Benchmarking Parallel 1d Heat Equation Solver in Java, C++ and CUDA

1st Mudasir Javed
ERP- 27096
IBA
Karachi, Pakistan
EMail

2nd Qamar Raza
ERP-27140
IBA
Karachi, Pakistan
q.raza.27140@khi.iba.edu.pk

***Abstract*—Numerical solutions to PDEs demand substantial computational resources, particularly when high accuracy and fine discretizations are required. This paper investigates the application of parallel computing, specifically using the domain decomposition technique, to accelerate its solution. We benchmark different implementations of 1D heat equation solvers using domain decomposition in four distinct programming environments: Java, C++, and CUDA. A key focus is the performance enhancement achieved through GPU acceleration with the CUDA implementation. We explore the trade-offs between C++'s low-level control and potential for high optimization, Java's managed runtime environment, and CUDA's specialized parallel architecture for massive throughput. Our papers provide practical insights for developers in selecting appropriate environments for parallelizing PDE solvers, balancing raw performance with factors like development time, particularly when considering GPU-based acceleration.**

***Index Terms*—Parallel Computing, Partial Differential Equations, 1D Heat Equations, Domain Decomposition, CUDA, Java, C++**

I. Introduction

The one-dimensional (1D) heat equation is a fundamental partial differential equation (PDE) that describes how temperature (or other scalar quantities) distributes and evolves over time. For a 1D loop of length L (i.e., $0 \leq x < L$) and for all times $t > 0$, the equation is given by:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad 0 \leq x < L, \quad t > 0, \tag{1}$$

where $u(x,t)$ represents the temperature at position x and time t, and α is the material's thermal diffusivity. Many simulations of heat equations often require high resolution for accuracy, leading to computationally intensive tasks. The initial condition for our study is defined as $u(0, x_i) = x_i$, and to model the loop, periodic boundary conditions are applied, meaning $u(t, x) = u(t, L + x)$.

Numerically solving Equation (1) involves discretizing the continuous domain. We use N grid points $x_i = i \cdot h$ for $i = 0, \ldots, N-1$, where h is the grid spacing, employing 2nd order finite differencing for the spatial derivative. For time discretization, the explicit Euler method is used, yielding the update rule:

$$u(t+\delta t, x_i) = u(t, x_i) + \delta t \cdot \alpha \frac{u(t, x_{i-1}) - 2u(t, x_i) + u(t, x_{i+1})}{h^2}, \tag{2}$$

where δt is the time step. Even for this 1D problem, applying Equation (2) iteratively across all grid points can be slow for fine discretizations or long simulations. Parallel computing offers a path to significantly reduce this computation time. This paper focuses on a domain decomposition strategy, where the spatial grid is divided into segments, each processed by a separate thread of execution applying Equation (2) locally. Communication of "ghost zones" (boundary values from adjacent segments) is managed, for instance, using queue-based inter-process communication (IPC) mechanisms.

This study investigates the implementation and performance of such a domain-decomposed parallel solver for the 1D heat equation. We may begin with a non-parallel implementation as a baseline before progressing to its parallelized version. The core of our work involves implementing the same fundamental parallel algorithm skeleton across four distinct programming environments: Java, C++, and CUDA. A particular emphasis is placed on exploring GPU acceleration via CUDA. GPUs, with their massively parallel architecture, are well-suited for the inherently data-parallel computations arising from discretized PDEs like the heat equation, potentially offering substantial speedups. Our goal is to provide a comparative performance benchmark of these environments, evaluating raw performance development overhead and the specific advantages or challenges encountered within each language.

II. Literature Review of Related Works

Solving Partial Differential Equations (PDEs) numerically, especially for problems requiring high accuracy, often demands significant computational power. This has led to extensive research into parallel computing techniques to speed up PDE solvers. This review focuses on parallelization strategies relevant to implementing a parallel 1D heat equation solver, particularly emphasizing domain decomposition and insights from various programming approaches.

Domain decomposition is a common and effective strategy for parallelizing PDE solvers, where the problem's domain is split into smaller sub-domains, each handled by a separate processing unit. Particularly relevant to the present study, Diehl et al. (2023) conducted a benchmarking study of a parallel 1D heat equation solver. They implemented a shared-

memory, domain-decomposed algorithm using asynchronous queues for ghost zone exchange across a wide array of programming languages, including C++ and Java (which are also part of our investigation). Their work provides a direct precedent for comparing language performance for this specific problem and parallelization strategy (Diehl et al., 2023). Further exemplifying domain decomposition, Xue and Feng (2018) proposed a parallel algorithm for parabolic equations (like the heat equation) using this method, achieving good accuracy (Xue & Feng, 2018). Rogenski, Petri, and De Souza (2014) also utilized domain decomposition for 2D Navier-Stokes equations (Rogenski et al., 2014), and Brozovsky and Krejsa (2017) applied it to speed up structural analysis (Brozovsky & Krejsa, 2017), demonstrating the broad applicability of the technique.

