Jacob Stultz 5/16/2008
6.338/18.337 Final Project Report Prof Alan Edelman

# Seam Carving:
# Parallelizing a novel new image resizing algorithm

## Introduction

Seam carving is a new method of automatically resizing images in a content aware fashion. The idea is to scale images up or down, without necessarily maintaining the image aspect ratio, while simultaneously attempting to minimize visually destructive image distortion and removal of important image information. An algorithm is used to determine, based on a specific per-pixel "importance" function, paths of pixels through the image which theoretically contain the least relevant image information. This process is fairly computationally and data intensive, so developing a scheme to operate on images in parallel could be valuable. The majority of processors sold in commercially available computers currently have at least two cores, with four and eight core processors on their way into prevalence as well. Therefore, a parallelized implementation of the seam carving algorithm should be useful to the majority of people doing seam carving on images.

## Motivation

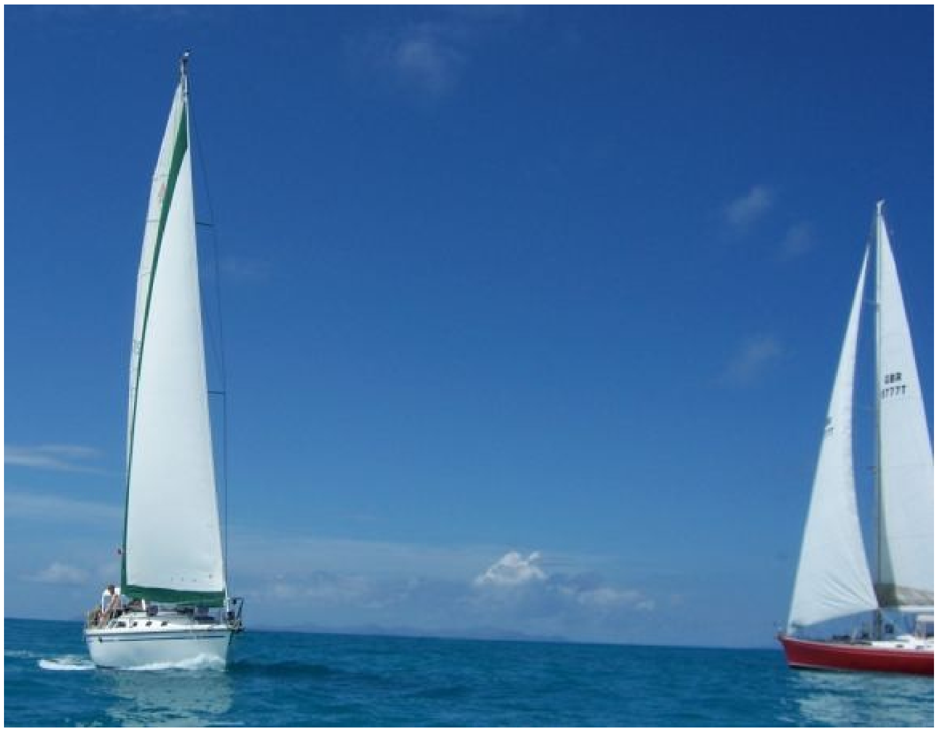
Displaying the layout of content content containing images often requires that the images be resized, whether the aspect ratio is changed or not. This can be necessary in many different areas, including multimedia websites, GUI computer applications, mobile applications on cell phones and PDAs, and even in developing on-paper layouts for magazines or newspapers. The layout often puts constraints on the dimensions of an image, and in some cases these constraints can change dynamically in real time (if a user resizes his browser window while viewing a website, for example).

It is desirable, in these cases, to maintain the overall coherency of the images contained in the layout. To do so, a few things must be prevented as much as possible: loss of important image information (people or objects in the image), and damaging distortion, via skewing, stretching, shrinking, or otherwise. Therefore, an algorithm should be used that can resize images while minimizing these potential negative effects.

## Existing Methods

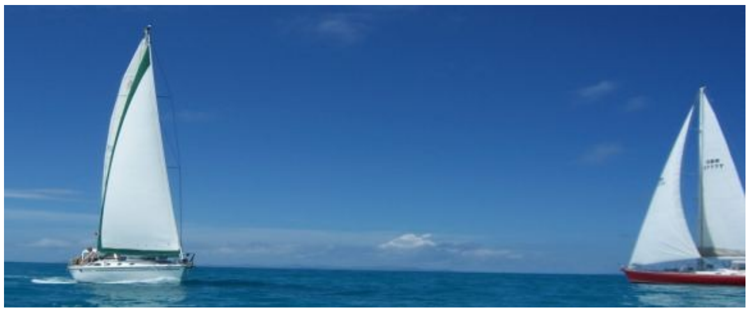
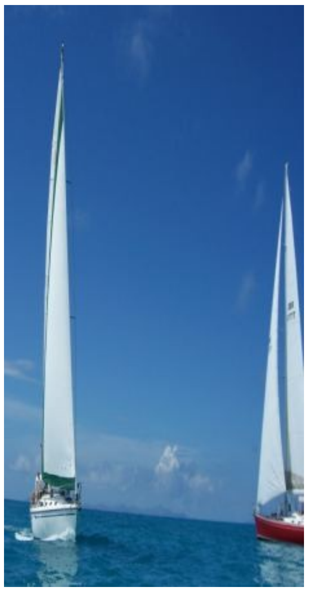
There are, of course, a number of existing methods for automatically resizing images, some more trivial than others. I will discuss a few of these and how well they accomplish the goals specified above.

One of the simplest and oldest image resizing methods available is cropping; removing large blocks from the outside of an image. The primary advantages of cropping is that the actual removal of information is trivial, and it is guaranteed to always maintain the aspect ratio of the image and will not result in any distortion. However, it has two significant disadvantages, particularly when automated. The first is that the important pixels in an image are not always in the center; often they are nearer to the edges of an image and any algorithm must have some way of reliably determining the primary subject(s) of the image so that they aren't removed. Secondly, some images simply can't be feasibly cropped. Examine the image below:



The two foci are at the edge of the image; there is no way that the image can be cropped without removing one of them, even if the cropping is done manually.

