E-HPC: A Library for Elastic Resource Management in HPC Environments*

William Fox[1], Devarshi Goshal[2], Abel Souza[3], Gonzalo P. Rodrigo[2], and Lavanya Ramakrishnan[2]

[1]School of Computer Science
Georgia Institute of Technology, Atlanta, Georgia
wfox7@gatech.edu

[3]Department of Computing Science
Umeå University, Sweden
abel@cs.umu.se

[2]Lawrence Berkeley National Laboratory, Berkeley, California
{dghoshal,gprodrigoalvarez,lramakrishnan}@lbl.gov

Abstract: Next-generation data-intensive scientific workflows need to support streaming and real-time applications with dynamic resource needs on high performance computing (HPC) platforms. The static resource allocation model on current HPC systems that was designed for monolithic MPI applications is insufficient to support the elastic resource needs of current and future workflows. In this paper, we discuss the design, implementation and evaluation of Elastic-HPC (E-HPC), an elastic framework for managing resources for scientific workflows on current HPC systems. E-HPC considers a resource slot for a workflow as an elastic window that might map to different physical resources over the duration of a workflow. Our framework uses checkpoint-restart as the underlying mechanism to migrate workflow execution across the dynamic window of resources. E-HPC provides the foundation necessary to enable dynamic resource allocation of HPC resources that are needed for streaming and real-time workflows. E-HPC has negligible overhead beyond the cost of checkpointing. Additionally, E-HPC results in decreased turnaround time of workflows compared to traditional model of resource allocation for workflows, where resources are allocated per stage of the workflow. Our evaluation shows that E-HPC improves core hour utilization for common workflow resource use patterns and provides an effective framework for elastic expansion of resources for applications with dynamic resource needs.

Key words: Elastic resource management, scientific workflows, HPC systems

*The paper has been re-typeset to match the thesis style. Reproduced with permission of ACM.

1 Introduction

Today, scientific workflows with experiment data are increasingly processed on High Performance Computing (HPC) systems. Next-generation scientific workflows have to support streaming data and real-time constraints with varying resource needs. Today, HPC platforms are primarily designed to support monolithic MPI applications and provide a static resource allocation model i.e., a resource allocation is fixed for the duration of the entire job. The static resource allocation model presents challenges to scientific workflows where every stage of the workflow may have different resource needs. Current allocation methods attempt to either split the workflow and incur queue wait times for each stage, or request one big allocation resulting in wastage of resources.

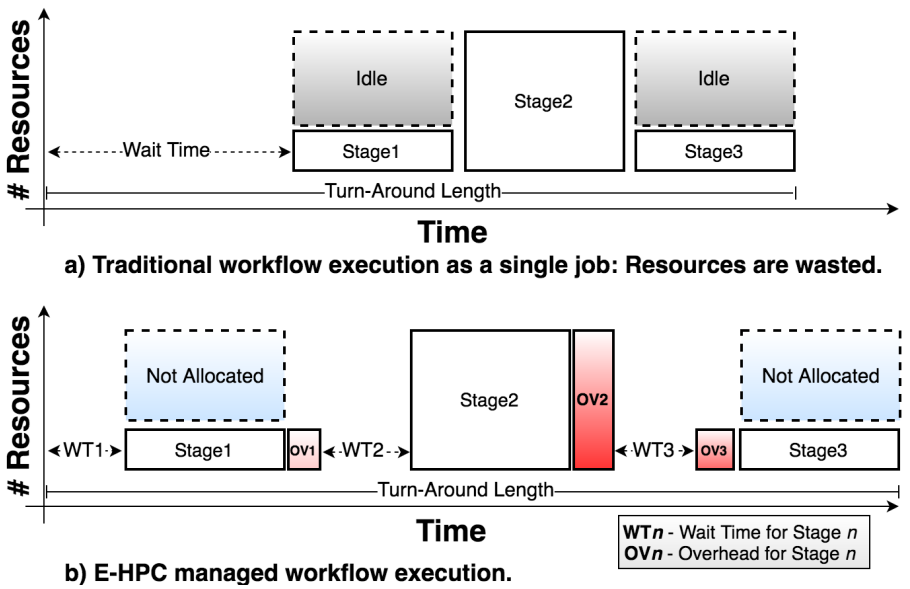


Figure 1: Comparison of traditional and E-HPC managed resource allocation for scientific workflows in HPC: a) shows the static allocation of resources for the entire duration of the workflow execution. b) shows the dynamic allocation where resources are requested as per the needs of a particular stage.

Current methods result in loss of efficiency or utilization, and the problems will only be exacerbated with next-generation dynamic workflows. We need a dynamic resource management model that considers resources to be elastic that can grow and shrink based on requirements of a scientific workflow. Resource elasticity has been extensively studied in the context of clouds [Sha+11; GB12; PMS16]. However, unlike HPC environments, cloud resources are not managed through batch schedulers. Elastic resource management in HPC environments has also been explored for specific applications [GHB04], but general methods are still not available.

In this paper, we present **Elastic-HPC** (**E-HPC**), an elastic framework for managing resources for scientific workflows in an HPC environment. It provides dynamic,

adaptable resource management and supports workflow execution. E-HPC is capable of growing and shrinking the allocated resources for a workflow during execution. It considers the resource slot to be an elastic window that maps to different physical resources based on availability. E-HPC uses checkpoint-restart mechanism to save and relaunch workflows on different resources when needed. Users can either submit a workflow description to E-HPC, or instrument existing workflows to use E-HPC to manage elastic resources. Thus, E-HPC can fit into current software ecosystems available in scientific collaboration or at HPC facilities. In this paper, we make two contributions.

- We design and implement an elastic framework to manage resources for scientific workflows in HPC environments.
- We evaluate the performance of the E-HPC framework across two HPC systems, and several synthetic and real scientific workflows to understand the performance of workflows and overheads in E-HPC.

The rest of the paper is organized as follows. In section 2, we describe background on workflows and current methods of resource management. We describe the design and implementation of E-HPC framework in Section 3 and results of our evaluation in Section 5. We present related work in Section 7 and conclusions in Section 8.

2 Background

Figure 1a shows an example of the resource allocation while running workflows on HPC resources in the absence of E-HPC. Traditionally, workflows are executed as a single batch job, where a set of HPC resources are allocated for the entire duration of the workflow. Workflows with different job resource requirements often end up wasting resources. Alternatively, users have to manually manage the different stages in the workflow by packaging them as separate jobs. In contrast, E-HPC provides elasticity for running workflows on HPC resources. Figure 1b shows the resource allocationusing E-HPC. E-HPC allocates only as many resources as required by a specific stage of the workflow. In case of asynchronous parallel tasks, E-HPC is able to also grow or shrink during run-time. E-HPC manages elasticity during workflow execution by dynamically resizing the allocated resources using checkpoint and restart.

2.1 Scientific Workflow Execution

Workflows are used to describe and execute tasks according to their data and control dependencies. Today, HPC workflows are implemented through ad-hoc scripts and workflow tools [Dee+05; Wil+11] with limited support for elastic resource management. Scientific workflows in HPC are run in static allocations as chained jobs (jobs with dependencies) or pilot jobs (the entire workflow contained in a job) [Rod+17]. Chained workflows have very long and unpredictable turnaround times because their critical path includes intermediate wait times. Pilot job workflows, i.e., the traditional method

in Figure 1a, typically has shorter turnaround times because there is no wait time between jobs. However, pilot jobs allocate the maximum resource set required by any task within the workflow, even if other tasks require significantly less resources, leading to resource wastage. E-HPC facilitates workflows to consider resource slots as dynamic elastic resources. E-HPC is able to eliminate resource wastage and job exceeding its allocation by dynamically adapting a workflow's resource allocation to its tasks resource requirements.

2.2 Elasticity and recovery use cases

Scientific workflows require elasticity and recovery. However, there are different requirements on how and when they occur. In this section, we outline these requirements in the context of E-HPC.

Scaling Between and During Stage Execution. Workflows are composed of stages with fixed but different resource requirements. or workflows where resource requirements are discovered during execution [KP11]. E-HPC is capable of scaling workflow resources between and during stages execution that support both cases.

