The Swept Rule for Breaking the Latency Barrier in Time-Advancing PDEs

by

Maitham Makki Alhubail

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computational Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2019

Author .... Signature redacted ....
Department of Civil and Environmental Engineering
May 17, 2019

Certified by .... Signature redacted ....
John R. Willaims
Professor, Civil and Environmental Engineering
Thesis Supervisor

Certified by ..... Signature redacted ....
Qiqi Wang
Associate Professor, Aeronautics and Astronautics
Thesis Supervisor

Accepted by................ Signature redacted ....
Heidi Nepf
Donald and Martha Harleman Professor of Civil and Environmental Engineering
Signature redacted ~~Chair,~~ Graduate Program Committee

Accepted by. ....
Nicolas Hadjiconstantinou
Co-Director, Center for Computational Engineering

The Swept Rule for Breaking the Latency Barrier in Time-Advancing PDEs

by

Maitham Makki Alhubail

Submitted to the Department of Civil and Environmental Engineering
on May 17, 2019, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computational Science and Engineering

Abstract

This thesis describes a method to accelerate parallel, explicit time integration of unsteady PDEs. The method is motivated by our observation that network latency, not bandwidth or computing power, often limits how fast PDEs can be solved in parallel. The method is called the swept rule of space-time domain decomposition. Compared to conventional, space-only domain decomposition, it communicates similar amount of data, but in fewer messages. The swept rule achieves this by decomposing space and time among computing nodes in ways that exploit the domains of influence and the domain of dependency, making it possible to communicate once per many time steps with no redundant computation. By communicating less often, the swept rule effectively breaks the latency barrier, advancing on average more than one time step per round-trip latency of the network. The thesis describes the algorithms, presents simple theoretical analysis to the performance of the swept rule, and supports the analysis with numerical experiments.

Thesis Supervisor: John R. Willaims
Title: Professor, Civil and Environmental Engineering

Thesis Supervisor: Qiqi Wang
Title: Associate Professor, Aeronautics and Astronautics

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, extreme-scale computer clusters can solve PDEs using over 1,000,000 cores[5]. Exa-scale computing promises to enable 1,000,000,000-core simulations[1, 7]. From an application perspective and compared to desktop-based tools, using such computing power resembles upgrading from a slide rule to a desktop computer[14, Chapter 6].

1.1 Motivations

There is strong demand to use the increasingly parallel computing power to accelerate solutions of unsteady PDEs. In designing rocket launch vehicles[12], for example, unsteady flow equations are solved to reduce unsteady aerodynamic loads on the vehicle structure. In designing jet engine combustors[25], unsteady reacting flow equations are solved to help reduce emission and mitigate dangerous vibrations. In designing gas turbine blades[21, 15], unsteady aero and thermodynamic equations are solved to design cooling mechanisms for increased component life and overall engine efficiency. In all these applications, the unsteady equations need to be solved long enough to reproduce the chaotic, multiscale fluid dynamics. That requires at least hundreds of thousands, often millions of time steps. Advancing this many time steps with today's technology takes days, weeks, sometime months, running on as many parallel processors as these simulations can effectively use. The time-to-solution of these simulations often becomes the bottleneck of new product development and technological innovation. Engineers desire to further scale these simulations, to run them faster.

Today, the demand to accelerate solutions of unsteady PDEs is unmet even for small and long unsteady simulations. A simulation is small if the PDE is discretized spatially on a few million grid points. In three spatial dimensions, this means that each dimension may be discretized into just a few hundreds grid points. A simulation is long if the PDE need to be time integrated for millions of time steps.

Small and long simulations arrise, for example, in aero-thermal design of turbomachinary components. The simulation can be small when it is used to design a local feature of a component, e.g., the trailing edge of a turbine blade. It can nevertheless be long, because the slow thermodynamics requires a long simulation time span, and the fast, unsteady aerodynamics forces this long time span to be divided into millions of tiny time steps. Other simulations can be small and long if it resolves tightly coupled physical processes with similar spatial scales but disparate time scales. Such simulations are difficult to parallelize, and even more difficult to scale to many parallel processors.

1.2 Scalability of PDE Solvers

Scaling of a parallel PDE solver is always limited. When the solver runs on a few processors, doubling the processors can half the run time. As the number of processors further doubles, however, the percentage reduction in run time starts to diminish. As the number of processors reaches the scaling limit, adding more processors no longer reduces the run time.

What is preventing these simulations from scaling? If the total amount of computation required to solve a PDE is fixed, and we divide the computation among twice as many computing nodes, shouldn't each node work half as much, and the PDE solved in half the time? This would be true if each node does its own work without communicating with others. In solving PDEs, however, nodes communicate to each other frequently. Each communication takes at least a few microseconds, and on a common network, tens of microseconds. This minimum communication time, regardless of how much information is communicated, is called the network latency. If network latency exceeds the computing time between consecutive communications, reducing the computation of each node does not accelerate the simulation.

For a well-engineered parallel PDE solver, what limits the scaling is the communication between computing nodes. As the same computational task is subdivided into more computing nodes, the nodes communicate more frequently to exchange smaller batches of data. Exchanging smaller batches of data, however, is not always faster. Even the smallest batch of data takes a finite, predictable amount of time to be exchanged. That amount of time is the network latency. It is a fundamental cause to the scaling limit. This barrier to scaling, called the latency barrier, is necessarily encountered as a PDE-based simulation is deployed to more nodes.

1.3 Objective

This thesis focuses on extending the scaling limit by circumventing the network latency. It investigates the swept rule, an idea to break the latency barrier via communicating less often through exchanging fewer, larger batches of data between computing nodes. Because of the network latency, how long it takes to exchange a hundred or fewer bytes of data does not depend on the amount of data. On most networks, exchanging these bytes in a one shot is almost exactly twice as fast as exchanging them in two batches, if the first batch must finish before the second starts. Latency affect the speed of data exchange in batches up to about a megabyte. For less than a megabyte of data, exchanging it in a single batch is noticeably faster on most networks than exchanging it into multiple batches.

The swept rule decomposes space and time among computing nodes in ways that exploit the domains of influence and the domain of dependency, making it possible to communicate once per many time steps.Therefore, by communicating similar amount of data in fewer batches, the algorithms described in this thesis delay the effect of network latency, thereby extending the scaling limit of parallel PDE solvers. The resulting algorithms enable simulations to be solved significantly faster than what is possible with spatial domain decomposition schemes typically found in today's PDE solvers.

1.4 Existing Methods

The same objective is shared by parallel-in-time methods[17, 9, 8, 11, 10, 19]. These methods first estimate the solution using cheap but inaccurate time integrators, then iteratively correct the solution with an expensive and accurate time integrator. Parareal is one of the most studied parallel in time algorithms and was proposed by by Lions, Maday, Turinici in 2001[17]. One advantage of the Parareal method is that it allows the use of existing classical time-stepping routines and yet gain parallelism. It is also important to mention that, due to Parareal's popularity, a Parareal-like algorithm has been extended to chaotic dynamical systems[24]. Another famous parallel in time algorithm is the Multigrid Reduction in Time (MGRIT). One major difference between the MGRIT and Parareal is the use of multilevel time integrators[9]. The third famous parallel in time method is the Parallel Full Approximation Scheme in Space and Time (PFASST). In away, PFASST is similar to Parareal except that it uses advance iterative time stepping and full approximation schemes that allow for making the cheap inaccurate time integrators as cheap as possible[8]. These three parallel in time methods share being iterative and involving a coarse solve.

The method presented in this thesis, technically differ from most parallel-in-time methods, in that it is not iterative, and does not involve a coarse solver. Parallel-in-time algorithms iterate over the time domain. These methods work best with PDE systems that do not have extreme non-linearity in their dynamics. For example, parallel-in-time algorithms are less efficient when solving chaotic systems. The Swept Rule method advances in time and does not iterate over time. Therefore, it works equally well to different dynamical systems and different schemes. In this aspect, the Swept rule perhaps is inspired by the Communication Avoiding (CA) algorithms [4, 16] by Demmel et al. The Swept rule also shares some similarity with diamond tiling for cache optimization[18, 6, 22]. When it comes to solving a PDEs, CA algorithms are found to contribute and focus more on communication avoidance for the basic operations of linear algebra. Those algorithms include but are not limited to LU, QR, Matrix-Matrix multiplication and Matrix-Vector multiplication. CA algorithms cover a wide spectrum of techniques to achieve their goal in communicating less frequently starting from the different levels of memory hierarchy, to CPU-GPU communication and up to node to node communications. Cache oblivious programming and computation domain overlapping are just two examples of the techniques that CA algorithms use to minimize or avoid communication latency while performing a specific linear algebra task[4, 16].

Unlike the CA algorithms, however, most of the swept rule algorithms do not incur redundant computation. When it comes to our objective, that is breaking the latency barrier when explicitly solving PDEs, we see that swept decomposition is different from CA algorithms in that it does not involve any overhead of overlapping parts of the computation domain among processors, which is the case in some CA algorithms. In fact, the way swept is designed and implemented makes the effort of developing an explicit numerical PDE solver based on the swept decomposition similar to that effort involved in developing a classic-straight decomposition based solver.

Chapter 2

Space-time Decomposition of a PDE Solver

We restrict our attention to the following case. A PDE is discretized with a finite difference, finite volume, or finite element scheme. These discretization schemes generate a graph in the spatial domain. Each point in this graph represent a grid point in finite difference, a control volume in finite volume, or an element in finite element. An edge between two points exists if they are neighboring grids, control volumes or elements, as defined by the spatial discretization.

The swept rule described in this thesis operates under the assumption that the scheme for advancing each time step computes a few quantities on every grid point at the next time step, using the quantities on its neighbors at the current time step. Alternatively, a stencil operation that needs access beyond its nearest neighbors should be decomposed into sub-steps that accesses only the nearest neighbor.

2.1 Elementary Sub-timesteps

To analyze and break the latency barrier, we attempt to decompose the solution procedure into elementary steps that require communication between computing nodes. We call such an elementary step a sub-timestep. Each sub-timestep has a set of input variables and a

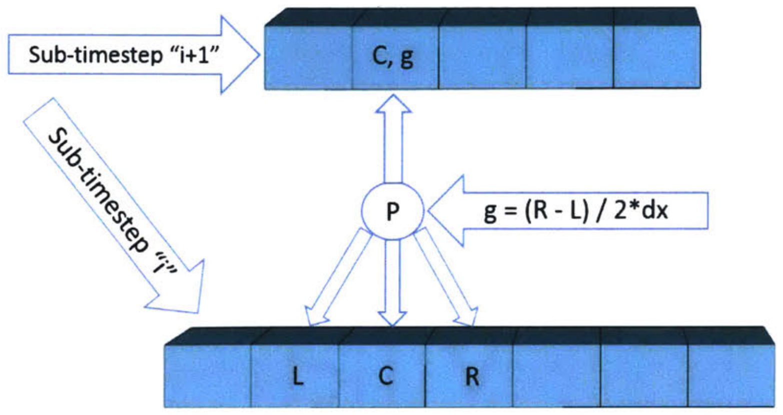


Figure 2-1: Illustration of Input, Output forward and variable calculations

set of output variables at each spatial point. From the input variables at each point and its neighbor points, the sub-timestep computes its output variables at that point and forwards any variables from the previous sub-timestep to the next sub-timestep if needed. Such variable forward thereby allows decomposition of a complex timestep into sub-timesteps, eliminating the need for neighbors of neighbors within a single sub-timestep. Figure 2-1 clarifies the above explanation. In the figure, the point "P" forwards the input of the sub-timestep below it to the sub-timestep above it as the first input. "P" also calculates the gradient using its left and right neighbors and sends that to the sub-timestep above it as the second input.

For example, finite difference gradient computation on a compact stencil is such a "sub-step". The inputs are the value of field variables; the outputs are the gradient of these variables. The following code exemplifies such a sub-timestep on a one-dimensional uniform grid. Here "p" denotes a spatial point; "p.Linputs" and "p.Rinputs" denote its left and right neighbors; "p.inputs[i]" and "p.outputs[i]" denote the "i" th input and output variable at the spatial point "p". Moreover, "p.Linputs[i]" and "p.Rinputs[i]" denote the "i" th inputs of the left and right neighbors respectively.

```
p.output[0] = p.inputs[0];
p.output[1] = (p.Rinputs[0] - p.Linputs[0]) / 2 * dx;
```

This sub-timestep has one input and two output variables. The first output variable is a carbon copy of the input; the second output variable is the finite difference derivative of the input.

Another example of a “substep" is flux computation and accumulation in finite volume. The inputs are the conservative variables and their derivatives; the outputs are the divergence of fluxes, or time derivatives of the conservative variables. The following code illustrates a substep that applies Godunov’s finite volume scheme to a scalar conservation law, and uses forward Euler for time integration.

```
fluxL = riemannSolver(p.Linputs[0] , p.inputs[0]);
fluxR = riemannSolver(p.inputs[0]  , p.Rinputs[0]);
p.outputs[0] = p.inputs[0] + dt * (fluxR - fluxL) / dx;
```

The input of this sub-timestep is the conservative variable at the *i*th time step, and the output is the conservative variable at the $i + 1$st time step.

If the forward Euler time integration is replaced by a multi-stage Runge-Kutta scheme, then each timestep require multiple sub-timesteps. In a two-stage Runge Kutta scheme, also known as the midpoint method, the first sub-timesteps can be

```
fluxL = riemannSolver(p.Linputs[0] , p.inputs[0]);
fluxR = riemannSolver(p.inputs[0]  , p.Rinputs[0]);
p.outputs(0) = p.inputs(0);
p.outputs(1) = p.inputs(0) + 0.5 * dt * (fluxR - fluxL) / dx;
```

It has a single input and two outputs, the first being a carbon copy of the input, and the second being the solution at the midpoint. These outputs become the inputs of the the second sub-timestep, whose single output is the solution at the next timestep:

```
fluxL = riemannSolver(p.Linputs[1] , p.inputs[1]);
fluxR = riemannSolver(p.inputs[1]  , p.Rinputs[1]);
outputs[0] = p.inputs[0] + dt * (fluxR - fluxL) / dx;
```

Every sub-timestep must use the same inputs as the outputs of the previous one. This means some sub-timesteps must “forward" some variables not involved in computation, by setting some of its outputs to the values of their corresponding inputs.

If we replace Godonov's scheme with a second order finite volume scheme, and use the forward Euler time discretization, then two sub-timesteps is required. The first sub-timestep can be similar to the finite difference derivative computation shown above, whose two outputs are the conservative variable itself and its derivative. The second sub-timestep, whose inputs are the outputs of the previous sub-timestep, can be encoded as following,

```
uL_minus = reconstructWithLimitor(p.Linputs[0],
                     p.inputs[0],p.Linputs[1], dx);
uL_plus  = reconstructWithLimitor(p.inputs[0],
                     p.Linputs[0], -p.inputs[1], dx);
fluxL    = riemannSolver(uL_minus, uL_plus);

uR_minus = reconstructWithLimitor(p.inputs[0],
                     p.Rinputs[0],     p.inputs[1], dx);
uR_plus = reconstructWithLimitor(p.Rinputs[0],
                    p.inputs[0], -p.Rinputs[1], dx);

fluxR = riemannSolver(uR_minus, uR_plus);
p.outputs(0) = p.inputs[0] + dt * (fluxR - fluxL) / dx;
```

Combining the second order finite volume scheme with a two-stage Runge-Kutta would result in two sub-timesteps per Runge-Kutta stage, thus four sub-timesteps per timestep. Solving a system of conservation laws, e.g., Euler equation, would require more inputs and outputs for each sub-timestep. But the total number of sub-timesteps would be no different from a scalar conservation law discretized with a similar scheme.

When such decomposition is possible, each sub-timestep can be viewed as an elementary operation in solving a PDE. The number of sub-timesteps involved in a calculation determines its domain of dependence and the domain of influence. Consider the outputs of the mth sub-timestep on the ith spatial point (grid point, cell or element). Its domain of dependence covers only its immediate neighbor points over one sub-timestep, i.e., among the inputs of the mth sub-timestep (or equivalently, the outputs of the $m-1$st sub-timestep).

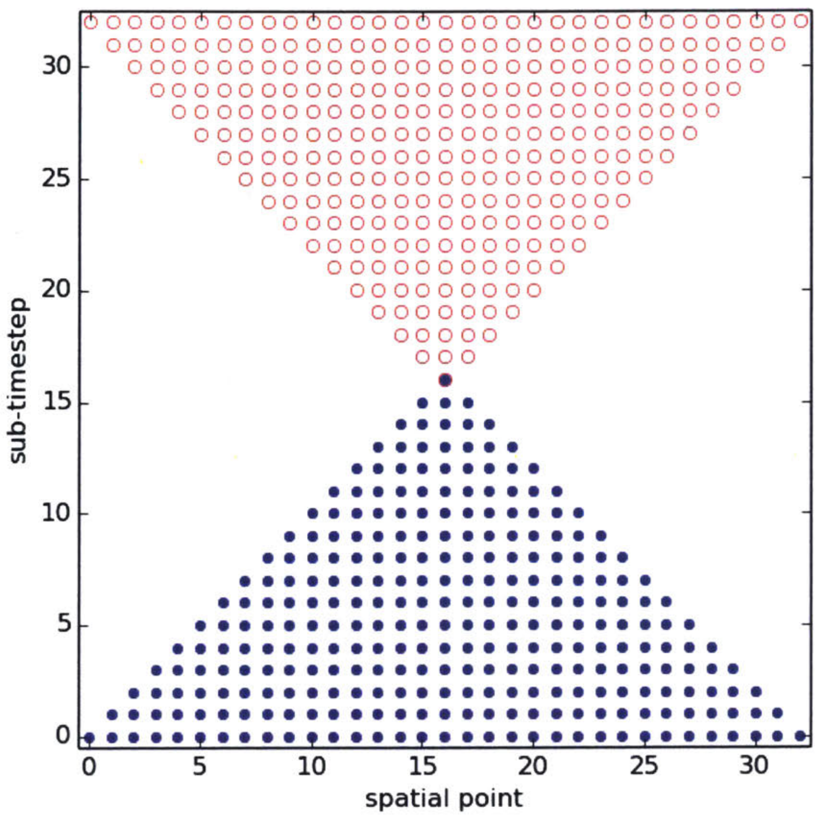


Figure 2-2: The domains of dependence and influence in a one-dimensional spatial domain

Over two sub-timesteps, the domain of dependence expands to the neighbors of neighbors. Over n sub-timesteps, it grows to all spatial points that connect to point i through n or fewer edges.

Figure 2-2 shows the domains of dependence in a one-dimensional spatial domain, as well as the domain of influence. The swept boundaries of these domains in the space-time diagram motivates the "swept" rule for solving PDEs. This rule decomposes along such space-time boundaries, and assigns the decomposed chunks of space-time among processors, thereby avoiding frequent latency-incurring communications.

2.2 More Advanced Example

Space-time decomposition is feasible for complex explicit schemes, and can be automated[23]. It eliminates the need of accessing the neighbors of neighbors. Here we illustrate such decomposition with an example. Consider the following scheme which requires two levels of neighbors.

$$u_{i,j}^{n+1} = u_{i-2,j}^{n} + u_{i+2,j}^{n} + u_{i,j-2}^{n} + u_{i,j+2}^{n} - 4u_{i,j}^{n} \tag{2.1}$$

It can be decomposed into 2 sub-steps. The first sub-step simply "pushes" its neighbor values to the second sub-step.

$$u_{i,j}^{n+\frac{1}{2}} = \begin{bmatrix} u_{i,j}^{n} \\ u_{i-1,j}^{n} \\ u_{i+1,j}^{n} \\ u_{i,j-1}^{n} \\ u_{i,j+1}^{n} \end{bmatrix} \tag{2.2}$$

The second sub-step has access to $u^{n+\frac{1}{2}}$ on the neighbors, which contains u^n on the neighbors of neighbors, which were "pushed" by the first sub-step. Using $u_{i,j;k}^{n+\frac{1}{2}}$ to denote the kth component ($k = 1, \ldots, 5$) of the vector $u_{i,j}^{n+\frac{1}{2}}$, the second sub-step should be

$$u_{i,j}^{n+1} = u_{i-1,j;2}^{n+\frac{1}{2}} + u_{i+1,j;3}^{n+\frac{1}{2}} + u_{i,j-1;4}^{n+\frac{1}{2}} + u_{i,j+1;5}^{n+\frac{1}{2}} - 4u_{i,j;1}^{n+\frac{1}{2}} \tag{2.3}$$

These 2 sub-steps (2.2-2.3) computes the same quantities as (2.1). But each sub-step only uses the immediate neighbors. Eliminating the need for further grid points can be achieved by decomposing into more sub-steps [23].

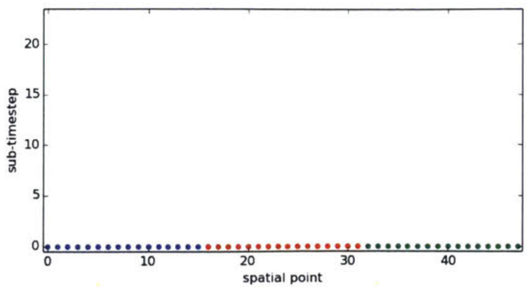
Chapter 3

The swept rule in 1D

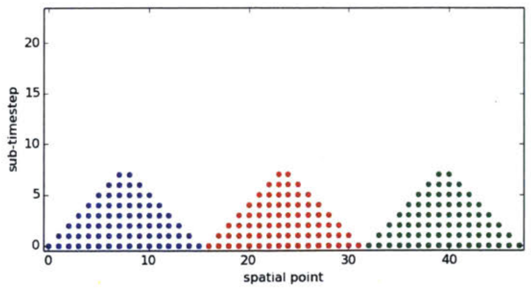
For illustration, we consider a one-dimensional spatial domain with periodic boundary condition. It is discretized into a series of spatial points, each representing a grid point in finite difference, a control volume in finite volume, or an element in finite element. Each point has two immediate neighbors, a left and a right one. When the PDE is initialized, the spatial points are decomposed uniformly among computing nodes, each node owning an even number of points. Each node starts by initializing the PDE on the points it owns.

This initial decomposition is illustrated in Figure 3-1(a). In these figures, a circle at horizontal coordinate i and vertical coordinate t represents the input variables of the tth sub-timestep at the ith spatial point, which for $t = 0$ are set by the initial condition of the PDE, and for $t > 0$ are the output variables of the $t - 1$st sub-timestep. At $t = 0$, the input variables of the first sub-timestep is colored by the computing node storing it. The variables at later sub-timesteps are greyed, indicating that they are not yet computed.

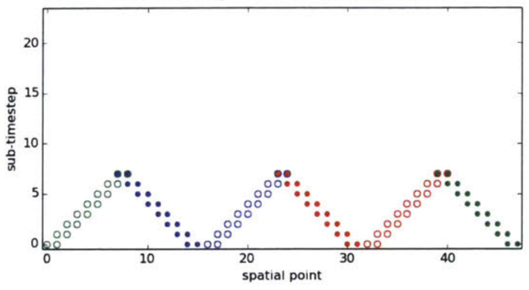
After setting the inputs of the first sub-timestep, we start the first local computation. During this stage, each parallel computing node computes all it can compute without communicating with other nodes. It computes all variables of which, at initialization, it owns the domain of dependence. These variables are colored accordingly in Figure 3-1(b). They include the output of the first sub-timestep on all but the two boundary points, the output of the second sub-timestep on all but four points, and the output of subsequent sub-timesteps



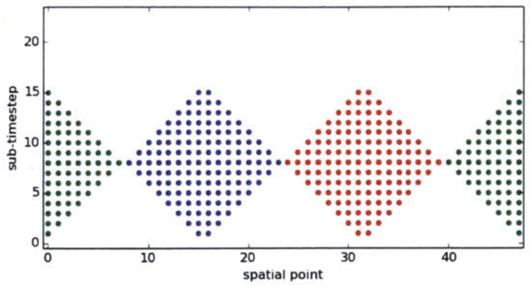
(a) Setting initial condition



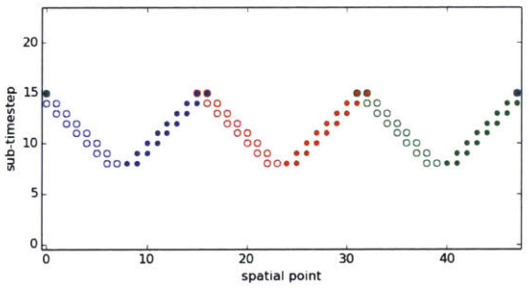
(b) 1st local computation



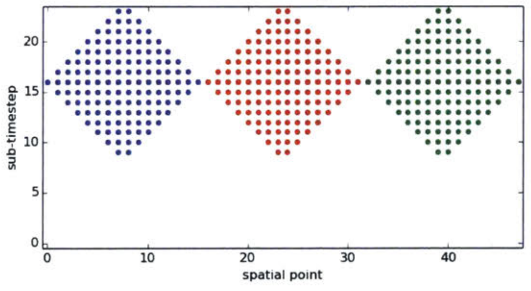
(c) 1st communication



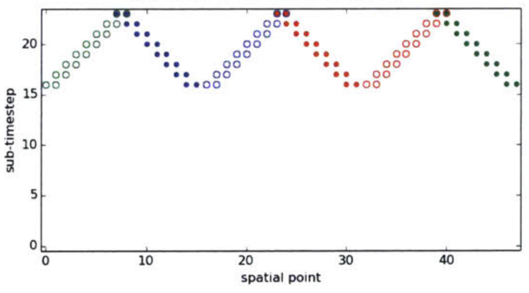
(d) 2nd local computation



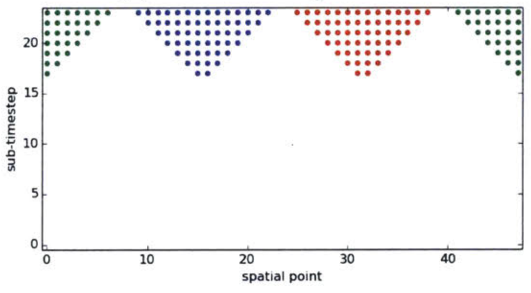
(e) 2nd communication



(f) 3rd local computation



(g) 3rd communication



(h) 4th local computation

Figure 3-1: The computing and communication stages for 1D Swept rule

on a shrinking subset of points. The stage completes when the output variables of a sub-timestep are computed at only two points, and nothing more can be computed without communicating with neighboring nodes.

Then, for the first time, the computing nodes communicate. Each node packs a subset of its computed variables and sends them to its neighboring node on the left. The transferred variables include two left-most outputs of every sub-timestep, forming a swept leading-edge of the space-time domain covered by the previous computing stage. In addition to sending the left leading-edge, each node keeps in its memory another swept leading-edge on the right side of the space-time domain. This right leading edge, represented by solid circles in Figure 3-1(c), combines with the left leading edge received from its right neighboring node, represented by open circles of the same color, to provide the variables needed for the next local computing stage.

