

Parallel Debugging

Shirley Moore
shirley@cs.utk.edu

Historical Anecdote

According to Grace Murray Hopper, one of the computer science pioneers, here is the origin of the term **debugging**: In the early 1950s, the programmers at Harvard University spent weeks in an unsuccessful attempt to find the error in one of their programs. Finally, an investigation of the computer's insides revealed that an insect had died there, and its remains kept a relay from closing. Once this bug was removed, the program worked perfectly. Since then, the process of removing errors from programs has been known as "debugging".

Issues in Debugging Parallel Programs

- One problem with debugging concurrent programs is *nondeterminism*:
 - Two executions of a concurrent program may yield different results.
- Process/thread control – synchronous vs. asynchronous
- Batch queuing environments not set up for interactive debugging
- Scalability issues
 - Overhead of controlling large number of processes
 - Displaying states of large number of processes

Nondeterminism in Concurrent Programs

- A *race* occurs when two or more processes, either executing concurrently or randomly interleaved, attempt to access the same variable and at least one of the processes is storing into (writing) that variable. A race can be either a "read-write" race or a "write-write" race.
- A *deadlock* occurs when two or more processes are waiting for events (e.g., waiting on a semaphore or lock) such that none of the events will occur because each event depends on the prior completion of another such event.
- Although not all races and deadlocks can be detected by static flow analysis, the *potential* for a race can be detected statically and a warning generated.

Handling Nondeterminism

- Deterministic replay
- Automated detection of race conditions/memory conflicts
 - Simulation of possible concurrent executions
 - Generation of different interleavings at runtime
 - Using sleep
 - Randomized context switching

Deterministic Replay

- Example: A production parallel MPI program could have run correctly for six months but suddenly fail due to a message race.
 - If a trace from the failing run had been saved, it could be replayed to enforce the same sequence of events until the error is identified.
 - A trace scheme need only record received messages. However, on modern computers, message passing is implemented using direct memory access (DMA) hardware, so hardware assist is needed for tracing.
 - Overhead and intrusion issues
 - Implemented in Charm++ (<http://charm.cs.uiuc.edu/>)
- May never have deterministic replay for shared memory programs because the frequency of events that can race is very high.
 - Events based on page transfers may be traceable with hardware assist.

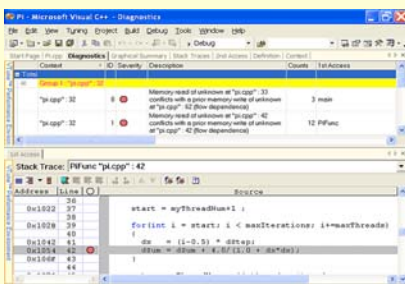
Simulation of Possible Concurrent Executions

- Simulating all possible concurrent executions greatly increases runtime and memory overhead.
- Approach used by the Assure and Intel Thread Checker tools is to offer different levels of checking for threaded programs.
 - <http://www.intel.com/software/products/threading/>
 - Assumption of correct serial execution
 - Validation that concurrent executions match serial on a given input set
 - Flagging of memory conflicts in context of source code

Memory Conflict Example

```
#include <stdio.h>
#include <omp.h>
int num_steps = 1000000;
int main()
{
    double x, pi, sum=0.0, step;
    step = 1./((double) num_steps);
    {
        #pragma omp parallel for
        for (int i=1; i < num_steps; i++)
        {
            x = (i - .5)*step;
            sum = sum + 4.0/(1. + x*x);
        }
    }
    pi = sum*step;
    return 0;
}
```

Intel Thread Checker Detection of Memory Conflicts



Memory Conflicts Resolved

```
#include <stdio.h>
#include <omp.h>
int num_steps = 1000000;
int main()
{
    double x, pi, sum=0.0, step;
    step = 1./((double) num_steps);
    #pragma omp parallel private(x) shared(sum)
    {
        #pragma omp parallel for reduction(+: sum);
        for (int i=1; i < num_steps; i++)
        {
            x = (i - .5)*step;
            sum = sum + 4.0/(1. + x*x);
        }
    }
    pi = sum*step;
    return 0;
}
```

