Parallel Programming HW2: Wandering Salesman Problem

9961244 EE14 吳德霖

Compile: gcc –pthread –lm -o WSP WSP.c
Run: ./WSP NUM_THREADS inputFile outputFile

◆ Implementation and Design

Acknowledgement:

The core algorithm for this homework is the so called dynamic programming commonly used for traveling salesman problem if the specified problem size is as small as 50 cities or below. Since the maximum size of the cities required this time is at 20, I think such algorithm would suit it well enough.

1. Dynamic Programming Algorithm

Here, we define a sub problem as:

For a subset S equals or belongs to {1, 2, 3, …, n} including the start point, here we just simply assume such start is from city 1. If city j belongs to S as well, then we can let C(S, j) be the length of the shortest path visiting each city exactly at once and ending at city j. And recursively, we can define C(S, j) = min{C(S-{j}, j) + distance from i to j} with I not equal to j. So the code of the algorithm is like this:

$C(\{1\}, 1) = 0$
for $s = 2$ to n:
  for all subsets $S \subseteq \{1, 2, \ldots, n\}$ of size s and containing 1:
    $C(S, 1) = \infty$
    for all $j \in S, j \neq 1$:
      $C(S, j) = \min\{C(S - \{j\}, i) + d_{ij} : i \in S, i \neq j\}$
return $\min_j C(\{1, \ldots, n\}, j) + d_{j1}$

As a result, to implement the code, we need two formats with n X 2^n-1 size to keep tracking the best route for the current state. The rows represent the current city, while the columns indicate whether it's in the set or not. Sets here are transformed into its binary form, such as 00110 indicates {1, 2}, which is a subset or {1, 2, 3, 4, 5}. One format stores the optimal length with another store the back traceable route. Since the original version requires the salesman to return to the start point, here I altered the algorithm to make a fake route to force the algorithm select such return path by transforming all the lengths of paths to 0 if they can ever visit my specified start point. That makes exactly the same route start from one city to the terminated destination. Therefore, totally we need n situations each start at 1, 2, 3 … to n. The time complexity of such algorithm is then evaluated as $O(n^2 \cdot 2^{n-1})$. It's a significant reduction from the brutal force method whose time complexity is expressed as O(n!).

2. Parallelize and preventing synchronization

To parallelize the problem, and considering that the program should still be functioning even if giving relatively less threads, I divide the whole task in parallel by the each possible starting point, making totally n cases to be parallelized. If the thread given is more than the size of the cities, considering the upper limit source of the processors, I terminated the excessive threads since the original version of parallelizing is functioning with sufficient efficacy. If the threads given is quite less to the size of the cities, each group with exactly the same numbers of start points would be conducted, while the left wait till they're terminated by the threads and such process keep recursively functioning till the last start point. Ever since the format is the essential factor and could definitely not suffer from synchronization, so I simply defined 20 such formats in order to meet any cases below 20. But for other not so important parameters such as time measuring variables, although the vitality is lower but still can't undergo synchronization, I simply applied the locking and condition variable to control the recording variable for the time, only if the whole series of one group is finished, the condition would signal a trigger condition to the header node for ending time recording, otherwise, the header node would be kept waiting till it senses the signal condition.

3. Reduce execution time and increase scalability

In fact, the way to parallelize the implementation itself implies the reduction of execution time, since the task is then distributed to a Size / NUM_THREADS scale down. If ever the given thread can divide the problem size or even equal to the problem size, it would reach the local maximum reduction of time since the thread is evenly assigned for sub tasks, taking full advantage of such algorithm. The result shows pretty good scalability, the figures would be shown later on in corresponding section.

◆ Experiment & Analysis:

1. Scalability

The format below is the required scalability results of conducted experiments.

Size	5	5	5	10	10	10	20	20	20
Test	Lv1	Lv2	Lv3	Lv1	Lv2	Lv3	Lv1	Lv2	Lv3
Time(s)	0.000	0.000	0.000	0.002	0.02	0.004	11.25	11.22	11.56
#thread	5	5	5	10	10	10	16	16	16

As one can see, this implementation solves all provided test cases, and is capable of dealing with all cases as long as its size is below 20.

