



SOFTWARE

- + 19.56 updatex
- + 399.70 updateien
- + 0.00 gene
- 0.00 <<iteration loop>>
- + 447.52 genbc

PRODUCTIVITY



FAST SOLUTIONS

- PAPI_L1_ICM
- PAPI_L2_DCM
- PAPI_L2_ICM
- PAPI_L1_TCM

MPI Runtime Error Detection with MUST

For the 12th VI-HPS Tuning Workshop

Joachim Protze and Felix Münchhalfen
CCC, RWTH Aachen University
Oktober 2013

- MPI Usage Errors
- Error Classes
- Avoiding Errors
- Correctness Tools
- Runtime Error Detection
- MUST
- Hands On

- MPI programming is error prone
- Bugs may manifest as:
 - Crashes
 - Hangs
 - Wrong results
 - Not at all! (Sleeping bugs)
- Simple Example:

```
MPI_Type_contiguous (2, MPI_INT, &newtype);  
MPI_Send (buf, count, newtype, target, tag, MPI_COMM_WORLD);
```

Error: Usage of un-comitted datatype

- Tools help to pin-point these bugs

- Complications in MPI usage:
 - Non-blocking communication
 - Persistent communication
 - Complex collectives (e.g. Alltoallw)
 - Derived datatypes
 - Non-contiguous buffers
- Error Classes include:
 - Incorrect arguments
 - Resource errors
 - Buffer usage
 - Type matching
 - Deadlocks

- MPI Usage Errors
- **Error Classes**
- Avoiding Errors
- Correctness Tools
- Runtime Error Detection
- MUST
- Hands On

- Complications
 - Calls with many arguments
 - In Fortran many arguments are of type INTEGER
 - Several restrictions for arguments of some calls⇒ Compilers can't detect all incorrect arguments
- Example:

```
MPI_Send(  
  buf,  
  count,  
  MPI_INTEGER,  
  target,  
  tag,  
  MPI_COMM_WORLD);
```

- Complications
 - Many types of resources
 - Leaks
 - MPI internal limits
- Example:

```
MPI_Comm_dup (MPI_COMM_WORLD, &newComm);  
MPI_Finalize ();
```

- Complications
 - Memory regions passed to MPI must not overlap (except send-send)
 - Derived datatypes can span non-contiguous regions
 - Collectives can both send and receive
- Example:

```
MPI_Isend (&(buf[0]), 5 /*count*/, MPI_INT, ...);  
MPI_Irecv (&(buf[4]), 5 /*count*/, MPI_INT, ...);
```


- Complications
 - Complex derived types
 - Types match if the signature matches, not their constructors
 - Partial receives
- Example 1:

Task 0

```
MPI_Send (buf, 1, MPI_INT);
```

Task 1

```
MPI_Recv (buf, 1, MPI_INT);
```

- Matches => Equal types match

- Example 2:

- Consider type $T1 = \{MPI_INT, MPI_INT\}$

Task 0

```
MPI_Send (buf, 1, T1);
```

Task 1

```
MPI_Recv (buf, 2, MPI_INT);
```

- Matches => type signatures are equal

- Example 3:

- $T1 = \{MPI_INT, MPI_FLOAT\}$

- $T2 = \{MPI_INT, MPI_INT\}$

Task 0

```
MPI_Send (buf, 1, T1);
```

Task 1

```
MPI_Recv (buf, 1, T2);
```

- Mismatch => $MPI_INT \neq MPI_FLOAT$

- Example 4:

- T1 = {MPI_INT, MPI_FLOAT}
- T2 = {MPI_INT, MPI_FLOAT, MPI_INT}

Task 0

```
MPI_Send (buf, 1, T1);
```

Task 1

```
MPI_Recv (buf, 1, T2);
```

- Matches => MPI allows partial receives

- Example 4:

- T1 = {MPI_INT, MPI_FLOAT}
- T2 = {MPI_INT, MPI_FLOAT, MPI_INT}

Task 0

```
MPI_Send (buf, 2, T1);
```

Task 1

```
MPI_Recv (buf, 1, T2);
```

- Mismatch => Partial send is not allowed

- Complications:
 - Non-blocking communication
 - Complex completions (Wait{all, any, some})
 - Non-determinism (e.g. MPI_ANY_SOURCE)
 - Choices for MPI implementation (e.g. buffered MPI_Send)
 - Deadlocks may be caused by non-trivial dependencies
- Example 1:

