procmon: Real-time process monitoring on the Cray XC-30

Douglas M. Jacobsen
Computational Systems Group
NERSC, Lawrence Berkeley National Laboratory
Berkeley, USA
Email: dmjacobsen@lbl.gov

***Abstract*—Increasingly complex workflows of data-intensive calculations are extremely challenging to characterize. In preparation for the increasing prevalence of this new style of workload, we describe a recent effort to implement the "procmon" system on the Cray XC30 system. The procmon system was developed to characterize data-intensive workflows of the mid-range clusters at NERSC, enabling efficient whole system monitoring of all running processes on the system with live real-time analysis of the data. procmon's resource-conscious implementation results in a scalable monitoring system that is minimally disruptive to both user and system processes, thereby providing useful monitoring opportunities on the large-scale Cray systems deployed at NERSC. Use of AMQP messaging enables flexible and fault-tolerant delivery of messages, while HDF5 storage of data allows efficient analysis using standardized tools. This approach results in an open monitoring system that provides users and operators detailed, real-time feedback about the state of the system.**

***Keywords*-monitoring, RabbitMQ, AMQP, HDF5, Data Analysis**

I. Introduction

In addition to classic HPC workloads, NERSC has engaged in providing data-intensive calculation platforms for several years, owing to the partnerships that NERSC maintains with PDSF and the Joint Genome Institute (JGI). Observing the JGI workload in particular, we found few means of easily discovering which codes comprised the workload since the codes rarely rely on MPI (cannot instrument mpirun or similar), the codes are not always directly compiled on the computational platform (some popular codes are statically linked binaries provided by the developer without source code) preventing systems like ALTD [1] from being informative, and in many cases, the batch scripts executed are perl or python scripts running a diverse and complex set of executables. To aid in system planning and resource prioritization, as well as more day-to-day tasks like job debugging, we needed to know the answer to a number of questions: which executables consume the most CPU time? Which executables block the most walltime? Which filesystems are used by which users and projects and which jobs have the most I/O to those filesystems? How does the batch workload differ from the interactive workload? Are the system resources being effectively used and managed?

Therefore, to answer these questions we constructed the procmon monitoring system to systematically and periodically sample all the running processes on a logical computational resource and centrally store those data. The data collected are typically similar to a combined output of a detailed "ps" and "lsof" command. This is distinct from traditional BSD-style process accounting in that the procmon data is sampled at an instant in time and is tagged with contextual information such as the batch job identifier, however, since the data are sampled many processes which run shorter than the sampling period may be missed entirely. Given the periodic nature of procmon sampling, a sufficiently long process is sampled numerous times providing a time-series trace of many different relevant process characteristics - a very simple job profile. We have used these data to see an at-a-glance view of a particular job, and have also summarized and aggregated the data to generate a system-wide distribution of the types, identities, and approximate resources consumed of all the processes on the system over a specified period of time - a detailed workload analysis.

It is critical that NERSC deliver as many system resources to the scientific applications as possible, thus there are a number of requirements to which a monitoring system needs to conform. The monitoring system should minimally impact a running job by ensuring that it uses very little CPU time, does not use any filesystems on the compute node, does not over-utilize network resources or engage in blocking network I/O. To get system information, the monitoring system should not fork processes if possible, or read data from interfaces other than the /proc, /sys or kernel interfaces that Linux provides (i.e., don't stat or read real files). Most importantly however, is that sampling of data is sufficient, on systems of large scale there is no need to capture every detail. The scale of NERSC systems require that monitoring systems be designed to handle large volumes of data, owing to the many thousands of nodes in a single logical computational resource which need to be simultaneously monitored. The large volumes of data typically lead to complications due to large bandwidth or filesystem I/O requirements. Therefore, it is also important that monitoring systems be scalable at every level, and that message producers must not be able to overwhelm network resources or services.

