Optimizations of Jos Stam's Fluid Algorithm

Parallel Computing - Phase II

Carlos Ribeiro
PG55926

Diogo Matos
PG55934

Júlio Pinto
PG57883

Abstract—**This report explores the optimizations made to Jos Stam's Fluid Algorithm for calculating real-time fluid dynamics. The optimized code was written in C++.**

I. Introduction

In the second phase of this parallel computing project, we were tasked with optimizing part of the code developed in the previous phase using OpenMP. For this phase, we had to change `SIZE` from 42 to 84 and modify the algorithm behind the function `lin_solve`.

II. Identifying Hot Spots

The primary **computational bottleneck** in the provided code was the **`lin_solve()`** function. This function involves multiple computationally intensive operations within nested loops, leading to a **large number of iterations**. Other smaller hot spots were identified in the loops outside `lin_solve()`, which were also optimized using OpenMP.

III. OpenMP Optimizations

To enhance the performance of the code, we employed OpenMP pragmas to parallelize the computational hot spots, focusing on distributing workloads efficiently across available CPU cores. The following approaches were implemented for different parts of the code:

A. *General Loop Parallelization*:

The **`#pragma omp parallel for`** directive was utilized for all independent `for` loops, specifically those that could be effectively parallelized. This enabled the iterations to be distributed across multiple threads, ensuring balanced workload allocation and improved execution speed.

The loops within the **`add_source()`, `set_bnd()`, `advect()`, and `project()`** functions exclusively employed `#pragma omp parallel for` due to the **absence of dependencies between iterations**. Additional clauses, such as `private` or `reduction`, were deemed unnecessary, as these functions **did not require private variables or the aggregation of values** across iterations.

Overall, the `collapse` clause wasn't utilized throughout the code, as its application did not produce sufficiently improved results to justify the associated **increase in overhead**.

B. *Optimizations in `lin_solve()`*:

The `lin_solve()` function, the most computationally intensive part of the algorithm, required special handling for parallelization. This function **performs multiple nested loop iterations** while solving linear equations iteratively using the Gauss-Seidel method. To **correctly compute the maximum value of `change`**, ensuring an accurate convergence, while maintaining thread safety, we used the **`reduction` clause**. The `reduction(max:max_c)` clause aggregates the maximum difference (`max_c`) across threads after each iteration, **preventing race conditions** while monitoring convergence. It also **eliminates the need for explicit synchronization**, like critical sections, **reducing overhead and improving efficiency** of the parallel code.

We also **used `private(old_x, change)`** to guarantee that each thread had its own set of `old_x` and `change` variables, **not allowing data races** to occur, since these variables are accessed in every iteration of the loop and would otherwise try be accessed by different threads, leading to a race condition.

IV. Strong Scalability

Analyzing our code's performance, **we observed near-linear speedup** for smaller thread counts, indicating efficient workload distribution at lower levels of parallelism. However, **as thread counts increased, diminishing returns became evident**. This behavior can be attributed to **synchronization overhead**, **resource contention**, and other factors that reduce the efficiency of additional threads.

The speedup graph (Fig. 1) illustrates this trend, showing an initial **rapid increase in speedup**, followed by a plateau and eventual **decline as overhead effects dominate**. Similarly, the time-per-thread graph (Fig. 2) highlights how increasing threads introduce diminishing returns, with **execution time per thread ceasing to improve significantly beyond a certain point**. The efficiency graph (Fig. 3) provides further insight into this behavior, showing the ratio of achieved speedup per number of threads.

Efficiency **starts high**, approaching 1 for low thread counts, but **drops steeply as thread counts increase**, reflecting the growing cost of synchronization and contention.

These benchmarks were conducted on the `day` partition, which has access to 24 physical cores, using multiple runs at different times to account for variability.

V. Conclusion

In conclusion, by leveraging OpenMP pragmas, we improved execution time and gained insights into parallel scalability, highlighting the need to balance workloads, optimize critical sections, and account for hardware limits to maximize efficiency.

