Benchmarking and Simulating the Fundamental Scaling Behaviors of a MapReduce Engine

Brenton Walker
Institut für Kommunikationstechnik (IKT)
Leibniz Universität Hannover
Hannover, Germany
brenton.walker@ikt.uni-hannover.de

***Abstract*—We present MRSperf, a tool for running simple stochastically-controlled streams of jobs on a Spark cluster, and forkulator, a modular simulator for models of parallel processing. While it is common for networking researchers to build models and experiments on top of simple stochastically well-defined building blocks, research and development in MapReduce systems tends to be more divergent, either focusing on emulating realistic but very complex applications, or on theoretical models that may be far from accurate representations of real MapReduce system architecture. The purpose of these tools was originally to validate and guide the development of theoretical models in Network Calculus, but we believe the tools have more general utility. We use these tools to compare the popular Fork-Join model to the Non-Idling Single-Queue model, which more accurately reflects the behavior of the default task manager in Apache Spark, and propose future directions for their development.**

I. Introduction

When researches experiment with, or simulate communications networks for the purpose of developing or validating models, they usually start with models of transmissions, delays, and packet losses that are non-deterministic, but still have some well-defined statistical properties. For example inter-packet arrival times may be modeled with an exponential or Weibull distribution and packet losses may be modeled as a two-state Markov process. Experimenting with these types of models is supported in simulators like ns–3 [1], network emulators such as Emulab [2], and commercial link emulators such as PacketStorm.

Experimentation and development on parallel computing systems, on the other hand, has seen more divergent focii. On the systems-oriented end we see benchmarking suites such as BigPetStore and SparkBench [3], [4] and the unit and integration testing tools included with MR engines like Apache Spark [5]. These benchmarking suites try to give a dataset and workloads representative of an operational cluster. For the purposes of evaluating models of parallel computation, however, the results are dauntingly complex. On the theoretical end there are a variety of models meant to capture the synchronization constraint of parallel processing, and seemingly small differences in the models can lead to drastically different scaling behavior. These simple models are usually simulated, but to our knowledge no one has made a comparison to the behavior of a real MapReduce system. We find a similar focus in MapReduce simulators, such as MRPerf [6] and MRSim [7] which aim to simulate the complex workloads a cluster might encounter in the greatest detail possible.

ISBN 978-3-901882-94-4

Our work runs in a different direction than most systems-oriented benchmarking work, which aims to capture diverse and operationally realistic system behavior, and most theoretical work, which focuses more on general models. Therefore we include some motivational argument for our research.

Figure 1 shows schematics and scaling behavior for three important models of parallel processing, **Split-Merge (SM)**, **Fork-Join (FJ)**, and **Non-Idling Single-Queue (NISQ)**. All three appear to be reasonable models of parallel computation, but their scaling behavior is drastically different. The first and last models, SM and NISQ, include results from experiments on an Apache Spark cluster. This demonstrates that these two models of computation are achievable on such a cluster, and that two programs running on the same cluster, doing the same type of computation, can scale completely differently. The computation being done in this case has negligible I/O requirements, so behavior of the program gives a baseline for how programs of this structure can scale. It can also work in the opposite direction; for example, observing scaling behavior matching FJ in a program could help a programmer to identify processing constraints in their program. So while operationally realistic benchmarking suites are essential for debugging and optimizing programs and system parameters, a set of more basic, fundamental, and statistically controllable workloads is needed to fully understand the properties of MR systems and programs and their design points.

In this paper we present `forkulator` [8], a modular simulator for parallel processing models, and `MRSperf` [9], a loading tool for Apache Spark that submits jobs and tasks with statistical properties and constraints matching the assumptions of many theoretical models. Both tools were developed to validate and winnow the results in [10], but we see a lot of potential future uses, and consider both to be works-in-progress.

