The Tiny-Tasks Granularity Trade-Off: Balancing Overhead Versus Performance in Parallel Systems

Stefan Bora, Brenton Walker, and Markus Fidler, *Senior Member, IEEE*

***Abstract*—Models of parallel processing systems typically assume that one has l workers and jobs are split into an equal number of $k = l$ tasks. Splitting jobs into $k > l$ smaller tasks, i.e. using "tiny tasks", can yield performance and stability improvements because it reduces the variance in the amount of work assigned to each worker, but as k increases, the overhead involved in scheduling and managing the tasks begins to overtake the performance benefit. We perform extensive experiments on the effects of task granularity on an Apache Spark cluster, and based on these, develop a four-parameter model for task and job overhead that, in simulation, produces sojourn time distributions that match those of the real system. We also present analytical results which illustrate how using tiny tasks improves the stability region of split-merge systems, and analytical bounds on the sojourn and waiting time distributions of both split-merge and single-queue fork-join systems with tiny tasks. Finally we combine the overhead model with the analytical models to produce an analytical approximation to the sojourn and waiting time distributions of systems with tiny tasks which include overhead. We also perform analogous tiny-tasks experiments on a hybrid multi-processor shared memory system based on MPI and OpenMP which has no load-balancing between nodes. Though no longer strict analytical bounds, our analytical approximations with overhead match both the Spark and MPI/OpenMP experimental results very well.**

***Index Terms*—Network calculus, parallel processing, performance bounds, processing overhead, Spark, synchronization constraints, task granularity, tiny-tasks.**

I. Introduction

PARALLEL processing systems improve performance by dividing large jobs into smaller tasks, and distributing those tasks to a cluster of many workers. To a first approximation, the total amount of processing time does not change, but the amount of time a user must wait for the result can be reduced by orders of magnitude. Because of the distributed computation, the size of the data set that can be operated on, and held in RAM, is correspondingly increased.

The first impulse, both in modeling such systems and in practice, is to divide each job into tasks so that it fits evenly on the available workers. If k is the number of tasks, and l is the number of workers, this means taking $k = l$. Using a finer granularity, taking $k > l$, so-called "tiny tasks", actually can have a great and positive impact on system performance. This has been noted by practitioners [2], [3], [4], but so far only [1], which this paper is an extension of, provides analytical results relating task granularity to parallel system performance.

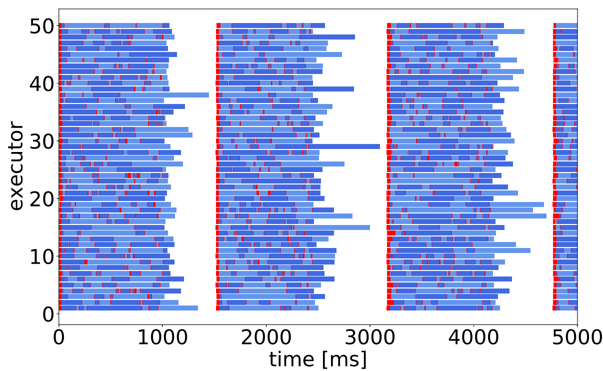


Fig. 1. Five second sample of 4 jobs containing 400 tasks per job running on an Apache Spark cluster with 50 executors. Execution times of successive tasks are represented by alternating dark and light blue. Anything other than task execution, i.e. overhead, is marked in red.

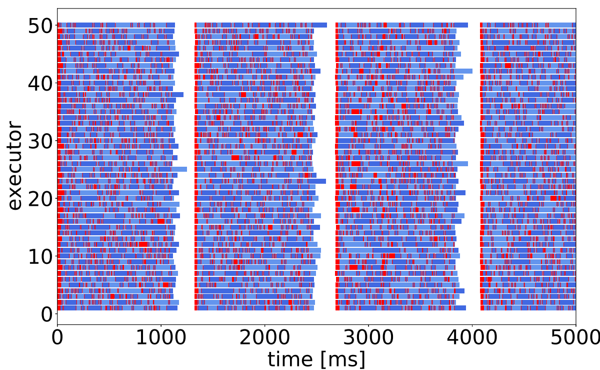


Fig. 2. Five second sample of 4 jobs, of the same mean size, containing 1500 tasks per job running on the same cluster.

Figs. 1 and 2 show diagrams of the activity of 50 executors (workers) in a standalone Apache Spark cluster servicing a sequence of four jobs, divided into tasks with different levels of granularity. In this case the jobs are processed as if they are being submitted from a single-threaded driver program, so each job does not begin until the previous one departs. In Fig. 1 the jobs are divided into 400 tasks, and in Fig. 2 they are divided into

Manuscript received 28 February 2022; revised 28 September 2022; accepted 29 November 2022. Date of publication 4 January 2023; date of current version 10 March 2023. This work was supported in part by the German Research Council (DFG) under Grant VaMoS FI 1236/7-1. Recommended for acceptance by B. DiMartino. *(Corresponding author: Brenton Walker.)*

The authors are with the Institute of Communications Technology, Leibniz Universität Hannover, 30167 Hannover, Germany (e-mail: stefan.bora@ikt.uni-hannover.de; brenton.walker@ikt.uni-hannover.de; markus.fidler@ikt.uni-hannover.de).

Digital Object Identifier 10.1109/TPDS.2022.3233712

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

1500 tasks. It is immediately apparent that more executors spend much more time idling in the case with coarser task division. Further, in the case with finer task division, the fourth job is almost complete after 5000 ms, whereas in the coarser case, the fourth job is just starting service.

The primary reason this happens is that when smaller tasks are used, the variance in the amount of the work assigned to each worker decreases. On the other hand, in real systems, there will always be some trade-off limiting the performance gains. As tasks are made smaller and smaller, at some point the overhead of scheduling and gathering results from tasks will dominate the operation.

This paper explores the issue of task granularity in three domains. The first is experimental. We perform experiments with a real Apache Spark system and a hybrid MPI/OpenMP cluster to understand the sources of overhead, and the practical limits of the performance gains that can be realized from refined task granularity. Related measurements have been done before, either to develop practical guidance for improving cluster performance [2], [3], or in evaluating distributed schedulers that could support larger degrees of parallelism [5]. In [6] the effects of task granularity, and flat versus recursive task spawning, on performance are investigated for several shared-memory task-centric parallel systems. In our case we focus on statistically principled experiments that use tasks with service times drawn from controlled distributions. The results of these experiments also serve to validate the other points.

Second we develop a model for system overhead based on our measurements from the Apache Spark system, and use simulation to study the effects of task tinyfication, and model the effects of different types of scheduling overhead.

Finally we present queuing theoretic results. In analyzing the split-merge system with tiny tasks, we will formulate the tiny tasks model as a direct refinement of the "big tasks" model. We will derive expressions for the stability region in both cases. In the limit as $k \to \infty$, the stability region approaches one. We then derive statistical performance bounds for both the split-merge and fork-join models with tiny tasks, and show how their performance improves over the equivalent big tasks model. As $k \to \infty$, the performance approaches that of the **ideal job partition**. This is achieved when the jobs are partitioned into $k = l$ equally sized tasks.

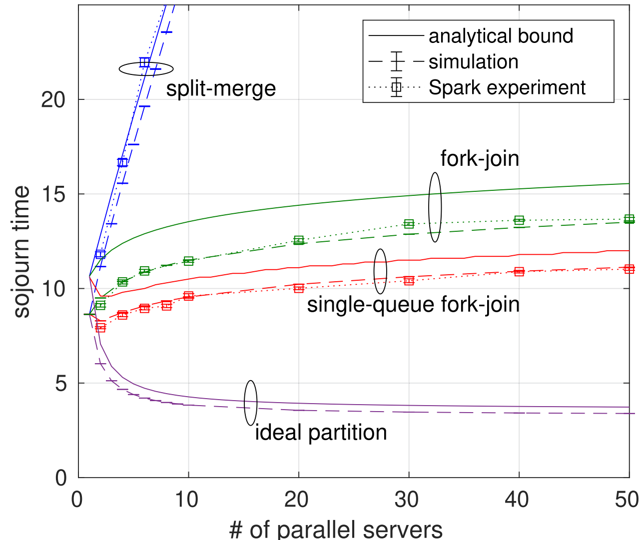


Fig. 3. Scaling of sojourn time quantiles of the conventional ($k = l$) split-merge, fork-join, and single-queue fork-join models for varying degrees of parallelism. Exponential arrivals ($\lambda = 0.2$) and task service times ($\mu = 1.0$). We also include results for the ideal job partition, where a job is partitioned into l equally sized tasks.

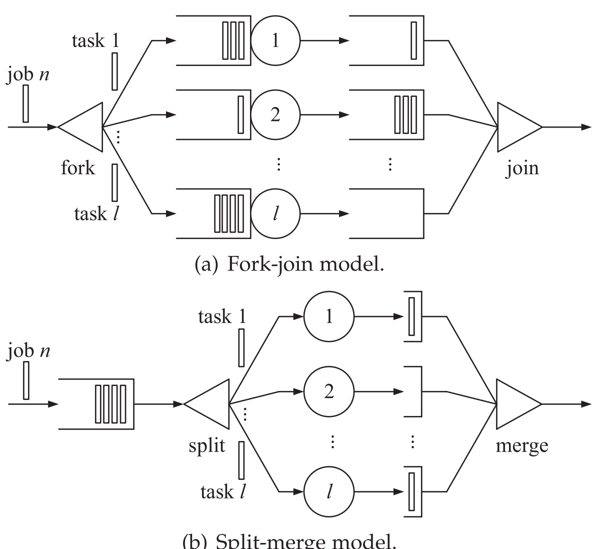


Fig. 4. Models of parallel systems.

A. Systems, Models, and Stability Regions

Our experiments with real systems are focused on Apache Spark, since it can be easily deployed on commodity hardware and serves as a good representative of many models of parallel processing with independent tasks. Spark is a popular parallel processing engine that implements a map-reduce API [7], [8]. Other comparable map-reduce engines include Hadoop MapReduce [9] and Flink [10]. Fig. 3 shows that, depending on the constraints put on the system, a Spark program may exhibit the scaling behavior of split-merge, fork-join, or single-queue fork-join, three different models of parallel systems with quite different scaling behavior. We will summarize those models and some prior analytical work in this section.

Across all of the models we will discuss, there are some common random processes: the arrival, departure, and task service process. Let $A(n)$ for $n \geq 1$ denote the **arrival time** of job n, and $D(n)$ the **departure time**. One key performance statistic we will focus on is the **job sojourn time** (often called "response time"), $T(n) = D(n) - A(n)$. Given l parallel servers and k tasks per job, let $Q_i(n)$ denote the **task service time** of task $i \in [1, k]$ of job $n \geq 1$. The **workload** of job n, $L(n) = \sum_{i=1}^{k} Q_i(n)$, is defined to be the total of the service required by all of its tasks. The **job service time** $\Delta(n)$ is the total time a job spends in service. That is, the time between when its first task begins service and when all of its tasks finish service. This is often referred to as the "makespan" in operations research. Note that

for the parallel models, $L(n)$ and $\Delta(n)$ are not necessarily equal. The table below summarizes the most important variables and symbols that we will use throughout this paper.

k	Number of tasks per job
l	Number of workers
$A(n)$	Arrival time of job n
$D(n)$	Departure time of job n
$T(n)$	Sojourn time of job n, also called the response time, equal to $D(n) - A(n)$
$Q_i(n)$	Service time of task i of job n
$E_i(n)$	Task execution time of task i of job n. The portion of $Q_i(n)$ spent on the intended task.
$O_i(n)$	Task service overhead of task i of job n. The portion of $Q_i(n)$ due to overhead.
$L(n)$	Workload of job n. The sum of the total service required by all of its tasks.
$\Delta(n)$	Job service time of job n, also referred to as the makespan. Time from when the first task starts service until the last task finishes.

Fig. 4(a) shows a schematic of the **fork-join** model. Jobs enter the system and are divided into k tasks (fork) that are assigned one by one to $l = k$ servers. Once all k tasks of a job are serviced, the job leaves the system (join). The difficulty in analyzing fork-join systems arises from the synchronization constraint of the join operation, and an exact solution is only known for the M | M|1 case with $k = l = 2$ [11], [12]. For broader classes of systems, a variety of approximation techniques have been used [13], [14], [15], [16], [17], [18]. More recently several researchers have used stochastic network calculus to derive performance bounds [1], [19], [20], [21], [22]. Many examples of the fork-join pattern being used in practice are given in [23].

