Performance Engineering of Directional Message-Passing Algorithms Through a Stencil-Based Approach for Applications in Molecular Dynamics

by

Isabel Rosa

B.S. Mathematics and Computer Science and Engineering,
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author . . . .
Department of Electrical Engineering and Computer Science
May 12, 2022

Certified by . . . .
Tao B. Schardl
Research Scientist
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Performance Engineering of Directional Message-Passing Algorithms Through a Stencil-Based Approach for Applications in Molecular Dynamics

by

Isabel Rosa

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The molecular dynamics method, used by scientists across the fields of physics, materials science, and biology, is an increasingly popular way to simulate particle interactions. Current implementations of molecular dynamics simulators can verify macromolecular structures, examine atomic-level phenomena that cannot be observed directly, and predict the behavior of unstudied proteins. The existing implementations, however, rely on inefficient directional message-passing algorithms on graph neural networks. This thesis presents a novel approach for the optimization of these algorithms using a stencil-like technique. The stencil-based algorithm, called StencilMD, provides both the benefits of parallelization and improved cache locality. The results show that StencilMD successfully reduces the amount of time required to run a molecular dynamics simulation by as much as 28.57% with a corresponding 26.92% decrease in cache misses.

Thesis Supervisor: Tao B. Schardl
Title: Research Scientist

Acknowledgments

I would like to thank my advisor, Tao B. Schardl, for all his help throughout the development of this thesis. Beyond providing invaluable technical guidance, he greatly aided in the improvement of my writing and presentation skills. I am grateful for his support, expertise, and plethora of interesting anecdotes.

I would also like to thank Charles E. Leiserson for inspiring me to pursue this field of research and dropping in on the occasional meeting with various entertaining stories and pearls of wisdom, related or not to the topic at hand.

Thank you as well to Dionysios Sema for providing me with valuable insight into the world of molecular dynamics, as well as pre-trained models and data sets on which to perform my experiments.

Finally, a major thank you goes to my friends and family who have supported me all this time. I send special thanks to my parents for always hanging my work up on the refrigerator, even when it requires several magnets.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Molecular dynamics simulators are powerful tools that produce crucial information for the study of protein structures [5], the research of nanotechnologies [18], and the development of new materials [20]. Since many of the phenomena of interest in the fields of physics, materials science, and biology occur on the atomic level, it can be hard for scientists to perform and observe the experiments they need. Sometimes, the objects of study may not even exist in the physical world yet, making direct experiments impossible. Molecular dynamics simulators solve both of those problems. While these simulators, however, can and do provide valuable information for all sorts of applications, they still have two main areas for improvement: accuracy and speed. In this thesis, I investigate novel methods for reducing the runtimes of the complex computational methods used in molecular dynamics simulators.

As the field of molecular dynamics exists today, the state-of-the-art simulators have advanced to use graph neural networks (GNNs). These GNNs are trained to accurately predict the energy and forces of a given system of atoms [3, 16, 24]. While using the laws of quantum mechanics to directly calculate the forces and energy of an atomic system would result in higher accuracy than using a GNN to approximate those interactions, even the most efficient true quantum mechanical approaches are far too expensive computationally to handle the large systems of atoms that most applications require [27]. Hence, modern simulators estimate the desired forces and energies by training GNNs on quantum mechanical data to allow for faster calculations post-training. In a graph used by such a GNN, often called an ***atom graph***, the nodes represent atoms and edges connect atoms that are within a certain distance, called the ***cutoff radius***, of one another. To accurately model continually changing atomic interactions on the atom graph, molecular dynamics simulators make use of directional message-passing GNN algorithms [16]. For every step of the simulation, the state of each atom and its neighbors must be updated using empirical interatomic potentials. Directional message-passing GNN algorithms seek to accurately approximate those interatomic potentials by allowing each atom to feel the forces from the other atoms in its neighborhood.

This approach, while faster than calculating the true effects of quantum mechanics, is still prohibitively slow. Modern simulators are so computationally intensive that it is only feasible to simulate systems over a timescale of microseconds [15]. Even when only dealing with microseconds of simulation time, the CPU-time required for the simulators to run can range from days to years. While these simulators provide highly accurate results over the microsecond range, it is an open problem that they cannot simulate longer timescales and that it takes so long to get results for the feasible timescales.

This thesis studies multicore CPU parallelization and cache-optimization of directional message-passing GNN algorithms used in the context of molecular dynamics in order to improve the speed at which simulations can be run. By applying a stencil-like approach, inspired by Frigo and Strumpen [8], I developed and analyzed a new molecular dynamics simulation algorithm, called StencilMD, to take advantage of improved cache locality and parallelism that results in up to a 28.57% reduction in runtime and 26.92% reduction in cache misses as compared to the normal looping routine.

Challenges with GNN Parallelization

A key challenge of this problem lies in the complexity of parallelizing a system where the state of each element depends on the states of other elements. For example, two nodes, representing atoms, that share an edge in the atom graph cannot both send update messages at the same time since the update messages from one would change the update messages from the other. Thus, in order to obtain deterministic and accurate results, naive parallelization does not suffice, and the problem of parallelizing molecular dynamics simulations necessitates a more clever approach. One possible technique might involve fine-grained locking on the atom graph, as described in Chapter 6. Chapter 6 also describes alternate approaches for the parallelization of graph computations such as chromatic scheduling [14], priority-DAG scheduling [28], and the use of ghost atoms [26]. These approaches, however, each have their downsides, inspiring me to investigate a new parallelization technique.

In this thesis, I explore a stencil-based approach, called StencilMD, that has several advantages over alternative methods such as fine-grained locking. Whereas fine-grained locking incurs costly overhead and nondeterminism, StencilMD avoids such issues by using a lock-free algorithm that deterministically subdivides the GNN into sub-graphs that can be run in parallel without conflict. Because this approach subdivides the GNN deterministically, StencilMD achieves deterministic results, an important benefit in contrast with a fine-grained locking approach. Not only does this approach reap the benefits of determinism and parallelism with lower synchronization overhead as compared to locks, but StencilMD also exploits cache locality for an even sharper decrease in runtime. For reasons related to graph connectivity, however, StencilMD is not generally applicable to GNNs with any architecture. To differentiate between general GNNs, where StencilMD may not work, and GNNs for applications in molecular dynamics, where StencilMD does work, I use the term ***MDGNN*** to refer to a GNN designed for molecular dynamics. I discuss the key insight that allows this technique to work for MDGNNs in Chapter 2.

Summary of Contributions

To evaluate this approach in practice, I implemented a serial molecular dynamics simulator, called LoopMD, in C++ using the NequIP MDGNN model [3] to predict interatomic potentials and the Nosé-Hoover thermostat [6] to simulate the physics of atomic movement given the predicted potentials. As discussed in Chapter 2, however, the use of the NequIP model and Nosé-Hoover thermostat are merely design choices, and StencilMD applies to general MDGNNs and physics engines. Using LoopMD as a reference, I then implemented StencilMD, as described in Chapter 4, using the same MDGNN model and physics simulator. Compared to LoopMD, I found that StencilMD sped up the simulation by 28.57% with a corresponding 26.92% decrease in last level cache miss percentages, measured on an input system with 2600 atoms simulated over 5 timesteps. I discuss these results in more detail in Chapter 5.

In summary, this thesis makes the following contributions:

- The algorithms involved in a StencilMD, a novel, stencil-based algorithm for the performance engineering of directional message-passing MDGNN algorithms with an application in molecular dynamics,
- A working prototype implementation of StencilMD,
- An empirical analysis of the runtime and cache performance of StencilMD as compared to LoopMD, and
- A discussion of possible extensions to StencilMD to further improve parallelism and cache locality.

The remainder of this thesis is organized as follows. Chapter 2 provides further background on the topics of MDGNNs and stencil algorithms, as well as details of the specific systems I use in my work. In Chapter 3, I discuss the reformulation of the molecular dynamics problem to permit a stencil-based approach. I describe the complete idea behind StencilMD and its implementation in Chapter 4 along with an analysis of the assumptions and design choices involved in StencilMD. Chapter 5 contains my full results and their implications. Chapter 6 puts my work into broader context through the lens of related work, and Chapter 7 discusses concluding thoughts as well as various paths forward for future research on this topic.

Chapter 2

Background

This chapter presents background on both the use of MDGNNs in molecular dynamics simulations and the idea behind the stencil computation, formulated by Frigo and Strumpen [8], that inspired StencilMD. I begin with an overview of the components required to create a successful molecular dynamics simulator and my implementation of these components, including the details of how the MDGNN is involved. I further specify the computation performed by the MDGNN, as well as the assumptions and implications of those mechanics. Finally, I provide a high-level description of Frigo and Strumpen's stencil algorithm, which I refer to as FSStencil, and its typical use cases.

GNNs for Molecular Dynamics (MDGNNs)

In the molecular dynamics simulation regime, the input takes the form of a system of atoms and a number of timesteps to simulate, and the simulator outputs the resulting state of that system after the desired number of timesteps. With input and output defined, I now discuss the common steps that state-of-the-art molecular dynamics simulators perform to convert input into output.

A full state-of-the-art molecular dynamics simulator requires several high-level components: the atom graph component, the calculator component, and the physics engine component. The atom graph component of the simulation performs the first step in going from input to output; it takes a system of atoms as input and converts that system into graph form. The simulator then passes the resulting atom graph into the calculator component, which uses some sort of model to output the predicted energies and forces of the given atoms. In the molecular dynamics simulators with which my work is concerned, the calculator component uses an MDGNN. Finally, the simulator sends the atom graph and the predicted energies and forces to the physics engine component, where the energies and forces are used to perform the necessary calculations to update the states of the atoms to reflect the passage of one timestep. This involves updating atomic attributes such as position and velocity. The simulator then repeats this three-step process until the desired number of timesteps have been simulated, and the resulting state of the system contains the desired output. I now discuss the specifics of these three components, focusing on the calculator component and its relationship to MDGNNs, and conclude with a discussion of my implementation of these components.

Atom Graphs

Atom graphs are designed to mimic atomic interactions by connecting a pair of atoms with an edge if the atoms in the pair exert forces upon one another. Since atomic-level forces decay rapidly as distance increases, it suffices to connect each atom with all other atoms within a certain distance, called the cutoff radius, in order to create a reasonable model for atomic interaction.

Atom graphs display three key features related to StencilMD. First, atom graphs represent a physical system, namely, atoms in 3-dimensional space. Second, atom graphs have nice connectivity properties by design through a reasonable choice of cutoff radius. I discuss the importance of this connectivity property for StencilMD in Chapter 4. Finally, atom graphs must be dynamic in order to model a system of atoms that move over time.

The question of how best to represent dynamic graphs has challenged researchers for decades, but my work sidesteps that issue entirely. Since the main bottleneck of modern molecular dynamics simulators lies in the calculator component, I chose the most naive approach for representing a dynamic graph; StencilMD simply rebuilds the atom graph from scratch after each step of the simulation. The dynamic nature of the atom graph is not the focus of this thesis, so I defer all further discussion on the topic of dynamic graph representation to Chapter 7.

MDGNN-Based Calculators

Besides the structure of the atom graph, StencilMD also relies on the nature of the computations performed by the MDGNN in the calculator component. My implementation of StencilMD uses the NequIP model specifically [3], so I will use NequIP as an example to describe the sorts of computations performed by MDGNNs. This is merely an implementation design choice, however, and any other MDGNN is also compatible with StencilMD.

Before discussing the importance of MDGNN architecture, I first define the terms "graph convolution" and "k-hop neighborhood". A ***graph convolution*** is a computation on an undirected graph (V, E) that updates the state of a vertex $v \in V$ by aggregating information from all vertices in a certain neighborhood of v. I write this neighborhood as the "k-hop neighborhood" of v for some value of k which depends on the application. For an undirected graph (V, E), the ***k-hop neighborhood*** of a vertex $v \in V$ is defined as follows:

$$k\text{-hop}(v) = \begin{cases} v & k = 0 \\ (k-1)\text{-hop}(v) \cup \{v_1 \in V \mid (v_1, v_2) \in E \text{ and } v_2 \in (k-1)\text{-hop}(v)\} & k > 0. \end{cases}$$

In other words, for $k > 0$, the k-hop neighborhood of vertex v is defined as union of the $(k-1)$-hop neighborhood of v and the neighbors of the vertices in the $(k-1)$-hop neighborhood of v.

I will now discuss locality, the key feature, for the purpose of this thesis, of MDGNN architectures as compared to general GNN architectures. In terms of locality, the crucial difference between general GNNs and MDGNNs lies in the fact that MDGNN architectures consist of just a limited number of graph convolution layers and layers that act only on individual atoms. The NequIP architecture, for example, consists of 3 convolutional layers and 2 self-interaction layers. The implication of this feature for locality is as follows: to calculate the energies and forces of a given atom v, the model does not need to know about the entire graph. Rather, the model only requires knowledge about the other atoms in the k-hop neighborhood of v, where k equals the number of convolutional layers. For NequIP, then, $k = 3$. Now, since edges connect atoms within the cutoff radius r of each other, the k-hop neighborhood of atom v can only include atoms within distance $k \cdot r$ of v. I define ***locality*** as this property of MDGNNs that the state of a given atom v can only depend on atoms within distance $k \cdot r$ of v, where k is the number of convolutional layers in the MGDNN. I leverage this locality to subdivide the atom graph for parallel processing, as described further in Chapters 3 and 4.

MDGNNs also display many other interesting properties to ensure invariance and equivariance to the desired transformations. Similarly, MDGNNs are often designed to predict energies directly, and then perform a step of gradient backpropagation to derive forces from energies in a manner that guarantees energy conservation. While all these features are interesting and important both for the accuracy and runtime of molecular dynamics simulations, my work makes no attempt to optimize the inner workings of MDGNNs, so I again defer further discussion of the details of MDGNN computations to Chapter 7. For the remainder of this thesis, I will treat MDGNNs, including the NequIP model, as black boxes that have the desired property of locality and correctly calculate the energies and forces of a given atom graph.

Molecular Dynamics Physics Engines

The physics engine composes the final part of a molecular dynamics simulator. StencilMD does not rely on the structure of this component in any way.

Implementation

To aid with the discussion of my implementation of the components of a molecular dynamics simulation, a brief pseudocode description of each component can be found

```
ATOM-GRAPH(positions, r)
1  // Create atom graph. Not implementation specific.
2  return edges of the atom graph created from positions and r

CALCULATOR(positions, types, edges)
1  // Calculate forces using MDGNN. Not model specific.
2  return forces of each atom

PHYSICS-ENGINE(positions, types, forces)
1  // Calculate new positions. Not physics engine specific.
2  return new position of each atom after the passage of one timestep

BASECASE(positions, types, r)
1  // Perform one step of molecular dynamics simulation.
2  edges = ATOM-GRAPH(positions, r)
3  forces = CALCULATOR(positions, types, edges)
4  new-positions = PHYSICS-ENGINE(positions, types, forces)
5  return new-positions
```

Figure 2-1: Pseudocode formalizing the inputs of each component of a molecular dynamics simulator, as well as a brief description of return values.

in Figure 2-1, which I will now describe in greater detail through an analysis of each step of BASECASE.

