MAP55640 - Comparative Analysis of Hybrid-Parallel Finite Difference Methods on Multicore Systems and Physics-Informed Neural Networks on CUDA Systems

LI YIHAI, Mathematics Institute High Performance Computing, Ireland
MIKE PEARDON*, Mathematics Institute High Performance Computing, Ireland

The Finite Difference Method (FDM) is a fundamental numerical technique for solving partial differential equations (PDEs). For large-scale discretized systems, there remains a strong demand for high-performance parallel solvers that fully utilize computational resources. As a pure MPI program, communication and synchronization between processes often become bottlenecks, leading to the increased adoption of hybrid parallel strategies, which consider both message-passing parallelism and shared memory parallelism.

Another common choice for parallelism is GPU computing, where Physics-Informed Neural Networks (PINNs) have emerged as a mainstream approach for solving physical problems or for inverse solving of physical parameters based on test data. In this paper, we first design a safer and more efficient N-dimensional matrix template class, utilizing C++ features, along with a corresponding MPI N-dimensional Cartesian topology communication environment.

For solving the heat conduction equation, three CPU parallel strategies are proposed: pure MPI parallelism, master-only parallelism without computation/communication overlap using MPI and OpenMP, and master-only parallelism with computation/communication overlap. Additionally, we constructed a PINN to solve the heat conduction equation and employed CUDA technology to implement GPU-accelerated training.

Finally, weak and strong scaling tests are conducted on the three CPU parallel strategies to analyze performance differences across various nodes. The time consumption and accuracy of the final results are compared with those of the PINN as references.

Additional Key Words and Phrases: Keywords

*Supervisor of this Final Project

Authors' addresses: Li Yihai, liy35@tcd.ie, Mathematics Institute and High Performance Computing, Dublin, Ireland; Mike Peardon, mjp@maths.tcd.ie, Mathematics Institute and High Performance Computing, Dublin, Ireland.

2024. Manuscript submitted to ACM

1 INTRODUCTION

Numerical methods for solving partial differential equations (PDEs) have demonstrated superior performance compared to traditional techniques such as finite difference methods (FDM) [6], finite element methods (FEM) [6], and Monte Carlo methods (MC) [18]. In recent years, the field of deep learning has primarily focused on advancing system architectures and learning methodologies, exemplified by convolutional neural networks (CNNs) [16] and Transformers [9]. Furthermore, there has been a growing interest in developing robust models tailored specifically for numerical simulations. Despite significant advancements, modeling and predicting the evolution of nonlinear multi-scale systems, which exhibit inhomogeneous cascades of scales, remain formidable challenges when approached using classical analytical or computational methods. These methods often entail substantial computational costs and are subject to multiple sources of uncertainty.

This project centers on optimizing performance through parallel computing frameworks, with a particular emphasis on evaluating both finite difference methods (FDMs) and neural networks (NNs). A template interface for multidimensional arrays was implemented and distributed using the Message Passing Interface (MPI) [7] with Cartesian topologies, designed for ease of use by researchers. The template leverages C++ smart pointers to balance efficiency, memory management, safety, and computational accuracy. Additionally, the implementation includes hybrid parallel strategies employing MPI for inter-process communication and Open Multi-Processing (OpenMP) [3] for shared memory parallelism.

Experimental validation was conducted using a four-node cluster and a GPU-accelerated node. The results confirm that the proposed parallel models are both accurate and high-performing, with Physics-Informed Neural Networks (PINNs) [15] achieving the highest performance metrics.

The key contributions of this project are summarized as follows:

(1) Utilizing meta-programming techniques, a polymorphic object-oriented framework for multidimensional arrays was developed. On this foundation, an MPI Cartesian topology environment was designed to facilitate array distribution and ghost exchange routines.
(2) Leveraging both message-passing and shared-memory parallelism, three distinct parallel models for FDMs were implemented, tailored to different computational topologies within a cluster.
(3) Utilizing Libtorch and CUDA technologies, PINNs were designed for various PDEs, and GPU acceleration was implemented to speed up the training process.

2 RELATED WORK

To gain well quality solution of various types of PDEs is prohibitive and notoriously challenging. The number of methods available to determine canonical PDEs is limited as well, includes separation of variables, superposition, product solution methods, Fourier transforms, Laplace transforms and perturbation methods, among a few others. Even though there methods are exclusively well-performed on constrained conditions, such as regular shaped geometry domain, constant coefficients, well-symmetric conditions and many others. These limits strongly constrained the range of applicability of numerical techniques for solving PDEs, rendering them nearly irrelevant for solving problems practically.

General, the methods of determining numerical solutions of PDEs can be broadly classified into two types: deterministic and stochastic. The mostly widely used stochastic method for solving PDEs is Monte Carlo Method [18] which is a popular method in solving PDEs in higher dimension space with notable complexity.

2.1 Finite Difference Method

The FDM is based on the numerical approximation method in calculus of finite differences. The motivation is quiet straightforward which is approximating solutions by finding values satisfied PDEs on a set of prescribed interconnected points within the domain of it. Those points are which referred as nodes, and the set of nodes are so called as a grid

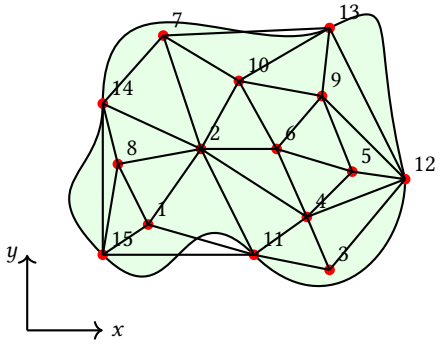


Fig. 1. The Schematic Representa of of a 2D Computiational Domain and Grid. The nodes are used for the FDM by solid circles. Nodes $11-15$ denote boundary nodes, while nodes $1-10$ denote internal nodes.

of mesh. A notable way to approximate derivatives are using Taylor Series expansions. Taking 2 dimension Possion Equation as instance, assuming the investigated value as, φ,

$$\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} = f(x, y) \tag{1}$$

The total amount of nodes is denoted with $N = 15$, which gives the numerical equation which governing equation 1 shown in equation 2 and nodes layout as shown in the figure 1

$$\frac{\partial^2 \varphi_i}{\partial x_i^2} + \frac{\partial^2 \varphi_i}{\partial y_i^2} = f(x_i, y_i) = f_i, \quad i = 1, 2, \ldots, 15 \tag{2}$$

In this case, we only need to find the value of internal nodes which i is ranging from 1 to 10. Next is aimming to solve this linder system 2.

2.2 Physics Informed Neural Networks

With the explosive growth of avaliable data and computing resouces, recent advances in machine learning and data analytics have yieled good results across science discipline, including Convolutional Neural Networks (CNNs) [16] for image recoginition, Generative Pre-trained Transformer (GPT) [21] for natual language processing and Physics Informed Neural Networks (PINNs) [15] for handling science problems with high complexity. PINNs is a type of machine learning model makes full use of the benefits from Auto-differentiation (AD) [2] which led to the emergence of a subject called Matrix Calculus [19]. Considering the parametrized and nonlinear PDEs of the general form [E.q. 3] of function $u(t, x)$

$$u_t + \mathcal{N}[u; \lambda] = 0 \tag{3}$$

The $\mathcal{N}[\cdot; \lambda]$ is a nonlinear operator which parametrized by λ. This setup includes common PDEs problems like heat equation, and black-stokz equation and others. In this case, we setup a neural network $NN[t, x; \theta]$ which has trainable weights θ and takes t and x as inputs, outputs with the predicting value $\hat{u}(t, x)$. In the training process, the next step is calculating the necessary derivatives of u with the respect to t and x. The value of loss function is a combination of the metrics of how well does these predictions fit the given conditions and fit the natural law [Fig. 2].

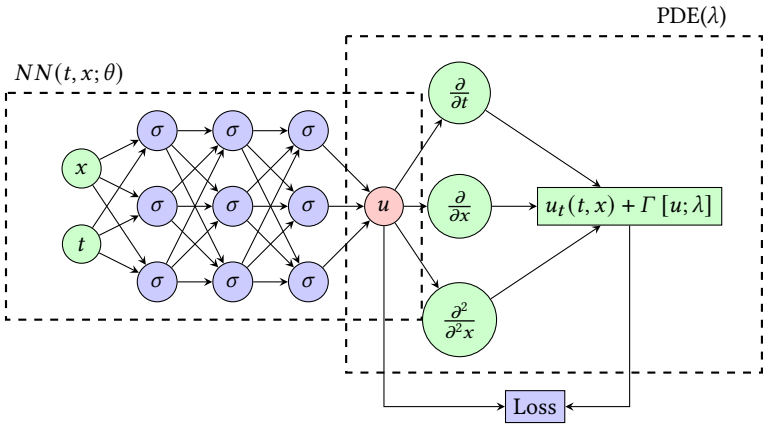


Fig. 2. The Schematic Representation of a structure of PINN, a FCN with 4 layers (3 hidden layers)

2.3 Finite Difference Time Domain Method

As described previously in the section 2.1, FDM could solve the PDEs in its original form where Finite Element and Finite Volume Methods gained results by solving modified form such as an integral form of the governing equation. Though the latter methods are commonly get better results or less computational hungery, the FDM has many descendants, for instance the Finite Difference Time Domain Method (FDTD) where it still finds prolific usage are computational heat equation and computational electromagnetics (Maxwell's equations). Assuming the operator $\mathcal{N}[\cdot; \lambda]$ is set to ∇ where it makes E.q. 3 become to heat equation 4.

$$\frac{\partial u}{\partial t} - \lambda \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = 0 \tag{4}$$

Using the key idea of FDM, assuming the step size in spatio-time space are Δx, Δy and Δt, we could have a series equations which have form [E.q. 5].

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = \lambda \left(\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j+1}^{n}}{\Delta y^2} \right) \tag{5}$$

when the time step size satisfies the Couran, Friedrichs, and Lewy condition (CFL[8]). We could get the strong results by iterating the equation 5, or more specifically using equation 6 to get the value u of next time stamp $n+1$ on nodes i, j.

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \frac{\lambda \Delta t}{\Delta x^2} \left(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n} \right) + \frac{\lambda \Delta t}{\Delta y^2} \left(u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j+1}^{n} \right) \tag{6}$$

also shown in the figure 3.

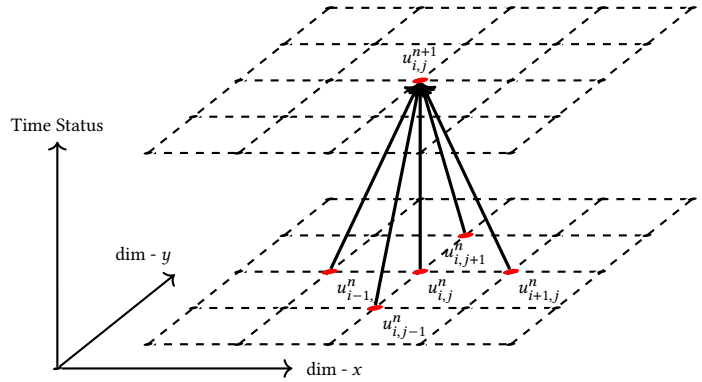


Fig. 3. The Schematic Representation the computational spatio-temporal domain of FDTD methods.

2.4 Scalability

High performance computing which also called parallel computing is solving large scale problems using Clusters, with a number of compute nodes and many CPUs. Parallel computing is many CPUs, thousands of CPUs in most case, are simultaneously processing the data and producing the exceptional results, which significantly reduce the time consumption totally. In this scenario, scaling test is a widely used to investigate the ability of hardware and programs to deliver more computational power when the amount of compute resources increase.

2.4.1 Amdalh's Law and Strong Scaling. The Amdalh's law for speedup is

Theorem 2.1. *Let f_s and f_p be the fraction of time spend on purely sequential tasks and that can be parallelized, then the Amdalh's formula [1] of final speedup is*

$$S_N = \frac{1}{f_s + \frac{f_p}{N}} \tag{7}$$

and it states that for a fixed problem, the upper limits of speedup is restricted by the fraction of sequential parts, where the limits is $\lim_{N \to +\infty} S_N = 1/f_s$, which is called the strong scaling.

2.4.2 Gustafsson-Barsis' Law. The Gaustafsson-Barsis' law for speedup is

Theorem 2.2. *Let f_s and f_p be the fraction of time spend on purely sequential tasks and that can be parallelized, then the Gaustafsson-Barsis' formula [12] of the final speedup is*

$$S_N = f_s + N f_p \tag{8}$$

With Gaustafsson-Barsis's law, the scaled speedup increases linearly with the number of processes, and there is no upper limits for the scaled speedup, which is called weak scaling.

The weak scaling tests of an high performance program runing on cluster is necessary for researching the inner relationships between the number of CPUs and the scale of problems. In order to get better weak scaling performance, the fraction of synchronization among processes and the cost of communications plays a minor role when the number of resources increase. In the meanwhile, unlike the strong scaling test, the iterations takes for converging on different problem sizes are various as well. Thus, the weak scaling is more effective if it only consider the scaling for single iteration, where the speedup on N CPUs has formula follows

$$S_N = N \frac{T_1 C_N}{C_1 T_N} \tag{9}$$

where T_N and C_N denote with the time and epochs that the solver takes for converging on N CPUs.

3 PROBLEM SETUPS

Due to the inherent limitations of current computing systems, obtaining sufficiently precise solutions is both computationally expensive and time-consuming. These challenges arise from the constraints imposed by the clock speed of computing units, as described by Moore's Law [11], as well as the relatively low communication speeds between these units. While modern numerical methods have advanced to a level where they can produce satisfactory results within acceptable time frames across many research domains, the increasing scale of problems we aim to solve has driven the search for more cost-effective approaches. This has led to a growing interest in neural networks as a promising alternative.

In this project, I aim to evaluate the performance of the Finite-Difference Time-Domain (FDTD) method and the Physics-Informed Neural Network (PINN) model within parallelized computing environments by find the steady-state solution of PDEs. These two methodologies broadly represent the current approaches to handling PDEs, specifically CPU-based parallelization and GPU-based parallelization.

