Dynamic Task Shaping for High Throughput Data Analysis Applications in High Energy Physics

Ben Tovar[1], Ben Lyons[1], Kelci Mohrman[2], Barry Sly-Delgado[1], Kevin Lannon[2], and Douglas Thain[1]

[1]Department of Computer Science and Engineering
University of Notre Dame
{btovar,blyons1,bslydelg,dthain}@nd.edu
[2]Department of Physics
University of Notre Dame
{kmohrman,klannon}@nd.edu

***Abstract*—Distributed data analysis frameworks are widely used for processing large datasets generated by instruments in scientific fields such as astronomy, genomics, and particle physics. Such frameworks partition petabyte-size datasets into chunks and execute many parallel tasks to search for common patterns, locate unusual signals, or compute aggregate properties. When well-configured, such frameworks make it easy to churn through large quantities of data on large clusters. However, configuring frameworks presents a challenge for end users, who must select a variety of parameters such as the blocking of the input data, the number of tasks, the resources allocated to each task, and the size of nodes on which they run. If poorly configured, the result may perform many orders of magnitude worse than optimal, or the application may even fail to make progress at all. Even if a good configuration is found through painstaking observations, the performance may change drastically when the input data or analysis kernel changes. This paper considers the problem of automatically configuring a data analysis application for high energy physics (TopEFT) built upon standard frameworks for physics analysis (Coffea) and distributed tasking (Work Queue). We observe the inherent variability within the application, demonstrate the problems of poor configuration, and then develop several techniques for automatically sizing tasks to meet goals of resource consumption, and overall application completion.**

I. Introduction

Large scale scientific instruments today routinely produce petabytes worth of data that must be searched, analyzed, and reduced in order to produce new insights. Distributed data analysis frameworks are a key tool to processing and understanding these large quantities of data. Typically, the end user writes a *processing function* that consumes a discrete unit of data in the dataset, whether a single image of the sky, a single particle collision event, or a single genome sequence. This function is given to the framework, which then arranges to divide the dataset into suitable pieces, apply the processing function to all of the items, distribute the work to many nodes of a cluster, and retrieve the results, and then summarize them in some way, typically with the help of a *reduction function* provided by the end user. This pattern is found with some variation in a wide variety of frameworks, such as Dask [6], Hadoop [19], Parsl [2], Spark [27], Work Queue [5], and others.

A central problem in the use of these frameworks is configuration. The end user is responsible for selecting a wide variety of parameters that affect the behavior and performance of the system: the number and size of cluster nodes to use, the blocking of data units into tasks, the quantity of resources (cores, memory, disk) needed by each task, the number of concurrent tasks of each type, and so on. Each of these parameters may have a drastic (and non-obvious) effect on the performance of the system. In this paper we focus on just two critical parameters: (1) the size of each task, measured in the number of data items that it consumes; and (2) the resources (cores and memory) assigned to each task. If the task size is chosen to be too small, performance may suffer as the overhead in the software stack for setting and managing tasks dominates the work actually done, and the scheduler may be overwhelmed with the sheer quantity of tasks. On the other hand, if the task size is too large, then tasks may fail to execute at all as they overwhelm the resources available at a node.

The typical advice given to end users is to adjust the controls manually for each run until a minimum is found. But this is not particularly helpful advice: once a workload is completed and the results obtained, the end user may have no reason to execute the same thing *again* for the sake of improving performance. Even if a given run is successful, little information is available to understand whether the configuration was near or far from optimal. Further, the next run of the framework may differ in essential ways: the next dataset may have different size and properties, the analysis kernel may change in ways that affect runtime and resources, or the system itself may have changed in scale or performance. This renders many approaches to autotuning [13] ineffective, because one run of the framework is unlikely to behave like another run. A more fine-grained, online approach is needed.

We explore this problem in the context of data analysis workflows for high energy physics. The Large Hadron Collider (LHC) produces petabytes of data each year via its four main experimental detectors – ALICE, ATLAS, CMS, LHCb – each

recording collisions between fundamental particles that may reveal new physical interactions. Coffea [21] is an emerging tool for performing late-stage analysis of high energy physics data using Python-native constructs. In its current form, Coffea performs a complete partitioning of a dataset into fixed size pieces and dispatches it to a distributed computing framework (e.g. Dask, Spark, Work Queue). This requires that the user select the framework configuration once prior to execution, and then either accept a complete execution or cancel it outright.

In this work, we develop a strategy for dynamic run-time task-shaping in Coffea. First, every task is executed under the care of a lightweight function monitor [14] that observes and enforces its resource consumption. Second, if the task is unable to execute within the assigned resources at all, then it is split into multiple tasks and reassigned. Third, the framework manager observes the time and resources taken by each task and incrementally builds a predictive model. Finally, the framework is re-worked to partition the input dataset incrementally on-demand, so that tasks may be of variable size. The manager chooses task sizes to achieve a performance policy, either for individual tasks or for the whole workload.

We have implemented this technique in the Coffea data analysis framework, making use of the Work Queue distributed execution system. Our implementation avoids the dual perils of too-small tasks making little progress and too-large tasks making no progress. We show that the dynamic approach converges quickly on effective task sizes and resource allocations. When considered in the context of an entire run, the automatic approach achieves performance very close to the ideal configuration selected statically. This technique has been deployed in the open-source Coffea framework and is now in production use by CMS data analysis teams.

II. Application Architecture

TopEFT Application. The TopEFT application is designed to analyze particle physics data collected by the CMS experiment [22] at the CERN LHC. Stretching 27km in circumference, the LHC is the largest and most powerful particle accelerator in the world. Two counter-rotating beams of protons are accelerated to nearly the speed of light before being made to cross at four interaction points around the accelerator ring, producing collisions approximately 40 million times per second. A particle detector is located at each interaction point; designed to measure the properties of the particles produced in the high-energy collisions, the detectors record approximately 90 petabytes of data per year. The CMS detector is one of the two general-purpose detectors at the LHC. A worldwide collaboration of more than 5000 physicists and engineers operate the CMS experiment and analyze data it collects in order to gain novel insights into the physics that governs fundamental particles and their interactions.

Some of these teams, including members of the CMS collaboration at the University of Notre Dame, the University of Nebraska, Ohio State University, the University of Oviedo, and the University of Zurich are searching for new physics that impacts interactions involving top quarks and heavy bosons, and has developed the TopEFT [3] application to perform this analysis. The analysis utilizes a flexible, model-independent mathematical framework known as effective field theory (EFT) to parametrize potential new physics effects associated with top quarks. Significantly heavier than all other known quarks, the top may have a unique relationship with yet-undiscovered phenomena; furthermore, interactions between the top quark and heavy bosons are very rare and difficult to produce. These elusive processes consequently represent an interesting probe of new physics at the energy frontier.

