Image Stitching using Parallel programming Techniques

Manoj Verma
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
E-mails: manojverma@oakland.edu

***Abstract*— In this project, I will develop algorithms for stitching and blending photographs using pthreads or TBB programming on a multicore computer learnt in this semester. An image stitch is the process of combining several individual images that overlap in order to form a composite image. Using Multicore processing system and parallel programming techniques allows us to take advantage of the parallel processing capabilities. Using TBB libraries, we can write parallel programs that are capable of running on intel multicore CPUs in order to significantly speed up certain computational tasks. The stitching of images is a method of computer vision that involves stitching together multiple overlapping images to create a larger panorama in a seamless manner. It is commonly used for the creation of wide-angle photos and the generation of 360-degree images.**

I. Introduction

Image Stitching is the process of merging multiple images into a single high-resolution or panoramic image. An image blending technique is used in compute vision and image processing fields to create seamless images. The aim of this project to implement image stitching and image blending algorithm using parallel programming techniques. The focus of this project based on the parallel programming knowledge that I gained during this semester.

While my research I found that there are many advance algorithms for both image stitching and blending, however I am focusing in developing an image stitching approach where I could apply the parallel programming techniques to optimize the performance and maximize the throughput.

II. Image Blending

The human visual system is highly sensitive to changes in brightness, which helps to discern different visual artifacts. It is quite easy for our visual system to distinguish two different images if we just stitch the image without applying any blending, as shown in the Figure-1, it has no homogenous blending.

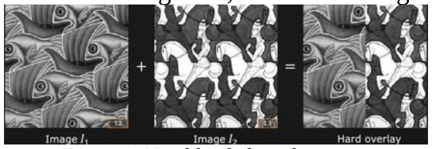


Figure 1: Non blended stitching

I will develop a blending function to make stitching more homogenous.

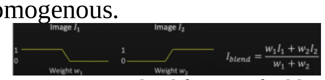


Figure 2: Weighted function for blending.

III. Implemantaion and Algorith Concept

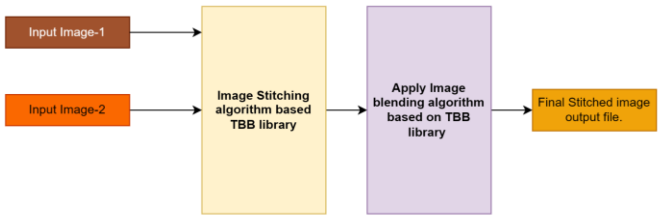
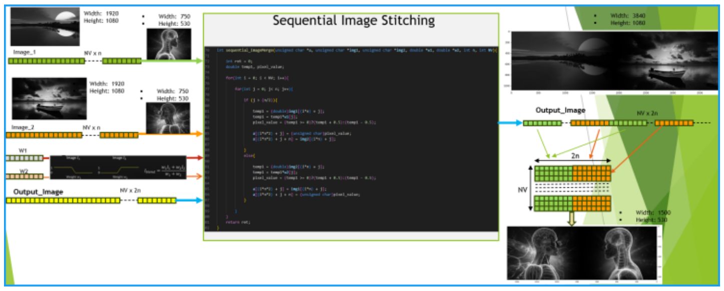

Input Image-1
Input Image-2
Image Stitching algorithm based TBB library
Apply Image blending algorithm based on TBB library
Final Stitched image output file.

Figure- 3: Block Diagram

IV. Image Stitching – Sequential Method

Figure-4 illustrates the sequential method. The function is based on nested "for-loops". There is an outer loop that accesses the Row element of both images and an inner loop that processes the Column element. One potential inefficiency of this method is that it can be computationally intensive and slow, especially for large images, due to the repeated access of each pixel in a nested manner. Additionally, this method does not take advantage of parallel processing capabilities, which could significantly speed up the operation by processing multiple rows or columns simultaneously. As a result, the sequential approach may not be optimal for applications requiring real-time image processing..

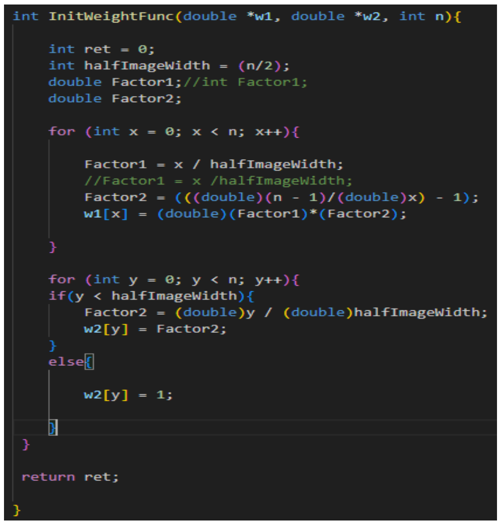


```
int sequential_imageMerge(unsigned char *a, unsigned char *img1, unsigned char *img2, double *w1, double *w2, int n, int NV){

    int ret = 0;
    double temp1, pixel_value;

    for(int i = 0; i < NV; i++){

        for(int j = 0; j< n; j++){

            if (j > (n/2)){

                temp1 = (double)img1[(i*n) + j];
                temp1 = temp1*w1[j];
                pixel_value = (temp1 >= 0)?(temp1 + 0.5):(temp1 - 0.5);

                a[(i*n*2) + j] = (unsigned char)pixel_value;
                a[(i*n*2) + j + n] = img2[(i*n) + j];

            }
            else{

                temp1 = (double)img2[(i*n) + j];
                temp1 = temp1*w2[j];
                pixel_value = (temp1 >= 0)?(temp1 + 0.5):(temp1 - 0.5);

                a[(i*n*2) + j] = img1[(i*n) + j];
                a[(i*n*2) + j + n] = (unsigned char)pixel_value;
            }

        }
    }
    return ret;
}
```

