Jobmon: So you can go home on the weekend

Geoffrey Phipps, Tom Fleming, Megan Moore, Disha Patel, Christine Pinho, Logan Sandar, Naris Silpakit, and Liming Xu
Institute for Health Metrics and Evaluation, University of Washington, Seattle, WA, USA

Abstract Jobmon is Python-based general-purpose job description and control system developed at the Institute for Health Metrics and Evaluation (IHME) for their Univa Grid Engine (UGE) cluster. Jobmon supports rich and fine-grained computational graphs, has automatic job retries for node failures, and “whole of application” resumption to recover from large scale outages. Usability by incidental programmers is a key design goal. Jobmon has reduced elapsed run times and eliminated manual monitoring at night outside of office hours. It is a practical solution for resource-challenged organizations. We describe the design goals of Jobmon, its implementation, and provide metrics on its performance at IHME.

Keywords: High Performance Computing, Distributed Computing

1 Introduction

IHME is an independent global health organization at the University of Washington. IHME conducts research into public health statistics, principally via the Global Burden of Disease (GBD) study. The first GBD was published in 1993, and it is now published annually [1]. Each GBD publication cycle is called a round, and is suffixed by the final year of results data.

1.1 Computing facilities

IHME owns and operates its own computing cluster, hosted within the University of Washington. The cluster consists of 450 computing nodes of 21,000 cores in total, all connected to a shared 6 Pebibyte Qumulo file system. The cluster nodes run Linux, the whole assemblage is managed by UGE (also known as the Sun Grid Engine or SGE) [2]. The other computing facilities play no significant role in scientific computation and will not be discussed in this paper.

1.2 Computing the GBD study

Simplifying for the purposes of this paper, the GBD study models global health as an eight-dimensional cube over location, year, age-group, sex, cause/sequelae, risk-factor, measure, and metric. This is referred to as the demographic cube. For GBD 2019 the dimensionality of the cube is approximately (1,000 x 30 x 30 x 3 x 1,000 x 1,000 x 4 x 3) or i.e. 32.4 trillion cells. The probability density function of the health outcome in each cell is recorded by 1,000 “draws,” each of which is a 64-bit floating point number. A naive storage implementation would occupy at least 236 TB, although in practice sparse-matrix and logical health rules reduce the size of the final outputs to 156 TB.

The draws are stored as HDF files using directory-based indexing. For the purpose of job control they could equally be stored in an auto-sharding file system such as Hadoop or Zarr. A typical application partitions the computation along demographic axes, for example the first phase of the Burdenator (which calculates the attributable burden of disease) application runs one job per (location, year) pair, i.e. 1000 x 30 = 30,000 jobs.

1.3 Solving problems here and now

The GBD must be produced every twelve months, with the staff and computing technologies already at hand. Although IHME is grateful for the support of its funders, IHME does not have the funds to build and maintain computing infrastructure with “five nines” of reliability, nor could it move to a public cloud within a 12-month cycle. In fact, no actual computing system can ever be 100% reliable. The result is that the software engineers are driven to solve the usability and reliability problems with the tools that they have at hand or with open-source projects.

IHME increases the size of the demographic cube every GBD round, by increasing the number of locations, age-groups, and risk factors. In practice the limiting factor to growth has been the ability to operate larger and larger applications that consist of thousands of separate jobs run by hundreds of people. In the early days of GBD the applications were operated manually – a data scientist would sit at the terminal and monitor it by hand, stop it when something failed, identify the root cause, determine which individual jobs needed to be re-run, patch the control code, and repeat. Dedicated data scientists and researchers in a team of 35 were able to operate applications that took up to five days with tens of thousands of jobs, but the limits of human scaling have been reached.

IHME staff are predominantly public health researchers and data scientists, with only a small number of software engineers (meaning those with formal computer science
training, or for whom it is their primary skill and interest). For the first two groups coding is a necessary evil that stands between them and their research. Technologies must therefore be quick to learn or they will simply not be used.

The challenge is to find appropriate solutions to the human factor problems of an organization with limited physical, time, and staff resources.

The problem is not cluster efficiency, although Jobmon also improved that metric.

1.4 Definitions

Job – A UGE job (i.e. a Linux process controlled by the UGE distributed scheduler) within an application.

Application – A scientific computation that is managed and operated as one entity. IHME applications have between hundreds and half-a-million separate jobs.

