

# MPI/FT: A model-based approach for low-overhead fault-tolerance

---

**Anthony Skjellum\*<sup>\$</sup>, Rajanikanth Batchu<sup>\$</sup>,  
Yoginder Dandass<sup>\$</sup>, Murali Beddhu\***

**\*MPI Software Technology**

**<sup>\$</sup>Mississippi State University**

June, 2002



# Outline

---

- Introduction
- Background
- Model-based approach
- Usage and implementation
- Results
- Future work
- Conclusions

# Outline

- Introduction
  - Clusters
  - MPI
  - Motivation
- Background
- Model-based approach
- Usage and implementation
- Results
- Future work
- Conclusions

# Introduction

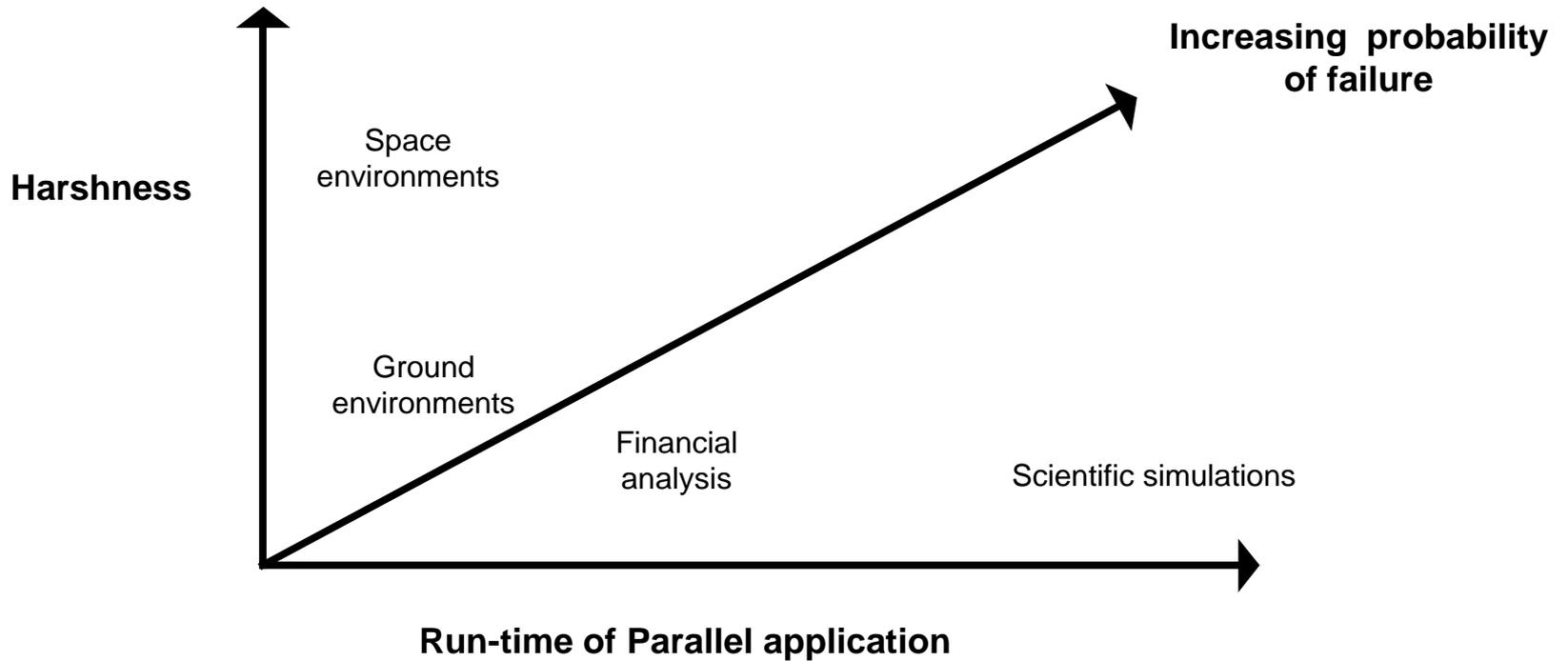
- Clusters
  - Compute nodes + High speed SANs
  - Widespread usage
  - Availability and reliability
  - Efficacy for parallel computing
- MPI (MPI Forum 1995)
  - Standard for message passing interface
  - Provides services and abstractions
  - Many implementations
    - Commercial
    - Academic



# Motivation

- Popularity of clusters + MPI
  - Wide deployment
  - Usage of clusters in harsh environments (external faults)
- High costs of failure
  - Control systems
  - Financial/e-commerce
  - Web
- Insufficient reliability
  - Shortcomings in the MPI standard
  - Inadequate features in other MPI implementations.

# Motivation, II.



# Motivation, III.

---

“There is a need for better way than simple restart of MPI applications upon failure.”

# Outline

- Introduction
- **Background**
  - MPI standard
  - Lessons learned from other research efforts
- Model-based approach
- Usage and implementation
- ...

# MPI Shortcomings

- MPI (MPI Forum 1995) standard's limitations
  - Main goals: performance, portability
  - Static process model
  - No comprehensive measures for reliability
- Detection
  - Limited detection
  - Signaled through error codes
- Recovery and Error Handling
  - geared towards graceful termination,
  - cannot always be invoked (fatal errors)
  - coarse grained (per communicator)
  - Inadequate



# Lessons Learned

---

- Management of redundancy is essential
- “One size fits all” is not true
- Applications have differing requirements
- Transparency results in overhead
- Design/decision process of middleware is complex
  - Contrasting inputs
  - Contrasting requirements



# Outline

- Introduction
- Background
- Model-based approach
  - Research approach
  - Other features
  - Models I and II
- Usage and implementation
- Results
- Future work
- Conclusions

# Model-based Research Approach

---

- Our main goal is low-overhead fault-tolerance
- All other efforts treat applications as “blackbox” and provide same services
  - User transparent checkpointing
  - Process level checkpoints
- Each application has features that help in achieving fault-detection and recovery

# Model-based Research Approach, II.

---

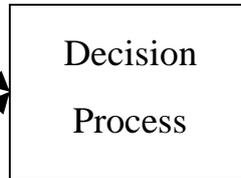
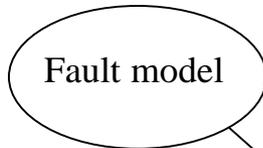
- Identify features that help in effective realization of fault-tolerance
  - Communication topology (master/slave)
  - Program structure (communication, computation loops)
  - Redundancy requirements
  - Support from underlying layers

# Model-based Research Approach, III.

- Classify models based on combination of features and requirements
  - Application models, models of execution
  - Predetermined models, most commonly used
  - Based on application features and requirement
  - Adaptable
  - Provide abstraction from complex decisions (next slide)
- Take all the principles, and incorporate selectively into existing middleware:
  - MPI/FT™ is derived from MPI/Pro™.

# Complexity of Decision Process

- Types of faults
- Occurrence rate
- Manifestations



- Communication patterns: (master/slave) and simple SPMD
- Resource requirements
- ABFT

- Resources, memory available
- Fault-tolerance in hardware (lead hardened)
- Characteristics



- Predictability, recovery time
- Performance, fault free overhead
- FT features, availability, and reliability requirements



# Other Features of MPI/FT

- Internal robustness
  - Features and techniques to mask faults in middleware space
  - Coordinator and Self Checking Thread (SCT) provide detection and internal robustness
  - SCT
    - Various levels of portability
    - Adaptable to various fault rates and fault types