A. Models of Parallel Processing

The most studied model we see is the **Fork-Join (FJ)** model shown in the center of figure 1. Incoming jobs are divided into k tasks which are queued at k parallel workers, and serviced as the workers become available. Besides parallel computing

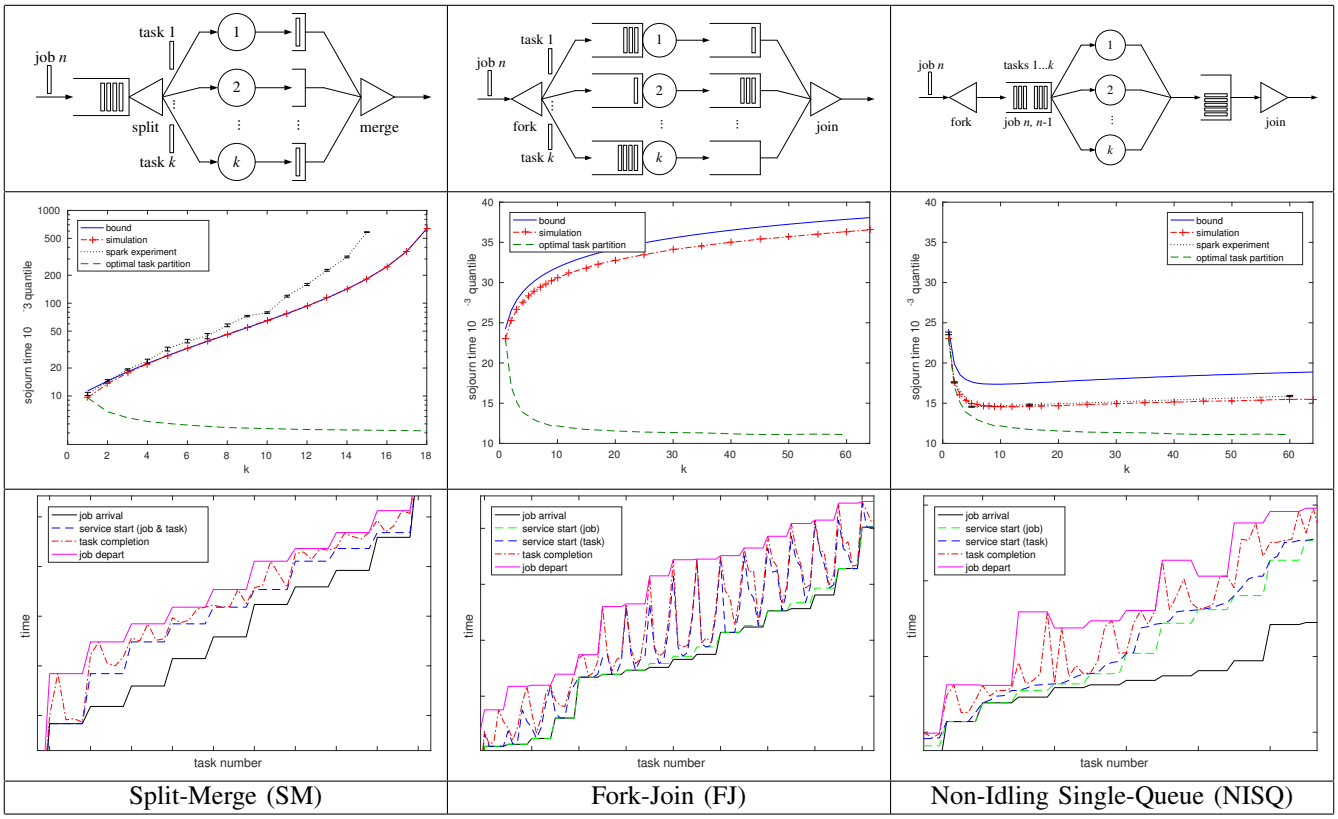


Fig. 1. Three models of parallel systems. Their schematics, 10^{-3} quantile of their sojourn times for increasing numbers of workers (k), and example experiment paths. Time units are in seconds.

systems, this model is cited as capturing the parallelism and synchronization constraint of such diverse real-world systems as supply chain networks, multipath routing, RAID storage, and military airfield coordination [11]. Since tasks are queued at the individual workers, it is possible for a worker to sit idle while tasks are waiting at other workers. However this property also guarantees that jobs must depart in the order they arrive; they cannot overtake each other.