Figure- 4: Sequential Processing - Block Diagram

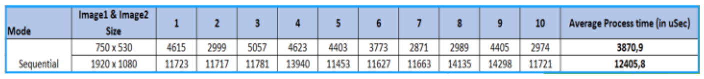
<u>**Steps**</u>:

- There are two grayscale images of same resolutions that are read and stored in two dynamically allocated memory sections.
- Update the weight function vectors w1 and w2 for Images 1 and 2.



```
int InitWeightFunc(double *w1, double *w2, int n){

    int ret = 0;
    int halfImageWidth = (n/2);
    double Factor1;//int Factor1;
    double Factor2;

    for (int x = 0; x < n; x++){

        Factor1 = x / halfImageWidth;
        //Factor1 = x /halfImageWidth;
        Factor2 = (((double)(n - 1)/(double)x) - 1);
        w1[x] = (double)(Factor1)*(Factor2);

    }

    for (int y = 0; y < n; y++){
    if(y < halfImageWidth){
        Factor2 = (double)y / (double)halfImageWidth;
        w2[y] = Factor2;
    }
    else{

        w2[y] = 1;

    }
  }

  return ret;

}
```

- Create dynamic memory sections to store the processed image.
- Access every pixel in the nested for-loop, multiply the respective weight function elements store in the vectors W1 and W2.
- Write the final processed image into a file.
- De-allocate the dynamic allocated memories.
- In this example, I have taken two examples and processed them sequentially. Below table shows the processing time for image sizes of 750x 530 and 1920 x 1920. The significant difference in processing times between the two image sizes can be attributed to the increased number of pixels in the larger image. Larger images contain more data, which requires more computational resources and time to process. Understanding these differences is crucial for optimizing performance and resource allocation in image processing tasks.



Mode	Image1 & Image2 Size	1	2	3	4	5	6	7	8	9	10	Average Process time (in uSec)
Sequential	750 x 530	4615	2999	5057	4623	4403	3773	2871	2989	4405	2974	**3870,9**
	1920 x 1080	11723	11717	11781	13940	11453	11627	11663	14135	14298	11721	**12405,8**

Figure- 5: Sequential Processing time measurement

V. IMAGE STITCHING – TBB PARALLEL PIPELINE

Using TBB parallel pipeline, we can optimize the processing time. This TBB pipeline implementation is divided into three stages. The details of each stages are given below.

- Read two grayscale images and store into two dynamically allocated memory sections.

```
char **inputImage1 = (char **)calloc((n*NV), sizeof(char *));
char **inputImage2 = (char **)calloc((n*NV), sizeof(char *));
double *w1 = (double*)calloc(n, sizeof(double));
double *w2 =(double *)calloc(n, sizeof(double));
NV = Number of Rows in the images.
n = Number of the pixel in each row.
Read Two image here and store into the above created dynamic memories.
```

- Update weight function vectors w1 and w2 for Image1 and Image2.

```
int image_hw = (n/2);
int factor1;
double factor2;
for (int x = 0; x<n; x++){
        factor1 = (i/image_hw);
    factor2 = (((n-1)/(double)x) – 1);
    w1[x] = factor2
    w2[x] = (double)x/(double) Image_hw;
}
```

- Create two dynamic memory section to store the intermediate processed image and final stitched output image.
  - Width will twice of the input image.

```
double **intermidiate_image = (double **)calloc((2*n*NV), sizeof(double *));
char **stitched_image = (char **)calloc((2*n*NV), sizeof(char *));
```

- Below Function will invoke the parallel_pipeline and perform the parallel operation.
- Basically, it will have three parallel pipelines.
- The function “ImageStitchPipeline” will invoke all the three parallel pileline.

```
void ImageStitchPipeline (int ntoken, int n, int NV, double **a, double **r, double *c) {
parallel_pipeline(ntoken, make_filter<void, MyPixelPair>(filter::serial_in_order,
Input_myImages(inputImage1, inputImage2, w1, w2, stitch_image, n, NV))
    &  make_filter<MyPixelPair, double *>(filter::parallel,
  PixelWeight_Calculator ())
    &  make_filter<double*, void>(filter::serial_in_order,
  My_StitchedImage(c, n)));
  }
```

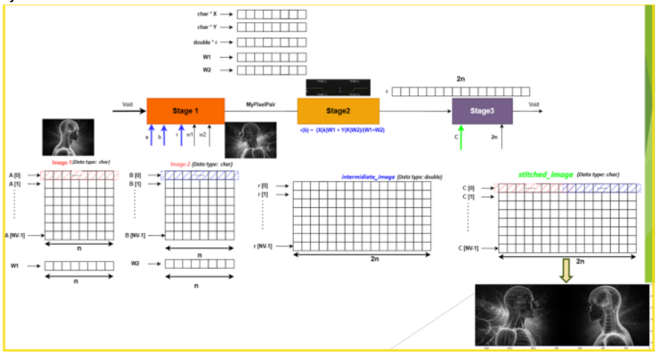


Figure- 6: Demonstration of pipeline and the operations at each stage.