# Other Features, II.

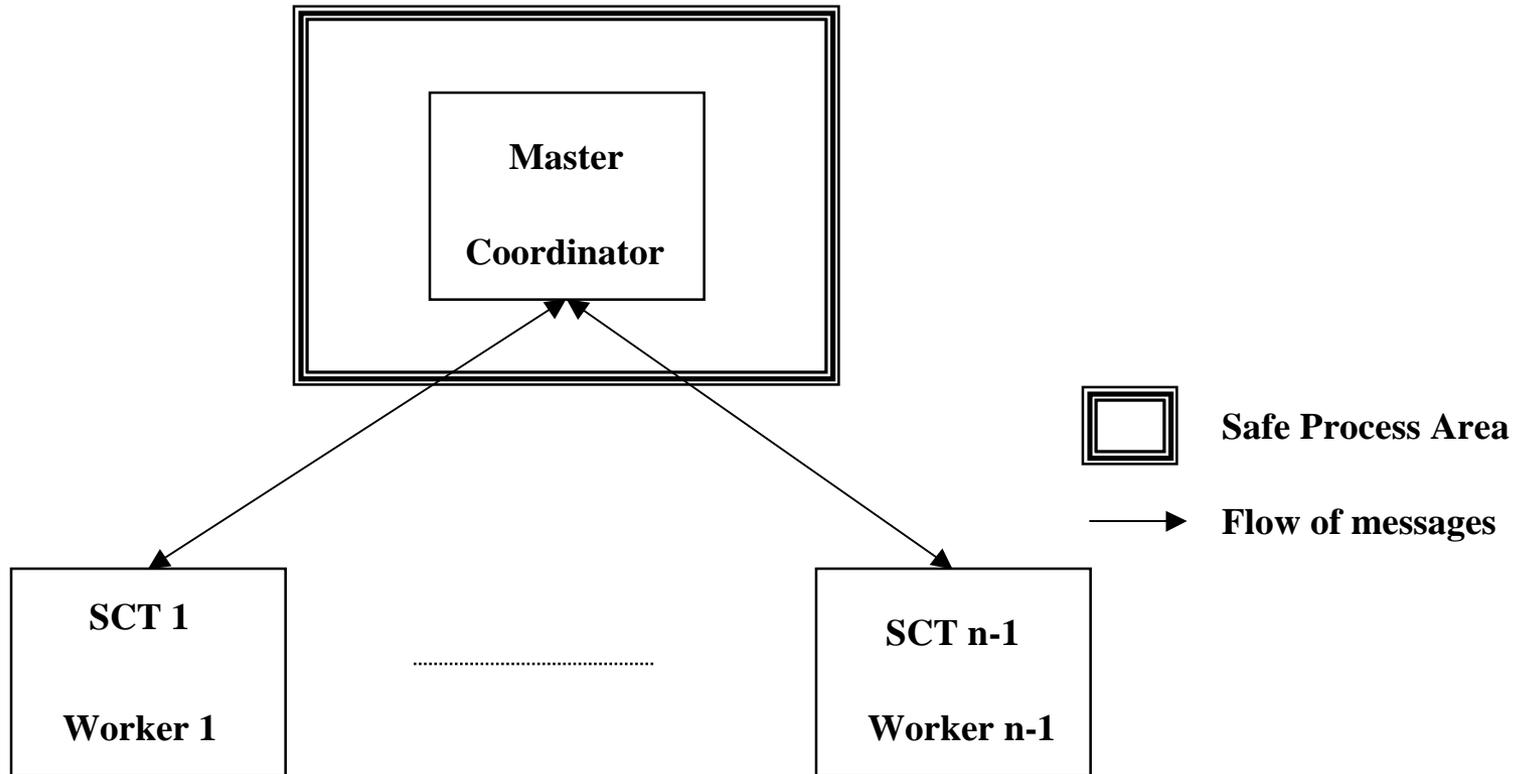
- Parallel NMR
  - MPI process level redundancy
  - Part of our current study/research
- Checkpointing
  - User Aware and user initiated
  - Functions to select data and recovery functions
- ABFT support
  - application initiated kill and recovery

# Model I.

- Features
  - Simple master/slave work model
  - Virtual star topology
  - Master process does not get faults
- FT services
  - Detection of slave deaths
  - Death of a single slave doesn't disable application
  - API for notification and recovery
- Examples
  - Pmandel, povray, many others in this category



# Topology for Model I.

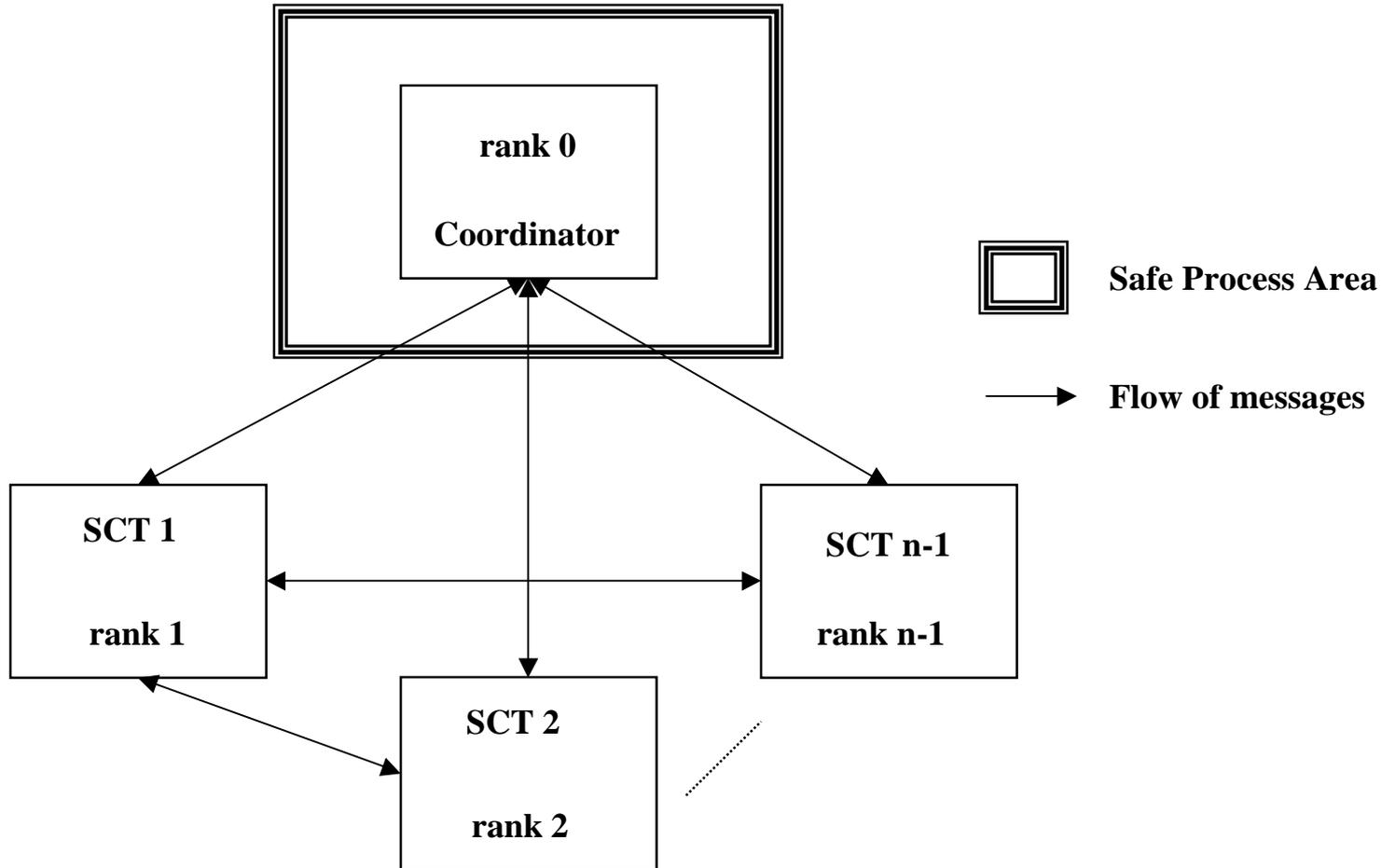


# Model II.

- Features
  - Complex SPMD
  - Virtual all-to-all topology
  - Programs written in iterative loops (communication + computation)
- FT services
  - Detection of slave deaths
  - User aware, middleware assisted checkpoint
- Examples
  - Game of life
  - Discrete simulation



# Topology for Model II.



# Outline

- Introduction
- Background
- Model-based approach
- Usage and implementation
  - User steps
    - Typical steps,
    - Code changes
  - Implementation
    - Steps in FT
    - Recovery in Model-I, Model-II
- Results
- Future work
- Conclusions