The inputs to the analysis are a set of data files containing information about each particle collision event to be analyzed. Comprising both real events recorded by the CMS detector and simulated collision events produced via Monte Carlo simulation, several billion collision events will be analyzed in total. The analysis workflow consists of reading the input data, processing the collision events to calculate relevant properties of the events, and producing output histograms with the relevant physics properties summarized. Though the number of collision events to be analyzed is large, each event is completely independent of the others, thus representing an optimal scenario in which to apply a parallel computational approach.

Coffea Framework. TopEFT is built upon the more general Coffea [21] framework for physics data analysis. A Coffea workflow is specified by a dataset to be analyzed, a *processor function*, and an *accumulator function*. In its original implementation, Coffea divides the dataset into static work units that have approximately the same number of events. The maximum number of events per work unit is called the *chunksize*, and it is a manual parameter provided by the user. As events are processed, their results are merged together by a second function, called the *accumulator function*. The accumulator function is fully commutative and associative, and so the results can be merged in any order, or in parallel via a reduction tree. There is also as part of the workflow, a preprocessing step in which metadata about the input is collected, but this step does not change across applications, and its only purpose is to collect metadata of the input files. Figure 2 shows the phases of a typical Coffea application. The preprocessing tasks are executed first, the bulk of the workflow consists of processing tasks, and accumulation tasks are created as processing tasks finish.

The memory resource is of special consideration in a TopEFT workflow. In order to exploit various vectorization optimizations, a processing function loads all events in a work unit simultaneously into memory, and the larger the work unit, the larger its memory usage. Further, for accumulator functions, the output of the TopEFT application is a collection of histogram-like data structures that summarize the relevant physics information about the full set of processed events. A trivial histogram would be quite small, consisting of a count of events at each scalar value. However, in the TopEFT approach, the weight of each signal event generated through Monte Carlo simulation is parameterized by an n-dimensional second-order polynomial, where n is the number of EFT parameters studied

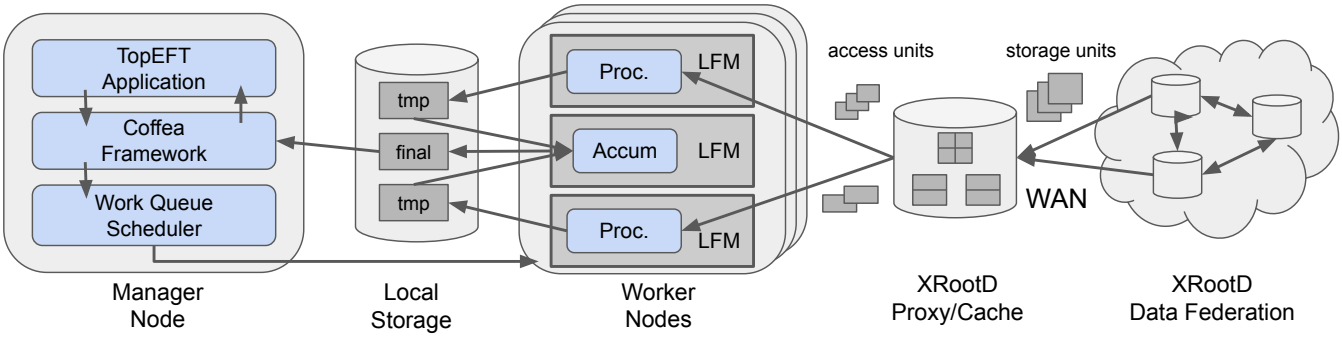


Fig. 1. Architecture of TopEFT Analysis Application. The TopEFT Application defines analysis and accumulator functions as native Python code, and passes them to the Coffea Framework, which arranges for the partitioning of the dataset into tasks. Tasks are given to Work Queue for scheduling to available workers, where they run under the control of lightweight function monitors. Data is delivered from the wide area XRootD federation to a local proxy, where it is accessed in smaller transfer units appropriate for each task. Temporary output is produced on local storage, and then summarized by accumulator functions to produce a final output, which is loaded back into the TopEFT application.

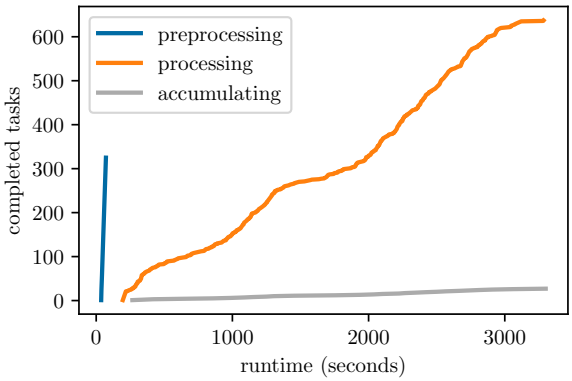


Fig. 2. Phases of a typical Coffea application using the Work Queue executor. One preprocessing task is created per input file, in which each file contains a number of events to process. Processing tasks are created by splitting the input files according to a given number of events, called the chunksize. As processing tasks finish, accumulation tasks are created that reduce the partial results into a final histogram.

in the analysis. TopEFT explores 26 EFT parameters, meaning that 378 quadratic fit coefficients are required to parameterize the 26-dimensional quadratic function for each event. Therefore, each bin of the histograms accumulated by TopEFT cannot be described by a single number; rather, each bin is described by the sum of the quadratic parameterizations of all of the events that fall into the given bin. These quadratically parameterized histograms are much larger than conventional histograms, and since the TopEFT processing tasks fill many such histograms, the memory usage in the accumulation stage can be a serious consideration.

Coffea applies the preprocessor, processor and accumulator functions using one of several provided *executor* modules. An executor is responsible for carrying out the execution of the workflow efficiently and reliably. A number of executor modules are provided with Coffea: a local executor simply spawns local threads on a single machine, while executors for distributed systems such as Spark, Dask, and Work Queue must arrange for data transfer, parallel execution, and the other issues that arise in distributed systems. Coffea defines each task to be run, in terms of the data to be processed and the functions to execute, and passes them to the underlying executor for completion.

Work Queue Executor. This work focuses on the use of Work Queue as the distributed execution system. Work Queue is a system for creating and managing scalable manager-worker style programs that scale up to tens of thousands of cores on clusters, clouds, and grids. A Work Queue manager accepts task definitions from Coffea and schedules the tasks to a fleet of remote shared-nothing workers. In this case, the tasks are Python functions, and so the manager must send to the worker the function itself, the function arguments, and perhaps the Python environment, and any other assets needed to execute. Each worker manages the resources available on a machine (e.g. cores, memory, disk) and will run as many tasks as can fit into those resources. For example, a 16-core worker could run two 4-core tasks and one 8-core task concurrently.

On the workers, each function invocation is executed within a *lightweight function monitor* (LFM) [14] to ensure that the function does not exceed its assigned resources. The LFM observes the utilization of cores, memory, disk, and other resources and reports these values back to the scheduler on completion. If the function should exceed its assigned resources, the task is terminated and returned to the scheduler for reconsideration, which is described below.

