Assignment A

Prefix / Suffix minima

Wang Yanhao - 3 July 2016

Introduction

Problem requirements:

Given an array $A = (a_1, \ldots, a_N)$ with elements drawn from a linear ordered set. The suffix minima problem is to determine $\min\{a_i, a_{i+1}, \ldots a_N\}$ for each i. The prefix minima are $\min\{a_1, a_2, \ldots a_i\}$ for each i.

Assignment A requires us to design and implement an efficient parallel program with time complexity O(log(N)) in both PThreads and OpenMP that computes the prefix and suffix minima of any given array. Moreover, an efficient sequential algorithm should be provided as well for later comparison of time taken between sequential implementation and parallel implementations, as described in Section Test Result.

Algorithm:

The intuitive sequential algorithm for prefix minima will have one for loop with i as the counter and check from the first element until the last element in the array. Another variable is needed to keep track of the current minima and update the prefix minima for each position. The sequential algorithm for suffix minima can be done similarly except this time the for loop counter goes from the index of the last element to the index of the first element.

However, since the value of the current minima depends on the position of the for loop counter, this sequential algorithm can not be easily paralleled. Hence, we need to use Balanced Tree method to implement a recursive algorithm where:

1. we generate an array consists of the minima of adjacent elements
2. we use a recursive call to compute the minima for the obtained array
3. we update the actual minima array from result generated above

Unlike the intuitive sequential algorithm, the first and third step for this Balanced Tree method can be parallelised easily.

Algorithm Proof

We only need to show the correctness for prefix minima because the proof of correctness for suffix minima is similar.

We prove the correctness of the algorithm by mathematical induction on k, where the size of the input is $n = 2^k$.

For prefix minima, when $k = 0$, the case is trivial because the result will be the only element itself.

Now assume the algorithm is true for some $k>0$, we look at $n=2^{k+1}$.

By induction, the result array after recursive step 2 holds the prefix minima of the sequence $(z_1, \ldots, z_{N/2})$ where $z_i = \min(a_{2i-1}, a_{2i})$ for $1 <= i <= n/2$. Therefore, it is easy to see that when i is even, $\text{prefixmin}(a_i) = z_{i/2}$. When i is odd, assume $i = 2j + 1$, $\text{prefixmin}(a_i) = \text{prefixmin}(a_{2j+1}) = \min(\text{prefixmin}(a_{2j}), a_{2j+1}) = \min(z_{(i-1)/2}, a_i)$

Therefore, if the algorithm is true for some $k>0$, it also works for the case of $k+1$.

Hence, we have proven the algorithm is true for all input size.

Analysis of time complexity

For step 1 and step 3, by using a parallelised implementation, the time complexity is $O(1)$ while the time complexity for step 2 is always halved. Therefore:

$T(n) = T(n/2) + O(1)$

and we can obtain the time complexity of the algorithm $T(n) = O(\log n)$.

Implementation

The sequential implementation is straightforward from the algorithm described in Section Introduction, however, there are a few points to take note of:

1. since prefix minima and suffix minima share the same part of algorithm (Step 1 and Step 2), we can shorten the code by using the same method with a flag to denote the if the current call is prefix minima or suffix minima.

2. Although the algorithm describe above assumes one-based array, the actual implementation use zero-based array, therefore, the odd/even index check in step 3 is reversed.

The OpenMP implementation comes directly from the sequential implementation by parallelising the for loop in Step 1 and Step 3.

