ATLAS Experience with HEP Software at the Argonne Leadership Computing Facility

Thomas D. Uram and Thomas J. LeCompte
Argonne National Laboratory, Argonne IL
E-mail: `turam@anl.gov, lecompte@anl.gov`

D. Benjamin
Duke University, Durham NC
E-mail: `benjamin@phy.duke.edu`

Abstract. A number of HEP software packages used by the ATLAS experiment, including GEANT4, ROOT and ALPGEN, have been adapted to run on the IBM Blue Gene supercomputers at the Argonne Leadership Computing Facility. These computers use a non-x86 architecture and have a considerably less rich operating environment than in common use in HEP, but also represent a computing capacity an order of magnitude beyond what ATLAS is presently using via the LCG. The status and potential for making use of leadership-class computing, including the status of integration with the ATLAS production system, is discussed.

1. Introduction

The computational needs of modern HEP experiments like ATLAS continue to grow every year; ATLAS' grid use is over a billion CPU-hours per year. At the same time, it is proving increasingly difficult to grow the grid: Moore's Law is stalling out, and with the global financial crisis, the competition for government support has grown more keen. This has placed ATLAS in the position of being limited not by the number of events that can be recorded, but by the number that can be simulated. Bringing new resources and new classes of resources can potentially mitigate this.

Modern High Performance Computing involves a very large number (in some cases hundreds of thousands) of processor cores, connected through a high speed network. Typical jobs are highly parallel, with each core calculating a small part of the problem and coordinating their activities via the processor interconnections. This doesn't exactly match the HEP paradigm, but nevertheless there are common features, such as the use of parallelization. If such systems could be adapted to HEP use, the corresponding cycles could be offloaded from the grid and the recovered cycles used in any way that the experiment chooses. For these supercomputers to be useful, it is not necessary that they be able to run any possible task, or even the majority of tasks; what matters is not how many **kinds** of jobs that can be run but the total number of cycles that can be offloaded from the grid.

Two kinds of tasks have been identified as relatively suitable for running on supercomputers: event generation and simulation. The most suitable are the event generators, which tend to be stand-alone, mostly computational, and require little in the way of external database access. These correspond to perhaps 10 or 15% of ATLAS' grid usage. Simulation is somewhat more difficult, as these tasks tend to be tied to a particular simulation's framework, and often require database access for geometry and run conditions. However, these tasks take up approximately two-thirds of the grid capacity, so there is a lot to be gained by moving even a fraction of them to other resources.

We have been exploring the use of the supercomputers Intrepid and Mira, operated by the Argonne Leadership Computing Facility. Intrepid is an IBM BlueGene/P with 163,840 PowerPC 450 cores, 512 MB of memory per core (480 MB usable), and a peak performance of 557 teraflops. Mira is an IBM BlueGene/Q with 768,432 PowerPC A2 cores, 1 GB of memory per core, and a peak performance of 10 petaflops. Both systems have smaller development systems that we used extensively.

2. Experience with HEP Software

Alpgen[1] is the generator we have used most extensively. It's written in a reasonably portable dialect of FORTRAN and compiles using either the Gnu or IBM compiler. It runs on both the BG/P and BG/Q systems with no changes to either the code itself or the build environment. We normally run it with a slight modification so that the random number seed is dependent on the MPI rank; otherwise each core would generate identical events. This is the only use of MPI in Alpgen.

MCFM[2] runs on the BG/Q systems; on the BG/P the generated code is too large for 24-bit relative jumps to reach from some branch sites. Sherpa[3] runs on the BG/Q systems as well; the memory footprint is too large to run efficiently on the BG/P. Sherpa has an interesting feature that its integration stage as well as its event generation phase can be parallelized via MPI.

Geant4[5] runs on the BG/P systems. While Geant itself runs without modifications, we discovered that the build environment needed to be substantially modified. Like all code that will be run on this class of hardware, it needs to be cross-compiled on the front end nodes to run on the worker nodes. However in this case it required a large number of by-hand adjustments to the build scripts to make a working version. These modifications are completed for the BG/P but not for the BG/Q. Unlike event generators, which are useful in standalone mode, Geant is a toolkit called by applications. These applications may require additional services which complicates their implementation on these systems.

Root[4] runs on the BG/P. In this case, the difficulty was in building a static executable. In particular, this requires that static versions of libraries be installed, which is not always the case, and is why there is not yet a BG/Q version. While CINT works, we have found the most effective way to run Root is via a C++ program that calls Root classes.

