# A Software Package for Simulating and Visualising Load Balancing Algorithms

Student Name: Daniel Stone
Supervisor Name: Dr Tom Friedetzky
Submitted as part of the degree of BSc Computer Science to the
Board of Examiners in the Department of Computer Sciences, Durham University

**Abstract**—A simple abstraction of load balancing systems is the balls into bins probability model, which can be modified to represent more load balancing scenarios. Performance analysis of algorithms used to allocate balls into bins is often complex and usually not intuitive. This project aims to create a software package to aid in collecting and analysing experimental data for various balls into bins algorithms. The package provides an easy-to-use API that can run simulations faster than equivalent plain Python code whilst reducing the implementation complexity required to run and parallelise experiments. Simulation results contain several metrics that are calculated after the experiments are run. These can be visualised in the provided Jupyter Notebook to compare and evaluate different algorithms. The software replicated results seen in literature with minimal effort to deliver an experimentation focused survey of commonly cited balls into bins algorithms.

**Index Terms**—Algorithm/protocol design and analysis, Distributed Systems, Distributed/Internet based software engineering tools and techniques, Load balancing and task assignment

## 1 Introduction

Load balancing is the process of allocating tasks to resources. The concept is heavily used in distributed systems where there are fewer resources than there are pending tasks. For instance, large web applications can use load balancing to accommodate more requests per second than an individual server could handle in this time, allowing a larger number of requests to be processed. If required, load balancers can be adjusted to distribute requests across various servers, enabling web applications to scale as they grow or support fluctuations in usage. Another valuable characteristic of load balancing is its ability to maintain availability by providing failover when servers become unavailable. If faults are detected in a server, the failing requests can be distributed across the remaining operational servers [1]. In practice, most load balancing systems are transparent to the user and operate on many web requests per second as soon as they are received.

The decision of where to send a web request is determined by the algorithm used, which influences the computational load that each server receives. Different algorithms can be evaluated with multiple objectives, including response time, execution time and maximum load (often referred to in the literature as makespan).

Research on load balancing's use within cloud computing is extensive. In this context, load balancing algorithms can be categorised as either static or dynamic [2]. Dynamic load balancing algorithms can consider the current state of the distributed system; however, these algorithms are often tricky to implement due to the communication required to track server load. Static algorithms cannot use this information and instead rely on random decisions or decisions based on the incoming data, such as the IP Address of the web request. These definitions are one example of the various meanings for the terms static and dynamic in load balancing. In practice, the large cloud providers provide only a small subset of algorithms. For example, Amazon Web Services 'Classic Load Balancer' only provides two algorithms - round robin and least outstanding requests [3].

### 1.1 Balls into Bins

A simple load balancing algorithm is random selection. This is when a server is chosen at random for each request. The algorithm and its effects can be abstracted as the basic 'Balls into Bins' probability problem. The problem is as follows: Throw $m$ balls into $n$ bins, with each bin chosen independently and uniformly at random. A web application that handles $m$ requests per minute and has $n$ available servers can be modelled with $m$ balls and $n$ bins, with the number of balls in each bin after all balls are thrown representing the requests sent to each server in one minute. The theoretical model has further applications in Computer Science, such as its use in the analysis of hashing algorithms [4]. However, this paper will focus on its application to load balancing.

The objective of a well-performing load balancing algorithm is to ensure that no particular bin becomes overloaded. After all the balls are thrown, the load of a bin is equal to the number of balls it contains. It follows that the maximum load is the highest load of any bin. For the random selection algorithm, it could be said that in the improbable event every ball is allocated to the same bin, the maximum load would be equal to the number of balls. Likewise, the minimum load could be zero. For this reason, most of the analysis of these load balancing algorithms is regarding the *expected* maximum load.

Determining an upper bound for the expected maximum load across every bin can help determine the capacity re-

quired by the servers, and a good algorithm should minimise this number. There is, however, a trade-off between the maximum load and the time taken to allocate the balls, an area that will be discussed later by analysing the communication complexity.

An optimal load balancer may keep track of every bin's load to choose the least loaded bin for every ball - a simple greedy algorithm. However, concurrent load balancers may be required in practical cloud deployments to reduce the impact of a failing load balancer. This poses further questions for the analysis of balls into bins models since balls are not necessarily thrown in sequential order. The greedy and round robin algorithms break down in these distributed scenarios as the multiple load balancers may receive stale data. For example, an algorithm running in parallel that looks for the lowest loaded bin may direct all the balls at a single bin at once, causing that bin to become overloaded. This process will continue for other bins causing oscillations to occur in the bin's load as they become overloaded and then underloaded repeatedly [5].

### 1.2 Project Aim

The project aims to create an easy-to-use API that can run simulations of variations of balls into bins model. The project will not focus on mathematical proofs for the worst-case performance of each algorithm since minor changes to an algorithm typically require complex mathematical analysis. The API will allow users to design and implement load balancing strategies with a simplified Python syntax that contains methods to perform common load balancing operations. It will provide a simulation engine that can run the algorithm on specified inputs and support several modifications to the balls into bins model. These modifications include the number of balls, bins, and the balls' properties. If required, the API will provide a method to gather more data about the current state of the simulation, such as the current load of a particular bin or the number of balls thrown in earlier iterations. A framework for running increasingly large inputs should be provided to produce data for plotting charts. The primary evaluation metric will be the maximum load across all bins after every ball has been thrown, although later deliverables may include additional metrics. The simulation must run at a speed comparable to C or C++ since these languages offer maximum performance for running comparisons at scale. Different methods for running the algorithms will be evaluated to balance implementation complexity with runtime. It is hoped that the final software package will be beneficial for future research in this area by reducing the barrier to entry for comparing load balancing strategies.

The intermediate deliverables for the system include the ability to sample the weights for the balls from a given distribution. Weights can be passed as a fixed parameter to the simulation so that the same distribution can be tested on multiple algorithms. Weighted balls can represent the amount of space a task takes on a storage medium or the amount of time a task may take to run relative to others. The weight distribution and order are likely to change the expected maximum load, especially if the weights are not evenly distributed. A further evaluation metric, communication complexity, will be used to record the number of times a bin was queried for its current load and can provide an indication of the algorithm's execution time. A simple way to simulate distributed load balancing systems will also be supported by including a parameter representing the number of concurrent load balancers.

The advanced deliverables for the system include parallelisation of the simulation so that larger simulations can take advantage of multicore systems for greater speedup. This feature will be opt-in to allow users to implement their own multiprocess solution. Additionally, a Jupyter Notebook [6] that contains example usage of the library will be provided to act as a tutorial for the software package. The results discovered using the software package will also be included in this notebook so that experiments can be modified for different algorithms. Another goal is integration with the `ipywidgets` library for interactive charts. These charts will show how the load distribution changes over time and can be controlled by an interactive slider. In summary, the notebook will be self-documenting and should contain the steps required to analyse load balancing algorithms.

By using the software package, we will compare and evaluate multiple load balancing algorithms by verifying the results seen in the literature.

## 2 Related Work

The 'Balls into Bins' problem is well known in probability theory and is formally described as the maximum load across $n$ bins after $m$ balls have been thrown into a bin. Traditionally, the balls are thrown into a bin selected independently and uniformly at random. In this paper, we will call this the UNIFORM algorithm. A known result for the expected maximum load when $m = n$, with high probability is $\frac{\ln n}{\ln \ln n} \cdot (1 + o(1))$. In this context, high probability means with probability $1 - \mathcal{O}(1/n)$. This also indicates that as the number of balls increases, the probability that the expected maximum load is bounded by the result becomes closer to one. In this section, it should be assumed that all the referenced results are with high probability unless otherwise stated.

An early appearance of the problem within Computer Science literature was in 1981. Gonnet [7] applied the model to uniform probing hashing with direct chaining and proved an accurate result for the expected $llps$ (length of the longest probe sequence). In the context of hash tables, uniform probing hashing is where keys hash to uniformly random locations. The mathematical result applies when the hash table is of size $n$ and is full ($m = n$). $\Gamma$ is the gamma function.

$$\mathbb{E}\left[llps\right] = \Gamma^{-1}(n) - \frac{3}{2} + o(1)$$

This result is easily applied to load balancing by considering a hash probe equivalent to a request on a server. The hashed value for a key is independent of the table's status and is uniformly random, as are the decisions made in the UNIFORM balls into bins algorithm. Since the algorithm is completely random, it can be used in distributed load balancing systems with no changes to the expected maximum load; Therefore, the load balancing algorithm can be

sequential or parallel (performed once for each ball thrown in sequential order or run on all the balls at once as a parallel algorithm). Raab and Steger [4] later derived more general upper and lower bounds for the algorithm for multiple cases where $m \neq n$ in their paper titled "Balls into Bins - A Simple and Tight Analysis". When $M$ is the random variable that counts the maximum number of balls in any bin, $M > k_\alpha$ with low probability when $\alpha > 1$ and high probability when $0 < \alpha < 1$.

$$k_\alpha = \begin{cases} \frac{\log n}{\log \frac{n \log n}{m}} \left(1 + \alpha \frac{\log \log \frac{n \log n}{m}}{\log \frac{n \log n}{m}}\right) & \text{when } \frac{n}{\text{polylog}(n)} \leq m \ll n \log n \\ (d_c - 1 + \alpha) \log n & \text{when } m = c \cdot n \log n \text{for some constant c} \\ \frac{m}{n} + \alpha \sqrt{2 \frac{m}{n} \log n} & \text{when } n \log n \ll m \leq n \cdot \text{polylog}(n) \\ \frac{m}{n} + \sqrt{\frac{2m \log n}{n} \left(1 - \frac{\log \log n}{2\alpha \log n}\right)} & \text{when } m \gg n \cdot (\log n)^3 \end{cases}$$

Simply by the number of mathematical cases required to describe this problem, it is clear that a lot of work is necessary to determine these bounds. Furthermore, these results rely on a value for $\alpha$, which may be found easiest through experimentation.

