Assignment C

List Ranking

Wang Yanhao - 3 July 2016

Introduction

Problem requirements:

Given a linked list L with N nodes, represented by an array S such that S(i) is the successor of i. The last node is represented by S(i) = 0.

Assignment C requires us to design and implement a parallel program with O(log(N)) time complexity using pointer jumping algorithm in both PThreads and OpenMP that calculates an array R such that each value R(i) is the distance from node i to the end (last node) of L.

Algorithm:

As the question requires, we use pointer jumping technique to solve this question. The algorithm works as following:

1. traverse through the tree and assign the distance to the root as 1 for all non-root node and 0 for the root.

2. Update the successor of each node by that successor's successor and update the distance to the root as the current distance to the node together with the successor's distance to the root until the root is found.

Algorithm Proof

The algorithm starts by checking if the current node is the root itself. trivial that the root itself will always pointing to itself and the distance to itself is 0. Hence the algorithm works for n=1. Assume algorithm works for n=k where k>=1, consider n=k+1.

After the first iteration, the node with index k+1 will update it's successor to node k's successor and having an updated distance of 1. Hence, the n=k+1 case has been converted to the n=k case with 1 extra value for distance. Therefore if n=k is true, n=k+1 must be true as well.

Hence, the algorithm always terminates and find the distance to root for every node with n >= 1.

Analysis of time complexity

Since each node's distance to the node will be halved at each iteration, it is obvious that the time complexity T(n) = T(log n)

Implementation

The sequential implementation is straightforward from the algorithm described in Section Introduction, however, there are one thing to take note of for Step 2:

Instead of implementing a while loop which terminates after meeting the root, we use a for loop through 1 to log n to replace the while loop because the for loop can be parallelised easily.

The OpenMP implementation comes directly from the sequential implementation by parallelising the for loops.

The pthread implementation use similar strategy as the OpenMP implementation but requires more details. Before running the for loop, we decide the portion of indexes each available thread needs to work on by averaging (number of work / number of threads). And each thread will perform calculation in their assigned index range, which is not overlapping with other threads.

Test Result

Environment:
The testing environment is the remote accessed computer via TUD277869.ws.tudelft.net

Result:
The raw test result generated by the code is shown below:

```
OpenMP:
|NSize|Iterations| Seq  | Th01 | Th02 | Th04 | Th08 | Par16|
| 4096 | 1000 | 0.135664 |  0.125213 |  0.092469 |  0.062730 |  0.050257 |  0.052396 |
| 8192 | 1000 | 0.285319 |  0.275876 |  0.175725 |  0.109941 |  0.079889 |  0.075198 |
| 16384 | 1000 | 0.600512 |  0.586636 |  0.340783 |  0.225242 |  0.147484 |  0.127989 |
| 32768 | 1000 | 1.280885 |  1.243800 |  0.719585 |  0.440673 |  0.276263 |  0.233994 |
| 65536 | 1000 | 2.703344 |  2.628511 |  1.475126 |  0.878000 |  0.539248 |  0.434579 |
| 131072 | 1000 | 5.711928 |  5.548582 |  3.066758 |  1.804299 |  1.090752 |  0.851764 |
| 262144 | 1000 | 12.001020 |  11.665019 |  6.385580 |  3.749155 |  2.221893 |  1.712691 |
Pthread:
|NSize|Iterations| Seq  | Th01 | Th02 | Th04 | Th08 | Par16|
| 4096 | 1000 | 0.130434 |  0.283218 |  0.300830 |  0.397011 |  1.297625 |  2.543787 |
| 8192 | 1000 | 0.282081 |  0.468126 |  0.418466 |  0.470996 |  1.363660 |  2.853766 |
| 16384 | 1000 | 0.600879 |  0.898196 |  0.588969 |  0.560992 |  1.567462 |  2.872546 |
| 32768 | 1000 | 1.281702 |  1.536629 |  0.922891 |  0.819090 |  1.759695 |  3.495048 |
| 65536 | 1000 | 2.705992 |  2.843296 |  1.757211 |  1.201443 |  2.052473 |  3.901216 |
| 131072 | 1000 | 5.719306 |  5.668574 |  3.536261 |  2.004347 |  2.428301 |  4.553647 |
| 262144 | 1000 | 11.999257 |  11.589019 |  7.042255 |  3.789041 |  3.242584 |  5.262051 |
```

From Assignment A, we have the following observations:

1. The time taken to execute the program increases with NSize.
2. OpenMP implementation perform better(faster) than pthread implementation.
3. For fixed input size, the time taken to execute the program will first decrease as the number of threads grow, then increase after the number of threads surpass a certain point.
4. For fixed number of threads, the Speedup increases as the input size grows.

5. For fixed input size, the speedup will increase as the number of threads grow to certain threshold, then decrease afterwards.

