SANDIA REPORT

SAND2011-7463
Unlimited Release
Printed October, 2011

SNL Software Manual for the ACS Data Analytics Project

Jon Stearley, David Robinson, Russell Hooper, Michael Stickland,
WIlliam McLendon, Aaron Williams, Arun Rodrigues

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.
Approved for public release; further dissemination unlimited.

Mailing Address:
Sandia National Laboratories
PO Box 5800
Albuquerque, New Mexico 87185

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2011-7463
Unlimited Release
Printed October, 2011

SNL Software Manual for the ACS Data Analytics Project

Jon Stearley David Robinson Russell Hooper Michael Stickland
William McLendon Aaron Williams Arun Rodrigues

Abstract

In the ACS Data Analytics Project (also known as "YumYum"), a supercomputer is modeled as a graph of components and dependencies, jobs and faults are simulated, and component fault rates are estimated using the graph structure and job pass/fail outcomes. This report documents the successful completion of all SNL deliverables and tasks, describes the software written by SNL for the project, and presents the data it generates. Readers should understand what the software tools are, how they fit together, and how to use them to reproduce the presented data and additional experiments as desired.

The SNL YumYum tools provide the novel simulation and inference capabilities desired by ACS. SNL also developed and implemented a new algorithm, which provides faster estimates, at finer component granularity, on arbitrary directed acyclic graphs.

Page intentionally blank

Contents

Page intentionally blank

1 Overview

1.1 Statement Of Work

This document's structure follows SNL's Statement of Work, which specifies the following three deliverables:

- Failure model and simulator source code along with any support libraries, modules, or documentation.
- Analysis tool source code and any support libraries, modules, or documentation.
- Any data generated for testing and validation.

Sections 2, 3, and 4 are dedicated to these deliverables, respectively. The primary delivery vehicle is an SVN repository, referred to as `yumyum`[1] throughout this report. Table 1 summarizes the successful completion of all project tasks and corresponding tools, and as such provides a worthwhile roadmap of this report. Throughout this report, we use *fault* to refer to a component malfunction which results in a job *failure*.

Table 1: SNL Statement of Work tasks, with completion summaries and section numbers where additional details are provided.

Task	Description	Summary	Section
3.1.1	The contractor shall work together with collaborators from the Advanced Computing System Research Program (ACS), Lawrence Livermore National Laboratory (LLNL), and Los Alamos National Laboratory (LANL) to define data requirements and interfaces for input to the system analytics tool.	Early in the project, the "4-data-model" was mutually decided upon.	2.3
3.1.2	The contractor shall collaborate with LANL in implementing the algorithm provided by the ACS into a scalable and extensible analysis tool that is parallelized for POSIX compliant architectures.	At LANL's request, SNL fully implemented the algorithm using SNL's Trilinos libraries. The parallelized tool is in `yumyum/trunk/frequentist/pfat/`, takes weights as input, and outputs estimated fault rates.	3.4
3.1.3	The contractor shall verify that no GPL source code is used in the implementation.	Verified: BSD-like dependencies are SST and Boost, GNU-Lesser dependencies are Trilinos and ParMETIS.	
3.1.4	The contractor shall assist in validation and testing of the analysis tool using data produced by the data collection tool and the simulation tool.	SNL verified the complete graph/simulator/analysis tool chain (see 3.1.7 below). No data from real systems was analyzed, as all collaborators agreed that graphs describing real systems were beyond the scope of the current funding (analysis requires both the data and the graph).	4
3.1.5	The contractor shall develop and implement a simple system failure model for testing and validation of the analysis tool.	A script was implemented to generate simple graphs of components which define true fault rates, and dependency structure. The script is in `yumyum/trunk/generator/`.	2.1

Continued on next page

[1] The `yumyum` repository is accessible via the following command: `svn co svn+ssh://software.sandia.gov/svn/private/yumyum`

Table 1 – continued from previous page

Task	Description	Summary	Section
3.1.6	The contractor shall implement the failure model into a simulator that provides the required system performance data according to the agreed upon interfaces. In addition, the simulator must produce keys for validation of analysis tool results.	The simulator was implemented using SNL's Structural Simulation Toolkit (SST), and is found in `yumyum/trunk/simulator/`. It takes an arbitrary directed acyclic graph and job list as input, and outputs "4-data-model" records and component fault logs. The latter determine observed fault rates.	2.3
3.1.7	The contractor shall freely provide system performance data gathered with the failure model and simulator as necessary for validation of the analysis tool.	Scripts to reproduce all data and plots are provided in `yumyum/trunk/runs/`. Plots compare true, observed, and estimated fault rates as a function of number of jobs, including confidence intervals (see 3.1.9). Execution times are also reported.	4.1
3.1.8	The contractor shall select, implement, and optimize numerical solvers as necessary in support of the analysis tool.	ACS's MLE algorithm was implemented using Newton's method, customized to robustly handle convergence issues arising from insufficient fault observations (e.g. when simulation time is short compared to fault rates).	3.4
3.1.9	The contractor shall develop algorithms for computing confidence intervals and implement them into the analysis tool and provide capability for graphic output (i.e. plots with error bars).	Confidence intervals for the MLE algorithm are estimated using Fisher information, based on the Jacobian of the likelihood surface minimized by the `pfat` tool. Confidence intervals for the CMLE algorithm are measured directly. See 3.1.7 summary above.	4.1
3.1.10	The contractor shall explore and develop additional analytic capability with the guidance of ACS.	SNL developed and implemented a Bayesian algorithm (CMLE) after approval by ACS, which allows for relaxation of the assumption that components having matching divisors (same "row" in the graph) also have matching fault rates. The tool is in `yumyum/trunk/bayesian/` and outputs fault rate estimates.	3.5
3.1.11	The contractor shall explore and develop and implement an algorithm, based on a directed graph, to generate weight parameters required by the analysis tool.	Tools were developed which output weights, given "4-data-model" records and an arbitrary directed acyclic graph as inputs. Two complete implementations are provided: Python scripts in `yumyum/trunk/walker/`, and a C++ program in `yumyum/trunk/yummy/`.	3.1
3.1.12	The contractor shall participate in project meetings as required, either directly or remotely by teleconferencing or other acceptable means.	SNL attended all teleconferences and traveled to the customer site in MD to present to the research and production groups (1/26/2011 and 4/20/2011 respectively).	

1.2 Tool Chain

The tools developed in this project fit together to form a simulation and analysis chain, which is depicted in Figure 1. A short example of the toolchain use is given in this subsection. In the `yumyum/trunk/runs/` directory, a single `make` command generates a graph, runs the simulation, records the observed failure rates,