### 2.1 GREEDY[$d$]

The GREEDY[$d$] algorithm is a sequential algorithm that is partially random. For each ball, $d$ many bins are chosen independently and uniformly at random. The current load of these bins is queried, and the bin with the lowest load is determined. This is the bin that the ball is thrown into, with ties broken arbitrarily. The value of $d$ can be in the range $1 \leq d \leq n$, such that the GREEDY[$n$] algorithm is in fact the standard GREEDY algorithm. In the mid 1990s, Azar et al. [8] showed that with high probability the expected maximum load is less than $\frac{\ln \ln n}{\ln d} + \mathcal{O}(1)$. For $d = 2$, this provides an exponential decrease compared to the uniform sequential algorithm, which is equivalent to when $d = 1$. This improvement in maximum load is significant due to the relatively small amount of coordination required to implement this algorithm when $d$ is small.

The GREEDY[$d$] algorithm was revisited by Mitzenmacher [9], who applied the strategy to a revised balls into bins problem called the 'Supermarket model'. The term supermarket comes from modelling customers and cashiers as balls and bins, respectively. Customers arrive following a Poisson process and leave the system after an exponential amount of time. In this dynamic scenario, the GREEDY[$d$] algorithm was shown to provide similar improvements in the probable maximum load.

Contrary to the load balancing terms used in cloud computing, algorithms such as UNIFORM and GREEDY[$d$] are generally called static algorithms since balls remain in bins after being thrown. In contrast, dynamic algorithms as those that allow two operations on bins, insertions and deletions [10]. These definitions will be used in the remainder of the paper.

For simplicity, we will only consider static algorithms. In particular, the UNIFORM, GREEDY[$d$], ALWAYS-GO-LEFT, $(1 + \beta)$-CHOICE, THRESHOLD, and ADAPTIVE algorithms.

### 2.2 ALWAYS-GO-LEFT

The ALWAYS-GO-LEFT algorithm is characterised by the algorithm's asymmetric tie-breaking strategy and was found to perform better than GREEDY[$d$] [11]. For each ball, the algorithm partitions the bins $1...n$ into $d$ groups $i \leftarrow 1..d$ and independently and uniformly at random chooses a bin from each group $i$. The load of each selected bin is compared, and the bin with the lowest load is chosen. The tie-breaking strategy is defined in the algorithm and states that the ball from the group with the smallest $i$ should be selected, i.e. the leftmost group. The expected maximum load is defined with a generalisation $\theta_d$ of the golden ratio ($\theta_2$) and is

$$\frac{\ln \ln n}{d \ln \theta_d} + \mathcal{O}$$

The paper concludes that the asymmetry provided by the tie-breaking strategy is required for this improvement in maximum load and that a similar partitioning algorithm with a fair tie-breaking strategy would perform the same as other partially random algorithms. Similarly, the asymmetry was only proven to improve performance for partitioned bins. Without the partitioning of bins, the tie-breaking strategy did not change the maximum load of the algorithm.

Experimental results are also presented in the paper, comparing ALWAYS-GO-LEFT with GREEDY[$d$]. This data will be omitted from this section as it will be shown in section 4 to compare and evaluate the correctness of the proposed solution. The procedures introduced by the paper may be helpful to guide our tests by providing suitable values for the number of repeats (100) and varying sizes of $n$ (up to $m = n = 2^{24}$). The software package should be able to simulate these scenarios in a reasonable amount of time.

### 2.3 1 + BETA

In 2010, Peres et al. presented the $(1 + \beta)$-CHOICE process [12]. This algorithm is parameterised by $\beta \in (0, 1)$ and is a combination of the UNIFORM and GREEDY[2] algorithms. With probability $\beta$, the load balancer queries two bins and chooses the lowest loaded (essentially GREEDY[2]). With probability $1 - \beta$, the ball is thrown into a randomly chosen bin (UNIFORM). Therefore, the algorithm is said to have a communication complexity of $1 + \beta$ for each ball thrown. The paper focuses on the expected gap between the average load ($\frac{m}{n}$) and the maximum load.

The earlier result for the UNIFORM algorithm's expected maximum load where $m \gg n \log n$ [4] was referenced and used for comparison. The average load, $\frac{m}{n}$, was subtracted to find the expected gap between the maximum load and the expected load for the UNIFORM algorithm: $\Theta(\sqrt{\frac{m \log n}{n}})$. The paper found that the $(1 + \beta)$-CHOICE algorithm had an expected gap of $\Theta(\frac{\log n}{\beta})$, and this was found to remain when a number of different weighted distributions were used.

### 2.4 THRESHOLD and ADAPTIVE

In 2001, Czumaj and Stemann introduced the M-THRESHOLD algorithm, which we will refer to here as THRESHOLD [13]. This algorithm was shown to achieve a reduced maximum load as well as a reduced *allocation time*. The allocation time is equivalent to the communication

complexity of the algorithm and is the number of bins that have their load queried. The GREEDY[$d$] algorithm, for example, has a communication complexity for each ball of $d$ since $d$ bins are queried to determine the minimum loaded.

The THRESHOLD algorithm is sequential and is performed in rounds. Multiple rounds may occur before a bin is selected, so the number of queries is not fixed for each ball. In each round, a bin is chosen independently and uniformly at random. If the load of the selected bin does not exceed the threshold $M$, it is chosen; otherwise, the algorithm repeats. With $n = m$ and $M = 2$, the maximum load is clearly 2. The communication complexity for this algorithm was shown to be at most $1.146194 + o(1)$, an improvement over GREEDY[2].

Berenbrink et al. [14] proposed an enhancement to the THRESHOLD algorithm, named ADAPTIVE. This aimed to maintain the same expected maximum load and communication complexity whilst removing the requirement to know the total number of balls, $m$, upfront. The changes begin by modifying the THRESHOLD algorithm to calculate the parameterised threshold instead, such that $M = \frac{m}{n} + 1$. When $m = n$, $M = 2$. The modification for ADAPTIVE is minor - change the $m$ to the position of the ball $i$ such that $M = \frac{i}{n} + 1$. The maximum load of this algorithm was shown to be $\lceil \frac{m}{n} \rceil + 1$, and does not require any conditions for $m$ and $n$.

### 2.5 Distributed Load Balancing

Although it is known that multiple-choice load balancing algorithms such as GREEDY[2] have a lower expected maximum load, the previous discussions have been strictly in the sequential case where each ball gets a different view of the current system load. The model can be modified such that balls arrive in *batches*, akin to parallel load balancing with as many balancers as there are balls in a batch. The load values available to the algorithm are updated after each batch, emulating the behaviour if all the balls were allocated simultaneously.

By fixing the number of bins to the size of each batch, $n = b$, and with $m \geq n$, the maximum load of all bins for GREEDY[2] is $\frac{m}{n} \pm \mathcal{O}(\log n)$, with high probability [15]. The gap between maximum and minimum load is, therefore, $\mathcal{O}(\log n)$. Very recently, Los and Sauerwald [16] presented a generalised analysis of these earlier results. The new findings can be applied to the $(1 + \beta)$-CHOICE algorithm and in scenarios where $b \geq n$.

### 2.6 Weighted Balls

Another area of interest is modifying the balls into bins problem so that the balls are of non-uniform weight. In these weighted scenarios, we may disregard the ALWAYS-GO-LEFT algorithm since tie-breaking is much less likely to occur [17]. Berenbrink et al. showed that given two equivalently sized sets of balls, the set with the weights that are more evenly distributed would have a lower expected maximum load when using a single choice algorithm such as UNIFORM [18]. For multiple-choice algorithms, including GREEDY[$d$], it was shown that if the number of balls is large, for some algorithms, the expected maximum load of equally-weighted balls was not always better than the allocation of weighted balls.

The paper also provided experimental results for weighted balls and the GREEDY[2] algorithm. Two scenarios were trialled for varying numbers of bins. The first indicated that the more evenly distributed the weights were, the lower the expected maximum load. The second experiment provided experimental evidence for lower expected maximum load when allocating balls in order of maximum to minimum weight, although, in the paper, it was not proven if this produced the minimum expected maximum load.

### 2.7 Simulation Software Packages

Although previous papers have presented experimental results, to the author's knowledge, there are no similar software packages for describing and simulating balls into bins load balancing scenarios. This makes the project unique. However, because of this, it is unproven whether a tool like this will be helpful for other researchers.

An inspiration for the simulation system was `ppsim` - 'A software package for efficiently simulating and visualizing population protocols' [19]. Population protocols are a model for interactions between identical finite state machines [20], and the `ppsim` simulation can be used to determine the state for which the finite state machines converge. The `ppsim` software package accepted a description of the system and provided methods to create plots of the resulting data. In the provided example Jupyter Notebook, an `ipywidgets` slider was used to show how the state of the simulation changed over its runtime. The paper implements an algorithm that is "quadratically faster than the naive algorithm". Unfortunately, it is reasonably assumed that no clever algorithm exists to speed up balls into bins simulations, so the focus of our solution was on code execution speed.

Software packages to run efficient simulations are common, with an example in a completely different domain being 'Pyroomacoustics: A Python Package for Audio Room Simulation and Array Processing Algorithms' [21]. This package similarly aims to reduce the time taken to implement simulations so that new algorithms can be developed and tested. This paper also addresses the popularity of scripting languages such as Python due to its improved code readability, although it solely relies on Python libraries to improve runtime.

## 3 METHODOLOGY

Many different implementation approaches were considered and evaluated by their complexity and runtime. All the approaches had to include an easy-to-use Python API for describing algorithms as simply as possible. Python was chosen due to its wide use in Academia and the project author's familiarity with the language. Its syntax is simple to understand at a glance, and there exist a large number of data science libraries that can be used for viewing and storing experimental data.

The available methods for running code as fast as possible were compared to identify the areas to focus on in the implementation. Then two approaches for describing and running the simulation were prototyped to determine if

they were flexible enough to support many different types of load balancers. After extensive experimentation, a new solution was proposed and developed, which provided the basic deliverables. Intermediate and advanced deliverables were built on top of this system to further aid comparisons with the literature.