Directed Acyclic Graph (DAG) – This has the usual meaning of a graph with directed edges but no cycles. For this paper, DAG always refers to a job dependency graph – a job cannot begin until all its upstream (antecedent) jobs have completed successfully. A DAG can have many start nodes and many end nodes, but cannot be disconnected.

2 Requirements in Detail

The goal was to allow staff to not work on the weekend.

There was to be no more active monitoring, no more heroic dives with bit tweezers to find the failed jobs in an application of tens of thousands of jobs, and no more slippage of deadlines due to hardware failures.

Usability aimed at IHME staff – The majority of IHME staff are incidental programmers – they write software to get their job done, but it is not their prime interest, nor are they professionally trained. This requirement drove the use of direct coding against a Python API rather than (say) a new Domain Specific language (DSL) or the meta-programming approach of Luigi. Usability is a corollary of the more general The Customer is Always Right, even When They are Wrong theorem. It is the customers choice to adopt your new product (or not). Your own opinion is irrelevant. For a technology to be voluntarily adopted it must immediately and easily solve one of the customer’s most pressing problems.

No Increase in Elapsed Runtime – The elapsed time of application must not increase, and preferably should drop.

Reliability – The system must Just Work, given normal cluster error rates. The service-level agreement allows for one annual outage requiring significant intervention.

Multi-language support – The majority of IHME’s code is written in R, although the largest applications are in Python/Pandas, and there is still significant amounts of Stata. The control language could be in any language, but it had to be able to control jobs written in any language. In practice any language can be wrapped using Bash.

Expressiveness – Jobmon must be able to express all DAGs used at IHME. Care must be taken expressiveness can conflict with usability by encouraging the customer to implement things that they did not want.

2.1.1 Non-goals

The impossible 10% use case – DAGs that were theoretically possible but not used at IHME were ignored.

Immediate cross-platform capability – although it is valuable to experimentally run small DAGs off-line in reality IHME will be using the UGE cluster for several years. Therefore, Jobmon was designed to be portable, rather than already be ported. There is no such thing as portable code, only code that has been ported, and code that has not. Jobmon can currently control UGE jobs and sequential jobs on a single Linux or Unix platform. It has internal interfaces to support porting onto other platforms, for example a container-based or cloud computing environment, but it does not do so directly today.

3 Resulting features

Direct programming in Python – The data scientists at IHME use R, Python, and Stata. Choosing a different language (e.g. Java, Go, Rust) was theoretically attractive but in practice would have led to an unused product due to the cost of adoption.

Job templates – All IHME applications are Single Instruction Multiple Data (SIMD) – they have a small number of sub-programs which are executed in parallel across different subsets of data. For usability it is important that job templates be well supported.

Static DAG of jobs – The compute graph is calculated once, at the beginning of the run. A survey of all large IHME applications revealed that all but two applications could statically compute their job DAG from their input arguments. Of the two applications that needed a dynamic DAG, one application encapsulated the dynamic portion within a larger job that was still amenable to Jobmon control, leaving only one unsupported application.
Centrally stored application state – UGE supports log files and has a traditional Linux command line interface. IHME experience is that a small DAG can be monitored with these tools, but anything beyond a few hundred jobs can only be monitored if the job state (not log events) is queryable in one place with an expressive language. Jobmon uses a central MySQL database for this purpose.

Automatic retries – Jobmon will automatically retry a job up to a configurable number of times (the default is three, set to 11 during difficult times). If the job has a bug, then it will always fail and hence there must be a maximum limit. Checking for job termination is more complex than simply checking the return code using the UGE `qacct` command. Jobs sometimes simply disappear without trace, or hang forever. Hence there is a thread that reconciles Jobmon’s run-list against UGE’s run-list, and a second thread to time-out and kill hung jobs. UGE does have a rerun feature but it does not have a maximum number of retries. Therefore, a job with a bug will be continually relaunched, blocking the entire computation.

Resuming an entire application – Jobmon must automatically checkpoint the DAG. If an application is stopped for whatever reason (change in priority, complete loss of the cluster, bad input data, etc) then the application can resume at the same point. Partially completed jobs are restarted, existing jobs are detected and killed. Jobs must be coded to be re-entrant, i.e. delete any pre-existing outputs as they start. Resumability must not require extra programming by the data scientist, or the feature will not be used.

