ITP 30002-02 Operating System, Spring 2020

Homework 3

Tae Eun Kim 21400217, 21400217@handong.edu

1. Introduction

The goal of this homework was to implement a TSP solving program by multi-threading. In this program, there are three kinds of threads. First, the main thread that interacts with the user. Second, the producer thread that produces the subtask. Third, the consumer thread that works on the subtask.

While the producer and the consumer threads work on solving the problem, user may interact with the program. User can give three commands, to view the current progress, to view the information of working threads, and to increase or decrease working threads. When the all the job is done, or the user terminates the program, it prints out the best result and terminates.

2. Approach

2.1 Overview

Here is the simple overview of my approach. In order to implement the producer-consumer mechanism, I used a bounded buffer structure as the data exchange platform. The producer produces subtasks and puts it in the bounded buffer. The consumer takes the subtask from the buffer.

This is the workflow. With the initial thread population given by the user, the main thread creates the producer and the consumer threads. After each subtask is finished, the consumer updates the data in global variable if a better solution is found. Then, it takes another subtask from the buffer and continues. This is repeated until the producer is done producing subtasks and consumers are finished with their final subtask.

2.2 Assigning subtasks

Since it is not multi-processing, I can't dump the whole copy of the data to assign a subtask as I did in hw2. So, I implemented a data structure that holds a set of data that can represent the subtask, "path_data". It holds data that defines a path, like length, path[], and used[]. It also contains data that is thread-specific, like numbers of subtasks done. Since there is only one producer, it has its unique "path_data" instance. For the consumer, each consumer thread is given an index from 0 to 7 at their creation. They can access their instance of "path_data" with this index

This is how subtask is given and taken. The producer fills up the path[] up to (n-11)th city, also updating the used[], and length. Then it queues it into the buffer by copying the whole instance into the slot in the bounded buffer. Now, consumer can dequeue from the bounded buffer by copying out "path_data" instances. Then the consumer can start from the (n-12)th city.

2.2.1 Bounded buffer

Assigning the subtask is mediated by the bounded buffer. The bounded buffer is initialized with the size of 16. It is to enable canceled subtasks to enter without delay. It can be manipulated with two functions, queue and dequeue. It provides synchronicity by maintaining one mutex and two conditional variables, each for queue and dequeue. If the producer wants to queue but the buffer is full, it will wait in the queue_cv. When a consumer dequeues, it will inform the queue_cv, and the producer will be able to check again whether if it can queue this time.

2.3 Synchronization

To protect the critical sections, like reading and writing on the shared data, I used pthread_mutex to use it as a lock. For the global variable that contains the data of the best path and the thread populations, I used a "global_lock". For each of the "path_data" instances of consumers, I used "local_lock"s. For the bounded buffer, there was a lock inside the buffer data structure.

To prevent the deadlock situation, I maintained the same ordering of the nested locks, global-then-local. I also had to free all the locks when the thread was to be killed.

2.4 User interactions

2.4.1 Stat & Threads

The first two operations are the relatively easy ones. First, "stat" prints out the best solution. Since the there may be better solution in the working subtasks, I need to check the consumer threads' data too. As we discussed in section 2.3, these data are protected by some locks. In order to access them, "stat" first acquires the global_lock, then acquires local_locks one by one. Then it compares the lengths and updates the best path if necessary. Finally, it can print out the best path.

The operation "thread" only acquires local_locks. It views and prints the thread ids, number of subtasks finished, and the routes searched in current subtask.

2.4.2 Increase / Decrease threads

When the user types in "num N", the thread population must be adjusted to the given N. Then, we have two cases. Increasing and decreasing the thread population. Increasing is rather easy. I just need to initialize more "path_data", and create more threads according the given N. New threads are given the remaining indexes.(sec. 2.2)

Decreasing the thread population is a challenge because we need to preserve the subtask. We first need to cancel the threads that exceed the new population. However, their data still remains in the "path_data" they accessed. With this data, we can restore the initial subtask that the consumer thread received. Logic is simple. The initial length can be easily retrieved by keeping a separate variable in the "path_data". Path can be retrieved by leaving only the first n-11 cities in the path[], and mark used[] according to the path[]. Now this "path_data" instance becomes the original subtask form, and we can queue this into the buffer, for the remaining consumer threads.

2.5 Termination

2.5.1 Thread Canceling

Safely canceling the threads is an issue. In this program, decreasing thread population requires thread canceling. I needed to make sure that the thread releases all its locks before it dies, because they will be used by other processes. So, I pushed a manual clean-up function in the function of consumer thread. It forces it to release all its locks when it is asked to die. This way, I could prevent the deadlocks.

2.5.2 Terminating program

This program in terminated under two situations. First, by user giving termination signal with ctrl+c, and second, by finishing all the task.

First case, I implemented a handler function that updates all data, prints the best result, terminates the threads and program. Then I set the termination signal to invoke this handler function. When the user gives the termination signal, the handler function is called automatically.

For the second case, I let the producer thread to terminate the program. When it is done producing all the subtasks, it waits until the all the consumers are also done with their subtasks. I made a variable in the bounded buffer that counts the threads waiting for the queue operation, inspired by the reader-writer problem. If this number reaches the number of consumer thread population, producer calls the handler function to terminate the program.

3. Evaluation

3.1 Functionality

I could check that all the user interactions were working fine. Threads were well controlled, and no deadlock was occurring under different scenarios.

3.2 Performance

I tested and compared the performance by the number of checked routes differing by hardware, implementations, and concurrency. Here is the result.

Threads/ processes	Multiprocess 40 cores	Multithread 40 cores	Multithread 4 cores
1	250,395,737	227,501,222	65,623,593
2	524,178,254	447,703,187	60,925,077
4	929,627,615	258,783,627	44,494,398
8	1,496,768,992	363,026,675	46,495,528

We can see that, unlike multiprocessing, multithreading doesn't always promise the performance. In some cases, concurrency led to lack of performance. The inefficient synchronizing mechanism can be another cause.

4. Discussion

Having the experience with both multiprocessing and multithreading, I now have a better picture of concurrency and synchronicity. Although multithreading seems more convenient and requiring less space, it is obvious that we need to put extra effort to maintain synchronicity and prevent deadlocks. Moreover, if not implemented carefully, it can lead to poor performance due to overheads in synchronization.

I first had hard time with the "path_data", because I used pointers for the path[] and used[]. It got interfered by other threads and the data was broken. So, I statically allocated the integer arrays inside the data structure and it solved it. Being careful to not make a deadlock was a challenge too. I had to keep in mind how all the locks are used in other threads. Keeping a consistent order of locks helped.

There was an interesting behaviour in the decreasing threads operation. Always, the first decrease took very long, about 4 seconds. However, after first time, it worked very fast. I fixed the first delay, because it was waiting to queue. I modified it to queue into extra space that normal queue operations would not access. However, I couldn't figure out why it was only the first time it took such a long time.

For suggestions, I think this can be solved with genetic programming approach. Each thread may be a descendant with genes represented as the order of cities. After each mutation, the threads with shorter path will survive duplicated to make more population. I think it will be very interesting.

5. Conclusion

I built a program that solves a TSP with multithreading. I implemented it using producer-consumer mechanism with a bounded buffer. I found out that multithreading would not perform great if it is not designed carefully.