# ECPS 204 – ASSIGNMENT 2

# GOALS

- Implementation of threads using different programming methods.
- Getting familiar with OpenMP and pthreads.
- Comparing the performance improvement with parallelization in terms of time.
- Comparing these two techniques.

# CHECKING THE PERFORMANCE OF THE REGULAR CODE

- To compile the code:
  - $gcc canny_local_c –lm –o ./just_canny
- To run the code and get the time information:
  - $time ./just_canny test.pgm 1.0 1.0 0.8
- This will yield couple of results and real x will give you the execution time.
- The acquired time will be your reference for future trials.

# OPENMP CODE

- We will modify inside of the gaussian_smooth function.
- Inside the function there are Blur in x-direction block and Blur in y-direction block.
- Blur in x-direction is already modified for you, your goal is to modify Blur in y-direction.
  - The modifications takes place inside the gaussian_smooth

```
/*******************************************************************************
* PROCEDURE: gaussian_smooth
* PURPOSE: Blur an image with a gaussian filter.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void gaussian_smooth(unsigned char *image, int rows, int cols, float sigma,
        short int **smoothedim)
{
    int r, c, rr, cc,     /* Counter variables. */
       windowsize,        /* Dimension of the gaussian kernel. */
       center;            /* Half of the windowsize. */
    float *tempim,        /* Buffer for separable filter gaussian smoothing. */
          *kernel,        /* A one dimensional gaussian kernel. */
          dot,            /* Dot product summing variable. */
          sum;            /* Sum of the kernel weights variable. */

    /****************************************************************************
    * Create a 1-dimensional gaussian smoothing kernel.
    ****************************************************************************/
    if(VERBOSE) printf("   Computing the gaussian smoothing kernel.\n");
    make_gaussian_kernel(sigma, &kernel, &windowsize);
    center = windowsize / 2;

    /****************************************************************************
    * Allocate a temporary buffer image and the smoothed image.
    ****************************************************************************/
```

```
/*******************************************************************************
* Blur in the x - direction.
*******************************************************************************/
```

```
/*******************************************************************************
* Blur in the y - direction.
*******************************************************************************/
```

# OPENMP CODE

- First, don't forget to include the library at the beginning as:
  - `#include <omp.h>`
- `n_thread` → number of threads (fixed to 4)
- `omp_set_dynamic(0)` → disabling the dynamic threads which are used to optimize system resources if they are set accordingly.

```
/*******************************************************************************
* Blur in the x - direction.
*******************************************************************************/

int n_thread = 4;                       //Edited by Jiang Wan for multi-thread
omp_set_dynamic(0);                     //Edited by Jiang Wan for multi-thread
omp_set_num_threads(n_thread);          //Edited by Jiang Wan for multi-thread
#pragma omp parallel private(c, cc, dot, sum)    //Edited by Jiang Wan for multi-thread
{
   printf("X Direction Thread ID = %d\n", omp_get_thread_num());//This is for you to see the
   //parallelization between the threads. Delete this after observing.
   //Edited by Ahmet Aksakal

   if(VERBOSE) printf("   Bluring the image in the X-direction.\n");
   #pragma omp for
   for(r=0;r<rows;r++){
      for(c=0;c<cols;c++){
     dot = 0.0;
         sum = 0.0;
         for(cc=(-center);cc<=center;cc++){
            if(((c+cc) >= 0) && ((c+cc) < cols)){
               dot += (float)image[r*cols+(c+cc)] * kernel[center+cc];
               sum += kernel[center+cc];
            }
         }
         tempim[r*cols+c] = dot/sum;
      }
   }
}
```

# OPENMP CODE

- `omp_set_num_threads()` → since dynamic is disabled, we are fixing the # of threads.
- `#pragma omp` → the pragma to call omp
- `#pragma omp parallel` → specifies that a structured block should be run in parallel by a team of threads.
- `#pragma omp parallel private()` → These are the values that should be private between threads. So each thread should have their own copies of these variables.

```
int n_thread = 4;                       //Edited by Jiang Wan for multi-thread
omp_set_dynamic(0);                     //Edited by Jiang Wan for multi-thread
omp_set_num_threads(n_thread);          //Edited by Jiang Wan for multi-thread
#pragma omp parallel private(c, cc, dot, sum)  //Edited by Jiang Wan for multi-thread
{
   printf("X Direction Thread ID = %d\n", omp_get_thread_num());//This is for you to see the
   //parallelization between the threads. Delete this after observing.
   //Edited by Ahmet Aksakal

   if(VERBOSE) printf("   Bluring the image in the X-direction.\n");
   #pragma omp for
   for(r=0;r<rows;r++){
      for(c=0;c<cols;c++){
     dot = 0.0;
         sum = 0.0;
         for(cc=(-center);cc<=center;cc++){
            if(((c+cc) >= 0) && ((c+cc) < cols)){
               dot += (float)image[r*cols+(c+cc)] * kernel[center+cc];
               sum += kernel[center+cc];
            }
         }
         tempim[r*cols+c] = dot/sum;
      }
   }
}
```

