Assignment B

Simple Merge

Wang Yanhao - 3 July 2016

Introduction

Problem requirements:

Let X = (x_1,...x_n) be a sequence of elements drawn from a linear ordered set S. The rank on an element $x \in S$ in X, denoted by rank(x : X) is the number of elements of X smaller than or equal to x. If Y = (y_1,...,y_M) is another sequence with elements from S (i.e., Y not equal to X), then rank(Y : X) = (r_1,...,r_M), where r_i = rank(y_i : X) is the rank of Y in X.

Merging two arrays A and B is equal to determining rank(A : AB) and rank(B : AB), where AB is the concatenation of A and B.

Assignment B requires us to design and implement an efficient parallel program with time complexity O(log(N + M)) in both PThreads and OpenMP that merges two sorted sequences using the method above.

Algorithm:

As suggested in the question description, merging two arrays A and B is equal to determining rank(A : AB) and rank(B : AB), where AB is the concatenation of A and B. And using the fact that rank(A:AB) = rank(A:A) + rank(A:B), (which will be proved later) we have the following algorithm (assuming C is the resulting array):

```
1. for each $a_i$ in A, C[rank($a_i$:B) + rank($a_i$:A)] = $a_i$
2. for each $b_i$ in B, C[rank($b_i$:A) + rank($b_i$:B)] = $b_i$
```

Algorithm Proof

Assuming all elements in C=AB is unique. Hence, rank of every elements of C in C must be unique as well. More specifically, rank(c_i:C) = i.

Since rank(c_i:A), rank(c_i:B) and rank(c_i:C) represents the number of elements smaller than c_i in A,B,C respectively, and given the definition that C=AB, it is obvious that rank(c_i:C) = rank(c_i:A) + rank(c_i:B).

Finally, by considering all elements of A in C, we have proven rank(A:AB) = rank(A:A) + rank(A:B). Therefore, the correctness of the algorithm is guaranteed.

Analysis of time complexity

Since we can use binary search method to find the rank, calculating rank(a_i:B) has a time complexity of O(log m) and calculating rank(b_i:A) has a time complexity of O(log n). By calculating the values in parallel, we can complete step 1 in max(O(log m), O(log n)).

Step 2 and step 3 are merely assigning values, hence can be done in parallel with O(1) time complexity.

Therefore: T(n) = max(O(log m), O(log n))

Implementation

The sequential implementation is straightforward from the algorithm described in Section Introduction. To be more specific, rank(a_i:A) and rank(b_i:B) can be obtained by taking the index i while rank(a_i:B) and rank(b_i:A) can be obtained by using binary search method.

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

Note:
The graphs generated in this section are not to scale due to the lack of time to write a proper code to generate graph, all credits of graphs goes to onlinecharttool.com

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:

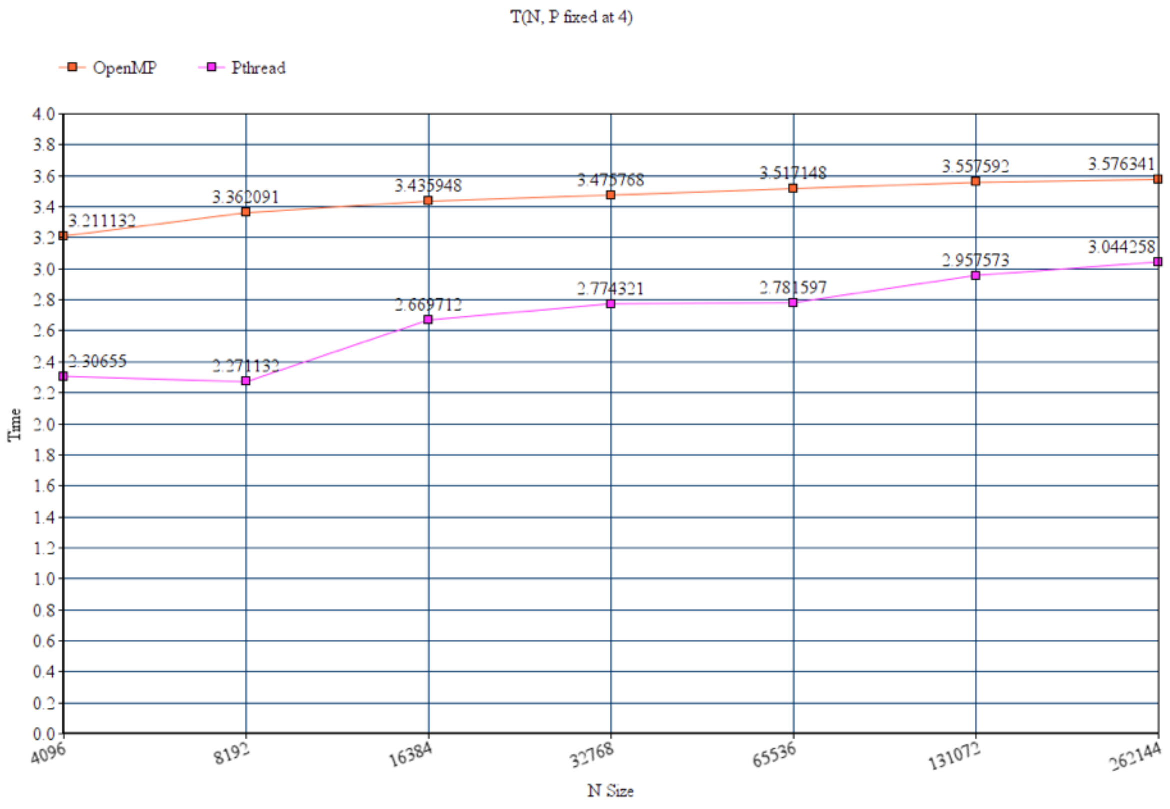
```
OpenMP:
|NSize|Iterations| Seq | Th01 | Th02 | Th04 | Th08 | Par16|
| 4096 | 1000 | 0.439238 |  0.430860 |  0.230817 |  0.136786 |  0.073611 |  0.049691 |
| 8192 | 1000 | 0.894656 |  0.906563 |  0.491129 |  0.266101 |  0.156480 |  0.098226 |
| 16384 | 1000 | 1.897118 |  1.922964 |  1.009634 |  0.552138 |  0.301987 |  0.204060 |
| 32768 | 1000 | 4.022858 |  4.073887 |  2.122028 |  1.157401 |  0.618444 |  0.424830 |
| 65536 | 1000 | 8.522268 |  8.621085 |  4.462231 |  2.423062 |  1.293848 |  0.883450 |
| 131072 | 1000 | 18.532097 |  18.700790 |  9.572899 |  5.209168 |  2.748964 |  1.891329 |
| 262144 | 1000 | 39.447476 |  39.812067 |  20.498193 |  11.030120 |  5.775731 |  4.030007 |
Pthread:
|NSize|Iterations| Seq | Th01 | Th02 | Th04 | Th08 | Par16|
| 4096 | 1000 | 0.426442 |  0.489590 |  0.309940 |  0.184883 |  0.224182 |  0.466968 |
| 8192 | 1000 | 0.896836 |  1.017169 |  0.656778 |  0.394885 |  0.294926 |  0.511153 |
| 16384 | 1000 | 1.901057 |  2.043196 |  1.287448 |  0.712083 |  0.442444 |  0.550441 |
| 32768 | 1000 | 4.031675 |  4.254991 |  2.579361 |  1.453211 |  0.769079 |  0.731828 |
| 65536 | 1000 | 8.531653 |  8.900821 |  5.269332 |  3.067177 |  1.497430 |  1.152908 |
| 131072 | 1000 | 18.546226 |  19.237645 |  11.025646 |  6.270758 |  3.388374 |  2.165347 |
| 262144 | 1000 | 39.459099 |  40.710103 |  23.067088 |  12.961808 |  6.857281 |  4.322758 |
```

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.

