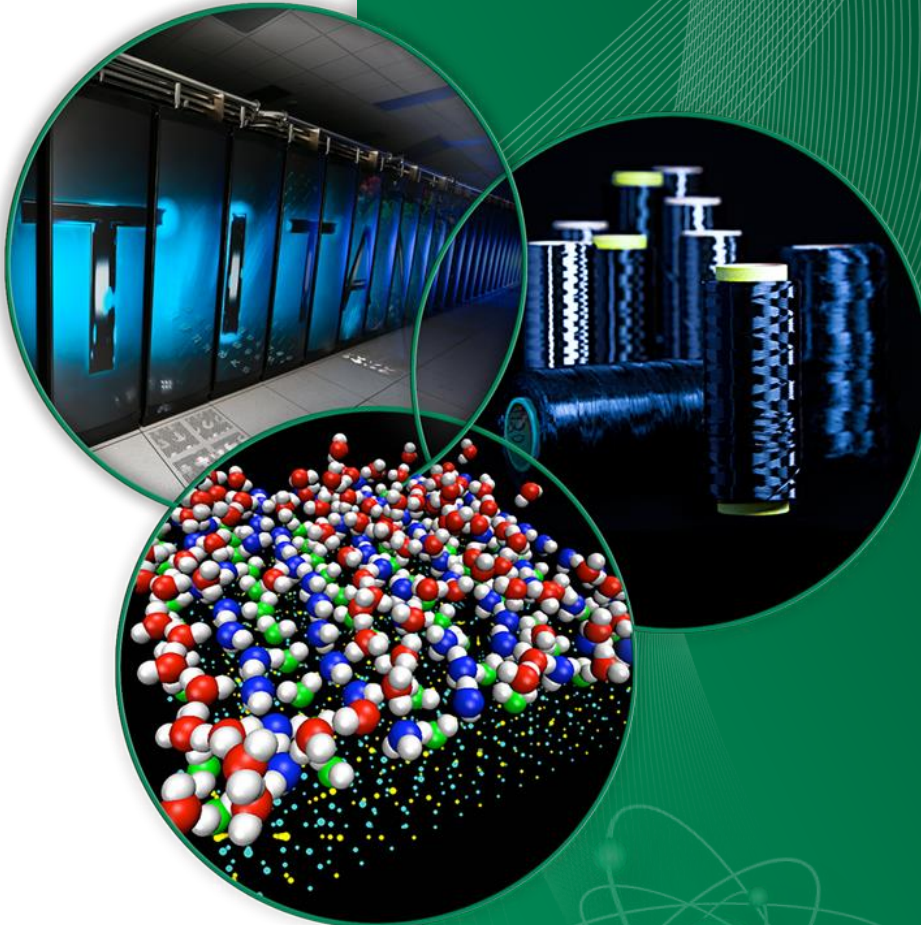
Toward Improved Support for Loosely Coupled Large Scale Simulation Workflows



Swen Boehm
Wael Elwasif
Thomas Naughton,
Geoffroy R. Vallee

Motivation & Challenges

- Bigger machines (e.g., TITAN, upcoming Exascale systems)
  - Environment targets coarse-grain, massively parallel executions
    - Relatively "heavy weight" tools for program startup, execution, and shutdown
    - "Few" active program dispatch instances on service nodes
- Growing use of large scale many-task computing:
  - Ensemble computing for Leadership class allocation
  - Genomics, bioinformatics, data analytics ...
  - Parameter sweep and optimizations (Industrial use)
- Runtime environment (RTE) is a crucial software component
  - Not supportive of this kind of workload

