HOME WORK 2 – OpenMP Threading

November 9, 2017

Introduction

This report is a basic summary of the N-body algorithm. N-body is an algorithm that predicts the individual motion of a group of objects (here-in referred to as particles), interacting with each other gravitationally. Its design emphasizes a series of experiments that generates a benchmark for analyzing parallel optimization with Strong and Weak scaling.

The following pages are a series of observations, deduced after working on tasks in Homework2 instruction using an Intel++ compiler on a private KNL node in Stampede2.

Code Modification

The step was to add a definition header, by so doing, the compiler acknowledges the implementation of OpenMP library. OpenMP is a standardized library for Parallel and Scientific Computing that supports both Fortran and C++.

Next steps were to twerk the acceleration, update and search routines by including an OpenMP parallel pragmas to the outer loop of each one of them respectively. This is enable the instantiating threads that will statically partition the loops across threads.

To ascertain the number of threads used at runtime, an OpenMP API function (omp_get_max_threads()) was used. Also, the code is setup such that the average time it takes to run the code, the number of particles specified, the memory used, the number of steps specified, the acceleration, the update, and the search, as well as the number of threads used to run the program are printed out especially for purpose of analysis.
Please see codes for actual implementation

Benchmark Analysis

Table Key

PARTICLES: Number of Particles
THREADS: Number of Threads
TIME: Runtime in microseconds
SPEEDUP: $S_p = T_s / T_p$
Where;
Sp = Speed-up,
Ts = Execution time of the sequential Algorithm and
Tp = Execution time of the parallel algorithm with P cores.
EFFICIENCY: $E_p = T_s / pT$
Where;
Ep = Efficiency
P = Number of Threads

Strong Scalability

Strong Scaling: Serial Benchmark

PARTICLES	RUNTIME
1000	2.586944
2000	10.07744
4000	42.55328
8000	175.8830

Strong Scaling: Parallel Benchmark for 1000 Particles

THREADS	TIME	SPEEDUP	EFFICIENCY
2	1.296650	1.995098	0.997548
4	0.738629	3.502359	0.875589
8	0.405520	6.379325	0.797416
16	0.316237	10.83537	0.511275
32	0.318190	8.130186	0.254068
64	0.527409	4.905001	0.076641

Strong Scaling: Parallel Benchmarks for 2000 Particles

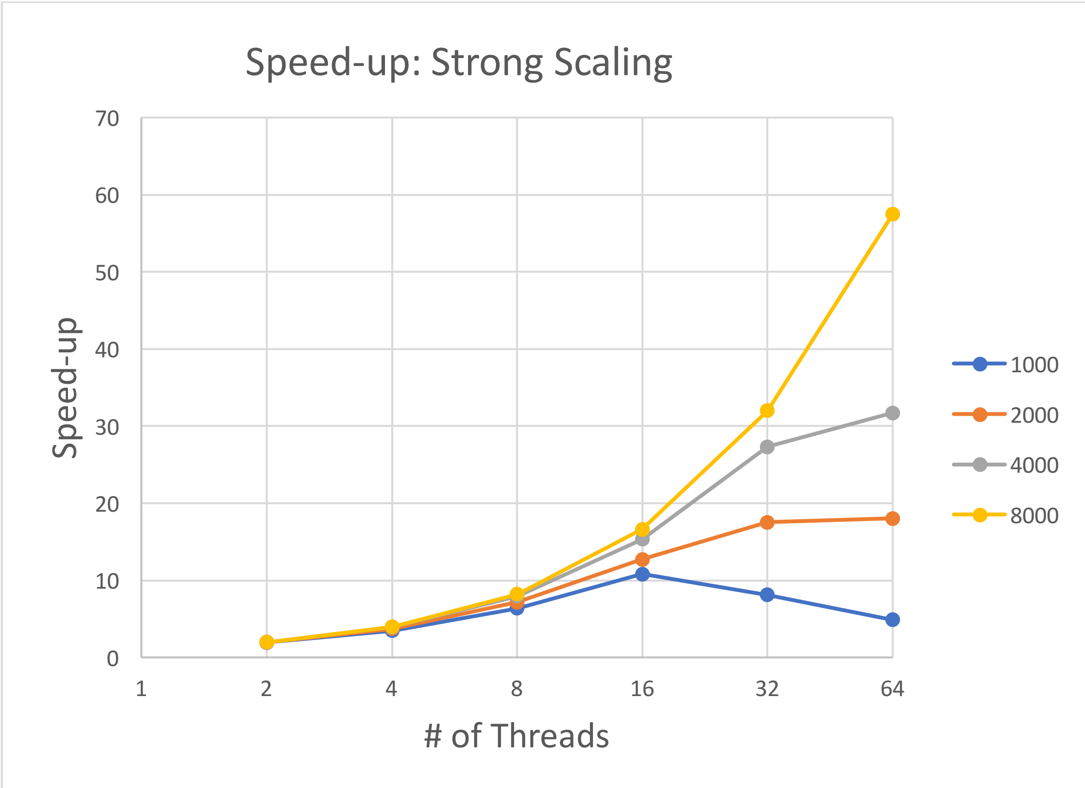
THREADS	RUNTIME	SPEEDUP	EFFICIENCY
2	5.189878	1.941749	0.970875
4	2.698743	3.734124	0.933531
8	1.404218	7.176550	0.897069
16	0.790354	12.75054	0.796909
32	0.573934	17.55854	0.548704
64	0.558174	18.05430	0.282098

