Parallel Computing (Phase 2)

Lucas Oliveira
PG57886

João Silva
PG55618

Rafael Gomes
PG56000

***Abstract*—In this phase of the practical assignment, we will build upon the previously provided program. The main goal is to investigate shared memory parallelism techniques by utilizing the OpenMP tool to optimize and minimize the code's execution time.**

I. Identifying Hot-Spots

To improve code performance, we identified the *lin_solve* function as the primary hotspot, consuming *58.37%* of total execution time (including child calls) using the *perf tool* (as shown in Section VI.A). Due to its significant impact, we focused on optimizing lin_solve and implemented parallelization strategies to enhance efficiency.

II. Analyse and Alternatives

To optimize the hot-spot we thought in this alternatives:

Option 1: Parallelizing Outer Loop of Red/Black Phases This approach works well when the outer loop has a big number of iterations, to take advantage of the overhead in the spawning of threads. It's also a good option if the first loop processes a large contiguous block of data, which can improve cache performance by allowing threads to work on their own block. However, this method might not be the best if there's invariant workload in the innermost loops, potentially causing imbalance between the work of different threads. This isn't our case, since every iteration of the innermost loops has the same level of workload.

Option 2: Parallelizing Inner Loop of Red/Black Phases This method provides fine-grained parallelism, maximizing thread utilization for large grids with a significant inner dimension. However, frequently spawning and destroying thread adds significant overhead. This option is viable only if the first loops have a very small number of iterations, which isn't the case.

III. Approach selected

After careful analysis, we chose Option 1 for its balanced workload distribution, parallelizing the outer loops (*k* and *j*) to minimize synchronization overhead. This approach ensures efficient execution with adequate granularity for moderate to large grids, making it the most practical and effective choice.

IV. Implementation and Optimisations

a) ***Collapse:*** We decided to collapse the two outermost loops, transforming these two in just a single loop, which ups the number of iterations that can be distributed between threads. By not collapsing the two loops, each thread would be given an iteration of the first loop, and then executing all iterations of the second loop sequently, this way we would not be taking full capacity of the threads, since the second loop doesn't have a lot of iterations.

b) ***Clauses:*** We opted for the *reduction(max:)* clause for the *max_c* variable, since there are no dependencies, we only care if in one iteration, it surpasses *tol*. In the end, the value of *max_c* will be the maximum value calculated by all threads, meaning that if one thread manages to reach the tolerance limit, the final value will also be greater than the limit.

For *old_x* and *change* we used the private clause, since each thread needs there two locally to update them.

c) ***Blocking:*** After parallelizing the code, we decided to implement the blocking algorithm in the *lin_solve* function, this consists of dividing the space of iterations into smaller blocks, this way, each thread will work within a smaller region in the memory, and, once if fetches a value and carries a block of data to cache, it's less likely for cache misses to occur.

d) ***More Parallelization:*** Given the number of cycles with big numbers of iterations and without data dependency, we decided to parallelize them, also collapsing them, so we can have more iterations to distribute for threads, as said before.

V. Measures and Conclusion

We made some tests with average times with multiple cores, and got the Fig.1

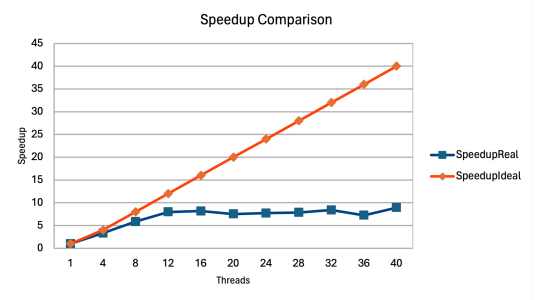

Fig.1. Graphic of the program's speedup with **N** threads

We can see that when the number of threads increase, it fails to achieve the ideal speedup, this can be caused because there are some parts of the code that are sequential and cannot be parallelized, other cause that can be its synchronization between processes, where there can be a delay in the barriers that some processes are waiting for others.

To conclude, we optimized successfully the code in performance and efficiency terms. By strategically parallelizing our code, we achieved significant improvements, though the results were still far from the ideal. Despite this, the knowledge acquired through our iterative process and the exploration of different parallelization techniques will provide valuable direction for the next phase.

