ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Comparative Benchmarking of Multithreading Solutions for JVM Languages: The Case of the Alchemist Simulator

Tesi di laurea in:
LABORATORY OF SOFTWARE SYSTEMS

Relatore
Dott. Danilo Pianini

Candidato
Kyrillos Ntronov

III Sessione di Laurea
Anno Accademico 2023-2024

Abstract

Re-implementing sequential algorithms with parallelization support often leads to a tangible improvement in the time and throughput of the execution. Designing for multithreading is especially beneficial in the context of long and slow computations such as discrete event simulation, where even a relatively small increment in throughput may cut down simulation time by a significant cumulative margin. Discrete event simulation is a notoriously challenging domain to parallelize due to the complexity of the nature of the discrete events and the necessity to account for and resolve the causality conflicts. A truly deterministic solution is difficult and is not always possible for some of the more complex simulation models and domains. However, some compromises may be reached by relaxing the determinism constraints and adopting the so-called optimistic approach to conflict resolution, sacrificing some degree of predictability and determinism for performance.

Furthermore, when considering the goodness of a solution, a simple naive approach is not sufficient. The programming languages running on the Java Virtual Machine (JVM) have certain quirks and properties that must be taken into consideration to produce valuable and insightful results. Hence adopting a thorough method of testing and benchmarking is necessary.

The recent developments in the Java programming language have introduced novel solutions to structuring multithreaded code such as virtual threads that compete with a more mature analogous implementation found in the Kotlin programming language.

This thesis project explores the optimistic parallelization of a general discrete event simulator "Alchemist", building a robust benchmarking harness and testbed and comparing the results of equivalent implementations of the algorithm in traditional Java threads, the new Java virtual threads, and the consolidated Kotlin co-routines.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Danilo Pianini, for their invaluable guidance, support, expertise, and patience throughout this journey of my master's thesis.

I am deeply thankful to my family and especially my mother, for their love and support during this process. Without their encouragement and motivation, I would not have been able to complete this journey.

Finally, I would like to extend my gratitude towards my colleagues and friends who gave their sincere support and compassion.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Computer Simulation . . . 1
1.2 Discrete Event Simulation . . . 3
1.3 Parallel Discrete Event Simulation . . . 4
1.4 Designing Parallel Systems . . . 7
1.5 Parallelism in JVM . . . 9
1.6 Benchmarking Parallel Architectures on JVM . . . 13
1.7 Chemical-Oriented Simulation . . . 16
1.8 Alchemist Simulator . . . 17
1.9 Project Goals and Motivations . . . 21

2 Analysis **23**
2.1 Launch Configuration Rework . . . 25
2.2 Parallel Engine . . . 27
2.3 Benchmark Harness . . . 29

3 Design **33**
3.1 Launch Configuration . . . 33
3.2 Parallel Engine . . . 38
3.3 Benchmark Harness . . . 42

4 Implementation **45**
4.1 CLI Refactor . . . 45
4.2 Arbitrary YAML Override . . . 46
4.3 Benchmarking Tests . . . 47
4.4 Batch Engine . . . 48

5 Validation **51**
5.1 Benchmarking Results . . . 51

5.2 Results Analysis 56

6 Conclusion **59**
6.1 Future Works 60

61

Bibliography **61**

List of Figures

1.1 Simulation Of Air Moving Through Lungs 2
1.2 Simulation Of 2017 Turin Stampede 5
1.3 Parallel Computers Memory Models 6
1.4 Java Virtual Threads Conceptual Scheme 13
1.5 JVM Architecture 15
1.6 Alchemist Reactions Model 18
1.7 Alchemist Metamodel 19

2.1 Alchemist Launcher Priority System 27

3.1 Configuration Overriding Design Diagram 35
3.2 ACLS Design Diagram 37
3.3 Alchemist Launcher Architecture Design 38
3.4 Alchemist Simulation Design 40
3.5 Alchemist Parallel Simulation Design 41
3.6 Alchemist Engine Configuration Design 42

5.1 Throughput Scores With Sequential Engine 56
5.2 Throughput Scores With Parallel Engine 4 Threads 56
5.3 Throughput Scores With Parallel Engine 8 Threads 57

List of Listings

listings/JavaThreads.java 10
listings/JavaExecutors.java 11
listings/JavaStreams.java 11
listings/JavaVirtualThreads.java 12
listings/KotlinCoRoutines.kt 13
listings/alchemist–old–launch.txt 25
listings/ACLS.yml 36
listings/ACLS.java 36
listings/Overrides.kt 47
listings/benchmark.kt 48
listings/BatchEngine.kt 49

Chapter 1

Introduction

1.1 Computer Simulation

Computer simulations are defined as the use of a mathematical/logical model as an experimental vehicle to answer questions about a referent system[PN94]. In simple terms, computer simulation is a step-by-step process in which a real-world system is modeled by an approximation defined as a mathematical model in a computer system consisting of states and variables representing various dynamic aspects of the simulation. The simulation computes and updates the system's state at a given time (t) and then advances the time to t+1 and so on. Once the simulation is complete, the sequence of variables is saved as datasets, which can then be translated into a visualization. Alternatively, a visualization could be performed live in some kind of graphical interface. Computer simulations are used in many real-world studies and applications such as pandemic response, cancer treatment, crowd dynamic movements, city traffic, financial markets, and many more. Over time, computer simulation has demonstrated benefits for: [LTF+18]

1. Visualizing complex interactions in dynamic systems
2. Providing results much faster than would be possible in real time
3. Allowing "what if" analysis when changes to an actual system are difficult to implement, costly, or impractical.

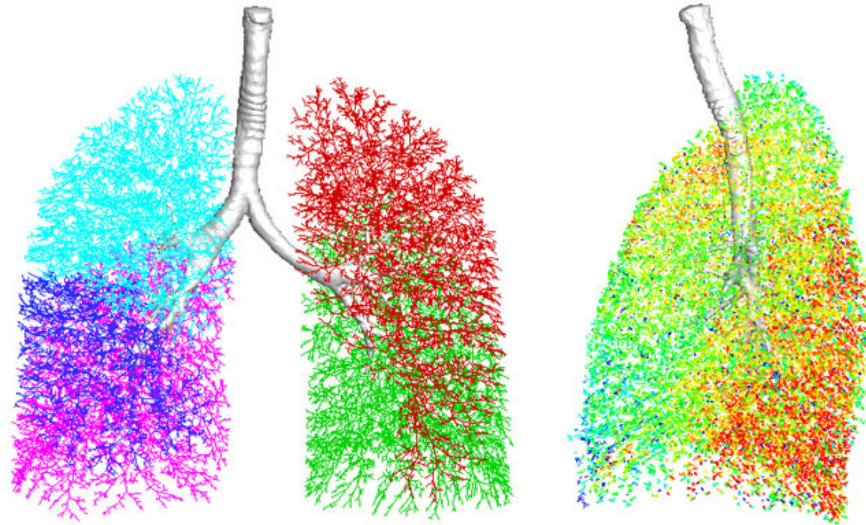

Figure 1.1: Simulation Of Air Moving Through Lungs

With the computation power of modern supercomputers, simulation is more advanced than ever and is evolving at a rapid pace, and with the computational resources at their disposal, simulations can achieve notable results and run an ever-increasing number of nuanced scenarios of very complex systems. However, due to the complex nature of the simulations, and the possible amount of variables processed to handle in case of larger simulations, even small-scale optimizations can lead to great gains in terms of speed of computation, which leads to being able to run more simulations in less times, hence achieving more results and more insights with less resources.

A common way to speedup the simulation computations is to parallelize the core algorithm. Modern processors have multiple available cores that can be used to compute task results in parallel. By re-thinking the simulation algorithm, we can use all of the available cores and resources to compute simulation in parallel. It is however notably complex to parallelize simulations as the algorithm needs to account for conflicts that may arise due to simulating events that may have mutual causality effects.

1.2 Discrete Event Simulation

Computer simulations are often used to study complex systems of dynamic objects and their interactions. Simulations are especially useful in enabling observers to measure and predict how the functioning of an entire system may be affected by altering individual components within that system. Typically a simulation is represented by a mathematical model in which parameters may be tuned to produce different observable outcomes.

Discrete Event Simulation (DES) is a form of computer-based modeling methodology characterized by the ability to simulate dynamic behaviors of complex systems and interactions between individuals, populations, and their environments. Although DES simulation models are typically a simplification of reality, they have proven useful as a tool to conduct "what-if" analyses with operational scenarios and rules tailored to the desired use cases. DES has proven particularly useful in healthcare applications and biochemistry domains.

DES models real-world systems as a set of logically separate processes that discretely progress through time at discrete intervals. The core concepts of DES are

- entities - Entities are domain objects that possess a set of attributes. An example of an entity is a Internet Of Things (IoT) device.
- attributes - Attributes are features that are possessed by an entity and carry some kind of defined information. An example of an attribute could be a serial number of an IoT device.
- events - Events are loosely defined as functions over an entity that are performed given certain conditions. Events may affect the state of an event and create or remove other scheduled events. An example of an event is a message received by an IoT device.
- resources and queues - Resources represent services used by an entity. In particular, a resource may have consumption constraints on the number of

entities that may use it. If a resource is unavailable to an entity that required it, the entity gets put into a queue. Typically queues follow either first-in-first-out (FIFO) or last-in-first-out (LIFO) policies. An example of a resource could be a message consumer and a queue is a message inbox inside an IoT device.

- time - Time is a fundamental component of DES. A simulation runs an explicit simulation clock initiated at the start of the simulation model run that keeps track of time and advances at discrete intervals.

DES can model many real-life situations that require complex systems of interacting actors that model real-world events and situations. An example of a DES use case is a simulation of the 2017 Turin Stampede, where an event of panic occurred in a packed crowd resulting in over 1500 injured people and 2 casualties. This simulation has been achieved with Alchemist chemical-oriented DES simulator [1]. This simulation modeled human psychology in panicky situations and lead to insights into the consequent physical interactions caused by the behavior of cognitive agents[2].

1.3 Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) is simply a DES designed to be executed on a parallel computer. Projects in discrete event simulation possess several characteristics that distinguish them from most software-intensive efforts[PN94]:

- Time - the which clearly establishes an ordering of behavioral events in the referent system and delimits the potential for parallel execution of the proposed model. This characteristic is that which has placed DES in the forefront of challenge for those interested in parallel computation.