Beyond basic domain decomposition, understanding how different parallel programming models perform is crucial. Martínez-Ferrer, Arslan, and Beltran (2023) investigated hybrid parallel implementations (MPI with OpenMP/OmpSs) for iterative linear algebra methods, which are key parts of many PDE solvers. They found that task-based hybrid approaches often outperformed MPI-only methods, suggesting that combining parallel paradigms can improve performance in the computational kernels of solvers like those for the heat equation (Martínez-Ferrer et al., 2023). Homenyk and Kozub (2021) explored OpenMP, a shared-memory model, for finite element analysis (FEA). Their findings showed OpenMP's effectiveness in reducing computation time for large FEA problems, highlighting benefits for aspects of PDE solving (Homenyk and Kozub, 2021). These studies underscore the importance of choosing appropriate parallel models, which is relevant to our comparison of Java, C++, and CUDA.

While our project primarily focuses on spatial parallelism, it's useful to note other advanced techniques. For instance, Ibrahim, Götschel, and Ruprecht (2023) explored Parareal, a parallel-in-time method, enhanced by a physics-informed neural network (PINN) and accelerated by GPUs, showing the potential of integrating machine learning with parallel PDE solvers (Ibrahim et al., 2023). Also relevant to GPU acceleration, which is a component of our study with CUDA, Ishii, Yamamoto, and Takaishi (2021) focused on accelerating linear equation solvers within PDE-based simulations using parallelized preconditioners. Their work demonstrates practical techniques for optimizing these core components on parallel architectures (Ishii et al., 2021).

Finally, rigorous performance evaluation is critical. Götschel et al. (2021) discussed common pitfalls in evaluating parallel-in-time methods, offering a guide for meaningful performance analysis in parallel PDE research (Götschel et al., 2021). Petcu (2005) constructed a performance model for parallel iterative methods in message-passing systems, providing a framework to understand factors affecting speedup (Petcu, 2005). These considerations will inform the benchmarking aspect of our project. The existing literature, especially the work by Diehl et al. (2023), provides a strong foundation for implementing and evaluating a parallel 1D heat equation solver using domain decomposition across different programming environments, highlighting both established methods and considerations for achieving efficient parallel performance.

III. Methodology: Parallel 1D Heat Solver Framework

This project implements a parallel solution to the one-dimensional (1D) heat equation, adopting the finite difference stencil method. The core idea involves calculating the value at a grid point x for the next time step $t + \delta t$ based on its current value and the values of its immediate neighbors $(x - h, x + h)$ at time t. Our primary focus is to demonstrate how the performance of this parallelization approach varies across different languages.

The approach involves discretizing the 1D spatial domain into N grid points. We apply a 2nd-order finite difference method for spatial derivatives and the explicit Euler method for time discretizations. This results in an update rule where the solution at each grid point is advanced based on its state and that of its neighbors from the previous time step.

To achieve parallelism, we employ a shared-memory, multi-threaded algorithm. The core parallelization strategy is domain decomposition, where the 1D grid is partitioned into multiple contiguous segments (local blocks). Each segment is assigned to a dedicated thread, which sequentially applies the stencil update rule to the grid points within its assigned block. **This aligns with the "map" parallel pattern, where the same operation (stencil update) is applied to different portions of the data.**

Data dependencies arise at the boundaries of these segments, as updating a point near a boundary requires values from an adjacent segment (handled by a different thread). To manage this, we implement ghost zone communication using asynchronous queues. Each segment maintains ghost zones—small regions at its edges that store copies of the required boundary data from neighboring segments. These queues facilitate the exchange of ghost zone values between threads. **These queues operate on a single-producer, single-consumer basis for each boundary exchange, allowing for efficient, potentially non-blocking communication.**

The domain decomposition approach, with asynchronous queue-based ghost zone exchange, is well-suited for the 1D heat equation. It exploits the inherent locality of the finite difference method, allowing substantial parallel computation on independent segments while managing the communication between inter-segment data dependencies efficiently. The core components of this parallel framework are summarized in Table I.