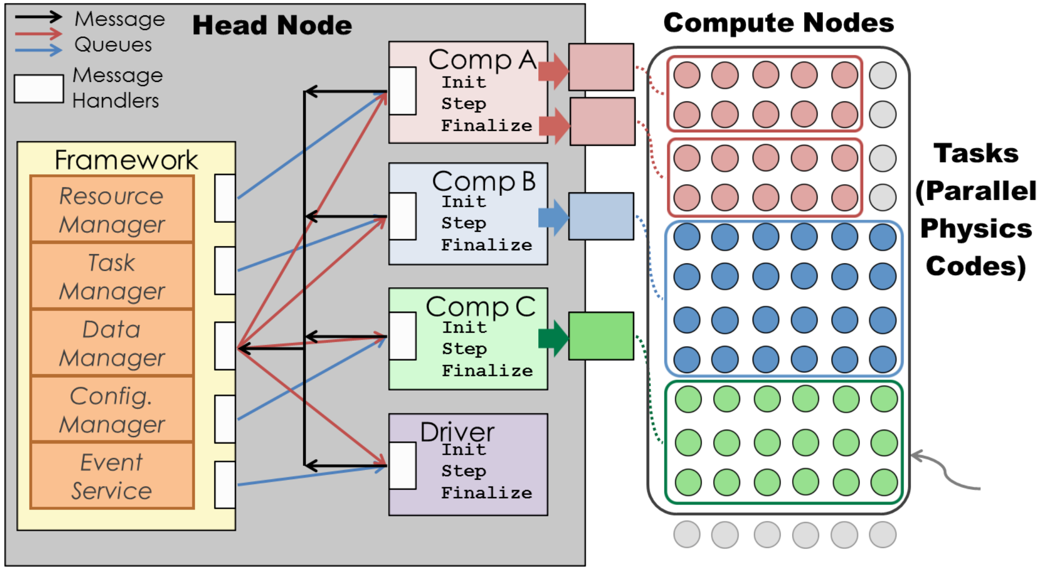
Can we provide user-level run-time environment to better support large scale loosely coupled workloads ?

Key Requirements

- Minimal (no?) incremental impact on service nodes as number of executing instances scales up.
- Low overhead for execution initiation, monitoring, and termination.
- Efficient resource utilization
  - Number of cores/node will only increase
- Current ALPS/aprun environment
  - Limits on number of concurrent `aprun` instances on service node
  - Relatively long startup/shutdown times per `aprun` invocation
  - Policy limits on node sharing.

Other tools

- Serial Tasks:
  - Relatively easy - rely on **`system()`** calls from compute nodes
  - BigJob, Parallel Command Processor (PCP) ..etc.
  - Cannot be extended to parallel tasks
- Parallel Tasks
  - Integrated Plasma Simulator (IPS)
  - Still uses **`aprun/mpirun`** under the hood
  - Need different runtime to use anything else

