Parallel Debugging Techniques

Le Yan
Louisiana Optical Network Initiative
Outline

• Overview of parallel debugging
  – Challenges
  – Tools
  – Strategies

• Get familiar with TotalView/DDT through hands-on exercises
Bugs in Parallel Programming

- Parallel programs are prone to the usual bugs found in sequential programs
  - Improper pointer usage
  - Stepping over array bounds
  - Infinite loops
  - ...

- Plus...
Common Types of Bugs in Parallel Programming

• Erroneous use of language features
  – Mismatched parameters, missing mandatory calls etc.

• Defective space decomposition

• Incorrect/improper synchronization

• Hidden serialization

• ......
Debugging Essentials

• Reproducibility
  – Find the scenario where the error is reproducible

• Reduction
  – Reduce the problem to its essence

• Deduction
  – Form hypotheses on what the problem might be

• Experimentation
  – Filter out invalid hypotheses

Terrence Parr, Learn The Essentials of Debugging
Challenges in Parallel Debugging

• Reproducibility
  – Many problems cannot be easily reproduced

• Reduction
  – Smallest scale might still be too large and complex to handle

• Deduction
  – Need to consider concurrent and interdependent program instances

• Experimentation
  – Cyclic debugging might be very expensive
A Nasty Little Bug

• What is the potential problem?

... integer*4 :: i,ista,iend
integer*4 :: chunksize=1024*1024
...
call MPI_Comm_Rank(MPI_COMM_WORLD, &
    myrank,error)
...
ista=myrank*chunksize+1
iend=(myrank+1)*chunksize
do i = ista,iend
    ...
enddo
...

