MPI Project Report : Edge Detection

Abdelmajid KOUBACH

January 2023

1 Introduction

Edge detection is a fundamental step in image processing and computer vision applications. The ability to detect edges in images can be used for tasks such as object recognition, feature extraction, and image segmentation. However, as the size of images and videos increases, the computational cost of edge detection becomes a bottleneck. To address this issue, parallel computing can be used to speed up the execution of edge detection algorithms. One popular approach to parallel computing is the use of MPI or OpenCV. In this report, we present a parallel implementation of the Sobel operator, a widely used edge detection algorithm, using MPI.

2 Algorithm

The **Sobel** operator applies two **3x3 kernels**, one for detecting horizontal edges and one for detecting vertical edges. The algorithm is applied to each pixel of the input image, by convolving the kernel with the **3x3 neighborhood of the pixel**. The convolution is computed by multiplying each element of the kernel with the corresponding element of the neighborhood and summing the products. The final output for each pixel is the sum of the absolute values of the convolution for the two kernels. To make the jump from a sequential implementation to a parallel one it is necessary to define the way data is distributed amongst the nodes which is explained in the next section.

3 Implementation

The basic parallel implementation of the Sobel operator is based on the following steps:

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

Table 1: Sobel operator kernels

1. Node 0 (master node) reads the input matrix from a **P5 pgm image** and broadcasts its dimensions to all nodes.
2. Each node computes the number of rows of the matrix it will be responsible for, based on the total number of rows and the number of nodes.
3. Node 0 uses the scatter function to distribute the rows of the matrix among the nodes.
4. Each node applies the Sobel operator to the portion of the matrix assigned to it.
5. Each node uses the gather function to send the output matrix back to the master node.
6. Node 0 prints the final output matrix into a new pgm file.

Besides scattering rows from the master node to the rest of the nodes because of the nature of the Sobel operator which requires the **3x3 neighborhood of the pixel** to compute the result for said pixel it is necessary to include **Send/Receive** calls between the nodes. In practice, each node (**excluding node 0 and n-1**) sends its first row to the node behind its ranking and its last row to the one after.

The initial distribution of data to compute is as follows :

- Master node scatters **rows/size** amount of rows to each process.
- The left amount of rows **rows%size** is then attributed to the last node.

The aforementioned distribution pattern is simple to implement using the function scatter however for small images or in large node clusters and in the **worst case** scenario the last node takes almost two times the time any other node requires to finish computation, for example a number of 41 rows with 6 nodes results in the distribution illustrated in figure 1.

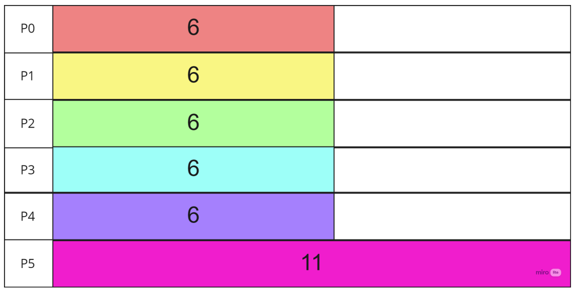

Figure 1: Simple load distribution of 41 rows on 6 nodes

To resolve the load balancing problem the **scatter** and **gather** functions were replaced by a more flexible variant **scatterv** and **gatherv** which require the user to specify the number of elements each node is gonna receive/send as well as the displacements of those elements in the data pointer.

Note : The distribution by rows was favored more since most images contain more rows than columns and therefore allow for better load balancing.

4 Optimizing Load Balancing

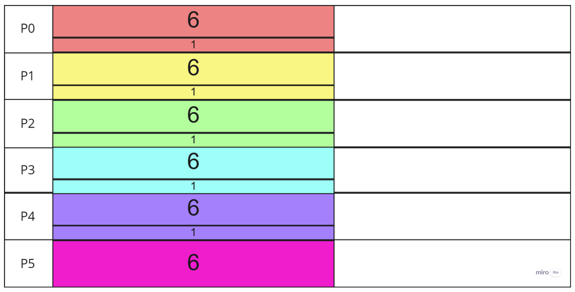

Figure 2: Optimized distribution of 41 rows over 6 nodes

A better balanced distribution can be achieved thanks to these variant functions. Instead of sending the left over rows to one node, we instead distribute them all over the nodes because it is guaranteed that the number of extra rows is less than the size of the cluster thanks to the modulo operation. The final configuration of such load balancing is as illustrated on figure 2.