Elasticity Triggers. Resource requirements of a workflow, might be known beforehand, discovered during its execution, or connected to external events (e.g., execution deadline) [YB05]. E-HPC supports all three cases. For known resource requirements, users may define an execution plan that will guide resource scaling operations (Section 3.1). Users can express new resource needs for workflows using the E-HPC API that will trigger a scaling process to support changes in resource requirements. The API can also be used from outside the workflow for externally driven events.

Recovery from Failures. Scientific workflows are composed of multiple long running stages and work is lost in case of failure. E-HPC transparently performs periodic checkpointing and recovery of the workflow. E-HPC is capable of successfully managing workflows where runtime might be unknown in advance [Sou15] or workflow failure might occur because of exceeding job boundaries on batch systems. In this case, using E-HPC, when a workflow job reaches its time limit, it can be checkpointed and restarted.

2.3 Tigres Workflow Library

We use Tigres, a workflow library to evaluate E-HPC. Tigres [Hen+16] is a programming library that allows users to compose large-scale scientific workflows and execute them on HPC environments. Workflows can be composed from existing executable scripts and binaries or new Python code. Tigres workflows are Python programs that are submitted as jobs to the batch scheduler directly. It manages workflow execution from within the job scripts, where resources are managed through different batch schedulers.

Tigres provides *"templates"* that enable scientists to easily compose, run, and manage computational tasks as workflows. These templates define common computation patterns used in processing and analyzing data and running scientific simulations. Currently, Tigres supports four basic templates to model serial (sequence), and concurrent (parallel, split and merge) execution. During serial execution, although Tigres launches

a task on a single compute node, it can expand to other compute nodes using MPI or other distributed libraries. In concurrent stages, Tigres manages task parallelism and employs SSH to deploy workers on each available compute node. E-HPC tracks and checkpoints distributed processes in applications, such as Tigres, that rely on SSH and MPI to support elasticity.

2.4 DMTCP

DMTCP (Distributed MultiThreaded Checkpointing) is a transparent user-level checkpoint restart library for distributed applications [RAC06] used in E-HPC. DMTCP is transparent because no changes are required to the *managed application* or the underlying operating system. The DMTCP managed application state is captured by monitoring system calls, processes, network communication, and open files. Also, DMTCP supports distributed applications by intercepting the creation of new local or remote (SSH or MPI) process. Instances of DMTCP in these new processes connect through sockets to a central daemon that coordinates the distributed checkpointing and restart operations.

DMTCP freezes the execution of all the processes of the application at checkpoint time. Then, for each process, a copy of its memory allocation, network, and I/O states are dumped to the file system. During restart, a process is created for every process checkpointed and its memory and state are restored from the copy in the file system. E-HPC takes advantage of DMTCP to shrink or enlarge resource allocations of an HPC workflow. DMTCP will restart an application on different hosts, even if their number differs, as long as the hosts are homogeneous (typical in HPC systems) and a mapping of processes over resources is provided (calculated by E-HPC).

3 Design and Implementation

Figure 2 shows the architecture of E-HPC. E-HPC has three main components - E-HPC Application Programming Interface (API), E-HPC Coordinator, and E-HPC Tracker. The E-HPC API provides the elastic functions that allows users to manage their resources dynamically. The E-HPC Coordinator accepts traditional user workflows and interacts with the E-HPC API to provide the elasticity needs. The E-HPC tracker tracks the execution of the workflow on the resources and manages the book keeping associated with the elastic resources.

A user submits a workflow by employing E-HPC command-line utilities on the HPC system. Next, E-HPC starts its coordinator, and generates and submits a batch job script that drives the workflow execution. Once the batch scheduler allocates resources to the job, the script executes the workflow and initiates services needed in the compute nodes to support elasticity. These services include a DMTCP controller, responsible for managing low level distributed checkpoint-restart operations, and an E-HPC tracker, which monitors the different states of the workflow. In this context, if the workflow requires resource scaling, it employs the E-HPC API to issue an elastic request for increasing or decreasing the number of resources allocated to the job. The E-HPC

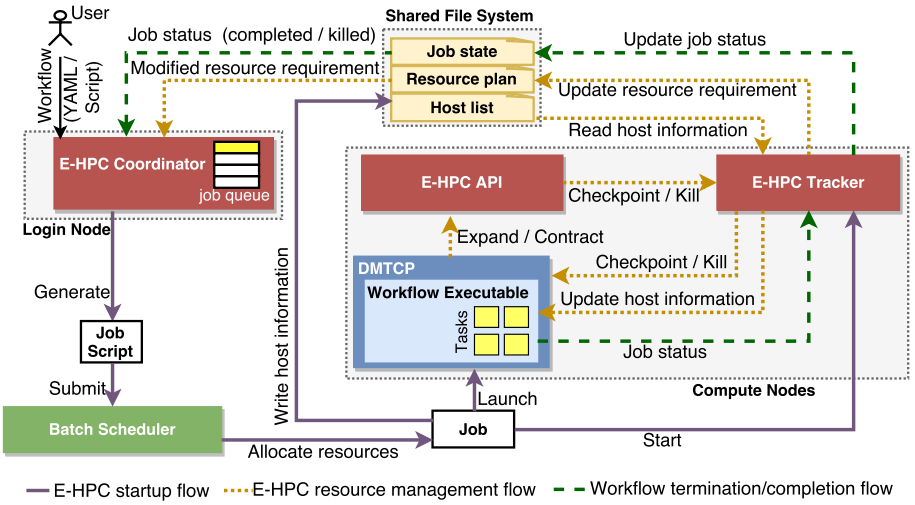


Figure 2: E-HPC architecture. E-HPC has three distinct control flows – a) *startup*: Users submit a workflow through E-HPC. E-HPC coordinator generates job scripts to request resources and run the workflow. Once the resources are allocated to the job through the batch scheduler, E-HPC launches the workflow executable via DMTCP. b) *resource management*: E-HPC tracker monitors the job execution and the states of the workflow. When the workflow issues an expand or a contract request through the E-HPC API, the tracker sends a checkpoint/kill signal to DMTCP. c) *termination/completion*: Once the job is killed, E-HPC coordinator resubmits another job script with a modified resource requirement. If the job completes successfully, E-HPC tracker notifies the coordinator.

API relays the request to the E-HPC tracker which issues checkpoint and kill requests to DMTCP. After the checkpoint is complete, and all the tasks are killed, E-HPC tracker informs the coordinator of the new resource requirements and ends the current workflow job. E-HPC coordinator generates a new job script that invokes DMTCP to restart the workflow using the checkpoint.

3.1 User Interface

E-HPC is designed as a library and users can interact with it in two different ways – a) plan-driven, or b) event-driven. In the plan-driven method, the user provides the elasticity plan, whereas in the event-driven method, external and internal events in the application trigger elasticity. Figure 3 provides an example of an elasticity plan that specifies the stages and resource requests for a workflow. The elasticity plan specifies the stages of the workflow and the associated resource requests. The resource requests in the plan specify the amount of resources and the duration for which the resources are required. The plan-driven method is minimally invasive, and requires no change to existing application programs or scripts.

The event-driven method allows users to modify their scripts using the E-HPC API, which induces elasticity from within the application program. This allows users to

```
workflow:
  command: python stage1.py
  stage1:
    nodes: 4
    ppn: 1
    walltime: 00:15:00
  stage2:
    nodes: 1
    walltime: 00:30:00
```

Figure 3: Workflow descriptions contain execution commands and resource requirements for each stage.

scale up or down conditionally based on an event in application characteristics and on resource requirements. We describe the API in detail in Section 3.5.

In addition to the two different ways to interact with E-HPC, a user can also request elastic resources in two different ways – i) **stage elasticity**, where a user can opt to grow or shrink resources between the stages of a workflow, or ii) **runtime elasticity**, where a user can request to change resources at any time during the execution. Stage elasticity is useful when different stages of the workflow have pre-defined, known resource requirements. On the other hand, runtime elasticity is useful when resource requirements can be changed due to external factors (e.g., available resources, system failures). However, runtime elasticity may not be suitable for all applications. For example, applications that are checkpointed in the middle of an I/O operation may end up in an unknown or unpredictable state.

