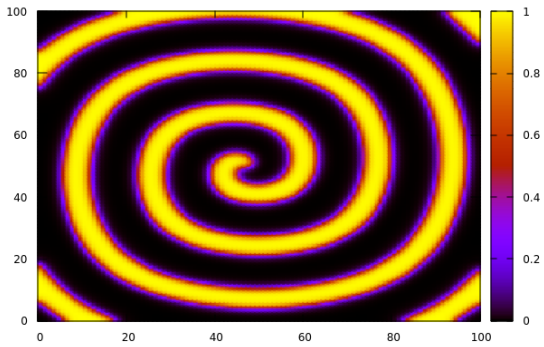
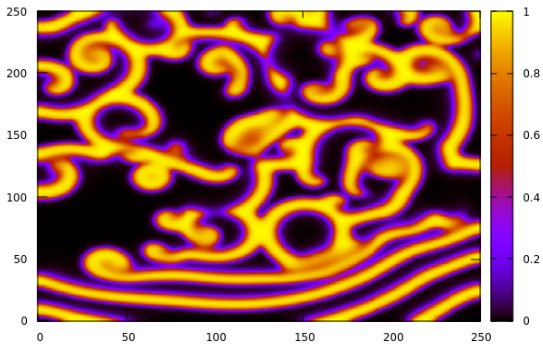
High Performance Computing (AERO70011) - Coursework Report

by Luís Marques (CID: 01715222)

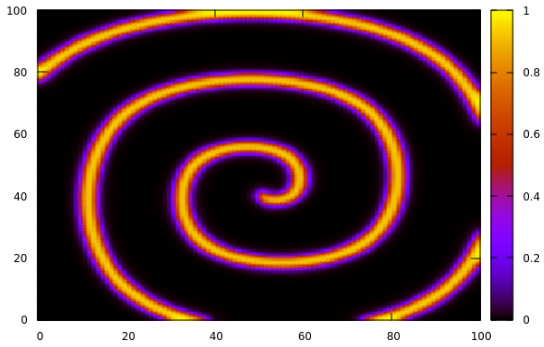
Solution of test cases



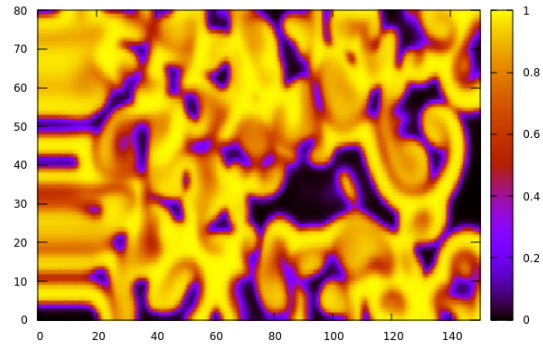
Test (a): u solution at $T = 100$s.



Test (b): u solution at $T = 100$s.



Test (c): u solution at $T = 100$s.



Test (d): u solution at $T = 100$s.

Parallelization approach

The code was parallelized via the *shared-memory paradigm*, through the use of the OpenMP API.

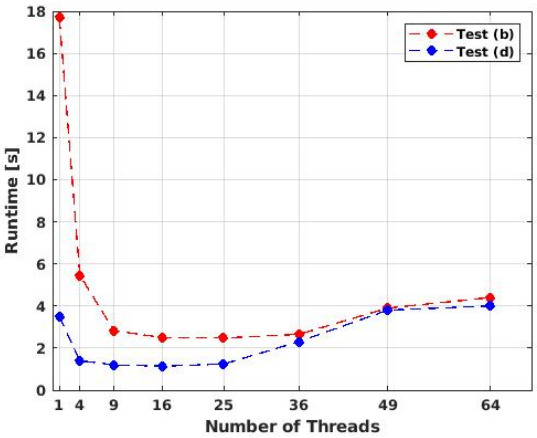
A *message passing paradigm*, as would be possible by following the MPI standard, was not adopted due to concerns that the latency cost associated with each communication would negatively affect the performance. One could perform non-blocking communication (via `Irecv`) and perform computations during said communication to offset its cost. However, without convoluting the solution method[1], a sensible approach would require a synchronization step (read latency cost) at the end of every iteration (after all the processes calculated their respective element updates, and before saving the global u^{n+1}). MPI would provide us with the flexibility of running our code on multiple machines, as each process would have its own memory space (while in OpenMP we are limited to a single shared memory space and thus a single computer). Possibly an hybrid OpenMP/MPI approach could be considered to try to leverage the benefits of both methods.