VI. Appendix

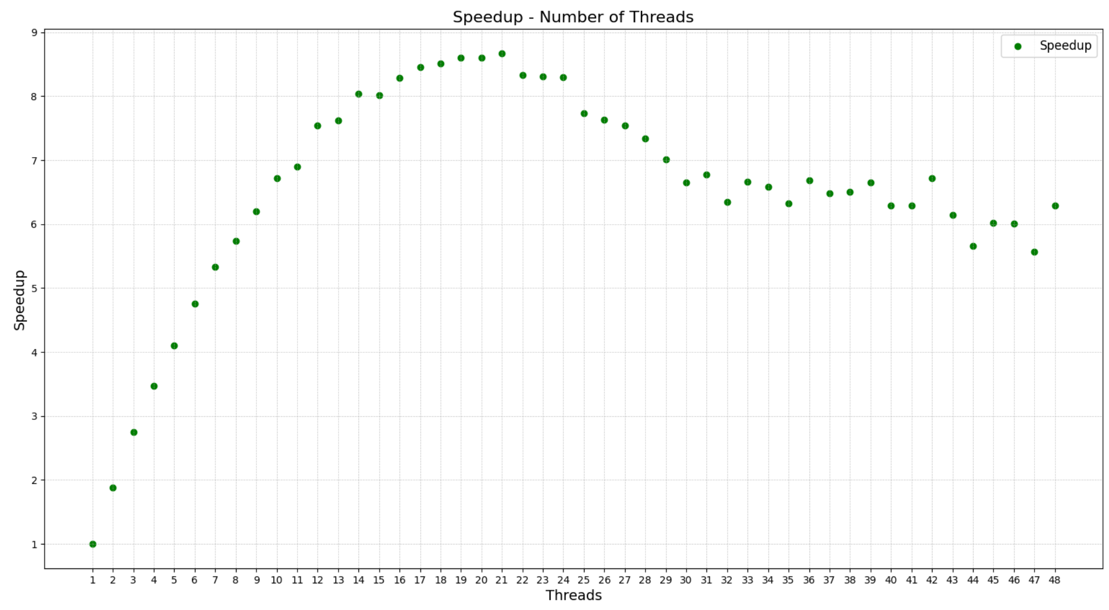

Fig. 1: Speedup Graph (partition `day`)

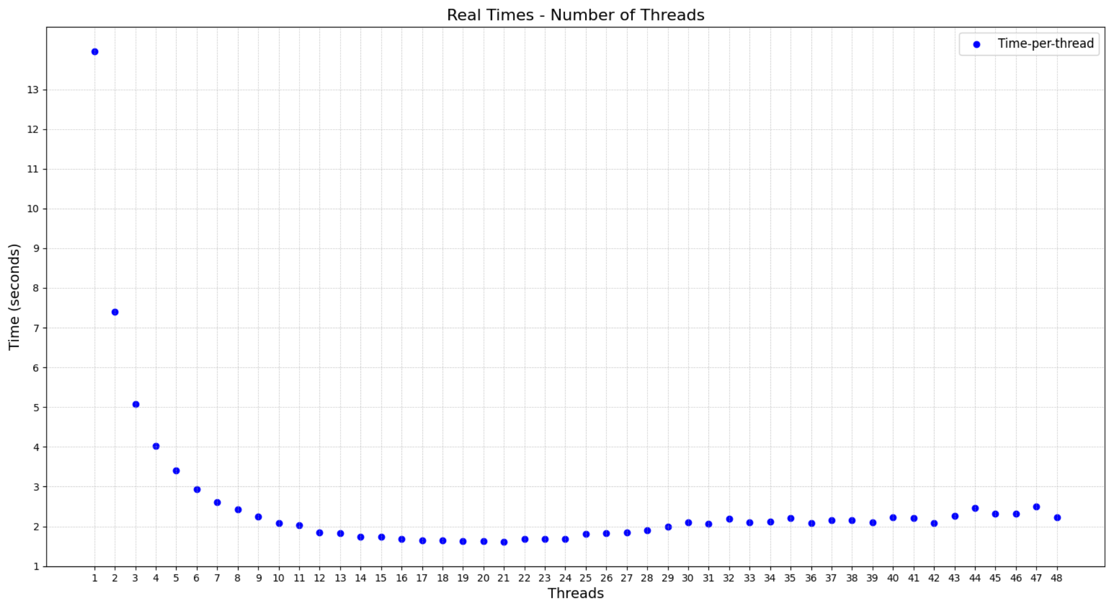

Fig. 2: Runtime Graph (partition `day`)

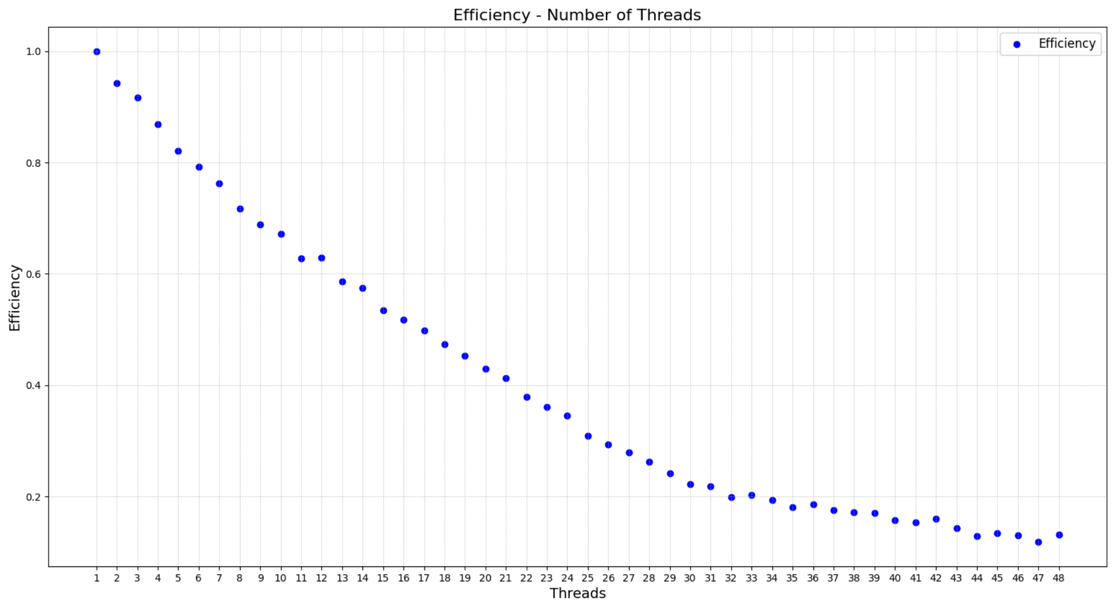

Fig. 3: Efficiency Graph (partition `day`)