2019

OPTIMIZATION OF SIMULATIONS IN OPENSIMPPLLE

Robin Lockwood
University of Montana, Missoula

OPTIMIZATION OF SIMULATIONS IN OPENSIMPPLLE

By

Robin Stanley Lockwood

Bachelor of Science, University of Montana, Missoula, MT, 2017

Thesis

Master of Science
in Computer Science

University of Montana
Missoula, MT
Spring 2019

Abstract

Computer software has become an integral tool in exploring scientific concepts and computational models. Models, such as OpenSIMPPLLE, use a complex set of rules developed by experts to predict the impact of fires, disease, and wildlife on large scale landscapes.

OpenSIMPPLLE's simulations are time-consuming when projecting far into the future. OpenSIMPPLLE needs to execute more efficiently to allow for faster completion of simulations. The increase in speed will also enable users to run simulations with more timesteps in shorter periods. There are plenty of ways to accomplish this.

The work described here identifies three different methods for increasing efficiency. The first method is refactoring expensive operations, the second is applying design patterns, and the third is to introduce parallelism. The main objective of this work is to examine whether the intersection of parallelism and efficient design will combine in an optimal runtime while analyzing the best approaches to implement parallel techniques.

Contents

List of Figures

List of Tables

Chapter 1

SIMPPLLE Model

1.1 Overview

OpenSIMPPLLE is a landscape modeling software package written in the Java programming language. It is an open-source project that is currently spearheaded out of the University of Montana and funded by the United States Forest Service (USFS). Jimmie Chew developed OpenSIMPPLLE in 2012 (Chew, Moeller, & Stalling, 2012). OpenSIMPPLLE is used to model landscape-level changes in vegetation with regards to change agents such as fire, insects, drought, bison grazing, among others. The utility of this software is to inform the scientists and forest managers about how different processes affect the landscape and to inform forest management decisions.

A landscape is a geographical area with many Existing Vegetative Units (EVUs). The software provides reports about how the geographic area was affected by processes, the effectiveness of treatments, and overall costs to treat units and fire damage. An EVU is the spatial unit of the landscape. An EVU can is like a polygon that describes the attributes of the encompassing area.

Separate from the area, are zone rules. These rules define the rate of process ignition (not exclusive to fire), succession/transition pathways, process spread rates, invasive species logic, and many others. The landscape relies on the zone to include the proper rules for each of the area's Existing Vegetative Units (EVUs). Therefore, zone definitions are an integral part of the package.

Pathways are a path through stages of vegetation development; it is much like a Tree data structure. If an EVU does not change because of a process, it transitions species by a succession pathway. Succession is the process of turning an EVUs spatial area over time. Succession describes the growth of a species over time unless a disturbance interrupts the growth. If an EVU gets affected by a process, the pathway defined for that process changes the EVU to either a new state or new species. For example, a severe fire process will likely return the EVU to the start of the Succession pathway, typically a seral species. A seral community is an intermediate succession community of grasses, shrubs, and saplings. Every timestep, each EVU undergoes a state change based on the Succession pathway. The state of the EVU contains information about species, density, active processes, previous processes, etc. If a pathway is not found for a lifeform in an EVU in the map, an error occurs.

Once a zone and area have been input by a user, a simulation can be run many times over many timesteps. One run computes the number of timesteps, then resets the simulation for the next run. Each timestep describes a decade of change. The growth over the decade timestep is meant to be realistic over ten years. This helps reduce computation by estimating a decade timespan instead of shorter timespans.

1.2 Timestep Functions

Each timestep goes through several algorithms that result in the overall change within the area. After each step, results are either saved to memory or to an output file. The algorithms in each stage are, in order, EVU initialization, apply treatments, ignite processes, spread processes, compute the next state by pathway, re-seed EVUs, save results.

The first step is initialization. Initialization is several protocols that either reset or initialize data required for every timestep. The protocols involved are examining road and trail status and resetting local data caches.

The treatment protocol follows the initialization step. Treatments are management actions that affect vegetation. These treatments may include actions such as deadfall cleanup, pruning, harvesting timber, or moving the condition of the EVU to the desired state.

After the treatment function, the software computes the ignition points.

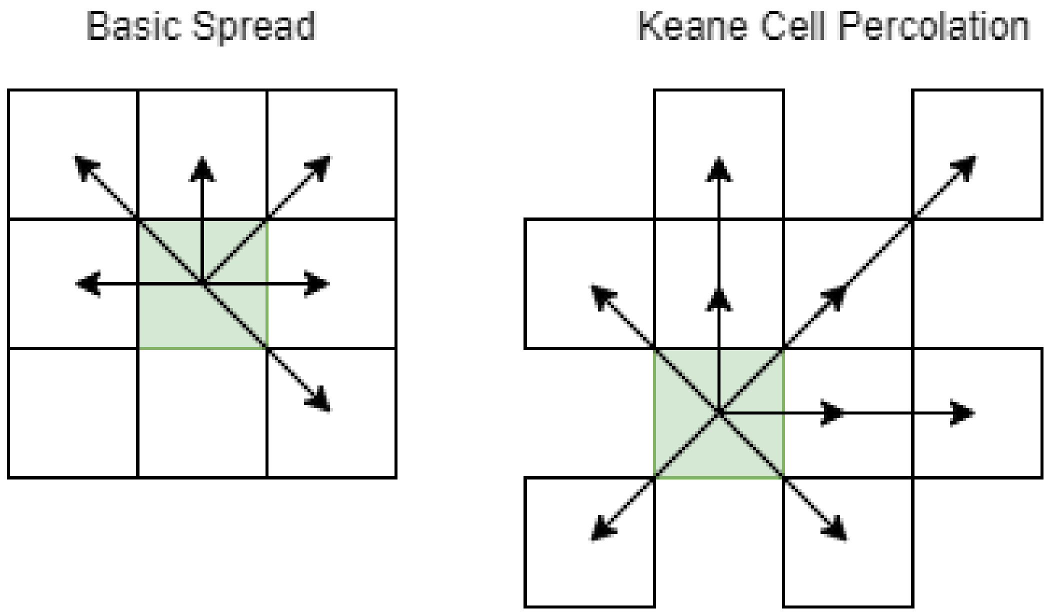
It accomplishes this by looping over all the EVUs and computing a process likelihood for each. If the likelihood meets the threshold for ignition in an EVU, the EVU records the process as affecting that EVU.

After determining ignition points, any process that is a spreading type executes its spread algorithm. OpenSIMPPLLE has two methods it can employ for predicting the spread of fires. The first method is the SIMPPLLE method. The SIMPPLLE method spreads fires from one EVU to its immediate neighbors. This method creates block-like fire events (Figure 1.1). The second method is the Keane Cell Percolation algorithm. The Keane Cell Percolation algorithm spreads fires to its neighbors and neighbor's neighbors by a differential function that takes slope, wind speed, and wind direction into account (Keane, Holsinger, & Pratt, 2006). This method creates more elliptical fires that are more realistic (Figure 1.1).

Depending on the conditions of the neighboring EVUs, processes spread to their immediate neighbors. An example of this is the invasive Mountain Pine Beetle. It will spread from one EVU to another EVU if the latter is an immediate neighbor and contains the proper species of Pine Trees. Other processes spread using the logic defined in their respective interfaces, where an interface describes a specific process implementation. For example, Mountain Pine Beetle is implemented slightly differently than Western Spruce Budworm

Once all spread events have concluded, each EVU undergoes state change depending on the processes involved in the previous steps via the Pathway

Figure 1.1: SIMPPLLE & Keane Fire Spread Methods



The SIMPPLLE method spreads events from the central node to its immediate neighbors. The Keane method spreads events from the central node to nodes along an elliptical based on wind speed and uphill slopes.

Tree. Succession is a process as well and typically means that another process did not affect the EVU.

To recap, the program models an area with zone rules. The area is a set of spatially explicit EVUs that contains information about its lifeforms and other properties like density, lifeform, acreage, etc. EVUs go through a treatment function and ignition function. Ignited events spread to neighboring EVUs, and the EVUs next state is computed through the Pathway Tree. The state of the simulation is saved to memory or disk once the resolution of the Next State is complete.

1.3 Parallelism

Parallelism was not implemented in OpenSIMPPLLE previously. OpenSIMPPLLEs features are indicative of parallelizable algorithms where many executing tasks use similar protocols. Because of the initial design, the program contains many global variables that work in a serial or sequential program. These globals will likely cause race conditions where local data is overwriting global data. Race Conditions are a class of computer bug where parallel events have the potential to read and write data at the same time. Race Conditions cause inconsistency when those two operations coincide. With that, global read-only data does not need to be refactored, while any variables with write-access do need refactoring.

1.4 Design Considerations

When optimizing software, it is vital to consider the design of the interfaces and possible refactoring solutions. An efficient design separates interfaces into a hierarchy that reflects the best decomposition of classes and functions. A refactored solution reduces computation and efficiently reuses code.

Chapter 2

Literature Review

2.1 Landscape Analysis

OpenSIMPPLLE is an analysis package for modeling landscape vegetation and disturbances. It is much like the LANDFIRE project proposal by Matthew G. Rollins, Robert E. Keane, and Zhiliang Zhu (2006). The concepts contained in the LANDFIRE conform to the Landscape Dynamic Simulation Model (LDSM).

2.1.1 Vegetation Maps

OpenSIMPPLLEs inputs include vegetation maps, wildland fuel maps, and fire regime maps. Attributes of the vegetation map are existing vegetation, potential vegetation, canopy height, and canopy cover. These are described in the vegetation map (Rollins, Keane, Zhu, & Menakis, 2006). A spatial unit, known as an Existing Vegetative Unit (EVU), is the most basic unit of the vegetation map and stores the species, size class, and density as attributes of the EVUs state.

Succession pathway models were created using the multiple pathway approach of Kessell and Fischer (1980) in which succession classes are linked along pathways defined by stand development and disturbance probabilities. The succession pathway defines how one species is affected by fires and other processes and how the EVU should transition by species or stand density. By providing general rules to follow when transitioning species by a process, decisions can be made using an expert knowledge system (Rollins et al., 2006).

2.1.2 Wildfire Behavior

Fuel maps describe fire behavior based on available fuel (Rollins et al., 2006) Opensimpplle models fuel as an attribute of the Existing Vegetative Unit (EVU). Fuel load is inferred based on the structure and density of the EVUs. Density is a function of succession over time. A tree stand will become denser as time continues forward without a disturbance affecting its pathway trajectory.

Fire regime maps are the simulated historical fire return interval and severity, fire regime condition class, and indices of departure from historical conditions (Rollins et al., 2006). OpenSIMPPLLE uses a historical set of initial conditions, and the rate of fire incidence is parameterized by spread rate, maximum acreage, and spread type.

2.2 Multithreading

As the field of computing and algorithmic development continues to grow, programs range in complexity and size. Algorithms of high complexity and large size can tax the resources of the computer system. High-performance computers can exploit caching, prefetching, and multiprocessor technologies (Ling, Mullen, & Lin, 2000) to handle high demand protocols. In a multithreaded system, the processor switches between threads so that each request can execute simultaneously and independently (Ling et al., 2000). Multithreading is a standard technique used to handle the requests commonly encountered in computing (Berg, 1996; Richter, 1996).

