Capturing Computer Performance Variability in Production Jobs (and other topics)

Patrick Worley
Oak Ridge National Laboratory

ICL Friday Lunch Talk
March 22, 2013
University of Tennessee
Knoxville, TN
Acknowledgements

• This work was sponsored by the Climate and Environmental Sciences Division of the Office of Biological and Environmental Research and by the Office of Advanced Scientific Computing Research, both in the Office of Science, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

• This work used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725 and of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

• These slides have been authored by a contractor of the U.S. Government under contract No. DE-AC52-07NA27344. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

And thanks to Marcia Branstetter and Kate Evans at Oak Ridge National Laboratory and Janghoon Seo at the Korea Institute of Science and Technology for allowing me to collect data from their production runs.
1. These are “proof of principle” results.
   - People have been complaining about performance variability “forever”, with many reasonable hypotheses as to the causes. There has not been a systematic study of the issue on recent Cray systems in a production environment (where it matters) to my knowledge, and this is not one either.
   - Data are from two codes (CESM, XGC1). This work is not meant to be specific to these codes. They simply provide an infrastructure that is easily hijacked for these studies and users (and projects) willing to work with me.

2. Focus here is on *data collection for projects*, data that
   - can (and will) be collected by users for production jobs (so low overhead),
   - are sufficient for quantification of costs and impacts for project throughput,
   - are sufficient to guide more detailed studies or for motivating and evaluating mitigation strategies, and
   - are sufficient to get center staff involved in development of system level solutions
1. Significant differences in execution rate between similar jobs on the same platform when using the same resource requests (e.g. processor count) and in the same computing environment (system software versions, etc.)

   a. Due to the two jobs being run on processor subsets with different ‘topologies’, affecting communication performance?

   b. Due to different sets of concurrently running jobs competing for shared resources (interconnect bandwidth? I/O?), and in different ways (see (a))? 

   c. Due to one of the jobs being allocated a resource that is running suboptimally (‘slow compute node’)

   *Many possible sources, some not easily identified by user, so difficult to diagnose.*
Performance Variability, Aspects of

2. Significant differences in execution rate during the execution of a single job (not related to changes in the job’s execution characteristics)
   a. Due to changes in competition for shared resources as other jobs come and go during the job?

   *This is an aspect of (1), but can be more difficult to diagnose than when two jobs demonstrate static differences in execution rate.*

3. Significant differences in execution time or execution rate between similar jobs on the same platform when using the same resource requests (e.g. processor count) but over a period of timer during which things have changed: CESM version, compiler version, communication library version, etc.

   *Some change is expected, but do not want a degradation in performance to pass unnoticed. This may reflect a performance bug, and require regression to earlier versions of the code or of the software stack.*
Performance Variability, Implications of

1. Performance variability indicates that individual jobs are not performing optimally (as measured by best observed execution rate).

2. Jobs exceed requested time, and are aborted
   a. Some waste of allocation, depending on frequency of checkpoints
   b. Some waste of person time, as the failure is identified and required actions taken
   c. Failed jobs are resubmitted to queue, and suffer typical queue delay, slowing project productivity.

3. Increased checkpoint frequency, to decrease loss in failed jobs, consumes allocation (unproductively) in all jobs and is itself a performance variability hotspot (I/O)

4. Decreased job simulation time for a given wallclock request, to decrease failure rate due to performance variability, requires submitting more jobs to achieve same total simulation duration and thus spending more time in the queue waiting to be scheduled. This also slows project productivity.
Performance Variability, Implications of

5. Performance variability can mask application performance issues that should be eliminated, and can be mistaken for (fictitious) issues that consume software engineering time trying to correct.

