Toward Improved Support for Loosely Coupled Large Scale Simulation Workflows

Swen Boehm, Wael R. Elwasif, Thomas Naughton, and Geoffroy R. Vallée
Computer Science & Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
Email: {bohms,elwasifwr,naughtont,valleegr}@ornl.gov

***Abstract*—High-performance computing (HPC) workloads are increasingly leveraging loosely coupled large scale simulations. Unfortunately, most large-scale HPC platforms, including Cray/ALPS environments, are designed for the execution of long-running jobs based on coarse-grained launch capabilities (e.g., one MPI rank per core on all allocated compute nodes). This assumption limits capability-class workload campaigns that require large numbers of discrete or loosely coupled simulations, and where time-to-solution is an untenable pacing issue. This paper describes the challenges related to the support of fine-grained launch capabilities that are necessary for the execution of loosely coupled large scale simulations on Cray/ALPS platforms. More precisely, we present the details of an enhanced runtime system to support this use case, and report on initial results from early testing on systems at Oak Ridge National Laboratory.**

***Keywords*-Framework, Runtime, Ensemble computing.**

I. Introduction

A growing number of scientific applications rely on a regime of loosely coupled large scale computational workloads for their underlying execution environment. Large scale multi-dimensional parameter sweeps and optimization studies typically involve the execution of a large number of low to medium parallelism tasks, with little or no inter-task communication or dependencies. This type of workload has demanding aggregate computational and memory requirements, rendering them as natural candidates for deployment on current leading high-performance computing (HPC) platforms. Extreme scale computing ensembles comprised of loosely coupled workloads have been recognized as an appropriate way to utilize leadership class computing resources managed by the U.S. Department of Energy [1]. This execution model has been successfully used in scientific disciplines that range from subsurface modeling, image-guided neurosurgery, climate data analysis, and many other domains.

Deploying such workloads on existing large-scale HPC systems can be inefficient, as the batch environment on such systems is usually designed to support a different model where a small number of application instances execute in the context of an allocation; and the workload is tightly-coupled, generating a large amount of communication. As a result, the task instantiation and management interfaces offered in many of these HPC systems do not readily support fine-grained launch capabilities needed for efficient execution of many-task computing workloads.

In this paper, we present recent work to build a runtime environment and user-level task management infrastructure to support the efficient execution of large scale loosely coupled workloads on leading HPC platforms. The focus of this work has been to provide additional support to users on the Cray XK7 system at Oak Ridge National Laboratory (ORNL), allowing for low-overhead instantiation and management of loosely coupled computational tasks within a single user's batch system allocation. The implementation leverages the Cray/ALPS task launcher for the base deployment, which is used to bootstrap the user specific environment on the compute nodes within the given job allocation. This approach alleviates the burden on the service nodes that would result from servicing a large number of (possibly concurrent) task instantiation requests, which can manifest itself in resource contention or un-availability on these shared nodes. The enhanced task management environment also offers the ability to customize the launch capabilities within a user's job, e.g., by allowing for dynamic node sharing between computational tasks that do not interfere with each other, i.e., a memory bound task is co-located with a central processing unit (CPU) bound task. This allows for time-sharing of the nodes in a space-shared allocation (i.e., within a user's job allocation on the system).

The remainder of the document is organized as follow: Section II presents related work and discusses the requirements needed to support loosely coupled workloads on leadership computing platforms as well as an overview of the Scalable runTime Component Infrastructure (STCI) [2] that has been extended to prototype the proposed architecture. In Section III, we describe the design of the proposed system and Section IV gives implementation details. In Section V, we presents preliminary evaluation results. Section VI concludes.

II. Background & Related Work

In this section, we define terminology used throughout the document. We then describe the requirements for the execution of loosely coupled scientific simulations on leadership computing platforms, as well as previous research. We

present STCI, which has been used to prototype the proposed solution, focusing on relevant characteristics and capabilities of importance the research presented in this paper. Finally, we present an overview of the job management of Cray systems.

A. Terminology

For clarification, we first define the concept of *job* and *task* that are used across this document. Since most HPC systems are using a batch system, we defined a job as a sequence of statements and data that represents a single unit of work that will execute on available computing resources. A job is instantiated via a single invocation of job submission command (e.g., `qsub`), typically using a batch job script to define the resources being requested, as well as the way in which these resources will be utilized to perform the set of operations that define the job. Since a job can be a complex work-flow with dependencies, we call *task* every execution unit composing a job. In the context of traditional HPC systems, a task is, for example, a single invocation of program dispatch command (e.g., `mpirun` or `aprun`) within a batch job script.

B. Loosely Coupled Large Scale Simulation Work-flows for High Performance Computing

Traditionally, an HPC application is realized via the execution of a massively parallel workload that utilizes a multitude of compute nodes, using a *tightly coupled* inter-process communication pattern that relies on the high bandwidth and low latency characteristics of modern HPC platforms to achieve scalable performance. Furthermore, a traditional HPC batch job involves the execution of one task, or a small number of tasks, within a single batch system allocation. Another model that is increasingly being deployed to large scale HPC platforms is one where a single HPC applications involves the execution of a large number of tasks, which are typically *loosely coupled*, with little or no inter-task communication. This computing regime is typically referred to as Many-Task Computing (MTC) [3], [4].

