

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



Innovative Computing Laboratory

COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF TENNESSEE

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 handle 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

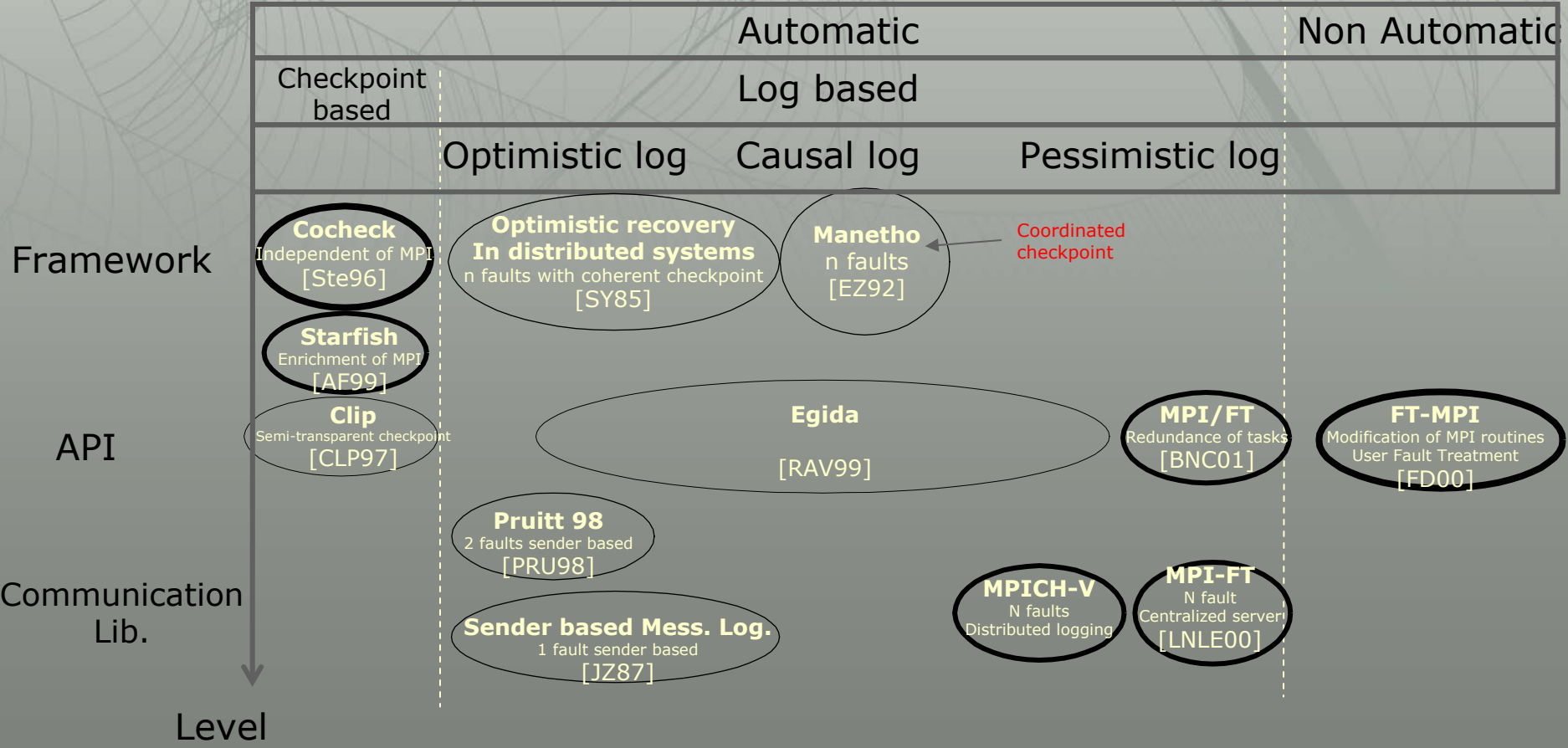
Message logging: none, pessimistic, optimistic, causal.

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



No automatic/transparent, n fault tolerant, scalable message passing environment

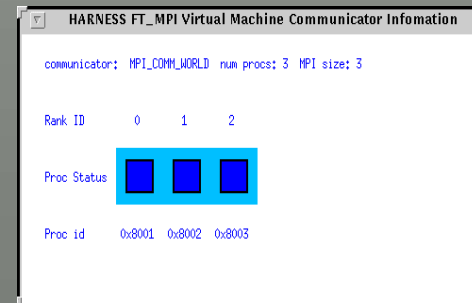
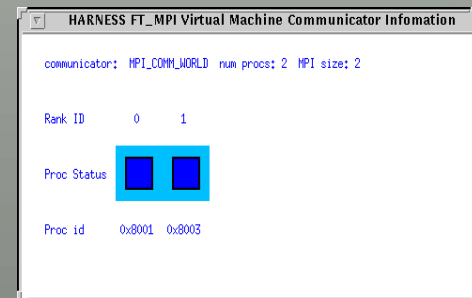
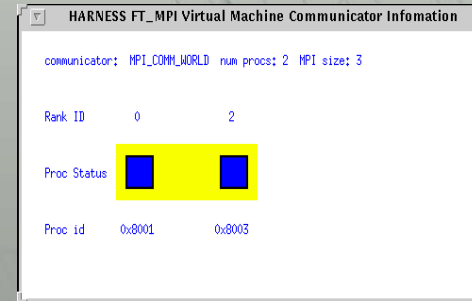
FT-MPI

- » 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

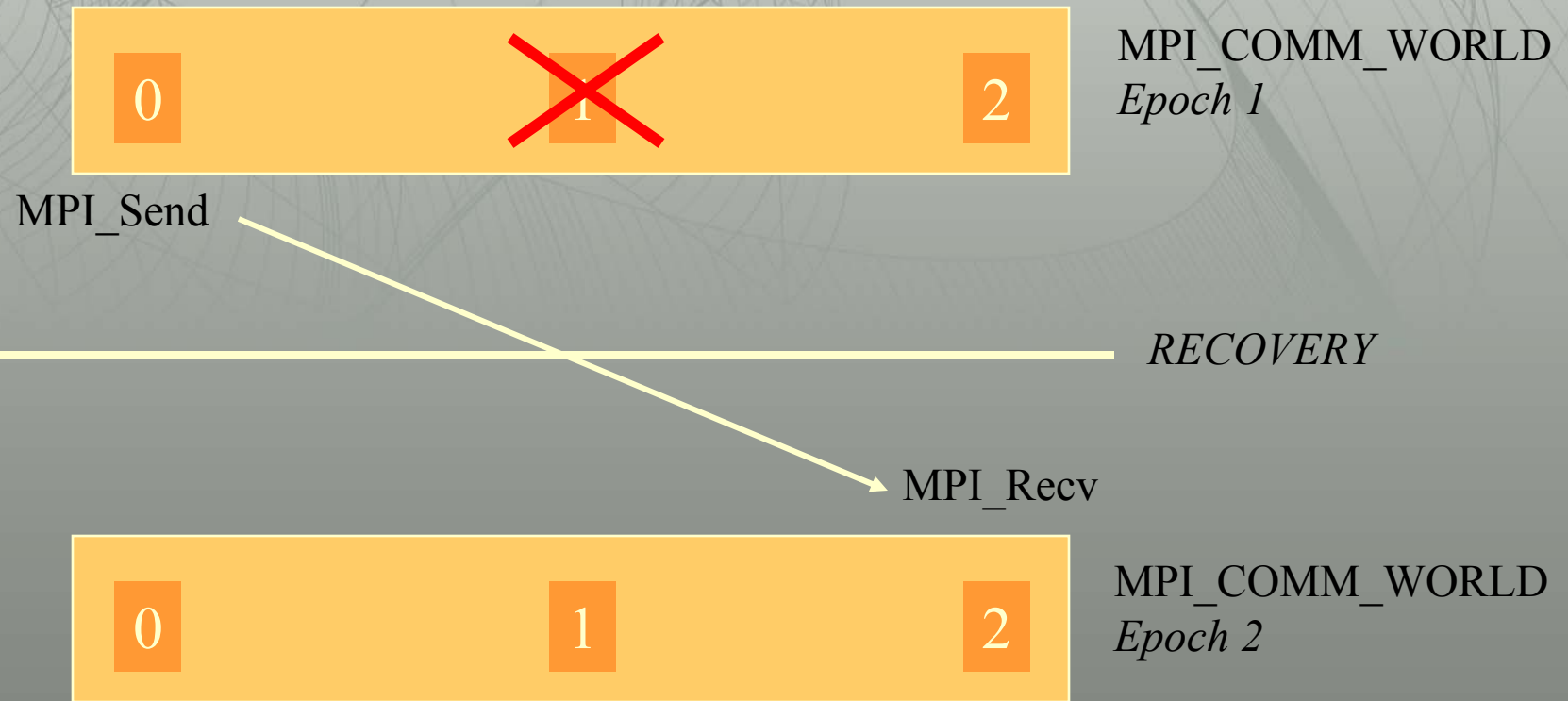
- » **ABORT**: just do as other implementations
- » **BLANK**: leave hole
- » **SHRINK**: re-order processes to make a contiguous communicator
 - » Some ranks change
- » **REBUILD**: re-spawn lost processes and add them to MPI_COMM_WORLD



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 and CONTINUE



RESET: a message posted in one epoch does not match a receive posted in another epoch

CONTINUE: epoch argument not regarded for message matching

Frequently asked questions

Question: How do we know a process has failed ?

Answer: The return code of an MPI operation returns MPI_ERR_OTHER. The failed process is not necessarily the process involved in the current operation!

Question: What do I have to do if a process has failed ?