6. Performance benchmarks are not reliable indicators of production run performance, and wallclock requirements:
   - Expensive to capture performance “envelope” (statistically significant sampling and retesting to capture code and system changes)
   - Expensive to use production-like benchmarks (typical runtime, variety of different code versions, variety of different configurations)

therefore

a. Users do not have dependable data for estimating wallclock requirements for individual jobs.

b. Project PIs do not have reliable estimates for project-wide allocation requirements.
Even when comparing between just two or three runs each for a sequence of processor core counts, the distributed solution of the linear system at the core of the Newton_Krylov solver for the ice sheet velocities exhibits significant performance variability for large processor counts, affecting total model performance.
43 jobs, each computing 400 simulation days. Data collected between May 15 and May 22, 2012

One exceeded 2 hour limit, sometime between simulation days 325 and 400.

Slowest successful job took 1 hour, 45 minutes; fastest took 1 hour, 25 minutes.

43 jobs, including 8 five-element ensembles (each submitted as a single job), each computing 280 simulation days. Data collected between May 23 and June 6, 2012

Two jobs exceeded 2 hour limit, both between simulation days 250 and 280.

Slowest successful ensemble took 1 hour, 40 minutes; fastest took 1 hour, 26 minutes.
35 jobs, each computing 150 simulation days. Data collected between May 15 and June 29, 2012.

Two exceeded 6 hour limit, one between simulation days 140 and 145 days, and one between 145 and 150.

Based on benchmark runs, expected to be able to complete 180 days in 6 hours.

Note that both I/O and non-I/O demonstrated performance variability.
Comparing performance between two jobs that exceeded the 6 hour wallclock limit and the fastest and slowest successful runs (completed in 4.5 and 5.75 hours, respectively). The failed experiments exhibit high internal performance variability. The successful runs have primarily different “base” (or static) performance levels. This was just happenstance - neither of these are necessary characteristics of “failed” or “successful” runs.
- Performance time series for all experiments, for the 5 fastest experiments, and for the 4 slowest (successful) experiments.
- Note very consistent performance for fastest experiments, with increased cost of “I/O” steps at specified simulation times and at end of experiment.
- Note performance variability in slow successful runs.
Aside: Gemini Interconnect Asymmetries

- 3D torus interconnect topology
- Two compute nodes per (X,Y,Z) coordinate, connected via a single Gemini switch
- Messages between nodes differing by one in either X, Y, or Z coordinates go through two Gemini switches
- For communication between nodes that are neighbors in the Y direction, performance differs depending on whether the smaller Y-coordinate is even ("faster") or is odd ("slower").
- For communication between nodes that are neighbors in the Z direction, every eighth link is "slower".
Process assignments for fastest successful job
- All processes assigned to “complete” node pairs
- 4 X-coordinates: 21, 22, 23, 24 (32 node pairs each)
- 2 Y-coordinates: 2, 3 (64 each, and no ‘slow’ Y links)
- 16 Z-coordinate: 8-23 (8 each)
So contiguous 4x2x16 allocation (no ‘holes’) with no “slow” Y links. Did include 1 “slow” Z link.
Example: CESM (T341f02.F1850r)

- Process assignment for slowest successful job
  - 272 processes assigned to 136 “incomplete” node pairs (53%)
  - 8 X-coordinates: 17-24
  - 10 Y-coordinates: 0-5, 8-9, 14-15
  - 24 Z-coordinates: 0-23

So widely scattered nodes, and perhaps including communication over a “slow” Y link.

Potential for interconnect contention from jobs running on other nodes in incomplete node pairs, and from other jobs sharing other interconnect links, based on noncompact node allocation (but nothing proven by these data).
Example: CESM (T341f02.F1850r)

- Process assignments for second fastest successful job
  - All processes assigned to “complete” node pairs
  - 4 X-coordinates: 17, 18, 19, 20 (34, 28, 34, 28 respectively)
  - 2 Y-coordinates: 8, 9 (64 each, and no ‘slow’ Y links)
  - 18 Z-coordinate: 4-21 (12 with 8 each, 5 with 6, 1 with 2)
    So contained within 4x2x18 “vertex cover” with 16 ‘holes’.

