ITP 30002-02 Operating System, Spring 2020

Homework 2

Tae Eun Kim 21400217, 21400217@handong.edu

1. Introduction

1.1 Goal

The ultimate goal of this homework was to implement an algorithm that solves a TSP problem through multiprogramming.

To do so, I set up a master-slave relationship between the parent and the child processes, where the parent process assigns a job to the child and later collects the results. The detailed approach is to be explained in section 2.

1.2 Requirements

There are two inputs of the program, the filename of the TSP instance data, and the maximum number of child processes. The file contains data of N cities. The parent process assigns the workload of checking 12! routes to the child by restricting the prefixes of the route. The output of the program is the best route and its length, with the total routes that are checked.

2. Approach

2.1 Overview

Here is an overview of the workflow. First, the parent process finds work for the child by making a prefix. This is done by making a prefix in the global variable and then forking. With different prefixes, the children can have non-overlapping tasks. After forking, the child starts exploring the routes, meanwhile the parent makes another prefix and forks another child. The parent continues to make a prefix and fork until the current child population reaches the limit set by the initial input. To maintain the population, parent will find the next prefix, but will wait for a child to terminate. A child will terminate after it explores all its given routes, sending its best result to the parent via pipe. Then the parent will stop waiting and read the data from the pipe, compare, and update its own best result if necessary. Then it will fork a new child and continue. When the parent is terminated, it kills all its child and reads the all data sequentially, finally producing the best result up to that point. The terminating signal was handled by a customized handler, which enabled the parent and the child to deliver their best solutions, either to the parent or the user.

2.2 TSP

Here I introduce how the parent makes the prefix and how the child explores the routes. Essentially, two tasks are not different. It is finding permutation. Since I have N cities and 12! of child's portion of work, the parent chooses the first N-12 cities, while the child has to find the rest, the order of 12 cities. The parent does not find all prefixes first. Every time the parent finds a new prefix, which happens in a very short time, the child is forked to search 12! routes.

Assigning the prefix and the rest of the work was simple. Selecting the first N-12 cities is same as in sequential programming. However from the (N-12)+1 city, it is given to the child process and parent doesn't care about the rest.

While keeping the data of the prefix, which is the first N-12 cities in the route, in the global variable, I fork a child. Then the child inherits all the data of the prefix, list of cities that has been marked visited or not visited, and the length of the route up to that point. Then the child can continue and complete that route.

2.3 Forking

Although forking was mentioned in section 2.2, I will explain further details of how I managed the forking in this program. Whenever a parent finds a new prefix, it forks a child. However, if the variable that keeps in track of the child population tells that the population is full, it waits by wait(). When a child terminates, the parent can move on.

2.4 Communication

In order for the child to communicate with the parent, I used piping mechanism. The sole purpose of this communication is to send the shortest found route to the parent. When the child is finished exploring all its assigned routes, it writes to the pipe the length of the shortest path, the shortest path, and the number of routes it explored. The format is as follows. "length path num_routes". The path was expressed in a string of the sequence of cities, separated with a comma. For example, "2,4,3,6,8…". After the child is terminated, the parent reads from the pipe. It first receives the string and divides it into length, path, and the num_routes. If the returned length is shorter than the shortest path in the parent's record, the length and the path is replaced with the new ones. The num_routes is added up to the total sum regardless of the length.

2.5 Termination

There are two cases of termination of the program. First is when all the routes are explored, thus all the child processes are already terminated, and the parent returning the shortest path and terminating. Second, which is more likely to happen, is when the user types ctrl+C to terminate the parent process. In this case, the parent has to force the children to stop exploring and return what they have.

It is implemented by defining and assigning a handler that handles the termination signal, which has the code SIGINT. In this handler, it is divided in two cases, when it is the parent, and when it is the child. The child part is simple. It simply writes its shortest path up to that point to the pipe. The mechanism is as same as we discussed in section 2.4. After writing, the child terminates. However, the parent part is little bit more complex. Parent first kills all the child processes with kill() function, with arguments of 0 and SIGINT. This means kill all the processes in the same group with the SIGINT signal. Then, as long as the return value of wait() is not -1, which means there is no more dead child, the parent reads from the pipe repeatedly. After each reading, the parent updates the shortest path if necessary. Lastly, parent prints out the shortest path data and the number of searched routes, and terminates.

3. Evaluation

3.1 Scenarios

To test this program is working correctly, we can think of some scenarios to run. Since the TSP problem is such a complicated one, we do not expect to search to the end of the problem. So, we will mainly focus on the case when the user terminates the program. Therefore, we can make a list of things to check in order to see the requirements are met.

a. Does a child explore all the given 12! routes?

b. When terminated, does a child stop in middle of its search and returns the shortest path up to that point?

c. Does the parent receive data correctly from the child?

d. Does the parent receive data from all children when terminated?

e. Are the data from the child reflected in parent's data?

f. Does the parent maintain the maximum population children?