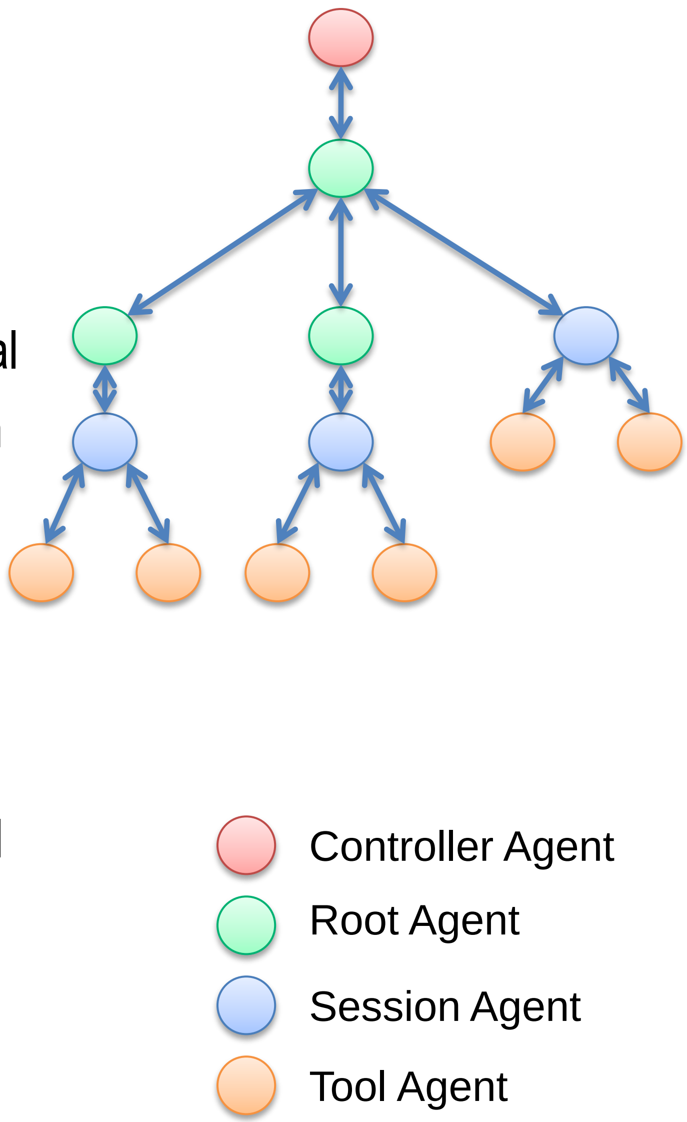


Scalable RunTime Component Infrastructure – STCI

- Goals
  - Scalable start-up and management of scientific simulations
  - Resilience/fault tolerance
  - *Ease the study and development of new system tools and/or applications for HPC*
- Key characteristics
  - User space modular architecture
  - Provide reusable components
- Lightweight front end tools
  - Task instantiation, monitoring, and termination
  - Better fit for handling many concurrent executing tasks.

STCI Architecture

- Agents
  - Instantiate both the STCI infrastructure and applications/tools
  - Different "types" of agents
    - *Frontend*: user frontend running on user's terminal
    - *Controller*: logical agent representing the job from a control point of view
    - *Root agent*: privileged agent for resource allocation; one per node; non-specific to a job
    - *Session agent*: local management of users' job; one per user and per node
    - *Tool agent*: instantiation of an application or a tool
- Topologies
  - Represent connections between agents
  - Examples: trees, meshes, binomial graphs

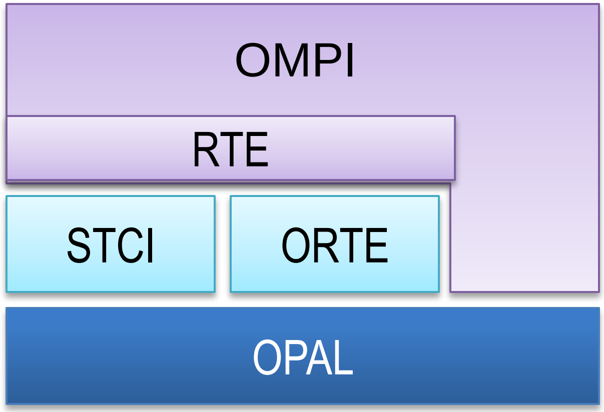


STCI Architecture (2)

- Launcher
  - Deploy a job by creating the necessary agents across the HPC platform
  - Two challenges
    - Scalable deployment method: by default, a tree-based topology
    - Method to create the required agents
      - Example: fork, ssh, ALPS
      - On Cray:
        - » Torque gives the list of target compute nodes
        - » ALPS is used to create the RAs
        - » then RAs create other agents
- Event system
  - Support for asynchronous execution model
  - Various progress models available: implicit or explicit progress

Alternate Runtime for MPI tasks on Crays

- Based on Open-MPI
  - Replace the default runtime (ORTE)
  - Benefit the RTE abstraction in Open-MPI
    - Out-of-band communications
    - Naming service
  - RTE mainly used for the deployment of MPI ranks
    - ***STCI communication substrates used during bootstrapping***
    - ***Open-MPI high-performance communication substrates once bootstrapping completed***