Answer: 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

Point – to – point semantics (I)

- » 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 operations 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 (...)
```



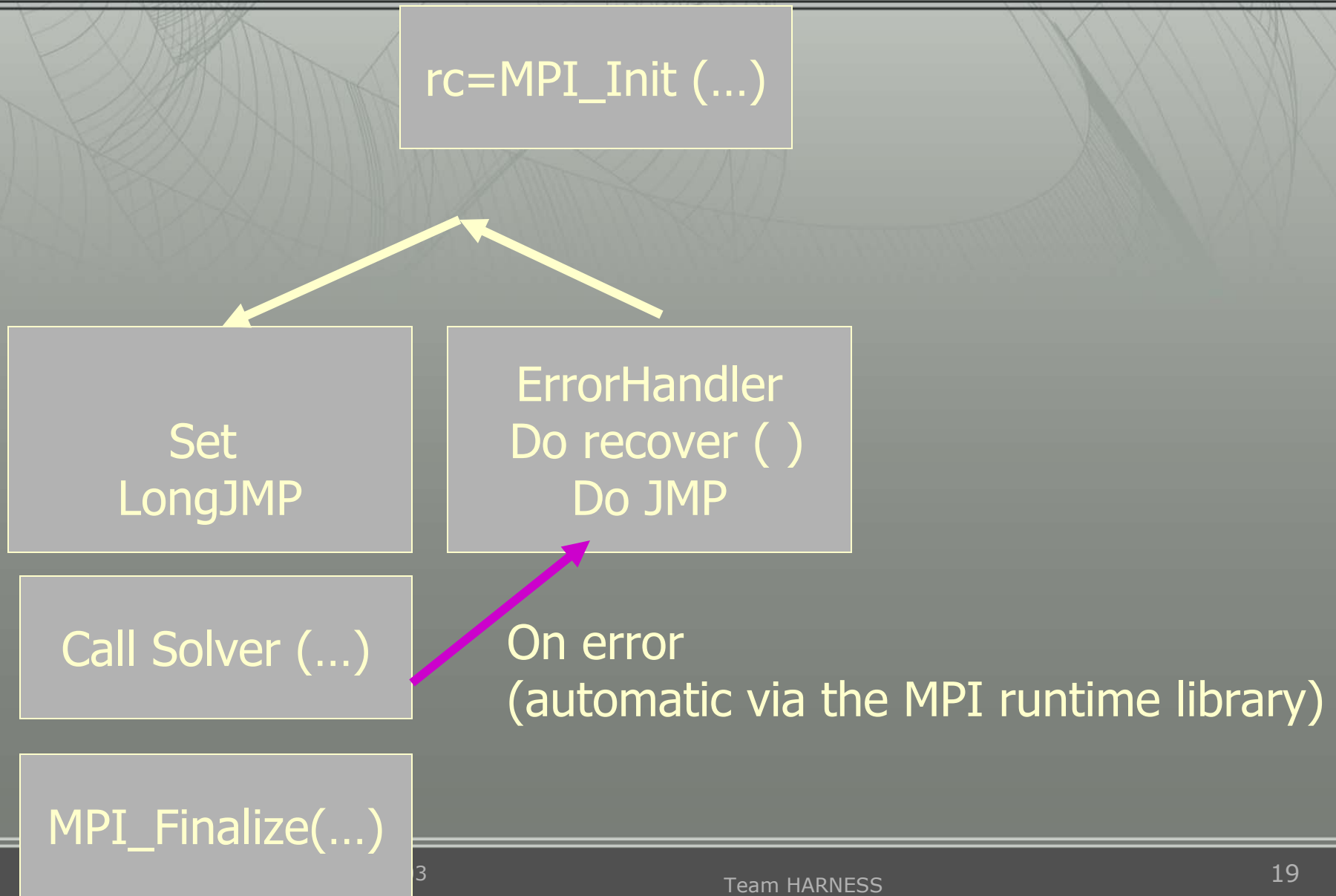
If normal startup

```
Install Error  
Handler & Set  
LongJMP
```

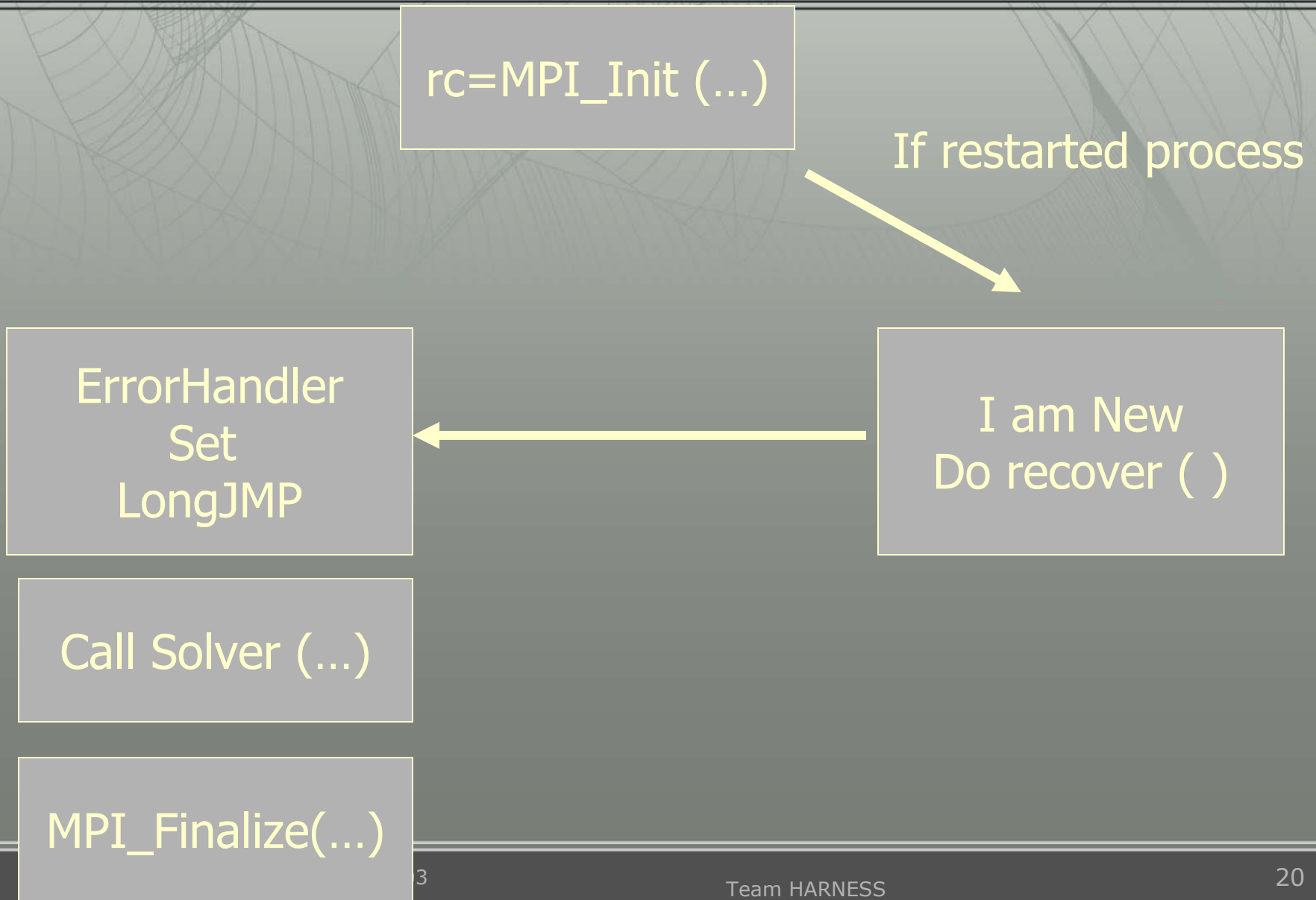
```
Call Solver (...)
```