Dataflow. The total data managed by the CMS experiment is far too large for any one analysis site to store in its entirety, and so the data is distributed across a wide-area federation of universities, making use of the XRootD file service. A local site typically operates an XRootD proxy/cache which provides an interface to the system. Each task requests data from the federation, where it is divided into *storage units* of files approximately 1 to 2 GB each. Tasks may request any subset of this data from the proxy/cache, which will deliver

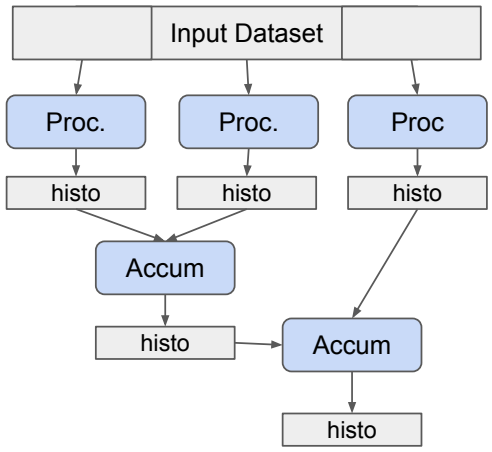


Fig. 3. Dataflow of TopEFT Application. The application writer provides two functions: a processing function is applied to an (arbitrary) partition of the input data, and produces an output histogram summarizing relevant properties. An accumulation function is used to perform a tree-reduce on multiple histograms and produces a final output.

only the portion of the requested file: these (*access units*) are correlated to the chunksize selected by the application. As processing tasks produce output histograms, these are returned to the manager node, and then consumed again by a tree of accumulator tasks, until the final output is produced, as shown in Figure 3.

III. The Configuration Challenge

In the default configuration, the end user of TopEFT must select two interacting configuration parameters: the *chunksize* of each task, and the *resources* allocated to each task.

The **chunksize** is the maximum number of events to be processed in each processing task. Coffea divides the number of events per file into the smallest equally sized number of work units such that no work unit has more than chunksize events. If the chunksize is chosen to be very small, then the total number of tasks will be large, creating opportunity for parallelism, but the scheduler will be dominated by the overheads of dispatching and managing tasks, while the proxy/cache will be overwhelmed by a large number of small file requests. If it is chosen to be very large, then the overall parallelism is reduced, and the runtime of outliers will dominate the overall execution time.

The **resources** are the quantity of cores, memory, and disk needed by each task. If the resources selected are too small, then tasks will fail as they exceed the allocation, and must be retried elsewhere with a larger allocation. On the other hand, if the resources selected are too large, then tasks are guaranteed to succeed, but concurrency at each worker will be reduced because fewer tasks will be packed into a given worker, and resources are left unused.

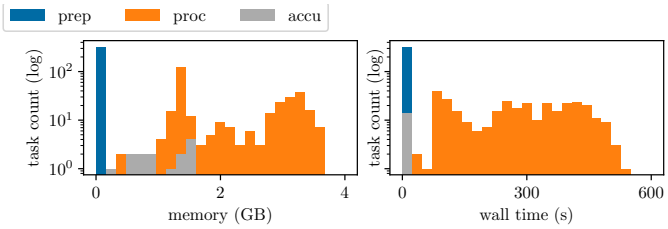


Fig. 4. Resources measured processing a whole file per task. The chunksize set so large that all events in a file are processed together. (a) Tasks memory distribution. (b) Tasks runtime distribution. These distributions show the opportunity for parallelism by judiciously shaping the tasks so that they require less time and memory so that the system can run more of them simultaneously.

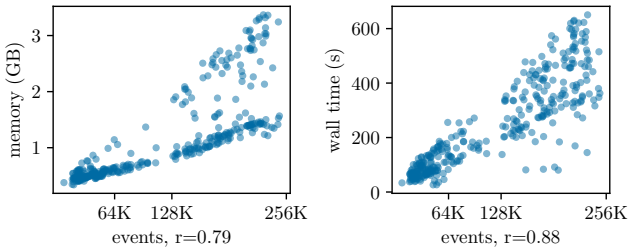
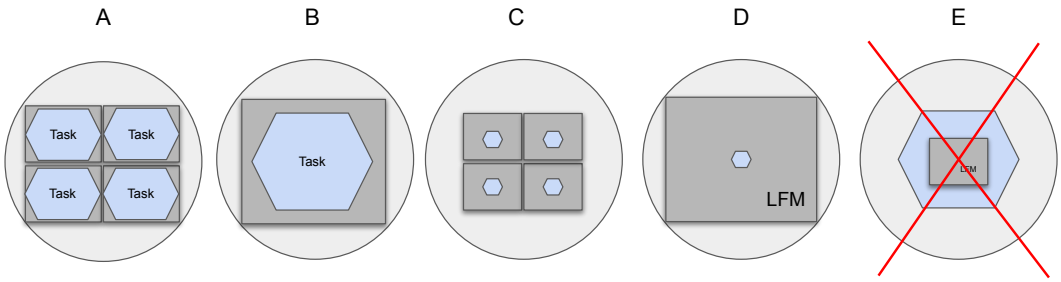


Fig. 5. Memory and wall time vs number of events per task. Chunksize chosen randomly for each task. In Coffea, all the events in a work unit are loaded simultaneously into memory. Even though the relationships are noisy, there is a strong correlation between resources needed (e.g. memory and compute time) and the number of events. We take advantage of this correlation to automatically find appropriate chunksizes given the resources available.

Figure 4 shows the inherent variation in the data. Using the existing Coffea implementation, we executed one task for each of 21 files in a standard TopEFT Monte Carlo signal samples dataset and observed the runtime and memory consumption using the LFM. The Monte Carlo datasets used for the studies in this paper consist of simulated proton-proton collision events, produced to simulate the geometry and conditions of the CMS detector during the 2017 and 2018 data-taking periods. While most tasks (note the log scale) consumed about 1.5GB RAM, there are many outliers ranging up to 4GB and down to 128MB. In a similar way, execution time varies from a few seconds to over 500 seconds. The problem is further complicated in two ways. First, there is a coupling between the number of events and the resources needed: larger tasks are likely to consume more memory, either by piling up larger histograms or by encountering more complex events. Second, the behavior of individual tasks can vary considerably because of the heterogeneous nature of the input data: physical events in the stream vary in complexity, as shown in Figure 5.

