A typesetting system to untangle the scientific writing process

João Paulo Gomes
PG55960

Mariana Antunes Silva
PG55980

Rodrigo Casal Novo
PG56006

***Abstract*—This project focuses on optimizing an existing 3D fluid dynamics simulation that uses a Red-black algorithm. The optimization effort targets improving computational performance and parallel efficiency using *OpenMP* library.**

***Index terms*—Paralel computing, Analysis, Optimization**

I. Code analysis

Before optimizing, we used a gprof call-graph to identify performance bottlenecks, showing function calls, their frequency, and time spent (Section IV.A).

From the analysis, lin_solver emerged as the primary optimization target, accounting for 62% of execution time. Vel_step also became significant, contributing 30%. Optimizing vel_step requires improving its dependent functions: add_source, diffuse (which calls lin_solver), advect, and project. While set_bnd itself takes little time, it will be our target too because it is crucial to prevent bottlenecks. With these targets in mind, we analyzed their code for parallelization opportunities:

A. Lin_solve analysis

The red-black algorithm allows "red" and "black" cells to be computed independently in parallel, but their interdependence prevents simultaneous computation. After this conclusion we tried to parallelize the loops but led to a data race on max_c.Also, the original loop order lacked memory locality optimization. Resolving these issues was key to improving performance.

B. Vel_step analysis

Vel_step swaps variables and calls other functions. While loops in project, advect, and set_bnd are parallelizable, subsequent loops depend on prior calculations, limiting full parallelization across blocks.

II. Optimizations and parallelization

A. Sequential version changes

Reason Loops lacked memory locality, and arithmetic calculations could be optimized.

Optimization We applied the optimizations that were doing in the previous assignment except tiling to preserve parallelization efficiency.

B. Replacing max_c with a local variable

Reason max_c caused a data race.

Solution Instead of marking a critical section for max_c updates (high overhead) or using reduction after each loop (unnecessary blocking), we assigned each thread a local_max_c variable. Threads reduced their values into the global max_c after the second loop, cutting synchronization overhead significantly.

C. Utilizing collapse in lin_solve

Reason omp for applied only to the first loop, causing uneven workloads.

Solution Using collapse(2) allowed omp for to cover two loops, balancing workloads across threads and improving performance.

D. Utilizing omp for

Reason Target function loops can be parallelized without compromising correctness.

Optimization Adding omp for distributed workloads among threads, improving performance. For lin_solve, schedule(static) ensured balanced workloads with minimal thread communication. No schedule type was specified for smaller functions due to lower impact.

E. Reducing recreation of threads

Reason Only parallelizing lin_solve required frequent thread recreation, a costly operation.

Solution Parallelizing other functions prevented thread destruction after lin_solve, improving overall performance. Even minor performance hits in small workloads were offset by the reduced thread management overhead.

III. Conclusion

With our optimizations, we achieved a 7x speedup compared to the optimized sequential version, as shown in Section IV.D, using 20 cores. Speedup is nearly linear up to 20 cores, after which it becomes unpredictable due to threads competing for resources beyond the 20 physical cores, but there is a performance drop.

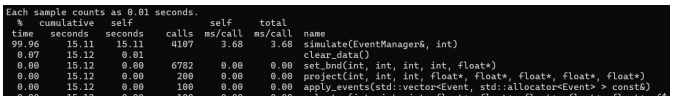
Theoretically, 20 physical cores could yield a 20x speedup, but synchronization in lin_solve after each loop and a critical section reduce performance, making it worse than sequential in those parts. The lowest execution time was 1.31 seconds, as shown at. **Section IV.C**

IV. Appendix

A. *gprof result.*

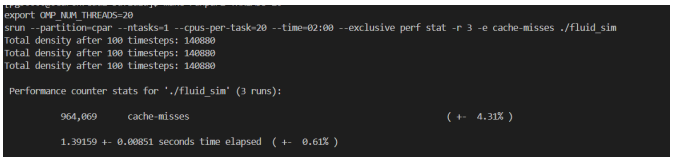
```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 52.25      7.22     7.22      300    24.07    24.07  lin_solve(int, int, int, int, float*, float*, float, float) [clone .constprop.11]
 36.83     12.31     5.09      100    50.91    78.34  vel_step(int, int, int, float*, float*, float*, float*, float*, float*, float, float)
  6.15     13.16     0.85      100     8.50    32.57  dens_step(int, int, int, float*, float*, float*, float*, float, float)
  4.20     13.74     0.58      200     2.90    27.13  project(int, int, int, float*, float*, float*, float*, float*)
```

B. *Callgraph with lin_solver optimizations.*



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.96     15.11    15.11     4107     3.68     3.68  simulate(EventManager&, int)
  0.07     15.12     0.01                             clear_data()
  0.00     15.12     0.00     6782     0.00     0.00  set_bnd(int, int, int, int, float*)
  0.00     15.12     0.00      200     0.00     0.00  project(int, int, int, float*, float*, float*, float*, float*)
  0.00     15.12     0.00      100     0.00     0.00  apply_events(std::vector<Event, std::allocator<Event> > const&)
```

C. *Final run time.*



```
export OMP_NUM_THREADS=20
srun --partition=cpar --ntasks=1 --cpus-per-task=20 --time=02:00 --exclusive perf stat -r 3 -e cache-misses ./fluid_sim
Total density after 100 timesteps: 140880
Total density after 100 timesteps: 140880
Total density after 100 timesteps: 140880

 Performance counter stats for './fluid_sim' (3 runs):

          964,069      cache-misses                                                ( +-  4.31% )

          1.39159 +- 0.00851 seconds time elapsed  ( +-  0.61% )
```

D. *SpeedUP graph*