2. Scalability Plot

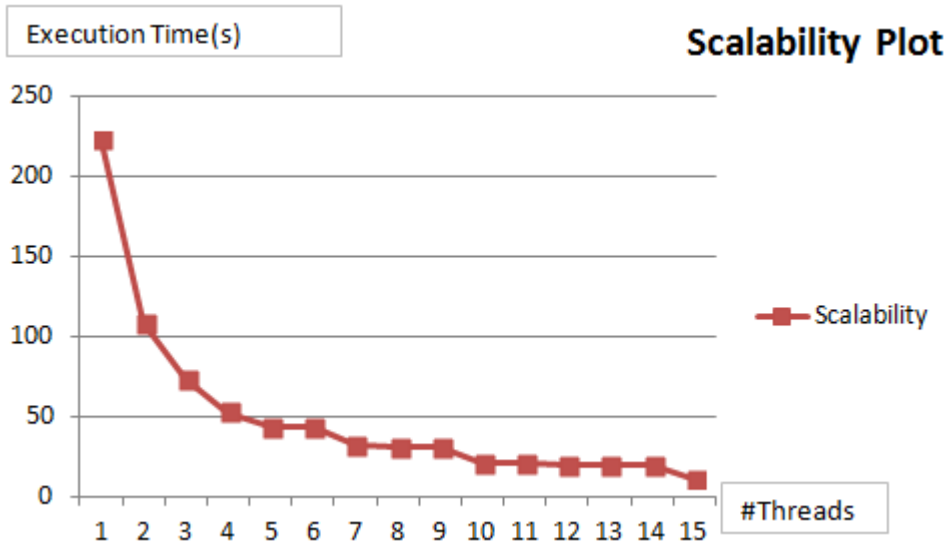
The result below is conducted by fixing the city size as the maximum indicated this time, that is, 20 cities in total. I initiate the clock() function in my indicated header thread which is exactly the thread 1, and terminated the recording if the whole series of one group is done. For example, if we are given 4 threads, then thread 0 would terminate its measurement due to a signal from all other threads, by summing up such measured time recursively till the end of the execution, at last store the information to the output file by appending up the results. Since the algorithm I implemented is rather map-independent, the time consumed in each type of map was merely the same.

#Threads	1	2	3	4	5	6	7
Time(s)	222.8	108.8	73.9	52.9	43.9	43.1	32.3

#Threads	8	9	10	11	12	13	16	20
Time(s)	30.5	31.5	21.5	20.8	19.7	20.2	19.2	11.4

(Input: 20c_lv2.in)

The plot is as shown below:



Observing the plot above, one can say that it shows relatively good scalability, but one can also notice that after some steep declining, there would exhibit a flat band which doesn't alter too much with the increment of the processors. The cause of this phenomenon is quite simple, the way I parallelize the implementation doesn't involve any load balancing. That is, if the given thread cannot divide the city size, say 11 threads for size of 20 cities, and then the first execution contains 11 threads while the second execution contains 9 threads with 2 extra threads not functioning. The total proper used time is still the sum of the longest thread used time in each execution, which makes slight difference

form the case given 10 threads. Such issue is considered can-be-improved, but I still adopt this method since this kind of threading is simplest for dynamic programming algorithm. To sum up, if the given thread can divide the size of cities, it would experience perfect scalability as long as the number of threads increases, in contrast, the indivisibility could somehow cause the scalability plot to undergo a not so significant change due to some kind of wasting of those remainder threads.

