Parallelizing A Content-Aware Image Resizing with OpenSHMEM

By

Bukola Grace Omotoso

A thesis submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Middle Tennessee State University

May 2020

Thesis Committee:

Dr. Ferrol Aderholdt

Dr. Suk Seo

Dr. Chrisila Pettey

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Ferrol Aderholdt for the support he gave me in the course of doing this thesis. I appreciate his patience with me, the source code he shared with me and most importantly making his wealth of knowledge available to be able to do this thesis. Thank you for all your feedback and for creating time to supervise my thesis despite your tight schedule.

I want to appreciate Dr. Seo for setting me on the path of research through the research method class I took under her tutelage. Also, thank you for joyfully accepting to be on my thesis committee. I am so thankful to Dr. Pettey who did not turn down my offer to be on my thesis committee. You told me you will be happy to be on my committee and you kept to your words even when the demand of being an interim chair has made you so busy. Thank you for doing this.

My big thanks goes to my lovely Husband who has been patient with me when the demand of doing this thesis has crept into family time. I also appreciate you teaching me how to analyze my results with the matplotlib library. Big thanks to my praying Mom who continually prays for strength to cope with my graduate program. Those prayers has really brought me this far.

I knew about Seam carving through Grace Carrillo, a friend whose quest and determination to go beyond her limit has challenged me to research more about Seam carving. Thank you Grace for your support and challenging me to do this.

ABSTRACT

Cropping and Scaling are the most common image resizing techniques but neither of these considers the content of the image. Seam carving is a content-aware image resizing technique that removes or duplicates the least visible pixels from an image, thereby making it more effective than Cropping and Scaling. For large batches of images, it may be unrealistic to do Seam carving on one processing element due to memory constraints. To solve this problem, a parallel approach to image resizing that helps to ease data transfer between processing elements needs to be considered.

Partitioned Global Address Space (PGAS) programming models have been attracting attention as a parallel computing model and it is often used to implement one-sided Remote Memory Access (RMA) from multi-host systems, such as computer clusters. OpenSHMEM is a distributed, PGAS programming model that has light-weight semantics and high-performance RMA and atomic memory operations.

In this thesis, we parallelize Seam carving using Pthreads, OpenSHMEM and MPI. We evaluate the relative performance gained with multiple threads and processing elements (PEs).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I.
INTRODUCTION

The transfer of images between devices has made the demand for an effective image resizing technique a necessity. Image resizing is increasingly important for exchanging and sharing pictures with different resolutions between devices of various sizes. The need for responsiveness has made an effective image resizing technique of great importance. For example, the size of an image displayed on a website opened from a desktop computer will be different if the web page were opened from a mobile device.

Scaling and Cropping are the most common image resizing techniques but neither considers the content of the image. Scaling enlarges or reduces the size of an image by increasing or decreasing the number of pixels in the image. Cropping is the removal of some peripheral areas from an image. Standard image scaling is not sufficient since it does not consider the image content when resizing. Cropping may remove visual information when important structures lie at the edges of an image [27]. Seam carving [14] is a content-aware image resizing technique that repeatedly removes or duplicates the least visible pixels from an image for reduction and enlargement, respectively. It is a more effective image resizing technique than cropping and scaling because it preserves the image content and causes little or no visible distortion to images. Figure 1 shows a comparison between image resizing with seam carving, scaling and cropping.

Seam carving uses an energy function that defines how important the pixels of an image are. “A seam is a connected path of low energy pixels crossing the image from top to bottom, or from left to right” [14]. By removing or inserting seams, the height and width of an image can be reduced or enlarged. Seam carving is getting more attention as it is being used in open source imagery applications including digiKam [47], ImageMagick [22] and Photivo [17]. For batches of high resolution and large images like satellite images, implementing Seam carving in real-time on a single-threaded CPU may become infeasible. For large

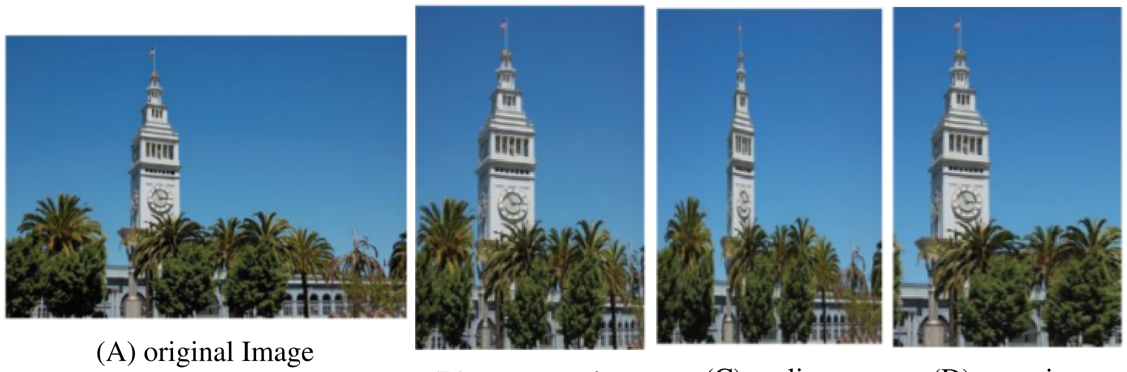


Figure 1: An example of image resizing [16].

images, Seam carving may take a significant amount time. For example, to reduce the height of a 1024 x 640 image by half takes more than 15 seconds with a 2GHz Intel Xeon processor [35].

Efficiently implementing Seam carving in parallel can be challenging. Dynamic programming is used to identify an optimal seam during image resizing, but the computational dependence of dynamic programming makes it difficult to parallelize Seam carving. Also, a large number of irregular memory access patterns is used when computing intermediate results for Seam carving, this worsens the program performance significantly [35]. For example, the size of the image keeps changing after every seam removal and all these changes need to be handled at run time.

Using a distributed programming model such as OpenSHMEM helps to implement Seam carving in a way that data transfer among Processing Elements (PEs) is simplified. This is made possible through the use of Remote Memory Access (RMA) using PUT and GET operations.

OpenSHMEM's Application Programming Interface (API) and its support for RMA makes it a great choice for parallel, High Performance Computing (HPC) applications. OpenSHMEM's API is constantly evolving along with current scientific and hardware changes [48]. OpenSHMEM has good vendor support, OpenSHMEM compliant solutions have been developed by vendors to help users write portable OpenSHMEM code that ensures programs have the capability to run on many platforms without bothering about specific vendor differences in implementation [4].

This work focuses on the adaptation of Seam carving algorithm to the Partitioned Global Address Space (PGAS) programming model with OpenSHMEM and has made the following contributions:

1. We design and prototype the parallelization of Seam carving with OpenSHMEM.
2. We evaluate the performance improvement obtained with the parallelization of Seam carving in comparison with a serial implementation.
3. We design and prototype the parallelization of Seam carving with MPI-3 one-sided and Pthreads. We also make performance comparisons between our Pthreads, OpenSHMEM and MPI prototypes.

This paper is organized as follows: Chapter 2 presents the background to this work. Chapter 3 discusses related work with the parallelization of Seam carving. Chapter 4 explains the methods used to achieve some of the contributions of this work. The results from the implementation is discussed in Chapter 5. Our conclusion is presented in Chapter 6.

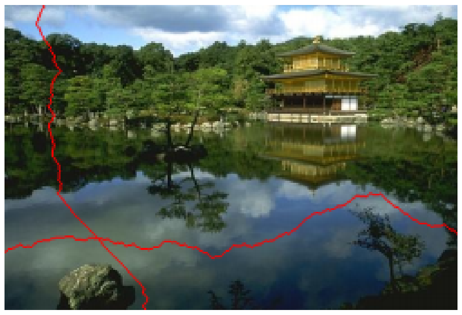
CHAPTER II.

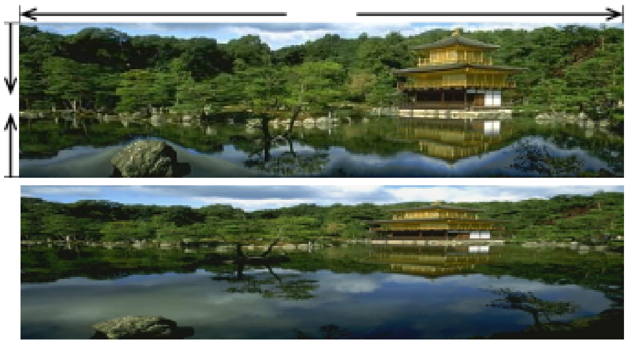
BACKGROUND

Image resizing is a standard tool in many image processing applications. It works by uniformly resizing the image to a target size. Recently, there is a growing interest in content-aware image resizing that seeks to change the size of the image while keeping the important features intact. A computational approach to detecting meaningful visual structure in an image through eye-tracking was presented by DeCarlo and Santella [25]. The study showed that when a human user gazes at an image for a short period of time, meaningful content can be derived from the image by using an eye-tracker to gather data and make predictions about which part of the image carries important information. Wang et al. [49] presented a content-aware “scale-and-stretch” warping method that computes a significant map by using gradient and salience based measures to characterize the importance of all image pixels. In the study, the distortion caused by image resizing is distributed to homogeneous regions of the image, thereby making the distortion less visible. Another content-aware image resizing technique that has been proposed uses Mesh Parametrization in a study carried out by Guo et al. [30]. In the study, the image to be resized is represented by a mesh that is consistent with the image structures, mesh transformation adapts images to a desired size with minimal visual distortion.

Seam Carving - A Content-Aware Image Resizing Technique

Avidan and Shamir proposed a content-aware image resizing called Seam carving [14] that uses an energy function to determine the important regions in an image. Seam carving repeatedly uses seams insertion or seams removal to enlarge or reduce the size of an image. “A seam is an optimal 8-connected path of pixels on a single image from top to bottom, or left to right, where optimality is defined by an image energy function” [14]. Horizontal seams and vertical seams are considered for image resizing that affects the height and width of an image, respectively. The red lines in Figure 2A shows the horizontal and vertical


(A) Vertical and Horizontal seams in an Image [14]


(B) Top: Seam carving, Bottom: Scaling [14]

seams identified in an image. Repeatedly removing those seams gives the image on top in Figure 2B. Comparing that image with the one below shows Seam carving does image resizing more effectively than scaling by reducing the level of distortion from resizing the original image. Seam carving has been applied to various areas including medical image resizing [18, 37, 43] , image compression [46, 13], and digital image forensics [36, 50]. Ko et al. [18] proposed using saliency strength map and Seam carving for the analysis of white blood cells. In the research, the energy map for Seam carving was generated by using the structural properties of white blood cells. The result of the study confirmed that better results were produced compared to other conventional methods of medical image resizing. Tanaka et al. [46] used Seam carving to improve the results of image dilusion. In the study, images are concentrated using Seam carving and storing the low energy seams that are carved out. The reverse of that process, *image dilusion* is achieved by *interpolating* the less important regions in the image until the original image is achieved. A recent forensics study was conducted by Lu and Niu [36] to consider seam carved images when detecting image forgery. In the study, a “novel binary pattern called local neighbourhood magnitude occurrence pattern (LNMOP) for the detection of Seam carving” was proposed. A similar study was done by Toshihiko et al. [50] with JPEG images.

<u>The OpenSHMEM PGAS Programming Model</u>

A Partitioned Global Address Space (PGAS) programming model works with shared and distributed memory parallel machines. The shared memory is available to all processors on a single node, whereas the distributed memory is owned by each processor and can be shared among other processors through a network [12]. The shared memory model aspect of PGAS models help to improve programmer productivity while its distributed side that differentiates between local and remote data access helps to optimize performance [24]. PGAS programming languages include Co-Array Fortran [39], Titanium [51], UPC [15], X10 [44], Chapel [20] and OpenSHMEM [4]. The development and usage of OpenSHMEM has increased over the past few years compared to the other PGAS programming languages.

OpenSHMEM implements the PGAS programming model by defining private data objects that are accessible to each Processing Element (PE) and remote data objects that are accessible to all participating PEs in the system. The remotely accessible data objects are allocated on a Symmetric Heap and are called Symmetric Data [38]. Symmetric Heap is a collection of memory partitions that are reserved for storing distributed data structures. [38]. Data transfer between remote data objects are one-sided in nature which implies that a local PE does not need to wait for an acknowledgement from a remote PE during data transfer operations. Communication and synchronization operations between the private and remote data objects are enhanced through the simple and standard OpenSHMEM's Application Programming Interface (API) [4] that is constantly being improved to keep up with current scientific and hardware changes [48]. OpenSHMEM programs can be implemented on various platforms without having issues with vendor differences because of its open specification [3].

OpenSHMEM's initial version 1.0 was released in 2012, its last stable version 1.4 was released in 2017, the draft for version 1.5 was recently released in February, 2020. The support for *collective operations*, *distributed locks*, *data transfer operations*, *atomic memory opera-*

tions, *collective and point-to-point synchronization mechanisms* was initially provided by OpenSHMEM 1.0 specification [5]. In June 2014, the 1.1 specification [6] of OpenSHMEM was released. Some of the major changes in that version is better documentation and allowing none standard API extension through the use of *shmemx.h* header file. In March 2015, the version 1.2 OpenSHMEM specification was released [7]. The release clarified some API descriptions, provided standardized namespaces, introduced a new initialization, finalization routine and an early exit routine using *shmem_init*, *shmem_finalize* and *shmem_global_exit* respectively. OpenSHMEM 1.3 specification [8] was released in February 2016 and some of the major changes that came with that release were the introduction of non-blocking Remote Memory Access (RMA) operations, atomic *Put* and *Get* operations. *C11* type interfaces for RMA and Atomic Memory Operations (AMO) also came with this release. In December 2017, OpenSHMEM 1.4 specification [9] was realeased and some major changes in that release include communication management using *contexts*, support for multithreading, shmem_sync and shmem_calloc. *C11* type interfaces for point-to-point synchronization came with that release. In February 2020, the draft for OpenSHMEM 1.5 specification [10] was released. Currently, there is an ongoing effort for the release of OpenSHMEM 1.5 specification in 2020.

OpenSHMEM has been adapted for use with GPU through NVSHMEM [1] implementation. "NVSHMEM is an implementation of the OpenSHMEM standard for NVIDIA GPU clusters which allows communication to be issued from inside GPU kernels" [41]. OpenSHMEM applications can automatically benefit from new GPU architectures and improvement in performance by using NVSHMEM [2]. Santella et al. implemented Breadth-First Search (BFS) using NVSHMEM, the result of the implementation shows a speedup of 75% compared to a MPI implementation. The speedup is due to reduced overhead in synchronizing between GPU and the CPU [42].

CHAPTER III.
RELATED WORK

A lot of work has been done to parallelize and improve the performance of the Seam carving algorithm. In this section, we will explore some of the efforts towards the parallelization of Seam carving. We will also discuss some of the algorithms that have been improved through the adaptation of the OpenSHMEM PGAS model.

Parallelizing Seam Carving on the GPU

Seam carving is time consuming because of its irregular memory patterns and complex computational model. This reduces its usage for real time image resizing [35]. The parallelization of Seam carving is one of the ways to speedup the execution time of Seam carving. However, parallelizing Seam carving is a challenge because the Seams identification phase of Seam carving involves dynamic programming.

Duarte and Sendag [27] used CUDA enabled GPU to accelerate Seam carving for videos and images. Result shows that the energy computation phase of Seam carving attained *100x* and *14x* speedup when compared to the faster single-threaded and multi-threaded implementations, respectively. Also, the overall execution times had *6x* and *2x* speedup when compared to the single-threaded and multi-threaded implementations, respectively. The Seam carving algorithm has also been implemented on Multi-GPU platform by Kim et al. [35]. The result from the implementation shows *140x* speedup on a two-GPU system compared to the sequential version of the algorithm [35].

Cešnovar et al. [19] also parallelized the Seam carving algorithm on the GPU by optimizing single seam removal which is based on the properties of the GPU they used. The study considered the performance of GPU implementations in comparison to sequential implementation. Result from the study shows decreased execution time compared to the single CPU version when the image size is above 150 x 150 pixels. Keiss et al. [34] implemented a fast parallel video resizing by combining Seam carving and cropping on a

CUDA enabled GPU. The result showed *10.5x* speedup by reducing a 960 x 540 video by quarter than the already efficient CPU implementation.

Seam carving has also been implemented in OpenCL [28]. It was observed that running the algorithm on the GPU integrated with an Intel Kaby Lake processor, the OpenCL implementation was almost *180x* faster than the serial implementation. The Seam carving algorithm was improved by Chiang et al. [21] to make the algorithm highly parallelizable and suitable for multicore architecture through multi-seam identification. The study was extended to achieve real time video re-targeting with GPU acceleration and the results show that the multi-seam algorithm was highly parellelizable compared to the original seam carving algorithm.

Other Parallel Implementations of Seam Carving

Stulz [45] parallelized the Seam carving algorithm with MPI and the results showed the energy map calculation scaled nearly perfectly with the increased number of processors, and the minimum path calculations were barely sped up for smaller images, but approached the expected limit of 2x speedup for larger images. Ali and Hayat [11] proposed a Restricted Shared Memory (RSM) model that does not require data movement between Processing Elements(PEs) when parallelizing Seam carving on a distributed system. The authors claim the “RSM model offers little or no memory access contention between one memory segment and another memory segment” [11]. The result shows that performance increases linearly with the number of PEs used and does not degrade after a certain peak value due to minimal shared memory overhead and communication cost.

OpenSHMEM has not be used for paralleling Seam carving. The focus of this thesis is to parallelize Seam carving with OpenSHMEM. We evaluate the performance of Seam carving with increasing PEs and make comparisons with a one-sided MPI implementation.

Parallelization with OpenSHMEM

OpenSHMEM is an ideal solution for High Performance Computing applications that are characterized by irregular and fine-grain communication, computation and memory access. Graph 500 is an one of such applications and OpenSHMEM has been adapted to its implementation [29]. D'Azevedo and Imam [23] implemented a prototype of Graph500 using OpenSHMEM based on the original MPI-2 one-sided implementation. The OpenSHMEM implementation was tested on an SGI cluster and a titan machine and result shows the competitiveness of OpenSHMEM implementation against the MPI-2 one-sided version. Another similar implementation of Graph500 using MPI and OpenSHMEM was carried out by Jose et al. [33] in which the "Concurrent Search" kernel of Graph500 was designed. Performance evaluation from the implementations using *MVAPICH2-X* shows 59% reduction in execution time compared to the best performing MPI only design using 8192 cores. The study [29] carried out by Grossman et al. explored the synergy between the communication patterns of Graph500 and the capabilities of OpenSHMEM. Comparisons between OpenSHMEM implementation and other state-of-the-art implementations of Graph500 was made. Result shows improved scaling of Graph500's Breadth First Search (BFS) kernel out to 1,024 nodes of the Edison supercomputer, achieving 2.5x performance improvement compared to the best performing non-OpenSHMEM implementation at that scale.

OpenSHMEM has been adapted to data-intensive computing applications like sort algorithms. Jose et al. [32] implemented the k-way parallel sort on a hybrid PGAS distributed system that uses MPI and OpenSHMEM. The results from the implementation shows a 45% performance improvement with 8192 processes compared to an MPI only design. Also, a 7x speedup was achieved when compared with Hadoop-based sort with the same number of processes with each having 1024 cores. The NAS Parallel Benchmarks (NPB), *a small set of programs designed to help evaluate the performance of parallel supercomputers* [26] was ported from MPI-1 to OpenSHMEM in a study carried out by

Pophale et al. [40]. Performance and scalability evaluations from the experiment shows that without optimization, OpenSHMEM has a comparable performance with MPI-1 and better performance than MPI-2.

CHAPTER IV.

