

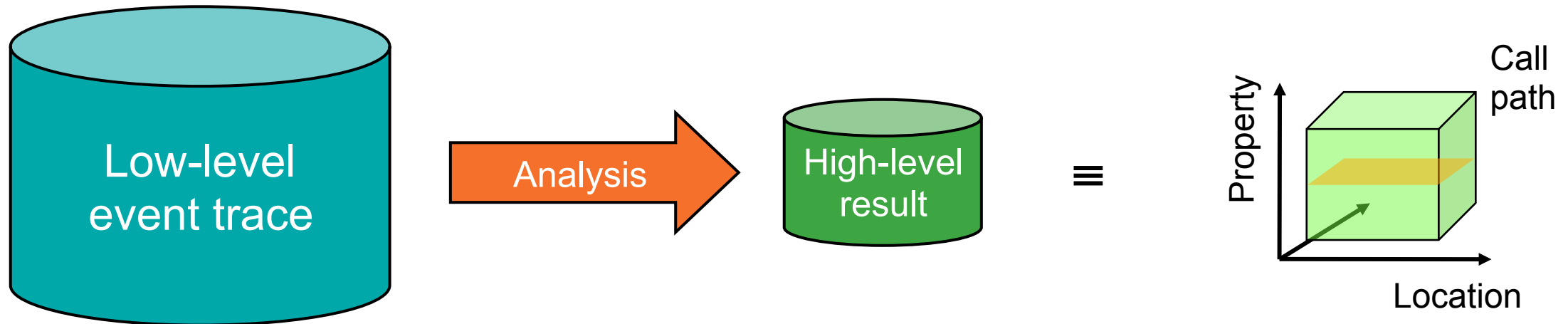
Automatic trace analysis with Scalasca

Brian Wylie
Jülich Supercomputing Centre

scalasca 

Automatic trace analysis

- Idea
 - Automatic search for patterns of inefficient behaviour
 - Classification of behaviour & quantification of significance



- Guaranteed to cover the entire event trace
- Quicker than manual/visual trace analysis
- Parallel replay analysis exploits available memory & processors to deliver scalability

The Scalasca project: Overview

- Project started in 2006
 - Initial funding by Helmholtz Initiative & Networking Fund
 - Many follow-up projects
- Follow-up to pioneering KOJAK project (started 1998)
 - Automatic pattern-based trace analysis
- Now joint development of
 - Jülich Supercomputing Centre
 - German Research School for Simulation Sciences
 - Technische Universität Darmstadt - Laboratory for Parallel Programming



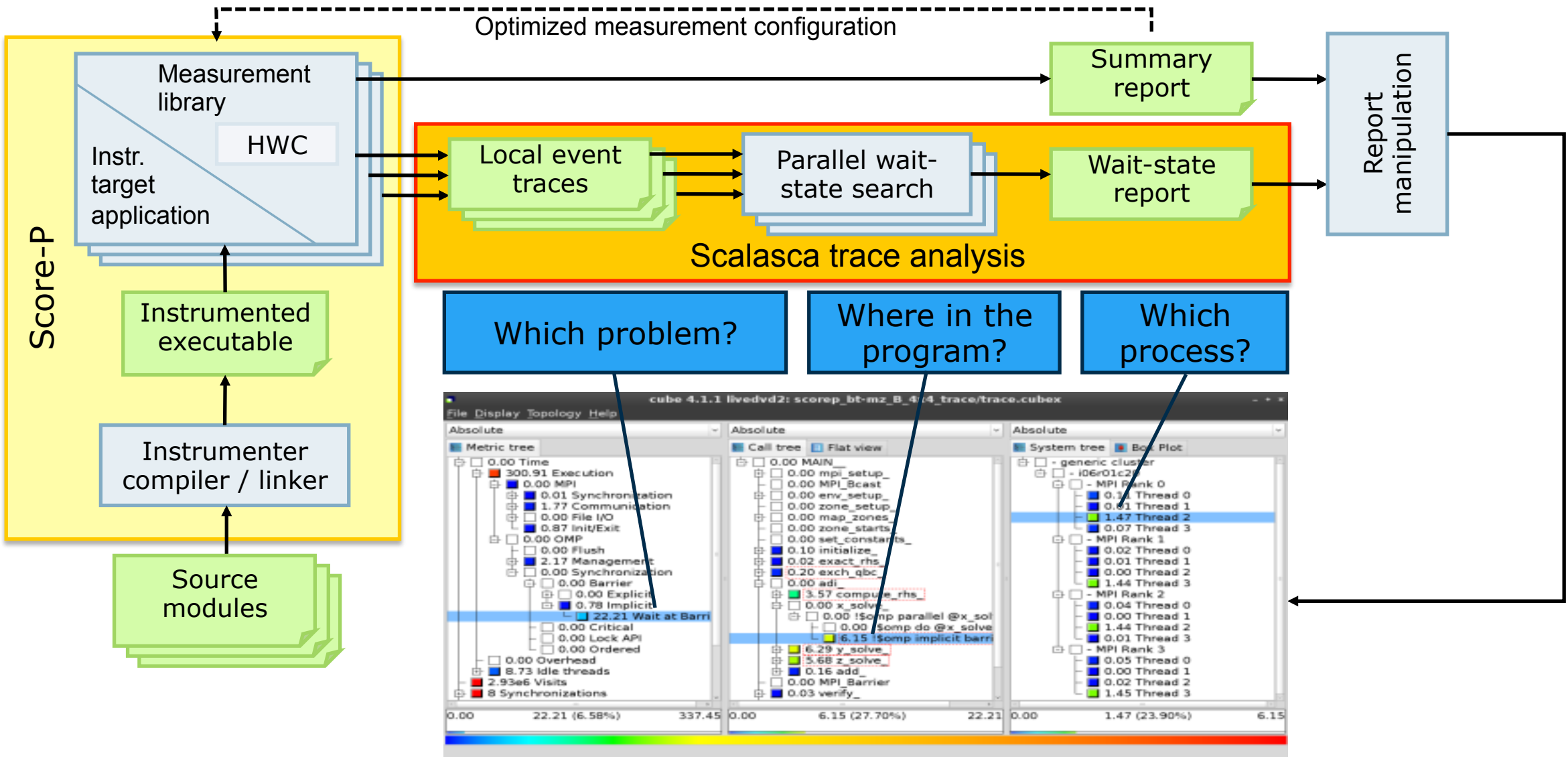
The Scalasca project: Objective

- Development of a **scalable** performance analysis toolset for most popular parallel programming paradigms
- Specifically targeting **large-scale** parallel applications
 - such as those running on IBM Blue Gene or Cray systems with one million or more processes/threads
- Latest release:
 - Scalasca v2.2 coordinated with Score-P v1.4 (January 2015)
 - initial support for Intel Xeon Phi (native mode only)
 - full support for traces in SIONlib format (if configured for OTF2)
 - basic support for POSIX threads and OpenMP tasking
 - added lock contention and root-cause/delay analysis
 - Scalasca v2.2.2 coordinated with Score-P 1.4.2 (June 2015)
 - bug-fixes and optimisations