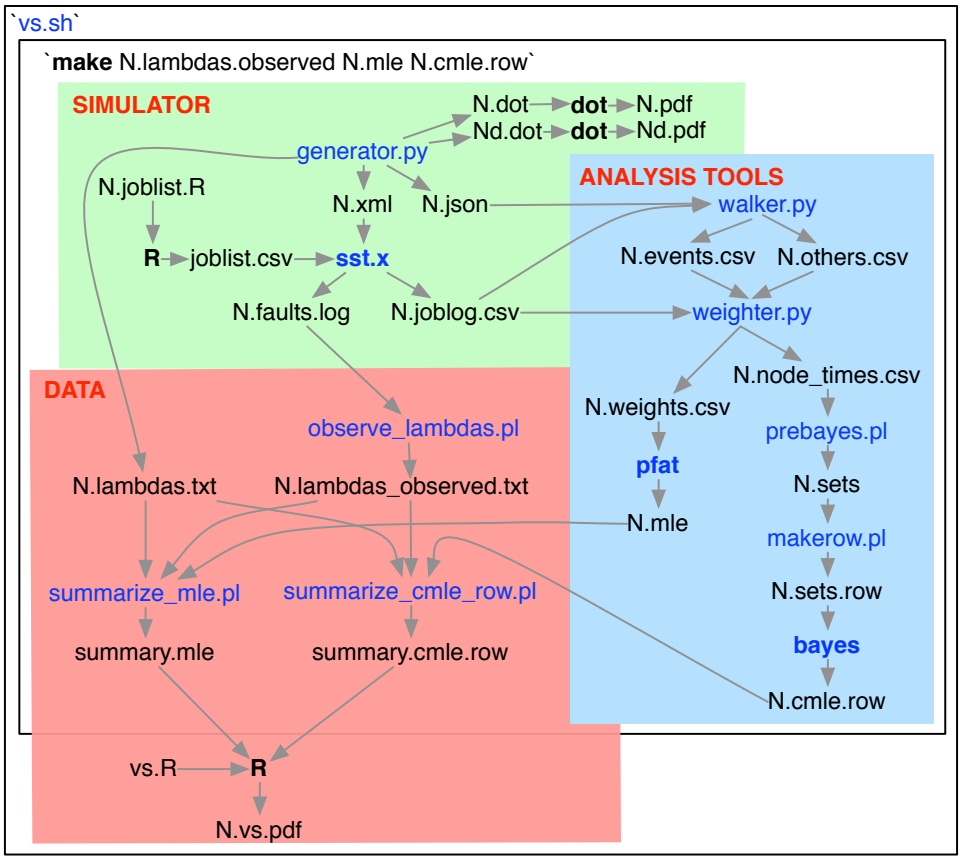


Figure 1: Relationship of data files (non-bold), `yumyum` programs (bold blue) and scripts (non-bold blue), and 3rd-party programs (bold black) used in the `yumyum/trunk/runs/` directory. The **vs.sh** script reproduces the analysis presented in Section 4.1, and utilizes **make**, which manages execution of the toolchain for an arbitrary number of compute nodes N.

and runs both the maximum likelihood estimator (MLE) and conditioned maximum likelihood estimator (CMLE), for an arbitrary number of compute nodes N. The output of such a command is shown in Figure 2.

Estimating the fault rates using ACS's MLE algorithm in "seconds to minutes" for "$O(10)$ failure modes with $O(10^6)$ events" is given as an initial performance target in Section 4.2.1 of the whitepaper provided by ACS early in the project (found in `yumyum/trunk/frequentist/whitepaper`). An N=60 node compute graph has 12 divisors (failure modes), and 750,000 jobs results in 1.06 million events (an event being a change in the utilization state of the compute nodes in the system). Thus, our example is for N=60 (set via the make target), and `numjobs<-750000` set inside the `joblist.R` file[2].

The total execution time of the tool chain shown in Figure 2 is 25 minutes (1500 seconds) on a four core 2.66 GHz Xeon MacPro with 3GB DRAM and SATA 7200RPM disk running MacOS 10.6.8. The calculation of the weights requires the most time (55%), in this case 830 seconds (`weighter.py`). Estimation of fault rates via these weights (ACS's MLE algorithm) is the next largest consumer of time (15%) at 235 seconds on four processors (`mpirun -np 4 pfat.exe`) - thus meeting the above initial performance target. `pfat.exe` is currently the only parallelized tool in the chain. CMLE estimation of fault rates is next (9%, `bayes`), at 142 seconds. Preprocessing the data into a form more easily parsed by the `bayes` tool requires another 138 seconds (9%). These account for almost 90% of total time, with the next most significant runtimes visible in Figure 2, and others omitted as negligible.

For finer control of the tool chain, any of the intermediate file names can be given to `make` as targets, and any tool can be run manually, as detailed in following sections of this report. Or for a coarser level of control, the `vs.sh` script can be used to reproduce all the analyses described in section 4.1. This script performs all steps for a comparative analysis of the estimation methods, on various graph sizes (N=4,6,8,15,30,32), with few to many jobs (100 to 8000), including confidence intervals.

1.3 Install or Build

Pre-compiled MacOS binaries and libraries, current as of the date of this report, are included in the `yumyum` repository. To install and use them in a `bash` shell, do the following:

```
tar -C /usr/local -xf yumyum/trunk/MacOS_Binaries.tar.gz
source yumyum/trunk/.profile
```

These use `/usr/local/yumyum` as the install directory. You may need change the `tar ...` command to `sudo tar ...` for sufficient write permissions. If an alternate directory or shell is preferred, simply untar where desired, and use the provided `.profile` as a guide to modify your environment. All prerequisite libraries are included, such that revision and building of the `yumyum` programs (`pfat.exe`, `bayes`, `sst.x`, and `yummy`) is possible without building everything from scratch.

To build and install everything from scratch, execute `make` in the `yumyum/trunk` directory. The Makefiles download, configure, build, and install all prerequisites and `yumyum` tools. The only exception is compilers, for example on MacOS the XCode package should be manually installed first. For an alternate install directory, change `PREFIX` in `yumyum/trunk/Makefile`,
`yumyum/trunk/simulator/Makefile`, and `yumyum/trunk/bayesian/Makefile` before building. For non-MacOS builds, you will also need to make the small edit described at line 46 of `yumyum/trunk/simulator/Makefile`. See the comments in the `Makefile`s for additional information. For help, contact the first author of this report.

[2]Simply edit `joblist.R` to set number of jobs as desired.

```
$ /usr/bin/time make 60.lambdas.observed 60.mle 60.cmle.row
../generator/generator.py --min 1 --max 24 -n 60 --lambdas=60.lambdas
Save new Lambdas file to 60.lambdas
Done

perl -ne '/^maxjobsize/?print "maxjobsize<-60\n":print' joblist.R > 60.joblist.R
/usr/bin/time R CMD BATCH 60.joblist.R
       11.95 real        11.42 user         0.29 sys

/usr/bin/time sst.x --sdl-file 60.xml &> 60.sst
       57.26 real        44.21 user         5.26 sys     # (extracted from 60.sst for this figure)

mv failures.log 60.failures.log
mv joblog.csv 60.joblog.csv
./observe_lambdas.pl 60.failures.log > 60.lambdas.observed

/usr/bin/time ../walker/squirrel_walker.py -g 60.json -j 60.joblog.csv -t 60.others.csv -e 60.events.csv
joblog reader            OK
processing line 10000 from joblog file
        Warning[16203]: Job 16206 has identical start and stop times (t=128131157).
...
processing line 800000 from joblog file
Wrote others data to '60.others.csv'.
Wrote events data to '60.events.csv'.
Done.
       81.01 real        77.88 user         1.49 sys

/usr/bin/time ../walker/weighter.py -g 60.json -j 60.joblog.csv -o 60.others.csv -e 60.events.csv -w 60.weights.csv -t 60.node_times.csv
events reader            OK
joblog reader            OK
others reader            OK
Processed 10000 event file lines.
...
Processed 1110000 event file lines.
Wrote weights data to '60.weights.csv'.
Wrote node_times data to '60.node_times.csv'.
Done!
      830.39 real       809.54 user         7.72 sys

/usr/bin/time mpirun -np 4 pfat.exe --filename=60.weights.csv > 60.mle
      235.08 real       700.54 user        22.08 sys
./summarize_mle.pl 60 >> summary.mle

/usr/bin/time ../bayesian/prebayes.pl 60.node_times.csv
      138.36 real       134.50 user         2.81 sys
./makerow.pl 60.sets > 60.sets.row
/usr/bin/time bayes -i 60.sets.row -o 60.cmle.row -s 60.bayes.summary.row
Assuming rowconstant lambdas!
      142.15 real       141.84 user         0.22 sys
./summarize_cmle_row.pl 60 >> summary.cmle.row

     1500.21 real      1922.82 user        40.04 sys     # (total toolchain execution time)
```

Figure 2: Example toolchain execution in `yumyum/trunk/runs/`.

2 Simulator

This section describes the tools in the chain from graph generation to simulation.