### 3.1 Load Balancing API

The API for the solution should allow users to describe any static load balancing algorithm without concerning the underlying list operations required to run a simulation. Table 1 shows the features necessary to describe the majority of load balancing algorithms and represents an object-based interface for interacting with the simulation. Functions on Python objects are easier to understand than functions that take in parameters, and allow editor tooling to show suggestions for available operations. The implementation will try to follow this API as closely as possible; however, there should be a way of easily extending this functionality if desired.

TABLE 1
Load Balancing Algorithm API

| Function | Invocation |
|---|---|
| **`Ball`** | |
| Return the total number of balls | `m` |
| Return the current ball's order | `position` |
| Return the current ball's weight | `weight` |
| **`Bins`** | |
| Return the total number of bins | `n` |
| Return $d > 1$ bins uniformly and independently at random, with replacement | `choose(d)` |
| Return one **`Bin`** uniformly and independently at random | `choose_one()` |
| Split bins into $n$ equally sized subsets and return as list of **`Bins`** | `split(n)` |
| Return the first bin with the current minimum load | `min()` |
| **`Bin`** | |
| Return the current load of this bin (sum of contained balls' weight) | `load` |
| Return True with probabilty $p$ | `probability(p)` |

### 3.2 Runtime Comparison

To spot relationships between the inputs and outputs of a load balancing algorithm, it is often required to run large simulations. These allow, for instance, log relationships to be differentiated from linear relationships. For this reason, the simulation needs to run as fast as possible. It was expected that an algorithm's execution speed in Python would be much slower than one implemented in a lower-level language such as C or C++ since it is primarily an interpreted language. Therefore, three acceleration techniques were considered to mitigate the slowdowns - C++, Cython and Numba.

#### 3.2.1 CPython

The default and most commonly used Python implementation is CPython. It is written in C and inherently runs slower due to the extra steps taken to compile the Python code to bytecode before it is interpreted. However, it is likely to be installed on the user's computer due to its popularity, making it a good candidate. There are a large number of packages built for the CPython implementation, including some designed for fast array manipulation, such as Numpy [22]. The Numpy library often runs faster than Python by providing a wrapper around C implementations of efficient algorithms. However, if the code is bottlenecked by Python control flows, these may still dominate the runtime. In our analysis, CPython will be used with an installation of Numpy - otherwise, the code will run much too slow. The implementation used to run a GREEDY[$d$] simulation can be seen in Algorithm 1. Each ball selects a bin at random and compares the load to another $d-1$ bins. Out of these options, the bin with the lowest load at this point in time is selected, and the load is incremented by one. Finally, the maximum load is returned by searching for the largest value in the load array.

**Algorithm 1** GREEDY[$d$] simulation implemented with Python. The Numpy package has been imported as `np` and is used to speed up random operations.

```
def python_greedy_d(m: int, n: int, d: int):
  load = np.zeros(n, dtype="int64")
  for ball in range(m):
    lowest_bin = np.random.randint(0, n)
    for i in range(d - 1):
      choice = np.random.randint(0, n)
      if load[choice] < load[lowest_bin]:
        lowest_bin = choice
    load[lowest_bin] += 1
  return np.amax(load)
```

#### 3.2.2 Cython

Another available tool is Cython [23]. This is a superset of the Python programming language that can be used to convert Python code to C. It allows Python control flows such as if statements and loops to be directly compiled as C code, but extra overhead is incurred when interacting with the Python framework. It has built-in support for indexing Numpy arrays but requires programs to be modified with relevant data types and compiled before the program is executed. Programs that make regular calls between C or C++ code and Python benefit from native communication, and in most cases, Cython code runs as fast if not faster than Python.

#### 3.2.3 Numba

A different approach is taken by Numba, a Just-In-Time (JIT) compiler for Python and Numpy code [24]. The library compiles regular Python functions immediately before they are executed for the first time and achieves performance that rivals compiled languages such as C. It does this with very little input from the user by inferring the data types used within a function - the only change to Algorithm 1 was a `@njit` annotation above the function definition. For Numba to succeed in compilation, functions must use a subset of Python and Numpy features. The available Numpy functions are exhaustive and represent re-implementations

of Numpy code in a way that can be JIT-compiled. To run the code, the Python function's bytecode is first transformed into Numba's internal representation and converted to an LLVM supported intermediate representation (IR). LLVM is a set of modular compiler components that are used by many companies, including Apple [25]. It provides a very low-level language (the IR) that can be compiled for different hardware and benefits from being "easy to work with". The LLVM IR produced by Numba is further optimised by the compilation toolchain so that it can run even faster.

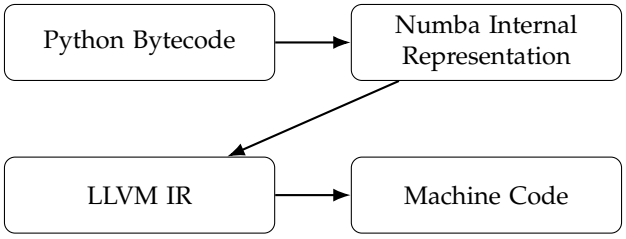


Fig. 1. Simplified Numba architecture.

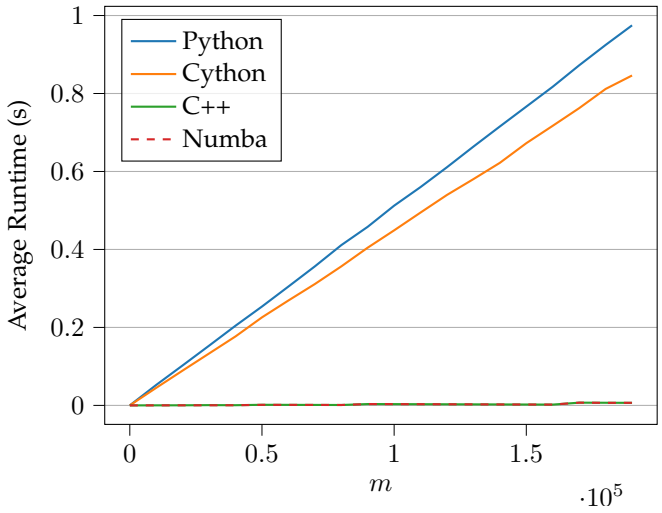


Fig. 2. Average runtime of 10 runs of GREEDY[2] implemented with each approach. The slowdown seen with Python and Cython is significant in comparison to C++.

#### 3.2.4 C++

Finally, maximum performance could be obtained by directly implementing the simulation in C or C++ since these are lower-level languages. In the GREEDY[$d$] implementation used for the following runtime tests, there are no advanced list operations. However, if there were, such as sorting, the user may have to implement efficient algorithms themselves or rely on third-party libraries.

The plot in Figure 2 shows the runtime in seconds of the GREEDY[2] algorithm when $m = n$. Runtime experiments were performed on a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor and 16 GB DDR4 RAM. Each data point was the result of averaging the time taken to run the algorithm ten times.

It is apparent that Python is much slower than C++ and is unsuitable for running large simulations. The speedup obtained from Cython is not as expected. This is likely because Cython cannot speed up the time it takes to call a Numpy function and has to interact with Python. This overhead is large since each iteration of the for loop makes a Numpy call. The Numba implementation is on par with C++ after JIT compilation and can simulate $m = n = 190010$ balls and bins in $0.0066$ seconds, around 150 times faster than Python. This is hugely impressive given the minimal changes required to the source code.

Therefore, plain Python and Cython solutions were disregarded for the project, and implementations that took advantage of the fast runtime of Numba or C++ were considered.

### 3.3 JSON Algorithm Description

Although C++ is seen to be very fast in Figure 2, this is for a direct implementation of the algorithm very similar to Algorithm 1. It should be expected that a general purpose simulation will need to perform more lines of code to produce the same output as general concepts such as groups of bins are introduced.

The first approach was to run an intermediate representation of the algorithm, similar to Numba. The intermediate representation will be generated by writing code in a simplified Python API. Once this is generated, it will be passed to a C++ simulation engine. The text based JSON format was chosen to simplify this process and aid with debugging. The format's portability between Python and C++'s many JSON libraries was another factor in this decision. In the future, the JSON intermediate representation could also be used with different backends written in other programming languages. For the C++ simulation engine, the `nlohmann/json` library was chosen as it provided a reasonable balance between speed and usability. The speed of the JSON parsing was not a priority for this implementation since it would only occur once at the start of each run. During the simulation, the code would work with C++ data structures which meant that the JSON parsing did not add any overhead to each ball throw.

Since the simple Python API consists of several defined functions, it was possible to create an algorithm description by tracking function calls. The available function calls described in Table 1 were stubs (functions with no implementation) that updated a global `Config` object every time they were called. The `Config` stored the number of balls and bins at the start of the simulation and was populated with a list of actions to perform. These describe which actions need to be run in turn to determine which bin should be chosen for each ball. To generate the `Config` object, the global variable was emptied, and the provided algorithm function was run.

The available bins were provided as a `Bins` object that pointed to a list of indices corresponding to the bins. The function stubs would add an `Action` object to the list, passing its associated inputs and outputs. For example, the `choose(d)` function would take the `Bins` as an input and return another `Bins` object that represented a subset of all the bins. These instances of `Bins` had corresponding id fields that were used during runtime to address the relevant indices in a C++ data structure. The contents of the `Config`

object can be seen in Figure 3.