The main implementation choice of ATOM-GRAPH, the first step of BASECASE, is the graph representation. STENCILMD does not rely on any specific graph representation, but my implementation uses the edge list representation due to its compact form when representing a sparse graph.

After ATOM-GRAPH, the next step of BASECASE involves a call to CALCULATOR. The edges of the atom graph constitute one of the arguments of the call to CALCULATOR, emphasizing the importance of the ATOM-GRAPH subroutine. As discussed above, STENCILMD relies on the MDGNN used in this step to contain only layers of graph convolutions and layers of self-interaction, as is typical in the state of the art for MDGNNs [3, 16, 24], but has no dependencies on which specific model is used. My implementation uses the NequIP model due to its promising results in terms of

```
LoopMD(positions, types, r, t)
1  for $i = 0$ to $t$
2      positions = Basecase(positions, types, r)
3  return positions
```

Figure 2-2: The formalization of LoopMD in pseudocode.

accuracy [3].

The final step in calculating the new positions of the atoms is the Physics-Engine subroutine. The importance of the Calculator subroutine is shown in the call to Physics-Engine, where the forces felt by each atom are crucial to compute updated positions. StencilMD has no dependencies on the specifics of the physics engine at all. My implementation makes use of the Nosé-Hoover thermostat due to its prevalence in modern-day molecular dynamics [6].

Using the subroutines in Figure 2-1, I now formally define LoopMD, the serial looping routine for molecular dynamics, through the pseudocode in Figure 2-2. Using calls to Basecase on line 2 of Figure 2-2, LoopMD repeatedly runs the atom graph step, the calculator step, and the physics engine step in serial on the whole system of atoms until completing the desired number of timesteps. I will use LoopMD as the baseline algorithm for a molecular dynamics simulator and refer to it for the purpose of comparison with StencilMD in the following chapters.

Trapezoidal Stencil Computations

StencilMD is inspired by the parallel cache-oblivious stencil algorithm by Frigo and Strumpen [8], which I refer to as FSStencil. The classic stencil computation arises in a scenario where stationary elements in a multidimensional grid are updated according to a fixed pattern called a ***stencil***. These updates are performed in a sequence of sweeps across the grid, often referred to as timesteps. The end result of a full stencil computation is the new state of each element in the grid after the desired number of timesteps have been completed. To update a given element, the stencil relies on the states of that element and its neighbors to compute the next state for that element. This reliance on other elements is called a ***data dependency***. Researchers have studied and developed many variations of parallel stencil computations, but I have chosen to focus on FSSTENCIL [8]. FSSTENCIL boasts both the benefits of parallelization as well as cache obliviousness, both in theory and in practice, and generalizes to n-dimensional space. To familiarize the reader with this approach, I now provide a high-level overview of the multi-dimensional version of FSSTENCIL [8] along with an example in 2-dimensional space.

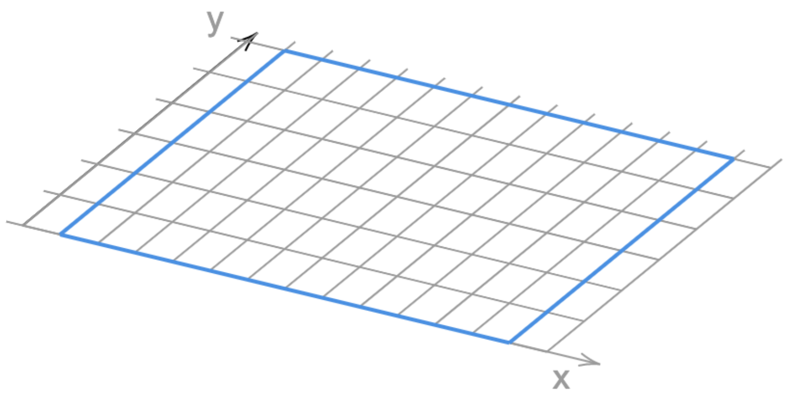


Figure 2-3: A diagram showing the straightforward representation of a 2-dimensional stencil computation in space. Each cell of the grid contains one element to be updated by the stencil calculation.

In FSSTENCIL, the first key idea is to modify the dimensionality of the problem to leverage the regular structure of a stencil computation. The most straightforward way to think of an n-dimensional stencil computation involves creating an n-dimensional grid enclosing the elements of interest. The cells of that grid, each of which represents one element in the stencil computation, are then updated after completing the calculations necessary to perform one timstep. In Figure 2-3, the blue box outlines this n-dimensional grid representation of a stencil computation when $n = 2$. FSSTENCIL, however, represents a series of timesteps over an n-dimensional space as an $n + 1$-dimensional space, where the extra dimension is time. In this representation, a 2-dimensional stencil computation over a space with dimensions $X \times Y$ becomes a 3-dimensional hypergrid with dimensions $X \times Y \times T$, where T is the desired number

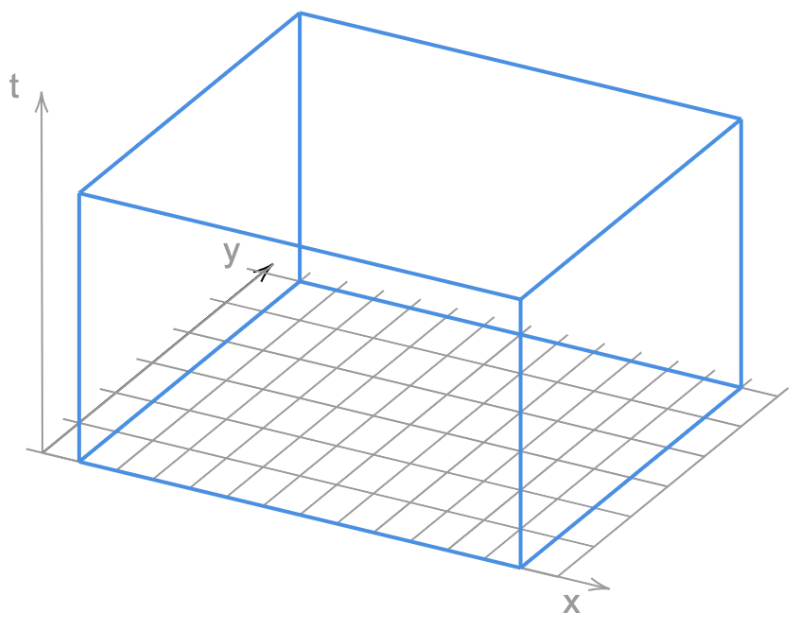


Figure 2-4: A diagram showing the spacetime representation of a 2-dimensional stencil computation. Now, each cell contains one element at the timestep corresponding with its position in the time dimension.

of timesteps to perform. Then, the problem becomes an issue of using each layer of the hypergrid, representing a timestep, to calculate the next layer. In Figure 2-4, the blue box outlines this spacetime representation of the stencil computation.

To leverage cache locality and parallelize the stencil computation, FSStencil recursively subdivides the spacetime hypergrid in a way that respects the data dependencies of the stencil by using time cuts and space cuts. A ***time cut*** splits the hypergrid into two pieces, one of which covers the first T_1 timesteps, and the second of which covers the last $T - T_1$ timesteps. In Figure 2-5, the yellow box, labeled τ_1, represents the hypergrid covering the first T_1 timesteps, and the uncolored box, labeled τ_2, represents the hypergrid covering the last $T - T_1$ timesteps. By performing the stencil computation on τ_1 and then τ_2 in order, FSStencil ensures that at each timestep, all the information needed to compute the next timestep has already been calculated. The time cut alone, however, provides no benefit as compared to a normal looping routine since the two pieces must be performed serially. For this reason, FSStencil also makes use of space cuts.

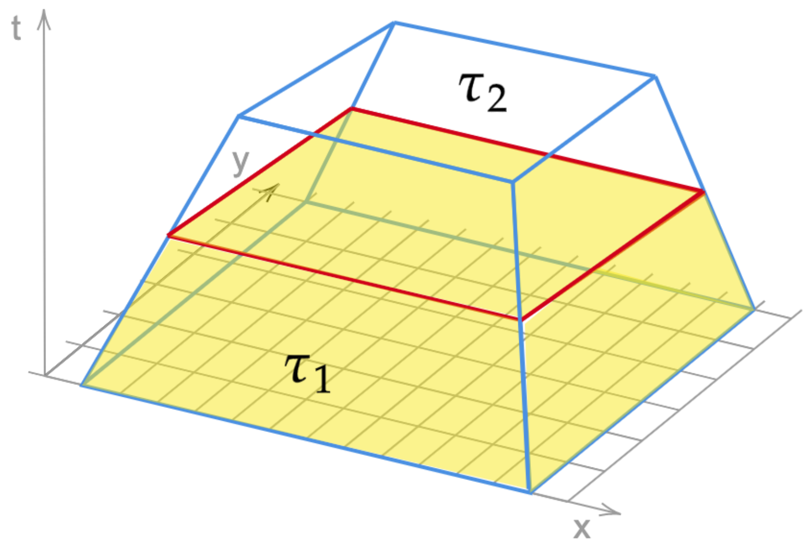


Figure 2-5: A diagram showing a time cut in 3-dimensional spacetime. The trapezoid resulting from the time cut that must be processed first is labeled τ_1, and the trapezoid that must be processed second is labeled τ_2.

A ***space cut*** can be performed in any of the n space dimensions by creating a cut along that dimension in space through the layers of time. By creating a subdivision in space, however, FSSTENCIL must take care not to violate the data dependencies of the stencil. Since each resulting hypergrid can only see the states of its own elements, the elements along the boundary of the space cut can no longer be properly updated due to the fact that the required information about their neighbors is not present in the new hypergrid. Thus, as time proceeds in each hypergrid, the number of elements that can correctly be updated must shrink as a result of the space cut, as diagrammed in Figure 2-6 through the shape of the spacetime hypergrid where the base is larger than the top. Since each hypergrid should be correct without any dependence on other hypergrids, the hypergrids resulting from space cuts end up taking on the form of trapezoidal prisms due to this shrinking phenomenon. In Figure 2-6, trapezoidal prisms τ_1 and τ_2, highlighted in yellow and green respectively, represent the resulting sections of spacetime that can be correctly computed as a result of the space cut. Since there are no data dependencies between τ_1 and τ_2 by design, those trapezoidal prisms can be computed in parallel. To complete the desired stencil computation,

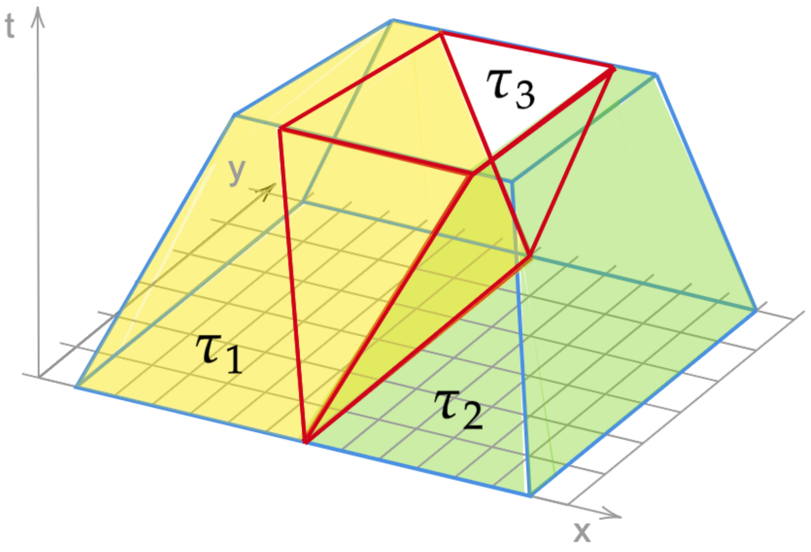


Figure 2-6: A diagram showing a space cut in 3-dimensional spacetime. The resulting left and right trapezoids are labeled τ_1 and τ_2 respectively, whereas the middle trapezoid is labeled τ_3.

however, the final piece, represented by the uncolored trapezoidal prism τ_3 in Figure 2-6, must be calculated after τ_1 and τ_2, since the information needed to fill in τ_3 has not fully been computed until τ_1 and τ_2 have been filled in. By computing the resulting trapezoidal prisms in that order, the space cut respects the data dependencies of the stencil computation.

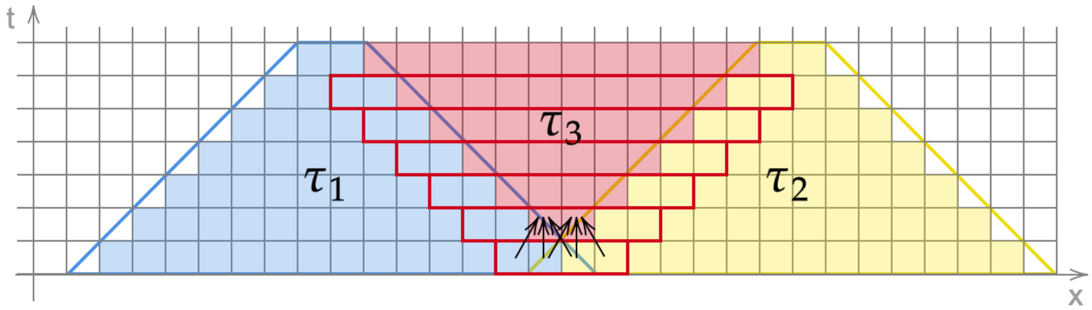


Figure 2-7: A diagram showing the information required to fill in the middle trapezoid. Each row highlighted in red corresponds with the box outlined in red the timestep before. A box outlined in red contains all the data dependencies of its corresponding red row, as diagrammed explicitly in the first row through black arrows.

In order to fill in the middle trapezoid resulting from a space cut, labeled τ_3 in the example from Figure 2-6, the output of FSStencil when called on the left (τ_1) and right (τ_2) trapezoids must return enough information to satisfy the data dependencies of all the elements in the middle trapezoid. To best illustrate the information needed to fill in τ_3, I will refer to Figure 2-7, which provides an example stencil computation in 2-dimensional spacetime in which the data dependencies of each element are its neighbors. Since the state of an element at a timestep t depends on the state of that element and its neighbors in timestep $t-1$, each row of τ_3 in Figure 2-7 that occurs at timestep t and spans the space $[x_0, x_1]$ in the x-dimension relies on the information spanning the space $[x_0 - 1, x_1 + 1]$ in timestep $t-1$. In Figure 2-7, each red row of τ_3 corresponds with the box outlined in red in the row prior. For each red row, the corresponding box outlined in red contains the elements necessary to fill in that red row. This relationship is diagrammed explicitly for the first row-box pair through the use of arrows to represent data dependencies. The diagram shows, then, that in order to fill in τ_3, it does not suffice to merely return the top row of atoms for each trapezoid. Interestingly, the diagram also shows that it does not suffice to return the last correctly computed state of each element either. To fill in τ_3, FSStencil must return the last correctly computed state of each element, as well as the data dependencies of those elements.