DESIGN AND IMPLEMENTATION

This section explains the methods we employed to parallelize Seam carving. Seam carving can be used for image reduction and enlargement, but our focus will be on image reduction only because image enlargement is similar to image reduction. While image reduction entails removing low energy seams repeatedly until the desired dimension is reached, image enlargement involves replicating the low energy seams until the desired dimension is reached. The methods described in this chapter for image reduction are applicable to image enlargement as well.

The Process

Image resizing involves some preliminary processes before applying Seam carving for the actual resizing. The image needs to be converted to a form that makes each pixel accessible. We first converted the image to be resized to its RGB values and stored as a one-dimensional array of unsigned characters with size (w x h x 3). In order to make the Seam carving process easy, we converted the 1D RGB buffer to a 2D (w x h) array with each cell representing a pixel and comprising 3 values which corresponds to the RGB values of the pixel. Once the RGB buffer is generated, Seam carving can be applied. Seam carving is a three step process:

1. **Energy Computation**: The first step with seam carving is to compute the energy of each pixel in the image before the actual image resizing. The energy values indicate the importance of each image pixel in terms of visibility. We will explain this process more in the next session.

2. **Seams Identification**: After defining the energy of each pixel, the next step is to traverse the energy matrix, treating some select entries as reachable from top to bottom for vertical seams identification, or left to right for horizontal seams identification.

3. **Seams Removal**: The pixels along the seam are removed for image reduction. For vertical Seam carving, the width w of the image will be reduced to $(w-1)$ and the height of the image will be reduced to $(h-1)$ for horizontal Seams carving.

   Repeating the steps above will reduce an image to any desired dimension.

Energy Computation

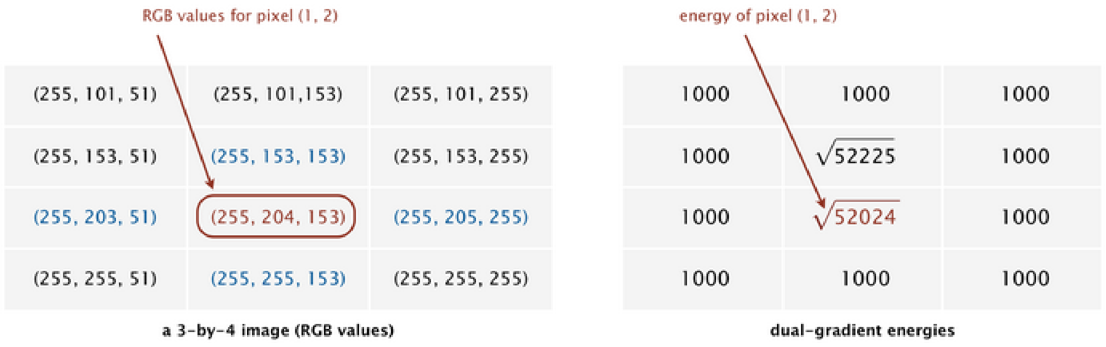
We use a dual-gradient energy function for our Seam carving implementation. Although there are several energy functions that can be used as suggested by Avidan and Shamir [14]. The dual-gradient energy is a simple energy function and it is the sum of the x gradient and y gradient of the image pixel as defined below; given an image pixel I,

$$e(I) = \left|\frac{\partial}{\partial x}I\right| + \left|\frac{\partial}{\partial y}I\right| \tag{4.1}$$

The equation above is used to compute the energy of each pixel relative to its surrounding pixels. Energy values measure how visible a pixel is in relation to its neighbouring pixels. The x gradient is the differences between a specific pixel and its neighbouring left and right pixels. The y gradient is the differences between a specific pixel and its neighbouring top and bottom pixels.

The energy values of the pixels of the image borders or edges are given a strictly larger constant value than the energy value of any interior pixel in the image. This makes all edges have the same chance of being part of the seam because a seam is a path from one edge to another. Since no interior energy value can be greater 625 which can be achieved if we assume the difference between a specific pixel and its neighbouring pixels is 255. We have $255^2 + 255^2 + 255^2$ for the x gradient. $255^2 + 255^2 + 255^2$ for the y gradient. Summing up those gradient values and finding the square root gives approximately 625. We assigned a value of 1000 to these pixels.

Figure 3 shows an example of energy value computation in a 3x4 image. The pixel of an image is referenced by (w,h) where w is the width and h is the height.The dimensions have zero indexes.


RGB values for pixel (1, 2)

(255, 101, 51)	(255, 101,153)	(255, 101, 255)
(255, 153, 51)	(255, 153, 153)	(255, 153, 255)
(255, 203, 51)	(255, 204, 153)	(255, 205, 255)
(255, 255, 51)	(255, 255, 153)	(255, 255, 255)

a 3-by-4 image (RGB values)

energy of pixel (1, 2)

1000	1000	1000
1000	$\sqrt{52225}$	1000
1000	$\sqrt{52024}$	1000
1000	1000	1000

dual-gradient energies

Figure 3: An example of dual-gradient energy in a 3x4 image [31].

Seams Identification

A seam is a connected path of low energy pixels crossing the image from top to bottom (vertical seam), or from left to right (horizontal seam). With the energy values computed, the pixels with lowest energy seam that either goes vertically or horizontally can now be identified. We used two data structures, which are the (w x h) distance array and the (w x h) edges array. The distance array values are initialized to maximum values. Moving through each pixel of the image updates the distance values to the shortest distance taken to reach an image pixel by looking at the neighbours of that pixel. The edges array keeps the index of the pixel that provides the shortest distance. It has values -1 (left bottom neighbour), 0 (bottom neighbour) and 1 (right bottom neighbour). After traversing through the image, the last rows in the distance array match the shortest distances taken to reach the last row of pixels in the image. The column that has the shortest distance will be the destination of the seam path. The edges array is used to backtrack from this destination pixel back to the first row to get the seam path.

In an *n* x *m* image *I*, a vertical seam is defined to be

$$\begin{aligned} s^x = \{s_i^x\}_{i=1}^n = \{(x(i), i)\}_{i=1}^n, \\ s.t.\ \forall i, |x(i) - x(i-1)| \leq 1, \end{aligned} \tag{4.2}$$

where x is a mapping from $[1, ..., n]$ to $[1, ...m]$.

An horizontal seam is defined as follows;

$$\begin{aligned} s^y = \{s_j^y\}_{j=1}^m = \{(j, y(j))\}_{j=1}^m, \\ s.t.\ \forall j, |y(j) - y(j-1)| \leq 1, \end{aligned} \tag{4.3}$$

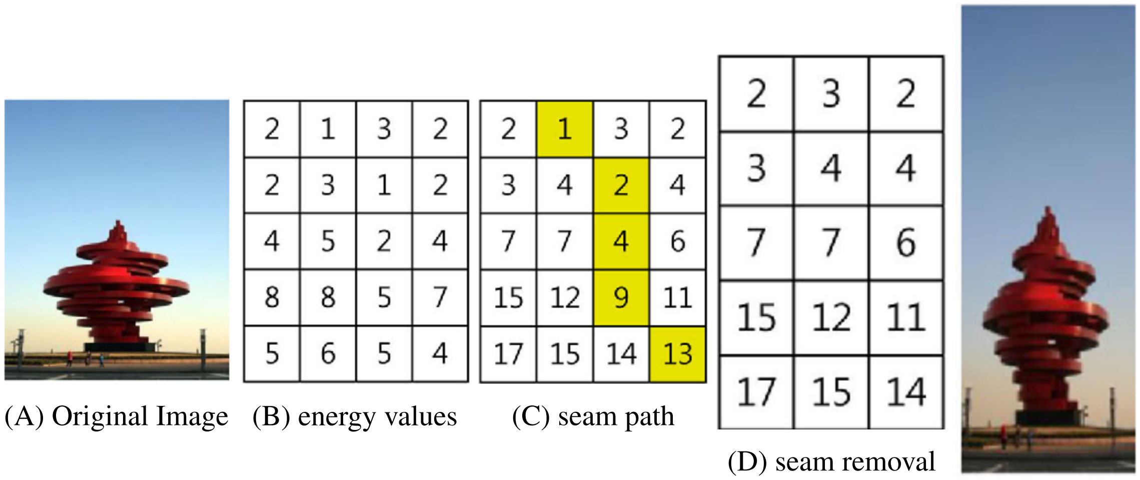
where y is a mapping from $[1, ..., m]$ to $[1, ...n]$.

The lowest-energy seam may not pass through all the lowest-energy pixels in the image. The lowest energy seam is the total energy of the seam being minimized, not the individual pixel energies. For vertical seams, we backtrack from the pixel in the last row whose cumulative energy is minimum to obtain an optimum vertical seam. For horizontal seams, we can also backtrack from the pixel in the last column whose cumulative energy is minimum to obtain an optimum horizontal seam. However, in this thesis, for horizontal seams identification, we transposed the image and then applied the procedures for vertical seams to identify the optimal horizontal seams.

Seams Removal

We remove from the image the pixels along the seam path. For vertical Seam carving, the width of the image will be reduced to $w - 1$ and the height of the image will be reduced to $h - 1$ for horizontal seam carving. Removing the pixels along that path and shifting the pixels to the right of the seam by one pixel to the left reduces the image width by one pixel.

Figure 4 shows the different stages of Seam carving to reduce the width of an image by one pixel. Figure 5 shows our result of removing 50 vertical seams from a 425 x 428 image producing a resized image of 375 x 428.



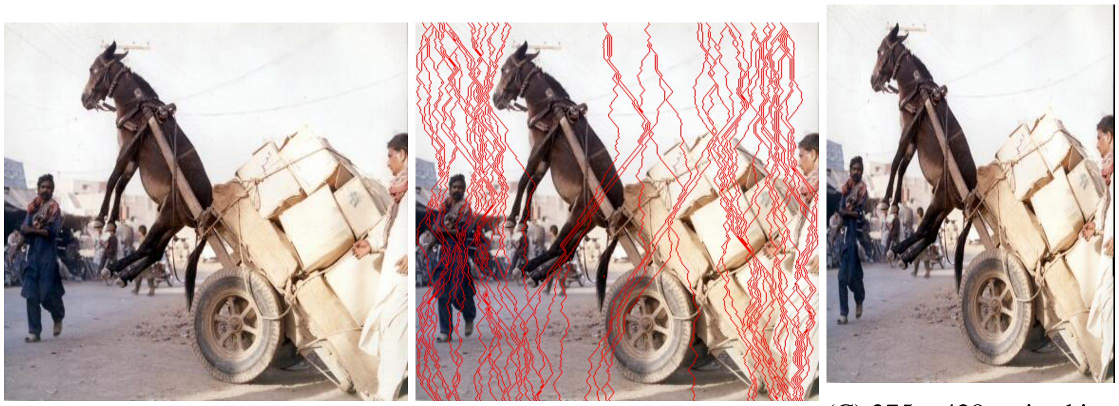
2	1	3	2
2	3	1	2
4	5	2	4
8	8	5	7
5	6	5	4

2	1	3	2
3	4	2	4
7	7	4	6
15	12	9	11
17	15	14	13

2	3	2
3	4	4
7	7	6
15	12	11
17	15	14

(A) Original Image (B) energy values (C) seam path (D) seam removal (E) new image

Figure 4: Image resizing with Seam Carving [35].



(A) 425 x 428 Original Image (B) 50 vertical seams (C) 375 x 428 resized image

Figure 5: Carving out 50 verticals seams from an image

Algorithmic Description of Seam Carving

Problem Description

Given an image of size w x h, find the energy values of the image pixels, identify the pixels along the path from top to bottom or left to right of the image with the lowest cumulative energy. Repeatedly remove this path until the desired width or height is obtained.

Inputs

- The image to be resized
- The number of vertical or horizontal seams s to be removed from or to the image

Output

- A reduced image $(w-s)$ x h or w x $(h-s)$ for image reduction

The Algorithm: Algorithm 1-3. The flowchart for the Seam carving algorithm is shown in Figure 6.

Algorithm 1 The Seam Carving Algorithm

```
function MAIN
    $width = image\_width$
    $height = image\_height$
    convert image to RGB buffer
    $buffer = RGBimagebuffer$
    $orientation = vertical$ or $horizontal$
    generateEnergyMatrix(width,height, buffer, orientation)
    identifySeam(width,height)
    removeSeam()
end function
```

Algorithm 2 The Seam Carving Algorithm for Energy Computation

```
 1: function COMPUTEENERGY($x$,y, buffer)
 2:     $Red_x = Red_{x-1} - Red_{x+1}$
 3:     $Green_x = Green_{x-1} - Green_{x+1}$
 4:     $Blue_x = Blue_{x-1} - Blue_{x+1}$
 5:     $squarediff_x = Red_x^2 + Green_x^2 + Blue_x^2$
 6:     $Red_y = Red_{y-1} - Red_{y+1}$
 7:     $Green_y = Green_{y-1} - Green_{y+1}$
 8:     $Blue_y = Blue_{y-1} - Blue_{y+1}$
 9:     $squarediff_y = Red_y^2 + Green_y^2 + Blue_y^2$
10:     return sqrt$(squarediff_x + squarediff_y)$
11: end function

12: function GENERATEENERGYMATRIX($width$,height, $buffer, orientation$)
13:     for $row \leftarrow 1, height - 1$ do
14:         for $column \leftarrow 0, width - 1$ do
15:             if $orientation = vertical$ then
16:                 $energyArray[row][column] = computeEnergy(column, row, buffer$
17:             else
18:                 $energyArray[row][column] = computeEnergy(row, column, buffer$
19:             end if
20:         end for
21:     end for
22: end function
```

Algorithm 3 The Seam Carving Algorithm for Seams Identification

```
 1: function IDENTIFYSEAM($width, height$)
 2:     for $row \leftarrow 0, height$ do
 3:         for $col \leftarrow 0, width$ do
 4:             relax(row, col, dist, edge, width)
 5:         end for
 6:     end for
 7:      return backtrack($dist$,height, width)
 8: end function

 9: function RELAX($row$,col,$dist$,edge,width)
10:     $nextrow \leftarrow row + 1$
11:     for $i \leftarrow -1, 0$ do
12:         $nextcol \leftarrow col + 1$
13:         if $nextcol < 0$ or $nextcol \geq width$ then
14:             continue
15:         end if
16:         $tmp \leftarrow dist[row][col] + energy[nextrow][nextcol]$
17:         if dist[nextrow][nextcol]$>$ $tmp$ then
18:             dist[nextrow][nextcol]$\leftarrow$ $tmp$
19:             edge[nextrow][nextcol]$\leftarrow$ $i$
20:         end if
21:     end for
22: end function

23: function BACKTRACK($dist$,height,width)
24:     $minCol \leftarrow 0$
25:     $minDist \leftarrow \infty$
26:     for $col \leftarrow 0, width$ do
27:         if dist[height-1][col]$\leq$ $minDist$ then
28:             $miDist \leftarrow dist[height-1][col]$
29:             $minCol \leftarrow col$
30:         end if
31:     end for
32:     for $row \leftarrow height - 1$,0 do
33:          seams[row]$\leftarrow$minCol
34:          $minCol \leftarrow minCol - edge[row][minCol]$
35:     end for
36: end function
```

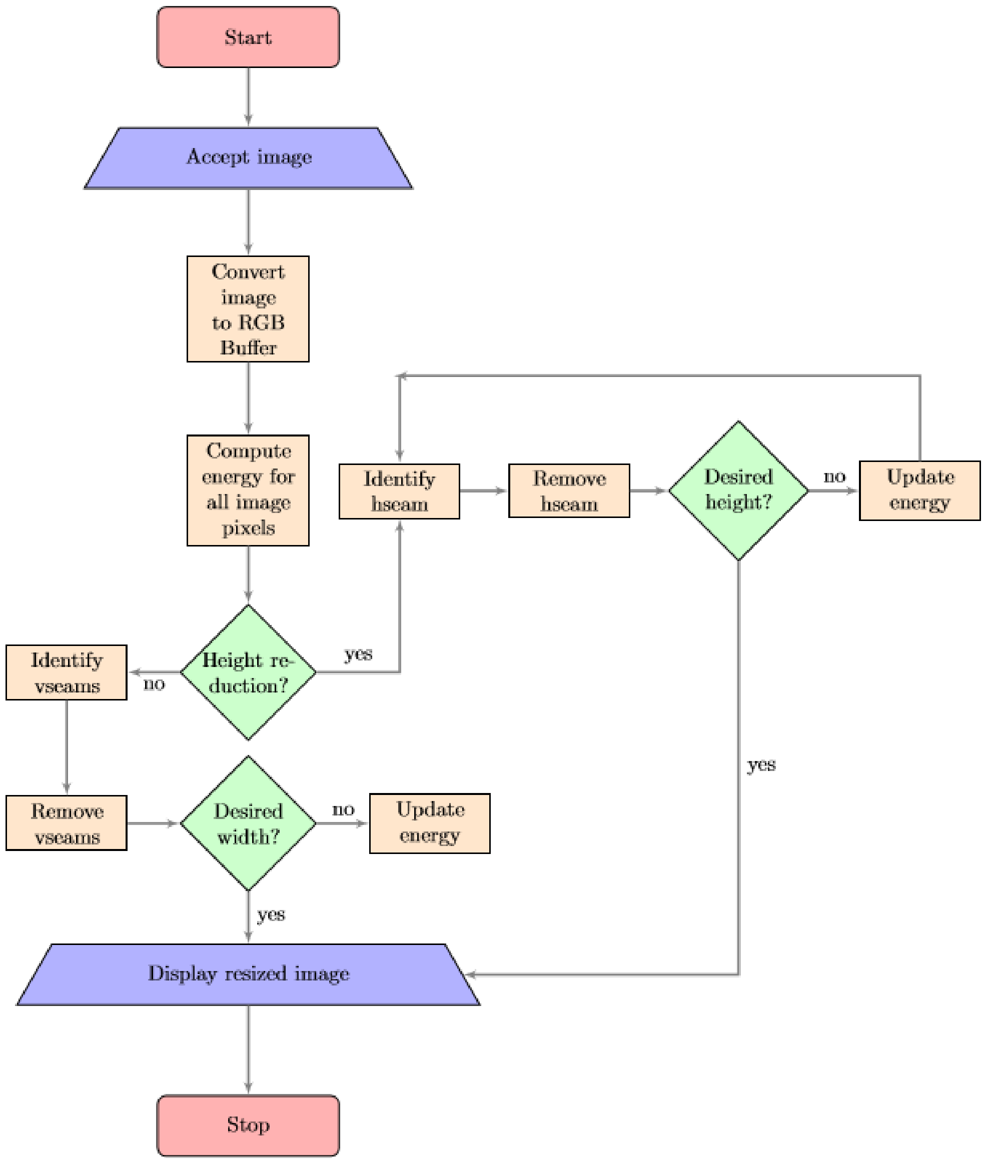


Figure 6: The Seam carving flowchart for Image reduction

<u>**Parallelizing Seam Carving**</u>

We parallelized Seam carving using Pthreads, OpenSHMEM and MPI. The energy computation phase of Seam carving is embarrassingly parallel in all cases since the RGB buffer values do not change across processing elements or threads. Although, the computational dependence of the seams identification phase makes it a bit challenging to parallelize with

OpenSHMEM and MPI, we found a way around it using Remote Memory Access (RMA) operations. Parallelizing the seam removal phase of Seam Carving did not give us much speedup so we did not consider that in our parallelization.

Parallelizing Seam Carving with Pthreads

We used a struct that has attributes start_col and stop_col to pass data to threads on what part of data they are to process. Each thread compute the energy values of their assigned buffer and are terminated when done with the computation. Seams identification is not very straight-forward. Threads are spawned and each thread initializes its distance array with the base energy if the row or column is on the edge, distance values are set to zero for inner rows and columns. Each thread hits a barrier when done with each row. This makes all thread have access to each other's data when computing distances for the next row and this is how the computational dependence for Pthreads is solved. We pass the control to the main thread to use backtracking to identify the seam and remove it. Algorithm 4 and 5 show the Pthreads algorithms for Seam carving energy computation and seams identification, respectively.