# Typical Usage for MPI Application

- Understand the problem and identify the parallelism in the program.
- Decide on a program process model (master/slave, SPMD, hybrid, etc.) with deterministic communication patterns.
- Implement (code) on middleware.
- Launch the application with required number of processes. In the absence of external faults the application will successfully complete. A typical program launch could be

```
$>mpirun -np 4 -mach_file myMachfile  
myParallelapp param1 param2
```

# Usage for MPI/FT

- Understand the problem and identify the parallelism in the program.
- Decide on a program process model (master/slave, SPMD, hybrid, etc.) with deterministic communication patterns.
- Identify impacts of faults and region in code for notification and recovery.
- Decide on application execution model.
- Implement (code) on a middleware and introduce FT specific code.
- Launch the application with required number of processes. Additionally static spare processes should be launched. A typical program launch with fault-tolerance could be

```
$> mpiftrun -np 4 -sp 2 -mach_file myMachfile  
-ftparam1 val1 -ftparam2 val2  
myFTParallelapp param1 param2
```



# Code Changes

- User applications need to be modified in order to utilize FT features
- Simple and effective API for each model
- Code changes
  - Do not alter existing structure
  - Constant number of new FT API required irrespective of original code length
- Model I : GetDeadRanks, RecoverRank
- Model II: Model-I API + RecoveryPoint, ChkptDo, ChkptRecover

# Model I. API description

INT

MPIFT\_GetDeadRanks (

OUT INT \*deadcount,

OUT INT \*deadarrayranks,

IN INT array\_size)

- Provides dead rank information to user thread
- Can be invoked only in the master (rank 0) process.



# Model I. API description, II.

INT

```
MPIFT_RecoverRank (  
    IN INT RankToRecover  
);
```

- Initiates recovery of a particular dead rank
- Can be invoked only in the master (rank 0) process.



# Model I code changes

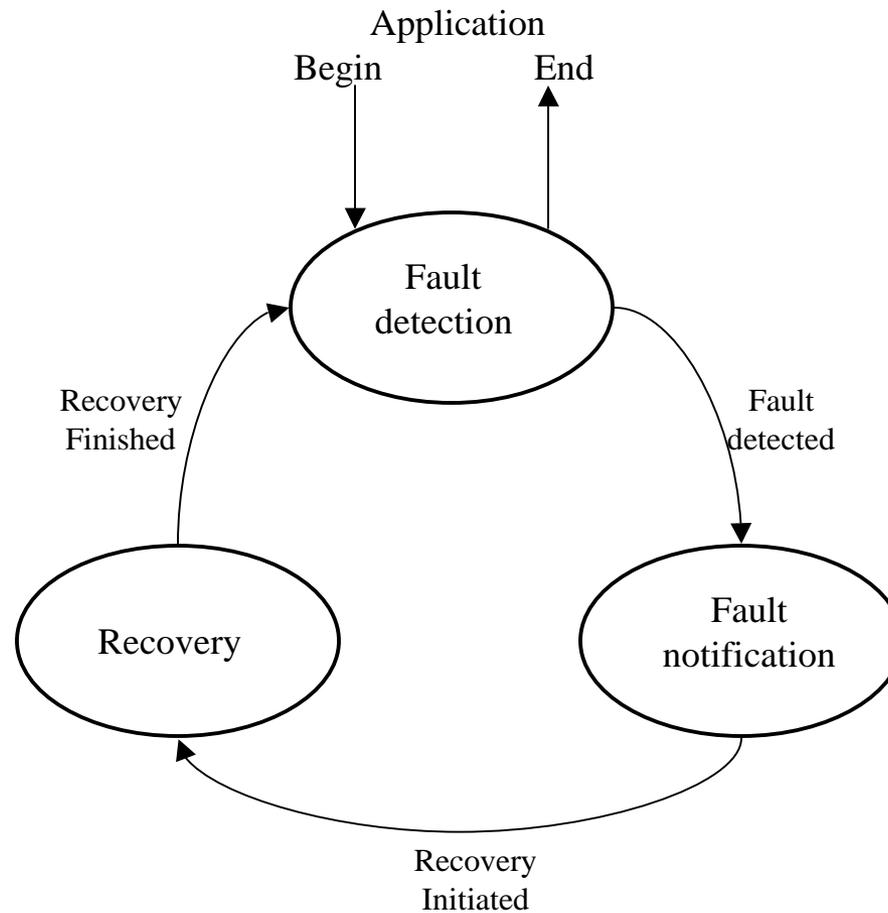
```
Create job array;
Send_Init(); Send_jobs();

While (! Jobs done){
    Recv_results();
    Send_jobs ();
};
```

```
Create job array;
Send_Init();Send_jobs();
Save_jobs () ;
While (! Jobs done){
    MPIFT_GetDeadRanks (&deadcount, &deadarray,
arraysize);
if (deadcount >1){
    for (I = 0.. Deadcount-1){
        Recover_job ();
        MPIFT_RecoverRank (deadarray[I]);
        Send_Init(); Send_jobs(); Save_jobs ();
    else{
        Recv_Results();
        Delete_jobs ();
        Send_jobs();
    }
};
```



# Steps in FT, I.



# Steps in FT, II.

- Detection
  - FT Threads: Coordinator and SCT
  - External heartbeats
    - Between ranks
  - Internal heartbeats
    - Between SCT and MPI/FT send and receive progress threads
  - User configurable frequency and timeouts
    - Support for passing them through command line (mpiftrun)

# Steps in FT, III.

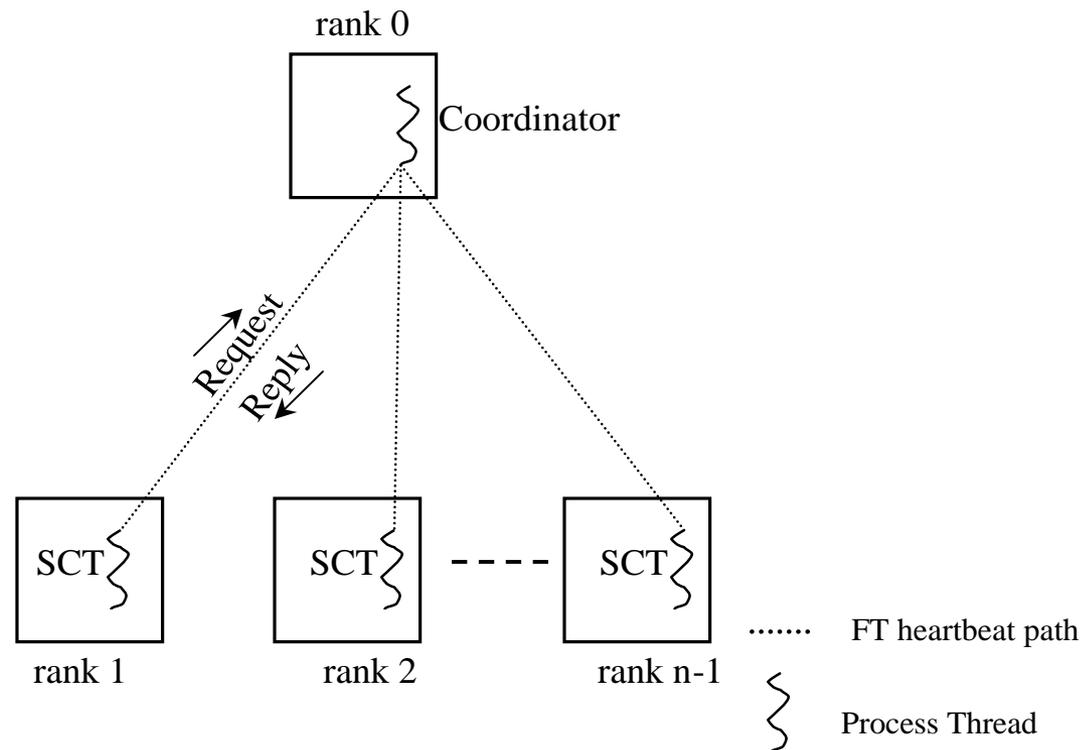
- Fault notification to Application
  - Model I polling based
  - Model II polling based / evaluating interrupt based
- Recovery
  - Model I:
    - Only in context of master (rank 0)
  - Model II:
    - All alive ranks must be involved in recovery
    - Checkpointing-based application recovery

