Assignment B - Simple Merge

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 second assignment, assignment B, it was required to merge two arrays A and B. It shall be implement in PThreads and OpenMP. The algorithm is based on the rank operation and should have a time complexity equal to $\mathcal{O}(\log_2(N+M))$, where N and M are the sizes of A and B, respectively. In addition a sequential implementation from which the remaining are inferred should also be given. Both arrays A and B are ordered.

The rank operation, $\text{rank}(x : X)$ returns the number of elements from X smaller than or equal to x. Since the input arrays are sorted computing the $\text{rank}(a : AB)$, where AB is the concatenation of A and B, returns the position of value a in the merged array.

To run the ranking operations in parallel we can split each array in subsets (A_i and B_j) and merge them independently. This is done in such a way that the resulting merged sequences $C_1, \ldots, C_k$ can be simply concatenated to obtain the solution.

The algorithm will be further explained in section 2.

Once again, sequentially this problem is trivial. Simply iterate over A and B, select the minimum value from both and iterate the respective index (i for A and j for B). Then, assign this value to C, the merged array and iterate the appropriate index, k.

2. Algorithm Proof

The parallelized algorithm should have a time complexity of $\mathcal{O}(\log_2(N+M))$. In order to obtain it[1] we will divide the array A in $\log_2(N)$ subsets and B in $\log_2(M)$ subsets. Thereafter each set will be merged with the other concurrently. Parallel Algorithms and Parallel Computers. Finally all are concatenated.

Intuitively one can infer this algorithm is correct. Since we consider all the values and they are ordered, at the end we will obtain the expect result.

The pseudocode for this algorithm is show in Algorithm 1. The auxiliary function $\text{seqmerge}(i, j)$ sequentially merges (using balanced tree) $A[i+1, i+2, \ldots]$ and $B[j+1, j+2, \ldots]$ into $C[i+j+j, i+j+2, \ldots]$ until it encounters an index for which a value for C has already been assigned.

3. Time-Complexity Analysis

The algorithm shown has three distinct steps that are executed concurrently. The first performs the rank operations and has a time complexity of $\mathcal{O}(\log_2(N+M))$ since each block has $\mathcal{O}(\log_2 n + m)$ elements.

The seconds assigns the 'border' values to C based on the previously computed rank values. These values will be later be checked by seqmerge. The time complexity of this step is the same as the first.

Finally the results are merged together in $\mathcal{O}(\log_2((n+m)/\log_2(n+m)))$.

Therefore the total time complexity is $\mathcal{O}(\log_2 n + m)$, as intended.

4. Implementation

Both implementations closely mimics the pseudocode shown in Algorithm 1. Three pgramas were used to schedule the three loops. The chunk is the ratio between the maximum input size and the number

[1] This algorithm is based on lecture number 5 from Parallel Parallel Algorithms and Parallel Computers

Algorithm 1 Merge ordered arrays A and B to C

```
1: $a \leftarrow \log_2(n)$; $b \leftarrow \log_2(m)$; $AA[0] \leftarrow 0$;
2: for $i \leftarrow 1$ to $n/a$, $j \leftarrow 1$ to $m/b$ pardo
3:     $AA[i] \leftarrow \mathrm{rank}(A[i \times a] - 1, B)$;
4:     $BB[j] \leftarrow \mathrm{rank}(B[j \times b], A)$;
5: end for
6: for $i \leftarrow 1$ to $n/a$, $j \leftarrow 1$ to $m/b$ pardo
7:     $C[AA[i] + a \times i] \leftarrow A[a \times i]$;
8:     $C[BB[i] + b \times i] \leftarrow B[b \times i]$;
9: end for
10: for $i \leftarrow 1$ to $n/a$, $j \leftarrow 1$ to $m/b$ pardo
11:     seqmerge($i \cdot a, AA[i]$);
12:     seqmerge($BB[j], j \cdot b$);
13: end for
```

of available threads. In addition a pragma to initialize the parallel area was also set, through which the number of threads was specified.

Pthreads is very similar to the just described OpenMP implementation. This time the threads are spawn by the main and synchronize (through a barrier) after the second step, since the next may need values computed by other threads.

The threads are joined after the algorithm finishes.

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	32	1000	0.001347	0.049287	0.076389	0.136729	0.758988	1.527234
OpenMP	32	1000	0.001391	0.009754	0.012192	0.022821	0.021166	0.024526

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 write an algorithm to merge two arrays based on the rank operation. For that, a algorithm from the Course lectures was chosen as it fulfilled the time complexity requirements, $\mathcal{O}(\log_2(N+M))$ From section 5 one can conclude that for the input size used the communication overhead is much greater than the gains from parallel execution.