Figure 6 demonstrates the importance of an appropriately configured workflow. When there are few resources allocated per task and a very small chunksize, each task is fairly short but the overall workflow runtime increases because of the I/O associated with sending/receiving a task. When there are too few resources per task and a very large chunksize, some tasks may not be able to complete because they require more resources than they have been allocated. There are problems



Conf	Chunksize	Resources	Avg Task Runtime (s)	Total Tasks	Concurrent Tasks per Worker	Total Workflow Runtime (s)
A	128K	1 core, 4GB RAM	181.73	803	4	1066.49
B	512K	4 core, 8GB RAM	409.68	219	1	2674.87
C	1K	1 core, 2GB RAM	23.76	49784	4	9374.88
D	1K	4 core, 8GB RAM	20.91	49784	1	29350.68
E	512K	1 core, 2GB RAM	Failed	219	1	Failed

Fig. 6. Impact of bad configurations. The diagrams above indicate the packing of tasks into resource allocations and workers at various configurations, and the performance observed when running on 40 workers of 4 cores and 16GB RAM each. A: The optimal case with proper chunksize and resource allocation. B: High resources per task and large chunksize. This does not maximize concurrency. C: Low resources per task, low chunksize. Does not utilize full resources of the worker. D: High resources per task, low chunksize. Each worker can only fit one task and the task is small. E: Low resources per task, high chunksize. The task is too large so it cannot fit the resources allocated: the entire workflow fails.

with allocating too many resources per task as well. If coupled with a very small chunksize, the workers waste resources that could be used for other tasks. If the chunksize is too large, then each worker may only be able to run one large task as opposed to multiple tasks concurrently. Each of these poor configurations yields runtimes far worse than optimal.

For the end user who is interested in analysing physics data, the behaviour of turning these knobs is a distraction that sometimes may result in outright failure. We seek a more automated approach in which the end user starts a workload with minimal information, and the framework seeks its own configuration to achieve acceptable performance while avoiding disasters.

IV. Dynamic Configuration Techniques

The existing implementation of Coffea statically divides the events in each file into equally sized work units given the chunksize parameter. The chunksize dictates the maximum number of events per work unit. Each work unit is given a fixed resource allocation (cores, memory, disk), where both parameters are given manually by the user. We have developed a more dynamic implementation via the following techniques.

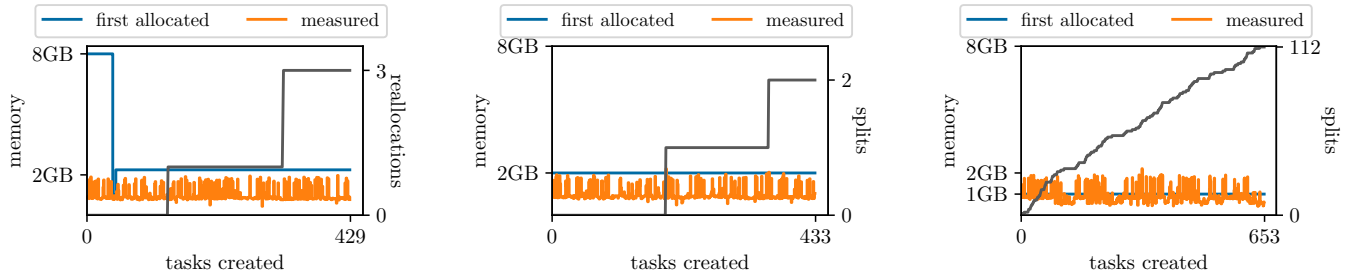
A. Measuring Task Resources

To start our discussion, we add two simplifying assumptions which we will later remove. First, we assume that the chunksize is fixed, and that the resources required of any task fit the resources available for execution. The objective here is to determine how many resources should be allocated per work unit to pack as many as possible in the resources available and increase parallelism.

To this end, we exploit the resource management capabilities already present in Work Queue. Workers measure how many cores, memory, and disk they have available and report this to the manager. In the steady state, tasks are labelled with the maximum resources that they expect to use. The manager then packs as many tasks per worker as resources allow. Each function evaluation on a worker is measured by the LFM, which continuously measures the resources used by each task[14]. If the task exceeds the assigned resources, then it is terminated and returned to the manager for reconsideration.

However, as the workflow starts, the resources needed for each task are unknown. The manager will conservatively assign a single task to use all the resources available at a worker, striving for task completion rather than task efficiency. As each task completes, the actual consumption is recorded at the manager. Once a threshold number of tasks (default 5) in a given category are completed, the manager begins to predict the resources needed for future tasks based on prior completions. If a task exceeds its predicted resource consumption, it is tried a second time using the conservative assumption of all resources on a worker. If that fails, it is tried yet again using the largest available worker, after which it is deemed to have permanently failed in its current shape. Note that since the runtime of tasks is measured in seconds to minutes, the cost/complexity of checkpointing tasks that exhaust resources so that they can be restarted in a larger allocation would likely exceed the benefit.

Work Queue may use strategies for predicting task resource consumption from prior behavior, including maximizing throughput, minimizing resource waste, or minimizing number of retries[23]. In general, minimizing number of retries works better for short running workflows (e.g. running less than 30 minutes) which consist of a few thousands tasks. Coffea, and thus TopEFT, are geared to interactive investiga-



(a) Updating allocations on exhaustion. (b) Splitting tasks on exhaustion (2GB). (c) Splitting tasks on exhaustion (1GB).

Fig. 7. Reallocating and splitting tasks. All measured values of tasks are shown in the order that tasks were created, rather than on completion. In (a), the allocation given to future tasks is adjusted as tasks complete and report their usage. Tasks with resource exhaustion are retried using the largest allocation possible (shown with a gray line). In (b) and (c) the allocation is fixed, and tasks that exhaust resources are split (shown with a gray line). Without updating task allocations (a) would run inefficiently, and without task splitting (b) and (c) would not complete at all.

tion, and therefore match this application profile. Work Queue minimizes task retries by keeping track of the largest resource measured and allocating this maximum when submitting new tasks to workers.

B. Splitting Failed Tasks

Once we have established resources for a given task size, we modified Coffea to change a task's size when needed. In particular, if a processing task should permanently fail due to resource exhaustion, the manager then splits the failed task by dividing it into two tasks, each with an equal number of events. These tasks are resubmitted to the queue, their resource consumption is predicted using the smaller number of events, and they eventually run on workers using smaller resource allocations. Tasks with unusually large resource consumption might end up being split multiple times before succeeding.

Splitting of processing tasks is safe because the computation performed on each event is independent, and therefore the computation of histograms is commutative. It doesn't matter what order events are analyzed in, since they will end up in the same histogram buckets regardless. However, this only applies to processing tasks. Preprocessing tasks cannot be split, because each one measure the metadata of one file. As for accumulation tasks, Work Queue executes them such that only the accumulated result and the next result to be accumulated are kept in memory. Therefore, should an accumulation task exhaust resources, it may be retried elsewhere with more resources, but cannot be split.

By default, tasks will only be split if they exhaust resources when running by themselves using a whole worker. However, maximum resources can also be set such that a task is split before they use a whole worker. This is useful when workers are run in large compute nodes with tens of cores, such as in HPC centers, or when an expert user has a precise idea of the resources needed per task.