# Fault Detection using External Heartbeats

---

- External Heartbeats
  - Between coordinator thread (master node) and SCTs (worker nodes)
  - Uses additional TCP connections between the master and workers to send special reply/request packets
  - User configurable frequencies and timeout factors

# Fault Detection using External Heartbeats, II.

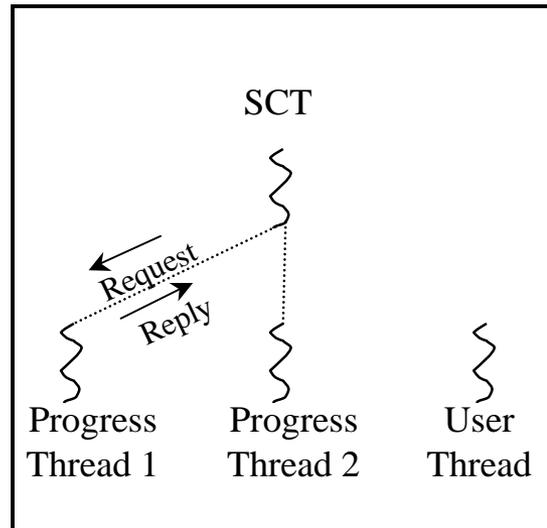


# Fault Detection using Internal Heartbeats

---

- Between SCT and progress threads (send and recv)
  - Shared memory area for heartbeat counters, and flags
- Progress threads inform SCT about “expected busy time” when sending long messages
  - SCT will allow the expected time to expire before initiating heartbeat requests

# Fault Detection using Internal Heartbeats, II.



rank x

..... FT heartbeat path

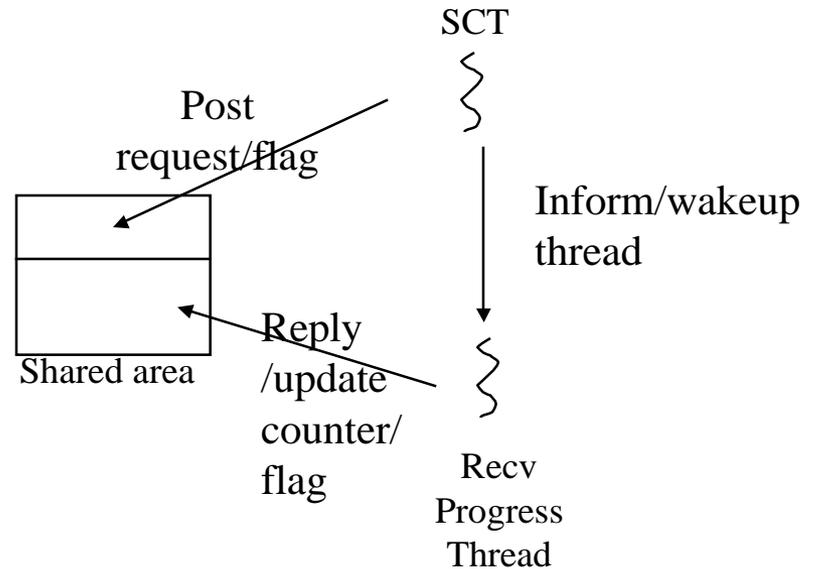


Process Thread



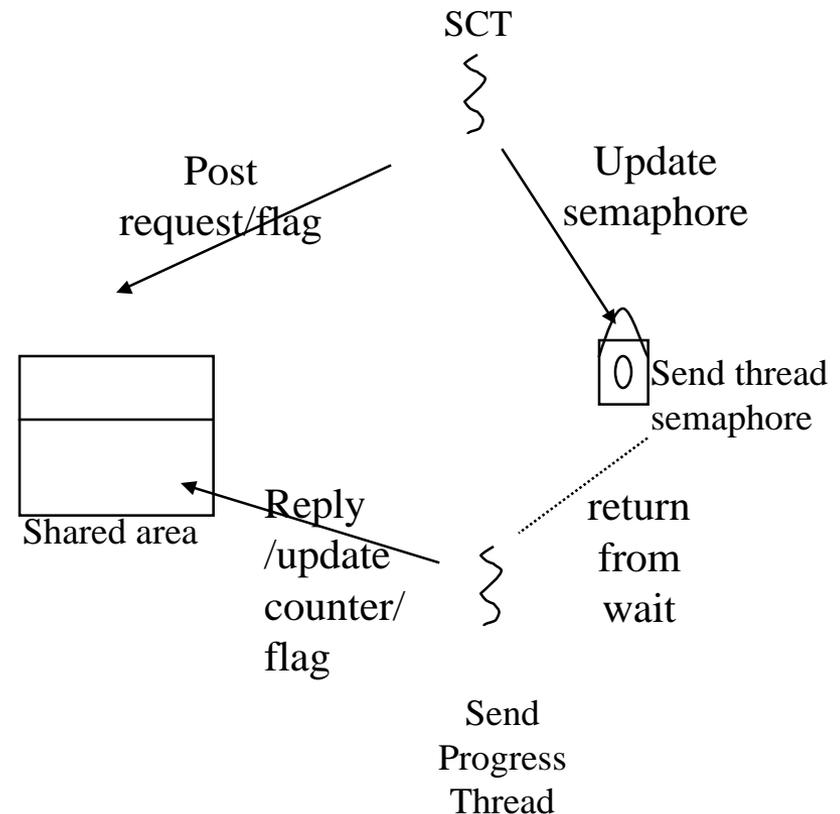
# SCT and Recv Thread

- SCT raises a heartbeat request flag and associates a timeout value
- SCT then sends a message through the TCP connection to recv thread
- Recv thread exits select loop, increments heartbeat counter
- SCT reads updated counter after a specified delay (heartbeat interval)



# SCT and Send Thread

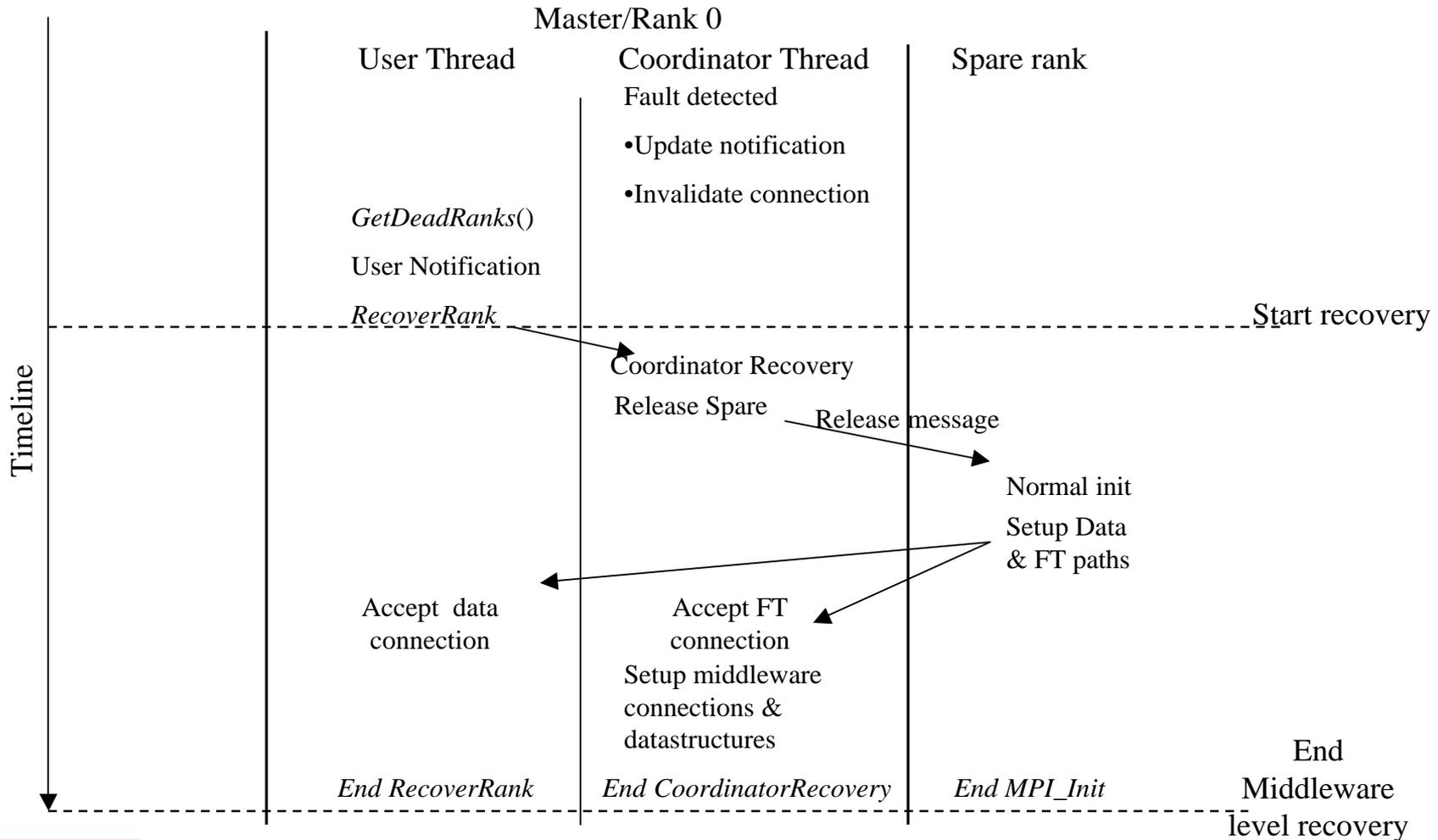
- SCT raises a heartbeat request flag and associates a timeout value
- SCT increments the “send” semaphore to wake the Send thread
- Send thread increments heartbeat counter
- SCT reads updated counter after a specified delay (heartbeat interval)



