CMSC 714
Lecture 9
Scientific Workflows – Pegasus & Swift/T

Alan Sussman

Notes

- MPI project due today, 6PM
  - Questions on project?
- OpenMP project will be posted on Monday or Tuesday, and due 2 weeks later
- Readings posted for next week
  - Don’t forget to send questions when you are assigned

Scientific Workflows

- Generically, a set of scientific applications that work together to solve a problem
  - A set of computations and their data requirements
- Examples include multi-stage simulation and data analysis pipelines (sometimes with input preprocessing), ensembles of simulations varying input parameters
- Many scientific workflow management systems have been built and used
  - Examples include Pegasus, Taverna, Galaxy, Kepler, Swift, Nimrod, MakeFlow, …
  - For a list of actively-developed open source workflow systems, see https://workflows.community/systems
- Can be run on a wide range of computational and storage resources
  - For computation, sets of local machines, campus clusters, national HPC infrastructure, commercial and academic clouds, …
  - For storage, from local filesystems to shared (parallel) filesystems to cloud provided object storage (e.g., Amazon S3), etc.

Pegasus

Pegasus Overview

- A scientific workflow management system (WMS) developed at USC ISI
  - Goal is to manage a workflow executing on potentially distributed data and compute resources
- Separates the workflow description from the description of the execution environment
  - workflows are based on a Directed Acyclic Graph (DAG) representation of a scientific computation, with tasks to be executed represented as nodes, and the data- and control-flow dependencies between them represented as edges
- Basic model
  - the user has access to a machine where the workflow management system resides
  - the input data can be distributed across diverse storage systems connected by wide area or local area networks
  - the workflow computations can also be performed across distributed heterogeneous platforms

Pegasus components

- Mapper
  - Generates an executable workflow based on an abstract workflow provided by the user or workflow composition system
  - Finds the appropriate software, data, and computational resources required for workflow execution
  - Can also restructure the workflow to optimize performance and add transformations for data management and provenance information generation
- Local Execution Engine
  - Submits the jobs defined by the workflow in order of their dependencies
  - Manages the jobs by tracking their state and determining when jobs are ready to run
  - Submits jobs to the local scheduling queue
- Job Scheduler
  - Manages individual jobs and supervises their execution on local and remote resources
- Remote Execution Engine
  - Manages the execution of one or more tasks, possibly structured as a sub-workflow on one or more remote compute nodes
- Monitoring Component – via runtime monitoring daemon
  - Monitors the running workflow, parses the workflow job and tasks logs and populates them into a workflow database.
  - Database stores both performance and provenance information, and sends notifications back to the user notifying them of events such as failures, success, and completion of tasks, jobs, and workflows

DAGs and Mapping

- DAG resource-independent workflow description (DAX) generated through API calls in Python, Java, Perl
  - Also hierarchical description (and execution) for complex DAGs – a node can represent a whole DAG from a lower level
- Mapping is technically the difficult part
  - Map/plan the DAX abstract workflow onto the execution environment
  - Includes computation invocation on the target resources, the necessary data transfers, and data registration
  - To find the necessary information, queries several catalogs:
    - *Replica Catalog* to look up the locations for the logical files referred to in the workflow
    - *Transformation Catalog* to look up where the various user executables are installed or available as stageable binaries
    - *Site Catalog* that describes the candidate computational and storage resources that a user has access to

Workflow Execution

- After mapping the executable workflow is generated in a way that is specific to the target workflow engine – default is *HTCondor DAGMan*
- Then submitted to the engine and its job scheduler on the submit host – default is *HTCondor Schedd*
- Each node in the DAG is a job in the executable workflow and is associated with a job submit file that describes how each job is to be executed
  - Includes executable that needs to be invoked, the arguments with which it has to be launched, the environment that needs to be set before the execution is started, and the mechanism of how the job is to be submitted to local or remote resources for execution
- When jobs are ready to run (their parent jobs have completed) DAGMan releases jobs to a local Condor queue that is managed by the Schedd daemon
  - By default, jobs run in the local HTCondor pool, but can also be submitted with several remote job management protocols, including ssh, SLURM, and other resource managers

More Pegasus Features

- Mapper performs transformations to improve the overall workflow performance, both at “compile” and ”runtime”
  - Reuse data (files) if they already have been computed and are available
  - *Site Selector* maps the jobs in the optimized workflow to the candidate execution sites specified by the user
  - Tasks can be clustered to deal with large numbers of short running tasks, to minimize overhead
  - Mapper accesses input data (files) through Replica Catalog to find and stage data (add nodes in DAG), and adds data stageout nodes to stage intermediate and output datasets to user specified storage locations
- For reliability (dealing with failures) jobs can be retried, file transfers also retried, only parts of the workflow that have not completed are retried if the entire workflow fails, and the retried workflow can be replanned (new mapping)

