Parallelized PDE Solver Using Adaptive Meshing and User Supplied Domain and Boundary Conditions

Laurel Koenig, Isaac Shirk

Abstract

Numerical methods for computing solutions to partial differential equations (PDEs) have been well developed over the years due to their being ubiquitous in nearly all fields of science. Not matter the field, be it manufacturing, to rocket science, PDEs need to be computed accurately and efficiently. However, numerically solving PDEs in two and three dimensions typically incurs a great computational cost, leading to impractically long computation times. This project aims to create a tool that quickly and accurately computes numerical solutions to PDEs on domains with boundary conditions that are user-input, allowing flexibility in its application. It will combine parallel algorithms and techniques with adaptive meshing to minimize error while also maintaining computational efficiency and speedy performance.

I. Introduction

THE Focus of this project is to produce a tool that can produce solutions to PDEs on domains with boundary conditions provided by users. The domains include rectangular and circular two dimensional areas, with edge conditions being set to constant values, or a constant derivative. The user defines the shape of the domain, the boundary conditions, and the initial state, and the solver will then compute the solution for some time in the future.

Often, when numerically solving PDEs for physical applications, a very finely discretized spatial grid is necessary to obtain small errors. However, increased grid resolution comes at the cost of disproportionately more computational cost. For example, in order to increase the resolution of the grid by two, the number of grid cells increases by four in two dimensions, and by eight in three dimensions. For large or complicated domains, this results in a PDE that is very expensive to solve. The techniques we bring to bear are adaptive mesh refinement and parallelization. Adaptive mesh refinement allows for starting with a relatively coarse grid for simulation, then using successively finer grids in areas that need more precision. The idea of variable density meshes is widely used in research [**1**], and is used in many commercial products like Comsol [**5**] and Matlab [**4**]. This way the extra computation time is used only where it is needed. The use of parallel algorithms to solve PDEs has been proven useful in research [**2**][**3**], will further speed up the computation times of our tool with no cost to accuracy.

II. Methodology

While many simple PDEs have analytical solutions more complicated PDEs require numerical solutions.

A. Finite Difference Method

The finite difference method is one of the simplest numerical PDE methods. This method aims to approximate derivatives over a short step, or difference, to get to an approximate solution. It divides the domain into a grid of discrete spatial points with a fixed distance between them. It then uses the distance between them as the difference needed to estimate derivatives.

The three most common variations are:

- Explicit Method
- Implicit Method
- Crank-Nicholson Method

B. Finite Element Method

The finite element method, similarly to the finite difference method, divides the domain into a collection of small elements that are easier to solve. This methods, however, uses a triangular mesh rather than a square grid which makes it compatible with irregular spaces.

C. Multigrid Method

The multigrid method uses the solutions from course grids to augment the solution on a finer grid. This increases the convergence of a solution. This method can be paired with the finite elemnet method as well.

D. Adaptive Grid

The rectangular grid of the finite difference method results in some unique challenges. Namely, curved or irregular boundaries can be difficult to discretize with a small error. If a boundary moves over time that can result in further error in due to the discretization errors. Traditionally the simplest solution to these problems is making the mesh much finer. Of course, a fine mesh leads to it's own complications. In higher dimensions we will see a n^2 or n^3 increase in computational time. It's also rather inefficient to use a fine mesh everywhere if most of the discretization errors occur in a single part of the grid.

Thus the proposed solution to this is a mesh grid that isn't the same resolution throughout. In areas where errors are occurring the grid resolution can be raised for more precision without having to do extra calculations in places where the lower resolution has acceptable error. This will better represent curves and changing boundaries will saving time and other computational resources.

III. Project Overview

The finite difference method was chosen because the rectangular grid is simpler to break into pieces for parallel computation than the irregular grid of the finite element method.

General flow of solution:

1) Get input of what the domain is. Basic rectangular, or a composition of rectangular shapes.
2) Get the boundary conditions at each section of the mesh.
3) Uniformly mesh the area, starting with some relative coarse mesh.
4) Set a maximum allowed discretization error
   One way to do this is to find the maximum approximation error of the refined mash, and any approx error on the coarse mesh higher than that value gets marked for refinement.
5) Evaluate approx discretization error at the nodes on the mesh. If they are above the threshold, then mark the node/edge for refinement.
6) Refine the marked edges. Can do this by expanding the grid. Multiple methods for this exist. I think for a rectangular mesh, we can just double the mesh precision.
7) Reevaluate on the refined grid to see if there are any points that still exceed the error threshold. re-expand.

Once you've got your mesh expanded, you need to break the mesh into chunks and figure out how to pass each chunk to a thread/process to evaluate. It also needs to communicate with the other processes.

We have options here. If we make a copy of the grid, then we just flip flop between them using the values from one to compute new values for the other, and we have local threads using the same memory, then as long as we sync up the threads at each timestep, ie waiting till all sections of the grid are computed before starting the next time step, then we don't have to worry about message passing between them.

If our mesh ends up being very large and taking up a large chunk of memory, especially with mesh refinement, then we shouldn't make a copy. Then we would have each thread have a chunk, and an edge that depends on the work of some other thread. It would have to request the information from the other thread to simulate its chunk, because it needs the old values before they get updated by the thread.