A closed form solution for the mean sojourn time of an FJ queue is only known for $k = 2$ with exponential arrivals and service times [12], [13]. A great deal of research has been done to approximate and find bounds for FJ queues in more general settings [14], [15], [16], [17], [18], [19], [20]. The figure also shows the scaling behavior of the sojourn time for the FJ model for varying degrees of parallelism. The plot shows results from simulation and a bound based on network calculus [21], [22]. In the case of exponential arrivals and service times, the quantiles of the sojourn time grow logarithmically with k. For comparison, the figure also includes simulated sojourn time data for jobs with an optimal task division. That is, jobs with the same distribution of total job size, but with all tasks equally sized. Under these conditions all three models behave the same.

We have not found a way to make Spark behave like a Fork-Join queue. It seems that the Fork-Join model makes better sense in a context like multipath routing, where, once a packet is sent along a route it cannot be recalled, and the delay along each route is not fungible. In the case of Spark, the default task manager holds a queue of tasks, and assigns them to the cores on the workers as they become available. The tasks are not restricted to any particular worker. We call this the **Non-Idling Single-Queue (NISQ)** model. The right side of figure 1 shows a schematic of this model and a plot of its sojourn time for increasing degrees of parallelism. We see a large initial benefit from load balancing, and then a gradual increase in sojourn times because of the synchronization constraint. The NISQ model was studied in [23] and more recently the bound shown in the figure was derived using network calculus [10].

The left side of figure 1 shows a **Split-Merge (SM)** model. Some authors refer to this as a "blocking" Fork-Join. In this model each job must wait for all of the tasks from the previous job to finish before it can begin service. This means that if there is a single straggler task, all the workers must sit idle waiting for it to finish. When we have exponential task service times, this model will be equivalent to a single queue whose service times are the maximum order statistic of k exponentials. The dramatic increase in sojourn time with k is apparent from the figure.

A (im)properly implemented Spark program can behave like a Split-Merge system. Sojourn time data from such a Spark program, which is doing the same computations as the NISQ example, with jobs arriving at less than half the rate

of the NISQ example, are shown in figure 1. Also plotted are simulation results and a bound derived using network calculus [21].

The queuing of programs in a cluster may also behave like a Split-Merge queue; a driver program places a number of executors on the workers, and those executors block a certain number of cores, whether they are being used or not. Clearly a system that behaves like an SM queue is very sensitive to intra-job task size variation and should be avoided when possible.

II. MRSperf Spark Experimental Tools

MapReduce Spark Performance Tool, MRSperf, is a set of tools designed to set up a Spark cluster and submit jobs and tasks in a streaming context matching the assumptions of queuing theoretic models of parallel systems. Some key aspects that are different from normal operation of a batch-style Spark program:

1. MRSperf executes as a single stream processing application. When an application is submitted to a Spark cluster, it claims a certain number of cores on a certain number of workers by placing executors on the workers. These executors are held for the duration of the application whether they are being used or not[1]. The MRSPerf application claims a configurable number, k, of cores and holds them for the entire experiment. It then launches a separate thread for each job as it arrives. The degree of parallelism (number of tasks per job) is controlled by creating a trivial RDD with k slices with `parallelize()`, and then running a `map()` on that RDD. Finally an aggregate function such as `collect()` or `count()` must be called on the result to force the job execution. An example of an empty job is shown below.

```
def runEmptySlices(
    spark:SparkContext,
    numSlices: Int,
    serviceTimes: List[Double],
    jobId: Int): Long =
{
 spark.parallelize(1 to numSlices, numSlices)
   .map { i =>
    val jobLength = serviceTimes(i-1)
    val startTime =
        java.lang.System.currentTimeMillis()
    val targetStopTime = startTime +
        1000*jobLength
     while
         (java.lang.System.currentTimeMillis()
         < targetStopTime) {
       val x = random * 2 - 1
       val y = random * 2 - 1
    }
    val stopTime =
        java.lang.System.currentTimeMillis()
    1
   }.count()
}
```

[1]In Spark there are some options for dynamic allocation and deallocation, but this happens after idle times on the order of one minute, which is not suitable for our purposes.