The second local computation, like the first one, proceeds independently on each computing node. Each node starts with a V-shape in the space-time domain, formed by a pair of swept leading-edges, the left inherited from the first local computation, and the right received during the first communication. As in every local computing stage, each node computes all it can compute without communicating with other nodes. It computes all variables, lying in a space-time diamond illustrated in Figure 3-1(d), whose domains of dependence is covered by the pair of swept leading-edges forming the V. As in other local computing stage, this stage completes when the output variables of a sub-timestep are computed at only two points, and nothing more can be computed without communicating again with neighboring nodes.

The second communication is just like the first one, except that the right swept leading-edge is sent to the right neighboring node, and the left swept leading edge is kept in its own memory. In subsequent communication steps, even-numbered stages send the right leading-edge towards the right neighboring node, and odd-numbered rule stages send the left leading-edge towards the left neighboring node. This rule ensures each node to alternate between two sets of spatial points during odd and even numbered computing stages, limiting each node to two sets of spatial points. The result of the second communication stage is illustrated by Figure 3-1(e), in which the solid circles represent variables each node keeps from the previous computing stage, and the open circles of the same color represent variables the same node receives from its neighboring node.

The subsequent computing stages and communication stages, illustrated in Figures 3-1(f) to 3-1(h), are similar to the previous ones. Each computing stage compute all each node can compute without communication; each communication stage transfers a swept leading-edge over each pair of neighboring nodes, enabling the next computing stage.

3.1 A simplified performance analysis of the swept rule in 1D

To qualitatively understand the performance of the swept rule, we perform a simplified analysis, by making the following two assumptions:

1. Communication between computing nodes takes time τ, regardless of how much data is communicated.

2. Each sub-timestep on each spatial point takes time s to compute.

Let N be the total number of spatial points, and p be the number of computing nodes. Then the number of spatial points per node is $n = N/p$. A cycle of the swept rule, advancing the PDE for n sub-timesteps, consists of two computing stages and two communication stages. The two computing stages perform a total of n^2 calculations per node, each applying a sub-timestep to a spatial point. The computing stages therefore take $n^2 s$ time, according to our simplifying assumption. The two communication stages take 2τ time. The entire cycle then takes $n^2 s + 2\tau$ time. Divided among the n sub-timesteps during the cycle, the amount of computing time per sub-timestep is:

$$ns + 2\frac{\tau}{n} \tag{3.1}$$

Interconnect	Typical latency (τ)
Amazon EC2 cloud	150 μs
Typical Gigabit Ethernet	50 μs
Fast 100-Gigabit Ethernet	5 μs
Mellanox 56Gb/s FDR InfiniBand	.7 μs

Table 3.1: The range of latency τ commonly encountered today.

By increasing or decreasing how many nodes we use, we can easily change n. The other two variables, τ and s, are set by the hardware and the discretization of the PDE; they are harder to change. To understand how fast the swept rule is, we need to know the typical values of τ and s.

Table 1 attempts to cover the range of latency τ one may encounter today. The latency can change over three orders of magnitude, from the fastest Infiniband to a cloud computing environment not designed for PDEs.
The range of s is even wider; it can span over eight orders of magnitude. s depends both on the computing power of each node and on the complexity of each sub-timestep. If one sub-timestep on one spatial point takes f floating point operations (FLOP) to process, then processing an array of them with a node capable of F floating point operations per second (FLOPS) takes $s = f/F$ seconds per step-point. Running a cheap discretization on a powerful computing node leads to small f and large F, therefore a small s; running an expensive discretization on a light node leads to large f and small F, therefore a large s.

Table 2 attempts to estimate the range of s by covering the typical f for solving PDEs in a single spatial dimension, as well as the highest and lowest F on a modern computing node. Consider f of the heat equation, discretized with finite difference. Each sub-timestep takes only 3 FLOPs. A sub-timestep of a nonlinear system of equations, discretized with high order finite element, may take thousands of FLOPs. The highest F is achieved today by GPU nodes. 11.5 TeraFLOPS has been achieved by the AMD Radeon R9 295X2 graphics card, as well as by combining two AMD Radeon R9 290X graphics cards in a single node. The Summit supercomputer, expected to be delivered to Oak Ridge Leadership

Computing node – FLOPS	FLOP per step-point	Computing time per step-point (s)
Single thread Intel Nehalem 10 GFLOPS	4000 (FE system)	400 ns
Single thread Intel Nehalem 10 GFLOPS	200 (FV system)	20 ns
Single thread Intel Nehalem 10 GFLOPS	3 (FD scalar)	0.3 ns
Oak Ridge 2017 Summit node 40 TFLOPS	4000 (FE system)	100 ps
Oak Ridge 2017 Summit node 40 TFLOPS	200 (FV system)	5 ps
Oak Ridge 2017 Summit node 40 TFLOPS	3 (FD scalar)	75 fs

Table 3.2: The typical time it takes to compute one sub-timestep on a single spatial point for solving PDEs in a single spatial dimension

Computing Facility (OLCF) in 2017, uses a similar architecture to achieve 40 TeraFLOPS per node. Using older CPU architecture is slower. Particularly slow is equivalence of using a single thread per node, e.g., in flat-MPI-style parallel programming. On the first generation Intel Core i3/i5/i7 architecture, codenamed Nehalem, a single thread is capable of about 10 GFLOPS. These different node architectures, running different PDE discretization schemes, yield orders of magnitude different values of s.

With these values of τ and s, the plot in Figure 3-2 shows, according to Equation 3.1, how fast the diamond scheme runs as a function of n, the spatial points per node. The up-sloping, dashed and dash-dot lines represent the ns term in Equation (1) for different values of s, and the down-sloping, solid lines represent the τ/n term for different values of τ. For each combination of s and τ, the total time per sub-timestep as a function of n, plotted as thin black curves in Figure 3-2, can be found by summing the corresponding up-sloping and down-sloping lines.

The total time per sub-timestep can be minimized by choosing n. This minimizing n can be found, for each τ and s, at the intersection of the corresponding upsloping and downsloping lines in Figure 3-2. It has a mathematical expression $n^* = \sqrt{2\tau/s}$. Is largest for low s and high τ. n^* can be as large as tens of thousands when advancing a simple PDE

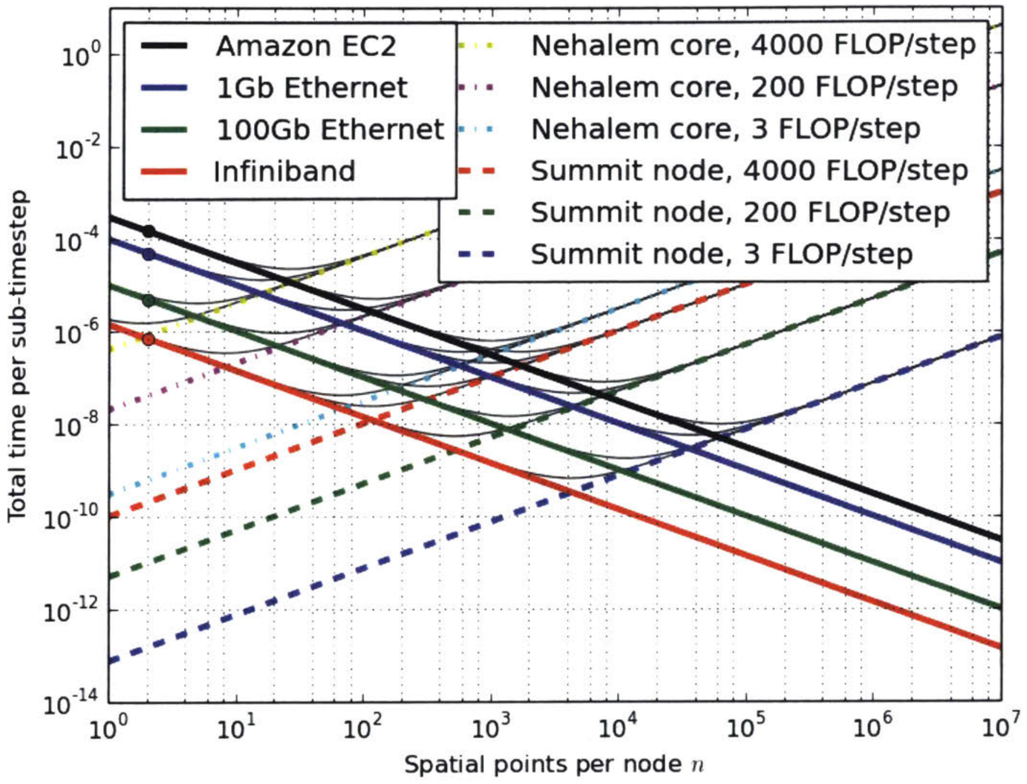


Figure 3-2: Analyzing the diamond performance using communication latency and CPU FLOP/step

with a cheap discretization on most powerful GPU instances in a cloud computing environment. n^* is smallest for high s and low τ. It can be as small as 1 or 2 when advancing a complex PDE with an expensive discretization in a older-CPU-based flat-MPI computing environment. These optimal values of n represent the limit of scaling. Above this optimum, decreasing n by scaling to more nodes would decrease the total time per sub-timestep, accelerating the simulation. But decreasing n beyond the optimum by scaling to even more nodes would not accelerate, but slow down the simulation.

At the optimal n, the minimum total time per sub-timestep also depends on τ and s. It has a mathematical expression $t^* = \sqrt{8\tau s}$. The fastest integration, unsurprisingly, is achieved for the lowest s and lowest τ. What is surprising is how fast it can be. The swept rule is theoretically capable of about 1 nanosecond per step, or almost a billion steps per second, if it uses the fastest Infiniband and the most powerful GPU nodes to efficiently advance the simplest PDE. This is about three orders of magnitude faster than what can be achieved with a method that requires communication every sub-timestep. To achieve this theoretical speed limit, it is necessary to not only minimize the latency τ, but also minimize s by fully utilizing the computational throughput of the most powerful computing nodes.

At the optimal n, the swept rule almost always breaks the latency barrier. A method that requires communication every sub-timestep takes at least τ per sub-timestep. This limit is the latency barrier. The swept rule, according to our simplified model, takes $t^* = \sqrt{8\tau s}$ per sub-timestep. It breaks the latency barrier whenever $\tau > 8s$, i.e., when the network latency exceeds the time it take for a computing node to advance one sub-timestep on 8 spatial points. In that case, it breaks the latency barrier by a factor of $\sqrt{\tau/8s}$. This ratio is largest for small s and large τ, i.e., when the discretization is cheap, each computing node is powerful, and the latency between nodes is high. For example, if Amazon EC2 upgrades their GPU instances to the most powerful available, using them to advance the simplest finite difference equation would break the latency barrier by a factor of about 13,000.

The swept rule has the potential to achieve 1 nanosecond per sub-timestep and breaking the latency barrier by a factor of 13,000. To get a realistic number for practical applications, we may assume that the discretization requires 100 FLOP per sub-timestep, and with good software engineering, a third of the computing power in a powerful node can be effectively utilized. Then with the fastest Infiniband, the swept rule can hope to achieve 10 nanosecond per sub-timestep, and with a cloud-computing-like latency, break the latency barrier by a factor of 1,300.

3.2 Interface and implementation of the swept scheme in 1D

The swept rule may seem challenging to implement, but it does not have to be. If a PDE solver can be decomposed into sub-timesteps, then the code that executes a sub-timestep on a spatial point can separate from the code that orders these executions and communicates with other computational nodes. The former code, which operates locally in space and time, implements the numerical scheme independent of the computer architecture. The later, global code, which decides when and where the local operations are executed, is optimized for computer architecture but is blind to what numerical scheme is used or what different equations is solved. The local and global codes share only an interface.

As of our current implementation of the swept rule in 1D, our interface consists of two simple functions. The first function will be called once per special point to set its initial data. The input variables to the initialization function are the global special point index and a special point structure. The following psedo-code exemplifies the initialization function interface

```
Init(pointIndex i, spacialPoint p)
{
        p.inputs[0] = variable 0 initial value
        p.inputs[1] = variable 1 initial value
        p.inputs[n] = variable n initial value
}
```

The second function in our interface is where the PDE solve actually happens. The input variables to the timestepping function are the index of which sub-timestep to be executed and a spacial point structure. The following psedo-code exemplifies the timestepping function of our interface.

```
timestep(substepIndex i, spacialPoint p)
{
#Based on the value of i, perform the proper operation in "p"
};
```

It is the responsibility of the swept rule scheme implementation developer to call the timestepping function with the proper sub-timestep index and spacial point.

The examples in Section 2 use this interface to implement spatial and temporal discretization schemes. If each time step of a PDE discretization is split into N sub-timesteps, then an application developer would implement a series of sub-timestep as functions and calls those properly in our timestepping interface function.

This interface, defined by the “SpatialPoint" and the “Global Point or Sub-timestep" indices, separates the concerns of application developers and computer scientists. A numerical analysts can program a solver in functions that operate on individual “SpatialPoint" structures. His code would be lightweight, thus easier to verify and to validate. It would also be portable to various current and future computer architectures. A computer scientist, on the other hand, can implement the swept rule interface to orchestrate the execution of atomic operations according to the swept rule. His code does not need to concern what these atomic operations are, thus is independent of the application. A simple working implementation can be found at `https://github.com/hubailmm/K-S_1D_Swept`.

3.3 Numerical Experiments

3.3.1 Swept rule 1D for finite difference – solution of the Kuramot-Sivashinsky equation

The swept rule is tested on the chaotic one space dimension Kuramoto-Sivashinsky (K-S) PDE. This equation was one of our choices to test the swept partitioning scheme as it contains high order derivatives and nonlinear terms. K-S PDE is knows to be stiff and produce solutions that exhibit spatio-temporal chaos, as shown in Figure 3-3. To solve the 1D K-S PDE, the special discretization was done using the finite difference scheme and the time integration was done using an explicit second-order Runga-Kutta integration scheme.

The picture in Figure 3-3 shows the solution obtained to the 1D K-S PDE using swept partitioning with periodic boundary conditions and an initial condition that is given by: $u(x, 0) = 2\cos\left(\frac{19x}{128}\right)$.

Figure 3-4 shows performance comparison between the classical straight and the swept decomposition that are applied to the one space dimension K-S PDE with the same spacial discretization. All the runs were conducted on a small two-node cluster that was formed on Amazon's Elastic Computing (EC2) services using "StarCluster", an open source cluster-computing toolkit for Amazon's EC2[2]. The EC2 instance type was "m1.xlarge" with an Amazon Machine Image (AMI) of "starcluster-base-ubuntu-13.04-x86_64".

As the main aim behind running this experiment is show how the swept decomposition breaks the latency barrier, the runs were conducted using a single MPI process residing in each compute node. The CPU in each node is "Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz" and the MPI latency between the nodes was measured and averaged to be around 150 *us*.

Figure 3-4 shows that the classic domain decomposition Is limited by the latency bar-

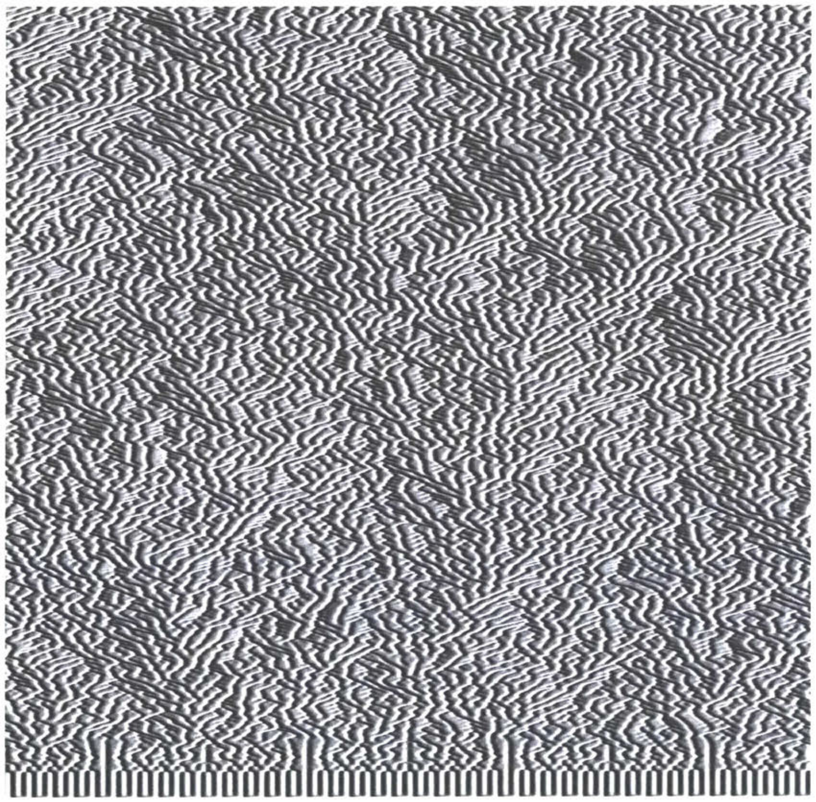


Figure 3-3: A chaotic solution of the Kuramoto-Sivashinsky equation. The X axis represents the space and the Y axix represents the time

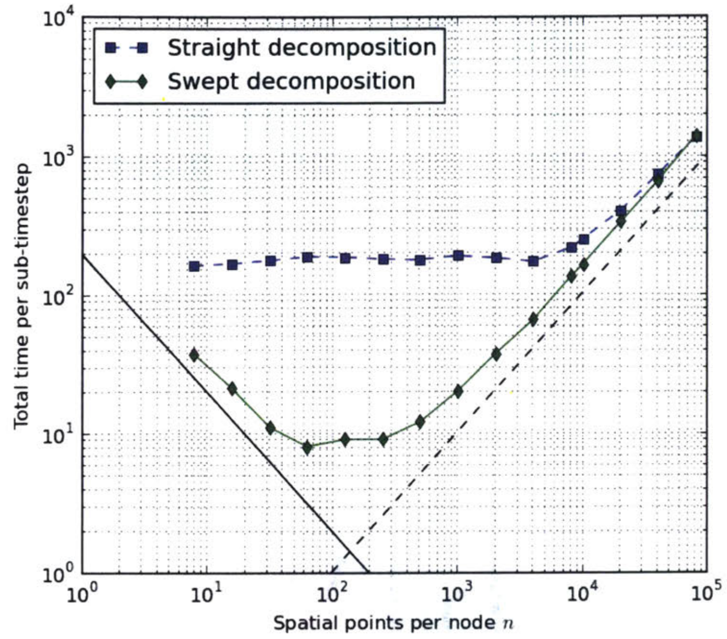


Figure 3-4: Performance comparison between the straight and the swept decomposition when solving the 1D K-S PDE. The black up-sloping, dashed line represent the ns term in Equation (1) , and the black down-sloping, solid lines represent the τ/n. The blue and orange lines represent the reported time per sub-timestep values for the straight and swept decompositions accordingly

rier. When each node has less than 10,000 grid points, it never takes less than 150 microseconds to integrate each sub-timestep. The swept decomposition rule, in contrast, breaks this latency barrier. Whenever each node has less than 10,000 grid points, it takes less than 150 microseconds to integrate each sub-timestep. At about 100 grid points per node, it takes minimum time to integrate each time step. This optimum coincide with the intersection of the ns and $2\tau/s$ lines, which is the same optimum predicted by our simplified analysis in Section 4. At that optimum, it takes less than 10 microseconds to integrate each time step, breaking the latency barrier by a factor of over 15.

A running implementation of the swept rule solving the 1D K-S PDE can found at `https://github.com/hubailmm/K-S_1D_Swept`

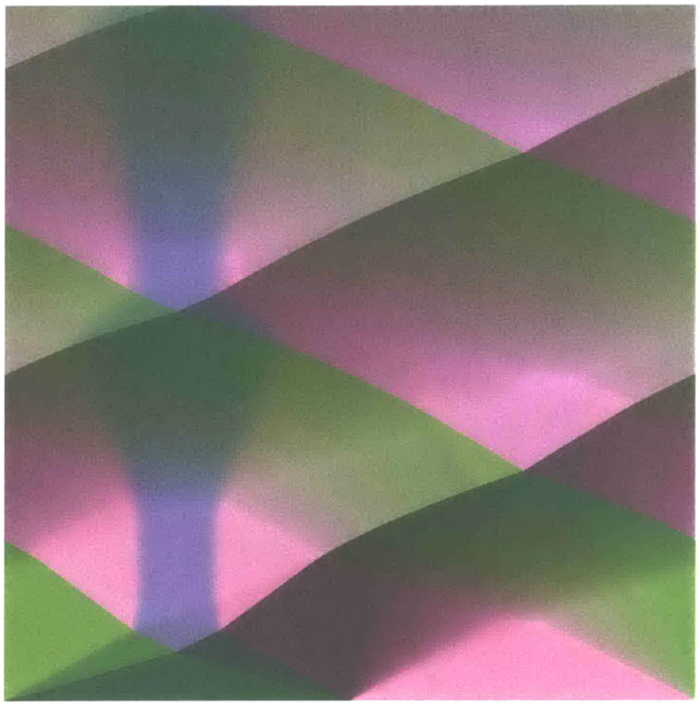


Figure 3-5: The solution to Euler's PDE obtained using the swept rule decomposition. The X axis represents the space and the Y axix represents the time

3.3.2 Swept rule 1D for finite volume – solution of the Euler equations of gas dynamics

The swept rule is tested on the 1D Euler equation for gas dynamics. We discretized the Euler equation with a second order finite volume scheme. Interface flux is computed with minmod limiter and scalar dissipation proportional to the spectral radius of the Roe matrix. A second order Runge Kutta scheme, also known as the midpoint rule, is used for time integration. The same discretization of the Euler equations is implemented both in classic and swept decomposition rules; their source code can be found at `https://github.com/qiqi/Swept1D/blob/master/euler_classic.cpp` and `https://github.com/qiqi/Swept1D/blob/master/euler_swept.cpp` .

The picture in Figure 3-5 shows our solution to the Euler equation. We use red, green and blue to visualize densities of mass, momentum and energy, using PngWriter by Frank Ham[13]. The spatial domain is periodic, and solution is initialized according to the SOD shock tube test case. The initial condition is $\rho = 0.125, u = 0, p = 0.1$ on the left half of the domain, and $\rho = 1, u = 0, p = 1$ on the right half of the domain.

Figure 3-6 summarizes the performance of the straight and swept decomposition rules, applied to the same discretization of the Euler equation. Both straight and swept decomposition are implemented with a flat MPI architecture. The performance is measured on 64 MPI processes running on 8 nodes on the voyager cluster at MIT, each node containing an Intel Xeon Processor E5-1620 quad core processor with hyperthreading.

Figure 3-6 shows that the classic domain decomposition Is limited by the latency barrier. When each node has less than 1,000 grid points, it never takes less than 60 microseconds to integrate each sub-timestep. The swept decomposition rule, in contrast, breaks this latency barrier. Whenever each node has less than 1000 grid points, it takes less than 60 microseconds to integrate each sub-timestep. At about 50 grid points per node, it takes minimum time to integrate each time step. This optimum coincide with the intersection of the ns and $2\tau/s$ lines, which is the same optimum predicted by our simplified analysis in Section 4. At that optimum, it takes less than 6 microseconds to integrate each time step, breaking the latency barrier by a factor of about 10.

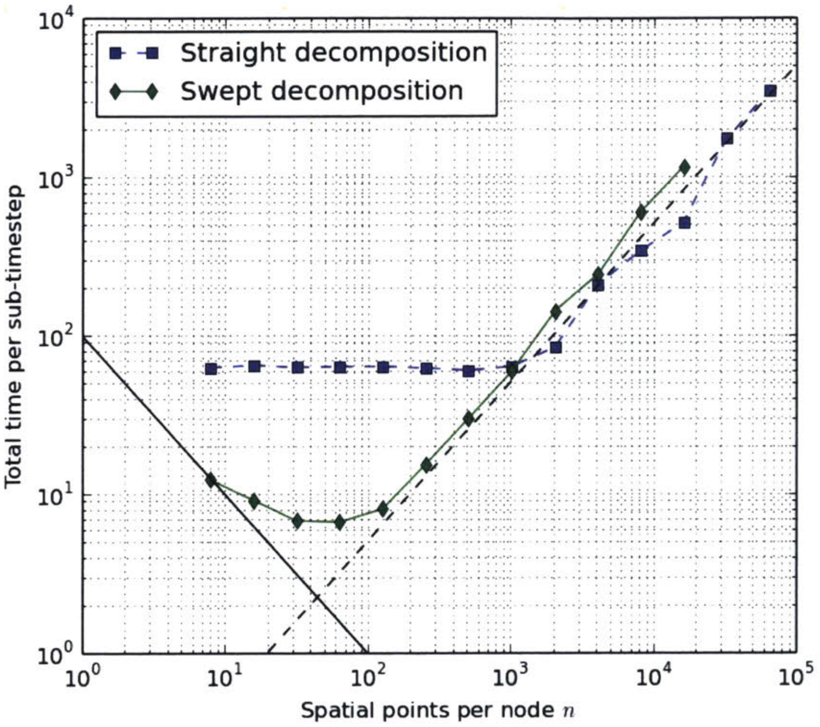


Figure 3-6: The performance of the straight and swept decomposition rules to Euler's equation. The black up-sloping, dashed line represent the ns term in Equation (1) , and the black down-sloping, solid lines represent the τ/n. The blue and green lines represent the reported time per sub-timestep values for the straight and swept decompositions accordingly

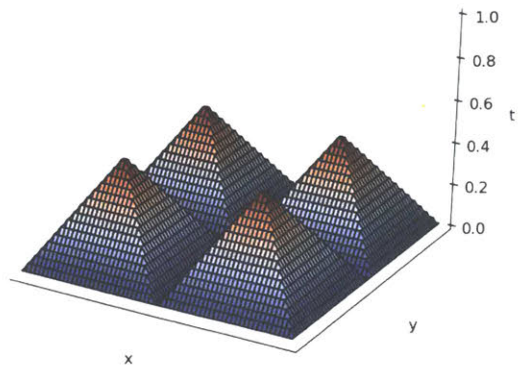
Chapter 4

The swept rule in 2D

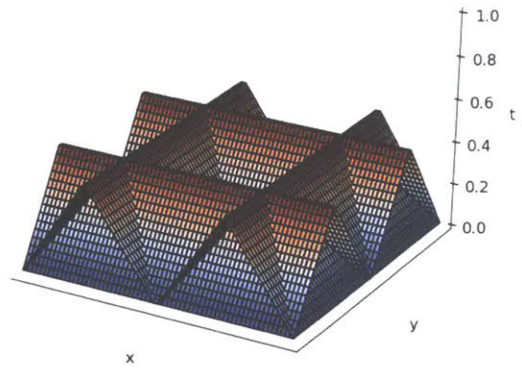
4.1 The Components of Swept Rule in 2D

We break the Swept Rule in two dimensional domains (Swept 2D) into four simple components. In this section, we first describes how these components work with each other to break the latency barrier. We then describe in detail how each of these components can be built. These four components are upward pyramid, longitudinal bridge, latitudinal bridge, and downward pyramid. All these components are subdomains in space-time, whose boundaries follow the discrete domain of dependence and the domains of influence.