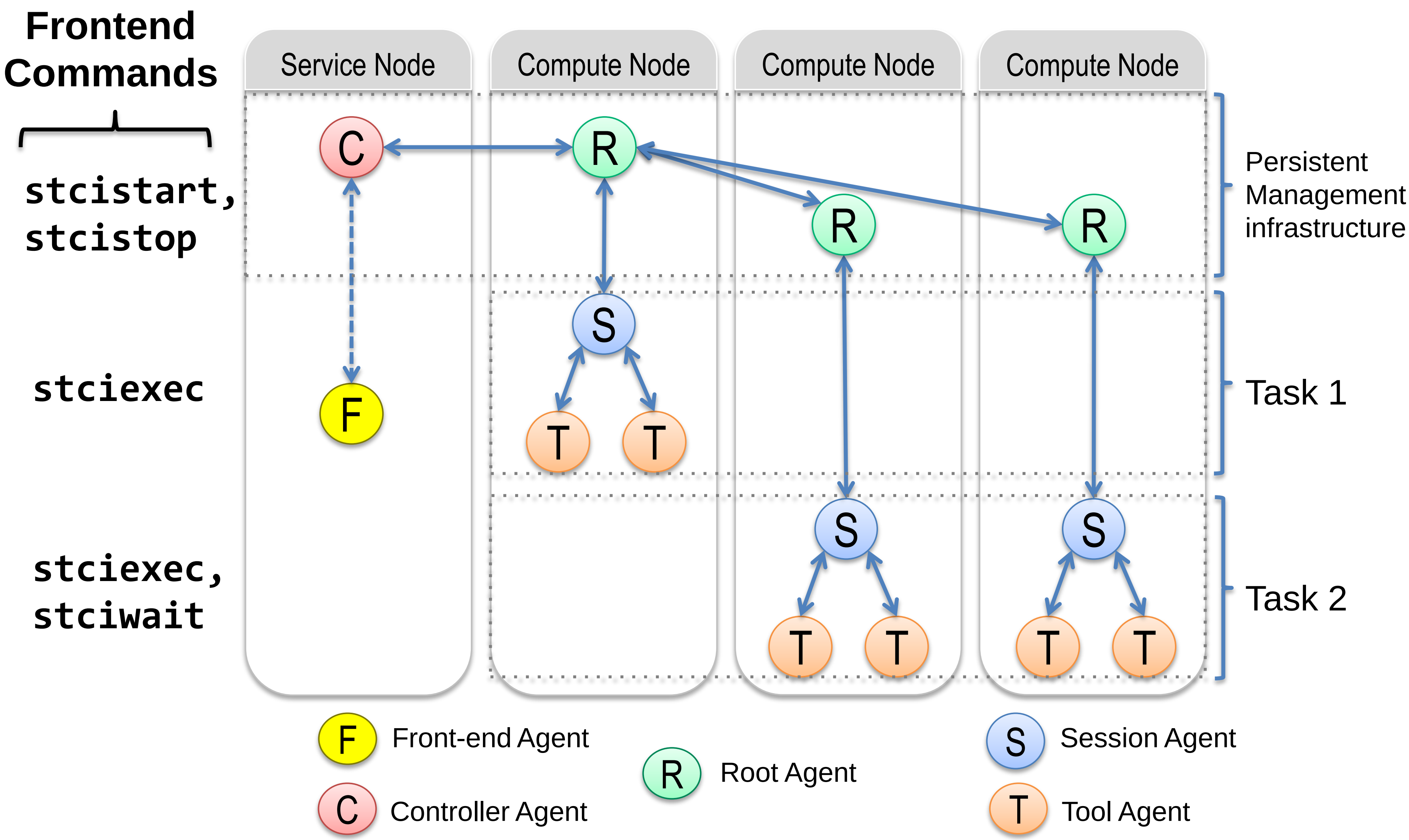


- Front end tools for task management
  - **`stcistart, stciexec, stciwait, stcikill, stcistop`**

STCI For Many Task Computing

- Original STCI supports a single startup-execute-shutdown cycle, for a single app invocation.
- Explicit STCI shutdown command
  - Keep STCI agents alive after tasks complete
- Add support for new frontend tools

STCI Many Task Overview



STCI Front End Commands

- Execute within a single batch allocation
- Targeting streamlined many-task management.
- Minimize impact on service node resources.
- Could be used directly, or as the backend for a smart workflow management interface.

STCI commands : `stcistart`

- Syntax **`stcistart -N #NNODES`**
  - **`#NNODES`:** Number of **STCI** managed nodes (e.g. **`$PBS_NUM_NODES`**)
- Only call to **`aprun`** in a STCI session
- Start STCI agents to support tasks on **`#NNODES`** compute nodes
- Returns: session id (**`sid`**) for use in future STCI commands
  - Returned as stdout string
- Blocking command
  - return after all STCI infrastructure agents have been initiated

STCI commands : `stciexec`

- Syntax : `stciexec -S sid -np #nprocs <prog> [args]`
  - **`sid`** : session id returned from stcistart.
  - **`#nprocs`** : number of ranks in MPI task
- Start **<prog>** on **#nprocs** free cores,
- Fail if not enough free cores are available
- Non-blocking:
  - Returns **`STCI`** task id - **`tid`** immediately upon successful launch.
  - Task id returned as stdout

STCI commands : `stciwait`

- Syntax : **`stciwait -S sid [-any] tid[,tid]*`**
  - **`sid`** : session id returned from **`stcistart`**
  - **`tid`** : task id returned from a prior call to **`stciexec`**
- Wait for one or more STCi tasks to finish
- Default: blocking wait for all **`tid`**’s to terminate
- **`-any`** causes return after one or more tasks finish
- Return immediately if all tasks have finished.
- Print list of **`taskid:retval`** on stdout