TABLE I
CORE PARALLEL FRAMEWORK COMPONENTS

Component	Description
Problem Distribution	Domain Decomposition (Map Pattern) *1D spatial grid partitioned into segments; stencil computation applied concurrently by threads.*
Data Communication	Asynchronous Queues (Single Producer/Consumer) *Used for exchanging ghost zone data between adjacent threads/segments.*

IV. CODE IMPLEMENTATION

A. Java Non Parallel

This Java implementation provides a sequential, non-parallel solution to the 1D heat equation. It employs a standard finite difference method, using two arrays to store the grid values at the current and next time steps, swapping them after each iteration. The work() method iterates through time steps and, within each step, sequentially calculates the new temperature for every grid point based on its neighbors using the heat() function. The entire computation is performed by a single thread of execution, without any domain decomposition or explicit parallel constructs for concurrent processing.

B. Java Parallel

This Java implementation parallelizes the 1D heat equation solver by employing domain decomposition and Java's concurrency utilities. It utilizes an ExecutorService (specifically a fixed thread pool) to manage a set of worker threads, each responsible for a ThreadPart representing a segment of the grid (map pattern). Communication of ghost zone data between these threads is achieved using Channel objects, which are thin wrappers around ArrayBlockingQueue. These thread-safe queues facilitate the exchange of boundary values necessary for the stencil computation, effectively implementing the asynchronous, single-producer/single-consumer communication pattern. A CountDownLatch is used to synchronize the completion of all worker threads before the main program finishes.

C. C++

The C++ implementation uses domain decomposition using std::thread for creating and managing multiple threads of execution, corresponding to the domain decomposition strategy. Each Worker object, derived from std::thread, manages a specific segment of the 1D grid. Communication of ghost zone data between adjacent Worker threads is handled by custom Queue objects. These queues are implemented using std::mutex and std::condition variable to ensure thread-safe, single-producer, single-consumer operations for pushing and popping boundary values, thus facilitating the asynchronous exchange required by the stencil computation. The main function orchestrates the setup, linking of neighboring worker threads, initiation of their run methods, and eventual joining, effectively parallelizing the heat equation solver across the available threads.

D. CUDA

The CUDA implementation leverages the massively parallel architecture of NVIDIA GPUs to accelerate the 1D heat equation solver. It defines a CUDA kernel, which performs the stencil update for a single grid point. This kernel is launched with a grid of thread blocks, where each thread within a block processes one grid point concurrently. Data for the grid is allocated on the GPU device memory (cudaMalloc) and initialized by copying data from the host (cudaMemcpyHostToDevice). The main loop iterates through time steps, launching the kernel in each step and swapping device pointers to update the grid. Periodic boundary conditions are handled directly within the kernel. Finally, after all time steps, the results are copied back to the host (cudaMemcpyDeviceToHost) and GPU memory is freed (cudaFree).

V. RESULTS

Component	Specification
CPU	Intel Core i3 (12th Gen)
RAM	16 GB DDR4
GPU	NVIDIA GeForce RTX 2060
Number of Threads Used	4 (for C and Java benchmarks)

TABLE II
SYSTEM CONFIGURATION FOR BENCHMARKING 1D HEAT EQUATIONS

For smaller problem sizes (e.g., NX:100, NT:100), an interesting observation is that the parallel versions of both C++ and Java sometimes perform worse than their single-threaded counterparts. For instance, with NX:100, NT:100, parallel Java (0.00117s) is slower than single-threaded Java (0.00053s), and parallel C++ (0.00165s) is slower than single-threaded C++ (0.00056s). This behavior is common in parallel computing. The overhead associated with creating threads, managing their synchronization (even with efficient queues), and distributing the small amount of work can outweigh the benefits of parallel execution when the computational task itself is very brief.

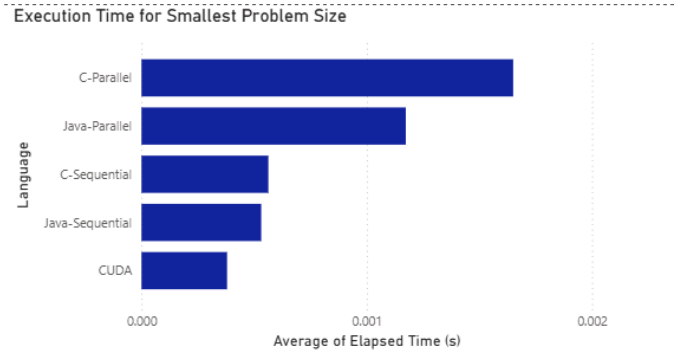
As the problem size increases significantly (eg NX:1,000,000, NT:1000 or NT:10000), the benefits of parallelism become evident. The parallel C++, Java, and particularly CUDA implementations consistently outperform their single-threaded versions by a substantial margin. CUDA, leveraging GPU acceleration, generally shows the best performance, especially as both NX and NT grow large, highlighting the GPU's strength in massively parallel, data-intensive computations.

