Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

Optimising Behavioural Oriented Concurrency for Parallelism

Author:
Ryan Ward

Supervisor:
Dr. Marios Kogias

Second Marker:
Dr. Lluis Vilanova

June 19, 2023

Abstract

Behavioural Oriented Concurrency is a new paradigm for implementing concurrent programs. This project analyses an implementation of BoC, namely Microsoft's Verona, implementing an optimisation that enables Atomic Scheduling and Parallel Execution. This optimisation improves the throughput and latency by enabling behaviours to be scheduled based on the dependencies of both whilst allowing execution to happen in parallel.

The project also provides a configurable benchmarking tool for Verona, to test the throughput and latency of the run-time. This benchmarking tool is used to demonstrate the optimisation proposed, displaying graphical outputs and quantitative data, with analysis of this data designed to determine the effectiveness of the optimisation in different simulated scenarios. The paper concludes that the optimisation is effective, whilst proposing additional developments that could be applied to the run-time to further optimise it, improving this newly emerging paradigm into a powerful, easy to use tool.

Acknowledgements

I would like to thank my project supervisor, Marios, who has provided me with significant support and guidance throughout the project and ensured that I was always on track and making good progress, especially during times when my progress was impacted due to exam constraints or poor mental health.

I would like to thank Microsoft for creating the Verona C++ run-time library, enabling this project to be achievable in the time I had to complete it.

I would like to thank my family for providing consistent moral support throughout my degree, always believing in me and showing their pride in me.

Contents

List of Figures

Chapter 1

Introduction

Concurrency is a core component in the development of modern technology. There are various ways to implement concurrency, with threads being the most common approach. The fundamental concept of a computer is that a processor, at its most abstracted level, is a physical device that takes in instructions one at a time, performs them, and produces an output. In more complex programs, programmers may need the processor to handle multiple tasks simultaneously. For instance, when developing a graphical program like a word processor, the programmer may need the processor to run graphical logic alongside tasks such as processing input or saving a file. Concurrency provides a paradigm for implementing this capability by scheduling tasks on the processor to interleave their execution. Without concurrency, computers would be fundamentally limited because programs would need to finish before others could execute. If a program were to perform a lengthy task, other tasks would be deprived of computation time, and essential functions like user interfacing and interaction would be practically impossible.

Similarly, modern computers typically have multi-core processors. Essentially, this means that a single chip incorporates multiple duplicate processors embedded onto it. Several factors contribute to this design choice. Firstly, advancements in computational power have been slowing down, so duplicating cores becomes a means of increasing processing power and circumventing limitations imposed by Moore's Law [1]. Parallel programming serves as a paradigm for effectively utilizing the multiple cores of a processor simultaneously. Typically, the Operating System Scheduler automatically handles this by assigning threads to execute on any available core. Consequently, concurrent processes can run concurrently rather than in an interleaved manner. Parallelism refers to the execution of a program that has been written using parallel programming techniques.

Behavioural Oriented Concurrency (BoC) is a model for concurrent programming that introduces **behaviours** and **concurrent owners (*cowns*)**. This project focuses on Project Verona, Microsoft's Research project on Behavioural Oriented Concurrency [2]. This project is both an entirely new language and a run-time for C++, to enable developers to experiment with Behavioural Oriented Concurrency using a language with much wider support [3].

However current implementations of BoC are somewhat limited, as they are not able to deduce behaviours that are available for parallel execution at schedule-time. The implementation that this project focuses on is Microsoft's Verona, a BoC run-time for C++ that enables "when-like" behaviours. [3] Each behaviour is provided to the scheduler upon calling the "when" function, so the scheduler tries to dynamically assign a "route" through each behaviour. This can lead to behaviour starvation if not set up correctly. The main focus of this project is to allow the programmer to specify behaviours that are to be executed in parallel in order to increase throughput and reduce latency for these behaviours, fundamentally increasing the efficiency of the available run-time.

A common approach to measure Parallelism is to compute the throughput of tasks, i.e. how many "blocks" of code can be swapped and completed in a period of time, or the latency of tasks, i.e. the time between a task being scheduled, and the start of execution. These measures are generally defined by comparing the difference between latency and throughput when improving the resources of a computer, with speedup formally established by Amdahl's Law [4]. These are important metrics in measuring how parallel the execution is. This project builds a benchmarking tool to measure how parallel Verona is, then optimises Verona to execute as parallel as possible before using the same benchmark to measure the improvement in performance.

1.1 Objectives

This project aims to implement an optimisation to the Verona C++ run-time [3] that improves its performance under a parallel workload. In order to develop these optimisations, the project must implement:

- A benchmark that analyses the throughput and latency of a program developed using Verona's when() blocks. This benchmark should be configurable to allow the user to define how many behaviours, the distribution of Concurrent Owners, and other metrics.
- A program that graphically displays the throughput and latency (rolling throughput and average throughput) once the benchmark has concluded.
- Optimisations to the Run-time that enable the concept of "Atomic Scheduling, Parallel Execution," increasing the execution throughput. A more detailed explanation is presented in Chapter 4.

1.2 Contributions

This project implements alterations to the scheduler, enabling developers to arrange programs in a specific way to increase throughput and reduce latency. It's main contributions are:

- A benchmark, `lotsofcowns.cc`, which can be called from the command line, or the script provided, to test the throughput and latency given a specific configuration.
  - A program, `graphOut.py`, that accepts a list of doubles (times) as command line args, and outputs a graph displaying behaviour latency and throughput, used in `lotsofcowns.cc`
  - A distribution package, `zipfDist.h`, that implements random number generators for the Zipfian, Uniform, Bimodal and Exponential Distributions.
- An optimisation to the run-time that enables Atomic Scheduling and Parallel Execution.

Chapter 2

Background

2.1 History

2.1.1 Concurrency

Primary motivations for the project are the drawbacks to currently available tools for concurrent programming. Concurrency is very useful in helping developers utilise the full power of a processor so long as the developer is adept enough to use these tools correctly. [5]. The main disadvantage is the inability of a concurrent program to know whether data accesses are being executed in the correct order [6]. As a result, threads and locks can be introduced. Threads are the independent sequences of instructions which can be executed concurrently, whilst locks enable communication between threads (by stopping another thread from continuing if the memory region is being accessed.) Unfortunately, this still has its downsides; if multiple threads are accessing the same regions of data and the locks aren't set up to be acquired and released correctly by the programmer, the program can deadlock. Resolving deadlocks is a primary topic of research within the world of concurrent programming as it is not intuitive and requires a lot of time and thought from the programmer [7]. Similarly, if the programmer avoids using locks so as to circumvent deadlocks, the threads may overlap accesses to memory. Since the developer cannot determine the scheduling algorithm, this ends up in non-deterministic execution. This problem is known as a race condition. [8]

In Figure 2.1, the trains are each following a track that leads to a shared piece of track (like a station, for example.) Without communication, these trains are on a collision course with each other. As a result, this scenario can be approached in 3 different ways. The first would be considering "polite" trains: both trains stop, and wait for the other to use the track. They would wait and wait and wait because both are waiting for each other to tell them that they have used the track and it's now safe to pass. If the trains were threads, and the track being code, this would be a deadlock. The second approach would be to do nothing and let the trains figure it out themselves. This would mean that either train 1 goes first, with train 2 following, or vice-versa.

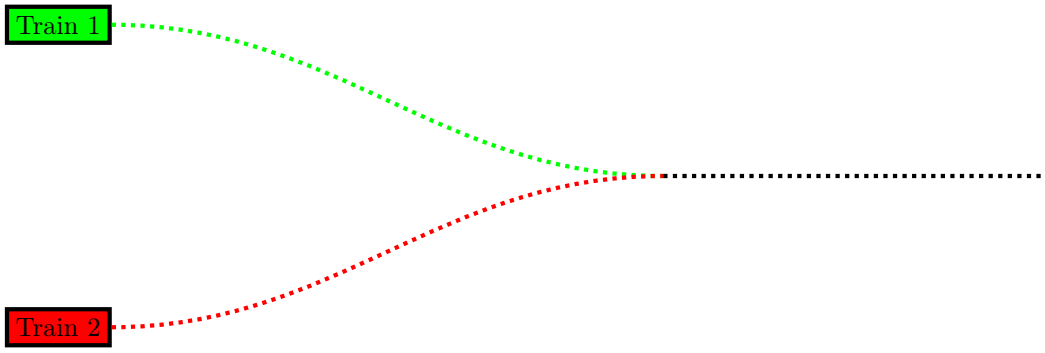


Figure 2.1: Train deadlocks & race conditions

Because this hasn't been controlled it is unknown what the order of the trains would be at the end: this would be the Race condition.

The final approach and the one used in the real world, would be to install a signalling system. This signalling system would tell the train that arrives first that the track is free, and would tell the other to wait until the first train has passed. Once the first train has passed, the second train is then told to use the track. If another train arrives (on track 1) Then a queue would form, ensuring train 2 is not left waiting. If this were the case, and the trains on the green track were always given priority, Train 2 would never progress; in a computational perspective, this is known as starvation. This signalling system stops trains from getting stuck or crashing into each other. Whilst this is usually now implemented with coloured lights, it was initially implemented with a mechanical signal known as a ***semaphore*** [9].

In the early days of operating system design, little thought was given to the idea of "doing multiple things at once." In MS-DOS, there was no pre-designed multi-tasking. It was an inherently "single-task" OS, however, there were ways in which a programmer could create a concurrent workload, albeit not extremely efficient [10]. Programmers could create Terminate and Stay Resident programs (TSR programs) - which would pre-emptively terminate as though it had concluded execution, but stays resident in the memory so that it could be recalled later and continue execution. This acts very similarly to a thread, but termination means that the Operating System Scheduler must reschedule the task, making execution very inefficient. MS-DOS was vulnerable to Malware as a result of TSR programs, as they could sit latent in the memory and take advantage of Disk IO / Executable events within the OS. [11]

2.1.2 Synchronisation Primitives

This paper exemplifies the same functionality using alternative models to display differences between currently available techniques. The functionality will be to simply create two integer variables, increment them both, and output the sum of them. The resulting functionality should output 2.

Locks

For modern, systems, concurrency is implemented through the use of threads. In order for threads to be synchronised, avoiding race conditions and deadlocks, Locks have been created to avoid conflicting thread accesses of memory locations. Locks are designed to utilise Operating System flags - two functions are important within the lock class: acquire and release. If the lock is free, it is acquired by the calling thread. If the lock is not free, then the yield system call is called and the thread is yielded, and system control is passed to another thread. This stops the execution of the thread and ensures the critical section of code is not executed until it is freed by the previous locking thread. The release call "frees" the lock. This allows a yielded thread to return to execution when the scheduler switches back to it.

Locks are an essential mechanism in ensuring the correctness of a program whilst maintaining concurrent execution of the program. There are plenty of benefits to the use of threads, they are simply included and are available on most systems and in many languages, whilst enabling programmers to finely tune the granularity of concurrency within their programs. The freedom to control synchronisation enables the programmer to write their program in any way they see fit, however, this allows for incorrect solutions with deadlocks or race conditions.

Figure 2.2 leads to a deadlock because neither thread can progress past the second command as a result of each waiting for the other to release the lock they acquired in the first command. As such, execution is effectively stopped. There is no real resolution to this other than having the programmer fix the problem, and this debugging can take a long time to figure out and is sometimes a result of the problem the programmer is trying to solve. As a result, concurrent programming requires more capable programmers and is more time-consuming than creating programs that

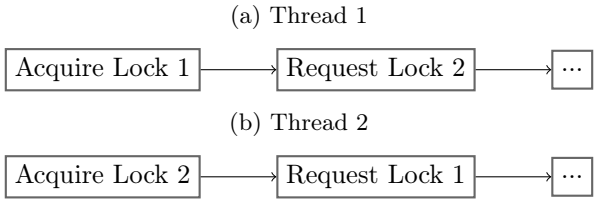


Figure 2.2: Thread Deadlock

execute sequentially [12].

Semaphores

The semaphore is another type of synchronisation primitive, similar to the lock, and is primarily used in concurrent programming to control access to shared resources. Semaphores differ from locks in that they are based on a counter - when initialising a semaphore, you define the value of the counter (as a positive integer) and use up/down to increment or decrement the counter. If the counter is 0, and down is called, the calling thread is blocked, and the thread is added to a queue. If more threads attempt to call down, they are appended to the queue and also blocked. As up is called, it either increases the counter or pop's from the front of the queue. This allows the programmer to enable a limited number of threads to access a shared resource and fairly control which thread gains access to the resource once it is freed.

The semaphore was first introduced by Dijkstra in 1965 as a mechanism for coordinating access to shared resources [13]. As mentioned before, they are named after the signals that trains used to use to determine if a shared piece of track is free to travel along. Just like with trains, however, these complex systems are prone to failure. Semaphores are notoriously difficult to implement and are prone to enabling a non-deterministic environment for execution. It is left to the programmer to ensure that data is only accessed in a safe manner.

Similarly, if they are small and frequent, regularly locking and unlocking (or incrementing/decrementing) locks and semaphores can accrue quite a large amount of execution time, especially if you are doing this almost as much as you are executing the program. As such, the idea of "lock granularity" exists - rather than using many locks and locking and unlocking them again and again, using less frequent locks for larger blocks of data may be beneficial. [14] The problem with this is that the more coarse the locks are, the less concurrent the program becomes, as the critical section increases in size (and hence, execution time.)

Below is an example of a program that increments two variables and outputs the result. Locks and semaphores here aren't necessary as there are no data dependencies between the threads.

```
int a = 0;
int b = 0;
void f1() {
        a++;
        return NULL;
}
void f2() {
        b++;
        return NULL;
}
void main() {
    pthread_t t1;
    pthread_t t2;
    pthread_create(&t1, NULL, &f1, NULL);
    pthread_create(&t1, NULL, &f2, NULL);
    pthread_join(&t1, NULL);
    pthread_join(&t2, NULL);
```

```
    std::cout << a + b << std::endl;
    return 0;
}
```

2.1.3 Alternatives to locks and semaphores

Futures

C++ is a very versatile language, hence unsurprisingly there are multiple ways to enact concurrency into C++ programs. A popular approach is to utilise **futures**. A future is a mechanism to schedule and execute asynchronous tasks, and the C++ package includes one main class, and one main function: `std::future<T>` and `std::async<T>`. These packages provide the main functionality, which is as follows:

- std::async takes a lambda function, which defines the behaviour of asynchronous tasks.
- std::async will return a std::future class.
- std::future contains a function, get(), which waits for the lambda function defined to return, and provides the result.
- std::future also contains a function wait() which waits for the future to finish but doesn't consume the result, allowing multiple futures to be awaited before continuing execution.

Futures are useful for organising functions that don't interact to execute in parallel, but defining functions that use the same variables isn't quite as easy. For starters, there are no thread-safety mechanisms: variables are accessed via lambda capture, which has two main drawbacks. Firstly, the default here is capture-by-copy, which means changes made within the future won't be reflected if called elsewhere in the program, and 2, if you specify a capture-by-reference, there is no guarantee the future won't be interrupted when accessing that variable. In fact, if the variable is used anywhere else, without first calling wait() on the future, you're almost guaranteed a race condition. One way to fix this, for futures, would be to schedule which async functions can run depending on the variables it uses. [6]

In a similar function to earlier, below is increments and sums two variables. However, this uses futures instead of threads.

```
#include <iostream>
#include <chrono>
#include <future>
int a = 0;
int b = 0;

void main() {
    auto f1 = std::async(std::launch::async, [=, &a]() {
        a++;
        return 0;
    });
    auto f2 = std::async(std::launch::async, [=, &b]() {
        b++;
        return 0;
    });
    f1.wait();
    f2.wait();
    std::cout << a + b << std::endl;
    return 0;
}
```

Transactions

Transactions are more commonly utilised in database management, in which data integrity is vital for the correct functioning of the system. Transactions must conform to the ACID properties

- Atomicity, Consistency, Isolation and Durability. Transactions consist of single commands or groups of commands that can be executed as a package and committed in a single step. In order to keep concurrency high, the granularity of transactions should be as fine as possible. If multiple updates are made, it's better to use a group of transactions than a single transaction that executes them all. Transactions have the unique ability to be Rolled-back. If a transaction fails - for example, an update cannot be made, or a lock is encountered in the middle of the transaction, then the whole transaction can be rolled back, returning the system to its previous state, enabling other transactions to make changes. MySQL, a common query language, includes the commands BEGIN TRANSACTION, COMMIT TRANSACTION and ROLLBACK TRANSACTION [15].

Transactions don't exist in C++, since they are a database concurrency model. Consider a database with a table, "vars", with rows [(ident, val), (a, 0), (b, 0)]. The below transactions update the table, with the final query returning the sum. This is an SQL-based version of the prior "increment and sum" program.

```
BEGIN TRANSACTION;
UPDATE vars SET val = val + 1 WHERE ident = "a";
COMMIT;
BEGIN TRANSACTION;
UPDATE vars SET val = val + 1 WHERE ident = "b";
COMMIT;
SELECT sum(val) FROM VARS;
```

Actors

The Actor model is designed around actors, which are the building blocks for its concurrent execution. Communication is made via messages, responding to which an actor can: make local decisions, create actors, send messages, and determine how to respond to forthcoming messages. Actors can only communicate with each other via messages, removing the need to use threads and locks. Actors were first formalised in 1973 [16] and were expanded upon in 1985 after being developed into a fully transition-based semantic model by Agha [17]. Parallelism is achieved in the actor model by decomposing the program into individual code blocks that can be executed in parallel regardless of the usage of their output. However, actors are limited in the fact that they rely on their isolation to provide parallelism, making processes that involve the use of multiple actors unfeasible. As such, many actor systems do not follow the actor model or have complex coordination systems atop of the underlying actor model [18].

The Actor Model is a message-based system, and is primarily used in web-request response behaviour. To create the "increment and sum" program, some assumptions would have to be made. Assume that there is a C++ header, "Actor.h," with a class that implements actor functionality.

```
#include "Actor.h"
class NumActor : public Actor {
public:
    NumActor() {};

    val = 0;
    int recieve(std::string message, int val) : override {
        if (message.compare("add") == 0) {
            this->val = this->val + val;
            return 200;
        }
        if (message.compare("get") == 0 {
            return this->val;
        }
        return 404;
    }
}

int main(int argc, char **argv) {
```

```
    NumActor* a = new NumActor();
    NumActor* b = new NumActor();
    a.recieve("add", 1);
    b.recieve("add", 1);
    std::cout << a.recieve("get", NULL) + b.recieve("get", NULL) << std::endl;
    return 0;
}
```

2.2 Measuring Parallelism

2.2.1 Speedup

Speedup is a standard term for measuring performance improvements in a parallel system. Formally defined as the improvement in throughput or latency of a particular task on two different machines or architectures, using the equations:

$$S_{latency} = \frac{L_1}{L_2} \qquad S_{throughput} = \frac{Q_2}{Q_1} [19, 20]$$

Where S is the speed-up of the program run on machine 2 against machine 1, L_i is the latency of the program on machine i, and Q_i is the throughput of the program on machine i. The numerator and denominator are opposed in the two equations, since a lower latency depicts a faster execution, as opposed to a higher throughput equating to faster execution.

This project augments the formal definition of speedup such that instead of comparing two systems, the comparison is between the throughput and latency of an unoptimised implementation of Verona and an altered version that includes the optimisation proposed.

2.3 Behavioural Oriented Concurrency

2.3.1 The BoC Model

Behavioural Oriented Concurrency introduces the concept of **behaviours**[1], which in many ways are very similar to actors in the actor model, and the asynchronous functions in the futures model. They are building blocks for concurrency within the system and are where the majority of logic is designed to be implemented in a concurrent system. BoC augments underlying language logic by including **Concurrent owners** and **behaviours**. Cowns represent unique entry points for specific data points in the program and are the only access point for those data points. As such, the cown protects the piece of separated data. Behaviours are the units of execution and are spawned with a list of cowns and a closure on the function. Once spawned, the behaviour can be run (that is to say, be executed) only when both the cowns are available and the behaviours that happen before it has been run. Once available, the cowns become acquired by the behaviour *atomically*, and become unavailable to other behaviours. During execution, the behaviour retains exclusive access to its cowns and their data, and cannot acquire any new cowns. Like actors, BoC decomposes the states of behaviours into separate entities (the cowns), however, unlike actors, behaviours are not confined to a single entity [24, 17].

2.3.2 Cowns and Behaviours

Two key language features of BoC are of interest; `cown<T>` and `when()`.

- The `cown<T>` type represents a cown encapsulating an object of type T. The cown class includes a `create()` [2] function. In the Verona run-time, [3], the create function is defined as `make_cown<T>()`

[1] As Behavioural Oriented Concurrency is a new, unreleased paradigm in and of itself, this section is primarily researched from Microsoft's run-time [2, 3, 21] and an accompanying public lecture at Cambridge University [22, 23] analysing the code in the tests already created, along with help from My supervisor, Dr Marios Kogias.

- the `when()` expression consists of a set of cowns and a closure function, which is used to spawn behaviours.

