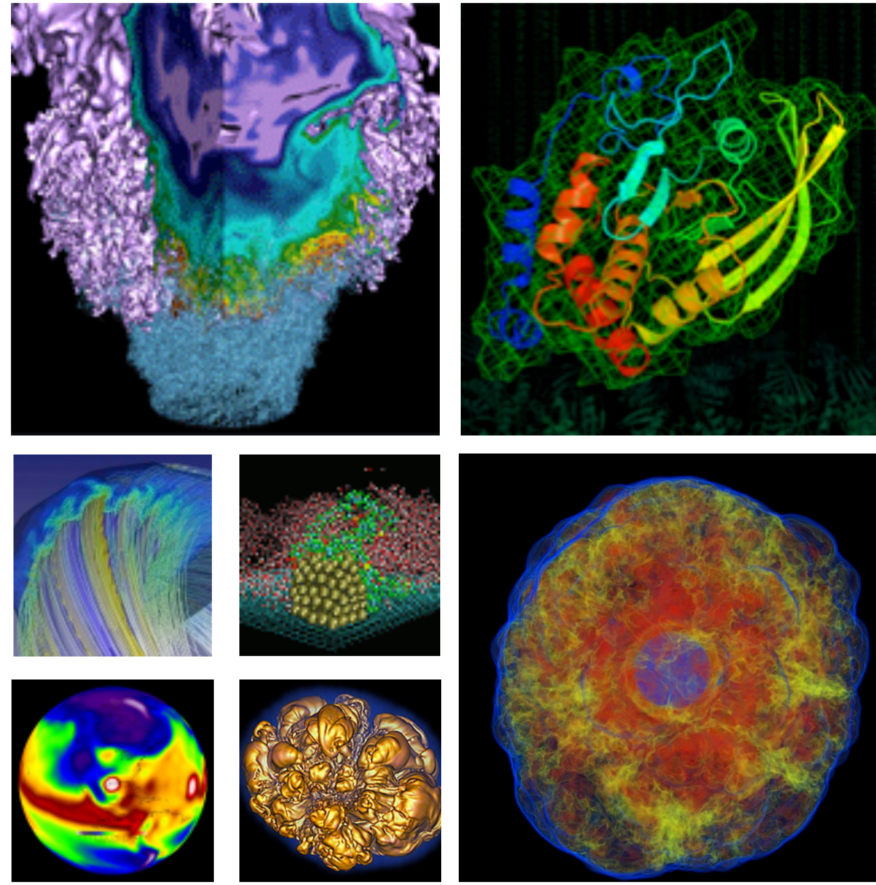
Real-time Process Monitoring on the Cray XC30



NERSC

Douglas Jacobsen
NERSC Computational Systems Group

Cray User Group 2015 – 2015/04/28

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Acknowledgements

- **Shane Canon**
- **NERSC**
- **Joint Genome Institute**

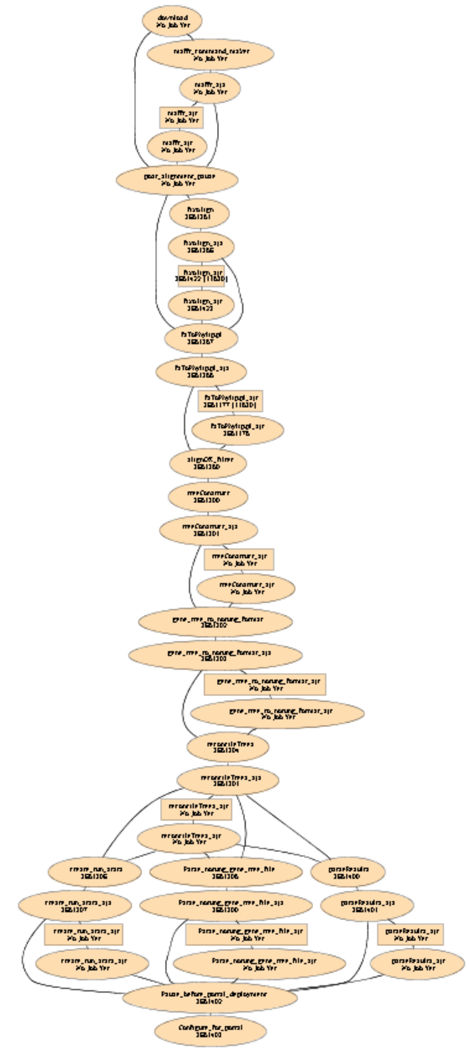
This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract No. **DE-AC02-05CH11231**.

Goals / Motivation

- NERSC developed a real-time monitoring system to discover what is running across a cluster system
- Real-time means that monitoring data is pushed out to central store as soon as it is available
- Want to evaluate if this approach for real-time monitoring will also function in the Cray environment

Goals / Motivation

- **procmon was developed to characterize the workload of a genomics computing resource**
  - Joint Genome Institute
  - Workload is completely different from "normal" NERSC usage
  - Complex hierarchies of processes and resource utilization
  - JGI has a large number of "interactive-only" machines



Goals / Motivation

- **Wanted to answer specific questions**
  - Which executables consume the most CPU time?
  - Which executables consume block the most wall time?
  - Which filesystems and how much IO are used by which users and jobs?
    - Can we identify performance bottlenecks from certain specific jobs running simultaneously?
  - Are there efficient threaded bioinformatics codes in use today?
  - How does batch job work differ from interactive use? Are interactive systems being effectively used and managed?

