Report

Name:Qianwen Zhang
Id:qz386
Username:qz1999

1. Particle Binning

In a two-dimensional plane, a particle can have eight neighboring cells: up, down, left, right, upper-left, lower-left, upper-right, and lower-right. Including the bin it resides in, there are a total of nine bins involved. As depicted in Figure 1, the size of each bin is set to the cutoff value, such that a particle at the center cell can potentially interact with particles in all four cardinal directions (up, down, left, right). To manage these bins, we employ a two-dimensional array Bin_array, the data structure for an individual Bin is as follows:

```
typedef struct
Bin {
std::list<particle_t*> plist;
}Bin;
Bin_array[bin_num][ bin_num]
```

The rationale behind opting for std::list instead of std::vector to store data is due to the difference in their time complexity for deletion operations. With std::list, removing a node requires constant time, i.e., O(1), which means that the time taken to delete an element does not increase with the size of the list. In contrast, when using std::vector, deleting a node necessitates shifting elements following the deleted position, thus resulting in a time complexity of O(n). As shown in Figure 2, when the number of particles increases, the time taken by the serial version to complete the simulation grows exponentially. This is expected because the workload in a serial implementation isn't divided among multiple processors or cores.

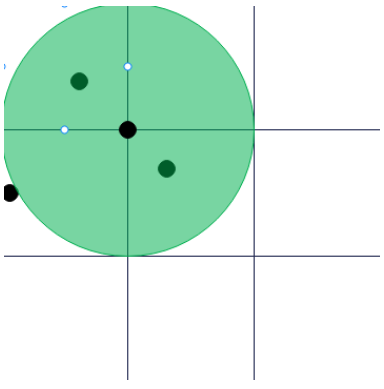

Fig1 Particle Bins

2. Parallelization with OpenMP

The parallel workflow of the Openmp_B scheme is depicted in Figure 3. After running the simulate_one_step function, at point A, the program utilizes the OpenMP pragma #pragma omp for collapse(2) to decompose the force analysis process into n subtasks, which are then distributed among n threads for computation. Due to Barrier A that exists at the end of this work-sharing construct, the multithreaded execution synchronizes here. At point B, another pragma #pragma omp for is employed to break down the movement processing stage into n equal parts. these tasks are executed concurrently by different threads, and synchronization occurs at Barrier B. Finally, the task of updating the map is handled similarly. It's also decomposed using #pragma omp for collapse(2), divided the work into multiple parallel subtasks. synchronization occurs at Barrier C. All three barriers A, B, and C are implicit barriers at the end of for worksharing directive.

The first plan Openmp_A involved using the #pragma omp critical directive to synchronize threads at points DEFG. after testing, it was observed that this approach resulted in lower efficiency compared to the serial versio. show in The use of #pragma omp critical sections can introduce significant contention among threads, which may lead to decreased parallelism and, under certain conditions, even slower execution times than a well-optimized sequential algorithm.

In fact Openmp_B at point C is decomposed into four sub-loops. For the size is limited, It was not depicted in the fig 4. If this decomposition were not performed, it would lead to contention that results in program errors. shown in fig 3, When particles are added or removed in the left and right 2 bins, a scan of the adjacent 8 directions takes place, with the central region being the area prone to race conditions. An interesting observation is that the for loop at point C can also run successfully on Intel CPUs without being broken down into four sub-loops, while still producing correct results. However, when executed on AMD CPUs, the program directly encounters errors. This suggests that Intel CPUs may have specific optimizations in their implementation of the OpenMP library to handle such scenarios effectively.

