Pencil: A pipelined algorithm for distributed stencils

Hengjie Wang
University of California Irvine
hengjiew@uci.edu

Aparna Chandramowlishwaran
University of California Irvine
amowli@uci.edu

***Abstract*—Stencil computations are at the core of various Computational Fluid Dynamics (CFD) applications and have been well-studied for several decades. Typically they're highly memory-bound and as a result, numerous tiling algorithms have been proposed to improve its performance. Although efficient, most of these algorithms are designed for single iteration spaces on shared-memory machines. However, in CFD, we are confronted with multi-block structured girds composed of *multiple connected iteration spaces* distributed across many nodes.**

In this paper, we propose a *pipelined stencil* algorithm called *Pencil* for distributed memory machines that applies to practical CFD problems that span multiple iteration spaces. Based on an in-depth analysis of cache tiling on a single node, we first identify both the optimal combination of MPI and OpenMP for temporal tiling and the best tiling approach, which outperforms the state-of-the-art automatic parallelization tool Pluto by up to 1.92×. Then, we adopt DeepHalo to decouple the multiple connected iteration spaces so that temporal tiling can be applied to each space. Finally, we achieve overlap by pipelining the computation and communication without sacrificing the advantage from temporal cache tiling. Pencil is evaluated using 4 stencils across 6 numerical schemes on two distributed memory machines with Omni-Path and InfiniBand networks. On the Omni-Path system, Pencil exhibits outstanding weak and strong scalability for up to 128 nodes and outperforms MPI+OpenMP Funneled with space tiling by 1.33-3.41× on a multi-block grid with 32 nodes.

***Index Terms*—Computational fluid dynamics, stencils, multiple connected iteration spaces, cache tiling, pipelining, distributed-memory machines**

I. Introduction

Stencils are the dominant computational pattern in Computational Fluid Dynamics (CFD) using structured grids. A stencil is characterized by a regular shape formed by a grid cell to update and its data-dependent neighboring cells. Figure 1 shows three different stencil shapes in a 2D structured grid, where the update of the blue cell depends on its pink neighbors. We define the radius of a stencil as the largest distance between the cell to update and its dependent neighbors. In stencil computations, the number of floating-point operations (flops) and DRAM I/O are proportional to the grid size. As a result, stencils are notorious for being memory-bound on modern architectures with high machine balance. Cache tiling, specifically temporal tiling with polyhedral techniques is an effective optimization for memory-bound applications. Temporal tiling views the iterations traversing spatial dimensions and the iterations in time as a single iteration space and decomposes this space into polyhedral tiles. The fused iterations can significantly improve cache reuse and performance. Several temporal tiling algorithms have been proposed in literature such as overlapped tiling [1]–[4], trapezoidal tiling [5], [6], diamond tiling [7]–[10], and tessellating tiling [11], [12]. A detailed summary of the state-of-the-art tiling algorithms can be found in [13], [14]. Tilings can also be applied via automatic parallelization tools like Pluto [15], [16] and domain-specific compilers like Pochoir [17], PolyMage[18], and Halide [19]. Nonetheless, the above tiling algorithms and tools are aimed at stencil computations on shared-memory machines and, more strictly, in a *single iteration space*.

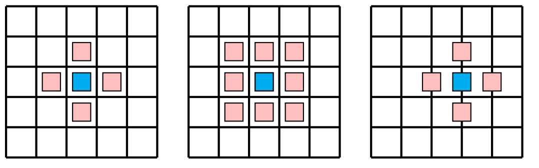


Fig. 1: Different stencil shapes in 2D with a radius of 1. The color represents the value assigned to the cell. In Star and Box stencils, the value is at the cell's center. For Staggered stencils, the value is located on the cell's face.

Structured grids are organized into blocks (i.e. rectangular shapes in spatial dimensions). Typically, structured grids for complex geometries such as an aircraft or turbo-machinery contain on the order of 100's of blocks [20]. In such *multi-block grids*, each block with the iterations in time forms an iteration space. The iteration spaces have dependencies where the blocks are connected. Therefore, in real CFD applications, we are confronted with *multiple connected iteration spaces*. On distributed systems, a multi-block grid is partitioned into sub-blocks and distributed across multiple nodes. Therefore, processors have to communicate to exchange data (called halo layers as thick as the stencil radius) at the blocks' boundaries that are connected. This halo exchange happens after each computation step in iterative stencil applications. To the best of our knowledge, none of the above state-of-the-art tiling algorithms and tools apply to multiple connected iteration spaces on distributed memory machines.

The halo exchange in distributed stencil computations is most commonly implemented using the Message Passing Interface (MPI), based on which two parallel models are widely used – flat MPI and MPI+threads. In flat MPI, the halo exchange happens not only among cores on different nodes (inter-node) but also cores on the same node (intra-node). Presumably, MPI+threads should outperform flat MPI because threads can avoid intra-node communication by accessing shared memory. Among the available thread packages, MPI+OpenMP is the most popular combination for OpenMP's simplicity and wide support from compilers. Prior works have compared MPI+OpenMP against flat MPI [21]–[26] and we refer readers to [27] for a comprehensive summary. We have no intention to make another comparison except to highlight that most studies overlook the single-node case.

In this paper, we are interested in optimizing communication at the user-level i.e. without modifying MPI or the underlying network software. One approach is to combine halo layers for several iterations together to reduce the number of communications at the expense of redundant computation. This optimization proposed at least as early as [28] has been widely used under different names such as overlap area [29], ghost cell expansion [30], and deep halo [31]. Although improvements have been found in some studies [30], [32], the effectiveness of this technique highly depends on the network.

A more important optimization is to overlap communication with computation. MPI implements non-blocking routines for this purpose. However, as discovered by several studies [33]–[36], simply inserting computation between non-blocking send/receive (put/get) and wait routines (window fences) does not result in an overlap. The MPI standard [37] 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 [33], [34], [38]–[41]. Alternatively, one can repeatedly poll `MPI_Test` to urge the network to make progress, which has been proven effective by various studies and applications [35], [42]–[44]. Both methods need the computation to be divided into communication-dependent and independent parts. Such division should not forfeit the benefit from temporal tiling. However, state-of-the-art overlap algorithms that work with cache tiling [42], [45] are still restricted to a single iteration space.