Bad node detection – Jobmon detects and reports nodes with abnormally high failure rates. Jobmon could automatically exclude such nodes from the UGE scheduler, but this has not been necessary in practice. If the number of bad nodes is low then re-launching a job will rarely land it on a bad node a second time, so most jobs succeed on the second try.

Job statistics and categorization – Jobmon automatically collects resource usage statistics. It also allows the application to annotate jobs and workflows with arbitrary information, enabling statistical analysis of performance.

DAG visualization – Jobmon can automatically create images of the job DAG. Figure 1 shows a toy example of CoDCorrect over a very limited data set. Although the visualization feature is popular the current implementation has poor automatic layout and does not scale to large DAGs.

4 Why not use technology X?

The Jobmon team at IHME did not set out to write a job control system. It was our original intent to find an acceptable open-source solution.

To be clear, IHME continually attempts to improve the reliability of its computing infrastructure, but no physical system will ever be 100% reliable. Large-scale distributed systems must be designed to expect failure, all that differs is the non-zero rate of underlying errors.

A number of existing job control systems were tried or examined before we decided to invest in Jobmon. Although many technologies appear to be suitable in theory, in practice they were not acceptable.

“Just use Technology X” is often a Turing Tarpit argument. Two technologies might be equivalent in theory, but experimental evidence is that they are not equivalent when used by real humans in a large organization. For example, both and C and Python are Turing complete, but a group of non-professional programmers operating under time constraints demonstrably produce more features using Python than they do with C.

4.1 Why not make better use of UGE?

IHME has operated a UGE cluster since 2010. UGE possesses many of the required features, but it proved to be unsuitable due to ease of use and reliability issues.

UGE implements job dependencies using holds – a downstream job will be held (i.e. not start) until the upstream job completes. For individual jobs this is identical
to Jobmon’s upstream task dependencies. The UGE scheduler stores the entire DAG for the entire cluster in memory, rather than Jobmon’s approach of only having one DAG per controlling process. In practice, large DAGs in IHME’s installation of UGE have caused the scheduler to crash or slow to a crawl due to the large memory required. Holds using regular expressions on job names were particularly dangerous. Therefore data scientists would either build monolithic jobs (to reduce the job count but cause job-fit problems), or use the UGE’s array job feature.

An array job is a group of identical SIMD jobs that vary only by an index parameter, they fulfill the templated job requirement. The actual parameters for each individual job are obtained by a lookup into an indexed parameter table. The problem is that the i\textsuperscript{th} job in downstream array can only depend on the i\textsuperscript{th} job of its upstream array. IHME DAGs frequently need to process tree structures, whereby a node interior to the tree must wait for many child nodes with arbitrary ids, so the i-depends-on-i rule was too limiting. In practise, users would manually implement holds by running jobs in waves, increasing the application’s elapsed time. UGE’s holds and array jobs proved to be only useful for smaller and simpler applications.

UGE does have a retry feature: the “-r” flag (meaning rerunnable). An upper limit on the number of retries can be set globally by the cluster administrators. In Jobmon each type of job often has a different number of retries. The standard is 3, but we never retry database uploads. A global limit is operational friction.

A more subtle problem is the usability of UGE. The commands (e.g., qsub, qstat, qacct) are old-school shell commands with multiple case-sensitive switches. Many of the incoming data scientists have neither Linux experience nor extensive programming experience and therefore only ever use the most basic commands. On the other hand they all know Python so there is less training needed for – and resistance to – a Python-based system.

It is possible that a perfectly configured UGE would solve some of these problems, but would achieve neither the desired usability nor possess the required expressiveness of the DAG. The theory that “UGE is a scheduler suitable for IHME” was experimentally tested for 9 years and disproven.

4.2 Other DAG-based systems

This section describes other systems as they were in late 2016, and it is probable they may have subsequently solved all these shortcomings.

Two separate IHME projects tried Luigi [4]. The projects were not as usable as we needed, for the following reasons:

1. Luigi uses backward-chaining from a goal artifact (typically a file). Several customers commented that they preferred to think forwards – first do this, then this. Thinking backwards was a barrier to adoption.
2. Luigi uses meta-programming for its job templates. Each task is a class, so a template for a set of tasks is described using meta-programming. That was simply too hard for time-pressed researchers.
3. Luigi describes the DAG in terms of both jobs and output artifacts, whereas Jobmon only has jobs. We considered using both jobs and artifacts in Jobmon’s domain model, but adding artifacts provided no benefit and increased the complexity.