Task 0

MPI_Recv (from:1);

Task 1

MPI_Recv (from:0);

- Deadlock: 0 waits for 1, which waits for 0

- How to visualise/understand deadlocks?
 - Common approach waiting-for graphs (WFGs)
 - One node for each rank
 - Rank X waits for rank Y \Rightarrow node X has an arc to node Y
- Consider situation from Example 1:

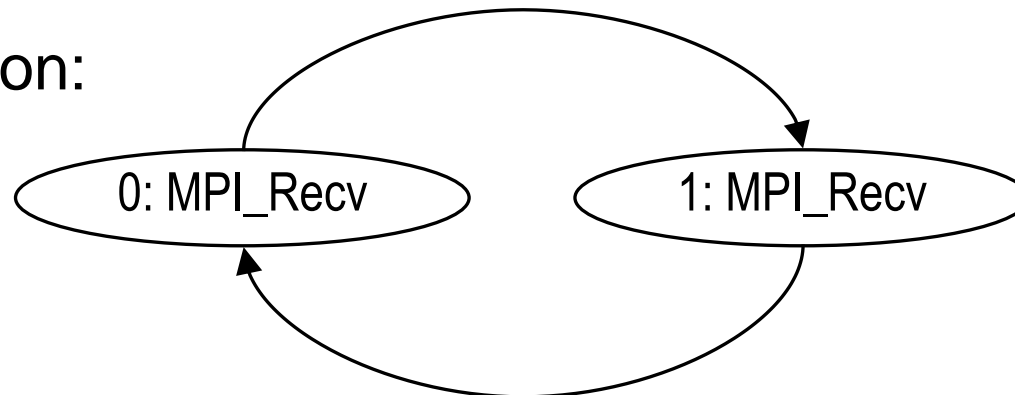
Task 0

Task 1

MPI_Recv (from:1);

MPI_Recv (from:0);

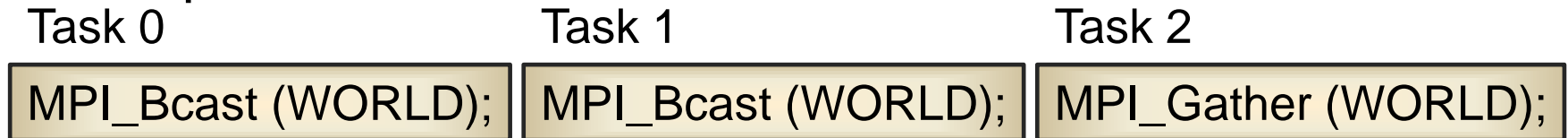
- Visualization:



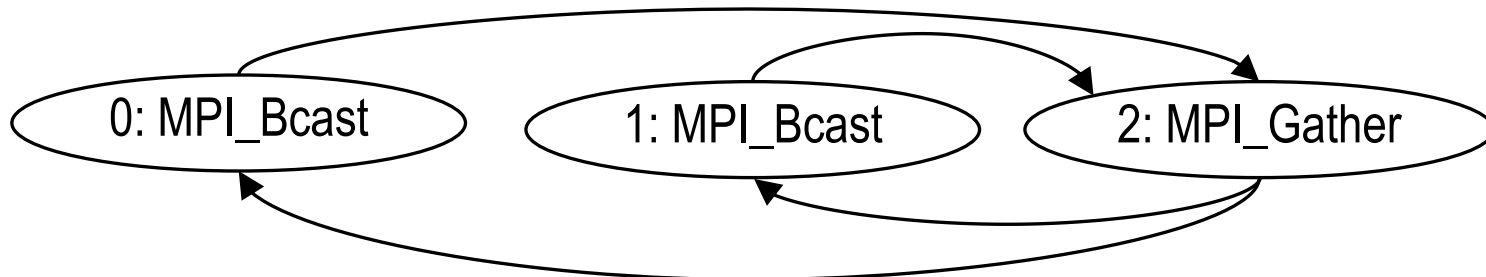
- Deadlock criterion: cycle (For simple cases)