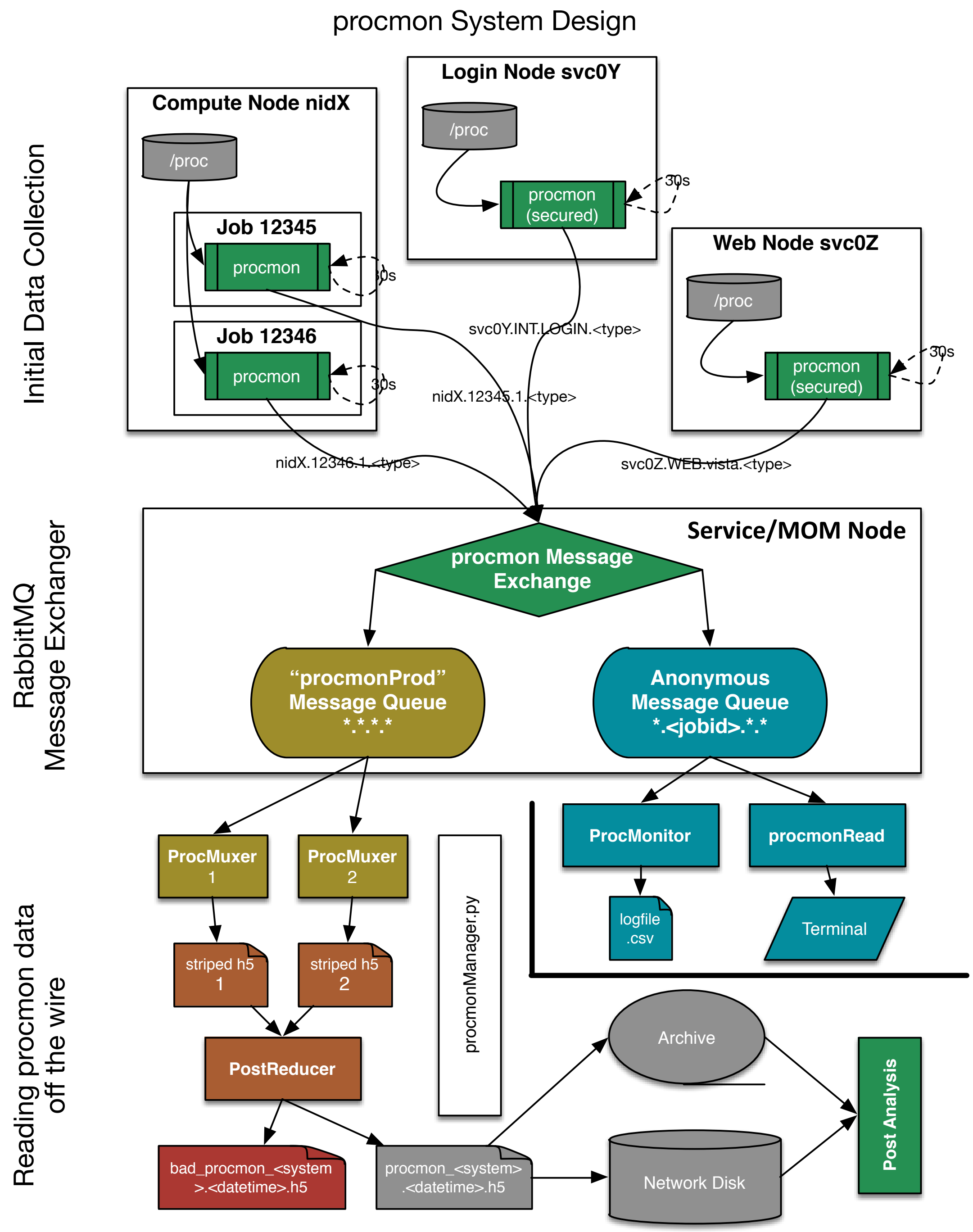
Requirements & Philosophy

- **Monitoring activities should minimally impact running job**
  - Use very little CPU time
  - Do not use any filesystems on the compute node
  - Do not hog bandwidth or use blocking network I/O
  - Do not stat any real files, only read the information the kernel can deliver through /proc
  - Sampling data is sufficient, do not try to capture everything
- **Must be scalable at every level**
  - Sensors must not overwhelm network provider
  - Network message exchanger should accept new messages (or deny them) with no delay
  - Queues of messages awaiting delivery should not build as a matter of course
  - Data recorders should not miss messages or run slower than the messages are generated
- **Process data must retain association with batch job information**
- **Interactive use of systems should be captured**
- **Data should be kept in as raw form as possible to maximize utility for later analysis**
- **Data should be stored in a form compatible with time-series analysis**
  - i.e., not necessarily a database
- **Data should be accessible by users as well as system administrators**
- **Data should be accessible for post-analysis as well as "live" analysis**

procmon Architecture

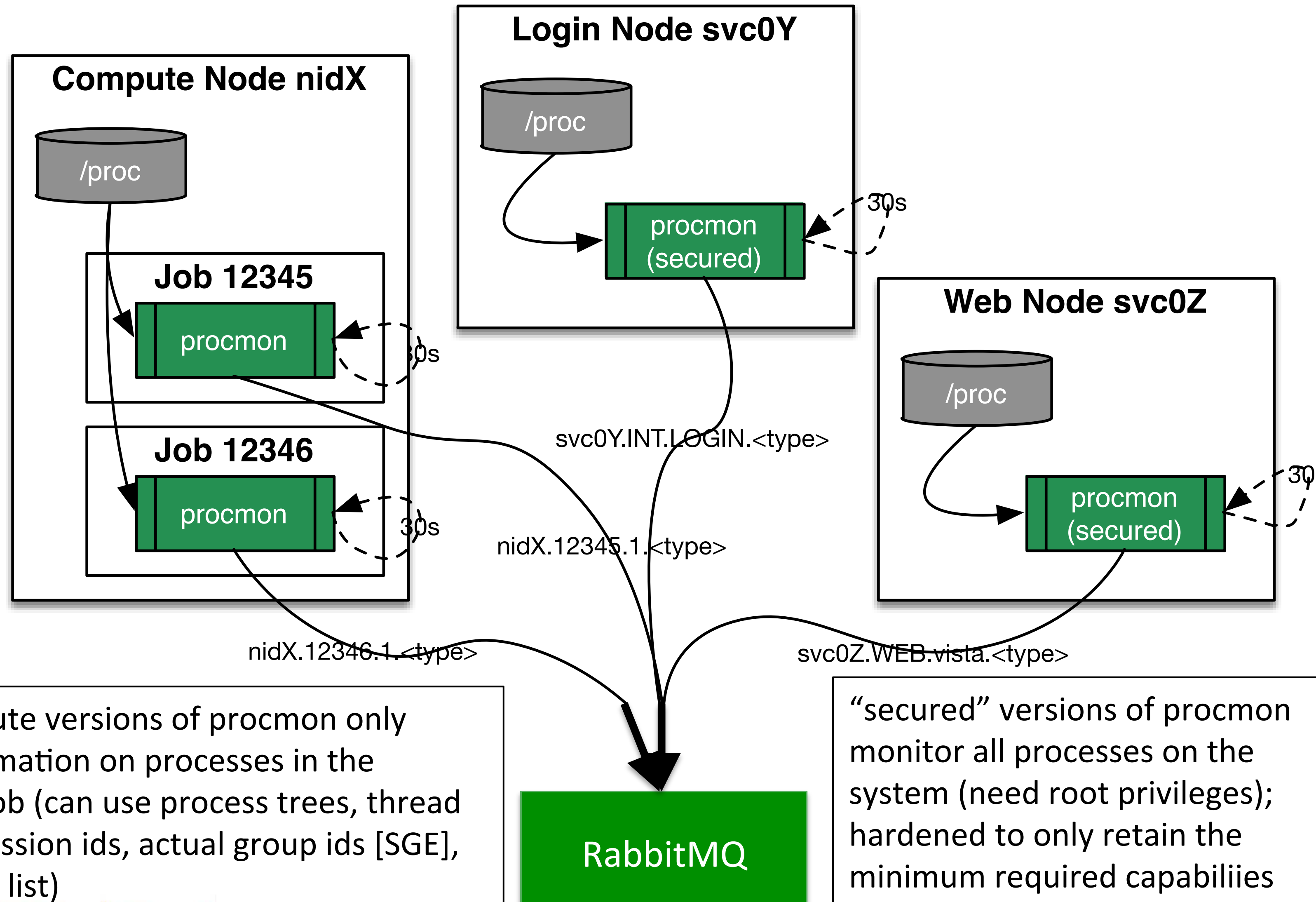
Three major components:

- **procmon sensor**
- **RabbitMQ communication system**
- **Clients to read, analyze, display or save data**



procmon sensor

Initial Data Collection



Batch compute versions of procmon only gather information on processes in the monitored job (can use process trees, thread group ids, session ids, actual group ids [SGE], cgroup tasks list)