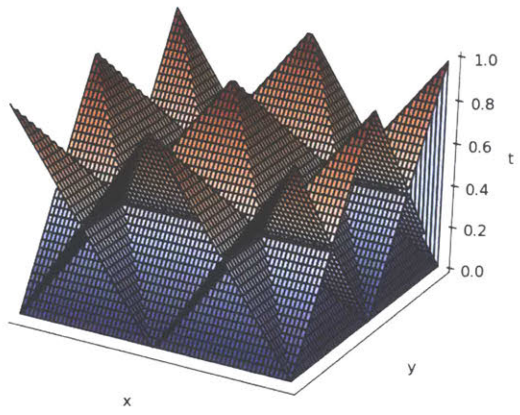
Figure 4-1 illustrates the idea of the Swept Rule in 2D. To start, we partition the spatial domain into squares subdomains. According to this partitioning, the initial condition of the PDE is distributed among the processors. Using the initial condition on a subdomain, each processor builds an upward pyramid, which is the first component of Swept 2D. These upward pyramids are illustrated in Fig 4-1(a). After that, each processor sends data to two neighboring processors, and receives data from two other neighboring processors. Once the data is received, each processor builds one longitudinal bridge and one latitudinal bridge, and then sends and receives data from its neighboring processors again. The bridges, which are the second and third components in Swept 2D, are illustrated in Fig 4-1(b). Finally, the processor fills the gaps between the generated bridges with downward pyramids, the fourth and last component. This completes what we call a half Swept 2D cycle. Repeating this process, illustrated in Fig 4-1(c-d), completes a full Swept 2D cycle. The rest of this section



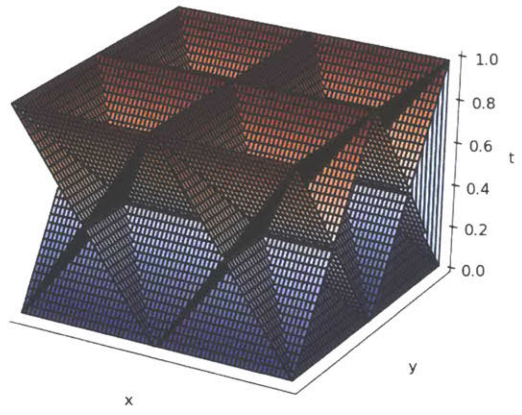
(a) Starting with square partitions and performing parallel computation without any communication to form pyramids



(b) Building the bridges that connect the tips of the pyramids and filling the gaps between the built bridges



(c) Growing the next set of pyramids



(d) Completing the Swept 2D cycle by building the next set of bridges and filling the gaps between them

Figure 4-1: The big picture of the Swept Rule in 2D. Because of periodic boundary condition, the half pyramids and quarter pyramids parts of full pyramids.

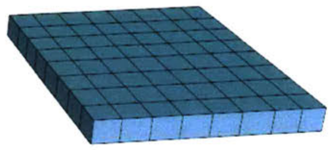
explains the details of each of the four components.

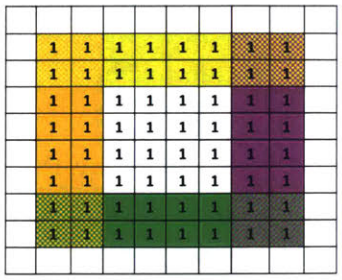
4.1.1 Upward Pyramid

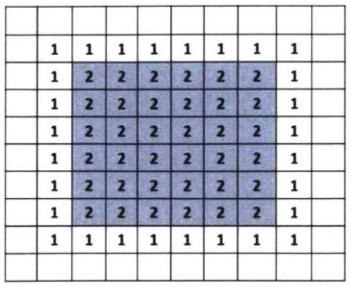
The upward pyramid is a square-pyramid-shaped subdomain in the three-dimensional, discrete space-time. The two spatial dimensions are discretized with a grid indexed by (i, j); the time dimension is discretized with time steps indexed by k. Without loss of generality, denote the first time step in the upward pyramid as 0, last time step as $n/2 - 1$. The base of the pyramid is a square subdomain of n by n grid points at time step 0. As the time step increases, the cross section of the upward pyramid maintains a square shape, whose side length decreases by 2 for every time step. At the last time step, the cross section is a 2 by 2 square.

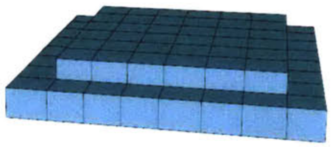
"Building" the upward pyramid means computing the values at all the space-time grids in the pyramid. After the pyramid is built, the values at the space-time grids on the four triangular sides, or upward pyramid panels, form the outputs, which feed into the other components of Swept 2D. Building the pyramid requires the values at the square base (time step 0) as inputs, then applying the stencil operation on the space-time grid points at time steps $1, 0+2, \ldots, 0+n/2-1.$. As an example, if the square base is 8 by 8 and each grid point has a value $u_{i,j}^0 = 1$; our stencil operation is incrementing by 1, i.e., $u_{i,j}^{k+1} = u_{i,j}^k + 1$. For illustration, we color code the 4 outputs, the north, south, west, and east triangular sides, with yellow, green, orange, and pink, respectively. Figure 4-2 illustrates the steps of building the upward pyramid.

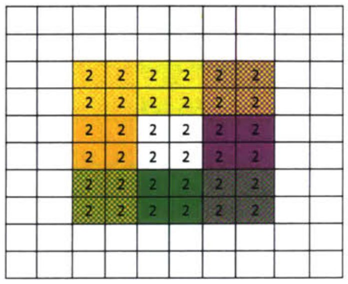
The algorithm for building the upward pyramid is described in Algorithm 1. The algorithm has two $(n+2)$ by $(n+2)$ internal arrays, $\mathcal{U}$ and $\mathcal{D}$.

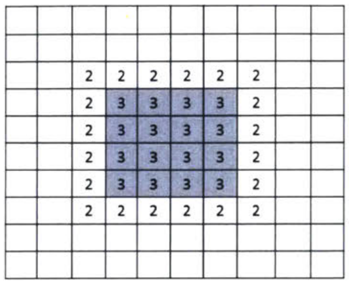

(a) Level 0 (8x8)

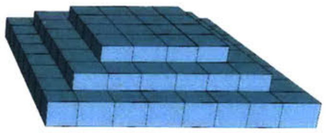

(b) Populating the sides

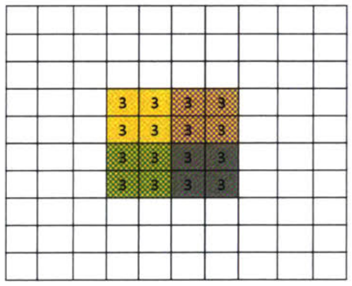

(c) Computation (6x6)

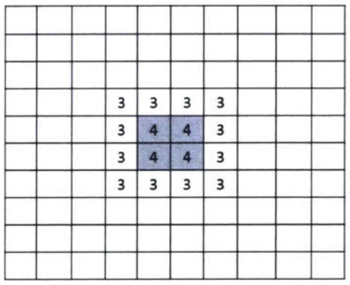

(d) Level 1 (6x6)

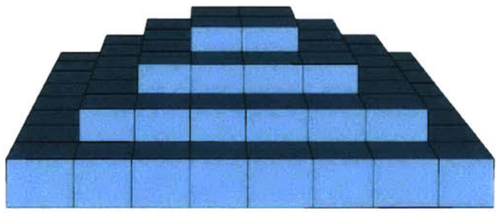

(e) Populating the sides


(f) Computation (4x4)


(g) Level 2 (4x4)


(h) Populating the sides


(i) Computation (2x2)


(j) Level 3 (2x2)

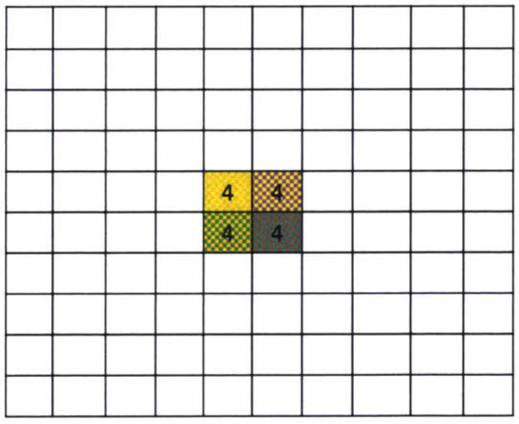

(k) Populating the sides

Figure 4-2: Illustrating the building process of the Upward Pyramid

Algorithm 1: Building The Swept 2D Up-Pyramid

```
$(\mathcal{N}, \mathcal{S}, \mathcal{W}, \mathcal{E})$ = function UpwardPyramid ($\mathbf{St}, \mathcal{B}$)
Input  : St: a list of stencil operations
         $\mathcal{B}$: an $n$ by $n$ array
Output: $\mathcal{N}, \mathcal{S}, \mathcal{W}, \mathcal{E}$: 4 arrays representing triangular sides
$\mathcal{D}_{1:n,1:n} \leftarrow \mathcal{B}$;
$\mathcal{N} \leftarrow \emptyset, \mathcal{S} \leftarrow \emptyset, \mathcal{W} \leftarrow \emptyset, \mathcal{E} \leftarrow \emptyset$;
for $k = 0, \ldots, \frac{n}{2} - 1$ do
    $\mathcal{N}^k \leftarrow \mathcal{D}_{k+1:n-k,k+1:k+2}$    $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{N}^k$
    $\mathcal{S}^k \leftarrow \mathcal{D}_{k+1:n-k,n-k-1:n-k}$    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}^k$
    $\mathcal{W}^k \leftarrow \mathcal{D}_{k+1:k+2,k+1:n-k}$    $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{W}^k$
    $\mathcal{E}^k \leftarrow \mathcal{D}_{n-k-1:n-k,k+1:n-k}$    $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}^k$
    for $i = 2 + k, \ldots, n - k - 1$ do
        for $j = 2 + k, \ldots, n - k - 1$ do
            $\mathcal{U}_{i,j} = \mathbf{St}_k(\{\mathcal{D}_{i',j'}, |i' - i| \leq 1, |j' - j| \leq 1\})$
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
```

Let us walk thought an example that has a square base of size 8 by 8. The Upward pyramid algorithm will proceed as follows. First, populate the 4 triangular sides, upward pyramid panels, with 2 layers from the four sides of the base. As shown in Figure 4-2(b), each triangular side now has 16 values. Figure 4-2(c) illustrates the subsequent stencil operation on level 0.

Proceeding to the next level, we further populate the 4 panels, each with 12 values from the new level, as shown in Figure 4-2(d,e). Now 12 new values will be added to each panel. Figure 4-2(f) illustrates the subsequent stencil operation on level 1.

Figure 4-2(g-i) illustrates the process on the next level, in which 8 more values are added to each panel. Figure 4-2(j-k) illustrates the process on the final level, in which the same 4 values are added to each panel, and no further computation is possible.

4.1.2 Longitudinal and Latitudinal Bridges

These two components differ only in their orientation. We hereby refer to both as bridges. The Swept 2D bridge is a three-dimensional, discrete space-time, structure. Similar to the Swept 2D upward pyramid component, the two spatial dimensions are discretized with a grid indexed by (i, j) and the time dimension is discretized with time steps indexed by k.

The Swept 2D bridges have the same height as the upward pyramid. Thinking of the three-dimensional discrete space-time, the bridge fills a valley that resides between two adjacent upward pyramids. "Building" the Swept 2D bridge means calculating all the space-time values in the valley, starting from 2 triangular sides, or panels, from two adjacent upward pyramids. Figures 4-3 and 4-4 visualize how the bridge component is built.

Depending on the orientation of the gap filled between the panels, we call the constructed bridge a Longitudinal or a Latitudinal bridge. One may think of the difference between the Longitudinal and Latitudinal bridges as if the building process has loops with "i" and "j" indices that are basically flipped.

The outputs of the bridge construction are two triangular sides. They have the same shape and size of the upward pyramid sides, except that they are flipped in the time axis. Their wider end is at the top and smaller end at the bottom. A longitudinal bridge requires North and South upward pyramid sides and generates West and East bridge sides. A latitudinal bridge requires West and East upward pyramid sides and produces North and South bridge sides.

The algorithms for building the Longitudinal and Latitudinal bridges are described in Algorithms 2 and 3 respectively. Just like the upward pyramid algorithm, both 2 and 3 algorithms have two $(n + 2)$ by $(n + 2)$ internal arrays, $\mathcal{U}$ and $\mathcal{D}$. Here n is the side length of the square subdomain partition, the base of the upward pyramid. If we denote the first time step in the Swept 2D bridge as 0, the last time step will be $n/2 - 1$.

Algorithm 2: Building The Swept 2D Longitudinal Bridge

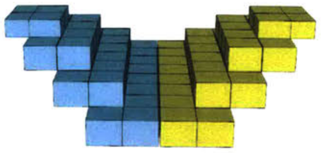
```
$(\mathcal{W}, \mathcal{E})$ = function LongitudinalBridge ($\mathbf{St}, \mathcal{N}, \mathcal{S}$)
Input  : St: a list of stencil operations
         $\mathcal{N}, \mathcal{S}$: 2 arrays representing triangular sides
Output: $\mathcal{W}, \mathcal{E}$: 2 arrays representing triangular sides
$\mathcal{W} \leftarrow \varnothing, \mathcal{E} \leftarrow \varnothing$
for $k = 0, \dots, \frac{n}{2} - 1$ do
    $\mathcal{D}_{k+1:n-k, \frac{n}{2}-k-1:\frac{n}{2}-k} \leftarrow \mathcal{N}^k$
    $\mathcal{D}_{k+1:n-k, \frac{n}{2}+k+1:\frac{n}{2}+k+2} \leftarrow \mathcal{S}^k$
    $\mathcal{W}^k \leftarrow \mathcal{D}_{k+1:k+2, \frac{n}{2}-k+1:\frac{n}{2}+k+2}$
    $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{W}^k$
    $\mathcal{E}^k \leftarrow \mathcal{D}_{n-k-1:n-k, \frac{n}{2}-k-1:\frac{n}{2}+k}$
    $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}^k$
    for $i = 2 + k, \dots, n - k - 1$ do
        for $j = \frac{n}{2} - k, \dots, \frac{n}{2} - k + 1$ do
            $\mathcal{U}_{i,j} = \mathbf{St}_k(\{\mathcal{D}_{i',j'}, |i' - i| \leq 1, |j' - j| \leq 1\})$
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
```

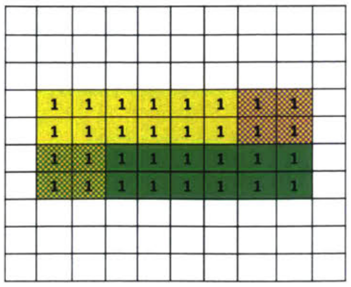
Algorithm 3: Building The Swept 2D Latitudinal Bridge

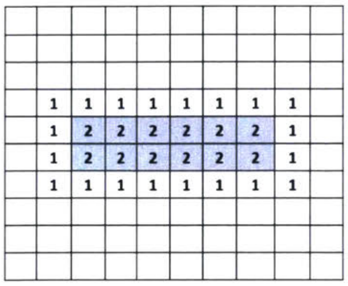
```
$(\mathcal{N}, \mathcal{S})$ = function LatitudinalBridge ($\mathbf{St}, \mathcal{W}, \mathcal{E}$)
Input  : St: a list of stencil operations
         $\mathcal{W}, \mathcal{E}$: 2 arrays representing triangular sides
Output: $\mathcal{N}, \mathcal{S}$: 2 arrays representing triangular sides
$\mathcal{N} \leftarrow \varnothing, \mathcal{S} \leftarrow \varnothing$
for $k = 0, \dots, \frac{n}{2} - 1$ do
    $\mathcal{D}_{\frac{n}{2}-k-1:\frac{n}{2}-k, k+1:n-k} \leftarrow \mathcal{W}^k$
    $\mathcal{D}_{\frac{n}{2}+k+1:\frac{n}{2}+k+2, k+1:n-k} \leftarrow \mathcal{E}^k$
    $\mathcal{N}^k \leftarrow \mathcal{D}_{\frac{n}{2}-k+1:\frac{n}{2}+k+2, k+1:k+2}$
    $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{N}^k$
    $\mathcal{S}^k \leftarrow \mathcal{D}_{\frac{n}{2}-k-1:\frac{n}{2}+k, n-k-1:n-k}$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}^k$
    for $i = \frac{n}{2} - k, \dots, \frac{n}{2} - k + 1$ do
        for $j = 2 + k, \dots, n - k - 1$ do
            $\mathcal{U}_{i,j} = \mathbf{St}_k(\{\mathcal{D}_{i',j'}, |i' - i| \leq 1, |j' - j| \leq 1\})$
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
```

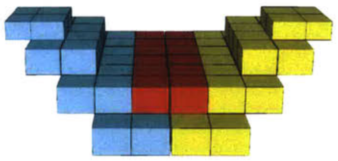
Figure 4-3 shows the building of a Longitudinal bridge for a square domain partition of size 8 x 8. Figure 4-3(a) shows the inputs, two upward pyramid triangular sides, or panels. Here we will use the same color codes given in Section 4.1.1 for the different upward pyramid sides in our example.

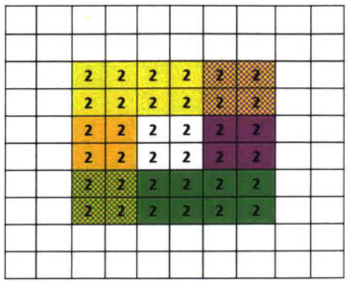
In the first step, we start by linking the first level of North and South upward pyramids triangular sides as shown in Figure 4-3(a). After that we populate the bottom part of newly generated West and East bridge triangular sides as illustrated in Figure 4-3(b) . Now stencil operation can be done to the inner blocks as shown in Figure 4-3(c).

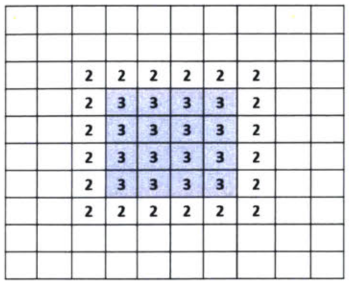

(a) Linking two panels

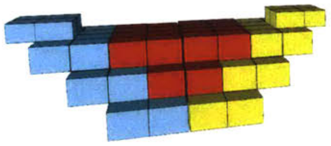

(b) Populating panels

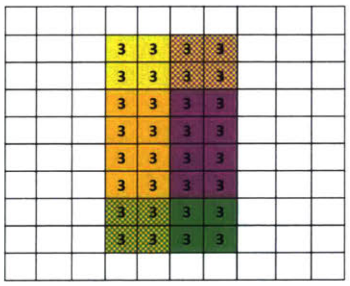

(c) Computation

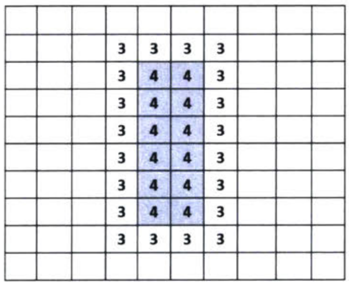

(d) Moving Up in time

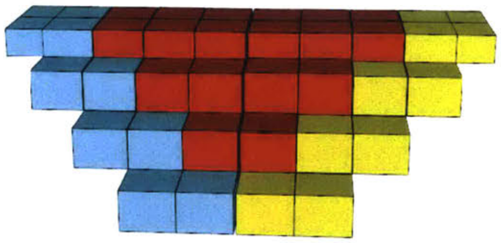

(e) Populating panels


(f) Computation


(g) Moving Up in time


(h) Populating panels


(i) Computation


(j) Reaching final level

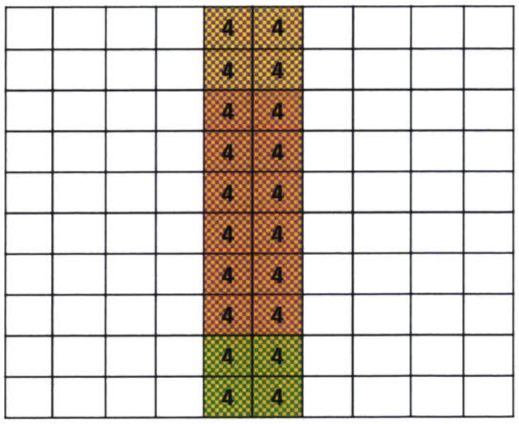
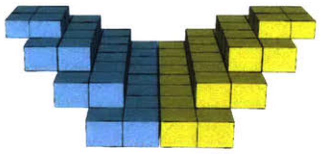
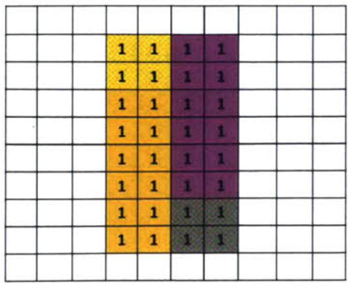
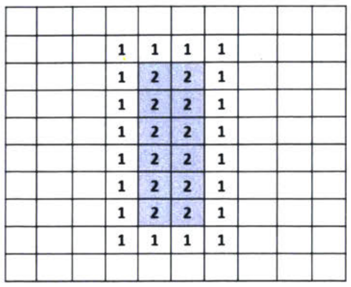
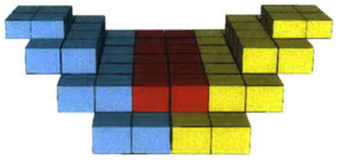

(k) Populating panels

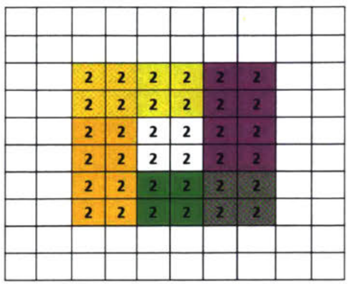
Figure 4-3: Illustrating the building process of the Longitudinal bridge

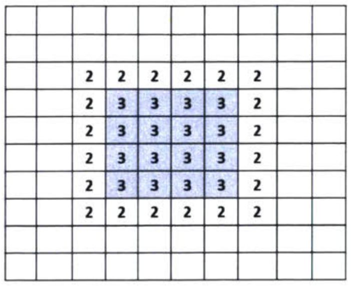

(a) Linking two panels

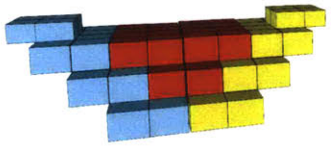

(b) Populating panels

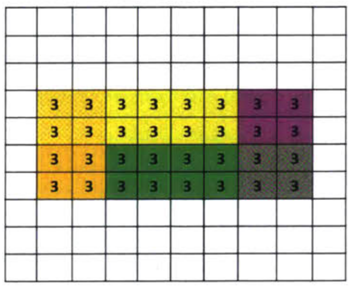

(c) Computation

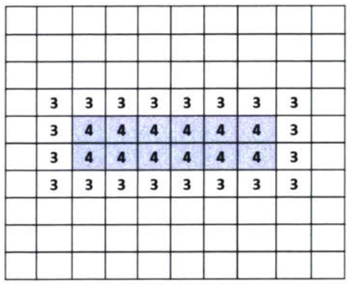

(d) Moving Up in time

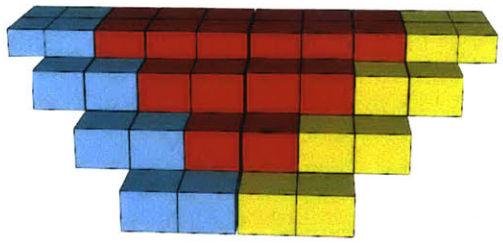

(e) Populating panels


(f) Computation


(g) Moving Up in time


(h) Populating panels


(i) Computation


(j) Reaching final level

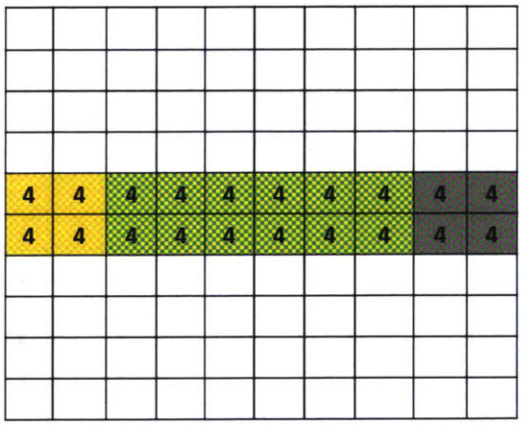

(k) Populating panels

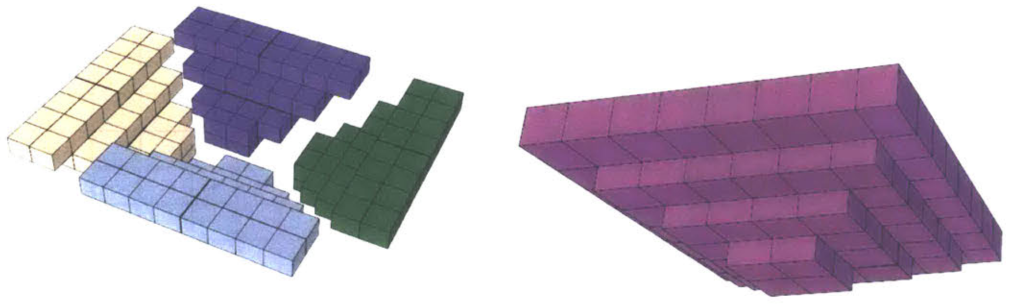
Figure 4-4: Illustrating the building process of the Latitudinal bridge

On the next level, as shown in Figure 4-3(d), we place the values contained in the second level of our input panels, the North and South Upward Pyramid panels, next to those we just computed in the previous step. We then populate the second level of the West and East bridge panels, our output, and then perform the second computation. Figures 4-3(d-f) clarify this process. 4-3(g-i) illustrate a similar process done on the next level, after which we reach the final level where no further computation is possible. We just populate the last level of the East and West bridge triangular sides, as shown in Figure 4-3(j,k).