The author's philosophy regarding the capture and storage of monitoring data is that the rawest possible form should be aggregated and stored for later analysis. The ensures that data will be available to most completely answer the questions the

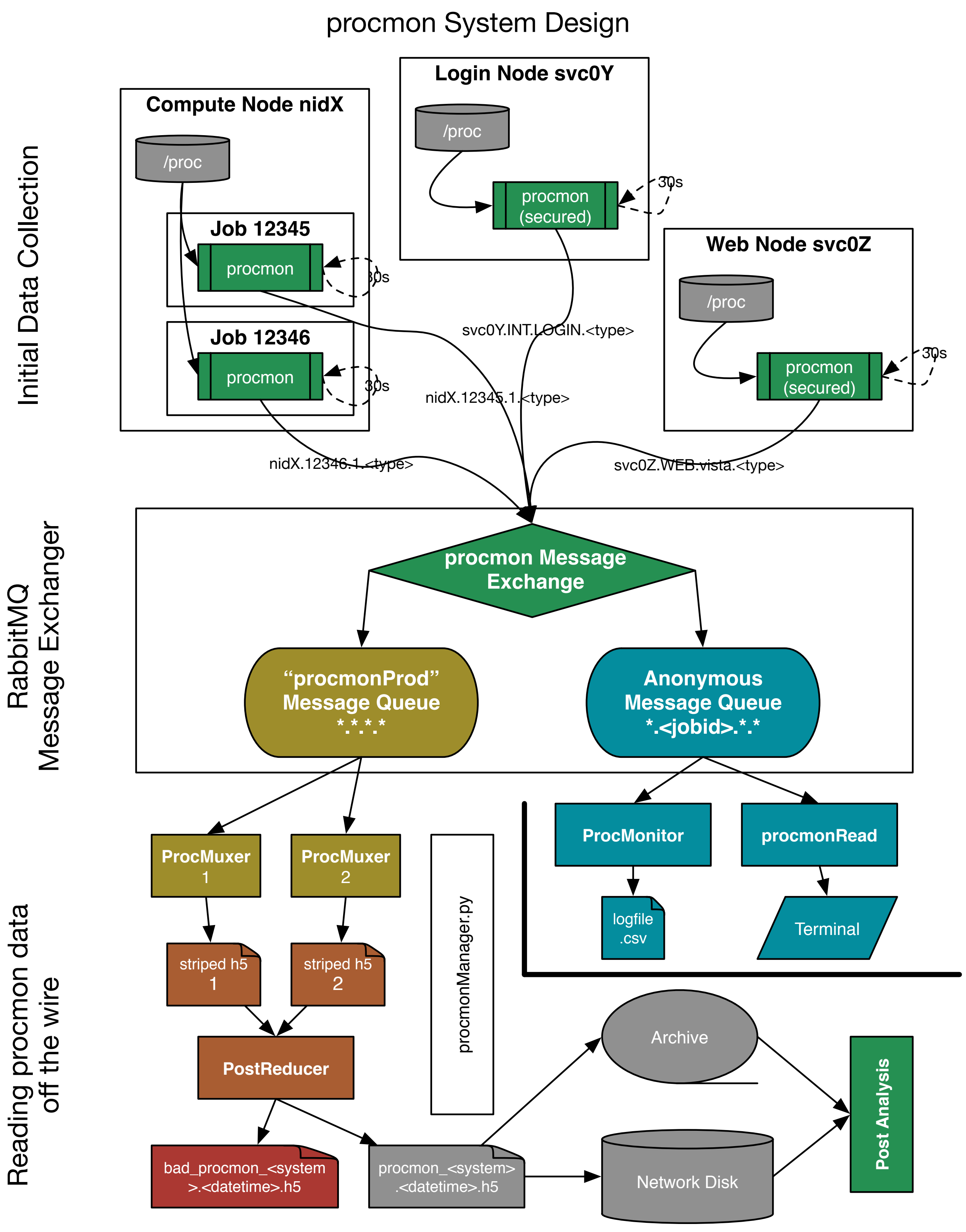

Fig. 1. Monitoring and Data flow through the procmon system.

monitoring system was designed for, but will also provide a rich dataset to serve future needs. Reducing and aggregating data too early typically leads to similar systems collecting similar data with a slightly different bias or perspective, due to changing needs.

