
Transparent Process-level Fault Tolerance for MPI: Challenges and Solutions

Frank Mueller

Dept. of Computer Science
North Carolina State University



NC STATE UNIVERSITY
Department of Computer Science

Problem Statement

- **Trend in HPC**: high end systems (tens of) thousands of processors
 - High probability of node failure
MTBF becomes shorter
 - CPU/memory/IO failures

System	# CPUs	MTBF/I, see [20]
ASCI Q	8192	6.5hrs
ASCI WHITE	8192	5/40 hrs
PSC Lemieux	3016	9.7hrs
Google	15000	20 reboots/day

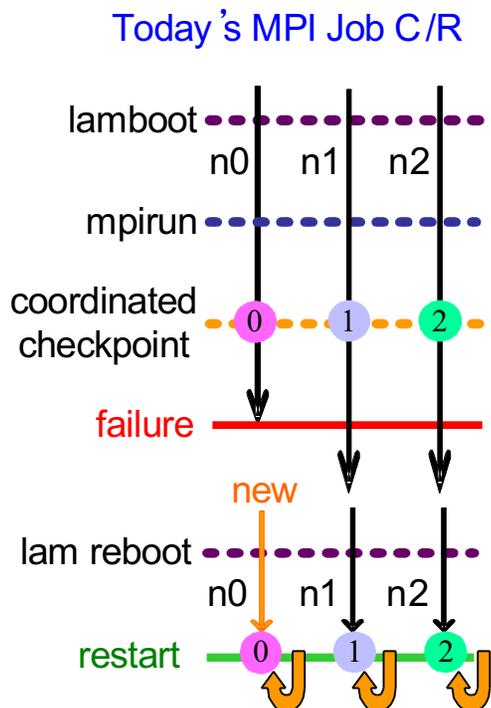
- **MPI** widely used for scientific apps
 - Problem with MPI: no recovery from faults in the standard
- Currently FT exist but...
 - not scalable
 - right level? (app/process/OS)
 - mostly reactive: process checkpoint/restart [used in DOE labs]
 - restart entire job → **inefficient** if only one/few node(s) fail
 - overhead: re-execute some of prior work
 - issues: checkpoint at what frequency?
 - 100 hr job → +150 hrs for chkpt on Pflop machine (w/o failure) [Philp'05]

Overview

1. Scalable network overlay (ICS'06)
 - track live nodes
 - group communication
2. Reactive fault tolerance (IPDPS'07)
 - Process virtualization
 - Job pause mechanism
3. Proactive fault tolerance (ICS'07, SC'08)
 - Process virtualization
 - health monitoring
 - live migration
 - back migration
4. Ongoing work
5. Discussion

(2) Job Pause (Process Virtualization)

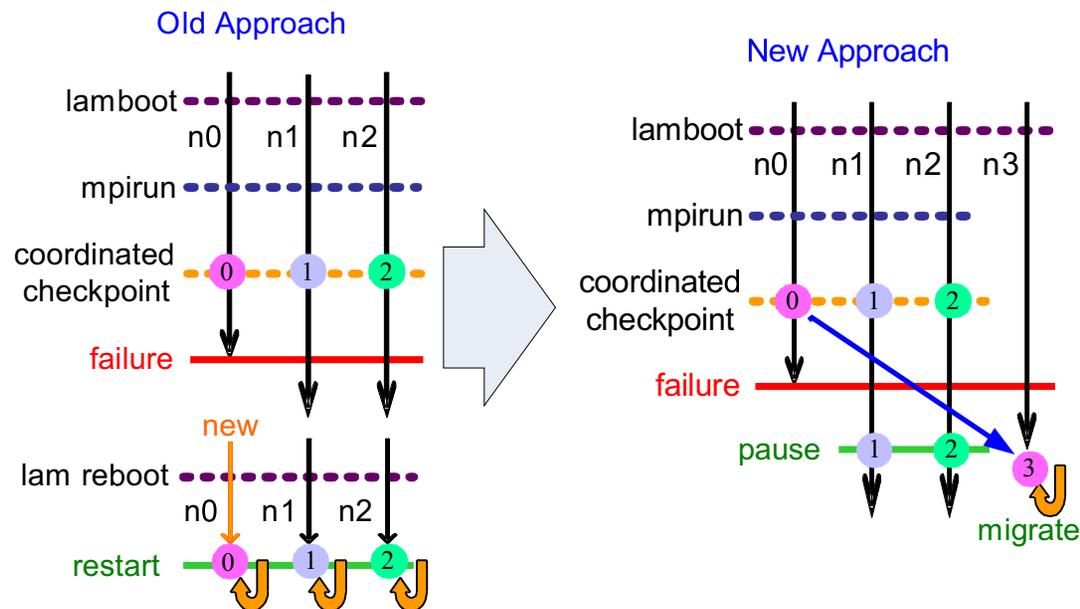
- Trends in HPC: high end systems with thousands of processors
 - Increased probability of node failure: MTTF becomes shorter
- MPI widely accepted in scientific computing
 - But no fault recovery method in MPI standard



- Extensions to MPI for FT exist but...
 - Cannot dynamically add/delete nodes transparently at runtime
 - Must restart
 - MPI runtime
 - Entire job
 - Inefficient if only one/few node(s) fail
 - Staging overhead
 - Requeuing penalty

Our Solution: Job-pause Mechanism

- Integrate *group communication*
 - Add/delete nodes
 - Detect node failures automatically
- *Processes on live nodes remain active* (roll back to last checkpoint)
- *Only processes on failed nodes dynamically replaced by spares*
 - resumed from the last checkpoint

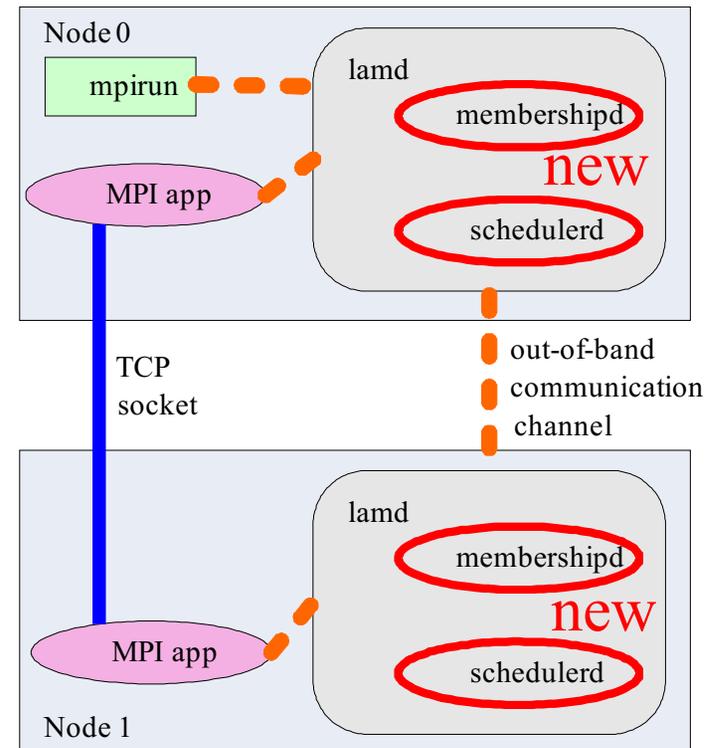


Hence:

- *no restart of entire job*
- *no staging overhead*
- *no job requeue penalty*
- *no MPI runtime restart*

Our Design & Implementation – LAM/MPI

- Decentralized scalable Membership and failure detector (ICS'06)
 - Radix tree → scalability
 - dynamically detects node failures
 - **NEW**: Integrated into lamd
- **NEW**: Decentralized scheduler
 - Integrated into lamd
 - Periodic coordinated checkpointing
 - Node failure → trigger
 1. process migration (failed nodes)
 2. job-pause (operational nodes)

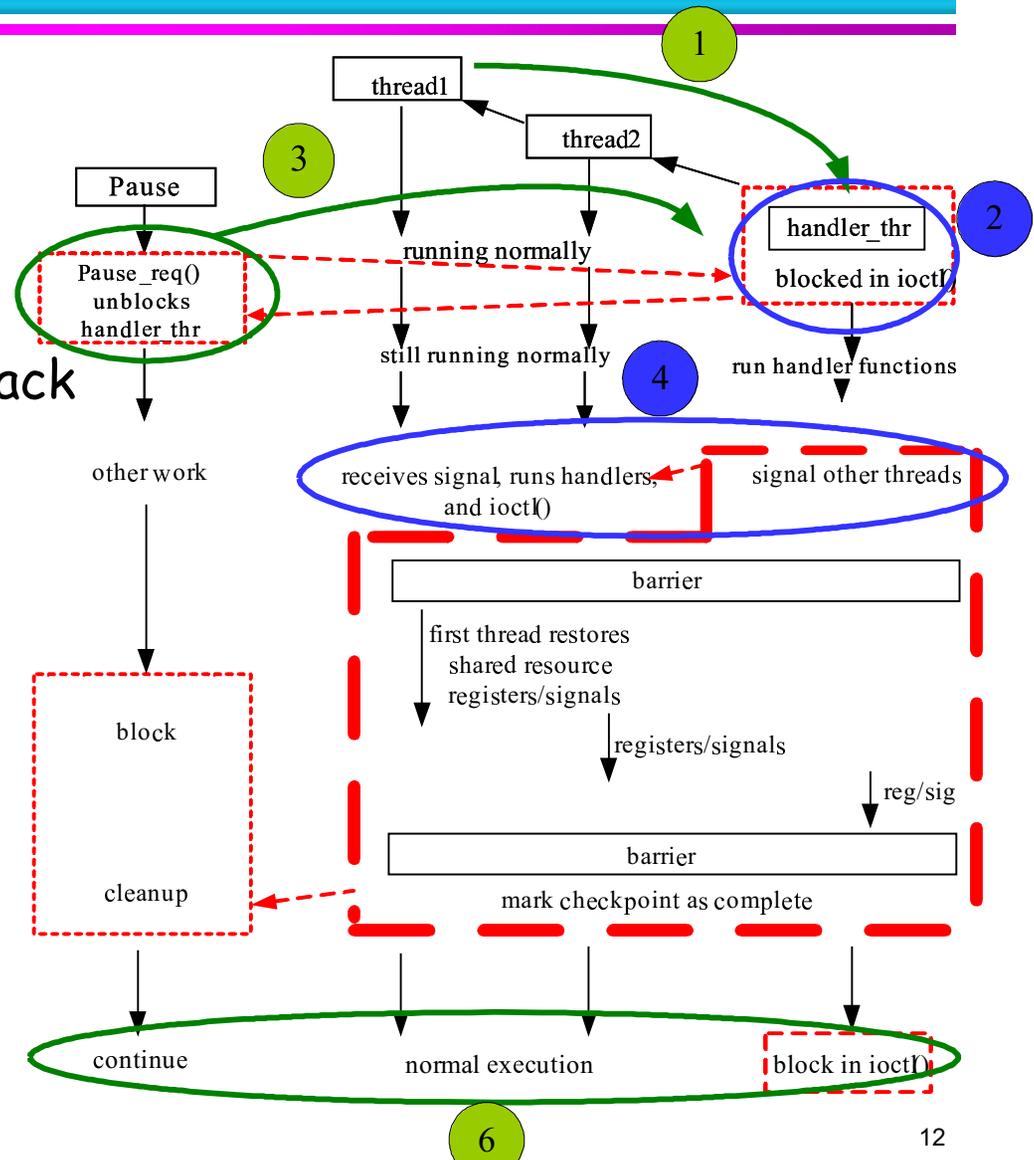


New Job Pause Mechanism - BLCR

Call-back kernel thread:
coordinates user command
process and app. process

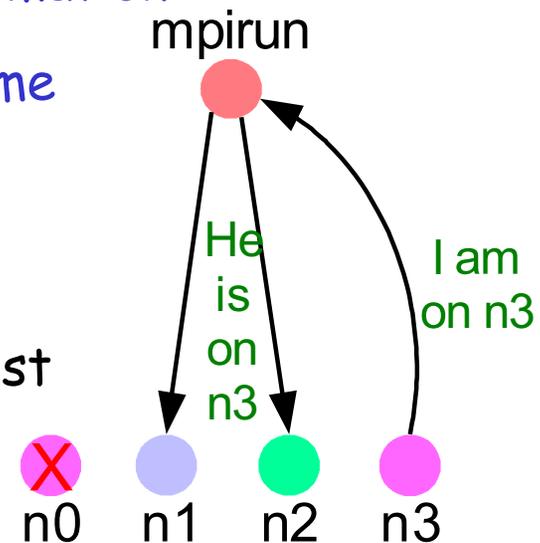
(In kernel: dashed lines/boxes)

1. app registers threaded callback
→ spawns callback thread
2. thread blocks in kernel
3. pause utility calls ioctl(),
unblocks callback thread
4. All threads complete
callbacks & enter kernel
5. **New: All threads restore
part of their states**
6. Run regular application
code from restored state



Process Migration – LAM/MPI

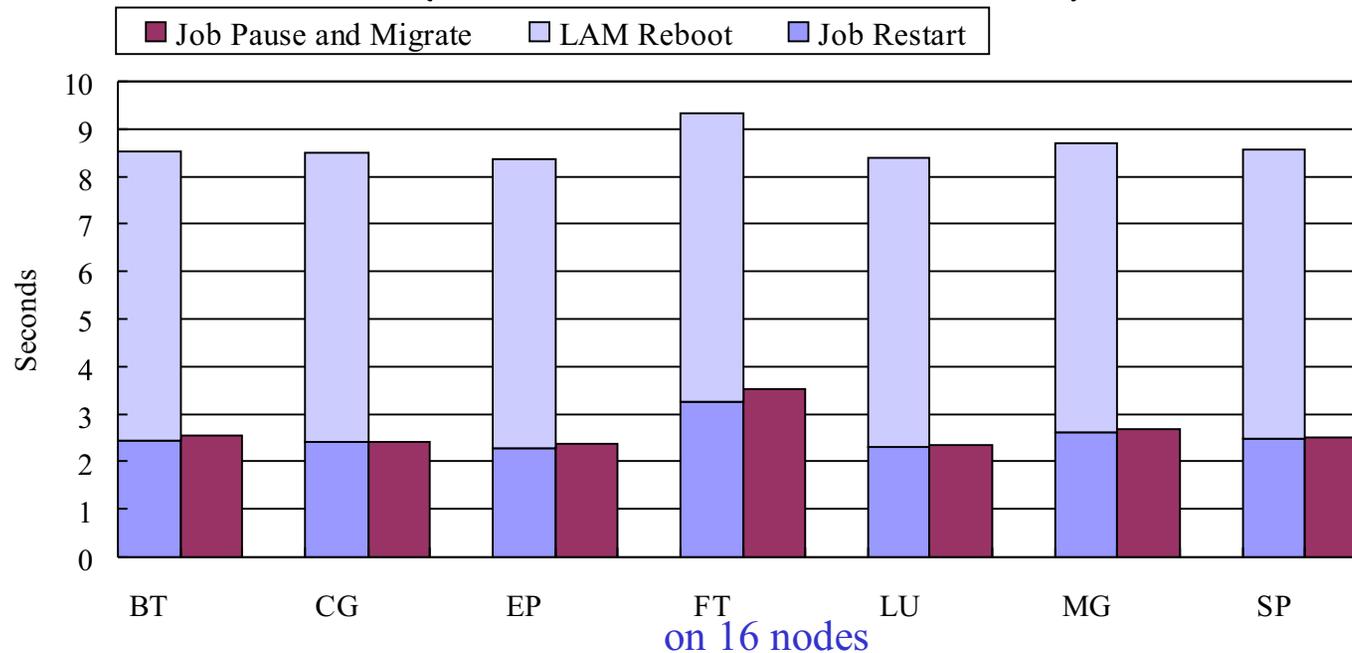
- Change addressing information of migrated process
 - in process itself
 - in all other processes
- Use node id (not IP) for addressing information
- Update addressing information at run time
 1. Migrated process tells coordinator (mpirun) about its new location
 2. Coordinator broadcasts new location
 3. All processes update their process list



- No change to BLCR for Process Migration

Job Migration Overhead

- 16 nodes, Lam/MPI + BLCR w/ our extensions
- NAS PB, Class C (IS omitted, run too short)



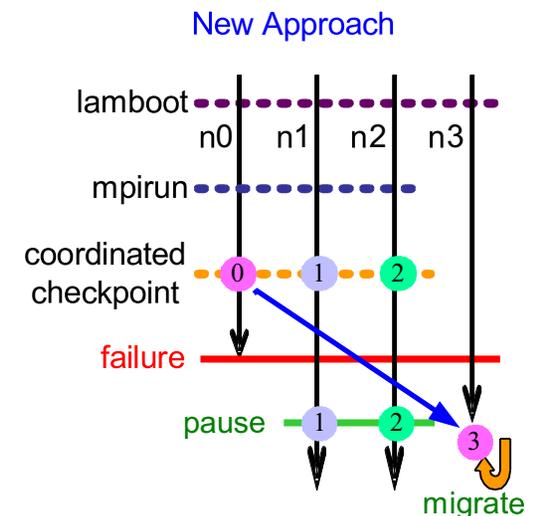
69.6% < job restart + lam reboot

- NO LAM Reboot
- No requeue penalty
- Transparent continuation of exec
- Less staging overhead

Contribution (2)

Job-Pause for fault tolerance in HPC

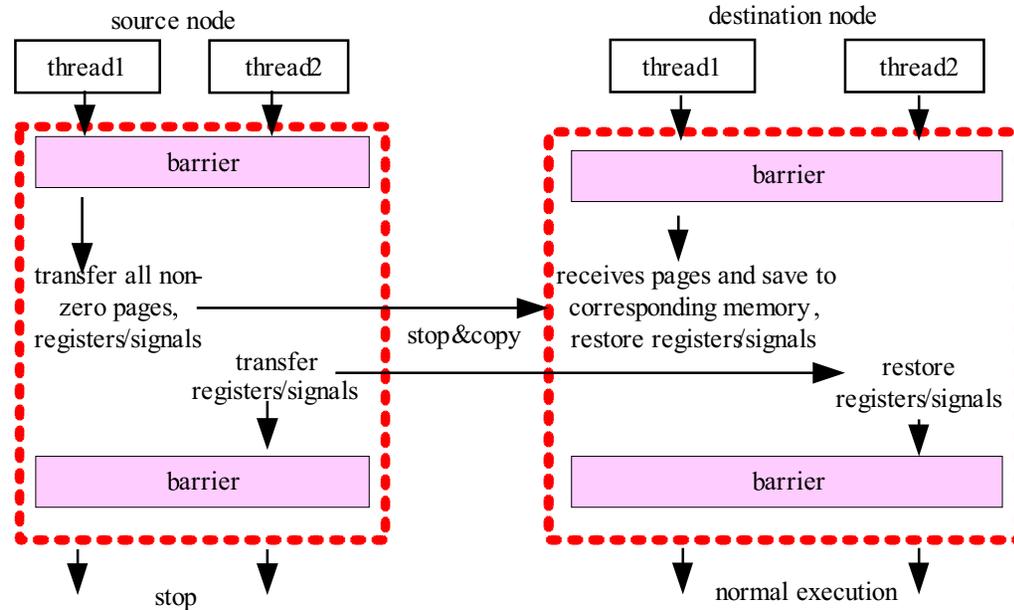
- Design generic for any MPI implementation / process C/R
- Implemented over LAM-MPI w/ BLCR
- Decentralized P2P scalable membership protocol & scheduler
- High-performance job-pause for operational nodes
- Process migration for failed nodes
- **Completely transparent, fast ~ 10 sec.**
- **Low overhead: 69.6% < job restart + lam reboot**
 - No job requeue overhead
 - Less staging cost
 - No LAM Reboot
- Suitable for proactive fault tolerance with diskless migration



(3) Proactive Process-Level Migration

- OS level
 - Higher-level encapsulation
 - More elegant?
- Process level
 - Less data / baggage
 - More complex?
 - Cheaper?
- Implemented over
 - BLCR extensions
 - Kernel enhancements (dirty bit tracking in PTEs)
 - Add'l LAM/MPI support

Process Migration w/o Precopy - BLCR

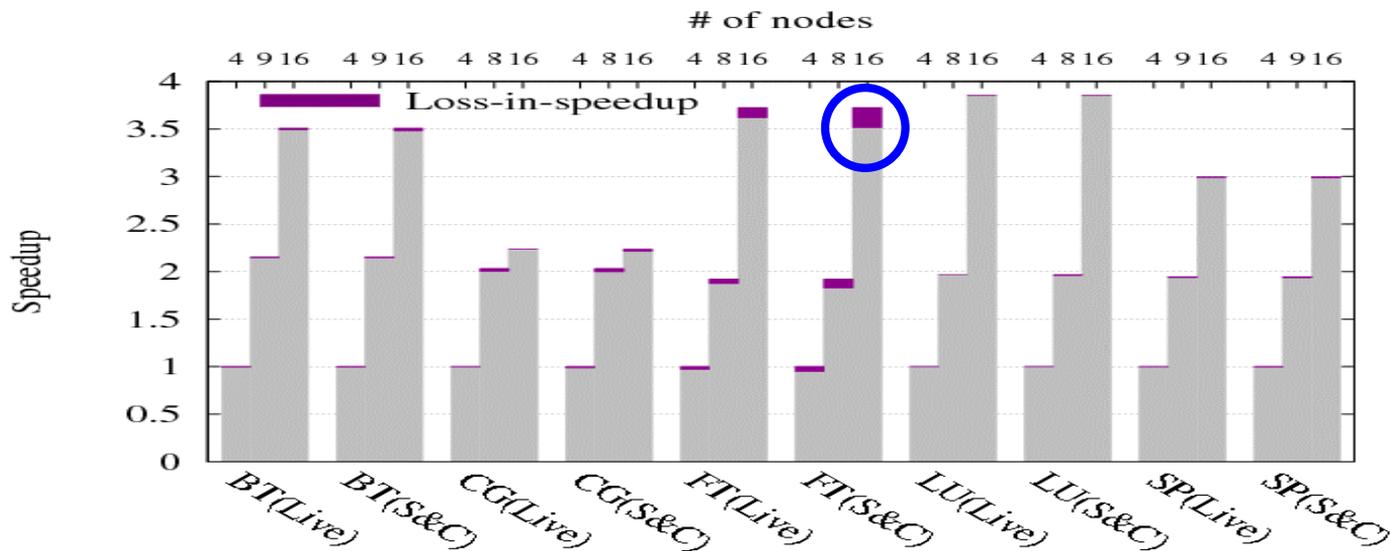


Live vs. Frozen migration (also for precopy termination conditions):

1. **Thresholds**, e.g., temperature watermarks, memory/difference / overhead thresholds
2. **Available network bandwidth** determined by dynamic monitoring
3. **Size of write set**

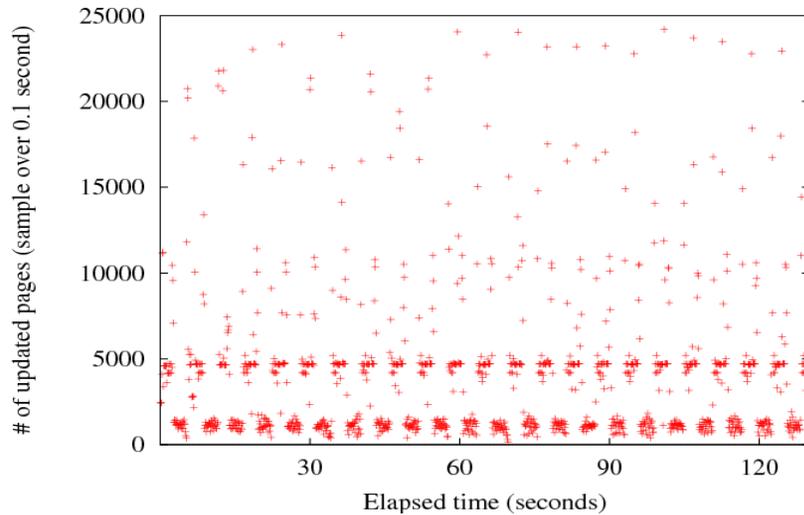
Future work: heuristic algorithm based on these conditions

Speedup

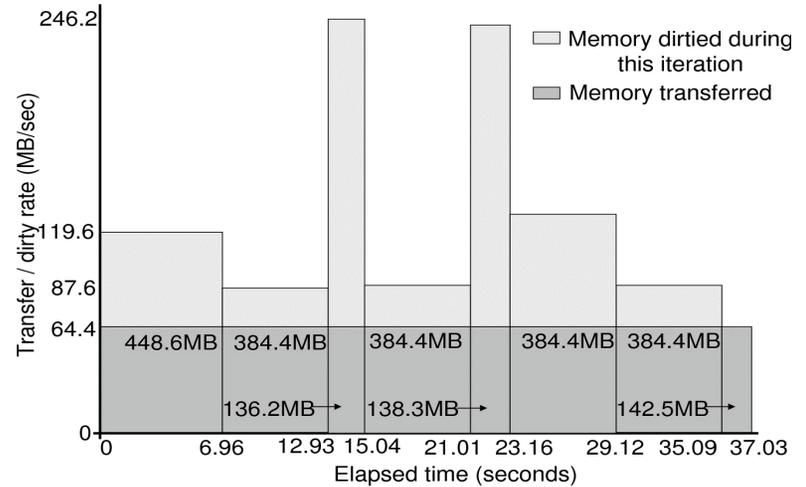


- Normalized to the wall-clock time on 4 nodes for NPB Class C
- Gray portion of the bars: the execution time w/ one migration
- Aggregate value of gray+purple portion: execution time w/o migration
- Purple portion of the bars: the loss in speedup due to migration
- FT 0.21 lost-in-speedup: relatively large overhead (8.5 sec) vs. short run time (150 sec)
- Limit of migration overhead: proportionate to memory footprint, limited by system hardware

Page Access Pattern & Iterative Migration



Page access pattern of FT



Iterative live migration of FT

- Page write patterns are in accord with aggregate amount of transferred memory
- FT: 138/384MB -> 1200/4600 pages/.1 second

Process-level vs. Xen Virtualization Migration

- **Xen virtualization solution:** 14-24 seconds for live migration, 13-14 seconds for frozen migration
 - Including a 13 seconds minimum overhead to transfer the entire memory image of the inactive guest VM (rather than transferring a subset of the OS image) for the transparency
 - 13-24 seconds of prior warning to successfully trigger live process migration
- **Process-level:** 2.6-6.5 seconds for live migration, 1-1.9 seconds for frozen migration
 - 1-6.5 seconds of prior warning

Proactive FT Complements Reactive FT

$$T_c = \sqrt{2 \times T_s \times T_f} \quad [\text{J.W.Young Commun. ACM '74}]$$

T_c: time interval between checkpoints

T_s: time to save checkpoint information (mean T_s for BT/CG/FT/LU/SP Class C on 4/8/16 nodes is 23 seconds)

T_f: MTBF, 1.25hrs [I.Philp HPCRI'05]

$$T_c = \sqrt{2 \times 23 \times (1.25 \times 60 \times 60)} = 455$$

70% faults [R.Sahoo et.al KDD '03] can be predicted and handled proactively

$$T_c = \sqrt{2 \times 23 \times (1.25 / (1 - 0.7) \times 60 \times 60)} = 831$$

Cut the number of chkpts in half: 455→831 seconds

Future work: use 1. better fault model 2. T_s/T_f on bigger cluster to measure the complementation

Incremental Checkpointing

- Diff since last chkpt
- BLCR enhancement
- Reuse dirty bit at PTE
- Hybrid: 1 full, k incr. Chkpts
- Model savings [Nakisanehaboon et al.]

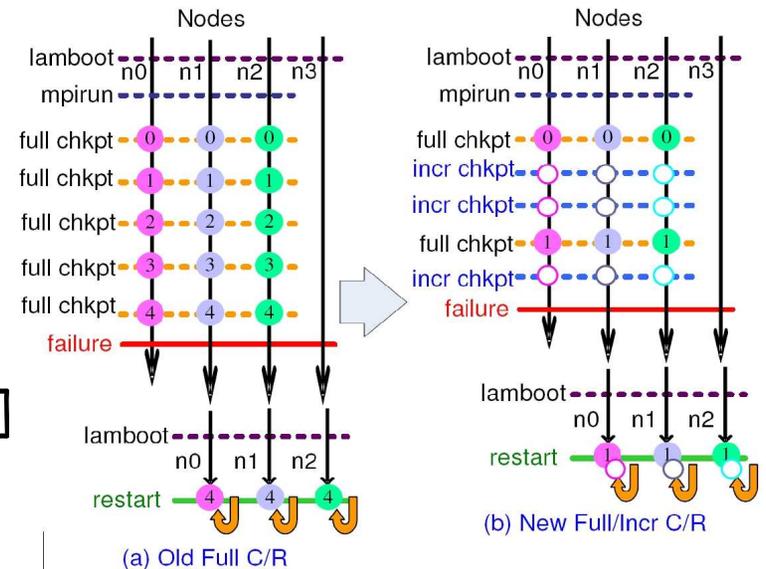
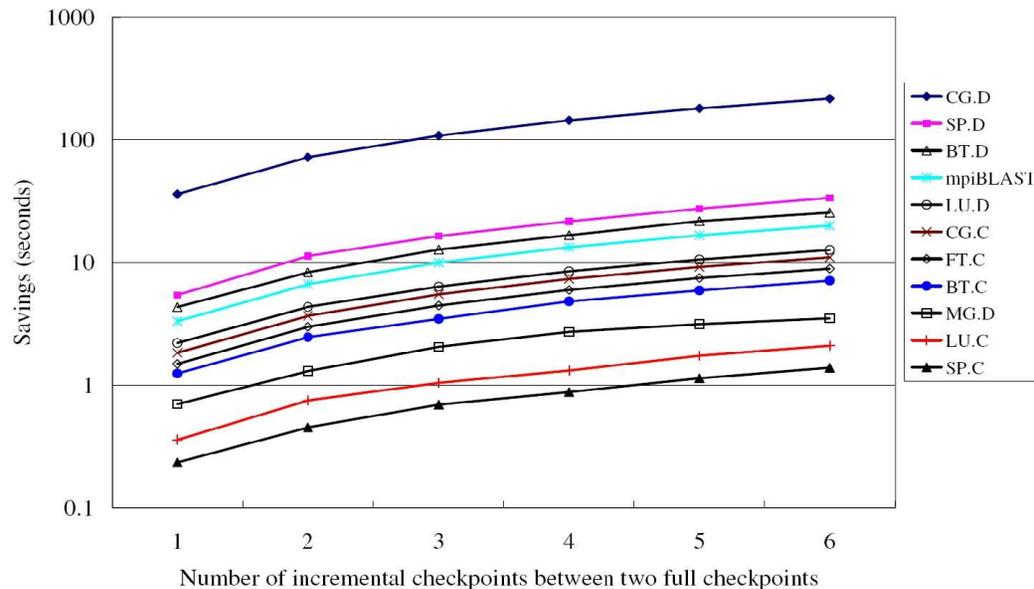
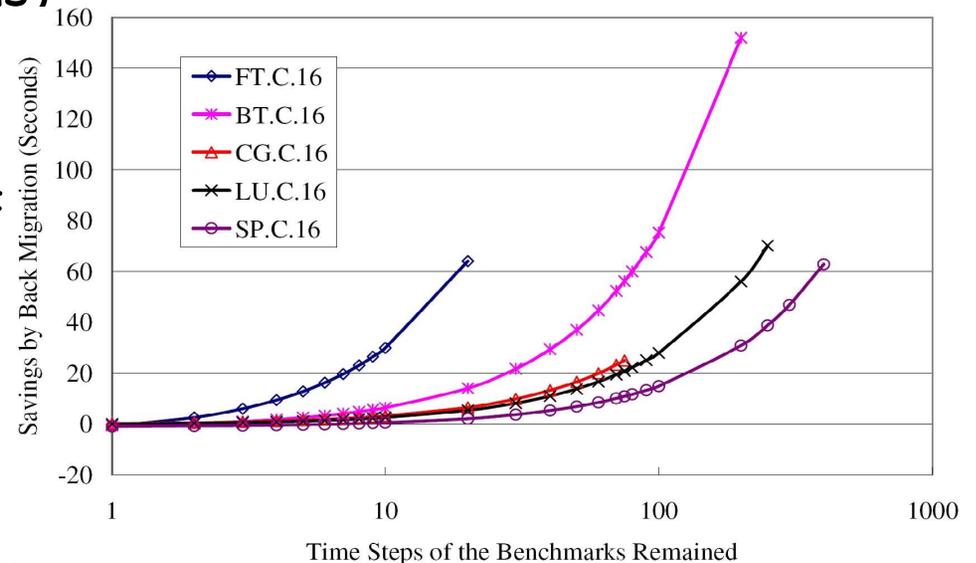


Fig. 1: Hybrid Full/Incremental C/R Mechanism vs. Full C/R

Back Migration

- Node fails → migrate
- Node recovers → migrate back
- Why?
 - Heterogeneous nodes
 - MPI task sharing on nodes
 - Increased hop counts (torus)
- Experiments
 - slower spare nodes:
CPU freq. nearly cut in half
- Benefits
 - Much less chkpt overhead
 - Reduced IO bandwidth



Contributions (3)

- Novel, proactive FT scheme w/ process virtualization
 - Provides transparent & automatic FT for arbitrary MPI apps
 - Less overhead than reactive
 - Also complements reactive → lower checkpoint frequency
 - Process-level: $\frac{1}{2}$ overhead of OS-level
 - $\frac{1}{2}$ the chkpts when 70% faults handled proactively
- Incr. Chkpt → less overhead and I/O pressure
- Back migration → original balance / performance

(4) Current BLCR Efforts

- Collaboration w/ Paul Hargrove (LBL)
- Upcoming FT features in BLCR:
 - Job Pause/Rollback → merged into V0.8.0, next BLCR release
 - Incremental checkpoints → in progress
 - Dirty vs. write-protect bits
 - Challenges: copy-on-write (fork), mprotect, map/unmap
 - Assess overheads of alternatives
 - Differential checkpoints → next release
 - Region-delimited checkpoints (semi-transparent) → next rel.

Contributions

1. Scalable network overlay (ICS'06)

- track live nodes
- group communication

2. Reactive fault tolerance (IPDPS'07)

- Process virtualization
- Job pause mechanism

3. Proactive fault tolerance (ICS'07, SC'08)

- OS/process virtualization
- health monitoring
- live migration
- back migration

Discussion

- Need studies on potential to detect health deterioration
- Need CS cluster resources w/ root access for scaling → virtualization
 - Later production deployment
- Job scheduler support for FT
 - Spare node pooling
- Chkpt PFS and I/O requirements
- Anomaly detection / threshold derivation
 - In progress: transparent fault detector for MPI

Discussion (cont.)

- MPI wish list:
 - Coordinated checkpointing:
 - `MPI_QUIESCE_START/END(comm, info)` → drain comm. Qs
 - Low-level control, MPI integrated
 - <https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/Quiescence>
 - `MPI_CHKPT(bool force)` → optional trigger
 - Higher-level abstraction
 - User/runtime defined, opt. MPI integrated
 - Just like `MVAPICH2_Sync_Checkpoint()`;
- Checkpt in no more than `m` / every `m` minutes
 - `MPI_NEXT_CHKPT(struct timespec abstime)`
 - `MPI_CHKPT_FREQ(struct timespec reltime)`

Acknowledgement

Supp. in part by DOE/NFS grants, Humboldt fellowship

DOE DE-FG02-05ER25664 and DE-FG02-08ER25837, NFS CCR-0237570, CNS-0410203, CCF-0429653

- NCSU students
 - Jyothish Varma
 - Arun Nagarajan
 - Chao Wang (ORNL)
 - Manav Vasavada



- ORNL collaborators
 - Christian Engelmann
 - Stephen L. Scott



- LBL collaborators
 - Paul Hargrove