Another fairly trivial image resizing technique is scaling of the image. This, like cropping, is extremely simple, albeit slightly more computationally intensive. It has the advantage that it retains all components of the image, and can easily be done automatically, but if the aspect ratio is changed, it inevitably causes damaging distortion, as shown below:

Even if the aspect ratio is not changed, growing or shrinking an image by a scale factor of more than 2 can still result in significant image degradation.

## New Approach

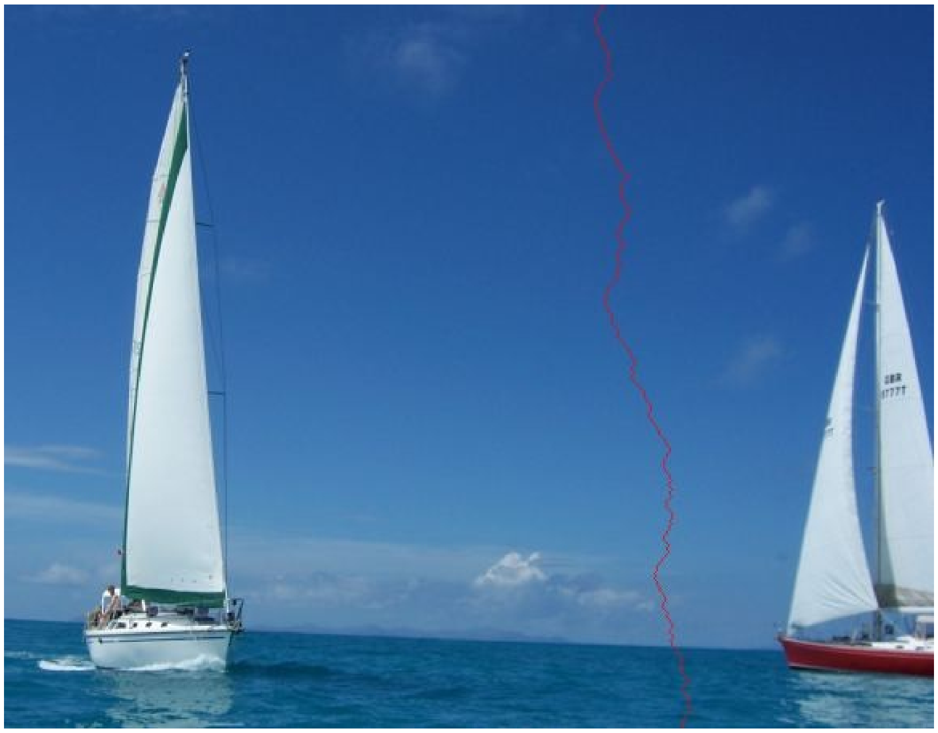
Since neither of the aforementioned techniques seem satisfactory, some new techniques should be investigated that attempt to dynamically determine the importance of various areas or pixels within the image, and not be restricted to only removing content around the edges. Assuming a metric to determine relative importance, one might develop a scheme to remove individual columns or rows of an image deemed to be the least important (instead of in large contiguous blocks, ala cropping).

This is still not an ideal solution, however. Individual columns and rows may contain both largely irrelevant pixels, as well as key components of the main image subject. Additionally, this will likely still result in significant distortion: imagine a straight diagonal line across an image, with columns or rows that cross it removed. The line would no longer be straight, which could be a hugely negative effect, particularly in images with many geometric shapes.

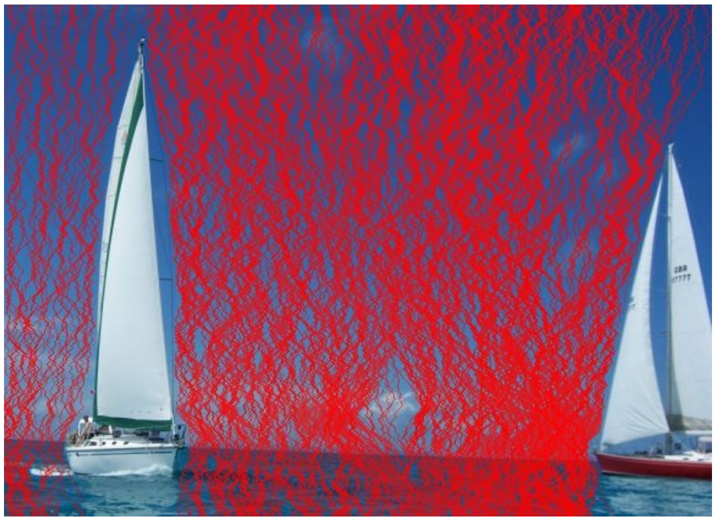
Instead, suppose that importance was determined on a per-pixel basis, and then the least important pixel in each column or row was removed; this solves the problem of columns or rows containing pixels of widely varying importance; however, the scattered removal of pixels could result in significant image distortion in the form of localized skewing.