3. Locking Operation Analysis

The only locking conditions I applied is for the time measurement, each thread except for the header thread, thread0, when they're conducting the WSP functions, they will lock a variable called Timer and execute Timer++, after the execution they then signal the condition. When the Timer happens to reach a certain value, the waiting function in the header node would sense the signal condition and thus terminate the waiting process to proceed to record the measurement of the execution time. Such locks and unlocks are called once throughout nodes but the header node, since the header node should as well lock the initiating of the time recording, so totally one series of a group will launch (NUM_THREADS-1)X1 + 2 locking and conditionings. And the waiting time of locking process from the header node is exactly the execution time for this series of one group.

The resulting lock counts recorded as follows, and since the waiting time in total is merely equal to the total execution time, I omitted it from the format.

#Threads	1	2	3	4	5
#Locks	10	8	7	7	6

(Input: 5c_lv2.in Waiting time of each lock = 0.000s)

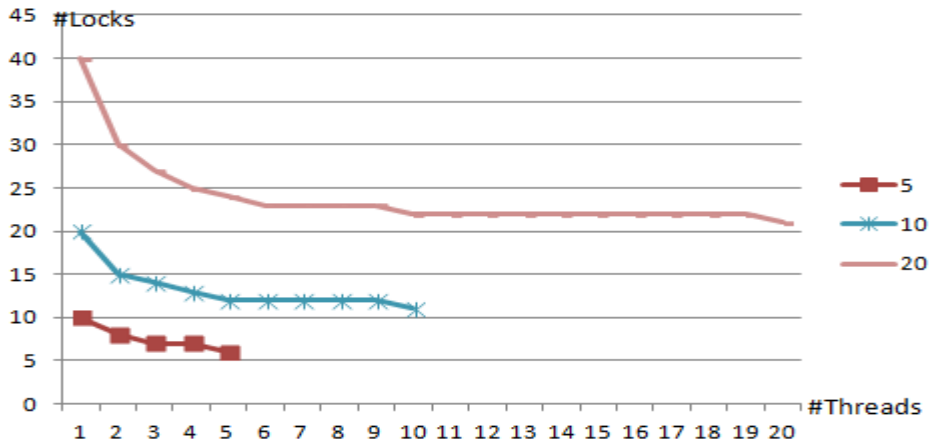
#Threads	1	2	3	4	5	6	7	8	9	10
#Locks	20	15	14	13	12	12	12	12	12	11

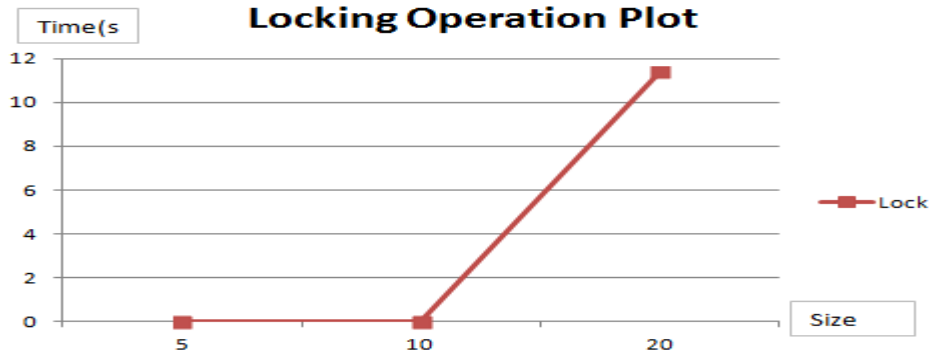
(Input: 10c_v2.in Waiting time of each lock = 0.004s)

#Threads	1	2	3	4	5	6	7	8	9	10
#Locks	40	30	27	25	24	23	23	23	23	22
#Threads	11	12	13	14	15	16	17	18	19	20
#Locks	22	22	22	22	22	22	22	22	22	21

(Input: 20c_v2.in Waiting time of each lock = 11.4s)

The corresponding graph is plotted as below.





One can see that the plot of locks also showing great scalability. The reason is again very simple, that is, the locks have a strong correlation to how many threads are functioning during whole execution. The more the threads are given, the less total execution series needed thereby saving the use of locks.