Given the large volumes of data that a large-scale monitoring system must manage, it is also critically important that a data management strategy be delivered with the monitoring software. Typically, this may involve simple log rotation. However, because of the strategic nature of the data being collected and the large volumes of complex data being generated, a more managed approach, ideally integrated with an archival system is in order. The form in which the data are stored is also critical, data analysis of time series data is not immediately amenable to common relational databases, because while the data are highly structured, the time-series nature frequently obscures relationships in the data.

The procmon system has been designed to deliver the answers to complex questions, and to do so in a highly scalable, robust method conserving the highest resolution data possible using data formats that enable large-scale data analysis. Since these features are related to the needs of a general purpose monitoring system at the scale of a Cray XC system, we decided to evaluate procmon's performance on the larger scale system.

This paper discusses the architecture of the procmon system; how data are collected, managed, and analyzed; methods for integrating procmon into an XC30; performance analysis; and finally future directions and conclusions.

II. Architecture of Monitoring System and Scalability

The procmon monitoring system has a multi-level architecture that discretely separates monitoring, communication, aggregation, reduction, and data management capabilities. Figure 1 shows a high-level view of the architecture and how all the major components of the system tie together. The same data structures and a common I/O framework are used at every stage of the monitoring hierarchy - enabling the infrastructure to be wired and connected in a variety of ways to best match the needs of the site being monitored. Special attention has been paid to minimize any single points of failure, which maximizes availability of the service while simultaneously enabling tremendous scalability. The major components used in an HPC environment are: the monitoring agent ("procmon"), one or more RabbitMQ [2] servers for network communication, one or more data collectors ("ProcMuxers") to aggregate and write data to disk, a post-collection reduction tool ("PostReducer"), and the procmonManager script to orchestrate the data management activities. There are a number of auxiliary tools for data and workload analysis to be discussed elsewhere.

The monitoring agent, "procmon", actively acquires the data and pushes it to the I/O framework to serialize and publish the data. The I/O framework has multiple possible formats or media for transmitting or saving the data to disk. Under normal circumstances, the AMQP backend (enabled by embedding the rabbitmq-c library [3] into the application) for the I/O framework is used to transmit monitored data. The AMQP back-end transfers data to a centralized RabbitMQ service, however it is possible to provide a list of RabbitMQ servers to increase scalability and reliability of the system. If a list of RabbitMQ servers is provided, then one server is selected at random. Any failure in connection will cause another random re-selection of the server – this implements a very inexpensive form of High-Availability fail-over.

The monitoring agent performs minimal data transformation and no data interpretation or aggregation. It simply reads data and pushes it out across the wire. Our reasoning for doing it this way is several-fold: (1) the agent minimizes CPU and memory requirements if no state is saved between observations or comparisons performed, (2) the agent reports the data as closely as possible to the observable state of the system while minimizing biases if no on-system filtering or aggregation is performed, and (3) it simplifies the programming and improves maintainability and extensibility of the system by strictly defining the roles of the various subcomponents of the system. The separation of these responsibilities gives the administrator of the system maximum flexibility in the way in which the data are collected, reduced, and summarized ensuring that raw data is losslessly stored and archived to serve a variety of data needs.

A. *Process Monitoring*

procmon was originally developed to determine which processes were running as part of a contextual batch job. This does not include system processes (e.g., crond, syslog, etc.) which were not started as part of the job. Thus, procmon has a variety of ways of selecting which processes are part of the selected job. These are closely related to the way the batch system starts a job and determines which processes are also part of a job. The simplest mechanism is the hierarchical process tracking – e.g., monitor every thing that is an eventual child of some ancestor process. This is frequently a part of the filter used, track everything that is a child of the batch script, for example; or pid 1 to track all the user-space processes. Process sessions IDs can be used for tracking related processes – this is a common approach used for Torque. procmon also supports the use of a UNIX Group ID (gid) for tracking – this is not the process group id, but one on the groups list in the process; this functionality is useful for Sun Grid Engine. Linux cgroup tasklist support is the latest addition, is the most performant option, and should work with most modern batch systems if the cgroup functionality is enabled. The use of process sessions, group IDs, or cgroups (in order of least to most effective), enables procmon to find processes which have left the batch job hierarchy, for example by daemonizing.

