Fault Tolerance in Charm++/AMPI

Sayantan Chakravorty

PPL, UIUC

April 19, 2007

Motivation

- Larger machines available, clusters as well as proprietary
- MTBF decreases as size of machines increases
- Long running applications have to tolerate faults

Background

Background

- Checkpoint
  - Coordinated: Cocheck, Starfish, Clip
  - Uncoordinated: suffers from cascading rollbacks
  - Communication: does not scale well

Background

- Checkpoint
  - Coordinated: Cocheck, Starfish, Clip
  - Uncoordinated: suffers from cascading rollbacks
  - Communication: does not scale well
- Message Logging
  - Pesssimistic: MPICH-V1, MPICH-V2 etc.
  - Optimistic: cascading rollback, complicated recovery
  - Causal Logging: causalty tracking, Manetho, MPICH-V3

Background

- Checkpoint
  - Coordinated: Cocheck, Starfish, Clip
  - Uncoordinated: suffers from cascading rollbacks
  - Communication: does not scale well
- Message Logging
  - Pesssimistic: MPICH-V1, MPICH-V2 etc.
  - Optimistic: cascading rollback, complicated recovery
  - Causal Logging: causalty tracking, Manetho, MPICH-V3
- Hybrid: Schultz et al, Bronevetsky et al

Solutions in Charm++

Solutions in Charm++

- Reactive: react to a fault
    - Disk based
    - In-memory
    - Message logging with fast recovery

Solutions in Charm++

- Reactive: react to a fault
    - Disk based
    - In-memory
    - Message logging with fast recovery
- Pro-active: act before a fault
    - Fault prediction
    - Evacuate processors after fault is predicted

Disk-based Checkpoint

Disk-based Checkpoint

- Blocking Coordinated Checkpoint
  - State of chares are checkpointed to parallel file system
  - Collective **MPI_Checkpoint(DIRNAME)**

Disk-based Checkpoint

- Blocking Coordinated Checkpoint
  - State of chares are checkpointed to parallel file system
  - Collective **MPI_Checkpoint(DIRNAME)**
- Restart
  - Whole job is restarted
  - Same job can be restarted on different # of processors
  - Runtime flag: **+restart DIRNAME**

Disk-based Checkpoint

- Blocking Coordinated Checkpoint
  - State of chares are checkpointed to parallel file system
  - Collective **MPI_Checkpoint(DIRNAME)**
- Restart
  - Whole job is restarted
  - Same job can be restarted on different # of processors
  - Runtime flag: **+restart DIRNAME**
- Simple yet effective for common cases

Drawbacks of disk-based checkpoint

- Checkpoints to the parallel file system are slow
- High **Recovery time**:

Drawbacks of disk-based checkpoint

- Checkpoints to the parallel file system are slow
- High **Recovery time**:
  - Time between the last checkpoint and the crash

Drawbacks of disk-based checkpoint

- Checkpoints to the parallel file system are slow
- High **Recovery time**:
  - Time between the last checkpoint and the crash
  - Time to resubmit the job and have it run

In-memory Double Checkpoint: Checkpoint

In-memory Double Checkpoint: Checkpoint

- Coordinated checkpoint

In-memory Double Checkpoint: Checkpoint

- Coordinated checkpoint
- Each object maintains 2 checkpoints:
    - On local processor
    - On a remote **buddy** processor

In-memory Double Checkpoint: Checkpoint

- Coordinated checkpoint
- Each object maintains 2 checkpoints:
  - On local processor
  - On a remote **buddy** processor
- Checkpoints are stored in memory

In-memory Double Checkpoint: Restart

- A *dummy* process is created to replace the crashed processor

In-memory Double Checkpoint: Restart

- A *dummy* process is created to replace the crashed processor
- New process starts recovery on other processors

In-memory Double Checkpoint: Restart

- A *dummy* process is created to replace the crashed processor
- New process starts recovery on other processors
- Other processors
    - Remove all objects
    - Use the buddy's checkpoint to recreate objects from the crashed processor
    - Recreate your own objects from their local copy of the checkpoint

In-memory Double Checkpoint: Pros and Cons

- Advantages:
  - Faster checkpoints than disk based
  - Reading checkpoints during recovery is also faster
  - Only one processor fetches checkpoint across the network

In-memory Double Checkpoint: Pros and Cons

- Advantages:
  - Faster checkpoints than disk based
  - Reading checkpoints during recovery is also faster
  - Only one processor fetches checkpoint across the network
- Drawbacks:
  - High memory overhead
  - All processors are rolled back even if one crashes
  - All the work since the last checkpoint is redone on all processors
  - Recovery time: Time between the crash and the previous checkpoint

Message logging

- Only processed **messages** affect the state of a processor
- After a crash, reprocess old messages to regain lost state

Message logging

- Only processed **messages** affect the state of a processor
- After a crash, reprocess old messages to regain lost state
- Messages are stored during execution
- After a crash, **only** crashed processors are rolled back
- Other processors resend their messages

Message logging

- Only processed **messages** affect the state of a processor
- After a crash, reprocess old messages to regain lost state
- Messages are stored during execution
- After a crash, **only** crashed processors are rolled back
- Other processors resend their messages
- **Caveat**: State of a processor is affected by the sequence of messages as well
  - Message processing sequence needs to be stored
  - Processors need to ignore messages they have already processed

Message logging: Challenges

- All the work of the crashed processor is redone by one processor
- Recovery time: **Same as checkpoint/restart**

Message logging: Challenges

- All the work of the crashed processor is redone by one processor
- Recovery time: **Same as checkpoint/restart**
- Most parallel applications are tightly coupled
- Other processors **have to wait** for the crashed processor to recover
- Fault free overhead is often high

Message logging: Objectives

- Fast recovery: **Faster** than time between the crash and the previous checkpoint
- Do not assume a stable storage
- Tolerate all single and most multiple processor faults
- **Low performance penalty** for the fault free case

Message logging: Our idea

- During restart distribute the work of the restarted processor among the waiting processors

Message logging: Our idea

- During restart distribute the work of the restarted processor among the waiting processors
- How can the work on one processor be divided ?

Message logging: Our idea

- During restart distribute the work of the restarted processor among the waiting processors
- How can the work on one processor be divided ?
- **Object based Processor Virtualization**

Message logging: Our idea

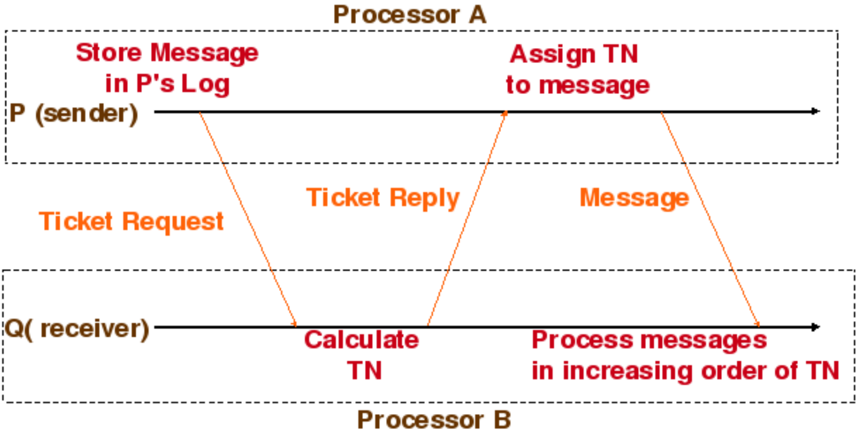
- During restart distribute the work of the restarted processor among the waiting processors
- How can the work on one processor be divided ?
- **Object based Processor Virtualization**
- Combine processor virtualization and message logging

Message logging: Our idea

- During restart distribute the work of the restarted processor among the waiting processors
- How can the work on one processor be divided ?
- **Object based Processor Virtualization**
- Combine processor virtualization and message logging
- Improves fault free performance as well

Message Logging and Virtualization

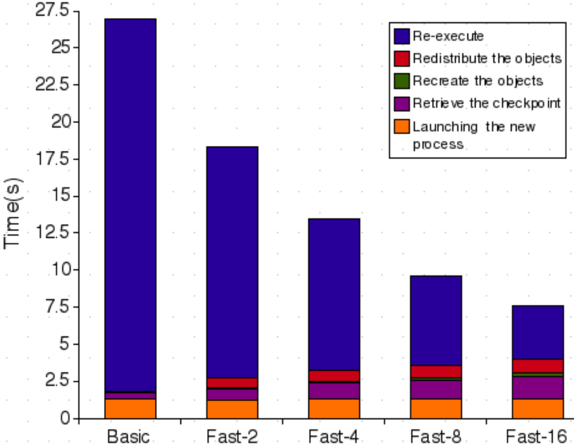
Virtual processors are the communicating entities



Modifying message logging to work with Virtualization

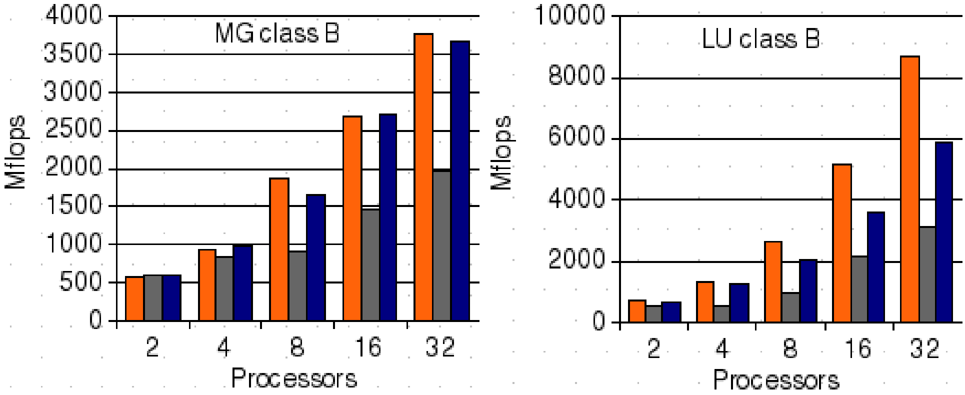
- When sender and receiver are on the same processor
- The receiver and message log are on the same processor
- If processor crashes not only does the log dissapear but more importantly its TN disappears
- Solved by storing some **meta-data** about such a message on a **buddy** processor
- During restart **redistribute** the VPs on the restarted processor among all processors

Fast Restart Performance



- 7 point stencil with 3D domain decomposition
- MPI program
- 16 processor run on Opterons with 1GB RAM and Gigabit
- Checkpoint every 30s
- Simulate fault after 27s
- 2-16 vps per processor

Fault free performance



AMPI AMPI-FT AMPI-FT with multiple VP

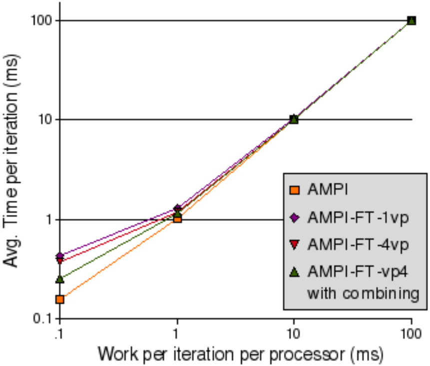
We got good performance for MG, SP and CG but bad for LU

Closer look at MG and LU

	MG on 8 processors		LU on 8 processors	
	AMPI	AMPI-FT	AMPI	AMPI-FT
Computation Time	68.18%	68.29%	86.56 %	87.81%
Idle Time	25.56%	22.75%	12.41 %	**48.28%**
Message Send	4.34%	5.01%	0.62 %	2.30 %
Ticket Request Send		4.54%		0.63%
Ticket Send		1.37%		1.01%
Local Message		2.10%		0.00%
Total	98.08 %	104.06%	99.59 %	140.03 %

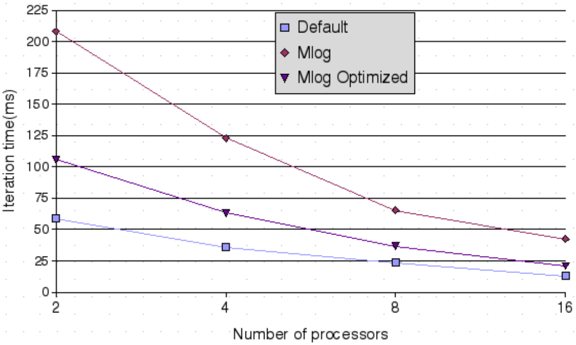
Lower granularity of LU increases **Idle time**

Optimizations



- Synthetic benchmark
- High overhead for low granularity
- Increasing vps helps
- 100 *us* case still pretty **high**
- **Combine** protocol messages
- Reduces cpu overhead
- Alleviates network congestion

Optimizations: Evaluation



- Real application: **leanMD**
- **BUTANE** molecular system is very small
- 16 processor test cluster
- Iteration time 13ms
- A message every 45μs on each proc
- **WORST CASE**

Future Work

- Load balancing with message logging
- Remove the need for extra processors

Pro-active Fault Tolerance

- Modern hardware can be used to **predict** failures
- Runtime system responds to warning
  - Low response time
  - No extra processors required
  - Efficiency loss should be proportional to loss in computational power

Processor Evacuation

- Migrate Charm++ VPs off processor
- Point to Point messaging should continue to work correctly
- Collective operations should continue to work
- Rewire reduction tree around a warned processor
- Can deal with multiple simultaneous failures
- Load balance after an evacuation

Summary

- Charm++/AMPI provides multiple fault tolerance protocols
- Disk based Checkpoint/Restart
- In memory Checkpoint/Restart
- Proactive fault tolerance

Summary

- Charm++/AMPI provides multiple fault tolerance protocols
- Disk based Checkpoint/Restart
- In memory Checkpoint/Restart
- Proactive fault tolerance
- Message logging with fast recovery **under development**