Optimizing Parallel Computing Performance Using OpenMP in Particle Collision Simulation

Implementation Overview

Algorithm

In this project, we implemented a parallel particle collision simulation using the **broad-phase collision detection algorithm**. The simulation involves assigning particles to square spaces by dividing the simulation space into a grid of cells, where collisions are then resolved in their own cells and direct neighbours.

Key Data Structures:

- Grid Cells**:** A two-dimensional grid where each cell contains a list of particle indices located within it.
- Neighbor Map**:** A precomputed map that stores, for each grid cell, the indices of neighboring cells that need to be checked for potential collisions. This reduces redundant neighbor calculations during simulation steps.
- Thread-Local Caches**:** Each thread maintains its own caches for particle-wall overlaps and particle-particle overlaps. This approach minimizes synchronization overhead and contention among threads.

Parallelization Strategy:

The focus was to parallelize computationally intensive tasks across multiple threads, making use of the multi-core processors to improve performance. Here are some of the parallelized components

1. Particle position updates: Each particle's position is updated independently, allowing for straightforward parallelization.
2. Grid Assignment**:**
    - Thread-Local Grids**:** Particles are assigned to thread-local grids to avoid write conflicts when multiple threads attempt to insert into the same grid cell.
    - Global Grid Merging**:** Thread-local grids are then merged into a global grid in parallel.
3. Collision Detection and Resolution:
    - Collision Detection**:** Grid cells are divided among threads to check for collisions within cells and with neighboring cells.
    - Thread-Local Overlap Caches**:** Threads store detected overlaps in local caches to minimize synchronization.
    - Collision Resolution**:** Collisions are resolved using data from the thread-local caches.

What OpenMP constructs did you use and why did you use it for those specific constructs?

#pragma omp parallel for

- Parallelizes loops where each iteration is independent. This was used extensively for updating particle positions, merging grids, and processing collision checks.

#pragma omp for nowait

- Parallelizes loops without an implicit barrier at the end, allowing threads to proceed independently. This was applied during grid assignment to thread-local grids to enhance performance by reducing waiting time.

#pragma omp parallel

- Creates a team of threads for concurrent execution of code blocks. This was used when multiple operations need to be performed in parallel sections.

#pragma omp parallel for collapse(2)

- Collapses nested loops into a single loop for better load balancing. This was used when initializing the neighbor map to distribute iterations more effectively across threads.

How work is divided among threads

Work is divided among threads to parallelize the main computational tasks, this includes:

- Particle Updates: Each thread updates a subset of particles, this ensures independent computation without a need for synchronization.
- Grid Assignment: Each thread would handle the particles that are assigned to their own portions of the grid, avoiding write conflicts.
- Grid Merging: Threads merge their local grids into the global grid in parallel, with each thread responsible for specific grid cells.
- Collision Detection and Resolution: Grid cells are distributed among threads for collision detection, and each thread uses their local overlap cache to store and resolve collisions.

How you handled synchronization in your program.

By using thread-local data structures such as thread-local grids and caches it reduces the need for locks and atomic operations. By doing so, we also ensured that threads operate on separate data or read-only shared data to prevent conflicts. In places where synchronisation of threads were necessary at the end of parallel regions, we relied on OpenMP's built-in barriers.

How and why your program's performance scales with the number of threads. Vary the number of threads and present the data and trends clearly

Our program's performance scales with the number of threads due to the effective division of work among threads. As more threads are utilized, the execution time decreases significantly. For instance, the program's runtime improves exponentially from 1.04 seconds with a single thread to 0.46 seconds when running on eight threads. This scalability is achieved by evenly distributing the computational load, ensuring that each thread contributes efficiently to the overall task.

Number of threads	Time taken/s
1	1.0342
2	0.6766
4	0.5425
8	0.4605

Description, visualization and data of execution

How and why your program's performance changes based on parameters in the input file.

Input file	Cache References (millions)	Cache Misses (millions)	Cycles (billions)	Instructions (billions)	Time taken/sec
Large 0.9	11095	971	240.613	98.054	7.80
Large 0.8	3662	677	127.782	44.038	4.12
Standard 0.9	498	0.257	14.024	5.938	0.46

From standard 0.9 to large 0.9, the density is similar but the difference is the size of the stimulation, where large 0.9 have 10 times the number of particles. The reason for this change is that for a bigger stimulation, there will be more particles and hence there will be more wall-particle collision and particle-particle collisions. This will result in the number of statements executed in the program to increase, hence resulting in the number of instructions and cycles executed by the program to increase exponentially. This also lead to more cache references and more cache misses, hence resulting in memory access, which is a slow process to increase. This lead to a massive increase in time taken to complete the program.