2. All executors are single-core. Normally each worker node in a Spark cluster runs an instance of org.apache.spark.deploy.worker.Worker which is allocated most, or all, of the cores on the node. When an application is submitted to the cluster it may place an **executor** on that worker that claims up to the maximum number of cores. Each executor runs inside a single JVM, and therefore the threads of an executor can potentially benefit from their shared memory space. In order to match the queuing theoretic model of independent identical workers, MRSperf runs several workers on each node, each in a separate Docker container, and each allocated a single processor core. A Docker container [24] is like a lightweight virtual machine, and are often compared to OpenVZ [25] or FreeBSD Jails [26]. The docker container specification we use is based on the docker containers used for Spark integration testing.

For the Spark master to communicate with the workers reliably, each worker needs to be allocated a distinct IP address. In Docker the default is for the host to act as a NAT for the container, which makes the container unreachable from outside the host. Our solution is to configure the ethernet interface of each worker node with enough IP addresses for the number of workers it will host, and we run the Docker containers in host mode. This means that the processes in each container have full access to the network stack. Then we start each worker in a separate container and configure them to listen on distinct IP addresses.

3. The job arrival process is a configurable stochastic process. MRSperf currently supports constant rate, exponential, Erlang-k, and Weibull inter-arrival times. Jobs are forked from the main thread of the driver program. The tool records the time that each job is submitted, handles its inter-job processing (compute next job arrival time, write out any data collected from completed jobs, etc), and then sleeps for the remainder of the inter-job time. This means there is effectively some small minimum inter-job time because of the processing done between job submissions, but it eliminates the potential long-term drift in the process because of inter-job processing.

4. The task service times are configurable stochastic processes. MRSperf supports the same process models as for inter-arrival times. MRSperf also supports multi-stage jobs, and in that case there is an option that tasks in subsequent stages have either statistically independent, or identical service times. During its run time each task simply generates random points in the unit square (these lines of code originated from the SparkPI example script), and repeatedly checks if its desired runtime has been reached. The time resolution is 1 millisecond, the same as Spark's logging.

MRSperf also includes some tools for setting up and launching Docker containers and processing Spark event logs to produce job data in tabular form and experiment paths.

III. Forkulator Parallel Systems Simulator

Forkulator is an event-driven simulator for a variety of parallel processing models. The job inter-arrival and task service times can be constant, exponential, Erlang-k, Weibull,

or variations on Gaussian distributions. There is also a service time model that combines a service time distribution and a random overhead (scheduler delay) distribution. Both inter-arrival times and service times can be regulated through leaky buckets. The simulator has a configurable sampling interval and uses a warm-up period that is $10 * sampling\_interval * number\_of\_stages$. In our results we use an interval of 100 jobs. The simulator can also be run on a Spark cluster, in which case each slice of the simulation initializes with its own warm-up.

Forkulator supports several queue types. We have already described the Split-Merge, Fork-Join, and NISQ models. It also has implementations of (k, l) systems, where only l out of k tasks need to complete for a job to depart, multi-stage versions of the standard queue types, and "thinning" servers where there are multiple servers, but each job must run on a single server, and optionally be resequenced before departing.

The simulator outputs a log of the waiting, service, and sojourn times of the sampled jobs. It can optionally also produce a full experiment path containing arrival, start, completion, and departure times for every task of every job.

IV. Experimental Results

The simulation and Spark experiment results in figure 1 were generated with forkulator and MRSperf. The Fork-Join and NISQ examples use an arrival rate of $\lambda = 0.7$ and a service rate of $\mu = 1.0$. The Split-Merge example has $\lambda = 0.28$ and $\mu = 1.0$. The simulation datapoints were each computed from at least 10^9 jobs which were sampled at intervals of 100 jobs to give at least 10^7 samples.

The Split-Merge and NISQ cases are the only ones with data from both the simulator and Spark experiment. The Split-Merge Spark datapoints are each computed from 172,800 jobs sampled at intervals of 10 jobs. The NISQ Spark datapoints are each computed from 604,800 jobs sampled at intervals of 100 jobs. Error bars for quantiles are computed using the method described in [27] section 2.2.2.

