Chido Nguyen

931506965

nguychid@oregonstate.edu

Program 2: OpenMP N-Body Problem

Coarse vs Fine and Static vs Dynamic

1) Program complication, execution and data gathering were done on OSU's engineering server; flip2.engr.oregonstate.edu. Server loads were around 1.0-1.5, not many people were on/using it early in the week and morning. Server loads went up around 1.23~ when I used it.

2). Tables:

MABC/s = Mega_Astro_Body_Calculations per second

Table 1: STATIC scheduling, and COARSE approach to parallelizing program 2 at various threads over 4 different runs.

Static-Coarse in MABC/s	Run 1	Run 2	Run 3	Run 4	Average
1-Thread	11.5733	11.103	11.1023	11.1051	11.22093
2-Thread	21.1729	21.6487	21.7979	21.7336	21.58828
4-Thread	41.4323	40.8968	41.3816	41.38	41.27268
8-Thread	73.6755	73.9735	74.5103	73.7419	73.9753
12-Thread	70.3995	71.6968	71.3152	69.8611	70.81815

Table 2: DYNAMIC scheduling, and COARSE approach to parallelizing program 2 at various threads over 4 different runs.

Dynamic-Coarse in MABC/s	Run 1	Run 2	Run 3	Run 4	Average
1-Thread	11.0918	11.0458	11.0611	11.066	11.06618
2-Thread	21.3973	25.0512	21.3875	21.4457	22.32043
4-Thread	45.7849	44.7968	44.2275	45.3234	45.03315
8-Thread	77.2641	78.1657	77.3788	78.8731	77.92043
12-Thread	92.2322	93.3166	93.9933	92.8226	93.09118

Table 3: STATIC scheduling, and FINE approach to parallelizing program 2 at various threads over 4 different runs.

Static-Fine in MABC/s	Run 1	Run 2	Run 3	Run 4	Average
1-Thread	10.2878	10.2809	11.0827	11.1838	10.7088
2-Thread	18.4416	19.2768	18.023	19.7798	18.8803
4-Thread	21.0992	22.2762	22.2609	23.4266	22.26573
8-Thread	17.0735	17.3448	16.7996	16.6882	16.97653
12-Thread	11.9362	12.3111	12.3795	12.2686	12.22385

Table 4: DYNAMIC scheduling, and FINE approach to parallelizing program 2 at various threads over 4 different runs.

Dynamic-Fine in MABC/s	Run 1	Run 2	Run 3	Run 4	Average
1-Thread	7.40168	7.81197	7.98333	11.8581	8.76377
2-Thread	11.2181	10.9106	11.079	10.7802	10.99698
4-Thread	11.1563	11.3898	11.2101	10.2247	10.99523
8-Thread	9.63345	9.62257	9.50037	9.59017	9.58664
12-Thread	8.40588	8.35022	8.34414	8.36211	8.365588

