Solving 3D Poisson Equation using High Performance Computing Techniques - APMA 2822B: Final Project

Daniel Healey, daniel_healey@brown.edu
Melvin He, melvin_he@brown.edu
Michelle Liu, michelle_h_liu@brown.edu

December 10, 2024

1 Overview

1.1 Problem

The problem that we decided to investigate was the application of high performance techniques to the 3-D Poisson equation.

The 3D Poisson equation is a partial differential equation (PDE) that describes the relationship between a scalar potential field and the distribution of a source, commonly in the context of electrostatics, gravitational fields, and fluid dynamics.

In 3D, the equation is typically expressed as:

$$\left(\frac{\delta^2}{\delta x^2} + \frac{\delta^2}{\delta y^2} + \frac{\delta^2}{\delta z^2}\right)\phi(x, y, z) = f(x, y, z)$$

Here, f is given, while ϕ is unknown and being solved for.

In our solver, we focused on a specific instance of the 3D Poisson equation, namely where:

$$\phi = \sin(n\pi x)\cos(m\pi y)\sin(k\pi z)$$

$$f = -(k^2 + m^2 + n^2)\pi^2 \sin(n\pi x)\cos(m\pi y)\sin(k\pi z)$$

$$x, y, z \in [0, 1]$$

Although we decided to solve for this specific situation, our code can be easily modified to fit any f and any domain.

The numerical method that we chose to approximate our solution was through the finite differences method with a three-point stencil. In other words, we used following approximations:

$$\frac{\delta^2}{\delta x^2}\phi \approx \frac{1}{(\Delta)^2}\left[\phi(x-\Delta) - 2\phi(x) + \phi(x+\Delta)\right]$$

$$\frac{\delta^2}{\delta y^2}\phi \approx \frac{1}{(\Delta)^2}\left[\phi(y-\Delta) - 2\phi(y) + \phi(y+\Delta)\right]$$

$$\frac{\delta^2}{\delta z^2}\phi \approx \frac{1}{(\Delta)^2}\left[\phi(z-\Delta) - 2\phi(z) + \phi(z+\Delta)\right]$$

ϕ was represented as values at $N \times N \times N$ grid points evenly spaced across the domain, where N was chosen. ϕ was then calculated using an iterative algorithm that updated the values at these grid points based on the approximations above and our given value of f.

This formula for the iterative updates was derived as follows:

$$f(x,y,z) = \left(\frac{\delta^2}{\delta x^2} + \frac{\delta^2}{\delta y^2} + \frac{\delta^2}{\delta z^2}\right)\phi(x,y,z)$$

$$\begin{aligned}
f(x,y,z) = \{ & \\
& \phi(x-\Delta,y,z) - 2\phi(x,y,z) + \phi(x+\Delta,y,z) \\
& + \phi(x,y-\Delta,z) - 2\phi(x,y,z) + \phi(x,y+\Delta,z) \\
& + \phi(x,y,z-\Delta) - 2\phi(x,y,z) + \phi(x,y,z+\Delta) \\
\}/\Delta^2 &
\end{aligned}$$

$$\begin{aligned}
\phi(x,y,z) = \frac{1}{6}\{ & \phi(x-\Delta,y,z) + \phi(x+\Delta,y,z) \\
& + \phi(x,y-\Delta,z) + \phi(x,y+\Delta,z) \\
& + \phi(x,y,z-\Delta) + \phi(x,y,z+\Delta) \\
& - \Delta^2 f(x,y,z)\}
\end{aligned}$$

This leads to our algorithm:

1. Initialize ϕ with Dirichlet boundary conditions (exact values at the boundaries of domain) and zero elsewhere.

2. Update values of ϕ at grid points using the above formula

3. Calculate change between old ϕ and new ϕ for convergence

4. Stop algorithm once we have converged (difference is less than set tolerance)

To gauge the correctness and accuracy of our algorithm, we also calculated the actual error at each intermediate step as well as after the process had converged. This allowed us to be sure that our algorithm was correctly calculating the solution to the 3D Poisson equation. This step would not be done outside of an experimental setting.