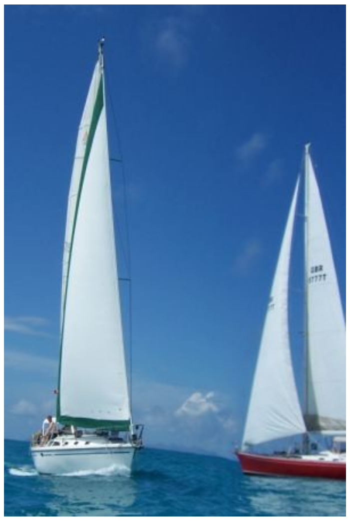
## Seam Carving

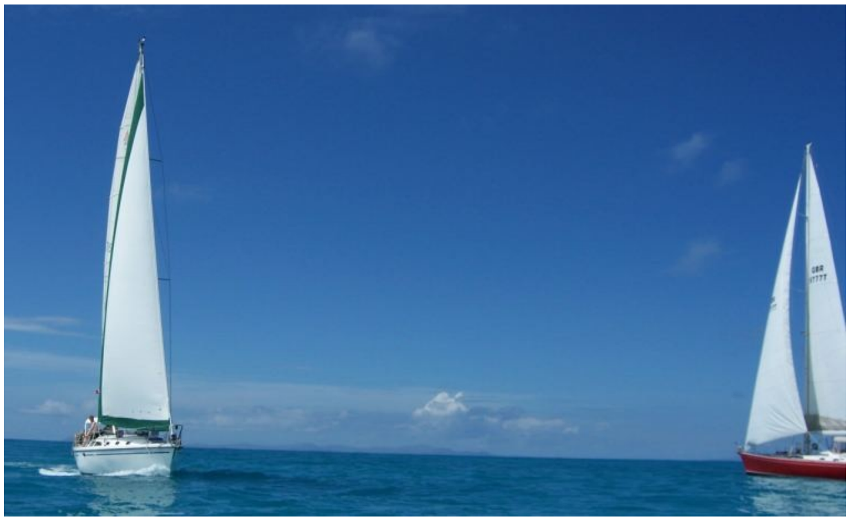
A sort of compromise between the two previous proposed methods is seam carving: instead of removing the columns and rows of least importance, a more flexible method is used. The least important contiguous paths from the top of the image to the bottom, or from the left side of the image to the right side (depending on the direction of resizing) are removed instead. In the case of a vertical path, or "seam", it contains only one pixel per row (per column in the horizontal case), and each pixel must be either directly or diagonally adjacent to the next. In this way, an image can be shrunk with less skewing than removing any pixel per row or column, while being able to maintain more relevant image information than simply removing entire rows and columns. The least important vertical seam is shown in red in the sailing picture below:



To shrink the image, this one pixel wide seam is removed, and then the process is repeated. Each iteration, the least important seam is found, and then removed, until enough seams are removed such that the image is the desired size. Conversely, to grow an image, these seams are found, but instead of removing them, new seams are added along side them with pixel values created by averaging those around them. The results of this can be seen below, where the image is shrunk horizontally by 180 pixels, and grown horizontally by 180 pixels. The first image shows all of the seams calculated in the process.







In order to generate these seams, however, there must be a way of determining the importance of each individual pixel. An "energy" function is used for this. There are a number of different potential functions that can be used, but the function used here is a simple difference function, where the importance of each pixel is determined by how much it differs from the surrounding pixels:

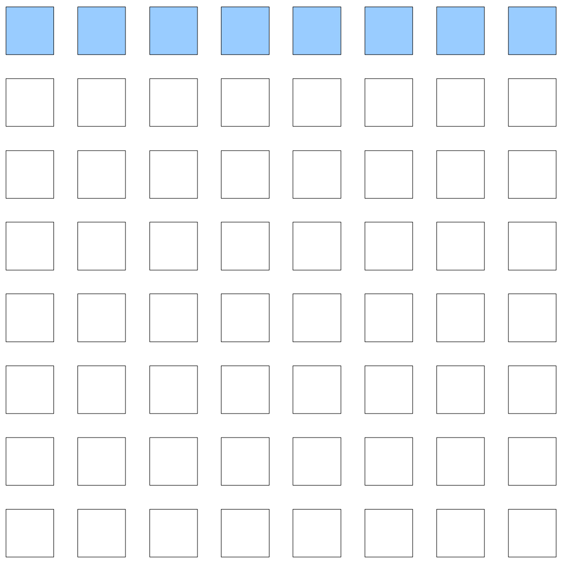
$$E(x,y) = |d/dx(x,y)| + |d/dy(x,y)|$$

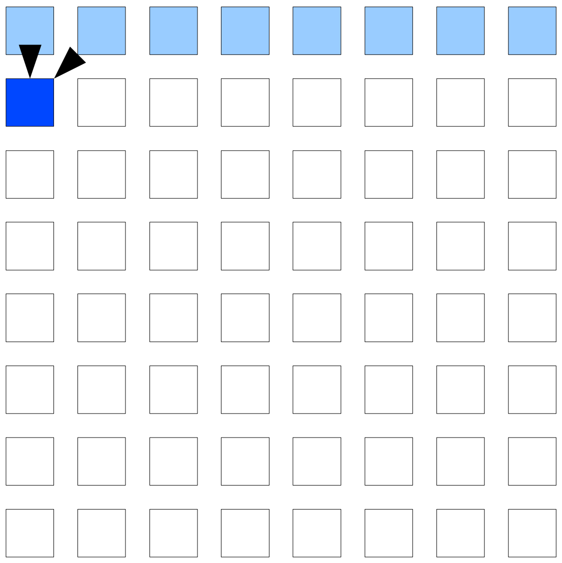
In practice, the average absolute difference of the pixel's color value from the four pixels directly adjacent to it determines that pixels importance. If a pixel is surrounded by pixels that are nearly the same color, it is likely not important, whereas higher contrast regions of an image are likely to contain relevant subjects.

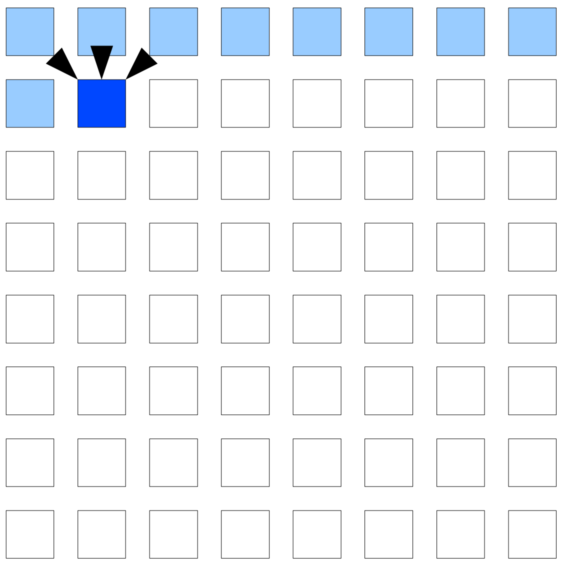
Once the energy value of every pixel in the image is determined, the lowest energy path from one side of the image to the other must be found. A dynamic programming algorithm is used to determine the minimum energy path ending at each pixel on the edge of the image. The basic formula of the algorithm is:

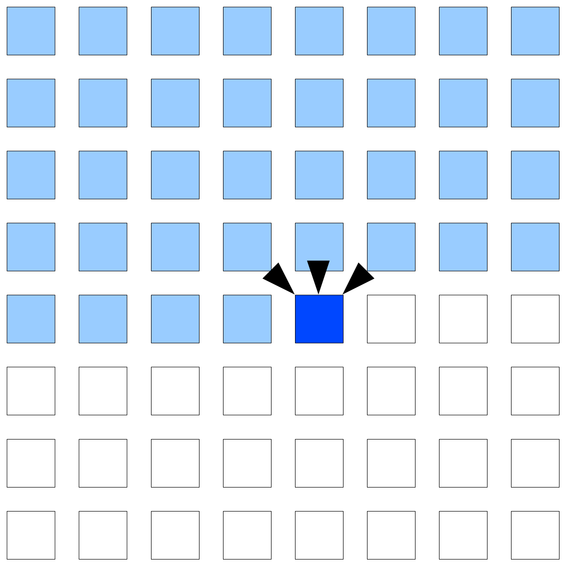
$$M(x,y) = E(x,y) + min[M(x-1, y-1), M(x, y-1), M(x+1, y-1)]$$

Assume that the image is being resized horizontally, so we are looking to discover minimum energy vertical seams to remove. The M (or minimum path) value for each pixel in the top row is set equal to that pixels energy. Then, the algorithm iterates down each successive row, determining each pixels minimum path value by adding it's own energy value to the minimum value of the three pixels above it. Once the last row is reached, the minimum path value for each pixel in that row represents the lowest energy required to reach that pixel from the top of the image. The lowest value is chosen, and then the path of the seam is easily determined from backtracking up the image and the minimum path values for the other pixels. The seam is found, and then removed, and the process is repeated again, until the correct number of seams have been removed. Part of this process is demonstrated graphically below, on a hypothetical 8x8 pixel image. The light blue blocks have been calculated, and the block currently being calculated is dark blue. The arrows represent which previously calculated values it compares and depends on.







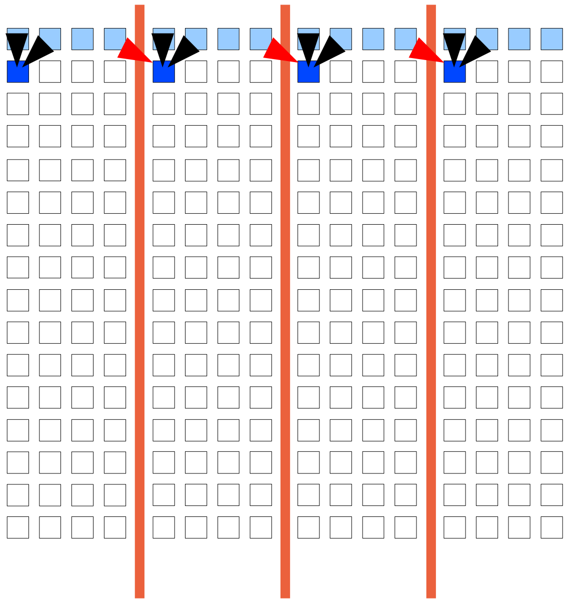


## Parallelization

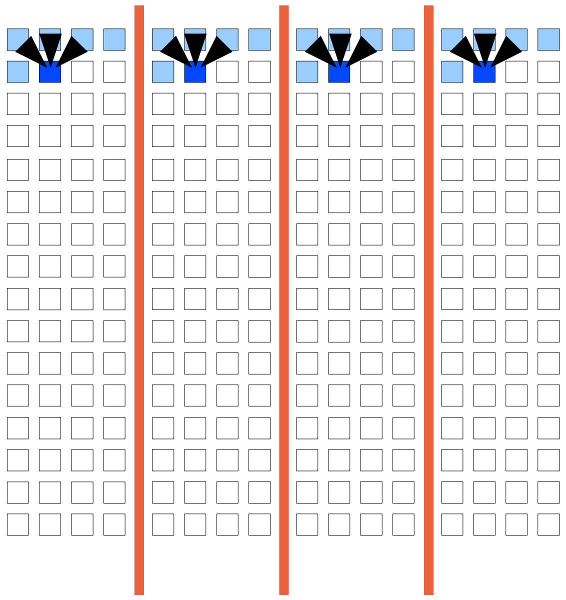
I chose to concentrate on parallelizing the energy and minimum path calculations, as they are the most computation and data intensive aspects of the algorithm. The determination of the lowest energy seam from the minimum path map and the removal of the pixels are both relatively straightforward and fast.

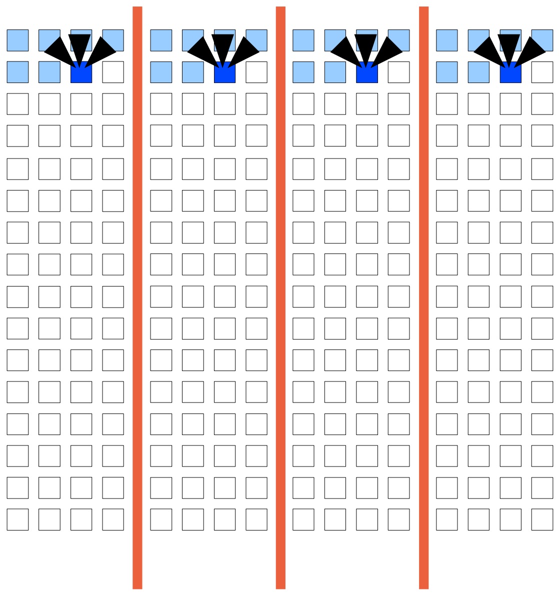
The energy map calculation is trivially parallelizable. If the image data is split across processors in large columns, the individual processors can calculate the energy of each pixel completely independently of the others. Only one bit of communication must be done at the beginning: each processor must have the two columns of pixels immediately adjacent to its block of columns. This can be done all at once at the initiation of the calculation. An important note to make about the energy map calculation, however, is that it only occurs once, at the start of the resize. After each seam is removed, the only pixels for which the energy must be recalculated are those directly or diagonally adjacent to the seam.

The minimum path calculation is slightly more complicated, since each processor must depend on some of the calculations done by those working on columns adjacent to its data set. Additionally, this calculation is done many times; each time a seam is removed, most of the minimum path map must be recalculated. The specific implementation of the parallel algorithm is outlined below, with diagrams representing a 16x16 pixel image. Red vertical lines delineate between processors, and red arrows indicate where information must be sent from one processor to another via MPI_Bsend (buffered send) and MPI_Recv (receive) calls.

