Assignment A - Prefix/suffix minima
IN4026 Parallel Algorithms & Parallel Computers

Delft University of Technology

Henrique Dantas, 4172922, H.N.D.M.P.N.Dantas@student.tudelft.nl

July 28, 2013

1. Introduction

For the first assignment, assignment A, it was required to implement an efficient parallel program in PThreads and OpenMP to compute the prefix and suffix minima of a given array. Moreover a sequential implementation from which the remaining are inferred should also be given.

The suffix minima problem is defined as determining the $min\{a_i, a_{i+1}, \ldots, a_N\}$ for each i. Where the array $A = (a_1, \ldots, a_N)$. On the other hand the prefix minima is the $mina_1, a_2, \ldots, a_i$ for each i.

The algorithm to compute the prefix and suffix minima should have a time complexity $\mathcal{O}(\log_2(N))$. The solution that was implemented will be first introduced in this section and further explained in section 2. Sequentially these problems are very simple to solve with one loop that checks what is the current minimum value of the array. However, in order to parallelize this algorithm and obtain the desired time complexity we will need organize the tasks in a binary tree with height $\log_2(N)$.

2. Algorithm Proof

Starting with sequential prefix minima. Initially an auxiliary variable, m, minimum is set as the first value of the array, $P[0]$. Thereafter one simply iterates over the complete P and, at each iteration, i checks, if the current value is lower than the minimum. If that is the case the minimum is updated to the current value of the array. Finally the array values is set as the current minimum, *i.e.* $P[i] = m$. The sequential version of the suffix minima is similar to the previous but the initial i is $N - 1$ (instead of 0), where N is the length of the array. Thereafter the index is decremented until 0. This algorithm has a time complexity of $\mathcal{O}(N)$, since it only has one loop with N iterations.

The parallelized algorithm should have a time complexity of $\mathcal{O}(\log_2(N))$. In order to obtain it[1] we will divide the tasks in a binary tree form, where tasks in the same level can be independently and concurrently executed. Therefore the height (and time complexity) is $\mathcal{O}(\log_2(N))$, as desired. The pseudocode for this algorithm is show in Algorithm 1. The preffixMinima and suffixMinima functions shown simply compare the middle values based on the min_i and max_i values and decide whether to concatenate or update the appropriate subarrays coming from their children.

Algorithm 1 Calculate Prefix and Suffix Minima for A

```
1: Δ ← 1
2: for i ← 0 to log2(N) do
3:    Δ ← Δ · 2
4:    for j ← 0 to N − 1 pardo
5:       prefixMinima(j,j + r − 1); // prefixMinima(min_i, max_i)
6:       suffixMinima(j,j + r − 1); // suffixMinima(min_i, max_i)
7:    end for
8: end for
```

At the bottom of the tree each thread will receive two values and compute the prefix and suffix minima of them. In the next level each thread will receive twice as many inputs, in this case 4. However since they

[1] This algorithm is based on a presentation by Ahmad Khayyat from Queen's University, that can be found at `http://www.slideshare.net/akhayyat/parallel-algorithms-for-trees`

are already sorted it is only necessary to look at the end and beginning value for the arrays coming from the left and right leaf, respectively. By inspecting these two it is possible to understand if the arrays can be concatenated or need to be updated. After finishing this process we have the suffix and prefix minima for those four values. Continuing this process, $\log_2(N)$ times, *i.e.* until N inputs are fed to a single thread we obtain the complete solution.

3. Time-Complexity Analysis

Observing algorithm 1 it is easy to see the time complexity of this algorithm is $\mathcal{O}(\log_2(N))$. The inner loop on line 4 is $\mathcal{O}(1)$ since all tasks are done concurrently. On the other hand the outer loop requires $\mathcal{O}(\log_2(N))$. Therefore the complete algorithm needs as much time as intended.

4. Implementation

This section will address the transition from the pseudocode to the Pthreads and OpenMP implementations. Starting with the latter, in essence only a pair of pragmas were necessary to parallelize the loops. In addition a chunk size was set based on the input size and the number of threads available.

Pthreads requires more work but also more control over the resulting program. Similarly to OpenMP each thread will compute the prefix and suffix minimas based on the outputs from their children. However the threads are spawn by the main thread and later joined. The join occurs when all tasks for that level are finished or when there are more tasks than threads. No barriers were used in the implementation.

5. Results

The program was tested with the example inputs from the manual and the results are summarized in table 5.1. The machine was the one provided by the course and was accessed remotely via ssh.

	NSize	Iterations	Seq	Th01	Th02	Th04	Th08	Th16
Pthreads	32	1000	0.000955	1.056421	0.578142	0.583764	2.072299	1.680798
OpenMP	32	1000	0.001256	0.010129	0.037573	0.087985	0.164634	0.276545

Table 5.1: Timing results for the Parallel Implementations

From table 5.1 one can see that for the input size used a threaded implementation is not advantageous. The work each task is responsible for is too small to compensate for the communication overhead. In addition the Pthreads implementation is significantly slower that OpenMP. Therefore it is likely not implemented in the most efficient way and should be revised for better performance. Nonetheless, the best OpenMP result is still around 10 times slower that the sequential algorithm.

6. Conclusions

In this assignment it was required to implement a threaded algorithm to compute the prefix and suffix minima using Pthreads and OpenMP. The algorithm chosen has a time complexity of $\mathcal{O}(\log_2(N))$ and it is based on a binary tree. From section 5 one can infer that for the input size used the communication overhead overwhelms the gains from concurrent execution.

On a personal note, this was the first time I programmed using threads. Therefore it was challenging, in particular at the beginning with Pthreads, to convert the pseudocode to a real, working code. Likely for that reason the results from Pthreads are not as satisfactory as OpenMP's.