Once the appropriate set of processes is selected, procmon then reads much of the data in /proc/pid/status, /proc/pid/stat, /proc/pid/statm, /proc/pid/io, and performs readlinks of cwd, exe, and a configurable number of file descriptors within /proc/pid/fd. This means that procmon is tracking most of the numeric and string data statistics for every identified process.

procmon datasets

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
```

procobs (never sent on wire)

identifier	char[IDENTIFIER_SIZE]
subidentifier	char[IDENTIFIER_SIZE]
recTime	unsigned long
recTimeUSec	unsigned long
startTime	unsigned long
startTimeUSec	unsigned long
pid	unsigned int

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

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
```

Fig. 2. Monitoring and Data flow through the procmon system.

See figure 2 for more information about the data structures of collected data.

B. Secured procmon for interactive/MOM nodes

Typically, procmon is run as a the user and initiated upon job or step start. However, on interactive and MOM nodes it can be advantageous to run procmon persistently (not as part of a batch job). This means that procmon will run all the time, and will be reading processes run by other users. Therefore it may be necessary to run procmon as root in order to read data that is otherwise kept private, for example performing the readlink for exe, cwd, and the file descriptors as a regular user is not possible for processes owned by other users.

procmon has been supplied with a "secured" mode, which allows it to run as root, or start with root privileges, retain the needed capabilities and switch to another less privileged user. Performing the needed readlink()s requires the CAP_SYS_PTRACE capability in Linux. Therefore the "secured" binary, will, upon initialization drop all other capabilities. Another facet of secured-mode is that procmon will actually become a multi-threaded executable wherein monitoring responsibilities are performed by one thread (which has the CAP_SYS_PTRACE capability), and I/O responsibilities are performed by another thread, which drops all capabilities. This ensures that the communication layer of the monitoring agent is less likely to be involved in providing an ability for someone to achieve heightened privileges on the system.

Another aspect of running procmon persistently is that it may need to collect data for multiple batch jobs – e.g., in the context of a MOM node in a Cray XE/XC system, or in the context of a multi-tenancy compute node like NERSC's edison serial queue. In those cases procmon will need to assign the job context for each identified process. This can be done in one of two ways by procmon, either by reading the process startup environment (/proc/pid/environ) and looking for telling environment variables, like $PBS_JOID, or by using the cgroup hierarchy and identifying job id based on the cgroup naming. The latter is much faster if it is available.

C. AMQP Network Communication

Once data has been collected by the procmon sensor it is serialized to an ASCII representation and sent via AMQP to a central (or one of several) RabbitMQ servers using the procmon I/O framework – which is itself supported by the rabbitmq-c communication library. The selection of AMQP over many other communication protocols was driven by (1) robustness of AMQP for handling exception scenarios – the administrator can make explicit choices about how to configure RabbitMQ to deal with clients producing messages at too-high rates, how to manage resources and users; (2) the robust message routing features of AMQP Topic Exchanges; (3) the automated queuing capabilities to allow temporary shortfalls in message consumption to be buffered; and (4) the general popularity and robustness of the RabbitMQ system combined with a desire to not write a communication framework from scratch.

The procmon monitoring system takes full advantage of the AMQP Topic Exchange concept. This is a form of message routing where each message includes a "Routing Key", a period-separated string, used by connecting clients (really the client queues) to subscribe to desired messages. In the procmon system, a four-tiered key is typically used: $hostname.Identifier.Subidentifier.dataType$. $hostname$ is typically the output of gethostname() on the monitored system. $Identifier$ is usually a job id, but could be any user-defined string that helps to provide context for the monitored processes. $Subidentifier$ is usually a task id, or job array task id, but could be any user-defined string that helps to provide context for the monitored processes. $dataType$ is the string identifier for the data structure represented by the message. The $dataType$ is used to help determine how to parse the message and handle it appropriately in the receiving application.

