Solving the Wave Equation using Concurrency

Domain Decomposition, MPI, and Finite Difference Stencils

Akhil Krishna Mohan

Github

Feb 22, 2022

Motivation

The wave equation is one of the most fundamental PDEs in Mathematics and Physics. It is a second-order linear PDE that describes the propagation of waves, and it looks something like this -

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \tag{1}$$

where u represents displacement, c represents the speed of the wave, and ∇ is the Del or Nabla operator for vector functions.

In this report, we document our use of the C++ MPI (Message Passing Interface) library to solve the 2D wave equation on a finite grid domain using concurrency. The application of concurrent programming to this problem is a good means to examine the practicality of the approach, as well as to quantify the nature of efficiency gained with increase in processing power.

Numerical Model

To begin with, we must note that equation (1) alone does not define a unique problem. To solve the wave equation on a given domain, we must also specify an initial condition and a boundary condition. For most of our analysis, we use a point disturbance as an initial condition, centered at coordinates $(3, 3)$. We implement 3 different kinds of boundary conditions - Dirichlet, Neumann, and Periodic boundary conditions.

We use a Finite Difference (FD) stencil to discretize our domain. This gives us a means to solve the PDE as an ODE. We define a fine grid over our domain, using function values at adjacent points to calculate derivatives. The FD version of equation (1) (in 2D) looks like this -

$$\frac{u_{i,j}^{n+1} + u_{i,j}^{n-1} - 2u_{i,j}^{n}}{\Delta t^2} = c^2 \left(\frac{u_{i,j+1} + u_{i,j-1} - 2u_{i,j}}{\Delta x^2} + \frac{u_{i+1,j} + u_{i-1,j} - 2u_{i,j}}{\Delta y^2} \right) \tag{2}$$

where u is the value of displacement at the point at row i and column j of the grid, $\Delta t, \Delta x, \Delta y$ are small increments in time, the x-direction, and the y-direction respectively, and the superscript n is the time step of concern (i.e. $u_{i,j}^n$ represents the function value at grid point i, j at time $t = t_0 + n\Delta t$). This can be re-arranged to give an explicit formula for the value of u at a later time-step based on values from the previous time-steps, namely -

$$u_{i,j}^{n+1} = c^2\Delta t^2 \left(\frac{u_{i,j+1} + u_{i,j-1} - 2u_{i,j}}{\Delta x^2} + \frac{u_{i+1,j} + u_{i-1,j} - 2u_{i,j}}{\Delta y^2} \right) + 2u_{i,j}^n - u_{i,j}^{n-1} \tag{3}$$

Users of the software can configure the domain size, the values of all parameters $\Delta x, \Delta y, c$, initial conditions, and boundary conditions. They also specify an interval Δt at which the code will output its plots. This, however, is a different value from that of Δt used in the above calculations. For numerical stability of the model, the time-step Δt must satisfy the CFL condition

$$\Delta t << \frac{\min(\Delta x, \Delta y)}{c}$$

This is used to define an internally used time-step for the iteration.

Concurrent Model

The idea of concurrent programming is to allow identical copies of one program to execute simultaneously on multiple CPUs or cores in a machine. This, however, necessitates a means of these seemingly independent processes to communicate with one another. This is the exact functionality that an MPI library provides.

For the wave equation, we chose to divide the domain into rectangular sub-domains, each assigned to an independent process. Each process can then work on a largely independent subset of the larger grid, allowing for efficient scaling with additional cores. As seen from equations (2) and (3), the function values at a given point depend on adjacent points in the x and y directions. At sub-domain boundaries, this necessitates the need for inter-process communication. We can see, from the figure, that while we must wait on information from other processes for boundary values, the internal calculations for each process can be done independently. This dictates our use of non-blocking communication, and doing the inner grid iteration while we wait for the information at sub-domain boundaries.

For example, based on Figure 1, with a non-periodic boundary, process 0 would only ever communicate with processes 1 and 3.