We cannot check all these requirements with only the final output, which just tells us the shortest path and its length, plus the number of searched routes. Thus, I needed to print some information in the middle of the process. I printed a message whenever a child was forked, I also printed out the data sent by the child and the data received by the parent to compare them whenever a child was terminated. With these printing messages, I ran the program.

I could check that the program forks up to the limited population first, then starts forking after whenever a child terminates itself. When a child terminates on its own, I could check that it traveled all 12! routes by checking the number of explored routes. When I terminated the program, the children, which matches the number of limited populations, all returned a data that contains their shortest path. I could see that it was immediate, and I could see that the number of routes they each checked was less than 12!. When I added up the routes checked by the children, it matched the total routes that the parent reported. Also, the shortest path reported by the child was reflected on the shortest path data of the parent. With different tsp instances and populations of child processes, the program was consistent with its functionalities.

3.2 Performance

The performance of the program can be measured by the routes that are checked in a given time.

3.2.1 relationship with the concurrency

There are 4 cases to compare. First, my program with maximum child populations(12), second, my program with moderate child populations(6), third, my program with one child, and finally the sequential TSP solving program. Given a minute each and the same instance file(gr17.tsp), I measured the performance of each of the runs with the number of routes explored. Results were as expected. 511067896 routes for sequential program, 487903584 with one child, 2610106557 with 6 child, 4474934037 with 12 child. We can see that the relationship between the number routes covered by my program is proportional to the number of children. However, the sequential program seems to have a bit of better performance than the concurrent one with one child. This is because of the overhead caused by the forking and the communication between the parent-child.

3.2.2 Relationship with number of cores

I had three ubuntu machines. One is peace server, which has 40 cores, another is my windows Linux subsystem ubuntu, which has 8 cores, and the ubuntu on repl.it, which has 4 cores. This time, I ran the instance with gr48.tsp and processes of 12 on each of the machines to compare the results. The result was as expected, but slightly different. 1749604857 routes were explored by 40 cores, 274300503 routes by 8 cores, 74455124 routes by 4 cores. The leap from 4 cores to 8 cores was dramatic, it was beyond the ratio of the cores, 4 times. Leap from 8 to 40 was also big, having 5 times of increase. However, the ratio was bigger in the leap from 4 to 8 than from 8 to 40. We can conclude that the increase of cores tend to be less dramatic as it gets larger.

4. Discussion

4.1 What I learned

I learned that the multi programming can really enhance the performance of programs, especially when the job can be easily distributed and independent. It was easy to see the big difference in the numbers or searched routes.

Also, I learned that the parent and the child can communicate through the pipe. I also learned that even though there are many children terminated and waiting for the parent to read their data from the same pipe, all the data is still kept in a buffer, so they are not lost. I would not have learned this with only one child process.

Another thing I learned is to deal with really big computations. I have never experienced computations that takes more than 1~2 minutes. I now know how to divide the task, fork if necessary and collect the data.

4.2 Difficulties

The most difficult part of this assignment was the transfer of the data between parent-child. Since I was not familiar to IPC communications, I needed some time to use read, and write libraries through pipe.

The part that I struggled was that I wanted to have safety in reading and writing, thus putting it in a loop that receives the data until it is sure that it is fully written or read. However, it did not workout well. When I tried to read or write iteratively, It fell into a infinite loop, malfunctioning. So I tried to just read/write at a single try, without iterations. With many tests, it was shown that it was fine. Data was always sent successfully.

I was also concerned that with one pipe, the data might get mixed up by different children if they all terminate and write at once. First, I tried to implement a structure that has as many pipes as the maximum population of the children. However, I could not manage to keep track of who is using what. So, I tried to use only one pipe and see what happens. However, no data was lost. Every data written by the children was received by the parent. Even though there are multiple writes on the pipe, there was a buffer that holds them. So, the parent could receive them sequentially.

Apart from the IPC, the workflow of the whole process was not a familiar one. I first had hard time keeping track of what is happening where. However, with many thinking and writing flowcharts, I could understand the whole process. After understanding, the implementation was only a matter of time.

4.3 Suggestions

I want to improve my program to minimize the overhead. Overhead usually takes place in forking, and transferring the data and updating the data. So, if I can use multithreading next time, I might be able to lessen the communication time. Forking needs to copy all the data of the parent, thus the bigger the data are, the slower is the forking process. That is why threading sounds more appealing.

5. Conclusion

The homework was to implement a program that computes the TSP in a concurrent manner. I used pipe, fork, and signal handling in order to accomplish the task. The performance was relevant to the number of children, and the number of available cores. It was shown that some overhead existed when it was compared to the sequential approach. However, starting from two children, overhead becomes trivial.

I learned how to manage the children process' population, the communications between parent and multiple child processes. I also know how to manage big computation through multiprogramming.

I thank the professor for giving me this opportunity to exercise my experience in system programming and multiprogramming.