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:

The optimal sequential algorithm for merging two sorted array A and B into a result array C will start with array A, compare the first element in array A with the first element in array B. If the element in A is smaller, we push the element in array A to array C, and comparing the second element in A with first element in B, vice versa. And this procedure continues until all elements in A and B are copied to C.

The above algorithm is implemented in code as function: “opt_seq_merge”

The parallel program make use of the concept rank runs differently. Assuming array A has size n, array B has size m, the partitioning algorithm runs as following:

1. we choose approximately n/log n elements from array A that partitions A into blocks of almost equal lengths.
2. for each element that partitioned array A into blocks, calculate the rank of it in B using binary search method
3. based on result from step 2, partition array B into log n blocks such that every element in block i of A and block i of B is larger than every element in block (i-1) of A or block (i-1) of B.
4. use opt_seq_merge to merge all elements in block Ai and block Bi pair
5. obtain the final result by concatenate every pair from step 4

Algorithm Proof

After step 1-3, array A and array B are both divided into log n subsequences. Since we calculate the rank of the last element of any subsequence of A in B, and partitions B based on the rank, together with the fact that A is a sorted array, it is obvious that each element in the subsequences A_i is larger than each element of A_{i-1} or B_{i-1}. Moreover, since B is also sorted and the smallest element in B_i is larger than the largest element in A_{i-1} (because its index in B is larger than the rank of largest element in A_{i-1} in B), each element in B_i must also be larger than each element of A_{i-1} or B_{i-1}. Hence, the correctness of this step is ensured.

Step 4 make use of opt_seq_merge which is the trivial way of merging two sorted sequence. Step 5 simply combines all the sorted subsequence together.

Therefore, the correctness of the algorithm is guaranteed.

Analysis of time complexity

Step 1 has a trivial time complexity of O(1).

Step 2 can be done in parallel, hence the time complexity depends on calculation of rank which is O(log m) if using sequential binary search.

Step 3 can be done in parallel with T(n) = O(1).

Step 4 and 5 can be done together by merging each pair of subsequences into the different positions of the same result array. Since each A_i has log n elements and each size of is O(log m). This sequential merge can be done in O(log n + log m).

Therefore: T(n) = O(log n + log m)

Implementation

The sequential implementation is straightforward from the algorithm described in Section Introduction.

The OpenMP implementation comes directly from the partitioning algorithm described above with parallelised for loop for Step 2-3 and Step 4-5.

Take note that we can combine Step 2 and 3 together for slightly better performance.

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:

```
|NSize|Iterations|Seq|OpenMP|Th01|Th02|Th04|Th08|Par16|
| 4096 | 1000 | 0.010785 |  0.023885 |  0.074231 |  0.061876 |  0.078962 |  0.217911 |  0.391247 |
| 8192 | 1000 | 0.014708 |  0.040972 |  0.080209 |  0.082279 |  0.098147 |  0.197954 |  0.398723 |
| 16384 | 1000 | 0.029261 |  0.080148 |  0.118478 |  0.091330 |  0.117385 |  0.224510 |  0.408379 |
| 32768 | 1000 | 0.058481 |  0.157909 |  0.200844 |  0.181502 |  0.165598 |  0.277305 |  0.444530 |
| 65536 | 1000 | 0.116672 |  0.314461 |  0.390422 |  0.254131 |  0.290611 |  0.362425 |  0.517671 |
| 131072 | 1000 | 0.230133 |  0.626454 |  0.818311 |  0.597606 |  0.551975 |  0.600460 |  0.952128 |
| 262144 | 1000 | 0.459849_|  1.257217 |  1.842567 |  1.082196 |  0.979705 |  0.932255 |  1.138490 |
```

From Assignment A, we have the following observations:

1. The time taken to execute the program increases with NSize.
2. 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.
3. For fixed number of threads, the Speedup increases as the input size grows.
4. For fixed input size, the speedup will increase as the number of threads grow to certain threshold, then decrease afterwards.
5. Time taken for OpenMP and sequential algorithm is similar.

