ITP 30002-01 Operating System, Spring 2020

Homework 1

21600212 Nam Jinwoo | 21600212@handong.edu

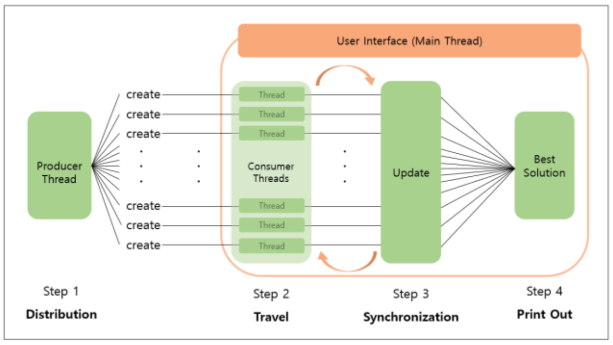
1. Introduction

Traveling Salesman Problem (TSP) is a well-known problem. A graph of the distance between cities is given in TSP problem. A Salesman must pass through every city on the shortest route. The most widely known method is Brute Force algorithm. However, Brute Force algorithm check all passible cases. Since this takes to much time, in this paper, I propose a faster method called Multi-Thread Traveling Algorithm (MTTA). To achieve faster result, a parallel structure using multi-threads has been designed. Working time is reduced because multiple processes perform tasks simultaneously. There were three key problems to design MTTA.

1. How to distribute tasks to threads.
2. How to synchronize shared resources.
3. How to build user interface.

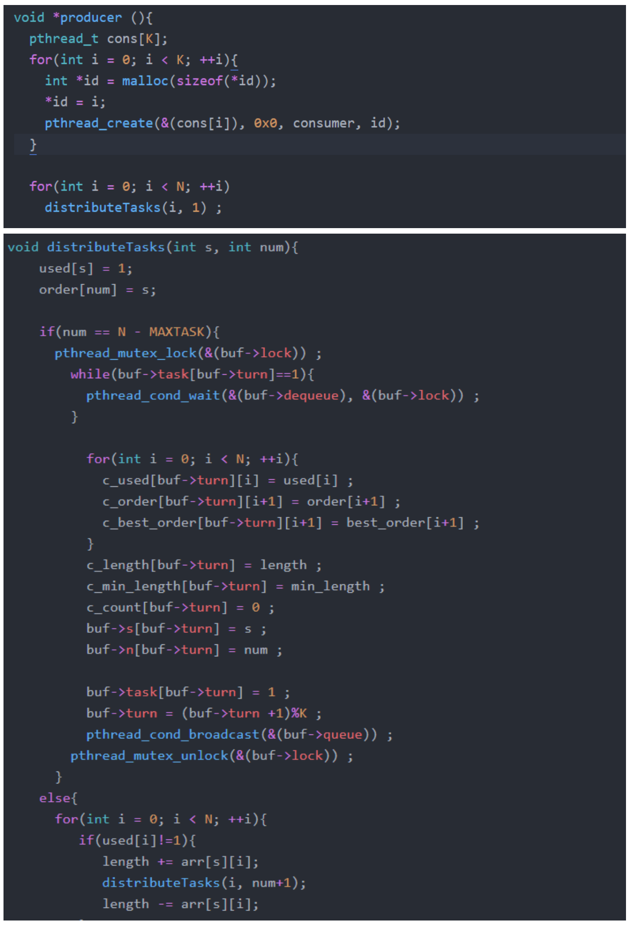
I describe ways to solve these problems on the Approach section. Furthermore, I will explain how I evaluate and measure my algorithm. At the last of the paper, I also mention the direction in which I should reinforce in the future.

2. Approach



< Figure 1. Overall workflow of the MTTA >

Figure 1. explains the overall workflow of MTTA(Multi-Thread Traveling Algorithm). First, producer thread distribute tasks and create consumer threads to solve distributed prefix task. Second, after receiving the prefix task, consumer threads search all possible cases with brute-force algorithm. And then, synchronize it's own best solution to shared resource. Repeat step2, 3 until all the task is done. Meanwhile, main thread provides an interface to monitor those progress.