3. Scheduling Jobs on Argonne Leadership Class systems

Argonne's leadership class systems include Mira and Intrepid. These systems are designed for high performance scientific computing, with facilities to support the large scale communication that occurs in the tightly-coupled applications that typically run on them. HEP event generation jobs consist, instead, of large ensemble runs: many processes that can run independently, with no intermediate communication required. Due to the scale and variety of jobs run on these systems, vacancies occur in the scheduler, creating opportunities for jobs that can run in smaller blocks of compute nodes, with less demand for communication, and for shorter runtimes. We seek to exploit this availability for HEP, by decomposing jobs in scale and time to utilize these cycles. The ALCF systems, including Mira and Intrepid, and their development counterparts Vesta and Surveyor, use the open source Cobalt [6] scheduler for job submission and execution, allowing us to analyze and leverage the internal scheduling state and algorithms for opportunistic sizing and placement of jobs.

In order to execute ATLAS jobs on the ALCF systems, a translation layer is required for retrieving jobs from the Open Science Grid (OSG), managing their execution, and returning the results to OSG. We have constructed a modular system that interacts with ALCF and OSG to conduct the necessary operations; the system consists of the following components:

(i) Data Stager: a data staging and message queuing system that acts as intermediary between OSG and ALCF, managing state transitions during execution and data transfers.

(ii) Balsam Service: a resource-side service that periodically queries the Data Stager for available jobs. The service resides on a login node to enable access to the scheduler, so that it can monitor job execution.

(iii) Balsam Daemon: a resource-side daemon that retrieves input data from the staging system, manages job execution, and stages output data to the staging system. The daemon resides on a login node so that it can submit jobs to the scheduler and stage data into and out of the resource environment from the Data Stager.

These components are described in more detail below.

3.1. Data Stager

The data stager processes OSG job descriptions, manages data transfers to and from OSG, and controls the distribution and monitoring of jobs to its target resources. The data stager's message handling employs the AMQP [8] standard to implement queues that describe the state of jobs in the system. When a job is retrieved from OSG, an *estimate* message is created for each of the target resources. Each resource's service responds to the estimate message with a prediction of when the job might run. The Data Stager scores the resources according to these predictions and other factors, finally determining where the job should be scheduled. At this point, a *job* message is queued for the target resource; the corresponding resource will execute the job based on this message.

Listing 1: A sample XML job description

```
<?xml version="1.0" encoding="UTF-8" ?>
<HPCjob>
  <program>Alpgen
    <process>zjet</process>
  </program>
  <identity>
    lecompte
  </identity>
  <computer>
    <name>vesta</name>
    <cpuhours>3090</cpuhours>
    <shape>
      <nodes>32</nodes>
      <cores_per_node>16</cores_per_node>
      <flag>—mode c16</flag>
      <presubmit_script>presubmit.sh
        <parameters>512 input.1.2</parameters>
      </presubmit_script>
    </shape>DataS
  </computer>
  <transfer>
    <inputfile>
      gsiftp://path/to/inputs
    </inputfile>
    <outputdir>
      gsiftp://path/to/outputs
    </outputdir>
  </transfer>
</HPCjob>
```

3.2. Balsam Service
The Balsam service is the gateway through which jobs enter the execution environment. The service polls the Data Stager for two types of message: *runtime estimates* and *jobs*. For runtime estimates, the service examines the jobs in the Cobalt scheduler to determine an estimate of when the job will run. Each resource returns its estimate to the Data Stager, so that it can select the resource to which the job should be assigned.

The service also polls the Data Stager for jobs that are ready to run. By this approach, we are able to run services on multiple compute resources, enabling metascheduling of jobs across them. When the service encounters a job that is ready to run, it retrieves the job description from the URL specified in the message; the job description (Listing 1) identifies the process to be run, the machine on which the job should be run, the parameters to use when submitting the job, and the input and output URLs. The service validates the job description, including ensuring that the requested executable can be found among the executables allowed to run on the resource. If the job description validates successfully, the service stores the job description and related metadata in its database. The service interacts with the database through the Django [7] Object Relational Mapper (ORM), which provides a Python class proxy interface to the underlying database tables. If the job description is invalid, the DataStager is notified immediately via an *error* message.