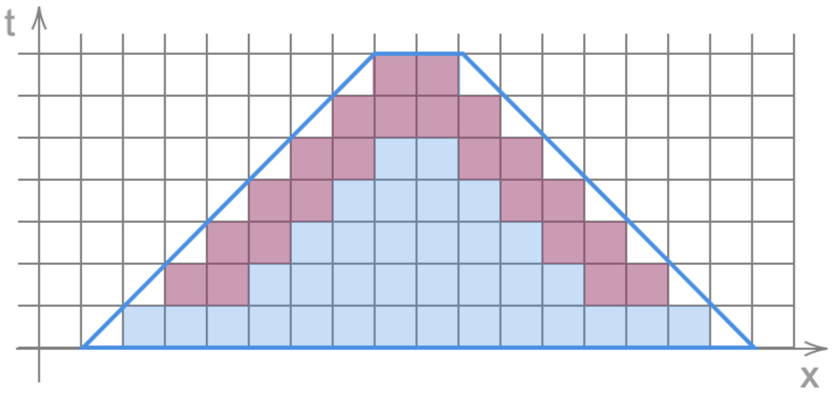


Figure 2-8: A diagram highlighting the output of FSStencil when given a trapezoid with the same data dependencies as in Figure 2-7.

To summarize succinctly through an example, the outputs to FSStencil, given the blue trapezoid in Figure 2-8 with the same data dependencies as in Figure 2-7, are the cells highlighted in red in Figure 2-8. Since the bottom row of the trapezoid is part of the input, FSSTENCIL does not need to return any information about that bottom row; all the important information about the bottom row is already known or can be deduced directly. For that reason, nothing in the bottom row is highlighted red.

```
FSSTENCIL($\tau$)
 1  if $\tau.height == 1$
 2        perform stencil calculation on $\tau$
 3  else
 4        if $\tau$ is wide enough
 5              // Perform space cut.
 6              $\tau_1$ = left-hand trapezoidal prism
 7              $\tau_2$ = right-hand trapezoidal prism
 8              $\tau_3$ = middle trapezoidal prism
 9              spawn FSSTENCIL($\tau_1$)
10              spawn FSSTENCIL($\tau_2$)
11              sync
12              fill in $\tau_3$ using $\tau_1$, $\tau_2$, and the stencil calculation
13        else
14              // Perform time cut.
15              $\tau_1$ = bottom trapezoidal prism
16              $\tau_2$ = top trapezoidal prism
17              FSSTENCIL($\tau_1$)
18              FSSTENCIL($\tau_2$)
```

Figure 2-9: A description of FSSTENCIL in pseudocode.

FSSTENCIL combines both time cuts and space cuts to achieve its performance results, which I illustrate with a discussion of the pseudocode in Figure 2-9. To decide when to perform a time cut and when to perform a space cut, Frigo and Strumpen first specify the meaning of a well-defined trapezoidal prism: a trapezoidal prism is ***well-defined*** when the number of elements in that trapezoidal prism that can be correctly updated at the final timestep is nonzero. Now, when the given trapezoidal prism is too tall to perform a space cut in any dimension, meaning that any space cut would produce an ill-defined trapezoidal prism, FSSTENCIL performs a time cut, as shown in line 13 of FSSTENCIL. If the given trapezoidal prism is wide enough in any dimension that a space cut can be performed in that dimension, meaning that the space cut will produce well-defined trapezoidal prisms, FSStencil performs that space cut on line 4. This process repeats until it reaches the base case, where the height of the trapezoidal prism is 1, shown in line 1 of FSStencil. This represents the case where there is only one timestep to perform. In this situation, FSStencil directly calculates the updates corresponding to the singular timestep using the stencil.

Not only does FSStencil reap the benefits of parallelization, but it also makes use of cache locality to reduce the cache miss count, further increasing performance. Frigo and Strumpen show that for a stencil computation in n-dimensional space, FSStencil incurs at most $\mathcal{O}(p/Z^{1/n})$ cache misses on a machine with an ideal cache of size Z for a stencil calculation that contains p spacetime points [8]. Besides this provably good cache performance, the other key point to note in terms of cache locality is that the elements in the base case of FSStencil fit into cache. The trapezoidal prisms decompose space into sections where all the elements reside near each other in cache, incurring fewer cache misses than in a regular looping routine. The benefits of the reduced cache misses in FSStencil are clear not only through the theoretical bounds, but also in practice [25]. In Chapter 5, I demonstrate that StencilMD obtains similar empirical benefits from parallelization and cache locality as FSStencil.

Chapter 3

Reformulating Molecular Dynamics

This chapter presents a reformulation of the molecular dynamics problem that permits a stencil-based way of thinking. I begin by describing the high-level idea behind StencilMD and its similarities and differences to FSStencil. Next, I provide the details of trapezoid representation, followed by a discussion of the exact input and output required by StencilMD.

High-Level Idea

At a high level, StencilMD follows roughly the same steps as FSStencil. As in FSStencil, StencilMD begins with a dimensionality change to create a nice relationship between space and time. I begin by defining the spatial dimensions of a system as the smallest bounding box that contains all of the atoms in the system, as seen in Figure 3-1. Then, instead of treating a molecular dynamics simulation as a question of updating various characteristics of atoms in a dynamic 3-dimensional bounding box over a series of timesteps, I rephrase the molecular dynamics problem as a question of filling in a 4-dimensional shape, where the fourth dimension is time. While this dimension change is difficult to visualize, I attempt such a visualization in Figure 3-2, where stacked bounding boxes represent subsequent states of the system as time passes. With this reformulation of the problem, a timestep now refers to a step forwards in the time dimension, and each new timestep can be calculated using the previously computed sections of spacetime.

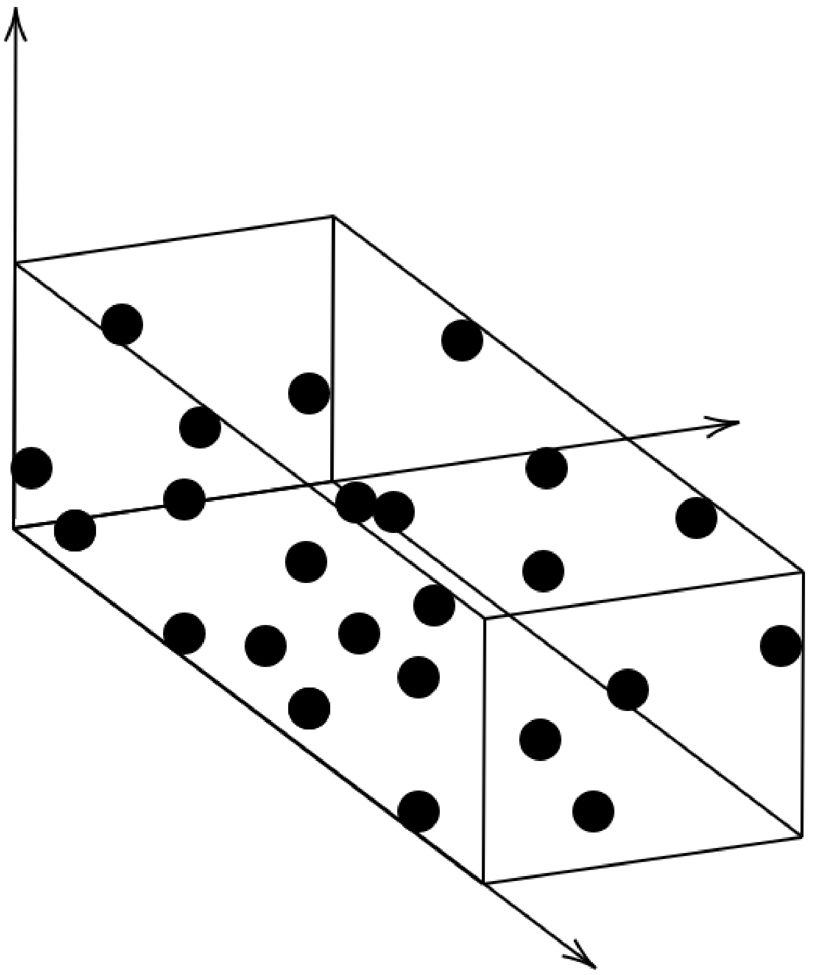


Figure 3-1: A diagram showing the spatial dimensions of a system of atoms in 3-dimensional space.

After the dimensionality change, I continue to follow in the footsteps of Frigo and Strumpen by treating the resulting 4-dimensional box as a trapezoidal prism that can be subdivided for parallel processing through cuts in any of the four spacetime dimensions. I use their terminology of a time cut to refer to the process of splitting the trapezoidal prism along the time dimension, resulting in two smaller trapezoidal prisms that must be processed in series. Figure 3-3 shows the two trapezoidal prisms, τ_1 and τ_2, that result from a time cut in the system from Figure 3-2. Similarly, a space cut refers to the process of splitting the trapezoidal prism along any of the three space dimensions. Whichever space dimension is chosen, the space cut results

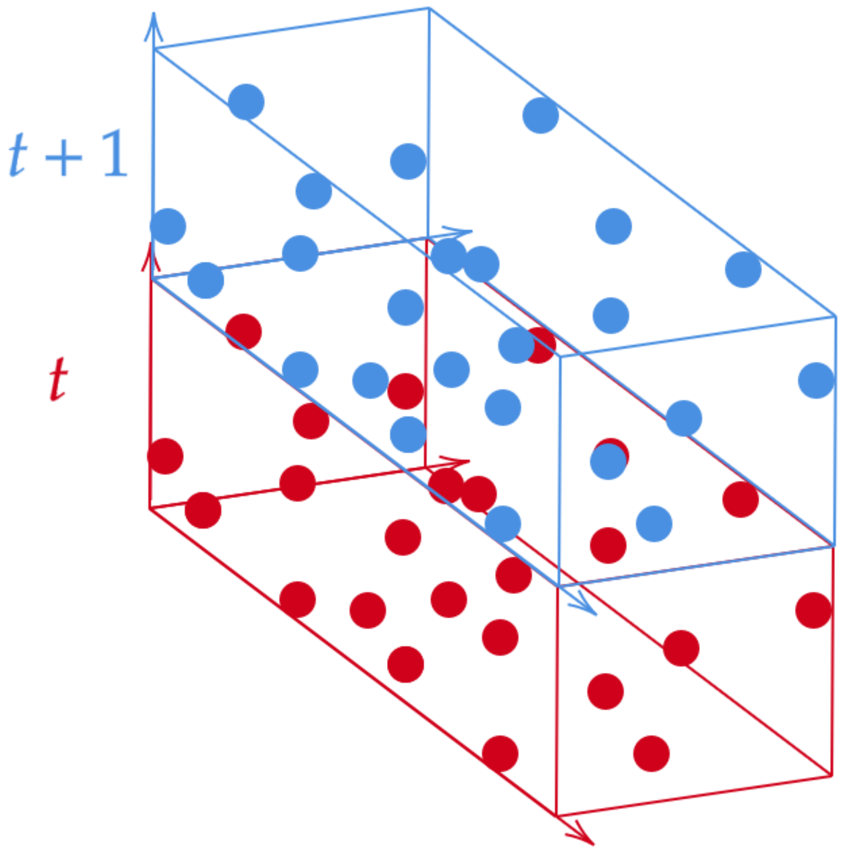


Figure 3-2: A diagram showing a possible visualization of the dimension change turning the problem of molecular dynamics from 3-dimensional into 4-dimensional. The red block, labeled t, represents the state of the system at timestep t, and the blue block, labeled $t+1$, represents the state of the system at timestep $t+1$.

in three smaller trapezoidal prisms. Figure 3-4 provides an example of what those three trapezoidal prisms, labeled τ_1, τ_2, and τ_3, might look like. I use the term ***left trapezoid*** to refer to the trapezoidal prism whose elements lie below the location of the space cut (τ_1 in the figure), and ***right trapezoid*** to refer to the trapezoidal prism whose elements lie above the location of the space cut (τ_2 in the figure), where the location of the space cut is defined by some pivot point, as discussed further in Chapter 4. The term ***middle trapezoid*** refers to the section of spacetime that cannot properly be filled in by either the left or right trapezoid (τ_3 in the figure), and must be calculated in serial after the completion of the left and right trapezoids. As in FSStencil, StencilMD takes advantage of the fact that there are no data dependencies between the left and right trapezoids, so they can be calculated in parallel.

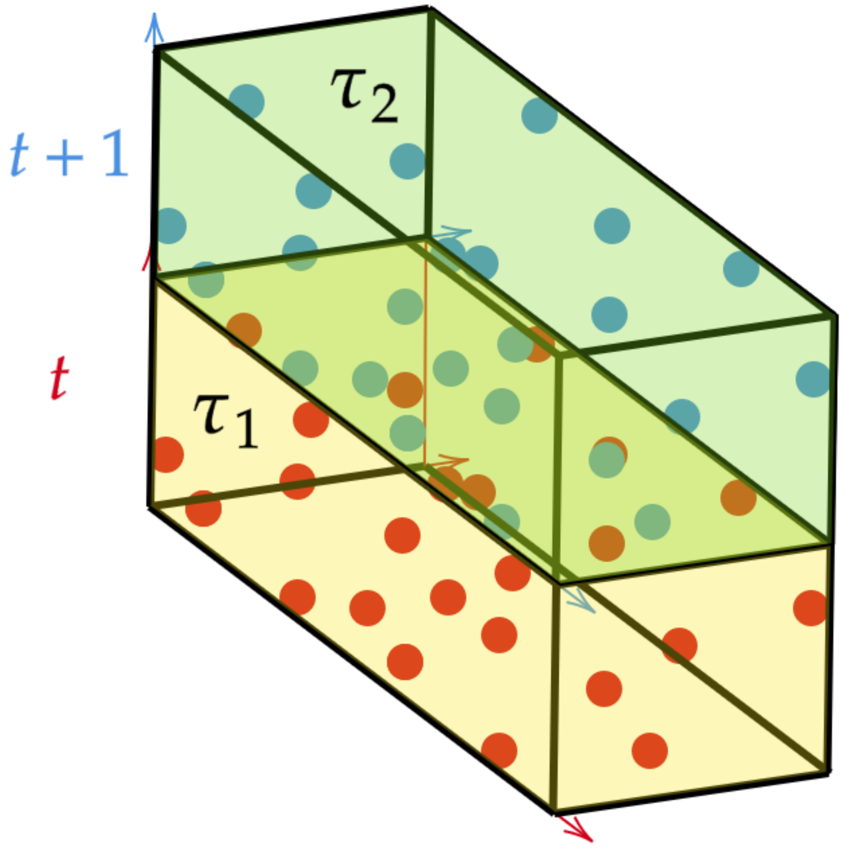


Figure 3-3: A diagram showing what a time cut looks like in the molecular dynamics regime. The trapezoid resulting from the time cut that must be processed first is labeled τ_1, and the trapezoid that must be processed second is labeled τ_2.

