ABSTRACT
Bandwidth availability, device count increases and new applications such as machine learning have resulted in a steady growth of cloud computing needs, that data centers are required to answer. Among the available axes of improvement, data center scheduling acts as a virtually free technique to optimize resource utilization. Sparrow [1], proposed in 2013, is a distributed scheduler with on the fly decisions that adapts the power of two choices [2] for load balancing to scheduling. This report aims at reproducing some simulation results from Sparrow and to discuss the authors’ adjustment of the power of two choices for their implementation.

1 INTRODUCTION
Modern data centers are mostly composed of clusters of commodity servers that need to work together to achieve desired compute capabilities[3]. The abstraction for workloads in these data centers is that users submit to clusters independent jobs, broken down in tasks, along with resource requirements (e.g. how much CPU, memory, or which data each task needs). This idea is best illustrated by the cloud computing infrastructures that expose compute only frameworks where users submit jobs with no carry over state.

Using arrays of commodity servers means that one job should not run on a single machine but rather that data centers should allocate work with a per-task (or finer) compute granularity. Scheduling is the problem of efficiently distributing these tasks on the available resources. Solutions focus on different aspects of resource usage optimization, such as latency [1] or total resource utilization [3–5]. Some solutions also take into account data dependence between tasks [4] and suppose schedulers submit a directed acyclic graph of computation to the scheduler [6]. Some solutions advocate for centralized designs for schedulers to maximize the information given to the scheduler; others argue that task frequency and latency requirements impose the use of distributed schedulers.

Sparrow [1] is a distributed scheduler that leverages and adapts the power of two choices for load-balancing [2] to make decisions. More precisely, whenever a job with \( t \) tasks is submitted to be scheduled, Sparrow probes \( 2t \) servers and select the \( t \) least loaded ones. Tasks are added to the queue of the selected servers and wait for resources to be available. To deal with the problem of stragglers—one or a few late tasks in a job taking significantly more time than expected and preventing the job from completing—, Sparrow waits for servers to signal that they are free before locking the assignment (henceforth referred to as late binding). This also avoids race conditions between different instances of the scheduler, but means that if tasks require data, the servers will be idle while the data is being transferred to the selected server.

This report presents a reproduction of figures 3, 4, and 5 from Sparrow in section 2, extensions and analysis of the simulation environment from Sparrow in section 3, before presenting potential future work in section 4.

2 REPRODUCED RESULTS
The code from the authors of Sparrow is publicly available online\(^2\). I used this code as a starting point to reproduce their figures. Due to the code being in Python 2.7, I had to make a few modifications to be able to reproduce their results\(^3\). Reproductions of figures 3, 4 and 5 from Sparrow are respectively presented in figures 1, 2, and 3.

In figure 1, Random corresponds to choosing a server at random for each task. Per-tasks corresponds to polling 2 servers for each task and choosing the one with the lowest expected completion time. Batch is similar except that for each job with \( t \) tasks, Sparrow polls \( 2t \) servers and chooses the \( t \) with lowest expected completion time. Batch+Late binding corresponds to batch scheduling with the late binding procedure highlighted in section 1. Omniscient represents the optimal response time from a central omniscient scheduler.

Reproductions of figures 4 and 5 from Sparrow are based on the formulas given in table 2. They exactly match the figures in the paper; while it is not clear in the paper that the figures are based on the formulas rather than on simulations, figures 2 and 3 appear to confirm it. The authors explicitly mention wait time in the experimental section, but do not show experimental graphs to prove that the assumptions required to get the formulas from table 2 hold.

3 EXTENSIONS
This section presents extensions of figure 3 testing the interpretation of [2] by the authors of Sparrow, a proof of the third equation from table 2 of Sparrow, and an analysis of the relevance of load as defined in Sparrow as a metric.

---
\(^2\)https://github.com/radlab/sparrow

\(^3\)See https://github.com/BenoitPitClaus/SparrowReproduction. I added Bruce to the collaborators since I had his ID from PA1; if either Nick or Sachin wishes to have access, please do not hesitate to email me.

---

See AWS Lambda or Azure Functions
3.1 Choice of Polling Ratio

The power of two choices [2] main result was that polling 2 random servers and choosing the least loaded instead of choosing 1 at random significantly increased performance in load balancing applications. The result is on a per-flow basis.

Sparrow adapts this result to scheduling: when a job is to be scheduled, Sparrow sends probes to twice as many random servers as there are tasks in the job and schedules the task on the least loaded one. The impact of probe ratio on response time for heavy loads is shown in figure 14 of the paper in their experimental setup.

While the initial power of two choices did not allow for granularity in the choice of servers polled (you cannot poll 1.5 servers), polling for batches of tasks enables finer granularity in the number of servers polled. The impact of the polling ratio on the response time vs load curves is shown in figure 4 for regular batch scheduling and figure 5 for batch scheduling with late binding. These results are averaged over 11 iterations. Not taken into account in these figures is the fact that the polling frequency can have an impact on the congestion of the network and slow down data transfers between nodes.

As expected, $r = 1$ has the same behavior as random polling, since it translates to polling exactly as many servers as there are tasks, and leaves no place for choice. Without late binding, we start seeing an improvement compared to per-task power of two choices polling around $r = 1.4$. While the previous experiments presented in the project report showed a decrease in the response time at 95% load, averaging over multiple runs show that this behavior was a random artifact. A point for 99% load was added compared to the previous report and to the paper to investigate this behavior but resulted in confirming that the decrease was due to variance in the simulations.