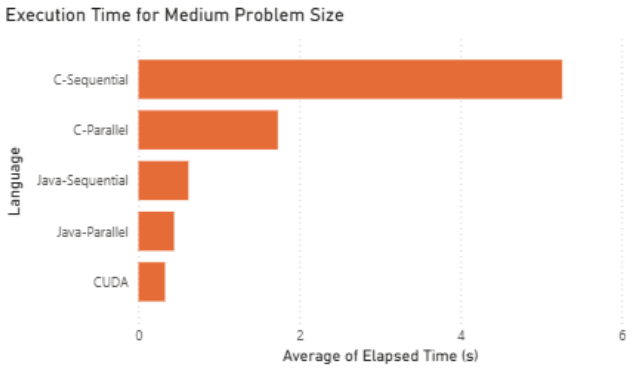
As expected CUDA implementation dwarfs the rest due to efficient GPU utilization. CUDA is particularly well suited for our 1d heat equations and the domain is split into grid points with the same calculations applied on many data points.

An unexpected result in some cases, particularly for larger problem sizes (e.g., NX:1,000,000, NT:1000 where Java takes 0.38s and C++ takes 1.62s), is that the parallel Java implementation outperforms the parallel C++ implementation.

While C++ is often expected to be faster due to its lower level control, modern Java Virtual Machines (JVMs) have highly optimized Just-In-Time compilers and efficient garbage collectors. For certain workloads and implementations, the JVM's runtime optimizations, potentially more mature concurrency libraries, or even differences in the specific implementation details of the queue and threading logic between the C++ and Java versions, could lead to Java showing superior performance.

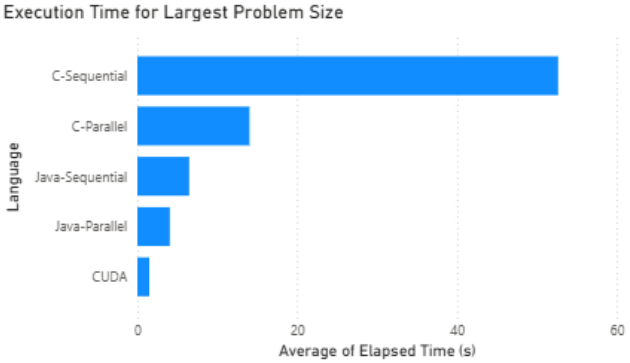
Possible reason for this unexpected result could be that single-threaded Java can outperform multi-threaded C++ for large problems, as seen in the report with times like 0.384s versus 1.61s for NX:1,000,000 and NT:1,000, due to several key factors: Java's JVM and Just in time(JIT) Compiler optimizes code during runtime, making repetitive calculations in large tasks more efficient; single-threaded Java avoids the overhead of managing multiple threads, which can slow down C++ if workload distribution or synchronization isn't perfect; and Java's memory management often ensures better cache efficiency, reducing data access delays compared to multi-threaded C++ where threads might interfere with each other's memory access.

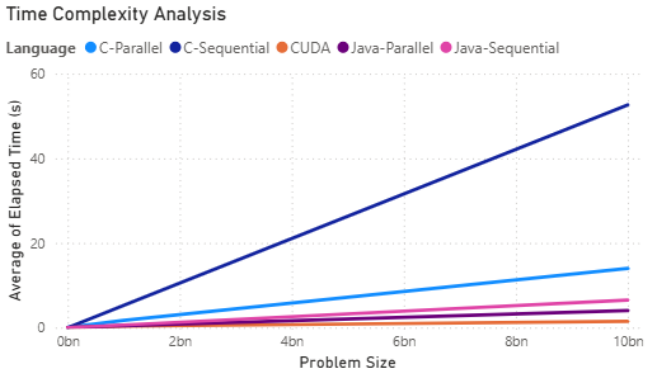




VI. Limitations

While this study provides valuable insights into the performance of different parallel implementations for the 1D heat equation, several limitations should be acknowledged. Firstly, the investigation is confined to a single, relatively





simple PDE—the 1D heat equation with an explicit Euler time-stepping scheme. The performance characteristics observed here may not directly translate to more complex PDEs (e.g., non-linear equations, systems of PDEs, or those requiring implicit solvers) or different numerical methods which might have different communication patterns or computational intensities. Secondly, our comparison primarily focuses on execution time as the performance metric. Other important factors, such as memory usage, energy consumption, and the qualitative aspects of code maintainability or developer productivity across the different languages, were not quantitatively assessed in depth.