A schematic of the split-merge model is shown in Fig. 4(b). The **split-merge** model, also referred to as "blocking fork-join" in [20], has the additional synchronization constraint that the system is blocked until the current job departs. Parallel systems that behave like split-merge arise easily in practice. For example, any Spark program with a single-threaded driver program, or a single user submitting jobs from a data analysis notebook. Related, but not exactly the same, many machine learning algorithms require that all tasks in a job start and depart simultaneously to facilitate communication and data transfers between the workers [24], [25], [26].

The analysis of the split-merge model turns out to be much simpler because it behaves like a single-server system with service times given by the service time of the largest task of each job [13], [27], [28]. The problem with the conventional ($k = l$) split-merge model is that it becomes unstable for utilizations well below one, and it becomes unstable more quickly as the degree of parallelism increases, as seen in Fig. 3. This has led some researchers to discount the model as impractical [20].

A third model arises in practice. When jobs are submitted by a multi-threaded driver program, MapReduce engines such as Apache Spark and Hadoop MapReduce behave like a **single-queue fork-join** system, where all tasks are held in a single FIFO queue and assigned to servers as they become available [29]. Compared to the fork-join model, where tasks are bound to particular servers and a large task can block tasks of subsequent jobs, in the single-queue fork-join model small jobs can overtake jobs with large straggler tasks. Mean sojourn times for such systems are derived in [30], and bounds on the sojourn time are derived using network calculus in [21].

Fig. 3 shows how job sojourn time scales with the number of servers for these three models in the case with $k = l$ and exponential inter-arrival and task service times. The plot shows performance bounds derived using network calculus in [20] and [21], simulation results, and experimental results from an Apache Spark cluster, and demonstrates that a Spark system may behave like any of these three parallel models, depending on how it is configured and how the driver program behaves. For comparison, the plot includes the equivalent sojourn time statistics for the ideal job partition. Both fork-join systems show a logarithmic increase in sojourn time as the degree of parallelism increases because of the synchronization constraint [20], [31]. The performance of the split-merge system appears catastrophic by comparison.

Other popular platforms for parallel computation include task-centric shared-memory systems such as Cilk [32], OpenMP [33], and Threaded Building Blocks (TBB) [34]. These are especially useful in applications that require coordination and data exchange between tasks, as all threads in the program can access shared data structures, but their scaling is limited when used on commodity hardware. Another popular tool is the Message Passing Interface (MPI) [35] which can be used to coordinate and exchange data between processes started across a cluster of separate machines. These two types of platforms can be combined to build hybrid parallel systems that offer the performance benefit of the shared-memory system locally, but can scale to a cluster of commodity machines [36], [37], [38], [39]. The nature of the system overhead and load balancing mechanisms do not quite conform to the conventional parallel models we presented in this section. In Section VII we will discuss in more detail how the systems differ and report on additional experiments on a hybrid MPI/OpenMP cluster and apply our tiny-tasks analytical results to it.

B. Introduction to Tiny Tasks

It is no surprise that parallel systems users have devised methods to increase the performance of their systems, even in the split-merge case. The simplest of these is to partition jobs into a larger number of tasks than there are servers, $k > l$. A common guideline for Spark systems is that the number of tasks should be about three times the number of servers, i.e., $k \approx 3l$ [2], or optimized through trial and error [3]. In shared-memory systems task granularity is often discussed in terms of a target task duration [4], [6]. Some researchers have proposed even more extreme task granularity, $k \gg l$, coining the term "tiny tasks" [40].

We would like to formalize our conception of the tiny tasks regime. We assume a system with l servers and jobs partitioned into $k \geq l$ tasks, where k may be orders of magnitude larger than l. We define $\varkappa = k/l$ to be the **factor of tinyfication** (i.e., $\varkappa$ is the average number of tasks from each job served by each of the

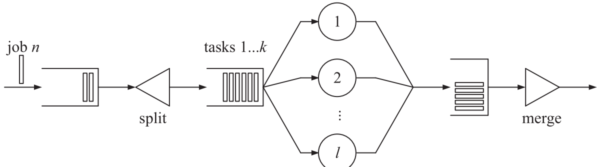


Fig. 5. Split-merge model with tiny tasks.

servers). The least granular case, where $\varkappa = 1$ and $l = k$, gives us the conventional parallel models. We will refer to the tasks in this case as "big tasks". When $\varkappa > 1$ we refer to them as "tiny tasks".

In the split-merge and fork-join cases, there may be some ambiguity as to how the tiny tasks variants of these models work. In both cases we model the tiny tasks system to behave in the same way that Spark would behave when given $k > l$ tasks per job.

A schematic of the split-merge tiny tasks model is shown in Fig. 5. Jobs are stored in a job queue, and if there is no job in service, the head-of-line job is partitioned into k tasks (split) which are then stored in the task queue. Since all servers are idle at the start of a job, the first l tasks start service immediately. Whenever a server finishes a task, it fetches the head-of-line task from the task queue. When all k tasks have finished service, the job leaves the system (merge) and the next job, if any, is partitioned and starts service.

In the case of fork-join, using tiny tasks only makes sense in the context of a single-queue model. In the standard fork-join model, where tasks are bound to specific servers on arrival, tiny tasks would make no difference. Therefore, throughout this paper, when we refer to fork-join with tiny tasks, it should be understood that we are referring to the single-queue fork-join model.

II. Tiny Tasks on Apache Spark

Apache Spark is a popular parallelized data analytic platform implementing the map-reduce paradigm, and therefore well suited to evaluate the performance of tiny tasks on a real cluster system. Some papers find that, on real systems, the scheduling and bookkeeping overhead required by tiny tasks outweighs the advantages [5] or makes it impracticable to run on platforms with a centralized scheduler like Apache Spark [41]. In this section we will report on our extensive measurements using varying number of tasks per job, investigate the sources and behavior of overhead, and present an overhead model suitable for use in simulation and analytical models, that produces sojourn time distributions matching real experiments.

A. Execution Model of Apache Spark

We experiment with Spark in stand-alone mode with the default scheduler [42]. Fig. 6 shows a schematic of the Apache Spark components. A cluster consists of numerous worker nodes which offer their resources to the cluster manager. The cluster manager allocates the resources to an application running a **driver** program. Depending on the requested resources, each worker node can host one or more **executors** which connect to the SparkContext in the driver program.

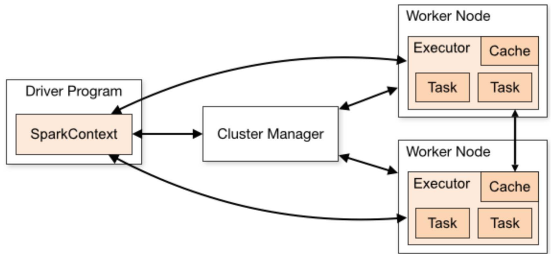


Fig. 6. Schematic of the Spark model [42].

The SparkContext in the application maintains a queue of jobs waiting to be scheduled. Every Spark job operates on a Resilient Distributed Dataset (RDD). An RDD is a memory representation abstraction [43] consisting of multiple partitions which can be distributed throughout the cluster. There are two types of operations that can be executed on an RDD. **Transformations** like `map` are used to create new RDDs from existing ones in a lazy operation. **Actions** execute the calculations on the cluster and block the thread until the result is returned. Spark internally divides jobs into one or more stages consisting of tasks which can run independently from each other. The resulting execution plan is represented as a Directed Acyclic Graph (DAG). A stage can include multiple transformations and ends with an action. The most common reason for multiple stages are operations that cause a shuffle, such as `reduceByKey`, which lead to repartitioning of the RDD. The DAG ensures that when there is a sequential dependency, tasks of the next stage can only be executed after the previous stage has finished.

B. Overhead

In any cluster with a central scheduler, like Apache Spark, there is overhead which cannot be avoided. For example a task must be scheduled for execution, and its code and data need to be serialized and sent to an executor. Depending on the number of tasks and executors handled by the scheduler, this can result in substantial overhead relative to the actual task execution time, especially with extremely small tasks with millisecond run times.

Some of the scheduling delays depend on the speed of the worker nodes and the network. The serialization technique used is also of crucial importance because of the associated processing and transmission time [44]. After receiving a task, the executor must deserialize it before the actual workload can run. Depending on the workload, additional data may have to be fetched over the network or loaded from disk. After the executor services the task, the result has to be serialized and sent to the driver, and written out to a disk or kept in memory. After a job finishes, the scheduler has to collect the results of its tasks and return a result that depends on the executed action.

There are two classes of overhead we will need to consider. **Task-service overhead** is attributable to individual tasks and blocks an executor core from servicing the next task. This type

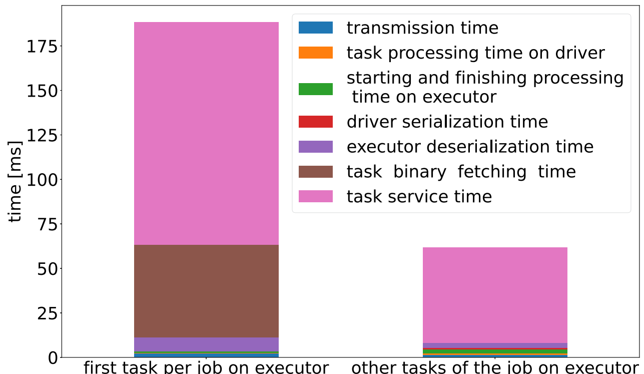


Fig. 7. Example of the Spark task duration with 1500 tasks/job with the remaining parameters as in Fig. 13. The left bar shows a sample of the first task of a job which ran on an executor whereas the right bar shows an example of a task which is not the first task. The measurement of these examples will be discussed in Section II.C.

of overhead is fairly well measured and represented in the Spark UI and logs. **Pre-departure overhead** is the additional delay after all tasks of a job complete, but before a job can depart. It does not necessarily block the tasks of subsequent jobs from being serviced. Since it is not attributable to individual tasks during their run time, it is not well represented in the Spark UI and task metrics. We deduce its statistical properties based on the overhead model required to fit the experimental sojourn time distributions on real Spark systems. This will be discussed in more detail in Section II.F.

Fig. 7 shows a breakdown of the execution and overhead times of two successive tasks running on a Spark cluster. They can be categorized as follows:

Transmission time: The time used to send the task information to the executor and to notify the driver about a finished task. When the task result size is below a hard-coded threshold, it is included in this transmission as well. If the result is larger, a reference is included and the result is fetched asynchronously later.

Task processing time on driver: Time between fetching the task out of the queue and sending it to an executor, and the time after receiving the message of a finished task and marking it as finished on the driver.

Starting and finishing processing time on executor: Time required for housekeeping on the executors. Includes parsing the task message from the driver, decoding the task description and putting the task in the execution queue before running the task. After the task execution is done the executor marks the task as finished, removes it from the list of running tasks and sends the task status including the task result, depending on it's size and the configuration, directly or using the block manager, to the driver.

Driver serialization time: The serialization time of a task on the driver. In Spark this is done in two parts. First the task itself is serialized. The main content is the task binary, which is a broadcast variable, the RDD (in serialized form only the identifiers), and the accompanying metadata such as stage and task ID. In a second step the task description is serialized. This object includes some redundant data such as the task and partition IDs, but also additional data such as the executor ID where the task will run, files and jars which needed to be added, some resources, and the serialized task.

Executor deserialization time: The time needed to deserialize the task on the executor. The standard Spark metrics include in this the time to fetch the task binary, even if this happens remotely. In our statistics we separate this into the item below.

Task binary fetching time: Time required to fetch the serialized task function and RDD stored in a broadcast variable. Each executor only has to fetch this remotely once. After that, the local copy is used.

Task execution time: The time the executor spends actually executing the task.

The task duration is the sum of the components above. As a sanity check, this sum should equal the time between the scheduling of the task until the driver receives its result. We consider everything in the list above, except task execution time, to be overhead.

Next to the listed overhead there are additional types of overhead not considered in our experiments. The time to load RDD data from the disk or over the network is beyond the scope of our investigations. Also the time to fetch additional jar files is not considered because it appears only once per application.

For clarity in the subsequent discussion, we need to introduce some terminology and notation for overhead and how it relates to the task service time process. Recall that the task service time, $Q_i(n)$, is the time between when the scheduler takes up task i of job n and when the worker becomes available to service the next task. In real systems we consider these service times as comprising two components.

$$Q_i(n) = E_i(n) + O_i(n) \tag{1}$$

where $E_i(n)$ is the **task execution time** of task i of job n, and $O_i(n)$ is the **task overhead**. When doing experiments on a Spark cluster, we can control the execution times of the tasks and draw them from known distributions, but the tasks' overhead is simply a property of the system and we need to measure and model it.