C. Dynamically Sizing Tasks

While splitting tasks is an acceptable reactive strategy for dealing with failures, it would clearly be more efficient if tasks had the proper size to begin with. To allow this, we must structure the application so that the size of a task may change over the lifetime of a run. Instead of a priori splitting the whole workload into an array of tasks, we modified Coffea to construct tasks according to previous resource measurements. Based on observations such as shown in Figure 5, we take advantage of the strong correlation between the chunksize and the resources needed to complete a work unit successfully.

In the absence of any information, the first tasks are constructed using a chunksize guess that allows us to explore the relationship between resources consumed and the task size. Further workflow runs can run with a previously discovered chunksize size. As the workflow executes and we gather more data, we further refine the task sizes to match a given resource consumption. In the current implementation we use a linear progression, but more sophisticated methods are worth exploring. The space of the relationship between chunksize and resources used is sampled by taking advantage of the way Coffea constructs work units. Coffea almost never constructs work units with the given chunksize, as this only occurs if the number of events in a file is a multiple of it. Instead, the events in a file are partitioned in the smallest number of work units so that no group is larger than the given chunksize. With this, once we compute a chunksize c for a target resource usage, we eliminate noisy fluctuations by rounding down to the closest power of 2, $\tilde{c}$, and then randomly use $\tilde{c}$ or $\tilde{c} - 1$ to avoid the pathological case where all the files have a multiple of $\tilde{c}$ events.

V. Performance Evaluation

In this section we present the results of applying the different strategies presented in Section IV. For each of these experiments, we evaluate the performance of a TopEFT run that analyzes 219 files totalling 203 GB of data, 51 million events with 30 hours of total CPU consumption, resulting in 412 MB in the final (uncompressed) histogram output. These input files are live production data from the CMS experiment. We make use of 40 workers allocated from a shared university cluster, each one providing 4 cores and 8GB of RAM, for a total of 160 cores and 320GB RAM. This scale was chosen to correspond to a typical end-user configuration that offers substantial parallelism for real work, without requiring extraordinary efforts to request (and wait for)

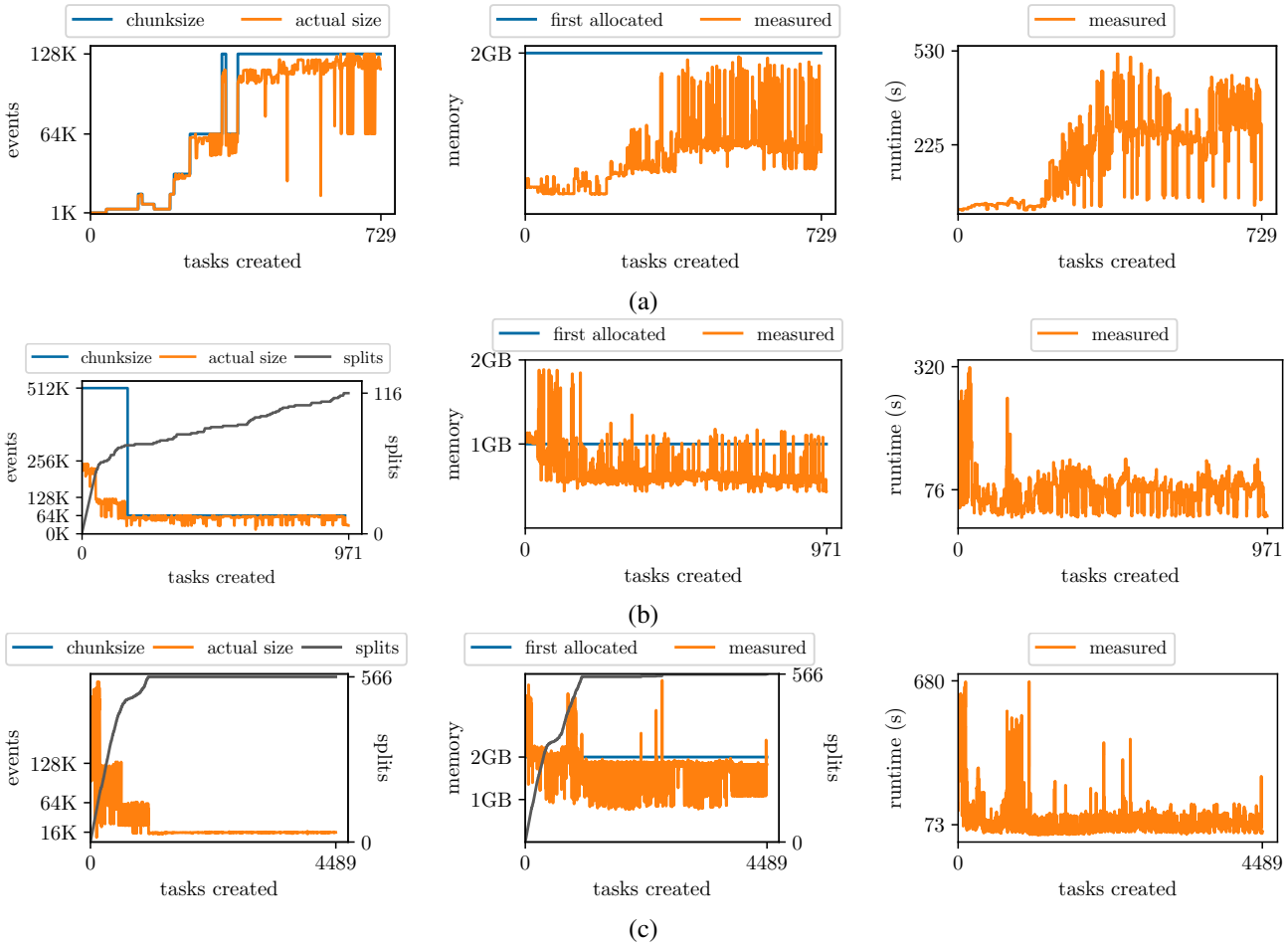


Fig. 8. Dynamic chunksize. All measured values of tasks are shown in the order that tasks were created, rather than on completion. In (a) we show the evolution of the dynamic chunksize, memory, and runtime when targeting a 2GB usage per task starting from a very small chunksize value running topEFT with the default parameters. In (b) the target usage is changed to 1GB, and the starting chunksize is set to too large, which causes all the first tasks submitted to be split up to three times (number of splits as tasks are created is shown in gray). Finally, in (c) for a target of 2GB per task, we show the effect of turning on one of the topEFT analysis option that greatly increases the memory consumption per task, with the corresponding decrease in the chunksize computed.

additional resources. While a real run would draw data over the wide area network, we have temporarily moved the input data for the run into a local Panasas shared filesystem, to eliminate performance variations due to congestion and uncoordinated outsider users.