3.2 Workflow States

Figure 4 shows different states a workflow may go through when using E-HPC. A workflow is at first in a *pending* state and is put in the queue that the E-HPC coordinator manages internally. Once the required resources are allocated for the job, it moves to the *running* state, where the tasks are distributed and executed on HPC resources. If the resource requirements change during the execution of the workflow, or the execution terminates due to failure, the workflow state moves to a *checkpoint* state. The workflow moves from the checkpoint state to the pending state, waiting for the

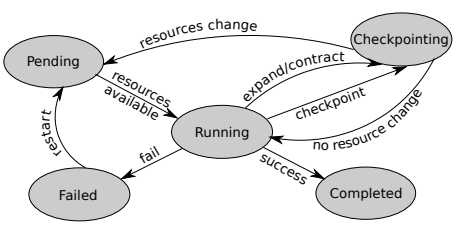


Figure 4: State transitions of a workflow in E-HPC.

new set of resources for the restart to happen. If the workflow execution terminates due to a failure, E-HPC restarts the failed workflow using the checkpoint. The workflow transitions to *completed* state when all the tasks of a workflow are executed successfully.

3.3 E-HPC Coordinator

E-HPC coordinator generates job scripts for running workflows on HPC resources, and coordinates the execution and dynamic resource allocation for workflows. When a user submits a workflow to E-HPC, it starts the coordinator on the login node of an HPC system. Users specify the initial resources required to execute the workflow and E-HPC coordinator generates a job script for running the workflow on HPC resources. The resource requirements are transformed into batch scheduler directives in the job script. The job script acts as a wrapper for launching the workflow tasks through DMTCP. E-HPC coordinator sets up all the environment variables and directory paths for enabling distributed checkpointing through DMTCP. DMTCP requires a centralized coordinator for checkpointing and restarting distributed tasks, which the job script launches on a compute node.

The state of the workflow tasks and the changes in resources are shared with E-HPC coordinator through the shared file system. E-HPC coordinator uses job status and resource requirements to determine the elasticity of workflows. If the job fails, terminates before completion, or the resource requirements change, E-HPC coordinator checkpoints the workflow and generates a restart command to launch the unfinished workflow through a job script. The job script also sets up the required commands to update the host list, once new resources are allocated to the job.

3.4 E-HPC Tracker

The job script generated by E-HPC coordinator starts a tracker on a compute node. Once the workflow job begins to execute, the tracker monitors the job progress. The workflow requests a change in resources through the E-HPC API. The API triggers the E-HPC tracker to checkpoint the workflow and terminate execution. In cases where the workflow is specified through a workflow description, E-HPC tracker checkpoints the execution prior to executing the next stage in the workflow. The workflow tasks run across multiple nodes, and E-HPC tracker waits and monitors all the tasks to be correctly checkpointed and terminated before updating the job state. Once all the tasks of the workflow have terminated, E-HPC tracker updates the resource plan to describe the number of resources required for the rest of the workflow and the expected time of execution. If there is no change in the resource requirements of a workflow, E-HPC tracker simply monitors and checkpoints the tasks as the execution progresses.

During a restart, E-HPC tracker reads the updated host file and sends a signal to the workflow in execution. It sends an updated host list along with the workflow so that new tasks can be launched on a different set of resources. Every workflow needs to be able to trap the signal to notify changes to an executing workflow. In our current implementation, E-HPC sends a SIGUSR1 signal to share the updated information

about the hosts. This provides a generic interface for any workflow or application to dynamically scale on a different set of HPC resources based on the requirements.

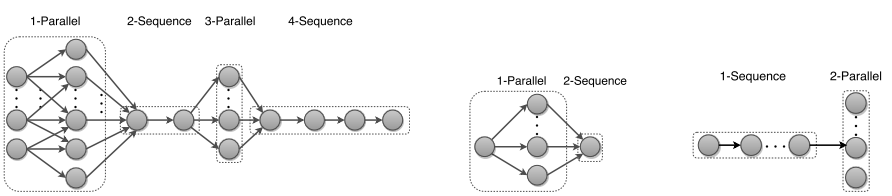
3.5 E-HPC API

E-HPC provides command-line utilities and an API to manage elastic workflows on HPC resources (Table 1). Users submit a workflow through the command-line as: `ehpc start <workflow-script>`. The `start` command triggers the launch of a workflow through the E-HPC coordinator. It generates job submission scripts and submits them through a batch scheduler. It ensures that a workflow script is launched through DMTCP and hence, is checkpointed and can be restarted as needed.

Interface	Description
init	registers a workflow to E-HPC and returns a workflow-id
expand	grows the resource allocation to the amount specified
contract	shrinks the resource allocation by the amount specified
done	notifies E-HPC about workflow completion

Table 1: E-HPC API provides four interfaces to transform a simple workflow into an elastic workflow.

Users can use the E-HPC API to add elastic calls in a workflow script to grow or shrink resources as the workflow executes. The E-HPC API provides four simple functions to manage elasticity in a workflow. The `init` function registers a workflow through E-HPC coordinator and returns a unique identifier for the workflow. The `expand` function is used to dynamically grow the number of resources for a workflow in execution. Similarly, the `contract` function allows for dynamically shrinking the number of resources for an executing workflow. Both `expand` and `contract` checkpoint, kill and restart the workflow on a different set of resources based on the resource requirements of a workflow. Once a workflow completes execution, allocated resources are released and checkpoint data is removed using `done`, which updates the status of the workflow to completed. E-HPC also calculates and collects performance and usage metrics for a workflow.



(a) Montage. (b) BLAST. (c) Synthetic.

Figure 5: Workflow graphs, stages are logically combined. Stages are combined together according to workflow resource requirements: subsequent parallel stages are combined together as a single logical stage and sequential stages are combined into a single sequential stage

3.6 Minimizing Queue Wait Time

The default mode of operation in E-HPC is to submit a workflow job with the new resource allocation request, only after the previous job has been terminated. This may incur large queue wait times depending on the current workload of the system. In order to minimize the queue wait times, E-HPC provides a **fast execution mode**, where the next stage of the workflow is submitted to the job queue, while the execution of the workflow continues in its current resource allocation. Once the requested resources are allocated to the job, the running workflow is signaled to be checkpointed and killed, and the placeholder job restarts the workflow using the newly created checkpoint. This eliminates the queue wait time for applications that can dynamically scale up or down.

Unlike the default E-HPC method, the fast mode requires all proceeding execution to be checkpoint-safe in order to function as expected. We define an execution as checkpoint-safe, if there are no active TCP connections, or I/O operations. In such cases, a restart may result in data loss or even failure to re-establish the TCP connections. This is a limitation of the DMTCP library and we plan to investigate alternative strategies in future to overcome this limitation.

The efficiency of the fast operation depends on the queue wait time of the placeholder job, the time required to finish the current job, and the checkpoint/restart overhead. If a job's runtime is small, the overhead of queue wait time and the cost of migrating to new resources might not justify the use of fast mode.

3.7 Workflow Plug-ins

E-HPC is implemented in Python and generates job scripts to be run on HPC resources through batch schedulers. It currently supports Slurm and Torque schedulers. Users specify the resource requirements and E-HPC generates batch scripts with the respective scheduler directives. Workflow status and resource requirements are updated through a YAML file on a shared file system. E-HPC is currently integrated with the Tigres workflow library and can be used with any Tigres workflow. However, workflow scripts written in Python can also be transformed into elastic workflows by using the E-HPC API directly. E-HPC currently uses DMTCP for checkpoint restart. However, the architecture of E-HPC is independent of DMTCP and other checkpoint restart mechanisms might be used. Thus, E-HPC has been designed such that it can be extended to work with other batch queue, workflow and checkpoint restart systems.

4 Formulating Performance Metrics

In this section, we formulate the performance goals for E-HPC. Specifically, E-HPC is targeted to improve the core-hours usage for a workflow, without increasing the overall turnaround time of the workflow. Mathematically, we define core-hours (C) as:

$$C = t * n$$

where, t is the execution time in hours, and n is the number of cores assigned during execution. Hence for a workflow with s stages, core-hours usage is calculated as the sum of core-hours used by each stage:

$$C = \sum_{i=1}^{s} t_i * n \tag{1}$$

Now, a workflow managed by E-HPC allocates separate resources to each stage of the workflow. Hence, each stage is assigned the number of cores required for its execution. In addition, the checkpoint-restart mechanism incurs an overhead that adds to the execution time of the stages. So, core-hours usage for an E-HPC managed workflow is given by:

$$C_{ehpc} = \sum_{i=1}^{s} (t_i + CR_i) * n_i \tag{2}$$

where, CR_i represents the checkpoint-restart overhead for the i-th stage. Using equations 1 and 2, we can derive that E-HPC minimizes core-hours usage for a workflow iff:

$$\sum_{i=1}^{s} CR_i * n_i < \sum_{i=1}^{s} (n - n_i) * t_i \tag{3}$$

Equation 3 shows that in order to provide better core-hours, checkpoint-restart overheads must be *less than* the core-hours gained for the workflow execution. In other words, the total amount of core-hours gained in workflow execution has to be more than the amount of core-hours lost due to E-HPC checkpoint-restart overheads. The result implies that the core-hours improvement in E-HPC is dependent on a number of external factors like the characteristics of the workflow, the performance of the checkpoint-restart library, and the I/O performance of the storage system on which the checkpoint images are written.