It is left to the underlying language to ensure that memory access is valid when accessing data inside cowns [21]. That is, it is left to the heap to ensure that the cown is the only access point for the underlying data in the cown. By making the underlying object a private member of the cown, this requirement will be fulfilled.

```
void main() {
    auto a = 1;
    a++; // Valid memory access
}

void main() {
    auto a = make_cown<int>(1);
    a++; // Invalid access
}

void main() {
    auto a = make_cown<int>(1);
    when(a: cown<int>) { a++; }; // Again, a valid memory access
}
```

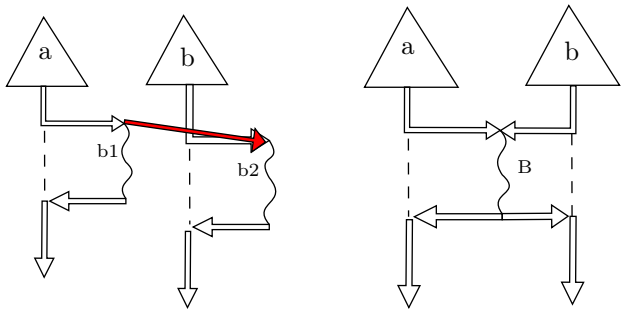
Behaviours also have the right to spawn new behaviours. By nesting a `when()` inside another, it is possible to have behaviours spawning with conditionally based logic inside another behaviour. [23] Note that in this scenario, `when(b)` does not have access to `a`'s value, and `when(a)` cannot access `b`.

Nested Behaviours: Figure 2.3 (a)

```
void main() {
    auto a = make_cown<int>(1);
    auto b = make_cown<int>(2);
    when(a: cown<int>) { // b1
        if (a > 5) {
            a--;
            when(b: cown<int>) { b++; } // b2
        }
    }
}
```

Multiple Cowns: Figure 2.3 (b)

```
void main() {
    auto a = make_cown<int>(1);
    auto b = make_cown<int>(2);
    when(a: cown<int>, b: cown<int>) { // B
        a--;
        b++;
    }
}
```



(a) Nested Behaviours (b) Multiple Cowns

Figure 2.3: Behaviour Diagrams

2.3.3 Causal Ordering, Dynamic Dependencies

Having no overlapping cowns between multiple behaviours implies that there are no overlapping data dependencies in the behaviours. As a result, these behaviours can be executed in parallel, or in any order. However, there is no rule stating that there cannot be overlapping cowns. As such, a formal rule is defined to ensure data dependencies are handled using a standardised procedure.

Formally, BoC introduces a definition of *happens before*, namely; "*behaviour b happens before behaviour b' iff b and b' have a common cown, and b is spawned before b'.*" This can be shown by example. [23]

```
void main() {
    auto a = make_cown<int>(1);
    auto b = make_cown<int>(2);
    auto c = make_cown<int>(3);

    when(a: cown<int>) { a--; } // spawned first, acquires a
    when(a: cown<int>, b: cown<int>) { // spawned second, waits for free a,
    // so executes after the above when
        a--;
        b++;
    }
    when(c: cown<int>) {
        c = c * 2; // executed concurrently at any point in the program.
    }


}
```

This is managed due to the dynamic scheduling algorithm. As each closure is formed, the scheduler will attempt to acquire the cown, and must wait for the previous acquisition of the cown to complete. As a result, the first behaviour to attempt to acquire the cown will be the first to be given access to it, ensuring that the above is true, creating a causal order over multiple cowns. This is true no matter how many behaviours attempt to acquire the cown.

```
void main1() {
    auto a = make_cown<int>(0);
    for (int i = 0; i < 10; i++) {
        when(a: cown<int>) { cout << i << " "; } // 1 2 3 4 5 6 7 8 9 10
    }
}

void main2() {
    std::vector<cown<int>> a = new std::vector<cown<int>>;
```

```
    for (int i = 0; i < 10; i++) {
        a.emplace_back(make_cown<int>(i));
    }

    for (int i = 0; i < 10; i++) {
        when(b: cown<int> = a[i]) { cout << b << " "; }  // Output in any random order
    }
}
```

In main1, each behaviour created by the for loop will be executed in order, because they are accessing the same resource. In main2, however, there are no overlapping cowns, and as such, the program will dynamically schedule and execute concurrently.

2.3.4 Verona

Microsoft's Verona is a research implementation of BoC, with a C++ runtime [3]. The runtime is fully-featured and includes very similar syntax to the defined syntax in this report, with a few key considerations (to make it C++ compilable.)

Firstly, as mentioned before (and as used in examples) `cown.create()` is instead named `make_cown<T>()` for the runtime, and is available in `namespace::verona::cpp` [3]. This does not create a cown, but a cown_ptr, which is instead used for controlling access to the underlying data.
The when() expression is a function, that accepts a variadic argument template for cowns, and passes them to a When object. This allows the programmer to write when(c1, c2, c3) for example, and doesn't expect a specific number of arguments, and has no upper limit to the number of cowns provided.
In order to provide the behaviour functionality and the closure on said behaviour, C++ lambda functions are used. The syntax for this is the following:

```
int main() {
    cown_ptr<int> cown_a = make_cown<int>(0);

    when(cown_a) << [=](int a){
        a++;
    };

    return 0;
}
```

The function, <<, provides the closure on the lambda function and tells the runtime to schedule the behaviour on the set of cowns. In this, it is also shown to be necessary to define the cown inside the parameter list for the lambda function. This allows the lambda function to access the private member of the behaviour - in the above example, `int a` accesses the encapsulated object of `cown_a`.

An implementation of the "increment and sum" concurrent program with BoC is proposed:

```
    cown_ptr<int> cown_a = make_cown<int>(0);
    cown_ptr<int> cown_b = make_cown<int>(0);
    when(cown_a) << [](int a){
        a++;  // First
    };
    when(cown_b) << [](int b){
        b++;  // Concurrently with above
    };
    when(cown_a, cown_b) << [](int a, int b){  // Causal ordering: required
        std::cout << a + b << std::endl;   // to wait for both above to finish
    };
```

Chapter 3

The benchmark

In order to determine the success of our optimisations, this project starts by developing a benchmark, testing the parallel performance metrics of Verona, the run-time for C++ that implements Behavioural Oriented Concurrency. This project particularly wanted to focus on optimising for improvements to the throughput of behaviours when scheduled using the runtime, along with reducing the latency of each behaviour being run. In order to test these metrics, the project begun by looking at ensuring our benchmark is fair and accurately depicts a typical program written in the BoC paradigm.

In order to create a fair benchmark, the project produces a large number of concurrent owners, before allocating specific Cowns to behaviours. This needed to be controllable by the user (or in this case, tester) as it should be able to show how Verona becomes less and less efficient the more frequent Cowns appear. Frequent Cowns depict a busy piece of memory, or a busy data point, which would be typical of a standard program. The project creates a specific amount of cowns, defined by the command line argument –cownCount, stored in a C++ vector, which is the global definition of all cowns in the system. Once this has been created, a for loop is used to pick subsets of the large C++ vector of cowns, and use these subsets of cowns for each behaviour. The behaviours themselves spin for a pre-defined amount of time, add scheduling latency to a global list and return.

3.0.1 Cown Generation

In order to do this, It was approached in a step-by-step fashion. As the project required knowledge of BoC and an ability to write programs using this new paradigm, it was necessary to research how it worked beforehand. The approach, therefore, was to gather a basic understanding of writing behaviours and assigning cowns, whilst using prior knowledge of C++ for the other parts of the project, before merging them together. The first step was to develop a program that scheduled 4 behaviours to run, using the `when()` function. This was acheived by creating a small number of cowns manually, and manually choosing which cowns to add to which behaviour. The general syntax from earlier was used to do this and it was relatively easy. The cowns themselves are essentially placeholders in the benchmark, designed purely to control the concurrent manner of the behaviours.

```
std::vector<cown_ptr<cown>> *cowns = new std::vector<cown_ptr<cown>>;
for (int i = 0; i < no_of_cowns; i++) {
    cowns->emplace_back(make_cown<cown>());
}
```

In order to test the parallelism of the run-time, the benchmark must organise cowns in a distributed, real-world random fashion. As such, the Zipfian random distribution is used: this distribution usually depicts the natural distribution of words in written text, yet it also depicts realistic memory access characteristics [25]. The Zipfian distribution accepts a real number (in the generator, this is a double) as a constant, which controls the rank of values (rank being an order of value popularity,) with 0 being a uniform distribution, becoming less and less uniform as the value increases. The Generator itself picks a new value at the call to nextInt(), seen below:

```
class ZipfianGenerator : public Generator
{
public:
    ...
    int nextInt(int itemCount) {
        if (itemCount != countForZeta) {
            if (itemCount > countForZeta) {
                zetan = zeta(countForZeta, items, theta, zetan);
                eta = (1 - pow(2 / items, 1 - theta)) / (1 - zeta2Theta / zetan);
            } else {
                if ((itemCount < countForZeta) && (allowItemDecrease)) {
                    std::cout << "WARNING: RECOMPUTING ZIPF. SLOW..." << std::endl;

                    zetan = zeta(itemCount, theta);
                    eta = (1 - pow(2 / items, 1 - theta)) / (1 - zeta2Theta / zetan);
                }
            }
        }
        double u = (double)((std::rand() % 1000) / 1000.0);

        int ret = base + (int)(itemCount * pow(eta * u - eta + 1, alpha));

        setLastValue(ret);
        return ret;

    };
    ...
};
```

See Appendix A.2

3.0.2 Scheduling Behaviours

The project's primary objective is to analyse The Verona Run-time's ability to execute multiple behaviours in parallel. In order to do this, the benchmark must schedule and execute multiple behaviours. The way in which behaviours are scheduled has been made relatively simple by the Verona C++ run-time, as mentioned above. As such, the benchmark uses a simple for loop to schedule a configurable number of behaviours. Simply, it takes the command-line argument –whenCount, and stores the value as an integer. The behaviours will be dynamically scheduled on each loop iteration.

3.0.3 Timing Behaviours

From this, determining what each behaviour does, simulating the act of processing something, is the next step. Behaviours are essentially code blocks, defined as finite amounts of work - so long as the code block completes. To simulate work in the benchmark, and the best way to do this is with a loop and a timer. This project implements a function, `spin(double seconds)` which takes the time epoch at which it is called, and checks in a while loop, the difference between the current time and the initial epoch, then comparing this duration to the parameter seconds. Once the loop ends and the time duration has elapsed, the function returns. This simulates work for a determined amount of seconds and enables a simple definition of the function. User input is used to determine how much work each behaviour should do, given by the –servTime argument.

```
std::vector<double>* timeList = new std::vector<double>;

for (int i = 0; i < no_of_whens; i++) {
  // Create sub_arr, and time variable t1 to be captured by the lambda
  when(*sub_arr->data()) << [=, &lock](auto){
```

```
    // Behaviour Functionality. Spins, gets time since t1 and adds to timeList.
    // A conditional block in when() determines based on timeList if it is the final
     // behaviour to execute.
  };
}
```

See Appendix A.1. The same library is then used in the behaviours to time the latency between scheduling and the beginning of execution. By using capture-by-copy, the behaviour is provided the variable "t1" which is taken just before the when() function is called and creates a variable "t2" in the very first line of the behaviour. By using capture-by-copy the benchmark ensures that the "t1" variable is unique for each behaviour as the exact time measurement, in milliseconds, of when the behaviour is scheduled. It is then possible to find the amount of time it takes to begin executing by taking the difference (as the C++ type "duration"), which is then stored in the "timeList" vector, which is thread-safe via locks.

```
auto spin(double seconds) {
  high_resolution_clock::time_point t1 = high_resolution_clock::now();
  duration<double> timespan = duration_cast<duration<double>>(t1 - t1);

  while (timespan.count() < seconds) {
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    timespan = duration_cast<duration<double>>(t2 - t1);
  }
}
```

The timeList vector is filled with 1 double per behaviour, representing the time it takes to begin executing the behaviour. In order to display this information, The behaviour measures the size of the vector, checking how many behaviours have been executed so far. By checking if the timeList vector's size is equal to the "no_of_whens" variable, it checks if all behaviours have been completed. If so, the behaviour uses the created function "printTList" to turn all of the values in the vector into a string, before making a bash call to the program "graphOut.py," written in Python, that accepts all of the time values as command line arguments and creates 2 graphs from them.

The first graph displayed is simply a time-series graph of the percentage of behaviours complete in a given time frame. This graph shows us the latency of the behaviours: i.e. how long does it take to execute 5, 25, 50, 75, 95% of behaviours? The program then creates a new array, counting how many behaviours are completed in each second - i.e. how many are completed before 1 second elapses? Between 1 and 2 seconds? This Rolling throughput is then used to calculate the average throughput. These graphs are then displayed using the Python module Matplotlib, which, due to its simplicity, was the primary reason for transferring the values to Python.

3.0.4 Picking Cowns

Random Distribution Generators

In order to test the run-time's parallel abilities, designing a system for picking cowns in a natural way, similar to how most programs would be picking them, is vital. As mentioned before, the Zipfian distribution is a perfect way to do this - the usage of memory will follow Zipf's law, meaning variable usage (and hence cown usage) will also follow Zipf's law. [25] As a result, developing an integer value generator for the Zipfian distribution was necessary. This project created the file `zipfDist.h`, a file within the test suite of Verona that includes a Zipfian distribution generator. This generator takes the standard formula for the Zipfian distribution, with an input constant. The Zipfian distribution is represented with the following probability mass function:

$f(x) = Cx^{-(s+1)}, x = 1, 2, ..., s > 0$

$C = [\sum_{x=1}^{\infty} x^{-(s+1)}]^{-1}$

$H(b, s) = \sum_{x=1}^{b} (\frac{1}{x})^s$

$F(x) = \frac{H(x,s)}{H(b,s)}$ [26]

Given that time is cumulative, it is expected that the behaviour latency graph roughly follows the cumulative mass function, or, $f(X \leq x)$.

Following from the Distribution generator, the project creates sub_array_random(), which takes the following parameters: **std::vector<cown_ptr<cown»*** arr, **std::vector<cown_ptr<cown>>*** sub_arr, **int** count, **bool** uniform, **double** zipfConstant = 0.99. If the `uniform` parameter is true, the uniform distribution is used. In this case, the behaviour latency follows a straight-line graph since each behaviour has a uniformally random selection of cowns, which generally leads to enough behaviours scheduled that there is a chance for concurrency. The function takes in a vector `arr`, which in this implementation is the vector of all cowns available. The vector sub_arr is the vector used to store the randomly selected cowns. `count` is the number of cowns to pick from `arr`. This then picks, using the designed generators, indices from 0 to arr->size() and emplaces them into sub_arr uniquely, allowing us to create new arrays for each behaviour dynamically.

Uniqueness and popularity

In order to ensure the program isn't picking multiple of the same index to include in the subarray, the functionality of `sub_array_random()` had to be altered to check against values already picked. This was difficult, as high values for the Zipfian constant leads to many, many calls to nextValue() within the generator - since the probability of popular cowns is increasing as the constant increases. As a result, a simpler solution was required: the subarray must not be too large (as a subarray of over around 25% leads to the Zipfian distribution repeatedly picking popular cowns) and there must be a quick way to check values are unique. A set was used to do this since C++ sets are simple, and include the function `emplace()`, which returns both the index in which the value was placed into the set (since these can have specific ordering requirements) and more importantly, a boolean to describe whether or not the value was unique and was able to be placed into the set. As such, a simple do-until control flow enabled the sub_array_random function to create unique indexes relatively quickly.

```
void sub_array_random(std::vector<cown_ptr<cown>>* arr, std::vector<cown_ptr<cown>>*
sub_arr, int count, bool uniform, double zipfConstant = 0.99)
{
    Generator *gen = new Generator;
    std::set<int> indexes;
    if (uniform) {
        gen = new UniformGenerator(0, int(arr->size()) - 1);
    } else {
        gen = new ZipfianGenerator(0, int(arr->size() - 1), zipfConstant);
    }
    for (int i = 0; i < count; i++) {
        int index = gen->nextValue();
        while (!indexes.emplace(index).second) {
          index = gen->nextValue();
        }
        cown_ptr<cown> ptr = arr->at(index);
        sub_arr->emplace_back(ptr);
    }
}
```

To control popularity with the benchmark, simply increase the zipfConstant provided in the command line argument to make popular cowns more popular, and reduce the zipfConstant to reduce their popularity, ensuring zipfConstant > 0.

3.0.5 Service Time

Choosing the service time generator configuration involves ensuring that they provide an equal average value. As a result, it is necessary to look at the expectation values for the service time distribution generators chosen and ensure the values picked for the service time equate to equal expectations from the generator. The three generators, fixed, bimodal and exponential have relatively simple equations for expectation; $E(fixed[a]) = a$, $E(bimodal[a, b, p]) = (a * p) + (b * (1 - p))$, and $E(exp[\lambda]) = \frac{1}{\lambda}$.[27]

After the benchmark had been realised, it was necessary to run the benchmark to get pre-optimisation figures for the throughput and latency of the run-time. To do this, it was necessary to find feasible values for the parameters of the benchmark. It is necessary to determine how the run-time handles different situations, so the parameter choices must depict a variety of workloads. By developing the benchmark first, which essentially tests the optimisation, the project has conducted a test-driven development style.

3.0.6 Configurability

The benchmark includes command line arguments, which enable easy configuration, controling what metrics the benchmark uses. These are:

- `-cownPop uniform | zipfian[zipfConstant]`. This controls the distribution generator used to choose cowns from the global cown list.
- `-servTime fixed[msecs] | bimodal[fast:slow:percentage] | exp[lambda]`. This controls the distribution generator used to determine the service time. If the benchmark is using the proposed optimisation, fixed[ms] should be used to ensure each behaviour takes the same amount of time (to ensure a fair test.)
- `-totalCowns no_of_cowns`. This determines the number of cowns in the global cown list - the more there are, the less likely cowns are to repeat (depending on the distribution used.)
- `-behaviourCowns pct`. This determines, as a percentage of totalCowns, how many cowns each behaviour should pick from the list. If you input –totalCowns 1000 –behaviourCowns 5, there would be 50 cowns per behaviour.
- `-whenCount no_of_whens`. This determines how many behaviours should be spawned and executed.
- `?-parallel`. This optional argument determines whether the benchmark tests Verona with or without the optimisation.

Chapter 4

Optimisation: Atomic Scheduling, Parallel Execution

As mentioned previously, the project's primary objective is to implement optimisations to the Verona run-time to increase parallel execution. In order to do this, the project takes a dive into scheduling multiple behaviours atomically, such that they are executable in parallel. This is left to the programmer to determine which behaviours they would like to split into an atomic block of multiple behaviours. Take the following behaviour:

```
cown_ptr<int> cown_a = make_cown<int>(0);
cown_ptr<int> cown_b = make_cown<int>(0);
when(cown_a, cown_b) << [](int a, int b){
    a++;
    b++;
};

when() << [](){
    when(cown_a, cown_b) << [](int a, int b){
        std::cout << a + b << std::endl;
    };
};
```

The behaviour `when(cown_a, cown_b)` accesses a and b separately, yet they don't require each other to be added. However, this functionality should be atomic; it needs to be executed at the right time. The when() is designed to state that the second when may be called at any point and could be executed whenever - as a result, the expected functionality of this program is an output of 0, or 2, to the command line. To parallelise the first behaviour, one may consider writing the following with the Verona run-time:

```
cown_ptr<int> cown_a = make_cown<int>(0);
cown_ptr<int> cown_b = make_cown<int>(0);
when(cown_a) << [](int a){
    a++;
};
when(cown_b) << [](int b){
    b++;
};

when() << [](){
    when(cown_a, cown_b) << [](int a, int b){
        std::cout << a + b << std::endl;
    };
};
```

However, this would be erroneous because it breaks the atomicity of the prior behaviour. Currently, there is no way to tell Verona that these two when blocks (the teal and magenta behaviours) should be scheduled together. If this were possible, the behaviours would be executed in parallel, since it can be executed on different cores in a multi-core machine. The next section of this report describes the process in which Verona is extended to enable this behaviour.

The scheduler, inside the BehaviourCore class, performs scheduling for the behaviours based on the Concurrent Owners currently available. The dynamic scheduler works in a 2-phase process. From the run-time's comments: "The first (acquire) phase performs exchange on each cown in same global assumed order. It can only move onto the next cown once the previous behaviour on that cown specifies it has completed its scheduling by marking itself ready, 'set_ready'. The second (release) phase is simply making each slot ready, so that subsequent behaviours can continue scheduling." [3]

After calling when(), the scheduler creates a `When` object. The When class takes as arguments the cowns each wrapped in an "Access" object - this is achieved using a feature of template variadic parameters, in that you can expand the template and each parameter will have the same function applied to it. As such, the when() function turns `when(cown1, cown2, ...)` into `When(Access(cown1), Access(cown2), ...)`. The when object's constructor creates a "cown_tuple" in which each "Access cown" is stored. This object includes a function `operator«()` which accepts a reference to a function - in this case, a lambda function. If there were no arguments provided to the when() function, the function is immediately scheduled (and hence, run.) In all other cases (i.e. there were cowns provided), the function creates an array of requests (the requests being requests to access each cown) and forwards the requests and function to the scheduler via the function schedule_lambda. The schedule_lambda function just sends its parameters to the `Behaviour::schedule` function.

the Behaviour class is a subclass of the BehaviourCore class and implements full when functionality. The Behaviour::schedule function creates a behaviour body from the function provided to the « operator and creates a list of slots (or memory locations) that the cowns can use. It sends the body to the BehaviourCore::schedule function, which actually schedules the behaviour itself. The BehaviourCore::schedule function is the most important; the first step is to get the number of cowns, and the slots (which store each cown), from the body and store them in a variable. It then sorts the slots based on the cown they store by creating an index list. From this, the first phase begins. The first phase attempts to acquire the cowns defined for the behaviour, and waits for the cown to become free if it isn't. It uses the function `Systematic::yield_until()` to do this. Once this has concluded for the behaviour, it is free to begin the second phase: release. This phase sets each slot (i.e. cown) to ready. The body is then resolved, which executes the behaviour.