C. Experiment Environment

For our Spark experiments we allocated 13 nodes from our institute's Emulab testbed [45], connected via a 1Gbit/s network. To run experiments with more executors than physical nodes, we ran several single-core executors inside Docker containers on each worker node. This way the executors cannot share JVM memory and behave independently. We used 12 nodes as worker nodes, and 1 node as the master and driver node. The worker nodes also hosted a Hadoop Distributed File System (HDFS) to store the experiment logs.

We used Apache Spark version 3.0.0 preview2, modified slightly to log more details about the processing of tasks and jobs [46]. Specifically we added a Spark listener which stores more detailed task metrics than what is available by default. To run the experiments we extended SparkBench [47], a tool created to benchmark Spark clusters. We implemented classes which make it possible to run workloads in the manner of a split-merge or single-queue fork-join system. We also added

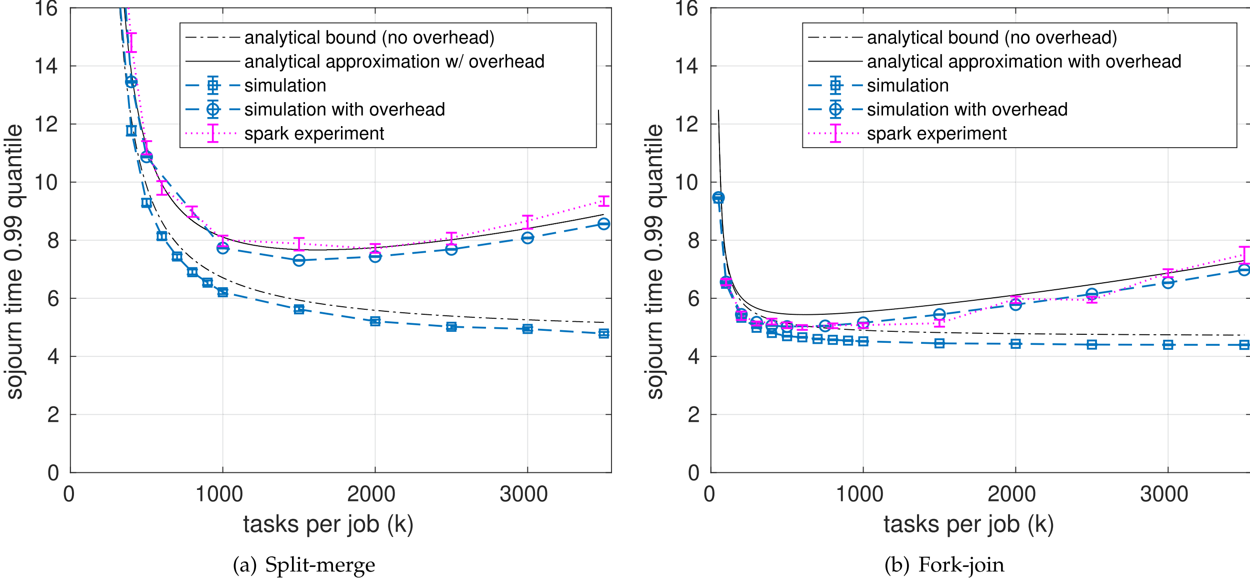


Fig. 8. Comparison of the sojourn time bounds of the single-queue fork-join and split-merge models with $l = 50$ servers and k tiny tasks. Jobs have exponential inter-arrival times with parameter $\lambda = 0.5\text{s}^{-1}$ and are composed of k tasks with exponential service times with parameter $\mu = \frac{k}{l}\,\text{s}^{-1}$. The analytical bound and analytical approximation with overhead will be explained in Sections IV, V, and VI.

workloads to create jobs composed of tasks with service times sampled from known distributions, in the experiments here an exponential service time distribution. The workloads used in our experiments simply take a sample from the service time distribution, sit in a loop until its time expires, and finally returns some internal task data.

Our environment makes it possible to perform experiments on a Spark cluster that match as closely as possible the statistical assumptions about task service time distributions made later in developing an analytical model for tiny tasks. This is essential when validating our analytical results through experiments and simulation.

D. Simulation

We ran additional experiments using `forkulator`, an event-driven simulator for parallel systems [48]. Because it is not constrained to running in real time, using a simulator allows us to generate orders of magnitude more data points than the Spark experiments. It also lets us run idealized experiments both with and without the scheduling and processing overhead inherent to real experiments, and allows us to experiment with the effects of different types of overhead and overhead distributions to see which most accurately matches the behavior of real systems.

E. Evaluation

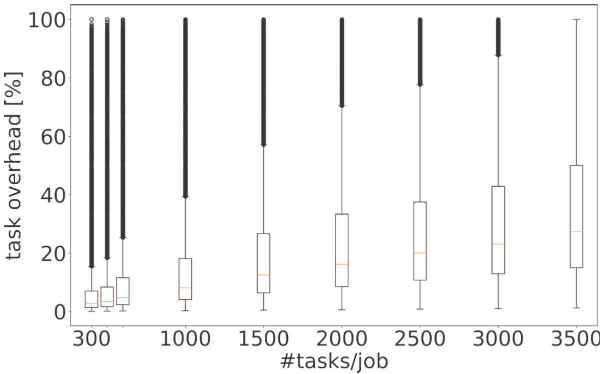
We configured the Spark experiments with $l = 50$ workers, independent and identically distributed (iid) exponential inter-arrival times with parameter $\lambda = 0.5\,\text{s}^{-1}$. Initially we create jobs with no tinyfication, $k = l = 50$ tasks with iid exponential task service times with mean 1000 ms. With tinyfication we take $k > l$ tasks per job with iid task service times drawn from an exponential distribution with parameter $\mu = \frac{k}{l}\,\text{sec}^{-1}$. This means that as the number of tasks per job, k, increases, the mean service time of the tasks will correspondingly decrease, keeping the expected job workload, $\text{E}[L(n)] = k/\mu = l\,\text{s}$, constant. For each configuration we submitted at least 30,000 jobs. The jobs were run in batches of 1000 to ensure the accumulated measurements of each batch could be stored without interfering with the experiment.

The performance benefit of using tiny tasks in both split-merge and single-queue fork-join systems can be seen in Fig. 8 which plots the $\varepsilon = 0.99$ quantile of job sojourn time. For the fork-join system in Fig. 8(b), a 12-fold tinyfication (going from $k = 50$ to $k = 600$ tasks per job), decreases the sojourn time quantile by 46.7%. The benefit is most dramatic at smaller tinyfication factors. Going from $\varkappa = 1$ to $\varkappa = 2$ alone (from $k = 50$ to $k = 100$), reduces the sojourn time quantile by 30.4%.

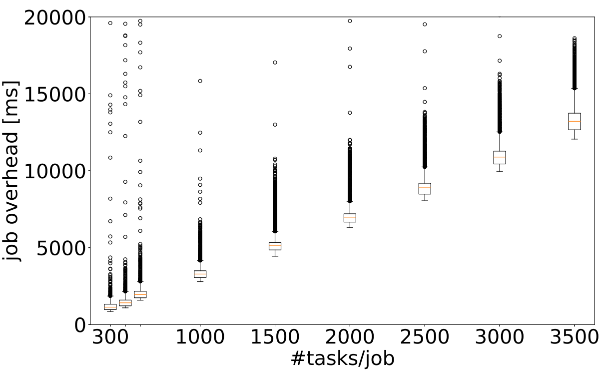
The results for the split-merge system are shown in Fig. 8(a). For the arrival and service parameters we used, the split-merge system is unstable in the big tasks case ($\varkappa = 1$). This is expected based on [20, Eq. 21] and the stability analysis that we will present in Section IV.B. A four-fold tinyfication, splitting the jobs into $k = 200$ tiny tasks, stabilizes the system. More extreme tinyfication reduces the sojourn times further.

The limits of tiny tasks start to become apparent at higher tinyfication factors, $\varkappa$. In this experiment, beyond about $k = 1000$ tasks per job ($\varkappa = 20$), the gains start to level off. With an increasing number of smaller and smaller tasks the sojourn time quantiles begin to increase.

To investigate this in more detail we look at the fraction of the total task service time that is due to overhead at different tinyfication factors. Fig. 9(a) shows a box plot of $O_i(n)/Q_i(n)$ versus k for the split-merge execution mode. Both the median and mean of the overhead fraction grow nearly linearly with increasing k. We see that there are outlier tasks which exhibit close to 0% and others close to 100% overhead. This is mainly due to the random task service time distribution which can



(a) Fraction of task service time due to overhead



(b) Total task overhead per job.

Fig. 9. Overhead on the example of the fork-join experiments with the configuration of Fig. 8.

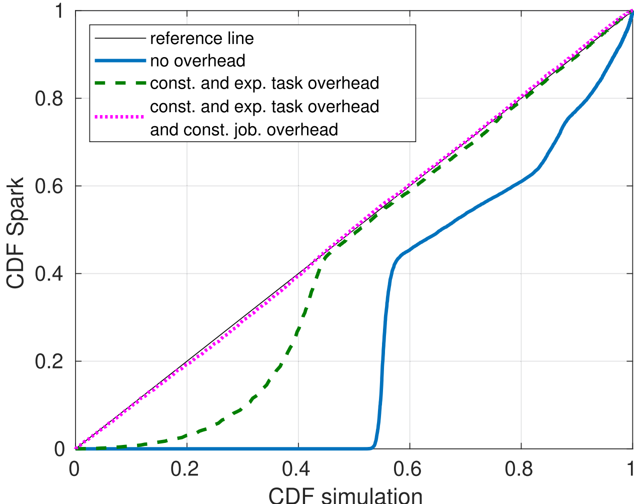


Fig. 10. Comparison of the single-queue fork-join sojourn time from Spark experiments and simulations. The configuration is like in Fig. 8 for the fork-join model with $k = 2500$ tasks per job. The task time overhead is added during the simulation and affects the service time. The job overhead is added after the simulation and can be interpreted as an asynchronous overhead of the Spark scheduler while the next job can already run.

produce large or very small service times. Fig. 9(b) shows a boxplot of the total overhead per job, $O(n) = \sum_{i=1}^{k} O_i(n)$, for a range of k values. The median shows a nearly linear increase with k. This growth in total overhead partially explains the growth in sojourn times observed in Fig. 8. The form and rate of this increase will depend on understanding the distribution of the overhead, and its effect on job waiting times, which we will explore throughout the rest of this paper.

F. Overhead Distribution

We observe that the task overhead has one or more random components, and we would like to characterize its distribution so it can be modeled and replicated in simulation. We will evaluate the accuracy of our overhead model using probability-probability plots (P-P plots) of the resulting job sojourn time distributions from simulation against those from Spark experiments.

Fig. 10 shows the P-P plot of the job sojourn times of a simulated single-queue fork-join system with exponential task execution times and $k = 2500$ tasks per job, both with and without simulated overhead, against those of the equivalent Spark experiments. The blue line is the P-P plot when no overhead is included in the simulation. It shows that the cumulative distribution function (CDF) of Spark sojourn times remains at or close to zero while at least half of the simulated jobs depart. After that the Spark CDF catches up gradually. A step-like pattern in a P-P plot means that the support of one of the distributions is offset by some amount. Based on this, and the linear growth of the job overhead in Fig. 9(b), we added a constant amount of overhead, c^{ts}_{task}, to every task in the simulation. We also observe a scattering of extreme outliers in the overhead. We model them by adding an additional exponential component to the task overhead with mean 0.5 ms ($\mu^{ts}_{task} = 2000\ \text{s}^{-1}$). This gives us a two-parameter model for task-service overhead.

$$O_i(n) \sim c^{ts}_{task} + \text{Exp}\left(\mu^{ts}_{task}\right) \tag{2}$$

The resulting P-P is plotted with a dashed green line in Fig. 10. The two distributions fit each other much better starting at around 50% of the samples. This can be interpreted to mean that the minimal sojourn times in the Spark experiment are higher compared to the simulation. We hypothesize that this is because of the processing time on the driver application, and simulated it by adding some amount, c^{pd}_{job}, of pre-departure overhead to every job. The amount of pre-departure overhead needed turns out to grow linearly with the number of tasks, with rate c^{pd}_{task}. We model this overhead with a deterministic linear function added to the simulated departure time.

$$D^o(n) = D(n) + c^{pd}_{job} + k \cdot c^{pd}_{task}. \tag{3}$$

In the fork-join case, this did not require modifying the simulation, since the pre-departure overhead is simply added to the simulated sojourn times. It does not affect the processing of subsequent jobs or tasks. In the split-merge case, delaying the departure of the job does block the tasks of subsequent jobs,

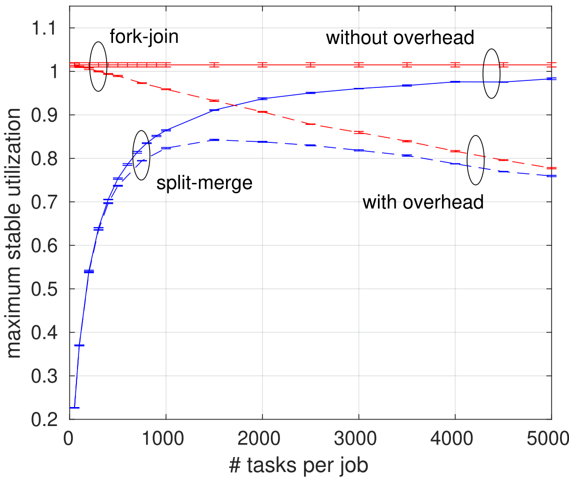


Fig. 11. The stability regions of split-merge and fork-join simulated with and without task and job overhead, with $l = 50$ parallel workers.

and therefore did require modifications to the corresponding scheduler class in the simulator.

task-service	c_{task}^{ts}	= 2.6 ms
	μ_{task}^{ts}	= 2000 s^{-1}
pre-departure	c_{job}^{pd}	= 20 ms
	c_{task}^{pd}	= 7.4×10^{-3} ms

The parameter values determined in these experiments are listed in the table above. The magenta dotted line in Fig. 10 shows the P-P plot of the simulation with both constant and exponential task overhead and linear pre-departure overhead against the Spark result. In comparison to the black reference line, simulation with these four overhead components, matches the distribution of the real Spark experiment acceptably well. Simulating an overhead distribution for the first task of each job on each executor, as might be expected based on Fig. 7, turns out not to be necessary to make these distributions match.

