UNIVERSITY OF CALIFORNIA,
IRVINE

Algorithm Design for High-Performance CFD Solvers on Structured Grids

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Mechanical and Aerospace Engineering

by

Hengjie Wang

2021

DEDICATION

To my family for their immense support

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

First and foremost, I thank Professor Aparna Chandramowlishwaran for taking me on as her PhD student. I am extremely grateful for her immense support and insightful advice during my PhD program. I switched to her lab in the middle of my PhD program with a background of Computational Fluid Dynamics (CFD) but not much experience in High Performance Computing (HPC). Her encouraging motivations and kind patience really ease the transition. For almost three years working with her, she has shaped my research characters. I could not have accomplished the achievements in this dissertation without her insightful guidance in various HPC topics and her tremendous support in writing, internship and career applying, etc. Thank you for everything.

I thank Michele Rosso and Professor Said Elghobashi for their guidance and support during my research on multi-phase turbulence. It built up my knowledge of CFD and led me to the fantastic world of high-performance algorithm designs.

I thank Professor Feng Liu's support in multiple occasions. I thank Professor Ramin Bostanabad for his advice on Deep Learning, especially when I am a novice in this area. Moreover, I thank you both for taking the time to serve on my dissertation committee. I thank Professor Xiaozhe Hu from Tufts University for his instructions and advice on Geometric Multigrid.

I am grateful to have the outstanding labmates at UC Irvine. I thank Ferran, Laleh, Rohit, Behnam, Shu-Mei, and Octavi for the research discussions. I especially thank Robit for his heroic help during my SC paper's deadline. I cherish all the happy memories with the group, playing painballs and board games, stucking in the escape room, etc.

At last but definitely not least, I am extremely grateful and indebted to my mother, for her endless love and years of sacrifices. I thank Lijing for accompanying me not only during our wonderful journeys but also throughout the stressful deadlines. Without her, my PhD life wouldn't be as pleasant as it is.

VITA

Hengjie Wang

EDUCATION

Doctor of Philosophy in Mechanical and Aerospace Engineering	**2021**
University of California, Irvine	*Irvine, California*
Master of Science in Mechanical Engineering	**2014**
Peking University	*Beijing, China*
Bachelor of Science in Mechanical Engineering	**2011**
Peking University	*Beijing, China*

RESEARCH EXPERIENCE

Graduate Research Assistant	**2020/01–2020/12**
University of California, Irvine	*Irvine, CA*
Research Aide	**2019/07–2019/12**
Argonne National Laboratory	*Lemont, IL*
Graduate Research Assistant	**2014/11–2019/7**
University of California, Irvine	*Irvine, CA*

TEACHING EXPERIENCE

Teaching Assistant	**2020/01–2020/03**
University of California, Irvine	*Irvine, CA*

REFEREED CONFERENCE PUBLICATIONS

Pencil: A Pipelined Algorithm for Distributed Stencils **Nov 2020**
The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC20)

Multi-Criteria Partitioning of Multi-Block Structured Grids **Jun 2019**
2019 International Conference on Supercomputing (ICS19)

SOFTWARE

Pencil `https://gitlab.com/UCI--CFD/pencil`
A pipelined algorithm for distributed stencil computation.

Multi-Block Structured Grid Partitioner `https://gitlab.com/UCI--CFD/SMP`
Partitioning algorithms for multi-block structure grids.

ABSTRACT OF THE DISSERTATION

Algorithm Design for High-Performance CFD Solvers on Structured Grids

By

Hengjie Wang

Doctor of Philosophy in Mechanical and Aerospace Engineering

University of California, Irvine, 2021

Associate Professor Aparna Chandramowlishwaran, Chair

Physics-based simulation, Computational Fluid Dynamics (CFD) in particular, has substantially reshaped the engineering design process by dramatically reducing the experimental cost. The effectiveness of CFD is primarily driven by the power of High Performance Computing (HPC) systems. However, the CFD community's top-priority task has been to develop high-order accurate schemes for complex physical phenomena like turbulence, instead of designing algorithms towards better usage of the hardware. Moreover, the hardware is evolving at a much faster pace than the numerical methods. As a result, we currently lack efficient algorithms for the practical CFD schemes to fully leverage the HPC systems. In this dissertation, we propose high-performance algorithm designs to improve the performance and productivity of CFD solvers using multi-block structured grids on modern supercomputers.

First, to parallelize simulations, we need to partition and distribute the grid across processors. Current grid partitioners only factor communication metrics and ignore the underlying network and architectures. To achieve portable performance, we introduce a novel cost function combining the algorithmic metrics with the network properties. Based on the cost function, we propose new partitioning algorithms that minimize the communication cost while balancing the computation workload.

Second, the computation in CFD solvers is typically memory-bound and optimized with cache tiling. The state-of-the-art tiling techniques are mainly designed for single-block grids on shared memory systems, whereas multi-block grids distributed across many nodes are the norm for realistic CFD applications. In this thesis, we demonstrate how to tile distributed computation for multi-block grids. Furthermore, we propose novel pipelined algorithms to hide the communication cost which improves both the performance and strong scaling of CFD solvers.

At last but not least, constructing a Deep Learning(DL) surrogate for CFD can further improve its productivity in engineering design. The surrogates must be able to predict for solutions across different boundary conditions and geometries unseen during training. In this dissertation, we design transferable DL surrogates for Boundary Value Problems (BVP) resembling the CFD simulations for steady-state flow. Furthermore, we propose an iterative inference method that transfers the prediction across arbitrary domain sizes and shapes.

Chapter 1

Introduction

The rapid evolution of *Computational Fluid Dynamics* (CFD) during the last several decades has fundamentally reshaped the design process in many engineering disciplines. An engineering design process involving many configurations can take up to $O(10^6)$ test cases. For instance, in aircraft design, just the evaluation of one aircraft configuration involves simulations under different combinations of the Reynolds (Re) and Mach (Ma) numbers, various angles of pitch, yaw, and roll, as well as numerous positions of flaps, stabilizers, etc, adding up to thousands of test cases. With CFD, most of the test cases can be simulated numerically on supercomputers, drastically reducing the experimental costs.

The first step to simulate a fluid phenomenon with CFD is to discretize the computation domain with a grid. Physical variables such as velocities and pressure are stored either in grid cells or on grid vertices. Then, a CFD solver solves the governing equations for each grid cell/vertex. There are two main types of grids, namely, structured and unstructured grids. Structured grids are characterized by the regular connectivity between grid cells, where each grid cell can locate its neighbor simply by shifting the cell's indices, for instance, grid cell (i, j)'s left neighbor is cell $(i - 1, j)$. Structured grids are organized into blocks (rectangular shapes in the i, j, k index space). Typically, grids for complex geometries such as an aircraft or turbo-machinery contain on the order of 100's of blocks [1]. The block structure and regular connectivity render an identical mapping between the grid's data to the memory layout in DRAM, which makes the structured grid an ideal testbed for optimization techniques such as cache tiling and vectorization. Unstructured grids, opposite to structured grids, sacrifice the regular connectivity for the ability to form arbitrarily connected shapes to resolve complex geometries, resulting in a more complex data layout and irregular memory accesses. Both grids are widely supported by the major CFD solvers used in engineering and have demonstrated similar accuracy in practice [2]. In this thesis, we focus on improving the performance of CFD solvers on structured grids.

The size of grids for engineering applications in 3D such as vehicle and aircraft typically exceeds the memory of a single processor. Therefore, the grid needs to be partitioned and distributed to multiple processors. The connectivity between different partitions results in data dependence. To parallelize the simulation, we need to exchange the dependent data (called *halos*) between processors in addition to computation. As a result, the distributed CFD simulation follows the workflow in Figure 1.1. The workflow starts with partitioning the grid and proceeds to an iterative process of solver computation and halo communication. The time cost of grid partitioning is typically negligible compared to the total cost of computation and communication. The iterative process stops when the numerical solution has either converged to a steady-state solution or reached a prescribed time.

When exploiting the above CFD workflow in an engineering design process, we estimate the overall simulation cost as $\bar{T}_{\text{solver}} \cdot N_{\text{sim}}$. $\bar{T}_{\text{solver}}$ denotes the average solve time of one test case using CFD and N_{sim} represents the number of cases numerically simulated. For instance, in aircraft design, the majority of the test cases are simulated with CFD. Therefore, $N_{\text{sim}} \approx N_{\text{test}}$, where N_{test} represents the total number of test cases.

The effectiveness of CFD in engineering design is mostly determined by how we can mini-

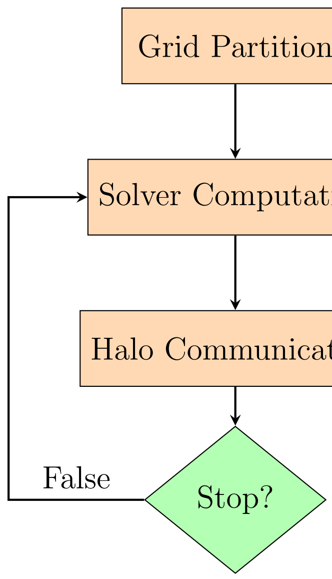


Figure 1.1: The general workflow of CFD solvers. The stop criterion for steady problems checks whether the solution has converged. For unsteady problems, it checks if the simulation time has reached the desired value.

mize $\bar{T}_{\text{solver}} \cdot N_{\text{sim}}$ with the power of modern *High Performance Computing* (HPC) systems. Nonetheless, the CFD community's top-priority task has been to develop high-order accurate schemes for complex physical phenomena like turbulence rather than designing algorithms that efficiently map to the underlying hardware. On the other hand, HPC architectures have been evolving at a rapid pace predicted by the Moore's Law, which has outpaced the development of CFD methods. As a result, revolutionary algorithmic developments are necessary to improve both the performance and scalability of current CFD solvers to fully leverage the power of modern HPC hardware [3]. More specifically, we need to

1. minimize $\bar{T}_{\text{solver}}$ by systematically optimizing the workflow of CFD solvers and,

2. reduce N_{sim} by designing fast surrogates for CFD solvers.

In the remainder of this chapter, we will first briefly describe the limitations of the state-of-the-art algorithms that prevent us from achieving the above goals, then list our contributions to overcome those limitations. We will provide more detailed context for understanding these limitations in Chapter 2.

The initial grid partitioning step plays an essential role in reducing T_{solver} despite its negligible time cost. It determines the workload of both solver computation and halo communication. The computation workload is directly reflected by the number of grid cells on a processor. The communication workload is typically measured by the data volume (referred to as *communication volume*) and the number of messages (referred to as *edge cuts*) transferred through the network. Both metrics depend on how the grid is partitioned across processors. As a result, the grid partitioner must balance the computation workload and minimize the communication cost across processors. Note that, in addition to the above metrics, the communication performance also relies on the network properties such as latency and bandwidth. Depending on those properties, some networks may be more efficient in transferring a few large messages while others may favor many small messages [4–6]. However, the state-of-the-art grid partitioners [7–14] only factor communication metrics and ignore the network properties, rendering the communication performance not portable across different networks. Moreover, current partitioners are primarily designed with a flat view of the architecture, i.e., mapping each partition to a core and averaging the communication cost among cores. However, modern multi-core architectures follow a core-socket-node hierarchy. The communication between cores on the same nodes (*intra-node*) is much faster than the communication across nodes (*inter-node*). The grid partitioner should only focus on minimizing the inter-node communication. As a result, we are currently in the need of novel partitioning algorithms that factor network properties and conform to modern HPC architectures.

After achieving an optimal partition of the grid, we optimize the iterative process in Figure 1.1 in two aspects. First, we want to optimize the computation on structure grids, which is typically memory-bound, i.e., limited by the time cost of moving data between CPU and DRAM. Second, we want to hide the communication cost by overlapping it with computation. There have been significant progresses in the algorithm design for both aspects from the HPC community [15–18]. Nonetheless, it is still quite challenging to directly apply these

```
// time loop
for(int t=0; t<tEnd; ++t)
  // space loop
  for(int i=0; i<iEnd; ++i)
    for(int j=0; j<jEnd; ++j)
      for(int k=0; k<kEnd; ++k)
        compute();
```

Figure 1.2: Pseudocode for single-block grids characterized by a perfectly nested loop.

```
// time loop
for(int t=0; t<tEnd; ++t){
  // nBlock - number of blocks
  for (int b=0; b<nBlock; ++b){
    // space loop for block b
    for(int i=0; i<iEnd[b]; ++i)
      for(int j=0; j<jEnd[b]; ++j)
        for(int k=0; k<kEnd[b]; ++k)
          compute();
    // resolve blocks' dependency
    exchange_block_boudary();
  }
}
```

Figure 1.3: Pseudocode for multi-block grids.

algorithmic advances to practical CFD solvers. This is because most studies are based on *cache tiling* techniques for *perfectly nested loops*, which only corresponds to a single-block structured grid, whereas we are mainly confronted with multi-block grids in engineering practice. As seen from Figure 1.2, a perfectly nested loop only does computation in the inner-most loop. To solve for multi-block grids, in addition to computation, we need to resolve the data dependency between outer loops as shown in Figure 1.3, which disrupts the perfectly nested loop structure and prohibits a direct application of the existing optimization techniques. This will be further discussed in Chapter 4. Furthermore, the optimizations for the computation step are mostly designed for shared memory machines. How to efficiently optimize the distributed computation on structured grids is yet to be explored. As a result, we currently need novel algorithms to optimize the distributed computation and overlap communication specifically for multi-block grids.

After systematically optimizing the CFD workflow in Figure 1.1, we want to reduce N_{sim} by building a surrogate for CFD solvers with *Deep Learning*, or say deep *neural networks*, which requires significantly less computation. The key idea is that a neural network with sufficient parameters can approximate any complex mapping from the flow inputs such as boundary/initial conditions and data snapshots to the flow solution. The parameters are determined via a tuning process (called *training*) using sample solutions generated by a CFD solver. Once properly trained, the neural network should be able to predict flow solutions using inputs unseen during training (called *inference*) with a limited loss in accuracy compared to CFD solvers. For example, the state-of-the-art Physics Informed Neural Network (PINN) [19] can infer the solution of a lid-driven cavity with less than 10% error compared to a CFD solver [20]. The reduction of accuracy is because the training can at best penalize the mean squared residual of governing equations by tuning parameters, which is prone to reaching plateaus. On the other hand, CFD solvers can explicitly enforce the governing equations at each grid cell/point to machine precision. Despite the loss in accuracy, the time for inference (T_{infer}) is typically negligible compared to the simulation time using a CFD solver ($\bar{T}_{\text{solver}}$), i.e., $\bar{T}_{\text{solver}} \gg T_{\text{infer}}$. Ideally, the simulation cost of a design process such as aircraft design can be reduced as follow,

$$\begin{gathered} \bar{T}_{\text{solver}} \cdot N_{\text{test}} \rightarrow \bar{T}_{\text{solver}} \cdot N_{\text{sim}} + T_{\text{infer}} \cdot N_{\text{infer}}, \\ N_{\text{test}} = N_{\text{sim}} + N_{\text{infer}} \end{gathered} \tag{1.1}$$

where N_{infer} denotes the number of test cases inferred by the surrogate. The number of CFD simulations is approximately reduced from $N_{\text{sim}} \approx N_{\text{test}}$, which covers all the test cases, to $N_{\text{test}} - N_{\text{infer}}$, which only involves some challenging test cases that the less accurate surrogate can not infer. Such cases may involve complex transient phenomena such as landing and taking off, which are still open topics for the state-of-the-art CFD solvers.

To benefit from surrogate modeling, we need to maximize the number of test cases that a surrogate can infer. This requires the model to be transferable, i.e., trained only on simple geometries, boundary, and initial conditions yet applicable to complex unseen geometries and conditions. In aircraft design, this ensures the model can be reused across cases with different pitch/yaw/roll angles (set by boundary conditions) and various configurations (corresponding to different geometries). Moreover, to surrogate a CFD solver is a very demanding task as both the data collection and training are very time-consuming. These costs can only be amortized when the model is applicable to a large number of test cases. Nonetheless, current studies are mostly designed to make predictions for a particular domain under specific boundary and initial conditions. For example, a PINN surrogate that infers the flow over an airfoil will become largely useless if the shape of the airfoil or the angle of attack varies. Some recent work [21–24] can generalize the model to infer a moderate range of unseen boundary conditions but they only apply to the specific geometries or grids used in training. As a result, there is a pressing need for transferable surrogate models that can infer flows across arbitrary geometries and conditions.

To address the above algorithmic needs in minimizing the simulation cost $\bar{T}_{\text{solver}} \cdot N_{\text{sim}}$, this thesis makes the following contributions:

- **Multi-Criteria Structured Grid Partitioner** (Chapter 3)

  We present the very first cost function that unifies the communication metrics and network properties for partitioning multi-block structured grids. Based on the multi-criteria cost function, we design novel partitioning algorithms for both cutting large blocks and grouping small blocks. Evaluated with a benchmark solver, our algorithms achieve up to 2x speedup in communication compared to the start-of-the-art partitioners on a structured grid based on SpaceX's Falcon-Heavy rocket.

- **Pipelined Distributed Stencil Computation** (Chapter 4)

  We demonstrate how to optimally extend the current optimizations for single-block grids and apply them to distributed computation on multi-block grids. Moreover, we propose a novel fine-grained algorithm to pipeline the computation and communication to achieve overlap. Combining the optimizations for computation and communication, we can improve the performance of the iterative process in Figure 1.1 by up to 3.4x.

- **Generalized Neural Network Surrogate for Solving BVP** (Chapter 5)

  We propose a novel deep learning model to solve the *Boundary Value Problem* (BVP) for elliptic PDEs. The model can predict solutions for boundary conditions unseen during training. Furthermore, we design an iterative inference method that can transfer the model's prediction to domains of unseen sizes and shapes. With an overlapped domain decomposition, the iterative inference can be performed in parallel and scaled on distributed machines. Evaluated by solving the Laplace equation, our network trained only on simple geometries can infer solutions for complex composite shapes with unseen boundary conditions and only introduces a limited loss of accuracy.

Chapter 2

Basics of CFD on Structured Grids

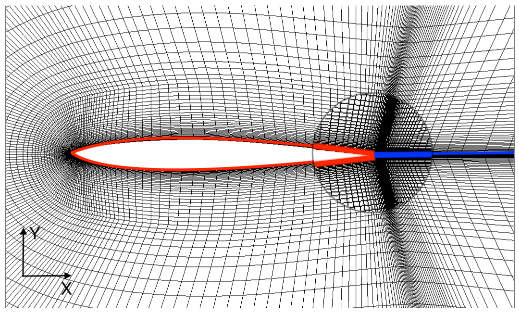
In this chapter, we introduce the basis concepts in developing high performance CFD solvers. Section 2.1 provides more context and examples for multi-block structured grids. Section 2.2 describes the dominant patterns of computation and communication for structured grids. Section 2.3 introduces the most widely used parallel programming models to implement the computation and communication.

2.1 Multi-Block Structured Grids

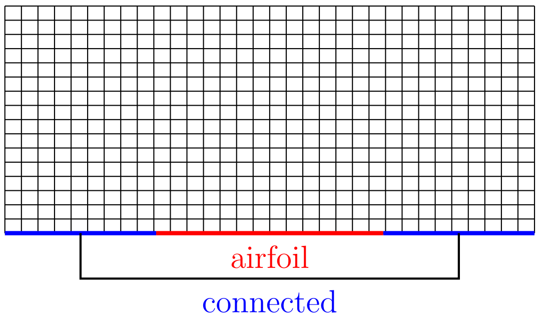
Structured girds are composed of blocks of rectangular shapes in the index space, i.e., i, j space in 2D or i, j, k space in 3D. Figure 2.1a presents a single-block structured grid surrounding an airfoil. The grid lines are curved in the x, y coordinates space. As shown in Figure 2.1b, the grid can be transformed into a rectangular shape by cutting it from the airfoil's trailing edge to the domain's right boundary and unfolding the airfoil's surface to a flat edge. Note that the blue boundary in Figure 2.1b are connected in the original grid, which results in data dependence within the block. We denote such boundary conditions as

Block2Block.

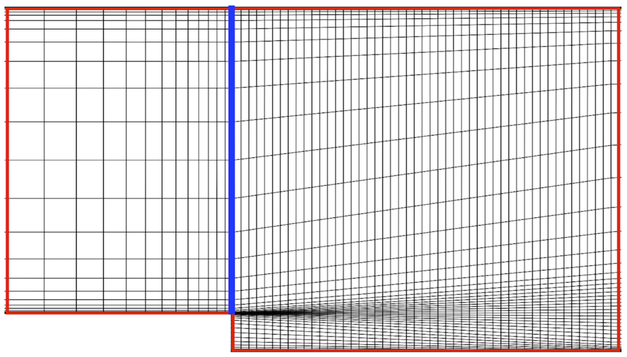
Figure 2.1c illustrates a structured grid for a backward facing step. Because of the step, this geometry cannot be properly resolved by a single block. Therefore, two blocks are used for parts before and after the step respectively. The boundary connecting the two blocks (marked blue) is Block2Block but unlike the airfoil, it introduces data dependence between blocks. In general, the number of blocks increases as the geometry becomes more complex. For geometries in aircraft design or turbo-machinery, the structured grid may need on the order of $10^2 \sim 10^3$ blocks connected by Block2Block boundaries.



(a) A single block surrounding an airfoil.



(b) The single block in index space.



(c) A 2-block structured gird for backward facing step.

Figure 2.1: Examples of structured grids. The red color marks physical boundary conditions and the blue color notes Block2Block boundaries.

2.2 Stencil Computation and Halo Exchange

Stencil computation is the dominant computational pattern on structured grids, which is characterized by the regular shape of the grid cell to update and its data-dependent neighboring cells. Figure 2.2a, 2.2b, and 2.2c show three different stencil shapes in a 2D structured grid, where the update of the blue cell depends on its pink neighbors. The size of a stencil is measured by the stencil radius, which is defined as the largest distance between the cell to update and its dependent neighbors. The three stencils in Figure 2.2 all have a unit radius.

The regular dependency between stencil cells indicates that both the floating-point operations (FLOPs) and DRAM traffic (data transferred between CPU and DRAM) are proportional to the grid size. Therefore, the stencil computation is typically memory-bound on modern architectures with high machine balance.

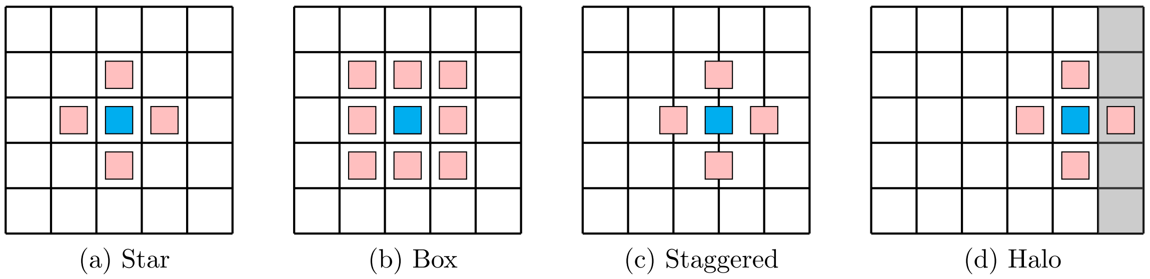


Figure 2.2: Different stencil shapes in 2D with a radius of 1. The color represents the value assigned to the cell. In Star and Box stencils, the value is at the cell's center. For Staggered stencils, the value is located on the cell's face. The update of a boundary cell needs cells in the halo (marked gray).

To update a block's boundary cell, the stencil needs cells outside the block's boundary. Hence, in Figure 2.2d, each block includes additional halo layers (marked gray) as thick as the stencil radius. For physical boundary conditions such as wall, far-field, etc, the halo values are set up numerically. For Block2Block boundaries, the halo values are exchanged between the connected blocks, which is referred to as *halo exchange*.

2.3 Programming Model

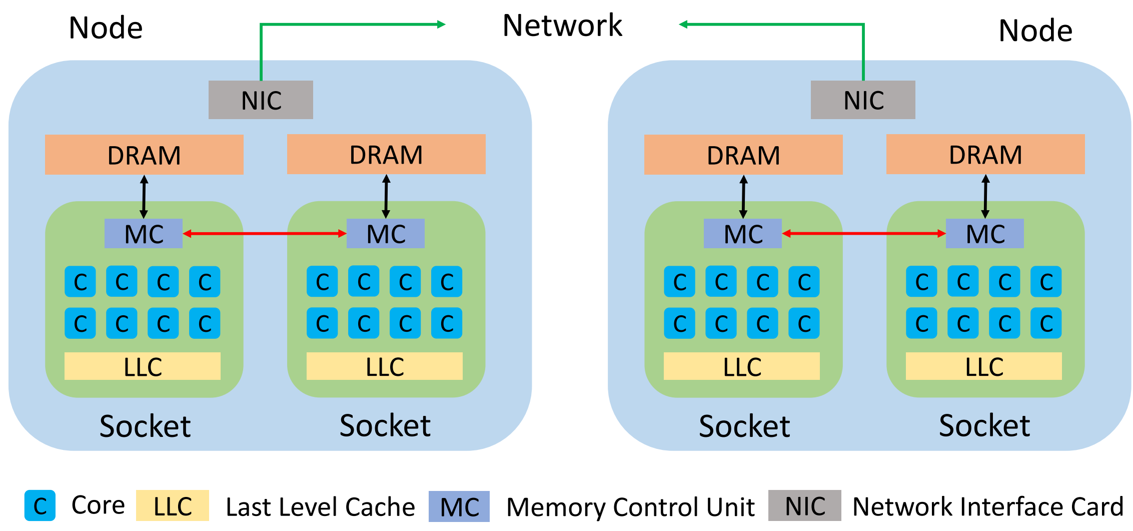


Figure 2.3: Multi-Core Architecture in Modern Supercomputers

Figure 2.3 illustrates the architecture of processor nodes in the modern supercomputers. The processor follows a core-socket-node hierarchy. Each core has its own CPUs and private cache (omitted in the figure). Cores on the same socket share access to the *Last Level Cache* (LLC) and DRAM via the memory control unit. On modern architectures, there are typically three levels of cache. So the last level cache is also frequently denoted as L3 cache. Sockets on the same node can access each other's memory via inter-socket connections as shown by the red arrow in Figure 2.3. However, accesses between a core to another socket's memory can be significantly more expensive than accessing its socket's own DRAM, which is known as the *Non-Uniform Memory Access* (NUMA) effect. In this thesis, we refer to both memory accesses as shared memory access despite the non-uniform cost. Furthermore, each node has at least one *Network Interface Card* (NIC) that connects the node to the network and facilitates data transfer across nodes. Such transfer is typically much more expensive that shared memory access.

The cores execute the computation in parallel and exchange data (such as halo exchange) to resolve data dependencies via communication. We denote the communication within the same node and across different nodes as *intra-node* and *inter-node* communication respectively. The intra-node communication is performed via shared memory access, while the inter-node communication is conducted through the network. Message Passing Interface (*MPI*) has been the de facto standard for communication on distributed-memory machines. MPI can handle both intra-node and inter-node communication internally under the same elegant functional interfaces. This results in a flat programming model, *MPI-everywhere*, where the computation is partitioned evenly among cores and each MPI process is mapped to a single core to execute that core's computation. The advantage of MPI-everywhere is obvious that it allows the domain scientists to ignore the architecture's hierarchy and eases the development of domain softwares. The disadvantage is that using communication protocols for intra-node communication can result in additional memory copies and overhead than directly transferring data via shared memory access [25]. Moreover, partitioning multi-block grids among all the cores tends to create a large amount of halo [26], potentially resulting in running out of memory for large grids [27].

To better leverage the on-node resources, domain scientist have introduced a new programming model, *MPI+threads*, where each MPI process is mapped to a node/socket and launches threads to execute computation on cores. This resolves MPI-everywhere's disadvantages in two ways. First, the threads directly transfer the on-node data via shared memory, avoiding unnecessary memory copies and overhead. Furthermore, it reduces the number of partitions (because of less MPI processes), which in turn reduces the amount of halo and memory usage.

MPI+threads supports four modes for inter-node communication, among which the *Funneled* and *Mutiple* modes are of great research interest. In the funneled mode, only one thread executes MPI routines for inter-node communication, overlapping with other threads' tasks. The multiple mode allows all the threads to communicate simultaneously, which can lead to better communication performance. However, the start-of-the-art MPI implementations still use conservative approaches, such as a global critical section, to maintain thread safety and MPI's ordering constraints, severely deteriorating the multiple mode's parallel efficiency. Only until recently, the pioneering work [28] enables scalable communication in the multiple mode but an efficient way to apply it to domain applications is yet to be explored. Therefore, we will focus on MPI+threads funneled in this thesis. Among the available thread packages, MPI+OpenMP is the most popular combination for OpenMP's simplicity and wide support from compilers. For the remainder of this thesis, we will use the term MPI+OpenMP for MPI+threads and assume using the funneled mode by default.

Algorithm 2.1 Distributed Stencil Computation with MPI-everywhere

```
for $t = 1 \rightarrow$ nIter do
    COMPUTE
    PACK_HALO_TO_BUFFER
    EXCHANGE_HALO
    UNPACK_BUFFER_TO_HALO
```

Algorithm 2.1 shows how the iterative process in Figure 1.1 is implemented with MPI-everywhere. The function pack_halo_to_buffer packs the data to be sent into a 1D buffer. The function unpack_buffer_to_halo unpacks the buffer's data to the halo region. The function exchange_halo consists of MPI routines that handle both intra-node and inter-node communication.

Algorithm 2.2 Distributed Stencil Computation with MPI+threads

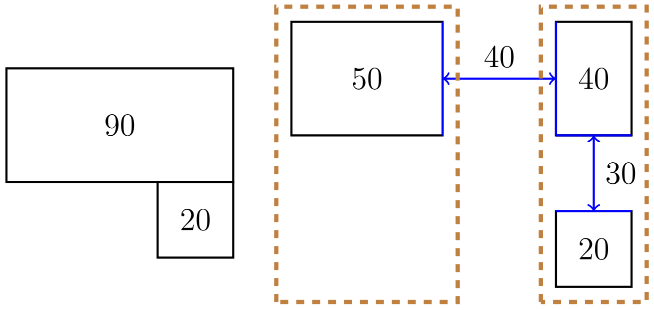
```
for $t = 1 \rightarrow$ nIter do
    COMPUTE
  ▷ #pragma omp barrier
    PACK_HALO_TO_BUFFER
  ▷ #pragma omp barrier
  ▷ #pragma omp master
    EXCHANGE_HALO
    COPY_HALO_SHARED_MEM
  ▷ #pragma omp barrier
    UNPACK_BUFFER_TO_HALO
  ▷ #pragma omp barrier
```

Algorithm 2.2 shows how to implement the the iterative process with MPI+OpenMP. All the functions are executed in parallel with threads except for exchange_halo which is only called by the master thread. The functions pack_halo_to_buffer and unpack_buffer_to_halo are essentially the same as MPI-everywhere but only handle the halos in inter-node communication. The master thread performs inter-node communication in function exchange_halo. The function copy_halo_shared_mem transfers halos between sub-blocks within the same partition using shared memory access. This step is overlapped with inter-node communication. A common mistake using threads is to allow multiple threads to access the same memory location simultaneously (called *Data Race*), which leads to indeterministic results. Data race is likely to occur between adjacent steps. For instance, in exchange_halo the master thread may access some location that another thread has not yet finished packing. To avoid data race, we add thread barriers between each step in Algorithm 2.2. The barrier ensures that all threads launched by the same process are synchronized before starting the next step.