When a procmon sensor starts on host-to-be-monitored, it will automatically form a connection to the configured RabbitMQ server, and will attempt to create a "procmon" topic exchange (or other configured exchange). If the exchange already exists, then no error is thrown, as the goal is that all related procmon sensors will communicate on a single exchange in a given RabbitMQ server. In this system, message producers like the procmon sensor $never$ create queues to attach to an exchange. This is because there is a tight division of labor between the message producers and message consumers (clients), and a philosophy that as important as the data might be, the communication agent should not be used for more than ephemeral routing and queuing. If no clients are connected to listen for messages, then no queues will exist and the exchange will simply drop the messages. This ensures that the central RabbitMQ resource is not overwhelmed or required to start writing data to disk due to large numbers of queued messages caused by a failed client.

Any number of clients can connect to an exchange and form a queue. When a client creates a queue on a RabbitMQ topic exchange, it binds a routing key template, like "*.*.*.*" to get all messages, or something like "*.123456.*.*" to just get messages for job "123456". In fact, multiple clients can connect and create the same queue. In this case, the messages are striped across the two (or more) clients – though the striping is not perfectly load balanced, nor is it fully guaranteed to be unique – every once in a great while a message is duplicated. The RabbitMQ system itself enables a high degree of resiliency. The way procmon uses RabbitMQ allows multiple peered instances to be used separately, but it is possible to cluster RabbitMQ instances and automatically funnel messages between them. This functionality has not be used thus far by procmon, but represents an intriguing possibility.

RabbitMQ requires authentication to connect to the server. In procmon, this is typically done with one account with a pre-shared password, but there are options available for bifurcating access to secure the communications if desired. RabbitMQ can optionally use TLS/SSL to secure the socket connection. On trusted networks the TLS/SSL may not be necessary, but can be enabled if secure communications is a priority.

D. Data Collection and Data Management

Owing to the nature of the data being collected by the procmon system, it is important to collect as much of the data as possible for archival for post-analysis, as well as enable streaming/live access to selected data. The choice of RabbitMQ as discussed in section II-C supports multiple consumers for the same datastream, as well as aggregating data streams. In this way a generic connecting client can subscribe to messages from a single procmon sensor, a subset of them, or all of them. This allows all data from all the nodes of a particular job to be captured, or all the data in the system if that is preferable.

The lower segment of figure 1 shows how the data archival and other data consuming clients connect to the RabbitMQ server(s). The typical configuration is that one or more "Proc-Muxer" processes connect to to a single named queue attached to procmon's topic exchange. This causes the messages to be semi-striped across the two or more ProcMuxer instances. This enables the data collection backend to scale up regardless of underlying filesystem performance or message load. This striping capability also creates a nice facility for increasing the reliability of data acquisition by load balancing across clients. If one client dies, the other can take the full load. It also suggests a mechanism for migrating clients without data loss – e.g., start with two clients one one listening host, then start two clients up on the target host, finally terminate the original two and copy the data to the new host.

The ProcMuxer processes each write data to separate HDF5 files, the structure is shown in 3. At the beginning of each hour the previous HDF5 file is closed and a new one is created. In this way, the archival data are made available one hour at a time. Since the ProcMuxers are constantly writing data to disk, it is recommended, if possible, to run the ProcMuxers on a system with local disk instead of targetting a remote networked filesystem. This ensures that the ProcMuxers can adequately keep up with the data volume. Since frequently two or more HDF5 files are generated per hour (owing to the multiple ProcMuxer instances running), the PostReducer process is run to merge all the files, detect any bad data (e.g., truncated messages), and de-duplicate the data. Each datatype defines a reduction function that PostReducer can use to skip/remove repeated or redundant observations. Consider the case of an idling bash process or sshd process, most often these processes will not change any counter information, and so many observations will be identical except for the changing timestamps. Using the procstat or procdata reduction function, the PostReducer will detect these repeated observations and only keep the most recent record changed record. For every observation, regardless of whether or not any data changed, a ProcObs record will be inserted per process. This ensures that that an effective heart-beat is kept for all the observed processes, and yet the detailed data is only kept for unique