VI. Appendices

A. *Results from* **perf record -g** *of the functions performance for the* ***non-optimised*** *code*

```
Samples: 215K of event 'cpu-clock', Event count (approx.): 53841750000
  Children      Self  Command    Shared Object        Symbol
+   58,37%    58,13%  fluid_sim  fluid_sim            [.] lin_solve
+   20,98%    20,87%  fluid_sim  fluid_sim            [.] lin_solve
+   11,42%     0,00%  fluid_sim  [unknown]            [k] 0000000000000000
+    8,54%     8,50%  fluid_sim  fluid_sim            [.] project
+    5,61%     5,59%  fluid_sim  fluid_sim            [.] vel_step
+    2,70%     2,69%  fluid_sim  fluid_sim            [.] dens_step
+    1,48%     1,46%  fluid_sim  fluid_sim            [.] add_source_3
+    0,77%     0,77%  fluid_sim  fluid_sim            [.] set_bnd
+    0,54%     0,54%  fluid_sim  fluid_sim            [.] add_source
```

B. *Results from* **perf record -g** *of the functions performance for the* ***optimised*** *code*

```
Samples: 325K of event 'cycles:ppp', Event count (approx.): 262369010664
  Children      Self  Command    Shared Object        Symbol
+   70,51%     0,00%  fluid_sim  [unknown]            [k] 0000000000000000
+   37,71%     0,18%  fluid_sim  libgomp.so.1.0.0     [.] gomp_thread_start
+   29,40%    29,00%  fluid_sim  libgomp.so.1.0.0     [.] gomp_team_barrier_wait_end
+   27,01%    26,72%  fluid_sim  fluid_sim            [.] set_bnd
+   26,25%     0,01%  fluid_sim  libgomp.so.1.0.0     [.] GOMP_parallel
+   22,44%     0,00%  fluid_sim  [unknown]            [.] 0x0000005400000054
+   14,18%    13,92%  fluid_sim  fluid_sim            [.] lin_solve
+   12,53%    12,31%  fluid_sim  fluid_sim            [.] lin_solve
+    5,95%     5,87%  fluid_sim  libgomp.so.1.0.0     [.] gomp_barrier_wait_end
+    1,80%     1,71%  fluid_sim  fluid_sim            [.] add_source_3
+    1,75%     1,61%  fluid_sim  fluid_sim            [.] project
+    1,71%     1,68%  fluid_sim  fluid_sim            [.] diffuse
+    1,49%     0,00%  fluid_sim  [unknown]            [.] 0x00007f9002bd0010
+    1,30%     1,28%  fluid_sim  fluid_sim            [.] advect_3
+    1,13%     1,12%  fluid_sim  fluid_sim            [.] advect
+    0,85%     0,00%  fluid_sim  [unknown]            [.] 0x00007f9002962010
+    0,84%     0,80%  fluid_sim  fluid_sim            [.] project
+    0,82%     0,00%  fluid_sim  [unknown]            [.] 0x00007f9002e3e010
+    0,73%     0,03%  fluid_sim  [kernel.kallsyms]    [k] apic_timer_interrupt
+    0,70%     0,59%  fluid_sim  fluid_sim            [.] set_bnd
+    0,66%     0,00%  fluid_sim  [kernel.kallsyms]    [k] smp_apic_timer_interrupt
+    0,60%     0,00%  fluid_sim  [kernel.kallsyms]    [k] local_apic_timer_interrupt
+    0,60%     0,59%  fluid_sim  libgomp.so.1.0.0     [.] gomp_barrier_wait
+    0,59%     0,01%  fluid_sim  [kernel.kallsyms]    [k] hrtimer_interrupt
+    0,54%     0,01%  fluid_sim  [kernel.kallsyms]    [k] __hrtimer_run_queues
+    0,52%     0,00%  fluid_sim  [kernel.kallsyms]    [k] tick_sched_timer
+    0,50%     0,00%  fluid_sim  [kernel.kallsyms]    [k] tick_sched_handle
```

C. *Graphic of the program's execution with* ***N*** *threads*