Opportunistic Computing with Lobster: Lessons Learned from Scaling up to 25k Non-Dedicated Cores

Matthias Wolf, Anna Woodard, Wenzhao Li, Kenyi Hurtado Anampa, Anna Yannakopoulos, Benjamin Tovar, Patrick Donnelly, Paul Brenner, Kevin Lannon, Mike Hildreth, Douglas Thain

University of Notre Dame, Notre Dame, IN 46556, USA

E-mail: `{mwolf3|awoodard}@nd.edu`

Abstract. We previously described Lobster, a workflow management tool for exploiting volatile opportunistic computing resources for computation in HEP. We will discuss the various challenges that have been encountered while scaling up the simultaneous CPU core utilization and the software improvements required to overcome these challenges.

Categories: Workflows can now be divided into categories based on their required system resources. This allows the batch queueing system to optimize assignment of tasks to nodes with the appropriate capabilities. Within each category, limits can be specified for the number of running jobs to regulate the utilization of communication bandwidth. System resource specifications for a task category can now be modified while a project is running, avoiding the need to restart the project if resource requirements differ from the initial estimates. Lobster now implements time limits on each task category to voluntarily terminate tasks. This allows partially completed work to be recovered.

Workflow dependency specification: One workflow often requires data from other workflows as input. Rather than waiting for earlier workflows to be completed before beginning later ones, Lobster now allows dependent tasks to begin as soon as sufficient input data has accumulated.

Resource monitoring: Lobster utilizes a new capability in Work Queue to monitor the system resources each task requires in order to identify bottlenecks and optimally assign tasks.

The capability of the Lobster opportunistic workflow management system for HEP computation has been significantly increased. We have demonstrated efficient utilization of 25 000 non-dedicated cores and achieved a data input rate of 30 Gb/s and an output rate of 500 GB/h. This has required new capabilities in task categorization, workflow dependency specification, and resource monitoring.

1. Introduction

High Energy Physics (HEP) datasets are commonly divided into chunks for processing. Each chunk is then sent to a local cluster or the WLCG [4] for processing. The Compact Muon Solenoid (CMS) experiment [2] follows this strategy for processing on dedicated resources. However this approach proved problematic when used on opportunistic resources, as it required administrator privileges and favored long-running jobs that were likely to be evicted by other users with higher priority.

Lobster was developed to overcome these problems by providing a HEP workflow management environment optimized for use on opportunistic resources [9, 8]. A Lobster project begins with