2.1 Generating the system graph (`generator.py`)

`generator/generator.py` is a Python script which generates directed graphs. Its usage appears in Figure 5. Given a total number of compute nodes (N), it determines all the divisors of N and connects them according to YumYum specification. Each node in the graph has a fault rate attribute ("lambda", in faults per year), the value of which is assigned by a uniform random number generator, ranging from the requested `--max` and `--min`. The unit of faults per year was selected because this is a typical unit for node faults in real systems. The fault rates of all components in a given divisor row are equal, unless `--unique_fault_rates` is given in which case all components are assigned a unique fault rate. Failure rates are sorted from highest at leaf nodes to lowest at root node, unless `--unsorted` is given in which case they are unsorted. If the filename given to the `--lambdas` option exists, the fault rates therein are used, otherwise the rates are randomly generated and saved in this file. The easiest way to manually set fault rates is to run the command once such that it creates the lambdas file as output, edit the file, and then rerun `generator.py` which will read it as input.

The graph is saved in several formats. The `.xml` is ingested by the `sst.x` simulator, the `.json` is used by the simulator postprocessing tool `walker.py`, and the `.dot` files are used by the publicly downloadable GraphViz program to visualize the graphs, if desired. These include the full graph `N.dot` and a graph of only the divisors `Nd.dot`, as shown in Figures 3 and 4. These can be produced via the commands `make N.pdf` and `make Nd.pdf` respectively in the `yumyum/trunk/runs` directory.

All downstream tools in the chain operate on arbitrary acyclic graphs, except for `pfat.exe` whose model is based on YumYum graph structure.

2.2 Generating the job list to simulate (`joblist.R`)

This is a very short script for the open-source **R** program, enabling job distributions and parameters to be easily explored by editing the script. It currently results in `joblist.csv` having three columns: `jobid, duration, size`. The first are integers ascending from 1, `duration` indicates the number of seconds the job will try to run, and `size` indicates the number of compute nodes the job requires. The latter two are drawn from an exponential random number generator. The mean job duration is set to 1 day, the mean job size is set to N/4. Random job sizes greater than N are set to N (aggregating the tail of the distribution at N, such that job sizes are not strictly exponentially distributed). The number of jobs is also hard-coded - simply edit the file to change it (or the distribution which sizes and durations are drawn). `make` edits numjobs automatically, resulting in an `N.joblist.R`. While running this script, `R` always saves the joblist to `joblist.csv`, since this filename is currently hardcoded in `sst.x`.

2.3 Simulating jobs and faults (`sst.x`)

`sst.x` is the binary name generated by the Structural Simulation Toolkit (SST). SST reads in an .xml file describing components, attributes, and connections. It then performs a simulation of the system. It can be used to simulate a wide variety of systems, ranging from circuits at the gate level to supercomputers at the node level (YumYum's case).

Two SST components have been written for the YumYum project. The `resil` component is a node which generates a fault event randomly drawn from an exponential distribution parameterized by the node's fault

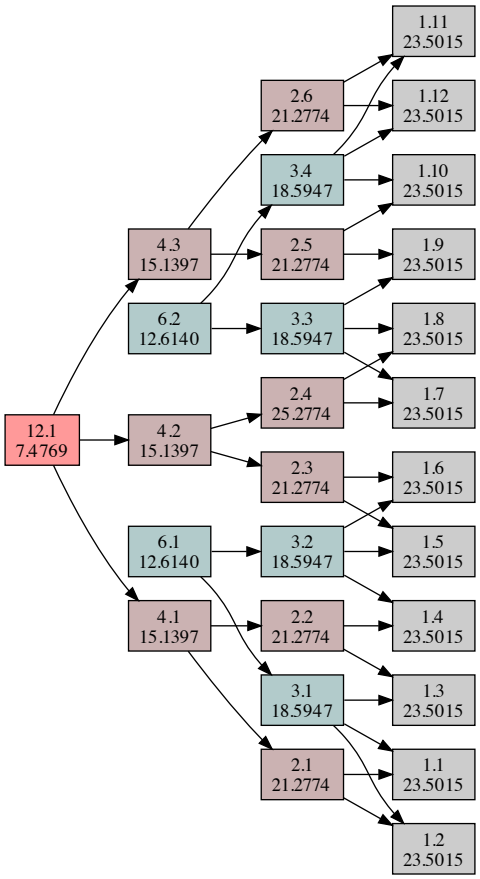


Figure 3: A sample N=12 components graph (`12.pdf`). The top number in each box is the node name D.i, indicating the i'th component on divisor row D. Fault rates appear below the node name, which `sst.x` uses to parameterize the random number generator used to generate node failure times. This value is in terms of faults per year, and we refer to it as the "true" fault rate. The longer a simulation runs, the closer the "observed" fault rate will be to the "true".

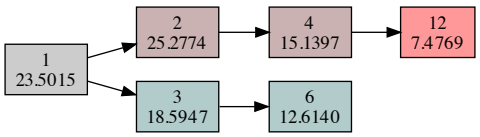


Figure 4: The `12d.pdf` divisors graph corresponding to the components graph in in Figure 3. The fault rate below the divisor indicates the maximum for all components in that divisor "row".

```
$ generator.py  --help
Usage:
        -n <N>  Set the number of nodes, N.  (REQUIRED)
                Must be a positive integer, greater than 1.
        -v      Verbose Mode (OPTIONAL)
        -d      Debug Mode (OPTIONAL)
        --lambdas=<filename>    Filename storing a CSV list lambdas.
        --max=N  Max number of faults per year (default=1)
        --min=N  Min number of faults per year (default=0)
        --unique_failure_rates  Assign a unique fault rate per component (default=False)
        --unsorted      Do not sort fault rates (default=False)
        --help  Print help message and exit
Output:
        N.xml  : SST input XML file for an N-component graph.
        N.json : JSON array representation of an N-component graph.
        N.dot  : GraphVIZ file for an N-component component graph.
        Nd.dot    : GraphVIZ file for an N-component divisor graph.
```

Figure 5: `generator.py` usage

rate (its "lambda" attribute in `N.xml`). Via connections to other `resil` components, the effects of these faults propagates through the system, eventually leading to job failures at leaf nodes.

The other component is `sched`, which reads in the aforementioned `joblist.csv`, and allocates them to leaf nodes in the graph (which represent compute nodes). The allocation process is very basic: given a job of size J, if at least J nodes are available (not already running a job), they are allocated to the job in lexical order of the $D.i$ names. It writes `joblog.csv` when jobs finish successfully (no fault encountered), or fail due to a fault on the leaf node or an upstream node (as determined by the graph structure). `joblog.csv` includes jobid, start time, end time, pass or fail (0 for pass, 1 for fail), and a list of the compute nodes the job utilized. This is the "4-data" model. In addition, it writes a `joblog-stream.csv` file. This is the same as `joblog.csv`, but includes job start records and is sorted by order of occurrence (like job logs from real systems), which enables streaming evaluation by the `yummy` tool (see section 3.3). The simulation terminates when all the jobs in `joblist.csv` have been simulated.

The time granularity of faults and jobs is a second. Fault propagation happens at subsecond granularity. Faulted nodes are availalbe for use by another job at the following second - the scheduler currently requires one simulated second between a job end and start. Jobs which start and end within the same simulated second (start, fault, fail) are omitted by downstream tools by default.

A `faults.log` file is also written, consisting of two columns: component name and time of fault. This is postprocessed into the `N.lambdas_observed.txt` file. The longer a simulation runs, the closer the "observed" fault rates will be to the "true" fault rates in the `N.json` graph. The `faults.log` file was also used to verify that simulated faults match their specified distributions.

```
$ walker.py --help
Usage: walker.py [options]

Options:
  -h, --help            show this help message and exit
  -g GRAPH_FILE, --graph=GRAPH_FILE
                        INPUT : Graph data file, in JSON format.  Default:
                        graph.json
  -j JOBLOG_FILE, --joblog=JOBLOG_FILE
                        INPUT : CSV JobLog file.  Default: joblog.csv
  -e EVENTS_FILE, --events=EVENTS_FILE
                        OUTPUT : Event changes list.  Default: events.csv
  -t OTHERS_FILE, --others=OTHERS_FILE
                        OUTPUT : Other job components listing.  Default:
                        others.csv
  --stats               Save and print out some stats (under development).
                        Default= none
  -v, --verbose         Verbose mode, adds some progress messages, etc.
                        Default= none
  -d, --debug           Debug mode.  Adds lots of debugging trace information.
                        Default= none
  --profile             Enable profiling of the app.  Default= none
  --experimental        Enable experimental code (developers only).  Default=
```

Figure 6: `walker.py` usage.