Algorithm 4 Pthreads Parallel Seam Carving Energy Computation

```
1:  function GENERATEENERGYMATRIX($height, startcol, stopcol, buffer$, orientation)
2:      for $row \leftarrow 1, height - 1$ do
3:          for $column \leftarrow start\_col, stop\_col - 1$ do
4:              if $orientation = vertical$ then
5:                  $energyArray[row][column] = computeEnergy(column, row, buffer$
6:              else
7:                  $energyArray[row][column] = computeEnergy(row, column, buffer$
8:              end if
9:          end for
10:     end for
11: end function
```

Parallelizing Seam Carving with OpenSHMEM

We parallelized the energy computation and seams identification phases of Seam carving with OpenSHMEM by columns.

Energy Computation The energy computation phase of Seam carving is embarrassingly

Algorithm 5 Pthreads Parallel Seam Carving Seams Identification

```
1: function IDENTIFYSEAMS($width, height$)
2:     for $row \leftarrow 0, height$ do
3:         for $col \leftarrow 0, width$ do
4:             relax(row, col, dist, edge, width)
5:         end for
6:         BARRIER
7:     end for
8: end function

9: function RELAX($row$,col,$dist$,edge,width)
10:     $nextrow \leftarrow row + 1$
11:     for $i \leftarrow -1, 0$ do
12:         $nextcol \leftarrow col + 1$
13:         if $nextcol < 0$ or $nextcol \geq width$ then
14:             continue
15:         end if
16:         $tmp \leftarrow dist[row][col] + energy[nextrow][nextcol]$
17:         if dist[nextrow][nextcol]$>$ $tmp$ then
18:             $dist[nextrow][nextcol] \leftarrow tmp$
19:             $edge[nextrow][nextcol] \leftarrow i$
20:         end if
21:     end for
22: end function
```

parallel as there is no computational dependence since at this phase, we only use the buffer that has the RGB values for all image pixels. All PEs have access to these buffer values. The image pixels are evenly distributed across PEs by columns so all PEs can be in sync when done with each row. Each PE computes the energy values of the pixels assigned to it independent of other PEs. For multiple seams removal, after removing the first seam, only the pixels surrounding the identified seams will have new energy values because they now have new neighbours. Only these values will need to be recomputed. The algorithm for Energy computation is shown below.

Algorithm 6 OpenSHMEM Parallel Seam Carving Energy Computation

```
1: function GENERATEENERGYMATRIX($height, startcol, stopcol, buffer$, orientation)
2:     for $row \leftarrow 1, height - 1$ do
3:         for $column \leftarrow start_col, stop_col - 1$ do
4:             if $orientation = vertical$ then
5:                 $energyArray[row][column] = computeEnergy(column, row, buffer$
6:             else
7:                 $energyArray[row][column] = computeEnergy(row, column, buffer$
8:             end if
9:             PUT($energyArray[row][column]$, $energyArray[row][column]$, 1, master)
10:        end for
11:    end for
12: end function
```

Seams Identification We used two data structures to manage the identification of a seam in an image. The distances array is a 2D array that keeps the minimum distance we can transverse to reach a pixel. The edges array keeps track of the top neighbouring pixel that gave us the minimum distance of a pixel. The values in the edges array can be either of -1, 0 or 1. A value of -1 means the left top pixel was used to attain a minimum distance, a value of 0 means the immediate top pixel was used and a value of 1 means the right top pixel gave the minimum distance. The seams identification process is not parallel friendly because of computational dependence. Each PE depends on some values from the previous and next PE. Unlike energy computation, with multiple seams removal, all pixels have to be evaluated for a new seam path. To handle computational dependence, we have leveraged the synchronization functions provided by OpenSHMEM and parallelization by columns also helps to tackle the problem with dependency. Parallelizing by columns makes us control what happens when all PEs are done with their assigned columns before moving to the next row. As we did with pthreads, all PEs come to a barrier at the end of each row. Whenever new values for distances and edges array are computed in the relax function, the master PE is updated with those new values. After every iteration of relaxing each row, the master PE puts its new values to other PEs by using PUT operations, this makes all PEs have updated values that will be needed for the next iteration. We used a barrier at the end of each iteration to ensure all PEs get all the updates from PE 0 before moving to the next iteration. We used backtracking to complete the seams identification after traversing through the energy matrix. To backtrack, we considered the column with the minimum distance value on the last row of the distance matrix, we then traced that column to the first row of the matrix using adjacent columns and subtracting the corresponding edges values from those columns. The master PE handled the backtracking. The algorithm for seams identification with OpenSHMEM is shown in Algorithm 7.

For seams removal, we applied the seams generated from seams identification to carve out the pixels that fall in the seam path, we did not parallelize the seams removal. The master PE handled seams removal.

Parallelizing Seam Carving with MPI

We parallelized the energy computation and seams identification phases of Seam carving with one-sided MPI. We could not directly use PUT and GET operations as we did with OpenSHMEM but had to access a memory area on the target that is remotely accessible to all PEs. This memory area is known as a Window. We allocated a window for each of the data structures accessible to all PEs. We had one window for the energy array, one window for the distances array and another window for the edges array.

Algorithm 7 OpenSHMEM Parallel Seam Carving Seams Identification

```
function IDENTIFYSEAMS($width, height$)
    for $row \leftarrow 0, height$ do
        for $col \leftarrow 0, width$ do
            relax(row, col, dist, edge, width)
        end for
        BARRIER
        if RANK = MASTER then
            for $pe \leftarrow npes - 1$ do
                PUT(distTo[row+1][0], distTo[row+1][0], width, pe)
                PUT(edgeTo[row+1][0], edgeTo[row+1][0], width, pe)
            end for
        end if
        BARRIER
    end for
    if RANK = MASTER then
        return backtrack($dist$,height, width)
    end if
end function

function RELAX($row$,col,$dist$,edge,width)
    $nextrow \leftarrow row + 1$
    for $i \leftarrow -1, 0$ do
        $nextcol \leftarrow col + 1$
        if $nextcol < 0$ or $nextcol \geq width$ then
            continue
        end if
        $tmp \leftarrow dist[row][col] + energy[nextrow][nextcol]$
        if dist[nextrow][nextcol]$>$ $tmp$ then
            $dist[nextrow][nextcol] \leftarrow tmp$
            $edge[nextrow][nextcol] \leftarrow i$
            PUT($distTo[row+1][0]$, $distTo[row+1][0]$, 1, master)
            PUT($edgeTo[row+1][0]$, $edgeTo[row+1][0]$, 1, master)
        end if
    end for
end function
```

Energy Computation: We used a 1D array to keep energy values of pixels and it requires some extra computation to be able to map this values to the 2D array buffer that has the RGB values of the pixels. There is no RMA operations involved in energy computation since all PEs have access to the same buffer values. The algorithm for energy computation with MPI is shown in algorithm 8.

Algorithm 8 MPI Parallel Seam Carving Energy Computation

```
function GENERATEENERGYMATRIX($height, startcol, stopcol, buffer$, orientation)
    for $row \leftarrow 1, height - 1$ do
        for $column \leftarrow start_col, stop_col - 1$ do
            $cell = row * width + column$
            if $orientation = vertical$ then
                $energy1DArray[cell] = computeEnergy(column, row, buffer$
            else
                $energy1DArray[cell] = computeEnergy(row, column, buffer$
            end if
            PUT($energyArray[row][column]$, $energyArray[row][column]$, 1, master)
        end for
    end for
end function
```

Seams Identification: Seams Identification is the most intensive phase of the Seam carving with one-sided MPI. We used one-sided RMA operations and made some calls to barrier. As we mentioned for OpenSHEM, the seams identification phase with one-sided MPI is not parallel friendly and it is even more challenging than that of OpenSHMEM. With OpenSHMEM, we were able to directly put data on a target PE but with one-sided MPI, we had to go through a memory window. Also, since we only could use a 1D array with one-sided MPI, we had to find what cell corresponds to a given row and column. Within the relax function, we used the PUT operation to put any new values of distance and edge into PE 0. Each call to PUT is preceded by an MPI_Win_lock_all function and it is succeeded by MPI_Win_flush_all and MPI_Win_unlock_all functions. All these functions take a window as an argument. The algorithm for the seams identification phase of Seam carving with MPI is shown in Algorithm 9. After relax, all the PEs get to a barrier and PE 0 broadcasts its new distance and edges values to all other PEs. We chose to use broadcast to make all the new values available instead of PUT operations as we did with OpenSHMEM because broadcast is more straight forward than MPI one-sided PUT operation.

Algorithm 9 MPI Parallel Seam Carving Seams Identification

```
 1: function IDENTIFYSEAMS($width$, $height$)
 2:     for $row \leftarrow 0, height$ do
 3:         for $col \leftarrow 0, width$ do
 4:             relax(row, col, dist, edge, width)
 5:         end for
 6:         BARRIER
 7:         $nextCell = (row + 1) * width$
 8:         $BCAST(dist1D[nextcell], width, MPI\_INT, 0, MPI\_COMM\_WORLD)$
 9:         $BCAST(edge1D[nextcell], width, MPI\_INT, 0, MPI\_COMM\_WORLD)$
10:         BARRIER
11:     end for
12:     if RANK = MASTER then
13:         return backtrack($dist$,height, width)
14:     end if
15: end function

16: function RELAX($row$,col,$dist$,edge,width)
17:     $nextrow \leftarrow row + 1$
18:     $cell \leftarrow row * width + col$
19:     for $i \leftarrow -1, 0$ do
20:         $nextcol \leftarrow col + 1$
21:         $nextcell \leftarrow row * width + nextcol$
22:         if $nextcol < 0$ or $nextcol \geq width$ then
23:             continue
24:         end if
25:         $tmp \leftarrow dist1D[cell] + energyID[nextcell]$
26:         if dist1D[nextcell]$>$ $tmp$ then
27:             $dist1D[nextcell] \leftarrow tmp$
28:             $MPI\_Win\_lock\_all(0, win1)$
29:             if pe$>$ 0 then
30:                 PUT(dist1D[nextcell],1,MPI_INT,0, nextcell, 1, MPI_INT, win1)
31:                 $MPI\_Win\_flush\_all(win1)$
32:                 $MPI\_Win\_unlock\_all(win1)$
33:                 $MPI\_Win\_lock\_all(0, win2)$
34:                 if pe$>$ 0 then
35:                     PUT(edge1D[nextcell],1,MPI_INT,0, nextcell, 1, MPI_INT, win2)
36:                     $MPI\_Win\_flush\_all(win2)$
37:                     $MPI\_Win\_unlock\_all(win2)$
38:                 end if
39:             end if
40:         end if
41:     end for
42: end function
```

CHAPTER V.
EXPERIMENTAL EVALUATION

This Chapter takes an in-depth look at the execution times of the different phases of Seam carving in single-threaded, multi-threaded and distributed implementations. We also did a relative performance analysis between multi-threaded and distributed versions of Seam carving with increasing number of pthreads and PEs, respectively. In order to evaluate the performance of Seam carving with OpenSHMEM, MPI and Pthreads, we used 5 images with varying dimensions from a small size to a large size. The image sizes are 912x513(1MB), 3984x1538(10MB), 5891x2271(20MB), 7345x2832(30MB) and 10453x7466(95MB). The testbed for this analysis was the Chameleon Cloud system that is physically distributed between the Texas Advanced Computing Center (TACC) and the University of Chicago (UC). Chameleon Cloud has over of 250 compute nodes. The node we used for our implementation has a Dual Intel Xeon E5-2670, 128 GB of DRAM, and a Mellanox Connect-X 3 FDR InfiniBand NIC.

In the following sections, we evaluate the results from the single-threaded, multi-threaded, MPI and OpenSHMEM versions of Seam carving. We evaluate what percentage of the actual Seam carving time is energy computation, seams identification and seams removal. This will give us an idea of which phase of Seam carving is computationally intensive in all cases. Also we analyse the relative performance of Seam carving as the number of threads and PEs increase with the multi-threaded and distributed versions, respectively. This gives us an idea of any bottleneck that may exist with increasing threads and PEs.

Single-Threaded Evaluation

Figure 7 shows that regardless of the image size, the energy computation phase of Seam carving takes about 68% of the total Seam carving execution time, seams identification takes about 21% and seams removal takes about 11% of the total Seam carving execution time. Our time analysis is specifically tailored towards Seam carving execution time, which is the time it takes from energy computation to seams removal. Our evaluation does not include image pre-processing timings including converting an image to RGB buffer and printing the identified seam on the image.

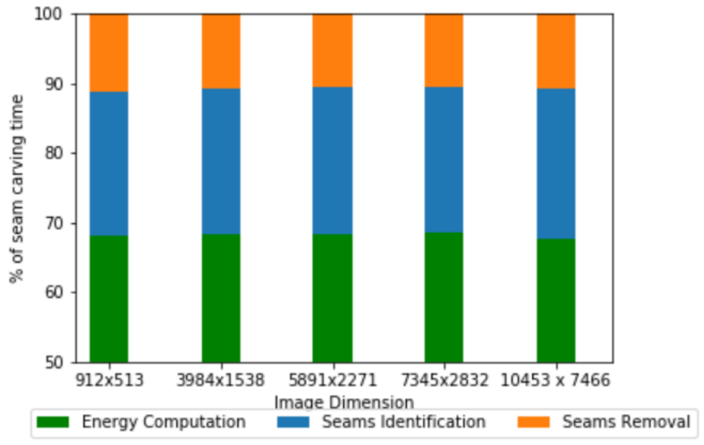


Figure 7: Result Analysis of Seam Carving Phases: Lower values mean faster execution times

Figure 8 shows that as the image size increases, the execution time also increases. This is valid since larger images require longer time to process due to more image pixels involved.

Pthreads Evaluation

We implemented Seam carving using multi-threading with Pthreads in order to know how the Seam carving algorithm performs when parallelized with a single processing element. In this Section, we analyse the performance gained by running Seam carving in parallel using Pthreads.

Figure 9 shows that just like the single-threaded version of Seam carving, energy computation time takes the most time followed by seams identification and seams removal takes the least time. Computing the energy value of a pixel involves considering the RGB values for the neighbouring pixels and finding the gradient value of those neighbouring pixels with the pixel in consideration. Since we need to do this for all image pixels, this takes time and therefore explains why the energy computation phase takes the most time.

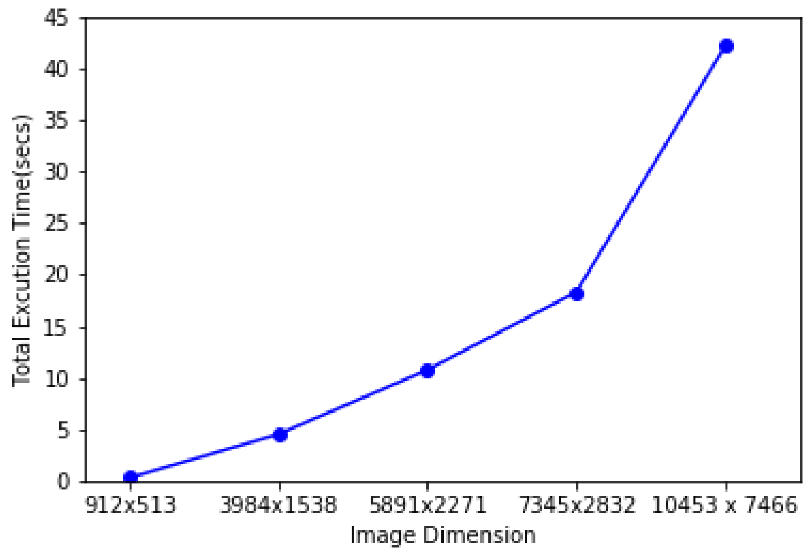


Figure 8: Serial Implementation Execution Time Analysis: The lower the values, the faster the execution time

Figure 10 shows that as the number of threads increase, the total execution time for image resizing decreases, this is expected as more threads are being spawned to work. However, a point is reached when the amount of work is not significant enough for the number of threads and at this point, the execution times becomes slightly stable and no longer decreases with increasing threads. With all our test images, that point is observed to be when the number of threads approach 8 and 16 for smaller images and larger images respectively . The relative performance attained with varying number of threads is shown in Table 1. There is an improved performance as image size increases with increasing threads. It will be observed that with the 912x513 image, there is a slow down as the number of threads approach 8, the slow down becomes rapid with 16 and 32 threads and execution times tend to take longer when compared to the single-threaded version. This is because there is not enough work for this thread as we explained before. For the 10453 x 7466 image, a gradual slow down is noticed with 16 or more threads.

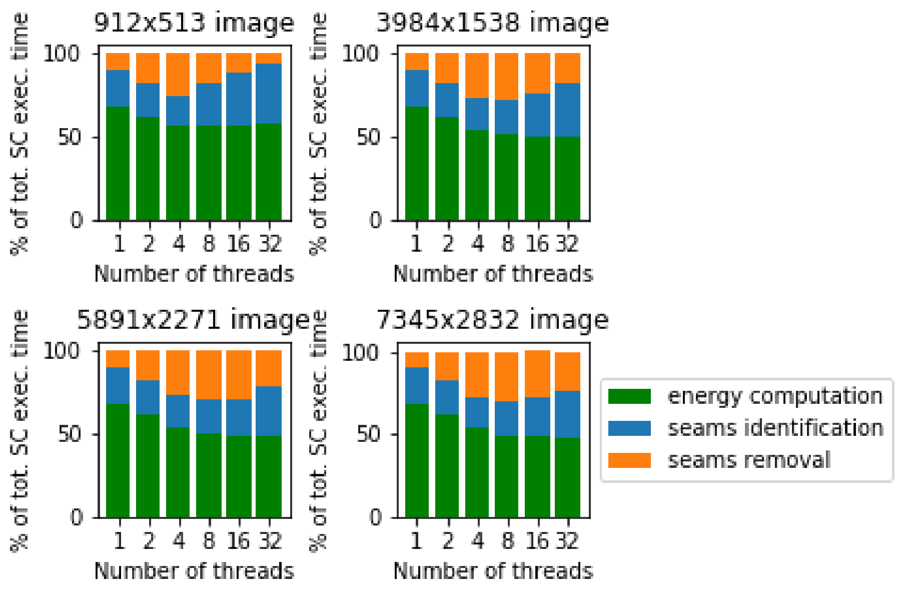


Figure 9: Pthreads Percentage Time Analysis of Seam Carving Phases

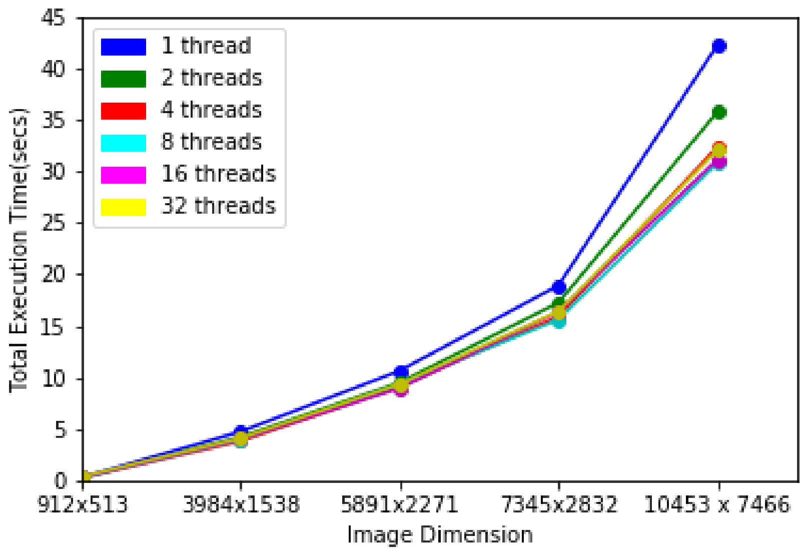


Figure 10: Pthreads Execution Time Analysis: The lower the values, the faster the execution times

Distributed Evaluation and Results

In this Section, we analyse the performance gained by running Seam carving in parallel using OpenSHMEM and MPI.

