CS515 Parallel Programming: Assignment #2

Due on May 4th, 2016 at 11:59pm

Jingke Li Spring 2016

Konstantin Macarenco

Primes

As you can see from the graph - parallel Primes algorithm performed better than sequential version in all cases, with one exception: OMP with one thread is always slower than the sequential implementation. This can be explained by overhead that is introduced by using OMP.
There are two possible solutions to the homework

1. Each thread works independently and starts crossing out multiples as fast as it finds a non zero entry. The main disadvantage of this approach is that, since there is no synchronization some thread will perform redundant jobs, and cross out some entries multiple times.

2. Implement synchronization to avoid redundant job. In this case we need set of explicit locks, waits, array partitioning, and a way to notify other threads about index being crossed out. This most likely will result in worse performance than the first case.

I used the first solution, and you can observe that increasing number of threads gives small performance boost, with 128 being the fastest.

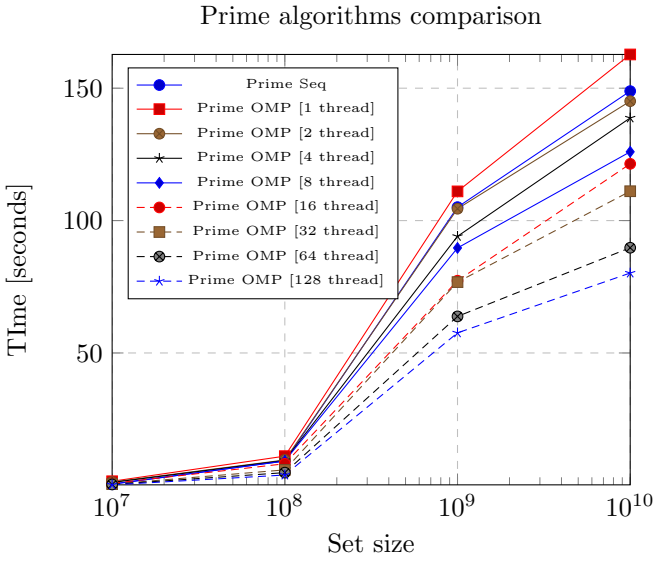


Table 1: Sequential execution (Time [sec])

	10^7	10^8	10^9	10^{10}
prime	0.533	9.616	105.080	148.891
quicksort	4.247	39.907	425.748	653.039

Table 2: Prime omp (Time [sec])

	10^7	10^8	10^9	10^{10}
1	1.548	11.032	111.028	162.742
2	1.268	9.572	104.474	145.092
4	1.097	9.413	94.059	138.766
8	0.400	9.203	89.660	125.969
16	0.753	8.149	77.292	121.433
32	0.478	5.845	76.765	111.086
64	0.337	4.807	63.811	89.739
128	0.192	3.904	57.598	80.190

Quick sort

Quick sort is much more suitable for parallelization since we only need to wait for the current chunk of the array to be partitioned and no other synchronization. At first I had trouble with placing OMP statements in the correct place. When pragma's are added to the quickSort routine, every time the method called new parallel region is declared, which creates exponential of number of threads, and leads to poor performance.

After some research I used "# pragma omp task firstprivate(array, low, high)" to declare a region that should be executed by a separate thread thread, and firstprivate indicates that array, low and high must be initialized before entering parallel region. In this case an existing thread is assigned new parallel region.

This approach nearly doubles the performance as soon as number of threads is two or more. Further increase of threads, gives slower performance boost, about %20 percent with each level (up to 16 threads). After number of threads reached 32 we can see some performance degradation by %8 in average (with each thread number increase). This behaviour usually rises when number of threads exceeds number of available work, managing these waiting threads creates additional penalty. Hence the slow down. And in this case threads have to wait for partitioning to be complete, and new values for low and high to be computed. According to my test run most optimal number of cores for qsort is 16.
Another thing worth mentioning, one thread OMP quicksort struggles from the same issue as prime, it is slower than sequential version by a constant factor.

qsort-pthd

Pthread version of qsort is faster than sequential, but ($\approx 2x$ in the best case) slower than OMP. This is expected, since qsort-pthd has multiple bottlenecks, such as shared number of producing methods, and queue. This version's performance peak is at NUMTHREADS=8, as number of threads increased performance drops to sequential level or much worse. In all tests, except 10^9, Pthread 32 threads performed the worst. I expected to see Pthd 128 being the slowest in all cases. This discrepancy, could be related to Babbage being a shared machine, with variable load.

PTHD vs OMP revisited

I decided to re-test quicksort to see if Pthread performance inconsistency is a result of public machine variable load, except this time I excluded array initialization from time measurement, since it can take up to %40 of total time. One complication I had is granularity of such a test, standard clock() function is useless since it gives a total CPU cycles of all participated cores, and C time library maximum granularity is 1 second. I omitted tests for world sizes 10^7 and less, since they were slower than 1 sec, and world size 10^{10}, since machine was too busy at the time.
OMP quicksort is about 3x time faster than the application with array init accounted for, where doubling threads gave about the same performance increase, with most optimal number of threads = 16.
On the other hand, Pthread quicksort had similar performance, with one exception: slowest case "threads = 16", not 32 as in the original run. Performance decline was again sporadic, with "threads = 128" being faster, than the worse case.

Other issues

When number of threads is greater than number of available cores, there most likely be slowdown, unless some cores are waiting and can run additional threads.

