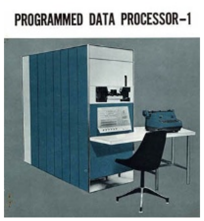
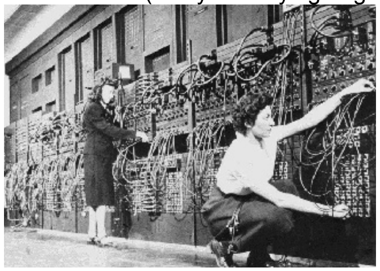
Job resiliency and application fault-tolerance

Created by Galen Arnold, last modified on May 28, 2019

Failure modes

- node failure
    - 

    - hardware
    - software (OOM killer , likely caused by application load balance)
    - *can lead to network failure in our Cray because "the network is the computer"*
- HSN network failure (really as varying degrees of degradation)
    - 
    - via node failure in your job or external to your job
    - *can lead to Lustre errors*
- Lustre outage (varying degrees of severity similar to network issues)
    - 
    - momentary (< 15 minutes)
    - via network degradation (seen as a slow filesystem)
    - disk failures causing degraded read performance during rebuilds
    - via storage network outage or storage server(s) down
    - metadata server Denial-of-Service (too many small i/o and directory operations, ex: find ...)
    - *can lead to system outage*
- Illinois

- 
- thunderstorms: NPCF-close lightning strikes
- issues with cooling water changeover Spring or Fall
- 
- *can lead to system outage*

For a system with > 22k nodes, Blue Waters is remarkably resilient. When most of the failures above occur (particularly single-node failures), they rarely result in a full system outage. The system can ride-through multi-node failures and even full-rack (cabinet) failures. Single-disk failures an almost daily event and mostly transparent to the user community. That said, it's good to plan for the worst while expecting the best.

Acceptance tests from early in the deployment

Test-286: Job Scheduler Job Fault Tolerance

Procedure:

**Still need to verify on JYC, hoping to update this soon.

Part 1 (9.9.3-2):
1. submit a 10 node job with 2 steps First step job: slow-hello, second step job: hello
2. kill one node (echo c > /proc/sysreq-trigger as root on compute node)
3. verify job is removed and nodes freed, hello has not been run

Part 2 (9.9.3-5):
1. submit a 10 node job with 2 steps, enable resiliency, First step job: slow-hello, second step job: hello
2. kill one node (echo c > /proc/sysreq-trigger as root on compute node)
3. verify both jobs are restarted and complete

Results:

BW:

Part1:

Submit 30 node:16ppn job without resiliency, killed 1 node

Job1 died and nodes checked and released.

Part2

Submit 30 node:16ppn job with 32p resiliency, killed 1 node

Job 1 relaunched using 1 less node and then job 2 ran fine with 1 less node.

Notes:

text here (if any)

Dashboard / ... / Test Procedures and Results

Edit Save for later Watch Share

Test-410 - Job resiliency to node failures

Created by Celso Mendes, last modified on Dec 01, 2012

Test 410: Job resiliency to node failures

Procedure:

Test #1: node failure through xtnmi

Using the xtnmi command, we injected a kernel-panic into a node using the out-of-band system. This makes a node unavailable. We used a Cray-provided MPI code to test this functionality. The test aims at verifying the scheduler does not stop the application even if one of the nodes the application is running on goes down.

Test #2: node failure through kernel panic

Using admin rights, we logged into a node and directly injected a kernel-panic. We used the second test provided by Cray, another MPI program that verifies the same feature.

Results:

Tests executed on Blue Waters on Nov.30/2012

Results from Test #1:

The MPI program detects the failure of a node and gracefully finishes execution. The PMI library provides the right interface to query the system for failed nodes. This test was successful, and the program output is shown in file test1_output.txt .

Results from Test #2:

This test was successful, and the program output is shown in file test2_output.txt .

Defenses and resiliency techniques for the community

Single-node failures

Cray-specific: aprun -R <pe_dec> ...
From "man aprun"

```
-R pe_dec Enables application relaunch so that should the application
          experience certain system failures, ALPS will attempt to
```

```
relaunch and complete in a degraded manner. pe_dec is the
processing element (PE) decrement tolerance. If pe_dec is
non-zero, aprun attempts to relaunch with a maximum of
pe_dec fewer PEs. If pe_dec is 0, aprun will attempt
relaunch with the same number of PEs specified with original
launch. Relaunch is supported per aprun instance. A
decrement count value greater than zero will fail for MPMD
launches with more than one element. Options -C and -R are
mutually exclusive.
```

Modify your job script to request more nodes than you need (-l nodes=n+1) and combine multiple aprun invocations with your checkpointing scheme.

