Linge Ge - CS475
Project 2 - N-body Problem -- Coarse vs Fine and Static vs Dynamic

Overview:

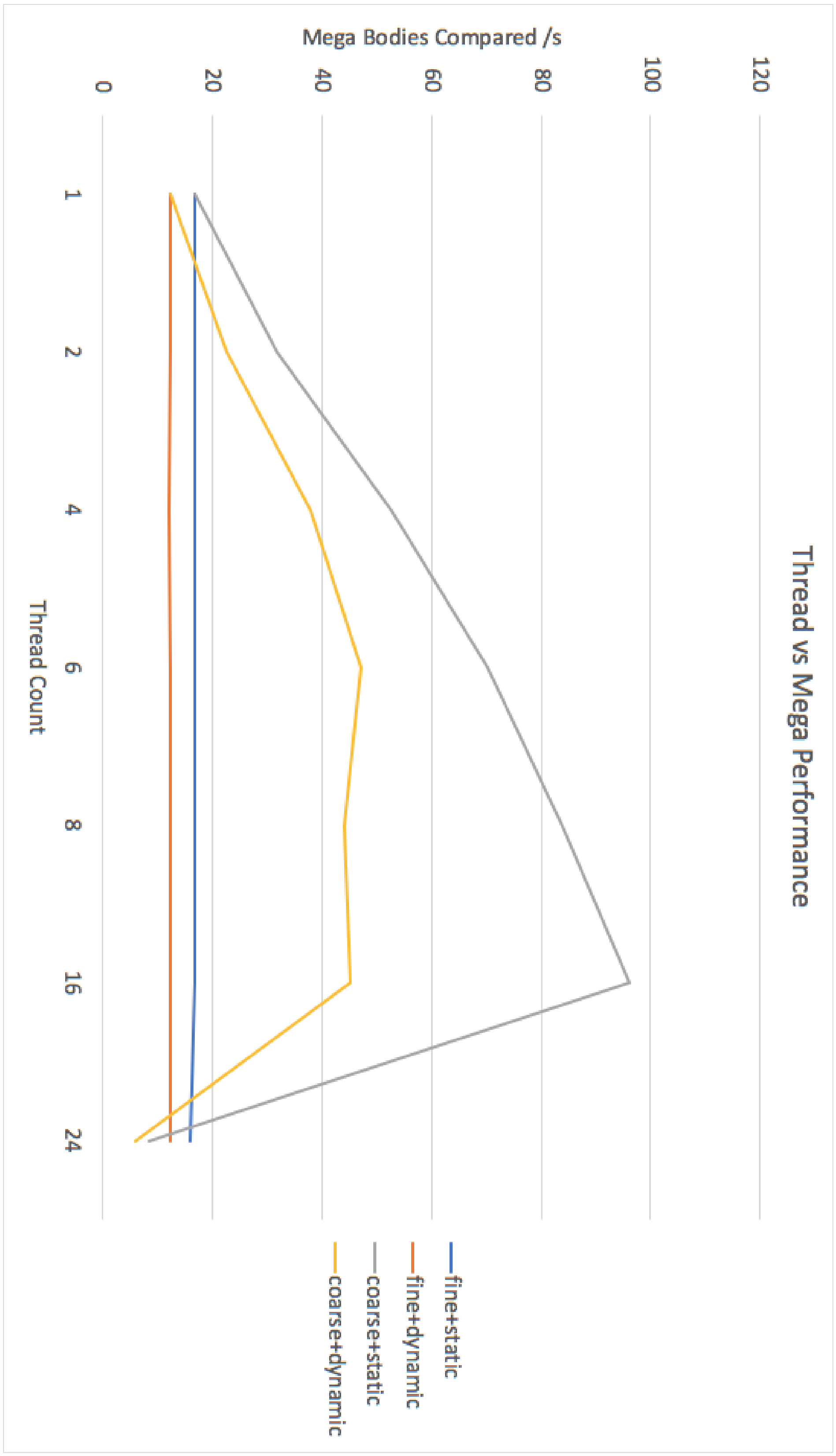
This program simulates the movement of 100 planets as they are affecting each other through their gravitational pulls. For each planet, their location on a 3D euclidean space is calculated by iterating through each other planet and considering the gravitational pull from each of those planets. This is done for each planet, and done 200 times to simulate a short view of how the planets move within those 200 “frames”. Through this program, we put in parallelism and take advantage of multi-threading at different loops with different thread allocation algorithms, and see the performance boost for each type.

