MPI Performance

Intel® MPI Library

Intel® Trace Analyzer and Collector

Notices & Disclaimers

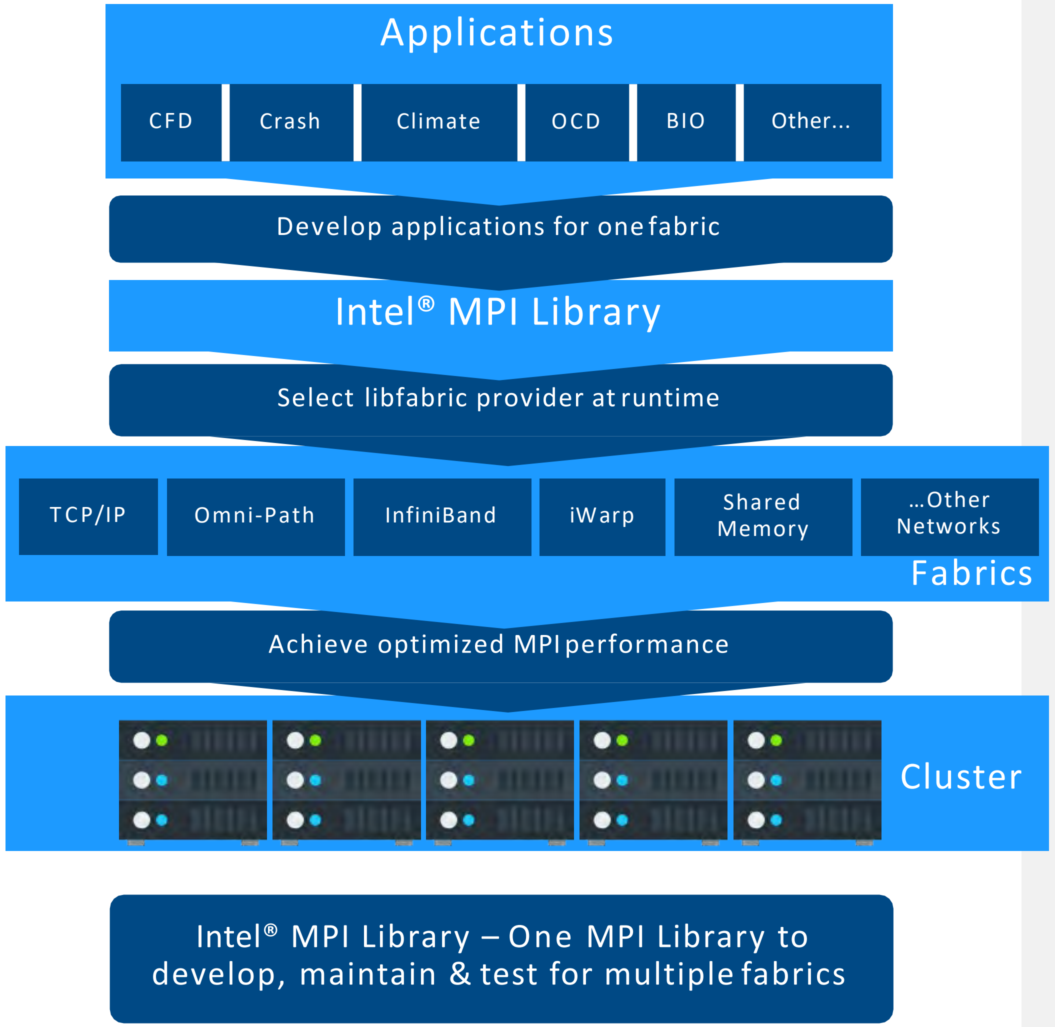
Agenda

- Distributed Performance with Intel® MPI Library
- Tuning MPI Application Performance with Intel® Trace Analyzer and Collector
- Related Tools – Intel® MPI Benchmarks
- Summary and Resources