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

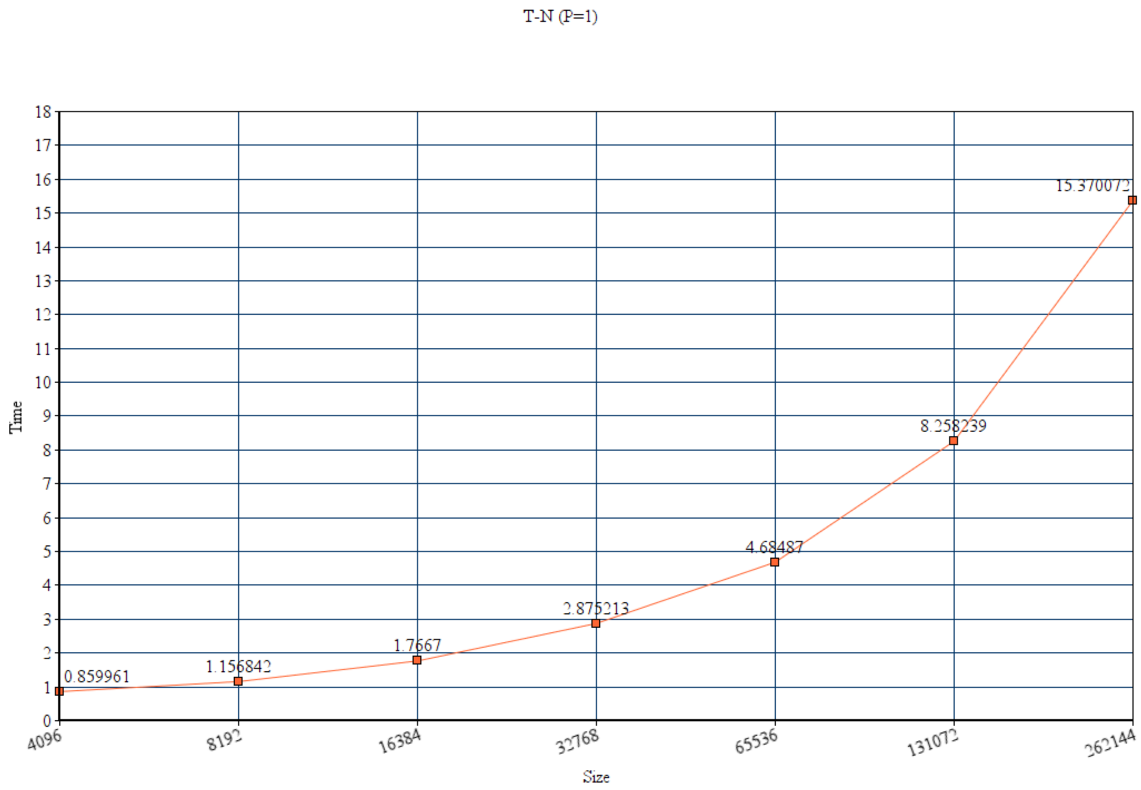
Note:
The graphs generated in this section are not to scale due to the lack of time to write a proper code to generate graph, all credits of graphs goes to onlinecharttool.com

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:

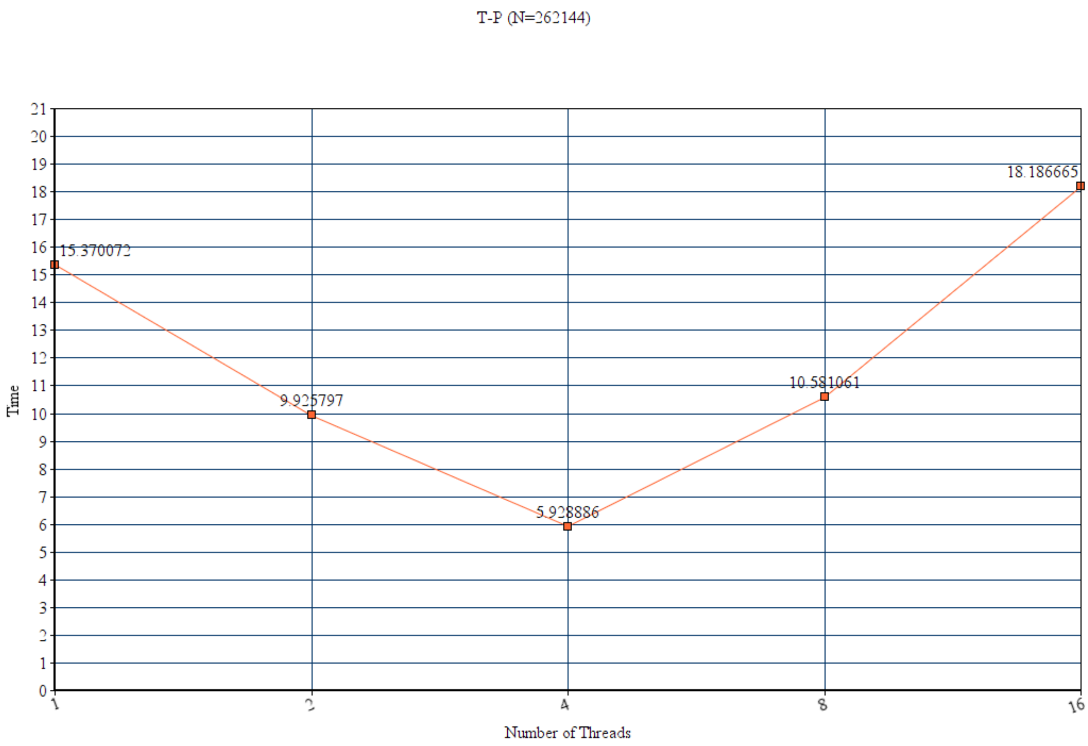
NSize	Iterations	Seq	OpenMP	Th01	Th02	Th04	Th08	Par16
4096	1000	0.178761	0.172715	0.859961	0.970091	1.522321	4.357034	9.037156
8192	1000	0.342576	0.342259	1.156842	1.162653	1.761615	4.711567	9.846965
16384	1000	0.686189	0.684161	1.766700	1.461060	2.164112	5.359732	10.754359
32768	1000	1.372049	1.369079	2.875213	1.993560	2.594418	5.963399	11.999298
65536	1000	2.733934	2.744722	4.684870	3.004588	3.044510	6.902683	12.935802
131072	1000	5.480619	5.496732	8.258239	5.462980	3.891766	8.382706	14.860656
262144	1000	10.954962	10.992808	15.370072	9.925797	5.928886	10.581061	18.186665