Table 1: Relative Performance Analysis of Pthreads vs Single-threaded implementation: Positive values show increased performance, negative values means decreased performance

Image Size	2threads(%)	4threads(%)	8threads(%)	16threads(%)	32threads(%)
912x513	10.95	14.92	9.16	-2.60	-24.47
3984x1538	10.96	19.50	16.34	15.04	13.55
5891x2271	10.70	14.54	12.52	15.86	12.10
7345x2832	8.84	15.91	17.45	13.47	13.30
10453 x 7466	15.36	23.36	27.06	26.25	24.13

OpenSHMEM Evaluation

Figure 11 shows that energy computation decreases as the number of PEs increase. This is because having more PEs help to reduce the work load on all the PEs and since energy computation does not involve any RMA operation or any operation like barrier that can slow down the PEs, energy computation increases with increasing PEs. It will also be observed that Seams identification time gradually decreases with more PEs. Although, there are RMA operations being done in the relax functions to put new distances value and edges values in the master PEs, these operations does not cause a significant impact. Therefore increasing the number of PEs helps to ease the work and reduce the workload of all the PEs. Seams removal takes more time as the number of PEs increase because only the master PE is involved in seams removal and therefore does not benefit from workload reduction that comes with increased PEs.

Figure 12 shows that as the number of PEs increases, the total execution time for image resizing decreases, this is expected as more PEs mean reduced workload. Similar to Pthreads result, there is an increase in execution times as 16 or more PEs are used. The relative performance attained with a varying number of PEs is shown in Table 2. There is noticeable performance as the image size increases with 2 or more PEs for larger images. Performance starts to drop with 8 or more threads. A rapid slow down is noticed with 32 PEs.

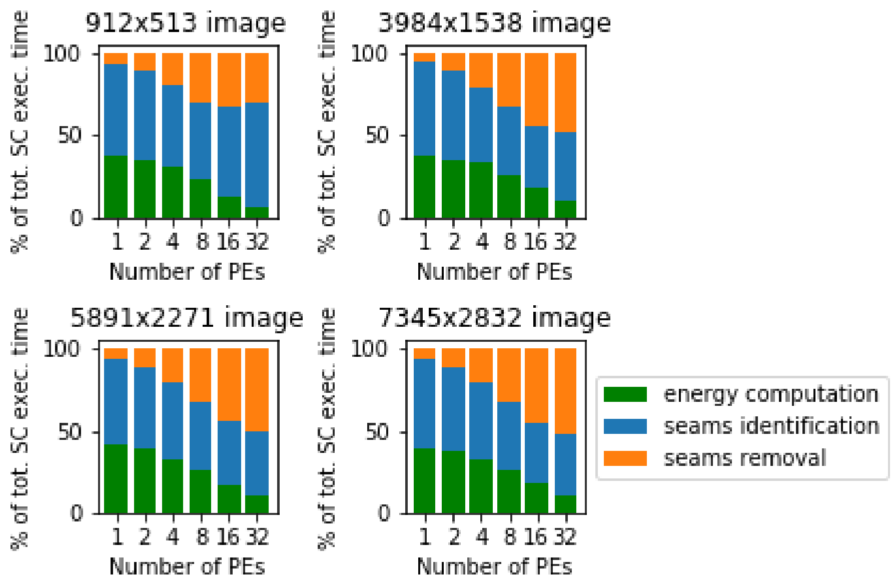


Figure 11: OpenSHMEM Percentage Time Analysis of Seam Carving Phases

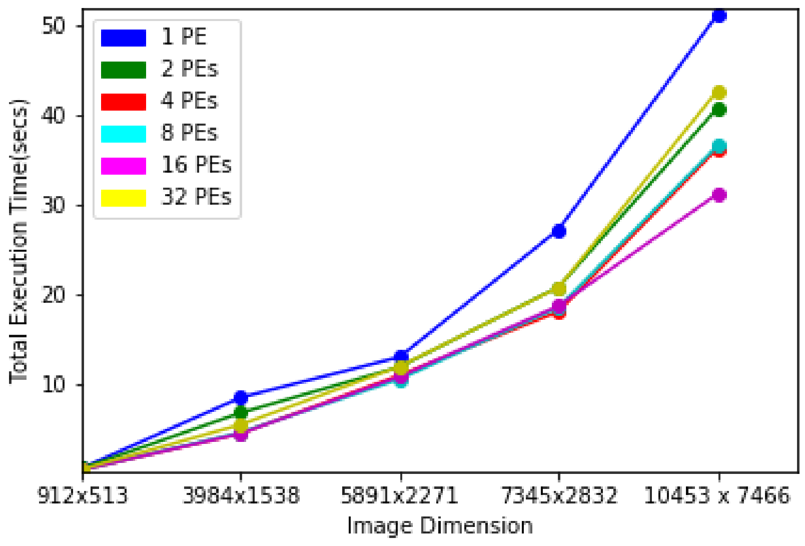


Figure 12: OpenSHMEM Execution Time Analysis: The lower the values, the faster the execution time.

MPI Evaluation

Figure 13 shows that the seams identification phase takes the most time in the Seam carving process. The relax function uses PUT operations to transfer new distances and

Table 2: Relative Performance Analysis of Single-PE vs Multiple-PEs OpenSHMEM implementation: The lower the value, the faster the speedup

Image Size	2PEs(%)	4PEs(%)	8PEs(%)	16PEs(%)	32PEs(%)
912x513	13.41	22.57	20.41	14.42	0.51
3984x1538	15.60	21.31	18.66	17.68	2.54
5891x2271	15.79	20.57	18.35	17.16	3.72
7345x2832	14.39	18.03	15.16	11.98	1.05
10453x7466	20.40	29.19	28.60	29.13	16.60

edges to PE 0. These RMA operations require a lock on the window that is managing the data structure. For other PEs to access that window, they have to wait for the window to be unlocked. After the relax function, PE 0 does a broadcast of the new values to all the other PEs. It will also be noticed that seams identification time increases as PEs increase. This is because there are more RMA calls in the relax function with more PEs.

Figure 14 shows that as the number of PEs increases, the total execution time for image resizing increases, this is a big shift from the trend we saw with Pthreads and OpenSHMEM evaluation. This shift is because of the extra work involved in running Seam carving with one-sided MPI. Some of this involve the use of a 1D array to manage the 2D energy, distances and edges data structures that Seam carving uses. We created a 1D array and did some calculations to find the 1D equivalent of a 2D data. There is no performance attained as the number of PEs increase.

Performance Evaluation between Pthreads and Distributed Seam Carving

Figure 15 shows the relative performance of energy computation time with increased threads and PEs. With OpenSHMEM, as the number of PE increases, energy computation time decreases. For smaller number of PEs and Pthreads ranging from 1 to 4, the Pthreads and MPI energy computation times are so close that they overlap on the graph while OpenSHMEM takes a longer execution time . For increased number of PEs ranging from 8-32, OpenSHMEM and MPI energy computation times are so close that they overlap on

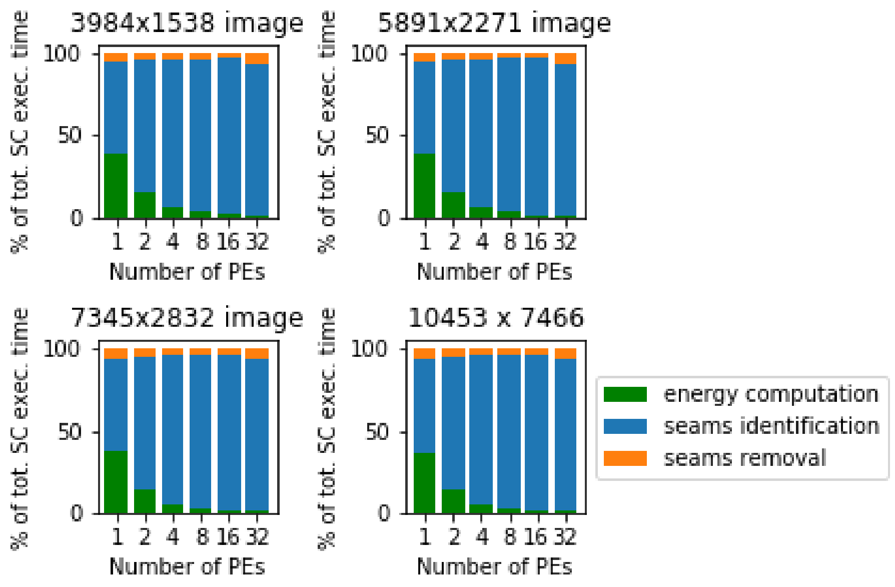


Figure 13: MPI Percentage Time Analysis of Seam Carving Phases

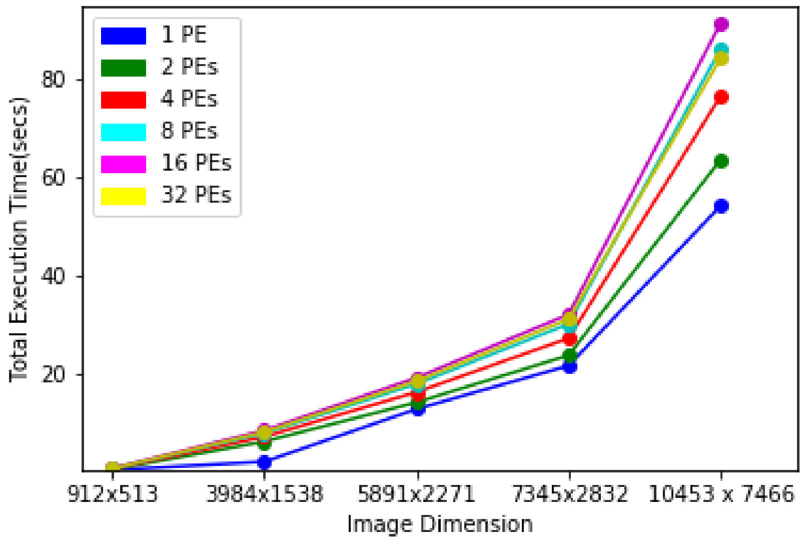


Figure 14: MPI Execution Time Analysis: The lower the values, the faster the execution time

the graph while the Pthreads version takes more time.

Figure 16 shows the relative performance of seams identification time with an increased number threads and PEs. MPI takes the most seams identification time for more than 1 PE. As the number of threads and PEs increase from 4 to 32, the seams identification times are so close between the Pthreads and OpenSHMEM versions that an overlap is noticed between them.

Fig 17 shows a comparison between the Pthreads and the distributed implementation of Seam carving. It was observed that Pthreads implementation have the best performance and it is noticeable with larger images. The performance of OpenSHMEM is close to Pthreads and MPI has the least performance. This is connected to the graph in Figure 9. Since seams identification is the most time consuming phase of the Seam carving process for MPI and we have established that with increased PEs, the seams identification time takes longer. This explains why the execution time for seam carving with one-sided MPI takes more time.

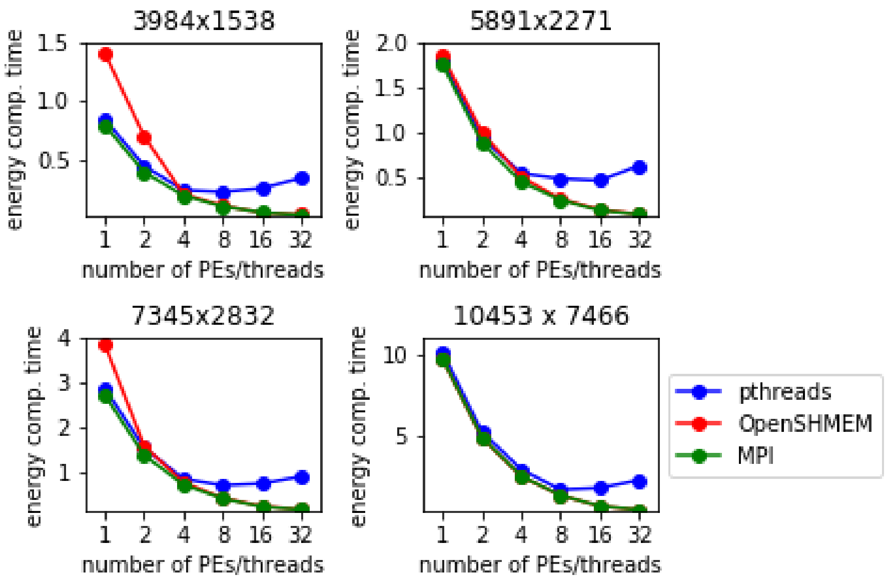


Figure 15: Pthreads vs Distributed Energy Computation Relative Performance: Lower values mean faster execution times

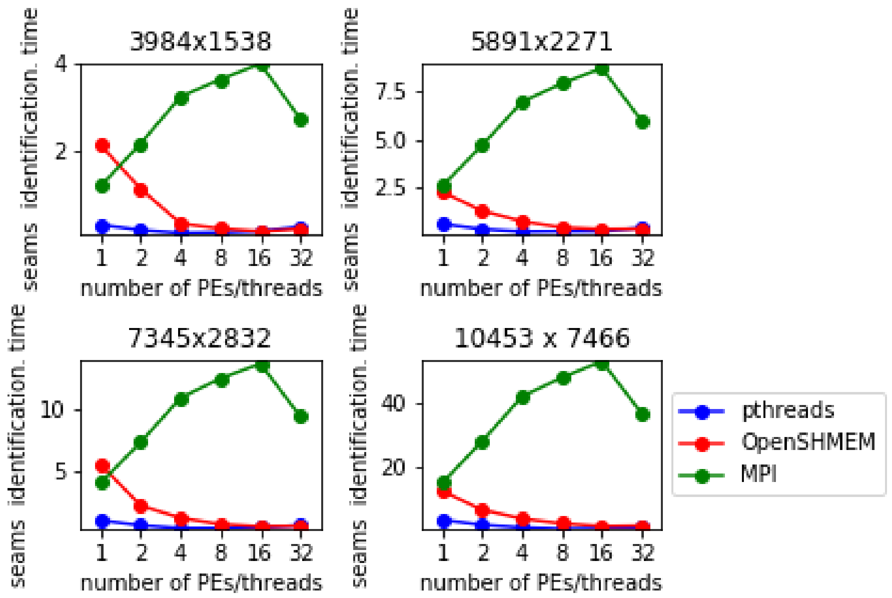


Figure 16: Pthreads vs Distributed Seams Identification Relative Performance: The lower the value, the faster the execution time

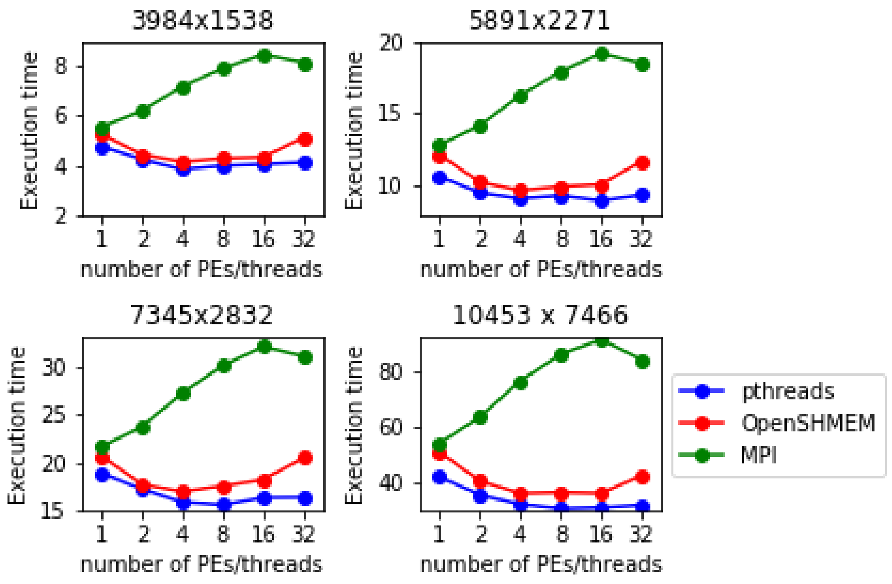


Figure 17: Pthreads vs Distributed Execution Time: Lower values mean faster execution times

CHAPTER VI.
CONCLUSION

In this thesis, we have evaluated and discussed the adaptation of Seam carving algorithm to the OpenSHMEM programming model, one-sided MPI and Pthreads. Our results show that for Pthreads and OpenSHMEM, we can obtain increased performance when running Seam carving in parallel with less than 16 threads or PEs for the image sizes we used in this thesis. For larger images, having more than 16 threads or PEs should also improve performance. If we have a big enough image that will give all threads and PEs enough work to do, performance will be improved. When there is no significant amount of work for all available PEs and threads, communication overheads take place and a slow down is noticed.

Although Pthreads performed better in terms of execution times, we have been able to show that we can run Seam carving with multiple PEs. An application of this parallel implementation of Seam carving with OpenSHMEM that comes to mind is a cloud based image streaming. For example, we can spread an image pixels across a number of PEs and resize the image. Also, we have been able to compare the performance of OpenSHMEM with one-sided MPI. Our results shows that OpenSHMEM's one-sided RMA operations adapt so well with the complexity involved with identifying seams in parallel. MPI one-sided RMA operations slowed down the performance as more PEs were introduced.

Future Work

The Seam carving algorithm we implemented only works with images. Seam carving can also be applied to video resizing. The approach with video resizing is different from that of image resizing. For image resizing, we remove a 1D seam from a 2D image buffer. However, with video resizing, a video is treated as a 3D cube and the seams identification phase involves removing a 2D connected seam from a 3D volume. It also involves replacing the dynamic programming approach of image resizing with Graph cuts.

As we extend this research, we will consider video re-targeting with Seam carving using all of the implementation approaches in this thesis. We will also evaluate the relative performance analysis of these implementations. This will give us an idea which of our implementations tend to improve or worsen with video re-sizing.

While our implementation covers both horizontal and vertical seams identification, its limitation is that it can not identify both horizontal and vertical seam together. Our image resizing technique is one-directional. In real life, image resizing can done in both ways; vertically and horizontally. We would like to improve our seam carving implementation to handle bi-directional seams identification. That way, we would be able to remove vertical seams along size horizontal seams.

BIBLIOGRAPHY

[1] https://developer.nvidia.com/nvshmem.

[2] Nvshmem early access. https://developer.nvidia.com/nvshmem.

[3] OpenSHMEM. https://www.csm.ornl.gov/openshmem/index.html.

[4] OpenSHMEM Application Programming Interface. `http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf`. Accessed: 2017-12-14.

[5] OpenSHMEM Application Programming Interface - Version 1.0. http://www.openshmem.org/site/sites/default/site_files/openshmem_specification-1.0.pdf.

[6] OpenSHMEM Application Programming Interface - Version 1.1. http://www.openshmem.org/site/sites/default/site_files/openshmem-specification-1.1.pdf.

[7] OpenSHMEM Application Programming Interface - Version 1.2. http://www.openshmem.org/site/sites/default/site_files/openshmem-specification-1.2.pdf.

[8] OpenSHMEM Application Programming Interface - Version 1.3. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf.

[9] OpenSHMEM Application Programming Interface - Version 1.4. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf.