Intel® MPI Library

Intel® MPI Library Overview

- Optimized MPI application performance
  - Support for all Intel® Xeon® and Intel® Xeon Phi™ processors
  - Optimized collectives with topology and architecture awareness
- Lower-latency and multi-vendor interoperability
  - Industry leading latency
  - Performance optimized support for the fabric capabilities through OpenFabrics* (OFI) / libfabric
- Sustainable scalability up to 340K cores
  - Efficient path by relying on libfabric
  - New: Faster startup and finalization
- More robust MPI applications
  - Seamless interoperability with Intel® Trace Analyzer and Collector
- Conditional Numerical Reproducibility
  - I_MPI_CBWR to control reproducible results across topologies and hardware



Applications
CFD
Crash
Climate
OCD
BIO
Other...
Develop applications for one fabric
Intel® MPI Library
Select libfabric provider at runtime
TCP/IP
Omni-Path
InfiniBand
iWarp
Shared Memory
...Other Networks
Fabrics
Achieve optimized MPI performance
Cluster
Intel® MPI Library – One MPI Library to develop, maintain & test for multiple fabrics

Intel® MPI Library Overview

- Streamlined product setup
  - Install as root, or as standard user
  - Environment variable script mpivars.(c)sh sets paths
- Compilation scripts to handle details
  - One set to use Intel compilers, one set for user-specified compilers
- Environment variables for runtime control
  - I_MPI_* variables control many factors at runtime
    - Process pinning, collective algorithms, device protocols, and more

Compiling MPI Programs

- Compilation scripts automatically passes necessary libraries and options to underlying compiler
  - *mpiifort*, *mpiicpc*, and *mpiicc* use the Intel compiler by default
  - *mpif77*, *mpicxx*, *mpicc*, and others use GNU compiler by default
- Multiple ways to specify underlying compiler
  - I_MPI_F77, I_MPI_CXX, etc. environment variables
  - -f77, -cc, etc. command line options
  - Useful for makefiles portable between MPI implementations
- All compilers are found via PATH

MPI Launcher

- Robust launch command

```
mpirun <mpi args> executable <program args>
```

- Options available for:
  - Rank distribution and pinning
  - Fabric selection and control
  - Environment propagation
  - And more

Process Placement

- Layout Across Nodes
  - Default placement puts one rank per core on each node
  - Use –ppn to control processes per node
  - Use a machinefile to define ranks on each node individually
  - Use arguments sets or configuration files for precise control for complex jobs
- Pinning on Node
  - Can pin to single or multiple cores
  - Multiple options for automatic distribution based on resources such as socket, shared cache level, NUMA arrangement
  - See documentation for details:
    - https://software.intel.com/en-us/mpi-developer-reference-linux-process-pinning
    - https://software.intel.com/en-us/mpi-developer-reference-linux-environment-variables-for-process-pinning
    - https://software.intel.com/en-us/mpi-developer-reference-linux-interoperability-with-openmp

Fabric Control via libfabric

- I_MPI_OFI_PROVIDER chooses provider (select based on interconnect hardware):
  - Default is normally fine
  - tcp – Ethernet
  - psm2 – Intel® Omni-Path Architecture
  - mlx – InfiniBand* (requires at least Intel® MPI Library 2019 Update 5 and UCX 1.4)
  - efa – AWS* EFA (Elastic Fabric Adapter), see https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/efa-start.html for setup process

Conditional Numerical Reproducibility

- I_MPI_CBWR
  - 0 (default) – no reproducibility controls, utilize all optimizations
  - 1 (weak) – disable topology aware optimizations, reproducible across different rank placements/topologies
  - 2 (strict) – disables topology aware optimizations and hardware optimizations, reproducible across hardware and topology
- MPI_Comm_dup_with_info
  - “I_MPI_CBWR”=“yes”, sets strict mode for communicator

Automatic Tuning via Autotuner

- Tuning happens behind the scenes during application run
- Tuning is per communicator
- To tune:
  - I_MPI_TUNING_MODE=auto
  - I_MPI_TUNING_BIN_DUMP=<tuning file> (optional)
- To use tuning results:
  - I_MPI_TUNING_BIN=<tuning file>
- Additional options for more control, see https://software.intel.com/en-us/mpi-developer-reference-linux-autotuning