In this paper, we present a distributed stencil algorithm called *Pencil* that extends polyhedral temporal tiling to multiple connected iteration spaces and further hides communication using pipelining. To that end, this paper makes the following contributions.

- We identify an optimal decomposition of MPI ranks and OpenMP threads for hybrid temporal tiling on a single node. This hybrid tiling is evaluated on 4 distinct stencils with 6 numerical schemes solving the well-studied 3D Poisson equation using weighted Jacobi (on star and box stencils) to 3D Burgers equation (on a staggered stencil) which is more complex and has received far less attention. It achieves $1.09 - 3.29\times$ speedup over OpenMP with space tiling and outperforms Pluto [15], [16] by up to $1.92\times$ (Section IV-A).
- We exploit *DeepHalo* for distributed stencils not to decrease communication but to enable temporal tiling for multi-block grids that span multiple connected iteration spaces (Section II-C).
- We propose a fine-grained pipeline algorithm, *Pencil* to overlap communication with computation that retains the benefit of temporal cache tiling (Section II-C). When combined with hybrid tiling, Pencil demonstrates $1.39 - 2.77\times$ and $1.27 - 3.36\times$ speedups over the best case between flat MPI and MPI+OpenMP using space tiling on two distributed memory machines with Omni-Path and InfiniBand networks respectively (Section IV-C). Moreover, Pencil exhibits excellent weak scaling and near-linear strong scaling on 16-128 nodes (Section IV-D). Finally, we apply Pencil to a multi-block grid with multiple connected iteration spaces and show $1.33 - 3.41\times$ speedup over MPI+OpenMP Funneled with space tiling (Section IV-E).

II. Methodology

In this section, we first discuss how the choice of the programming model impacts the single-node performance of stencil computations. Then, we present spatial and temporal tiling optimizations and a novel pipelined distributed stencil algorithm that combines the advantages of popular and well-studied tiling optimizations to target multiple connected iteration spaces.

A. Distributed Stencils

The classical distributed stencil computation follows Algorithm 1, where communication is performed at each iteration t to exchange the halos of grid blocks between processes with data dependence. The iterative algorithm executes for $nIter$ steps until the user-defined convergence constraints are satisfied. Instead of using MPI datatypes to define the halos as several sub-arrays, we pack and unpack the halos explicitly to a 1D buffer which creates opportunities for data locality optimizations.

Algorithm 1 Classical distributed iterative stencil algorithm

```
for $t$ = 1 $\rightarrow$ nIter do
    compute( )
    pack_halo_to_buffer( )
    exchange_halo( )
    unpack_buffer_to_halo( )
```

B. Programming Models

The two popular choices for implementing Algorithm 1 is the traditional *flat MPI* model (also known as *MPI everywhere*) and a hybrid model using *MPI+threads*.

- *Flat MPI.* Each process is mapped to a core and its communication consists of both intra- and inter-node communications. Intra-node communication is optimized using *MPI shared memory* [46].
- *MPI+threads.* Typically, one MPI process is assigned to a node or socket with one thread per core. We choose the `MPI_THREAD_FUNNELED` mode where each MPI

process launches multiple threads but only one thread is responsible for calling MPI communication routines[1].

On a single node, the stencil computation using flat MPI follows the same workflow as the distributed Algorithm 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 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 IV-A. With flat MPI, 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 2 shows an example with a domain of size $2 \times N_j \times N_k$ distributed among 2 processes (P0, P1) versus 2 threads (T0, T1). Each thread has a $j-k$ face that is twice 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.

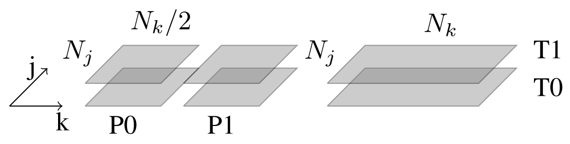


Fig. 2: A 2 plane domain divided among 2 processes (MPI) or 2 threads (OpenMP). Processes P0 and P1 have a $j-k$ plane of size $N_j \cdot N_k/2$. Threads T0 and T1 have a $j-k$ plane of size $N_j \cdot N_k$.

Moreover, threads[2] and processes have different synchronization patterns. Using OpenMP requires a global barrier at the end of each computation iteration to avoid data race. Flat MPI has no such safety guards and processes only wait for their data-dependent neighbors in communication completion routines like `MPI_Waitall`. In Algorithm 1, 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 Gold processors, we observe that the socket's bandwidth is saturated by just 4 and 10 cores respectively. In Section IV-A, 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.

[1]`MPI_THREAD_MULTIPLE` is also frequently referred to in literature which allows each thread to call MPI communication routines concurrently. However, current MPI implementations only create a single network endpoint per process which serializes the threads' communication [47]. Therefore, it is not expected to perform better than `MPI_THREAD_FUNNELED` on state-of-the-art MPI implementations.

[2]We will assume OpenMP threads for the rest of this paper.

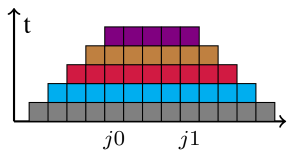
C. Stencil Optimizations

a) ***Cache Tiling:*** In 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. Blocking the computation of a 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 provided the range is carefully chosen. 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 dimension of k 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 performance, several studies have considered tiling in both space and time where the latter is commonly referred to as *temporal tiling*.

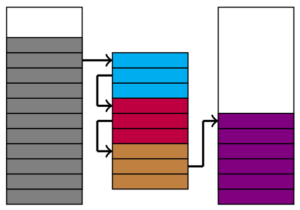
The idea of temporal tiling is to fuse multiple iterations of a patch while it still resides in cache. An efficient way to implement temporal tiling for stencils is to march along the i^{th} dimension and repeatedly use the most recently updated $j-k$ patch to update the patch beneath it. This is referred to as *wavefront blocking* [3], [7], [32]. The following code snippet 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.