- What about collectives?
 - Rank calling coll. operation waits for all tasks to issue a matching call
 - ⇒ One arc to each task that did not call a matching call
 - One node potentially has multiple outgoing arcs
 - Multiple arcs means: waits for all of the nodes

- Example 2:

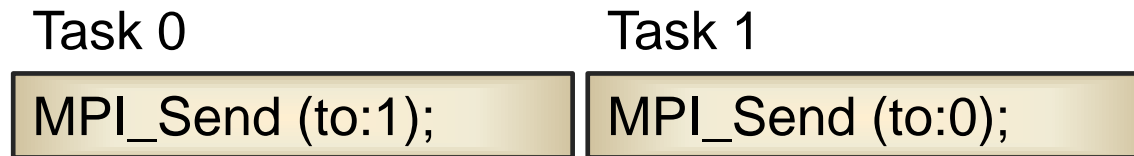


- Visualization:

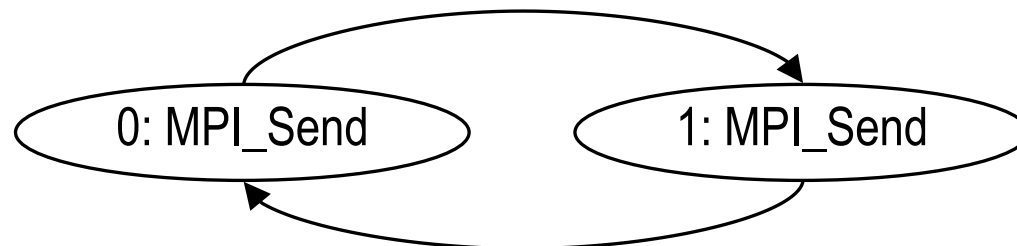


- Deadlock criterion: cycle (Also here)

- What about freedom in semantic?
 - Collectives may not be synchronizing
 - Standard mode send may (or may not) be buffered
- Example 3:



- This is a deadlock!
 - These are called “potential” deadlocks
 - Can manifest for some implementations and/or message sizes
- Visualization:



- What about timely interleaving?
 - Non-deterministic applications
 - Interleaving determines what calls match or are issued
 - Causes bugs that only occur “sometimes”

- Example 3:

Task 0	Task 1	Task 2
<pre>MPI_Send(to:1) MPI_Barrier()</pre>	<pre>MPI_Recv(from:ANY); MPI_Recv(from:2) MPI_Barrier()</pre>	<pre>MPI_Send(to:1) MPI_Barrier()</pre>

- What happens:
 - Case A:
 - ◆ Recv (from:ANY) matches send from task 0
 - ◆ All calls complete
 - Case B:
 - ◆ Recv (from:ANY) matches send from task 2
 - ◆ Deadlock

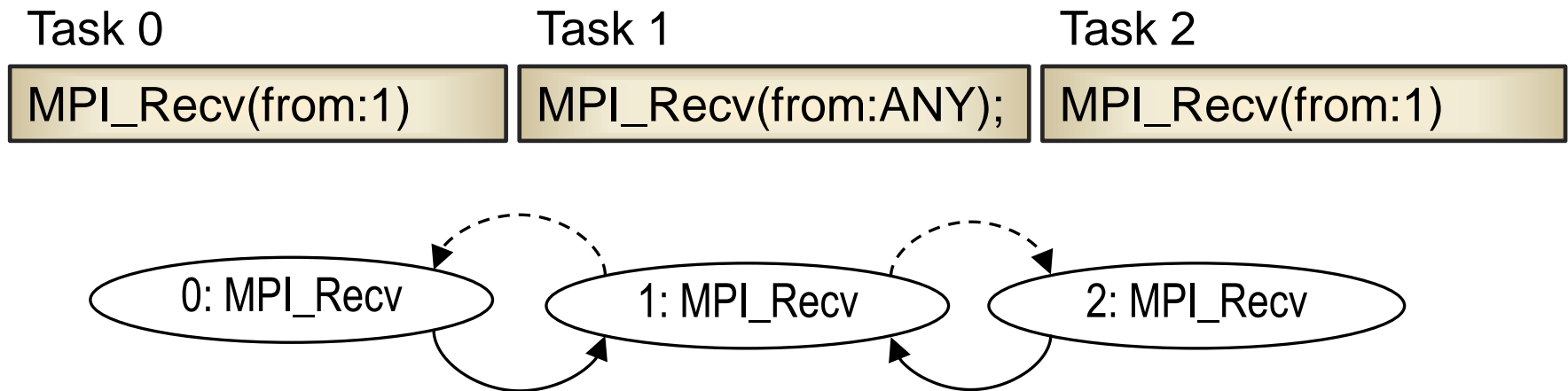
- What about “any” and “some”?
 - MPI_Waitany/Waitsome and wild-card (MPI_ANY_SOURCE) receives have special semantics
 - These wait for at least one out of a set or ranks
 - This is different from the “waits for all” semantic

