Assignment C - List Ranking (Pointer Jumping)
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

The goal of the third and last assignment, assignment C, was to implement a pointer jumping algorithm. This should read input S and write to an array R such that each value $R(i)$ is the distance from node i to the end (last node) of S. The desired time complexity is $\mathcal{O}(\log_2(N))$, where N is the length of S.

The particular example that will be presented is a subset of the pointer jumping as each node only points to another, and no node has more that two pointers to it. It resembles a linked list.

In addition a sequential implementation from which the remaining are inferred should also be given.

According to lecture 2 of this course, "Pointer jumping is a technique suitable for fast access in pointer accessible data structures having the form of a forest of rooted-directed trees". In our case the root of the forest is $S(i_{\text{Root}}) = 0$.

In order to do parallel pointer jumping at each iteration the path (or the jump) doubles in size. In other words, update the successor of a node with that successor's successor until the root is found. It is important to note that the successor of the root is itself. No other node has this property.

This algorithm will be further explained in section 2.

The sequential version of this algorithm is essentially the same as the parallel except it does not use threads, thus we will refrain from explaining it.

2. Algorithm Proof

The parallelized algorithm should have a time complexity of $\mathcal{O}(\log_2(N))$. In order to obtain it[1] we will use the parallel strategy to solve the pointer jumping problem.

The pseudocode for this algorithm is show in Algorithm 1. Prior to running this algorithm the array D is set to 1 if the node i is not the root, *i.e.* if $S[i] \; != 0$.

Algorithm 1 Merge ordered arrays A and B to C

```
1: for $i \leftarrow 1$ to $N$ pardo
2:   if S[i] != S[S[i]] then
3:     $D[i] \leftarrow D[i] + D[S[i]]$;
4:     $S[i] \leftarrow S[S[i]]$;
5:   else
6:     $D[i] \leftarrow D[i]$;
7:     $S[i] \leftarrow S[i]$;
8:   end if
9: end for
```

To verify the correctness of the algorithm each step will be explained and thereafter it will be proven by induction.

The initial if checks if the successor of current index is equal to its successor's successor. If that is the case it means it is already pointing to the root and the values are not updated. Otherwise the D, which accounts for the distance to the root is updated with the old distance value plus the distance value of the successor. In addition, S is set to the value of the current node successor's successor.

[1]This algorithm is based on lecture number 2 from Parallel Parallel Algorithms and Parallel Computers

If we start at a leaf of the tree it is clear that at each step the node will point further away from its original parent. In particular, it always doubles the distance until it reaches the root. Also it updates the D array every time it does so to keep track of how many hops it is from the root. Running this algorithm $\log_2(N)$ times for all nodes it is simple to understand that D will be equal to the distance of the original node to the root.

3. Time-Complexity Analysis

The time complexity analysis is very simple for this algorithm. Since at each iteration the node points two times as far as in the previous it will require $\log_2(N)$ iterations to reach the root for the node farthest away. Hence the complexity is $\mathcal{O}(\log_2(N))$.

4. Implementation

Both implementations are very simple and remarkably close to the pseudocode shown in Algorithm 1. A small detail is that a couple of auxiliary arrays, D_{new} and S_{new} were added. These were responsible to update the D and S arrays, respectively, at the end of each iteration. The auxiliary arrays were necessary to ensure that each thread read the value for the current iteration and not for the next.

In OpemMP two pgramas were used to schedule the three loops. The chunk size used is the ratio between the input size and the number of available threads. In addition a pragma to initialize the parallel section was also set, through which the number of threads was specified.

Pthreads is very similar. As usual the threads are spawned and joined (at the end of the algorithm) by the main thread. Each thread synchronizes at the end of each iteration through the use of a barrier.

5. Results

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

	NSize	Iterations	Seq	Th01	Th02	Th04	Th08	Th16
Pthreads	16	1000	0.002624	0.042643	0.120167	0.223224	1.061668	3.544720
OpenMP	16	1000	0.002823	0.008643	0.026304	0.040445	0.047708	0.060699

Table 5.1: Timing results for the Parallel Implementations

From table 5.1 it is perceptible that given the current input, the threaded implementations offer no advantage over the sequential one. Each thread should be assigned more work in order to compensate for the communication overhead. Added to this the Pthreads implementation is significantly slower than OpenMP. A possible, and most obvious, explanation is that PThreads is not implemented in the most efficient manner. Regardless, OpenMP is still much slower than the sequential versions.

6. Conclusions

In this assignment we were asked to implement a parallel solution for the pointer jumping problem. To fulfill that task an algorithm present at the second lecture of this Course lectures was chosen. In addition this algorithm has a time complexity equal to $\mathcal{O}(\log_2(N))$. From section 5 one can conclude that for the input size used the communication overhead is much greater than the gains from parallel execution.