One IHME project used Airflow [4], but its design center was for slowly-changing DAGs with relatively few large processes. The number of messages increased rapidly with the number of jobs – DAGs with more than 1,000 nodes had unacceptably slow performance. Airflow is designed to have fine control over the actual jobs, meaning that to operate as it was designed it needs to replace UGE, which was not organizationally possible. However, Airflow concepts and terminology did heavily influence Jobmon’s design.

Some IHME projects used Hadoop, Spark, Dask, and other map-reduce technologies. The map-reduce paradigm is not sufficiently expressive for all of IHME’s needs.

4.3 Why not centralized log management?

The earliest version of Jobmon only implemented centralized job state, allowing an operator to use simple SQL commands to discern the state of the entire application. This was considerably easier and more reliable than tailing hundreds of log files. However, Centralized Log Management certainly improves alarming and trouble-shooting.

5 Product design

5.1 Motivating example

The following pseudo-code fragment is taken from IHME’s Burdenator (so-called because it calculates the burden of disease). The Burdenator is a five-phase application that would be difficult to model with UGE’s array-job hold structure. The primary partitioning is by location hierarchy, jobs in the second (interior node of the location hierarchy) phase depend on many jobs in the first phase. The jobs are typically partitioned by location, year, and sex:

```python
for y in years:
    for loc in leaf-locations:
```
for sex in sexes:
    qsub burdenate-leaf-loc args
    // wait by polling log files or UGE

New code with Jobmon:

```python
Workflow w = Workflow(args elided)
for y in years:
    for loc in aggregate-locations:
        for sex in sexes:
            w.addTask(burdenate-leaf-loc args)
w.run()
```

The goal was to replace only the innermost line of code within the existing for-loops. The change in coding was so small that no team rejected it as being too hard to convert.

The usual pattern is to create a function that creates tasks of a specific type (meeting the job template requirement). The initial version of Jobmon used subclassing of a base job class for this purpose, but the Python users were more comfortable with simple functions than subclassing.

Dependencies are added within the job template function by finding the upstream jobs by their input arguments. For example, if a phase-2 job is partitioned by location and year, then it will depend on two phase-1 jobs which are partitioned by location year and sex.

```python
def create_phase_2_task(loc, year, job_dict, args):
    job = location_aggregation(args)
    job.add_upstream(
        job_dict.get("phase1", loc, year, "male"))
    job.add_upstream(
        job_dict.get("phase1", loc, year, "female"))
```

5.2 Classes and objects

Jobmon is implemented in 11,000 lines of Python 3. The object-oriented paradigm is suited to Jobmon because there are obvious objects with distinct states and behavior.

**Job** – This is the intention to run a job, the actual execution of a Job is a JobInstance (defined below). To use a programming analogy, a Job is a *function definition*, whereas a JobInstance is an *actual call* of that function. Job is subclassed in two different axes. The first approach tried was subclassing by the health function of the job (e.g. burdenation of leaf locations in a location hierarchy). Users however were more comfortable with subclassing by the technology of the actual job – Bash, Python, R or Stata.

**JobInstance** – A JobInstance is created when a Job is launched as an operating system process. The JobInstance tracks the execution state and records all resource usage (e.g. CPU usage). If a Job is retried then a new JobInstance is created for every attempt to run that Job.

**Workflow (and DAG)** – A DAG is a computational graph of Jobs. A Workflow is an intention to run an application with a specific set of arguments, e.g. specific years, sexes, and diseases. The arguments to the workflow dictate the jobs that must be run, and therefore the DAG. The DAG is statically determined by the Workflow arguments, it cannot change.

**WorkflowRun** – Similar to JobInstance and Job, a WorkflowRun is an actual execution of a Workflow. If the Workflow is stopped for any reason, then it can be manually resumed which creates a new WorkflowRun.

5.3 Modeling of artifacts

Many build (and dataflow and workflow) systems model the outputs of tasks. Downstream tasks are triggered by the creation of these artifacts, rather than by the completion of the task that creates them. The advantages of modeling outputs are that 1) downstream jobs can be started earlier if the necessary upstream outputs are created significantly before the end of the creating job, and 2) it also creates a complete registry of all data (i.e., provenance). However the first situation does not occur at IHME, and the second was deliberately excluded because projects with multiple goals are more likely to fail than projects with a single goal.