Debugging MPI Applications

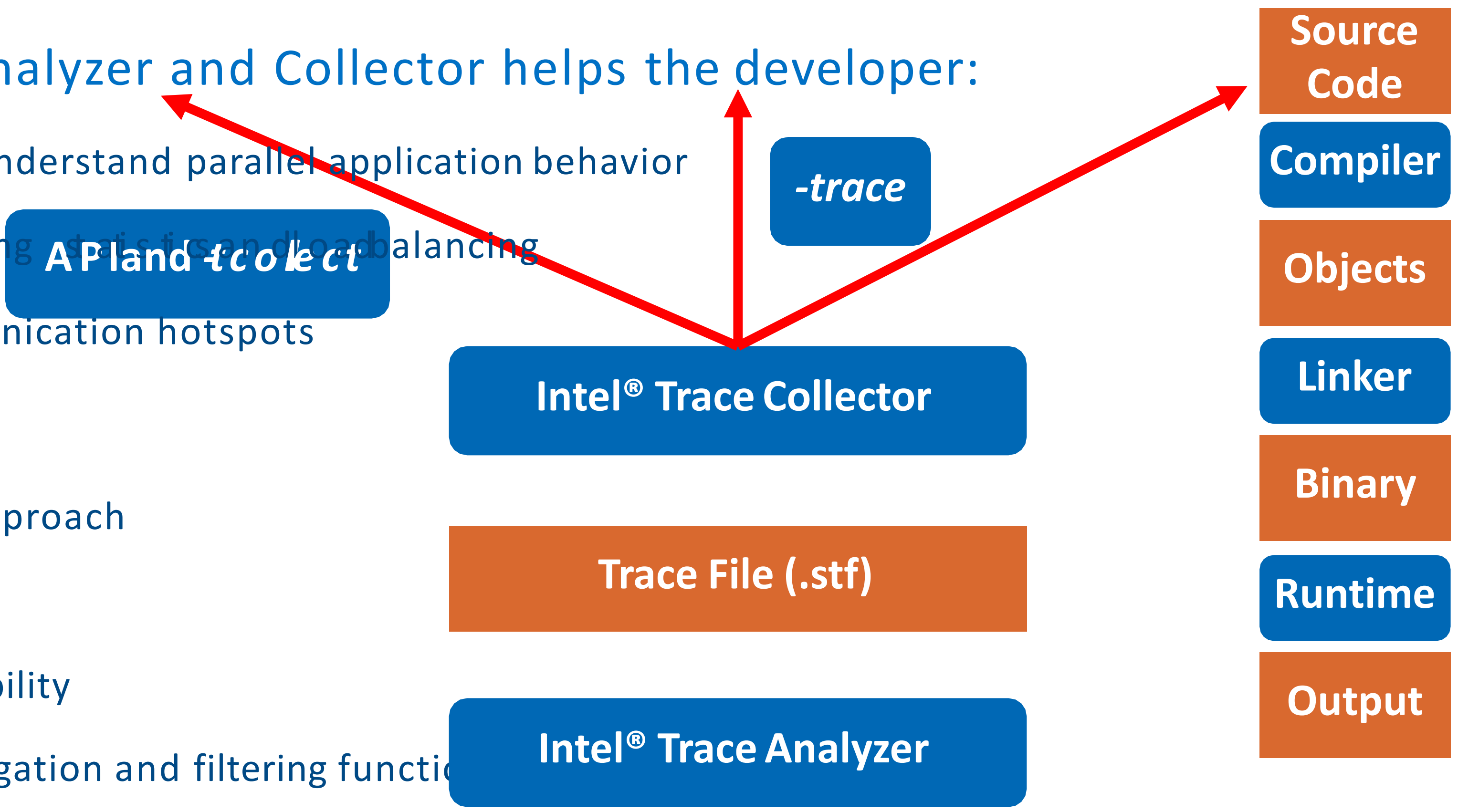
- GDB*
  - mpirun <mpi options> -gdb <application and options>
  - mpirun –n <nranks> -gdba <mpirun pid>
- Allinea* DDT*
  - ddt mpirun …
- gtool (https://software.intel.com/en-us/mpi-developer-reference-linux-gtool-options)
  - Set via –gtool option, -gtoolfile option, or I_MPI_GTOOL
  - "<prepend>:<rank set>[=launch mode][@arch]