While the high-level idea of a space cut in StencilMD is similar to the idea of a space cut in FSStencil, the details of how to implement this space cut differ due to the fact that StencilMD performs a graph computation instead of a stencil computation. Specifically, while an atom graph has a natural embedding in 3-dimensional space and may be more regular in structure than a general graph, it does not have the same precise geometric properties of a stencil computation. Whereas FSStencil operates on a hypergrid, StencilMD operates on an atom graph, which is both dynamic and more general than a hypergrid. Atoms are distributed throughout space instead of positioned on a regular grid, so StencilMD cannot use the same concept

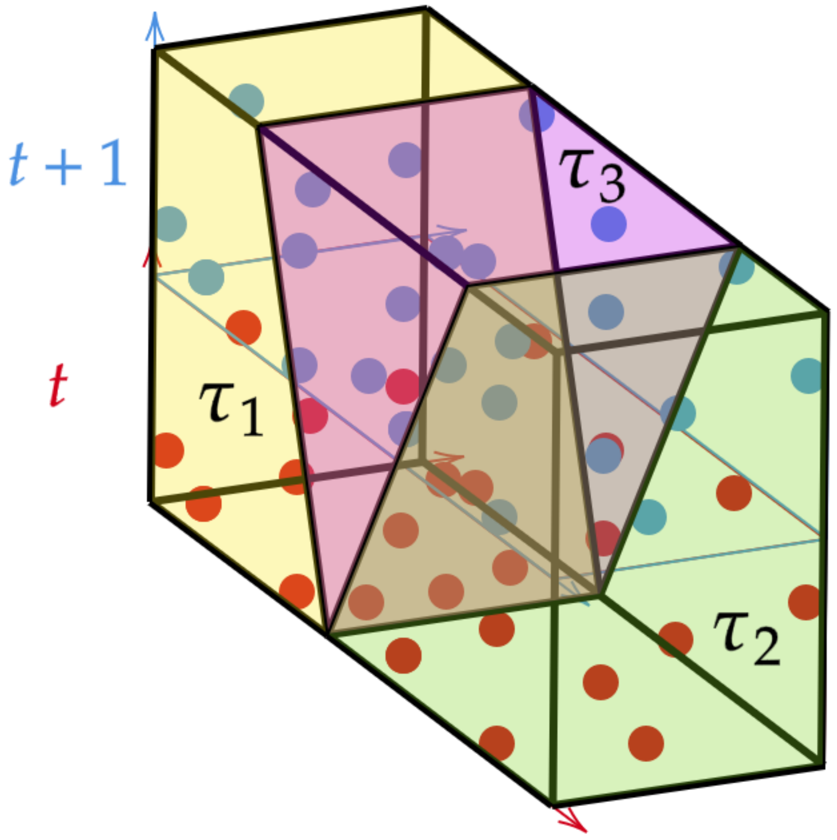


Figure 3-4: A diagram showing what a space cut looks like in the molecular dynamics regime. The resulting left and right trapezoids are labeled τ_1 and τ_2 respectively, whereas the middle trapezoid is labeled τ_3.

of "slope" to define a space cut as in FSStencil. Instead, StencilMD relies on the structure of data dependencies to define a space cut. I discuss this phenomenon in further depth in the following section.

Finally, StencilMD makes use of the same base-case condition, that the number of timesteps to evaluate equals 1, as in line 1 of FSStencil. When given a section of spacetime where only one timestep can be performed, StencilMD directly calculates the results of that timestep. Since a molecular dynamics simulation is not actually a stencil, however, this calculation differs significantly from the calculation found in the base case of FSStencil. The base-case calculation in StencilMD more closely mirrors a step in a classic molecular dynamics simulation as opposed to a stencil computation.

Trapezoid Representation

Before describing StencilMD in any further detail, I now discuss how I represent a trapezoidal prism in StencilMD. As shown in Figure 3-2, a trapezoidal prism in the context of StencilMD represents a collection of atom graphs ordered by their associated timestep. Now, since I have reformulated the problem of molecular dynamics simulation into the process of filling in a trapezoidal prism with the correct information, I will use the term ***trapezoid*** to refer to the ordered collection of atom graphs representing a given problem. There are three key types of information required to define a trapezoid in StencilMD: time information, atom information, and space cut information. At a high level, time information defines the height of the trapezoid, atom information determines the dimensions of the base of the trapezoid, and space cut information determines the slope of each side of the trapezoid.

I now discuss exactly what information is required to define a trapezoid using the three categories outlined above as reference. The time information required to define a trapezoid is twofold: the time at which the simulation represented by the trapezoid starts, and the time at which the simulation represented by the trapezoid ends. I refer to this starting time as T_0 and the ending time as T. Using ***height*** to refer to the range of the trapezoid in the time dimension, T_0 and T are enough to specify the height of the trapezoid. Now, the atom information required to define a trapezoid is merely the state of the set of atoms at T_0. For my purposes, the atomic number, often referred to as ***type***, and position of each atom suffices to represent the state, but the specifics of what information is contained by atom state can be changed without impact on StencilMD.

The space cut information required to define a trapezoid relates to the notion of the "slope" of each face of the trapezoid. With both FSStencil and StencilMD, data dependencies dictate what can be computed in a timestep given the information from the previous timestep. Because the data dependencies for ordinary stencils are defined to be neighboring points in a static k-dimensional hypergrid, there exists a simple algebraic formulation of the subgrid that can be computed in the next time step.

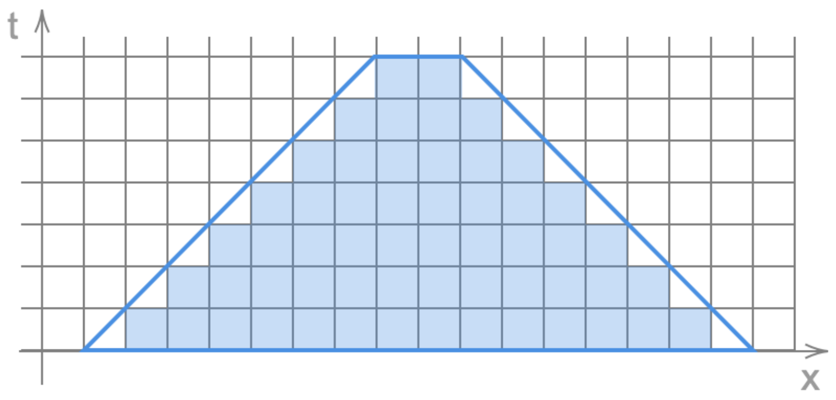


Figure 3-5: A diagram showing the shrinking effect of space cuts as a result of data dependencies in 2-dimensional spacetime with a grid superimposed over the atom graph. In this example, atoms exist in 1-dimensional space, with one atom per grid square, and each atom depends only on the atoms in the adjacent grid squares.

That algebraic formulation leads to the notion of slope. For StencilMD, however, there are two problems: the atom graph does not reside on a simple hypergrid, and the atom graph is not static.

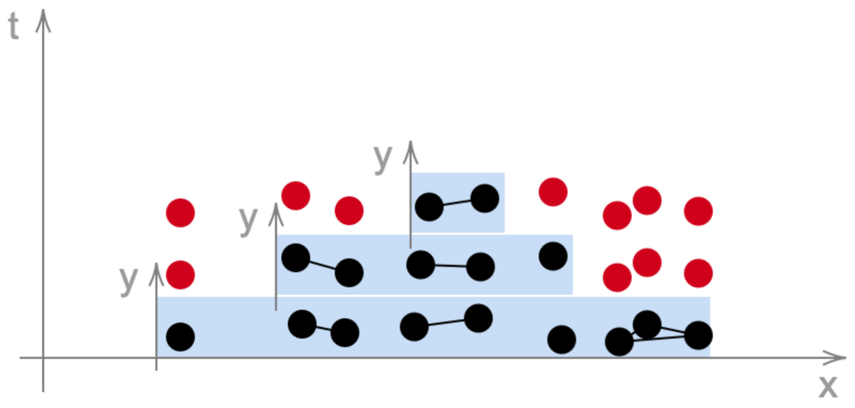


Figure 3-6: A diagram showing the irregular shrinking effect of space cuts in the x-dimension on atom graphs in 3-dimensional spacetime. Each blue bounding box represents a 2-dimensional space representing the set of atoms that remain valid at the corresponding timestep. Atoms that lie outside of the blue bounding boxes, marked in red, are invalid at that timestep.

Several possible solutions can help overcome the challenge that an atom graph does not reside on a 3-dimensional hypergrid in StencilMD. One idea involves imposing a 3-dimensional hypergrid over the space occupied by the atoms in the atom graph and associating each atom with a cell of the hypergrid. Figure 3-5 provides a basic example of what a trapezoid defined using a hypergrid would look like if the atoms existed in 1-dimensional space, with one atom per grid square, and each atom had dependencies only on the atoms in the adjacent squares of the grid. That hypergrid approach benefits from simplicity in that the hypergrid defines an easy-to-compute slope, but suffers when the atoms are not uniformly distributed in space. Another more general way to work around the irregular structure of an atom graph involves using the k-hop neighborhood of each atom to directly determine which subgraph can be computed in the subsequent timestep of the simulation based on data dependencies. In Figure 3-6, I stack atom graphs, encapsulated in bounding boxes, in 3-dimensional spacetime to demonstrate the k-hop neighborhood method of defining the trapezoid. In this example, $k = 1$, and any atom that shares an edge with the most extremal atoms in the bounding box at timestep t will not be present in the bounding box at timestep $t + 1$. The resulting trapezoid no longer has the regular structure as in Figure 3-5, but it allows for a more adaptable interpretation of slope when atoms are not distributed uniformly in space.

As for the issue that atom graphs are dynamic, the use of dynamic bounding boxes resolves this concern. By defining a new bounding box at each step of StencilMD that encapsulates all correct atoms at that step, StencilMD loses no information due to atomic movement.

Now, since the data dependencies of the atom graph fully define the "slope" of each face of a trapezoid, StencilMD does not require a numerical "slope" value to find the space cut information necessary to define a trapezoid. Instead, all StencilMD needs in terms of space cut information is the information about which sides border space cuts. If the side of the trapezoid borders a space cut, it has a "slope" defined by data dependencies, and if the side does not border a space cut, it has no slope. Since there are two sides to the trapezoid in each of three spatial dimensions, the space cut information must contain six values representing whether each of the six sides borders a space cut.

With the time information, atom information, and space cut information outlined above, a trapezoid inStencilMD is fully defined. I now discuss how StencilMD uses this idea of a trapezoid in its input and output.

Input and Output

Since StencilMD converts a molecular dynamics simulation into the process of filling in a trapezoid, the input to StencilMD must define the trapezoid representing the simulation. StencilMD operates on a structure $\tau = \{t_{start}, t_{end}, \mathit{positions}, \mathit{types}, \mathit{x\text{-}cut\text{-}upper}, \mathit{x\text{-}cut\text{-}lower}, \mathit{y\text{-}cut\text{-}upper}, \mathit{y\text{-}cut\text{-}lower}, \mathit{z\text{-}cut\text{-}upper}, \mathit{z\text{-}cut\text{-}lower}\}$, representing a trapezoid. I define the fields of τ as follows:

- t_{start} is an integer designating the starting time, measured in steps, of the simulation represented by the trapezoid;
- t_{end} is an integer designating the ending time, measured in steps, of the simulation represented by the trapezoid;
- *positions* is an array of floats designating the atom positions at time t_{start};
- *types* is an array of integers designating the atom types;
- *x-cut-upper* is a boolean designating whether the upper side of the trapezoid in the x-dimension borders a space cut;
- *x-cut-lower* is a boolean designating whether the lower side of the trapezoid in the x-dimension borders a space cut;
- *y-cut-upper* is a boolean designating whether the upper side of the trapezoid in the y-dimension borders a space cut;
- *y-cut-lower* is a boolean designating whether the lower side of the trapezoid in the y-dimension borders a space cut;
- *z-cut-upper* is a boolean designating whether the upper side of the trapezoid in the z-dimension borders a space cut;
- *z-cut-lower* is a boolean designating whether the lower side of the trapezoid in the z-dimension borders a space cut.

```
MD($positions, types, steps$)
1  $\tau = \{0, steps, positions, types, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}\}$
2  return StencilMD($\tau$)
```

Figure 3-7: Pseudocode for the top-level call to StencilMD.

These inputs define the trapezoid representing the simulation. As an example, Figure 3-7 shows what the top-level call to StencilMD looks like. Since each of the boolean inputs are false, the beginning set of atoms will equal the ending set of atoms with their updated states, as desired.

The output of StencilMD, however, does not represent a trapezoid, and therefore takes on a different form than the input. The output of a classic molecular dynamics simulation is the state of the system after the desired number of timesteps, so StencilMD returns a generalization of that output that takes into account the effects of space cuts. Instead of returning the final state of each atom, StencilMD returns, for each atom where the input contains all of that atom's data dependencies required to complete the first timestep, the last state of that atom that could be calculated correctly, as well as the states of certain atoms that are considered "close" to the boundary of a space cut. In Chapter 4, I will discuss what comprises the states of the output atoms and analyze the importance of each attribute in that state. In contrast to the states of the input atoms, the states of the output atoms contain more information than just position and type. Also in Chapter 4, I will discuss what I mean by "close" to the boundary of a space cut.

The difference between the output of a classic molecular dynamics simulation and the output of StencilMD is subtle, but important. When a trapezoid τ is not the result of a space cut, as with the top-level call to StencilMD shown in Figure 3-7, the output is the same; for each atom, the last state that could be correctly calculated is the final state of that atom, and no atoms are "close" to a space cut since no space cuts have occurred. When a trapezoid τ is the result of a space cut, however, the output must also include some intermediary states that would not normally be returned in a classic molecular dynamics simulation. These ***intermediary states*** refer to the states of atoms sufficiently "close" to the boundary of a space cut on a timestep earlier than $\tau.t_{end}$. The intermediary states are necessary to fill in the middle trapezoid resulting from a space cut. The explanation for why StencilMD requires these states to complete the middle trapezoid is the same as in FSStencil, described in Chapter 2.

Chapter 4

Stencil-Based Algorithm for Molecular Dynamics

This chapter presents StencilMD, the novel stencil-based algorithm for the parallelization of molecular dynamics simulators. StencilMD consists of three main components: the base case, the time cut, and the space cut. I describe each component in turn, and conclude with a discussion of the assumptions and design choices involved in StencilMD. Throughout this chapter, I refer to the pseudocode in Figure 4-1 to aid in the explanation of StencilMD.