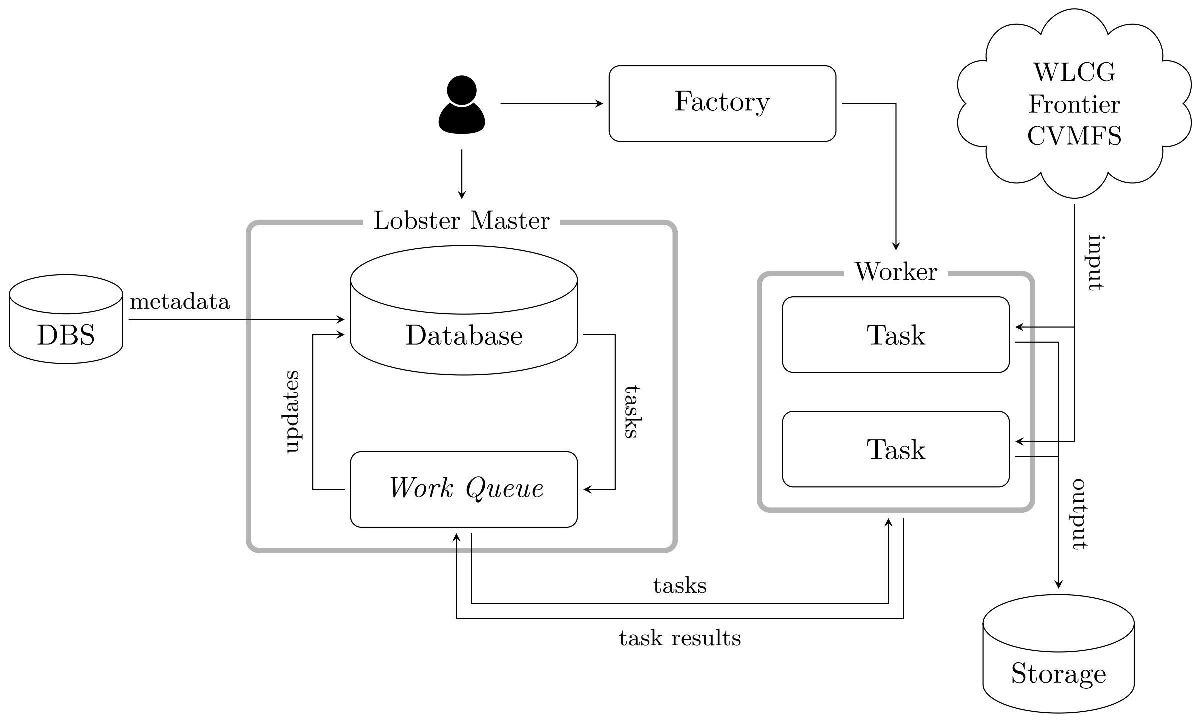

Figure 1. Overview of the Lobster service and task processing.

a configuration file prepared by the user, which defines one or more *workflows*, each specifying input data, working environments, executable commands and runtime parameters. The user then starts the main Lobster process, the master, which reads the specifications from the configuration file.

The division of a project into tasks begins by identifying smallest sets of input data that can be processed independently, referred to here as data *units*. A single input file need not be physically divided, but fractions of it may be assigned to different data units. The metadata defining the units is stored by the master process in a database.

The units are then grouped into *tasks* for processing. This grouping is done "on the fly", as resources become available over the course of the run. If the workflow is a simulation rather than the analysis of actual data, the metadata is generated internally by the master process and each data unit is processed as a separate task.

The Lobster process hands the prepared tasks to an internal *Work Queue* [5] master, which distributes the tasks and the files they require to *workers* running on a cluster. The user is responsible for submitting workers, or can initiate a *factory* process that handles submission automatically. Each worker process can run multiple tasks in parallel, by invoking the Lobster task wrapper, which sets up the execution environment. If the CMS software environment is not present, the wrapper will utilize parrot [7], a user-space tool that provides the required software via CVMFS [1]. This allows workers to run on machines that are not part of the WLCG or Open Science Grid [3] without requiring administrative access for software installation.

Predicting the processing time each task will require can be difficult, however choosing the right task size is essential in opportunistic distributed computing. If tasks are too small, system overhead increases because each task requires setup of the execution environment and transfers of large data files. If tasks are too large, run times become excessive and the eviction rate increases. Under Lobster, output metadata is continuously collected by Work Queue and returned to

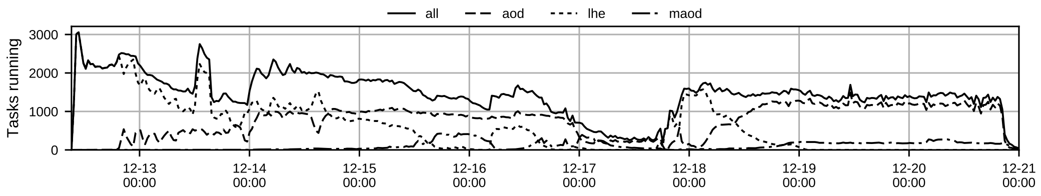

Figure 2. Example of a project using concurrent workflow processing. As tasks of the root step "lhe" finish, the "aod" step starts processing, producing results that are then consumed by the "maod" step. As each task of latter consumes the output of several "aod" tasks, the total and concurrent task counts are lower. Most of the fluctuation of the total number of tasks stems from stern competition over resources at the time this project was run.

the Lobster master, which uses the results to adjust task sizes dynamically and thus optimize throughput.

2. Workflow Dependency Specification

Event simulation and analysis in HEP is generally a multistep process. Often, the steps cannot be easily merged into a single workflow because each requires specific setup and configuration. Consequently, in the conventional approach, each workflow must be run to completion on the entire dataset before the next can be started.

We observed that this leads to two associated problems. Often most of the tasks in a workflow are completed quickly but some require a long time to complete, creating a "tail" which delays completion of the entire workflow even though only a small number of tasks are still being performed. Second, during this long tail much of the computational resource is unused.

In order to utilize these unused resources and improve throughput, we added the capability to process multiple workflows concurrently, even when the second requires output from the first. To accomplish this we construct an acyclic dependency graph, a specification that indicates how each workflow depends on prior steps. This graph, or set of dependencies is also stored in the Lobster database. Upon completion of each task, the output metadata is added to the table of available data units, and tasks in the subsequent workflow can begin as soon as enough data is available to create a new task.