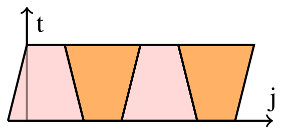
```
1 for (int t=1; t<nIter; t+=f)
2   for (int i=0; i<N_i; ++i)
3     for (int tt=0; tt<f; ++tt) {
4       int p = f-1-tt;
5       for (int j=j0-p; j<=j1+p; ++j)
6         for (int k=0; k<N_k; ++k)
7           compute(t+tt, i-tt, j, k)
8     }
```

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 3a illustrates this effect in time and space dimensions with four iterations fused in the order of ⬛ → 🟦 → 🟥 → 🟫 → 🟪. We denote the trapezoid formed by the patch and the fused iterations as a *time-space tile*. The cells outside the tile's own patch, i.e. cells outside the range $[j0, j1]$ in Figure 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) [1]–[4], where the tiles overlap each other as shown in Figure 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) [5], [6], diamond tiling [7]–[10], and tessellating tiling [11], [12] follow this strategy.

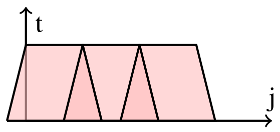


(a) Time-Space Tile


(b) OT Data Flow



(c) TT Time-Space



(d) OT Time-Space

Fig. 3: Data flow and time-space shape for Overlapped Tiling (OT) and Trapezoidal Tiling (TT).

Figure 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 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 are 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 [3], [32], [48] 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,

$$\begin{aligned} &((s_j + 2h_o)(s_k + 2h_o) - s_j s_k) \cdot \frac{N_j}{s_j} \cdot \frac{N_k}{s_k} \\ &= N_j N_k \left(2h_o \frac{s_j + s_k}{s_j s_k} + 4\frac{h_o^2}{s_j s_k}\right) \end{aligned} \quad (1)$$

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 3c. Tiles of the same type can be executed in parallel. The upward tiles (pink) are identical to the time-space tile in Figure 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 [49] or dynamic tasking based on the tiles' dependencies [17]. 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 and numerical schemes involving multiple variables. The disadvantage is that intermediate results are written to DRAM and the space updated by both trapeziums is fetched from DRAM twice. As a result, we expect OT to exhibit better performance for smaller stencils and lightweight numerical schemes due to less DRAM traffic while TT might perform better on larger stencils and systems with smaller caches.

*b) **Multiple Connected Iteration Spaces and DeepHalo**:* The previously discussed tiling algorithms and state-of-the-art polyhedral auto parallelization tools and compilers such as Pluto [15], [16] and Pochoir [17] specialize at optimizing a single iteration space which corresponds to a single-block structured grid in CFD. The iteration space is typically composed of an outer loop for time and 3 nested inner loops for the 3 spatial dimensions. In real applications, the common case is multi-block structured grids composed of *multiple connected iteration spaces*.

```
for (int t=1; t<nIter; ++t) {
  for (int block=1; block<nBlocks; ++block) {
    get_block_range(block, range);
    for (int i=range[0]; i<range[3]; ++i)
      for (int j=range[1]; j<range[4]; ++j)
        for (int k=range[2]; k<range[5]; ++k)
          compute(block, i, j, k);
  }
  for (int block; block<nBlocks; ++block)
    exchange_block_boundary(block);
}
```

The above code snippet demonstrates the nested loops for solving multi-block grids where each block forms a single iteration space at lines 4-6. Blocks can be connected and their boundaries' data are exchanged by the function `exchange_block_boundary` at line 10. This exchange happens at every timestep and prevents state-of-the-art polyhedral techniques to directly tile the time and space loop together for each block.

We propose to first tile the outer temporal loop by f so that each block has an iteration space composed of f temporal iterations and 3 space dimensions. 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. As a result, we have broken down the multiple connected iteration spaces to multiple single iterations spaces, 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*.

The scenario of multiple connected iteration spaces 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 [30], [32]. However, the performance of Deephalo is still network-specific on modern systems.

*c) **Overlap of Computation and Communication**:* 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 [33]–[36] 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:

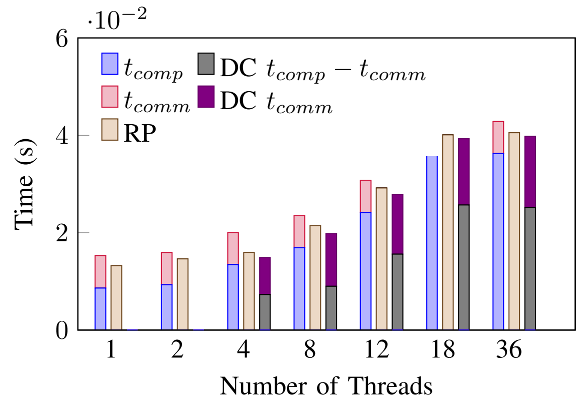
- *DedicatedCore (DC).* Dedicate one core for communication while the other cores perform computation.
- *RepeatedPoll (RP).* Repeatedly call functions such as `MPI_Test` during computation to urge the underlying network software to make progress on communication.

For flat MPI, it's non-trivial to implement DedicatedCore and we refer readers to Casper [33], [38], [50]. 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:

$$\eta = \frac{t_{comp} + t_{comm} - t_{ovlp}}{\min(t_{comp}, t_{comm})} \tag{2}$$

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.



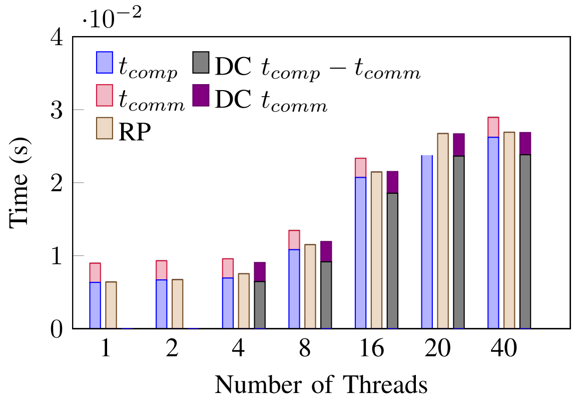
(a) Performance of RepeatedPoll and DedicatedCore on Bebop

# 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

Fig. 4: Computation and communication overlap with DedicatedCore and RepeatedPoll on Bebop.

