A study in Concurrency Benchmarking
— Final Report —

Simon Charalambous, Jordan Loh, Yerzhan Mazhkenov,
Karolis Mituzas, Ethan Moss, Andreas Voskou
{sc1016, jpl216, ym1916, km2016, edm115, av1316}@doc.ic.ac.uk

Supervisors: Professor Sophia Drossopoulou, Juliana Franco
Course: CO530, Imperial College London

17th May, 2017

Contents

1 Introduction

In response to slowing gains in sequential performance over the course of the past decade, the computer industry has seen a move towards multi-core parallel computing[1]. This in turn has led to growing interest in concurrent programming languages, and subsequently the performance comparison of these languages[2]. While concurrency does not guarantee parallelism, concurrent workloads are considerably easier to parallelise than sequential ones[3], offering much better performance scaling with increasing numbers of cores.

This project investigates three different programming languages with a strong focus on concurrency and identifies how their performance differs in a number of situations. To this end, we developed a series of benchmarks in each language, and a tool that runs the benchmarks, collects information on their space and time requirements, and optionally generates graphs to permit easy visual comparison of the results.

The overall goal of the project is to produce a benchmarking suite for concurrent programs that can be easily extended by other users, as well as to gather information on the relative performance of the languages being investigated. This should allow researchers and developers to compare the properties of these languages and make more informed decisions about the best application of each.

In order to make useful comparisons and comments on performance differences, we focused on benchmarking programming languages that provide high-level linguistic support for concurrent programming. More specifically, we focused on languages based on the programming paradigm of either the actor[4] or communicating sequential processes (CSP)[5] models. The languages selected for our project were Scala[6] with the Akka actor library[7] (Scala-Akka), Pony[8], and Go[9]. Scala-Akka and Pony are based on the actor model, while Go uses the CSP model. These three languages share a number of features which have made them popular for writing concurrent programs.

To allow other developers to use and extend our suite, we created a front-end tool for running the benchmarks and collecting results. This is written in Python, with a focus on ease of use and extensibility. For example, it was a requirement that the front-end was able to automatically detect, compile, and run benchmarks included in a specified directory, and produce graphical representations of the results, permitting researchers and developers to add additional benchmarks to the suite and obtain new results with minimal overhead.

2 Specification

We implement a series of benchmarks across the three languages, and a front-end tool to run the benchmarks and compare the results.

An existing benchmark suite for actor-model languages, the Savina suite[2], is used as the basis of the Sleeping Barber problem benchmark, with the Scala-Akka implementation thereof being ported to our suite. All other benchmarks are developed by our team in order to minimise variance between implementations, and ensure that the runtime parameters and format of results are standardised.

2.1 Benchmarks

Each benchmark permits the analysis of distinct aspects of the performance of the languages. We select three benchmarks to write:

- **Ping-Pong Test**

  Tests the overheads of passing messages between concurrent execution units.

- **Sleeping Barber Problem**

  Tests inter-actor communication and synchronisation.

- **N-Body Simulation**

  Tests the performance of parallel numerical calculations.

2.2 Languages

Each benchmark is implemented in three languages: Scala-Akka, Pony, and Go. Benchmarks must utilise the concurrency paradigm of the language they are written in; those written in Scala-Akka and Pony must be actor-based, those in Go must use goroutines and channels. This ensures that the concurrency features of the languages are utilised and benchmarked appropriately.

2.3 Front-End Script

In order to simplify the task of running the benchmarks in each of the languages, we use a command line tool. The initial specifications for this tool were to:

- Provide a single interface to run the entire suite and collect performance results.
- Provide a facility to parse configuration files containing parameters for each of the tests.
- Notify users if some of the dependencies are missing.
- Enable the easy addition of benchmarks and the extension of languages covered by the suite.
- Run on Linux systems which have the required compilers and/or virtual machines.

Based on these requirements, the final specification for the front-end script is to provide an interface to perform the following tasks:

- Automatically discover all the available tests in a given source tree.
- Compile the tests in different languages, using compilation rules provided for each of the languages to make it relatively easy to add new ones.
- Run the tests and collect the required metrics.
- Create human-readable reports containing the results.

2.4 Extended Goals

We also identified possible extensions to the project, if time permitted:

- **Add languages or benchmarks**

  A natural extension is to add either additional languages or benchmarks. Additional languages could maintain a focus on the actor-model (e.g. Encore[10], Erlang[11]), or extend coverage to concurrent languages focusing on different paradigms (e.g. Java).

- **Benchmark visualisations**

  An interesting extension is to generate visualisations for some of the benchmarks, for example the movement of the bodies in the N-Body simulation. These must be an optional feature to avoid any interference with the benchmark performance results.

- **Investigate reasons for performance differences**

  Providing explanations for the performance differences between languages is a key extension, but ensuring our conclusions are accurate would require an in-depth investigation of the languages and compilers used. For this reason, a detailed analysis of the underlying reasons for the observed performance differences is not within the scope of this project. Nevertheless, with the information we have available we are able to offer tentative explanations for some of our observations.

- **Collision Detection**

  We initially planned to write a collision-detection benchmark testing real-time performance and garbage collection. However, in the course of our development of the N-Body simulation, we observed that it tests many of the same language features as a collision-detection benchmark — both benchmarks involve many entities sharing information, and a central processing entity that coordinates communication. To extend the N-Body simulation into a collision-detection problem would only require calculating the trajectory for each body and checking whether any two bodies would eventually collide. Considering the similarities of the two benchmarks and the project time constraints, we therefore decided not to develop the collision-detection benchmark.

3 Model and Language Background

When writing our benchmarks, we adopt the actor model paradigm for Pony and Scala-Akka, and the CSP paradigm in Go. Both are based on a message-passing model that provides a higher level of abstraction for writing concurrent and distributed programs. This type of message-passing design frees the programmer from lower-level concurrency management, such as explicit locking and thread control, making it easier to write correct and safe concurrent and parallel programs.

3.1 Actor Model

Hewitt et al. introduced the actor model in 1973[4], and subsequently outlined a framework for viewing control structures as patterns of messages[12], which became the basic building block of the actor model. Actors are lightweight processes that encapsulate state, send/receive messages, have a mailbox/buffer/queue to store unprocessed messages, and are guaranteed to be scheduled on at most one thread for execution — with many actors potentially sharing a single thread. Execution within an actor is strictly sequential, and communication occurs solely through message passing. This means that they require minimal or no synchronization, thus the actor model simplifies concurrency programming through isolated mutability — the internal states of actors are never shared.

As actors encapsulate state and behavior, in the implementation of our benchmarks any program entity that is required to keep both state and a set of behaviors must be an actor, and communicate exclusively by exchanging messages.

3.1.1 Scala-Akka

Scala is a general-purpose programming language supporting functional programming and running on the Java virtual machine (JVM) — it is also compatible with existing Java programs. When used with the Akka actor toolkit, Scala-Akka simplifies concurrent and distributed programming, emphasizing actor-based concurrency.

3.1.2 Pony

Pony is an open-source, object-oriented, actor-model, high performance programming language. It has the distinctive feature of providing capabilities-secure facilities, whereby restrictions are imposed on the actions that can be applied to objects, in order to guarantee safe concurrent execution. Pony is also based on the actor model, which when combined with its capabilities security makes it an extremely powerful concurrent programming language that is type safe, memory safe, exception safe, data-race free, and deadlock free.

3.2 Communicating Sequential Processes Model

When choosing the languages to use in this project, we decided to focus on those that could implement the actor model. While Pony and Scala-Akka were obvious choices, Go's concurrency model is heavily based on Hoare's CSP. Our initial expectation, based on our knowledge of the language at the time, was that the CSP model used in Go was sufficiently similar to the actor model as to not cause significant issues: goroutines could fulfil the role of actors, and channels could be used to pass messages between them. While we were broadly successful in this effort, further experience in writing concurrent Go programs demonstrated the issues with this approach.

3.2.1 Go

Go is an open-source programming language developed at Google, and was chosen as one of our benchmark languages due to the ease of writing concurrent programs through the use of two concurrency features it provides: goroutines and channels. Its most distinctive feature is the adoption of channels as first class objects. In the actor model, actors send and receive messages to other actors. In Go, goroutines — concurrently executing threads — send/receive messages on named channels.

Goroutines start with a small stack space which is usually a few kilobytes and can grow or shrink as needed, making them cheaper than threads. They share an address space, so access to shared memory has to be synchronized. Channels provide a communication mechanism that lets one goroutine exchange information with another in a synchronised, thread-safe manner. Writing to and reading from channels can be blocking, providing a simple but powerful tool for inter-thread synchronisation, or non-blocking to buffered channels, behaving more like the asynchronous communication of actors.

4 Design of Benchmark Suite

4.1 Benchmarks

In order to usefully compare performance between languages, it is important that the fundamental execution design of the benchmarks be as similar as possible across them. However, if a feature of one language can be used to improve performance, it of course should not be discounted simply because it is not present in another, nor should language constraints — such as Pony's capabilities system which restricts data sharing to avoid race conditions — be duplicated in languages without them merely to ensure consistent design. There is no single solution to these conflicting requirements, but we have aimed to keep the specification for the benchmarks sufficiently high-level as to not impose such unnecessary restrictions on our implementations.

In planning the high-level, language-independent structure of the benchmarks, we made use of sequence diagrams. UML sequence diagrams are models used to show the logical flow of a solution to a problem. Commonly used to illustrate dynamic and concurrent programs, they show the behaviour and communication between each actor as execution progresses. The diagrams used in section 4 show the message passing between actors (or goroutines). The vertical axis represents time, showing when

the actors send and receive messages, and what processing they perform. An example is shown in figure 1. These diagrams aided in the visualisation and documentation of solutions before any actual code was written. We base our sequence diagrams on the Microsoft specifications[13].

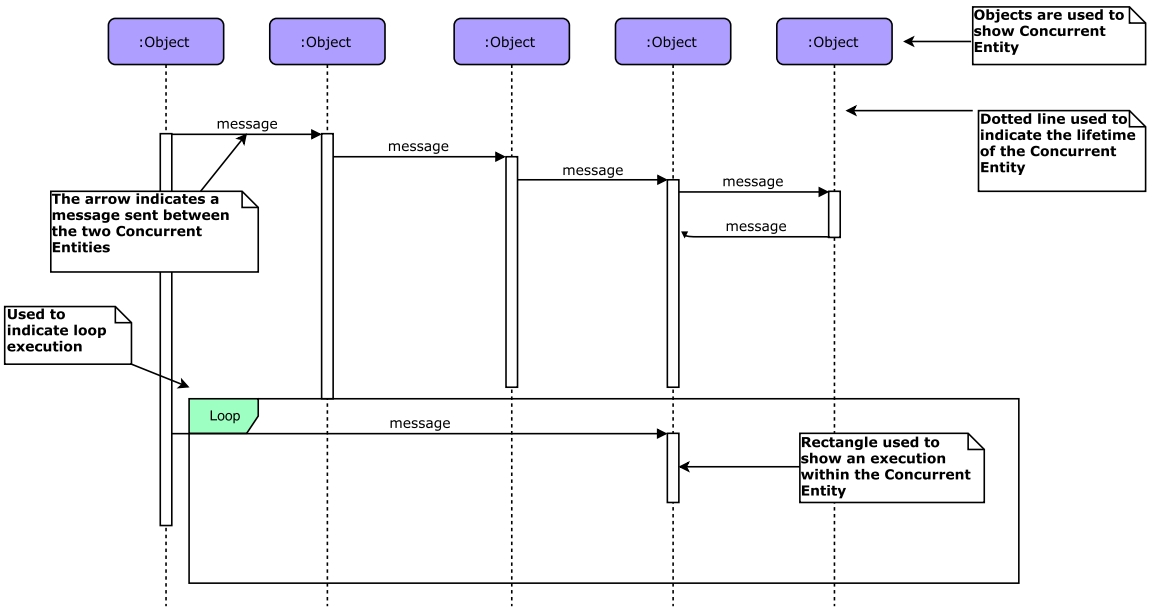


Figure 1: Explanation of the format of sequence diagrams.
The term 'concurrent entity' refers to either an actor or a goroutine.

4.1.1 Ping-Pong Test

In this simple message passing benchmark, a pair of players pass a message between them a fixed number of times. Our implementation endeavours to minimise any overheads not related to message passing. The content of the message being passed between the actors is not important, as long as it remains of consistent size. We choose to pass a value representing the total number of 'pings' and 'pongs' in order to make the message meaningful, but any message would suffice.

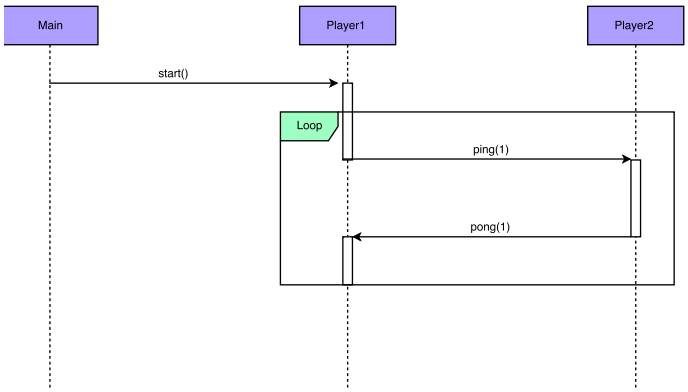


Figure 2: Execution diagram for Ping-Pong test.
Players remain in the loop until the total loop iterations reach a specified value.