8/3/2009 Scaling to Petascale Virtual Summer School
A Nasty Little Bug

- A bug that shows up only when running with more than 4096 cores

```fortran
... integer*4 :: i,ista,iend
integer*4 :: chunksize=1024*1024
...
call MPI_Comm_Rank(MPI_COMM_WORLD, &
myrank,error)
...
ista=myrank*chunksize+1
iend=(myrank+1)*chunksize
do i = ista,iend

... Integer overflow if
myrank ≥ 4096 !
... enddo
...```

8/3/2009 Scaling to Petascale Virtual Summer School
printf/write Debugging

• Extremely easy to use, therefore dangerously attractive, but...
  – Need to edit, recompile and rerun when additional information is desired
  – May change program behavior
  – Only capable of displaying a subset of the program’s state
  – Output size grows rapidly with increasing core count and harder to comprehend

• Not scalable, not recommended
Compilers Can Help

• Most compilers can (at runtime)
  – Check array bounds
  – Trap floating operation errors
  – Provide traceback information

• Relatively scalable, but...
  – Overhead added
  – Limited capability
  – Non-interactive
Parallel Debuggers

• Capable of what serials debuggers can do
  – Control program execution
  – Set action points
  – View/change values of variables

• More importantly
  – Control program execution at various levels
    • Group/process/thread
  – View MPI message queues
An Ideal Parallel Debugger

• Should allow easy process/thread control and navigation
• Should support multiple high performance computing platforms
• Should not limit the number of processes being debugged and should allow it to vary at runtime
How Parallel Debuggers Work

• Frontend
  – GUI
  – Debugger engine

• Debugger Agents
  – Control application processes
  – Send data back to the debugger engine to analyze

User processes

Interactive node

Compute nodes

Debugger engine

GUI

Agent

Agent

Agent
At Very Large Scale

• The debugger itself becomes a large parallel application

• Bottlenecks
  – Debugger framework startup cost
  – Communication between frontend and agents
  – Access to shared resources, e.g. file system
Validation Is Crucial

- Have a solid validation procedure to check the correctness
- Test smaller components before putting them together
General Parallel Debugging Strategy

• Incremental debugging
  – Downsbyte if possible
    • Participating processes, problem size and/or number of iterations
    • Example: run with one single thread to detect scope errors in OpenMP programs
  – Add more instances to reveal other issues
    • Example: run MPI programs on more than one node to detect problems introduced by network delays
Strategy at Large Scale

• Again, downscale if possible
• Reduce the number of processes to which the debugger is attached
  – Reduces overhead
  – Reduces the required number of license seats as well
• Focus on one or a small number of processes/threads
  – Analyze call path and message queues to find problematic processes
  – Control the execution of as few processes/threads as possible while keeping others running
    • Provides the context where the error occurs
Trends in Debugging Technology

• Lightweight trace analysis tools
  – Help to identify processes/threads that have similar behavior and reduce the search space
  – Complementary to full feature debuggers
  – Example: Stack Trace Analysis Tool (STAT)

• Replay/Reverse execution
  – ReplayEngine now available from TotalView
  – Checkpointing supported in DDT 2.4

• Post-mortem statistical analysis
  – Detect anomalies by analyzing profile dissimilarity of multiple runs
Hands-on Exercise

• Debug MPI and OpenMP programs that solve a simple problem to get familiar with
  – Basic functionalities of parallel debuggers
    • TotalView: Pople, Bluefire and Athena
    • DDT: Ranger
  – Some common types of bugs in parallel programming

• Programs and instructions can be found at http://www.cct.lsu.edu/~lyan1/summerschool09
Problem

A 1-D periodic array with \( N \) elements

Initial value
- C: \( \text{cell}(x)=x \mod 10 \)
- Fortran: \( \text{cell}(x)=\text{mod}(x-1,10) \)

In each iteration, all elements are updated with the value of two adjacent elements:
- \( \text{cell}(x)_{i+1}=[\text{cell}(x-1)_i+\text{cell}(x+1)_i]\mod 10 \)

Execute \( N_{\text{iter}} \) iterations

The final outputs are the global maximum and average
Sequential Program

- Use an integer array to hold current values
- Use another integer array to hold the calculated values
- Swap the pointers at the end of each iteration
- The result is used to check the correctness of the parallel programs
  - Chances are that we will not have such a luxury for large jobs
MPI Program

- Divide the array among $n$ processes
- Each process works on its local array
- Exchange boundary data with neighbor processes at the end of each iteration
- Ring topology
OpenMP Program

- Each thread works on its own part of the global array
- All threads have access to the entire array, so no data exchange is necessary
Three Ways to Start TotalView/DDT

- Start with core dumps
- Start by attaching to one or more running processes
- Start the executable within TotalView/DDT
### Status Code Description

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank</td>
<td>Exited</td>
</tr>
<tr>
<td>B</td>
<td>At breakpoint</td>
</tr>
<tr>
<td>E</td>
<td>Error</td>
</tr>
<tr>
<td>H</td>
<td>Held</td>
</tr>
<tr>
<td>K</td>
<td>In kernel</td>
</tr>
<tr>
<td>M</td>
<td>Mixed</td>
</tr>
<tr>
<td>R</td>
<td>Running</td>
</tr>
<tr>
<td>T</td>
<td>Stopped</td>
</tr>
<tr>
<td>W</td>
<td>At watchpoint</td>
</tr>
</tbody>
</table>

### TotalView – Root Window

<table>
<thead>
<tr>
<th>ID</th>
<th>Rank</th>
<th>Host</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>&lt;local&gt;</td>
<td>B</td>
<td>cell_mpi.0 (2 active threads)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>&lt;local&gt;</td>
<td>B</td>
<td>cell_mpi.1 (2 active threads)</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>&lt;local&gt;</td>
<td>B</td>
<td>cell_mpi.2 (2 active threads)</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>&lt;local&gt;</td>
<td>B</td>
<td>cell_mpi.3 (2 active threads)</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>&lt;local&gt;</td>
<td>B</td>
<td>cell_mpi.4 (2 active threads)</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>&lt;local&gt;</td>
