29 April 2018
Project #2
OSU CS475 Sp2018
Joshua L. Rogers
rogerjos@oregonstate.edu

N-body Problem — Coarse vs. Fine and Static vs. Dynamic

Because my personally-owned computer is long overdue for an upgrade and only has two cores offering a maximum of four threads, I chose to run the experiment on flip to take advantage of more threads, although I did not use all 24 available in my tests. I logged onto flips 1-3 and ran my code on each machine periodically over several days whenever I noticed that uptime reported a low load. For the next project I intend to automate this by adding an additional loop to my script that sleeps, checks uptime, and then executes the code and saves the results if the uptime is below threshold, but for this one I did it the more difficult way for no good reason. I considered using rabbit, but because the results I saw on flip didn't indicate that adding cores would be of much use I opted not to burn the power and CPU time on this project, leaving the machine for those whose projects can more productively use rabbit. I'm really hoping that we're assigned something that takes advantage of all those threads.

In order to minimize variance due to processor load I wrapped the code in a for loop that executed an arbitrary number of iterations as defined by the macro ITERATIONS, then returned both average and peak performance. This wasn't required by the project specification, but it seemed prudent to ensure reliable results.

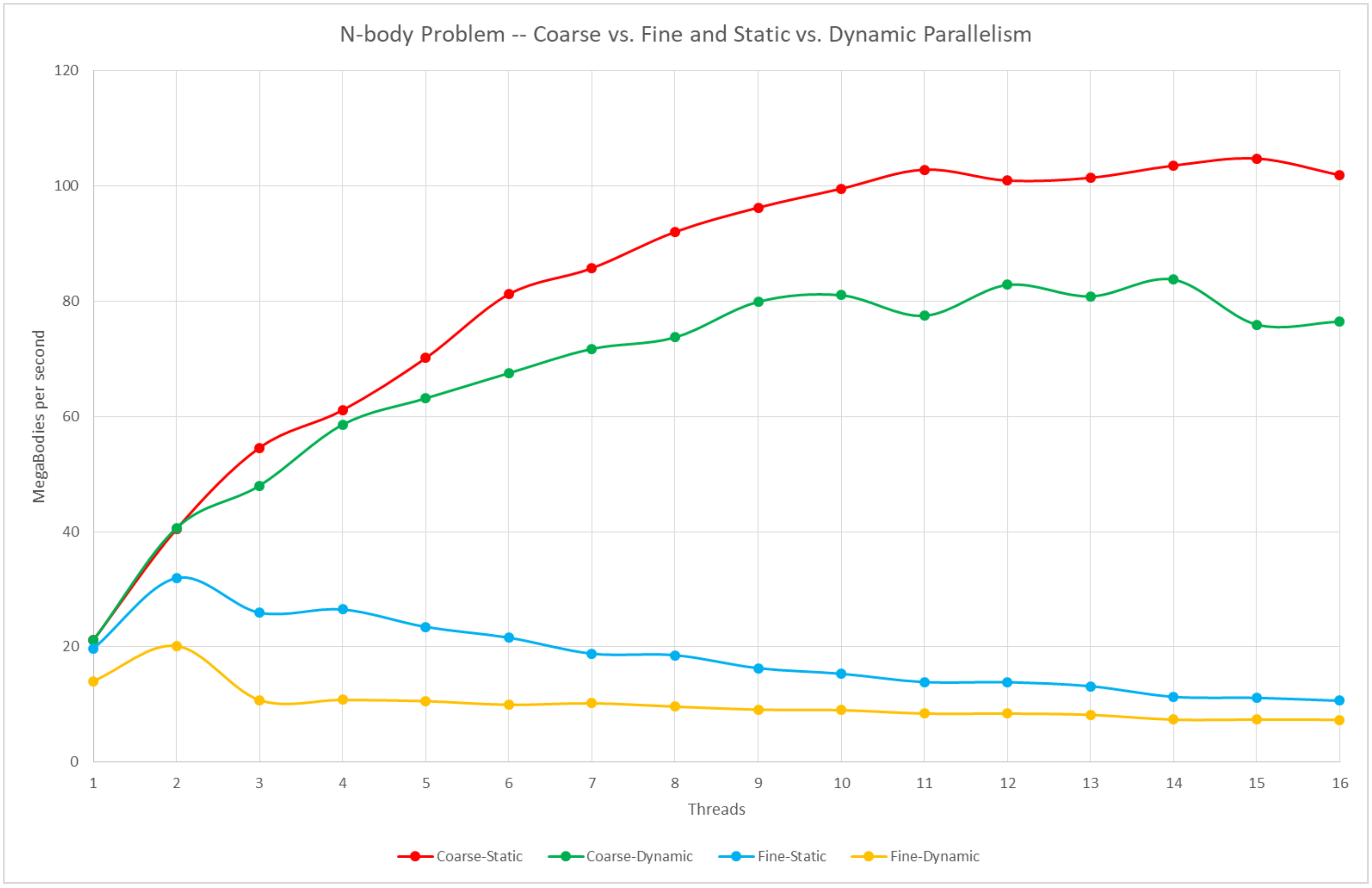
For simplicity's sake I integrated all four cases (Coarse/Static, Coarse/Dynamic, Fine/Static, Fine/Dynamic) into a single file proj2.c and used #if directives predicated on the value of the GRAIN macro to determine whether to run Coarse- or Fine-grained parallelism. Similarly, the value of macro OMP_SCHED in [1..4] is used to set the macro SCHEDULE to one of static, dynamic, guided, or auto respectively. Only the first two were used, but I found the others in the OpenMP specification and thought it would be interesting to try them. The macro NUMTHREADS determines the number of threads in the thread team.