This benchmark tests the efficiency of the message passing system of the languages. Because each actor performs a minimum of computation and is very lightweight, the limiting factor for this test is the speed of the message passing processes of the language in question. As shown in the sequence diagram figure 2, the execution of this benchmark is strictly sequential, with each player waiting until they receive a message from their partner before proceeding. Based on this, there is minimal room of performance improvement as the number of available CPU threads is increased.

Arguments: N – total number of 'ping-pong' cycles to complete.

Actors: *Ping* and *Pong*.

Ping/Pong:

- Pass an integer to their partner.
- One of the players increments the integer before passing it back.
- When the value of the integer reaches N, the benchmark execution terminates.

4.1.2 Sleeping Barber Problem

The premise of this problem, first proposed by Dijkstra[14], is as follows: A barber is either cutting customers hair or sleeping; the barbershop has a waiting room of fixed size; when a customer arrives, if the waiting room is full they leave, if the barber is busy they wait, and if the barber is sleeping they wake him and have a haircut; upon finishing a haircut, the barber either starts another, or if no customers are present in the shop goes to sleep.

The primary challenge for the implementation of this problem is to ensure that state is synchronised — on entry to the shop, customers always see an accurate state of the world (otherwise, for example, a customer could see the barber's state as busy and wait, when the barber's true state is sleeping. In this case the barber would not wake and a deadlock could occur). Using the actor model can greatly simplify this challenge, as all actions within actors are strictly sequential, and state can be determined by message passing between actors. As such an implementation of this benchmark is present in the Savina suite, we used the same execution design as found there.

Arguments: N – number of haircuts to be completed, S – number of spaces in the waiting room, APR – average customer production rate, AHR – average haircut rate.

Actors: *Factory*, *Shop*, *Barber*, and multiple *Customers*.

Factory:

- Creates N customer actors. After creating each customer, the factory busy-waits for a random time up to a maximum of APR.
- When all customers have been created, the factory listens for returning customers.
- If a returning customer reports that they were unable to get a haircut due to the shop being full, they are sent back to the shop.
- If a returning customer reports that they have had a haircut, a counter of completed haircuts is updated. When all required haircuts have been completed, execution of the benchmark terminates.

Shop:

- Contains a 'waiting room' of size S.
- When a customer arrives, if the waiting room is not full, the customer is placed in it, otherwise they are turned away.
- Keeps track of whether the barber is asleep. If he is, when a new customer arrives, the shop sends them to the barber.
- When it receives a request for a new customer from the barber, it either sends a customer from the waiting room or informs the barber that no customers are currently waiting.

Barber:

- Initially the barber is 'asleep' — inactive, listening for further messages.
- When the shop sends him a new customer, he sends a 'starting haircut' message to that customer, busy waits for a random time up to a maximum of AHR, then sends a 'haircut finished' message.
- When the barber finishes a haircut, he requests the next customer from the shop. If no customers are currently waiting, the barber sleeps.

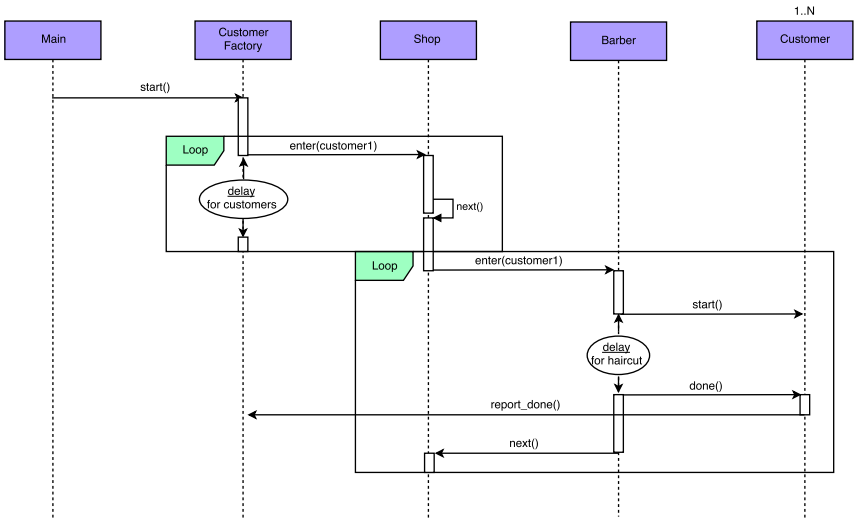


Figure 3: Execution diagram demonstrating customers arriving at and entering the waiting room — customers successfully enter the shop, and are sent to the barber to have their hair cut. On completion, they report back to the factory.

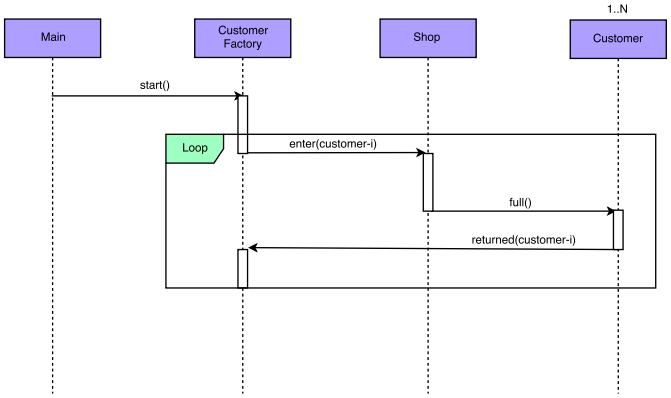


Figure 4: Execution diagram when the shop is full — customers are turned away by the shop. They report this back to the factory, which attempts to send them back to the shop. This repeats until space is made in the waiting room.

4.1.3 N-Body Simulation

The N-Body problem refers to the evolution of the orbits of N bodies interacting gravitationally with each other, with no external forces applied to them.

For each time step, a coordinating actor, referred to as the 'World' actor, dispatches information on the system to each of the N body actors, which then concurrently calculate their updated positions and pass them back. This benchmark therefore tests the performance of running concurrent calculations, message passing, and coordinating multiple actors.

Arguments: N – number of bodies in the system, S – the total number of steps to be calculated.

Actors: *World*, multiple *Bodies*.

World:

- Initiates N bodies with mass, initial position and velocity.
- Requests step updates from all bodies.
- Maintains the positions and velocities of all bodies for the last four steps.
- Keeps track of number of steps.

Body:

- Calculate next step position and velocity, and report back to *World*.

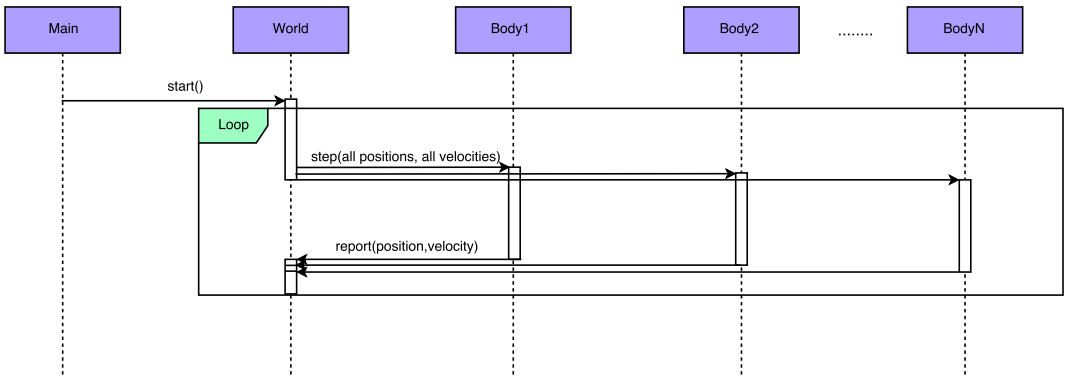


Figure 5: Execution diagram for N-Body simulation.
Each body receives the positions of all the others from 'World', calculates its next position, and passes it back. The loop repeats once for each time step.

The initial conditions of the Bodies are defined to form a planetary system with a massive star in the middle and $N-1$ planets around it.

The differential equations of the motions that describe the system are:

$$\frac{d^2\vec{\mathbf{r_j}}}{dt^2} = \sum_{i \neq j}^{N} \frac{Gm_i\vec{\mathbf{r_{ji}}}}{|r_{ji}|^3}, \forall j \in [1, N]$$

$\vec{\mathbf{r_i}}$: The position vectors, and $\vec{\mathbf{r_{ij}}} = \vec{\mathbf{r_j}} - \vec{\mathbf{r_i}}$

Nonlinear dynamical systems such as this do not usually have analytical solutions. This property means that the only way to predict the motion of the bodies is through a numerical method. We choose to use the multistep method Adams-Bashforth 4th order (AB4). This method uses the positions and velocities of the current and the previous three states of the system in order to calculate the next one. Analytically, a step of AB4 for the body m is:

$$r_m(t_4) = r_m(t_3) + h(\frac{55}{24}u_m(t_3) - \frac{59}{24}u_m(t_2) + \frac{37}{4}u_m(t_1) - \frac{3}{8}u_m(t_0))$$

$$u_m(t_4) = u_m(t_3) + h(\frac{55}{24}f_m(t_3) - \frac{59}{24}f_m(t_2) + \frac{37}{4}f_m(t_1) - \frac{3}{8}f_m(t_0))$$

$h = \Delta t$: The discrete time unit

The AB4 method must be combined with another methodology, specifically a one-step method, to

calculate the first four states of the system before the AB4 based algorithm can be run. For this initialization process we use Euler's method, with a much smaller time unit of h'=0.001h in order to maintain high accuracy. Since we are using a time unit which is one thousand times smaller, it is necessary to run 1000 cycles of the Euler's step to calculate the next state. A step of the Euler method for a body m is:

$$r_m(t_1) = r_m(t_0) + h'u_m$$
$$u_m(t_1) = u_m(t_0) + h'f_m$$

$h' = \Delta t'$: The discrete time unit

In this implementation, each body is an actor that moves concurrently in the phase space step by step, calculating its position based on the last four coordinates of all the other bodies in the system. This is illustrated in the execution diagram in figure 5. The procedure for calculating the next position is separated into two subroutines. In the first, we use the position vectors to calculate the last four states of the forces on the body. The algorithm we use for this task belongs to the family of the direct gravitational N-Body simulations, where forces are calculated without any kind of simplifying approximations. The program has been designed in this way in order to maximise the parallelism and to improve the accuracy of results. The second subroutine runs a step of the Adams Bashforth method using the four force states that have been calculated and the last four values of the body's velocity.

The execution time of the first subroutine is a linear function of $O(N-1)$ performance, because it sums all the forces from the interactions with the $N-1$ other bodies. On the other hand, the execution time of an AB4 step for each body is constant and independent of N. These $O(N)$ and $O(1)$ subroutines combine to give $O(N)$ execution time for the whole step procedure. Running the procedures for each of the N bodies gives us $O(N^2)$ total execution time. In the case of a sequential execution, the expected performance is of the form $T_s(N) = \alpha N^2$. For parallel execution the expected time is locally linear, but, as N increases, the gradient increases each time $NmodC = 1$ (where C is the number of processor cores), resulting in a running time of $T_p(N) = aN\frac{N}{C}$.

4.2 Front-end Script

The front-end script is split into four modules:

Crawler – Crawls a specific directory structure (for details see manual in appendix A.2) and generates information for the compile phase. Saves all the tests discovered and also optionally overrides test names if a special override file is discovered in test-specific directory.

Input to the Crawler is a single directory which contains the test tree, output is a file or data structure containing information — language and source directory — for all the tests discovered. The Crawler does not check whether the test directories discovered contain valid sources for the compilation phase as this adds unnecessary complexity — the assumption is that user of the script has relevant background and knows what they are doing. If the test does not contain valid source code (or malformed source code), a compile error will be generated in the following phase.

Compilation Executor – Takes information generated by the Crawler, as well as language-specific instructions (Python scripts were selected as the format of these instructions), and compiles the tests. Saves the logs of all the compilations and warns the user if any errors have occurred. The Compilation Executor creates a working directory where it saves all the executables to be used in the later phase. It also passes commands required to execute the binaries downstream (this feature is necessary because Scala, for example, requires a prefix to the executable such as `scala -cp <some_classpath>`, whereas the other languages do not).

Test Executor – Takes the executables and execution commands created in the previous phase, as well as test-specific configuration files, and runs the tests while saving all the relevant data from

each run. Multiple runs will be executed for each test step to minimise errors in individual steps. Since each test can have different parameters, configuration files are required. If a test has more than one parameter that might affect execution time or any other metric, a decision has to be made in the configuration file which parameters are kept constant and which single one is iterated through — it is important to have only one variable changing in each run to collect useful data.

Data Interpreter – Takes the data generated by the Test Executor and converts it to human readable form. The final data formats chosen were CSV files, to maximise compatibility, and graphical plots.

