Shared and distributed memory parallelization of the Burger's Equations

Fabian Lyck Josua Rieder

September 20, 2021

Abstract

Based on the Python implementation of the two-dimensional viscid Burger's equations by Ubbiali (2019) [1], we present an optimized C++ implementation. We evaluate the performance of shared-memory versus distributed memory parallelism.

1 Introduction

We implement a numerical method to solve the two-dimensional viscid Burgers' equations:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$
$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)$$

With $\mathbf{v} = (u, v)$ the velocity vector field and μ being the diffusion coefficient. To permit accurate verification of our results, we limit ourselves to two analytically solvable boundary value problems, one by Zhao et al. (2011):

$$u(x, y, t) = -2\mu\frac{2\pi\, \mathrm{e}^{-5\pi^2\mu t}\cos(2\pi x)\sin(\pi y)}{2 + \mathrm{e}^{-5\pi^2\mu t}\sin(2\pi x)\sin(\pi y)}$$
$$v(x, y, t) = -2\mu\frac{\pi\, \mathrm{e}^{-5\pi^2\mu t}\sin(2\pi x)\cos(\pi y)}{2 + \mathrm{e}^{-5\pi^2\mu t}\sin(2\pi x)\sin(\pi y)}$$

And the Hopf-Cole test case by Zhu et al. (2010):

$$u(x, y, t) = \frac{3}{4} - \frac{1}{4\left(1 + \mathrm{e}^{(-t-4x+4y)/32\mu}\right)}$$
$$v(x, y, t) = \frac{3}{4} + \frac{1}{4\left(1 + \mathrm{e}^{(-t-4x+4y)/32\mu}\right)}$$

$\mathbf{v}$ is approximated by a finite difference method quantized on a regular Cartesian grid. The spatial first derivatives are calculated using a fifth-order upwind advection scheme by Baldauf (2008). The

second-order spatial derivatives are calculated using a fourth-order centered difference. The third-order Runge-Kutta scheme by Wicker and Skamarock (2002) is used for time-stepping. This numerical model taken directly from the original implementation[1].

2 Shared Memory Parallelism

We provide a shared memory parallelized implementation that is based off of the single-threaded C++ implementation, which in turn is based off of the preexisting Python reference implementation. OpenMP is used as the choice of foundational parallelization toolkit by which the single-threaded C++ implementation is extended.

Each iteration of the main loop is comprised of two principal steps: invoking the integrator and re-enforcing the boundary conditions. For a discretization of size $m \times n$, enforcing the boundary condition takes only $\mathcal{O}(m+n)$ steps by virtue of the width of the boundary being constant. Taking that into consideration, we only parallelized the actual integration phase which takes $\mathcal{O}(mn)$ steps to complete.

Each integration step consists of four advection steps, followed by four diffusion steps, which are then followed by computing a linear combination of the vector fields. Every step takes $\mathcal{O}(mn)$ to execute. Computing the linear combination is very likely going to be memory bound, i.e. bottlenecked by the throughput of the memory bus; in addition, it only contributes to 9.4% of the time spent within the integration routine, as compared to the 63.5% by the advection and the 27.1% by the diffusion (with this difference stemming from the fact that the advection pass employs two stencils of size 7, whereas the diffusion pass only one of size 5). For these two reasons, we elected not to parallelize the linear combination computation.

As part of each of the four advection steps and the four diffusion steps, henceforth called the eight stencil steps, we perform a two-dimensional iteration over one or two matrices, evaluate a stencil-based formula and write back the result into a separate output matrix. The parallelization is achieved by splitting up the work load of the two-dimensional iterations onto different threads. The threads are synchronized at the end of every stencil step in order to prevent race conditions. The distribution of tasks onto the threads can be performed in four different ways, all of which we have implemented and investigated for performance differences.