Multithreading speed ups do not come without a cost (Ling et al., 2000). Every request for a thread incurs a cost to create, initialize, and destroy the thread. In a concurrent threading model, any request has to allocate and deallocate space for the thread (Richter, 1996); this takes time and CPU cycles (Ling et al., 2000). A high number of requests would cause many allocations and deallocations and becomes a significant factor affecting performance (Ling et al., 2000). When establishing a multithreading solution, it is a good idea to balance the number of threads with the number of requests.

There are several methods for implementing multithreading: Thread-per-request, Thread-per-Connection, Thread-per-Servant, Thread-Pools, and

Hybrid Architectures (Schmidt et al., 1998). Thread-per-Request could be thought of as a Thread-per-Task, similar to a Producer-Consumer model of multithreading. This work will focus on the Thread-per-Request model and the Thread-Pool model.

2.2.1 Thread-per-Request

A Thread-per-Request architecture handles each request in a separate thread of control. When a request or task is made, the architecture spawns a new thread. The thread runs the operation and exits (Schmidt et al., 1998). This architecture can incur a high cost for creating threads for many requests made at once for short-duration tasks.

Since many simulations are not based on requests but tasks, there should not be a need to increase or decrease the number of Threads-per-CPU based on the number of requests (Ling et al., 2000). When increasing the Thread-Pool size in this manner - memory space becomes a factor. Typically a useful heuristic for determining the maximum size of the pool is to set the capacity to be two times the number of megabytes of RAM on the machine (Richter, 1996).

2.2.2 Thread-Pools

Multithreading systems use the system resources more efficiently; however, creating and destroying thread objects does not come without cost (Ling et al., 2000). Thread-Pools can be leveraged for an even more significant boost to efficiency. If a program requires threads for many different algorithms, a Thread-Pool can be used to share threads for any number of processes. A Thread-Pool architecture works by pre-spawning and managing a pool of threads (Ling et al., 2000). Threads in the pool are re-used, so that thread creation and destruction overhead are only incurred once per thread (Ling et al., 2000).

Under a Thread-Pool architecture, maintaining threads in the pool incurs a run-time overhead for context-switching (Ling et al., 2000). Each thread runs on a lightweight process (LWP) (Lewis & Berg, 1995). A context-switch refers to the action of removing an active thread from its LWP and replacing it with another thread that is waiting to be run (Ling et al., 2000). On a SPARC station 10/41, a context switch requires about 20 microseconds (Lewis & Berg, 1995). Additional overhead is introduced by ensuring shared data is consistent and maintained when context switching (Agesen et al., 1999).

Knowing the optimal number of threads to create is the key to an efficient Thread-Pool architecture. If the Thread-Pool is too large, threads go unused (Ling et al., 2000). If the Thread-Pool is too small, new threads must be created to handle the new requests (Ling et al., 2000). On simulations where there is a fixed number of operations to compute per step, Thread-Pool size can likely be a fixed number. A good rule of thumb is to use two times the number available processors on the machine to appropriately leverage the hyperthreading pipeline (Richter, 1996).

2.3 Race Conditions

When establishing a multithreaded solution in a previously serial executed algorithm, the software engineer must identify and resolve race conditions. A race condition occurs when two or more threads are running in parallel attempt to read data that could be in the process of being written to by another thread. In concurrent programs, race conditions are a standard class of bugs and can be challenging to find. Traditional unit testing usually is unable to help to find all race conditions because their occurrence depends so much on timing (Claessen et al., 2009).

A recent mishap at NASDAQ during Facebook's initial public offering illustrates the effects that process-level race conditions can have. In spite of unprecedented testing efforts used on the auction system used to conduct the offering, race conditions occurred between the auction process and the calculation process. The result was corrupted data, causing the system to go into an infinite loop. This corrupted data produced a backlog of unfulfilled orders that resulted in over $40 million in damages (Yu, Srisa-an, & Rothermel, 2017).

Along with process-level race conditions, system calls can cause race conditions. Thus, testing for process-level race conditions requires accounting for the effects of write and read access and synchronization operations involving system calls. System calls operating on shared resources are typically treated as black boxes by engineers who use them to develop applications (Yu et al., 2017).

Certain antipatterns affect the correct execution of multithreaded implementation. Antipatterns are a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences (Brown, Malveau, McCormick, & Mowbray, 1998). Antipatterns such as using too many static variables can cause race conditions if these static variables are written to during asynchronous function calls.

2.4 Design Solutions

Since OpenSIMPPLLE is written in Java, and Object-Oriented Design plays a significant role in the development of the software. Java is class-based, meaning that static variables can be used to hold class data. Object-Oriented Programming (OOP) encourages the programmer to write for re-use (Smith, 1987). The re-use of code reduces the size of the program, making compilation more efficient. There is no metric for measuring when an interface is the most decomposed it can be It is up to the programmer to decide if the decomposition is sufficient (Smith, 1987).

2.4.1 Principles

Design patterns are the exact opposite of antipatterns. Design patterns are solutions to recurring problems and are the best practices where proven and working solutions exist (Hermann, 2005). Since Java is polymorphic, it is suited well to handle objects or groups of an object with different roles or interfaces (Steimann, 2000).

The model has interfaces for the generalized process event occurrence. The interfaces of Processes are ProcessOccurrence, ProcessOccurrenceSpreading, and ProcessOccurrenceSpreadingFire. Processes have methods and logic that defines their role. Spread rules and probability rules are highly reliant on this polymorphic paradigm since the protocol for spreading processes can have many implementations such as interfaces listed previously.

2.4.2 Design Patterns

Design patterns provide a target for the reorganization or refactoring of class hierarchies. Moreover, by using design patterns early in the lifecycle, one can avert refactoring at later stages of design (Gamma, Helm, Johnson, & Vlissides, 1993). The Strategy Pattern described in the Methods section is a good example of a reusable code. The Strategy Design Pattern splits an interface into separate interfaces based on differences in the execution of their functions. With this decomposition of protocol objects, managing protocol dependencies is not only possible during the design and implementation phases (between protocol classes), but also at run-time (between protocol objects) (Garbinato, Guerraoui, et al., 1997).

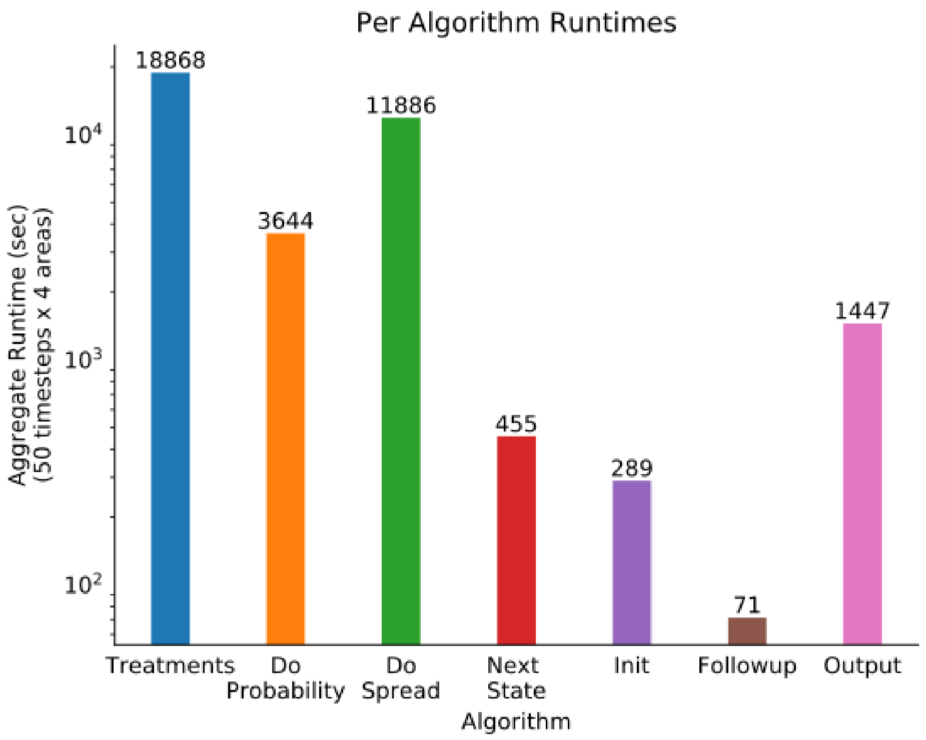
Chapter 3

Methods

It will be essential to gather information about the different algorithms to achieve an optimal runtime. By using a program clock, much like a stop-watch, the time spent in each algorithm can be determined. After gathering times for all the algorithms, the algorithms with the most extended runtime will require optimization (Figure 3.1). The algorithms that took the longest are described previously in the Introduction. In order of most time spent, the algorithms include applying treatments, spreading processes, computing probabilities of igniting a process, output to save location, the next state pathway traversal, and then initialization of EVU state.

Two of these functions will not provide any means of increasing efficiency. The initialization step is several algorithms put into one role. The total amount spent in each of the initialization algorithms is negligible, and the time optimizing the more expensive algorithms will be more productive. The

Figure 3.1: Time Spent in Algorithms



Most of the time computing the forward projection is spent in treatments, probability, spread, and next state.

output to save location is locked by the output stream regardless of any parallelization. Therefore, this study tests the treatment, probability, spread, and next state functions.

There are many ways to parallelize programs using various hardware configurations and Application Programming Interface (APIs). Several parallel techniques were considered regarding this research. Graphics/General Processing Unit (GPU) computing using Compute Unified Device Architecture (CUDA), Single Instruction, Multiple Data (SIMD) using Advanced Vector Extensions (AVX) or bit-packing, Single Program, Multiple Data (SPMD) using blocks of units, and Tasks using the native CPU threading architecture provided by the Java Virtual Machine (JVM) (Gueron & Krasnov, 2016; Muresano, Meyer, Rexachs, & Luque, 2017; Nvidia, 2012).

Considering the complex class hierarchies of OpenSIMPPLLE, implementing a solution that requires access to primitive types such as integers or floats could be difficult, leaving GPU and SIMD untenable. To achieve these solutions would require a rewrite of the EVU interface.

With parallel implementations, there is a need to identify all possible race conditions. In Java, static variables are considered class variables, essentially making them global to the program. Any variables contained outside the EVU in the Area class and Simulation class will also be global to the EVU. To handle these race conditions, any static or global variable that is not read-only will need to be made local to the EVU or containing interface.

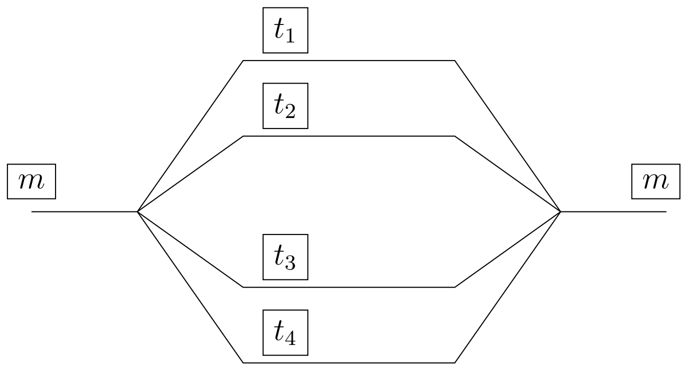
The following sections detail the procedure for implementing parallelism using a Spawn-Join model (Figure 3.2). This model effectively spawns a predetermined number of threads, and each thread goes on to complete the tasks assigned to it. Once completed, the threads join back up, and the main thread continues its execution.