Cray-specific: multi-aprun example

```
#!/bin/bash
#PBS -l nodes=1025:ppn=32:xe,walltime=08:00:00
#
# assert: job is designed to run through walltime (saves state via checkpoints)
...
aprun -N 1024 -n 16484 -d 2 ./a.out
# in the event a node fails, run again, a.out knows to read from available checkpoint via configuration/input file
aprun -N 1024 -n 16484 -d 2 ./a.out

# alternative, use a bash loop
#while true
#do
#  aprun -N 1024 -n 16484 -d 2 ./a.out
#done
```

To do : re-check XSEDE for the slurm NONSTOP plugin/config and make a note if found (no Xsede systems appear to support slurm NONSTOP – documentation not found)

Network degradation

Cray-specific: topology-aware-scheduling and balanced injection
https://bluewaters.ncsa.illinois.edu/topology-aware-scheduling

Do Nothing

If no special flags are specified, the topology-aware scheduler optimizes communication performance by placing the job into a convex cuboid node allocation.

Specify Application Communication Properties

qsub -l flags=commintolerant

The **commintolerant** flag prevents the scheduler from placing the job next to a large job. For jobs which are larger than half of any torus dimension, the shortest path between some nodes routes traffic outside the large job's allocation. If a job is extremely sensitive to interference from outside communication traffic, specifying the **commintolerant** flag causes the scheduler to schedule a placement not adjacent to any large jobs, eliminating such interference. Since the **commintolerant** flag limits potential placement locations, jobs may experience longer queue times when using this flag.

qsub -l flags=commtolerant

The **commtolerant** flag is now set by default. The job will still receive a convex cuboid node allocation, but the **commtolerant** flag does allow the scheduler to place the job adjacent to large jobs. If a job is sensitive to outside communication interference, adjacent large jobs may affect performance.

qsub -l flags=commlocal

The **commlocal** flag (also now set by default) allows my job to be placed next to others where my prism dimensions are over half the dimension span where it could route communication through others, but my task placement keeps most communication pairs within half the dimension away which will limit the interference imposed on others. (the shortest path between most pairs remains within my placement prism)

qsub -l flags=commlocal:commintolerant

Jobs are both local and intolerant.

https://bluewaters.ncsa.illinois.edu/balanced-injection

```
To "protect" the network from data loss Cray has enabled a method of throttling
to ensure that the packets on the HSN get to their destination. This is a
```

```
global throttling across the torus. To avoid this congestion protection
Cray has developed a concept called balanced injection. Balanced Injection
is a mechanism that attempts to reduce compute node injection bandwidth
in order to prevent global throttling and which may have the effect of
improving application performance for certain communication patterns.

A runtime environment variable called APRUN_BALANCED_INJECTION exists
that enables the user to set the balanced injection parameter for
all the nodes in the user's batch job. The range is from 0
(use the system default value) to 100 (no balanced injection)
and the relationship of the reduction in compute node injection
to APRUN_BALANCED_INJECTION is not linear. The environment variable
needs to be set before the aprun call in the batch script.

export APRUN_BALANCED_INJECTION=64
setenv APRUN_BALANCED_INJECTION 64
```

Lustre, I/O errors and long term I/O waits

For short duration events (up to a few minutes), most I/O calls will block or progress slowly and the only side effect will be the addition to walltime.

Fortran IOSTAT

http://wg5-fortran.org 2003
9.10.4 IOSTAT= specifier
Execution of an input/output statement containing the IOSTAT= specifier causes the scalar-int-variable in the IOSTAT= specifier to become defined with
(1) A zero value if neither an error condition, an end-of-file condition, nor an end-of-record condition occurs,
(2) A processor-dependent positive integer value if an error condition occurs,
(3) The processor-dependent negative integer value of the constant IOSTAT END (13.8.2.5) if
an end-of-file condition occurs and no error condition occurs, or
(4) The processor-dependent negative integer value of the constant IOSTAT EOR (13.8.2.6) if
an end-of-record condition occurs and no error condition or end-of-file condition occurs.