Chapter 3

Multi-Criteria Structured Grid Partitioner

Following the CFD workflow in Chapter 1, the first step to run a CFD solver in parallel is to partition the multi-block structured grids into sub-blocks and distribute them across processors. A "good" partitioner should not only balance the computation workload among the processors but also minimize the communication cost in the network. The workload can be measured by the number of grid cells per processor. The communication cost is typically estimated using algorithmic metrics. There are two primary metrics – the communication volume and edge cuts. The former denotes the volume of data transferred through the network and the latter refers to the number of messages communicated between processors. Figure 3.1 exemplifies the metrics by partitioning two blocks into two partitions and assigning one partition per processor. The two blocks have workloads of 90 and 20 respectively. The large block in Figure 3.1a is cut into two sub-blocks. The larger sub-block forms one partition (marked by brown). The smaller sub-block is grouped with the small block to construct the second partition. The deviation from the average workload ($(90+20)/2 = 55$) is 5, resulting in an imbalance of $5/55$ in Table 3.1c. The blue color in Figure 3.1b marks the halo exchanges



(a) 2 Blocks (b) 2 Partitions

Metric	value
Load Imbalance	5/55
Communication Volume	80 bytes
Edge Cuts	2
Shared Memory Copy	60 bytes

(c) Metrics

Figure 3.1: An example of partitioning 2 blocks into 2 partitions. Each solid box represents a block/sub-block with the workload marked at its center. The brown dashed box denotes a partition. The blue color indicates Block2Block boundaries and their halo exchanges with message size noted in bytes. Each arrow represents two messages so the values are doubled in the right table.

and the value above the arrow denotes the message size. Each halo exchange accounts for two messages (one send and one receive). There is one halo exchange across processors, adding up to two edge cuts and 80 bytes in communication volume. The halo exchange within the partition is implemented by shared memory copy, leading to 60 bytes in DRAM traffic.

In addition to the metrics, the network properties such as bandwidth and latency also have a substantial effect on the communication performance. Therefore, the estimation of communication cost is both application-specific and architecture-specific. As a result, there are numerous factors and trade-offs to consider when devising an optimal grid partitioner.

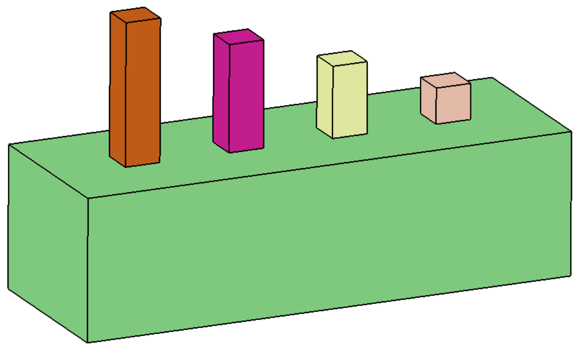
The state-of-the-art work on partitioning structured grids can be categorized into two classes – *top-down* and *bottom-up* strategies. The *top-down* strategy either cuts off chunks of large grid blocks or groups small blocks to fill the available capacity of partitions. Greedy heuristics are typically suitable for this type of approach. A classical algorithm was proposed in [7] and later studies [8–12, 29] can all be viewed as an improvement of a greedy framework. The *bottom-up* strategy treats the grid as a graph and partitions it with a graph partitioner. However, a direct application of graph partitioners such as METIS [30], CHACO [31] to structured grids is not feasible since it fails to preserve the regular connectivity. The remedy is to first cut the blocks into smaller sub-blocks and then partition the grid as a graph of sub-blocks [13]. Finally, the sub-blocks within the same partition are merged. A large number of equal-size sub-blocks are desirable for graph partitioners but this is likely to result in excessive blocks and halo regions which can, in turn, harm the communication performance [14].

Prior works [7–14] mostly share the following drawbacks. First of all, most partitioners focus on minimizing algorithmic metrics without regard for network properties like bandwidth and latency. However, as shown by several studies, the same benchmark can have very different communication performance depending on the network properties [4–6]. Therefore, the blindness towards the network will render the performance non-portable. Second, current partitioners are primarily aimed at reducing the communication volume by techniques such as cutting at the longest edge [7–9]. Edge cuts are only implicitly factored by avoiding splitting Block2Block boundaries [12]. Even for those well-developed graph partitioners [30, 31], the users still need to choose between minimizing communication volume or edge cuts. A cost function that combines both algorithmic metrics is yet to be explored. Finally, current partitioners have only been studied with an MPI-everywhere model where each MPI process is mapped to one core. Their communication mixes the fast shared memory access with slow inter-node communication. Such performance can be misleadingly optimistic and will misguide the design of partitioning algorithms. We argue that the grid partitioner should be studied in the context of MPI+threads and focus on minimizing the inter-node communication.

To overcome the above drawbacks, we propose a novel cost function in this chapter unifying the algorithmic metrics and network properties, based on which we design new partitioning algorithms following the top-down strategy. The remainder of this chapter is organized as follows. Section 3.1 introduces two baseline heuristics following the top-down and bottom-up strategies respectively for performance comparison. Section 3.2 presents our novel cost function unifying the algorithmic metrics and network properties and details our new partitioning algorithms. Section 3.3 compares our algorithms over the baseline heuristics by partitioning two test structured grids. The resulting partitions are evaluated with a MPI+OpenMP based benchmark solver. Section 3.4 summarizes the related works. Finally, Section 3.5 concludes our contributions and points out further directions for partitioning structured grids.

3.1 Classic Partitioning Algorithms

In this section, we describe two baseline partitioning heuristics following the top-down and bottom-up strategies respectively. Among the *top-down* algorithms, the classical greedy heuristic [7] is most widely used in CFD software such as elsA [32]. Although researchers [10, 12] claim improvements over the greedy heuristic, their performance enhancement is observed in the context of a flat MPI model. As a result, the partitions' effect on inter-node communication is not entirely clear. In addition, their implementation details are not available to re-produce their partitioners. Therefore we choose the classical greedy heuristic as the baseline for *top-down* algorithms. As for the *bottom-up* algorithm, the creation of sufficient small sub-blocks is critical while the choice of the actual graph partitioner is not as important. Therefore, we choose METIS for its popularity and widespread use.


(a) Geometry

Block ID	Size
0	$224 \times 64 \times 80$
1	$16 \times 16 \times 16$
2	$16 \times 32 \times 16$
3	$16 \times 48 \times 16$
4	$16 \times 64 \times 16$

(b) Block Sizes

Figure 3.2: Illustration of the geometry of the Bump3D grid with 5 blocks and its corresponding block sizes.

We explain different partitioning strategies using a synthetic grid called *Bump3D* which consists of 5 blocks as shown in Figure 3.2. Bump3D has one block that is significantly larger than the others which challenges the algorithms' ability to cut large blocks. Bump3D resembles the flow through a pipe with outlets on its upper surfaces.

Both the baseline heuristics and our new algorithms in section 3.2 will be evaluated in the context of MPI+threads. From now on, we assume each partition is by default assigned to a single processor node. When computing the communication metrics, we view the Block2Block connection across partitions as inter-node communication and the halo exchange within one partition as shared memory copy.

3.1.1 Greedy Method

The greedy algorithm [7] follows the top-down strategy. At any step, it chooses the unassigned block with the largest workload (i.e., the maximum number of grid cells) and the partition with the most remaining room. If the block exceeds the available capacity of the partition, a sub-block is cut off to fill the remaining capacity and the remainder is added to the list of unassigned blocks. Otherwise, the entire block is assigned to that partition. This process is iterated until all blocks/sub-blocks have been assigned. The algorithm always cuts across the longest edge so that the Block2Block boundary has the minimum surface compared to cutting at other edges.

In the greedy algorithm, the cut position is round up to an integer, which can result in load imbalance. For instance, if the largest block has a size of $8 \times 8 \times 32$ and the average workload $\overline{W} = 544$, splitting along the longest edge at z = 8 or 9 would introduce 6.75% load imbalance. Even worse, the algorithm may fail if the smallest surface of a block has more cells than $\overline{W}$. To address this limitation, we propose the following remedy. Given an imbalance tolerance ϵ, a block of size $N_x \times N_y \times N_z$, $N_x < N_y < N_z$, and remaining capacity

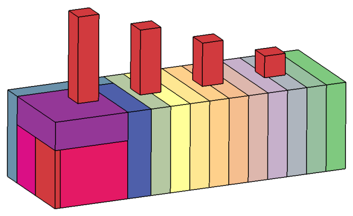
R of the most underload partition, if a cut position c_z satisfying $|R - N_x N_y c_z| < \epsilon \overline{W}$ cannot be found along the longest edge, the algorithm traverses possible cut positions c_z and c_y in the longest and second longest direction to minimize the difference $|R - N_x c_y c_z|$. Once cut positions are located, the block is cut into four sub-blocks and a block of size $N_x c_y c_z$ is assigned to the partition.

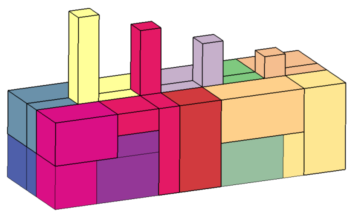
We denote the greedy algorithm [7] with the above fix as *pure greedy* (PG). PG strictly satisfies the tolerance for load imbalance with a potential trade-off of communication cost. When the prescribed tolerance is small, PG may create very small sub-blocks to fit in the remaining room of available partitions. This will result in a large number of small sub-blocks, which in turn increases both the communication volume and number of edge cuts. Moreover, the arrangement of small blocks in PG does not respect the connectivity of blocks and leads to increased communication volume. To see those effects, we partition Bump3D into 16 partitions with PG. As seen from Figure 3.3, PG creates small blocks at the end of the original large block. The communication volume is also large compared to our algorithms in the next section (Figure 3.6).

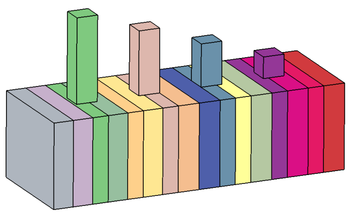
3.1.2 METIS + Over-Decomposition

The bottom-up algorithm first decomposes the original blocks into smaller sub-blocks. The sub-blocks along with their Block2Block connectivity are organized as a graph and partitioned using a graph partitioner. After partitioning, the sub-blocks within the same partition are merged if permissible by their shapes and connectivities to reduce the number of Block2Block boundaries.

Graph partitioners like METIS move vertices between partitions to achieve load balance and reduce communication cost. If there are too few vertices to move or the vertices have large variances in weight, the graph partitioner may produce imbalanced partitions. Therefore, it is desirable to have the number of small sub-blocks to be at least several times the number of partitions and to be of equal size. We denote such decomposition as over-decomposition and examine two over-decomposition methods. Both methods try to create sub-blocks of as large as one-quarter of the average workload $\overline{W}$. The first method is to decompose the original blocks into elementary blocks [10] i.e., blocks with only one boundary condition on each surface. If the elementary block is still too large to fit in one partition, it is further cut by our IF algorithm proposed in Section 3.2.2. The second method is to directly decompose the large blocks with IF.


(a) PG


(b) METIS, elementary cuts


(c) METIS, greedy cuts

Algorithm	Imbalance ratio	Total volume	Total edge cuts	Total cost
PG	0.035	2.19E+06	60	2.79E-3
METIS 1	0.214	1.72E+07	138	3.35E-3
METIS 2	0.103	2.47E+06	38	2.85E-3

(d) Metrics

Figure 3.3: Results of partitioning Bump3D into 16 partitions with PG and METIS+Over-Decomposition. The latency and bandwidth are set to $\alpha = 10^{-4}$(s) and $\beta = 10^{9}$(bytes/s) respectively. The tolerance for load imbalance is preset to 5%. Blocks of the same color belong to the same partition.

As shown in Figures 3.3, different decomposition strategies can result in very different partitions and metrics. The first method results in too many sub-blocks in this case. The second method generates simple connectivities in the graph of sub-blocks and therefore easier to partition. This indicates that the quality of partitions highly depends on the over-decomposition heuristic. However, to the best of our knowledge, there is no theoretical guidance for choosing the best over-decomposition heuristic. The practical choice is based on the developer's experience and can be application-specific. In this thesis, we will use our IF algorithm.

Note that in Figure 3.3, the four original small blocks are grouped with their connected sub-blocks in the same partition. This validates the graph partitioner's strength in preserving graph connectivities. However, it is also prone to load imbalance and large communication cost compared with PG and our algorithms in the next section (Figure 3.6).

3.2 Partitioning Algorithms Factoring Network Properties

In this section, we first present a novel cost function which unifies communication metrics (communication volume and edge cuts) and network properties (latency and bandwidth). Then, we detail our new partitioning algorithms using the above cost function for cutting large blocks and grouping small blocks.

3.2.1 Cost Criteria

The time to issue an inter-node message can be approximated using the $\alpha - \beta$ model [33, 34],

$$t_{\text{msg}} = \alpha + \frac{S}{\beta}, \tag{3.1}$$

where α denotes the latency, β is the bandwidth of the network, and S is the size of the message. The latency represents the start-up time for issuing a communication. The second term in Equation 3.1 describes the time to transfer a message of size S through the network. For a Block2Block boundary condition with surface area A, the communication cost t_{b2b} is estimated by

$$t_{b2b}(A) = \alpha + \frac{A \cdot \#\text{halo} \cdot S_{\text{cell}}}{\beta}. \tag{3.2}$$

The thickness of halo layers (#halo) and the size of data per grid cell (S_{cell}) depend on the specific CFD solver. We define the *Total Communication Cost* by summing the above cost over all the inter-node messages, i.e., all the Block2Block boundaries connecting blocks across different partitions,

$$\sum t_{\text{msg}} = \alpha \cdot \text{edge cuts} + \frac{\text{communication volume}}{\beta}. \tag{3.3}$$

Using Equation 3.3, we not only combine communication volume and edge cuts but also include the effects of network properties. Moreover, Equation 3.3 indicates that the edge cuts contribute to the overall latency and the communication volume's effect is coupled with the network's bandwidth.

Note that we use the $\alpha-\beta$ model solely as a cost function to capture both the communication metrics and network properties rather than as a prediction of the actual communication time. Such a prediction would not be realistic since this model's simple formulation is derived from several ideal assumptions about the network such as free of congestion, minimal queue lengths, etc [35]. To profile the communication time accurately, more realistic models such as logGPS [36] can be used and is beyond the scope of this thesis.

3.2.2 Algorithms for Cutting Large Blocks

The elementary operation in grid partitioning is to cut off a sub-block of a given workload. In Algorithm 3.1, we demonstrate how to identify the optimal cut of a large block using our new cost function. The function find_min_cut takes in a large block B, a workload W_{cut} to cut off, a tolerance ϵ for imbalance, and optionally a partition P, then returns the optimal cut introducing minimum communication cost δt_{cut}.

Algorithm 3.1 Find the cut of a block to fit in a given workload

```
1:  function FIND_MIN_CUT($B$, $W_{cut}$, $\epsilon$, cut, $P$)
        ▷ $B$: block to be cut.
        ▷ $W_{cut}$: workload to be cut off
        ▷ $\epsilon$: tolerance
        ▷ cut: data structure for cut info
        ▷ $p$: current partition (optional input, empty by default)
2:      $\delta t_{min} = \infty$
3:      for i = x, y, z do
4:          Get area of i's norm face $A_i$
5:          posFloor = FLOOR($W_{cut}(1-\epsilon)/A_i$)
6:          posCeil = CEILING($W_{cut}(1+\epsilon)/A_i$)
7:          for pos∈[posFloor, posCeil] do
8:              $\delta t_{cut} = \sum_{b2b \ni pos} \alpha + t_{b2b}(A_i) - \sum_{B_i \in P} t_{b2b}(\text{cut}, B_i)$
9:              if $\delta t_{cut} < \delta t_{min}$ then
10:                 $\delta t_{min} = \delta t_{cut}$
11:                 cut.pos = pos
```

Algorithm 3.1 starts by finding the possible cutting positions allowed by the tolerance ϵ (lines 5-6). The communication cost of cutting a block, δt_{cut}, is evaluated at line 8 for each position. The first term t_{cut} includes the latency introduced if the cut splits any Block2Block boundary on the orthogonal surfaces (each cutting plane is orthogonal to four surfaces of a block). Adding this term encourages the algorithm to align the sub-blocks' boundary with Block2Block boundaries. The second term adds the communication cost of the new Block2Block boundary created by the cut using Equation 3.2. When the cut-off sub-block is assigned to partition P, the Block2Block boundaries in contact with the blocks in P become shared memory accesses, which is much faster than inter-node communication and therefore subtracted from δt_{cut} in the last term. The last term takes advantage of the blocks' connectivity to reduce the overall communication cost. This way, we embed the cost function in Equation 3.3 into cutting large blocks.

Given a large block fit evenly into multiple partitions, we use the function find_min_cut in Algorithm 3.1 to improve two state-of-the-art algorithms, namely, *Recursive Edge Bisection*

(REB) [37] and *Integer Factorization* (IF).

Recursive Edge Bisection (REB) The classical REB recursively bisects the block at the longest edge until each resulting sub-block fits in a partition. Figure 3.4 illustrates REB by partitioning a 2D block into 7 partitions. REB starts by bisecting the block at its longest edge, the vertical edge in Figure 3.4, with a ratio of 4:3, and the two sub-blocks will be split into 4 and 3 partitions respectively during later steps. The bisection ratio is chosen to cut the block as even as possible while balancing the workload. The same procedure is repeated recursively in step 2 and 3.

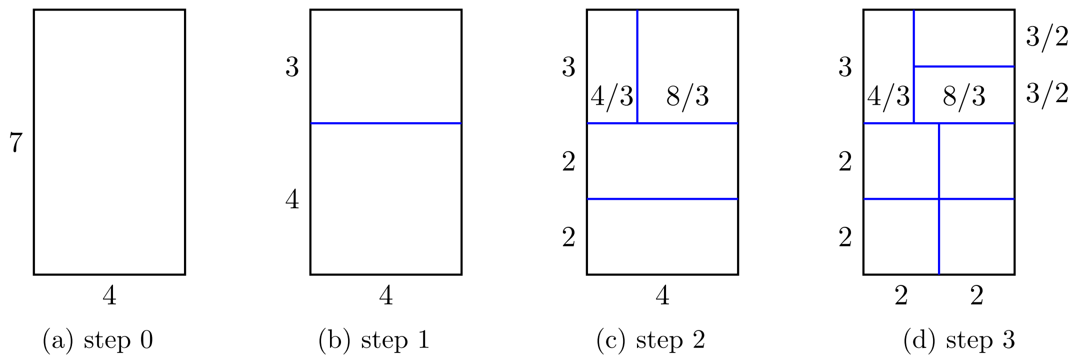


Figure 3.4: Partitioning a 2D block of size 4×7 to 7 partitions. The bisections are marked in blue. The number near each edges denotes its length.

Note that the bisection may intersect Block2Block boundaries and increases the edge cuts. We improve REB in Algorithm 3.2 using find_min_cut to find the optimal bisection position. The improved REB still chooses the bisection ratio based on the number of partitions (lines 4-5). Instead of cutting at the longest edge, the function find_min_cut traverses all the edges and finds the bisection position that aligns the Block2Block boundaries and introduces minimum communication cost (line 6). After bisecting the block, this procedure is applied to the two sub-blocks recursively (lines 7-9).

To verify that the improved REB factors the network properties, we recursively bisect the Bump3D grid into 16 partitions with latency $\alpha = 10^{-5}$s and $\alpha = 10^{-4}$s while fixing band-

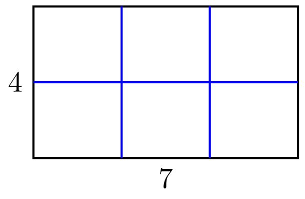
Algorithm 3.2 Improved Recursive Edge Bisection

```
1: function REB_BLOCK($B$, $n_p$)
       ▷ Block $B$ fits in $n_p$ partitions.
2:     if $n_p == 1$ then
3:         return
4:     $W = B$'s workload
5:     $W_l = W \cdot \frac{\lfloor n_p/2 \rfloor}{n_p}, W_r = W - W_l$
6:     FIND_MIN_CUT($B$, $W_l$, $\epsilon$, cut)
7:     cut $B$ into $B_l$ with workload $W_l$ and $B_r$ with workload $W_r$
8:     REB_BLOCK($B_l$, $\lfloor n_p/2 \rfloor$)
9:     REB_BLOCK($B_r$, $\lceil n_p/2 \rceil$)
```

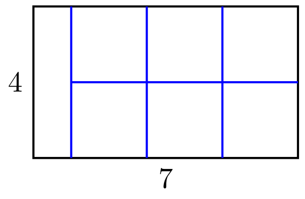
width $\beta = 10^9$byte/s. The resulting partitions are shown in Figure 3.6a and 3.6b respectively, and the metrics are listed in Table 3.6d. We can see that our improved REB creates much fewer edge cuts for larger latency values, successfully bounding the communication cost. Note that applying the partition created for a low-latency network directly to a high-latency network will be very inefficient for the excessive edge cuts. In this example, using the partition with $\alpha = 10^{-5}$s for $\alpha = 10^{-4}$s results in a communication cost of 8.17×10^{-3}s, much higher than current result (6.72×10^{-3}s). This further validates the necessity to factor network properties into grid partitioning.

Integer Factorization (IF) The classical IF finds a factorization of the given number of partitions n_p according to the block's length ratio, i.e., $n_p = n_x \cdot n_y \cdot n_z$ with $n_x : n_y : n_z \approx l_x : l_y : l_z$. Figure 3.5a exemplifies this by partitioning a 2D block of size 7×4 into the 6 partitions, where the ideal factorization is $6 = 3 \times 2$. There are two limitations to this decomposition. First, when n_p is prime, the unique factorization $1 \cdot 1 \cdot n_p$ may not be proportional to the length ratio. Second, it may split Block2Block boundaries and increase the communication cost. In [10], when n_p is prime, a sub-block is cut off to feed one partition and the algorithm searches for an optimum factorization of $n_p - 1$ to decompose the remaining sub-block. This remedy is shown in Figure 3.5b. We propose a more generalized solution in Algorithm 3.3 to address both limitations.

Algorithm 3.3 compares two cases. First, decompose the block B according to the factorization of n_p which introduces the minimum communication cost (lines 4-8). The cost of a factorization $t_{b2b}(B, n_x, n_y, n_z)$ is the maximum communication cost among $n_x \cdot n_y \cdot n_z$ sub-blocks. Second, cut off a sub-block, B_{cut} of average workload and decompose the remaining sub-block B_{rem} by the factorization of $n_p - 1$ which results in minimum overall communication cost (lines 9-13). If the first case costs less than the second, the factorization for block B is the optimum decomposition. Otherwise, the same comparison repeats on B_{rem}.



(a) 6 partitions



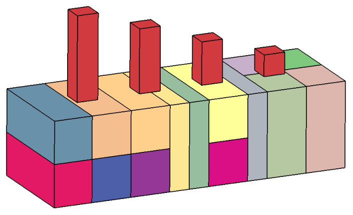
(b) 7 partitions

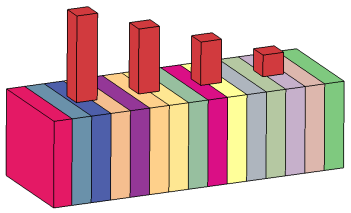
Figure 3.5: Examples of Partitioning a 2D block of size 7×4 with IF. For 6 partitions, the factorization close to the length ratio $7:4$ is $6 = 3 \times 2$. For 7 partitions, the leftmost sub-block is cut off to resolve the prime number. The remaining sub-block is factorized into 6 partitions based on the length ratio $6:4$.

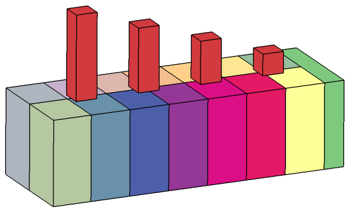
Algorithm 3.3 Integer Factorization

```
 1: function FACTORIZE_BLOCK($B$, $n_p$)
    ▷ Block $B$ fits in $n_p$ partitions.
 2:     if $n_p == 1$ then
 3:         return
 4:     $t^0_{min} = \infty$, $t^1_{min} = \infty$
 5:     for factorization $n_x \cdot n_y \cdot n_z = n_p$ do
 6:         if $t_{b2b}(B, n_x, n_y, n_z) < t^0_{min}$ then
 7:             $t^0_{min} = t_{b2b}(B, n_x, n_y, n_z)$
 8:             $n^0_x = n_x, n^0_y = n_y, n^0_z = n_z$
 9:     FIND_MIN_CUT($B$, $\overline{W}$, $\epsilon$, cut)
10:     for factorization $n_x \cdot n_y \cdot n_z = n_p - 1$ do
11:         $t^1 = \max(t_{b2b}(B_{cut}), t_{b2b}(B_{rem}, n_x, n_y, n_z))$
12:         if $t^1 < t^1_{min}$ then
13:             $t^1_{min} = t^1$
14:     if $t^0_{min} < t^1_{min}$ then
15:         cut $B$ by $n^0_x, n^0_y, n^0_z$
16:     else
17:         FACTORIZE_BLOCK($B_{rem}$, $n_p - 1$)
```

Figure 3.6c and Table 3.6d illustrate the 16 partitions created by IF. Each cut in IF cuts through the entire block. Therefore, compared with REB, IF is more apt to align Block2Block boundaries and reduces the edge cuts. On the other hand, the flexibility of choosing cut positions renders REB to have less communication volume than IF.


(a) REB


(b) REB $\alpha = 10^{-4}$s


(c) IF

Algorithm	α	Imbalance ratio	Total volume	Total edge cuts	Total cost
REB	1.0E-5	0.047	1.57E+06	66	2.23E-3
REB	1.0E-4	0.034	2.32E+06	44	6.72E-3
IF	1.0E-5	0.035	1.61E+06	66	2.27E-3

(d) Algorthmic Metrics

Figure 3.6: Partitions created by the different algorithms for the Bump3D grid when $n_p = 16$. Blocks of the same color belong to the same partition.

3.2.3 Algorithms for Grouping Small Blocks

To design partitioning algorithms for small blocks, we initialize the communication cost by summing up all the Block2Block connections as inter-node communication. Grouping several small blocks into one partition can convert the blocks' inter-node communication to shared memory access and reduce the overall communication cost. Therefore, the grouping algorithm should pack blocks connected by large Block2Block boundaries into the same partition. Keeping this goal in mind, we propose the following two algorithms.

Cut-Combine-Greedy (CCG) The idea of CCG is to keep feeding the optimal small block that converts the most inter-node communication to shared memory copy to the partition until the partition's remaining room is fulfilled. Algorithm 3.4 depicts the function find_min_company to identify the optimal small block. The function takes in the partition to fulfill, P, and the remaining room, W_{ub}, then finds the ID of the optimal small block, cmpny. It traverses all the unassigned blocks. If a block fits in the remaining room W_{ub}, CCG computes the saved communication cost by subtracting the cost of all the Block2Block boundaries connecting that block and partition P (line 6). If a block exceeds W_{ub}, CCG uses the function find_min_cut in Algorithm 3.1 to the find the optimal cut-off sub-block that minimizes the communication cost. The optimal block is identified as the block or sub-block that saves the most communication cost (lines 7-9 and 12-14). This algorithm is a greedy heuristic since each optimal block only minimizes the communication cost for that step but not for the final result.

Algorithm 3.4 Find company to fit in one partition

```
 1: function FIND_MIN_COMPANY($P$, $W_{ub}$, cmpny )
      ▷ $P$: the partition to be filled
      ▷ $W_{ub}$: upper bound of the company's work load
      ▷ cmpny: ID of the company block
 2:     $\delta t_{min} = \infty$
 3:     for block $B \in$ {unassigned blocks} do
 4:         Get $B$'s work load $W$
 5:         if $W < W_{ub}$ then
 6:             $\delta t = -\sum_{B_i \in P} t_{b2b}(B, B_i)$
 7:             if $\delta t < \delta t_{min}$ then
 8:                 $\delta t_{min} = \delta t$
 9:                 cmpny = $B$.ID
10:         else
11:             FIND_MIN_CUT($B$, $W_{ub}$, $\epsilon$, cut, $P$)
12:             if cut.$\delta t < \delta t_{min}$ then
13:                 $\delta t_{min} = \delta t$
14:                 cmpny = cut.ID
```

Figure 3.7 exemplifies CCG using 4 small blocks connected as a graph where each circle denotes a small block with its workload marked at the center and each edge represents a Block2Block connection with its communication cost noted. The average workload per partition is preset to 160. For a partition containing only block A (marked blue) in Figure 3.7a, CCG moves on to include block B because it reduces the most communication cost (6) to shared memory copy as shown in Figure 3.7b. In the *step 3*, CCG first tries to append block C to the partition to hide 8 (4+4) units of communication cost. However, this would result in a total workload of 180 ($A + B + C = 180$), exceeding the average workload $\bar{W} = 160$. Instead, CCG cuts block C into sub-block C_1 and C_2 with find_min_cut. C_1 fits in the partition's remaining room and introduces minimum communication cost.