5.4 Deployment architecture

Jobmon is deployed in three parts:

**Microservice** – The state of all jobs, workflows etc is stored in a central instance of MySQL, fronted by microservices serving JSON over https, implemented using Python, UWSGI, Flask and NGINX. The microservice is deployed as Docker container on a VM which is not part of the UGE cluster (i.e. not a “submit host”).

**Master client library** – The users’ Python application imports a Python package, which creates and runs the workflow. The client package communicates with the microservice to record all state. The client package is deployed on the cluster so therefore it can issue UGE commands to detect lost jobs, kill errant jobs, etc. IHME’s infrastructure separates long-running microservices from the cluster, so the master must (currently) be on the cluster.

**Worker nodes** – Jobs that are launched by Jobmon execute within a small Python wrapper. The wrapper sends state updates to the microservice, including all exceptions. Python jobs typically use Conda environments, whereas R jobs are launched in Singularity containers.

6 Results in practice

Unfortunately we do not have precise metrics for some of the effects of Jobmon. The project goal was to improve IHME operations, not to measure the Jobmon itself.
6.1 Adoption

The use of Jobmon is not mandated at IHME; application owners only adopt it if they see a benefit in doing so. Jobmon is used by 10 of the 12 major GBD applications (see Table 1). Interviews with the owners of non-Jobmon applications revealed that the final two will be converted this round, time permitting.

Table 1 – Jobmon adoption by major GBD applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>Max. DAG size in jobs</th>
<th>Max elapsed runtime</th>
<th>Uses Jobmon?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burdenator</td>
<td>350,000</td>
<td>30 days</td>
<td>Yes</td>
</tr>
<tr>
<td>CoDCorrect</td>
<td>9,992</td>
<td>2.5 days</td>
<td>Yes</td>
</tr>
<tr>
<td>CODEm</td>
<td>800</td>
<td>15 days</td>
<td>Yes</td>
</tr>
<tr>
<td>COMO</td>
<td>194,522</td>
<td>8 days</td>
<td>Yes</td>
</tr>
<tr>
<td>Dalynator</td>
<td>139,422</td>
<td>25 hours</td>
<td>Yes</td>
</tr>
<tr>
<td>Dismod-MR</td>
<td>815</td>
<td>7 days</td>
<td>Yes</td>
</tr>
<tr>
<td>EPIC</td>
<td>15,000</td>
<td>1.5 days</td>
<td>Yes</td>
</tr>
<tr>
<td>HALE</td>
<td>82,569</td>
<td>15 hours</td>
<td>Yes</td>
</tr>
<tr>
<td>PAF Calculator</td>
<td>135,000</td>
<td>7 days</td>
<td>No</td>
</tr>
<tr>
<td>PAF Compiler</td>
<td>823</td>
<td>6 hours</td>
<td>No</td>
</tr>
<tr>
<td>SEV Calculator</td>
<td>56,862</td>
<td>1 day</td>
<td>Yes</td>
</tr>
<tr>
<td>ST-GPR</td>
<td>10,000</td>
<td>&lt; 1 day</td>
<td>Yes</td>
</tr>
<tr>
<td>Custom</td>
<td>450,000</td>
<td>Abandoned</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Jobmon was an important enabler of IHME’s first set of annual results. Prior GBD rounds only produced estimates for every fifth year (1990, 1995 and so on). The annual run now produces them for every year, requiring five times as much computation. The final run of the Burdenator therefore increased from six to thirty days. This is the longest application ever run for the GBD study and yet it had no stoppages due to cluster issues. Earlier attempts at producing annual results had failed.

6.2 Metrics

We do not have some of the earlier databases, but Table 2 provides a nearly-complete count from May 2018 to March 18th, 2019.

Table 2 – General population metrics

<table>
<thead>
<tr>
<th>Measure</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of workflows</td>
<td>4,242</td>
</tr>
<tr>
<td>Total number of jobs</td>
<td>5,860,486</td>
</tr>
<tr>
<td>Largest workflow (in jobs)</td>
<td>386,853</td>
</tr>
<tr>
<td>Median workflow size (in jobs)</td>
<td>122</td>
</tr>
</tbody>
</table>

6.3 Effect on Performance

In practice Jobmon sped up large applications by packing jobs more tightly in time. Pre-Jobmon applications would launch each job type in a phase, and then wait for each entire phase to complete before moving to the next phase. All application owners reported material speed improvements.