# OPENMP CODE

- `#pragma omp for` → Specifies a parallel loop. Each iteration of the loop is executed by one of the threads in the team.
- `omp_get_thread_num ()` → you can get the thread number with the following command. Each thread is given a number, so you can check your coding.
- Don't forget to free the memory at the end.

```
int n_thread = 4;                       //Edited by Jiang Wan for multi-thread
omp_set_dynamic(0);                     //Edited by Jiang Wan for multi-thread
omp_set_num_threads(n_thread);              //Edited by Jiang Wan for multi-thread
#pragma omp parallel private(c, cc, dot, sum)     //Edited by Jiang Wan for multi-thread
{
   printf("X Direction Thread ID = %d\n", omp_get_thread_num());//This is for you to see the
   //parallelization between the threads. Delete this after observing.
   //Edited by Ahmet Aksakal

   if(VERBOSE) printf("   Bluring the image in the X-direction.\n");
   #pragma omp for
   for(r=0;r<rows;r++){
      for(c=0;c<cols;c++){
     dot = 0.0;
         sum = 0.0;
         for(cc=(-center);cc<=center;cc++){
            if(((c+cc) >= 0) && ((c+cc) < cols)){
               dot += (float)image[r*cols+(c+cc)] * kernel[center+cc];
               sum += kernel[center+cc];
            }
         }
         tempim[r*cols+c] = dot/sum;
      }
   }
}

free(tempim);
free(kernel);
```

# OPENMP CODE

- This is the portion for x-direction, which targets the columns of the test.pgm.
- For y-direction you need to implement the same methodology for the rows of the test.pgm.

```
/*******************************************************************************
* Blur in the x - direction.
*******************************************************************************/
if(VERBOSE) printf("   Bluring the image in the X-direction.\n");
for(r=0;r<rows;r++){
   for(c=0;c<cols;c++){
      dot = 0.0;
      sum = 0.0;
      for(cc=(-center);cc<=center;cc++){
         if(((c+cc) >= 0) && ((c+cc) < cols)){
            dot += (float)image[r*cols+(c+cc)] * kernel[center+cc];
            sum += kernel[center+cc];
         }
      }
      tempim[r*cols+c] = dot/sum;
   }
}

/*******************************************************************************
* Blur in the y - direction.
*******************************************************************************/
if(VERBOSE) printf("   Bluring the image in the Y-direction.\n");
for(c=0;c<cols;c++){
   for(r=0;r<rows;r++){
      sum = 0.0;
      dot = 0.0;
      for(rr=(-center);rr<=center;rr++){
         if(((r+rr) >= 0) && ((r+rr) < rows)){
            dot += tempim[(r+rr)*cols+c] * kernel[center+rr];
            sum += kernel[center+rr];
         }
      }
      (*smoothedim)[r*cols+c] = (short int)(dot*BOOSTBLURFACTOR/sum + 0.5);
   }
}

free(tempim);
free(kernel);
```

# OPENMP CODE

- Compiling the code:
  - $gcc canny_local_omp.c –lm –fopenmp –o ./canny_omp
  - You should mention that you are using openmp with –fopenmp
  - Math library with –lm
- Executing the file
  - time ./canny_omp test.pgm 1.0 1.0 0.8
  - Get the real time of computation and compare it with your reference, see the improvement.