3.1 General Form

Starting with the general form of the PDEs, rather than the specific euqations, is because different equations give perform differently on the same compute system. To this end, consider the previously discussed form of PDEs shown in equation 3 which parametrized by number λ and an operator $\mathcal{N}[\cdot;\lambda]$. Moreover, we assume the variable x is a 2D or 3D spatio vector which is written in $\vec{x} \in \mathbb{R}^d$, $d = 2, 3$.

$$\begin{aligned} &\frac{\partial u}{\partial t}(t,\vec{x}) + \mathcal{N}[u(t,\vec{x});\lambda] = 0, && \vec{x} \in \Omega, t \in [0,+\infty) \\ &u(0,\vec{x}) = \varphi(\vec{x}), && \vec{x} \in \Omega \\ &u(t,\vec{x}) = g(t,\vec{x}), && \vec{x} \in \overline{\Omega}, t \in [0,+\infty) \end{aligned} \tag{10}$$

The domain of this PDE system is considered between 0 and 1, where is denoted with $\Omega = [0,1]^d$, $d = 2, 3$. To these setups, we have the general form of the PDEs we are going to investigated, shown in equations 10 The boundary condition shown in E.q. 10 is Dirichlet Condition known as first type boundary condition, where as the second type boundary condition (Von Neumman) [E.q. 11] gives the other form of $u(t,\vec{x})$ at the boundary $\overline{\Omega}$.

$$\frac{\partial u}{\partial \vec{x}} = g(t,\vec{x}), \quad \vec{x} \in \overline{\Omega}, \quad t \in [0,+\infty) \tag{11}$$

3.2 Specific Form

With general form proposed in section 3.1[E.q. 10], I specify a particular form of this heat problem to help us to have better understand the quality of our solutions and programs. In 2 dimension space, the domain $\Omega = [0,1]^2 \in \mathbb{R}^2$ and its

boundary denoted with $\overline{\Omega}$, the initial condition $\varphi(x, y) = 0$. such problem has the certain form below

$$\begin{aligned} &\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial u^2}{\partial^2 x} + \frac{\partial u^2}{\partial^2 y} \right) && (x, y) \in \Omega,\ t \in [0, +\infty) \\ &u(0, x, y) = \varphi(x, y) = 0 && (x, y) \in \Omega \\ &u(t, x, y) = g(x, y) = \begin{cases} y, & x = 0, y \in (0, 1) \\ 1, & x = 1, y \in (0, 1) \\ x, & y = 0, x \in (0, 1) \\ 1, & y = 1, x \in (0, 1) \end{cases} && t \in [0, +\infty) \end{aligned} \tag{12}$$

With given format, and $\alpha = 1$, we have the analytical solution of this equations, where is

$$u(t, x, y) = x + y - xy, \quad (x, y) \in \Omega,\ t \in [0, +\infty) \tag{13}$$

In 3 dimension space, similarly, with identical initial condition set up to 0, coefficient $\alpha = 1$, the boundaries are

$$u(t, x, y, z) = g(x, y, z) = \begin{cases} y + z - 2yz, & x = 0, \\ 1 - y - z + 2yz, & x = 1, \\ x + z - 2xz, & y = 0, \\ 1 - x - z + 2xz, & y = 1, \\ x + y - 2xy, & z = 0, \\ 1 - x - y + 2xy, & z = 1 \end{cases}, t \in [0, +\infty) \tag{14}$$

and the equation is

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial u^2}{\partial^2 x} + \frac{\partial u^2}{\partial^2 y} + \frac{\partial u^2}{\partial^2 z} \right) (x, y, z) \in \Omega,\ t \in [0, +\infty) \tag{15}$$

In such case, the analytical solution has for form below

$$u(t, x, y, z) = x + y + z - 2xy - 2xz - 2yz + 4xyz, \quad (x, y, z) \in \Omega,\ t \in [0, +\infty) \tag{16}$$

3.3 Discretization

To begin with discretizing the objects or regions we intend to evaluate via matrices, we consider a straightforward approach: using the coordinates in $d = 2, 3$ dimensional spaces and the function values at those points to simplify the objects. This naive approach works well for investigating objects with regular shapes, such as a cube.

For the FDTD (Finite-Difference Time-Domain) method, we use a finely generated d-cube with shape $\{n_i\}_i^d$. Including the boundary conditions, the cube has $\prod_i (n_i + 2)$ nodes. It requires $4 \prod_i (n_i + 2)$ bytes for float32 or $8 \prod_i (n_i + 2)$ bytes for float64 to store in memory. With this setup, for equally spaced nodes, we have:

$$\Delta x_i = \frac{1}{n_i - 1} \tag{17}$$

Unlike the previously generated regular grid of points with dimensions $n_x n_y n_z$, another strategy is to randomly generate the same number of points based on the same known conditions, covering both the central part and the boundary. In this scenario, shown in figure, there are $n_x n_y n_z$ central points, with function values set according to the

boundary conditions, and $2 \times (n_x n_y + n_y n_z + n_z n_x)$ boundary points to be solved. This set of points can be used for training a PINN model.

4 N-DIMENSIONAL MATRIX IMPLEMENTATION

4.1 General Setups

Initially, we need to determine the data types to be used and define macros for assertions and helper functions to ensure that the program can detect common bugs and report their locations. These features can also be disabled in the release version for performance optimization. Such details are defined in the `assert.hpp`, `helper.hpp`, `types.hpp`, and other related header files located library and its sub-folder seeing in the appendix B.

4.2 Template Multi-dimension Matrix Detail Design

The FDTD method is a type of FDM, and the main idea behind FDM is briefly outlined in Section 2.1. The challenge lies in implementing it in a computer system to ensure it runs both correctly and efficiently. Also, the data types I am using for storing sizes are `unit32_t` and `unit64_t` while I defined as trait, such as `Dworld` and `Qworld` respectively, as `size_type` and `super_size_type` in `__detail` namespace.

4.2.1 *Performance Balancing.*

Template Class. Instead of doing this by using hierarchy in C++, which will cause the memory of object becomes complicated and unpredictable, and leading to scattered data members between base and derived class objects. This scattering can increase the cache misses which accessing these members, as the data might not be contiguous in memory. Also, with deep inheritance hierarchies, program has higher change to occur diamond problems, since it increases the code complexity. In detail, the diamond problem leads the duplicate inheritance and ambiguity in the method resolution, which will drop performance down again. Although, it is solvable by using `virtual` inheritance which is called meta-programming, but again, it increase the complexity.

In such case, I chose to use template class to design the matrix object, which implement compile-time polymorphism as opposed runtime polymorphism provided by inheritance and virtual functions. With such template design, the compiler makes the decisions about which function or class instantiate is made at compile time, eliminating the need for `vtables` and indirect function calls, which leads to more efficient code.

Memory Management. Rather than using the standard library's (STL) vector module, which can be slower due to the overhead of row pointers, I opted to build the Matrix object using a unique pointer (`std::unique_ptr`), which includes only basic features such as reset, swap, and most importantly, a destructor that automatically deletes pointers. This approach enhances the safety of memory management in our programs. Additionally, from a safety perspective, given that I implemented many features within the matrix object, I followed standard library conventions for naming. This includes using the `__detail` namespace within namespace `multi_array` to hide objects and features that are not intended for direct use by the end user.

4.2.2 *Template Object Design of Matrix Shape.*

Strides. Besides that, in the multidimensional cases, the size in each dimension is not enough for accessing variables, this is where we need the `strides` member variable, which stores the number of element the operator needs to skip in each dimension. The `__multi_array_shape` object is encapsulated within the `__detail` namespace and serves as a member variable of the later template object for the multi-dimensional matrix. This object includes a member variable defined using the STL vector, as the shape object primarily stores the sizes for each dimension, which typically

requires only a small amount of space. Additionally, this object provides member functions 1 to access the size of a given dimension.

Algorithm 1 Stride implementation

```
dims                                          # STL vector, stores the matrix's size in each dimension.
strides                                       # STL vector, has the same size with dims.
n = dims.size()                               # Store the dimension of matrix.
strides[d-1] = 1                              # Stride is 1 in the first dimension.
for d = n - 1; d > 0; –d do
  strides[d-1] = strides[d] * dims[d]         # Determine stride in the latter dimension.
end for
return strides
```

Performance Balancing. In certain scenarios, we only require the shape information of a matrix without needing to access the entire matrix object. Accessing the shape information through well-defined operators is a more efficient way to handle multidimensional matrices. This is particularly crucial in parallel programming, where understanding the shape of a matrix is of critical importance. Sometimes, a process may need to know the shape of matrices stored on other processes.

In such cases, using this matrix object as a local variable within functions increases the likelihood that the compiler will store it in a register, which is generally faster than using heap or stack memory. In addition, it includes check and cast functions that allow the user to verify if the template data type `__T` is signed using `constexpr`. The `constexpr` keyword ensures that this check occurs at compile time, and if the data type is not legal, the program will assert and provide a message indicating that the indexing value must be a non-negative number.

4.2.3 Template of Multi-dimensional Matrix Implementation. The Matrix in this project is designed to support various data types in C++. Consequently, the matrix is implemented as a template class with several essential features, template variable `__T` and `__NumD`, for the value data type and number of dimension, also including iterators, swap functionality, fill operations, and support for the IO stream operator «. To facilitate this, the `__array_shape` object is used to explicitly manage and access the array's shape information.

Operator `()`. The hard part of this object designed is the support template number of dimension, whereas the dimension is integer not less than 1, the operator of access element is designed by following algorithm

Algorithm 2 Operator (`Ext ... exts`) of template matrices object `__detail::__array`

```
__NumD, __T;                                  # Template variables: dimension, data type.
FINAL_PROJECT_ASSERT_MSG                      # Number of Arguments must Match the dimension.
index = 0, i = 1                              # Initialize variables in advance.
indices[] = __shape.check_and_cast(ext)       # The indexes must non-negative.
for i < __NumD; ++i do
  index += indices[i] * __shape.strides[i]
end for
FINAL_PROJECT_ASSERT_MSE                      # Boundary checking.
return __data[index]
```

Overload operator «. In order to print the multi-dimension array with operator «, I designed a recursive helper function to print the matrix on given dimension. Thus we could call the function on the first dimension, and it will recursively print all dimensions.

Algorithm 3 Recursive Function to Print Multi-Dimensional Array

```
current_dim, offset;                                          # Parameters: current dimension, offset.
Dims __Dims;                                                  # Template variable: number of dimensions.
if current_dim == __Dims - 1 then
  os « "|"                                                    # Start printing last dimension.
  for i from 0 to arr.__shape[current_dim] - 1 do
    os « std::fixed « std::setprecision(5) « std::setw(9) « arr.__data[offset + i];   # Print array elements with formatting.
  end for
  os « " |\n";                                                # End of current row in the last dimension.
else
  for i from 0 to arr.__shape[current_dim] - 1 do
    next_offset = offset;                                     # Initialize next offset.
    for j from current_dim + 1 to __Dims - 1 do
      next_offset *= arr.__shape[j];                          # Update next offset based on shape.
    end for
    next_offset += i * arr.__shape[current_dim + 1];          # Finalize next offset for recursion.
    self(self, arr, current_dim + 1, next_offset);            # Recursive call to print next dimension.
  end for
  os « "\n";                                                  # Print a newline after each dimension.
end if
```

4.3 Template Multi-dimension Matrix Interface Design

With contiguity of safety, this object of multi-dimension array is accessible to users without direct visit to the memory space where store values of matrix.

4.3.1 Resource Acquisition Is Initialization (RAII). This private object has only a member variable, a unique pointer to the template `__array`, and other member function provide necessary features to operating on it. Smart pointers acquire resources in their constructor and automatically release them in their destructor, which is the essence of RAII. By releasing resources in the destructor, smart pointers help prevent resource leaks. When an exception occurs, smart pointers automatically release resources, preventing resource leaks, thus it enhanced the safety level of using resources, reduce the potential memory leak problems.

4.3.2 Template Multi-dimension IO for writing to/reading from file. Initially, the multi-dimension matrix has variables shape, and values which given dimension and size in each dimension. This template design end up with these variable can be stored in given data types also leads with lower portability. To avoid such problems and from other point of views, I chose to store the matrices in binary format, rather than other type files. There are couple benefits of doing so,

(1) Compatibility and Portability: The format of binary files is relatively stable and can be easily used in different programming environments or applications. Unlike `.txt` files, `.mat` files those has less compatibility across different platforms.

(2) I/O Performance: Binary files can perform block-level I/O operations directly without needing to parse text formats or convert data types. This usually makes reading and writing binary files much faster than `.txt` files, especially when dealing with large-scale multidimensional matrix data.
(3) Support MPI IO: Binary files support the MPI IO, which provides a significant reduction in the cost of communication, when storing and reading the large scale matrices.

However, the IO does not play a critical role in effects performance of FDTD algorithms, if and only if we need to store or load the data during evolving the arrays.

5 PARALLELIZATION OF MULTI-DIMENSIONAL MATRICES ON CARTESIAN TOPOLOGIES

5.1 MPI Parallel Environment Design Scheme

5.1.1 *MPI Setups.*

Environment. Similar to how `malloc` in C and `new` in C++ require manual memory management, the MPI environment also necessitates explicit initialization and finalization. However, unlike the efficient implementation of smart pointers in the STL, Boost.MPI[17] - a high-level parallelism library - may not be the optimal choice for high-performance programs. Therefore, I chose to design a custom MPI environment that encapsulates the necessary features specific to this project.

The `mpi` namespace, a sub-namespace of `final_project`, provides the `environment` class. This class integrates MPI initialization using the constructor, which invokes `MPI_Init_thread`, provides multi-threading shared memory parallelism in MPI, and MPI finalization through the destructor, which calls `MPI_Finalize`.

It also offers direct access to the rank and the number of processors within the MPI communicator. Furthermore, I have explicitly deleted the copy and move assignment operators to enhance safety. This design decision aligns with the RAII principle, ensuring that MPI environment resources are automatically managed, thereby preventing leaking and using-uninitialized problems.

Types and Assertions. Aligned meta-programming with polymorphism principles, I designed a template function to retrieve the corresponding MPI basic data types, leveraging the fundamental data types I defined as traits at the outset. Moreover, I provides some MPI macros in assert file, these macros provide a unified interface for dealing with MPI-related errors, ensuring that MPI errors are handling consistently, safely.