3.3. Balsam Daemon
The Balsam Daemon is responsible for exchanging inputs and outputs with the Message Queuing System, and for managing the execution of jobs on the compute resources. The daemon periodically queries the database for jobs that are ready to run. The job description can include a presubmission script that is to be executed to prepare the necessary inputs before job submission. In the case of Alpgen, the presubmit script creates work directories into which each of the processes will write their outputs.

Each job is processed in its own thread within the daemon, to eliminate the potential for one job to affect others. The job thread manages the transfer of input files using the transfer module; for ALCF systems, the transfer module uses GridFTP to transfer files from the input location specified in the job description. When the input files have been successfully transferred, the job is submitted via the scheduler module which, for ALCF systems, interacts with the Cobalt scheduler. The daemon monitors the job periodically by polling the queue. When the job has completed, the daemon begin the transfer of outputs to the output directory specified in the job description. When this is complete, the daemon marks the job as having completed its execution. The Balsam service, upon discovering jobs in this state, notifies the DataStager that the job is complete. If an error occurs at any point in the execution of the job, the job is marked as having failed, and the service sends an *error* message to the DataStager, including a message that describes the failure.

In its current deployment on ALCF systems and processing HEP jobs, Balsam fetches jobs from the DataStager using AMQP-formatted messages, and carries out their execution on the ALCF machines using GridFTP for transfers and Cobalt for scheduling jobs. Balsam has, however, been designed in a modular fashion to accommodate a pluggable architecture in which alternative job sources, transfer protocols, and job schedulers can be integrated. These abstractions have been exploited in trial environments that utilized different components, such as a Galaxy job source, scp for transfers, and ssh for running (submitting) jobs.

4. Results

At ALCF we have generated over 81 million Alpgen events on Intrepid and delivered them to ATLAS. These were a mix of Balsam and non-Balsam jobs. These events are presently undergoing validation, however we expect no problems as samples of order 100,000 events are bit-for-bit identical to ones produced on an x86.

Additionally, we have run one million CPU-hours of Root on Intrepid, primarily limit setting with the Bayesian Analysis Toolkit [9]. As is typical of HEP computing, this job can be divided into subtasks that naturally fit within the 512 node and 6 hour limits of a short job on Intrepid, which in turn allows Cobalt to schedule these jobs opportunistically: at times we were running more jobs than the policy strictly allowed, so that the scheduler could keep the machine busy. The key to doing this efficiently is to be able to "reshape" a job: adjust the duration and the number of nodes to be able to fit into an opportune opening.

5. Future Work

Due to the ensemble nature of event generation, a single event generation job can be decomposed into a collection of smaller jobs, which can be executed independently and aggregated into a complete result set when all jobs have concluded. We intend to design a solution for parceling the execution of jobs in this manner, monitoring the execution of the individual jobs to guarantee the successful completion of the whole, which will enable us to run smaller jobs and adapt to smaller availability counts as needed.

We plan to integrate more closely with Cobalt and other schedulers, which already examine queues to prioritize job execution and, therefore, are best suited to provide estimates of queue wait time for jobs of particular sizes and durations. This will allow us to provide better estimates to the Message Queueing System and potentially improve multi-resource job packing. This integration will also enable us to run jobs opportunistically, decomposing our jobs into slots that become available while the machine is draining queues to make room for upcoming large jobs.

References

[1] M. L. Mangano, M. Moretti, F. Piccinini, R. Pittau, and A. D. Polosa, "*ALPGEN, a generator for hard multiparton processes in hadronic collisions*," *JHEP* **07** (2003) 001
[2] J. Campbell, R.K. Ellis, `http : //mcfm.fnal.gov/`; J. Campbell, R.K. Ellis, Phys. Rev. **D65** 113007 (2002).
[3] T. Gleisberg and *et al.*, "*Event generation with SHERPA 1.1*," JHEP **02** (2009) 007.
[4] R. Brun and F. Rademakers. http://root.cern.ch/root/doc/RootDoc.html.
[5] S. Agostinelli *et al.* [GEANT4 Collaboration], "*Geant4 - a simulation toolkit*", Nucl. Instrum. Meth. A **506** (2003) 250.
[6] W. Tang, et al. "*Fault-aware, utility-based job scheduling on Blue Gene/P systems.*" Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on. IEEE, 2009.
[7] Django Web Framework, https://www.djangoproject.com
[8] Advanced Message Queuing Protocol, http://www.amqp.org
[9] A. Caldwell, D. Kollar, K. Krninger, "*BAT - The Bayesian Analysis Toolkit*", Computer Physics Communications 180 (2009) 2197-2209