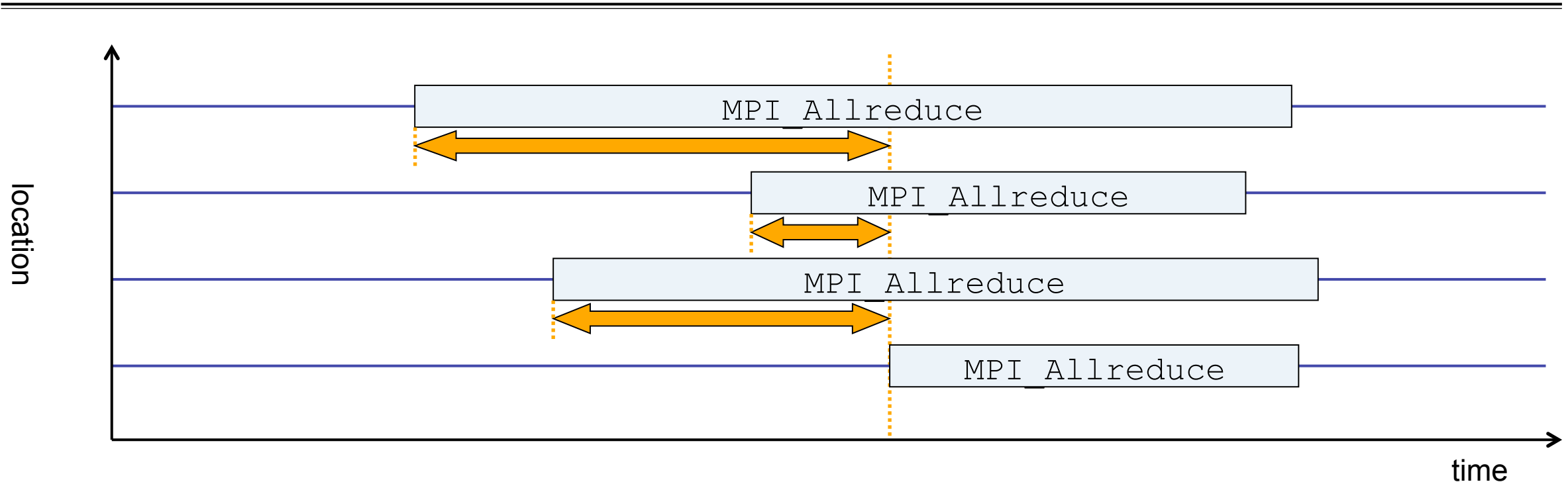
Scalasca 2.2 features

- Open source, New BSD license
- Fairly portable
 - IBM Blue Gene, Cray XT/XE/XK/XC, SGI Altix, Fujitsu FX10/100 & K computer, Linux clusters, Intel Xeon Phi (native MIC) ...
- Uses Score-P instrumenter & measurement libraries
 - Scalasca 2 core package focuses on trace-based analyses
 - Supports common data formats
 - Reads event traces in OTF2 format
 - Writes analysis reports in CUBE4 format
- Current limitations:
 - Unable to handle traces containing CUDA or SHMEM events, or OpenMP nested parallelism
 - PAPI/rusage metrics for trace events are ignored

Scalasca workflow

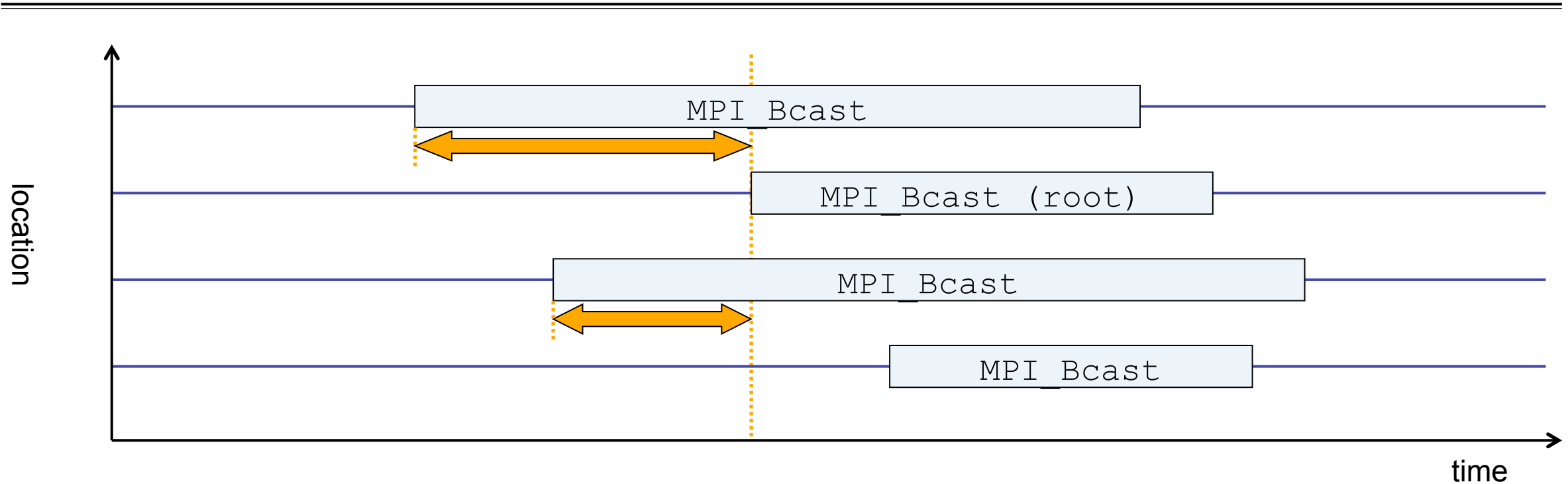


Example: Wait at NxN



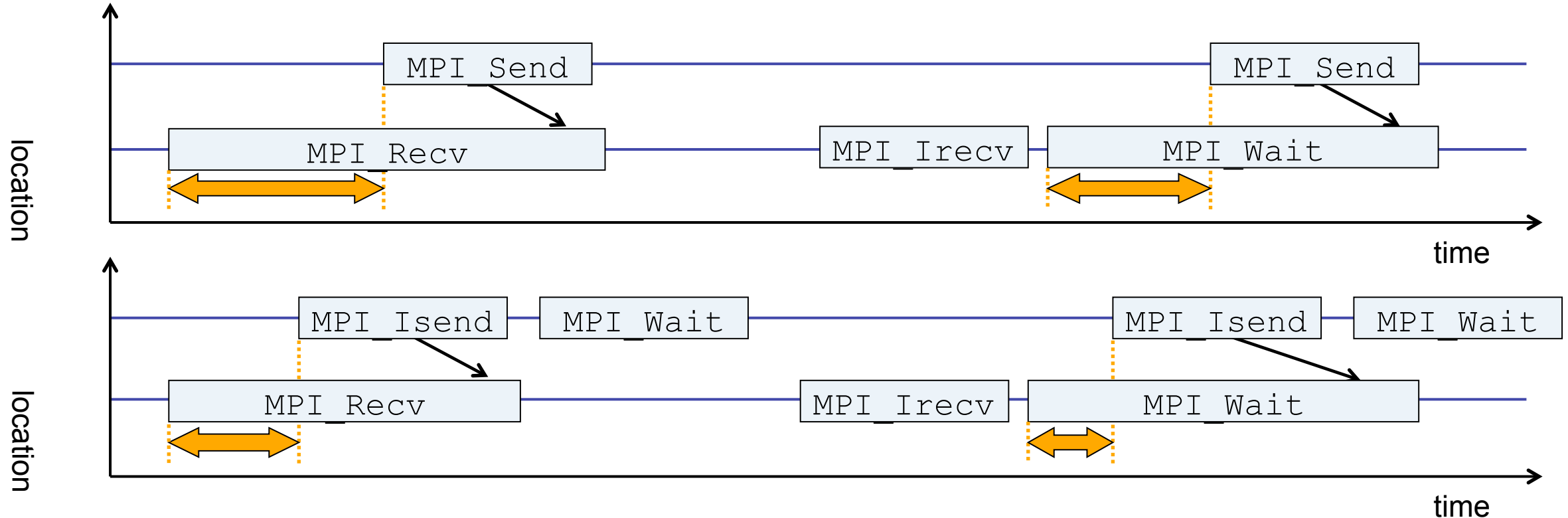
- Time spent waiting in front of synchronizing collective operation until the last process reaches the operation
- Applies to: MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Reduce_scatter, MPI_Reduce_scatter_block, MPI_Allreduce