Other STCI Commands

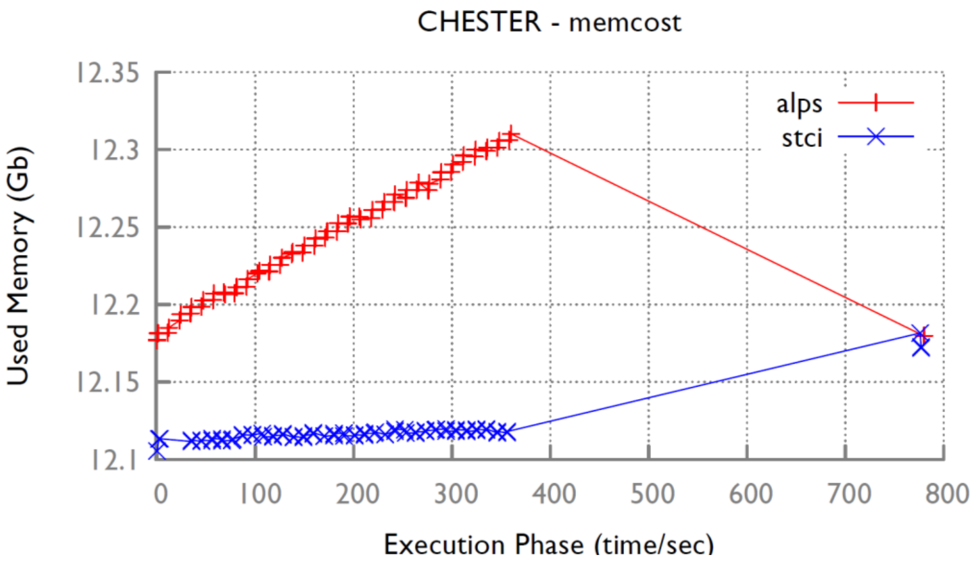
- **`stcikill –S sid tid`**
  - Kill task **`tid`** started under session **`sid`**
- **`stcilist –S sid`**
  - List status of all tasks for session **`sid`**
- **`stcistop –S sid`**
  - Terminate session **`sid`** and all its remaining tasks

(Very) Preliminary Results using STCI

- Tests run on Chester
  - development Cray XK7 at ORNL
  - 80 compute nodes * 16 cores/node
- Using a single core user task
  - Support for user tasks with **np > 1** currently in testing
- Three tests:
  - Impact on service node
  - Task initiation/shutdown overhead
  - Node sharing between distinct MPI tasks

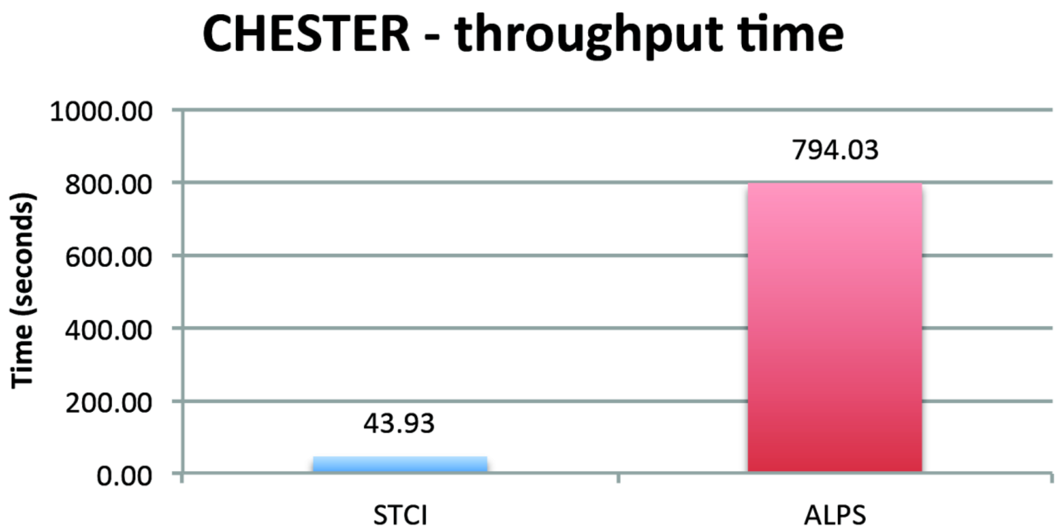
Impact on service node memory

- 32 concurrent mpisleep tasks, launched 10 sec apart.
- Using background aprun (red) and stcistart/stciwait (blue)
- Sleep time chosen to have all tasks concurrently active