- Example 4:

Task 0	Task 1	Task 2
MPI_Recv(from:1)	MPI_Recv(from:ANY);	MPI_Recv(from:1)

- What happens:
 - No call can progress, Deadlock
 - 0 waits for 1; 1 waits for either 0 or 1; 2 waits for 1

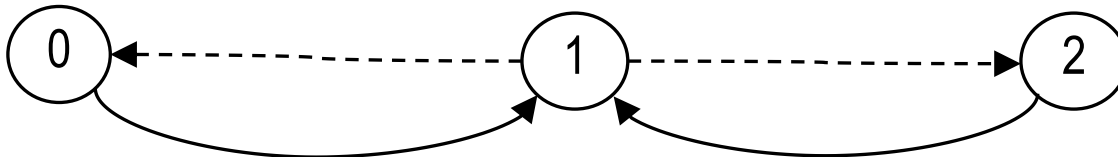
- How to visualize the “any/some” semantic?
 - There is the “Waits for all of” wait type => “AND” semantic
 - There is the “Waits for any of” wait type => “OR” semantic
 - Each type gets one type of arcs
 - AND: solid arcs
 - OR: Dashed arcs
- Visualization for Example 4:



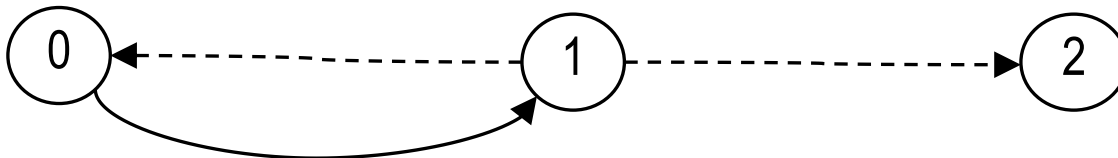
- Deadlock criterion for AND + OR
 - Cycles are necessary but not sufficient
 - A weakened form of a knot (OR-Knot) is the actual criterion
 - Tools can detect it and visualize the core of the deadlock

- Some examples:

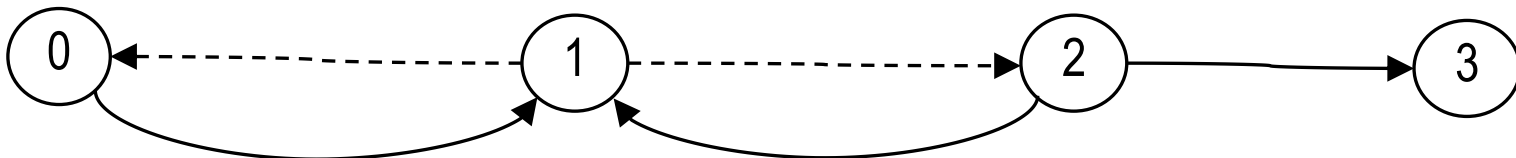
- An OR-Knot (which is also a knot, Deadlock):



- Cycle but no OR-Knot (Not Deadlocked):



- OR-Knot but not a knot (Deadlock):



- MPI Usage Errors
- Error Classes
- **Avoiding Errors**
- Correctness Tools
- Runtime Error Detection
- MUST
- Hands On