The SM and NISQ cases demonstrate good agreement between the simulator and Spark. In these experiments we did not try to simulate scheduler or task deserialization overhead, so we see the sojourn times of Spark experiment tend to be slightly larger than the simulated ones in the NISQ case. The SM example diverges more. This is partly because even with a load of $\lambda/\mu = 0.28$, the system is close to being unstable at $k = 15$ servers, and the system overhead makes a more dramatic difference. The other reason is that to generate data more quickly we ran the SM experiment with time units of 0.1sec versus 1.0sec for the NISQ case. Therefore the relative overhead in the SM experiment was 10x higher.

The NISQ results in figure 1 also demonstrate the ability of MRSperf to run larger-scale experiments. In this case we ran 15 workers on each of four 24-core servers for a total of 60 workers. We think this is a fairly effective way to realistically experiment with MapReduce scaling behavior using a small hardware setup.

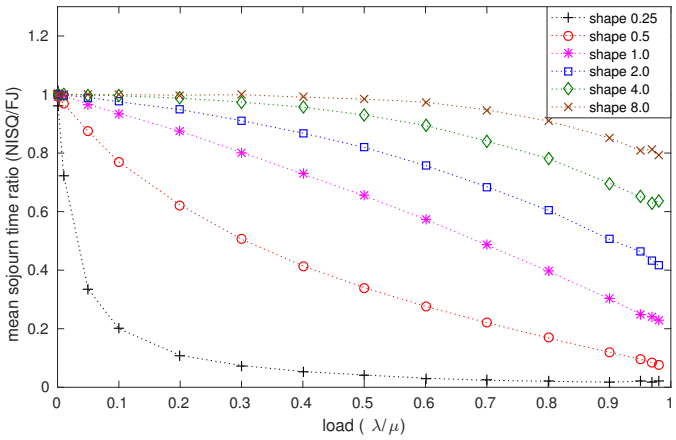


Fig. 2. Ratio of mean sojourn times for Non-Idling Single-Queue to Fork-Join at varying loads with exponential arrivals and Weibull service times with different shape parameters, $k = 16$. We see that the more heavy-tailed the service times are, the larger the difference between the two types of parallel queues.

A. FJ vs NISQ With Weibull Service Times

Fork-Join is commonly put forward as a model for parallel computation, and MapReduce in particular, but the Non-Idling Single-Queue model is a more accurate representation of how the default task manager works in Spark. We wanted to investigate how much their performance differs under different circumstances.

Under very low load, most jobs will arrive to an empty system, and we expect these two models to perform the same. Under heavy load we expect NISQ's average performance to be strictly better than FJ, but to what extent depends on the arrival and service processes. We ran our experiments with exponential arrivals and Weibull service times. By varying the shape parameter of a Weibull distribution we can produce service time distributions that range from heavy-tailed (shape parameter < 1) to bell curve-shaped (shape parameter > 1). This is compelling because some researchers have observed a heavy-tailed distribution in certain aspects of task service times, but in other situations we might expect the tasks within a job to have highly-correlated service times. Based on an idea in [28] we fix the mean service rate, and vary the shape parameter. We can solve for the Weibull scale parameter that gives us the desired mean rate.

Figure 2 shows the ratio of the mean sojourn times of the FJ and NISQ systems at loads (λ/μ) ranging from 0.0001 to 0.98 and Weibull shape parameters 0.25, 0.5, 1.0, 2.0, 4.0, and 8.0. All results are for $k = 16$ servers. The data were generated by the forkulator simulator. Each data point is the mean of 10^9 jobs sampled at intervals of 100 jobs.

As expected, the systems converge to the same mean at very low load. For smaller shape parameters (heavy-tailed service times) they diverge more quickly as the load increases. This is expected; for small shape parameters we are likely to see occasional tasks with abnormally long service times. In the NISQ system this temporarily reduces the number of available

