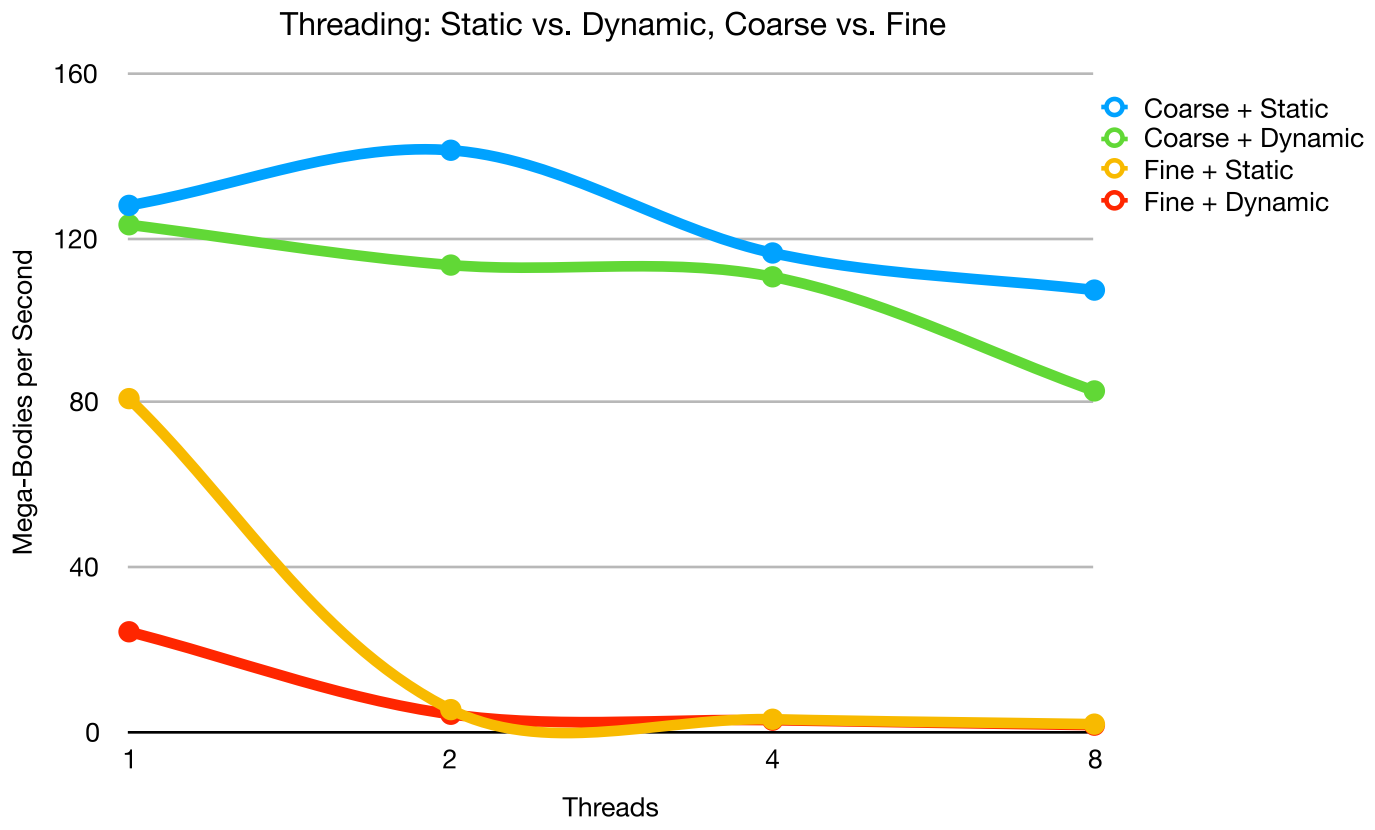
Aaron Berns
Project 2: OpenMP: N-body Problem

Machine
MacBook Pro, i5, 2 cores

Results

Mega-Bodies Compared per Second

	coarse + static	coarse + dynamic	fine + static	fine + dynamic
1 thread	127.86	123.15	80.89	24.22
2 threads	141.26	113.35	5.30	4.17
4 threads	116.29	110.50	2.96	2.73
8 threads	107.27	82.76	1.77	1.52



Discussion

One pattern that the data shows is that coarse-grained parallelism consistently outperformed fine-grained. After getting more information on the difference in the approaches, my understanding is that the coarse-grain approach allowed the parallel partitions to be created less often and to stick around longer than in the fine-grain approach, which would lead to less of a performance hit due to creation and destruction overhead. Another factor in the disparity in performance is that coarse-grain parallelism keeps the threads busier than fine-grained, allowing the work to be completed more quickly[1].

The total amount of work being done in the the parallelized portion of this program is approximately NUMBODIES^2 because of the nested for loop. This leads to 10000 iterations in the example code. The coarse-grain approach divides the work according to the number of outer loop iterations. The amount of work for each thread in this approach is roughly (NUMBODIES / NUMTHREADS) * NUMBODIES. With 4 threads, each thread would ideally be doing 2500 of the total iterations. The fine-grained approach divides the work of the inner loop among the threads NUMBODIES times. So a thread team is created and destroyed 100 times. That's a lot of overhead. The amount of work for each thread, of which there would be about 400 created in total, would be 25 iterations. The major takeaways of coarse-grained parallelism giving more work to each thread and limiting overhead by keeping the thread teams around longer are clear in this dataset.

Another pattern is that static scheduling outperforms dynamic scheduling in all scenarios. My guess is that since the work being down is inside of for loops, splitting the work during compilation is relatively easy to do and leads to each thread getting an equal number of loops without the overhead of monitoring thread activity and idleness and assigning loop iterations during runtime.

A third pattern is that having a greater number of threads, for the most part, leads to lower performance. The fine-grained approach's performance plummeted after the introduction of a second thread. Since this machine has two cores, the opposite would be expected. Only the coarse-static approach, which has the least runtime overhead, saw a slight performance increase with two threads, after which performance declined just like the other three approaches. Since the code uses an array, Bodies[], my first thought was that as soon as more than one thread is working, false sharing would become an issue as multiple threads could access array values on the same cache line. There seems to be no place in the code where Bodies[] is being written to though, so the decline in performance from saving and reloading changed cache lines wouldn't be an issue. If false sharing were occuring, there also wouldn't be an increase in performance with two threads in the coarse/static approach. My only guess is that as more threads are introduced, the performance hit from overhead increases.

1. https://www.umr-cnrm.fr/aladin/IMG/pdf/obrien_asm_copenhagen_2015-2.pdf