5.1.2 *MPI Topology (`Cartesian`).*

The namespace `topology` is a sub-namespace of `mpi`, the template Cartesian structure is the mainly used object in following problems. To optimize memory usage, this object maintains only essential multi-dimension matrices' global and local shape member variables. It also contains a MPI Communicator and MPI value data type, halo data type along with the neighbors' rank in the source and dest sites. To ensure the MPI security, the copy and move constructors as well as assignment operators are manually removed. Additionally, the destructor is customized for properly release halo data type and Cartesian communicator.

Determine the local matrix's localtion. Evenly distributing tasks across processes is of critical importance. To address this, I designed an algorithm to divide an integer N evenly to n clients, where I could put it in use in many cases. Rather than implementing a standalone function, I chose to implement a lambda function, which is a feature in C++ that do not significantly impact the performance. It allows me to design a small function which is not frequently use or play a key role in performance. Ideally, this function will be only called when I construct the MPI topology based multi-dimension

Algorithm 4 Lambda Function (decomposition): Split tasks evenly to n processes evenly

```
n, rank                                  # const Integers, total number and current rank of Processor.
N                                        # constant Integer, Problem size.
s, e                                     # Integer, start, end indexes.
n_loc = n / N                            # Divide the problem evenly.
remain = n % N                           # Get the remaining tasks.
s = rank * n_loc + 1                     # Calculate the start indexes.
if rank < remain then
  s += rank                              # Give a task to process the rank is smaller than remain.
  ++n_loc                                # Update local number of tasks.
else
  s += remain                            # Add the remain to start index, after split remains.
end if
e = s + n_loc + 1                        # Get the ending indexes
if e > n or rank == N - 1 then
  e = n                                  # If it is the end of all.
end if
```

array, where I need cut the global matrix's shape evenly and create local matrices with local shape. Using local shape to create local matrices is obviously a memory-saving techniques when the problem size gets larger. Eventually, the lambda function is only applied in constructor of template Cartesian structure, which constructed from an input global and a MPI topology environment.

Moreover, the topology information is determined by `MPI_Cart_coords` for the coordinates and `MPI_Cart_shift` the neighbors of each process in all dimension. Below is a example [Fig. 4] of cartesian topology of 24 processors, with no period in all dimensions.

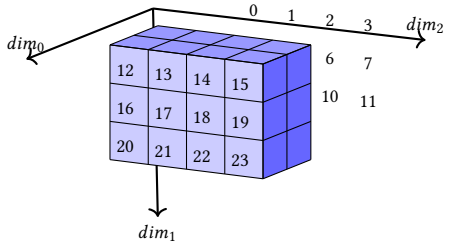


Fig. 4. The Schematic representation of the MPI Cartesian topology scheme of 24 processors on 3 dimension space, which has a $2 \times 3 \times 4$ grid of processes

Determine MPI datatypes for communication. In order to do MPI communications, the source and the destination of every process are necessary, also the datatype. When working with the meta-designed multidimensional matrix, we need to utilize the function `MPI_Type_Create_subarray` follow the routine 1 to create essential halo datatypes which follows the routine 5 below Figure 5 showing the details of communications in 3D Cartesian topology distributed matrices. For each process, it has to communicate with all the neighbors from three directions, sending and receiving data from 6 other processes which means totally 16 MPI communications each.

```
1  MPI_Type_create_subarray(
```

Algorithm 5 Creating MPI Datatype for communication of Ghost

```
array_size, array_subsize, array_starts={0}   #std::array<Integer, NumD>, the information of matrix.
for i = 0 : NumD do
   Split tasks in dimension i by calling decomposition
   array_size = local shape
   array_subsize = array_size - 2
end for
for i=0 : NumD do
   temp = array_subsize[i]                                          # Store the number temporally.
   array_subsize[i] = 1
   MPI_Type_Create_subarray and MPI_Type_commit()                   # Create halo in dimension i and commit.
   Restore temporal array sub-size.
end for
```

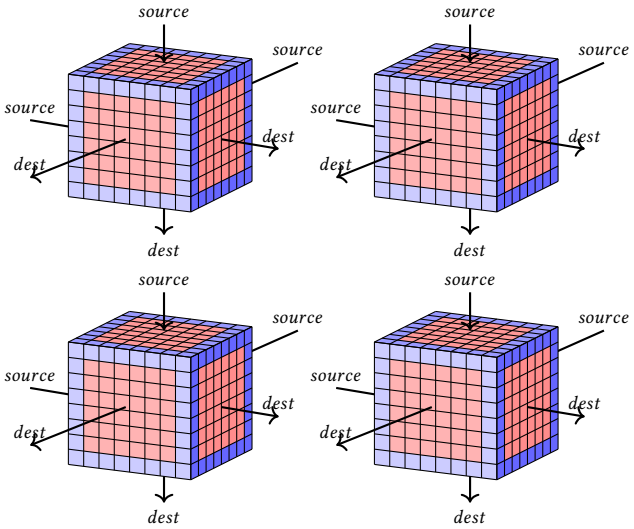


Fig. 5. Representation of a 3D MPI Communication Scheme of 4 among 24 processes between 3 dimension sub-arrays. The reddish small cube stands for the memory space stored the ghost data that needed for MPI communications. The bluish small cube stands for the space stored the junk values which are not gonna be used in the entire progress. The source/destination directions show the relationships between local process and other neighbor processes.

```
    dimension,                /// Dimension of this array  : NumD
    array_sizes.data(),       /// shape of local array     : local_shape
    array_subsizes.data(),    /// shape of sub-array       : N_cpy
    array_starts.data(),      /// starting coordinates     : {0, 0, ..., 0}
    MPI_ORDER_C,              /// Array storage order flag : Row major in C/C++
    value_type,               /// Old MPI Datatype         : get_mpi_type<T>()
    &sbuf_block);             /// New MPI Datatype
```

Listing 1. Routine for creating sub-array `MPI_Datatype` as send buffer.

Besides that, we could see the figure 5 shows the bluish cube which stands for the memory space for storing junk values. As the sub-array of each process gets smaller, the percentage of junk values goes up which means updating the values in this array is a low-efficient operation.

5.2 Template Distributed Multi-dimension Matrix Design

5.2.1 Detail Object Design. Adhering to the STL safety routines, I chose to create detail template class object, hidden from users, named `__array_Cart<class __T, __size_type __NumD>`. Here, the `__T` represents the value type, `__size_type` specifies the type of number of dimensions. Since it is internal and not exposed from users, I decided to directly use other detail objects as member variables rather than smart pointers. In this context, Cartesian matrix has public member variables `__array` and `topology::Cartesian`, and provides memory operations. But the copy, move constructors and assignment operators are removed. This approach enhances both performance both performance and simplicity by avoiding unnecessary abstractions in the internal design while maintain a robust memory management.

Distributed operator «. The STL os stream operator « prints the matrices of all processes in sequence which build on multidimensional matrix's. Thus the Unix standard `fflush` function is utilized for flushing the cache in terminal, to ensure the stdout is print immediately.

5.2.2 User Interface Design. The `array_Cart<class T, size_type NumD>` is an object exposed to users, whereas only provides limited access to member variables by smart pointer. As the size of matrices stored, it becomes clear that memory management is critically important. Secondly, especially in MPI distributed matrices, exposing direct memory access by set it to public member is dangerous. With the profits mentioned in Section 4.3.1, using unique pointer brings more benefits in this scenario,

(1) MPI program memory management has higher complexity level. Adhering RAII routines, the resources are bind with object, including MPI objects, and will be deleted as the object destructed.
(2) Simplifying Concurrency Control. Synchronization between processes is a critical issue. By using the RAII, user could unsure the resources are locked or released automatically, preventing the risk of deadlocks and resource contention.

5.3 Template Gather of Cartesian Distributed Multi-dimension Matrix

The inverse of distributing multidimensional matrices [Sec. 5.2] based on MPI Topology [Sec. 5.1] is gather all distributed matrices to root from all processes. Unlike operating on ordinary objects, gather function is operating on template objects which makes MPI operations harder. Rather than applying full specialization for each dimension, continuing using meta-programming skills on this brings couple promotions

(1) Meta-programming, significantly reduce the redundancy of code, where I found the gathering operations on each dimension are highly repeatable.
(2) Creating MPI Datatype also can be simplified due to the benefits of polymorphism. Especially when handling the high-dimension matrices.
(3) Using template makes the function is eligible to apply on various value basic data types, `Float`, `Double` etc, and provides consistency when handling uncompatible data type.
(4) Meta-polymorphism is more likely to have better performance, the initializations of template function are completed in compile time, which means there will be no run-time type-checking or type-casting.

5.3.1 Implement of Gather. The `Gather` has following prototype, with the value type of entities of matrices and number of dimension as `size_type`,

```
template <typename T, size_type NumD>
void Gather(gather, loc, root);
```

Listing 2. Prototypee of `final_project::mpi::Gather`

where:

- `gather` is a reference of `multi_array::array_base<T, NumD>`, which will collect local data and store as the global matrices.
- `loc` is a constant reference of the local matrices of `const array_Cart<T, NumD>`, which holds local values and will sending data to root process.
- `root` is a constant `Integer`, stands for the rank of root process.

The main idea of this function follows the following algorithm

Algorithm 6 Scheme of Gather local matrices to Root process.

```
Create gather object.
Determine the sending address and sizes on each dimension, considering the boundary.
Send the local size, starts information to root process.
if rank != root then
    Create local sub-array MPI Datatype, send to root.
else
    for pid = 0 : number process do
        if pid != root then
            Root creates MPI Datatype using received local size information.
            Root receives data from others.
        else
            Move the values by recursively applying local memory copy.
        end if
    end for
end if
```

5.3.2 Determine Local Information. Initially, the matrix object is designed on compatibility, it can read/load data from binary `.bin` files. Thus the distributed matrices are saving on the root process, by iterating through all processes, calculating the sizes and start indices on each dimension. Receiving them using `MPI_Recv` from other processes send by `MPI_Send`.

Considering boundaries, given a D dimensional matrices, with global size $\mathcal{N}_{glob} = \{N^d_{glob} + 2\}^{D-1}_{d=0}$. The local matrices has shape $\mathcal{N}_{loc} = \{N^d_{loc} + 2\}^{D-1}_{d=0}$, starts from $\mathcal{S} = \{s_d\}$ and ends from $\mathcal{E} = \{e_d\}$ globally. In the d dimension, as long as the matrix starts s_d equals to 1, it aligns the global boundary from source site which means the index should step back to 0 to include the boundary. In other side, the global boundary locates at the contiguous address in d dimension, the matrix should send a more dice in this dimension, which means the sending size $N^d_{loc} + 1$. Moreover, for the special case when there is only one process.

Adhering the RAII design, Gather function is exposed to user, thus I have not apply `swap` memory operation between Cartesian distributed matrices and none-distributed matrix just for handling the corner case. Eventually, I solve this by introducing an additional variable called `back`, which means the Root will copy for layer of data to another matrix. By default, `back` is 0, and will be set to 1 if and only if the number of process is 1.

With above analysis, the scheme of find local information follows

Algorithm 7 Scheme of Finding Local Sending Information: starts, shapes, indexes.

```
Make copies of local shape N_cpy, starts starts_cpy.
back = 0                                          # For handling 1 process case.
if number process == 1 then
  ++back
end if
index = {1, 1, ..., 1};                           # Default sending index of local matrices.
for d = 0 : NumD do
  if starts[d] == 1 then
    - starts_cpy[d], -index[d]
    ++ N_cpy[d]
  end if
  if ends[d] == global_shape[d] - 2 then
    ++N_cpy[d]
  end if
  MPI_Gather( ..., Root)                          # Send starts_cpy, N_cpy to Root;
end for
```

5.3.3 Creating Send/Recv MPI Datatype. Local MPI communications are low efficient operations comparing to local memory copy operation, `memcpy` was used to recursively copy contiguous data from local matrix to global matrix exclusively on root process. In a matrix with undefined dimension, creating data types for communicating is harder than in a specialized matrix. For the none-root processes, they should send their data without ghost values, thus the `MPI_Type_create_subarray` was used to create a sub-array `MPI_Datatype` for sending message to root process. The local arrays have shape $\mathcal{N}_{loc}$, and the sub-arrays have shape $\{N\_cpy\}_{d=0}^{D-1}$ determined by above routines [Alg. 7] respectively, the starts indexes are unified as 0s. On the root process, the only difference is that the global shape of array `array_sizes` are equal to the shape of global matrix.

5.3.4 Local Copy Recursive Function. The `memcpy` can only copy limited data to global matrix once, since the elements we want to gather are not aligning on contiguous memory, but only a small amount of them - on the 0 dimension - are continuously located. Dut to the dimension of matrix is polymorphic and from the efficiency concern mentioned above [Sec. 5.3.3], the smallest copy operation using `memcpy` was encapsulated in a lambda function for recursively call and only visible in limited scope.

5.3.5 Performace. Gather function is a computationally expensive operation when the scale is large and runs across thousands of processes. With this reason, gathering results is not a optimal choice for getting results in practical. However, the performance balancing is still important since helpful for debugging and stay a health routine of coding. In a brief, I specific designed some performance tunning features for following reasons

(1) Using lambda function can slow down the performance, however, gather operation is not always the core of a program since it's an IO operation for debugging or get the final results at the very end. Besides, it's only

Algorithm 8 `copy_recursive(size_type)`: Copy data using `memcpy` on given dimension

```
Given current dimension dim.
if dim == Num - 1 then
  memcpy(                                                  # Copy data from local to global.
    gather.begin() + gather.get_flat_index(loc_idx),       # Get saving index in global matrix.
    loc.data() + loc.get_flat_index(loc_idx),              # Get copy start address of local matrix.
    n_list_cpy[dim][pid] * sizeof(T)                       # Number of elements.
  );
else
  for i = start_cpy[dim] : loc.ends[dim] + back do
    local_indexes[dim] = 1
    copy_recursive(dim + 1)                                # Recursive calling, copy next dimension.
  end for
end if
```

available when the root process is receiving data, which is a small scope compare to the other parts. In all the drawback can be ignored in this case.

(2) The gather matrix is only initialized when we need gather the results. The reason for initializing gather object in this function, is that gather matrix is conventionally very large. If it is create in advance, it will consuming large amount of memory space, which will reduce the rate of memory hit rate, affect the speed of latter computing.