Intel® Trace Analyzer and Collector

Event-based Tracing for Distributed Applications

Intel® Trace Analyzer and Collector Overview

- Intel® Trace Analyzer and Collector helps the developer:
  - Visualize and understand parallel application behavior
  - Evaluate profiling statistics and load balancing
  - Identify communication hotspots
- Features
  - Event-based approach
  - Low overhead
  - Excellent scalability
  - Powerful aggregation and filtering functions
  - Performance Assistance and Imbalance Tuning



Strengths of Event-based Tracing

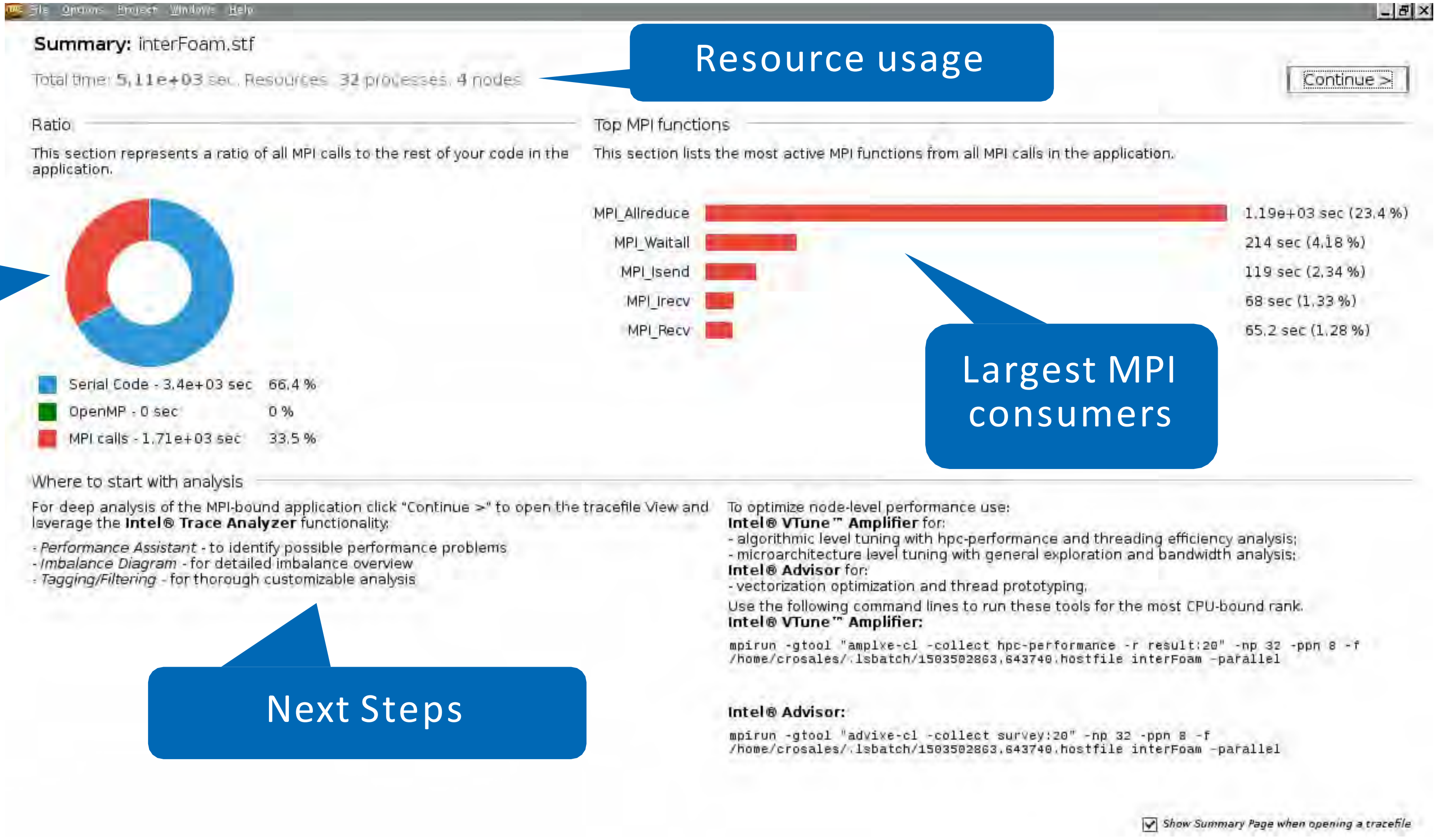
Predict Detailed MPI program behavior

Record Exact sequence of program states – keep timing consistent

Collect Collect information about exchange of messages: at what times and in which order

An event-based approach is able to detect temporal dependencies!

Summary page shows computation vs. communication breakdown



Summary: interFoam.stf

Total time: 5,11e+03 sec. Resources: 32 processes, 4 nodes

Resource usage

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

🟦	Serial Code - 3.4e+03 sec	66.4 %
🟩	OpenMP - 0 sec	0 %
🟥	MPI calls - 1.71e+03 sec	33.5 %

Is your application MPI-bound?

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Allreduce	1.19e+03 sec (23.4 %)
MPI_Waitall	214 sec (4.18 %)
MPI_Isend	119 sec (2.34 %)
MPI_Irecv	68 sec (1.33 %)
MPI_Recv	65.2 sec (1.28 %)

Largest MPI consumers

Where to start with analysis

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use:
Intel® VTune™ Amplifier for:
- algorithmic level tuning with hpc-performance and threading efficiency analysis;
- microarchitecture level tuning with general exploration and bandwidth analysis;
Intel® Advisor for:
- vectorization optimization and thread prototyping.

Use the following command lines to run these tools for the most CPU-bound rank.
Intel® VTune™ Amplifier:

```
mpirun -gtool "amplxe-cl -collect hpc-performance -r result:20" -np 32 -ppn 8 -f
/home/crosales/.lsbatch/1503502863.643740.hostfile interFoam -parallel
```

Intel® Advisor:

```
mpirun -gtool "advixe-cl -collect survey:20" -np 32 -ppn 8 -f
/home/crosales/.lsbatch/1503502863.643740.hostfile interFoam -parallel
```

Next Steps

Views and Charts

- Helps navigate the trace data
- A View can show several Charts
- All Charts in a View are linked to a single:
  - time-span
  - set of threads
  - set of functions
- All Charts follow changes to View (e.g. zooming)

