ITP 30002-01 Operating System, Spring 2020

Homework 2

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-Process Traveling Algorithm (MPTA). To achieve faster result, a parallel structure using multi-process has been designed. Working time is reduced because multiple processes perform tasks simultaneously. There were three key problems to design MPTA.

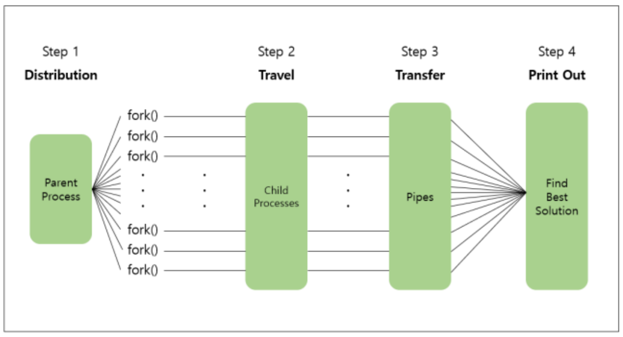
1. How to distribute tasks to child process.

2. How to build pipe for communication between parent and child process.

3. How to handle unexpected suspension command.

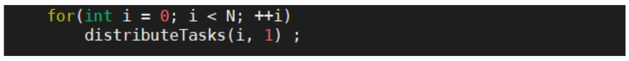
I will describe how to solve these problems on the Approach section. Furthermore, I also propose some other ideas to improve computation performance on the Evaluation section.

2. Approach

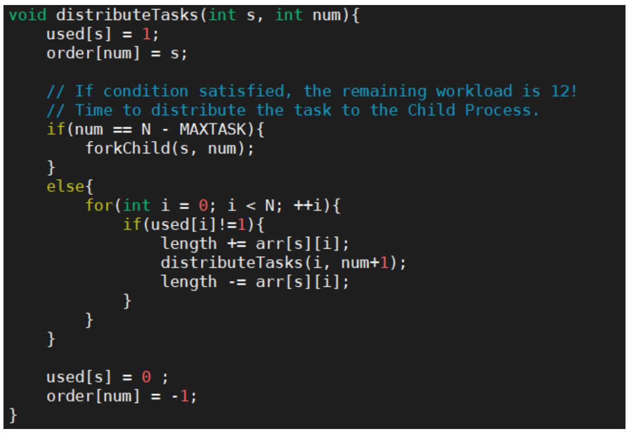


< *Figure 1. Overall workflow of the MPTA* >

Figure 1. explains the overall workflow of MPTA(Multi-Process Traveling Algorithm). First, parent process distribute task and fork() child processes to solve distributed prefix task. Second, after receiving the prefix task, child processes search all possible cases with brute-force algorithm. And then, child processes transfer it's own best solution to parent process using pipes. Finally, MPTA print out best solution includes length of the path, order of the route and the number of counted routes.



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



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

    // If condition satisfied, the remaining workload is 12!
    // Time to distribute the task to the Child Process.
    if(num == N - MAXTASK){
        forkChild(s, num);
    }
    else{
        for(int i = 0; i < N; ++i){
            if(used[i]!=1){
                length += arr[s][i];
                distributeTasks(i, num+1);
                length -= arr[s][i];
            }
        }
    }