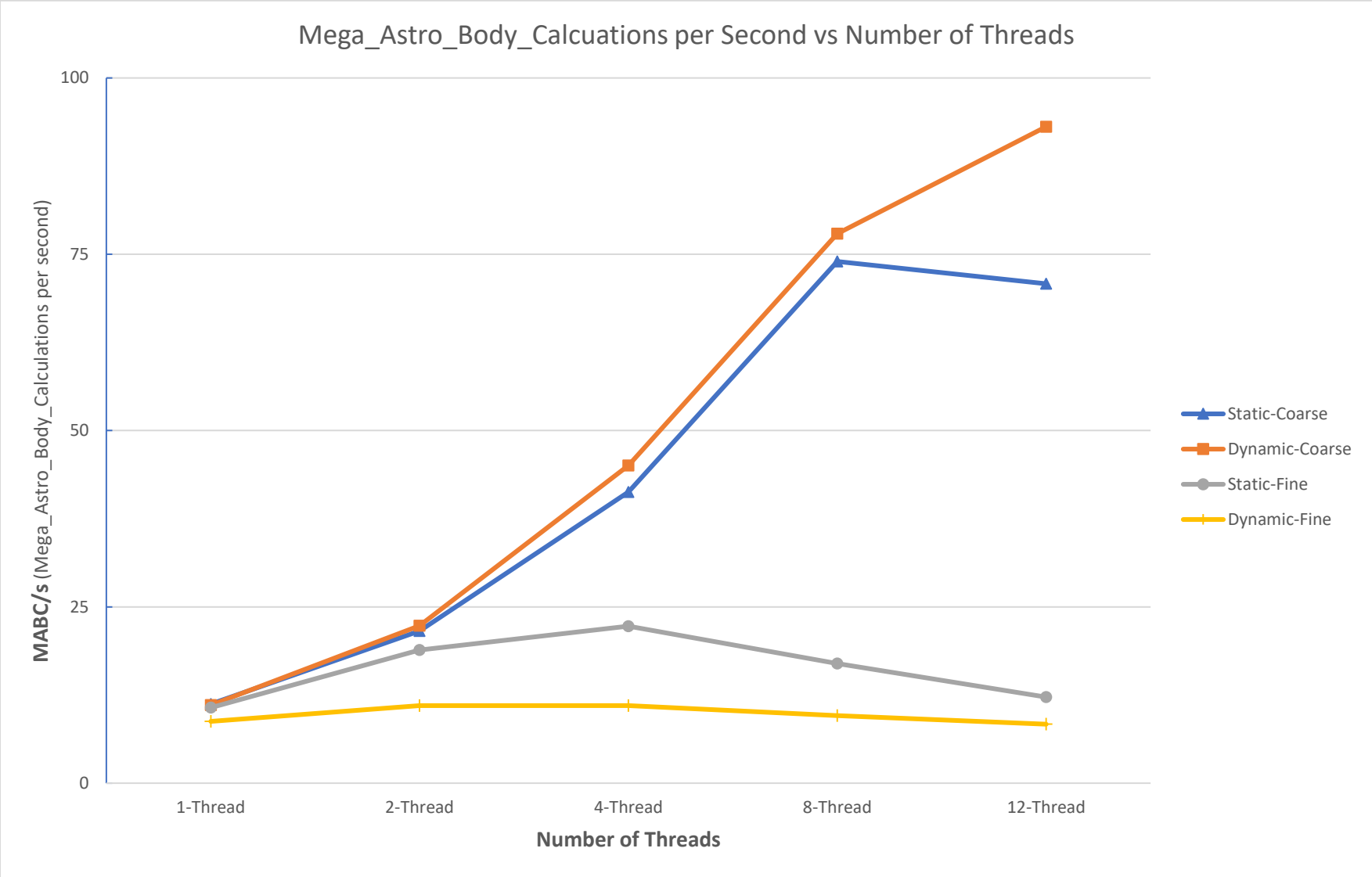
Table 5: Speed-Up values of the different approaches when threads are increased. (Speedup = T1/Tn)

Speed-Up	1-Thread	2-Thread	4-Thread	8-Thread	12-Thread
Static-Coarse	Reference	1.9239	3.6782	6.5926	6.3113
Dynamic-Coarse	Reference	2.0170	4.0694	7.0413	8.4122
Static-Fine	Reference	1.7631	2.0792	1.5853	1.1415
Dynamic-Fine	Reference	1.2548	1.2546	1.0939	0.9546

Table 6: Parallel fraction values based on the different speed-up values from table 5 with an added average.

Parallel Fraction	1-Thread	2-Thread	4-Thread	8-Thread	12-Thread	Average
Static-Coarse	Reference	0.9605	0.9708	0.9695	0.9181	0.9547
Dynamic-Coarse	Reference	1.0084	1.0057	0.9805	0.9612	0.9890
Static-Fine	Reference	0.8656	0.6921	0.4219	0.1352	0.5287
Dynamic-Fine	Reference	0.4061	0.2706	0.0981	-0.0519	0.1807

3) Graph 1: Visual representation of MABC/s as we increase the number of threads. Each line represents a different approach Static-Coarse, Static-Fine, Dynamic-Coarse, Dynamic-Fine.



4) Speed Trends Observations all possible why/discrepancy will be addressed in section (5).

Static-Coarse as shown in table 1:

The speed values were what I expected. Across four runs the performance remained quite stable throughout. As we increase the number of threads we also see that our performance value gets better which is expected as we increase the amount of "workers" outputting work. Except when we reach 12-threads the speed seems to taper off a little bit compared to 8-threads.

Dynamic-Coarse as shown in table 2:

Similar to before we see stable speeds throughout the 4 runs to test for data reliability. And as we increase the number of threads we see that speed performance also gets better. The biggest difference here is that our Dynamic approach at 12 thread performs better than our Static approach in a Coarse parallelism setup.

Static-Fine as shown in table 3:

Performance speed for this approach is greatly lower than both Coarse approach. Adding threads do no increase the performance value dramatically as seen in the 2 Coarse approaches. Though speed does increase up to 4 threads before tapering downwards again at 8 and 12 threads.

