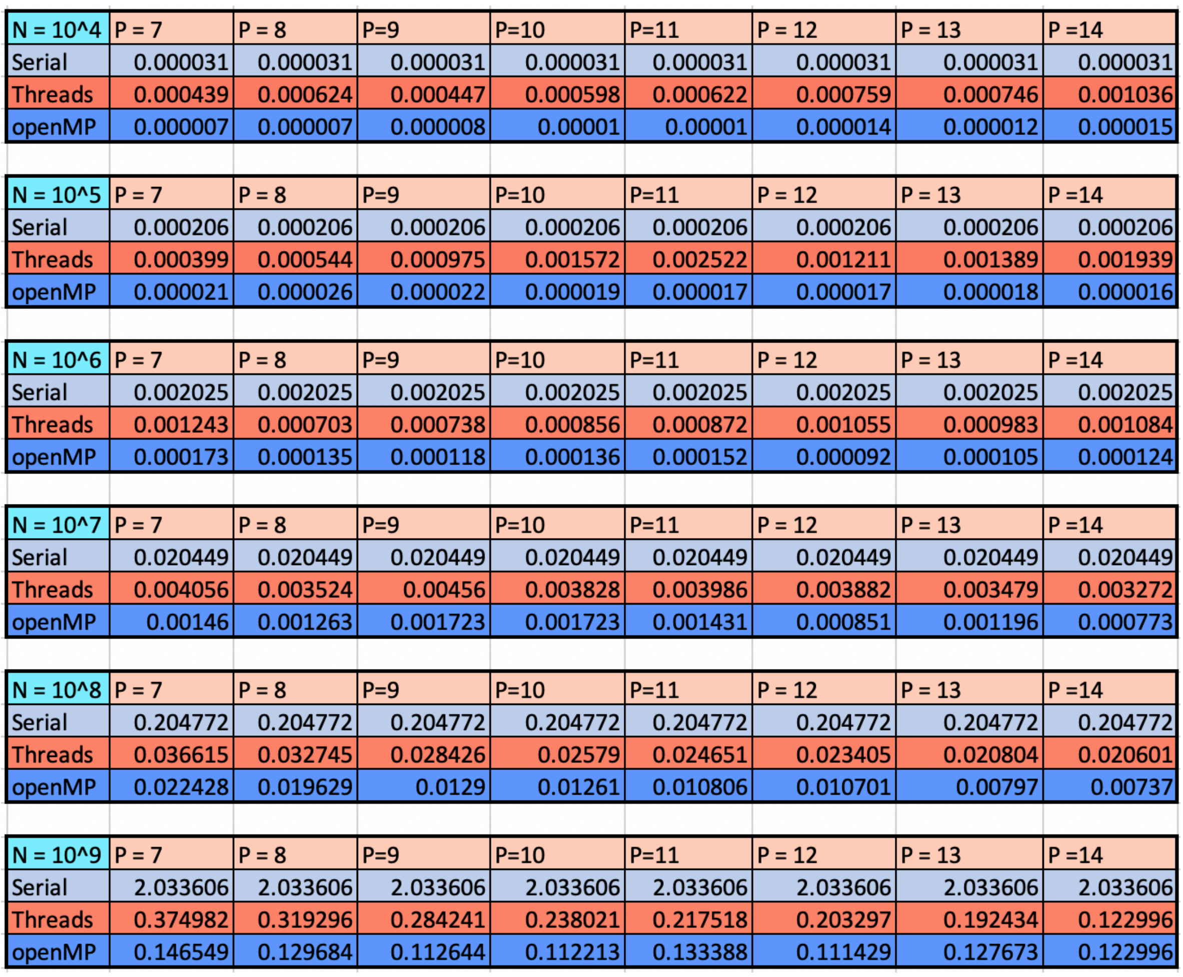
Daniel Marzec
djm6267
3/24/19

Homework 3 : OpenMP

Part1:

For our website that takes in a large amount of survey data and needs to determine the mean and standard deviation of the data, we have a few options for how to proceed with the calculations. Ignoring internet latency and other factors, we will be examining the specific computation time over a variety of N values and P values, as the survey participants and available computational resources is not something that will always be constant for our complex project. Thus we chose to examine three separate implementations, serial, stdthreads, and openMP based on N values ranging from 10^4 to 10^9 as well as number of threads ranging from 7 to 14. These ranges are appropriate given that our surveys require at least 1000 responses for statistical significance and we will almost never exceed 1 billion participants. Our base computational set up will be able to support 7 threads with ease. Below this paragraph you will find a data table of the time recorded for each computation with stated N and P values.

Timing Table (s): (Serial timing is the average across 8 runs since timing varied minimally)



N = 10^4	P = 7	P = 8	P=9	P=10	P=11	P = 12	P = 13	P =14
Serial	0.000031	0.000031	0.000031	0.000031	0.000031	0.000031	0.000031	0.000031
Threads	0.000439	0.000624	0.000447	0.000598	0.000622	0.000759	0.000746	0.001036
openMP	0.000007	0.000007	0.000008	0.00001	0.00001	0.000014	0.000012	0.000015

N = 10^5	P = 7	P = 8	P=9	P=10	P=11	P = 12	P = 13	P =14
Serial	0.000206	0.000206	0.000206	0.000206	0.000206	0.000206	0.000206	0.000206
Threads	0.000399	0.000544	0.000975	0.001572	0.002522	0.001211	0.001389	0.001939
openMP	0.000021	0.000026	0.000022	0.000019	0.000017	0.000017	0.000018	0.000016

