Fault Tolerant MPI

Edgar Gabriel
Graham E. Fagg*, Thara Angskun, George Bosilca,
Antonin Bukovsky, Jack J. Dongarra
<egabriel, fagg, dongarra>@cs.utk.edu
harness@cs.utk.edu
Overview

» Overview and Motivation
» Semantics, Concept and Architecture of FT-MPI
» Implementation details
» Performance comparison to LAM and MPICH
  » Pt2pt performance
  » HPL benchmark
  » PSTSWM benchmark
» About writing fault-tolerant applications
  » General issues
  » A fault tolerant, parallel, equation solver (PCG)
  » A Master-Slave framework
» Tools for FT-MPI
» Ongoing work
» Summary
» Demonstration
Motivation

» HPC systems with thousand of processors
  » Increased probability of a node failure
  » Most systems nowadays are robust – machines do not crash because of a node failure

» Node and communication failure in distributed environments
» Very long running applications
» Security relevant applications
MPI and Error handling

- MPI_ERRORS_ARE_FATAL (Default mode):
  - Abort the application on the first error
- MPI_ERRORS_RETURN:
  - Return error-code to user
  - State of MPI undefined
  - “...does not necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handler is to allow a user to issue user-defined error messages and take actions unrelated to MPI...An MPI implementation is free to allow MPI to continue after an error...” (MPI-1.1, page 195)
  - “Advice to implementors: A good quality implementation will, to the greatest possible extent, circumvent the impact of an error, so that normal processing can continue after an error handler was invoked.”
Related work

**Transparency**: application checkpointing, MP API+Fault management, automatic.
- **application ckpt**: application store intermediate results and restart from them
- **MP API+FM**: message passing API returns errors to be handled by the programmer
- **automatic**: runtime detects faults and handles recovery

**Checkpoint coordination**: none, coordinated, uncoordinated.
- **coordinated**: all processes are synchronized, network is flushed before ckpt;
  all processes rollback from the same snapshot
- **uncoordinated**: each process checkpoint independently of the others
  each process is restarted independently of the others

**Message logging**: none, pessimistic, optimistic, causal.
- **pessimistic**: all messages are logged on reliable media and used for replay
- **optimistic**: all messages are logged on non reliable media. If 1 node fails, replay is done according to other nodes logs. If >1 node fail, rollback to last coherent checkpoint
- **causal**: optimistic+Antecedence Graph, reduces the recovery time
# Classification of ft message passing systems

<table>
<thead>
<tr>
<th>Framework</th>
<th>API</th>
<th>Communication Lib.</th>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No automatic/transparent, n fault tolerant, scalable message passing environment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Automatic</th>
<th>Non Automatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoint based</td>
<td>Log based</td>
<td></td>
</tr>
<tr>
<td>Optimistic log</td>
<td>Causal log</td>
<td>Pessimistic log</td>
</tr>
</tbody>
</table>

- **Cocheck** Independent of MPI [Ste96]
- **Starfish** Enrichment of MPI [AF99]
- **Clip** Semi-transparent checkpoint [CLP97]
- **Optimistic recovery in distributed systems** n faults with coherent checkpoint [SY85]
- **Manetho** n faults [EZ92]
- **Egida** [RAV99]
- **Coordinated checkpoint**
- **MPI/FT** Redundance of tasks [BNC01]
- **FT-MPI** Modification of MPI routines User Fault Treatment [FD00]
- **MPICH-V** N faults Distributed logging
- **MPI-FT** N fault Centralized server [LNLE00]
- **Pruitt 98** 2 faults sender based [PRU98]
- **Sender based Mess. Log.** 1 fault sender based [JZ87]
FT-MPI

» Define the behavior of MPI in case an error occurs
» Give the application the possibility to recover from a node-failure
» A regular, non fault-tolerant MPI program will run using FT-MPI
» Stick to the MPI-1 and MPI-2 specification as closely as possible (e.g. no additional function calls)