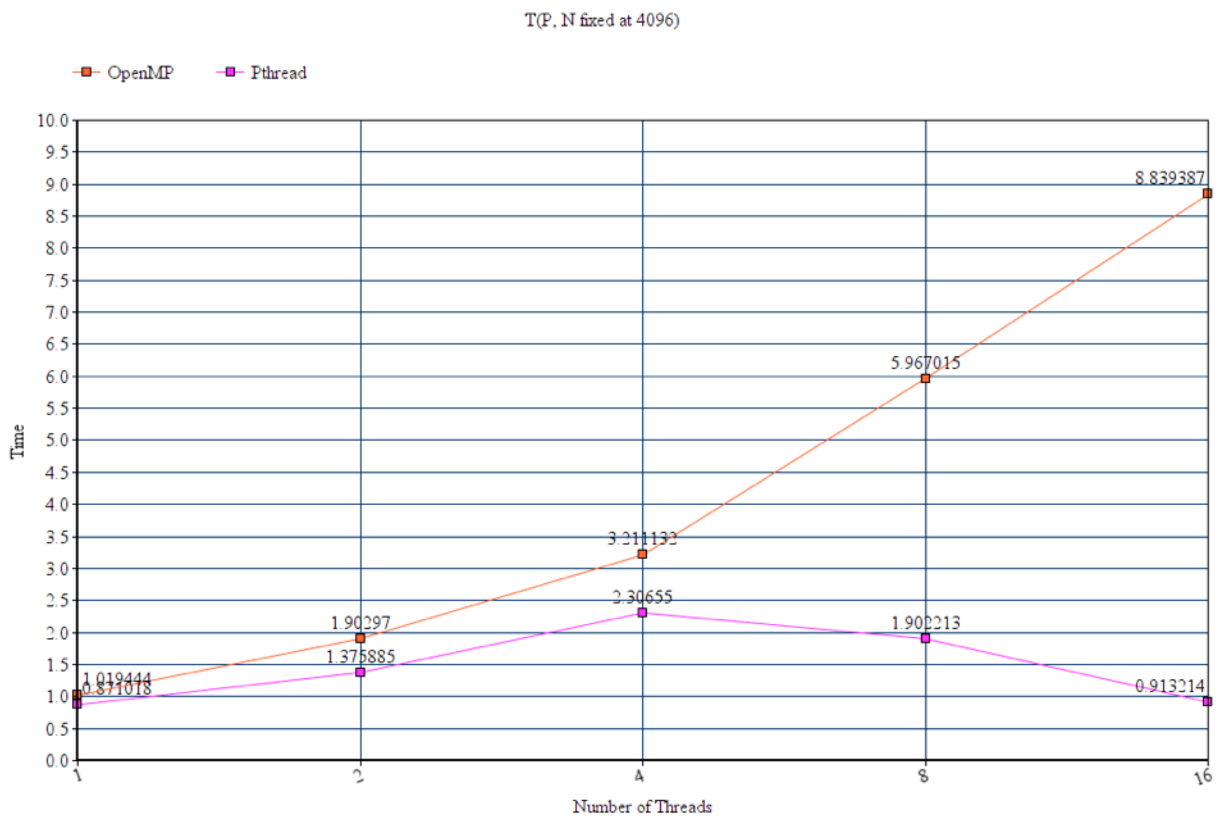
From the result generated above, one can easily check that Observation 1 and Observation 2 holds for all sets of results.
Observation 3 holds only for sets where NSize is small (4096, 8192 and 16384 for Pthread implementation) because the number of thread is limited to 16, hence we can't conclude this observation is not valid based on data for NSize larger than 32768.
Since Observation 1 to 3 have been verified, I believe there is no point to plot T(N,P fixed) and T(P, N fixed) graph. Hence we will focus on verifying Observation 4 and 5 through the plots.

Speedup(N, P fixed at 4)



From the Speedup(N, P fixed at 4) graph, we can see that when number of threads is fixed, the Speedup increases as the input size grows for both OpenMP and Pthread implementation (with a small fluctuation at N=8192 for Pthread implementation which I believe is a reasonable fluctuation and doesn't affect the conclusion). Hence, observation 3 is re-confirmed.

Speedup(P, N fixed at 4096)



From the Speedup(P, N fixed at 4096) graph, we observe that for Pthread implementation, speedup increases until thread number equals to 4, and decrease afterwards. This is in accordance with the Observation 4 stated above.
However, for OpenMP implementation, the speedup seems keep increasing. This might be caused by the fact that the Pthread implementation is less optimal than the OpenMP implementation, hence we need more threads (much larger than 16) to observe the pattern for OpenMP implementation.

By plotting the two graphs related to speedup, we have confirmed that except for OpenMP implementation, other observations made in Assignment A also hold for data gathered in Assignment B.

Conclusion

In this exercise, we have learnt to solve the problem of merging of two sorted array into another sorted array with the help of concept rank.

Moreover, we have studied how to use OpenMP and pthreads to implement the algorithm. Once again, although it is easier to implement via OpenMP, pthread grants us more control over the how each thread should perform and may leads to better performance under certain conditions.

From the test result, we have re-confirmed the findings we gathered from Assignment A and we can see 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.