[1] https://alchemistsimulator.github.io/index.html
[2] https://alchemistsimulator.github.io/showcase/2022-turin/index.html

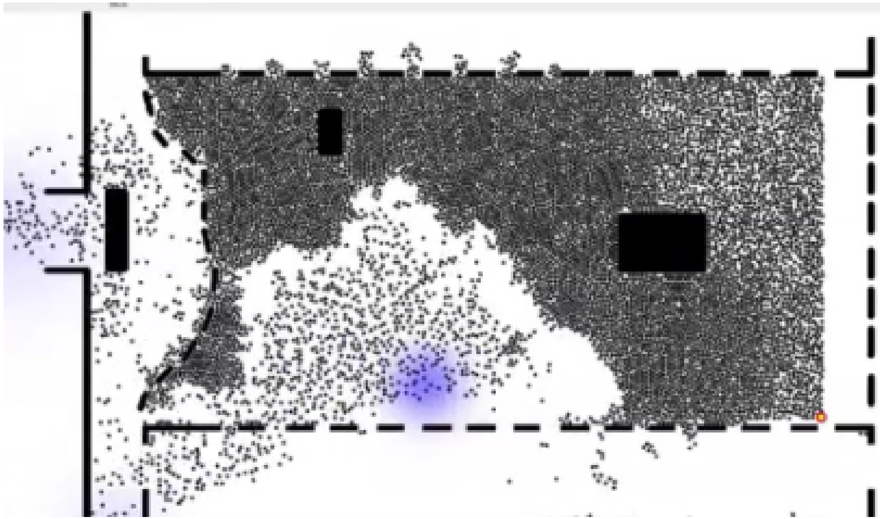

Figure 1.2: Simulation Of 2017 Turin Stampede

- Correctness - The existence of the referent system in most projects and the insistence that model behavior reflect system behavior within some prescribed tolerance levels forces a definitive statement of correctness that admits no renegotiation of requirements. A consequent relaxation of the tolerance levels oc- curs with a clear admission of deficiency.

- Computational Intensiveness - While model development costs are considerable, as is the human effort throughout the period of operation and use, the necessity for repetitive sample generation for statistical analysis and the testing of numerous alternatives forces concerns for execution efficiency that are seen in few software-intensive projects.

Sequential DES usually processes a large amount of data and require a substantial amount of time to execute. Parallelizing DES yields a tangible reduction of processing time which leads to more timely results and opportunities to run more simulations of more scenarios. Another benefit is more efficient resource utilization. Modern consumer-grade machines are often equipped with multi-core processors that can run tasks in parallel. Sequential simulations only utilize one of the available cores, hence only a fraction of the available resources. Beyond consumer-grade machines, clusters and cloud systems have an option of dynam-

ically allocating more resources allowing to scale the computation power to the desired level. For PDES, there are two basic classes of parallel computers:

- shared memory 1.3(a) - common memory is shared by processors which are accessed with synchronization mechanisms such as locks.
- distributed memory 1.3(b) - each processor has its memory, and communications between processors are exchanged by messages in some kind of physical or virtual network.

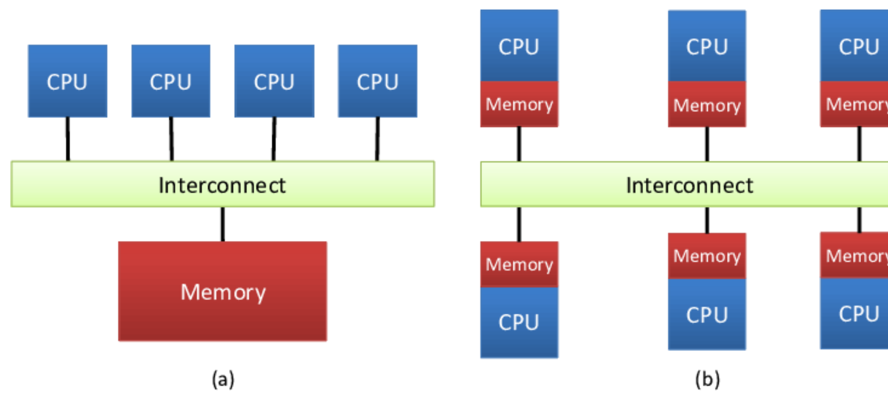

Figure 1.3: Parallel Computers Memory Models

Implementing PDES is very challenging as DES relies on the sequentiality of the events. At each time step a DES dequeues and executes the event with the smallest timestamp. The event's execution may affect other events and the environment and change the timestamp of the other events in the queue. Preserving sequentiality is not an issue in a regular sequential DES as the events are processed one at a time and changes to the environment are applied before processing the next event, whereas it is a major challenge in PDES as multiple events are processed in parallel. This leads to a possibility of *causality errors*, which in simple terms is an errors that occures when a future event is influencing the past event:

1. Let events E_1, E_2 with timestamps T_1, T_2 where $T_1 < T_2$

2. If E_1 changes a state variable used by E_2, E_1 must be executed first

This poses a challenging question: how can a simulation be sure that E_2 gets executed after E_1? This is a trivial problem to solve in a sequential DES, but most PDES strategies assume that processes do not have access to shared-state variables, so some kind of *sequencing constraints* mechanism must be applied. Sequencing constraints are by their nature highly complex and data-dependent as we cannot know how E_1 affects E_2 without simulation E_1 first and thus applying its effects. An event execution can affect other events through a complex network of causes and effects. No general solution exists for sequencing constraints, however, we can categorize the sequencing constraints strategies as:

- Conservative - No causality errors can occur. It requires a complex strategy to determine if an event can be processed, i.e. if all the events that can affect the candidate event have been processed.
- Optimistic - Allows causality errors to occur. A detection and recovery strategy can be applied to roll back the error state.

Optimistic PDES strategies are considered to be the best for a general-case solution, while conservative strategies are best in cases where a good look-ahead can be achieved. However, it should be noted that conservative approaches do not scale easily and are not robust to small changes in the applications. Ultimately the choice of the best strategy is highly problem-dependent.

1.4 Designing Parallel Systems

To design a parallel system, the problem must be analyzed to identify exploitable concurrency to choose the most appropriate parallel architecture. Typically a problem can be decomposed in terms of:

- Tasks - the problem can be naturally decomposed into a collection of independent tasks (divide-at-impera principle) e.g. producer/consumer pattern
- Data - The problem's data can be decomposed into independently processed units e.g. matrix multiplication

Another important point to consider are resources and dependencies of the decomposed tasks or data units. In particular *temporal* and *resource* dependencies must be accounted for. Taking into consideration decomposition and dependencies, a most fitting parallel architecture is chosen. Several conceptual classes for parallel architectures are proposed in literature [CG89]:

- Result parallelism - Typical for data decomposition strategies, the system is designed around the result of an independent unit of processing computed in parallel
- Specialist parallelism - The system is designed around a logical network of independent agents (logical processes) specialized in a specific task within the context of a global computation pipeline. The agents typically consume the results of the other agents in a pipeline to construct the final aggregated result.
- Agenda parallelism - The system is designed around an agenda of tasks where generalist agents are dynamically assigned from the available pool to the items in the agenda according to the current pending step in the process.

The classes of the architecture are not mutually exclusive and the optimal design often lies in between. Reasoning in terms of the classes of architecture helps in viewing the problem from different perspectives to find the best-fitting solution.

Beyond the conceptual classes, several specific architectures are typically considered first when designing a parallel implementation:

- Masters-Workers - The architecture is composed of coordinated agents.
  - master (manager) - The agent used to assign units of work (tasks) to A dynamic set of worker agents and aggregate their results.
  - worker - Agents used to compute units of work and produce partial or intermediate results. In some implementations, a worker may become a master for other workers in a tree-like fashion.

  A typical example of this pattern is *fork-join* implementations.

- Filter-Pipeline - The architecture is composed of a pipeline of chained agents applying a partial transformation or an action to the data in the pipeline's step. Agents can be classed as
  - generators -Aagents initiating the pipeline by generating data.
  - filters - Agents applying a transformation step to a unit of data to be passed on to the next agent in the chain.
  - sinks - Agents collecting the results and terminating the pipeline.

  This pattern is used in Java parallel steams or Appache Spark data pipelines.
- Shared-Space - The architecture is composed of a set of independent agents interacting indirectly through some kind of shared space where information items may be put, queried, or removed. An example of shared spaces are tuple spaces and actor systems.
- Announcer-Listener - The architecture better known as *event-driven* composed of
  - announcers - Agents posting an event to a topic channel.
  - listeners - Agents registered to consume an event upon posting on a topic channel.

  This architecture is particularly popularized with modern asynchronous distributed systems such as microservices and WebSockets.

1.5 Parallelism in JVM

Java and the JVM platform is a popular choice when designing and implementing parallel systems. When designing JVM parallel systems, an important consideration to account for is the disparity between application's demand and available resources decreases the application performance. On one side, if the application demand is represented by its degree of parallelism and required resources exceed the available processor capability, the system will exhibit low scalability.

On the other side, low application demand means that the system might be underutilized[CCH11]. The trade-off is between scalability and efficient resources utilization.

The JVM platform presents multiple options of various paradigms and approaches for designing parallel systems. Discussing these options is a broad topic that is best viewed in the historical context of the platform's evolution. For the scope of this thesis, we present a brief overview of Java's concurrency options with a comparison to the similar options available in Kotlin.

Platform Threads

One of the first and simplest options for concurrency in Java is the platform threads. Java exposes an Application Programming Interface (API) to create a thin wrapper around an Operating System (OS) thread. A platform thread runs Java code on its underlying OS thread, and the platform thread captures its OS thread for the platform thread's entire lifetime. Consequently, the number of available platform threads is limited to the number of OS threads[3]. The execution is synchronized and governed by low-level mechanisms such as locks, mutex, and semaphores. While powerful, this approach is usually deemed too low-level by today's standards due to the complex and error-prone nature of such a low-level concurrency framework.

```
1 Runnable r1 = () -> { System.out.println("Thread is created and running
      successfully..."); };
2 Thread t1 = new Thread(r1, "My Thread");
3 t1.start();
```

[3] https://kotlinlang.org/

Executors

As of Java Development Kit (JDK) 5 Java offers a new higher-level concurrency abstraction, the Executors API framework. Executors framework provides a way to create (thread) pools of workers to be used in a task-based concurrency architecture: a unit of work is submitted to a thread pool and the results may be retrieved or flow synchronized via the "Future" asynchronous concurrency abstractions. In addition to basic task-based submission, an implementation of the Fork-Join architecture was also introduced. Fork-Join pools allow for tasks to generate and submit child tasks, hence the "fork" part, and await their results, hence the "join" part.

```java
1  ExecutorService executorService = Executors.newFixedThreadPool(10);
2
3  Callable<String> callableTask = () -> {
4      TimeUnit.MILLISECONDS.sleep(300);
5      return "Task's execution";
6  };
7
8  List<Callable<String>> callableTasks = new ArrayList<>();
9  callableTasks.add(callableTask);
10 callableTasks.add(callableTask);
11 callableTasks.add(callableTask);
12
13 List<Future<String>> futures = executorService.invokeAll(callableTasks);
```

Parallel Streams

With the introduction of lambda streams in JDK 8, a new concurrency model was offered: parallel streams. Parallel streams are an effort to naturally and easily extend the regular sequential streams with the parallel pipeline concurrency model.

```java
1  List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
2  int sum = listOfNumbers.parallelStream().reduce(0, Integer::sum) + 5; // sum is
       equal to 5
```