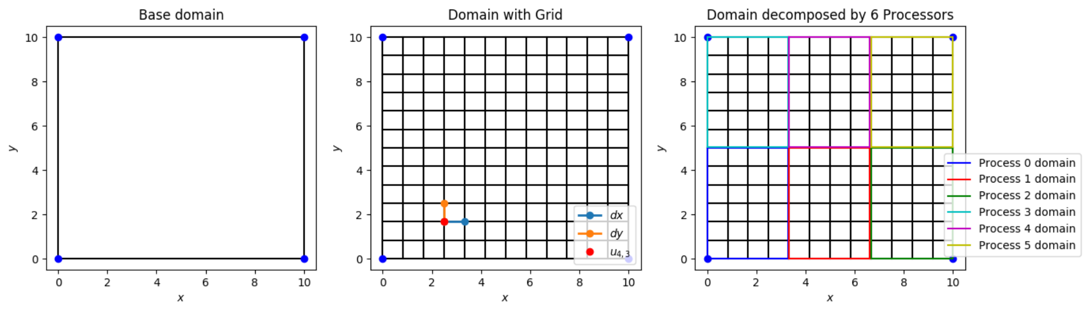

Figure 1: Domain Decomposition of a sample domain with 6 *processes*

Visualizing Results

The outputs of the program are text files containing a matrix grid of u-values. To avoid issues with file-handling in parallel, each process write to its own files for each iteration. A Python script, which can be found under the `tools/` directory of the Github repository then collates these grids for each iteration and generates images describing the evolution of the equation. A bash script is also included in the repository which allows a user to input a file to dictate the parameters of the program, compile, run the code with a given number of processes, and visualize results of a given run. Animated GIFs made via this process can be found in the `images/` directory.

Performance Analysis

The code was tested using a High-Performance Computing (HPC) system, with each benchmark time being run 5 times and execution time averaged between those runs. The inputs to the program were all the same - a square domain spanning from $(0, 0)$ to $(10, 10)$, $\Delta x = \Delta y = 0.025$, and a point disturbance centered at $(3, 3)$. Δt was 0.0025, and the code was run, stepping from $t_0 = 0$ to $t_n = 30.0$. This gives us a $400x400$ grid to do computations on, over 12000 time steps. All outputs from runs on the HPC system are found in the Github Repository under the `hpc_outputs/` directory.

Despite running the code multiple times with the same number of cores and processes, there was considerable variance in run-time performance. We suspect that this is due to inconsistencies between machines in the HPC facility, as well as the load on said machines. Nonetheless, the results are summarized in Table 1. Speedup is the ratio of time taken on N cores vs that on 1 core, while efficiency is the speedup / N.

Table 1: Time taken (s) vs Number of cores

Number of Systems	Number of cores	Time for solve	Speedup	Efficiency
1	1	40.619	1	1
1	2	21.991	1.84	0.92
1	4	13.927	2.91	0.73
1	8	6.134	6.62	0.83
1	16	3.913	10.38	0.65
1	32	1.758	23.11	0.72
2	32	2.130	19.06	0.60
2	64	1.364	29.79	0.47
4	128	0.767	52.92	0.41

We see a steep drop-off in efficiency early on, and this is to be expected as we move from serial to concurrent code (1 to 2 cores). Our efficiency does not seem to improve for the most part as we add more cores, but this is, we believe within the margin of error for the inconsistent timings of HPC runs. In terms of testing philosophy, a finer grid would probably see higher values of efficiency, as each process would be responsible for more values in the sub-domain. That being said, from this data, it looks like the sweet spot for efficiency is using all 32 cores on a given machine, giving us a hefty 23x speedup and 0.72 efficiency. We see more drop-off in efficiency as we have more than one machine in use, as the cache locality is lost when we make this step up. Figure 2 plots the scaling efficiency vs the number of processes in use. Despite the efficiency being far from ideal, the near 53x speedup on using 128 cores shows the real potential of concurrency when applied to this problem. Balancing the efficiency with the speedup will remain as much art as it is science, but in the real world, practicalities make that decision for us.

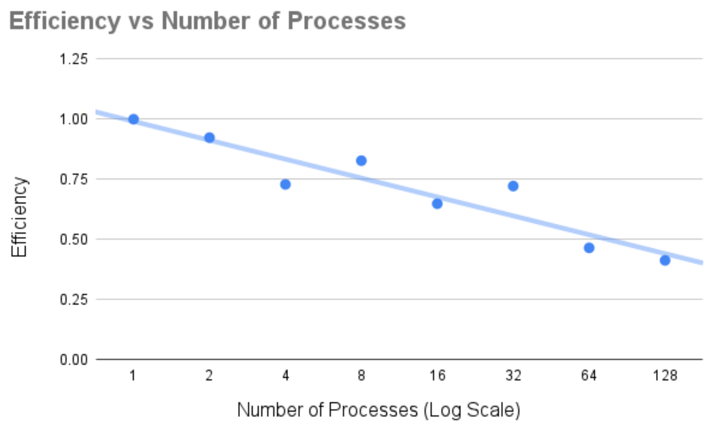

Figure 2: Scaling efficiency curve

Conclusion

Overall, concurrent programming is a versatile tool that will always have its place in computational science. As Moore's Law reaches its limits, the only way to increase the efficiency of computations without drawing ever-increasing amounts of power is to run programs concurrently. While the problem approached in this report is far from the most daunting, this provides a useful blueprint towards sustainable and scalable computing, not only for solving PDEs, but a wide variety of problems.