We benchmark the overlap on two clusters – Bebop and HPC3 – summarized in Table II with Omni-Path and InfiniBand networks. The results are presented in Figures 4 and 5 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 $t_{comp} - t_{comm}$). On Bebop, RP can achieve up to 40% of the ideal overlap. DC performs slightly better 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.



(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

Fig. 5: Computation and communication overlap with DedicatedCore and RepeatedPoll on HPC3.

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

To realize overlap irrespective of which method (DC or RP) is used, the computation must not have data dependence on the overlapped communication. Therefore, the domain has to be decomposed into a communication dependent part and an independent part. The classic decomposition is to divide the domain into an outer layer and an inner chunk whose update does not depend 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 [27].

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 [45]. 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 6 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.

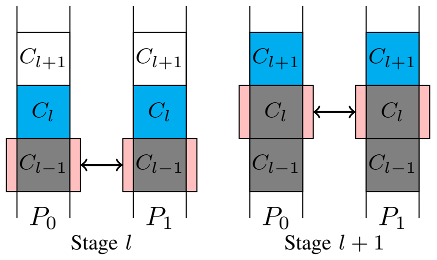


Fig. 6: Cut domain into chunks and pipeline communication and computation for overlap.

III. Experimental Setup

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

A. Case study

To systematically evaluate our proposed algorithms, we choose 4 stencils across 6 numerical schemes whose characteristics are summarized in Table I. The stencils have different shapes (illustrated in Figure 1) and radius. The numerical schemes have various number of input/output variables, and

TABLE I: Stencils and Numerical Schemes.

Test	# Pts	Shape	r_s	Scheme	AI (NT)	AI	# Inputs	# Outputs
WJ7	7	Star	1	Weighted Jacobi 7pt	0.42	0.31	2	1
WJ13	13	Star	2	Weighted Jacobi 13pt	0.67	0.5	2	1
WJ27	27	Box	1	Weighted Jacobi 27pt	1.25	0.94	2	1
Weno3	13	Star	2	3rd order WENO	1.96	1.64	4	1
Upwind	13	Star	2	2nd order Upwind	0.85	0.71	4	1
Burgers	24	Staggered	1	Central Difference	2.50	1.67	3	3

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 stencils test cases.

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

$$\nabla^2 p = b \tag{3}$$

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 stencils with 27 points. Equation 3 is solved with the weighted Jacobi methods, which is one of the standard smoothers for multigrid [51]. Jacobi methods are the most widely studied stencil computations in the space and temporal tiling body of research [2], [3], [5]–[12], [15]–[17], [52] 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 4 simulates the convection phenomena in fluid dynamics, in which ψ denotes a scalar propagated by the velocity field $\vec{u}$.

$$\partial_t \psi + \vec{u} \cdot \nabla \psi = 0 \tag{4}$$

Here we consider the memory-bound 2nd order upwind and the 3rd order WENO schemes (which have higher flops per grid cell) [53] on a star stencil with 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,

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

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

- **3D Burgers equation.** Equation 5 represents the conservative form of Burgers equation, which is solved in simulating incompressible flows ($\nabla \cdot \vec{u} = 0$).

$$\partial_t \vec{u} + \nabla \cdot (\vec{u}\vec{u}) = \nu \nabla^2 \vec{u} \tag{5}$$

The three components of velocity $\vec{u}$ (the 3 inputs) are discretized on grid cell's face center using a staggered stencil. Figure 7 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 IV-A, the coupled dependency between components and the large data volume is highly challenging for the tiling algorithms.

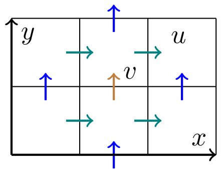


Fig. 7: 2D Staggered.

B. 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 II. We choose a domain 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.

IV. Results and Discussion

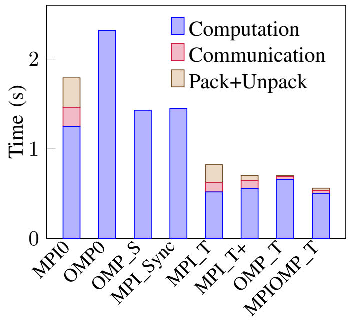
In this section, we first present the single-node performance breakdown with spatial and temporal tiling and compare the performance of our hybrid decomposition against the state-of-the-art polyhedral tiling tool, Pluto[15], [16]. We also compare

TABLE II: Evaluation platforms and their parameters.

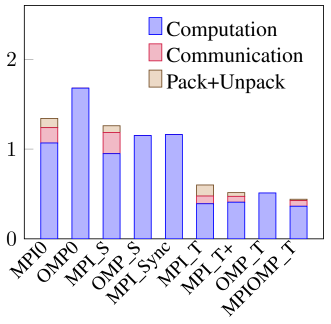
	Bebop	HPC3
Architecture	Intel Xeon E5-2695v4 (Broadwell)	Intel Xeon 6248 (Gold)
CPU Frequency	2.4 GHz	2.5 GHz
Sockets	2	2
Cores/Socket	18	20
GFlops/s (DP)	1200	2207
L2 cache	32 KB	1024 KB
L3 cache	90 MB	55 MB
DRAM Bandwidth	120.3 GB/s	194.4 GB/s
Network	Omni-Path	InfiniBand
Nodes	653	38
Compiler	Intel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

the two temporal tiling algorithms – overlapped and trapezoidal tiling and present an analysis of scenarios where one outperforms the other. Then, we evaluate the *performance and scalability* of Pencil on two distributed memory clusters up to 128 nodes. Finally, we apply Pencil to multiple connected iteration spaces distributed on 32 nodes which represents practical CFD problems of interest with multi-block grids.

A. Single Node Performance - MPI/OpenMP



(a) WJ7 on Broadwell



(b) WJ7 on Gold

Fig. 8: Performance of spatial and temporal tiling with WJ7 on a single node.