3.1 Algorithms

There are two ways of implementing spawn-and-join by Single Program, Multiple Data (SPMD), or by Tasks. Spawning tasks using a thread pool assigns tasks in a Round-Robin manner where each thread gets. An Atomic Integer determines the index of the EVU. The only source of synchronization is when accessing and incrementing Atomic Integer. The SPMD approach allocates a predetermined block of EVUs for each thread to process. SPMD does not require any synchronization since each thread has a local copy of its owned EVU block.

SPMD (Single Program, Multiple Data) is a parallel technique for splitting tasks amongst threads. When using SPMD, the number of blocks should be equivalent to the number of available threads. The block size can be quickly determined by dividing the search space by the number of blocks. Each thread works on an independent block offset by the threads index and the block size.

Figure 3.2: Spawn Join Model



The main thread m spawns t_n threads. Then waits for all threads to join before continuing.

3.1.1 Treatments

Applying treatments is a set of applications where each application applies a treatment randomly to the collection of EVUs. An application gathers EVUs based on three criteria. System treatments come from user/system knowledge. Process treatments only affect EVUs with specific processes and Attribute treatments that treat based on lifeforms. Each of these criteria loops over every EVU and inserts the applicable EVUs into Vectors.

Currently, the gather phase of treatments is the most expensive operation since gathering the applicable EVUs requires a search over all EVUs. The time spent applying treatments takes much less time as the gathering process has reduced the search space. The gathering process executes three times, and the application step executes three times for every application.

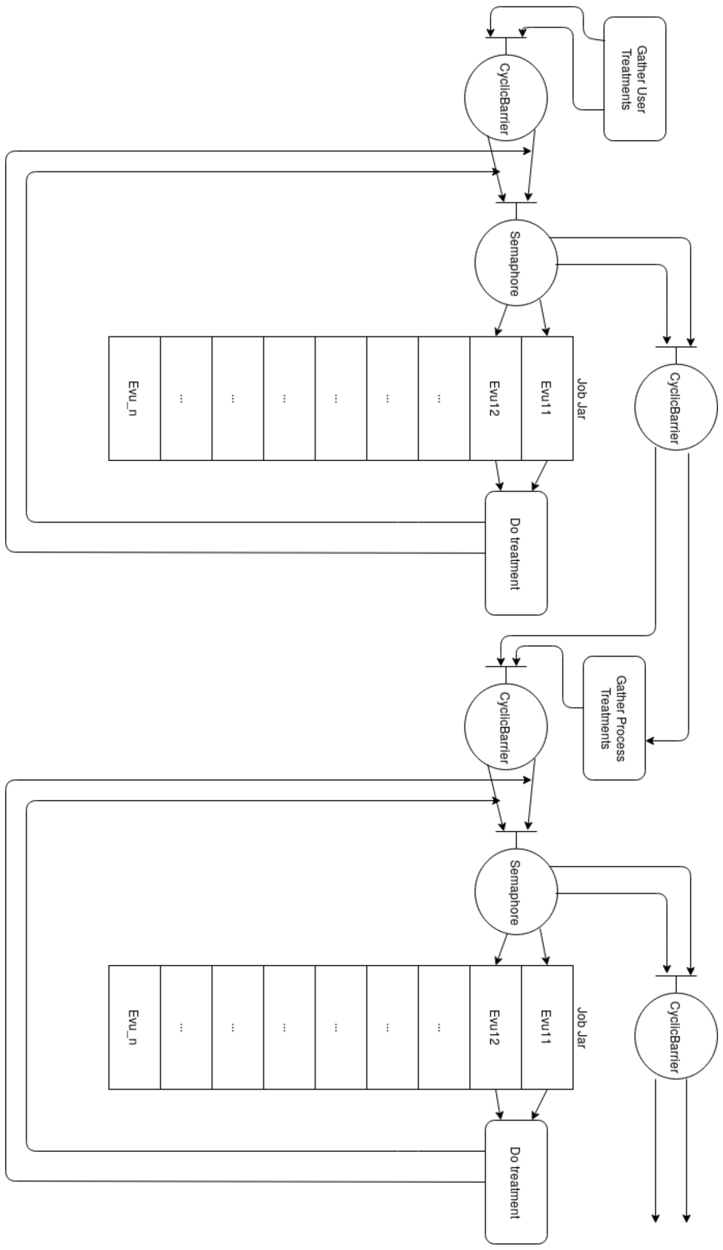
The best approach appears to be to implement a spawn-and-join threading method over the gathering process for each criterion (Figure 3.3). Since the requirement is to gather the units and place them into a vector, the gathering process gets bottlenecked by the vectors append function. Vectors are a synchronized data structure and require each thread to wait to read or write to it. The read and write operations might cause a concurrent modification error since the append and remove function synchronize at the function level. The gathering process is write-only as each applicable EVU is added to the vector.

Treatments can be refactored to apply treatments to EVUs as they are found to be applicable. Refactoring, in this manner, will reduce the number of times the Thread-Pool architecture needs to context switch. This approach eliminates the need for synchronization since the applicable EVUs are treated at once.

3.1.2 Ignition Points

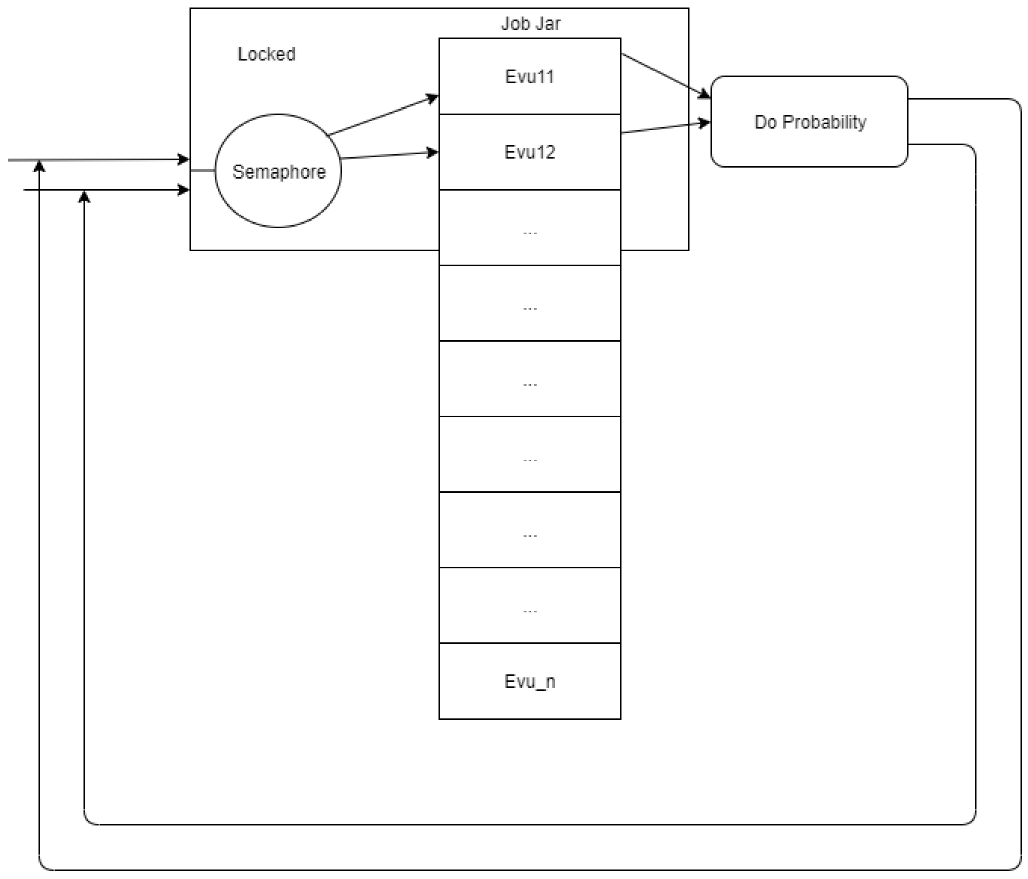
Igniting processes is a straight forward process. Every EVU computes a probability that any of the available processes ignites within its space. There is no overlap between EVUs, so there should be no need for synchronization. Much like applying treatments in a parallel fashion, igniting processes can be implemented using Tasks or SPMD (Figure 3.4 and 3.5).

Figure 3.3: Diagram of Parallel Treatments



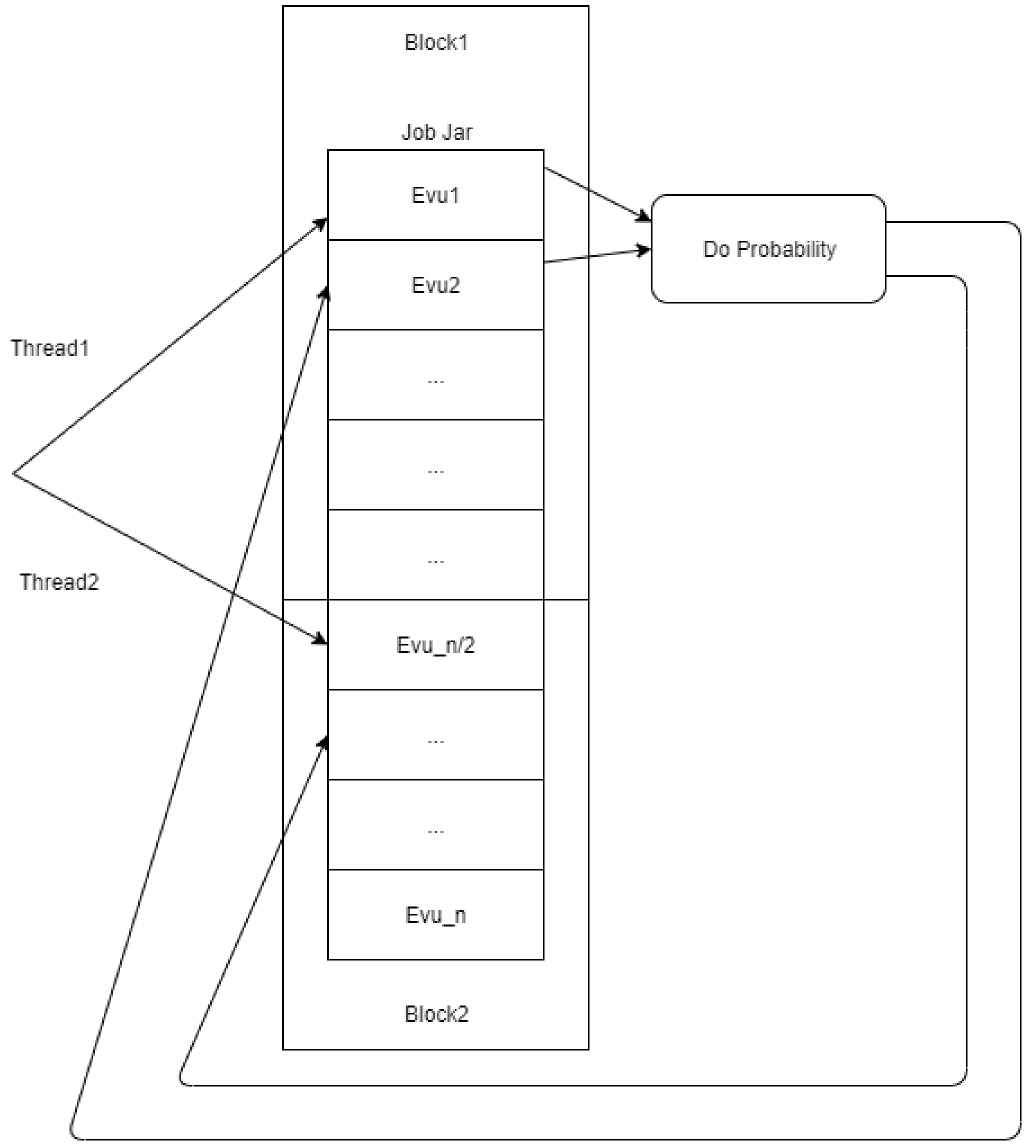
Treatments come in three parts. Pictured here are two parts for gathering and processing treatments for User Treatments and Process Treatments. The next step is to compute Attribute Treatments. The process is the same as the previous two.