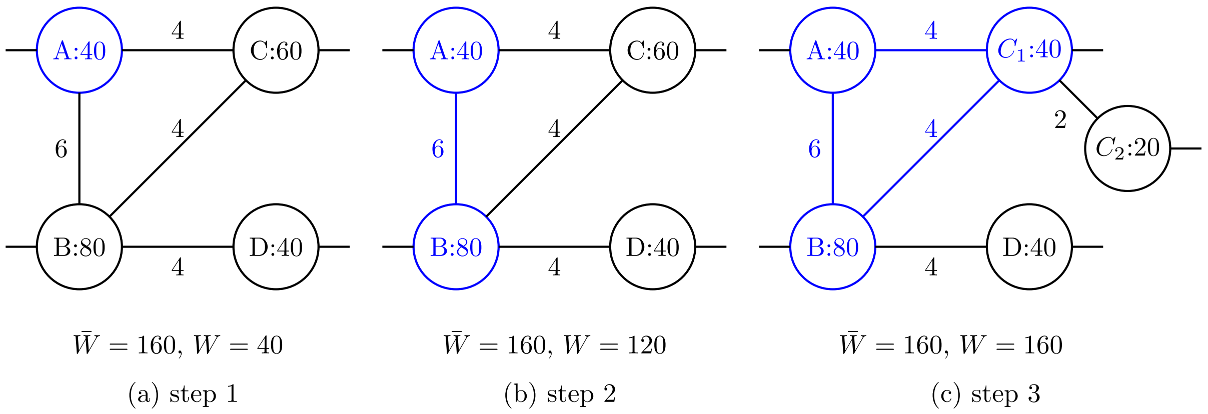


Figure 3.7: An example of grouping small blocks with CCG. CCG keeps including the small block that reduces the most inter-node communication to shared memory copy. If that block exceeds the available space of one partition, CCG will cut the block and only add the sub-block that fits.

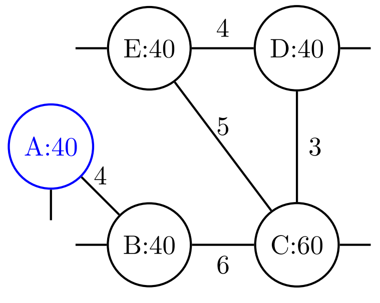
Graph-Growth-Sweep (GGS) The idea of GGS is to repeatedly use graph growing to group small blocks. As shown in Algorithm 3.5, GGS first assigns each empty partition a small block (lines 4-5). Using the small block as a seed, GGS starts the graph growing procedure for each partition. If including a block that has been added to another partition can reduce the overall communication cost, the block will be saved for the current partition (lines 8-14). The saved blocks are sorted by the amount of communication reduced and then assigned to the current partition until it is fulfilled. We define the application of graph growing to all partitions as one sweep. The partitions are swept repeatedly until no more blocks can be shifted across partitions to reduce the communication cost.

Algorithm 3.5 Graph-Growth-Sweep (GGS)

```
1: Assign each partition a small block
2: while blocks can be assigned do
3:     for all $p \in P$ do
4:         if $p$.empty() then
5:             Assign a block to $p$
6:         else
7:             for all Block $B$ connected to blocks in $p$ do
8:                 if $B$ fits in $p$'s room then
9:                     $B.\delta t = -\sum_{B_i \in p} t_{b2b}(B, B_i)$
10:                    if $B$.assigned() then
11:                        $p_B :=$ $B$'s partition
12:                        $B.\delta t = B.\delta t + \sum_{B_i \in p_B} t_{b2b}(B, B_i)$
13:                    if $B.\delta t < 0$ then
14:                        Save $B$ in block array $blks$
15:            Sort $blks$ in ascending order of $blks[i].\delta t$
16:            while !$p$.full() and $i < blks$.size() do
17:                Assign $blks[i]$ to $p$
18:                i++
```

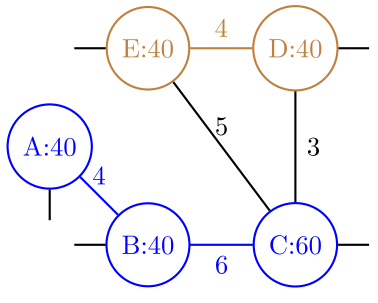
Figure 3.8 illustrates the process of GGS by grouping 5 small blocks into two partitions. The average workload per partition is preset to $\bar{W} = 120$. We initialize the two empty partitions with block A and D respectively. After one sweep, the two partitions marked by blue and brown in Figure 3.8b contains connected blocks found via graph growing. Note that the Block2Block connections between block C and the second partition (marked brown) introduce a communication cost of 8, more than the cost (6) saved by assigning C to the first partition (marked blue). Therefore, GGS shifts C to the second partition during the second sweep, reducing the overall communication cost by 2 (8 - 6). As a result, the first partitions become underloaded and will be filled with other blocks in future sweeps.

Compared with CCG, GGS avoids cutting small blocks as much as possible and therefore is better at reducing edge cuts. On the other hand, CCG is more aggressive at minimizing communication volume for its greedy nature.



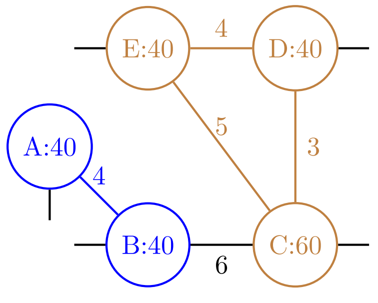
$\bar{W} = 120$, $W_1 = 40$, $W_2 = 40$

(a) Inital state



$\bar{W} = 120$, $W_1 = 120$, $W_2 = 80$

(b) 1st sweep



$\bar{W} = 120$, $W_1 = 80$, $W_2 = 120$

(c) 2nd sweep

Figure 3.8: An example of grouping 5 small blocks into 2 partitions with GGS. The partitions are marked by blue and brown respectively. The average workload $\bar{W}$ is preset to 120. W_1 and W_2 denotes the workload for the first and second partition respectively. Two sweeps are depicted.

3.2.4 Combined Algorithms and Partition Adjustment

We combine the algorithms for cutting large blocks and grouping small blocks via Algorithm 3.6. The algorithm traverses all the blocks. If a block's workload exceeds the average, it will be cut into a large sub-block B_l that fits evenly into multiple partitions and a small sub-block B_s which is appended to the list of small blocks. The large sub-block B_l is partitioned with REB or IF. After all the large blocks are partitioned, the small blocks are grouped with CCG or GGS.

Unlike PG, both REB and IF may generate blocks that exceed the prescribed load imbalance tolerance at the trade-off of reducing communication cost. In case the computation becomes too imbalanced, we use a greedy heuristic to adjust the workload between overloaded and underloaded partitions. We denote a shift between two partitions as a sub-block cut from a block in one partition and moved to the other partition. The adjustment heuristic sorts all possible shifts from overloaded partitions to underloaded partitions according to its communication cost using an efficient *Bucket List* structure introduced in [38]. The adjustment starts from the shift that results in the minimum communication cost and updates the par-

Algorithm 3.6 Combine the algorithms for cutting large blocks and grouping small block for general multi-block grids

```
1: Compute average workload per partition $\bar{W}$
2: smallBlocks = []
3: for block $B \in$ {all blocks} do
4:     Get $B$'s work load $W$
5:     if $W > \bar{W}$ then
6:         $W_l = \lfloor W/\bar{W} \rfloor \cdot \bar{W}$
7:         FIND_MIN_CUT($B$, $W_l$, $\epsilon$, cut) $\rightarrow$ large sub-block $B_l$, small sub-block $B_s$
8:         Partition $B_l$ with REB or IF
9:         smallBlocks.append($B_r$)
10:    else
11:        smallBlocks.append($B_r$)
12: Group small Blocks with CCG or GGS.
```

tition and *Bucket List* until all the overloaded partitions are within the given imbalance tolerance.

3.3 Experiments and Results

In Section 3.2, we proposed a novel cost function and incorporated it into two heuristics for cutting large blocks (REB and IF) and two novel algorithms for grouping small blocks (CCG and GGS). This leads to four combinations namely, REB+CCG, REB+GGS, IF+CCG, and IF+GGS. To evaluate their performance, we apply all four combinations to two multi-block structured grids including Bump3D shown in Figure 3.2 and a rocket model based on SpaceX's Falcon-Heavy. We simulate a benchmark solver using a 3D 13-points star-shaped stencil with the resulting partitions. The experiments are conducted on the Mira supercomputer at the Argonne National laboratory. For the remainder of this section, we first describe the test grids and the experiment setup in detail, then analyze the results for the above grids.

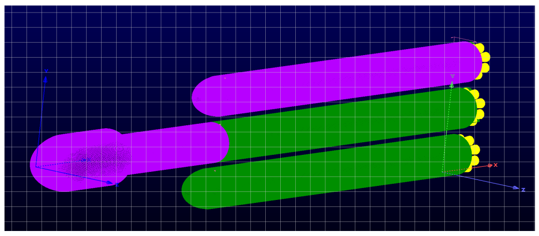
3.3.1 Experiment Setup

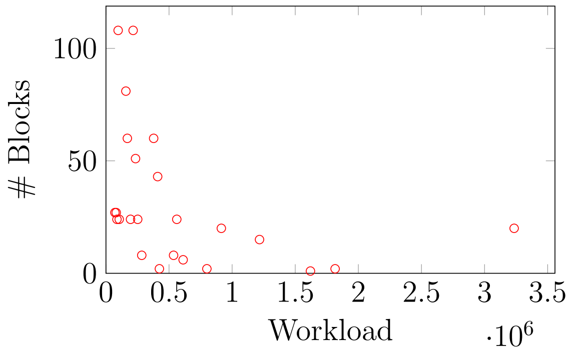
Test Multi-block Structured Grids The first test grid is Bump3D introduced in Section 3.1. Here we refine the grid 4 times in each dimension as shown in Table 3.1. Since the grid is dominated by a single large block (block 0), this test is aimed at evaluating the partitioner's capability to decompose large blocks.

Table 3.1: Refined blocks of the Bump3D grid.

Block ID	0	1	2	3	4
Workload	7.3E+7	2.6E+5	5.2E+5	7.8E+6	1.0E+6

The second grid is based on the released geometric info of SpaceX's Falcon-Heavy rocket [39]. As shown in Figure 3.9, the complex geometry results in 769 blocks with varying block workloads. There are hundreds of small blocks and a few blocks significantly larger than the others. Therefore, this grid will challenge the partitioner in both cutting large blocks and grouping small blocks.


(a) Gemeotry



(b) Block Distribution

Figure 3.9: A Rocket Model consisting of 769 blocks.

Benchmark Solver The experiments are conducted with a hybrid MPI+OpenMP Jacobi solver. The computation is a finite difference scheme using a 3D 13-points star-shaped stencil. The communication consists of the exchange of one double precision variable i.e., 8 bytes per halo cell, for two halo layers extended outside each block's boundary. The numerical experiment is done in the manner outlined in Algorithm 2.2 for 512 iterations and the average running time per iteration is reported. We are using the MPI+threads programming model where we assign one partition per processor node. Note that both the estimated communication cost and the measured communication performance only include the effect from inter-node communication, excluding shared memory copy.

Platform and Architecture We run the benchmark solver with the partitioner on the Mira supercomputer at the Argonne National Laboratory. Each node has a PowerPC A2 processor with 16 cores clocked at 1.6 GHz with 1GB DDR3 memory. The interconnect is a 5D torus. We obtain the latency and bandwidth of Mira using a *ping-pong* benchmark, i.e., timing two adjacent nodes exchanging messages and fitting Equation 3.1 with the least squares method as shown in Figure 3.10. The latency and bandwidth are measured as 1.73e-05 s and 1.77e+09 bytes/s respectively.

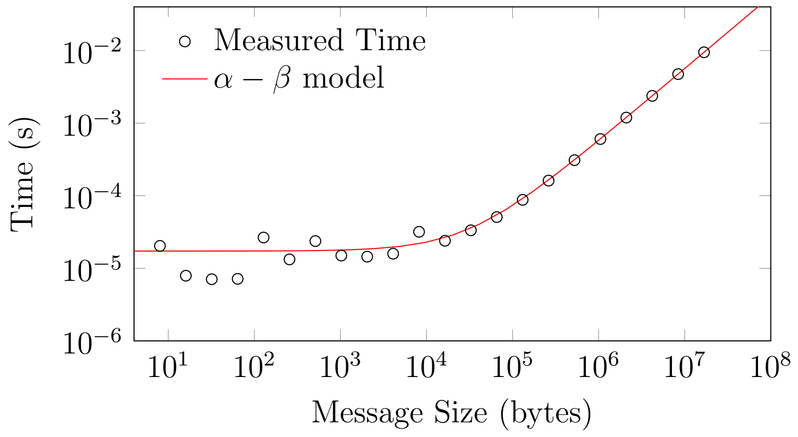


Figure 3.10: $\alpha - \beta$ model fitted on Mira.

3.3.2 Metrics

Table 3.2a shows the imbalance ratio and the total number of sub-blocks of different partitioning algorithms for Bump3D and the rocket models. Figure 3.11 compares the communication volume, edge cuts, and communication cost of different heuristics for both grids. All the metrics are obtained for 64 - 4096 partitions with the latency and bandwidth of Mira as inputs. Note that the number of partitions equals the number of processor nodes since we assign one partition per node.

Bump3D This test challenges REB and IF's capabilities to minimize the communication cost since the decomposition of Bump3D is dominated by cutting a single large block. Using CCG or GGS leads to similar results. Across the board, all the schemes result in significantly less communication volume compared to PG at all processor counts. This is because PG does not take into account the connections between blocks at all. On the other hand, METIS has the highest number of total edge cuts in Figure 3.11c, since the over-decomposition creates an excessive number of sub-blocks as shown in Table 3.2a. As a result, both PG and Metis have a higher total communication cost compared to REB and IF. The results also depict that IF leads to more communication volume but fewer edge cuts than REB, which confirms the analysis in Section 3.2 that IF is better at reducing edge cuts while REB is better at reducing communication volume.

The total communication cost captures the effect of both communication volume and edge cuts as given by Equation 3.1. For instance, on 64 nodes, PG creates fewer edge cuts but more communication volume than all the other algorithms. The total cost indicates that in this case communication volume plays a more important role than edge cuts and PG results in a higher cost than the other algorithms. On the other hand, on 4096 nodes, IF has the lowest communication cost. This indicates that at the highest processor count used for Bump3D, reducing the edge cuts is more critical than reducing the communication volume. This further validates the need to factor network properties into grid partitioning.

Note that in Table 3.2a, PG is the only algorithm that bounds the load imbalance within the given tolerance ($\epsilon = 5\%$) for all cases while METIS and IF cause more than 20% imbalance at 4096 nodes. Unlike PG, REB and IF don't pay the penalty of creating an excessive number of sub-blocks to satisfy the load balance tolerance. This trade-off between load balance and communication cost will be further discussed in Section 3.3.3.

Rocket Model For 64-256 nodes, METIS results in the lowest communication volume, edge cuts, and communication cost. Second to METIS are algorithms using CCG to group small blocks. To explain this, we introduce two parameters $\bar{n}_{sm}$ and $\%n_{psm}$. $\bar{n}_{sm}$ denotes the average number of small blocks per partition after cutting large blocks and $\%n_{psm}$ is the percentage of partitions filled with small blocks. As seen from Table 3.2b, for 64-256 nodes, more than 60% of the partitions are made up of small blocks and such partitions have more than 4 sub-blocks on average. Therefore, the partitioner's ability to group small blocks determines the partition's quality. As a graph partitioner, METIS is good at exploiting connectivity to reduce communication cost. As shown in Figure 3.12, although METIS creates more halos than other algorithms, it maps a large percentage of halo exchange to shared memory copy. A similar trend can also be observed for CCG, which uses connectivity in a greedy fashion to reduce communication. On the contrary, PG does not take into account the blocks' connectivity and hence introduces the highest communication cost.

For 512-4096 nodes, $\bar{n}_{sm}$ decreases with the number of partitions and the significance of grouping small blocks damps. METIS loses its strength at large nodes count (1024 - 4096) and produces the most edge cuts and the worst communication cost due to the large number of small sub-blocks created by over-decomposition. REB+GGS results in the lowest communication cost because REB minimizes the communication volume in cutting large blocks while GGS avoids introducing new edge cuts between small blocks. Surprisingly, PG results in comparable or even less cost compared with our algorithms at 1024 and 2048 nodes. Unlike Bump3D where the largest block occupies the majority of the partitions, the largest block here only occupies 12 and 25 partitions on 1024 and 2048 processors respectively. As a result, PG's greedy heuristic of cutting at the longest edge leads to a near optimum partition.

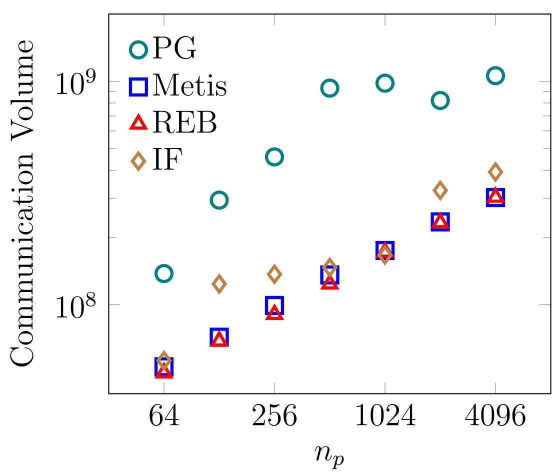
Table 3.2: Results of load imbalance and the number of sub-blocks (n_{sub}) for different partitioning heuristics for Bump3D and the rocket model. In all tests, the number of partitions (n_p) equals the number of processor nodes used, which spans from 64 to 4096. The bold black and red fonts represent the best and worst metrics respectively.

(a) Bump3D

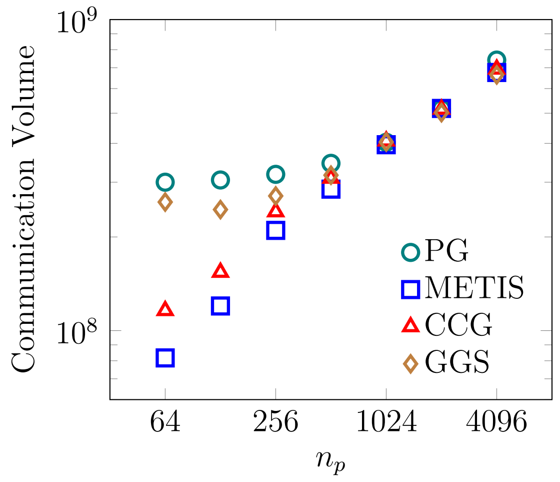
n_p	64	128	256	512	1024	2048	4096
Imbalance PG	0.035	0.046	0.045	0.048	0.042	0.047	0.050
Imbalance METIS	**0.256**	**0.517**	**0.491**	**0.516**	**0.226**	**0.253**	**0.257**
Imbalance REB+CCG	0.035	0.050	0.044	**0.075**	**0.106**	**0.100**	**0.134**
Imbalance REB+GGS	0.035	0.050	0.044	**0.075**	**0.106**	**0.100**	**0.122**
Imbalance IF+CCG	0.035	0.019	0.035	0.035	0.043	**0.086**	**0.214**
Imbalance IF+GGS	0.035	0.019	0.035	0.035	0.043	**0.086**	**0.214**
n_{sub} PG	65	211	275	601	1090	2505	4855
n_{sub} METIS	**89**	**166**	**421**	**866**	**1638**	**3532**	**7103**
n_{sub} REB+CCG	**67**	131	259	514	**1025**	**2048**	**4096**
n_{sub} REB+GGS	**67**	131	259	514	**1025**	**2048**	**4096**
n_{sub} IF+CCG	**67**	**129**	**258**	**513**	**1025**	**2048**	**4096**
n_{sub} IF+GGS	**67**	**129**	**258**	**513**	**1025**	**2048**	**4096**

(b) Rocket Model

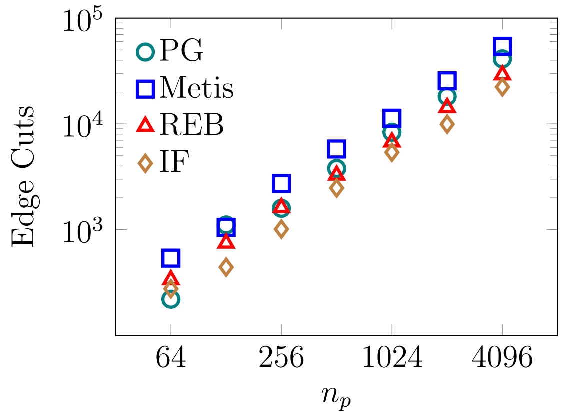
n_p	64	128	256	512	1024	2048	4096
$\overline{n_{sm}}$	12	7.1	4.3	2.6	1.7	1.6	1.8
$\%n_{psm}$	100%	84.3%	69.5%	57.2%	41.5%	22%	9.8%
Imbalance PG	0.005	0.028	0.022	0.040	0.047	0.049	0.049
Imbalance METIS	**0.097**	**0.210**	**0.283**	**0.311**	**0.729**	**0.973**	**0.236**
Imbalance REB+CCG	**0.100**	**0.100**	**0.100**	**0.097**	**0.100**	**0.089**	**0.098**
Imbalance REB+GGS	0.049	0.049	0.034	0.048	0.047	**0.066**	**0.118**
Imbalance IF+CCG	**0.100**	**0.100**	**0.100**	**0.059**	**0.097**	**0.078**	**0.099**
Imbalance IF+GGS	0.049	0.049	0.034	0.048	**0.059**	**0.082**	**0.109**
n_{sub} PG	**769**	**789**	**847**	995	**1466**	2445	4585
n_{sub} METIS	**810**	**843**	**1005**	**1527**	**2652**	**4715**	**9006**
n_{sub} REB+CCG	787	817	927	1147	1668	2620	4620
n_{sub} REB+GGS	**769**	**789**	**847**	**993**	1477	2443	4443
n_{sub} IF+CCG	787	817	927	1147	1668	2617	4614
n_{sub} REG+CCG	**769**	**789**	**847**	**993**	1486	**2441**	**4429**



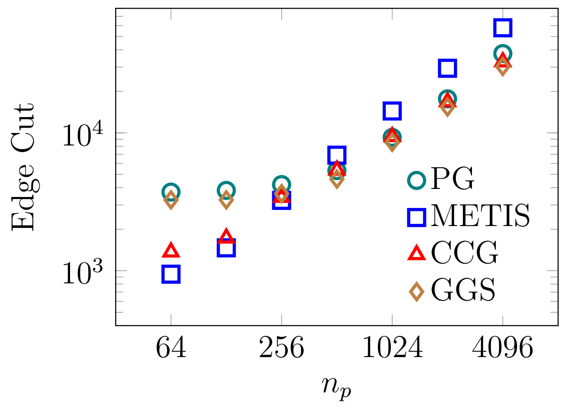
(a) Bump3D, Communication Volume



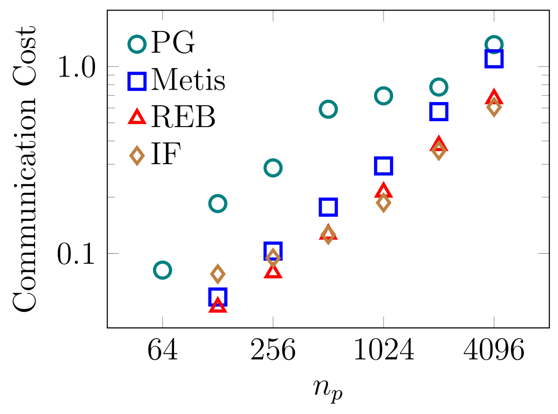
(b) Rocket Model, Communication Volume



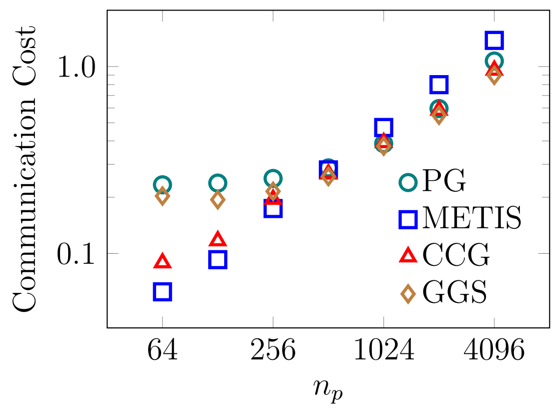
(c) Bump3D, Edge Cuts



(d) Rocket Model, Edge Cuts



(e) Bump3D, Communication Cost



(f) Rocket Model, Communication Cost

Figure 3.11: Metrics of different partitioning algorithms for Bump3D and Rocket Model. n_p denotes the number of partitions. For Bump3D, IF and REB are coupled with CCG. For Rocket Model, CCG and GGS are applied with REB.

However, at 4096 nodes, the largest block occupies 51 partitions and PG's disadvantage of creating excessive blocks re-appears and increases the communication cost.

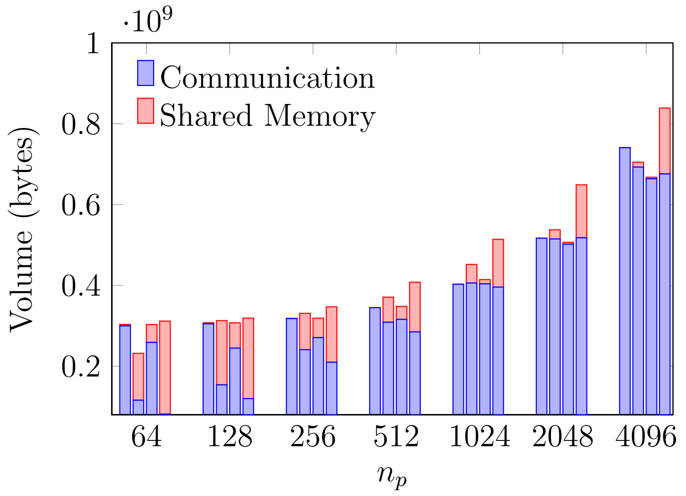
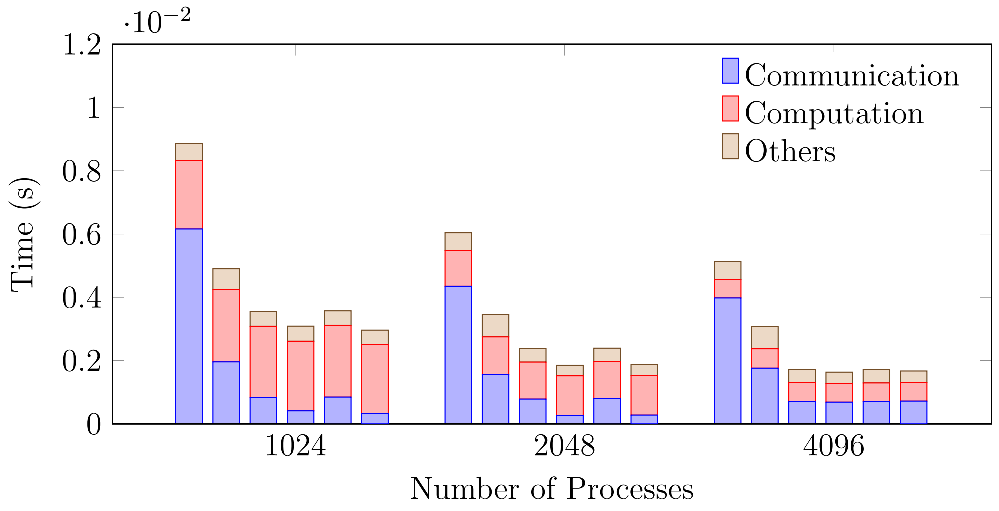


Figure 3.12: The total amount of halo data for PG, REB+CCG, REB+GGS, and METIS from left to right for the rocket model.

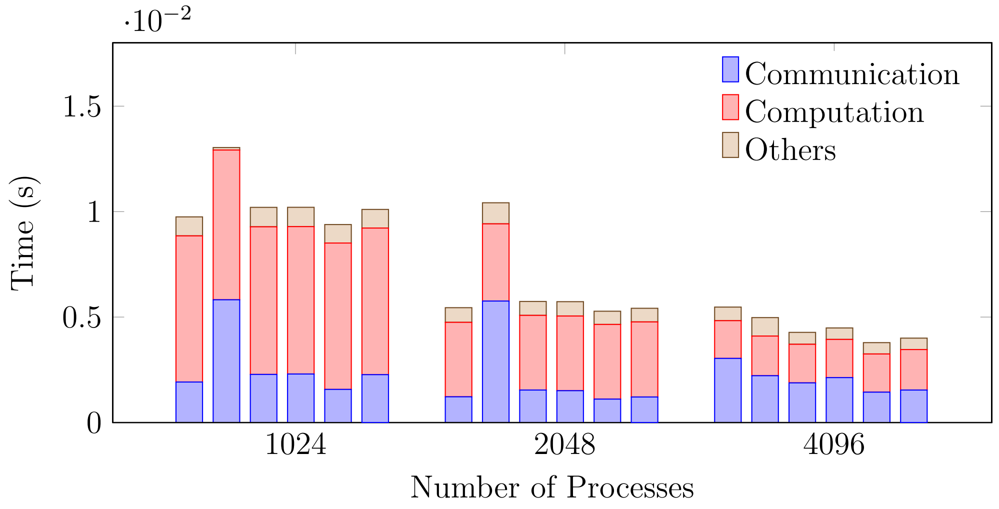
3.3.3 Performance Results

We report the running times for different partitioning algorithms coupled with the Jacobi solver at large nodes count (1024 - 4096) in Figure 3.13. The time consists of communication, computation, and others which is mainly the time for packing and unpacking the halos for inter-node communication (refer to Algorithm 2.2)

Bump3D The results in Figure 3.13a confirms two expectations based on the trend of communication cost in Figure 3.11e. First, PG and METIS achieve the worst and the second worst performance for 1024-4096 nodes respectively. Second, IF achieves the best performance across the broad. Compared to PG, IF obtains 5.8-15x speedup in communication and up to 3.14x in the overall running time. IF's superior performance for Bump3D indicates the significance of reducing edge cuts at large nodes count, further validating the necessity to incorporate edge cuts in the cost function. Moreover, Table 3.2a shows that IF results in as much as 20% load imbalance while PG maintains the imbalance under the prescribed tolerance. This concludes that the trade-off of computation balance for lower communication cost can be beneficial especially for large nodes count.



(a) Bump3D



(b) Rocket Model

Figure 3.13: The average running time of 1 iteration of the Jacobi solver coupled with the different paritioners for the Bump3D (left) and Falcon Heavy (right) grids. Each bar from the left to right represent the performance of the solver coupled with PG, METIS, REB+CCG, IF+CCG, REB+GGS, and IF+GGS respectively.

Rocket Model The performance results for the rocket model as seen from Figure 3.13 are likewise consistent with the communication cost estimation in Figure 3.11f except for

METIS at 4096 nodes. METIS achieves the worst performance at 1024 and 2048 nodes as predicted by the communication cost in Figure 3.11f. However, the unexpected reduction of its communication time at 4096 nodes requires further investigation. PG leads to slightly better performance than algorithms using CCG and IF+GGS at 1024 and 2048 nodes but has the worst runtime at 4096 nodes. As explained in Section 3.3.2, the sub-optimal performance at 4096 node is due to PG's drawback of creating too many small blocks for partitioning large blocks. The best performance comes from REB+GGS, which achieves 1.5× overall speedup and 2.1× better communication performance compared to PG.