2 Shared Memory

For the first approach to parallelizing our problem was with a shared memory model. In this model, we run on multiple processors which have the same shared and coherent view of memory.

In order to implement our solution in this shared memory model, we added the use of OpenMP. This is a framework that allows for easy conversion of sequential C++ code into new code that used the multithreaded shared memory model.

Furthermore, it should be noted that when compiling we used the `-O3` flag, which instructs the compiler to apply aggressive optimizations to the code, aiming to maximize performance.

3 Shared Memory with GPU

After implementing a shared memory approach, we decided to use GPU acceleration for certain tasks. This would allow us to increase the parallelization for these tasks. Specifically, we used HIP which is an API provided by AMD to preform GPU programming in C++.

We specifically focused on using GPU to parallelize the following:

- Initialization
- Updating ϕ
- Calculating convergence/error

We copied two matrices for ϕ (the new and old phi) from the host to the device, and then only copied the error and convergence back from the device to the host. In this manner, we were able to minimize the amount of communication between host and device, allowing us to decrease the amount of memory bandwidth that we used in our program.

Besides the improvements of the three tasks described above, the implementation was roughly the same as original non-GPU shared memory approach.

4 Distributed Memory (MPI)

4.1 Domain Partitioning

One of the primary complexities for our distributed memory approach was partitioning our domain between nodes. This was necessary to decrease the amount of resources and inter-node communication, as each node can work on calculating ϕ for its own partition with minimal information from other nodes.

The approach that we took to partitioning the domain was based off of slicing the domain based on the z-coordinate. Each node is assigned a roughly equal number of $N \times N$ slices on the x-y plane. Each node is assigned at most one more $N \times N$ slice than any other node. This z-coordinate slicing is illustrated in the figure below:

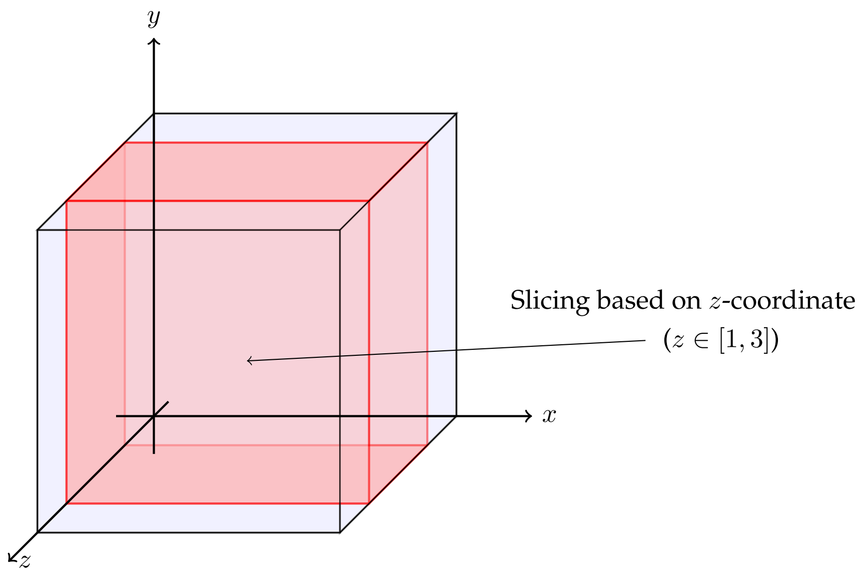

Figure 1: Domain decomposition

By partitioning the domain in this way, we are able to mitigate the amount of memory that is being used by each node. Each node now holds only a small portion of the grid in memory. However, this poses an issue on the borders of the partition, where the grid points do not have data for their neighbors in the iterative update calculation. In order to solve this, we use communication of so-called ”halo points” between neighboring nodes.