The techniques described here have been incorporated to the stable versions, and have been tested with Coffea [21] commit fec5f645, topEFT [3] commit 8c64a6a8, and Work Queue [5] commit 9b1ed9db.

A. Dynamically Sizing Allocations

Our first result comes from just measuring the resources used per task, and adjusting the allocations accordingly. This is shown in Figure 7.(a) using a fixed chunksize of 128K events, and using 40 workers with 4 cores and 2GB of memory per core. Initially tasks are given all the resources of a worker, and as they return, future tasks are allocated the maximum value so far seen, 2.1GB plus some margin (e.g. round up to the next multiple of 250MB).

Since the memory requirement per task is very close to 2GB, ideally, we would wish to have each core to run a task in these 4 core 8GB workers, as this would divide the memory evenly among the cores. However, in this run the maximum memory value was 2.1GB, which just barely causes the concurrency per worker to be 3 instead of 4. We specify that a processing task cannot use more than 2GB to equally divide memory among the cores, and any task that goes above the limit is split and retried. An example run is shown in Figure 7.(b). Since there are two splits, this means that 4 more tasks were created as compared to Figure 7.(a).

The example in Figure 7.(b) shows a best case scenario where just a handful of splits lead to an efficient run. However, different worker configurations and workflows may produce a high number of splits. As we show in Figure 7.(c), tasks are limited to use less than 1GB each, which quickly increases the number of splits needed.

The previous examples assumed that the chunksize was fixed for the given resources. However, as we described in Section IV, an adequate chunksize depends both on the

workflow and the resources available. In Figure 8.(a), we show a run for adapting a very small chunksize (1K events) for our sample workflow to the previous configuration of 40 workers with 4 cores, 2GB per core. As tasks return and report their memory usage, the chunksize is updated to target a memory usage of 2GB for maximum concurrency per worker. Figure 8.(a) shows the evolution of the maximum chunksize, with the corresponding changes in memory and runtime per task. As before, tasks that would go above 2GB of memory would be split, but in this run that was not necessary. Note that even when the maximum chunksize stabilizes, the actual size of the work units varies greatly because different files contain different number of events.

B. Splitting Large Tasks

In Figure 8.(b) we show the effect of choosing a chunksize that is too large for the resources available. For this run, for processing tasks, we use 40 workers with only 1 core and 1GB of memory. These workers are too small to run accumulation tasks, and therefore an extra worker with 1 core and 2GB of memory was also deployed to process them, with Work Queue automatically sending accumulation tasks to it as needed. Setting the initial chunksize to 512K all of the first tasks submitted fail, and they are split until they complete under the memory constraint. In fact, task splitting completely dominates the initial part of the workflow, and no real work is completed until about 1/5 of the workload has been submitted, when a more adequate chunksize for the remaining tasks is used. Note that even when a chunksize has been found that achieves the target memory usage, the relationship between chunksize and memory is noisy, and some tasks are still split because they exceed the worker size. Further, considering all the execution time provided by the workers, 19% was lost in tasks that needed to be split, which indicates opportunities for improvement, such as a better initial chunksize guess from historical data.

Finally, in Figure 8.(c) we show the drastic effects in resource consumption that the different topEFT analysis options have. Simply turning one of this option, greatly increased the memory consumption per task, and for a target of 2GB, the chunksize found is only 16K. Given the stringent memory target and the large difference between the initial guess and the final chunksize, workers wasted 32% of the time in tasks that needed to be split. This example shows why we decided to compute the chunksize dynamically, rather than with an offline model, as there is wide variability in applications and the arguments used.

C. End to End Performance

In a production setting, it is rarely the case that the desired number of workers are instantly available. Instead, the cluster batch system may deliver a variable number of workers over time, depending on the offered load and other users. Figure 9 shows a run that highlights the resilience of TopEFT under these conditions. 10 workers arrive at first, followed by 40 more. All workers are removed around 1000s (perhaps a higher priority user preempted) and then 30 of them return a few minutes later, until the entire workflow is done. Note that the task allocation adjusts at several points early in the workflow.

Figure 10 shows the end-to-end performance of TopEFT across a varying number of workers, with ten runs at each point. The auto mode uses the previously described techniques to converge to the optimal resource allocation during a single run. The fixed mode runs with the optimal setting found from a previous run of the auto mode from the start. All configurations pull the environment from a shared filesystem. Generally, the total runtimes decrease as more workers are added. The curve eventually flattens out, which can be attributed to the load placed on the shared filesystem where the data is stored. The figure shows that dynamic task sizing and automatic resource allocation performs similarly to fixing the allocation to the best setting in advance.

Note that for Figure 10, given the overlap of the error bars, we do not conclude that the auto mode is ”faster”, simply that it is no worse than the fixed manual configuration. The variance in performance shown by the error bars has several contributions, primarily due to the non-determinism in a distributed systems: tasks can be scheduled in different orders, concurrent tasks may compete for network bandwidth, etc. Further, our tests were made at our university cluster where they competed for resources against other applications.

D. Delivering the Environment

When executing TopEFT tasks we need to ensure that the correct python environment is available on the remote resources. If a shared filesystem is available, then the environment can be configured in a location that all workers can access. In the more general case, however, we need to deploy the correct environment to workers that do not have a shared filesystem, and that cannot be configured ahead of time.

TopEFT was modified to automatically construct a tarball with the correct python environment. The tarball construction is done by installing required packages based on an environment specification[20], which is then transformed into a single file using `conda-pack` [1]. Once the environment file is ready, a wrapper script unpacks and activates it as needed. The environment created is 260MB compressed, 850MB uncompressed, and its activation takes around 10s.

We tested four methods of delivering the environment: 1) via the shared filesystem, 2) with a factory that creates workers that also run inside the wrapper script, 3) sending and setting up the environment with the first task that executes in a worker, and 4) setting up the environment per task. As we present in Figure 11, activating the environment once per task does noticeably worse than the other methods, but such a delivery method may still be useful for one-shot long running functions with special environment requirements. We found the per worker method to be most useful for rapid developing of TopEFT code, while the factory method to be more adequate for production runs as it minimizes data transfer to the workers.