An example of this behavior is shown in Figure 2, where three subsequent workflows are run: "lhe", "aod", and "maod". As the first "aod" tasks finish successfully, Lobster injects their output units into the database for the "aod" step, and subsequently starts to create new tasks to process them.

3. Categories

Under our original approach, all workflows used the same resource specification for each task. This was inefficient because different tasks often require different quantities of system resources. Some workflows, for example event generation, must be run single-threaded while others are more efficient with multiple CPUs, and memory requirements differ similarly, as shown in Figure 4. Allocating the same resources for every task would leave some machines underutilized and overload others.

To better allocate resources, we allow the user to divide workflows into categories and specify the resource requirements for each category individually. We also added the capability to limit the number of concurrent tasks, to avoid exhausting global system resources such as input bandwidth.

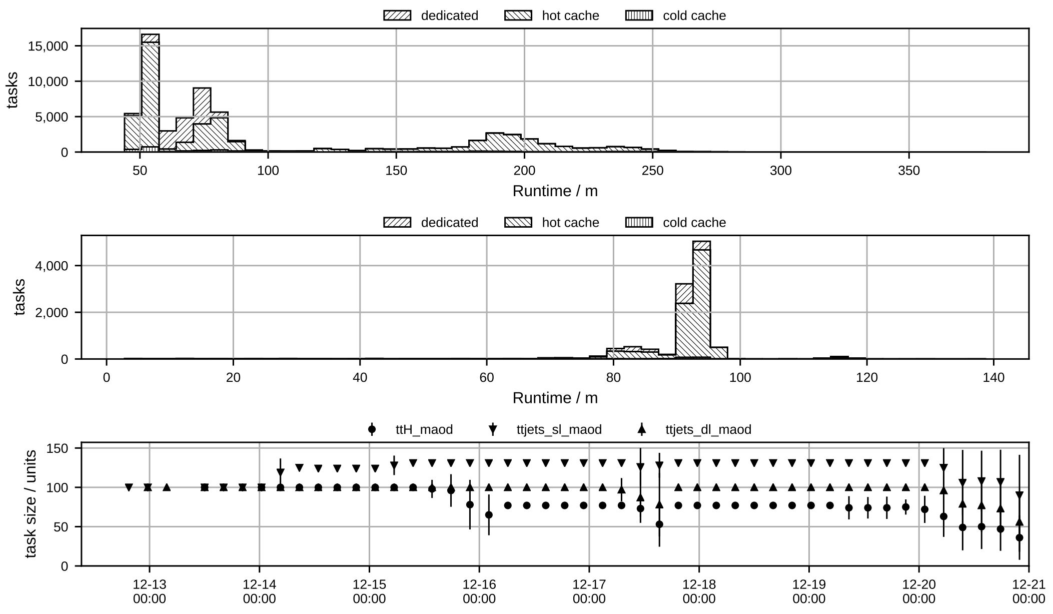

Figure 3. Runtime distributions for two categories and related task sizing. No runtime limitation has been placed on the workflows in the top category, while a limit of 90 minutes was used for the category shown in the middle, and was observed within the error margin granted to the tasks. The bottom plot shows the tasksize adjustment to optimally reach the desired task size. After an initial time period to gather statistics, the task sizes for the different workflows in the category diverge from their user-provided default. At the end, tapering of the tasksize to reduce the overall project runtime can be seen.

The resource requirements for each task are passed to Work Queue, which takes them into account when assigning tasks to workers for execution. This is important in the opportunistic computing environment as the available machines may vary considerably in capabilities.

Initial estimates of required resources may not be reliable, and external factors can also change and affect performance. To respond to these uncertainties, some of the category parameters are tunable during execution.

Finally, if runtime for a task is excessive it can result in eviction of the worker process from the machine by the cluster management system. This results in the loss of all work in progress on that machine. To minimize the risk of eviction, we use a feature of the CMS software framework to limit the maximum runtime for any task in a given category. If a task reaches the assigned time limit, it stages out all completed work and reports the unprocessed units to the master before exiting, which returns them to the pool of available tasks in the database. Figure 3 shows the effect of the runtime limitation and following task size adjustment.