(3) Adhering RAII routine, I chose to make an copy of each parameters of local and global metrics. While these objects are relatively short, creating them locally could help us make the original data safe, and make them have higher chance stored in register or cache which a lot faster than reading/writing from/to memory.

5.4 Parallel MPI-IO for Distributed Matrices

Building a gather function is indeed convenient, as it allows us to collect data from all processors, and just using IO features of N-Dimension matrix designed previously. However, as discussed in earlier section 5.3.5, even with optimizations on resources usage and MPI performance, the gather function still becomes increasingly inefficient as the problem scale grows. Also, the gather demands more memory space as it requires enough memory on the root processor to store the full-scale final solution. Given that memory is often limited and highly expensive, it is more reasonable to use MPI-IO to implement the parallel I/O, where MPI-IO enables all processes to write results directly to disk. Using MPI-IO offers several significant advantages:

(1) Significantly reduces the I/O and communication burden on the root processor, as the root no longer needs to gather data from all other processors before performing I/O operations.

(2) Completely eliminates the overhead associated with the complexity of MPI communication, which previously consumed substantial resources and execution time in the gather function. By using all processors to perform I/O operations directly to disk, this overhead is entirely removed.

From the other perspective, applying MPI-IO will not improve performance of latter FDTD methods or PINNs, but it is a good habit for meta-programming which allows us do debugging work conveniently on larger scale problems.

In order to do this, `MPI_Win_set_view` will be used. Before that, the file data type (filetype) should be set first to facilitate non-contiguous memory access. I had not chose to use `MPI_Type_create_darray`, since it is used to generate

the data types corresponding to the distribution of an N-dimensional array of oldtype elements onto an N-dimensional grid of logical processes without considering ghost boundaries.

Due to I have determine the start indexes using lambda function [Alg. 4], the MPI type create function `MPI_Type_create_subarray` was used for creating newtype for creating file views. Displacement was the designed for holding size of N-dimension array, and it is read/write from/to root process. After the file view created, all processes write data into the file associated with file view collectively. Note that the `MPI_File_write_all` and `MPI_File_read_all` are collective operations, which means the function will return only when all processes return. However, the side-effect of such blocking operation is limited for getting final results.

6 PDE SOLVER IMPLEMENTATION

6.1 General Setups

In general, the building a template solver is no longer a performance-efficient choice, since the optimization of compiler more unpredictable and details across different dimensions are various, especially for latter MPI/SMP hybrid solver implementation. However, for the same PDE, it is quite obvious that there are many commons among different dimensional version. If I design fully isolated objects for each dimension is an expensive idea, which significantly increases the complexity, redundancy and makes the program harder to maintain the consistency. In this scenario, I chose to follow a pattern called template method pattern, which designed a base object, includes a framework of solving this PDE and basic features required. This pattern has many advantages, such as

(1) Significantly decrease redundancy, complexity of program, the repeated parts are extracted into base objects.
(2) More flexible for designing features specifically for each derived objects. Makes the programer truly focus on the features designed for each object, with no need for worrying the general features of base object.

6.2 Initial/Boundary Condition Object Design

6.2.1 Initial Condition Class. `InitialConditions` is the `namespace` where all initial condition classes located. In the long tern, this it will include initial conditions for heat equation in different dimension space, so as other type of PDEs. For this project, I implemented two classes, `Init_2D` and `Init_3D` for 2D and 3D heat equations. Both of the classes have private member `std::function` objects, that store the function in mathematical form $\varphi(\vec{x})$, such as the equation 12. Default constructors of them are set the $\varphi(\vec{x}) = 0$, once the classes are initialized, the boolean value `isSetUpInit` will be set to `true`. Since inits objects are `friend` of the PDEs objects, user can apply this objects to PDEs objects by calling `SetUpInit`. The syntax `friend` brings several benefits

(1) It will enhance encapsulation, especially in this case, init objects and PDEs object don not need closely work together, exposing internal implementation details to each other would violate the principles of encapsulation.
(2) Also, it helps keep the class interface clean, only exposing the necessary members while keeping the rest the implementation details hidden.

6.2.2 Boundary Condition Class. Boundary Condition is `friend` class of PDE classes that located on cube domain, and can be a tedious and hard implementation, due to there are 6 functions in 3D space and 4 in 2D space. Thus, I had not chose to use virtual inheritance pattern on this implementation. Moreover, there are mainly three type of boundary conditions, Dirichlet, Von Neumman and Robin, also known as first, second and third type of boundary conditions.

In this project, I chose to imply first and second type conditions, which denoted with member variable `std::array<Bool, 2*NumD> isDirchletBC` and other one for `std::array<Bool, 2*NumD> isNeumannBC`. Similarly to init classes, the `SetBC` is also a function for apply this boundary condition to PDE objects and the status of setup is stored as a boolean `isSetUpBC`. However, unlike the initial condition, the Von Neumman boundary conditions need to be update during evolving processes. Thus, an additional member called `UpdateBC` is designed for handling this case. This member function will update the boundaries which is set as Von Neumman conditions and leave the Dirichlet conditions' alone.

Implementing boundary conditions can be a tough task, since the domain of a problem commonly is commonly not regular shape such as cube or cylinder. Especially the derivatives are required in the Von Neumman and Robin conditions.

6.3 PDE Solver Objects Design

For solving a PDE system, it's necessary to store the parameters of basic this PDE. Take Heat Equation [E.q. 4] as an example, the domain $\Omega = [0, 1]^2$, and coefficient $\lambda = 1$. Besides that, there are extra setups needed for the FDTD methods' iteration, including

(1) Determine time and space step sizes and other parameters needed from FDTD.
(2) Exchanging (Communication) between different processors.
    (a) Using blocking MPI communication methods, `MPI_Sendrecv`.
    (b) Using non-blocking MPI communication methods, `MPI_Irecv`, `MPI_Isend`.
(3) Updating (Evolving) function to compute the values of next status in time domain.
    (a) Purely MPI, no threading updating strategy.
    (b) Hybrid of MPI/OpenMP, MPI process's has threading.

6.3.1 Abstract Base Class. The default copy/move constructors, copy/move assignment operators of base object are deleted due to it is abstract base class and only provides a unified design. The constructor, member variables and virtual functions are set as protected, which can be accessed by derived classes exclusively.

Constructor. Constructor takes template arguments of extents as inputs, and using extents to determine domain and grid sizes. According to the general setups in section 3.2, and CFL rules, the D dimension heat equation has time step size

$$\Delta t \leqslant \frac{1}{2\lambda \sum_{i=1}^{D} \frac{1}{\Delta x_i^2}} \tag{18}$$

Moreover, I chose to use the modified CFL [8] shown below

$$\Delta t \leqslant \frac{\min_{i=1}^{D}\{\Delta x_i\}^2}{2D\lambda} \leqslant \frac{1}{2\lambda \sum_{i=1}^{D} \frac{1}{\Delta x_i^2}} \tag{19}$$

The weights and diagonal coefficients are

$$\begin{cases} w_i = \dfrac{\lambda \Delta t}{\Delta x_i^2} \\ diag_i = -2 + \dfrac{\Delta x_i^2}{2\lambda \Delta t} \end{cases} \tag{20}$$

Pure virtual functions . Virtual functions overcome the problems with type-field solution, which I used in earlier deprecated versions, it allowing me to declare functions in the base class in advance, and redefined in each derived class. The compiler and linker will guarantee the correct correspondence between objects and the functions applied to them. The type-field solution has several drawbacks:

(1) Using constructs like `switch-case` and `if-else` requires frequent modifications to the conditional expressions whenever a new type is added.
(2) The reliance on type casting increases the risk of errors, making the program more prone to runtime issues.
(3) As the number of types increases, the codebase becomes increasingly redundant and difficult to maintain.

The `virtual` functions in base class stands for

- `virtual Exchanging` functions mean that the derived class will provide specific definition of different exchange strategies.
- `virtual Evolving` functions shows the derived classes have 2 different hybrid updating strategies.

6.3.2 PDE solver class. Derived template classes PDE solver objects are `Heat_2D<typename T>` and `Heat_3D<typename T>` inherit from `Heat_Base<typename T, size_type NumD>` with level `protected`. Each PDE solver class has integrated solver functions, and stands for two types of methods that are pure MPi parallelism [Sec. 6.4.1] and hybrid parallelism. Moreover, the hybrid parallelistic solver has two type of hybrid strategies which I will demonstrate in latter section and section 6.4.2 and 6.4.3.

Safety. The PDE solvers are unified interface objects with integrated IO features, solvers and functions allow to interacting with boundary conditions and initial condition objects, same as previous sections, I chose to use unique pointer to create objects of boundary and initial conditions which adhere RAII protocol. Also, redefining `virtual` functions of base class using `override final` phrases that means these functions are not safe for overloading if latter program want to override in second derived classes.

Performance. Every solver class has two objects of Cartesian distributed arrays, the updating strategy I was using is called ping-pong strategy or red-black strategy. This requires us to update values between two objects, one of which store the current status, the other for next status. Obviously, this is a memory-hungary strategy, which requires twice as much as the actual memory space needed for storing local array. However, due to our parallelistic topology, the local arrays are quite smaller than the actual scale of problem, more importantly, it increase the efficiency of updating values in one CPU clockwise.

6.4 Parallel Strategies

The MPI-3.x introduced shared memory programming unlike OpenMP library, Independent Software Vendor (ISV) and application libraries need not to be thread-safe No additional OpenMP overhead and problems. In general, the Parallel Programming Models on Hybrid Platforms have 13 types [20].

In this project, I chose to implement three parallel strategies

- Pure MPI: only one MPI process on each core.
- hybrid MPI + OpenMP with non-overlapped communication/computation: inter-node communication OpenMP: inside of each SMP node, MPI only outside of parallel regions of the numerical application code.

- funneled hybrid + Open with overlapped communication/computation: MPI communication by one or a few threads while other threads are computing.

6.4.1 Pure Message Passing Parallel. Pure MPI parallelism is a type of strategy that do not take shared memory programming into considering. With no worry about NUMA structure, pure MPI only needs to determine what is the efficient way for communication, such as

(1) Blocking communication, it is a type of communication with internal barrier, which is only favorable when we need to ensure the latter operations depend on this MPI communication, such as reduction, gather, broadcasting and scattering.
(2) Non-blocking communication, it is the other type of communication with no barrier. This is advantageous in scenarios where subsequent operations do not depend on the completion of the communication, allowing computation and communication to overlap, which can improve overall efficiency.
(3) Remote memory access (RMA), the need for explicit send/receive pairs, RMA operation is directly access memory of other CPUs. This can be very efficient, particularly on systems with hardware support like RDMA (Remote Direct Memory Access), However, on systems without such hardware support, RMA may not provide performance benefits and could be slower compared to traditional message-passing methods.

Thus, this projects focused on blocking and non-blocking MPI communications, since RMA requires the hardware support to gain the benefits. In general, Pure MPI strategy follows algorithm The update function `update_ping_pong()`

Algorithm 9 Pure MPI PDE solver Mechanism

```
The i^th iteration
Determine current time of i^th step.
Exchange ghost values.
Update, store in the next status.
Update boundary conditions at current time.
MPI_Allreduce the local difference check if it is converge
if No Debug then
    Store the current results if in Debug mode.                    # Debug mode
    Print global difference on root process.
end if
Switch current / next arrays.
if glocal difference ⩽ tolerance then
    converge = true, break the iteration.
end if
```

follows the values except ghost values, as the idea of FDM described in equation 6 or more specifically for a D dimension heat equation

$$u^{n+1}_{1,\ldots,d,\ldots,D} = u^{n}_{1,\ldots,d,\ldots,D} + \sum_{d=1}^{D} \frac{\lambda \Delta t}{\Delta x_d^2} \left(u^{n}_{1,\ldots,d+1,\ldots,D} - 2u^{n}_{1,\ldots,d,\ldots,D} + u^{n}_{1,\ldots,d-1,\ldots,D} \right) \tag{21}$$

The figure 3 shows the evolving scheme of 2D heat equations.

Exchange. Blocking communication is applied with `MPI_Sendrecv` function which is a unified operation of an `MPI_Send` to the destination site and an `MPI_Recv` from source site. The communication topology is shown in figure 5, for each process, there are two `MPI_Sendrecv` operations, for the destination and source sites communications.

6.4.2 No comm./comp overlapped Hybrid Parallel. The hybrid programming takes shared memory into consider, and the most simple idea is using OpenMP multi-threads for updating arrays of each MPI process.

In the meanwhile, the update function `update_ping_pong` has a nested loop which is suitable for using `parallel for` syntax. Thus the update function of the first hybrid strategy is `update_ping_ping_omp` with OpenMp pragma `omp for collapse(NumD)`. In order to collect differences from all threads of MPI processes, the local difference of threads are collected to MPI local differences by using pragma `omp critical`, with a `omp barrier` at the end. The boundary condition updates and gathering results or print global difference are set to `single`.

Algorithm 10 Master-only Hybrid with non-overlapped Communication/Computation

```
for iterations do
   #pragma omp single
   MPI communications.
   for for grid values do
      #pragma omp for updates.
   end for
   #pragma omp single Switch and other operations.
end for
```

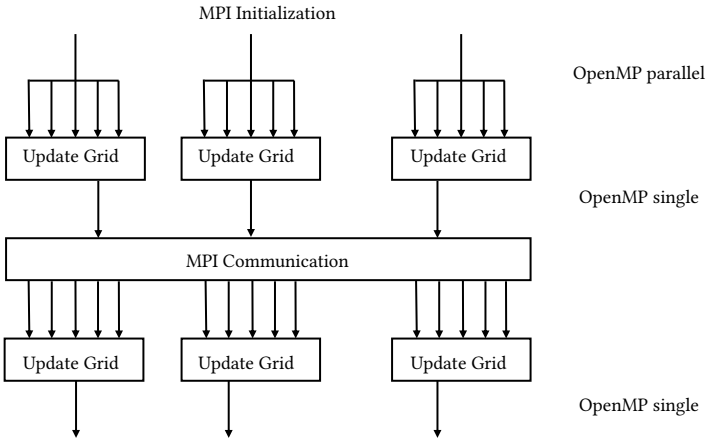


Fig. 6. Schematic view of the master only hybrid strategy with non-overlapping of computation and communication (comm./comp.). Using OpenMP barrier pragma for threads barrier to ensure all grids are updated. Then applied single for applying MPI communications, exchanging ghost data.

