Comparison of OPENMP Parallelization Techniques

Vladyslav Bondarenko

I. Optimizing Step Function

	Experiment Name						
N	Core CPU	Mult vs Pow	Parallel Inner Critical	Parallel Inner Atomic	Parallel Outer	Schedule Static	Schedule Dynamic
64	1.3	0.5	1.9	0.9	0.1	0.1	0.1
128	5.1	1.8	4.4	2.9	0.4	0.4	0.4
256	19.8	7.3	15.1	9.6	1.7	1.3	1
512	78.5	29.5	56	33.6	5.4	5.1	4.8
1024	311.4	116.7	223.6	113.8	18.9	18.2	16.2
2048	1249.4	467	917.5	514.1	68.9	67.7	64.6
4096	4980.6	1860.6	3538.1	1632.2	278.2	272	263.3
8192	16778.7	7420.8	14178.7	7056.8	1106	1059.3	1014.8

Fig. 1: Full Results for Step Optimization in ms/I

The code was benchmarked by obtaining the run-time per iteration in milliseconds in debug mode for different values of N (number of bodies) with D (activity dimensions) fixed to 4. Each value was obtained over an appropriate number of iterations for a few repetitions and averaged to get an accurate estimate of the actual time complexity. The full table of results for the optimization of step function can be found in Fig. 1 with appropriate descriptions for each experiment outlined below.

A. Serial CPU Implementation

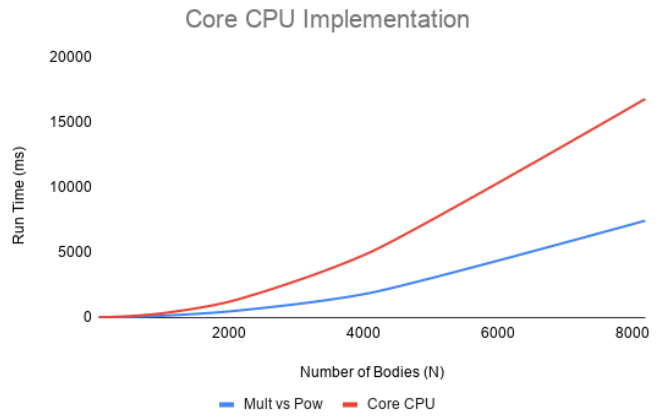


Fig. 2: Graph of initial implementation and minor optimization

Initial CPU implementation results are graphed in Fig. 2 represented by a red line. Running the performance profiler immediately identified a particular line of code as a critical contributor to the run-time with over 40% of the whole run. The initial code is displayed at the top of Fig. 3 with optimized version at the bottom. It was found that *pow* function is the

```
sum_x += (b_j->m * diff_x) / (float)pow(pow(diff_x, 2) + pow(SOFTENING, 2), 1.5f);
sum_x += (b_j->m * diff_x) / (float)pow(pow(diff_y, 2) + pow(SOFTENING, 2), 1.5f);
```

```
sum_x += (b_j->m * diff_x) / (float)pow(diff_x*diff_x + SOFTENING2, 1.5f);
sum_x += (b_j->m * diff_x) / (float)pow(diff_y*diff_y + SOFTENING2, 1.5f);
```

Fig. 3: Replacing power function with multiplication in critical section of the code. Top - initial implementation; Bottom - improved version.

major contributor to the large run-time and so it was replaced by simple multiplication where appropriate and by a new constant for the softening factor. This alone had a great impact on the performance resulting in more than half the initial run-time. Further improvement could be obtained by removing the last *pow* function with an appropriate approximation. The optimized version of the serial implementation is used as a baseline for all further experiments.