- **Stage-1:**
  - The purpose of stage-1 to reads the rows from Image-1 and Image-2 and create a two n-element vector for the next stage.
  - Below code snapshot realizes the first stage of the parallel pipeline. The parameter passed this first stage are:
    - Two Images (two 2D data arrays – 2 NV x n matrices).
    - Two Vectors (Both vectors will have n elements)
    - Intermediate 2D array (NV x 2n)
    - Value of **n** (Elements in each vector)
    - Value of **NV** (total no. of rows in the images)

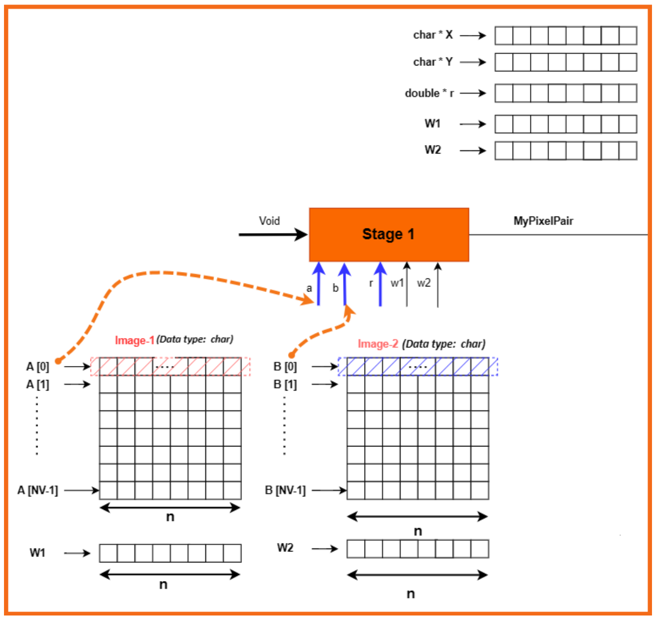


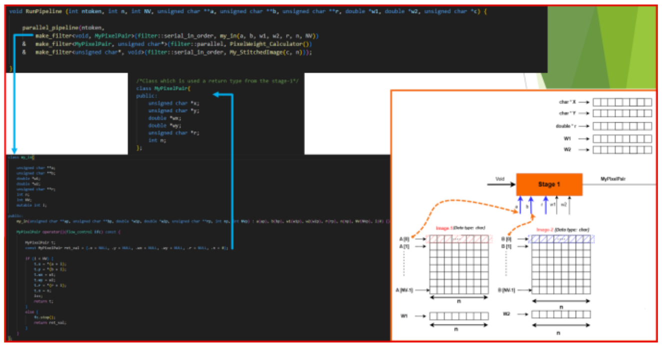
Figure- 7: Stage-1 of the three stage TBB parallel_pipeline

- ***Stage-1 class definition and Functor***.

```
class Input_myImages{
char **I1;
char **I2;
double *w1
double *w2
double **r;
 int n;
int NV;

mutable int i;
public:
Input_myImages (char **I1p, char **I2p, double *w1p, double *w2p,
double **rp, int np, int NVp) : I1(I1p), I2(I2p), w1(w1p), w2(w2p), r(rp),
n(np), NV(NVp), i(0) {}

MyPixelPair operator ()(flow_control &fc) const {
 MyPixelPair t;
 const MyPixelPair ret_val = {.x = NULL, .y = NULL, .xw = NULL, .yw = NULL,
.r = NULL, .n = 0};
 if (i < NV) {
        t.x = *(I1 + i);
        t.y = *(I2 + i);
        t.xw = w1 + i;
       t.yw = w2 + i;
       t.r = *(r + i);
       t.n = n;
       i++;
       return t;
            }
            else {
              fc.stop();
              return ret_val;
           }
       }
  };
```

<u>MyPixelPair class:</u>

```
Class MyPixelPair {

Public:
char *x;     // A n-element vector from image-1.
char *y;     // A n-element vector from image-2.
double *r;  // A n-element vector from intermediate image.
double w1; // Image-1 weight vector.
double w2; // Image-2 weight vector.
int n;          // Total element into each vector.
}
```