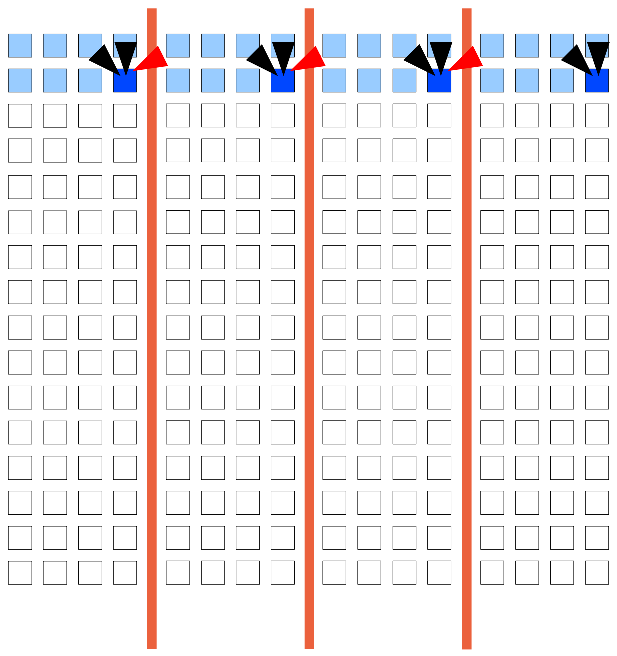
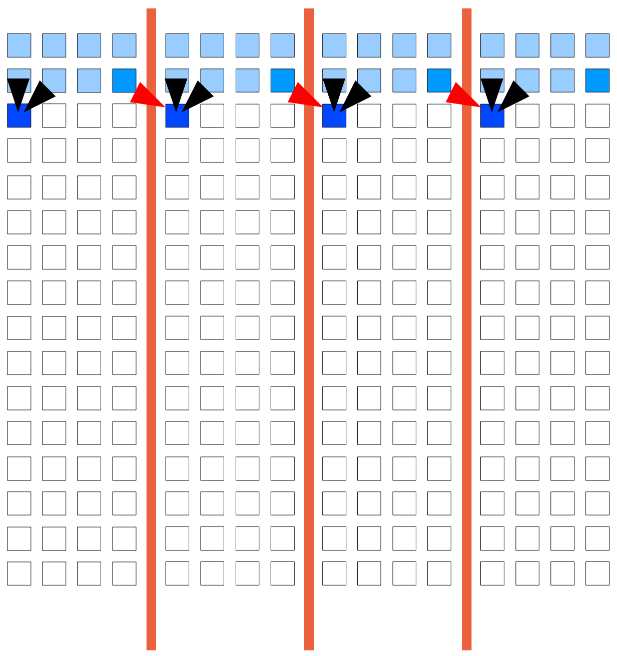


Each processor begins to calculate minimum path values in parallel; for the first calculations, and any later pixels which are on the edge of a processor's data set, a message must be sent from the adjacent processor with the minimum path value of the diagonal pixel. This proceeds across the row of pixels like so:

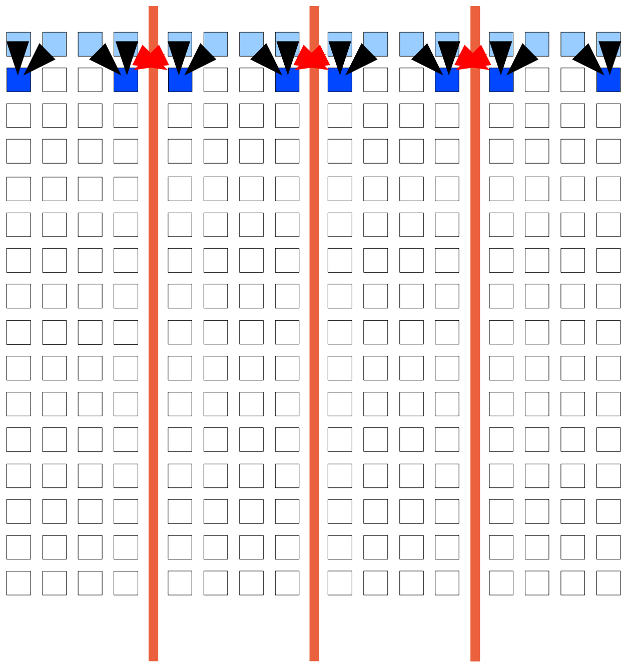




On the last pixel per row, the same MPI message send occurs, in the other direction.

This row by row procedure is continually repeated, until the entire minimum path map has been created. However, there is a potential message passing bottleneck. As highlighted in the image on the right, at the start of the next row, the calculation of the leftmost pixel in the new row depends on the result of the right most pixel in the previous row on the processor adjacent to the left. The processors are not guaranteed to be completely synchronized, however, so it is possible that one processor may be attempting to calculate the value of the pixel on the new row before the processor it is waiting for has finished the calculations on the previous row. Instead, once a processor starts working on a new row, it should calculate the minimum path value for the pixels on both ends of the row and perform a buffered send of both values, so that when the other processor needs that value, it is more likely to be available right away.



## Algorithm Analysis

### Energy Map Calculations

As previously explained, the energy map calculation is trivially parallelizable. For a serial implementation, the computation time should be O(width * height), as it depends entirely on the number of pixels in the image, and should be constant time per pixel. For the parallel implementation, each processor should be able to complete its calculations in the same amount of time for the number of pixels it has to process, so the computation time should be O(width * height / P).

The communication required for the parallelized energy calculation is small. As mentioned before, each processor needs the single columns of pixels adjacent to its set of columns. There should be on the order of P such columns, so the total communication required is O(height * P), and this communication can be done all at once before the calculations begin, so it should be fast and efficient, minimally impacting performance.

Based on the above analysis, an approximate 2x speedup should be expected for the parallel implementation on a 2 CPU system.