Running a large number of loosely coupled tasks is a challenge for the software stacks in most HPC systems: a task can be short lived, and the creation of many short lived tasks can put excessive load on the service nodes, where the tools for task launching, monitoring, and termination are typically executed. It is therefore a requirement that to support large scale, many-task computing applications, the tools used to manage the lifecycle of such tasks minimize, or totally eliminate, the per-task incremental overhead (both in terms of wall clock time, and resource utilization). This minimization goal is typically not the highest priority for default task management infrastructure, with its focus on few massively parallel tasks. Use of the many-task computing paradigm is finding increasing use in many disciplines that include biology, genomics, among other scientific disciplines. A common mode for such use involves the execution of a *parameter sweep*, in which the same program is executed repeatedly with different input parameters to explore a (typically large) parameter space for regions of interest or special significance. The ability to *concurrently* execute a large number of program instances on leadership class computing platforms has made this approach increasingly popular with scientists who rely on this mode of study. However, the mis-match between the default program execution model, and the light weight approach needed for the efficient execution of *massively concurrent* programs has slowed down the use of this technique on leadership class computing platforms.

Several attempts have been made to enable the efficient execution of parameter sweep applications on modern CRAYs and other HPC platforms. Eden [5] is a tool developed to ease the execution of concurrent serial jobs, by supporting interactions with PBS on a SGI Altix 1000. Eden is composed of a set of shell scripts that automates the generation of PBS job scripts, which get submitted to the system job scheduler. Additionally, Eden manages the standard output and the standard error, and collects timing data. At the end of the job, this data is collected and put into a summary file. The Parallel Command Processor (PCP) [6] tool is another tool for the execution of serial jobs on the Cray system. Originally developed by the Ohio Supercomputer Center (OSC) and ported by the National institute for Computational Sciences (NICS), this tool processes a text file containing a list of single-core commands that are then farmed out to available compute-node cores, and executed via a `system()` call. While this approach is adequate for serial jobs, it does not support the need for small to medium parallel tasks that are becoming more common in many-task computing applications.

SAGA-BigJob [7] is another tool that targets efficient execution of a large number of serial tasks on available compute resources. While the tool targets primarily distributed systems, it can also be used to manage serial task execution on Crays [8]. Written in Python, the tool relies on the Python subprocess module's `Popen` class to launch the target serial task on a compute node, and as such cannot be used to handle parallel tasks since those can not be launched using a `system` call from a task running on a compute node.

Another class of tools that can benefit from improved efficiency in task dispatch are tools for loosely coupled multi-phyisics simulations. One such tool, The Integrated Plasma Simulator (IPS) [9] is a lightweight Python framework for file-based, loosely coupled simulations. The IPS relies on the existing program dispatch environment on target platforms for task execution. The multi-level concurrency support in the IPS [10] allows a single coupled simulation running within the framework to make efficient use of available computational resources. When coupled with the

DAKOTA optimization toolkit [11] and used in a parameter sweep mode [12] the IPS can generate a massive number of concurrent tasks, taxing resources on service nodes and limiting the scalability of the overall application.

C. Scalable runTime Component Infrastructure – STCI

In this section, we present the relevant details of STCI, highlighting the existing features and justifying the proposed extensions. STCI is developed as part of ongoing system software and resilience research at ORNL [2] and was initially designed to support parallel applications based on a message passing paradigm. As such, the message passing interface (MPI) is currently the only supported execution model.

The STCI runtime infrastructure is based on three main concepts: (i) sessions, (ii) jobs and (iii) tasks. A session represents all the resources allocated to a given user and a user's job is executed in the context of a session.

Fortunately, for genericity and portability purposes, STCI has been designed based on the Modular Component Architecture (MCA) (from the Open MPI [13] project). MCA provides an interface to define frameworks, components and modules which can be loaded at run time, which allows for a high level of modularity and easy extensibility. The current STCI implementation provides modular building blocks that implement various system services. These building blocks can be composed to form new runtimes for HPC.