Strong Scaling: Parallel Benchmark for 4000 Particles

THREADS	TIME	SPEEDUP	EFFICIENCY
2	21.766666	1.954975	0.977488
4	10.775733	3.948917	0.987248
8	5.333935	7.977840	0.997230
16	2.771328	15.35483	0.959677
32	1.557943	27.31376	0.855926
64	1.134183	31.712870	0.586233

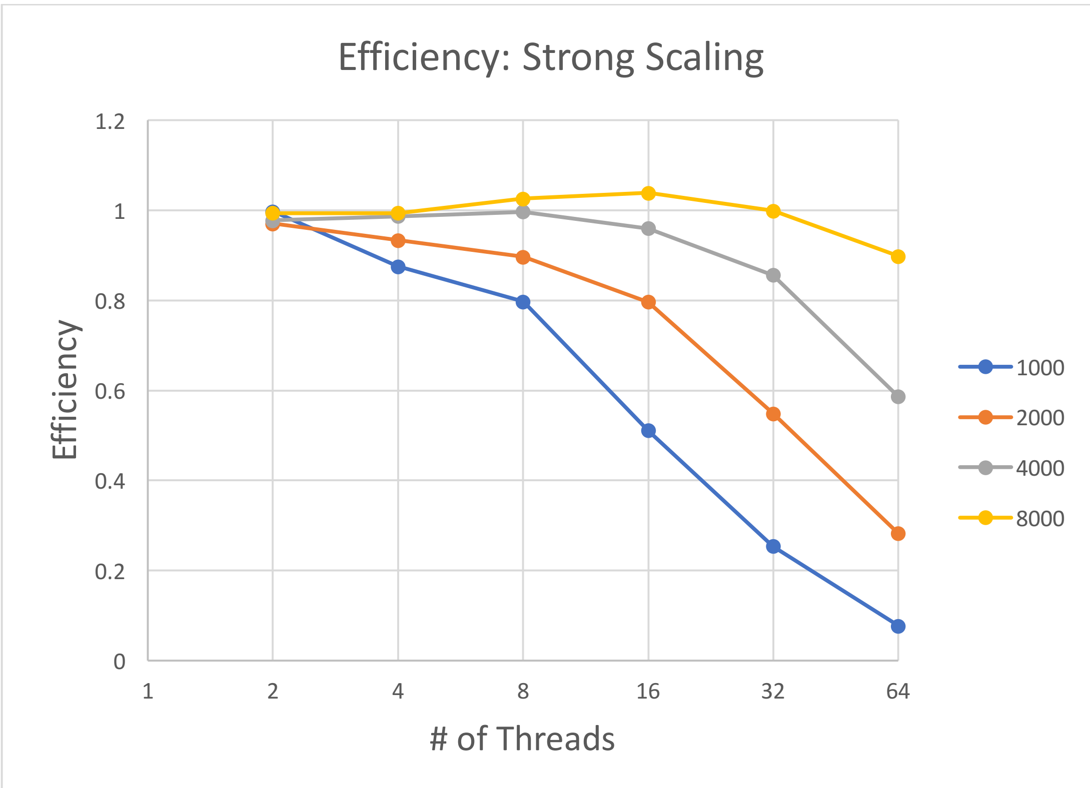
Strong Scaling: Parallel Benchmark for 8000 Particles