As mentioned earlier, the Latitudinal bridge can be build in a similar way. So, we just summaries the process of building the Latitudinal bridge in Figure 4-4.

4.1.3 Downward Pyramid

This is the last component we need to complete the entire Swept Rule in 2D. This component is also three dimensional in space-time. Just like the Swept 2D upward pyramid and bridge components, the two spatial dimensions of the downward pyramid are discretized with a grid indexed by (i, j) and the time dimension is discretized with time steps indexed by k. Again, if we denote the first time step in the Swept 2D downward pyramid as 0, the last time step will be $n/2$; where n is the side length of the square subdomain partition. Note that the Swept 2D downward pyramids are one level higher in time than the upward pyramids and bridges components of Swept 2D. "Building" the downward pyramid means calculating all the values in space-time of this three-dimensional component. We construct a downward pyramid by filling a gap between four triangular sides of two Longitudinal and two Latitudinal bridges. Figure 4-5(a) shows the four triangular sides and the result obtained by filling the hollow area between them is shown in Figure 4-5(b). The end result in 4-5(b) is what we call the downward pyramid, which is a square pyramid that is similar in shape as the upward pyramid. It is not difficult to see that the gap between the 4 triangular sides in Figure 4-5(a) looks like a square pyramid that points downwards. The algorithm for building the downward pyramid is described in Algorithm 4. The algorithm has two $(n + 2)$ by $(n + 2)$ internal arrays, $\mathcal{U}$ and $\mathcal{D}$.



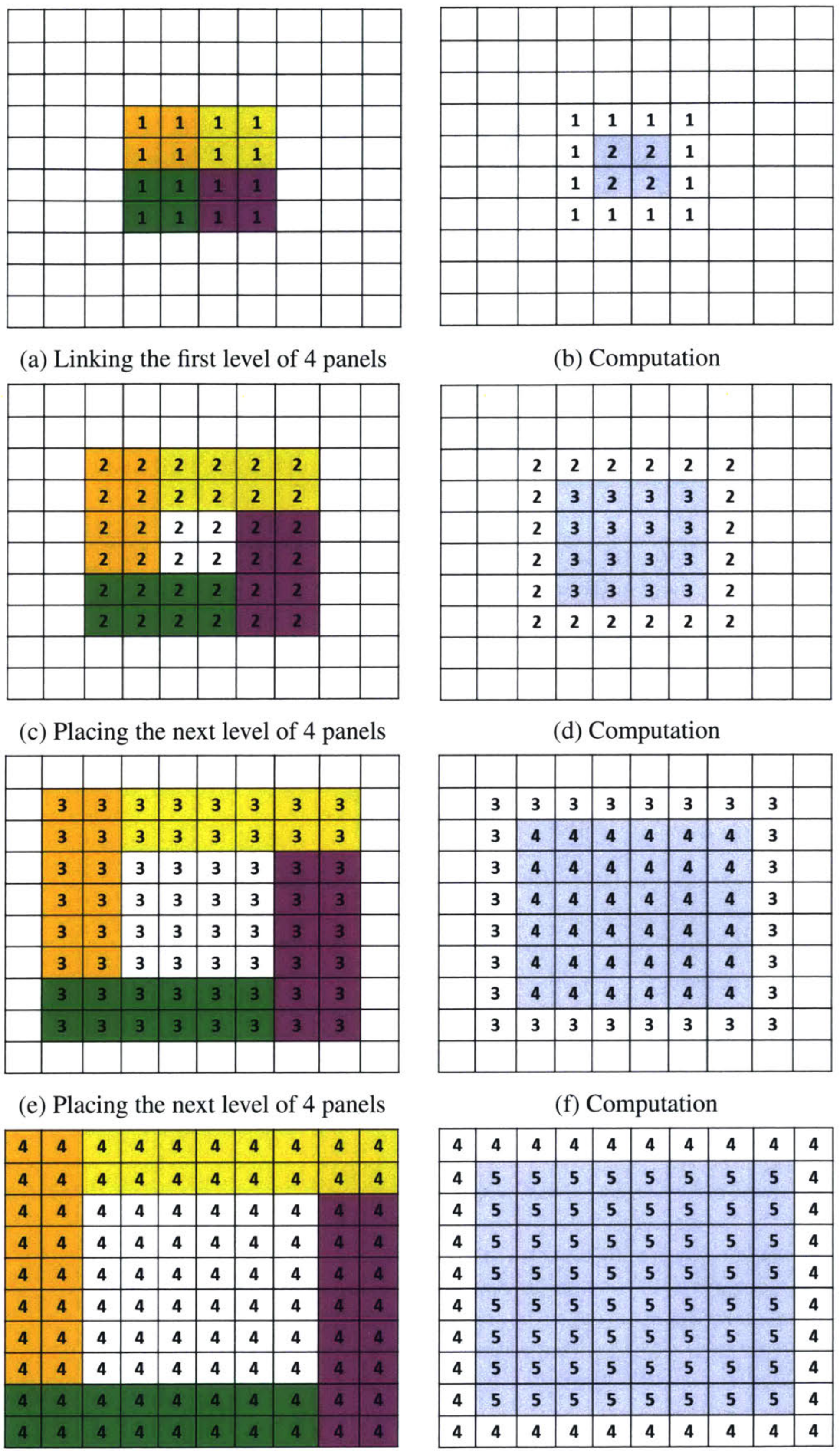
(a) Staring with 4 bridges triangular sides (b) Computation is done to fill the gap

Figure 4-5: Abstracted view for building the Swept 2D Downward Pyramid. The computations in right figure fits into the gap between the triangular sides in the left figure.

Algorithm 4: Building The Swept 2D Downward Pyramid

```
($\mathcal{B}$) = function DownwardPyramid (St, $\mathcal{N}, \mathcal{S}, \mathcal{W}, \mathcal{E}$)
Input  : St: a list of stencil operations
         $\mathcal{N}, \mathcal{S}, \mathcal{W}, \mathcal{E}$: 4 arrays representing triangular sides
Output: $\mathcal{B}$: an $n$ by $n$ array
for $k = 0, \dots, \frac{n}{2}$ do
    $\mathcal{D}_{\frac{n}{2}-k-1:\frac{n}{2}-k, \frac{n}{2}-k-1:\frac{n}{2}+k} \leftarrow \mathcal{W}^k$
    $\mathcal{D}_{\frac{n}{2}+k+1:\frac{n}{2}+k+2, \frac{n}{2}-k+1:\frac{n}{2}+k+2} \leftarrow \mathcal{E}^k$
    $\mathcal{D}_{\frac{n}{2}-k+1:\frac{n}{2}+k+2, \frac{n}{2}-k-1:\frac{n}{2}-k} \leftarrow \mathcal{N}^k$
    $\mathcal{D}_{\frac{n}{2}-k-1:\frac{n}{2}+k, \frac{n}{2}+k+1:\frac{n}{2}+k+2} \leftarrow \mathcal{S}^k$
    for $i = \frac{n}{2} - k, \dots, \frac{n}{2} + k + 1$ do
        for $j = \frac{n}{2} - k, \dots, \frac{n}{2} + k + 1$ do
            $\mathcal{U}_{i,j} = \mathbf{St}_k(\{\mathcal{D}_{i',j'}, |i' - i| \leq 1, |j' - j| \leq 1\})$
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
$\mathcal{B} \leftarrow \mathcal{D}_{1:n,1:n}$
```

As an example, we build a downward pyramid starting from 4 bridge triangular sides,



(a) Linking the first level of 4 panels

(b) Computation

(c) Placing the next level of 4 panels

(d) Computation

(e) Placing the next level of 4 panels

(f) Computation

(g) Placing the last level of 4 panels

(h) Computation

Figure 4-6: Illustrating the building process of the Downward Pyramid component of Swept 2D

North, South, West, East, and our output will be the top level of a Downward Pyramid, which represents a solution of the PDE at the same time level.

Starting with 4 panels, we simply place their first level values appropriately to form a 4 by 4 square as shown in Figure 4-6(a), where computation for the inner 2 by 2 square is possible. Figure 4-6(b)clarifies the process. Proceeding to the next level, we place the next level of our four inputs next to the results from the previous level to form a 6 by 6 square. That allows us to produce the inter 4 by 4 square of the next level, as clarified by Figures 4-6(c,d). We proceed through the next levels until the computation is done to the entire 8 by 8 square subdomain as illustrated in Figures 4-6(e-h).

4.2 Connecting the Swept Rule 2D components

Notice how each block at any level of our 4 Swept 2D components has the complete set of immediate neighbors from the level below. Let us relate that to the numerical stencil-based 2D simulation world. Assume that each block is a grid point in finite difference, a controlled volume in finite volume, or an element in finite element discretization. The blocks below each block represent the complete 9-point 2D stencil in a numerical discretization.

After the visual clarification of our 4 Swept 2D components, it is now time to show how these components work with each other to build the Swept Rule in 2D. Assume that we are starting with a square computational domain, with periodic boundary conditions, that can be decomposed into 4 subdomains. Each of these subdomains is assigned to a different processor. For simplicity in illustrating how the components are built, we will show a top view of our square domain with numbers representing the level of each grid point, i.e. timestep, of each block. Figure 4-7 illustrates our square domain decomposed into 4 smaller square subdomains.

As each processor starts to explicitly solve the PDE, following the domain of influence the domains of dependency to progress without the need to communicate with neighbour-

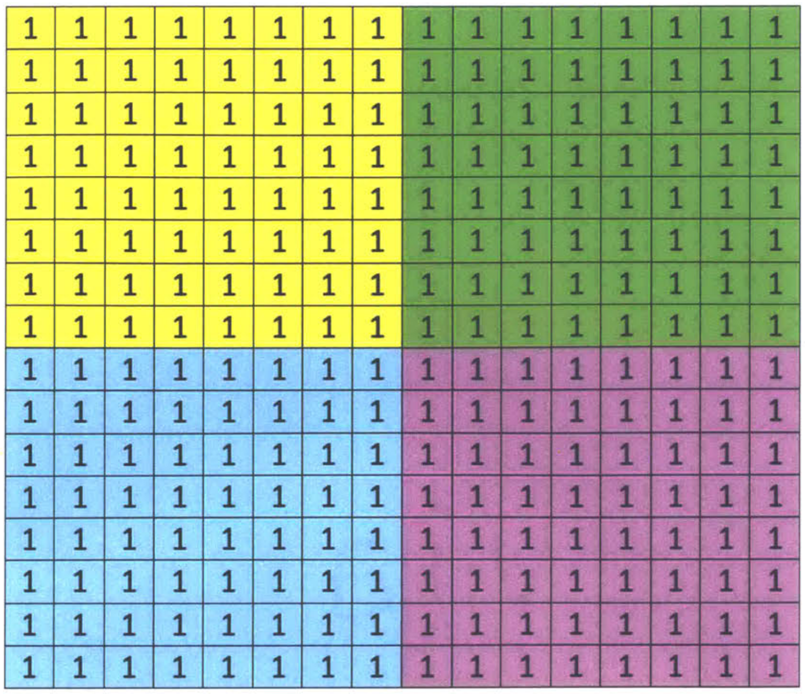
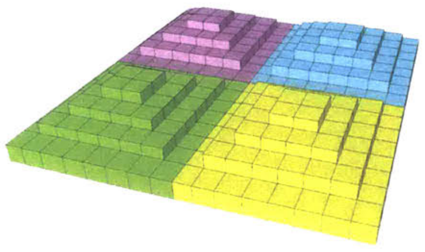


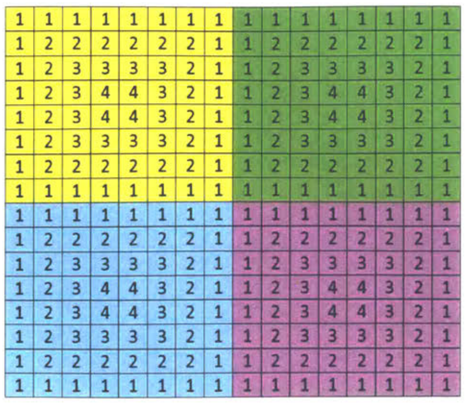
Figure 4-7: Illustration of a square domain partitioned into 4 sub-domains. Initial time level is 1

ing processes, each processor will be building an upward pyramid. Figure 4-8(a) shows the end result of the first stage of the Swept Rule in 2D. A 2D top view of the end result of stage 1 is shown in Figure 4-8(b) with numbers representing the time level of each block. Notice how the blocks are at different time levels. The tip of the Upward Pyramids at each subdomain is at level 4.

The next stage in Swept 2D will be to build the Longitudinal and Latitudinal bridges. Those will fill the Longitudinal and Latitudinal aligned gaps between two pyramids. Figures 4-9(a,b) show the locations of the Longitudinal and Latitudinal bridges to be built. Notice that the arrows pointing outside the domain represent the bridges that are to be build due to the periodic boundary conditions. Basically, there will be a set of Longitudinal and Latitudinal bridges between the boundary pyramids. We use this technique to avoid working in a fraction, i.e. quarter or half, Swept 2D component.

In order to proceed from stage 1 to stage 2, that is building the bridges, the first com-

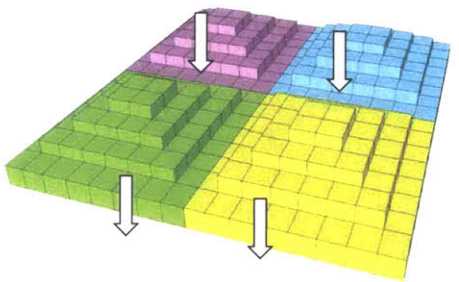

(a) The square domain state after Swept 2D stage 1 is complete



1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	2	2	2	2	2	2	1	1	2	2	2	2	2	2	1
1	2	3	3	3	3	2	1	1	2	3	3	3	3	2	1
1	2	3	4	4	3	2	1	1	2	3	4	4	3	2	1
1	2	3	4	4	3	2	1	1	2	3	4	4	3	2	1
1	2	3	3	3	3	2	1	1	2	3	3	3	3	2	1
1	2	2	2	2	2	2	1	1	2	2	2	2	2	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	2	2	2	2	2	2	1	1	2	2	2	2	2	2	1
1	2	3	3	3	3	2	1	1	2	3	3	3	3	2	1
1	2	3	4	4	3	2	1	1	2	3	4	4	3	2	1
1	2	3	4	4	3	2	1	1	2	3	4	4	3	2	1
1	2	3	3	3	3	2	1	1	2	3	3	3	3	2	1
1	2	2	2	2	2	2	1	1	2	2	2	2	2	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(b) A top-view of the square domain

Figure 4-8: The square domain state after Swept 2D stage 1 is complete


(a) A logical illustration of the locations of the Longitudinal bridges

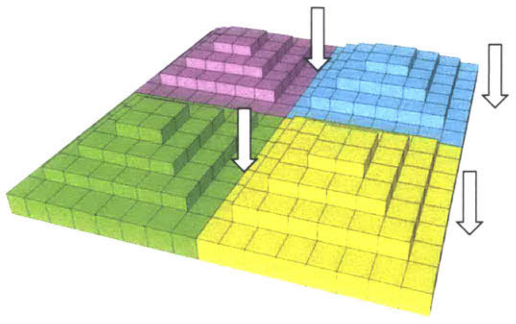
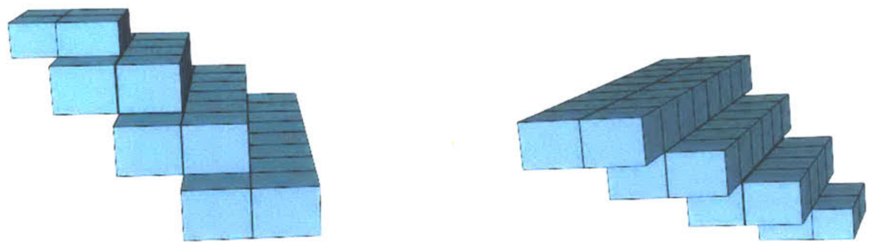

(b) A logical illustration of the locations of the Latitudinal bridges

Figure 4-9: Locations of the Longitudinal and Latitudinal bridges of Swept 2D



(a) The shape of an Upward Pyramid triangular side (panel) (b) The shape of a Bridge triangular side (panel)

Figure 4-10: The shapes of upward pyramid and bridges triangular sides

munication between the processors needs to take place. Each process needs to send data to two of its neighbors and receive data from its other two neighbors. To be more specific, each process will exchange 2 triangular sides, panels, of the upward pyramid that it built in stage 1. Figure 4-10(a) clarifies what we mean by an upward pyramid triangular side or panel.

If we agree that each upward pyramid has 4 triangular side (panels) and those panels can be named as North, South, West and East, then we notice that in order to build a Longitudinal bridge, each process needs a set of North and South panels. On the other hand, to build a Latitudinal bridge, each process needs a set of West and East panels. For this reason, each process will communicate its East and South upward pyramid panels to its East and South neighbors respectively. Each process will also receive 2 upward pyramid panels from its North and West neighbors. With these panels, each process can build a pair of bridges; one Longitudinal and one Latitudinal bridge.

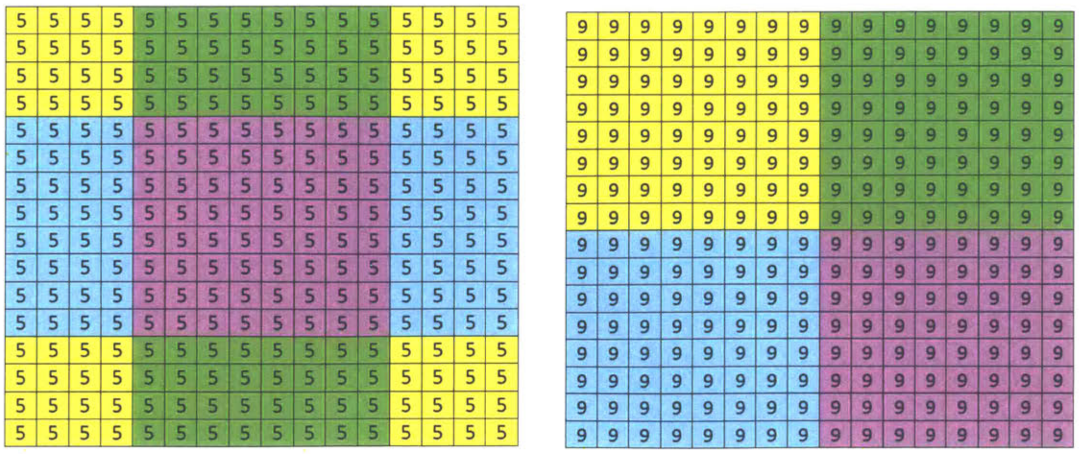
Notice that building the bridges at the boundary will cause a shift of the entire computational domain due to the periodic boundary condition assumed at the beginning. Again, this is to avid splitting a single Swept 2D component between processors. This way, each process will work on a complete component throughout the Swept 2D process .

The last stage will be to fill the remaining gaps between the bridges. This is the step where the Downward Pyramids are built. From our previous explanation of building the downward pyramid, we see that 4 triangular sides (panels) are needed. The panels this time come from the bridges that we built in stage 2. Notice that each Longitudinal bridge generates West and East panels. Also, each Latitudinal bridge generates North and South panels. The panels generated by the bridges are similar to those previously generated by the Upward pyramids, except that they are flipped in the time axis. Figure 4-10(b) illustrates bridge panels.

So, in order to proceed with building the Downward Pyramids, a second communication needs to take place. The communication this time is also for exchanging 2 bridge panels. Again, each process will send its East and South bridges panels to its East and South neighbors respectively. Each process will also receive 2 panels from its North and West neighbors. After the panels of the bridges are properly exchanged between the processes, each can build a downward pyramid.

At the end of this stage, the entire computational domain is at a consistent state. Meaning that all blocks, grid points or cells, are at the same time level. However, the domain arrangement has changed as a result of the shift that took place due to the periodic boundary condition. We call what we did so far a half Swept 2D cycle. Figure 4-11(a) clarifies what we mean by a computational domain shift due to periodic boundary condition. The view in Figure 4-11(a) is logical; the actual work was done to complete Swept 2D components. Each color in the figure represents work done by a single process

The other half of the Swept 2D cycle can be performed in the same way, except that the data exchange between the processes will happen between the other two neighbors. For example, instead of sending the East and South triangular sides (panels) of the Upward Pyramid and bridges to East and South neighbors, we send the North and West triangular sides (panels) of upward pyramids and bridges to the North and West neighbors. At the end of the second half of the Swept 2D cycle, the domain will have its original arrangement. Figure 4-11(b) shows the computational domain after a complete Swept 2D cycle.



(a) The state of square domain after a half Swept 2D cycle

(b) The state of square domain after a complete Swept 2D cycle

Figure 4-11: Showing the state of square domain after a half and a complete Swept 2D cycle

So, a complete Swept 2D cycle, requires a total of 4 communications to take place. 2 communications happen at each half of the Swept 2D cycle. Notice how starting with an initial condition as time level 1, and working with square subdomains of 8 by 8 allows a complete Swept 2D cycle to promote the computational domain 8 time levels involving only 4 communications between the processors.

We showed earlier how the Swept 2D components are simple and easy to build. Now we show the algorithm that connects, or glues, the Swept 2D components together making them construct the Swept Rule in 2D. Going through Algorithm 5 clarifies many points that may not have been clarified. One such point, is how we mange the communication of the panels of Upward Pyramids and Bridges. For simplicity, we abbreviated the names of the panels that get exchanged between the processors by their direction and whether they are upward (belonging to an upward pyramid) or downward (belonging to a bridge). For example, $\mathcal{E}_{\mathcal{U}}$, stands for an East-Up panel and $\mathcal{N}_{\mathcal{D}}$, stands for a North-Down panel. Moreover, we used the "$\odot\!\!\rightarrow$" symbol to denote sending to a process and the "$\leftarrow\!\!\odot$" to denote receiving from a process. We also abbreviated the neighboring processes with "$\mathbb{P}$" symbol combined with the direction of that process. For example, $\mathbb{P}_N$ means the North neighboring process. So, in algorithm 5, a line that reads $\mathcal{N}_{\mathcal{U}} \odot\!\!\rightarrow \mathbb{P}_N[\mathcal{N}_{\mathcal{U}}]$, simply means sending the

North triangular side of the Upward Pyramid to the North process.

Algorithm 5: The Swept 2D

```
($\mathcal{B}$) = function Swept 2D ($\mathbf{St}, \mathcal{B}, C$)
Input  : St: a list of stencil operations
         $\mathcal{B}$: an $n$ by $n$ array representing a square subdomain
         $C$: The number of Swept 2D cycles to perform
Output: $\mathcal{B}$: The square subdomain partition after applying $C * n$ subtimesteps
for $c = 1, \ldots, C$ do
    $\mathcal{N_U}, \mathcal{S_U}, \mathcal{W_U}, \mathcal{E_U} \leftarrow$ UpwardPyramid ($\mathbf{St}, \mathcal{B}$)
    $\mathcal{N_U} \odot\!\!\rightarrow \mathbb{P}_N[\mathcal{N_U}]$;    $\mathcal{W_U} \odot\!\!\rightarrow \mathbb{P}_W[\mathcal{W_U}]$
    $\mathcal{N_U} \leftarrow\!\!\odot \mathbb{P}_S[\mathcal{N_U}]$;    $\mathcal{W_U} \leftarrow\!\!\odot \mathbb{P}_E[\mathcal{W_U}]$
    $\mathcal{W_D}, \mathcal{E_D} \leftarrow$ LongitudinalBridge ($\mathbf{St}, \mathcal{S_U}, \mathcal{N_U}$)
    $\mathcal{N_D}, \mathcal{S_D} \leftarrow$ LatitudinalBridge ($\mathbf{St}, \mathcal{E_U}, \mathcal{W_U}$)
    $\mathcal{N_D} \odot\!\!\rightarrow \mathbb{P}_N[\mathcal{N_D}]$;    $\mathcal{W_D} \odot\!\!\rightarrow \mathbb{P}_W[\mathcal{W_D}]$
    $\mathcal{N_D} \leftarrow\!\!\odot \mathbb{P}_S[\mathcal{N_D}]$;    $\mathcal{W_D} \leftarrow\!\!\odot \mathbb{P}_E[\mathcal{W_D}]$
    $\mathcal{B} \leftarrow$ DownwardPyramid ($\mathbf{St}, \mathcal{S_D}, \mathcal{N_D}, \mathcal{E_D}, \mathcal{W_D}$)

    $\mathcal{N_U}, \mathcal{S_U}, \mathcal{W_U}, \mathcal{E_U} \leftarrow$ UpwardPyramid ($\mathbf{St}, \mathcal{B}$)
    $\mathcal{S_U} \odot\!\!\rightarrow \mathbb{P}_S[\mathcal{S_U}]$;    $\mathcal{E_U} \odot\!\!\rightarrow \mathbb{P}_E[\mathcal{E_U}]$
    $\mathcal{S_U} \leftarrow\!\!\odot \mathbb{P}_N[\mathcal{S_U}]$;    $\mathcal{E_U} \leftarrow\!\!\odot \mathbb{P}_W[\mathcal{E_U}]$
    $\mathcal{W_D}, \mathcal{E_D} \leftarrow$ LongitudinalBridge ($\mathbf{St}, \mathcal{S_U}, \mathcal{N_U}$)
    $\mathcal{N_D}, \mathcal{S_D} \leftarrow$ LatitudinalBridge ($\mathbf{St}, \mathcal{E_U}, \mathcal{W_U}$)
    $\mathcal{S_D} \odot\!\!\rightarrow \mathbb{P}_S[\mathcal{S_D}]$;    $\mathcal{E_D} \odot\!\!\rightarrow \mathbb{P}_E[\mathcal{E_D}]$
    $\mathcal{S_D} \leftarrow\!\!\odot \mathbb{P}_N[\mathcal{S_D}]$;    $\mathcal{E_D} \leftarrow\!\!\odot \mathbb{P}_W[\mathcal{E_D}]$
    $\mathcal{B} \leftarrow$ DownwardPyramid ($\mathbf{St}, \mathcal{S_D}, \mathcal{N_D}, \mathcal{E_D}, \mathcal{W_D}$)
end
```

4.3 A simplified performance analysis of the swept rule in 2D

To simplify our analysis, let us consider a half Swept 2D cycle. We will also assume that if we partition the computational domain into small square subdomains, bandwidth issues are not encountered during communications. Meaning that, Communication between computing nodes takes time τ, regardless of how much data is communicated.

Now, if we partition our computational domain into square sub-domains of side length n, then within a process, each Swept 2D half cycle will perform $n/2$ substeps for n^2 grid points, which is the area of each subdomain. We know that each Swept 2D half cycle requires 2 communications between the computation processes. Let us assume that each physical compute node contains a single MPI process, then for a half Swept 2D cycle we will encounter 2 communication latencies between our compute nodes.