Example: Late Broadcast



- Waiting times if the destination processes of a collective 1-to-N operation enter the operation earlier than the source process (root)
- Applies to: MPI_Bcast, MPI_Scatter, MPI_Scatterv

Example: Late Sender



- Waiting time caused by a blocking receive operation posted earlier than the corresponding send
- Applies to blocking as well as non-blocking communication

Detecting (Hidden) Correctness Issues in MPI Applications

VI-HPS Team



Content

- Motivation
- Runtime Correctness Workflow
- MUST
- Datatypes and Deadlocks

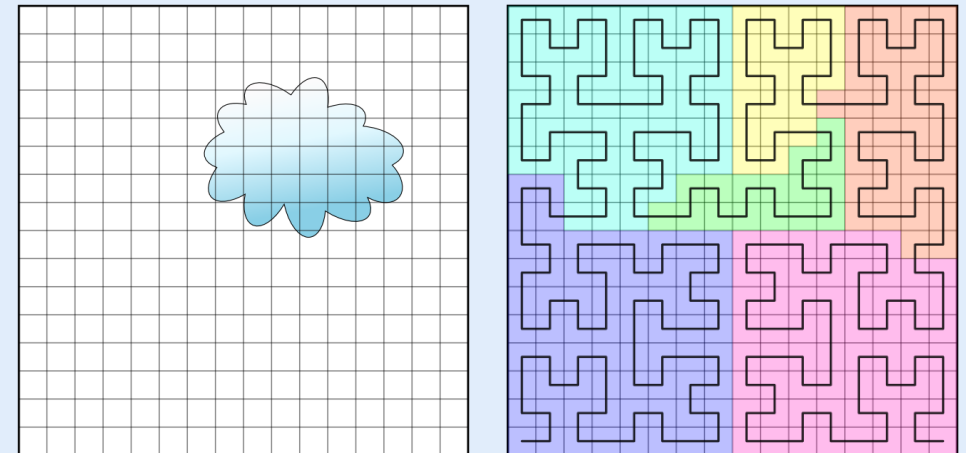
Motivation

- MPI programming is error prone
- Portability errors
(just on some systems, just for some runs)
- Behaviour of an application run:
 - Crash
 - Sys-admin calls you
 - Application hanging
 - Finishes
- Questions:
 - Why crash/hang?
 - Is my result correct?
- Results similar on another system?



Obviousness

Dynamic load balancing
Benchmark
(Development Version):



Starting at 256 processes it
crashes
within the MPI implementation

Motivation (2)

- C code:

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

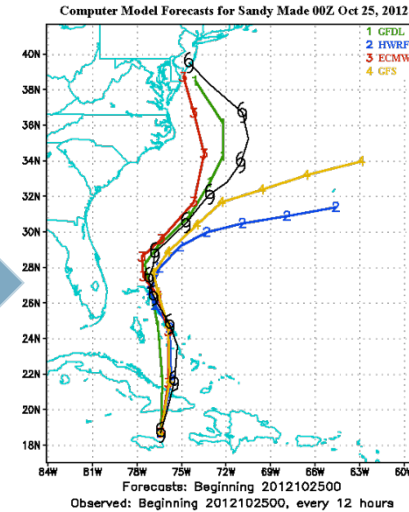
Fortran type in C

Use of uncommitted type

- Tools:

- Runtime correctness tools can detect such errors
- Strength of such tools:
 - Test for conformance to 600+ page MPI standards
 - Understand complex calls, e.g., MPI_Alltoallw with:
 - 9 Arguments, including 5 comm sized arrays

Runtime Correctness Analysis Workflow



Results

Rank	Thread	Type	Message	From	References	MPI-Standard Reference
1		Error	A send and a receive operation use datatypes that do not match! Mismatch occurs at (CONTIGUOUS)[0] (MPL_INT) in the send type and at (MPL_BYTE)[0] 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: MPL_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): { MPL_INT } Typemap = {(MPL_INT, 0), (MPL_INT, 4)}) (Information on receive of count 8 with type:MPL_BYTE)	MPI_Sendrecv from: #0 main@test.c:54 start_main@libc.so	reference 1 rank 0: MPI_Sendrecv from: #0 main@test.c:54 #1 start_main@libc.so reference 2 rank 1: MPI_Sendrecv from: #0 main@test.c:54 #1 start_main@libc.so reference 3 rank 0: MPI_Type_contiguous from: #0 main@test.c:35 #1 start_main@libc.so reference 4 rank 0: MPI_Type_commit from: #0 main@test.c:51 #1 start_main@libc.so	

Correctness report

MUST – Overview

- MPI runtime error detection tool
- Open source (BSD license)
<http://tu-dresden.de/zih/must>
- Wide range of checks, strength areas:
 - Overlaps in communication buffers
 - Errors with derived datatypes
 - Deadlocks
- Largely distributed, can scale with the application
- When to use:
 - After code changes
 - When hunting a manifest defect (hunting the origin of a crash/deadlock/unexpected-result)
 - Unit testing



MUST – Correctness Reports

▪ C code:

```

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

Use of uncommitted type

▪ Tool Output:

MUST Output: 14:11 2014.

Who?
What?
Where?
Details

Rank(s)	Type	Message	From	References
0	Error	Argument 3 (datatype) is not committed for transfer, call MPI_Type_commit before using the type for transfer! (Information on datatypeDatatype created at reference 1 is for Fortran, based on the following type(s): { MPI_INTEGER}Typemap = {(MPI_INTEGER, 0), (MPI_INTEGER, 4)})	Representative location: MPI_Send (1st occurrence) called from: #0 main@test.c:17	References of a representative process: reference 1 rank 0: MPI_Type_contiguous (1st occurrence) called from: #0 main@test.c:14

MUST – Basic Usage

- Apply MUST with an mpiexec wrapper, that's it:

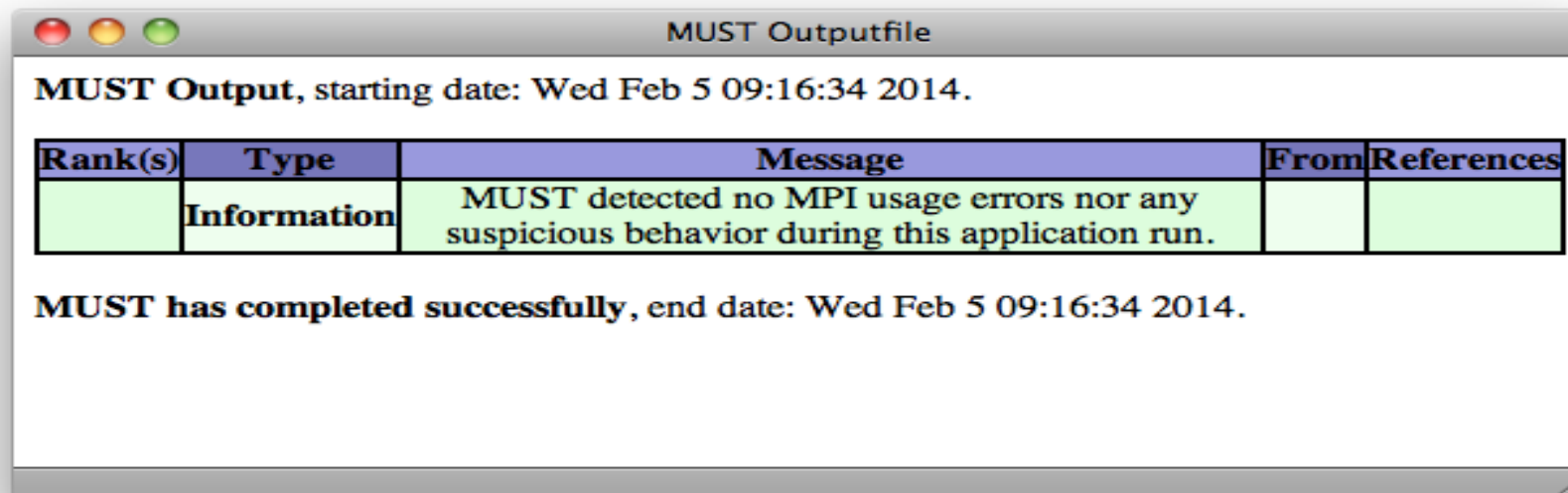
```
% mpicc source.c -o exe  
% mpiexec -np 4 ./exe
```