- The bugs you don't introduce are the best one:
 - Think, don't hack
 - Comment your code
 - Confirm consistency with asserts
 - Consider a verbose mode of your application
 - Use unit testing, or at least provide test cases
 - Set up nightly builds
 - MPI Testing Tool:
 - <http://www.open-mpi.org/projects/mtt/>
 - Ctest & Dashboards:
 - http://www.vtk.org/Wiki/CMake_Testing_With_CTest

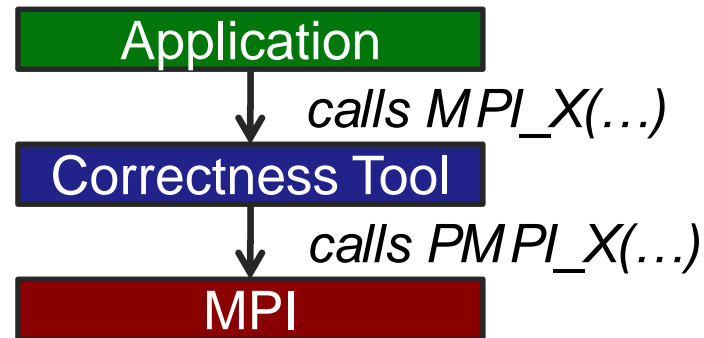
- MPI Usage Errors
- Error Classes
- Avoiding Errors
- **Correctness Tools**
- Runtime Error Detection
- MUST
- Hands On

- Debuggers:
 - Helpful to pinpoint any error
 - Finding the root cause may be very hard
 - Won't detect sleeping errors
 - E.g.: gdb, TotalView, Alinea DDT
- Static Analysis:
 - Compilers and Source analyzers
 - Typically: type and expression errors
 - E.g.: MPI-Check
- Model checking:
 - Requires a model of your applications
 - State explosion possible
 - E.g.: MPI-Spin

- Runtime error detection:
 - Inspect MPI calls at runtime
 - Limited to the timely interleaving that is observed
 - Causes overhead during application run
 - E.g.: Intel Trace Analyzer, Umpire, Marmot, **MUST**
- Formal verification:
 - Extension of runtime error detection
 - Explores ALL possible timely interleavings
 - Can detect potential deadlocks or type mismatches that would otherwise not occur in the presence of a tool
 - For non-deterministic applications exponential exploration space
 - E.g.: ISP

- MPI Usage Errors
- Error Classes
- Avoiding Errors
- Correctness Tools
- **Runtime Error Detection**
- MUST
- Marmot
- Hands On

- A MPI wrapper library intercepts all MPI calls



- Checks analyse the intercepted calls
 - Local checks require data from just one task
 - E.g.: invalid arguments, resource usage errors
 - Non-local checks require data from multiple task
 - E.g.: type matching, collective verification, deadlock detection

- Workflow:
 - Attach tool to target application (Link library to application)
 - Configure tool
 - Enable/disable correctness checks
 - Select output type
 - Enable potential integrations (e.g. with debugger)
 - Run application
 - Usually a regular mpirun
 - Non-local checks may require extra resources, e.g. extra tasks
 - Analyze correctness report
 - May even be available if the application crashes
 - Correct bugs and rerun for verification

- MPI Usage Errors
- Error Classes
- Avoiding Errors
- Correctness Tools
- Runtime Error Detection
- **MUST**
- Hands On

- MPI runtime error detection tool
- Successor of the Marmot and Umpire tools
 - Marmot provides many local checks
 - Umpire provides non-local checks
 - Goal: merging functionality and improving scalability
- OpenSource (BSD licenses)
 - Get MUST at: <http://tu-dresden.de/zih/must>
- Currently Version 1.2, next release Nov. 2013
- Partners:



- Local checks:
 - Integer validation
 - Integrity checks (pointers valid, etc.)
 - Operation, Request, Communicator, Datatype, Group usage
 - Resource leak detection
 - Memory overlap checks
- Non-local checks:
 - Collective verification
 - Lost message detection
 - Type matching (For P2P and collectives)
 - Deadlock detection (with root cause visualization)

