RADICAL-Cybertools

Research in Advanced DIstributed Cyberinfrastructure & Applications Laboratory (RADICAL)

http://radical.rutgers.edu &
http://radical-cybertools.github.io
Why is it a challenge...

... to build “Scalable, Extensible and Flexible” tools?

- Every HPC system & environment is different, if not unique:
  - Interoperability across distinct interfaces and semantics:
    - Batch queue systems, data access/movement etc.
    - Portability: Software environments
- Traditional use of HPC: Optimize “ecosystem” for single large job.
  - Missing abstractions, gaps in capabilities when number of concurrent > 1.
  - Uniform and resource management models
- Balance between specific requirements and distinct solutions?
  - Generality and extensibility are often orthogonal to performance.

Need tools that provide balance generality, extensibility and performance.
The Case for “Building Blocks” for Workflows

- Initially “Monolithic” Workflow systems with “end-to-end” capabilities
  - Workflow systems were developed to support “big science” projects.
  - Software infrastructure was “fragile”, unreliable, missing services

- Workflows aren’t what they used to be!
  - More pervasive, sophisticated but no longer confined to “big science”
  - Importance of applications based upon “more than a single task”

- Extend traditional focus from end-users to workflow system/tool developers!
  - Building Blocks (BB) permit workflow tools and applications can be built.

- Diverse workflow requirements and use cases
  - Diverse “design points”; unlikely “one size fits all” paradigm
  - Balance between proliferation of workflow systems and single system

- Need for agile, experimental and often unique workflows
  - Run many times, or many users: amortisation of development overhead
  - End-users develop interfaces, not performance critical components.
Introduction to RADICAL-Cybertools
RADICAL’s Laws of CI (With apologies to Zawinski*)

- **RADICAL’s First Law**: Every tool “shims” to submit to distinct middleware (such as batch-queue systems) and claim *interoperability*.

---

*Zawinski’s Law*: Every program attempts to expand until it can read *mail*. Those programs which cannot so expand are replaced by ones which can.
RADICAL’s Laws of CI (With apologies to Zawinski*)

- **RADICAL’s First Law**: Every tool grows “shims” to submit to distinct middleware (such as batch-queue systems) and claim interoperability.

- **Corollary**: Interoperability should be provided explicitly at the lowest level possible (Principle of Subsidiarity)

*Zawinski’s Law: Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can.*
RADICAL’s Laws of CI (With apologies to Zawinski*)

- **RADICAL’s First Law**: Every tool grows “shims” to submit to distinct middleware (such as batch-queue systems) and claim interoperability.

- **RADICAL’s Second Law**: Every application execution tool eventually claims to become a workflow system.

*Zawinski’s Law: Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can.*
RADICAL’s Laws of CI (With apologies to Zawinski*)

- **RADICAL’s First Law**: Every tool grows “shims” to submit to distinct middleware (such as batch-queue systems) and claim interoperability.

- **RADICAL’s Second Law**: Every application execution tool eventually claims to become a workflow system.

- **Corollary**: To prevent proliferation of workflow systems we need to determine common components across (most) workflow systems.

*Zawinski’s Law*: Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can.
RADICAL-Cybertools

- Four Layers:
  - L4: Application
  - L3: Workload Management (WLMS)
  - L2: Task Run-time (TRS)
  - L1: Resource Access Layer
RADICAL-Cybertools

- Four Layers:
  - L4: Application
  - L3: Workload Management (WLMS)
  - L2: Task Run-time (TRS)
  - L1: Resource Access Layer

- Abstractions & Building Blocks:
  - L1: RADICAL-SAGA  Distributed job submission & standard interface
  - L3: RADICAL-WLMS, Ensemble Toolkit

- Cross-layer: RADICAL-Analytics
RADICAL-Cybertools: “Building Blocks” for Workflows

- A “laboratory” for testing ideas while supporting production grade workflows and workflow tools.
- Stand alone, vertical integration and horizontal extensibility
- Integrated with existing tools:
  - Swift, Firework, PanDA, Binding Affinity Calculator...
  - Need “faster” start, “scalable” (more tasks) and “better” (resource utilization on platforms).
- Novel tools and libraries:
  - ExTASY, Replica-Exchange..
From “Big Science” to the Long Tail of Science

The ATLAS experiment at the Large Hadron Collider in Switzerland uses SAGA in conjunction with PanDA as a workload management system.

The Super-Kamiokande project searches for neutrinos to understand the creation of matter in the universe. It uses SAGA to simulate collisions on HPC clusters.

RADICAL-Pilot is being used by Chemistry researchers to support large-scale and multidimensional replica exchange simulations on supercomputers.

Nektar++ is a finite element package which uses SAGA in the backend to submit jobs to a variety of clusters. It tackles problems such as modeling air flow around automobiles.

Researchers at UCL London are using RADICAL-Pilot to advance understanding of HIV drug resistance and make personalized treatment possible.