```
% mpicc -g source.c -o exe  
% mustrun -np 4 ./exe
```

- After run: inspect "MUST_Output.html"
- "mustrun" (default configuration) uses an extra process:
 - I.e.: "mustrun -np 4 ..." will use 5 processes
 - Allocate the extra resource in batch jobs!
 - Default configuration tolerates application crash; BUT is very slow (details later)

MUST – With your Code

- Chances are good that you will get:



MUST Outputfile

MUST Output, starting date: Wed Feb 5 09:16:34 2014.

Rank(s)	Type	Message	From	References
	Information	MUST detected no MPI usage errors nor any suspicious behavior during this application run.		

MUST has completed successfully, end date: Wed Feb 5 09:16:34 2014.

- Congratulations you appear to use MPI correctly!
- Consider:
 - Different process counts or inputs can still yield errors
 - Errors may only be visible on some machines
 - Integrate MUST into your regular testing

Why you want to use MUST

- Examples supported by MUST:

Process 0	Process 1
<code>MPI_Send (to:1, count:1, MPI_INT, ...);</code>	<code>MPI_Recv (from:0, count:1, MPI_DOUBLE, ...);</code>

⇒ MPI Datatype matching error

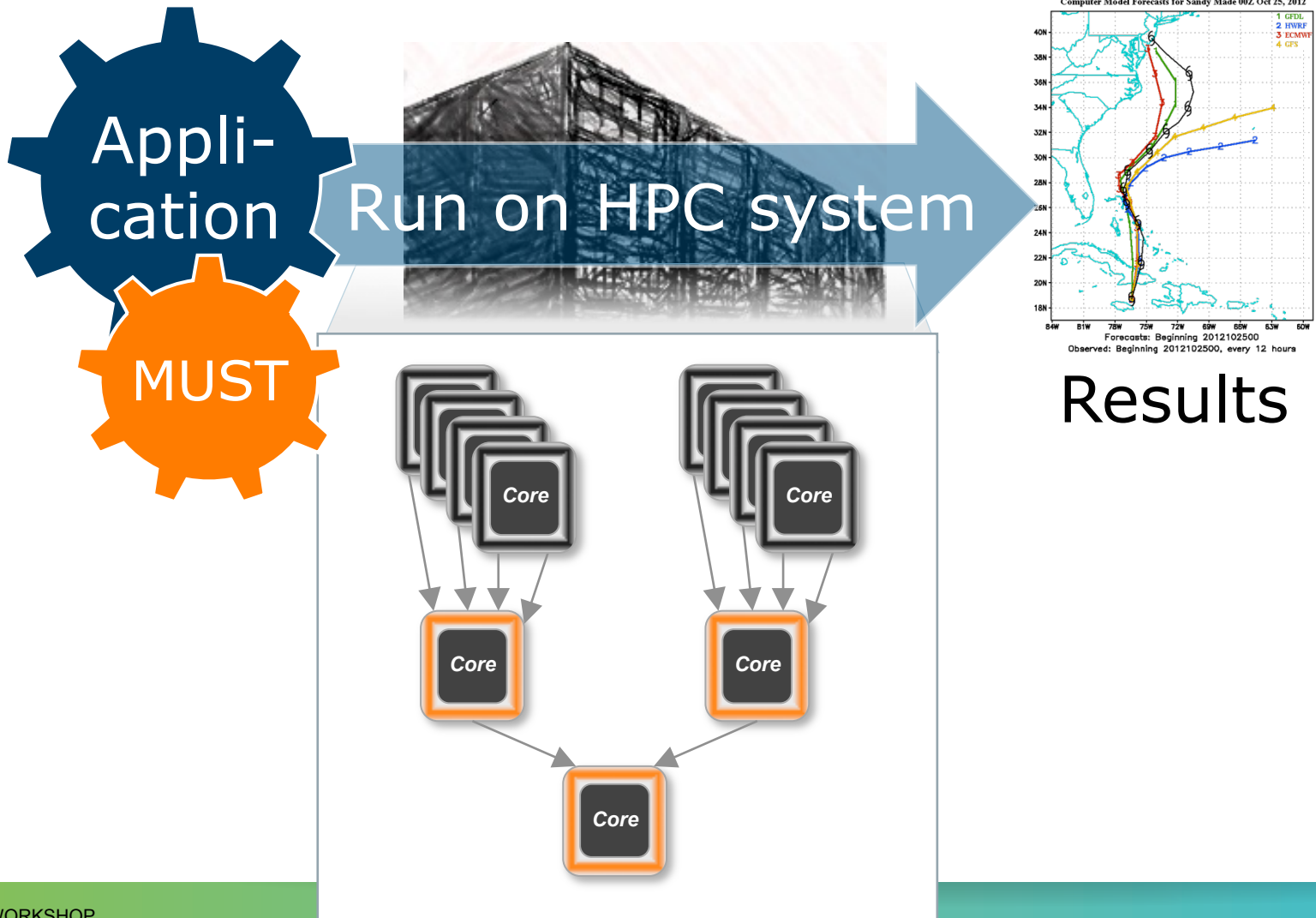
Process 0	Process 1
<code>MPI_Bcast (...);</code>	<code>MPI_Reduce (...);</code>
<code>MPI_Reduce (...);</code>	<code>MPI_Bcast (...);</code>

⇒ Collective matching error, likely a deadlock

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

⇒ Communication buffer overlap at buf[4]

Scalability



Scalability – Operation Modes

- MUST causes overhead at runtime
- MUST expects application crash at any time => MUST has to tolerate that
- Basic operation modes (centralized):

Centralized, application known to crash

mustrun -np X exe

+ All checks enabled

+ Requires only one extra process

- Very slow => use for small test cases at < 32 processes

Centralized, application does not crash

mustrun -np X

--must:nocrash exe

+ All checks enabled

+ Requires only one extra process

- Application must not crash or hang

- Use for < 100 processes

Scalability – Operation Modes (2)

- Advanced operation modes:

Distributed, no crash

```
mustrun -np X  
  --must:fanin Z  
  exe
```

- Uses tree network:
Layer 0: X ranks
Layer 1: $\text{ceil}(X/Z)$ ranks
...
Layer k: 1 rank
- ~ 10.000 process scale

Centralized, crash