Task Initiation/Termination overhead

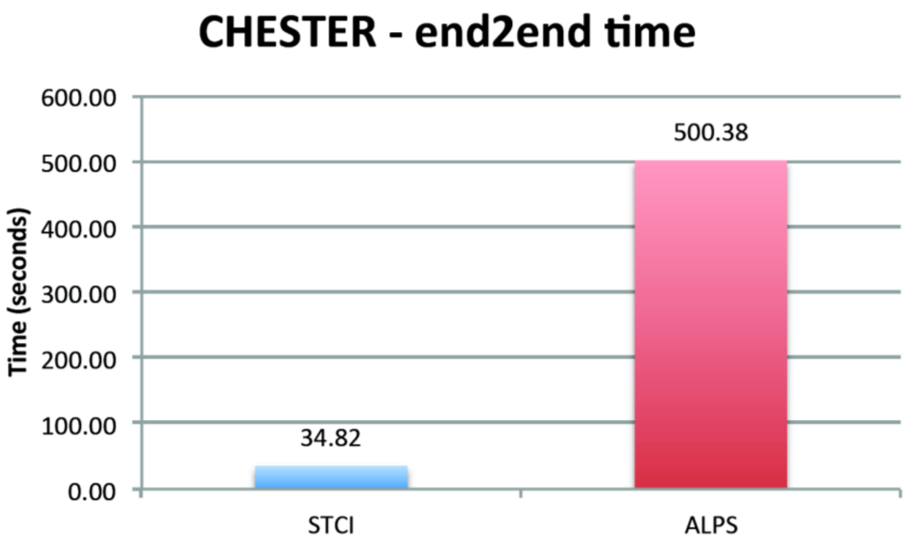
- 100 **`mpisleep`** 0 tasks executed sequentially
- Repeated 10 times
- STCI time include **`stcistart`** ,**`stcistop`**, and 10 sec delay between **`stcistart`** and 1st task



Average time for 10 runs

End-to-End Execution time

- 100 `mpisleep` tasks uniformly distributed in `[1 - 4]` sec.
- Repeated `12` times on `2` Chester nodes (16 cores/node)
- Total **STCI** time includes **`stcistart`**, **`stcistop`**, and 10 sec delay.
- Node sharing among separate MPI tasks
- Ongoing tests using **`n>1`** MPI tasks



Average time for 12 runs

Task Output Management

- Controller logs output from tasks launched via ‘stciexec’
  - STDOUT -> stci-stdout-sid-<SID>.log
  - STDERR -> stci-stderr-sid-<SID>.log
- The individual task output can be extracted using a post-processing script, **`pyramid-chopjob.pl`**

```
$ ./pyramid-chopjob.pl  < stci-stdout-sid-31683.log
Created: /tmp/stdout-stcijob-1.log
Created: /tmp/stdout-stcijob-2.log
…
…
Created: /tmp/stdout-stcijob-14.log
$ cat stdout-stcijob-14.log
( 0) My rank is: 0  sleeping 3 sec. (host=nid00078)
```

Combined output file from all tasks

Output Delimiters

Task Identifiers

```
$ head  /tmp/stci-stdout-sid-31683.log
==14-start==
(    0) My rank is: 0  sleeping 3 sec. (host=nid00078)
==14-end==
==28-start==
(    0) My rank is: 0  sleeping 2 sec. (host=nid00079)
==28-end==
  ...
```

Conclusion

- STCI enables lightweight, fast task management infrastructure for ALPS based Cray systems.
- Open MPI-based flexible runtime environment.
- Future work
  - Complete support for n>1 MPI tasks
  - User-controlled placement of tasks
  - Fault tolerance and recovery
  - Explore in-memory caching of binary image on compute nodes
  - Integration with more sophisticated front-end tools (e.g. the IPS).

Acknowledgment

- Individuals that contributed to the STCI project, including Richard Graham, Wesley Bland, Joshua Hursey, Christos Kartsaklis, Rainer Keller, Gregory Koenig, Pavel Shamis and Chao Wang.
- 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.

Thank You

Questions?

Wael Elwasif : *elwasifwr@ornl.gov*