The different distribution strategies are $\{\textit{shallow}, \textit{collapsed}\} \times \{\textit{static scheduling}, \textit{dynamic scheduling}\}$, where *shallow* stands for the strategy of only distributing the outer loop while the inner loop is always executed as a whole by the same thread, *collapsed* stands for the strategy of fusing the two nested loops together and treating them as one loop for the sake of distribution and lastly *static scheduling* and *dynamic scheduling* refer to whether the tasks are allocated prior to the execution in a rigid manner or flexibly during the execution. Our prediction is that a *shallow* strategy employing *static scheduling* is going to be the most performant.

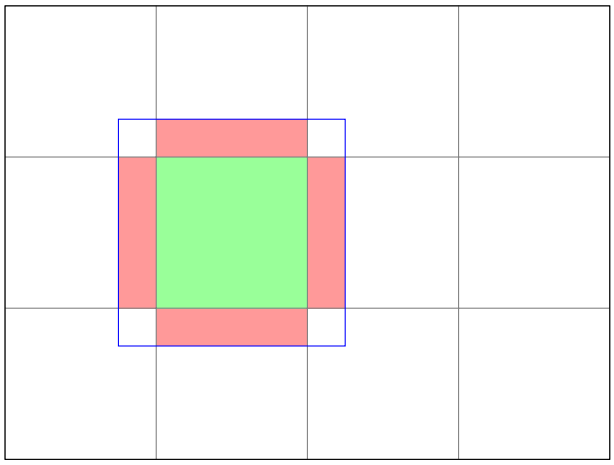

Figure 1: Distributed memory layout for 12 ranks arranged in a 4x3 grid

3 Distributed Memory Parallelism

A distributed memory parallelization is implemented using the Partitioned Global Address Space library UPC++. The two-dimensional grid representing $\mathbf{v}$ is split vertically and horizontally such that every rank is assigned exactly one rectangular matrix block. The matrix is padded in all directions by ghost cells matching the radius of the applied stencils. This is visualized by the red border around the assigned block in green in Figure 1. The blue border marks the effective size of the matrix assigned to each rank. Since it is not always possible to evenly divide the grid representing $\mathbf{v}$, the blocks of the bottom- and rightmost ranks can be slightly smaller than the rest.

During the initialization, the given initial solution is distributed from the first rank to all other ranks. The computation of each time step is split into two parts: communication and stencil computation. Before switching between these steps, all ranks are synchronized. During the communication, each rank asynchronously requests all four borders from its neighbouring ranks. If a rank does not have a neighbour in a particular direction, it instead applies the boundary condition on the corresponding border. The computation applies the advection and diffusion stencils on the matrix block assigned to the current rank, only reading from but not modifying any of the ghost cells. When using an analytical solution for verification, the error is calculated on each rank separately and the sum is logarithmically reduced to the first rank.

4 Results

The following results were obtained on an AMD Ryzen 3900X 12-core CPU with support for hyperthreading. A benchmarking algorithm automatically determined the number of runs on the fly, but at minimum 20 runs were performed per data point.

4.1 Shared Memory Parallelism

As can be observed in Figure 2, the program using the shallow strategy and static scheduling has as predicted come out on top. On the flip side, the program using collapsing and dynamic scheduling was unusually slow compared to the other four programs, so we omitted it from the plot. Its runtimes were in the range of 40s to 130s making it two orders of magnitudes slower.