#particles	1’000	10’000	100’000	1’000’000
Times(s)/serial	0.0433	0.562	6.102	184.8
Openmp_A(256t)	0.193	1.067	12.63	135
Openmp_B(256t)	0.012	0.369	2.177	26.89

Table1 result of speed test

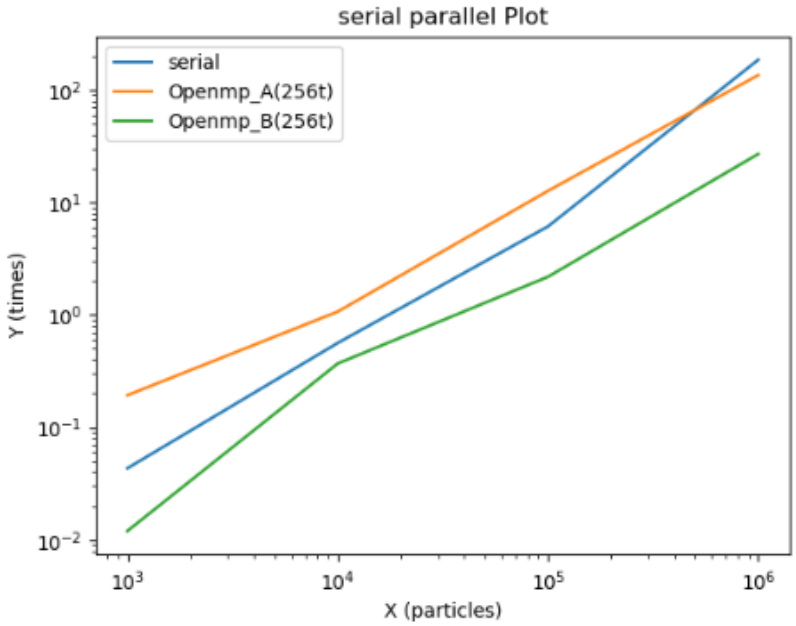

Fig2 serial and parallel log-log plot

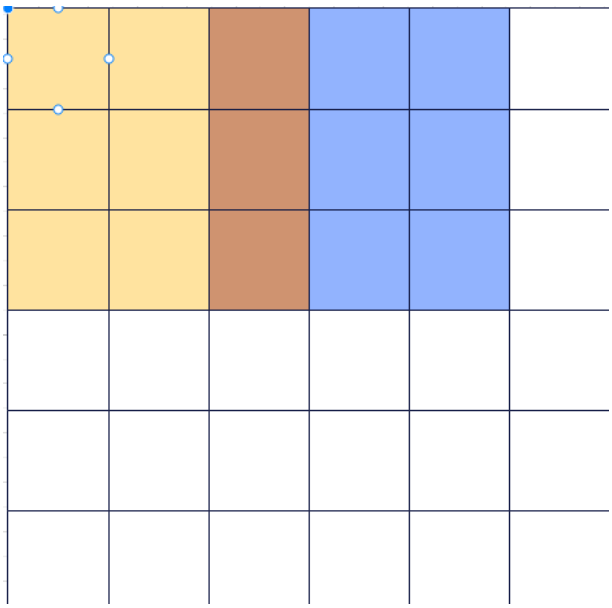

Fig3 the Competition of update map

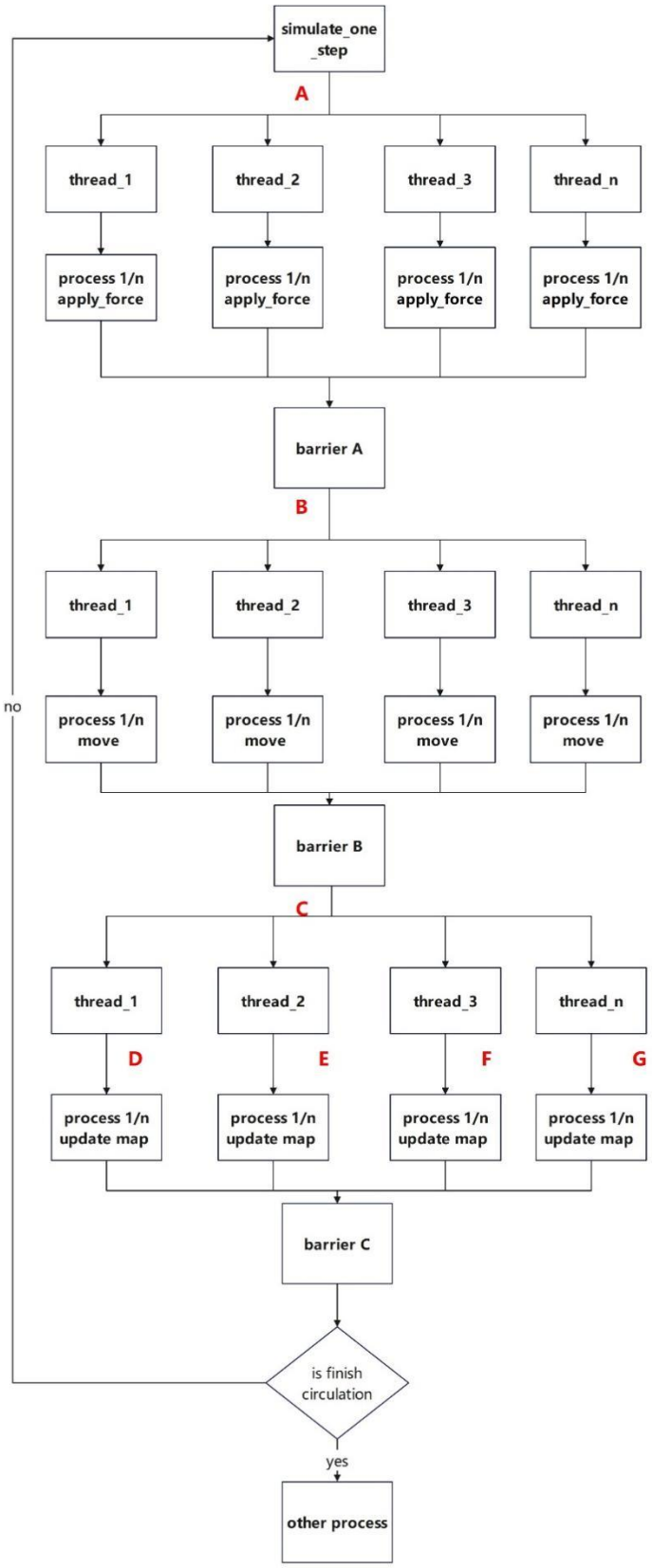

Fig4 Parallel Flow chart

3. Amdahl's Law and Scaling Benchmarks

I attempted to lease two Perlmutter nodes . after testing, observed that while the number of processors increased, there was no change in the runtime, indicating that only a single node was actively working. for this test, I adopted threads as the processing units and examined both strong and weak scaling by increasing the number of threads.

Strong Scaling

In Figure 5, this implies a decrease in scalability and efficiency with respect to thread count, where ideally, the runtime should decrease proportionally with the increase in parallel processing resources.

To get better performance and scalability, we can do the following improvements:

- optimize algorithms to minimize non-parallelizable parts and maximize parallelizable components.
- Dynamic load balancing: Implementing dynamic scheduling to ensure that all threads are assigned a workload that is approximately equal.
- Optimizing Data Access Patterns: Employing suitable data structures and access patterns to facilitate reference locality and effective cache utilization.。

In summary, achieving near-linear acceleration is a challenging task, especially for highly parallel systems. Our objective should be to identify an optimal point where adding more threads still yields substantial speed improvements without incurring excessive overheads. In this experiment, the optimal point was found to be 16 threads, beyond which no significant increase in performance could be observed.。

#particles(1'000'000)									
threads	1	2	4	8	16	32	64	128	256
Times(s)	215.3	111.2	90.5	44	24.3	25.22	22.55	19.7	26.8