The front-end script was designed not only to satisfy the specifications but also to be easily split up amongst members of the group so that work could take place relatively independently and without sequential dependencies.

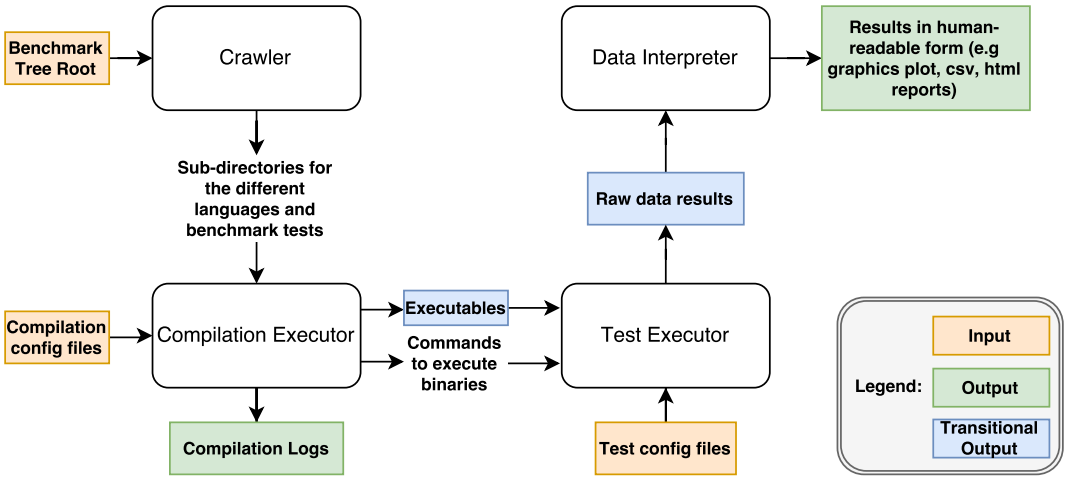


Figure 6: Front-end script component and flow diagram.

5 Methodology

This was a non-standard software development project, consisting of a number of equally weighted, largely independent modules implemented in several languages, and with a research focus. As such, we adopted a bespoke development strategy with inspiration taken from the Scrum and Extreme Programming approaches.

The modular nature of the project allowed us to independently implement each specific benchmark in a given language. As such, our development strategy emphasized accomplishing these subtasks in iterations similar to the sprints of the Scrum development approach. The time required for each varied depending on the complexity of the benchmark, the ability of the programmers, and their familiarity with the language in question.

5.1 Objectives and Considerations

While our chosen development process primarily focused on producing a useful end product quickly, we also considered other aspects of the project. We felt it was crucial for every member in our team to be able to program and understand all the languages used, and collectively analyse and interpret the benchmarking results. A strong understanding of a language is also needed to create benchmarks in it with good performance characteristics. Ideally, the benchmarks should take advantage of all language features to run as quickly and efficiently as that language permits. While this was of course not possible given the limited time available and relative inexperience of all team members, both to

the languages being used and, indeed, programming in general, it was nevertheless important for us to aim for this goal.

The best way to learn a language is to produce programs written in it. We used the development of our benchmarks as the main way to gain language experience, but also dedicated some time to learning the basics of the languages - for example, to help us learn Pony we produced a basic version of a boids simulation[15]. To ensure adequate exposure for all team members, we decided that each member should be directly involved in producing at least one benchmark in each language.

5.2 Testing Methodology

The main purpose of our benchmarking suite project is to compare relative performance between the languages. As such, our testing strategy for the benchmarks emphasized consistent results between languages, rather than being overly concerned with ensuring that they meet an external formal standard of correctness. For example, it was not a priority to ensure that the calculations of the N-Body simulation are scientifically accurate to a high standard. That said, we used a Python graphics module to generate a visualisation of the simulation, providing an intuitive check that it exhibits the expected behaviour of the low-mass planets orbiting around a high-mass star.

To confirm that the implementations of a given benchmark were doing the same calculations across all languages, we compared the results they generated - if at all stages the output values were the same across all languages, this was strong evidence that the implementations were consistent.

The front-end script was more suited to a traditional unit-testing approach. The independent modules were tested individually, and manual system-level tests carried out to ensure correct overall functioning.

5.2.1 Code Reviews

In addition to comparison of results, we performed code reviews to ensure that the overall structure of the programs was consistent between languages. Ideally, we would implement exactly the same program structure in a given benchmark for each language. This was not be possible in practice due to fundamental differences in the languages, but we nevertheless aimed for this goal. In the code reviews, the team that wrote each implementation walked through what the code does so that the rest of the group were satisfied that its design followed the structure diagram for that benchmark.

5.2.2 Unit Tests

Unit tests were used to validate deterministic code - for example, the functions involving vector math in the N-Body simulation. Language specific coverage tools were used to ensure that the tests are as comprehensive as possible. We emphasized method/function and statement/line coverage over branch coverage — these metrics were considered sufficient for benchmarks with tightly controlled input parameters, and issues were encountered with code coverage tools in two of the languages, as discussed below. See appendix A.4 for code coverage data.

5.2.3 N-Body Simulation System-level Testing

As well as unit testing of the vector math, an additional testing step was taken to test the N-Body simulation: the benchmark was run in all of the languages with the same parameters, and the resulting output was saved in text files. These text files were then parsed using a small Python script to determine whether any of the calculations were different (i.e. whether particular lines of output from any of the implementations differed from the others by a pre-set margin). No issues were found, giving us confidence that our implementation was consistent across the languages.

5.2.4 Language Test Frameworks

- **Scala-Akka**

  We used the ScalaTest test suite[16] to create tests, and the code coverage tool built into the IntelliJ IDEA Integrated Development Environment (IDE) to generate coverage reports. More specifically, we adopted a function/method-oriented testing style, using ScalaTest's FunSuite[16]. We also used the Akka TestKit module[7] to test inter-actor interactions, such as passing a message to a specific actor and testing whether the reply from that actor is valid. Unfortunately, the IntelliJ IDEA built-in coverage report generator does not provide information on branch coverage. A coverage tool providing this feature, 'Scoverage', does exist, but does not appear to be actively maintained; problems were encountered trying to run it with the version of Scala being used in this project, so it was ultimately not used.

- **Pony**

  The PonyTest[17] testing framework was used for unit testing. No code-coverage tools exist for Pony, so code coverage metrics are unavailable.

- **Go**

  For testing Go we used the 'testing' package, again choosing a function oriented approach. This package can generate a report of total statement coverage of the tested packages, but no other coverage metrics are provided.

5.2.5 Front-End Testing

Python's built-in unittest module is used for unit tests in the front-end. All the major blocks have unit tests except the plotting sub-module of Data Interpreter, for which tests were not developed due to time constraints. Manual testing was used for system-level testing, as well as the pylint tool for static analysis and code cleanup.

5.3 Development Schedule

While the team members were focused on implementing the benchmarks, we recognized that it was important to maintain involvement from all members throughout the development process. We discussed these tasks at our group meetings:

- Ensuring coherence between benchmark programs in different languages by adopting similar approaches and code structure.
- Documentation of process and the writing of reports.
- Discussing and implementing front-end scripts and command-line interface.

The group met weekly with our supervisors to discuss progress, and scheduled internal meetings on a weekly basis (when permitted by other commitments such as exams and holidays). The team was in constant communication through the use of the Slack messaging platform.

We foresaw that most of the front-end implementation would be pushed towards the later parts of development when we had accomplished a considerable amount of work on the benchmark program subtasks. That said, we recognized that it was a vital part of our product, so the parameters and implementation of the front-end interface were continually discussed as we moved down the development schedule. In practice this iteration and refinement of the front-end continued throughout its development, resulting in feature creep which caused a considerable delay in developing this section of the project, as modules were repeatedly modified to add additional features when we became aware that they would be necessary to meet the requirements we had set.

5.4 Benchmarks

We split our group into three pairs. Each pair specialised in one of the three languages, and had varying levels of exposure to the others, as depicted in table 1. The pair with the highest specialisation in a given language implemented the most challenging benchmark, the N-Body simulation, in it. They also had overall responsibility for all benchmark programs in that language; although they did not code all of them, they were ultimately responsible for ensuring that they worked correctly, and were available to provide assistance to other pairs if they encountered a problem with that language. For example, the Pony-focus pair helped the Go-focus pair implement the waiting room queue for the Pony implementation of the Sleeping Barber problem.

Table 1: Initial Pair Assignments to Language Specialisation

Language specialisation	Karolis & Yerzhan (Pony focus)	Andreas & Simon (Go focus)	Ethan & Jordan (Scala-Akka focus)
Pony	**High**	Medium	Low
Go	Low	**High**	Medium
Scala-Akka	Medium	Low	**High**

In practice the realities of the development process meant that this plan was not strictly adhered to. For example, the Pony language proved more challenging to develop in than Scala-Akka and Go, due to having less pre-existing community support (for example, questions on StackOverflow[1]). This necessitated the Pony focused group spend more time on this language, reducing their exposure to the other languages. Further discussion of the final work allocations are present in section 6.

5.5 Front-end

We initially prototyped the front-end script in Bash. The final final product was written in Python, selected for its ease of use and popularity. Python also makes it easy to implement many of the required functions through either its standard library or the PyPI (Python Package Index). In addition, the use of Python helped us achieve a key objective of the front-end script: to make it easy for other developers and researchers to add new languages and benchmarks. This is achieved by writing language-specific compilation and execution scripts in Python, and test-specific configuration files in plain text.

Benchmark compilation and execution is carried out by the subprocess module from the standard library, and the plotting of results is accomplished using the matplotlib library.

Separating the design into into semi-independent modules greatly improves flexibility of execution, ease of implementation, and testing. In our design, the only stage that cannot be executed stand-alone is the Crawler stage - this is because we could not see a use-case where only the data generated by the Crawler would be useful in itself. Execution of a specific stage of the front-end can be specified by passing command-line flags. For a discussion of some use cases of these separated modules, see the user manual in appendix A.2.

5.6 Model and Language

As previously mentioned, a serious challenge we faced was how to tackle the differences between the languages we used. Unlike the Savina suite, which created benchmarks in languages that run on the JVM and use the actor model, the three languages we used show considerable differences. One example of this is how they handle the termination of actors.

In Scala-Akka, actors must be terminated explicitly, either individually or as an entire 'actor system'. Execution will not end until all actors have been terminated, regardless of whether they are executing

[1] As of May 15 2017: 22,335 questions tagged 'go', 65,873 questions tagged 'scala' , 5,273 questions tagged 'akka', 12 questions tagged 'ponylang'

any code. In Pony, actors automatically terminate when they finish execution of their code, and the program ends when all actors have terminated. In Go, execution halts when the `main` function terminates, regardless of what any active goroutines are doing. In order to use multiple goroutines, `main` must be kept alive, for example by blocking on a response from a goroutine. See appendix A.3 for code demonstrating these differences.

In the course of developing the benchmarks for the different languages, we observed that asynchronous message passing is a central concept shared by both the Actor and CSP models of concurrency. As such, here we provide an illustrative example of the message passing structure and syntax of the three languages using code snippets from the Sleeping Barber benchmark. Additionally, there are data-race freedom features in the Pony language that restrict the data types that can be passed as messages. As such we include a discussion on the Pony capabilities system using the N-Body benchmark.

5.6.1 Scala-Akka

In Scala-Akka, sending a message to an actor uses the following syntax:

```
RecipientActor ! Message
```

where message is declared as a case class or case object.

A case class is used to represent a message if the message contains arguments, and a case object is used if the message contains no arguments. Case classes allow one to create immutable objects whose structure may vary, while a case object defines a singleton object with no arguments, and both are decomposable through pattern matching.

```
//Message definitions
case class Haircut(customer: ActorRef, shop: ActorRef)
case object Wait
case object Exit

//Actor definition
class BarberActor ... {
   def receive = {
      case Haircut(customer, shop) => {
         //Message Customer actor that the haircut is starting
         customer ! Begin
         //Simulate time taken for haircut
         busyWait(...)
         //Message Customer actor that the haircut has finished
         customer ! End
         //Message Shop actor to request the next customer
         shop ! Next
      }
      case Wait => {
         //Do nothing - Shop actor sends a Wait message if there are no customers
      }
      case Exit => {
         //Terminate actor
         context.stop(self)
      }
   }
}
```

'Behavior response' is a function defining the actions to be taken by an actor in reaction to receiving a message. Every time a message is sent to an actor in Scala-Akka, it is matched against the currently defined set of behavior responses of that actor using the syntax `def receive = { ... }`.

In the Sleeping Barber problem, we show how the Barber actor responds to three different kind of messages: `Haircut(customer, shop)`, `Wait`, and `Exit`. A `Haircut` message is defined as a case class, as each `Haircut` message contains a different ActorRef value representing a unique Customer actor,

which requires it to be pattern matched. When the Barber actor receives a `Haircut` message (from the Shop actor), it can then respond by starting the haircut process on the particular Customer actor specified in the message. When the Barber actor receives a `Wait` message, it does nothing, and when it receives an `Exit` message, it terminators its own process. As such, the `Wait` and `Exit` messages are represented as case objects since they take no arguments and require no pattern matching.