4. Resource Monitoring

The use of categories allows the user to specify resource requirements, but they are still responsible for estimating what these requirements are. This can be difficult, given the rapidly evolving nature of both HEP software and the software environment. Overestimating the resources that will be used leads to underutilization, while underestimation can lead to memory exhaustion and excessive swapping, or eviction by the cluster manager. We use the Cooperative Computing Tools *resource monitor* [6] to make this process easier and more accurate.

The resource monitoring is tightly integrated into Work Queue, and is used to provide

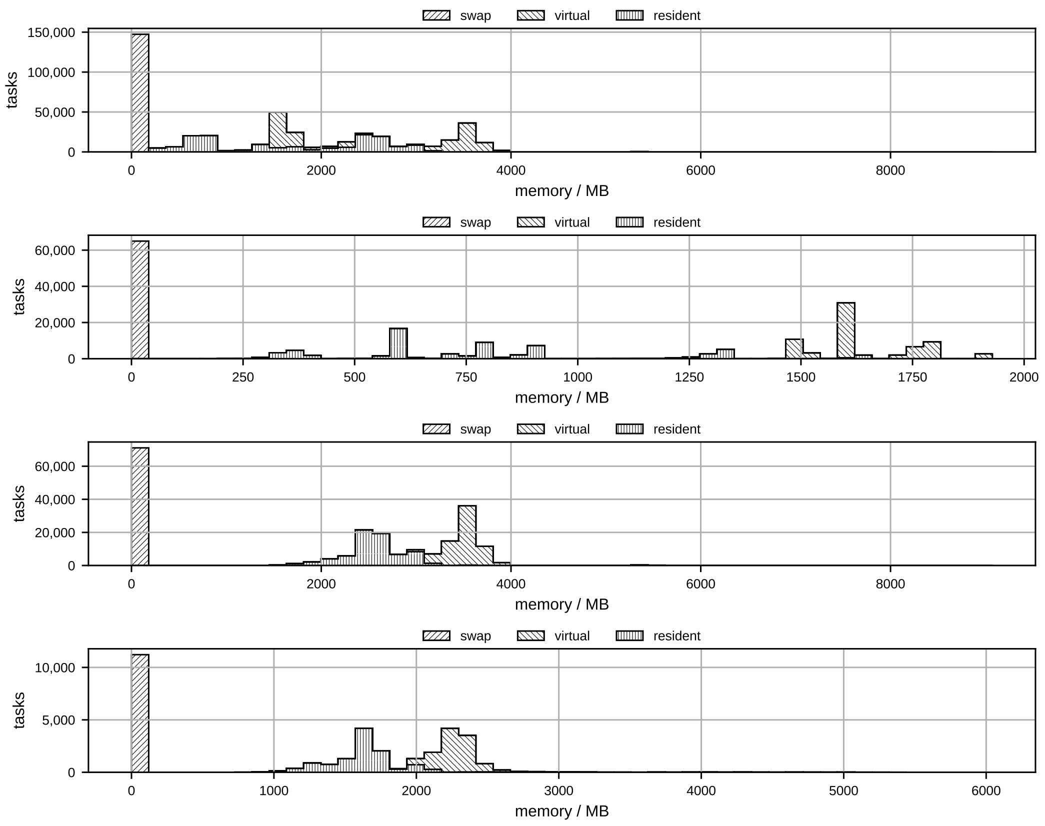

Figure 4. Memory consumption broken down into different categories. From top to bottom: all tasks, event generation (“lhe”), detector simulation and reconstruction (“aod”), and event content slimming (“maod”). Notice the different scales on the x-axes, the memory consumption varies greatly between the different categories.

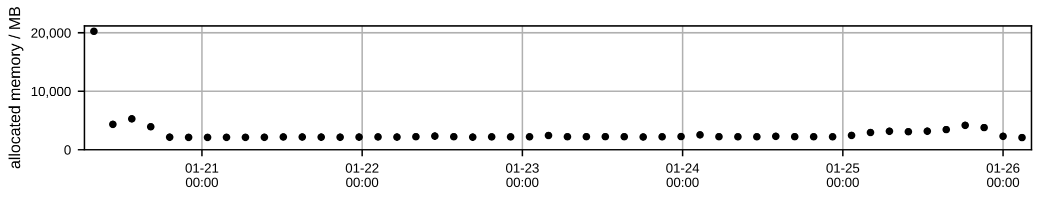

Figure 5. Evolution of memory allocation over time. The first tasks in this category occupy a whole worker before Work Queue receives enough statistics to assign an optimal memory allocation, which is then tuned as the project progresses.