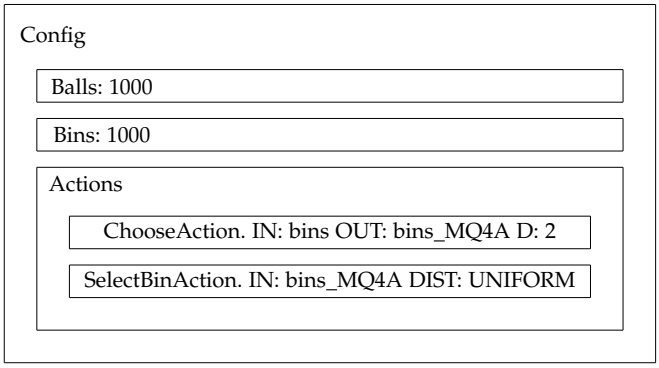


Fig. 3. Contents of the `Config` class for GREEDY[2]. The number of balls and bins of the simulation is stored in separate fields. The actions are stored as a list and each specifies its input, output and other parameters such as the distribution.

Next, the `Config` was converted into a JSON string. The `PyBind11` library was used to call and compile the C++ simulation engine code. When the simulation was run, the simulation engine received the JSON string to be parsed into C++ data structures. For each subset, lists of indices were initialised. The diagram in Figure 4 shows the contents of the C++ data structures after two bins have been chosen in the GREEDY[2] algorithm. The chosen bins indices are populated in `Bins` with id 1, and these are used later in the algorithm by the SelectBinAction to access the 'Bin Load' array and find the minimum loaded bin.

Once loaded, the simulation was run by looping and executing the list of actions for each ball. Unfortunately, as seen by the C++ simulation line in Figure 6, the amount of time taken to run this for loop and perform each action negatively impacted the runtime of the simulation. This could have been due to the overhead incurred by checking the type of each action and running the corresponding code, making it harder for the compiler to optimise. On large inputs, it was clear that the runtime of the simulation was closer to the direct Python implementation than the C++ equivalent. Therefore, it was determined that the simpler API was not worth the slight performance improvement and no further investigation into this technique was performed.

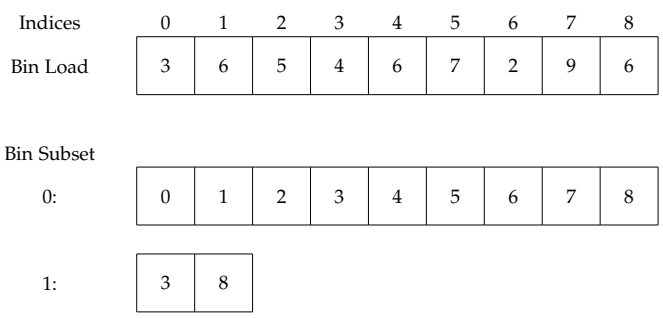


Fig. 4. Data structure used to represent bins in the C++ simulation. The bin subset lists allow subsets to be passed around in the description.

### 3.4 Numba Code Generation

An attempt at speeding up the runtime was by using the `Config` object to generate Python code that could be accelerated with Numba. The result was generated code that would run and return the results of a load balancing simulation with very little overhead.

An early revision of the API was used to describe the GREEDY[$d$] algorithm, Algorithm 2. A key difference in this early revision was the `lb.sequential()` context manager that specified that a single load balancer should be performing sequential load allocation. This functionality was later moved out of the load balancing algorithm description as a simulation parameter.

**Algorithm 2** GREEDY[2] described with an early revision of the simulation API.

```
def k_greedy(balls: lb.Balls, bins:
  ↪  lb.Bins):
  with lb.sequential():
    chosen_bins = bins.choose(2)
    balls.select_bin(chosen_bins,
     ↪  distribution=Distribution.MIN)
```

The implementation made extensive use of the built-in Python `AST` module. AST stands for abstract syntax tree, a representation of the Python syntax that can be used to parse and reason with source code. Reference implementation code was written for each `Action` in the `Config` and was parsed into abstract syntax trees. Once these were loaded, `NodeTransformers` were used to traverse the template code and insert copies of the syntax trees int to the basic simulation loop structure based on the order and number of actions. This allowed programmatic changing of variable names that correspond to the `Bins` ids, similar to those used by the C++ simulation (strings instead of integers). The result of this procedure was a new abstract syntax tree that represented the algorithm described in the `Config` object. Although this tree can be directly compiled and executed without first converting to Python code, the `unparse()` method was used to produce code that was used to verify that the system worked correctly. Figure 5 is a flow chart with the four main stages of the process.

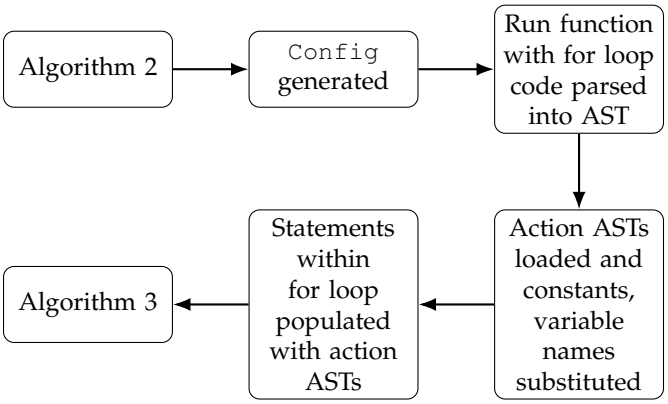


Fig. 5. Process to convert the load balancer description into Python and Numba code.

The code in Algorithm 3 is pure Python and Numpy code that is further optimised by Numba's JIT compilation before being executed. The highlighted lines were generated from the two actions specified by the GREEDY[2] algorithm, `choose` and `select_bin`. Line 7 shows a random choice of 2 bins from the original list of bin indices, with the result assigned to a variable corresponding to the random id specified in the config. Since this random id is known, it is substituted in the required locations in the next action (lines 8-13).

**Algorithm 3** Generated GREEDY[2] Python / Numpy implementation. A for loop is introduced, and action's code is expanded in place.

```
1  import numpy as np
2
3  def run(m, n):
4    """m balls into n bins"""
5    (load, bins) = (np.zeros(n,
     ↪  dtype=np.int64), np.arange(0, n,
     ↪  dtype=np.int64))
6    for ball in range(m):
7      bins_3XMY = np.random.choice(bins,
       ↪  size=2, replace=True)
8      min_index = bins_3XMY[0]
9      for i in range(1, bins_3XMY.size):
10       index = bins_3XMY[i]
11       if load[index] < load[min_index]:
12         min_index = i
13     load[min_index] = load[min_index] + 1
14   return np.amax(load)
```

In some ways, this approach was similar to how Numba processes code, with a few key differences. Both this approach and Numba convert source code into an intermediate representation before converting to a final representation to be run. The approach here differs by executing the source code to generate a description of the algorithm. In contrast, Numba determines the types and methods used in the function by analysing its bytecode. The internal representation for the proposed approach is much simpler and less powerful but has a similar purpose - it is easier to convert from code to this intermediate representation than it is to skip this step. Instead of outputting LLVM IR, this approach outputs more Python code. This is fairly unique as it converts simple Python code to more advanced Python code. However, the concept of converting code from one language to another is commonly referred to as transpiling code.

It is clear that the approach is not nearly as capable, complicated or impressive as Numba, but the parallels are interesting to observe.

Figure 6 shows that the Numba simulation had a runtime that was comparable to the direct C++ implementation. However, it was previously seen in Figure 2 that Numba should be on par with C++. Therefore, it was determined that the slowdown was likely due to the differences in the generated code. The C++ and earlier Numba implementation was written with the algorithm in mind and did not have to keep lists of bin indices, unlike the generated code in Algorithm 3.

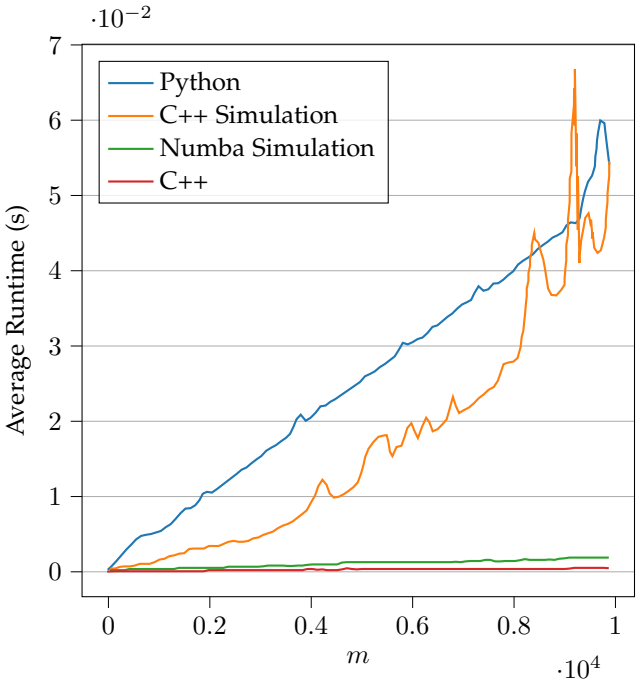


Fig. 6. Average runtime of 10 runs of GREEDY[2], implemented directly in Python and C++ compared with the two simulation approaches. The C++ simulation that parses and runs the given schema is almost as slow as Python. The Numba code generation is close in runtime to the direct C++ implementation.

### 3.5 Solution

Ultimately, the `Config` generation was not flexible enough to support most load balancing algorithms. In particular, algorithms that required branch operations proved challenging to capture in Python. The approach proved effective for simple algorithms. However, the $(1+\beta)$-CHOICE algorithm required simple branching based on a biased coin toss. A way to run and track the actions within each branch was required. In theory, this could return a tree-like data structure that can be used to generate code. However, it was not possible to determine where branches end without comparing the structure of each branch. Similarly, keeping track of variable scope would require a large amount of implementation effort, at which point the main focus of the project would become writing a code transpiler.

A new approach was needed to provide ultimate flexibility without sacrificing too much performance. The supporting Numba feature for this approach was the `@jitclass` annotation. Jitclasses are an experimental feature that compiles Python classes to native data structures and supports function calls on the class instance. Instead of the Python API providing stubs for each action, it will offer implementations that work with or without Numba's just in time compilation. Another benefit of directly running the user's code is that any functionality that is not provided by the library can be created without requiring updates to the software package. These may include the usage of Python and Numpy functions supported by Numba and is useful for performing calculations based on the current load or ball index.