5.6.2 Pony

Message passing is implemented in a similar way in Pony for the Sleeping Barber benchmark, but instead of explicitly passing a message to the actor, a method-like asynchronous behaviour is called. This approach provides a different intuition to message passing — instead of thinking of a single mailbox per actor as in Scala-Akka, it is now possible to think of multiple mailboxes, one per behaviour.

```
actor Barber
   be enter(customer: Customer, room: WaitingRoom) =>
      //Call 'start' behavior on Customer actor
      customer.start()
      busyWaiting(...)
      //Call 'done' behavior on Customer actor
      customer.done()
      //Call 'next' behavior on Room actor
      room.next()
   be wait() =>
      //Do nothing

   //Terminate actor
   be exit() => true
```

In Pony however, there is an added consideration of the data type that is passed. Pony has a unique reference capabilities feature that ensures safety when passing mutable data between actors and sharing immutable data amongst actors.

For example, in the N-Body implementation in Pony, state variables (variables that hold positions and velocities of bodies) in the World actor need to be copied into an immutable type — this adds overhead, but guarantees that race conditions between actors are impossible. An example of creating an immutable copy from a mutable variable — an array of immutable unsigned integers (`val` capability means that a variable is immutable) — follows:

```
fun build_sendable(non_sendable: Array[U32 val]): Array[U32 val] val =>
    let array_iso = recover iso Array[U32 val] end
    array_iso.reserve(array.size())
    for value in array.values() do
       array_iso.push(value)
    end

    consume array_iso
```

This function copies the values to a mutable but isolated variable, then changes the variable to become immutable. This adds both computational and memory overhead, but the guarantees of safety offset this cost. Similar function would have to be called on any mutable object that is shared amongst the Body actors.

However, for the N-Body simulation there is another issue with Pony's capabilities: when receiving updates from the bodies, we need to use the same reference, but Pony's safety features won't allow us to access an empty reference. In fact, the compiler won't allow us to consume a field at all (even if it is an `iso` field). This is because it thinks that some other code (within the same actor) accessing this field might find it empty, and therefore crash. In order to solve this, we can create a new 'placeholder' isolated object and consume it, instead of consuming a field. Here, we can make use of a feature of Pony called 'destructive read':

```
actor Bar
   var data_field: ClassA iso
   fun foo(data_param: ClassA iso) =>
      var data_local: ClassA val = data_field = consume data_param
```

In Pony, whenever an assignment is made, the old value is returned. Using destructive reads we can effectively swap object references. In the example above we consume `data_param` and assign it to the field `data_field`. At the same time, we assign the old value of `data_field` to `data_local`, which is immutable and can be shared.

Overall, using these techniques in World actor we are able to implement N-Body in Pony that is consistent with Scala-Akka and Go for performance benchmarking purposes.

5.6.3 Go

Rather than implicit mailboxes, goroutines pass messages through explicitly declared channels. These typically accept a specific data type, though they can also be declared as type `interface` to accept arbitrary data types.

```
//Message definitions
const WAIT int = 1
const BEGIN int = 2
const NEXT int = 3
const EXIT int = 4

//Channel declaration for communication between barber and shop
b.ch_barber = make(chan interface{})

//Message receiving function of the barber goroutine
func(b *Barber) receive(){
   //Loop listening for customers
   for {
      //Read from channel into 'msg' variable
      msg := <-b.ch_barber
      switch msg.(type)  {
      //Customer has arrived
      case *Customer:
         //Extract customer message channel
         customer := msg.(*Customer)
         customer.cust <- BEGIN
         busyWait(...)
         //Signal customer that haircut is complete by closing its channel
         close(customer.cust)
         b.ch_barber <- NEXT
      //Message from shop
      case int:
         //Extract message
         msg = msg.(int)
         if(msg == WAIT){
            //Do nothing
         } else if (msg == EXIT) {
            //Terminate goroutine
            return
         }
      }
   }
}
```

The receive method of the barber goroutine listens on a channel which can accept any value, and performs pattern matching with the switch statement. If the message is a pointer to a customer, the communication channel to that customer is extracted and messages are sent down it. If the message is an integer, its value is extracted and the barber reacts according to this value.

Note that the syntax is considerably more complex than in either Scala-Akka or Pony. This reflects the fact that channels were not designed to function in the same manner as actor mailboxes. While they can be used to carry out the same role, they are lower level constructs and so more details of their usage must be explicitly defined in order to utilise them in this way.

As a point of comparison, and to investigate the performance overhead of emulating the actor model, we also developed an alternative implementation of the Sleeping Barber benchmark not constrained to so closely follow the actor model version. In this version, for example, rather than the 'shop' actor maintaining a queue of customers, we use a buffered channel to which customers attempt to write. If they successfully write to the channel, they are in the shop waiting room. The shop actor can then attempt to read from this channel, blocking while it is empty, and passing customers to the barber when it successfully reads them from the channel. This implementation is considerably cleaner than that shown above, and also shows a performance improvement – this is discussed in more detail in section 7.

6 Group Work

We initially divided the team into language specialisations as shown in table 1, and intended for each pair to implement one benchmark in each language. However, due to the varying development effort required and the increasing requirements for the front-end, this plan was not fully implemented. The N-Body simulation, as the most challenging benchmark, was tackled in each language by the group with ownership of that language, but the other benchmarks were split out differently. The development of the N-Body simulation in Pony took longer than that in the other languages, giving the Pony focused team less opportunity to work on other benchmarks.

Although not explicitly planned, different group members also took responsibility for each benchmark, deciding on what structure the overall design should follow when multiple choices were available, and guiding the implementations.

The modules of the front-end were also divided between group members, with Karolis responsible for overall design and integration. Scripts for compiling the languages, used by the compilation executor, were created by members proficient in those languages.

Information on which group members completed which tasks is presented in table 2. Karolis was the group leader and documentation editor, and Ethan kept the meeting logbook and coordinated report writing.

Table 2: Final Group Work Allocation

Group Member	**Benchmark (Language)**	**Front-end Script**
Andreas	N-Body (overall)	Data Interpreter - Plotter
	N-Body (Go)	N-Body visualisation tool
Ethan	Sleeping Barber (overall)	Data Interpreter
	Sleeping Barber (Scala-Akka)	
	Sleeping Barber (Go)	
	N-Body (Scala-Akka)	
Jordan	N-Body (overall)	Test Crawler
	N-Body (Scala-Akka)	Scala-Akka compile script
	Ping-Pong (Scala-Akka)	
Karolis	N-Body (Pony)	Front-end Script (overall)
		Pony compile script
Simon	Ping-Pong (overall)	Data Interpreter - Plotter
	Ping-Pong (Go)	Go compile script
	Sleeping Barber (Pony)	
	N-Body (Go)	
Yerzhan	Sleeping Barber (Pony)	Test Executor
	N-Body (Pony)	

By far the biggest challenge in our group work was managing the many different aspects of the project and coordinating the workflow between members. When developing the benchmarks, ensuring consistency between the languages required each member to provide frequent updates and communicate their progress to the group clearly. Coordinating the communication of the individual modules in the front-end script work also took considerable coordinate, made harder by the changing specifications for this aspect of the project. This is discussed further in the section 7.

We also saw the benefits of writing clean code, as we had to constantly check what each member had written, especially when debugging each other's code and linking the front-end modules together.

7 Results and Reflections

7.1 Front-end

Development of the front-end script proved to be more challenging than first anticipated. The main hurdle was underestimation of the time required to complete it — the design and implementation of the script was left until the last month of the project, which proved barely enough time to implement all required features. As it stands, some aspects could use improvements if more time was available. One example is the dependencies on Python graphics modules (see appendix A.1 for dependencies) which prevent the Data Interpreter module from being run on headless systems.

The bulk of development time was spent on integrating the discrete modules into a working system — due to simplistic initial specifications, no per-module interface specifications had been deemed necessary, resulting in different interpretations of the specification and causing bugs. In addition, features were added ad-hoc to allow for use-cases discovered while using the script. Even though this growth resulted in a useful system, it was not a sustainable approach. In hindsight, the either the front-end script should have had a more concrete specification with no additional features allowed, or a non-trivial prototype should have been used to properly estimate usability and requirements before finalising the specification.

While we spent more time than anticipated incorporating additional features, they ultimately proved useful. For example, the plotting functionality let us quickly iterate through different benchmark inputs to find sets of test cases that provided useful data. Many of the plots in this report were automatically generated.

7.2 Benchmark Results

7.2.1 System Specifications

We ran the benchmarks on several systems, including the Imperial College Department of Computing (DoC) laboratory computers and batch servers, and our personal machines, but the results presented here were generated on five systems, referred to hereafter by the names in table 3. One of the systems was running a virtual machine (VM) in order to vary the number of CPU execution threads available to the benchmark programs.

Table 3: Machine Specifications

Machine ID	Description	Processor	RAM	OS
Lab-8T	DoC Lab computer	Intel i7-4790, 64-bit, 3.6GHz, 4 cores, 8 threads	16 GB	Ubuntu 16.04.2
Lab-VM-nT	DoC Lab computer (VM, n-threads)	Intel i7-4790, 64-bit, 3.6GHz, 4 cores, 8 threads	8 GB	Ubuntu 17.04
DevX-VM-2T	Developer laptop ThinkPad X230 (VM, 2 threads)	Intel i5-3320M, 64-bit, 2.6GHz, 2 cores, 4 threads	4 GB	Ubuntu 16.04.1
DevT-4T	Developer laptop ThinkPad T430	Intel i5-3320M, 64-bit, 2.6GHz, 2 cores, 4 threads	8 GB	Slackware Linux 14.2
DevMac-4T	Developer laptop MacBook Pro Core i5 2.4 13” Late 2013	Intel i5-4258U, 64-bit, 2.4GHz, 2 cores, 4 threads	4 GB	MacOS Sierra 10.12.4

7.2.2 Performance Expectations

According to Amdahl's law of parallelisation[18], if P is the proportion of a program that can be made parallel, and $1 - P$ is the proportion that remains serial, the maximum speedup that can be achieved using N processors is $\frac{1}{(1-P)+(P/N)}$.

For the Ping Pong Test and Sleeping Barber problems, the portions of the programs that can be made parallel are minimal, so as one would expect, we observe no significant speedup as the number of processor cores increase. While the Ping Pong players and Sleeping Barber customers are actors/goroutines that can exist on different threads, the computational steps they can perform independently of the rest of the system are relatively insignificant, hence the proportion of total execution time they occupy (and that can be made parallel) with respect to the overall execution time is very small. The only exception to this is the busy-waits performed by the customer factory and barber actors, which do not rely on inter-actor communication and can thus run in parallel.

On the other hand, in the N-Body simulation, each body needs to perform a significant number of computational steps independently of the rest of the system. The proportion of time that these actors/goroutines spend that can be made parallel is thus significant in the overall execution time of the program. The theoretical speedup from increased parallel execution is therefore meaningful in this case. Given that there are a significant number of bodies specified in the benchmark, we expect an exponential, non-linear speed up with an increase in CPU execution threads, as described in section 4.1.3. The actual increase we see should depend on the proportion of time taken to perform these calculations versus that required for inter-actor communication.

It is also worth noting that, due to the Hyper-threading technology present in modern Intel processors[19], CPU execution thread count differs from physical core count. Tasks running on two virtual threads sharing a single physical core have the potential to be slower than those running on two separate cores due to physical resource contention. Therefore we might expect performance scaling to depend on core count rather than thread count under some workloads.

7.2.3 Machine performance

Presented here are representative figures of each benchmark test case on each machine configuration. These results demonstrate the relative performance of the machines.