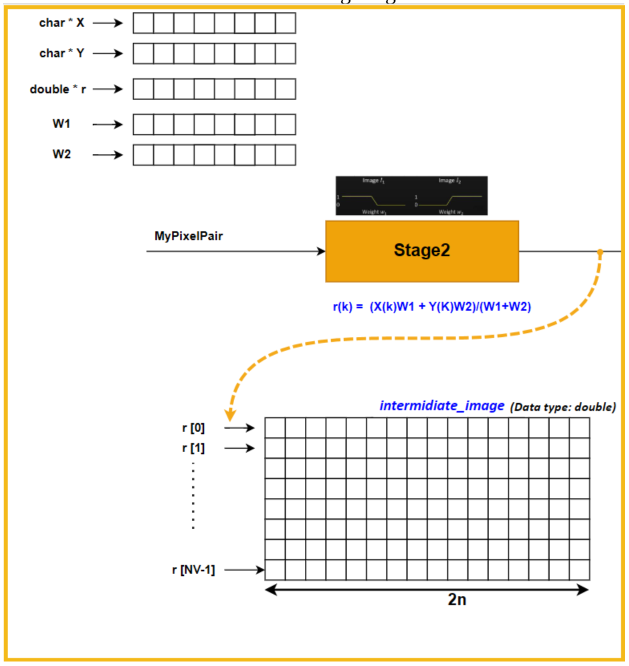


```
void RunPipeline (int ntoken, int n, int NV, unsigned char **a, unsigned char **b, unsigned char **r, double *w1, double *w2, unsigned char *c) {

    parallel_pipeline(ntoken,
        make_filter<void, MyPixelPair>(filter::serial_in_order, my_in(a, b, w1, w2, r, n, NV))
    &   make_filter<MyPixelPair, unsigned char*>(filter::parallel, PixelWeight_Calculator())
    &   make_filter<unsigned char*, void>(filter::serial_in_order, My_StitchedImage(c, n)));

}
```

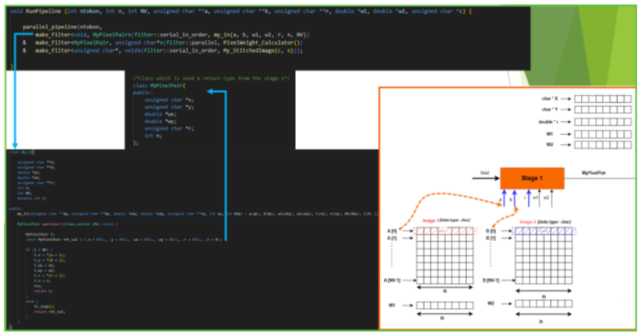
```
/*Class which is used a return type from the stage-1*/
class MyPixelPair{
public:
    unsigned char *x;
    unsigned char *y;
    double *wx;
    double *wy;
    unsigned char *r;
    int n;
};
```

```
class my_in{

    unsigned char **a;
    unsigned char **b;
    double *w1;
    double *w2;
    unsigned char **r;
    int n;
    int NV;
    mutable int i;

public:
    my_in(unsigned char **ap, unsigned char **bp, double *w1p, double *w2p, unsigned char **rp, int np, int NVp) : a(ap), b([illegible]

    MyPixelPair operator()(flow_control &fc) const {

        MyPixelPair t;
        const MyPixelPair ret_val = {.x = NULL, .y = NULL, .wx = NULL, .wy = NULL, .r = NULL, .n = 0};

        if (i < NV) {
            t.x = *(a + i);
            t.y = *(b + i);
            t.wx = w1;
            t.wy = w2;
            t.r = *(r + i);
            t.n = n;
            i++;
            return t;
        }
        else {
            fc.stop();
            return ret_val;
        }
    }
};
```

- **<u>Stage-2:</u>**
  - The output of the stage-1 will be vector of data type "MyPixelPair". It will apply weight coefficient on each pixel of the image. No parameters passed to the functor. It is configured to run in **parallel**, so that items can be processed concurrently. The Functor of this stage is given below:



```
class PixelWeight_Calculator{
        public:
        double * operator() (MyPixelPair input) const{

        int i;
        double *result = input.r;
        for(i = 0; i < input.n; i++){

          result[i] = input.w1[0]*input.x[i];        // Apply weight coefficient on image-
        1 pixels.
           result[i + n] = input.w2[0]*input.y[i]; // Apply weight coefficient on image-
        2 pixels.

         }

         return result;
        }

        };
```

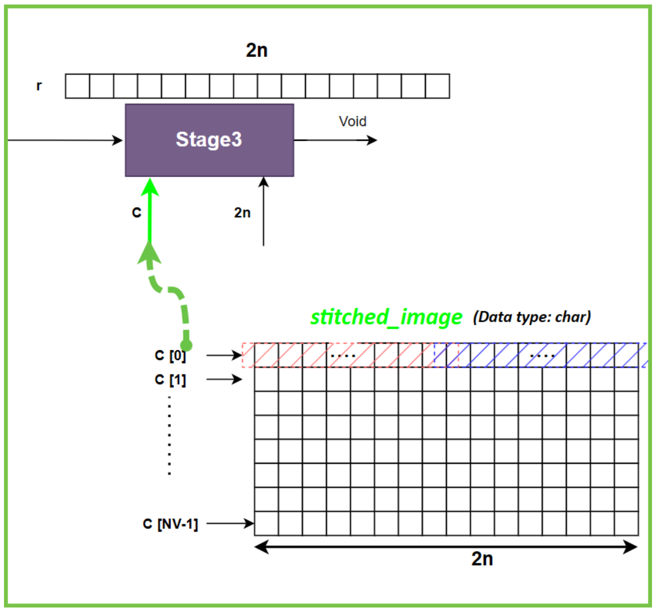


```
void RunPipeline (int ntoken, int n, int NV, unsigned char **a, unsigned char **b, unsigned char **r, double *w1, double *w2, unsigned char *c) {

    parallel_pipeline(ntoken,
        make_filter<void, MyPixelPair>(filter::serial_in_order, my_in(a, b, w1, w2, r, n, NV))
    &   make_filter<MyPixelPair, unsigned char*>(filter::parallel, PixelWeight_Calculator())
    &   make_filter<unsigned char*, void>(filter::serial_in_order, My_StitchedImage(c, n)));

}
```