### 3.6 Basic and Intermediate Deliverables

The implementation was divided into two components - `core` and `simulation`. Each component was a module that could be imported by the user in either a Python file or Jupyter Notebook.

#### 3.6.1 Core: Load Balancing Algorithm Description

The `core` module contains implementations of the functions in Table 1. The file consists of three Python classes annotated with `@jitclass`, and helper functions annotated with `@njit`.

These classes are used to describe a load balancing algorithm. The algorithm is input as a Python function that takes in a `core.Ball`, the set of `core.Bins` and optional arguments. The method is called once for each of the $m$ balls and is expected to return the `core.Bin` selected by the algorithm. The functions defined in each class can be used to interact with the bins and retrieve the current properties of the system. Additionally, `core` helper functions such as the probability function can be used for further operations.

The `core.Ball` class represents the current ball that the algorithm is operating on and contains three properties. `m` is the total number of balls in the simulation and remains constant. `weight` is the weight of the current ball, an intermediate deliverable. The `position` is the position of the ball that is being thrown and is in the range $[1, m]$ for a sequential load balancing simulation.

The `core.Bins` class is a container that represents a subset of all the bins. It similarly contains a property, $n$ - the total number of bins in the simulation. The algorithm is initially provided with all the bins, an instance of this class with all the integer indices $[0, n)$. The class stores the load of the bins in a private array so that the load for each bin can be accessed internally. It provides four functions. `choose(d: int)` is used to choose a random subset of size $d$, and returns a new instance of `core.Bins` with the randomly selected indices. The `choose_one()` function is used to return a single, random, `core.Bin` instance. `split(n: int)` returns a list of `core.Bins` instances that represent the bins split into $n$ sub-arrays. To get the `core.Bin` instance with the minimum load, the `min()` function can be called.

The `core.Bin` class represents a single bin and contains the index of a bin. There is a `load` property that can be used to access the load for this particular bin.

One of the intermediate deliverables was to track another evaluation metric, communication complexity. In the implementation, this is equivalent to the number of times that the load for a bin was queried. This is achieved by associating a `core.Counters` object with every data structure so that applicable operations can update the global counters within. The `core.Counters` class currently contains a single `load` attribute that is initialised to zero when created. Operations such as fetching a bin's load with `core.Bin.load` will increase this number by one, and more complicated operations such as `core.Bins.min()` increase this by the number of bins in the subset as every bin's load has to be accessed and compared.

#### 3.6.2 Simulation: Experiment Framework

To run the simulation, the Python function must be as described in section 3.6.1 and should be passed into the `Simulation` class. This module handles the Numba JIT compilation of the user's algorithm and provides the necessary data structures.

The `Simulation` class determines the name of the algorithm based on the function name that was passed into it. After this, it runs the Numba `njit` function on the input to return a JIT-compiled version of the algorithm. The class has two uses - it overrides the Python `__call__` function, which allows it to be called directly. It also provides a useful function to run the simulation at varying capacities, `run(min_balls: int, max_balls: int, step: int | None, num: int | None, bins: int | None, *args)`. The run function performs the simulation with a varying number of balls and determines the step size based on its arguments. If `step` is provided, the simulation will run for every step of size `step` from `min_balls` to `max_balls`. If `num` is provided, this will be the number of samples generated from a geometric progression - useful to provide more data points at lower numbers. If `bins` is not provided, it will be set to `max_balls`.

Directly calling the simulation allows the user to provide a value for $m$ and $n$ and any extra parameters. Python's keyword arguments can be used to specify the weights of each ball, satisfying the intermediate requirement. When weights are not specified, the weight of each ball will be set to one. An option to repeat simulations and return averages was implemented to approximate the expected maximum load.

The algorithm is passed to a Numba accelerated internal function, `load_balancer` for execution. This function initialises the `core.Bins` instance that is passed to the balancer with the internal load array and counters set to zero. It then loops through $m$ times (once for every ball) and runs the user's algorithm to choose a bin. It provides the total number of balls, position and weight of the current ball, and the set of bins to the algorithm. The chosen bin is returned by the user's algorithm, and its load is incremented by the ball weight at the end of each iteration.

Finally, the array containing the load for each bin and the counters are returned to Python for analysis. The maximum load, minimum load and standard deviation are calculated from the load array, and the number of load accesses is determined from the `core.Counters` object. These results are contained in a `SimulationResult` class with the values of $m$ and $n$ for reference.

Another intermediate requirement was support for distributed load balancing scenarios. This was implemented as an extra parameter to the `Simulation` class. When the number of balancers is greater than one, the algorithm is run multiple times with the same inputs, to simulate each balancer performing the algorithm in parallel. Once this is complete, the changes to the bin's load are committed and used in the next round of load balancing. Although not a perfect approximation to the distributed load balancing problem (it assumes synchronicity between balancers), the solution can be used to give an intuition of expected performance in a distributed scenario.

### 3.7 Advanced Deliverables

Once the basic and intermediate deliverables were achieved, the system was parallelised to run faster on multicore systems. The final software package was used in the Jupyter Notebook for visualisation and analysis.

#### *3.7.1 Parallelisation*

The `Simulation` class runs the accelerated load balancing algorithm on a single thread. To improve the performance when running multiple experiments, a `ParallelSimulation` class was created. This class is initialised with a list parameter containing pairs of algorithms and arguments. If called directly, each core on the machine is assigned a `SimulationJob` for each algorithm.

The `run` function creates jobs for each step from `min_balls` to `max_balls`. These `SimulationJobs` are run on a pool of threads equal to the number of physical cores on the machine, and the status is reported with a progress bar that provides the current and estimated runtime. Since the size of each job varies, the system shuffles the order of jobs to help stabilise the progress bar and provide accurate estimates of runtime. Additionally, the results of this function are sorted by the number of balls before being returned.

Since a new Python process is initialised on each thread, the system has to run Numba's optimisations for each simulation once per thread. Once run, this is cached for use by later jobs on the same thread. Results are returned with their algorithm id so that they can be aggregated at the end based on the algorithm used to run them. The algorithm id is generated by concatenating the function name with the arguments used to run them. For example, the GREEDY[$d$] algorithm 'greedy_d' run with the parameter $d = 2$ would have the id 'greedy_d_2'.

#### *3.7.2 Jupyter Notebook*

A Jupyter Notebook was written to aid with the results. A notebook is a collection of Python and text cells that can be run in order or individually. They allow easy modifications of particular code sections and can be exported to an HTML or PDF format for sharing. A file `algorithms.py` is provided alongside the notebook and contains implementations of the different load balancing algorithms written with the software package. The code in the notebook shows how to use one of these pre-made implementations or write your own. A screenshot can be seen in Figure 7.

The data from the `Simulation` class is returned in a data structure that can be converted for the user's plotting library of choice. For plotting charts, the Matplotlib library was used due to its popularity and simplicity [26], and the examples show how this data can be converted into the format required by its line and bar charts.

The `tikzplotlib` library was included to convert the Matplotlib figures into PGFPlots code that renders charts in Latex [27]. The output from the library was used to produce the charts in this document.

### 3.8 Efficiency of Solution

The solution was compared against the C++ implementation to ensure that its runtime was still acceptable. Figure 8 shows that the simulation does run slower than a direct implementation in C++ but is, in fact, much faster than the Python equivalent. This is likely due to the extra number of function calls and data structures required to provide the simple API. This was concluded to be acceptable due to the

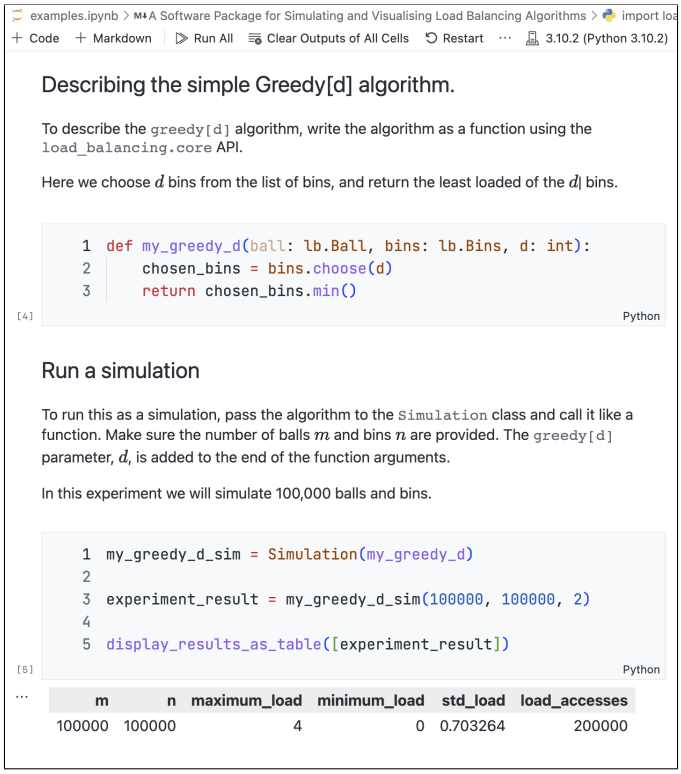


Fig. 7. Screenshot of a section of the provided Jupyter Notebook. The notebook walks through the steps of running a simulation and then recreates the results from this paper.

