**Accelerating PDE Approximations Through Parallel Architectures**

**Peter Wu**

Carnegie Mellon University

hwu3@andrew.cmu.edu

# Abstract

Partial differential equations can be approximated by finite element method, a process often highly parallelizable due to minimal sharing only among neighboring elements. This project utilizes SIMD vectorized intrinsic, message passing, CUDA threads, and OpenMP tasks to explore the difference in their performance caused by different choices of step sizes and attempts to combine different schemes to derive an optimized algorithm.

# Introduction

Partial differential equations are used to describe most phenomena in our physical world, modeling from water and electricity flow to stock prices, creating airplanes to 3D animation. Unfortunately, most PDEs cannot be solved precisely. The huge demand for solving them despite this impossibility marks the importance of efficiently approximating the solutions, and the most popular method for this approximation is finite element method.

Finite element method relies on the function being smooth enough (e.g. twice differentiable or continuous), a condition usually satisfied by real world functions. Using this assumption, we can approximate the function at any given point only using nearby points, without having to know what the function is.

The simplest example is the Euler method, which solves a first-degree ordinary differential equation given the initial conditions. For instance, for the ODE we are given, and we want to solve for. First, we must determine a step size, a small interval in which we assume that the function is linear. Larger step sizes often result in faster computation, whereas smaller ones are often more accurate. In fact, the precise solution at the given point is the limit of these results with decreasing step sizes. Take. Then we can approximateby our linear assumption, and the update for the derivative can be derived from our ODE: . Here is the table of how different choices of step size result in different accuracies and time:

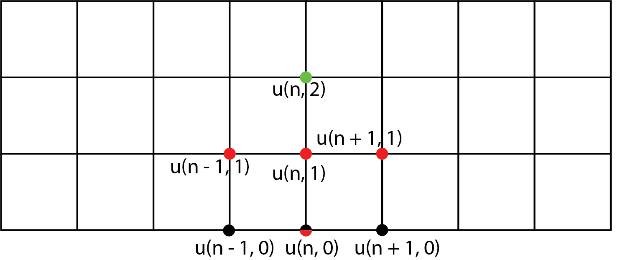
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **f(h=1)** | **f(h=0.1)** | **f(h=0.01)** | **f(real)** |
| 0 | 2 | 2 | 2 | 2 |
| 1 | 2 | 2.34001 | 2.34936 | 2.35039 |
| 2 | 3.71828 | 3.49042 | 3.51181 | 3.51414 |
| 3 | 4.95249 | 6.62397 | 6.68177 | 6.68806 |
| 4 | 15.1806 | 15.142 | 15.2991 | 15.3162 |
| 5 | 29.237 | 38.2964 | 38.7235 | 38.77 |
| 6 | 94.9391 | 101.237 | 102.398 | 102.524 |
| 7 | 218.551 | 272.326 | 275.482 | 275.825 |
| 8 | 664.532 | 737.395 | 745.973 | 746.906 |
| 9 | 1656.89 | 2001.58 | 2024.9 | 2027.44 |
| 10 | 4794.3 | 5438.01 | 5501.39 | 5508.28 |
| Time\*(microsec) | 1002 | 4002 | 58001 |  |

\*This is for deriving, as smaller inputs report negligible execution times.

Extending this method to ODEs of higher degree is also straightforward: we update the function value with all its derivatives simultaneously. While this can be optimized via parallel computation, commonly used ODEs rarely have degrees high enough that parallelism is needed: note that the number of computations that can be executed in parallel is only approximately the degree of ODE. The iterative method for updating function values in ODEs is inherently sequential because of high dependency of updated values.

Partial differential equations, however, have a much larger potential to be optimized, due to the multiple dimensions along which data updates are usually independent. Consider the simplest example of one-dimensional wave equations, where we havein some interval based onand, and initial conditions, boundary conditions, or their mixture.

It is important to recognize that distinct types of conditions often lead to different approaches when we approximate. For 1D wave equations, we are given the initial values and corresponding derivatives in one dimension, and the boundary values of the same dimension, so we can simply parallelize updates on that dimension. If, for example, we have, we can draw a grid, and easily observe that we can simultaneously update all thevalues for different values. For simplicity, all step sizes in the graph are 1.



We obtainby approximating with its lower neighbor,, and upper nodes can be derived by solving to get, where.

Note that each red node only depends on the black node immediately below, and that each green node only reads, not modify, the four corresponding red nodes. These features make concurrent data accesses much more manageable. This paper specifically tackles an 1D wave equation to demonstrate the impact of problem parameters, which in this case are just step sizes in time and space, on the speedup of different parallel architectures.

# Background

## Problem Statement

, ,

Note thatandare both independent of other points, so they do not require any communication as inputs. Therefore, we trivialize the terms to simplify the problem, understanding that the choice does not affect the speedups. To ensure a numerically stable solution, we need to satisfy the CFL condition that. In our choices for pairs (), therefore, we need, or.