```
MPI_Finalize(...)
```

Application scenario



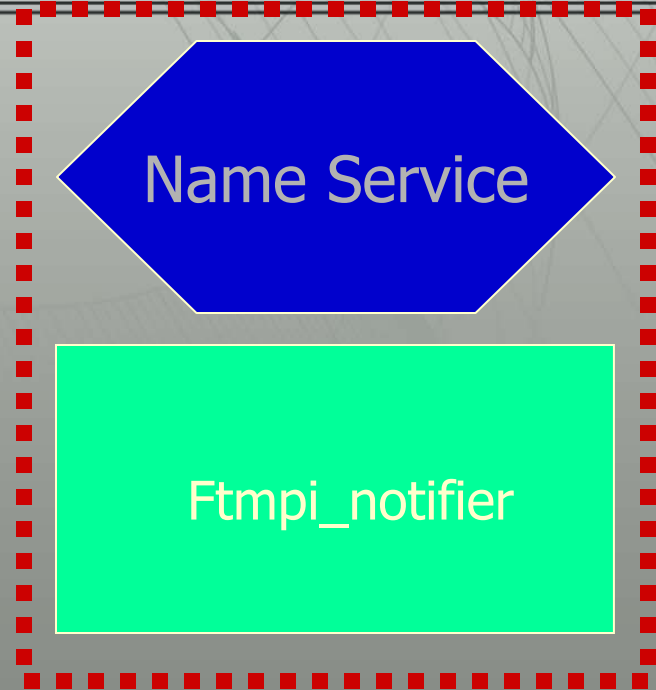
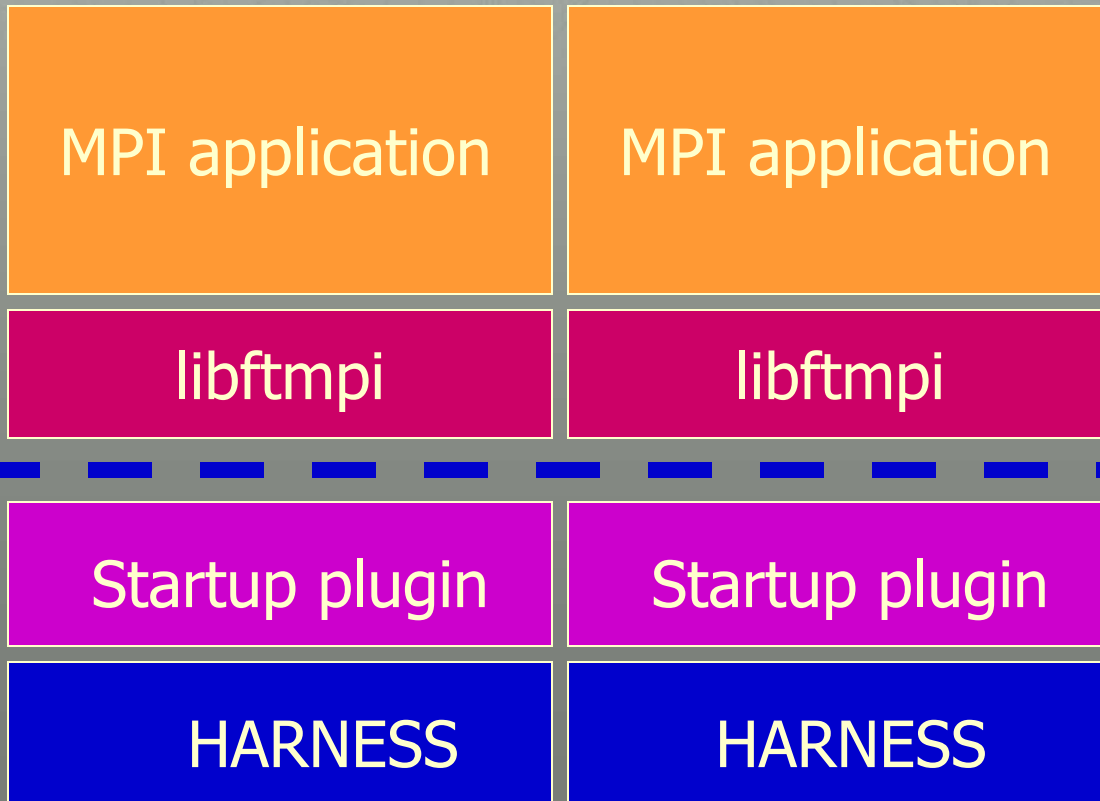
Application scenario



Architecture

High level services

Running under a HARNESS Core

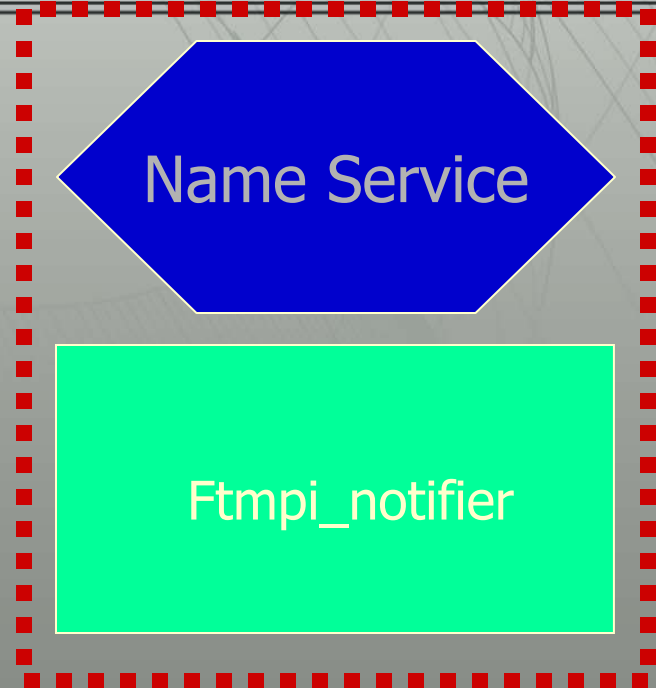
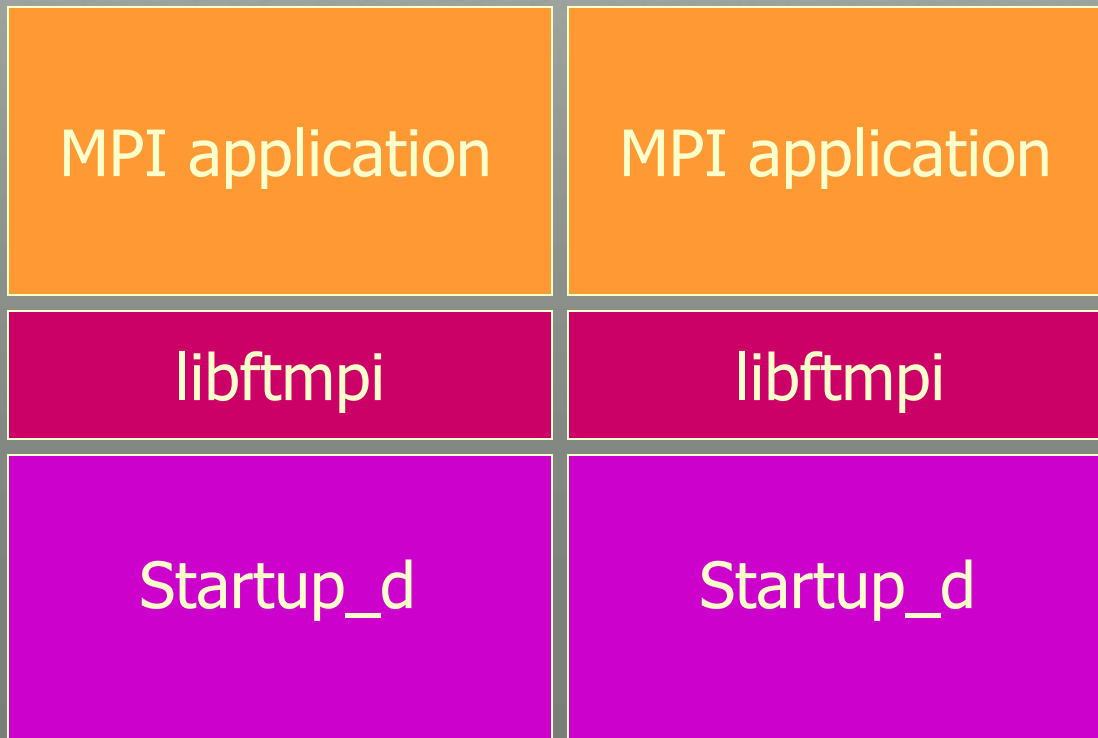


One startup plug-in per 'core'

Architecture

High level services

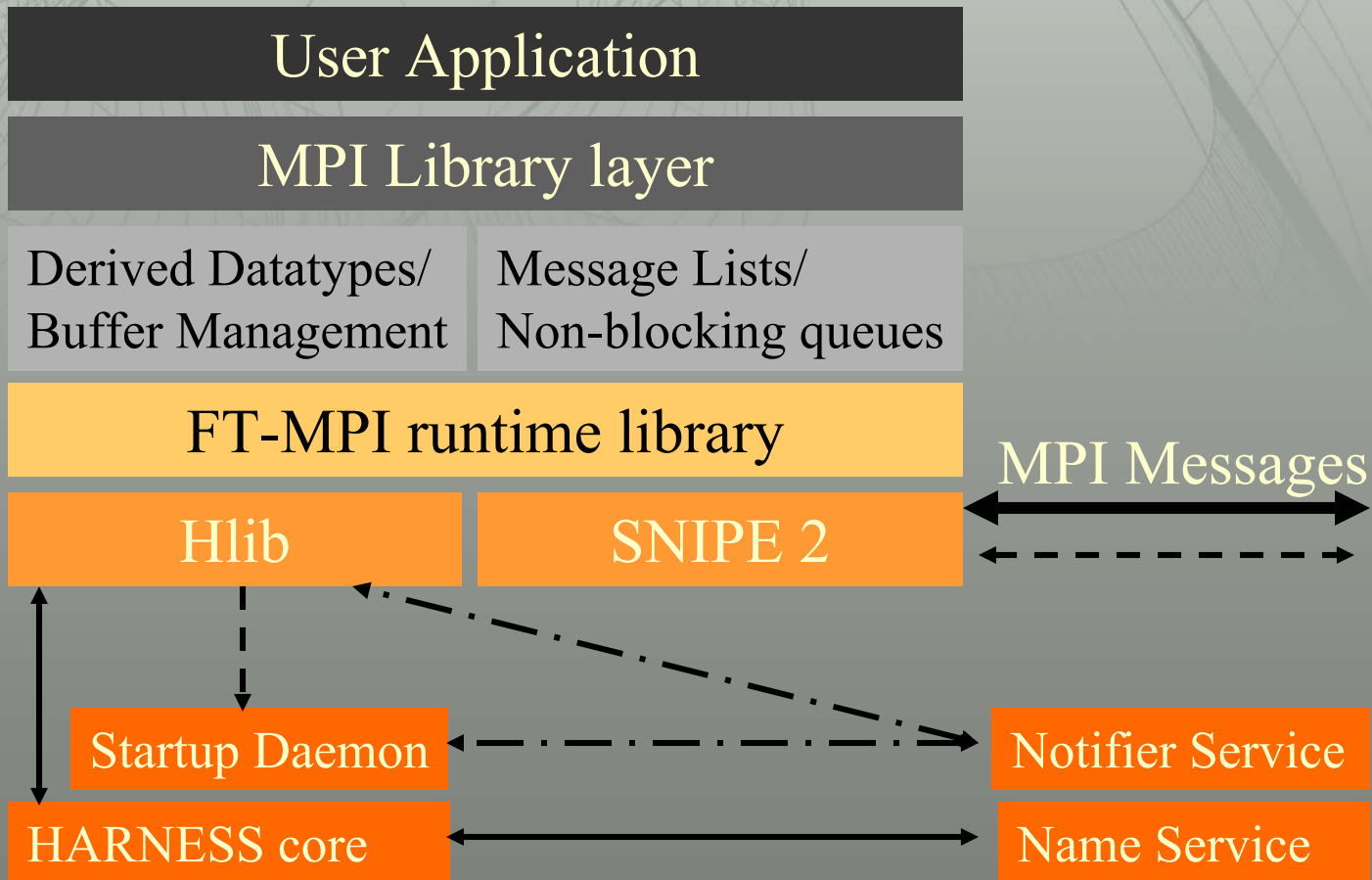
Running outside of a Core



— — — —
One startup daemon per 'host'

Implementation details

Implementation Details



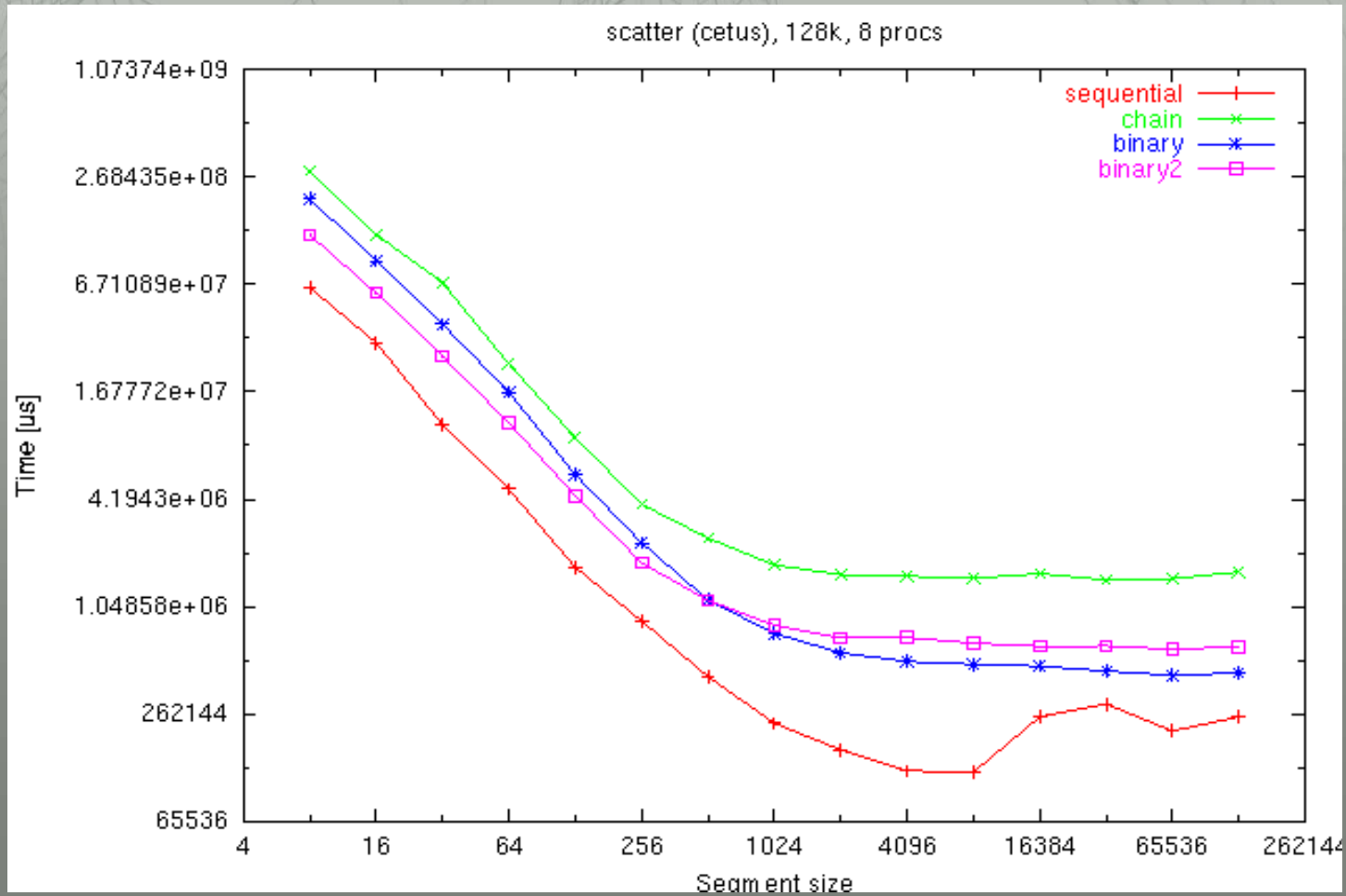
Implementation Details

- » 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



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

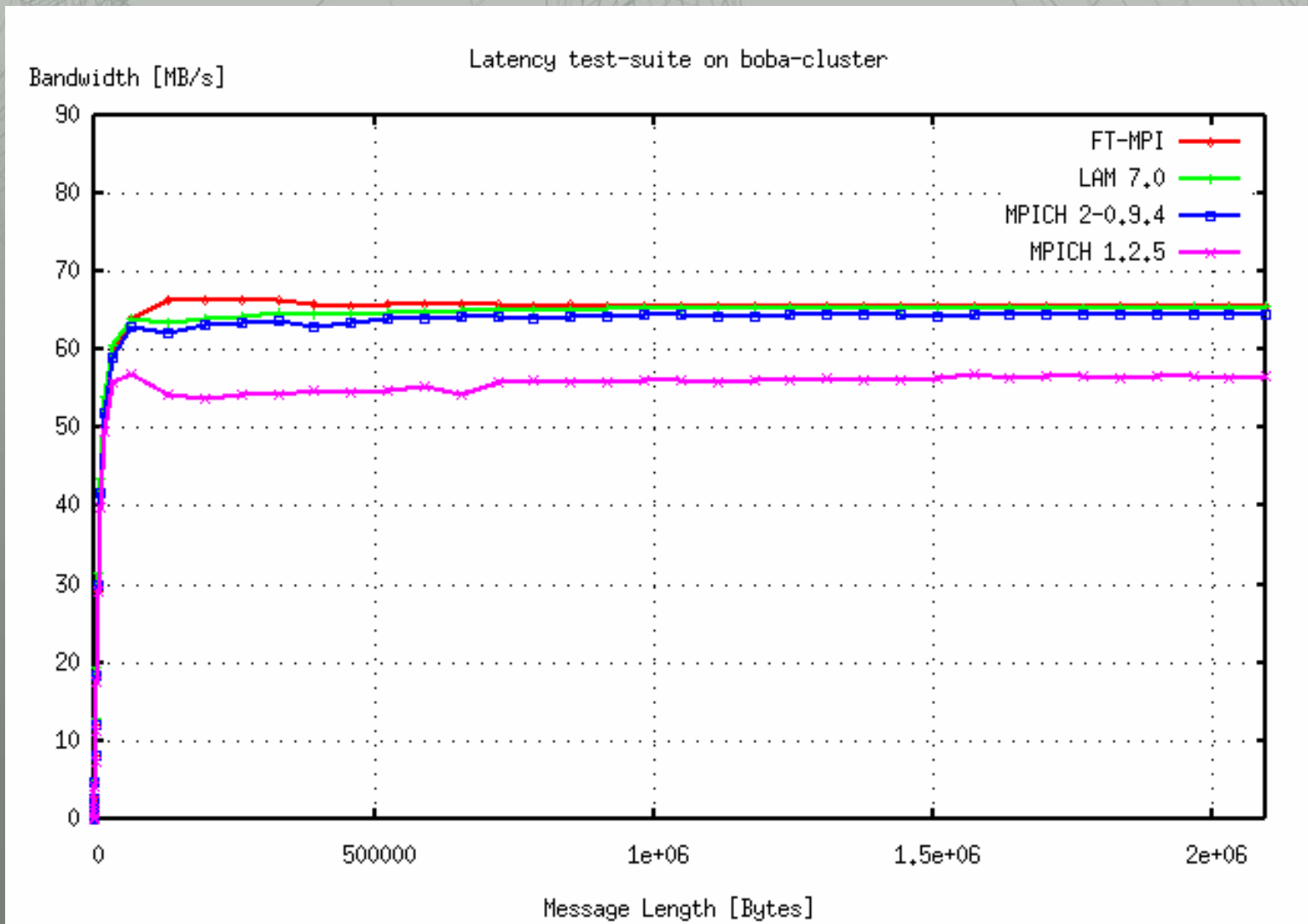
Data conversion

- » 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

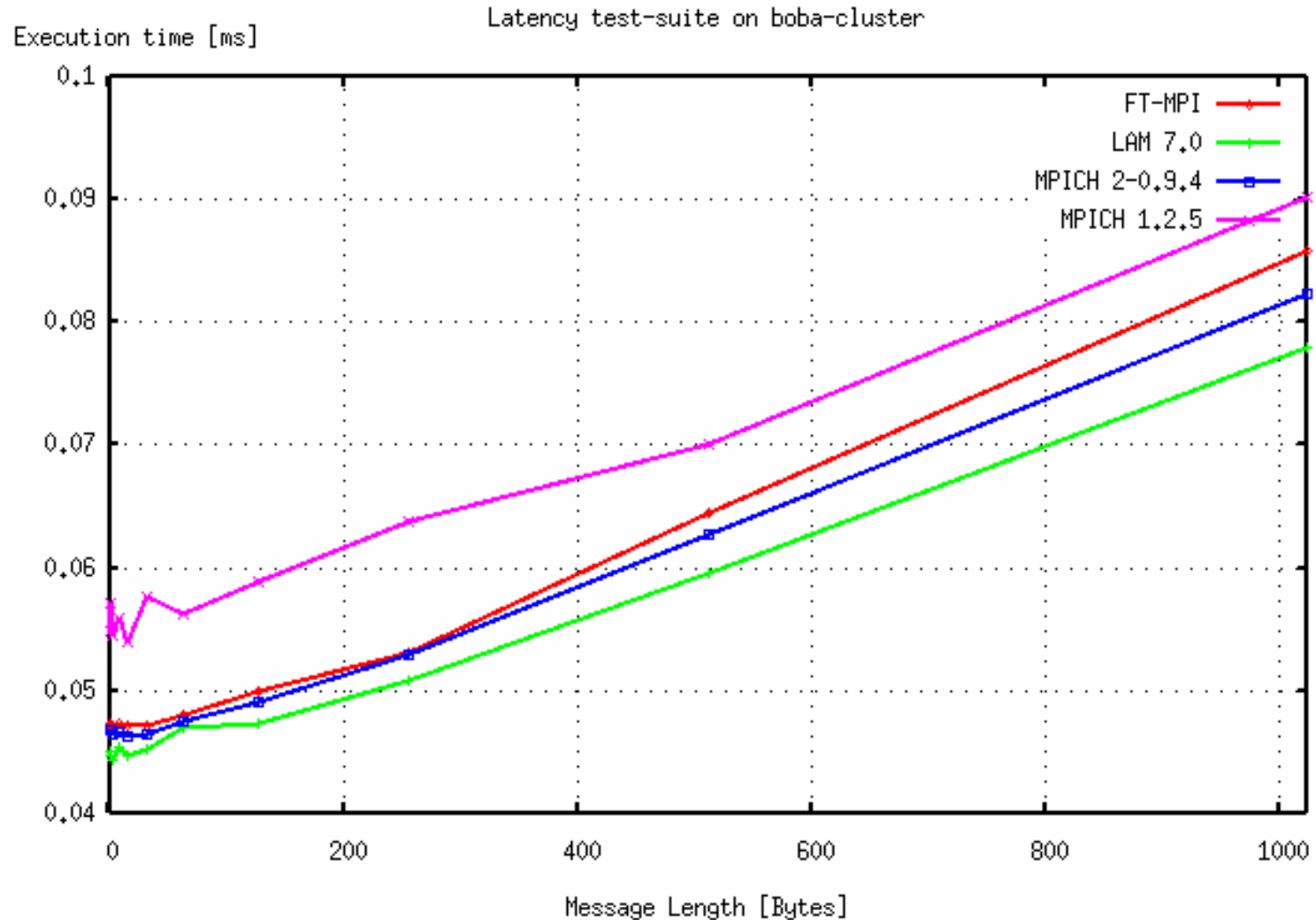
