Parallel Computing
Practical Assignment Phase 2

António Silva
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG57867

Duarte Leitão
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG57872

Vasco Faria
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG57905

***Abstract*—In this phase, the objective was to optimise the same problem addressed in Phase 1 using a different solver and increasing the problem's data size (N=84). The focus was on leveraging shared memory techniques, utilizing the OpenMP tool, to achieve a reduction in execution time.**

***Index Terms*—OpenMP, Shared Memory parallelism, Multi-threading**

I. Code Analysis

After thorough analysis, several computational hot-spots were identified. One notable hot-spot is the *add_source* function, which traverses the entire data array to update each element. Another significant hot-spot is the *set_bnd* function, which applies boundary conditions through nested loops spanning all grid dimensions. However, the most computationally intensive component of the code is the *lin_solve* function (Appendix A), which implements the red-black Gauss-Seidel solver. This function involves triple nested loops over the 3D grid and iteratively updates values based on neighboring cells until a convergence criterion is satisfied. The combination of complex computations, multiple iterations, and a reduction operation to evaluate convergence makes *lin_solve* the most time-consuming section of the code.

II. Implementation of Parallelism techniques

A. Compiler Flags

Regarding the compiler flags, the *-msse4.1* flag was selected to take advantage of SIMD (Single Instruction, Multiple Data) instructions, which optimise performance by enabling parallel processing of multiple data elements in a single instruction. The *-ftree-vectorize* flag was used to allow the compiler to automatically convert loops into vectorized operations, further improving the efficiency of repetitive calculations. Additionally, the *-Ofast* flag was chosen to enable aggressive optimisations aimed at achieving maximum performance. This flag includes all optimisations from -O3, while also enabling additional compiler optimisations that may break strict standard compliance in favor of improved speed.

B. Parallelism exploration

To enhance the performance of the fluid solver, shared-memory parallelism was implemented using OpenMP to distribute computational tasks across multiple threads. In the *add_source* function, the loop responsible for updating the array was parallelized using the directive *#pragma omp parallel for simd*, as each iteration is independent and can be executed concurrently. Similarly, in the *set_bnd* function, separate loops were parallelized with the *#pragma omp parallel* directive, enabling simultaneous updates and improving computational efficiency. In the *project* function, shared variables were employed to ensure that all threads operate on the same global data structures, facilitating efficient memory usage while ensuring consistency in the updates.

Parallelism was also applied in the *lin_solve* function to accelerate the Gauss-Seidel method. Both phases of the solver were parallelized using *#pragma omp parallel for collapse(2)*, flattening the two outer loops to improve load balancing across threads. A *reduction(max_c)* operation ensured consistent computation of the maximum residual across threads, a critical aspect for convergence. Thread safety was maintained by declaring variables such as *old_x* and *change* as *private*, as each thread required independent copies, while global structures like *M*, *N*, *O*, *x*, *x0*, *a*, and *inverso_c* were declared as *shared* to allow shared access to data. The choice of scheduling strategies was also analysed (Appendix B). *Schedule(static)* showed superior performance due to its low runtime overhead, whereas *schedule(dynamic)* incurred higher execution times due to increased scheduling overhead, particularly as the number of threads increased.

The graph (Appendix C) shows how thread count impacts CPI. Performance initially improves as threads distribute tasks efficiently. However, diminishing returns occur at higher thread counts due to synchronization overhead and memory bandwidth saturation, causing CPI to plateau or increase slightly.

III. Conclusion

The results highlight the effectiveness of parallel programming techniques in addressing computational bottlenecks and demonstrate the significant performance improvements achieved through targeted optimisations. Notably, the optimal performance was achieved with 20 threads, indicating an optimal balance between parallelism and the associated overhead.

IV. Appendixs

Appendix A

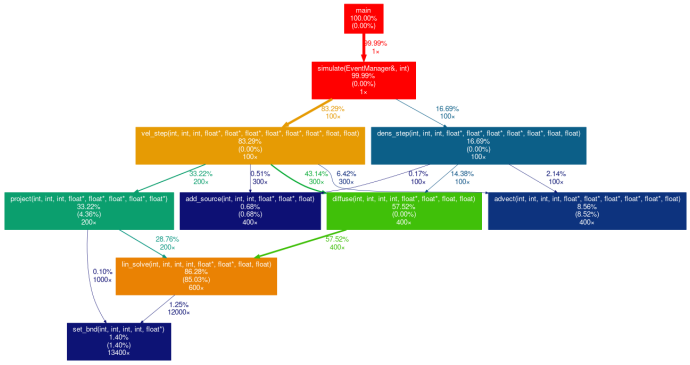

Fig. 1. Initial execution flow visualization from gprof.

Appendix B

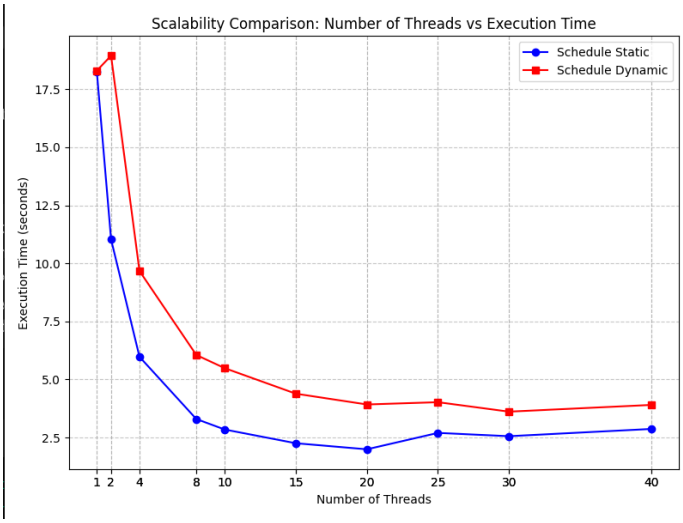

Fig. 2. Scalability comparison schedule Dynamic and Static

Appendix C

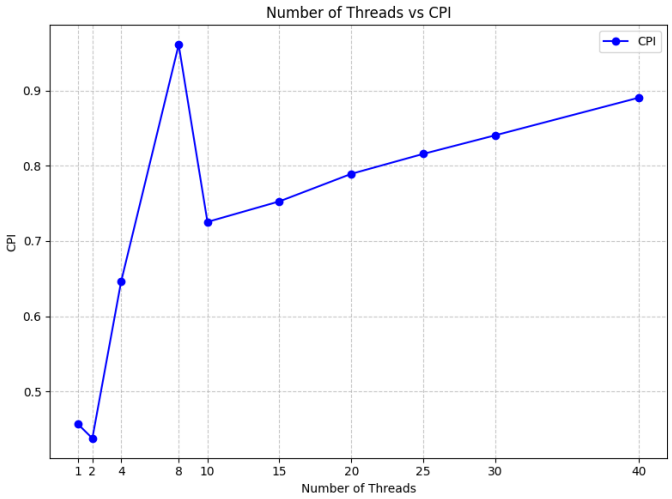

Fig. 3. CPI of different number of threads.