A Semi-automatic System for Application-level Checkpoint-Recovery

Keshav Pingali
Cornell University

Joint work with Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Martin Schulz(LLNL), Paul Stodghill,
Why CPR?

- Program runtimes are exceeding MTBF of hardware
  - Protein-folding on Blue Gene may take one year per protein
  - Fault-tolerance is critical
- Fault tolerance comes in different flavors
  - Mission-critical systems: no down-time, fail-over, redundancy
  - Computational science applications: restart after failure, minimizing lost work
- Fault models
  - Fail-stop:
    - Failed process dies silently w/o corrupting data
  - Byzantine:
    - Arbitrary misbehavior is allowed
- Our focus:
  - Computational science applications
  - Fail-stop faults
  - One solution: checkpoint/restart (CPR)
Why CPR? (contd.)

- **Grid computing: utility computing**
  - Programs execute wherever there are computational resources
  - Program are mobile to take advantage of changing resource availability
  - Key mechanism: checkpoint/restart
  - Identical platforms at different sites on grid
    - Platform-dependent checkpoints (cf. Condor)
  - Different platforms at different sites on grid
    - Platform-independent (portable) checkpoints
Two approaches to CPR

- **System-level checkpointing (SLC) (eg) Condor**
  - core-dump style snapshots of computations
  - mechanisms very architecture and OS dependent
  - checkpoints are not portable

- **Application-level checkpointing (ALC)**
  - programs are self-checkpointing and self-restarting
    - (eg) n-body codes save and restore positions and velocities of particles
  - amount of state saved can be much smaller than SLC
    - IBM’s BlueGene protein folding: megabytes vs terabytes

- **Disadvantage of current application-level check-pointing**
  - manual implementation
  - requires global barriers in programs
Our approach

- Original Application
  - Precompiler
  - Application + State-saving
    - Thin Coordination Layer
      - Reliable communication layer
      - MPI Implementation
      - MPI Implementation
  - Failure detector
Cornell Checkpointing Compiler ($C^3$) Project

- Automate application-level check-pointing of C programs
  - minimize programmer effort
- MPI programs: [PPoPP 2003, ICS2003, SC2004]
  - coordination of single-process states into a global snapshot
  - non-blocking protocol: no barriers needed in program
- OpenMP programs: [EWOMP 2004, ASPLOS 2004]
  - blocking protocol
- Portable MPI checkpointing: [SC 2005]
  - Requires type information for each object created at runtime
  - Pre-compiler analyzes C programs and flags potential portability problems
  - Successfully restarted and completed 64 processor PSC Lemieux checkpoints on Cornell Windows cluster
- Ongoing work
  - program analysis to reduce amount of saved state
  - Other languages: C++
Outline

• **Pre-compiler:**
  – saving state at application level

• **Check-pointing MPI programs**
  – Non-blocking protocol

• **Check-pointing OpenMP programs**
  – Blocking protocol

• **Portable check-pointing**
  – Restart on different platform

• **Ongoing work**
Precompiler

• Where to checkpoint
  – At calls to `potentialCheckpoint()` function
    • Mandatory calls in main process (initiator)
    • Other calls are optional
      – Process checks if global checkpoint has been requested, and if so, joins in protocol to save state
  – Inserted by programmer or automated tool
    • Currently inserted by programmer

• Transformed program can save its state only at calls to `potentialCheckpoint()`
Application-level checkpointing: saving position in program

main()
{
    int a;
    if(restart)
        load LS;
        copy LS to LS.old
        jump dequeue(LS.old)
    // ...
    LS.push(label1);
    label1:
        function();
    LS.pop();
    // ...
}

- Recovery structure LS (Location Stack) keeps track of function calls that could lead to potentialCheckpoint
- Code for updating LS is inserted by pre-compiler
- On recovery, function calls on LS are repeated to rebuild the stack frames
- Portable way of saving the PC
Saving Application State

• **Stack**
  – Location stack (LS): track which function invocations led to place where checkpoint was taken
  – Variable Description Stack (VDS): records local variables in these function invocations that must be saved
  – On recovery
    • LS is used to re-execute sequence of function invocations and re-create stack frames
    • VDS is used to restore variables into stack

• **Heap**
  – special malloc that tracks memory that is allocated and freed

• **Globals**
  – precompiler inserts statements to save them
Sequential Experiments (vs Condor)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Bench</th>
<th>Class</th>
<th>Size in megabytes</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Condor</td>
<td>$C^3$</td>
</tr>
<tr>
<td>Solaris</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BT</td>
<td>A</td>
<td></td>
<td>308.85</td>
<td>306.39</td>
</tr>
<tr>
<td>CG</td>
<td>B</td>
<td></td>
<td>429.89</td>
<td>427.44</td>
</tr>
<tr>
<td>EP</td>
<td>A</td>
<td></td>
<td>3.46</td>
<td>1.00</td>
</tr>
<tr>
<td>FT</td>
<td>A</td>
<td></td>
<td>421.28</td>
<td>418.69</td>
</tr>
<tr>
<td>IS</td>
<td>A</td>
<td></td>
<td>100.45</td>
<td>96.00</td>
</tr>
<tr>
<td>LU</td>
<td>A</td>
<td></td>
<td>46.99</td>
<td>44.54</td>
</tr>
<tr>
<td>MG</td>
<td>B</td>
<td></td>
<td>436.99</td>
<td>435.48</td>
</tr>
<tr>
<td>SP</td>
<td>A</td>
<td></td>
<td>82.09</td>
<td>79.63</td>
</tr>
<tr>
<td>Linux</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BT</td>
<td>A</td>
<td></td>
<td>307.13</td>
<td>306.39</td>
</tr>
<tr>
<td>CG</td>
<td>B</td>
<td></td>
<td>428.17</td>
<td>427.44</td>
</tr>
<tr>
<td>EP</td>
<td>A</td>
<td></td>
<td>1.74</td>
<td>1.00</td>
</tr>
<tr>
<td>FT</td>
<td>A</td>
<td></td>
<td>419.43</td>
<td>418.69</td>
</tr>
<tr>
<td>IS</td>
<td>A</td>
<td></td>
<td>96.74</td>
<td>96.00</td>
</tr>
<tr>
<td>LU</td>
<td>A</td>
<td></td>
<td>45.27</td>
<td>44.54</td>
</tr>
<tr>
<td>MG</td>
<td>B</td>
<td></td>
<td>435.24</td>
<td>435.55</td>
</tr>
<tr>
<td>SP</td>
<td>A</td>
<td></td>
<td>80.36</td>
<td>79.63</td>
</tr>
</tbody>
</table>

- Checkpoint sizes are comparable.
Runtime Overheads (Linux)
Check-pointing MPI programs
Need for Coordination

- Horizontal Lines – events in each process
- **Recovery Line**
  - line connecting checkpoints on each processor
  - represents global system state on recovery
- **Problem with Communication**
  - messages may cross recovery line
Late Messages

- Record message data at receiver as part of checkpoint
- On recovery, re-read recorded message data
Early Messages

- Must suppress the resending of message on recovery
- What about non-deterministic events before the send?
  - Must ensure the application generates the same early message on recovery
  - Record and replay all non-deterministic events between checkpoint and send
  - In our applications, non-determinism arises from wild-card MPI receives
Difficulty of Coordination

- No communication $\rightarrow$ no coordination necessary
Difficulty of Coordination

- No communication → no coordination necessary
- BSP Style programs → checkpoint at barrier
Difficulty of Coordination

- No communication → no coordination necessary
- BSP Style programs → checkpoint at barrier
- General MIMD programs
  - System-level checkpointing (eg. Chandy-Lamport)
    - Forces checkpoints to avoid early messages
    - Only consistent cuts
Difficulty of Coordination

- No communication → no coordination necessary
- BSP Style programs → checkpoint at barrier
- General MIMD programs
  - System-level checkpointing (eg. Chandy-Lamport)
    - Only late messages: consistent cuts
  - Application-level checkpointing
    - Checkpoint locations fixed, may not force
    - Late and early messages: inconsistent cuts
    - Requires new protocol
MPI-specific issues

• Non-FIFO communication – tags
• Non-blocking communication
• Collective communication
  – MPI_Reduce(), MPI_AllGather(), MPI_Bcast()…
• Internal MPI library state
  – Visible:
    • non-blocking request objects, datatypes, communicators, attributes
  – Invisible:
    • buffers, address mappings, etc.
Implementation

• **Two large parallel platforms**
  – Lemieux: Pittsburgh Supercomputing Center
    • 750 Compaq Alphaserver ES45 nodes
    • Node: four 1-GHz Alpha processors, 4 GB memory, 38GB local disk
    • Tru64 Unix operating system.
    • Quadrics interconnection network.
  – Velocity 2: Cornell Theory Center
    • 128 dual-processor Windows cluster

• **Benchmarks:**
  – NAS suite: CG, LU, SP
  – SMG2000, HPL
Overheads on Lemieux
Outline

• **Pre-compiler:**
  – saving state at application level

• **Check-pointing MPI programs**
  – Non-blocking protocol

• **Check-pointing OpenMP programs**
  – Blocking protocol

• **Portable check-pointing**
  – Restart on different platform

• **Ongoing work**
Blocking Protocol

- **Protocol:**
  - Barrier
  - Each thread records own state
  - Thread 0 records shared state
  - Barrier
• Protocol introduces new barriers into program
• May cause errors or deadlock
• If checkpoint crosses barrier:
  – Thread A reaches barrier, waits for thread B
  – Thread B reaches checkpoint, calls first barrier
  – Both threads advance
    • Thread B recording checkpoint
    • Thread A computing, possibly corrupting checkpoint
Deadlock due to locks

- **Suppose checkpoint crosses dependence**
  - Thread B grabs lock, will not release until after checkpoint
  - Thread A won’t checkpoint before it acquires lock
- **Since checkpoint is barrier: deadlock**
Experimental Setup

• SPLASH-2 benchmarks
  – Generic shared memory benchmarks
  – Can be specialized to any shared memory API

• Test platforms:
  – 2-way Athlon / Linux
    • Compared to BLCR
  – 4-way Alpha EV68 / Tru64
  – Windows/x86
Linux/86

Checkpoints saved to local disk
Tru64/Alpha

1 Checkpoint Overheads

Checkpoints saved to network file system.
Portable Checkpointing Problem

• Migrate checkpoint to a different platform and restart there

• Dimensions of heterogeneity
  – Architecture (Pentium, Alpha, Sparc, Itanium, …)
  – OS (Windows, Linux, Solaris, TRU64 Unix, …)
  – Compilers (GNU, SUNWSPRO, Intel, HP, Visual C)
  – MPI implementations (MPIPro, MPICH, LAM)

• Approach:
  – Each data object in checkpoint must have an associated type
  – Migrating checkpoint requires translating between representations of types
Ensuring Correctness

- C is a fairly low-level language with weak typing guarantees
- Portable checkpointing requires strong type information
- Program analysis
  - Identify constructs that could lead to errors in state translation
  - Flag warnings that allow programmers to rectify portability problems with programs
Performance Results

• **Sequential**
  – 3.6 Ghz Dell Dimension Desktop (Windows, Enterprise Linux)
  – Sun UltraSPARC IIIi 1060MHz(Solaris 9)

• **MPI Blocking Checkpointing**
  – Velocity Cluster at Cornell Theory Center
  – Lemieux Cluster at Pittsburgh Supercomputing Center
Comparing Performance with Condor on Linux

![Checkpointing Overhead on x86-lin-gcc](image)
Portable vs Non-portable Restart

Comparison of restart and completion time for native and portable checkpoints on 16 processors on Velocity

![Bar graph showing comparison of restart and completion time for native and portable checkpoints on 16 processors on Velocity.](image-url)
Ongoing work

- Integration of MPI and OpenMP sub-systems
- Integration with Pittsburgh Supercomputing Center (PSC) system for saving checkpoint data
- Compiler analysis to reduce the amount of saved state (with Radu Rugina)
  - Identify live data
  - Incremental checkpointing
  - Recomputation vs state-saving
- Other languages
  - C++
Summary

• System for CPR of MPI and OpenMP apps
  – Application-level checkpointing for C programs
  – Programs become self-checkpointing and self-restarting
• Precompiler-based single-process checkpointer
  – Minimal programmer annotations
• Novel protocols
  – Work with any single-process checkpointer
  – Portable across MPI and OpenMP implementations
• Components orthogonal
  – Can be used/applied independently
• Portable checkpointing
  – Need type information for each object created at runtime
• Overhead is low
• For more information: http://iss.cs.cornell.edu