B. Parallelazation over Inner/Outer Loop

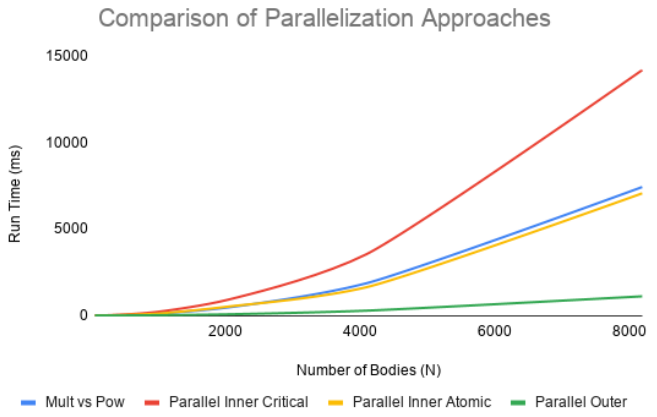


Fig. 4: Comparing different approaches to parallelization of the step function loops

At first, an attempt was made to parallelize the inner loop of the step function. Since *sum_x* and *sum_y* variables are shared and can be accessed by multiple threads at the same time, a race condition is likely to occur. The first solution presented to this problem is shown in Fig. 5 using a critical section with results denoted by the red line in Fig. 6. As can be observed the run-time when using critical is even worse than with serial CPU implementation. This is likely caused by a large amount of work that has to be done in the critical

```
#pragma omp parallel for default(none) private(b_j, diff_x, diff_y) shared(b_i, sum_x, sum_y) if(mode == OPENMP)
    for (j = 0; j < N; j++) {
        if (i != j) {
            b_j = &bodies[j];

            diff_x = b_j->x - b_i->x;
            diff_y = b_j->y - b_i->y;
#pragma omp critical
            {
                sum_x += (b_j->m * diff_x) / (float)pow(diff_x*diff_x + SOFTENING2, 1.5f);
                sum_y += (b_j->m * diff_y) / (float)pow(diff_y*diff_y + SOFTENING2, 1.5f);
            }
            temp_x = (b_j->m * diff_x) / (float)pow(diff_x*diff_x + SOFTENING2, 1.5f);
            temp_y = (b_j->m * diff_y) / (float)pow(diff_y*diff_y + SOFTENING2, 1.5f);

#pragma omp atomic
            sum_x += temp_x;
#pragma omp atomic
            sum_y += temp_y;
```

Fig. 5: Using critical(top)/atomic(bottom) section to avoid raise condition when parallelalizing inner loop of the step function.

section with additional time complexity added with thread initialization done for each repetition of the outer loop.

A solution to the problem of high computation inside a critical section is to store the calculation in the private temporary variable and only perform the update of the shared variable inside the section. In combination with the atomic section, this produces a satisfactory result denoted by the yellow line in Fig. 6. Although a great improvement over the previous attempt, it produced only a small boost in run-time in comparison to the CPU implementation. The main problem of spinning up threads every iteration of the outer loop still remains.

Finally, parallelizing over the outer loop produced by far the best results displayed by the green line in Fig. 6. It eliminates the need to set up threads as often, as well as gets rid of any assignments to shared variables. Achieving an overall 6-7 fold speed up with a greater improvement for larger values of N.

C. Comparison of Scheduling Approaches

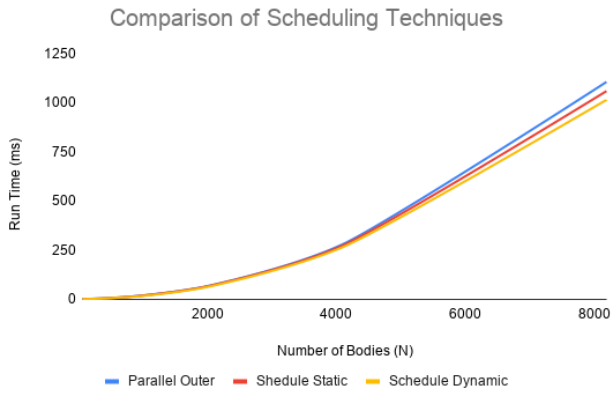


Fig. 6: Static vs Dynamic scheduling

Choosing appropriate scheduling resulted in a further speed up by a small margin. Although the workload is balanced between threads, it was found that dynamic scheduling was best. Since there are no common memory access patterns between threads, the chunk-size was set to default 1.

II. Optimizing Activity Calculation

The optimized CPU implementation is used as a baseline and experiments are run for the varying values of D with N fixed to 256. There are three loops in the calculation, and each one has been parallelized with full results shown in Fig. 7 and plotted in Fig. 9.