Virtual Threads

With the upcoming release of JDK, 21 comes a major change in Java's concurrency model: virtual threads. These new features are a fruit of the "project Loom" OpenJDK initiative and are factually a major shift in Java's concurrency landscape. With a few simple changes platform threads can now become "virtual" threads. A virtual thread isn't tied to a specific OS thread. A virtual thread still runs code on an OS thread. However, when code running in a virtual thread calls a blocking I/O operation, the Java runtime suspends the virtual thread until it can be resumed. The OS thread associated with the suspended virtual thread is now free to perform operations for other virtual threads[4].

```
1  ExecutorService executorService = Executors.newVirtualThreadPerTaskExecutor(); //
       very simple to refactor existing code
2
3  Callable<String> callableTask = () -> {
4      TimeUnit.MILLISECONDS.sleep(300);
5      return "Task's execution";
6  };
7
8  List<Callable<String>> callableTasks = new ArrayList<>();
9  callableTasks.add(callableTask);
10 callableTasks.add(callableTask);
11 callableTasks.add(callableTask);
12
13 List<Future<String>> futures = executorService.invokeAll(callableTasks);
```

Kotlin Co-routines

The introduction of virtual (also called "light-weight") threads is not a completely novel concept on the JVM platform. The Kotlin programming language has implemented a similar concurrency idea over the JDK 8 threads called coroutines. A

[4]https://docs.oracle.com/en/java/javase/20/core/

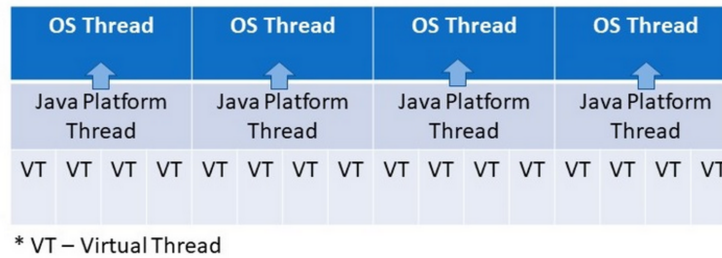

Figure 1.4: Java Virtual Threads Conceptual Scheme

```
val job = launch {
    delay(1000L)
    println("World!")
}
println("Hello")
job.join()
println("Done")
```

coroutine is an instance of suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one[5].

1.6 Benchmarking Parallel Architectures on JVM

Benchmarks are not a straightforward matter, especially because they do not always represent real-world usage patterns. It is often quite easy to produce the outcome you want, so skepticism is a good thing when looking at benchmark results[6]. This is especially true for benchmarking code on the JVM platform, given that the JVM is an adaptive virtual machine. When benchmarking the JVM, the

[5] https://kotlinlang.org/docs/home.html
[6] https://www.oracle.com/technical-resources/articles/java/

peculiarities of the platform should be taken into consideration, such as garbage collection cycles, Just In Time (JIT) compilers, and unwanted optimizations due to the narrowness of micro benchmarking only specific parts and control flow paths of the code.

Furthermore, a distinction between micro and macro benchmarking should be made.

- Micro-benchmarking - benchmarking a specific, usually performance critical, piece of the entire code base.
- Macro-benchmarking - benchmarking the entire program's execution, or at least a meaningful part of it.

Fortunately, these issues are well-known and multiple options for meaningful benchmarking are available. The de-facto standard in the Java benchmarking world is the Java Microbenchmark Harness (JMH) framework.

JMH

JMH is a tool developed under the OpenJDK umbrella that allows users to specify benchmarks through Java annotations, using a syntax that is similar to the well-known JUnit framework[CBLA19]. JMH provides a very solid foundation for writing and running benchmarks whose results are not erroneous due to unwanted virtual machine optimizations. JMH itself does not prevent the pitfalls that we exposed earlier, but it greatly helps in mitigating them. JMH is popular for writing microbenchmarks, that is, benchmarks that stress a very specific piece of code. However JMH is a general-purpose benchmarking harness, so it is also useful for concurrent and macro benchmarks[7].

The key concepts in JMH are:

[7] https://www.oracle.com/technical-resources/articles/java/

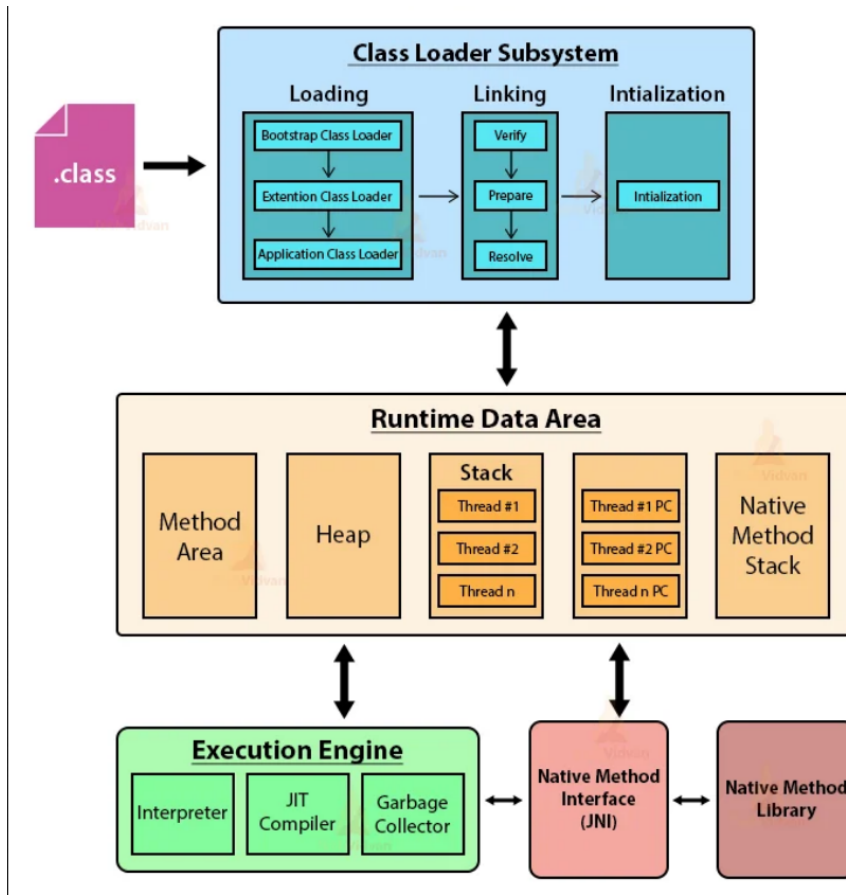

Figure 1.5: JVM Architecture

- Benchmark modes - determines what kind of metrics should be gathered from the benchmark runs
- Warmup phase - Before executing the benchmark, JMH performs a *warmup* phase by running the benchmark multiple times to ensure that JVM is fully optimized and resembles an ideal setup for an application
- Forking - Forking helps simulate the number of threads that should be used to measure the benchmark's performance

Although there are multiple benchmark modes in JMH, they essentially measure the performance in terms of:

- Throughput Mode - measures the number of method invocations that can

be completed within a single unit of time.

- Time-based Modes - test out the time it takes for the single invocation of the method. In particular, the following modes are available:
  - Average Time - measures the average time it takes for the single execution of the benchmark.
  - Sample Time - measures the total time it takes for the benchmark method to execute, including minimum and maximum execution time for a single invocation.
  - Single Shot Time - runs a single iteration of the compared methods without warming up the JVM.

The basic configuration is usually sufficient for macro-benchmarks, however when running micro-benchmarks, some further considerations such be taken into account such as constant folding, dead code elimination, and other configurable hints to avoid undesired optimizations by the JIT compiler.

Lastly, the environment running the benchmark needs to be taken into consideration as the different hardware will produce different results. It is important to benchmark on the machine as close as possible to the production deployment node, or in the case of a consumer application, on an array of various consumer-grade machines.

1.7 Chemical-Oriented Simulation

Chemical-oriented simulation represents an innovative and novel approach to simulation by combining both principles of (bio)chemistry and computer science. Traditionally, chemical-oriented simulation deals with the virtualization and simulation of chemical systems by modeling molecules, concentrations, energy, and reactions where computation can be seen as chemical reactions between data represented as molecules floating in a chemical solution[BFR06]. The intricate dynamics of chemical systems, characterized by random events and fluctuations at the molecular level, pose significant challenges to traditional deterministic modeling approaches.

Stochastic simulation algorithms exploit the principles of probability theory to capture the inherent randomness and variability of chemical systems. While deterministic models assume perfect knowledge of the precise initial conditions and interactions, stochastic simulation takes into account the inherent uncertainties and fluctuations that occur due to molecular-level interactions, environmental factors, and the inherent variability of biological systems. As a result, these algorithms allow researchers to simulate and analyze the behavior of chemical systems with greater fidelity, capturing the full spectrum of possible outcomes and uncovering emergent properties that may not be evident in deterministic models. However the principles and the ideas behind the chemical-oriented simulation are not exclusive to the (bio)chemical domain. One such example is the Alchemist [8] simulator, which is a highly extensible state-of-the-art generalist DES simulator based on chemical oriented simulation models.

1.8 Alchemist Simulator

The properties of complex and emergent computational systems characterized by situatedness, adaptivity, and self-organization are not confined solely to the traditional conception of (bio)chemistry. The Alchemist simulation framework [9] extends the basic computational model of chemical reactions to general use cases while maintaining high performance and extensibility for complex computational systems [PMV13], such as pervasive computing systems. in other words, Alchemist allows to model systems of possibly mobile, interconnected, and communicating agents that operate and interact with the system according to a set of chemical-like laws.

Alchemist Metamodel

The Alchemist simulator [10] bridges the gap between traditional chemical computing and DES by extending the basic computational model of chemical reactions to complex computational systems, benefiting from the performance of the former

[8] https://alchemistsimulator.github.io/index.html
[9] https://alchemistsimulator.github.io/index.html
[10] https://alchemistsimulator.github.io/index.html

and the emerging complexity of the latter. Alchemist's algorithm is based on an optimized version of Gillespie's SSA called Next Reaction[Gil77], suitably extended with the ability to handle a dynamic environment that allows for the addition and removal of reactions, data items, and connections. The meta-model and simulation framework are designed to be generic, and as such can have a wide range of applications such as pervasive computing, social interactions, and topological interactions.

Alchemist achieves three key properties of complex systems: Situatedness, Adaptivity, and Self-organization by modeling a mapping between computational abstractions onto chemical abstractions in a cohesive meta-model. An agent-oriented perspective is adopted to describe the meta-model by considering an agent as an autonomous entity that performs actions on the system based on stimuli received from the environment and encapsulates the strategy for choosing among the set of available actions. Consequently, the agent's state can be modeled as a set of "molecules" and its internal behavior through a set of chemical-type reactions that may or may not be enabled based on the perception of the external environment. Agents perceive the environment through the molecules that come from the environment and enter its boundaries. Multiple agents, which constitute a society, can communicate by exchanging molecules with neighboring agents. Finally, the environment is responsible for connecting agents and defining their neighborhood.