another layer between the worker and the wrapper of a task. The Work Queue master may take an initial guess for resource consumption from the Lobster master if the user has provided one. Otherwise, it will start tasks consuming all resources of a worker to acquire resource measurements. All information about resource usage is sent back to the Work Queue master when a task finishes. While the task is executed, the resource monitor will constantly measure the resource consumption of the task and abort it if the allocation is exceeded. The master will restart such aborted tasks with a new, expanded allocation, until the task finishes successfully. Whenever the Lobster master creates a new task and passes it to the internal Work Queue

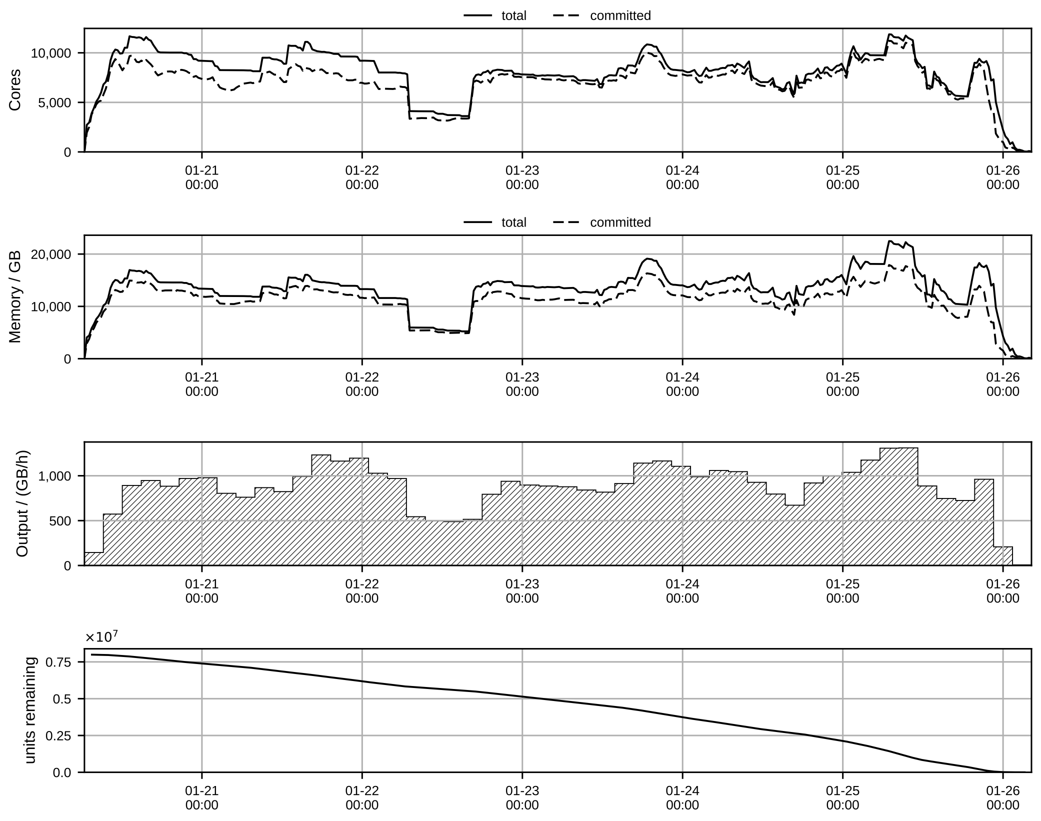

Figure 6. Six day processing run with $\mathcal{O}(10\,000)$ cores. Top two plots: number of cores and memory in use. This varies somewhat over the run because the workload manager is opportunistic and uses more cores and memory when they are available. Memory use closely parallels the number of cores allocated, but depending on the worker size, the allocation of one may be closer to the available capacity than the other. In this case, at the start of the project, workers with less memory were available, leaving some cores unoccupied, while for the second half, the memory allocation of the workers increased such that the available cores were used more efficiently. Middle: instantaneous output performance. A measure of the total output transferred back to the output storage element. This also includes output that will be merged by a later task, and thus deleted from the storage element, to reduce the amount of files and metadata to be handled. An output performace of more than 500 Gbit/s has been consistently achieved throughout the project runtime. Bottom: number of remaining work units. The work units were completed at a fairly steady pace over the six day period, with a small tail at the end of the processing period.

instance, it is assigned a new resource allocation. The initial resource measurement and subsequent adjustments can be seen in Figure 5. There are several different modes of the resource allocation, a *fixed* mode, where the user specified resource usage is taken unmodified, a *maximum* mode, using the maximal resource consumption of the task's assigned category, and modes *minimizing waste of resources* and *maximizing throughput*. For the latter two modes, Work Queue will use previous tasks to calculate resource quantiles that lead to either a minimum of task failures due to exhausted resources, and thus wasted computational efforts, or maximize the task throughput by more aggressive allocation.

