Scalable Temporal Order Analysis for Large Scale Debugging

Presenter: Brandon Hill

Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, Martin Schulz

October 15, 2009
Introduction

- Problem
- Current Solutions
- New Solution
- Experimental Results
- Questions?
HPC programs continue to grow in complexity
  - Program size and solver complexity
  - Scale of parallelism (now on the order of 100K processes)

Debugging techniques aren't scaling
  - gdb requires sessions for each process
  - Totalview and DDT don't scale

Differential & statistical methods are too expensive
  - CBI & DIDUCE aren't easy to parallelize (and require too much inter-process relationships)
  - HPC resources charge per CPU/hour
  - Hundreds of large scale debug runs is too costly
Previous papers introduced STAT (Static Trace Analysis Tool)
- Gathers stack traces across nodes
- Groups similar processes into equivalence classes by execution progress
- Gives users a subset of processes to analyze
- Can be done during a paused run or as a post-mortem
STAT Problems

- Only works at the function name level
  - Too coarse grained - too much information lost
  - Obscures relationships between execution states
  - eg. `foo()` may call `bar()` in several places
  - eg. One task may be stuck in an early loop iteration but looks the same as other tasks
How to improve

- Finer-grained stack information (to the statement level)
- Show tasks’ relative progress through execution
- Show temporal relationships
All code compiled with debug info

Static analysis gathers needed info from AST and data flow analysis

Dynamic retrieval of needed information

Result is a function name, line number, and a task tuple for each task:

\[
\langle \text{par pos, iter, branch, local pos} \rangle
\]

Identify tasks’ stack positions in an call graph
Relative Progress

- Relative progress given by the function name, line number, and tuple
- \(\langle \text{par pos, iter, branch, local pos} \rangle\)
  - \text{par pos} relative position of the parent block
  - \text{iter} iteration count from current loop
  - \text{branch} branch number
  - \text{local position} relative position in local block
Merging

Results from global query of all tasks are then merged to group tasks into subsets that represent tasks all at the same state. The results can be displayed as a stack trace tree with annotations showing which branch represents the state of various tasks.

- Reduces the information the programmer needs to examine
- Gives a partial ordering of the tasks
  - Improves upon previous parallel partial orders since it is not based upon inter-process communication
Temporal Analysis

Statement location, iteration counts, and branch position are used to figure out temporal states within task subsets

- Iteration counting
  - They don’t instrument (considered too costly)
  - Use static analysis to find Loop Order Variables (LOVs)
  - LOV properties:
    - monotonically increasing (or decreasing)
    - consistent across tasks

- LOV discovery method considered one of the contributions of the paper

- Relative count of different iteration counts for similar tasks further refines the partial ordering

- To keep trees reasonable, LOV ordering is only applied as requested by programmer
Time costs:

- Standard cost of debug flags
- 256 task job took 10x longer to sample tasks (now 6 sec) than STAT
- Merge time is the same
Using fault injection several fault types were created in a standard MPI benchmark code (BT). They classified six major fault types. For 5 of 6 fault types, the offending task is the least progressed task. For the final fault type resulted in the faulty task being the most progressed task. In all cases the offending tasks were alone on the execution graph.

The tool was also used to successful analyze an actual code being developed that was hanging when scaled to 4,096 processors. Analysis took more than the simple outlier identification in the benchmark tests.
Questions?