Finally, it is important to ensure that the communication overhead between the nodes is kept to a minimum. This can be achieved by using non-blocking communication calls and overlapping computation and communication through the use of **Isend** or **Ireceive** functions.

5 Performance Evaluation

To evaluate the performance of the parallel implementation of the Sobel operator using MPI, we can compare its execution time to that of a sequential implementation. We can also compare the performance of our implementation with different number of nodes to see the effect of increasing the number of nodes on the performance of the algorithm but we will be only using an input pgm image P5 of size 2000x2000 and up (generated randomly for measuring purposes) to compensate for the lack of work per process.

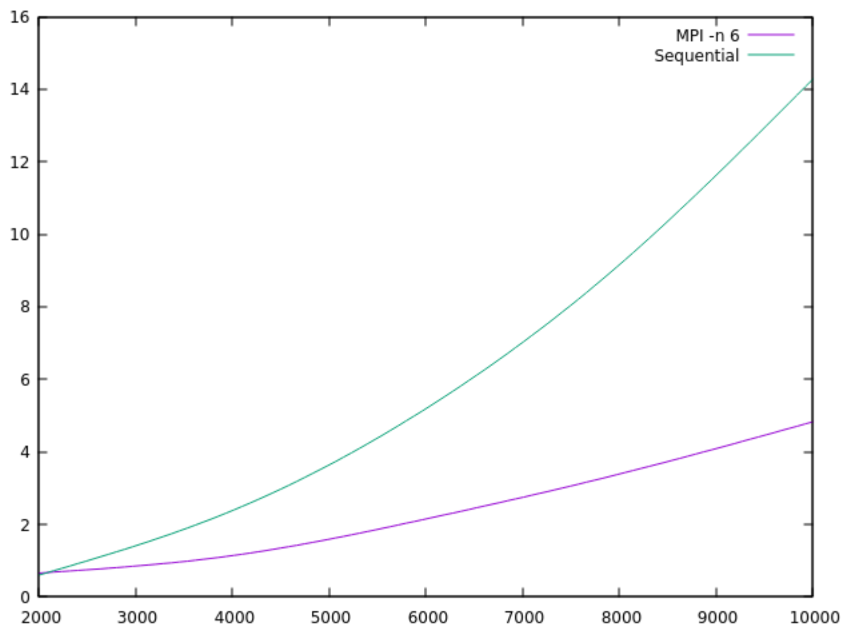

Figure 3: Performance comparison between Sequential and MPI implementations (size/time)

We can clearly see at the beginning (for image 2000x2000) that the communication overhead was still too much to ignore however as size continues to grow it immediately lost its effect on the overall execution time as show on figure 3.

6 Experimental Results

input	sequential	MPI (3 nodes)	speedup	MPI (6 nodes)	speedup
sample 256x256	0.03s	0.44s	NONE	0.4s	NONE
sample 640x426	0.08s	0.44s	NONE	0.44s	NONE
sample 1280x853	0.26s	0.54s	NONE	0.49s	NONE
sample 1920x1280	0.57s	0.65s	NONE	0.55s	3%
sample 5184x3456	4.01s	1.84s	117%	1.2s	234%
sample 10000x10000	18.28s	8.61s	119%	4.72s	287%

Table 2: Comparison of the performance between sequential and MPI (images are available at **https://filesamples.com/formats/pgm)**

Lets compare the execution time of the sequential and parallel implementations of the Sobel operator. The table 2 shows the effect of increasing the number of nodes or input dimensions of the pgm image on the execution time.

It appears that the computation time is negligible in front of the communication overhead when there is little to compute (i.e. small sized image) which is why we only start to gain speedup when dimensions go over **2000x2000** which is why a **10000x10000** size image was generated only for the purpose of measurement. This issue would not have surfaced if it this implementation used the **canny** edge detection algorithm instead which computes many times the detection many times to achieve the best results. Another important aspect shown in the results is that it is not always good to use more nodes for the same amount of work, and sometimes it might better to use less nodes when there is little work to do to minimize the communication overhead. Overall, the ratio work to the number of nodes is a very important aspect of the MPI usage.

7 Conclusion

In this report, we presented a parallel implementation of the Sobel operator using MPI in C++. We described the algorithm and the steps followed in the implementation. We also discussed the importance of load balancing in parallel implementations and presented different strategies that can be used to optimize the performance of the algorithm.

The results of the performance evaluation showed that the parallel implementation using MPI is significantly faster than the sequential implementation when the computation phase is significant enough. The results also showed that increasing the number of nodes improves the performance of the algorithm.