Figure 3.4: Diagram of Parallel Ignition Points Using Tasks



A Task based solution for computing ignition points. EVUs are sequentially accessed from a list by any number of threads.

Figure 3.5: Diagram of Parallel Ignition Points Using SPMD



A SPMD based solution for computing ignition points. EVUs are sequentially accessed from a within their respective blocks.

3.1.3 Spreading Events

Events propagate over the map. An event corresponds to an ignition point if and only if the ignition point is a spreadable process. Spreadable processes include fires and various insects like Mountain Pine Beetle. The processes capable of spreading across the landscape implement their spread function within their respective interfaces.

Fire events spread using two algorithms using a spread queue. The event is still active while there are EVUs in the spread queue. The two methods are the SIMPPLLE algorithm and the Keane Cell Percolation algorithm (Keane et al., 2006).

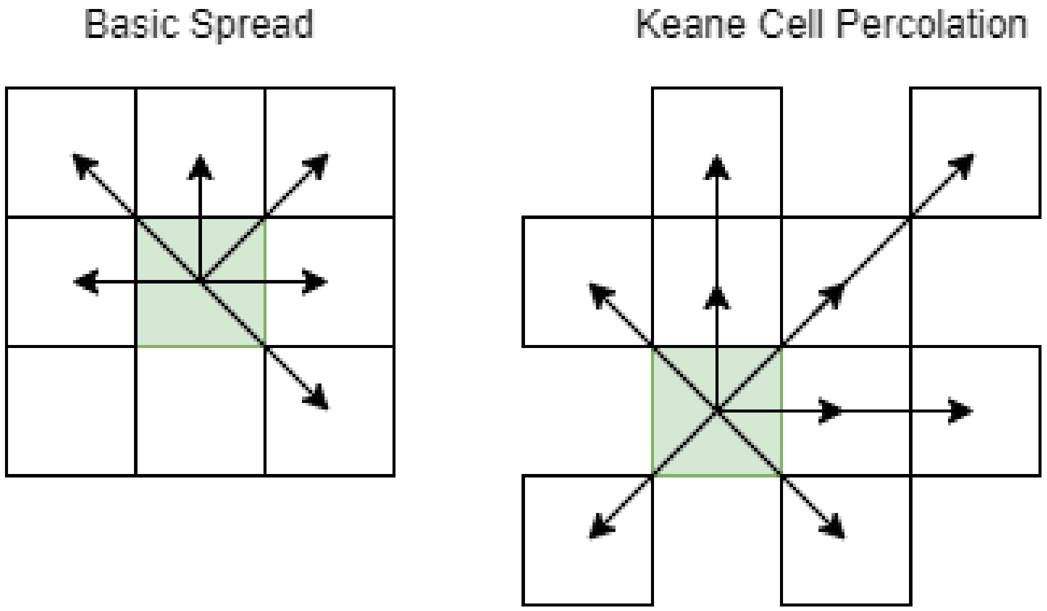
The SIMPPLLE algorithm spreads the fire events to immediate neighbors taken from the spread queue. If the fire event spreads to another EVU, that EVU is put into the spread queue.

The Keane Cell Percolation (KCP) algorithm spreads fire events along a path corresponding to the slope of the landscape and a wind vector (Keane et al., 2006). Fires spread faster uphill and in the direction of the wind. The KCP algorithm creates oblong-shaped fires that are more realistic than the block-like SIMPPLLE algorithm.

Parallel events have a possible race condition in that events can contain overlapping EVUs (Figure 3.6). If two events are trying to write a process to the same EVU, it is not known which event should modify the EVU. This work investigated two ways of evaluating the overlap of events.

The first solution to handling the overlap of events is to lock the EVU

Figure 3.6: Overlapping Events



Both of the spread algorithms experience event overlap. If run in parallel, these methods will experience race conditions over these overlapping regions.

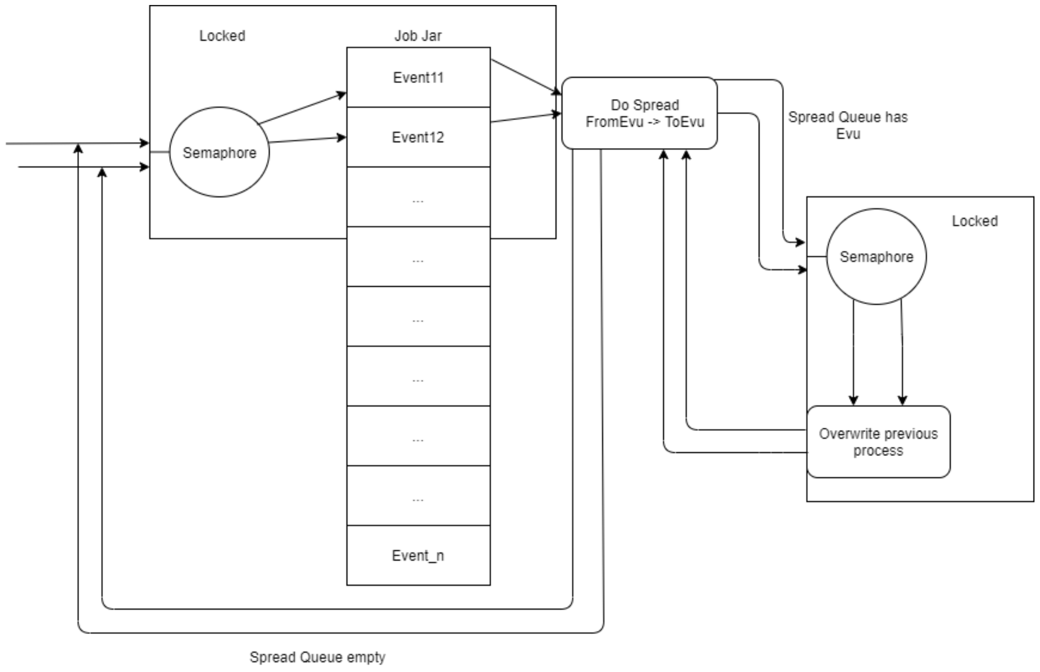
to which the event would like to spread (Figure 3.7). This method will be referred to as the Locked Overwrite method. By using a pool of semaphores, an EVU can hash to a particular semaphore. If the EVU already has been accessed, the semaphore associated with its hash will be locked. Locking the EVUs ensures that access to the global spread list stays consistent with the data associated with EVUs.

The number of Semaphores can be a fixed number of about 100 or so. Considering there are only a maximum number of EVUs accessed at any one time, having a moderate amount of semaphores ensures a broad distribution of semaphores for EVUs to handle. Some EVUs will clash with EVUs other than itself but is not likely to happen very often due to the relatively small number of concurrent events.

The second solution is to allow all of the events to spread to their maximum size and to filter out the overlapping sections in post-processing (Figure 3.6). Achieving the Overlap Filtering technique requires that spread events keep a local list of the processes being spread.

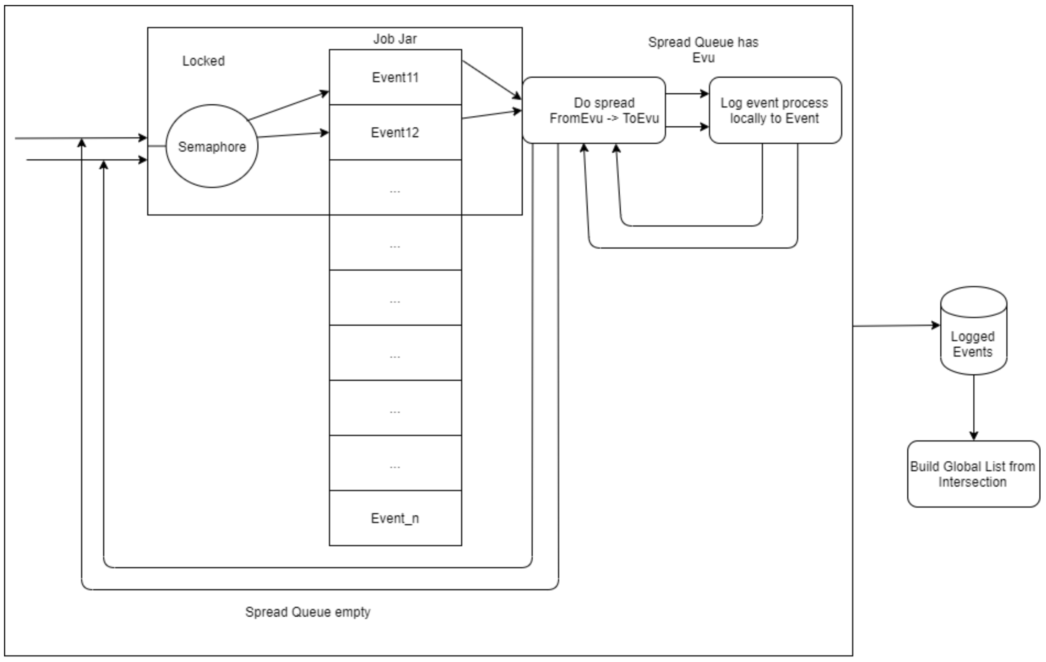
When all spread events have concluded, the local process lists are gathered and filtered. The filtering used is an upper triangular search over all the event lists. Each search checks for EVUs that are shared amongst the other lists and chooses which process has precedence in the process hierarchy. The priority of processes is that fires burn everything, and other processes fall in line by the most recently established process.

Figure 3.7: Diagram of Parallel Spread: Locked Overwrite



The Locked Overwrite (LO) method allows processes to be overwritten. The EVU is locked so that other spread events cannot access the EVU until the current spreading event is finished.

Figure 3.8: Diagram of Parallel Spread: Overlap Filter



The Overlap Filter (OLF) method allows all events to spread independently until the event concludes. The events are coalesced and processes are filtered so that each EVU contains one process. The process is then applied to the EVU.