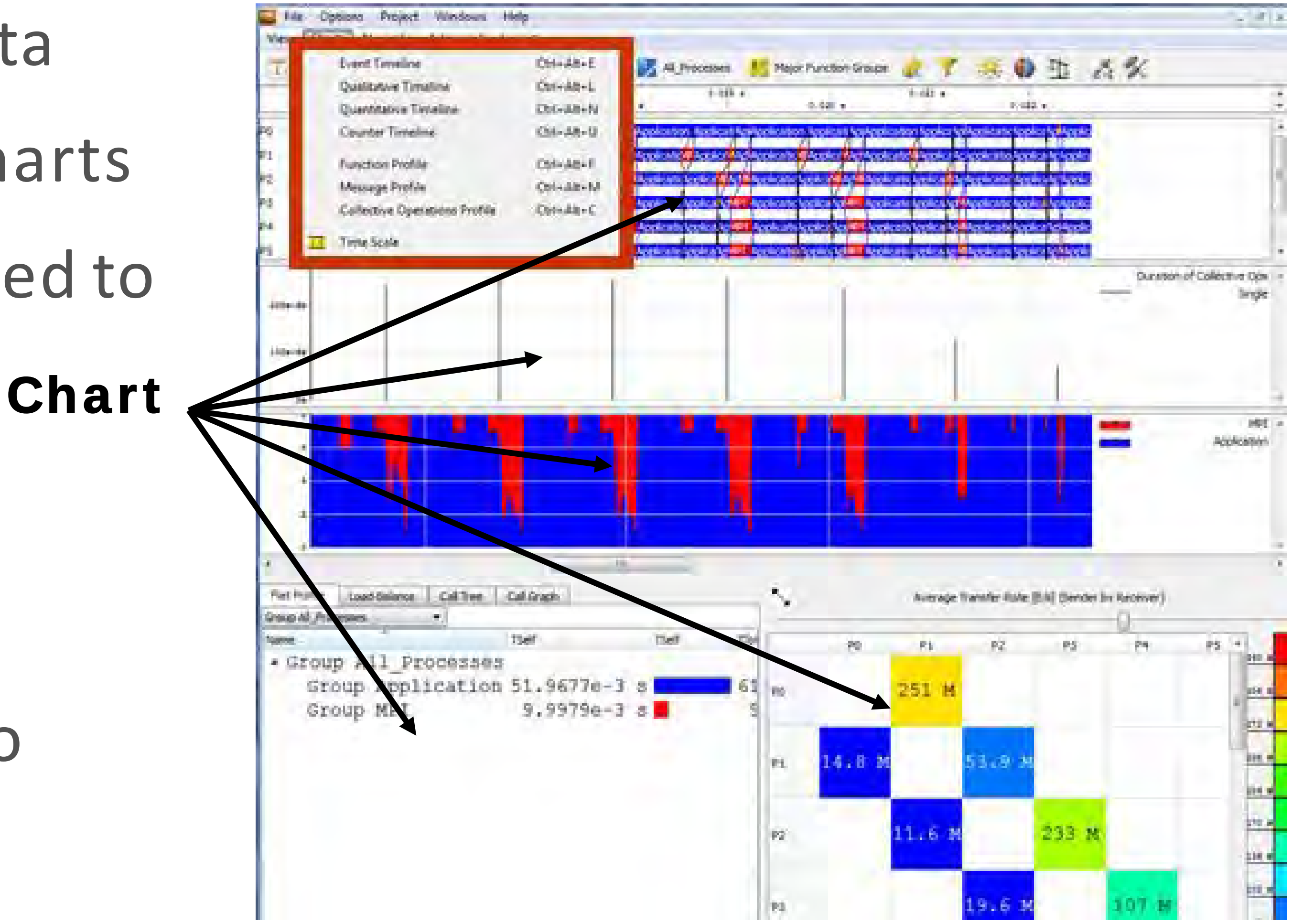