Figures 4 and 5 show that other polling ratios than $2t$ are acceptable. While increasing polling frequency above $2$ does not seem to yield significant benefits, reducing the polling frequency to $1.8$ does not reduce performance significantly, and might even have better performance at very heavy loads.
The sweet spot from Sparrow’s figure 14 corresponding to the trade-off between added messaging and oversampling does not appear on the simulation figures since the simulations do not take messaging into account.

### 3.2 Probability of zero wait time for Batch Sampling

As in the Sparrow paper, let:

- \( \rho \): the load, or fraction of non-idle workers,
- \( d \): the ratio of probes used per task,
- \( m \): the number of tasks per job.

The probability of experiencing zero wait time for a job using batch sampling is the probability that the \( m \) least loaded workers among the \( dm \) polled workers are idle, which is equal to the probability that at least \( m \) workers among the \( dm \) polled ones are idle.

The probability of exactly \( i \) workers being idle among the \( dm \) polled is \( \binom{dm}{i} (1 - \rho)^i \rho^{dm-i} \).

Using:

\[
1 = 1^{dm} = ((1 - \rho) + \rho)^{dm} = \sum_{i=0}^{dm} \binom{dm}{i} (1 - \rho)^i \rho^{dm-i},
\]

we get:

\[
P(m \text{ least loaded in } dm \text{ are idle}) = 1 - P(\text{at most } m - 1 \text{ in } dm \text{ idle})
\]

\[
= 1 - \sum_{i=0}^{m-1} P(\text{exactly } i \text{ workers idle among } dm)
\]

\[
= 1 - \sum_{i=0}^{m-1} \binom{dm}{i} (1 - \rho)^i \rho^{dm-i}
\]

Thus the probability of experiencing zero wait time:

\[
\sum_{i=m}^{dm} \binom{dm}{i} (1 - \rho)^i \rho^{dm-i}.
\]

### 3.3 Characterization of load as a proxy for arrival rate

Sparrow’s authors define load as the fraction of non-idle workers and evaluate the mean request arrival rate as the load times the number of workers divided by the number of tasks per job times the mean task service time (this formula, shown in table 2, probably comes from Little’s Law \([7]\)). While Little’s Law is independent of the service distribution, it measures the total load as the amount of computing to be done divided by the amount of available computing power. This is different from the per core Boolean: “is this core idle?” used as load in the paper.

The primary metric a data center will see is the request arrival rate. If we consider the worst possible scheduling algorithm (schedule all tasks from all jobs to a single worker), the load will be constant regardless of the request submission rate, and equal to the inverse of the number of workers.

From that observation I wanted to see how well the fraction of non-idle workers reflected the arrival rate. This can be broken down into two question: (i) is there a linear relationship between arrival rate and load (at least until we reach a fully loaded state), and (ii) for the considered schedulers of figures 4 and 5 from Sparrow, does this relationship depend on the type of scheduling.

To test this, I consider \( N \) jobs with arrival modeled as a non-stationary Poisson process. The inter-arrival time is thus given by an exponential distribution. The mean of this exponential distribution is determined empirically to ensure the corresponding loads span from 0 to 1. The number of tasks per job, and the number of cores per worker are left as parameters. Tasks are 100s long.

I simulate each job coming in one after the other spaced by random samples from an exponential distribution. The corresponding tasks are scheduled according to either random, per-task, or batch scheduling. From when the first task
completes, to when the last job is submitted, I measure the fraction of workers with at least one idle core, and average over the time of the experiment. The time window is chosen as such to consider only steady state. Results are averaged over 5 iterations.

Results from these simulations are presented in figure 6. These graphs show (i) that load is a linear function of arrival rate per server, but only for loads under about 0.8, and (ii) that function might be different for random scheduling compared to the two other scheduling strategies. While these results were obtained by averaging multiple runs, the observed differences do not look to be significant, except maybe at high load.

As expected, the load is directly proportional to the number of tasks per job: for 1 core, a fully loaded cluster corresponds to about $1.2 \cdot 10^{-3}$ 10-task jobs per worker per units of time, and about $1.2 \cdot 10^{-4}$ 100-task jobs per worker per units of time. A similar proportionality is observed for the number of cores.

It therefore appears that load as defined in the paper is a good proxy for arrival rate using random, per-task, and batch scheduling. However, if unexpected results from other experiments where to appear a high load, looking at arrival rate might be a good lead.

### 4 Future Steps

The project proposal mentioned exploring the effects of having tasks of different duration in each job. This aspect is still to be investigated.

Section 3.3 focused on schedulers that were shown in figure 4 and 5 from Sparrow. For completeness, Omniscient and Batch with late binding should be analysed as well.

Finally, taking into account data transfer costs and network congestion would be required to provide a full analysis of how Sparrow would behave in a production environment.

---

**Figure 6:** Fraction of non-idle cores. The job arrival rates are measured as fractions of the tasks duration. For example, here, tasks are 100 units of time long, and for 1 core-workers and 10 tasks per job, we reach a fully loaded cluster when about $1.2 \cdot 10^{-7}$ jobs arrive per worker per units of time in average.

---

### References