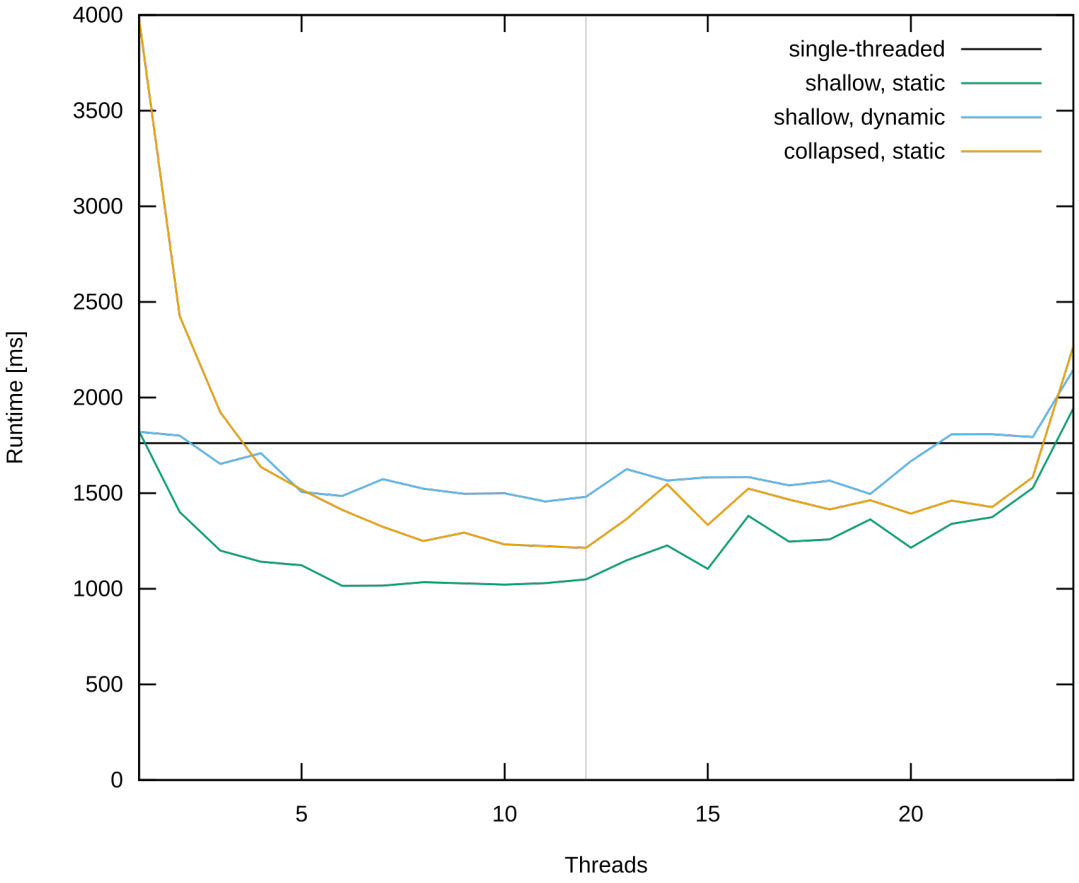

Figure 2: Runtime of the OpenMP implementation on a grid of size 480x480 running 229 iterations

Despite us utilizing a 12-core machine, we have not obtained a speedup in the vicinity of 12. This is in parts owed to the fact that not the entire main loop body was parallelized (cf. section 2) but more importantly to the fact that at a certain point the computing power is superseded by the memory's data throughput as the role of the limiting factor. There will always be a hard limit, defined by the memory bus, to how fast one can solve a problem with a single shared memory, irrespective of the number of cores.

A consistent uptick across all parallelized implementations can be observed at the transition from 12 to 13 threads (indicated by the vertical gray line at exactly 12 threads); this observation is linked to the fact that the CPU in question has 12 physical cores and 24 logical cores. The two hyper-threaded

logical cores of a single physical core still share a lot of infrastructure as compared to two separate physical cores. At 12 threads the program was already saturated with computing power and therefore further threads were merely additionally incurred scheduling overhead with no performance benefits. These two facts combined explain the uptick.

4.2 Distributed Memory Parallelism

We benchmarked the distributed memory parallelization on two different grid sizes, 480x480 in Figure 3 and 48x48 in Figure 4. The number of iterations was adjusted such that both cases have similar computational cost. We tested two different partitioning strategies: linear partitioning (1x1, 1x2, 1x4, 1x8, 1x12) and square partitioning (1x1, 2x2, 3x2, 3x3, 4x3). For 12 threads, the speedups were as follows:

Grid Size	Partitioning	Speedup
48x48	12x1	3.24x
48x48	4x3	4.45x
480x480	12x1	7.41x
480x480	4x3	8.61x

As expected, a smaller grid size increases the communication overhead between each thread's individual memory allocation. A partitioning with a high aspect ratio increases communication overhead further, since it doesn't minimize the number of ghost cells that have to be transmitted on each halo update.