3.1.4 Next State

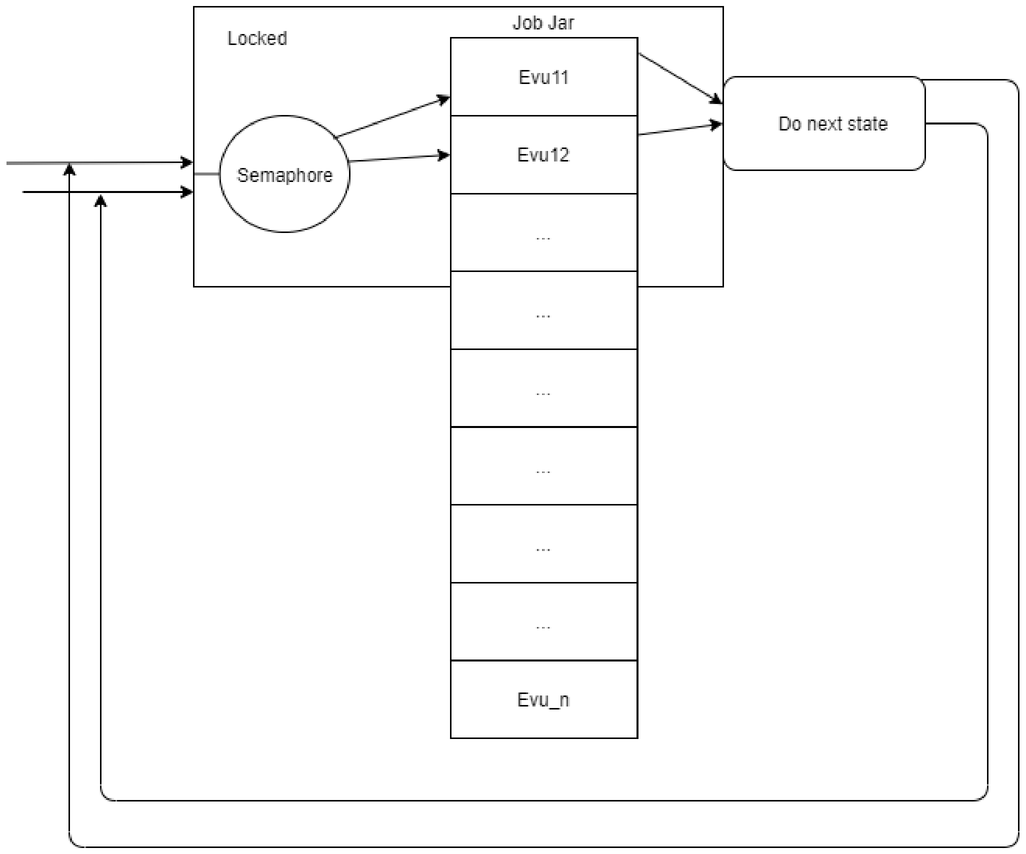
The next state algorithm is much like the ignition process. The algorithm does not overlap, and Tasks or SPMD can be utilized to parallelize over the collection of EVUs (Figure 3.9).

3.2 Handling Race Conditions

The state of EVUs is a critical factor for proper execution in these algorithms. Proper implementation is required because EVUs contain a history of past states, so any change of an EVUs state affects the execution of the algorithm in future timesteps. If those are inconsistent, then the output will also be inconsistent.

The first part of state data that requires attention is a static variable that holds a current life form. The global lifeform is derived from multiple lifeform simulations where every EVU that has several different lifeforms. Since static variables are common to all instances of a class, the program has to access the class information across the entire hierarchy. The second piece of state information is a static ProcessOccurrence stored in the FireEvent class. The FireEvent class is a non-instantiated abstract class. Refactoring this variable will require a redesign of how the FireEvent is stored in the threads interface.

Figure 3.9: Diagram of Parallel Next State



The Next State algorithm is another straightforward process that can be parallelized by accessing EVUs from a list and assigning them to any number of threads.

3.2.1 Converting CurrentLifeform to Local Variable

Solving the global lifeform problem requires local data to be passed down the Process interface. Parameter references are local to the function being called, resolving the issue with global access to CurrentLifeform. The Process interface is overloaded to pass the CurrentLifeform from the EVU. Any instances of state in the Process interface will have to give the local lifeform to the state information. Luckily, the function to do so already existed in the EVU interface.

The appropriate interfaces will have to reflect this overloaded parameter change to preserve functionality. This causes the monotonic growth of the classes since the original functions are preserved, and overloaded functions are added to the code. The monotonic increase is perhaps one of the most significant drawbacks to this pattern. Considering that there are many places the current lifeform is used, it could be challenging to ensure that all the interfaces are adhering to the new interface if they require it.

To get around this limitation, and reduce the monotonic growth of code, the current lifeform can be saved as a state of the EVU. Saving the variable as a state variable should resolve the issue of overloading interfaces to fit the new parameter. The solution requires keeping one reference of the currentLifeform per EVU, then acquiring the reference from the EVU wherever the global variable is read. This method is a much simpler solution to implement since the EVU is always present where the CurrentLifeform is set and accessed, and no overloading must occur.

3.2.2 Handling Static FireEvent

Upon inspecting the rest of the algorithms, another global variable surfaced. This variable is a current event being processed as a FireEvent. FireEvent is an abstract class, so it has no instantiated object to hold its state. Instead, another container will have to contain the FireEvents state.

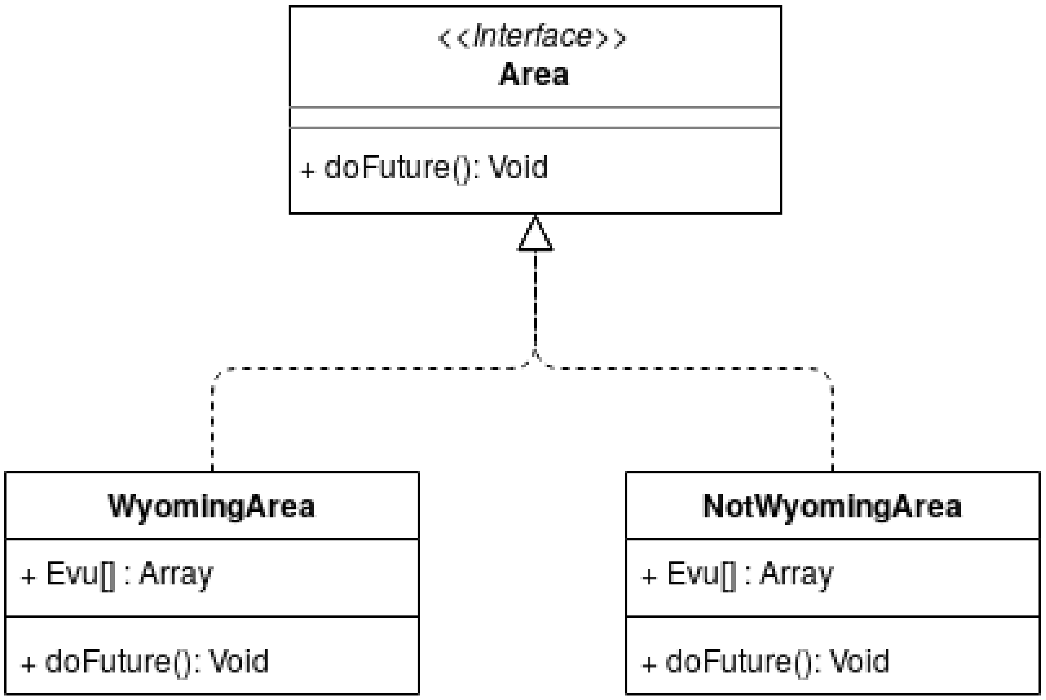
The event is being spread by the Area, which is a global object; the event can be cached in a small array in the Area. The size of the array should be the number of threads being used to spread the processes. Each thread has a unique identifier that can be used to index into the array and grab the FireEvent object. The index of the thread can then be passed down the call stack to where the event is required.

3.3 Design Pattern Solutions

3.3.1 Strategy Pattern

There are many different Design Patterns; this work focused on the Strategy Pattern. The Strategy Pattern separates interfaces to handle different sets of protocols. OpenSIMPPLLE contains a number of protocols that are specific to the Wyoming landscape. These protocols rely on a function referred to as isWyoming. To reduce the number of calls to isWyoming, we can define an interface for each approach as an implementation of the Area class (Figure 3.10). The implementations of the Area interfaces allow us to explicitly

Figure 3.10: Diagram of Strategy Pattern



The Strategy Pattern separates the protocols between the two extensions of the base class Area. The WyomingArea contains protocols for Wyoming areas, and the opposite is true for NotWyomingArea. This should cut down on branch statements that query if the Area isWyoming.

define behavior appropriately for areas that require the Wyoming algorithms without having to check for them.

With that notion of implementing based on behavior, any other classes that refer to the isWyoming function can also be reimplemented so that there is an implementation for Wyoming calls and one without Wyoming. Adding these separate classes will cause the growth of the codebase since several other methods will need to be reimplemented as well.

Chapter 4

Results

4.1 Defining Benchmarks

To get a reliable benchmark for runtime overall, and reduce the influence of the stochastic process of igniting processes, four areas at ten runs with five timesteps each were recorded. Table 4.1 shows the sizes of each area. The areas vary slightly in size, and each has different system knowledge and treatment schedules. This method of testing multiple runs allows a bit of variability in the testing process to get an average runtime with ten data points per timestep.

The computing environment was a Ryzen 7 1800x 8-core CPU (Central Processing Unit) at stock 3.6 GHz along with 16 GB DDR4 2400 MHz DRAM. The Ryzen chips are hyperthreaded, which means this particular CPU has 16 logical processors. The Java 8 environment was run with 8 GB

Table 4.1: Area names and sizes

Area Name	# of Units
Montane East	55,431
Montane West	50,713
NRV Climate	30,884
East Prairie	53,733

of heap space to account for the large areas.

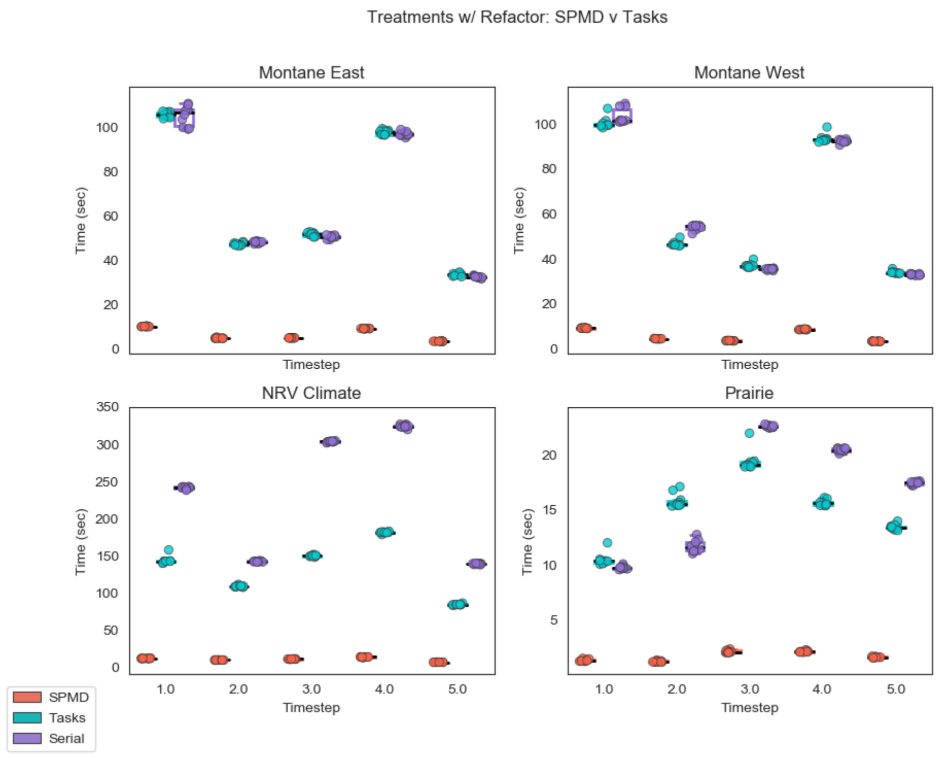
4.2 Parallelization

The following sections detail the results of parallelizing OpenSIMPPLLE. Each result was gathered without parallelization of the other algorithms.

