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

This is the abstract of this project.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

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 of solving partial differential equations (PDE) have demonstrate far better performance than many other methods such as finite difference methods (FDM) [**<empty citation>**], finite element methods (FEM) [**<empty citation>**], Lattice Boltzmann Method (LBM) [**<empty citation>**] and Monte Carlo Method (MC) [**<empty citation>**]. In recent years, researchers in the field of deep learning have mainly focused on how to develop more powerful system architectures and learning methods such as convolution neural networks (CNNs) [**<empty citation>**], Transformers [**<empty citation>**] and Perceivers [**<empty citation>**] . In addition, more researchers have tried to develop more powerful models specifically for numerical simulations. Despite of the relentless progress, modeling and predicting the evolution of nonlinear multiscale systems which has inhomogeneous cascades-scales by using classical analytical or computational tools inevitably encounts severe challanges and comes with prohibitive cost and multiple sources of uncertainty.

This project focuses on the promotions on performance gained from the parallel compute systems, in general, the FDMs and Neural Networks (NNs) are evaluated. Moreover, it prompted series Message Passing and Shared Memory hybrid parallel strategies using Message Passing Interface (MPI) [**MPI**] and Open Multi-processing (OpenMP) [**OpenMP**].

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 [**Monte Carlo Method**] which is a popular method in solving PDEs in higher dimension space with notable complexity.

2.1 Finite Difference Method

The Finite Difference Method(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 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

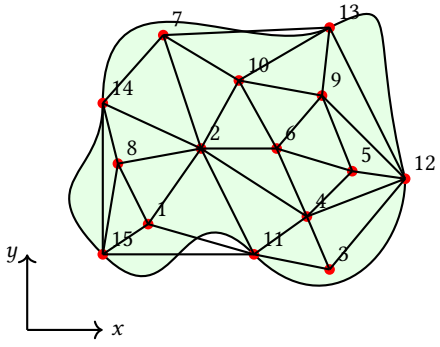


Fig. 1. The Schematic Representa of of a 2D Computiational Domain and Grid. The nodes are used for the FDM by solid circles. Nodes $11 - 15$ denote boundary nodes, while nodes $1 - 10$ denote internal nodes.

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) [**CNN**] for image recoginition, Generative Pre-trained Transformer (GPT) [**GPT**] for natual language processing and Physics Informed Neural Networks (PINNs) [**PINN**] 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) [**AD**] which led to the emergence of a subject called Matrix Calculus [**Matrix_Calculus**]. 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].

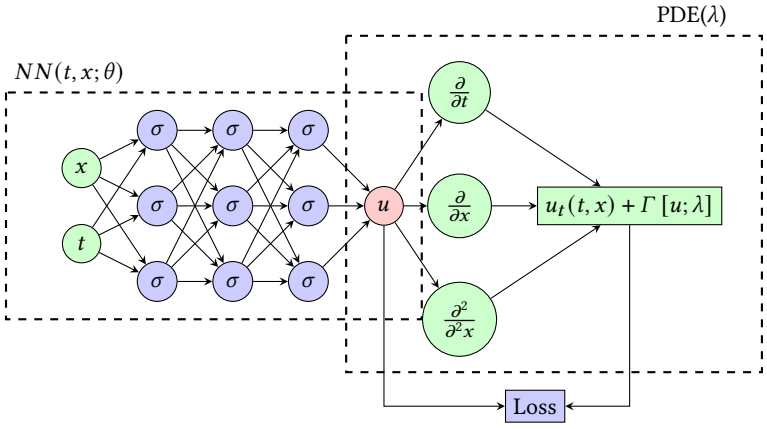


Fig. 2. The Schematic Representation of a structure of PINN, a FCN with 4 layers (3 hidden layers)

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 [**Maxwell_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[**CFL_limitation**]). 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.