- The definitions takes place before the gaussian_smooth

# PTHREAD CODE

```
/*******************************************************************************
* Blur in the x - direction.
*******************************************************************************/
```

```
/*******************************************************************************
* Blur in the y - direction.
*******************************************************************************/
```

- We will modify the gaussian_smooth function for pthreads.
- But first we need to define the parameters to be targeted by the threads, before defining the gaussian_smooth.
- Then we need to insert the algorithms for x and y-blur inside the gaussian_smooth.
- At the end we need to join the threads we created.

```
/*******************************************************************************
* PROCEDURE: gaussian_smooth
* PURPOSE: Blur an image with a gaussian filter.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void gaussian_smooth(unsigned char *image, int rows, int cols, float sigma,
        short int **smoothedim)
{
   int r, c, rr, cc,     /* Counter variables. */
      windowsize,        /* Dimension of the gaussian kernel. */
      center;            /* Half of the windowsize. */
   float *tempim,        /* Buffer for separable filter gaussian smoothing. */
         *kernel,        /* A one dimensional gaussian kernel. */
         dot,            /* Dot product summing variable. */
         sum;            /* Sum of the kernel weights variable. */

   /****************************************************************************
   * Create a 1-dimensional gaussian smoothing kernel.
   ****************************************************************************/
   if(VERBOSE) printf("   Computing the gaussian smoothing kernel.\n");
   make_gaussian_kernel(sigma, &kernel, &windowsize);
   center = windowsize / 2;

   /****************************************************************************
   * Allocate a temporary buffer image and the smoothed image.
   ****************************************************************************/
```

# PTHREAD CODE

- First, we are defining the class for thread_args_x objects.
- We will use these variables for each thread by pointing to them.
- In order to feed a function to pthread we need to use pass by reference logic.

```
//Added by Jiang Wan for multi-thread
struct thread_args_x
{
    unsigned char *image;
    int rows;
    int cols;
    int col_s;
    int col_e;
    int center;
    float *kernel;
    float *tempim;
};
```

# PTHREAD CODE

- `void *blur_x(…)` → is the way to define functions for pthreads.
- You are pointing the given variables to the ‘arguments’ pointer via *args object, which holds each variable with a different type.
- Then, we will feed the arguments to the blur_x function with a pointer.
- Each thread will use these variables during the computation.

```
//Added by Jiang Wan for multi-thread
void *blur_x(void *arguments)
{
    /*****************************************************************************
    * Blur in the x - direction.
    *****************************************************************************/
    struct thread_args_x *args = arguments;

    unsigned char *image = args->image;
    int rows = args->rows;
    int cols = args->cols;
    int col_s = args->col_s;
    int col_e = args->col_e;
    int center = args->center;
    float *kernel = args->kernel;
    float *tempim = args->tempim;

    int r, c, cc;
    float dot, sum;

    if(VERBOSE) printf("   Bluring the image in the X-direction.\n");
    for(r=0;r<rows;r++){
       for(c=col_s;c<col_e;c++){
          dot = 0.0;
          sum = 0.0;
          for(cc=(-center);cc<=center;cc++){
             if(((c+cc) >= 0) && ((c+cc) < cols)){
                dot += (float)image[r*cols+(c+cc)] * kernel[center+cc];
                sum += kernel[center+cc];
             }
          }
          tempim[r*cols+c] = dot/sum;
       }
    }
}

//Edited by Jiang Wan for multi-thread
#define N_T 4
```

```
//Added by Jiang Wan for multi
struct thread_args_x
{
    unsigned char *image;
    int rows;
    int cols;
    int col_s;
    int col_e;
    int center;
    float *kernel;
    float *tempim;
};
```

# PTHREAD CODE

- Defining the variables for the x – direction which are the columns.
- You will have to do a similar implementation for y direction:
  - Define new class for y
  - Rows will need `int r, rr, c;`
- At the end, you need to define the number of threads as (4 threads):
  - `#define N_T 4`

```
//Added by Jiang Wan for multi-thread
void *blur_x(void *arguments)
{
    /*****************************************************************************
    * Blur in the x - direction.
    *****************************************************************************/
    struct thread_args_x *args = arguments;

    unsigned char *image = args->image;
    int rows = args->rows;
    int cols = args->cols;
    int col_s = args->col_s;
    int col_e = args->col_e;
    int center = args->center;
    float *kernel = args->kernel;
    float *tempim = args->tempim;

    int r, c, cc;
    float dot, sum;

    if(VERBOSE) printf("   Bluring the image in the X-direction.\n");
    for(r=0;r<rows;r++){
        for(c=col_s;c<col_e;c++){
            dot = 0.0;
            sum = 0.0;
            for(cc=(-center);cc<=center;cc++){
                if(((c+cc) >= 0) && ((c+cc) < cols)){
                    dot += (float)image[r*cols+(c+cc)] * kernel[center+cc];
                    sum += kernel[center+cc];
                }
            }
            tempim[r*cols+c] = dot/sum;
        }
    }
}

//Edited by Jiang Wan for multi-thread
#define N_T 4
```

```
//Added by Jiang Wan for multi
struct thread_args_x
{
    unsigned char *image;
    int rows;
    int cols;
    int col_s;
    int col_e;
    int center;
    float *kernel;
    float *tempim;
};
```