"secured" versions of procmon monitor all processes on the system (need root privileges); hardened to only retain the minimum required capabiliies Appropriate for interactive nodes or MOM nodes

procmon sensor: Data Collected

procdata	
identifier	char[IDENTIFIER_SIZE]
subidentifier	char[IDENTIFIER_SIZE]
recTime	unsigned long
recTimeUSec	unsigned long
startTime	unsigned long
startTimeUSec	unsigned long
pid	unsigned int
ppid	unsigned int
execName	char[EXEBUFFER_SIZE]
cmdArgBytes	unsigned long
cmdArgs	char[BUFFER_SIZE]
exePath	char[BUFFER_SIZE]
cwdPath	char[BUFFER_SIZE]
IDENTIFIER_SIZE = 24	
EXEBUFFER_SIZE = 256	
BUFFER_SIZE = 1024	

procfd	
identifier	char[IDENTIFIER_SIZE]
subidentifier	char[IDENTIFIER_SIZE]
recTime	unsigned long
recTimeUSec	unsigned long
startTime	unsigned long
startTimeUSec	unsigned long
pid	unsigned int
ppid	unsigned int
path	char[BUFFER_SIZE]
fd	int
mode	unsigned int
IDENTIFIER_SIZE = 24	
EXEBUFFER_SIZE = 256	
BUFFER_SIZE = 1024	

Optional

The data are broken up into different message types

Base-level functionality has data structures:

- procdata - for the string-like items
- procstat – for the volatile numeric counters
- procfd – for the file descriptors (optional)

Plugin functionality can add more: (in progress)

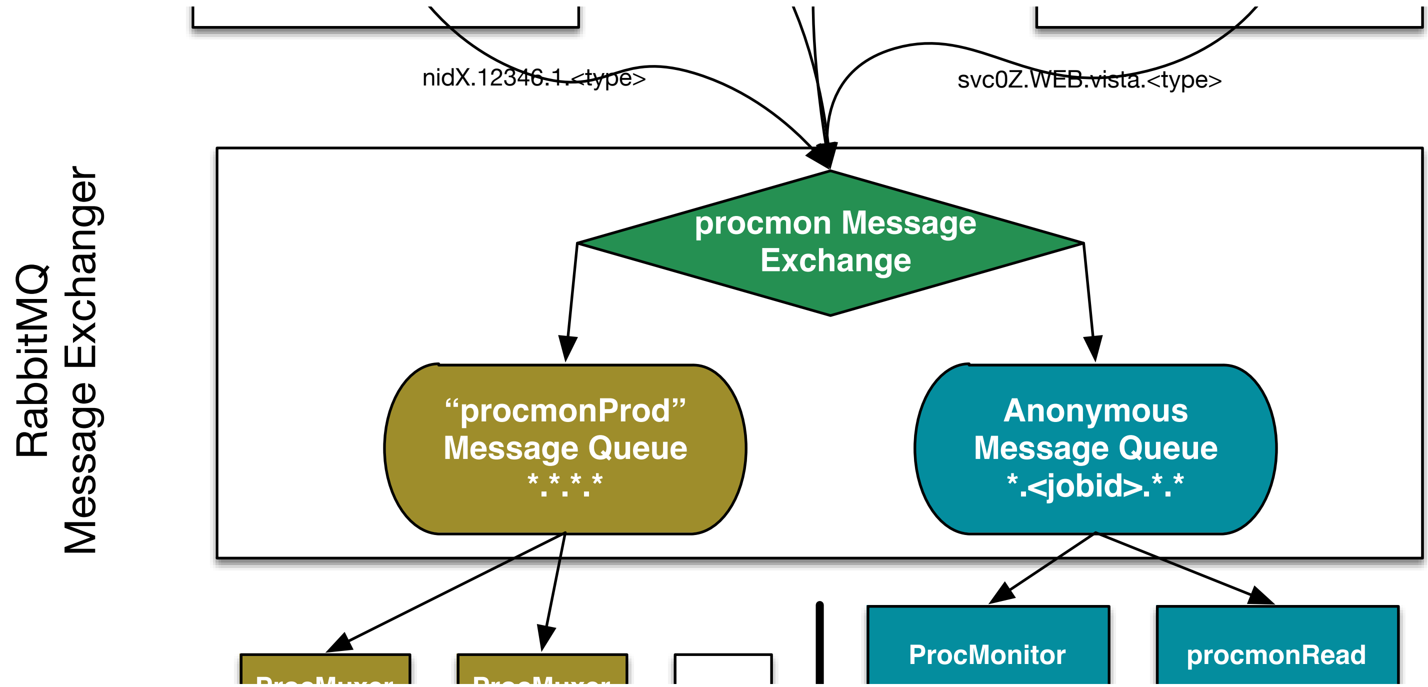
- E.g., mpirank info

procmon sensor: Data Collected

procstat	
identifier	char[IDENTIFIER_SIZE]
subidentifier	char[IDENTIFIER_SIZE]
recTime	unsigned long
recTimeUSec	unsigned long
startTime	unsigned long
startTimeUSec	unsigned long
pid	unsigned int
ppid	unsigned int
state	char
pgrp	int
session	int
tty	int
tpgid	int
realUid	unsigned long
effUid	unsigned long
realGid	unsigned long
effGid	unsigned long
utime	unsigned long (ticks)
stime	unsigned long (ticks)
priority	long
nice	long
numThreads	long
vsize	unsigned long (bytes)
rss	unsigned long (bytes)
rsslim	unsigned long (bytes)
signal	unsigned long
blocked	unsigned long
sigignore	unsigned long
sigcatch	unsigned long
rtPriority	unsigned int
policy	unsigned int
delayacctBlkIOTicks	unsigned long long (ticks)
guestTime	unsigned long
vmpeak	unsigned long (bytes)
rsspeak	unsigned long (bytes)
cpusAllowed	int
io_rchar	unsigned long long (bytes)
io_wchar	unsigned long long (bytes)
io_syscr	unsigned long long (count)
io_syscw	unsigned long long (count)
io_readBytes	unsigned long long (bytes)
io_writeBytes	unsigned long long (bytes)
io_cancelledWriteBytes	unsigned long long
m_size	unsigned long (bytes)
m_resident	unsigned long (bytes)
m_share	unsigned long (bytes)
m_text	unsigned long (bytes)
m_data	unsigned long (bytes)

```
IDENTIFIER_SIZE = 24
EXEBUFFER_SIZE  = 256
BUFFER_SIZE     = 1024
```

RabbitMQ (AMQP) Network Layer



procmon sensor connects to RabbitMQ server at startup, reuses connection for lifetime of sensor (can reconnect if necessary)