To automate running the code I wrote a short bash script that loops from 1 to 16 threads, and for each thread value loops through all required parallelism/schedule cases using the arbitrary (but presumably reasonable) iteration count of 32 for each thread/case combination. I did experiment with using 128 iterations but didn't see a significant difference. This is a case where having something more powerful, e.g., an i7-8700K, at home would have been beneficial since I didn't want to leave my code running for hours executing huge numbers of iterations. The script set all required macros via the gcc -D flag. Output was produced in comma-separated value format and sent to a file, then imported into Excel to be converted to tables and graphs.

I ran the script several dozen times over several days in order to ensure a "clean" result that wasn't obviously impacted by other students using the machine. My results in tabular form with performance values in MegaBodies compared per second are as follows:

Coarse-Static Mbps	21.17008	40.43527	54.56848	61.09169	70.12817	81.21167	85.73724	92.02203	96.24319	99.53333	102.8467	100.9418	101.4285	103.551	104.7756	101.8992
Coarse-Dynamic Mbps	21.11228	40.62121	47.98364	58.58231	63.14829	67.50824	71.72618	73.7255	79.89375	81.03142	77.47181	82.82452	80.84412	83.73648	75.94696	76.46867
Fine-Static Mbps	19.68757	31.94151	25.9323	26.49461	23.46521	21.5975	18.81991	18.54997	16.30103	15.33595	13.87984	13.86666	13.16786	11.32645	11.14975	10.63731
Fine-Dynamic Mbps	14.06864	20.15463	10.76139	10.7992	10.57289	9.948522	10.19539	9.623547	9.094171	9.011928	8.413764	8.415884	8.200053	7.361808	7.375205	7.34417
Threads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**

Represented as a graph with number of threads on the X axis and MegaBodies compared per second the data appears as follows:



The differing approaches to parallelism and scheduling display clear patterns:

- **Coarse parallelism with static scheduling** is the highest performance case for all thread counts greater than two, and is faster than and sort of fine parallelism at any thread count. The performance seems to scale linearly with thread count until about eleven threads, at which point the performance levels out to just over 100 MegaBodies per second.

- **Coarse parallelism with dynamic scheduling** is the second highest performance case for all thread counts greater than two, and nearly indistinguishable from coarse parallelism with static scheduling at one and two threads. As with coarse parallelism with static scheduling, coarse parallelism with dynamic scheduling scales linearly with thread count, albeit more slowly, then levels out, albeit sooner at about nine threads. Then the performance levels out to around 80 MegaBodies per second.