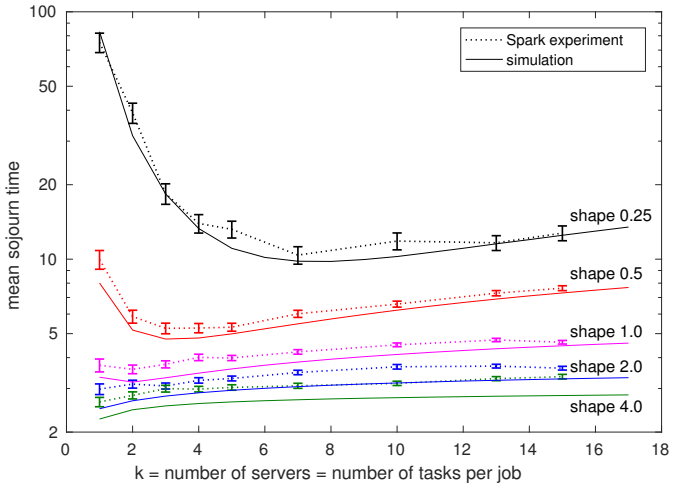
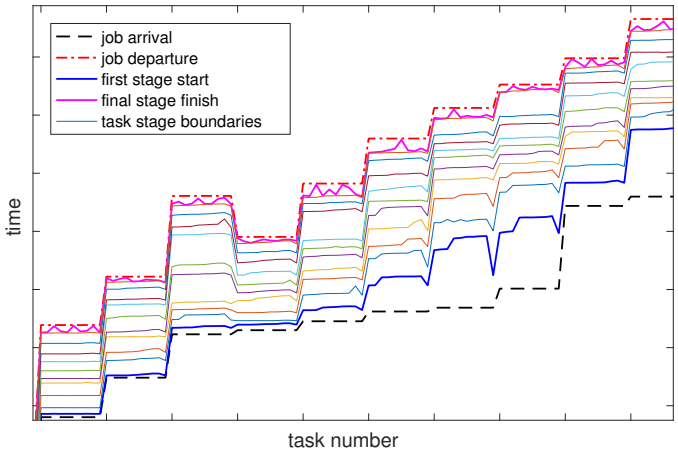
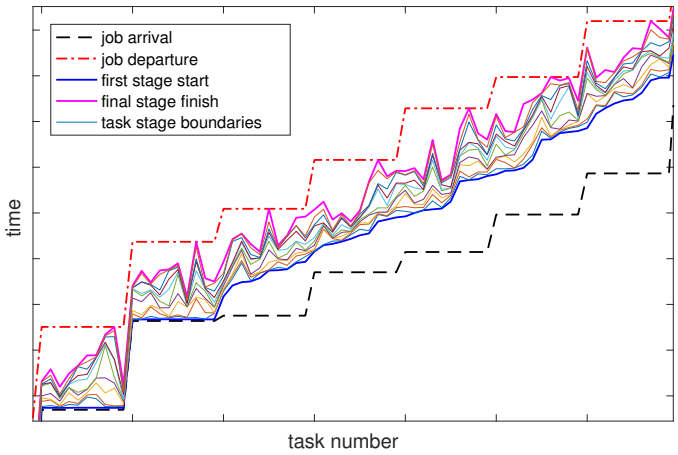


Fig. 3. Mean sojourn times for NISQ server with exponential arrivals ($\lambda = 0.7$) and Weibull service times for varying degrees of parallelism.



(a) Synchronization point between stages.



(b) Pipelined tasks.

Fig. 4. Example experiment paths for two modes of multi-staged experiments run on a Spark system. $k = 10$ tasks per job.

workers, but since tasks are not tied to a particular worker, the incoming jobs and tasks can "go around" the obstruction. In the FJ case tasks are tied to particular workers, so a single straggler task can hold up hundreds, or even thousands, of jobs behind it.

For shape parameters greater than 1.0 the differences between the systems are much less. This is also expected. In the extreme case, the work of each job would be equally divided between its tasks, and both systems would behave identically. From the figure we see that with shape parameter 8.0, as we approach 100% load, NISQ performs only about 20% better than FJ on average.

B. Scaling of NISQ with Weibull Service Times

Since we have not found a way to make Spark behave like a Fork-Join system, the comparison in the previous section was based only on simulation. However we can validate the NISQ part of the result by comparing to Spark experiments. We used MRSperf to run experiments with exponential job arrivals and Weibull service times with expected rates controlled as in section IV-A.

Figure 3 shows mean service times for simulation and the real Spark system for $k \in \{1, 2, 3, 4, 5, 7, 10, 13, 15\}$. In these experiments we again used time units of 0.1sec, so relative overhead is a non-trivial additive factor in the service times of the Spark experiment. The error bars are standard errors. Each Spark experiment ran 72,000 jobs, and sampled every 10 jobs. Note that the y-axis in the figure is logarithmic.