For both grids, the communication time stops scaling at 4096 nodes for the Jacobi solver while the communication cost estimated by the $\alpha - \beta$ model still continues to scale. As remarked in Section 3.2.1, the $\alpha - \beta$ model is only used as a cost function rather than as a prediction of the communication runtime. Nevertheless, it is still worthwhile to analyze the gap between the cost model and the actual measured time. Two factors may contribute to this gap. First, in our performance experiments, we ignore the topology of the network. Communication cost between any two partitions is estimated based on the latency and bandwidth values measured using a *ping-pong* benchmark between two adjacent nodes. However, two partitions may be mapped to two nodes that are physically separated by several hops. Second, some fraction of the total communication time is spent on waiting for other processes to issue their messages sends. This idle time may take up to 80% of the total communication time [40] for some applications. The waiting time, in turn, depends on several factors such as the overall time of communicating processes, load imbalance, and the congestion in the network. Nevertheless, as observed by the experimental data, the cost function is still an effective guide for grid partitioning and leads to better performance than the current state-of-the-art heuristics for complex structured grids.

3.4 Related Work

Among the top-down strategies, the greedy heuristic [7] is the most widely adopted method. Recursive Edge Bisection (REB) [37] is a good alternative to the former. In [41], a greedy heuristic is combined with REB. At any step, the largest block is assigned to the most underloaded partition. After assigning all blocks, if a certain number of partitions is overloaded, then the same number of large blocks is bisected in half at the longest edge. This process is repeated until all partitions are within the load imbalance tolerance. It is hard to say if this hybrid approach is better than the classical greedy heuristic [7] since no comparison has been made.

A decomposition according to the block's aspect ratio is optimum in the number of edge cuts. This idea is used in [10] for 2D grids. Compared to REB [37], this strategy results in less imbalance but more communication volume. No performance comparison is made. More recently, this algorithm has been extended to 3D problems [42]. Given the number of partitions, all their test grids are made of blocks larger than the average workload. Therefore, the grouping of small blocks is not clearly shown.

In [12], large blocks are cut into cubic sub-blocks as much as possible because, for a fixed volume, the cubic shape has the minimum surface area. The residual blocks are assumed to have a minor effect on performance. Coupled with a CFD solver, they demonstrate $1.2-1.4\times$ speedup up to 800 processes against the greedy heuristic [7]. However, the communication time stops scaling between 300-400 processes.

In the above works [7, 10, 12, 37, 41], the communication volume can be viewed as the cost function. In [11, 43], the total running time is chosen as the cost function and balanced by grouping small sub-blocks with a greedy heuristic similar to [7]. The communication time is estimated using the network's bandwidth but misses the effect of the latency.

Compared to top-down strategies, unfortunately, there is considerably limited literature on *bottom-up* strategies for multi-block structured grids. The idea is first proposed in [13] where the original blocks are split into small sub-blocks, the number of which needs to be three times more than the partitions. Then, a graph partitioner is used to partition the small blocks. This algorithm is compared with *top-down* algorithms on a 2D multi-element and demonstrates improved performance. More recently, a new method for decomposing the original blocks is proposed in [14] which results in fewer blocks and communication volume than decomposing the blocks with REB. The effect on the performance of a solver is not yet assessed.

To the best of the author's knowledge, we present the very first work to unify the algorithmic metrics and network properties in partitioning multi-block grids. Our grid partitioner not only combines the communication volume and edge cuts but also factors the network's latency and bandwidth.

3.5 Summary and Future Work

In this chapter, we have introduced a novel cost function based on the $\alpha - \beta$ model to express the communication cost incurred in partitioning multi-block structured grids. To the best of our knowledge, this is the first cost model that unifies the communication volume, edge cuts, and network properties. Based on the cost function, we have improved two heuristics for cutting large blocks (REB and IF) and proposed two new algorithms for grouping the small blocks (CCG and GGS). Those algorithms are combined to derive a new grid partitioner following the top-down strategy. We evaluate the partitioner with a hybrid MPI+OpenMP Jacobi solver for two test structured grids, Bump3D and the rocket model, on the Mira supercomputer. Compared with PG, our algorithms result in $5.5-15\times$ speedup in communication for Bump3D and $1.5\times$ speedup for Falcon Heavy at 4096 nodes.

We believe the present grid partitioner is a critical precursor to achieve portable communication performance for CFD solvers. Nonetheless, factoring the network's latency and bandwidth is not good enough. The actual communication time is also influenced by other factors including the network contention, synchronization, etc. A further step towards portable performance is to incorporate more accurate communication models such as [36, 44].

The present partitioner follows the top-down strategy, which is still largely a greedy strategy. The algorithms except for GGS are aimed at minimizing the communication cost at each step of cutting large blocks or grouping small blocks. Like any greedy method, our partitioner is prone to missing the global optimal solution. This limitation can be overcome using *reinforcement learning* (RL). A well-tuned RL model should be able to predict the optimal cutting/grouping actions to achieve the global optimal partition. We will pursue this in our future research.

Chapter 4

Pencil: Piplined Distributed Stencil Computation

After designing an optimal partitioner for structured grids, we move on to the iterative process of solver computation and halo communication in the CFD workflow. As described in Chapter 2, stencils are the dominant computational pattern for CFD solvers using structured grids. In stencil computations, both the number of floating-point operations (flops) and memory accesses are proportional to the grid size. As a result, stencils are typically memory-bound on modern architectures with high machine balance. Moreover, stencils are also characterized by significant data reuse. We illustrate this by solving a 2D Laplace equation, $\nabla^2 p = 0$ using Finite Difference on a uniform grid (constant cell size h). After discretization, the value at grid cell (i, j) is advanced from the n^{th} to the $(n+1)^{th}$ iteration by averaging its neighboring cells,

$$p_{i,j}^{n+1} = w_0 \frac{p_{i-1,j}^n + p_{i+1,j}^n + p_{i,j-1}^n + p_{i,j+1}^n}{4} + w_1 p_{i,j}^n, \tag{4.1}$$

where w_0 and w_1 denote the weights in average. This corresponds to the 2D star-shaped stencil in Figure 2.2a. As the computation moves from cell (i, j) to $(i, j + 1)$, the values $p^n_{i,j}$ and $p^n_{i,j+1}$ are reused. Similar reuse can also be found in the i^{th} dimension, i.e., the entire i^{th} row is reused to update the $(i + 1)^{th}$ row.

Cache tiling, specifically temporal tiling with polyhedral techniques can effectively exploit data reuse patterns to optimize memory-bound applications. Temporal tiling views the iterations traversing the spatial dimensions and the iterations in time as one *iteration space* and decomposes this space into polyhedral tiles. The iterations in one tile are fused and executed entirely in cache, significantly improving cache reuse and performance. Several temporal tiling algorithms have been proposed in literature such as overlapped tiling [45–48], trapezoidal tiling [49, 50], diamond tiling [51–54], and tessellating tiling [55, 56]. A detailed summary of the state-of-the-art tiling algorithms can be found in [15, 16]. Tilings can also be applied via automatic parallelization tools like Pluto [57, 58] and domain-specific compilers like Pochoir [59], PolyMage[60], and Halide [61]. As highlighted in Chapter 2, the above tiling algorithms and tools are aimed at stencil computations on shared-memory machines and, more strictly, for single-block grids resulting in perfectly nested loops. However, in realistic CFD applications, we are confronted with multi-block grids distributed across multiple processors. To the best of our knowledge, none of the above state-of-the-art tiling algorithms and tools apply to multi-block grids on distributed systems.

In addition to optimizing the memory-bound computation, we also want to reduce the communication time. We demonstrated in Chapter 3 that an optimal partition can effectively minimize the communication cost. Here we want to further hide the communication running time by overlapping with computation. MPI implements non-blocking routines for this purpose. However, as discovered by several studies [62–65], simply inserting computation between non-blocking send/receive (put/get) and wait routines (window fences) does not result in an overlap. The MPI standard [25] does not specify that communication can progress outside MPI functions. So the actual communication is likely to congest in the `MPI_Wait` calls. There are two remedies. The first is to dedicate one thread or one core to communication [62, 63, 66–69]. Alternatively, one can repeatedly poll `MPI_Test` to urge the network to make progress, which has been proven effective by various studies and applications [18, 64, 70, 71]. Both methods need the computation to be divided into communication-dependent and independent parts. In distributed stencil computation, this requires dividing the block into a halo-dependent part and a halo-independent part. Such division shall not forfeit the benefit from temporal tiling. However, the state-of-the-art overlap algorithms that work with cache tiling [17, 18] are still restricted to single-block grids.

In this chapter, we propose *Pencil*, a novel pipelined algorithm for distributed stencil computation that overcomes the aforementioned limitations. Section 4.1 makes an in-depth comparison of MPI-everywhere and OpenMP on a single-node, which leads to the optimal combination of MPI and OpenMP for temporal tiling. Section 4.2 demonstrates how Pencil extends the state-of-the-art tiling methods for single-block grids to multi-block grids. Section 4.3 describes the pipelined algorithm in Pencil that effectively overlaps communication with computation while retaining the benefit from temporal tiling. Section 4.4 presents the evaluation of Pencil on two clusters and demonstrates a significant speedup over the state-of-the-art as well as excellent weak and strong scaling. Section 4.5 summarizes the related works. Finally, Section 4.6 concludes our contributions and points out future directions for distributed stencil computation.

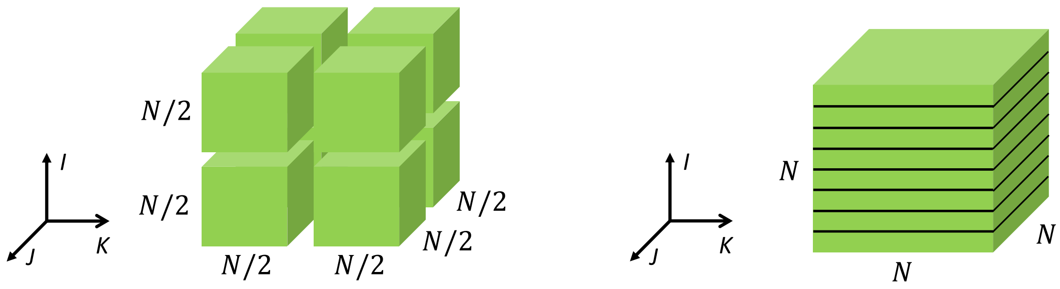
4.1 MPI vs OpenMP On a Single-Node

We have already introduced the two major parallel programming models, namely MPI-everywhere and MPI+OpenMP in Chapter 2. Prior works have focused on comparing MPI+OpenMP against MPI-everywhere [4, 5, 72–75] in distributed computing and we refer readers to [26] for a comprehensive summary. We have no intention to make another comparison except to highlight that most studies overlook the single-node case.

On a single-node, the stencil computation using MPI-everywhere follows the same workflow as the distributed Algorithm 2.1, where intra-node communication is necessary for halo exchange. This need vanishes with threads since all threads have shared access to the data on the node. This leads to the common intuition that threads should outperform flat MPI on a single-node.

It is, however, non-trivial to realize this intuition in practice due to memory arrangement and synchronization as we show in Section 4.4.2. With MPI-everywhere, each process allocates its own data. A process looping over the space following $i \rightarrow j \rightarrow k$ with i being the least rapidly changing dimension and k the most rapidly changing dimension naturally accesses contiguous data in memory, which is preferred for both prefetching and vectorization. With threads, the data is allocated as a single array and each thread gets its own share of the loop $i \rightarrow j \rightarrow k$. To emulate the streaming memory access of MPI, only the i dimension or collapsed dimensions (for instance, with OpenMP `collapse` clause) is distributed among threads. Figure 4.1 shows an example with a domain of size N^3 distributed among 8 cores on a single-node. With MPI, the domain is decomposed into 8 sub-cubes of size $(N/2)^3$ to minimize the intra-node communication. Each process handles one sub-cube and allocates it as an array in DRAM. With OpenMP, the entire cube is allocated as a contiguous data chunk and decomposed into 8 slices along the i^{th} dimension. Each thread accesses one slice and has a $j - k$ plane that is 4 times that of a process. In stencil computations, the update of each $j - k$ plane benefits from several previously accessed planes remaining in cache to increase locality. As a result, the large $j - k$ plane in threads makes it easier to spill out of cache compared to flat MPI.

Moreover, threads and processes have different synchronization patterns. Using OpenMP requires a global barrier at the end of each computation iteration to avoid data race. Flat



(a) MPI, $2 \times 2 \times 2$ (b) OpenMP $8 \times 1 \times 1$

Figure 4.1: An example of optimal domain decompositions on a single-node for MPI and OpenMP respectively. The domain is a cube of size N^3 and is divided evenly among 8 processes/threads. The products in sub-titles denote the number of processes/threads in each dimension. Each green cube/sub-cube is allocated as an array in DRAM.

MPI has no such safety guards and processes only wait for their data-dependent neighbors in communication completion routines like `MPI_Waitall`. In Algorithm 2.2, the computation fetches much more data than packing and exchanging halos. An OpenMP barrier forces all the threads to compute synchronously and therefore competes for memory bandwidth. On the other hand, in MPI, some processes can get a larger share of the available bandwidth at any time while other processes are packing or exchanging halos. As a result, computation in OpenMP suffers a severe memory congestion compared to MPI. On modern architectures, due to the disproportionate increase in the number of cores compared to other on-node resources, memory bandwidth is typically saturated with only a fraction of the cores. In our experiments on 18-cores of Intel Broadwell and 20-cores of Intel Gold processors, we observe that the socket's bandwidth is saturated by just 4 and 10 cores respectively. In Section 4.4.2, we demonstrate the impact of memory allocation and bandwidth competition on MPI and OpenMP's performance on a single-node which suggests the optimal model selection.

4.2 Cache Tiling

The section describes how we extend the start-of-the-art temporal tiling algorithms for single-block grids to multi-block grids. Section 4.2.1 describes two state-of-the-art temporal tiling algorithms namely, the Overlapped Tiling (OT) and Trapezoidal Tiling (TT). Section 4.2.2 demonstrates how to extent the tilings to multi-block grids using DeepHalo.

4.2.1 Temporal Tiling

In 3D stencil computations, updating a $j-k$ plane at i depends on data from previously visited $i-1,\ldots,i-r_s$ planes, where r_s denotes the stencil radius. If the $j-k$ plane's area is large, the data can spill the L3 cache. In that case, blocking the computation of a small range $j0-j1$ in the j^{th} dimension can help retain the data of space $[i-r_s,i]\times[j0,j1]\times[0,N_k]$ in L3 cache. This is referred to as *space tiling*. Here we denote $[j0,j1]\times[0,N_k]$ as a *patch* of the $j-k$ plane. A patch always spans the entire k^{th} dimension for efficient prefetching and vectorization. Although space tiling improves locality, all the data are still read from and written to DRAM at every iteration. Therefore, tiling in space alone is not sufficient for highly memory-bound stencils. To further improve data reuse, several studies have considered tiling in both space and time, which is commonly referred to as *temporal tiling*.

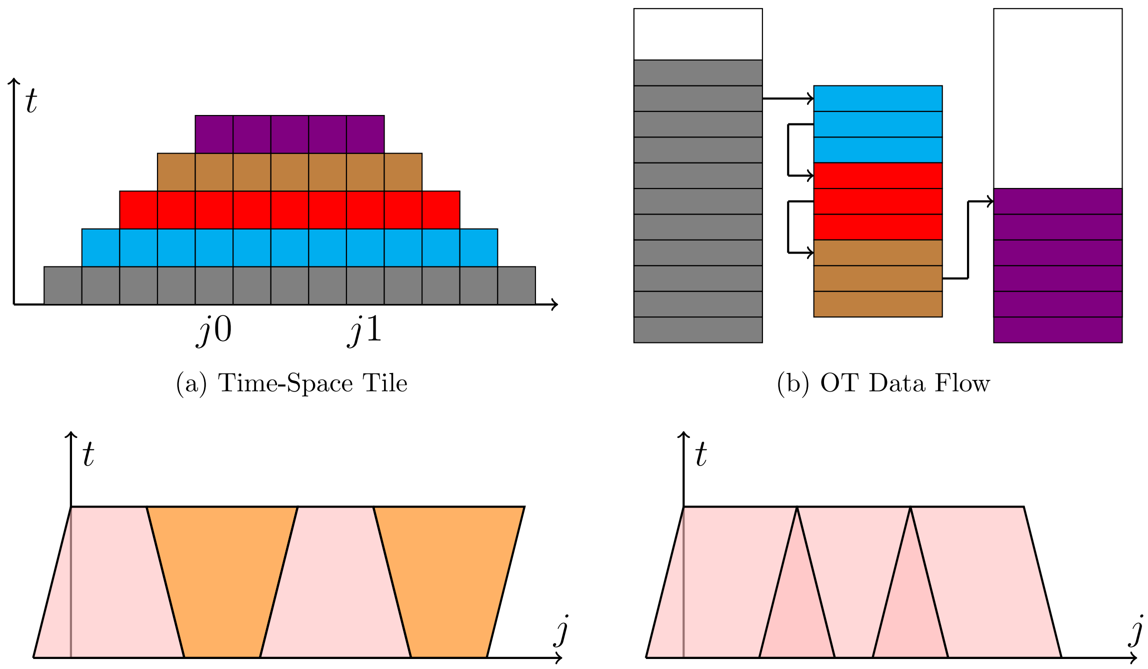
The idea of temporal tiling is to fuse multiple iterations of a patch while it still resides in cache. An efficient way to implement temporal tiling for stencils is to march along the i^{th} dimension and repeatedly use the most recently updated $j-k$ patch to update the patch beneath it. This is referred to as *wavefront blocking* [47, 51, 76]. The code snippet in Figure 4.2 shows an example of fusing f iterations on a patch $[j0,j1]\times[0,N_k]$ with a stencil of unit radius ($r_s=1$). For simplicity, we omit the inputs, outputs, and boundary conditions.

```
1 for (int t=1; t<nIter; t+=f)
2   for (int i=0; i<N_i; ++i)
3     for (int tt=0; tt<f; ++tt) {
4       int p = f-1-tt;
5       for (int j=j0-p; j<=j1+p; ++j)
6         for (int k=0; k<N_k; ++k)
7           compute(t+tt, i-tt, j, k)
8     }
```

Figure 4.2: An example of wavefront blocking

As shown by lines 3-7, immediately after the i^{th} patch is computed at iteration t, it is used to advance the $(i-1)^{th}$ patch at the next iteration, $t+1$. This procedure is repeated until the $(i-f)^{th}$ patch of $[j0, j1] \times [0, N_k]$ is updated at iteration $t+f$. To fuse the f iterations, we need to start with a wider patch extended by $f-1$ cells on both sides of the j^{th} dimension and drop one cell on each side per iteration (line 4). This is because the update of a cell uses one cell from the current iteration on both sides (assuming $r_s = 1$). Figure 4.3a illustrates this effect in time and space dimensions with four iterations fused in the order of ◼ → 🟦 → 🟥 → 🟫 → 🟪, where each color stands for one time step. We denote the trapezoid formed by the patch and the fused iterations as a *time-space tile*. With temporal tiling, the entire tile can be executed in L3 cache, significantly improving the performance.

The cells outside the tile's own patch, i.e., cells outside the range $[j0, j1]$ in Figure 4.3a, overlap with cells in the neighboring tiles. This introduces data dependencies between tiles which prohibits parallel execution. There are two ways to break this dependency. The first is to let each tile update a copy of the dependent data in other tiles. This leads to the idea of *overlapped tiling* (OT) [45–48], where the tiles overlap each other as shown in Figure 4.3d. This removes the dependencies at the cost of redundant computation in the overlapped area. The second is to maximize the number of concurrent tiles by designing special polyhedral shapes and arrangements of tiles. *Trapezoidal tiling* (TT) [49, 50], diamond tiling [51–54], and tessellating tiling [55, 56] follow this strategy.



(a) Time-Space Tile (b) OT Data Flow

(c) TT Time-Space (d) OT Time-Space

Figure 4.3: Data flow and time-space tiles for Overlapped Tiling (OT) and Trapezoidal Tiling (TT).

Figure 4.3b illustrates the dataflow of OT. With OT, each thread has a local array to store the intermediate results of the fused iterations. The left, middle, and right stacks in Figure 4.3b represent the input, local, and output arrays respectively. Each row in the arrays represents a $j-k$ patch and rows of the same color have been updated with the same number of iterations. Given $r_s = 1$, it takes three patches to update a patch. The second patch in the input array is ready to be updated and it is stored in the local array (as shown by ⬛ → 🟦). Then, the iterations 🟦 → 🟥 → 🟫 are performed entirely in the local array and three patches are stored in each iteration. The final result 🟪 is written to the output array. If the local array fits in the cache, then ideally each patch in the input and output arrays is only transferred from/to DRAM once during the fused iterations, significantly reducing the DRAM traffic. Our implementation of OT only synchronizes the threads after the domain has been updated for f iterations, differing from [47, 76, 77] where a barrier or a set of spin-locks are used for each $j-k$ plane. The size of the local array for a tile covering a $j-k$

patch $s_j \times s_k$ is calculated as $(s_j + 2h_o) \cdot (s_k + 2h_o) \cdot (2r_s + 1) \cdot (f - 1)$ where f denotes the number of fused iterations and $h_o = (f - 1) \cdot r_s$ represents the thickness of the overlapped area. Note that the number of $j - k$ patches stored in cache increases linearly with f and is proportional to the size of the stencils. The redundant computation can be estimated by the overall overlapped area as,

$$\begin{aligned} &((s_j + 2h_o)(s_k + 2h_o) - s_j s_k) \cdot \frac{N_j}{s_j} \cdot \frac{N_k}{s_k} \\ &= N_j N_k (2h_o \frac{s_j + s_k}{s_j s_k} + 4\frac{h_o^2}{s_j s_k}) \end{aligned} \tag{4.2}$$

where N_j and N_k are the size of j and k dimensions. Note that the overlapped area increases inversely with the size of the $j - k$ patch $s_j \times s_k$. From the above equations, the size of the local array to fit in L3 cache can be reduced by decreasing the tile sizes but at the expense of increasing redundant computation.

In TT, there are two types of tiles which are colored pink and orange in Figure 4.3c. Tiles of the same type can be executed in parallel. The upward tiles (pink) are identical to the time-space tile in Figure 4.3a. The downward trapeziums (orange) depend on the upward ones. TT sweeps the i^{th} dimension twice, first only updating the upward trapeziums with f iterations and then makes a second pass over the downward trapeziums to update the remaining iteration space. As a result, no redundant computation is introduced. The tiles can be distributed to threads using either static scheduling with synchronization after the execution of tiles of the same type [78] or dynamic tasking based on the tiles' dependencies [59]. In this paper, we follow the latter using OpenMP tasks.

In comparison to OT, TT has the advantage of reducing the required cache quota by shrinking the tile sizes without introducing redundant computation. This is beneficial for large stencils (large r_s) and numerical schemes involving multiple variables. The disadvantage is that intermediate results are written to DRAM and the data updated by both trapeziums is fetched from DRAM twice. As a result, we expect OT to exhibit better performance for heavily memory-bound numerical schemes due to less DRAM traffic while TT might perform better on larger stencils and systems with smaller caches.

4.2.2 Multi-Block Grids with DeepHalo

The previously discussed tiling algorithms and state-of-the-art polyhedral auto parallelization tools and compilers such as Pluto [57, 58] and Pochoir [59] specialize at optimizing perfectly nested loops which correspond to single-block grids in CFD. The iteration space is typically composed of an outer loop for time and 3 nested inner loops for the 3 spatial dimensions, as shown in Figure 1.2. In real applications, the common case is multi-block structured grids, which disrupts the perfectly nested loop structure.

```
 1  for (int t=0; t<tEnd; ++t) {
 2    for (int block=0; block<nBlocks; ++block) {
 3      get_block_range(block, range);
 4      for (int i=range[0]; i<range[3]; ++i)
 5        for (int j=range[1]; j<range[4]; ++j)
 6          for (int k=range[2]; k<range[5]; ++k)
 7            compute(block, i, j, k);
 8    }
 9    for (int block; block<nBlocks; ++block)
10      exchange_block_boundary(block);
11  }
```

Figure 4.4: Pseudocode for multi-block grids

Figure 4.4 demonstrates the nested loops for solving multi-block grids where each block introduces 3 loops for the spatial dimensions at lines 4-6. Blocks can be connected and their halo data is exchanged by the function `exchange_block_boundary` (line 10) at every time iteration. This introduces data dependence between blocks and prevents state-of-the-art polyhedral techniques to directly tile the time and space loop together for each block.

The code snippet in Figure 4.5 illustrates how we resolve the data dependency for multi-block grids. We tile the outer temporal loop by f so that each block has perfectly nested loops composed of f temporal iterations and 3 space dimensions (lines 5-9). To fuse f iterations on a grid cell, we need $f \cdot r_s$ halo cells on both sides in each dimension. So, we attach $f \cdot r_s$ halo layers to the blocks' boundaries and the halos at the connected boundaries are exchanged with `exchange_block_boundary` once every f iterations (lines 11-12). As a result, we have broken down the loop structure for multi-block grids in Figure 4.4 into multiple perfectly nested loops for single-block grids, each of which can be optimized using OT or TT. We refer to the addition of adequate halo layers to fuse f iterations as *DeepHalo*.

```
 1 for (int t=0; t<tEnd; ++t) {
 2   for (int block=0; block<nBlocks; ++block) {
 3     get_block_range(block, range);
 4     // fused iterations
 5     for (int tt=0; tt<f; ++tt)
 6       for (int i=range[0]; i<range[3]; ++i)
 7         for (int j=range[1]; j<range[4]; ++j)
 8           for (int k=range[2]; k<range[5]; ++k)
 9             compute(block, i, j, k);
10   }
11   for (int block; block<nBlocks; ++block)
12     exchange_block_boundary(block);
13 }
```

Figure 4.5: Pseudocode for tiling multi-block grids with DeepHalo

The temporal tiling for multi-block grids can be easily generalized to distributed memory systems by adding inter-node communication to function `exchange_block_boundary`. With DeepHalo, each process has to communicate up to 26 messages for a rectangular block, i.e. 6 for faces, 12 for edges, and 8 for corners regardless of the shape of the stencil. DeepHalo was originally proposed to reduce communication cost by reducing the rounds of communication and performance improvement has been reported in literature [76, 79]. However, as shown in Section 4.4.3, we cannot solely rely on DeepHalo to improve the communication performance because its effect is still network-specific on modern systems.

4.3 Overlap Communication and Computation

In this section, we present our pipelined algorithm to overlap communication with computation. Section 4.3.1 describes how to effectively overlap communication and computation with MPI+OpenMP. Section 4.3.2 introduces the pipelined overlap algorithm in Pencil.

4.3.1 MPI+OpenMP Models for Overlap

The overlap of communication and computation becomes possible in modern architectures that support RDMA where the Network Interface Card (NIC) takes over the communication without the involvement of the CPU. The network software underlying most MPI implementations already make use of this feature. However, as highlighted by several studies [62–65] merely using MPI's non-blocking or RMA routines does not achieve overlap since the MPI standard does not guarantee the communication to make progress outside MPI function calls.

In practice, there are two popular workarounds:

- *DedicatedCore (DC).* Dedicate one core for communication while the other cores perform computation.
- *RepeatedPoll (RP).* Repeatedly call functions such as `MPI_Test` during computation to urge the underlying network software to make progress on communication.

For flat MPI, it's non-trivial to implement DedicatedCore and we refer readers to Casper [62, 66, 80]. In this paper, we only consider MPI+threads model for overlap. The degree of overlap can be estimated using the effective overlap ratio, η defined as follows:

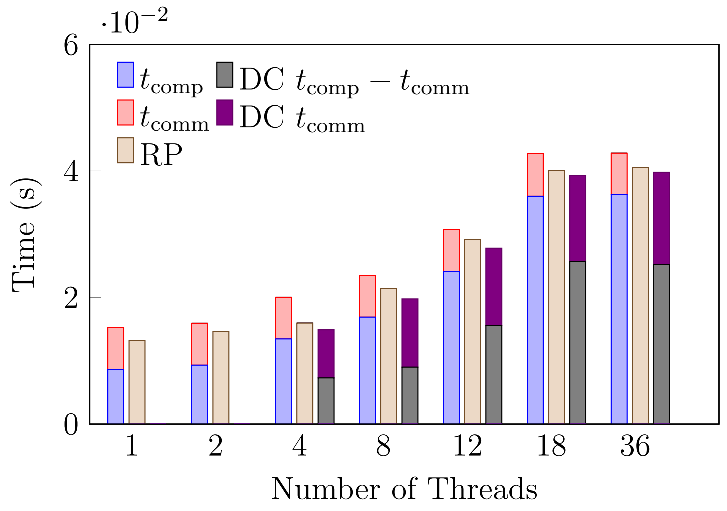
$$\eta = \frac{t_{\text{comp}} + t_{\text{comm}} - t_{\text{ovlp}}}{\min(t_{\text{comp}}, t_{\text{comm}})}, \tag{4.3}$$

where t_{comp}, t_{comm}, and t_{ovlp} are the measured computation time, communication time, and the time for the overlapped computation and communication respectively.

We are interested in understanding how much overlap is achievable in practice by DC and RP for a *memory-bound* computation. For this purpose, we benchmark both methods by exchanging a large message between 2 nodes using `MPI_Isend/Irecv` while the cores are busy with a memory-bound computation $a[i] = w_0 a[i] + w_1 b[i] + w_2 c[i]$. On both nodes, we start from a single thread and keep increasing the number of threads until the entire node is occupied. Each thread is assigned a fixed workload large enough to spill the L3 cache. The overall data volume increases with the number of threads and saturates the DRAM bandwidth. After saturation, the computation time starts to increase proportionally with the number of threads. With RP, `MPI_Test` is called periodically during computation to make progress on communication. With DC, one thread waits at `MPI_Wait` while the other threads split the total workload. So, DC has one less core participating in computation compared to RP where all cores are involved in computation.

We benchmark the overlap on two clusters – Bebop and HPC3 – summarized in Table 4.2 with Omni-Path and InfiniBand networks. The results are presented in Figures 4.6 and 4.7 respectively. The leftmost stacked bar shows the time for communication and computation without overlap. The second bar represents RP and the third shows DC with its communication time (DC t_{comm}) highlighted on top of the non-overlapped computation (DC $t_{\text{comp}} - t_{\text{comm}}$). On both clusters, the non-overlapped computation time t_{comm} remains the same for a small number of cores (2 for Bebop and 4 for HPC3). This is because the bandwidth has not yet been saturated and each core has the same computation workload. As we add more threads into the benchmark, the bandwidth becomes saturated and t_{comm} start to increase. Therefore, we can tell from t_{comm} that 4 and 8 cores can saturate a socket's bandwidth for Bebop and HPC3 respectively.