Exchange. The exchange method of first hybrid solver is identical with pure MPI exchanger, however, only single thread will call it. Such strategy is called Master-Only Hybrid ,Page 101 in [20], it can be using `omp single` for calling MPI communications rather than using `omp master`. In general the master-only hybrid has such communication scheme shown in figure 6. In the figure, we could clearly see that the details of cooperations between OpenMP and MPI. Using OpenMP single is the easiest way and an efficient method for hybrid communications, due to the `master` requires the root threads (thread id is 0) available for continue working.

6.4.3 Funneled Hybrid Parallel with overlapped comm./comp. Basic on the work in section 6.4.2, the second hybrid strategy is still master-only hybrid but with more optimizations on update function. Due to the inner part of grid is independent with ghost values, thus the update function can be separated the update function into two steps, update the bulk and update the edges.

Algorithm 11 Funneled Master-only Hybirid with overlapped Communication/Computation

```
1:  for iterations do
2:    #pragma omp single
3:    MPI communications. (Isends/Irecvs)
4:    for for bulk points do
5:      Update bulk points.                    # Update the points that do not need ghosts.
6:    end for
7:    MPI_Waitall()                            # Wait all MPI Communications complete
8:    for for edge pints do
9:      update edge values.
10:   end for
11:   #pragma omp single Switch and other operations.
12: end for
```

Comparing to the previous scheme 10, the scheme 11 overlaps part of MPI communications and computation, rather than only take the advantages of shared memory computing.

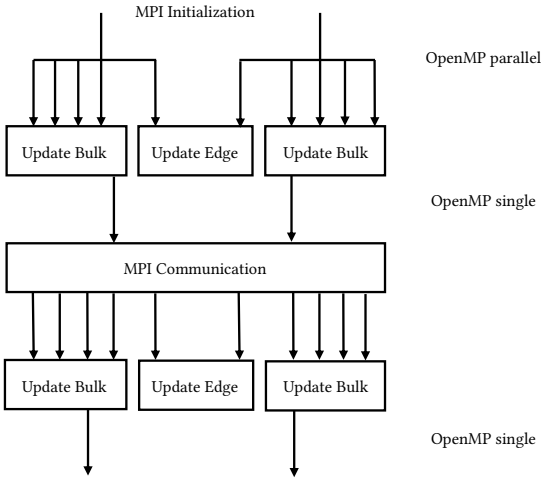


Fig. 7. Scheme: Master only hybrid strategy with computation/communication overlapping of computation and communication (comm./comp.). The OpenMP threads are separated into two parts, one of which will do the MPI non-blocking communication and the other threads will update the bulk. Using MPI wait all to ensure all threads are available, then update the edges.

Also, due to the asynchronous communication is not guaranteed by non-blocking MPI communication, which means that the implementation of multiple overlapping of communication / computation. Besides that, load balancing is a problem as well since the true load of each processes is hidden before we actually run it. Thus, applying multiple overlapping hybrid method can be very hard to do even on simple question, and highly hardware dependent.

6.5 Physics Informed Neural Networks

6.5.1 General Ideas. The ideas of earlier neural networks mainly focus on multiplying matrices and compute the difference between the outputs and the given labels. AlexNet introduced the convolution product into neural networks since ImageNet Challenge 2014 [16], After that, the type of neurons in networks explosively emerged, including long-short term memory (LSTM) neuron, pooling neural and so on. In the recent years, Maziar, Paris and George [15] brings the differential operators into neural networks, which becomes the Physics Informed Neural Networks (PINN).

In the latter years, many research team combined other type of networks with PINN, such as the transformer operator based PINN [9]. However, many papers were focusing on evaluate the quality of solution which is implemented by `Python` or `Julia`, rather than the speed of training. From the performance perspective, the interpreted programming languages like `Pyhon` have lower speed comparing to compiling languages such as `C/C++`. Thus, in this project, I chose to evaluate the training performance on CUDA device implemented by the `C/C++` `Pytorch` library called `Libtorch` [**Libtorch**].

6.5.2 Dataset. The first part of training a neural network is getting dataset. For the simple purpose, I chose to apply the PINN network for Dirichlet boundary condition which is randomly generate the nodes on boundaries and set the value on that point using boundary condition. This implementation is integrated in to the class `dataset` under namespace `PINN`, which can generate training datasets for 2D and 3D heat equation and send the data to a `torch::device`. Moreover, it can produce a validation grid for showing the outputs of trained network.

6.5.3 Structure. The network has the simplest structure, which is a sequential of multiplication of matrices and inputs plus bias (three hidden layers). The activation functions are $\tanh(\cdot)$, and the network loss function is mean square error (MSE). Totally, the loss function follows the setting in section 2.2 which is the summation of $loss_{PDE}$ and $loss_{NN}$ where

$$\begin{cases} loss_{PDE} = \dfrac{\partial^2 \widehat{u}}{\partial x^2} + \dfrac{\partial^2 \widehat{u}}{\partial y^2} + \dfrac{\partial^2 \widehat{u}}{\partial z^2} \\ loss_{NN} = MSE(\widehat{u}, u) \end{cases} \tag{22}$$

In this case, computing the derivative of a output from a large model is expensive in old days. However, with the benefits from GPU parallel and Auto-differentiation [2], we are able to compute the derivatives and treat them as regularizer of neural networks.

7 EXPERIMENTS

7.1 Experimental Setup

I verify the proposed PINN methods on generated dataset, and FDTD methods with hybrid and pure MPI strategies on the server with two Intel (R) Xeon (R) Platinum 9242 CPU nodes (96 cores per node) and 4 NUMA nodes per node. While the dataset is using `std::mt19937` STL random device with given seed 42 [5]. The PINN models I mentioned are trained using train-from-scratch strategy, and maximin number of training times is $1'000'000$ epochs. In the setting of learning rate and optimizer, I chose to use Adam with constant learning speed 10^{-3}. The details of PDEs are determined in previous section 3.2.

Compiling. Compiling the program is also a critical important processes. I chose to use the macros for defining the scale of problems in advance, this is because the compiler will have more aggressive optimizations if it knows more predefined parameters.

Running. For finely manipulate the resource allocation on cluster, following command line 3 is used for this the script arguments `rsc` stands for the resource type such as `numa`, `node` and `socket`, `-report-bindings` is a error message, for showing the details of threads and CPUs tasks allocated on cluster.

```
mpirun --map-by ppr:$ppr:$rsc:pe=$threads --report-bindings <executable> <arguments>
```

Listing 3. main command line for launching program on cluster

and `threads` means the number of threads per resource. The `ppr` is the number of CPU tasks per resource. The last `<argument>` is command line arguments for the program, which is designed by programmer. In this case, I designed three type of arguments

(1) `-S, -s` Strategy, for specifying the pure mpi, hybrid 0 or 1 strategy.
(2) `-F, -f` file name, if this argument is defiend, the results will be stored in the file.
(3) `-H, -h` Helper message, the usage information.
(4) `-V, -v` Showing the version of program.

7.1.1 *Computational Topology.*

The computational topology is critically important when we are programming parallel PDEs solver softwares. Put the strongly speed-dependent data into the slow memory could make entire program slower.

Cluster. The cluster we are using for this project has 2 sockets per compute node, and each CPU has 24 cores with hyper-threads. The Non-Uniform Memory Access (NUMA) nodes are layout as following Accessing the other NUMA node's memory reduces the bandwidth and also the latency, though the bandwidth is commonly high enough, the latency can increase by 30% to 400% [13]. This latency becomes dangerous when writing shared memory parallel programs.

7.2 Comparison on single node

On single node, the CPUs are connected by high-bandwidth, low-latency internal bus which is faster than connection between nodes. However, for the 4 total NUMA nodes per compute node, memory accessing between them has higher latency than cache. Thus, the first tests set were run on the platform with single node, to evaluate the parallelistic performance of heat equation on 2 and 3 dimension spaces with 3 strategies.

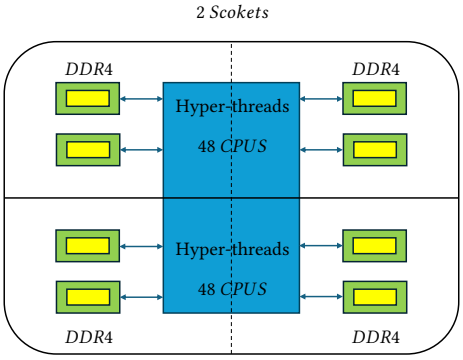


Fig. 8. NUMA topology of single node on Cluster

7.2.1 Strong Scaling. Figure 9 visualizes the comparison of my proposed parallelistic program using pure MPI on two dimension space heat euqation with the number of CPUs and various problem scales. Overall, the more CPUs brings more performance among all scales from 512^2 to the $32'768^2$ but can not break the speedup limits. Compared with large scale bigger than 4096^2, the light problems has less speedup as the number of CPUs increases. By seeing the trend of speedup ratios drop as the CPU gets more, the trend can be readily discovered which is the as the scale of problems gets larger, the latter it will have performance-dropping. Once the problem size is large enough, (4096^2 and larger), the solvers can get the more benefits from more CPUs.

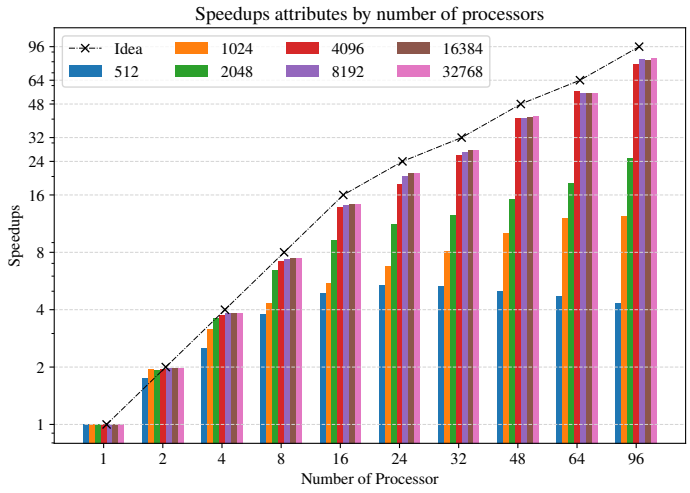
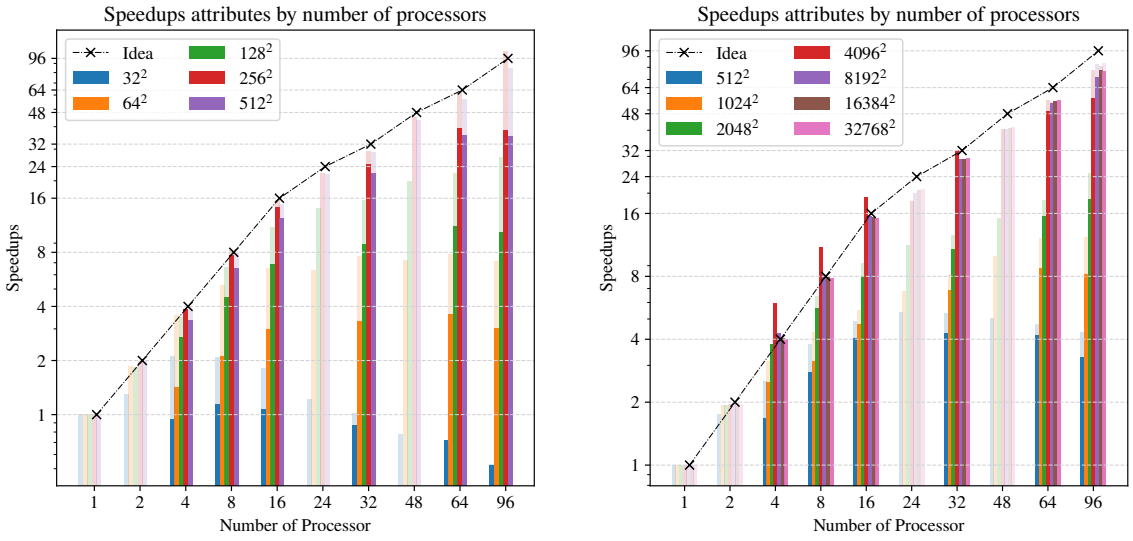


Fig. 9. Comparison of speedup ratios of strong scaling tests of pure MPI parallelized program. The investigated problem scales are the power of 2, exponents ranging from 9 to 15. The number of CPUs are also set as power of 2, with additional numbers 24, 48 and 96 matched the topologies of CPU.

On the other hand, I also include some unconventionaly number of CPUs in scaling tests such as 24, 48 and 96 for comparison. and the results are also shown in the figure 9. From this figure, it is hard to tell the difference of these where it ought to indicate some information about its NUMA structure. This is because the MPI communication does

not strongly effected by memory structure, while the hybrid does. Figure 10 shows the difference, the hybrid strategy brings lower performance with small scale across all CPUs and approximately identical in the large scale cases. The most visible change in figure 10(a) is that the speedup ratio of problem with scale 4096^2 exceded the limits on 4, 8 and 16 CPUs which is 1, 2, 4 threads of each MPI process. We can also see that when the number of threads is 8 and 4 MPI processes, the performances on large scale is better than pure MPI parallelism.



(a) No overlapping comm./comp.

(b) With overlapping comm./comp.

Fig. 10. Comparison of speedup ratios of strong scaling tests of mater-only parallelized program with overlapping and no overlapping of computation and communication. The vague background is the results of pure MPI parallelization from figure 9 and problems sclaes are identical as well. The number of threads are set to 1, 2, 4, 8, 16, and 24, tasks per CPU are 1, 2 and 4.

For the other funneled hybrid parallelization, the figure 10(b) in the appendix shows the details of results, this strategy has nearly idential performance of master only with no overlapping on large problem scales. However, the behaviour of it on small scales has a different pattern. This indicates that the overlappings of computation and communication are not as good as previous one, which means the overload management is not well on these tests.