Let us break the half Swept 2D cycle to its basic components and try to understand how the 2 communication latencies are distributed among these components. In half a cycle, we build one Upward Pyramid, one Longitudinal Bridge, one and Latitudinal Bridge and finally a Downward Pyramid.

Building each of the 4 basic components involves some overhead that is a function of n (assuming linear relationship for small values of n). Let us denote each of these overhead values as $\alpha_u(n)$ for Upward Pyramids, $\alpha_d(n)$ for Downward Pyramids, and $\alpha_b(n)$ for both Longitudinal and Latitudinal Bridges. Also, assume that the communication latency between the physical compute nodes, regardless of how much data is exchanged, is τ and the time to perform each sub-timestep per grid point is s.

Knowing that half Swept 2D cycle performs $n/2$ sub-timesteps for n^2 grid points, we

Computing node – FLOPS	FLOP per step-point	Computing time per step-point (s)
Single thread Intel Nehalem 10 GFLOPS	8000 (FE system)	800 ns
Single thread Intel Nehalem 10 GFLOPS	400 (FV system)	40 ns
Single thread Intel Nehalem 10 GFLOPS	6 (FD scalar)	0.6 ns
Oak Ridge 2017 Summit node 40 TFLOPS	8000 (FE system)	200 ps
Oak Ridge 2017 Summit node 40 TFLOPS	400 (FV system)	10 ps
Oak Ridge 2017 Summit node 40 TFLOPS	6 (FD scalar)	150 fs

Table 4.1: The typical time it takes to compute one sub-timestep on a single spatial point for solving PDEs in 2 spatial dimensions

can calculate the time needed to perform each sub-timestep as follows:

$$[(n^2 s)\frac{n}{2} + \alpha_u(n) + \alpha_d(n) + 2\alpha_b(n) + 2\tau]/\frac{n}{2} \tag{4.1}$$

Based on our implementation and experiments with Swept 2D algorithms, we found that Swept 2D components overhead values are very small and can be ignored. Therefore, if we ignore the overhead cost values and simplify equation 4.1, we get:

$$n^2 s + \frac{4\tau}{n} \tag{4.2}$$

Now we present a simplified theoretical performance model for the Swept Rule in 2D similar to that we presented for the Swept Rule in a single space dimension[3]. To understand how fast the swept rule in 2D is, let us use the same typical values of τ that we used in Swept rule 1D analysis, as listed in table 3.1. For the values of s let us consider the values listed in table 4.1 which attempts to estimate the range of s by covering the typical f for solving PDEs in a two spatial dimensions, as well as the highest and lowest F on a modern computing node.

With these values of τ and s, the plot in Figure 4-12 shows, according to our sub-timespte cost formula and ignoring overhead cost, how fast the Swept 2D scheme runs as a function of n, the spatial points per node. The up-sloping, dashed and dash-dot lines repre-

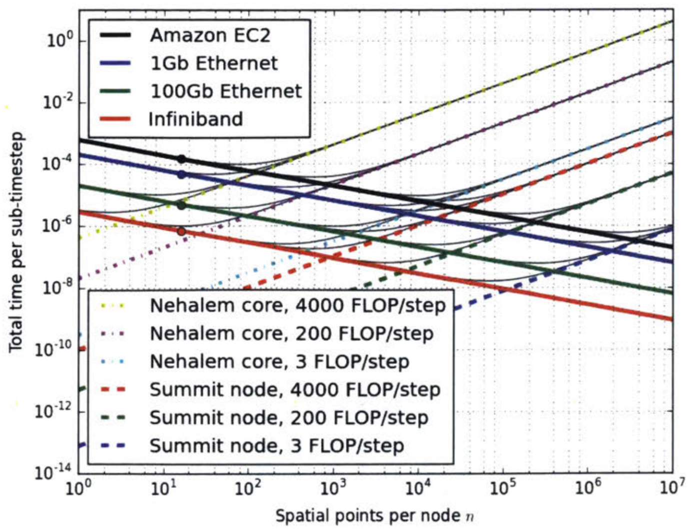


Figure 4-12: Analyzing the Swept 2D performance using communication latency and CPU FLOP/step

sent the $n^2 s$ term in our cost formula for different values of s, and the down-sloping, solid lines represent the $\frac{4\tau}{n}$ term for different values of τ. For each combination of s and τ, the total time per sub-timestep as a function of n, plotted as thin black curves in Figure 4-12, can be found by summing the corresponding up-sloping and down-sloping lines.

Just like the Swept Rule in 1D, The total time per sub-timestep can be minimized to the scaling limit by choosing an optimal n.

At the optimal n, the Swept Rule in 2D breaks the latency barrier. A method that requires communication every sub-timestep takes at least τ per sub-timestep.

4.4 More advanced performance analysis of the Hierarchical swept rule in 2D

From the previous analysis, we know that in a half Swept 2D cycle, the $\frac{n}{2}$ substeps that are performed on the n^2 grids points are distributed among the Swept 2D components and can preciously be calculated for each individual component. So, we are basically trying to define the breakdown of $(n^2 * s) * \frac{n}{2}$ sub-timesteps cost among the Swept 2D components. Looking at the way our algorithms build the Swept 2D components, we can define the following:

$\gamma_u(n) = \sum_{i=1}^{\frac{n}{2}-1} 4i^2 * s$ For the cost of the Upward Pyramid

$\gamma_d(n) = \sum_{i=1}^{\frac{n}{2}} 4i^2 * s$ For the cost of the Downward Pyramid

$\gamma_b(n) = \sum_{i=1}^{\frac{n}{2}-1} (2in - 4i^2) * s$ For the cost of the Bridge

Now we can re-write the sub-timestep cost formula as:

$$[\gamma_u(n) + \gamma_d(n) + 2 * \gamma_b(n) + \alpha_u(n) + \alpha_d(n) + 2 * \alpha_b(n) + 2 * \tau]/\frac{n}{2} \tag{4.3}$$

To fully exploit the power of the Swept Rule in 2D, we need to fully utilize each compute node by building each of the Swept 2D components in parallel. This basically means treating each compute node as a single processing unit. So, let us consider adding some parallelism to the Swept 2D scheme by doing what we call "hierarchical Swept 2D" to see if we can break the latency barrier even further. One way we could think of this is to start the scheme by assigning a big square partition to each node. This square partition then is then sub divided into smaller squares which can be processed by multiple threads. In other words, we are trying to start by building a big Upward Pyramid utilizing parallel processing.

This big pyramid, when fully built in parallel, should, in theory, allow Swept 2D to communicate after performing many more sub-timesteps and thus break the latency barrier with a larger factor than the non-Hierarchical Swept 2D. To further clarify this consider the following. A single Upward Pyramid with a square base of area 32 x 32 ($N = 32$) can be built starting with 4 small Upward Pyramids with square bases of area 8 x 8 ($n = 8$). If we carefully think about this, we see that any big Swept 2D Upward Pyramid can be broken into smaller Swept 2D components, which somehow can be built in parallel and the end

Level	Up-Pyramid	Lon-Bridge	Lat-Bridge	Down-Pyramid
3	9	6	6	4
2	4	2	2	1
1	1	0	0	0

Table 4.2: The typical time it takes to compute one sub-timestep on a single spatial point for solving PDEs in a single spatial dimension

result should be the same. The same is true for the big Swept 2D Bridges and Downward Pyramids. These are what we call the big Swept 2D components.

Starting with the big Upward Pyramid, let us try to come up with its recipe of the smaller Swept 2D components. Having a numeric example will definitely simplify this point. Assume that we want to build a big Upward Pyramid with a square base of size 24 x 24 ($N = 24$). Let us break the base into 9 8 x 8 ($n = 8$) squares. Those 9 small squares will be arranged as 3 x 3. The ratio between the big square and the small squares is $N/n = 24/8 = 3$ which means that we will need 3 levels of operations. Each of these levels will have its different smaller Swept 2D components. Let us go slowly to each of these levels for building a big Upward Pyramid starting with 9 small Up Pyramids. Table 3 lists the small Swept 2D components, i.e building blocks, of the 3-level big Upward Pyramid. Notice that we are listing the levels and their components in decreasing order, the same way they are built.

In the components list in Table 3, components of the same type at the same level might be built in parallel. Otherwise, synchronization needs to take place!! Now, we can simply derive the general overhead formula for any levels of Hierarchy for the Big Upward Pyramid. To start with, let us assume that all the components are built serially; that is to say, we will not deal with any synchronization issues for now. For the serial processing case, general overhead formula is as follows, where: L = number of levels:

$$\sum_{l=1}^{L}(l*l)*\alpha_u(n)+((l-1)*l)*\alpha_b(n)+(l*(l-1))*\alpha_b(n)+((l-1)*(l-1))*\alpha_d(n)$$

And since the Longitudinal and Latitudinal bridges shares the same overhead cost, the formula above can be simplified as follows:

$$\sum_{l=1}^{L}(l * l) * \alpha_u(n) + 2 * ((l-1) * l) * \alpha_b(n) + ((l-1) * (l-1)) * \alpha_d(n)$$

And the Big Upward Pyramid cost formula can be written as:

$$\sum_{l=1}^{L}(l * l) * \gamma_u(n) + 2 * ((l-1) * l) * \gamma_b(n) + ((l-1) * (l-1)) * \gamma_d(n)$$

Similarly, one can easily derive the general overhead and cost formulas for any levels of Hierarchy for the Big Bridges and the Downward Pyramid as follows:
Big Swept 2D Bridge, both Longitudinal and Latitudinal overhead and cost:

$$\sum_{l=1}^{L}((L-l)*l)*\alpha_u(n)+((L-l+1)*l)*\alpha_b(n)+((L-l)*(l-1))*\alpha_b(n)+((L-l+1)*(l-1))\alpha_d(n)$$

$$\sum_{l=1}^{L}((L-l)*l)*\gamma(n)+((L-l+1)*l)*\gamma(n)+((L-l)*(l-1))*\gamma_b(n)+((L-l+1)*(l-1))\gamma_d(n)$$

Big Swept 2D Downward Pyramid overhead and cost:

$$\sum_{l=1}^{L}((L-l)*(L-l))*\alpha_u(n)+2*((L-l+1)*(L-l))*\alpha_b(n)+((L-l+1)*(L-l+1))*\alpha_d(n)$$

$$\sum_{l=1}^{L}((L-l)*(L-l))*\gamma_u(n)+2*((L-l+1)*(L-l))*\gamma_b(n)+((L-l+1)*(L-l+1))*\gamma_d(n)$$

Now that we properly setup the overhead and cost formulas for our big Swept 2D components, let us consider the parallel execution whenever possible. As mentioned earlier, components of the same type at the same level can be executed in parallel. Moreover, the Longitudinal and Latitudinal bridges can be considered as one type and therefore can be built in parallel as well.

Adding parallelism is not free as at the minimum it involves synchronization cost, which has to be added to our big Swept 2D components overhead formulas. For simplicity, let us assume that the time it takes to spawn and synchronize threads is the same regardless of the number of threads. Also, let assume that we have enough compute resources so that components can be truly built in parallel without any queuing. Let us introduce β as a quantity that represents the threads spawn and join overhead and hence, at the worst case, the parallel versions of our performance overhead formulas will look the same and can be written as:

$$\sum_{l=1}^{L} \alpha_u(n) + \alpha_b(n) + \alpha_d(n) + 3 * \beta = L * (\alpha_u(n) + \alpha_b(n) + \alpha_d(n) + 3 * \beta)$$

And our Big Swept 2D components cost formulas can be written as:

$$\sum_{l=1}^{L} \gamma_u(n) + \gamma_b(n) + \gamma_d(n) = L * (\gamma_u(n) + \gamma_b(n) + \gamma_d(n))$$

If we now use the above overhead and cost formulas in our Swept 2D sub-timestep cost formula we get:

$$[(L * (\gamma_u(n) + \gamma_b(n) + \gamma_d(n))) + 4 * L * (\alpha_u(n) + \alpha_b(n) + \alpha_d(n) + 3 * \beta) + 2 * \tau]/\tfrac{N}{2}$$

Given enough compute resource per node and solving a PDE with simple stencil operations, the hierarchical swept rule in 2D should be able to easily achieve speed-up factors of more than 20. For example, according to our sub-timestep formula above, building the big Swept 2D components with hierarchy level of 8, should result in a speed-up factor of about 22.

4.5 Interface and implementation of the Swept scheme in 2D

The interface we present for our implementation of the Swept Rule in 2D is very simple. It is similar to that we presented earlier for the Swept Rule in single space-dimension[3]. We basically present a data initialization function and a time-stepping function.

The input variables to the initialization function are the global "i" and "j" indicies for each spacial point and a spacial point structure representing a 2D stencil. The following pseudo code exemplifies the initialization function interface:

```
Init(pointIndex i,pointIndex j,spacialPoint2D p)
{
    Based on the location of the point (i,j), set its initial data
        p.inputs[0] = variable 0 initial value
        p.inputs[1] = variable 1 initial value
        p.inputs[n] = variable n initial value
}
```

The second function in our interface is where the PDE solve is performed. The input variables to the timestepping function are the index of which sub-timestep to be executed and a 2D 9-point stencil spacial point structure. The following psedocode shows the timestepping function of our interface.

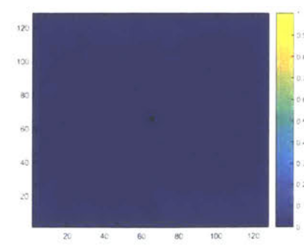
```
timestep(timestep i,spacialPoint2D p)
{
    Based on the value of i, perform the proper operation on p

}
```

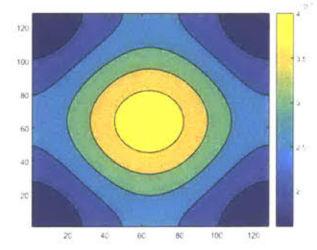
We standardized our implementations of the Swept Rule on this interface so that it is easy to keep improving on the internal implementation of the Swept 2D components, while not modifying the numerical aspects of the implemented PDE solver. A C++ implementation of the Swept Rule in 2D can be found at:`https://github.com/hubailmm/Wave2D`

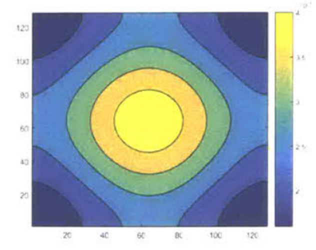
4.6 Numerical Experiments

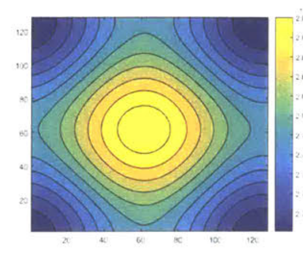
4.6.1 Swept rule in 2D – solution of the 2D Wave Equation

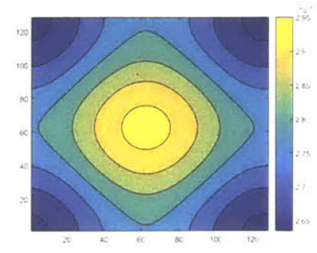
To start with, we tested our implementation of the Swept Rule in 2D in solving the two-dimensional wave equation. Our PDE configuration was periodic boundary conditions, CFL Number equals 0.3 and an initial wave source located at the center of the domain.


(a) Intial condition


(b) 1280 timesteps


(c) 2560 timesteps


(d) 3840 timesteps


(e) 5120 timesteps

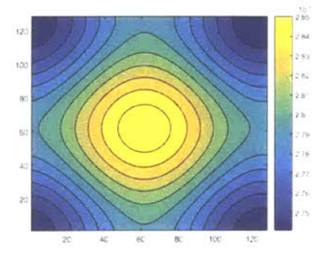

(f) 6400 timesteps

Figure 4-13: Contour plots showing the propagation of the Wave. This verifies our Swept 2D implementation

The verification of our solution is shown in Figure 4-13.

An experiment was conducted in a small 9-node Amazon EC2 cluster. StartCluster was used to form the 9-node cluster with an EC2 instance type of "c3.large" and an Amazon Machine Image (AMI) of "ami-6b211202"[2]. A single MPI process was assigned to each node. Figure 4-14 plots in log scale the performance comparison between the straight "Classical" and Swept domain decomposition. Notice how the Swept Rule in 2D gains a speed-up factor of more than 3 when we assign about 1000 grid points per MPI process. Our implementation of the Swept Rule based solver of the 2D Wave equation can be found at:`https://github.com/hubailmm/Wave2D`

4.6.2 Swept rule in 2D – solution of the 2D Euler Equation for Gas-Dynamics

The Swept Rule in 2D was also tested on the Euler Equation for Gas-Dynamics with conservative, skew-symmetric, and compressible flow with periodic boundary conditions[20].

In the experiment, we used Finite Difference spacial discretization along with a 4-stage

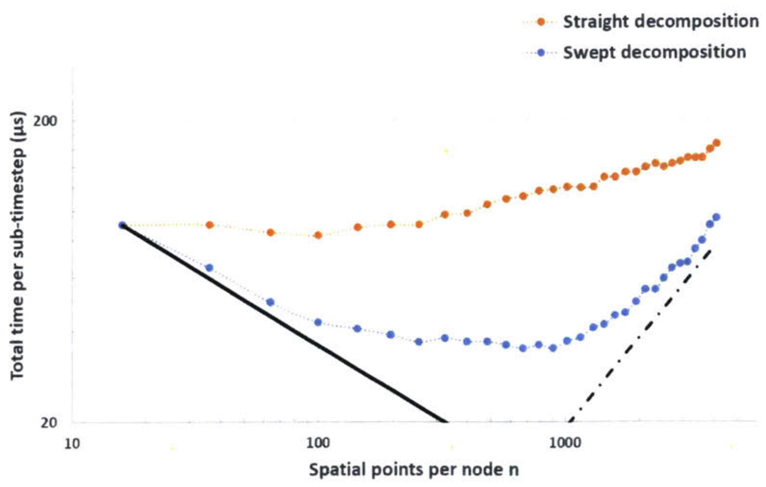
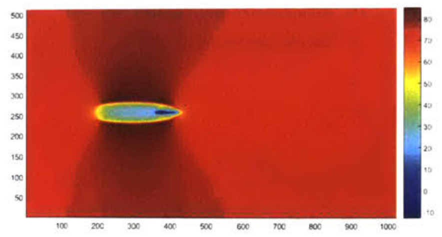


Figure 4-14: The performance of Straight and Swept 2D domain partitioning schemes when solving the 2D wave equation. The x-axis represents the number of spacial points assigned to each process. The y-axis represents the time, in microseconds, needed to perform a timestep. The black up-sloping, dashed line represents the $n^2 s$ term in Equation 4.2, and the black down-sloping, solid line represents the $\frac{4\tau}{n}$ term in the Equation

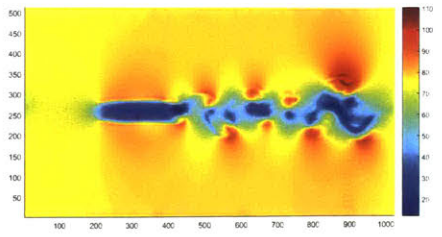
Runge-Kutta time integration scheme. The experiment setup was a rectangular wind tunnel with $lx = 50, ly = 25, nx = 1024, ny = 512, dx = \frac{lx}{nx}, dy = \frac{ly}{ny}, dt = 1e-6$ and an obstacle, given by $e^{(x^2+(y+.25*\frac{ly}{nx})^2)^8}$. The simulation initial condition was $\rho = 1.084, M = 0.2, u = c * M, v = 0.0, p = 101325$

The implementation was done in C++ and MPI. Solution verification is shown in Figures 4-15(a,b). The Figures show the results obtained for the x momentum after 3200 and 32000 timesteps respectively.

Our experiment was conducted in a cluster that was formed in Amazon's EC2 cloud services. StartCluster was used to form the cluster with an EC2 instance type of "c3.large" and an Amazon Machine Image (AMI) of "ami-6b211202"[2]. A single MPI process was assigned to each node. Figure 4-16 compares the performance of the classic and Swept 2D partitioning schemes. Notice how the Swept Rule in 2D achieves a speed-up factor of 4 when assigning about 400 grid points per node. Our implementation of the Swept Rule based solver of the 2D Euler PDE can be found at:`https://github.com/hubailmm/`



(a) 3200 timesptes



(b) 32000 timesteps

Figure 4-15: Showing contour plot of the Horizontal Momentum (u) when solving the 2D Euler equation after 3200 and 32000 timesteps. The X-axis and Y-axix represent the spacial location.

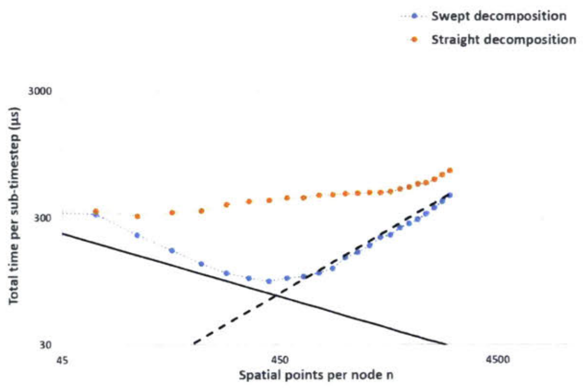


Figure 4-16: The performance of Straight and Swept 2D domain partitioning schemes when solving the 2D Euler equation. The x-axis represents the number of spacial points assigned to each process. The y-axis represents the time, in microseconds, needed to perform a sub-timestep. The black up-sloping, dashed line represents the $n^2 s$ term in Equation 4.2, and the black down-sloping, solid line represents the $\frac{4\tau}{n}$ term in the Equation

`Euler2D`

Chapter 5

The swept rule in 3D

5.1 Preliminary

Before we go into the details of the Swept Rule in three dimensional domains (Swept 3D), let us briefly review some aspects of the Swept1D and Swept2D. To be more specific, let us see what happens to their space-time components as the time dimension advances. In Swept1D, the domain has space-time partitions that looks like diamonds. If we split the diamond horizontally into to 2 halfs, we get 2 triangles. If we carefully look at these space-time triangles, we notice that one shrinks as time advances, meaning that the the x-axis of the spacial partition shrinks as the PDE advances in time until it becomes of size 2. However, the other triangle spatially starts from 2 points, and as time advances, it grows until in gets back to the size of the initial domain spacial partition. Between the shrinking and the growing triangles, Swept1D encounteres 1 communication between neighboring processes. In summary, Swept1D has 2 space-time components where as the time advances, the x-axis shrinks in one and grows in the other. Also, 1 communication latency takes place as the Swep1D scheme switches between these two components. Table 5.1 lists this summary.

Let us move to the Swept2D space-time components. Swept2D has Upward pyramids, Downward pyramids, longitudinal bridges, and latitudinal bridges. In the Upward pyramid, the x and y axes shrink as the PDE advances in time. The Downward pyramids is the complete opposite where both x and y axes grow as the PDE advanced in time. The bridges

Space-Time Component	X-axis
Shrinking Triangle	↘
COMMUNICATE	-
Growing Triangle	↗

Table 5.1: Computation-Communication Patters for Swept 1D

Space-Time Component	X-axis	Y-axis
Upward Pyramid	↘	↘
COMMUNICATE	-	-
Latitudinal Bridge	↗	↘
Longitudinal Bridge	↘	↗
COMMUNICATE	-	-
Downward Pyramid	↗	↗

Table 5.2: Computation-Communication Patters for Swept 2D

however, differ on that as time advances, one of their spacial dimensions grows and the second shrinks. So, for one bridge, the x-axis grows and the y-axis is shrinks. And for the other bridge, the y-axis grows and the x-axis is shrinks. As the Swept2D scheme advances in time, it encounters its first communication after building the Upward pyramids and before building the bridges. The second communication is encountered after building the bridges and before building the Downward pyramids. In summary, Swept2D, has 3 main space-time components. One has both x and y axises shrinking, one has both x and y axises growing, and one has an axis growing and another shrinking. As Swept2D switches between its 3 main space-time component, it encounters a data exchanges between neighboring processes. This summary is listed in Table 5.2.

After this brief review of the space-time components of Swept1D and Swept2D, the reader should have developed an idea about the number of the space-time components of Swept3D and number of communications Swept3D encounters to get to a consistent time level in the entire 3D computational domain.

5.2 The Components of Swept Rule 3D

The Swept Rule in 3D will have 4 main space-time components. Two of these space-time components have 3 permutations. Therefore, in total, Swept3D will have 8 space-time components if we include all possible permutation of X, Y, and Z axes. Swept3D starts with a cubic subdomain partition. This cube shrinks as Swept3D advances in time until it becomes a small 2 by 2 by 2 cube. So, let us call the first space-time component of Swept3D a "Shrinking Cube". The first communication in the Swept3D scheme takes place after the shrinking cube is done. After that, Swept3D connects the 2 by 2 by 2 cubes together. Let us call the component that connects the small cubes together a "Beam". Since we are working with 3 axises, Swept3D will be constructing 3 beams. Let us name these 3 beams with their directions as Latitudinal, Longitudinal, and Altitudinal. Swept3D encounters its second communication after it constructs its 3 beams. After the second communication is done, Swept3D starts building internal growing faces of cubes to be used as cores for the last Swept3D components. And for this reason, we call the third space-time component of Swept3D a "Core". Again, working on X, Y, and Z axes makes Swept3D build Latitudinal, Longitudinal, and Altitudinal cores. After building the cores, Swept3D performs its third, and last, communication. Building the last space-time component of Swept3D takes place after the third communication is done. This fourth component starts with a 2 by 2 by 2 cube which grows as time advances until its size gets equivalent to the size the shrinking cube that Swept3D started with. So, we call this last component a "Growing Cube". This was a general overview of how the Swept scheme in 3D works. Table 5.3 lists the order at which Swept3D space-time components are built and where communications between neighboring processes occur. The following sections list more detail about each space-time component of Swept3D.

5.2.1 Swept 3D Shrinking Cube

The Shrinking Cube components is four-dimensional in discrete space-time. The three spatial dimensions are discretized with a grid indexed by (i, j, k); the time dimension is discretized with time steps indexed by L. Without loss of generality, denote the first time

Space-Time Component	X-axis	Y-axis	Z-axis
Shrinking Cube	↘	↘	↘
COMMUNICATE	-	-	-
Latitudinal Beam	↗	↘	↘
Longitudinal Beam	↘	↗	↘
Altitudinal Beam	↘	↘	↗
COMMUNICATE	-	-	-
Latitudinal Core	↘	↗	↗
Latitudinal Core	↗	↘	↗
Altitudinal Core	↗	↗	↘
COMMUNICATE	-	-	-
Growing Cube	↗	↗	↗

Table 5.3: Computation-Communication Patters for Swept 3D

step in the shrinking cube as 0 and the last time step as $n/2 - 1$. The initial cubic subdomain is of size n by n by n grid points at time step 0. As the time step increases, the 6 faces of the cube shrink maintaining a cubic shape, whose side length decreases by 2 for every time step. At the last time step, the cube size is 2 by 2 by 2.