The dashed lines in Fig. 8 show the corresponding sojourn time quantiles from simulations with this overhead model. For both the fork-join and split-merge models, the simulations with overhead match the Spark experiments very reasonably.

Fig. 11 shows the simulated stability regions as a function of tasks-per-job, k, for both split-merge and fork-join systems. The solid lines are for simulation without overhead and the dashed lines with the simulated overhead model discussed above. The maximum stable utilization in the split-merge system is dramatically increased by using tiny tasks, but begins dropping again around 2000 tasks per job ($\varkappa \approx 36$) due to overhead. The fork-join system is stable up to a utilization of 1.0 in general, so no stability improvement is possible. In fact, the overhead reduces the stability region gradually as the tinyfication factor increases.

III. Network Calculus for Parallel Computing Systems

We start by presenting the analytical tools and notation needed to derive and understand our analytical models of tiny tasks. We build on the approach of [20], [21], making use of a max-plus version of the stochastic network calculus.

A. Notation and Background

We use the same naming and notation for the random processes that were discussed in Section I-A, and introduce some additional items. As before $A(n)$ and $D(n)$ for $n \geq 1$ denote the **arrival time** and **departure time** of job n, and $Q_i(n)$ denotes the **task service time** of task $i \in [1, k]$ of job n. By convention we take $A(0) = 0$. $A(m,n) = A(n) - A(m)$ is the **inter-arrival time** between jobs $n \geq m$. $L(n) = \sum_{i=1}^{k} Q_i(n)$ denotes the (total) **workload** of the job. The **job service time** $\Delta(n)$ is the total time a job spends in service. Note that for the parallel models, $L(n)$ and $\Delta(n)$ are generally not equal.

Servers are modeled using a definition of **max-plus server** with **service process** $S(m,n)$, adapted from [49, Def. 6.3.1].

Definition 1 (Max-plus server): A system with arrivals $A(n)$ and departures $D(n)$ is an $S(m,n)$ server under the max-plus algebra if it holds for all $n \geq 1$ that

$$D(n) \leq \max_{m \in [1,n]} \{A(m) + S(m,n)\}.$$

Applying this definition to the **sojourn time**, $T(n) = D(n) - A(n)$ for $n \geq 1$, we obtain:

$$T(n) \leq \max_{m \in [1,n]} \{S(m,n) - A(m,n)\}. \quad (4)$$

In the case of first-come first-served service, an expression for the **waiting time** $W(n) = [D(n-1) - A(n)]_+$, where $[X]_+ = \max\{0, X\}$, can be derived in the same way.

In the case of single-server systems, the service process corresponds to the cumulative service time of jobs m to n and we have the relationship $S(m,n) = \sum_{\nu=m}^{n} \Delta(\nu)$ where $\Delta(\nu)$ is the service time of job ν. When we move to the multi-server setting, the definition of $S(m,n)$ becomes more subtle. For example $S(m,n)$ may not generally be defined in increments of $\Delta(n)$.

Just as in [20], [21] we will make use of **moment generating functions** (MGFs) of the arrival and service processes. The MGF of a random variable X is defined as $\mathsf{M}_X(\theta) = \mathsf{E}[e^{\theta X}]$ where θ is a free parameter. The MGF has the properties that $\mathsf{M}_{X+Y}(\theta) = \mathsf{M}_X(\theta)\mathsf{M}_Y(\theta)$ for X and Y independent, and $\mathsf{M}_{cX}(\theta) = \mathsf{M}_X(c\theta)$ for any constant, c.

A common class of MGF models are (σ, ρ)**-envelopes** defined in [49, Def. 7.2.1]. These are adapted to max-plus servers in [21, Def. 2].

Definition 2 ((σ, ρ)-Arrival and Service Envelopes): An arrival process, $A(m,n)$, is (σ_A, ρ_A)-lower constrained if for all $n \geq m \geq 1$ and $\theta > 0$ it holds that

$$\mathsf{E}\left[e^{-\theta A(m,n)}\right] \leq e^{-\theta(\rho_A(-\theta)(n-m) - \sigma_A(-\theta))}.$$

A service process, $S(m,n)$, is (σ_S, ρ_S)-upper constrained if for all $n \geq m \geq 1$ and $\theta > 0$ it holds that

$$\mathsf{E}\left[e^{\theta S(m,n)}\right] \leq e^{\theta(\rho_S(\theta)(n-m+1) + \sigma_S(\theta))}.$$

Max-plus servers with (σ, ρ)-envelopes are models of G | G|1 queues, and a variety of stochastic processes satisfy the definition including Markov and periodic processes [49], [50],

[51]. In this work we restrict ourselves to GI | GI|1 queues. In the iid case we have $\sigma_A(-\theta) = \sigma_S(\theta) = 0$.

As an example, consider the classical M | M|1 queue. The arrival process has iid inter-arrival times $A(n, n+1) \sim \text{Exp}(\lambda)$, and MGF $\mathsf{E}[e^{-\theta A(n,n+1)}] = \lambda/(\lambda+\theta)$ for $n \geq 1$ and $\theta > 0$. It follows that

$$\rho_A(-\theta) = -\frac{1}{\theta} \ln\left(\frac{\lambda}{\lambda+\theta}\right), \tag{5}$$

for $\theta > 0$. Similarly, for iid service times $\Delta(n) \sim \text{Exp}(\mu)$ we have $\mathsf{E}[e^{\theta\Delta(n)}] = \mu/(\mu-\theta)$ for $n \geq 1$ and $\theta \in (0, \mu)$ so that

$$\rho_S(\theta) = \frac{1}{\theta} \ln\left(\frac{\mu}{\mu-\theta}\right), \tag{6}$$

for $\theta \in (0, \mu)$. In this example, parameter $\rho_A(-\theta)$ decreases with $\theta > 0$ from the mean inter-arrival time to the minimal inter-arrival time (possibly zero), and $\rho_S(\theta)$ increases with $\theta > 0$ from the mean service time to the maximal service time (possibly infinity).

Performance bounds are obtained using a basic theorem of the stochastic network calculus, e.g., [21, Th. 1].

Theorem 1 (Statistical sojourn time bound): Given an $S(m,n)$ server with iid inter-arrival times with envelope rate $\rho_A(-\theta)$ and iid service times with envelope rate $\rho_S(\theta)$. For any $\theta > 0$ that satisfies $\rho_S(\theta) \leq \rho_A(-\theta)$, the waiting time for all $n \geq 1$ is bounded by

$$\mathsf{P}[W(n) > \tau] \leq e^{-\theta\tau},$$

and the sojourn time by

$$\mathsf{P}[T(n) > \tau] \leq e^{\theta\rho_S(\theta)} e^{-\theta\tau}.$$

B. State of the Art in Parallel Systems

Here we will summarize prior results for split-merge, fork-join, single-queue fork-join, and ideal partitioning parallel systems for the "big-tasks" case, where the number of tasks per job, k, equals the number of servers, l [20].

1) Split-Merge: In the big-tasks split-merge model all tasks in a job start simultaneously. Therefore the system can be modeled like a single-server system where each job's service time is determined by that of its maximal task $\Delta(n) = \max_{i\in[1,l]}\{Q_i(n)\}$. Hence, for $n \geq m \geq 1$ the model can be expressed as a max-plus server with service process [20], [21]

$$S(m,n) = \sum_{\nu=m}^{n} \max_{i\in[1,l]}\{Q_i(\nu)\}. \tag{7}$$

For iid $Q_i(n) \sim \text{Exp}(\mu)$ it also follows that the service process of the split-merge model (7) has service envelope

$$\rho_S(\theta) = \frac{1}{\theta}\sum_{i=1}^{l} \ln\left(\frac{i\mu}{i\mu-\theta}\right), \tag{8}$$

for $\theta \in (0, \mu)$ [20]. The sojourn time bound depicted in Fig. 3 is obtained by substitution of (8) into Theorem 1 and optimizing subject to $0 < \theta < \mu$.

2) Fork-Join: The service process of the fork-join model is

$$S(m,n) = \max_{i\in[1,l]}\left\{\sum_{\nu=m}^{n} Q_i(\nu)\right\}, \tag{9}$$

for $n \geq m \geq 1$ [20]. This says that $S(m,n)$ is determined by the maximal sequence of tasks that are assigned to a server. Clearly, for a given set of task service times $Q_i(n)$, the service process $S(m,n)$ of the fork-join model (9) will be less than or equal to that of the split-merge model (7). The sojourn time can be obtained from (4) by substitution of (9), substitution of $Q_i(m,n) = \sum_{\nu=m}^{n} Q_i(\nu)$, and reordering of the maxima to give

$$T(n) \leq \max_{i\in[1,l]}\left\{\max_{m\in[1,n]}\{Q_i(m,n) - A(m,n)\}\right\}.$$

Then $T_i(n) = \max_{m\in[1,n]}\{Q_i(m,n) - A(m,n)\}$ are the individual task sojourn times at server $i \in [1,l]$. For each server $i \in [1,l]$ Theorem 1 can be used to derive $\mathsf{P}[T_i(n) > \tau]$ and applying the union bound, [20], [21] gives us $\mathsf{P}[T(n) > \tau] \leq \sum_{i=1}^{l} \mathsf{P}[T_i(n) > \tau]$. The same steps can be used to derive a waiting time bound. For the homogeneous case it follows from Theorem 1 that

$$\mathsf{P}[T(n) > \tau] \leq l e^{\theta\rho_Q(\theta)} e^{-\theta\tau},$$

for any $\theta > 0$ satisfying $\rho_Q(\theta) \leq \rho_A(-\theta)$. For the case of iid exponential inter-arrival and task service times, we can use $\rho_A(-\theta)$ from (5) and substitute $\rho_S(\theta)$ from (6) for $\rho_Q(\theta)$ to obtain the fork-join sojourn time bound plotted in Fig. 3.

Since in this case $\rho_Q(\theta)$ and $\rho_A(-\theta)$ converge towards the mean task service time and the mean inter-arrival time, respectively, as $\theta \to 0$, the condition $\rho_Q(\theta) \leq \rho_A(-\theta)$ implies that the fork-join model is stable up to a utilization of one.

3) Single-Queue Fork-Join: The service process of the single-queue fork-join model is more involved. The corresponding results in Fig. 3 are obtained from [21, Th. 4]. The single-queue fork-join model is also a special case (for $k = l$) of Theorem 2 in this paper.

4) Ideal Partition: If jobs are composed of k iid exponential tasks with parameter μ, then the jobs' total workload has distribution $L(n) \sim \text{Erlang}(k, \mu)$. If jobs with this workload distribution were instead divided into l equally-sized tasks, then the tasks would have an $\text{Erlang}(k, l\mu)$ distribution, so that

$$\rho_Q(\theta) = \frac{k}{\theta} \ln\left(\frac{l\mu}{l\mu-\theta}\right) \tag{10}$$

for $\theta \in (0, l\mu)$. Since the tasks of each job are equisized, all tasks of each job start and finish in unison. Hence, the system functions identically to a single server. The sojourn time bound depicted in Fig. 3 follows by substitution of (10) into Theorem 1.

