Laboratory 1D:

Parallel Sort Assignment/Lab

Total Marks: 15

Due: Week 8 Lab Session

Useful Resources

Lecture notes from learning@griffith.

Information

- This lab is due and assessed at the beginning of your scheduled lab session in Week 8. In Week 8 you **will** demonstrate this programs created in this lab and supply the answer to written question 3.
- There is no preliminary.
- This lab is to be completed as an assignment. There will be no scheduled assessed lab sessions this week. However the Thursday lab will have a demonstrator for anyone that needs assistance.
- The labs this week can be done at home, or in the University labs. N44 level 3.05 (Pis), N44 0.17 (PCs), N44 Level -1 (PCs). These labs are available during the week and the Technical Officers are available to let you into the labs. Note that if using the PCs, you must choose the Mint Linux app in the app list from the start button.

Experimental (15 marks)

Make sure that you save each question separately. Merged questions will only be marked as one question. Therefore you should have 2 programs created and saved.

Question 1 (5 marks)

Write a program that performs a sample sort on a dataset that you create. The dataset is a space delimited list of integers between 0 and 1000. The program should open a file called dataset.txt which contains the data to be sorted. The number of bins used is your choice. The program should also time the length of time taken to complete the sort.

Hints: *The timing functions can easily be found in any slightly advanced C programming textbook or website. Also make sure that you use a large enough dataset to get a meaningful time.*

Question 2 (5 marks)

Modify Q1 so that it uses threads to perform a parallel sample sort.

Question 3 (5 marks)

The aim of part 3 is to determine the effect of increasing the number of threads on the time taken to perform the parallel sort. This will be demonstrated by plotting a graph of the average time for the sort versus the number of threads used.

You should choose the number of threads so that range is 1 <= number of threads <= 10.

Repeat this procedure 3 times and calculate the average taken time for the different number of threads.

- Plot the graph of the average time for the sort vs the number of threads used.
- Write down the processor type and the number of cores for the computer that you used to perform this question.
- Explain the graph.

Question 3

Processer: Intel® Core™ i5-3230M CPU @ 2.60GHz, 2601Mhz, 2 Core(s), 4 Logical Processors(s)

Thread	Avg time
1	3.584375
2	3.332813
3	3.78125
4	4.171875
5	4.003125
6	4.21875
7	3.98125
8	4.496875
9	4.445312
10	4.209375

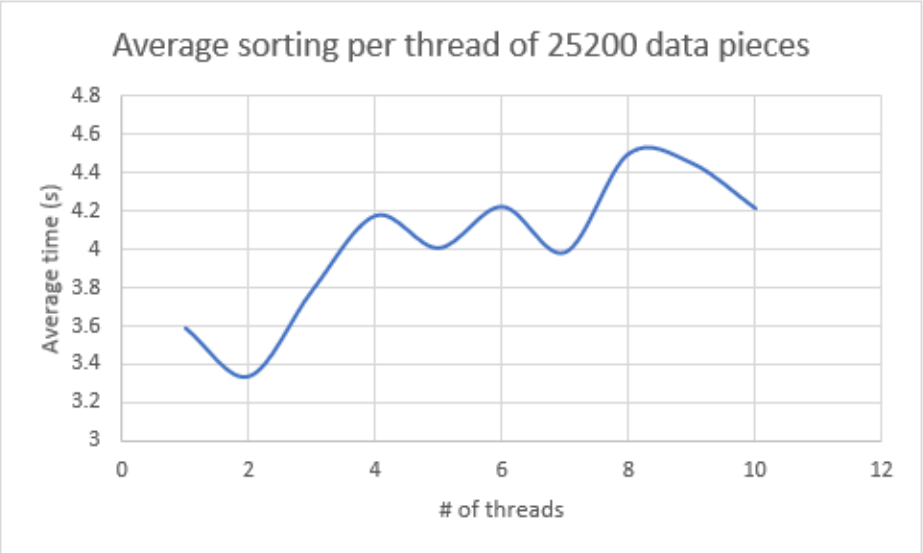

Figure 1: Average time taken for each # of thread used in the parallel programming of a sample sort used in conjuncture with a Bucket Sort via multithreading.