    used[s] = 0 ;
    order[num] = -1;
}
```

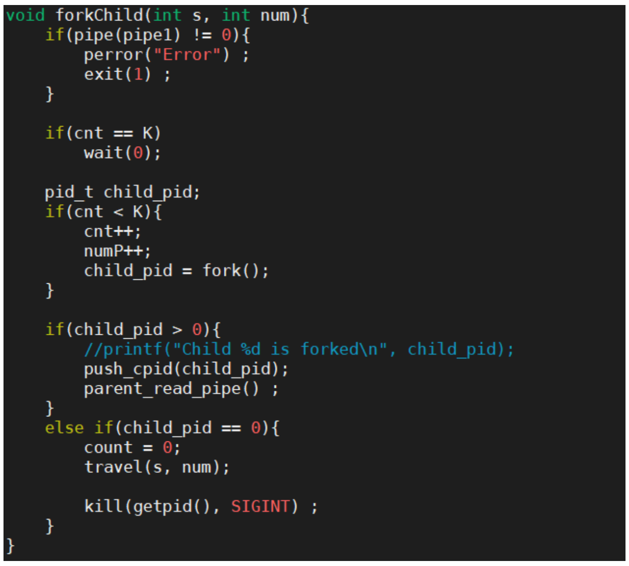
< *Figure 2. Task Distribution Function*>

2.1 How to distribute tasks to child process.

Each child process has an upper limit to perform only '12!' of task. In order to travel all possible cases, parent process need to spawn exactly N!/12! (N=num of cities) processes.

$$N!/12! = N * N-1 * \ldots * 13$$

I used a recursive function to make this amount processes. Recursion stops when the remaining number of prefix task is 12 factorial. And then it calls forkChild() function.

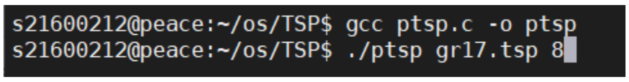


```
void forkChild(int s, int num){
    if(pipe(pipe1) != 0){
        perror("Error") ;
        exit(1) ;
    }

    if(cnt == K)
        wait(0);

    pid_t child_pid;
    if(cnt < K){
        cnt++;
        numP++;
        child_pid = fork();
    }

    if(child_pid > 0){
        //printf("Child %d is forked\n", child_pid);
        push_cpid(child_pid);
        parent_read_pipe() ;
    }
    else if(child_pid == 0){
        count = 0;
        travel(s, num);

        kill(getpid(), SIGINT) ;
    }
}
```

< *Figure 3. Fork child process and start to travel*>

Inside the forkChild() function, first, pipe is implemented for figure 1. Step 3. Transfer. Moreover, under the pipe codes, one conditional statement 'if(cnt == K)' is written.

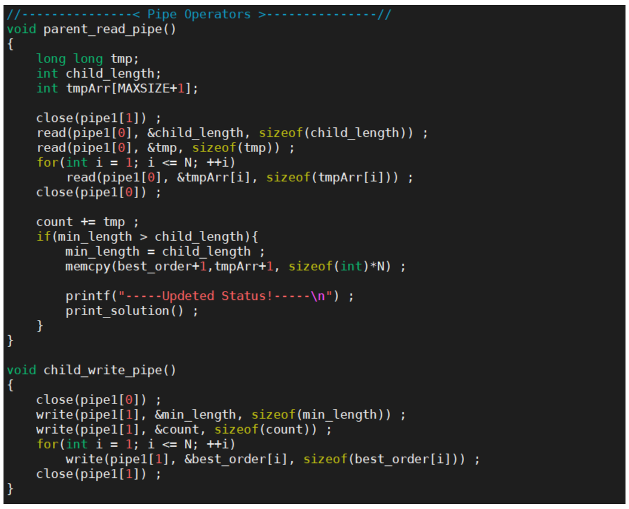
This conditional statement is implemented to limit the maximum number of processes that can work at the same time. When running this program, parent process receive K value with the graph file.



```
s21600212@peace:~/os/TSP$ gcc ptsp.c -o ptsp
s21600212@peace:~/os/TSP$ ./ptsp gr17.tsp 8
```

So if conditional statement satisfy, it means the number of running child processes is same with K. Then, parent should wait to spawn another child process. If the value of cnt is smaller than K, so when there is a seat, finally parent spawn child process using fork(). Spawned child now start to travel prefix task.

Within the travel() function, brute force algorithm is used to search all possible cases. After finish travel(), child process will kill itself using kill function. However, just before the child terminates itself, it has to transfer data to the parent process.