The different modules required for the implementation of new system services are loaded at run time. The instantiation of all the required services, in addition to some required STCI-specific services (e.g., out-of-band communication services), is done in the context of *agents* that are connected to each other for the creation of a distributed runtime system. STCI is based on two types of agents: *infrastructure agents* and *user agents*. Infrastructure agents ensure that all services can be bootstrapped and terminated so that a user's agents can be deployed and run successfully. The infrastructure agents are: (i) *Front-end* (FE); (ii) *Controller* (CTRL); (iii) *Root Agent* (RA); and (iv) *Session Agent* (SA). The FE generally runs on the login, management node or on the end-user's computer. It is the end-user's interface to the STCI runtime environment. Since the FE is only the user interface with the STCI infrastructure, a CTRL is deployed for each user, tracking running jobs and ensuring, on behalf of the user, that the scientific applications are correctly deployed and terminated on compute nodes. CTRLs are usually executed on the head/login or service node. As such, the FE usually starts and interacts with the CTRL for any given job. Since STCI has been designed to support long-running large-scale scientific simulations, it allows the user to disconnect from the HPC platform during the application execution; the FE attaching to and detaching from the CTRL. Like FE and CTRL, RAs and SAs are part of the STCI management infrastructure. The RAs are responsible for the resource management on the compute nodes. They typically abstract the resource allocation protection mechanism specific to the target HPC platform (i.e., some resources may require privileged access for allocation but still need to be isolated from a user's application for security reasons). Additionally, since RAs are by design privileged agents, they may be part of the bootstrapping and responsible for starting additional infrastructure agents, as well as user agents. Finally, *Tool agents* (TA) instantiate the user's executable such as an MPI application. For that, the STCI infrastructure creates an execution context on allocated compute nodes and ensure access to required system services. TAs are accompanied on each node by one SA.

While STCI provides different agents for most HPC platforms, this approach also allows for new agent types to be developed to easily extend the system. For instance, a Cray specific RA has been developed to accommodate the specificities of Cray systems in terms of resource management (only the ALPS software, presented in Section II-D can be used for bootstrapping of any process on compute nodes).

STCI relies heavily on topologies. The topologies describe how the different agents are connected, how messages should be routed and how processes should be mapped to available resources. Figure 1 shows an example of a topology containing a FE, the CTRL, three RAs and SAs and six TAs. To deploy all the agents required for the

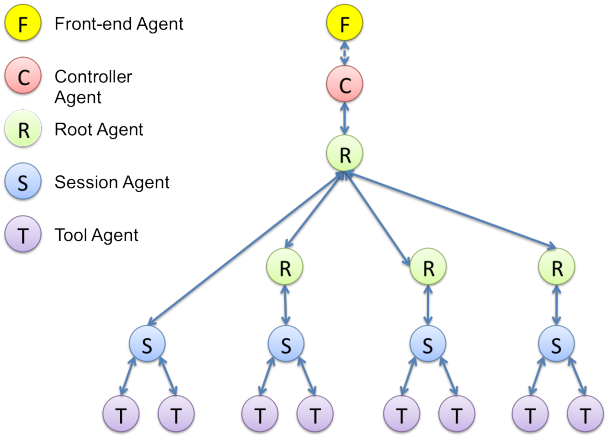

Figure 1: Default layout of the different STCI agents required for the execution of a job

execution of a job [1], STCI maps the different agents in a topology onto available resources. Based on this deployment topology, STCI's launcher actually creates the different agents across the different compute nodes allocated to the job. The launcher abstracts the system configuration of the

[1]STCI internally tracks each set of remote agent launches with a distinct identifier called a Job Identifier (JobID). This "STCI job" should not be confused with the batch queue system "job".

HPC platform, e.g., forking a process (local process launch), implicitly uses *ssh* on clusters where nodes are directly accessible via ssh, or uses ALPS on Cray systems.

D. Job Management on Cray Systems

The Cray ALPS (Application Level Placement Scheduler) system provides remote launch functionality for running executables on compute nodes in the system. The ALPS system also performs placement and reservation management as well as resource management at the node level to ensure cleanup between user jobs. A more detailed description of ALPS and its use for management and monitoring of Cray compute resources is available in [14]. A set of utilities, running on service/login nodes, can be used to query about ALPS status and placement information (`apstat`) and to launch task on the remote nodes (`aprun`). All remote executables must be launched using `aprun`, which assigns a unique ALPS identifier (*apID*) for tracking this execution instance within the system. At the compute node level, there are ALPS interfaces that can be used to acquire global context about the ALPS execution instance, which can be used to efficiently initialize node local functionality, e.g., STCI RA Identifiers.

III. Design

The execution of loosely coupled large scale simulation workflows requires an execution model that is different from the one traditionally supported on leadership computing platforms as well as STCI, i.e., the execution of message passing applications (referred to as the *MPI execution model* in this document). First, the runtime must support the execution of many tasks within a job, whereas the MPI execution model typically requires execution of a few tasks within a job. Because the runtime must start/finalize more tasks, it is necessary to deploy a "persistent" runtime infrastructure so the many tasks composing a job can be efficiently deployed and terminated. As a result, this also requires a two-level resource management model for which the first level assigns resources to a job, and the second level to a task within a job (e.g., a task may require less nodes than the pool of nodes allocated to the associated job). Furthermore, based on the user's requirements and the tasks to execute, the runtime may have to co-locate different tasks on the same compute nodes, whereas the MPI execution model typically assumes that only one task can be mapped to a given compute node. Finally, because many tasks are executed instead of a few for the MPI execution model, the runtime must provide a set of tools that allows users to query the state of their overall job (set of launched tasks), as well as any task within a job (particular task). Figure 2 gives an example of how STCI agents are deployed on HPC platforms when a loosely coupled large scale simulation is executed.

These capabilities were not supported by the version of STCI that was strictly focused on the MPI execution model.