## Sequential Algorithm

We split our algorithms into three stages: (1) initializing values onaxis; (2) calculating second row values based onand; (3) iterating the “cross” pattern calculation described in Introduction. While combining the stages into a single function should be more efficient, we deliberately put them into sections to allow for modifications by clients without changing solution structures.

1. Initialization:

init(float array, float array):

for (long i = 0 ... ):

 = sin(i ),  = cos(i )

1. Second row:

second(float array, float array):

for (long i = 0 ... ):

 = +

1. Iteration:

iterate(float array, float array:

for (long i = 0 ... ):

if (i == 0): = (2–2s2) + s2 -

elseif (i == :

= (2–2s2) + s2 -

else: = (2–2s2) + s2

-

For the main function, we have:

main():

create arraysand

sumTime = 0

for (int n = 0 ... numTests):

start timer

init()

second()

for (long t = 0 ... maxT):

iterate(

stop timer

sumTime += time passed

report (sumTime / numTests)

optional: report result matrix

The bottleneck of this workload is obviously theupdates of the input arrays. For each entry, each update requires the value of current entry for the past two iterations, as well as the values of neighboring entries of the last iteration.

A picture containing doll, toy, mask, vector graphics

Description automatically generated

This graph demonstrates the sharing occurring in our iterations. To update a current entry, we are givenand , but to get, we also need to take in the red nodes from other threads/tasks. This minimal sharing is inherent to finite element method as communication only appears with nearby nodes.

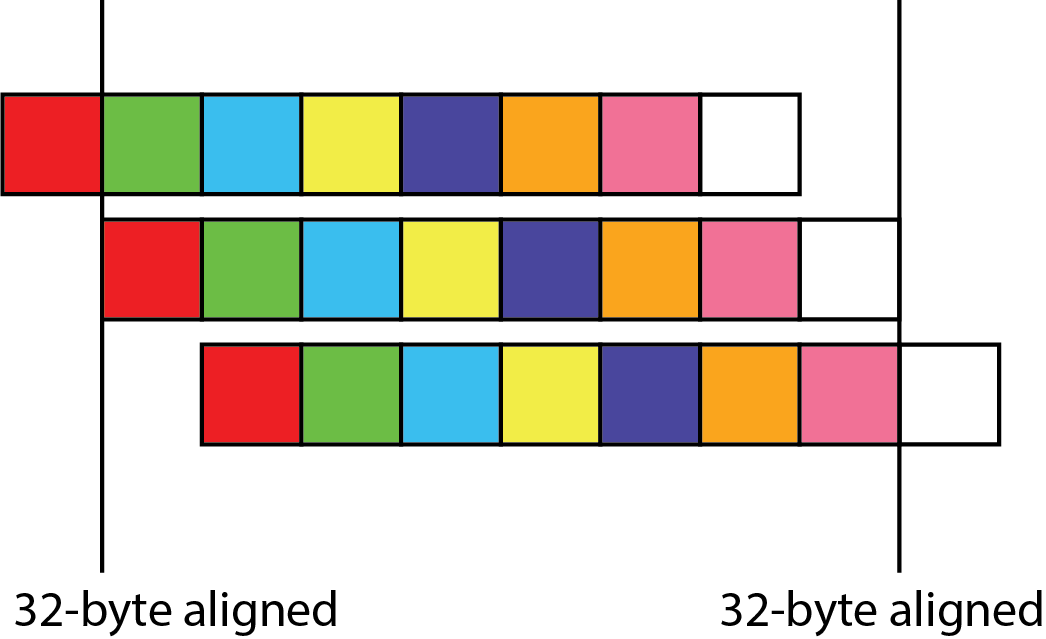
# Approaches

After implementing the sequential algorithm, we explore how different parallel computing schemes improve its performance and attempt to combine these schemes to achieve optimal performance.

## SIMD

We first convert the initialization and second row for loops, both of which are data-parallel, into SIMD trivially.

Moving on, we attempt to translate the updates, requiring the same arithmetic operations for every node except the left-most and right-most ones, into SIMD vector instructions. This seemingly trivial conversion, however, is complicated by the off-by-one sharing in the boundaries for each 8-wide block. Specifically, the boundary computations have to fetch data from nearby blocks and handling them separately disrupts our original goal of using same instructions for all data, thus middle elements have to wait idly for boundary values to be computed. To avoid this, we load three blocks of data, recognizing that the previous block should still be in cache, and the next cache-line have to be loaded because we have to use the first element in the next block. This also effectively prefetches data for next iteration.



Again, although we may seem to have wasted memory access in loading the same central elements three times, we only fetch that cache-line once. Now that we have corresponding data aligned across the three blocks, all our instructions can indeed by conducted in parallel. All other alignment issues in stack and heap are resolved via padding. The leftmost and rightmost blocks are managed separately by sequential for loops, as we cannot load data outside the boundaries.