From the test result, it is obvious and intuitive to see that the time taken to execute the program increases with NSize. The graph for T(N,P fixed at 1) :



Although the x-axis is not to scale, we can infer from the result that the time taken increases linearly as size of the input grow.

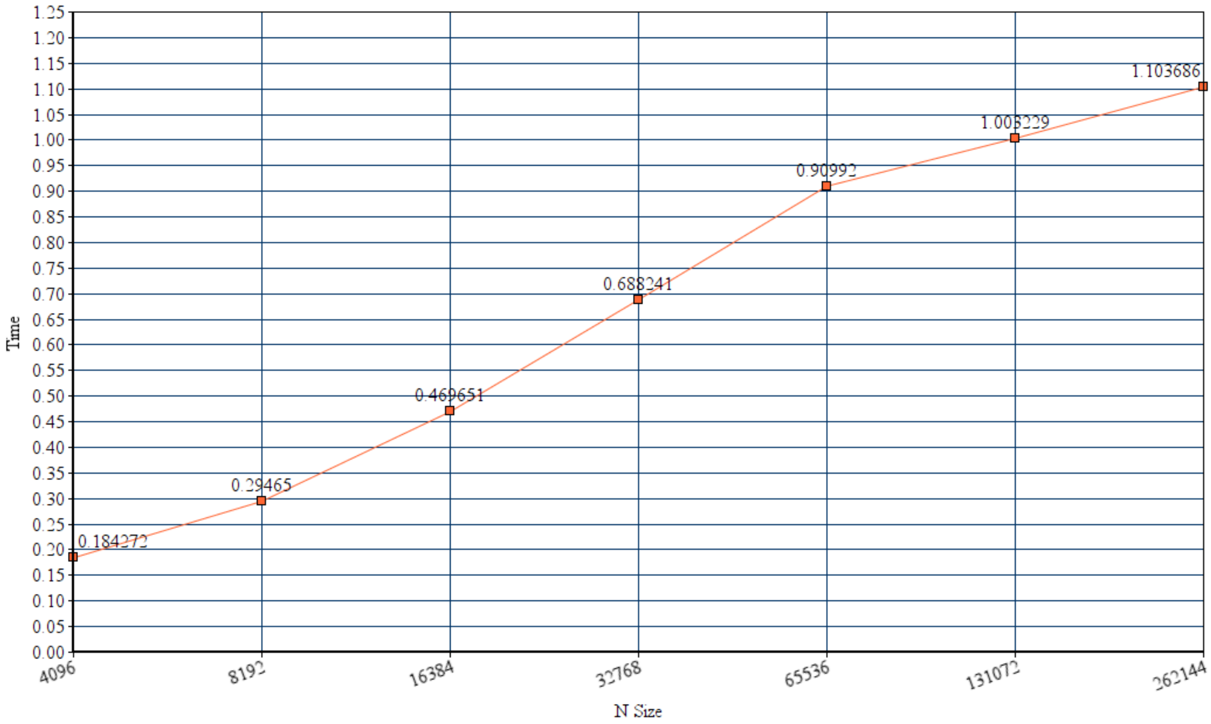
Another interesting graph to see is the T(P, N fixed at 262144) :



Although the x-axis is not to scale, we can still see that there is a decrease when the number of threads increases, which is in accordance to our expectation.
However, the time taken to execute the program increases after certain point. This would be caused by the time overhead of creating and destroying threads. While concurrent execution can save us time, for small calculations, the communication overhead will outweigh the benefit gain if the number of threads go beyond the optimum number.