As expected the Spark experimental results have slightly larger sojourn times than the simulated results, but the difference appears to be additively consistent and diminishes slightly as k increases. For shape parameters greater than 1.0 the service times become closer and closer to deterministic, and this is reflected by a reduction in the mean sojourn times, and also in flatter scaling behavior. Interestingly, for shape parameters less than 1.0, the mean sojourn times decrease between $k = 1$ to $k = 2$, but for shape parameters greater than 1.0 we see a small trend in the opposite direction. We expect that the increase in sojourn time with k is due to the synchronization constraint, and is simply masked by the large load balancing benefit obtained in the cases with small shape parameter.

C. Multi-Stage Jobs and Pipelining

The basic mode of MRSperf runs a `map()` on an RDD with a controlled number of slices and then forces the execution with a reducing function. It is also possible to chain a series of `map()`s together, either with or without a synchronization point in between. When there is an operation requiring a shuffle between maps, Spark must wait for all tasks to complete before continuing. When there is no synchronization between maps, Spark will "pipeline" the maps, allowing the full chain of maps to run on a slice independently of the other slices, only synchronizing at the final stage. In this case the system is essentially equivalent to a single-stage system where the task service times have the distribution of the sum of the individual stage service times. For example, if the tasks have exponential service times, then the pipelined r-stage task would have an

Erlang-r service time distribution. The pipelined case is a lower bound for the performance of the staged case.

Example experiment paths for these two types of multi-stage experiments are shown in figure 4. In the first case, with no pipelining, whenever there is a synchronization between stages, a job cannot start the tasks from its next stage until all tasks of the previous stage finish. This means that tasks from incoming jobs will begin occupying the workers. We find that it is much more common for jobs to overtake each other in this case. We omit any further multistage results because of space limitations.

V. Conclusion and Future Work

We have presented forkulator and MRSperf, two tools for experimenting with models of parallel systems in simulation and on a real Spark cluster. Unlike other benchmarking suites, MRSperf is designed to run simple, statistically controlled experiments in a streaming context, to experiment with scaling behavior of MapReduce systems. We run single-core Spark workers in Docker containers, allowing us to run a large number of workers on a relatively small number of servers. We presented results comparing the popular Fork-Join model of parallel processing to the Non-Idling Single-Queue model which more accurately reflects the behavior of the Spark task manager. We also validated some simulation results against experimental data from a real cluster, and presented some data from multi-stage experiments in different modes.

In the future we would like to extend MRSperf and forkulator to experiment with jobs whose tasks have non-trivial shuffle and reduce phases and more substantial I/O requirements. We also would like to experiment with and model the distribution of data in an RDD during a computation. This is essential to understanding the scaling behavior of parallel systems, because the the impact of the synchronization constraint is mainly dependent on the intra-job task size variation.

References

[1] G. F. Riley and T. R. Henderson, *The ns-3 Network Simulator*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12331-3_2

[2] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. of USENIX OSDI*, Dec. 2002, pp. 255–270, http://www.emulab.org.

[3] R. J. Nowling and J. Vyas, "A domain-driven, generative data model for big pet store," in *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, Dec 2014, pp. 49–55.

[4] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF '15. New York, NY, USA: ACM, 2015, pp. 53:1–53:8. [Online]. Available: http://doi.acm.org/10.1145/2742854.2747283

[5] Apache Software Foundation, "Apache spark," https://spark.apache.org/, 2016.

[6] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "Using realistic simulation for performance analysis of mapreduce setups," in *LSAP '09*. New York, NY, USA: ACM, 2009, pp. 19–26. [Online]. Available: http://doi.acm.org/10.1145/1552272.1552278

[7] S. Hammoud, M. Li, Y. Liu, N. K. Alham, and Z. Liu, "Mrsim: A discrete event based mapreduce simulator," in *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, vol. 6, Aug 2010, pp. 2993–2997.

[8] B. Walker, "forkulator – fork-join queueing simulator," https://github.com/brentondwalker/forkulator, 2016.