RADICAL-Pilot supports multi-physics and coupled simulations, such as hybrid CFD-MD simulations to understand Couette Flow, as well as PBM-DEM simulations for Cybermanufacturing.
RADICAL SAGA: Schematic

http://radical-cybertools.github.io/saga-python
SAGA: A Standardized Interoperability Layer

• SAGA – Simple API for Distributed (“Grid”) Applications:
  – Allows access to different middleware / services through Python implementation of Open Grid Forum GFD.90.
  – Also unified semantics across middleware, backend plug-ins (“adaptors”).

• HOW SAGA is Used?
  – Uniform Access-layer to DCI, e.g, XSEDE,
  – Build tools, middleware services and capabilities
    • Pilot-Jobs, “Workflow systems”, science gateways and web portals
  – Domain-specific (distributed) applications, libraries and frameworks
Pilot Abstraction: Schematic

A system that generalizes a placeholder job to allow application-level control of acquired resources via a scheduling overlay.
Pilot-Abstraction: Overview

- **Working definition:**
  - “.. defined as an abstraction that generalizes the reoccurring concept of utilizing a placeholder job as a container for a set of compute tasks”

- **Advantages of Pilot-Abstractions:**
  - Decouples workload from resource management
  - Flexible Resource Management
    - Enables the fine-grained (ie “slicing and dicing”) of resources
    - Tighter temporal control and other advantages of application-level Scheduling (avoid limitations of system-level only scheduling)
  - Build higher-level frameworks without explicit resource management
Executing Tasks Using RADICAL-Pilot

A simplified “step-by-step” animation
An application is comprised of workload. A user of the application wants to utilize a remote resource via radical.pilot.
The user is responsible for creating a **PilotManager** and a **UnitManager**.
Those managers spawn additional components for pilot and unit lifetime management:

- pilot launcher
- umgr scheduler
- umgr input staging
- umgr output staging
Multiple instances for each component can co-exist for load sharing and scaling.

Components communicate via ZMQ message queues.

RP is configurable in great detail, but the default configuration works out of the box.
The user describes **Pilots** and **ComputeUnits** to be created:

- `rp.ComputePilotDescription`
- `rp.ComputeUnitDescription`

Those descriptions contain requirements like core counts, unit executable, env settings, etc.

They are passed to the respective managers which create **Pilots** and **ComputeUnits (CUs)**.
The components communicate *Pilot* and *CU* handles via *ZMQ message queues.*
Pilots are passed to the Pilot Managers' **PilotLauncher** component, which prepares a **job** for submitting the **Pilot**:

- connect to the resource
- stage RP software stack
- create batch submission script
- submit the job to the **batch system**
Eventually, the batch system will run the **pilot job** on the target system: the **bootstrap** process is placed on the **landing node** (compute or MOM node). The other **compute nodes** are accessible for the **pilot** to use.
The bootstrapper installs the RP software stack and spawns the pilot components (using nodes as needed):

- input / output staging
- unit scheduler
- unit executor

The pilot is now ready to receive and execute compute units.
We will now look into the unit execution path:

When a UnitManager receives new requests to execute CUs, its scheduler will assign them to an available Pilot, and the input stager will transfer the CU's input data.
The CUs will then be sent to the Pilot which will again stage data if needed, schedule the units on a subset of compute cores, and pass them on to the executor(s).

Note that CUs can be submitted at any time -- pilots are utilized as they become available.
The executors will run the CUs on the compute nodes.

The executor's performance is optimized for high throughput, ensuring high system utilization.

RP can mix MPI / non-MPI jobs, GPU support is coming soon.
Once completed, the CUs are collected by the Pilot's output staging component, are then passed back to the unit manager's output staging, and finally the application is notified about their completion.
RADICAL-Pilot Architecture

https://github.com/radical-cybertools.radical.pilot
Ensemble Toolkit: Overview

- Ensemble ToolKit (EnTK) to support the large-scalable execution of ensembles

- Python toolkit provides **constrained flexibility** to compose ensemble-based “workflows”.

- Promotes **ensemble execution patterns** with API and performance runtime system, independent of simulation and analysis kernels.

- Simplifies application development.
  - Separates description from execution
  - Lowers user responsibility (execution)
1. Select required **execution pattern**. Create pattern object with required parameters
2. Select **kernel plugins** to fill the various stages of the execution pattern to create the workload
3. Create **resource handle** with resource details of the HPC system.
4. **Allocate** resources using the resource handle and “run” or “execute” the workload
5. Once execution is complete, run another workload or simply **deallocation**
RADICAL-Pilot: Further Information

- Repository: https://github.com/radical-cybertools/radical.pilot
Ensemble Toolkit: Further Information

- Documentation: http://radicalensemblemd.readthedocs.org/en/latest/
- Repository: https://github.com/radical-cybertools/radical.ensemblemd