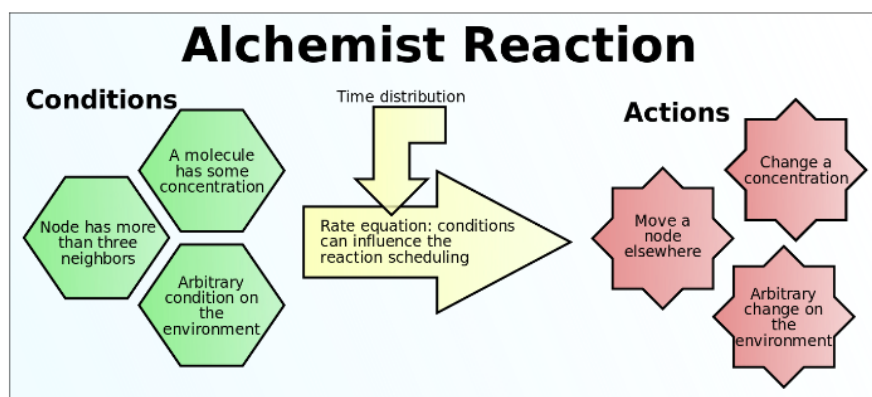

Figure 1.6: Alchemist Reactions Model

In summary, The key concepts of Alchemist are:

- Molecules - data template described by a concentration value and a set of properties.
- Reactions - with a more extensive definition than traditional chemistry. A reaction is modeled as a set of conditions (Boolean functions) on the state of the system, which trigger the execution of a set of actions.
- Neighborhood - The concept of the neighborhood must be extended beyond the physical concept of agents within a radius and be able to represent other types of relationships such as, for example, human relationships.
- Propensity function - a function of reaction speed, conditions, and the state of the environment.

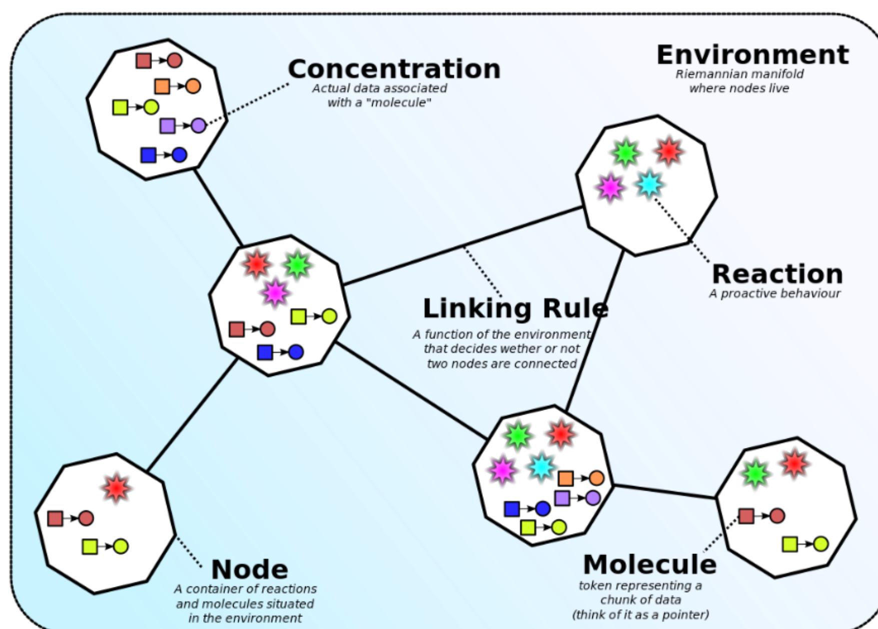

Figure 1.7: Alchemist Metamodel

Incarnations

Alchemist is designed for flexibility and extensibility. These goals are achieved by a very loose interpretation of molecule and concentration concepts. While these terms have a very precise definition in traditional chemistry, in Alchemist they are simply composed of:

- generic identifier
- piece of data

An incarnation includes a type definition of concentration, and possibly a set of specific conditions, actions, and (rarely) environments and reactions that operate on such types. In practical terms, an incarnation is a concrete instance of the Alchemist meta-model. As of the time of writing of this thesis, the following incarnations are available:

- Sapere Incarnation [ZOA+15] [PMV11] - The first available stable incarnation. The core concept of this incarnation is the usage of Live Semantic Annotation (LSA).
- Protelis Incarnation [11]- The goal of the Protelis language is to make it easier to build a resilient and well-behaved networked system out of an assortment of different potential mobile devices. Protelis is designed for the paradigm of "aggregate programming", a way of thinking about and decomposing problems that can be solved with a network of distributed sensors and computers.
- Biochemistry Incarnation [12] - models biochemical reactions of biological cells that share a common environment
- Scafi incarnation [13] - ScaFi (Scala Fields) is a Scala-based library and framework for Aggregate Programming. It implements a variant of the Higher-Order Field Calculus (HOFC) operational semantics, which is made available as a usable domain-specific language (DSL), and provides a platform

[11] https://protelis.github.io/
[12] https://alchemistsimulator.github.io/explanation/biochemistry/
[13] https://scafi.github.io/

and API for simulating and executing Aggregate Computing systems and applications.

1.9 Project Goals and Motivations

Alchemist simulations are powerful and expressive, however, the complex nature of the engine has a notable cost in terms of speed. The goal of this thesis project is to attempt to improve the current speed of the simulation computation by re-designing the DES alchemist core into a PDES. This is a daunting problem to solve, both given the inherent complexity of implementing a PDES, and the complexity of the simulator. To achieve the desired results, we may allow for the relaxation of some of the simulation constraints. In particular, for the benefit of simulation speed-up, we may accept some degree of non-determinism caused by causality errors and control flow based on the non-deterministic thread scheduling. These relaxed constraints should not excessively degrade the quality of the simulation, so some kind of mitigation policy or strategy should be designed. The new PDES mode should also not replace the original DES core but be presented as an alternative a user can optionally opt-in. This requires a substantial redesign of the current simulator's launch configuration as the current Command Line Interface (CLI) implementation does not have the flexibility and extensibility required to configure a complex simulation launch. Furthermore, an option to arbitrarily override some of the simulation parameters is needed to allow for easier automated launches via configuration templates and scripts. Finally, the performance of the PDES should be properly and thoroughly benchmarked to assert that the benefits over the sequential execution outweigh the costs. Given the available options for modern concurrent architectures on the JVM, the PDES should be implemented in each, and performances compared to find the best fitting concurrency implementation for this case.

Chapter 2

Analysis

After a preliminary study, the following key deliverable parts of the project have been identified:

- Launch Configuration Rework
- Parallel Engine
- Benchmark Harness

Alchemist multi-project structure

Alchemist utilizes the Gradle build automation system. Gradle is a task-based build automation tool for multi-language software development. It controls the development process in the tasks of compilation and packaging to testing, deployment, and publishing and is highly extensible. One of the project structure options offered by Gradle is the multi-project build. A multi-project build in Gradle consists of one root project, and one or more subprojects. This structure helps to create natural boundaries of the different independent modules of the project and loosen the coupling.

Alchemist currently consists of the following modules:

- alchemist-api
- alchemist-engine

- alchemist-euclidean-geometry
- alchemist-full
- alchemist-fxui
- alchemist-grid
- alchemist-implementationbase
- alchemist-incarnation-biochemistry
- alchemist-incarnation-protelis
- alchemist-incarnation-sapere
- alchemist-incarnation-scafi
- alchemist-loading
- alchemist-maintenance-tooling
- alchemist-maps
- alchemist-multivesta-adapter
- alchemist-physics
- alchemist-sapere-mathexp
- alchemist-smartcam
- alchemist-swingui
- alchemist-test
- alchemist-ui-tooling
- alchemist-web-renderer

The scope of the project will mainly affect the api, engine, loading and implementationbase modules.

2.1 Launch Configuration Rework

Alchemist is historically launched via CLI and configured with a list of command line parameters. The parameters are parsed and based on the given parameters a simulation launcher is chosen from the pool of the available launchers. The launcher is then responsible for loading and governing the execution of a simulation. The actual configuration of the simulation (i.e. the layout of the molecules, reactions, etc...) is given in a separate configuration file in Yet Another Markup Language (YAML) format.

For example, a launch of a simulation defined in the file simulation.yml that runs for fifty steps in a headless mode is configured with the following CLI parameters:

```
1 -y simulation.yml -hl -t 50
```

Where

- -y simulation.yml is the path to the simulation configuration file
- -hl hints that the launcher to be used is "headless"
- -t 50 tells that the simulation is to end after fifty time units

Therefore, the parameters can be grouped into three categories:

- simulation configuration path - a unique and mandatory parameter that contains the path to the simulation configuration file.
- launcher hints - typically a flag used to hint at what loader is to be used.
- launcher parameters - typically a path to the additional configuration file, or a flat value that a certain simulation launcher can recognize and use.

Alchemist design is governed by the principles of extensibility and modularity. While this CLI design was sufficient in the earlier iterations of the simulator, it

does not scale well, as every time a new simulation launcher is added, new configuration parameters need to be hard coded and added to the growing list of possible options. In particular, in the context of this thesis project, we would require a complex launch configuration to decide what mode an engine should run in (parallel or regular) and provide additional configuration values such as the level of parallelism, contextually for the type of engine chosen.

For the needs of this project and for the benefit of the project as a whole, a CLI redesign is proposed. There are several considerations to be made about the historical CLI implementation.

1. A lot of the complexity is brought from the launcher choice system, the flags, and the hints interact with a *Service Loader* design pattern-based system that dynamically loads the possible implementation of the launcher and based on priorities governed by the flags, the best matching candidate is chosen.

2. The simulation file contains the actual simulation configuration and some of the newest additions to the file configuration have supplanted the need for the CLI parameters, for example, the termination conditions.

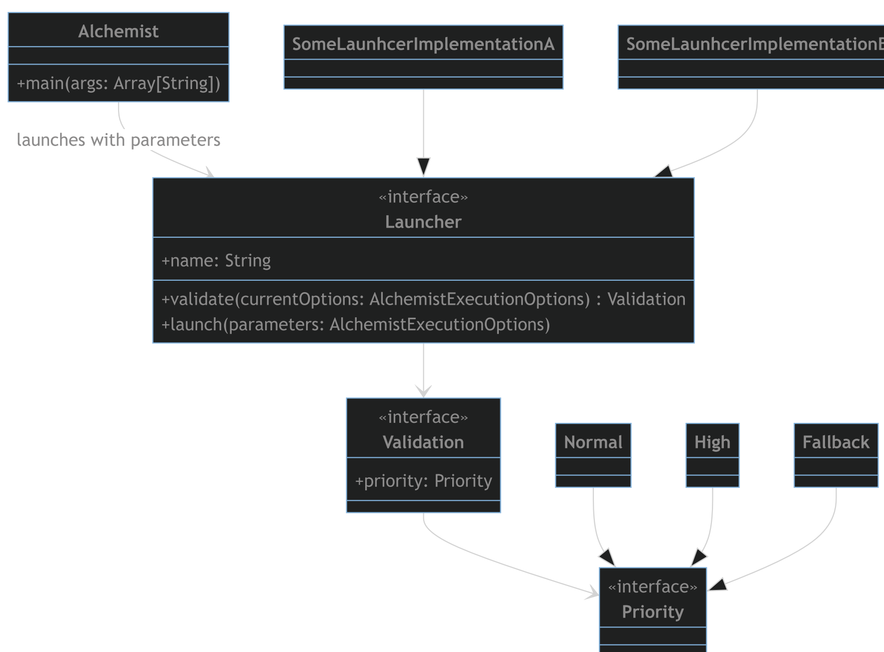

Figure 2.1: Alchemist Launcher Priority System

The new CLI design should address these two points and streamline the configuration without sacrificing flexibility and modularity.

