Collecting Application-Level Job Completion Statistics

Matt Ezell, *National Institute for Computational Sciences*

ABSTRACT: *Job failures can be difficult to troubleshoot on large high performance computing systems. This is due to the massive quantity of log data and the difficulty in mapping log messages to correctable system problems. By collecting and analyzing information more effectively, system administrators can work to resolve the underlying issues and prevent future failures. This paper describes a set of tools to log and analyze applications in real-time as they run on the system.*

KEYWORDS: Job failure, aprun, alps, apwrap

1. Introduction

Large high performance computing systems have the capacity to run enormous quantities of jobs concurrently and generate massive amounts of log data each day. From a system administrator's point of view, this is too much raw data to analyze manually, and it can be difficult to get a clear picture of the hardware and software issues present on the system. On Cray XT systems, some error messages are printed only to the users' standard output and standard error, and are absent from the system logs. In addition, sometimes it is difficult to correlate the job ID with the ALPS ID number because the mapping is stored in a difficult-to-parse file present only on a single node. Not all of the data necessary to understand failures experienced on the system is easily accessible to the system administrators.

At the National Institute for Computational Sciences (NICS), a tool called *apwrap* was developed to provide a mechanism to quantify the types and frequency of errors experienced by users on our Cray XT systems.

2. Previous Work

Don Maxwell from Oak Ridge National Laboratory (ORNL) submitted a paper entitled "Restoring the CPA to CNL" [1] to CUG 2008. Maxwell developed several tools to scan various log files and put relevant events into a database. It looked at TORQUE, Moab, Alps, console logs, and syslog to create a clear picture of the system at any given time. It also included an aprun wrapper to capture the job exit codes. This approach is very thorough, but it is unable to log errors that do no appear in the system logs.

Also at CUG 2008, Nick Cardo from the National Energy Research Scientific Computing Center (NERSC) presented on "Detecting System Problems With Application Exit Codes" [2]. Cardo's method involved a TORQUE epilogue that scanned process accounting records as well as job standard output and standard error. One issue with this method is users are permitted to redirect standard output and standard error to an arbitrary location, making it impossible for the epilogue to examine the contents. Another possible issue with this approach surfaces when jobs output large volumes of standard output or standard error: the epilogue can take an extended time to run while processing all the data.

While both approaches are very useful, the author believes that more data can be collected and analyzed by using an alternative method to complement the existing solutions.

3. Design

The *apwrap* tool was designed to be a wrapper to the Cray-provided *aprun* binary. Users unknowingly execute *apwrap* when they desire to run an application. The *apwrap* tool executes Cray's *aprun* and processes the standard output and standard error. *apwrap* has several features not present in Cray's *aprun* that allow the system administrators to enforce policy and collect information.

The current version of *apwrap* was written in perl. An interpreted language was chosen to allow rapid development and to decrease the maintenance burden. Perl was an easy choice due to its powerful built-in regular expression engine and setuid program designed to allow processes to be run with different privileges.

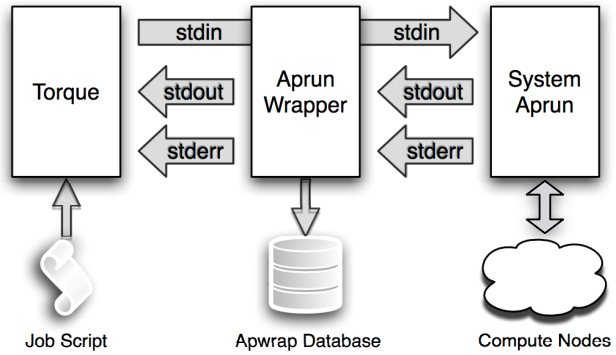

Figure 2: *apwrap* Integration Diagram

3.1 Design Goals

The following goals were considered when developing the *apwrap* tool:

1. Collect the error messages presented to the user and store them in a central database
2. Provide an unmodified user experience, unless alternative behavior is specifically desired
3. Fail gracefully

3.2 Prologues and Epilogues

With *apwrap*, arbitrary system-defined programs can be run before and after the real *aprun* to collect information and possibly prevent an application from being launched. These programs can be any executable code, including shell scripts, perl or python scripts, and even compiled binary code.

The programs should be placed in the prologues or epilogues directory, and they will be run in alphanumeric order. It is recommended to prefix the filename with a two-digit number if execution order is important. The programs will be executed with the same command line arguments provided to the original *aprun* command. All of the wrapper's internal variables are made available to the prologues and epilogues as environment variables.

If the program detects no errors and exits normally, it should exit with code zero. If there is a warning that should be passed to the user, the program should write a message to standard error and exit with code 1. If there is a fatal error that should prevent the application from being launched, the program should print a message to standard error and exit with code 2.