In addition to minimizing the core-hours usage, E-HPC also targets improving the total turnaround time of a workflow. The turnaround time (T) of a workflow is defined as:

$$T = t + q \tag{4}$$

where, t is the execution time, and q is the time to wait in the queue for getting the requested resource allocation. Since E-HPC allocates resources separately to each stage of the workflow, the turnaround time in E-HPC is:

$$T_{ehpc} = \sum_{i=1}^{s} (t_i + q_i + CR_i) \tag{5}$$

where, t_i, q_i and CR_i are respectively the execution, queue-wait and checkpoint-restart times for the i-th stage of the workflow. Using equations 4 and 5, we can derive that the turnaround time in E-HPC is better if:

$$\begin{aligned} \sum_{i=1}^{s} (t_i + q_i + CR_i) &< t + q \\ &< \left(\sum_{i=1}^{s} t_i \right) + q \end{aligned} \tag{6}$$

Assuming the checkpoint-restart overhead is negligible i.e., $CR_i \approx 0$, equation 6 implies:

$$\sum_{i=1}^{s} q_i < q \tag{7}$$

Equation 7 shows that E-HPC improves the turnaround time if the sum of individual queue wait times for each stage of the workflow is less than the queue wait time of the complete workflow. In other words, E-HPC achieves better turnaround time if smaller jobs (stages of a workflow) have significantly less wait time than larger jobs (the complete workflow). But as the queue wait time is a system parameter, controlled by the resource scheduler, E-HPC cannot control it. However, to minimize the queue wait times for each stage of the workflow, E-HPC provides a fast execution mode as described in Section 3.6.

5 Evaluation

In this section, we evaluate the performance and resource usage of scientific workflows through E-HPC. We compare our results against running the workflows without E-HPC (i.e., they are submitted through a single large job – see Figure 1a).

5.1 Systems

We evaluated the impact of elasticity on HPC workflows through E-HPC on two systems – i) Gordon and ii) Cori. Gordon [Gor15] is a dedicated XSEDE cluster with 1024 compute nodes. Each compute node contains two 8-core 2.6 GHz Intel EM64T Xeon E5 (Sandy Bridge) processors and 64 GB of DDR3 RAM. The file system is Lustre with a peak I/O bandwidth of 100 GB/s and the resources are managed by TORQUE. Cori [Cor16] is a Cray XC40 supercomputer hosted at the National Energy Research Scientific Computing Center (NERSC), which has 2388 compute nodes, each with two sockets and 32-core Intel Xeon "Haswell" processor at 2.3 GHz per socket and 128 GB DDR4 memory (2133 MHz, four 16 GB DIMMs per socket). The file system used during job execution is a Lustre file system with a peak performance of 700 GB/s.

5.2 Workflows

We use two real science workflows (Montage and BLAST) and one synthetic workflow to evaluate E-HPC (Figure 5). All the workflows are built using the Tigres templates. The different stages in the workflows are logically grouped together into parallel and sequential stages based on the resource requirements at each stage of the workflow.

Montage [Jac+09] is an I/O intensive workload [Juv+13] that constructs a JPEG image from sky survey data formatted as Flexible Image Transport System (FITS) files [Pen+10]. As shown in Figure 5a, Montage is composed of nine stages, and we logically group them into four stages – i) *1-Parallel*, ii) *2-Sequence*, iii) *3-Parallel*, and iv) *4-Sequence*. All experimental runs of Montage construct the image for survey *M17* on *band j* and degree 8.0 from *2mass* Atlas images.

BLAST is a memory-intensive workflow that matches DNA sequences against a large sequence database ($>$ 6 GB). The workflow splits an input file (a few KBs) into several small files and then uses parallel tasks to compare the input against the large sequence database. The database is loaded in-memory on all the compute nodes during the parallel stage. Finally, all the outputs from the parallel stage are merged into a single file. As shown in Figure 5b, BLAST is composed of three stages. As the first stage runtime is short, we logically group them into two stages – i) *1-Parallel* and ii) *2-Sequence*. BLAST is used to illustrate the resource usage for an use-case where a parallel stage execution time is substantially larger than the sequential one.

Synthetic workflow is composed of sequence and a parallel stages (Figure 5c). The workflow is written in Python. The memory-intensive version of the synthetic workflow consists of tasks that do a large number of memory allocations for over one billion integers, prior to calculating the values of their sum and multiplication. The first stage contains one billion tasks, calculating the sum in sequence, whereas the second parallel stage contains ten million tasks, calculating the multiplication in parallel. In contrast to the other two workflows, this workflow is designed to have a longer sequential stage, followed by a shorter parallel stage. We also use the memory-intensive version of the synthetic workflow that consists of 10 thousand parallel tasks. Unless otherwise specified, we use the two stage memory-intensive synthetic workflow for our evaluation, and use the single stage memory-intensive fully parallel workflow for measuring E-HPC overheads.

Table 2 lists and summarizes the metrics for our experiments. It includes workflow runtime, stage execution time, stage checkpoint and restart times, process kill times, core-hours used and queue time (inter-stage queue wait time). The runtime of a workflow is calculated as the time between the execution start of the first stage and completion of the last stage of the workflow. The core-hours measured correspond to the resource allocated for the entire duration, including possibly resources left unused by a workflow. Wait time values are not included since jobs do not consume core-hours when they wait. In our evaluation, we use E-HPC's regular mode, unless otherwise specified.

Metric (unit)	Description
Stage execution time(s)	Execution time for a workflow stage
Workflow runtime(s)	Workflow end time - Workflow start time
Checkpoint time(s)	Time to checkpoint a stage
Restart time(s)	Time to restart a workflow stage
Queue time(s)	Time a workflow stage waits in queue prior to execution
Core-hours used(hrs)	∑ Task execution time * Number of cores allocated

Table 2: Metrics for evaluation.

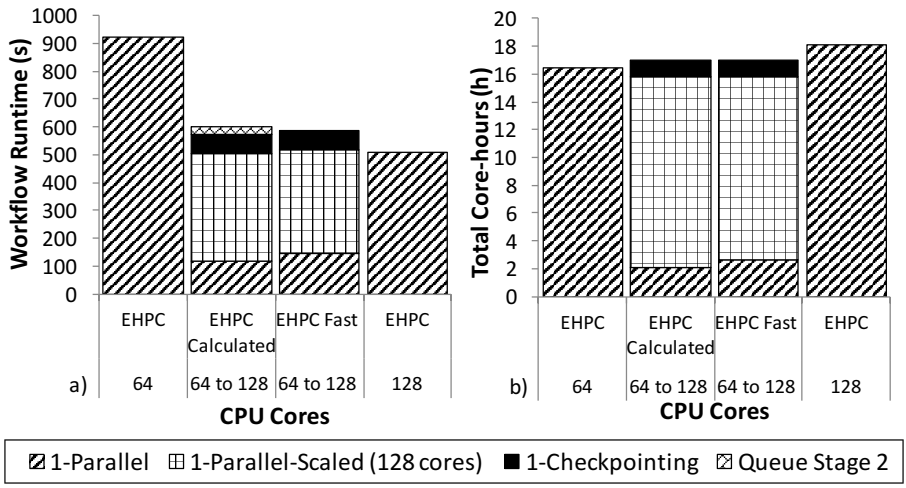
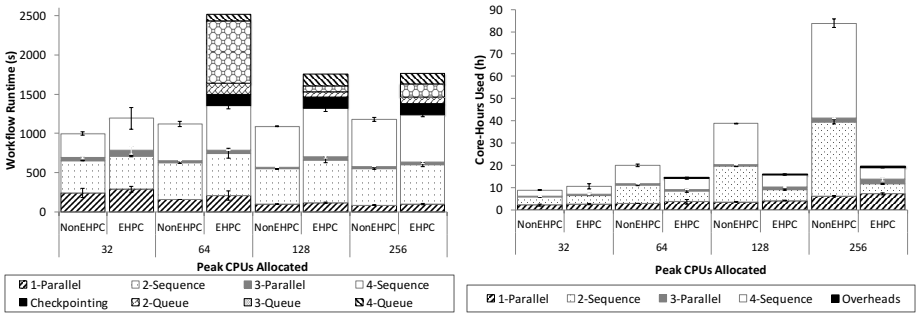


Figure 6: Synthetic (Gordon) - effects of dynamic resource scaling using E-HPC (a) Runtime, (b) Core-hours usage. E-HPC scales from 64 to 128 cores and achieves better performance than running the workflow over 64 cores.