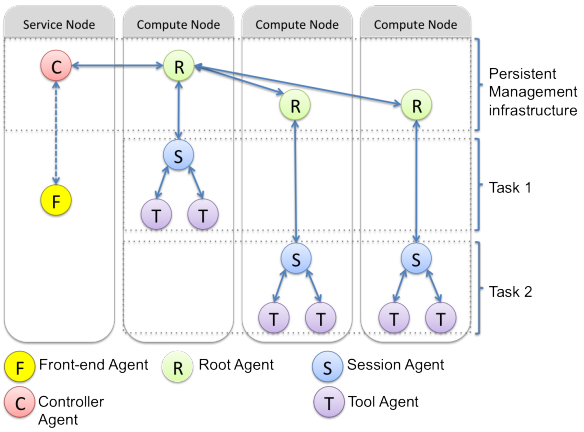

Figure 2: Example of a mapping of the agents required for the execution of a loosely coupled simulation workflow

To accommodate this loosely coupled execution model, STCI required a set of extensions that can be summarized as follows:

1) Persistent runtime infrastructure.
2) Two-level resource management for user jobs.
3) The ability to map multiple tasks on the same node (co-location).
4) Tools to query for specific job/tasks statuses.

To illustrate the difference between the two execution models, Figure 3 and Figure 4, respectively presents the pseudo-code of the MPI execution model and the loosely coupled large scale simulation execution model. For simplification, we assume that for the MPI execution model, a job is always composed of a single task; whereas a job is composed of many tasks for the loosely coupled model. As

```
1. Deploy job FE
2. Deploy CTRL
3. Deploy RAs
4. Deploy SAs
5. Deploy TAs
6. Wait for the job to end (TA termination – application driven)
7. Termination of SAs
8. Termination of RAs
9. Termination of CTRL
10. Termination of job FE
```

Figure 3: Pseudo-code for the MPI execution model

illustrated in Figure 4, the execution model associated with loosely coupled large scale simulations is quite different in nature: several types of FEs are required (job management versus task management), and some agents (i.e., CTRL and RAs) need to be persistent in order to share them between different tasks.

To add the capability to control the task's processes on compute nodes, STCI needs to place infrastructure agents onto the available resources. These infrastructure agents

```
1. Initiate job
    1−A. Deploy job FE
    1−B. Deploy CTRL
    1−C. Deploy RAs
2. for i in {i..<number of tasks>} ; do
    2−A. Deploy task FE
    2−B. Connect to CTRL
    2−C. Deploy SAs through existing RAs
    2−D. Deploy TAs
    2−E. Wait for the task to end (TA termination − application driven)
    2−F. Termination of SAs
    2−G. Termination of task FE
3. Wait for all tasks to terminate
4. Terminate job
    4−A. Deploy FE (that will orchestrate the overall job termination
    4−B. Termination of RAs
    4−C. Termination of CTRL
    4−D. Termination of job FE
```

Figure 4: Pseudo-code for the loosely coupled large scale simulation execution model

need to be available throughout the entire lifetime of the resource allocation. Since STCI already provides a set of infrastructure agents (RAs and SAs) and the capability to interact with the native process management system (i.e., ALPS), it seems natural to add the functionality to the existing design. It is sufficient to have a single persistent agent on a compute node. The RA is already part of the process management and will be part of the persistent management infrastructure (e.g., see Figure 5).

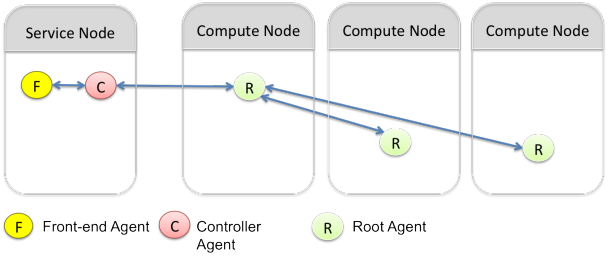

Figure 5: Mapping of the infrastructure agents

With a persistent infrastructure in place STCI can use the existing launcher framework to instantiate new SAs and TAs on the available resources. This opens up the possibility to implement very flexible mapping policies and the ability to give the user better control over the placement of a single task or processes within a task. For example, the user could decide to over-subscribe and run several collaborative tasks on the same physical node, i.e., selectively co-locate tasks.

Since a task has no active FE, STCI needs to provide a way to monitor its progress. Since the CTRL is available throughout the entire job allocation (the CTRL is a persistent agent), the CTRL has the responsibility of keeping track of task states. A task can have multiple states: a) launching, b) running, c) finished, d) failed and e) canceled. A separate FE can query for task status.

To keep track of the states of a task, the infrastructure agents specific to that task have to report the state changes to the CTRL. For example, the CTRL hands off a task to the launcher, changing its state to *launched*. The launcher sends launch requests to the persistent RAs that are part of this particulars task's mapping. After the RAs have started the SAs and TAs of the task and all participating agents are locally reported as *running*, a message is sent to the CTRL by the corresponding SA. When the CTRL received the messages for all SAs, the task can be globally assumed as *running*. A similar mechanism is used to monitor the shutdown of tasks

IV. Implementation

This section gives details about the runtime support for the execution of loosely coupled large scale computational work-loads, as well as changes that were made to the existing runtime system to accommodate the proposed design, i.e., adding the loosely coupled execution model to STCI.

