1 Rough Draft Area:

In this section we are going to be introducing concepts of high performance computing, just some real basics.
For this case I want to:

- Conceptually explain HPC

- Conceptually explain HPC and relate it to Amdahls Law or any other topic covered.

- The goals

- Find some sort of Analog perhaps add it to each section or have a concurrent example developing for the layman explanation.

- Explain Amdahls Law

- Explain Gustafason's Law

Amdahl's Law

In High performance computing and parallel computing, Amdah'ls Law is a theoretical limit for the achievable acceleration of the computational workload when parellelizing is applied.

Amdahl's Law defines a fundamental constrain imposed by the serial fraction of a program, no matter how manny processors are available. This constraint will be our performance boundary.

Mathematically Amdahl's Law is as follows

The speedup $S(N_p)$ achieved with N_p processors is given by:

$$S(N_p) = \frac{1}{(1 - P) + \frac{P}{N_p}}$$

where:

- $P \in [0, 1]$ is the fraction of the program that is parallelizable,

- $(1 - P)$ is the inherently sequential portion,

- N_p is the number of processors.

As $N_p \to \infty$, the maximum theoretical speedup approaches:

$$S_{\text{max}} = \frac{1}{1 - P}$$

Thus, the sequential fraction $(1 - P)$ becomes the ultimate bottleneck.

Constraints

We are dominated by a sequential restrain, for even a small sequential fraction we will see a hard limit in how much can be scaled.

Example: If 5% of the program is sequential and 95% can be parellelized, the best achievable speedup is obly about 20x.

$$S_{\text{max}} = \frac{1}{1 - P} \Rightarrow \frac{1}{1 - 0.95} \Rightarrow \frac{1}{0.05} = 20$$

$$S_{\text{max}} = \frac{1}{1 - P} \Rightarrow \frac{1}{1 - 0.99} \Rightarrow \frac{1}{0.01} = 100$$

For this reason, adding more processors only has diminishing returns , no matter how many are added. Eventually we experience a sharp decrease on the marginal benefits achieved. This happens once the parallelized portion has been fully exploited.

The strategy of optimizing P, in doing so we can have a reduction in sequention code sections, like serial intialization. This often yields greater gains than simply increassing the count on processors.

Amdahl's Law assumes *strong scaling*, where the problem size remains fixed as the number of processors increases. This makes it a useful tool for determining whether performance limitations are due to hardware constraints or the algorithm's inherent sequential nature.

Amdah's Law being used for strong scaling analysis assume a fixed problem size and no overhead from parallelization, thread creation, communication, and synchronization to name a few.

In the case where the problem size increases along with the processor count, we refer to this as *weak scaling*. Under such conditions, *Gustafson's Law* provides a more realistic model for evaluating performance.

To summarize, Amdahl's Law shows that limitations in performance are not purely hardware-related. Serial bottlenecks impose fundamental constraints, regardless of how many processors are used. Therefore, optimizing parallelization and profiling code to identify sequential sections are essential strategies, extending beyond simply increasing processor count.

2 MARKERIntroduction

Finite difference methods are widely used for numerical solutions of partial differential equations (PDEs). This report investigates a 2D finite difference scheme for computing numerical derivatives on a unit domain, as well as its parallel implementation and performance evaluation on the CARC supercomputing platform.

3 Problem Description

The problem domain is the unit square $\Omega = [0, 1] \times [0, 1]$, where the function $u(x, y)$ is defined as:

$$u(x, y) = \cos((x + 1)^{1/3} + (y + 1)^{1/3}) + \sin((x + 1)^{1/3} + (y + 1)^{1/3}).$$

Dirichlet boundary conditions $g(x, y)$ are defined as:

$$g(x, y) = u(x, y) \quad \text{for } (x, y) \in \partial\Omega.$$

4 Task 1: Serial Implementation and Error Analysis

4.1 Methodology

The finite difference stencil used for approximating the second derivatives is:

$$\frac{1}{h^2} \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

This was implemented using the provided 'poisson.py' file, creating a CSR matrix for efficient calculations.

4.2 Analysis of CSR Matrix Rows

For a grid size of $n \times n$, we analyzed specific matrix rows from the CSR representation of the finite difference stencil to understand how different types of points (corner, domain wall, interior) are represented. The following matrix rows were printed from the terminal output:

```
Matrix row for corner point (0, 0):
Data: [-4.  1.  1.]
Indices: [0 1 5]

Matrix row for edge point (0, 1):
Data: [ 1. -4.  1.  1.]
Indices: [0 1 2 6]

Matrix row for interior point (1, 1):
Data: [ 1.  1. -4.  1.  1.]
Indices: [ 1  5  6  7 11]
```

4.3 Explanation of Differences

- **Corner Point (0, 0)**: This matrix row corresponds to a corner of the grid. It only has three non-zero entries, reflecting the contributions from the central point and its two neighbors (right and below). This is due to fewer adjacent grid points compared to interior points.

- **Domain Wall Point (0, 1)**: This row represents a point on the top boundary but not a corner. It has four non-zero entries, accounting for contributions from the left, right, below, and the central point itself. The presence of an additional neighboring point compared to the corner point explains the extra non-zero value.