3 Analysis Tools

This section describes the portion of the tool chain from the reading of simulation results to the writing of fault rate estimates.

3.1 Determining "events" from job logs (`walker.py` and `squirrell_walker.py`)

This is a python script which loads the `.json` graph into memory, and then makes a single pass through `joblog.csv` in streaming fashion (it does not load the entire file into memory). It forms the set of all non-compute nodes depended upon by each job, and writes them to `others.csv` (two columns, jobid and components). As is does this, it also determines event boundaries, defined as a change in the utilization state of nodes in the system. After all jobs have been processed, it writes these to `events.csv`, along with a list of which jobs are starting, ending, or still running at the time. This is basically a Turing tape which maps jobs into events. `walker.py`'s usage appears in Figure 6.

`walker.py` walks the graph for every event, whereas `squirrel_walker.py` walks it once at startup and uses a hash during event processing, which makes it a bit faster. Otherwise they are identical.

3.2 Calculating event utilizations and weights (`weighter.py`)

This python script computes the utilization and weight of each D'th row for each event, by making a single pass through `events.csv` in streaming fashion. It also reads through `joblog.csv` and `others.csv` in order to determine which components are used during which events, which jobs passed or failed, and thus which components may have caused the job failure(s). Its usage appears in Figure 7.

It outputs `weights.csv` which has many columns. The first is a time (in seconds), corresponding to the end of an event period. It then has one column for each divisor (in ascending order), indicating the percentage of each divisor row which was utilized by jobs during the event. It then has one column for each divisor (in numerical D'th ascending order), indicating the percentage of each divisor row which could have contributed to the event ending with a job interruption. These are labeled UD and WD where D is the divisor.

```
$ weighter.py --help
Usage: weighter.py [options]

Options:
  -h, --help            show this help message and exit
  -g GRAPH_FILE, --graph=GRAPH_FILE
                        INPUT : Graph data file, in JSON format.  Default:
                        graph.json
  -j JOBLOG_FILE, --joblog=JOBLOG_FILE
                        INPUT : CSV JobLog file.  Default: joblog.csv
  -e EVENTS_FILE, --events=EVENTS_FILE
                        INPUT : Event changes list.  Default: events.csv
  -o OTHERS_FILE, --others=OTHERS_FILE
                        INPUT : Other job components listing.  Default:
                        others.csv
  -w WEIGHTS_FILE, --weights=WEIGHTS_FILE
                        OUTPUT : CSV file containing the weights.  Default:
                        weights.csv
  -t NODE_TIMES_FILE, --node-times=NODE_TIMES_FILE
                        OUTPUT : CSV file containing the node times.  Default:
                        node_times.csv
  -m MIN_SETS_FILE, --min-sets=MIN_SETS_FILE
                        OUTPUT : file containing the min sets for each event.
                        Default: min_sets.dat
  -c COUNT_MIN_SETS_FILE, --count-min-sets=COUNT_MIN_SETS_FILE
                        OUTPUT : file containing the count of nodes in each
                        min set for each event.  Default: count_min_sets.dat
  --keep-zero-time-jobs
                        Keep jobs that start and stop on same cycle.  The
                        default behavior is to ignore jobs that started and
                        ended on the same clock cycle (i.e., event). If this
                        option is provided then those jobs will beincluded in
                        the calculations.  Default=none
  --stats               Save and print out some stats (under development).
                        Default= none
  -v, --verbose         Verbose mode, adds some progress messages, etc.
                        Default= none
  -d, --debug           Debug mode.  Adds lots of debugging trace information.
                        Default= none
  --profile             Enable profiling of the app.  Default= none
```

Figure 7: `weighter.py` usage.

The script reads as far into joblog and others as needed to process each line in events, adding and pruning memory as it goes. If the first job starts at $t = 0$ and is the last to finish, the entire contents of the jobs and others is kept in memory and swapping or out-of-memory errors could occur. This has not been observed, even with N=1890 (32 divisors, including the first five primes: $2, 3, 5, 7, 9$, resulting in 5760 components) and 10^6 jobs.

`weighter.py` also outputs an `N.node_times.csv` file, which has four columns: TIME, NODE, DURATION, and INTERRUPT. One line is written each time any node transitions from being utilized by a job to going idle. DURATION indicates how long the node was utilized when the transition occurred. The `prebayes.pl` script simply reformats this data into a form more easily parsed by the `bayes` tool (reformatting is easier in Perl than C).

3.3 Toward a single analysis tool (`yummy`)

The `yumyum/trunk/yummy/` tool was written to address a number of issues:

- `squirrell_walker.py` consumes 55% of total tool chain execution time (as described in section 1.2).
- `squirrell_walker.py` and `weighter.py` each pass through the full simulation results (two passes total).
- Neither tool addresses the "same-event binning" issue described in section 3.1.2 of ACS's whitepaper

(yumyum/trunk/frequentist/whitepaper/YUMYUM.pdf).

- Neither tool supports dynamic graphs (where the graph structure changes during the simulation). While this capability is not within scope of this contract, it is relevant for future work.
- The tool chain could be shortened.

yummy is written in C++ and uses the Boost library for set and graph operations (a parallelized version of the graph library is also available). Our long-term goal is for yummy to replace the entire analysis section of the tool chain - a single parallelized binary which reads simulation results and writes fault rate estimates in streaming (single-pass) fashion. However, it currently only replaces the functionality of squirrell_walker.py, weighter.py, and prebayes.pl, via a single pass on joblog-stream.csv (described in section 2.3). Its runtime is comparable to the sum of those scripts (no optimization has been attempted), and does not yet support dynamic graphs or "same-event-binning". Furthermore, it is currently limited to simulations whose duration in seconds can be represented by a long integer, whereas the scripts have no such limit. Thus, yummy is not yet the preferred tool, but it is the recommended path for future work. See section 6.4 for next steps.

3.4 Estimation of fault rates via the MLE algorithm (pfat.exe)

A 1030-line C++ implementation of ACS's maximum likelihood estimation (MLE) algorithm is in yumyum/trunk/frequentist/pfat/. It is parallelized using MPI, and utilizes SNL's Trilinos solver libraries. Extensive documentation on the algorithm is available in yumyum/trunk/frequentist/whitepaper/YUMYUM.pdf, with section 4.2.2 providing detailed notes on the Newton solver method used. The tool's usage appears in figure 8. The only required option is --weights, which specifies the name of the input weights file. Other options control the maximum number of Newton iterations (--max-iters), solver tolerance threshold indicating convergence (--tol), and initial values of the fault rates being solved for (--init-value and --init-value-filename).

In order to deal with convergence difficulties arising when a small number of faults is observed, the method has been customized to dynamically set unknowns to zero and remove them from the set of equations. This is described in section 4.2.3 of the aforementioned YUMYUM.pdf whitepaper. pfat.exe automatically performs this elimination dynamically, but unknowns can also be eliminated manually via the --make-zero option.

Although the above method has yielded the greatest level of robustness, an "interior point" algorithm has also been implemented. This algorithm was explored in an attempt to mimic Mathematica's behavior, and is described in detail in pages 39-41 of the Constrained Optimization Tutorial[3] of the Wolfram Mathematica Tutorial Collection. The --barrier-p options control this algorithm, but we found the above elimination strategy to provide better results, so it is the default robustness strategy in pfat.exe.

3.5 Estimation of fault rates via the CMLE algorithm

After multiple discussions with ACS and LANL collaborators, SNL developed and implemented a conditioned MLE (CMLE) algorithm to estimate component fault rates. This algorithm is extensively documented in yumyum/trunk/bayesian/whitepaper/YumYum_notes.pdf and yumyum/trunk/bayesian/whitepaper/IEEE/yumyum_ieee.pdf, the latter of which has been submitted for publication in the *IEEE Transactions on Reliability* journal.