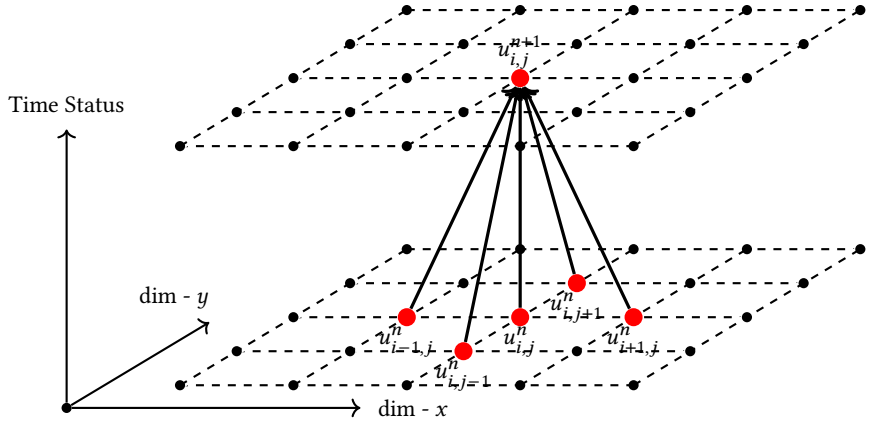


Fig. 3. The Schematic Representation the computational spatio-temporal domain of FDTD methods.

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 [***Amdalh'sLaw***] 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.

261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312

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 [**GaustafssonLaw**] 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.

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 [**Moore_Law**], 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 $\vec{x} \in \mathbb{R}^d$, $d = 2, 3$.

$$\begin{aligned} &\frac{\partial u}{\partial t}(t,\vec{x}) + \mathcal{N}[u(t,\vec{x});\lambda] = 0, && \vec{x} \in \Omega, t \in [0,+\infty) \\ &u(0,\vec{x}) = \varphi(\vec{x}), && \vec{x} \in \Omega \\ &u(t,\vec{x}) = g(t,\vec{x}), && \vec{x} \in \overline{\Omega}, t \in [0,+\infty) \end{aligned} \tag{9}$$

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 9 The boundary condition shown in E.q. 9 is Dirichlet Condition known as first type boundary condition, where as the second type boundary condition (Von Neumman) [E.q. 10] gives the other form of $u(t,\vec{x})$ at the boundary $\overline{\Omega}$.

$$\frac{\partial u}{\partial \vec{x}} = g(t,\vec{x}), \quad \vec{x} \in \overline{\Omega}, \quad t \in [0,+\infty) \tag{10}$$

3.2 Specific Form

With general form proposed in section 3.1[E.q. 9], 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

$$\begin{aligned} &\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial u^2}{\partial^2 x} + \frac{\partial u^2}{\partial^2 y}\right) && (x, y) \in \Omega,\ t \in [0, +\infty) \\ &u(0, x, y) = \varphi(x, y) = 0 && (x, y) \in \Omega \\ &u(t, x, y) = g(x, y) = \begin{cases} y, & x = 0, y \in (0, 1) \\ 1, & x = 1, y \in (0, 1) \\ x, & y = 0, x \in (0, 1) \\ 1, & y = 1, x \in (0, 1) \end{cases} && t \in [0, +\infty) \end{aligned} \tag{11}$$

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

$$u(t, x, y) = x + y - xy, \quad (x, y) \in \Omega,\ t \in [0, +\infty) \tag{12}$$

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

$$u(t, x, y, z) = g(x, y, z) = \begin{cases} y + z - 2yz, & x = 0, \\ 1 - y - z + 2yz, & x = 1, \\ x + z - 2xz, & y = 0, \\ 1 - x - z + 2xz, & y = 1, \\ x + y - 2xy, & z = 0, \\ 1 - x - y + 2xy, & z = 1 \end{cases}, t \in [0, +\infty) \tag{13}$$

In such case, the analytical solution has for form below

$$u(t, x, y, z) = x + y + z - 2xy - 2xz - 2yz + 4xyz, \quad (x, y, z) \in \Omega,\ t \in [0, +\infty) \tag{14}$$

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:

$$\Delta x_i = \frac{1}{n_i - 1} \tag{15}$$

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.