2.2 Parallel Engine

The simulator implementation is based on two key data structures:

- Dynamic Indexed Priority Queue: a binary tree of reactions proposed in Gibson and Bruck (2000). Each node stores a reaction whose putative occurrence time is less than that of each of its children so that the next reaction to be executed is always in the root of the tree and can be accessed in constant time. In Alchemist it is extended over the original definition to allow insertion and removal of dynamic nodes.

- Dynamic Dependency Graph: an oriented graph in which nodes are triggered reactions and arcs connect a reaction r to all those that depend on it, i.e., those whose activation time must be updated with the execution of r

The algorithm can be summarized with the following pseudo-code:

```
1: cur_time = 0
2: cur_step = 0
3: for each node n in environment do
4:   for each reaction nr in n do
5:     generate a new putative time for nr
6:     insert nr in DIPQ
7:     generate dependencies for nr
8: while cur_time < max_time and cur_step < max_step do
9:   r = the next reaction to execute
10:  if r's conditions are verified then
11:    execute all the actions of r
12:    for each reaction rd which depends on r do
13:      update the putative execution time
14:  generate a new putative time for r
```

To parallelize this sequential DES algorithm, several considerations must be made. Firstly, we should decide between data-oriented and task-oriented approaches to parallelization. The data-oriented approach makes more sense as we can divide the queue of events into batches, i.e. sequences of events to be processed in parallel. The second consideration to be made is how to handle causality errors, in other words, whether we should adopt optimistic or pessimistic PDES architecture. While pessimistic would be preferable to keep the simulation reproducible and accurate, the cost of development of such an algorithm is very large due to the inherent complexity of pessimistic PDES, and the pessimistic approaches tend to break and require additional work when changing existing algorithm, which would compromise Alchemist's guiding principles of modularity and extensibility. When adopting an optimistic PDES architecture we allow for causality errors to happen. A rollback mechanism would be optimal to maintain simulation determinism, however, this would require events to be able to be "undone" and simulation state snapshots to be maintained which is very complex. An alternative approach would be to accept causality errors and limit their impact on the downstream events. Considering batched processing of the events, we can limit causality errors

to the bounds of the batch. A more intuitive way to view this solution is to imagine that the events in the batch would have happened within a small enough time bound to have their order of processing not influence the rest of the simulation in a significant way. This would require a strategy to build batches given on scheduled time proximity.

The proposed changes to support batch processing of the events are:

```
1: cur_time = 0
2: cur_step = 0
3: for each node n in environment do
4:   for each reaction nr in n do
5:     generate a new putative time for nr
6:     insert nr in DIPQ
7:     generate dependencies for nr
8: while cur_time < max_time and cur_step < max_step do
9:   r[] = the next reactions batch to execute
10:   results = for each r in r[] create and submit task
11:    if r's conditions are verified then
12:      execute all the actions of r
13:    for each reaction rd which depends on r do
14:      update the putative execution time
15:    generate a new putative time for r
16:  await results
```

2.3 Benchmark Harness

To understand the impact of parallelization on the simulator, a robust benchmark testbed is needed. Several benchmarking harnesses are available for the JVM platform, the de-facto standard is the JMH framework. There are multiple options on how to integrate benchmarking into the Alchemist project. Benchmarking could either be treated as a test source dependency for each module of the project, which makes sense in context of micro-benchmarking of specific parts of the codebase,

or it can be integrated as a stand-alone module in the Alchemist multi-module project. The latter approach is adopted as it makes more sense in the context of macro-benchmarking.

JMH

JMH is a Java harness for building, running, and analyzing nano/micro/milli/-macro benchmarks written in Java and other languages targeting the JVM[1].

There are multiple options for running the JMH engine:

- Command Line
- IDE plugin
- Build Systems

Since Alchemist is built using the Gradle build automation system, it would make more sense to utilize its jhm integration. This integration consists of a plugin that scans the source set for runnable benchmarks and runs them, outputting the aggregated results into a text file.

Typically JMH benchmarks are inserted into the source sets of the projects as if they were unit tests. This approach makes sense in the context of micro-benchmarking, where the output results of the critical parts can be parsed and automatically checked for performance regressions, but since for the scope of the project, we require macro benchmarks, a separate module that runs the simulator as if it were run by a user is preferable.

JHM offers multiple scoring based on either operations throughput or execution time:

- Throughput - Measures the number of operations per second, meaning the number of times per second your benchmark method could be executed.

[1]https://github.com/openjdk/jmh

- Average Time - Measures the average time it takes for the benchmark method to execute (a single execution).
- Sample Time - Measures how long time it takes for the benchmark method to execute, including max, min time, etc.
- Single Shot Time - Measures how long time a single benchmark method execution takes to run. This is good to test how it performs under a cold start (no JVM warm up). All Measures all of the above.

While time-based scoring could be interesting, since we are benchmarking the system as a whole and not the core algorithm, external dependencies and events may skew the time-based metrics. So the throughput is the better choice here.

Another important aspect to consider when setting up successful benchmarks are the forking parameters i.e. forks - how many times the benchmark will be executed and warmup - how many times a benchmark will dry run before results are collected. Multiple forked runs are important to achieve compilation profile separation and the natural execution variance due to external OS and hardware dependencies and events. Running benchmarks with at least a few forks will average out the variables and produce results that represent the average execution better. Warmups are also important due to the nature of the just-in-time compilator, dry runs would enable its dynamic optimizations and ensure that the tested run is representative of a real execution. For the macro benchmarks of the project, the jam default warmup parameter is suitable, while the fork parameter is set to 3, lower than the default.

testbed

A baseline with the DES alchemist implementation should be scored to base the performance score comparison on. Parallel algorithms are sensitive to the amount of resources available, so good testbeds should target different resource scenarios. Alchemist is typically run on consumer-grade hardware, so it would make sense to test against the common resource configurations. The following test cases have been selected:

- single thread, DES implementation - This tests the baseline to which the parallel solutions are compared.
- 4 threads, PDES implementation - 4 cores are typical of lower-grade consumer processors, this test should saturate such a processor
- 8 threads, PDES implementation - 8 cores are typical of higher grade consumer processors, this test should saturate such a processor

To build the testbed, each of these tests should be tested on the cartesian product of possible additional PDES configurations to understand how tuning variables may affect the execution.

Several different multithreading options are currently available on the JVM platform. To choose the most appropriate option, the testbed should be run on each, and scores compared. Furthermore, different JDK versions may affect the performance of the same implementation as the compiler produces a more efficient bytecode. The most insightful comparison is to compare classic Java threading options with the new virtual threads and Kotlin co-routines. Each of the testbeds is to be tested with the following implementations:

- JDK 11 - classic executors
- JDK 11 - kotlin co-routines
- JDK 19 - classic executors
- JDK 19 - virtual threads executors
- JDK 19 - kotlin co-routines

Chapter 3

Design

3.1 Launch Configuration

CLI Redesign

As previously stated, Alchemist is configured via the CLI options. The goal of the project is to move the launch configuration into the simulation configuration file. The CLI needs to be streamlined and redesigned. The following CLI interface is proposed:

- subcommand - If the application has a rich command line interface and executes different actions with different arguments, subcommands can be useful. A subcommand is the first positional argument in the cli and is used to contextualize the following parameters. This allows for the rich multi-modal launcher.
- option - Command line entity started with some prefix (-/–) and can have value as next entity in command line string.
- argument - Command line entity whose role is connected only with its position.

The only subcommand required for the scope of the project is *run* which contextualizes a simple simulation launch. Two options are proposed: –verbosity, to indicate the logging verbosity, as it is configured outside of the simulation model,

and –overrides, to provide manual CLI overrides as a valid YAML string to the simulation configuration. Lastly, the only argument is the simulation configuration file descriptor.

In practical terms, this is an example of the expected complete CLI

```
run the simulation.yaml --verbosity info --overrides "foo: bar"
```

Arbitrary Model Overrides

Applying overrides to the configuration file can be approached from many angles. For this project, a simple approach is adopted. The overrides are provided as a CLI option with a valid YAML as an argument. The override yaml is then merged with the original configuration file. The following acceptance criteria for the override system are proposed:

- Overrides should be arbitrary and allow for variable type changes.
- Overrides should be able to override complex objects and not just the values.
- Overrides should be able to create new variables.

The overriding module is best placed inside the simulation model loading logic, just before the simulation model map is interpreted, in this way, the overriding is transparent for the rest of the systems.

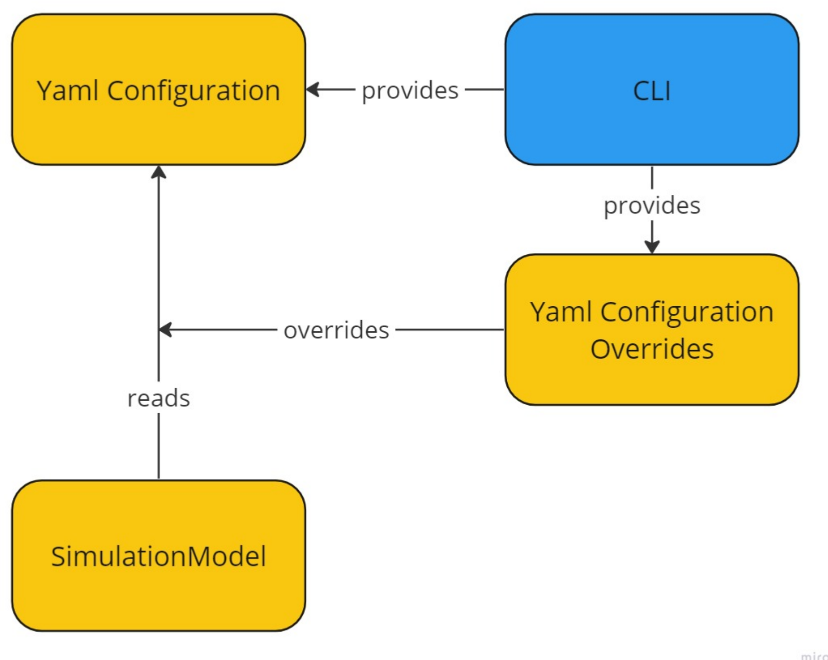

Figure 3.1: Configuration Overriding Design Diagram

Simulation Model

To parse and interpret the YAML configuration file, Alchemist utilizes a proprietary system called Arbitrary Class Loading System (ACLS)[1] that loads arbitrary types conforming to the expected interface (or Scala trait). The expected type depends on where the class is requested. The principle behind the system is to map YAML properties to arbitrary Java or Kotlin classes. To achieve that, Alchemist uses Kotlin's (and most importantly Java's) Reflection and Introspection [2] capabilities. The YAML structure is parsed where:

- *type* - The name of an instanceable class compatible with the expected interface. It can be either a qualified name or a simple name.
- *parameters* - The list of parameters the constructor of type should be passed. Alchemist automatically provides contextual information to the constructors.