IV. Split-Merge Systems With Tiny Tasks

In this section, we extend the split-merge model to cases with finer task granularity, to understand how using tiny tasks extends

its stability region and improves its sojourn time. As before we assume l workers and $k \geq l$ tasks per job.

Lemma 1 (Tiny tasks split-merge model): The split-merge model with l workers and $k \geq l$ tasks per job is a max-plus server. Given iid exponential task service times with parameter μ, its service process has envelope rate $\rho_S(\theta) = \rho_X(\theta) + (k - l)\rho_Z(\theta)$, where

$$\rho_X(\theta) = \frac{1}{\theta} \sum_{i=1}^{l} \ln\left(\frac{i\mu}{i\mu - \theta}\right),$$

for $\theta \in (0, \mu)$, and

$$\rho_Z(\theta) = \frac{1}{\theta} \ln\left(\frac{l\mu}{l\mu - \theta}\right),$$

for $\theta \in (0, l\mu)$. The expected job service time is

$$\mathsf{E}[\Delta(n)] = \frac{1}{\mu}\left(\frac{k}{l} + \sum_{i=2}^{l} \frac{1}{i}\right).$$

We note that for the special case $k = l$, Lem.1 recovers the envelope rate (8) and stability condition of the conventional split-merge model. Sojourn time and waiting time bounds follow by substitution of Lemma 1 into Theorem 1.

Proof: First, we show that the tiny tasks split-merge model is a max-plus server. Let $V_i(n)$ be the time task $i \in [1, k]$ of job $n \geq 1$ starts service. Since the first l tasks of a job start at the same time, we have for $i \in [1, l]$ that

$$V_i(n) = \max\{A(n), D(n-1)\}. \tag{11}$$

For $i \in [l+1, k]$ we have

$$V_i(n) = V_{i-1}(n) + Z_{i-1}(n), \tag{12}$$

where $Z_{i-1}(n)$ is the time from the start of task $i-1$ of job n until the next server becomes available.

We can express the departure time $D(n)$ of job n relative to the start time of its last task,

$$D(n) = V_k(n) + X(n), \tag{13}$$

where

$$X(n) = \max_{i \in [1,l]}\{Y_i(n)\} \tag{14}$$

and $Y_i(n)$ for $i \in [1, l]$ are the residual service times of the tasks, including task k, that are in service when task k starts service at $V_k(n)$. By repeated substitution of (12) into (13), it follows that

$$D(n) = V_l(n) + \left[\sum_{i=l}^{k-1} Z_i(n)\right] + X(n).$$

With (11) this becomes

$$D(n) = \max\{A(n), D(n-1)\} + \Delta(n), \tag{15}$$

where we write the service time of job n as

$$\Delta(n) = \left[\sum_{i=l}^{k-1} Z_i(n)\right] + X(n). \tag{16}$$

By recursive insertion of (15) we obtain

$$D(n) = \max_{m \in [1,n]} \left\{A(m) + \sum_{\nu=m}^{n} \Delta(\nu)\right\},$$

i.e., the tiny tasks split-merge model is a max-plus server with service process $S(m, n) = \sum_{\nu=m}^{n} \Delta(\nu)$.

Next, we consider the distribution of $X(n)$ and $Z_i(n)$. Due to the memorylessness of the iid exponential task service times, the residual service times $Y_i(n)$ are also iid exponential with the same parameter μ. Regarding $Z_i(n)$, note that when any task $i \in [l, k]$ of job n starts service all servers are busy, so that the time until the next server becomes idle is the minimum of the residual service times of the l tasks that are in service. Thus $Z_i(n)$ for $i \in [l, k-1]$ is the minimum of l iid exponential random variables with parameter μ, and therefore the $Z_i(n)$ are iid exponential with parameter $l\mu$.

To derive the MGF of (16), we apply the identity (used by the authors of [20] to compute the stability region of the split-merge model) $\max_{i \in [1,l]}\{Y_i(n)\} =_d \sum_{i=1}^{l} Y_i(n)/i$ to (14), obtaining

$$\mathsf{M}[X(n)](\theta) = \prod_{i=1}^{l} \mathsf{M}\left[\frac{Y_i(n)}{i}\right](\theta) = \prod_{i=1}^{l} \frac{i\mu}{i\mu - \theta}, \tag{17}$$

for $\theta \in (0, \mu)$. Also, we have

$$\mathsf{M}\left[\sum_{i=l}^{k-1} Z_i(n)\right](\theta) = \mathsf{M}[Z_i(n)]^{k-l} = \left(\frac{l\mu}{l\mu - \theta}\right)^{k-l}, \tag{18}$$

for $\theta \in (0, l\mu)$. The MGF of (16) follows as the product of (17) and (18). Taking the logarithm and dividing by θ gives $\rho_S(\theta)$.

Finally, the expected value $\mathsf{E}[\Delta(n)]$ can then be derived by substituting (14) into (16) and using the identity (19). With $\mathsf{E}[Z_i(n)] = 1/(l\mu)$, and $\mathsf{E}[Y_i(n)/i] = 1/(i\mu)$ this gives us

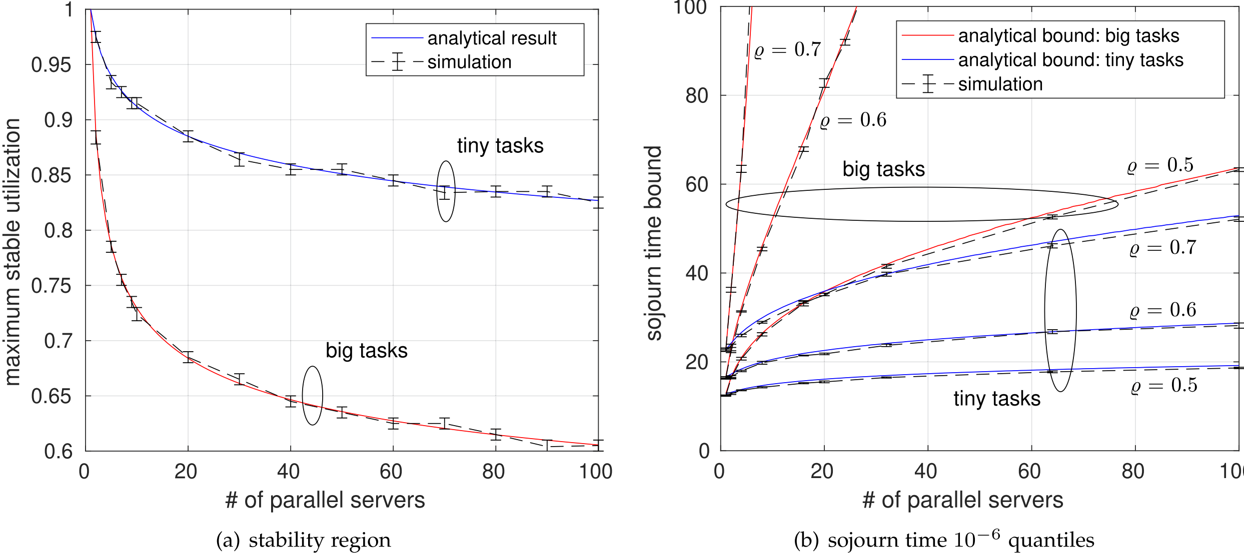
$$\mathsf{E}[\Delta(n)] = \frac{k-l}{l\mu} + \frac{1}{\mu} \sum_{i=1}^{l} \frac{1}{i}.$$

Some reordering of the terms completes the proof. □

Note that the step deducing that the $Y_i(n)$ are iid exponential, and thereby equation (17), required the assumption of iid exponential task service times. This is one reason that we cannot directly use a task service time that includes arbitrary non-exponential overhead in the tiny tasks model.

A. Direct Refinement Into Tiny Tasks

For most of the comparisons in this paper we fix the number of servers, l, and increase the number of tasks per job, k, making the tasks smaller to compensate; for example in Figs. 8, 11, and 13. Another view is to fix the factor of tinyfication $\varkappa = k/l$. Then both k and l must increase proportionately. Because of the relative simplicity of the $k = l$ case of the split-merge model, in this way we can make an especially direct comparison of the effects of tiny tasks, wherein the distribution of the jobs' workloads, $L(n)$, does not change. The following will not be possible with fork-join systems, and will only be used in this section.



(a) stability region (b) sojourn time 10^{-6} quantiles

Fig. 12. Direct refinement of big tasks into tiny tasks for the split-merge model with $\text{Exp}(\lambda)$ arrivals. Big-tasks jobs have $k = l$ $\text{Erlang}(\varkappa, \mu)$ tasks. Tiny-tasks jobs have $k = \varkappa l$ $\text{Exp}(\mu)$ tasks, and are therefore a direct refinement of the corresponding big-tasks jobs. In all plots $\mu = \varkappa = 20$ so the utilization is determined by the arrival rate $\varrho = \varkappa\lambda/\mu = \lambda$.

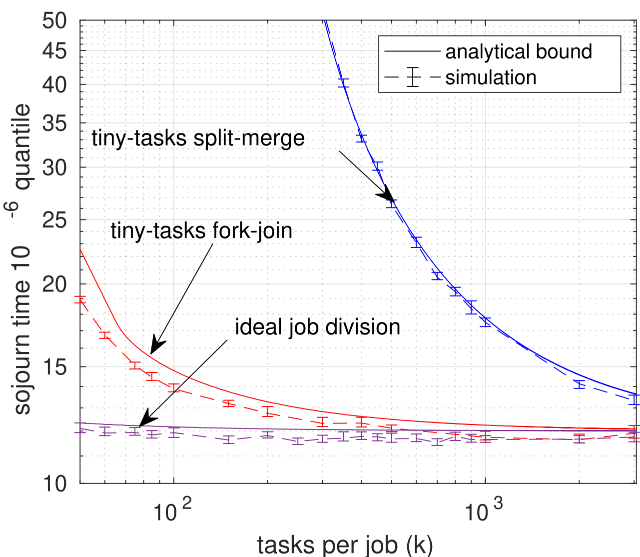


Fig. 13. Comparison of the sojourn time bounds of the single-queue fork-join and split-merge models with $l = 50$ servers and k tiny tasks. As a reference, the sojourn time bounds of a system with ideal partition, where a job is partitioned into l equisized tasks, is shown. Jobs have exponential inter-arrival times with parameter $\lambda = 0.5\text{s}^{-1}$ and are composed of k exponential tiny tasks with parameter $\mu = \frac{k}{l}$. The bounds are exceeded with probability at most $\varepsilon = 10^{-6}$.

Fix some integer tinyfication factor, $\varkappa$. In the tiny tasks model we assume $k = \varkappa l$ tasks per job, with $\text{Exp}(\mu)$ service times. In the equivalent big tasks split-merge model , we have $k = l$ tasks per job with $Q_i(n) \sim \text{Erlang}(\varkappa, \mu)$ service times. In this way, the tiny tasks model is a direct refinement of the equivalent big tasks model, and importantly, the distribution of the total workload of each job stays the same. The key property here is that the uniformly random partitioning of $\text{Erlang}(\varkappa, \mu)$ samples into $\varkappa$ sub-intervals, produces iid $\text{Exp}(\mu)$ samples [52], whereas random partitioning of most random variables results in non-independent sub-intervals.

B. Stability of Split-Merge With Tiny Tasks

To deduce the stability region of the big tasks split-merge model with iid exponential task service times, the authors of [20] use the identity

$$\max_{i\in[1,l]}\{Q_i(n)\} =_d \sum_{i=1}^{l} \frac{Q_i(n)}{i}, \tag{19}$$

where $=_d$ denotes equality in distribution. It follows that for iid task service times $Q_i(n) \sim \text{Exp}(\mu)$, the mean job service time is $\mathsf{E}[\Delta(n)] = \sum_{i=1}^{l} 1/(i\mu)$. For iid inter-arrival times $A(n, n+1) \sim \text{Exp}(\lambda)$, i.e., with mean inter-arrival time $1/\lambda$, the split-merge system is stable if and only if [20, Eq. 21]

$$\frac{1}{\lambda} > \frac{1}{\mu}\sum_{i=1}^{l}\frac{1}{i}.$$

Recall that $\varrho = \lambda/\mu$ is the utilization. The term $\sum_{i=1}^{l} \frac{1}{i}$ is the lth harmonic number. These have the logarithmic asymptotic limit $\gamma + \ln l$, where $\gamma \approx 0.577$ is the Euler constant. Hence, the maximum stable utilization decays proportionally to $1/\ln l$.