4.1 Base Case

This section describes the computation performed in the base case of StencilMD. In this base case, StencilMD is tasked with performing one step of a molecular dynamics simulation on a given set of atoms and returning the corresponding states of those atoms after the timestep. To compute those states, StencilMD performs three key steps. The first step involves a direct call to Basecase along with some filtering of the resulting information, and the last two steps perform additional bookkeeping that becomes relevant during the space cut.

As shown in Figure 4-1, the first step of the base case, found on line 4, is a call to Basecase, a subroutine described in further detail in Chapter 2. As is standard

```
STENCILMD($\tau$)
$\Delta_t = \tau.t_{end} - \tau.t_{start}$
if $\Delta_t == 1$
    // Perform base case.
    $new\text{-}positions$ = BASECASE($\tau.positions, \tau.types, r$)
    $new\text{-}positions\text{-}valid = \{position \in new\text{-}positions \mid position$ is valid$\}$
    $new\text{-}types = \{type \mid \exists i, \tau.types[i] == type$ and $\tau.positions[i]$ is valid$\}$
    $time\text{-}of\text{-}last\text{-}update = \tau.t_{end}$ for each valid atom
    $borders\text{-}which\text{-}sides$ = which space cuts atom is close to for each valid atom
    return $new\text{-}positions\text{-}valid, new\text{-}types, time\text{-}of\text{-}last\text{-}update, borders\text{-}which\text{-}sides$
else
    $x\text{-}med$ = median $x$ value of $\tau.positions$
    $positions\text{-}below = \{position \in \tau.positions \mid position \leq x\text{-}med\}$
    $positions\text{-}above = \tau.positions \setminus positions\text{-}below$
    $positions\text{-}far\text{-}below = \{position \in positions\text{-}below \mid position < x\text{-}med - C \cdot \Delta_t\}$
    $positions\text{-}far\text{-}above = \{position \in positions\text{-}above \mid position > x\text{-}med + C \cdot \Delta_t\}$
    if $positions\text{-}far\text{-}below.length \geq \epsilon$ and $positions\text{-}far\text{-}above.length \geq \epsilon$
        // Perform space cut in $x$-dimension.
        return SPACE-CUT-X($\tau, x\text{-}med, positions\text{-}below, positions\text{-}above$)
    $y\text{-}med$ = median $y$ value of $\tau.positions$
    $positions\text{-}below = \{position \in \tau.positions \mid position \leq y\text{-}med\}$
    $positions\text{-}above = \tau.positions \setminus positions\text{-}below$
    $positions\text{-}far\text{-}below = \{position \in positions\text{-}below \mid position < y\text{-}med - C \cdot \Delta_t\}$
    $positions\text{-}far\text{-}above = \{position \in positions\text{-}above \mid position > y\text{-}med + C \cdot \Delta_t\}$
    if $positions\text{-}far\text{-}below.length \geq \epsilon$ and $positions\text{-}far\text{-}above.length \geq \epsilon$
        // Perform space cut in $y$-dimension.
        return SPACE-CUT-Y($\tau, y\text{-}med, positions\text{-}below, positions\text{-}above$)
    $z\text{-}med$ = median $z$ value of $\tau.positions$
    $positions\text{-}below = \{position \in \tau.positions \mid position \leq z\text{-}med\}$
    $positions\text{-}above = \tau.positions \setminus positions\text{-}below$
    $positions\text{-}far\text{-}below = \{position \in positions\text{-}below \mid position < z\text{-}med - C \cdot \Delta_t\}$
    $positions\text{-}far\text{-}above = \{position \in positions\text{-}above \mid position > z\text{-}med + C \cdot \Delta_t\}$
    if $positions\text{-}far\text{-}below.length \geq \epsilon$ and $positions\text{-}far\text{-}above.length \geq \epsilon$
        // Perform space cut in $z$-dimension.
        return SPACE-CUT-Z($\tau, z\text{-}med, positions\text{-}below, positions\text{-}above$)
    // No space cuts possible, perform time cut.
    return TIME-CUT($\tau$)
```

Figure 4-1: A description of STENCILMD in pseudocode.

in molecular dynamics simulations, I treat the cutoff radius r as a known constant, hence why it does not appear in the input to StencilMD. For clarity, I include r as a parameter in the call to Basecase to emphasize the subroutine in which the cutoff radius becomes relevant. At this point, StencilMD also filters out the invalid atoms, shown on lines 5 and 6. These are the atoms whose data dependencies are not satisfied by the trapezoid, meaning that the states calculated for these atoms in the Calculator subroutine are incorrect. I describe how to check the validity of an atom in Section 4.3.

The last two steps of the base case, found on lines 7 and 8 of Figure 4-1, do not relate to updating the positions of the atoms; rather, they serve the purpose of filling in the remaining attributes of the states of the output atoms. Each output atom has a state with four attributes: position, type, time of last update, and which space cuts the atom borders. The previously-discussed three steps allow StencilMD to fill in the position and type of each atom. The last two steps of the base case, then, correspond to filling in the last two attributes of output atom state. First, on line 7, each atom must be updated with the timestep at which its latest update occurred; since the base case performs a timestep from $\tau.\, t_{start}$ to $\tau.\, t_{end}$, the time of latest update for each atom must be $\tau.\, t_{end}$. Second, on line 8, StencilMD requires that each output atom have an attribute containing the information about which space cuts the atom is close to. This involves performing a check for closeness to each existing space cut, which I describe in Section 4.3, and marking the results of those checks in the state of the atom. In my implementation, each of the six possible space cuts (upper and lower x-dimension, upper and lower y-dimension, upper and lower z-dimension) corresponds to a unique prime number. Then, the *borders-which-sides* attribute takes on the value of least common multiple of the prime numbers corresponding to the space cuts an atom borders, or value 1 if the atom does not border any space cuts.

With the four output states calculated, the base case of StencilMD is now complete. I have now described exactly how StencilMD computes these states in the base case, but I postpone a discussion of why these specific states are necessary for Section 4.3.

4.2 Time Cut

This section provides the details of TIME-CUT, the subroutine of STENCILMD in which a time cut is performed. As discussed in Chapter 2, the time cut is an important way to subdivide spacetime in a stencil algorithm. STENCILMD makes virtually no change to either the mechanics of a time cut or the scenario in which a time cut is performed as compared to the time cut in FSSTENCIL, but for clarity, I now provide a description of the time cut used by STENCILMD. Figure 4-2 provides pseudocode for reference.

```
TIME-CUT($\tau$)
1   $halftime = \lceil(\tau.t_{end} - \tau.t_{start})/2\rceil$
2   // The lower trapezoid $\tau_1$ ends after $halftime$
3   $\tau_1 = (\tau.t_{start}, \tau.t_{start} + halftime, \tau.positions, \tau.types,$
            $\tau.x\text{-}cut\text{-}upper, \tau.x\text{-}cut\text{-}lower, \tau.y\text{-}cut\text{-}upper,$
            $\tau.y\text{-}cut\text{-}lower, \tau.z\text{-}cut\text{-}upper, \tau.z\text{-}cut\text{-}lower)$
4   $haltime\text{-}states = \text{STENCILMD}(\tau_1)$
5   $new\text{-}positions = \{position \mid \exists i, halftime\text{-}states.positions[i] == position$ and
                        $halftime\text{-}states.time\text{-}of\text{-}last\text{-}update[i] == halftime\}$
6   $new\text{-}types = \{type \mid \exists i, halftime\text{-}states.types[i] == type$ and
                        $halftime\text{-}states.time\text{-}of\text{-}last\text{-}update[i] == halftime\}$
7   // The upper trapezoid $\tau_2$ starts after $halftime$ and gets input from $\tau_1$
8   $\tau_2 = (\tau.t_{start} + halftime, \tau.t_{end}, new\text{-}positions, new\text{-}types,$
            $\tau.x\text{-}cut\text{-}upper, \tau.x\text{-}cut\text{-}lower, \tau.y\text{-}cut\text{-}upper,$
            $\tau.y\text{-}cut\text{-}lower, \tau.z\text{-}cut\text{-}upper, \tau.z\text{-}cut\text{-}lower)$
9   $end\text{-}states = \text{STENCILMD}(\tau_2)$
10  // Return a combination of states from $halftime\text{-}states$ and $end\text{-}states$
11  return $\{state \in halftime\text{-}states \mid$
            BORDERS-AT-LEAST-ONE($state.borders\text{-}which\text{-}sides$)$\}$
            $\cup\ end\text{-}states$
```

Figure 4-2: A description of the time cut algorithm in pseudocode.

A time cut is performed on a trapezoid τ by calling TIME-CUT, shown in Figure 4-2, which works as follows. Line 1 first computes the midpoint between the $\tau.t_{start}$ and $\tau.t_{end}$. Then, finding τ_1, the input to the first recursive call of STENCILMD, is trivial. As shown in pseudocode form on line 3 of Figure 4-2 and in diagram form

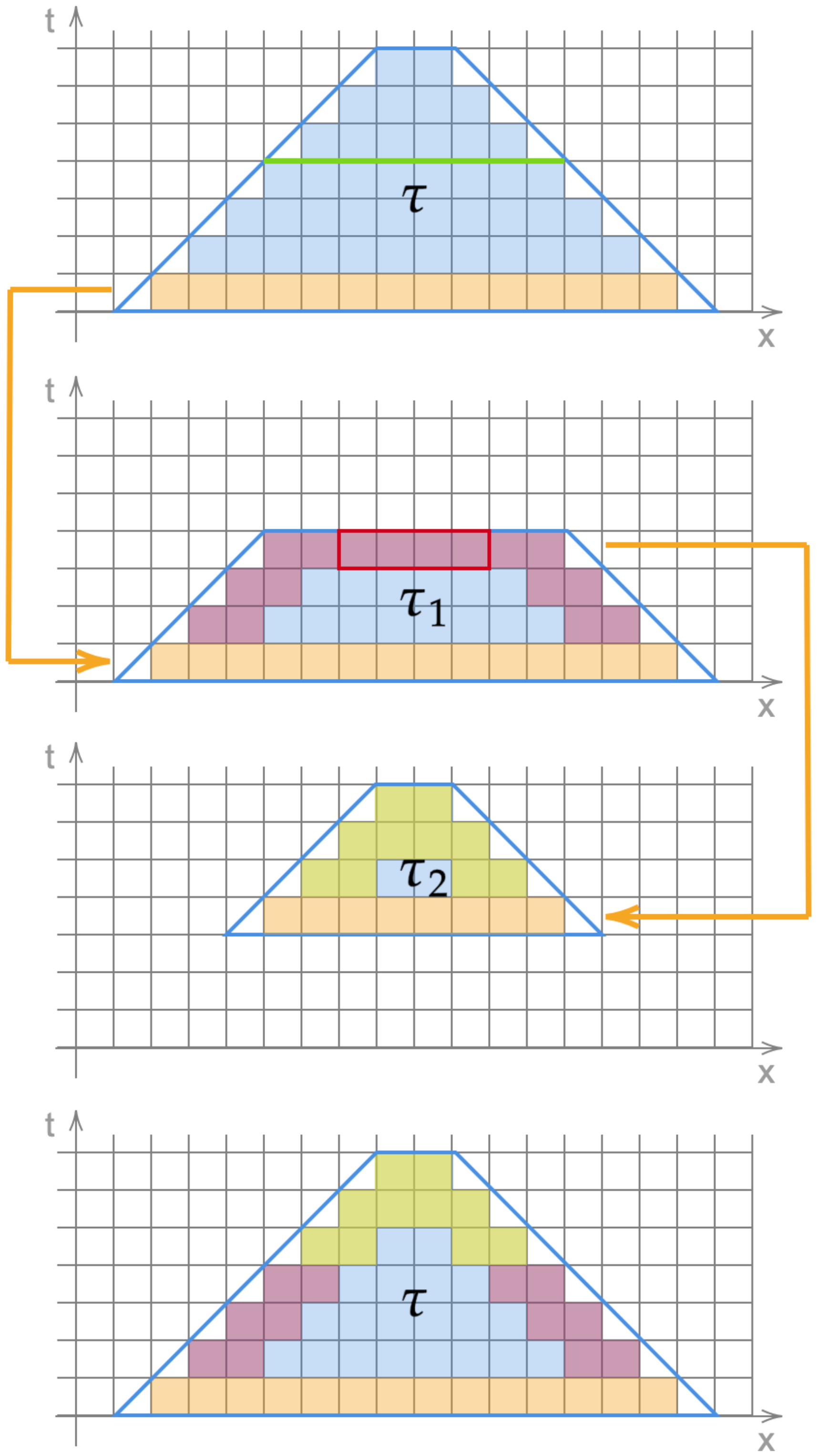


Figure 4-3: A diagram showing the flow of information during a time cut. Arrows demonstrate which sets of atoms become the inputs ot τ_1 and τ_2, and highlighted cells denote input and output.

in Figure 4-3, the set of atoms from the input of τ exactly equals the set of atoms composing the input of τ_1. In Figure 4-3, this relationship is denoted by the arrow pointing from the bottom row of τ to the bottom row of τ_1. Since TIME-CUT performs a time cut and not a space cut, the input variables representing which space cuts have occurred also do not change between τ and τ_1. The only input variable to change between τ and τ_1 is the end time of the simulation.

Finding τ_2, the input to the second recursive call of STENCILMD, requires more work. As shown on line 8, the start and end time can be calculated directly as the end time of τ_1 and the end time of τ respectively, and the variables representing which space cuts have occurred do not change, but finding the set of atoms to provide as input to τ_2 is not as obvious. Since the states of the atoms at the start time of τ_2 are unknown at the beginning of the call to TIME-CUT, TIME-CUT must wait for the first recursive call of STENCILMD to finish before determining the set of atoms that defines τ_2. Now, as shown on lines 5 and 6, TIME-CUT can use the *time-of-last-update* attribute of atom state to determine which atoms to use as input to τ_2; the atoms that were updated on the last step of the simulation represented by τ_1 are the new inputs to τ_2. Figure 4-3 presents this relationship with the arrow pointing from the top row of τ_1 to the bottom row of τ_2.

The last step of TIME-CUT, shown on line 11, involves combining the outputs of the first and second recursive calls to produce the desired result. TIME-CUT cannot merely return the union of both results, however, since the outputs of τ_1 include extra intermediary states that are not desired in the outputs of τ. Specifically, τ_1 outputs the states of all atoms at timestep $\tau.t_{start} + \mathit{halftime}$, but τ should output only the states of the atoms at that timestep that border one or more space cuts. These undesirable intermediary states are boxed in red in τ_1 in Figure 4-3. By combining the output of τ_2 with the output of τ_1 excluding the undesirable states, TIME-CUT produces the correct output for the result of the time cut, as shown in Figures 4-2 and 4-3.

The algorithm for excluding the undesirable states, BORDERS-AT-LEAST-ONE, is dependent upon the representation of the attribute *borders-which-sides*. In my implementation, the *borders-which-sides* attribute takes on value 1 for an atom if and only if that atom does not border any space cuts. Therefore, it suffices for BORDERS-AT-LEAST-ONE to check that the *borders-which-sides* attribute takes on a value not equal to 1 to determine which states to include or exclude.

