020187434	3.296744143	0.86325352	0.37697906
4x4 Standard	6.20126846	3.172502647	0.867452272	0.476803075
3x3 Standard	6.096714461	3.221660722	0.862989549	0.51163174
5x5 Cart	6.020187434	1.78730892	0.547609624	0.4369765
4x4 Cart	6.20126846	1.665345561	0.50845076	0.465019909
3x3 Cart	6.096714461	1.671111122	0.508067368	0.456256519

As we can see from the above graph, we have the cartesian and standard implementations, with the cartesian implementation outperforming the standard. Note that Nprocs = 1 refers to the serial code (simply calling convolve2D on whole image) and is the same data for both, and not dependent on our parallelization algorithms. The times were obtained as an average of 3 separate runs. Also refer to the stats text file for raw data, submitted with project.

We see the improved performance of the cartesian version. This is likely because of the optimized inter-process communication. Instead of each process needing to receive their own unique slice via a send from process 0, SlicesXY is sent to all via broadcast, and the relevant item is worked on via cartesian topology co-ordinates. There seems to be diminishing returns as nprocs rises.

Gains:

Recall from course lessons:

$$s = \frac{time\ for\ (best) serial\ algorithm}{time\ for\ parallel\ algorithm} = \frac{T_s}{T_p}$$

Solving with our values, for the case of p=4, we find for 5x5 case, s = 1.826 for the standard case, and s = 3.368 for the cartesian case. But as previously seen on the graph this diminishes as p increases. For p=64, for 5x5 case, standard is s= 15.97 and cartesian is 13.77.

As we desire s to be as close to p as possible, in some context we will find lower values for number of processes may be better in terms of efficiency, even if they take more time. There seems to be room for improvement in terms of parallel performance, especially for higher p.

Conclusion

There are clear benefits of applying the principles of parallel programming towards applications such as image convolution. In this case, we could perform many image processing tasks from different algorithms by replacing out call to convolve2D and re-using the same parallelization of the image matrix. As such this approach is generalizable to many image processing problems at large, not just convolution. However there does seem to be room for more optimization to make the most out of parallelization and obtain higher efficiency.