On Bebop, RP can achieve up to 40% of the ideal overlap. DC performs slightly better



(a) Performance of RepeatedPoll and DedicatedCore on Bebop

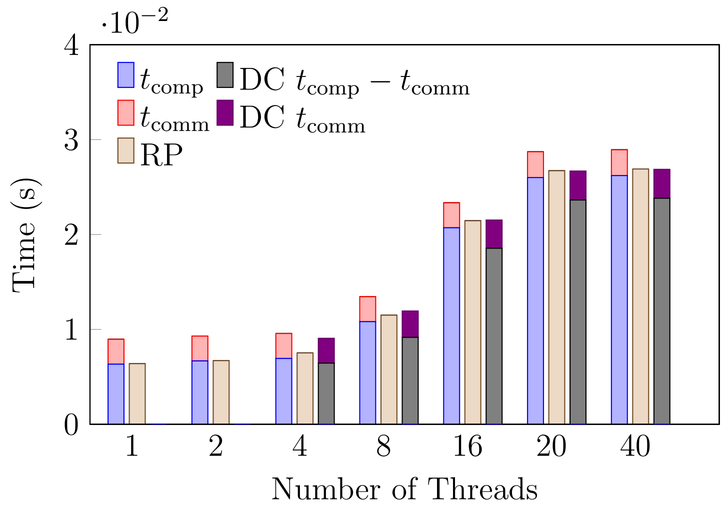
# Cores	1	2	4	8	12	18	36
η_{RP}	0.31	0.19	0.29	0.31	0.24	0.39	0.35
η_{DC}	-	-	0.42	0.56	0.44	0.51	0.48

(b) Overlap ratio on Bebop

Figure 4.6: Computation and communication overlap with DedicatedCore and RepeatedPoll on Bebop.

but still only attains half of its potential. As long as the communication time is shorter than computation, it is completely hidden. The lack of efficiency comes from using one less core for computation. Furthermore, the actual communication in DC increases as the bandwidth is gradually saturated. When the socket is fully occupied with 18 threads, the communication time increases by 2.2× over the non-overlapped case. This is because, in some PCI express (PCIe) connections, the messages written by the NIC and the DRAM I/O issued by CPUs all go through the path between Root Complex (RC) and DRAM. Though the NIC can issue stores without CPU's involvement, the actual data transfer still competes for bandwidth with CPUs, especially in memory-bound applications.

On HPC3 with InfiniBand, RP's performance starts to drop when the bandwidth becomes saturated. However, it still attains over 70% of the potential benefit from the overlap. As the number of threads increases, the downside of using one less core for computation



(a) Performance of RepeatedPoll and DedicatedCore on HPC3

# Cores	1	2	4	8	16	20	40
η_{RP}	0.98	0.99	0.78	0.74	0.72	0.74	0.75
η_{DC}	-	-	0.19	0.57	0.69	0.75	0.76

(b) Overlap ratio on HPC3

Figure 4.7: Computation and communication overlap with DedicatedCore and RepeatedPoll on HPC3.

gradually vanishes and DC achieves similar performance and overlap as RP. Moreover, the actual communication time of DC only increases 16% over non-overlapped communication. Contrasting the results from the two clusters, we conclude that the effectiveness of the overlap highly depends on the software and hardware of the network in addition to the application characteristics. Though NIC can fully support RDMA, the saturation of bandwidth by applications running on CPUs can still affect the communication performance.

4.3.2 Pipeline Communication and Computation

To realize overlap irrespective of which method (DC or RP) is used, the computation must not have data dependence on the overlapped communication. In the CFD scenario, this means we need to divide the computational domain into a halo-dependent part and a halo-independent part. The computation of the latter can be overlapped with the exchange of halos. The classic decomposition is to divide the domain into an outer layer and an inner chunk whose update does not rely on the halo region. It is, however, challenging to compute the outer layer efficiently in parallel. In order to divide the computation evenly among threads, one must take into account the difference between the length of the contiguous data segment in the i, j, and k boundaries. Such division is highly non-trivial [26].

An alternate approach is to categorize the cache tiles based on their dependence on the halo region and overlap the communication with the halo-independent tiles. For OT and TT, the domain can be split only in the j^{th} dimension. Therefore, one can partition the grid block in the j^{th} dimension so only the tiles touching the j boundary depends on halos. The computation of the remaining tiles can then overlap with communication. This idea has been exploited to improve performance with diamond tiling in [17]. However, for multi-block grids, imposing a 1D partition is not feasible since blocks can be connected in *any* dimension.

We propose a *pipelined algorithm* to break the data dependence and achieve overlap. The idea is to cut the domain into chunks along the i^{th} dimension. This way, each chunk's computation has no dependence on the previously updated chunks' halo layers and can be overlapped with the communication of the previous chunk. Figure 4.8 illustrates this idea using two processors, P_0 and P_1 whose domains are cut into multiple chunks. Each chunk has a rectangular shape and is suitable for temporal tiling with OT or TT. At stage l, chunk C_l is being updated (marked as cyan) while chunk C_{l-1} has already been updated (gray). So, the communication of C_{l-1}'s halo (pink) is overlapped with C_l's computation. Similarly, in stage $l+1$, C_{l+1}'s update overlaps with C_l's communication and so on. Together with DeepHalo, we can achieve overlap on multiple connected iteration spaces without losing the performance gain from temporal tiling. Furthermore, Pencil does not impose any limitation on the global decomposition, i.e. communication can happen in any dimension.

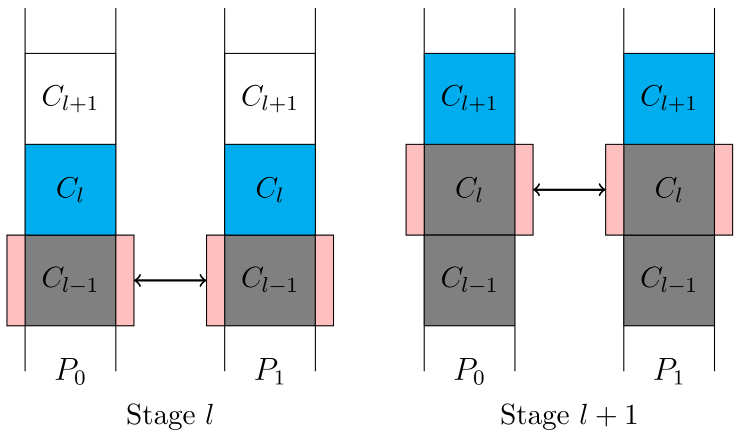


Figure 4.8: Cut domain into chunks and pipeline communication and computation for overlap.

4.4 Experiments and Results

In this section, we evaluate Pencil against the state-of-the-art tiling algorithms on both shared-memory and distributed-memory systems.

4.4.1 Experiment Setup

In this section, we describe the case studies and platforms used for evaluating the single-node and distributed-memory performance of Pencil.

Case study To systematically evaluate our proposed algorithms, we choose 4 stencils across 6 numerical schemes whose characteristics are summarized in Table 4.1. The stencils have different shapes (illustrated in Figure 2.2) and radius. The numerical schemes have various numbers of input/output variables and span a wide range of arithmetic intensity (AI) from 0.42 - 2.50. Here we calculate AI with and without non-temporal (NT) stores, which if supported by the compiler can bypass write-allocate and improve performance. Below, we outline the numerical schemes that give rise to the stencil test cases.

Table 4.1: Stencils and numerical schemes. Each column from left to right represents the test name, the number of points in stencil, the shape and radius of stencil r_s, the numerical scheme, the arithmetic intensity (AI) with and without non-temporary store (NT), and the number of input and output variables respectively. For the schemes, WJ denotes Weighted Jacobi. WENO 3rd, Upwind 2nd and CD 2nd stand for 3rd order WENO, 2nd order Upwind, and 2nd order Central Difference respectively.

Test	#Points	Shape	r_s	Scheme	AI (NT)	AI	#In	#Out
WJ7	7	Star	1	WJ 7pt	0.42	0.31	2	1
WJ13	13	Star	2	WJ 13pt	0.67	0.5	2	1
WJ27	27	Box	1	WJ 27pt	1.25	0.94	2	1
WENO3	13	Star	2	WENO 3rd	1.96	1.64	4	1
Upwind	13	Star	2	Upwind 2nd	0.85	0.71	4	1
Burgers	24	Staggered	1	CD 2nd	2.50	1.67	3	3

- **Weighted Jacobi for 3D Poisson equation.** The Poisson Equation 4.4 is typically used to solve for the pressure p in incompressible flows with source function b derived from the velocity field.

$$\nabla^2 p = b \tag{4.4}$$

  Various stencils can be used depending on the order of accuracy. Here we consider the star stencils with radius 1 and 2, consisting of 7 and 13 points respectively in 3D, and the box stencil with 27 points. Equation 4.4 is solved with the weighted Jacobi methods, which is one of the standard smoothers for multigrid [81]. Jacobi methods are the most widely studied stencil computations in the space and temporal tiling body of research [46, 47, 49–59, 82] for its simplicity, where the updated value is essentially a weighted average of the stencil cells' values. However, practical numerical schemes in CFD can be considerably complex as in the following cases.

- **Upwind and WENO schemes for 3D advection equation.** Equation 4.5 simulates the convection phenomena in fluid dynamics, in which ψ denotes a scalar propagated by the velocity field $\vec{u}$.

$$\partial_t \psi + \vec{u} \cdot \nabla \psi = 0 \tag{4.5}$$

Here we consider the memory-bound 2nd order upwind and the 3rd order WENO schemes (which have higher flops per grid cell) [83] on a star stencil of radius 2. Both schemes use different cells in the stencil depending on the sign of the velocity. Such computation is typically implemented with a ternary operation, for instance in 1D,

$$\phi_i \mathrel{-}= u_i > 0 \text{ ? } f(u_{i-1}, u_i) : f(u_i, u_{i+1})$$

where $f(u_{i-1}, u_i)$ can be a simple expression (Upwind) or a complex inlined function (WENO3).

- **3D Burgers equation.** Equation 4.6 represents the conservative form of Burgers equation, which is solved in simulating incompressible flows ($\nabla \cdot \vec{u} = 0$).

$$\partial_t \vec{u} + \nabla \cdot (\vec{u}\vec{u}) = \nu \nabla^2 \vec{u} \tag{4.6}$$

The three components of velocity $\vec{u}$ (the 3 inputs) are discretized on grid cell's face centers using a staggered stencil. Figure 4.9 illustrates the staggered stencil in 2D for velocity $\vec{u}(u, v)$, where each square represents a grid cell with velocity components marked by arrows at the face centers. To update the v component of velocity (brown arrow), not only are the surrounding v components (blue arrows) required but also the adjacent u components (green arrows) of velocity. Similar dependencies apply in 3D. Moreover, all the three components of velocity are both read from and written to DRAM during the update. As we show in Section 4.4.2, the coupled dependency between components and the large data volume is highly challenging for the tiling algorithms.

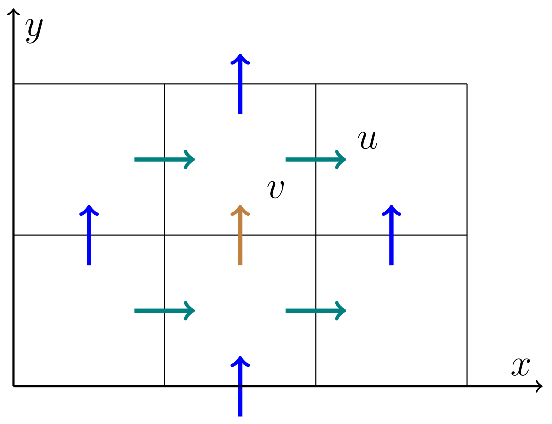


Figure 4.9: An example of 2D staggered grid. The velocity components are located at grid cells' faces. The green and blue arrows mark the u and v components respectively. The brown arrow marks the v component to be updated in this example, which depends on all the velocities highlighted.

Platforms and Architectures We evaluate the performance of the stencils on two distributed memory machines – Bebop equipped with 653 Intel Xeon E5-2695v4 (Broadwell) nodes at the Argonne National Laboratory and HPC3 with 38 Intel Xeon 6248 (Gold) nodes at the University of California Irvine. The key parameters of these systems appear in Table 4.2.

Table 4.2: Evaluation platforms and their parameters.

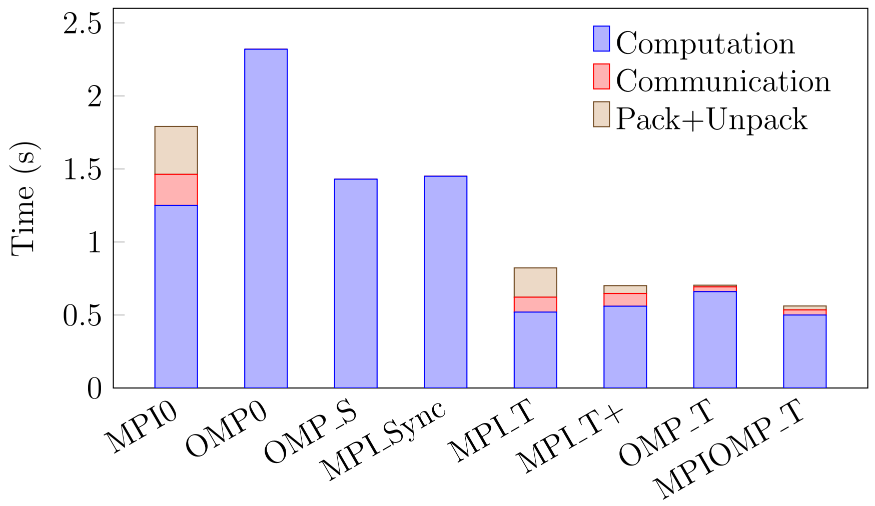
	Bebop	HPC3
Architecture	Intel Xeon E5-2695v4 (Broadwell)	Intel Xeon 6248 (Gold)
CPU Frequency	2.4 GHz	2.5 GHz
Sockets	2	2
Cores/Socket	18	20
GFlops/s (DP)	1200	2207
L2 cache	32 KB	1024 KB
L3 cache	90 MB	55 MB
DRAM Bandwidth	120.3 GB/s	194.4 GB/s
Network	Omni-Path	InfiniBand
Nodes	653	38
Compiler	Intel 2017	GCC 8.4.0

We choose a block size of 480^3 per node on both systems for the experiments which is large enough for cache tiling to be effective but at the same time, not too large to overshadow the communication cost. In our experiments, fusing more than 10 halos results in a performance drop. Therefore, in the results presented in the following section, we limit the number of fused iterations to not exceed 10 layers of halo (i.e. $f \cdot r_s \leq 10$) and the total number of iterations to 60 which is large enough to maintain a steady solve time per iteration.

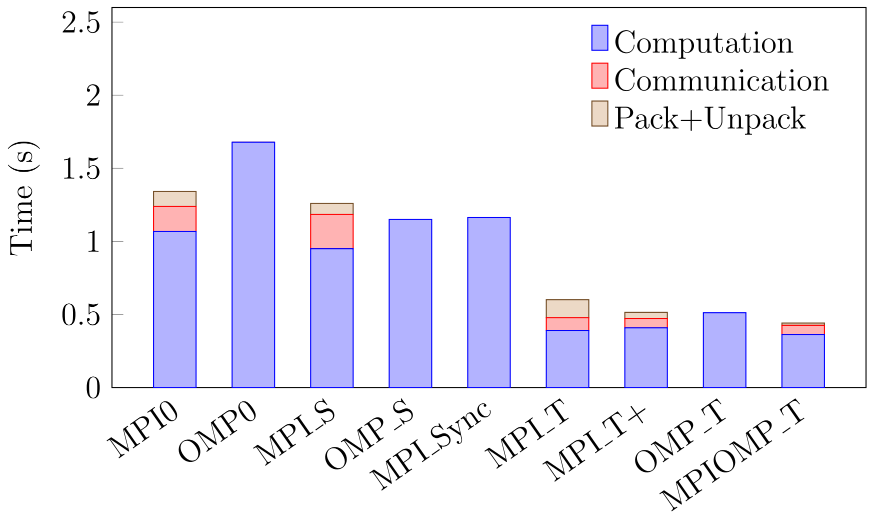
4.4.2 Single-Node Performance

In this section, we first present a step-by-step analysis of the the single-node performance breakdown with spatial and temporal tilings and identify the optimal decomposition of MPI processes and OpenMP threads for temporal tiling, which will be referred to as *Hybrid Tiling* for the remainder of this chapter. Next, we compare the performance of hybrid tiling against the state-of-the-art polyhedral tiling tool, Pluto [57, 58]. Moreover, we compare the two temporal tiling algorithms – overlapped and trapezoidal tiling and present an analysis of scenarios where one outperforms the other.

Optimal Decomposition of MPI and OpenMP We demonstrate how to choose the optimum combination of MPI processes and OpenMP threads step by step using the WJ7 stencil on Broadwell as an example. The MPI baseline (*MPI0*) decomposes each dimension of the block evenly to reduce intra-node communication volume and achieve load balance. In our experiments, a decomposition of $4 \times 3 \times 3$ in the i^{th}, j^{th}, k^{th} dimensions delivers the best baseline performance. Unlike MPI0, the OpenMP baseline (*OMP0*), splits the domain only in the i^{th} dimension to ensure threads access contiguous data as discussed in Section 4.1. As seen from Figure 4.10a, even though communication and buffer preparation (i.e., packing and unpacking) take a considerable time, MPI0 still outperforms OMP0 by 30% on Broadwell. This is because OMP0 has a $j - k$ plane that is $9\times$ larger than MPI0 and spills out of the cache. To remedy this, we add spatial tiling (denoted by *OMP_S*) in the j^{th} dimension, which has no effect on MPI0 on Broadwell (therefore not shown in Figure 4.10a) but improves the OpenMP baseline by 1.6×.



(a) WJ7 on Broadwell



(b) WJ7 on Gold

Figure 4.10: Performance of spatial and temporal tiling with WJ7 on a single-node.

Note that the computation in MPI0 still takes less time than OMP_S. This is because of the competition for bandwidth (described in Section 4.1). Figure 4.11 presents the trace for MPI0, where the blue color marks the time spent on computation and the white background indicate the processor is busy with the intra-node communication or buffer preparation.

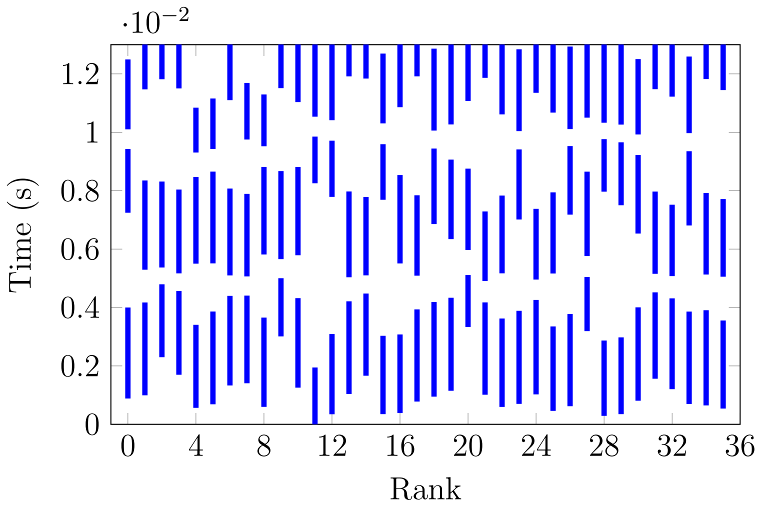


Figure 4.11: Trace for WJ7 on a single Broadwell node. The blue color marks a rank's time spent on computation. The white background denotes the time spent on intra-node communication and buffer preparation.

Since MPI processes are only loosely synchronized by routines like `MPI_Wait`, the computations of different ranks can be asynchronous, i.e., unaligned in Figure 4.11. When a rank's computation overlaps with other ranks' communication or buffer preparation, the computation gets a larger share of bandwidth for its larger DRAM traffic. With OpenMP, all threads are synchronized by barriers and compete for the bandwidth simultaneously. Therefore, the computation in MPI0 appears faster than OMP_S. This explanation can be validated by enclosing the computation in MPI0 with `MPI_Barrier`s. This implementation denoted by *MPI_Sync* leads to similar computation performance as OMP_S in Figure 4.10a.

Moreover, with spatial tiling, the measured arithmetic intensity (AI) in Table 4.3 matches our theoretical estimate in Table 4.1 with non-temporal stores except for WENO3 and Burgers. In the case of Burgers, the compiler fails to generate NT stores for the large loop writing three variables. On the other hand, the flops of WENO3 depend on compiler optimizations of ternary operators (Section 4.4.1) and it is challenging to match the theoretical estimates which further underscores the complexity of these two stencil case studies.

On Gold, we observe a similar behavior. OMP_S outperforms MPI_S and the measured AI matches the estimate without NT stores except for Weno3. The GCC compiler does not generate NT instructions and results in write-allocate when writing to DRAM, which lowers the theoretical AI. Nonetheless, GCC executes all the possible paths in ternary operator and masks the values not used, which leads to higher AI for Weno3. Our experiments thus far establish that contrary to popular wisdom, OpenMP outperforms MPI on a single-node *only if* spatial tiling is applied.

Now we add temporal cache tiling to both flat MPI and OpenMP. It is best to have a small $j - k$ area so that more planes can reside in cache while performing the wavefront blocking in the i^{th} dimension. For MPI with temporal tiling, the optimal decomposition turns out to be 1 x 6 x 6 for the i^{th}, j^{th} and k^{th} dimensions, which also minimizes the communication volume. Each MPI rank applies either overlapped tiling (OT) or trapezoidal tiling (TT) on its local domain. In Figure 4.10a, *MPI_T* presents the best performance obtained when combining MPI with either OT or TT. Although MPI_T achieves 2× speedup over OMP_S, we observe that the packing and unpacking of halo can take longer than the intra-node communication time. The cost of copying the halos' data to/from the buffer is determined by the data's layout in DRAM. Since the block's data is contiguous along the k^{th} dimension, the halos attached to i and j boundaries are organized into contiguous covering the block's k^{th} dimension, whereas the halos on k boundaries are composed of short segments whose length equals the thickness of the halos. Therefore, packing and unpacking the k halo involves copying a large number of short segments to/from a 1D buffer, which is highly inefficient and expensive. We work around this issue by merging the packing and unpacking of the k boundaries into computation as shown in Figure 4.12. Now, the k halo's data is used immediately after it is loaded into cache and written to the buffer while still in cache. This optimization denoted by *MPI_T+* further improves MPI_T by 17% on Broadwell.

For temporal tiling implemented with OpenMP, we let OT and TT decompose the iteration space in the j^{th} dimension and leave the k^{th} dimension unsplit for prefetching and SIMD. In

```
for (int i=iBegin; i<iEnd; ++i)
  for (int j=jBegin; j<jEnd; ++j) {
    unpack_k_halo(i, j);
    for (int k=kBegin; k<kEnd; ++k)
      // computation
    pack_k_halo(i,j);
  }
```

Figure 4.12: Merge packing and unpacking k boundaries into computation

Figure 4.10a, *OMP_T* represents the best performance obtained with OpenMP and temporal tiling. Despite the similar overall performance between MPI_T+ and OMP_T, MPI_T+ still computes 18% faster than OMP_T. The reason lies in the shape of the $j-k$ plane. Each process in MPI_T+ has a $j-k$ patch of size 80×80 and threads in OMP_T have a patch of size 13×480 (or 14×480). In OT, if 6 iterations are fused, then the $j-k$ patch including the overlapped area in OMP_T (25×492) becomes 1.5x larger than MPI_T+ (92×92). Therefore, OMP_T fits fewer $j-k$ planes in L3 cache compared to MPI_T+. A similar analysis also applies to TT.

To combine the advantages of MPI and OpenMP, we first decompose the block in the k^{th} dimension among MPI ranks and then perform temporal tiling within each rank using OpenMP threads. The cache tiles split the j^{th} dimension and perform wavefront blocking in the i^{th} dimension. This combines MPI's advantage of a small $j-k$ area and only introduces limited intra-node communications by using OpenMP to compute cache tiles. In our experiments, using 2 MPI ranks with 1 rank per socket is sufficient to emulate the computation performance of MPI_T with minimal intra-node communication. The hybrid algorithm denoted as *MPIOMP_T* achieves the best performance not only for WJ7 in Figure 4.10a but across all the stencils. Across the broad, MPIOMP_T achieves 15% - 34% speedup over the best case between MPI_T+ and OMP_T on Broadwell and up to 35% on Gold.

Comparison with the state-of-the-art: Pluto. We now compare our hybrid algorithm with the polyhedral tiling tool Pluto [57, 58], which generates codes using diamond tiling. Following the guidelines in [54], we set a large tile size for the k^{th} dimension and perform an exhaustive search for the optimal tile sizes of the other dimensions. Table 4.3 summarizes the performance comparison across the different stencil case studies. On Broadwell, our hybrid algorithm achieves similar performance compared to Pluto for weighted Jacobi schemes WJ7 and WJ13 but suffers a 9% slow-down for WJ27. The sub-optimal behavior for WJ27 needs further investigation. However, we observe a significant speedup of $1.92\times$ and $1.49\times$ for more complex schemes such as Upwind and WENO3 respectively. As for Burgers, its coupled dependencies between the three velocity components result in a huge linear programming system with $10^3 \sim 10^4$ constraints. Pluto fails to generate diamond tiles in this case and downgrades to a tiling that requires a pipelined start, leading to performance even below the baseline. On Gold, our hybrid algorithm outperforms Pluto by $1.08 - 1.74\times$ on all tests except for Burgers where Pluto fails like on Broadwell.

Note that the temporal tiling algorithm must be evaluated on top of spatial tiling rather than the naive baseline. If a significant speedup is not observed with temporal tiling, then spatial tiling is preferred for fewer code modifications. As shown in Table 4.3, Pluto's temporal tiling for Upwind and WENO3 are not effective since their performance are emulated by spatial tiling alone on both Broadwell and Gold processors. The hybrid algorithm on the other hand outperforms space tiling by $1.47-2.83\times$ on Broadwell and $1.09-3.29\times$ on Gold. Overall, it achieves $1.47 - 4.76\times$ speedup over the baseline with OpenMP.

Overlapped Tiling vs Trapezoidal Tiling Figure 4.13 compares the two temporal tiling algorithms (OT and TT) on WJ13 and Upwind schemes using the metric billion cells updated per second (GCells). Both schemes use the same star stencil with $r_s = 2$ but exhibit different performance behavior with OT and TT. Each curve shows tiling with a fixed tile size. For

Table 4.3: Summary of single-node performance of the stencils in Table 4.1 on the two systems in Table 4.2. AI denotes the arithmetic intensity; MPIOMP_T lists the temporal tiling algorithm (first entry) and number of fused iterations (second entry) that achieves the best performance with our hybrid algorithm; The three columns to its right present the speedup of our hybrid algorithm with temporal tiling over the OpenMP baseline, OpenMP with spatial tiling, and Pluto respectively.

(a) Performance Results for a Single Broadwell Node

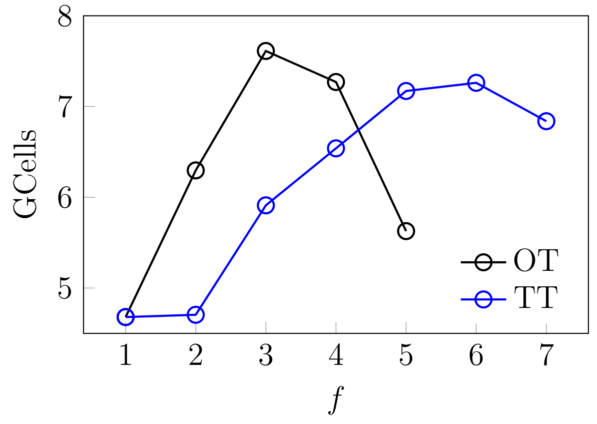
Test	AI(NT)	MPIOPT_T	vs OMP0	vs OMP_S	vs Pluto
WJ7	0.42	TT 10; OT 6	4.63×	2.83×	0.98×
WJ13	0.69	OT 3; TT 6	3.46×	1.63×	0.98×
WJ27	1.28	TT 8	2.68×	1.69×	0.91×
Upwind	0.87	TT 5	2.86×	1.67×	1.92×
WENO3	1.71	TT 3	2.06×	1.47×	1.49×
Burgers	1.63	OT 2	2.42×	1.58×	4.63×

(b) Performance Results for a Single Gold Node

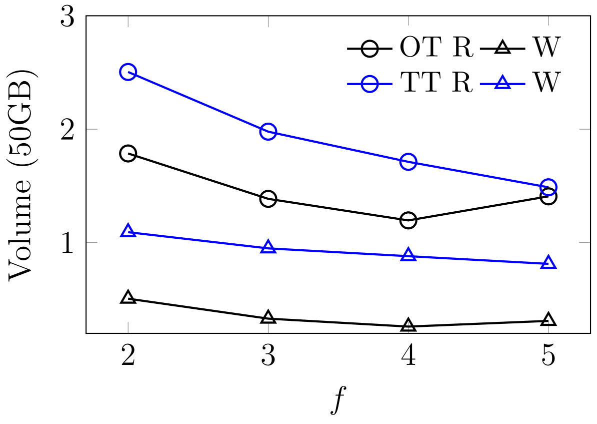
Test	AI	MPIOMP_T	vs OMP0	vs OMP_S	vs Pluto
WJ7	0.29	TT 10; OT 4	4.76×	3.29×	1.15×
WJ13	0.48	TT 5;	3.52×	1.89×	1.29×
WJ27	0.94	TT 5	2.06×	1.46×	1.10×
Upwind	0.71	TT 3	2.72×	1.57×	1.74×
WENO3	2.40	TT 2	1.59×	1.09×	1.08×
Burgers	1.52	OT 2; TT4	1.87×	1.25×	5.03×

OT, we limit one tile per thread to reduce redundant computation. For TT, we find $2\,nt-1$ is an optimal number of tiles in our experiments where nt denotes the number of threads. This generates enough independent tiles for each thread to start computation concurrently but not too many to increase DRAM traffic discussed in Section 4.2.