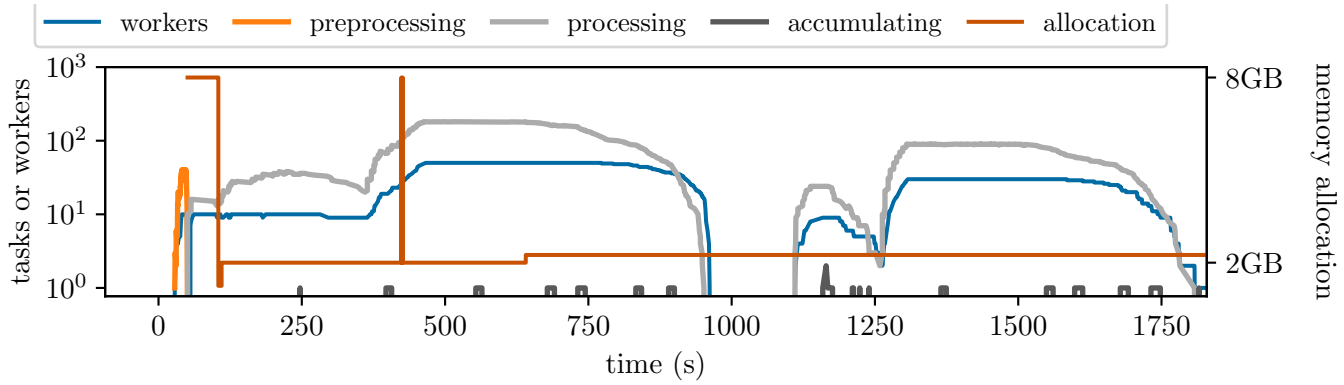


Fig. 9. Resilience to Dynamic Resources. Workflow starts with 10 4-core workers, and after a while 40 more workers connect. Around 1000s all workers go disconnect (perhaps because the opportunistic resources they ran on become unavailable). Later on 30 more workers connect to finish the workflow. Shown are the counts of currently executing tasks per category. On the right axis we show the memory allocation of processing tasks, which adjusts several times during the first half of the run.

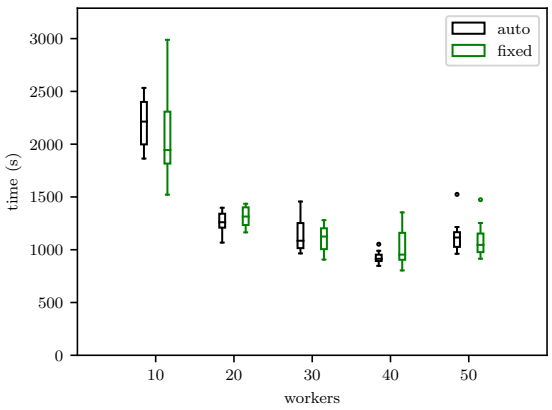


Fig. 10. Scalability of TopEFT in auto and fixed Modes

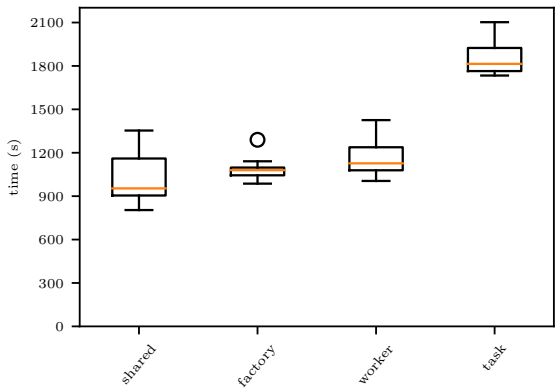


Fig. 11. Environment delivery modes.

VI. Related Work

A variety of approaches have been explored for auto-configuring distributed applications by predicting *end-to-end* performance metrics such as execution time and cost from *input features* such as data size and function properties. Hadoop [19] is a particularly popular target for such studies. [13]. For example AROMA [16] demonstrates autoconfiguration of Hadoop applications in the cloud by observing many prior runs, and then predicting that the current run will exhibit behavior similar to that of its neighbors in the label space. In a similar way, Morpheus [15] observes multiple runs of periodic data analysis jobs, considering the "skyline" of jobs running over time, in order to predict a in order to predict a future Server Level Objective (SLO) for that job. As noted above, this work does point out that user-defined-functions experience substantial churn over the course of a month, and so performance predictions must allow for code and resource evolution. Wang et al [24] compare the use of genetic algorithms (GA) and particle swarm optimization (PSO) for autotuning Spark applications; but once again are evaluating multiple runs of the entire MROnline [17] takes a different approach, focusing on a number of controls that can be adjusted for individual tasks within a run, however, these are limited to selecting the resources available for a given task, rather than reshaping the tasks themselves in order to meet the available resources. Our work differs from these prior approaches by performing *autotuning during the course of a single run* and taking advantage of the ability to resize individual tasks, change resource allocations for tasks, and deal with the unavoidable outliers in a heterogeneous application.

Many data analysis workflows are a special case of *malleable applications* [11], [12], which can scale up or down, even as they run, by adding or removing workers, taking into account observed performance or competition for resources. In this case, the individual tasks comprising the workflow are *moldable* [9], [26], meaning that their size can be chosen arbitrarily prior to (but not during) execution. The combination of a *malleable* workflow composed of *moldable* tasks gives a large number of degrees of freedom, which, depending on your view, make the application capable of executing almost anywhere, or make it difficult to configure.

Members of the CMS collaboration have developed generic tools for distributed analysis of physics data. For example, CRAB [4], and LOBSTER [25] have been used to execute workflows on thousands of cores in opportunistic resources. Unlike Coffea, in which the user analysis is written in Python assuming columnar data, they are mostly written in C++ and assume that each event is processed at a time.

The computing model used here is similar to map-reduce, however, unlike technologies such as MapReduce [7] or Hadoop [19], note that the computation is not moved to where the data is located, but rather, data comes from the repositories of the XRootD [8] federation. This allows to run workflows on generic resources without any previous setup, but does not take advantage of any data locality. To attack this problem, new approaches, such as ServiceX [10], aim to enable on-demand data delivery tailored for nearly-interactive vectorized analysis. In ServiceX, a user gives transformations (e.g. filters) to extract just the data required when querying events. These transformations are colocated with the data, which gives the potential of greatly reducing the amount of data transferred.

In the current implementation work units may only have events from the same file. As files vary in the number of events, this makes the size of the work units variable and the resource usage less uniform, which leads to a less efficient resource utilization. Approaches such as lazy arrays in uproot [18] (the backend that Coffea uses to read files from XRootD) or ServiceX [10], are promising alternatives for considering all the workload as a single stream of events that can be more uniformly partitioned.

VII. Conclusion and Future Work

In this paper we have demonstrated techniques for the automated shaping of both task size and resources allocated to tasks inside the run of a single application. This differs from prior auto-configuration approaches in that it applies to the execution of a single workflow, building up information from prior tasks, rather than relying upon the complete execution of prior workflows. Of course, this work has been deliberately limited to only two key parameters, and the problem becomes more challenging as the number of adjustable parameters increases: more samples are needed to build a suitable model, resulting in more time before effective actions can be taken. At some level, human design expertise is needed to identify the essential parameters to be turned over to automatic techniques.