	Experiment Name			
D	Core CPU	Parallel Inner Loop	Parallel Middle Loop	Parallel Outer Loop
4	1	1.5	1.1	1.3
16	1.5	2.2	1.3	1.7
64	7.1	17.9	2.7	2.2
256	90.4	243.5	25.3	18.8
512	279.2	929.1	93.6	67.9
1024	914.1	3836.6	341.2	233.1

Fig. 7: Table of results for Activity Optimization in ms/I

Parallelizing the most inner loop again resulted in the worst performance denoted by the red line in Fig. 9. Since that loop iterates over each body, it is possible that as the number of bodies is increased the results would improve. However, the cost of initialising threads inside a nested loop would likely still overweight any potential speedups. The two outer loops both iterate over D and parallelising either one resulted in similar speed up with slightly better results for the outer loop. This is again due to the reduced number of thread initialisations. In any case, the activity array has shared access with the possibility or raise condition if not managed properly. Both critical section and atomic has been considered, however since the key operation is a simple addition, atomic is for sure a more optimal solution and was used throughout.

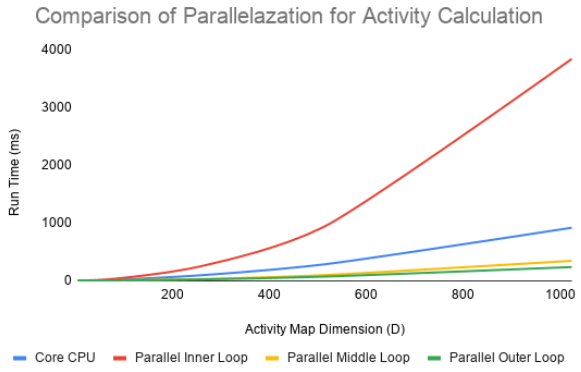


Fig. 8: Graph of results for Activity Calculation optimizations

Some scheduling approaches also have been considered; however, it was found that they had either no effect or slightly increased the run-time. This is likely due to slight computational overhead of setting up schedules, but no real advantages and the threads are very balanced, and there are no common memory access patterns.

	Experiment Name					
N	Core CPU	Parallel Second Loop	Parallel First Inner	Parallel First Outer	Schedule Static (4)	Schedule Dynamic (N/64)
64	0.3	0.2	0.4	0.5	0.2	0.2
128	1.1	1.2	1.2	0.5	0.5	0.5
256	4.2	4.8	3.8	2	2.1	1.3
512	16.4	19.4	12.5	6.1	5.6	4.5
1024	65.1	77.8	48.5	22.3	20.1	16
2048	260.1	278	176.9	94.9	68.1	56.6
4096	1042.8	1072.6	692.2	320.1	214.4	186.2
8192	4155.9	4182.5	2721.7	1258.9	876.8	685.4

Fig. 9: Table of results for second version of step function

III. Better Step Algorithm

The initial implementation of step function iterates over everybody twice with time complexity $O(n^2)$. However, it is easy to recognise that the force exerted by the first body on the second one is equal and opposite to the force that the second one exerts on the first. This means that the key calculation needs to happen only once between any pair of bodies. This allows to code a solution that has $O(nlog(n))$ time complexity shown in Fig. 10, with improvement for the CPU version highlighted in Fig. 11. It is worth noting that the calculation of the sum and actual update are split into two separate loops which does not make much sense for the CPU implementation. However, for parallel implementation, there is a possibility that final update for a given body could happen before interaction with all other bodies have been calculated, even if the sequential clause is used.

```
for (i = 0; i < N; i++) {
    b_i = &bodies[i];
    for (j = i+1; j < N; j++) {
        b_j = &bodies[j];

        diff_x = b_j->x - b_i->x;
        temp_x = (b_j->m * diff_x) / (float)pow(diff_x*diff_x + SOFTENING2, 1.5f);
        sum_x[i] += temp_x;
        sum_x[j] -= temp_x;

        diff_y = b_j->y - b_i->y;
        temp_y = (b_j->m * diff_y) / (float)pow(diff_y*diff_y + SOFTENING2, 1.5f);
        sum_y[i] += temp_y;
        sum_y[j] -= temp_y;
    }
}
```

