Practical 2 - Static pThreads

Roan J. Song[†]
EEE4084F Class of 2017
University of Cape Town
South Africa
[†]SNGROA001

***Abstract*—This report concerns the partitioning of data for multi-threaded applications, using a median filter on JPEG images as the test case.**

I. Introduction

Parallelisation of applications through the division of labour amongst static threads is a forward-looking approach to improving application performance. Applications comprised of multiple similar instructions operating on different sets of data (SISD systems) can allocate their resources amongst a set of static threads; each of which can carry out the instructions on their local resources.

The application used in this report is a median filter - a method to remove speckled noise from datasets by setting each pixel of an image to the median of its surrounding pixel values. It involves highly independent data operations, and is thus well-suited to parallelisation.

II. Methodology

A. Hardware

The experiments conducted in this report were performed on a computer with the following specifications:

- CPU: i5-2500 (4 cores @ 3.3-3.7 GHz)
- RAM: 8 GB DDR3
- OS: Ubuntu 15.04

B. Establishing a Golden Measure

Before any parallelisation of an application can be done, it is of best interest to determine the performance of the sequential case. This establishes performance ranges and outputs that are then known and can be used to ensure that one is getting the correct output from the parallel implementation, as well as allowing one to calculate the speed-up of the parallel implementation.

1) Median Filter: A median filter is used to reduce the effects of speckled noise on a signal by setting each component of the signal to the median of its surrounding values. In images, this equates to a 'window' of pixels around each pixel which is sorted in order to find the median. A square 9x9 window (81 pixels in total) was decided upon. Other window shapes and sizes are viable and are best determined on a per-application basis.

Each pixel of a JPEG file is comprised of three colour components. To avoid looping over the image three times (once for each colour), and to pre-empt potential complications when allocating resources to threads, all of the colour components of each pixel are included, and sorted separately within one array. This resulted in an array of 243 integers being assigned for each pixel. I had initially created three arrays of 81 integers each, but encountered reduced performance due to the increase in memory access times. Keeping a larger array cached proved to be approximately three times as efficient. This may not be the case if the array is too large to be cached in its entirety.

```
void medFilter(int xmin=0,int xmax=Input.Width-1,
               int ymin=0,int ymax=Input.Height-1){
  int size = MASK_SIZE;
  int side = (int)(size-1)/2;
  int mask[size*size*3] = {0}; // each pixel has 3 components
  int x,y,r,b,a;
  for(y = ymin; y <= ymax; y++){
    for(x = xmin x <= xmax*3; x+=3){
      int mask[size*size*3] = {0};
      for(r = 0; r < 3; r++){
        for(b = y-side; b <= y+side; b++){
          for(a = x-(side*3)+r; a <= x+(side*3)+r; a += 3){
            // Test edge cases
            // Set mask elements to zero if out of bounds
            if(a<0)
              mask[(b+side-y)*size*3+a+3-x]=Input.Rows[y][x+r];
            else if(b<0)
              mask[(b+side-y)*size*3+a+3-x]=Input.Rows[y][x+r];
            else if(a>=Input.Width*3)
              mask[(b+side-y)*size*3+a+3-x]=Input.Rows[y][x+r];
            else if(b>=Input.Height)
              mask[(b+side-y)*size*3+a+3-x]=Input.Rows[y][x+r];
            else
              mask[(b+side-y)*size*3+a+3-x]=Input.Rows[b][a];
          }}      }
      bubbleSort(mask,sizeof(mask)/sizeof(mask[0]),Output,y,x);
    }}}
```

2) Sorting Pixel Values: I opted to use a slightly optimised Bubble Sort implementation to sort the mask values. Bubble Sort is a very naïve method of sorting, of complexity $O(n^2)$, and is not suitable for large pixel mask sizes. The sorting of each pixel mask in order to determine the median is the most costly operation within the median filter, and performance improvements should be easily achieved through the implementation of a more efficient sorting algorithm such as quicksort or even insertion sort. The code below shows every value being compared with a value three places away. This is due to the inclusion of the three colour components of each pixel in the mask, which allows for three $O(n^2)$ sorts within a larger array and eliminates the need for a separate set of data accesses between each sort.

```
void bubbleSort(int * arr, int length, JPEG& Output,int y, int x){
  int out[3] = {0};
  int n = length;
  bool swap = 1;
  int temp;
  // sorting is complete when no more swaps are made when traversing the array
  while(swap == 1){
    swap = 0;
    for(int i =3; i < n; i++){
      if(arr[i-3] > arr[i]){
        temp = arr[i-3];
        arr[i-3] = arr[i];
        arr[i] = temp;
        swap = 1;}}
    // n can be reduced on every loop, as the highest value is guaranteed to be sorted.
    n = n-1;}
  //finding the median of the array and its neighbours gives the RGB values
  temp = (int)length/2;
  Output.Rows[y][x] = arr[temp-1];
  Output.Rows[y][x+1] = arr[temp];
  Output.Rows[y][x+2] = arr[temp+1];
  return;}
```

C. Parallelisation

1) Implementation: Parallelisation of the median filter application is achieved through the use of static pThreads. A main thread method is declared, which is used by every thread. A struct containing the data to be used by each thread is passed in as a void pointer, which is subsequently dereferenced and cast to the struct type. To enable the use of threading, pThreads need to be created and allocated data. Once they are created they run their assigned method and complete, attempting to join through the *pthread_join* method whereupon the threading section is complete and the output can be written.