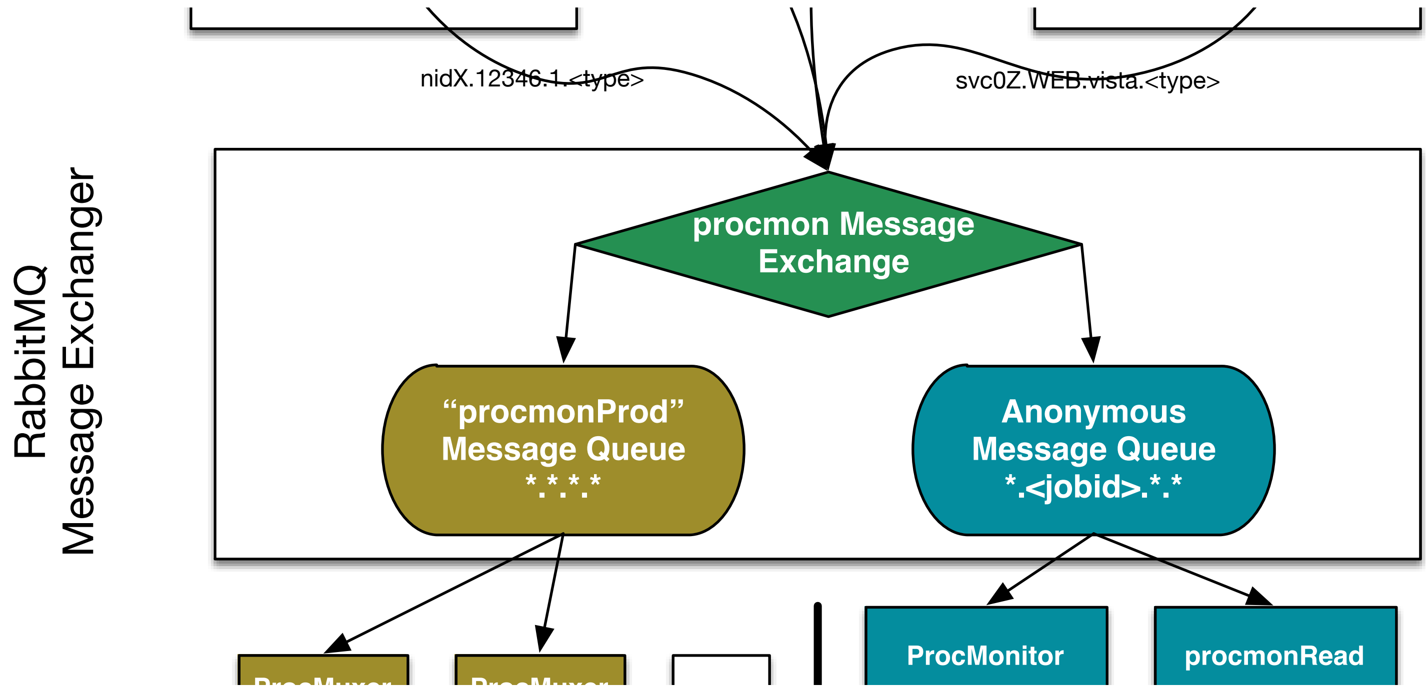
Uses AMQP Topic Exchanges. procmon messages have a routing key like:

<hostname>.<jobid>.<array_taskid>.<type>

Topic exchange allows clients to selectively listen for messages.

RabbitMQ (AMQP) Network Layer

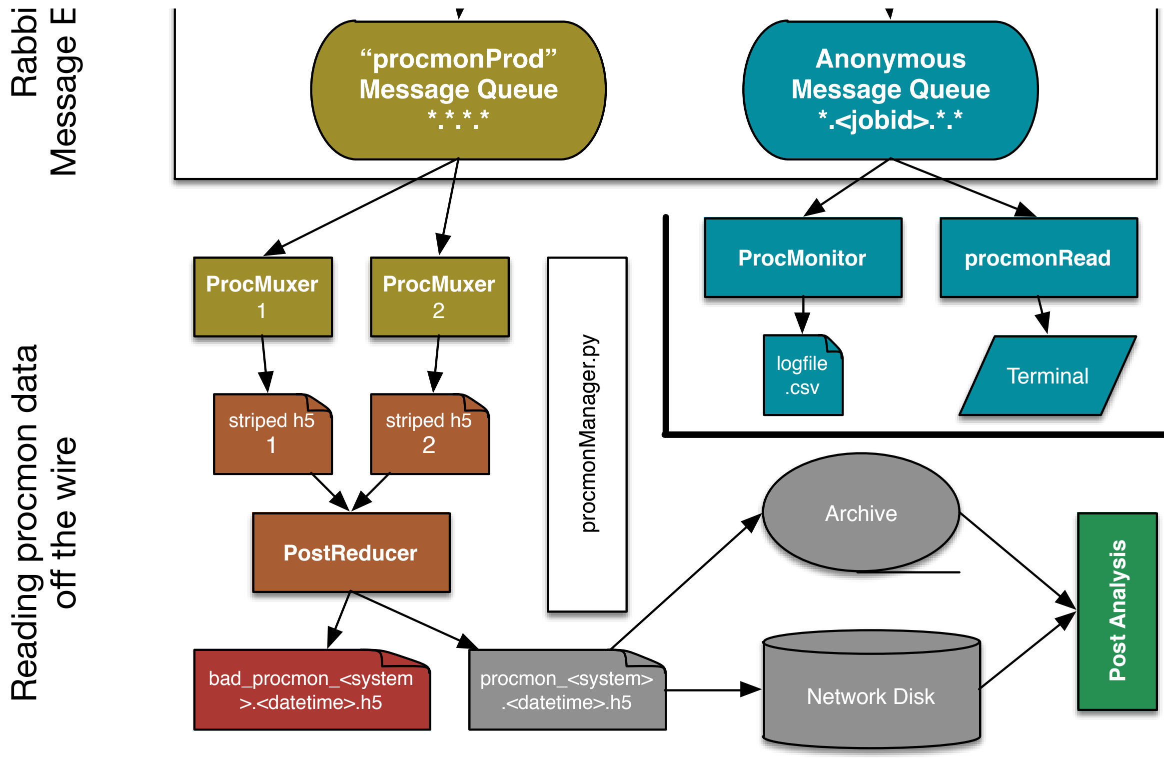
procmon sensors ensure that the *Exchange* is created
The exchange accepts and routes messages.



Listening clients create *queues* bound to the *Exchange* specifying routing tags for in which they are interested.
This division of labor ensures that messages are not stored or kept if there are no listeners.

Data Reduction and Storage

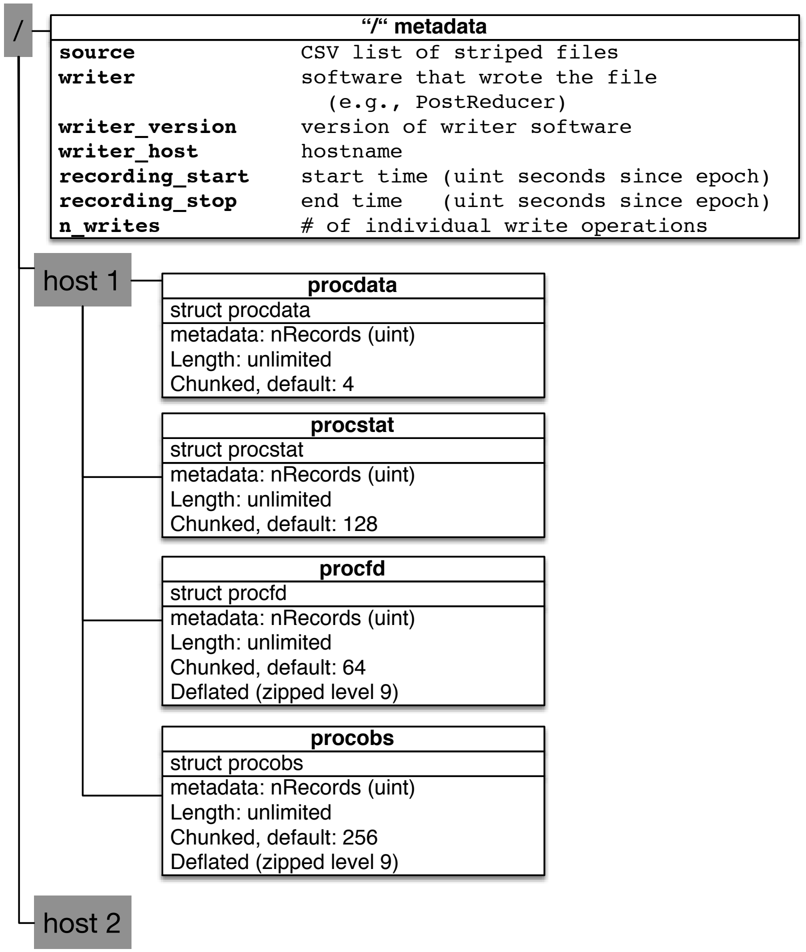
- **A “ProcMuxer” process connects to RMQ reading ‘*.*.*.*’**
  - Multiple muxers can connect sharing the same queue; **stripes messages**
- Muxer writes *all* data to hdf5 file
- Files are written for one hour, then new file