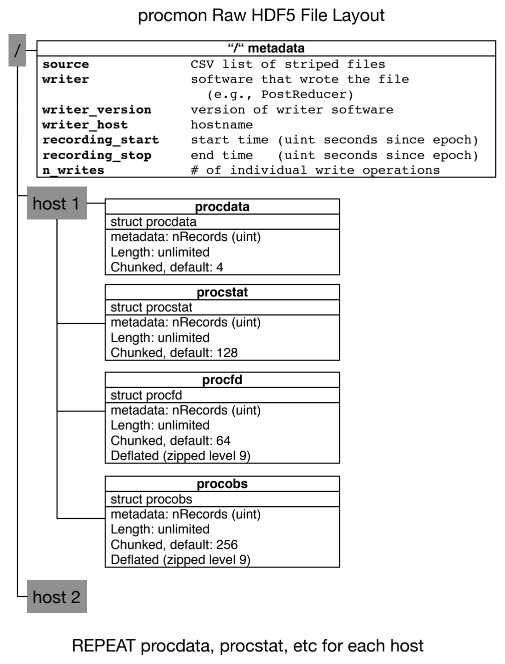
procmon Raw HDF5 File Layout
/
"/" metadata
source CSV list of striped files
writer software that wrote the file (e.g., PostReducer)
writer_version version of writer software
writer_host hostname
recording_start start time (uint seconds since epoch)
recording_stop end time (uint seconds since epoch)
n_writes # of individual write operations
host 1
procdata
struct procdata
metadata: nRecords (uint)
Length: unlimited
Chunked, default: 4
procstat
struct procstat
metadata: nRecords (uint)
Length: unlimited
Chunked, default: 128
procfd
struct procfd
metadata: nRecords (uint)
Length: unlimited
Chunked, default: 64
Deflated (zipped level 9)
procobs
struct procobs
metadata: nRecords (uint)
Length: unlimited
Chunked, default: 256
Deflated (zipped level 9)
host 2
REPEAT procdata, procstat, etc for each host

Fig. 3. Structure of HDF5 files used for data collection

observations. On the genepool system, this generates and approximately 18-fold reduction of storage requirements.

The procmonManager.py python script is responsible for managing the ProcMuxer processes, including restarting Proc-Muxers if they should die. procmonManager.py also detects once all of a particular set of HDF5 files are closed and initiates the PostReducer process as well as the data archival procedures. If a site has configured HPSS or JAMO archival solutions, then the procmonManager.py will automatically deposit the reduced HDF5 files into the archive as soon as they are created – in addition to copying the data to a configurable staging location.

procmon also enables live access to the data. This is typically used by special-purpose filters, such as the included "jobtop", which emulates a multi-node "top" to display all the performance data for a given job. Other than jobtop, these filters tend to be site- or analysis-specific.

III. Methods for System Integration

The value of process monitoring is entirely tied to the context or correctly assigning the context of the monitored processes. One of the central goals of procmon is to be able to trivially trace which process belonged to which job and be able to show the most pertinent data.

Thus, there are a few recommended minimal modes of running procmon on a Cray XC30 system. First the secured procmon can be started persistently on all login nodes, typically with an identifier of "LOGIN" or similar to help identify those processes as coming from a login node. On the MOM nodes, for schedulers making use of MOM nodes, enabling context discovery or "MOM-mode", will examine either the process environment or cgroup membership to identify which job or job-context each process belongs to. A default identifier or "MOM" should be provided to identify processes running outside of a job (i.e., system processes and direct-ssh-to-MOM-node-started processes).

Finally, if compute node monitoring is desired, there a few strategies that can be used:

- **Opt-in Monitoring Only**: Use the built-in run_procmon wrapper to automatically start procmon for select jobs
- **Monitor all CCM jobs**: Modify /opt/cray/ccm/default/sbin/ccm_init_local to start procmon
- **Monitor everything**: Either start procmon persistently in "MOM-mode" on all compute nodes to automatically determine job context, or, start and terminate procmon with included RUR staging plugin

IV. Performance Characteristics and Systemic Impact

It is critical to characterize the performance and resource requirements of any monitoring system. This is because there is always a prevailing concern that monitoring activity will disrupt applications or introduce a large amount of overhead. Based on our monitoring of process performance data (procmon included) for over one year on the genepool system at NERSC, we have found that the procmon sensor (the only component running on a compute node) typically uses less than 0.03% of a single core and about 2MB of memory throughout the course of a typical batch job. Network consumption is low, procmon typically transferring less than 3KB per process per iteration.