IOSTAT example

```

READ (FMT = "(E8.3)", UNIT = 3, IOSTAT = IOSS) X
IF (IOSS < 0) THEN
   ! Perform end-of-file processing on the file connected to unit 3.
   CALL END_PROCESSING
ELSE IF (IOSS > 0) THEN
   ! Perform error processing
   CALL ERROR_PROCESSING
```

```
END IF
```

C/C++ : check return values for posix i/o routines (fopen, f{read,write,puts,gets...}, fclose), use ferror()

from "man ferror"
Return Value
The ferror() function returns a nonzero value to indicate an error on the given stream. A return value of 0 means that no error has occurred.

This example puts data out to a stream, and then checks that a write error has not occurred.

ferror example

```c
#include <stdio.h>

int main(void)
{
   FILE *stream;
   char *string = "Important information";
   stream = fopen("mylib/myfile","w");

   fprintf(stream, "%s\n", string);
   if (ferror(stream))
   {
      printf("write error\n");
      clearerr(stream);
   }
   if (fclose(stream))
      perror("fclose error");
}
```

Illinois

Checkpoint your code.

- for timestepping codes, a configurable per N timesteps checkpoint setting
- look at the checkpointing interval calculator: https://bluewaters.ncsa.illinois.edu/storage

Checkpointing

All applications should implement some form of checkpointing that limits loss from hardware or software failures on the system. As the node count of a job increases or the wallclock increases, the likelihood of an interruption to the job increases proportionally.

To assist with determination of a proper checkpoint interval (the time between checkpoints that will provide a balance between loss of data due to a job interruption and the time spent performing checkpoint IO) we provide a utility that reports a recommended checkpoint interval using recent data on node failures and system interrupts, the desired number of XE nodes, XK nodes or both and the time the application takes to perform a checkpoint. The formula used in the utility is equation 37 from the 2004 paper by J.T. Daly "A higher order estimate of the optimum checkpoint interval for restart dumps". A mean time to interruption (MTTI) is computed and used to calculate a checkpoint interval (time between checkpoints).

Please remove commas when entering the requested node counts. Note that the time to write a checkpoint file is in hours.

NUMBER XE NODES	1024
NUMBER XK NODES	
TIME FOR A CHECKPOINT (IN HOURS)	0.10

Calculate

A checkpoint interval of 3.867 hours is recommended. (MTTI = 77.364 hours)

- One could set the interval dynamically with the C alarm() function and handler. The handler could set a global checkpoint flag indicating the need for a checkpoint at the next iteration.

- **https://www.gnu.org/software/libc/manual/html_node/Handler-Returns.html#Handler-Returns**

```c

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

/* This flag controls termination of the main loop. */
```

```c
volatile sig_atomic_t keep_going = 1;
int needcheckpoint = 0;

/* The signal handler just clears the flag and re-enables itself. */
void
catch_alarm (int sig)
{
  needcheckpoint = 1;
  keep_going = 0;
  signal (sig, catch_alarm);
}

void
do_stuff (void)
{
  puts ("Doing stuff while waiting for alarm....");
}

int
main (void)
{
  /* Establish a handler for SIGALRM signals. */
  signal (SIGALRM, catch_alarm);

  /* Set an alarm to go off in a little while: 2 seconds from now */
// if (myrank == 0) , alarm() could take a value from a configuration file or runtime parameter
  alarm (2);

  /* Check the flag once in a while to see when to quit. */
  while (keep_going)
    do_stuff ();

  return EXIT_SUCCESS;
}
```

Reference material:

man Intro_pmi

Test-410 - Job resiliency to node failures

Test-286 - Job Scheduler Job Fault Tolerance

http://man7.org/linux/man-pages/man3/ferror.3.html

https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_71/rtref/ferror.htm

https://wg5-fortran.org/N1601-N1650/N1601.pdf section 9.10.4 IOSTAT=specifier

https://bluewaters.ncsa.illinois.edu/storage checkpointing interval calculator

man aprun , see -C and -R options

https://slurm.schedmd.com/nonstop.html

https://bluewaters.ncsa.illinois.edu/balanced-injection

No labels