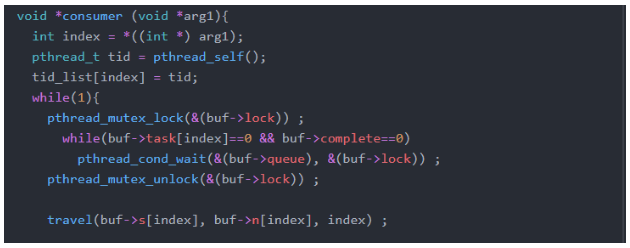
```
void *producer (){
  pthread_t cons[K];
  for(int i = 0; i < K; ++i){
    int *id = malloc(sizeof(*id));
    *id = i;
    pthread_create(&(cons[i]), 0x0, consumer, id);
  }

  for(int i = 0; i < N; ++i)
    distributeTasks(i, 1) ;
```

```
void distributeTasks(int s, int num){
    used[s] = 1;
    order[num] = s;

    if(num == N - MAXTASK){
      pthread_mutex_lock(&(buf->lock)) ;
        while(buf->task[buf->turn]==1){
          pthread_cond_wait(&(buf->dequeue), &(buf->lock)) ;
        }

          for(int i = 0; i < N; ++i){
            c_used[buf->turn][i] = used[i] ;
            c_order[buf->turn][i+1] = order[i+1] ;
            c_best_order[buf->turn][i+1] = best_order[i+1] ;
          }
          c_length[buf->turn] = length ;
          c_min_length[buf->turn] = min_length ;
          c_count[buf->turn] = 0 ;
          buf->s[buf->turn] = s ;
          buf->n[buf->turn] = num ;

          buf->task[buf->turn] = 1 ;
          buf->turn = (buf->turn +1)%K ;
          pthread_cond_broadcast(&(buf->queue)) ;
        pthread_mutex_unlock(&(buf->lock)) ;
      }
    else{
      for(int i = 0; i < N; ++i){
         if(used[i]!=1){
            length += arr[s][i];
            distributeTasks(i, num+1);
            length -= arr[s][i];
```

< Figure 2. producer thread & distribute function>

2.1 How to distribute task

In order to distribute subtasks, first, MTTA creates consumer threads. And then run the recursive Brute Force algorithm. In the function, when the size of the subtask reaches 11!, producer thread distribute task through shared resources to consumer threads. In order to efficiently allocate tasks, shared resources were created for each consumer threads. After checking whether each shared resource has a task or not, fill in the new task if not.

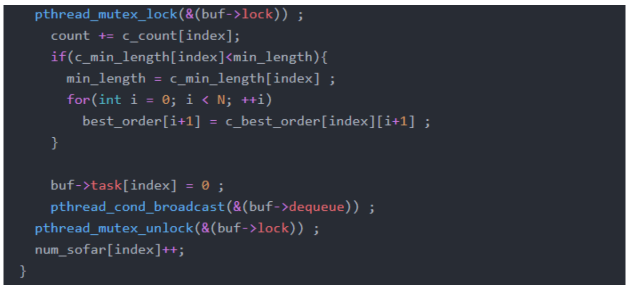


```
void *consumer (void *arg1){
  int index = *((int *) arg1);
  pthread_t tid = pthread_self();
  tid_list[index] = tid;
  while(1){
    pthread_mutex_lock(&(buf->lock)) ;
      while(buf->task[index]==0 && buf->complete==0)
        pthread_cond_wait(&(buf->queue), &(buf->lock)) ;
    pthread_mutex_unlock(&(buf->lock)) ;

    travel(buf->s[index], buf->n[index], index) ;
```