```
//---------------< Pipe Operators >---------------//
void parent_read_pipe()
{
    long long tmp;
    int child_length;
    int tmpArr[MAXSIZE+1];

    close(pipe1[1]) ;
    read(pipe1[0], &child_length, sizeof(child_length)) ;
    read(pipe1[0], &tmp, sizeof(tmp)) ;
    for(int i = 1; i <= N; ++i)
        read(pipe1[0], &tmpArr[i], sizeof(tmpArr[i])) ;
    close(pipe1[0]) ;

    count += tmp ;
    if(min_length > child_length){
        min_length = child_length ;
        memcpy(best_order+1,tmpArr+1, sizeof(int)*N) ;

        printf("-----Updeted Status!-----\n") ;
        print_solution() ;
    }
}

void child_write_pipe()
{
    close(pipe1[0]) ;
    write(pipe1[1], &min_length, sizeof(min_length)) ;
    write(pipe1[1], &count, sizeof(count)) ;
    for(int i = 1; i <= N; ++i)
        write(pipe1[1], &best_order[i], sizeof(best_order[i])) ;
    close(pipe1[1]) ;
}
```

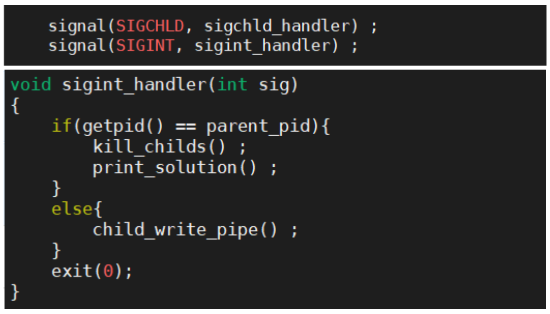
< Figure 4. Pipe Operators>

2.2 How to build pipe for communication between parent and child process.

As I mentioned before, pipe for communication is already called in forkChild function. So to transfer data through pipe, writing and reading functions are designed. Right before child process terminated, it transfer data using Figure 4. child_write_pipe() function. The data transferred includes three information. Shortest length of the routes, order of the this route, and the number of searched routes.

Parent process receives transferred data using Figure 4. parent_read_pipe() function. After complete receiving data through read pipe function, parent process compare passed data with old data. If passed solution is better, in other words, shorter route, replace the old information.

Let’s go back to when the child process terminated, and see how and where it call child_write_pipe() function.



```
    signal(SIGCHLD, sigchld_handler) ;
    signal(SIGINT, sigint_handler) ;
```

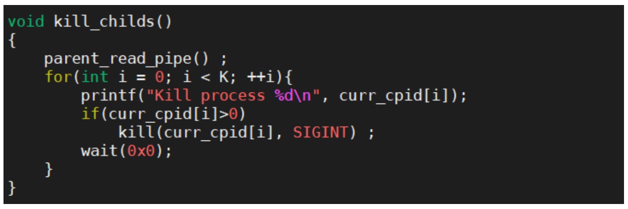
```
void sigint_handler(int sig)
{
    if(getpid() == parent_pid){
        kill_childs() ;
        print_solution() ;
    }
    else{
        child_write_pipe() ;
    }
    exit(0);
}
```

< Figure 5. SIGINT Handler>

Within the main function, signal function is implemented so to catch SIGINT signals. When child process try to kill itself using kill(getpid(), SIGINT) function in Figure 3, above Figrue 5. sigint_handler function is called. To verify this signal with parent process termination, variable ‘parent_pid’ contains the id of the parent process. Now the signal came from child process, so the child_write_pipe() function will be called.

2.3 How to handle unexpected suspension command.

When unexpected suspension command is given, in other words, when the parent process receives a kill signal, it calls kill_childs() function.



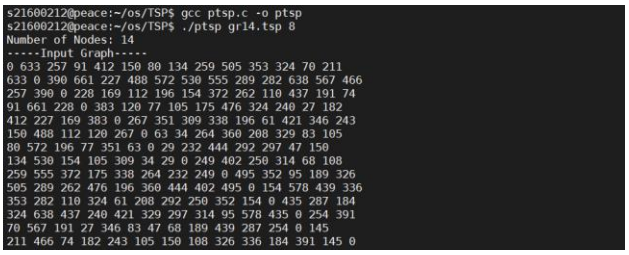
```
void kill_childs()
{
    parent_read_pipe() ;
    for(int i = 0; i < K; ++i){
        printf("Kill process %d\n", curr_cpid[i]);
        if(curr_cpid[i]>0)
            kill(curr_cpid[i], SIGINT) ;
        wait(0x0);
    }
}
```

< Figure 6. Kill all children >

Array curr_cpid[] contains current running process ids. The function kill_childs() kills all running child processes. At this time, SIGINT signal called to run child_write_pipe function(Figure 5). Therefore, child process can transfer best results that it have implemented so far.

3. Evaluation

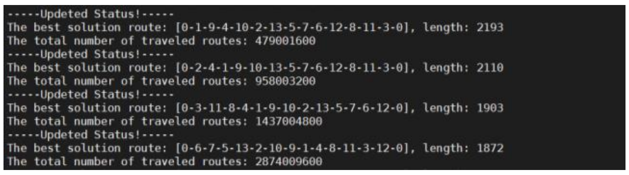
In order to check if MPTA fulfilled requirements or not, I used printf() to print out some values.



