IN4026 Assignment B - Simple Merge

He Cheng 4420837

August 30, 2015

1 Introduction

The requirement of this assignment is to design and implement an efficient parallel program with time complexity O(log(M+N)) in OpenMP and PThreads that merges two sorted sequence, where M and N are respectively the sizes of two arrays A and B. The merge algorithm is based on the rank operation rank(x:X) that is the number of elements of X smaller that or equal to x. Merging array A and array B equals to determining rank(A:AB) and rank(B:AB), where AB is the concatenation of A and B.

2 Algorithm Proof

The pseudocode for the algorithm is shown below.

Algorithm 1 Simple Merge

1. $a \leftarrow log(n)$; $b \leftarrow log(m)$
2. **for** $i \leftarrow 1$ **to** n/a, $j \leftarrow 1$ **to** m/b **pardo**
   $AA[i] \leftarrow rank[A[i \times a], B]$;
   $BB[j] \leftarrow rank[B[j \times b], A]$;
   end for
3. **for** $i \leftarrow 1$ **to** n, $j \leftarrow 1$ **to** m **pardo**
   $C[AA[i] + a \times i] \leftarrow [A[a \times i]$;
   $C[BB[j] + b \times j] \leftarrow B[b \times j]$;
   end for
4. **for** i $\leftarrow 1$ **to** n, $j \leftarrow 1$ **to** m **pardo**
   $merge(a \times i, AA[i])$;
   $merge(BB[j], b \times j)$;
   end for

In order to the time-complexity O(log(M+N)), first array A is divided into log(N) subsets and B in log(M) subsets. Then each set is merged with the other concurrently. Since A and B are both already arranged in order, it is obvious that generates the right result, supported by executing program test.

3 Analysis of the Time-complexity

The time-complexity of four steps is analyzed as below. The first step has a time-complexity of O(1). The second has a time-complexity of O(log(M+N)) because rank function is executed and each for loop has O(log(M+N)) steps. The time complexity of the third step is O(1). The fourth step including merge-sort function has a time-complexity of O(log(M+N)).To sum up, the proposed algorithm has a time-complexity of O(log(M+N)).

4 Implementation

The sequential implementation is translated directly from the pseudocode. In the OpenMP implementation, the chunk, the ratio of the input size and the number of available threads, is first decided and then pragmas are used to dynamically schedule the three for loops. Pthreads implementation is similar to but more complicated than OpenMP implementation. In the Pthreads version, barriers are used to synchronize data by forcing threads to wait until all threads finish. A thread joins only after prior threads terminate, which is realized by pthread$_{j}oinfunction$.

5 Program Test

In this section, the execution time of the algorithm is measured at different problem sizes and with different number of threads, shown below.

Table 1: OpenMP Execution Time (1000 Times)

NSize	Seq	Th01	Th02	Th04	Th08	Th16
32	0.015575	0.015922	0.009781	0.010473	0.011900	0.012340
64	0.055018	0.055865	0.029446	0.022618	0.020000	0.018039
128	0.204099	0.204772	0.106749	0.061272	0.038252	0.037125
256	0.704075	0.376752	0.189566	0.199550	0.106168	0.104071
512	1.874564	1.477282	0.739827	0.718588	0.251471	0.198003
1024	6.305684	5.850432	2.927295	1.935513	0.835368	0.726014
2048	23.623991	23.281045	11.646865	5.855320	4.370193	2.915495
4096	93.137763	92.852298	48.312040	23.281677	12.763069	11.536784

Table 2: Pthreads Execution Time (1000 Times)

NSize	Seq	Th01	Th02	Th04	Th08	Th16
32	0.015883	0.036213	0.053640	0.092154	0.297106	0.605734
64	0.054978	0.049431	0.061537	0.083755	0.305868	0.619504
128	0.203878	0.084882	0.078789	0.084974	0.296085	0.633370
256	0.784498	0.210153	0.170377	0.125678	0.273496	0.618420
512	1.842923	0.531267	0.339223	0.262141	0.410365	0.608749
1024	6.285258	1.535931	0.912870	0.638165	0.726852	0.742136
2048	23.596675	4.005192	2.484563	1.437967	1.293210	1.534184
4096	93.237077	12.164975	8.313629	4.091729	2.980654	3.305902

According to the two table above, the speed-ups of OpenMP and Pthreads are calculated as below.

Table 3: OpenMP Speed-ups

NSize	Th01	Th02	Th04	Th08	Th16
32	0.978	1.592	1.487	1.299	1.262
64	0.985	1.868	2.432	2.751	3.050
128	0.997	1.912	3.331	5.336	5.498
256	1.869	3.714	3.528	6.632	6.765
512	1.269	2.534	2.609	7.454	9.467
1024	1.078	2.154	3.258	7.549	8.685
2048	1.015	2.028	4.035	5.406	8.103
4096	1.003	1.928	4.000	7.297	8.073

Table 4: Pthreads Speed-ups

NSize	Th01	Th02	Th04	Th08	Th16
32	0.439	0.296	0.172	0.053	0.026
64 8	1.112	0.893	0.656	0.180	0.089
128	2.402	2.588	2.399	0.689	0.322
256	3.733	4.604	6.242	2.868	1.269
512	3.469	5.433	7.030	4.491	3.027
1024	4.092	6.885	9.849	8.647	8.469
2048	5.892	9.497	16.410	18.247	15.381
4096	7.664	11.215	22.787	31.281	28.203

Since usually the 8-core computer can handle 8 threads concurrently at most, it is expected that running the algorithm with 8 threads would achieve the largest speed-ups if the input size is large enough. The result of Pthreads implementation is as what I expect. However, in OpenMP implementation, running the algorithm with 16 threads obtains better performance than 8 threads. Maybe it is because this 8-core computer can run 16 threads concurrently. With

the growth of input size, the performance of 16 threads may exceed the performance of 8 threads in Pthreads implementation.

6 Conclusion

In this assignment, OpenMP and Pthreads are used to implement the simple merge. Both implementations achieves significant speed-ups when input size is relatively large because in this case the decreasing execution time benefiting from parallel execution is much larger than the overheads generated by OpenMP mechanism and Pthreads mechanism.