- Local checks largely scalable
- Non-local checks:
 - Executed on a central process
 - This process is an MPI task taken from the application
 - Limited scalability ~100 tasks (Depending on application)
 - Can be disabled to provide scalability
- Future versions will provide distributed non-local checks
- Two types of outputs:
 - Logging to `std::cout`
 - Logging to an HTML file
- Uses a scalable tool infrastructure
 - Tool configuration happens at execution time

- 1) Compile and link application as usual
 - Link against the shared version of the MPI lib (Usually default)
 - 2) Replace “mpiexec” with “mustrun”
 - E.g.: *mustrun -np 4 myApp.exe input.txt output.txt*
 - 3) Inspect “MUST_Output.html” in run directory
 - “MUST_Deadlock.dot” exists in case of deadlock
 - Visualize with: *dot -Tps MUST_Deadlock.dot -o deadlock.ps*
- The mustrun script will use an extra process for non-local checks (Invisible to application)
 - I.e.: “mustrun -np 4 ...” will issue a “mpirun -np 5 ...”
 - Make sure to allocate the extra task in batch jobs

- Example “vihps12_2013.c” :

```
(1)  MPI_Init (&argc,&argv);
(2)  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
(3)  MPI_Comm_size (MPI_COMM_WORLD, &size);
(4)
(5)  //1) Create a datatype
(6)  MPI_Type_contiguous (2, MPI_INT, &newType);
(7)  MPI_Type_commit (&newType);
(8)
(9)  //2) Use MPI_Sendrecv to perform a ring communication
(10) MPI_Sendrecv (
(11)     sBuf, 1, newType, (rank+1)%size, 123,
(12)     rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 123,
(13)     MPI_COMM_WORLD, &status);
(14)
(15) //3) Use MPI_Send and MPI_Recv to perform a ring communication
(16) MPI_Send ( sBuf, 1, newType, (rank+1)%size, 456,
              MPI_COMM_WORLD);
(17) MPI_Recv ( rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 456,
              MPI_COMM_WORLD, &status);
(18)
(19) MPI_Finalize ();
```

- Runs without any apparent issue with OpenMPI
- Are there any errors?
- Verify with MUST:
 - `mpicc vihps12_2013.c -o vihps12_2013.exe`
 - `mustrun -np 4 vihps12_2013.exe`
 - `firefox MUST_Output.html`

- First error: Type mismatch

MUST Outputfile

MUST Output, date: Thu Aug 25 09:04:01 2011.

Rank	Thread	Type	Message	From	References	MPI-Standard Reference
0		Error	A send and a receive operation use datatypes that do not match! Mismatch occurs at (CONTIGUOUS)[0](MPI_INT) in the send type and at (MPI_BYTE) in the receive type (consult the MUST manual for a detailed description of datatype positions). The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPI_COMM_WORLD) (Information on send of count 1 with type: Datatype created at reference 3 is for C, committed at reference 4, based on the following type(s): MPI_INT) (Information on receive of count 8 with type:MPI_BYTE)	call MPI_Sendrecv	reference 1: call MPI_Sendrecv@rank 3 reference 2: call MPI_Sendrecv@rank 0 reference 3: call MPI_Type_contiguous@rank 3 reference 4: call MPI_Type_commit@rank 3	