References

[1] Sakane, S., Takaki, T., and Aoki, T. (2022). Parallel-GPU-accelerated adaptive mesh refinement for three-dimensional phase-field simulation of dendritic growth during solidification of binary alloy. Materials Theory, 6(1). https://doi.org/10.1186/s41313-021-00033-5

[2] Solve a 2D Heat Equation Using Data Parallel C++. Intel. (2022). Retrieved 2 September 2022, from https://www.intel.com/content/www/us/en/developer/articles/technical/solve-a-2d-heat-equation-using-data-parallel-c.html#gs.b31uc2.

[3] Horak, Verena & Gruber, Peter. (2005). Parallel Numerical Solution of 2-D Heat Equation. 3. 47-56.

[4] Create adaptive 2-D mesh and solve PDE. MathWorks. (2022). Retrieved 2 September 2022, from https://www.mathworks.com/help/pde/ug/adaptmesh.html.

[5] Preney, T., Wheeler, J.D., & Namy, P. (2018). Adaptive mesh refinement: Quantitative computation of a rising bubble using COMSOL Multiphysics®.

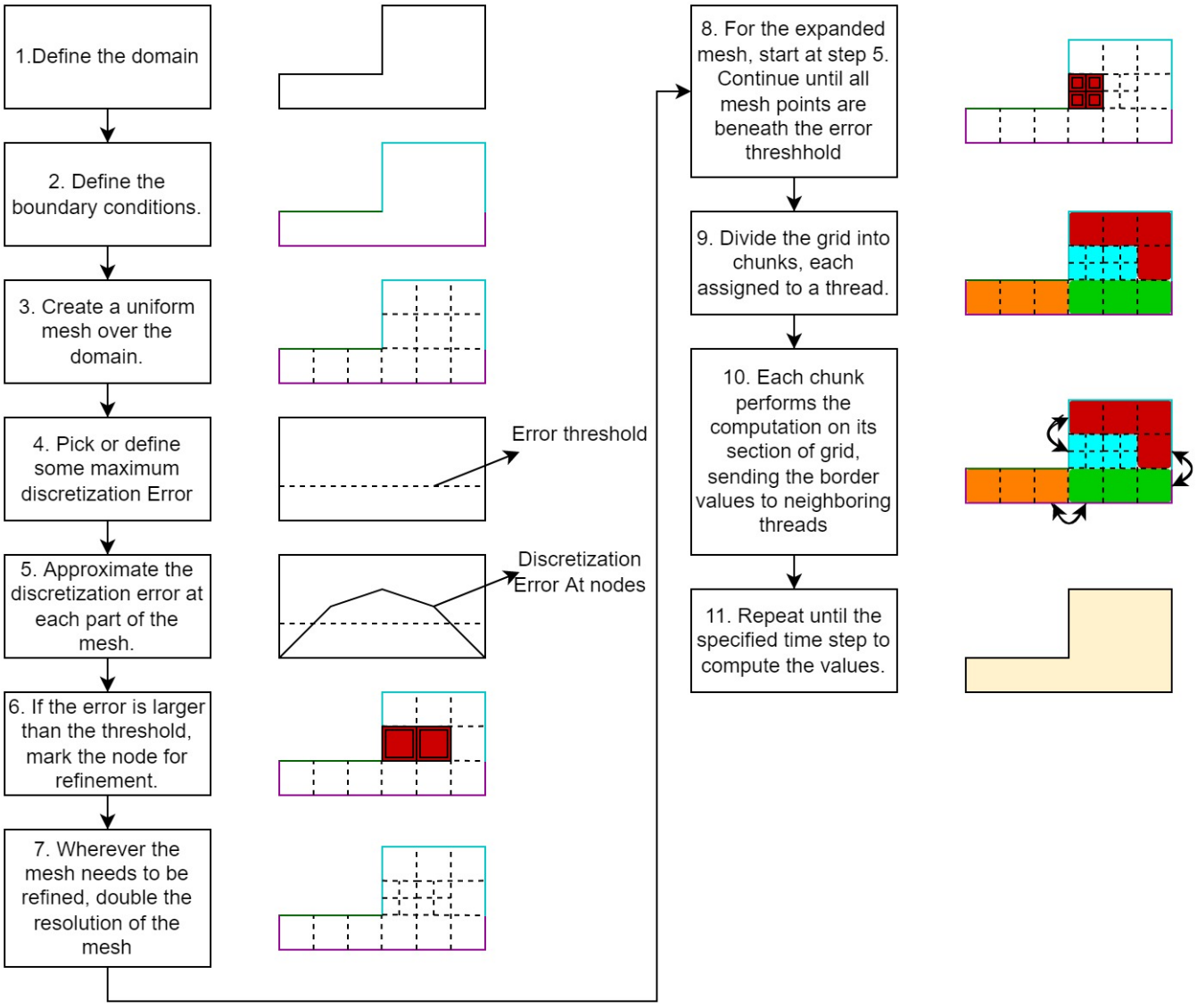

Fig. 1: Caption