We performed a detailed study NERSC's Cray XC30 edison, to see how procmon behaved on the system. Using a similar protocol to that used to understand another real-time monitoring system, LDMS [4], we used PSNAP [**?**] to determine the CPU-disruption profile of the monitoring as well as a large-scale job to determine total effect on the system. Figure 4 shows the results of running the psnap benchmark with and without procmon. The psnap benchmark runs an inner loop a calculated number of iterations and time how long it takes in real time. This procedure is iterated many many times, and on a completely uninterrupted system, each iteration will perform the inner loop should complete in precisely the same time. The plots show histograms of how many iterations completed within each of a set of time windows. The primary feature is that when psnap is run with procmon we see that a few of the loop iterations (note the log scale of the Y axis), are right-shifted to about 13ms. Our interpretation of this is that when procmon is running, it remains scheduled on the CPU for up to 13 ticks (13ms) when all features are enabled. Since we typically do not see such a long scheduling time per iteration for procmon (data not shown), our conclusion is that this may be due to some of the more aggressive CPU scheduling

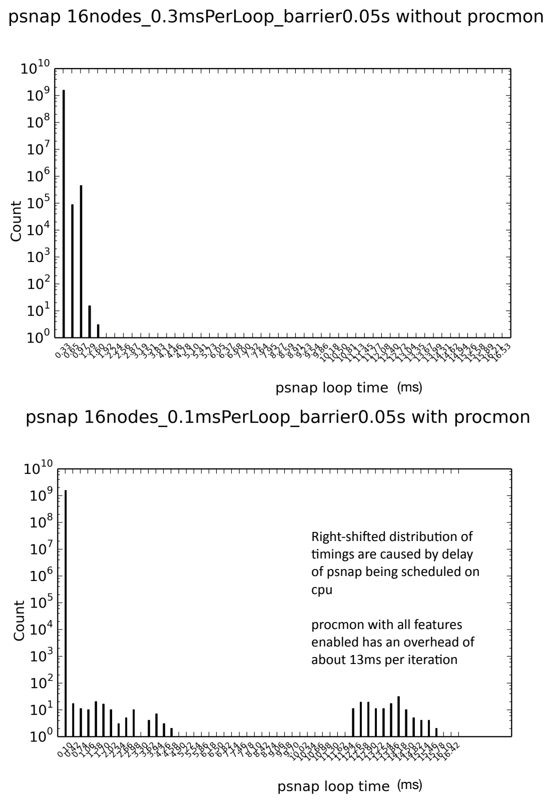

Fig. 4. 16-node psnap with and without procmon reveals 13ms average scheduling time for procmon per iteration

techniques employed in CLE. However, the most notable result is that on a Cray XC30 system, the "cost" of running procmon to compute jobs is approximately 13ms per iteration.

TABLE I
HPCG performance data
NOT PRODUCTION LENGTH RUN

	Measured GFLOP/s	Target Wall-time	Reported Execution Time	Date
HPCG only	82893.2	335 s	346.26 s	2015-03-06
HPCG & procmon	82131.5	335 s	349.518 s	2015-03-25

Another aspect of our investigation of procmon on edison was a nearly full-scale test using the Intel-optimized version of the HPCG [5] [6] benchmark. Table I shows the results of the HPCG test. The test was to run HPCG on 130,800 cores (5,450 nodes or 97.7% of edison) with two processes per node (10,900 processes) and 12 threads per process. We ran this test with and without procmon. Overall, the measured difference in performance was less than 1%. Since the without-procmon and with-procmon runs were on different days with slightly different sets of nodes, it is entirely possible that this minor effect is simply noise; without running the test several more times it is difficult to assess the statistical significance of the result. Empirically, we have found that running a "warmup" hpcg run can be very helpful in getting full performance out