» What FT-MPI does not do:
  » Recover user data (e.g. automatic checkpointing)
  » Provide transparent fault-tolerance
FT-MPI Semantics, Concept and Architecture
FT-MPI failure modes

» **ABORT**: just do as other implementations

» **BLANK**: leave hole

» **SHRINK**: re-order processes to make a contiguous communicator
  » Some ranks change

» **REBUILD**: re-spawn lost processes and add them to MPI_COMM_WORLD
FT-MPI communication modes

» **RESET**: ignore and cancel all currently active communications and requests in case an error occurs. User will re-post all operations after recovery.

» **CONTINUE**: all operations which returned MPI_SUCCESS will be finished after recovery... The code just keeps on going

» **FT-MPI defaults:**
  » communicator mode: REBUILD
  » communication mode: RESET
  » error-handler: MPI_ERRORS_RETURN
RESET: a message posted in one epoch does not match a receive posted in another epoch

CONTINUE: epoch argument not regarded for message matching
<table>
<thead>
<tr>
<th>Question:</th>
<th>How do we now a process has failed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answer:</td>
<td>The return code of an MPI operation returns MPI_ERR_OTHER. The failed process is not necessarily the process involved in the current operation!</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question:</th>
<th>What do I have to do if a process has failed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answer:</td>
<td>You have to start a recovery operation before continuing the execution. All non-local MPI objects (e.g. communicators) have to be re-instantiated or recovered.</td>
</tr>
</tbody>
</table>
» If no error occurs: identical to MPI-1
» If a process fails:
  » All point-to-point operations to failed processes will be dropped
  » If an operation returns MPI_SUCCESS this point-to-point operation will be finished successfully (CONTINUE mode) – unless the user wishes to cancel all ongoing communications (RESET mode)
» If an asynchronous operation has been posted successfully (Isend/Irecv) the operation will be finished (CONTINUE mode) [Application still has to call MPI_Wait/Test]– or the user wishes to cancel it (RESET mode)
» Waitall/Testall etc.: you might have to check the error code in status to determine which operation was not successful
Point-to-point semantics (II)

» If you are using the BLANK mode: Communication to a blank process will be treated as communication to MPI_PROC_NULL

» In the RESET communication mode: if you use the SHRINK mode, all requests will be redirected to the new ranks of your process (not yet done).
Collective semantics

» Ideally: atomic collective operations – either everybody succeeds or nobody
  » Possible, but slow

» Alternative: if an error occurs the outcome of the collective operations is undefined ... welcome back to MPI
  » Not that bad: no input buffer is touched, operation can easily be repeated after recovery
  » Not that bad: user can check whether operation has finished properly (e.g. executing MPI_Barrier after operations)
  » It is bad, if you use MPI_IN_PLACE (MPI-2)
The recovery procedure

» Your communicators are invalid
  » MPI_COMM_WORLD and MPI_COMM_SELF are re-instantiated automatically
  » Rebuild all your communicators in the same order like you did previously (CONTINUE mode)

» Check how many processes have failed
» Check who has failed (or has been replaced in the REBUILD mode)
» Check which requests you want to cancel/free

» Continue the execution of your application from the last consistent point
Application view

» Line by line checking

/* check return value */
ret = MPI_Send ( buf, count, datatype, tag, dest, comm );
if ( ret == MPI_ERR_OTHER ){
    /* call recovery function */
}

» Usage of error-handlers

/* install recovery handler just once */
MPI_Comm_create_errhandler (my_recover_function, &errh);
MPI_Comm_set_errhandler (MPI_COMM_WORLD, errh);

/* automatic checking. No modification necessary */
MPI_Send (...)
MPI_Scatter (...)

Some modification to top level control
Application scenario

rc = MPI_Init (...)

If normal startup

Install Error Handler & Set LongJMP

Call Solver (...)