< Figure 3. consumer thread>

2.2 How to synchronize shared resources

As same as producer thread, consumer thread also has pthread_cond_wait function. Consumer threads wait until new task is arrived. These kind of updating shared resource behavior can easily occur concurrency problems. Some times it could occur serious deadlock problem. To prevent such problems in advance, only one thread should approach to critical section at a time. That is why Fiqure 2. pthread_mutex_lock function is needed.



```
  pthread_mutex_lock(&(buf->lock)) ;
    count += c_count[index];
    if(c_min_length[index]<min_length){
      min_length = c_min_length[index] ;
      for(int i = 0; i < N; ++i)
        best_order[i+1] = c_best_order[index][i+1] ;
    }

    buf->task[index] = 0 ;
    pthread_cond_broadcast(&(buf->dequeue)) ;
  pthread_mutex_unlock(&(buf->lock)) ;
  num_sofar[index]++;
}
```

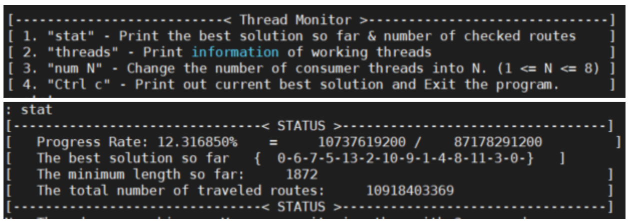
< *Figure 4. consumer thread synchronization*>

After traveling subfix routes, consumer thread needs to update latest information. As same as last time, only one thread at a time should be guaranteed access to the critical section using the mutex.

2.3 How to build user interface

It may seem quite difficult to receive commands from user when code is running. However, since my model MTTA has multi threads, main thread can easily receive commands.

To enable the user to monitor progress three functions were inserted into the user interface. 1. Print out current status. 2. Print out threads informations. 3. Change the number of threads. While providing convenience to users, the three functions also help me evaluate my algorithms.



```
[------------------------< Thread Monitor >---------------------------]
[ 1. "stat" - Print the best solution so far & number of checked routes    ]
[ 2. "threads" - Print information of working threads                      ]
[ 3. "num N" - Change the number of consumer threads into N. (1 <= N <= 8) ]
[ 4. "Ctrl c" - Print out current best solution and Exit the program.      ]
```

```
: stat
[----------------------------< STATUS >--------------------------------]
[   Progress Rate: 12.316850%    =     10737619200 /    87178291200        ]
[   The best solution so far   {  0-6-7-5-13-2-10-9-1-4-8-11-3-0-}   ]
[   The minimum length so far:     1872                                ]
[   The total number of traveled routes:     10918403369               ]
[----------------------------< STATUS >--------------------------------]
```

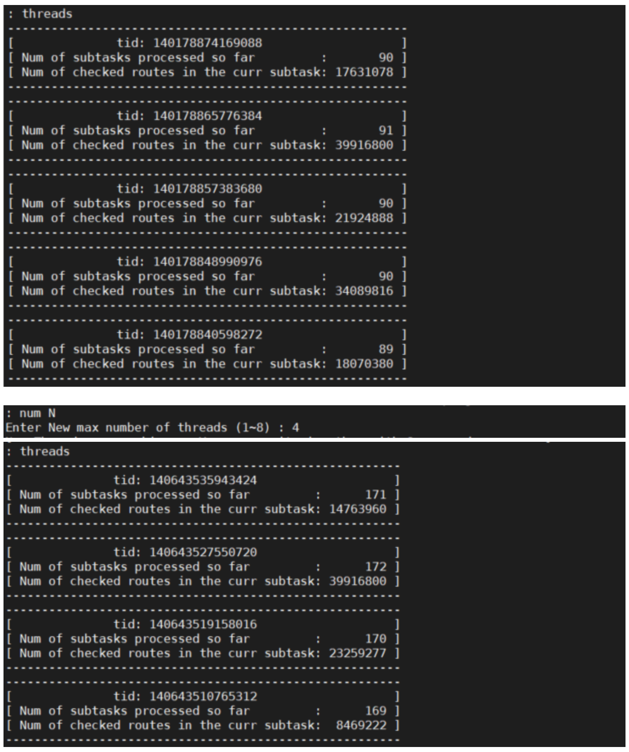
< *Figure 5,6 User Interface, stat command example*>

3. Evaluation

In order to check whether I fullfill requirements or not, I print out some examples using user interface functions.