Superliner Speedup. Conventionally, the actuall speedup won't excedes the theoratical predictions of Amdalh's law. However, the scaling of two hybrid programs did exceded the limits but exclusively on the problem scale 4096^2 and 1 to 8 threads of each 4 MPI processes. Considering the details of the CPU used for these tests,

- It has 4 NUMA node per CPU and once of which has 12 CPUs with 2 threads.
- It has `32KB` L1 data and L1 instruction cache, `1024KB` L2 cache and `36608KB` L3 cache.

The data type for this solver is `Float` which takes 4 bytes, and 4096^2 `Float` numbers takes 64 `MB` to store. In the case of 4 MPI processes, each process own a quater of number which uses 16 `MB` for handling sub-problems. On the other hand, the L3 cache is 36608 `KB` = 35.75 `MB` for 2 NUMA node due to the hyper-thread, which is just bigger than the sub-problem scale 16 `MB`. When the problem size gets larger, such as 8096^2 which takes 128 `MB` to store the sub-problem. In such case, the L3 case is no longer able to hold it, thus part of the numbers will be stored in the DDR4 memories

Table 1. Weak Scaling on Single Node of 2D Heat Equation

Strategy	Size	Number of CPUs 4	16	64	$f_p(\%)$
Pure MPI		4.006	12.497	47.849	75.1
No Overlap	512^2	2.876	11.206	42.754	67.0
With Overlap		3.173	10.818	42.282	66.2
Pure MPI		3.838	9.304	33.707	53.2
No Overlap	1024^2	3.947	12.995	33.447	54.1
With Overlap		4.024	12.932	33.361	54.0
Pure MPI		2.376	8.245	31.203	49.0
No Overlap	2048^2	3.874	8.972	31.510	49.8
With Overlap		3.740	8.989	31.430	49.7
Pure MPI		3.543	8.245	31.203	77.5
No Overlap	4096^2	3.953	13.799	49.515	78.0
With Overlap		3.948	13.800	49.989	78.7

which is lower bandwidth and higher latency than cache. Moreover, due to the CPU enables hyper-threading, a NUMA node actually holds 12 CPUs, which makes the superlinear speedup disappear when the threads is 16 and 24.

7.2.2 Weak Scaling. Table 1 lists the weak scaling comparison of three different parallel strategies. Overall, both three strategies have good weak scaling results across all problem sizes. For example, according to the Gaustafsson's Law 2.2, the program using overlapped strategy has sequential fraction

$$f_s = \frac{49.989 - 64}{1 - 63} \approx 0.222$$

on the problem scale 4096^2 with 64 CPUs and $f_s \approx 0.147$ with 16 CPUs. For the small size problems, like 512^2, the pure MPI has significantly higher weak scaling efficiency comparing to the other two, more specifically about 7.8%, 7.5% higher in 64, 16 CPUs, and 22.5% higher in 4 CPUs. However, as the scale gets larger, this advantages gradually disappear and even worse than other two. Especially when 4096^2 on 16 and 64 cores, the forward leap of hybrid versions gets larger, about 34.4% and 28.1% higher than pure MPI respectively.

On the other hand, the superlinear speedup appears for the problem size with 512^2 and 1024^2 on 4 CPUs. This is because the sub-problem for each CPUs only take 512 KB and 1024 KB to store, which is the L2 cache size of the Xeon Platinum 9242. Thus the four processes could take the performance advantages of L2 cache individual for each CPU. Once the problem size gets larger, such as 2048^2, part of the problem will be allocated on L2 cache and other parts on L3 cache. These memory allocations requires extra synchronization between cache accessing, which drop the performance down. However, as the size gets to 4096^2, the sub-problem will be fully make use of L3 cache, which require less synchronization of cache, the weak scaling goes up but we can not see the superlinear speedup again.

Moreover, since Gaustafsson' theorem 2.2 gives us a linear module for valuating the parallel problem. Thus I chose to use the linear model from equation 8 as a linear polymonimal fitting. Considering the original point has to be on the line, which stantds for $f_s + f_p = 1$, I manually create an extended version of speedups by concatenate a inverse negative one to ensure their means are 0 which makes the fitting line across the $(0, 0)$. Eventually, the results of f_p are shown in the table 1 as well. These fitting lines also indicate that the hybrid parallelized versions has more parallel fraction than pure MPI when the scale of problem gets larger than 512, which means that hybrid strategies reduced the cost of synchronization and brought higher efficiency.

7.3 Comparison on multi-node

Running programs on multi-node also utilized the command line 3 to specify the resource allocation on cluster. In this setup, I ran the tests on 4 nodes with total 16 NUMA nodes or 384 CPUs, all other settings are maintained from the single-node tests.

7.3.1 Strong Scaling. Figure 11 demonstrates speedup ratio of all candidates for test and their relationships with the number of CPUs. In the big picture, it is easy to determine that the small scale problems (smaller than 4096^2) had pool speedup performances on these settings. Especially for the 512^2, 1024^2 and 2048^2, parallelization even brought worse performance in the end, which is strongly different from the big scales. In the large scale problems, the most large two maintained good strong scaling performances at the end, but the pure MPI failed on launching the program when using 384 processes. Moeover, the problem with 4096^2 numbers show superlinear speedup on 4, 8 and 16 processes which is similar to single-node hybrid tests. However, unlike gaining benefits from shared memory operations on single NUMA node, these superlinear speedup came from the node structure. Where the pure MPI processes were allocated on four different CPU, which has individual CPU clock cycle. Thus, the processes can behave like operating on shraed memory model. In addition, the patterns of the strong scaling are different from the Amdalh'S law 2.1 demonstrates. Which has

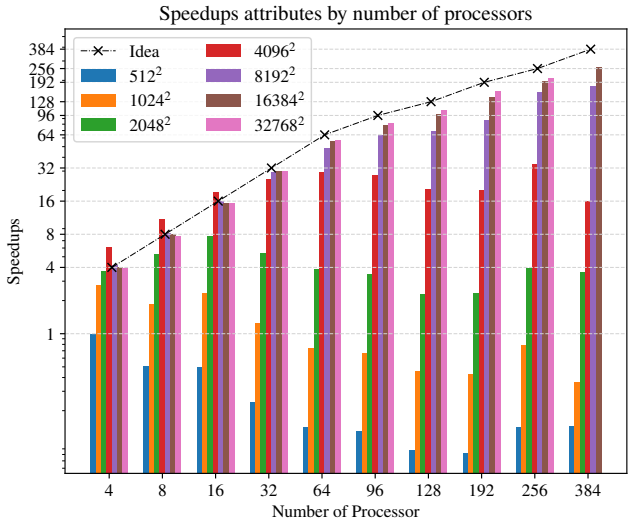
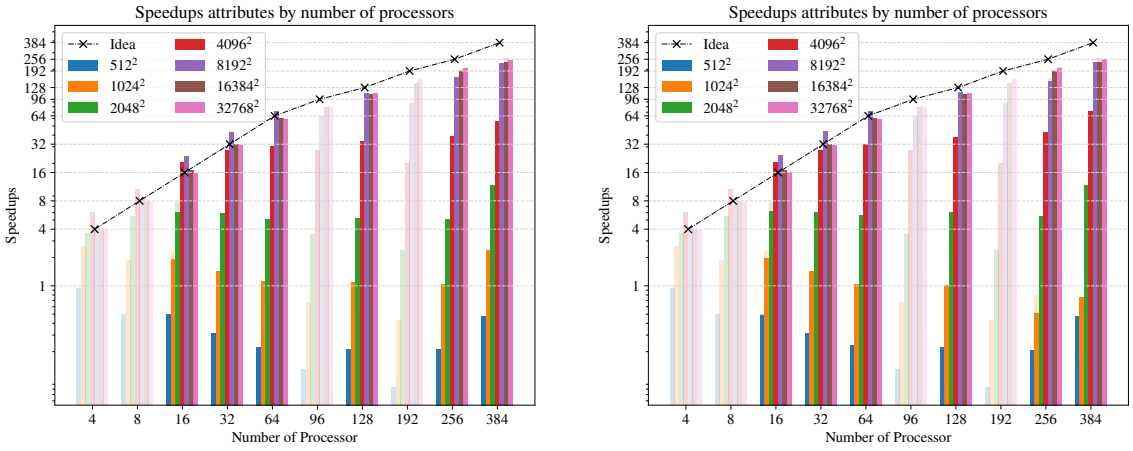


Fig. 11. Comparison of speedup ratios of strong scaling tests of pure MPI parallelized program. The investigated problem scales are the power of 2, exponents ranging from 9 to 15. The number of CPUs are also set as power of 2, with additional numbers 96, 192 and 384 matched the topologies of CPU.

may visible exceptional performance decreases, due to the topologies of CPUs, especially when using 32, 48 and 96 CPUs per node.

On the other hand, based on comparison of the strong scaling speedup ratios and the amount of resources, two hybrid versions performed better than pure MPI parallel and the results are shown in the figure 12. From the figure 12(a) and 12(b), we could see that the superlinear speedup appears on the scale 4096^2 and 8019^2 which are effected by the previously mentioned two reasons, the high performance local NUMA node shared memory and individual CPU

clock cycle. More specifically, 8092^2 `Float` numbers takes 256 MB to store, 64 MB on each node and 16 MB per NUMA node, which just fits it's L3 cache size.



(a) No overlapping comm./comp.

(b) With overlapping comm./comp.

Fig. 12. Comparison of speedup ratios of strong scaling tests on 4 nodes of mater-only parallelized program with overlapping and no overlapping of computation and communication. The vague background is the results of pure MPI parallelization from figure 11 and problems sclaes are identical as well. The number of threads are set to 1, 2, 4, 8, 16, and 24, tasks per CPU are 1, 2 and 4.

As the amount of resources increase, the speedups are not increase as expected for all three parallel models. However, hybrid parallel models were successfully launched on the 384 CPUs, and also brought more performance for all tested number of CPUs. These shows that the advantages of combination of shared memory programming and message-passing programming.

7.3.2 Weak Scaling. Table 2 lists the weak scaling comparison results of multi-node tests. Overall, the hybrid models have better scaling speedup ratios than pure MPI model on the size larger than 512^2. Although the speedups of pure MPI model are better than hybrid models on 16 CPUs with 2048^2 problem size, the hybrid model with no comm./comp. overlapping still have better performance when testing is running on 64 and 256 CPUs. Controversially, the hybrid with no overlapping has better results comparing to the one with overlapping, and we have opposite results when the scale gets larger.

On the other hand, similar to previous section 7.2.2, the fraction of parallelizable is approximated by linder regression based on the model proposed in theorem 2.2. As the table 2 lists results, the fraction of parallelizable increases as the size gets larger. Moreover, the overlapping has the most large fraction then the model with no overlapping, and both of then are significantly larger than pure MPI model.

There are some important performance leap forward due to the memory structure mentioned in previous section 7.2.2, the L3 cache for single NUMA node just fits the problem with 2048^2 `Float` numbers.

- The first one is the number of CPUs increases from 16 to 64, in the case of the problem size is 1024^2 running on hybrid models, the problem scale of each NUMA node increases from 1024^2 to 2048^2 which makes the program requires less cache synchronization and brings about 52.6% more scaling speedup performance.

Table 2. Weak Scaling on Multi-node of 2D Heat Equation

Strategy	Size	Number of CPUs 4	16	64	256	$f_p(\%)$
Pure MPI		3.300	10.379	26.000	124.375	48.2
No Overlap	512^2	-	8.094	25.931	127.648	49.3
With Overlap		-	8.386	27.126	116.951	45.5
Pure MPI		3.848	13.037	30.143	-	49.3
No Overlap	1024^2	-	13.702	44.040	-	69.8
With Overlap		-	13.834	44.090	-	69.9
Pure MPI		3.787	9.172	32.013	-	50.6
No Overlap	2048^2	-	13.936	34.368	-	55.7
With Overlap		-	14.274	34.414	-	55.9
Pure MPI		3.884	13.859	51.041	-	80.2
No Overlap	4096^2	-	15.315	53.157	-	83.8
With Overlap		-	15.294	53.401	-	84.2

- The second is the size of problems increas from 1024^2 to 2048^2 when the number of CPUs is 16. With such doubled size, the program can make more use of L3 cache on single NUMA node and brings about speedup 46.6% improvements.

Both of the improvements are about 50%, these results indicates that the hybrid strategies I was implementing are effective, also, the overlapping of computation and communication promoted the speedup further.

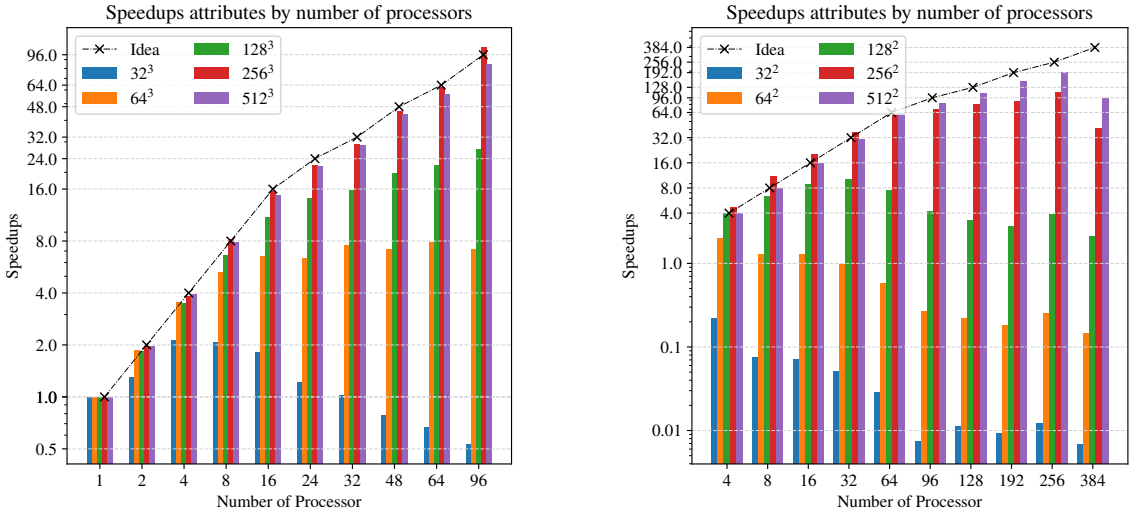
7.4 Comparison on Dimension

7.4.1 Strong Scaling. The figures 13 show the results of solving 3D heat equation with Dirichlet boundary conditions specified by euqations 15 in single precision. Conventionally, solving a 3D problem is extreamly computational expensive comparing to solve a 2D problem and hard to be parallelized. Overall, both of them have great strong sclaing speedups in a reasonable problem size and amount of resources. However, considering the size of the problems are common cube of number which is smaller than 1024, we can barely get solution with fine quality in 3D dimension case.

