ITP 30002-02 Operating System, Spring 2020

Homework 2

Eunjun Jang (21800633), jangej1031@naver.com

1. Introduction

Traveling Salesman Problem (TSP) is a famous NP-Completeness problem. TSP is finding the route with the shortest distance traversing all cities.

In this paper, I will try to solve the TSP using Multi-processing in the Linux environment. To simplify the problem, I assumed that it is an undirected graph. To implement TSP in parallel (PTSP), create a child process using Fork and Unnamed Pipe, divide them into subtasks, and collect them in the main process. Subtasks limit the number of search cases up to 12!. When a Terminate signal is received, PTSP prints the minimum value at that point.

To implement PTSP, I first implement a TSP algorithm using basic Depth First Search (DFS) in graph and create child processes through Fork at the point where the number of case becomes 12!, so that the created process can search deeper and the parent process moves on to the next path. In the child process, the number of checked paths, the shortest distance, and the shortest path are written through the pipe. Also, when you need to terminate by the SIGINT, child process writes the intermediate result to the pipe. More details will be covered in the Approach section.

This PTSP can solve the TSP problem very quickly while using all the CPU's resources.

The other aspects of this PTSP will be discussed in the discussion section.

2. Approach

First of all, I will define some symbols. M is the adjacency matrix. It is an undirected complete graph. N is the number of vertices. p is N integers, current sequence of city. d is the current distance. min_p is a p which is shortest distance.

2.1. Subtask Distribution

The input of PTSP is i, which is index of path which have to decide. In the existing DFS-based TSP algorithm, Fork was used to divide subtasks and pass tasks to children. When Fork is executed, the memory of the process is copied as it is, so the parent process does not have to go through the process of passing values to the child process. It is determined sequentially from the first p in the parent process, and forks at the point where the number of child path cases reaches 12!. If it is time to decide the city to enter the i-th path, the number of possible cases from i is i! to be. To make the child process number of case to 12!, the parent process needs to set the (N-13)th city and then fork it. In other words, the parent process forks at (N-13) so that the number of child process cases is 12!.

When the number of child processes reaches the maximum creation limit, the parent process yields CPU resources to other processes through usleep function until any child process dies.

2.2. IPC Using Unamed Pipe

An unnamed pipe is used for communication between the parent and child processes. The child process writes the results it finds through a pipe. The parent process receives the result written by the child process by reading the pipe. PTSP creates only one pair of pipe to communication at the beginning. Using fork copies pipe file description, so child can use pipe. There is no need to create all pipes by child. A series of message structures are needed to communicate properly with each other. The message structure of PTSP is like this:

<count>, <dist>, <best path string> \n

Count is the total number of paths the child process has searched. Dist is the shortest distance, and last is the shortest distance path string. The child process uses sprintf and write to write the result to the pipe. The parent process reads the pipe byte by byte until a newline character is encountered. After that, we divide the message into three parts using the strtok function. Each part uses sscanf to store the values. In the case of path string, it is separated by space. It is decomposed again using strtok and stores the value.

2.3. Result Printing

When the child process checks all paths, it passes the result through the pipe. In the parent process, the pipe is read through SIGCHLD handler to receive the result written by the child process.

When a Terminate signal is received, the minimum result can be output up to that point. When SIGINT or SIGTERM occurs, parent process read the pipe from all currently created children to receive the result. When several children are terminated at the same time, there is a phenomenon that SIGCHLD cannot be properly propagated to parent process. Therefore, for SIGINT and SIGTERM, ignore SIGCHLD and the waitpid function is used in the loop until all child results are received without ignore. When SIGINT occurs, it is propagated to all child

processes, and the result is passed through the pipe from the child SIGINT handler. In contrast, SIGTERM doesn't propagate to child processes. In the case of SIGTERM, the logic that propagates SIGINT to the child processes by the parent process has been added to parent_sigint_handler. In this way, both SIGTERM and SIGINT output properly.

3. Evaluation

Several experiments were conducted to demonstrate the correct implementation of PTSP. All experiments are done in the peace server which is Linux server of CSEE department.

At the first experiment, I will show that PTSP uses all possible CPU cores. To do this, after running PTSP, I will check the number of processes with ps command.

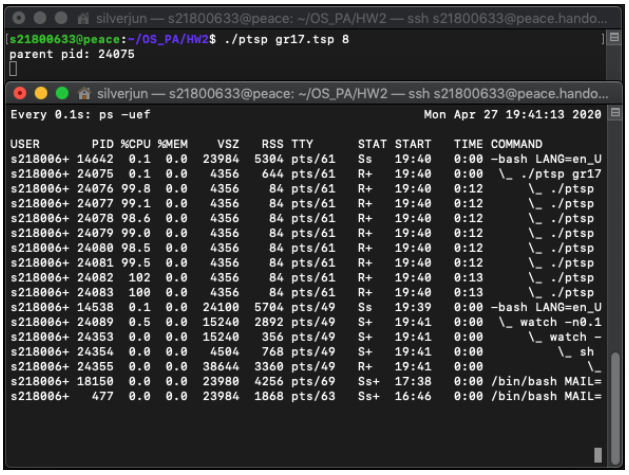

Figure 1. First experiment

In Fig.1, number of children process is 8 which I provided into parent process. Furthermore, CPU usage of each children processes are about 99%. It shows that PTSP highly utilize the CPU resources.

