



Compiler Assisted, Application-level, Checkpointing for Parallel Applications

Keshav Pingali, Paul Stodghill
Greg Bronevetsky, Dan Marques

mailto:stodghil@cs.cornell.edu
<http://www.cs.cornell.edu/Info/Projects/Bernoulli/>
<http://www.asp.cornell.edu/>



My Aim

- Enumerate a number of different dimensions of checkpointing complexity.
 - Describe where the ASP/Bernoulli FT project fits in.
 - Have you tell me where you fit in.
-

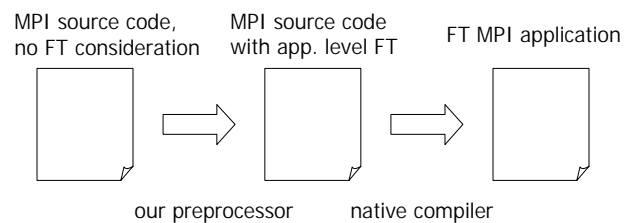
The Design Dimensions

- Fault-models
- Complexity of (the model of) parallelism
- Approach to sequential processor state saving
- Portability
- Scalability
- Visibility

ASP/Bernoulli FT

Our goal:

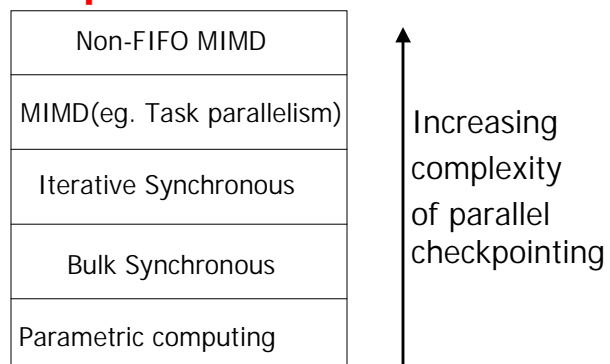
- Develop a preprocessor that will transparently add application-level checkpointing to MPI applications
 - As easy to use as system-level checkpointing
 - As efficient as user-specified application-level checkpointing
- Applications
 - C/C++/Fortran
 - Multiple source files
 - Message-passing parallelism (MPI) (not shared memory)



Fault-models

- Predictable n-way fail/stop
 - Batch systems, Job migration
- N-way fail/stop (← we are here)
- K-way fail/stop
- ...
- Byzantine

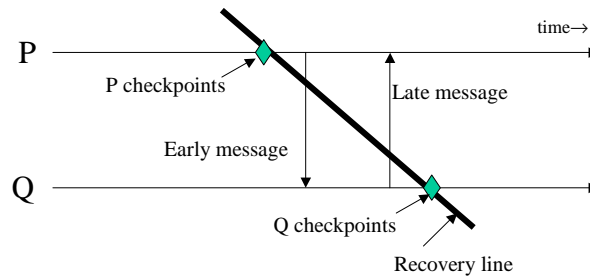
Complexity of (the model of) parallelism



- We handle all models
- We want to handle all models efficiently.

Parallel Application-level Checkpointing

- Must ensure that a “consistent” global checkpoint is computed.
 - Identify the recovery line
 - Handle late messages – on recovery, resend or replay from log
 - Handle early (inconsistent) messages – almost all existing systems prevent
 - Save state of the message passing system



Approach to sequential processor state saving

- System-level
 - Simple for the programmer
 - Can checkpoint at any time
 - Not efficient
 - Alegra system (Sandia Lab)
 - App. level restart file only 5% of core size
 - IBM's BlueGene protein folding
 - Sufficient to save positions and velocities of bases
- Application- (i.e., source-) level (← we are here)
- What needs to be saved
 - Global variables
 - Stack frames (ie, execution context, local variables)
 - Heap allocated objects
 - Hidden state

Sequential checkpointing (cont.)

- Global variables
 - Currently – explicitly registered
 - A better approach – linker magic (à la C++ constructor/destructors)
- Stack frames
 - Compilation
 - Generate stack frame descriptors
 - Insert labels on call sites
 - Execution
 - Caller – push/pop call site label
 - Callee – push/pop stack frame descriptors

Example: Saving/Restoring Stack

- Before

```
void f(int x, int y) {
    int z;
    g(x+y);
    h(x+y);
}
```

- Comments

- Pointers are not portable.
- Requires the same stack base on restart.

- After

```
void f(int x, int y) {
    int z;
    PushFrame();
    FrameVar(&z, sizeof(z));
    if (restart) {
        FrameRestoreVars();
        switch (FrameReadCall()) {
            case 1: goto L1;
            case 2: goto L2;
        }
    }
    L1:
    FrameWriteCall(1);
    g(x+y);
    L2:
    FrameWriteCall(2);
    h(x+y);
    PopFrame();
}
```

Saving/restoring the Heap

- Currently – save/restore the whole heap
 - Region-based allocation
- ```

rid = region_new();
void* p =
 region_malloc(rid,bytes);
region_hint(rid1,
 REGION_DONT_SAVE);
region_hint(rid2,
 REGION_SAVE_ONCE);
region_hint(rid3,
 REGION_SAVE_INCREMENTALLY);

```
- Conservative Garbage Collection
- ```

list* p;
list_push(p,1);
list_push(p,2);
list_push(p,3);
ignore_root(p);
ignore_root(q,
    func_to_recompute)
;

```
- Two approaches can be combined.

Sequential checkpointing (cont.)

- Hidden state
 - Library state (e.g., random seed, PETSc – matrix repr.)
 - OS state (e.g, open files)
 - Our decision: no hidden state, except MPI

Portability

- **Sequential state**
 - Can't be done in C.
 - Prove me wrong, please!
 - Change the language - Safe C, Managed C++, ...
 - Use a different language
 - Java, C#, O'Caml, ...
 - HPF(?)
- **Distributed state**
 - Being able to restart with a different number of processors.
 - Two approaches
 - Save with no decomposition (ie, the global view)
 - Over decomposition
- **Our decision: non-portable**

The Rest

- **Scalability**
 - If the application has not global synchronization, can we afford to introduce it?
 - If no, can't use MPI_BARRIER for checkpointing
 - Non-blocking protocols are necessary.
 - Our decision: no additional global synchronization
- **Visibility**
 - Do external processors rely on failed processors?
 - If yes, rollback is not sufficient
 - Message logging is required
 - Our decision: visibility is not an issue

Ongoing work

- **Currently:**
 - Compiler adds sequential state saving to MPI applications
 - Layer b/w application and MPI to handle non-FIFO, MIMD applications with application-level checkpoints.
 - Rough implementations are done.
 - Debugging and performance evaluation has started.
 - Will identify performance bottlenecks
- **The fun stuff: Compiler analysis and optimization**
 - Eliminate explicit frame saving/restoring.
 - Identify dead and read-only data.
 - Identify data that can be recomputed instead of being saved.
 - Identify optimal points for checkpointing.
 - Select optimal parallel checkpointing protocol.

Conclusions

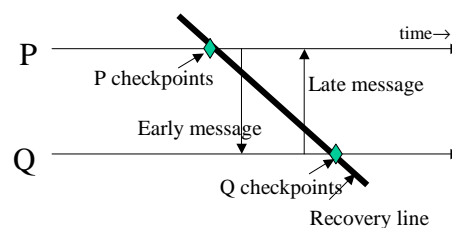
- **The Dimensions**
 - Fault-models
 - Complexity of (the model of) parallelism
 - Approach to sequential processor state saving
 - Portability
 - Scalability
 - Visibility
- **Implications**
 - Limitations on the application
 - Parallel model
 - Source language
 - Restrictions/Requirements of underlying system
 - MPI + Application-level \Rightarrow new snapshot algorithms

Related work

- **Compiler for portable checkpoints**
 - PORCH system, Ramkumar, Strumpfen (Iowa / MIT)
- **Compiler optimization of checkpointing**
 - CATCH, Li, Fuchs (Illinois)
 - Live/Clean/Dead variable analysis, Plank, Beck, Kingsly (Univ. Tennessee)
- **Conservative Garbage Collection**
 - Hans Boehm
 - Joel Bartlett
- **Checkpointing Parallel Apps.**
 - “A Survey of Rollback-Recovery Protocols in Message-Passing Systems”. Elnozahy, et al. *ACM Computing Surveys*, 34:3, Sept. 2002.

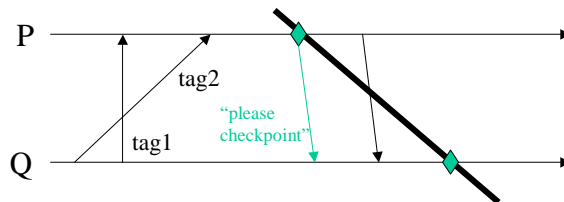
Parallel Application-level Checkpointing

- **From Distributed Systems**
 - System can force application to checkpoint at any time
 - Communications channels are FIFO
 - Early messages are prevented for occurring.
- **Scalability**
 - Can't afford to block (ie, Can't use MPI_BCAST or MPI_BARRIER)
 - One processor decides that checkpointing should occur
 - Propagates “please checkpoint” to other processors
 - Every time a processor calls the checkpointing function
 - Poll to see if checkpointing is underway. If not, doesn't checkpoint



MPI+Application-level Checkpointing: Implications for snapshots

- Application-level checkpointing
 - Checkpoints can only be taken at calls to checkpoint function
 - Early messages must be handled
- MPI
 - Multiple tags and communicators
 - Non-FIFO semantics



Implications for Underlying MPI

- Require low-level FIFO communication? No!
 - Limits MPI implementation
 - e.g., LA-MPI uses multiple network channels for reliability
 - Does not eliminate inconsistent messages
 - Inherent to non-blocking application-level checkpointing
 - Protocol for non-FIFO messages and delayed checkpoints
 - Developed and implemented
 - Being documented and debugged
- Must save/restore library state
 - Communications, Groups, Datatypes
 - Request and status objects
 - Or, provide a thin layer b/w application and MPI to manage state
- Other hooks for our protocol
 - Message ID's, additional bytes in message headers.

How to do parallel checkpointing

- Parametric computing
 - No communication means no complications
 - Sequential checkpointing sufficient
- Bulk Synchronous
 - Global barriers are present
 - Leverage for checkpointing
- ...
- Non-FIFO MIMD
 - Scalability: No global synchronization to be exploited.
 - MPI: Communication channels are non-FIFO
 - Application-level checkpointing: Delayed checkpointing gives rise to “inconsistent” messages.
 - No algorithm in the distributed systems literature
 - We have developed such an algorithm (another talk).