On the other hand, I specify the maximun scale of problem is 512^3 rather than 1024^3. This is because the pattern of accessing memories is $O(n^3)$ space complexity which leads more time consumption per epoch, although the problem has identical scale $1024^3 = 32768^2$. It also makes us harder to see a large superlinear speedup occurs comparing to 2D problems. In this scenario, only the figure 13(b) shows that the problem with scale 256^3 has superlinear speedup on 8, 16 and 32 CPUs due to the separated nodes has dependent CPU clock whic increase the rate of hitting cache.

The figure 14 shows the details of hybrid parallel models on single and four nodes. Overall, the figures 14(a) and 14(b) show that both hybrid model have worse performance on single node. Even with the benefits of caching, the 256^3 scale problem still a little bit loss on speedup on 4, 8 and 16 CPUs. However the figures 14(c) and 14(b) shows completely different results. On the multi-nodes test, overall, all hybrid models have better performance comparing to pure MPI parallelization. Especially for the largest problem 512^3 which the pure MPI scaling speedup ratio is 96 under 384 CPUs, the hybrid models gained approximately 130 and 140 speedups which is about 30% 40% improvements. Moreover, the problem with 256^3 scale investigated on 8 16 and 32 CPUs reached superlinear speedup at the end, which were making use of L3 cache and asynchronized CPU clock cycle.

As for the promotion gets from the cache and CPU asynchronization, we can discovered that the 2D problems got less benefits while the 3D problems got bigger. This is the 3D problem relys more on the memory accessing, and the shared memory parallelization can do optimizations better than MPI.



(a) Strong scaling on single node

(b) Strong scaling on four node

Fig. 13. Comparison of speedup ratios of strong scaling tests pure MPI models on 4 nodes of and 1 node. The problem sizes are set to 32^3, 64^3, 128^3, 256^3 and 512^3 with single precision.

7.4.2 Weak Scaling. The weak scaling results of 3 dimension problems are listed in the table 3. For the single node scaling speedups, we could see that the pure MPI model has better performance over all problem scales and amount of resources. However, for the multi-node tests, the hybrid models have approximately same or better performance than pue MPI model. Especially for the non-overlapped hybrid model.

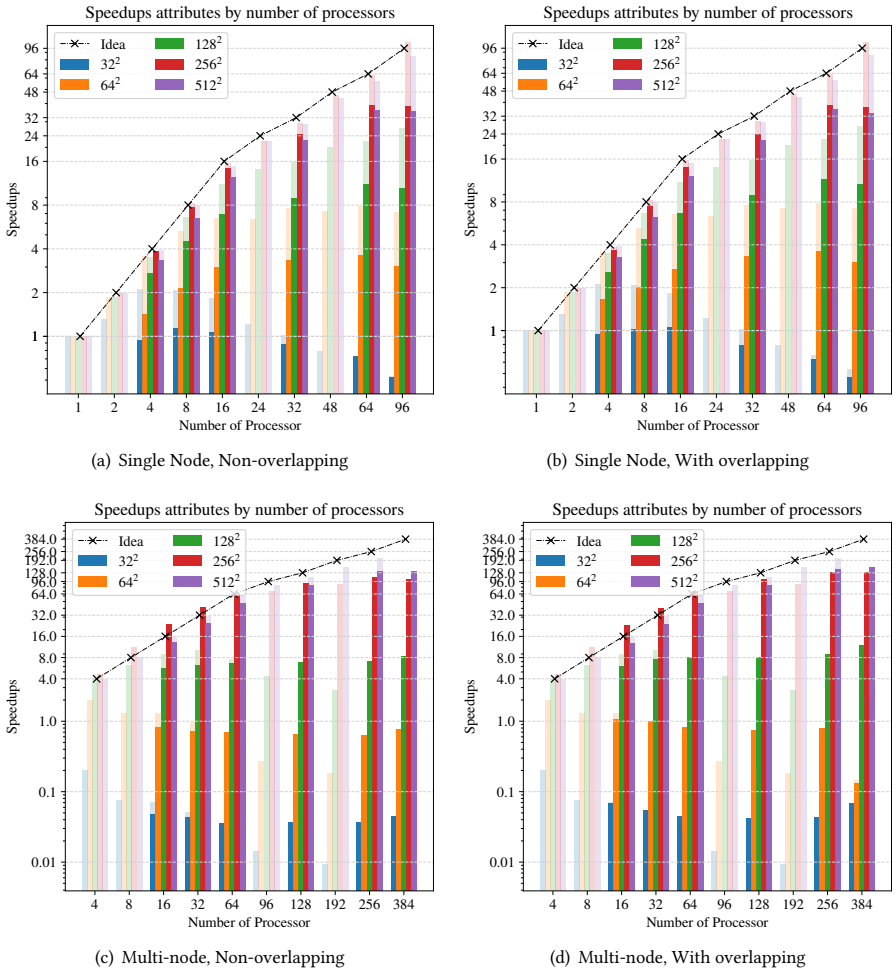
Table 3. Weak Scaling of 3D Heat Equation

Strategy	Size	Number of CPUs 8	64	$64_{\text{Multi-node}}$	$f_p(\%)$	$f_p(\%)_{\text{Multi-node}}$
Pure MPI		5.243	26.405	9.078	41.6	14.2
No Overlap	32^3	2.124	13.386	8.094	21.0	12.7
With Overlap		2.014	13.884	9.586	21.8	39.7
Pure MPI		7.937	32.034	30.106	50.8	47.1
No Overlap	64^3	5.393	20.007	33.460	31.8	52.3
With Overlap		5.200	19.558	35.254	31.1	31.6
Pure MPI		3.513	24.239	25.428	38.0	39.7
No Overlap	128^3	3.303	15.235	35.254	24.1	55.1
With Overlap		3.187	15.107	20.096	23.9	31.4

On the other hand, comparing the results from 2 dimension versions shown in table 1 and 2. It is readily to determine that the weak scaling speedup of 3 dimension problems are worse than 2 dimensions. Which means that the fraction of parallelizable parts in 3D problems are significantly lower than 2D version.

7.5 Comparison on Accuracy, with PINN

Scalability is important for parallel computing to be effective, so as the quality of results. In this section, the major comparison are about comparing the quality of numerical solutions of 2D, 3D FDTD solvers and a state-of-art method



(a) Single Node, Non-overlapping

(b) Single Node, With overlapping

(c) Multi-node, Non-overlapping

(d) Multi-node, With overlapping

Fig. 14. Comparison of strong scaling speedup ratios of hybrid models on 4 compute nodes for solving 3D heat euqations. The problem scales are set to the cude of 32, 64, 128, 256 and 512 with single precision. The value background is the strong scaling results of figure 13.

PINN. The scale of memory space, tolerance in settings, converging time / training time and quality of results are considered as metrics in this section.

7.5.1 Memory Space. The memory space used for solving the problem is critically important, not only because the caching, vectorization problems in CPU parallelization or graphical memory usage in GPU parallelization. One of the highlight of neural network is solving same scale problem and getting results the have same quality.

In the FDTD method, the memory space used for ping-pong strategy requires doubled memory space for storing single array. For parallelized FDTM, the memory cost of storing juck values and ghost values increases, which is dependent with the size of sub-array in each processes. For a 2D problem with scale NX by NY , the memory usage is

at least

$$\frac{(NX+2)\times(NY+2)}{1024^2}\times \texttt{sizeof(maintype)}\quad \texttt{(MB)} \tag{23}$$

For the PINN method, the usage of memory is associated with the cost of optimizer and initialization of `Libtorch` library. Besides the part independent with problem itself, the usage majorly for storing the weights of neural networks. The other key difference between is that although the module has predefined structure, it can make predictions on various size of grids, where the FDTD requires user to specify the scale of problem in advance. The structure of PINN I was using is shown in the figure 2 with a parameter called hidden size denoted with h, which is the number of neurons in every hidden layer. Thus, for a N dimension PINN with single output, the amount of memory needed for storing the netowrk has formula

$$\frac{2h^2+(3+N)h+1}{1024^2}\times \texttt{sizeof(maintype)}\quad \texttt{(MB)} \tag{24}$$

and the maintype means the value type used for weights of network.

7.5.2 Accuracy. In typical numerical method like FDM and FDTD, since the tolerance of the FDTD is the highest precision of the numerical approximations. Thus, when we tried to represent numbers using arithmetic in binary, decimal or hexadecimal, truncation always affects the precision of every number, or so called as round-off-error.

Round-off Error. In IEEE-754 [14] standards, a 32-bit floating pointer number, single precision, obligatorily represented with 23-bit mantissa, 8-bit exponent and 1-bit for sign. Round-off errors are a manifestation of the fact that on a digital computer, which is unavoidable in numerical computations. In such case, the precision of the number depends on how many bytes are used to store single number. For instance, a float32 number provides $2^{-23}\approx 1.2\times 10^{-7}$, and a double precision number gives $2^{-53}\approx 2.2\times 10^{-16}$, such number is called machine ϵ which is the smallest number the machine can represent with given format.

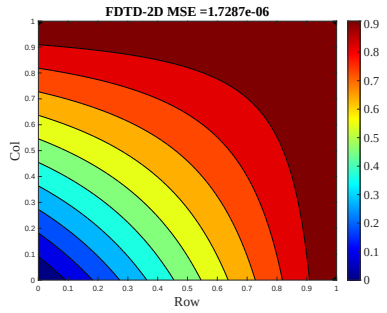
Truncation Error. Truncation error is the other type of error comes from the precision loss of numerical approximations to the derivatives. For example, using first order forward difference with stepsize Δ for approximating the derivative is based on the Taylor Series, and the truncation error is $O(\Delta x^2)$ [6].

PINN. For the neural network, the round-off and truncation errors are barely mentioned in researches, due to the complexity of network, doing such numerical analysis is hard commonly. In stead, the conventional method for evaluation is considering the fitting is over/under, the quality of optimizer, the memory space and the quality of the output of trained model.

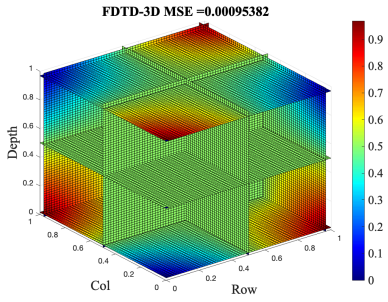
In numerical methods I investigated, the FDTDs are conventionally using double precision number so that the programs can treat extremely large and small numbers simultaneously in the same computation, without worrying about the round-off errors. However, as mentioned, fp32 and fp16 are also popular use in scientific computing, especially in machine learning training process. While the lasted training GPUs are integrated compute accelerate unit for low precision floating numbers. [4]. How to define the the hyber-parameters is another deep topic for computer science researching, in this project, the important hyper-parameters are the tolerance 10^{-2} and 10^{-4}, and the h which is the size of hidden layers.

The figures 15 visualize some of the solutions/preridctions produced by FDTD/PINN with grid size 128 in each dimension. Overall, we can see both of them have good performance on approximating the exact solutions. Since the tolerance of loss function in PINN model is 10^{-4} and the hidden size $h=10$, the predictions of models are fairly well. On

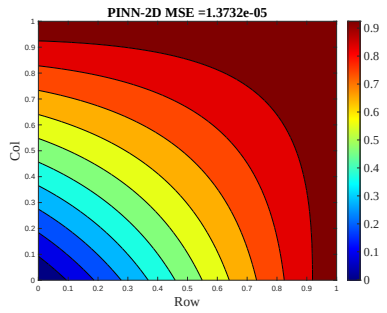
the other hand, the results of FDTD in the figure 15(a) and 15(b) shown that, although the tolerance of FDTD methods are set to 10^{-8}, the final approximations are still not likely to reach the precision required by user.



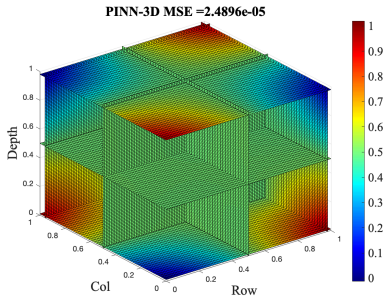
(a) FDTD numerical approximation of 2D Heat Equation, with convergence tolerance 10^{-4}.



(b) FDTD numerical approximation of 3D Heat Equation, with convergence tolerance 10^{-4}.



(c) 2D-PINN model's prediction of 2D Heat equation, with training tolerance 10^{-4}.



(d) 3D-PINN model's prediction of 3D Heat equation, with training tolerance 10^{-4}.

Fig. 15. Comparison of Accuracy of numerical approximations and predictions of FDTD and PINN for 2D and 3D heat euqations. The grid size of FDTD models is 128 in each dimension, and the boundary sample size of PINN is 128 in each dimension which means the PINN is trained on same amount of data used in FDTD.

7.5.3 Training time. In order to produce a usable results, the FDTD methods commonly requires many times to converge due to the temporal step size is linearly associated with the square of grid resolution. In the other hand, a parallel requires fine-tunning for every cluster system inorder to get a high-efficient scaling, which we investigated in above sections. However, the optimal choice is hard to determine in advance as the complexity of problem increases. Figure 16 shows the results of time spending on training/iterating the 2D and 3D PINN/FDTD models on compute nodes, with different tolerance 10^{-2} and 10^{-4}.