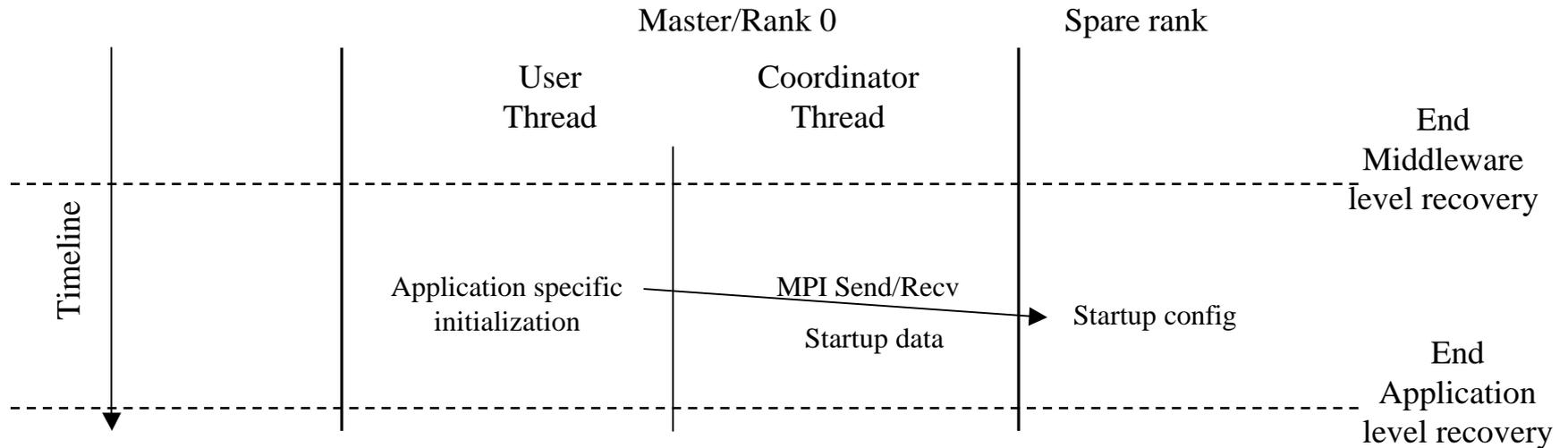
# Recovery

- Middleware level recovery
  - bring process back into application group
- Application level recovery
  - Bring new rank to required state
- Model I: Recovery in context of master process alone
- Model II: Recovery in context of all alive ranks