[10] OpenSHMEM Application Programming Interface - Version 1.5. http://www.openshmem.org/site/sites/default/site_files/openshmem-1.5rc1.pdf.

[11] Ali, S. and Hayat, L. An efficient memory model for implementing image resizing algorithms in a distributed environment. In *2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation*, pages 381–385, March 2014.

[12] Almasi, G. *PGAS (Partitioned Global Address Space) Languages*, pages 1539–1545. Springer US, Boston, MA, 2011.

[13] Anh, N., Yang, W., and Cai, J. Seam carving extension: a compression perspective. pages 825–828, 01 2009.

[14] Avidan, S. and Shamir, A. Seam carving for content-aware image resizing. *ACM Trans. Graph.*, 26(3):10, 2007.

[15] Bachan, J., Baden, S. B., Bonachea, D., Hargrove, P. H., Hofmeyr, S. A., Ibrahim, K. Z., Jacquelin, M., Kamil, A., Lelbach, B. A., and van Straalen, B. Upc++ specification v1.0, draft 4. 2017.

[16] Bohari, M. U. A. and Kamalapur, P. S. M. op3D Image Retargeting. 2014.

[17] Brown, R. Photivo. https://www.photivo.com/.

[18] ByoungChul, K., SeongHoon, K., and JaeYeal, N. Image resizing using saliency strength map and seam carving for white blood cell analysis. *BioMedical Engineering OnLine*, 9(54), September 2010.

[19] Cešnovar, R., Bulić, P., and Dobravec, T. Optimization of a single seam removal using a gpu. 07 2011.

[20] Chamberlain, B., Callahan, D., and Zima, H. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, 08 2007.

[21] Chiang, C., Wang, S., Chen, Y., and Lai, S. Fast jnd-based video carving with gpu acceleration for real-time video retargeting. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(11):1588–1597, Nov 2009.

[22] community, T. I. Liquid rescale library. https://imagemagick.org/.

[23] D'Azevedo, E. F. and Imam, N. Graph 500 in openshmem. In Gorentla Venkata, M., Shamis, P., Imam, N., and Lopez, M. G., editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies*, pages 154–163, Cham, 2015. Springer International Publishing.

[24] De Wael, M., Marr, S., De Fraine, B., Van Cutsem, T., and De Meuter, W. Partitioned global address space languages. *ACM Comput. Surv.*, 47(4):62:1–62:27, May 2015.

[25] DeCarlo, D. and Santella, A. Stylization and Abstraction of Photographs. 21(3):769–776, July 2002.

[26] Division, N. A. S. Nas parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html.

[27] Duarte, R. and Sendag, R. Accelerating and Characterizing Seam Carving Using a Heterogeneous CPU-GPU system. *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, 2012.

[28] Enkh-Amgalan, B. Implementing seam carving (liquid rescaling) in OpenCL . http://bandi.me/seam-carving/.

[29] Grossman, M., Pritchard, H., Budimlić, Z., and Sarkar, V. Graph500 on openshmem: Using a practical survey of past work to motivate novel algorithmic developments. In *Proceedings of the Second Annual PGAS Applications Workshop*, PAW17, pages 2:1–2:8, New York, NY, USA, 2017. ACM.

[30] Guo, Y., Liu, F., Shi, J., Zhou, Z., and Gleicher, M. Image retargeting using mesh parametrization. *IEEE Transactions on Multimedia*, 11(5):856–867, Aug 2009.

[31] Hug, J., Ginsburg, M., and Wayne, K. Programming Assignment 2: Seam Carving. http://coursera.cs.princeton.edu/algs4/assignments/seam.html.

[32] Jose, J., Potluri, S., Subramoni, H., Lu, X., Hamidouche, K., Schulz, K., Sundar, H., and Panda, D. K. Designing scalable out-of-core sorting with hybrid mpi+pgas programming models. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 7:1–7:9, New York, NY, USA, 2014. ACM.

[33] Jose, J., Potluri, S., Tomko, K., and Panda, D. K. Designing scalable graph500 benchmark with hybrid mpi+openshmem programming models. In Kunkel, J. M., Ludwig, T., and Meuer, H. W., editors, *Supercomputing*, pages 109–124, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[34] Kiess, J., Gritzner, D., Guthier, B., Kopf, S., and Effelsberg, W. Gpu video retargeting with parallelized seamcrop. pages 139–147, 03 2014.

[35] Kim, I., Zhai, J., Li, Y., and Chen, W. Optimizing seam carving on multi-gpu systems for real-time content-aware image resizing. *J. Supercomput.*, 71(9):3500–3524, September 2015.

[36] Lu, M. and Niu, D. S. Detection of image seam carving using a novel pattern. *Computational Intelligence and Neuroscience*, 4(12), November 2019.

[37] Moorthu, M. and Amutha, D. R. An improved medical image coding technique based on seam identification and spiht. *International Journal of Scientific an Engineering Research*, 4(12), December 2013.

[38] Namashivayam, N., Cernohous, B., Kandalla, K., Pou, D., Robichaux, J., Dinan, J., and Pagel, M. Symmetric memory partitions in openshmem: A case study with intel knl. In Gorentla Venkata, M., Imam, N., and Pophale, S., editors, *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*, pages 3–18, Cham, 2018. Springer International Publishing.

[39] Numrich, R. W. and Reid, J. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[40] Pophale, S., Nanjegowda, R., Curtis, A. R., Chapman, B., Jin, H., Poole, S. W., and Kuehn, J. A. Openshmem performance and potential: A npb experimental study. 1 2012.

[41] Potluri, S., Goswami, A., Rossetti, D., Newburn, C. J., Venkata, M. G., and Imam, N. Gpu-centric communication on nvidia gpu clusters with infiniband: A case study with openshmem. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 253–262, Dec 2017.

[42] Potluri, S., Goswami, A., Venkata, M. G., and Imam, N. Efficient breadth first search on multi-gpu systems using gpu-centric openshmem. In Gorentla Venkata, M., Imam, N., and Pophale, S., editors, *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*, pages 82–96, Cham, 2018. Springer International Publishing.

[43] Salma, E. and Kumar, J. P. J. Efficient image compression based on seam carving for arbitrary resolution display devices. In *2013 International Conference on Communication and Signal Processing*, pages 529–532, April 2013.

[44] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O., Grove, D., Shinnar, A., Takeuchi, M., Tardieu, O., Agarwal, P. M. I. S., Alpern, B., Bacon, D., Barik, R., Blainey, B., Bloom, B., Cheng, P., Dolby, J., Fink, S., Fuhrer, R., Gallop, P., Grothoff, C., Horii, H.,

Kawachiya, K., Kielstra, A., Ko, S., Peshansky, I., Sarkar, V., Solar-lezama, O., Alex, S., Spoon, E., Sur, S., Suzumura, T., Praun, C. V., Unnikrish, L., Varma, P., N, K., Venkata, I., Vitek, J., Wang, H. C., Zakirov, S., Zibin, Y., Shyamasundar, R. K., Rajan, V. T., Tip, F., Vaziri, A., and Xue, H. X10 language specification version 2.5, 2014.

[45] Stultz, J. Seam Carving: Parallelizing a novel new image resizing algorithm . http://courses.csail.mit.edu/18.337/2008/projects/reports/stultz-6338.pdf.

[46] Tanaka, Y., Hasegawa, M., and Kato, S. Improved image concentration for artifact-free image dilution and its application to image coding. In *2010 IEEE International Conference on Image Processing*, pages 1225–1228, Sep. 2010.

[47] Team, T. D. Digikam: Professional photo management with the power of open source, 2020 (Last Release). https://www.digikam.org/.

[48] Venkata, M. G., Imam, N., and Pophale, S. *OpenSHMEM and Related Technologies*. Springer, 2018.

[49] Wang, Y.-S., Tai, C.-L., Sorkine, O., and Lee, T.-Y. Optimized Scale-and-Stretch for Image Resizing. *ACM Trans. Graph*, 27(5):1181–1188, 2008.

[50] Yamasaki, T., Matsunami, T., and Aizawa, K. Detecting resized jpeg images by analyzing high frequency elements in dct coefficients. pages 567 – 570, 11 2010.

[51] Yelick, K., Graham, S. L., Hilfinger, P., Bonachea, D., Su, J., Kamil, A., Datta, K., Colella, P., and Wen, T. *Titanium*, pages 2049–2055. Springer US, Boston, MA, 2011.

Serial Implementation of Seam Carving

```
#include <pngwriter.h>

#include <lqr.h>

#include <getopt.h>

#include "liquidrescale.h"

#include <math.h>

#include <limits>

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h>

#include <string.h>

#include <unistd.h>

#include <sys/time.h>

#include <ios>

#include <fstream>




using namespace std;

pngwriter pngwrt(1,1,0,"out_serial.png");

int BASE_ENERGY = 1000;

int ROWS_PER_THREAD = 64;

int width = 0;

int height = 0;

gfloat rigidity = 0;

gint max_step = 1;

int channels = 3;
```

```
int** energyArray;
guchar* seams;
guchar* buffer;
int* verticalSeams;
int** distTo;
int** edgeTo;



/*Copied from the liblqr example
 convert the image in the right format */
guchar * rgb_buffer_from_image(pngwriter *png)
{
    gint x, y, k, channels;
    gint w, h;
    guchar *buffer;

    /* get info from the image */
    w = png->getwidth();
    h = png->getheight();
    channels = 3;                   // we assume an RGB image here

    /* allocate memory to store w * h * channels unsigned chars */
    buffer = g_try_new(guchar, channels * w * h);
    g_assert(buffer != NULL);

    /* start iteration (always y first, then x, then colours) */
    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            for (k = 0; k < channels; k++) {
               /* read the image channel k at position x,y */
               buffer[(y * w + x) * channels + k] = (guchar) (png->dread(x
                  + 1, y + 1, k + 1) * 255);
               /* note : the x+1,y+1,k+1 on the right side are
                *       specific the pngwriter library */
            }
         }
      }

      return buffer;
   }



   int computeEnergy(int x, int y, guchar* buffer){
      if (x == 0 || y == 0 || (x == width - 1) || (y == height- 1))
            return BASE_ENERGY;

     //Declare values for the RGB top, bottom, left and right neighbours
     double top_RB, top_GB, top_BB = 0.0;
     double bottom_RB, bottom_GB, bottom_BB = 0.0;
     double left_RB, left_GB, left_BB = 0.0;
     double right_RB, right_GB, right_BB = 0.0;

     /*Declare values for the differences of the vertical and horizontal
       RGB neighbours difference*/
     double redDiffH, greenDiffH, blueDiffH = 0.0;
     double redDiffV, greenDiffV, blueDiffV = 0.0;
```

```
/*Declare values for the sum of the vertical and horizontal RGB
   neighbours differences*/
double valueH, valueV, valueSum;


//Get the RGB values of the top neighbor of the current pixel

top_RB = buffer[((y-1)*width+(x))*3+0];
top_GB = buffer[((y-1)*width+(x))*3+1];
top_BB = buffer[((y-1)*width+(x))*3+2];

//Get the RGB values of the bottom neighbor of the current pixel
bottom_RB = buffer[((y+1)*width+(x))*3+0];
bottom_GB = buffer[((y+1)*width+(x))*3+1];
bottom_BB = buffer[((y+1)*width+(x))*3+2];

//Get the RGB values of the right neigbor of the current pixel
right_RB = buffer[((y)*width+(x+1))*3+0];
right_GB = buffer[((y)*width+(x+1))*3+1];
right_BB = buffer[((y)*width+(x+1))*3+2];

//Get the RGB values of the left neigbor of the current pixel
left_RB = buffer[((y)*width+(x-1))*3+0];
left_GB = buffer[((y)*width+(x-1))*3+1];
left_BB = buffer[((y)*width+(x-1))*3+2];
```

```
    //Get the absolute difference of the red horizontal neighbours
    redDiffV = abs(top_RB - bottom_RB);

    //Get the absolute difference of the red horizontal neighbours
    redDiffH = abs(right_RB - left_RB);

    //Get the absolute difference of the green horizontal neighbours
    greenDiffV = abs(top_GB - bottom_GB);

    //Get the absolute difference of the green horizontal neighbours
    greenDiffH = abs(right_GB - left_GB);

    //Get the absolute difference of the blue horizontal neighbours
    blueDiffV = abs(top_BB - bottom_BB);

    //Get the absolute difference of the blue horizontal neighbours
    blueDiffH = abs(right_BB - left_BB);

    //Get ths sum of the square of differences of vertical neighbours
    valueH = pow(redDiffH,2) + pow(greenDiffH,2) + pow(blueDiffH,2);

    //Get ths sum of the square of differences of horizontal neighbours
    valueV = pow(redDiffV,2) + pow(greenDiffV,2) + pow(blueDiffV,2);
    valueSum = valueV + valueH;

    //Return the squareroot of the sum of differences
    return round(sqrt(valueSum));
}
```

```
void generateEnergyMatrix(int width, int height, char* orientation){
   //Declare a dynamic 2D array to hold the energy values for all pixels
   for (int row = 1; row < height; row++) {
        for(int column= 0; column < width; column++) {
            if (orientation[0] == 'v')
                energyArray[row][column] = computeEnergy(column, row,
                    buffer);
            else
                energyArray[row][column] = computeEnergy(row, column,
                    buffer);
        }
    }

}




/*Declare a relax function to optimize the computation of a
   shortest path energy values*/

void relax(int row, int col, int** edgeTo, int** distTo, int width) {
   int relax = 0;
        int nextRow = row + 1;
        for (int i = -1; i <= 1; i++) {
            int nextCol = col + i;
            if (nextCol < 0 || nextCol >= width)
                continue;
            if (distTo[nextRow][nextCol] >= distTo[row][col] +
                energyArray[nextRow][nextCol]) {
                distTo[nextRow][nextCol] = distTo[row][col] +
                    energyArray[nextRow][nextCol];
                edgeTo[nextRow][nextCol] = i;

            }
        }
    }

int* backTrack(int** edgeTo, int** distTo, int height, int width){
// Backtrack from the last row to get a shortest path
   int* seams = new int[height];
        int minCol = 0;
        int minDist = std::numeric_limits<int>::max();
        for (int col = 0; col < width; col++) {
            if (distTo[height - 1][col] < minDist) {
                minDist = distTo[height - 1][col];
                minCol = col;
            }
        }
    for (int row = height - 1; row >= 0; row--) {
            verticalSeams[row] = minCol;
            minCol = minCol - edgeTo[row][minCol];
        }
   return verticalSeams;

}
```

```
//A Transpose Matrix that makes findVerticalSeams re-usable
 guchar* transposeRGBuffer(guchar* buffer, int width, int height) {
   guchar* transposedRGBuffer;
   int size = 3 * width * height;
      transposedRGBuffer = g_try_new(guchar,size);
      g_assert(transposedRGBuffer != NULL);
   for(int col = 0; col < width; col++){
      for(int row = 0; row < height; row++){
         for (int color = 0; color < 3; color++){
            transposedRGBuffer[(col*height + row)*3 + color] =
                buffer[(row*width+col)*3+color];
         }
      }
   }
      return transposedRGBuffer;
}

//TO BE PARALLELIZED
//Find the indexes of the vertical seams to be carved out
int * identifySeams( int width, int height){

   for (int i = 0; i < height; i++)
      distTo[i] = new int[width];

   //Declare an array to hold the paths taken to reach a pixel

   for (int i = 0; i < height; i++)
```

```
        edgeTo[i] = new int[width];



    //Initialize distTo to maximum values

         for (int row = 0; row < height; row++) {
             for (int col = 0; col < width; col++) {
                 if (row == 0)
                     distTo[row][col] = BASE_ENERGY;
                 else
                     distTo[row][col] = std::numeric_limits<int>::max();
             }
        }
     for (int row = 0; row < height - 1; row++) {
             for (int col = 0; col < width; col++) {
                 relax(row, col, edgeTo, distTo, width);
             }
         }
}



//Carve out the vertical seams
guchar* carveVertically(int* vertical_seams, guchar* buffer, int width,
     int height){
    guchar* carved_imageV;
    int size = 3 * width * height;
```

```
/* allocate memory to store w * h * channels unsigned chars */
   carved_imageV = g_try_new(guchar, size);
   g_assert(carved_imageV != NULL);
seams = g_try_new(guchar, size);
   g_assert(seams != NULL);



for (int row = 0; row < height; row++){
   //Get the RGB value before the seam
   for (int col = 0; col < vertical_seams[row]; col++){
      for (int color = 0; color < 3; color++){
         carved_imageV[(row * width + col) * 3 + color] = buffer[(row *
             width + col) * 3 + color];
         seams[(row * width + col) * 3 + color] = buffer[(row * width +
             col) * 3 + color];
      }
   }

   //Get the RGB values after the seams
   for (int col = vertical_seams[row]; col < width-1; col++) {
      for (int color = 0; color < 3; color++){
         carved_imageV[(row * width + col) * 3 + color] = buffer[(row *
             width + col+1) * 3 + color];
         seams[(row * width + col+1) * 3 + color] = buffer[(row * width
             + col+1) * 3 + color];
      }
         }
   }

   return carved_imageV;
}

LqrRetVal printSeams(LqrCarver *carver, pngwriter *pngwrt){

    gint x, y;
    guchar *rgb;
    gdouble red, green, blue;

    lqr_carver_scan_reset(carver);

    /* readout (no need to init rgb) */
    while (lqr_carver_scan(carver, &x, &y, &rgb)) {
        /* convert the output into doubles */
        red = (gdouble) rgb[0] / 255;
        green = (gdouble) rgb[1] / 255;
        blue = (gdouble) rgb[2] / 255;

   if(!rgb[0] && !rgb[1] && !rgb[2])
      pngwrt->plot(x + 1, y + 1, 1.0, 0.0, 0.0);
   else
      pngwrt->plot(x + 1, y + 1, red, green, blue);

}
      return LQR_OK;
}
```

```
LqrRetVal write_carver_to_image(LqrCarver *carver, pngwriter *pngwrt,
    char* orientation){

    gint x, y;
       guchar *rgb;
    gdouble red, green, blue;
    /* resize the image canvas as needed to
     * fit for the new size */



   //Resize based on the orientation
   if(orientation[0] == 'v'){
      TRAP(lqr_carver_resize(carver, width-1 , height));
         pngwrt->resize(width-1, height);
   }
   else{
      TRAP(lqr_carver_resize(carver, width , height-1));
      pngwrt->resize(width,height-1);
      }
    lqr_carver_scan_reset(carver);



    /* readout (no need to init rgb) */
    while (lqr_carver_scan(carver, &x, &y, &rgb)) {
        /* convert the output into doubles */
        red = (gdouble) rgb[0] / 255;
        green = (gdouble) rgb[1] / 255;
        blue = (gdouble) rgb[2] / 255;

        /* plot (pngwriter's coordinates start from 1,1) */
        pngwrt->plot(x + 1, y + 1, red, green, blue);
    }
       return LQR_OK;
}

/* Get current time*/

double timestamp()
{
    struct timeval tval;

    gettimeofday( &tval, ( struct timezone * ) 0 );
    return ( tval.tv_sec + (tval.tv_usec / 1000000.0) );
}

int main(int argc, char **argv){
   double total_begin = timestamp();
   char * original_img = argv[1];
   char* orientation = argv[2];
        char* filename = argv[3];
   pngwrt.readfromfile(original_img);
   width = pngwrt.getwidth();
   height = pngwrt.getheight();

   int size = 3 * width * height;
   buffer = g_try_new(guchar,size);
buffer = rgb_buffer_from_image(&pngwrt);

   g_assert(buffer != NULL);



LqrCarver *carver;
LqrCarver *carved_seams;
double ec_b = 0;
double ec_e = 0;
double si_b = 0;
double si_e = 0;
double sr_b = 0;
double sr_e = 0;
std::ofstream log(filename, std::ios_base::app);
//Check the orientation to determine how to carve
if(orientation[0] == 'v'){

   verticalSeams = new int[height];
   distTo = new int*[height];
   edgeTo = new int*[height];
   //Declare a dynamic 2D array to hold the energy values for all pixels
   energyArray = new int*[height];
   for (int i = 0; i < height; i++)
   energyArray[i] = new int[width];
   ec_b = timestamp();
   generateEnergyMatrix(width, height, orientation);
   ec_e = timestamp();
```

```
        identifySeams(width, height);
        int* v_seams = backTrack(edgeTo, distTo, height, width);
        si_e = timestamp();
        guchar* carved_imageV = carveVertically(v_seams,buffer, width,
            height);
        carver = lqr_carver_new(carved_imageV, width, height, 3);
        carved_seams = lqr_carver_new(seams, width, height, 3);
        sr_e = timestamp();
    }
    else{
        verticalSeams = new int[width];
        distTo = new int*[width];
        edgeTo = new int*[width];
        //Declare a dynamic 2D array to hold the energy values for all pixels
        energyArray = new int*[width];
        for (int i = 0; i < width; i++)
            energyArray[i] = new int[height];
        generateEnergyMatrix(height, width, orientation);

        cout<<"Removing horizontal seams"<<endl;
        identifySeams(height, width);
        int* h_seams = backTrack(edgeTo, distTo, width, height);
        guchar* transBuffer = transposeRGBuffer(buffer, width, height);
        guchar* carved_imageH = carveVertically(h_seams, transBuffer, height,
            width);
        carver = lqr_carver_new(transposeRGBuffer(carved_imageH,
            height,width), width, height, 3);
        carved_seams = lqr_carver_new(transposeRGBuffer(seams, height,width),
            width, height, 3);
    }

    //Create a Carver object with the carved image buffer
       TRAP(lqr_carver_init(carver, max_step, rigidity));
    //write_carver_to_image(carver, &pngwrt, orientation);
    printSeams(carved_seams, &pngwrt);
    lqr_carver_destroy(carver);
    pngwrt.close();
    double total_end = timestamp();
    log<<(ec_e -ec_b)<<","<<(si_e-ec_e)<<","<<(sr_e -
        si_e)<<","<<(total_end - total_begin)<<endl;
    return 0;
}
```