Overall, the FDTD models required at least 10 times more memory to store the results during evolving process. Besides that, train the networks are also an efficient approach comparing to iterating matrices of FDTD which takes thousands of seconds on 384 CPUs parallelized systems. The tolerance directly shown the quality of the solution that

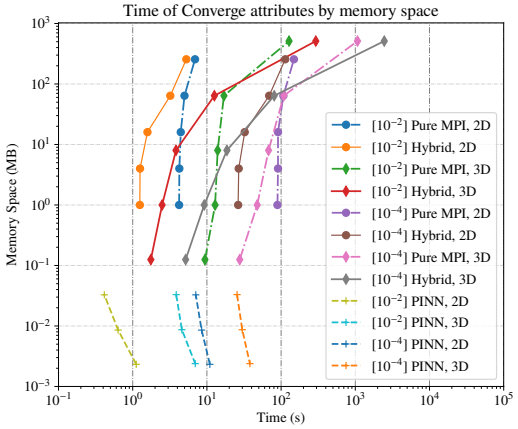


Fig. 16. Comparison of memory usage of FDTD and PINN attributed by time of trainning/converging time consumption for both 2 dimension and 3 dimension heat equations with single precision. The predefined tolerance for training and converging are 10^{-2} and 10^{-4}. FDTD was testing on four compute nodes (384 CPUs), and using pure MPI and non-overlapping hybrid models. The PINN models are trained on single NVIDIA RTX 4090 graphics card.

the FDTD model will produce, where the output of PINN is not predictable in advance. In this case, the predictions have higher precision than the predefined tolerance, commonly 100 times more precise.

8 DISCUSSION

The major part of this project is to implement a efficient FDM PDE solver on parallel systems and validate the quality of the performance on accuracy, scaling. Due to the time constrains, as well as the lack of hardware, software and financial support, I have not do full profiling of the FDTD and PINN models but only partial analysis using `gperftools` [10].

8.1 Further Work

Though out the FDTD method, various of `C++` features are used for designed a safer and faster library, also a number of MPI concepts have been used along with the `C++`, `OpenMP` features to designed a fast, efficient and user-friendly implementation. Also, a variety of `C++` concepts and `Libtorch` concepts are used for implementing a fast PINN solver for PDEs.

However, this project provides precise, efficient and fast PDE solver for cluster environment, there are still many things need to do.

Memory. FDTD is implemented by ping-pong update strategy which is designed for higher cache hit rate. However, for some system, the memory is limited resource for storing doubled size of the required solutions. In such case, using single matrix to store the grid and updating it using Gauss-Seidel or red-black strategy can leads more memory efficiency.

Workload Management. Although the current workload is managed by OpenMP `singe`, `barrier`, `critical` and others. It's still hard to know that exact workload for each distributed CPUs. In general, using `sections`, `tasks` can make it better. However, in latter version MPI-3.X, it support MPI shared memory programming without the worries of OpenMP threads. Thus, using Hybrid MPI + MPI with shared memory nodes can have better performance.

PDEs. Due to the time constrainment, I had not implement the full weak/strong scaling testing of Von Neumman Boundary Conditions. In further work, applying the different type of boundary conditions efficiently is also worth to do. Also, the domain of PDE in this project was set to the $[0, 1]^d$ in d-dimension space, which is not likely happen in real case. Thus, implementing an efficient mesh generator, and mesh-based FDM method is an other work direction.

PINN. Implementing neural network in `C++` allows us have better performance. Also, parallel training of neural network on GPU is implemented by MPI. Thus, using MPI and Libtorch to implement a parallelized neural network training program is also valuable as a comparison for numerical methods.

8.2 Conclusion

In this project, the ideas were creating a versatile library which includes a high efficient matrix library that can be used in parallelized environment, different MPI parallel environment such as Cartesian topology, Cylinder topology etc., a generalized user-friendly PDE solver based on FDM algorithm which allows user specifying the initial conditions, the type of boundary conditions and the parallel strategies such a pure MPI or hybrid of MPI/SMP.

The first is to implement of template multi-dimension array object is created with deep features which have direct access to memory. Subsequently, a user interface template object of multi-dimension array created for specific features which will be used in latter FDTD and PINN models. On the other, an environment of MPI communication object was build for save initializing and finalizing MPI environment. Moreover, MPI Cartesian Topology was integrated into a object aligned with other features for distributing template multidimensional array. Enhance, defining distributed IO

based on template array called `gather`, MPI-IO is also constructed for parallel IO which is designed for saving/loading large scale problems without communication and saving memory usage.

The FDTD models are implemented in three ways, pure MPI parallelism, master only with no computation / communication overlapping and funneled master only with overlap of communication and computation. Also provide choice of two type of boundary condition, Dirichlet and Von Neumman Boundary Condition. A large proportion of this project is to build the library and running weak/strong scaling test to determine the differences between three strategies. In general, on single node or on single chip, pure MPI has similar performance comparing to OpenMP. On multi-nodes, with a reasonable resources allocation, the hybrid strategies have better performance than pure MPI. These results also indicate that managing the overload of each processes is difficult.

The neural network is the major role of the state-of-art comparison. This project avoid the conventional approach for implementing neural networks using `Python` by using `C++`. For comparing the performance, efficiency, I constructed a neural network with modified loss function and the training strategy in `C++` which is called Physics Informed Neural Network.

Overall, this projects provides a polymorphic-library of finite difference methods, a fast and stable parallel FDTD PDE solver with three type parallelization for different scenarios, a comparison between one of the most popular simulation tool, PINN.

REFERENCES

[1] Gene M. Amdahl. 2007. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, n.j., apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, 12, 3, 19–20. DOI: 10.1109/N-SSC.2007.4785615.

[2] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18, 1, (Jan. 2017), 5595–5637.

[3] OpenMP Architecture Review Board. 2023. *OpenMP Application Programming Interface*. (5.2 ed.). Accessed: 2024-09-07. OpenMP Architecture Review Board. https://www.openmp.org/specifications/.

[4] NVIDIA Corporation. 2024. *NVIDIA H200 Tensor Core GPU*. HBM2 High Bandwidth Memory Technology. NVIDIA. https://www.nvidia.com/en-us/data-center/h200/.

[5] cppreference.com. 2024. C++ documentation: random number generation. https://en.cppreference.com/w/cpp/numeric/random. Accessed: 2024-09-07. (2024).

[6] Germund Dahlquist and ke Bjrck. 2008. *Numerical Methods in Scientific Computing: Volume 1*. Society for Industrial and Applied Mathematics, USA. ISBN: 0898716446.

[7] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard*. (Version 4.0 ed.). Accessed: 2024-09-07. Message Passing Interface Forum. https://www.mpi-forum.org/docs/.

[8] Nickolay Y. Gnedin, Vadim A. Semenov, and Andrey V. Kravtsov. 2018. Enforcing the courant–friedrichs–lewy condition in explicitly conservative local time stepping schemes. *Journal of Computational Physics*, 359, 93–105. DOI: https://doi.org/10.1016/j.jcp.2018.01.008.

[9] Somdatta Goswami, Katiana Kontolati, Michael D. Shields, and George Em Karniadakis. 2022. Deep transfer operator learning for partial differential equations under conditional shift. *Nature Machine Intelligence*, 4, 12, (Dec. 2022), 1155–1164. DOI: 10.1038/s42256-022-00569-2.

[10] gperftools Development Team. 2024. Gperftools: cpu profiling with `CPUPROFILE`. https://gperftools.github.io/gperftools/cpuprofile.html. Accessed: 2024-09-07. (2024).

[11] David Padua, editor. 2011. *Moore's law. Encyclopedia of Parallel Computing*. Springer US, Boston, MA, 1177–1184. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_81.

[12] John L. Gustafson. 1988. Reevaluating amdahl's law. *Commun. ACM*, 31, 5, (May 1988), 532–533. DOI: 10.1145/42411.42415.

[13] MAP55621-HPC Hardware and Architectures. 2024. Trinity college dublin lecture 14. HPCHardware_14.pdf. Accessed: 2024-09-07. (2024).

[14] William Kahan. 1998. IEEE 754: An Interview Saga. Tech. rep. Accessed: 2024-09-07. University of California, Berkeley. https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF.

[15] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. 2021. Physics-informed machine learning. *Nature Reviews Physics*, 3, 6, (June 2021), 422–440. DOI: 10.1038/s42254-021-00314-5.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60, 6, (May 2017), 84–90. DOI: 10.1145/3065386.

[17] Boost C++ Libraries. 2024. *Boost.MPI: C++ library for message passing*. Version 1.82.0. Boost. https://www.boost.org/doc/libs/release/doc/html/mpi.html.

[18] Nicholas Metropolis and Stanislaw Ulam. 1949. *The Monte Carlo Method*. Number 247. Vol. 44. Journal of the American Statistical Association, 335–341.

[19] MIT OpenCourseWare. 2023. Matrix calculus for machine learning and beyond. https://ocw.mit.edu/courses/18-s096-matrix-calculus-for-machine-learning-and-beyond-january-iap-2023/. Accessed: 2024-09-07. (2023).

[20] OpenMP. 2013. Sc13 tutorial: hybrid mpi/openmp parallel programming. https://www.openmp.org/events/sc13-tutorial-hybrid-mpi-openmp-parallel-programming/. Accessed: 2024-09-07. (2013).

[21] Gokul Yenduri et al. 2024. Gpt (generative pre-trained transformer)— a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions. *IEEE Access*, 12, 54608–54649. DOI: 10.1109/ACCESS.2024.3389497.

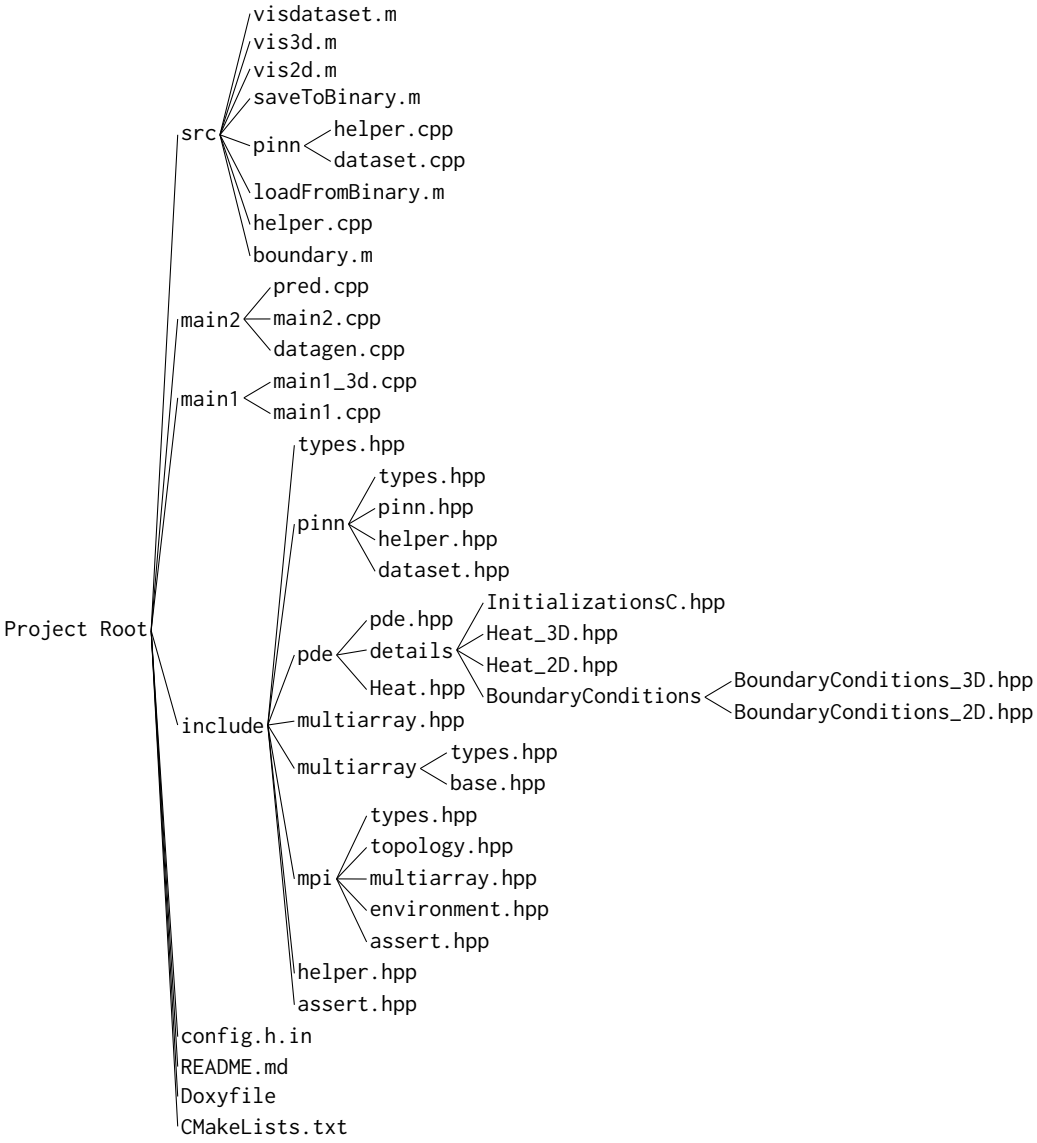
A IMPLEMENTATION DETAILS

The network structure of PINN is shown in the figure 2, that is the h is the number of neuraons in each hidden layers. The input size of PINN is depending on the dimension of heat equation, as well as the dataset size. The Adam optimizer has a learning rate without decay strategy. This is a small model, thus I did not chose to use batch size for these training tasks.

Table 4. Configuration of PINN

Module	**value**
Dimension	2, 3
tolerance	0.01, 0.0001
Dense	h
Data type	`float32`
Activation Function	tanh
Input size	Dimension
Output size	1
Optimizer	Adam
learning rate decay	None
learning rate	0.001
dataset size	16896, 2×10^6
max epochs	1000000
batch size	full dataset

B FILE TREE


Project Root
src
visdataset.m
vis3d.m
vis2d.m
saveToBinary.m
pinn
helper.cpp
dataset.cpp
loadFromBinary.m
helper.cpp
boundary.m
main2
pred.cpp
main2.cpp
datagen.cpp
main1
main1_3d.cpp
main1.cpp
include
types.hpp
pinn
types.hpp
pinn.hpp
helper.hpp
dataset.hpp
pde
pde.hpp
details
InitializationsC.hpp
Heat_3D.hpp
Heat_2D.hpp
Heat.hpp
BoundaryConditions
BoundaryConditions_3D.hpp
BoundaryConditions_2D.hpp
multiarray.hpp
multiarray
types.hpp
base.hpp
mpi
types.hpp
topology.hpp
multiarray.hpp
environment.hpp
assert.hpp
helper.hpp
assert.hpp
config.h.in
README.md
Doxyfile
CMakeLists.txt