To support the proposed design a set of new FEs were developed, which can be divided into three categories: a) starting and stopping the persistent runtime services for a given job, b) starting and stopping tasks (end-user applications), and c) querying for the status of jobs and tasks.

The `stcistart` command is executed on the service node to start up the CTRL related to the job. The CTRL starts the RAs on the compute nodes allocated to the job; the RAs then connect back to the CTRL. The CTRL and the RAs form the "control task". After the control task is established `stcistart` returns a unique identifier (ID) to the user. That unique ID can be used by users for querying the job's status. Moreover, this unique ID is assigned to the session associated to the job. Ultimately, the ID and the session allow users to control and monitor the execution of jobs. We will present later in this section how a similar capability is available for tasks, giving to users a unique and unified method for the control and monitoring of both jobs and tasks within a job.

The `stcistop` command is used to shut down the control task. After the execution of `stcistop` the runtime environment is shut down, i.e., all agents are terminated and all allocated resource are freed.

The `stciexec` command is used to start tasks within a job, connecting to the CTRL specific to the job. Once the connection to the CTRL is established, the task information is transferred to the CTRL, which starts the various SAs and TAs on the requested/available nodes. The TAs are the end-user executable to run on the remote resources, e.g., MPI application. As with the `stcistart` command, a unique ID is returned to the user to enable control (e.g., kill the task) and monitoring (i.e., query the task's status).

The `stcikill` command is used to terminate the given task within a given job. The `stcilist` command connects

to the CTRL to query a list of the currently running jobs and tasks that are associated with a particular session. The command can also be used to lookup a specific job and query all the tasks contained in the job.

The `stciwait` command provides support for selectively waiting for the termination of tasks within the job. For that, the user can specify a list of IDs; the `stciwait` then waits until all the tasks associated with these IDs terminate. Users may also specify the "any" flag, in which case, the command blocks until any of the listed tasks in the job terminate.

The CTRL is responsible for launching processes on the compute nodes and gathering the output of the applications. To establish a persistent RA network, the task startup was separated into two distinct phases. During the first phase, STCI's capabilities are used to deploy the RAs, by interacting with the native Cray/ALPS task launcher (`aprun`). After the persistent management layer is started, the CTRL is waiting for a FE to connect in order to either submit a new job, query for status information or shut down the persistent infrastructure. In response to the request to launch a new task, the second phase of the task startup is performed during which SAs and TAs are deployed. This is done using the process management capabilities provided by STCI (`stciexec`).

In order to support co-location of applications, we had to modify the STCI policy for task and job mapping on allocated resources. As presented in the previous section, STCI assumed a one-to-one relationship between the job mapping and task mapping, meaning that the same set of resources is used and that no persistent infrastructure needs to be mapped on available resources. The mapping algorithm used in the second phase was adjusted to support co-locating tasks By adjusting the mapping algorithms used during the second phase, it is possible to co-locate tasks (taking full benefit of the modularity of the STCI architecture): the mapping algorithm no longer assumes exclusive use of a given compute node by a single task but places all the tasks on available compute nodes based on the user's requirements, potentially sharing compute nodes between tasks within the same job.

Previously the task shutdown sequence was tailored for supporting the MPI execution model; no persistent infrastructure was required, all agents implicitly finalized as soon as the tasks composing the job (which were executed concurrently) terminated. For the loosely coupled execution model case, the persistent infrastructure can only terminate upon explicit notification from the user, via the `stcistop` command. This policy difference has been implemented by extending the STCI launcher and through the implementation of new agents specific to the support for the execution of loosely coupled simulations.

V. Experiments

To evaluate the proposed runtime we conducted a set of experiments to assess the performance of the enhanced version of STCI and compare it to the performance of running the tests solely with the native process launcher ALPS. All experiments where run on a Cray XK6 (Table III), *Chester*, system at the Oak Ridge Leadership Computing Facility. The system configuration for the *CHESTER* machine is detailed in Table III. This is a 80 node Cray XK6 development platform with a Gemini interconnect.

All tests used a MPI application, `mpisleep`, that accepts as argument the time to delay (N_{sleep}) using the standard C library `sleep()` function. The program does the following: initializes the MPI library, prints MPI rank, hostname and sleep time, sleeps for N_{sleep} seconds, and then finalizes MPI and exits.

Experiment-I "throughput": The first experiment evaluates the throughput of the enhanced version of STCI. The experiment measures the time it takes to run 100 iterations of `mpisleep` with no wait time, i.e., $N_{sleep} = 0$, and was repeated 10 times. Figure 6 shows the average time for the ALPS (794.03s, $\sigma = 1.8773$) and STCI (43.92s, $\sigma = 1.4847$) experiments to execute the 100 `mpisleep 0` tasks. The ALPS case loops over calling `mpisleep 0`, measuring time before and after the loop. In the STCI case, time begins before the `stcistart` and time ends after the `stcistop`, which includes all the `stciexec` launches for the 100 `mpisleep 0` followed by a `stciwait` to block until each task completes. Note, the STCI tests also include a 10 second delay between `stcistart` and the `stciexec` loop to ensure the control infrastructure is up and ready.