4.2.1 Treatments

Parallelizing treatments proved to be the most difficult to implement. The parallel implementation performed much better with SPMD rather than Tasks (Figure 4.1 and 4.2). It is the only result that included a refactored solution that improved runtime (Figure 4.3). Combining the gathering step with the processing step allowed the amount of the thread pool to context switch from 6x times per application to 3x per application. If a timestep had 90 applications, the unrefactored version would context switch 540x as opposed to 270x. Some treatment schedules contain up to 300 applications for one timestep. The refactored solution saved, on average, 10% was saved over the SPMD implementations.

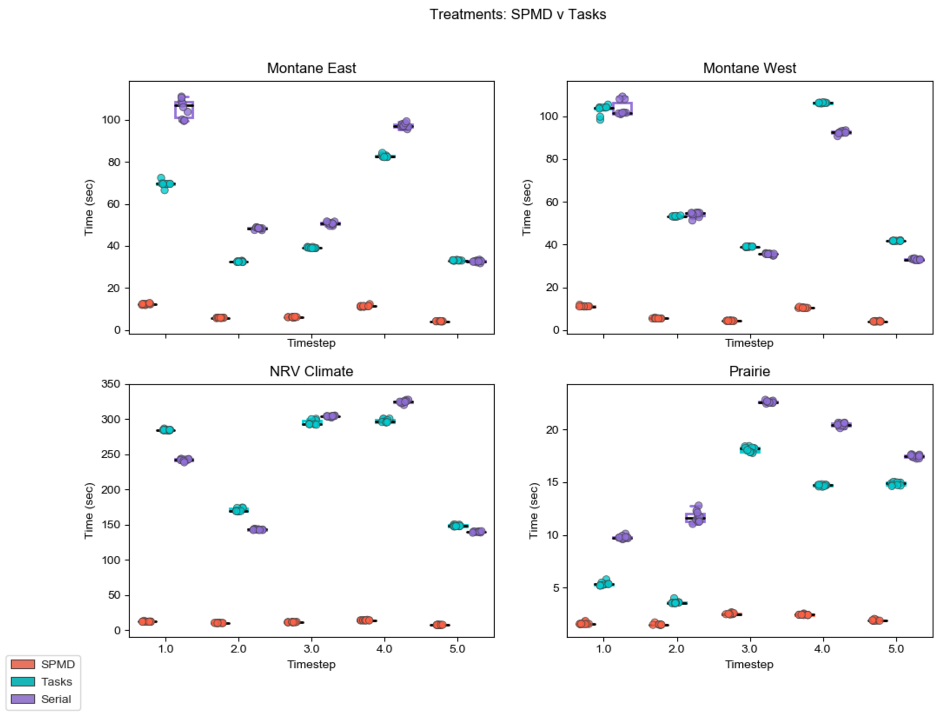
Figure 4.1: Treatments: SPMD v Tasks



Without the refactor of the Treatment algorithm, the Tasks approach did not see a significant increase in runtime. The Tasks approach ran about 1.8% faster than serial. The SPMD version performed very well with an improvement of 1,350% or 13.5x faster than that of serial.

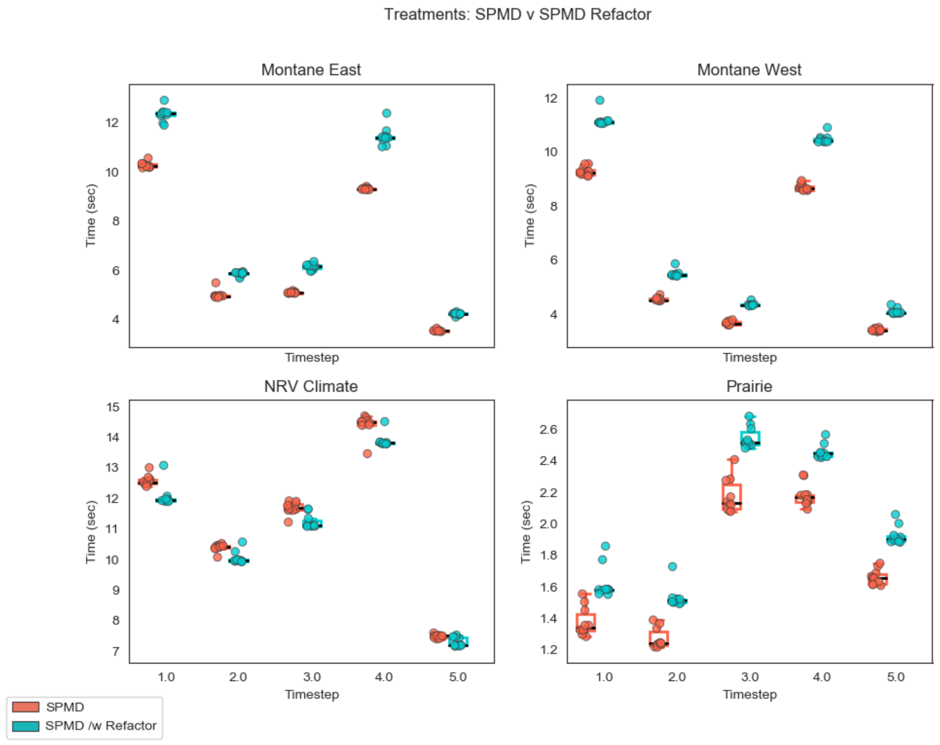
With that, the serial method of computing treatments took, on average, 94.34 seconds per timestep. The parallel Tasks approach took on average 92.64 seconds per timestep and 69.72 seconds with the refactor while the SPMD approach took on average 7.01 seconds per timestep and 6.4 seconds with the refactor. The best approach with SPMD and the refactor resulted in a 14.75x increase in runtime efficiency.

Figure 4.2: Treatments w/ Refactor: SPMD v Tasks



After refactoring the treatment algorithm, the Tasks approach improved significantly. The Tasks approach improved runtime by 26.1% from serial. The SPMD improved slightly by 0.7 seconds. SPMD with the refactor improved the runtime by 1,474% or about 14.74x the runtime of serial.

Figure 4.3: Treatments: SPMD Comparison



With decreasing the amount of context switching and synchronization with the gather process garnered these fastest times. The runtime was reduced by 10% in the SPMD implementations.

4.2.2 Probability

Computing the ignition points is a straight forward process where every EVU computes a series of logical steps. The average serial runtime was about 18.43 seconds per timestep. The parallel implementation ran on average, 4.74 seconds per timestep (Figure 4.4).

When the SPMD approach was implemented, there was little difference in runtime compared to the Tasks approach. The SPMD version ran at an average of 4.81 seconds compared to the 4.74 seconds of the Tasks approach. Overall, both methods garnered a 4x increase in runtime efficiency.

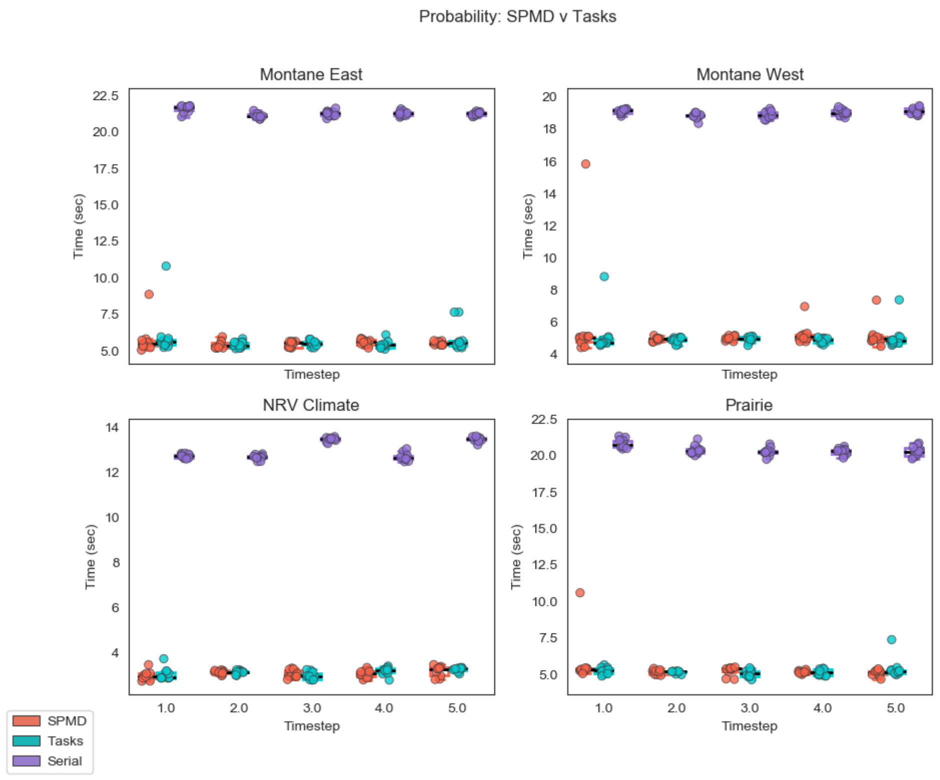
4.2.3 Spread

Computing the spread of process across the map is a much more complicated function. Each event that is spreading can overlap. These overlap points require synchronization or local event access on some level with the implementations. The two methods implemented were Locked Overwrite and the Overlap Filter.

The Locked Overwrite (LO) locks access to EVUs based on a collection of Semaphores. If an EVU hashed to a Semaphore and it was locked, the EVU had to wait for the lock to be released before it's spread process could continue. The Locked Overwrite method performed well at, on average, 12.6 seconds per timestep (Figure 4.5).

