Strong scalability analysis

Strong scalability focuses on the change in computing time when increasing computing resources (the number of threads) under a fixed problem size. Ideally, adding threads should linearly reduce computation time, but in practice the performance improvement may not be so ideal due to factors such as thread creation, management overhead, and synchronization delays.

Part 1 (C++ Parallelization): The data shows that as the number of threads increases, the calculation time does not decrease significantly, but increases in some cases. This may be because thread management and synchronization overhead offset the benefits of parallel computing, especially when the number of threads is large.

Part 2 (OpenMP Custom Parallel Reduce): Performance improves as you add threads, especially when moving from single to dual threads. This shows that OpenMP's thread management is more efficient than the C++ standard library implementation, but as the number of threads continues to increase, the performance improvement gradually decreases.

Part 3 (OpenMP built-in reduction operation): Similar to Part 2, performance improves when adding threads in the early stages, but as the number of threads increases, the improvement decreases. This may be because the built-in reduction operations optimize the synchronization overhead of some parallel computations.

Weak scalability analysis

Weak scalability concerns the change in computation time when increasing computational resources and correspondingly increasing problem size. Ideally, the computation time should remain roughly constant.

All parts: Weak scalability tests show that the computation time does not remain constant as the problem size and the number of threads increase simultaneously. This points out bottlenecks in the parallelization process, such as memory access latency, data synchronization, and thread management overhead.

Summary and comparison

Performance: In terms of strong scalability, all methods fail to show ideal linear speedup. This shows that thread overhead and synchronization latency have a significant impact on performance. In terms of weak scalability, as the problem size increases, the increase in computation time also implies a decrease in parallel efficiency.

Method comparison: The OpenMP method (Part 2 and Part 3) performs better than the pure C++ method (Part 1) in most cases, especially when the number of threads is small. This may be due to the fact that the parallelization constructs provided by OpenMP are more efficient and easier to manage than threads in the C++ standard library.

Recommendation: For applications that require parallelization, it is recommended to use OpenMP, especially its built-in reduction operations, because it simplifies the writing of parallelization code and usually provides better performance. However, developers still need to pay attention to the selection of the number of threads to avoid the management overhead caused by too many threads.