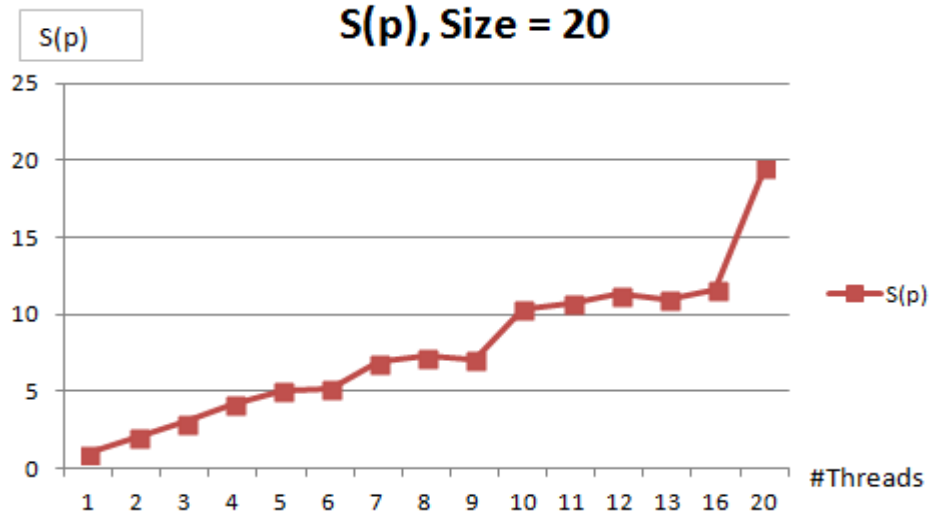
4. Additional Analysis

The performance efficiency is analyzed as below

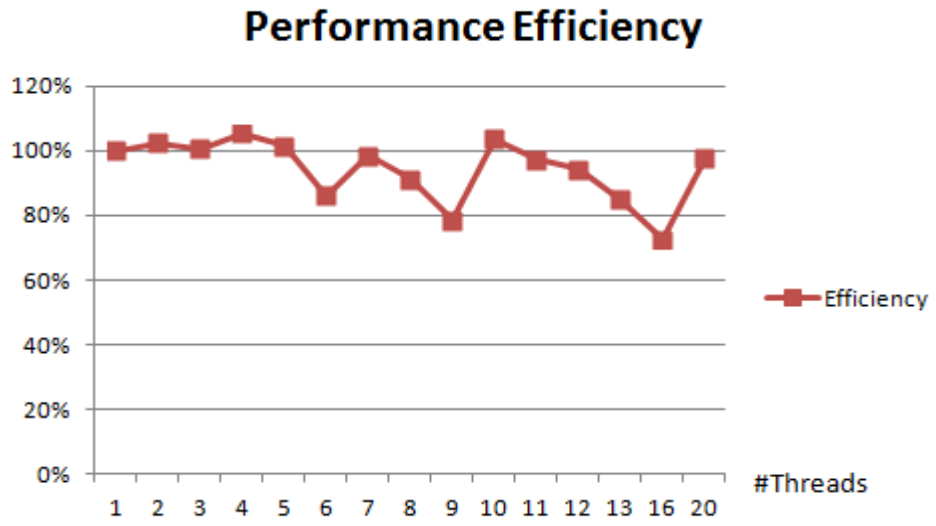
The best enhancement occurs when number of threads is equal to 20, which is 19.5267 / 20 = 97.6%.

The experiment is conducted under the same situation as that for analyzing the scalability, since the size of 5 and 10 provided is not as applicable and obvious as the size of 20, I simply think this would thus represent my performance efficiency. One can see that the S(p) increases merely linearly as the threads increases, which in turns shows again the well done scalable implementation. When the problem size is divisible by the number of the given threads, the system's performance efficiency would reach its peak at around 100%. The plotted result is shown as below, with the same test case of size = 20 fixed.

The plot of S(p):



The plot of the performance efficiency:



5. Source codes

Please refer to the attached files and the instructions of how to compile the source code as well as execute the executable file at the very beginning of this report.