The hardware environment used for benchmarking was specific (Intel Core i3, NVIDIA GeForce RTX 2060). Performance outcomes, especially the relative performance between CPU-based (Java, C++) and GPU-based (CUDA) solutions, can vary significantly with different hardware architectures, CPU/GPU generations, and available memory bandwidth. The number of threads for CPU benchmarks was fixed at four, and a more extensive study varying the thread count could reveal further scalability characteristics.

VII. Source Code Availability

https://github.com/Qar-Raz/1D-Heat-Equations-Solver.git

References

- Anuprienko, D. (2022). Parallel efficiency of monolithic and fixed-strain solution strategies for poroelasticity problems. *arXiv preprint arXiv:2210.06206*.
- Brozovsky, J., & Krejsa, M. (2017). Parallelization of Computational Analysis of Reinforced Concrete Slabs on Foundation. *Key Engineering Materials, 738*, 319–326.
- Caklovic, G., Speck, R., & Frank, M. (2023). A parallel implementation of a diagonalization-based parallel-in-time integrator. *arXiv preprint arXiv:2103.12571*.
- Dünnebacke, J., Turek, S., Lohmann, C., Sokolov, A., & Zajac, P. (2021). Increased space-parallelism via time-simultaneous Newton-multigrid methods for nonstationary nonlinear PDE problems. *The International Journal of High Performance Computing Applications, 35(2)*, 181–202.
- Escapil-Inchauspé, P., & Ruz, G. A. (2023). H-analysis and Data-parallel Physics-informed Neural Networks. *arXiv preprint arXiv:2302.08835*.
- Geiser, J., Martínez, E., & Hueso, J. L. (2020). Serial and Parallel Iterative Splitting Methods: Algorithms and Applications to Fractional Convection-Diffusion Equations. *Mathematics, 8(11)*, 1950.
- Götschel, S., Minion, M. L., Ruprecht, D., & Speck, R. (2021). Twelve Ways To Fool The Masses When Giving Parallel-In-Time Results. *arXiv preprint arXiv:2102.11670*.
- Homenyk, S. I., & Kozub, V. Y. (2021). Application of parallel computing in finite element analysis of constructions. *IOP Conference Series: Materials Science and Engineering, 1164(1)*, 012029.
- Hsieh, Y. M. An Efficient Scheme for Parallel Parametric-Study in Finite Element Analyses. *Parallel and Distributed Computing-PDPTA*.
- Ibrahim, A. Q., Götschel, S., & Ruprecht, D. (2023). Parareal with a physics-informed neural network as coarse propagator. *arXiv preprint arXiv:2303.03848*.
- Ishii, G., Yamamoto, Y., & Takaishi, T. (2021). Acceleration and Parallelization of a Linear Equation Solver for Crack Growth Simulation Based on the Phase Field Model. *Mathematics, 9(18)*, 2248.
- Martínez-Ferrer, P., Arslan, T., & Beltran, V. (2023). Improving the performance of classical linear algebra iterative methods via hybrid parallelism. *Journal of Parallel and Distributed Computing, 179*, 1–17.
- Niethammer, C., Glass, C. W., & Gracia, J. (2014). Avoiding Serialization Effects in Data / Dependency Aware Task Parallel Algorithms for Spatial Decomposition. *arXiv preprint arXiv:1401.4441*.
- Petcu, D. (2005). Speedup in solving differential equations on clusters of workstations. *International Journal of Computational Science and Engineering, 1(3)*, 204–215.
- Rogenski, J. K., Petri, L. A., & De Souza, L. F. (2014). Effects of parallel strategies in the transitional flow investigation. *Journal of Aerospace Engineering, 28(2)*, 04014074.
- Speck, R. (2018). Parallelizing spectral deferred corrections across the method. *Computing and Visualization in Science, 21(4)*, 127–143.
- Totounferoush, A., Ebrahimi Pour, N., Roller, S., & Mehl, M. (2021). Parallel Machine Learning of Partial Differential Equations. In *PDSEC 2021: 4th Workshop on Parallel and Distributed Machine Learning in HPC Environments* (pp. 1–8).
- Xue, G., & Feng, H. (2018). A new parallel algorithm for solving parabolic equations. *Advances in Continuous and Discrete Models, 2018(1)*, 1–14.