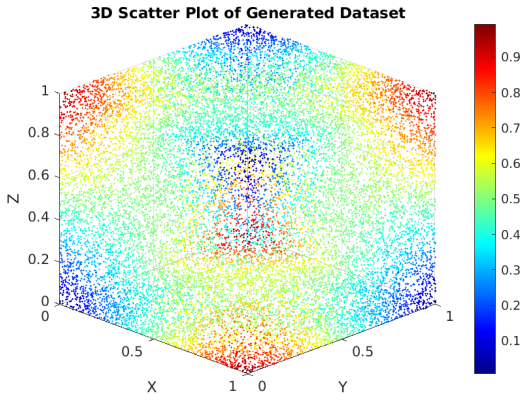


Fig. 4. Randomly general dataset for 3D PINN model training.

3.4 Accuracy

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.

3.4.1 Round-off Error. In IEEE-754 [**IEEE_754**] standards, a 32-bit floating pointer number, single precision, obligatorily represented with 23-bit mantissa, 8-bit exponent and 1-bit for sign. Where as 64-bit floating number, double precision, also ubiquitous used, which has 11-bit exponent and 52-bit mantissa. After almost three decades development, not only single and double precisions (float32, float64) are ubiquitously in use, also more formats such as fp4, fp8, and fp16 etc. Both of them follows the simple form of exponent k, sign n and mantissa N. [**IEEE_754_p2_eq1**]

$$2^{k+1-N}n$$

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.

In numerical methods I investigated, the FDTDs are conventionally using double precision number so that the programs can treat extreamly large and small numbers simultaneously in the same computation, without worring 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 accelarate unit for low precision floating numbers. [**NVIDIA_HB200_PAPER**].

3.4.2 Floating-point Arithmetic. The other type loss comes from the arithmetic operations on two numbers x, y. The standard model holds that

$$fl(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \quad |\delta| < \epsilon \tag{16}$$

where the op stands for the four elementary operations: $+, -, \times, /$. [**Germund**, **NMSC**, **V1**, **P112**].

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`, and other related header files located in the subfolder.

4.2 Template Multi-dimension Matrix Detail Design

The FDTD method is a type of FDM. 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 used them as Dworld and Qworld respectively, also are defined 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, 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 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
```

Performace 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_MSE                      # Number of Arguments must Match the dimension.
index = 0, i = 1                              # Initialize variables in advance.
indices[] = __shape.check_and_cast(ext)       # The indexes number must none-negative number.
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[**BOOST_MPI**] -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. 5] of cartesian topology of 24 processors, with no period in all dimensions.

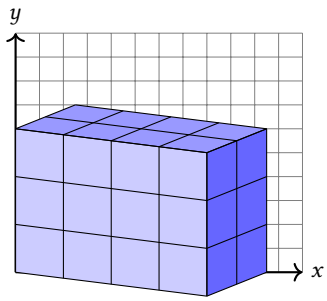


Fig. 5. An example of MPI Cartesian topology Scheme of 24 processors.

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` to create essential halo datatypes.

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,