MPI_Finalize(...)
Application scenario

rc=MPI_Init (...)

Set LongJMP

ErrorHandler
Do recover ()
Do JMP

Call Solver (...)

MPI_Finalize(...)

On error (automatic via the MPI runtime library)
Application scenario

1. `rc=MPI_Init(...)`

   If restarted process

2. `ErrorHandler`
   - `Set` LongJMP

3. Call Solver (...)

4. `MPI_Finalize(...)`

   I am New
   Do recover ()
Architecture

High level services

Running under a HARNESS Core

Name Service

MPI application

MPI application

libftmpi

libftmpi

Startup plugin

Startup plugin

HARNESS

HARNESS

One startup plug-in per ‘core’
Architecture

High level services

Running outside of a Core

One startup daemon per ‘host’

- Name Service
- Ftmpi_notifier
- Startup_d
- libftmpi
- MPI application
- MPI application

ICL
EuroPVM/MPI 2003
September 2003
Team HARNESS
Implementation details
Implementation Details

User Application

MPI Library layer

Derived Datatypes/
Buffer Management  Message Lists/
Non-blocking queues

FT-MPI runtime library

Hlib

SNIPE 2

Startup Daemon

HARNESS core

Notifier Service

Name Service

Failure detection

Failure event notification

MPI Messages
Implementation Details

» Layer approach
  » Top layer handles MPI Objects, F2C wrappers etc
  » Middle layer handles derived data types
    » Data <-> buffers (if needed)
    » Message Queues
    » Collectives and message ordering
  » Lowest layer handles data movement / system state
    » Communications via SNIPE/2
    » System state
      » Coherency via Multi-phase commit algorithm
      » Dynamic leader that builds complete state
      » Atomic distribution and leader recovery
Automatically tuned collective operations

» Collective (group) communications

» Built a self tuning system that examines a number of possibilities and chooses the best for the target architecture

» Like ATLAS for numeric software

» Automatic Collective Communication Tuning
Implementation Details

scatter (cetus), 128k, 8 procs

Time [us]

Segment size

sequential
chain
binary
binary2
Implementation Details

» Security
  » Simple model is via Unix file system
    » Startup daemon only run files from certain directories
    » Hcores are restricted to which directories they can load shared objects from
  » Signed plug-ins via PGP

  » Complex uses openSSL library and X509 certificates
    » One cert for each host that a startup daemon executes on
    » One for each user of the DVM

  » Currently testing with globus CA issued certs

  » startup times for 64 jobs
    » X509/ssl 0.550 seconds
    » Unix file system and plain TCP 0.345 seconds
Data conversion

- Single sided data conversion
  - Removes one “unnecessary” memory operation in heterogeneous environments
  - No performance penalty in homogeneous environments
  - Requires everybody to know the data representation of every other process
  - Currently: receiver side conversion, sender side conversion possible
  - Single sided conversion for long double implemented

- Problems
  - Pack/Unpack: don’t know where the message is coming from => XDR conversion
  - Mixed Mode communication: will be handled in the first release
  - Does not work if user forces different data length through compile-flags

- Fall back solution: enable XDR-conversion through configure-flag
Performance comparison

» Performance comparison with non fault-tolerant MPI-libraries
  » MPICH 1.2.5
  » MPICH2 0.9.4
  » LAM 7.0

» Benchmarks
  » Point-to-point benchmark
  » PSTSWM
  » HPL benchmark
Latency test-suite (large messages)
Latency test-suite (small messages)
### Shallow Water Code (PSTSWM)