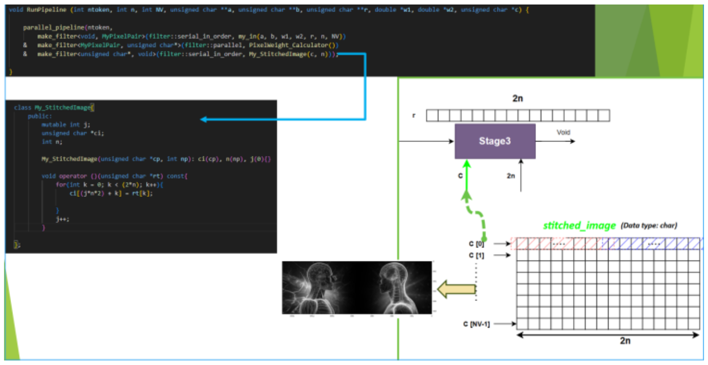
```
/*Class which is used a return type from the stage-1*/
class MyPixelPair{
public:
    unsigned char *x;
    unsigned char *y;
    double *wx;
    double *wy;
    unsigned char *r;
    int n;
};
```

```
class my_in{

    unsigned char **a;
    unsigned char **b;
    double *w1;
    double *w2;
    unsigned char **r;
    int n;
    int NV;
    mutable int i;

public:
    my_in(unsigned char **ap, unsigned char **bp, double *w1p, double *w2p, unsigned char **rp, int np, int NVp) : a(ap), b(bp), w1(w1p), w2(w2p), r(rp), n(np), NV(NVp), i(0) {}

    MyPixelPair operator()(flow_control &fc) const {

        MyPixelPair t;
        const MyPixelPair ret_val = {.x = NULL, .y = NULL, .wx = NULL, .wy = NULL, .r = NULL, .n = 0};

        if (i < NV) {
            t.x = *(a + i);
            t.y = *(b + i);
            t.wx = w1;
            t.wy = w2;
            t.r = *(r + i);
            t.n = n;
            i++;
            return t;
        }
        else {
            fc.stop();
            return ret_val;
        }
    }
};
```

- **Stage-3:**
    - This stage performs rounding and saturation of each pixel and store the data into output 2D array. Syntax-wise, the stage has no outputs, but this stage places the result in the array c (provided as an input parameter to its functor).
    - Here the total number of elements in the intermediate vector will be 2n. So below for loop need to process for all the 2n element and store into the final 2D array.



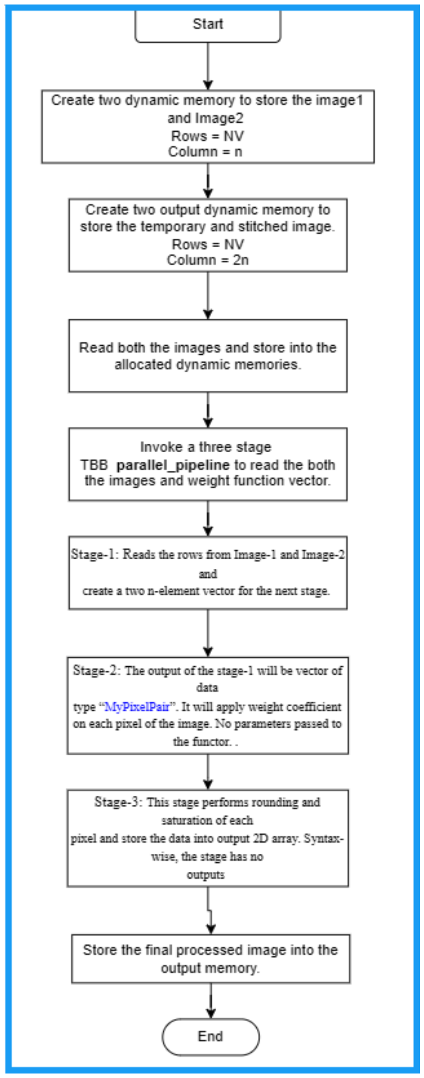
```
class My_StitchedImage {

    public:
       mutable int j;
       double *ci;
       int n;

       My_StitchedImage (double *cp, int np): ci(cp), n (np), j(0){}

       void operator () (double *rt) const{

          int k;

    double tmp;

          for (k = 0; k < 2n; k++ ) {

             if ( rt[k] > = 0) {

                tmp = rt[k] + 0.5;

             }
             else
                       tmp = rt[k] – 0.5;
             }
             ci[k + j] = tmp; // Update the final buffer. This buffer will have the stitched image.

             j++;
          }
       };
```



```
void RunPipeline (int ntoken, int n, int NV, unsigned char **a, unsigned char **b, unsigned char **r, double *w1, double *w2, unsigned char *c) {

    parallel_pipeline(ntoken,
        make_filter<void, MyPixelPair>(filter::serial_in_order, my_in(a, b, w1, w2, r, n, NV))
    &   make_filter<MyPixelPair, unsigned char*>(filter::parallel, PixelWeight_Calculator())
    &   make_filter<unsigned char*, void>(filter::serial_in_order, My_StitchedImage(c, n)));

}
```

```
class My_StitchedImage{
    public:
        mutable int j;
        unsigned char *ci;
        int n;

        My_StitchedImage(unsigned char *cp, int np): ci(cp), n(np), j(0){}

        void operator ()(unsigned char *rt) const{
            for(int k = 0; k < (2*n); k++){
                ci[(j*n*2) + k] = rt[k];

            }
            j++;
        }

};
```

- High-level flowchart is given here.