MPI Deadlock Detection

- Deadlocks can occur with synchronous send (MPI_ssend), regular send (MPI_send) that does not use buffering, and synchronizing collection communications
 - Examples:
 - <http://www.lam-mpi.org/faq/category5.php3>
 - <http://www.mpi-forum.org/docs/mpi-11-html/node86.html>
- Deadlock detection tools
 - MPI-CHECK 2.0
 - <http://andrew.ait.iastate.edu/HPC/Papers/mpicheck2/mpicheck2.htm>
 - UMPIRE
 - http://www.sc2000.org/techpapr/papers/pap_pap208.pdf

Memory Debugging

- Memory leaks, heap allocation errors, etc.
- Valgrind for Linux systems
 - <http://www.valgrind.org/> (Linux/x86 only, works with LAM MPI and MPICH)
- TotalView memory debugging
 - <http://www.etnus.com/TotalView/Memory.html>
- TAU memory profiling
 - <http://www.cs.uoregon.edu/research/paracomp/tau/>

Parallel Debugger Features

- Management of process/thread groups
- Setting breakpoints, watchpoints, evaluation points
- Examining, modifying, and visualizing data
- Examining core files post-mortem
- MPI-specific features
- OpenMP and Pthread specific features

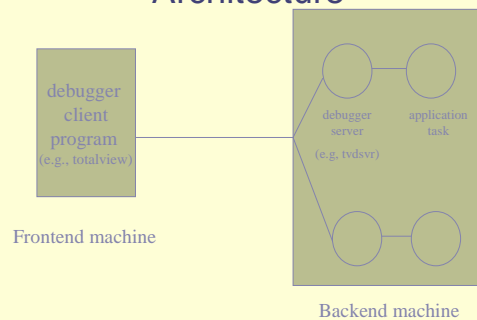
Typical Parallel Debugging Process

- Start up or attach to faulty program with parallel debugger and acquire processes and/or threads
 - Stop execution and set breakpoint before suspected error occurs
 - Step through program examining program state at each step
- OR
- Set data watchpoint(s) that stop execution when data location(s) is(are) changed
 - But watchpoints can be expensive if not supported by hardware

Process/Thread Control Issues

- Breakpoint semantics
 - Stop all processes/threads where they are when any reaches the breakpoint or when all reach the breakpoint? (Latter is called *barrier breakpoint* if processes are held until all reach the breakpoint).
- Step all processes/threads together or one at a time
 - Deadlock issues
 - Some operating systems do not support asynchronous thread control.

Parallel Debugger Architecture



Debugging Tools

- dbx
- gdb
- TotalView
- Distributed Debugging Tool (DDT)
- Guard relative debugger
- Vendor-specific tools
- printf?

dbx

- man dbx
- Sun dbx supports multithreaded programs
- SGI dbx supports multithreaded and multiprocess programs
- Intel Debugger supports dbx syntax

gdb and mpigdb

- GNU Debugger
 - <http://www.gnu.org/software/gdb/gdb.html>
 - Used as backend for some parallel debuggers (e.g., p2d2, mpigdb, Charm++)
- mpigdb
 - Distributed with MPICH
 - <http://www-unix.mcs.anl.gov/mpi/mpich/docs/userguide/node26.htm#Node29>
- Intel Debugger supports gdb syntax

TotalView

- <http://www.etnus.com/>

Distributed Debugging Tool

- <http://www.allinea.com/>

Relative Debugging

- Guard debugger
 - <http://www.csse.monash.edu.au/~davida/guard/>
- Detects errors by comparing the contents of data structures between program versions at runtime
- Assumes correct reference version
- User specifies assertions that specify locations at which data structures should be identical
- Moves iteratively through data flow to determine where program new version starts producing different answers
- Supports versions written in different languages and running on different machines in a heterogeneous network

Vendor-specific Tools

- Intel Debugger (IDB)
 - http://www.intel.com/software/products/compilers/techtopics/intel_debugger.pdf
- Portland Group PGDBG
 - <http://www.pgroup.com/products/pgdbg.htm>
- SGI ProDev Workshop Debugger
 - <http://www.sgi.com/products/software/irix/tools/prodev.html>
- IBM AIX pdbx
 - man pdbx

printf?

- <http://www-unix.mcs.anl.gov/mpi/mpich/docs/userguide/node26.htm#Node27>
- Parallel Print Function
 - <http://www.llnl.gov/CASC/ppf/>