Peta-scale Tools: A LLNL User Perspective (U)

Software Development Tools for Peta-scale Computing Workshop
Washington, DC

August 1-2, 2007

Brian Pudliner
Lawrence Livermore National Laboratory
P. O. Box 808
Livermore, CA 94551

Performance Tools

- Presently, most codes rely on internal timing systems built on top of TAU/PAPI for performance analysis.
  - Some analysis has been done with Vampir and MPIP, but we have not really pushed performance analysis tools to large scales.
- How scalable do the performance tools need to be?
  - Maybe not much of an issue, until we need to diagnose a real performance problem.
  - At scale, we are more likely to need to diagnose a focused performance problem that might be amenable to simplified code kernels or more directed tools.
  - In a mixed parallelism environment, we need to be able to measure parallel performance without perturbing the result.
  - Under what circumstances would we need to generate trace data at scale?

Performance Tools

- As the chip architecture and algorithms deviate from past experience, the need for detailed performance analysis tools will become acute.
    - We are unlikely to push these to scale

- Memory Characterization
    - With memory sizes being limited and/or shared, we'll need a way to find memory bottlenecks.
    - Is there a way to measure how efficiently we are using the memory available?

- Threading performance analysis
    - What exists out there for doing this?
    - How do we get at threading efficiency and overhead?
    - Is there a way to get a visual view of thread parallelism/performance? (Upshot for threads)

GAP

Performance Tools

- Load balance characterization and optimization
  - How efficiently am I allocating the work I have?
  - How do I diagnose the kernel responsible for load imbalance?
- We've gotten used to fat-tree style networks without topology limitations
  - If this changes, we'll need tools to help us characterize the performance of the network topology and help optimize mapping our problems onto them.
  - Can we leverage hooks in the existing MPI interface for topologies?
  - We will also need a way to make our partitioning tools topology aware.

Debugging Tools

- Today's perspective on debugging
  - Totalview handles most of our pedestrian debugging needs (< 1024 procs)
  - When Totalview falls over, a suitable application of grey matter, VisIt, and printf is the norm.
- Under what circumstances do we need scalable debugging?
  1. When memory limitations preclude shifting jobs to a more manageable number of processors for debugging.
  2. When the complexity of the parallelism or physics limits our ability to reproduce the problem.
  3. When an increasing fraction of our jobs are capability class jobs.
- All three of the above circumstances appear to apply to the peta-scale environment:
  - Even at the tera-scale we are starting to run jobs that cannot feasibly be scaled down in processor count due to memory size
  - Current proposals for peta-scale computing all involve novel architectures with mixed parallelism
  - The Capability Computing model is driving an increasing number and diversity of capability class calculations.

Debugging Tools

- Totalview doesn’t cut it at tera-scale, so we’re not expecting to see it at peta-scale
  - Can we get by with light-weight debuggers or debugging limited to a smaller number of processors?
  - Can we instrument debuggers into our code?
- Thread debugging
  - Ouch. Its gotta be there.
  - Will this be manageable with compiler-based threading?
- Python debugging at scale would be nice, but isn’t anywhere near the top of the priority list.
- Thread correctness
  - We’ve used Assure in the past, but we do not have much recent experience.
  - Given the direction things are going, this will probably be a priority.

Debugging Tools

- Memory fault and leak detection
  - We’ve used most everything in the medicine chest: Valgrind, Purify, Totalview (to a limited extent), ZeroFault
  - Some scalable memory checker will be imperative.
  - Can we improve performance by focusing the tool at specific sections of code, types of memory events, or by activating it from within the code?
  - We are willing to accept some false positives, but it would be nice if the tool could be taught what we considered to be important.

Development Environment Infrastructures

- Given the size of the other peta-scale alligators, an IDE is not a high priority
  - To date we haven't invested in determining its worth
  - But, if we're going to be dealing with a threading compiler, one could imagine integrating compiler feedback and performance into a source editor. Then again, isn't this just a working thread analysis tool?
- Please <insert deity>, limit the amount of cross compiling
  - Some codes have 30+ 3rd party libraries.
  - autoconf inline tests drive us nuts when cross compiling
  - If it must exist, is there a way to make it invisible?
- A way to build our codes in parallel will be necessary
  - Some codes use gmake –j others actually submit parallel jobs.

Development Environment Infrastructures

- Dynamically linked executables
  - We'd like to minimize what we need to do dynamically, but we will need the tools to be compatible with dynamic libraries

- Compilers / Loaders
  - Ought to support the C++ standard the day it's delivered.
  - A robust dynamic loader to handle large multi-language codes.

Suggestions

- At scale, stripped down, configurable tools will probably be most advantageous
  - I don’t need the machine shop, I just need a really good socket set. (preferably metric and English)
- Tools that I can compile into my code and can be guided to address focused problems/issues could possibly reduce overhead.
- We are going to rely more and more on interacting with our code while it is running for debugging and analysis, so this capability needs to exist.
- We’ve kind of let threads slip for the past 7-8 years and we have not been pushing on the tools to make them mature, so I suspect sizeable gaps will appear.

Suggestions

- For novel architectures (many cores, mixed parallelism, compiler threading), we are going to need early access tools that help us understand performance and correctness.

- In the past, long-term relationships between code/tool developers and the vendor(s) have proved profitable. With a noticeable change in technology, this will be even more important.

- Our major codes are unlikely to be radically re-written for initial architectures.
  - We need to learn some lessons before we start to dismantle a decade of hard work and a million lines of code.
  - Modest changes are more likely to be the norm for the big codes.
  - We are more likely to do our algorithmic and language exploration with simpler codes and code kernels.

Top Six

1. A means of debugging at scale
2. Memory debugging
3. Performance analysis tools: Serial, Parallel (at scale), Thread
4. Memory characterization tool
5. Thread correctness tool if necessary
6. A means of characterizing / optimizing for topology if necessary