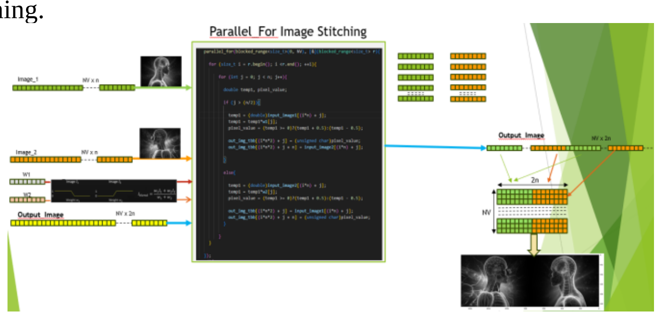
Start
Create two dynamic memory to store the image1 and Image2 Rows = NV Column = n
Create two output dynamic memory to store the temporary and stitched image. Rows = NV Column = 2n
Read both the images and store into the allocated dynamic memories.
Invoke a three stage TBB parallel_pipeline to read the both the images and weight function vector.
Stage-1: Reads the rows from Image-1 and Image-2 and create a two n-element vector for the next stage.
Stage-2: The output of the stage-1 will be vector of data type "MyPixelPair". It will apply weight coefficient on each pixel of the image. No parameters passed to the functor. .
Stage-3: This stage performs rounding and saturation of each pixel and store the data into output 2D array. Syntax-wise, the stage has no outputs
Store the final processed image into the output memory.
End

- I have taken two examples and processed them using parallel pipeline methods. Below are the time measurements. The following table illustrates the processing time for images with sizes of 750 x 530 and 1920 x 1920 respectively.

Mode	Image1 & Image2 Size	1	2	3	4	5	6	7	8	9	10	Average Process time (in uSec)
TBB Pipeline	750 x 530	5404	5040	5420	4819	5004	5290	5423	5334	4776	5459	5196,9
(2D Array)	1920 x 1080	19997	19878	19645	18806	19951	19996	20468	20107	20550	19408	19880,6

VI. Image Stitching – Parallel_For Techniques

I implemented the same stitching algorithm as TBB Parallel For in order to evaluate its performance. The use of TBB parallel provides significant advantages, including improved parallelism and scalability. Compared to other parallel algorithms, TBB Parallel For offers more efficient workload distribution and dynamic scheduling, which can lead to better performance on multi-core processors. Unlike basic thread-based approaches, TBB handles thread management automatically, reducing the complexity for developers. Additionally, it often results in better cache utilization and reduced overhead, making it a preferred choice for optimizing performance in complex applications. However, TBB Parallel For may not always be the best choice for every application. It can introduce overhead when the workload is too small or when the task granularity is not appropriately set. Furthermore, TBB's abstraction might limit fine-grained control over thread management, which can be crucial for certain specialized applications needing precise tuning.

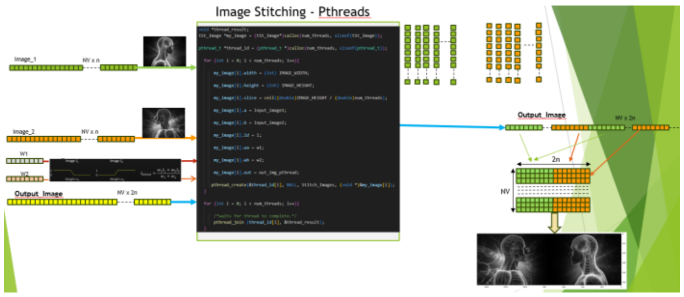


```
parallel_for(blocked_range<size_t>(0, NV), [&](blocked_range<size_t> r){

  for (size_t i = r.begin(); i <r.end(); ++i){

    for (int j = 0; j < n; j++){

      double temp1, pixel_value;

      if (j > (n/2)){

        temp1 = (double)input_Image1[(i*n) + j];
        temp1 = temp1*w1[j];
        pixel_value = (temp1 >= 0)?(temp1 + 0.5):(temp1 - 0.5);

        out_img_tbb[(i*n*2) + j] = (unsigned char)pixel_value;
        out_img_tbb[(i*n*2) + j + n] = input_Image2[(i*n) + j];

      }

      else{

        temp1 = (double)input_Image2[(i*n) + j];
        temp1 = temp1*w2[j];
        pixel_value = (temp1 >= 0)?(temp1 + 0.5):(temp1 - 0.5);

        out_img_tbb[(i*n*2) + j] = input_Image1[(i*n) + j];
        out_img_tbb[(i*n*2) + j + n] = (unsigned char)pixel_value;
      }

    }
  }

});
```

Mode	Image1 & Image2 Size	1	2	3	4	5	6	7	8	9	10	Average Process time (in uSec)
TBB Parallel_For	750 x 530	2252	1737	1954	2096	1938	2359	2003	2251	2443	2034	2106,7
(Row Parallel)	1920 x 1080	3641	5552	4116	3615	3919	3650	7011	3750	5325	4594	4517,3

VII. Image Stitching – Pthreads Techniques

I implemented the same stitching algorithm using pthreads in order to evaluate its performance. In the implementation, the user is able to enter the number of threads that can be used to process each image. Using pthreads provides significant advantages, including improved parallelism and scalability. This allows for better utilization of multi-core processors, leading to faster image processing times. Additionally, it offers a flexible and efficient way to manage thread workloads, enhancing overall performance.