These ”halo points” were communicated between two nodes before they began their calculations for updating ϕ. All nodes except for the bottom node (rank 0) communicated their last $N \times N$ grid points to the node above them (one greater than their rank). All nodes except for the top node (rank size-1) communicated their first $N \times N$ grid points to the node below them (one less than their rank). Each node then received these grid points, stored them in a buffer and then used that buffer in updating ϕ.

We determined that, for our architecture, partitioning on z was more efficient over partitioning on x or y. This was because we stored our grid points in contiguous memory, meaning that communication for z-based partitioning could be accomplished using a sequential read, while x-based or y-based could not. This led us to using z-based partitioning of the domain between nodes.

5 Results

We noticed the GPU model had significant overhead and startup time, but at larger N values its average iteration time outperformed everything but shared. In fact, shared performed remarkably well, beating the distributed and GPU models. The distributed model also performed consistently well, with a much lower average iteration time than baseline.

5.1 Configurations

Everything runs with the devel partition on the AMD cluster. The shared was run with 6 threads and compiled with the `-O3` flag. Details on the exact compilation flags are provided in Section 7.1.

5.2 Tables of Iteration Times

In order to be able to compare the different approaches, we gathered some statistics for runs of each of our approaches. Specifically, we measured the following: Total Runtime, Number of Iterations, Average Iteration Runtime, Minimum Iteration Runtime, Maximum Iteration Runtime, and Final Error.

To allow for the model time to even out, we have a 'spin-up' period – notably, we only measure iteration times from the 10th iteration on and this is reflected in our average, min, and max iteration times for the tables below. This is to account for initial iterations acting as an outlier for our averages.

We measure our runtimes in microseconds for consistency and also keep track of the maximum and minimum runtimes for the iteration to show the variance in runtime across iteration. Total time, in our analysis results, counts from before initialization (the very start) until the end of the computations.

The true error reported in the table below is measured by the sum of squared errors between our solution and the actual solution. The convergence threshold is 10^{-6} and is calculated using squared differences between each iteration.

	Baseline	Shared	Distributed	Shared+GPU
Total Time (us)	761	899	368.23	273299
Iterations	37	37	37	37
Average Iteration Time (us)	16	15	10.44	80
Min Iteration Time (us)	16	14	9.80	78
Max Iteration Time (us)	18	17	10.34	83
Error	0.096	0.096	0.097	0.097

Table 1: Iteration Times for N=10

	Baseline	Shared	Distributed	Shared+GPU
Total Time (us)	5518	2646	3163.80	279057
Iterations	85	85	85	85
Average Iteration Time (us)	60	26	38.06	88
Min Iteration Time (us)	59	18	37.07	87
Max Iteration Time (us)	66	60	40.48	91
Error	0.057	0.057	0.057	0.057

Table 2: Iteration Times for N=15

	Baseline	Shared	Distributed	Shared+GPU
Total Time (us)	23456	5431	14942.55	288861
Iterations	153	153	153	153
Average Iteration Time (us)	148	31	98.65	113
Min Iteration Time (us)	146	29	97.31	112
Max Iteration Time (us)	156	46	100.56	117
Error	0.038	0.038	0.039	0.039

Table 3: Iteration Times for N=20

	Baseline	Shared	Distributed	Shared+GPU
Total Time (us)	2280844	333357	1595039.80	1428737
Iterations	935	935	935	935
Average Iteration Time (us)	2430	354	1709.50	1155
Min Iteration Time (us)	2398	345	1660.94	1086
Max Iteration Time (us)	2545	422	2097.82	3429
Error	0.013	0.013	0.013	0.013

Table 4: Iteration Times for N=50

5.3 Error per Iteration

In Figure 2, we show the error per iteration for $N = 15$. The errors are from the shared model, although because all the models share the same update procedure, we expect the error per iteration to be the same or very close. In Figure 3 we show the residual per iteration for $N = 15$ – the sum of squared differences between ϕ and ϕ from the previous iteration.