The tiny tasks split-merge model is stable as long as the expected inter-arrival time $\mathsf{E}[A(n, n+1)]$ is larger than the expected job service time $\mathsf{E}[\Delta(n)]$. For iid inter-arrival times $A(n, n+1) \sim \text{Exp}(\lambda)$, the condition $\lambda\mathsf{E}[\Delta(n)] < 1$ implies stability. Since the expected total workload of a job is $\mathsf{E}[L(n)] = \sum_{i=1}^{k} \mathsf{E}[Q_i(n)] = k\mathsf{E}[Q_i(n)]$, the mean service provided to each job by each of the l servers will be $\varkappa\mathsf{E}[Q_i(n)]$. The utilization of each server is then $\varrho = \lambda\varkappa\mathsf{E}[Q_i(n)]$. Since $\lambda < 1/\mathsf{E}[\Delta(n)]$ for stability, the stability region, i.e., the maximum

stable utilization for the tiny tasks model, is

$$\varrho < \frac{\varkappa \mathsf{E}[Q_i(n)]}{\mathsf{E}[\Delta(n)]} = \frac{1}{1+\frac{1}{\varkappa}\sum_{i=2}^{l}\frac{1}{i}}, \qquad \text{(tiny tasks)} \qquad (20)$$

where we inserted $\mathsf{E}[\Delta(n)]$ from Lemma 1 and $\mathsf{E}[Q_i(n)] = 1/\mu$.

For comparison, consider the equivalent big task split-merge model where the number of tasks k equals the number of servers l and $Q_i(n) \sim \text{Erlang}(\varkappa, \mu)$. From (7) the service process of the big task model is determined by the maximal task, $\Delta(n) = \max_{i\in[1,l]}\{Q_i(n)\}$. Since $\Delta(n)$ is non-negative, we can derive the expected value by integration of the complementary cumulative distribution function (CCDF) as

$$\begin{aligned}\mathsf{E}[\Delta(n)] &= \int_0^\infty 1 - \mathsf{P}[\Delta(n) \le x]dx \\ &= \int_0^\infty 1 - (\mathsf{P}[Q_i(n) \le x])^l dx, \qquad (21)\end{aligned}$$

where we used that $\mathsf{P}[\max_{i\in[1,l]}\{Q_i(n)\} \le x] = \mathsf{P}[Q_1(n) \le x \cap Q_2(n) \le x \cap \dots \cap Q_l(n) \le x] = (\mathsf{P}[Q_i(n) \le x])^l$ for iid random variables $Q_i(n)$. Finally, we insert the Erlang-$\varkappa$ CDF

$$\mathsf{P}[Q_i(n) \le x] = 1 - e^{-\mu x}\sum_{i=0}^{\varkappa-1}(\mu x)^i/i! \qquad (22)$$

and solve (21) numerically. Again, $\lambda\mathsf{E}[\Delta(n)] < 1$ implies stability, and with $\varrho = \lambda\mathsf{E}[Q_i(n)]$, where $\mathsf{E}[Q_i(n)] = \varkappa/\mu$ is the expected service time of the big tasks, the stability region for the big-tasks model follows as

$$\varrho < \frac{\mathsf{E}[Q_i(n)]}{\mathsf{E}[\Delta(n)]} = \frac{\varkappa}{\mu\mathsf{E}[\Delta(n)]} \qquad \text{(big tasks)} \qquad (23)$$

where $\mathsf{E}[\Delta(n)]$ is given by (21).

The stability region of the split-merge model with both big tasks and tiny tasks for $\varkappa = 20$ is shown in Fig. 12(a). The model with tiny tasks shows a clear improvement of the stability region.

C. Sojourn Time Bounds

Fig. 12(b) compares sojourn time bounds of the big tasks and tiny tasks models for equivalent parameters. In the case of tiny tasks, the sojourn time bound is derived by substitution of parameter $\rho_S(\theta)$ from Lemma 1 into Theorem 1. In the case of big tasks, we first have to derive the envelope rate $\rho_S(\theta)$ of the service process $S(m,n)$ defined in (7) for iid tasks with $Q_i(n) \sim \text{Erlang}(\varkappa, \mu)$. We derive the MGF of $S(n)$ by integration of the CCDF as

$$\mathsf{E}[e^{\theta S(n)}] = \int_0^\infty 1 - \mathsf{P}[e^{\theta S(n)} \le x]dx.$$

Since for $\theta > 0$ it holds that $e^{\theta S(n)} \ge 1$ we have

$$\mathsf{E}[e^{\theta S(n)}] = 1 + \int_1^\infty 1 - \mathsf{P}\left[S(n) \le \frac{\ln(x)}{\theta}\right]dx.$$

By definition of $S(n) = \max_{i\in[1,l]}\{Q_i(n)\}$ it follows that $\mathsf{P}[S(n) \le x] = (\mathsf{P}[Q_i(n) \le x])^l$ so that

$$\mathsf{E}[e^{\theta S(n)}] = 1 + \int_1^\infty 1 - \left(\mathsf{P}\left[Q_i(n) \le \frac{\ln(x)}{\theta}\right]\right)^l dx.$$

We insert the Erlang-$\varkappa$ CDF (22) and solve the integral numerically. The envelope rate follows as $\rho_S(\theta) = \ln(\mathsf{E}[e^{\theta S(n)}])/\theta$ and the sojourn time bound is derived by use of Theorem 1.

The sojourn time bounds in Fig. 12(b) are shown for iid inter-arrival times $A(n, n+1) \sim \text{Exp}(\lambda)$. Three different λ are used, corresponding to utilizations of 0.5, 0.6, and 0.7. The use of tiny tasks improves the sojourn time bounds significantly. The improvement is larger under higher utilizations, where the big tasks split-merge model becomes unstable for even relatively small numbers of servers l.

A more in-depth examination of the relationship between stability, performance, and idle times in the big tasks versus tiny tasks cases is given in [1].

V. Single-Queue Fork-Join With Tiny Tasks

We consider a single-queue fork-join model with tiny tasks. The model is similar to the split-merge model with tiny tasks depicted in Fig. 5, with one difference: there is no synchronization constraint at the start of a job. I.e. a new job can start service as soon as a worker becomes idle and there are no unserviced tasks from the previous job. As a consequence, workers will not idle at the end of a job if there are other jobs waiting. Furthermore, jobs can overtake each other and finish service out of sequence. For analytical tractability we study a model where the jobs depart in sequence, $D(n) \le D(n+1)$. That is, jobs that finish service must wait in a queue until their predecessors have departed. This added constraint does not affect the waiting time, $W(n)$, and means that the sojourn time bounds produced for this model will be strictly larger than those of the plain single-queue fork-join with tiny tasks model.

Theorem 2 (Tiny tasks fork-join model): Given a fork-join model with l servers and $k \ge l$ iid exponential tiny tasks with parameter μ and iid inter-arrival times with envelope rate $\rho_A(-\theta)$. For any $\theta \in (0, \mu)$ that satisfies $k\rho_Z(\theta) \le \rho_A(-\theta)$, the waiting time of task $i \in [1, k]$ of job $n \ge 1$ is bounded by

$$\mathsf{P}\left[W_i(n) \ge \tau\right] \le e^{\theta(i-1)\rho_Z(\theta)}e^{-\theta\tau},$$

and the sojourn time of job $n \ge 1$ by

$$\mathsf{P}\left[T(n) \ge \tau\right] \le e^{\theta((k-1)\rho_Z(\theta)+\rho_X)}e^{-\theta\tau}.$$

The parameters $\rho_X(\theta)$ and $\rho_Z(\theta)$ are given in Lemma 1.

As a special case for $k = l = 1$, Theorem 2 recovers the single server case Theorem 1 for exponential jobs with envelope rate (6). Also, for $k = l$, Theorem 2 recovers the waiting time bound for the single-queue fork-join model (without tiny tasks) [21, Th. 4]. For the sojourn time bound [21] uses a slightly different derivation technique that can provide tighter bounds mostly at low utilizations.

The proof of Theorem 2 is more involved, and can be found in [1]. It is important to note that the random variables, X and Z, represent the same things as in Lemma 1 and have the same MGFs. This will be important when we incorporate overhead into the model in Section VI.

Fig. 13 compares sojourn time bounds obtained for the single-queue fork-join and split-merge models with $l = 50$ servers and a varying number k of tiny tasks per job to the equivalent system

with the ideal partition of jobs into l equisized tasks. The bounds in the figure are evaluated with violation probability $\varepsilon = 10^{-6}$. As in Fig. 8 we increase the number of tasks per job, k, and decrease the task service time proportionally, so that the mean job workload $\mathsf{E}[L(n)]$ remains constant. In case of the ideal partition, we substitute $\mu = \frac{k}{l}$ into (10) to get the corresponding envelope rate, $\rho_Q(\theta)$ that can be inserted into Theorem 1.

As k increases, the sojourn time bound of the fork-join model with tiny tasks quickly approaches that of the ideal partition. For the tiny tasks split-merge model, for small k, the divergence between the models is quite large, a consequence of the restricted stability region of the split-merge model. For large k, both models approach the ideal partition.

A more detailed look at the sojourn time bounds relative to the optimal, and how the utilization level affects the rate of convergence, is given in [1].

VI. Incorporating Overhead Into the Tiny-Tasks Analytical Models

Both Lemma 1 and Theorem 2 define two random variables: $X(n)$ is the time from the start of task k of job n until the job departs, and $Z_i(n)$ is the time from the start of task $i > l$ of job n until the next server becomes available. As long as the task service time distributions are exponential, the memorylessness of the exponential distribution makes it possible to derive expressions and MGFs for $X(n)$ and $Z_i(n)$. Without the assumption of memorylessness, the time remaining for each task-in-progress would depend on how long it had already been running, and these random variables would become extremely complicated if not impossible to solve for.

We have, however, created and simulated a detailed model of the overhead in a Spark system based on experimental measurements. It is worth considering how this model can be incorporated into the analytical models to provide some approximation of system performance with overhead. This is important for approximating the optimal range of k and l under differing overhead conditions.

Section II.F identified two main classes of overhead: task-service overhead which effectively increases the service times of the tasks (it blocks subsequent tasks from starting), and pre-departure overhead that delays the departure of the job, but does not affect the processing of subsequent tasks within a job. In the simulations we modeled the task overhead as having a constant and an exponential component (2). In the analytical model, a problem arises when we take the MGF of (2), because the very small exponential limits us to $\theta < \mu_{task}^{ts}$. For our purpose here, we model the entire task-service overhead using its mean

$$\mathsf{E}\left(O_i(n)\right) = c_{task}^{ts} + \frac{1}{\mu_{task}^{ts}}. \tag{24}$$

This means that we neglect the outlier task overhead values noted in Section II.F.

The pre-departure overhead was modeled in Section II.F using both a per-task and per-job constant (3). In the fork-join case this is simple to model, since it just adds to each job's sojourn time, and does not affect the waiting time. In the split-merge case, however, pre-departure overhead does not block subsequent tasks within a job, bit it does block subsequent jobs from starting. It therefore does affect waiting times and needs to be incorporated into the analytical model.

A. Overhead in the Fork-Join Model

We need to calculate how constant task overhead affects $X(n)$ and $Z_i(n)$ and their MGFs. We refer to these modified random variables as $X^o(n)$ and $Z_i^o(n)$. In the fork-join case, since the pre-departure overhead is non-blocking, $X^o(n)$ must be taken to be the time from the start of task k of job n until the job is ready to depart, absent the pre-departure overhead. Therefore we have

$$X^o(n) = X(n) + c_{task}^{ts} + \frac{1}{\mu_{task}^{ts}} \tag{25}$$

A constant, c, has MGF $e^{c\theta}$, $\theta > 0$. Because the MGF of a sum of independent random variables is the product of their MGFs, we can easily compute the MGF of $X^o(n)$, and then with Lemma 1

$$\rho_{X^o}(\theta) = c_{task}^{ts} + \frac{1}{\mu_{task}^{ts}} + \frac{1}{\theta}\sum_{i=1}^{l} \ln\left(\frac{i\mu}{i\mu - \theta}\right). \tag{26}$$

We incorporate task overhead into $Z_i(n)$ similarly, but with a key difference. $Z_i(n)$ is the time from the start of task i of job n until the next worker becomes available, and therefore has the distribution of the minimum of l exponentials, which is Exp$(l\mu)$. We can add the constant overhead to $Z_i(n)$, but this has the effect of adding the full task overhead to each active task each time a new task is scheduled. On average, each task would pay the task overhead l times during its execution. We make the more reasonable approximation that each active task pays a $1/l$ fraction of the task overhead each time a new task is scheduled. That is, we make the approximation

$$Z_i^o(n) = Z_i(n) + \left(c_{task}^{ts} + \frac{1}{\mu_{task}^{ts}}\right)/l. \tag{27}$$

which gives

$$\rho_{Z^o}(\theta) = \frac{c_{task}^{ts} + \frac{1}{\mu_{task}^{ts}}}{l} + \frac{1}{\theta}\ln\left(\frac{l\mu}{l\mu - \theta}\right). \tag{28}$$

This allows us to evaluate an approximate waiting time distribution with overhead using Theorem 2. In order to approximate the distribution of sojourn time with overhead, we still need to add in the pre-departure overhead. If τ_ϵ is the sojourn time quantile approximation obtained using Theorem 2, then we add

$$\tau_\epsilon^o = \tau_\epsilon + c_{job}^{pd} + k \cdot c_{task}^{pd}. \tag{29}$$

The resulting approximate 0.99 sojourn time quantiles for $l = 50, \lambda = 0.5$, and $\mu = k$, and varying number of tasks per job (k) are plotted in Fig. 8(b) along with corresponding simulation and Spark experimental data. The increase in the sojourn time due to task overhead matches very well with the simulation and experimental data.