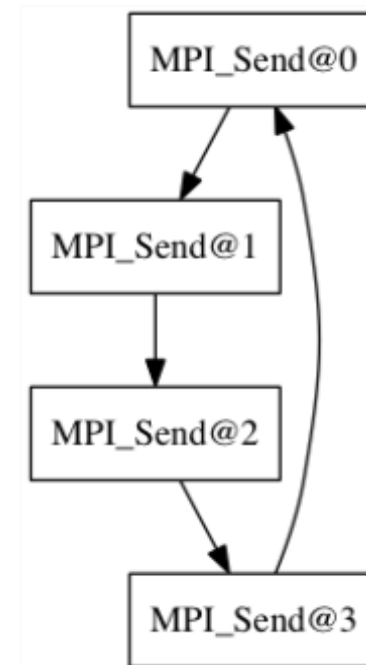
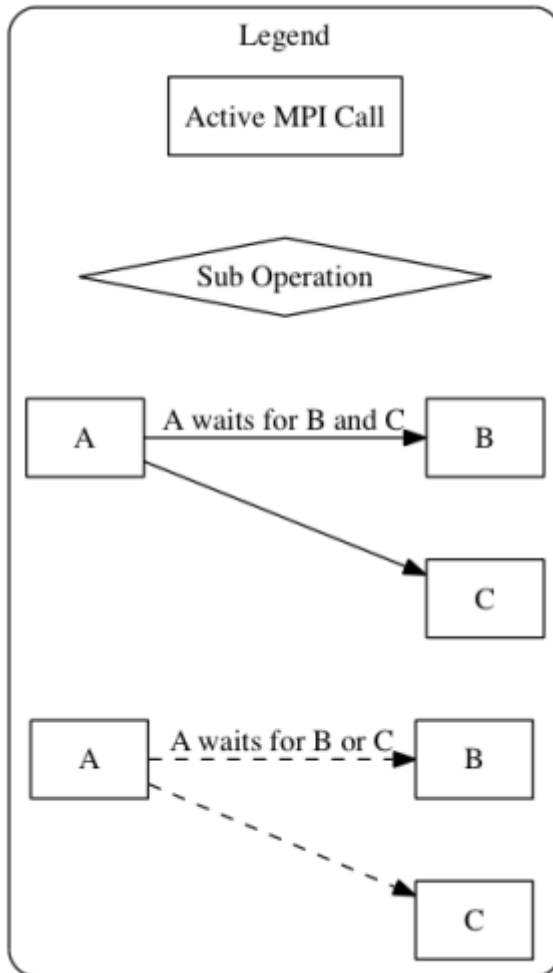
- Second error: Send-send deadlock

MUST Outputfile

MUST Output, date: Thu Aug 25 09:04:01 2011.

Rank	Thread	Type	Message	From	References	MPI-Standard Reference
		Error	<p>The application issued a set of MPI calls that can cause a deadlock! A graphical representation of this situation is available in the file named "MUST_Deadlock.dot". Use the dot tool of the graphviz package to visualize it, e.g. issue "dot -Tps MUST_Deadlock.dot -o deadlock.ps". The graph shows the nodes that form the root cause of the deadlock, any other active MPI calls have been removed. A legend is available in the dot format in the file named "MUST_DeadlockLegend.dot", further information on these graphs is available in the MUST manual. References 1-4 list the involved calls (limited to the first 5 calls, further calls may be involved). The application still runs, if the deadlock manifested (e.g. caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger.</p>		<p>reference 1: call MPI_Send@rank 0 reference 2: call MPI_Send@rank 1 reference 3: call MPI_Send@rank 2 reference 4: call MPI_Send@rank 3</p>	

- Visualization of deadlock (MUST_Deadlock.dot)



- Third error: Leaked datatype

MUST Outputfile

MUST Output, date: Thu Aug 25 09:04:01 2011.

Rank	Thread	Type	Message	From	References	MPI-Standard Reference
		Error	<p>There are 1 datatypes that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these datatypes:</p> <p>-Datatype 1: Datatype created at reference 1 is for C, committed at reference 2, based on the following type(s): MPI_INT</p>		<p>reference 1: call MPI_Type_contiguous@rank 0</p> <p>reference 2: call MPI_Type_commit@rank 0</p>	

- MUST causes overhead at runtime
- Default:
 - MUST expects a crash at any time
 - Blocking communication is used to ensure error detection
 - This can cause high overheads
- If your application doesn't crashes:
 - Add “**--must:nocrash**” to the mustrun command
 - MUST will use aggregated non-blocking communication in that case
 - Provides substantial speed up
- There are more options to mustrun, use “mustrun --help”

- MPI Usage Errors
- Error Classes
- Avoiding Errors
- Correctness Tools
- Runtime Error Detection
- MUST
- Marmot
- **Hands On**