5.3 E-HPC Elasticity

Figure 6 shows the benefits of E-HPC for dynamic resource scaling of an application. In this experiment, we run the synthetic parallel stage on 64 cores, 128 cores and scaling from 64 to 128 cores and we use fast mode to minimize the wait times. The fast mode allows applications to continue making progress while other resources are requested. In Figure 6, the E-HPC calculated bar is generated by taking the data from running E-HPC in fast mode and adding the queue time for the second job. We see that queue times are not substantial in this case. However, E-HPC in the fast mode provides even more significant benefits in cases where queue times are significant.

Figure 6a shows that fast mode results in around 30 percent improvement in workflow runtime as compared to maintaining the resources at 64 cores. The application is able to benefit from the added cores and complete the application sooner. Figure 6b shows corresponding core-hours expenditure for each run.



(a) Runtime.

(b) Core-hours.

Figure 7: Montage workflow performance (Cori): (a) workflow runtime and (b) core-hours usage for Montage with and without E-HPC. Montage shows shorter runtimes without E-HPC. For values of n larger than 32, E-HPC runs consume less core-hours.

5.4 Effect of Stage Elasticity

The stage elasticity in E-HPC allows workflows to request resource changes between the stages of a workflow (as described in Section 3.1). In this section, we present the results of using stage elasticity in E-HPC for different workflows.

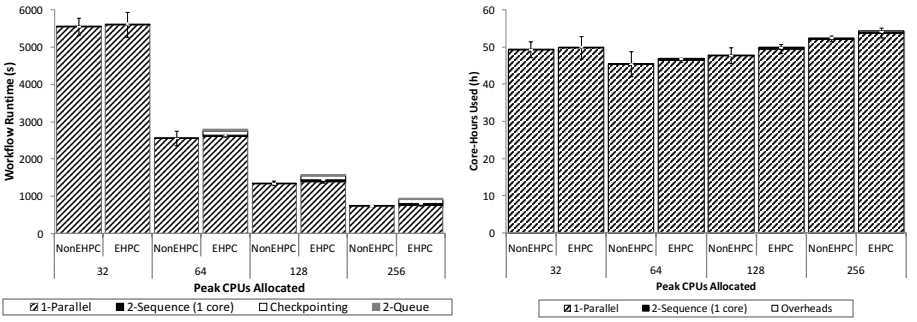
In our evaluation, we measure E-HPC performance using workflow runtime and allocated core-hours. Values observed in each experiment are presented in Figures 7 to 9. Each bar in the figure presents the average value (with standard deviation bars) over three repetitions of an experiment.

Experiments include runs with and without E-HPC and different resource allocation (32, 64, 128 and 256 cores). The X-axis represents the peak CPUs (n) allocated for the workflow in a particular experiment. In nonE-HPC runs, a value n on the X-axis corresponds to the number of CPU cores allocated during the complete lifecycle of a workflow. In E-HPC runs, n is the maximum number of CPU cores allocated during the duration of a workflow.

5.4.1 Montage

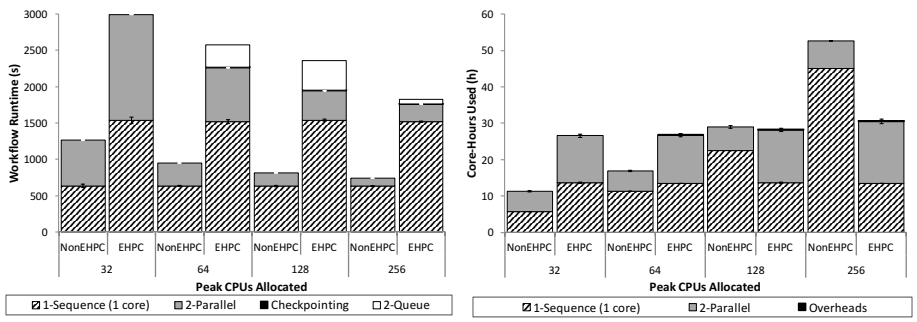
In this section, we evaluate the performance and resource usage of Montage.

Workflow Runtime. Figure 7a compares the workflow runtime for Montage with and without E-HPC. When running without E-HPC, the workflow runtime does not change substantially across different values of n and the shortest one is observed for $n = 32$ (single node on Cori). For larger n, the runtime of sequence stages (*2-Sequence* and *4-Sequence*) increases equally or more than the runtime gains in the parallel ones (*1-Parallel* and *3-Parallel*). This is likely due to the inter-stage data caching for different values of n. For instance, if $n = 32$, *1-Parallel* runs on a single node and all its output data (4.5 GiB) is cached locally (and will eventually be written to the file



(a) Runtime. (b) Core-hours.

Figure 8: BLAST workflow performance (Cori): (a) workflow runtime and (b) core-hours usage for BLAST with E-HPC and without E-HPC. BLAST execution is dominated by its first parallel stage. BLAST runtimes and core-hours are similar under both approaches, with slightly longer times and higher core-hour numbers in E-HPC.



(a) Runtime. (b) Core-hours.

Figure 9: Synthetic workflow performance (Cori): (a) workflow runtime and (b) core-hours usage for the Synthetic workflow with E-HPC and without E-HPC. Synthetic workflows show significantly shorter runtimes without E-HPC. Until 128 cores, less core-hours are allocated under E-HPC. For higher values of n, E-HPC uses less core-hours.

system). As a consequence, *2-Sequence* reads its input data mainly from memory. However, for $n = 64$, *1-Parallel* runs across two nodes, caching one half of its output data on each node. When *2-Sequence* starts on one of the two nodes, only half of its input data is locally cached. The runtime of the I/O intensive stage becomes 15% longer than for $n = 32$. This effect is also observed for *4-Sequence* since it is also an I/O intensive sequential stage preceded by a parallel one (*3-Parallel*). However, since its input data is larger than for *2-Sequence* (38 GiB), the effect is more noticeable, e.g., runtime of *4-Sequence* from $n = 32$ to $n = 64$ increases 52%.

When run with E-HPC, Montage workflow runtime presents a different pattern. Again, workflows running on a single node ($n = 32$) present the shortest runtime because the tasks run on a single node and in the same job allocation. For $n > 32$, the inter job wait time increases the workflow runtime significantly compared to $n = 32$. This is expected since for $n > 32$, E-HPC runs each section of the workflow in a separate job to adjust the resource allocation to the desired size (Figure 5a). As n increases, the runtime for the sequence stages does not change and parallel stages becomes shorter, decreasing the overall runtime.

The comparison between running Montage with or without E-HPC shows that runtime is longer with E-HPC in all cases. When run on a single node, the 20% runtime increase is due to monitoring overhead of DMTCP. For $n > 32$, the workflow runtime difference is contributed by the inter-job wait time and longer runtime of the stages. The inter job wait time is dependent on the current workflow of the system and out of the control of E-HPC. For most cases, total stage runtime is $\approx 20\%$ longer with E-HPC. As we scale to $n = 256$, nonE-HPC runs can no longer benefit from inter-stage data caching due to data being distributed across multiple node, and hence, stage runtime overhead in E-HPC is reduced to 10% as compared to nonE-HPC.

Workflow core-hours. Core-hours consumed by all the experiments with Montage on Cori are detailed in Figure 7b. In cases without E-HPC, larger allocations increase the core-hours consumed. Without elasticity, the sequence stages consume significantly more core-hours since their runtime is not reduced by the larger resource allocation. Also, parallel stages, when scaling up from $n = 32$ to $n = 256$, consume slightly more core-hours due to the increasing overheads of the initial setup in Tigres for launching the parallel tasks across multiple nodes For Montage, $n = 32$ (single node on Cori), 72% of the workflow runtime is consumed by serial stages (*2-Sequence* and *4-Sequence*).