For example, CoDCorrect has seven phases. If all jobs in one phase can be run in parallel on the cluster then the elapsed time for a phase would be the slowest run time of any of the jobs, as shown in Figure 2. For the final annual run of the Burdenator in GBD 2017 the ratio between the slowest and fastest job in the first phase was 125/40 (due to hardware differences), from 26,626 UGE jobs. However, in practice there is insufficient cluster space for all jobs in one phase to be running simultaneously, so the speed-up of 3.125 is only achieved for the last “wave” of jobs launched in a phase. The elapsed time for the Burdenator dropped by about 30%.

6.4 Reliability

6.4.1 Job retries

Standard practise at IHME before Jobmon was that an application with job failures would either to stop at the end of the phase; or run to the end of the phase, and then automatically rerun those jobs that failed, repeating until all jobs were complete. The latter method is similar to Jobmon, but is less effective at job packing because subsequent phases must wait while a small number of jobs re-run even if there is cluster space available for more jobs. In addition the retry logic was specific to each application. The job failure rate by month is shown in Figure 3.

For a manually monitored application the exact set of jobs to rerun must be identified by hand, and the code temporarily altered to restart at that point. Jobmon controlled 5.6 million cluster jobs between May 2018 and March 2019. 1.23% of these jobs were automatically restarted by Jobmon. Without Jobmon every one of these restarts would have been investigated and restarted manually. Assuming (optimistically) one hour of investigation and relaunch time per failure, then the time
saved is 68,628 hours, which is 43 full-time employees over that same time period. This calculation assumes that solving one failure does not solve any other problem (typically false, although the dependencies must be analyzed). However, it is indicative of the improvement in efficiency (not to mention morale). During GBD 2016 an entire seven-day run of the Burdenator was abandoned because calculating the job graph by hand was too hard to perform following a cluster outage.

If an application implements its own relaunch failed jobs at the end of each phase logic, then Table 3 shows the number of extra cycles needed per phase, assuming the cluster job failure rate during crunch time (2.25%). These numbers were calculated by simulating phased relaunching until the number of job failures fell below one. For reference the first phase of the Burdenator had 26,627 jobs in the final annual run of GBD 2017.

The effect on staff morale was not measured directly but is reflected in the voluntary adoption rate of Jobmon (see Table 1). With manual monitoring the Burdenator team would have been on-call for 30 days, probably with 10 to 15 manual restarts. As it happened the team was able to spend the time productively on other tasks.

Table 3 – Number of correction cycles per phase

<table>
<thead>
<tr>
<th>DAG Size</th>
<th>Retry cycles</th>
<th>Extra time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>10,000</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1,000</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

6.4.2 Resuming a stopped workflow

Every application except one uses the resume feature. The only exception was Dismod, which will soon be extended to make use of the feature. 13% of workflows were resumed during the peak period of GBD 2017. Customers are extremely grateful for the resume feature because it often saves a week of repeated work during peak season.

7 Future work

Jobmon will be open-sourced by June 2019.

Jobmon will move from an onsite Docker stack to Microsoft’s Azure Kubernetes Service so as to increase performance, improve reliability, and enable hot-updates.

The database is a single-source of failure. Eventually Jobmon will move to a replicated persistent store.

8 Conclusions

If you have a similar organization with similar organizational challenges then consider using Jobmon. No large cluster can be 100% reliable, and a system like Jobmon will reduce wasted time and lost morale. Jobmon is general purpose, not limited to public health research.

No single idea within Jobmon is novel. What is novel is the application of these ideas to large-scale computing for a large customer base of incidental programmers. Jobmon has succeeded through careful attention to the needs of the customers, specifically usability, reliability, and performance. It has enabled IHME to produce results that were heretofore organizationally impossible.

9 Acknowledgements

We would like to extend our gratitude to the Bill & Melinda Gates Foundation for generously funding IHME and for its support of this research. We also thank the Jobmon user community and IHME’s infrastructure team for their help and patience with the inevitable bugs.

10 References