N = 10^6	P = 7	P = 8	P=9	P=10	P=11	P = 12	P = 13	P =14
Serial	0.002025	0.002025	0.002025	0.002025	0.002025	0.002025	0.002025	0.002025
Threads	0.001243	0.000703	0.000738	0.000856	0.000872	0.001055	0.000983	0.001084
openMP	0.000173	0.000135	0.000118	0.000136	0.000152	0.000092	0.000105	0.000124

N = 10^7	P = 7	P = 8	P=9	P=10	P=11	P = 12	P = 13	P =14
Serial	0.020449	0.020449	0.020449	0.020449	0.020449	0.020449	0.020449	0.020449
Threads	0.004056	0.003524	0.00456	0.003828	0.003986	0.003882	0.003479	0.003272
openMP	0.00146	0.001263	0.001723	0.001723	0.001431	0.000851	0.001196	0.000773

N = 10^8	P = 7	P = 8	P=9	P=10	P=11	P = 12	P = 13	P =14
Serial	0.204772	0.204772	0.204772	0.204772	0.204772	0.204772	0.204772	0.204772
Threads	0.036615	0.032745	0.028426	0.02579	0.024651	0.023405	0.020804	0.020601
openMP	0.022428	0.019629	0.0129	0.01261	0.010806	0.010701	0.00797	0.00737

N = 10^9	P = 7	P = 8	P=9	P=10	P=11	P = 12	P = 13	P =14
Serial	2.033606	2.033606	2.033606	2.033606	2.033606	2.033606	2.033606	2.033606
Threads	0.374982	0.319296	0.284241	0.238021	0.217518	0.203297	0.192434	0.122996
openMP	0.146549	0.129684	0.112644	0.112213	0.133388	0.111429	0.127673	0.122996

Observations 1:_As you can see, there are a few quick trends that can be deduced. It appears that for N = 10^4 and N = 10^5 , openMP out performs stdthreads as well as serial. The more interesting result is that serial actually out performs stdthreads by almost an average factor of 10 for almost all number of threads. The serial implementation actually does not use the number of threads since it is serial. For such a small computation, the overhead required to set up stdthreads is much larger than the time required for computation. Of course, as the number of threads increases for openMP, it also increases it's overhead causing slower slightly slower times for this range, but in comparison to the times it takes for stdthreads and serial this difference is extremely negligible. Clearly for the lower end of data calculations, openMP wins as it scales the best and has the least overhead. In fact, according to reference A, this is due to the fact that thread pooling allows for much lower overhead for initializing threads than standard threads.

Observations 2: Jumping into the N = 10^6 and N = 10^7 range, we see that again openMP is the clear winner. Serial in fact outperforms stdthreads for almost all number of threads at N= 10^6 but finaly gets overtaken by stdthreads at N=10^7 for all values and for all number of threads. For N = 10^7, the number of threads does not scale very well for stdthreads, but as openMP's threads increase it achieves a 2x improvement in time where as threads sees slightly below a 1.5x improvement. It seems that as the computational requirements are increasing threads is beginning to perform much better, but still not as good as openMP which is again the winner for this section.

Observations 3: For the upper end of our computation needs, N = 10^8 and N= 10^9, openMP seems to outperform all methods again. Stdthreads finally has a significant performance advantage over serial for every data point. This is very interesting because it seems that once stdthreads begins firing at full steam it can actually equal the time of openMp at N=10^9 and threads equal to 14 (of course this is due to rounding in excel and openMP actually performed ever so slightly better). As can seen threads gains an almost 2x speed up from P =7 to P = 14 in N= 10^9 where as openMP does not get much better past P = 10. In the N = 10^8 range, openMP scales amazingly well, where as stdthreads appears lackluster in comparison. Therefore, openMP wins this section of data as well.

Final Recommendation: In the best interest of our complex project, openMP is my recommendation for computational method. Serial clearly is not beneficial in any scenario. While it would be very interesting to examine how stdthreads compares to openMP for N greater than N=10^9, it appears that the overhead in the lower range makes it pale in comparison to openMP. OpenMP has the best performance time amongst all of the data range that we require and the least overhead due to thread pooling and thus it is my final recommendation. In the future if computational requirements keep increasing, it would be interesting to test the data for stdthreads after N=10^9, since it appears in this range the number of threads begins to have minimal scaling performance improvement. The best range to scale our number of threads in openMP appears to be in the N=10^7 and N =10^8 range, with very minimal advantage to scaling in the N=10^9 range.

Part2:

This part of the assignment proved to be much more tricky than I had anticipated, hence my resubmission for it. Essentially what I did was take the OMP code I had and extend its functionality. To do this, after the standard deviation calculation I calculated the upper and lower threshold for which values would be selected based of the formulas: upper threshold = mean + user entered sigma * standard deviation, and lower threshold = mean – user entered sigma * standard deviation. This proved to be the easiest part. From there I created a vector of vectors of pairs of size, I know we were warned of the performance inefficiency of vectors but it seemed the only way to continuously add to a list without dynamic allocation or over allocating memory for an array, but most likely I may be missing something. From there I wrote a parallel region and then for each of the P portions check every value to see if it was above or below the threshold. From there I had a vector of multiple vectors, each a different size. This was easy to combine into one vector by just using one for loop. I added each of the original values and index value that was in the original data from the array.

Reference List:

Reference A: https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code