A 388-line C program in yumyum/trunk/bayes/ implements this algorithm. It also contains a sliver of C++ in order to use random number generators from Boost (rather than GSL, per task 3.1.3 in section

[3] See http://www.wolfram.com/learningcenter/tutorialcollection/ConstrainedOptimization/ConstrainedOptimization.pdf

```
$ pfat.exe --help
Usage: pfat.exe [options]
  options:
  --help                          Prints this help message
  --pause-for-debugging           Pauses for user input to allow attaching a debugger
  --echo-command-line             Echo the command-line but continue as normal
  --verbose               bool    Enable verbose output.
  --no-verbose                    (default: --no-verbose)
  --max-iters             int     Maximum nonlinear iterations
                                  (default: --max-iters=100)
  --weights               string  Filename for weights csv data
                                    (default: --filename="")
  --tol                   double  Solver tolerance, e.g. converged if L2-norm of residual vector is < tol*initial_L2-norm.
                                  (default: --tol=1e-12)
  --init-value            double  Initial value used for all problem unknowns
                                  [ use EITHER this option OR --init-value-filename option ]
                                  (default: --init-value=20)
  --init-value-filename   string  filename from which to read in initial values for each problem unknown
                                  [ use EITHER this option OR --init-value option ]
                                  (default: --init-value-filename="")
  --barrier-p-init        double  Initial value for barrier parameter used to constrain pi-values > 0.
                                  (default: --barrier-p-init=0)
  --barrier-p-final       double  Final value for barrier parameter used to constrain pi-values > 0.
                                  (default: --barrier-p-final=0)
  --make-zero             string  Select indices to force pi[index] = 0.0.
                                  (default: --make-zero="")
```

Figure 8: `pfat.exe` Usage

```
$ ./bayes --help
Usage: bayes [options]

  options:
  --help                Prints this help message.
  --input               Input filename containing sets of suspect nodes.
  --output              Output filename containing sample estimates.
  --scale               Gamma scale parameter (fault rate prior, default is 20).
  --shape               Gamma shape parameter (strength of prior, default is 1).
  --burnin              Number of burn-in iterations to perform (default is 500).
  --iterations          Number of iteration samples to output (default is 300).
  --summary             Filename to save summary info to.
```

Figure 9: `bayes` Usage

1.1). Its usage appears in figure 9. The only required options are `--input` and `--output`, which determine the files read from and written to. The `--scale` and `--shape` options control the Bayesian prior, which are the parameters of the Gamma distribution function from which the estimated fault rates are drawn. After performing a fixed number of burn-in iterations, it outputs a fixed number of samples and terminates. These are controlled via the `--burnin` and `--iterations` options respectively.

4 Data

This section describes example data produced by the YumYum tool chain. These results are fully reproducible using the tools described in previous sections. In particular, the `vs.sh` script mentioned in section 1.2 reproduces the data and plots described in section 4.1.

4.1 Comparison of MLE and CMLE results

While the MLE and CMLE acronyms are similar, the algorithms they refer to are mathematically distinct in nature. MLE is a frequentist approach, because it treats the unknowns (fault rates) as fixed values to be solved for, given data and a model (which assumes statistical distributions for the unknowns). CMLE is Bayesian, because it treats the unknowns as random variables (of assumed distributions), and solves for the distribution parameters which best fit the data. Their formulations and `yumyum` implementations are completely distinct.

Figure 10 visualizes the estimated fault rates from each approach, as a function of simulation length. The intent of these plots is to compare the approaches regarding how much data is needed in order to obtain accurate and certain answers. Following the model in ACS's whitepaper (`yumyum/trunk/frequentist/whitepaper/YUMYUM.pdf`), all components within a distinct divisor row ($D.x$) have identical true fault rates for these experiments. In addition, $D.x$ fault rates are unchanged among multiple graph sizes (via use of `generator.py`'s `--lambda` option), for example the true fault rate of $4.x$ nodes is always 20 faults/year. The ordinate (Y-axis) labels indicate the maximum and minimum estimated fault rates (in black), with the true rate given in red (and a corresponding red line which indicates ground truth for each divisor row). The abscissa (X-axis) exactly matches among all panes within a subfigure, whereas only the scale of the ordinate matches (panes may have different Y-offsets). This facilitates a visual comparison of uncertainties across divisor rows, but unfortunately bounds detail by the least certain row.

Red circles indicate the observed fault rate, which are determined from the `faults.log` file described in section 2.3. The estimates from both methods generally follow the observed rate, which converges to the true fault rate as simulation length increases. A green dot indicates the MLE solution, and a blue square indicates the median sample of the CMLE output. The high degree of agreement between MLE and CMLE provides some cross-verification of their formulations and implementations.

Not only do their estimates (blue and green dots) generally agree, but their 90% confidence intervals do as well (blue and green bars), with MLE's tending to be slightly tighter. The significance of the tighter bounds is unclear however, as MLE's intervals are estimated, versus CMLE's which are directly measured. This contrast is a direct consequence of their frequentist vs Bayesian roots. Details on estimating MLE's confidence interval are given in section 7 of the aforementioned `YUMYUM.pdf`, but it is based on the sharpness of the likelihood surface at its solved peak (Fisher information[4]).

When MLE iterations fail to converge due to constraining unknowns to non-negative values (negative fault rates would not be a physically meaningful solution), `pfat.exe` automatically sets the troubling unknown to zero and restarts. This corresponds to the manual `--make-zero` option described in section 3.4. Fault rates for which this has occurred (values of exactly zero) are circled in black, and have no error bars. In effect, `pfat.exe` gives up on these unknowns and does the best it can on the others. This generally occurs when there are too few fault observations to work with. In some cases, this unfortunately results in severe overestimation of remaining rows. Yet, we have found this strategy to yield better results than the interior point method also available in `pfat.exe`, as mentioned in section 3.4.

CMLE's response to the sparse data challenge is different. Instead of convergence issues, it simply follows the prior it was given. MLE is given an initial value for the unknowns themselves, and CMLE is given initial

[4] Also see `http://www.weibull.com/LifeDataWeb/confidence_bounds_exp.htm` equations 15 through 18. The 90% confidence intervals in figure 10 are computed using $K_\alpha = 1.645$ for $\alpha = 0.05$.

values for distribution parameters describing the unknowns (the "prior"). For figure 10, initial rates of 20 faults/year are used in both approaches, via the default values of `pfat.exe --init-value` and `bayes --scale`. The more data given to CMLE, the less the effect of the prior on the final solution. How quickly the prior is overwhelmed by the data is controlled via `bayes`'s `--shape` option. For additional information, readers are directed to `yumyum/trunk/bayesian/whitepaper` and references therein on Markov chain Monte Carlo methods.

Due to the unique structure of the graphs which `generator.py` produces, there are always a total of N $1.x$ nodes, versus a single $N.x$ node. Since this results in a factor of N more simulated $1.x$ node hours than $N.x$ node hours, the $1.x$ nodes have tighter confidence bounds (more data, more certainty). The higher the divisor, the lower the number of simulated node hours, and the greater the uncertainty. Also note that the more compute nodes there are in a system (N), the shorter the simulation time required to simulate a given number of jobs (e.g. 1.98 yrs to run 1500 jobs on $N = 4$ versus 0.81 yrs on $N = 16$). These combined facts make the direct comparison of subfigures in figure 10 difficult. The problem being solved is complex, and more exploration is needed in order to fully understand the results. This document describes tools which enable such exploration.

Early in the project, the unit of faults/year for fault rate was chosen because it is a commonly used to describe node failure rates in real systems (e.g. node MTTF of 5 years). This unit may give the unfortunate impression that the estimation methods require far more data than would be practical for real systems - no one would wait years for an estimate. However, the time unit is irrelevant to the purpose of the experiments - e.g. replace 20 faults/year with 20 faults/day, and observe that actionable estimates are available within days. The YumYum tools enable exploration of what can be known about complex systems from single bits of event information (pass/fail) and dependency structure. This is a general capability, versus units which are case-specific.