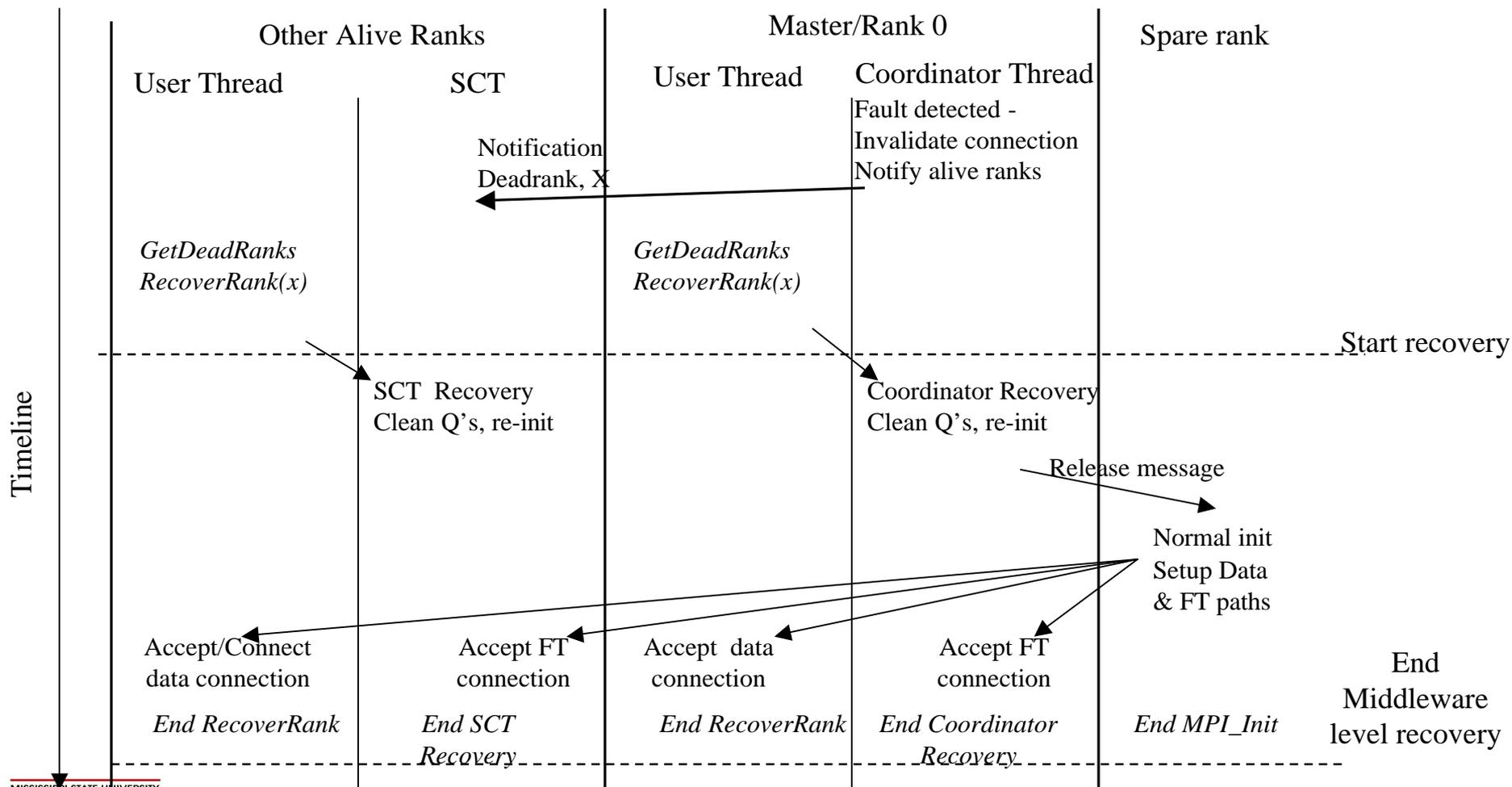
# Model I. Recovery



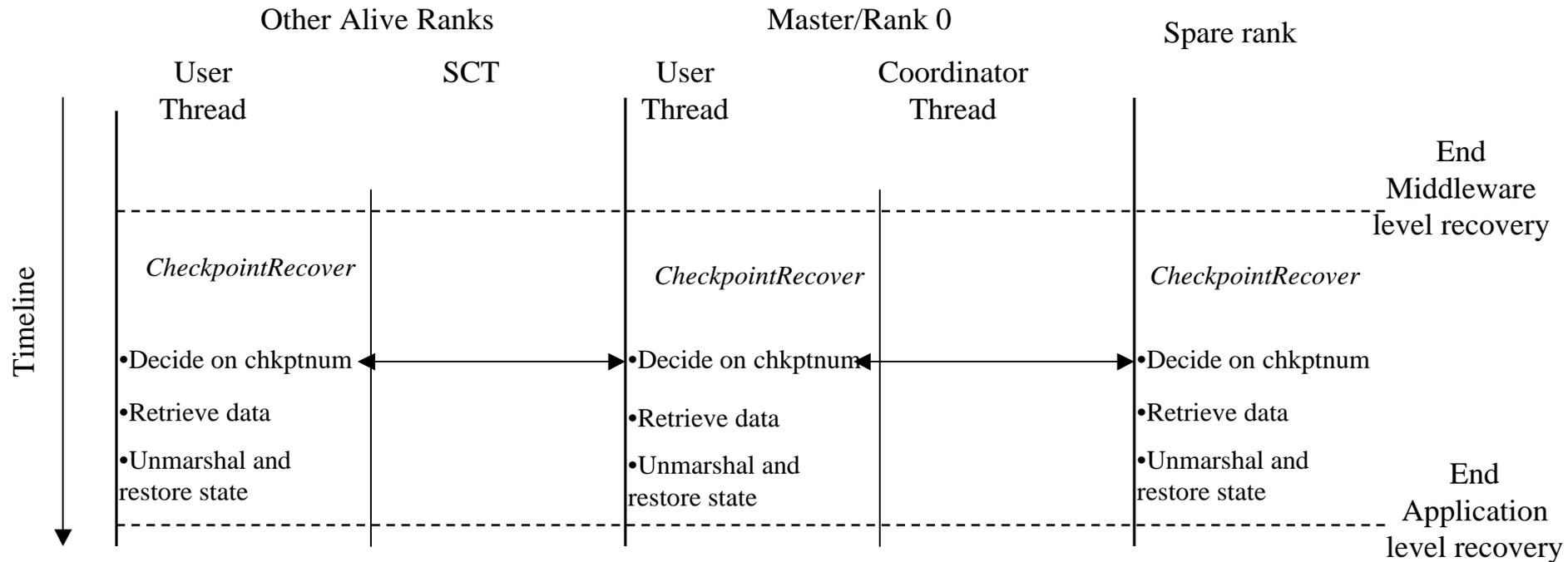
# Model I. Recovery, II.



# Model II. Recovery



# Model II. Recovery, II.



# Model I. Recovery Recommendations

---

- Recovery of a dead slave is optional, as applications can continue without recovering (in most load balanced applications)
- Deciding factors
  - Time for recovery (is recovery justified?)
  - Application progress (no need for slaves at the end of the application)
  - Loss of performance is acceptable (fewer slaves)

# Model II. Recovery Recommendations

---

- Dead ranks must be recovered
- Other option (not currently supported):
  - shrink communicators and reorganize ranks

# Outline

- Introduction
- Background
- Model-based approach
- Usage and implementation
- **Results**
  - Parameters
  - Message-passing overheads
  - Model I
  - Model II
- Future work
- Conclusion

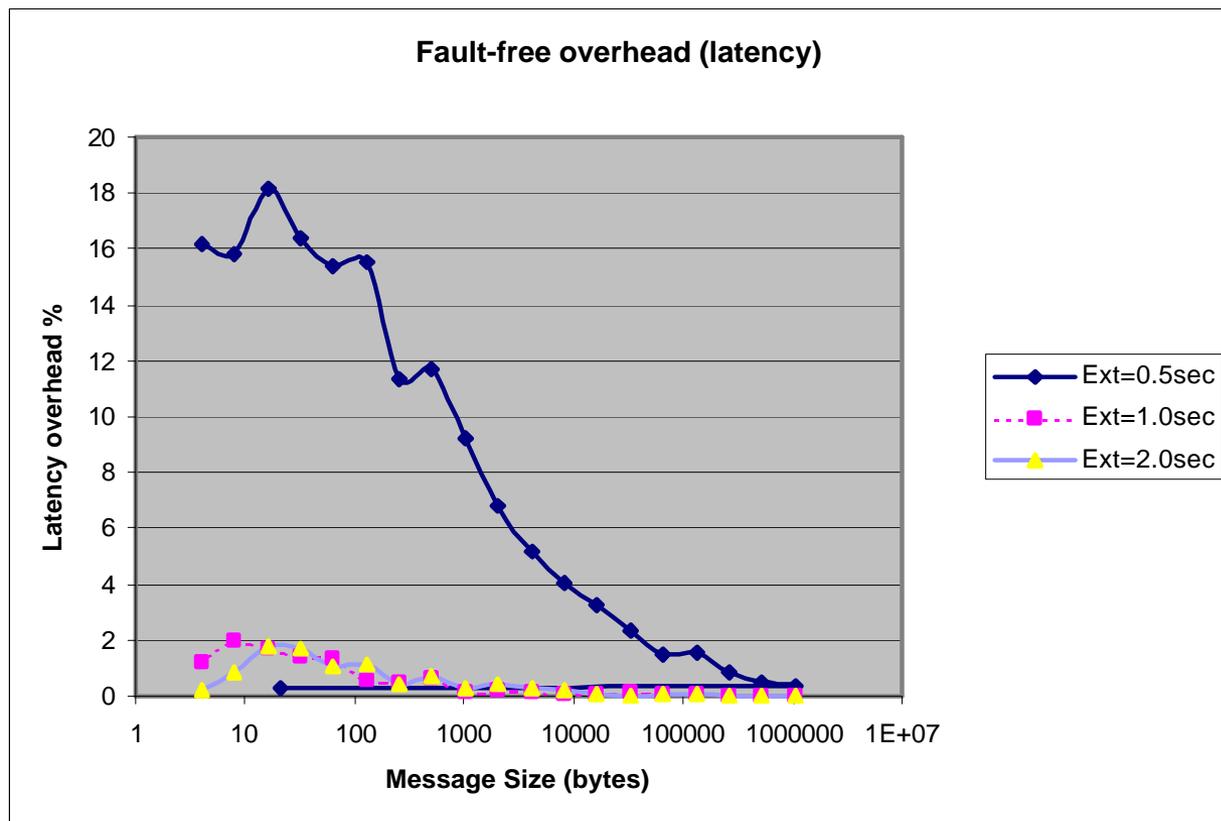
# Parameters

- Fault-free Overhead
  - Overhead incurred in the absence of faults
  - Increase in latency
  - Decrease in bandwidth
  - Increase in run-time
  - Mainly because of detection, and checkpointing and calls to FT API
- Fault-injected Overhead
  - Overhead incurred in the presence of faults
  - Recovery time

Note: All increases and decreases obtained in comparison with standard 1.5 version of MPI/Pro™

# Message Passing Overheads (Latency)

- Percentage increase in latency
- Measured using a ping-pong test
- Setup: PIII 750mhz/Linux 2.4, Fast Ethernet
- “Ext” is frequency of heartbeats