4.3 Space Cut

This section presents the space cut algorithm used by STENCILMD to subdivide spacetime in any of the three space dimensions. The space cut algorithm is key for performance because it introduces opportunities both for parallelism and cache locality. To aid in the description of the space cut algorithm, I refer to both the pseudocode in Figure 4-5 and the diagram in Figure 4-4 throughout this section. For simplicity, I describe a space cut in the x-dimension, a process referred to as SPACE-CUT-X, since a space cut in any other space dimension can be described symmetrically.

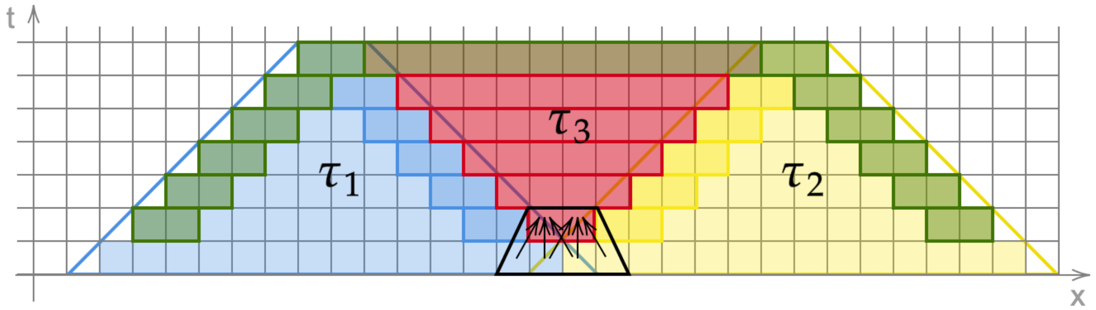


Figure 4-4: A diagram showing the process of finding the states to output as a result of a space cut. In this example, atoms exist in 1-dimensional space, and each cell of the grid may contain any number of atoms. A cell represents a section of Euclidean space with length $c \cdot r$, where c is the number of convolutional layers in the MDGNN used by STENCILMD, and r is the cutoff radius used to construct the atom graph. For a discussion of the importance of the number $c \cdot r$, refer to Section 4. The green, dark blue, and dark yellow cells represent outputs. The black trapezoid shows why the outputs of τ_1 and τ_2 are necessary to fill in τ_3 through the use of black arrows representing data dependencies.

Before calling SPACE-CUT-X, lines 11-16 of Figure 4-1 describe the manner for checking whether a space cut is possible. This first entails choosing a candidate lo-

```
SPACE-CUT-X($\tau$, $med$, $positions\text{-}below$, $positions\text{-}above$)
1   $types\text{-}below = \{type \mid \exists i, \tau.types[i] == type$ and $\tau.positions[i] \le med\}$
2   $types\text{-}above = \tau.types \setminus types\text{-}below$
3   $\tau_1 = (\tau.t_{start}, \tau.t_{end}, positions\text{-}below, types\text{-}below,$
                 TRUE, $\tau.x\text{-}cut\text{-}lower, \tau.y\text{-}cut\text{-}upper,$
                 $\tau.y\text{-}cut\text{-}lower, \tau.z\text{-}cut\text{-}upper, \tau.z\text{-}cut\text{-}lower)$
4   $\tau_2 = (\tau.t_{start}, \tau.t_{end}, positions\text{-}above, types\text{-}above,$
                 $\tau.x\text{-}cut\text{-}upper$, TRUE, $\tau.y\text{-}cut\text{-}upper,$
                 $\tau.y\text{-}cut\text{-}lower, \tau.z\text{-}cut\text{-}upper, \tau.z\text{-}cut\text{-}lower)$
5   $left\text{-}states$ = spawn STENCILMD($\tau_1$)
6   $right\text{-}states$ = spawn STENCILMD($\tau_2$)
7   sync
8   $\tau_3\text{-}latest\text{-}row\text{-}output = \{\}$
9   // Filling in $\tau_3$ using $\tau_1$ and $\tau_2$
10  for $t = \tau.t_{start}$ to $\tau.t_{end}$
11      $positions = \{position \mid \exists i, left\text{-}states.positions[i] == position$ and
                            BORDERS-XR($left\text{-}states.borders\text{-}which\text{-}sides[i]$) and
                            $left\text{-}states.time\text{-}of\text{-}last\text{-}update[i] == t\} \cup$
                     $\{position \mid \exists i, right\text{-}states.positions[i] == position$ and
                            BORDERS-XL($right\text{-}states.borders\text{-}which\text{-}sides[i]$) and
                            $right\text{-}states.time\text{-}of\text{-}last\text{-}update[i] == t\} \cup$
                     $\tau_3\text{-}latest\text{-}row\text{-}output.positions$
12      $types = \{type \mid \exists i, left\text{-}states.type[i] == type$ and
                     BORDERS-XR($left\text{-}states.borders\text{-}which\text{-}sides[i]$) and
                     $left\text{-}states.time\text{-}of\text{-}last\text{-}update[i] == t\} \cup$
                 $\{type \mid \exists i, right\text{-}states.types[i] == type$ and
                     BORDERS-XL($right\text{-}states.borders\text{-}which\text{-}sides[i]$) and
                     $right\text{-}states.time\text{-}of\text{-}last\text{-}update[i] == t\} \cup$
                 $\tau_3\text{-}latest\text{-}row\text{-}output.types$
13      $\tau_3\text{-}latest\text{-}row\text{-}input = (t, t+1, positions, types,$
                            TRUE, TRUE, $\tau.y\text{-}cut\text{-}upper,$
                            $\tau.y\text{-}cut\text{-}lower, \tau.z\text{-}cut\text{-}upper, \tau.z\text{-}cut\text{-}lower)$
14      $\tau_3\text{-}latest\text{-}row\text{-}output$ = STENCILMD($\tau_3\text{-}latest\text{-}row\text{-}input$)
15  return $\{state \in left\text{-}states \mid$ (BORDERS-AT-LEAST-ONE($state.borders\text{-}which\text{-}sides$) and
                            NOT BORDERS-XR($state.borders\text{-}which\text{-}sides$)) or
                            $state.time\text{-}of\text{-}last\text{-}update == \tau.t_{end}\} \cup$
        $\{state \in right\text{-}states \mid$ (BORDERS-AT-LEAST-ONE($state.borders\text{-}which\text{-}sides$) and
                            NOT BORDERS-XL($state.borders\text{-}which\text{-}sides$)) or
                            $state.time\text{-}of\text{-}last\text{-}update == \tau.t_{end}\} \cup$
        $\tau_3\text{-}latest\text{-}row\text{-}update$
```

Figure 4-5: A description of the space cut algorithm in pseudocode.

cation in space at which to perform the cut. Many different locations are acceptable for performing a space cut, but my implementation uses the median value in each dimension as the candidate location, as shown on line 11 for the x-dimension. I analyze this design choice, including its implications relating to the span of StencilMD, in Section 4.4. After choosing this location, the final step in determining whether a space cut can occur involves checking the validity of the trapezoids produced by the space cut. In my implementation, this entails estimating the number of atoms left at the top of the trapezoid and confirming that sufficiently many remain. I describe this approach and the related assumptions in Section 4.4.

With the validity check in lines 11-16 of Figure 4-1, Space-Cut-X can now assume that some preprocessing has been done to produce its input. Specifically, Space-Cut-X takes in the space cut location, *med*, as an argument, as shown in Figure 4-5, and assumes that the resulting space cut will be valid. Similarly, to avoid repeated calculations, Space-Cut-X also takes as arguments the positions of the atoms that fall below, *positions-below*, and above, *positions-above*, the location of the space cut. The final argument to Space-Cut-X is the trapezoid τ on which to perform the space cut. With these inputs, the last step of preprocessing before constructing the left, τ_1, and right, τ_2, trapezoids is a simple bookkeeping step to find the types corresponding with the atoms in each of the two positions arrays, shown in lines 1 and 2 of Figure 4-5.

After the necessary bookkeeping, lines 3-6 of Space-Cut-X construct and process the left and right trapezoids, labeled τ_1 and τ_2 in Figure 4-4. These trapezoids both have the same start and end time as τ, as well as the same values for whether any space cuts have occurred in the y- or z-dimensions. Because Space-Cut-X performs a space cut, however, the values for whether any space cuts have occurred in the x-dimension might change. Specifically, the left trapezoid, τ_1, is now the result of a space cut on its upper side in the x-dimension, so the variable representing whether that space cut has occurred must now be true for τ_1. Symmetrically, the right trapezoid, τ_2, is now the result of a space cut on its lower side in the x-dimension, so the variable representing whether that space cut has occurred must now be true for τ_2. Finally, the left trapezoid gets the positions and types corresponding to the atoms that fall below the space cut, and the right trapezoid gets the rest of the positions and types. Now that the left and right trapezoids have been constructed to have no data dependencies between each other, SPACE-CUT-X processes τ_1 and τ_2 in parallel before syncing on line 7.

The remainder of SPACE-CUT-X, starting on line 8, is concerned with filling in the middle trapezoid, labeled τ_3 in Figure 4-4. To produce the desired output for SPACE-CUT-X, shown in green on the top and outer edges of τ in Figure $4-4$, SPACE-CUT-X must calculate the top row of τ_3. Of course, the top row of τ_3 cannot be calculated without the rest of the rows of τ_3, so SPACE-CUT-X uses a loop, beginning on line 10, to iteratively calculate the next row of τ_3, starting from the bottom row. To be explicit, I define the calculation of the bottom row as the calculation that takes in the states of certain atoms at time $\tau.t_{start}$ and finds their state at time $\tau.t_{start} + 1$.

By providing all the data dependencies of a row of τ_3 as input to STENCILMD, the problem of computing that row of τ_3 essentially becomes the base case of STENCILMD where the input trapezoid is the result of space cuts on both its upper and lower edges in the x-dimension. In Figure 4-4, this relationship is diagrammed explicitly for the first row of τ_3 with arrows and a black trapezoid. The arrows represent the data dependencies of the first row of τ_3. The black trapezoid shows how the trapezoid that takes in all data dependencies as its input produces the first row of τ_3 as output. Constructing the black trapezoid to have space cuts on the upper and lower edges of the x-dimension is not strictly necessary, since the algorithm could instead perform the proper post-processing to enforce that *τ_3-latest-row-output* contains only the information pertaining to the desired row of τ_3 without any additional incorrect information. I choose to construct the black trapezoid with space cuts on the upper and lower edges of the x-dimension, however, as a reminder that not all atoms passed as input to this call to STENCILMD will still be valid in the output of that call. Now, while not shown explicitly in the diagram, this process of constructing base-case trapezoids can be repeated to produce a trapezoid for each row of τ_3 that satisfies the data dependencies of that row.

Producing the input, written as τ_3*-latest-row-input* on line 13 in Figure 4-5, for each call to StencilMD involves a combination of information from the left, right, and middle trapezoids. Specifically, in order to respect all the data dependencies of the row of τ_3 that occurs at time t, StencilMD requires information about the atoms bordering the upper space cut of τ_1 at time $t-1$, the atoms bordering the lower space cut of τ_2 at time $t-1$, and all the atoms of τ_3 at time $t-1$. In Figure 4-4, the atoms bordering the upper space cut of τ_1 are boxed and highlighted in dark blue, the atoms bordering the lower space cut of τ_2 are boxed and highlighted in dark yellow, and the atoms of each row of τ_3 are boxed and highlighted in red. Together, the dark blue, red, and dark yellow boxes of each row cover all data dependencies of the red box in the row above, as desired. On lines 11 and 12 of Figure 4-5, Borders-XR signifies a function that uses the state of an atom to determine whether it borders the upper space cut in the x-dimension. The function Borders-XL is defined symmetrically. In my implementation, where the six possible space cuts (upper and lower x-dimension, upper and lower y-dimension, upper and lower z-dimension) are each represented by a unique prime number, this function merely performs a check that the *borders-which-sides* attribute is divisible by the desired prime number. With this function, then, as well as a check that the *time-of-last-update* attribute contains the proper value, lines 11 and 12 fully define the method for calculating τ_3*-latest-row-input*.

The pseudocode in Figure 4-5 corresponding to the process of filling in τ_3, starting on line 8, shows the importance of each attribute of the state of an output atom, listed in Section 4.1. Specifically, Space-Cut-X requires the attribute *borders-which-sides* to determine which atoms from τ_1 and τ_2 are required to fill in τ_3. Similarly, Space-Cut-X makes use of the *time-of-last-update* attribute to ensure that the input to each call to StencilMD while filling in the row of τ_3 that occurs at timestep t only uses information from timestep $t-1$. The attributes *position* and *type* are generally necessary to provide the desired output of the molecular dynamics simulation, but Space-Cut-X also uses those attributes as input in the calls to StencilMD for filling in τ_3, further emphasizing their relevance.

At this point, it is important to discuss what I mean by an atom bordering a space cut. I use the terminology of being ***close*** to a space cut synonymously with the terminology of bordering a space cut. As hinted at in previous chapters, the definition of close to a space cut depends on the structure of the data dependencies of the atom graph. At timestep t, I consider an atom in trapezoid τ close to a space cut if τ does not satisfy that atom's data dependencies at timestep t or if τ will not satisfy that atom's data dependencies at timestep $t+1$. In other words, an atom in trapezoid τ is close to a space cut at timestep t if that atom will become invalid at timestep $t+1$ or $t+2$. I choose this definition specifically for its application in Space-Cut-X. With this definition of closeness to a space cut, it suffices for each trapezoid to return intermediary information only about atoms close to space cuts in order to obtain all the information necessary to fill in the middle trapezoid produced by a space cut. This relationship can be seen in Figure 4-4; the atoms in τ_1 and τ_2 whose data dependencies are not satisfied at timestep t become part of the row of τ_3 that occurs at timestep $t+1$. The remaining data dependencies of the row of τ_3 that occurs at timestep $t+1$ then must be the atoms at timestep t in τ_1 and τ_2 whose data dependencies are not satisfied at timestep $t+1$. Therefore, the row of τ_3 that occurs at timestep $t+1$ relies on the row of τ_3 that occurs at timestep t and the atoms of τ_1 and τ_2 at timestep t whose data dependencies are not satisfied at timestep t or $t+1$, which corresponds exactly with the definition of the atoms of τ_1 and τ_2 that are close to space cuts.

Now that I have defined closeness to a space cut, I describe how to check for this characteristic of an atom, which simultaneously provides a more mathematical definition of closeness to a space cut. Since closeness depends on the structure of data dependencies, the check for closeness will depend on the cutoff radius r and the number of convolutional layers c in the MDGNN used by StencilMD. For the NequIP MDGNN, used by my implementation, $c = 3$.