5 Verification

To verify the correctness of our implementation, it was tested against the Python implementation [1] on some sample cases. The numerical solution is also compared to the given analytical solution.

6 Future Work

The distributed memory implementation currently doesn't run any computation while waiting for communication. Instead it would be possible to continue computation for the center of the assigned block that is not affected by the ghost cells while awaiting the ghost cell update.

References

[1] S. Ubbiali, "A numerical model of the two-dimensional viscid Burgers' equations," https://github.com/GridTools/gt4py/blob/master/examples/demo_burgers.ipynb, 2019.

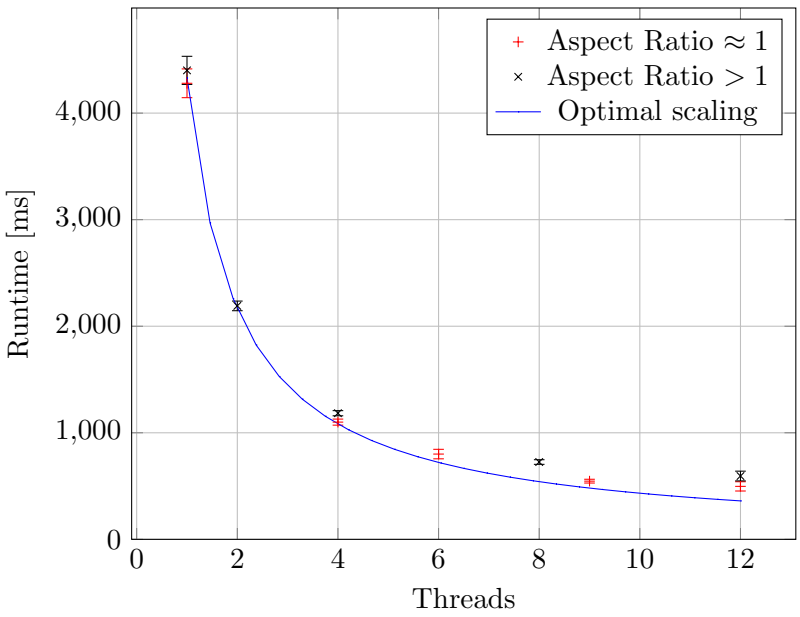

Figure 3: Strong scaling of UPC++ implementation on a grid of size 480x480 running ∼230 iterations.

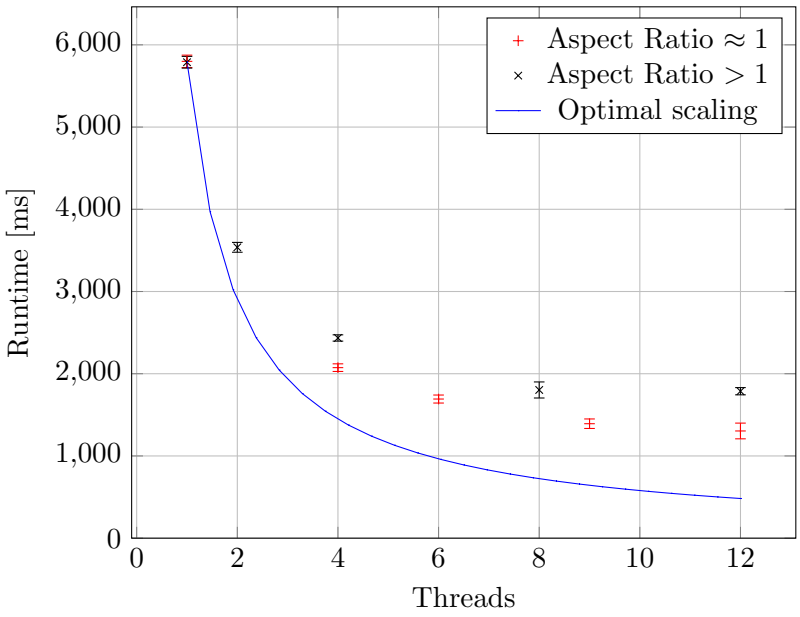

Figure 4: Strong scaling of UPC++ implementation on a grid of size 48x48 running ∼22k iterations.