# PTHREAD CODE

- Modifications inside the gaussian_smooth function.
- `pthread_t thread[N_T];` → is the pthread object. Class is defined in the library. It takes # of threads as input.
- `int iret[N_T];` → iret is the array that holds the pthread objects.

```
#define N_T 4
/*******************************************************************************
* PROCEDURE: gaussian_smooth
* PURPOSE: Blur an image with a gaussian filter.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void gaussian_smooth(unsigned char *image, int rows, int cols, float sigma,
        short int **smoothedim)
{
   int r, c, rr, cc,      /* Counter variables. */
      windowsize,         /* Dimension of the gaussian kernel. */
      center;             /* Half of the windowsize. */
   float *tempim,         /* Buffer for separable filter gaussian smoothing. */
         *kernel,         /* A one dimensional gaussian kernel. */
         dot,             /* Dot product summing variable. */
         sum;             /* Sum of the kernel weights variable. */
```

```
pthread_t thread[N_T];
int iret[N_T];
struct thread_args_x args[N_T];

int col_per_t = cols/N_T;
int i;

for(i=0; i<N_T; i++)
{
   args[i].image = image;
   args[i].rows = rows;
   args[i].cols = cols;
   args[i].col_s = col_per_t*i;
   args[i].col_e = col_per_t*(i+1);
   args[i].center = center;
   args[i].kernel = kernel;
   args[i].tempim = tempim;
   iret[i] = pthread_create(&thread[i], NULL, &blur_x, &args[i]);
}

for(i=0; i<N_T; i++)
{
 pthread_join(thread[i], NULL);
```

```
//Added by Jiang Wan for multi
struct thread_args_x
{
    unsigned char *image;
    int rows;
    int cols;
    int col_s;
    int col_e;
    int center;
    float *kernel;
    float *tempim;
};
```

# PTHREAD CODE

- `struct thread_args_x args[N_T];`

→ is the arguments object. Its class was predefined before the gaussian_smooth.

- `int col_per_t = cols/N_T;`

→ dividing the number of columns to number of threads to match each thread with a certain amount of columns. i.e. (each thread is responsible for 1024/4 = 256 columns)

```
#define N_T 4
/*******************************************************************************
* PROCEDURE: gaussian_smooth
* PURPOSE: Blur an image with a gaussian filter.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void gaussian_smooth(unsigned char *image, int rows, int cols, float sigma,
        short int **smoothedim)
{
   int r, c, rr, cc,      /* Counter variables. */
      windowsize,         /* Dimension of the gaussian kernel. */
      center;             /* Half of the windowsize. */
   float *tempim,         /* Buffer for separable filter gaussian smoothing. */
         *kernel,         /* A one dimensional gaussian kernel. */
         dot,             /* Dot product summing variable. */
         sum;             /* Sum of the kernel weights variable. */
```

```
pthread_t thread[N_T];
int iret[N_T];
struct thread_args_x args[N_T];

int col_per_t = cols/N_T;
int i;

for(i=0; i<N_T; i++)
{
   args[i].image = image;
   args[i].rows = rows;
   args[i].cols = cols;
   args[i].col_s = col_per_t*i;
   args[i].col_e = col_per_t*(i+1);
   args[i].center = center;
   args[i].kernel = kernel;
   args[i].tempim = tempim;
   iret[i] = pthread_create(&thread[i], NULL, &blur_x, &args[i]);
}

for(i=0; i<N_T; i++)
{
 pthread_join(thread[i], NULL);
```

```
//Added by Jiang Wan for multi
struct thread_args_x
{
    unsigned char *image;
    int rows;
    int cols;
    int col_s;
    int col_e;
    int center;
    float *kernel;
    float *tempim;
};
```