- Set up modules

module load UNITE must

Juqueen

module load UNITE must

Juropa

- Go into the NPB directory
- Edit config/make.def
- Disable any other tool (i.e. use mpif77/mpiifort)
 - For Bluegene use mustf77 linking wrapper
- Build:

```
% make bt-mz NPROCS=4 CLASS=B
```

```
=====
=   NAS PARALLEL BENCHMARKS 3.3   =
=   MPI+OpenMP Multi-Zone Versions =
=   F77                           =
=====
```

```
cd BT-MZ; make CLASS=B NPROCS=4
```

```
make[1]: Entering directory
```

```
...
```

```
mustf77 -O2 -o ../bin/bt-mz.B.4 bt.o initialize.o ...
make[1]: Leaving directory
```

```
mpiifort -O2 -o ../bin/bt-mz.B.4 bt.o initialize.o ...
make[1]: Leaving directory
```

- Go to bin directory

```
% cd bin
```

- Preparation step for Juqueen:

```
mustrun --must:mode prepare --exe ./bt-mz_B.4 --np 4
```

- Create and edit the jobscript

```
cp ../jobscript/juqueen/run.ll ./  
vim run.ll
```

```
cp ../jobscript/run.msub ./  
vim run.msub
```

- Jobscript:

MUST needs one extra process!

```
...
#@bg_size=32
...
CLASS=B
NPROCS=4
EXE=./bt-mz_${CLASS}.${NPROCS}
export OMP_NUM_THREADS=8
export NPB_MZ_BLOAD=0 # disable load balance
```

```
module load UNITE must
module list
```

```
mustrun \
  --must:nocrash \
  --must:mode run \
  --np $NPROCS \
  --ranks-per-node=2 \
  --exe $EXE
```

```
#!/bin/sh
...
#MSUB -l nodes=2:ppn=16
...
cd $PBS_O_WORKDIR

# benchmark configuration
export OMP_NUM_THREADS=4
PROCS=4
CLASS=B
EXE=./bt-mz_${CLASS}.$PROCS
```

```
module load UNITE must
```

```
mustrun --must:nocrash \
-np $PROCS --envall $EXE
```

- Submit the jobscript:

```
lsubmit run.ll
```

```
msub run.msub
```

- Job output should read:

```
% cd bin
% mustrun --must:nocrash -np 4 bt-mz.A.4
Weaver ... success
Code generation ... success
Build file generation ... success
Configuring intermediate build ... success
Building intermediate sources ... success
Installing intermediate modules ... success
Generating P^nMPI configuration ... success
Search for preloaded P^nMPI ... not found ... success
Executing application:
NAS Parallel Benchmarks (NPB3.2-MZ-MPI) - BT-MZ MPI+OpenMP Benchmark
...
Total number of threads: 16 ( 4.0 threads/process)
Calculated speedup = 15.64

Time step 1
...
Verification Successful
```

- Open the MUST output: <Browser> MUST_Output.html

MUST Outputfile

MUST Output, date: Thu Aug 25 12:19:02 2011.

Rank	Thread	Type	Message	From	References	MPI-Standard Reference
		Error	There are 1 communicators that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these communicators: -Communicator 1: Communicator created at reference 1 size=4		reference 1: call MPI_Comm_split@rank 1	
		Error	There are 1 communicators that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these communicators: -Communicator 1: Communicator created at reference 1 size=4		reference 1: call MPI_Comm_split@rank 2	
		Error	There are 1 communicators that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these communicators: -Communicator 1: Communicator created at reference 1 size=4		reference 1: call MPI_Comm_split@rank 3	
		Error	There are 1 communicators that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these communicators: -Communicator 1: Communicator created at reference 1 size=4		reference 1: call MPI_Comm_split@rank 0	

- Many types of MPI usage errors
 - Some errors may only manifest sometimes
 - Consequences of some errors may be “invisible”
 - Some errors can only manifest on some systems/MPIs
- Use MPI correctness tools
- Runtime error detection with MUST
 - Provides various correctness checks
 - Verifies type matching
 - Detects deadlocks
 - Verifies collectives
 - Currently limited scalability

- MUST is a runtime MPI error detection tool
- Usage:
 - Compile & link as always
 - Use “**mustrun**” instead of “mpirun”
 - Keep in mind to allocate 1 extra task in batch jobs
 - Add “**--must:nocrash**” if your application does not crash
 - Open “**MUST_Output.html**” after the run completed/crashed
- For Juqueen:
 - Link with `must$(language)` linker wrapper