With E-HPC, for values of $n > 32$, doubling the allocated resources induces small variations in consumed core hours. For example, stepping n up from 64 to 128, increase core-hours consumption by 11%. Core-hour usage increases are attributed to the natural overhead of less than perfect parallelism in the code, and the initial overhead of distributing the tasks through Tigres across multiple nodes. Otherwise, with E-HPC, there is no resource wastage and checkpointing core-hours are very small ($< 1\%$). However, there is a larger step between $n = 32$ and $n = 64$, with an increase of 35% of core hours. This is due to the loss in efficiency in the sequence stages due to lack of caching of intermediate data.

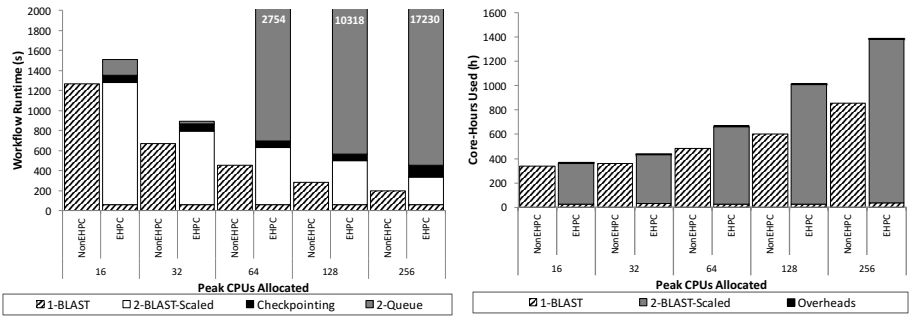
Finally, comparing runs of Montage with the two approaches, for values $n > 32$ (elasticity is possible), E-HPC requires significantly less core-hours (76% for 256 cores) than nonE-HPC due to elastic management of resources. The core-hour results in Montage show that with increasing parallelism, E-HPC utilizes resources more efficiently than nonE-HPC due to diverse level of parallelism.

5.4.2 BLAST

This section focuses on evaluating the impact of E-HPC on BLAST.

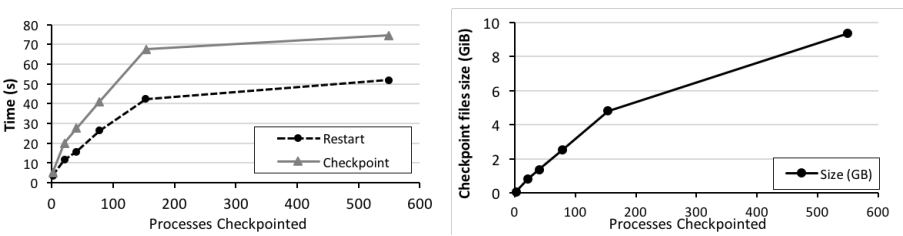
Workflow runtime. The runtime of all the experiments running BLAST on Cori are presented in Figure 8a. When run without E-HPC, BLAST's runtime is dominated by the first parallel stage (*1-Parallel* occupies $> 99\%$ of the total runtime in all cases). This stage scales well over more resources and overall workflow runtime is significantly reduced when run over more resources. For instance, for $n = 64$ the runtime is less than half (53% shorter) than for $n = 32$.

Similar workflow runtimes are observed in BLAST when run with E-HPC. DMTCP's monitoring overhead is relatively small but becomes more significant for larger values of n. e.g., DMTCP increases BLAST's runtime by 1% for $n = 32$ and 8% for $n = 256$. For all values of n, checkpoint, restart, and queue times increase the overall workflow runtime by ≈ 2 minutes, which is not significant compared to the overall workflow runtime. In summary, BLAST scales well as the workflow allocation is increased, and has very little overhead when run with E-HPC.



(a) Runtime. (b) Core-hours.

Figure 10: Runtime elasticity on BLAST (Gordon) vs static allocation: (a) workflow runtime and (b) core-hour usage for BLAST. The E-HPC coordinated job starts on a single 16 core node and expands to the peak core allocation after 60 seconds of execution.



(a) Performance overhead (b) Checkpoint overhead

Figure 11: E-HPC overheads: (a) shows time required for checkpoint and restart vs. the number of processes/tasks being tracked by E-HPC, (b) shows total storage required on filesystem for a checkpoint versus the number of processes/tasks being tracked by E-HPC.

Workflow core-hours. Core-hours consumed with BLAST (on Cori) are detailed in Figure 8b. Similar core-hours are observed with and without E-HPC and different values of n, e.g., the maximum values differ less than 9% from the average. This is caused by the domination of (*1-Parallel*) over the execution of the workflow that makes other non-scaling stages irrelevant in terms of core-hours.

The comparison between $n = 32$ and $n = 64$ cases present an unexpected result: the core-hours are reduced when parallelism is increased. This is caused by the unexpected super-linear reduction of runtime in that step of n observed in Figure 8a. In the next steps of n, core-hours increase slowly from the expected imperfection of parallelism in the code.

Comparison between using and not using E-HPC shows thatE-HPC consumes 1.5% to 5% more core-hours with no clear correlation to n. Checkpointing overhead in all cases consumes less than 0.5%. This leads to the conclusion that the additional core-hours consumed by E-HPC are for DMTCP execution overhead.

5.4.3 Synthetic

This section describes our evaluation of the Synthetic workflow with E-HPC.

Workflow runtime. The runtimes observed of all experiments with Synthetic workflow are presented in Figure 9a. When run without E-HPC, the workflow runtime becomes shorter for larger values of n. This reduction is the result of shorter stage runtimes of the *2-Parallel* stage when more resources are available (*1-Sequence* is sequential and thus its runtime is constant): *2-Parallel* runtime is reduced $40-49\%$ each time n is doubled.

The Synthetic runs with E-HPC present much longer runtimes than without E-HPC. This is due to DMTCP monitoring overhead that slows down the execution of all stages by a $2.2-2.4$ factor. Detailed analysis of the workflow reveals that most of the operations performed by the workflow were memory management (allocation and free). These operations are heavily monitored by DTMCP that traps all the memory management calls. The workflow runtime evolution for larger values of n is as expected: *1-Sequence* runtimes remain constant, and *2-Parallel* runtime is reduced significantly (again $40-49\%$). Finally, E-HPC checkpoint overheads are minimal (6 seconds for all values of n) and the queue times for the second job is typically a few minutes.

Workflow core-hours. Core-hour consumed in all experiments with Synthetic are detailed in Figure 9b. For nonE-HPC, a larger resource allocation implies a significant increase in core-hours consumed by the workflow. This increase is mainly due to the wastage of the resources by the *1-Sequence* stage.

Runs of Synthetic with E-HPC consume almost the same core-hours for all values of n. This is caused by the constant resource consumption of both stages in the workflow. *1-Sequence* consumes the same core hours because elasticity allows to execute *1-Sequence* over 32 cores in all cases. *2-Parallel* runtime decreases proportionally to the increase in assigned resources, keeping its core-hours consumption almost unchanged.

5.5 Effect of Runtime Elasticity

Figure 10a shows the use of E-HPC for inducing elasticity in the middle of a workflow stage (runtime elasticity), as it expands from 16 cores (one node on Gordon) into a larger set. Although E-HPC is capable of scaling up in the middle of a workflow stage, the results in the figure show that the total workflow runtime is affected when using E-HPC. The sequential stages in BLAST are extremely short in comparison to the longer parallel stage. The runtime for 16 cores with and without E-HPC are similar because in both cases, all the stages use 16 cores (one node on Gordon). However, as we scale up to 32 cores, E-HPC takes $\approx 6\%$ more time. When using E-HPC the first sequence stage, 1-BLAST is executed on one node and the elasticity is induced after 60 seconds. During this time, some of the tasks in the second stage, 2-BLAST-Scaled, which is a parallel stage, have already started executing. The higher degree of parallelism during the initial parallel stage and the checkpoint/restart overhead of DMTCP, the overall runtime performance deteriorates with E-HPC. The pattern continues for larger cores, and for up to 256 cores (with E-HPC taking $\approx 20\%$ more time than without E-HPC), E-HPC runtime elasticity performs poorly compared to when executed without E-HPC. This is a significant result, because it shows that the time when elasticity is induced is also critical to certain workflows, and may result in performance degradation if the required resources are not allocated at the right time.