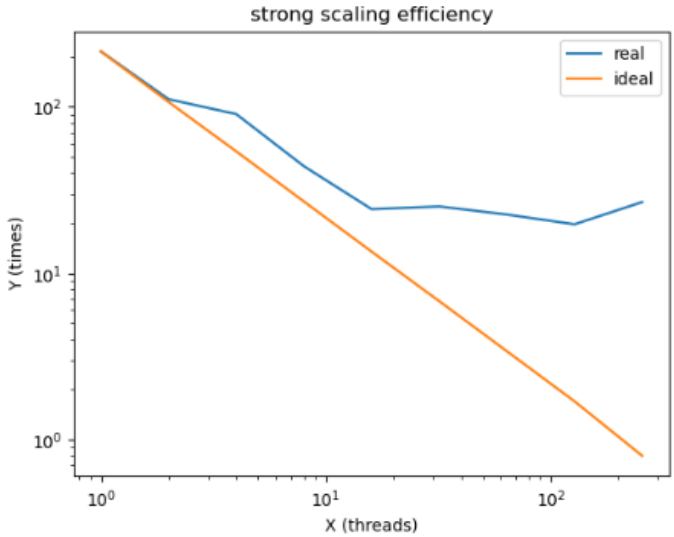

Fig 5 strong scaling

Weak Scaling

Unexpectedly, the actual performance turned out to be better than the ideal scenario. In this experiment, the total runtime can be divided into several components::

- Computation Time: This refers to the time spent on actual computations and operations, encompassing all thread-local work. In a weak scaling scenario, ideally, when the number of threads (p) is increased, if each thread's workload is proportionally equal, then the computation time for each individual thread should remain constant.。
- Computation Time: This refers to the time spent on actual computations and operations, encompassing all thread-local work. In a weak scaling scenario,
- Communication Time: This encompasses the time spent on transmitting data between different threads, maintaining cache coherence, and accessing shared resources.

In this experiment, there was a noticeable fluctuation in the data when transitioning from 8 to 16 threads. This could potentially be attributed to an increase in synchronization overheads, or it may be due to the problem decomposition not being perfectly balanced, leading to some threads experiencing longer wait times.

#particles	100'000	200'000	400'000	800'000	1'600'000
threads	1	2	4	8	16
Times(s)	5.887	12.75	23.7	30.42	56.6

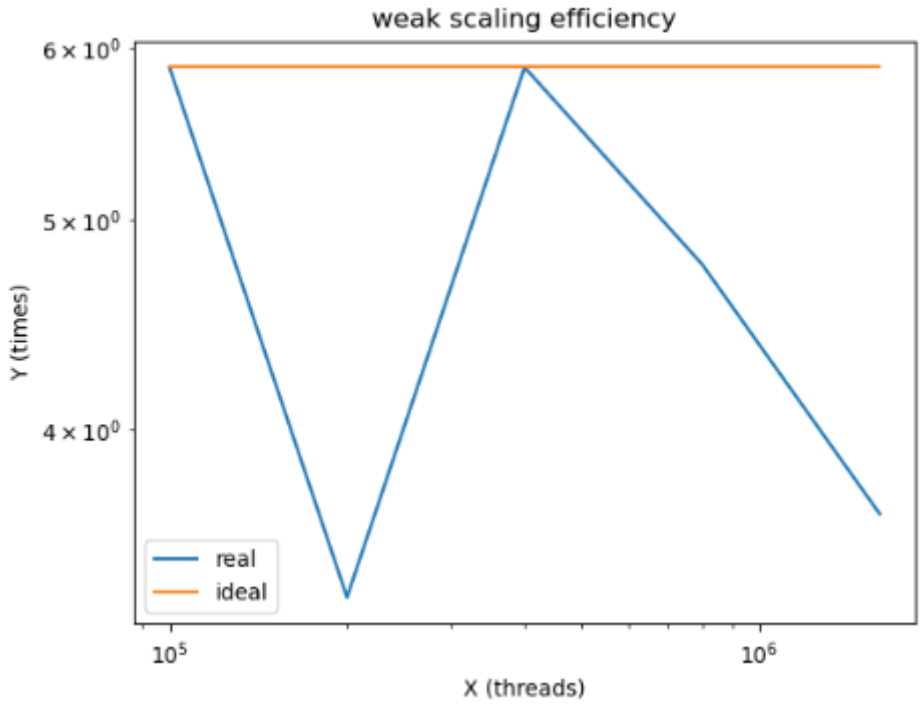

Fig 6 weak scaling