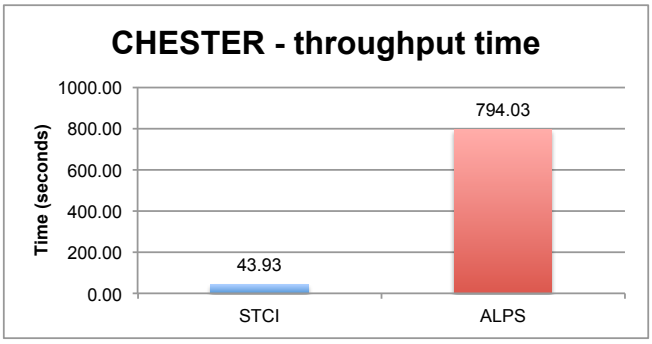

Figure 6: Throughput test with 1 node and 100 tasks, each task using a sleep time of zero (0) seconds. Times shown in seconds, and STCI includes addition 10 second sleep after stcistart.

Experiment-II "end2end": The second experiment compares STCI with the native application launcher, i.e., ALPS. For this, we measure the run time for an application using both ALPS and the enhanced version of STCI. In contrast to the throughput experiment a random distribution of wait times (N_{sleep}) is used to for the 100 tasks, with STCI

	Experiment-I: "throughput"	Experiment-II: "end2end"	Experiment-III: "memcost"
Application	`mpisleep`	`mpisleep`	`mpisleep`
N_{sleep}	0 sec.	$Random > 0$ sec.	420 sec.
$N_{iterations}$	100	100	100
N_{runs}	10	12	15

Table I: Summary of experiment parameters.

co-locating tasks. A series of sleep times was generated using a uniform distribution and both ALPS and STCI tests were run using this series for N_{sleep}. Figure 7 shows the results for running 100 tasks on 2 nodes with STCI and ALPS. Note, the STCI tests also include a 10 second delay between `stcistart` and the `stciexec` loop to ensure the control infrastructure is up and ready.

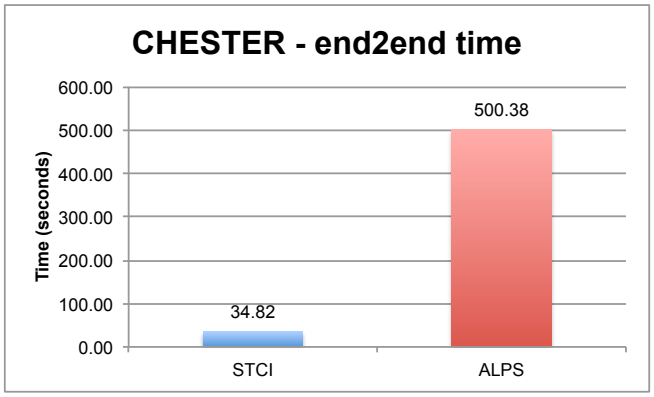

Figure 7: End-to-end test with 2 nodes and 100 tasks, each task taking sleep time from a uniform distribution of value. Times shown in seconds, and STCI includes addition 10 second sleep after stcistart.

Experiment-III "memcost": The last experiment looks at the overhead and resource consumption on the login node. A series of tasks are run using both environments (STCI and ALPS) with the memory consumption measured on the login node over the course of the experiment. The login node has 16 Gb of total memory.

Figure 8 shows the memory utilization when running 32 `mpisleep` commands concurrently. The data was gathered using `vmstat -s` and graphs the value of global "used memory" for the system over the duration of the test runs. The times are normalized in order to show a side-by-side comparison of the memory utilization. A total of 20 runs were performed; 5 runs with the highest Std.Dev for total system used memory were removed from both the ALPS and STCI data sets. The graph in Figure 8 shows the 15 remaining runs. The graph shows the average for each step in the execution to illustrate system memory usage over the duration of the experiment.

The `mpisleep` executable was run with a sleep argument sufficiently long (420 seconds) to allow for startup of all tasks and to gather memory usage with all tasks still running. A brief sleep was placed between each command to spread the load out over time to avoid excessive load on the login node. In order to run 32 (distinct) tasks concurrently, we used 32 nodes in the ALPS case and 2 nodes in the STCI case. There are 16 cores per node so this places 1 task on each core of the 2 nodes for the STCI case. In the ALPS case, due to each task being a distinct `aprun` invocation, the tasks are spread over 32 nodes. This spread is due to ALPS resource policies, which were discussed previously in Section II.

The increased memory usage in the ALPS case is due to the fact that the 32 concurrent `aprun` commands are run as background tasks to avoid blocking within a script. In contrast, the STCI case requires the initial `stcistart` to launch the CTRL and persistent RAs on the compute nodes. Then each `stciexec` invocation sends the task launch request to the CTRL, disconnects and terminates. The reuse of a persistent CTRL process with frontend requests result in the relatively flat memory usage shown in the graph for the STCI tests.

