Exploring Shared Memory Parallelism Practical Assignment Phase 2

Rita Lino
School of Engineering
University of Minho
Braga, Portugal
pg54056@alunos.uminho.pt

Diogo Barros
School of Engineering
University of Minho
Braga, Portugal
a100600@alunos.uminho.pt

Pedro Silva
School of Engineering
University of Minho
Braga, Portugal
a100745@alunos.uminho.pt

***Abstract*—In this paper, we explain how we were able to explore the concepts of memory parallelism using OpenMP directives in the assignment. Taking the previously optimized code, which was focuses on optimizing a single-threaded fluid dynamic simulation program, we now introduce the concept of multi-threading to improve the overall execution time.**

***Index Terms*—OpenMP, Multi-Threading, Parallel Computing.**

I. Introduction

Utilizing almost all the code from the previous assignment — the exception being a new definition of the `lin_solver` function, which now uses a red-black iterations with convergence check and the increase of the data size from 42 to 84, the main focus of this second assignment is set on exploring memory parallelism. After identifying and analyzing the code's hot-spots, we applied several optimizations, detailed in the following sections.

II. Code Analysis

Through analysis, several computational hot-spots were identified. The `add_source` function contributes some overhead by iterating over the data array, while `set_bnd` imposes further costs with nested loops **(loops within loops)** for boundary conditions. However, the most significant bottleneck is the `lin_solve` function, which uses a red-black Gauss-Seidel solver, adding substantial load due to its iterative nature and convergence check. The `advect` function also adds to the cost, with nested loops performing interpolation and boundary clamping, along with memory accesses that increase computational overhead. Similarly, the `project` function calculates divergence, solves for pressure using `lin_solve`, and updates the velocity components, making it another key performance bottleneck. These functions **(advect, project, and lin_solve)** are the most computationally expensive and critical for optimization.

III. Optimization Techniques

A. Parallelism Tools

The main targets to parallelize were the several loops throughout the program. Therefore, when possible, we applied `#pragma omp parallel for`, so several iterations could run across multiple threads, which enabled concurrent execution. Furthermore, the `#pragma omp simd` was used with several clauses for various purposes; the `collapse` clause, used in nested loops, substitutes a sequence of parallel work-sharing regions with a single parallel region and multiple inner work-sharing constructs; the `schedule` clause, i.e., scheduling, is used to distribute iterations to different threads in a for loop, having `static` as an input, since the static scheduling type is appropriate when the iterations require the same computational costs. In terms of variable management, OpenMP directives such as `shared` and `private` are used to control data access between threads. Shared variables are accessible by all threads, ensuring that data is consistently updated across parallel sections.

B. Compiler Flags

The flags used to compile the program were chosen as the results of the previous assignment. Thus, we chose the ones that produced the best time in the sequential execution. Furthermore, to be able to support parallel execution, the `-fopenmp` flag was added to ensure compatibility with OpenMP directives, allowing critical code sections to be parallelized across multiple threads. Finally, the `-msse4.1` flag enhances vectorized operations and boosts performance for computation-heavy parts of the program.

IV. Results

As shown in Figure 2, the number of threads used to run the program and the time it took to execute it were inversely proportional until the number of threads hit the number 16. Such might've happened due to overhead. The lack of a proportional time improvement shown when the number of threads increased shows that the system's reliability peaks when less than 32 threads are being used.

V. Conclusion

The implementation of OpenMP significantly improved the application's performance by effectively parallelizing computational hotspots. The optimized red-black solver and increased data size demonstrated scalable shared memory parallelism, reducing execution time while maintaining accuracy. These enhancements highlight the potential of OpenMP in efficiently addressing computational challenges in larger-scale problems.

APPENDIX A

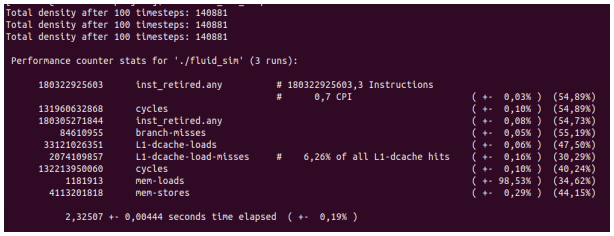

Fig. 1. Result obtained by running our optimized program.

APPENDIX B

```
OMP_NUM_THREADS=1 below
Total density after 100 timesteps: 140881

real    0m22.219s
user    0m22.175s
sys     0m0.039s
OMP_NUM_THREADS=2 below
Total density after 100 timesteps: 140881

real    0m11.244s
user    0m22.358s
sys     0m0.073s
OMP_NUM_THREADS=4 below
Total density after 100 timesteps: 140881

real    0m5.889s
user    0m23.452s
sys     0m0.050s
OMP_NUM_THREADS=8 below
Total density after 100 timesteps: 140881

real    0m3.316s
user    0m26.333s
sys     0m0.069s
OMP_NUM_THREADS=16 below
Total density after 100 timesteps: 140881

real    0m2.309s
user    0m36.555s
sys     0m0.119s
OMP_NUM_THREADS=32 below
Total density after 100 timesteps: 140881

real    0m2.589s
user    1m21.780s
sys     0m0.332s
OMP_NUM_THREADS=40 below
Total density after 100 timesteps: 140881

real    0m2.013s
user    1m19.707s
sys     0m0.055s
```

Fig. 2. Scalability exploration results with different number of threads.