Pthreads Implementation of Seam Carving

```
#include <pngwriter.h>

#include <lqr.h>

#include <getopt.h>

#include "liquidrescale.h"

#include <math.h>

#include <limits>

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h>

#include <string.h>

#include <unistd.h>

#include <sys/time.h>

#include <thread>

#include <ios>

#include <fstream>


using namespace std;

pngwriter pngwrt(1,1,0,"out_pthreads.png");

const int BASE_ENERGY = 1000;

int width;

int height;

gfloat rigidity = 0;

gint max_step = 1;

int channels = 3;

double** energyArray;

guchar* seams;
```

```
guchar* buffer;
int* verticalSeams;
int** distTo;
int** edgeTo;
guchar* carved_imageV;
pthread_barrier_t mybarrier;
struct ThreadData {
   int num_rows;
   int num_cols;
   int start_col;
   int stop_col;
   int thread_id;
   int thread_num;
   char* orientation;
};

/*Copied from the liblqr example
  convert the image in the right format */
guchar * rgb_buffer_from_image(pngwriter *png)
{
   gint x, y, k, channels;
   gint w, h;
   guchar *buffer;

   /* get info from the image */
   w = png->getwidth();
   h = png->getheight();
   channels = 3;                   // we assume an RGB image here
   /* allocate memory to store w * h * channels unsigned chars */
   buffer = g_try_new(guchar, channels * w * h);
   g_assert(buffer != NULL);

   /* start iteration (always y first, then x, then colours) */
   for (y = 0; y < h; y++) {
      for (x = 0; x < w; x++) {
         for (k = 0; k < channels; k++) {
            /* read the image channel k at position x,y */
            buffer[(y * w + x) * channels + k] = (guchar) (png->dread(x +
                1, y + 1, k + 1) * 255);
            /* note : the x+1,y+1,k+1 on the right side are
             *       specific the pngwriter library */
         }
      }
   }

   return buffer;
}

double computeEnergy(int x, int y, guchar* buffer){
   if (x == 0 || y == 0 || (x == width - 1) || (y == height- 1))
      return BASE_ENERGY;

   //Declare values for the RGB top, bottom, left and right neighbours
   double top_RB, top_GB, top_BB = 0.0;
   double bottom_RB, bottom_GB, bottom_BB = 0.0;
   double left_RB, left_GB, left_BB = 0.0;
   double right_RB, right_GB, right_BB = 0.0;

   /*Declare values for the differences of the vertical and horizontal
     RGB neighbours difference*/
   double redDiffH, greenDiffH, blueDiffH = 0.0;
   double redDiffV, greenDiffV, blueDiffV = 0.0;

   /*Declare values for the sum of the vertical and horizontal RGB
     neighbours differences*/
   double valueH, valueV, valueSum;



   //Get the RGB values of the top neighbor of the current pixel

   top_RB = buffer[((y-1)*width+(x))*3+0];
   top_GB = buffer[((y-1)*width+(x))*3+1];
   top_BB = buffer[((y-1)*width+(x))*3+2];

   //Get the RGB values of the bottom neighbor of the current pixel
   bottom_RB = buffer[((y+1)*width+(x))*3+0];
   bottom_GB = buffer[((y+1)*width+(x))*3+1];
   bottom_BB = buffer[((y+1)*width+(x))*3+2];

   //Get the RGB values of the right neigbor of the current pixel
   right_RB = buffer[((y)*width+(x+1))*3+0];
   right_GB = buffer[((y)*width+(x+1))*3+1];
   right_BB = buffer[((y)*width+(x+1))*3+2];
```

```
//Get the RGB values of the left neigbor of the current pixel
left_RB = buffer[((y)*width+(x-1))*3+0];
left_GB = buffer[((y)*width+(x-1))*3+1];
left_BB = buffer[((y)*width+(x-1))*3+2];



//Get the absolute difference of the red horizontal neighbours
redDiffV = abs(top_RB - bottom_RB);

//Get the absolute difference of the red horizontal neighbours
redDiffH = abs(right_RB - left_RB);

//Get the absolute difference of the green horizontal neighbours
greenDiffV = abs(top_GB - bottom_GB);

//Get the absolute difference of the green horizontal neighbours
greenDiffH = abs(right_GB - left_GB);

//Get the absolute difference of the blue horizontal neighbours
blueDiffV = abs(top_BB - bottom_BB);

//Get the absolute difference of the blue horizontal neighbours
blueDiffH = abs(right_BB - left_BB);

//Get ths sum of the square of differences of vertical neighbours
valueH = pow(redDiffH,2) + pow(greenDiffH,2) + pow(blueDiffH,2);
```

```
    //Get ths sum of the square of differences of horizontal neighbours
    valueV = pow(redDiffV,2) + pow(greenDiffV,2) + pow(blueDiffV,2);
    valueSum = valueV + valueH;

    //Return the squareroot of the sum of differences
    return round(sqrt(valueSum));
}

//Generate energyArray
void generateEnergyMatrix(int width, int height, char* orientation){
    //Declare a dynamic 2D array to hold the energy values for all pixels
    for (int row = 1; row < height; row++) {
        for(int column= 0; column < width; column++) {
            if (orientation[0] == 'v')
                energyArray[row][column] = computeEnergy(column, row, buffer);
            else
                energyArray[row][column] = computeEnergy(row, column, buffer);
        }
    }

}

/*Declare a relax function to optimize the computation of a
  shortest path energy values*/

void relax(int row, int col, int** edgeTo, int** distTo, int width) {
    int relax = 0;
    int nextRow = row + 1;
  for (int i = -1; i <= 1; i++) {
    int nextCol = col + i;
    if (nextCol < 0 || nextCol >= width)
      continue;
    if (distTo[nextRow][nextCol] >= distTo[row][col] +
       energyArray[nextRow][nextCol]) {
      distTo[nextRow][nextCol] = distTo[row][col] +
        energyArray[nextRow][nextCol];
      edgeTo[nextRow][nextCol] = i;
    }
  }
}

//Backtrack to identify seams which is the shortest path across the energy
    array
int* backTrack(int** edgeTo, int** distTo, int height, int width){
  // Backtrack from the last row to get a shortest path
  int* seams = new int[height];
  int minCol = 0;
  int minDist = std::numeric_limits<int>::max();
  for (int col = 0; col < width; col++) {
    if (distTo[height - 1][col] < minDist) {
      minDist = distTo[height - 1][col];
      minCol = col;
    }
  }

  for (int row = height - 1; row >= 0; row--) {
        verticalSeams[row] = minCol;
        minCol = minCol - edgeTo[row][minCol];
    }
    return verticalSeams;

}

//A Transpose Matrix that makes findVerticalSeams re-usable for horizontal
     seams
guchar* transposeRGBuffer(guchar* buffer, int width, int height) {
    guchar* transposedRGBuffer;
    int size = 3 * width * height;
    transposedRGBuffer = g_try_new(guchar,size);
    g_assert(transposedRGBuffer != NULL);
    for(int col = 0; col < width; col++){
        for(int row = 0; row < height; row++){
            for (int color = 0; color < 3; color++){
                transposedRGBuffer[(col*height + row)*3 + color] =
                     buffer[(row*width+col)*3+color];
            }
        }
    }
    return transposedRGBuffer;
}

//Generate Energy Matrix
void *generateEnergyMatrix(void *arguments) {
    struct ThreadData *data = (struct ThreadData*)arguments;
```

```
   int num_rows = data -> num_rows;
   int num_cols = data -> num_cols;
   int start_col = data -> start_col;
   int stop_col = data -> stop_col;
   char* orientation = data -> orientation;
   for (int i = 0; i < num_rows; i++)
      energyArray[i] = new double[width];

   for (int row = 1; row < num_rows; row++){
      for (int column = start_col; column < stop_col; column++){
         if (orientation[0] == 'v')
            energyArray[row][column] = computeEnergy(column, row, buffer);
         else
            energyArray[row][column] = computeEnergy(row, column, buffer);
      }
   }
}

void *identifySeams(void *arguments){
   struct ThreadData *data = (struct ThreadData*)arguments;
   int num_rows = data -> num_rows;
   int num_cols = data -> num_cols;
   int start_col = data -> start_col;
   int stop_col = data -> stop_col;
   char* orientation =data -> orientation;

   for (int row = 0; row < num_rows; row++){
      for (int col = start_col; col < stop_col; col++){
         if (row == 0)
            distTo[row][col] = BASE_ENERGY;
         else
            distTo[row][col] = std::numeric_limits<int>::max();
      }
   }
        for (int row = 0; row < num_rows-1; row++){
      for (int col = start_col; col < stop_col; col++){
         relax(row, col, edgeTo, distTo, num_cols);
      }
   //pause all threads
    pthread_barrier_wait(&mybarrier);


   }


}


 //Carve out the seams from the image
/*void *carveSeams(void *arguments){
   struct ThreadData *data = (struct ThreadData*)arguments;
   int height = data -> num_rows;
   int width = data -> num_cols;
   int start_col = data -> start_col;
   int start_row = data -> start_row;
   int stop_col = data -> stop_col;
   int stop_row = data -> stop_row;
*/
void carveSeams(int width, int height){
```

```
  for (int row = 0; row < height; row++){
    //Get the RGB value before the seam
    for (int col = 0; col < verticalSeams[row]; col++){
      for (int color = 0; color < 3; color++){
        carved_imageV[(row * width + col) * 3 + color] = buffer[(row *
             width
                + col) * 3 + color];
        seams[(row * width + col) * 3 + color] = buffer[(row * width +
             col) *
            3 + color];
      }
    }

    //Get the RGB values after the seams
    for (int col = verticalSeams[row]; col < width-1; col++) {
      for (int color = 0; color < 3; color++){
        carved_imageV[(row * width + col) * 3 + color] = buffer[(row *
             width
                + col+1) * 3 + color];
        seams[(row * width + col+1) * 3 + color] = buffer[(row * width
             + col+
                1) * 3 + color];
      }
    }
  }

  //return carved_imageV;
}
```

```
LqrRetVal printSeams(LqrCarver *carver, pngwriter *pngwrt){

   gint x, y;
   guchar *rgb;
   gdouble red, green, blue;

   lqr_carver_scan_reset(carver);

   /* readout (no need to init rgb) */
   while (lqr_carver_scan(carver, &x, &y, &rgb)) {
      /* convert the output into doubles */
      red = (gdouble) rgb[0] / 255;
      green = (gdouble) rgb[1] / 255;
      blue = (gdouble) rgb[2] / 255;

      if(!rgb[0] && !rgb[1] && !rgb[2])
         pngwrt->plot(x + 1, y + 1, 1.0, 0.0, 0.0);
      else
         pngwrt->plot(x + 1, y + 1, red, green, blue);
   }
   return LQR_OK;
}

LqrRetVal write_carver_to_image(LqrCarver *carver, pngwriter *pngwrt,
    char* orientation){

   gint x, y;
   guchar *rgb;
```

```
gdouble red, green, blue;

/* resize the image canvas as needed to
   fit for the new size
   Resize based on the orientation*/

if(orientation[0] == 'v'){
   TRAP(lqr_carver_resize(carver, width-1 , height));
   pngwrt->resize(width-1, height);
}
else{
   TRAP(lqr_carver_resize(carver, width , height-1));
   pngwrt->resize(width,height-1);
}
lqr_carver_scan_reset(carver);



/* readout (no need to init rgb) */
while (lqr_carver_scan(carver, &x, &y, &rgb)) {
   /* convert the output into doubles */
   red = (gdouble) rgb[0] / 255;
   green = (gdouble) rgb[1] / 255;
   blue = (gdouble) rgb[2] / 255;

   /* plot (pngwriter's coordinates start from 1,1) */
   pngwrt->plot(x + 1, y + 1, red, green, blue);
}
return LQR_OK;
```

```
}

/* Get current time*/

double timestamp()
{
   struct timeval tval;

   gettimeofday( &tval, ( struct timezone * ) 0 );
   return ( tval.tv_sec + (tval.tv_usec / 1000000.0) );
}

int main(int argc, char **argv){
   double total_begin = timestamp();
   char * original_img = argv[1];
   char* orientation = argv[2];
   int num_threads = atoi(argv[3]);
        char* filename = argv[4];
   pngwrt.readfromfile(original_img);
   width = pngwrt.getwidth();
   height = pngwrt.getheight();

   //cout<<"Width: "<<width<<" Height: "<<height<<endl;
   double begin, end;

   std::ofstream log(filename, std::ios_base::app);
   int size = 3 * width * height;
   buffer = g_try_new(guchar,size);
buffer = rgb_buffer_from_image(&pngwrt);


g_assert(buffer != NULL);

carved_imageV = g_try_new(guchar, size);

g_assert(carved_imageV != NULL);

seams = g_try_new(guchar, size);

g_assert(seams != NULL);


LqrCarver *carver;

LqrCarver *carved_seams;

int* v_seams;

 double ec_b = 0;

      double ec_e = 0;

      double si_b = 0;

      double si_e = 0;

      double sr_b = 0;

      double sr_e = 0;

//Check the orientation to determine how to carve

//Vertical Orientation

if(orientation[0] == 'v'){

   begin = timestamp();

   verticalSeams = new int[height];

   distTo = new int*[height];

   edgeTo = new int*[height];

// Declare a dynamic 2D array to hold the energy values for all pixels

   energyArray = new double*[height];

    for (int i = 0; i < height; i++)

               distTo[i] = new int[width];
   for (int i = 0; i < height; i++)
           edgeTo[i] = new int[width];
//Spawn up threads to generate the energy matrix for (int i = 0; i <
    num_threads; i+
pthread_barrier_init(&mybarrier, NULL, num_threads);

pthread_t threads[num_threads];

struct ThreadData data[num_threads];

int col_per_thread = (width + num_threads - 1)/ num_threads;

for (int i = 0; i < num_threads; i++) {
   data[i].start_col = i*col_per_thread;
   data[i].stop_col = (i + 1)*col_per_thread;
   data[i].num_rows = height;
   data[i].num_cols = width;
   data[i].thread_id = i;
   data[i].thread_num = num_threads;
   data[i].orientation = orientation;
}
data[num_threads - 1].stop_col = width;
         ec_b = timestamp();
for (int i = 0; i < num_threads; i++){
   pthread_create(&threads[i], NULL, &generateEnergyMatrix,
       (void*)&data[i]);
}
for (int i = 0; i < num_threads; i++){
        pthread_join(threads[i], NULL);
    }
    ec_e = timestamp();



    for (int i = 0; i < num_threads; i++){
        pthread_create(&threads[i], NULL, &identifySeams, (void*)&data[i]);
    }
    for (int i = 0; i < num_threads; i++){
        pthread_join(threads[i], NULL);
    }

    verticalSeams = backTrack(edgeTo, distTo, height, width);
    si_e = timestamp();
    carveSeams(width, height);

    carver = lqr_carver_new(carved_imageV, width, height, 3);
    carved_seams = lqr_carver_new(seams, width, height, 3);
    sr_e = timestamp();

}
//Horizontal Seams
else{
    begin = timestamp();
    verticalSeams = new int[width];
    distTo = new int*[width];
    edgeTo = new int*[width];
```

```
//Declare a dynamic 2D array to hold the energy values for all pixels
energyArray = new double*[width];

for (int i = 0; i < width; i++)
                distTo[i] = new int[height];
         for (int i = 0; i < width; i++)
                edgeTo[i] = new int[height];

  pthread_t threads[num_threads];

  pthread_barrier_init(&mybarrier, NULL, num_threads );

struct ThreadData data[num_threads];
int col_per_thread = (height + num_threads - 1)/num_threads;
//Spawn up threads that will generate the energy matrix
for (int i = 0; i < num_threads; i++){
   data[i].start_col = i * col_per_thread;
   data[i].stop_col = (i + 1)*col_per_thread;
   data[i].num_rows = width;
   data[i].num_cols = height;
   data[i].thread_id = i;
   data[i].thread_num = num_threads;
   data[i].orientation = orientation;
}

     data[num_threads - 1].stop_col = height;

for (int i = 0; i < num_threads; i++){
    pthread_create(&threads[i], NULL, &generateEnergyMatrix,
        (void*)&data[i]);
}

for (int i = 0; i < num_threads; i++){
    pthread_join(threads[i], NULL);
}

cout<<"Removing horizontal seams"<<endl;

for (int i = 0; i < num_threads; i++){
                    pthread_create(&threads[i], NULL, &identifySeams,
                        (void*)&data[i]);
            }
            for (int i = 0; i < num_threads; i++){
                    pthread_join(threads[i], NULL);
            }
verticalSeams = backTrack(edgeTo, distTo, width, height);

buffer = transposeRGBuffer(buffer, width, height);
carveSeams(height, width);
/*for (int i = 0; i < NUM_OF_THREADS; i++){
    pthread_create(&threads[i], NULL, &carveSeams, (void*)&data[i]);
}
for (int i = 0; i < NUM_OF_THREADS; i++){
    pthread_join(threads[i], NULL);
}*/
```

```
        carver = lqr_carver_new(transposeRGBuffer(carved_imageV,
            height,width), width, height, 3);
        carved_seams = lqr_carver_new(transposeRGBuffer(seams, height,width),
            width, height, 3);
        end = timestamp();
        cout<<"Total SeamCarving Time: "<<(end - begin)<<endl;
    }

    //Create a Carver object with the carved image buffer
    TRAP(lqr_carver_init(carver, max_step, rigidity));
    //write_carver_to_image(carver, &pngwrt, orientation);
    printSeams(carved_seams, &pngwrt);
    lqr_carver_destroy(carver);
    pngwrt.close();
    double total_end = timestamp();
    log<<(ec_e - ec_b)<<","<<(si_e - ec_e)<<","<<(sr_e -
        si_e)<<","<<(total_end-total_begin)<<endl;
    //printf("%s%5.2f\n","Total Processing Time: ",
        (total_end-total_begin));
    pthread_barrier_destroy(&mybarrier);
    return 0;
}
```

