Operating Systems Homework 3

Jessica Jassal, jj5dp@virginia.edu

I completed the assignment for finding the max of a set of integers using threads.

1. Problem:

The problem for homework 3 was to find the max of a given set of integers using threads. The method is called parallel, binary reduction in which N/2 threads are used to compare the values for N integers. The threads were to be reused and the code execution was to be synchronized using a barrier.

2. Approach

Two types of data structures were used to implement the solution, arrays and structs. Arrays were used for many purposes such as holding the parameters for the comparison function and storing the tids of the threads. Most importantly an array was used to hold the integer values that were going to be compared, and ultimately the max number was going to be held at index 0 of that array. The structs were used to group the parameters to be passed to the comparison function. The struct contained a pointer to the array of integers, a pointer to the barrier instance, the tid of the thread using the particular struct, and two index values.

The main algorithm of this solution was implemented in the comparison function called by each thread. The comparison function took in an array of integers and two index values (one for each integer being compared) as parameters. The integers at the first and second index values were compared and the greater value was stored at the first index. In approaching the problem this way, the index values were recalculated to reference the maxes of the previous round. This way, in the following rounds these maxes will be compared eventually leaving one integer at index 0, which will be the max of the whole set.

Each comparison of two integers was done in a thread. To insure that all the comparisons of a round were completed before the thread moved on to next round, a barrier wait() was called at the end of the comparison. After the appropriate amount of threads in that round finished comparing, the barrier released the threads. Since the number of threads active in each round were half the amount of the previous round, the barrier primitive was reset to reflect that. Each round, only the threads with a tid less than the number of threads active executed the comparison code.

The barrier wait() method was implemented so that a certain number of threads were placed on a wait queue. When placed on the queue, the thread’s execution is paused until it is released. When the last thread called the barrier wait(), the all threads on the queue were released and the barrier primitive was divided by two. To insure that the barrier wait() executed correctly, a semaphore was used to verify that a thread had been added to the wait queue before signal was called.

Finally, the main function set up the parameters and created the threads that were going to do the comparison. Join was called on all the threads before printing the max value from index 0 of the integer array.

3. Problems Encountered

Some problems I had was understanding how the threads shared global variables and which variables needed to be thread specific. However, after trying different ways of passing variables to the comparison function, either through the struct or as a global variable, I realized that certain things, like the number of threads that round, had to be updated individually for each thread. This is true for my implementation of the solution and may not apply to other implementations. Other than that I did not run into any major problems.

One other issue I had at the beginning was my program would finish before fully executing all the threads. This is where I understood the value of join(). After creating the threads, the main() would also run as its own thread. If the main() thread finished before the comparisons were done, the whole program would exit. After adding the join(), the comparison threads finished execution before the main() finished and the correct max was printed.

4. Testing

To test my solution I wrote a quick python script to randomly generate numbers within a min and max range. I also made it so the size of the set varied from 2 to 8192. This way I was able to test an unordered set of integers of varying sizes. For all the inputs, my tests passed. I did not run into any calculation errors, and the only problems I had with code are covered in the previous section.

5. Conclusion

I leaned that using threads to solve problems with large inputs can greatly increase the speed of the computation. Had this solution been implemented in a traditional sequential manner, the runtime of the program would have been much higher. Also, understanding how threads interact with each other and with relevant data took some time to understand, but this assignment helped put the concepts to use and clear up some confusion on the topic.

Pledge: I have neither given nor received aid on this assignment

Source Code:

```
/*
 *
 * Jessica Jassal
 * Operatig Systems Homework 3
 * Spring 2018
 *
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

using namespace std;

///////////////////// CREATE BARRIER //////////////////////

class Barrier {
  private:
    int value, init;
    sem_t mutex;
    sem_t waitq;
    sem_t throttle;
  public:
    Barrier(int in);
    void wait(int newVal);
};
```

```
// BARRIER CONSTRUCTOR
Barrier::Barrier(int in){
  init = in;
  value = init;
  sem_init(&mutex, 0, 1);
  sem_init(&waitq, 0, 0);
  sem_init(&throttle, 0, 0);
}

// WAIT() METHOD
void Barrier::wait(int newVal){
  sem_wait(&mutex);
  value--;
  if(value != 0){ // NOT LAST THREAD
    sem_post(&mutex);
    sem_wait(&waitq);
    sem_post(&throttle);
  } else { // LAST THREAD
    int i = 0;
    for (i = 0; i < init-1; i++){ // RELEASE WAITING THREADS
      sem_post(&waitq);
      sem_wait(&throttle);
    }
    init = newVal;
    value = init;
    sem_post(&mutex);
  }
}

//////////////////// GLOBAL VARS //////////////////////

typedef struct{
    float * array;
    pthread_t tid;
    int first;
    int second;
    int numThreads;
    Barrier * bar;
}thread_arg;

int count = 0;
unsigned int numOfRounds = 0;

//////////////////// HELPER FUNCTION FOR ROUND NUMS //////////////////////

unsigned int getLog2 (unsigned int n){
    unsigned int logVal = 0;
    --n;
    while (n > 0) {
        ++logVal;
        n >>= 1;
    }
    return logVal;
}
```

```
//////////////////// THREAD FUNCTION //////////////////////

void *compare(void * arguments){
  thread_arg * arg = (thread_arg *) arguments;
  int i = 1;
  for(i = 1; i <= numOfRounds; i++){

    // DECREASE NUMBER OF THREADS TO USE
    if((int)arg->tid < arg->numThreads){

      // COMPARING
      if (arg->array[arg->first] < arg->array[arg->second])
       arg->array[arg->first] = arg->array[arg->second];

      // BARRIER WAITING
      arg->bar->wait(arg->numThreads/2);

      // SETTING NEW ARG VALS
      arg->first = arg->first*2;
      arg->second = arg->second*2;
      arg->numThreads = arg->numThreads/2;
    }
  }
}

int main(){

  /////////  READ IN INPUT  //////////////

  char buffer[5];
  float * nums = new float[4*8193];
  count = 0;

  // REAN IN INPUTS
  while (1){
    if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
      break;
    }

    if (sscanf(buffer, "%f", (nums + (count))) != 0) {
      count++;
    } else {
      break;
    }
  }

  /////////  SET UP AND CREATE THREADS  //////////////

  // DECLARE VARIABLES
  int numOfThreads = count/2;
  numOfRounds = getLog2(count);
  pthread_t threadsArr[count/2];
  thread_arg argsArr[numOfThreads];
  Barrier * b = new Barrier(numOfThreads);
  pthread_attr_t attr;
  // SET ARGUMENT VALS AND CREATE THREADS
  int i = 0;
  int j = 0;
  pthread_attr_init(&attr);
  for (i=0; i < count-1; i+=2){
    argsArr[j].array = nums;
    argsArr[j].tid = j;
    argsArr[j].first = i;
    argsArr[j].second = i+1;
    argsArr[j].numThreads = numOfThreads;
    argsArr[j].bar = b;

    // CREATE THREADS
    pthread_create(&threadsArr[j], &attr, compare, &argsArr[j]);
    j++;
  }

  // WAIT FOR THREADS TO TERMINATE
  int l;
  for (l=0; l < count/2; l++){
    pthread_join(threadsArr[l], NULL);
  }

  // PRINT MAX VALUE
  printf("%d \n", int(nums[0]));

  free(nums);
  free(b);
  return 0;
}
```