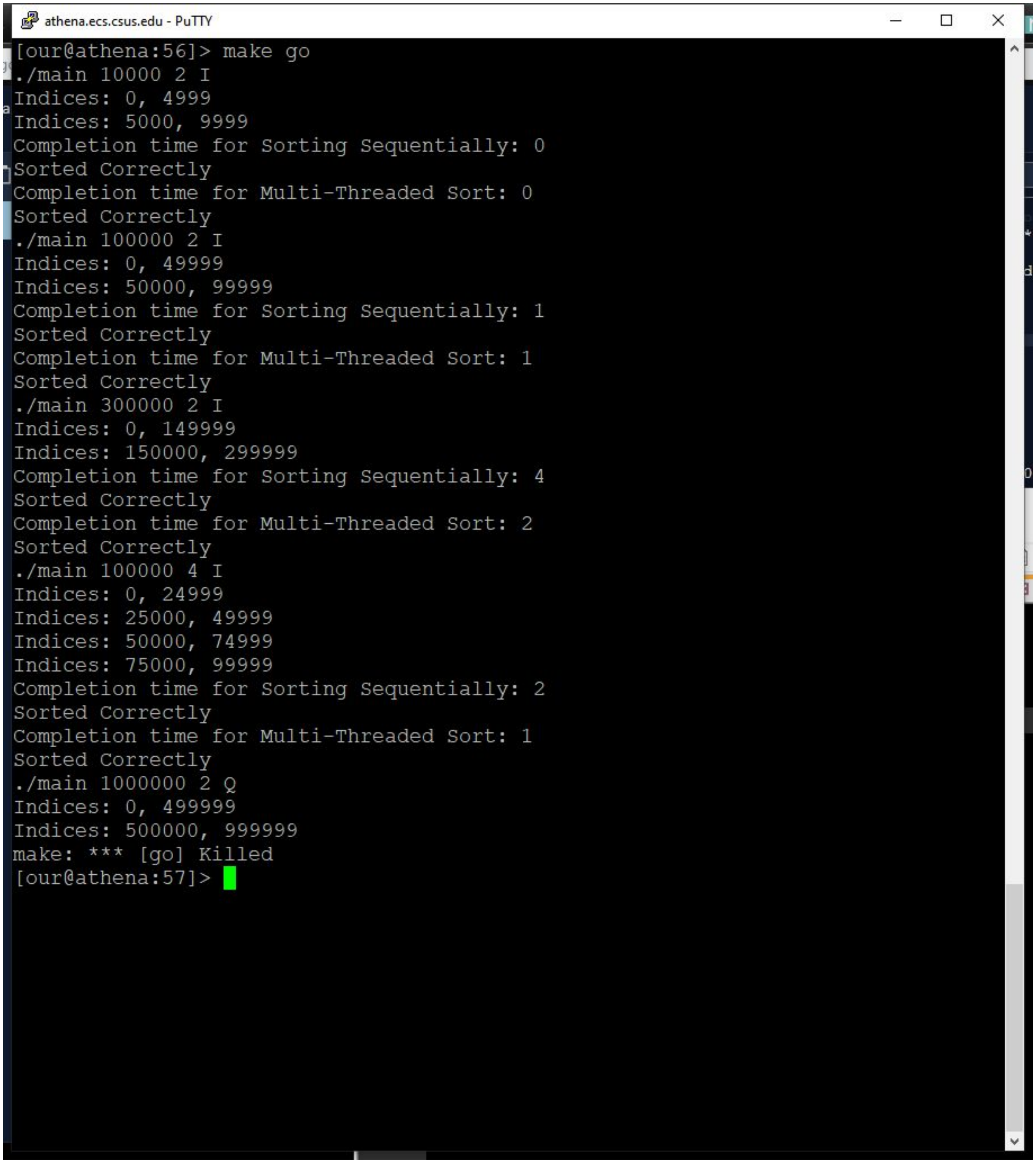
CSC139 A4
RONGGUANG OU

# Assignment 4 - Multi-Threading

Sorting different input size with different thread count.
Note: due to athena restriction, it was not able to sort 1M records in time.