5.6 E-HPC Overheads

In this section, we evaluate the different overheads in E-HPC. Table 3 shows the runtime overhead of various workflows, with and without the queue wait times, when running with E-HPC on both Cori (C) and Gordon (G). As E-HPC resubmits a job while scaling up, it incurs an additional queue wait time in addition to the checkpoint and restart overheads of DMTCP. As can be seen from the table, the overheads including the queue wait time are significantly higher than excluding the wait time (for e.g., runtimes are 86.3% longer with queue wait time vs 10.3% longer without the queue wait time for Montage on Cori). This is because the queue wait time dominates the overheads in these cases and is a system-dependent variable on which neither E-HPC, nor DMTCP have any control. On the other hand, the overheads without the queue wait time only include DMTCP checkpoint and restart times, which has a maximum of $\approx 36\%$ overhead (for BLAST on 256 cores on Gordon). BLAST is a memory-intensive application, with a large memory footprint that generates large checkpoint images. The overheads are smaller on Cori ($\approx 13\%$) than on Gordon, because of the large I/O bandwidth of the Lustre file system (700 GB/s), as compared to the peak I/O bandwidth on Gordon (100 GB/s). For all other workflows, the runtime overhead varies between $0.2\% - 11\%$, when there are no queue wait times. Hence, with current advancements in storage system (e.g., burst buffers) and checkpoint restart systems, E-HPC overheads can be minimized.

Figure 11a) shows the overheads in E-HPC due to the checkpoint and restart phases. Both checkpoint and restart overheads are proportional to the number of workflow tasks in execution, and the overheads increase linearly up to 150 tasks/processes. The

		% Overhead, Wait (Without wait)			
Workflow	Sys.	32	64	128	256
Montage	C	N/A	86.1(10.3)	32.8(10.3)	42.3(11.1)
BLAST	C	N/A	5.7(3.9)	10.5(7.6)	18.3(13.6)
Synth	C	N/A	13.8(0.29)	21.4(0.36)	3.5(0.37)
BLAST	G	13.11(9.3)	448(10.9)	2085(13.4)	5210(36.3)
Synth	G	4.5(0.8)	4.7(1.8)	5(2.0)	N/A

Table 3: E-HPC overheads including (left) and excluding system dependent wait times (brackets). E-HPC controlled overheads vary between $0.2\% - 36\%$. BLAST supports higher overheads due to its larger memory footprint and hence, larger checkpoints.

overhead is due to the added communication between the workflow tasks and the DMTCP coordinator, and the I/O overhead of writing the checkpoint image to disk.

Figure 11b) shows that the storage space overhead also increases linearly with the increasing number of tasks. The total amount of memory and compute requirements increase with increasing tasks, thereby increasing the total checkpoint size. An important observation from Figure 11b is that the checkpoint size may become so large that it can result in I/O performance bottlenecks that can significantly affect the overall E-HPC performance. DMTCP provides optimizations for writing checkpoint images to memory, and also provides compressed checkpointing to minimize the memory and storage footprint of checkpoints and restarts. These optimizations can be used to minimize the overheads in E-HPC.

5.7 Summary

In this section, we summarize the experimental results. The workflow runtimes are $\approx 6\% - 20\%$ time longer in E-HPC as compared to running the workflow without E-HPC. The runtime results for the workflows show that the performance of workflows with E-HPC is affected due to the checkpoint-restart overhead, queue wait time and the underlying application characteristics (Figure 7a, Figure 7b).

E-HPC improves the core-hours used for running the workflows by up to 76%. The core-hour results show that with increasing parallelism, and longer sequential stages, E-HPC utilizes resources more efficiently than its counterpart by allocating only as many resources as needed for a stage in the workflow (Figure 8a, Figure 8b).

The runtime overheads in E-HPC vary between $0.2\% - 36\%$, when excluding the highly variable queue wait times. Further evaluation shows that the overheads are solely due to the underlying file system, and DMTCP (checkpoint/restart library) (Table 3).

6 Discussion

In this section, we discuss the trade-offs of having a checkpoint-restart based elastic framework for HPC workloads, and the lessons learned while building and evaluating E-HPC.

6.1 E-HPC Trade-offs

E-HPC acts as a middleware to manage resource elasticity for HPC workflows. An alternative approach is to integrate elasticity in the HPC resource scheduler. However, the two approaches have their own pros and cons as discussed below.

Overheads vs Portability. As shown through our experiments, E-HPC cannot control several overheads including the queue wait time and checkpoint-restart. By integrating the elastic resource management into HPC schedulers, queue wait times can be minimized by only queueing jobs when resource changes can be made. However, this is not a portable solution, as the current HPC schedulers need to be changed. A middleware approach like that of E-HPC provides a portable elastic resource management solution that can work with different HPC schedulers and workloads.

Global vs Local Optimization. E-HPC minimizes the queue wait times by submitting placeholder jobs for future resource changes. However, this kind of optimization that is local to a single workflow may adversely affect the turnaround time of other workloads by increasing their queue wait times. In order to avoid such scenarios, schedulers can do global optimizations based on all the workloads.

6.2 Lessons Learned

- The elasticity decisions are largely dependent on the workflow characteristics and the undelying system architecture. Not all workflows will benefit by using elastic resources as shown through our experiments. In addition, resource schedulers, and memory and I/O bandwidth of HPC systems can dictate the decisions of elastic resource management through checkpoint-restart.
- The turnaround time of workflows running through an elastic resource management framework like E-HPC is not only influenced by the workflow characteristics, but also by the types of workloads in an HPC environment. This is because the queue wait times are dependent on the distribution of large and small jobs in the system.
- The queue wait times can be largely minimized by improving the resource scheduling strategies and making them aware of the workflow characteristics.
- The use of Burst Buffers for using checkpoint-restarts can significantly minimize the runtime overheads of elastic workflows.

7 Related Work

Elasticity. In cloud computing, schedulers [Ver+15], [Bur+16], [Arm+09], satisfy user performance demands by dynamically altering the resource allocation to jobs [M+11]. Also, cloud workflow managers [Vav+13], [Kul+15], perform fine-grained allocation for each workflow stage. In all cases, dynamic resource allocation presents

a key challenge of resource liberation, i.e., deallocation of occupied resources to enlarge allocations when the required amount is not free. In cloud environments, resource liberation is achieved by applying workload consolidation [SKZ08], workload preemption [Ver+15], or resource oversubscription [TT13]. These techniques imply a potential performance reduction or cancellation of some applications to benefit others [Sha+11].

In HPC, individual application performance is subordinated to overall objectives such us high utilization and performance efficiency [MF01]. Elasticity in the HPC space has shown progress through the creation of applications built for malleability [Rod+15], [ME08] or moldability [Fei+97], [KP11]. Others methods include elastic job bundling, where numerous smaller jobs are submitted in order to deploy a large set of nodes more quickly [LW15]. Finally, some modern schedulers support special jobs which aggregate resources to a running job upon start [Zho+13]. However, effectiveness of these techniques depends on the synchronization between the start of different jobs, which is hard to accomplish. E-HPC provides elasticity in HPC without the caveats of the techniques described in this section. E-HPC does not significantly impact the overall system performance since preemption is not required, and it enables the automatic restart of jobs that run over their time limit.

Resource management in scientific workflows. Systems like Pegasus [Dee+04], Askalon [Fah+07], Koala [ME05], VGRaDS [Ram+09], or DAGMan [Cou+07] are used to run scientific workflows. They provide functions such as workflow mapping (i.e., task grouping for efficient execution), monitoring, fault tolerance, execution, and meta-scheduling. These systems usually schedule workflow tasks across different sites and rely on the local scheduler for fine-grained resource management. However, local HPC scheduler rarely incorporate workflow aware mechanisms [Rod+17] and workflow tasks might be scheduled inefficiently.E-HPC does not provide the high level functions of other workflow systems. However, it enables fine grained resource allocation to workflows by providing elastic execution of tasks.