[9] ——, "Mrsperf – statistically controllable arrivals and workloads for spark," https://github.com/brentondwalker/spark-arrivals, 2016.

[10] M. Fidler, B. D. Walker, and Y. Jiang, "Non-asymptotic delay bounds for multi-server systems with synchronization constraints," *CoRR*, vol. abs/1610.06309, 2016. [Online]. Available: http://arxiv.org/abs/1610.06309

[11] C. Willits, "Nested fork-join queuing networks and teir application to mobility airfield operations analysis," Ph.D. dissertation, Air Force Institute of Technology, 1997.

[12] L. Flatto and S. Hahn, "Two parallel queues created by arrivals with two demands. I," *SIAM Journal on Applied Mathematics*, vol. 44, no. 5, pp. 1041–1053, Oct. 1984.

[13] R. D. Nelson and A. N. Tantawi, "Approximate analysis of fork/join synchronization in parallel queues." *IEEE Trans. Computers*, vol. 37, no. 6, pp. 739–743, 1988. [Online]. Available: http://dblp.uni-trier.de/db/journals/tc/tc37.html#NelsonT88

[14] I. N. de Recherche en Informatique et en Automatique, F. Baccelli, and A. Makowski, *Simple Computable Bounds for the Fork-join Queue*, ser. Rapports de recherche. Institut National de Recherche en Informatique et en Automatique, 1985.

[15] S. Varma and A. M. Makowski, "Interpolation approximations for symmetric fork-join queues," *Perform. Eval.*, vol. 20, no. 1-3, pp. 245–265, May 1994. [Online]. Available: http://dx.doi.org/10.1016/0166-5316(94)90016-7

[16] E. Varki, "Mean value technique for closed fork-join networks," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, no. 1, pp. 103–112, May 1999. [Online]. Available: http://doi.acm.org/10.1145/301464.301484

[17] D.-R. Liang and S. K. Tripathi, "On performance prediction of parallel computations with precedent constraints," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 5, pp. 491–508, May 2000.

[18] S.-S. Ko and R. F. Serfozo, "Response times in m/m/s fork-join networks," *Advances in Applied Probability*, pp. 854–871, 2004.

[19] R. Osman and P. G. Harrison, "Approximating closed fork-join queueing networks using product-form stochastic petri-nets," *J. Syst. Softw.*, vol. 110, no. C, pp. 264–278, Dec. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2015.08.036

[20] G. Kesidis, Y. Shan, B. Urgaonkar, and J. Liebeherr, "Network calculus for parallel processing," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 2, pp. 48–50, Sep. 2015. [Online]. Available: http://doi.acm.org/10.1145/2825236.2825256

[21] A. Rizk, F. Poloczek, and F. Ciucu, "Stochastic bounds in fork—join queueing systems under full and partial mapping," *Queueing Syst. Theory Appl.*, vol. 83, no. 3-4, pp. 261–291, Aug. 2016. [Online]. Available: http://dx.doi.org/10.1007/s11134-016-9486-x

[22] M. Fidler and Y. Jiang, "Non-asymptotic delay bounds for (k, l) fork-join systems and multi-stage fork-join networks," in *INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, 2016, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/INFOCOM.2016.7524362

[23] R. Nelson, D. Towsley, and A. N. Tantawi, "Performance analysis of parallel processing systems," in *SIGMETRICS '87*. New York, NY, USA: ACM, 1987, pp. 93–94. [Online]. Available: http://doi.acm.org/10.1145/29903.29916

[24] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[25] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, ser. PDP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 233–240. [Online]. Available: http://dx.doi.org/10.1109/PDP.2013.41

[26] "FreeBSD Handbook," https://www.freebsd.org/doc/handbook/book.html, Accessed 2017.

[27] J.-Y. Le Boudec, *Performance evaluation of computer and communication systems*, ser. Computer and communication sciences. Lausanne: EPFL Press London, 2010. [Online]. Available: http://opac.inria.fr/record=b1131863

[28] M. Dell'Amico, D. Carra, M. Pastorelli, and P. Michiardi, "Revisiting size-based scheduling with estimated job sizes," in *MASCOTS 2014*, Paris, FRANCE, 03 2014. [Online]. Available: http://www.eurecom.fr/publication/4268