To reduce the time-to-solution it's important to consider how to distribute the workload (implicitly handled by OpenMP) and especially how to minimize the time that threads spend idle (waiting for others to finish computations). One of the advantages of the chosen paradigm is the ease of developing a working parallel code, from an already working serial code (MPI required substantial refactoring). A downside is the difficulty of having good cache-locality, especially as the number of threads increases (not trivial to obtain performant OpenMP code with multiple threads). False-sharing is a common problem of OpenMP, where multiple threads are accessing different datapoints within the same cache line (thus migrating the cache line between the threads and degrading performance with this sort of "communication"). "Memory trashing" was avoided by reading from one array and writing to a different one. A single `parallel` directive was used at each time-step, to reduce the number of times that threads were "forked" and "joined" (has associated overhead, which increases with #threads). If this directive was outside the loop, we would have less 100'000 forks/joins (taking $T = 100$ and $\Delta t = 0.001$), but this is not feasible as the time-loop should only be run once. The update of the

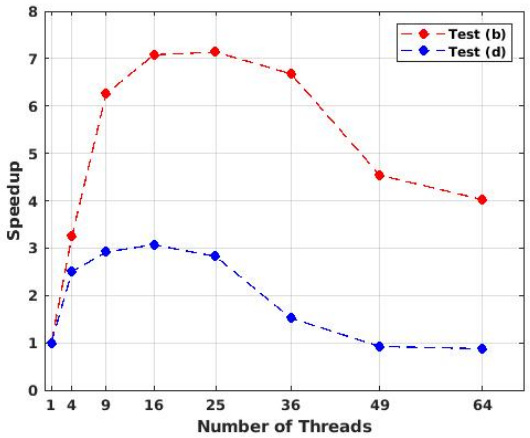
[1] One idea to avoid this synchronization step at the end of each update, would be to structure the code such that for a given $u_{i,j}^{n+1}$ element, it only needs to wait to receive the values of the $u_{i,j}^{n}, u_{i+1,j}^{n}, u_{i-1,j}^{n}, u_{i,j+1}^{n}, u_{i,j-1}^{n}, v_{i,j}^{n}$ elements.

4 corner nodes was done via `single nowait` clauses such that: statements were only evaluated by a single thread and the threads not evaluating them could progress. The edge nodes were updated through a `#pragma omp for nowait` statement (1 for-loop for edges with fixed x and another for those with fixed y), each thread thus updates a node at the time, and there's no implicit synchronization step at the end of the loop (would degrade performance). The central nodes were updated via a nested for-loop, with a `#pragma omp for nowait` placed before the outer loop (over the columns) but no statement placed before the inner loop (over the rows), each thread thus updates a row of values. Above refers to the update of u, similar occurs for v[2]. The implicit barrier at the end of the parallel region ensured that all the elements of the domain have been updated, before saving the u^{n+1}, v^{n+1} arrays.

Parallel Scaling



Runtime in seconds vs Number of Threads used.



Speedup, relative to serial code, vs Number of Threads used.

All the execution time measurements were performed on Spitfire[3] (values shown are averages of multiple measurements, where care was taken to account for the CPU load). Left plot shows execution time in seconds *vs* number of threads and right plot the speedup (relative to serial) *vs* number of threads. From Amdahl's Law [4] beyond a certain number of threads the speedup should plateau (number of threads at which plateau starts and maximum speedup/plateau value is dependent on the % of the serial execution time that can be paralellized). Despite the code clearly benefiting from parallelization, beyond a certain number of threads performance actually degrades. Spitfire has 40 CPU cores (20 on each of the 2 processor) so, despite it supporting 80 hardware threads via hyper-threading, there are only 40 FPUs, i.e. only 40 floating operations can be performed at any time. For more than 40 threads, new threads would necessarily[5] have to share a core's clock cycle with a pre-existing thread (thus there's only appearance of parallelism, as in reality they interleave resources). There's a trade-off, more threads might lead to better memory performance (threads running on same processor will have access to the same L2 and L3 cache) but once they surpass the number of CPU cores, more threads will necessarily lead to worse operational performance (more time spend idle as cycles are shared). Furthermore, with more threads the overheads of "forking/joining" threads at the parallel regions will be greater, and if this isn't offset by an increase in performance (as we've reached the limit of FPUs), then it makes sense that the execution time will increase.

Some optimizations

Multiple optimizations were considered (for both the serial and parallel version of the code), most of which are outlined below:

- Any functions inside the time-loop were *inlined*

[2]Reason for this is in Optimizations section.

[3]The code consistently ran faster on Typhoon, but since this only has 40 hardware threads it was not used.

[4]Amdahl's Law states that if we apply P processors to a task with a serial fraction f, the the predicted net speedup is given by Speedup $= 1/(f + \frac{1-f}{P})$.