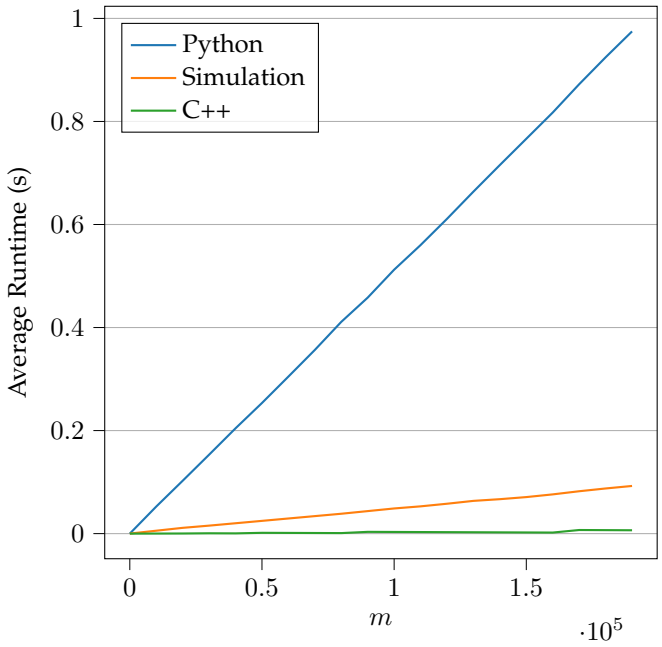


Fig. 8. Average runtime of 10 runs of GREEDY[2] in Python, the proposed simulation and C++. The simulation can be seen to be much faster than Python, but is slower than C++.

benefit of the low learning curve required to implement a load balancing algorithm with the system.

The simulation will be tested for correctness in section 4. By comparing the simulation's results to known data from the literature, it will be possible to identify any bugs in the implementation. Furthermore, the API design will be considered and evaluated based on how easy it is to create these experiments.

## 4 Results

The solution was applied to different load balancing algorithms based on the research in section 2. By comparing the experimental results produced by the software package with the literature, it will be possible to evaluate the correctness of the results.

### 4.1 Uniform Load Balancing

We will begin by analysing the classical balls into bins model, referred to here as the Uniform load balancing algorithm. This is the algorithm in which each ball is allocated a bin uniformly and independently at random. Algorithm 4 shows the simplicity of this algorithm when described using the produced software package. The function is called once for each ball and chooses one bin at random to return to the simulation system.

**Algorithm 4** Uniform described with the software package. The advanced deliverable, parallelisation, is used.

```
def uniform(ball: lb.Ball, bins: lb.Bins):
  return bins.choose_one()

data = ParallelSimulation((uniform, [])
  ↪  ).run(100, 70000100, num=40, repeats=5)
```

Raab and Steger [4], proved tight bounds for the maximum load for any value of $m$. We will focus on the singular result for the maximum load $M$ when $\frac{n}{\text{polylog}(n)} < m \ll n \log n$, which states for $\alpha > 1$ there is a low probability that $M > k_\alpha$ and for $0 < \alpha < 1$ there is a high probability that $M > k_\alpha$, where

$$k_\alpha = \frac{\log n}{\log \frac{n \log n}{m}} \left(1 + \alpha \frac{\log \log \frac{n \log n}{m}}{\log \frac{n \log n}{m}}\right) \quad \text{for } \frac{n}{\text{polylog}(n)} \leq m \ll n \log n \tag{1}$$

For this experiment, the simulation was run on 40 different values of $m$, with $n = 70000100$. The maximum value for $m$ was much smaller than $n \log n$, which means that Equation 1 should hold. Each data point was calculated as an average over five runs to approximate the expected maximum load. Since the standard deviation of the Uniform results is very low, there was less need to run a larger number of repeats. The results for the experiment shown in Figure 9 agree with the paper's suggestions since a value of $\alpha = 2$ was found to bound the data closely.

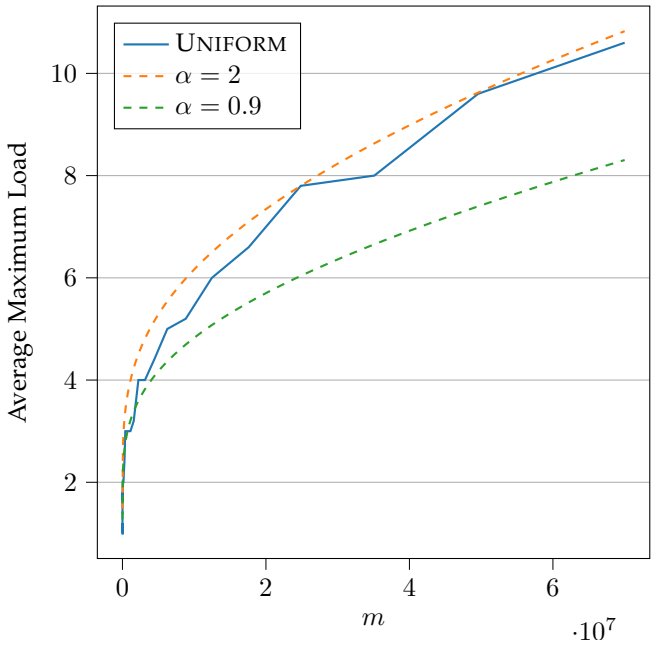


Fig. 9. Average maximum load of any bin over five repeats for $m < n$, $n = 70000100$. Values of $\alpha$ were chosen to bound the maximum load.

### 4.2 Always-Go-Left

Another area of investigation was to check that the simulation system produced similar results to those found in the Always-Go-Left paper discussed in section 2. Since the original paper aimed to prove how asymmetry helps reduce the maximum load in randomised load balancing, the experimental results should show an improvement in maximum load when compared to the Greedy[2] algorithm. Additionally, we will run a modified simulation with a fair tie-breaking strategy to validate the claim that the asymmetry provided by the tie-breaking strategy is the sole cause for improvement in maximum load. This strategy, Fair Tiebreak, will choose a random bin of the two choices when both bins are equally loaded.

These algorithms were implemented and executed in parallel using the software package, and the code to do this can be observed in Algorithm 5. The code shows that the software package's simple syntax can make more complicated algorithms easy to understand at a glance and can be written without Numpy or Numba experience. It is seen that in both algorithms, the bins are split into two halves, `left_bins` and `right_bins`, and one bin is chosen from each half. In the Always-Go-Left algorithm, the left bin (with the lower index) will be chosen if its load is smaller than or equal to the right bin. The right bin is only ever chosen if its load is strictly smaller. These algorithms require branching operations that would not have been possible to simulate with the early project approaches.

Our experimental results are extremely similar to those presented in the original paper, indicating that the simulation works correctly. The bar chart in Figure 10 shows the variation in the average maximum load for different algorithms when $m = n = 2^{20}$. Since $m = n$, we can expect the bins in the optimal solution to contain only one ball. None of the algorithms achieve this, but Always-Go-

**Algorithm 5** ALWAYS-GO-LEFT and FAIR TIEBREAK described with the software package.

```
def always_go_left(ball: lb.Ball, bins:
  ↪ lb.Bins):
  left_bins, right_bins = bins.split(2)
  left_bin = left_bins.choose_one()
  right_bin = right_bins.choose_one()
  if left_bin.load <= right_bin.load:
    return left_bin
  else:
    return right_bin

def fair_tiebreak(ball: lb.Ball, bins:
  ↪ lb.Bins):
  left_bins, right_bins = bins.split(2)
  left_bin = left_bins.choose_one()
  right_bin = right_bins.choose_one()
  if left_bin.load < right_bin.load:
    return left_bin
  elif left_bin.load > right_bin.load:
    return right_bin
  else:
    if lb.probability(0.5):
      return left_bin
    else:
      return right_bin

simulation = ParallelSimulation((uniform,
  ↪ []), (greedy_d, [2]), (always_go_left,
  ↪ []), (fair_tiebreak, []))
data = simulation(1048576, 1048576,
  ↪ repeats=100)
```

LEFT is the closest. We can see that the UNIFORM algorithm performs the worst with an average maximum load of 8.78 in our testing. From the experimental results, ALWAYS-GO-LEFT can be seen to improve on the GREEDY[2] algorithm with a decrease in average maximum load of 22.75%. In addition, the claim made in the paper regarding the tie-breaking strategy can be confirmed by these results since the FAIR TIEBREAK algorithm (a modified ALWAYS-GO-LEFT with a fair tiebreak) does not see any improvement over GREEDY[2].

### 4.3 Distributed Algorithms

For this section, the results from [15] will be evaluated experimentally by using the intermediate deliverable that allows multiple load balancers to be simulated.

The paper states that for the GREEDY[2] algorithm, the expected load gap between maximum load and minimum load of any bin is $\mathcal{O}(\log n)$ for $m \gg n$. We will begin by fixing the number of load balancers to the number of bins, $b = n$, and running the experiment for varying numbers of balls, $m > n$. Each experiment was run twenty times so that an estimate of the expectation could be determined.

Figure 11 contains the experimental results for different values of $n$. The dashed lines represent $C \cdot (\log n)$, which was found to best match the data when $C = 1.3$. The results clearly support the paper since once $m$ becomes large enough, the lighter solid lines representing the experimental data are bounded by their equivalent dashed lines. This result is interesting since the maximum load gap on any bin

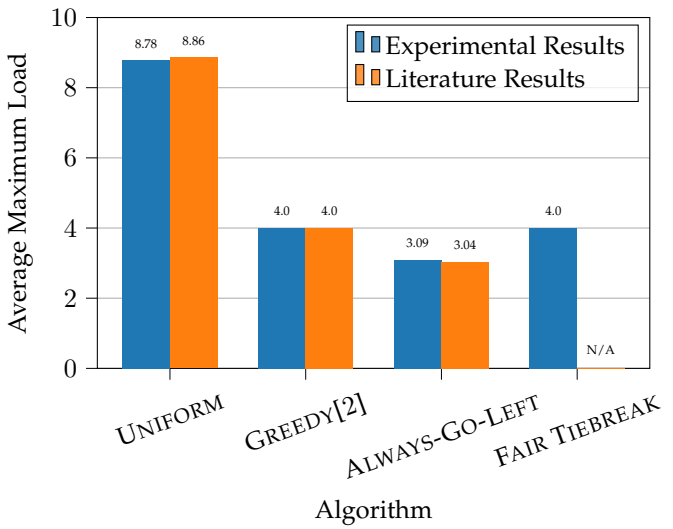


Fig. 10. Average maximum load for $2^{20}$ balls and $2^{20}$ bins over 100 repeats. The experimental results match those from the literature very closely.