"Building" the shrinking cube means computing the values at all the space-time grids as the cube shrinks until it gets to a 2 by 2 by 2 cube. After the shrinking cube is fully built, the values at the space-time grids on the six square shrinking faces of the cube form the output, which feed into the other components of Swept 3D. Building the shrinking cube requires the values at the initial cubic partition (time step 0) as inputs, then applying the stencil operation on the space-time grid points at time steps $1, 0 + 2, \ldots, 0 + n/2 - 1$. As an example, if the cubic subdomain is 8 by 8 by 8 and each grid point has a value $u^0_{i,j,k} = 1$; our stencil operation is incrementing by 1, i.e., $u^{L+1}_{i,j,k} = u^L_{i,j,k} + 1$. For illustration, we color code the fourth dimension, the time dimension, with blue for $L = 0$, orange for $L = 1$, yellow for $L = 2$, green for $L = 3$, and red for $L = 4$. We maintain this color coding throughout this thesis. The algorithm for building the shrinking cube is described in Algorithm 6. The algorithm has two $(n + 2)$ by $(n + 2)$ by $(n + 2)$ internal arrays, $\mathcal{U}$ and $\mathcal{D}$.

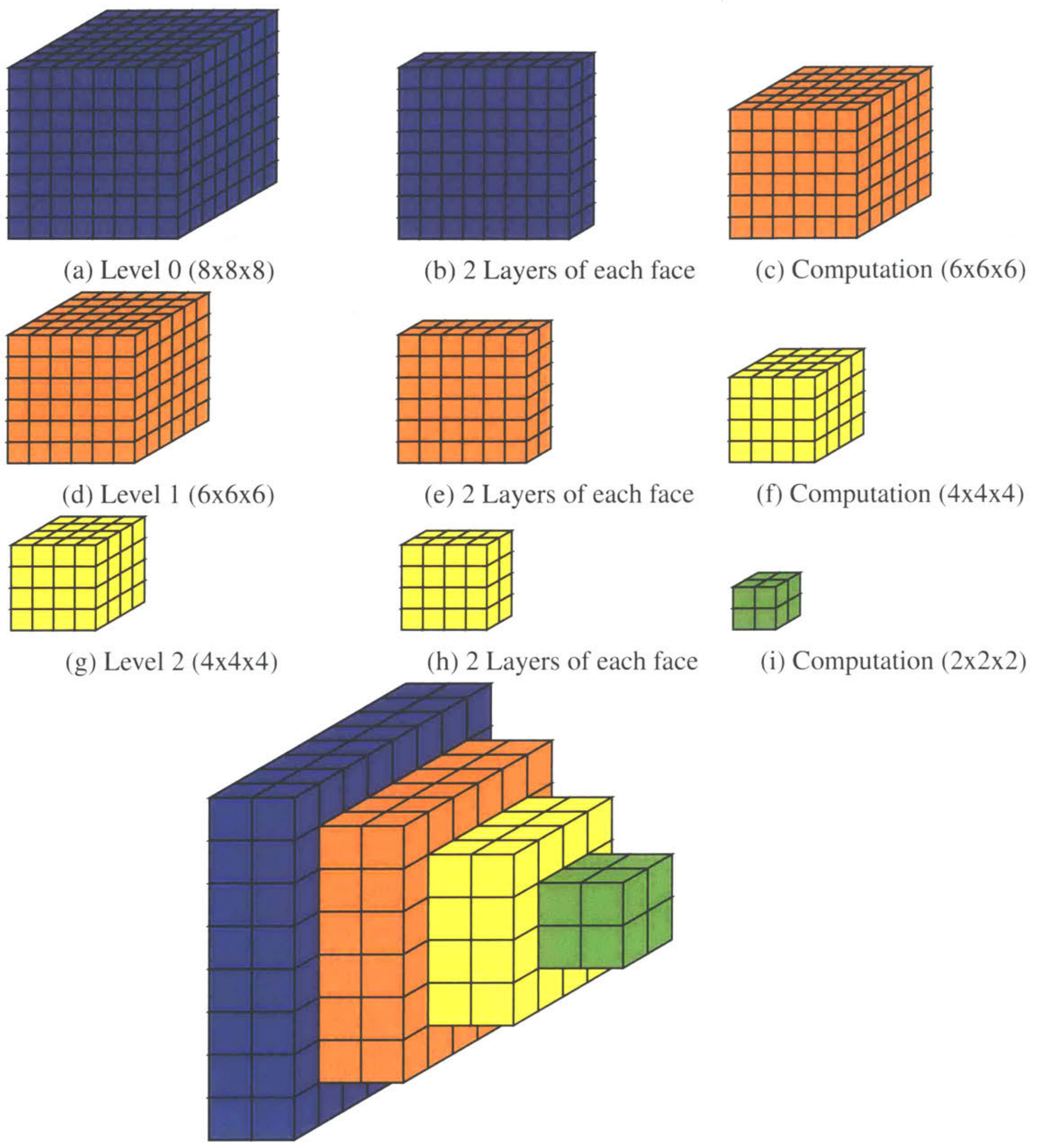
Algorithm 6: Building The Swept 3D Shrinking Cube

```
$(\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}, \mathcal{T}, \mathcal{B})$ = function ShrinkingCube ($\mathbf{St}, C$)
Input  : $\mathbf{St}$: a list of stencil operations
         $C$: an $n$ by $n$ by $n$ array representing the Cubic partition
Output: $\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}, \mathcal{T}, \mathcal{B}$: 6 arrays representing the shrinking faces of the cube
$\mathcal{D}_{1:n,1:n,1:n} \leftarrow C$;
$\mathcal{N} \leftarrow \emptyset, \mathcal{S} \leftarrow \emptyset, \mathcal{E} \leftarrow \emptyset, \mathcal{W} \leftarrow \emptyset, \mathcal{T} \leftarrow \emptyset, \mathcal{B} \leftarrow \emptyset$;
for $L = 0, \ldots, \frac{n}{2} - 1$ do
    $\mathcal{N}^L \leftarrow \mathcal{D}_{L+1:n-L, L+1:L+2, L+1:n-L}$    $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{N}^L$
    $\mathcal{S}^L \leftarrow \mathcal{D}_{L+1:n-L, n-L-1:n-L, L+1:n-L}$    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}^L$
    $\mathcal{E}^L \leftarrow \mathcal{D}_{n-L-1:n-L, L+1:n-L, L+1:n-L}$    $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}^L$
    $\mathcal{W}^L \leftarrow \mathcal{D}_{L+1:L+2, L+1:n-L, L+1:n-L}$    $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{W}^L$
    $\mathcal{T}^L \leftarrow \mathcal{D}_{L+1:n-L, L+1:n-L, L+1:L+2}$    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}^L$
    $\mathcal{B}^L \leftarrow \mathcal{D}_{L+1:n-L, L+1:n-L, n-L-1:n-L}$    $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{B}^L$
    for $k = 2 + L, \ldots, n - L - 1$ do
        for $j = 2 + L, \ldots, n - L - 1$ do
            for $i = 2 + L, \ldots, n - L - 1$ do
                $\mathcal{U}_{i,j,k} = \mathbf{St}_L(\{\mathcal{D}_{i',j',k'}, |i' - i| \leq 1, |j' - j| \leq 1, |k' - k| \leq 1\})$
            end
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
```

Let us walk through an example that has a cubic subdomain partition of size 8 by 8 by 8 at time $L = 0$ as shown in Figure 5-1(a). The Shrinking Cube algorithm will proceed as follows. First, take a copy of 2 layers of each of the 6 square faces of the cube. Figure 5-1(b) shows a sample face of the cube at this level. Figure 5-1(c) illustrates the subsequent stencil operation on level 0 for the internal 6 by 6 by 6 cube.

Proceeding to the next level, we copy 2 layers of the 6 faces of the cube. Figure 5-1(e) shows a sample face of the cube at level 1. Figure 5-1(f) illustrates the subsequent stencil operation on level 1 for the internal 4 by 4 by 4 cube.

Figure 5-1(g-i) illustrates the process on the next level in which stencil operation is



(a) Level 0 (8x8x8)

(b) 2 Layers of each face

(c) Computation (6x6x6)

(d) Level 1 (6x6x6)

(e) 2 Layers of each face

(f) Computation (4x4x4)

(g) Level 2 (4x4x4)

(h) 2 Layers of each face

(i) Computation (2x2x2)

(j) Sample Output of the Shrinking Cube

Figure 5-1: Illustrating the building process of the Swept 3D Shrinking Cube

done to the internal 2 by 2 by 2 cube. After this level, no further stencil operation can be done. Notice how the last 2 by 2 by 2 cube in Figure 5-1(i) will be shared by all the 6 faces of the cube. By now, the algorithm properly packed the 6 shrinking faces of the cube and prepared them for output. Figure 5-1(j) shows one of the outputs of the shrinking cuble algorithm; namely, the west faces.

5.2.2 Swept 3D Latitudinal, Longitudinal, and Altitudinal Beams

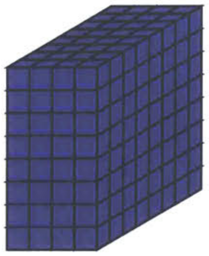
These three components differ only in their orientation. We hereby refer to them as beams. The Swept 3D beam is a four-dimensional, discrete space-time, structure. Similar to the Swept 3D shrinking cube, the three spatial dimensions are discretized with a grid indexed by (i, j, k) and the time dimension is discretized with time steps indexed by L.

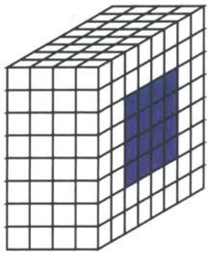
The Swept 3D beams have the same time span as the shrinking cube. Thinking of the four-dimensional discrete space-time, each Swept 3D beam connects two adjacent 2 by 2 by 2 cubes. "Building" the Swept 3D beam means calculating all possible space-time values in the space between two adjacent 2 by 2 by 2 by 2. The process starts by linking 2 sets of shrinking cube faces, 2 outputs of a shrinking cube, and ends with a beam of length $n + 2$. Figures 5-2 visualize how the Swept 3D beam component is built.

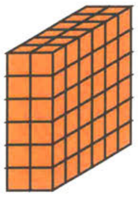
Depending on the orientation of the constructed beam, we call the constructed beam Longitudinal, Latitudinal, or Altitudinal. One may immediately figure out that the Latitudinal beam will have dimensions of $(n + 2)$ by 2 by 2. And the Longitudinal beam will be a 2 by $(n + 2)$ by 2 structure. Finally, the Altitudinal beam is of size 2 by 2 by $(n + 2)$.

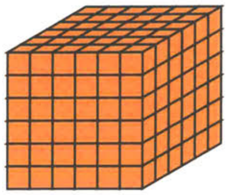
The outputs of the beam construction are four four-dimensional structures. A latitudinal beam requires East and West cube shrinking faces and generates North, South, Top, and Bottom sides. A longitudinal beam requires North and South cube shrinking faces and generates East, West, Top, and Bottom sides. An Altitudinal beam requires Top and Bottom cube shrinking faces and generates North, South, East, and West sides. Notice the relationship and the pattern between the beam orientation and the output it generates.

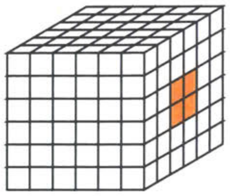
The algorithms for building the Latitudinal, Longitudinal, and Altitudinal beams are described in Algorithms 7, 8, and 9 respectively. Just like the shrinking cube algorithm, the three beam algorithms have two $(n+2)$ by $(n+2)$ by $(n+2)$ internal arrays, $\mathcal{U}$ and $\mathcal{D}$. Here n is the side length of the cubic subdomain. If we denote the first time step in the Swept 3D beam as 0, the last time step will be $n/2 - 1$.

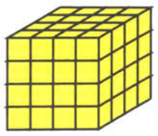

(a) Level 0 (4x8x8)

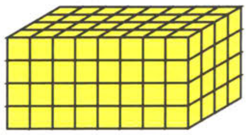

(b) Copying 4 sides

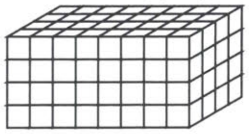

(c) Computation (2x6x6)

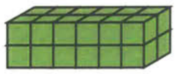

(d) Level 1 (6x6x6)

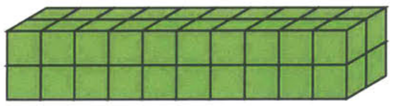

(e) Copying 4 sides

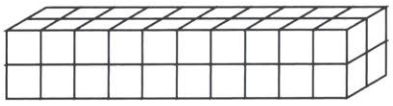

(f) Computation (4x4x4)


(g) Level 0 (8x4x4)


(h) Copying 4 sides


(i) Computation (6x2x2)


(j) Level 3 (10x2x2)


(k) Copying 4 common sides

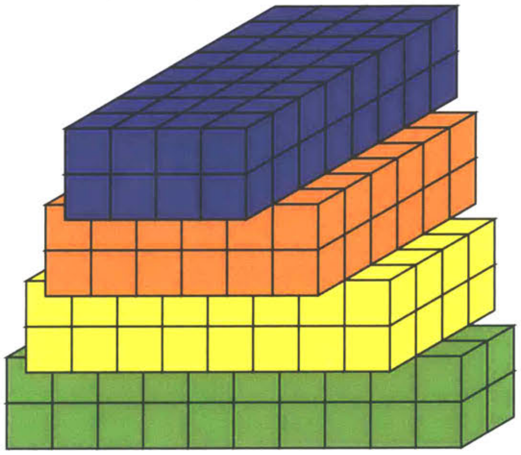

(l) Sample Beam Building output

Figure 5-2: Illustrating the building process of the Swept 3D Beam

Algorithm 7: Building The Swept 3D Latitudinal Beam

```
$(\mathcal{N}, \mathcal{S}, \mathcal{T}, \mathcal{B})$ = function LatitudinalBeam ($\mathbf{St}, \mathcal{E}, \mathcal{W}$)
Input  : St: a list of stencil operations
         $\mathcal{E}, \mathcal{W}$: 2 a representing the shrinking East and West faces
         of the shrinking cube
Output: $\mathcal{N}, \mathcal{S}, \mathcal{T}, \mathcal{B}$: 4 arrays representing the outer-sides of
         the octahedron
$\mathcal{N} \leftarrow \varnothing, \mathcal{S} \leftarrow \varnothing, \mathcal{T} \leftarrow \varnothing, \mathcal{B} \leftarrow \varnothing$ ;
for $L = 0, \ldots, \frac{n}{2} - 1$ do
    $\mathcal{D}_{n/2+L+1:n/2+L+2,L+1:n-L,L+1:n-L} \leftarrow \mathcal{E}^L$
    $\mathcal{D}_{n/2-L-1:n/2-L,L+1:n-L,L+1:n-L} \leftarrow \mathcal{W}^L$
    $\mathcal{N}^L \leftarrow \mathcal{D}_{n/2-L-1:n/2+L+2,L+1:L+2,L+1:n-L}$    $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{N}^L$
    $\mathcal{S}^L \leftarrow \mathcal{D}_{n/2-L-1:n/2+L+2,n-L-1:n-L,L+1:n-L}$    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}^L$
    $\mathcal{T}^L \leftarrow \mathcal{D}_{n/2-L-1:n/2+L+2,L+1:n-L,L+1:L+2}$    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}^L$
    $\mathcal{B}^L \leftarrow \mathcal{D}_{n/2-L-1:n/2+L+2,L+1:n-L,n-L-1:n-L}$    $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{B}^L$
    for $k = 2 + L, \ldots, n - L - 1$ do
        for $j = 2 + L, \ldots, n - L - 1$ do
            for $i = n/2 - L, \ldots, n/2 + L + 1$ do
                $\mathcal{U}_{i,j,k} = \mathbf{St}_L(\{\mathcal{D}_{i',j',k'}, |i' - i| \leq 1, |j' - j| \leq 1, |k' - k| \leq 1\})$
            end
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
```

Algorithm 8: Building The Swept 3D Longitudinal Beam

```
$(\mathcal{E}, \mathcal{W}, \mathcal{T}, \mathcal{B})$ = function LongitudinalBeam ($\mathbf{St}, \mathcal{N}, \mathcal{S}$)
  Input  : St: a list of stencil operations
           $\mathcal{N}, \mathcal{S}$: 2 a representing the shrinking North and South faces
           of the shrinking cube
  Output: $\mathcal{E}, \mathcal{W}, \mathcal{T}, \mathcal{B}$: 4 arrays representing the outer-sides of
           the octahedron
  $\mathcal{E} \leftarrow \emptyset, \mathcal{W} \leftarrow \emptyset, \mathcal{T} \leftarrow \emptyset, \mathcal{B} \leftarrow \emptyset$ ;
  for $L = 0, \ldots, \frac{n}{2} - 1$ do
      $\mathcal{D}_{L+1:n-L,n/2-L-1:n/2-L,L+1:n-L} \leftarrow \mathcal{N}^L$
      $\mathcal{D}_{L+1:n-L,n/2+L+1:n/2+L+2,L+1:n-L} \leftarrow \mathcal{S}^L$
      $\mathcal{E}^L \leftarrow \mathcal{D}_{n-L-1:n-L,n/2-L-1:n/2+L+2,L+1:n-L}$    $\mathcal{E} \leftarrow \mathcal{N} \cup \mathcal{N}^L$
      $\mathcal{W}^L \leftarrow \mathcal{D}_{L+1:L+2,n/2-L-1:n/2+L+2,L+1:n-L}$    $\mathcal{W} \leftarrow \mathcal{S} \cup \mathcal{S}^L$
      $\mathcal{T}^L \leftarrow \mathcal{D}_{L+1:n-L,n/2-L-1:n/2+L+2,L+1:L+2}$    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}^L$
      $\mathcal{B}^L \leftarrow \mathcal{D}_{L+1:n-L,n/2-L-1:n/2+L+2,n-L-1:n-L}$    $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{B}^L$
      for $k = 2 + L, \ldots, n - L - 1$ do
          for $j = n/2 - L, \ldots, n/2 + L + 1$ do
              for $i = 2 + L, \ldots, n - L - 1$ do
                  $\mathcal{U}_{i,j,k} = \mathbf{St}_L(\{\mathcal{D}_{i',j',k'}, |i' - i| \leq 1, |j' - j| \leq 1, |k' - k| \leq 1\})$
              end
          end
      end
      $\mathcal{U} \leftrightarrow \mathcal{D}$
  end
```

Algorithm 9: Building The Swept 3D Altitudinal Beam

```
$(\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W})$ = function AltitudinalBeam ($\mathbf{St}, \mathcal{T}, \mathcal{B}$)
Input  : St: a list of stencil operations
         $\mathcal{T}, \mathcal{B}$: 2 a representing the shrinking Top and Bottom faces
         of the shrinking cube
Output: $\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}$: 4 arrays representing the outer-sides of
         the octahedron
$\mathcal{N} \leftarrow \emptyset, \mathcal{S} \leftarrow \emptyset, \mathcal{E} \leftarrow \emptyset, \mathcal{W} \leftarrow \emptyset$ ;
for $L = 0, \dots, \frac{n}{2} - 1$ do
    $\mathcal{D}_{L+1:n-L,L+1:n-L,n/2-L-1:n/2-L} \leftarrow \mathcal{T}^L$
    $\mathcal{D}_{L+1:n-L,L+1:n-L,n/2+L+1:n/2+L+2} \leftarrow \mathcal{B}^L$
    $\mathcal{N}^L \leftarrow \mathcal{D}_{L+1:n-L,L+1:L+2,n/2-L-1:n/2+L+2}$    $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{N}^L$
    $\mathcal{S}^L \leftarrow \mathcal{D}_{L+1:n-L,n-L-1:n-L,n/2-L-1:n/2+L+2}$    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}^L$
    $\mathcal{E}^L \leftarrow \mathcal{D}_{n-L-1:n-L,L+1:n-L,n/2-L-1:n/2+L+2}$    $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}^L$
    $\mathcal{W}^L \leftarrow \mathcal{D}_{L+1:L+2,L+1:n-L,n/2-L-1:n/2+L+2}$    $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{W}^L$
    for $k = n/2 - L, \dots, n/2 + L + 1$ do
        for $j = 2 + L, \dots, n - L - 1$ do
            for $i = 2 + L, \dots, n - L - 1$ do
                $\mathcal{U}_{i,j,k} = \mathbf{St}_L(\{\mathcal{D}_{i',j',k'}, |i' - i| \leq 1, |j' - j| \leq 1, |k' - k| \leq 1\})$
            end
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
```

Now we walk through an example that builds a Latitudinal beam for a cubic partition of side length n. The inputs are East and West shrinking faces that were generated by a shrinking cube. In the first step of the first level, we start by linking the first 2 layers of the 8 by 8 East and West cube faces. This will form a domain partition of size 4 by 8 by 8 at time level $L = 0$ as shown in Figure 5-2 (a). After that, the algorithm takes a copy of the outer 2 layers of the North, South, Top, and Bottom sides. The copied layers are shown in white in Figure 5-2 (b). The last step in this level is to perform stencil operations for the inner 2 by 6 by 6 partition to move it to time level $L = 1$ as show in Figure 5-2 (c).

On the next level, we load the next 2 layers of size 6 by 6 from both the East and West cube faces, our inputs, and place them to the East and West sides of the partition we just generated from the previous level. This will form a partition of size 6 by 6 by 6 at time level $L = 1$ a shown in Figure 5-2 (d). The algorithm then takes a copy of the outer 2 layers of the North, South, Top, and Bottom sides. As before, the copied layers are shown in white in Figure 5-2 (e). We end the work in this level by performing stencil operations to the inner 4 by 4 by 4 partition to move it to time level $L = 2$ as showon in Figure 5-2 (f).

Proceeding to the next level and doing similar work as done in the previous 2 levels, this level ends with a 6 by 2 by 2 partition at time level $L = 3$. Figures 5-2 (g-i) summarize the work done at this level. The algorithm ends with loading the last 2 layers of our East and West inputs and placing them next to the 6 by 2 by 2 partition to form a 10 by 2 by 2 partition and copy this common partition to the North, South, Top, and Bottom sides as shown in Figures 5-2 (j-k). This ending 10 by 2 by 2 partition is what we call a Latitudinal beam. The 4 outputs of the algorithm are the stacked North, South, Top, and Bottom sides. Figure 5-2 (l) demonstrates a sample output for the Top sides.

The Longitudinal and Altitudinal Beams can be build in a similar way which is clear from algorithms 8 and 9.

5.2.3 Swept 3D Latitudinal, Longitudinal, and Altitudinal Cores

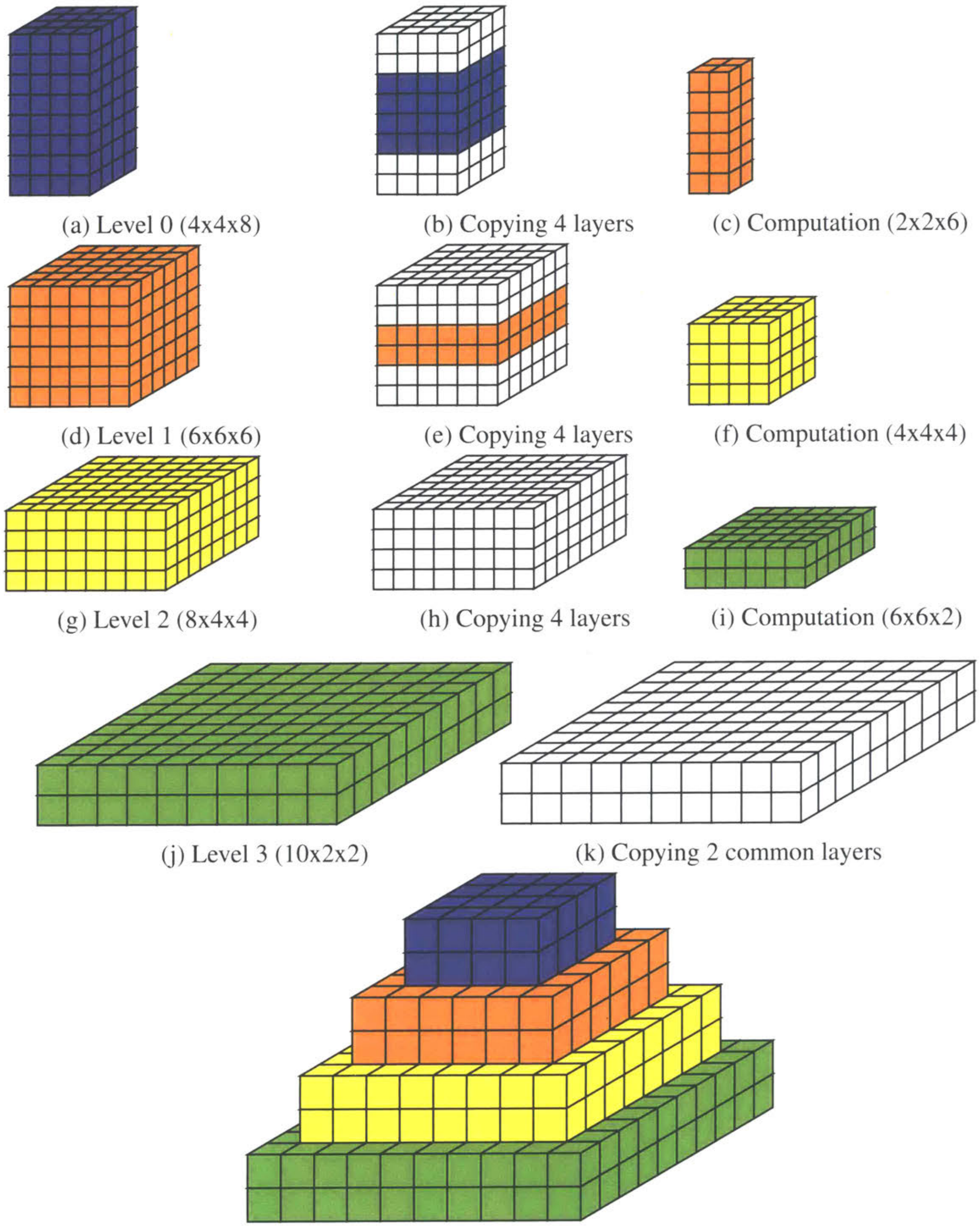
Similar to the Swept 3D beams, these three components also differ only in their orientation and we will call them cores. The Swept 3D core is a four-dimensional, discrete space-time, structure. Similar to the previous Swept 3D components, the three spatial dimensions are discretized with a grid indexed by (i, j, k) and the time dimension is discretized with time steps indexed by L.