The Overlap Filter (OLF) method does not require synchronization at the

Figure 4.4: Parallel Probability: SPMD v Tasks



Runtimes of the doProbability function comparing SPMD and Tasks implementations. There was no marked increase from using one implementation over the other. In terms of engineering these solutions, SPMD is a better choice since it relies on the data structure that was previously used by the program.

global level since it keeps a local copy of spread summaries for each thread. The drawback is that there is an extra step to compute the overlap after all spread events have completed. The Overlap Filter did marginally worse than the Locked Overwrite method in three areas and had a spike in runtimes in the NRV area. On average, with the peaks factored in, it ran in 86.4 seconds per timestep,

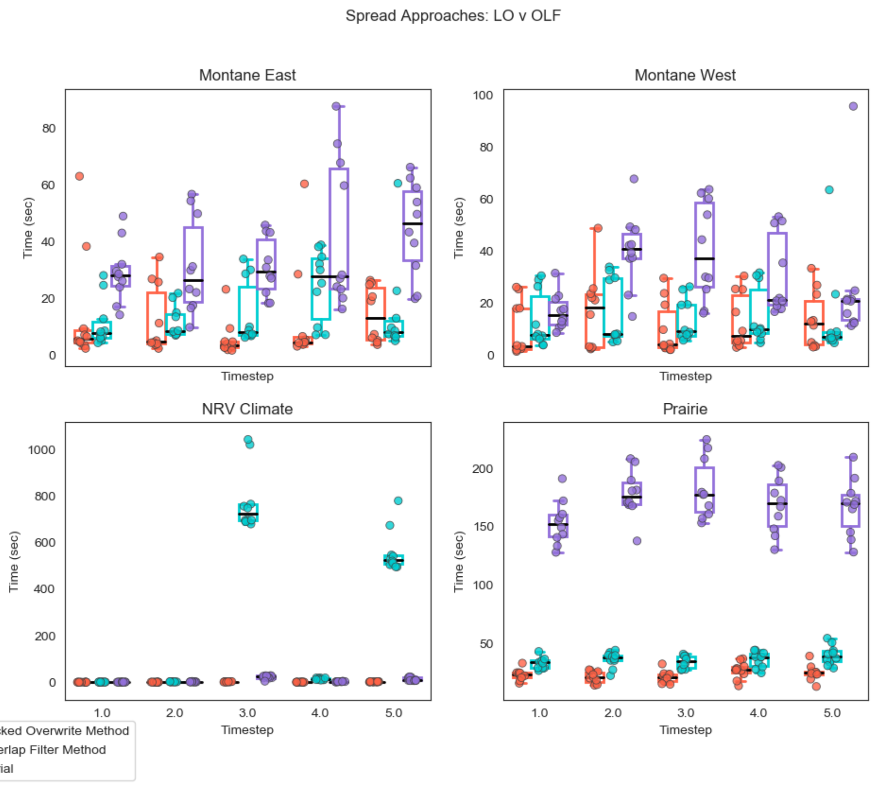
It is important to note that the runtime of the Overlap Filter could have been influenced by the fact that events spread to their full extent and are not stopped by other events. This is likely caused by many fire events that grow to the most extreme size they can get.

The serial version of the spread algorithm clock in at 61.09 seconds per timestep while the Locked Overwrite method, the program gained more than 5x the amount of runtime efficiency.

4.2.4 Next State

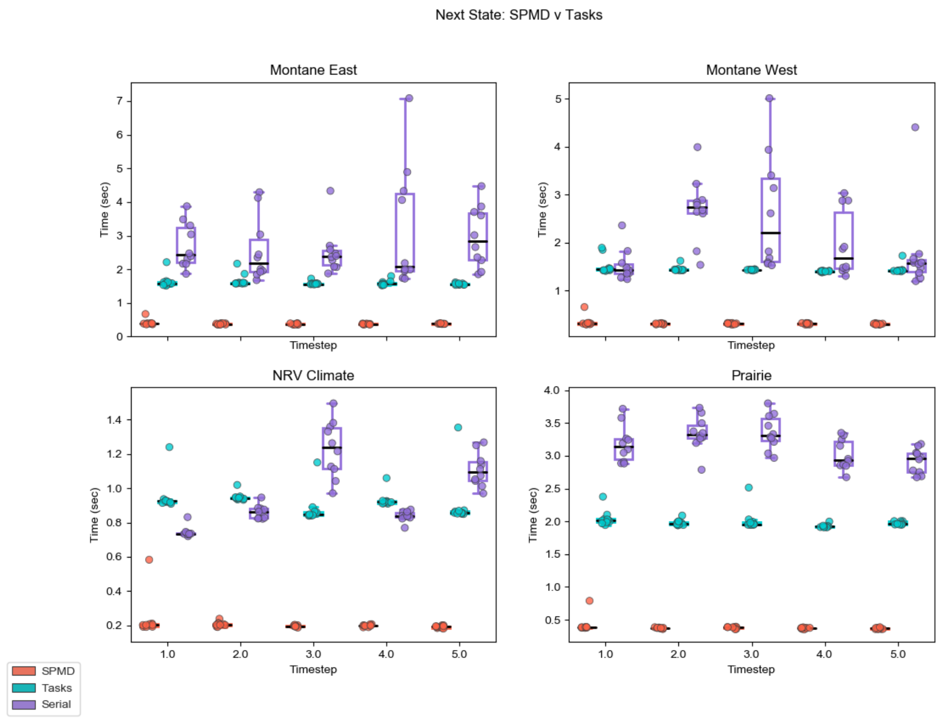
Parallelizing the next state algorithm using the pathways tree yielded positive results (Figure 4.6). The serial approach took, on average, 1.6 seconds per timestep, while the Tasks method took, on average, 1.4 seconds to complete per timestep. The SPMD approach, on average, took 0.325 seconds per timestep. The work done during the next state algorithm was so little that any synchronization did not net a significant increase in runtime efficiency. The SPMD method netted a 5x increase in runtime efficiency.

Figure 4.5: Parallel Spread Results



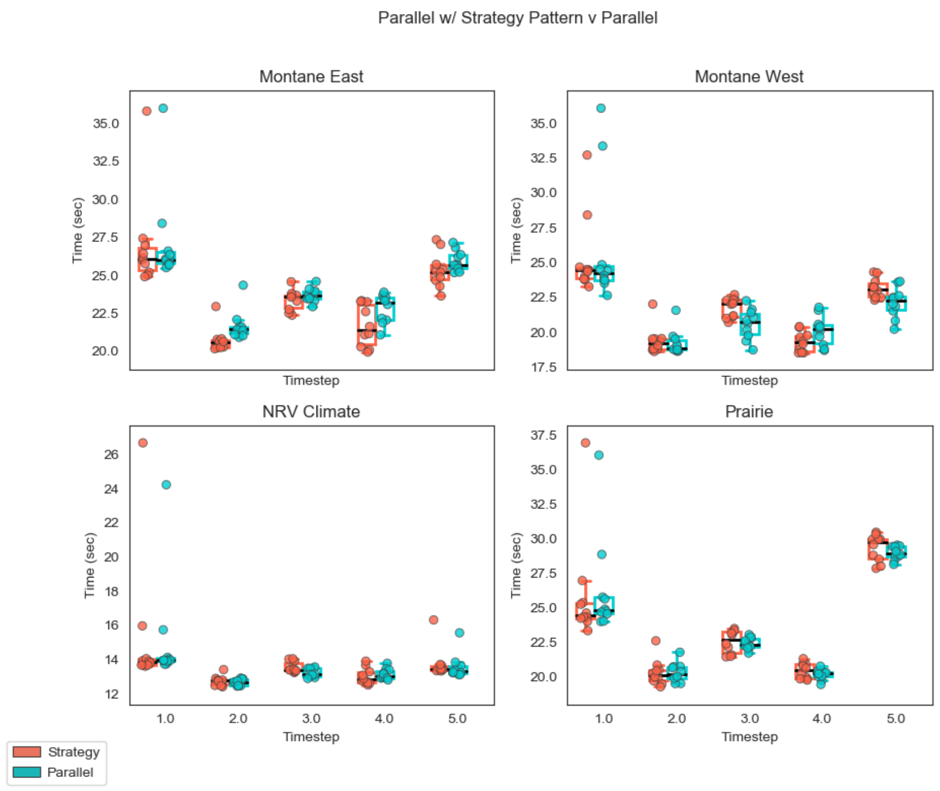
After parallelization, the spreading algorithm gave some interesting results. On par, the Locked Overwrite did the best while an edge case caused the Overlap Filter to run a long time in one area. The edge case is likely that events are spreading to their largest size, or this parallelization technique causes a climate event that takes a long time to resolve. The LO method improved runtime by 484%.

Figure 4.6: Parallel Next State: SPMD v Tasks



Parallelizing next state with Tasks provided a speed up overall with one area being slower on average. The difference is minute at $\approx 0.2sec$. Over 200 timesteps it would only accrue 40 secs. Given that it is such a short amount of time, synchronization was an issue. SPMD resolved that contention and allowed better parallel performance. The runtime improved by 492%.

Figure 4.7: Strategy Pattern Results



The strategy pattern did not improve runtime significantly. Less than 1% of runtime was saved with the strategy pattern.

4.3 Strategy Pattern

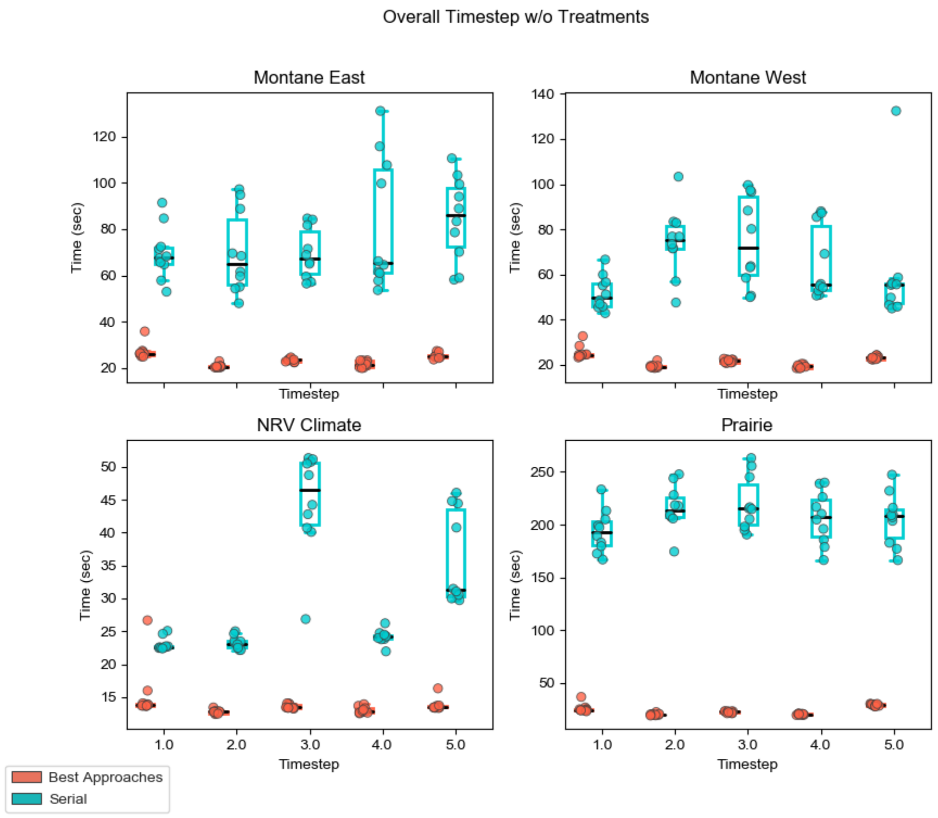
The strategy pattern, on average, took slightly less time to run. (Figure 4.7). Most likely, the Just-In-Time (JIT) compiler in Java 8 is optimizing the isWyoming variable already, or the time to check these variables is not a sufficient amount of time to optimize. The runtime did improve by 0.33%.