**16 processes**

<table>
<thead>
<tr>
<th></th>
<th>MPICH 1.2.5 [sec]</th>
<th>MPICH 2 – 0.9.4 [sec]</th>
<th>FT-MPI [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>t42.l16.240</td>
<td>40.76</td>
<td>36.60</td>
<td>35.79</td>
</tr>
<tr>
<td>t42.l17.240</td>
<td>43.20</td>
<td>38.88</td>
<td>38.01</td>
</tr>
<tr>
<td>t42.l18.240</td>
<td>45.69</td>
<td>41.11</td>
<td>40.32</td>
</tr>
<tr>
<td>t85.l16.24</td>
<td>33.65</td>
<td>30.77</td>
<td>30.50</td>
</tr>
<tr>
<td>t85.l17.24</td>
<td>36.02</td>
<td>32.75</td>
<td>32.31</td>
</tr>
<tr>
<td>t85.l18.24</td>
<td>38.14</td>
<td>34.64</td>
<td>34.25</td>
</tr>
<tr>
<td>t170.l1.12</td>
<td>9.01</td>
<td>8.22</td>
<td>8.03</td>
</tr>
<tr>
<td>t170.l2.12</td>
<td>17.78</td>
<td>16.35</td>
<td>16.14</td>
</tr>
<tr>
<td>t170.l3.12</td>
<td>26.76</td>
<td>24.53</td>
<td>24.18</td>
</tr>
</tbody>
</table>
### Shallow Water Code (PSTSWM) 32 processes

<table>
<thead>
<tr>
<th></th>
<th>MPICH 1.2.5 [sec]</th>
<th>MPICH 2 – 0.9.4 [sec]</th>
<th>FT-MPI [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>t42.l16.240</td>
<td>40.69</td>
<td>38.00</td>
<td>30.33</td>
</tr>
<tr>
<td>t42.l17.240</td>
<td>43.41</td>
<td>40.59</td>
<td>32.35</td>
</tr>
<tr>
<td>t42.l18.240</td>
<td>45.89</td>
<td>42.50</td>
<td>34.20</td>
</tr>
<tr>
<td>t85.l16.24</td>
<td>33.54</td>
<td>31.54</td>
<td>26.20</td>
</tr>
<tr>
<td>t85.l17.24</td>
<td>35.57</td>
<td>33.34</td>
<td>27.84</td>
</tr>
<tr>
<td>t85.l18.24</td>
<td>37.63</td>
<td>35.11</td>
<td>29.57</td>
</tr>
<tr>
<td>t170.l1.12</td>
<td>8.69</td>
<td>8.29</td>
<td>6.58</td>
</tr>
<tr>
<td>t170.l2.12</td>
<td>17.33</td>
<td>16.54</td>
<td>13.29</td>
</tr>
<tr>
<td>t170.l3.12</td>
<td>25.92</td>
<td>24.52</td>
<td>20.16</td>
</tr>
</tbody>
</table>
Communication pattern 32 processes
Number of messages
FT-MPI using the same proc. distribution like MPICH 1.2.5 and MPICH 2

<table>
<thead>
<tr>
<th></th>
<th>MPICH 1.2.5 [sec]</th>
<th>MPICH 2 – 0.9.4 [sec]</th>
<th>FT-MPI [sec]</th>
<th>FT-MPI prev. [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>t42.l16.240</td>
<td>40.69</td>
<td>38.00</td>
<td>36.79</td>
<td>30.33</td>
</tr>
<tr>
<td>t42.l17.240</td>
<td>43.41</td>
<td>40.59</td>
<td>39.27</td>
<td>32.35</td>
</tr>
<tr>
<td>t42.l18.240</td>
<td>45.89</td>
<td>42.50</td>
<td>41.37</td>
<td>34.20</td>
</tr>
<tr>
<td>t85.l16.24</td>
<td>33.54</td>
<td>31.54</td>
<td>31.22</td>
<td>26.20</td>
</tr>
<tr>
<td>t85.l17.24</td>
<td>35.57</td>
<td>33.34</td>
<td>32.93</td>
<td>27.84</td>
</tr>
<tr>
<td>t85.l18.24</td>
<td>37.63</td>
<td>35.11</td>
<td>34.99</td>
<td>29.57</td>
</tr>
<tr>
<td>t170.l1.12</td>
<td>8.69</td>
<td>8.29</td>
<td>7.88</td>
<td>6.58</td>
</tr>
<tr>
<td>t170.l2.12</td>
<td>17.33</td>
<td>16.54</td>
<td>15.96</td>
<td>13.29</td>
</tr>
<tr>
<td>t170.l3.12</td>
<td>25.92</td>
<td>24.52</td>
<td>23.77</td>
<td>20.16</td>
</tr>
</tbody>
</table>
### MPICH 1.2.5 using the same process distribution like FT-MPI