Each thread accesses a set of pixels and their neighbours. Accessing image data is a memory read, which is a non-blocking operation. Each thread writes to a distinct set of pixels. There is no overlap between threads, so no possibility of conflict during each write. Therefore image access does not require each thread to go into a critical section.

The data partitioning method decided upon was to use large contiguous blocks of each image per thread, which minimizes out-of-cache memory accesses. If each pixel is building its mask from nearby pixels which are also stored in the local cache, it will be much faster than accessing out-of-cache memory as it would be if rows were separated.

```
struct param_struct{
  int xmin;
  int xmax;
  int ymin;
  int ymax;
};

void* Thread_main(void* argument){
  struct param_struct *params = (struct param_struct*)argument;
  medFilter(params->xmin,params->xmax,
            params->ymin,params->ymax);
  return 0;}
```

```
pthread_t Thread   [Thread_Count];
int j;
struct param_struct structs[Thread_Count];
int ind0,ind1,xmin,xmax,ymin,ymax;

// Allocating data to threads
int perthread = Input.Height/Thread_Count;
int leftover = Input.Height%Thread_Count;
int temp =0;
for(j = 0; j < Thread_Count; j++){
  ind0 = temp;
  ind1 = temp+(perthread)-1;
  if(leftover > 0){
    ind1++;
    leftover--;}
  temp = ind1+1;

  structs[j].xmin = 0;
  structs[j].xmax = Input.Width-1;
  structs[j].ymin = ind0;
  structs[j].ymax = ind1;
  pthread_create(Thread+j, 0, Thread_main, &structs[j]);
}

// wait for threads to finish
for(j = 0; j < Thread_Count; j++){
  if(pthread_join(Thread[j], 0)){
    pthread_mutex_lock(&Mutex);
    printf("Problem joining thread %d\n", j);
    pthread_mutex_unlock(&Mutex);}
}
```

2) Data Allocation: The input image is divided up according to the number of threads, with each threads getting n rows of the image, where n is the height of the array divided by the number of threads. If the number of threads is not a perfect factor of the height of the array, then some rows will be leftover. This problem was dealt with by calculating the 'leftover' (height%threads) and assigning one extra row to each thread while decrementing the leftover, until no extra rows remain.

3) Validation: To ensure that the results achieved by the parallel version mirror those of the Golden Measure exactly, it is necessary to assert that the output of the parallel application is exactly equal to that of the Golden Measure. Linux has a built-in function called 'diff' that compares two files and outputs the difference. If no output is received, then the two files are identical. It is less computationally complex than a correlational comparison. Thus 'diff output.jpg desired.jpg' was the method chosen to validate the parallel implementation.

III. Results & Discussion

A. Golden Measure

The Golden Measure is the most basic sequential implementation of the desired application, with correct and consistent results required for the implementation to be considered successful.

1) Median Filter: The median filter algorithm performed as expected; removing noise from each image and emphasising the edges present within each image. Increasing the window size led to a more pronounced effect.

2) Performance: The performance of the sequential implementation left much to be desired, with the largest test image taking $\tilde{5}$ minutes to process.

Filename	Size [pixels]	Time Taken [s]
small.jpg	91200	4.05051
fly.jpg	840704	51.1478
greatwall.jpg	4915200	299.951

TABLE I
SEQUENTIAL PERFORMANCE ON TEST IMAGES

B. Parallel Implementation

Filename	Size [pixels]	Thread Count	Time Taken [s]	Speed-up
small.jpg	91200	2	2.263	1.789
small.jpg	91200	4	1.170	3.462
small.jpg	91200	8	1.1151	3.632
small.jpg	91200	16	1.104	3.668
fly.jpg	840704	2	26.225	1.952
fly.jpg	840704	4	14.102	3.629
fly.jpg	840704	8	14.072	3.637
fly.jpg	840704	16	13.913	3.677
greatwall.jpg	4915200	2	144.840	2.071
greatwall.jpg	4915200	4	77.575	3.866
greatwall.jpg	4915200	8	76.203	3.936
greatwall.jpg	4915200	16	75.965	3.949

TABLE II
MULTI-THREADED PERFORMANCE ON TEST IMAGES

IV. CONCLUSIONS

The parallel implementation of the median filter application significantly outperformed the sequential implementation. Image operations involving simple instructions executed on multiple sets of data seem suited to threaded implementations. Careful management of data accesses led to significant performance improvements, suggesting that a parallelism does not necessarily outweigh the improvements that can be made through improving the sequential algorithm.