These values of c and r dictate which atoms are close to a space cut. Each atom is connected with an edge in the atom graph to all other atoms within distance r, so the state of each atom trivially depends on all other atoms within distance r. Because the MDGNN performs c graph convolutions, however, each atom also depends on all other atoms within its c-hop neighborhood. Fortunately, the cutoff radius dictates that all atoms in the c-hop neighborhood of a given atom must also be within a distance of $c \cdot r$ of that atom in Euclidean space. With that nice trait mapping the c-hop neighborhood of an atom into Euclidean space, the check to determine if an atom's data dependencies are not satisfied as a result of a space cut becomes a simple matter of checking whether the atom is within a distance of $c \cdot r$ of the most extremal atom in the direction of that space cut. For example, to check if an atom's data dependencies are not satisfied due to the upper space cut in the x-dimension, it suffices to check that the atom is within a distance of $c \cdot r$ of the atom with the largest x-coordinate.

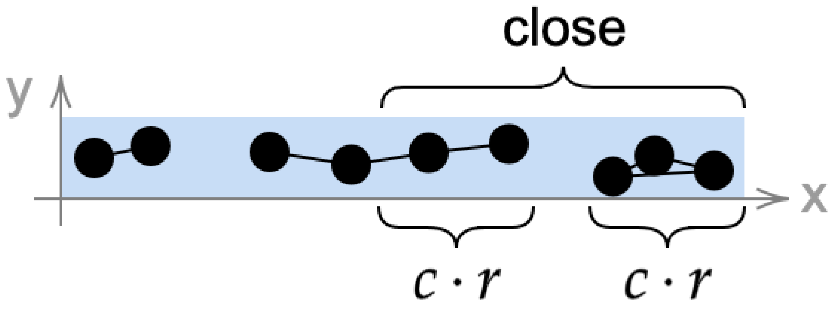


Figure 4-6: A diagram showing which atoms are considered close to the upper space cut in the x-dimension. Each black dot, representing an atom, is connected with an edge to any other atom within distance r.

To determine if an atom in trapezoid τ is close to a space cut at timestep t, then, StencilMD must simply perform this procedure twice, as shown in Figure 4-6. First, StencilMD determines the set of all atoms whose data dependencies are not satisfied by τ at timestep t by finding the atoms that are within $c \cdot r$ of the most extremal atom in the direction of the space cut of interest. Then, StencilMD determines the set of all atoms whose data dependencies are not satisfied by τ at timestep $t + 1$ by excluding the atoms whose data dependencies were not satisfied by τ at timestep t, and performing the check for atoms within $c \cdot r$ of the new most extremal atom. As displayed in the figure, the union of those two sets composes the set of atoms close to the space cut.

To check if an atom is invalid on lines 5 and 6 in Figure 4-1, a similar process works. In this case, however, the check for which atoms are within $c \cdot r$ of the most extremal atoms in the direction of space cuts only occurs once. In the base case of StencilMD, the atoms of τ within $c \cdot r$ of the most extremal atoms are incorrect, since their data dependencies are not satisfied by τ. After excluding those atoms, however, the check for closeness can be performed as usual. This loss of $c \cdot r$ atoms due to invalidity at each step is the phenomenon that makes $c \cdot r$ a natural choice for a smooth estimate of the "slope" of the side of a trapezoid. This "slope" is used to estimate the number of atoms remaining at the top of a trapezoid, as described in Section 4.4.

After filling in the middle trapezoid using the atoms of the left and right trapezoids that border space cuts, the final step of Space-Cut-X is to return information about the desired atoms. The desired output, shown in green on the top and outer edges of τ in Figure 4-4, includes the states of all atoms that are still valid at time τ_{end}, as well as the intermediary states on the lower side of the left trapezoid and the intermediary states on the upper side of the right trapezoid. Using the Borders-At-Least-One function described in Section 4.2 and the Borders-XL and Borders-XR functions from earlier in Space-Cut-X, along with a check of the *time-of-last-update* attribute, Space-Cut-X can easily compose the desired set of return atoms from the left, right, and middle trapezoids.

4.4 Assumptions and Design Choices

The StencilMD algorithm makes use of a few key assumptions that can be divided into two groups: assumptions for correctness and assumptions for applicability. My implementation of StencilMD also involves certain design choices. In this section, I discuss those assumptions and design choices.

StencilMD relies on two heuristics in order to guarantee correctness: first, that the atoms do not move "too much", and second, that error incurred by rearranging floating point calculations is acceptable.

The first condition for correctness results from the method StencilMD uses to check if a space cut is possible. Since StencilMD would have to run the full simulation in order to know with complete confidence how many atoms remain at the top of the trapezoid, negating any performance benefits that the stencil approach may have, StencilMD must instead produce some sort of estimate for that number of atoms. In my approach, this involves checking that sufficiently many atoms lie sufficiently far below the space cut. To determine whether an atom lies sufficiently far below the space cut, StencilMD estimates a smooth version of the slope of the side of the trapezoid. On lines 14 and 15 in Figure 4-1, this slope is represented as C. In general, C will be some constant c multiplied by the cutoff radius r, representing roughly how much space is lost at each timestep due to the invalidation of atoms. My implementation uses $C = 3r$, where the constant factor of 3 comes from the 3 convolutional layers of the NequIP architecture, as discussed in Section 4.3. To check whether sufficiently many atoms lie sufficiently far below the space cut, my implementation of StencilMD compares the number of atoms sufficiently far below the space cut with a constant. This constant can be tuned to match the application, and it reflects some measure of how much atom movement is considered "too much". Because this constant can be tuned, I consider the assumption that atoms do not move "too much" a reasonable one to make. This assumption merely enforces an upper bound on how many graph edges are allowed to be broken at each timestep, so any molecular dynamics application should be able to find a constant ensuring that this check works to determine the validity of trapezoids.

The second assumption for correctness relates to floating point error. StencilMD performs the floating point calculations in the physics engine in a different order than in LoopMD due to the subdivision of the atom graph. Floating point calculations are not associative, however, so the new order of calculations results in some small error as compared to LoopMD. To guarantee correctness, I make the assumption that this small error is acceptable. Since this error is unavoidable in almost any modification of LoopMD, I consider this a reasonable assumption to make.

Beyond the two assumptions for correctness, StencilMD also makes an assumption in terms of applicability relating to graph connectivity. While StencilMD may be correct, it remains unclear that it is always beneficial. Specifically, if the atom graph is constructed with a cutoff radius large enough that each atom is connected to nearly all other atoms, each space cut will result in huge losses of valid atoms at every timestep. In this situation, StencilMD may not be able to perform many space cuts due to this aggressive loss of valid vertices at each step. This would result in a lack of parallelism and cache locality, rendering the main benefits of StencilMD useless. I resolve this scenario through the assumption of a reasonable choice of cutoff radius. With a relatively small cutoff radius, as is typically chosen for molecular dynamics simulations, it is acceptable to assume that the atom graphs provided as input to StencilMD have the desired connectivity property for StencilMD to apply. This assumption is a key insight that allows StencilMD to work for the application of molecular dynamics. For other applications, however, StencilMD may not provide the same benefits as in molecular dynamics where the graph connectivity assumption is safe to make.

Finally, it is important to list the design choices involved in my implementation of the StencilMD algorithm and their implications. The first is the use of the median for producing candidate locations for space cuts. I chose this approach to ensure that each trapezoid resulting from the space cut begins with an even number of atoms, regardless of how the atoms are distributed in space. This evenness is a nice guarantee, but it comes at the price of having to perform the non-trivial median calculation, which adds a linear component to the span of StencilMD when computed serially. Increased span tightens the theoretical bounds on parallelism, so this design choice has important implications for the parallel scalability of StencilMD. For applications where atoms are relatively evenly-spaced, a simple average calculation might perform better than the median. This choice of space cut location is an important design choice for any implementation of StencilMD and certainly warrants further exploration.

Another key design choice in implementing StencilMD is the choice of graph representation. My implementation makes use of a static edge list that regenerates at each timestep. I made this choice due to ease of implementation, along with the assumption that graph creation would not dominate runtime. As molecular dynamics simulation runtime decreases, however, further optimizations in the realm of dynamic graph representation for application in molecular dynamics constitute a promising area of study. I discuss this idea further in Chapter 7.

The last important design choices of StencilMD are the choices of MDGNN and physics engine. As the state of the art in molecular dynamics simulations progresses, the MDGNN and physics engine of choice will surely change as well. While these MDGNNs continue to display the desired property of locality discussed in Chapter 2, however, StencilMD will continue to apply. The degree to which StencilMD succeeds in reducing runtime may change depending on the MDGNN, and an exploration of the results of StencilMD on different MDGNNs would be an interesting area for future work.

Chapter 5

Results

In this chapter, I present a summary of the performance gains, both through decreased runtime and decreased cache miss percentage, achieved by StencilMD as compared to LoopMD. I show that StencilMD can obtain up to a 28.57% speedup and a 26.92% decrease in cache miss percentage. This chapter discusses the details of the experiments I conducted to obtain these performance metrics, as well as the implications of these results in terms of the success of StencilMD.

Throughout the course of my experiments, I compared a prototype C++ implementation of StencilMD, parallelized using Cilk [7] and compiled using OpenCilk [22], to an implementation of LoopMD. The prototype implementation of StencilMD differs from the StencilMD algorithm described in Chapter 3 in that the prototype performs space cuts only in the x-dimension, as opposed to the full algorithm which performs space cuts in all three dimensions. I expect a full implementation to perform better in terms of runtime, parallel scalability, and cache locality, so the performance results presented in this chapter are especially promising given the potential for future improvements.

In the first experiment, I performed a comparison of the runtime of the prototype version of StencilMD, which makes calls to a pre-trained NequIP model in the Calculator subroutine, with the runtime of LoopMD. I ran both algorithms on two different inputs. Both inputs are systems containing Hafnium atoms and Oxygen molecules, but one of the inputs contains 768 atoms whereas the other contains 2600

	768 atom runtime	2600 atom runtime
LoopMD	148.66 s	1346.14 s
StencilMD	118.23 s	961.56 s
Speedup	**20.47%**	**28.57%**

Table 5.1: Summary of the results of my runtime experiments.

atoms. For each of the two inputs, I ran both algorithms for 5 timesteps on a 2.6 GHz 6-core Intel Core i7 CPU with 16 GB 2667 MHz DDR4 memory and 12MB shared L3 cache. I summarize the resulting performance numbers in Table 5.1. Notably, StencilMD reports a larger percent speedup for the larger input as compared to the smaller input, which is promising in terms of scalability. The relative speedup percentages are bolded in Table 5.1 to emphasize their importance.

For the next experiment, I conducted a comparison of the cache miss percentage of the prototype of StencilMD with LoopMD. Again, I ran both algorithms for 5 timesteps on two sets of inputs containing Hafnium atoms and Oxygen molecules, one with 768 atoms and the other with 2600 atoms. For each program execution, I measured the percentage of LLC cache misses out of all cache references. The resulting performance numbers are summarized in Table 5.2. As the size of the input increases, the problem may not fit into cache, so each algorithm incurs a higher percentage of cache misses, and the relative improvement of StencilMD goes down. Therefore, the decrease in relative improvement seen in Table 5.2 is not surprising. The relative improvement percentages are bolded in Table 5.2 to emphasize their importance.

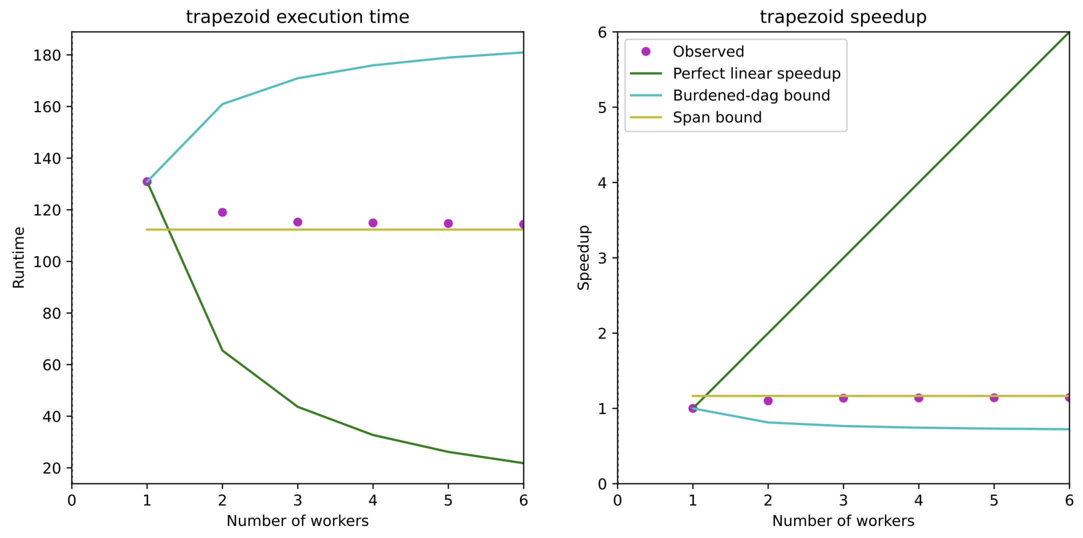
These results allow for many interesting conclusions to be drawn. The cache miss results in conjunction with the runtime results show that the improved cache locality of StencilMD as compared to LoopMD accounts for a non-trivial part of the speedup displayed in Table 5.1. The cache miss results also imply that as the problem size increases, a larger percentage of the runtime speedup is due to the subdivision of the graph and the resulting parallelism as opposed to the cache locality. Because a larger problem size permits for more space cuts, which creates more parallelism, these results are to be expected.

	768 atom LLC miss pctg.	2600 atom LLC miss pctg.
LoopMD	20.66%	53.42%
StencilMD	15.10%	45.52%
Percent decrease	**26.92%**	**14.78%**

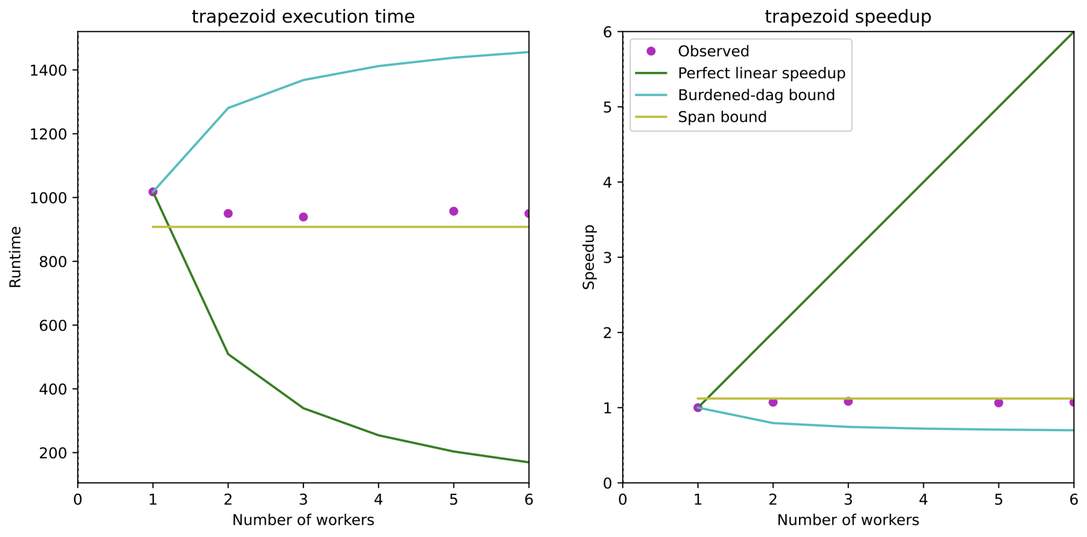
Table 5.2: Summary of the results of my cache miss experiments.