In the second experiment, I will show PTSP distributes subtasks whose number of cases is 12!. To show this, I will print the count returned after the child process ends. Also, I will check the output of the count returned after the child process terminated.

Figure 2. Second experiment

In Fig.2, Each child process returned 479001600 Returned value is 12! which I expected.

The last experiments, I will show that PTSP prints the result when termination signal is occurred.

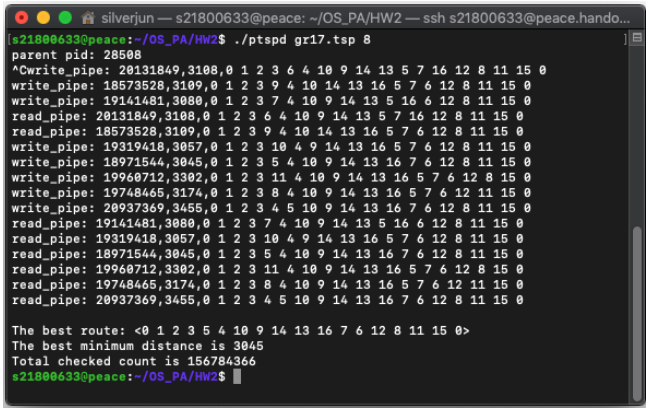

Figure 3. Third experiment

In Fig.3, PTSP successfully showed the intermediate result caused by SIGINT.

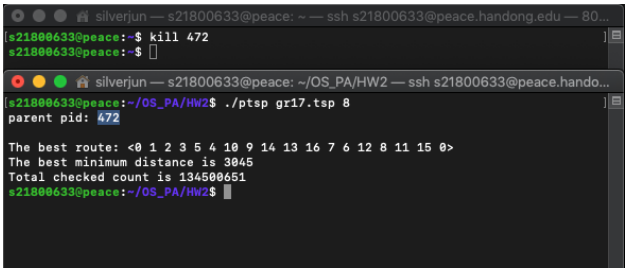

Figure 4. An expriment SIGTERM case

Furthermore, when I pass SIGTERM signal using kill command to PTSP, it showed the result like when SIGINT raised.

4. Discussion

I. divided discussion section into two part: Learning things and Further algorithm.

4.1. Learned Things

In Learning things section, I will describe what I learn through doing this homework.

First thing is about pipe. At the beginning of doing homework, I think that I need to make each pipe by child to communicate each child processes. But I found that this method is quite complex, and I need to figure out. Google said that write and pipe is atomic operation, so they not have kind of a synchronization problem. Therefore, I just made a communication through one pipe. Also, I tested the situation that children process writes a sentence at the same time. But parent process read very well with no corruption.

Second thing is about handling multiple SIGCHLD. I had an experience that SIGCHLD is not working properly when I send a SIGINT signal. I thought that child processes are sending SIGCHLD at the same time. So, parent cannot handle some SIGCHLD. My thought is right. Therefore, I made handling logic into SIGINT handler to receive all child process results using waitpid function. Also, I know that SIGINT affects all child process but SIGTERM only affects parents process. So, in SIGTERM, I need to propagate SIGINT to all child to communication. That is very valuable experience.

4.2. Further Algorithm

If TSP implemented in Dynamic programming (DP), performance will be much better than DFS-based. Because there are many overlapping computation. Because it is an Un-digraph, the computation value is the same because it is the same as the going and coming. Eventually it overlaps. A-D-C and C-D-A have the same distance, but this kind of path are computed multiple times. However, if I use DP, I need a way to share a large array that stores the computed path among child processes. When I fork, I cannot share memory because the physical memory address is different. Even if it does, it can cause bottlenecks due to synchronization. Then, there is a method of passing the computed path to the parent process, but rather, such communication may increase the burden of the parent process, and it is likely that a bottleneck may occur. Then I think it might be better to do it in one process than to create one. I thought that it would be better to implement multi-thread in one process than apply multi-process to DP.

5. Conclusion

In this paper, I proposed a method for TSP algorithm using Multi-processing named PTSP. PTSP showed powerful utilizing CPU resources to compute possible path. PTSP consists with multiple processes. Parent process distribute subtask to child process by forking and receive result through pipe. Child processes explore all possible path in the subtask and write result through pipe. In the case of termination signal, child processes send their intermediate result, and parent receive and show minimum result. There can be more effective way to solve TSP problem. PTSP has some limitations which is cannot compute over 2^64 cases. Because PTSP uses count value with unsigned long long.