THREADS	TIME	SPEEDUP	EFFICIENCY
2	88.480447	1.987817	0.993909
4	44.233051	3.976280	0.994070
8	21.432792	8.206257	1.025821
16	10.580967	16.62258	1.038911
32	5.497110	31.99554	0.999406
64	3.058725	57.50206	0.898470



From the graph above, we see that for 1000 particles, there is an incline from 2 threads through 8 threads to about a ratio of 11 Speed-up just before 16 threads. Even though we might expect a further increase in performance by increasing the number of threads, a further increment in the number of threads, (to say 16 and above) for a number amount of work like this (# of particle = 1000), causes a decline, as seen in the figure above. This happens because, by increasing the number of threads, we reduce the amount of work per thread linearly, also every time we create a number of threads and cancel them, (in this case, 16, 32 and 64 respectively), we have to synchronize them back from sibling threads to a master thread and there is a scheduling overhead involved with the system associated with this task which tends to increase with an increase in the number of threads. Thus, a decline in the performance boost that would have been expected.
However, for a larger amount of work (E.G, in this case of 8000 particles), we see that the Speed-up increases relatively to an increase in the number of threads (at least, through 64 threads in the case). Note that when the number of threads is 2, the Speed-up is about 2, when the number of

threads is about 4, the Speed-up is about 4, and this holds true for 8, 16 and 32 thread values. Except 64, which is at about 58. This confirms the theory that doubling the number of threads increases performance by about a factor of $\log_2$.



From the above graph, we can see that for small amounts of work (in this case 1000 particles), we experience a Speed-up slightly above 100 percent using 2 threads. An increment in the number of particles (2000 through 8000 particles), shows a very slight decline in Speed-up especially for Series 2 and Series 3. This decline infers that we are not experiencing the expected Speed-up. Also, observe that an increment in the number of threads causes a further decline in the Speed-up, but for a large enough amount of work or particles (as in the case of 8000 particles), the worst performance seen, which is at when we use 64 threads is a Speed-up of about 90 percent. This signifies that we are using the full performance of the machine in parallel.

Weak Scalability

Weak Scaling: Parallel Benchmark

PARTICLES	THREADS	TIME	EFFICIENCY
200	1	0.129437	1
283	2	0.134213	0.964415
400	4	0.155836	0.830598
566	8	0.182114	0.710747
800	16	0.240038	0.539235
1131	32	0.348039	0.371904
1600	64	0.658022	0.196706

Weak Scaling: Parallel Benchmark

PARTICLES	THREADS	TIME	EFFICIENCY
1000	1	2.515760	1
1414	2	2.498109	1.007066
2000	4	2.690993	0.934881
2828	8	2.668286	0.942837
4000	16	2.754389	0.913364
5657	32	2.851426	0.882281
8000	64	3.101783	0.811069

Weak Scaling: Parallel Benchmark

PARTICLES	THREADS	TIME	EFFICIENCY
2000	1	10.133930	1
2828	2	9.904273	1.023188
4000	4	10.484766	0.966538
5657	8	10.375848	0.9766825
8000	16	10.565696	0.959135
11314	32	10.314394	0.982504
16000	64	10.994298	0.921744

Weak Scaling: Parallel Benchmark for 200 Particles

THREADS	RUNTIME	EFFICIENCY
1	0.132201	1
2	0.090290	1.464182
4	0.074501	1.774486
8	0.083074	1.591364
16	0.132957	0.9943140
32	0.242103	0.5460527
64	0.438493	0.3014894