## CUDA

With large array operations on CUDA, the most natural scheme is to divide the array into blocks of threads sharing common memory within each block. If we put the read-only input into shared memory, we can avoid going to global memory three times for each entry, instead only reading global memory once, then read from block-local memory twice afterwards. This significantly improves total stall for memory access.

Copying global memory into shared memory can be parallelized by each node writing one entry, plus the two boundary elements that can be managed by any single thread. After copying, we can just operate on shared memory within each block. It is important to note, however, that we are allocating extra space equal to the original data size, which should not be a problem for 1D array updates but might be expensive for higher dimensions.

CUDA threads work only on GPUs.

A picture containing object

Description automatically generated

## MPI

The minimal amount of sharing indicates that message passing should be efficient, as processors can execute instructions independently while only waiting for very few short messages on the boundaries. Each processor initializes and updates its own piece of data, communicating only if necessary, and even then, attempts to hide the latency behind computations by calling receive as late as possible. Specifically, each processor follows the workflow of asynchronous sending boundary data 🡪 computing middle elements 🡪 synchronous receiving and updating boundary data.

A picture containing object, clock

Description automatically generated

\*note that Isend = non-blocking send; Receive = blocking receive

One problem with the naïve scheme is that Isend uses a message buffer that cannot be released manually, so even after implementing Waitall to wait for all messages to finish transmission every chunk, with large data sizes Isend still frequently run out of memory. This issue, if not resolved, can be the largest drawback of MPI.

MPI works best with multiple clusters.

## OpenMP Tasks

OpenMP dynamically assigns threads to processors, which improves worker balance. The omp pragmas hides implementation details, so we do not need to determine the actual workflow ourselves. This is because our loops have the same predetermined number of iterations for all workers, and each iteration has roughly the same work.

OpenMP works mostly on multi-thread single nodes.

## SIMD+

SIMD instructions can be integrated into both MPI and OpenMP to accelerate workflow. The integration into MPI is trivial, while we are still investigating errors in integration with OpenMP. Integration with CUDA seems impossible, as SIMD instructions are host code, unable to be called in device code.

## Notes on Runtime Environment

The biggest bottleneck of this project has been setting up the runtime environments. We originally wanted to target multiple machines with both CPUs and GPUs in a cluster, which would carry out correct speedup graphs for our algorithms, but due to the lack of information we have on running our programs on GPUs, and the incompatibility issues with GHC and Latedays clusters, we end up only being able to run our programs on a Windows 10 PC with Intel i7 6-core CPU (12 logical processors) and NVDIA GeForce GTX 1050Ti GPU. Not being able to run code on a cluster also limits our ability to implement an OpenMP + MPI + SIMD scheme, which in theory should be the most efficient approach.

# Results

Our input parameters in experiments are:,,, number of processors for MPI and OpenMP, and block size for CUDA. We select a representative graph to demonstrate time costs of different parameters with different schemes.

In this graph, OMP-t stands for OpenMP with t threads; MPI(SIMD)-n-m stands for MPI (with SIMD) with n processors and 1 << m chunk size for waiting; CUDA-m stands for CUDA with each block size 1 << m.

There are several patterns worth noting in this graph. CUDA approaches are labeled by shared memory size, as CUDA has inherent shared memory limits, and we can only create more blocks to manage extra data. With relatively small data sizes, creating more tasks doesn’t impact total runtime by much. This may change with sufficiently large data that forces threads to wait for further blocks. CUDA-12 also has the largest speedup with large data sizes.

SIMD also creates speedup, though not close to 8x as expected. This may be caused by extra instructions and loading neighbor cache-lines. Its real speedup is about 2x-3x.

MPI creates a slightly higher speedup than SIMD, and with larger data sizes has greater speedup, because the amortized cost of additional data decreases with number of data entries. MPI with SIMD is nearly twice as expensive as normal MPI, which should not happen in a cluster with actual separated memory storage.

OpenMP, which should have created large speedup with our 12-thread PC, actually performs similarly to the sequential algorithm. This may be caused by incorrect configurations of our compilers.

# Further Improvements

The project has many areas that can be extended and improved. Several areas include:

1. Further research on configuring OpenMP, implementing OpenMP + SIMD;
2. Research on using MPI in clusters, implementing OpenMP + MPI + SIMD;
3. Research on integrating SIMD with CUDA threads;
4. Larger range of data sizes to demonstrate drawbacks of different approaches;
5. Extension to higher dimensional PDEs.

# References

J. David Logan, *Applied Partial Differential Equations*, Chapter 6: Numerical Computation of Solutions

Stacy Pschenica, *Parallel Algorithms for Solving Partial Differential Equation*s

Tutorials on OpenMP, MS-MPI, CUDA, SIMD

Tutorials on setting up Visual Studio, Intel Compiler, CUDA Compiler