CyberShake Application

- Large scale earthquake modeling application from Southern California Earthquake Center (SCEC)
- Over 6 studies in 5 years, Pegasus executed
  - hundreds of millions of tasks on 6 different HPC resources using millions of core hours, taking thousands of hours to execute (makespan)
  - producing hundreds of millions of files, totaling tens of TB of data
- For more info on Pegasus, see https://pegasus.isi.edu/

Swift/T

Swift/T Overview

- Dataflow language implementation designed for scalability, based on Swift
- Includes a distributed dataflow engine that balances program evaluation across large numbers of nodes using dataflow-driven task execution and a distributed data store for global data access
- Compiler translates a user script into a scalable, decentralized MPI program through associated libraries
- Extends the Swift dataflow programming model of external executables with file-based data passing to finer-grained applications using in-memory functions and in-memory data
- Targeted at *many-task* applications with:
  - Non-trivial coordination and data dependencies between tasks - not from a static DAG but from dynamic evaluation of programs written in a concurrent language (Swift)
  - Irregular or unpredictable computational structure – load balancing can be difficult
  - Orchestration of large application codes - by composing complex executable programs or library functions using implicitly parallel dataflow

Programming Model

- Hierarchical programming
  - Scripting to coordinate components, with performance-critical code remaining in lower level languages (e.g., C, C++, Fortran, etc.), using threads or MPI for fine-grained parallelism
- Implicit parallelism
  - When control enters a code block, any Swift statement in that block can execute concurrently with other statements
  - Swift is a *functional* language, so no mutable state and uses write-once variables to schedule execution based only on data dependencies
    - Each operation can be realized as an asynchronous task, eligible to be executed anywhere in a distributed-memory system
- Determinism by default
  - Swift has a deterministic sequential interpretation for most language features, with non-determinism only introduced through explicit non-deterministic constructs
    - All core data types in Swift, including arrays, guaranteed to be deterministic and *referentially transparent*: i.e., querying the state of variable **x**, or any copy of **x** with operation **f** always returns the same result, regardless of where **f (x)** is in the program – even in the case of operations that insert data into an array.

Swift/T architecture

- STC compiler
  - Swift front end, a series of optimization passes, and a Turbine code generator
  - Generates the Swift-IC intermediate representation -flatter than Swift code, broken down to individual Turbine operations, to simplify analysis and optimization
- Turbine
  - enables distributed execution of large numbers of user functions and of control logic used to compose them.
  - Turbine programs are MPI programs that use the ADLB (Asynchonous Dynamic Load Balancer) and Turbine libraries
  - Requires the compiler to break user program code into discrete fragments, to enable all work to be load balanced as discrete *tasks* using ADLB - the fragments are either user-defined leaf tasks, such as external compiled procedures or executables, or control fragments for dataflow coordination logic.
    - invocation of a fragment, combined with input and output addresses, is a *task*

Turbine (cont.)

- Turbine
  - Tracks data dependencies between tasks in order to know when each is eligible to run.
    - Turbine provides a globally-addressable *distributed future store* that drives data-dependent execution
    - A task becomes ready once its data dependencies have been satisfied
    - When a task runs, it fetches its input data, executes, then produces output data, notifying the Turbine data dependency engine, which then releases newly-runnable work
  - Distributed future store used to pass data and to track data dependencies between tasks
    - Provides *write-once* variables for primitive types, which are used as *futures* for output of asynchronous tasks
    - Every Turbine data item starts off in an open state, and once the value is final (i.e., a write-once variable has been written, or a container associative array has had all values inserted), it is switched to the closed state
    - Each data item has a unique 64-bit ID, which is hashed to find its location, allowing any node in the cluster to access the data
  - Each execution of a Swift function is realized as the execution of one or more Turbine tasks
    - data dependency tracking used to launch each task at the correct time

Performance

- Tests on IBM BlueGene P/Q systems at Argonne National Laboratory
- Goal was to measure Swift/T's ability to manage a large-scale system and rapidly launch tasks on available processors, and to run a "real" application
- Results from synthetic patterns show that Swift/T can generate large numbers of tasks quickly, and get good load balance and system utilization
- For the SciColSim application, result shows that Swift/T can deliver computing cycles to real application (high utilization) codes as coordinated by complex application scripts
  - With one optimization (task priorities) to deal with long-tail effects (some long-running tasks complete late)

- For more info on Swift/T, see http://swift-lang.org/Swift-T/