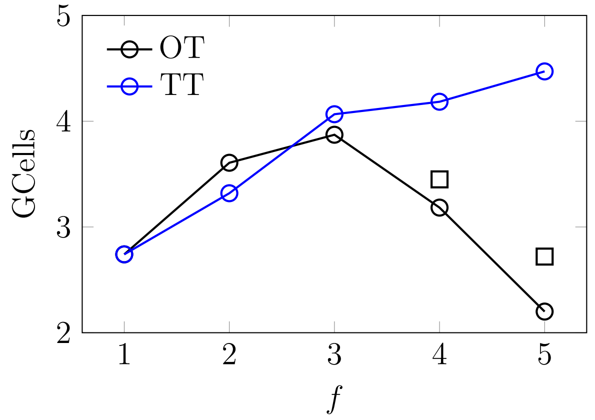
Column *MPIOMP_T* in Table 4.3 lists the parameter configuration that results in the best performance with hybrid MPI+OpenMP. If OT and TT achieve performance within an 8% difference, both configurations are listed with the first being the faster one. On Broadwell, for stencils with similar performance, OT reaches its peak with a smaller number of fused iterations, f, as seen from Figure 4.13a for the WJ13 stencil. This is because, for small f, OT's local array still fits in the L3 cache and results in less DRAM traffic than TT. As



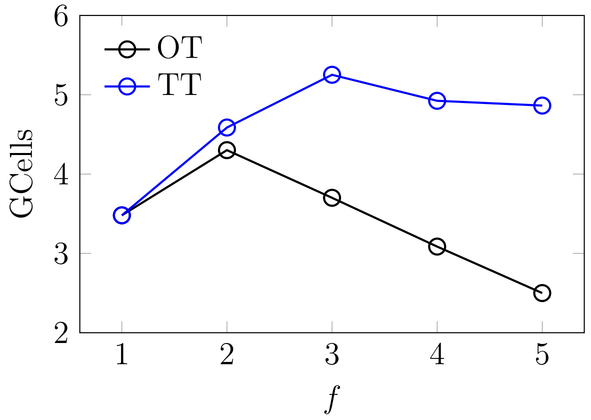
(a) WJ13 on Broadwell



(b) Upwind on Broadwell



(c) Upwind on Broadwell



(d) WJ13 on Gold

Figure 4.13: Comparison of Overlapped and Trapezoidal tiling with increasing number of fused iterations (f).

shown in Figure 4.13b, at $f = 3$, OT's reading and writing volumes is only 34% and 70% of TT. As f increases, OT introduces too much redundant computation and the local array starts to spill from the L3 cache, which causes the performance to drop drastically. TT has the advantage of using small tiles without introducing redundant computation (Section 4.2) which allows us to start with twice as many tiles as OT. At any time, each thread executes a tile using half the cache as overlapped tiles in OT. Therefore TT supports more fused iterations without spilling out of the L3 cache.

Figure 4.13c compares OT and TT on Upwind which has the same stencil radius as WJ13 but loads 3 additional variables from DRAM. Since it is desirable to keep more data in the L3 cache, TT outperforms OT for its lower cache requirement. Using smaller tiles improves OT's performance as shown by the black squares in Figure 4.13c. Nonetheless, it does not reach TT's peak performance because it introduces additional redundant computation.

To summarize, OT can achieve similar or better performance as TT for highly memory-bound stencils such as WJ7 and WJ13 with fewer fused iterations since it results in less DRAM traffic. This can lead to fewer DeepHalo layers in distributed computing which may result in less communication cost. TT outperforms OT for less memory-bound numerical schemes like WJ27 or schemes loading multiple variables, except for Burgers in our experiments. Burgers is a special case because three components of velocity $\vec{u}$ need to be written to DRAM which highlights OT's advantage at reducing writing volume.

The L3 cache in Gold is only about half that of Broadwell which makes it challenging for OT. As expected, TT outruns OT in most cases including highly memory-bound stencils such as WJ13 as shown in Figure 4.13d, and achieves similar performance for Burgers.

4.4.3 Communication with DeepHalo and Pipelined Overlap

We compare Pencil over MPI-everywhere and MPI+OpenMP Funneled on 32 nodes of Bebop and 32 nodes of HPC3. Each node is assigned a block of size 480^3 and each block is connected to 6 other blocks by face. For MPI+OpenMP Funneled, we assign one process per node and bind each thread to a core. For flat MPI, the block is further divided evenly among the processors. We apply spatial tiling to both. For the hybrid temporal tiling, we map one MPI rank per socket with one OpenMP thread per core.

Table 4.4 presents the measured effective overlap ratio, η, and speedups over MPI-everywhere and MPI+OpenMP Funneled for the 6 case studies. We observe that the optimal choice of

Table 4.4: Summary of the performance of six test cases in Table 4.1 on 32 nodes of Bebop and 32 nodes of HPC3 summarized in Table 4.2. MPIOMP_T lists the temporal tiling algorithm (first entry) and the number of iterations fused (second entry) that delivers the best performance with Pencil; η_{RP} and η_{DC} are the measured overlap ratios for RepeatedPoll and DedicatedCore; "vs non-ovlp" is the speedup of the best of RP and DC over the same tiling algorithm without overlap. "vs baseline" is the speedup of the best of RP and DC over the best of MPI-everywhere and MPI+OpenMP Funneled with spatial tiling.

(a) Performance Results for 32 Gold Nodes with InfiniBand Connections

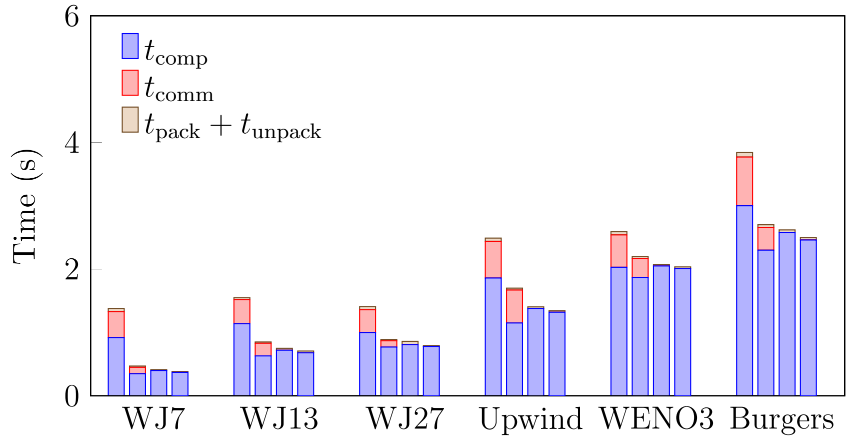
Test	MPIOPT_T	η_{RP}	η_{DC}	vs non-overlap	vs baseline
WJ7	TT 10	0.77	0.50	1.20×	3.36×
WJ13	TT 5	0.75	0.61	1.21×	2.19×
WJ27	TT 6	0.93	0.59	1.11×	1.86×
Upwind	TT 3	0.66	0.55	1.26×	1.84×
WENO3	TT 3	0.50	0.40	1.08×	1.27×
Burgers	OT 2	0.56	0.48	1.08×	1.54×

(b) Performance Results for 32 Broadwell Nodes with Omni-Path Connections

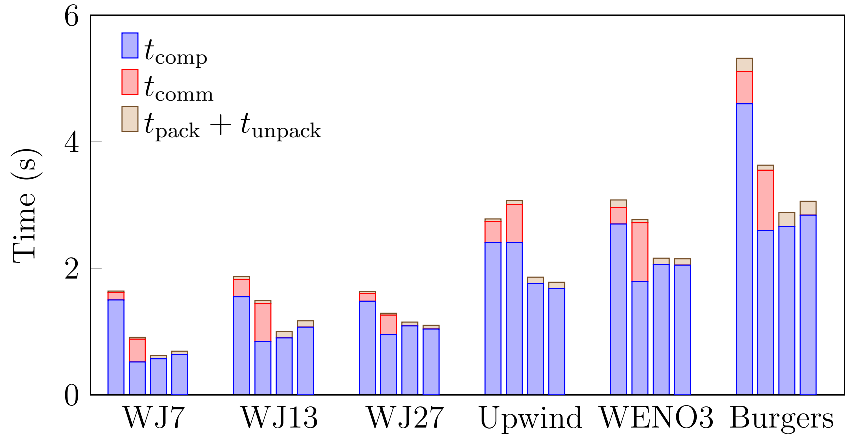
Test	MPIOPT_T	η_{RP}	η_{DC}	vs non-overlap	vs baseline
WJ7	OT 5	0.72	0.90	1.48×	2.77×
WJ13	OT 3	0.69	0.92	1.49×	1.97×
WJ27	TT 8	0.73	0.60	1.20×	1.61×
Upwind	TT 4	0.69	0.57	1.24×	1.68×
WENO3	TT 3	0.69	0.70	1.27×	1.39×
Burgers	OT 2	0.78	0.82	1.24×	1.69×

temporal tiling algorithm (OT vs TT) and the number of fused iterations (f) align closely with the single-node results for all 6 numerical schemes on the 4 stencils. Across both clusters, our pipelined algorithm achieves 50% - 90% of the potential benefit from overlapping the computation and communication leading to a speedup of up to 1.48× over the non-overlapped case. Note that RP's overlap ratio is 8-34% higher than DC among the different stencils on the InfiniBand cluster whereas lower or comparable to DC for most numerical schemes on the Omni-Path cluster. This confirms again the performance of RepeatedPoll highly depends on the software stack of the network. Compared to our baseline with spatial tiling, the overlapped algorithm improves the performance by $1.39 - 2.77\times$ on Bebop and up to

3.36× on HPC3, which validates the effectiveness of Pencil on distributed systems.



(a) HPC3



(b) Bebop

Figure 4.14: Performance breakdown for 6 test cases on HPC3 and Bebop. Each bar from left to right represents the best case of MPI-everywhere and MPI+OpenMP with spatial tiling, Hybrid Tiling with DeepHalo, Hybrid Tiling with DC, and Hybrid Tiling with RP respectively.

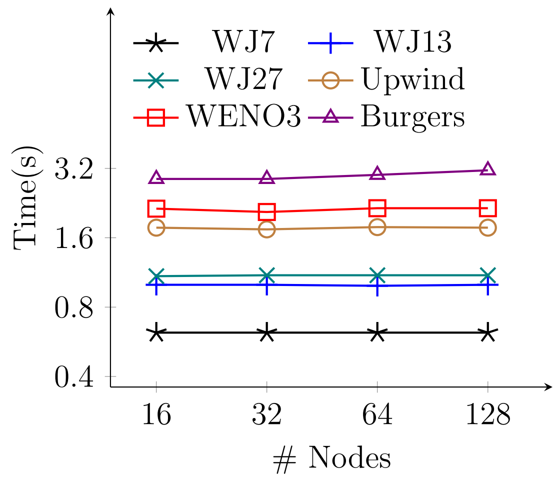
Figure 4.14 breaks down the running time for all the six test cases. On HPC3, we observe a $1.18 - 2.98\times$ speedup after applying the hybrid tiling with DeepHalo. Note that using DeepHalo significantly reduces the communication cost in all the cases except for Upwind. The reduced communication cost only makes a small fraction of the total running time. As a result, when we employ DC or RP for overlap, there is only a minor speedup ($1.08 - 1.20\times$ in Table 4.4). On the contrary, using DeepHalo on Bebop increases the communication time for all the test cases. The worst case appears to be WENO3, where the communication time is increased by 3.5×. Since the communication takes a considerable part of the total time, overlapping with computation effectively improves the overall performance by $1.20-1.48\times$. Contrasting DeepHalo's network-specific performance, we cannot always rely on it to reduce the communication cost. Instead, hiding the communication cost by overlapping with computation is a more reliable and effective optimization.

4.4.4 Weak and Strong Scaling

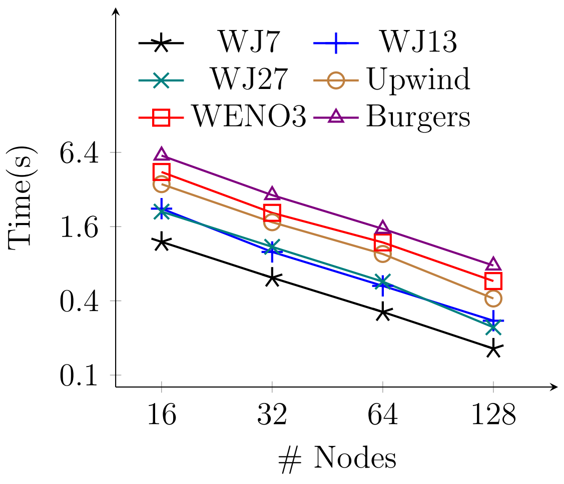
To evaluate the weak and strong scalability of Pencil, we choose Bebop since it has a larger number of nodes. For weak scalability, we maintain the same block sizes and connections per node as in Section 4.3. Pencil exhibits excellent weak scalability up to 128 nodes or 4608 cores across the board for all 6 case studies including Upwind, WENO3, and Burgers as shown in Figure 4.15a. The maximum variance of running time appears in solving the Burger equation, which is about 5% of the average solve time.

The strong scalability results are reported in Figure 4.15b. We test all the numerical schemes on a grid of size $1920 \times 1920 \times 960$. The grid is partitioned evenly into as many blocks as the number of nodes in all the scaling tests from 16 to 128 nodes. Periodic conditions are set on the grid's boundaries so that each node exchanges halos on all the 6 faces of its block. Therefore, both the computation workload and the communication metrics (communication volume and edge cuts) are balanced in all the tests.

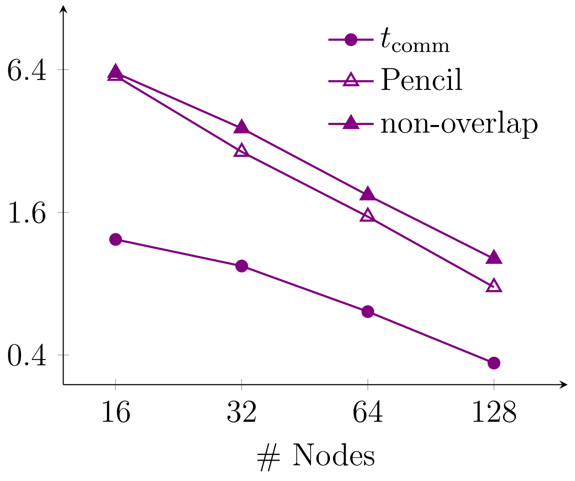
Pencil exhibits near-linear scaling across all the 6 numerical schemes on 4 different stencils. However, the scaling of the non-overlapped temporal tiling algorithm is less efficient because the communication does not scale linearly with the number of nodes. This phenomena is highlighted in Figure 4.15c. When solving the 3D Burgers equation without overlap, the portion of communication in the overall solve time increases from 20% to 37% as the number of nodes increases from 16 to 128. Therefore, the poorer scalability of communication seen in Figure 4.15c has a negative effect on the performance at larger node counts. Pencil effectively hides the communication penalty and exhibits improved strong scaling compared to the non-overlapped case.



(a) Weak scalability



(b) Strong Scalability

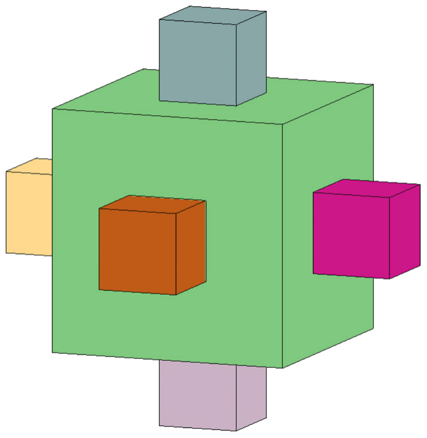


(c) Strong Scaling for Burgers

Figure 4.15: Weak and strong scalability of Pencil up to 128 nodes (or 4608 cores) of Bebop. In the largest simulation for weak scaling, there are 14 billion grid cells. The strong scaling is on a grid of size $1920 \times 1920 \times 960$.

4.4.5 Application to Multi-Block Grids

To apply Pencil to multi-Block grids, we need to attach DeepHalo to the boundaries that connect two blocks and communicate halos of that boundary if the connected blocks reside on different nodes. Once blocks have been partitioned across nodes, how they were originally connected only affects the communication relation and has no influence on temporal tiling and pipelined overlap. Therefore, we can apply Pencil to distributed multi-block grids in the same way as partitioned single-block grids. To demonstrate this capability, we apply it to a multi-block grid illustrated in Figure 4.16a where 5 small blocks are connected to 5 different boundaries of a large block. This geometry emulates a multi-exit pipe transition in engineering.



(a) Geometry

Block	Size
0	$1440 \times 1440 \times 1440$
1	$480 \times 480 \times 480$
2	$480 \times 480 \times 480$
3	$480 \times 480 \times 480$
4	$480 \times 480 \times 480$
5	$480 \times 480 \times 480$

(b) Sizes

Figure 4.16: Geometry and sizes of the multi-block mesh with 6 blocks that are connected at 5 faces.

The gird blocks are partitioned into 32 sub-blocks of size 480^3 and assigned to 32 nodes. Physical boundary conditions are set at all the external faces and communication only occurs at the connection between sub-blocks. Note this test is intractable for overlap methods that decompose the domain only in one dimension [17] but not for Pencil which uses DeepHalo to resolve the blocks' dependencies. Table 4.5 presents the performance results of the 6 test cases on the multi-block grid on 32-nodes of Bebop. We achieve $1.33 - 3.41\times$ speedup over

MPI+OpenMP with spatial tiling.

Table 4.5: Summary of performance on the multi-block grid. Optimal knobs list the overlap method (first entry), temporal tiling algorithm (second entry), and the number of fused iterations (third entry) that achieve the best performance.

Stencils	Optimal knobs	Speedup
WJ7	DC OT 5	3.41×
WJ13	DC OT 3	2.46×
WJ27	DC TT 6	2.30×
Upwind	RP TT 4; DC TT 3	1.55×
Weno3	RP TT 3; DC TT 3	1.33×
Burgers	DC OT 2; RP OT 2	2.20×

4.5 Related Work

Pencil represents a pipelined algorithm for distributed stencil computation. The term *pipelined stencil* or *stencil pipelines* has taken different definitions in various studies aimed at vastly different tasks. In image processing, the stencil pipeline refers to multiple stencil computation stages. Different stages can have completely different stencils and grids in contrast to CFD applications where a fixed set of stencils are used on one grid over many temporal iterations. Domain-Specific Languages (DSLs) and compliers such as PolyMage[60] and Halide [61] embed cache tiling to optimize the stencil pipelines but are still far from delivering a performance that is on-par with hand-tuned codes for real CFD applications [84]. For stencil studies on FPGA or other custom architectures [85], the stencil update is first explicitly broken down into tasks of memory load, store, and arithmetic operations. These tasks are then pipelined among multiple processing elements to enable parallel execution [85–90]. The pipelined approach in Pencil applies exclusively to overlap communication and computation in distributed systems and thereby, differs from prior works.

The most related pipelined algorithm to Pencil is [91, 92] where a pipelined execution is employed to overlap communication and computation among processors. Pencil's key distinction is that the pipelining happens within each process. Moreover, prior work assumes that the data dependence between sub-blocks is only one-way, i.e. a block only sends halo to its target block but does not receive any halo back. This assumption, in general, does not apply to CFD applications such as the Equations 4.4, 4.5, and 4.6 in Section 4.4.1.

Several studies have applied temporal tiling techniques to distributed computation. The Geometric Multi-Grid Solver developed in [76, 93] combines DeepHalo with the overlapped tiling [47] and demonstrates significant speedups for solving the Helmholtz equation in a 3D box. In [17], diamond tiling is extended to distributed systems by partitioning the grid in the j^{th} dimension among processes so that only the two j boundaries need to communicate. Each process's iteration space is decomposed into diamond tiles [51] in a similar way to how we arrange the trapeziums. The communication overlaps with the update of diamonds inside the domain since only diamonds touching the j boundaries depend on halo. The restriction to 1D decomposition works for single-block grids but not multi-block grids where blocks can be connected on any face. Pencil does not impose any constraint on the decomposition.

Domain-Specific Languages (DSLs) such as Distributed Halide [94] and the Oxford Parallel library for Structured meshes (OPS) [95, 96] can tile loops over a single block and distribute tiles among processors with communication routines automatically generated. Nonetheless, they mostly lack the support of sophisticated tiling methods like trapezoidal tiling [49, 50] or diamond tiling [51, 52].

Pluto [57, 58] represents the start-of-the-art in automatic parallelization tools using diamond tiling with polyhedral techniques [53, 54]. First. it decomposes the iteration space into tiles, then distributes the tiles among processes, and finally generates the corresponding MPI routines [97]. Communication is only needed for tiles with data dependencies across nodes and can be overlapped with the computation of tiles satisfying their dependencies within the node. An effective overlap is demonstrated in [18]. However, the application is still limited to the perfectly nested loop structure formed by a single-block grid. To the best of the author's knowledge, none of the state-of-the-art polyhedral compilers or DSLs can directly tile multi-block grids on distributed systems. Therefore, the present study contributes to the state-of-the-art in distributed stencil computation for multi-block grids used in real CFD applications.

4.6 Summary and Future Work

We propose Pencil, a novel pipelined algorithm to extend temporal tiling to multi-block grids for distributed stencil computation. Through an in-depth analysis of single-node performance, we demonstrate how to combine MPI and OpenMP to obtain the best performance of temporal tiling. This optimal hybrid tiling method can be extended to multi-block grids using DeepHalo. Evaluated with various stencils and numerical schemes, Pencil's hybrid tiling outperforms the start-of-the-art tool based on the polyhedral model, Pluto [57, 58] on a single-node by up to 1.9×.

On distributed systems, Pencil achieves overlap by pipelining computation and communication and significantly outperforms MPI-everywhere and MPI+OpenMP on both clusters considered in this thesis. Moreover, it exhibits excellent weak and strong scaling up to 128 nodes on Bebop. At last, we apply Pencil to a multi-block grid with 6 connected blocks and achieve a 1.33-3.41× speedup over MPI+OpenMP with spatial tiling.

The implementation of sophisticated tiling algorithms and overlap methods can be time-consuming and error-prone. For now, all our optimizations are manually implemented for each test case, which works for a small number of benchmarks. To extend Pencil to various complex CFD solvers, further studies will incorporate Pencil into DSLs like OPS [95, 96] or auto-parallelizing tools such as Pluto [57, 58].

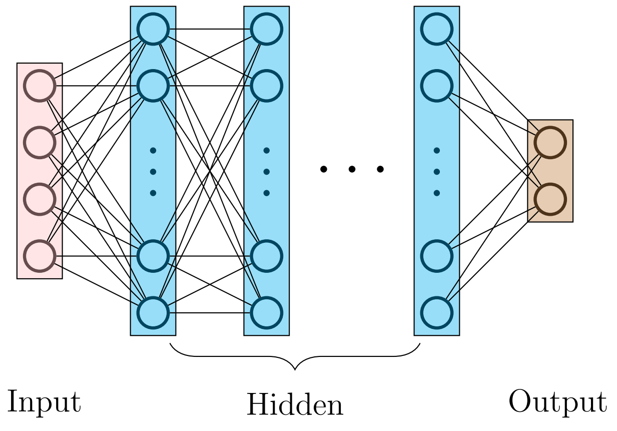
Chapter 5

Transferable Deep Learning Model for BVP

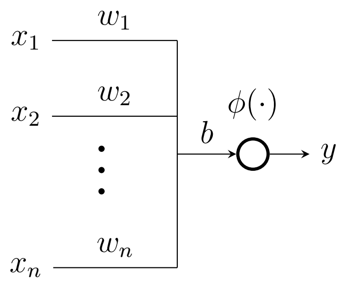
In chapters 3 and 4, we have presented systematic optimizations to each step of the CFD workflow. Nonetheless, the simulation in real engineering applications can still be time-consuming. A further optimization is to design a fast surrogate for CFD solvers, which can infer the flow solution without solving governing equations and only has an acceptable loss of accuracy. Owing to the recent technological advances in *Deep Learning* (DL) or say Deep *Neural Networks*, we can now build accurate surrogates for extremely complicated systems with high dimensional inputs and outputs. Figure 5.1a illustrates the structure of a fully connected neural network, which is composed of layers of neurons. The layers between the input and output are referred to as *hidden layers*. As shown in Figure 5.1b, each neuron forms a linear combination of the input vector $\vec{x}$ and passes that combination through a non-linear activation function ϕ as follows,

$$y = \phi(\sum w_i \cdot x_i + b), \tag{5.1}$$

where w_i and b are referred to as the neuron's *weights* and *bias* respectively. The fully connected neural network passes its input through all the hidden layers, using the prior layer's output as the following layer's input. The weights and biases from all neurons are determined by a tuning process (called *training*), which minimizes a loss function measuring the error between the network's output and the ground truth with gradient decent.



(a) Fully connected network.



(b) A neuron

Figure 5.1: An example of fully connected neural network.

According to the Universal Approximation Theorem [98, 99], a fully connected network with sufficient parameters can approximate any complex mapping between the given inputs and outputs. In the CFD scenario, the network can be used to approximate the mapping between flow snapshots to steady-state solutions [100, 101], or between the domain and boundary info to a PDE's solution [21, 102], etc. These studies require moderate/large data sets for training, which are typically generated from prior flow simulations by CFD solvers. These data-driven approaches face two limitations. First, it is challenging and time-consuming to generate high-quality data for realistic engineering applications. Moreover, the DL model built based on data does not respect the underlying physical laws, leading to solutions violating the conservations laws in fluid dynamics [101].

The pioneering work, Physics-Informed Neural Network (PINN) [19], makes a significant advance towards resolving both limitations by embedding the PDEs governing physical phenomena into the loss function.

For general PDE systems on a spatio-temporal domain $\Omega \times [0, T]$,

$$H(\vec{u}(t, \vec{x})) = f(\vec{x}), \quad \vec{x} \in \Omega, t \in [0, T],$$

where $H(\cdot)$ denotes a non-linear differential operator with derivatives in time and space, PINN approximates the solution $\vec{u}(t, \vec{x})$ with a neural network $\mathcal{N}(t, \vec{x})$ and defines the loss function as

$$\begin{aligned} Loss &= MSE_{pde} + MSE_{bc} + MSE_0 \\ MSE_{pde} &= \frac{1}{N_{col}} \sum^{N_{col}} (H(\mathcal{N}(t, \vec{x}_i)) - f(\vec{x}_i))^2 \\ MSE_{bc} &= \frac{1}{N_{bc}} \sum^{N_{bc}} (\mathcal{N}(t, \vec{x}_i^{bc}) - \vec{u}(t, \vec{x}_i^{bc}))^2 \\ MSE_0 &= \frac{1}{N_0} \sum^{N_0} (\mathcal{N}(0, \vec{x}_i^0) - \vec{u}(0, \vec{x}_i^0))^2, \end{aligned} \tag{5.2}$$

where all the terms are expressed as *Mean Squared Error* (MSE). The first term MSE_{pde} denotes the residual of the governing PDE system. The derivatives in $H(\cdot)$ are computed using the network's auto-differentiation (AD). Minimizing this term during training informs the model of the physical laws. To compute MSE_{pde}, the domain is discretized with N_{col} collocation points. The distribution of collocation points can be adaptively adjusted based on the solution and the domain's geometry, which is much simpler and flexible than the mesh generation in CFD. The second term MSE_{bc} denotes the error in satisfying the boundary condition, where the boundary is resolved by N_{bc} boundary points $\vec{x}^{bc}$. The last term MSE_0 represents the error in satisfying the initial condition on N_0 initial points $\vec{x}^0$, which do not have to be the same as collocation points. Note that the loss function in Equation 5.2 eliminates the need for data in training.

Given a well-posed PDE system, the minimization of Equation 5.2 leads to a neural network $\mathcal{N}$ converging to the exact solution $\vec{u}(t, \vec{x})$. However, when training with gradient descent, the gradients of MSE_{pde} and MSE_{bc} may have orders-of-magnitude difference in eigenvalues, resulting in a stiff system prone to divergence. To remedy, the loss function in Equation 5.2 is augmented with weight averaging and datasets in [20, 103],

$$\begin{aligned}
Loss &= w_0 MSE_{pde} + w_1 MSE_{bc} + w_2 MSE_0 + w_3 MSE_{data} \\
MSE_{pde} &= \frac{1}{N_{col}} \sum^{N_{col}} (H(\mathcal{N}(t, \vec{x}_i)) - f(\vec{x}_i))^2 \\
MSE_{bc} &= \frac{1}{N_{bc}} \sum^{N_{bc}} (\mathcal{N}(t, \vec{x}_i^{bc}) - \vec{u}(t, \vec{x}_i^{bc}))^2 \\
MSE_0 &= \frac{1}{N_0} \sum^{N_0} (\mathcal{N}(0, \vec{x}_i^0) - \vec{u}(0, \vec{x}_i^0))^2 \\
MSE_{data} &= \frac{1}{N_{data}} \sum^{N_{data}} (\mathcal{N}(t, \vec{x}_i^{data}) - \vec{u}(t, \vec{x}_i^{data}))^2
\end{aligned} \tag{5.3}$$

where the weights (different from the weights in neural network) w_0, w_1, w_2 and w_3 prescribes the contribution of each loss term. Assigning proper weights can balance the gradients of different terms and help the training to converge. The fourth term in Equation 5.3 denotes the error on N_{data} data points $\vec{x}^{data}$ inside the domain. The datasets used here can be much smaller than the ones in purely data-driven approaches. Although using data is not necessary, minimizing a share of data error can still ease the training. With a properly tuned loss and network, the state-of-the-art PINN model can infer the steady-state solution of a lid-driven cavity benchmark with $\sim$3% L^2 error in velocity magnitude [20].

Despite the empirical success of physics-informed DL models, it is still very challenging to generalize these models to realistic applications. This is because the developments so far only result in problem-specific models that infer the solution of a PDE system in a particular domain under specific conditions. As mentioned in Chapter 1, in practical engineering designs, we need to simulate flow with different geometries and conditions. For example, we can train a PINN model to substitute CFD solvers in simulating the flow over an airfoil. This model applies to the unique configuration and flow conditions used in training and will become largely useless when simulating under different angles of attack (set by boundary condition) or for different airfoils (set by geometry). Therefore, to truly benefit from DL surrogates in engineering, we need to design *transferable* models that can be trained once on particular geometries and conditions and applied widely to unseen geometries and conditions.

To this end, we propose a novel transferable DL model in this chapter as a surrogate for solving the 2D *Boundary Value Problem* (BVP) of elliptic PDEs, which covers the widely used Poisson-Helmholtz type equations, incompressible Navier-Stokes equations, etc. Section 5.1 describes our network for solving BVP with arbitrary conditions and domain shapes. Section 5.2 introduces a novel iterative inference method that transfer the network's prediction for simple geometries to complex composite domains unseen during training. Section 5.3 presents our results for the 2D Lapace equation. Section 5.4 summarizes the recent works on designing transferable DL models. Finally, Section 5.5 concludes our contributions and points out future directions.

5.1 Neural Network for BVP

We aim at solving the 2D BVP for elliptic PDE(s) on an arbitrary domain Ω,

$$\begin{aligned} H(u) &= f(x,y), \quad (x,y) \in \Omega, \\ u &= g(x,y), \quad (x,y) \in \partial\Omega, \end{aligned} \tag{5.4}$$

where $H(\cdot)$ denotes an elliptic partial differential operator composed by spacial derivatives. Functions f and g represent the source/sink in space and the boundary condition respectively. If $f = 0$, the PDE becomes *homogeneous*. Note that u, f, and g can be vectors, in which case Equation 5.4 represents a system of PDEs such as Navier-Stokes (NS) equations.