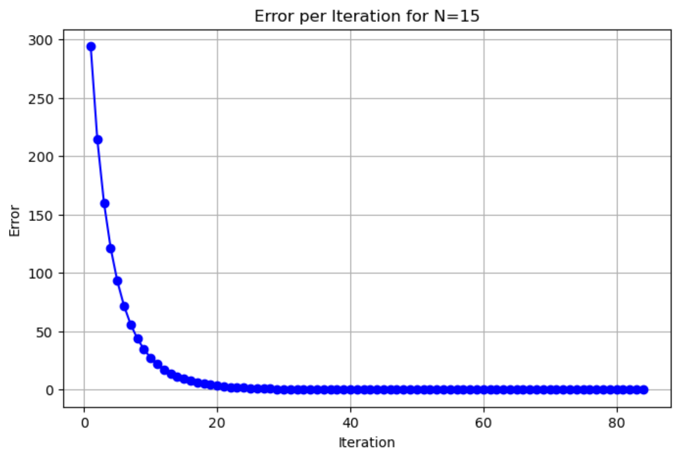

Figure 2: Error per Iteration (N=15)

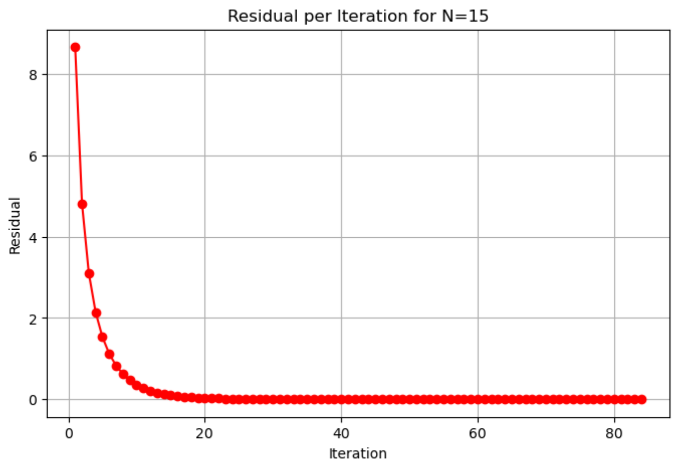

Figure 3: Residual per Iteration (N=15)

5.4 Roofline Model

In `update_phi()`, we do

- 9 floating-point operations: 5 additions, 1 subtraction, 2 multiplications, and 1 division
- 7 reads and 1 write (8-bytes each)

In total, given an N^3 grid, that is $9N^3$ floating-point operations and $8 \cdot 8 \cdot N^3$ bytes of memory reads, where 8 is the number of bytes in a double. The arithmetic intensity is thus $\frac{9}{64}$.

According to information that we computed above, the arithmetic intensity of all of our operations is approximately $9N^3$. From the internet, we believe that the maximum flop-rate and maximum bandwidth are 47.9 TFLOPs and 3.2 TB/s, respectively.

Below, we have plotted the roofline model along with the results for the runs in the tables above:

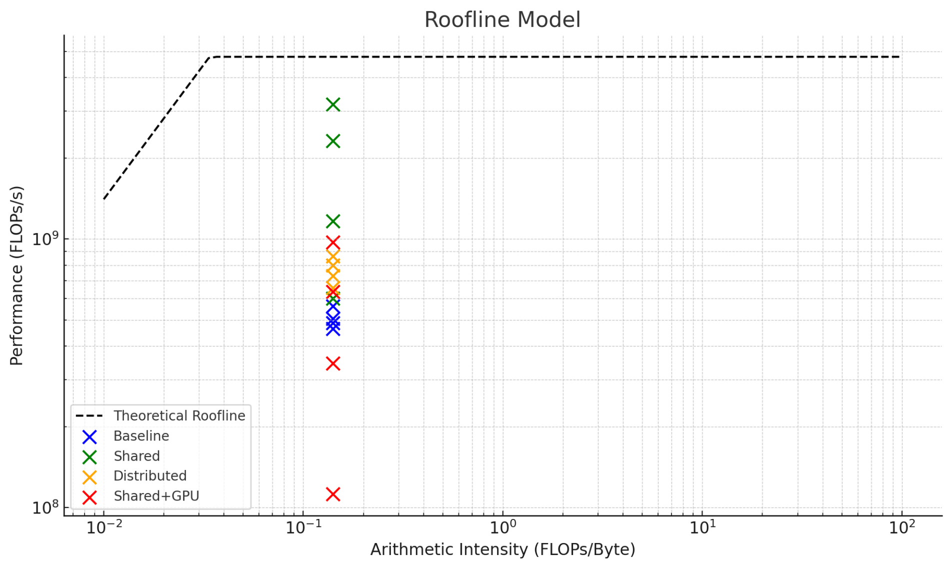

Figure 4: Roofline Model

5.5 GPU Profiler

In order to profile the work being done by the GPU in our shared+GPU implemented we ran a profiler. This showed us a breakdown of the work being done by the GPU during the duration of the run.

We ran the profiler `rocprof` using the following command:

```
rocprof --hsa-trace --hip-trace ./a.out
```

We got some of the results in the following table:

Name	**Calls**	**Total Dur. (ns)**	**Avg. (ns)**	**%**
`hsa_exec_freeze`	2	6664562	3332281	34.79
`hsa_queue_create`	1	5123355	5123355	26.75
`hsa_sig_wait_scacq`	183	2852262	15586	14.89
`hsa_amd_mem_pool_alloc`	16	1236520	77282	6.45
`hsa_amd_mem_async_cpy`	73	1048776	14366	5.47

Table 5: Profiling Data Summary

The profiling results highlight the functions that dominate the execution time in the application. Among the top contributors, `hsa_executable_freeze` accounts for the largest share of runtime at 34.79%. This is associated with freezing the executable and making it ready to run, which did not seem like a viable area of optimization for us. The second highest contributor, which is `hsa_queue_create`, takes up 26.75% of the total runtime. Queue creation is a critical setup step for dispatching GPU tasks, but we only use one queue (the default queue), so this did not seem like a area of optimization for us.

Additionally, `hsa_signal_wait_scacquire` consumes 14.89% of the runtime due to frequent synchronization calls, with 183 invocations showing moderate average latency. This could be a possible source of optimizations, and signals that we might be synchronizing between threads too much. Similarly, memory-related operations such as `hsa_amd_memory_pool_allocate` and `hsa_amd_memory_async_copy` collectively account for approximately 12% of the runtime. These functions are essential for memory allocation and data transfer, which could be an area of optimization.

In all, if we were to further optimize our code, we would focus on optimizing our memory transfers and synchronization based off of our profiler results.

6 Limitations

One major limitation is our distributed model, which does not work properly on multiple threads. For instance, on $N = 20$ with 4 threads, though we get better iteration average times at 41.59 us, our error indicates something is wrong with 137.99 error as opposed to 0.038. Furthermore, none of the models, including the original, can run on over $N = 60$ without a segmentation fault error.

7 Extra

7.1 Compilation Commands

Description	**Command**
Devel partition in AMD	`salloc -p devel -t 00:30:00`
Baseline compilation	`g++ original.cpp -o original.out`
Baseline run	`./original.out`
Shared compilation	`g++ shared.cpp -fopenmp -O3 -o shared.out`
Shared run	`OMP_NUM_THREADS=6 ./shared.out`
Distributed compilation	`mpic++ distributed.cpp -lm -o distributed.out`
Distributed run	`mpirun -np 1 ./distributed.out`
Shared + GPU compilation	`hipcc shared_with_gpu.cu -o shared_gpu.out`
Shared + GPU run	`./shared_gpu.out`

Table 6: Compilation and Execution Commands

8 Resources

ChatGPT (`www.chatgpt.com`) was used to assist in building visuals, although we edited all diagrams significantly for greater fidelity and accuracy. The AMD website was used to find the GPU bandwidth specifications (`www.amd.com/en/products/accelerators/instinct/mi200/mi250x.html`).