- **Fine parallelism with static scheduling** is the second lowest performing case and exhibits surprising behavior, first speeding up about 50% between one thread and two threads, then immediately losing performance linearly with an increase in thread count. The case becomes less efficient when parallelized than when run non-parallelized around seven threads, and continues to become less efficient until finally performance at sixteen threads is slightly better than half of single-threaded performance.

- **Fine parallelism with dynamic scheduling** is the case with the worst performance at all thread counts and exhibits a pattern similar to its static equivalent, first peaking at two threads, then slowly declining until it has dropped to slightly better than 50% of its original speed at sixteen threads.

These results are not exactly what I expected, but upon reflection the data does make sense. First, and most simply, there is a clear cost to using dynamic vs. static scheduling. With static scheduling each loop iteration is pre-assigned to a given thread, and then the threads each work their way through their assigned iterations. In dynamic scheduling the iterations are not pre-assigned, and when a thread becomes idle it is assigned the next pending iteration. The dynamic approach is very useful in cases where the set of tasks is composed of elements with differing time costs in order to avoid the waste incurred by idle threads, but when the tasks required by the parallelized code show very little variance in evaluation time there is no tangible benefit to dynamically assigning tasks to threads because there will be no significant difference from static scheduling. In fact, because assigning tasks under dynamic scheduling is not free, in cases such as the project code where each task requires nearly the same amount of time to complete dynamic scheduling must necessarily exhibit lower performance than static scheduling simply due to this added cost. This is borne out by the difference in performance between the two approaches to coarse parallelism in the graph above, where the difference in performance increases as the thread count increases until each case's performance levels out as described.

However, this increasing difference is not evident in the cases of fine parallelism, and for good reason. Recall that OpenMP uses the fork-join model of parallel execution. That is, when a parallelized region of code is encountered the master thread creates a thread team of parallel threads. The thread team handles the parallelized region, then are joined back into the master thread. This model results in massively increased performance under ideal conditions, but creating a thread team is computationally expensive. Note that the cases using coarse parallelism invariably exhibited higher performance when multiple threads were used, whereas the performance of the cases using fine parallelism only increased when two threads were used. Now examine the structure of the relevant code to the right, with only directly relevant code represented for convenience:

```
for( int t = 0; t < NUMSTEPS; t++ )
{
    #pragma omp parallel for <options>
    for( int i = 0; i < NUMBODIES; i++ )
    {
        Body *bi = &Bodies[i];
        #pragma omp parallel for <options>
        for( int j = 0; j < NUMBODIES; j++ )
        {
            Body *bj = &Bodies[j];
        }
    }
}
```

The two OpenMP directives shown herein are mutually exclusive, being selected at compile time as described on the first page. The red directive represents coarse parallelism, and the blue directive represents fine parallelism. The macros NUMSTEPS and NUMBODIES have been set to 200 and 100 respectively. Under coarse parallelism a thread team is forked and joined NUMSTEPS, or 200 times over the course of execution. Fine parallelism, however, forks and joins a thread team NUMSTEPS * NUMBODIES, or 20,000 times. This incurs a significant cost under fine parallelism. Because the amount of work is relatively small per iteration, this additional cost is noticeable and is reflected in the results above. The two cases of fine parallelism do not diverge as thread count increases as do the cases of coarse parallelism because the additional cost per thread allocated across 20,000 team fork/joins is greater than the cost of dynamic scheduling, so the effect of scheduling differences across cases is most significant at lower thread counts, and the results tend to converge as the thread count increases and cost of parallelism becomes increasingly dominant.

Fine parallelism also incurs a performance penalty because the value of j is used as an index in the Bodies array, so whereas under coarse parallelism each thread steps through all values of j sequentially under fine parallelism the iterations of the j loop are distributed between threads, each thread access indices of Bodies non-sequentially which generates cache misses and results in reduces performance when compared with coarse parallelism.