# PTHREAD CODE

- `int i;` → iteration counter
- The for loop prepares the arguments for each thread.
- Since we separated the columns for each thread, we should define the number of the start and end columns for each thread.
- Then we have to create the threads and point the arguments to each thread .

```
#define N_T 4
/*******************************************************************************
* PROCEDURE: gaussian_smooth
* PURPOSE: Blur an image with a gaussian filter.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void gaussian_smooth(unsigned char *image, int rows, int cols, float sigma,
        short int **smoothedim)
{
    int r, c, rr, cc,      /* Counter variables. */
       windowsize,         /* Dimension of the gaussian kernel. */
       center;             /* Half of the windowsize. */
    float *tempim,         /* Buffer for separable filter gaussian smoothing. */
          *kernel,         /* A one dimensional gaussian kernel. */
          dot,             /* Dot product summing variable. */
          sum;             /* Sum of the kernel weights variable. */
```

```
pthread_t thread[N_T];
int iret[N_T];
struct thread_args_x args[N_T];

int col_per_t = cols/N_T;
int i;

for(i=0; i<N_T; i++)
{
   args[i].image = image;
   args[i].rows = rows;
   args[i].cols = cols;
   args[i].col_s = col_per_t*i;
   args[i].col_e = col_per_t*(i+1);
   args[i].center = center;
   args[i].kernel = kernel;
   args[i].tempim = tempim;
   iret[i] = pthread_create(&thread[i], NULL, &blur_x, &args[i]);
}

for(i=0; i<N_T; i++)
{
 pthread_join(thread[i], NULL);
}
```

```
//Added by Jiang Wan for multi
struct thread_args_x
{
    unsigned char *image;
    int rows;
    int cols;
    int col_s;
    int col_e;
    int center;
    float *kernel;
    float *tempim;
};
```

# PTHREAD CODE

- `iret[i] = pthread_create(&thread[i], NULL, &blur_x, &args[i]);`

→ `pthread_create:` creates the threads from pthreads library.

→ `&thread[i]:` is the pointer for the thread that we create.

→ `NULL:` for OS and security, don't change.

→ `&blur_x:` is the function that we want to call. We will reach to the memory location of this function for each thread.

→ `&args[i]:` is the arguments that we want to feed to the blur_x function.

```
pthread_t thread[N_T];
int iret[N_T];
struct thread_args_x args[N_T];

int col_per_t = cols/N_T;
int i;

for(i=0; i<N_T; i++)
{
   args[i].image = image;
   args[i].rows = rows;
   args[i].cols = cols;
   args[i].col_s = col_per_t*i;
   args[i].col_e = col_per_t*(i+1);
   args[i].center = center;
   args[i].kernel = kernel;
   args[i].tempim = tempim;
   iret[i] = pthread_create(&thread[i], NULL, &blur_x, &args[i]);
}

for(i=0; i<N_T; i++)
{
 pthread_join(thread[i], NULL);
}
```

# PTHREAD CODE

```
#define N_T 4
/*******************************************************************************
* PROCEDURE: gaussian_smooth
* PURPOSE: Blur an image with a gaussian filter.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void gaussian_smooth(unsigned char *image, int rows, int cols, float sigma,
        short int **smoothedim)
{
    int r, c, rr, cc,      /* Counter variables. */
       windowsize,         /* Dimension of the gaussian kernel. */
       center;             /* Half of the windowsize. */
    float *tempim,         /* Buffer for separable filter gaussian smoothing. */
          *kernel,         /* A one dimensional gaussian kernel. */
          dot,             /* Dot product summing variable. */
          sum;             /* Sum of the kernel weights variable. */
```

- After we are done with the threads that we have created for parallelism, we should recombine the threads to terminate the threads.
- To do so a simple for loop that uses `pthread_join(thread([i]),NULL)` is enough.
- At the end don't forget to free the memory.

```
pthread_t thread[N_T];
int iret[N_T];
struct thread_args_x args[N_T];

int col_per_t = cols/N_T;
int i;

for(i=0; i<N_T; i++)
{
   args[i].image = image;
   args[i].rows = rows;
   args[i].cols = cols;
   args[i].col_s = col_per_t*i;
   args[i].col_e = col_per_t*(i+1);
   args[i].center = center;
   args[i].kernel = kernel;
   args[i].tempim = tempim;
   iret[i] = pthread_create(&thread[i], NULL, &blur_x, &args[i]);
}

for(i=0; i<N_T; i++)
{
 pthread_join(thread[i], NULL);
}
```

```
free(tempim);
free(kernel);
```

# PTHREAD CODE

- Compiling the code:
  - $gcc canny_local_omp.c –lm –lpthread –o ./canny_ptr
  - You should mention that you are using pthread library with –lpthread
  - Math library with –lm
- Executing the file
  - time ./canny_ptr test.pgm 1.0 1.0 0.8
  - Get the real time of computation and compare it with your reference, see the improvement.
- Compare your results with OpenMP and your reference.
- Make sure you comment on the way you implement your modifications and comment o the results.