### Minimum Path Map Calculations

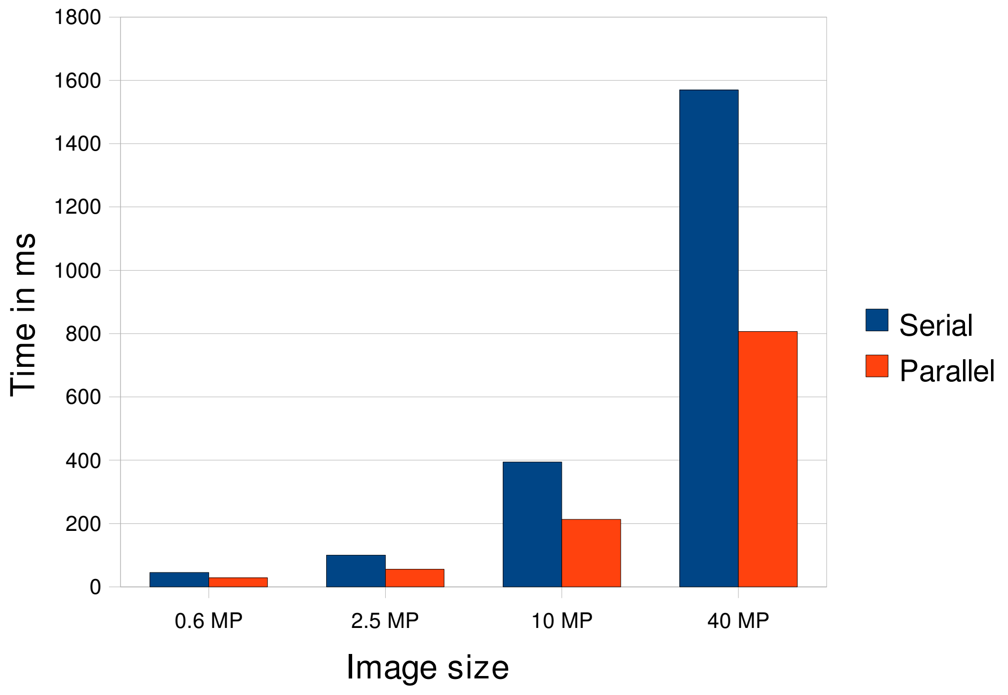
The minimum path map calculation is also constant time per pixel, so the overall computation time for the serial implementation should be O(width * height). The same again, is true of the parallel implementation: each processor is operating on 1/P of the total pixels, so the expected computation time should be O(width * height / P).

The primary difference here is in the communication. The total data to be communicated is O(height * P) again, but it cannot all be sent at once; the processors must stay somewhat synchronized, as they are depending on each other for data to complete their calculations. If the processors progress at different rates, some may be stalled waiting to receive messages, and so some overhead should be expected due to communication. This overhead should be lower with larger images, but higher with more processors, as it is essentially a function of pixels per row per processor.

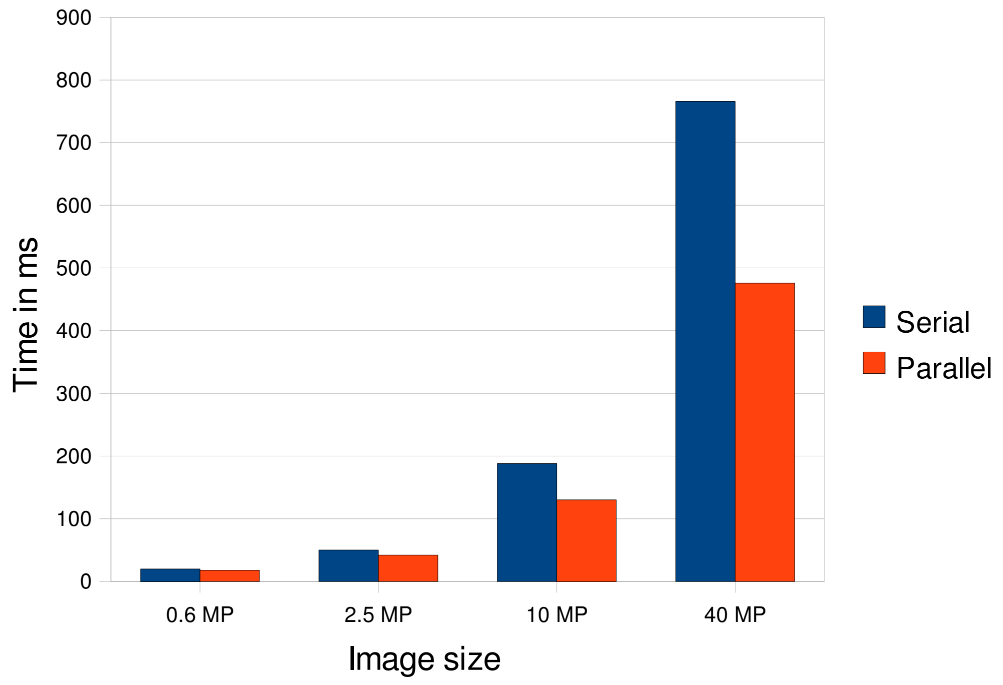
The expected speedup, therefore, for the minimum path calculation, is less than 2x, but approaching that as the size of the image increases.

## Results

Energy



Minimum Path



## Results Analysis

The results are mostly as expected; 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.

## Conclusion

The core components of the seam carving algorithm were relatively easily parallelized, and provided fairly significant improvements in computation time. As expected, communication bottlenecks limited the performance of the algorithm on smaller image sizes, but as digital cameras and scanners are released with increasingly high resolutions, and storage media continues to grow in size, there should be a trend towards larger and larger images which will make this parallel algorithm more relevant.

A final note to make regarding the minimum path algorithm is that it is not strictly necessary to recompute values for the entire image every time a seam is removed; rather, pixels that are above the downward diagonals from all pixels in the seam will not be affected by the removal of a seam. However, in images that are relatively square, at best these pixels will make up approximately half of the image, so for larger images being processed on more than two processors, the naïve recalculation of the entire map, in parallel, should still prove faster than determining which pixels require recalculation and only processing those.

## Future Work

Due to hardware limitations, I was unable to the algorithm on more than two processors; the larger machines available to me did not have the necessary image processing libraries available (pngwriter and libpng, specifically). In the future, it would be useful to test the algorithm on larger machines to verify how well it scales on systems with many more than 2 CPUs.

Additionally, it would be very useful to integrate the parallel algorithm in a plug-in for common image manipulation software such as Photoshop or The GIMP. I had intended to integrate my changes to the algorithm into a GIMP plug-in, but was unable to resolve conflicts between GIMP and the MPI libraries.

Finally, there are a few other features outlined in the original seam carving paper. The ability to remove particular features of an image, and pre-calculation of many different images sizes so that an image can be dynamically resized with minimal computation in more lightweight environments, such as a web browser or mobile phone. These would both have been very interesting to parallelize.