OpenSHMEM Implementation of Seam Carving

```
#include <pngwriter.h>

#include <lqr.h>

#include <getopt.h>

#include "liquidrescale.h"

#include <math.h>

#include <limits>

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h>

#include <string.h>

#include <unistd.h>

#include <sys/time.h>

#include <shmem.h>



using namespace std;

pngwriter pngwrt(1,1,0,"out_SHMEM.png");

int BASE_ENERGY = 1000;

int ROWS_PER_THREAD = 64;

int width = 0;

int height = 0;

gfloat rigidity = 0;

gint max_step = 1;

int channels = 3;

int** energyArray;

guchar* seams;
```

```
guchar* buffer;
int* verticalSeams;
int** distTo;
int** edgeTo;
static long pSync[SHMEM_BCAST_SYNC_SIZE];


/*Copied from the liblqr example
 convert the image in the right format */
guchar * rgb_buffer_from_image(pngwriter *png)
{
    gint x, y, k, channels;
    gint w, h;
    guchar *buffer;

    /* get info from the image */
    w = png->getwidth();
    h = png->getheight();
    channels = 3;                   // we assume an RGB image here

    /* allocate memory to store w * h * channels unsigned chars */
    buffer = g_try_new(guchar, channels * w * h);
    g_assert(buffer != NULL);

    /* start iteration (always y first, then x, then colours) */
    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            for (k = 0; k < channels; k++) {
                /* read the image channel k at position x,y */
                buffer[(y * w + x) * channels + k] = (guchar) (png->dread(x
                   + 1, y + 1, k + 1) * 255);
                /* note : the x+1,y+1,k+1 on the right side are
                 *       specific the pngwriter library */
            }
        }
    }

    return buffer;
}



int computeEnergy(int x, int y, guchar* buffer){
    if (x == 0 || y == 0 || (x == width - 1) || (y == height- 1))
            return BASE_ENERGY;

   //Declare values for the RGB top, bottom, left and right neighbours
   double top_RB, top_GB, top_BB = 0.0;
   double bottom_RB, bottom_GB, bottom_BB = 0.0;
   double left_RB, left_GB, left_BB = 0.0;
   double right_RB, right_GB, right_BB = 0.0;

   /*Declare values for the differences of the vertical and horizontal
      RGB neighbours difference*/
   double redDiffH, greenDiffH, blueDiffH = 0.0;
   double redDiffV, greenDiffV, blueDiffV = 0.0;
```

```
/*Declare values for the sum of the vertical and horizontal RGB
   neighbours differences*/
double valueH, valueV, valueSum;



//Get the RGB values of the top neighbor of the current pixel


top_RB = buffer[((y-1)*width+(x))*3+0];
top_GB = buffer[((y-1)*width+(x))*3+1];
top_BB = buffer[((y-1)*width+(x))*3+2];

//Get the RGB values of the bottom neighbor of the current pixel
bottom_RB = buffer[((y+1)*width+(x))*3+0];
bottom_GB = buffer[((y+1)*width+(x))*3+1];
bottom_BB = buffer[((y+1)*width+(x))*3+2];

//Get the RGB values of the right neigbor of the current pixel
right_RB = buffer[((y)*width+(x+1))*3+0];
right_GB = buffer[((y)*width+(x+1))*3+1];
right_BB = buffer[((y)*width+(x+1))*3+2];

//Get the RGB values of the left neigbor of the current pixel
left_RB = buffer[((y)*width+(x-1))*3+0];
left_GB = buffer[((y)*width+(x-1))*3+1];
left_BB = buffer[((y)*width+(x-1))*3+2];



//Get the absolute difference of the red horizontal neighbours
```

```
    redDiffV = abs(top_RB - bottom_RB);

    //Get the absolute difference of the red horizontal neighbours
    redDiffH = abs(right_RB - left_RB);

    //Get the absolute difference of the green horizontal neighbours
    greenDiffV = abs(top_GB - bottom_GB);

    //Get the absolute difference of the green horizontal neighbours
    greenDiffH = abs(right_GB - left_GB);

    //Get the absolute difference of the blue horizontal neighbours
    blueDiffV = abs(top_BB - bottom_BB);

    //Get the absolute difference of the blue horizontal neighbours
    blueDiffH = abs(right_BB - left_BB);

    //Get ths sum of the square of differences of vertical neighbours
    valueH = pow(redDiffH,2) + pow(greenDiffH,2) + pow(blueDiffH,2);

    //Get ths sum of the square of differences of horizontal neighbours
    valueV = pow(redDiffV,2) + pow(greenDiffV,2) + pow(blueDiffV,2);
    valueSum = valueV + valueH;

    //Return the squareroot of the sum of differences
    return round(sqrt(valueSum));
}
```

```
//Build memory structure for energy matrix
int** initializeEnergyArray(int rows, int columns){
   int** energyArray;
   energyArray = (int**) shmem_malloc(rows*sizeof(int*));
   for (int i = 0; i < rows; i++)
      energyArray[i] = (int*) shmem_malloc(columns*sizeof(int));
   return energyArray;
}

int** initializeEdgeTo(int rows, int columns){
        int** edgeTo;
        edgeTo = (int**) shmem_malloc(rows*sizeof(int*));
        for (int i = 0; i < rows; i++)
                edgeTo[i] = (int*) shmem_malloc(columns*sizeof(int));
        return edgeTo;
}

int** initializeDistTo(int rows, int columns){
        int** distTo;
        distTo = (int**) shmem_malloc(rows*sizeof(int*));
        for (int i = 0; i < rows; i++)
                distTo[i] = (int*) shmem_malloc(columns*sizeof(int));
   for (int row = 0; row < rows; row++) {
            for (int col = 0; col < columns; col++) {
                if (row == 0)
                    distTo[row][col] = BASE_ENERGY;
                else
                    distTo[row][col] = std::numeric_limits<int>::max();
            }
        }

        return distTo;
}


void generateEnergyMatrix(int num_rows, int start_col, int stop_col, int
    num_pes, char* orientation){
    for (int row = 1; row < num_rows; row++) {
                for(int column = start_col; column < stop_col; column++) {
                        if (orientation[0] == 'v')
                                energyArray[row][column] =
                                    computeEnergy(column, row, buffer);
                        else
                               energyArray[row][column] = computeEnergy(row,
                                   column, buffer);
      //Put the new value in PE 0
      shmem_int_put(&energyArray[row][column], &energyArray[row][column],
          1, 0);
        }
    }
}


/*Declare a relax function to optimize the computation of a
   shortest path energy values*/
```

```
void relax(int row, int col, int width, int start_col, int stop_col, int
    me, int npes) {
   int relax = 0;
        int nextRow = row + 1;
        for (int i = -1; i <= 1; i++) {
            int nextCol = col + i;
            if (nextCol < 0 || nextCol >= width)
                continue;
            if (distTo[nextRow][nextCol] >= distTo[row][col] +
                energyArray[nextRow][nextCol]) {
                distTo[nextRow][nextCol] = distTo[row][col] +
                    energyArray[nextRow][nextCol];
                edgeTo[nextRow][nextCol] = i;

      /*Put the new values in PE 0, it will be broadcast later*/
      shmem_int_put(&distTo[nextRow][nextCol], &distTo[nextRow][nextCol],
          1, 0);
      shmem_int_put(&edgeTo[nextRow][nextCol], &edgeTo[nextRow][nextCol],
          1, 0);
            }
        }
    }

int* backTrack(int** edgeTo, int** distTo, int height, int width){
// Backtrack from the last row to get a shortest path
   int* seams = new int[height];
        int minCol = 0;
        int minDist = std::numeric_limits<int>::max();
        for (int col = 0; col < width; col++) {
            if (distTo[height - 1][col] < minDist) {
                minDist = distTo[height - 1][col];
                minCol = col;
            }
        }
     for (int row = height - 1; row >= 0; row--) {
            verticalSeams[row] = minCol;
            minCol = minCol - edgeTo[row][minCol];
        }
    return verticalSeams;

}

//A Transpose Matrix that makes findVerticalSeams re-usable
 guchar* transposeRGBuffer(guchar* buffer, int width, int height) {
   guchar* transposedRGBuffer;
   int size = 3 * width * height;
      transposedRGBuffer = g_try_new(guchar,size);
      g_assert(transposedRGBuffer != NULL);
   for(int col = 0; col < width; col++){
      for(int row = 0; row < height; row++){
         for (int color = 0; color < 3; color++){
            transposedRGBuffer[(col*height + row)*3 + color] =
                buffer[(row*width+col)*3+color];
         }
      }
   }
```

```
      return transposedRGBuffer;
}

//Find the indexes of the vertical seams to be carved out
int * identifySeams( int width, int height, int start_col, int stop_col,
    int me, int npes){
   distTo = initializeDistTo(height, width);
   edgeTo = initializeEdgeTo(height, width);

   //Initialize distTo to maximum values

    for (int row = 0; row < height - 1; row++) {
            for (int col = start_col; col < stop_col; col++) {
                relax(row, col, width, start_col, stop_col, me, npes);
            }
      //All PEs wait for one another
      shmem_barrier_all();

      /*PE 0 shares its new edges and distance values
         so all PEs have up-to-date values*/
      if (me == 0)  {
         for (int pe = 1; pe < npes; pe++){
            shmem_int_put(&distTo[row+1][0], &distTo[row+1][0], width, pe);
            shmem_int_put(&edgeTo[row+1][0], &edgeTo[row+1][0], width, pe);
         }
      }
      shmem_barrier_all();
        }
```

```
}



//Carve out the vertical seams
guchar* carveVertically(int* vertical_seams, guchar* buffer, int width,
    int height){
   guchar* carved_imageV;
   int size = 3 * width * height;

   /* allocate memory to store w * h * channels unsigned chars */
      carved_imageV = g_try_new(guchar, size);
      g_assert(carved_imageV != NULL);
   seams = g_try_new(guchar, size);
      g_assert(seams != NULL);


   for (int row = 0; row < height; row++){
      //Get the RGB value before the seam
      for (int col = 0; col < vertical_seams[row]; col++){
         for (int color = 0; color < 3; color++){
            carved_imageV[(row * width + col) * 3 + color] = buffer[(row *
                width + col) * 3 + color];
            seams[(row * width + col) * 3 + color] = buffer[(row * width +
                col) * 3 + color];
         }
      }
      //Get the RGB values after the seams
      for (int col = vertical_seams[row]; col < width-1; col++) {
        for (int color = 0; color < 3; color++){
          carved_imageV[(row * width + col) * 3 + color] = buffer[(row *
             width + col+1) * 3 + color];
          seams[(row * width + col+1) * 3 + color] = buffer[(row * width
             + col+1) * 3 + color];
        }
          }
    }

    return carved_imageV;
  }

  LqrRetVal printSeams(LqrCarver *carver, pngwriter *pngwrt){

     gint x, y;
     guchar *rgb;
     gdouble red, green, blue;

     lqr_carver_scan_reset(carver);

     /* readout (no need to init rgb) */
     while (lqr_carver_scan(carver, &x, &y, &rgb)) {
        /* convert the output into doubles */
        red = (gdouble) rgb[0] / 255;
        green = (gdouble) rgb[1] / 255;
        blue = (gdouble) rgb[2] / 255;
```

```
   if(!rgb[0] && !rgb[1] && !rgb[2])
       pngwrt->plot(x + 1, y + 1, 1.0, 0.0, 0.0);
   else
       pngwrt->plot(x + 1, y + 1, red, green, blue);

}
       return LQR_OK;
}

LqrRetVal write_carver_to_image(LqrCarver *carver, pngwriter *pngwrt,
    char* orientation){

    gint x, y;
       guchar *rgb;
    gdouble red, green, blue;
    /* resize the image canvas as needed to
     * fit for the new size */


   //Resize based on the orientation
   if(orientation[0] == 'v'){
      TRAP(lqr_carver_resize(carver, width-1 , height));
         pngwrt->resize(width-1, height);
   }
   else{
      TRAP(lqr_carver_resize(carver, width , height-1));
      pngwrt->resize(width,height-1);
      }
    lqr_carver_scan_reset(carver);



    /* readout (no need to init rgb) */
    while (lqr_carver_scan(carver, &x, &y, &rgb)) {
        /* convert the output into doubles */
        red = (gdouble) rgb[0] / 255;
        green = (gdouble) rgb[1] / 255;
        blue = (gdouble) rgb[2] / 255;

        /* plot (pngwriter's coordinates start from 1,1) */
        pngwrt->plot(x + 1, y + 1, red, green, blue);
   }
       return LQR_OK;
}

/* Get current time*/

double timestamp()
{
    struct timeval tval;

    gettimeofday( &tval, ( struct timezone * ) 0 );
    return ( tval.tv_sec + (tval.tv_usec / 1000000.0) );
}

int main(int argc, char **argv){
```

```
//static long pSync[SHMEM_BCAST_SYNC_SIZE];
 for (int i = 0; i < SHMEM_BCAST_SYNC_SIZE; i++)
            pSync[i] = SHMEM_SYNC_VALUE;

shmem_init();
int me = shmem_my_pe();
int npes = shmem_n_pes();

double total_begin = timestamp();

char * original_img = argv[1];
char* orientation = argv[2];

/*read the image and get the dimension*/

pngwrt.readfromfile(original_img);
width = pngwrt.getwidth();
height = pngwrt.getheight();



     if (me == 1)
   cout<<"Width: "<<width<<" Height: "<<height<<endl;

double begin, end;

int size = 3 * width * height;
   buffer = g_try_new(guchar,size);
buffer = rgb_buffer_from_image(&pngwrt);
```

```
   g_assert(buffer != NULL);


LqrCarver *carver;
LqrCarver *carved_seams;
//Check the orientation to determine how to carve
begin = timestamp();
if(orientation[0] == 'v'){

   verticalSeams = new int[height];

   //initialize energy array
   energyArray = initializeEnergyArray(height, width);

   /*Divide the work among the available PEs*/
   int col_per_pe = (width + npes - 1)/npes;
   int start_col = me * col_per_pe;
   int stop_col = (me + 1) * col_per_pe;
   /*Ensure that the last pe does not exceed the last row*/
   if (me == npes - 1)
      stop_col = width;

   int start_col_en = start_col;
   int stop_col_en = stop_col;

   //consider adjacent pixels for energy computation
   if(me > 0 )
```

```
      start_col_en = start_col - 1;
   if (me < npes - 1)
      stop_col_en = stop_col + 1;
   //Fill the energy matrix with the energy values of each pixel
   generateEnergyMatrix(height, start_col_en, stop_col_en, npes,
       orientation);
   shmem_barrier_all();

   //Find the vertical seam in the image
   identifySeams(width, height, start_col, stop_col, me, npes);
    if (me == 0){
                  cout<<"Removing vertical seams"<<endl;
   int* v_seams = backTrack(edgeTo, distTo, height, width);

   /*PE 0 will remove the identified seams
     there is no need to parallelize this process*/
      guchar* carved_imageV = carveVertically(v_seams,buffer, width,
          height);
      carver = lqr_carver_new(carved_imageV, width, height, 3);
      carved_seams = lqr_carver_new(seams, width, height, 3);

      end = timestamp();
      cout<<"Total Seam Carving Time: "<<(end-begin)<<endl;
   }
}
else{
   verticalSeams = new int[width];
```

```
//initialize energy array
energyArray = initializeEnergyArray(width, height);

/*divide the work among the available PEs*/
int col_per_pe = (height + npes - 1)/npes;
int start_col = me * col_per_pe;
int stop_col = (me + 1) * col_per_pe;

/*Ensure the last PE does not exceed the last row*/
if (me == npes - 1)
   stop_col = height;

/*consider adjacent pixels for energy computation*/
int start_col_en = start_col;
int stop_col_en = stop_col;

         if(me > 0 )
               start_col_en = start_col - 1;

if (me < npes - 1)
                 stop_col_en = stop_col + 1;

/*Fill the energy matrix with the energy values of each pixel */

    generateEnergyMatrix(width, start_col_en, stop_col_en, npes,
    orientation);
          shmem_barrier_all();
   /*Find the horizontal seams in the image*/

   identifySeams(height, width, start_col, stop_col, me, npes);

    if (me == 0){

                   cout<<"Removing horizontal seams"<<endl;


   int* h_seams = backTrack(edgeTo, distTo, width, height);


   /*Revome the identified horizontal seams*/

      guchar* transBuffer = transposeRGBuffer(buffer, width, height);

      guchar* carved_imageH = carveVertically(h_seams, transBuffer,

          height, width);

      carver = lqr_carver_new(transposeRGBuffer(carved_imageH,

          height,width), width, height, 3);

      carved_seams = lqr_carver_new(transposeRGBuffer(seams,

          height,width), width, height, 3);


    end = timestamp();

            cout<<"Total Seam Carving Time: "<<(end-begin)<<endl;

   }

}


//Create a Carver object with the carved image buffer

   if (me == 0){

   TRAP(lqr_carver_init(carver, max_step, rigidity));

//write_carver_to_image(carver, &pngwrt, orientation);

   printSeams(carved_seams, &pngwrt);

   lqr_carver_destroy(carver);

   pngwrt.close();
```

```
        double total_end = timestamp();
        printf("%s%5.2f\n","Total Processing Time: ",
            (total_end-total_begin));
    }

    shmem_finalize();
    return 0;

}
```