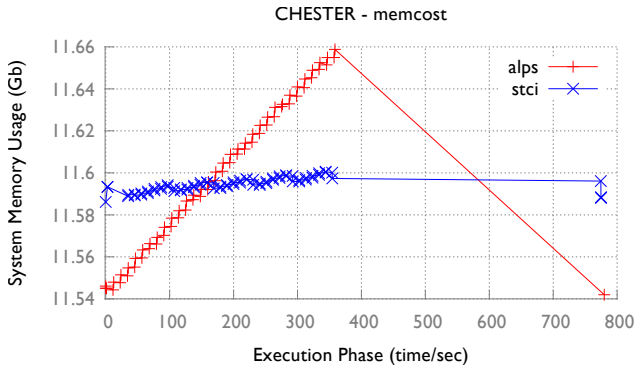

Figure 8: Used memory on login node when running series of concurrent tasks via ALPS (`aprun`) and STCI (`stciexec`).

Another distinction worth noting is that the `stciwait` command provides the ability to wait on "any" active task to complete in order to break the blocking wait command (as presented in Section IV). This allows for performing additional tasks at intermediate stages, e.g., starting new

	STCI	ALPS
Num. runs	12	12
Minimum	25.59	494.67
Maximum	49.28	505.04
Mean	34.8225	500.3841
Median	33.405	501.02
Variance	77.5982	9.3344
Std.Dev.	8.8089	3.0552

Table II: The "end2end" statistics over 12 runs using 2 nodes and 100 tasks per run. Note, the STCI tests include a 10 second delay after the `stcistart` to ensure Controller is ready.

tasks. This is achieved without having to force the FE (`stciwait`) to poll, instead the CTRL is able to register an event and notify the FE when the job is complete.

```
 # Get Baseline
log_meminfo
NSEC=420

for i in {1..32} ; do
    log_meminfo
    aprun −n 1 mpisleep $NSEC
    log_meminfo
done
 # Wait on all to finish
log_meminfo
wait
log_meminfo
```

Figure 9: Pseudo-code for the "memcost" experiment using ALPS (aprun).

VI. Conclusions and Future Work

Because of their complexity and scale, leadership HPC systems, such as the Cray systems at ORNL, enforce strict usage constraints to users in terms of job and resource management. These constraints greatly limit the options available to users regarding the execution of their applications, which ultimately also limits the design and implementation of these applications.

The proposed work addresses a gap between user needs and characteristics of HPC systems. Our main contributions in this paper can be summarized as follows:

- The proposed runtime system enables efficient execution of large scale loosely coupled workloads on leadership HPC systems, mainly by decreasing the job start-up time.
- The proposed architecture extends the execution model of leadership HPC systems, by (i) enabling better control over job placement and execution (enhanced task management environment); (ii) enabling finer grain interactions with the system software for the management of many loosely coupled workloads (enhanced system commands to launch, wait and terminate user jobs); (iii) extending the execution model of leadership HPC system by supporting time-sharing of compute nodes in a space-shared allocation,
- We evaluated the prototype's support for service node resource overheads and time to completion of synthetic workloads running on leadership platforms at ORNL. The experiments showed the prototype offers better memory efficiency and lower time to solution for the synthetics applications tested.

```
 # Get Baseline
log_meminfo

 # Start CTRL and RAs
log_meminfo
stcistart −N $PBS_NUM_NODES
log_meminfo

 # Get SesionID
SID=`stcilist␣-T`
NSEC=420

for i in {1..32} ; do
    log_meminfo
    stciexec −S $SID −np 1 mpisleep $NSEC
    log_meminfo
done

while [ count($live_jobs) > 0 ] ; do
    live_jobs=get_job_string()

    log_meminfo
    stciwait −S $SID −−any −j $live_jobs
    log_meminfo
done

 # Stop CTRL and RAs
log_meminfo
stcistop −S $SID
log_meminfo
```

Figure 10: Pseudo-code for the "memcost" experiment using STCI (stciexec).

Current operating system and runtime research for exascale systems includes the concept of an *enclave* [15]. An enclave is used to abstract the environment required for the execution of scientific applications in order to provide control over application resilience, energy consumption, as well as increased scalability. This concept will drastically change the execution model of exascale systems since a

Component	Description
System	login nodes / 1 CPUs / 6 cores / 16 GB memory 80 compute nodes / 1 CPUs / 16 cores 2640 GB memory / 80 GPUs
Login nodes	1 CPUs / 6 cores / 16 GB memory
Login CPU	AMD Opteron(tm) Processor 23 (D0) / 6 cores / 2200 Mhz
Compute node	1 CPUs / 16 cores / 33 GB memory
Compute CPU	AMD Opteron(tm) Processor 6274 / 16 cores / 2200 Mhz
Compute GPU	Nvidia Tesla K20X
Network	10 Gbps Ethernet Cray Gemini 3-D Torus 20 GB/s
OS	Cray Linux Environment (CLE) (XTOS Ver 4.2.34) (2.6.32.59-0.7.1_1.0402.7496-cray_gem_c x86_64)

Table III: System configuration for the Cray XK6 *CHESTER* platform.

job is executed using multiple enclaves; enclaves that are instantiated within a user's allocation. Because STCI has been extended to offer capabilities for the implementation of persistent system service (e.g., persistent RAs), STCI is a candidate runtime environment for the implementation and support of enclaves.