Table 4: Ping-Pong Test Results

Machine ID	Scala-Akka		Pony		Go	
	n=low	n=high	n=low	n=high	n=low	n=high
Lab-8T	2.30	32.21	0.47	9.42	0.34	7.42
Lab-VM-4T	5.55	93.98	0.44	8.81	0.37	7.58
DevX-VM-2T	3.28	34.01	0.44	9.00	0.48	9.66
DevT-4T	3.07	37.19	0.19	4.91	0.52	10.68
DevMac-4T	5.26	83.85	0.35	6.92	0.67	11.32

variable n = no. of messages (low: 1,000,000, high: 20,000,000)

Table 5: Sleeping Barber Problem Results

Machine ID	Scala-Akka		Pony		Go	
	n=low	n=high	n=low	n=high	n=low	n=high
Lab-8T	3.57	3.19	0.15	0.42	0.03	0.21
Lab-VM-4T	2.03	5.19	0.10	0.27	0.03	0.22
DevX-VM-2T	2.58	23.77	13.79	11.54	0.05	0.35
DevT-4T	1.84	3.63	0.07	0.59	0.03	0.27
DevMac-4T	2.53	15.85	0.15	2.14	0.05	0.39

variable n = no. of haircuts (low: 5,000, high: 45,000)

Table 6: N-Body Simulation (changing no. of bodies) Results

Machine ID	Scala-Akka		Pony		Go	
	n=low	n=high	n=low	n=high	n=low	n=high
Lab-8T	3.47	564.20	1.86	49.47	0.80	52.26
Lab-VM-4T	5.91	798.21	2.14	60.84	0.83	130.65
DevX-VM-2T	10.26	1584.28	4.28	407.62	1.24	199.56
DevT-4T	8.70	1076.69	2.68	98.33	1.15	109.34
DevMac-4T	7.62	1129.61	2.78	106.01	0.06	116.56

fixed no. of steps = 100,000
variable n = no. of bodies (low: 5, high: 95)

Table 7: N-Body Simulation (changing no. of steps) Results

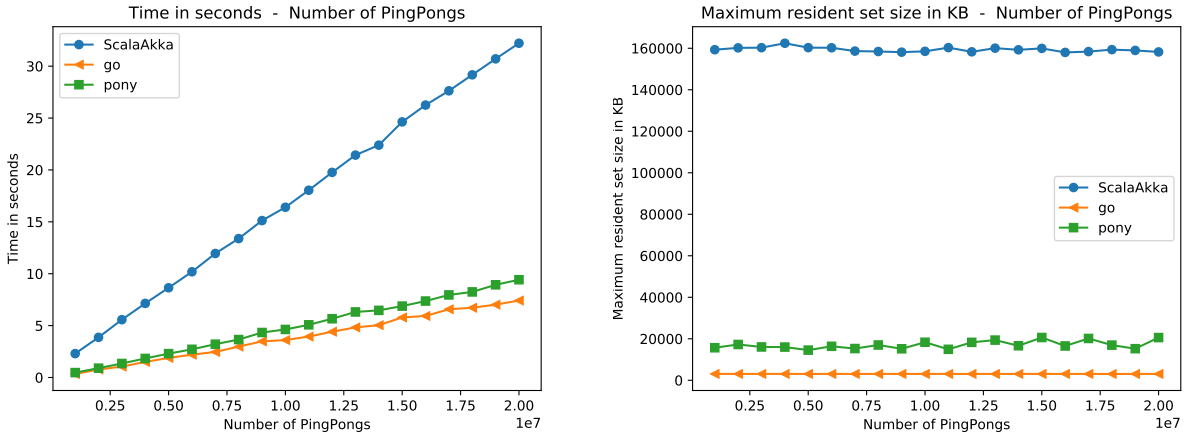
Machine ID	Scala-Akka		Pony		Go	
	n=low	n=high	n=low	n=high	n=low	n=high
Lab-8T	18.70	167.00	2.69	21.71	2.79	22.59
Lab-VM-4T	25.99	232.35	3.11	26.13	4.65	40.66
DevX-VM-2T	54.84	462.01	16.68	146.86	7.19	65.12
DevT-4T	35.26	319.33	4.40	37.46	4.74	40.51
DevMac-4T	38.21	351.94	4.65	39.49	4.08	35.05

fixed no. of bodies = 50
variable n = no. of steps (low: 10,000, high: 100,000)

These preliminary results suggest that the Ping-Pong test (table 4) behaves primarily as expected, with the exception of some unusual results from the Scala-Akka implementation. The Sleeping Barber problem (table 5) again shows some variation in Scala-Akka, as well as an anomalous result in Pony when running on a dual-threaded machine, discussed in further detail in section 7.2.5. The N-Body simulation (tables 6 and 7) behaves as we would expect, showing highly thread-dependant performance.

These figures also demonstrate the overall trend in performance differences between the three languages.

7.2.4 Ping-Pong Test



(a) Execution time

(b) Memory usage

Figure 7: Ping-Pong test across languages. Run on Lab-8T.

As anticipated, in this benchmark we observe a linear relationship between the total number of messages passed and the execution time, as seen in figure 7a. Performance in Pony and Go is quite similar over this range of values, while Scala-Akka execution takes significantly longer.

Although we might expect an initial difference in execution time between Scala-Akka and the other languages due to the overhead involved in starting the JVM, this overhead should be constant across all runs of the benchmark, so we are able to account for it in our analysis of the results. As we can clearly observe a steeper gradient for the Scala-Akka data points, the performance penalty is not due to this factor.

Given Scala-Akka is run on the JVM and compiled to bytecode, we might also expect it to demonstrate lesser performance than Go and Pony, both of which are compiled to machine code. This observation holds true through all the results we have obtained. However, in this particular test the calculation performed is minimal, suggesting that the performance difference observed here may alternately reflect higher message passing overheads.

The second parameter examined was the peak memory usage for each language, shown in figure 7b. As expected the peak memory usage stays unchanged, since there are no memory dependencies other than the number of actors involved, which remains constant. However, Scala-Akka memory usage seems to be much higher than the other languages, at 160MB. This is likely due to the memory used by the JVM garbage collector, which reserves additional memory beyond the current program needs. Due to this factor, the maximum resident set size measurement may not be a useful metric for comparing Scala-Akka with the other two languages.

Varying the number of CPU execution threads (figure 8) shows some interesting results. Given that this is a purely sequential benchmark, we would expect changing the number of CPU threads to have no effect on performance. While this is indeed observed in Go, in both Scala-Akka and Pony we instead see that execution time significantly increases when more execution threads are made available.

Our initial theory for this unexpected behaviour was that it was an artefact of the VM environment. However, after analysing the data from DevT-4T and Lab-8T we concluded that the same behaviour occurs even when this factor is eliminated. We therefore have no current theory to explain this observation, which strongly encourages further investigation.

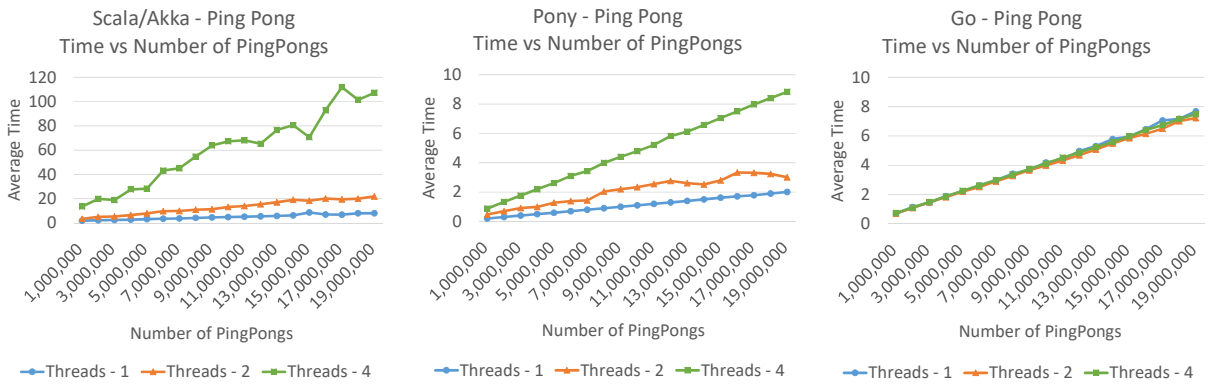
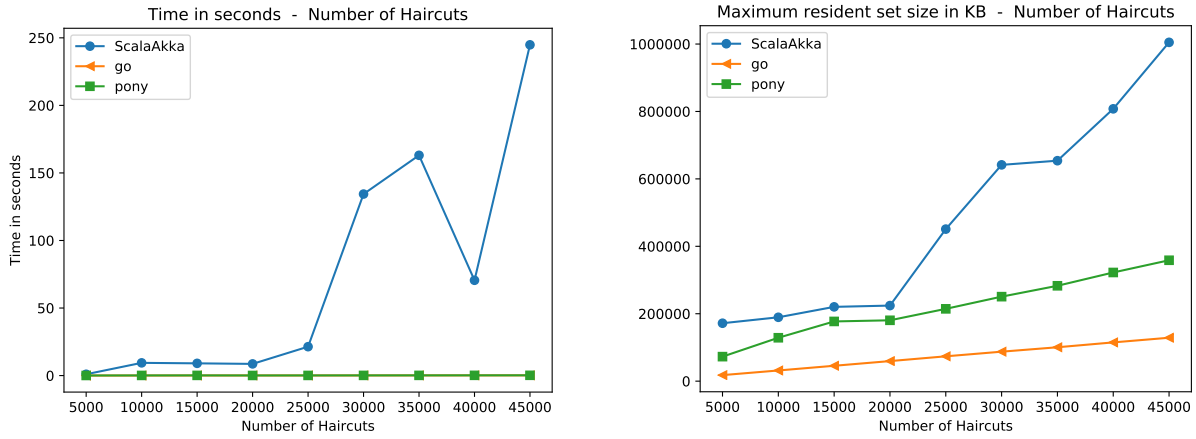


Figure 8: Execution time of Ping-Pong in all three languages with varying CPU execution threads. Run on Lab-VM-1T, 2T, and 4T.

7.2.5 Sleeping Barber Problem



(a) Execution time

(b) Memory usage

Figure 9: Sleeping Barber problem across languages. Run on Lab-8T.

These results display a similar pattern to those for the Ping-Pong test. Execution speed (figure 9a) in Go and Pony shows the expected small rise as total haircuts increases, with Go performing slightly better than Pony. This performance difference is dwarfed, however, by Scala-Akka. Here we observe a rather concerning pattern of execution time rising precipitously past 25000 total haircuts, suggesting a step change in the performance characteristics of this implementation past this value.

Memory usage (figure 9b) in both Pony and Go increases linearly with number of haircuts. This is an expected behaviour, as more actors will consume more memory. In Scala-Akka, however, though we see a similar small increase (possibly masked by the JVM memory allocation characteristics) as the total number of customers rises to 20000, past this value we again see a significant change in the memory consumption profile.

The performance of Scala-Akka with changing number of CPU threads (figure 10) is noisy, so while it is suggestive of better performance with a greater number of threads, we draw no conclusions from it. Performance in Pony shows a small improvement when three threads are available compared to four, while performance with only two threads is order of magnitude worse. Results are not presented for one thread as execution consistently failed due to memory exhaustion in this test case. Finally, Go displays the non-intuitive result of four CPU threads giving the worst performance, though the difference as thread count varies is small.

Taken together, these results suggest that there may be significant difference in our implementations of this benchmark across the languages. If, however, this is not the case, they instead reveal a striking disparity between the three languages for this particular workload.

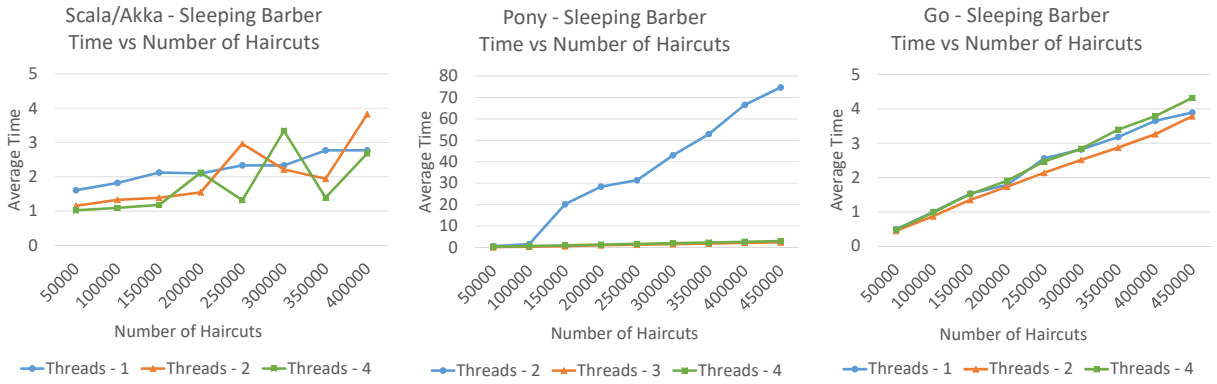


Figure 10: Execution time of the Sleeping Barber problem in all three languages with varying CPU execution threads. Scala-Akka and Go run on Lab-VM-1T, 2T, and 4T, Pony on 2T, 3T, and 4T.

The implementation in Go produced good performance compared to Pony and Scala-Akka, but focused too heavily on matching the actor model implementations in the other languages, neglecting useful language features found in Go. Therefore, we developed an alternate version of this benchmark solely in Go. This version does not attempt to mimic the structure diagram presented in section 4.1.2, freeing it to better utilise the features of Go. For example, rather than the 'shop' actor maintaining a queue of customers, we use a buffered channel to which customers attempt to write. If they successfully write to the channel, they are in the shop waiting room. The shop actor can then attempt to read from this channel, blocking while it is empty, and passing customers to the barber when it successfully reads them from the channel.