```
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
```

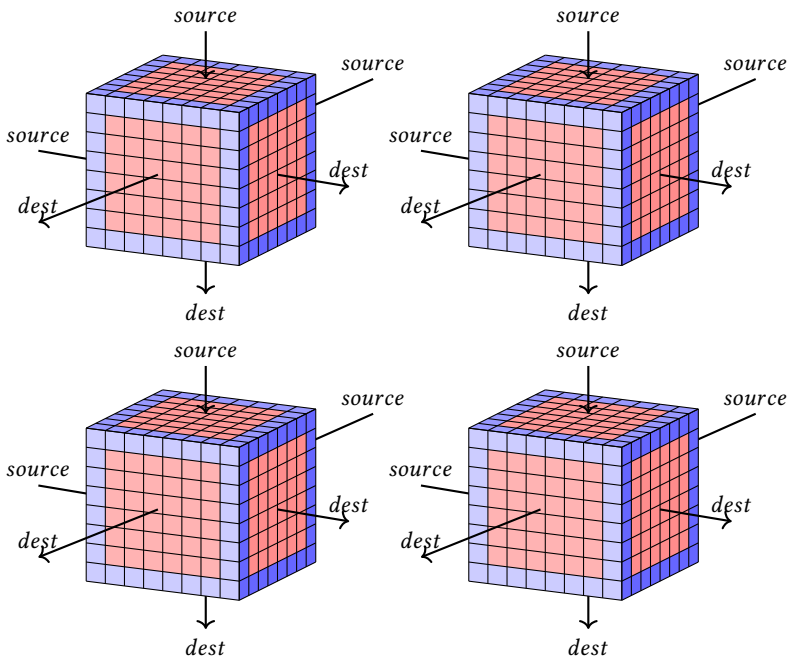


Fig. 6. A 3D MPI Communication Scheme of 8 processors between 3 dimension matrices.

`__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 1. 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 5 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

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.

```
MPI_Type_create_subarray(
    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}
```

Algorithm 6 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
```

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

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. 6] 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 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.

Algorithm 7 `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
```

(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_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 11. 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 `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

$$\Delta t \leqslant \frac{1}{2\lambda \sum_{i=1}^{D} \frac{1}{\Delta x_i^2}} \tag{17}$$

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

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

The weights and diagonal coefficients are

$$\begin{cases} w_i = \dfrac{\lambda \Delta t}{\Delta x_i^2} \\ diag_i = -2 + \dfrac{\Delta x_i^2}{2\lambda \Delta t} \end{cases} \tag{19}$$

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 [**SUPERsmith**].

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 8 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

$$u^{n+1}_{1,\ldots,d,\ldots,D} = u^{n}_{1,\ldots,d,\ldots,D} + \sum_{d=1}^{D} \frac{\lambda \Delta t}{\Delta x_d^2} \left(u^{n}_{1,\ldots,d+1,\ldots,D} - 2u^{n}_{1,\ldots,d,\ldots,D} + u^{n}_{1,\ldots,d-1,\ldots,D} \right) \tag{20}$$

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 6, 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 9 Master-only MPI+OpenMP with non-overlapped Communication/Computation

```
1: for iterations do
2:    #pragma omp single
3:    MPI communications.
4:    for for grid values do
5:       #pragma omp for updates.
6:    end for
7:    #pragma omp single Switch and other operations.
8: end for
```

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 [**Master-Only**], 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

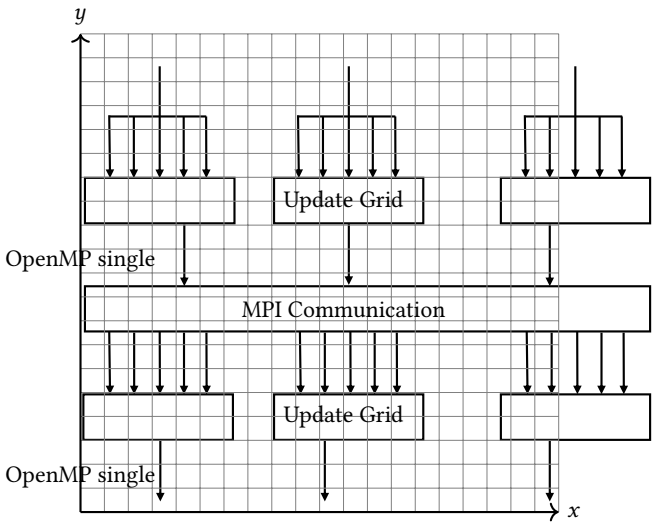


Fig. 7. Scheme: Master only hybrid strategy

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.

Comparing to the previous scheme 9, the scheme 10 overlaps part of MPI communications and computation, rather than only take the advantages of shared memory computing.

Algorithm 10 Funneled Master-only MPI+OpenMPI with overlapped Communication/Computation