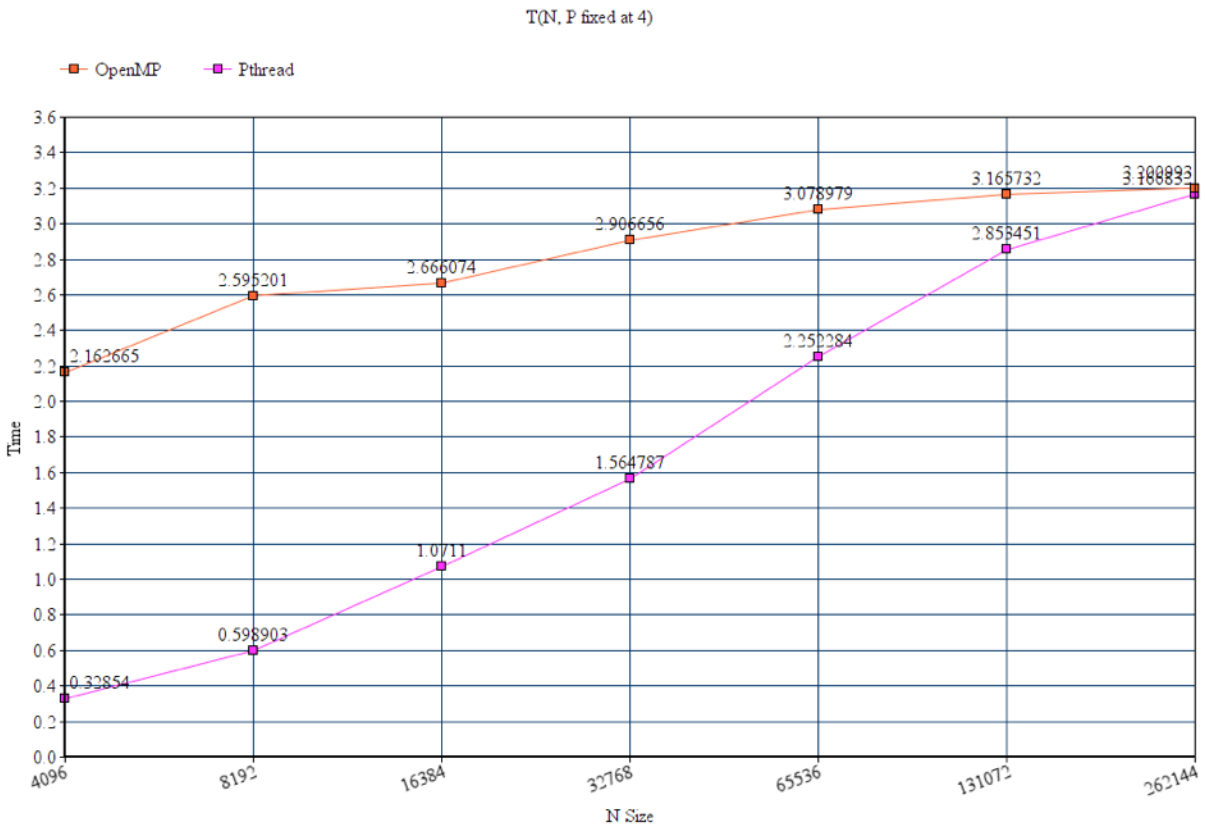
And we have confirmed the reoccurrence of all the observations from Assignment B
Now, we compare the result from the table above and confirm the validity of the 5 observations.

One can easily check that Observation 1 and 2 hold for all sets.

One can also easily check that Observation 3 holds for Pthread implementation while it only hold for OpenMP implementation when N size is small. This has been explained in Assignment B as current version of Pthread implementation is not optimal, hence OpenMP performs better and the number of threads may not be large enough for this observation to take place for large value of N size.

Since speedup = time taken for sequential algorithm / time taken for parallelised algorithm, we can confirm Observation 5 by taking each row of the table, and check if the time taken decreases when number of threads increases until certain point, and increases afterwards.
And indeed this is the case, hence Observation 5 holds for all data from Pthread implementation and for small N-size (N=4096) OpenMP implementation as well.

Hence, we only need to validate Observation 4 through Speedup (N, P fixed at 4) graph:



Conclusion

In this exercise we have learnt to find the distance to the root of an array using the pointer jumping method and two ways to parallelise the sequential algorithm.

Similar to previous two assignments, we have studied how to create multi-threaded programs in C programming language using OpenMP and pthreads libraries. It is much easier to use OpenMP to parallelise C programs whereas pthread grants us more control over the how each thread should perform.

From the test result, we have re-confirmed that sequential algorithms generally perform better if the input size is small and not many calculation is involved whereas the benefit gained from concurrent execution will be surpassed by the overhead of creating and destroying threads when the input size is small.

However, if there are a lot of calculation needed for each thread to perform, parallelising the execution will grant us great performance benefits.

In practice, it might be useful to us to perform some experiment in order to estimate the communication overhead and determine the optimal number of threads to use for parallelising tasks.