The domain Ω is discretized with boundary points $\vec{x}^{bc} \in \partial\Omega$, data points $\vec{x}^{data} \in \Omega$, and collocation points $\vec{x} \in \Omega$. Both the boundary points and data points are extracted from grid points while the collocation points are adaptively distributed based on the solution and the domain's geometry.

To design a neural network transferable across different boundary conditions, we embed the boundary condition $g(x, y)$ and domain geometry $\partial\Omega$ into the network's input as shown in Equation 5.5, where N_{bc}, N_{var} and N_{dim} denote the number of boundary points, variables, and spacial dimensions respectively. $N_{dim} = 2$ since we only study 2D BVP in this thesis. $N_{var} = 1$ for scalar PDEs such as the Laplace equation, Poisson equation, etc. $N_{var} = 3$ for the incompressible NS equations, which involves two velocity components and pressure.

$$\underbrace{g(\vec{x}_1^{bc}),\ g(\vec{x}_2^{bc}),\ \cdots,\ g(\vec{x}_{N_{bc}}^{bc})}_{G\,:\,N_{var}\cdot N_{bc}},\ \underbrace{\vec{x}_1^{bc},\ \vec{x}_2^{bc},\ \cdots,\ \vec{x}_{N_{bc}}^{bc}}_{\partial\Omega\,:\,N_{dim}\cdot N_{bc}},\ \underbrace{\vec{x}}_{N_{dim}} \tag{5.5}$$

The first part of Equation 5.5, G, denote the boundary condition, which is represented by values at all the boundary points. The second part, $\partial\Omega$, depicts the shape of the boundary using the coordinates of boundary points. The last part consists of the coordinates of one input point. As a result, there are in total $(N_{var} + N_{dim}) \cdot N_{bc} + N_{dim}$ elements in the input. If we fix the domain's geometry and only train the network for different boundary conditions, we can omit $\partial\Omega$, reducing the input to $N_{var} \cdot N_{bc} + N_{dim}$ elements.

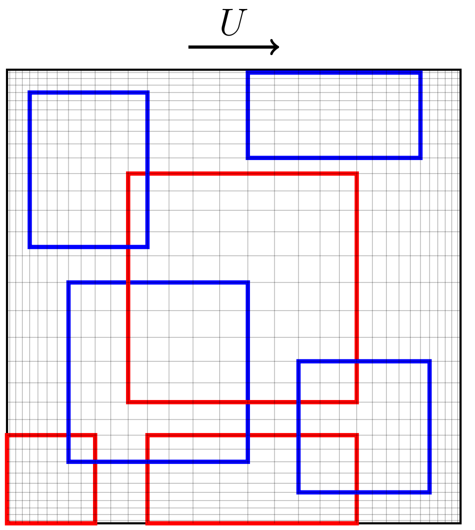
The output of our neural network is the solution $u(\vec{x})$ of N_{var} components at the input point. The loss function of our network is defined similar to Equation 5.3,

$$\begin{aligned} loss = &\frac{w_0}{N_{bc} + N_{data}} \sum^{N_{bc}+N_{data}} (\mathcal{N}(G, \partial\Omega, \vec{x}_i^*) - g(\vec{x}_i^*))^2 \\ &+ \frac{w_1}{N_{col}} \sum^{N_{col}} (H(\mathcal{N}(G, \partial\Omega, \vec{x}_i)) - f(\vec{x}_i))^2, \end{aligned} \tag{5.6}$$

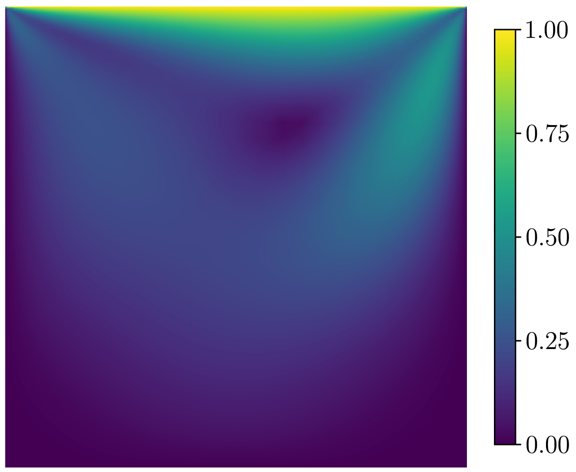
where $\mathcal{N}(G, \partial\Omega, \vec{x}_i^*)$ denotes our neural network approximating the solution $u(\vec{x})$ for a given boundary condition G and boundary shape $\partial\Omega$. In the first term $\vec{x}_i^*$ represents either a boundary point or a data point. Their contributions are merged since both of them are extracted from grid points. The second term stands for the PDE's residual. The derivatives in $H(\cdot)$ is calculated using the network's auto-differentiation with respect to the collocation point's coordinates $\vec{x}_i$. During training, we test different combinations of weights w_0 and w_1 to achieve the most accurate model.

For the hidden layers, we use the fully connected network architecture presented in Figure 5.1. Using more advanced architectures such as the Convolutional Neural Network (CNN) or the Recurrent Neural Networks (RNN) may help to improve the model's accuracy but is beyond the scope of this thesis.

We propose two methods to generate sufficient distinct boundary conditions for the training. First, for simple PDEs such as the Laplace and Poisson equations, there exists a unique solution as long as the boundary condition $g(\vec{x})$ is a smooth function. In this case, we generate $g(\vec{x})$ using a random process and minimize the correlation between samples. For complex PDEs such as the NS equations, the boundary condition must satisfy the underlying physical laws such as mass conservation, momentum conservation, etc. As a result, it becomes highly non-trivial to create realistic boundary conditions from scratch. We propose to first solve the PDEs with realistic boundary conditions on a large domain, then extract solutions for small sub-domains, referred to as *genome*. The solution on each genome's edges forms the boundary condition for that genome. Figure 5.2 shows an example of extracting genomes from a Lid-Driven Cavity benchmark with a non-uniform grid. In general, we can extract boundary points at arbitrary positions and set their values by interpolating the solutions in nearby grid cells. For simplicity, we choose boundary points from grid points and use a genome of fixed size, i.e., 10×10 in this example. As seen from Figure 5.2a, although the genomes have the same number of grid points in the i^{th} and j^{th} dimension, their boundary shapes are still different because the grid is non-uniform. The boundary points inherit their values from the flow solution shown in Figure 5.2b. Moreover, genomes do not have to be rectangular. If we extract boundaries from a curved grid such as the airfoil in Figure 2.1a, the genomes can also be curved.



(a) Genomes extracted from a non-uniform grid.



(b) Solution for the Lid-Driven Cavity benchmark.

Figure 5.2: An example of extracting genomes from the Lid-Driven Cavity benchmark with a non-uniform grid. The colored rectangles mark the genomes. The red and blue colors are only for visual clarity.

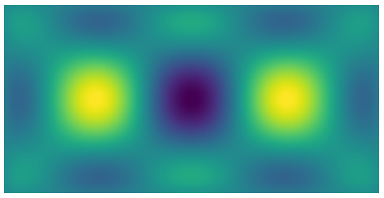
5.2 Iterative Inference Method

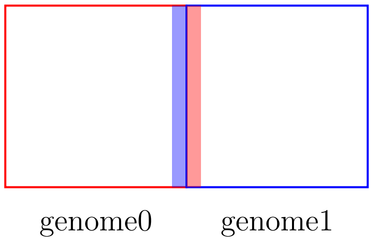
In this section, we introduce an iterative inference method to transfer our model's prediction on genomes to domains of unseen sizes and shapes. For elliptic PDEs, we can decompose the domain into small genomes and solve the BVP on the genomes iteratively. The solution converges to the same solution obtained by directly solving for the entire domain. We substitute the conventional PDE solver with the neural network designed in Section 5.1, which leads to our iterative inference method. We train the neural network by solving BVP for simple genomes with a sufficient amount of distinct boundary conditions. Then, for an unseen domain geometry, we decompose the domain into multiple sets of overlapped genomes. and use our model to infer the solution for genomes iteratively. Genomes in the first set, referred to as *basic genomes*, cover the entire domain and inherit boundary conditions from the domain's boundaries. The following sets of genomes overlap the shared border of the basic genomes to speed up the propagation of boundary information. We infer the solution for genomes set by set and iterate this procedure until the solution on the global domain converges.

We illustrate the iterative procedure by solving the following Poisson equation on a domain of size 2×1.

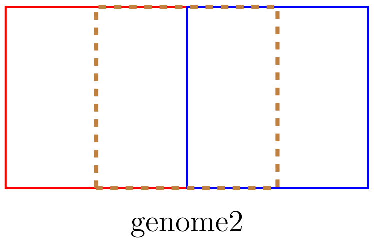
$$\nabla^2 u = -8\pi^2 \cos(2\pi x)\cos(2\pi y)$$

The boundary condition is set uniformly to zero. We discretize the domain with a uniform grid of size 64×32 using Finite Difference and solve the Poisson equation via Algebraic Multi-Grid (AMG) provided by the PyAMG library [104]. The solution is shown in Figure 5.3a. Instead of obtaining the solution in one solve for the entire domain, we can partition the domain into two genomes as shown in Figure 5.3b. Genome 0 and 1 are unit squares sharing the vertical border at the center of the domain. Each genome has three edges aligned with the domain's boundary, which inherit the boundary conditions. We solve for the two genomes by initializing the shared border with zero values. Note that genome 0 and 1 are connected at the shared border just like two blocks are connected in the multi-block grids (Chapter 4). Therefore, we can exchange the halo (marked by red and blue) between genome 0 and 1 and iterate the solve for them. To solve BVP for elliptic PDEs is essentially to propagate the boundary condition into the inside of the domain. Solving for the whole domain ensures each inner grid point receives the boundary info instantly. When solving iteratively with two genomes, one genome's boundary info is gradually passed to the other genome through halo exchange. Hence, we expect the result to converge to the exact solution but with a large number of iterations. This is validated by the convergence history in Figure 5.3d, where the *mean absolute error* (MAE) compared to the solution by AMG drops below the 10^{-14} after 300 iterations.

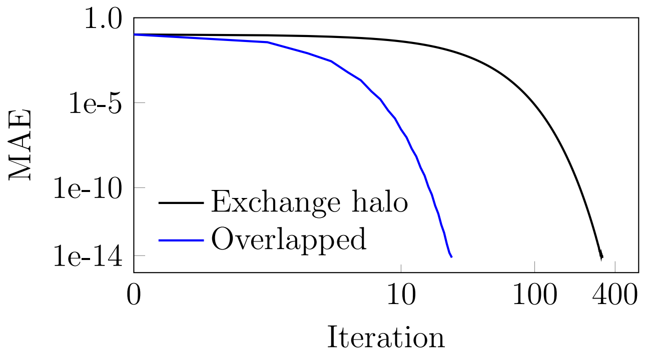

(a) Solution by AMG.



(b) Two genomes.



(c) Overlapped genomes.



(d) Convergence History

Figure 5.3: An example of solving elliptic PDEs by iterating genomes.

We can speed up the propagation of boundary info by adding a third genome overlapping the shared border as shown in Figure 5.3c. After one iteration for genome 0 and 1, we solve for genome 2 using genome 0 and 1's latest results as boundary conditions. This passes genome 0 and 1's boundary info to cells near the shared border. As shown in Figure 5.3d, using overlapped genomes significantly speeds up the convergence and reduces the number of iterations to reach machine precision by 12×. In this example, each genome is still solved by a conventional numerical method and in next section we will demonstrate the effect of employing neural network to infer genomes.

5.3 Experiments and Results

In this section, we present experimental evaluations of our DL model by solving the 2D Laplace equation with Dirichlet boundary conditions.

5.3.1 Laplace Equation

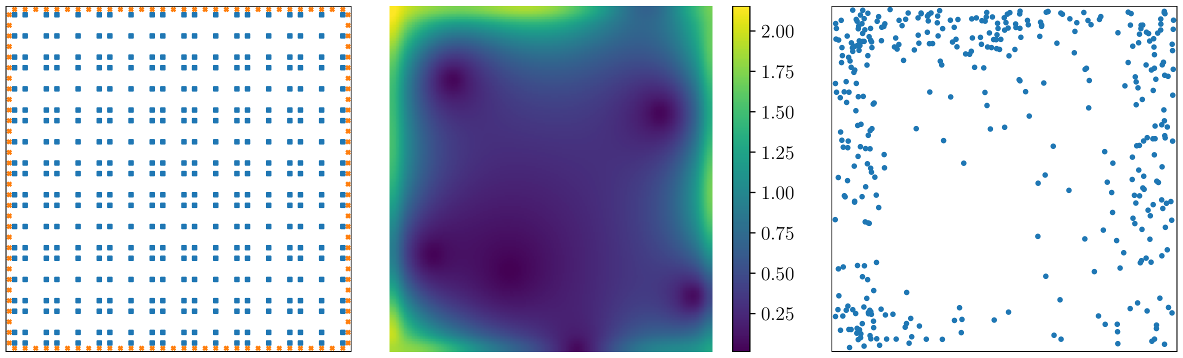
To solve the following Laplace equation, we fix the geometry and focus on our model's ability to handle different boundary conditions.

$$\begin{aligned} \nabla^2 u &= 0, \qquad (x,y) \in \Omega, \\ u &= g(x,y), \quad (x,y) \in \partial\Omega, \end{aligned} \tag{5.7}$$

Training Samples We construct our genomes by solving Equation 5.7 on an unit square, i.e., $\Omega = [0,1] \times [0,1]$. To generate boundary conditions, we sample smooth functions along the perimeter $\partial\Omega$ using Gaussian Process (GP) with a minimization of inter-sample correlation.

Similar to solving the Poisson equation in Section 5.2, we discretize Equation 5.7 using FD on a uniform grid of size 32×32 and solve it with AMG. The boundary points consist of all the $4 \times 32 = 128$ grid cells on the boundaries. The data points are extracted uniformly from grid cells inside the domain. Both the boundary points and data points are fixed during training. The collocation points are initially distributed with a larger density at near-boundary regions, then adapted according to the network's spatial gradients. Figure 5.4 illustrates the distribution of 128 boundary points, 20×20 data points, and 400 collocation points. During training, we update the spatial derivatives of $\mathcal{N}(G, \Omega, \vec{x})$ with network's auto-differentiation and shift more collocation points towards regions with larger gradients, as shown by Figures 5.4b and 5.4c.

Neural Network Architecture Since we fix the geometry, we can omit the coordinates of boundary points in Equation 5.5. For genome size 32×32, we have in total $N_{var} \cdot N_{bc} + N_{dim} = 128 + 2 = 130$ elements in the network's input vector including the boundary values and one boundary/data/collocation point's coordinates. The output is the inferred solution to



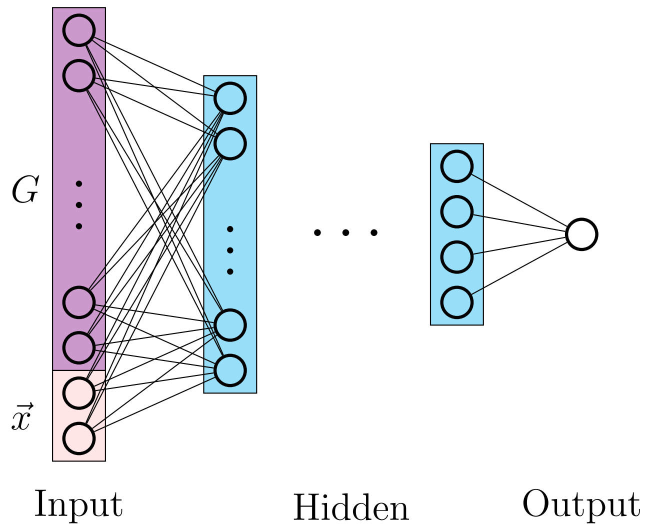
(a) Boundary and data points (b) $||\nabla\mathcal{N}||$ (c) Collocation points

Figure 5.4: Distribution of 128 boundary points, 400 data points, and 400 collocation points

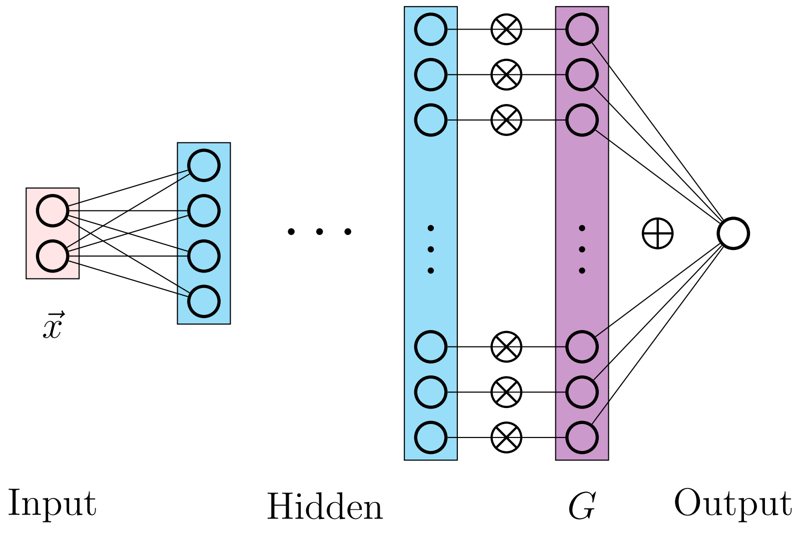
Equation 5.7 at the input point. We consider two architectures for the hidden layers, referred to as Arch 0 and Arch 1 respectively. Arch 0 is composed of fully connected layers following our strategy in Section 5.1, resulting in a coned structure as shown in Figure 5.5a. The second architecture is based on the linearity of Equation 5.7, where the solution at each inner point is essentially a linear combination of the discretized boundary values. As shown in Figure 5.5b, Arch 1's fully connected layers start from the input coordinates $\vec{x}$ and end with a layer containing the same number of neurons as the boundary points. The inference follows $\mathcal{N}(G, \vec{x}) = \sum \alpha_i g(\vec{x}_i^{bc})$, where $\vec{\alpha}$ denotes the outputs from the last hidden layer.

We evaluate the two architectures using a moderate size of samples listed in Table 5.1. For Arch 0, we use 8 hidden layers of sizes $128 \times 128 \times 96 \times 96 \times 64 \times 64 \times 32 \times 32$. We inverse these hidden layers for Arch 1 so that they have the same number of parameters. The networks are implemented with Tensorflow [105]. All layers in both architectures employs the `tanh` activation function. The loss function 5.6 is minimized by the ADAM [106] optimizer. We reduce the learning rate η by 20% when the validation loss descent reaches a plateau and terminates the training when η becomes so small that the loss barely decreases over time. In our experiments, the combination $w_0 = 1$ and $w_1 = 0.001$ leads to the minimum terminal loss.

After training, both models are evaluated using 400 unseen boundary conditions generated



(a) Arch 0



(b) Arch 1

Figure 5.5: Two network architectures for solving BVP for the Laplace equation. Arch 0 passes the boundary values G (violet) and input coordinates $\vec{x}$ (pink) together through fully connected hidden layers (blue). Arch 1 only passes the input coordinates through the hidden layers. The last hidden layer's output and the boundary values are multiplied using vectors' dot product.

Table 5.1: The key parameters and evaluations for Arch 0 and Arch 1. In the training column, from left to right, we list the number of layers, the number of sample boundary conditions, and the number of data/collocation points used in training. For evaluations with unseen boundary conditions (BC), we report the Mean Absolute Error (MAE) and Mean Relative Absolute Error (MRAE) compared to the solution by PyAMG, and the Mean Absolute Residual for Equation 5.7.

Network	Training			Unseen BC		
	Layers	Samples	Points	MAE	MRAE	MAR
Arch 0	8	2000	100\400	2.24e-3	3.28e-2	1.12e-1
Arch 1	8	2000	100\400	7.54e-4	8.58e-3	2.92e-2
Arch 1	14	8000	400\400	6.02e-4	7.06e-3	1.63e-2
Arch 1 DP	14	8000	400\400	5.99e-4	6.98e-3	1.71e-2

by GP. We measure the model's accuracy with three metrics, Mean Absolute Error (MAE), Mean Relative Absolute Error (MRAE), and Mean Absolute Residual (MAR), which are defined as follows,

$$\text{MAE:}\quad |\mathcal{N}(G,\Omega,\vec{x}) - u(\vec{x})| \tag{5.8}$$

$$\text{MRAE:}\quad |\mathcal{N}(G,\Omega,\vec{x}) - u(\vec{x})|/|u(\vec{x})| \tag{5.9}$$

$$\text{MAR:}\quad |H(\mathcal{N}(G,\Omega,\vec{x})) - f(\vec{x})| \tag{5.10}$$

As seen from Table 5.1, all the metrics of Arch 1 is better than Arch 0 by one order of magnitude. This is because Arch 0 applies a non-linear activation to the boundary values, which conflicts with the linearity of Equation 5.7. Note that it is necessary to use non-linear activation functions to compute the second order spatial derivatives in Equation 5.7 with auto-differentiation. However, Arch 1 restricts the non-linear activation to spatial coordinates and retains the linearity of the boundary values, resulting in superior accuracy over Arch 0.

We further improve Arch 1's accuracy by increasing the network's depth and the number of boundary conditions used in training. As shown in Table 5.1, the MAE and MAR are improved by 20% and 80% respectively. The networks so far are built in single precision

(SP), i.e., all the weights and biases are 4-bytes float variables. We also train the same network using double precision (DP). Although it doesn't improve the accuracy compared to SP, it later helps to transfer the model's prediction to unseen domains.

Domains of Unseen sizes and shapes We use our iterative inference method to extend the network's prediction to domains with unseen sizes and shapes. First, we consider domains of the same square shape but with a larger size. Figure 5.6 shows how a 2×2 square is decomposed with three sets of overlapped genomes. The basic genome set consists of four genomes of size 1×1. The remaining two genome sets overlap the vertical and horizontal shared borders between the basic genomes. Each iteration updates all the genomes set by set and each genome extracts its boundary conditions from genomes in the prior set. We terminate the iterations when the difference between adjacent iterations is below a prescribed tolerance, 10^{-6}.

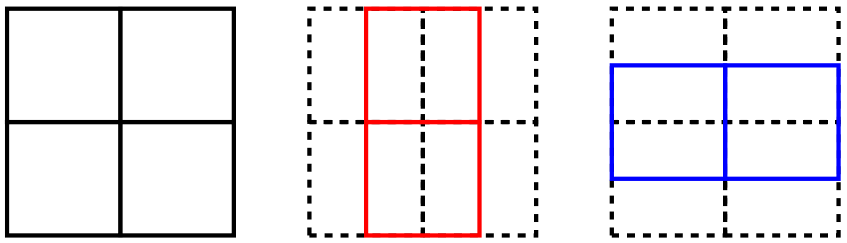
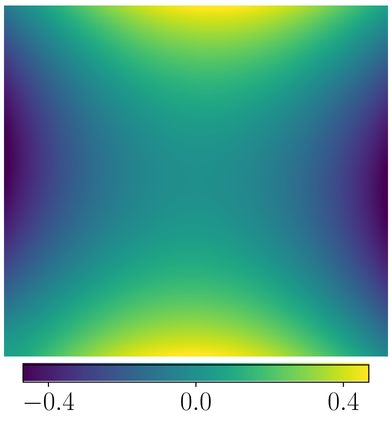


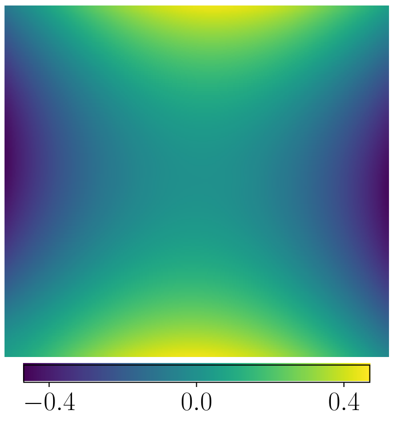
Figure 5.6: Overlapped genomes for a square domain of size 2×2.

Figure 5.7 presents the error between our iterative inference and the solution from AMG. For networks in both SP and DP, most of the errors appear at the genomes' borders. The genomes' skeletons are still visible in the error plots. As shown in Table 5.2, the iterative inference of the DP model results in significantly less error than the SP model, improving the MAE and MAR by 65% and 16% respectively.

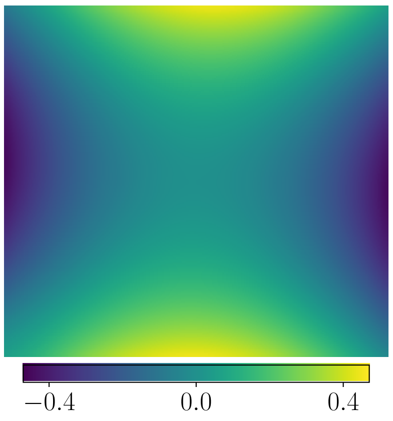
We further increase the domain size to 4×4 and decompose it similarly with three sets of genomes. Note that some of the genomes are completely immersed in the domain, i.e., none



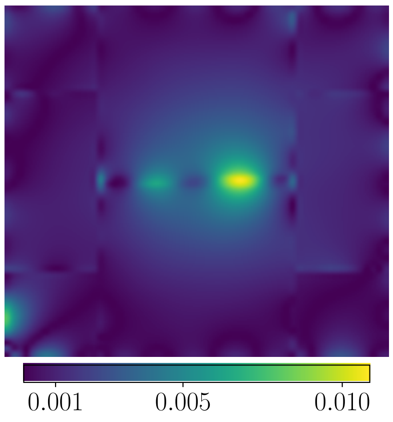
(a) Solution by AMG



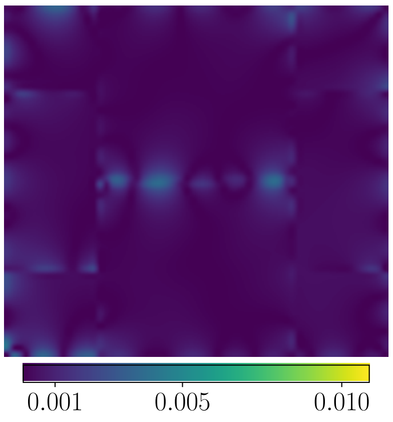
(b) Solution by the SP model



(c) Solution by the DP model



(d) Error from the SP model



(e) Error from the DP model

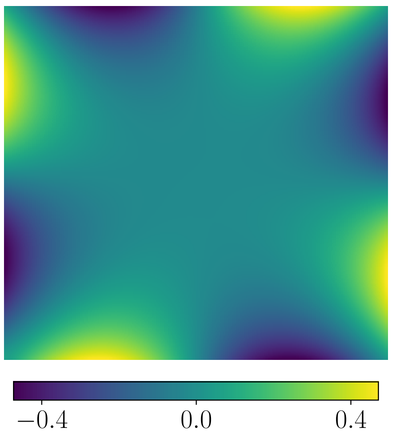
Figure 5.7: Iterative inference for the square domain of size 2×2.

of the genome's boundary overlaps with the domain's boundary, which makes it more difficult to propagate the boundary condition into the domain. The errors of the iterative inference is illustrated in Figure 5.8 and listed in Table 5.2. We observe similar trends that the border of genomes are prone to errors, and the iterative inference of the DP model is significantly more accurate than that of the SP model, even though both models have comparable accuracy for a single genome (Table 5.1).

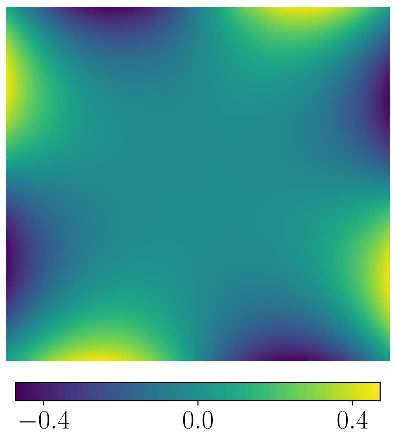
This is because, at each iteration, the DP model accumulates less round-off error than the SP model. Figure 5.9 shows the convergence history of iterative inference for both domains. For the 2×2 square, the errors of the DP and SP models start from the same value and reach plateaus after about the same number of iterations. However, the DP model has accumulated much less round-off error than the SP model, which results in a much better accuracy. The accumulated round-off error also slows down the propagation of boundary info and increases the number of iterations. As shown in Figure 5.9, the SP model uses twice the number of iterations for convergence compared to the DP model for the 4×4 square domain. As a result, even though the SP and DP model have similar accuracy in inferring solutions for different boundary conditions, it is necessary to employ DP to minimize the round-off error when transferring our model's inference to unseen domains.

For both SP and DP models, the error increases as we expand the domain. Besides the round-off error, the error from inference on a single genome, i.e., $O(10^{-4})$ in Table 5.1 also accumulates during iterations. Large domains require considerable iterations to propagate the boundary info and therefore, increase the accumulated error. Nonetheless, for the 4×4 square, which is $16\times$ larger than the genome, our DP model still limits the MRAE below 10%, comparable to state-of-the-art non-transferable DL models.

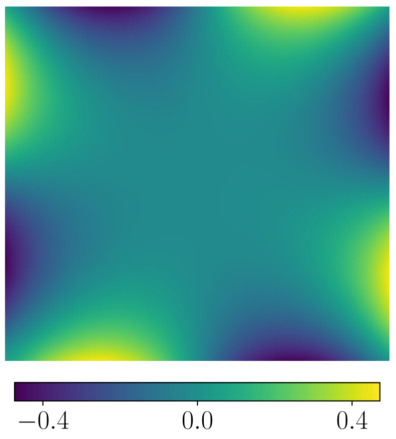
The tests on square domains validate that the iterative inference can transfer our model's prediction to larger domains. We want to further evaluate its ability to transfer prediction across different domain shapes. To this end, we test our model on the cross-shaped domain



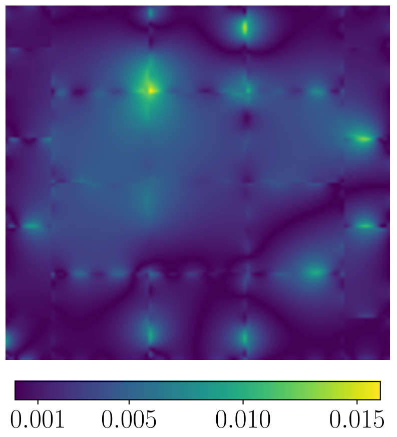
(a) Solution by AMG



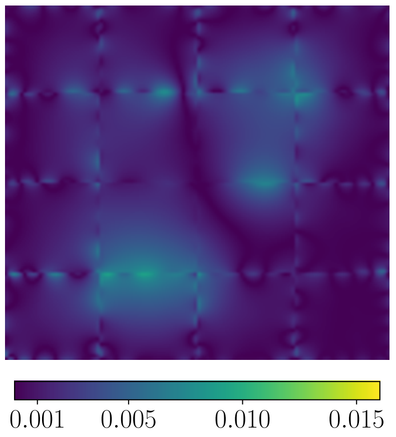
(b) Solution by the SP model



(c) Solution by the DP model



(d) Error from the SP model



(e) Error from the DP model

Figure 5.8: Iterative inference for the square domain of size 4×4.