A comparison of MLE and CMLE execution times for an N=210 graph are shown in figure 11a. For this problem, on a single processor, the CMLE tool runs in roughly one quarter of the time required by the MLE tool. Since the MLE tool uses a convergence criteria and the CMLE tool does not, it is expected that as number of jobs grows large, MLE time will be less than CMLE time (which will always linearly increase with number of jobs). We recommend the addition of a convergence criteria to the `bayes` tool in section 6.3. As requested by ACS, the MLE tool has been parallelized, Figure 11b shows its scaling characteristics for a large N=1890 graph and 100,000 jobs. `pfat.exe` scales well only for a small number of processes - additional optimizations are possible if greater scalability at larger process counts is desired.

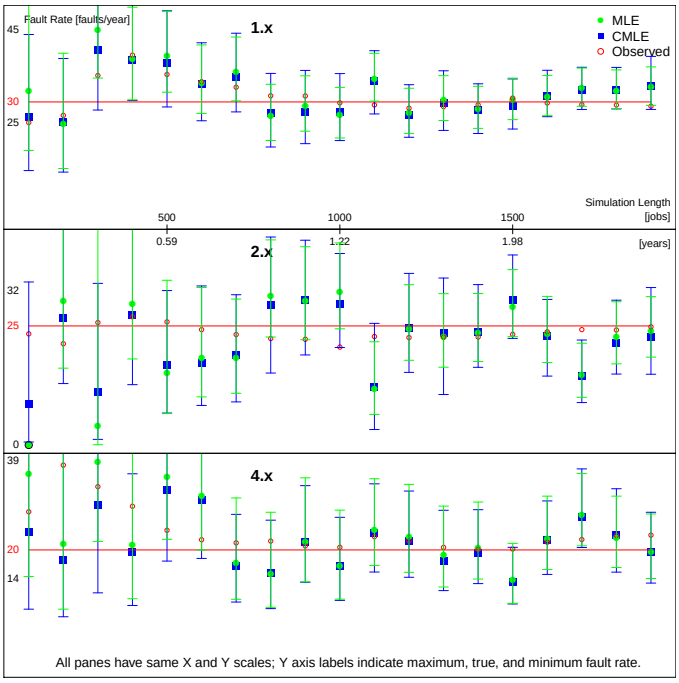
4.2 Estimation of individual node fault rates via the CMLE algorithm

A key assumption in the original problem statement is that all nodes within a divisor row have identical fault rates. This presents a useful starting point, but does not necessarily describe real systems. Whereas it is reasonable to expect similar fault rates among similar component types, defective or damaged components will exhibit unusual fault rates. Efficient and confident identification of such components is important for real systems. As the MLE algorithm was being rethought (`YUMYUM.pdf` section 6), it was recognized that a CMLE approach would enable a natural relaxation of this assumption. A small amount of code in `bayes` effects adherence or relaxation. It may also be possible to revise the MLE formulation and tool, but this was not attempted.

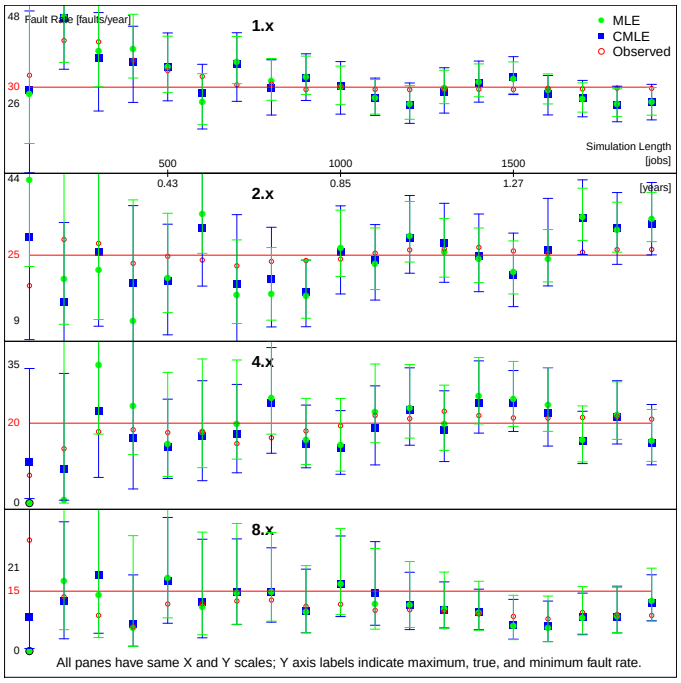
Figure 12 depicts the probability density functions (PDFs) for the estimated fault rates of each node in an N=8 graph. The only experiment parameter which changes between subfigures is the simulation length. It is easy to observe that longer simulations yield sharper peaks, indicating less uncertainty. When seeking to resolve problems, the uncertainty of the options must be weighed against their costs. For supercomputers, the certainty of a component being a root cause must be weighed against the cost of replacing or servicing it (which may require system-wide downtime). Non-financial costs must also be considered. The degree of overlap among choices is critical to the decision making process, and CMLE's ability to quickly estimate complex PDFs offers significant value.

In figure 12, long simulation times are required in order to conclude with certainty that nodes 1.3 and 2.3 have higher true fault rates than their divisor-row-peers. This is not surprising, as the amount of information is very small - a single job pass/fail bit combined with the graph structure to determine sets of suspect nodes[5]. Additional information such as anomaly signals from system logs could be folded into the CMLE analysis, reducing uncertainty without extending observation time. Hooks for adding such information to the CMLE method are mentioned in section 2.3 of `yumyum/trunk/bayesian/YumYum_notes.pdf`. This is a topic of future work.

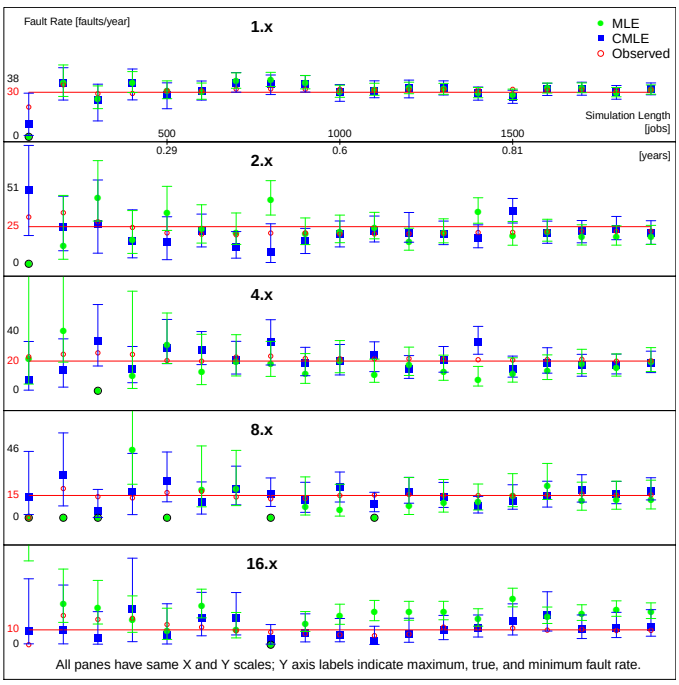
[5]MLE uses even less information - per-divisor summary statistics on sets of suspect nodes and total utilization.



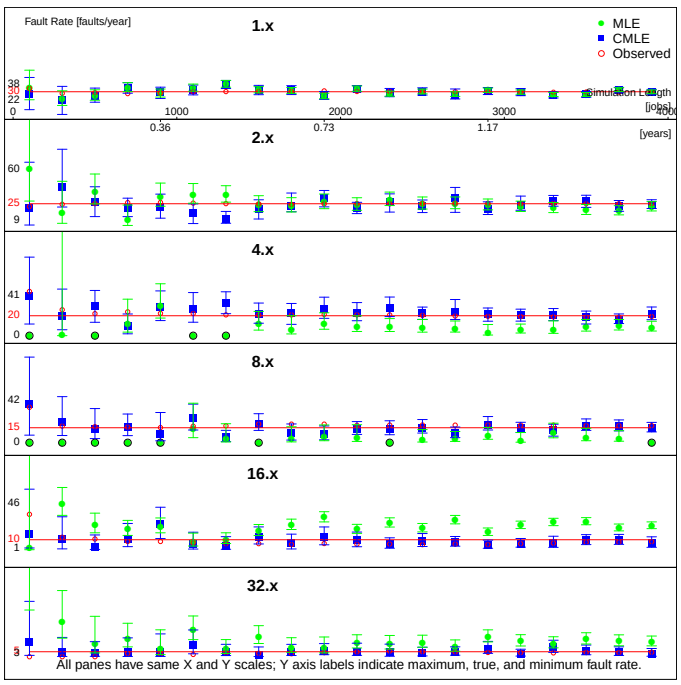
(a) Graph with N=4 compute nodes.