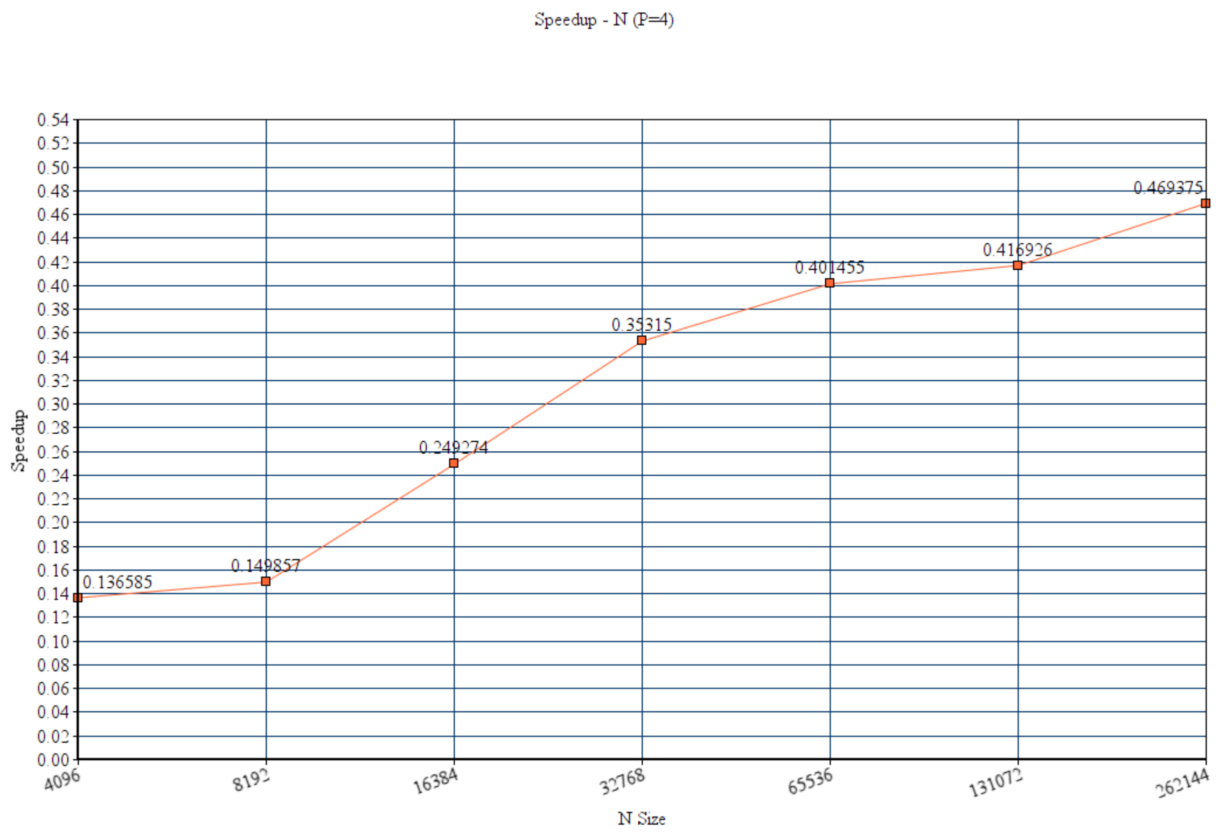
From the result generated above, one can easily check that Observation 1 holds for all sets of results except when NSize increases from 4096 to 8192 when Thread number is 8.
I personally believe this is an outlier which is caused by inherit fluctuation when running program on machines.

Observation 2 holds for all sets except when NSize is 8192 where values for Thread number 1 and 2 are very close hence I think this is also reasonable.

Observation 5 does not hold any more because the implementation for OpenMP does not directly come from parallelising the for loop in sequential algorithm.

Since Observation 1 and 2 can be easily 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 3 and 4 through the plots.

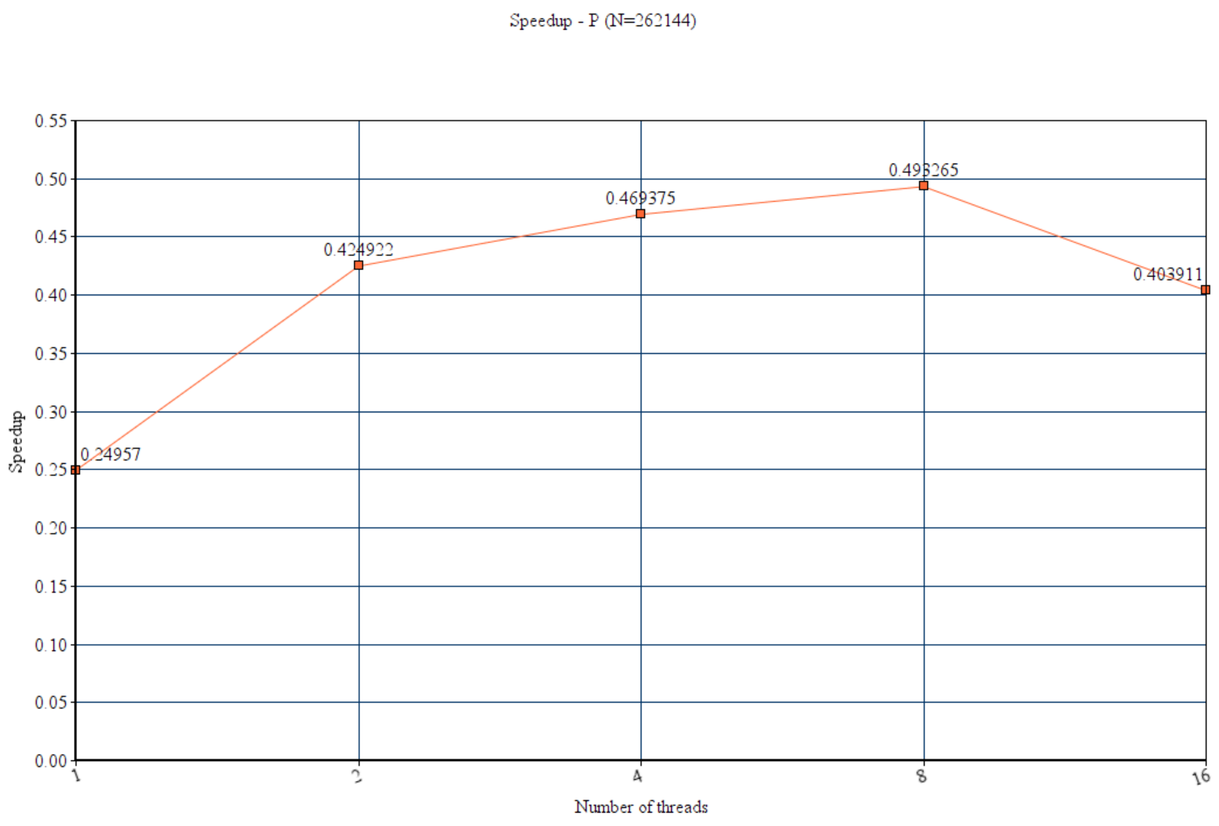
Speedup(N, P fixed at 4)



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. Hence, observation 3 is re-confirmed.

From the Speedup(P, N fixed at 262144) graph, we observe that speedup increases until thread number equals to 8, and decrease afterwards. This is in accordance with the Observation 4 stated above.

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, using partitioning method. 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.