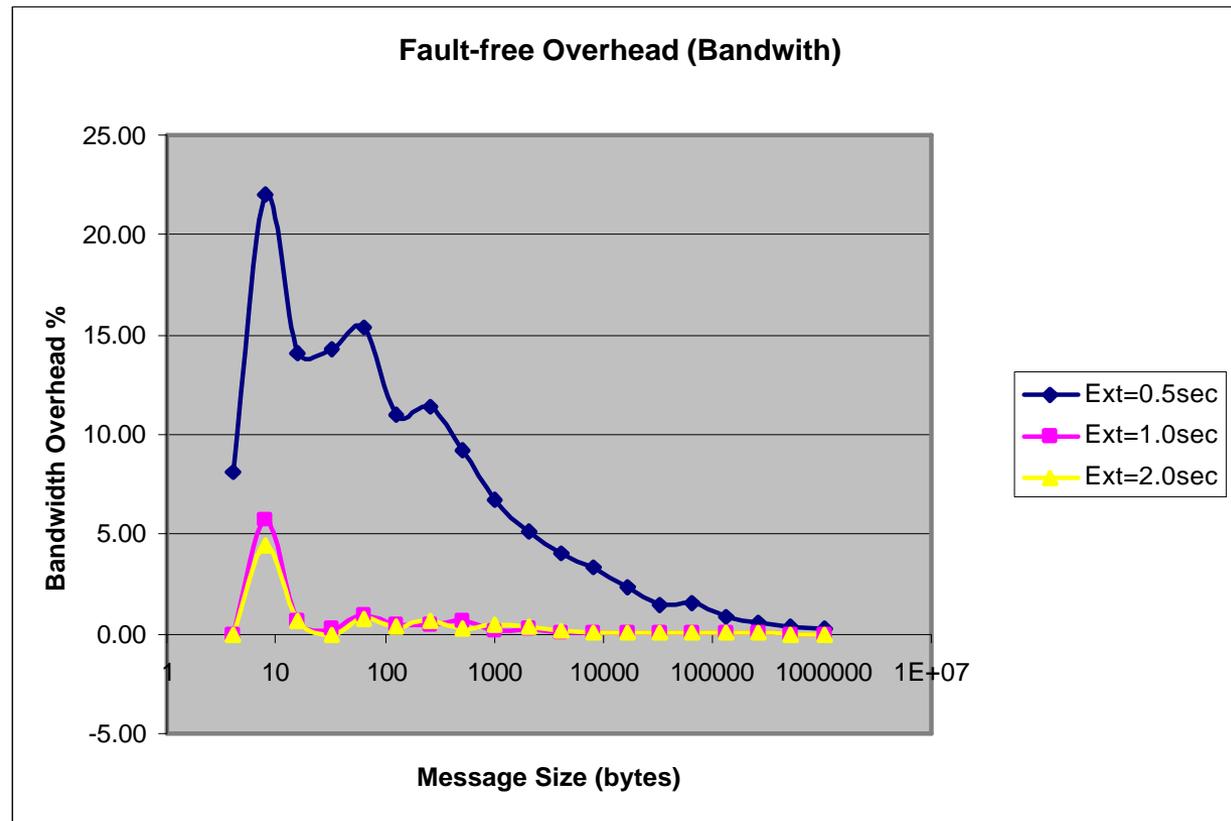


Note: Logarithmic X axis



# Message Passing Overheads (Bandwidth)

- Percentage decrease in bandwidth
- Measured using a modified ping-pong test
- Setup: PIII  
750mhz/Linux 2.4, Fast Ethernet
- “Ext” is frequency of heartbeats

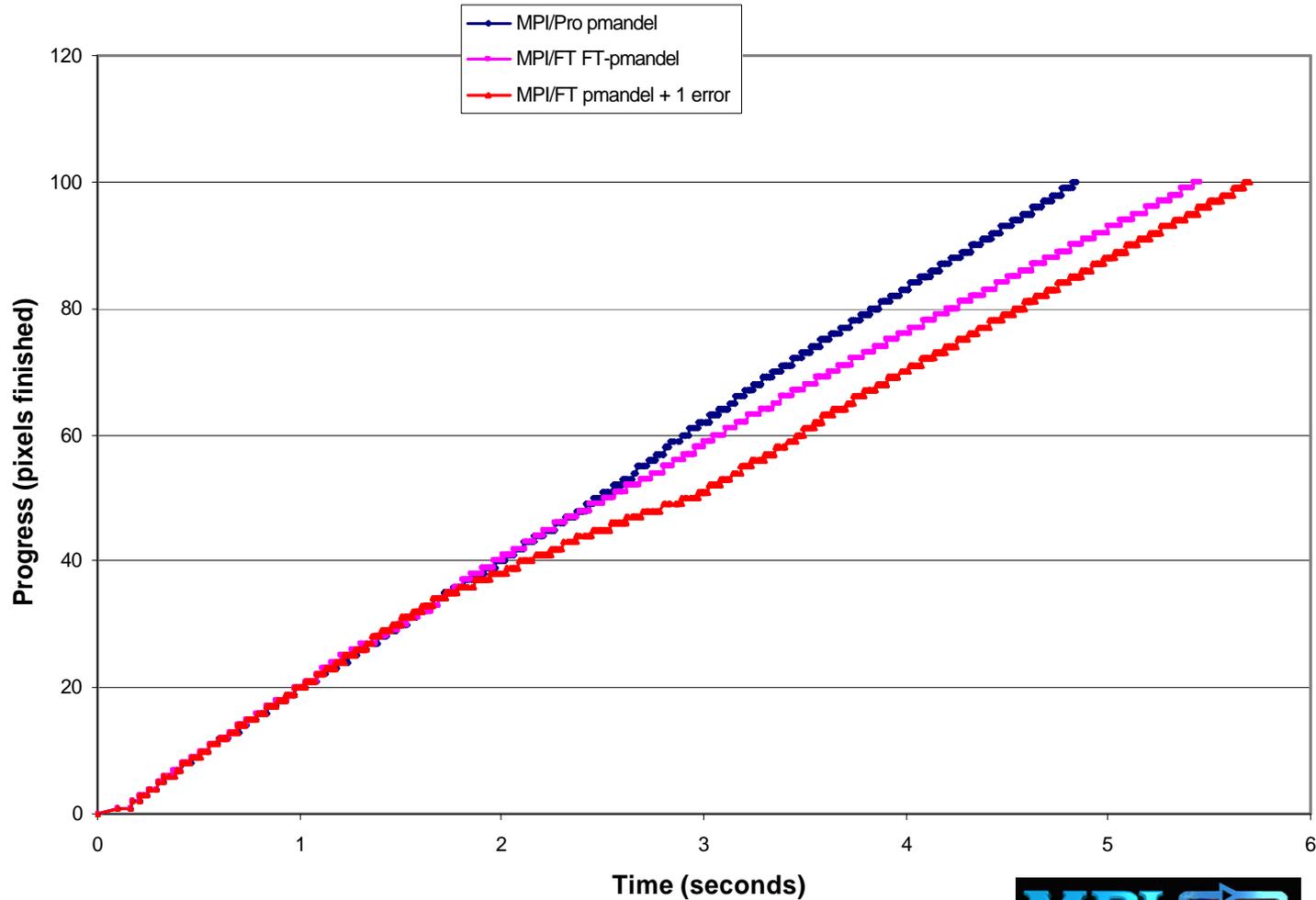


Note: Logarithmic X axis



# Model I. results

- Example program: pmandel fractal program
- Run-time increase (~ 12 %)
- Middleware Recovery time ~32 milliseconds
- Application level recovery time ~ 250 milliseconds
- Pentium 500 Mhz, 128mb RAM, Linux 2.2



# Model II. Results

---

- Example program: Game of life
- Run-time overhead (~25%)
- Overheads vary with data-size and checkpointing frequency
- Implementation and experiments are still underway

# Outline

- Introduction
- Background
- Model-based approach
- Usage and Implementation
- Results
- Future work
  - More models
  - Scalable FT
  - Parallel NMR
- Conclusion