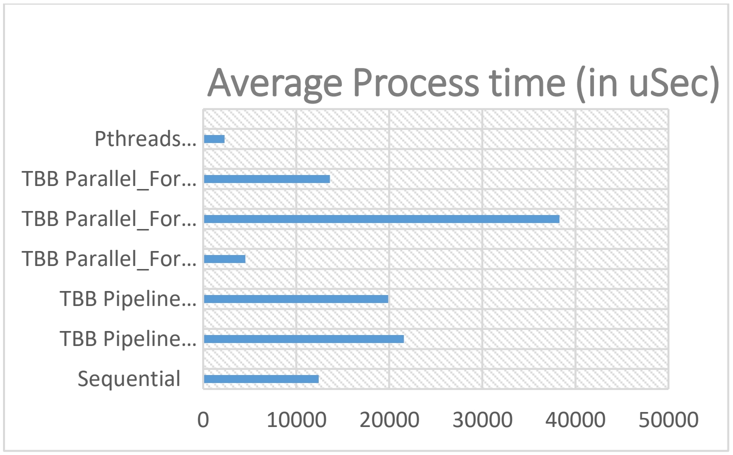


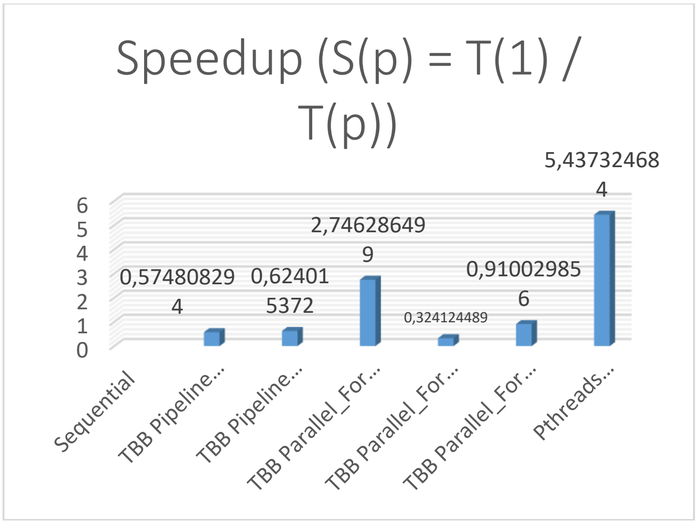
```
void *thread_result;
tSt_Image *my_Image = (tSt_Image*)calloc(num_threads, sizeof(tSt_Image));

pthread_t *thread_id = (pthread_t *)calloc(num_threads, sizeof(pthread_t));

  for (int i = 0; i < num_threads; i++){

      my_Image[i].width = (int) IMAGE_WIDTH;

      my_Image[i].height = (int) IMAGE_HEIGHT;

      my_Image[i].slice = ceil((double)IMAGE_HEIGHT / (double)num_threads);

      my_Image[i].a = input_Image1;

      my_Image[i].b = input_Image2;

      my_Image[i].id = i;

      my_Image[i].wa = w1;

      my_Image[i].wb = w2;

      my_Image[i].out = out_img_pthread;

     pthread_create(&thread_id[i], NULL, Stitch_Images, (void *)&my_Image[i]);
  }

  for (int i = 0; i < num_threads; i++){

      /*waits for thread to complete.*/
      pthread_join (thread_id[i], &thread_result);
  }
```

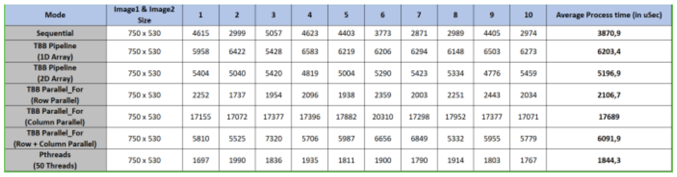
VIII. Processing Time and Speedup

An analysis of the performance of image stitching algorithms using different parallel programming techniques is provided in the following table. The results indicate that parallel programming significantly enhances the efficiency and speed of image stitching algorithms.