Using the WJ7 stencil on Broadwell as an example, we describe how to choose the optimum combination of MPI processors and OpenMP threads. The MPI baseline (*MPI0*) decomposes each dimension of the grid evenly to reduce intra-node communication volume and achieve load balance. In our experiments, a decomposition of 4 x 3 x 3 in the i^{th}, j^{th}, k^{th} dimensions delivers the best baseline performance. Unlike MPI0, the OpenMP baseline (*OMP0*), splits the iteration space only in the i^{th} dimension to ensure threads access contiguous data as discussed in Section II-B. As seen from Figure 8a, even though communication and buffer preparation (i.e. pack and unpack) take a considerable time, MPI0 still outperforms OMP0 by 30% on Broadwell. This is because OMP0 has a $j-k$ plane that is 9× larger than MPI0 and spills out of the cache. To remedy this, we add space tiling (denoted by *OMP_S*) in the j^{th} dimension, which has no effect on MPI0 on Broadwell (therefore not shown in Figure 8a) but improves the OpenMP baseline by 1.6×. Note that the computation in MPI0 still takes less time than OMP_S. This is because of the competition for bandwidth (described in Section II-B) which can be validated by enclosing the computation in MPI0 with `MPI_Barrier`. This implementation denoted by *MPI_Sync* leads to similar computation performance as OMP_S. Moreover, with space tiling, the measured arithmetic intensity (AI) in Table III matches our theoretical estimate in Table I 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 III) 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* space 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 8a, *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 in flat MPI can take longer than the communication time. Copying the halo attached to the k boundaries ($i-j$ planes) is particularly expensive due to the short length of the contiguous data segment. We work around this issue by merging the packing and unpacking of the k boundaries into the computation as demonstrated by the following code snippet:

```
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);
```

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.

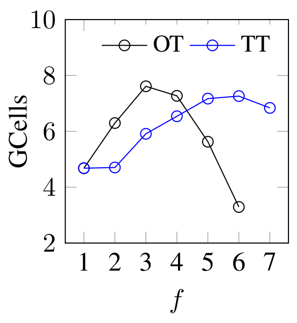
For temporal tiling implemented with OpenMP, we let OT and TT decompose the iteration space in the j^{th} dimension and leave the k^{th} dimension unsplit for prefetching and SIMD. In Figure 8a, *OMP_T* represents the best performance obtained with OpenMP and temporal tiling. Despite the similar overall 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 less $j-k$ 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 iteration space 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 8a 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.

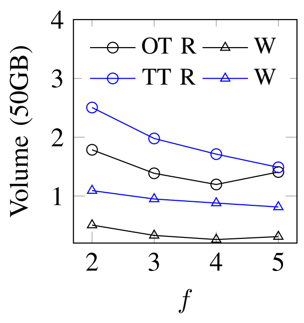
Comparison with the state-of-the-art: Pluto. We now compare our hybrid algorithm with the polyhedral tiling tool Pluto [15], [16], which generates codes using diamond tiling. Following the guidelines in [10], we set a large tile size for the k^{th} dimension and perform an exhaustive search for the optimal tile sizes of the other dimensions. Table III 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 $\mathcal{O}(10^3) \sim \mathcal{O}(10^4)$ 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 space tiling rather than the baseline. If a significant speedup is not observed with temporal tiling, then space tiling might be favorable for fewer code modifications. As shown in Table III, Pluto's temporal tiling for Upwind and WENO3 are not effective since their performance are emulated by space tiling alone on both Broadwell and Gold systems. 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.

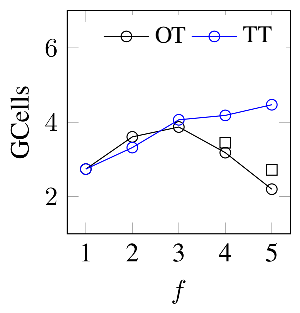
B. Single Node Performance - OT/TT



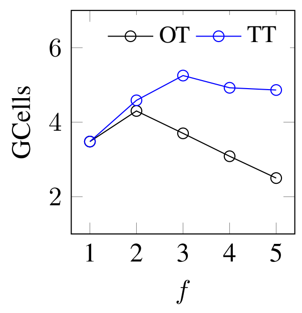
(a) WJ13 on Broadwell



(b) Upwind on Broadwell



(c) Upwind on Broadwell



(d) WJ13 on Gold

Fig. 9: Comparison of Overlapped and Trapezoidal tiling with increasing number of fused iterations (f).

Figure 9 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 exhibits different performance behavior with OT and TT. Each curve shows tiling with a fixed tile size. For 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 is 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 II-C.

Column *MPIOMP_T* in Table III 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 9a 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 shown in Figure 9b, at $f=3$, OT's reading and writing volumes is only 34% and 70% of TT. As f increases, OT

TABLE III: Summary of single-node performance of the stencils in Table I on the two systems in Table II. 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 space tiling, and Pluto respectively.

Test	Broadwell					Gold				
	AI	MPIOMP_T	vs OMP0	vs OMP_S	vs Pluto	AI	MPIOMP_T	vs OMP0	vs OMP_S	vs Pluto
WJ7	0.42	TT 10; OT 6	4.63×	2.83×	0.98×	0.29	TT 10; OT 4	4.76×	3.29×	1.15×
WJ13	0.69	OT 3; TT 6	3.46×	1.63×	0.98×	0.48	TT 5;	3.52×	1.89×	1.29×
WJ27	1.28	TT 8	2.68×	1.69×	0.91×	0.94	TT 5	2.06×	1.46×	1.10×
Upwind	0.87	TT 5	2.86×	1.67×	1.92×	0.71	TT 3	2.72×	1.57×	1.74×
Weno3	1.71	TT 3	2.06×	1.47×	1.49×	2.40	TT 2	1.59×	1.09×	1.08×
Burgers	1.63	OT 2	2.42×	1.58×	4.63×	1.52	OT 2; TT4	1.87×	1.25×	5.03×

introduces too much redundant computation and the local array starts to spill from the L3 cache, which cause the performance to drop drastically. TT has the advantage of using small tiles without introducing redundant computation (Section II-C) 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 9c 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 9c. 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 less 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 9d, and achieves similar performance for Burgers.

C. Communication with DeepHalo and Pipelined Overlap

We compare Pencil over flat MPI and MPI+OpenMP Funneled on 128 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 space tiling to both. For the hybrid temporal tiling, we map one MPI rank per socket with one OpenMP thread per core.

Table IV presents the measured effective overlap ratio, η, and speedups over flat MPI and MPI+OpenMP Funneled for the 6 case studies. We observe that the optimal choice of 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 space tiling, the overlapped algorithm improves the performance by $1.39 - 2.77\times$ on Bebop and up to $3.36\times$ on HPC3, which validates the effectiveness of Pencil on distributed systems.

D. Weak and Strong Scalability

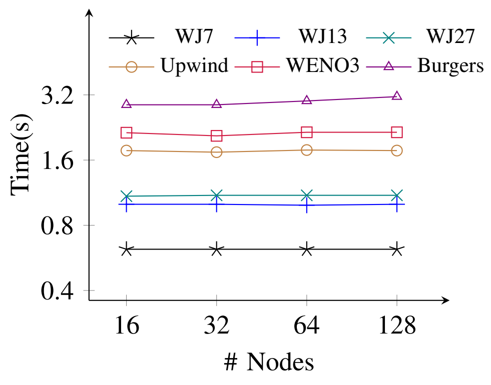
To evaluate the weak and strong scalability of Pencil, we choose Bebop since it has a larger number of nodes. For weak scalability, we maintain the same block sizes and connections per node as in Section IV-C. Pencil exhibits excellent weak scalability up to 128 nodes or 4608 cores across the board for all 6 case studies including Upwind, WENO3, and Burgers as shown in Figure 10a.

The strong scalability results are reported in Figure 10b. 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 (proportional to the size of the block) and the communication volume (proportional to the surface area of a block) is balanced in all the tests.

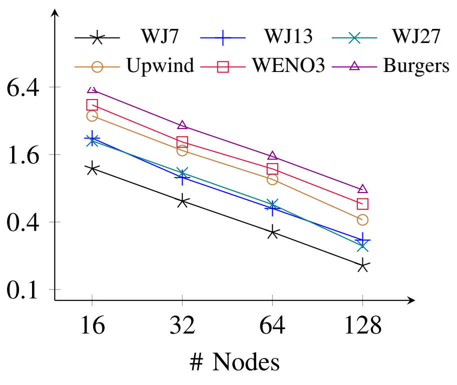
Pencil exhibits near-linear scaling across all the 6 numerical schemes on 4 different stencils. However, the scaling of the non-overlapped temporal tiling algorithm is less efficient because the communication does not scale linearly with the number of nodes. This phenomena is highlighted in Figure 10c.

TABLE IV: Summary of the performance of six stencils in Table I on 128 nodes of Bebop and 32 nodes of HPC3 summarized in Table II. 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 flat MPI and MPI+OpenMP Funneled with space tiling.

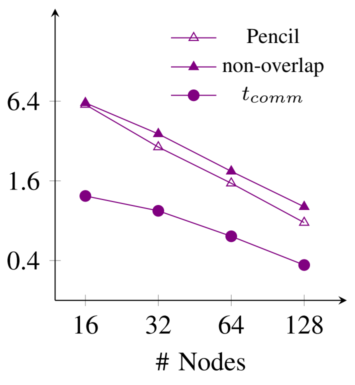
Test	Broadwell					Gold				
	MPIOMP_T	η_{RP}	η_{DC}	vs non-ovlp	vs baseline	MPIOMP_T	η_{RP}	η_{DC}	vs non-ovlp	vs baseline
WJ7	OT 5	0.72	0.90	1.48×	2.77×	TT 10	0.77	0.50	1.20×	3.36×
WJ13	OT 3	0.69	0.92	1.49×	1.97×	TT 5	0.75	0.61	1.21×	2.19×
WJ27	TT 8	0.73	0.60	1.20×	1.61×	TT 6	0.93	0.59	1.11×	1.86×
Upwind	TT 4	0.69	0.57	1.24×	1.68×	TT 3	0.66	0.55	1.26×	1.84×
Weno3	TT 3	0.69	0.70	1.27×	1.39×	TT 3	0.50	0.40	1.08×	1.27×
Burgers	OT 2	0.78	0.82	1.24×	1.69×	OT 2	0.56	0.48	1.08×	1.54×



(a) Weak scalability



(b) Strong Scalability

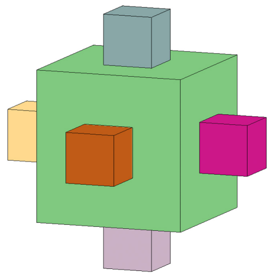


(c) Strong Scaling for Burgers

Fig. 10: 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 $1920 \times 1920 \times 960$.

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 10c 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.

E. Multiple Connected Iteration Space


(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

Fig. 11: Geometry and sizes of the multi-block mesh with 6 blocks that are connected at 5 faces.

To apply Pencil to multiple connected iteration spaces, 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. The hybrid tiling and the pipelined overlap algorithm do not require any change. To demonstrate this capability, we apply it to a multi-block grid illustrated in Figure 11a where 5 small blocks are connected to 5 different boundaries of a large block. This geometry emulates a multi-exit pipe transition in engineering.

TABLE V: Summary of performance on the multi-block mesh. 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×

The gird blocks are partitioned to 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. Table V presents the performance results of the 6 test cases on the multi-block mesh on 32-nodes of Bebop. We achieve 1.33-3.41$\times$ speedup over MPI+OpenMP Funneled with space tiling.

V. Related Work

The term *pipelined stencil* or *stencil pipelines* has taken different definitions in various studies aimed at vastly different tasks. In image processing, stencil pipeline refers to multiple stencil computation stages. Different stages can have completely different stencils and grids in contrast to CFD applications where fixed stencils are used on one grid over several temporal iterations. Domain Specific Languages (DSLs) and compliers such as PolyMage[18] and Halide [19] embed cache tiling to optimize the stencil pipelines but are still far from delivering performance that are on-par with hand-tuned codes for real CFD applications [54]. For stencil studies on FPGA or other custom architectures [55], 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 [55]–[61]. 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 [62], [63] 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 3, 4, and 5 solved in this study.

Several studies have applied temporal tiling techniques to distributed computation. The Geometric Multi-Grid Solver developed in [32], [64] combines DeepHalo with the overlapped tiling [3] and demonstrates significant speedups for solving the Helmholtz equation in a 3D box. In [45], 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 [7] 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 [65] and the Oxford Parallel library for Structured meshes (OPS) [66], [67] 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 [5], [6] or diamond tiling [7], [8].

Pluto [15], [16] represents the start-of-the-art in automatic parallelization tools using diamond tiling with polyhedral techniques [9], [10]. First. it decomposes the iteration space into tiles, then distributes the tiles among processes, and finally generates the corresponding MPI routines [68]. 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 [42]. However, the application is still limited to a single iteration space 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 multiple connected iteration spaces. Therefore, the present study contributes to the state-of-the-art in distributed stencil computation for multiple connected iteration space representative of real CFD applications.

VI. Conclusions

We propose Pencil, a novel algorithm to extend temporal tiling to multiple connected iteration spaces 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. We evaluate Pencil on 4 different stencils across 6 numerical schemes with arithmetic intensity from 0.4 to 2.5, including practical and complex schemes like WENO3 and Burgers on a staggered grid. Pencil's hybrid tiling outperforms the start-of-the-art tool based on the polyhedral model, Pluto [15], [16] on a single node by up to 1.9$\times$.

On distributed systems, Pencil achieves overlap by pipelining the computation and communication along the least rapidly changing dimension and significantly outperforms flat MPI and MPI + OpenMP Funneled on both clusters considered in this study. Pencil decouples the multiple connected iteration spaces with DeepHalo and performs temporal tiling on the individual iteration spaces. Moreover, it exhibits excellent weak and strong scaling up to 128 nodes for all 6 case studies. Applied to a multi-block grid with 6 connected blocks, Pencil demonstrates a 1.33-3.41$\times$ speedup over MPI+OpenMP Funneled with space tiling. Looking forward, we anticipate this study will open new research directions into incorporating Pencil in auto-parallelizing tools and compilers.

Acknowledgments

This work is supported by the National Science Foundation (NSF) under the award number 1750549. We greatly acknowledge the computing resources provided and operated by the Laboratory Computing Resource Center (LCRC) at the Argonne National Lab (ANL) and the Research Cyber-Infrastructure Center (RCIC) at the University of California Irvine (UCI). We thank Rohit Zambre from our research lab at UCI for his timely assistance in collecting results and Uday Bondhugula at the Indian Institute of Science for his assistance in using Pluto.

References

[1] 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*, pp. 246–253.

[2] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 235–244, 2007.

[3] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. 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*.

[4] B. Mostafazadeh, F. Marti, B. Pourghassemi, F. Liu, and A. Chandramowlishwaran, "Unsteady Navier-Stokes computations on GPU architectures," in *23rd AIAA Computational Fluid Dynamics Conference, 2017*.

[5] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," *Proceedings of the International Conference on Supercomputing*, vol. 1, no. 212, pp. 361–366, 2005.

[6] ——, "The cache complexity of multithreaded cache oblivious algorithms," *Theory of Computing Systems*, vol. 45, no. 2, pp. 203–233, 2009.

[7] R. Strzodka, M. Shaheen, D. Pajak, and H. P. Seidel, "Cache accurate time skewing in iterative stencil computations," *Proceedings of the International Conference on Parallel Processing*, pp. 571–581, 2011.

[8] D. Orozco and G. Gao, "Mapping the FDTD application to many-core chip architectures," *Proceedings of the International Conference on Parallel Processing*, pp. 309–316, 2009.

[9] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2012*, pp. 1–11.

[10] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1285–1298, 2017.

[11] L. Yuan, Y. Zhang, P. Guo, and S. Huang, "Tessellating stencils," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*.

[12] L. Yuan, S. Huang, Y. Zhang, and H. Cao, "Tessellating Star Stencils," *ACM International Conference Proceeding Series*, 2019.

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

[14] E. Hammami and Y. Slama, "An overview on loop tiling techniques for code generation," *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, vol. 2017-Octob, pp. 280–287, 2018.

[15] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 101–113, 2008.

[16] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International Conference on Compiler Construction*. Springer, 2008, pp. 132–146.

[17] Y. Tang, R. Chowdhury, C.-k. Luk, B. C. Kuszmaul, and C. E. Leiserson, "The Pochoir Stencil Compiler Categories and Subject Descriptors," *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 117–128, 2011.

[18] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic Optimization for Image Processing Pipelines," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 429–443, 2015.

[19] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide:A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[20] H. Wang and A. Chandramowlishwaran, "Multi-criteria partitioning of multi-block structured grids," *Proceedings of the International Conference on Supercomputing*, pp. 261–271, 2019.

[21] L. Shi, M. Rampp, B. Hof, and M. Avila, "A hybrid MPI-OpenMP parallel implementation for pseudospectral simulations with application to Taylor-Couette flow," *Computers and Fluids*, vol. 106, pp. 1–11, 2015.

[22] N. Drosinos and N. 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)*, vol. 18, no. C, pp. 193–202.

[23] H. Gahvari, M. Schulz, and U. M. Yang, "An approach to selecting thread + process mixes for hybrid MPI + OpenMP applications," *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, vol. 2015-Octob, pp. 418–427, 2015.

[24] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using MPI and OpenMP on multi-core parallel systems," *Parallel Computing*, vol. 37, no. 9, pp. 562–575, 2011.

[25] J. M. Bull, J. Enright, X. Guo, C. Maynard, and F. Reid, "Performance evaluation of mixed-mode OpenMP/MPI implementations," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 396–417, 2010.

[26] M. J. Chorley and D. W. Walker, "Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters," *Journal of Computational Science*, vol. 1, no. 3, pp. 168–174, 2010.

[27] R. Rabenseifner, G. Hager, and G. 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*, no. c, pp. 427–436, 2009.

[28] F. Bassetti, K. Davis, and D. Quinlan, "Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures," in *International Symposium on Computing in Object-Oriented Parallel Environments*. Springer, 1998, pp. 107–118.

[29] A. Sawdey and M. O'Keefe, "Program analysis of overlap area usage in self-similar parallel programs," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1997, pp. 79–93.

[30] C. Ding and Y. He, "A ghost cell expansion method for reducing communications in solving pde problems," in *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. IEEE, 2001, pp. 55–55.

[31] F. B. Kjolstad and M. Snir, "Ghost cell pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, 2010, pp. 1–9.

[32] P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker, "Compiler generation and autotuning of communication-avoiding operators for geometric multigrid," *20th Annual International Conference on High Performance Computing, HiPC 2013*, pp. 452–461, 2013.

[33] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures," *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2015*, pp. 665–676.

[34] G. Schubert, H. Fehske, G. Hager, and G. Wellein, "Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems," *Parallel Processing Letters*, vol. 21, no. 03, pp. 339–358, 2011.

[35] A. Denis and F. Trahay, "MPI overlap: Benchmark and analysis," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 258–267.

[36] H. S. B, S. Chakraborty, and D. K. Panda, "Designing Dynamic and Adaptive MPI Point-to-Point Communication Protocols for Efficient Overlap of Computation and Communication," vol. 10524, pp. 334–354, 2017.

[37] Message Passing Interface Forum, "MPI: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 2015.

[38] M. Si and P. 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)*. IEEE, 2017, pp. 206–214.

[39] T. H. Kaiser and S. B. Baden, "Overlapping communication and computation with OpenMP and MPI," *Scientific Programming*, vol. 9, no. 2-3, pp. 73–81, 2001.

[40] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping Communication and Computation by Using a Hybrid MPI / SMPSs Approach," 2010.

[41] M. Jiayin, S. Bo, W. Yongwei, and Y. Guangwen, "Overlapping communication and computation in MPI by multithreading," *Proc. of Inter-*

national Conference on Parallel and Distributed Processing Techniques and Applications, no. February, pp. 2–7, 2006.

[42] Y. Barigou and E. Gabriel, "Maximizing Communication–Computation Overlap Through Automatic Parallelization and Run-time Tuning of Non-blocking Collective Operations," *International Journal of Parallel Programming*, vol. 45, no. 6, pp. 1390–1416, 2017.

[43] P. R. Eller, T. Hoefler, and W. Gropp, "Using performance models to understand scalable Krylov solver performance at scale for structured grid problems," *Proceedings of the International Conference on Supercomputing*, pp. 138–149, 2019.

[44] N. Li and S. Laizet, "2DECOMP & FFT-A Highly Scalable 2D Decomposition Library and FFT Interface," *Cray User Group 2010 conference*, pp. 1–13, 2010.

[45] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, "Multicore-optimized wavefront diamond blocking for optimizing stencil updates," *SIAM Journal on Scientific Computing*, vol. 37, no. 4, pp. C439–C464, 2015.

[46] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "MPI+ MPI: A new hybrid approach to parallel programming with MPI plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[47] R. Zambre, A. Chandramowlishwaran, and P. Balaji, "Scalable communication endpoints for mpi + threads applications," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 803–812.

[48] P. Basu, "Compiler optimizations and autotuning for stencils and geometric multigrid," Ph.D. dissertation, The University of Utah, 2016.

[49] T. Henretty, R. Veras, F. Franchetti, L. N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector SIMD architectures," *Proceedings of the International Conference on Supercomputing*, pp. 13–24, 2013.

[50] M. Si, A. J. Pena, J. Hammond, P. Balaji, and Y. 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*, pp. 811–816.

[51] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial, Second Edition*, 2000.

[52] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N. Vasilache, "Tiling and optimizing time-iterated computations on periodic domains," *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pp. 39–50, 2014.

[53] X. D. Liu, "Weighted essentially non-oscillatory schemes," *Journal of Computational Physics*, vol. 115, no. 1, pp. 200–212, 1994.

[54] B. Mostafazadeh, F. Marti, F. Liu, and A. 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*, pp. 753–762.

[55] M. Christen, O. Schenk, P. Messmer, E. Neufeld, and H. 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)*, no. June 2014, pp. 47–54, 2008.

[56] W. Luzhou, K. Sano, and S. Yamamoto, "Domain-specific language and compiler for stencil computation on fpga-based systolic computational-memory array," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2012, pp. 26–39.

[57] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 695–705, 2014.

[58] K. Dohi, K. Okina, R. Soejima, Y. Shibata, and K. Oguri, "Performance modeling of stencil computing on a stream-based FPGA accelerator for efficient design space exploration," *IEICE Transactions on Information and Systems*, 2015.

[59] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "OpenCL-based FPGA-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1390–1402, 2017.

[60] H. R. Zohouri, A. Podobas, and S. 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*, vol. 2018-February, pp. 153–162, 2018.

[61] ——, "High-performance high-order stencil computation on fpgas using opencl," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 123–130.

[62] 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*, vol. 63, no. 11, pp. 1138–1151, 2003.

[63] G. Goumas, N. Anastopoulos, N. Koziris, and N. Ioannou, "Overlapping computation and communication in SMT clusters with commodity interconnects," *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, pp. 1–10, 2009.

[64] P. Basu, M. Hall, S. Williams, B. V. Straalen, L. Oliker, and P. Colella, "Compiler-Directed Transformation for Higher-Order Stencils," *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, pp. 313–323.

[65] T. Denniston, S. Kamil, and S. Amarasinghe, "Distributed halide," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, 2016.

[66] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The ops domain specific abstraction for multi-block structured grid computations," in *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. IEEE, 2014, pp. 58–67.

[67] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Loop tiling in large-scale stencil codes at run-time with OPS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 873–886, 2018.

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