<td>B</td>
<td>cell_mpi.5 (2 active threads)</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>&lt;local&gt;</td>
<td>B</td>
<td>cell_mpi.6 (2 active threads)</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>&lt;local&gt;</td>
<td>B</td>
<td>cell_mpi.7 (2 active threads)</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>qb630</td>
<td>B</td>
<td>cell_mpi.8 (2 active threads)</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>qb630</td>
<td>B</td>
<td>cell_mpi.9 (2 active threads)</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>qb630</td>
<td>B</td>
<td>cell_mpi.10 (2 active threads)</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>qb630</td>
<td>B</td>
<td>cell_mpi.11 (2 active threads)</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>qb630</td>
<td>B</td>
<td>cell_mpi.12 (2 active threads)</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>qb630</td>
<td>B</td>
<td>cell_mpi.13 (2 active threads)</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>qb630</td>
<td>B</td>
<td>cell_mpi.14 (2 active threads)</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>qb630</td>
<td>B</td>
<td>cell_mpi.15 (2 active threads)</td>
</tr>
</tbody>
</table>

### TotalView ID

- **TotalView ID:**

### MPI Rank

- **MPI Rank:**

---

**8/3/2009**

**Scaling to Petascale Virtual Summer School**
TotalView – Process Window

- Stack trace pane
  - Call stack of routines
- Stack frame pane
  - Local variables, registers and function parameters
- Source pane
  - Source code
- Action points, processes, threads pane
  - Manage action points, processes and threads
Controlling Execution

• The process window (TotalView) or main window (DDT) always focuses on one process/thread

• Switch between processes/threads
  – TotalView: p+/p-, t+/t-, double click in root window, process/thread tab
  – DDT: click on process rank in process window

• Need to set the appropriate scope when
  – Giving control commands
  – Setting action points
## Control Commands

<table>
<thead>
<tr>
<th>TotalView</th>
<th>DDT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>Play/Continue</td>
<td>Start/resume execution</td>
</tr>
<tr>
<td>Halt</td>
<td>Pause</td>
<td>Stop execution</td>
</tr>
<tr>
<td>Kill</td>
<td></td>
<td>Terminate the job</td>
</tr>
<tr>
<td>Restart</td>
<td></td>
<td>Restarts a running program</td>
</tr>
<tr>
<td>Next</td>
<td>Step over</td>
<td>Run to the next source line without stepping into another function</td>
</tr>
<tr>
<td>Step</td>
<td>Step into</td>
<td>Run to next source line</td>
</tr>
<tr>
<td>Out</td>
<td>Step out</td>
<td>Run to the completion of current function</td>
</tr>
<tr>
<td>Run to</td>
<td>Run to line</td>
<td>Run to the indicated location</td>
</tr>
</tbody>
</table>
Process/Thread Groups - TotalView

- Scope of commands and action points
  - Group(control)
    - All processes and threads
  - Group(workers)
    - All threads that are executing user code
  - Rank X
    - Current process and its threads
  - Process(workers)
    - User threads in the current process
  - Thread X.Y
    - Current thread
  - User defined group
    - Group -> Custom Groups, or
    - Create in call graph
Process/Thread Groups - DDT

- Create custom groups
  - Ctrl+click on all desired processes
  - Right click on the process window then “create group”
Action Points

• Breakpoints stop the execution of the processes and threads that reach it
  – Unconditional
  – Conditional: stop only if the condition is satisfied
  – Evaluation: stop and execute a code fragment when reached
    • Useful when testing small patches
• Process barrier points synchronize a set of processes or threads
  – TotalView only
• Watchpoints monitor a location in memory and stop execution when its value changes
Setting Action Points - TotalView

• Breakpoints
  – Right click on a source line -> Set breakpoint
  – Click on the line number

• Watch points
  – Right click on a variable -> Create watchpoint

• Barrier points
  – Right click on a source line -> Set barrier

• Edit action point property
  – Right click on a action point in the Action Points tab -> Properties
Setting Action Points - DDT

• Breakpoints
  – Double click on a source code line
  – Right click in the Breakpoints tab -> Add breakpoint

• Watch points
  – Right click on a variable -> Add to Watches
  – Right click in the Watches tab -> Add Watch
Viewing/Editing Data

- View values and types of variables
  - At one process/thread
  - Across all processes/threads
- Edit variable value and type
- Array Data
  - Slicing
  - Filtering
  - Visualization
  - Statistics
Viewing/Editing Data - TotalView

• Viewing data in
  – Stack frame
  – Expression list
  – Variable window (dive on a variable by double clicking on its name)

• Editing data by clicking on the value in
  – Stack frame
  – Variable window
Viewing/Editing Data - DDT

• Viewing data in
  – Variable window (in the main window)
  – Evaluation window

• Editing data
  – Right click on the variable name in the evaluation window
  – Then choose “Edit value” or “Edit type”
Viewing Dynamic Arrays in C/C++

• **TotalView**
  - Edit “type” in the variable window
  - Tell TotalView how to interpret the memory from a starting location
  - Example
    • To view an array of 100 integers
      - Int * -> int[100]*

• **DDT**
  - Drag a pointer variable into the evaluation window
  - Right click on the variable -> “View as vector”
MPI Message Queues

- Detect
  - Deadlocks
  - Load balancing issues
- TotalView
  - Tools -> Message Queue Graph
  - More options available
- DDT
  - View -> Message Queues
TotalView - Call Graph

- Tools -> Call graph
- Quick view of program state
  - Nodes: functions
  - Edges: calls
- Look for outliers

8/3/2009 Scaling to Petascale Virtual Summer School
DDT - Parallel Stack View

• Allow users to see the position of each process/thread in the source code in same window

• Hover over any function to see a list of processes that are currently at that location
References

• TotalView user manual
• DDT user manual
• LLNL TotalView tutorial
  – https://computing.llnl.gov/tutorials/totalview
• NCSA Cyberinfrastructure Tutor
  – “Debugging Serial and Parallel Codes” course
• HPCBugBase
  – http://hpcbugbase.org/index.php/Main_Page