(b) Graph with N=8 compute nodes.

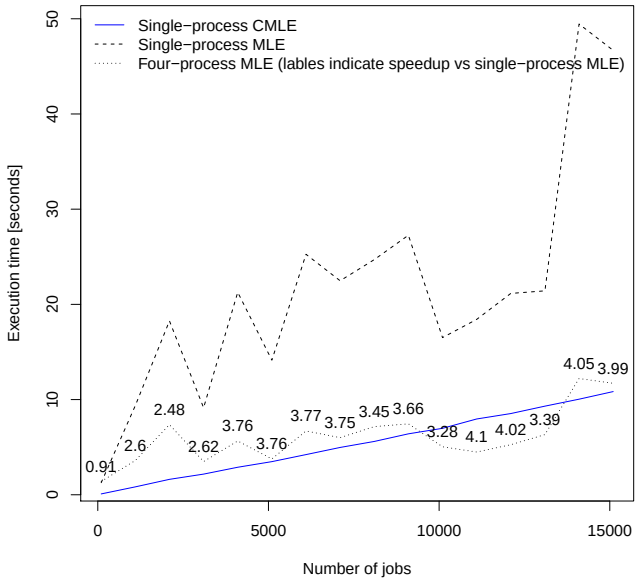


(c) Graph with N=16 compute nodes.

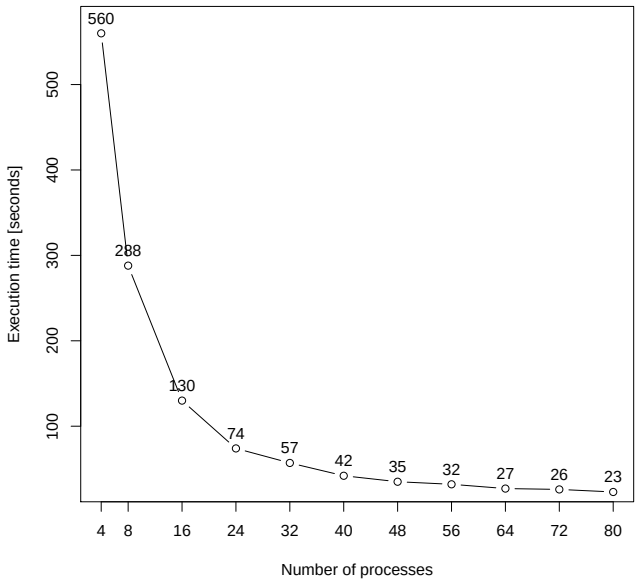


(d) Graph with N=32 compute nodes.

Figure 10: Comparison of the accuracy and uncertainty of MLE and CMLE fault rate estimates. Red lines indicate the true fault rate of D.x components, red dots indicate corresponding observed fault rate (the longer the simulation, the closer observed will be to truth). Bars indicate 90% confidence intervals.

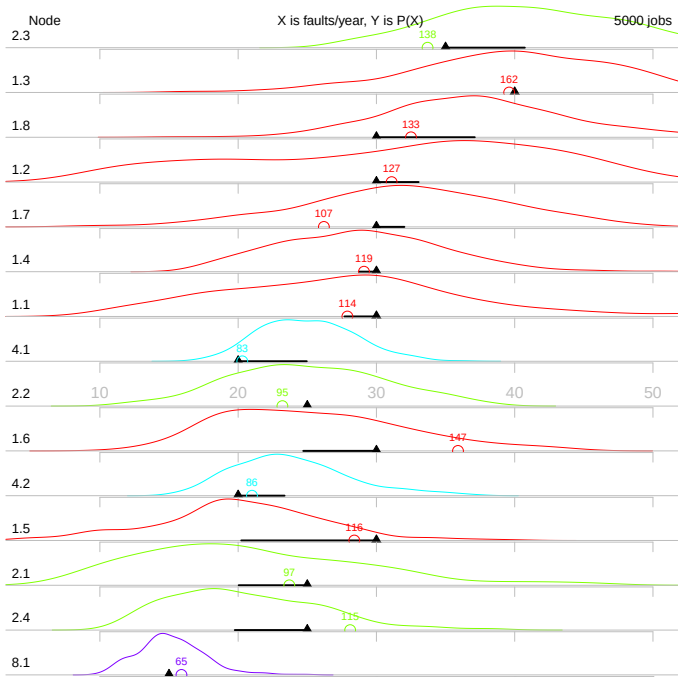


(a) Comparison of MLE and CMLE runtimes as a function of number of jobs on an N=210 graph.

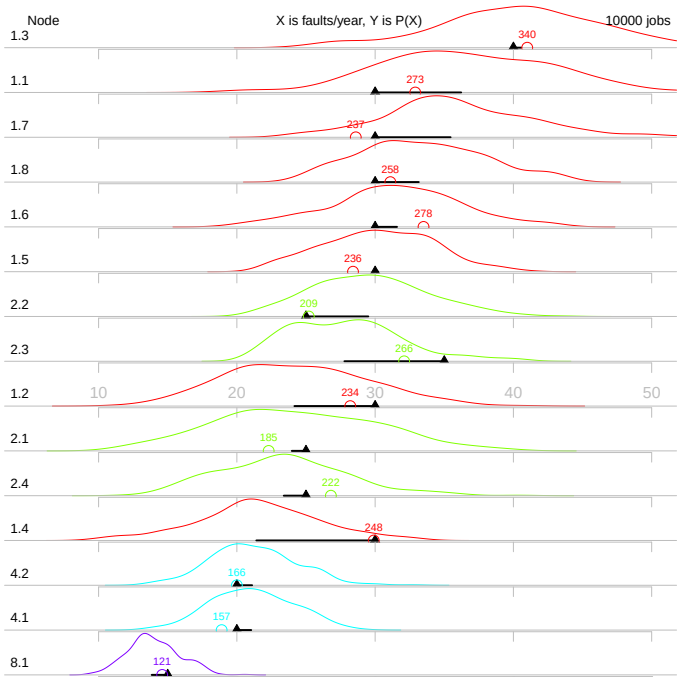


(b) Parallelized MLE tool (`pfat.exe`) scaling performance, on an N=1890 graph (32 divisors) with 100,000 jobs.

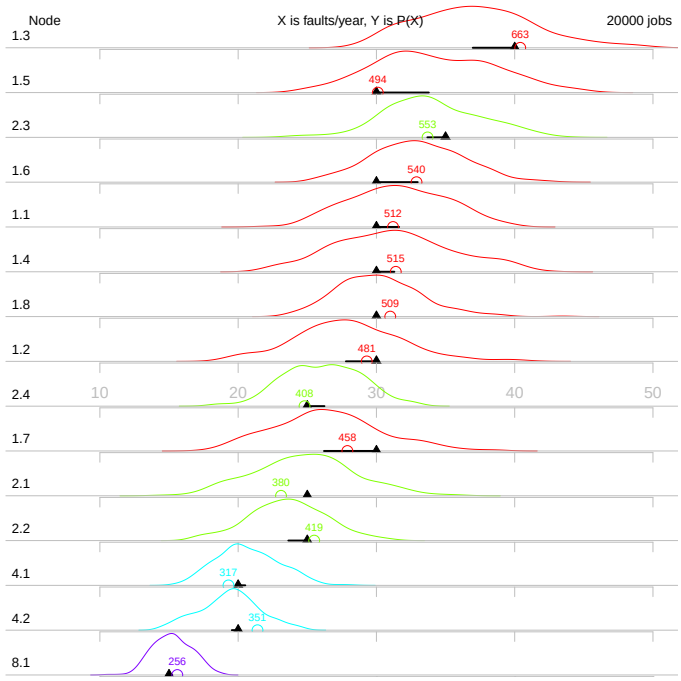
Figure 11: Example analysis tool runtimes.