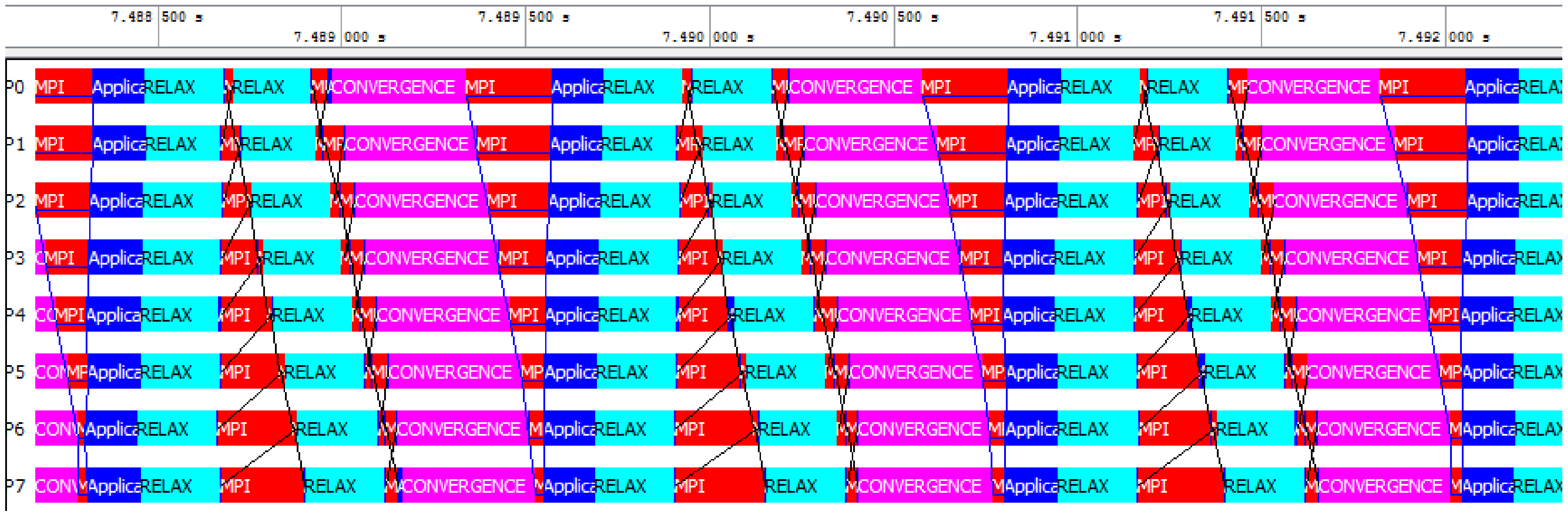