This project re-implements when(), into `when2()`, which takes in, as a parameter, two vectors of cowns, and two lambda functions.

```
cown_ptr<int> cown_a = make_cown<int>(0);
cown_ptr<int> cown_b = make_cown<int>(0);

when2(cowns(cown_a), cowns(cown_b), []() { fn1 }, []() { fn2 });
```

Upon calling when2(), instead of applying « to form the closure, it instead calls a new lambda_closure() function, which accepts both of the lambda functions. By scheduling this way, both functions and cown sets are passed to the scheduler at the same time, enabling the scheduler to take into consideration both behaviours during scheduling.

That's exactly what this project realises. There is a similar call stack, with new, overloaded function definitions that take double the parameters. The behaviours and their distinct cown lists are transferred to the scheduler through these new overloaded functions to a new BehaviourCore function `BehaviourCore::schedule2()`.

```
void lambdaClosure(F1&& f1, F2&& f2, std::tuple<Access<T1>> cown_tuple1,
```

```
    std::tuple<Access<T2>> cown_tuple2)
    {
      // If no cowns, forward behaviours directly to scheduler.
      ...
      // Otherwise, do the same as << for when, but double the requests arrays
      ...
      // And call schedule_lambda_atomic(size, requests, fn1, size2, requests2, fn2) instead.
    }
```

See Appendix A.5.1

`schedule2()` schedules the behaviours by first creating a map from slot (as in the original schedule function) to the body of the behaviour that needs it. Then, it then creates a list of these mappings from both behaviours and sorts them based on the cown each slot contains. By factoring the cowns of both behaviours, this will be scheduled in the same way as a single behaviour dependent on both cowns.

The general two-phase locking process is logically the same as the original implementation. The main difference is that any reference to the body of the behaviour itself now is taken from the mapping instead of just the body because the resolution of the body is per cown. As such, the correct body needs to be resolved for each cown.

```
    template<TransferOwnership transfer = NoTransfer>
    static void schedule2(BehaviourCore* body1, BehaviourCore* body2)
    {
      // Create unified slot array
      ...
      // Sort the unified slot array
      ...
      // Phase 1 (Acquire) combined
      for (size_t i = 0; i < (count1 + count2); i++)
      {
        // Same as schedule, on unified array
        ...
        prev->set_behaviour(slotsBodyMap[indexes[i]]->body); // gets correct behaviour
        yield();
      }

      // Phase 2 - Release phase; combined cown list.
      // Again, mostly similar to schedule()
      ...
      // resolve both bodies.
      body1->resolve(ec1);
      body2->resolve(ec2);
    }
```

See appendix A.6

This implementation is algorithmically valid and correct due to the Dynamic dependencies of BoC itself. Taking one of the previous examples:

```
void main() {
    auto cown_a = make_cown<int>(0);
    auto cown_b = make_cown<int>(0);
    when(a: cown<int>, b: cown<int>) << [](int a, int b){
        a++;
        b++;
    }
}
```

This behaviour is a perfect example of something that can be re-imagined with respect to the proposed optimisation. The cown list provided to the scheduler above is [a, b]; if this is split into `when2([a], [b], [](){a--;}, [](){b++;})` it is evident that the merged cown list is the same - it is, again, [a, b]. As a result, the scheduler allocates the behaviours in the same way as it would if the implementation were left as it was. The difference between just doing that, and the optimisation, is that there are two behaviours passed to the scheduler. Since they are distinct cown sets, and the optimised scheduler resolves each body against its own specific cowns, the behaviours are added to the scheduler with no shared dependencies. As such, the run-time will execute these concurrently. Given the system has more than 1 core available at that moment, it will execute in parallel.

Using the optimisation, the previous example can be re-written as:

```
cown_ptr<int> cown_a = make_cown<int>(0);
cown_ptr<int> cown_b = make_cown<int>(0);

when2(cowns(cown_a),
cowns(cown_b), [](int a){
    a++;
}, [](int b){
    b++;
});

when() << [](){
    when(cown_a, cown_b) << [](int a, int b){
        std::cout << a + b << std::endl;
    };
};
```

4.1 Benchmarking the Optimisation

4.1.1 Making Graphical Tests

In order to get any useful data from the benchmark, it's vital to turn the raw data into quantitative data that can be evaluated. Doing this involved creating a function that takes in the direct durations of behaviour latency, and turning it into values that retain meaning. As such, the approach taken was to include a program, graphOut.py, that accepts command line values and returns a graph displaying both the behaviour latency and behaviour throughput. See Appendix A.3

4.1.2 Matplotlib

This project chose to use Python to utilise Matplotlib, a powerful graph plotting library for Python. This library is well documented, providing an easy way to create graphs without constructing them manually or attempting to use a less maintained library. In order to produce the graphs, the following algorithms were created:

- read each command line argument - the args given to the benchmark are also passed to `graphOut.py`, in order to organise the file output (and CSV file, see "Making a CSV file".)
- Calculate behaviour latency for 5%, 25%, 50%, 75% and 95% of the behaviours. These are included on the graph as red marks.
- Create a list from 0 to 1 for the percentage of how many behaviours have been executed at each time point. This is particularly easy because the number of behaviours is equal to the number of time points.
- Sort the time points.

- For each second, count how many time points are between each second. This provides us with a rolling throughput - the number of time points between each second describes the number of "behaviours per second" executed.
- Take the final time point and divide that by the number of behaviours. This gives us the average throughput of all behaviours.
- plot both graphs, using matplotlib's subplots() and plot() functions.
- save the figure in a suitable size and resolution with a suitable filename that describes the configuration of the benchmark.

4.1.3 Utilising when2()

Currently, the benchmark is still only testing throughput and latency on the original Verona implementation. As such, a new command line argument is included, –parallel, which enables the project to determine whether the user wishes to test Verona with or without the optimisation.

The benchmark was then designed to conditionally execute based on whether –parallel is included. essentially:

```
if (parallel && fixed_servicetime) {
   for (int i = 0; i < (no_of_whens); i++) {

      when2(*sub_arr1, *sub_arr2,
        [=](auto){
          // behaviour 1 (half service time)
        }, [=](auto){
          // behaviour 2 (half service time)
      );
    }
} else {
    when(*sub_arr) << [](){
        // behaviour (full service time)
    }
}
```

See Appendix A.1

To ensure correctness, the behaviours for when2() needed to have a slightly altered implementation. The alterations made were

- Service time must be fixed to ensure that the benchmark works correctly. The idea is that service time when executing in parallel should be half the service time, but two behaviours. This ensures an equal amount of work is being scheduled, making comparing the original values to the values for the optimisation fair.
- The `sub_array` needs to be split in two, to ensure that the dependencies are valid. Making 2 sub-arrays doesn't guarantee two unique cown lists.
- when2() is called instead, using the two subarrays.
- Two timelists are needed instead of 1, to ensure that the parallel functions aren't being locked out of running in parallel due to synchronisation issues adding to the vector. The two time lists are then combined afterwards.

4.1.4 Making a CSV file

Once the optimisation had been added to the benchmark, the caller script was altered to use nested loops on environment variables to run through each test. This equates to a very large number of tests, and so the graph creator was updated to also add the following values to a CSV file, (Appendix A.7):

- The configuration of the benchmark
- The 5%, 25%, 50%, 75% and 95% behaviour latency values
- Average and Maximum throughput.

These values are the important metrics to enable evaluation of the optimisations made. Utilising a CSV file enables quantitative evaluation via simple calculations, enabling the evaluation of improvements to the run-time, whilst the graphical output of the full benchmark enables these improvements to be visually apparent.

Chapter 5

Evaluation

5.1 The Benchmark

5.1.1 Usability

The benchmark can be used to determine the throughput and latency of each cown in a given program and outputs the result into a graphical format. This output is easy to understand and comparison between graphs clearly displays the effect of different configurations on these metrics tested against. The benchmark's configurability is relatively free, and the user can decide what inputs they wish to provide. There are a few drawbacks to the command line arguments, for example, inputs for the service time distribution values are integer-only, so as to enable the functionality of allowing the distribution generator to be a child class of the parent `Generator` class. This means that `ExponentialGenerator`, for example, requires an integer input, reducing its overall configurability, and means that it is much more difficult to result in exactly equal service times between tests.

Similarly, the benchmark only allows the user to test for behaviour latency and throughput. There are more performance metrics available and could have been included in the benchmark, however behaviour latency and throughput were the focus of the project and as a result, the benchmark was built around this requirement.

The program is relatively simple since the project includes a script that calls the benchmark from the correct place. The script itself, however, does not read command-line arguments. As such, to configure the benchmark, the user must enter the script and input configuration parameters themselves. As this project is primarily intended for use by programmers that wish to incorporate BoC into their program, this is accessible enough for the average user.

5.1.2 Accuracy and Precision

To ensure that the benchmark is Accurate, timings are taken at the same time in every benchmark. To measure latency and throughput, both times are taken at the end of the work of the behaviour. No calls to the vector storing the times are made before this occurs, so as to reduce any possible overheads of the behaviour. In order to determine the service time, a call must be made to the service time generator. This incurs a computational cost and as such reduces the accuracy of our benchmark. Storing values to the time list also requires that the time list be thread safe, and requires the introduction of a locking mechanism, also potentially reducing the accuracy of the benchmark. Without this, however, there is the possibility for multiple behaviours to create a race condition for adding to the list, so this requirement was unavoidable.

Similarly, this allowed us to calculate latency and throughput using the same time values. Creating two separate lists for the latency times and throughput times would have ended up causing even more overheads on each behaviour, further reducing the precision of the benchmark.

5.1.3 Efficiency

C++ is named as such as it shares many similarities with C. It is backwards compatible with C, and the C++ compiler can compile any C program. C++ itself, however, includes many more libraries and many of these include objects that manage memory themselves, rather than requiring the programmer to perform memory management and garbage collection. C-style arrays, for example, must be allocated and freed to avoid memory leaks, yet C++ standard libraries, like vector, do not require this. As a result, C-style arrays are avoided in the benchmark as much as possible, with vectors used in their place. Any variables necessary for the optimised branch of when calls are defined inside the if statement of said branch. This reduces the memory usage of the benchmark since no variable defined goes unused.

The subarrays of the initial cown list are also a worry - if the global list of cowns in the program is large, and the number of behaviours is high, there is a risk of using large amounts of memory storing these arrays. As the array is passed to the behaviour, however, `when()` captures the cowns in the subarray and stores them in a tuple. On the next pass, the subarray is reused, ensuring that there is no storage of multiple arrays. The time taken to schedule the behaviours is minimal, hence, the planning stage of the behaviours is efficient.

The main concern for efficiency, in reality, is the `sub_array_random()` function and the loop it makes ensuring each cown index is unique. Since the generators are random, by design, they repeat values. Due to this, as the size of the requested sub-array approaches the size of the global cown array, execution time (before any behaviours begin executing) tends towards infinity. As a result, the size of the subarray must be below approximately 20% of the global cown array's size. 20% was chosen as a value as a semantically accepted value. There were no issues when 20% and below were chosen, but above this value, the generator would sometimes get stuck in an infinite loop.

5.1.4 Unresolved Issues

Generally, the benchmark executes without any issues. After implementing the benchmark, the optimisation and running benchmark tests, Verona was updated to use a newer version of snmalloc. This newer version caused SystematicTestHarness to throw an error stating that there was non-deallocated memory. This issue persisted until detect_leaks was set to false. It isn't clear why this error is being thrown and would be an area for future work due to the fact that this issue occurred very late in the development cycle.

5.2 The Optimisation

5.3 Determining the benchmark parameters

5.3.1 Real-world simulation

To determine the improvement in performance when the optimisation is put to use in a real-world environment, careful consideration of benchmark parameters is required. A variety of benchmark parameters are available, allowing this to be the case. In terms of realistic usage, it's important to consider:

- Small programs that use little memory and only have a small number of behaviours. This includes:
  - A small number of cowns.
  - A small number of behaviours.
  - Popularity of cowns may be closer to random, or very popular cowns. This would represent two types of programs: ones that have a few variables that are very frequently updated and ones that update objects like arrays.
- Medium-sized programs that are more hardware intensive but aren't so large that they take more than a few seconds to execute. This would be representative of programs that include

GUIs, or programs that make changes to files, for example. This would require more cowns and behaviours.

- Larger programs that may involve hundreds of variables, all being updated by lots of threads. These would be programs more representative of research projects, long-running mathematical algorithms and Operating System Kernel programs.
- Combinations of all of the above. Developers create programs for a multitude of purposes, potentially creating products that may only perform a few tasks but include lots and lots of variables, for example.

5.3.2 Test Variables

To test the implementation, a varied number of configurations are required to ensure that the project equates to real-world conditions. As such, the following configurations are used:

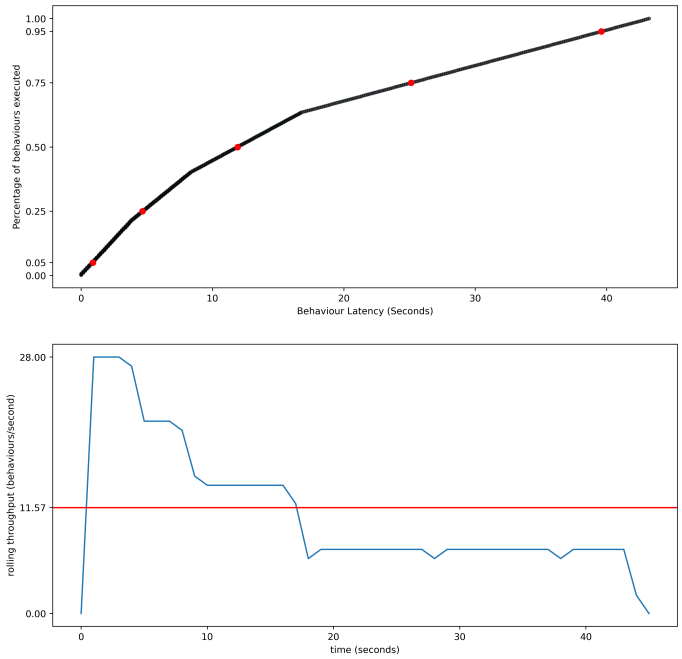
- –cownPop: uniform, zipfian[0.99] and zipfian[2]
- –servTime: fixed[145], bimodal[100:1000:95], exp[7] - these are the closest values possible using our distribution functions to equal average service times. Unless service time is fixed, our benchmark will not run the parallel test (as it would not be a fair test.)
- –totalcowns: 100, 200, 500
- –behaviourCowns: 5%, 10%, 15%
- –whenCount: 50, 100, 200, 500
- –parallel and none. The benchmark should be run for both standard Verona and our optimisation to evaluate the improvements made.

5.4 The results

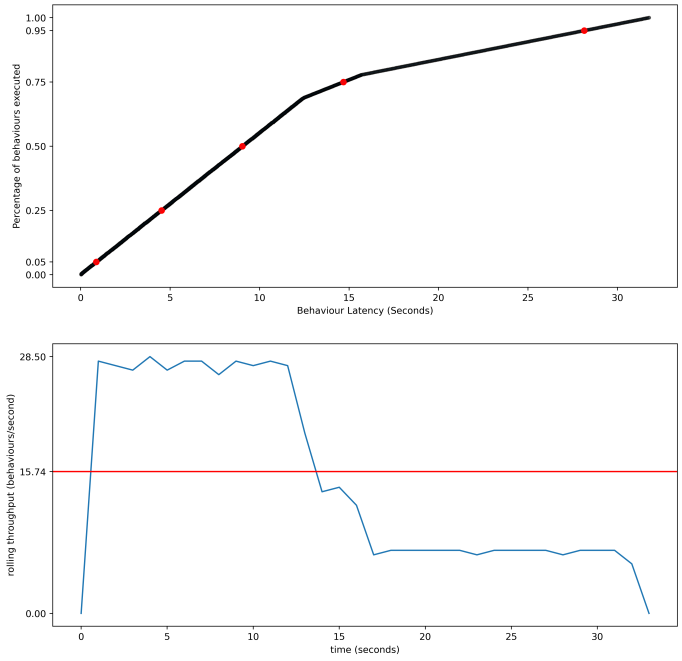
Figures: 5.1, 5.2, 5.3, 5.4, 5.5, 5.6

A subset of the total tests have been used to demonstrate the graphical output of the benchmark, showing the effectiveness of the optimisations in some areas, and the drawbacks of the benchmark in others. Since running the tests over so many configurations resulted in a very large number of results, only a few important results have been picked in order to demonstrate the contributions of the project.

Parallel	5%	25%	50%	75%	95%	Max T	Avg T
True	0.873024	4.515739	9.037366	14.703151	28.165464	28.5	15.73072324
False	0.890339	4.670613	11.906196	25.109938	39.60106	28	11.56749881



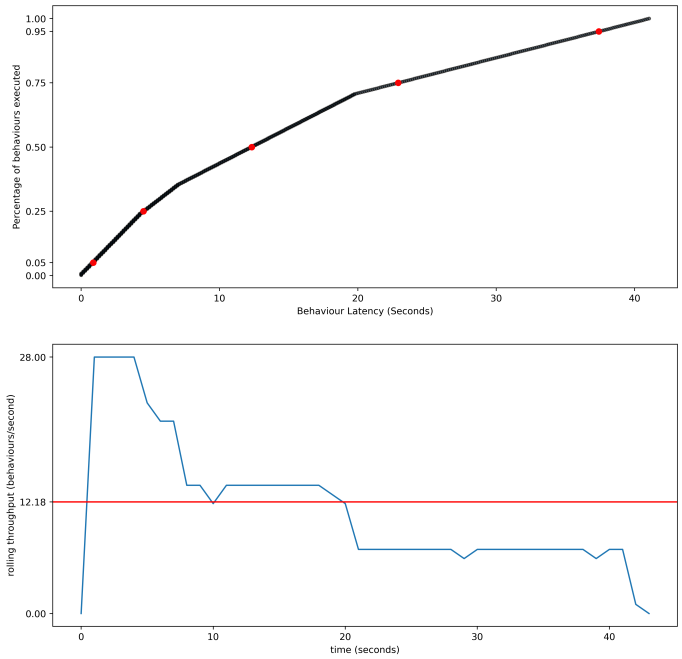
(a) Standard Verona



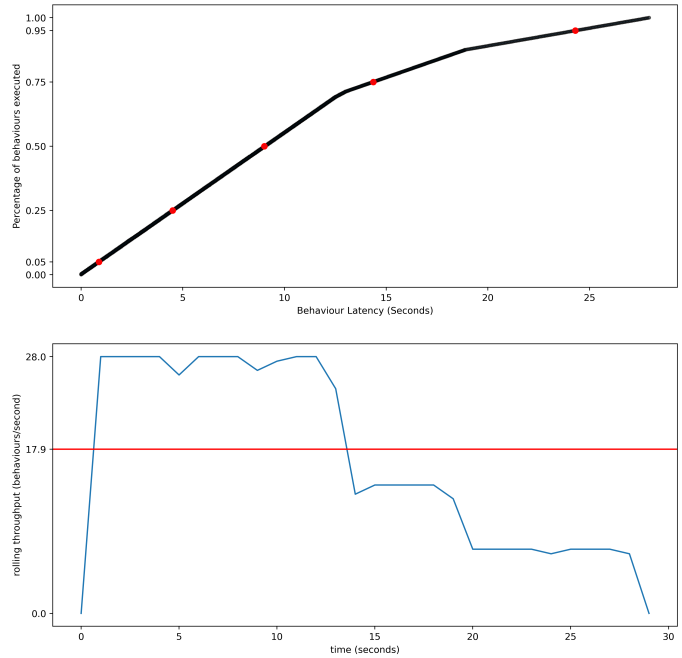
(b) Optimised Verona

Figure 5.1: 500 cowns (5% per behaviour), 500 behaviours, zipfian[2], 145ms fixed service time

Parallel	5%	25%	50%	75%	95%	Max T	Avg T
True	0.880435	4.505211	9.006094	14.365592	24.300989	28	17.90484905
False	0.876216	4.503131	12.331348	22.920769	37.421608	28	12.18114068