```
[our@athena:56]> make go
./main 10000 2 I
Indices: 0, 4999
Indices: 5000, 9999
Completion time for Sorting Sequentially: 0
Sorted Correctly
Completion time for Multi-Threaded Sort: 0
Sorted Correctly
./main 100000 2 I
Indices: 0, 49999
Indices: 50000, 99999
Completion time for Sorting Sequentially: 1
Sorted Correctly
Completion time for Multi-Threaded Sort: 1
Sorted Correctly
./main 300000 2 I
Indices: 0, 149999
Indices: 150000, 299999
Completion time for Sorting Sequentially: 4
Sorted Correctly
Completion time for Multi-Threaded Sort: 2
Sorted Correctly
./main 100000 4 I
Indices: 0, 24999
Indices: 25000, 49999
Indices: 50000, 74999
Indices: 75000, 99999
Completion time for Sorting Sequentially: 2
Sorted Correctly
Completion time for Multi-Threaded Sort: 1
Sorted Correctly
./main 1000000 2 Q
Indices: 0, 499999
Indices: 500000, 999999
make: *** [go] Killed
[our@athena:57]>
```

Source Code :

```
#include <sys/timeb.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <ctype.h>

long gRefTime;
int *seqArray;
int *threadArray;
int *finalArray;
int indice[16][2];
char *algorithm;


//method to get the time in milliseconds
long GetMilliSecondTime(struct timeb timeBuf){
	long mliScndTime;
	mliScndTime = timeBuf.time;
	mliScndTime *= 1000;
	mliScndTime += timeBuf.millitm;
	return mliScndTime;
}

//method to get the current time in milliseconds
long GetCurrentTime(void){
	long crntTime=0;
	struct timeb timeBuf;
	ftime(&timeBuf);
	crntTime = GetMilliSecondTime(timeBuf);
	return crntTime;
}

//method to start the timer
void SetTime(void){
	gRefTime = GetCurrentTime();
}

//method to return the current - SetTime in milliseconds
long GetTime(void){
	long crntTime = GetCurrentTime();
```

```
    return (crntTime - gRefTime);
}

//method to swap two values
void Swap(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

//RNG
int Rand(int x, int y){
    int range = y-x+1;
    int r = rand() % range;
    r += x;
    return r;
}

//method for insertion sort
void InsertionSort(int data[], int start, int size){
    int i, j, temp;
    for(i = 0; i < size; i++){
        if(&data[i] == NULL){
            printf("Error with array index %d", i);
        }
    }
    for(i=start; i<size+1; i++){
        temp = data[i];
        for(j=i-1; j>=0 && data[j]>temp; j--)
            data[j+1] = data[j];
        data[j+1] = temp;
    }
}

//method to partition the data for quicksort
int Partition(int data[], int p, int r , int size){
  int i, j, x, pi;
    for(i = 0; i < size; i++){
        if(&data[i] == NULL){
                printf("Error, content at index is NULL %d\n", i);
                return -1;
        }
  }
```

```
            pi = Rand(p, r);
            Swap(&data[r], &data[pi]);

            x = data[r];
            i = p-1;
            for(j=p; j<r; j++) {
                        if(data[j] < x){
                                    i++;
                                    Swap(&data[i], &data[j]);
                        }
            }
            Swap(&data[i+1], &data[r]);
            return i+1;
}

//method for quick sort
void QuickSort(int data[], int p, int r , int size){

  for(int i = 0; i < size; i++){
          if(&data[i] == NULL){
                  printf("Error , content at index is NULL %d\n", i);
                  return;
          }
  }

            if(p >= r) return;
            int q = Partition(data, p, r , size);
            QuickSort(data, p, q-1 , size);
            QuickSort(data, q+1, r, size);
}


//Validate if is really a number
int isNumber(char number[]){
  int i = 0;
  for(i = 0 ; number[i] != 0 ;i++){
    if(number[i] >= 48 && number[i] <= 57){
      //Valid Digit
    }else{
      return 0;
    }
  }
  return 1;
}

/* fill the array with random numbers */
void fillArray(int array[], int size){
  int count = 0;
  int random;
  while(count <= size){
     random = rand() % size;
     count = count + 1;
  }
}

//function to merge two arrays
void merge(int arrA[], int arrB[], int arr_size, int num_threads){


            int num;
            int k = 0;
            for (int i = 0; i < arr_size; i++){
                        num = 0;
                        for (int j = 0; j < num_threads; j++){
        if (indice[j][0] <= indice[j][1]){
          if (num < 0){
            num = arrA[indice[j][0]];
          }
          if (num >= arrA[indice[j][0]]){
            num = arrA[indice[j][0]];
            k = j;
          }
        }
    }
    indice[k][0] = indice[k][0] + 1;
    arrB[i] = num;
            }
}


//function to check whether the array is sorted correctly or not
int isSorted(int arr[], int size){
            for(int w = 0; w < size; w++){
      if(&arr[w] == NULL){
              printf("Error , array content is NULL %d", w);
          return -1; /* Indicate error has occured */
    }
  }

        for(int i = 0; i < size-1; i++){
                if(arr[i] > arr[i+1]){
                        return 0;
                }
        }
        return 1;
}

void *threadedInsertion(void *i){
        int start = indice[*((int*)(&i))][0];