Chart
Event Timeline Ctrl+Alt+E
Qualitative Timeline Ctrl+Alt+L
Quantitative Timeline Ctrl+Alt+N
Counter Timeline Ctrl+Alt+U
Function Profile Ctrl+Alt+F
Message Profile Ctrl+Alt+M
Collective Operations Profile Ctrl+Alt+C
Time Scale
Flat Profile
Load Balance
Call Tree
Call Graph
Group All_Processes
Group Application 51.9677e-3 s
Group MPI 9.9979e-3 s
Average Transfer Rate [B/s] (Sender by Receiver)
251 M
14.8 M
53.9 M
11.6 M
233 M
19.6 M
107 M

Event Timeline



Get detailed impression of program structure

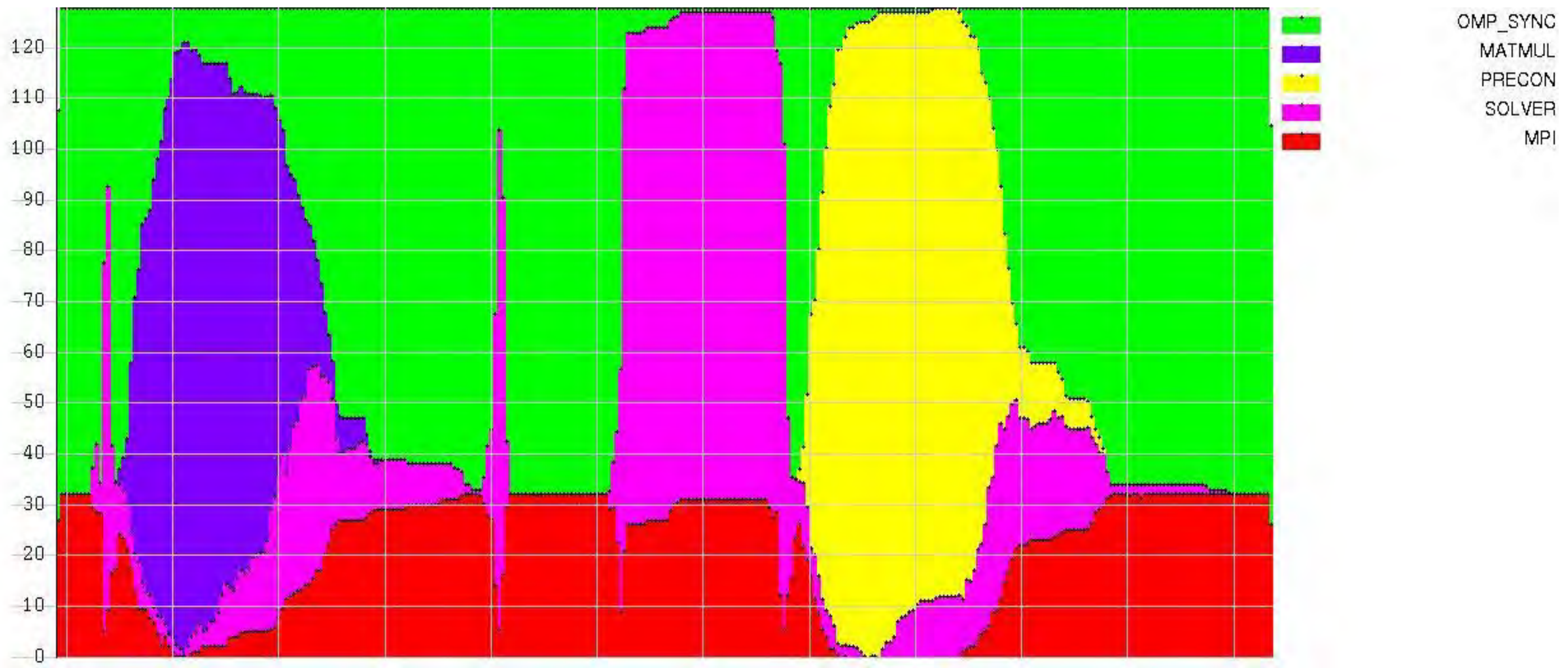
Display functions, messages, and collective operations for each rank/thread along time-axis