4.4 Overall

Overall, with the best methods of parallelization and refactorization, Open-SIMPPLLE gained efficiency. The program was run without treatments and achieved a 4.71x speedup. Seen in Figure 4.8, the runtime is reduced from 94.71 seconds to 20.77 seconds per timestep.

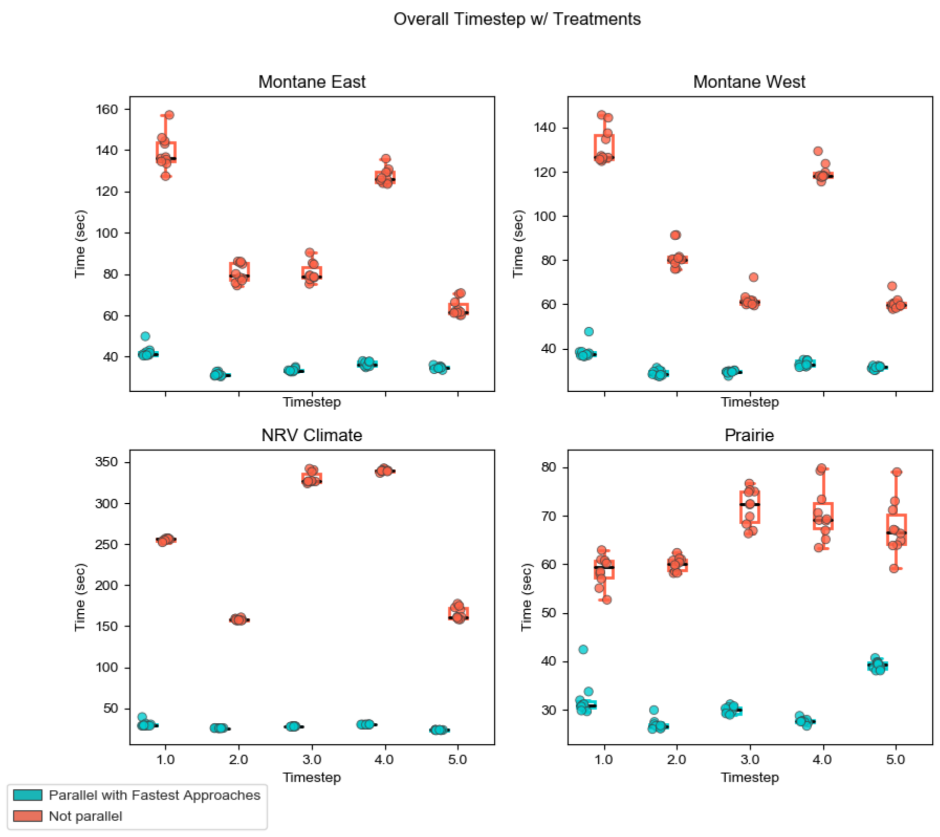
As evidenced in Figure 4.9, with the treatment schedule in place, the runtime for each timestep was reduced from 126.24 seconds to 31.6 seconds. The increase in efficiency is about 4x overall. Without SPMD blocks, the average timestep was about 117.62 seconds.

Figure 4.8: Overall Best Approaches w/o Treatments



Overall runtime savings without treatments was around 4.71x.

Figure 4.9: Overall Best Approaches With Treatments



Running simulations with treatments usually takes longer since the run time of the treatment algorithm dominates the run time. Parallelizing treatments improved run time by 4x.

Chapter 5

Discussion

5.1 Parallelism

Adding parallelism improved the runtime of all of the main algorithms. Overall, the SPMD and Task approaches were mixed in terms of which algorithm performed better. The SPMD approach improved the runtime for the treatment algorithm and the next state algorithm over the Job Jar approach. SPMD performs better in these cases due to the reduced contention over access to the EVUs data structure. The results of this improvement can be seen in Figures 4.1 and 4.6.

Parallelizing treatments benefitted from a refactor seen in Figure 4.2 and Figure 4.3. Since there are several separate algorithms in treatments that are run as a parallel program, switching between threads became a significant bottleneck. By reducing the number of runnable parallelizable functions from

6 to 3, the runtime was reduced by 21.6% and 10% for Tasks and SPMD, respectively.

The SPMD performed the best when the Job Jar had the most contention. Seen in Figure 4.6, this contention frequently occurred in the Next State algorithm, where there is little work to be done that threads accomplish their task before having to wait on access to the Job Jar. Taking out the contention and allocating blocks of EVUs to a thread allowed threads to run without any synchronized sections or semaphores, which improved performance.

5.2 Refactoring and Patterns

5.2.1 Refactoring

Refactoring can have an impact on parallel implementations. Considering that refactoring treatments to search for EVUs and apply treatments inline saved 10% of the runtime of the algorithm. The previous implementation, otherwise, added EVUs to a Vector that created a synchronization zone that slowed down the algorithm.

5.2.2 Strategy Pattern

Reducing the number of calls to the isWyoming function, reduced the runtime over several timesteps (Figure 4.7). The runtime was not impacted by the Strategy Design Pattern over many runs, however. It might be the case that the isWyoming variable was optimized during runtime by the JIT.

5.3 Overall Results

By combining the best approaches, each area ran much faster than the serial version (Figures 4.8 and 4.9). There are significant bottlenecks to consider when engineering solutions to parallel strategies. The primary concern in OpenSIMPPLLE was thread synchronization over algorithms that take relatively small amounts of time to complete. By engineering out the synchronization zones, threads no longer have to wait for other threads to complete their tasks.

5.4 Conclusion

Engineering solutions to solve slow runtimes can be complicated, given how the target algorithm behaves. If there is a large amount of work to do per thread, synchronization, and locked sections will not have a significant effect on reducing efficiency. If the amount of work is small, an SPMD approach may be advantageous. Also, reducing the amount of time context switching between threads can improve runtimes, so figuring out ways to combine steps will reduce the amount of time the JVM spends managing the thread pools state.

By Amdahl's argument, the maximum speedup is the inverse of the fraction of time the task must take on a single thread (Rodgers, 1985). With 16 threads, a task that takes 16 seconds serially should take 1 second in parallel. The algorithms involved in this work did not exhibit this rule. Other factors play into the runtime. Treatments were close to the standard except for some serial follow-up routines that clean up data after applying treatments. Spread events overlap and create a bottleneck when accessing EVUs. Computing ignition points and the next state also did not follow this rule for an unknown reason. Likely the overhead was incurred by incrementing atomic integers or accessing synchronized data structures.

After parallelization, applying the Strategy Pattern did not increase efficiency in the program's runtime. With that in mind, it is easy to see that using patterns to make code more efficient is not as important as adding asynchronous or parallel implementations to take advantage of resources that would otherwise sit idle.

This research established that Parallel implementations improve runtime significantly. Parallel programs run more efficiently when the amount of synchronization is reduced. The amount of time engineering a design strategy did not enhance runtime due to JIT optimizations. From here, OpenSIMPPLLE has the potential to improve it's efficiency by implementing faster, refactored, and unsynchronized algorithms.

References

Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y., & White, D. (1999). An efficient meta-lock for implementing ubiquitous synchronization.

Berg, C. (1996, 11). How do threads work and how can i create a general-purpose event? *Doctor Dobbs Journal*, *21*, 111-+.

Brown, W. H., Malveau, R. C., McCormick, H. W., & Mowbray, T. J. (1998). *Antipatterns: refactoring software, architectures, and projects in crisis.* John Wiley and Sons, Inc.

Calcote, J. (1997). Thread pools and server performance.

Chew, J. D., Moeller, K., & Stalling, C. (2012). Simpplle, version 2.5 user's guide. *Gen. Tech. Rep. RMRS-GTR-268WWW. Fort Collins, CO: US Department of Agriculture, Forest Service, Rocky Mountain Research Station. 363 p.*, *268*.

Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., & Wiger, U. (2009). Finding race conditions in erlang with quickcheck and pulse. *ACM Sigplan Notices*, *44*(9), 149–160.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In O. M. Nierstrasz (Ed.), *Ecoop' 93 — object-oriented programming* (pp. 406–431). Berlin, Heidelberg: Springer Berlin Heidelberg.

Garbinato, B., Guerraoui, R., et al. (1997). Using the strategy design pattern to compose reliable distributed protocols. In *Coots* (pp. 221–232).

Gueron, S., & Krasnov, V. (2016). Fast quicksort implementation using avx instructions. *Computer Journal*, *59*(1), 83–90.

Hermann, U. (2005). Antipatterns–theory and practice.

Keane, R. E., Holsinger, L. M., & Pratt, S. D. (2006). Simulating historical landscape dynamics using the landscape fire succession model landsum version 4.0. *Gen. Tech. Rep. RMRS-GTR-171. Fort Collins, CO: US Department of Agriculture, Forest Service, Rocky Mountain Research Station. 73 p.*, *171*.

Lewis, B., & Berg, D. J. (1995). *Threads primer: a guide to multithreaded programming*. Prentice Hall Press.

Ling, Y., Mullen, T., & Lin, X. (2000, April). Analysis of optimal thread pool size. *SIGOPS Oper. Syst. Rev.*, *34*(2), 42–55. Retrieved from `http://doi.acm.org/10.1145/346152.346320` doi: 10.1145/346152.346320

Muresano, R., Meyer, H., Rexachs, D., & Luque, E. (2017). An approach for an efficient execution of spmd applications on multi-core environments. *Future Generation Computer Systems-The International Jour-*

nal Of Escience, *66*, 11–26.

Nvidia, C. (2012). Nvidias next generation cuda compute architecture: Kepler gk110. *Whitepaper (2012)*.

Richter, J. (1996). *Advanced windows*. Microsoft Press.

Rodgers, D. P. (1985). Improvements in multiprocessor system design. In *Proceedings of the 12th annual international symposium on computer architecture* (p. 225231). Washington, DC, USA: IEEE Computer Society Press.

Rollins, M. G., Keane, R. E., Zhu, Z., & Menakis, J. P. (2006). An overview of the landfire prototype project. *In: Rollins, Matthew G.; Frame, Christine K., tech. eds. 2006. The LANDFIRE Prototype Project: nationally consistent and locally relevant geospatial data for wildland fire management Gen. Tech. Rep. RMRS-GTR-175. Fort Collins: US Department of Agriculture, Forest Service, Rocky Mountain Research Station. p. 5-43*, *175*.

Schmidt, D. C., et al. (1998). Evaluating architectures for multi-threaded corba object request brokers.

Smith, R. (1987, 1). Panel on design methodology. *SIGPLAN Not.*, *23*(5), 91–95. Retrieved from `http://doi.acm.org/10.1145/62139.62151` doi: 10.1145/62139.62151

Steimann, F. (2000). On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering*, *35*(1), 83–106.

Yu, T., Srisa-an, W., & Rothermel, G. (2017). An automated framework to support testing for process-level race conditions. *Software Testing, Verification and Reliability*, *27*(4-5), e1634. Retrieved from `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1634` (e1634 stvr.1634) doi: 10.1002/stvr.1634