Performance comparison

- » 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

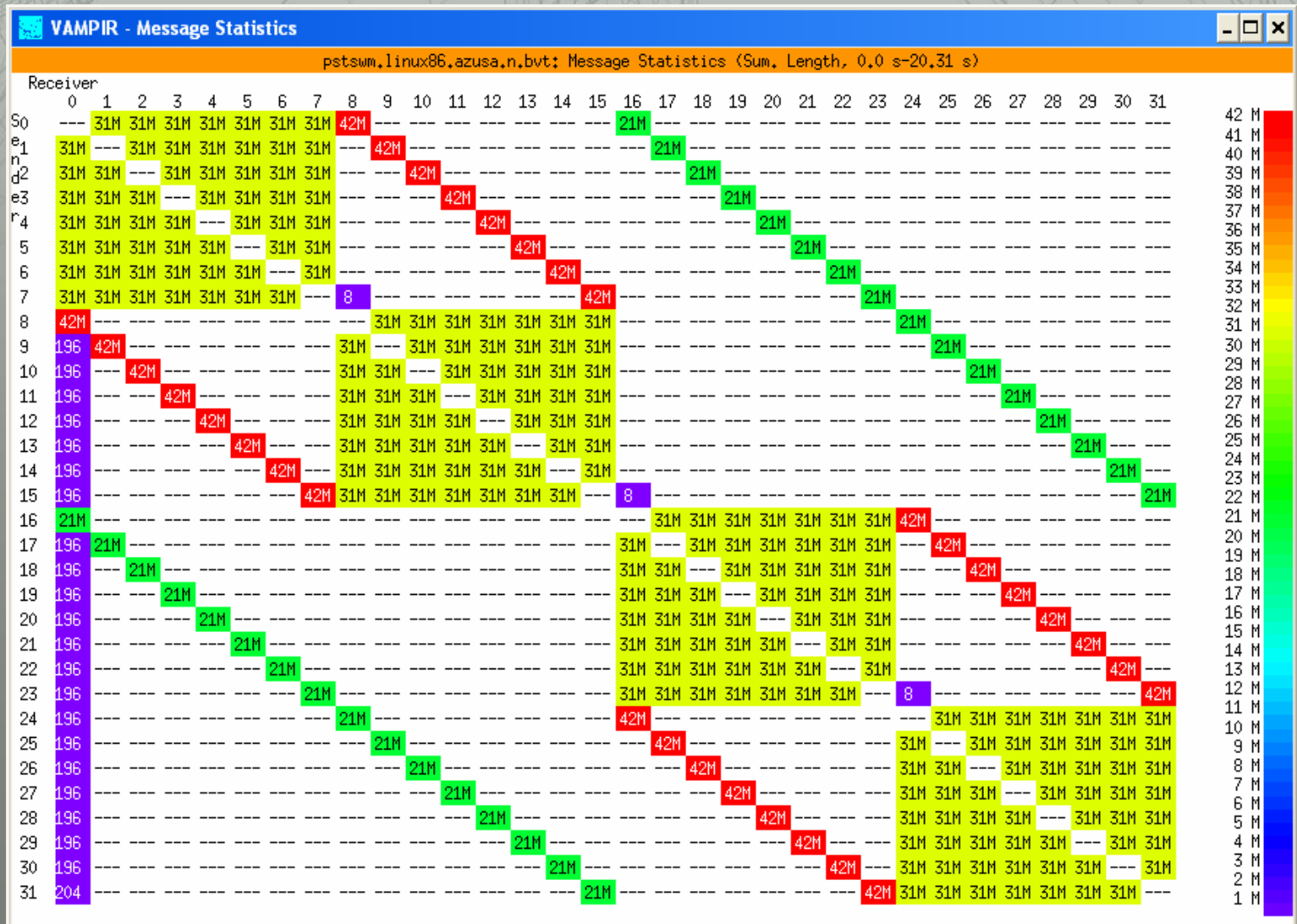
	MPICH 1.2.5 [sec]	MPICH 2 – 0.9.4 [sec]	FT-MPI [sec]
t42.l16.240	40.76	36.60	35.79
t42.l17.240	43.20	38.88	38.01
t42.l18.240	45.69	41.11	40.32
t85.l16.24	33.65	30.77	30.50
t85.l17.24	36.02	32.75	32.31
t85.l18.24	38.14	34.64	34.25
t170.l1.12	9.01	8.22	8.03
t170.l2.12	17.78	16.35	16.14
t170.l3.12	26.76	24.53	24.18

Shallow Water Code (PSTSWM)

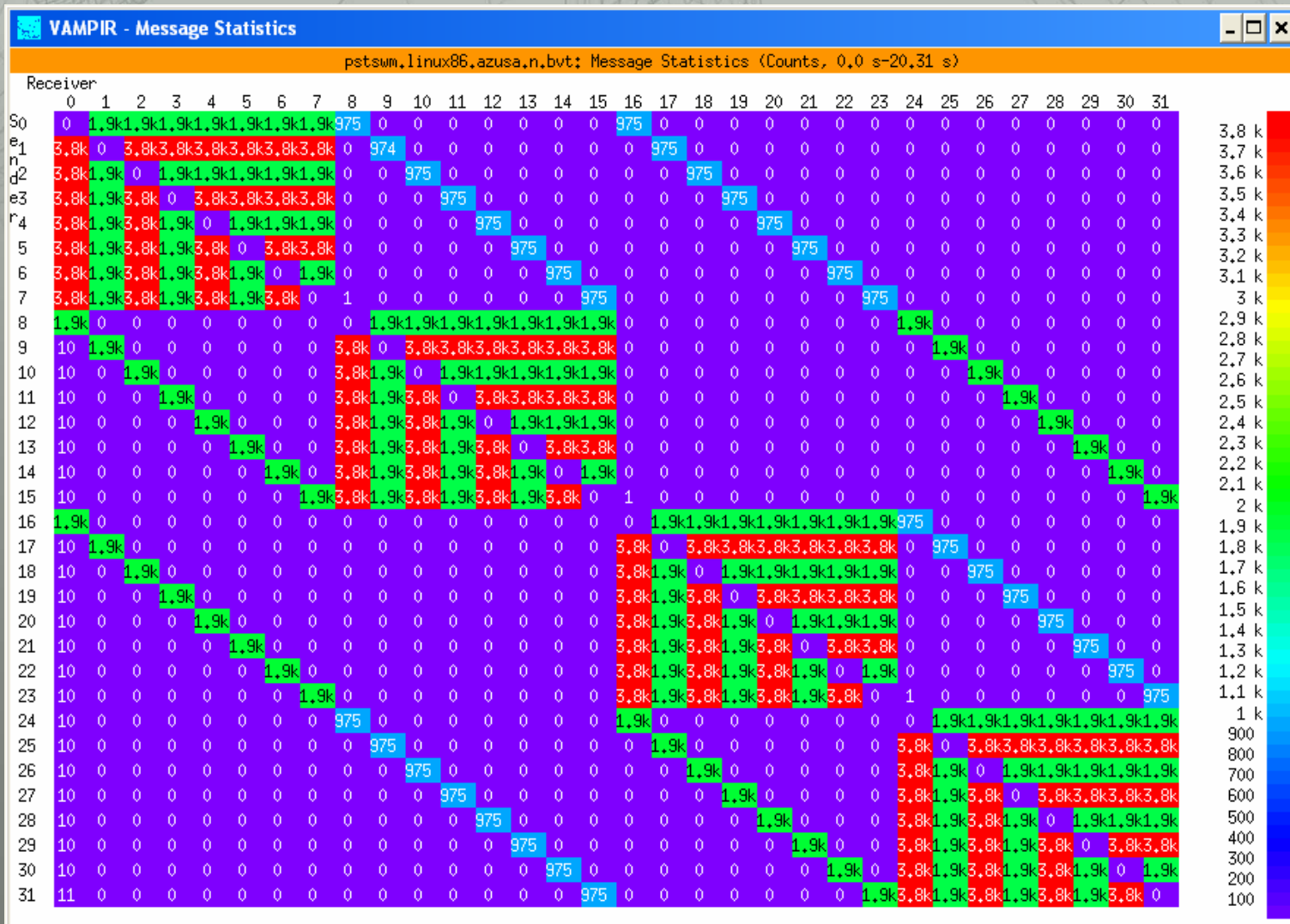
32 processes

	MPICH 1.2.5 [sec]	MPICH 2 – 0.9.4 [sec]	FT-MPI [sec]
t42.l16.240	40.69	38.00	30.33
t42.l17.240	43.41	40.59	32.35
t42.l18.240	45.89	42.50	34.20
t85.l16.24	33.54	31.54	26.20
t85.l17.24	35.57	33.34	27.84
t85.l18.24	37.63	35.11	29.57
t170.l1.12	8.69	8.29	6.58
t170.l2.12	17.33	16.54	13.29
t170.l3.12	25.92	24.52	20.16

Communication pattern 32 processes



Number of messages



FT-MPI using the same proc. distribution like MPICH 1.2.5 and MPICH 2

	MPICH 1.2.5 [sec]	MPICH 2 – 0.9.4 [sec]	FT-MPI [sec]	FT-MPI prev. [sec]
t42.l16.240	40.69	38.00	36.79	30.33
t42.l17.240	43.41	40.59	39.27	32.35
t42.l18.240	45.89	42.50	41.37	34.20
t85.l16.24	33.54	31.54	31.22	26.20
t85.l17.24	35.57	33.34	32.93	27.84
t85.l18.24	37.63	35.11	34.99	29.57
t170.l1.12	8.69	8.29	7.88	6.58
t170.l2.12	17.33	16.54	15.96	13.29
t170.l3.12	25.92	24.52	23.77	20.16

MPICH 1.2.5 using the same process distribution like FT-MPI

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

HPL Benchmark

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

Blocksize	MPICH 1.2.5 [sec]	MPICH 2 0.9.4 [sec]	LAM 7.0 [sec]	FT-MPI [sec]
48	28.16	27.50	27.85	27.95
80	18.18	17.40	17.81	18.10
240	19.88	19.15	19.58	20.03

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