<table>
<thead>
<tr>
<th>Scenario</th>
<th>MPICH 1.2.5 prev. [sec]</th>
<th>MPICH 1.2.5 [sec]</th>
<th>MPICH 2 – 0.9.4 [sec]</th>
<th>FT-MPI [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>t42.l16.240</td>
<td>40.69</td>
<td>34.49</td>
<td>-</td>
<td>30.33</td>
</tr>
<tr>
<td>t42.l17.240</td>
<td>43.41</td>
<td>36.81</td>
<td>-</td>
<td>32.35</td>
</tr>
<tr>
<td>t42.l18.240</td>
<td>45.89</td>
<td>38.95</td>
<td>-</td>
<td>34.20</td>
</tr>
<tr>
<td>t85.l16.24</td>
<td>33.54</td>
<td>28.93</td>
<td>-</td>
<td>26.20</td>
</tr>
<tr>
<td>t85.l17.24</td>
<td>35.57</td>
<td>30.75</td>
<td>-</td>
<td>27.84</td>
</tr>
<tr>
<td>t85.l18.24</td>
<td>37.63</td>
<td>32.55</td>
<td>-</td>
<td>29.57</td>
</tr>
<tr>
<td>t170.l1.12</td>
<td>8.69</td>
<td>7.48</td>
<td>-</td>
<td>6.58</td>
</tr>
<tr>
<td>t170.l2.12</td>
<td>17.33</td>
<td>14.92</td>
<td>-</td>
<td>13.29</td>
</tr>
<tr>
<td>t170.l3.12</td>
<td>25.92</td>
<td>22.31</td>
<td>-</td>
<td>20.16</td>
</tr>
</tbody>
</table>
HPL Benchmark

4 Processes, Problem size 6000, 2.4 GHz Dual PIV, GEthernet

<table>
<thead>
<tr>
<th>Blocksize</th>
<th>MPICH 1.2.5 [sec]</th>
<th>MPICH 2 0.9.4 [sec]</th>
<th>LAM 7.0 [sec]</th>
<th>FT-MPI [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>28.16</td>
<td>27.50</td>
<td>27.85</td>
<td>27.95</td>
</tr>
<tr>
<td>80</td>
<td>18.18</td>
<td>17.40</td>
<td>17.81</td>
<td>18.10</td>
</tr>
<tr>
<td>240</td>
<td>19.88</td>
<td>19.15</td>
<td>19.58</td>
<td>20.03</td>
</tr>
</tbody>
</table>
Writing fault tolerant applications
Am I a re-spawned process? (I)

» 1st Possibility: new FT-MPI constant

rc = MPI_Init ( argc, argv );
if ( rc == MPI_INIT_RESTARTED_PROC) {
    /* yes, I am restarted */
}

» Fast (no additional communication required)
» Non-portable
Am I a re-spawned process? (II)

2nd Possibility: usage of static variables and collective operations

```c
int sum, wasalivebefore = 0;

MPI_Init (&argc, &argv);
MPI_Comm_size ( MPI_COMM_WORLD, &size );
MPI_Allreduce ( &wasalivebefore, &sum, ..., MPI_SUM, ... );
if ( sum == 0 )
    /* Nobody was alive before, I am part of the initial set */
else {
    /* I am re-spawned, total number of re-spawned procs is */
    numrespawned = size - sum;
}
wasalivebefore = 1;
```

The Allreduce operation has to be called in the recovery routine by the surviving processes as well.