Any output printed to standard error will be directed to the user's standard error. Output to standard output should be in *key=value* form, as it will be interpreted as environment variables to set for further prologue and epilogue phases. These environment variables will not be propagated to the application.

At NICS, the prologue facility is used to integrate with the Application Library Tracking Database (ALTD) project to be presented at CUG 2010 [3]. Before an application is launched, the binary is scanned for information about the libraries it was linked with. This is stored in a database for later processing. The prologue facility could also be used to prevent known-troublesome application from being launched.

3.3 Standard Output and Standard Error Processing

The tool will "read" standard output and standard error from the application, looking for information that may be not be logged anywhere else. The rule set is defined as an array of hashes containing error descriptions and regular expressions. Typically, the tool can find errors such as failed nodes or segmentation faults, the exit code of the application, and the ALPS ID of the application. This information can be logged to a central server for processing later. The list of failure patterns is customizable by the administrator, so site-specific errors can be tracked.

```
rules   => [{
    name        =>      'NODEFAIL',
    pattern     =>      '^\[NID (\d+)\] \d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2} Apid \d+ killed. Received node
                        failed or halted event for nid \d+',
    message     =>      'A compute node had a hardware failure. Please resubmit your job.'
    },{
    name        =>      'SEGFAULT',
    pattern     =>      '^_pmii_daemon\(SIGCHLD\): PE \d+ exit signal Segmentation fault',
    message     =>      'A node experienced a segmentation fault. This happens when the code attempts to access a
                        memory location that it is not allowed to.'
    }]
```

Figure 1: Sample Rule Definitions

3.4 Instantaneous Feedback to Users when Errors Occur

Some of the error messages returned by the Cray binaries and daemons can be difficult to understand, especially for new users. This tool gives administrators the ability to add information to the job's standard output or standard error to explain what happened. It could provide advice to resubmit the job or give instructions to submit a trouble ticket for assistance.

3.5 Standard Error Archiving

The aprun wrapper keeps a configurable-length ring buffer of lines of standard error. In the case of a non-zero exit code, the wrapper can dump the error buffer to a file. This allows the administrator to go back and review the job errors. If necessary, new rules could be implemented to catch the specific error in the future. It is also possible to go back and edit existing entries to more appropriately reflect the problem experienced.

3.6 Supported Databases

The wrapper uses the perl DBI subsystem, so theoretically any database that has a DBD adapter should work. In practice, the SQL is slightly different for each server type, so only a small subset is actually supported. NICS runs the tool in production with PostgreSQL, but MySQL has been tested also. For single batch node testing, we have used SQLite. In theory, this should work across multiple nodes as long as the database were stored in an NFS target, but further testing should be done to determine if this is actually feasible for production.

4. Security

Security is always a paramount concern when developing new tools. The difficult issue present here concerns allowing user-level tools to insert information into a database while protecting the database credentials. It is undesirable to allow end users to insert arbitrary data into the database.

The easiest and most obvious solution involves obfuscating the password stored in a configuration file. Although this is enough to thwart the curious user from accessing the database directly, it is not really safe. If the aprun wrapper can "decode" the password, the end user could also.

Another possible solution involves converting the wrapper program into a compiled language so the obfuscated password and method of deobfuscating it are not clearly visible to the end user. This would thwart even more users from obtaining the password, but it is still just "security through obscurity." A determined end-user would still be able to determine the database credentials.

The security method chosen for this implementation requires the creation of a new, non-privileged user. That user is set as the owner of the database configuration file. Permissions on the database configuration file are locked down so that only the owner can read it. The wrapper script is then made setuid as that user. Perl's suidperl program launches the script as the non-privileged user, so it is able to read in the database credentials. Before the "real" aprun is executed, the wrapper script "drops privileges" so it is running as the end user. That way, the end user owns all files created by the application. This method effectively makes it impossible for the end user to access the database directly, and mitigates the risk in case a security problem exists in the wrapper.

5. Results

The *apwrap* tool has been running in production at the NICS on Athena, a 48 cabinet, 166 TF Cray XT4 system. Athena has been running in a semi-dedicated mode that has a very restricted user base. The results provided in this paper cover the one-month period from April 1, 2010 to April 30, 2010.