“PostReducer” is run on the muxer files, performs:

- General Q/A – throws out bad data
- Merges data from multiple muxers
- Compresses by keeping only latest unique records

HDF5 File Structure



REPEAT procdata, procstat, etc for each host

- **procmon data archived in per-hour hdf5 files**
- **All data grouped by host**
  - output of gethostname() on the node
- **Plugins/other AMQP producers can generate their own datasets for future expansion**
  - All time/host-related data in single place

Analysis Tools

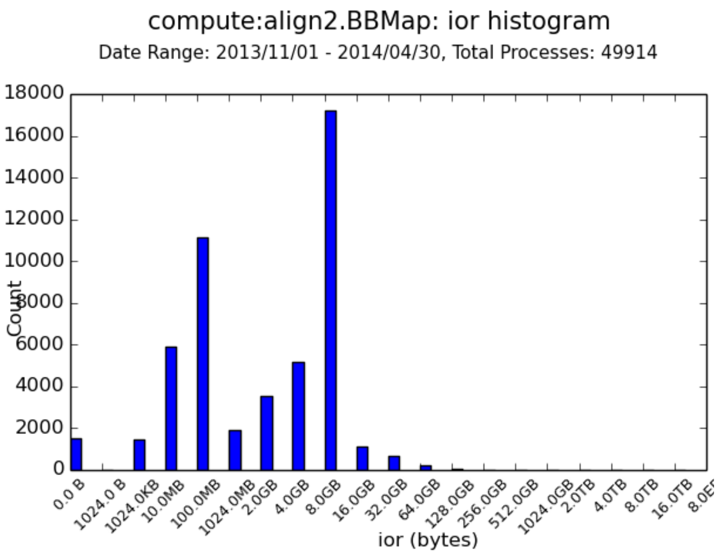
- **qqproc** -- provides row-level filtering and query capabilities, e.g.,

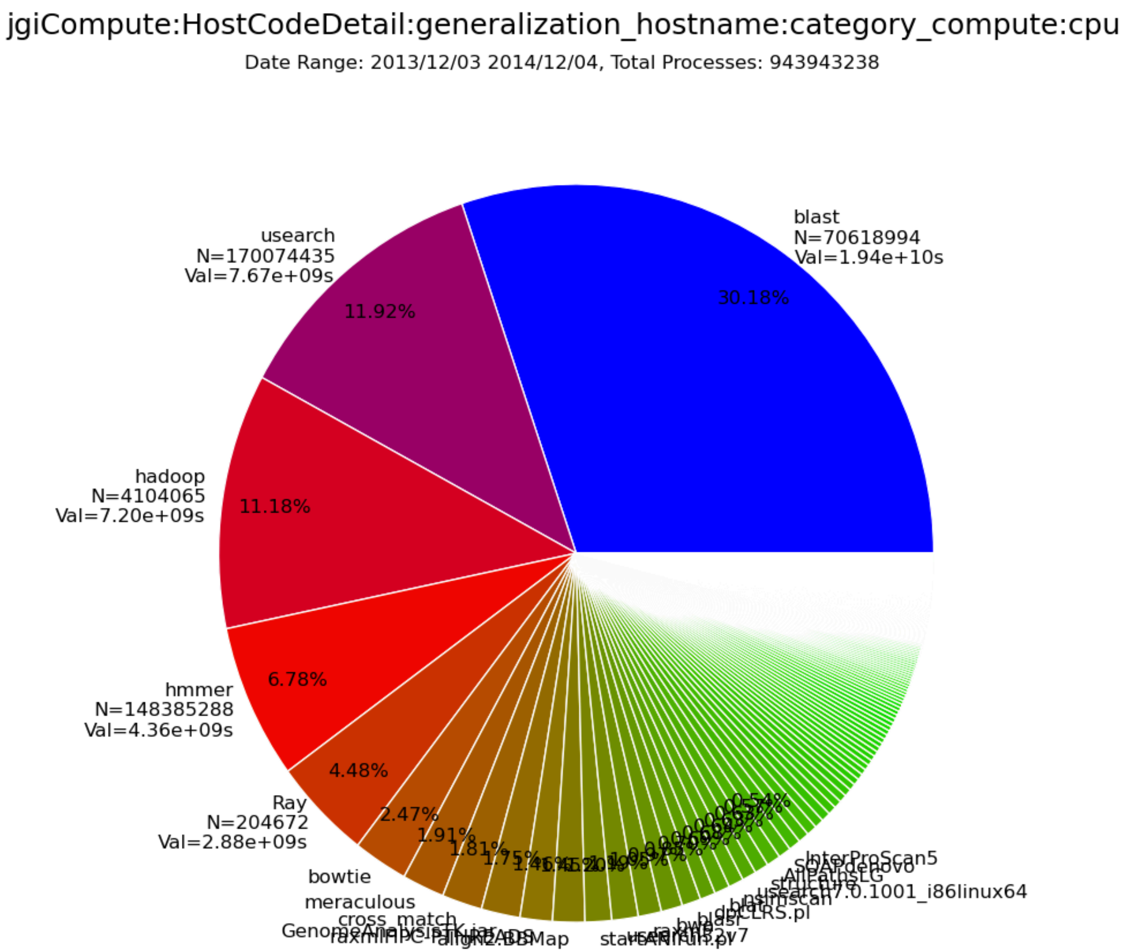
```
qqproc -S 2015-01-05 -q 'user=="gbp" && vmpeak > 10G'
```

- **Python interface using h5py**
- **catjob –** dump process summarizations for a given jobid
- **jobtop –** multinode "top" showing streaming data process data from all messages matching your job
- **Automated process summarization and workload analysis pipeline**

Workload Analysis

- **Analyzed data from 12/3/2013 – 12/4/2014**
- **Included over 1.4 billion processes**

