C[3]: A System for Automating Application-level Checkpointing of MPI Programs

Greg Bronevetsky, Daniel Marques,
Keshav Pingali, Paul Stodghill

Department of Computer Science,

Cornell University

The Problem

- Decreasing hardware reliability
  - Extremely large systems (millions of parts)
  - Large clusters of commodity parts
  - Grid Computing
- Program runtimes greatly exceed mean time to failure
  - ASCI, Blue Gene, PSC, Illinois Rocket Center
- ∴ Fault-tolerance necessary for high-performance computing

What kinds of failures?

Fault Models

- Byzantine: a processor can be arbitrarily malicious (e.g., incorrect data, a hacker, etc.)
- Fail-silent: a processor stops sending and responding to messages
- Fail-Stop: fail-silent + surviving processors can tell

Number of component failures

- 1, *k*, *n*

In this work, Fail-Stop faults, *n* processors

- Necessary first step
- Usually sufficient in practice

What is done by hand?

- Application-level (i.e., source code) Checkpointing
  - Save key problem state vs system (core) state
  - Used at Sandia, BlueGene, PSC, …
- Advantages:
  - Minimizes amount of state saved
    - e.g., Alegra (application state = 5% of core size)
    - Crucial for future large systems (BlueGene: Mb's vs Tb's)
  - Can be portable across platforms and MPI implementations
- Disadvantages:
  - Lots of manual work
  - Correctly checkpointing programs without barriers requires a coordination protocol
- We want to automate this process

Goals

- A tool to convert existing MPI applications into fault-tolerance MPI applications
- Requirements
  - Use Application-level checkpointing
  - Use native MPI implementation
  - Handle full range of MPI semantics
- Desirable features
  - Minimize programmer annotations
  - Automatically optimize checkpoints sizes
- Necessary technologies
  - Program transformations for application-level checkpointing
  - Novel algorithm for distributed application-level checkpointing

Programmer's Perspective

- Programmer places calls to *potentialCheckpoint()*
- Precompiler transforms program to save application state at *potentialCheckpoint()* calls
- Runtime system decides at each *potentialCheckpoint()* whether or not a checkpoint is taken

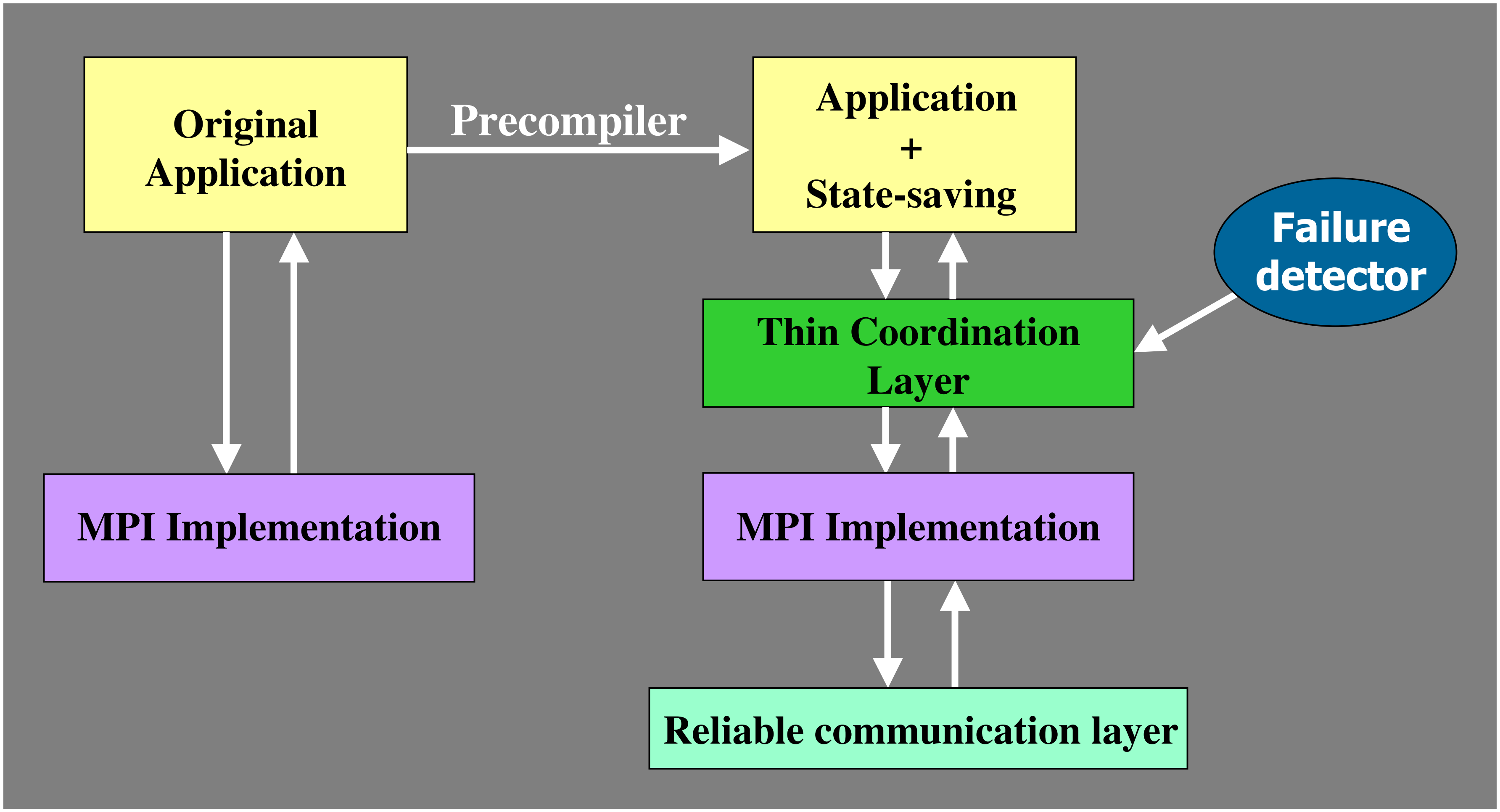
```
int main()
{
    MPI_Init();

    initialization();
→   potentialCheckpoint();

    while (t < t_max) {
        big_computation();
        …
→       potentialCheckpoint();
    }

    MPI_Finalize();
}
```

C^3 Architecture



Outline

- Introduction
- The paper
  - Precompiler
  - Coordination Layer
  - Performance
- *Current work*
  - *Optimizing Checkpoint Size*
  - *Other Current Work*
- Related Work
- Conclusions

Precompiler

Transformation to save application state:

- Parameters, local variables, program location
  - Record local variables and function calls
  - Checkpoint: save record
  - Recovery: reconstruct application state from record
  - Only functions on path to *potentialCheckpoint()* calls must be instrumented
- Globals
  - main() is instrumented to record global locations
- Heap
  - Custom, checkpointable heap allocator

What about the network “state”?

```
int main()
{
    if (recovery) { … goto Lx; … }
    add_globals(…);
    push_locals(&t, &tmax);
    MPI_Init();

    initialization();
    L1: potentialCheckpoint();

    while (t < tmax) {
        big_computation();
        …
        L2: potentialCheckpoint();
    }

    MPI_Finalize();
    pop_locals();
}
```

Can existing Distributed Systems solutions be used?

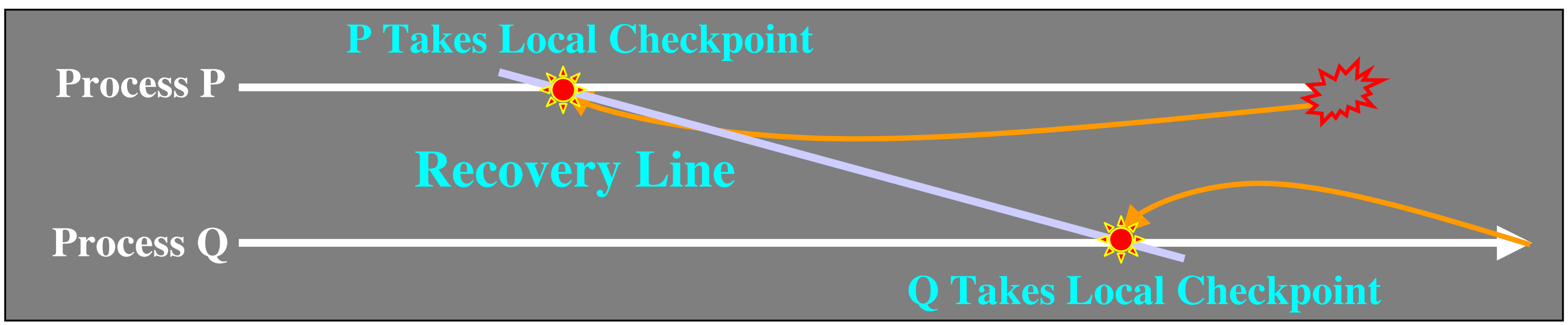
- Why not checkpoint at barriers?
  - What barriers?
  - MPI is non-FIFO! Messages cross barriers!
- Why not use message logging?
  - Does not handle *n* failures
  - Constant overhead, even when no failures
  - Message logs fill memory in minutes (seconds)
  - Checkpointing to clear logs
- Why not use Chandy-Lamport (or your other favorite distributed snapshot algorithm)?
  - Requires system-level checkpointing for correctness or progress
  - Can Tb's of data be saved before a component fails?

Distributed Application-level Checkpointing

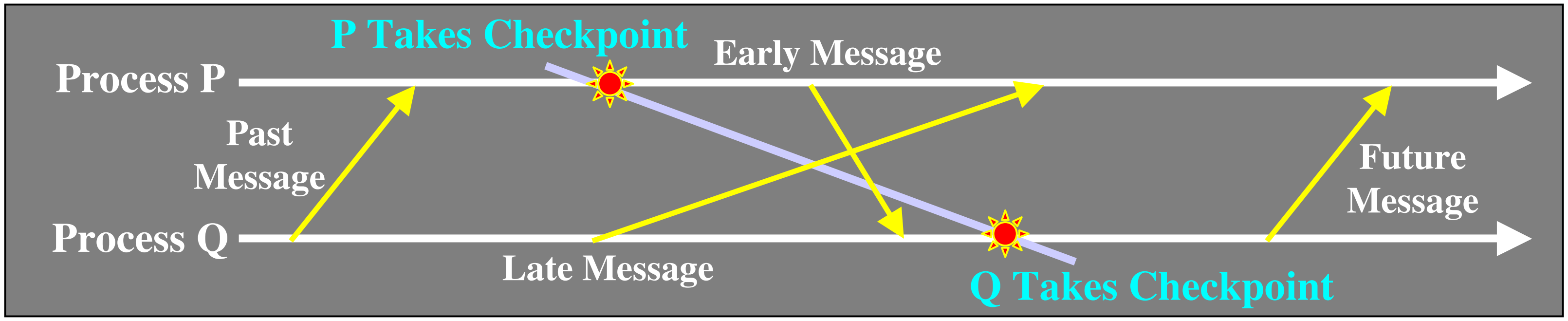
- Potential checkpoint locations are fixed in program source code
- May not force checkpoints to happen at any other time

Distributed Application-level Checkpointing (cont.)

- Recovery Line
  - A set of checkpoints, one per processor
  - represents global system state on recovery
  - When one node fails, everybody rolls back to a recent recovery line
- Problems to solve
  - How to select *potentialCheckpoint()*'s for recovery line?
  - What about MPI messages that cross recovery line?



Distributed Application-level Checkpointing (cont.)

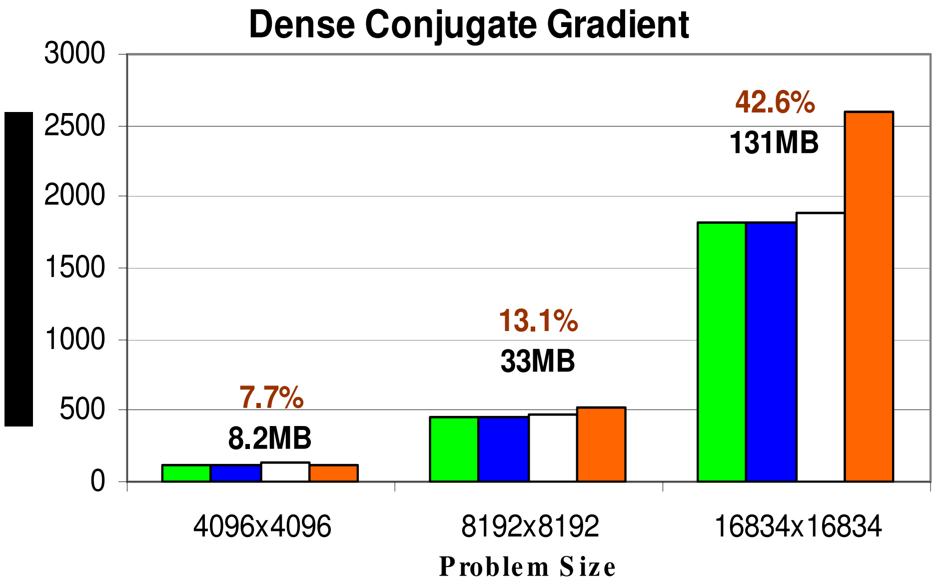


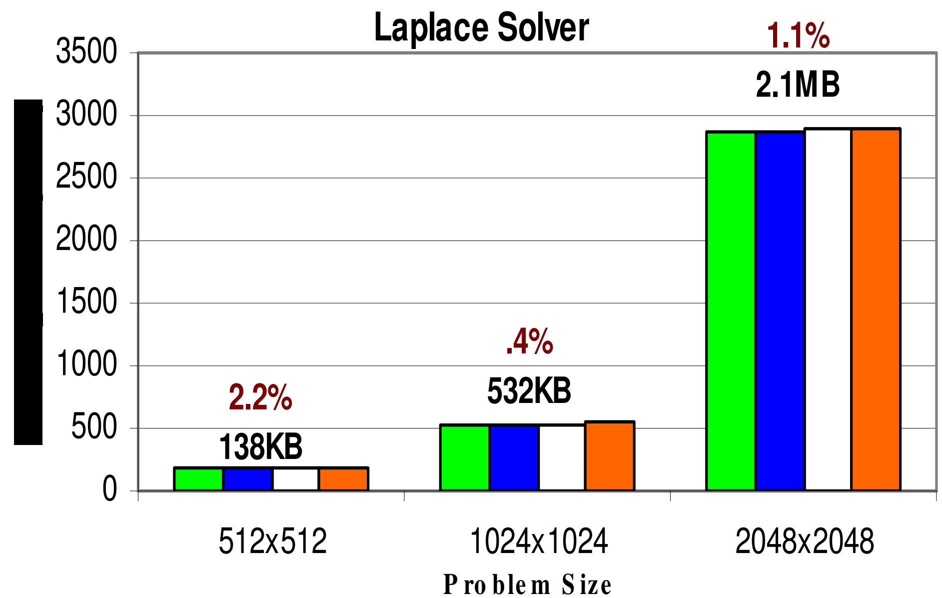
- Past and Future Messages
  - Do not require coordinate
- Late messages
  - Require recording and replaying
- Early messages
  - a.k.a., Inconsistent messages
  - Require suppression
  - Recording non-determinism
- Collective communication
  - Combinations of message types
- Hidden state
  - MPI_Request, MPI_Communication
- Synchronization semantics
  - MPI_Barrier, MPI_SSend, …
- Protocol details in the paper

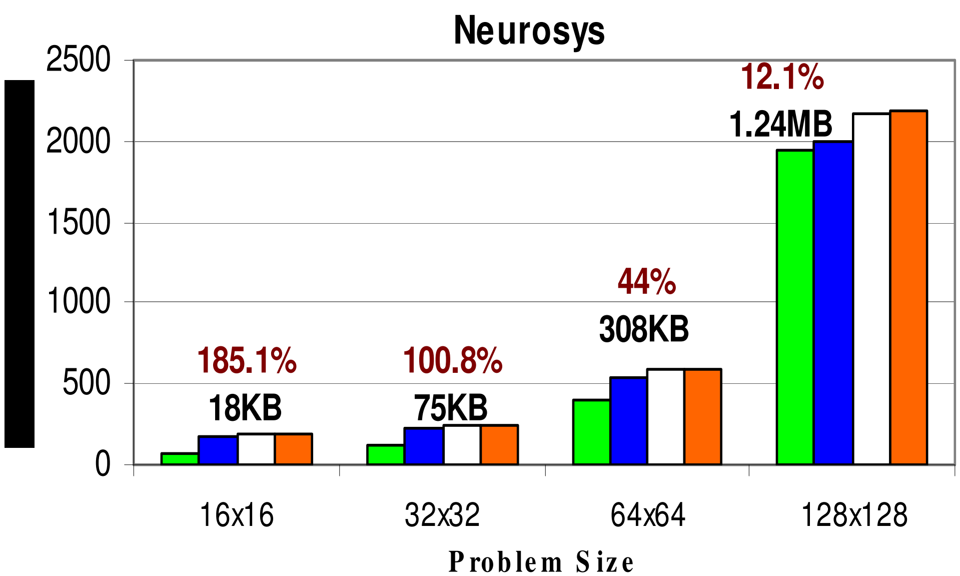
Performance

- Prototype implementation
  - Precompiler without optimizations
  - Point-to-point protocol, no collective, no synchronization
- Three benchmarks scientific codes
  - Dense Conjugate Gradient
  - Laplace Solver
  - Neuron Simulator
- 16 processors of Velocity cluster at CTC
- 30 second checkpoint interval
  - ***<u>Overheads amplified for better resolution</u>***

Overheads







- Original Application
- Piggybacking Control Data, No Recording, No Checkpointing
- Piggybacking Control Data, Recording, No Checkpointing
- Piggybacking Control Data, Recording, Checkpointing

The numbers above each of bars are the total overhead and the size of the application state, respectively.

Most test show overheads .4%-12.1% despite 30 second checkpoint intervals!

Outline

- Introduction
- The paper
  - Precompiler
  - Coordination Layer
  - Performance
- *Current work*
  - *Optimizing Checkpoint Size*
  - *Other Current Work*
- Related Work
- Conclusions

Live Variables

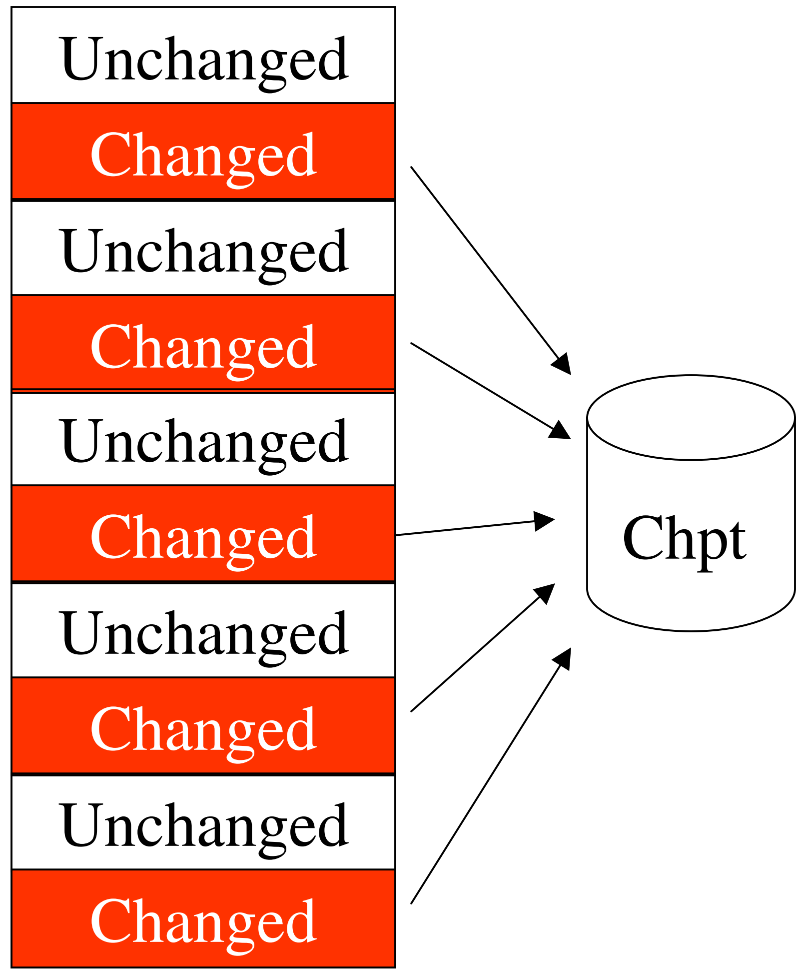
- Recent work by Jim Ezick
- Context-Sensitive Gen/Kill analysis
  - Utilizes new technique for encoding functions
- Works with full C language
- Analysis generates three levels of output each admitting a different C^3 optimization
  - Flow-Insensitive/Context-Insensitive : Eliminate push/pop instructions
  - Flow-Sensitive/Context-Insensitive : Generate Exclusion List
  - Flow-Sensitive/Context-Sensitive : Generate DFA to determine liveness

Live Variables (cont.)

- Effectiveness
  - Run on “treecode”, a popular Barnes-Hut algorithm for n-body simulation written in C
  - Given checkpoint location:
    - Finds a live variable set competitive with programmer provided state saving routine
    - Live variable <50% of total “in-scope” variables
    - Only two of 27 elements of the live variable set require a DFA
  - It remains to reduce the amount of heap saved

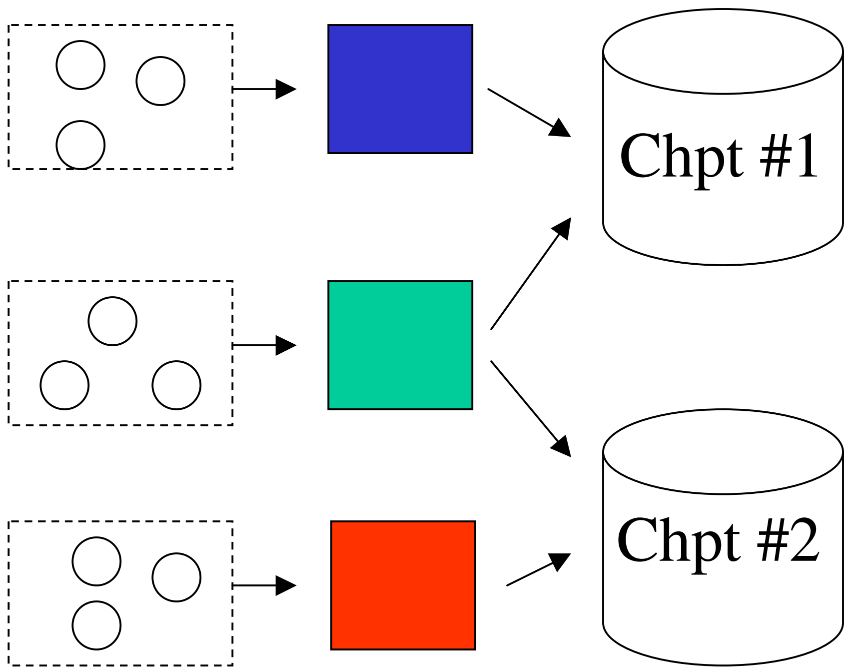
Optimizing Heap Checkpointing

- Saving the whole heap
  - Saves “dead” values
  - Saves unchanged since previous checkpoint
- Incremental checkpointing: Save only changed pages
  - Changed and unchanged on same page: false sharing
  - Still saves “dead” values
  - Saving every fragmented page set is slowing than saving the whole heap



Optimizing Heap Checkpointing (cont.)

- Allocation coloring
  - Assign each allocated object to a color
  - No two colors assigned to same page
  - Checkpoint: save subset of colors
  - Similar to (but different from) region analysis
- Automatic Allocation Coloring:
  - assign colors to allocation sites
  - Assign colors to *potentialCheckpoint()* calls
- Such that,
  - Minimize number of colors saved at checkpoints
  - Minimize number of pages saved at checkpoints
- Live Variables is necessary for Automatic Allocation Coloring



Other Current work

- Precompiler
  - Multiple source files
  - Colored heap allocation
  - Release by 4Q03
- Coordination layer
  - Complete reimplementation
  - All pt-to-pt and collective calls, communicators, datatypes, etc.
  - Correctness, performance, robustness
  - Release by 4Q03
- Shared memory
  - Model shared memory objects as “processors”, g_i
  - Shared memory reads: $g_i \rightarrow p_j$
  - Shared memory writes: $g_i \rightarrow p_j$
  - How to obtain consistent value of g_i?
- Grid computing
  - Goal: Migrate running application between clusters
  - Different number of processors: over decomposition, threaded execution
  - Heterogeneity:
Type-safe languages – Cyclone

Related Work

- Fault Tolerant MPI
  - FT-MPI, LA-MPI, CoCheck, …
  - None allow application-level checkpointing
- Precompiler
  - Similar to work done with PORCH (MIT)
    - PORCH is portable but not transparent to programmer
- Checkpoint optimization
  - CATCH (Illinois): uses runtime learning rather than static analysis
  - Beck, Plank and Kingsley (UTK): memory exclusion analysis of static data

Conclusions

- *C*[3] – Automatic fault-tolerance for MPI codes
  - Precompiler
  - Communication coordination layer
  - Performance results are encouraging
- Ties together many areas of compiler and systems
  - Language design
  - Interprocedural data-flow analysis
  - Region analysis
  - Memory allocation
  - Message passing, shared memory
  - Grid computing