B. Overhead in the Split-Merge Model

Approximating the waiting and sojourn time distributions with overhead using Lemma 1 is handled in much the same way, except that now the pre-departure overhead is blocking. This does not change the approximation for $Z_i^o(n)$, but now we have

$$X^o(n) = X(n) + c_{task}^{ts} + \frac{1}{\mu_{task}^{ts}} + c_{job}^{pd} + k \cdot c_{task}^{pd} \tag{30}$$

and we no longer add the pre-departure overhead directly to the sojourn time as in equation (29). This gives

$$\rho_{X^o}(\theta) = c_{task}^{ts} + \frac{1}{\mu_{task}^{ts}} + c_{job}^{pd} + k \cdot c_{task}^{pd} + \frac{1}{\theta} \sum_{i=1}^{l} \ln \left(\frac{i\mu}{i\mu - \theta} \right) \tag{31}$$

which, along with (28) can be used directly with Lemma 1. The resulting approximated sojourn time quantiles are plotted in Fig. 8(a), and again, fit the simulation and experimental results very well. Importantly, in both the split-merge and fork-join cases, the analytical approximations provide good estimates for the optimal number of tasks-per-job, balancing the benefits of task tinyfication with the cost of scheduling and processing overhead.

VII. Tiny Tasks in Shared Memory Clusters

We have focused on a model of parallel computing that matches well with the "embarrassingly parallel" design of modern map-reduce engines such as Spark and Hadoop, where the tasks are completely independent until the synchronization point at the job's departure. Another popular class of platforms and APIs for parallelization are **task-centric shared-memory systems**, such as Cilk [32], OpenMP [33], and TBB [34]. Typically these systems maintain a thread pool on a single computer, and break jobs up into tasks which are serviced by the threads in the pool. Parallel programs for these systems are typically written in C or C++, and the programmer has tremendous control over how the division of work and dependencies between tasks will be handled. Unlike in map-reduce systems, the division of work can be recursive, with tasks being further subdivided into smaller tasks, and instead of a centralized scheduler assigning tasks to workers as they become available, these systems typically achieve load balancing through work stealing within the thread pool, rather than a centralized scheduler as with Spark's standalone scheduler.

A benchmarking study of several versions of the three shared memory systems mentioned above, which includes an investigation of task granularity, is presented in [6]. The authors find that some parallel systems handle finer granularity tasks better or worse than others, and that this also depends on the type of workload being benchmarked.

Unless one has access to massively parallel SMP or NUMA equipment, the degree of parallelism achievable by shared-memory systems on a commodity-grade cluster is limited by the number of cores in a single machine. Intel briefly released a "cluster" version of their OpenMP compiler and runtime which allowed programmers to write programs in the shared-memory style of OpenMP and have it execute across a cluster of separate machines. The complexity of emulating shared memory behavior over a network appears to have suffered from poor performance at scale [38], [53], and since at least 2011 the project was unsupported. Thus, deploying shared memory parallelism on a commodity-grade cluster requires another solution.

A popular tool for implementing parallel computations, that naturally scales to clusters of many independent machines, is the Message Passing Interface (MPI) [35]. MPI implementations provide a standardized API to launch processes across a cluster of machines, create barriers within those processes, and exchange data between them. Notably, however, current MPI implementations do not inherently support work stealing (or other load balancing) between these distributed processes. Load balancing can be implemented by the developer however, for example [54], [55].

A hybrid approach to building parallel programs on commodity clusters has therefore become popular. For example, in a hybrid MPI/OpenMP program, MPI is used to start many processes across a cluster, and within each of these processes, OpenMP maintains a thread pool and performs its normal load balancing. The tasks executing in the OpenMP threads or the parent processes can use the MPI API to synchronize their state and exchange or collect results. This type of hybrid approach encompasses a range of different configuration possibilities, and performance comparisons between some of these have been carried out [37], [38], [39]. We use this hybrid approach in our experiments, with MPI spawning a single multi-threaded OpenMP process on each machine, and OpenMP managing access to the threads locally. Each process uses the MPI API to get information about its execution environment, and at its completion, each process calls `MPI_Gather()`, which acts as a departure barrier and also collects the task execution data at the master node.

The third component needed is a job scheduler to which users submit programs, and which schedules them on the available CPUs. Slurm (Simple Linux Utility for Resource Management) [56], [57] seems to be by far the most popular tool of its type, and is what we use. Slurm has many advanced scheduling features, and can integrate with several popular MPI implementations to monitor processes and core availability. We use Slurm in a relatively basic mode, as a FIFO scheduler that starts one MPI process on each of p servers, and waits for the previous job to depart before starting the next one (the split-merge mode of operation). Each process runs a pool of r OpenMP threads. Thus, there are $l = p \cdot r$ workers in total. Each job contains k tasks, and in these experiments we require that k be a multiple of p, so that each MPI process receives the same number of tasks.

We will refer to a system of this type as a (p, r, l) **hybrid split-merge system**. A schematic of the hybrid Slurm/MPI/OpenMP system we use is shown in Fig. 14. Note that since there is no load balancing between the r MPI processes, this system does not exactly conform to the split-merge model we have studied so far. It is similar, however, and stands to benefit from the improved load balancing of tiny-tasks within each MPI process.

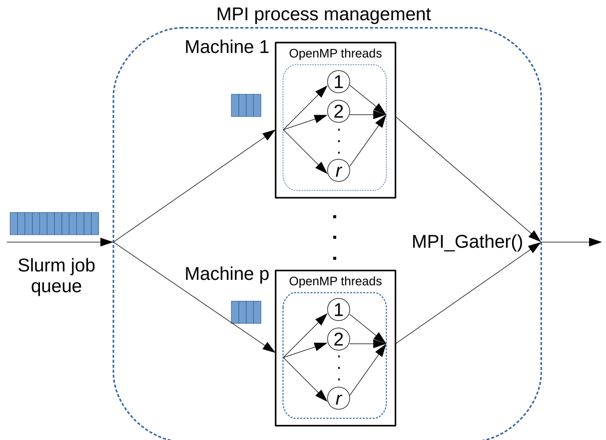


Fig. 14. The (p, r, l) hybrid split-merge model with MPI/OpenMP. Each job uses MPI to spawn p processes, one on each of p multi-core machines. Jobs are divided into k tasks, and each process is assigned a batch of k/p tasks. Inside each process OpenMP executes r threads which service the processes' batch of tasks.

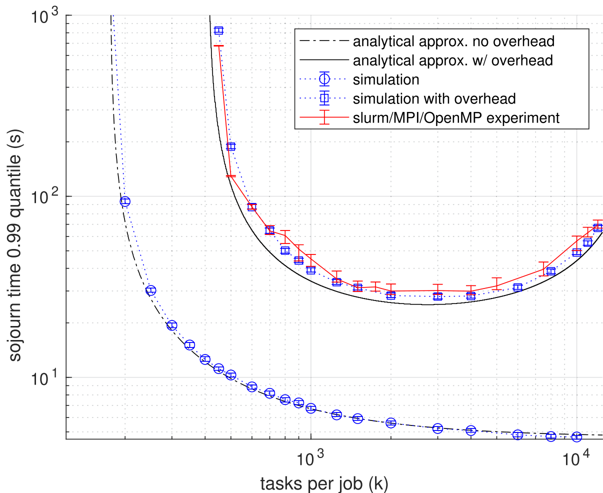


Fig. 15. Sojourn time 10^{-2} quantiles for a range of task granularities on a hybrid Slurm/MPI/OpenMP parallel system with $p = 5$ MPI processes (each on a different server), $r = 10$ OpenMP threads per process, for a total of $w = 50$ workers. The tasks have exponential service times, and the task service rate is scaled to maintain a constant expected per-job workload.

There has been some research on analytically modeling the performance of shared memory systems. These tend to focus on analyzing the properties of the Directed Acyclic Graph (DAG) that represents the temporal dependencies in a chain of parallel operations. These sorts of analyses are important, as in shared memory systems the programmer has extensive control over the nature of the parallelism, even more so than in modern map-reduce engines. In order to make the problems tractable, these have focused on a restricted class of DAGs [32], [58]. Even so, the analytical models produced are not amenable to queueing theoretic analysis, in the sense that we cannot use them to answer questions about parallel systems that have jobs arriving from a random process, and whose task service times are drawn from a probability distribution. They are focused on computing deterministic Worst-Case Response Time bounds, or estimates of mean behavior; not the probability distribution of waiting and sojourn times that we obtain using stochastic network calculus.

We carried out simple benchmarking experiments on the hybrid Slurm/MPI/OpenMP cluster, comparable to those done on the Spark cluster. Fig. 15 plots sojourn time quantiles from experiments with $p = 5$ MPI processes (each on a different server), $r = 10$ OpenMP threads per process, for a total of $l = 50$ workers, for a range of task granularities. The jobs have exponential arrivals with rate $\lambda = 0.5\ \text{s}^{-1}$, and the tasks have exponential service times, with rate $\mu = k/l$, scaled to maintain a constant expected per-job workload. Just as in the Spark experiments, the workload is a dummy workload where each task chooses its service time from the desired distribution, and generates random numbers until that time expires. The task service data is gathered in a shared-memory data structure in each MPI process, and then collected at the master node with an `MPI_Gather()`, where it is logged to a file. Comparing this to Fig. 8 for conventional split-merge and fork-join systems, we notice that the sojourn times in the MPI/OpenMP system increase much more slowly with k, allowing us to use task granularities more than double what was possible with Spark. Based on our measurements, the task service overhead in this OpenMP system is almost negligible, and was only possible to measure with millisecond resolution when at least 1000 tasks are serviced serially by a single thread. On the other hand, the job and task pre-departure overhead has a much greater impact. This is due partly to the heavier burden of starting and disposing of new processes for each job, and also because of the logging done by Slurm and the job itself. Because the job submission is handled through Slurm and executed by MPI, each job is a separate invocation of the executable, and it is not possible to gather the job and task data in RAM to be logged later.

Fig. 15 also includes data from simulations of a $(p, r, l) = (5, 10, 50)$ hybrid system with and without simulated overhead. The parameters used in the four-parameter overhead model for the simulations in Fig. 15 are given in the table below. Comparing to the corresponding table for our Spark system in Section II.F, we observe that the task service overhead is indeed much smaller, but the job pre-departure overhead is correspondingly larger.

task-service	c^{ts}_{task}	=	2.86×10^{-3} ms
	μ^{ts}_{task}	=	$3.49 \times 10^{5}\ \text{s}^{-1}$
pre-departure	c^{pd}_{job}	=	560 ms
	c^{pd}_{task}	=	0.023 ms

Fig. 15 also contains sojourn time quantiles estimated by our analytical approximation, both with and without overhead. We see that the analytical approximation with overhead matches real system results and the simulated (p, r, l) hybrid system fairly well, and, importantly, accurately predicts the region of optimal task granularity. Note that even in the case with no overhead, for the hybrid shared-memory system the analytical model is no longer a bound, but only an approximation. It will tend to underestimate the sojourn time distribution because the hybrid system has no load balancing between MPI processes. This

discrepancy will increase as the number of threads per process approaches one, but will be reduced as the task granularity becomes finer, since the variance in the workload assigned to each process decreases. Extending the analytical model to one that can provide strict performance bounds for (p, r, l) hybrid systems is an area for future work.