- Process assignments for third fastest successful job
  - 4 processes assigned to 2 “incomplete” node pairs (< 1%), both without active processes in other node at beginning, but a process running on each of these by the end
  - 4 X-coordinates: 21, 22, 23, 24 (32 node pairs each)
  - 4 Y-coordinates: 8, 9, 10, 11 (43, 44, 21, 21 respectively)
  - 18 Z-coordinate: 12-23, 0-5 (14 with 32 each, 2 with 16, 1 with 5)
    So contained within 4x4x18 allocation “vertex cover” with 288 ‘holes’.
Example: CESM (T341f02.F1850r)

- Process assignments for fourth fastest successful job
  - All processes assigned to “complete” node pairs
  - 5 X-coordinates: 4, 5, 6, 7, 8 (24, 24, 26, 28, 26 node pairs respectively)
  - 2 Y-coordinates: 14, 15 (65, 63 respectively, and no ‘slow’ Y links)
  - 14 Z-coordinate: 2-15 (11 with 10 each, 1 with 8, 2 with 10)
  
  So contained within 5x2x14 “vertex cover” with 12 ‘holes’.

- Process assignments for fifth fastest successful job
  - All processes assigned to “complete” node pairs
  - 5 X-coordinates: 4, 5, 6, 7, 8 (24, 24, 26, 28, 26 node pairs respectively)
  - 4 Y-coordinates: 10, 11, 12, 13 (14, 16, 49, 49 respectively)
  - 15 Z-coordinate: 20-23, 0-10 (10 with 10 each, 2 with 8, 2 with 5, 1 with 2)
  
  So contained within 5x4x15 allocation “vertex cover” with 172 ‘holes’.
Example: CESM (T341f02.F1850r)

- Process assignment for second slowest successful job
  - 40 processes assigned to 20 “incomplete” node pairs (8%)
  - 10 X-coordinates: 7-8, 17-24 (varying between 6 and 30 node pairs each)
  - 12 Y-coordinates: 0-3, 6-11, 14-15 (varying between 1 and 22 each)
  - 24 Z-coordinates: 21-23, 0-17 (varying between 2 and 16 each)
    So widely scattered nodes, with many holes in the vertex cover.

- Process assignment for fourth slowest successful job (third slowest had some missing data)
  - 12 processes assigned to 6 “incomplete” node pairs (2%)
  - 9 X-coordinates: 4-12 (varying between 3 and 22 node pairs each)
  - 8 Y-coordinates: 6-9, 12-15 (varying between 6 and 24 each)
  - 14 Z-coordinates: 16-23, 0-5 (varying between 1 and 18 each)
    So widely scattered nodes, with many holes in the vertex cover.
Time spent in queue for a T341 experiments - short queue time for many job submissions (25% under 1 minute), but over 40% were in the queue over 4 hours and 4 jobs were in the queue over a day (max. of 3 days). Same data plotted on left (linear-log) and on right (linear-linear).
Time spent in queue for a climate train submissions – short queue time for many job submissions, but still includes delays as long as 9 hours. Same data plotted on left (linear-log) and on right (linear-linear). In my experience, large job submissions can stay in queue for multiple days.
Current (Augmented) CESM Instrumentation

1. Data collected per user (by login name), per experiment case (user-defined), and per “job id” (timestamp for job start), in a Lustre project directory, so not swept.

2. Before jobs starts, collect
   - xml and namelist input (“provenance”)
   - output from xtdb2proc (or xtprocadmin on Edison), to capture what nodes are up and “topology”
   - output from ‘qstat –f’, to capture data about other jobs running or queued (used only to capture queue time currently)
   - output from showq, for another view of running and queued jobs (not using yet)
   - output from xtnodestat, to determine what other jobs are running and where
   - (sometimes) output from mdiag, for a different view of system configuration and where other jobs are running, including which nodes have GPUs
3. When job starts running, collect
   – process to node/core mapping by setting MPICH_CPUMASK_DISPLAY to 1 (goes to standard output).
   – MPI_COMM_WORLD to component model (subcommunicator) mapping (CESM also sends to standard output)
     Both are copied to the archive as soon as generated by a background job.
4. While job running, collect
   – output of user-specified profile data (GPTL timers) for selected processes and global statistics over all processes, periodically during execution. Frequency of output is a runtime parameter. Copied to archive periodically by background job.
   – output from showq and xtnodestat periodically during execution, collected using background job. Frequency of output is a (different) runtime parameter.
   – “tail” of model output files periodically during execution, collected using background job, to capture progress.
5. When job complete, collect
   – GPTL profile data for full execution (selected processes and global statistics)
   – standard CESM-specific summarization of profile data
   – some minimal end-to-end performance data (e.g., time spent in queue, time postprocessing, …)
   – Final output from showq, xtnodestat
Next CESM-related Steps