[5]Note that the same can happen for less threads, as the 40 FPUs are shared between users.

- All the divisions were replaced by multiplications of the reciprocal (floating point multiplication is faster than division)
- Any constants/coefficients repeatedly used in the time-loop were pre-calculated
- Removed every conditional statements in the time-loop (previously used them to apply BCs to edge and corner nodes)
- Used `std::swap()` to pass the updated solution arrays u^{n+1}, v^{n+1} to the u^n, v^n arrays at the end of every time-step (previously copied all elements of array, and also tried swapping pointers through a third temporary pointer)
- Used only 1 `parallel` region (per time-step) to minimize thread forking/joining. Used `nowait` clause for all directives within parallel region (to remove implicit barriers) and `single` for corner nodes (only ran by 1 thread)
- Optimization flags were used[6]: O3 (highest opt. level, tries to vectorize loops) and `-march=native` (produces object code specific to the system's CPU, thus making it less portable but more performant)
- Cache-locality was optimized through 2 different methods: 1) when updating the central nodes using a nested for-loop, the inner loop went over the rows, which maximizes cache locality in a column-stored matrix; 2) in the time-step loop, the u and v solution fields were updated one at a time (i.e. first updated entire u solution field and then started updating v). While this is not perfect, as for the update of $u^n_{i,j}$ the value of $v^n_{i,j}$ is used in the reaction-term, it helps with cache-locality as most threads will share some lower level cache (still same processor). Crucial to consider as accessing data costs 100-1000s of compute cycles, and so represents "lost" possible operations. One must be really carefully about when and how data is accessed.

Inlining any function, previous called inside the time-loop, lead to considerable performance increases since each function was called at least 100'000 times (non-negligible as calling a function requires at least adding and removing 1 entry to the stack). The cache-locality similarly played a great role. Removing all conditional statements was very effective. Compiler optimization flags, pre-calculating constants and using **std::swap()** (instead of copying all the elements via a for-loop) all lead to noticeable improvements. The `schedule(static)` clause was tested, but it resulted in no performance benefits. To better exploit multiple thread cache-locality, one could try dividing the domain such that at a given time-step, the same threads would always read and write to the same memory addresses/datapoints (if same thread is always accessing same data, we could repeatedly exploit the smaller, but faster, caches). However, this would likely require the introduction of conditional statements. The use of `OMP_PROC_BIND=true` was investigated but it didn't lead to noticeable improvements.

BLAS & LAPACK?

One could transform the element-wise numerical scheme provided into matrix-form by defining the vector $\mathbf{u}^n = [u_{0,0}, u_{0,1}, u_{0,2}, \cdots u_{0,N_y-1}, u_{1,0} \cdots u_{N_x-1,N_y-1}]^T$ which holds the u solution, at time-step n, for every node in the grid (and $\mathbf{f}_1, \mathbf{f}_2$ with similar row-wise structure). The update equation would become $\mathbf{u}^{n+1} = \mathbf{u}^n + \frac{\mu_1 \Delta t}{h^2}\mathbf{A}\mathbf{u}^n + \Delta t \mathbf{f}_1 \Rightarrow \mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \Delta t \mathbf{f}_1$, with $\mathbf{B} = (\mathbf{I} - \frac{\mu_1 \Delta t}{h^2}\mathbf{A})$. Here $\mathbf{A}$[7] is a symmetrical banded matrix that encapsulates the discretized ∇^2 term as well as the BCs (a similar process could be followed for v). A BLAS implementation was tested, however despite it performing decently in serial, a loop based approach proved to be more effective as: 1) the compiler is not able to optimize BLAS routines (they're written in Fortran) while the loop approach is quite suitable for optimizations; 2) while we could pre-calculate $\mathbf{B}$ (for u,v separately) and use banded storage (very sparse storage still), multiplying it by $\mathbf{u}^n$ involves a $[Nx-2, NxNy]$ matrix and a $[NxNy, 1]$ vector at every time-step (plus we still needed to add the force/reaction term to the result). Futhermore, the BLAS based approach would be hard to parallelize. Thus BLAS was not used.

LAPACK provides routines for *solving* systems of linear equations (or providing least-square solutions to these), solving e-value problems and SV problems. Since the numerical scheme provided uses explicit time integration (and not implicit), there's no need to solve a system of linear equations and so LAPACK was not used.

[6]Refs: Gentoo - GCC Optimization and GNU GCC Docs

[7]On diagonal: -2, -3 or -4 depending on indexes of $u_{i,j}$ (i.e. BC); 1st upper diagonal +1, except it is 0 at every element where $i = N_x - 1$; next $Nx - 2$ diagonals are 0 (leads to sparse banded storage); next diagonal of +1