```
for iterations do
  #pragma omp single
  MPI communications. (Isends/Irecvs)
  for for bulk points do
    Update bulk points.                    # Update the points that do not need ghosts.
  end for
  MPI_Waitall()                            # Wait all MPI Communications complete
  for for edge pints do
    update edge values.
  end for
  #pragma omp single Switch and other operations.
end for
```

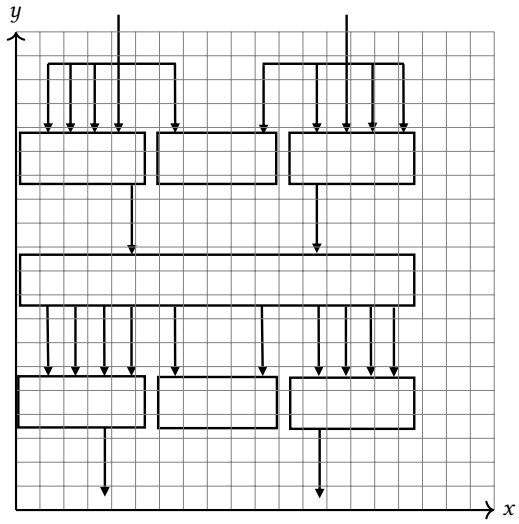


Fig. 8. Scheme: Master only hybrid strategy with computation/communication overlapping

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. 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

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 [**AlexNet**], 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 [**FIRST_PINN**] 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 [**Transformer_PINN**]. 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**].

6.5.1 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.

6.5.2 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 sumption of $loss_{PDE}$ and $loss_{NN}$ where

$$\begin{cases} loss_{PDE} = \dfrac{\partial^2 \widehat{u}}{\partial x^2} + \dfrac{\partial^2 \widehat{u}}{\partial y^2} + \dfrac{\partial^2 \widehat{u}}{\partial z^2} \\ loss_{NN} = MSE(\widehat{u}, u) \end{cases} \tag{21}$$

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::m19937` STL random device with given seed 42 [**STL:RANDOM_SEED**]. 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 is `Callan` [**Callan_TCD**] which has 2 CPUs per compute node, and each CPU has 32 cores with single thread. 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% [**NUMA_Latency_TCD**]. 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.

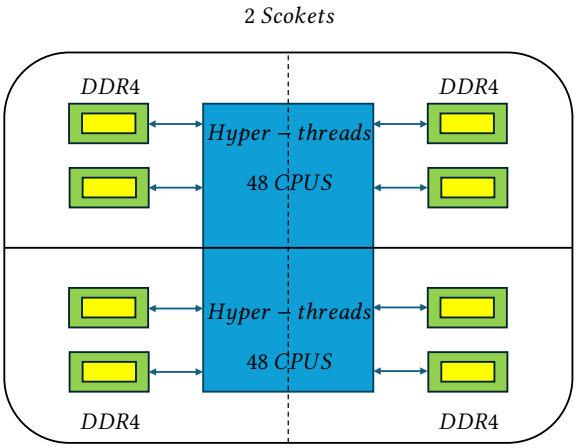


Fig. 9. NUMA topology of single node on Cluster

7.2.1 Strong Scaling. Figure 10 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.

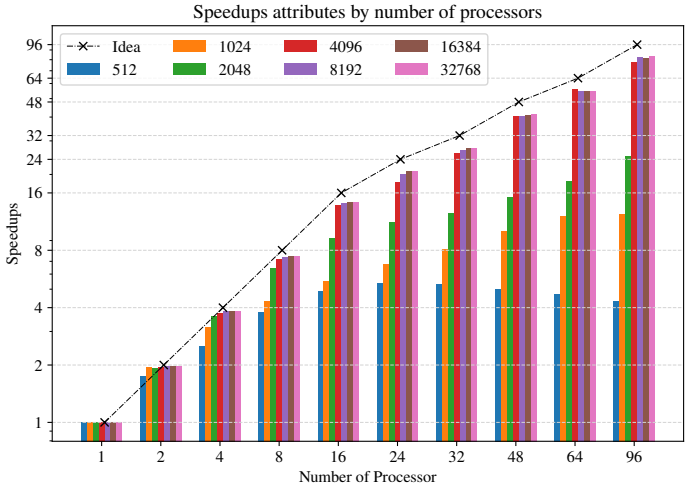