Portable, requires however communication!

Works just for the REBUILD mode.
Which processes have failed? (I)

1st Possibility: two new FT-MPI attributes

/* How many processes have failed? */
MPI_Comm_get_attr ( comm, FTMPI_NUM_FAILED_PROCS, &valp, &flag );
numfailedprocs = (int) *valp;

/* Who has failed? Get an errorcode, who’s error-string contains
   the ranks of the failed processes in MPI_COMM_WORLD */
MPI_Comm_get_attr ( comm, FTMPI_ERROR_FAILURE, &valp, &flag );
errorcode = (int) *valp;

MPI_Error_get_string ( errcode, errstring, &flag );
parsestring ( errstring );
Which processes have failed? (II)

» 2nd Possibility: usage of collective operations and static variables (see ‘Am I a re-spawned process’, 2nd possibility)

```c
int procarr[MAXPROC];

MPI_Allgather ( &wasalivebefore,..., procarr, ..., MPI_COMM_WORLD );
for ( I = 0; I < size; I++) {
    if ( procarr[I] == 0 )
        /* This is process is respawned */
}
```

» Similar approaches based on the knowledge of some application internal data are possible
A fault-tolerant parallel CG-solver

» Tightly coupled
» Can be used for all positive-definite, RSA-matrices in the Boeing-Harwell format
» Do a “backup” every \( n \) iterations

» Can survive the failure of a single process
» Dedicate an additional process for holding data, which can be used during the recovery operation
» Work-communicator excludes the backup process

» For surviving \( m \) process failures \((m < np)\) you need \( m \) additional processes
The backup procedure

If your application shall survive one process failure at a time

If your application shall survive one process failure at a time

<table>
<thead>
<tr>
<th>Rank 0</th>
<th>Rank 1</th>
<th>Rank 2</th>
<th>Rank 3</th>
<th>Rank 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>26</td>
</tr>
</tbody>
</table>

or

\[ b_i = \sum_{j=1}^{np} v_i(j) \]

Implementation: a single reduce operation for a vector
Keep a copy of the vector \( v \) which you used for the backup
The backup procedure

If your application shall survive two process failures

\[
x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} =
\]

\[
x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} =
\]

with \( x \) determined as in the Red-Solomon Algorithm
The recovery procedure

- Rebuild work-communicator
- Recover data

<table>
<thead>
<tr>
<th>Rank 1</th>
<th>Rank 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank 0</th>
<th>Rank 2</th>
<th>Rank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

- Reset iteration counter
- On each process: copy backup of vector \( \mathbf{v} \) into the current version
PCG overall structure

```c
int iter=0;
MPI_Init (&argc, &argv);
if (!respawned ) initial data distribution
register error handler
set jump-mark for longjmp;
create work communicator;
if recovering : recover data for re-spawned process
    all other processes: go back to the same
    backup
    do {
        if ( iter%backupiter==0) do backup;
        do regular calculation ...
        e.g. MPI_Send (...);
        iter++;
    } (while (iter < maxiter) && (err < errtol));
MPI_Finalize ();
if error
```
## PCG performance

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Number of processes</th>
<th>Execution time</th>
<th>Recovery time</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4054</td>
<td>4 + 1</td>
<td>5 sec</td>
<td>1.32 sec</td>
<td>26.4%</td>
</tr>
<tr>
<td>428650</td>
<td>8 + 1</td>
<td>189 sec</td>
<td>3.30 sec</td>
<td>1.76%</td>
</tr>
<tr>
<td>2677324</td>
<td>16 + 1</td>
<td>1162 sec</td>
<td>11.4 sec</td>
<td>0.97%</td>
</tr>
</tbody>
</table>

Raw FT-MPI recovery times on Pentium IV GEthernet are:
- 32 processes - 3.90 seconds
- 64 processes - 7.80 seconds
A Master-Slave framework