From large 0.8 to large 0.9, the difference is the density of the stimulation. This result in the program to be taking twice as long. This is because with a denser stimulation, there will be

more collisions between the particles, hence for every single timesteps, more collisions have to be detected and resolved. This will result in the number of statements executed in the program to increase, hence resulting in the number of instructions and cycles executed by the program to increase exponentially. This also lead to more cache references and more cache misses, hence resulting in memory access, which is a slow process to increase. This lead to a massive increase in time taken to complete the program.

How and why your program's performance is affected by the type of machine you run it on. Include a comparison of at least three different hardware types.

Hardware	Cache References (millions)	Cache Misses (millions)	Cycles (billions)	Instructions (billions)	Time taken/sec
xs-4114	77.067	0.142	11.129	6.432	0.597
i7-9700	525.237	0.552	10.705	18.525	0.335
i7-7700	498.061	1.773	14.024	5.938	0.461

The results shown above is produced by using the standard 0.9 density test case for 8 threads across all hardware architectures.

The performance of your program is affected by the type of machine you run it on due to several factors related to the hardware architecture, particularly CPU design, cache sizes, memory bandwidth, and clock speeds.

The i7-9700 is the fastest of 0.3351 seconds followed by the i7-7700 of 0.46053 seconds and lastly the xs-4114 of 0.59705 seconds. This shows that the i7-9700 have the most efficient architecture with better instruction throughput and memory access optimisation. The xs-4114 is the slowest as it has the lowest frequency for its clock cycles of 2.581 GHz as compared to other hardware architecture of 3.971 GHz for the i7-7700 and 4.268 GHz for the i7-9700.

The i7-7700 have a lower instructions/cycle as compared to other hardware architectures. That is because we are running the stimulation using 8 threads. For the other hardware architectures such as the xs-4114 with 10 cores and 20 execution context and the i7-9700 with 8 cores, there will be no contention for execution of the 8 threads on the cores. However, for the i7-7700, with 4 cores and 8 execution context, although 8 threads can be executed in parallel, 2 threads will have to utilise the same core at any point of time. Although there is hyperthreading where there are 2 execution context for every single core, there will still be contention for resources on the core when 2 threads are trying to execute various instructions as there is only 1 data cache, 1 fetch and decode and 1 alu for every single core.

The i7-9700 have the lowest cache miss rate as compared to other hardware architecture. This means that it utilize its cache more efficiently and have a lower memory access latency.

This is in contrast to the i7-7700, which have much higher cache misses as compared to the i7-9700, which means that more memory access are being handled outside the fast-access cache, slowing down the execution of the program.

Performance optimisation attempted

Optimisation 1: Precomputing Neighbor Map

We precomputed a neighbor map for each grid cell during the initialization phase. This map stores neighboring grid indices that need to be checked for potential collisions, eliminating the need to calculate neighbors during each simulation step.

Hypothesis:By reducing redundant neighbor calculations and minimizing the number of comparisons between grid cells (from eight to four), we can decrease computational overhead and improve performance.

Results:

- Reduced Computations: The number of neighbor calculations per simulation step decreased, leading to faster collision detection.
- Improved Scalability: The optimization had a more significant impact on larger simulations where the number of grid cells and particles is higher.
- Performance Gain: Although specific timing data is not available (due to it being implemented at the start), observational results indicated smoother and faster execution, especially in high-density scenarios.

Optimisation 2: Using thread caches

We introduced thread-local caches for storing particle-wall and particle-particle overlaps during collision detection. Each thread maintains its own cache, which is later merged into global caches for collision resolution.

Hypothesis: Using thread-local caches can Minimize Synchronization Overhead by Reducing the need for locks or atomic operations since threads operate on their own data. Enhance Cache Locality by Improving memory access patterns, reducing cache misses and memory latency and Reduce Contention by avoiding threads competing for shared data structures, leading to better parallel efficiency.

Supporting measurements:

	Cache References (millions)	Cache Misses (millions)	Cycles (billions)	Instructions (billions)	Time taken/sec
Before	602	5.535	30.804	7.850	1.28
After	545	5.699	8.291	7.687	0.61

Appendix

We generally use slurm to run our programs, the values in the tables otherwise stated is using standard of density 0.9 and a 8 threaded program.

The command used to run most of the performance analysis comes from
srun --partition i7-7700 perf stat -e cache-references,cache-misses,cycles,instructions --
./sim.perf tests/standard/10k_density_0.9.in 8, changing the hardware(i7-7700, i7-9700, xs-4114) and number of threads(1,2,4,8) as needed

The command below is used to get the computer processing speed in GHz
srun --partition i7-7700 perf stat -e -- ./sim.perf tests/standard/10k_density_0.9.in 8