```
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)

```
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

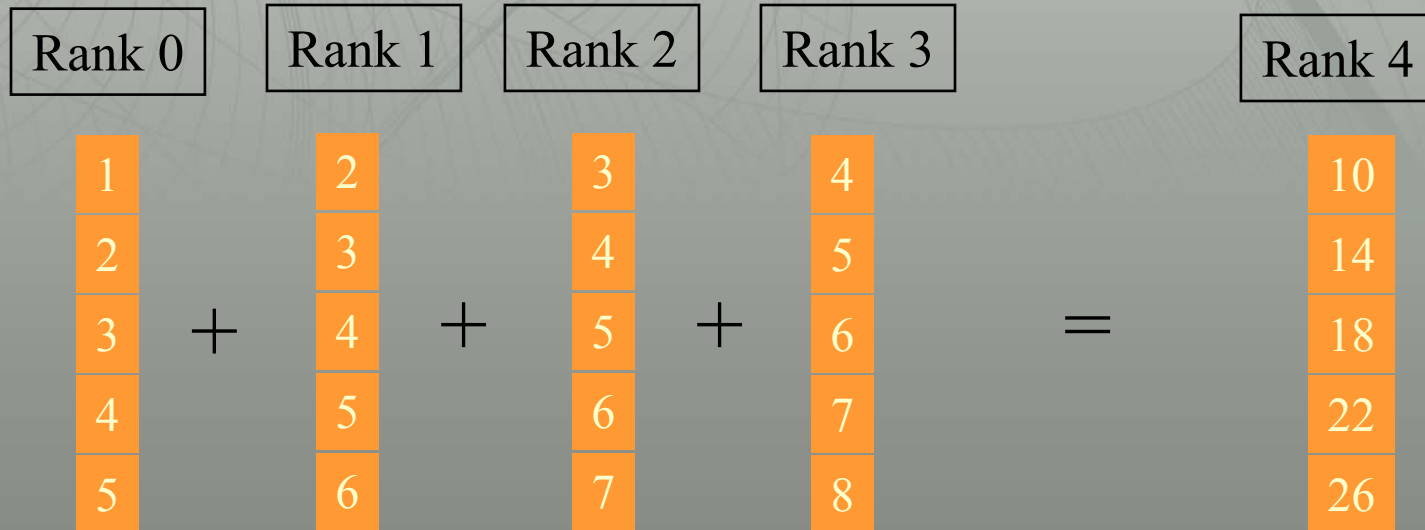
- » Tightly coupled
- » Can be used for all positive-definite, RSA-matrices in the Boeing-Harwell format
- » Do a "backup" every n iterations

- » 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



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

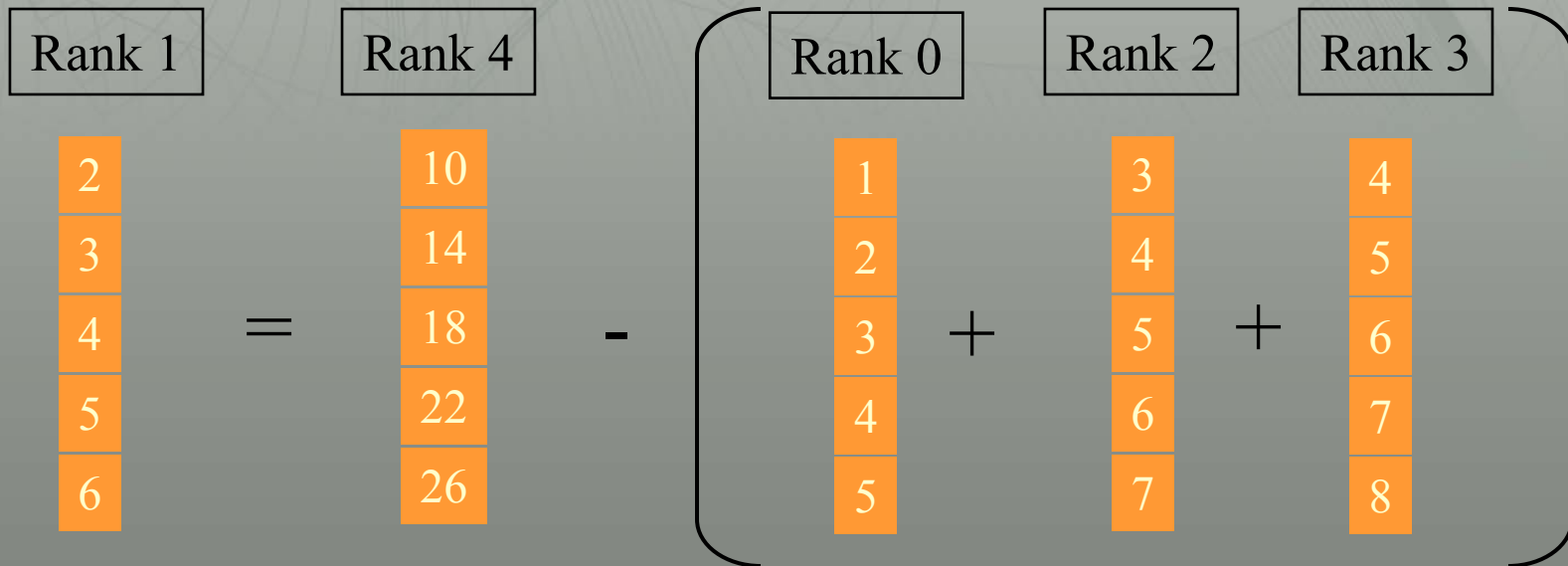
If your application shall survive two process failures

Rank 0	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5
$x_{1,1}$	$x_{2,1}$	$x_{3,1}$	$x_{4,1}$	=	
1	2	3	4		10
2	3	4	5		14
3	4	5	6		18
4	5	6	7		22
5	6	7	8		26
$x_{1,2}$	$x_{2,2}$	$x_{3,2}$	$x_{4,2}$	=	...
1	2	3	4		...
2	3	4	5		...
3	4	5	6		...
4	5	6	7		...
5	6	7	8		...

with x determined as in the Red-Solomon Algorithm

The recovery procedure

- » Rebuild work-communicator
- » Recover data



- » Reset iteration counter
- » On each process: copy backup of vector v into the current version

PCG overall structure