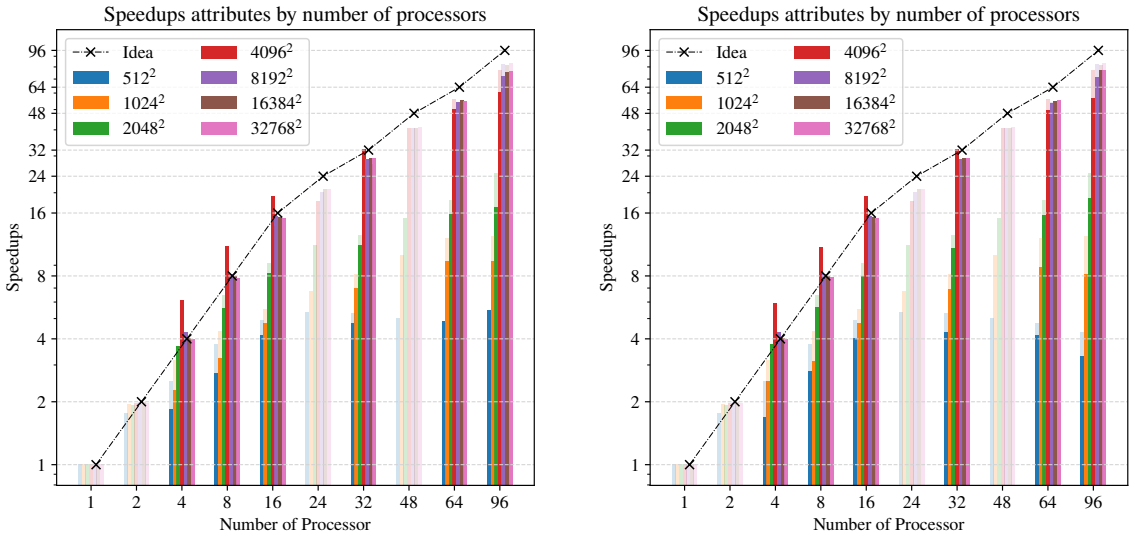


Fig. 10. Comparison of speedup ratios of strong scaling tests of pure MPI parallelized program. The investigated problem scales are the power of 2, exponents ranging from 9 to 15. The number of CPUs are also set as power of 2, with additional numbers 24, 48 and 96 matched the topologies of CPU.

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 10. 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 11 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 11(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. 11. 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 10 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 11(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 `Double` which takes 8 bytes, and 4096^2 `Double` numbers takes 128 `MB` to store. In the case of 4 MPI processes, each process own a quater of number which uses 32 `MB` for handling sub-problems. On the other

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

Strategy	Size	Number of CPUs		
		4	16	64
Pure MPI		4.006	12.497	47.849
No Overlap	512^2	2.876	11.206	42.754
With Overlap		3.173	10.818	42.282
Pure MPI		3.838	9.304	33.707
No Overlap	1024^2	3.947	12.995	33.447
With Overlap		4.024	12.932	33.361
Pure MPI		2.376	8.245	31.203
No Overlap	2048^2	3.874	8.972	31.510
With Overlap		3.740	8.989	31.430
Pure MPI		3.543	8.245	31.203
No Overlap	4096^2	3.953	13.799	49.515
With Overlap		3.948	13.800	49.989

hand, the L3 cache is 36608 KB = 35.75 MB which is just bigger than the sub-problem scale 32 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 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. 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.

Table 1 lists the 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.

7.3 Comparison on multi-node

7.4 Comparison

7.5 Visualization

8 EXPERIMENTS

9 CONCLUSION

10 ACKNOWLEDGEMENT

11 DISCUSSION

12 CONCLUSION

13 FURTHER WORK