As a final experiment, I performed an analysis of the parallelism of the prototype of StencilMD. Using the Cilkscale visualizer [12], I obtained the charts in Figure 5-1 presenting the runtime and speedup of StencilMD in comparison with various bounds. Two important observations can be made from these charts; first, that StencilMD achieves almost ideal performance in comparison with the span bound, but second, that the span bound is quite tight. These observations imply that StencilMD has minimal overhead, but that it does not permit for much parallelism. Since the prototype only performs space cuts in the x-dimension, however, these results are still promising. I expect parallelism to increase as a result of space cuts in all three dimensions, possibly achieving theoretical performance bounds similar to those calculated for FSStencil [8].

Overall, the results of my experiments are highly promising for the success of StencilMD. While parallelism is low, the prototype of StencilMD leaves room for multiple opportunities to improve the bounds on parallelism. Even without space cuts in the y- and z-dimensions, the prototype of StencilMD shows a large percent speedup in overall runtime as compared to LoopMD, and a similarly exciting decrease in cache miss percentages. In Chapter 7, I discuss various paths forward to continue improving the numbers presented in this chapter.



(a) Analysis of parallelism on a system of 768 atoms.



(b) Analysis of parallelism on a system of 2600 atoms.

Figure 5-1: Charts presenting an analysis of the parallelism of StencilMD for systems of 768 and 2600 atoms. The yellow line represents the theoretical span bound of StencilMD, and the pink dots show actual results from running StencilMD with varying numbers of parallel workers.

Chapter 6

Related Work

This chapter summarizes related work on MDGNNs, parallel graph computations, and stencil-based algorithms.

MDGNNs

In 2017, Chmiela et al. [4] proposed gradient-domain machine learning as a method for predicting forces between atoms by explicitly requiring that the system obey the law of energy conservation. By training the gradient of the energy estimator, the resulting model accurately represents the relationship between the positions of atoms and the forces between them. This method, however, scales poorly due to a growth rate that is quadratic with respect to the number of atoms.

Schütt et al. [23, 24] proposed two more solutions for this same problem in that same year: deep tensor neural networks and SchNet. Deep tensor neural networks proved to be accurate on a small set of molecular dynamics simulations, and SchNet, a neural network with continuous-filter convolutional layers designed specifically to obey the physical constraints of atomic interactions, proved to achieve an even higher level of accuracy than the deep tensor neural networks. Similarly to Gilmer et al. [10], Schütt et al. were able to achieve these unprecedented levels of accuracy by using MDGNNs, which do not require manual feature engineering in order to conform to the laws of physics.

Those solutions were then outperformed by the work of Klicpera et al. [16], which considers directional information such as angular potentials when passing messages. Invariance to translation, rotation, and inversion are important physical properties of molecules, and encoding directional information in messages ensures the MDGNN's equivariance with respect to those translations. Using this idea, Klicpera et al. created DimeNet, a state-of-the-art molecular dynamics simulator with respect to accuracy on the common QM9 benchmark [21].

In 2021, Batzner et al. [3] improved upon DimeNet by creating NequIP, which uses vectors instead of scalars to represent the features and relative positions of atoms. With this extension, Batzner et al. were able to achieve higher accuracy over longer periods of simulation time with NequIP than Klicpera et al. saw with DimeNet. The current state of the art in molecular dynamics, then, involves MDGNNs with the structure required by StencilMD, as described in Chapter 2.

While the current implementation of StencilMD uses the NequIP model as the MDGNN of choice, other, more recent MDGNNs such as Allegro [19] should work without issue. By combining the benefits of StencilMD with the parallel scalability of an MDGNN such as Allegro, I expect to see further improvements in the performance of molecular dynamics simulations in the future.

Parallel Graph Computations

I now discuss related work in the field of parallel graph computations. Besides StencilMD, many other techniques for parallel graph computation exist and are currently in use for various applications. Some key examples of these other approaches to parallel graph computation include fine-grained locking, chromatic scheduling, priority-DAG scheduling, and the use of ghost atoms. These approaches, however, each have their downsides.

For parallel graph computations that utilize fine-grained locking, the approach associates a lock with each vertex and grabs the locks on all neighboring vertices before performing an update. The main concern with this approach, however, involves the significant overhead incurred by fine-grained locking [17]. The cost of this overhead means that such an approach should be avoided if possible for an application like molecular dynamics. Similarly, fine-grained locking techniques can lead to nondeterministic results, another undesirable feature for applications in molecular dynamics.

Chromatic scheduling is another common technique used for the parallelization of dynamic graph computations [14]. This approach schedules a parallel graph computation based on a coloring of the conflict graph. In the conflict graph, two vertices are connected with an edge if updating those vertices in parallel would produce a race. By coloring the conflict graph, then, vertices of the same color can be processed in parallel without races. This technique solves the problems of overhead and nondeterminism present in fine-grained locking techniques, but chromatic scheduling is unable to leverage any specific attributes of the graph computation it performs. Specifically, the graph in a molecular dynamics simulation is an atom graph, which has nice properties relating to connectivity and an inherent relationship to the physical world. In StencilMD, I make use of those specific properties to perform optimizations that would not be allowed in the chromatic scheduling of a generic graph computation.

A related technique for parallel graph computation scheduling is called priority-DAG scheduling [28]. In this approach, a directed acyclic graph (DAG) is constructed to represent the data dependencies of each step of the computation. An algorithm then applies various heuristics to determine the best way to schedule the graph computation represented by the DAG. Similarly to chromatic scheduling, however, priority-DAG scheduling applies to generic graph computations, and cannot harness any of the nice properties that atom graphs display.

For scientific computing in particular, a common approach for parallel graph computation involves using a spatial partitioning of the graph with additional "ghost" atoms for each partition [26]. However, not only does this approach require the additional overhead of processing those ghost atoms, but this spatial partitioning technique also involves a great deal of computationally intensive communication between partitions. Because this approach generally runs using the MPI parallel communication standard, communication between partitions might require inter-machine message passing, which can drastically increase runtimes. StencilMD, on the other hand, requires neither ghost atoms nor inter-machine communication, solving both main concerns with the ghost atom approach.

Stencil Algorithms

I conclude this chapter with a brief discussion of stencil algorithms. My work is based on FSStencil, the trapezoidal stencil algorithm developed by Frigo and Strumpen [8], but researchers have studied plenty of other variations on the trapezoidal stencil algorithm [9, 25]. In cases where an approximation of the result of a stencil computation is acceptable, many algorithms make use of Krylov methods instead of trapezoidal decompositions [2, 11, 13]. Finally, recent developments in stencil computations involve fast Fourier transforms to achieve theoretically good bounds on runtime [1]. In the following chapter, I discuss paths forward for future work relating to the parallelization of molecular dynamics simulations that involve further exploration of the field of stencil algorithms.

Chapter 7

Conclusion

In this thesis, I have presented StencilMD, a novel approach for the parallelization of directional message-passing algorithms for molecular dynamics in which I leverage the nice connectivity features of atom graphs to apply a stencil-based approach for graph decomposition. I have analyzed the two main areas in which molecular dynamics simulations benefit from this approach — decreased runtime and decreased cache miss percentage — and shown exciting preliminary results in both areas.

As foreshadowed in previous chapters, several avenues remain to be explored in future work related to this thesis. The path forward that I deem most important involves exploring more opportunities for parallelism by decreasing the span of the algorithm. First and foremost, it will be valuable to perform experiments on a full parallel implementation of StencilMD with space cuts in all dimensions to measure the increase in parallelism resulting from those additional space cuts. Similarly, this topic will benefit from an exploration of what methods can be used to parallelize the rest of the algorithm. For example, an interesting path forward involves researching the use of parallel partition and parallel prefix sum methods to replace the current serial method for partitioning the set of atoms about the median for a space cut, as shown on line 11 of Figure 4-1. This path forward might also involve an investigation into the use of reducers to parallelize certain aspects of the bookkeeping required by StencilMD, or even the parallelization of the MDGNN itself.

Beyond finding more avenues for parallelization, another valuable area for exploration involves a foray into the field of dynamic graph representation. As the parallelism of StencilMD continues to improve, the graph representation will start to become a more important component of overall runtime. I would be interested to see a study of the effects of various dynamic graph representations on the performance of StencilMD. Especially for larger inputs, I expect that the impact of not rebuilding the graph at every step of the simulation will provide a non-trivial performance boost.

In terms of cache efficiency, future work should involve experiments relating to whether StencilMD translates to other parallel systems, such as distributed computing clusters or systems with multiples GPUs. In a distributed environment, the goal is not just to minimize cache misses, but also to minimize communication between machines. I believe the cache efficiency numbers presented in Chapter 5 signify a promising possibility for similar results in distributed environments.

The final path forward that I discuss in this thesis is an exploration of other stencil methods besides the trapezoidal approach used in StencilMD. For example, an analysis of the tradeoff between the performance and accuracy achieved by Krylov methods instead of a trapezoidal approach might be enlightening. Similarly, a comparison of performance results between variations of the trapezoidal approach might illuminate additional paths forward in terms of improving parallelism. The recently developed methods that use fast Fourier transforms in stencil computations might also provide interesting results when applied to the problem of molecular dynamics through StencilMD.

Overall, I am excited both by the promising results of StencilMD so far and the numerous avenues for future research in this area. My work applies the idea of stencil algorithms to the seemingly unrelated field of MDGNNs and shows that this novel approach not only works, but works well. This work exists at an exciting intersection of many different fields, and I anticipate compelling results from a diverse body of work related to this topic in the future.

Bibliography

[1] Zafar Ahmad, Rezaul Chowdhury, Rathish Das, Pramod Ganapathi, Aaron Gregory, and Yimin Zhu. Fast stencil computations using fast fourier transforms, 2021.

[2] Oliviero Andreussi, Ismaila Dabo, and Nicola Marzari. Revised self-consistent continuum solvation in electronic-structure calculations. *The Journal of chemical physics*, 136:064102, 02 2012.

[3] Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P. Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E. Smidt, and Boris Kozinsky. SE(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials, 2021.

[4] Stefan Chmiela, Alexandre Tkatchenko, Huziel E. Sauceda, Igor Poltavsky, Kristof T. Schütt, and Klaus-Robert Müller. Machine learning of accurate energy-conserving molecular force fields. *Science Advances*, 3(5):e1603015, 2017.

[5] Chinmayee Choudhury, U. Deva Priyakumar, and G. Narahari Sastry. Dynamics based pharmacophore models for screening potential inhibitors of mycobacterial cyclopropane synthase. *Journal of Chemical Information and Modeling*, 55(4):848–860, 2015. PMID: 25751016.

[6] D. J. Evans and B. L. Holian. The Nosé-Hoover thermostat. *The Journal of Chemical Physics*, 83(8):4069–4074, 1985.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, may 1998.

[8] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, page 361–366, New York, NY, USA, 2005. Association for Computing Machinery.

[9] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache-oblivious algorithms. *Theor. Comp. Sys.*, 45(2):203–233, jun 2009.

[10] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry, 2017.

[11] Yifei Guan and Igor Novosselov. Two relaxation time lattice boltzmann method coupled to fast fourier transform poisson solver: Application to electroconvective flow. *Journal of Computational Physics*, 397:108830, 2019.

[12] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The cilkview scalability analyzer. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, page 145–156, New York, NY, USA, 2010. Association for Computing Machinery.

[13] Matthias Kabel, Thomas Böhlke, and Matti Schneider. Efficient fixed point and newton-krylov solvers for fft-based homogenization of elasticity at large deformations. *Computational Mechanics*, 54:1497–1514, 12 2014.

[14] Tim Kaler, William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. *ACM Trans. Parallel Comput.*, 3(1), jul 2016.

[15] John L Klepeis, Kresten Lindorff-Larsen, Ron O Dror, and David E Shaw. Long-timescale molecular dynamics simulations of protein structure and function. *Current Opinion in Structural Biology*, 19(2):120–127, 2009.

[16] Johannes Klicpera, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs, 2020.

[17] Mark Lewis and Cameron Swords. Lock-graph: A tree-based locking method for parallel collision handling with diverse particle populations. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2014.

[18] Zugang Mao, Ajay Garg, and Susan B Sinnott. Molecular dynamics simulations of the filling and decorating of carbon nanotubules. *Nanotechnology*, 10(3):273, 1999.

[19] Albert Musaelian, Simon Batzner, Anders Johansson, Lixin Sun, Cameron J. Owen, Mordechai Kornbluth, and Boris Kozinsky. Learning local equivariant representations for large-scale atomistic dynamics, 2022.

[20] Marta Pinto, Juan Perez, and Jaime Rubio-Martinez. Molecular dynamics study of peptide segments of the BH3 domain of the proapoptotic proteins Bak, Bax, Bid and Hrk bound to the Bcl-xL and Bcl-2 proteins. *Journal of computer-aided molecular design*, 18:13–22, 02 2004.

[21] Raghunathan Ramakrishnan, Pavlo O. Dral, Matthias Rupp, and O. Anatole von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1, 8 2014.

[22] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. *SIGPLAN Not.*, 52(8):249–265, jan 2017.

[23] Kristof T. Schütt, Farhad Arbabzadah, Stefan Chmiela, Klaus R. Müller, and Alexandre Tkatchenko. Quantum-chemical insights from deep tensor neural networks. *Nature Communications*, 8(1), Jan 2017.

[24] Kristof T. Schütt, Pieter-Jan Kindermans, Huziel E. Sauceda, Stefan Chmiela, Alexandre Tkatchenko, and Klaus-Robert Müller. Schnet: A continuous-filter convolutional neural network for modeling quantum interactions, 2017.

[25] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 117–128, New York, NY, USA, 2011. Association for Computing Machinery.

[26] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, 2022.

[27] Christopher J Woods, Frederick R Manby, and Adrian J Mulholland. An efficient method for the calculation of quantum mechanics/molecular mechanics free energies. *The Journal of chemical physics*, 128(1):01B605, 2008.

[28] Wei Zheng, Lu Tang, and Rizos Sakellariou. A priority-based scheduling heuristic to maximize parallelism of ready tasks for dag applications. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 596–605, 2015.