```
mustrun -np X  
  --must:nodesize Y  
  exe
```

- Three layer network:
Layer 0: X
Layer 1: $\text{ceil}(X/(Y-1))$
Layer 2: 1
- + < 100 processes
+ All checks
- Currently not on all systems

Distributed, crash

```
mustrun -np X  
  --must:nodesize Y  
  --must:fanin Z  
  exe
```

- Uses tree network:
Layer 0: X
Layer 1: $A = \text{ceil}(X/(Y-1))$
Layer 2: $B = \text{ceil}(A/Z)$
...
Layer k: 1
- + ~ 10.000 process scale

Scalability – “--must:info”

- Use “--must:info” to learn about a configuration:

```
% mustrun --must:info \  
--must:fanin 16 \  
--must:nodesize 12 \  
-np 1024  
[MUST] MUST configuration ... distributed checks  
with application crash handling  
[MUST] Required total number of processes ... 1125  
[MUST] Number of application processes ... 1024  
[MUST] Number of tool processes ... 101  
[MUST] Total number of required nodes ... 94  
[MUST] Tool layers sizes ... 1024:94:6:1
```

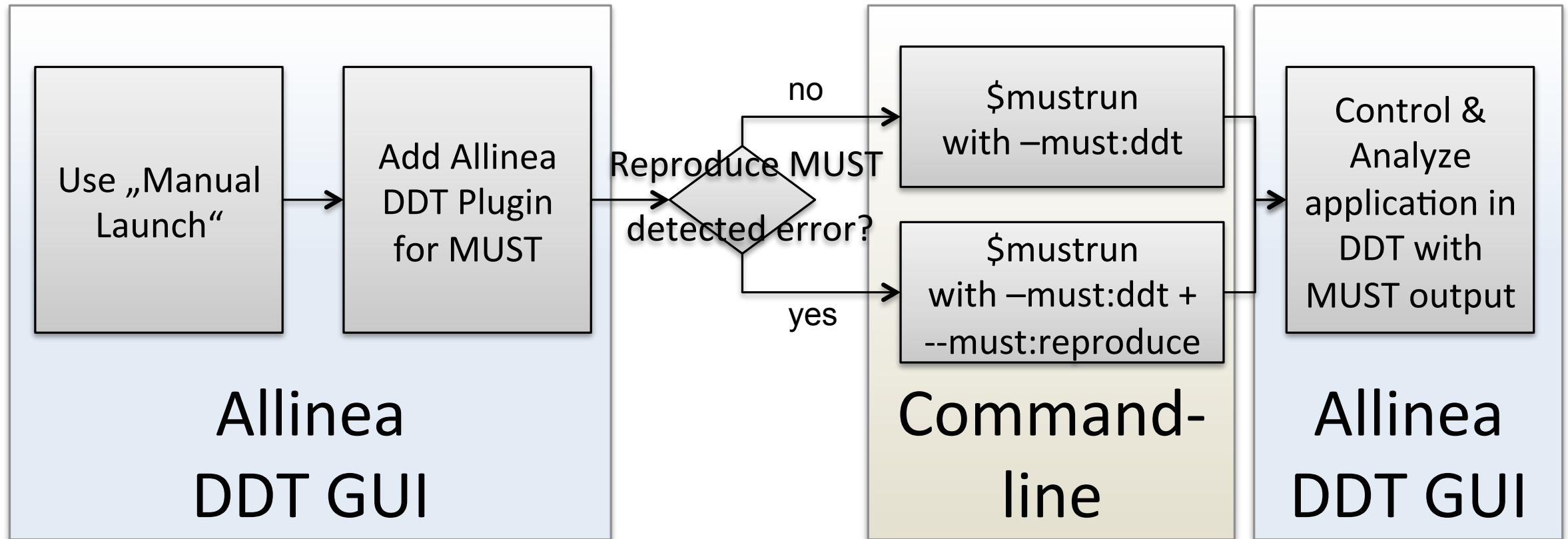
Configuration type

Total number processes
used

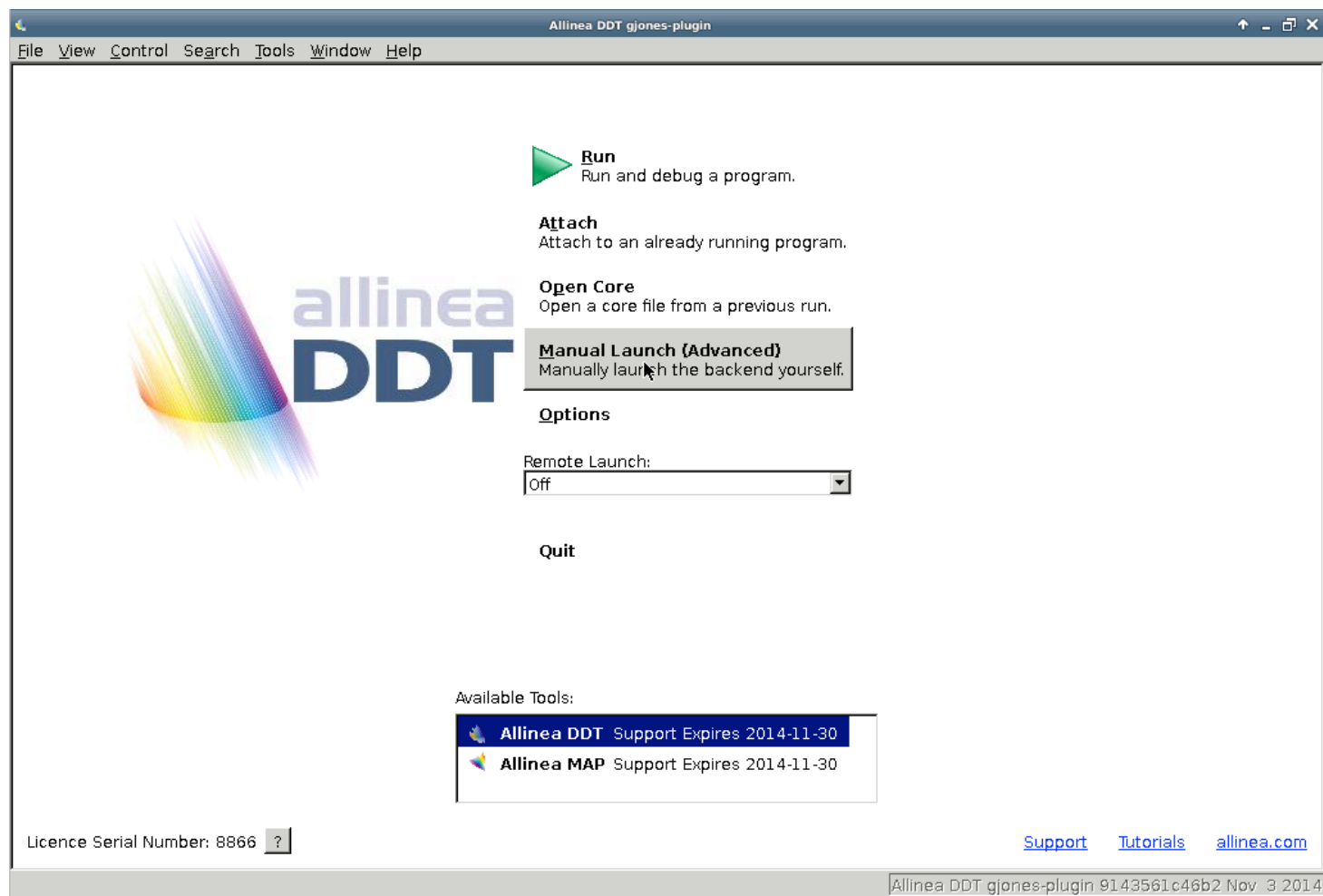
Number of compute nodes

Tree layout

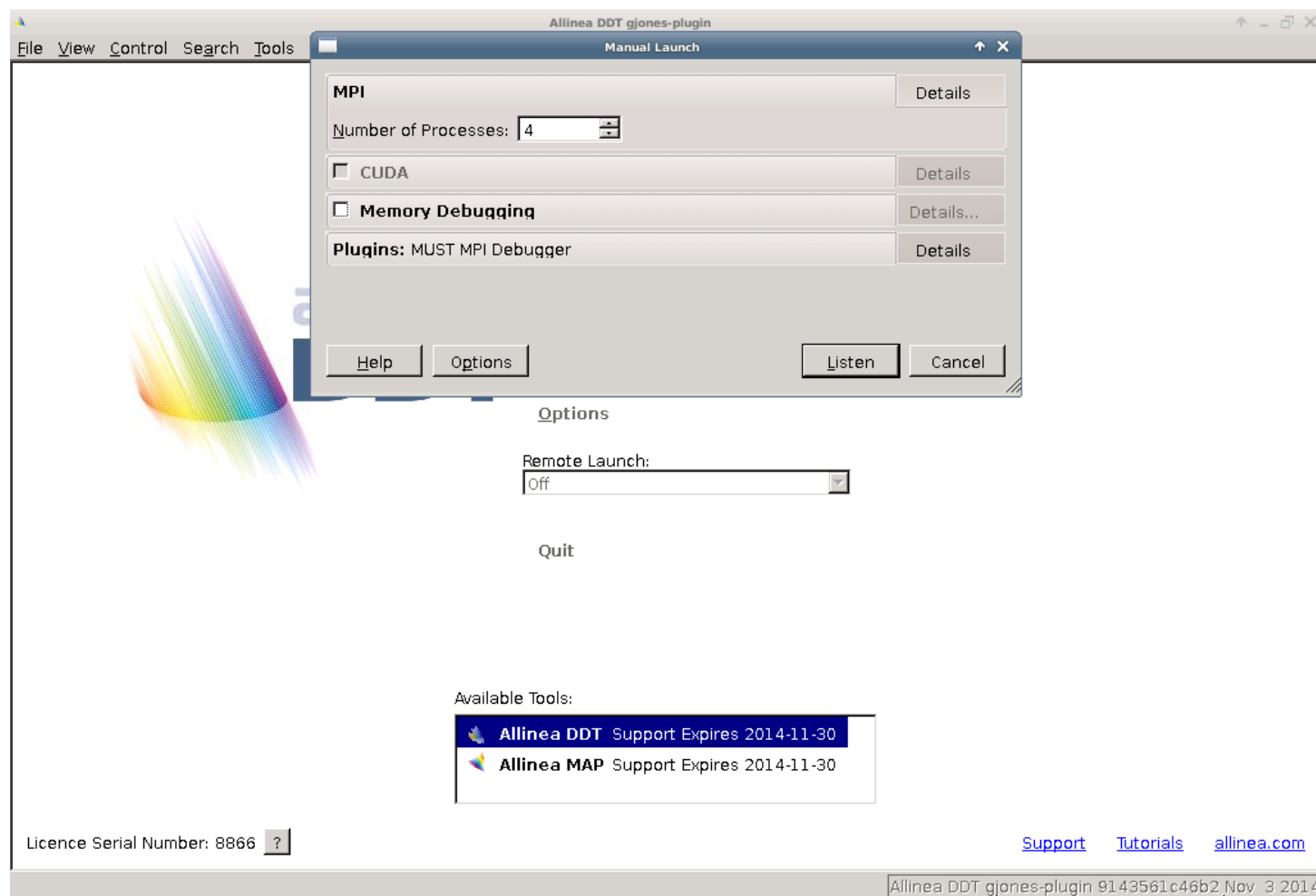
MUST with Alinea DDT Debugger



MUST with Allinea DDT Debugger – Example



MUST with Allinea DDT Debugger – Example (2)



MUST with Allinea DDT Debugger – Example (3)

- Once DDT is listening within the manual launch mode:

```
% mustrun --must:ddt -np 4 ./exe
```

- After run: inspect “MUST_Output.html”
- “mustrun” (default configuration) uses an extra process:
 - I.e.: “mustrun -np 4 ...” will use 5 processes
 - Allocate the extra resource in batch jobs!
 - Default configuration tolerates application crash; BUT is very slow (details later)

MUST with Alinea DDT Debugger – Example (4)

The screenshot displays the Alinea DDT debugger interface for the 'gJones-plugin'. The main window shows the source code of 'heatC-MPI-MUST-1.c' with the following content:

```
407 }
408 }
409
410 /* Main program and time stepping loop.
411 * Main program and time stepping loop.
412 * Main program and time stepping loop.
413 */
414 int main (int argc, char** argv)
415 {
416     heatGrid mygrid;
417     double dt, dthetamax, energyInitial, energyFinal;
418     int step, nsteps=20;
419     dataMPI mympi;
420
421     /* initialize MPI */
422     MPI_Init(&argc, &argv);
```

The 'Stacks' window shows the current stack frame:

Processes	Function
5	main (heatC-MPI-MUST-1.c:414)

The 'Evaluate' window shows the current expression and value:

Expression	Value
i	<No symbol "i" in current
j	<No symbol "j" in current

The 'Current Line(s)' window is empty. The 'Locals' window shows 'Type: none selected'. The 'Input/Output' window is empty. The 'Breakpoints' window is empty. The 'Watchpoints' window is empty. The 'Tracepoints' window is empty. The 'Tracepoint Output' window is empty. The 'Logbook' window is empty. The 'Ready' button is visible at the bottom right.

MUST with Allinea DDT Debugger – Example (5)

The screenshot displays the Allinea DDT debugger interface. A central dialog box titled "MUST Error" contains the following text:

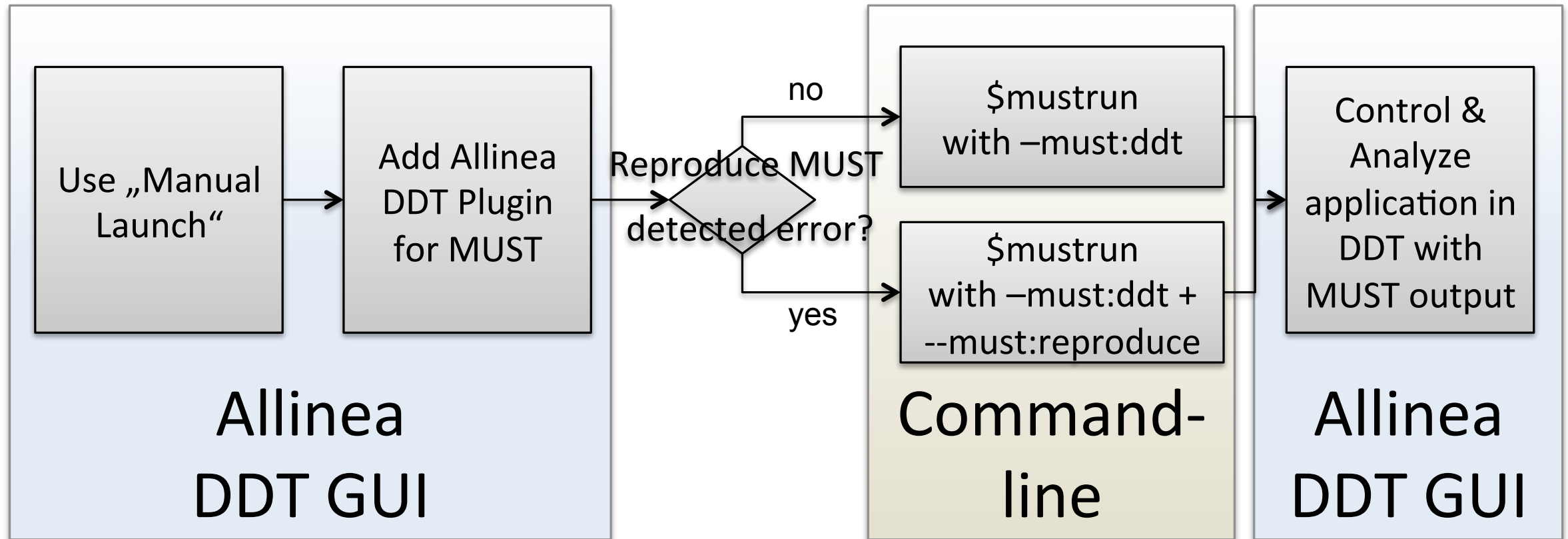
Processes 0-1,3-4:
There are 20 requests that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize...

Buttons: Continue, Pause, Pause All, Show Details...