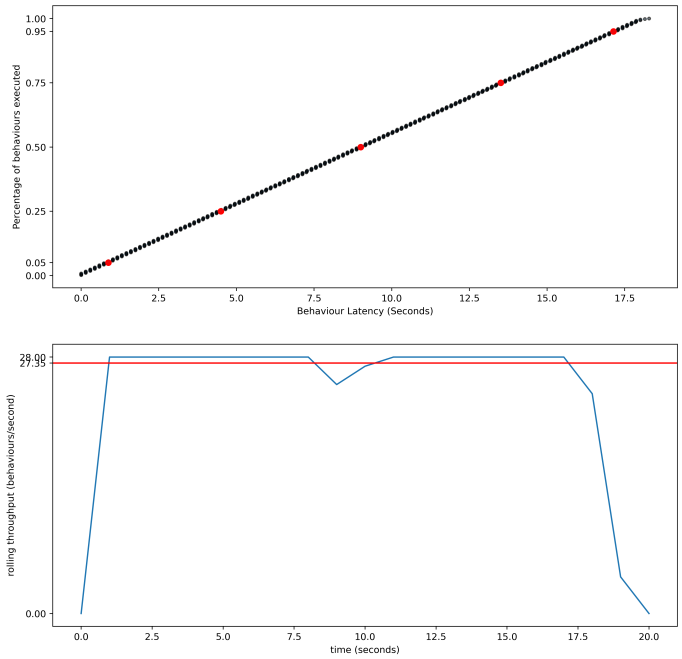
(a) Standard Verona



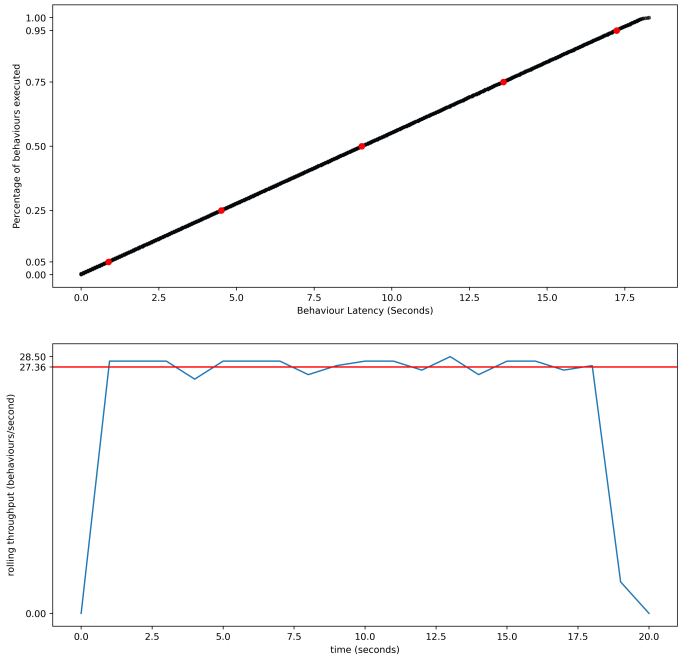
(b) Optimised Verona

Figure 5.2: 100 cowns (10% per behaviour), 500 behaviours, zipfian[2], 145ms fixed service time

Parallel	5%	25%	50%	75%	95%	Max T	Avg T
True	0.880175	4.510043	9.022194	13.584256	17.223261	28	27.45408496
False	0.872655	4.497613	9.011225	13.512837	17.131179	28	27.54631376



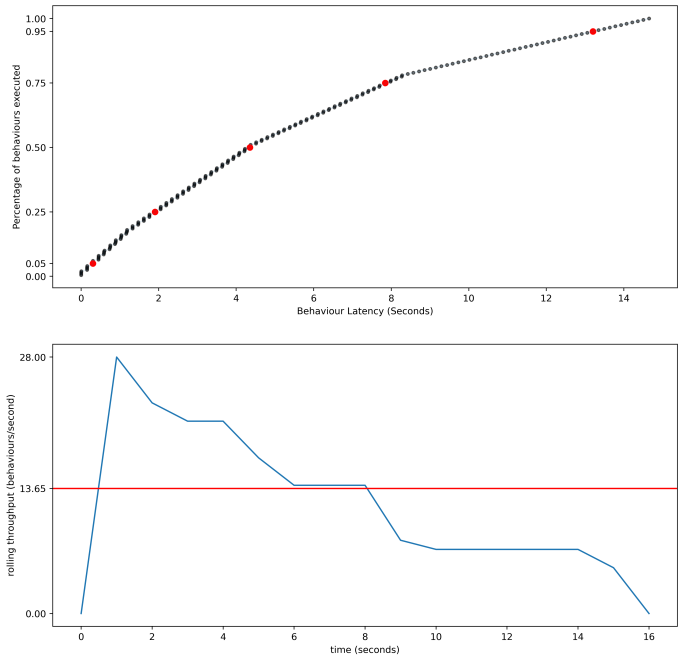
(a) Standard Verona



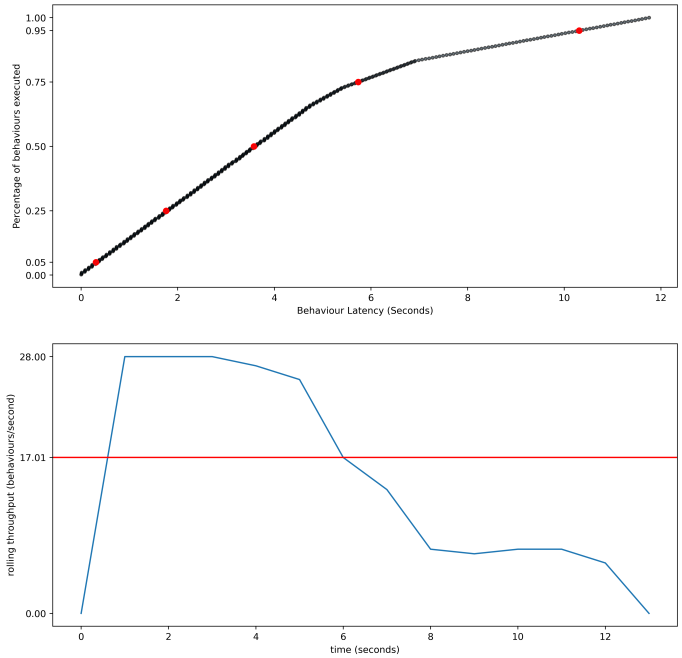
(b) Optimised Verona

Figure 5.3: 500 cowns (15% per behaviour), 500 behaviours, uniform cowns, 145ms fixed service time. Effectively no improvement with uniform distribution

Parallel	5%	25%	50%	75%	95%	Max T	Avg T
True	0.306715	1.761479	3.578907	5.736759	10.31061	28	17.01456634
False	0.30182	1.904426	4.352973	7.844981	13.202452	28	13.65216788



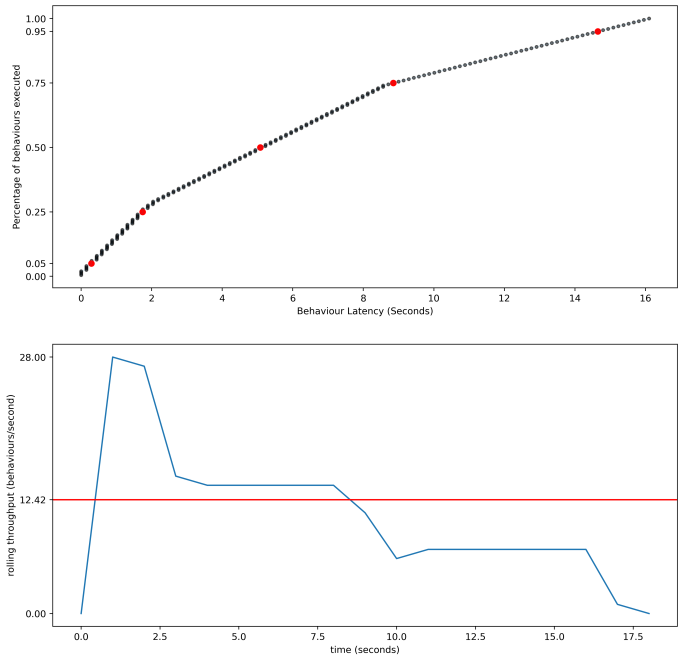
(a) Standard Verona



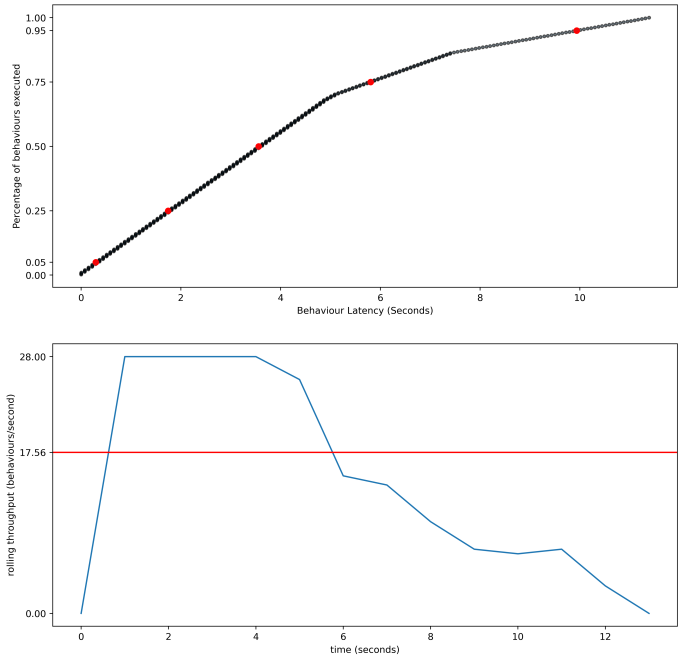
(b) Optimised Verona

Figure 5.4: 200 cowns (15% per behaviour), 200 behaviours, zipfian[2] cowns, 145ms fixed service time.

Parallel	5%	25%	50%	75%	95%	Max T	Avg T
True	0.2954	1.745805	3.561314	5.805578	9.938211	28	17.56325781
False	0.291351	1.743687	5.080825	8.850945	14.650102	28	12.42224066



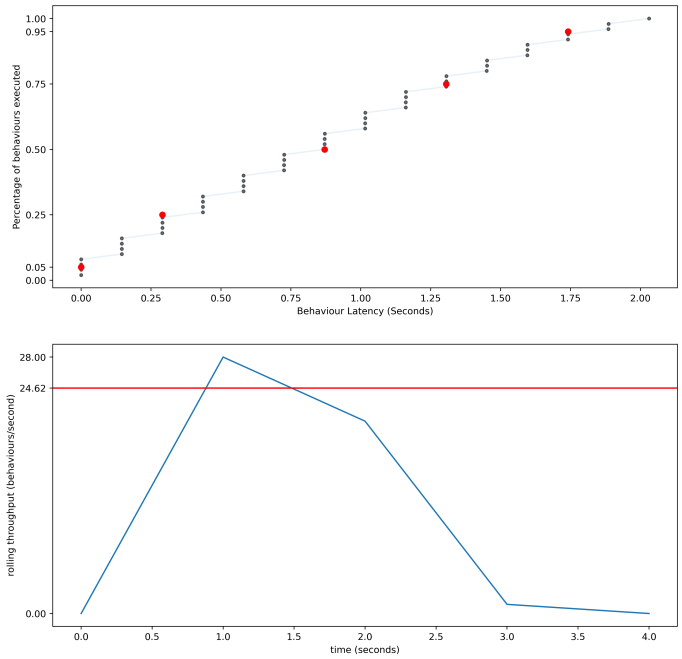
(a) Standard Verona



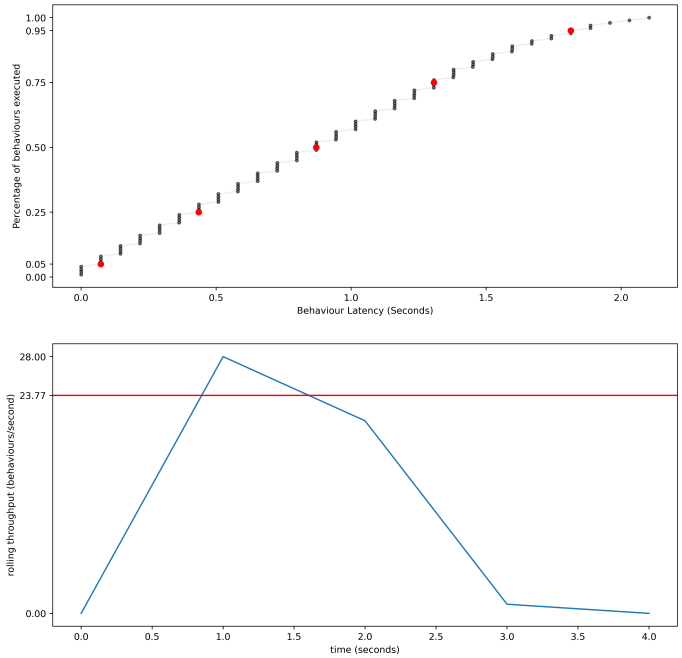
(b) Optimised Verona

Figure 5.5: 100 cowns (15% per behaviour), 200 behaviours, zipfian[2] cowns, 145ms fixed service time. Fewer cowns resulted in a slightly bigger performance improvement.

Parallel	5%	25%	50%	75%	95%	Max T	Avg T
True	0.072833	0.435704	0.870795	1.306558	1.813695	28	23.7722127
False	0.000117	0.290496	0.870557	1.305765	1.741104	28	24.62122704



(a) Standard Verona



(b) Optimised Verona

Figure 5.6: 100 cowns (15% per behaviour), 50 behaviours, zipfian[0.99] cowns, 145ms fixed service time. Reduction in parallelisation (too small a program for the overhead)

The figures provided display an example of the speedup available to the developer under large workloads. Figure 5.3 shows that utilising the optimisation with a uniform distribution of cowns resulted in no discernible improvement to the throughput or latency of the benchmark. This behaviour is expected since the optimisation is designed to improve performance by enabling parallelism on behaviours that otherwise would not have been executed in parallel. A uniform distribution of cowns will mean each behaviour is unlikely to have overlapping cowns resulting in the parallelisation of behaviours without optimisation anyway.

Figures 5.1 and 5.2 display a greatly improved throughput and latency, with 5.1 and its corresponding row from the CSV file exhibiting a **1.36x** speedup in average throughput and **1.40x** speedup in latency for 95% of behaviours.

Figure 5.2 displays an even larger speedup, with a **1.47x** speedup in average throughput and **1.54x** speedup in latency for 95% of behaviours.

Figures 5.4, 5.5 display a group of medium-workload simulations. Figure 5.4 displays how even in a medium workload, the run-time achieves an average throughput speedup of **1.25x** and 95% latency speedup of **1.28x**. The speedup is less significant, as a result of fewer behaviours scheduled and as such, popular cowns are not being acquired as many times, so there is less chance for parallelisation when there wouldn't usually be. 5.5 shows a better performance increase, with an average throughput speedup of **1.41x** and 95% latency speedup of **1.47x** The test was specifically to see the effect of reducing the number of cowns. With fewer total cowns in the program, there are less cowns for the distribution generator to choose from, meaning cowns are picked more frequently. This suggests, along with figure 5.3, that as the cowns picked become more and more popular (and less uniform), the more effective the optimisation becomes.

For very small workloads, very small values were used, with programs only taking a second or two to execute. This led to very few values for throughput to gather, however, the results are shown in figures 5.6. It appears that the overhead for spawning more behaviours resulted in a slowdown, as opposed to a speedup in execution. In particular, the optimisation resulted in **0.96x** the average throughput and **0.95x** the latency for 95% of behaviours.

5.5 Result Analysis

From the results above, it is evident that the optimisation should only be used under certain circumstances. Since the behaviour spawning has a large enough overhead to reduce performance under small workloads, it is vital that when2() should only be used to parallelise when acting under a slightly larger workload. As the workload increases, so do the performance benefits.

In order to ensure optimum effectiveness of the optimisation, it should be implemented in instances where there is a disparity in the popularity of cowns. If they are uniformly random, or there aren't many to choose from, there is very little benefit from using the optimisation.

Implementing multiple service time generators helped to determine the effect of having different workloads for different behaviours, but in order to test the optimisation, it is vital that the behaviours have equal workloads (because the behaviour is essentially split in two.) As a result, the alternative service time distribution generators are practically unused in the benchmark and could be taken out of further versions of the project.

Chapter 6

Conclusion

The objective of the project was to optimise Behavioural Oriented Concurrency for Parallelism. That goal was achieved via the optimisation of atomic scheduling and parallel execution and verified using the implemented benchmark.

This project introduces a new benchmark for the Verona run-time, enabling users to verify the performance of updates to the Verona run-time as it is being improved upon with further research. The project also introduces a new way to formally define separate behaviours that should be executed in parallel, but act atomically.

This optimisation saw a 1.54x increase in performance under an ideal workload for the optimisation, but under very short execution times saw a decrease in performance as a result of having to spawn double the behaviours. This decrease in performance was itself very minimal, executing at 0.95x the original efficiency, with the benefits clearly outweighing the drawbacks. Similarly, it hasn't affected the original implementation in any way, such that smaller programs can use the original implementation and still access the performance of unoptimised Verona for small tasks.

6.1 Future Work

- more optimisations

- Yielding Behaviours: Another way to optimise BoC for parallelism would be to enable behaviours to be yielded during execution, to allow for other behaviours to begin their execution cycle. This optimisation would be to reduce latency, which would dramatically increase if behaviours could stop execution and allow others, that aren't dependent on the acquired cowns, to begin.
- Random-Scheduling: Include the ability to add randomised, or delayed scheduling into the benchmark. This would be simulating the type of behaviour spawning that a Network-attached server may depict, since network requests, without any prior knowledge of the system or when a request may come through, receive requests in a pseudo-random fashion.
- Incorporate Redundancy measurements into the benchmark. How much more computation is needed when parallelism is introduced? This could help identify when behaviours should be passed back to Verona's standard scheduling implementation, avoiding the large overhead slowing down smaller programs.
- Debugging snmalloc to pinpoint the issue with the SystematicTestHarness, ensuring that the benchmark executes and returns correctly every time.

6.2 Ethical Considerations

As the project heavily revolves around developing optimisations for a project already developed by a large corporation, any ethical problems possible have already been addressed. The project doesn't

involve any large data sets or any sensitive information and as such data protection considerations are not needed.

Ethically and morally, the project is safe. There is no real way that the project can be used for harm, other than the ways in which C++ itself could be used for harm. Legally, our project has no issues due to the fact that Microsoft has publicly released Verona, the C++ runtime built upon, under the MIT license. This gives us permission to alter, update, contribute and release the project and provides us relative creative freedom when it comes to the project.

The only true way in which this project could be potentially unsafe is due to the fact that it hasn't been formally tested, along with Verona and as such it is not recommended that vital systems include these optimisations, as it is not formally verified and may crash potentially damaging technology, or the people that technology relies upon.

6.3 Final Remarks

Behavioural Oriented Concurrency is a very new, emerging paradigm and a subset of concurrency that was a pleasure to work upon. This paper hopefully will introduce this new concept to more people, and may even convince some people to see what they can do with it. Microsoft has truly created something wonderful, and it was enriching to be a part of the development of something so new and innovative.

This paper is not intended to intimidate people from standard forms of concurrency. Threads and Locks are still a very good way to implement concurrency into programs and improve the performance of any project. Behavioural Oriented Concurrency is a step in the right direction, however. It is far simpler to understand, whilst simultaneously being more efficient with resources than its counterparts.

Appendix A

Relevant Code

Note: Include and Import commands have been omitted.

A.1 lotsOfCowns.py

```
using namespace verona::cpp;
void sub_array_random(std::vector<cown_ptr<cown>>* arr, std::vector<cown_ptr<cown>>*
sub_arr, int count, bool uniform, double zipfConstant = 0.99) {
  Generator *gen = new Generator;
  std::set<int> indexes;
  if (uniform) {
    gen = new UniformGenerator(0, int(arr->size()) - 1);
  } else {
    gen = new ZipfianGenerator(0, int(arr->size() - 1), zipfConstant);
  }
  for (int i = 0; i < count; i++) {
    int index = gen->nextValue();
    while (!indexes.emplace(index).second) {
      index = gen->nextValue();
    }
    cown_ptr<cown> ptr = arr->at(index);
    sub_arr->emplace_back(ptr);
  }

}

double parseZipfInput(char *input) {
  // input[0-7] = "zipfian["
  // input[8-n] = "number"
  // input[n+1] = "]"
  // extract 8-n and turn into double. Need to find n+1 = >;
  int n = 8;
  while (input[n] != ']') {
    n++;
  }

  char zipfNo[n-8];
  for (int i = 0; i < n-8; i++) {
    zipfNo[i] = input[i+8];
  }

  char *ptr;

  return strtod(zipfNo, &ptr);
```

```
}

void parseServiceTime(char *input, bool fixed_servicetime, int *serviceTime1,
int *serviceTime2, int *serviceTime3) {
  // Fixed or exponential take 1 input
  if (fixed_servicetime || input[0] == 'e' || input[0] == 'E') {
    // "fixed[" is length 6, so index 6 is first value of time;
    std::string s = input;
    // remove naming
    s.erase(0, s.find('[') + 1);
    std::string serviceTime = s.substr(0, s.find(']'));
    *serviceTime1 = atoi(serviceTime.c_str());
  } else {
    // Bimodal takes 3 inputs
    std::string s = input;
    s.erase(0, s.find('[') + 1);

    std::string low_time = s.substr(0, s.find(':'));
    s.erase(0, s.find(':') + 1);
    std::string high_time = s.substr(0, s.find(':'));
    s.erase(0, s.find(':') + 1);
    std::string percentage = s.substr(0, s.find(']'));

    *serviceTime1 = atoi(low_time.c_str());
    *serviceTime2 = atoi(high_time.c_str());
    *serviceTime3 = atoi(percentage.c_str());
  }
}

/**
 * Spins for the time given via the 'seconds' parameter.
*/
auto spin(double seconds) {
  high_resolution_clock::time_point t1 = high_resolution_clock::now();
  duration<double> timespan = duration_cast<duration<double>>(t1 - t1);

  while (timespan.count() < seconds) {
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    timespan = duration_cast<duration<double>>(t2 - t1);
  }

}



/**
 * Function to create a printable list of doubles.
 * This allows us to use as input for argv inside python script (which gives us graph)
*/
std::string printTlist(std::vector<double> *timeList, int length) {

  std::string ret = "";

  for (int i = 0; i < length; i++) {
    ret = ret + std::to_string(timeList->at(i)) + " ";
  }
  return ret;
```

```
}

std::string printParallelTlist(std::vector<double> *timeList1, std::vector<double>
*timeList2) {

  std::vector<double> *mergedList = new std::vector<double>;

  for (int i = 0; i < timeList1->size(); i++) {
    mergedList->emplace_back(timeList1->at(i));
  }
  for (int i = 0; i < timeList2->size(); i++) {
    mergedList->emplace_back(timeList2->at(i));
  }

  // std::cout << mergedList->size() << std::endl;
  std::sort(mergedList->begin(), mergedList->end());

  std::string ret = "";

  for (size_t i = 0; i < mergedList->size(); i++) {
    ret = ret + std::to_string(mergedList->at(i)) + " ";
  }
  return ret;

}

using namespace verona::cpp;
using namespace verona::rt;
void test_body(int argc, char** argv)
{
  bool parallel = false;
  bool uniform_cowns = false;
  double cownConstant = double(2);
  bool fixed_servicetime = true;
  bool expo_servicetime = false;
  int serviceTime1 = 100;
  int serviceTime2 = 1000;
  int serviceTime3 = 95;
  int no_of_cowns = int(1000);
  int sub_arr_percentage = int(5);
  int no_of_whens = int(1000);
  std::string argString = "";


  for (int i = 0; i < argc; i++) {
    if (i > 0) {
      argString = argString + argv[i] + " ";
    }
    if (strcmp(argv[i], "--parallel") == 0) {
      parallel = true;
      std::cout << "Atomic Schedule, parallel execution" << std::endl;
    }
    if (strcmp(argv[i], "--cownPop") == 0) {
      if (strcmp(argv[i+1], "uniform") == 0) {
        uniform_cowns = true;
        std::cout << "Uniform Cown Population" << std::endl;
      } else {
        cownConstant = parseZipfInput(argv[i+1]);
```

```
      std::cout << "Zipfian cown population with constant:" << std::to_string(cownConstant) << s
    }
  }
  if (strcmp(argv[i], "--totalCowns") == 0) {
    std::cout << argv[i+1] << " cowns" << std::endl;
    no_of_cowns = atoi(argv[i+1]);
  }
  if (strcmp(argv[i], "--behaviourCowns") == 0) {
    std::cout << argv[i+1] << " cowns" << std::endl;
    sub_arr_percentage = atoi(argv[i+1]);
  }

  if (strcmp(argv[i], "--whenCount") == 0) {
    std::cout << argv[i+1] << " when blocks" << std::endl;
    no_of_whens = atoi(argv[i+1]);
  }
  if (strcmp(argv[i], "--servTime") == 0) {
    // Fixed, bimodal, e.g. fixed[10] (input in milliseconds)

    if (argv[i+1][0] == 'f') {
      std::cout << argv[i+1] << std::endl;
      fixed_servicetime = true;
      std::cout << "Fixed Service Time" << std::endl;
    } else {
      fixed_servicetime = false;
      if (argv[i+1][0] == 'b') {
        expo_servicetime = false;
        std::cout << "Bimodal Service Time" << std::endl;
      } else {
        expo_servicetime = true;
        std::cout << "Exponential Service Time" << std::endl;
      }
    }
    parseServiceTime(argv[i+1], fixed_servicetime,
    &serviceTime1, &serviceTime2, &serviceTime3);
  }
}

int sub_arr_size = (double)no_of_cowns * ((double)sub_arr_percentage / (double)100);
std::vector<cown_ptr<cown>> *cowns = new std::vector<cown_ptr<cown>>;
for (int i = 0; i < no_of_cowns; i++) {
  cowns->emplace_back(make_cown<cown>());

}

/**
 * Created implementations for "lists" of cowns to be passed.
 * When accepts a variable argument count, so we can provide a pointer to an array
 instead,
 * which provides us with our arguments.
*/

// TODO: move into for loop, and make timeList cumulative (during post-execution).
high_resolution_clock::time_point t1;

pthread_mutex_t lock;

pthread_mutex_init(&lock, NULL);
```

```
// UniformGenerator *gen = new UniformGenerator(0, no_of_cowns);
// ScrambledZipfianGenerator *gen = new ScrambledZipfianGenerator(0, no_of_cowns);

Generator *timeDist;

if (fixed_servicetime) {
  timeDist = new FixedGenerator(serviceTime1);
} else {
  if (expo_servicetime) {
    timeDist = new ExponentialGenerator(serviceTime1);
  } else {
    timeDist = new BimodalGenerator(serviceTime1, serviceTime2, serviceTime3);
  }
}



if (parallel && fixed_servicetime) {

  std::vector<double> *timeList1 = new std::vector<double>;
  std::vector<double> *timeList2 = new std::vector<double>;

  bool *finished = new bool(false);

  for (int i = 0; i < (no_of_whens); i++) {

    t1 = high_resolution_clock::now();

    std::vector<cown_ptr<cown>>* sub_arr = new std::vector<cown_ptr<cown>>;
    std::vector<cown_ptr<cown>>* sub_arr1 = new std::vector<cown_ptr<cown>>();
    std::vector<cown_ptr<cown>>* sub_arr2 = new std::vector<cown_ptr<cown>>();

    sub_array_random(cowns, sub_arr, sub_arr_size, uniform_cowns, cownConstant);

    // std::cout << sub_arr_size << std::endl;
    // std::cout << sub_arr->size() << std::endl;

    for (int i = 0; i < sub_arr->size() / (size_t)2; i++) {
      sub_arr1->push_back(sub_arr->at(i));
    }
    for (int i = sub_arr->size() / (size_t)2; i < sub_arr->size(); i++) {
      sub_arr2->push_back(sub_arr->at(i));
    }

    // std::cout << "sub arr 1: " << sub_arr1->size() << std::endl;
    // std::cout << "sub arr 2: " << sub_arr2->size() << std::endl;

    when2(*sub_arr1, *sub_arr2,
      [=](auto){
        double time = double(timeDist->nextValue()) / (double)1000;

        // double time = 0.1;

        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        spin(time / double(2));

        duration<double> total = duration_cast<duration<double>>(t2 - t1);
```

```
        std::cout << "(1): " << timeList1->size() << " of " << no_of_whens
        << "\t" << total.count() << " seconds" << std::endl;
        // pthread_mutex_lock(&lock);
        timeList1->emplace_back(total.count());
        if (timeList1->size() == size_t(no_of_whens)) {
          if (*finished) {
            std::string bashCall = "python3 /Users/ryanward/Documents/git_repos
            /verona-rt/graphOut.py --csv " + argString +
            printParallelTlist(timeList1, timeList2);
            // pthread_mutex_unlock(&lock);
            system(bashCall.c_str());
          } else {
            *finished = true;
            // pthread_mutex_unlock(&lock);
          }
        }

      }, [=](auto){
        double time = double(timeDist->nextValue()) / (double)1000;

        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        spin(time / double(2));
        duration<double> total = duration_cast<duration<double>>(t2 - t1);
        std::cout << "(2): " << timeList2->size() << " of " << no_of_whens
        << "\t" << total.count() << " seconds" << std::endl;
        // pthread_mutex_lock(&lock);
        timeList2->emplace_back(total.count());
        if (timeList2->size() == size_t(no_of_whens)) {
          if (*finished) {
            std::string bashCall = "python3 /Users/ryanward/Documents/git_repos
            /verona-rt/graphOut.py --csv " + argString +
            printParallelTlist(timeList1, timeList2);
            // pthread_mutex_unlock(&lock);
            system(bashCall.c_str());

          } else {
            *finished = true;
            // pthread_mutex_unlock(&lock);
          }
        }

      }
    );
  }
} else {

  std::vector<double>* timeList = new std::vector<double>;

  for (int i = 0; i < no_of_whens; i++) {

    std::vector<cown_ptr<cown>>* sub_arr = new std::vector<cown_ptr<cown>>;

    sub_array_random(cowns, sub_arr, sub_arr_size, uniform_cowns, cownConstant);

    t1 = high_resolution_clock::now();

    if (parallel) {
```

```
        std::cout << "Cannot be parallel - not fixed service time" << std::endl;
        return;
      }
      when(*sub_arr->data()) << [=, &lock](auto){

        double time = double(timeDist->nextValue()) / (double)1000;
        std::cout << time << " spin " << std::endl;
        // double time = 0.1;
        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        spin(time);

        duration<double> total = duration_cast<duration<double>>(t2 - t1);

        pthread_mutex_lock(&lock);
        std::cout << timeList->size() << " of " << no_of_whens << "\t" << total.count()
        << " seconds" << std::endl;
        timeList->emplace_back(total.count());
        if (timeList->size() == size_t(no_of_whens)) {
          std::string bashCall = "python3 /Users/ryanward/Documents/git_repos/verona-
          rt/graphOut.py --csv " + argString + printTlist(timeList, timeList->size());
          pthread_mutex_unlock(&lock);
          system(bashCall.c_str());
        } else {
          pthread_mutex_unlock(&lock);
        }
      };
    }
  }
}

/**
 * Function Main body. Purely for calling test_body. Uses Systematic testing.
*/
int main(int argc, char** argv) {

  SystematicTestHarness harness(argc, argv);

  // An update for snmalloc was causing a debug allocation error,
  // so this needs to be disabled.
  harness.detect_leaks = false;

  harness.run(test_body, argc, argv);

  // test_body(argc, argv);

  return 0;
}
```

A.2 zipfDist.h

```
// Note that this is not the whole file; More Generators can be found in the repository.
class cown
{
public:

  ~cown()
  {
```

```
    std::cout << "cown" << std::endl;
  }
};

class Generator
{
public:
    virtual int nextValue() {};
    void setLastValue(long lastValue) {};
};

class ZipfianGenerator : public Generator
{
public:
    int items;
    int base;
    double zipfConstant;
    double alpha, zetan, eta, theta, zeta2Theta;
    int countForZeta;
    bool allowItemDecrease = false;
    constexpr static double ZIPF_CONST = 0.99;
    int lastValue;

    void setLastValue(int lastValue) {
        this->lastValue = lastValue;
    };

    ZipfianGenerator(int items) : ZipfianGenerator(0, items - 1) {}

    ZipfianGenerator(int min, int max) : ZipfianGenerator(min, max, ZIPF_CONST) {}

    ZipfianGenerator(int items, double zipfConstant) :
      ZipfianGenerator(0, items - 1, zipfConstant) {}

    ZipfianGenerator(int min, int max, double zipfConstant) : ZipfianGenerator(min,
      max, zipfConstant, zetaStatic(max - min + 1, zipfConstant)) {}

    ZipfianGenerator(int min, int max, double zipfConstant, double zetan) {
        items = max - min + 1;
        base = min;
        this->zipfConstant = zipfConstant;
        theta = zipfConstant;
        zeta2Theta = zeta(2, theta);
        alpha = 1 / (1 - theta);
        this->zetan = zetan;
        countForZeta = items;

        eta = (double)(1.0 - pow(2.0 / items, 1.0 - theta)) /
          (1.0 - zeta2Theta / zetan);

        nextValue();

    };

    double zeta(int n, double thetaVal) {
        countForZeta = n;
        return zetaStatic(n, thetaVal);
    };
```

```
     double zeta(int st, int n, double theta, double initSum) {
        countForZeta = n;
        return zetaStatic(st, n, theta, initSum);
    };

    static double zetaStatic(int n, double theta) {
        return zetaStatic(0, n, theta, 0);
    };

    static double zetaStatic(int st, int n, double theta, double initSum) {
        double sum = initSum;

        for (int i = st; i < n; i++) {
            sum += 1 / pow(i + 1, theta);
        }

        return sum;
    };


    int nextInt(int itemCount) {
        if (itemCount != countForZeta) {
            if (itemCount > countForZeta) {
                zetan = zeta(countForZeta, items, theta, zetan);
                eta = (1 - pow(2 / items, 1 - theta)) / (1 - zeta2Theta / zetan);
            } else {
                if ((itemCount < countForZeta) && (allowItemDecrease)) {
                    std::cout << "WARNING: RECOMPUTING ZIPF. SLOW..." << std::endl;

                    zetan = zeta(itemCount, theta);
                    eta = (1 - pow(2 / items, 1 - theta)) / (1 - zeta2Theta / zetan);
                }
            }
        }


        double u = (double)((std::rand() % 1000) / 1000.0);
        double uz = u * zetan;


        int ret = base + (int)(itemCount * pow(eta * u - eta + 1, alpha));

        setLastValue(ret);
        return ret;

    };

    int nextValue() override {
        return nextInt(items);
    };
};
```

A.3 graphOut.py

```
values = []
```

```
parallel = False
save_to_csv = False

totalCowns = 0
cownDist = ""
behaviourCowns = 0
whenCount = 0
serviceTime = ""

for i in range(13):
    if (sys.argv[i] == "--csv"):
        save_to_csv = True
    elif (sys.argv[i] == "--cownPop"):
        cownDist = sys.argv[i+1]
    elif (sys.argv[i] == "--servTime"):
        serviceTime = sys.argv[i+1]
    elif (sys.argv[i] == "--totalCowns"):
        totalCowns = sys.argv[i+1]
    elif (sys.argv[i] == "--behaviourCowns"):
        behaviourCowns = sys.argv[i+1]
    elif (sys.argv[i] == "--whenCount"):
        whenCount = sys.argv[i+1]
    elif (sys.argv[i] == "--parallel"):
        parallel = True

if save_to_csv:
    timeVals = sys.argv[13:]
else:
    timeVals = sys.argv[1:]

print(timeVals)

for i in timeVals:
    values.append((float(i)))

x = [(i+1) / len(values) for i in range(len(values))]
values = sorted(values)

throughputList = [0] * (math.ceil(max(values)) + 2)
currentTime = 1
for i in values:
    if i >= currentTime:
        currentTime += 1
    if parallel:
        throughputList[currentTime] += 0.5
    else:
        throughputList[currentTime] += 1

avgThroughput = len(values) / values[-1]
maxThroughput = max(throughputList)

if parallel:
    avgThroughput /= 2


total_count = len(values)
_5 = values[int((total_count / 100) * 5 - 1)]
_25 = values[int((total_count / 100) * 25 - 1)]
```

```
_50 = values[int((total_count / 100) * 50 - 1)]
_75 = values[int((total_count / 100) * 75 - 1)]
_95 = values[int((total_count / 100) * 95 - 1)]

data = {
    "cowns": [totalCowns],
    "cownDist": [cownDist],
    "behaviourCowns": [behaviourCowns],
    "whenCount": [whenCount],
    "serviceTime": [serviceTime],
    "parallel": [parallel],
    "5%": [_5],
    "25%": [_25],
    "50%": [_50],
    "75%": [_75],
    "95%": [_95],
    "max": [maxThroughput],
    "avg": [avgThroughput]
}

df = pd.DataFrame(data)
print(df)

df.to_csv(path_or_buf="/Users/ryanward/Documents/git_repos/verona-rt/data.csv",
  mode="a", index=False, header=False)

fig, (plot, plot2) = plt.subplots(2)

plot2.plot([i for i in range(len(throughputList))], throughputList)
plot2.axhline(y = avgThroughput, color="r")


plt.rcParams.update({"font.size": 6})

plt.sca(plot2)
plt.yticks([0, max(throughputList), avgThroughput])
plt.ylabel("rolling throughput (behaviours/second)")
plt.xlabel("time (seconds)")


plot.scatter(values, x, facecolors='black',alpha=0.55, s=10)
plot.plot(values, x, alpha=0.1)

plt.sca(plot)
plt.yticks([0, 0.05, 0.25, 0.50, 0.75, 0.95, 1])
plt.xticks()
plt.ylabel("Percentage of behaviours executed")
plt.xlabel("Behaviour Latency (Seconds)")

plot.scatter(values[int((total_count / 100) * 5 - 1)], 0.05, color="red")
plot.plot([values[int((total_count / 100) * 5 - 1)] for i in range(100)],
  [0.04 for i in range(100)], alpha=0.3)

plot.scatter(values[int((total_count / 100) * 25 - 1)], 0.25, color="red")
plot.plot([values[int((total_count / 100) * 25 - 1)] for i in range(100)],
  [0.24 for i in range(100)], alpha=0.3)
```

```
plot.scatter(values[int((total_count / 100) * 50 - 1)], 0.50, color="red")
plot.plot([values[int((total_count / 100) * 50 - 1)] for i in range(100)],
  [0.49 for i in range(100)], alpha=0.3)

plot.scatter(values[int((total_count / 100) * 75 - 1)], 0.75, color="red")
plot.plot([values[int((total_count / 100) * 75 - 1)] for i in range(100)],
  [0.74 for i in range(100)], alpha=0.3)

plot.scatter(values[int((total_count / 100) * 95 - 1)], 0.95, color="red")
plot.plot([values[int((total_count / 100) * 95 - 1)] for i in range(100)],
  [0.94 for i in range(100)], alpha=0.3)

fname = "/Users/ryanward/Documents/git_repos/verona-rt/figures/"


fname += "cowns_" + str(totalCowns) + "__cownDist_" + str(cownDist) +
"__behaviourCount_" + str(behaviourCowns) + "__whenCount_" + str(whenCount) +
"__serviceTime_" + str(serviceTime) + "__parallel_" + str(parallel) + ".png"

plt.gcf().set_size_inches(12, 12)
plt.savefig(fname, dpi=400)

plt.close
```

A.4 bm-test.sh

```
#!/bin/bash

# Ensure you run from current directory.
cd /Users/ryanward/Documents/git_repos/verona-rt

# Recompiles any changes to the benchmark that have been made, then runs the benchmark
tools that we have created -


# Full compilation:

# rm -rf build_ninja
# mkdir build_ninja
# cd build_ninja
# cmake .. -GNinja -DCMAKE_BUILD_TYPE=Debug
# ninja

# After fully compiled once, only these need to run upon further builds:
cd build_ninja
ninja

# Test if the build failed, and terminate if so.
if [ $? -ne 0 ]
then
    echo build failed
    date
    exit -1
fi



for totalCowns in 100
```

```
do
    for cownPop in "zipfian[2]"
    do
        for behaviourCowns in 15
        do
            for whenCount in 200
            do
                for servTime in "fixed[145]"
                do
                    ./test/bm-con-lotsofcowns --cownPop "$cownPop" --servTime
                    "$servTime" --totalCowns "$totalCowns" --behaviourCowns
                    "$behaviourCowns" --whenCount "$whenCount" --notParallel
                    ./test/bm-con-lotsofcowns --cownPop "$cownPop" --servTime
                    "$servTime" --totalCowns "$totalCowns" --behaviourCowns
                    "$behaviourCowns" --whenCount "$whenCount" --parallel

                done
            done
        done
    done
done

# USAGE: ./test/bm-con-lotsofcowns --cownPop uniform|zipfian[zipfConst] --servTime
fixed[ms]|bimodal[fast|slow|pct]|exp[lambda] --totalCowns no_of_cowns --behaviourCowns
PERCENTAGE_PER_BEHAVIOUR --whenCount no_of_behaviours --parallel|--notParallel
# ./test/bm-con-lotsofcowns --cownPop "$cownPop" --servTime "$servTime" --totalCowns
"$totalCowns" --behaviourCowns "$behaviourCowns" --whenCount "$whenCount" --notParallel
```

A.5 Atomic Scheduling, Parallel Execution

A.5.1 when.h

```
namespace verona::cpp
{
  using namespace verona::rt;

  template<typename T1, typename T2, typename F1, typename F2>
  class When2
  {

    template<class T>
    struct is_read_only : std::false_type
    {};
    template<class T>
    struct is_read_only<Access<const T>> : std::true_type
    {};


    // Friend necessary, same as when's friend - Args2 == Args.
    template<typename Ty1, typename Ty2, typename Fn1, typename Fn2>
    friend auto when2(cown_ptr<Ty1>& cowns1, cown_ptr<Ty2>& cowns2, Fn1&& f1, Fn2&& f2);

    std::tuple<Access<T1>> cown_tuple1;
    std::tuple<Access<T2>> cown_tuple2;

    /**
     * This uses template programming to turn the std::tuple into a C style
```

```
 * stack allocated array.
 * The index template parameter is used to perform each the assignment for
 * each index.
 */
template<size_t index = 0, typename T>
void array_assign(Request* requests, std::tuple<Access<T>> cown_tuple)
{
  if constexpr (index >= sizeof(T))
  {
    return;
  }
  else
  {
    auto p = std::get<index>(cown_tuple);
    if constexpr (is_read_only<decltype(p)>())
      requests[index] = Request::read(p.t);
    else
      requests[index] = Request::write(p.t);
    assert(requests[index].cown() != nullptr);
    array_assign<index + 1>(requests, cown_tuple);
  }
}

/**
 * Applies the closure to schedule the behaviour on the set of cowns.
 * We need to figure out how to make this schedule both function atomically.
 * Probably new implementations of schedule_lambda.
 */
void lambdaClosure(F1&& f1, F2&& f2, std::tuple<Access<T1>> cown_tuple1,
std::tuple<Access<T2>> cown_tuple2)
{
  if constexpr (sizeof(T1) == 0 && sizeof(T2) == 0)
  {
    verona::rt::schedule_lambda_atomic(std::forward<F1>(f1), std::forward<F2>(f2));
  }
  else
  {
    verona::rt::Request requests1[std::tuple_size<decltype(cown_tuple1)>::value];
    verona::rt::Request requests2[std::tuple_size<decltype(cown_tuple2)>::value];
    array_assign(requests1, cown_tuple1);
    array_assign(requests2, cown_tuple2);

    verona::rt::schedule_lambda_atomic(
      sizeof(T1),
      requests1,
      [=, f = std::forward<F1>(f1), cown_tuple = cown_tuple1]() mutable {
        /// Effectively converts ActualCown<T>... to
        /// acquired_cown... .
        auto lift_f = [f =
                        std::forward<F1>(f)](Access<T1> args) mutable {
          f(access_to_acquired<T1>(args));
        };

        std::apply(lift_f, cown_tuple);
      },
      sizeof(T2),
      requests2,
      [=, f = std::forward<F2>(f2), cown_tuple = cown_tuple2]() mutable {
```

```
          /// Effectively converts ActualCown<T>... to
          /// acquired_cown... .
          auto lift_f = [f =
                           std::forward<F2>(f)](Access<T2> args) mutable {
            f(access_to_acquired<T2>(args));
          };

          std::apply(lift_f, cown_tuple);
        });
    }
  }

  /**
   * Converts a single ‘cown_ptr‘ into a ‘acquired_cown‘.
   *
   * Needs to be a separate function for the template parameter to work.
   */
  template<typename C>
  static acquired_cown<C> access_to_acquired(Access<C> c)
  {
    assert(c.t != nullptr);
    return acquired_cown<C>(*c.t);
  }

public:

  When2(Access<T1> *cowns1, Access<T2> *cowns2, F1&& f1, F2&& f2) :
    cown_tuple1(*cowns1),
    cown_tuple2(*cowns2) {
    lambdaClosure(std::move(f1), std::move(f2), cown_tuple1, cown_tuple2);
  }

};


/**
 * Allows us to directly type cown_ptr<T>& cowns into
 std::tuple<Access<cown_ptr<T>&>>.
 * Needed because parameter packs wont work when we want to provide specific lists of
 cowns, for more than one lambda.
 * Cannot implement Atomic-Schedule-Parallel-Execution without this.
*/

template<typename T>
std::vector<Access<T>> create_cown_Access_vector(std::vector<cown_ptr<T>> cowns) {
  std::vector<Access<T>> vector_ret = std::vector<Access<T>>();
  for (int i = 0; i < cowns.size(); i++) {
    cown_ptr<T>& cown = cowns.at(i);
    vector_ret.emplace_back(Access(cown));
  }
  return vector_ret;
}

/**
 * Implements a Verona-like ‘when‘ statement, but accepts two sets of cowns, and two
 * lambda functions. This can be passed for atomic scheduling and parallel execution.
 * To do this, we have implmented a "schedule_lambda_atomic" function, that takes all
 * of the arguments as usual, doubled,
```

```
   * and creates a new function that creates 2 p_threads, and executes the lambdas in
   * parallel. It then joins the threads to ensure that they return after parallel exec.
   *
   * Uses ‘<<‘ to apply the closure.
   *
   * This should really take a type of
   *   ((cown_ptr<A1>& | cown_ptr<A1>&&)...
   * To get the universal reference type to work, we can’t
   * place this constraint on it directly, as it needs to be
   * on a type argument.
   */

  template<typename T, typename... Args>
  auto cowns(Args&&... args) {
    std::vector<T> vector_of_cowns { {args...}};
    return vector_of_cowns;
  }

  template<typename F1, typename F2, typename T1, typename T2>
  auto when2(std::vector<cown_ptr<T1>> args1, std::vector<cown_ptr<T2>> args2, F1&& f1,
  F2&& f2)
  {
    std::vector<Access<T1>> access_cowns1 = create_cown_Access_vector(args1);
    std::vector<Access<T2>> access_cowns2 = create_cown_Access_vector(args2);

    When2 when = When2(access_cowns1.data(), access_cowns2.data(), std::move(f1),
    std::move(f2));

    return when;
  }
}
```

A.6 behaviourcore.h

```
namespace verona::rt
{
  ...

  struct BehaviourCore {
    ...

    class body_cown_class
    {
    public:
      BehaviourCore *body;
      Slot* slot;

      body_cown_class(BehaviourCore* body1, Slot* slot) {
        this->body = body1;
        this->slot = slot;
      }
    };

    template<TransferOwnership transfer = NoTransfer>
    static void schedule2(BehaviourCore* body1, BehaviourCore* body2)
    {
      // Same idea as schedule
      // Create "unified slots" consisting of both body1 slots and body2 slots
```

```
      auto count1 = body1->count;
      auto count2 = body2->count;

      auto slots1 = body1->get_slots();
      auto slots2 = body2->get_slots();

      body_cown_class **slotsBodyMap = (body_cown_class **)malloc(sizeof(body_cown_class) * (count

      for (int i = 0; i < count1; i++) {
        slotsBodyMap[i] = new body_cown_class(body1, &slots1[i]);
      }
      for (int i = 0; i < count2; i++) {
        slotsBodyMap[count1 + i] = new body_cown_class(body2, &slots2[i]);
      }

      StackArray<size_t> indexes(count1 + count2);
      for (size_t i = 0; i < (count1 + count2); i++)
        indexes[i] = i;

      auto compare = [slotsBodyMap](const size_t i, const size_t j) {
#ifdef USE_SYSTEMATIC_TESTING
        return slotsBodyMap[i]->slot->cown->id() > slotsBodyMap[j]->slot->cown->id();
#else
        return slotsBodyMap[i]->slot->cown > slotsBodyMap[j]->slot->cown;
#endif
      };

      // Sort the unified slot array
      if (count1+count2 > 1)
        std::sort(indexes.get(), indexes.get() + (count1 + count2), compare);

      size_t ec1 = 1;
      size_t ec2 = 1;

      // Complete first and second phase based on sorted unified slots

      // Phase 1 (Acquire) combined
      for (size_t i = 0; i < (count1 + count2); i++)
      {
        auto prev = slotsBodyMap[indexes[i]]->slot->cown->last_slot.exchange(
          slotsBodyMap[indexes[i]]->slot, std::memory_order_acq_rel);

        yield();

        if (prev == nullptr)
        {
          Logging::cout() << "Acquired cown: " << slotsBodyMap[indexes[i]]->slot->cown
          << " for behaviour " << slotsBodyMap[indexes[i]]->body << Logging::endl;

          if (slotsBodyMap[indexes[i]]->body == body1) {
            ec1++;
          } else {
            ec2++;
          }

          if (transfer == NoTransfer)
          {
            yield();
```

```
            Cown::acquire(slotsBodyMap[indexes[i]]->slot->cown);
          }
          continue;
        }

        Logging::cout() << "Waiting for cown: " << slotsBodyMap[indexes[i]]->slot->cown
        << " from slot " << prev << " for behaviour " << slotsBodyMap[indexes[i]]->body
        << Logging::endl;

        yield();
        while (prev->is_wait())
        {
          // Wait for the previous behaviour to finish adding to first phase.
          Aal::pause();
          Systematic::yield_until([prev]() { return !prev->is_wait(); });
        }

        if (transfer == YesTransfer)
        {
          Cown::release(ThreadAlloc::get(), slotsBodyMap[indexes[i]]->slot->cown);
        }

        yield();
        prev->set_behaviour(slotsBodyMap[indexes[i]]->body);
        yield();
      }

      // Phase 2 - Release phase; combined cown list.
      Logging::cout() << "Release phase for behaviours " << body1 << " and " << body2
                      << Logging::endl;
      for (size_t i = 0; i < (count1 + count2); i++)
      {
        yield();
        Logging::cout() << "Setting slot " << slotsBodyMap[indexes[i]]->slot << " to
        ready" << Logging::endl;
        slotsBodyMap[i]->slot->set_ready();
      }

      yield();
      body1->resolve(ec1);
      body2->resolve(ec2);
    }
}
```

A.7 Benchmark Results: data.csv

cowns, cownDist, bCowns, whenCount, servTime, parallel	5%	25%	50%	75%	95%	max	avg
100, uniform, 5, 50, fixed[145], False	8.8e-05	0.29027	0.870405	1.305386	1.595802	28	28.723729994640152
100, uniform, 5, 50, fixed[145], True	0.072639	0.435264	0.870623	1.305805	1.668504	28.0	26.513890096743882
100, uniform, 5, 50, bimodal[100:1000:95], False	0.00012	0.200238	0.60038	0.900529	1.100708	40	38.44772685504515
100, uniform, 5, 50, exp[7], False	7.1e-05	0.472182	0.829589	1.433224	1.674742	28	25.269550292975165
100, uniform, 5, 100, fixed[145], False	0.145161	0.870436	1.740773	2.611058	3.336403	28	27.579047756154885
100, uniform, 5, 100, fixed[145], True	0.145266	0.870622	1.741383	2.683913	3.409128	28.0	27.576583619785094
100, uniform, 5, 100, bimodal[100:1000:95], False	0.100158	0.600457	1.400867	2.100888	3.001142	36	31.240383819355603
100, uniform, 5, 100, exp[7], False	0.102187	0.852438	1.814497	2.807099	3.423806	28	27.27445052210117
100, uniform, 5, 200, fixed[145], False	0.290335	1.741162	3.481835	5.367121	6.81758	28	27.576191984345545

100, uniform, 5, 200, fixed[145], True	0.291425	1.742478	3.55546	5.368478	6.818415	28.0	27.574180750409273
100, uniform, 5, 200, bimodal[100:1000:95], False	0.200412	1.500663	3.301656	5.101385	6.30194	36	28.162274971101986
100, uniform, 5, 200, exp[7], False	0.344557	1.793695	3.646498	5.194819	6.849277	36	27.020520193650665
100, uniform, 10, 50, fixed[145], False	0.000144	0.290317	0.870563	1.305648	1.596043	28	28.72059513668418
100, uniform, 10, 50, fixed[145], True	0.072685	0.435438	0.87092	1.305881	1.668878	28.0	28.70893617314474
100, uniform, 10, 50, bimodal[100:1000:95], False	0.000161	0.200319	0.60063	1.000902	1.301214	36	33.32318087089467
100, uniform, 10, 50, exp[7], False	9.9e-05	0.395273	0.802444	1.40058	1.647894	29	27.81675213827374
100, uniform, 10, 100, fixed[145], False	0.145131	0.870661	1.741102	2.611557	3.336673	28	26.514754798176313
100, uniform, 10, 100, fixed[145], True	0.145757	0.871274	1.742373	2.684156	3.410284	28.0	27.572394077890912
100, uniform, 10, 100, bimodal[100:1000:95], False	0.100235	0.600692	1.70084	2.701537	3.401517	35	27.766933247737136
100, uniform, 10, 100, exp[7], False	0.030203	0.754994	1.70044	2.700695	3.329832	29	28.08088755822923
100, uniform, 10, 200, fixed[145], False	0.29074	1.741401	3.481998	5.367298	6.816952	28	27.038141895627906
100, uniform, 10, 200, fixed[145], True	0.292317	1.743552	3.556356	5.369979	6.818998	28.0	27.301640951478568
100, uniform, 10, 200, bimodal[100:1000:95], False	0.200689	1.601004	3.302396	5.101347	6.302729	35	27.01860379473588
100, uniform, 10, 200, exp[7], False	0.393703	1.814668	3.667379	5.215565	6.866974	36	26.131408319587123
100, uniform, 15, 50, fixed[145], False	8.9e-05	0.290378	0.870402	1.305955	1.596227	28	28.714261512265296
100, uniform, 15, 50, fixed[145], True	0.072662	0.435378	0.870924	1.306434	1.669142	28.0	27.56638951418649
100, uniform, 15, 50, bimodal[100:1000:95], False	0.000106	0.200399	0.600407	0.900668	1.10112	40	41.63062148689593
100, uniform, 15, 50, exp[7], False	9.2e-05	0.3953	0.80248	1.400748	1.647812	29	28.588009617006435
100, uniform, 15, 100, fixed[145], False	0.145178	0.870661	1.741187	2.611793	3.337131	28	27.573336803897327
100, uniform, 15, 100, fixed[145], True	0.145521	0.871512	1.742428	2.684765	3.410441	28.0	27.035875796105156
100, uniform, 15, 100, bimodal[100:1000:95], False	0.100215	0.600595	1.700668	2.701412	3.401682	35	27.768321121751313
100, uniform, 15, 100, exp[7], False	0.102307	0.829686	1.781749	2.773884	3.39376	28	26.69243376272562
100, uniform, 15, 200, fixed[145], True	0.292488	1.743897	3.557362	5.370867	6.820784	28.0	27.577420159921463
100, uniform, 15, 200, bimodal[100:1000:95], False	0.200551	1.402219	3.102054	4.901636	6.101489	36	28.162013246002928
100, uniform, 15, 200, exp[7], False	0.312853	1.717551	3.57873	5.125345	6.780183	36	27.09230116723116
100, zipfian[0.99], 5, 50, fixed[145], False	0.000126	0.290368	0.87046	1.305414	1.740775	28	24.622415261957872
100, zipfian[0.99], 5, 50, fixed[145], True	0.072688	0.435505	0.870821	1.30589	1.740582	28.0	26.51422753449501
100, zipfian[0.99], 5, 50, bimodal[100:1000:95], False	0.000107	0.20042	0.600653	1.20068	1.600775	32	26.310582747835163
100, zipfian[0.99], 5, 50, exp[7], False	8.2e-05	0.395199	0.802289	1.400347	1.748655	29	26.282493704028635
100, zipfian[0.99], 5, 100, fixed[145], False	0.145243	0.870618	1.740964	2.611268	3.771155	28	24.623421700227567
100, zipfian[0.99], 5, 100, fixed[145], True	0.145336	0.870936	1.741636	2.683882	3.408594	28.0	26.516674215079927
100, zipfian[0.99], 5, 100, bimodal[100:1000:95], False	0.100262	0.600509	1.200819	2.201174	2.800875	40	27.020870650281573
100, zipfian[0.99], 5, 100, exp[7], False	0.072294	0.802342	1.748913	2.742372	3.812973	29	22.878107047120434
100, zipfian[0.99], 5, 200, fixed[145], False	0.290586	1.740754	3.482082	5.367475	7.542135	28	23.77506977091413
100, zipfian[0.99], 5, 200, fixed[145], True	0.291463	1.742235	3.554715	5.368126	6.889939	28.0	26.26501850173566
100, zipfian[0.99], 5, 200, bimodal[100:1000:95], False	0.200631	1.400638	3.20102	5.501859	7.501788	37	22.217739176184008
100, zipfian[0.99], 5, 200, exp[7], False	0.329798	1.749182	3.609695	5.236586	7.417792	37	23.215974261842295
100, zipfian[0.99], 10, 50, fixed[145], False	9.6e-05	0.290376	0.870658	1.305752	1.740802	28	22.980127245560585
100, zipfian[0.99], 10, 50, fixed[145], True	0.072728	0.435399	0.871106	1.306459	1.740908	28.0	25.53497039730882
100, zipfian[0.99], 10, 50, bimodal[100:1000:95], False	0.000161	0.200475	0.600624	0.900755	2.000637	39	20.826981104096582
100, zipfian[0.99], 10, 50, exp[7], False	0.0001	0.395233	0.802412	1.400585	2.189688	29	18.13100521556496
100, zipfian[0.99], 10, 100, fixed[145], True	0.145555	0.871407	1.742275	2.684633	3.771522	28.0	23.773003947269576
100, zipfian[0.99], 10, 100, bimodal[100:1000:95], False	0.100362	0.701081	1.601122	2.301622	4.101446	32	19.225611590736715
100, zipfian[0.99], 10, 100, exp[7], False	0.102252	0.853053	1.814723	2.806666	3.92859	28	21.04165005253048
100, zipfian[0.99], 10, 200, fixed[145], False	0.291123	1.741097	3.482347	5.367815	8.412526	28	20.27905397807611
100, zipfian[0.99], 10, 200, fixed[145], True	0.292002	1.743905	3.557056	5.370193	7.470208	28.0	23.77345042867097
100, zipfian[0.99], 10, 200, bimodal[100:1000:95], False	0.201149	1.901527	3.20186	5.302189	7.30161	40	23.251063707100773
100, zipfian[0.99], 10, 200, exp[7], False	0.330315	1.787177	3.635491	5.243366	8.434215	38	20.766012516506386
100, zipfian[0.99], 15, 50, fixed[145], False	0.000117	0.290496	0.870557	1.305765	1.741104	28	24.621227043167902
100, zipfian[0.99], 15, 50, fixed[145], True	0.072833	0.435704	0.870795	1.306558	1.813695	28.0	23.772212755598833
100, zipfian[0.99], 15, 50, bimodal[100:1000:95], False	0.000131	0.200531	0.60046	0.900595	1.201038	39	35.697864696525315
100, zipfian[0.99], 15, 50, exp[7], False	0.000157	0.514436	0.821542	1.461966	2.024083	29	19.292713682238205
100, zipfian[0.99], 15, 100, fixed[145], False	0.145277	0.870868	1.740895	2.612054	3.771434	28	24.622651706928394
100, zipfian[0.99], 15, 100, fixed[145], True	0.145438	0.870847	1.742715	2.683762	3.699696	28.0	22.98011140298406

100, zipfian[0.99], 15, 100, bimodal[100:1000:95], False	0.100296	0.601309	1.800805	2.902042	3.601075	34	25.634084719624635
100, zipfian[0.99], 15, 100, exp[7], False	0.072409	0.803041	1.749293	2.742294	3.642772	29	23.057630934486582
100, zipfian[0.99], 15, 200, fixed[145], False	0.290834	1.741989	3.482519	5.368466	6.963176	28	25.53615389516362
100, zipfian[0.99], 15, 200, fixed[145], True	0.292374	1.743836	3.556694	5.370782	7.254555	28.0	22.981254765019543
100, zipfian[0.99], 15, 200, bimodal[100:1000:95], False	0.200902	1.702965	3.601092	5.301261	6.602356	35	23.250579869461944
100, zipfian[0.99], 15, 200, exp[7], False	0.344966	1.79553	3.64732	5.194649	7.308867	36	23.836470371744056
100, zipfian[2], 5, 50, fixed[145], False	0.000126	0.290524	1.015439	1.740743	3.191107	24	13.788561781305688
100, zipfian[2], 5, 50, fixed[145], True	0.072965	0.435605	0.870709	1.450863	2.176041	28.0	20.27460739236515
100, zipfian[2], 5, 50, bimodal[100:1000:95], False	0.000135	0.200549	0.700452	1.20074	2.200967	33	19.991539580449555
100, zipfian[2], 5, 50, exp[7], False	0.000132	0.5394	0.993733	1.978599	3.135977	25	13.49860302957247
100, zipfian[2], 5, 100, fixed[145], False	0.145164	0.870665	1.886062	3.626372	6.526995	28	13.789134079610461
100, zipfian[2], 5, 100, fixed[145], True	0.145313	0.870809	1.741455	3.119007	4.569303	28.0	18.887667486391436
100, zipfian[2], 5, 100, bimodal[100:1000:95], False	0.100238	0.700802	1.501313	4.80148	8.601937	32	10.986596352450011
100, zipfian[2], 5, 100, exp[7], False	0.030558	0.755229	1.971523	4.629573	6.994211	29	12.790407808083511
100, zipfian[2], 5, 200, fixed[145], False	0.291475	1.741333	4.352168	8.558018	14.359143	28	12.650710283616906
100, zipfian[2], 5, 200, bimodal[100:1000:95], False	0.201391	1.501047	4.401711	8.602821	14.403783	37	12.266814751654348
100, zipfian[2], 5, 200, exp[7], False	0.345412	1.918418	4.507059	8.346717	15.038896	28	12.318201059771793
100, zipfian[2], 10, 50, fixed[145], False	0.000153	0.290868	1.015881	1.886188	3.33623	24	13.258136717375503
100, zipfian[2], 10, 50, fixed[145], True	0.073196	0.435404	0.870678	1.378324	2.538862	28.0	17.233671699243477
100, zipfian[2], 10, 50, bimodal[100:1000:95], False	0.000176	0.200859	0.700859	2.200949	3.201002	30	11.360914908110647
100, zipfian[2], 10, 50, exp[7], False	0.000146	0.395789	1.096969	2.176052	3.370024	23	12.696735923128882
100, zipfian[2], 10, 100, fixed[145], False	0.145854	0.870826	2.176824	4.061327	6.672036	28	13.518805943353769
100, zipfian[2], 10, 100, fixed[145], True	0.145866	0.871931	1.742877	2.685433	4.642138	28.0	18.63243168512085
100, zipfian[2], 10, 100, bimodal[100:1000:95], False	0.100856	0.600841	1.501774	3.701197	6.401784	38	12.817342992995705
100, zipfian[2], 10, 100, exp[7], False	0.073241	0.803733	1.910384	4.392799	6.756873	29	13.190880616594525
100, zipfian[2], 10, 200, fixed[145], False	0.292023	1.741188	4.498891	9.284063	15.084401	28	12.09588187839127
100, zipfian[2], 10, 200, fixed[145], True	0.29394	1.745203	3.558253	5.442701	9.574526	28.0	18.14166243663571
100, zipfian[2], 10, 200, bimodal[100:1000:95], False	0.202096	1.703614	3.702078	9.103801	14.004117	31	12.575215507755765
100, zipfian[2], 10, 200, exp[7], False	0.33082	1.788654	4.540564	8.47462	15.162154	28	12.225457485788212
100, zipfian[2], 15, 50, fixed[145], False	0.00074	0.436337	1.306425	2.176397	3.481333	19	12.767256990583892
100, zipfian[2], 15, 50, fixed[145], True	0.072667	0.436099	0.871721	1.451954	2.466636	28.0	17.67442090643621
100, zipfian[2], 15, 50, bimodal[100:1000:95], False	0.000733	0.301295	1.400849	3.201319	5.101303	19	9.257155827740624
100, zipfian[2], 15, 50, exp[7], False	0.000717	0.558482	1.466866	2.929411	4.22739	20	10.42678499260324
100, zipfian[2], 15, 100, fixed[145], False	0.14508	1.015641	2.612671	4.932564	7.832599	24	11.685576656331163
100, zipfian[2], 15, 100, fixed[145], True	0.146043	0.8717	1.74462	2.758957	5.222891	28.0	16.813487645281143
100, zipfian[2], 15, 100, bimodal[100:1000:95], False	0.100092	0.700496	2.30133	5.202543	9.002533	30	10.523613990050343
100, zipfian[2], 15, 100, exp[7], False	0.103043	1.001459	2.531683	4.50446	6.245437	24	14.145510052435991
100, zipfian[2], 15, 200, fixed[145], False	0.291163	1.743426	5.08046	8.850593	14.649702	28	12.422553916834357
100, zipfian[2], 15, 200, fixed[145], True	0.295303	1.745666	3.560907	5.804881	9.93769	28.0	17.563999702114565
100, zipfian[2], 15, 200, bimodal[100:1000:95], False	0.201296	1.805533	4.90416	7.40301	11.903945	32	15.499175753833411
100, zipfian[2], 15, 200, exp[7], False	0.332202	1.7907	5.223559	8.576161	15.262365	28	12.151125984089315
200, uniform, 5, 50, fixed[145], False	8.8e-05	0.290276	0.870348	1.305613	1.595931	28	28.720248694377496
200, uniform, 5, 50, fixed[145], True	0.072637	0.435562	0.870927	1.306293	1.669106	28.0	27.57216600870288
200, uniform, 5, 50, bimodal[100:1000:95], False	0.000125	0.200294	0.600264	1.000922	1.50079	35	29.402753626682646
200, uniform, 5, 50, exp[7], False	0.000108	0.395236	0.802431	1.400534	1.647765	29	28.592554384468066
200, uniform, 5, 100, fixed[145], False	0.145167	0.870622	1.740616	2.611419	3.336659	28	27.578652247811842
200, uniform, 5, 100, fixed[145], True	0.145463	0.870779	1.741687	2.684744	3.4102	28.0	27.566229934885808
200, uniform, 5, 100, bimodal[100:1000:95], False	0.100228	0.600765	1.60075	2.401205	3.201427	35	22.72090054743738
200, uniform, 5, 100, exp[7], False	0.07217	0.802445	1.748692	2.742563	3.363342	29	27.42206785426158
200, uniform, 5, 200, fixed[145], False	0.290375	1.74135	3.481295	5.367712	6.818175	28	27.03535683967493
200, uniform, 5, 200, fixed[145], True	0.291148	1.742703	3.55639	5.369879	6.820309	28.0	27.29869701954191
200, uniform, 5, 200, bimodal[100:1000:95], False	0.200448	1.601776	3.30249	5.101123	6.302919	32	29.839781262467437
200, uniform, 5, 200, exp[7], False	0.393285	1.814932	3.666265	5.215804	6.867865	36	27.44533849159322
200, uniform, 10, 50, fixed[145], False	0.000115	0.290361	0.870507	1.305934	1.596339	28	28.712958847438863
200, uniform, 10, 50, fixed[145], True	0.072756	0.435779	0.871331	1.306902	1.669487	28.0	27.573701747400627

200, uniform, 10, 50, exp[7], False	8.1e-05	0.395325	0.802592	1.400839	1.647984	29	28.58537823600774
200, uniform, 10, 100, fixed[145], False	0.145167	0.870574	1.741536	2.611427	3.337499	28	28.712752740416903
200, uniform, 10, 100, fixed[145], True	0.145871	0.870917	1.743262	2.68612	3.411022	28.0	27.56592597942424
200, uniform, 10, 100, bimodal[100:1000:95], False	0.10018	0.600642	1.401649	2.100789	3.001281	36	32.233276087035
200, uniform, 10, 100, exp[7], False	0.072261	0.822045	1.786534	2.778151	3.392037	29	26.645882878021812
200, uniform, 10, 200, fixed[145], False	0.290475	1.740883	3.483232	5.368239	6.819115	28	27.575853589865982
200, uniform, 10, 200, fixed[145], True	0.291914	1.744267	3.558074	5.372218	6.822747	28.0	27.57031118610236
200, uniform, 10, 200, bimodal[100:1000:95], False	0.200555	1.602292	3.101579	4.90252	6.102595	35	28.971009145713023
200, uniform, 10, 200, exp[7], False	0.329683	1.748577	3.611101	5.157082	6.809481	37	26.346767725841858
200, uniform, 15, 50, fixed[145], False	0.000139	0.290531	0.870925	1.305467	1.596589	28	28.72118905722697
200, uniform, 15, 50, fixed[145], True	0.072803	0.435328	0.871429	1.307048	1.669955	28.0	27.556757275121704
200, uniform, 15, 50, bimodal[100:1000:95], False	0.000131	0.20055	0.600489	0.900638	1.101508	40	41.61146212691163
200, uniform, 15, 50, exp[7], False	0.000115	0.514357	0.853121	1.466965	1.706399	28	21.98614960519471
200, uniform, 15, 100, fixed[145], True	0.14547	0.871481	1.743749	2.686907	3.410447	28.0	27.032360438681145
200, uniform, 15, 100, bimodal[100:1000:95], False	0.100271	0.600974	1.401809	2.101625	3.001543	36	32.229079137859244
200, uniform, 15, 100, exp[7], False	0.072288	0.821304	1.786583	2.778452	3.392417	28	27.50490618764122
200, uniform, 15, 200, fixed[145], False	0.29058	1.742174	3.483809	5.369237	6.820938	28	28.130087590060235
200, uniform, 15, 200, fixed[145], True	0.292292	1.745321	3.559974	5.37404	6.824946	28.0	27.558196365652623
200, uniform, 15, 200, exp[7], False	0.39314	1.815585	3.669462	5.218091	6.870798	36	27.425493504614682
200, zipfian[0.99], 5, 50, fixed[145], False	0.000119	0.29041	0.870587	1.305628	1.595917	28	26.513791679017675
200, zipfian[0.99], 5, 50, fixed[145], True	0.072701	0.435679	0.870816	1.306418	1.740918	28.0	25.53378374927867
200, zipfian[0.99], 5, 50, bimodal[100:1000:95], False	0.000118	0.20038	0.600709	0.900626	1.100987	40	38.43959832157338
200, zipfian[0.99], 5, 50, exp[7], False	0.00012	0.514277	0.821635	1.45068	1.785677	29	19.74892012904734
200, zipfian[0.99], 5, 100, fixed[145], False	0.14528	0.870759	1.741166	2.611465	3.336892	28	26.514262684754694
200, zipfian[0.99], 5, 100, fixed[145], True	0.145388	0.871507	1.742504	2.684224	3.409282	28.0	26.515422695610905
200, zipfian[0.99], 5, 100, bimodal[100:1000:95], False	0.100304	0.800475	1.801093	3.200788	3.800922	29	23.249169016576424
200, zipfian[0.99], 5, 100, exp[7], False	0.03033	0.755453	1.718038	2.710632	3.408496	29	26.804412756863336
200, zipfian[0.99], 5, 200, fixed[145], False	0.290597	1.741956	3.483278	5.367596	6.817347	28	27.037085553585
200, zipfian[0.99], 5, 200, fixed[145], True	0.291198	1.743177	3.557398	5.370458	6.89009	28.0	26.515855089260985
200, zipfian[0.99], 5, 200, bimodal[100:1000:95], False	0.200609	1.801457	3.602298	5.301319	7.20154	35	25.635865601411
200, zipfian[0.99], 5, 200, exp[7], False	0.32978	1.785848	3.635646	5.182683	7.221186	38	25.51890459591644
200, zipfian[0.99], 10, 50, fixed[145], False	0.000104	0.290601	0.871046	1.305595	1.595767	28	26.511851593017624
200, zipfian[0.99], 10, 50, fixed[145], True	0.072835	0.435319	0.871615	1.30647	1.668827	28.0	27.56614634646566
200, zipfian[0.99], 10, 50, bimodal[100:1000:95], False	0.000126	0.200576	0.601158	0.901448	1.900849	37	19.992898522444825
200, zipfian[0.99], 10, 50, exp[7], False	0.000125	0.395517	0.782903	1.369871	1.873811	29	20.067370175152018
200, zipfian[0.99], 10, 100, fixed[145], False	0.145567	0.870637	1.742023	2.611795	3.336023	28	27.579618220860894
200, zipfian[0.99], 10, 100, fixed[145], True	0.145921	0.87152	1.743506	2.684753	3.554347	28.0	24.622009072225463
200, zipfian[0.99], 10, 100, bimodal[100:1000:95], False	0.100481	0.600524	1.501065	2.300873	3.301406	36	26.309129109946376
200, zipfian[0.99], 10, 100, exp[7], False	0.102579	0.82958	1.795389	2.787044	3.537684	28	24.422058100564662
200, zipfian[0.99], 10, 200, fixed[145], False	0.29089	1.743028	3.484352	5.367564	6.81792	28	25.072082236429736
200, zipfian[0.99], 10, 200, fixed[145], True	0.292598	1.74582	3.559985	5.373267	7.038289	28.0	24.1905977194314
200, zipfian[0.99], 10, 200, bimodal[100:1000:95], False	0.200844	1.302624	3.70315	5.70202	7.401127	40	24.38545305306482
200, zipfian[0.99], 10, 200, exp[7], False	0.345355	1.79479	3.648053	5.194976	7.309695	36	24.123429941639394
200, zipfian[0.99], 15, 50, fixed[145], False	0.000231	0.290819	0.871148	1.306251	2.030873	28	21.543311103508717
200, zipfian[0.99], 15, 50, fixed[145], True	0.072934	0.435392	0.871698	1.30677	1.742007	28.0	24.617493387741273
200, zipfian[0.99], 15, 50, bimodal[100:1000:95], False	0.0002	0.200791	0.601195	1.001196	1.600688	35	19.993210305780156
200, zipfian[0.99], 15, 50, exp[7], False	0.000202	0.514705	0.822305	1.462238	2.212992	28	17.979583822977332
200, zipfian[0.99], 15, 100, fixed[145], False	0.145629	0.87027	1.742193	2.612883	4.061507	28	20.892799463807197
200, zipfian[0.99], 15, 100, fixed[145], True	0.146148	0.873377	1.742786	2.68573	3.627868	28.0	23.77273267562106
200, zipfian[0.99], 15, 100, bimodal[100:1000:95], False	0.100639	0.601905	1.702758	3.002492	4.701437	33	19.226188221691025
200, zipfian[0.99], 15, 100, exp[7], False	0.102892	0.830716	1.79381	2.787386	3.728359	28	24.108625752882247
200, zipfian[0.99], 15, 200, fixed[145], False	0.291095	1.743605	3.485281	5.370266	7.977777	28	22.240887269060355
200, zipfian[0.99], 15, 200, fixed[145], True	0.292395	1.74536	3.561175	5.375252	7.256655	28.0	24.189772636698525
200, zipfian[0.99], 15, 200, bimodal[100:1000:95], False	0.201185	1.203674	2.701985	4.20532	7.202229	40	23.80427553353713
200, zipfian[0.99], 15, 200, exp[7], False	0.31335	1.745561	3.597475	5.144728	7.570417	37	24.504080296930642

200, zipfian[2], 5, 50, fixed[145], False	0.000447	0.290926	0.871032	1.740821	3.046161	26	14.36290463893094
200, zipfian[2], 5, 50, fixed[145], True	0.07286	0.435318	0.871384	1.306221	2.03106	28.0	20.889735619506
200, zipfian[2], 5, 50, exp[7], False	0.000441	0.395892	0.847955	2.077931	3.323096	26	12.849843411808184
200, zipfian[2], 5, 100, fixed[145], False	0.145616	0.871205	2.031475	4.352065	7.252328	28	12.535431396843176
200, zipfian[2], 5, 100, fixed[145], True	0.146059	0.871147	1.742974	2.684942	4.642014	28.0	18.632622629208388
200, zipfian[2], 5, 100, exp[7], False	0.072608	0.80312	2.212802	4.185949	6.24715	29	14.141891251684653
200, zipfian[2], 5, 200, fixed[145], False	0.290894	1.742957	4.062461	8.848933	14.649242	28	12.422822439170252
200, zipfian[2], 5, 200, fixed[145], True	0.29376	1.745076	3.558179	5.37079	8.921783	28.0	19.283821991810548
200, zipfian[2], 5, 200, bimodal[100:1000:95], False	0.202441	1.503655	4.202249	7.903865	12.803905	31	14.488567687002314
200, zipfian[2], 5, 200, exp[7], False	0.331377	1.787797	4.474321	8.792961	15.480407	28	11.992132441590371
200, zipfian[2], 10, 50, fixed[145], False	0.000505	0.292367	1.01666	1.886892	3.336377	24	13.258407421314002
200, zipfian[2], 10, 50, fixed[145], True	0.073459	0.435834	0.873317	1.38048	2.249356	28.0	19.147103751645215
200, zipfian[2], 10, 50, bimodal[100:1000:95], False	0.00059	0.20247	0.901303	1.401649	5.001309	26	9.432006457528901
200, zipfian[2], 10, 50, exp[7], False	0.00053	0.476513	1.299272	2.653987	3.951432	21	11.063892874963626
200, zipfian[2], 10, 100, fixed[145], False	0.14635	0.874566	2.324497	4.206552	6.962819	26	13.008229916821476
200, zipfian[2], 10, 100, fixed[145], True	0.14863	0.874609	1.747115	2.977125	4.643574	28.0	18.631893591019875
200, zipfian[2], 10, 100, bimodal[100:1000:95], False	0.101452	0.704146	2.61959	3.808479	6.617323	30	14.051183966812227
200, zipfian[2], 10, 100, exp[7], False	0.076184	0.825377	2.026108	4.222883	6.585483	28	13.497074441629405
200, zipfian[2], 10, 200, fixed[145], False	0.293134	9.893023	12.935206	17.286162	23.084981	28	8.151787921455078
200, zipfian[2], 10, 200, fixed[145], True	0.297344	1.750096	3.565799	6.100572	10.22885	28.0	17.127417021090015
200, zipfian[2], 10, 200, bimodal[100:1000:95], False	0.206349	1.90934	5.607804	8.80577	14.904179	30	12.575769008274856
200, zipfian[2], 10, 200, exp[7], False	0.33699	1.941814	4.809565	8.172073	14.484576	29	12.754210547528055
200, zipfian[2], 15, 50, fixed[145], False	0.004605	0.43974	1.307335	2.178449	3.626759	20	12.310486219887935
200, zipfian[2], 15, 50, fixed[145], True	0.073148	0.437064	0.876479	1.453074	2.467652	28.0	17.673933599031468
200, zipfian[2], 15, 50, bimodal[100:1000:95], False	0.004635	0.304959	1.303738	2.301321	3.401459	18	13.509048225410897
200, zipfian[2], 15, 50, exp[7], False	0.004655	0.552057	1.465479	2.819285	4.115606	20	10.676011450662841
200, zipfian[2], 15, 100, fixed[145], False	0.145381	1.022871	2.328896	4.501722	7.398357	23	12.31152289598536
200, zipfian[2], 15, 100, fixed[145], True	0.149793	0.87577	1.749711	2.766664	5.296875	28.0	16.6110335468128
200, zipfian[2], 15, 100, bimodal[100:1000:95], False	0.100406	0.909844	2.406825	5.906538	8.803041	25	9.801904490438682
200, zipfian[2], 15, 100, exp[7], False	0.076361	0.960875	2.251891	4.54027	6.900896	25	12.946705018259385
200, zipfian[2], 15, 200, fixed[145], False	0.301353	1.903585	4.352328	7.843923	13.201449	28	13.652953526370169
200, zipfian[2], 15, 200, fixed[145], True	0.306446	1.760996	3.578055	5.735192	10.305084	28.0	17.022255918723495
200, zipfian[2], 15, 200, bimodal[100:1000:95], False	0.211586	1.319075	3.00215	6.01423	10.006235	39	18.17583502739689
200, zipfian[2], 15, 200, exp[7], False	0.3403	1.817157	4.294519	7.09463	13.773867	28	13.361538650655623
500, uniform, 5, 50, fixed[145], False	7.2e-05	0.290399	0.87065	1.306105	1.596364	28	28.712381755233835
500, uniform, 5, 50, fixed[145], True	0.07271	0.435423	0.871296	1.306872	1.669832	28.0	27.55454007890518
500, uniform, 5, 50, bimodal[100:1000:95], False	9e-05	0.200384	0.600647	1.001033	1.301079	36	35.68041480623037
500, uniform, 5, 50, exp[7], False	0.000108	0.395298	0.802586	1.400739	1.648137	29	28.585737775023386
500, uniform, 5, 100, fixed[145], False	0.145165	0.870786	1.741447	2.612093	3.337603	28	27.573975461367755
500, uniform, 5, 100, fixed[145], True	0.145436	0.871584	1.743529	2.686119	3.411748	28.0	28.10698869030989
500, uniform, 5, 100, bimodal[100:1000:95], False	0.100224	0.600719	1.201446	2.102535	3.202708	40	30.276469581987925
500, uniform, 5, 100, exp[7], False	0.072208	0.802608	1.749122	2.743313	3.364267	29	27.696534609589648
500, uniform, 5, 200, fixed[145], False	0.29048	1.741717	3.483146	5.368955	6.819871	28	27.578888030266175
500, uniform, 5, 200, fixed[145], True	0.290756	1.743456	3.558072	5.372604	6.823869	28.0	27.57255372856751
500, uniform, 5, 200, bimodal[100:1000:95], False	0.200374	1.201947	2.901843	4.604387	5.804821	40	32.24059219519744
500, uniform, 5, 200, exp[7], False	0.393387	1.81502	3.668818	5.216869	6.870182	36	27.442773926565604
500, uniform, 10, 50, fixed[145], False	9.2e-05	0.290496	0.871146	1.305882	1.597158	28	28.698357306027805
500, uniform, 10, 50, fixed[145], True	0.07286	0.436008	0.871877	1.307764	1.670906	28.0	27.5639124437834
500, uniform, 10, 50, bimodal[100:1000:95], False	0.000114	0.200487	0.601211	1.001863	1.501843	35	31.206953907953217
500, uniform, 10, 50, exp[7], False	9.8e-05	0.395414	0.802698	1.401347	1.649006	29	28.57288170655536
500, uniform, 10, 100, fixed[145], False	0.145265	0.871354	1.742363	2.613273	3.338437	28	28.700638847520104
500, uniform, 10, 100, fixed[145], True	0.145732	0.872307	1.744512	2.688189	3.414293	28.0	27.566670682211697
500, uniform, 10, 100, bimodal[100:1000:95], False	0.100456	0.600337	1.401921	2.1026	3.003779	36	32.21343332383037
500, uniform, 10, 100, exp[7], False	0.072338	0.802976	1.749651	2.744459	3.365468	29	27.68417450774769
500, uniform, 10, 200, fixed[145], False	0.290666	1.742744	3.48518	5.372111	6.823557	28	27.568422412190095

500, uniform, 10, 200, fixed[145], True	0.291513	1.746066	3.562022	5.377318	6.829491	28.0	27.563285648389773
500, uniform, 10, 200, bimodal[100:1000:95], False	0.200525	1.603652	3.603325	5.704558	7.007451	35	26.997395561250208
500, uniform, 10, 200, exp[7], False	0.329773	1.750025	3.613067	5.159644	6.812811	37	27.651047649806117
500, uniform, 15, 50, fixed[145], False	0.000208	0.290795	0.871825	1.307621	1.597981	28	28.68948169008589
500, uniform, 15, 50, fixed[145], True	0.073006	0.43553	0.873164	1.309373	1.672717	28.0	27.540865135688332
500, uniform, 15, 50, bimodal[100:1000:95], False	0.000144	0.200717	0.601539	1.002521	1.302282	36	33.2757662577075
500, uniform, 15, 50, exp[7], False	0.00037	0.472859	0.831459	1.434342	1.676753	28	28.0288697358279
500, uniform, 15, 100, fixed[145], False	0.145341	0.871818	1.743339	2.614896	3.340754	28	28.684058576289697
500, uniform, 15, 100, fixed[145], True	0.145963	0.873594	1.746577	2.691134	3.412216	28.0	27.5074256295487
500, uniform, 15, 100, bimodal[100:1000:95], False	0.100498	0.601954	1.60462	2.704151	3.405238	35	27.764396880947654
500, uniform, 15, 100, exp[7], False	0.072403	0.803487	1.750378	2.746341	3.36765	29	27.66939062594249
500, uniform, 15, 200, fixed[145], False	0.291001	1.744386	3.487617	5.371818	6.826263	28	27.56147759837553
500, uniform, 15, 200, bimodal[100:1000:95], False	0.200777	1.60476	3.308505	5.108696	6.311399	32	29.82169904358829
500, uniform, 15, 200, exp[7], False	0.330482	1.786738	3.657127	5.216728	6.864924	36	27.037575335134125
500, zipfian[0.99], 5, 50, fixed[145], False	0.000186	0.290515	0.870943	1.305842	1.740809	28	24.617893368087234
500, zipfian[0.99], 5, 50, fixed[145], True	0.072807	0.435763	0.871759	1.307211	1.67021	28.0	25.52972911425626
500, zipfian[0.99], 5, 50, bimodal[100:1000:95], False	0.000164	0.200518	0.701324	1.201317	1.602058	31	17.849614144891028
500, zipfian[0.99], 5, 50, exp[7], False	0.000145	0.472413	0.830263	1.433724	1.675679	28	19.686713515795045
500, zipfian[0.99], 5, 100, fixed[145], False	0.145235	0.871472	1.74163	2.611946	3.482552	28	24.623591469008993
500, zipfian[0.99], 5, 100, fixed[145], True	0.145418	0.8722	1.744145	2.683931	3.41225	28.0	25.533894585358915
500, zipfian[0.99], 5, 100, bimodal[100:1000:95], False	0.10013	0.701676	1.402687	2.801023	3.502405	32	25.634268710709073
500, zipfian[0.99], 5, 100, exp[7], False	0.07257	0.822533	1.787422	2.779986	3.393038	28	27.1953071777934
500, zipfian[0.99], 5, 200, fixed[145], False	0.29076	1.743327	3.48513	5.370082	7.542742	28	23.371860677066444
500, zipfian[0.99], 5, 200, fixed[145], True	0.291443	1.74588	3.561321	5.374559	6.8911	28.0	25.069162686205907
500, zipfian[0.99], 5, 200, bimodal[100:1000:95], False	0.200822	1.602096	4.505293	6.303641	8.601773	34	19.22691275576841
500, zipfian[0.99], 5, 200, exp[7], False	0.344625	1.79541	3.647598	5.196868	7.580674	36	23.444473815160254
500, zipfian[0.99], 10, 50, fixed[145], False	0.000158	0.290865	0.871799	1.305662	1.74214	28	22.973696954882417
500, zipfian[0.99], 10, 50, fixed[145], True	0.07289	0.435572	0.873034	1.308873	1.671691	28.0	26.491413603022988
500, zipfian[0.99], 10, 50, bimodal[100:1000:95], False	0.000186	0.20085	0.601786	1.001687	1.402193	36	29.38740773088658
500, zipfian[0.99], 10, 50, exp[7], False	0.000183	0.472723	0.831334	1.43442	1.676652	28	19.74197242046453
500, zipfian[0.99], 10, 100, fixed[145], False	0.145551	0.872739	1.743287	2.614837	3.772321	28	22.240573166259185
500, zipfian[0.99], 10, 100, fixed[145], True	0.145774	0.874094	1.746123	2.689776	3.483359	28.0	26.01399292679532
500, zipfian[0.99], 10, 100, bimodal[100:1000:95], False	0.101243	0.701754	1.902472	3.304443	7.202652	30	12.984360467660307
500, zipfian[0.99], 10, 100, exp[7], False	0.103166	0.832444	1.793861	2.787283	3.582183	28	23.43772888407113
500, zipfian[0.99], 10, 200, fixed[145], True	0.293408	1.750292	3.565822	5.379976	6.894825	28.0	26.51672695023566
500, zipfian[0.99], 10, 200, bimodal[100:1000:95], False	0.202084	1.206688	2.808718	4.406536	6.404732	40	28.161386711739905
500, zipfian[0.99], 10, 200, exp[7], False	0.331353	1.7918	3.641559	5.188278	7.555112	37	23.394737891790214
500, zipfian[0.99], 15, 50, fixed[145], False	0.000258	0.291535	0.870876	1.306253	1.742462	28	22.978870469026322
500, zipfian[0.99], 15, 50, fixed[145], True	0.073104	0.435299	0.873523	1.310388	1.672705	28.0	27.562377795307448
500, zipfian[0.99], 15, 50, bimodal[100:1000:95], False	0.000261	0.201539	0.601946	1.002821	1.401056	36	20.82460435334189
500, zipfian[0.99], 15, 50, exp[7], False	0.000319	0.381096	0.757862	1.352271	1.60558	29	23.02570682012226
500, zipfian[0.99], 15, 100, fixed[145], False	0.146524	0.87421	1.745822	2.616924	3.917652	28	21.543552445624076
500, zipfian[0.99], 15, 100, fixed[145], True	0.147107	0.875239	1.747997	2.690045	3.487585	28.0	24.616754064349177
500, zipfian[0.99], 15, 100, bimodal[100:1000:95], False	0.101787	0.705218	1.606206	2.302178	4.102644	32	21.73138744695097
500, zipfian[0.99], 15, 100, exp[7], False	0.073908	0.805967	1.72268	2.713581	3.800815	29	21.6270131775554
500, zipfian[0.99], 15, 200, fixed[145], False	0.291246	1.744738	3.489718	5.377854	6.824313	28	24.192066623016114
500, zipfian[0.99], 15, 200, fixed[145], True	0.294006	1.754152	3.569657	5.386197	6.834469	28.0	26.77395167596236
500, zipfian[0.99], 15, 200, bimodal[100:1000:95], False	0.201322	2.510282	4.311428	6.207814	8.808957	33	20.404067877396447
500, zipfian[0.99], 15, 200, exp[7], False	0.33152	1.753226	3.60925	5.163426	6.945252	37	25.658703830049063
500, zipfian[2], 5, 50, fixed[145], False	0.000736	0.293782	1.307608	2.177182	3.626519	20	12.311592620234396
500, zipfian[2], 5, 50, fixed[145], True	0.073524	0.437871	0.871514	1.454055	2.684532	28.0	16.41327847359136
500, zipfian[2], 5, 50, bimodal[100:1000:95], False	0.000696	0.203621	1.301839	3.302041	4.301368	21	10.867075239716812
500, zipfian[2], 5, 50, exp[7], False	0.000728	0.517288	1.347763	2.424192	3.353433	21	12.751445758920147
500, zipfian[2], 5, 100, fixed[145], False	0.14671	0.876821	2.180908	4.49988	7.397909	26	12.311375871353016
500, zipfian[2], 5, 100, fixed[145], True	0.149452	0.872737	1.748745	2.836184	5.44128	28.0	16.22053438550533

500, zipfian[2], 5, 100, bimodal[100:1000:95], False	0.101698	0.702076	2.006582	5.004546	7.902485	30	11.901843119425473
500, zipfian[2], 5, 100, exp[7], False	0.07372	0.803968	2.314491	5.265707	7.627695	29	11.832506142549752
500, zipfian[2], 5, 200, fixed[145], False	0.302191	1.893634	4.508307	9.869875	15.666041	28	11.685767150455478
500, zipfian[2], 5, 200, fixed[145], True	0.304597	1.753934	3.570349	5.804997	11.110075	28.0	15.923438832219583
500, zipfian[2], 5, 200, bimodal[100:1000:95], False	0.211916	1.614487	4.011601	9.508754	15.305249	36	12.266928361138373
500, zipfian[2], 5, 200, exp[7], False	0.34008	1.945884	4.57816	8.748575	15.432415	29	12.027663867448407
500, zipfian[2], 5, 500, fixed[145], False	0.890339	4.670613	11.906196	25.109938	39.60106	28	11.567498807853573
500, zipfian[2], 5, 500, fixed[145], True	0.873024	4.515739	9.037366	14.703151	28.165464	28.5	15.730723241394807
100, uniform, 5, 500, fixed[145], False	0.870617	4.496587	8.993946	13.49084	17.116925	28	27.358568512148025
100, uniform, 5, 500, fixed[145], True	0.872501	4.498215	9.000989	13.569537	17.195765	28.0	27.031281896912635
100, uniform, 5, 500, bimodal[100:1000:95], False	0.601155	4.001424	8.302507	12.004977	15.605546	36	28.8956827133832
100, uniform, 5, 500, exp[7], False	0.856108	4.244602	8.912831	12.921232	16.757693	37	27.477784486020894
100, uniform, 10, 500, fixed[145], True	0.872564	4.49952	9.000581	13.565793	17.193115	28.0	26.930078743550244
100, uniform, 10, 500, bimodal[100:1000:95], False	0.602631	4.204628	8.505475	12.504745	15.407554	36	27.315726198960977
100, uniform, 10, 500, exp[7], False	0.803966	4.465929	9.234185	13.820693	17.689967	37	26.163558871931997
100, uniform, 15, 500, fixed[145], False	0.870815	4.497491	8.995079	13.493675	17.118046	28	26.722463169765135
100, uniform, 15, 500, fixed[145], True	0.872448	4.503209	8.999315	13.56544	17.191172	28.0	27.140029306889243
100, uniform, 15, 500, bimodal[100:1000:95], False	0.600501	3.80117	8.101897	12.004203	15.608535	38	28.894134779581094
100, uniform, 15, 500, exp[7], False	0.810736	4.477779	9.244052	13.832716	17.702368	38	26.163434287735296
100, zipfian[0.99], 5, 500, fixed[145], False	0.871577	4.497194	8.994337	13.925485	19.871399	28	22.531081288671505
100, zipfian[0.99], 5, 500, fixed[145], True	0.874644	4.501252	9.002923	13.566793	17.333462	28.0	24.97756764649668
100, zipfian[0.99], 5, 500, bimodal[100:1000:95], False	0.601831	4.202388	8.702989	13.604031	21.804826	37	20.571808221210464
100, zipfian[0.99], 5, 500, exp[7], False	0.803155	4.465988	9.234122	13.855084	20.12653	37	22.268211522356484
100, zipfian[0.99], 10, 500, bimodal[100:1000:95], False	0.600576	4.203999	8.503678	12.602869	22.305988	37	20.15697282904565
100, zipfian[0.99], 10, 500, exp[7], False	0.757554	4.433893	9.195424	14.074134	22.308811	36	19.260246296948004
100, zipfian[0.99], 15, 500, fixed[145], False	0.871531	4.497252	8.994708	13.494146	17.552471	28	23.61194467834699
100, zipfian[0.99], 15, 500, fixed[145], True	0.874379	4.502586	9.00426	13.567149	17.841322	28.0	23.448900928909453
100, zipfian[0.99], 15, 500, bimodal[100:1000:95], False	0.603783	3.802843	8.307769	13.402977	18.406714	40	22.930406948683768
100, zipfian[0.99], 15, 500, exp[7], False	0.856457	4.529389	9.298052	13.882395	19.750224	36	21.366902275224675
100, zipfian[2], 5, 500, fixed[145], False	0.870673	4.497256	11.025718	23.208617	37.711633	28	12.09556307652292
100, zipfian[2], 5, 500, fixed[145], True	0.872736	4.500467	8.998911	16.467293	23.724225	28.0	18.577272928413926
100, zipfian[2], 5, 500, bimodal[100:1000:95], False	0.60224	6.004132	12.904138	27.806554	45.909384	32	10.13993413301586
100, zipfian[2], 5, 500, exp[7], False	0.811111	4.477097	12.611888	26.924595	42.378535	38	10.86213976419511
100, zipfian[2], 10, 500, fixed[145], False	0.876216	4.503131	12.331348	22.920769	37.421608	28	12.181140675632815
100, zipfian[2], 10, 500, bimodal[100:1000:95], False	0.801115	4.407541	13.004967	23.908767	38.409492	33	11.958914717965499
100, zipfian[2], 10, 500, exp[7], False	0.829642	4.526229	13.21943	26.228235	41.680177	36	11.029569990375377
100, zipfian[2], 15, 500, bimodal[100:1000:95], False	0.705017	4.601269	11.708765	23.311846	36.010338	31	12.403770617268437
100, zipfian[2], 15, 500, exp[7], False	0.806632	4.65495	13.031117	22.687062	38.137126	34	11.964807576460743
200, uniform, 5, 500, bimodal[100:1000:95], False	0.601706	4.001862	8.704366	12.906236	15.611773	40	27.92063370010436
200, uniform, 5, 500, exp[7], False	0.919074	4.321269	8.983016	12.99479	16.831571	38	27.355194729594995
200, uniform, 10, 500, exp[7], False	0.756944	4.423022	9.198479	13.78544	17.650645	36	26.240476543847944
200, uniform, 15, 500, fixed[145], False	0.873841	4.496967	8.99622	13.495856	17.123406	28	27.56404920293039
200, uniform, 15, 500, bimodal[100:1000:95], False	0.603991	4.008289	8.310184	12.012638	15.613706	36	28.553790487133377
200, uniform, 15, 500, exp[7], False	0.811368	4.477319	9.243595	13.839329	17.705622	38	26.53702304809448
200, zipfian[0.99], 5, 500, bimodal[100:1000:95], False	0.601348	3.805141	8.40413	13.604174	18.804382	36	23.251328069356664
200, zipfian[0.99], 5, 500, exp[7], False	0.822227	4.49751	9.265083	13.85218	17.903924	37	24.72355114056664
200, zipfian[0.99], 10, 500, bimodal[100:1000:95], False	0.703776	4.001766	9.310482	13.309617	17.405626	39	27.315350145093678
200, zipfian[0.99], 10, 500, exp[7], False	0.857101	4.530914	9.296528	13.885949	18.575342	36	23.502446440157293
200, zipfian[0.99], 15, 500, bimodal[100:1000:95], False	0.603663	4.408851	8.413236	13.114851	20.209191	40	22.020860537353922
200, zipfian[0.99], 15, 500, exp[7], False	0.760517	4.427599	9.197119	13.799458	20.87918	36	20.397032313386532
200, zipfian[2], 5, 500, fixed[145], False	0.871416	4.497296	11.171842	20.165254	34.376364	28	13.155343638755948
100, uniform, 10, 500, fixed[145], True	0.870892	4.498276	9.005974	13.567543	17.193142	28.0	27.0335384566142
100, uniform, 15, 500, fixed[145], True	0.874234	4.499405	8.999946	13.566386	17.189785	28.0	27.243863142741986
100, zipfian[0.99], 5, 500, fixed[145], True	0.871993	4.498762	8.998763	13.565814	17.3337	28.0	24.978647003608966
100, zipfian[0.99], 15, 500, fixed[145], True	0.874396	4.502834	9.005973	13.567722	17.842741	28.0	23.44838078144449

100, zipfian[2], 10, 500, fixed[145], True	0.880435	4.505211	9.006094	14.365592	24.300989	28.0	17.904849048514837
100, zipfian[2], 15, 500, fixed[145], True	0.874465	4.513166	9.012434	15.018641	27.129341	28.0	16.258805321845166
200, uniform, 5, 500, fixed[145], True	0.873171	4.506727	9.00501	13.569456	17.194806	28.0	27.35704616561726
200, uniform, 10, 500, fixed[145], True	0.872281	4.505736	9.004772	13.565607	17.203367	28.0	27.466330947534583
200, zipfian[0.99], 15, 500, fixed[145], True	0.877635	4.504617	9.008071	13.567469	18.428689	28.0	24.10450905213502
200, zipfian[2], 5, 500, fixed[145], True	0.875207	4.500271	9.004648	13.566042	21.833003	28.0	19.64094930521713
200, zipfian[2], 10, 500, fixed[145], True	0.883807	4.518259	9.025239	14.892964	25.610334	28.0	17.105410759583897
500, uniform, 10, 500, fixed[145], True	0.876754	4.500099	9.022494	13.590363	17.195532	28.0	27.461194448818404
500, uniform, 15, 500, fixed[145], True	0.880175	4.510043	9.022194	13.584256	17.223261	28.0	27.454084964682515
500, zipfian[0.99], 5, 500, fixed[145], True	0.87665	4.501895	9.011261	13.574648	17.191372	28.0	24.71000340009647
500, zipfian[0.99], 10, 500, fixed[145], True	0.874242	4.512859	9.023291	13.586632	17.412868	28.0	26.21230879693471
200, uniform, 15, 200, fixed[145], False	0.29068	1.742317	3.48426	5.370565	6.820615	28	27.57237507198113
200, uniform, 15, 200, fixed[145], True	0.292338	1.745476	3.560209	5.39281	6.846086	28.0	27.21646835840605
200, zipfian[2], 15, 200, fixed[145], False	0.30182	1.904426	4.352973	7.844981	13.202452	28	13.65216788234671
200, zipfian[2], 15, 200, fixed[145], True	0.306715	1.761479	3.578907	5.736759	10.31061	28.0	17.01456634038967
50, zipfian[2], 15, 200, fixed[145], False	0.291934	1.886634	4.932098	9.864194	15.672124	28	11.676952212715303
50, zipfian[2], 15, 200, fixed[145], True	0.293225	1.743717	3.557974	5.369695	8.704776	28.0	19.69520488724692
100, zipfian[2], 15, 200, fixed[145], False	0.291351	1.743687	5.080825	8.850945	14.650102	28	12.422240655447105
100, zipfian[2], 15, 200, fixed[145], True	0.2954	1.745805	3.561314	5.805578	9.938211	28.0	17.56325780517763

Bibliography

[1] Andrew Huang. Moore's Law is Dying (and that could be good). *IEEE Spectrum*, 52(4): 43–47, 4 2015. ISSN 00189235. doi: 10.1109/MSPEC.2015.7065418.

[2] Microsoft. GitHub - microsoft/verona: Research programming language for concurrent ownership, [Accessed 2nd Feb 2023], 2023. URL https://github.com/microsoft/verona.

[3] Microsoft. GitHub - microsoft/verona-rt: The runtime for the Verona project [Accessed 5th Feb 2023]. URL https://github.com/microsoft/verona-rt.

[4] D R Gene and M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings - 1967 Spring Joint Computer Conference, AFIPS 1967*, pages 483–485, 4 1967. doi: 10.1145/1465482.1465560. URL https://dl.acm.org/doi/10.1145/1465482.1465560.

[5] A W Roscoe. The Theory and Practice of Concurrency. 1997.

[6] Nicolai M. Josuttis. *The C++ standard library : a tutorial and reference*. Addison-Wesley, 2nd edition, 2012. ISBN 978-0-321-62321-8.

[7] A. N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7): 373–ff., 7 1969. ISSN 15577317. doi: 10.1145/363156.363160.

[8] Robert H.B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 3 1992. ISSN 15577384. doi: 10.1145/130616.130623. URL https://dl.acm.org/doi/10.1145/130616.130623.

[9] Rail Safety and Standards Board. *RS521 Signals, Handsignals, Indicators and Signs*. RSSB, 6 edition, 2020.

[10] C. Y. Teo, O. Chutatape, and C. H. Tan. Microcomputer-based multi-tasking on a SCADA system using interrupts in MS-DOS. *Computers in Industry*, 22(1):93–102, 6 1993. ISSN 0166-3615. doi: 10.1016/0166-3615(93)90085-F.

[11] Muhammad Zulkifl Hasan, M Zunnurain Hussain, and Zaka Ullah. Computer Viruses, Attacks, and Security Methods. *Lahore Garrison University Research Journal of Computer Science and Information Technology*, 3(3):20–25, 9 2019. ISSN 2521-0122. doi: 10.54692/LGURJCSIT.2019.030380. URL https://lgurjcsit.lgu.edu.pk/index.php/lgurjcsit/article/view/80.

[12] Kang Su Gatlin. Trials and Tribulations of Debugging Concurrency. *Queue*, 2(7):66–73, 10 2004. ISSN 15427749. doi: 10.1145/1035594.1035623. URL https://dl.acm.org/doi/10.1145/1035594.1035623.

[13] C. M. Tobar and J. M. Adán-Coello. Semaphores , Are They Really Like Traffic Signals ? 2009.

[14] Daniel R. Ries and Michael Stonebraker. Effects of locking granularity in a database management system. *ACM Transactions on Database Systems (TODS)*, 2(3):233–246, 9 1977. ISSN 15574644. doi: 10.1145/320557.320566. URL https://dl.acm.org/doi/10.1145/320557.320566.

[15] Transactions and Concurrency - ADO.NET | Microsoft Learn [Accessed 15th June 2023]. URL https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/transactions-and-concurrency.

[16] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. *Sessions 8: Formalisms for Artificial Intelligence*, pages 235–245, 1973.

[17] Gul A Agha. Actors: A model of concurrent computation in distributed systems. *Technical Report. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab.*, 1985.

[18] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why DO scala developers mix the actor model with other concurrency models? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7920 LNCS:302–326, 2013. ISSN 16113349. doi: 10.1007/978-3-642-39038-8{_}13/COVER. URL https://link.springer.com/chapter/10.1007/978-3-642-39038-8_13.

[19] John L. Hennessy, David A. Patterson, and Andrea C. Arpaci-Dusseau. *Computer architecture : a quantitative approach*. Morgan Kaufmann, 5 edition, 2012. ISBN 0123735904.

[20] Jean-Loup. Baer. *Microprocessor architecture : from simple pipelines to chip multiprocessors*, volume 2011 12 30. Cambridge University Press, 1 edition, 2010. ISBN 0511671466.

[21] Project Verona - Microsoft Research [Accessed 8th Feb 2023]. URL https://www.microsoft.com/en-us/research/project/project-verona/.

[22] Matthew Parkinson. talks.cam : When Concurrency Strikes [Accessed 17th June 2023]. URL https://talks.cam.ac.uk/talk/index/192623.

[23] Microsoft Research Matthew Parkinson. Concurrency Meeting: When Concurrency Strikes: Behaviour Oriented Concurrency - YouTube [Accessed 17th June 2023], 2022. URL https://www.youtube.com/watch?v=Fqc-JioEv0M&t=480s.

[24] Gul Agha and Carl Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. *Research Directions in Object-Oriented Programming*, pages 49–74, 1987. URL http://osl.cs.illinois.edu/publications/books/mit/shriverW87/AghaH87.html.

[25] Yue Yang and Jianwen Zhu. Write Skew and Zipf Distribution: Evidence and Implications. 19:pages, 2016. doi: 10.1145/2908557. URL http://dx.doi.org/10.1145/2908557.

[26] N. Unnikrishnan Nair, P.G. Sankaran, and N. Balakrishnan. Discrete Lifetime Models. *Reliability Modelling and Analysis in Discrete Time*, pages 107–173, 2018. doi: 10.1016/B978-0-12-801913-9.00003-8.

[27] Exponential Distribution | Definition | Memoryless Random Variable [Accessed 18th June 2023]. URL https://www.probabilitycourse.com/chapter4/4_2_2_exponential.php.