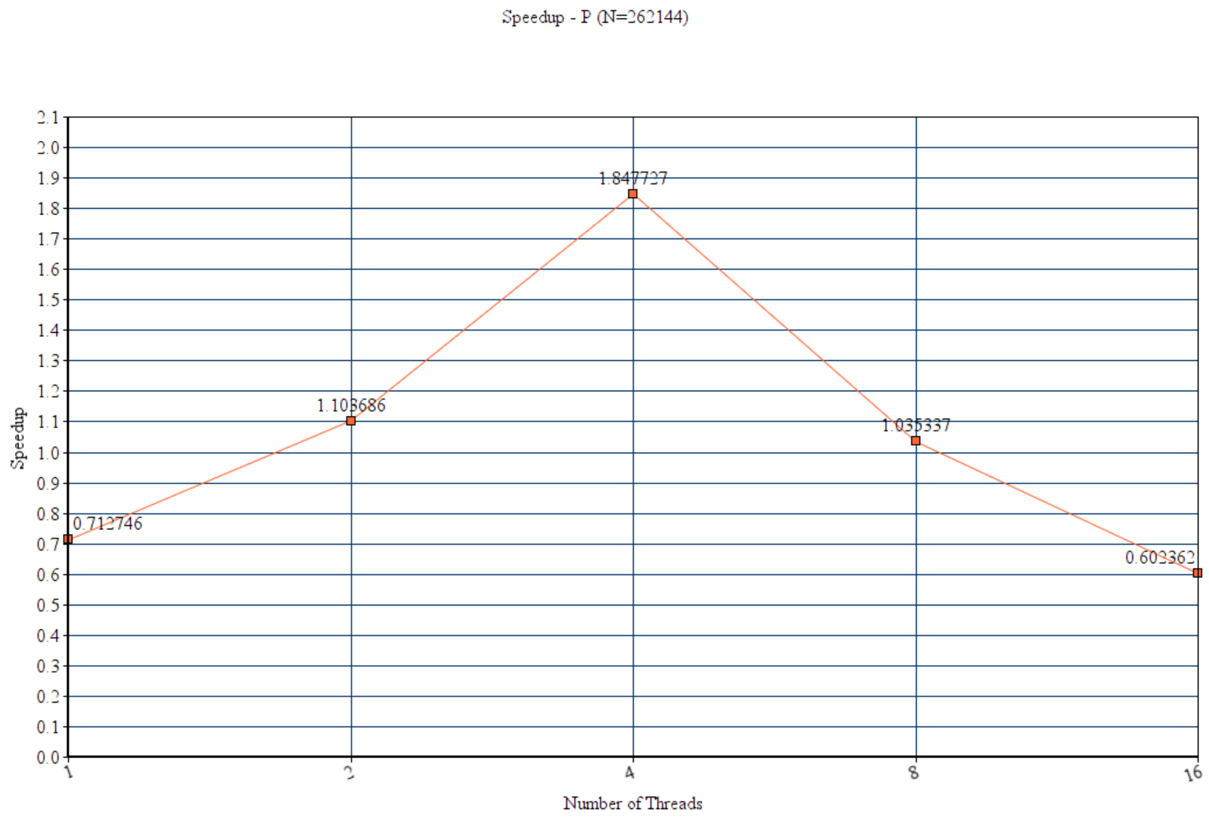
From the above analysis, we can see that for fixed input size, the time taken to execute the program will first decrease as the number of threads grow, then increase after the number of threads surpass a certain point.
Hence, it is intuitive to infer that, for fixed number of threads used in parallel execution, the ratio of the time taken for sequential execution over that for parallel execution will be small if the input size is small, and increase as the input size grow because there are more work to work on, and the overhead of creating and destroying threads will in turn have less weightage.
The Speedup(N, P fixed at 2) graph shown in the next page is in accordance with our intuition.

Speedup - N (P=2)



From the T(P, N fixed at 262144) graph we already observed that when input size is fixed at 262144, the time taken to execute the program will first decrease as the number of threads grow to 4, then increase afterwards.
Since the Speedup is defined as the ratio of time taken for sequential execution over time taken for parallel execution, it is intuitive for us to deduce that when input size is fixed at 262144, the speedup will increase as the number of threads grow to 4, then decrease afterwards.

Indeed, the Speedup(P, N fixed at 262144) graph shown in the next page is in accordance with out deduction.



What is also interesting to see is that the OpenMP parallelisation generally performs the same as the sequential algorithm, and I don't have any inspiring explanation for that at the moment.

Conclusion

In this exercise we have learnt to solving prefix minima and suffix minima problem using the Balanced Tree method and two ways to parallelise the sequential algorithm.

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 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.