1. Further develop, and then automate, performance variability analysis, including identifying anomalies and trends and calculating true critical path and costs for project as a whole

2. Improve performance data archiving (currently working with TAU PerfDB developers).

3. Convince CESM software engineers that something like this capability is worth including in a release.

4. Develop user-level scheduling procedures that can mitigate performance variability:
   a. Run time evaluation and abort if potentially bad node allocation?
   b. Overallocation and use more efficient subset?

5. Discuss using less aggressive scheduling algorithms with NCCS and NERSC, or let users place restrictions on characteristics of allocations willing to accept?
1. Performance variability has been observed for many years, and is likely to continue to be an issue for CESM target platforms in the future.

2. Large and frequent performance variability is costly.

3. First steps are identification, quantification, and diagnosis
   a. Without this, cannot identify mitigation strategies, nor convince those who might be able to address the issues directly that there is in fact a problem worth addressing.
   b. Also need to capture full costs, including time spent compiling, time spent in queue waiting to be scheduled, time postprocessing, in order to understand the true impacts, and to understand what are the true performance bottlenecks in the project workflow.

4. Preliminary work on augmentation to existing performance data capture logic that can be used to document and diagnose performance variability is promising, but needs to be included in released CESM versions and used in more (all) production runs. (Will be adding similar logic to other application codes I have access to.)
Tracking XGC1 performance

One volunteer so far (Janghoon Seo, running XGC1 on Hopper)

1. Experiments on 19200 cores (800 nodes, 9600 MPI processes, 2 OpenMP threads per process)
   - 48 jobs completed within allocation time
   - 21 jobs ran out of time
   - 5 jobs failed during first checkpoint write
   - 2 jobs failed for other reasons

(Janghoon is an excellent experimentalist, with a disciplined use of case names, etc. Running out of time is not an issue for Janghoon. Checkpoint/restart is a natural solution for his development efforts, and much of the performance variability is due to changes in problem formulation from run to run. Running out of time only affects the ability to collect all of the performance data that we would like.)
Tracking performance

Other details of one of Janghoon Seo’s cases

2. 10 experiments: 1 failed during first checkpoint, other 9 completed within requested time
   a. Checkpoint write cost was no more 2% (for 18 checkpoint writes during 3 hour simulation)
   b. Variability between these simulations was small, between 2 hours 43 minutes and 2 hours 54 minutes.
   c. Variability within these simulations was small at a coarse level (13-15 minutes for each 300 timestep snapshot, 11 samples per simulation)
   d. Differences between fastest and slowest were all in communication routines in SHIFT, which includes any load imbalances. This can be simulation dependent, but appears to be due to I/O here.
Some details of one of Janghoon Seo’s simulations