Workflow turnaround time reduction. Previous work studied methods to reduce initial and intermediate job wait times that elongate turnaround time in workflows. For example, Mesos [Hin+11], Omega [Sch+13], Koala [ME08], or A2L2 [Rod+15] describe schedulers that minimize intermediate wait times by managing workflows separately from the rest of the workload. Approaches like WoAS [Rod+17], bring workflow aware scheduling to classical HPC schedulers by extending the queue model. Other systems propose job bundling [LW15] and task clustering [Sin+08] to efficiently execute workflows with multiple tasks. Workflow runtime in E-HPC can be increased by intermediate job wait times. This effect is eased by submitting jobs before their precursors are completed (i.e., *fast mode*). However, effective use of this technique requires further investigation in combination with queue wait time prediction methods.

8 Conclusions and Future Work

In this paper, we present the design, implementation and evaluation of E-HPC, a flexible elastic framework that provides increased efficiency of resource utilization, failure recovery, elasticity, and faster execution times. The overheads of E-HPC vary based on the characteristics of the application (e.g., amount of I/O), resource configuration and run-time characteristics (e.g., queue wait time). The overheads are from checkpoint and restart, while providing significant benefits in dynamic elastic resource management. E-HPC is designed to work independently as well as with existing software ecosystems. E-HPC provides an effective library for the fine tuned control of resources in HPC environment, where before now, real time control was difficult, if not impossible. E-HPC is the foundational tool needed to address the resource management needs of next-generation real-time and streaming workflows.

References

[Arm+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. *Above the clouds: A berkeley view of cloud computing*. Tech. rep. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[Bur+16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, omega, and kubernetes". In: *Communications of the ACM* 59.5 (2016), pp. 50–57.

[Cor16] NERSC Cori. `http://www.nersc.gov/users/computational-systems/cori/configuration/`. 2016.

[Cou+07] Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. "Workflow management in condor". In: *Workflows for e-Science* (2007), pp. 357–375.

[Dee+04] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. "Pegasus: Mapping scientific workflows onto the grid". In: *Grid Computing*. Springer. 2004, pp. 11–20.

[Dee+05] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. "Pegasus: A framework for mapping complex scientific workflows onto distributed systems". In: *Scientific Programming* 13.3 (2005), pp. 219–237.

[Fah+07] T. Fahringer, R. Prodan, R.Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. "ASKALON: A Development and Grid Computing Environment for Scientific Workflows". In: *Workflows for e-Science*. Ed. by I. Taylor et al. Springer-Verlag, 2007, pp. 450–471.

[Fei+97] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. “Theory and practice in parallel job scheduling”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1997, pp. 1–34.

[GB12] Guilherme Galante and Luis Carlos E de Bona. “A survey on cloud computing elasticity”. In: *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*. IEEE. 2012, pp. 263–270.

[GHB04] Duane A. Gilmour, James P. Hanna, and Gary Blank. “Dynamic Resource Allocation in an HPC Environment”. In: *Proceedings of the 2004 Users Group Conference*. DOD UGC 2004. Washington, DC, USA: IEEE Computer Society, 2004, pp. 260–265. ISBN: 0-7695-2259-9. DOI: 10.1109/DOD_UGC.2004.11. URL: http://dx.doi.org/10.1109/DOD_UGC.2004.11.

[Gor15] XSEDE Gordon. *http://www.sdsc.edu/support/user_guides/gordon.html*. 2015.

[Hen+16] Valeria Hendrix, James Fox, Devarshi Ghoshal, and Lavanya Ramakrishnan. “Tigres Workflow Library: Supporting Scientific Pipelines on HPC Systems”. In: *Cluster, Cloud, and Grid Computing (CCGrid), 2016 16th IEEE ACM International Symposium* (May 2016).

[Hin+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.” In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[Jac+09] Joseph C Jacob, Daniel S Katz, G Bruce Berriman, John C Good, Anastasia Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas Prince, et al. “Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking”. In: *International Journal of Computational Science and Engineering* 4.2 (2009), pp. 73–87.

[Juv+13] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. “Characterizing and profiling scientific workflows”. In: *Future Generation Computer Systems* 29.3 (2013), pp. 682–692.

[KP11] Cristian Klein and Christian Perez. “An rms architecture for efficiently supporting complex-moldable applications”. In: *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. IEEE. 2011, pp. 211–220.

[Kul+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. “Twitter heron: Stream processing at scale”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 239–250.

[LW15] Feng Liu and Jon B. Weissman. “Elastic Job Bundling: An Adaptive Resource Request Strategy for Large-scale Parallel Applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’15. Austin, Texas: ACM, 2015, 33:1–33:12. ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807610. URL: http://doi.acm.org/10.1145/2807591.2807610.

[M+11] Peter Mell, Tim Grance, et al. “The NIST definition of cloud computing”. In: (2011).

[ME05] H. H. Mohamed and D. H. J. Epema. “Experiences with the KOALA co-allocating scheduler in multiclusters”. In: *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID2005)*. IEEE Computer Society, 2005, pp. 784–791.

[ME08] Hashim Mohamed and Dick Epema. “KOALA: a co-allocating grid scheduler”. In: *Concurrency and Computation: Practice and Experience* 20.16 (2008), pp. 1851–1876.

[MF01] A. W. Mu’alem and D. G. Feitelson. “Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling”. In: *IEEE Transactions on Parallel and Distributed Systems* 12.6 (June 2001), pp. 529–543. ISSN: 1045-9219. DOI: 10.1109/71.932708.

[Pen+10] William D Pence, L Chiappetti, Clive G Page, RA Shaw, and E Stobie. “Definition of the flexible image transport system (fits), version 3.0”. In: *Astronomy & Astrophysics* 524 (2010), A42.

[PMS16] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier. “soCloud: a service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds”. In: *Computing* 98.5 (2016), pp. 539–565.

[RAC06] Michael Rieker, Jason Ansel, and Gene Cooperman. “Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux”. In: *The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications*. Las Vegas, NV, June 2006, pp. 492–498.

[Ram+09] Lavanya Ramakrishnan, Charles Koelbel, Yang-Suk Kee, Rich Wolski, Daniel Nurmi, Dennis Gannon, Graziano Obertelli, Asim YarKhan, Anirban Mandal, T Mark Huang, et al. “VGrADS: enabling e-Science workflows on grids and clouds with fault tolerance”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE. 2009, pp. 1–12.

[Rod+15] Gonzalo Pedro Rodrigo Álvarez, Per-Olov Östberg, Erik Elmroth, and Lavanya Ramakrishnan. “A2l2: An application aware flexible hpc scheduling model for low-latency allocation”. In: *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*. ACM. 2015, pp. 11–19.

[Rod+17] Gonzalo P Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. "Enabling Workflow-Aware Scheduling on HPC Systems". In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM. 2017, pp. 3–14.

[Sch+13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 351–364.

[Sha+11] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. "A cost-aware elasticity provisioning system for the cloud". In: *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*. IEEE. 2011, pp. 559–570.

[Sin+08] Gurmeet Singh, Mei-Hui Su, Karan Vahi, Ewa Deelman, Bruce Berriman, John Good, Daniel S Katz, and Gaurang Mehta. "Workflow task clustering for best effort systems with Pegasus". In: *Proceedings of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids: Understanding the spectrum of distributed computing requirements, applications, tools, infrastructures, interoperability, and the incremental adoption of key capabilities*. ACM. 2008, p. 9.

[SKZ08] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. "Energy aware consolidation for cloud computing". In: *Proceedings of the 2008 conference on Power aware computing and systems*. Vol. 10. San Diego, California. 2008, pp. 1–5.

[Sou15] Pegasus University of Southern California Information Sciences Institute. "Pegasus Montage Tutorial". In: (2015).

[TT13] Luis Tomás and Johan Tordsson. "Improving cloud infrastructure utilization through overbooking". In: *Proceedings of the 2013 ACM Cloud and Autonomic Computing conference*. ACM. 2013, p. 5.

[Vav+13] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[Ver+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 18.

[Wil+11] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. "Swift: A language for distributed parallel scripting". In: *Parallel Computing* 37.9 (2011).

[YB05] Jia Yu and Rajkumar Buyya. “A taxonomy of workflow management systems for grid computing”. In: *Journal of Grid Computing* 3.3-4 (2005), pp. 171–200.

[Zho+13] Xiaobing Zhou, Hao Chen, Ke Wang, Michael Lang, and Ioan Raicu. “Exploring distributed resource allocation techniques in the slurm job management system”. In: *Illinois Institute of Technology, Department of Computer Science, Technical Report* (2013).