MPI Implementation of Seam Carving

```
#include <pngwriter.h>

#include <lqr.h>

#include <getopt.h>

#include "liquidrescale.h"

#include <math.h>

#include <limits>

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h>

#include <string.h>

#include <unistd.h>

#include <sys/time.h>

#include <mpi.h>



using namespace std;

pngwriter pngwrt(1,1,0,"out_MPI.png");

int BASE_ENERGY = 1000;

int ROWS_PER_THREAD = 64;

int width = 0;

int height = 0;

gfloat rigidity = 0;

gint max_step = 1;

int channels = 3;

int** energyArray;

guchar* seams;
```

```
guchar* buffer;
int* verticalSeams;
int** distTo;
int** edgeTo;
int* flattenedEnergyArray;
int* flattenedDistTo;
int* flattenedEdgeTo;
MPI_Win win;
MPI_Win win1;
MPI_Win win2;

/*Copied from the liblqr example
 convert the image in the right format */
guchar * rgb_buffer_from_image(pngwriter *png)
{
    gint x, y, k, channels;
    gint w, h;
    guchar *buffer;

    /* get info from the image */
    w = png->getwidth();
    h = png->getheight();
    channels = 3;                   // we assume an RGB image here

    /* allocate memory to store w * h * channels unsigned chars */
    buffer = g_try_new(guchar, channels * w * h);
    g_assert(buffer != NULL);
    /* start iteration (always y first, then x, then colours) */
    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            for (k = 0; k < channels; k++) {
                /* read the image channel k at position x,y */
                buffer[(y * w + x) * channels + k] = (guchar) (png->dread(x
                    + 1, y + 1, k + 1) * 255);
                /* note : the x+1,y+1,k+1 on the right side are
                 *       specific the pngwriter library */
            }
        }
    }

    return buffer;
}



int computeEnergy(int x, int y, guchar* buffer){
    if (x == 0 || y == 0 || (x == width - 1) || (y == height- 1))
            return BASE_ENERGY;

   //Declare values for the RGB top, bottom, left and right neighbours
   double top_RB, top_GB, top_BB = 0.0;
   double bottom_RB, bottom_GB, bottom_BB = 0.0;
   double left_RB, left_GB, left_BB = 0.0;
   double right_RB, right_GB, right_BB = 0.0;

   /*Declare values for the differences of the vertical and horizontal
   RGB neighbours difference*/
double redDiffH, greenDiffH, blueDiffH = 0.0;
double redDiffV, greenDiffV, blueDiffV = 0.0;

/*Declare values for the sum of the vertical and horizontal RGB
   neighbours differences*/
double valueH, valueV, valueSum;



//Get the RGB values of the top neighbor of the current pixel


top_RB = buffer[((y-1)*width+(x))*3+0];
top_GB = buffer[((y-1)*width+(x))*3+1];
top_BB = buffer[((y-1)*width+(x))*3+2];

//Get the RGB values of the bottom neighbor of the current pixel
bottom_RB = buffer[((y+1)*width+(x))*3+0];
bottom_GB = buffer[((y+1)*width+(x))*3+1];
bottom_BB = buffer[((y+1)*width+(x))*3+2];

//Get the RGB values of the right neigbor of the current pixel
right_RB = buffer[((y)*width+(x+1))*3+0];
right_GB = buffer[((y)*width+(x+1))*3+1];
right_BB = buffer[((y)*width+(x+1))*3+2];

//Get the RGB values of the left neigbor of the current pixel
left_RB = buffer[((y)*width+(x-1))*3+0];
left_GB = buffer[((y)*width+(x-1))*3+1];
```

```
left_BB = buffer[((y)*width+(x-1))*3+2];



//Get the absolute difference of the red horizontal neighbours
redDiffV = abs(top_RB - bottom_RB);

//Get the absolute difference of the red horizontal neighbours
redDiffH = abs(right_RB - left_RB);

//Get the absolute difference of the green horizontal neighbours
greenDiffV = abs(top_GB - bottom_GB);

//Get the absolute difference of the green horizontal neighbours
greenDiffH = abs(right_GB - left_GB);

//Get the absolute difference of the blue horizontal neighbours
blueDiffV = abs(top_BB - bottom_BB);

//Get the absolute difference of the blue horizontal neighbours
blueDiffH = abs(right_BB - left_BB);

//Get ths sum of the square of differences of vertical neighbours
valueH = pow(redDiffH,2) + pow(greenDiffH,2) + pow(blueDiffH,2);

//Get ths sum of the square of differences of horizontal neighbours
valueV = pow(redDiffV,2) + pow(greenDiffV,2) + pow(blueDiffV,2);
valueSum = valueV + valueH;
```

```
   //Return the squareroot of the sum of differences
   return round(sqrt(valueSum));
}

//Build memory structure for energy matrix
int** initializeEnergyArray(int rows, int columns){
   int** energyArray;
   energyArray = (int**) malloc(rows*sizeof(int*));
   for (int i = 0; i < rows; i++)
      energyArray[i] = (int*) malloc(columns*sizeof(int));
   return energyArray;
}



int* initialize1DArray(int rows, int columns){
        int* array1D = (int*)malloc(rows*columns*sizeof(int));
    for ( int i = 0; i < rows*columns; i++)
                array1D[i] = 0;
        return array1D;
}



/*int* initializeDistTo1D(int rows, int columns){
        int* array1D = (int*)malloc(rows*columns*sizeof(int));
         for ( int i = 0; i < rows*columns; i++)
      int col = getCol(i, columns);
      int row = getRow(i, columns);
      if (row == 0)
```

```
                array1D[i] = BASE_ENERGY;
            else
          array1D[i] = std::numeric_limits<int>::max();
        return array1D;
}*/

int** initializeEdgeTo(int rows, int columns){
        int** edgeTo;
        edgeTo = (int**) malloc(rows*sizeof(int*));
        for (int i = 0; i < rows; i++)
                edgeTo[i] = (int*) malloc(columns*sizeof(int));
        return edgeTo;
}

int** initializeDistTo(int rows, int columns){
        int** distTo;
        distTo = (int**) malloc(rows*sizeof(int*));
        for (int i = 0; i < rows; i++)
                distTo[i] = (int*) malloc(columns*sizeof(int));
   for (int row = 0; row < rows; row++) {
            for (int col = 0; col < columns; col++) {
                if (row == 0)
                    distTo[row][col] = BASE_ENERGY;
                else
                    distTo[row][col] = std::numeric_limits<int>::max();
            }
       }
```

```
      return distTo;
}

//Convert a 2D array to 1D

int* flattenArray(int** arrayToFlatten, int num_rows, int num_cols){
   int* flattenedArray = (int*) malloc(num_rows*num_cols*sizeof(int));
   int counter = 0;

   for (int row = 0; row < num_rows; row++){
      for(int col = 0; col < num_cols; col++){
         flattenedArray[counter] = arrayToFlatten[row][col];
         counter++;
      }
   }
      return flattenedArray;
}

//Convert a 1D array to 2D
int** unflattenArray(int* flattenedArray, int num_rows, int num_cols){
   int** unflattenedArray = (int**) malloc(num_rows*sizeof(int*));
        for (int i = 0; i < num_rows; i++)
                unflattenedArray[i] = (int*) malloc(num_cols*sizeof(int));
   int counter = 0;
   for (int row = 0; row < num_rows; row++){
      for (int col = 0; col < num_cols; col++){

         unflattenedArray[row][col] = flattenedArray[counter];
```

```
            counter++;
        }
    }
    return unflattenedArray;
}

//Given a 1D array, find the row of a given cell if it were to be a 2D
    array
int getRow(int cell, int num_cols){
        return (cell/num_cols);
}

//Given a 1D array, find the col of a given cell if it were to be a 2D
    array
int getCol(int cell, int num_cols){
    return cell%num_cols;
}

int* initializeDistTo1D(int rows, int columns){
        int* array1D = (int*)malloc(rows*columns*sizeof(int));
         for ( int cell = 0; cell < rows*columns; cell++){
                // int col = getCol(i, columns);
                 int row = getRow(cell, columns);
                 if (row == 0)
                         array1D[cell] = BASE_ENERGY;
                 else
                         array1D[cell] = std::numeric_limits<int>::max();
    }
```

```
       return array1D;
}



void generateEnergyMatrix(int num_rows, int start_col, int stop_col, int
   me, int num_pes, char* orientation){
   for (int row = 1; row < num_rows; row++) {
              for(int column = start_col; column < stop_col; column++) {
        int cell = row*width+column;
                    if (orientation[0] == 'v')
                            flattenedEnergyArray[cell] =
                               computeEnergy(column, row, buffer);
                    else
                           flattenedEnergyArray[cell] =
                              computeEnergy(row, column, buffer);

      /*Put the new value in PE 0*/
      MPI_Win_lock_all(0, win);
        if(me > 0)
          MPI_Put(&flattenedEnergyArray[cell], 1, MPI_INT, 0, cell, 1,
             MPI_INT, win);
      MPI_Win_flush_all(win);
      MPI_Win_unlock_all(win);
       }
   }
}
```

```
int* backTrack(int** edgeTo, int** distTo, int height, int width){
// Backtrack from the last row to get a shortest path
   int* seams = new int[height];
        int minCol = 0;
        int minDist = std::numeric_limits<int>::max();
        for (int col = 0; col < width; col++) {
            if (distTo[height - 1][col] < minDist) {
                minDist = distTo[height - 1][col];
                minCol = col;
            }
        }
    for (int row = height - 1; row >= 0; row--) {
            verticalSeams[row] = minCol;
            minCol = minCol - edgeTo[row][minCol];
        }
   return verticalSeams;

}

//A Transpose Matrix that makes findVerticalSeams re-usable
 guchar* transposeRGBuffer(guchar* buffer, int width, int height) {
   guchar* transposedRGBuffer;
   int size = 3 * width * height;
      transposedRGBuffer = g_try_new(guchar,size);
      g_assert(transposedRGBuffer != NULL);
   for(int col = 0; col < width; col++){
      for(int row = 0; row < height; row++){
         for (int color = 0; color < 3; color++){
            transposedRGBuffer[(col*height + row)*3 + color] =
                buffer[(row*width+col)*3+color];
         }
      }
   }
      return transposedRGBuffer;
}

/*Declare a relax function to optimize the computation of a
 *        shortest path energy values*/

void relax(int row, int col, int width, int start_col, int stop_col, int
    me, int npes) {
        int relax = 0;
        int nextRow = row + 1;
        int cell = row*width + col;
        for (int i = -1; i <= 1; i++) {
            int nextCol = col + i;
            int nextCell = nextRow*width + nextCol;
            if (nextCol < 0 || nextCol >= width)
                continue;
      if (flattenedDistTo[nextCell] >= flattenedDistTo[cell] +
          flattenedEnergyArray[nextCell]){
         flattenedDistTo[nextCell] = flattenedDistTo[cell] +
             flattenedEnergyArray[nextCell];
         flattenedEdgeTo[nextCell] = i;
                /*Put the new values in PE 0, other PEs will fetch these
```

```
                  values later*/
          MPI_Win_lock_all(0, win1);
          if (me > 0)
                          MPI_Put(&flattenedDistTo[nextCell], 1, MPI_INT, 0,
                             nextCell, 1, MPI_INT, win1);
          MPI_Win_flush_all(win1);
              MPI_Win_unlock_all(win1);

          MPI_Win_lock_all(0, win2);
          if (me > 0)
                          MPI_Put(&flattenedEdgeTo[nextCell], 1, MPI_INT, 0,
                             nextCell, 1, MPI_INT, win2);
          MPI_Win_flush_all(win2);
          MPI_Win_unlock_all(win2);
                }
         }
     }



//Find the indexes of the vertical seams to be carved out
int * identifySeams( int width, int height, int start_col, int stop_col,
   int me, int npes){
   for (int row = 0; row < height - 1; row++) {
           for (int col = start_col; col < stop_col; col++) {
              relax(row, col, width, start_col, stop_col, me, npes);

          }
```

```
      MPI_Barrier(MPI_COMM_WORLD);
      int nextCell = (row+1)*width;

      /*Other PEs get the new values for the next row from PE 0*/

      if (npes > 1){
                 MPI_Bcast(&flattenedDistTo[nextCell], width, MPI_INT, 0,
                    MPI_COMM_WORLD);

                 MPI_Bcast(&flattenedEdgeTo[nextCell], width, MPI_INT, 0,
                    MPI_COMM_WORLD);
        }
      MPI_Barrier(MPI_COMM_WORLD);
   }

}




//Carve out the vertical seams
guchar* carveVertically(int* vertical_seams, guchar* buffer, int width,
    int height){
   guchar* carved_imageV;
   int size = 3 * width * height;

   /* allocate memory to store w * h * channels unsigned chars */
      carved_imageV = g_try_new(guchar, size);
      g_assert(carved_imageV != NULL);
```

```
  seams = g_try_new(guchar, size);
    g_assert(seams != NULL);



  for (int row = 0; row < height; row++){
    //Get the RGB value before the seam
    for (int col = 0; col < vertical_seams[row]; col++){
      for (int color = 0; color < 3; color++){
        carved_imageV[(row * width + col) * 3 + color] = buffer[(row *
           width + col) * 3 + color];
        seams[(row * width + col) * 3 + color] = buffer[(row * width +
           col) * 3 + color];
      }
    }

    //Get the RGB values after the seams
    for (int col = vertical_seams[row]; col < width-1; col++) {
      for (int color = 0; color < 3; color++){
        carved_imageV[(row * width + col) * 3 + color] = buffer[(row *
           width + col+1) * 3 + color];
        seams[(row * width + col+1) * 3 + color] = buffer[(row * width
           + col+1) * 3 + color];
      }
        }
  }

  return carved_imageV;
}
```

```
LqrRetVal printSeams(LqrCarver *carver, pngwriter *pngwrt){

    gint x, y;
    guchar *rgb;
    gdouble red, green, blue;

    lqr_carver_scan_reset(carver);

    /* readout (no need to init rgb) */
    while (lqr_carver_scan(carver, &x, &y, &rgb)) {
        /* convert the output into doubles */
        red = (gdouble) rgb[0] / 255;
        green = (gdouble) rgb[1] / 255;
        blue = (gdouble) rgb[2] / 255;

   if(!rgb[0] && !rgb[1] && !rgb[2])
      pngwrt->plot(x + 1, y + 1, 1.0, 0.0, 0.0);
   else
      pngwrt->plot(x + 1, y + 1, red, green, blue);

}
      return LQR_OK;
}

LqrRetVal write_carver_to_image(LqrCarver *carver, pngwriter *pngwrt,
    char* orientation){
```

```
 gint x, y;
    guchar *rgb;
 gdouble red, green, blue;
 /* resize the image canvas as needed to
  * fit for the new size */



//Resize based on the orientation
if(orientation[0] == 'v'){
   TRAP(lqr_carver_resize(carver, width-1 , height));
      pngwrt->resize(width-1, height);
}
else{
   TRAP(lqr_carver_resize(carver, width , height-1));
   pngwrt->resize(width,height-1);
   }
 lqr_carver_scan_reset(carver);



 /* readout (no need to init rgb) */
 while (lqr_carver_scan(carver, &x, &y, &rgb)) {
     /* convert the output into doubles */
     red = (gdouble) rgb[0] / 255;
     green = (gdouble) rgb[1] / 255;
     blue = (gdouble) rgb[2] / 255;

     /* plot (pngwriter's coordinates start from 1,1) */
     pngwrt->plot(x + 1, y + 1, red, green, blue);
  }
     return LQR_OK;
}

/* Get current time*/

double timestamp()
{
    struct timeval tval;

    gettimeofday( &tval, ( struct timezone * ) 0 );
    return ( tval.tv_sec + (tval.tv_usec / 1000000.0) );
}

int main(int argc, char **argv){
   MPI_Init(&argc, &argv);
   int me, npes;
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &me);

   double total_begin = timestamp();

   char * original_img = argv[1];
   char* orientation = argv[2];

   /*read the image and get the dimension*/

   pngwrt.readfromfile(original_img);
```

```
width = pngwrt.getwidth();
height = pngwrt.getheight();

int cells_num = height*width;

// Allocate a window for each buffer
MPI_Win_allocate(cells_num*sizeof(int), sizeof(int), MPI_INFO_NULL,
   MPI_COMM_WORLD, &flattenedEnergyArray, &win);
MPI_Win_allocate(cells_num*sizeof(int), sizeof(int), MPI_INFO_NULL,
   MPI_COMM_WORLD, &flattenedDistTo, &win1);
MPI_Win_allocate(cells_num*sizeof(int), sizeof(int), MPI_INFO_NULL,
   MPI_COMM_WORLD, &flattenedEdgeTo, &win2);

    if (me == 0)
   cout<<"Width: "<<width<<" Height: "<<height<<endl;

double begin, end;

int size = 3 * width * height;
   buffer = g_try_new(guchar,size);
buffer = rgb_buffer_from_image(&pngwrt);

   g_assert(buffer != NULL);



LqrCarver *carver;
LqrCarver *carved_seams;
//Check the orientation to determine how to carve
```

```
begin = timestamp();

if(orientation[0] == 'v'){

  verticalSeams = new int[height];

  /*initialize energy 1D and 2D Array*/
  energyArray = initializeEnergyArray(height, width);
  flattenedEnergyArray = initialize1DArray(height, width);

  /*initialize distTo 1D and 2D arrays*/
  distTo = initializeDistTo(height, width);
  flattenedDistTo = initializeDistTo1D(height, width);

  /*initialize edgeTo 1D and 2D arrays*/

  edgeTo = initializeEdgeTo(height, width);
  flattenedEdgeTo = initialize1DArray(height, width);

  /*Divide the work among the available PEs*/
  int cell_per_pe = (width*height + npes - 1)/npes;
  int col_per_per = (width + npes - 1)/npes;

  int start_cell = me * cell_per_pe;
  int stop_cell = (me + 1) * cell_per_pe;

  int start_col = me*col_per_per;
  int stop_col = (me+1) * col_per_per;
```

```
/*Ensure that the last pe does not exceed the last row*/
if (me == npes - 1)
   stop_cell = width*height;

if (me == npes - 1 )
          stop_col = width;

int start_col_en = start_col;
int stop_col_en = stop_col;

//consider adjacent pixels for energy computation
if(me > 0 )
      start_col_en = start_col - 1;
   if (me < npes - 1)
      stop_col_en = stop_col + 1;



   //Fill the energy matrix with the energy values of each pixel


   generateEnergyMatrix(height, start_col_en, stop_col_en, me, npes,
       orientation);


   MPI_Barrier(MPI_COMM_WORLD);


   /* PE 0 broadcasts the values of the 1D energy Array*/
   if (npes > 1)
      MPI_Bcast(&flattenedEnergyArray[0], width*height, MPI_INT, 0,
         MPI_COMM_WORLD);

   MPI_Barrier(MPI_COMM_WORLD);

   identifySeams(width, height, start_col, stop_col, me, npes);
   if (me == 0){
      distTo = unflattenArray(flattenedDistTo, height, width);
      edgeTo = unflattenArray(flattenedEdgeTo, height, width);
            cout<<"Removing vertical seams"<<endl;

   int* v_seams = backTrack(edgeTo, distTo, height, width);

   /*PE 0 will remove the identified seams
     there is no need to parallelize this process*/
      guchar* carved_imageV = carveVertically(v_seams,buffer, width,
         height);
      carver = lqr_carver_new(carved_imageV, width, height, 3);
      carved_seams = lqr_carver_new(seams, width, height, 3);

      end = timestamp();
      cout<<"Total Seam Carving Time: "<<(end-begin)<<endl;
   }
}
else{
   verticalSeams = new int[width];

   //initialize energy array
   energyArray = initializeEnergyArray(width, height);
```

```
/*divide the work among the available PEs*/
int col_per_pe = (height + npes - 1)/npes;
int start_col = me * col_per_pe;
int stop_col = (me + 1) * col_per_pe;

/*Ensure the last PE does not exceed the last row*/
if (me == npes - 1)
  stop_col = height;

/*consider adjacent pixels for energy computation*/
int start_col_en = start_col;
int stop_col_en = stop_col;

         if(me > 0 )
              start_col_en = start_col - 1;

if (me < npes - 1)
                stop_col_en = stop_col + 1;

/*Fill the energy matrix with the energy values of each pixel */

   generateEnergyMatrix(width, start_col_en, stop_col_en, me,npes,
   orientation);
         //shmem_barrier_all();

/*Find the horizontal seams in the image*/
identifySeams(height, width, start_col, stop_col, me, npes);
```

```
     if (me == 0){
                      cout<<"Removing horizontal seams"<<endl;

    int* h_seams = backTrack(edgeTo, distTo, width, height);

    /*Revome the identified horizontal seams*/
       guchar* transBuffer = transposeRGBuffer(buffer, width, height);
       guchar* carved_imageH = carveVertically(h_seams, transBuffer,
           height, width);
       carver = lqr_carver_new(transposeRGBuffer(carved_imageH,
           height,width), width, height, 3);
       carved_seams = lqr_carver_new(transposeRGBuffer(seams,
           height,width), width, height, 3);

     end = timestamp();
                cout<<"Total Seam Carving Time: "<<(end-begin)<<endl;
    }
 }

 //Create a Carver object with the carved image buffer
    if (me == 0){
    TRAP(lqr_carver_init(carver, max_step, rigidity));
 //write_carver_to_image(carver, &pngwrt, orientation);
    printSeams(carved_seams, &pngwrt);
    lqr_carver_destroy(carver);
    pngwrt.close();

    double total_end = timestamp();
    printf("%s%5.2f\n","Total Processing Time: ",
        (total_end-total_begin));
  }
  MPI_Win_free(&win);
  MPI_Finalize();
  return 0;

}
```