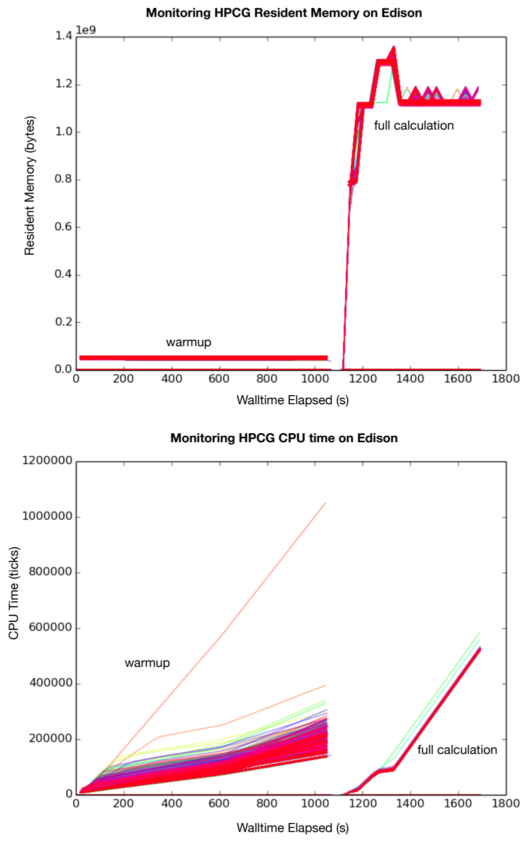

Fig. 5. procmon monitoring HPCG on 5450 nodes of NERSC's edison. top: Resident Set Size (rss) per process. bottom: CPU-time consumption per process. Left side shows a "warmup" calculation indended to cache all shared libraries. Right side shows a full, size standard HPCG calculation.

of the next run. This is likely because of the overhead in loading shared libraries, and that the second instance runs correctly because of caching of those libraries. Figure 5 shows the memory and cpu-consumption footprints for the "with-procmon" HPCG runs. Each plot has 10,900 lines for each of the "warmup" and "full calculation" runs. The "full calculation" results show most of the lines as a superposition of each other with little variation, as expected. The warmup, however, shows a wide spectrum of CPU consumption rates, which was a surprising result, given that HPCG does perform an MPI_Barrier prior to starting the at-large calculation. However, after examining the logs it was caused by a pathological set of inputs – an invalid configuration – for HPCG which caused inefficient communication resulting in the long execution time and strange band of CPU times (MPI_Allreduce() was very slow). The "warmup" run demonstrates some of the utility of using procmon to collect and then visualize job performance data collected at very large scale.

V. Future Directions

procmon remains in heavy development despite being operational at NERSC on two of the large clusters. The current planned directions include:

- introduce plugin architecture enabling simpler expansion of per-process or per-host monitoring capabilities
- expand support of cgroup job-context identification to enable tighter integration on the Cray platforms, and with SLURM
- complete work on query-able "live cache" of data, enabling immediate discovery of any recent data without an express-purpose client connected

VI. Conclusion

procmon is an extremely scalable monitoring platform, which based on the study presented herein, achieves the necessary scale and performance required to operate on the Cray XC30 platform. This platform can provide detailed accounting of a sample of all processes running on a large-scale computational platform to aid debugging, system visualization, and perform longitudinal workload analyses. procmon is an open source project which can be obtained at https://bitbucket.org/berkeleylab/nersc-procmon.

Acknowledgment

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.

The authors also wish to acknowledge the technical input and support from Shane Canon, Cary Whitney, Larry Pezzaglia, and Chris Daley from NERSC.

References

[1] M. Fahey, N. Jones, and H. Bilel, "The automatic library tracking database," *Proceedings of the Cray User Group*, 2010.

[2] "Rabbitmq," https://www.rabbitmq.com/.

[3] "rabbitmq-c c communication library," https://github.com/alanxz/rabbitmq-c.

[4] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications," *Supercomputing*, 2014.

[5] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," *Sandia Report*, 2013.

[6] "Intel optimized technology preview for high performance conjugate gradient benchmark," 2014, https://software.intel.com/en-us/articles/intel-optimized-technology-preview-for-high-performance-conjugate-gradient-benchmark.