The Swept 3D cores have the same time span as the shrinking cube and beams. Thinking of the four-dimensional discrete space-time, the cores represent growing faces of cubes. "Building" the Swept 3D core means calculating all possible space-time values in the cube growing faces. The process starts by linking 2 pair of sides, 4 outputs of Swept 3D beams, and ends with a set of growing cube faces. Figure 5-3 visualize how the Swept 3D core component is built.

Again, depending on the orientation of the constructed core, we call the constructed core Longitudinal, Latitudinal, or Altitudinal. Note that the Latitudinal cores grow spatially in the Y and Z axes. The Longitudinal cores grow in the X and Z axes and finally the Altitudinal cores grow in the X and Y axes.

The outputs of the core construction are 2 four-dimensional structures. A latitudinal core requires North, South, Top and Bottom sides and outputs East and West growing faces. A longitudinal core requires East, West, Top, and Bottom sides and outputs North and South cube growing faces. An Altitudinal core requires East, West, North, and South sides and generates Top and Bottom cube growing faces. Notice the relationship and the pattern between the core orientation, its inputs and its outputs.

The algorithms for building the Latitudinal, Longitudinal, and Altitudinal cores are described in Algorithms 10, 11, and 12 respectively. Just like the previous algorithms, the three core algorithms have two $(n + 2)$ by $(n + 2)$ by $(n + 2)$ internal arrays, $\mathcal{U}$ and $\mathcal{D}$. As mentioned before, n is the side length of the cubic subdomain and if we denote the first time step in the Swept 3D beam as 0, the last time step will be $n/2 - 1$.



(a) Level 0 (4x4x8) (b) Copying 4 layers (c) Computation (2x2x6)

(d) Level 1 (6x6x6) (e) Copying 4 layers (f) Computation (4x4x4)

(g) Level 2 (8x4x4) (h) Copying 4 layers (i) Computation (6x6x2)

(j) Level 3 (10x2x2) (k) Copying 2 common layers

(l) Sample Core Building output

Figure 5-3: Illustrating the building process of the Swept 3D Cores

Algorithm 10: Building The Swept 3D Latitudinal Core

```
$(\mathcal{E}, \mathcal{W})$ = function LatitudinalCore ($\mathbf{St}, \mathcal{N}, \mathcal{S}, \mathcal{T}, \mathcal{B}$)
Input  : St: a list of stencil operations
         $\mathcal{N}, \mathcal{S}, \mathcal{T}, \mathcal{B}$: 4 arrays representing the North, South, Top, and Bottom
         sides of the shrinking Octahedrons
Output: $\mathcal{E}, \mathcal{W}$: 2 arrays representing cube cores
$\mathcal{E} \leftarrow \emptyset, \mathcal{W} \leftarrow \emptyset$ ;
for $L = 0, \dots, \frac{n}{2} - 1$ do
    $\mathcal{D}_{L+1:n-L, n/2-L-1:n/2-L, n/2-L-1:n/2+L+2} \leftarrow \mathcal{N}^L$
    $\mathcal{D}_{L+1:n-L, n/2+L+1:n/2+L+2, n/2-L-1:n/2+L+2} \leftarrow \mathcal{S}^L$
    $\mathcal{D}_{L+1:n-L, n/2-L-1:n/2+L+2, n/2-L-1:n/2-L} \leftarrow \mathcal{T}^L$
    $\mathcal{D}_{L+1:n-L, n/2-L-1:n/2+L+2, n/2+L+1:n/2+L+2} \leftarrow \mathcal{B}^L$
    $\mathcal{E}^L \leftarrow \mathcal{D}_{n-L-1:n-L, n/2-L-1:n/2+L+2, n/2-L-1:n/2+L+2}$   $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}^L$
    $\mathcal{W}^L \leftarrow \mathcal{D}_{L+1:L+2, n/2-L-1:n/2+L+2, n/2-L-1:n/2+L+2}$   $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{W}^L$
    for $k = n/2 - L, \dots, n/2 + L + 1$ do
        for $j = n/2 - L, \dots, n/2 + L + 1$ do
            for $i = 2 + L, \dots, n - L - 1$ do
                $\mathcal{U}_{i,j,k} = \mathbf{St}_L(\{\mathcal{D}_{i',j',k'}, |i' - i| \le 1, |j' - j| \le 1, |k' - k| \le 1\})$
            end
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
```

Algorithm 11: Building The Swept 3D Longitudinal Core

```
$(\mathcal{N}, \mathcal{S})$ = function LongitudinalCore ($\mathbf{St}, \mathcal{E}, \mathcal{W}, \mathcal{T}, \mathcal{B}$)
Input  : St: a list of stencil operations
         $\mathcal{E}, \mathcal{W}, \mathcal{T}, \mathcal{B}$: 4 arrays representing the East, West, Top, and Bottom
         sides of the shrinking Octahedrons
Output: $\mathcal{N}, \mathcal{S}$: 2 arrays representing cube cores
$\mathcal{N} \leftarrow \emptyset, \mathcal{S} \leftarrow \emptyset$ ;
for $L = 0, \dots, \frac{n}{2} - 1$ do
    $\mathcal{D}_{n/2+L+1:n/2+L+2,L+1:n-L,n/2-L-1:n/2+L+2} \leftarrow \mathcal{E}^L$
    $\mathcal{D}_{n/2-L-1:n/2-L,L+1:n-L,n/2-L-1:n/2+L+2} \leftarrow \mathcal{W}^L$
    $\mathcal{D}_{n/2-L-1:n/2+L+2,L+1:n-L,n/2-L-1:n/2-L} \leftarrow \mathcal{T}^L$
    $\mathcal{D}_{n/2-L-1:n/2+L+2,L+1:n-L,n/2+L+1:n/2+L+2} \leftarrow \mathcal{B}^L$
    $\mathcal{N}^L \leftarrow \mathcal{D}_{n/2-L-1:n/2+L+2,L+1:L+2,n/2-L-1:n/2+L+2}$    $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{N}^L$
    $\mathcal{S}^L \leftarrow \mathcal{D}_{n/2-L-1:n/2+L+2,n-L-1:n-L,n/2-L-1:n/2+L+2}$    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}^L$
    for $k = n/2 - L, \dots, n/2 + L + 1$ do
        for $j = 2 + L, \dots, n - L - 1$ do
            for $i = n/2 - L, \dots, n/2 + L + 1$ do
                $\mathcal{U}_{i,j,k} = \mathbf{St}_L(\{\mathcal{D}_{i',j',k'}, |i' - i| \le 1, |j' - j| \le 1, |k' - k| \le 1\})$
            end
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
```

Algorithm 12: Building The Swept 3D Altitudinal Core

```
$(\mathcal{T}, \mathcal{B})$ = function AltitudinalCore ($\mathbf{St}, \mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}$)
Input  : $\mathbf{St}$: a list of stencil operations
         $\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}$: 4 arrays representing the North, South, East, and West
         sides of the shrinking Octahedrons
Output: $\mathcal{T}, \mathcal{B}$: 2 arrays representing cube cores
$\mathcal{T} \leftarrow \emptyset, \mathcal{B} \leftarrow \emptyset$ ;
for $L = 0, \dots, \frac{n}{2} - 1$ do
    $\mathcal{D}_{n/2-L-1:n/2+L+2, n/2-L-1:n/2-L, L+1:n-L} \leftarrow \mathcal{N}^L$
    $\mathcal{D}_{n/2-L-1:n/2+L+2, n/2+L+1:n/2+L+2, L+1:n-L} \leftarrow \mathcal{S}^L$
    $\mathcal{D}_{n/2+L+1:n/2+L+2, n/2-L-1:n/2+L+2, L+1:n-L} \leftarrow \mathcal{E}^L$
    $\mathcal{D}_{n/2-L-1:n/2-L, n/2-L-1:n/2+L+2, L+1:n-L} \leftarrow \mathcal{W}^L$
    $\mathcal{T}^L \leftarrow \mathcal{D}_{n/2-L-1:n/2+L+2, n/2-L-1:n/2+L+2, L+1:L+2}$    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}^L$
    $\mathcal{B}^L \leftarrow \mathcal{D}_{n/2-L-1:n/2+L+2, n/2-L-1:n/2+L+2, n-L-1:n-L}$    $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{B}^L$
    for $k = 2 + L, \dots, n - L - 1$ do
        for $j = n/2 - L, \dots, n/2 + L + 1$ do
            for $i = n/2 - L, \dots, n/2 + L + 1$ do
                $\mathcal{U}_{i,j,k} = \mathbf{St}_L(\{\mathcal{D}_{i',j',k'}, |i' - i| \le 1, |j' - j| \le 1, |k' - k| \le 1\})$
            end
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
```

Now we walk through an example that builds an Altitudinal core for a cubic partition of side length n. The inputs are North, South, East and West sides that were generated by Swept3D Latitudinal and Longitudinal beams. In the first step of the first level, we start by linking the first 2 layers of North, South, East and West sides. This will form a domain partition of size 4 by 4 by 8 at time level $L = 0$ as shown in Figure 5-3 (a). After that, the algorithm takes a copy of the outer 4 layers where the upper 2 go as a Top growing face and the lower 2 go as Bottom growing faces. The copied layers are shown in white in Figure 5-3 (b). The last step in this level is to perform stencil operations for the inner 2 by 2 by 6 partition to move it to time level $L = 1$ as show in Figure 5-3 (c).

On the next level, we load the next 2 layers from our North, South, East, and West inputs and place them to the North, South, East and West of the partition we just generated from the previous level. This will form a partition of size 6 by 6 by 6 at time level $L = 1$ a shown in Figure 5-3 (d). The algorithm then takes a copy of the outer 4 layers. Again, the upper 2 layers go as a Top growing face and the lower 2 layers go as Bottom growing faces. Figure 5-2 (e) illustrates the 4 layers to be copied in white. We end the work in this level by performing stencil operations to the inner 4 by 4 by 4 partition to move it to time level $L = 2$ as showon in Figure 5-3 (f).

Proceeding to the next level and doing similar work as done in the previous 2 levels, this level ends with a 6 by 6 by 2 partition at time level $L = 3$. Figures 5-3 (g-i) summarize the work done at this level. The algorithm ends with loading the last 2 layers of our inputs and placing them appropritely around to the 6 by 6 by 2 partition to form a 10 by 10 by 2 partition and copy this common partition to the Top, and Bottom as growing faces. Figures 5-3 (j-k) clarify this step. The 2 outputs of the algorithm are the stacked Top and Bottom growing faces, which we refer to here as cores. Figure 5-3 (l) demonstrates a sample output for the Bottom growing faces.

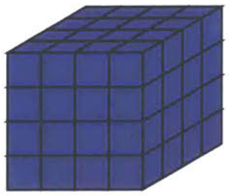
The Latitudinal and Longitudinal Cores can be built in a similar way which is clear from algorithms 10 and 11.

5.2.4 Swept 3D Growing Cube

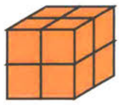
This is the last space-time component of Swept 3D. The Swept 3D core is a four-dimensional, discrete space-time, structure. In this structure, the three spatial dimensions are discretized with a grid indexed by (i, j, k) and the time dimension is discretized with time steps indexed by L.

The Swept 3D Growing cubes do an extra level of stencil operation as it grows. Thinking of the four-dimensional discrete space-time, the growing cubes represent a small cube of size 2 by 2 by 2 which grows with time until it becomes of size n by n by n. "Building" the Swept 3D growing cube means calculating all space-time values in the cube until the entire n by n by n cubic domain is at the same time level. The process starts by linking 3 pair of growing faces, 6 outputs of Swept 3D cores, and ends with an n by n by n cube. Figure 5-4 visualize how the Swept 3D growing cube component is built.

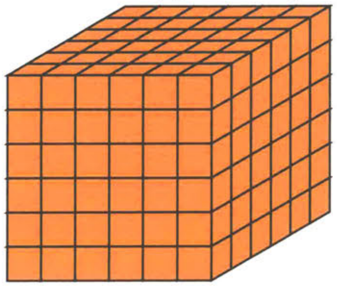
Notice how in this space-time component the generated partition after every level grows in the X, Y, and Z axes. The algorithms for building the growing cube is described in Algorithm 13. Again, this algorithm has two $(n + 2)$ by $(n + 2)$ by $(n + 2)$ internal arrays, $\mathcal{U}$ and $\mathcal{D}$ where n is the side length of the cubic subdomain. If we denote the first time step in the Swept 3D growing cube as 0, the last time step will be $n/2$.



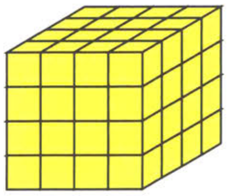
(a) Level 0 (4x4x4)



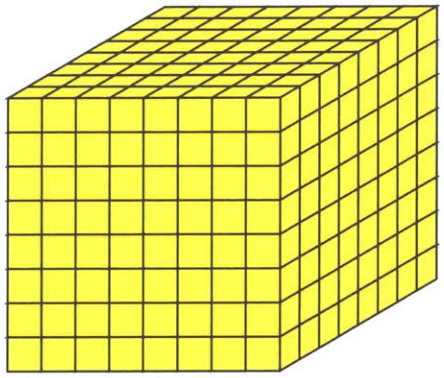
(b) Computation (2x2x2)



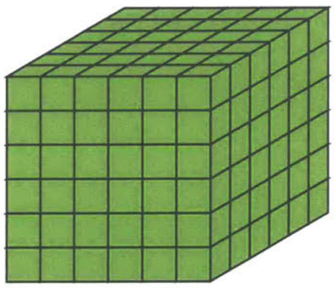
(c) Level 1 (6x6x6)



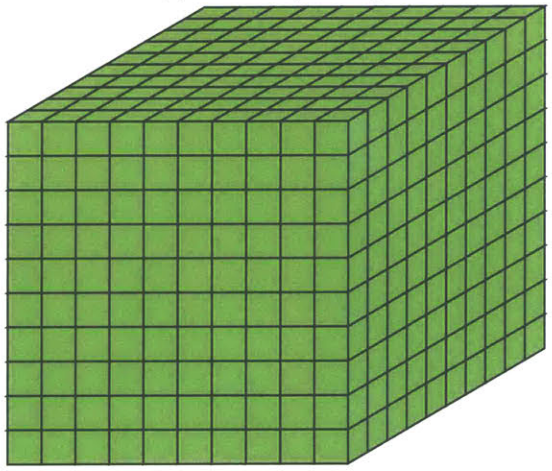
(d) Computation (4x4x4)



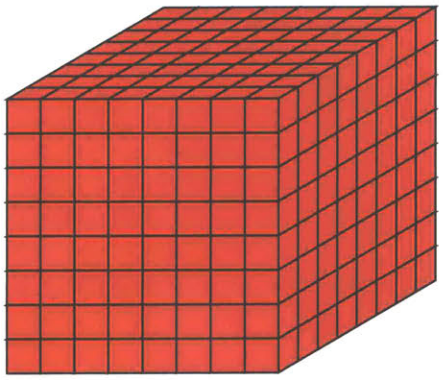
(e) Level 2 (8x8x8)



(f) Computation (6x6x6)



(g) Level 3 (10x10x10)



(h) Computation (8x8x8)

Figure 5-4: Illustrating the building process of the Swept 3D Growing Cube

Algorithm 13: Building The Swept 3D Growing Cube

```
(C) = function GrowingCube (St, $\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}, \mathcal{T}, \mathcal{B}$)
Input  : St: a list of stencil operations
         $\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}, \mathcal{T}, \mathcal{B}$: 6 arrays representing the North, South,
         East, West, Top, and Bottom cores
Output: $C$: an $n$ by $n$ by $n$ array representing the Cubic partition
$C \leftarrow \varnothing$;
for $L = 0, \dots, \frac{n}{2} - 1$ do
    $\mathcal{D}_{n/2-L-1:n/2+L+2, n/2-L-1:n/2-L, n/2-L-1:n/2+L+2} \leftarrow \mathcal{N}^L$
    $\mathcal{D}_{n/2-L-1:n/2+L+2, n/2+L+1:n/2+L+2, n/2-L-1:n/2+L+2} \leftarrow \mathcal{S}^L$
    $\mathcal{D}_{n/2+L+1:n/2+L+2, n/2-L-1:n/2+L+2, n/2-L-1:n/2+L+2} \leftarrow \mathcal{E}^L$
    $\mathcal{D}_{n/2-L-1:n/2-L, n/2-L-1:n/2+L+2, n/2-L-1:n/2+L+2} \leftarrow \mathcal{W}^L$
    $\mathcal{D}_{n/2-L-1:n/2+L+2, n/2-L-1:n/2+L+2, n/2-L-1:n/2-L} \leftarrow \mathcal{T}^L$
    $\mathcal{D}_{n/2-L-1:n/2+L+2, n/2-L-1:n/2+L+2, n/2+L+1:n/2+L+2} \leftarrow \mathcal{B}^L$
    for $k = n/2 - L, \dots, n/2 + L + 1$ do
        for $j = n/2 - L, \dots, n/2 + L + 1$ do
            for $i = n/2 - L, \dots, n/2 + L + 1$ do
                $\mathcal{U}_{i,j,k} = \mathbf{St}_L(\{\mathcal{D}_{i',j',k'}, |i' - i| \le 1, |j' - j| \le 1, |k' - k| \le 1\})$
            end
        end
    end
    $\mathcal{U} \leftrightarrow \mathcal{D}$
end
$C \leftarrow \mathcal{D}$
```

The work done by this algorithm is simple and to show that let us walk through an example that builds a growing cube for a cubic subdomain partition of size 8 by 8 by 8. At the first level, the algorithm load the first 2 layers of the North, South, East, West, Top, and Bottom growing faces, our inputs, to form a 4 by 4 by 4 partition. Notice that this partition is at time level $L = 0$ as shown in Figure 5-4 (a). A stencil operation can now be performed to the inner 2 by 2 by 2 cube to be moved to time level $L = 1$ as shown in Figure 5-4 (b). The work for one level is now done. The next level starts by loading the next growing faces from our inputs and placing them around the 2 by 2 by 2 cube that was generated from the previous level. This will generate a 6 by 6 by 6 partition at time level $L = 1$ as shown Figure 5-4 (c). Stencil operations are performed to the inner 4 by 4 by 4 cube to move it time level $L = 2$ as illustrated in Figure 5-4 (d). Going through next levels in a similar way, the algorithm will end with a cube of size 8 by 8 by 8 at time level $L = 4$. Figures 5-4 (e-h) summarize the reminer of the work done by our growing cube algorithm.

5.3 Connecting The Swept 3D Components

After listing all the algorithms and visually illustrating the Swept 3D components, it is now time to show how these components work with each other to build the Swept Rule in 3D. Carefully looking at the computation stage of each Swept 3D component, one might immediately notice that each grid point involved in the computation is surrounded by a complete set of immediate neighbors. These neighbors represent a 27-point 3D stencil in a 3D numerical discretization scheme.

Assume that we are starting with a cubic computational domain, with triply-periodic boundary conditions, that can be decomposed into 8 cubic-subdomains of equal size. Each of these subdomains is assigned to a different processor. As each processor starts to explicitly solve the PDE, following the domain of influence the domains of dependency to progress without the need to communicate with neighbouring processes, each processor will be building a shrinking cube. The result of this step is 3 pairs of shrinking faces as illustrated in section 5.2.1.

The next stage in Swept 3D will be to build the Longitudinal, Latitudinal, and Altitudinal beams. But before building the beams, data exchange between the parallel processors is required. In order to proceed from stage 1 to stage 2, that is building the beams, each process needs to send data to three of its neighbors and receive data from its other three neighbors. To be more specific, each process will exchange 3 shrinking faces from the shrinking cube that it built in stage 1.

If we agree that each shrinking cube has 6 shrinking faces and those faces can be named as North, South, East, West, Top, and Bottom, then we notice that in order to build a Longitudinal beam, each process needs a set of North and South shrinking faces. On the other hand, to build a Latitudinal beam, each process needs a set of East and West shrinking faces. And for an Altitudinal bean, a set of Top and Bottom faces is needed. For this reason, each process will communicate its North, West, and Top shrinking faces to its North, West, and Top neighbors respectively. Each process will also receive South, East, and Bottom shrinking faces from its South, East, and Bottom neighbors respectively. After this data exchange, each process has what it needs to build 3 beams, which completes stage 2 of Swept 3D. Details about building Swept 3D beams were previously given in section 5.2.2.

Proceeding to stage 3 of Swept 3D will be to build the Swept 3D cores. As the output of each Swept 3D stage feeds into the next stage as input, a second data exchange between processors will be needed to start stage 3 of Swept 3D. Referring back to Swept 3D beams section 5.2.2, we see that building each beam results in 4 of what we call sides. And as each process built 3 beams, each process will have 12 sides in total, where each 2 belong to one direction. To further clarify this, each process will have a set of 2 North sides, a set of 2 South sides and so on for the remaining directions. The communication will be to exchange 6 sides with the neighboring processes. So each process will communicate its 2 North, 2 West, and 2 Top sides to its North, West, and Top neighbors respectively. Each process will also receive 2 South, 2 East, and 2 Bottom sides from its South, East, and Bottom neighbors respectively. After the data is properly exchanged, each process can proceed and build 3 cores. Section 5.2.3 gives all the details about building the Swept 3D cores.

The last stage of Swept 3D will be to build the Swept 3D growing cube which needs 6 sets of growing faces. These growing faces were the result of building the Swept 3D cores in stage 3. So, a third communication between the processors needs to be done to exchange the growing faces in order to processed with this last stage of Swept 3D. This time, each process will communicate its North, West, and Top growing faces to its North, West, and Top neighbors respectively. Also, each process will receive South, East, and Bottom growing faces from its South, East, and Bottom neighbors respectively. After the data is properly exchanges, each process can build a growing cube. Details about building the growing cube are listed in section 5.2.4.

At the end of this stage, the entire computational domain is at a consistent state. Meaning that all blocks, grid points or cells, are at the same time level. However, the domain arrangement has changed as a result a shift caused by the triply-periodic boundary condition assumed at the beginning. We call what we did so far a half Swept 3D cycle.

The other half of the Swept 3D cycle can be performed in the same way, except that the data exchange between the processes will happen between the other three neighbors. For example, instead of sending the North, West, and Top shrinking faces of the shrinking cube to North, West, and Top neighbors, we send the South, East, and Bottom shrinking faces of the shrinking cube to the South, East, and Bottom neighbors. The same is applicable to the rest of Swept 3D components. At the end of the second half of the Swept 3D cycle, the domain will have its original arrangement.

So, a complete Swept 3D cycle, requires a total of 6 communications to take place. 3 communications happen at each half of the Swept 3D cycle. Notice how starting with an initial condition as time level 0, and working with cubic subdomains of 12 by 12 by 12 allows a complete Swept 3D cycle to promote the computational domain 12 time levels involving only 6 communications between the processors.

We listed earlier algorithms for building the Swept 3D components, now we show the algorithm that connects these components together making them construct the Swept Rule in 3D. Going through Algorithm 14 clarifies many points such as what to communicate and what is needed to build each Swept 3D component. As we did in Swept 2D algorithm, we abbreviated the direction of the components that get exchanged between the processors by this first letter in the direction as capital. Moreover, we used the "$\odot\!\!\rightarrow$" symbol to denote sending to a process and the "$\leftarrow\!\!\odot$" to denote receiving from a process. We also abbreviated the neighboring processes with "$\mathbb{P}$" symbol combined with the direction of that process. For example, $\mathbb{P}_N$ means the North neighboring process. So, in algorithm 14, a line that reads $\mathcal{N}_{\mathcal{P}} \odot\!\!\rightarrow \mathbb{P}_N[\mathcal{N}_{\mathcal{P}}]$, simply means sending the value of array N_P to the North process, which should receive it in an array called N_P .