```
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

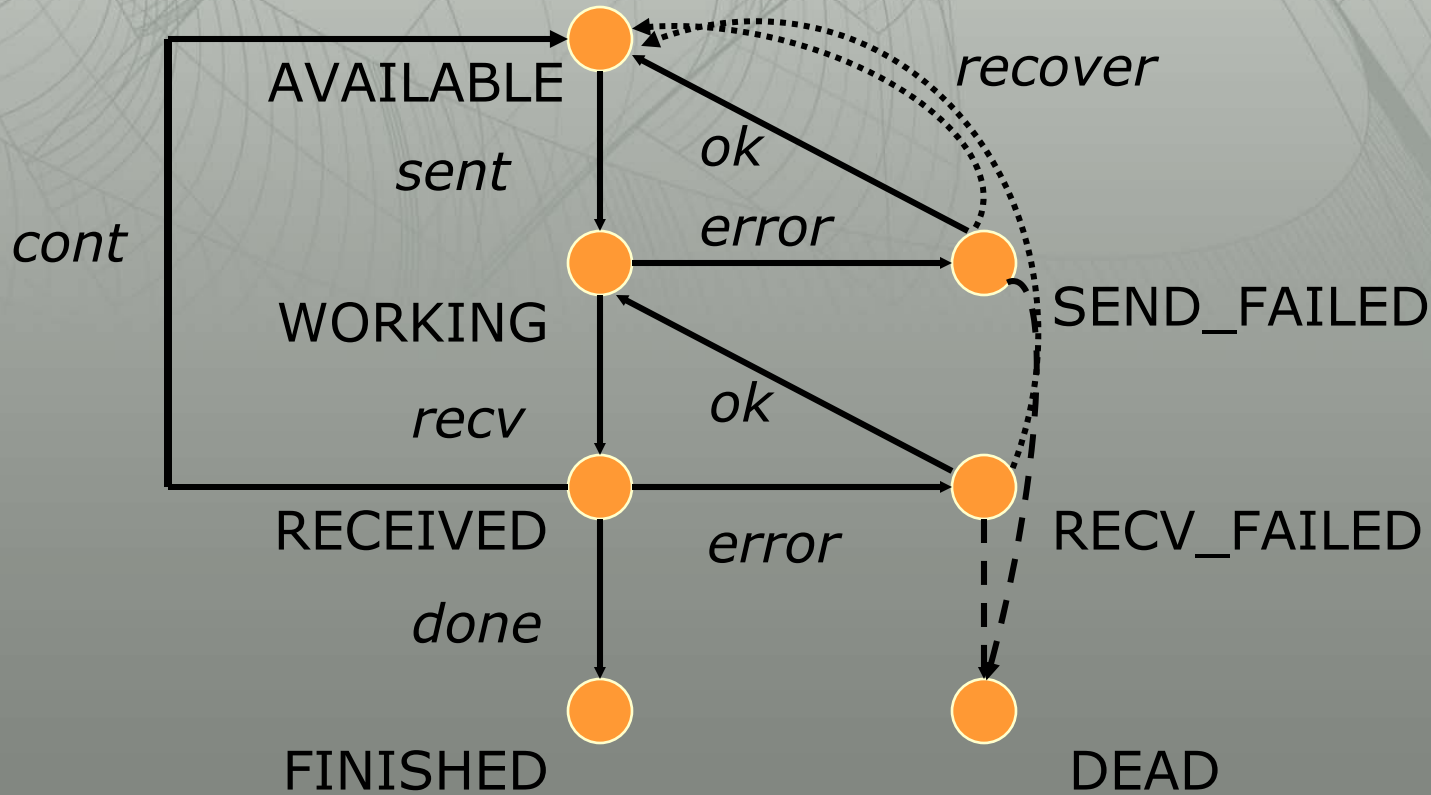
Problem size	Number of processes	Execution time	Recovery time	Ratio
4054	4 + 1	5 sec	1.32 sec	26.4%
428650	8 + 1	189 sec	3.30 sec	1.76%
2677324	16 + 1	1162 sec	11.4 sec	0.97%

Raw FT-MPI recovery times on Pentium IV G Ethernet 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)

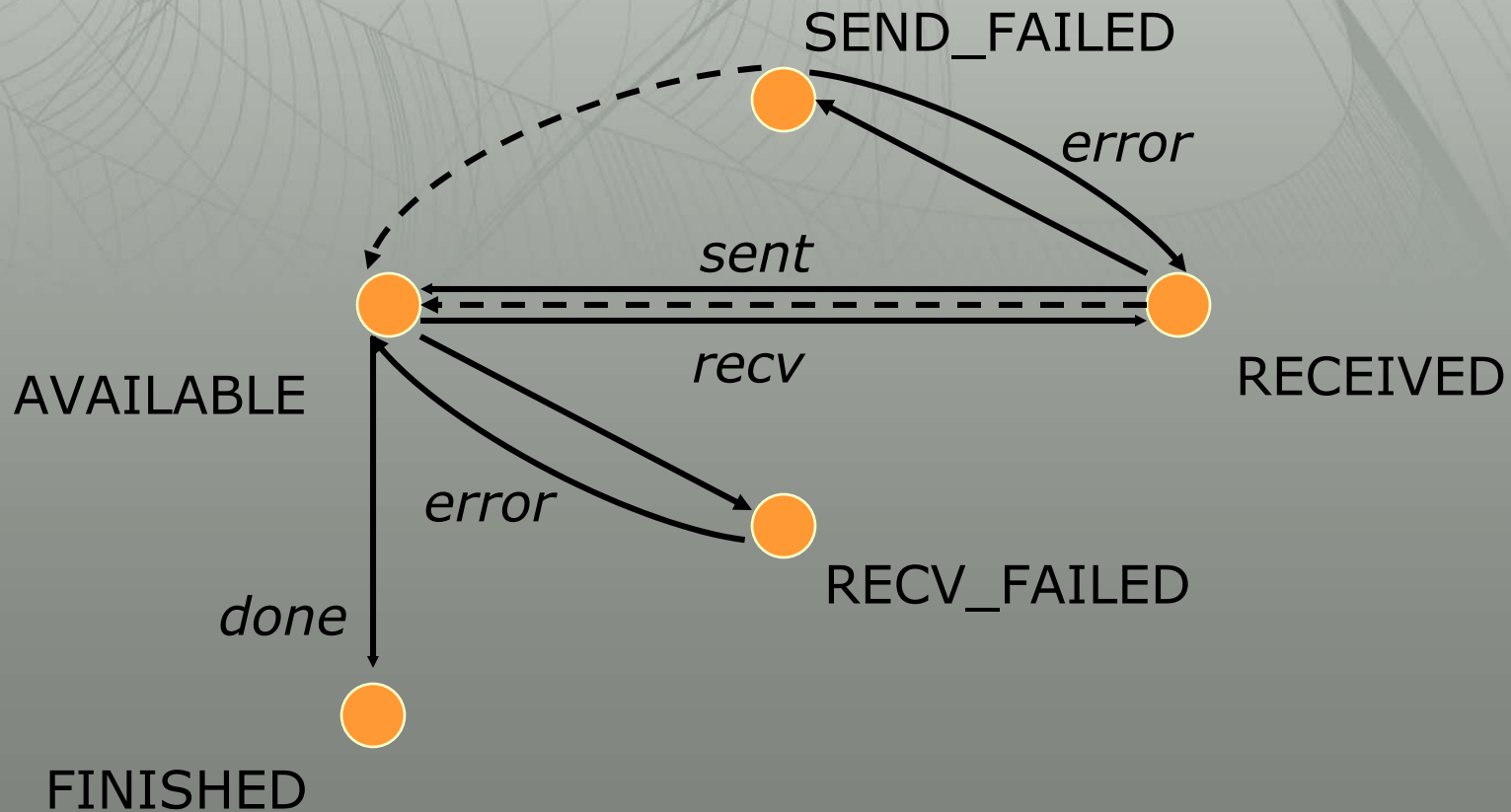
Master process: transition-state diagram