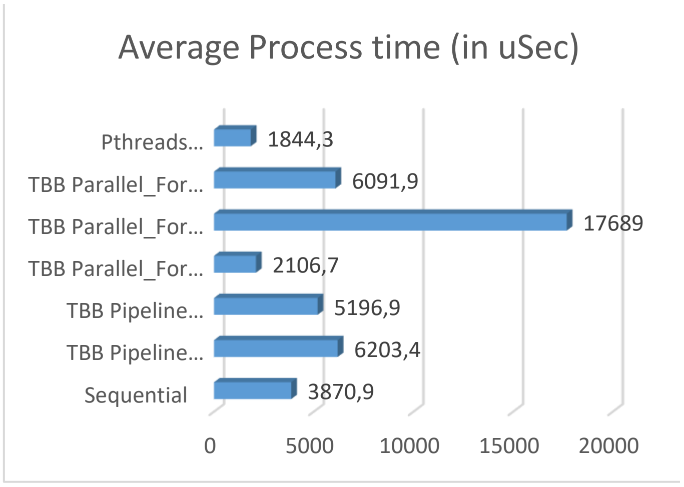
Mode	Image1 & Image2 Size	1	2	3	4	5	6	7	8	9	10	Average Process time (in uSec)	Speedup (S(p) = T(1) / T(p))
Sequential	1920 x 1080	11723	11717	11781	13940	11453	11627	11663	14135	14298	11721	12405,8	
TBB Pipeline (1D Array)	1920 x 1080	21739	21950	21690	21947	21362	21693	21310	21548	21292	21294	21582,5	0,574808294
TBB Pipeline (2D Array)	1920 x 1080	19997	19878	19645	18806	19951	19996	20468	20107	20550	19408	19880,6	1,085606068
TBB Parallel_For (Row Parallel)	1920 x 1080	3641	5552	4116	3615	3919	3650	7011	3750	5325	4594	4517,3	4,400991743
TBB Parallel_For (Column Parallel)	1920 x 1080	36797	38772	37052	38832	39951	39516	37033	36404	39229	39162	38274,8	0,118022824
TBB Parallel_For (Row + Column Parallel)	1920 x 1080	13280	14179	14041	13366	13800	13428	13806	13369	14067	12987	13632,3	2,807655348
Pthreads (50 Threads)	1920 x 1080	2174	2285	2088	2083	2403	2467	2224	2314	2622	2156	2281,6	5,974886045

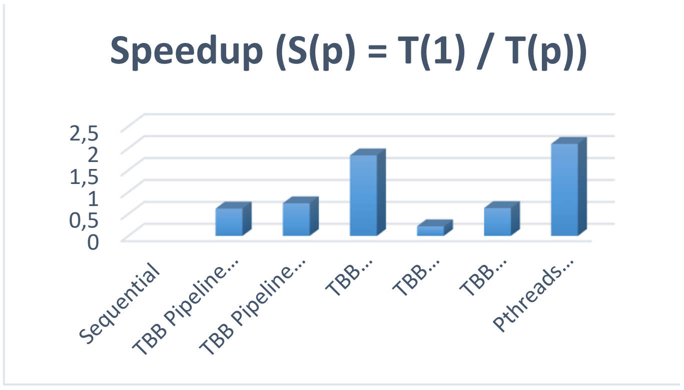






Mode	Image1 & Image2 Size	1	2	3	4	5	6	7	8	9	10	Average Process time (in uSec)
Sequential	750 x 530	4615	2999	5057	4623	4403	3773	2871	2989	4405	2974	**3870,9**
TBB Pipeline (1D Array)	750 x 530	5958	6422	5428	6583	6219	6206	6294	6148	6503	6273	**6203,4**
TBB Pipeline (2D Array)	750 x 530	5404	5040	5420	4819	5004	5290	5423	5334	4776	5459	**5196,9**
TBB Parallel_For (Row Parallel)	750 x 530	2252	1737	1954	2096	1938	2359	2003	2251	2443	2034	**2106,7**
TBB Parallel_For (Column Parallel)	750 x 530	17155	17072	17377	17396	17882	20310	17298	17952	17377	17071	**17689**
TBB Parallel_For (Row + Column Parallel)	750 x 530	5810	5525	7320	5706	5987	6656	6849	5332	5955	5779	**6091,9**
Pthreads (50 Threads)	750 x 530	1697	1990	1836	1935	1811	1900	1790	1914	1803	1767	**1844,3**





}

IX. Conclusion

- For seamless output, stitching images is a very practical and very useful application that requires a large amount of computing power. Recent advancements in Parallel programming techniques have significantly reduced computational speed. These improvements allow for more efficient processing of high-resolution images, enabling stunning panoramas and 3D models.

- Parallel programming techniques are most commonly used in multi-core CPU-based hardware accelerators designed for such SIMD-based computations, and they can fulfill many of the computing requirements for image processing. Parallel programming allows for faster processing times by distributing tasks across multiple cores, enabling simultaneous execution of computations. This results in improved performance and efficiency, particularly when handling large datasets and complex algorithms in image processing. Additionally, it can enhance the accuracy and quality of image analysis by enabling more detailed and comprehensive computations.

- Our knowledge of High-performance Embedded Programming has been instrumental in the development of this project. By optimizing code for efficiency and speed, we significantly enhanced the system's performance. This expertise allowed us to minimize resource consumption, ensuring the software could run smoothly on same CPU hardware.

X. References

- https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2021-6/onetbb-developer-guide.html
- https://en.wikipedia.org/wiki/POSIX
- https://en.wikipedia.org/wiki/Image_stitching
- https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html
- https://neptune.ai/blog/image-processing-python
- https://courses.cs.washington.edu/courses/cse576/05sp/papers/MSR-TR-2004-92.pdf
- https://github.com/natandrade/Tutorial-Medical-Image-Registration
- https://en.wikipedia.org/wiki/Image_stitching
- https://www.cmor-faculty.rice.edu/~zhang/caam699/p-files/Im-Align2005.pdf