Algorithm 14: The Swept 3D

```
(C) = function Swept3D (St, ℛ, C)
Input  : St: a list of stencil operations
         C: an n by n by n array representing a cubics subdomain
         ℛ: The number of Swept 3D cycles to perform
Output: C: The cubic subdomain partition after applying ℛ * n subtimesteps
for c = 1, ..., ℛ do
    $\mathcal{N}_\mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{E}_\mathcal{P}, \mathcal{W}_\mathcal{P}, \mathcal{T}_\mathcal{P}, \mathcal{B}_\mathcal{P} \leftarrow$ ShrinkingCube (St, $C$)
    $\mathcal{N}_\mathcal{P} \odot\!\!\rightarrow \mathbb{P}_N[\mathcal{N}_\mathcal{P}]$;   $\mathcal{W}_\mathcal{P} \odot\!\!\rightarrow \mathbb{P}_W[\mathcal{W}_\mathcal{P}]$;   $\mathcal{T}_\mathcal{P} \odot\!\!\rightarrow \mathbb{P}_T[\mathcal{T}_\mathcal{P}]$
    $\mathcal{N}_\mathcal{P} \leftarrow\!\!\odot \mathbb{P}_S[\mathcal{N}_\mathcal{P}]$;   $\mathcal{W}_\mathcal{P} \leftarrow\!\!\odot \mathbb{P}_E[\mathcal{W}_\mathcal{P}]$;   $\mathcal{T}_\mathcal{P} \leftarrow\!\!\odot \mathbb{P}_B[\mathcal{T}_\mathcal{P}]$
    $\mathcal{N}_\S, \mathcal{S}_\S, \mathcal{T}_\S, \mathcal{B}_\S \leftarrow$ LatitudinalBeam (St, $\mathcal{W}_\mathcal{P}, \mathcal{E}_\mathcal{P}$)
    $\mathcal{E}_\dagger, \mathcal{W}_\dagger, \mathcal{T}_\dagger, \mathcal{B}_\dagger \leftarrow$ LongitudinalBeam (St, $\mathcal{S}_\mathcal{P}, \mathcal{N}_\mathcal{P}$)
    $\mathcal{N}_\ddagger, \mathcal{S}_\ddagger, \mathcal{E}_\ddagger, \mathcal{W}_\ddagger \leftarrow$ AltitudinalBeam (St, $\mathcal{B}_\mathcal{P}, \mathcal{T}_\mathcal{P}$)
    $\mathcal{N}_\S, \mathcal{N}_\ddagger \odot\!\!\rightarrow \mathbb{P}_N[\mathcal{N}_\S, \mathcal{N}_\ddagger]$;   $\mathcal{W}_\dagger, \mathcal{W}_\ddagger \odot\!\!\rightarrow \mathbb{P}_W[\mathcal{W}_\dagger, \mathcal{W}_\ddagger]$;   $\mathcal{T}_\S, \mathcal{T}_\dagger \odot\!\!\rightarrow \mathbb{P}_T[\mathcal{T}_\S, \mathcal{T}_\dagger]$
    $\mathcal{N}_\S, \mathcal{N}_\ddagger \leftarrow\!\!\odot \mathbb{P}_S[\mathcal{N}_\S, \mathcal{N}_\ddagger]$;   $\mathcal{W}_\dagger, \mathcal{W}_\ddagger \leftarrow\!\!\odot \mathbb{P}_E[\mathcal{W}_\dagger, \mathcal{W}_\ddagger]$;   $\mathcal{T}_\S, \mathcal{T}_\dagger \leftarrow\!\!\odot \mathbb{P}_B[\mathcal{T}_\S, \mathcal{T}_\dagger]$
    $(\mathcal{E}_C, \mathcal{W}_C) \leftarrow$LatitudinalCore (St, $\mathcal{S}_\ddagger, \mathcal{N}_\ddagger, \mathcal{B}_\dagger, \mathcal{T}_\dagger$)
    $(\mathcal{N}_C, \mathcal{S}_C) \leftarrow$LongitudinalCore (St, $\mathcal{W}_\ddagger, \mathcal{E}_\ddagger, \mathcal{B}_\S, \mathcal{T}_\S$)
    $(\mathcal{T}_C, \mathcal{B}_C) \leftarrow$AltitudinalCore (St, $\mathcal{S}_\S, \mathcal{N}_\S, \mathcal{W}_\dagger, \mathcal{E}_\dagger$)
    $\mathcal{N}_C \odot\!\!\rightarrow \mathbb{P}_N[\mathcal{N}_C]$;   $\mathcal{W}_C \odot\!\!\rightarrow \mathbb{P}_W[\mathcal{W}_C]$;   $\mathcal{T}_C \odot\!\!\rightarrow \mathbb{P}_T[\mathcal{T}_C]$
    $\mathcal{N}_C \leftarrow\!\!\odot \mathbb{P}_S[\mathcal{N}_C]$;   $\mathcal{W}_C \leftarrow\!\!\odot \mathbb{P}_E[\mathcal{W}_C]$;   $\mathcal{T}_C \leftarrow\!\!\odot \mathbb{P}_B[\mathcal{T}_C]$
    $(C) \leftarrow$GrowingCube (St, $\mathcal{S}_C, \mathcal{N}_C, \mathcal{W}_C, \mathcal{W}_C, \mathcal{B}_C, \mathcal{T}_C$)
    $\mathcal{N}_\mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{E}_\mathcal{P}, \mathcal{W}_\mathcal{P}, \mathcal{T}_\mathcal{P}, \mathcal{B}_\mathcal{P} \leftarrow$ ShrinkingCube (St, $C$)
    $\mathcal{S}_\mathcal{P} \odot\!\!\rightarrow \mathbb{P}_S[\mathcal{S}_\mathcal{P}]$;   $\mathcal{E}_\mathcal{P} \odot\!\!\rightarrow \mathbb{P}_E[\mathcal{E}_\mathcal{P}]$;   $\mathcal{B}_\mathcal{P} \odot\!\!\rightarrow \mathbb{P}_B[\mathcal{B}_\mathcal{P}]$
    $\mathcal{S}_\mathcal{P} \leftarrow\!\!\odot \mathbb{P}_N[\mathcal{S}_\mathcal{P}]$;   $\mathcal{E}_\mathcal{P} \leftarrow\!\!\odot \mathbb{P}_W[\mathcal{E}_\mathcal{P}]$;   $\mathcal{B}_\mathcal{P} \leftarrow\!\!\odot \mathbb{P}_T[\mathcal{B}_\mathcal{P}]$
    $\mathcal{N}_\S, \mathcal{S}_\S, \mathcal{T}_\S, \mathcal{B}_\S \leftarrow$ LatitudinalBeam (St, $\mathcal{W}_\mathcal{P}, \mathcal{E}_\mathcal{P}$)
    $\mathcal{E}_\dagger, \mathcal{W}_\dagger, \mathcal{T}_\dagger, \mathcal{B}_\dagger \leftarrow$ LongitudinalBeam (St, $\mathcal{S}_\mathcal{P}, \mathcal{N}_\mathcal{P}$)
    $\mathcal{N}_\ddagger, \mathcal{S}_\ddagger, \mathcal{E}_\ddagger, \mathcal{W}_\ddagger \leftarrow$ AltitudinalBeam (St, $\mathcal{B}_\mathcal{P}, \mathcal{T}_\mathcal{P}$)
    $\mathcal{S}_\S, \mathcal{S}_\ddagger \odot\!\!\rightarrow \mathbb{P}_S[\mathcal{S}_\S, \mathcal{S}_\ddagger]$;   $\mathcal{E}_\dagger, \mathcal{E}_\ddagger \odot\!\!\rightarrow \mathbb{P}_E[\mathcal{E}_\dagger, \mathcal{E}_\ddagger]$;   $\mathcal{B}_\S, \mathcal{B}_\dagger \odot\!\!\rightarrow \mathbb{P}_B[\mathcal{B}_\S, \mathcal{B}_\dagger]$
    $\mathcal{S}_\S, \mathcal{S}_\ddagger \leftarrow\!\!\odot \mathbb{P}_N[\mathcal{S}_\S, \mathcal{S}_\ddagger]$;   $\mathcal{E}_\dagger, \mathcal{E}_\ddagger \leftarrow\!\!\odot \mathbb{P}_W[\mathcal{E}_\dagger, \mathcal{E}_\ddagger]$;   $\mathcal{B}_\S, \mathcal{B}_\dagger \leftarrow\!\!\odot \mathbb{P}_T[\mathcal{B}_\S, \mathcal{B}_\dagger]$
    $(\mathcal{E}_C, \mathcal{W}_C) \leftarrow$LatitudinalCore (St, $\mathcal{S}_\ddagger, \mathcal{N}_\ddagger, \mathcal{B}_\dagger, \mathcal{T}_\dagger$)
    $(\mathcal{N}_C, \mathcal{S}_C) \leftarrow$LongitudinalCore (St, $\mathcal{W}_\ddagger, \mathcal{E}_\ddagger, \mathcal{B}_\S, \mathcal{T}_\S$)
    $(\mathcal{T}_C, \mathcal{B}_C) \leftarrow$AltitudinalCore (St, $\mathcal{S}_\S, \mathcal{N}_\S, \mathcal{W}_\dagger, \mathcal{E}_\dagger$)
    $\mathcal{S}_C \odot\!\!\rightarrow \mathbb{P}_S[\mathcal{S}_C]$;   $\mathcal{E}_C \odot\!\!\rightarrow \mathbb{P}_E[\mathcal{E}_C]$;   $\mathcal{B}_C \odot\!\!\rightarrow \mathbb{P}_B[\mathcal{B}_C]$
    $\mathcal{S}_C \leftarrow\!\!\odot \mathbb{P}_N[\mathcal{S}_C]$;   $\mathcal{E}_C \leftarrow\!\!\odot \mathbb{P}_W[\mathcal{E}_C]$;   $\mathcal{B}_C \leftarrow\!\!\odot \mathbb{P}_T[\mathcal{B}_C]$
    $(C) \leftarrow$GrowingCube (St, $\mathcal{S}_C, \mathcal{N}_C, \mathcal{W}_C, \mathcal{W}_C, \mathcal{B}_C, \mathcal{T}_C$)
end
```

5.4 A simplified performance analysis of the swept rule in 3D

To simplify our analysis, let us consider a half Swept 3D cycle. We will also assume that if we partition the computational domain into small cubic subdomains, bandwidth issues are not encountered during communications. Meaning that, Communication between computing nodes takes time τ, regardless of how much data is communicated.

Now, if we partition our computational domain into cubic sub-domains of side length n, then within a process, each Swept 3D half cycle will perform $\frac{n}{2}$ substeps for n^3 grid points, which is the volume of each subdomain. We know that each Swept 3D half cycle requires 3 communications between the computation processes. Let us assume that each physical compute node contains a single MPI process, then for a half Swept 3D cycle we will encounter 3 communication latencies between our compute nodes.

Let us break the half Swept 3D cycle to its basic components and try to understand how the 3 communication latencies are distributed among these components. In half a cycle, we build one Shrinking Cube, three Beams, three Cores, and finally a Growing Cube. Swept3D encounters one communication latency before switching from one space-time component to the next.

Knowing that half Swept 3D cycle performs $\frac{n}{2}$ sub-timesteps for n^3 grid points and ignoring any internal overhead caused by the internal implementation of Swept 3D , we can calculate the time needed to perform each sub-timestep as follows:

$$[(n^3 s)\frac{n}{2} + 3\tau]/\frac{n}{2} \tag{5.1}$$

Simplifying Equation 5.1, we get:

$$n^3 s + \frac{6\tau}{n} \tag{5.2}$$

Computing node – FLOPS	FLOP per step-point	Computing time per step-point (s)
Single thread Intel Nehalem 10 GFLOPS	8000 (FE system)	1200 ns
Single thread Intel Nehalem 10 GFLOPS	400 (FV system)	60 ns
Single thread Intel Nehalem 10 GFLOPS	6 (FD scalar)	0.9 ns
Oak Ridge 2017 Summit node 40 TFLOPS	8000 (FE system)	300 ps
Oak Ridge 2017 Summit node 40 TFLOPS	400 (FV system)	15 ps
Oak Ridge 2017 Summit node 40 TFLOPS	6 (FD scalar)	225 fs

Table 5.4: The typical time it takes to compute one sub-timestep on a single spatial point for solving PDEs in 3 spatial dimensions

Now we present a simplified theoretical performance model for the Swept Rule in 3D similar to that we presented for the Swept Rule in a single and two space dimensions[3]. To understand how fast the swept rule in 3D is, we need to know the typical values of τ and s.

For τ we will use the same values given in table 3.1. For s, we will use the values listed in table 5.4. Table 5.4 attempts to estimate the range of s by covering the typical f for solving PDEs in a three spatial dimensions, as well as the highest and lowest F on a modern computing node.

With these values of τ and s, the plot in Figure 5-5 shows, according to our sub-timespte cost formula and ignoring overhead cost, how fast the Swept 3D scheme runs as a function of n, the spatial points per node. The up-sloping, dashed and dash-dot lines represent the $n^3 s$ term in our cost formula for different values of s, and the down-sloping, solid lines represent the $\frac{6\tau}{n}$ term for different values of τ. For each combination of s and τ, the total time per sub-timestep as a function of n, plotted as thin black curves in Figure 5-5, can be found by summing the corresponding up-sloping and down-sloping lines.

The total time per sub-timestep can be minimized by choosing n. This minimizing n can be found, for each τ and s. The optimal value of n represent the limit of scaling. Above this optimum, decreasing n by scaling to more nodes would decrease the total time per sub-

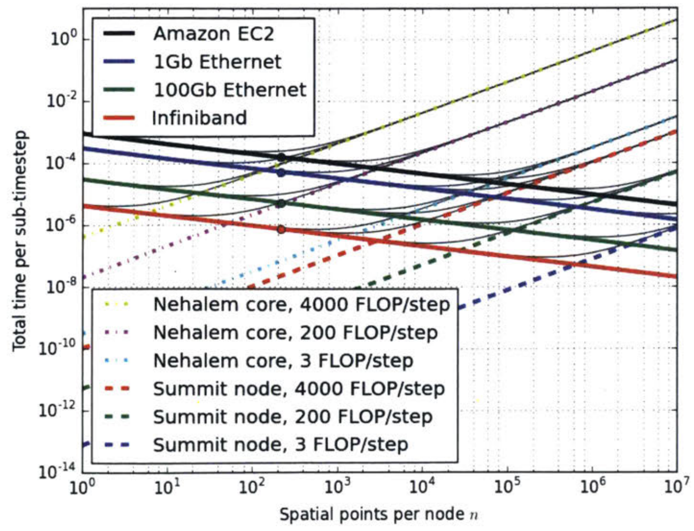


Figure 5-5: Analyzing the Swept 3D performance using communication latency and CPU FLOP/step

timestep, accelerating the simulation. But decreasing n beyond the optimum by scaling to even more nodes would not accelerate, but slow down the simulation.

At the optimal n, the swept rule 3D almost always breaks the latency barrier. A method that requires communication every sub-timestep takes at least τ per sub-timestep.

5.5 Interface and implementation of the swept scheme in 3D

The interface we present for our implementation of the Swept Rule in 3D is very simple and similar to that given for Swept 1D and 2D[3]. As usual, we present a data initialization function and a time-stepping function.

The input variables to the initialization function are the global "i", "j", and "k" indicies for each spacial point and a spacial point structure representing a 3D stencil. The following pseudo code exemplifies the initialization function interface:

```
Init(pointIndex i,pointIndex j,pointIndex k,spacialPoint3D p)
{
    Based on the location of the point (i,j,k), set its initial data
        p.inputs[0] = variable 0 initial value
        p.inputs[1] = variable 1 initial value
        p.inputs[n] = variable n initial value
}
```

The second function in our interface is where the PDE solve is performed. The input variables to the timestepping function are the index of which sub-timestep to be executed and a 3D 27-point stencil spacial point structure. The following psedocode shows the timestepping function of our interface.

```
timestep(timestep i,spacialPoint3D p)
{
    Based on the value of i, perform the proper operation on p

}
```

5.6 Verifying the Swept rule in 3D – solution of the 3D Heat Equation

To verify our Swept 3D algorithms and confirm that our implementation is correct, we tested our implementation by solving the three-dimensional heat diffusion equation. Our PDE configuration was periodic boundary conditions and an initial heat source equals 1 located at the center of the domain. The verification of our solution is shown in Figure 5-6.

The experiment was conducted in a cubic domain of size 32 by 32 by 32 which was decomposed into 8 subdomains using 8 MPI processes each containing a 16 by 16 by 16 cube. The whole purpose behind this run was to verify the design of the Swept 3D algorithms and confirm that the communication between the Swept 3D components is correct. The behavior of the heat diffusion in the obtained solution certifies the corrections of our design and implementation of Swept 3D.

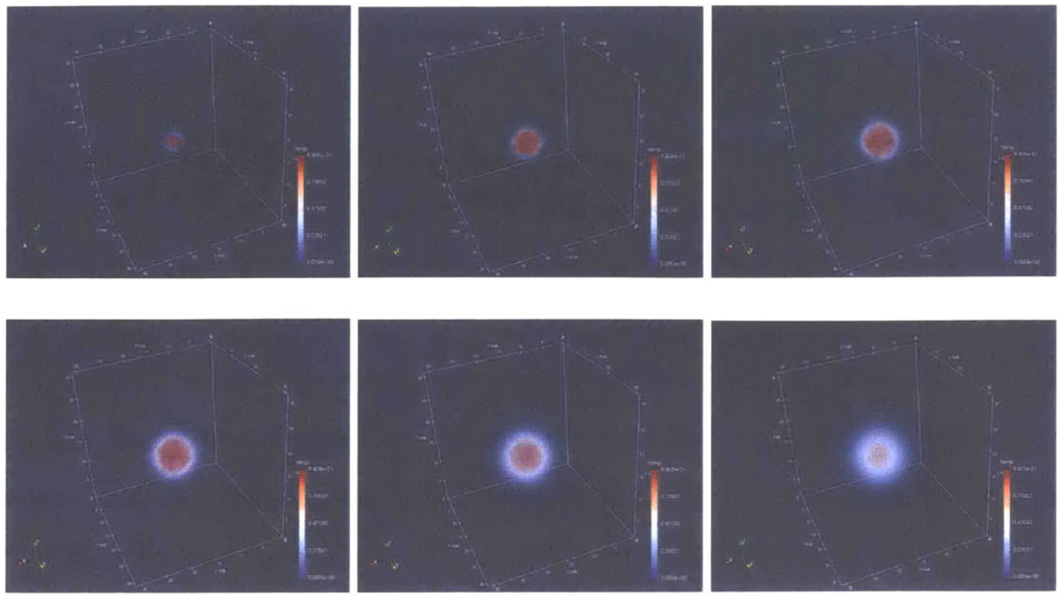


Figure 5-6: 3D Contour plots showing the diffusion of the Heat. This verifies our Swept 3D implementation

An experiment was conducted in a small 8-node Amazon EC2 cluster. StartCluster was used to form the 8-node cluster with an EC2 instance type of "c3.large" and an Amazon Machine Image (AMI) of "ami-6b211202"[2]. A single MPI process was assigned to each node. Figure 5-7 plots in log scale the performance comparison between the straight "Classical" and Swept domain decomposition. Notice how the Swept Rule in 3D gains a speed-up factor of more than 2 when we assign about 1000 grid points per MPI process. Our implementation of the Swept Rule based solver of the 3D Heat equation can be found at:`https://github.com/hubailmm/Heat3D`

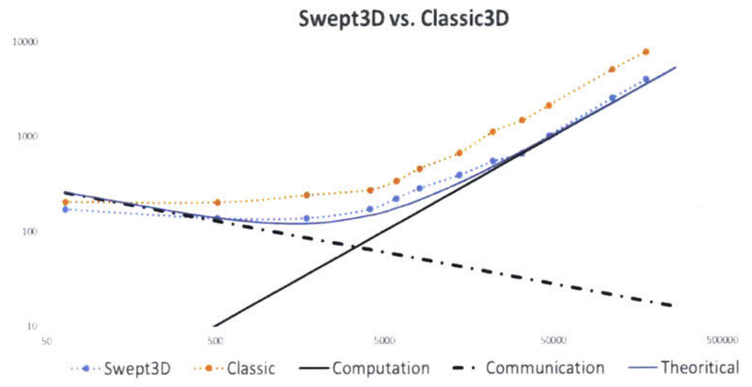


Figure 5-7: The performance of Straight and Swept 2D domain partitioning schemes when solving the 3D heat equation. The x-axis represents the number of spacial points assigned to each process. The y-axis represents the time, in microseconds, needed to perform a timestep. The black up-sloping, dashed line represents the $n^3 s$ term in Equation 4.2, and the black down-sloping, solid line represents the $\frac{6\tau}{n}$ term in the Equation

Chapter 6

Conclusion

This thesis introduced the swept rule for decomposing space and time in solving PDEs. The swept rule breaks the latency barrier, advancing each sub-timestep in a fraction of the time it takes for a message to travel from one computing node to another.

In our experiments with the swept rule in 1D, solving the Kuramoto-Sivashinsky equation on Amazon EC2, over 15 sub-timesteps can be integrated during each latency time. In another experiment with the 1D Euler equation on an Ethernet-based cluster, about 10 sub-timesteps can be integrated during each latency time.

The swept rule in 2D also broke the latency barrier, advancing each sub-timestep in a fraction of the time it takes for a message to travel from one computing node to another. In our experiment with the 2D Wave and Euler equations on Amazon EC2, Swept 2D had a speed-up factor of more than 3 compared to the classical straight way of domain decomposition. Other experiments in a dedicated cluster showed that, over 1Gb Ethernet, Swept 2D can gain speed-up factors of more than 8 compared to classical space-only partitioning schemes.

Experiments with the swept rule in 3D solving the 3D heat diffusion PDE, swept rule 3D broken the latency barrier and achieved a speed-up factor of more than 2 compared to space-only based decomposition.

To examine how fast the swept space-time decomposition rule can be, we conducted a simplified theoretical analysis. The analysis shows that time integration can be made faster not only by reducing latency, but also by beefing up each node. 1 nanosecond per time step

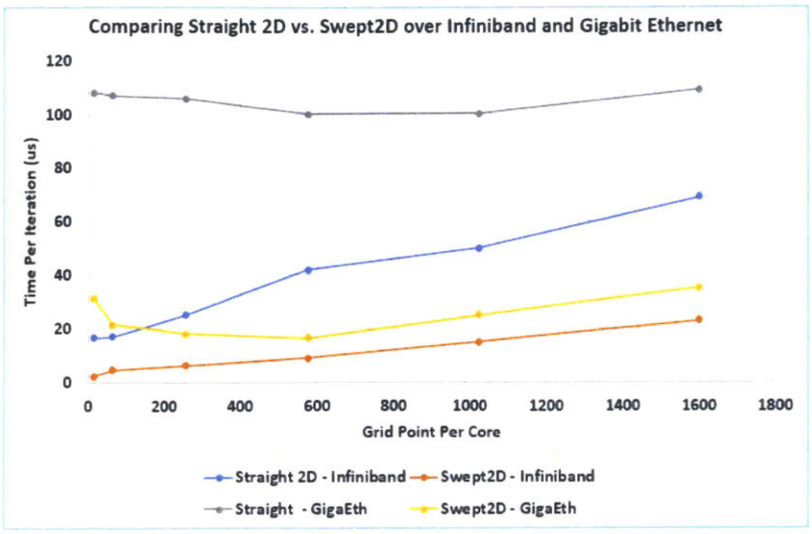


Figure 6-1: Comparing Swept2D and Stright2D partitioning on 1Gb Ethernet and 56Gb Infiniband

may be possible with currently commissioned hardware, but only if the the power of each node can be fully utilized.

As it is known that during the development of numerical simulation applications, the concerns of numerical engineers are usually very different from those of software engineers or computer scientists, the swept decomposition rule can be implemented through a standard interface, which separates numerical schemes from its computational implementation. The interface allows different numerical schemes to use the same implementation of the swept decomposition rule. It is also easy to switch between different implementations sharing the same interface.

We also emphasize that our space-time partitioning algorithms are efficient even when used in professional HPC clusters. Such clusters have low latency interconnects between their nodes and one may not expect space-time partitioning to be necessary. Our experiments showed that our Swept algorithms are not just faster than classical over low latency networks, our algorithms can achieve low latency HPC cluster level performance over cheap high-latency 1Gb Ethernet connected clusters. Our results for some experiments solving the linear system that arises from Poisson's 2D equation using Jacobi iterative linear solver are given in Figure 6-1.

6.1 Future Work

As far as our future work is concerned, our first priority is to extend the Swept Rule to explicitly solve PDEs in unstructured grids and utilize Co-processor, such as Nvidia GPUs to further accelerate the Swept Rule. We also plan to utilize the Swept Rule in implicit PDE solvers. It is not difficult to realize that the Swept rule, presented in this thesis, can be used as a fixed-point iterative liner solver, such as the Jacobi linear solver. This can be done when the time dimension is thought of as iteration. We did some initial tests solving the linear system arising from the 2D Poisson PDE. The initial results show that Swept 2D had a speed-up factor of more than 10 compared to a space-only parallel Jacobi iterative linear solver. Knowing that the linear system solve, is usually, the bottleneck of most implicit PDE solvers, could allow the Swept Rule to boost the speed of implicit PDE solvers.

Bibliography

[1] Exascale computing study: Technology challenges in achieving exascale systems (tr-2008-13). Accessed: 2015-04-01.

[2] Starcluster. Accessed: 2015-04-01, Office of Educational Innovation and Technology, MIT.

[3] M. Alhubail and Q. Wang. The swept rule for breaking the latency barrier in time advancing pdes. *JCP*, 307:110–121, 2016.

[4] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32:866–901, 2011.

[5] I. Bermejo-Moreno, J. Bodart, J. Larsson, B. Barney, J. Nichols, and S. Jones. Solving the compressible navier-stokes equations on up to 1.97 million cores and 4.1 trillion grid points. 2013.

[6] I. Bertolacci, C. Olschanowsky, B. Harshbarger, B. Chamberlain, D. Wonnacott, and M. Strout. Parameterized diamond tiling for stencil computations with chapel parallel iterators. 2015.

[7] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, and M. Valero. The international exascale software project: A call to cooperative action by the global high-performance community. *The International Journal of High Performance Computing Applications*, 32:309–322, 2009.

[8] M. Emmett and M.L Minion. Toward an efficient parallel in time method for partial differential equations. *Communications in Applied Mathematics and Computational Science*, 7:105–132, 2012.

[9] R.D Falgout, S. Friedhoff, TZ.V Kolev, S.P Maclachlan, and J.B Schroder. Parallel time integration with multigrid. *PAMM*, 14, 2014.

[10] C. Farhat, J. Cortial, C.Dastillung, and H. Bavestrello. Time-parallel implicit integrators for the near-real-time prediction of linear structural dynamic responses. *International Journal for Numerical Methods in Engineering*, 67:697–724, 2006.

[11] M. Gander. *Multiple Shooting and Time Domain Decomposition Methods*, chapter 50 Years of Time Parallel Time Integration, pages 69–113. Springer International Publishing, Switzerland, 2015.

[12] G.Brauckmann, C. Streett, W. Kleb, S. Alter, K. Murphy, and C. Glass. Computational and experimental unsteady pressures for alternate sls booster nose shapes. *AIAA SciTech*, 2015.

[13] F. Ham. Pngwriter. Accessed: 2015-04-01.

[14] T. Ishii, editor. *Handbook of Microwave Technology, Volume 1: Components and Devices*. Academic Press, San Diego, California, 1995.

[15] J. Joo and P. Durbin. Simulation of turbine blade trailing edge cooling. *Journal of Fluids Engineering*, 131, 2009.

[16] A. Khabou, J. Demmel, L. Grigori, and M. Gu. Lu factorization with panel rank revealing pivoting and its communication avoiding version. *SIAM Journal on Matrix Analysis and Applications*, 34:1401–1429, 2013.

[17] J. Lions, Y. Maday, and G. Turinici. A parareal in time discretization of pde's. *Comptes Rendus Mathematique*, 332:661–668, 2001.

[18] T. MALAS, G. HAGER, H. LTAIEF, H. STENGEL, G. WELLEIN, and D. KEYES. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. 2014.

[19] V. Rao and A. Sandu. An adjoint-based scalable algorithm for time-parallel integration. *Journal of Computational Science*, 5:76–84, 2014.

[20] J. Reiss and J. Sesterhenn. A conservative, skew-symmetric finite difference scheme for the compressible navier-stokes equations. *Computers and Fluids*, 101:208–219, 2014.

[21] S. Roy, S. Kapadia, and J. Heidmann. Film cooling analysis using des turbulence model. *ASME*, 6:1113–1122, 2003.

[22] Y. Tang, R. Chowdhury, B. Kuszmaul, C.Luk, and C.Leiserson. The pochoir stencil compiler. pages 117–128, 2011.

[23] Q. Wang. Decomposition of stencil update formula into atomic stages. 2016.

[24] Q. Wang, S.A Gomes, P.J Blonigan, A.L Gregory, and E.Y Qian. Towards scalable parallel-in-time turbulent flow simulations. *Physics of Fluids*, 25, 2013.

[25] P. Wolf, R. Balakrishnan, G. Staffelbach, L. Gicquel, and T. Poinsot. Using les to study reacting flows and instabilities in annular combustion chambers. *Flow, Turbulence and Combustion*, 88:191–206, 2012.