The debugger window shows the following components:

- Project Files:** A tree view showing files like LocationInfo.h, lru_cache.h, malloc_check.c, mca.h, mca_base_var.h, mpi.h, **MsgLoggerDdt.cpp**, MsgLoggerDdt.h, mutex.h, mutex_unix.h, and Node.h.
- Stacks:** A table showing the call stack for process 4. The stack includes:
 - XMPI_Finalize NewStack
 - NQ_Finalize
 - MPI_Finalize
 - must::FinalizeNotify::notify
 - finalizeNotify
 - must::LeakChecks::finalizeNotify
 - must::LeakChecks::reportRequests
 - must::CreateMessage::createMessage
 - must::CreateMessage::createMessage
 - handleNewMessage
 - must::MsgLoggerDdt::log (MsgLoggerDdt.cpp:79)
 - must::MsgLoggerDdt::logStrided (MsgLoggerDdt.cpp:110)
 - myDdtBreakpointFunctionError (MsgLoggerDdt.cpp:166)
- Current Line(s):** A table with columns for Variable Name and Value.
- Evaluate:** A table with columns for Expression and Value, showing expressions like "<No symbol 'i' in current" and "<No symbol 'j' in current".

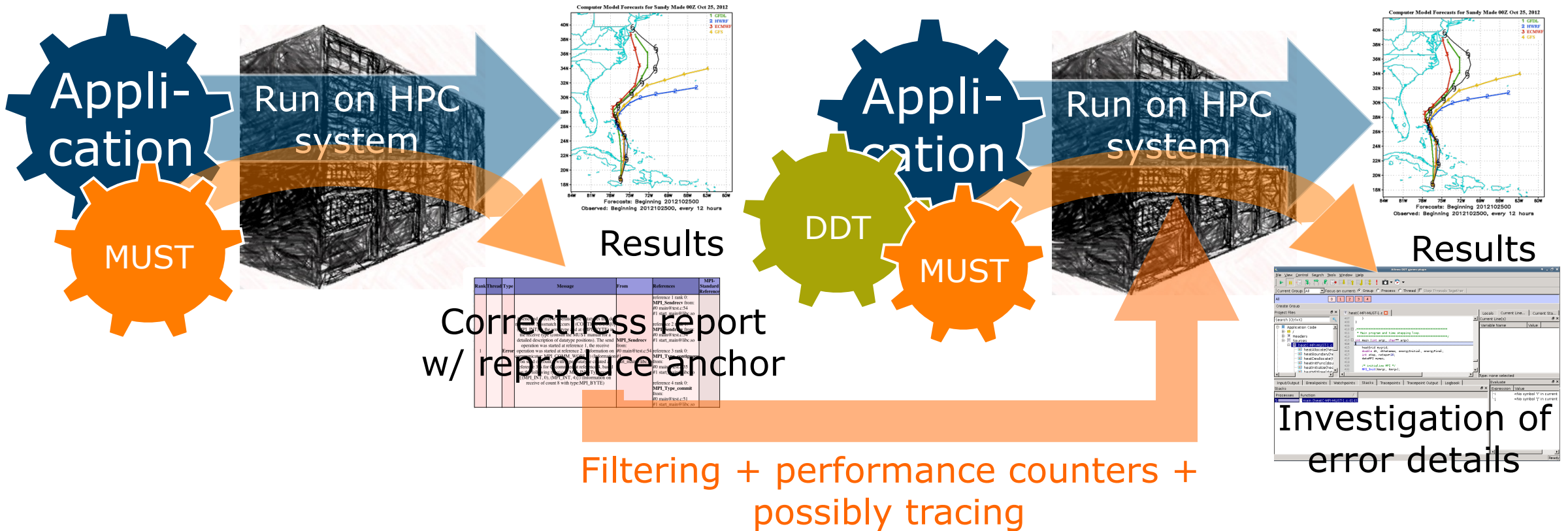
MUST with Alinea DDT Debugger – Reproduce Mode



MUST with Alinea DDT Debugger – Reproduce Mode Workflow

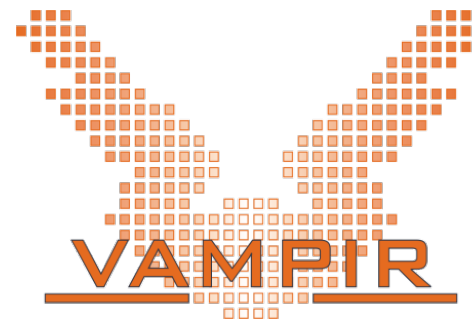
▪ First run with MUST “--reproduce”

▪ Run w/ “--must:reproduce --must:ddt”



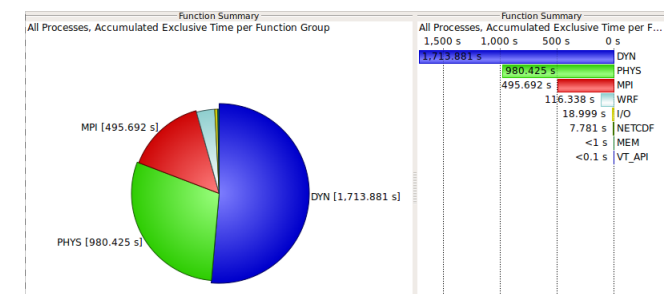
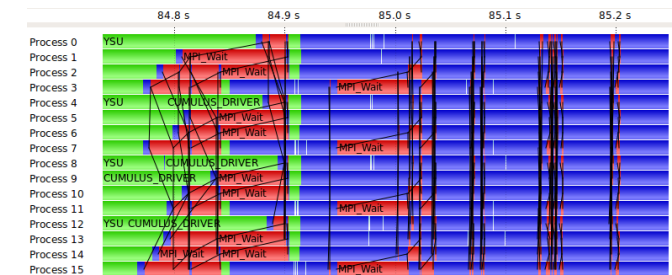
A Quick(!) Look at Vampir: Visualization of Parallel Application Behaviour

VI-HPS Team



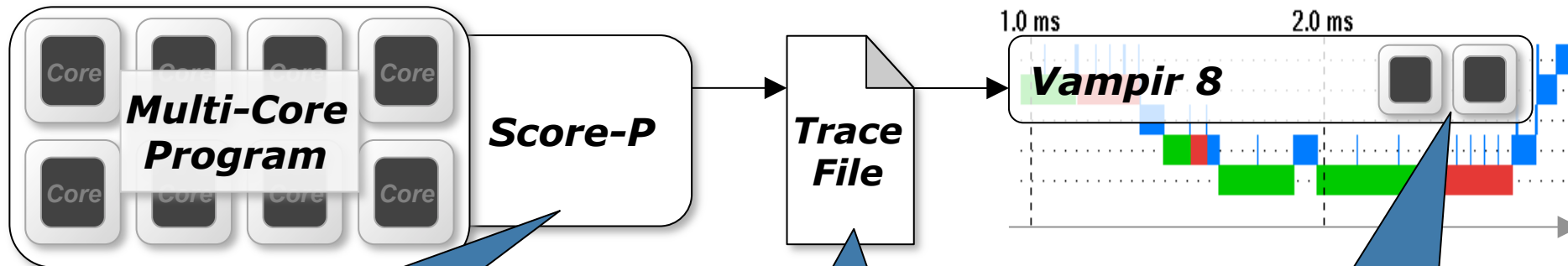
Event Trace Visualization with Vampir

- Alternative and supplement to automatic analysis
- Show dynamic run-time behavior graphically at any level of detail
- Provide statistics and performance metrics
- **Timeline charts**
 - Show application activities and communication along a time axis
- **Summary charts**
 - Provide quantitative results for the currently selected time interval



Vampir Workflow