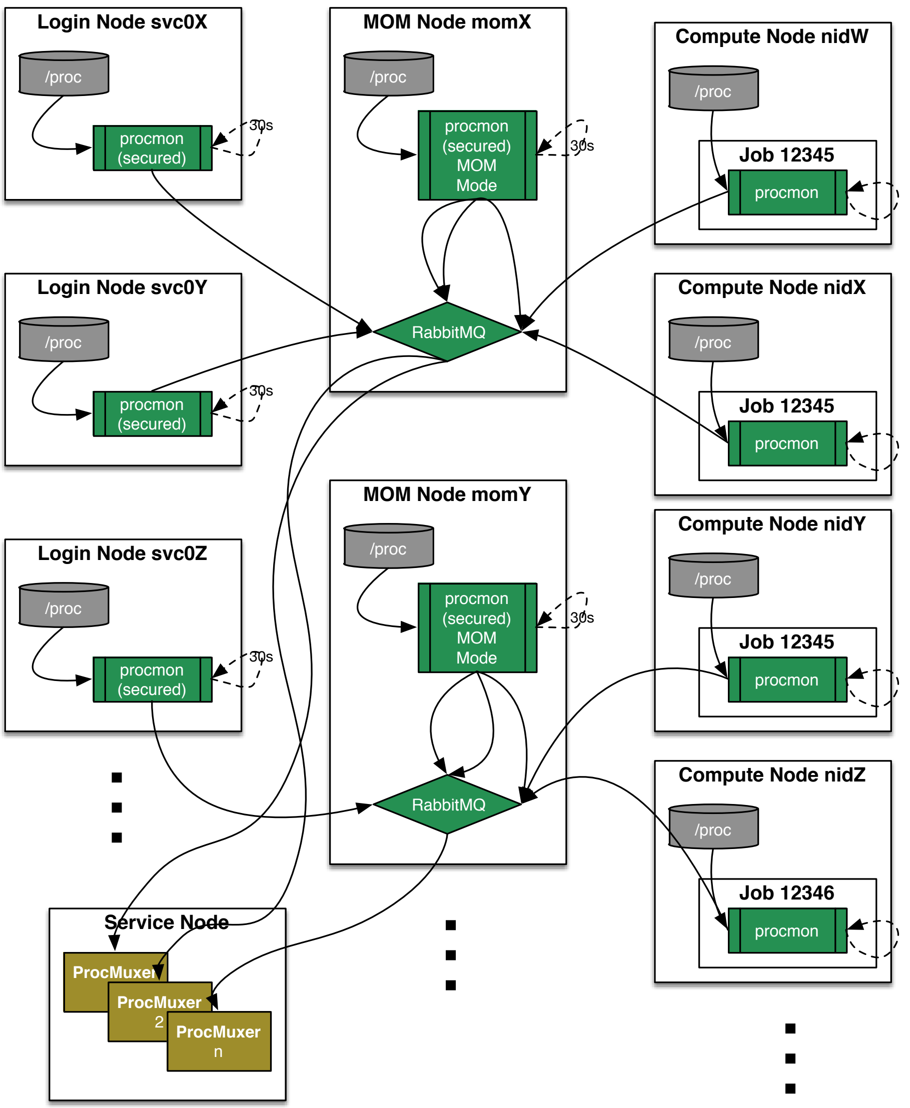




Monitoring Scalability

- Can run multiple RabbitMQ servers to partition monitoring
- Sensors select RMQ server randomly
- Implements a form of cheap High Availability
- Typically, a single RMQ is sufficient for up to several thousand message producers

procmon on the Crays



Options on the Cray

- **Built-in support for Cray cpusets for process tracking (instead of process hierarchy discovery)**
- **Persistent procmon can be run on login and MOM nodes to record interactive usage**
- **MOM or service nodes are the natural location for RabbitMQ servers**
- **Starting on compute nodes:**
  - Opt-in with binary wrapper:

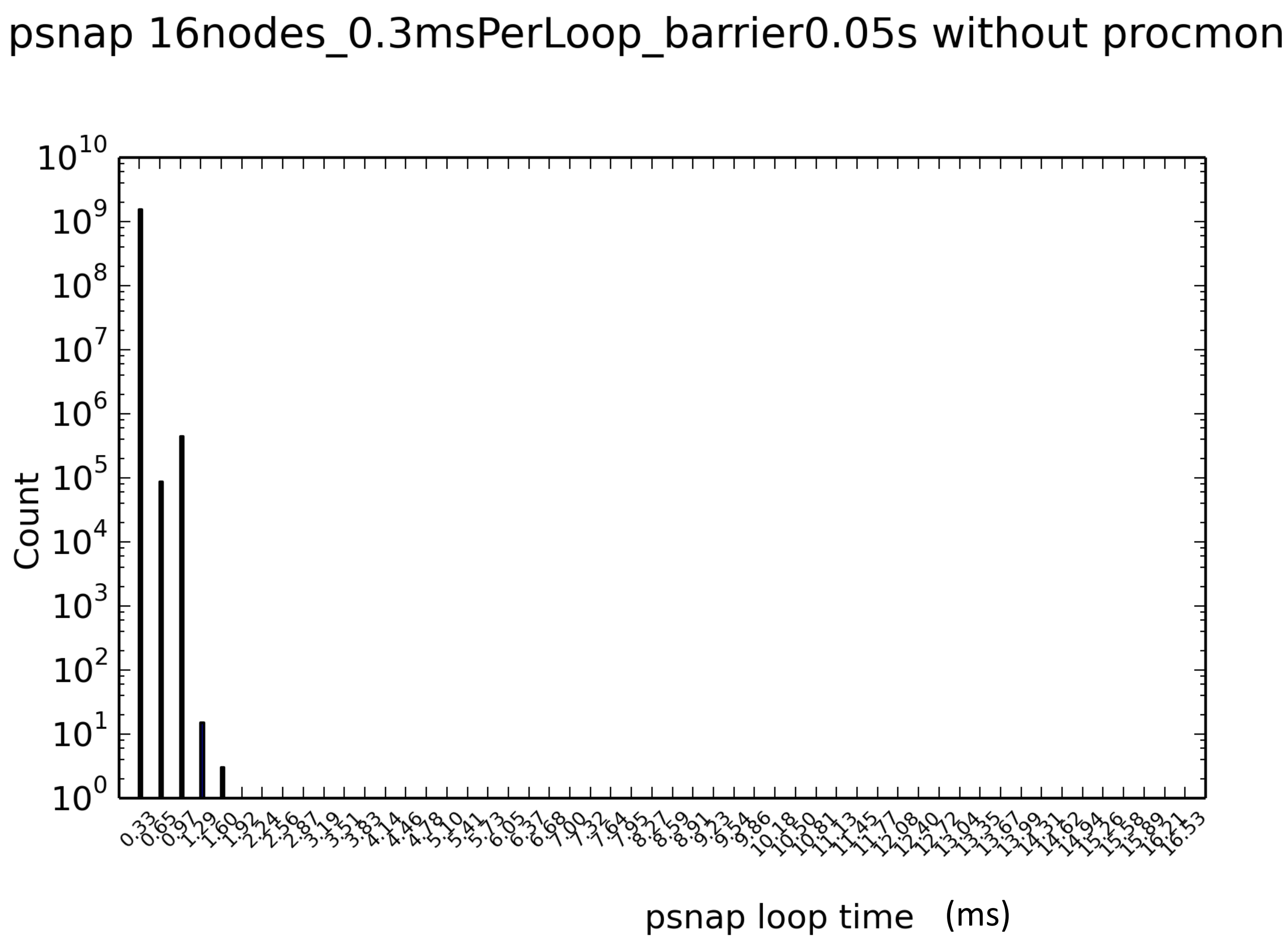
    module load procmon

    aprun **run_procmon** ./my_application <arguments>
  - Monitor CCM:
    - Start procmon in do_postmount() of /opt/cray/ccm/default/sbin/ccm_init_local
  - Monitor Everything:
    - Run persistently everywhere
    - Start/terminate with RUR plugin