The increasing of sorting ranges via the addition of another thread to sort a particular range of integers shows that certain thread numbers (using parallel sorting of 2 – 3) could be beneficial with dips in 5 and 7 where the larger of the resources being chewed up by the program from the system is happening around 8 to 10 threads.

The higher peaks happening at thread numbers 4 and 6 could be from a range for one bucket containing a majority of the randomised data, the distribution could be held entirely by accident from a poor sampling algorithm, decreasing the overall time can be achieved by re-evaluating how a random sample of the whole dataset is obtained, post sorting and acquiring splitters from the sorted random sample could help distribute the whole dataset evenly amongst the buckets (threads).

Question 1 code:

```
// Joseph Simeon
// Comsys Lab 1D -- Sorting
// pre-processors
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// #define
#define limit -1
#define thr 3

// global array
int *splitter, *split;

// pre-defined functions
void PrintArray(char *, int *, int);
FILE *OpenFile(char *, char *);
int CheckDataset(FILE *);
int CheckThreadMod(int);
void QuickSort(int *, int, int);
void BubbleSort(int *, int);

// main.c
int main(void){
// variables
    // clock
    clock_t start, end;
    double cpu_time_used;
    // non-pointer variables
    int data_size, new_thr, elements, bucket_size, data, primecheck;
    int SplitterNumber, SplitterElement, SplitInterval;
    // loop variables
    int i, j, k;
    // FILE pointer
    FILE *fp;

// clock start
    start = clock();
// check how much data the file has
    fp = OpenFile("dataset.txt", "r");
    data_size = CheckDataset(fp);
        primecheck = 1;
    for(i = 3; i < data_size/2; i++){
        if(data_size % i == 0){
            primecheck = 0;
        }
    }
```

```
    if(data_size < 10 || primecheck == 1){
        int *array;
        array = malloc(data_size*sizeof(int));
        fp = OpenFile("dataset.txt", "r");
        i = 0;
        while(fscanf(fp, "%i\n", &array[i++]) != EOF);
        PrintArray("Unsorted array", array, data_size);
        QuickSort(array, 0, data_size);
        PrintArray("Sorted array", array, data_size);
        fclose(fp);
        free(array);
    // clock end
        end = clock();
        cpu_time_used = ((double)(end-start))/CLOCKS_PER_SEC;
        printf("Time taken: %f\n", cpu_time_used);
        return(0);
    }

// create master array
    // reopen file
    fp = OpenFile("dataset.txt", "r");
    // create dataset array
    int *MasterArray = malloc(data_size*sizeof(int));
    // input data
    i = 0;
    while(fscanf(fp, "%i\n", &MasterArray[i++]) != EOF);
    // print master array data
    PrintArray("Master array", MasterArray, data_size);

// create sub set of array
    // determine slave array parameters
    new_thr = CheckThreadMod(data_size);
    elements = data_size/new_thr;
    // create slave array
    int *SlaveArray[new_thr];
    for(i = 0; i < new_thr; i++){
        SlaveArray[i] = malloc(elements*sizeof(int));
    }
    // input slave array data
    k = 0;
    for(i = 0; i < new_thr; i++){
        for(j = 0; j < elements; j++){
            SlaveArray[i][j] = MasterArray[k++];
        }
    }
    // print slave array data
    for(i = 0; i < new_thr; i++){
        PrintArray("Slave array", SlaveArray[i], elements);
```

```
    }
    // free mastery array memory
        // master array memory is unneeded and will only take up space
    free(MasterArray);

// create splitter array
    // determine splitter parameters
    SplitterNumber = (elements/(new_thr))*new_thr;
    SplitterElement = 1;
    for(i = 1; i < SplitterNumber/2; i++){
        if(SplitterNumber % i == 0){
                if(i > SplitterElement){
                    SplitterElement = i;
                }
        }
    }
    SplitterElement = SplitterNumber/SplitterElement;
    printf("The number of elements in the splitter is %i with data gathered at
 every %i interval\n", SplitterNumber, SplitterElement);
    // create splitter memory (splitter is global)
    splitter = malloc(SplitterNumber*sizeof(int));
    k = 0;
    for(i = 0; i < new_thr; i++){
        for(j = 0; j < elements; j++){
            if( (j+1) % SplitterElement == 0){
                splitter[k] = SlaveArray[i][j];
                k++;
            }
        }
    }
    // print splitter data
    PrintArray("Splitter", splitter, SplitterNumber);
    // sort splitter
    //QuickSort(splitter, 0, SplitterNumber);
    BubbleSort(splitter, SplitterNumber);
/*  for(i = 0; i < SplitterNumber; i++){
        for(j = 0; j < SplitterNumber-1; j++){
            if(splitter[j] > splitter[j+1]){
                tempvar = splitter[j+1];
                splitter[j+1] = splitter[j];
                splitter[j] = tempvar;
            }
        }
    } */
    // print sorted splitter data
    PrintArray("Sorted splitter", splitter, SplitterNumber);
    // free slave array memory
    for(i = 0; i < new_thr; i++){
```

```
        free(SlaveArray[i]);
    }

// create split array for bucket evaluation
    // determine split parameters
    SplitInterval = SplitterNumber/SplitterElement;
    printf("Interval of splitter is %i\n", SplitInterval);
    // create split array
    split = malloc(new_thr*sizeof(int));
    for(i = 0; i < new_thr; i++){
        split[i] = splitter[i*SplitInterval];
    }
    // print split array data
    PrintArray("Split array", split, new_thr);
    // free splitter memory
    free(splitter);

// creating buckets to be sorted
    // create buckets
    int *bucket[new_thr];
    bucket_size = data_size;
    for(i = 0; i < new_thr; i++){
        bucket[i] = malloc(bucket_size*sizeof(int));
    }
    // zero value the buckets
    for(i = 0; i < new_thr; i++){
        for(j = 0; j < bucket_size; j++){
            bucket[i][j] = limit;
        }
    }
    // create an index corresponding to each bucket
    int *bindex = malloc(new_thr*sizeof(int));
    for(i = 0; i < new_thr; i++){
        bindex[i] = 0;
    }
    // boverflow -> Bucket Overflow for when the malloc parameter size of
    // the bucket has been overflow, the corresponding thread to the bucket
    // will be equal to 1 -- however i cannot figure out how to realloc
/*  int *boverflow = malloc(new_thr*sizeof(int));
    for(i = 0; i <new_thr; i++){
        boverflow[i] = 0;
    } */
    // MasterArray is freed therefore file pointer to be used
    fp = OpenFile("dataset.txt", "r");
    while(fscanf(fp, "%i\n", &data) != EOF){
        for(j = 0; j < new_thr; j++){
            if(j == 0){
                if(data < split[j]){
```

```
                    bucket[j][bindex[j]] = data;
                    bindex[j]++;
                }
            }else if(j == (new_thr-1)){
                if(data >= split[j]){
                    bucket[j][bindex[j]] = data;
                    bindex[j]++;
                }
            }else{
                if(data >= split[j-1] && data < split[j+1]){
                    bucket[j][bindex[j]] = data;
                    bindex[j]++;
                }
            }
        }
    }
    // print unsorted buckets
    for(i = 0; i < new_thr; i++){
        PrintArray("Bucket", bucket[i], bucket_size);
    }
    // bubble sort each bucket with their limit being zero
    for(i = 0; i < new_thr; i++){
        //QuickSort(bucket[i], 0, bucket_size);
        BubbleSort(bucket[i], bucket_size);
    }
    // print sorted buckets
    for(i = 0; i < new_thr; i++){
        PrintArray("Bucket", bucket[i], bucket_size);
    }

// create final sorted array
    // create array
    int *SortedArray;
    SortedArray = malloc(data_size*sizeof(int));
    for(i = 0; i < data_size; i++){
        SortedArray[i] = 0;
    }
    k = 0;
    for(i = 0; i < new_thr; i++){
        for(j = 0; j < bucket_size; j++){
            if(bucket[i][j] != limit){
                SortedArray[k++] = bucket[i][j];
            }
        }
    }
    // print sorted array data
//  QuickSort(SortedArray, 0, data_size);
    PrintArray("Sorted array", SortedArray, data_size);
```

```c
// closing file
    fclose(fp);
//free rest of memory
    for(i = 0; i < new_thr; i++){
        free(bucket[i]);
    }
    free(split);
    free(SortedArray);

// clock end
    end = clock();
    cpu_time_used = ((double)(end-start))/CLOCKS_PER_SEC;
    printf("Time taken: %f\n", cpu_time_used);
    return(0);
}

// functions
void PrintArray(char *Msg, int *Arr, int Num){
    int i;

    printf("%s:\n", Msg);
    for(i = 0; i < Num; i++){
        if(i == 0){
            printf("%i", Arr[i]);
        }else{
            printf(", %i", Arr[i]);
        }
    }
    printf("\n");
}

FILE *OpenFile(char *file, char *permission){
    FILE *fp;
    fp = fopen(file, permission);
    if(fp == NULL){
        printf("Error openining file %s\n", file);
        exit(-1);
    }
    return(fp);
}

int CheckDataset(FILE *fp){
    int data, C;
    C = 0;
    while(fscanf(fp, "%i\n", &data) != EOF){
        C++;
    }
```

```
    fclose(fp);
    return(C);
}

// checks to see if the data size of the file is divisible by the
// number of threads going to be used, if not then the for loop will find
// the nearest number, if the number is prime it will return the new
// thread number of 1
int CheckThreadMod(int ds){
    int new_thread_mod, i;

    if(ds % thr != 0){
        for(i = thr; i < ds; i++){
            if(ds % i == 0){
                new_thread_mod = i;
                break;
            }else if(i == ds){
                new_thread_mod = 1;
            }
        }
    }else{
        new_thread_mod = thr;
    }
    return(new_thread_mod);
}

void QuickSort(int array[], int first, int last){
    int i, j, pivot, temp;

    if(first < last){
        pivot = first;
        i = first;
        j = last;

        while(i < j){
            while(array[i] <= array[pivot] && i < last)
                i++;
            while(array[j] > array[pivot])
                j--;
            if(i < j){
                temp = array[i];
                array[i] = array[j];
                array [j] = temp;
            }
        }
        temp = array[pivot];
        array[pivot] = array[j];
        array[j] = temp;
```

```
        QuickSort(array, first, j-1);
        QuickSort(array, j+1, last);
    }
}

void BubbleSort(int *array, int size){
    int i, j, temp;

    for(i = 0; i < size; i++){
        for(j = 0; j < size-1; j++){
            if(array[j] != limit && array[j+1] != limit){
                if(array[j] > array[j+1]){
                    temp = array[j+1];
                    array[j+1] = array[j];
                    array[j] = temp;
                }
            }
        }
    }
}
```

Question 2 code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define thr 4
#define limit -1

// global
int current_n = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t number = PTHREAD_COND_INITIALIZER;

// pre-defined function
void *BucketSort(void *);
FILE *OpenFile(char *, char *);
int CheckDataset(FILE *);
void PrintArray(char *, int *, int);
void BubbleSort(int *, int);

struct threadvals{
    int *t_array;
    int *splitvals;
    int t_index;
    int datasize;
    int threadnumber;
    int interval;

};

// main.c
int main(void){
    clock_t start, end;
    double cpu_time_used;

    int *array, *splitter, dataset, thread_number, interv, split_number;
    int i;

    FILE *fp, *fs;

    // clock start
    start = clock();

    fs = OpenFile("sorteddataset.txt", "w");
    fclose(fs);

    // aquire data length
```

```
    fp = OpenFile("dataset.txt", "r");
    dataset = CheckDataset(fp);

//  thread_number = CheckThreadMod(dataset);
    thread_number = thr;
    interv = dataset / thread_number;
    split_number = thread_number - 1;
    if(split_number == 0){
        split_number = 1;
    }

    array = (int *)malloc(dataset*sizeof(int));
    fp = OpenFile("dataset.txt", "r");
    i = 0;
    while(fscanf(fp, "%i\n", &array[i++]) != EOF);
    PrintArray("Unsorted array", array, dataset);

    splitter = (int *)malloc(split_number*sizeof(int));
    for(i = 0; i < split_number; i++){
        splitter[i] = array[i*interv];
    }

    // if statement here to either show values of splitter or reveal thread is
 one
    if(thread_number == 1){
        // do nothing
    }else{
        PrintArray("Unsorted splitter", splitter, split_number);
        BubbleSort(splitter, split_number);
        PrintArray("Sorted splitter", splitter, split_number);
    }

    // setting struct vals
    struct threadvals *t_val[thread_number];
    for(i = 0; i < thread_number; i++){
        t_val[i] = (struct threadvals *)malloc(sizeof(struct threadvals));
    }
    for(i = 0; i < thread_number; i++){
        t_val[i]->t_array = (int *)malloc(dataset*sizeof(int));
        t_val[i]->splitvals = (int *)malloc(split_number*sizeof(int));
    }
    for(i = 0; i < thread_number; i++){
        t_val[i]->t_array = array;
        t_val[i]->splitvals = splitter;
        t_val[i]->t_index = i;
        t_val[i]->datasize = dataset;
        t_val[i]->threadnumber = thread_number;
        t_val[i]->interval = interv;
```

```
    }

    pthread_t *thread = (pthread_t *)malloc(thread_number*sizeof(pthread_t));
    for(i = 0; i < thread_number; i++){
        pthread_create(&thread[i], NULL, (void *)BucketSort, (void *)t_val[i])
;
    }
    for(i = 0; i < thread_number; i++){
        pthread_join(thread[i], NULL);
    }

    fs = OpenFile("sorteddataset.txt", "r");
    i = 0;
    while(fscanf(fs, "%i\n", &array[i++]) != EOF);
    PrintArray("Sorted array", array, dataset);


// clock end
    end = clock();
    cpu_time_used = ((double)(end - start) / CLOCKS_PER_SEC);
    printf("Time: %f\n", cpu_time_used);

    fclose(fp);
    fclose(fs);

    free(array);
    free(splitter);
    for(i = 0; i < thread_number; i++){
        free(t_val[i]);
    }

    return(0);
}

// functions
    // thread process
void *BucketSort(void *ptr){
    int *t_array, *t_splitter, myindex, arraysize, thread_n, split_number, cou
nt;
    int i, j;
    FILE *fp;

    struct threadvals *my_t_vals = (struct threadvals *) ptr;
    myindex = my_t_vals->t_index;
    arraysize = my_t_vals->datasize;
    thread_n = my_t_vals->threadnumber;
    split_number = thread_n - 1;
```

```
    t_array = (int *)malloc(arraysize*sizeof(int));
    t_array = my_t_vals->t_array;
    t_splitter = (int *)malloc(split_number*sizeof(int));
    t_splitter = my_t_vals->splitvals;

    int *subarray = (int *)malloc(arraysize*sizeof(int));
    for(i = 0; i < arraysize; i++){
        subarray[i] = limit;
    }

    //~ printf("Checking parameters of thread %i: datasize - %i, # of threads
- %i, # of splits - %i\n", myindex, arraysize, thread_n, split_number);
    //~ PrintArray("Checking unsorted array in thread", array, arraysize);
    //~ PrintArray("Checking sorted splitter in thread", splitter, split_numbe
r);
    // if thread happens to be 1
    if(thread_n == 1){
        BubbleSort(t_array, arraysize);
        fp = OpenFile("sorteddataset.txt", "w");
        for(i = 0; i < arraysize; i++){
            if(t_array[i] != limit){
                fprintf(fp, "%i\n", t_array[i]);
            }
        }
    }else{
        j = 0;
        for(i = 0; i < arraysize; i++){
            if(myindex == 0){
                if(t_array[i] < t_splitter[myindex]){
                    subarray[j++] = t_array[i];
                }
            }else if(myindex == split_number){
                if(t_array[i] >= t_splitter[myindex - 1]){
                    subarray[j++] = t_array[i];
                }
            }else{
                if(t_array[i] >= t_splitter[myindex - 1] && t_array[i] <t_spli
tter[myindex]){
                    subarray[j++] = t_array[i];
                }
            }
        }

        if(myindex == 0){
            printf("Thread process(%i): splitter[%i]: <%i\n", myindex, myindex
, t_splitter[myindex]);
        }else if(myindex == split_number){
```

```
            printf("Thread process(%i): splitter[%i]: <%i\n", myindex, myindex
, t_splitter[myindex - 1]);
        }else{
            printf("Thread process(%i): splitter[%i]: %i to %i\n", myindex, my
index, t_splitter[myindex - 1], t_splitter[myindex]);
        }
        PrintArray("Unsorted bucket", subarray, arraysize);
        BubbleSort(subarray, arraysize);
        PrintArray("Sorted bucket", subarray, arraysize);

        count = 0;
        for(i = 0; i < arraysize; i++){
            if(subarray[i] != limit){
                count++;
            }
        }
        int *filearray = (int *)malloc(count*sizeof(int));
        for(i = 0; i < count; i++){
            filearray[i] = subarray[i];
        }
        PrintArray("Array for file", filearray, count);

        pthread_mutex_lock(&mutex);
        while(myindex >  current_n){
            pthread_cond_wait(&number, &mutex);
        }
        // in this section -- i = current_n
        printf("Thead %i has started critical condition\n", myindex);
        fp = OpenFile("sorteddataset.txt", "a");
        for(i = 0; i < count; i++){
                fprintf(fp, "%i\n", filearray[i]);
        }

        printf("Thread %i has finished critical section\n", myindex);
        current_n++;
        pthread_cond_broadcast(&number);
        pthread_mutex_unlock(&mutex);

        free(subarray);
        free(filearray);
    }



    fclose(fp);

    return(NULL);
}
```

```
    // sort(s)
void BubbleSort(int *array, int size){
    int i, j, temp;

    for(i = 0; i < size; i++){
        for(j = 0; j < size-1; j++){
            if(array[j] != limit && array[j+1] != limit){
                if(array[j] > array[j+1]){
                    temp = array[j+1];
                    array[j+1] = array[j];
                    array[j] = temp;
                }
            }else{
                break;
            }
        }
        if(array[j] == limit && array[j+1] == limit){
            break;
        }
    }
}

    // other function(s)
    void PrintArray(char *Msg, int *Arr, int Num){
    int i;

    printf("%s:\n", Msg);
    for(i = 0; i < Num; i++){
        if(Arr[i] != limit){
            if(i == 0){
                printf("%i", Arr[i]);
            }else{
                printf(", %i", Arr[i]);
            }
        }else{
            break;
        }
    }
    printf("\n");
}

FILE *OpenFile(char *file, char *permission){
    FILE *fp;
    fp = fopen(file, permission);
    if(fp == NULL){
        printf("Error openining file %s\n", file);
        exit(-1);
```

```
    }
    return(fp);
}

int CheckDataset(FILE *fp){
    int data, C;
    C = 0;
    while(fscanf(fp, "%i\n", &data) != EOF){
        C++;
    }
    fclose(fp);
    return(C);
}

        // checks to see if the data size of the file is divisible by the
        // number of threads going to be used, if not then the for loop will f
ind
        // the nearest number, if the number is prime it will return the new
        // thread number of 1
int CheckThreadMod(int ds){
    int new_thread_mod, i;

    if(ds % thr != 0){
        for(i = thr; i < ds; i++){
            if(ds % i == 0){
                new_thread_mod = i;
                break;
            }else if(i == ds){
                new_thread_mod = 1;
            }
        }
    }else{
        new_thread_mod = thr;
    }
    return(new_thread_mod);
}
```