  int end = indice[*((int*)(&i))][1];

        InsertionSort(threadArray, start, end);
        pthread_exit(0);
}

void *threadedQuick(void *i){
        int start = indice[*((int*)(&i))][0];
  int end = indice[*((int*)(&i))][1];

        //QuickSort(threadArray, start, end);
        QuickSort(threadArray, start, end, sizeof(threadArray)/sizeof(threadArray[0]));
        pthread_exit(0);
}

int main(int argc,char *argv[]) {
        int idx;
        int arr_size = atoi(argv[1]);
        int num_threads = atoi(argv[2]);
        algorithm = argv[3];

        //seqArray = new int [arr_size];
        //finalArray = new int[arr_size];
        //threadArray = new int [arr_size];
        seqArray = (int*)malloc(arr_size * sizeof(int));
        finalArray = (int*)malloc(arr_size * sizeof(int));
        threadArray =  (int*)malloc(arr_size * sizeof(int));

        //argument validation
    if(argc != 4){
        fprintf(stderr, "ERROR: Expected 4 arguments, provided: %d\n", argc);
        return -1;
    }else if(argc == 4){
        if(isNumber(argv[1]) && isNumber(argv[2]) && !isNumber(argv[3])){
            if(arr_size < 1 || arr_size > 100000000){
                fprintf(stderr, "ERROR: Array size must be between 1 and
100,000,000");
                return -1;
            }

            if(num_threads < 1 || num_threads > 16){
                fprintf(stderr, "ERROR: number of threads must be between 1 and
16");
                return -1;
            }

            if(*algorithm ==  'I' || *algorithm == 'i'){
                *algorithm = 'I';
            }else if(*algorithm == 'Q' || *algorithm == 'q'){
                *algorithm = 'Q';
            }else{
                fprintf(stderr, "ERROR: enter (I) for InsertionSort or (Q) for
Quicksort");
                return -1;
            }
        }else{
            fprintf(stderr, "ERROR: Invalid input for: array size (1 - 100000000),
number of threads (1-16), or Quicksort (Q) or Insertionsort (I)");
            return -1;
        }
    }

    //set the indices and print them
    idx = arr_size/num_threads;
    for (int i = 0; i < num_threads; i++){
        indice[i][0] = i * idx;
        indice[i][1] = (i+1) * idx - 1;

        printf("Indices: %i, ", indice[i][0]);
        printf("%i\n", indice[i][1]);
    }
    //fill array with arr_size number of random elements
    fillArray(seqArray, arr_size);

    //set time to start counting
    SetTime();

    //sort based on user input on algorithm (Q or I)
    if (*algorithm == 'Q'){
        for (int i = 0; i < num_threads; i++){
            QuickSort(seqArray, indice[i][0], indice[i][1],
sizeof(seqArray)/sizeof(seqArray[0]));
        }

    }
    else if (*algorithm == 'I'){
        for (int i = 0; i < num_threads; i++){
            InsertionSort(seqArray, indice[i][0], indice[i][1]);
        }
    }

    //combine the sub-arrays back together and re-sort
    merge(seqArray, finalArray, arr_size, num_threads);

    printf("Completion time for Sorting Sequentially: %ld \n", GetTime());
    if(isSorted(finalArray, arr_size)){
        printf("Sorted Correctly\n");
    } else {
        printf("Sorted Incorrectly\n");
    }

    //fill the threaded array with arr_size number of random elements
 fillArray(threadArray, arr_size);

    //set time to start counting
    SetTime();

    //initialize pthread variables
    pthread_t threads[num_threads];
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    int rc;
    int jc;
```

```
      //depending on the user input for algorithm, run the threaded insertion sort or
      //threaded quick sort
      for(int i = 0; i < num_threads;i++){
            if (*algorithm == 'I'){
                  rc = pthread_create(&threads[i],&attr,threadedInsertion,NULL);
            }else if (*algorithm == 'Q'){
                  rc = pthread_create(&threads[i],&attr,threadedQuick,NULL);
            }
            if(rc){
                  fprintf(stderr, "Error: pthread_create rc: %d\n", rc);
            }
      }

      //will return 0 if the thread has terminated successfully
      for (int i = 0; i < num_threads; i++){
            pthread_join(threads[i],NULL);
            if(jc){
                  fprintf(stderr, "Error: pthread_join jc: %d\n", jc);
            }
      }

      //combine the sub-arrays back together and re-sort
      merge(threadArray, finalArray, arr_size, num_threads);

      printf("Completion time for Multi-Threaded Sort: %ld \n", GetTime());
    if(isSorted(finalArray, arr_size)) {
        printf("Sorted Correctly\n");
    }else{
        printf("Sorted Incorrectly\n");
    }



      //Wipe memory
      memset(seqArray,0,arr_size);
      memset(finalArray,0,arr_size);
      memset(threadArray,0,arr_size);

      return 0;
}
```

Makefile :

```
A3: main.c
	g++ -O3 main.c -lpthread -o main

clean:
	-rm main

go:
	./main 10000 2 I
	./main 100000 2 I
	./main 300000 2 I
	./main 100000 4 I
	./main 1000000 2 Q
	./main 10000000 2 Q
	./main 100000000 2 Q
	./main 1000000 4 Q
	./main 10000000 4 Q
	./main 100000000 4 Q
```