Our technique relies upon the ability to construct (simple) performance models for tasks based on their input parameters. Ideally, every task behaves independently of the others. However, data delivery is an inherent bottleneck in this system: at large scales, task runtime will increase as a function of concurrency, due to competition for data bandwidth. We would like to close this loop and make the number of workers also a function of the network capacity as a resource. For example, if the bandwidth reported by tasks go below a given minimum, then the manager can reduce the number of concurrent tasks.

Looking forward, there are opportunities to study the three-way interactions between task sizing, task resources, and resource provisioning. In production, end users are confronted not only with the question of how to size tasks to the available resources, but also what resources to obtain: both university clusters and cloud providers offer machines of different capabilities. Should one acquire resources, and then configure the application to the resources? Or is it better to configure the application, and then acquire resources to meet it? It remains to be seen whether these high level approaches are equivalent, or even whether the solution space is convergent.

Acknowledgements

We thank Lindsey Gray and the Coffea developers team for their very helpful feedback and support in the implementation of the ideas presented in this paper. We also thank Jim Pivarski for his insights in uproot and IO performance. This work is supported by the National Science Foundation grant OAC-1931348.

References

[1] Anaconda Inc. conda-pack: 0.6.0. https://conda.github.io/conda-pack/.

[2] Y. N. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. T. Foster, M. Wilde, and K. Chard. Parsl: Pervasive Parallel Programming in Python. *CoRR*, abs/1905.02158, 2019.

[3] A. Basnet, K. Bloom, F. Canelli, S. S. Cruz, J. E. P. Cortezon, J. R. G. Fernández, A. T. Fernandez, R. Goldouzian, B. A. Gonzalez, M. Hildreth, K. Lannon, J. Lawrence, S. P. Liechti, C. E. Mcgrady, K. Mohrman, H. Nelson, B. Tovar, Y. Wan, A. Wightman, B. Winer, F. Yan, B. R. Yates, H. Yockey, and M. Zarucki. TopEFT/topcoffea: TopCoffea 0.1. https://doi.org/10.5281/zenodo.5258003, Aug. 2021. Source code: https://github.com/TopEFT/topcoffea.

[4] S. Belforte, I. Sfiligoi, J. Letts, F. Fanzago, M. D. Saiz Santos, and T. Martin. Using SSH as Portal - The CMS CRAB over glideinWMS Experience. Technical report, CERN, Geneva, Oct 2013.

[5] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011. Source code: https://github.com/cooperative-computing-lab/cctools.

[6] Dask Development Team. *Dask: Library for Dynamic Task Scheduling*, 2016.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[8] A. Dorigo, P. Elmer, F. Furano, and A. Hanushevsky. XROOTD - A Highly Scalable Architecture for Data Access. *WSEAS Transactions on Computers*, 4:348–353, 04 2005.

[9] I. Fujiwara, M. Tanaka, K. Taura, and K. Torisawa. Effectiveness of Moldable and Malleable Scheduling in Deep Learning Tasks. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 389–398. IEEE, 2018.

[10] Galewsky, B., Gardner, R., Gray, L., Neubauer, M., Pivarski, J., Proffitt, M., Vukotic, I., Watts, G., and Weinberg, M. ServiceX A Distributed, Caching, Columnar Data Delivery Service. *EPJ Web Conf.*, 245:04043, 2020.

[11] A. Gupta, B. Acun, O. Sarood, and L. V. Kal? Towards Realizing the Potential of Malleable Jobs. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014.

[12] N. Hazekamp, B. Tovar, and D. Thain. Dynamic Sizing of Continuously Divisible Jobs for Heterogeneous Resources. In *IEEE International Conference on e-Science*, 2019.

[13] H. Herodotou, Y. Chen, and J. Lu. A Survey on Automatic Parameter Tuning for Big Data Processing Systems. *ACM Comput. Surv.*, 53(2), Apr. 2020.

[14] G. Juve, B. Tovar, R. F. da Silva, D. Krol, D. Thain, E. Deelman, W. Allcock, and M. Livny. Practical Resource Monitoring for Robust High Throughput Computing. In *Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications at IEEE Cluster Computing*, 2015.

[15] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, Nov. 2016. USENIX Association.

[16] P. Lama and X. Zhou. AROMA: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. Association for Computing Machinery.

[17] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller. MRONLINE: MapReduce Online Performance Tuning. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, pages 165–176, New York, NY, USA, 2014. Association for Computing Machinery.

[18] J. Pivarski, P. Das, C. Burr, D. Smirnov, M. Feickert, T. Gal, L. Kreczko, N. Smith, N. Biederbeck, O. Shadura, M. Proffitt, benkrikler, H. Dembinski, H. Schreiner, J. Rembser, M. R., C. Gu, Edoardo, E. Rodrigues, JMSchoeffmann, J. Rübenach, L. Koch, M. Peresano, N. Eich, R. Turra, and bfis. scikit-hep/uproot3: 3.14.4. https://doi.org/10.5281/zenodo.4537826, Feb. 2021.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[20] B. Sly-Delgado, T. Shaffer, D. Simonetti, B. Tovar, and D. Thain. PONCHO: Lightweight Packaging for Distributed Python Applications. In *IPDPS (submitted for review)*, 2022.

[21] N. Smith, L. Gray, M. Cremonesi, B. Jayatilaka, O. Gutsche, A. Hall, K. Pedro, M. A. Flechas, A. Melo, S. Belforte, and J. Pivarski. Coffea - Columnar Object Framework For Effective Analysis. *CoRR*, abs/2008.12712, 2020.
Source code: https://github.com/CoffeaTeam/coffea.git.

[22] The CMS Collaboration. The CMS Experiment at the CERN LHC. *Journal of Instrumentation*, 3(08):S08004–S08004, aug 2008.

[23] B. Tovar, R. F. da Silva, G. Juve, E. Deelman, W. Allcock, D. Thain, and M. Livny. A Job Sizing Strategy for High-Throughput Scientific Workflows. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):240–253, 2018.

[24] Y. Wang, Q. Liu, J. Yu, and Z. Yu. An Experimental Comparison Between Genetic Algorithm and Particle Swarm Optimization in Spark Performance Tuning. In *Proceedings of the First Workshop on Emerging Technologies for Software-Defined and Reconfigurable Hardware-Accelerated Cloud Datacenters*, ETCD'17, New York, NY, USA, 2017. Association for Computing Machinery.

[25] M. Wolf, A. Woodard, W. Li, K. H. Anampa, Yannakopoulos, B. Tovar, P. Donnelly, P. Brenner, K. Lannon, M. Hildreth, and D. Thain. Opportunistic Computing with Lobster: Lessons Learned from Scaling up to 25K Non-Dedicated Cores. In *International Conference on Computing in High Energy Physics*, 2016.

[26] D. Ye, D. Z. Chen, and G. Zhang. Online Scheduling of Moldable Parallel Tasks. *Journal of Scheduling*, 21(6):647–654, 2018.

[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, Boston, MA, June 2010. USENIX Association.