U.S. DEPARTMENT OF ENERGY | Office of Science

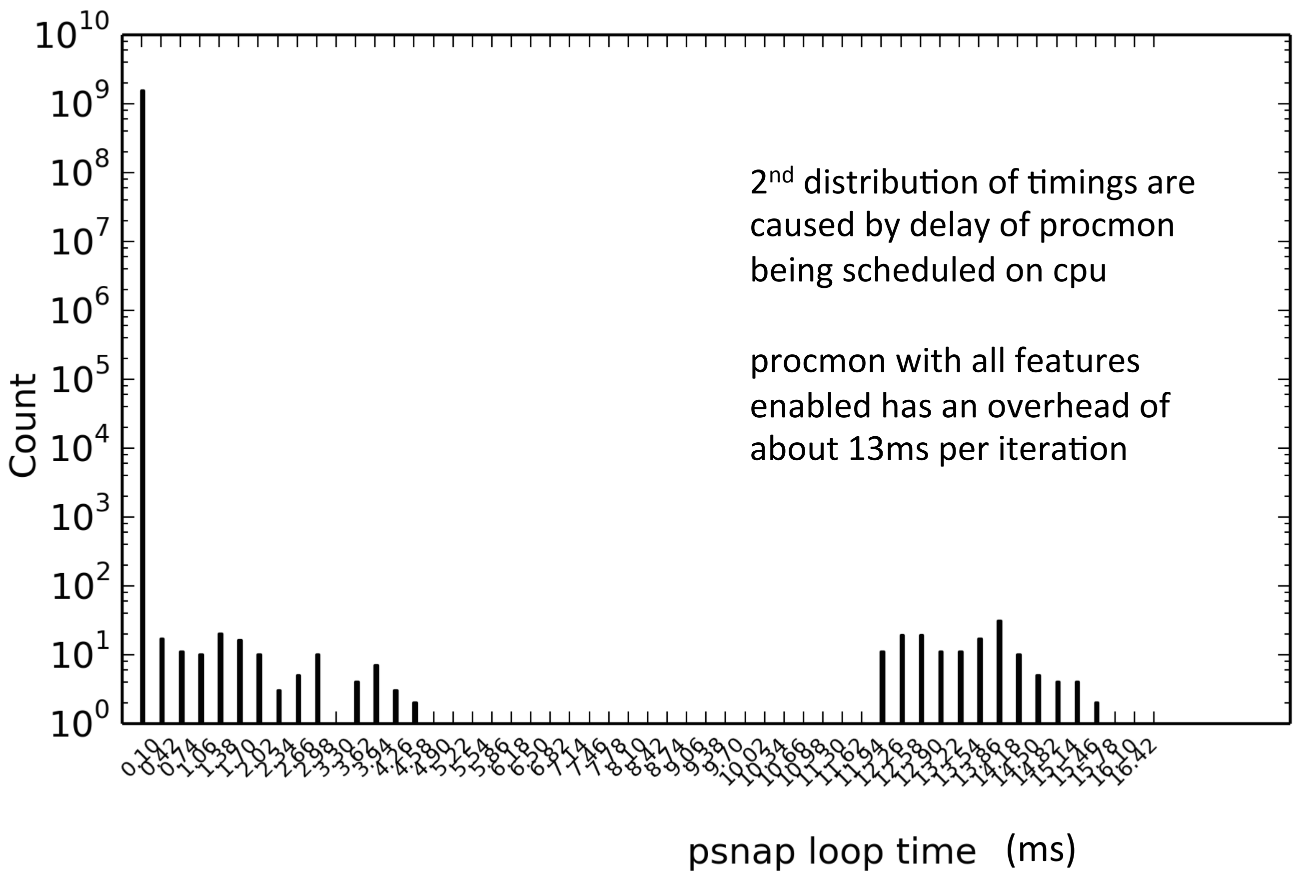
BERKELEY LAB

Performance tests on Edison



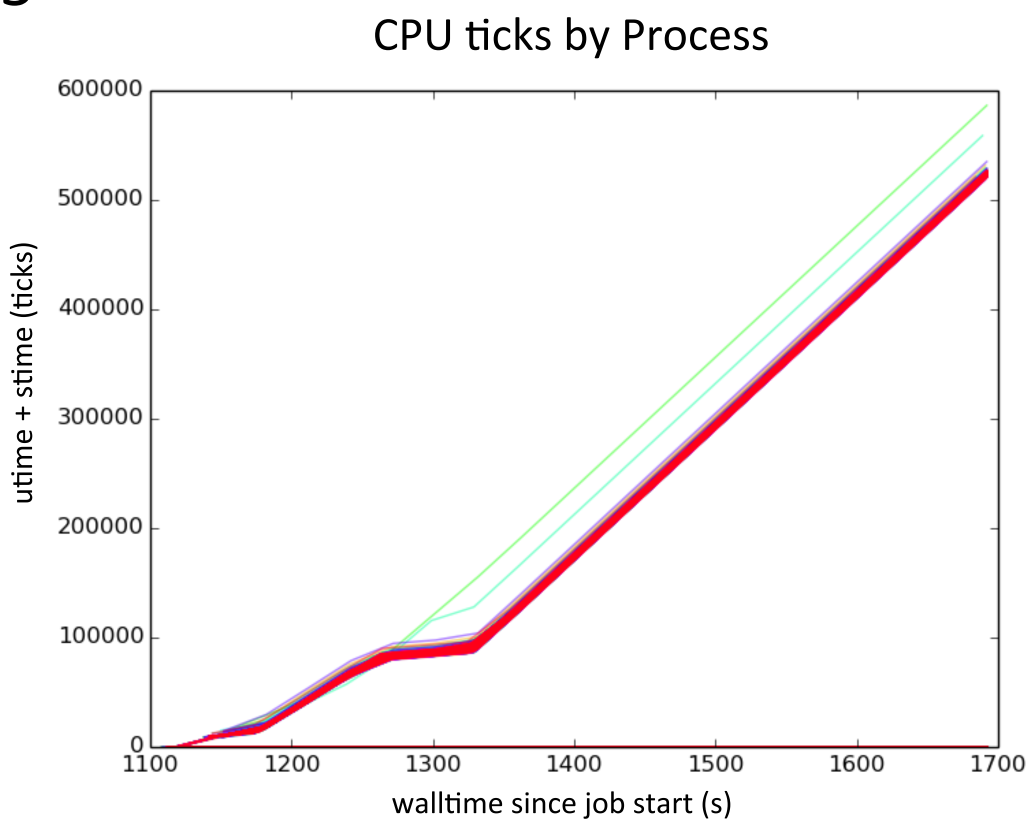
Performance tests on Edison

psnap 16nodes_0.1msPerLoop_barrier0.05s with procmon



Performance Results on Edison

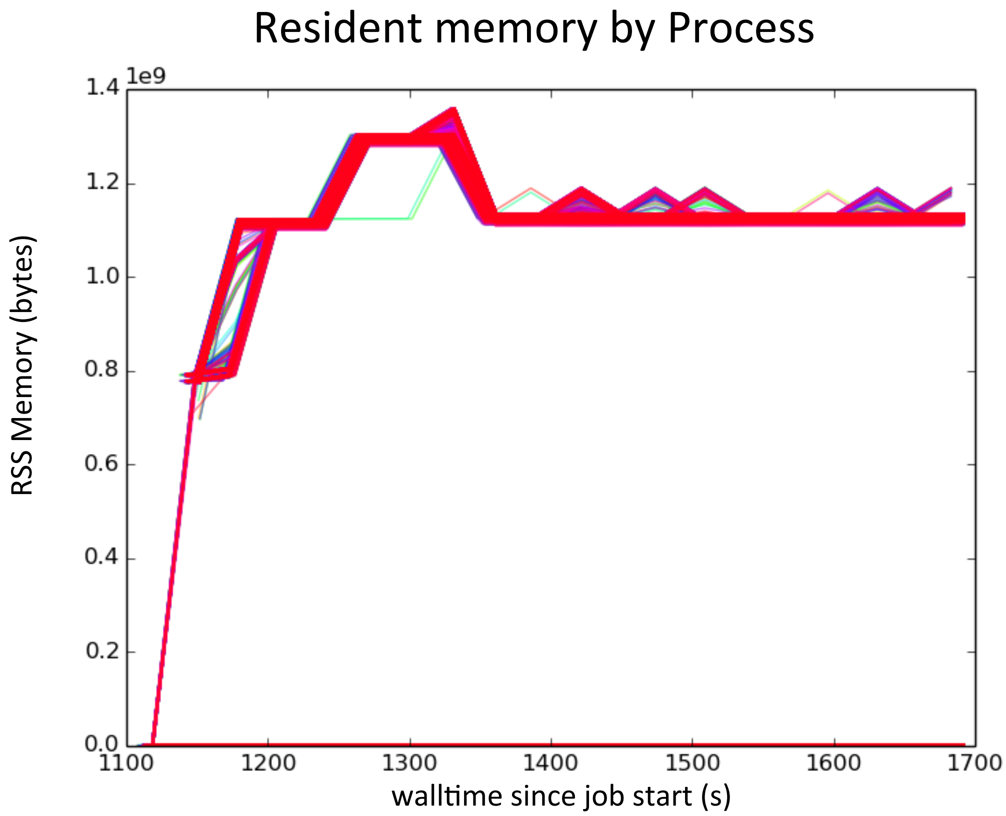
- Ran **non-production** HPCG run targeting 5 minutes with and without procmon monitoring the progress
- Ran on 130,800 cores – 5450 compute nodes
- Without procmon: **82893.2 Gflop/s**
- With procmon: **82131.5 Gflop/s**



Less than 1% observed difference on final score

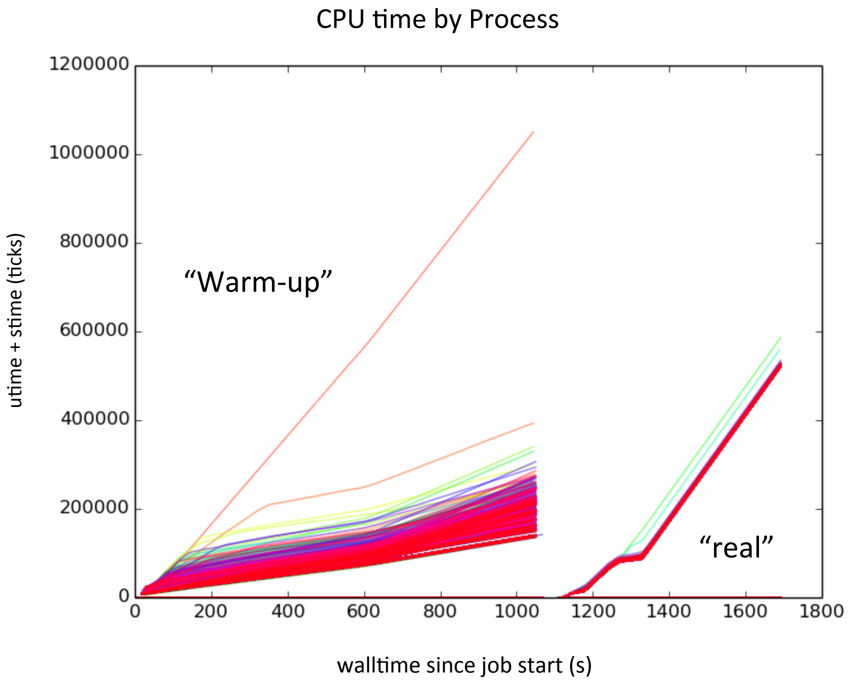
Performance Results on Edison

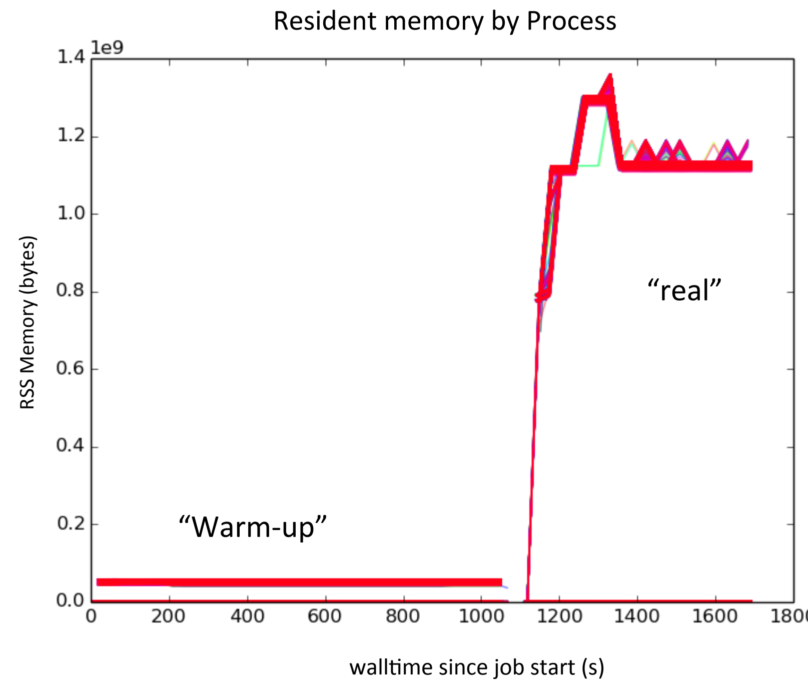
- Ran **non-production** HPCG run targeting 5 minutes with and without procmon monitoring the progress
- Ran on 130,800 cores – 5450 compute nodes
- Without procmon: **82893.2 Gflop/s**
- With procmon: **82131.5 Gflop/s**



Less than 1% observed difference on final score

Wait – what were those first 1100s?





- This hpcg was TBB-optimized version. Had noticed that 2nd run performs "better" than first.
- In this job, ran hpcg twice, first time with small matrix targeting 5s of walltime; second time with large matrix targeting 335s walltime
- Hypothesis: shared libraries for TBB, etc loaded asymmetrically across system

Future Directions

- **Completing transition to “pluggable” framework**
  - Developer creates new data structure representing new monitored data along with C++ template specializations for the monitoring, I/O, reduction, and query facilities.
  - Plugin monitoring can be executed on a per-host basis (e.g., /proc/meminfo) or per-process (e.g., /proc/<pid>/<something>)
- **Completing query-able cache of live data**
  - Allows any query to be run against recent observations without waiting for HDF5 file turnover or having to write particular filter to listen for RabbitMQ data

Conclusions

- procmon can **scale** to provide monitoring and minimal profiling services for running applications
- Built-in data management and data analysis software enables rapid deployment to usable monitoring data
  - Data can be directed where it is most useful to you!
- Can be deployed entirely in user-space – though there are some advantages to a system administrator assisted deployment
  - procmon.secured run as root can read privileged information from /proc

procmon is available and open source (BSD license):
https://bitbucket.org/berkeleylab/nersc-procmon

NERSC

Thank you.