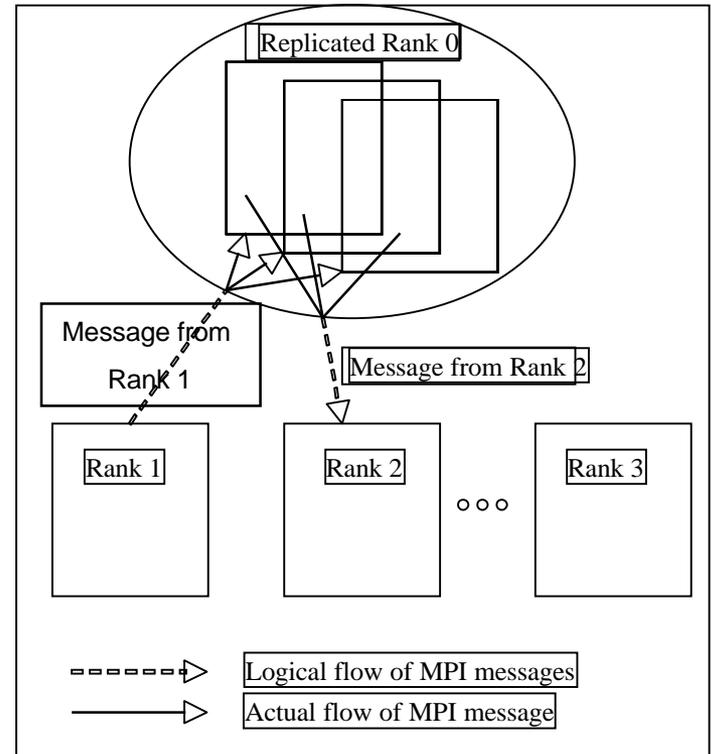
# Future work

- More models
  - Identify more models
  - Refine existing API to users based on feedback
- Scalable Fault tolerance (initial ideas)
  - Scalable detection:
    - Research existing scalable detection methods
    - Some to consider: hierarchical, probabilistic, gossiping, etc.
  - Scalable recovery
    - In some applications processes organize into cohorts, except during startup and finish
    - Frequent communication within cohort, and infrequent global communication
    - Death of single process affects only others in cohort



# Future work, II.

- Parallel NMR
  - Redundant active/passive copies of important processes
  - Issues with ordering of messages between copies
- Fault-tolerance benchmarks
  - Common parameters to evaluate impact/effectiveness of FT middleware
  - Programs and metrics
- Real-time MPI/FT?



# Outline

---

- Introduction
- Background
- Model-based approach
- Usage and implementation
- Results
- Future work
- **Conclusions**

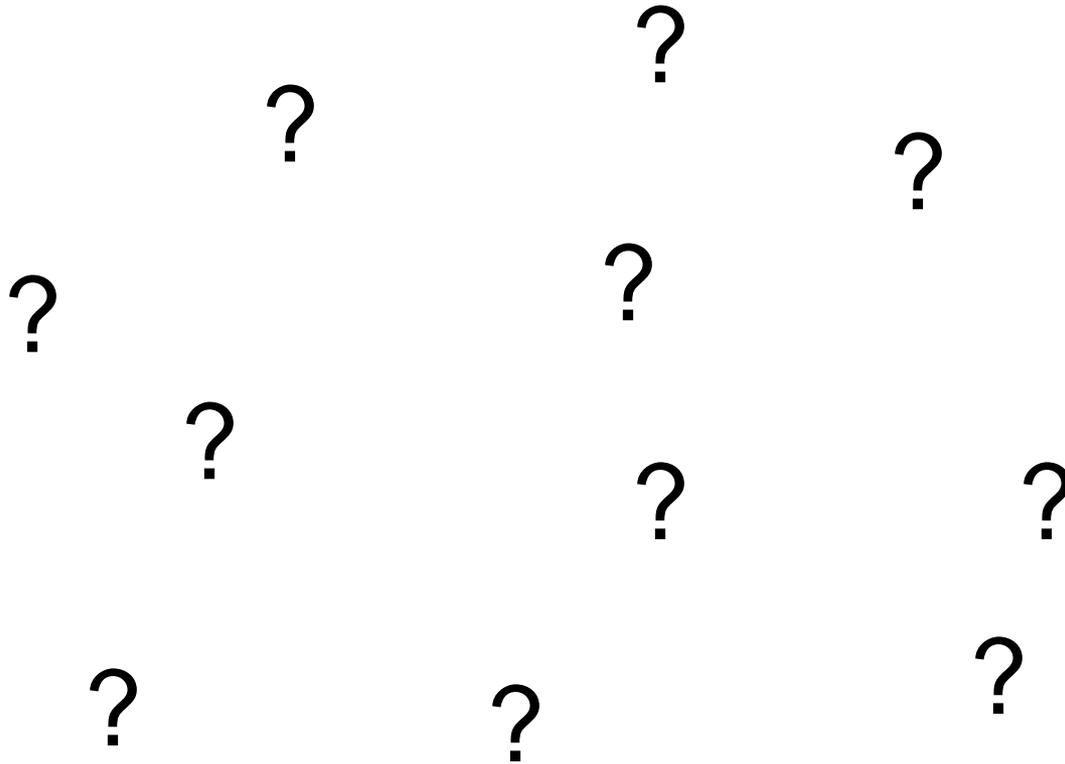
# Conclusions

---

- Need for fault-tolerant MPI presented
- Drawbacks of existing efforts outlined
- Model-based approach
- Preliminary implementation and results
- Future work

# Questions

---



---

# Reserved Slides



# CoCheck

- Features
  - Extension to Condor's single process checkpoint
  - User aware
  - Flushes messages before checkpoint for consistency
  - Main emphasis:
    - Relocation for load balancing
    - Stalling for later resumption
- Drawbacks
  - High overheads
  - Checkpoints entire process state

# Egida

- Features
  - A generic tool for checkpointing in distributed systems
  - User transparent
  - Ported to MPICH
  - Developers can compose their own flavor of logs, and roll-back recovery

# Egida, II.

- Main emphases
  - Low overhead rollback and recovery
  - Recovery coordination
  - Protocol composition (for testing various protocols)
- Disadvantages
  - Checkpoints process state and messages
  - Recovery requires detection of domino states in some cases



# FT-MPI

- Features
  - (Fagg and Dongarra 2000) Part of Harness research
  - Fault-tolerance as an add-on option
- Main Emphasis
  - Health of communicator
  - Methods to expand and shrink communicator
- Drawbacks
  - Describes only recovery of MPI data structures, no comprehensive recovery
  - No methods to recover at application level



# Starfish

- Features
  - Partial implementation of MPI2
  - Implements an event model for notification
- Main emphasis
  - Adapt to dynamic cluster changes
  - Provide ability for dynamic change in number of processes
  - Fault tolerance is a byproduct
- Disadvantages
  - No explicit detection
  - No specific recovery procedure implementation at MPI level

# Model II. API Description

- **MPIFT\_GetDeadRanks**
  - Same functionality as in model-1
  - Can be invoked at any rank
- **MPIFT\_RecoverRank**
  - Same functionality as in model-1
  - Must be invoked by alive ranks. Recovery is a collective operation among alive ranks

# Model II. API Description, II.

INT

```
MPIFT_ChkptDo (  
    IN  MPI_Comm comm,  
    IN  void *data_to_store,  
    IN  INT data_size,  
    OUT INT *chkpt_num  
);
```

- Stores user provided data
- Currently uses a file for each number/slave ( later move to incremental chkpt )
- Collective call across all ranks
  - Must be called by all ranks
  - Succeeds only when all ranks are alive
  - Each check pointed information has same identifier number across all ranks (for use )



# Model II. API Description, III.

INT

```
MPIFT_ChkptRecover (  
    IN  MPI_Comm comm,  
    OUT void *data_retrived,  
    IN  INT    in_data_size,  
    OUT INT    *out_data_size,  
    OUT INT    *chkpt_num_retrieved  
);
```

- Retrieves user provided state information
  - Internal protocol to retrieve highest numbered information
  - Protocol steps down to next highest number (if any rank has problems opening current highest information)
- Collective call across all ranks
  - Must be called by all ranks
  - Succeeds only when all ranks are alive



# Model II. code changes

---

```
Distribute Data;  
Initialize conditions;  
While (! enuf_ iterations){  
    Communicate_part();  
    Compute_part();  
    MPI_Barrier();  
};
```

Original code for Model-II applications

# Model II. code changes. (continued)

```
Distribute Data;
Initialize conditions;
While (! enuf_iterations){
    MPIFT_GetDeadRanks (&deadcount, &deadarray,
                        arraysizes);
    if (deadcount >1){
        for (I = 0.. Deadcount-1){
            MPIFT_RecoverRank (deadarray [I]);
        };
        MPIFT_ChkptRecover (&state_retrieved,
                            in_size, &out_size, &chkpt_num);
        Restore_AppState (state_retrieved);
    };
}
```

A

→ B

```
// normal run
Communicate_part ();
Compute_part ();
MPI_Barrier ();
Get_AppState (&state_tostore);
MPIFT_ChkptDo (&state_tostore
               in_data, &chkpt_num);
};
```