2. Slowest run out of successful 9 simulations, process 0 view

<table>
<thead>
<tr>
<th>timer name</th>
<th># of calls</th>
<th>total time</th>
<th>max</th>
<th>min</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN_LOOP</td>
<td>3600</td>
<td>10410.9</td>
<td>19.52</td>
<td>2.50</td>
</tr>
<tr>
<td>PUSH_LOOP</td>
<td>7200</td>
<td>2750.6</td>
<td>0.48</td>
<td>0.29</td>
</tr>
<tr>
<td>DIAGNOSIS</td>
<td>7200</td>
<td>966.9</td>
<td>3.64</td>
<td>0.00</td>
</tr>
<tr>
<td>SHIFT_IET_RED</td>
<td>28049</td>
<td>384.8</td>
<td>2.85</td>
<td>0.00</td>
</tr>
<tr>
<td>SHIFT_IET_SR_RL</td>
<td>20849</td>
<td>827.2</td>
<td>0.18</td>
<td>0.00</td>
</tr>
<tr>
<td>CHARGEI_SRCHLOOP</td>
<td>7200</td>
<td>3681.2</td>
<td>0.52</td>
<td>0.51</td>
</tr>
<tr>
<td>POISSON</td>
<td>7200</td>
<td>406.0</td>
<td>0.59</td>
<td>0.04</td>
</tr>
<tr>
<td>GET_POT_GRAD</td>
<td>7200</td>
<td>326.2</td>
<td>0.05</td>
<td>0.04</td>
</tr>
<tr>
<td>RESTART_WRITE</td>
<td>6</td>
<td>67.0</td>
<td>13.72</td>
<td>8.89</td>
</tr>
</tbody>
</table>
3. Tracking performance

More details of one of Janghoon Seo’s simulations

<table>
<thead>
<tr>
<th>timer name</th>
<th>max (over threads)</th>
<th>min (over threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN_LOOP</td>
<td>10411.0 (2031 0)</td>
<td>10410.8 (8855 0)</td>
</tr>
<tr>
<td>PUSH_LOOP</td>
<td>2820.1 (9450 0)</td>
<td>2723.4 (289 1)</td>
</tr>
<tr>
<td>DIAGNOSIS</td>
<td>972.9 (7200 0)</td>
<td>2.9 (6691 0)</td>
</tr>
<tr>
<td>SHIFT_IET_RED</td>
<td>1977.6 (6298 0)</td>
<td>190.7 (1200 0)</td>
</tr>
<tr>
<td>SHIFT_IET_SR_RL</td>
<td>1022.8 (2338 0)</td>
<td>212.4 (7455 0)</td>
</tr>
<tr>
<td>CHARGEI_SRCHLOOP</td>
<td>3816.3 (3343 1)</td>
<td>3667.2 (6012 1)</td>
</tr>
<tr>
<td>POISSON</td>
<td>561.6 (2338 0)</td>
<td>340.8 (6646 0)</td>
</tr>
<tr>
<td>GET_POT_GRAD</td>
<td>339.2 (8117 0)</td>
<td>321.7 (237 0)</td>
</tr>
<tr>
<td>RESTART_WRITE</td>
<td>67.0 (0 0)</td>
<td>66.8 (2336 0)</td>
</tr>
</tbody>
</table>
Adapting to job running out of time

Issue: requested time for job is not sufficient, because …

Throughput rate decreases due to recent code changes,
Performance characteristics change, and throughput rate drops, due to nature of simulation, e.g. particle distribution,
Throughput rates drops because of allocation of nodes, including possible resource contention with neighboring jobs,
Throughput rate drops because of contention for, or other issues involving, the parallel file system,
This is the first time the user has used this system, run this problem, etc. and has no idea how much time to request, or
The user does not care, and will use checkpoints to resubmit after job aborts.

and the job aborts before writing out final performance data and wastes the CPU hours since the last checkpoint.
Adapting to job running out of time

Approach

a. Determine how much of the request time is remaining when application code calls MPI_INIT . (Implemented using background job and call to qstat.)

b. Estimate number of simulation steps to use half of remaining time, then guarantee that at least one checkpoint has been saved by then.

c. At every checkpoint request, estimate how much time is remaining.

d. If estimate indicates that time will expire before the requested number of simulation steps are completed, take a final checkpoint and finish early.
Adapting to job running out of time

Implementation

a. Modification to performance instrumentation routines to add `remaining time’ estimation logic, and a minor modification to timestep management logic to support additional checkpoints and early job termination.

b. Augmentation of new performance tracking scripts and background job to estimate remaining time when MPI_INIT is called.

is complete and working in my test cases. Next is to find friendly users and to implement something similar into the CESM.