-----> BLANK/SHRINK: mark failed processes as DEAD

.....> REBUILD: mark failed processes as AVAILABLE

Worker process: transition-state diagram



-----> REBUILD: Master died, reset state to AVAILABLE

Master-Slave: user interface

» Abstraction of user interface:

```
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

Harness console

```
LINUX>console
```

```
Welcome to the Harness/FT-MPI console
```

```
con> conf
```

```
Found 4 hosts
```

HostID	HOST	PORT
0	torc4.cs.utk.edu	22503
1	torc1.cs.utk.edu	22503
2	torc2.cs.utk.edu	22500
3	torc3.cs.utk.edu	22502

```
con>ps
```

ProcID	RunID	HostID	Command	Comment	Status	Time
4096	20482	0	./bmtest	FTMPI:proces	exited(val:0)	5s
4097	20482	1	./bmtest	FTMPI:proces	exited(val:0)	5s

```
con>
```

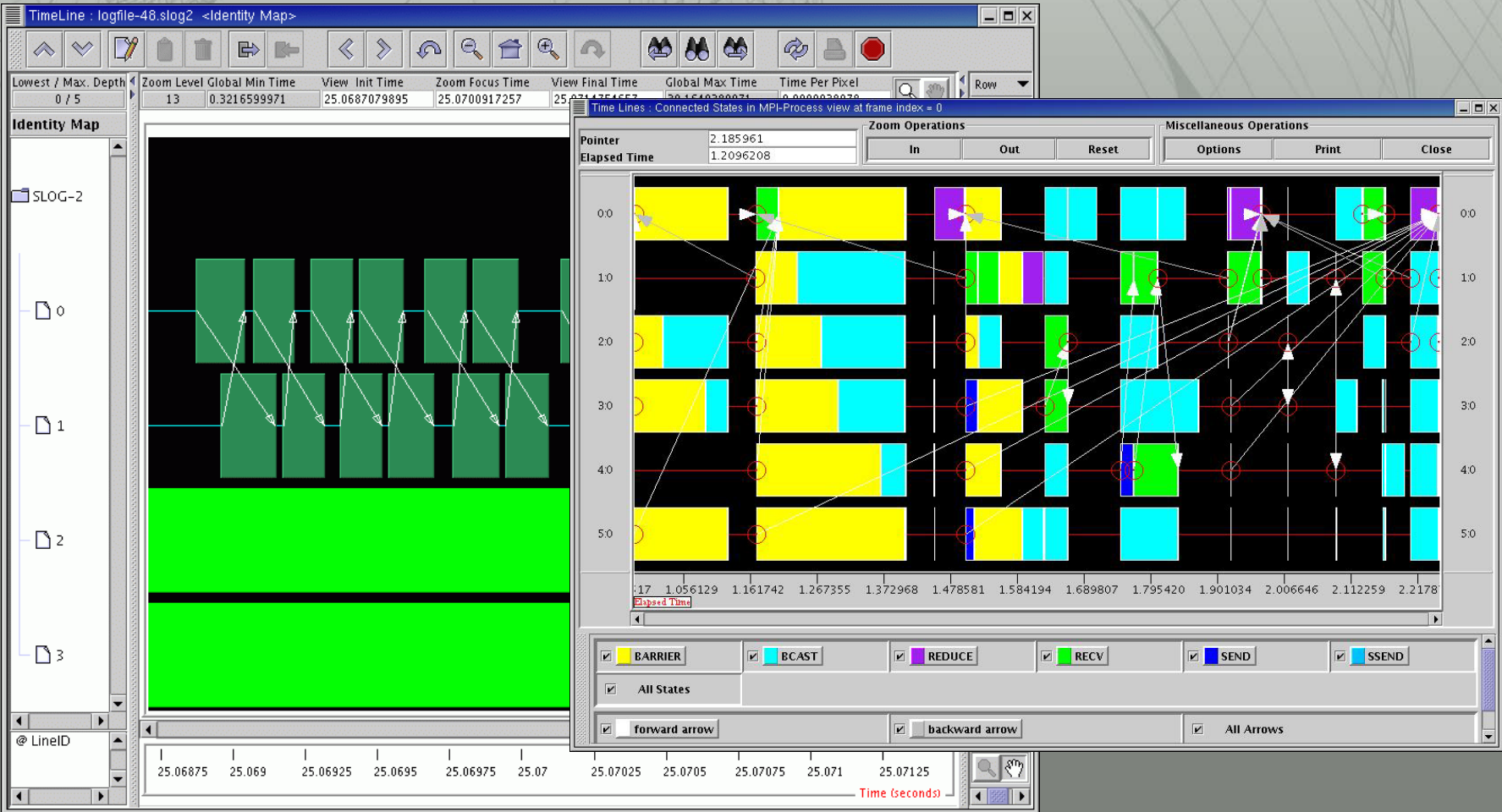
Harness Virtual Machine Monitor

The screenshot displays the Harness Virtual Machine Monitor (HVM) interface. It features three main windows:

- Tree View:** A hierarchical tree structure showing the organization of Distributed Virtual Machines (DVMs). The root is 'DVM', which branches into 'Hcores', 'Services', and 'Jobs'. Under 'Hcores', there are four core configurations (HCore-2, HCore-3, HCore-4), each containing 'STD' and 'neverending' sub-items. 'Services' also contains 'STD' and 'neverending' sub-items for each core. 'Jobs' contains a list of 'neverending' sub-items.
- DVM Setup Window:** A configuration window for a specific DVM named 'DVM boba411'. It shows fields for 'NameService', 'Host' (boba411.sinrg.cs.utk.edu), and 'Port' (1968). There are buttons for 'Update NS Info' and 'Validate NS', and 'Add DVM' and 'Remove DVM' at the bottom.
- Main Monitor Window:** Displays a network diagram of a 'Virtual Machine' (represented by a large black sphere) connected to several smaller nodes (green and grey spheres) via red lines. The text 'DVM' and coordinates 'X: 254 Y: 81' are visible in the top right of this window.

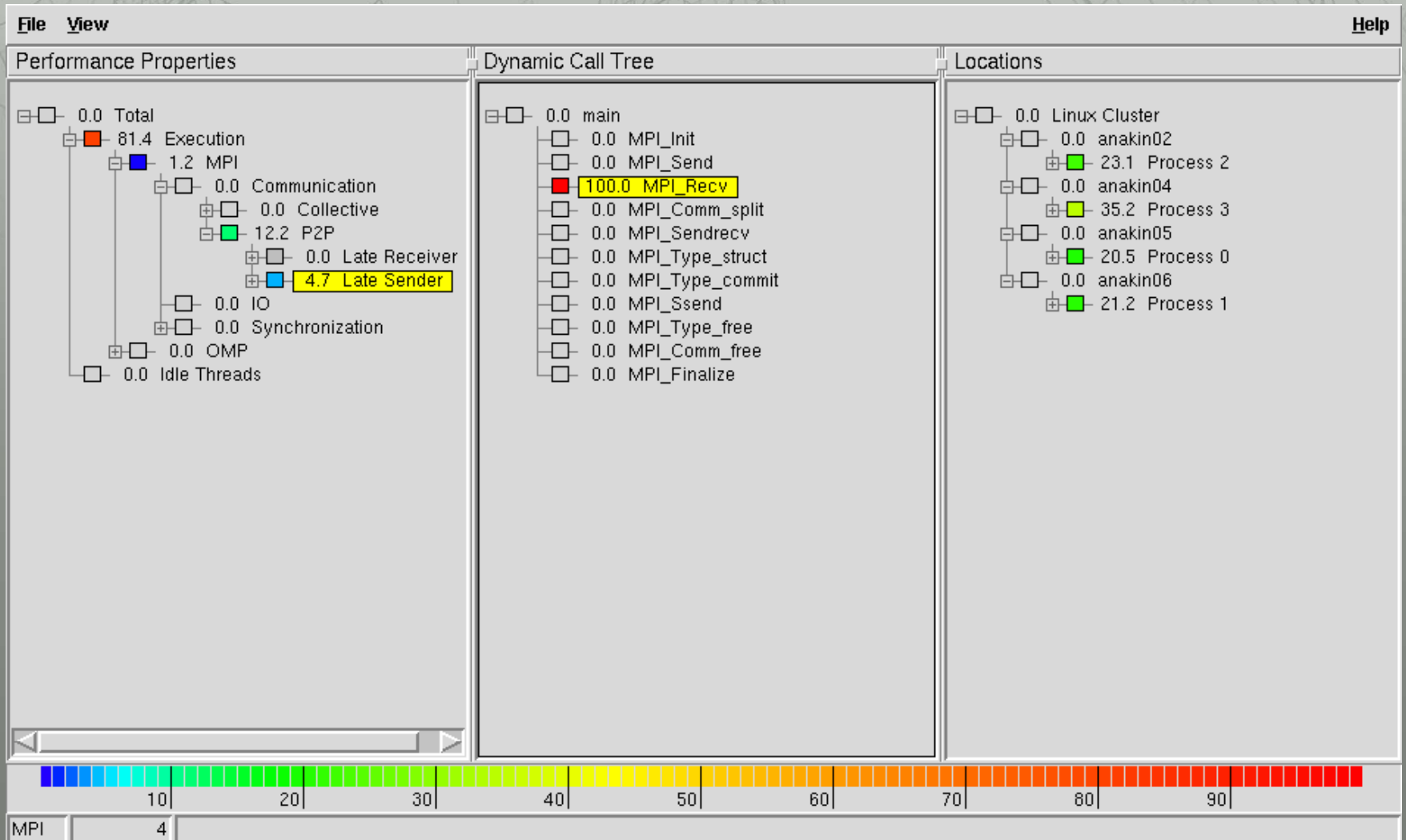
The Windows taskbar at the bottom shows the Start button and several open applications, including 'idmasc (68.47.179.12...', 'FTMPI_gdfs - Micros...', 'Builder 8 - C:\work\j...', 'C:\work\java\jmon...', 'Harness Virtual Mach...', and 'Paint Shop Pro'. The system clock shows 11:27 AM.

MPE with FT-MPI



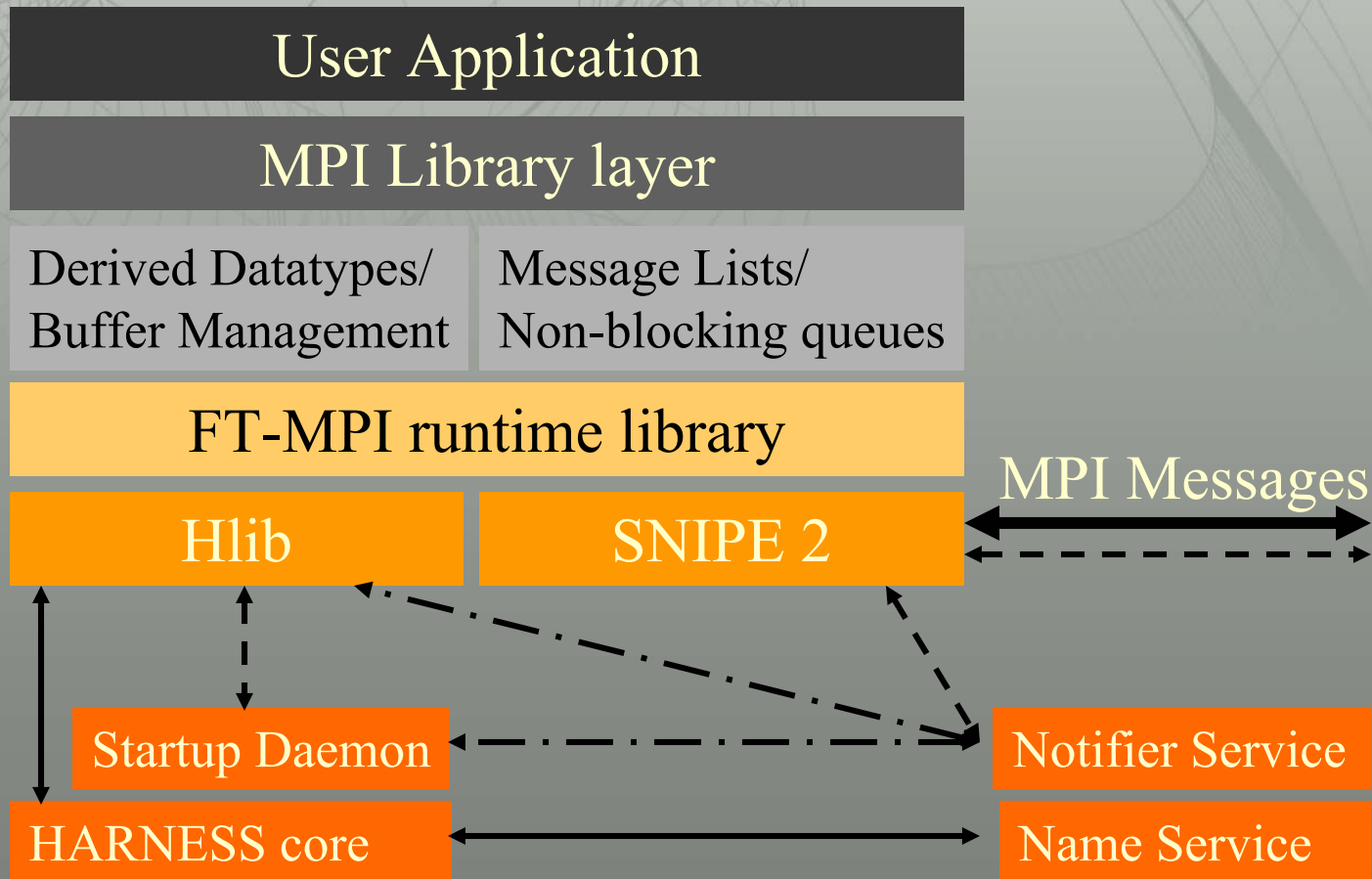
FT-MPI HPL trace via Jumpshot using MPE profiling

KOJAK/PAPI/Expert



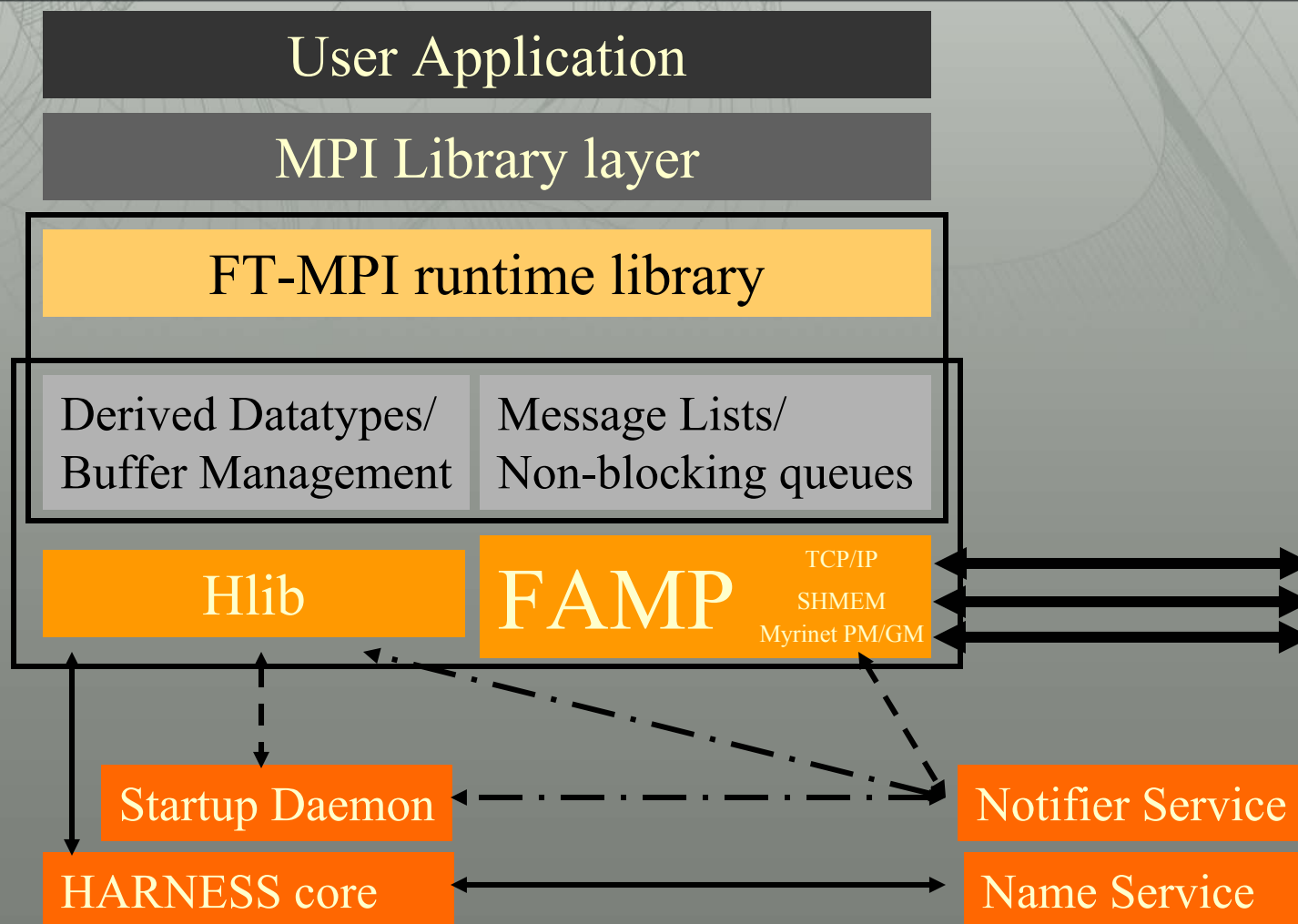
Ongoing work

Current design



FAMP – Device

Fast Asynchronous Multi Protocol Device



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

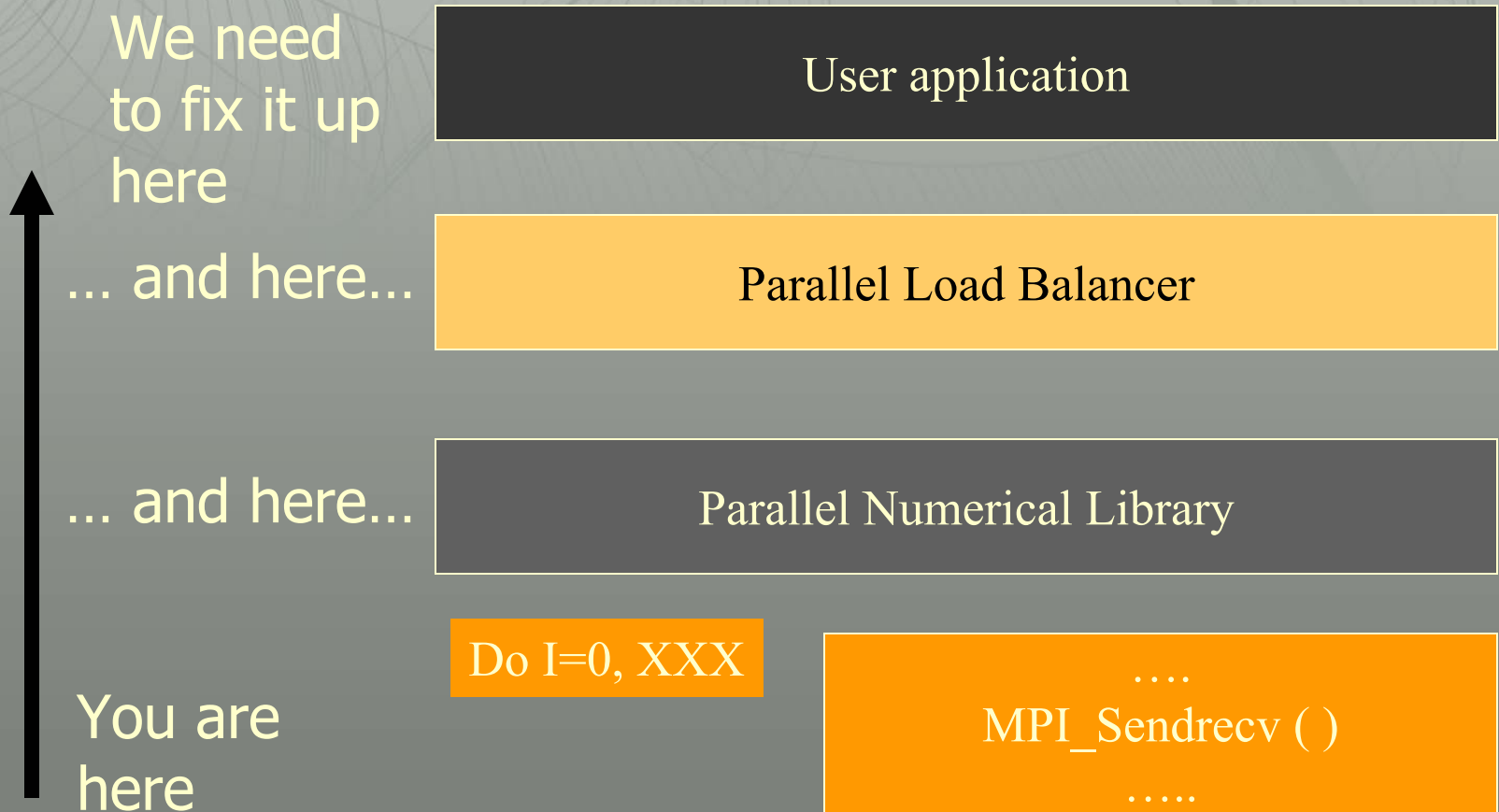
Distribution contains both HARNESS and FT-MPI source codes & documentation

- » 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.)



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 a 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/>