[1] https://alchemistsimulator.github.io/reference/yaml/index.html
[2] https://kotlinlang.org/docs/reflection.html

For example, this YAML property:

```
1 deployments:
2       type: Point
3       parameters: [0, 0]
```

Is used to construct the following Java object:

```
1 public Point(final Environment<?, P> environment, final double x, final double y)
      {
2     ...
3 }
```

The ACLS system can be summarized with the following diagram:

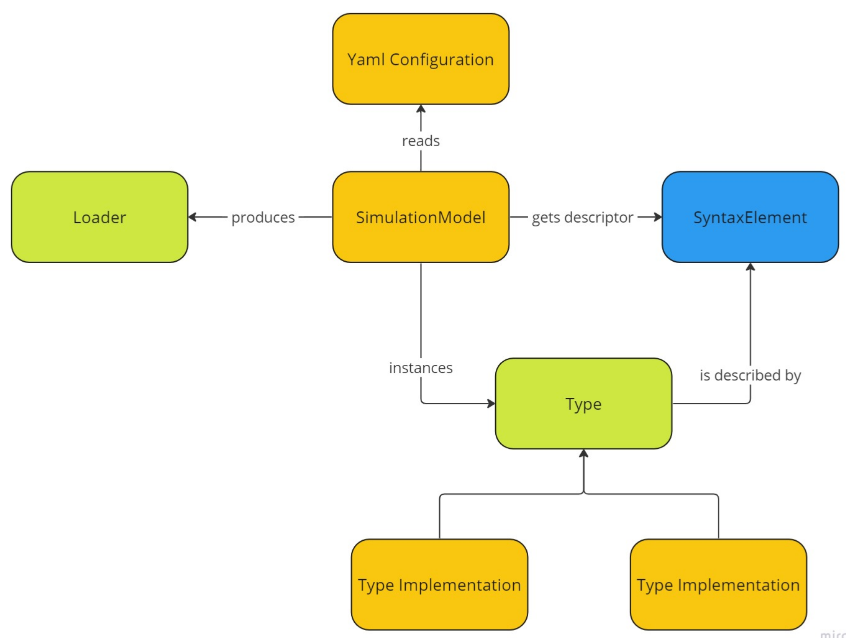

Figure 3.2: ACLS Design Diagram

Where

- *Yaml Configuration* - Loaded YAML configuration file. It is provided as a map of variables.
- *SimulationModel* - Converts an alchemist model defined as a Map into a loadable simulation environment and relative exports.
- *Loader* - Instanced loadable simulation environment.
- *SyntaxElement* - Describes the interpretable syntax elements in the configuration map, it also describes the interface for the constructable class.
- *Type* - Interface of the constructable class described by SyntaxElement.
- *Type Implementation* - Constructable class of the described type.

To bring configuration into the simulation model, the following actions must be taken:

- The Loader interface should be expanded with a method exposing the parsed simulation launch configuration.
- Syntax should be updated to include the simulation launcher configuration.

A natural choice is to directly instance the *Launcher* type from the simulation model. *Launcher* is an entity that can take responsibility for performing an Alchemist run, hence the configuration for the specific Launcher is simply its constructor parameters. The complete simulation launch architecture can be condensed with the following diagram:

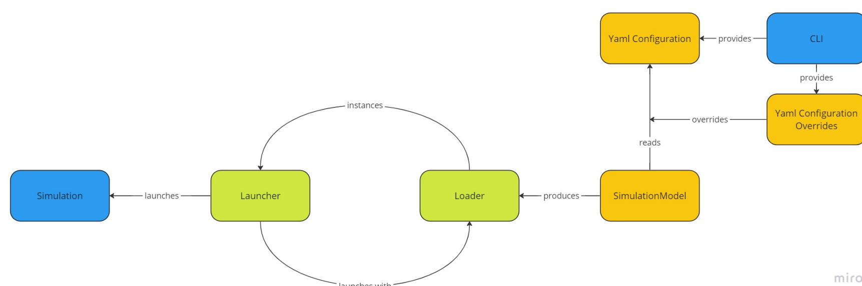

Figure 3.3: Alchemist Launcher Architecture Design

3.2 Parallel Engine

PDES Algorithm Design

As stated previously, the intent is to implement an optimistic PDES that allows for causality errors within the bounds of a timeframe. In other words, at each batch step of the simulation, a batch of the events is retrieved from the queue. The events in the batch are processed in parallel and the changes are applied dynamically and independently. The batch step awaits the results of all the batch events before advancing the simulation steps by the size of the batch and repeats the process until the termination conditions are met (time, step, empty event queue,

etc.....). We can assume that the simulation is synchronized after the end of each batch (time window) processing and that no event with the timestamp less than the maximum of the time window will be met.

An important thing to consider is the size of the processing batches. Since each event processing can be seen as an independent task, in a traditional parallel system the best way to utilize the available resources is to have the number of tasks equal to the available threads for processing. However, while this naive implementation may have optimal resource utilization, the resulting batch of a fixed size may have the events of a sparse time distribution. In other words, while the causality errors of the events that would have been scheduled at a closer time proximity may have a lesser impact on the simulation outcome, larger time gaps may reduce the quality of the simulation. Hence, an alternative method to build a batch is proposed, the events from the queue get added to the batch while the scheduled time difference is lesser than a certain tunable value. We name this method *epsilon* batch. In summary, the following batch construction methods are proposed:

- Fixed Size Batch - Given an integer n, a batch is the next n events in the queue
- Epsilon Batch - Given a value *epsilon*, a batch is the next m events in the queue where $t(e2) - t(e1) < epsilon$

The simulation may have output monitors attached to the execution, for example, GUI publishers. There is an important consideration to be made as to when and how to signal the results to the monitors. Two options are proposed:

- Aggregate - Only the latest simulation state is sent to the output monitors.
- Replay - The simulation states are stored and sent to the output monitors in their temporal order.

Batch Engine Design

The core of the Alchemist's DES algorithm is found in the *Simulation* interface. The actual sequential implementation is found in the *Engine* class. The engine is composed of the two core data structures:

- DependencyGraph - an interface for an implementation of the Dynamic Dependency Graph
- Scheduler - an interface for an implementation of the Dynamic Indexed Priority Queue

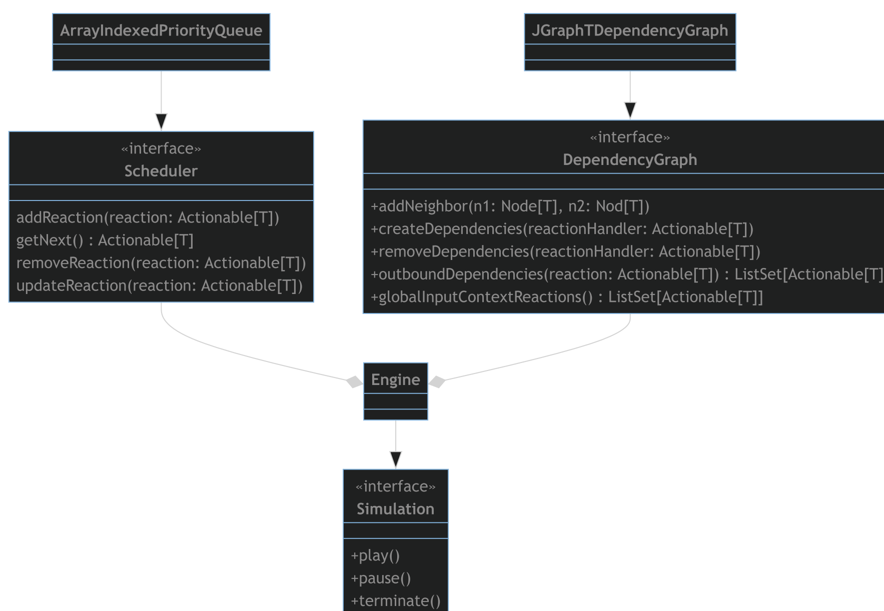

Figure 3.4: Alchemist Simulation Design

There is no need to re-implement the parallel engine from scratch as a lot of the ancillary logic is already present in the Engine class. The PDES part of the algorithm requires changes to the simulation step logic, where before a step would have processed a single event, in the batch engine a single step processes a batch of the events. For this purpose, the step is extracted as a new protected method inside the *Engine* class. The parallel engine is therefore implemented in the *BatchEngine* class that extends the base *Engine* class.

The scheduler should also be changed to be able to retrieve a batch of events

from the queue instead of a single event. The logic for batch construction is left up to the implementation of the interface. We propose an extension to the *Scheduler* interface: *BatchedScheduler*. As per the proposed algorithm specification, two concrete implementations are provided

- ArrayIndexedPriorityFixedBatchQueue - naive fixed size batch implementation

- ArrayIndexedPriorityEpsilonBatchQueue - epsilon dynamic batch implementation

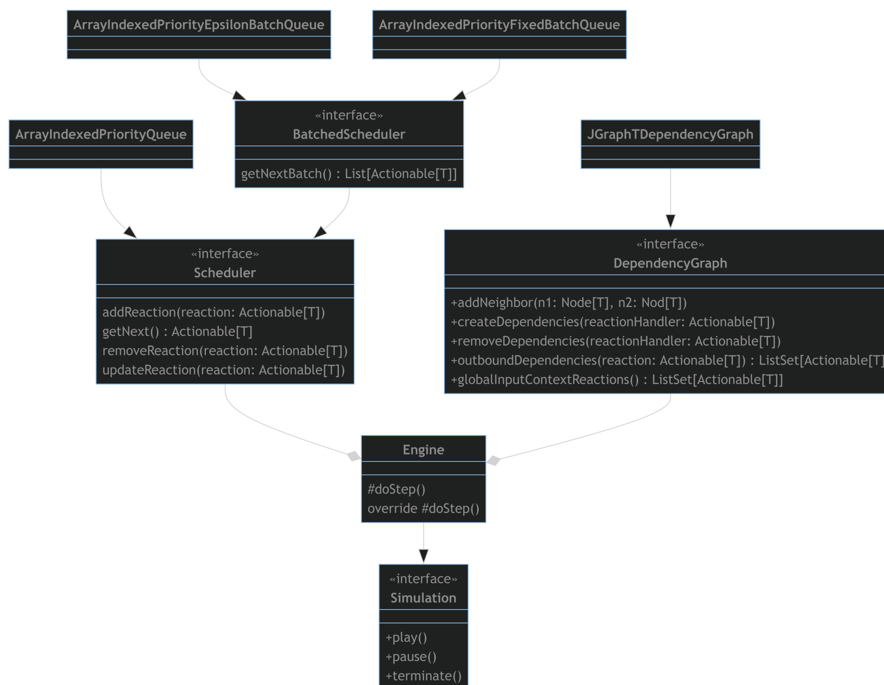

Figure 3.5: Alchemist Parallel Simulation Design

Batch Engine Configuration

As for the launcher configuration, the engine is optionally configured inside the simulation configuration file by using the ACLS [3]. The configuration is mapped to the *EngineConfiguration* type which is implemented with the following subtypes:

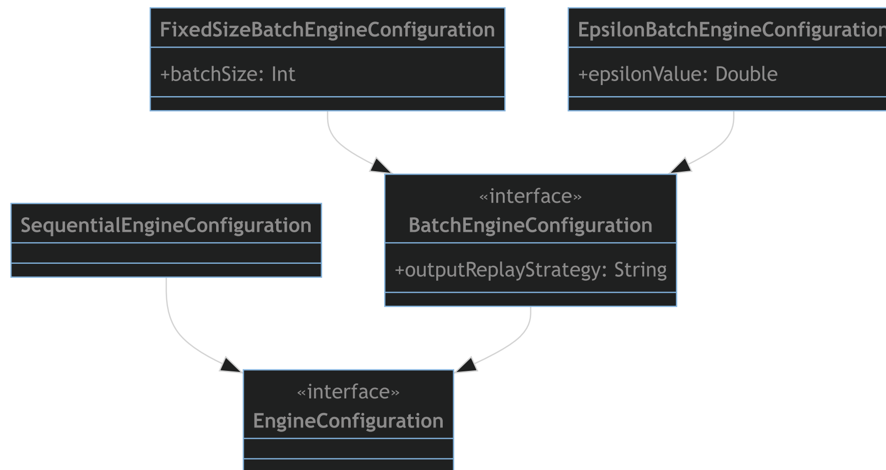

Figure 3.6: Alchemist Engine Configuration Design

The required engine and its dependencies such as Scheduler are constructed according to the configuration type. If no configuration is provided, it will default to the sequential one.