Fig. 10: nlog(n) version of the step function (first loop)

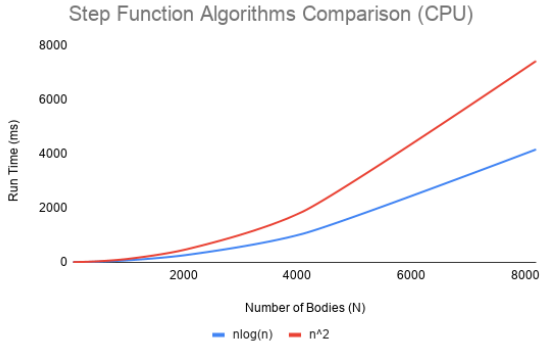


Fig. 11

A. Parallelizing Over Different Loops

A first interesting result is that parallelizing the second loop, which performs the update did not have any advantage, which is likely because of its small computational requirements. The red line in Fig. 12 demonstrates this, which is almost equivalent to the CPU version. As before it was found that inner loop parallelization improves the run-time by a bit, but the parallel outer loop is by far the optimal solution. It is interesting to note that although CPU version is much better than in previous step algorithm, with parallelism applied there is almost no difference for unscheduled version.

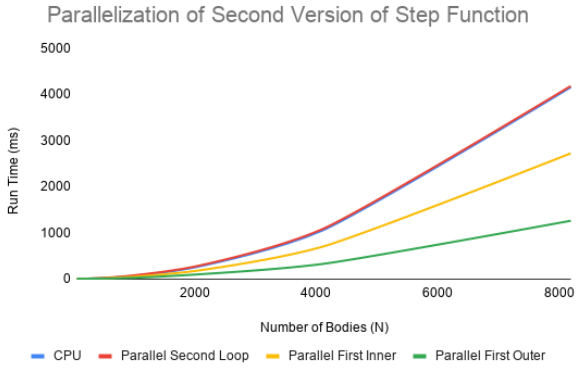
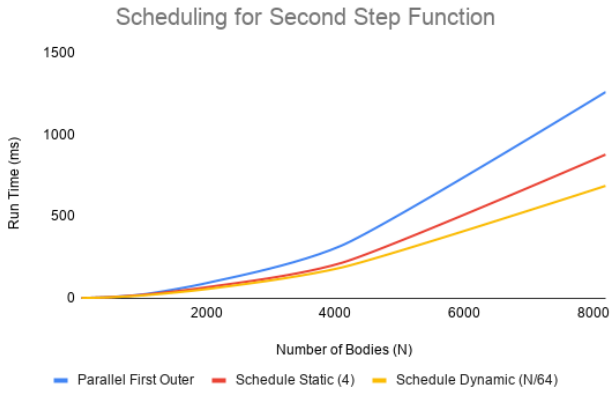


Fig. 12: Graph of results for parallel versions of nlog(n) algorithm

B. Scheduling

Static scheduling with chunk size 4 had some speed up, but particular success was achieved with dynamic scheduling. At first, fixed values were tested, but it was quickly found that optimal chunk size varied depending on the number of bodies N. Final solution set chunk size dependent on N with the optimal value found to be N/64, ceiled to have min value of 1. Guided scheduling was also tested as the amount of work reduces with time, but it did not have much impact. Final results achieved with scheduling were significantly better than for the first version of the step algorithm, highlighting the importance of scheduling for imbalanced workloads.



IV. Final Results

N	Release mode (CPU)	Release mode (OpenMP)
64	0.2	0.1
128	0.8	0.4
256	3.7	1.2
512	14.6	4.3
1024	58	13.1
2048	231.3	43.3
4096	919.4	152.3
8192	3702.9	585.3

Fig. 13

Results tested for optimal combination of activity calculation and step function in release mode are shown in table in Fig. 13. It was decided to report final results for different values of N as it is less likely to have higher values of D in real use cases.