Furthermore, the proposed solution addresses some of the major limiting factors for the design, implementation and execution of scientific simulations that are not typical message passing applications. By extending the execution models, scientists have the opportunity to extend and/or modify their application focusing on the science instead of the constraints imposed by leadership HPC systems. Therefore, this work could enable the execution of scientific applications that were not good candidates so far because of conflicting constraints between an application's design and execution.

Moreover, the proposed runtime infrastructure, thanks to persistent services, enables the efficient execution of scientific workloads. This infrastructure could be further extended to improve application start-up by making the runtime infrastructure aware of the HPC parallel file system for better I/O throughput.

Finally, the proposed prototype being based on STCI, it would be possible to benefit from the STCI's fault tolerance capabilities that have been developed in the context of MPI applications at large scale. For example, STCI provides fault detection, notification and recovery mechanisms that could be leveraged in the context of loosely coupled applications by ensuring that even if compute nodes fail, the other running tasks are minimally impacted.

ACKNOWLEDGMENTS

The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This document describes activities performed under contract number De-AC0500OR22750 between the U.S. Department of Energy and Oak Ridge Associated Universities. All opinions expressed in this report are the authors' and do not necessarily reflect policies and views of the U.S. Department of Energy or the Oak Ridge Institute for Science and Education.

REFERENCES

[1] INCITE, "U.S. Department of Energy Innovative and Novel Computational Impact on Theory and Experiment (INCITE) Program: Frequently Asked Questions (FAQ)," *FAQ-13: Can I meet the computationally intensive criterion by loosely coupling my jobs?* (Last viewed: 10 Jan 2014) https://hpc.science.doe.gov/allocations/incite/faq.do#faq_13.

[2] G. Vallée, T. Naughton, S. Böhm, and C. Engelmann, "A runtime environment for supporting research in resilient HPC system software & tools," in *Proceedings of the First International Symposium on Computing and Networking – Across Practical Development and Theoretical Research (CANDAR)*. Matsuyama, Japan: IEEE Computer Society, Dec. 2013.

[3] I. Raicu, I. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, Nov 2008, pp. 1–11.

[4] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, "Toward loosely coupled programming on petascale systems," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 22:1–22:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413393

[5] S. Simmerman, J. Osborne, and J. Huang, "Eden: Simplified management of atypical high-performance computing jobs," *Computing in Science Engineering*, vol. 15, no. 6, pp. 46–54, Nov 2013.

[6] "PCP (parallel-command-processor) Source Code," (Last viewed: 1 May 2014) http://svn.nics.tennessee.edu/repos/pbstools/trunk/src/parallel-command-processor.c.

[7] A. Luckow, L. Lacinski, and S. Jha, "Saga bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, May 2010, pp. 135–144.

[8] A. Thota, S. Michael, S. X. T. G. Doak, and R. Henschel, "Tools to execute an ensemble of serial jobs on a cray," in *Cray Users Group Meeting (CUG2013), Napa Valley, CA*, May 2013.

[9] W. Elwasif, D. E. Bernholdt, A. G. Shet, S. S. Foley, R. Bramley, D. B. Batchelor, and L. A. Berry, "The design and implementation of the SWIM Integrated Plasma Simulator," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 2010, pp. 419–427. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/PDP.2010.63

[10] W. R. Elwasif, D. E. Bernholdt, S. S. Foley, A. G. Shet, and R. Bramley, "Multi-level concurrency in framework for integrated loosely coupled plasma simulations," in *9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2011), Sharm El-Sheikh, Egypt*, December 2011.

[11] "The DAKOTA project: Large-scale engineering optimization and uncertainty analysis," http://dakota.sandia.gov/.

[12] W. R. Elwasif, D. E. Bernholdt, S. Pannala, S. Allu, and S. S. Foley, "Parameter sweep optimization of loosely coupled simulations using the DAKOTA toolkit," in *15th IEEE International Conference on Computational Science and Engineering*, December 2012.

[13] J. M. Squyres and A. Lumsdaine, "The component architecture of open MPI: Enabling third-party collective algorithms," in *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, Eds. St. Malo, France: Springer, July 2004, pp. 167–185.

[14] *Cray XT^{TM} System Management*, S-2393-22 ed., Cray, Jul. 2009, uRL: http://docs.cray.com/books/S-2393-22 (Last visited: 28jan2011). [Online]. Available: http://docs.cray.com/books/S-2393-22

[15] R. Brightwell, R. Oldfield, A. B. Maccabe, and D. E. Bernholdt, "Hobbes: Composition and virtualization as the foundations of an extreme-scale OS/R," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '13. New York, NY, USA: ACM, 2013, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2491661.2481427