## References

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

Code for PNG reading (rgb_buffer_from_image) copied from liblqr (Liquid Rescale Library), an open source seam carving library. http://liblqr.wikidot.com/

## Appendix A: Serial Code

```
#include <pngwriter.h>
#include <assert.h>
#include <time.h>

using namespace std;

char *infile = NULL;

double min(double a, double b) {
   return a<b?a:b;
}

double min(double a, double b, double c) {
   return min(a,min(b,c));
}

double
compute_e (unsigned char * buf, int x, int y, int w, int h)
{
  double gx, gy;

  if (y == 0)
    {
      gy = fabs(buf[((y+1)*w+x)*3] - buf[(y*w+x)*3]) +
           fabs(buf[((y+1)*w+x)*3+1] - buf[(y*w+x)*3+1]) +
           fabs(buf[((y+1)*w+x)*3+2] - buf[(y*w+x)*3+2]);
    }
  else if (y < h - 1)
    {
      gy = (fabs(buf[((y+1)*w+x)*3] - buf[((y-1)*w+x)*3]) +
           fabs(buf[((y+1)*w+x)*3+1] - buf[((y-1)*w+x)*3+1]) +
           fabs(buf[((y+1)*w+x)*3+2] - buf[((y-1)*w+x)*3+2]))/2;
    }
  else
    {
      gy = fabs(buf[(y*w+x)*3] - buf[((y-1)*w+x)*3]) +
           fabs(buf[(y*w+x)*3+1] - buf[((y-1)*w+x)*3+1]) +
           fabs(buf[(y*w+x)*3+2] - buf[((y-1)*w+x)*3+2]);
    }

  if (x == 0)
    {
      gx = fabs(buf[(y*w+x+1)*3] - buf[(y*w+x)*3]) +
           fabs(buf[(y*w+x+1)*3+1] - buf[(y*w+x)*3+1]) +
           fabs(buf[(y*w+x+1)*3+2] - buf[(y*w+x)*3+2]);
    }
  else if (x < w - 1)
    {
      gx = (fabs(buf[(y*w+x+1)*3] - buf[(y*w+x-1)*3]) +
           fabs(buf[(y*w+x+1)*3+1] - buf[(y*w+x-1)*3+1]) +
           fabs(buf[(y*w+x+1)*3+2] - buf[(y*w+x-1)*3+2]))/2;
    }
  else
    {
      gx = fabs(buf[(y*w+x)*3] - buf[(y*w+x-1)*3]) +
           fabs(buf[(y*w+x)*3+1] - buf[(y*w+x-1)*3+1]) +
           fabs(buf[(y*w+x)*3+2] - buf[(y*w+x-1)*3+2])/2;
    }
  return (gx + gy)/2;
}


/*** MAIN ***/

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

  clock_t start, load, en, minpath;

  char * infile = argv[1];

  // open input files
  pngwriter png(1,1,0,"");
  png.readfromfile (infile);

  int w = png.getwidth ();
  int h = png.getheight ();

  /* convert the image into rgb buffers */

  unsigned char *rgb_buffer;

  start = clock();
  rgb_buffer = rgb_buffer_from_image (&png);

  load = clock();

  // generate energy map
  double * energy = (double *)malloc(sizeof(double) * w * h);

   for (int y = 0; y < h; y++) {
      for (int x = 0; x < w; x++) {
         energy[y*w+x] = compute_e(rgb_buffer, x, y, w, h);
       }
   }

   en = clock();
  // generate seams
  double * map = (double *)malloc(sizeof(double) * w * h);
   // generate first row
   for (int x = 0; x < w; x++) {
      map[x] = energy[x];
   }

   //iteratively find each successive lowest energy path per row
   for (int y = 1; y < h; y++) {
      for (int x = 0; x < w; x++) {
         if (x == 0) {
            map[y*w+x] = energy[y*w+x] + min(map[(y-1)*w+x],
map[(y-1)*w+x+1]);
         }
         else if (x == w - 1) {
            map[y*w+x] = energy[y*w+x] + min(map[(y-1)*w+x-1],
map[(y-1)*w+x]);
         }
         else {
            map[y*w+x] = energy[y*w+x] + min(map[(y-1)*w+x-1],
map[(y-1)*w+x], map[(y-1)*w+x+1]);
         }
      }
   }
   minpath = clock();

   printf("loading: %f\n", (double)(load-start)/CLOCKS_PER_SEC);
   printf("energy: %f\n", (double)(en-load)/CLOCKS_PER_SEC);
   printf("minpath: %f\n", (double)(minpath-en)/CLOCKS_PER_SEC);

  return 0;
}

/* convert the image in the right format */
unsigned char *
rgb_buffer_from_image (pngwriter * png)
{
  int x, y, k, bpp;
  int w, h;
  unsigned char *buffer;

  /* get info from the image */
  w = png->getwidth ();
```

```
  h = png->getheight ();
  bpp = 3;                      // we assume an RGB image here

  /* allocate memory to store w * h * bpp unsigned chars */
  //buffer = g_try_new (unsigned char, bpp * w * h);
  buffer = (unsigned char *)malloc(bpp * w * h);
  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 < bpp; k++)
            {
              /* read the image channel k at position x,y */
              buffer[(y * w + x) * bpp + k] =
                (unsigned char) (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;
}
```