» Useful for parameter sweeps
» Basic concept: Master keeps track of the state of each process and which work has been assigned to it
» Works for the REBUILD, SHRINK and BLANK mode

» Does not use the longjmp method, but continues execution from the point where the error occurred
» Implementation in C and Fortran available
» Implementation with and without the usage of error-handlers available

» If master dies, the work is restarted from the beginning (REBUILD) or stopped (BLANK/SHRINK)
Master process: transition-state diagram

- \textit{cont}
  - available
  - working
  - received
  - finished
- \textit{send}
  - ok
  - error
- \textit{recv}
  - ok
  - error
- \textit{BLANK/SHRINK}: mark failed processes as DEAD
- \textit{REBUILD}: mark failed processes as AVAILABLE
Worker process: transition-state diagram

- AVAILABLE
- SEND_FAILED
- RECV_FAILED
- RECEIVED
- FINISHED
- done

REBUILD: Master died, reset state to AVAILABLE
Master-Slave: user interface

» Abstraction of user interface:

```c
void FT_master_init();
void FT_master_finalize();

int FT_master_get_workid();
int FT_master_get_work(int workid, char** buf,
                        int* buflen);
int FT_master_ret_result(int workid, char* buf,
                         int* buflen);

void FT_worker_init();
void FT_worker_finalize();
int FT_worker_dowork(int workid, char *buf,
                     int buflen);
int FT_worker_getres(int workid, char *buf,
                      int* buflen);
```
Tools for FT-MPI
Harness console

- HARNESS user interfaces
  - Manual via command line utilities
    - hrun or ftmpirun
  - HARNESS Console
    - Has much of the functionality of the PVM + the addition to control jobs via ‘job-run’ handles
      - When a MPI job is started all processes get a job number. The whole application can be signed or killed via this number with a single command.
    - Can use hostfiles and script command files
LINUX>console
Welcome to the Harness/FT-MPI console
con> conf
Found 4 hosts

<table>
<thead>
<tr>
<th>HostID</th>
<th>HOST</th>
<th>PORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>torc4.cs.utk.edu</td>
<td>22503</td>
</tr>
<tr>
<td>1</td>
<td>torc1.cs.utk.edu</td>
<td>22503</td>
</tr>
<tr>
<td>2</td>
<td>torc2.cs.utk.edu</td>
<td>22500</td>
</tr>
<tr>
<td>3</td>
<td>torc3.cs.utk.edu</td>
<td>22502</td>
</tr>
</tbody>
</table>

con>ps

<table>
<thead>
<tr>
<th>ProcID</th>
<th>RunID</th>
<th>HostID</th>
<th>Command</th>
<th>Comment</th>
<th>Status</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>20482</td>
<td>0</td>
<td>./bmtest</td>
<td>FTMPI:proces</td>
<td>exited(val:0)</td>
<td>5s</td>
</tr>
<tr>
<td>4097</td>
<td>20482</td>
<td>1</td>
<td>./bmtest</td>
<td>FTMPI:proces</td>
<td>exited(val:0)</td>
<td>5s</td>
</tr>
</tbody>
</table>

con>
MPE with FT-MPI

FT-MPI HPL trace via Jumpshot using MPE profiling
KOJAK/PAPI/Expert

Performance Properties
- 0.0 Total
  - 81.4 Execution
  - 1.2 MPI
  - 0.0 Communication
  - 0.0 Collective
  - 12.2 P2P
  - 0.0 Late Receiver
  - 4.7 Late Sender
  - 0.0 IO
  - 0.0 Synchronization
  - 0.0 DMP
  - 0.0 Idle Threads