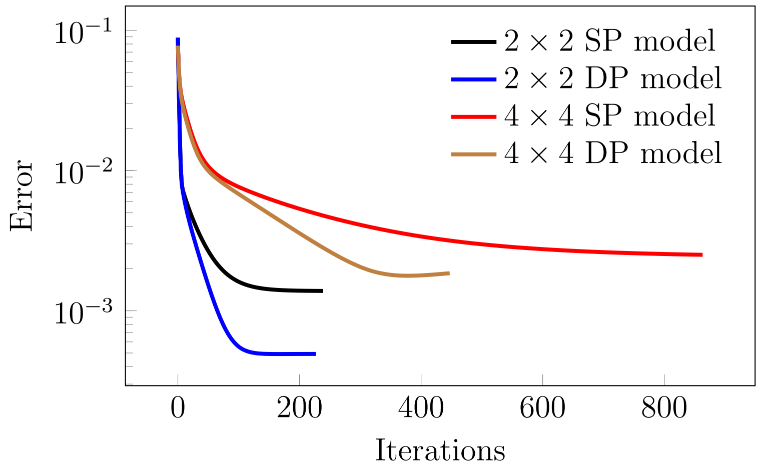


Figure 5.9: Convergence History of SP and DP models for the square domain tests

Table 5.2: Summary of results for the iterative inference of the SP and DP models.

Test	SP Model			DP Model		
	MAE	MRAE	MAR	MAE	MRAE	MAR
Square 2×2	1.38E-3	7.36E-2	3.99E-3	4.93E-4	1.3eE-2	3.35E-3
Square 4×4	2.43E-3	9.39E-2	1.87E-3	1.85E-3	8.12E-2	1.31E-3
Cross	3.18E-3	1.12E-1	4.69E-3	1.70E-3	5.02E-2	3.73E-3

in Figure 5.10. The domain is decomposed into 5 unit square genomes, which form the basic genome set. We include two additional genome sets (marked red and blue) to cover the vertical and horizontal shared borders respectively.

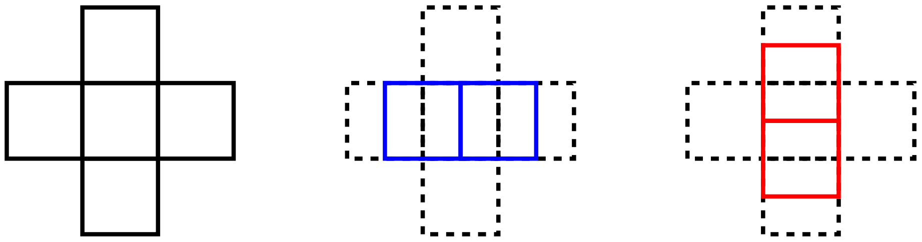
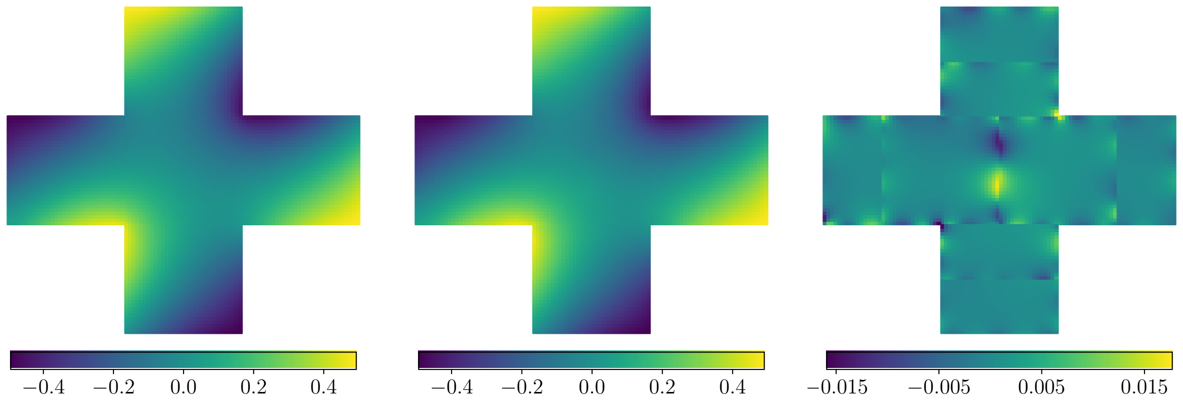


Figure 5.10: Overlapped genomes for a cross-shaped domain.

The solution and inference are illustrated in Figure 5.11 and the error metrics are listed in Table 5.2. The results are consistent with our prior observation that the errors tend to accumulate at the genomes' boundaries. Using the DP model with iterative inference only introduces 5% error compared to the solution by AMG, which validates iterative inference's capability of transferring prediction across unseen shapes.



(a) Solution by AMG (b) Inference (c) Inference Error

Figure 5.11: The solution and DP model's iterative inference for the cross-shaped domain.

5.4 Related Work

The empirical success of PINN [19] has sparkled a large body of work applying DL models to physics-based simulations. Nonetheless, studies specialized at transferring DL model's inference across domains of unseen shapes and sizes are quite limited. In this section, we briefly summarize these works and highlight our contributions to this topic.

The first step to transfer inference across various geometries is to embed the geometry info into the DL model. In [21], Convolutional Neural Networks (CNN) are employed to solve the 2D Poisson equation on rectangular domains. With the domain's width and height as input, the model is transferable across rectangles of different aspect ratios. In [102], a DL model is exploited as a surrogate for the RANS turbulence model in solving Navier-Stokes equations. The model is trained specifically for the flow over a backward facing step. Taking the step height as an input, the model can infer flows over various steps. In [107], the DL model infers the lift and drag of various elliptic objects with the ellipse's aspect ratio as input. Note that such generalizations, though effective in their target applications, are still restricted to the specific type of geometries (rectangle, ellipse, etc).

Several studies [23, 24, 108] have proposed more general techniques to represent the geometry info. CNN is used in [108] to solve the 3D Poisson equation for plume flow over rigid objects, where the object's shape is captured by a 3D array marking whether a grid cell belongs to the object. With the marker array input to the network, the model can predict flows across various object shapes, yet its usage is limited to uniform Cartesian girds of prescribed sizes. In [23], CNN is exploited to predict the flow over different airfoils. The airfoil's shape is resolved by a uniform Cartesian field measuring the distance from each grid cell to the airfoil's surface, which is passed as an image to the convolution layers. The model can potentially predict for geometries other than airfoils but is still limited to the fixed Cartesian grids. The model in [24] takes a 3D single-block structured grids as input and generalize its inference to various domain shapes that can be resolved with that particular block. Generalization methods based on CNN primarily rely on Cartesian grids and are limited to the size and resolution of the grid.

There are fewer works to transfer the model's inference from a small segment to larger domains. In [103], the domain is decomposed into non-overlapped sub-domains. The governing PDEs are penalized on each sub-domain with a separate network and their solutions are stitched together by regularizing the continuity conditions on the sub-domains' interfaces. Like our model, this method can combine sub-domains' solution to infer the global domain. However, it is not transferable at all, i.e., if the boundary condition or the domain shape changes, the model is no longer applicable.

Our model differs from the prior works as follows. First, we present the first work that can transfer a network's inference to domains of both unseen shapes and sizes. Moreover, our model is essentially mesh-free. The extraction of data points from grid points is merely to improve the training but not a necessity. At last but not least, the genomes' update can be performed in parallel, which potentially allows the model to scale on distributed systems.

5.5 Summary and Future Work

We design a DL model based on fully connected neural networks to solve 2D BVP for elliptic PDEs. Based on the model, we introduce a novel iterative inference method that can transfer the model's prediction to domains of unseen sizes and shapes. We present evaluations of our model by solving the linear Laplace equation. The results validate the iterative inference's ability to transfer prediction across various unseen domains with a limited loss of accuracy compared to the conventional PDE solvers.

To extend our model to the non-linear NS equations, the neural network needs to infer multiple flow variables (velocities, pressure, etc) and regularize a system of PDEs, which significantly increases the complexity of training. Moreover, it is unclear whether the iteration of overlapped genomes would converge to the correct solution of the NS equations due to their non-linear nature. An alternative approach is to optimize the values at the shared borders to satisfy the flow's continuity across genomes, which could lead to a more robust convergence. We will investigate these possibilities in our future research.

We have only evaluated our methods using rectilinear geometries, whereas the model can potentially handle arbitrary boundary shapes. Future studies involve extracting genomes from flow over curved objects such as cylinders, ellipse, etc, and evaluations of the iterative inference with unseen curved bodies.

We only implement our algorithm in 2D experiments, whereas the same idea also applies to 3D PDEs. In 3D, the input vector in Equation 5.5 includes a large number of boundary points covering the surfaces of a genome. For a genome of size $32 \times 32 \times 32$, we face $\sim 10^4$ elements in the input, which highly challenges the training of fully connected networks. Moreover, we have observed that most of the errors accumulate near the shared borders. For 3D domains, the shared borders, i.e., surfaces occupy one more degree of freedom, rendering the iterative inference more error-prone. We will address these challenges in our future research.

Chapter 6

Concluding Remarks

6.1 Summary

This dissertation presents high-performance algorithm designs to systemically improve the performance and productivity of CFD solvers using multi-block structured grids on modern architectures. We summarize the main contributions of this dissertation below.

Multi-block structure grid partitioner. The CFD workflow starts with partitioning multi-block grids, which has a substantial impact on load balance and communication cost. In Chapter 3, we present the first cost model in grid partitioning that unifies the algorithm metrics and network properties. Based on the cost model, we propose novel partitioning algorithms, effectively balancing the workload between processors and minimizing the communication cost. Evaluated with the MPI+Threads programming model, our grid partitioner speeds up the communication of the benchmark solver by up to 15×.

Pipelined Distributed Stencil Computation. After finding the optimal partitions for the multi-block grids, we move on to optimize the iterative process of solver computation and halo communication in the CFD workflow. In Chapter 4, we introduce *Pencil*, a pipeline algorithm for distributed stencil computation. Pencil identifies the optimal combination of MPI and OpenMP for temporal tiling and extends the state-of-the-art temporal tiling algorithms for single-blocks grids to multi-block grids via DeepHalo. Moreover, Pencil overlaps the communication with computation via pipelining, which effectively hides the communication cost and enhances strong scaling. Evaluated with numerous stencils and numerical schemes, Pencil outperforms MPI+OpenMP with spatial tiling by up to 3.4$\times$ for a multi-block grid.

Transferable Deep Learning Model for BVP. After optimizing all the critical steps in the CFD workflow, we propose to further improve the productivity of CFD solvers with surrogate modeling. In Chapter 5, we propose a DL surrogate for solving the general BVP for 2D elliptic PDEs. The model is transferable across different boundary conditions and shapes resolved by a given number of boundary points. Furthermore, we introduce a novel iterative inference method to transfer the model's prediction to domains of unseen shapes and sizes. Our experiments demonstrate the model's transferability in solving Laplace and Navier-Stokes equations with a limited loss of accuracy.

6.2 Future Directions

6.2.1 Optimizations for GPU

Current and upcoming supercomputers such as Summit and Aurora all feature multiple GPUs per node. The state-of-the-art CPU-based CFD solvers will be inevitably ported to

GPUs for higher performance. Applying our optimizations on CPU to GPU faces several challenges. Compared to CPU, GPU has much higher flops but disproportionately less cache, which challenges the tiling algorithm. Moreover, the preferred programming model for GPUs is MPI+threads+GPU, where each GPU is driven by one thread and communicates with other GPUs using GPUDirect [109]. How to extend our pipelined algorithm using MPI+threads to exploit MPI+threads+GPU is yet to be explored. We will address these challenges in future studies.

6.2.2 Deep Learning Models for CFD

We anticipate that the surrogate modeling based on DL will impact CFD simulations the same way as CFD replaces experiments in the engineering design process. Towards this trend, there are some fundamental questions to be answered. For instance, what type of hidden layers (Convolutional, Recurrent, Fully Connected, etc) can learn 3D unsteady simulations more efficiently in realistic engineering applications? Can neural networks maintain the Galilean invariance underlying the physical phenomena? How does the inference error propagate over time for unsteady simulations? Future investigations on these issues will help to broaden DL's application in physics-based simulations.

Bibliography

[1] Hengjie Wang and Aparna Chandramowlishwaran. Multi-criteria partitioning of multi-block structured grids. In *Proceedings of the International Conference on Supercomputing*, pages 261–271, 2019.

[2] Edward N. Tinoco, Olaf P. Brodersen, Stefan Keye, Kelly R. Laflin, Edward Feltrop, John C. Vassberg, Mori Mani, Ben Rider, Richard A. Wahls, Joseph H. Morrison, David Hue, Christopher J. Roy, Dimitri J. Mavriplis, and Mitsuhiro Murayama. Summary data from the sixth AIAA CFD drag prediction workshop: CRM cases. *Journal of Aircraft*, 55(4):1352–1379, 2018.

[3] J. Slotnick, A. Khodadoust, J. Alonso, and D. Darmofal. CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences. Technical Report March, 2014.

[4] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang, and Barbara Chapman. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*, 37(9):562–575, 2011.

[5] J. Mark Bull, James Enright, Xu Guo, Chris Maynard, and Fiona Reid. Performance evaluation of mixed-mode OpenMP/MPI implementations. *International Journal of Parallel Programming*, 38(5-6):396–417, 2010.

[6] Hengjie Wang and Aparna Chandramowlishwaran. Pencil: A Pipelined Algorithm for Distributed Stencils. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[7] Anders Ytterström. A Tool for Partitioning Structured Multiblock Meshes for Parallel Computational Mechanics. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(4):336–343, 1997.

[8] Kurt Sermeus and Eric Laurendeau. Parallelization and Performance Optimization of Bombardier Multiblock Structured Navier-Stokes Solver on IBM eserver Cluster 1600. *Aerospace*, (January):1–24, 2007.

[9] K P Apponsah and D W Zingg. A Load Balancing Tool for Structured Multi-Block Grid CFD Applications. In *20th Annual Conference of the CFD Society of Canada*, 2012.

[10] E. Ahusborde and S. Glockner. A 2D block-structured mesh partitioner for accurate flow simulations on non-rectangular geometries. *Computers and Fluids*, 43(1):2–13, 2011.

[11] M Jahed Djomehri, Rupak Biswas, Noe Lopez-Benitez, and Bryan Biegel. Load balancing Strategies for Multi-Block Overset Grid Applications. 2002.

[12] Min Xiong, Chuanfu Xu, Xiang Gao, Dali Li, Dandan Qu, Zhenghua Wang, and Xiaogang Deng. Improved grid partitioning algorithms for load-balancing high-order structured aerodynamics simulations. *Computers and Electrical Engineering*, 67:70–84, 2018.

[13] J Rantakokko. Partitioning strategies for structured multiblock grids. *Parallel Computing*, 26(12):1661–1680, 2000.

[14] Hongkang Liu, Chao Yan, Yatian Zhao, and Boxi Lin. An improved partitioning strategy for structured multiblock grids. *Proceedings of 2016 7th International Conference on Mechanical and Aerospace Engineering, ICMAE 2016*, pages 322–326, 2016.

[15] David G Wonnacott and Michelle M Strout. On the Scalability of Loop Tiling Techniques. *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, 2013.

[16] Emna Hammami and Yosr Slama. An overview on loop tiling techniques for code generation. *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, 2017-Octob:280–287, 2018.

[17] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates, 2015.

[18] Youcef Barigou and Edgar Gabriel. Maximizing Communication–Computation Overlap Through Automatic Parallelization and Run-time Tuning of Non-blocking Collective Operations. *International Journal of Parallel Programming*, 45(6):1390–1416, 2017.

[19] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[20] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and Mitigating Gradient Pathologies in Physics-Informed Neural Networks. *arXiv*, pages 1–28, 2020.

[21] Ali Girayhan Özbay, Sylvain Laizet, Panagiotis Tzirakis, Georgios Rizos, and Björn Schuller. Poisson CNN: Convolutional Neural Networks for the Solution of the Poisson Equation with Varying Meshes and Dirichlet Boundary Conditions. pages 1–36, 2019.

[22] Chiyu Max Jiang, Soheil Esmaeilzadeh, Kamyar Azizzadenesheli, Karthik Kashinath, Mustafa Mustafa, Hamdi A Tchelepi, Philip Marcus, Anima Anandkumar, and Others. MeshfreeFlowNet: A Physics-Constrained Deep Continuous Space-Time Super-Resolution Framework. *arXiv preprint arXiv:2005.01463*, 2020.

[23] Saakaar Bhatnagar, Yaser Afshar, Shaowu Pan, Karthik Duraisamy, and Shailendra Kaushik. Prediction of aerodynamic flow fields using convolutional neural networks. *Computational Mechanics*, 64(2):525–545, 2019.

[24] Nils Wandel, Michael Weinmann, and Reinhard Klein. Fast Fluid Simulations in 3D with Physics-Informed Deep Learning. *arXiv*, pages 1–10, 2020.

[25] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, Knoxville, TN, USA, 2015.

[26] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009*, (c):427–436, 2009.

[27] Rohit Zambre, Aparna Chandramowlishwaran, and Pavan Balaji. Scalable communication endpoints for MPI+Threads applications. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 803–812. IEEE, 2018.

[28] Rohit Zambre, Aparna Chandramowliswharan, and Pavan Balaji. How i learned to stop worrying about user-visible endpoints and love MPI. In *Proceedings of the International Conference on Supercomputing*, 2020.

[29] Y. P. Chien, F. Carpenter, A. Ecer, and H. U. Akay. Load-balancing for parallel computation of fluid dynamics problems. *Computer Methods in Applied Mechanics and Engineering*, 120(1-2):119–130, 1995.

[30] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[31] R Leland. The Chaco User's Guide Version 2.0. Technical report, Technical Report SAND95-2344, Sandia National Laboratories, Albaquerque, NM 87185-1110, 1995.

[32] Laurent Y.M. Gicquel, N. Gourdain, J. F. Boussuge, H. Deniau, G. Staffelbach, P. Wolf, and Thierry Poinsot. High performance parallel computing of flows in complex geometries. *Comptes Rendus - Mecanique*, 339(2-3):104–124, 2011.

[33] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical Systems Theory*, 1994.

[34] Roger W Hockney and Chris R Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Institute of Physics Publishin, 1981.

[35] Torsten Hoefler, William Gropp, Rajeev Thakur, and Jesper Larsson Träff. Toward performance models of MPI implementations for understanding application scaling issues. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6305 LNCS:21–30, 2010.

[36] F Ino, N Fujimoto, and K Hagihara. LogGPS: A parallel computational model for synchronization analysis. *ACM SIGPLAN Notices*, 36(7):133–142, 2001.

[37] Shahid H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, 1987.

[38] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation - 25 years of DAC*, pages 241–247, 1988.

[39] SpaceX. *Falcon User's Guide*. 2020.

[40] Qingyu Meng, Justin Luitjens, and Martin Berzins. Dynamic task scheduling for scalable parallel AMR in the Uintah framework. Technical report, Citeseer, 2010.

[41] Kwesi Parry Apponsah. *MULTI-BLOCK CFD APPLICATIONS APPLIED TO A PARALLEL NEWTON-KRYLOV ALGORITHM*. PhD thesis, University of Toronto, 2012.

[42] P. Lubin and S. Glockner. Numerical simulations of three-dimensional plunging breaking waves: Generation and evolution of aerated vortex filaments. *Journal of Fluid Mechanics*, 767:364–393, 2015.

[43] M. Jahed Djomehri and Rupak Biswas. Performance enhancement strategies for multi-block overset grid CFD applications. *Parallel Computing*, 29(11-12 SPEC.ISS.):1791–1810, 2003.

[44] Paul R. Eller, Torsten Hoefler, and William Gropp. Using performance models to understand scalable Krylov solver performance at scale for structured grid problems. pages 138–149, 2019.

[45] F. Rastello and T. Dauxois. Efficient tiling for an ODE discrete integration program: Redundant tasks instead of trapezoidal shaped-tiles. *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2002*, (January 2002):246–253, 2002.

[46] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. *ACM SIGPLAN Notices*, 42(6):235–244, 2007.

[47] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010*, (November), 2010.

[48] Bahareh Mostafazadeh, Ferran Marti, Behnam Pourghassemi, Feng Liu, and Aparna Chandramowlishwaran. Unsteady Navier-Stokes computations on GPU architectures. In *23rd AIAA Computational Fluid Dynamics Conference, 2017*, 2017.

[49] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. *Proceedings of the International Conference on Supercomputing*, 1(212):361–366, 2005.

[50] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems*, 45(2):203–233, 2009.

[51] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans Peter Seidel. Cache accurate time skewing in iterative stencil computations. *Proceedings of the International Conference on Parallel Processing*, pages 571–581, 2011.

[52] Daniel Orozco and Guang Gao. Mapping the FDTD application to many-core chip architectures. *Proceedings of the International Conference on Parallel Processing*, pages 309–316, 2009.

[53] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, pages 1–11, 2012.

[54] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298, 2017.

[55] Liang Yuan, Yunquan Zhang, Peng Guo, and Shan Huang. Tessellating stencils. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*, 2017.

[56] Liang Yuan, Shan Huang, Yunquan Zhang, and Hang Cao. Tessellating Star Stencils. *ACM International Conference Proceeding Series*, 2019.

[57] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices*, 43(6):101–113, 2008.

[58] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4959 LNCS:132–146, 2008.

[59] Yuan Tang, Rezaul Chowdhury, Chi-keung Luk, Bradley C Kuszmaul, and Charles E Leiserson. The Pochoir Stencil Compiler Categories and Subject Descriptors. *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 117–128, 2011.

[60] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic Optimization for Image Processing Pipelines. *ACM SIGARCH Computer Architecture News*, 43(1):429–443, 2015.

[61] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide:A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[62] Min Si, Antonio J. Pena, Jeff Hammond, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures. *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, pages 665–676, 2015.

[63] GERALD SCHUBERT, HOLGER FEHSKE, GEORG HAGER, and GERHARD WELLEIN. Hybrid-Parallel Sparse Matrix-Vector Multiplication With Explicit Communication Overlap on Current Multicore-Based Systems. *Parallel Processing Letters*, 21(03):339–358, 2011.

[64] Alexandre Denis and François Trahay. MPI overlap: Benchmark and analysis. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 258–267. IEEE, 2016.

[65] Mustafa Abduljabbar B, George S Markomanolis, and Huda Ibeid. Communication Reducing Algorithms for Distributed Hierarchical N-Body Problems with Boundary Distributions. 10524:79–96, 2017.

[66] Min Si and Pavan Balaji. Process-Based Asynchronous Progress Model for MPI Point-to-Point Communication. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 206–214. IEEE, 2017.

[67] Timothy H. Kaiser and Scott B. Baden. Overlapping communication and computation with OpenMP and MPI. *Scientific Programming*, 9(2-3):73–81, 2001.

[68] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping Communication and Computation by Using a Hybrid MPI / SMPSs Approach. 2010.

[69] Mao Jiayin, Song Bo, Wu Yongwei, and Yang Guangwen. Overlapping communication and computation in MPI by multithreading. *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, (February):2–7, 2006.

[70] Paul R. Eller, Torsten Hoefler, and William Gropp. Using performance models to understand scalable Krylov solver performance at scale for structured grid problems. *Proceedings of the International Conference on Supercomputing*, pages 138–149, 2019.

[71] Ning Li and Sylvain Laizet. 2DECOMP & FFT-A Highly Scalable 2D Decomposition Library and FFT Interface. *Cray User Group 2010 conference*, pages 1–13, 2010.

[72] Liang Shi, Markus Rampp, Björn Hof, and Marc Avila. A hybrid MPI-OpenMP parallel implementation for pseudospectral simulations with application to Taylor-Couette flow. *Computers and Fluids*, 106:1–11, 2015.

[73] Nikolaos Drosinos and Nectarios Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2004 (Abstracts and CD-ROM)*, 18(C):193–202, 2004.

[74] Hormozd Gahvari, Martin Schulz, and Ulrike Meier Yang. An approach to selecting thread + process mixes for hybrid MPI + OpenMP applications. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2015-Octob:418–427, 2015.

[75] Martin J. Chorley and David W. Walker. Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters. *Journal of Computational Science*, 1(3):168–174, 2010.

[76] Protonu Basu, Anand Venkat, Mary Hall, Samuel Williams, Brian Van Straalen, and Leonid Oliker. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. *20th Annual International Conference on High Performance Computing, HiPC 2013*, pages 452–461, 2013.

[77] Protonu Basu. *Compiler optimizations and autotuning for stencils and geometric multigrid*. PhD thesis, The University of Utah, 2016.

[78] Tom Henretty, Richard Veras, Franz Franchetti, Louis Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. *Proceedings of the International Conference on Supercomputing*, pages 13–24, 2013.

[79] Chris Ding and Yun He. A ghost cell expansion method for reducing communications in solving PDE problems. 94720(November):50–50, 2001.

[80] Min Si, Antonio J. Pena, Jeff Hammond, Pavan Balaji, and Yutaka Ishikawa. Scaling NWChem with efficient and portable asynchronous communication in MPI RMA. *Proceedings - 2015 IEEE/ACM 15th International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2015*, pages 811–816, 2015.

[81] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial, Second Edition*. 2000.

[82] Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. Tiling and optimizing time-iterated computations on periodic domains. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 39–50, 2014.

[83] Xu Dong Liu. Weighted essentially non-oscillatory schemes. *Journal of Computational Physics*, 115(1):200–212, 1994.

[84] Bahareh Mostafazadeh, Ferran Marti, Feng Liu, and Aparna Chandramowlishwaran. Roofline guided design and analysis of a multi-stencil cfd solver for multicore performance. *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium, IPDPS 2018*, pages 753–762, 2018.

[85] Matthias Christen, Olaf Schenk, Peter Messmer, Esra Neufeld, and Helmar Burkhart. Accelerating stencil-based computations by increased temporal locality on modern multi-and many-core architectures. *High-performance and hardware-aware computing: Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'08)*, (June 2014):47–54, 2008.

[86] Wang Luzhou, Kentaro Sano, and Satoru Yamamoto. Domain-specific language and compiler for stencil computation on FPGA-based systolic computational-memory array. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012.

[87] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):695–705, 2014.

[88] Keisuke Dohi, Koji Okina, Rie Soejima, Yuichiro Shibata, and Kiyoshi Oguri. Performance modeling of stencil computing on a stream-based FPGA accelerator for efficient design space exploration. *IEICE Transactions on Information and Systems*, 2015.

[89] Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Shunsuke Tatsumi, and Masanori Hariyama. OpenCL-based FPGA-platform for stencil computation and its optimization methodology. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1390–1402, 2017.

[90] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. *FPGA 2018 - Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018-Febru:153–162, 2018.

[91] N. Koziris, A. Sotiropoulos, and G. Goumas. A pipelined schedule to minimize completion time for loop tiling with computation and communication overlapping. *Journal of Parallel and Distributed Computing*, 63(11):1138–1151, 2003.

[92] Georgios Goumas, Nikos Anastopoulos, Nectarios Koziris, and Nikolas Ioannou. Overlapping computation and communication in SMT clusters with commodity interconnects. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, pages 1–10, 2009.

[93] Protonu Basu, Mary Hall, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Phillip Colella. Compiler-Directed Transformation for Higher-Order Stencils. *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, pages 313–323, 2015.

[94] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed halide. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, 2016.

[95] István Z. Reguly, Gihan R. Mudalige, Michael B. Giles, Dan Curran, and Simon McIntosh-Smith. The OPS domain specific abstraction for multi-block structured grid computations. In *Proceedings of WOLFHPC 2014: 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Stor*, 2014.

[96] Istvan Z. Reguly, Gihan R. Mudalige, and Michael B. Giles. Loop tiling in large-scale stencil codes at run-time with OPS. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):873–886, 2018.

[97] Uday Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, pages 1–12, 2013.

[98] Kurt Hornik. Approximation Capabilities of Multilayer Neural Network. *Neural Networks*, 4(1991):251–257, 1991.

[99] Chen Debao. Degree of approximation by superpositions of a sigmoidal function. *Approximation Theory and its Applications*, 9(3):17–28, 1993.

[100] Renee Swischuk, Laura Mainini, Benjamin Peherstorfer, and Karen Willcox. Projection-based model reduction: Formulations for physics-based machine learning. *Computers and Fluids*, 179:704–717, 2019.

[101] Octavi Obiols-Sales, Abhinav Vishnu, Nicholas Malaya, and Aparna Chandramowlishwaran. CFDNet: a deep learning-based accelerator for fluid simulations. In *Proceedings of the International Conference on Supercomputing*, 2020.

[102] Romit Maulik, Himanshu Sharma, Saumil Patel, Bethany Lusch, and Elise Jennings. Accelerating RANS turbulence modeling using potential flow and machine learning. pages 1–21, 2019.

[103] Ameya D Jagtap and George Em Karniadakis. Extended physics-informed neural networks (XPINNs) : A generalized space-time domain decomposition based deep learning framework for nonlinear partial di ff erential equations. (November):1–23, 2020.

[104] L N Olson and J B Schroder. PyAMG: Algebraic Multigrid Solvers in Python v4.0, 2018.

[105] Eugene Brevdo Martín Abadi, Ashish Agarwal, Paul Barham, Andy Davis Zhifeng Chen, Craig Citro, Greg S. Corrado, Ian Goodfellow Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Yangqing Jia Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Mike Schuster Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Jonathon Shlens Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Paul Tucker Benoit Steiner, Ilya Sutskever, Kunal Talwar, Fernanda Viégas Vincent

Vanhoucke, Vijay Vasudevan, Martin Wicke Oriol Vinyals, Pete Warden, Martin Wattenberg, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015.

[106] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, 2015.

[107] Shantanu Shahane, Purushotam Kumar, and Surya Pratap Vanka. Convolutional Neural Network for Flow over Single and Tandem Elliptic Cylinders of Arbitrary Aspect Ratio and Angle of Attack. *arXiv*, 2020.

[108] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. *34th International Conference on Machine Learning, ICML 2017*, 7:5258–5267, 2017.

[109] NVIDIA. GPUDirect, https://developer.nvidia.com/gpudirect.