Results continued on the next page...

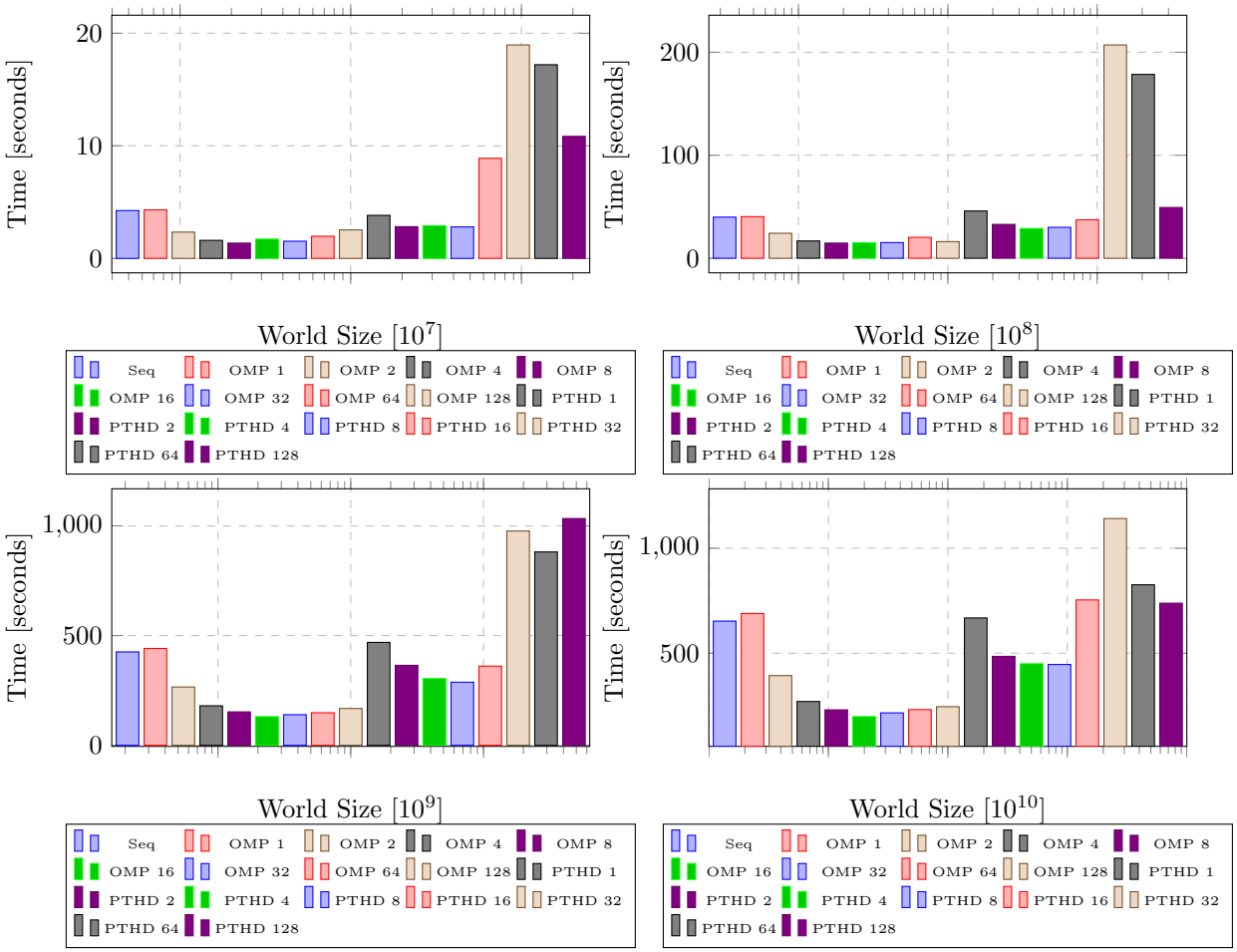
Quicksort OMP vs Pthread (Array init time t included)

Table 3: Quick Sort omp (Time [sec])

	10^7	10^8	10^9	10^{10}
1	4.313	40.367	441.530	689.361
2	2.336	24.254	265.905	393.473
4	1.603	16.793	180.163	270.030
8	1.359	14.699	152.552	230.375
16	1.710	14.954	130.645	197.896
32	1.527	15.191	139.620	215.855
64	1.965	20.295	148.494	231.565
128	2.531	16.134	167.827	245.310

Table 4: Quick Sort pthread (Time [sec])

	10^7	10^8	10^9	10^{10}
1	3.825	45.857	469.146	668.053
2	2.820	32.870	364.625	485.091
4	2.897	28.845	303.976	449.671
8	2.796	30.014	287.594	446.297
16	8.892	37.327	361.015	753.872
32	18.957	207.082	976.707	1141.691
64	17.203	178.471	881.260	826.079
128	10.841	49.304	1033.700	738.495



Quicksort OMP vs Pthread (Array init time is not included)

Table 5: Quick Sort OMP (no init) (Time [sec])

	10^8	10^9
1	32	371
2	16	188
4	10	99
8	5	61
16	5	42
32	6	52
64	7	60
128	10	80

Table 6: Quick Sort Pthread (no init)(Time [sec])

	10^8	10^9
1	36	408
2	27	264
4	22	231
8	22	224
16	33	348
32	58	920
64	51	582
128	44	548