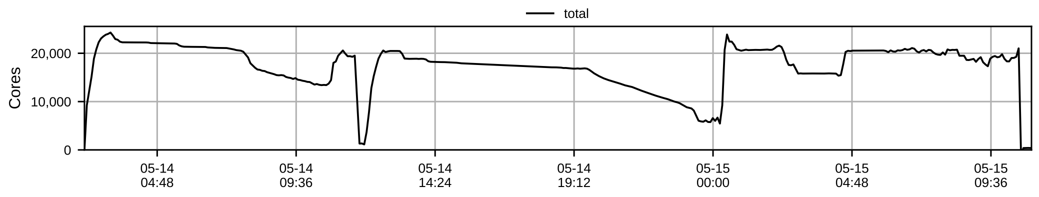

Figure 7. Number of cores used. This graph records our initial run of more than 20 000 cores. The number of utilized cores steadily declines due worker timeouts caused by excessive load on the master.

5. Performance

The aforementioned changes have allowed us to use resources at hand more efficiently and with less user intervention. Figure 6 shows core and memory usage, output performance, and unit completion from a typical six day processing run with $\mathcal{O}(10\,000)$ cores. The workload manager ran unattended for the entire period, opportunistically adapting to changes in the available resources and finally completing all the assigned tasks. This represents a typical project generating event simulation for private group use, in this case providing a training sample with larger statistics than available corresponding official samples.

While we have the ability to run on the scale of 10 000 cores throughout the year on the cluster available to us, we have been permitted to scale up to 25 000 cores during maintenance periods, with no opportunistic competition by other users for a short time. Results are shown in Figure 7, where we were able to scale to more than 20 000 cores, albeit not able to sustain peak core usage for prolonged periods of time. The Lobster master handles a large enough volume of completing tasks, starving the Work Queue master of time to keep all connections to workers alive. As a result, workers slowly time out until an equilibrium is reached. To alleviate this, all database transactions have been optimized in a first step to reduce the time that the Lobster master spends processing returned tasks, and thus allowing the Work Queue master more time to keep connections alive. At the same time, we are also investigating the use of Work Queue foremen to introduce a load-balancing between the Work Queue master and workers.

6. Conclusion

The capabilities of the Lobster workflow management tool for opportunistic computing in high energy physics has been significantly improved. Workflows can now be assigned to categories depending on their resource requirements, allowing for optimal assignment of tasks to nodes. A bottleneck in the analysis sequence caused by relatively long periods of inefficient running during processing tails has been circumvented by the addition of a new workflow dependency specification feature. Duties which previously had to be handled by the user, such as estimating resource requirements and monitoring system performance, are gradually being automated. These refinements have made it possible to scale the system up and opportunistically utilize as many as 25 000 cores.

References

[1] CernVM File System, `http://cernvm.cern.ch/portal/filesystem`.

[2] CMS Experiment Public web site, `http://cms.web.cern.ch`.

[3] Open Science Grid, `https://www.opensciencegrid.org`.

[4] Worldwide LHC Computing Grid, `http://wlcg.web.cern.ch`.

[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.

[6] 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.

[7] D. Thain and M. Livny. Parrot: Transparent User-Level Middleware for Data Intensive Computing. In *Workshop on Adaptive Grid Middleware at PACT*, 2003.

[8] A. Woodard, M. Wolf, C. Mueller, N. Valls, B. Tovar, P. Donnelly, P. Ivie, K. H. Anampa, P. Brenner, D. Thain, K. Lannon, and M. Hildreth. Scaling Data Intensive Physics Applications to 10k Cores on Non-Dedicated Clusters with Lobster. In *IEEE Conference on Cluster Computing*, 2015.

[9] A. Woodard, M. Wolf, C. N. Mueller, B. Tovar, P. Donnelly, K. H. Anampa, P. Brenner, K. Lannon, and M. Hildreth. Exploiting Volatile Opportunistic Computing Resources with Lobster. In *Computing in High Energy Physics*, 2015.