1. The code ran on:
    - Linux flip2.engr.oregonstate.edu 3.10.0-693.11.1.el7.x86_64 #1 SMP Mon Dec 4 23:52:40 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
    - 96GiB
    - 12 (hyperthreading x2) CPUs @ Intel(R) Xeon(R) CPU X5650 @ 2.67GHz

2.

Thread Count	Type	Mega-Bodies Compared/s
1	fine+static	16.873859
1	fine+dynamic	12.304465
1	coarse+static	16.809198
1	coarse+dynamic	12.243651
2	fine+static	16.868586
2	fine+dynamic	12.304173
2	coarse+static	31.799816
2	coarse+dynamic	22.665535
4	fine+static	16.836634
4	fine+dynamic	12.247647
4	coarse+static	52.679642
4	coarse+dynamic	38.010838

6	fine+static	16.793198
6	fine+dynamic	12.285108
6	coarse+static	70.261856
6	coarse+dynamic	47.17939
8	fine+static	16.878731
8	fine+dynamic	12.295985
8	coarse+static	83.881676
8	coarse+dynamic	44.126743
16	fine+static	16.878122
16	fine+dynamic	12.434954
16	coarse+static	96.013512
16	coarse+dynamic	45.257355
24	fine+static	16.060751
24	fine+dynamic	12.284969
24	coarse+static	8.535907
24	coarse+dynamic	5.941777


Thread vs Mega Performance
Mega Bodies Compared /s
0
20
40
60
80
100
120
1
2
4
6
8
16
24
Thread Count
fine+static
fine+dynamic
coarse+static
coarse+dynamic

3.

4.

- No performance improvement at all when using Fine+static or Fine+dynamic on any number of threads
- Fine+static is slightly more performant than Fine+dynamic
- Coarse+static is more performant than Coarse+dynamic, but both yield good multiplicative speedups when multi-threading
- Optimal performance is seen in the range of 6-16 threads, but performance is worse than even single threading when using 24 threads

5.

- Why performance fails at 24 threads may be the easiest to explain. The server that the program was ran on has 12 physical cores, but counts 24 threads due to hyperthreading capability. Hyperthreading is best when a thread would be otherwise idle waiting for a resource (I/O, etc.). However, since this program is heavily numerically computational, hyperthreading has no benefit because two hyperthreads are sharing same computational resources, and it actually hinders progress because of poor temporal coherence: the threads are constantly flushing out the shared cache lines.
- Fine-grained is not performant because the program has poor temporal coherence in the inner loop, since the inner loop iterates through a set of data, and never reuses old data for computation. Hence it’s constantly flushing the cache, plus it’s bringing in large data into the cache since the array is global, and resulting in more cache misses.
- Coarse-grained performs well because it takes advantage of the cache with much better temporal coherence. On each outer iteration, it reuses the data in Bodies[i] many times within the inner loop, hence taking advantage of the same data in the cache and generating lots of cache hits.
- Static allocation is faster than dynamic allocation. This makes sense, as the latter has much more overhead: increased complexity in breaking down the array in memory in smaller chunks and diverting them to different threads as needed. This may be useful for projects where the threads are working on different aspects of the program (ex: SimCity, etc.), and my have very different amounts of work. However in this program, it is very clear that each thread does the same amount of work (taking an element from the array and doing the same computation), hence there’s not much need for dynamic allocation. Static allocation can thus be used, and would be faster, because there’s less complexity involved (take total memory, divide by number of threads, done), and the splitting is done before execution.