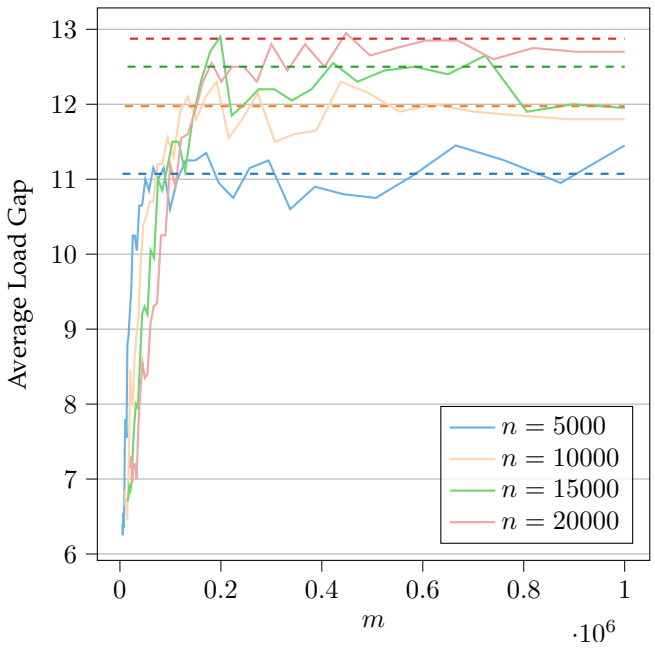


Fig. 11. Solid lines: Average gap between maximum and minimum load for the GREEDY[2] algorithm. Dashed lines: $1.3 \cdot (\log n)$. The dashed lines bound the experimental results closely.

does not depend on the value for $m$ and remains constant as $m$ grows.

It was also claimed that even with batches, the multiple-choice algorithm GREEDY[2] improves upon the UNIFORM algorithm. It was given that the expected load gap for the UNIFORM algorithm with $b = n$ was $\Theta(\sqrt{\frac{m \log n}{n}})$. When $C = 2.55$, this was found to be accurate since the dashed lines in Figure 12 closely match the experimental data. By refering to the scale in Figure 12 it can be seen that the expected load gap is much larger than in Figure 11. This indicates that the GREEDY[2] algorithm still provides an

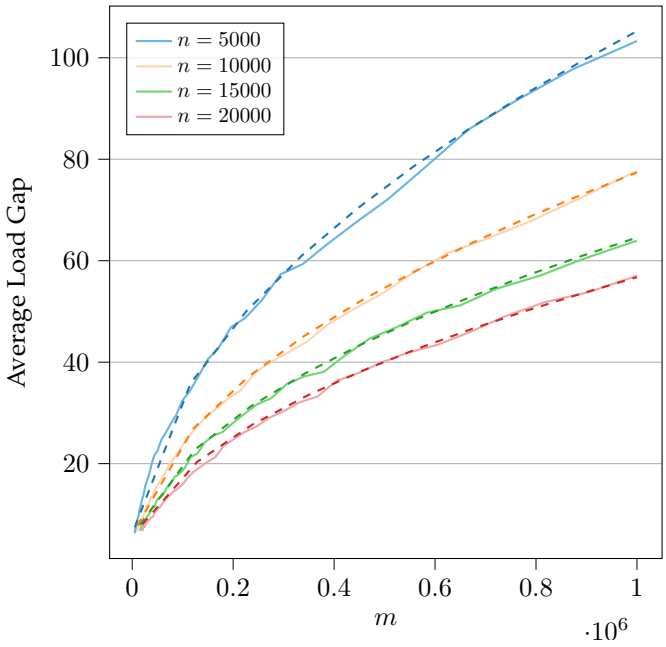


Fig. 12. Solid lines: Average gap between maximum and minimum load for the UNIFORM algorithm. Dashed lines: $2.55 \cdot (\sqrt{\frac{m \log n}{n}})$. The dashed lines are very similar to the experimental results.

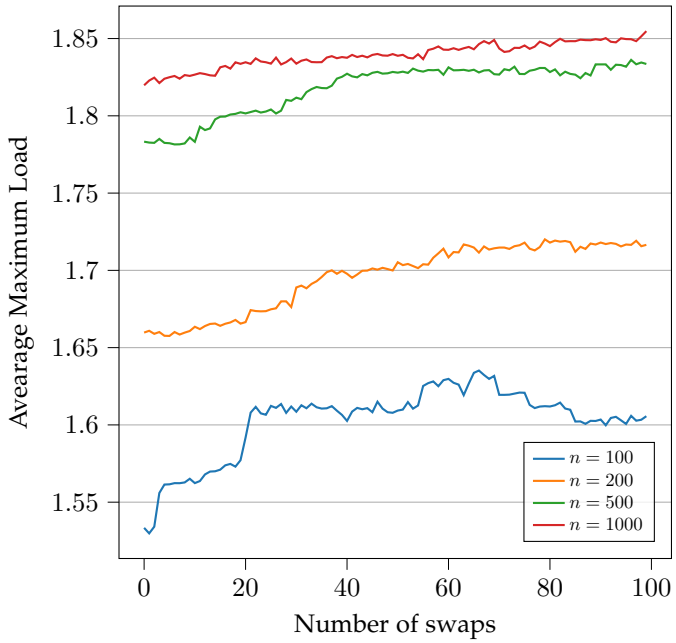


Fig. 13. Average maximum load as the weighted balls becomes progressively less sorted. When the number of swaps is increased, the average maximum load increases

improvement in expected load.

Although these results indicate that the GREEDY[2] algorithm may be suitable for use in a distributed load balancing system, there are assumptions made in the model that cannot be applied to real-world load balancing. For example, both the paper and the simulation assume that balls are thrown in synchronised batches and can retrieve the current load for the selected bins at the same time. In reality, the time taken to query selected bins load may not be constant, introducing stale data into the decision process. This could lead to oscillations in server load, a topic discussed in section 1.

### 4.4 Weighted Balls

Previous empirical studies showed that the order of allocation of weighted balls can influence the expected maximum load. In particular, results showed that allocating balls in decreasing weight order produced a load distribution with a lower maximum load [18]. In this section, we will perform similar experiments.

The experiment utilised the weighted balls intermediate deliverable of the software package, allowing the user to pass an array of weights representing each ball. Weights for the balls were initially sampled from a uniform distribution between 0 and 1 and sorted in decreasing order. The software package uses the order of the weight distribution when running the simulation, which is useful since this is required for this experiment.

The GREEDY[2] algorithm was tested for varying size experiments, where the number of balls and bins were both equal to $n$. For each value of $n$, the balls would undergo 100 swapping steps. The expected maximum load for each step was determined by running the simulation 10,000 times and was plotted in Figure 13.

The lines seen in our results are not as smooth as the results from the paper; however, the general idea that the expected maximum load will increase as the balls become less sorted can be seen by the upward trend. The differences may be since fewer repeats were performed than the 100,000 specified in the paper. Furthermore, our experiment was performed for only one distribution of weights and a single pass through the swapping algorithm. The paper's experiments may have averaged results across multiple sets of ball swaps. Hence this is an area of further research for the project.

The same paper showed that more balanced weight distributions produced a lower maximum load. An experiment was created to try and replicate these results, but the author was unable to reproduce the same results. It was believed that this was down to two factors. Firstly, the number of repeats may not have been enough to produce an accurate expected maximum load. Secondly, although multiple algorithms were used to balance the weights, it is possible that a different approach was used by the paper. Therefore, future work is to be done to replicate these results with the software package.

### 4.5 Communication Complexity

This section will compare the communication complexity of the algorithms explored in section 2. The number of times a bin is queried for its load is tracked by the software package and can be retrieved from the result's `load_accesses` attribute (an intermediate deliverable). As previously discussed, the communication complexity is an important metric since it can indicate the amount of data that needs to be transferred to make a load balancing decision, possibly slowing down execution. A high communication complexity would be detrimental to a web application since it may lead to the load balancing server becoming a bottleneck.

Table 2 shows the communication complexity and the maximum load of sequential, non-weighted load balancing simulations, repeated 100 times to calculate an average. As previously discovered, the UNIFORM algorithm performs worse than the others. However, it does not require any knowledge of the bin's load, making it applicable in distributed load balancing systems.

The paper for the $(1+\beta)$-CHOICE algorithm states that it is expected to have a $1+\beta$ communication complexity for each ball thrown, hence for this experiment the overall number of load accesses should be $(1+0.5)\cdot 2^{20} = 1{,}572{,}864$. This does not match our findings. The reason for this is likely because the simulation doesn't count the uniform allocation as a load query, so a better estimation for the software package would be $2 \cdot \beta$ queries for each ball - for this experiment, this would be $(2 \cdot 0.5) \cdot 2^{20} = 1{,}048{,}576$. This is much closer to the experimental results.

It appears that the THRESHOLD and ADAPTIVE algorithms provide a good trade-off between maximum load, and the load access count, performing almost identically in our experiment. The main difference between these two algorithms is that the ADAPTIVE algorithm does not require knowledge of the number of balls upfront and hence may be more suitable for real-life load balancing where the number of requests in a time period is unknown.

TABLE 2
Average Algorithm Communication Complexity for $n = m = 2^{20}$

| Algorithm | Maximum Load | Load Access Count |
|---|---|---|
| UNIFORM | 8.74 | 0 |
| $(1+\beta)$-CHOICE, $\beta = 0.5$ | 7.06 | 1,048,475 |
| GREEDY[2] | 4.00 | 2,097,152 |
| THRESHOLD | 2.00 | 1,201,834 |
| ADAPTIVE | 2.00 | 1,201,847 |

## 5 EVALUATION

The software was tested on different scenarios throughout its development, which allowed bugs to be discovered and fixed, producing software that the author is confident produces accurate results. The basic and intermediate deliverables were achieved by the project, and all but one of the advanced deliverables were implemented.