3.3 Benchmark Harness

The benchmarking harness is to be implemented in a stand alone repository that imports Alchemist as a dependency. This project relies on the Gradle JMH plugin and exposes the benchmarking as a gradle task. The benchmarks are exposed as methods in the *Benchmarks* class inside the JMH sourceset. The simulations run the entire Alchemist execution via the "main" method parametrized to the desired

[3] https://alchemistsimulator.github.io/reference/yaml/index.html

run configuration. The results are stored in a simple .txt file in the output folder.

The following benchmarks are proposed:

- sequential run
- parallel run 4 threads, fixed size batch
- parallel run 8 threads, fixed size batch
- parallel run 4 threads, epsilon batch
- parallel run 8 threads, epsilon batch

Each runs the same simulation with the same temporal termination conditions. The operations throughput for each benchmark is measured.

Chapter 4

Implementation

4.1 CLI Refactor

Original Alchemist implementation relied on the Apache-cli external Java dependency. The entire cli parsing system was rewritten using Kotlinx-cli, which is a Kotlin-native cli dependency. Kotlinx-cli is a pure Kotlin implementation of a generic command-line parser that aims to be Declarative, Platform-agnostic, and Hackable[1]. There are 2 base entities: option and argument.

- Option - command line entity started with some prefix (-/–) and can have value as next entity in command line string.
- Argument - command line entity whose role is connected only with its position.

Command line entities can be of several types:

- ArgType.Boolean
- ArgType.Int
- ArgType.String
- ArgType.Double

[1] https://github.com/Kotlin/kotlinx-cli

- ArgType.Choice
- Custom Type

If the application has a rich command line interface and executes different actions with different arguments, subcommands can be useful.

The implementation provides the following entities:

- simulationFile - Argument entity, file containing simulation configuration to be executed.
- verbosity - Option entity, Simulation logging verbosity level, a choice of one of the values from the available logging levels.
- overrides - Option entity, multiple, valid YAML files used to override simulation config. files are applied sequentially.

In addition to that, a subcommand "run" is provided. While no other possible commands are currently available, the choice to add a subcommand is an effort to future-proof the development of the next cli features in Alchemist.

Therefore, an example of the launch is

```
run the simulation.yaml --verbosity info --overrides "foo: bar"
```

4.2 Arbitrary YAML Override

Alchemist treats the loaded YAML configuration files as maps (i.e. dictionary data structure) of other maps or variables. To arbitrarily override a piece of the configuration, a simple map-merging algorithm is sufficient. The designed algorithm is a simple recursive map traversal:

```
fun mergeInto(key: String, value: Any?, newMap: MutableMap<String, Any?>) {
        when (value) {
            is MutableMap<*, *> -> {
                if (
                    value.isNotEmpty() &&
                    newMap[key] is MutableMap<*, *>
                ) {
                    val currentTreeNode = newMap[key] as MutableMap<*, *>
                    value.forEach { entry ->
                        if (
                            entry.key is String &&
                            currentTreeNode.isNotEmpty() &&
                            currentTreeNode.keys.toList()[0] is String
                        ) {
                            mergeInto(entry.key as String, entry.value,
                                currentTreeNode as MutableMap<String, Any?>)
                        }
                    }
                } else {
                    newMap[key] = value
                }
            }

            else -> newMap[key] = value
        }
    }
```

4.3 Benchmarking Tests

JMH benchmarks are annotation-driven. The JMH engine will scan the source sets for methods annotated with @Benchmark and infer the configuration from the rest of the JMH annotation.

All of the benchmarks run the same simulation, the newly implemented configuration override system can be leveraged to re-use the same base configuration file as a template and override the engine configuration as needed.

The following is an example of a single Benchmark:

```kotlin
@Benchmark
@BenchmarkMode(Mode.Throughput)
@Fork(value = 3)
@Threads(4)
@Suppress("unused")
fun multiThreadedSimulationFourThreadsFixedBatch() {
    Alchemist.main(
        arrayOf(
            "run",
            "simulation.yml",
            "--verbosity",
            "warn",
            "--override",
            """
            launcher:
                parameters:
                    parallelism: 4
            engine-configuration:
                type: FixedBatchEngineConfiguration
                parameters:
                    outputReplayStrategy: aggregate
                    batchSize: 4
            """.trimIndent(),
        ),
    )
}
```

4.4 Batch Engine

The batch engine version used in the final codebase was implemented using Kotlin co-routines. Most of the existing engine code can be re-used if we allow to override single-step execution. For this reason, the step engine logic was extracted in a protected method "doStep". Protected methods are visible for extending classes, but not outside of the class boundaries. Another change to the Engine class is a refactor to leverage dependency injection for the Scheduler class. Previously it was instantiated on the object's creation, now it is provided as a dependency in the constructor. This change was needed to differentiate between sequential and batch engines. The step logic implementation advances the simulation by steps equal to the size of the processed batch. For each element in the batch, a new asynchronous

co-routine is created. Their results are then awaited and unwrapped. Finally, once all of the co-routines are synchronized, output monitors are notified using either the ”Aggregate” or ”Replay” strategy.

```
override fun doStep() {
        val batchedScheduler = scheduler as BatchedScheduler<T>
        val nextEvents = batchedScheduler.nextBatch
        val batchSize = nextEvents.size
        if (nextEvents.isEmpty()) {
            newStatus(Status.TERMINATED)
            LOGGER.info("No more reactions.")
            return
        }
        val sortededNextEvents =
            nextEvents.stream().sorted(Comparator.comparing(Actionable<T>::tau)).
                collect(Collectors.toList())
        val minSlidingWindowTime = sortededNextEvents[0].tau
        val maxSlidingWindowTime = sortededNextEvents[sortededNextEvents.size -
            1].tau
        runBlocking {
            val taskMapper =
                Function { event: Actionable<T> ->
                    async {
                        doEvent(
                            event,
                            minSlidingWindowTime,
                        )
                    }
                }
            val tasks = nextEvents.stream().map(taskMapper).collect(Collectors.
                toList())
            try {
                val futureResults = tasks.awaitAll()
                val newStep = step + batchSize.toLong()
                setCurrentStep(newStep)
                val resultsOrderedByTime = futureResults
                    .sortedWith(Comparator.comparing { result: TaskResult ->
                        result.eventTime })
                setCurrentTime(if (maxSlidingWindowTime > time)
                    maxSlidingWindowTime else time)
                doStepDoneAllMonitors(resultsOrderedByTime)
            } catch (e: InterruptedException) {
                LOGGER.error(e.message, e)
                Thread.currentThread().interrupt()
            }
        }
    }
```

Chapter 5

Validation

5.1 Benchmarking Results

As per the analysis, the following test cases were performed:

- singleThreadedSimulation - sequential run
- multiThreadedSimulationFourThreadsFixedBatch - parallel run 4 threads, fixed size batch
- multiThreadedSimulationEightThreadsFixedBatch - parallel run 8 threads, fixed size batch
- multiThreadedSimulationFourThreadsEpsilonBatch - parallel run 4 threads, epsilon batch
- multiThreadedSimulationEightThreadsEpsilonBatch - parallel run 8 threads, epsilon batch

In the following Environments:

- JDK 11 - classic executors
- JDK 11 - Kotlin co-routines
- JDK 19 - classic executors

- JDK 19 - virtual threads executors
- JDK 19 - Kotlin co-routines

On the machine with the following specifications:

- OS - Microsoft Windows 10 Pro
- Processor Model - Intel(R) Core(TM) i7-8565U @ 1.80GHz
- Processor Cores - 4 Cores
- Processor Logical Processors - 8 Logical Processors

The tests produce a score that measures the throughput of the operation for the selected execution. The score is measured in ops/s. The scores also provide an error variance value to better gauge the actual median performance.

JDK 11 - Sequential

This case tests the currently used JDK version in Alchemist with the original sequential implementation of the algorithm.

The test has produced the following results:

Benchmark	Score	Error Variance
singleThreadedSimulation	0.302 ops/s	±0.008

Table 5.1: JDK 11 - Sequential Results

JDK 11 - Java Classic Executors

This case tests the currently used JDK version in Alchemist with a traditional task-based concurrency model in Java (Executors).

The test has produced the following results:

Benchmark	Score	Error Variance
multiThreadedSimulationEightThreadsEpsilonBatch	0.488 ops/s	± 0.063
multiThreadedSimulationEightThreadsFixedBatch	0.519 ops/s	± 0.075
multiThreadedSimulationFourThreadsEpsilonBatch	0.416 ops/s	± 0.014
multiThreadedSimulationFourThreadsFixedBatch	0.409 ops/s	± 0.006

Table 5.2: JDK 11 - Java Classic Executors Results

JDK 11 - Kotlin Co-Routines

This case tests the currently used JDK version in Alchemist with a co-routines concurrency model in Kotlin.

The test has produced the following results:

Benchmark	Score	Error Variance
multiThreadedSimulationEightThreadsEpsilonBatch	0.743 ops/s	± 0.048
multiThreadedSimulationEightThreadsFixedBatch	0.717 ops/s	± 0.075
multiThreadedSimulationFourThreadsEpsilonBatch	0.462 ops/s	± 0.118
multiThreadedSimulationFourThreadsFixedBatch	0.403 ops/s	± 0.021

Table 5.3: JDK 11 - Kotlin Co-Routines Results

JDK 19 - Sequential

This case tests the Alchemist updated to JDK 19 version with the original sequential implementation of the algorithm.