```
s21600212@peace:~/os/TSP$ gcc ptsp.c -o ptsp
s21600212@peace:~/os/TSP$ ./ptsp gr14.tsp 8
Number of Nodes: 14
-----Input Graph-----
0 633 257 91 412 150 80 134 259 505 353 324 70 211
633 0 390 661 227 488 572 530 555 289 282 638 567 466
257 390 0 228 169 112 196 154 372 262 110 437 191 74
91 661 228 0 383 120 77 105 175 476 324 240 27 182
412 227 169 383 0 267 351 309 338 196 61 421 346 243
150 488 112 120 267 0 63 34 264 360 208 329 83 105
80 572 196 77 351 63 0 29 232 444 292 297 47 150
134 530 154 105 309 34 29 0 249 402 250 314 68 108
259 555 372 175 338 264 232 249 0 495 352 95 189 326
505 289 262 476 196 360 444 402 495 0 154 578 439 336
353 282 110 324 61 208 292 250 352 154 0 435 287 184
324 638 437 240 421 329 297 314 95 578 435 0 254 391
70 567 191 27 346 83 47 68 189 439 287 254 0 145
211 466 74 182 243 105 150 108 326 336 184 391 145 0
```

< Figure 7. Graph Information >

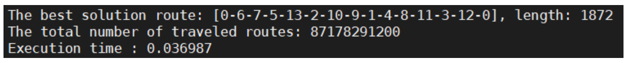
First, my program print out input graph information to check if successfully open the file or not.



```
-----Updeted Status!-----
The best solution route: [0-1-9-4-10-2-13-5-7-6-12-8-11-3-0], length: 2193
The total number of traveled routes: 479001600
-----Updeted Status!-----
The best solution route: [0-2-4-1-9-10-13-5-7-6-12-8-11-3-0], length: 2110
The total number of traveled routes: 958003200
-----Updeted Status!-----
The best solution route: [0-3-11-8-4-1-9-10-2-13-5-7-6-12-0], length: 1903
The total number of traveled routes: 1437004800
-----Updeted Status!-----
The best solution route: [0-6-7-5-13-2-10-9-1-4-8-11-3-12-0], length: 1872
The total number of traveled routes: 2874009600
```

< Figure 8. Updated data from child process>

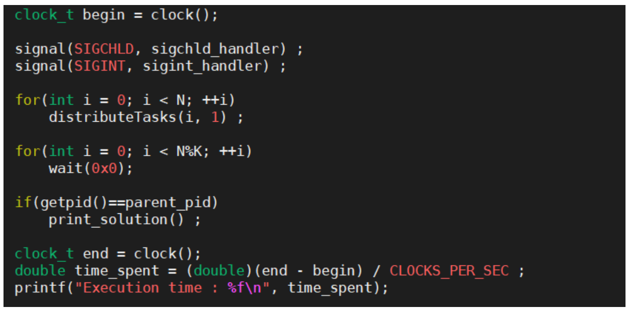
As I mentioned, parent process update best solution if passed data from child process is better than older one(you can check this out on Figure 4. if statement).