Figure 5 shows user main interface that gives three options with exit guide. When user enter “stat”, it shows user best solution so far and total number of traveled routes with progress rate.

Figure 7, 8 shows us that threads are working simultanously. And when user input new value of the number of threads, it changes the number. we can check number of threads changed from 5 to 4.

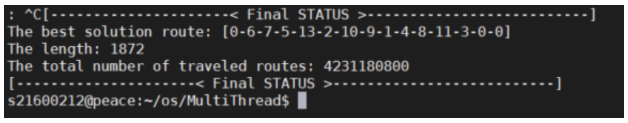


```
: threads
------------------------------------------------------
[               tid: 140178874169088                  ]
[ Num of subtasks processed so far          :      90 ]
[ Num of checked routes in the curr subtask: 17631078 ]
------------------------------------------------------
------------------------------------------------------
[               tid: 140178865776384                  ]
[ Num of subtasks processed so far          :      91 ]
[ Num of checked routes in the curr subtask: 39916800 ]
------------------------------------------------------
------------------------------------------------------
[               tid: 140178857383680                  ]
[ Num of subtasks processed so far          :      90 ]
[ Num of checked routes in the curr subtask: 21924888 ]
------------------------------------------------------
------------------------------------------------------
[               tid: 140178848990976                  ]
[ Num of subtasks processed so far          :      90 ]
[ Num of checked routes in the curr subtask: 34089816 ]
------------------------------------------------------
------------------------------------------------------
[               tid: 140178840598272                  ]
[ Num of subtasks processed so far          :      89 ]
[ Num of checked routes in the curr subtask: 18070380 ]
------------------------------------------------------
```

```
: num N
Enter New max number of threads (1~8) : 4
```

```
: threads
------------------------------------------------------
[               tid: 140643535943424                  ]
[ Num of subtasks processed so far          :     171 ]
[ Num of checked routes in the curr subtask: 14763960 ]
------------------------------------------------------
------------------------------------------------------
[               tid: 140643527550720                  ]
[ Num of subtasks processed so far          :     172 ]
[ Num of checked routes in the curr subtask: 39916800 ]
------------------------------------------------------
------------------------------------------------------
[               tid: 140643519158016                  ]
[ Num of subtasks processed so far          :     170 ]
[ Num of checked routes in the curr subtask: 23259277 ]
------------------------------------------------------
------------------------------------------------------
[               tid: 140643510765312                  ]
[ Num of subtasks processed so far          :     169 ]
[ Num of checked routes in the curr subtask:  8469222 ]
------------------------------------------------------
```

< *Figure 7,8 threads, num N function*>



```
: ^C[--------------------< Final STATUS >-------------------------]
The best solution route: [0-6-7-5-13-2-10-9-1-4-8-11-3-0-0]
The length: 1872
The total number of traveled routes: 4231180800
[--------------------< Final STATUS >-------------------------]
s21600212@peace:~/os/MultiThread$ 
```

<*Figure 9. sigint handler*>

MTTA also can handle with unexpected termination. When termination signal came, it display to user latest solution. I implement this with signal handler function.

4. Discussion

The most interesting result of this assignment was efficiency of multi threading. I expect more threads more fast, but it was not. I’ve got interested in how to maximize efficiency. In the future, I want to find out relationship between the number of threads and eficiency, and I also want to implement various methods to improve multi threading like thread pools. And also I don’t think I fully understand canceling and terminating a thread when it’s running. It would be great to supplement that part my self.

5. Conclusion

HW3 seemed easy comare to multi process assignment. Because last time, data sharing was most difficult part to me. However, at this time, preventing threads from approaching critical section at the same time was as difficult as last time. In order to solve three difficulties of this probem, I made lot’s of test c codes. Implementation of those functions helps me a lot with understanding multi threading problems. I could feel a huge difference between knowing the theory and acturally implementing it.