VIII. Conclusion

Using "tiny tasks" in parallel processing systems refers to the practice of splitting jobs into k tasks, where k is larger than the number of workers, l. Using moderately tiny tasks in task-centric parallel systems has long been a practical performance enhancement employed by practitioners, but at some point the performance benefit of dividing jobs into more smaller tasks is outweighed by the various types of system overhead.

We performed extensive experiments on an Apache Spark system using carefully controlled task size distributions to quantify the performance benefits of using tiny tasks, and measure and model the types of overhead that interfere. We developed a model for how this overhead scales with job size, and implemented it in a simulator. We presented analytical models looking at the improved stability region of split-merge systems using tiny tasks, and derived analytical bounds on the waiting and sojourn times of both split-merge and fork-join systems with tiny tasks. We used these analytical bounds, along with our model of scheduler overhead, to produce an analytical approximation for the sojourn time quantiles of both split-merge and fork-join systems with tiny tasks with overhead. Finally we performed experiments with a hybrid Slurm/MPI/OpenMP cluster, simulated the corresponding (p, r, l) hybrid split-merge system, and compared our performance approximations to the experimental results.

In all cases these approximations fit very well to our experimental and simulation results. Besides illuminating the fundamental properties of parallel systems and how they are affected by task granularity, our analytical approximation model which incorporates system overhead should be a useful tool for optimizing task granularity on real systems. The reliability of the approximation, even for hybrid shared-memory systems, demonstrates the power and flexibility of the analytical model.

Acknowledgments

This manuscript is a revised and extended version of [1] which appeared in the IEEE Infocom 2020 proceedings.

References

[1] M. Fidler, B. Walker, and S. Bora, "Tiny tasks - a remedy for synchronization constraints in multi-server systems," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 1063–1072.

[2] Apache Spark Documentation, "Tuning Spark," 2019. Accessed: Jul. 25, 2019. [Online]. Available: https://spark.apache.org/docs/latest/tuning.html

[3] S. Ryza, "How-to: Tune your apache spark jobs (part 2)," 2015. Accessed: Jul. 25, 2019. [Online]. Available: https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

[4] T. Gautier, C. Perez, and J. Richard, "On the impact of openMP task granularity," in *Evolving OpenMP for Evolving Architectures*. Berlin, Germany: Springer, 2018, pp. 205–221.

[5] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. 24th ACM Symp. Oper. Syst. Princ.*, 2013, pp. 69–84.

[6] A. Podobas, M. Brorsson, and K. Faxén, "A comparative performance study of common and popular task-centric programming frameworks," *Concurr. Comput. Pract. Exp.*, vol. 27, no. 1, pp. 1–28, 2015.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. USENIX Conf. Hot Top. Cloud Comput.*, 2010, pp. 10–10.

[8] Apache Software Foundation, "Spark," 2022. Accessed: Sep. 01, 2022. [Online]. Available: https://spark.apache.org

[9] Apache Software Foundation, "Apache Hadoop," 2022. Accessed: Sep. 01, 2022. [Online]. Available: https://hadoop.apache.org/

[10] P. Mika, "Flink: Semantic Web technology for the extraction and analysis of social networks," *J. Web Semant.*, vol. 3, no. 2/3, pp. 211–223, 2005. [Online]. Available: http://dblp.uni-trier.de/db/journals/ws/ws3.html#Mika05

[11] L. Flatto and S. Hahn, "Two parallel queues created by arrivals with two demands," *SIAM J. Appl. Math.*, vol. 44, no. 5, pp. 1041–1053, 1984.

[12] R. Nelson and A. N. Tantawi, "Approximate analysis of fork/join synchronization in parallel queues," *IEEE Trans. Comput.*, vol. 37, no. 6, pp. 739–743, Jun. 1988.

[13] A. S. Lebrecht and W. J. Knottenbelt, "Response time approximations in fork-join queues," in *Proc. Annu. UK Perform. Eng. Workshop*, 2007, pp. 1–8.

[14] S.-S. Ko and R. F. Serfozo, "Sojourn times in G/M/1 fork-join networks," *Nav. Res. Logistics*, vol. 55, no. 5, pp. 432–443, May 2008.

[15] X. Tan and C. Knessl, "A fork-join queueing model: Diffusion approximation, integral representations and asymptotics," *Queueing Syst.*, vol. 22, no. 3/5, pp. 287–322, Sep. 1996.

[16] 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.

[17] E. Varki, "Mean value technique for closed fork-join networks," *ACM Sigmetrics Perform. Eval. Rev.*, vol. 27, no. 1, pp. 103–112, May 1999.

[18] F. Alomari and D. A. Menascé, "Efficient response time approximations for multiclass fork and join queues in open and closed queuing networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1437–1446, Jun. 2014.

[19] G. Kesidis, B. Urgaonkar, Y. Shan, S. Kamarava, and J. Liebeherr, "Network calculus for parallel processing," in *Proc. MAMA Workshop ACM SIGMETRICS*, 2015, pp. 48–50.

[20] 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, pp. 261–291, Aug. 2016.

[21] M. Fidler, B. Walker, and Y. Jiang, "Non-asymptotic delay bounds for multi-server systems with synchronization constraints," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 7, pp. 1545–1559, Jul. 2018.

[22] B. Walker, S. Bora, and M. Fidler, "Stability and scaling of parallel systems with blocking start or departure barriers," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 460–469.

[23] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. Amsterdam, The Netherlands: Elsevier, 2012.

[24] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in tensorflow," 2018, *arXiv:1802.05799*.

[25] Uber Engineering, "Meet Horovod: Uber's open source distributed deep learning framework for TensorFlow," 2021. Accessed: Jun. 28, 2021. [Online]. Available: https://eng.uber.com/horovod/

[26] T. Foldi, C. von Csefalvay, and N. A. Perez, "JAMPI: Efficient matrix multiplication in spark using barrier execution mode," *Big Data Cogn. Comput.*, vol. 4, no. 4, Nov. 2020, Art. no. 32.

[27] P. Harrison and S. Zertal, "Queueing models with maxima of service times," in *Proc. Int. Conf. Modelling Techn. Tools Comput. Perform. Eval.*, 2003, pp. 152–168.

[28] G. Joshi, Y. Liu, and E. Soljanin, "On the delay-storage trade-off in content download from coded distributed storage systems," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 989–997, May 2014.

[29] B. Walker, "Benchmarking and simulating the fundamental scaling behaviors of a MapReduce engine," in *Proc. IFIP Netw. Conf. Workshops*, 2017, pp. 1–6.

[30] R. Nelson, D. Towsley, and A. N. Tantawi, "Performance analysis of parallel processing systems," *IEEE Trans. Softw. Eng.*, vol. 14, no. 4, pp. 532–540, Apr. 1988.

[31] F. Baccelli, A. M. Makowski, and A. Shwartz, “The fork-join queue and related systems with synchronization constraints: Stochastic ordering and computable bounds,” *Adv. Appl. Probability*, vol. 21, no. 3, pp. 629–660, Sep. 1989.

[32] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, 1996.

[33] R. Menon and L. Dagum, “OpenMP: An industry-standard API for shared-memory programming,” *Comput. Sci. Eng.*, vol. 01, pp. 46–55, Jan. 1998.

[34] TBB (Intel Threading Building Blocks). in *Encyclopedia of Parallel Computing*, D. A. Padua Ed., Berlin, Germany: Springer, 2011, Art. no. 2029.

[35] Message Passing Interface Forum, “MPI: A message-passing interface standard version 4.0,” Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[36] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, “Big Data analytics in the cloud: Spark on hadoop versus MPI/OpenMP on beowulf,” *Procedia Comput. Sci.*, vol. 53, pp. 121–130, 2015.

[37] N. Drosinos and N. Koziris, “Performance comparison of pure MPI versus hybrid MPI-OpenMP parallelization models on SMP clusters,” in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, 2004, Art. no. 15.

[38] R. Rabenseifner, G. Hager, and G. Jost, “Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes,” in *Proc. 17th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process.*, 2009, pp. 427–436.

[39] H. Brunst and B. Mohr, “Performance analysis of large-scale openMP and hybrid MPI/OpenMP applications with vampir NG,” in *Proc. Int. Workshop OpenMP*, 2005, pp. 5–14.

[40] K. Ousterhout et al., “The case for tiny tasks in compute clusters,” in *Proc. USENIX Conf. Hot Top. Oper. Syst.*, 2013, pp. 14–14.

[41] E. Totoni, S. R. Dulloor, and A. Roy, “A case against tiny tasks in iterative analytics,” in *Proc. 16th Workshop Hot Top. Oper. Syst.*, 2017, pp. 144–149.

[42] A. S. Documentation, “Cluster mode overview,” 2015. Accessed: Jun. 17, 2021. [Online]. Available: https://spark.apache.org/docs/latest/cluster-overview.html

[43] M. Zaharia et al., “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, Art. no. 2.

[44] EsotericSoftware, “Kryo: A fast and efficient object graph serialization framework for Java,” 2021. [Online]. Available: https://github.com/EsotericSoftware/kryo

[45] The University of Utah, “Emulab,” 2021. Accessed: May 17, 2021. [Online]. Available: https://www.emulab.net

[46] S. Bora, “3.0.0-SNAPSHOT_additional_logging branch of Apache Spark,” 2022. [Online]. Available: https://github.com/stefan1723/spark/tree/3.0.0-SNAPSHOT_additional_logging

[47] D. Agrawal et al., “Sparkbench – A spark performance testing suite,” in *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things*, R. Nambiar and M. Poess Eds., Berlin, Germany: Springer, 2016, pp. 26–44.

[48] B. Walker and S. Bora, “Forkulator – Simulator for parallel queueing systems,” 2022. [Online]. Available: https://github.com/brentondwalker/forkulator

[49] C. S. Chang, *Performance Guarantees in Communication Networks*. Berlin, Germany: Springer-Verlag, 2000.

[50] F. P. Kelly, “Notes on effective bandwidths,” in *Stochastic Networks: Theory and Applications*. London, U.K.: Oxford University, 1996, no. 4, pp. 141–168.

[51] M. Fidler, “A survey of deterministic and stochastic service curve models in the network calculus,” *IEEE Commun. Surv. Tuts.*, vol. 12, no. 1, pp. 59–86, First Quarter 2010.

[52] F. W. Steutel, “Random division of an interval*,” *Statist. Neerlandica*, vol. 21, no. 3/4, pp. 231–244, 1967.

[53] C. Terboven, D. A. Mey, D. Schmidl, and M. Wagner, “First experiences with intel cluster OpenMP,” in *OpenMP in a New Era of Parallelism*, R. Eigenmann and B. R. de Supinski Eds., Berlin, Germany: Springer, 2008, pp. 48–59.

[54] P. Samfass, J. Klinkenberg, and M. Bader, “Hybrid MPI+OpenMP reactive work stealing in distributed memory in the PDE framework sam(oa)^2,” in *Proc. IEEE Int. Conf. Cluster Comput.*, 2018, pp. 337–347.

[55] K. Ouyang, M. Si, A. Hori, Z. Chen, and P. Balaji, “CAB-MPI: Exploring interprocess work-stealing towards balanced MPI communication,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–15.

[56] M. A. Jette, A. B. Yoo, and M. Grondona, “SLURM: Simple Linux utility for resource management,” in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Berlin, Germany: Springer-Verlag, 2002, pp. 44–60.

[57] SchedMD SLURM Support and Development, “SLURM workload manager,” 2022. [Online]. Available: https://slurm.schedmd.com/

[58] J. Sun, N. Guan, J. Sun, and Y. Chi, “Calculating response-time bounds for OpenMP task systems with conditional branches,” in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 169–181.

Stefan Bora received the bachelor’s and master’s degrees in computer engineering from Leibniz Universität Hannover, in 2016 and 2018, respectively. He was a doctoral student with Leibniz Universität Hannover, Germany.

Brenton Walker received the bachelor’s of science from the University of Wisconsin-Madison and the Ph.D. degree in mathematics from the University of Maryland-College Park in 2014. He is a postdoctoral researcher with Leibniz Universität Hannover, Germany. He was previously an ERCIM fellow with SICS in Stockholm Sweden.

Markus Fidler (Senior Member, IEEE) received the doctoral degree in computer engineering from RWTH Aachen University, Germany, in 2004. He was a postdoctoral fellow of NTNU Trondheim, Norway, in 2005 and the University of Toronto, ON, Canada, in 2006. During 2007 and 2008, he was an Emmy Noether research group leader with Technische Universität Darmstadt, Germany. Since 2009, he has been a professor of communications networks with Leibniz Universität Hannover, Germany.