Dynamic-Fine as shown in table 4:

Data for this approach was unexpected from an instinct perspective as what some of the other tables were showing. Will save speculation and expectation explanation for next section. As for speed we see only 1 data anomaly at the fourth run on 1-thread jumping from ~8 to ~11. Aside from that all data stays stable through four runs. We see that increasing 1->2 threads and 1->4 threads show some improvement in performance but barely an upgrade. Increased in ~3-4 MABC/s in performance. As for 2->4 thread there is a negligible increase/decrease in performance variation in the two. After 4 threads 8 and 12 threads decrease in performance to barely do better than 1-thread's performance value; roughly ~1-2 MABC/s increase at 8 and 12 vs 1-thread.

5) Why?

Static-Coarse vs Dynamic-Coarse

First off, I believe that the drop off in performance for Static-Coarse from 8 threads to 12 threads is observed is probably due to a "not big enough data sample size" to take advantage all 12 threads. In return the overhead for running 12 threads has diminishing effect on the overall speed of the program as we see in table 1.

The switch to Dynamic scheduling in case 2 / table 2 actually benefits the usage of twelve threads more than static. My educated guess is this is because of how dynamic scheduling works. Static would divide all of the workload amongst the threads equally i.e. if I had 12 tasks to do with 12 threads each thread gets a task (or in this case 12 iterations of a for loop). Now that we invoked dynamic scheduling over static we can picture it as being micro-managed by a manager at work. Every worker stands around (our thread pool) the manager assigns tasks to the workers depending on the workers "busy-ness" rather than each worker having a set list of things to do.

This is probably why we see a bump in speed for Dynamic-Coarse from 8-12 threads as oppose to Static-Coarse from 8-12 threads. Being static most likely had multiple threads finishing its "to-do" assignment first and idle-ing while the other threads finished led to a slower performance value. Now when we change it to dynamic, these idle-ing threads are now forced by our "manager" to pick-up other threads unfinished to-do assignments. This change scheduling is probably why we see a bump in performance as oppose to a diminishing performance value at 12 threads.

Fine vs Coarse

The change from Coarse to a Fine parallelism approach showed a dramatic reduction in performance speeds.

The two most prominent reasons I can think of for this is 1) False sharing and 2) cache-misses.

With an increase in our data sample size (100 for loop iterations parallelized vs 100 for loops on top of another 100 for loop iterations that is parallelized) we are more likely to encounter false sharing. False sharing negatively impacts the program's performance due to multiple threads attempting to access (read/write) from the same cache line (thread has to reload cache line if another thread has edited something/anything in the cache line causing it to become invalid). To continue with the idea of 100 to 10000 data sample size scaling, with each data sample we'll have to do some work on it. This translate to 100 chances to false share vs 10,000 chances to false share. That in my opinion is one reason why we see Fine approach has such bad performance values.

Now to look at cache-misses. When we were working on 100 tasks there were less cache misses. We bring in a cache-line and when we get to the inner for loop there was a chance that the inner for loop would use some values in this cache-line if not we'll bring in a new cache line. Also, there is some temporal cohesion, the last address is saved somewhere to be accessed again if needed. Specifically I am referring to the first access of Bodies[i] (Body *bi = &Bodies[i];) by the outer "I" for loop and then , by the "J" for loop (Body *bj = &Bodies[j];). Say we have a cache miss when we get to the inner j-for loop, temporal cohesion would save our recent Bodies[i] address somewhere "we just used it maybe we'll need it again soon" and then bring in a new cache line for *bj. Long story short the smaller sample size has less chances for cache-misses due to better spatial and temporal cohesion.

When we divide the 100 big tasks into 100 smaller tasks (our 10,000-data sample) we increase our possibility for cache misses. The inner for loop works on 1 value of Bodies[j] on each thread and is managed via dynamic scheduling. The next "task" on the thread could be some j value that puts Bodies[j] outside of the 64-byte cache line that was brought in and now we have a cache miss. There is also no guarantee that the following Bodies[j] would use any of the previous 2 cache line either, so now not only are we cache-missing we also lack temporal cohesion.

Dynamic Fine vs Static Fine:

The last section slightly addressed this comparison but not explicitly. The reason why I think we see static doing better than dynamic here is because of how chaotic dynamic becomes. Changing the inner loop to dynamic scheduling causes a drop in temporal and spatial cohesion, leading to more cache misses. I mentioned in the last section about the potential of Bodies[j] jumping around one job could be working on j=1, the next j = 99 (different cache line), then after that possibly j=33. We see how we have to reload our cache line AND it throws temporal cohesion out the door. The OS might save address to j = 1 thinking we'd need it but now we're at 33.

There is also an increase in false sharing instances (100 data sample vs 100 (100) data samples for fine). Since we're using Bodies as the same "resource" at 100 data sample we have 100 chances of accessing the same memory region causing false sharing, and 10,000 chances with a fine-approach. This led and shows on the graph how much our performance diminished as we moved from doing large problems to "fine-grained" problems.