Weak Scaling: Parallel Benchmark for 1000 Particles

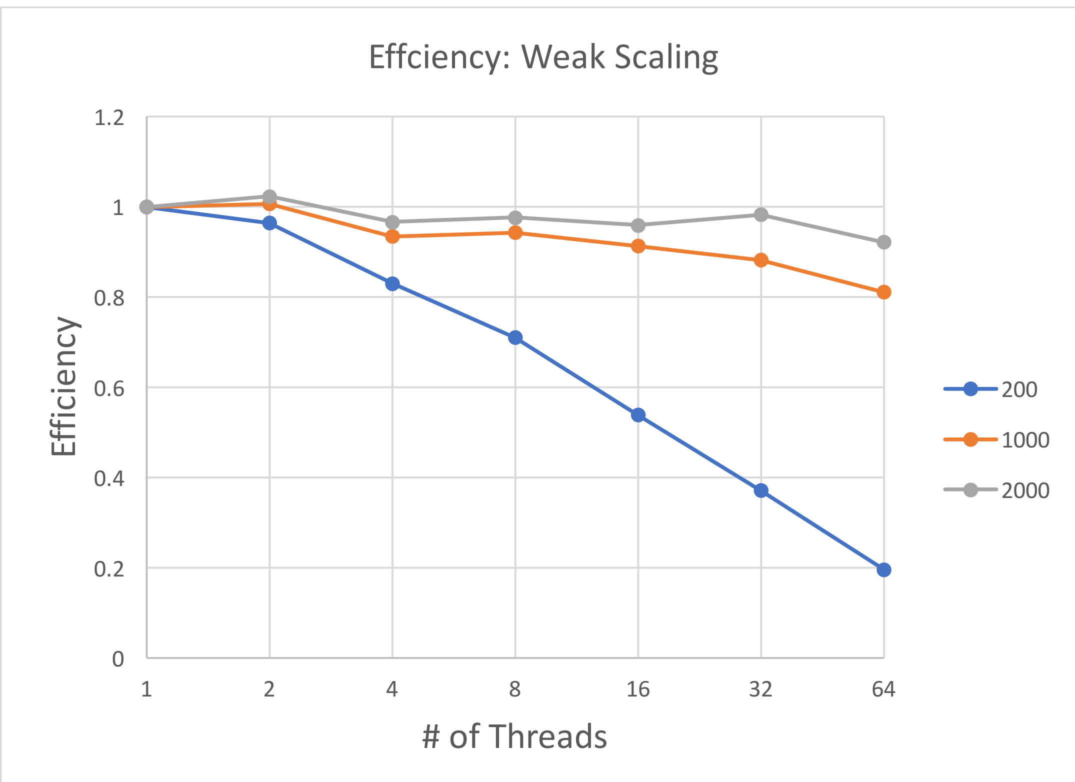
THREADS	TIME	EFFICIENCY
1	2.523516	1
2	1.339738	1.883589
4	0.695511	3.628290
8	0.424887	5.939264
16	0.281822	8.954290
32	0.328043	7.692638
64	0.482276	5.232514

Weak Scaling: Parallel Benchmark for 2000

THREADS	TIME	EFFICIENCY
1	10.164595	1
2	5.106774	1.990414
4	2.691575	3.77645
8	1.413210	7.192558
16	0.812912	12.50393
32	0.568803	17.87015
64	0.639607	15.89194

Weak Scaling: Parallel Benchmark

THREADS	PARTICLES = 200	PARTICLES = 1000	PARTICLES = 2000
1	1	1	1
2	1.464182	1.883589	1.990414
4	1.774486	3.62829	3.77645
8	1.591364	5.939264	7.192558
16	0.994314	8.95429	12.50393
32	0.5460527	7.692638	17.87015
64	0.3014894	5.232514	15.89194



The graph above is an illustration of the combined Weak Scaling Efficiency values of 200, 1000 and 2000 particles when parallelized with 1, 2, 4, 8, 16, 32 and 64 threads respectively.

As seen in the graph above, for a small amount of work (in this case, 200 particles), the Efficiency continually declines relatively to an increase in the number of threads. However, for larger amount of work like 1000 and 2000 particles, we start off with a performance boost that suddenly declines once the number of threads increases to 2. We see a slight boost in Efficiency with a further increment in the number of threads to 4 for both cases (i.e, 1000 and 2000 particles) and a slight decline with an increase in the number of threads to 16.

At 32 threads, for 2000 particles we see a similar boost in Efficiency that again declines starting at 32 threads. This denotes that an increase in the amount of work relatively to an increase in the number of threads, increases the Efficiency, as the Efficiency tends to improve because the amount of work increases proportionately with the number of threads. By so doing, we cushion the overhead, since weak scaling aims to keep the work load per thread constant.

Conclusion

1. After the following analysis, we can conclude that the n-body algorithm scales in parallel.
2. The scalability of the algorithm is not constant but dependent on how it is implemented using a specified number of threads and particles respectively as deemed fit.

Challenges Faced

1. Implementing OpenMP Parallel Pragmas
2. Calculating the Efficiency of Weak Scaling, specifically, how to increase the number of Particles in relation to increasing the number of threads.
   Resolution:
   I. I reviewed class resources and recommended textbook to better understand some missing underlining concepts.
   II. Emailed Dr. Stone to get help with calculating the Efficiency of Weak Scaling.