The test has produced the following results:

Benchmark	Score	Error Variance
singleThreadedSimulation	0.301 ops/s	± 0.034

Table 5.4: JDK 19 - Sequential Results

JDK 19 - Java Classic Executors Results

This case tests the Alchemist updated to JDK 19 version with a traditional task-based concurrency model in Java (Executors).

The test has produced the following results:

Benchmark	Score	Error Variance
multiThreadedSimulationEightThreadsEpsilonBatch	0.447 ops/s	± 0.135
multiThreadedSimulationEightThreadsFixedBatch	0.428 ops/s	± 0.115
multiThreadedSimulationFourThreadsEpsilonBatch	0.356 ops/s	± 0.098
multiThreadedSimulationFourThreadsFixedBatch	0.338 ops/s	± 0.094

Table 5.5: JDK 19 - Java Classic Executors Results

JDK 19 - Java Virtual Threads Executors Results

This case tests the Alchemist updated to JDK 19 version with the new virtual thread concurrency model in Java.

The test has produced the following results:

Benchmark	Score	Error Variance
multiThreadedSimulationEightThreadsEpsilonBatch	0.614 ops/s	± 0.022
multiThreadedSimulationEightThreadsFixedBatch	0.628 ops/s	± 0.022
multiThreadedSimulationFourThreadsEpsilonBatch	0.456 ops/s	± 0.013
multiThreadedSimulationFourThreadsFixedBatch	0.427 ops/s	± 0.012

Table 5.6: JDK 19 - Java Virtual Threads Executors Results

JDK 19 - Kotlin Co-Routines Results

This case tests the Alchemist updated to JDK 19 version with co-routines concurrency model in Kotlin.

The test has produced the following results:

Benchmark	Score	Error Variance
multiThreadedSimulationEightThreadsEpsilonBatch	0.580 ops/s	± 0.176
multiThreadedSimulationEightThreadsFixedBatch	0.635 ops/s	± 0.184
multiThreadedSimulationFourThreadsEpsilonBatch	0.486 ops/s	± 0.148
multiThreadedSimulationFourThreadsFixedBatch	0.449 ops/s	± 0.116

Table 5.7: JDK 19 - Kotlin Co-Routines Results

5.2 Results Analysis

The visual and comparative analysis is performed by plotting the scores for sequential and parallel engine for each benchmark:

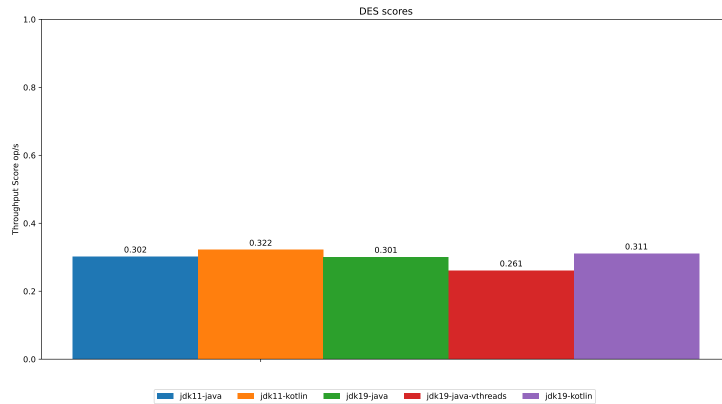

Figure 5.1: Throughput Scores With Sequential Engine

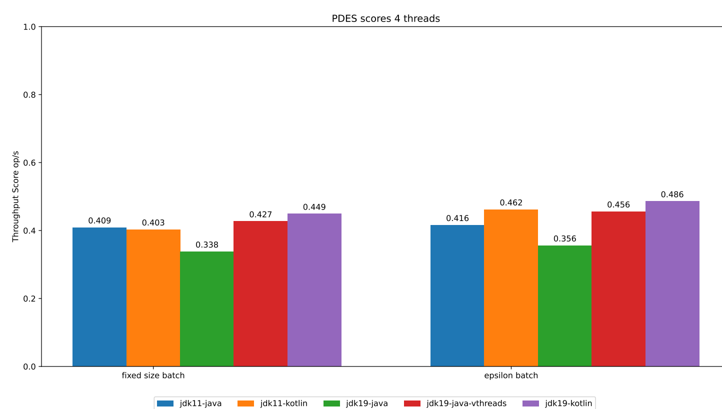

Figure 5.2: Throughput Scores With Parallel Engine 4 Threads

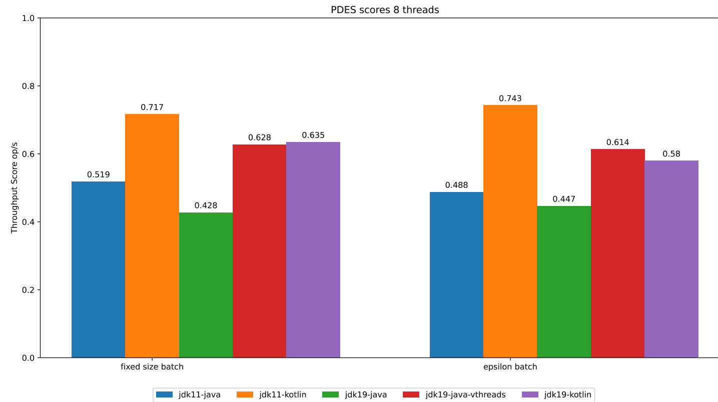

Figure 5.3: Throughput Scores With Parallel Engine 8 Threads

With the following three-sigma variance:

Engine	three-sigma lower bound	three-sigma lower bound
Sequential	0.23749232680838345	0.36130767319161655
Parallel Fixed	0.28754842631917255	0.5528515736808275
Parallel Epsilon	0.27278306787153 53	0.8870169321284647

Table 5.8: Three-Sigma Variance of Benchmark Scores

By comparing the results with the expectation we can deduce the following insights:

- Intuitively, utilizing more threads leads to better throughput scores across all the experiments.
- JDK 19 experiments seem to perform worse than their JDK 11 analogs, however, it may be attributed to the error variance.
- Fixed batch size performs better than epsilon batch. This is expected due to better resource utilization of the fixed batch solution.

- Kotlin co-routines are at a clear advantage over traditional concurrency models in JDK 11, however, JDK 19 virtual threads perform at the same level. This is expected as both solutions utilize virtual threads instead of OS threads, which suites this concurrency problem better.

In conclusion, the results were in line with the expectations. Scaling the resources improves the results and a fixed batch that can saturate the available resources better tends to perform better than an epsilon dynamically sized batch due to synchronization points after each batch processing. Most interestingly, JDK 19 virtual threads perform at the same level as Kotlin's co-routines. This is an important result because this concurrency solution was not previously available in the Java programming language and it is a big benefit for the problems that better fit it.

Chapter 6

Conclusion

This dissertation project was both practical and experimental in its nature. The project has refactored and improved major parts of the Alchemist simulator to allow for a more streamlined and scalable configuration and opened the codebase for the possibility to use different DES implementations. This project has once again confirmed the difficulty of designing and implementing PDES systems, even when sacrificing some of the DES properties. Another important point touched upon in this project is the need to properly asses implementation results by utilizing a robust benchmarking harness instead of naive methods to account for the dynamic properties of the JVM platform. Lastly, the comparison of the same PDES implementation in the different JDK versions and concurrency options has produced interesting, but ultimately expected results. In particular, this project showed that the latest advancements in the historical and consolidated language such as Java allow it to keep up with the more modern and experimental languages such as Kotlin, especially in the cases that benefit lightweight thread concurrency.

Despite the challenges, the project goals were achieved fully and the gained results were insightful and the contributions made to the Alchemist project have laid a foundation for the future works and path forward for continued growth and evolution of the project.

6.1 Future Works

The goal of the project was to experiment with a PDES implementation that optimistically allows for causality errors and study the different concurrency options on the JVM platform. However, this is not the only option for a possible PDES implementation. If the events could be constrained to implement an "undo" mechanism, the simulation could be made deterministic despite the concurrency and the events could be processed with maximum throughput as against in batched by rollbacking the simulation state when a causality error is encountered and corrected. One such example is the Time Warp Simulation [Wil19]. With the new extensibility, more engine implementations and tunable parameters can be made available. More parts of the Alchemist simulator can be made parallel beside the core DES/PDES algorithm. And lastly, the benchmarks were run on a single hardware configuration, which is quite limiting. A thorough benchmarking analysis run across different machines with different consumer and enterprise-grade specs could give a much better sense of the possible throughput variance.

Bibliography

[BFR06] J.-P. Banâtre, P. Fradet, and Y. Radenac. Generalised multisets for chemical programming. *Mathematical Structures in Computer Science*, 16(4):557 – 580, 2006. Cited by: 49.

[CBLA19] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. What's wrong with my benchmark results? studying bad practices in jmh benchmarks. *IEEE Transactions on Software Engineering*, 06 2019.

[CCH11] Kuo-Yi Chen, J. Morris Chang, and Ting-Wei Hou. Multithreading in java: Performance and scalability on multicore systems. *IEEE Transactions on Computers*, 60(11):1521 – 1534, 2011. Cited by: 30; All Open Access, Green Open Access.

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Comput. Surv.*, 21(3):323–357, 1989.

[GB00] Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry A*, 104(9):1876 – 1889, 2000. Cited by: 1206; All Open Access, Green Open Access.

[Gil77] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[LTF+18] Lauren F. Laker, Elham Torabi, Daniel J. France, Craig M. Froehle, Eric J. Goldlust, Nathan R. Hoot, Parastu Kasaie, Michael S. Lyons, Laura H. Barg-Walkow, Michael J. Ward, and Robert L. Wears. Understanding emergency care delivery through computer simulation model-

ing. *Academic Emergency Medicine*, 25(2):116 – 127, 2018. Cited by: 17; All Open Access, Bronze Open Access, Green Open Access.

[PMV11] Danilo Pianini, Sara Montagna, and Mirko Viroli. A chemical inspired simulation framework for pervasive services ecosystems. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Federated Conference on Computer Science and Information Systems, FedCSIS 2011, Szczecin, Poland, 18-21 September 2011, Proceedings*, pages 667–674, 2011.

[PMV13] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simulation*, 7(3):202–215, 2013.

[PN94] Ernest H. Page and Richard E. Nance. Parallel discrete event simulation: A modeling methodological perspective. page 88 – 93, 1994. Cited by: 18; All Open Access, Bronze Open Access.

[Wil19] Philip A. Wilsey. Time warp simulation on multi-core platforms. In *2019 Winter Simulation Conference, WSC 2019, National Harbor, MD, USA, December 8-11, 2019*, pages 1454–1468. IEEE, 2019.

[ZOA+15] Franco Zambonelli, Andrea Omicini, Bernhard Anzengruber, Gabriella Castelli, Francesco L. De Angelis, Giovanna Di Marzo Serugendo, Simon A. Dobson, Jose Luis Fernandez-Marquez, Alois Ferscha, Marco Mamei, Stefano Mariani, Ambra Molesini, Sara Montagna, Jussi Nieminen, Danilo Pianini, Matteo Risoldi, Alberto Rosi, Graeme Stevenson, Mirko Viroli, and Juan Ye. Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive Mob. Comput.*, 17:236–252, 2015.