Dynamic Call Tree
- 0.0 main
  - 0.0 MPI_Init
  - 0.0 MPI_Ssend
  - 100.0 MPI_Rcv
  - 0.0 MPI_Comm_split
  - 0.0 MPI_Sendrecv
  - 0.0 MPI_Type_struct
  - 0.0 MPI_Type_commit
  - 0.0 MPI_Ssend
  - 0.0 MPI_Type_free
  - 0.0 MPI_Comm_free
  - 0.0 MPI_Finalize

Locations
- 0.0 Linux Cluster
  - 0.0 anakin02
    - 23.1 Process 2
  - 0.0 anakin04
    - 35.2 Process 3
  - 0.0 anakin05
    - 20.5 Process 0
  - 0.0 anakin06
    - 21.2 Process 1
Ongoing work
FAMP – Device
Fast Asynchronous Multi Protocol Device

User Application

MPI Library layer

FT-MPI runtime library

Derived Datatypes/
Buffer Management

Message Lists/
Non-blocking queues

Hlib

FAMP
TCP/IP
SHMEM
Myrinet PM/GM

Startup Daemon

HARNESS core

Notifier Service

Name Service
Current status

- FT-MPI currently implements
  - Whole MPI-1.2
  - Some parts of MPI-2
    - Language interoperability functions
    - Some external interfaces routines
    - Most of MPI-2 derived data-type routines
    - C++ Interface (Notre Dame)
    - Dynamic process management planned
  - ROMIO being tested
    - Non-ft version
  - Ported to both 32 and 64 bit OS
    - AIX, IRIX-6, Tru64, Linux-Alpha, Solaris, Linux
  - Compilers: Intel, gcc, pgi, vendor-compilers,
Distribution details

Distribution contains both HARNESS and FT-MPI source codes & documentation

» To build
   » Read the readme.1st file !
   » Configure
   » Make

» To use
   » set the environment variables as shown in the readme using the example ‘env’ stubs for bash/tcsh
   » Use the ‘console’ to start a DVM
      » Via a hostfile if you like
   » Compile & Link your MPI application with ftmpicc/ftmpif77/ftmpiCC
   » Run with ftmpirun or via the console
      » Type help if you need to know more

» Distribution contains the same solver using REBUILD, SHRINK and BLANK modes with and without MPI error-handlers
   » I.e. all combinations available in the form of a template
MPI and fault tolerance (cont.)

We need to fix it up here
... and here...
... and here...

You are here

User application

Parallel Load Balancer

Parallel Numerical Library

Do I=0, XXX

MPI_Sendrecv ()

.....
Suggestion for improved Error-handlers

» Application/libraries can replace an error handler by another error-handler

» Better: add an additional function which would be called in case of an error
  » e.g. like done for attribute caching or
  » e.g. unix atexit() function
MPI, IMPI, FT-MPI and the Grid

» IMPI: specifies the behavior of some MPI-1 functions
» MPI-2 dynamic process management would profit from an extended IMPI protocol (IMPI 2.0 ?)

» MPI-2 one-sided operations a powerful interface, feasible for the Grid
» MPI-2 language interoperability functions
» MPI-2 canonical pack/unpack functions, specification of external32 data representation

» Further Grid specific functionalities:
  » Which process is on which host ?
    » e.g. MPI-2 JoD Cluster attributes
    » specification of what MPI_Get_processor_name()
Summary

- HARNESS is an alternative GRID environment for collaborative sharing of resources
  - Application level support is via plug-ins such as FT-MPI, PVM etc

- Fault tolerance for MPI applications is an active research target
  - Large number of models and implementations available
  - Semantics of FT-MPI is very close to the current specification
  - Design of FT-MPI is in the “spirit” of MPI

- FT-MPI first full release is by Supercomputing 2003
  - Beta release by end of the month
  - Seeking for early beta-tester
Contact information

» Links and contacts

» HARNESS and FT-MPI at UTK/ICL
   http://icl.cs.utk.edu/harness/

» HARNESS at Emory University
   http://www.mathcs.emory.edu/harness/

» HARNESS at ORNL
   http://www.epm.ornl.gov/harness/