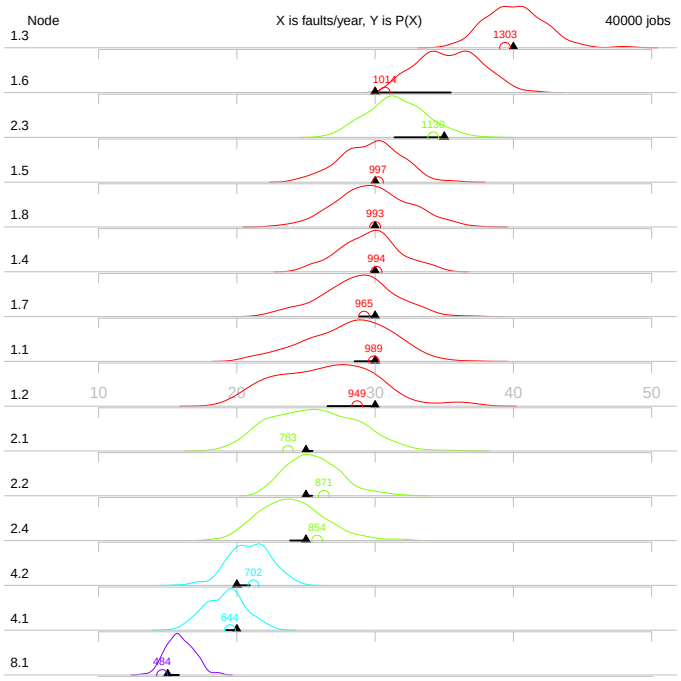
(a) Simulation length of 5,000 jobs (4.2 yrs).



(b) Simulation length of 10,000 jobs (8.4 yrs).



(c) Simulation length of 20,000 jobs (16.8 yrs).



(d) Simulation length of 40,000 jobs (33.8 yrs).

Figure 12: Probability density functions of the CMLE fault rates for each node of an N=8 graph (compute nodes are 1.1, 1.2, ..., 1.8). The black triangle indicates the true fault rate for each node, with a black bar drawn to the median of the Bayesian estimate. A half-circle indicates the observed fault rate, annotated with the number of faults observed. The plots thus visualize the accuracy (size of black error bar) and uncertainty (width of peak) of Bayesian estimates, both of which improve as simulation length increases. Nodes are sorted by decreasing median. Note that in this example the true fault rate of nodes 1.3 and 2.3 do not match the others in their divisor row $D.i$ (which share a common color).

5 Concluding Remarks

This document describes the software written by SNL for the ACS Data Analytics Project, and provides a glimpse of the data it generates. The tools enable experiments on inferring the root cause of job failures on supercomputers, given limited information. SNL has successfully completed all tasks and deliverables. The project was also successful in increasing the dialog between ACS, SNL, LANL, and LLNL on this important research topic, and significantly broadened the range of expertise being applied to HPC resilience at SNL, yielding commensurate advances. SNL is greatful to ACS for the opportunity to participate in the project, and welcomes follow-on work. A proposal for such has been delivered separately.

ACS provided excellent problem definition, algorithms, and ongoing guidance. The implementation of these algorithms shows them to be effective. SNL also found the opportunity to develop additional analytic capabilities to be rewarding. The resulting CMLE algorithm is more robust to sparse-data conditions, faster to execute (1/4 of execution time vs MLE, using a single processor), and simpler to implement (1/3 the lines of code). It estimates the fault rates of individual nodes or groups of nodes (e.g. divisor rows), and eliminates graph structure assumptions (other than directed acyclic). By treating fault rates as random variables, further statistical analysis is possible, including straightforward measurement of confidence intervals (rather than complex estimation). We consider it to be a particularly valuable contribution to the program.

6 APPENDIX: Improvement Ideas

In this section, we recommend tasks to improve the `yumyum` tool chain. Rather than scattering such items throughout the report, they are collected here as a to-do list for future work. Items range from minor to major in both effort required and significance to project goals. They are grouped by tool, and sorted by decreasing priority recommendation within groups.

6.1 Simulator (`sst.x`)

1. The current simulator exits when the input job list is exhausted. An `sst.x` option to indicate a maximum simulation time would enable more intuitive control of experiments - via time rather than number of jobs (which are themselves parameterized by duration and size distributions). Generation of job lists inside `sst.x` would also be handy, but the ability to trace jobs from arbitrary sources (e.g. real systems) should be retained.

2. The current simple allocator being used results in low compute nodes (low in lexical $D.i$ order) being utilized significantly more than high compute nodes. Allocating nodes in numerical i'th order would offer a slight improvement during results review. More significantly, a realistic SST scheduler component is being developed in another SNL project, and recommended as a simulation enhancement, especially if the project targets deployment to real systems.

3. Nodes currently have zero repair time, which is not realistic.

6.2 MLE tool (`pfat.exe`)

1. Another barrier parameter could be added, such that both upper and lower constraints are enforced.

2. The values of these constraints could be automatically determined, using the logic mentioned in `YUMYUM.pdf` section 4.2.3 and implemented in `yumyum/trunk/frequentist/hpc_sim/bound_parse_data`.

6.3 CMLE tool (`bayes`)

1. Currently, `bayes` terminates at fixed number of iterations (controllable via the `--iterations` option). A convergence criteria and associated command-line option could be added, such that iterations would stop upon convergence or a maximum number of iterations, whichever occurs first. This would ensure that the estimates had reached a sufficiently steady state, and that unnecessary iterations are not performed.

2. With a little more parsing logic, `bayes`'s input file could be much shorter. `yummy` outputs such a file, and `weighter.py` could be revised to as well. This would render `prebayes.pl` unnecessary.

3. Instead of relying on the ID column of the input file to map components into same-fault-rate-groups, an option could be added which specifies a filename containing an arbitrary node-to-fault-rate-group mapping. `makerow.pl` could then generate this file instead of munging the ID column based on the D in $D.i$ node names.

4. The CMLE algorithm could be easily parallelized[6], enabling shorter execution times.

[6] See `http://darrenjw.wordpress.com/2010/12/14/getting-started-with-parallel-mcmc/`

6.4 Single analysis tool (`yummy`)

1. The current representation of time as integers in the"4-data" model limits the total possible time span, unless arbitrary precision libraries are used. `yummy` suffers from this limitation. Larger data types could be used, but the real fix would be for joblog times to be formatted and parsed as proper timestamps (e.g. YYYY-MM-DD HH:MM:SS), as they are in real system logs.

2. `yummy` could be linked with the estimation routines of `pfat.exe` and/or `bayes`, yielding a single binary that reads job logs and writes fault rate estimates. Weights and sets would become optional intermediate outputs for verification or alternate analysis tools.

3. `yummy` could be made fully streaming and used as an online tool for monitoring jobs in a live fashion and updating fault rate estimates accordingly (as mentioned in `YUMYUM.pdf` section 4.2.1). Both the MLE and CMLE approaches are amenable to this, via initial values and priors respectively.

DISTRIBUTION:

1	MS 1319	Jim Ang , 1422
1	MS 1322	Sudip Dosanjh , 1420
1	MS 1318	Russel Hooper , 1445
1	MS 1027	Curtis Johnson , 5635
1	MS 1319	Sue Kelly , 1423
1	MS 1326	William McLendon , 1461
1	MS 1327	David Robinson , 1464
1	MS 1319	Arun Rodrigues , 1422
1	MS 1319	Jon Stearley , 1422
1	MS 1027	Michael Stickland , 5635
1	MS 0968	Aaron Williams , 5763
1	MS 0899	Technical Library, 9536 (electronic)
1	MS 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories