

Towards Fine-grained Parallelism in Parallel and Distributed Python Libraries

Jamison Kerney and Ioan Raicu
College of Computing
Illinois Institute of Technology
Chicago, Illinois
jkenrey@hawk.iit.edu, iraicu@cs.iit.edu

John Raicu and Kyle Chard
Department of Computer Science
University of Chicago
Chicago, Illinois
johnny.raicu@gmail.com, chard@uchicago.edu

Abstract—There is a growing need, for example in machine learning and analytics, to decompose applications into smaller schedulable units. Such decomposition can improve performance, reduce energy consumption, and increase resource utilization. Unfortunately, enabling fine-grained parallelism comes with significant overheads and requires improvements at all layers of the programming stack. We consider the challenges of supporting fine-grained parallelism in the increasingly popular Python-based programming libraries. Specifically, we focus on Parsl, a Python library that is widely used to parallelize the execution of fine-grained Python functions. Parsl’s Python-based runtime supports a maximum throughput of around 1200 tasks per second—insufficient to meet modern application needs. We perform a comprehensive analysis of Parsl and identify areas that prohibit it from achieving higher throughput. We first profile Parsl components and identify that, with fine-grained tasks workers are often not saturated. We find that tasks spend a majority of their time in the components between the scheduler and worker, however, we also learned that the scheduler is capable of submitting thousands of tasks per second. We then focused on developing new optimizations and implementing crucial components in C to improve throughput. Our new implementation increases Parsl’s throughput 6 fold.

1. Introduction

The past decade has seen the case made for fine-grained parallelism in HPC [1] and Data centers [2], [3]. Small tasks enable clusters to make precise scheduling decisions and rapidly respond to changes in compute load. Increasingly, we see that scientific programs run many very short tasks (e.g., for machine learning inference) across large-scale HPC systems comprised of thousands of nodes and tens of thousands (or more) cores. The demand for these characteristics in software systems has led to a plethora of frameworks for both HPC [4], [5], [6] and Data centers [7], [8], [9] that are optimized for small tasks.

Python has become one of the most pervasive programming languages, in part because it is a language that enables beginners and experts to quickly develop programs. Python offers a simple interface, clear error messages, and rapid development time. Given Python’s robust numerical

libraries and extensive ecosystem of scientific frameworks, many make use of Python for scientific computations. However, addressing modern workload requirements increasingly relies on the use of parallel and distributed computing resources; unfortunately, Python’s heavy interpreter and Global Interpreter Lock (GIL) make it difficult to scale. While there are plans to remove the GIL, Parsl was constructed around the GIL, and thus our improvements are constructed with the GIL in mind. Various Python-based libraries have been developed to overcome these limitations and enable distributed execution in Python (e.g., Dask [10], Ray [11], and Parsl [12]). We focus on Parsl, a parallel programming library that maintains Python’s accessible user interface while dispatching code for concurrent and asynchronous execution on both local and remote computing resources.

Parsl is used for a diverse range of scientific applications and is deployed on large supercomputers at enormous scales (thousands of nodes and hundreds of thousands of cores [12]). A review of various Parsl applications shows that tasks can run for short durations, effectively capping the scale of deployment due to the limited throughput. An analysis of Globus Compute workloads [13], a Python-based serverless computing framework for HPC [14] that relies on Parsl for task execution, found that the median task time was 340ms.

Motivated by these small tasks we seek to understand the limitations of Parsl’s performance and indirectly the limitations of using Python for such purposes. We conduct an extensive empirical evaluation of Parsl and illustrate a detailed picture of Parsl’s runtime. We identify areas for improvement and conduct experiments to evaluate the efficacy of these changes. Ultimately, we identify shortcomings that cannot be resolved in Python and implement a C version of Parsl’s scheduler. We show that our optimizations and new C implementation can achieve six-fold better performance on microbenchmarks. Furthermore, we benchmark our improvements using a common scientific application and demonstrate throughput improvements.

In this work we contribute the following.

- In-depth analysis of a modern Python workflow system

- Scheduler improvements for fine-grain parallelism
- Comparison of task-based parallel workflow systems

The paper is structured as follows. In [section 2](#) we describe the Parsl architecture. In [section 3](#) we analyze Parsl’s performance from two angles. We first understand Parsl by profiling each component in its execution pipeline. Second, we place timestamps on each task to understand in which component do tasks spend most of their time. Informed by profiling and tagging data in [section 4](#) we describe the changes we made to Parsl’s components to improve throughput. [section 5](#) we evaluate the optimizations described in [section 4](#). We discuss the limitations of our solutions in [section 7](#). In [section 6](#) we discuss related work and conclude our findings in [section 8](#).

2. Parsl Architecture

To give a background for our analysis we briefly describe Parsl’s architecture. The relevant parts of Parsl’s architecture are shown in Figure 1. We explain each component and relate them to common elements of parallel programming frameworks.

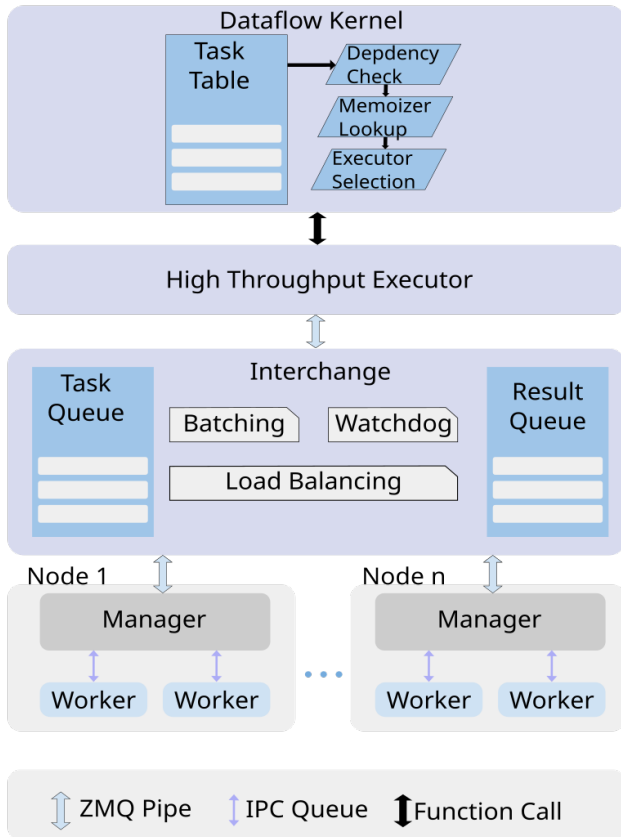


Figure 1: Parsl’s Execution Pipeline

2.1. Dataflow Kernel

The DataFlow Kernel (DFK) is Parsl’s scheduler. It is responsible for selecting an appropriate resource for execution and dispatching tasks to that resource. The DFK maintains a dictionary of Task Records that maps task id to task record objects. Each task t , with dependencies $t_0, \dots, t_n$, is assigned a callback function that attempts to launch all dependent tasks $t_0, \dots, t_n$ when it (task t) has finished. The DFK automatically memoizes tasks to decrease redundant computation. The DFK is written entirely in Python.

2.2. Executor

The *executor* is an abstraction for computation resources. Parsl maintains several executors that are built for different types of workloads: HighThroughput Executor, Low Latency Executor, and Extreme-Scale Executor. Several external executors have also been integrated, such as RADICAL-Cybertools [15], Flux [16], and WorkQueue [17]. The executor is associated with a *provider*, which allows Parsl to elastically provision compute resources via different interfaces (e.g., batch scheduler, container orchestration system, or cloud API). The DFK, executor, and provider are all started by the same Python interpreter. The executor manages computation resources and partitions compute nodes into blocks. The user allocates a minimum number of blocks, configures the number of nodes per block, and sets a maximum number of blocks. Given that information, the executor will dynamically add blocks and remove blocks. In this work, we focus on Parsl’s default executor, the High Throughput Executor (HTEX).

2.3. Interchange

The interchange is a critical part of Parsl’s execution framework. It enables the use of supercomputers, clouds, and clusters. The interchange is deployed on the same node as the DFK and executor but lies within a separate Python interpreter. The interchange receives tasks from the executor and sends tasks to managers via ZMQ sockets. The interchange maintains a queue for tasks and results. It combines tasks into batches and sends the batches to managers. The interchange chooses the manager based on its advertised capacity (in terms of number of tasks). The interchange is responsible for tracking the status of workers, it does this by occasionally sending messages to workers. Since the interchange interacts directly with managers it is responsible for load balancing. The interchange tracks each manager’s capacity. It sends batches of tasks to a manager so long as the number of tasks assigned to it is less than its capacity.

2.4. Manager

Managers are responsible for a subset of workers on a node. Managers can reside on the same node as DFK,

executor, and interchange, however, they typically are deployed on separate compute nodes. Managers communicate with the interchange via ZMQ pipes and communicate with their workers via IPC queues. The same result and task queues are shared by the manager and all of its workers. During initialization, the manager creates separate processes for each worker and begins sending them tasks. Managers effectively allow for multiplexing of communication from the interchange to the many workers deployed on a node and allow Parsl to consume fewer ports on each node.

2.5. Worker

Workers receive tasks from managers, execute tasks, and return their results back to managers. They are single-threaded Python processes that always reside on the same node as their manager.

3. Analysis of Parsl's Throughput

To build a comprehensive image of Parsl we employ two methods to measure performance. First, we profile the Python processes for each component. Some of the components are I/O heavy (e.g., interchange) thus they are multi-threaded processes. For these processes, we profile each thread. Second, we augment the Parsl codebase to capture the timestamps when entering and exiting each component. This approach captures where time is spent from the task's perspective. For both methods we used a no-op workload, so we could isolate system overheads from execution of the task. It is essential to note that we chose to augment the code rather than use Parsl's existing logging mechanism as our initial profiling data showed that logging consumed a significant portion of the time in the DFK. When we disabled all logging Parsl's throughput increased from 1200 to 4000 tasks/second. Our experiments used Python version 3.10. With the exception of the multi-node experiment, all experiments were executed on a testbed with 192 cores (2 hardware threads per core) and 770GiB of ram.

3.1. Profiling

The following profiling data is obtained from an experiment where we executed 10k no-op tasks on a single node with 192 workers. We configured Parsl such that each manager is responsible for eight workers, resulting in 24 managers. [Figure 2](#) shows the results from profiling. The raw profiling data includes a list of functions called within Parsl and the time spent in each of those functions. We categorized each function and divided the time consumed by each category by total thread time, returning the proportion of time dedicated to each category. We discuss each thread in order of task submission.

3.1.1. Dataflow Kernel and Executor. The first bar in [Figure 2](#) corresponds to DFK and HTEX. The DFK and HTEX live in the same process and the DFK invokes HTEX

via Python function calls, thus they occupy the same thread. Our initial observation is that the bulk of time is spent submitting a task. Communication, composed of categories Send and Poll are the most expensive. The functions apart of the categories are responsible for queuing tasks within the interchange. Parsl operations, consume a fifth of total time and are concerned with managing Parsl's task launch state. Those functions include (in order of invocation) `dfk.submit`, `dfk.launch_if_ready`, `dfk.launch`, and `htex.submit`.

3.1.2. Interchange. The interchange is a multi-threaded process. Its main thread is responsible for sending results back to HTEX and sending tasks to available managers. Its command thread ensures that workers are alive and allows the user to manually kill workers. Lastly, its Task puller thread receives work from HTEX. The profiling results of each thread are represented by separate rows in [Figure 2](#)

Task Puller Thread. The task puller thread pops tasks from a ZMQ socket that connects the interchange and executor. The tasks it receives are immediately placed in an in-process queue within the interchange. From this queue, the main thread pulls tasks. The third row in [Figure 2](#) shows where time is spent in the task puller thread. The task puller thread spends upwards of 95% of time receiving tasks. The next most significant cost is deserializing the Python objects it receives which consumes $\sim 1.5\%$ of time. Finally placing the tasks on the internal queue consumes $\sim 1\%$ of time.

Main Thread. Similar to the DFK, communication dominates work performed by the interchange's main thread. However, in this case, Poll accounts for most of communication time. The main thread waits for tasks on an in-process queue between it and the task puller thread. After polling, Send, which involves sending a batch of tasks to a manager consumes the next most time.

Command Thread. The command thread has little activity. The Python interpreter spends the least time on this thread. Almost all of its time is spent receiving confirmation messages from workers. Most of the functions called in this thread are invoked less than 10 times.

3.1.3. Manager. The manager is a dual-threaded process. It contains a thread that pulls tasks from the interchange and sends those tasks to workers and a thread that pulls results from the workers and pushes results to the interchange. Both the threads within the manager are I/O bound.

Task Puller thread. Polling the ZMQ socket consumes most (approximately 90%) time in this thread. While the other operations consume little time.

Result Pusher thread. The result pusher thread's profiling information is displayed in the sixth row of [Figure 2](#). Like the interchange's main thread, polling accounts for roughly three-quarters of thread time. Unlike the previous threads, queue operations are the next most expensive category. This category is composed of functions that wait for and pop results from the worker's results queue. Little time is spent sending results to the interchange, roughly 2.5%. The fact

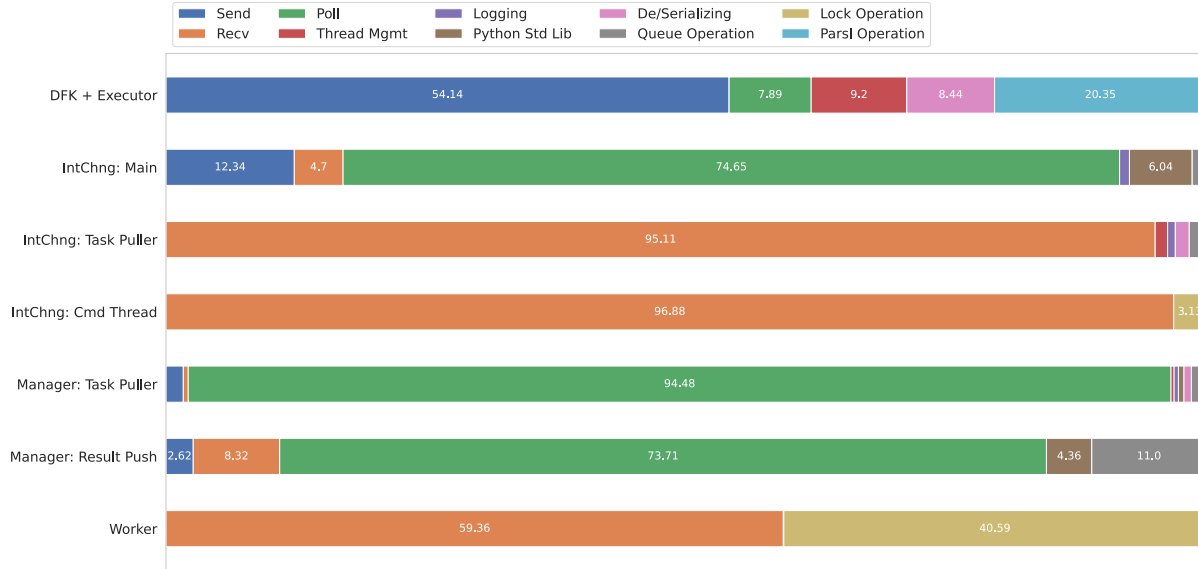


Figure 2: Profiling data showing the time spent in the major components of the Parsl architecture. The figure categorizes time spent in each function in the Parsl source code among ten categories. Time is normalized for each component.

that the majority of the time is spent polling suggests that even with 100s of workers computing results, the workers are unable to saturate the queue.

3.1.4. Worker. Workers are single-threaded Python processes that lie at the end of the execution pipeline. Workers pop tasks off the queue they share with their manager and place results in a separate queue. They receive tasks as serialized Python objects, so deserialization is a cost that workers pay for each task as well as serializing results. The last row of [Figure 2](#) displays the profiling data from the worker.

The worker’s profiling data shows two functions consume most of its time. Reading bytes from the task queue consumes $\sim 60\%$ of time. The other $\sim 40\%$ of time is spent contending over the Semaphore that protects the tasks queue.

3.1.5. Profiling Summary. For all processes, communication in some form consumed most time. Every process/thread downstream of the interchange’s task puller thread spent most of its time polling. While the interchange’s task puller thread spent most of its time receiving Python objects, this suggests that tasks might face a bottleneck within the interchange. Although receive accounted for most of the worker’s time, workers spent a substantial amount of time contending for a semaphore.

3.2. Tagging

Profiling showed where time was spent from the perspective of the process. This information yielded insights that informed our optimizations; however, it does not tell the complete story. We now explore performance from the perspective of a task (rather than the processes). We use

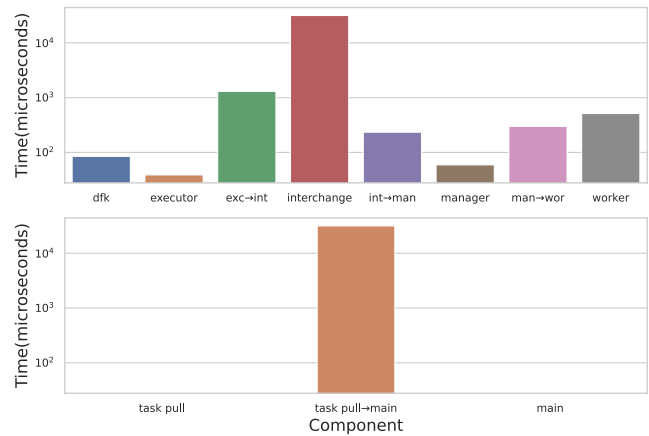


Figure 3: Average time (microseconds) spent in each Parsl component and communication channel for each task in a workload with 10k no-op tasks

a method we call *tagging* to track where tasks spent their time during execution. [Figure 3](#) summarizes our results for a no-op workload with 10k tasks using 192 workers and a single manager.

[Figure 3](#) is a logarithmic graph that shows the average number of microseconds tasks spent in each component and communication channel throughout the entire 10k no-op workload. Tasks spend an order of magnitude more time within the interchange than other components. [Figure 3](#) shows that tasks are piling up in the queue between the interchange’s main and task puller threads. Tasks spend more time on average within that queue than they do in all other components combined.

The second most costly component is the ZMQ connec-

tion between HTEX and the interchange. Tasks likely begin to pile up in this component after the interchange's internal queue.

After the HTEX-interchange ZMQ socket, tasks spend the most time within workers and the manager-worker in-process queue. Tasks spending significant time within the worker may be surprising because the workload is a no-op and the worker code is simple. However, recall that workers spend upwards of 80% of their time contending for a Semaphore, and thus tasks stall, waiting within the worker.

As mentioned previously managers can be deployed on separate nodes thus the queue between the interchange and manager is unique to each manager and uses a ZMQ socket. This communication channel is the least expensive channel on average.

Tasks spend little time in the DFK, HTEX, and Managers. The DFK determines if a task can be launched. When HTEX is invoked, it stores some state about the task and places the task on the pipe. The difference in responsibilities explains the gap in time cost. Workers place themselves in a queue when they can receive work, and the manager sends work to workers in the queue. When a worker receives work it pops itself off that queue. The simplicity of the manager's role explains its low cost.

4. Optimizations

We build upon the detailed analysis performed in previous sections to motivate several important optimizations. We both augment Parsl's architecture as well as modify data structures used for communication.

4.1. Cut out the Middleman

The tagging data showed us that task spend on average $10^4 \mu s$, which is at least 10x more time than any other component or communication channel. The interchange is a component that performs multiple roles (e.g., fault tolerance, and load balancing). Given its complexity and that tasks pile up in its internal queue we decided to remove the interchange altogether. In this exercise, we also removed the manager because it exists to reduce the number of ports consumed by Parsl on a singular node. We call this the DIRECT to worker EXECutor (DIREX).

Removal of the interchange restricts Parsl to operate on a single node. The interchange is core to Parsl's fault tolerance, thus removing it exposes our experiments to the failures of workers. Workers may crash during computation, if a worker was computing a task that has dependencies when it crashed then none of its dependencies could be launched, causing the entire computation to fail. Using no-ops for our benchmarks temporarily buries that concern, since there are no dependencies. However, for some workloads, the exchange of fault tolerance for performance may not be possible.

4.2. Worker Queues

Profiling showed that workers spent a large portion of their time contending over the semaphore that protects the task queue between them and their manager. Semaphore contention is expensive even when the worker count is small. To reduce this cost we assign a task queue to each of the workers. Though semaphores are still protecting the queues, decreasing the number of workers contending for that semaphore to one worker minimizes its cost. Workers are assigned tasks in a round-robin manner.

4.3. Implementing the DFK in C

The DFK, like any scheduler, is core to the performance of the entire runtime system. If the DFK cannot process a million tasks/second then Parsl could never execute a million tasks/second. Thus it is critical that the DFK is fast. In [Figure 3](#) we showed that the DFK and HTEX are inexpensive for each task, however, it is important to note that the average task submit time was $100 \mu s$. To achieve 10k tasks/second, we would need to have an average turnaround time for an entire task of $100 \mu s$. Of course, many components can be parallelized, Parsl can have multiple executors, thus multiple interchanges and multiple managers supervising many workers, however, in its current form every task will pay that $100 \mu s$ toll.

When investigating the causes of the submit cost in the DFK we found many necessary, but computationally trivial, operations that collectively are very expensive. Even the operations that we believed should be inexpensive, such as creating a Task Record object, took $5 \mu s$ which is a non-trivial amount of time if we aim for a total time of $100 \mu s$.

Memory footprint is an important consideration when aiming to support fine-grained parallelism. Python represents data as objects and uses garbage collection for memory management. Invoking 10s of thousands of tasks implicitly creates at least 10s of thousands of objects. Since Parsl represents tasks as a Task Record object which is a dictionary, many objects are created for each task.

Implementing a C DFK represents tasks as structs as shown below, and manages its memory using a wrapper for system memory allocation and deallocation functions provided by Python's C API. Managing its own memory decreases the memory footprint within the Python Interpreter, and thus lightens the burden on the garbage collector when the workload has many tasks. The task struct consumes 128 bytes plus the size of its Python objects. Using Python's sys library one can measure the size of an empty Task Record. An empty Task Record in Python consumes 232 bytes plus the size of its Python objects.

4.4. Bringing it all Together

Removing the interchange and giving each worker its task queue will decrease the cost of sending tasks to workers. Improving the latency of the execution pipeline increases throughput, assuming that the DFK can launch

tasks quickly enough. The C implementation of the DFK can produce many more tasks/second than the Python DFK.

5. Evaluation

We evaluate the performance of our optimizations and the C implementation of the Parsl DFK. The evaluation was performed on the Mystic testbed, as discussed previously.

5.1. Removing the Interchange

To remove the interchange we modified DIREX such that it would spawn its workers. Using a no-op workload with 10k tasks we compare throughput achieved by HTEX and DIREX as a function of the number of workers in [Figure 4](#).

[Figure 4](#) shows that with the interchange removed Parsl using DIREX can achieve the same throughput with 1 worker as Parsl using HTEX can with hundreds of workers. This clearly demonstrates that workers are not saturated. Workers could handle many more tasks if Parsl were able to produce more of them. While removing the interchange increases the throughput of Parsl with low worker counts, at scale, the throughput improvements are minimal at $\sim 15\%$. Moreover, DIREX experiences a slight decrease in throughput as the worker count increases.

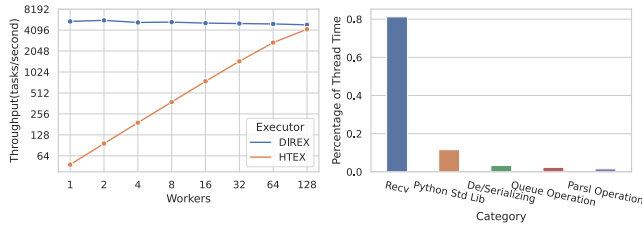


Figure 4: DIREX vs HTEX throughput compared using 10k no-op tasks. Left: throughput as we increase the number of workers. Right: profiling Data from a DIREX worker.

[Figure 4](#) also shows profiling data from a DIREX worker. Using a single worker resulted in semaphore contention accounting for an immeasurable amount of time. Like all previous threads, communication dominates, however, the category Python Std Lib now consumes 10% of total thread time. Functions in the Python Std Lib category include *time.sleep* and *exec*. Exec is used to execute the task and *time.sleep* is the task body itself. Thus removing the interchange has increased the utilization of the worker.

5.2. Worker Queues

[Figure 5](#) shows the throughput of Parsl when workers each pop tasks of the same queue and when workers have their own task queues. As expected, neither scales well, and both show similar trends in throughput. However, the multi-queue model has consistently lower throughput. The difference in throughput is not trivial either, it erases the

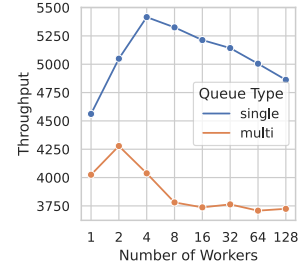


Figure 5: Comparison of throughput when workers share a single task queue (single) and when they have their task queue (multi). Results are shown as we increase the number of workers.

slight throughput gains from removing the interchange and manager.

[Figure 6](#) show the profiling data collected from the workers. Profiling single-queue workers shows that semaphore contention becomes even more expensive when the interchange is removed, however, as shown in the profiling data of multi-queue workers all of that cost is shifted back to reading bytes off the queue.

We conclude that while semaphore contention is expensive, the use of semaphores does not impede throughput. This further motivates our optimization to remove this lock—workers spend 85% of their time acquiring this lock.

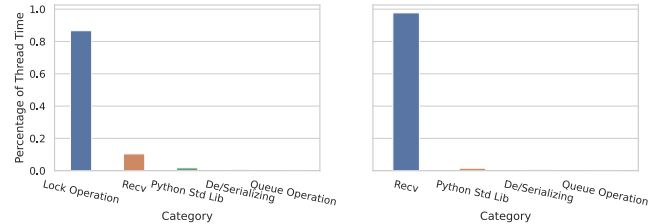


Figure 6: Left: Profiling data from a worker receiving tasks via a single queue. Right: Profiling data from a worker receiving tasks with multi-queue.

5.3. Moving the DFK to C

We now explore throughput and memory footprint of the C DFK. We finally evaluate performance when combining the C implementation with DIREX.

5.3.1. Throughput. Results from above suggest that the workers were not saturated. Profiling showed that Semaphore contention consumed significant time for the workers, but, we did not achieve a significant throughput improvement by removing these semaphores. Similarly, the modest gains to maximum throughput from removing the interchange combined with the increased utilization of the workers implies that there are still throughput gains to be realized from increasing the throughput of Parsl’s scheduler.

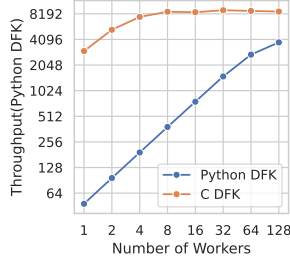


Figure 7: Throughput for the Python and C implementations of the DFK for a 100k no-op workload with varying workers.

Figure 7 compares the throughput accomplished during a 100k no-ops workload with varying amounts of workers. The C implementation of the DFK achieves a maximum throughput double that of standard Python implementation. While the Python DFK scales linearly with a worker count of up to 128 workers, the C DFK never experiences linear scaling. Its maximum throughput is reached with 8 workers. Furthermore, its throughput with 1 worker is similar to the Python DFK’s throughput with many workers. The benchmark in Figure 7 does not include any of the previous changes. The current C DFK implementation is integrated into Parsl without changes to the executor, interchange, manager, and worker.

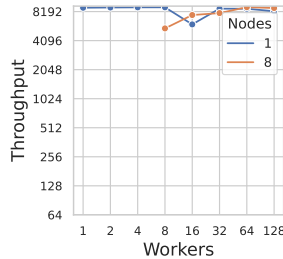


Figure 8: Throughput for the Python and C implementations of the DFK for a 10k no-op workload with varying workers.

Given that CDFK is functional with HTEx (i.e. it can communicate with the interchange), we can deploy work tasks over multiple nodes. Figure 8 shows the throughput achieved using one node and eight nodes. Each node has 32 cores and 64GB of DRAM. With few workers the single node outperforms multi-node. As the number of workers increases the performance of both converge. This demonstrates that our improvements are applicable to large-scale Python workflows.

5.3.2. Memory footprint. With small task counts, necessarily the number of Python objects will be small, and as the task count increases the number of Python objects increases. A large number of Python objects becomes a problem for Python’s garbage collector. Figure 9 shows the throughput difference between the Python DFK with the garbage collector turned on and turned off.

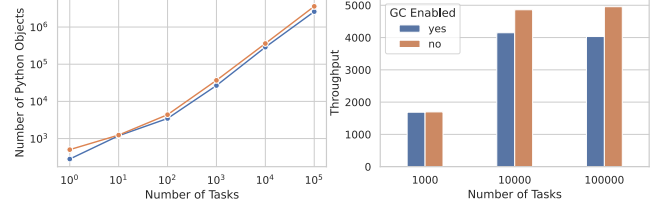


Figure 9: Left: number of objects created in the Python and C DFK as we increase the number of tasks. Right: Throughput comparison with and without the garbage collector for the Python DFK as we increase the number of tasks.

Turning off the garbage collector is simple, however, we do not want our target users, domain scientists, disabling and enabling the garbage collector for improved performance. It could also be dangerous with many task workloads. Having many millions or tens of millions of outstanding Python objects, many of which are Task Records, and the objects they store could cause other issues. Figure 9 is a log plot displaying the difference in object count between standard Parsl’s DFK and C DFK Parsl. The workload used was no-op with varying amounts of tasks.

The gap in object count is roughly 10 times the task count. For example, the final data point shows that C DFK Parsl had approximately 2.6 million Python objects after executing 100k no-ops, while Parsl had 3.6 million Python objects. Careful readers will point out that since the DFK has been moved to C, Python’s garbage collector should not impact our performance. However, recall that the executor and DFK are in the same process. Therefore the executor and DFK share the garbage collector and C DFK creates and tracks Python objects for its tasks, so polluting the interpreter with 100s of thousands more objects could slightly impact throughput.

5.3.3. C DFK + DIREX. We finally integrate the above optimizations and evaluate performance. Recall, the C DFK, which increases the number of tasks/second that can be scheduled, and DIREX, which minimizes the latency of the execution pipeline complement each other. We compare performance against standard Parsl as well as against Dask [10] and Ray [11], two of the most widely used Python parallelism libraries (although more than 70% of Ray is implemented in C++).

Figure 11 shows a peak throughput of roughly 14k tasks/second is achieved with C DFK using DIREX with 4 workers. Previous experiments showed that neither C DFK nor DIREX achieved linear scaling by themselves, however, Parsl achieves linear scaling. While C DFK + DIREX achieves a higher maximum throughput it too does not scale. C DFK with or without DIREX achieves similar throughput. After reaching a maximum throughput with 4 workers it experiences decreased throughput when worker count is increased. The throughput of C DFK using HTEx or DIREX during a no-op workload with 1 worker is around 8k tasks/second. Given this high baseline throughput linear

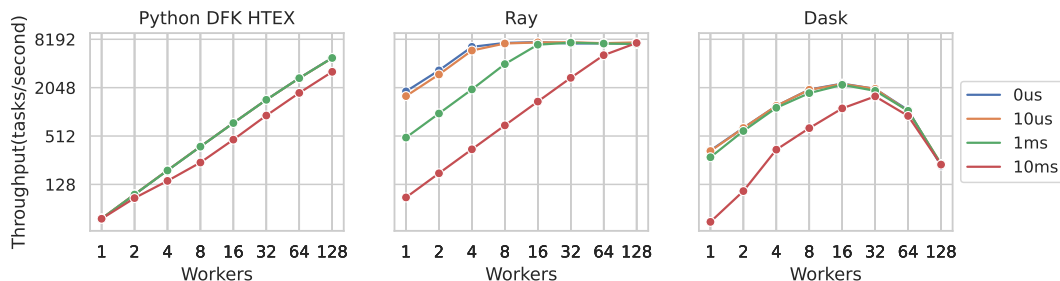


Figure 10: Throughput as we change the granularity (run time) of tasks for the Python DFK, Ray, and Dask.

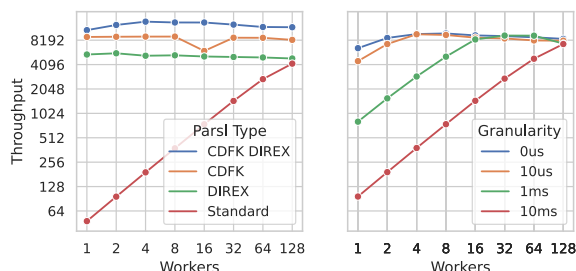


Figure 11: Left: Throughput for the C DFK, DIREX, and standard Parsl as we increase the number of workers. Right: granularity cdfk vs all and granularity

scaling with C DFK would imply $\sim 500k$ tasks/second with hundreds of workers.

Figure 11 also shows the same experiment performed using Parsl’s Python DFK with HTEX. Throughput for tasks of granularity $0\mu s$, $1\mu s$, and $10\mu s$ have the same trend lines and are unaffected by task duration. Task duration begins to affect throughput at $10ms$ granularity.

While CDFK + DIREX achieves higher maximum throughput on finer granularity tasks ($0\mu s$ $10\mu s$) Ray, shown in Figure 10 achieves better scaling and higher throughput for coarse grain tasks. Executing $10ms$ tasks Ray achieves nearly 8k tasks/second with 128 workers, while CDFK + DIREX does not scale past 8 workers and peaks at around 7000 tasks/second. Ray scales with $1ms$ tasks similarly to how it scales with $10ms$ tasks.

Figure 10 also shows the performance of Dask. Dask achieves peak performance with around 16 workers, afterwards, it experiences decreased throughput per worker. In comparison Parsl scales much better, handling up to 128 workers before throughput drops. Important to note is that Dask’s throughput for millisecond and microsecond no-ops tasks are the same. Suggesting that the minimum task granularity for Dask is much higher than for Ray or Parsl. Similar conclusions were made in Slaughter et al. [18].

5.4. Scientific Application

Finally, we explore performance of our techniques on a real-world protein docking workflow. Protein docking is a computational method used in molecular biology to predict

the structure of protein complexes formed when two or more proteins interact. This technique is vital in understanding biological processes and designing therapeutic drugs. It involves simulating the process by which proteins fit together or ‘dock’ to form a stable complex. This is akin to finding the correct way two puzzle pieces fit together among myriad possibilities. This docking process is driven by several factors, including shape complementarity, electrostatic attractions, and hydrophobic interactions. As proteins are highly flexible and complex molecules, the problem of predicting their interactions is computationally demanding.

A typical docking computation can take over 10 minutes on a single core; a typical workload involves multiple protein receptors with millions of possible ligands to dock, yielding a total compute complexity for a brute force approach to be over 21M CPU-hours. This high computational requirement has motivated researchers to leverage machine learning methods to expedite screening.

ParslDock [19], a docking simulation coordinated with Parsl, aims to identify the “best” ligands from a large dataset of potential molecules by efficiently combining simulation (aka docking) and machine learning algorithms on high-performance computing resources. The computational complexity of brute force docking applications is reduced through machine learning methods.

Key to this work is the workflow graph generated by ParslDock. Each task consists of a molecule that is formatted and passed to a machine-learning module for inference. This generates a bag-of-tasks graph, such a task graph is highly parallelizable and has very short-running tasks. We use this scientific application to demonstrate the improved performance achieved by our optimizations.

We execute ParslDock on the Mystic testbed mentioned previously. The docking simulation displayed in Figure 12 shows runtime with varying numbers of workers and batch size. The simulation was run across 100k total molecules. For serial, standard Parsl with Python DFK, and CDFK. We see the best runtimes are achieved with the largest batch size. This suggests that there is still an opportunity to improve performance for fine-grained tasks.

Figure Figure 12a shows the runtime of the docking simulation as a function of the batch size using serial code. Using a batch size of 1000 molecules the serial code achieves a runtime of around ~ 35 seconds. Figure

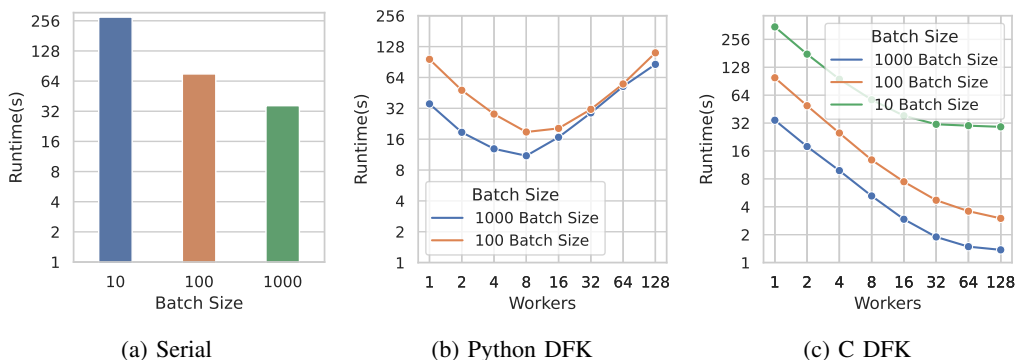


Figure 12: ParslDock runtime for Serial, Standard Parsl, and CDFK implementations for different batch sizes as we scale the number of workers.

Figure 12b displays the runtime achieved by standard Parsl with the Python DFK. Parsl achieves its lowest runtime using 8 workers with a batch size of 1000, ~ 11 s. With standard Parsl using many workers or few workers returns a high runtime, ~ 100 seconds. Figure 12c shows the runtime achieved by CDFK Parsl. It scales linearly with the number of workers used, plateauing with hundreds of workers. CDFK Parsl with 128 workers and a batch size of 1000 simulates 100k molecules in ~ 1.5 seconds. This runtime is 10x faster than standard Parsl’s lowest runtime and 30x faster than serial.

6. Related Work

Rust Dask Scheduler. The most relevant related work was an investigation of Dask’s throughput when its scheduler is implemented in Rust [20]. As discussed in their paper Dask utilizes a work-stealing scheduler. When all of a task’s dependencies have been resolved the task is given to a worker such that its time to start is minimized. If there is a load imbalance, saturated workers will have tasks stolen from them by unsaturated workers. In their work, they first compare Dask’s scheduler to a random scheduler. In this exercise, they find that the random scheduler achieves performance near Dask’s work-stealing schedule for many workloads. The random scheduler achieves up to 1.4x speedup and at worst its performance is twice as slow. Second, they compare RuSt Dask Scheduler(RSDS), a scheduler they implemented in Rust that uses work-stealing, to Dask’s scheduler. In so doing they find that RSDS achieves up to 4x speedup and at worst throughput slightly decreases. In their final experiment, they compare RSDS with random scheduling to Dask’s scheduler. This experiment demonstrates the limits of Python as RSDS can outperform Dask’s scheduler in many cases despite using a random scheduler. Similar to the previous experiment RSDS reaches a maximum speedup of 4x while at worst being half as performant. They conclude that the performance gains of an intricate scheduler cannot be actualized if the underlying runtime system is inefficient.

Ray [11] is a distributed computing framework tailored towards AI applications. Since many AI applications are written in Python users interact with Ray using a Python API. Much like Parsl, Ray models computation with a dynamic task graph. In Ray, a task represents stateless computation while an actor represents stateful computation. Ray uses a *bottom-up* scheduler, each node has a local scheduler for its workers, who communicate with a global scheduler when necessary. When a task is created on a node, its local scheduler will attempt to assign a worker to that task. When a local scheduler is unable to assign the task to a local worker it will forward that task to the global scheduler. Using this scheduling system along with other components Ray achieves a million tasks/second at scale. Though Ray is accessed via its Python API, 72% of Ray’s System layer is written in C++. This shows that Python frameworks can achieve high throughput when their underlying system code is written in a performant language.

TaskBench [18] is a framework that measures the performance of parallel runtime systems that make use of task parallelism. One of the major contributions of TaskBench was Minimum Effective Task Granularity (METG). METG is a metric that quantifies the efficiency of a workflow system at a given task granularity and is parameterized by efficiency. Slaughter et al. apply this metric to OpenMP, OpenMP + MPI, Dask, OpenMP task, StarPU, Tensorflow, and more. They measure METG as a function of task granularity and find the threshold at which efficiency dips below 50%. Implicitly our work seeks to decrease METG for Parsl. We do not integrate Parsl into TaskBench, however, we measure the throughput of Parsl at different granularities with and without the optimizations we apply in later sections.

7. Limitations

DIREX imposes the greatest limitations on Parsl. DIREX sends tasks to workers using an interprocess queue between itself and its workers. In contrast, HTEx uses ZMQ sockets to communicate with managers, who ultimately use interprocess queues to send tasks and receive results from workers. The usage of interprocess queues instead of ZMQ

sockets disables DIREX from coordinating workflows across multiple nodes.

8. Conclusions and Future Work

In this work, we investigated and improved Parsl’s ability to manage fine-grained tasks. We first rigorously investigated how processes and task spend their time and then used the data we collected to motivate changes to Parsl’s execution pipeline. The profiling data we collected led us to believe that the semaphores on the manager-worker IPC were a bottleneck in throughput; however, we did not see a significant improvement in throughput after removing the semaphores. The optimizations motivated by tagging data (i.e., C-DFK and removing interchange) increased throughput. We conclude that while profiling data can be useful, for task-based parallel systems, where the journey of the task is most important for throughput tracking the individual task throughout a workload returns the most useful data.

This work lead us to conclude that the barrier to high throughput in Python-based parallel workflow systems is much closer to the throughput experienced by the system than first thought. Python is a high-level language that sacrifices performance for accessibility. Given that, one should expect Python frameworks to be less performant than C frameworks, however, we did not expect the barrier to be at our doorstep. Through rigorous exploration of Parsl’s execution pipeline, we found that trivial operations consume non-trivial amounts of time, leading us to move critical components of Parsl out of Python.

In the future, we plan to collect more scientific applications to further validate our findings. Given that machine learning inference tasks effectively have no dependencies and do not require I/O inside the worker, we seek applications that will place a greater strain on our systems.

References

- [1] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, “Falcon: a fast and light-weight task execution framework,” in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 1–12.
- [2] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, “The case for tiny tasks in compute clusters,” in *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, 2013.
- [3] C. Lee and J. Ousterhout, “Granular computing,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 149–154.
- [4] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, “Toward loosely coupled programming on petascale systems,” in *SC’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008, pp. 1–12.
- [5] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, “Design and evaluation of the gemtc framework for gpu-enabled many-task computing,” in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 153–164.
- [6] P. Nookala, P. Dinda, K. C. Hale, K. Chard, and I. Raicu, “Enabling extremely fine-grained parallelism via scalable concurrent queues on modern many-core architectures,” in *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2021, pp. 1–8.
- [7] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, “Sparrow: distributed, low latency scheduling,” in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 69–84.
- [8] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, “Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 361–378.
- [9] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy, “Efficient scheduling policies for {Microsecond-Scale} tasks,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1–18.
- [10] Dask Distributed, Available at <https://distributed.dask.org/en/latest/> Accessed 2023/05/02.
- [11] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, “Ray: A distributed framework for emerging {AI} applications,” in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 561–577.
- [12] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster *et al.*, “Parsl: Pervasive parallel programming in python,” in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 25–36.
- [13] A. Bauer, H. Pan, R. Chard, Y. Babuji, J. Bryan, D. Tiwari, I. Foster, and K. Chard, “The globus compute dataset: An open function-as-a-service dataset from the edge to the cloud,” *Future Generation Computer Systems*, vol. 153, pp. 558–574, 2024.
- [14] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, “Funcx: A federated function serving fabric for science,” in *Proceedings of the 29th International symposium on high-performance parallel and distributed computing*, 2020, pp. 65–76.
- [15] A. Alsaadi, L. Ward, A. Merzky, K. Chard, I. Foster, S. Jha, and M. Turilli, “Radical-Pilot and Parsl: Executing heterogeneous workflows on HPC platforms,” *arXiv preprint arXiv:2105.13185*, 2021.
- [16] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, “Flux: A next-generation resource management framework for large hpc centers,” in *2014 43rd International Conference on Parallel Processing Workshops*, 2014, pp. 9–17.
- [17] P. Bui, D. Rajan, B. Abdul-Wahid, J. A. Izaguirre, and D. Thain, “Work queue + python: A framework for scalable scientific ensemble applications,” in *Workshop on Python for High Performance and Scientific Computing*, 2011.
- [18] E. Slaughter, W. Wu, Y. Fu, L. Brandenburg, N. Garcia, W. Kautz, E. Marx, K. S. Morris, Q. Cao, G. Bosilca *et al.*, “Task bench: A parameterized benchmark for evaluating parallel runtime performance,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.
- [19] R. John, V. Hayot-Sasson, K. Chard, and I. Foster, “Navigating the molecular maze: A python-powered approach to virtual drug screening,” in *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, under review, p. Poster.
- [20] S. Böhm and J. Beránek, “Runtime vs scheduler: Analyzing dask’s overheads,” in *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2020, pp. 1–8.
- [21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: a system for large-scale machine learning,” in *Osdi*, vol. 16, no. 2016. Savannah, GA, USA, 2016, pp. 265–283.
- [22] L. Dagum and R. Menon, “Openmp: an industry standard api for shared-memory programming,” *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.