```
% module load vampir
% vampir
```



With:
SCOREP_ENABLE_TRACING=true

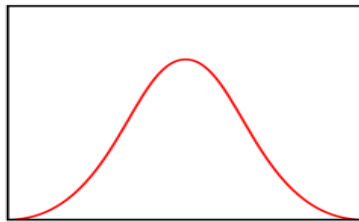
Small/Medium
sized trace

Thread parallel

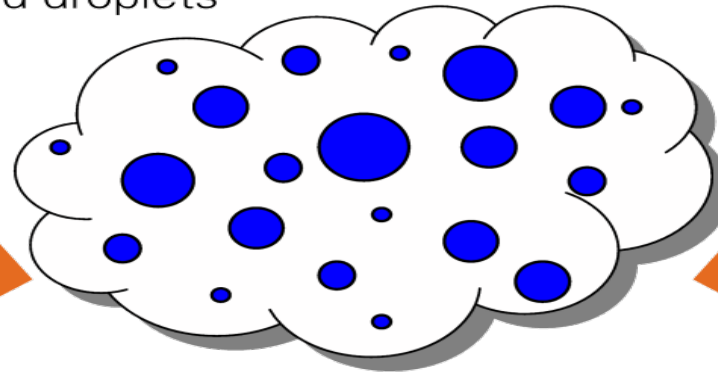
Story 1 from Motivation

- COSMO-SPECS a coupling of:
 - Weather forecast model
 - Detailed cloud microphysics scheme

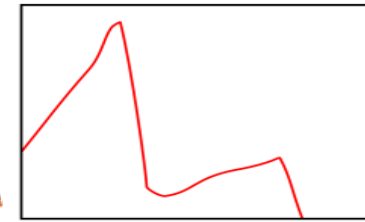
COSMO: Approximation
of cloud droplet size



Cloud droplets



COSMO-SPECS:
Bin-wise discretization of
cloud droplet size



Developer observation:

Runtime per iteration increases over time, why?

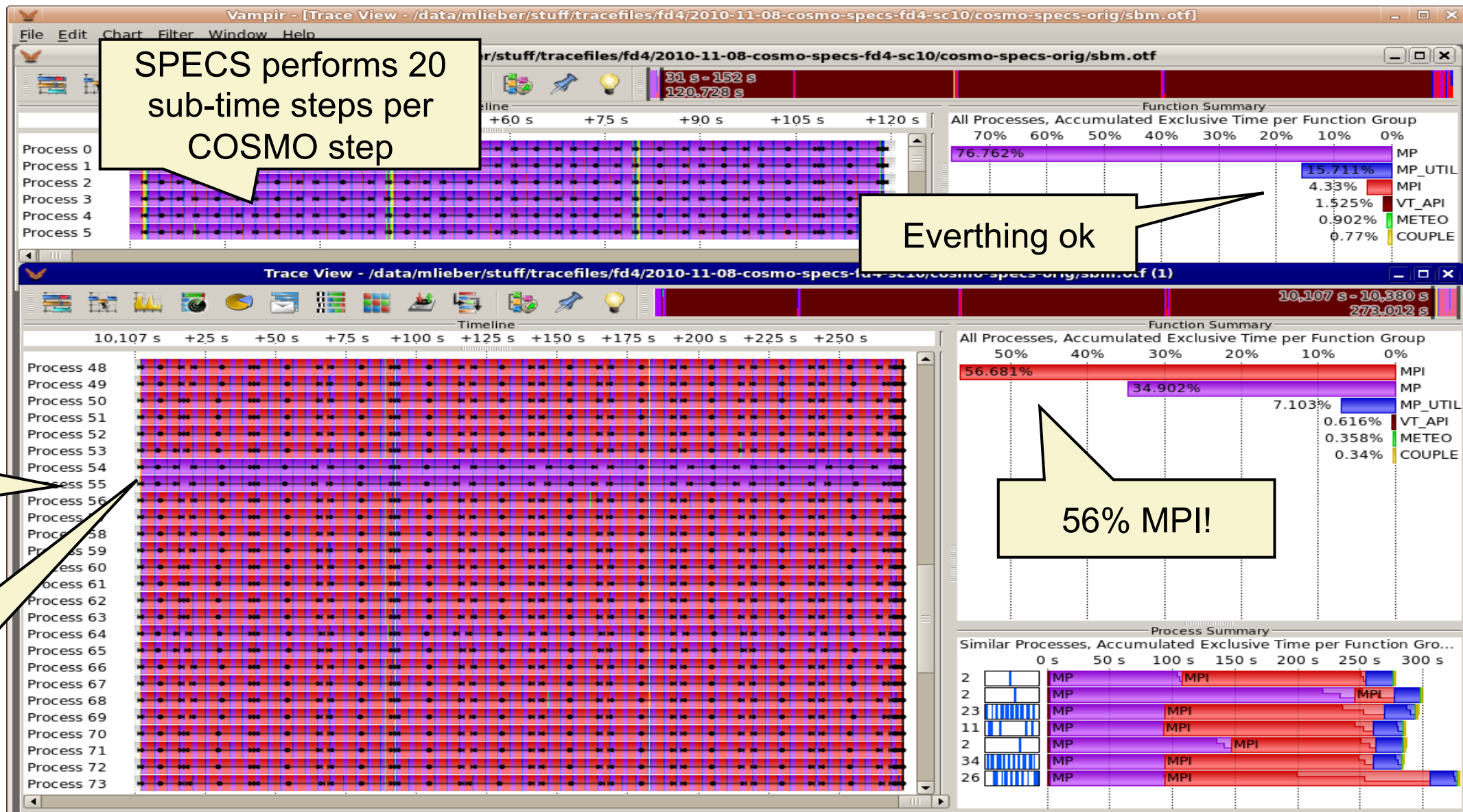
First 3 time steps of COSMO-SPECS run

SPECS performs 20 sub-time steps per COSMO step

Last 3 time steps of COSMO-SPECS run

Heavy load imbalance

Cloud grows in grid cells of these MPI ranks



Everything ok

56% MPI!