```
The best solution route: [0-6-7-5-13-2-10-9-1-4-8-11-3-12-0], length: 1872
The total number of traveled routes: 87178291200
Execution time : 0.036987
```

< Figure 8. Output of the MPTA on 14 nodes graph >

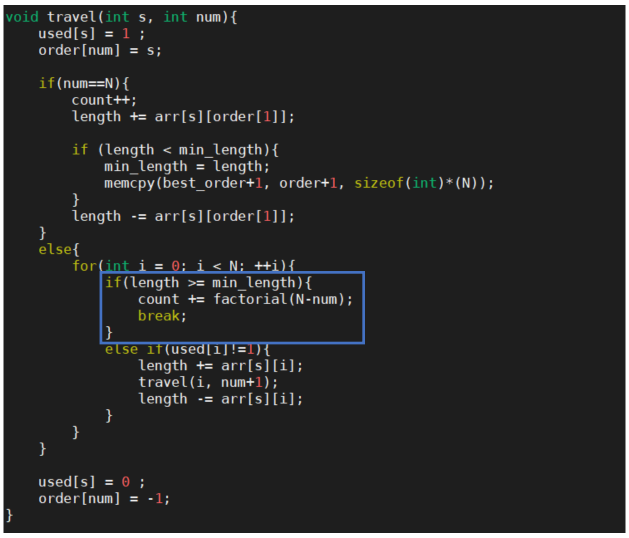
In order to evaluate MPTA, I made graph has 14 nodes. And MPTA prints out right solutions. Especially we know that the number of all possible routes of 14 nodes is ’14!’. As you can check MPTA has traveled exactly 14 factorial routes that is 87,178,291,200.



```
clock_t begin = clock();

signal(SIGCHLD, sigchld_handler) ;
signal(SIGINT, sigint_handler) ;

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

for(int i = 0; i < N%K; ++i)
    wait(0x0);

if(getpid()==parent_pid)
    print_solution() ;

clock_t end = clock();
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC ;
printf("Execution time : %f\n", time_spent);
```

< Figure 9. Execution time >

Furthermore, *Figure 9.* execution time is also checked for evaluation. In this time, for the 14 nodes graph, MPTA executed 0.036987 seconds.



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

    if(num==N){
        count++;
        length += arr[s][order[1]];

        if (length < min_length){
            min_length = length;
            memcpy(best_order+1, order+1, sizeof(int)*(N));
        }
        length -= arr[s][order[1]];
    }
    else{
        for(int i = 0; i < N; ++i){
            if(length >= min_length){
                count += factorial(N-num);
                break;
            }
            else if(used[i]!=1){
                length += arr[s][i];
                travel(i, num+1);
                length -= arr[s][i];
            }
        }
    }

    used[s] = 0 ;
    order[num] = -1;
}
```

< Figure 10. Bound method to shorten computing time >

3.1 Advanced Algorithm

Figure 10. Shows bound method. In order to improve computation performance, I use bound method. If computing length is lager than shortest length, algorithm stopped to check that routes so to shorten computing time.

4. Discussion

The most difficult part of this project was to deal with pipe information in parallel. I successfully completed multi-processing and also complete transferring data through the pipe. However, when connecting pipes to multi-process, the parent process waits to read data coming from child process. This phenomenon occurs the problem that parent process did not spawn multiple children at the same time. Even though I checked the speed of multi-process and I checked transferring through pipes, it is too bad that the two technologies could no be combined. In the future, I will study how to combine the two skills.

5. Conclusion

In this paper, I proposed one effective method to Multi-Process Traveling Algorithm(MPTA) to solve Traveling Salesman Problem. There were three key points. Distributing, piping and signal handler. First, through this project, I learned how to deal with multiple processes and how to distribute the work to them. And I also learned how to transfer via pipe between parent and child processes. To handle unexpected terminate signal, it was also possible to learn signal handling. In conclusion, because unite those technologies into one was really hard, I spent a lot of time and so I could develop myself a lot.