- **Interior Point (1, 1)**: This row corresponds to a point within the grid (not on the boundary). It has five non-zero entries, indicating contributions from all four surrounding neighbors (left, right, above, below) and the central point. This full set of adjacent points results

in more non-zero entries compared to edge or corner points.

4.4   Interpretation

The differences between these rows illustrate how boundary conditions and the finite difference stencil shape the matrix's sparsity pattern. Corner points have fewer connections due to being adjacent to only two grid points, edge points have slightly more connections, and interior points have the most due to their central placement within the grid.

5   Results

5.1   Convergence Analysis

We computed the L2 norm of the error between the exact and numerical derivatives, defined as:

$$\|e\|_{L2} = \left(\int_{\Omega} \left(\frac{\partial^2 u}{\partial x^2}\Big|_{\text{approx}} - \frac{\partial^2 u}{\partial x^2}\Big|_{\text{exact}} + \frac{\partial^2 u}{\partial y^2}\Big|_{\text{approx}} - \frac{\partial^2 u}{\partial y^2}\Big|_{\text{exact}} \right)^2 dx\,dy \right)^{1/2}.$$

Figure 1: Log-log plot of error vs. grid size for the serial implementation.

The plot in Figure 1 indicates that the numerical solution converges quadratically, consistent with the expected second-order accuracy of the finite difference method.

6   Task 2: Parallel Implementation and Performance Analysis

6.1   Timing Analysis

Performance metrics were collected both on a local machine and the CARC computing platform.

These preliminary metrics performed locally on multiple thread counts demonstrate performance increase. This is more notably observed at three

thread counts where we are able to see a noticeable increase. The computations being observed in Table 1 and plotted in Figure 2.

Timing (s)	1 Thread	2 Threads	3 Threads
Timing 1	8.5668e-03	8.2240e-03	8.0276e-03
Timing 2	3.6757e-02	3.7280e-02	3.4571e-02
Timing 3	1.0515e-01	1.1006e-01	8.9323e-02
Timing 4	1.9577e-01	2.0168e-01	1.7296e-01
Timing 5	2.9620e-01	3.1055e-01	2.7161e-01

Table 1: Local machine timings for different thread counts

Figure 2: Parallel efficiency vs. number of threads.

7   Parallel Validation

To validate that the parallel code produces the same results as the serial code, we analyzed the error for thread counts ranging from 1 to 3. The log-log plots of the error for each thread count are shown below.

Figure 3: Error plot for 1-thread computation.

Figure 4: Error plot for 2-thread computation.

Figure 5: Error plot for 3-thread computation.

These error plots confirm that our numerical method achieves the expected quadratic convergence as $h \to 0$. This behavior aligns with the theoretical performance of the second-order finite difference scheme, demonstrating the accuracy of our implementation.

Each thread was assigned a 1D domain partition with "halo" regions to ensure accurate computation at boundaries shared with neighboring

threads. This approach helped in distributing the computation load evenly across threads. Each thread handled $\frac{n}{\text{nt}}$ rows. Halo regions allowed threads to access neighboring data for finite difference calculations.

7.1   CARC Performance Analysis

To understand the scalability and performance of the parallel implementation, we conducted strong scaling studies on the CARC supercomputing platform. This analysis involved running the parallel code on a single CARC node using problem sizes specified by option "2" in the homework skeleton.

7.1.1   Parallel Efficiency

Parallel efficiency was calculated using the formula:

$$E_p = \frac{T_1}{T_p \cdot p},$$

where T_1 is the execution time for the serial code, T_p is the execution time for the parallel code using p threads, and p is the number of threads.

Figure 6 shows the parallel efficiency as the number of threads increased. The efficiency remained high for lower thread counts but gradually decreased as more threads were used. This behavior indicates that, while parallelization significantly reduced computation time, the benefits of adding more threads beyond a certain point were constrained by factors such as data synchronization and the halo region data transfer between threads.

Figure 6: Parallel efficiency vs. number of threads.

7.1.2   Execution Time

Figure 7 illustrates the execution time for different numbers of threads on CARC. The execution time decreased as the number of threads increased, showing improved performance due to parallel computation. This trend confirms that CARC's multi-core architecture effectively distributes the workload and reduces overall computation time. However, diminishing

7

returns were observed as more threads were utilized, suggesting that synchronization and communication overheads limited further speedups.

Figure 7: Execution time vs. number of threads.

8 Conclusion

The CARC performance analysis revealed:
Strong Scaling Performance: The implementation demonstrated strong scaling up to a moderate number of threads, aligning with theoretical expectations for parallel finite difference computations.

Overhead Analysis: As the number of threads increased, the communication overhead associated with managing halo regions and synchronization became more pronounced. This resulted in a plateau in performance improvements.

Optimal Thread Count: The results suggest that an optimal number of threads exists beyond which adding more threads does not yield significant performance benefits. This threshold is a function of the problem size and the computational complexity of managing shared data.

The use of the CARC platform allowed for uniform results and greater computational resources, which would not have been possible on a local machine. This analysis highlights the importance of understanding the limitations of parallel programming, including overhead and memory bandwidth constraints, in large-scale scientific computing.