Each time the *aprun* binary was called, an entry was placed in the database. As the job progresses and more information becomes available, the database entry is updated to reflect the current state of the job. An example entry, taken from the database, is shown below (edited for clarity):

id	189
username	*user1*
system	athena
pbsserver	nid00004
batchid	68122.nid00004
batchidnum	68122
apid	1290954
batch_node	aprun3
pwd	/lustre/scratch/*user1*
arguments	-n 4096 -N 1 -d 4 *binary*
pes	4096
pes_per_node	1
depth	4
user_binary	/lustre/scratch/*user1*/*binary*
mpmd	f
pid	18367
start_time	1270358965
exit_time	1270366985
duration	8020
exit_code	1
error_name	NODEFAIL
error_string	[NID 15050] 2010-04-04 03:42:45 Apid 1290954 killed. Received node failed or halted event for nid 15051

Table 1: Summary of Results

Total Entries	37925
Entries per day	1264.17
Exited with code 0	32902
Exited non-zero	4991
Exit status was NULL	32
Unique Jobs	13874
Average apps per job	2.73
Max apps per job	86
Unique Users	12

Table 1 shows a summary of the applications run during the one-month reporting period. The most important statistic involves the number of successful versus unsuccessful application invocations. That breakdown is graphically shown in Figure 3.

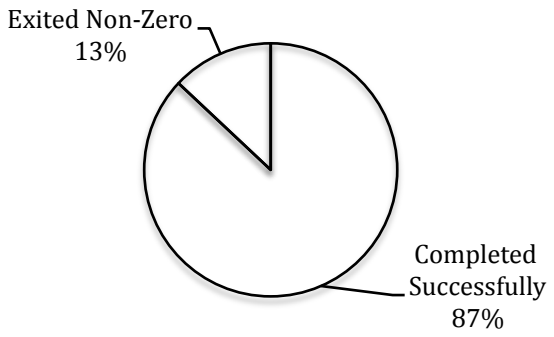

Figure 3: Application Completion Rate

Although the percentage of non-successful applications is slightly higher than one might like, further investigation shows the reasons for this number. Since these users were new to Athena, there was some startup time-cost associated with porting their code and scripts to this machine.

Table 2 shows a breakdown of the different error types along with their relative frequency. By far the most common failure reason was due to MPI_ABORT. In these jobs, the code chose to call the MPI_Abort() function. Many reasons exist for this, including memory allocation problems, numerical instability, or failed sanity checks. The next most common failure reason was because the job was killed. This could be attributed to either a user calling *qdel* on their own job or the batch system aborting it due to exceeded walltime limits. Applications also exit non-zero and print out the message "initiated application termination". It is not immediately clear why these programs abort. During the month of April 2010, Athena experienced four node failures that caused applications to abort. All the other errors appear to be caused by the user or their code.

Table 2: Failure Reason Breakdown

APRUN_ARGS	13
EXCEEDS_ALLOC	29
EXE_NOTFOUND	333
FLOAT_EXCEPTION	33
KILLED	803
MPI_ABORT	3050
NID_UNKNOWN	505
NODEFAIL	4
OOM	87
SEGFAULT	112

6. Future Work

The most important work in developing the *apwrap* tool involves examining the standard error dumps that correspond to unknown error messages and exit codes to develop new rules to catch these situations. In the near future this tool will be deployed on Kraken, the 88 cabinet petaflop Cray XT5.

Future versions of the Cray Linux Environment promise to make these user-level error messages available to the system administrators in the Cray Management Services (CMS) database. According to an ALPS developer:

> "Additional logging has been implemented to put more information into CMS. For instance, all ALPS warning and fatal messages received by aprun from an apshepherd on a compute node are now written to CMS. Aprun claim information including the exit codes and signals are now written to CMS. Batch reservation information is written by apsched to CMS. This type of information was only previously available within the batch job output files or through accounting records or ALPS logfiles." [4]

Hopefully these new features will allow future work to focus on building tools to analyze and report the errors instead of collecting them.

7. Conclusions

Large high performance computing systems are guaranteed to have occasional problems, and jobs will fail as a result of these problems. In order to perform root cause analysis on the types of errors experienced by users, tools must be developed that transparently observe the user experience. Armed with this data, system administrators can begin improving the user experience by addressing the problems. The *apwrap* tool has already proven useful in production by helping track down job problems.

References

[1] Maxwell, Don, et al. "Restoring the CPA to CNL", CUG 2008 Proceedings, Helsinki, May 2008.

[2] Cardo, Nicholas. "Detecting System Problems With Application Exit Codes", CUG 2008 Proceedings, Helsinki, May 2008.

[3] Fahey, Mark, Nicholas Jones, and Bilel Hadri. "The Automatic Library Tracking Database", CUG 2010 Proceedings, May 2010.

[4] Kohnke, Marlys. Comment #5, Cray BugZilla #752527. February 10, 2010.

Acknowledgments

The author would like to thank Don Maxwell and Nick Cardo for their excellent work in this area. The author would also like to thank his colleagues at the National Institute for Computational Sciences for their advice and assistance in developing this software.

About the Author

Matt Ezell is a HPC Systems Administrator at the National Institute for Computational Sciences at the University of Tennessee. He is the lead system administrator for the Athena project. He can be reached via e-mail at ezell@nics.utk.edu.