The results in section 4 show that the simulation can replicate and empirically verify results from relevant literature. This report hopefully demonstrates that the tool will be helpful to guide research into balls into bins protocols in the future.

### 5.1 Strengths

Upon surveying the relevant literature, several key algorithms and proofs were highlighted. Although the project was built with these algorithms in mind, the simulation system was found to be flexible enough to work for many types of load balancing algorithms. The ability to pass parameters into algorithms was a vital feature that enabled this flexibility since it allowed the GREEDY[$d$] algorithm to be described once and used for different values of $d$. Another benefit of the implementation approach was the support for more advanced Python features syntax in the algorithm description. The THRESHOLD and ADAPTIVE algorithms take advantage of this by using while loops and if statements in their description.

Although there has been no particular study into the project's ease of use, it is clear that the level of comprehension required to understand a load balancing algorithm is much lower when written with the software package. Since the API is simple and compact, users should find it much easier to design and test new balls into bins algorithms.

The package provides a consistent method for collecting and visualising experimental data. These experiments can be useful to determine bounds since the mathematical analysis for new algorithms is often complicated and time-consuming. Alternatively, experimental data can be used to verify proofs empirically. The Jupyter Notebook provides a framework for users less familiar with programming to set up experiments with minimal effort.

The software package provides the originally desired API, although the implementation was not as initially planned. Ideally, the simulation would run as fast as a lower-level language such as C++; however, as discussed in section 3, this was not possible without implementing a bespoke code transpiler. If the project's scope had been larger, this implementation approach could have been considered further. Even though the optimum runtime could not be achieved, the simulation system still runs much faster than the plain Python equivalent. Therefore, the project has been successful in balancing being simple to use and fast. The viability of these different code acceleration techniques was considered in-depth, and this analysis could be helpful for future projects.

### 5.2 Limitations

There was not much innovation in displaying the results and their visualisations. One of the advanced deliverables was the inclusion of the `ipywidgets` library for interactive charts - for example, a slider to view the load distribution over time. Unfortunately, this was not included due to time constraints. Similarly, alternative display styles were not investigated in the results section, and only line and bar charts were used due to their prevalence in the literature.

Further modifications to the simple model were not considered, such as dependencies between balls or restrictions on the bins that balls can be allocated into. Restrictions on bins would be useful to simulate situations where a web request requires a specific piece of hardware and can therefore only run on a subset of the servers. If these modifications were to be built into the project, care should be taken to avoid over-complicating the API.

A survey of potential users that are both established in and new to the balls into bins field could be performed to confirm that the API is easy-to-use. This could be achieved by describing multiple algorithms in either text or pseudocode and asking participants to code a simulation with the software package. A short survey would ask questions regarding the available package functions and the utility of the provided Jupyter Notebook. Feedback from this study could be used to improve and evaluate the software package.

### 5.3 Approach

Many of the limitations above arose due to time constraints, and it was unfortunate that many of the earlier attempts for code acceleration exceeded the project's scope. Consequently, more of the time spent on the project was focused on creating the simulation than visualising results. Although some interesting results were found and validated, the author would have liked to design their own load-balancing algorithm to compare with existing ones. Such a task would have required extensive reading to ensure that a proposed algorithm had not been seen before and, unfortunately, was not possible due to time constraints. The experience will help prevent the author from overcommitting to an implementation technique before its viability has been analysed.

Although there were difficulties encountered during the process, the implementation part of the project went reasonably smoothly and allowed the author to learn and experiment with many software technologies. A linear approach was taken to the project - most of the results section was written and planned after the methodology was completed. This allowed the author to evaluate the flexibility of the software package but required more time spent planning up-front. Regardless, the results produced are exciting and show that the project can be applied to verify relevant literature.

## 6 Conclusion

The project has been successful in its primary aim of providing a software package with an easy-to-use API that can run fast balls into bins simulations. The software was used to verify results in the literature, and in the case of the Always-Go-Left algorithm, it was able to extend the experimental results provided in the paper.

Section 3 evaluated the different approaches for accelerating Python code in a simulation environment. The runtime of many programming languages and tools were compared, and reasons for slowdowns were discussed. Ultimately, the complexity required to create a flexible algorithm description from Python source code meant that both the C++ simulation engine and code generation approaches could not be continued. For maximum flexibility, an API that utilised Numba's Jitclasses was designed and evaluated by its runtime speed. Although this system was slightly slower than the previous approaches, it supported more possible algorithms. The research into the more advanced techniques could be used to guide further work on this project or similar code generation projects.

Regardless, there were still challenges and tricks required to simplify the API in the chosen solution to balance the runtime and user experience. These challenges were overcome by utilising the flexibility of the Numba JIT compiler, proving that it is a powerful and valuable tool for accelerating Python code.

Section 4 confirmed some interesting findings from the literature. In section 4.2 the experimental results from running the Always-Go-Left algorithm were almost identical to the paper and showed that the asymmetric tie-breaking strategy leads to a reduction in the expected maximum load. Furthermore, this experiment provided empirical evidence to confirm that switching the fair tie-breaking strategy with an asymmetric strategy is required to see this reduction. In section 4.3, tests for a distributed load balancing system were conducted. These showed that for $m \gg n$, the average gap between minimum and maximum load does not depend on the value of $m$ in a distributed load balancing system.

There is undoubtedly more work that can be done to extend the convenience and functionality of the software package. For example, the system cannot simulate dynamic load balancing where the number of balls in each bin decreases at a defined rate. The addition of such a feature would allow more complicated models to be simulated, such as the 'Supermarket model' [9].

A set of standardised tests could be defined to generate a report for each load balancing algorithm. And different charts could be produced with one click instead of modifying the existing Notebook code. If these features were implemented, the software package could be a one-size-fits-all solution for load balancing algorithm analysis.

It is hoped that the project can be used in the future to reduce the barrier to adding empirical data to research and encourage more experimentation in the field.

## REFERENCES


[1] *Availability and load balancing in cloud computing*, vol. International Conference on Computer and Software Modeling, Singapore 14. IACSIT Press, 2011.

[2] S. Afzal and G. Kavitha, "Load balancing in cloud computing–a hierarchical taxonomical classification," *Journal of Cloud Computing*, vol. 8, no. 1, p. 1–24, 2019.

[3] AmazonWebServices, "How elastic load balancing works." [Online]. Available: https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html

[4] *"Balls into bins"—A simple and tight analysis*, vol. International Workshop on Randomization and Approximation Techniques in Computer Science. Springer, 1998.

[5] S. Fischer and B. Vöcking, "Adaptive routing with stale information," *Theoretical Computer Science*, vol. 410, no. 36, p. 3357–3371, 2009.

[6] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, and S. Corlay, *Jupyter Notebooks-a publishing format for reproducible computational workflows.*, 2016, vol. 2016.

[7] G. H. Gonnet, "Expected length of the longest probe sequence in hash code searching," *Journal of the ACM (JACM)*, vol. 28, no. 2, p. 289–304, 1981.

[8] *Balanced allocations*, vol. Proceedings of the twenty-sixth annual ACM symposium on theory of computing, 1994.

[9] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and . . .* , 2001.

[10] *On balls and bins with deletions*, vol. International Workshop on Randomization and Approximation Techniques in Computer Science. Springer, 1998.

[11] B. Vöcking, "How asymmetry helps load balancing," *Journal of the ACM (JACM)*, vol. 50, no. 4, p. 568–589, 2003.

[12] Y. Peres, K. Talwar, and U. Wieder, "The (1+ beta)-choice process and weighted balls-into-bins," *Proceedings of the twenty-first annual ACM . . .* , 2010.

[13] A. Czumaj and V. Stemann, "Randomized allocation processes," *Random Structures and Algorithms*, vol. 18, no. 4, p. 297–331, 2001.

[14] *Balls-into-bins with nearly optimal load distribution*, vol. Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures, 2013.

[15] P. Berenbrink, A. Czumaj, M. Englert, T. Friedetzky, and L. Nagel, "Multiple-choice balanced allocation in (almost) parallel," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2012, pp. 411–422.

[16] D. Los and T. Sauerwald, "Balanced allocations in batches: Simplified and generalized," *arXiv preprint arXiv:2203.13902*, 2022.

[17] *Balanced allocations: the weighted case*, vol. Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, 2007.

[18] P. Berenbrink, T. Friedetzky, Z. Hu, and R. Martin, "On weighted balls-into-bins games," *Theoretical Computer Science*, vol. 409, no. 3, p. 511–520, 2008.

[19] *ppsim: A software package for efficiently simulating and visualizing population protocols*, vol. International Conference on Computational Methods in Systems Biology. Springer, 2021.

[20] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta, "Computation in networks of passively mobile finite-state sensors," *Distributed computing*, vol. 18, no. 4, p. 235–253, 2006.

[21] *Pyroomacoustics: A python package for audio room simulation and array processing algorithms*, vol. 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2018.

[22] C. Harris, K. Millman, S. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. Smith, R. Kern, M. Picus, S. Hoyer, M. van Kerkwijk, M. Brett, A. Haldane, J. Del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. Oliphant, "Array programming with numpy," *Nature*, vol. 585, no. 7825, p. 357–362, 2020.

[23] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science and Engineering*, vol. 13, no. 2, p. 31–39, 2010.

[24] S. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," *Proceedings of the Second Workshop on . . .* , 2015.

[25] LLVM, "The llvm compiler infrastructure - users." [Online]. Available: https://llvm.org/Users.html

[26] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science and engineering*, vol. 9, no. 03, p. 90–95, 2007.

[27] C. Feuersänger, "Manual for package pgfplots," vol. 17, 2011. [Online]. Available: http://www.ctan.org/tex-archive/help/Catalogue/entries/pgfplots