Another major simplification involved the treatment of customers turned away from a full shop: In the Savina-based version, upon being turned away from a full shop customers signal to the factory actor, which then immediately sends them back to the shop. This eliminates the need to delete and create new customers each time this occurs, reducing thread spawning and memory allocation overhead by reusing existing customers. The alternate implementation, however, does not use this optimisation. Instead, if a customer is turned away from the shop, it immediately terminates. The factory must therefore continue to produce new customers until the required number of haircuts have occurred, rather than creating a fixed number initially and waiting for them all to successfully enter the shop. The factory executes a loop in which it checks whether any customers have completed a haircut and, if they have not, spawns a new customer and sends it to the shop. A casual consideration of this situation might suggest that, due to repeated initialisation of new goroutines as customers are spawned and terminate, performance would be worse than when a fixed pool of threads is maintained. However, when the benchmarks are run with identical arguments, we actually observe that the alternate implementation exhibits approximately 50% better performance (figure 11). A detailed investigation suggests that this observation could be a result of the handling of goroutines:

Although goroutines are cheap, each thread of execution runs only a single one at a time. In the first implementation, it is possible for a customer to be turned away from a full shop, report its failure to the factory, and then immediately attempt to re-enter the full shop. While executing this loop, the customer effectively starves other customers waiting to execute on the same thread, while not advancing the state of the program. In the alternate implementation, if a goroutine fails to enter the shop, it immediately terminates and allows a new customer to take over the thread. This increased cycling of threads improves the chance that a thread capable of usefully advancing the state of the program, for example a customer that has been sent to the barber, will run.

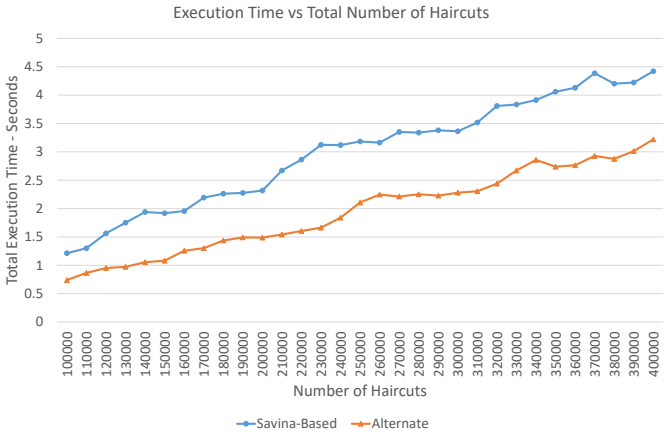
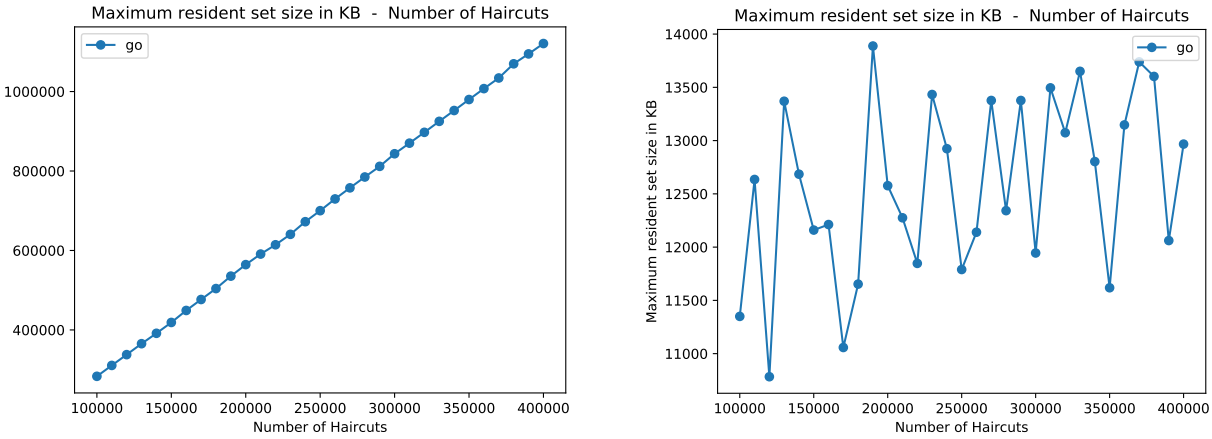


Figure 11: Comparison of execution time for Savina-based and alternate implementations of the Sleeping Barber problem in Go. Run on Dev-VM-2T.

Another possible explanation for this significant performance advantage is that, in the Savina-based implementation, all customers are spawned as soon as program execution begins, and only after this point does the factory begin listening for returning customers. In the alternate implementation, listening for returns and producing customers occur semi-concurrently, with either one or the other occurring in each loop iteration. The result of this change on execution is clearly demonstrated by the dramatic difference in the memory profiles of the two benchmarks: Figure 12a shows the maximum memory used by the initial implementation - as all customers are created at once, and do not terminate until customer generation is complete and the factory begins listening for returns, the peak memory usage increases linearly with the number of customers generated. Figure 12b shows the same data for the alternate implementation. As customers can and do terminate while new ones are being generated, the peak number of customers existing at any one time, and thus peak memory consumption, depends entirely on the order in which the different goroutines execute, which appears to be effectively random. An overall trend of increasing peak memory consumption, while still visible, is almost completely obscured in this small dataset by the stochastic variation.



(a) Initial implementation.

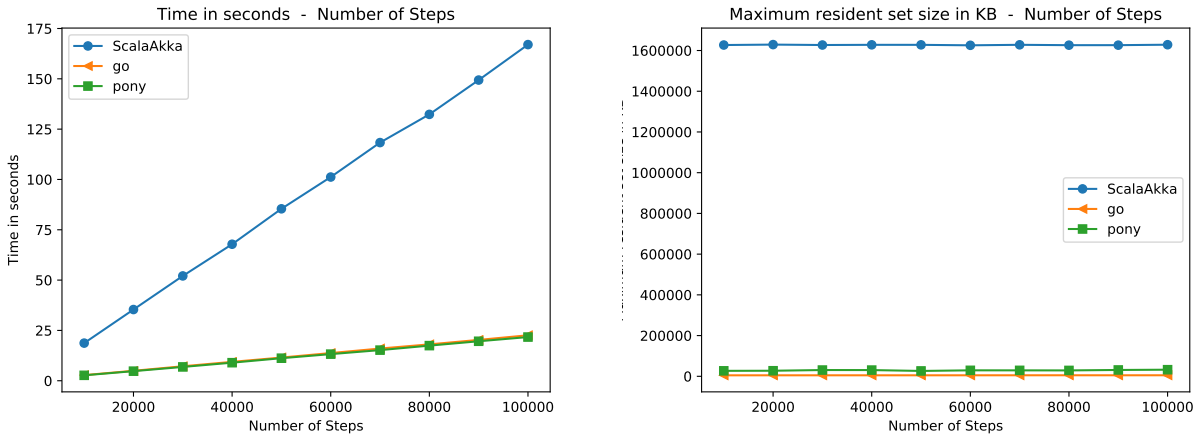
(b) Alternate implementation

Figure 12: Maximum memory consumption of Sleeping Barber problem. Run on Dev-VM-2T.

Despite these clear differences in memory execution profile, further investigation suggests that this aspect may not be the main source of the observed performance improvement: When the listening and customer production were separated into two different goroutines, thus permitting true parallel execution of these tasks rather than forcing them to share a single thread, no significant change in performance was observed on either a 2 or 64 thread machine.

7.2.6 The N-Body Simulation

The N-Body simulation benchmark is designed to test the performance of parallel numerical calculations, message passing, and the management of a large number of actors.



(a) Execution time

(b) Memory usage

Figure 13: N-Body across languages as number of steps is varied. Run on Lab-8T.

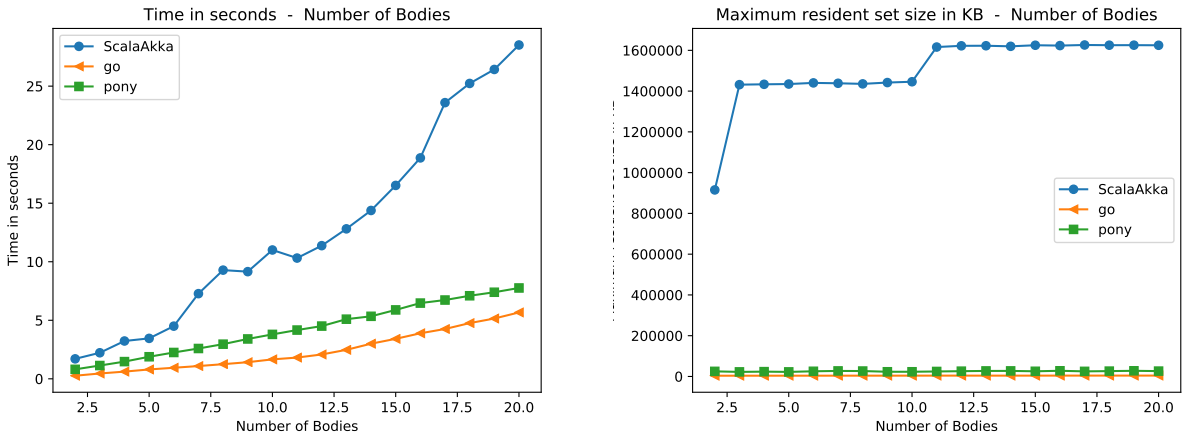
The first test we conducted held the number of bodies constant, varying total step count. The results for the execution time (figure 13a) show that the performance is purely linear for all the languages. Again the runtime for Scala-Akka is significantly higher than the other two languages, whose performance is almost equal. One explanation is that there is some overhead dependant on the total number of steps executed present in the Scala-Akka implementation not found in the others, but as this result is consistent with that of the other benchmarks, it more likely represents the inherent performance characteristics of the language.

Memory consumption (figure 13b) shows the same pattern as seen in the Ping-Pong test, again matching our expectations of no change in memory requirements when the total number of actors remains constant.

The second test held total step count at 100,000 and varied the number of bodies from 2 to 20. As seen in figure 14a, Scala-Akka once again has a higher runtime in comparison to the other 2 languages. In addition, Scala-Akka's overall behaviour is very different than the other two languages. Its runtime starts with a linear increase as the number of bodies increases from 2 to 5, but then rapidly rises and fluctuates.

On the other hand, the runtimes of the other languages seem to be much lower and demonstrate similar behaviours. Go is faster than Pony for this test space but the two of them seem to converge. However, due to the significantly greater runtime of Scala-Akka, this plot is insufficient for comparing Go and Pony's performance. This point, in combination with the convergence of the two curves, invites more detailed investigation into the performance of these two languages, carried out in the third test.

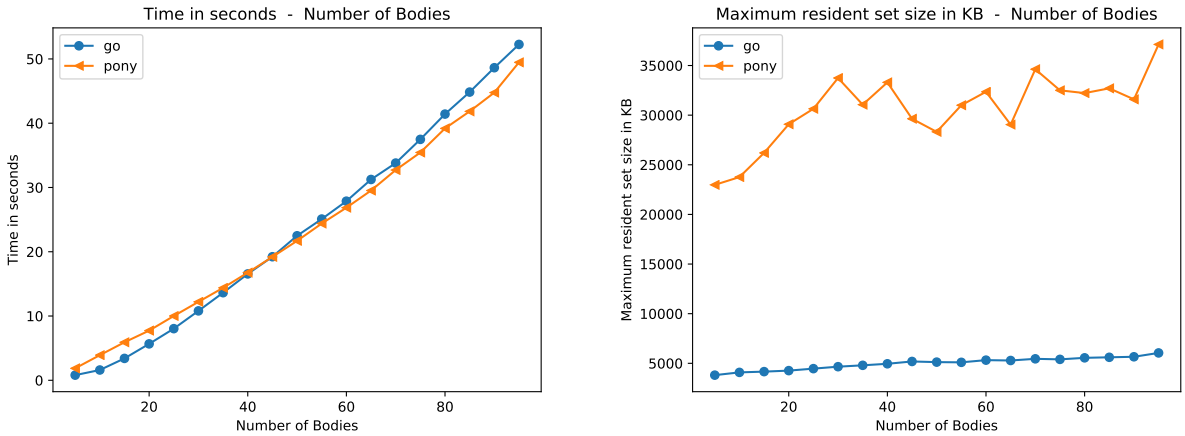
Similarly to the first test, the results here (figure 14b) show that the peak memory usage is constant as the number of bodies increases between the three languages. While this does not match our expectations of increased memory usage as the total number of actors rise, this effect is observed in the third test (figure 15b) when the number of actors has a higher range. As mentioned before, Scala-Akka has a much higher apparent memory usage due to the JVM garbage collector. Apart from this there is a discrepancy in Scala-Akka's results — memory usage shows step changes rather than a smooth rise. This may again be a result of the memory allocated to the garbage collector increasing by fixed amounts past certain thresholds.



(a) Execution time (b) Memory usage

Figure 14: N-Body across languages as number of number of bodies is varied. Run on Lab-8T.

The third test is a further study on the performance of Go and Pony for 5 to 95 bodies. Figure 15a illustrates the runtime performance of those two languages and provides an interesting observation: Even though Go is almost two times faster for a small number of Bodies, Pony surpasses it for $N > 45$ and remains faster up to $N = 95$. This could lead to the conclusion that Pony is better able to handle a large number of actors, though again could be reflective of implementation, rather than language, differences.



(a) Execution time (b) Memory usage

Figure 15: N-Body across Pony and Go as number of steps is varied. Run on Lab-8T.

This test can also show the memory usage difference between Go and Pony in more detail. Figure 15b shows that there is a large difference (an order of magnitude) between the language memory usage. Go seems to be using very little memory which linearly increases at a low rate as the number of bodies rises, while Pony shows fluctuating memory usage with an apparently random pattern.

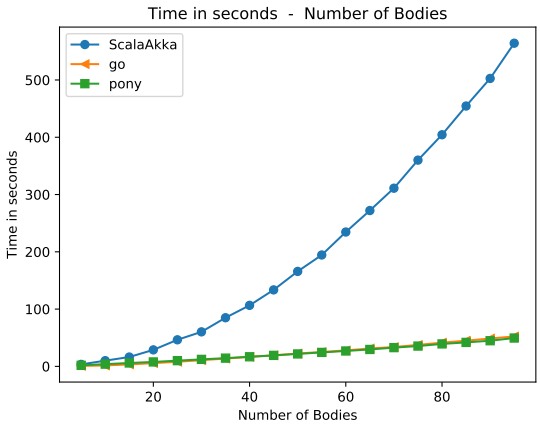


Figure 16: Execution time of N-Body across languages as number of bodies is varied. Run on Lab-8T.

Our fourth test includes the performance of all 3 languages in the same test space of test three, giving an overall picture of the runtimes. The results (figure 16) show that Scala-Akka is approximately two orders of magnitude slower than Pony and Go at $N = 95$.

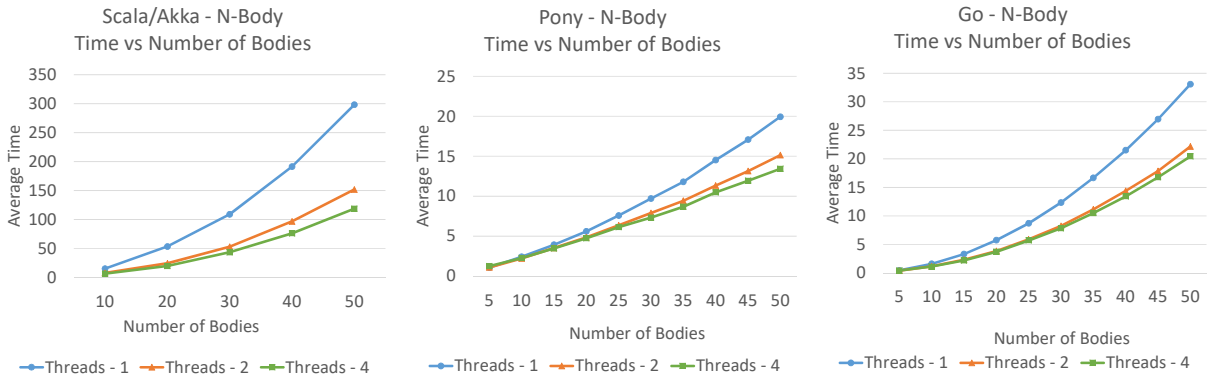


Figure 17: Execution time of The N-Body simulation in all three languages with varying CPU execution threads. Run on Lab-VM-1T, 2T, and 4T.

Figure 17 shows the performance scaling as CPU thread count is changed. Here we observe the anticipated effect of increased performance as thread count rises in all cases. However, the performance scaling is not as dramatic as we might have hoped. While Scala-Akka shows a doubling in performance when thread count rises from one to two, Pony and Go show an approximate increase of only 25%. The improvement when available threads rise from two to four, while still consistently observed, is small.

This suggests that the proportion of time spent performing parallel calculations in insufficient to accurately gauge the relative extent to which parallelism improves performance across the languages. If we had more time, this issue could be addressed by increasing the execution costs of the parallel segments of our code — the position calculations performed independently by the bodies.

7.2.7 Visualisation of N-Body Simulation

In order to have a good picture of the N-Body results and visually confirm the correctness of the simulation, we developed a visualisation Python script, based on graphics.py[20]. The script reads a film file containing all the necessary data for the graphical representation of the system's motion.

When the film file is read, the script generates an animation of the planetary motion. Since the N-Body simulation is 3D, the script has a 2D and a 3D version. The 2D version is appropriate only in the case of motions on a 2D plane, but allows for an efficient investigation on whether the dynamics work correctly.

The 3D version is useful when the movements take place in the 3-dimensional space. The extra functionality is provided by geometrical transformations which give a sense of a third dimension, varying the radius of each body as a function of body mass and z coordinate.

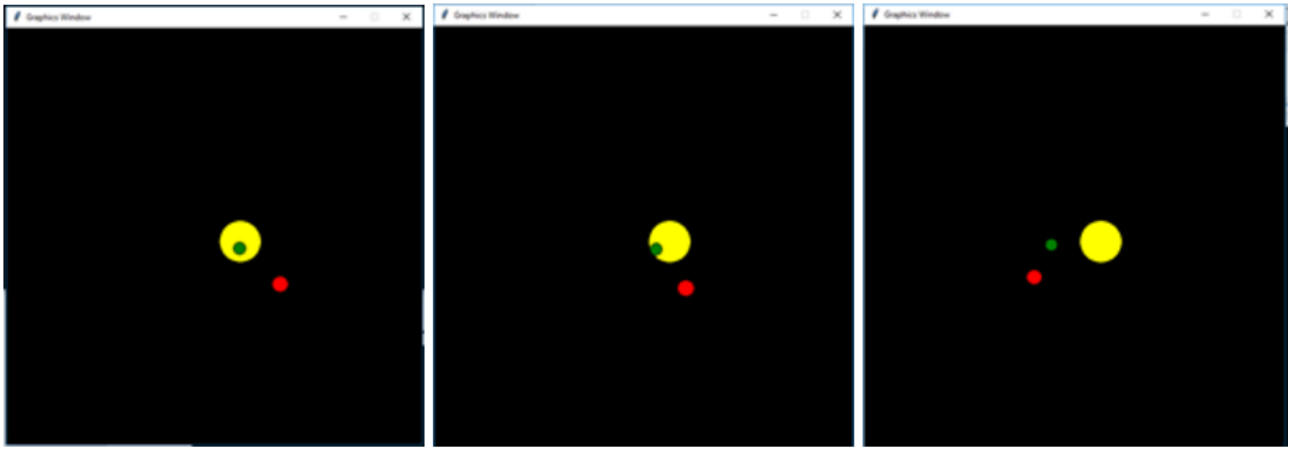


Figure 18: 3D visualisation of N-Body simulation.

7.3 Language Discussion

7.3.1 Scala-Akka

Writing actor model programs in Scala using the Akka framework was easy and fairly intuitive. There are no restrictive features necessitating complex workarounds. The syntax for message passing is simple, and the ability to represent messages as case classes or objects is a powerful feature permitting complex data structures to be sent in messages trivially.

However, it is clear that this convenience comes with a performance penalty - Scala-Akka implementations consistently performed worse than those in Pony and Go, sometimes by a dramatic margin. While writing concurrent code may be easy in Scala-Akka, it appears to be much less suited to performance critical tasks. Even if these major performance differences are due to our group's inexperience with this language, we are equally inexperienced with the other languages. From this we can conclude that, even if it is possible to obtain good performance in Scala-Akka, this is much more challenging than doing so in Pony or Go.

Additionally, other aspects of Scala-Akka development leave a lot to be desired. While it is possible to compile and run programs without using a framework known as SBT (simple build tool), up-to-date documentation for achieving this was not available. Configuring and running SBT was very challenging and time-consuming, and both compiling and running compiled code using it introduced significant overhead, making it unsuitable for performance measurements. This necessitated compiling to standalone executables, a complex and time consuming process.

7.3.2 Pony

Writing Pony actor model programs has a steep learning curve due to the unique security capability features which guarantee deadlock safety and data-race freedom — every time a variable is declared its reference capabilities have to be considered (these compile-time checks also result in slightly longer

compilation times compared to Go, though for our small benchmarks this was not significant). Once past the initial learning curve, however, one can quickly see the benefits. Overall, the language is modern and it is very convenient for concurrent programming.

On the other hand, Pony has not yet gained as much popularity as Go and Scala-Akka, resulting in a paucity of learning resources available online. This proved quite challenging when learning of the language, though the official tutorial is helpful. Despite the limited third-party learning resources, the official Pony team on github and IRC were helpful in answering our queries and addressing newly discovered bugs quickly. As an example, during our N-Body benchmarking process, we had encountered a segmentation fault error when the number of actors used by the program became large[21]. We highlighted this on Pony's official github account and found that other users shared the same issue. The Pony development team acknowledged and fixed the bug within a few weeks. While we spent more time than expected picking up the language, we also benefited from having the opportunity to be involved in the language's development by highlighting bugs and getting direct help and support from the development team.

The performance of the language was consistently between that of Scala-Akka and Go, but much closer to Go. The only exception to this observation was when we run the N-Body simulation for large numbers of bodies, whereupon Pony performs better than Go. This may imply that Pony has a better control mechanism for large numbers of concurrent units. These results suggest that Pony may be a better choice for concurrent programming than Go when the benefits of its safety guarantees outweigh the small performance penalty and higher learning curve.

7.3.3 Go

Go is a powerful language that strikes a good balance between performance and ease of use. Goroutines and channels offer an intuitive way to use powerful concurrency features which are very easy for developers to utilise. While not designed around the actor model of concurrency, the ability for channels to pass arbitrary data types, and goroutines to receive and pattern-match the data passed through them made it easy, even as a beginner, to emulate an actor model style program. However, the syntax of such an implementation was less elegant than in languages specifically designed for it, so while this approach is possible, and demonstrates the flexibility of the language, it is probably not an ideal way to approach the development of a concurrent program in Go.

In all our benchmarks, Go demonstrates not only the best performance of the three languages tested, but also extremely consistent results. This suggests that it may be a good fit for real-time applications which must consistently meet performance deadlines. Memory usage was also consistently the lowest of the three languages, which combined with the performance characteristics mean that Go would be the best fit of these three languages for concurrent programs in resource-constrained systems.

8 Conclusions and Extensions

The broad focus of this project made it was very challenging — any one of the benchmark development, front-end tool development, or results analysis tasks could have been an entire project in and of themselves. Despite these challenges, we have managed to meet the goals of the project, successfully creating an extensible benchmark suite which has enabled us to make performance comparisons between the three languages tested.

The clear extension of this project is thus to add to the benchmarking suite. There are many other concurrent languages which could be compared, and implementing more benchmarks would allow different performance aspects to be tested. Our design of the front-end tool and thorough documentation should make this task as easy as possible, hopefully encouraging other developers and researchers to expand upon the work we have done here.

References

[1] Jeff Magee and Jeff Kramer. *State models and java programs*. wiley, 1999.

[2] Shams Imam and Vivek Sarkar. Savina-an actor benchmark suite. In *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE*, 2014.

[3] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[4] Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

[5] Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[6] Scala programming language. http://www.scala-lang.org/Scala. Last accessed: 17 May 2017.

[7] Akka toolkit. http://akka.io. Last accessed: 17 May 2017.

[8] Pony programming language. https://www.ponylang.org. Last accessed: 17 May 2017.

[9] Go programming language. https://golang.org. Last accessed: 17 May 2017.

[10] Encore programming language. https://www.gitbook.com/book/stw/the-encore-programming-language/details. Last accessed: 17 May 2017.

[11] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. 1993.

[12] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.

[13] Microsoft. Uml sequence diagrams. https://msdn.microsoft.com/en-us/library/dd409377.aspx. Last accessed: 17 May 2017.

[14] Edsger W Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer, 1968.

[15] Christopher Hartman and Bedrich Benes. Autonomous boids. *Computer Animation and Virtual Worlds*, 17(3-4):199–206, 2006.

[16] Scalatest funsuite. http://www.scalatest.org. Last accessed: 17 May 2017.

[17] Ponytest. https://github.com/ponylang/pony-tutorial/blob/master/testing/ponytest.md. Last accessed: 17 May 2017.

[18] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[19] Michael E Thomadakis. The architecture of the nehalem processor and nehalem-ep smp platforms. *Resource*, 3:2, 2011.

[20] John Zelle. graphics.py. mcsp.wartburg.edu/zelle/python/graphics.py. Last accessed: 17 May 2017.

[21] Bug report: Nondeterministic failure when high number of threads are used. https://github.com/ponylang/ponyc/issues/1781. Last accessed: 17 May 2017.

A Appendix

Code repository: `https://gitlab.doc.ic.ac.uk/g1653011/actor_benchmarking`

A.1 Pre-requisites to run benchmarks using the front-end tool

For Linux:

- Python 3.5+
  - matplotlib
  - numpy
  - tkinter
- Java Runtime Environment 8 (JRE)
- SBT (sbt) (has to be in PATH environment variable)
  - Scala Compiles using SBT build tool, with the SBT native packager plugin
- Pony Compiler (ponyc) (has to be in PATH environment variable)
- Go Compiler (go) (has to be in PATH environment variable)

For Mac OS X:

- All to the pre-requisites for Linux (above)
- GNU time

A.2 Front-End Manual

```
### Description

run_tool script is a helper tool used to compile and run multiple benchmarks
for multiple programming languages and generate corresponding plots and CSV
data for various performance metrics.

### Synopsis

main.py bench_dir [-h] [-v | -q] [-c | -r | -p] [-w WORKDIR] [-t BENCH [BENCH ...]]
        [-o] [-e REPETITION_COUNT] [-g] [-f PLOT_FORMAT]



### Functionality

Steps that run_tool can perform:
1. iterate the directory tree passed as a positional argument assuming a pre-set
structure (see Benchmark directory structure for details) and
automatically create a list of available languages and benchmarks discovered
in the directory tree for use in following step

2. compile discovered benchmarks provided that compilation scripts are available
for languages discovered (for compilation scripts requirements, please see
Compilation Scripts section)

3. execute the compiled benchmarks through Unix time utility to generate the
performance data (for benchmark configuration files, please see Benchmark
Configuration section)

4. process the data and output plots as well as CSV files
```

```
The tool has been designed such that steps 1 and 2, step 3 and step 4 can be
run separately. This provides flexibility in how the whole benchmark suite is
run. The reason for steps 1 and 2 to always be executed together is that step
1 is relatively short compared to step 2 and these two actions are closely
coupled.

For clarity, the following names are going to be used:
- Steps 1 and 2 - Compilation Phase (CP)
- Step 2 - Execution Phase (EP)
- Step 3 - Processing Phase (PP)

Data for CP and EP is saved as text files to allow EP and PP respectively to
be run separately. Use-cases for this functionality are described in Usage
section.

### Benchmark directory structure

The assumed directory structure is such that first-level subdirectories at the
tree provided are available languages, and all second-level subdirectories
(every directory inside the language directories) are benchmarks. Visually,
this can be represented as:

benchmark_dir
|-- lang1
|   |-- benchmark1
|   |-- benchmark2
|   .
|   .
|   .
|   |-- benchmarkn-1
|   `-- benchmarkn
|
|-- lang2
|   |-- benchmark1
|   |-- benchmark2
|   .
|   .
|   .
|   |-- benchmarkn-1
|   `-- benchmarkn
.
.
.
|
`-- langn
    |-- benchmark1
    |-- benchmark2
    .
    .
    .
    |-- benchmarkn-1
    `-- benchmarkn

### ignore.txt and rename.txt

Crawler also takes into account information provided in files named as either
ignore.txt or rename.txt.

ignore.txt can be located at either the top-level benchmark_dir as indicated
in the section above, or in any of the language directories. The purpose of
this file is to force Crawler to ignore directories specified in the
ignore.txt - this is useful when only a certain subset of languages is to be
used or if benchmark tree contains folder that should not be interpreted as
languages or benchmarks.
```

```
rename.txt can only be located in benchmark directories. It is used to force a
name interpretation that does not necessarily correspond to the benchmark
directory name.  As an example, this is used to rename Scala-Akka benchmarks
to lowercase letters since the convention of Scala is to use CamelCase for
project names whereas both Go and Pony uses all-lower project names.

### Usage

Simplest possible call:

    main.py BENCHMARK_DIR
    main.py /directory/to/benchmarks
    This will crawl through BENCHMARK_DIR, discover all the languages and
    benchmarks and will compile them, then run them using default options (see
    Options section) and will process the data and output the resulting plots.

Re-run pre-compiled benchmarks

    main.py BENCHMARK_DIR -r
    main.py /directory/to/benchmarks -r
    Uses intermediate data from previous invocation (either full run as
    specified above or run with -c flag set) to re-execute the benchmarks.

    This could be useful if a subset of benchmarks should be executed as well
    as modifying the benchmark configuration files (see !!! Benchmark config
    files !!!  section) before re-executing the benchmarks.

Only process the results

    main.py BENCHMARK_DIR -p
    main.py /directory/to/benchmarks -p
    Uses intermediate data from previous invocation (either full run or run
    with -r flag set) to process the data again.

    This is useful if plots of different dependent variables is required (in
    conjunction with -f option)

### Options
-h Print brief help message and exit.

-v, -q Adjust verbosity level of the output produced by the tool. These flags
       are mutually exclusive (-v stands for [v]erbose and -q stands for
       [q]uiet). Verbose option prints all the possible messages, including
       debug information. Quiet option only prints warnings and errors. If
       none of the verbosity options are provided, default verbosity level is
       used which is a middle ground between quiet and verbose

-c, -r, -p Select which phase of the tool to execute. These options are
           mutually exclusive (-c stands for compilation phase, -r stands for
           execution phase and -p stands for processing phase). -r should only
           be selected after either a CP or the full run has been executed
           before and -p should only be selected after either a EP or the full
           run has been executed before. If none of these options is selected,
           all of the phases are executed in turn by default.

-w WORKDIR Specify a working directory for all the output files generated by
           the script. This directory will contain intermediate files for the
           phases, output plots and CSVs as well as benchmark binaries. It is
           worth mentioning, however, that the tool does not enforce the
           storage of benchmark binaries in the WORKDIR - this depends on the
           implementation of the compilation script. If the option is not
           specified in command line, WORKDIR defaults to work in the
           directory where the script has been run from.

-t BENCH [BENCH ...] Specify which benchmarks to compile and/or run. This
                     option is ignored if the tool is run with -p. The list of
```

```
                    benchmarks that results after this option is applied is
                    effectively an intersection of benchmarks available
                    (discovered or compiled) and benchmarks provided with
                    this option. This implies that if EP is being executed,
                    no more benchmarks will be compiled even if the
                    benchmarks specified with the -t option are valid. By
                    default, all available benchmarks are used.

-o Overwrite the working directory if it already exists. The default behaviour
   if this flag is unset is to move the existing working directory to
   <directory_name>_<timestamp>_bak.

-e REPETITION_COUNT Specify the number of times each test point is repeated.
   This is useful to get more reliable data by averaging the times at each
   point and statistical error data can also be extracted from CSVs. Default
   value if this option is not specified in command line is 2.

-g Show the plots using matplotlib's tk interface. Requires X-server running
   on the machine where the script is being executed. Default is to output the
   resulting plots as image files.

-f PLOT_FORMAT Specify which dependent variables will be plotted in the
               output. PLOT_FORMAT should be a single string without
               whitespace and should be a subset of following characters:
               "eUSPMwc"
               For meaning of the characters, please refer to time(1) man
               pages - meaning of each character is euquivalent to time(1)
               format string with prepended '%' character. Default value is
               "eM". If -g is specified, this option is ignored and
               PLOT_FORMAT is assumed to be "e".

### Compilation Scripts

Compilation scripts are written in python and are located in the sub-directory
of run_tool script called lang_cfg.

The script has to define LangUnit class as well as LANG_NAME variable at the
module scope, e.g. (minimal language script that does nothing):
    LANG_NAME = "test_lang"

    class LangUnit:
        pass

The LANG_NAME variable has to correspond to the language name found in the
benchmark directory structure (see Benchmark Directory Structure).

For the language script to perform any useful work, constructor that takes in
a working directory as well as compile() method that takes name of the
benchmark (1st argument - a string) and path of the benchmark (2nd argument -
a string) needs to be defined.

compile() method is expected to perform housekeeping tasks and sanity checks,
such as making sure that working directory exists and the tool has permission
to write to it. In addition to that, compile method also saves both stderr and
stdout of the compilers for potential review if compilation does not work as
expected (i.e. the executable was not generated).

The return value of compile() method is either None if there were issues
compiling the benchmark or string containing the command that would need to be
issued in the shell to run the executable.

### Benchmark Configuration

Each benchmark in the benchmark tree has to contain a corresponding
configuration file. The file follows standard INI file structure as follows:
```

```
    [Case Name]
    arguments = variable, const_arg1, const_arg2
    argument_names = Iterations, Other Parameter, And Another One

    start = 1
    stop = 10
    step = 2

In the snippet above, Case Name can be any name as long as it provides enough
information for the user of what that particular test case is used for.

The arguments option allows to select which single argument will be varying
throughout the particular test case - or everything could be set as constants,
meaning that only a single datapoint will be generated.

It is worth mentioning that script makes an assumption about arguments to the
benchmarks - the first argument is assumed to be a boolean and it indicates
whether anything should be printed by the benchmark or not. I\O is usually
extremelly slow and since the tool was not designed to benchmark I\O, it
disables the printing in the benchmarks (implicitly passing 0 as the first
argument to the benchmark).

The range definition describes how the variable argument is going to be
changed throughout the test case and arguments follow the definition of Python
range() class.

Alternatively, variable values can be specified as a comma-separated list:
    steps = 1, 2, 3

The range or discrete step definitions of variable should be specified in
mutual exclusion.
```

A.3 Demonstration of Actor Termination

A.3.1 Scala-Akka - Termination

```
case object Start
class TestActor()
object PingPongTest extends App{
    //create ActorSystem and TestActor
    val system = ActorSystem("MainSystem")
    val testactor = system.actorOf(Props(new TestActor()), name = "testactor")

    //message testactor to start
    testactor ! Start

    class TestActor() extends Actor {
        def receive = {
            case Start(thesender: ActorRef) =>
                //receive message, do nothing
                //explicitly calling ActorSystem to terminate
                context.system.terminate()
      }
    }
}
```

In Scala-Akka, actors and the actor system have to be explicitly terminated, or the application will run indefinitely. Within the receive method of an actor, an actor can

- Terminate itself using `context.stop(self)`
- Terminate other actors in the system using `context.stop(actorRef)` or `actorRef ! PoisonPill`
- Terminate its entire actor system (all actors) using `context.system.terminate()`

Alternatively, within the actor system instance, termination can be called using `system.terminate()`.

A.3.2 Pony - Termination

```
actor Main
  new create(env: Env) =>
    env.out.print("Test actor is created")
    let another_actor = TestActor(env)
    test_actor.last_behaviour(env)  // calling TestActor behaviour

actor TestActor

  new create(env: Env) =>
    env.out.print("TestActor is created")

  be last_behaviour(env: Env) =>
    env.out.print("Program will terminate here")
```

In Pony, the program terminates when the last actor finishes its execution without calling any other behaviours, as shown in the example above. Therefore, there is no need to explicitly terminate actors.

A.3.3 Go - Termination

```
func main() {
    //create synchronisation channel
    done := make(chan uint64)

    //start goroutine
    go testroutine(done)

    //block until goroutine messages through channel
    <-done
    //execution terminates here
}

func testroutine(done chan uint64){
    //perform operations
    //signal completion to main by writing arbitrary value to channel
    done <- 0
}
```

In Go, execution of the entire program terminates with `main`. This can be delayed by performing a blocking read on a channel in `main`, and writing to the channel when all goroutines have finished execution, as shown above. Alternately, synchronisation primitives such as `WaitGroup`, found in the 'sync' package, can be used to similar effect.

A.4 Test Coverage Data

Name	Stmts	Miss	Branch	BrPart	Cover
compilation_executor.py	24	0	12	0	100%
crawler.py	33	0	14	1	98%
data_interpreter.py	138	19	70	2	85%
main.py	130	12	28	4	90%
plotter.py	117	104	58	1	8%
test_executor.py	104	2	54	2	97%
TOTAL	546	137	236	10	73%

Figure 19: Front-End test coverage.

Language	Class	Method	Statement
Pony	N/A	N/A	N/A
Go	N/A	N/A	77%
Scala-Akka	81%	84%	88%

Table 8: Code coverage statistics for the N-Body simulation (metrics not provided by some test suites).

```
./nbody.go:52:          Set                     100.0%
./nbody.go:61:          K                       100.0%
./nbody.go:73:          Distance                100.0%
./nbody.go:79:          Fgravity                100.0%
./nbody.go:85:          Vgravity                100.0%
./nbody.go:90:          Force                   100.0%
./nbody.go:108:         TForce                  100.0%
./nbody.go:121:         Potential               100.0%
./nbody.go:132:         TV                      100.0%
./nbody.go:146:         Energy                  100.0%
./nbody.go:160:         euler_step              100.0%
./nbody.go:170:         run_system_euler        100.0%
./nbody.go:228:         AdamsBashforth4_step    100.0%
./nbody.go:257:         calculate_initial_4     100.0%
./nbody.go:267:         run_system_AdamsBash4   75.0%
./nbody.go:335:         InitialConditions       100.0%
./nbody.go:349:         OpenFile                71.4%
./nbody.go:365:         main                    0.0%
./vector.go:14:         Magnitude               100.0%
./vector.go:22:         Sum                     100.0%
./vector.go:36:         Sum4                    100.0%
./vector.go:50:         Dif                     100.0%
./vector.go:64:         Dot                     100.0%
./vector.go:80:         Cros                    100.0%
./vector.go:94:         Mltp                    100.0%
./vector.go:108:        Addc                    100.0%
total:                  (statements)            77.4%
```

Figure 20: N-Body simulation test coverage in Go.

[all classes] [com.nbody]

Coverage Summary for Package: com.nbody

Package	Class, %	Method, %	Line, %
com.nbody	80.8% (21/ 26)	83.6% (117/ 140)	88% (382/ 434)

Class ▲	Class, %	Method, %	Line, %
Calculate	100% (2/ 2)	100% (35/ 35)	100% (152/ 152)
Main	76.2% (16/ 21)	75% (69/ 92)	79.2% (198/ 250)
VectorMethods	100% (3/ 3)	100% (13/ 13)	100% (32/ 32)

generated on 2017-03-08 20:55

Figure 21: N-Body simulation test coverage