## Appendix B: Parallelized Code

```
#include <pngwriter.h>
#include <assert.h>
#include <time.h>
#include <mpi.h>

using namespace std;

char *infile = NULL;


double min(double a, double b) {
   return a<b?a:b;
}

double min(double a, double b, double c) {
   return min(a,min(b,c));
}

double
compute_e (unsigned char * buf, int x, int y, int w, int h)
{
  double gx, gy;

  if (y == 0)
    {
      gy = fabs(buf[((y+1)*w+x)*3] - buf[(y*w+x)*3]) +
           fabs(buf[((y+1)*w+x)*3+1] - buf[(y*w+x)*3+1]) +
           fabs(buf[((y+1)*w+x)*3+2] - buf[(y*w+x)*3+2]);
    }
  else if (y < h - 1)
    {
      gy = (fabs(buf[((y+1)*w+x)*3] - buf[((y-1)*w+x)*3]) +
           fabs(buf[((y+1)*w+x)*3+1] - buf[((y-1)*w+x)*3+1]) +
           fabs(buf[((y+1)*w+x)*3+2] - buf[((y-1)*w+x)*3+2]))/2;
    }
  else
    {
      gy = fabs(buf[(y*w+x)*3] - buf[((y-1)*w+x)*3]) +
           fabs(buf[(y*w+x)*3+1] - buf[((y-1)*w+x)*3+1]) +
           fabs(buf[(y*w+x)*3+2] - buf[((y-1)*w+x)*3+2]);
    }

  if (x == 0)
    {
      gx = fabs(buf[(y*w+x+1)*3] - buf[(y*w+x)*3]) +
           fabs(buf[(y*w+x+1)*3+1] - buf[(y*w+x)*3+1]) +
           fabs(buf[(y*w+x+1)*3+2] - buf[(y*w+x)*3+2]);
    }
  else if (x < w - 1)
    {
      gx = (fabs(buf[(y*w+x+1)*3] - buf[(y*w+x-1)*3]) +
           fabs(buf[(y*w+x+1)*3+1] - buf[(y*w+x-1)*3+1]) +
           fabs(buf[(y*w+x+1)*3+2] - buf[(y*w+x-1)*3+2]))/2;
    }
  else
    {
      gx = fabs(buf[(y*w+x)*3] - buf[(y*w+x-1)*3]) +
           fabs(buf[(y*w+x)*3+1] - buf[(y*w+x-1)*3+1]) +
           fabs(buf[(y*w+x)*3+2] - buf[(y*w+x-1)*3+2])/2;
    }
  return (gx + gy)/2;
}


/*** MAIN ***/

int
main (int argc, char **argv)
{
  clock_t begin, load, en, minpath;

char * infile = argv[1];

  // open input file
  pngwriter png(1,1,0,"");
  png.readfromfile (infile);

  int w = png.getwidth ();
  int h = png.getheight ();

  /* convert the image into rgb buffer */

  unsigned char *rgb_buffer;

  int rank, n;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &n);

  MPI_Status status;
  begin = clock();
  rgb_buffer = rgb_buffer_from_image (&png);
  load = clock();

  // generate energy map
  double * energy = (double *)malloc(sizeof(double) * w * h);

   for (int y = 0; y < h; y++) {
      for (int x = rank*w/n; x < (rank+1)*w/n; x++) {
         energy[y*w+x] = compute_e(rgb_buffer, x, y, w, h);
       }
   }

   en = clock();
  // generate seams
  double * map = (double *)malloc(sizeof(double) * w * h);
   // generate first row, calculating ends of segment and
sending first
   int start = rank*w/n;
   int end = (rank+1)*w/n-1;
   map[start] = energy[start];
   map[end] = energy[end];
   if (rank > 0) {
      MPI_Bsend(&map[start], 1, MPI_DOUBLE, rank-1, 0,
MPI_COMM_WORLD);
   }
   if (rank < n - 1) {
      MPI_Bsend(&map[end], 1, MPI_DOUBLE, rank+1, 0,
MPI_COMM_WORLD);
   }
   for (int x = start + 1; x < end; x++) {
      map[x] = energy[x];
   }

   double left, right;
   //iteratively find each successive lowest energy path per row
   for (int y = 1; y < h; y++) {
      if (rank > 0) {
         MPI_Recv(&left, 1, MPI_DOUBLE, rank-1, 0,
MPI_COMM_WORLD, &status);
      }
      if (rank < n-1) {
         MPI_Recv(&right, 1, MPI_DOUBLE, rank+1, 0,
MPI_COMM_WORLD, &status);
      }
      if (rank == 0) {
         map[y*w] = energy[y*w] + min(map[(y-1)*w],
map[(y-1)*w+1]);
         map[y*w+end] = energy[y*w+end] + min(map[(y-1)*w+end-1],
                                             map[(y-1)*w+end],
                                                   right);
         if (y < h - 1)
            MPI_Bsend(&map[y*w+end], 1, MPI_DOUBLE, rank+1, 0,
MPI_COMM_WORLD);
      }
      else if (rank == n-1) {
         map[y*w+end] = energy[y*w+end] + min(map[(y-1)*w+end],
map[(y-1)*w+end-1]);
         map[y*w+start] = energy[y*w+start] + min(left,
                                                  map[(y-1)*w+start],
                                                  map[(y-1)*w+start+
1]);
         if (y < h - 1)
            MPI_Bsend(&map[y*w+start], 1, MPI_DOUBLE, rank-1, 0,
MPI_COMM_WORLD);
      }
      else {
         map[y*w+start] = energy[y*w+start] + min(left,
                                                  map[(y-1)*w+start],
                                                  map[(y-1)*w+start+
1]);
         map[y*w+end] = energy[y*w+end] + min(map[(y-1)*w+end-1],
                                                  map[(y-1)*w+end],
                                                  right);
         if (y < h - 1) {
            MPI_Bsend(&map[y*w+end], 1, MPI_DOUBLE, rank+1, 0,
MPI_COMM_WORLD);
            MPI_Bsend(&map[y*w+start], 1, MPI_DOUBLE, rank-1, 0,
MPI_COMM_WORLD);
         }
      }

      for (int x = start+1; x < end-1; x++) {
         map[y*w+x] = energy[y*w+x] + min(map[(y-1)*w+x-1],
map[(y-1)*w+x], map[(y-1)*w+x+1]);
      }
   }
   minpath = clock();

   printf("loading: %f\n", (double)(load-begin)/CLOCKS_PER_SEC);
   printf("energy: %f\n", (double)(en-load)/CLOCKS_PER_SEC);
   printf("minpath: %f\n", (double)(minpath-en)/CLOCKS_PER_SEC);

   MPI_Finalize();

  return 0;
}
```

```
/* convert the image in the right format */
unsigned char *
rgb_buffer_from_image (pngwriter * png)
{
  int x, y, k, bpp;
  int w, h;
  unsigned char *buffer;

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

  /* allocate memory to store w * h * bpp unsigned chars */
  //buffer = g_try_new (unsigned char, bpp * w * h);
  buffer = (unsigned char *)malloc(bpp * w * h);
  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 < bpp; k++)
            {
              /* read the image channel k at position x,y */
              buffer[(y * w + x) * bpp + k] =
                (unsigned char) (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;
}
```