Retrieval of detailed event information

Quantitative Timeline

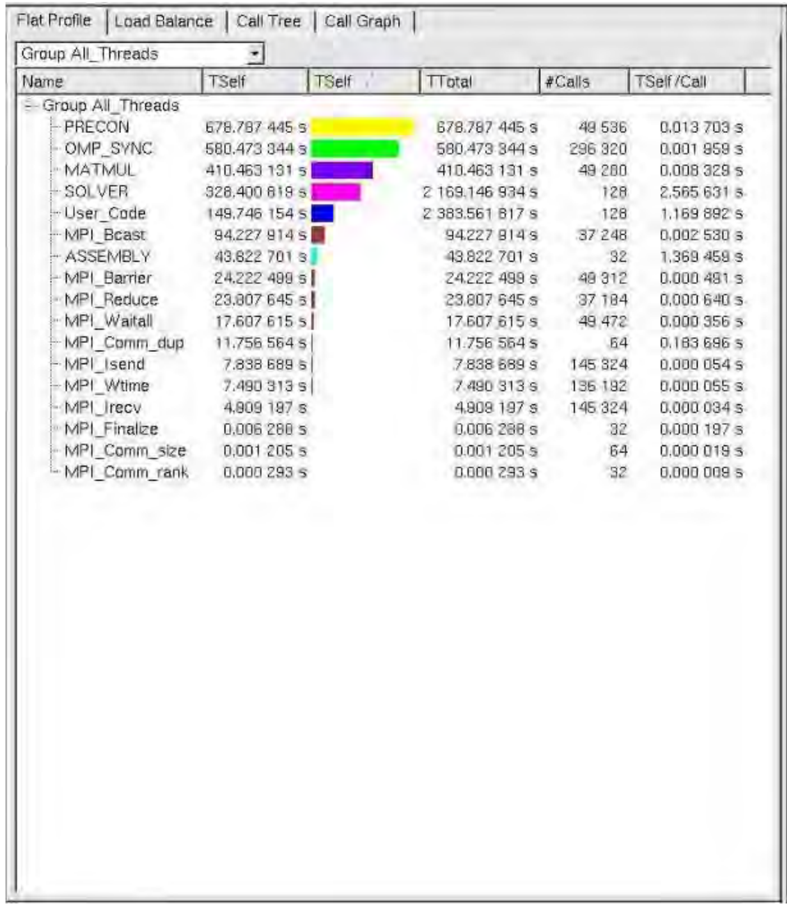
Get impression on parallelism and load balance

Show for every function how many threads/ranks are currently executing it



Flat Function Profile

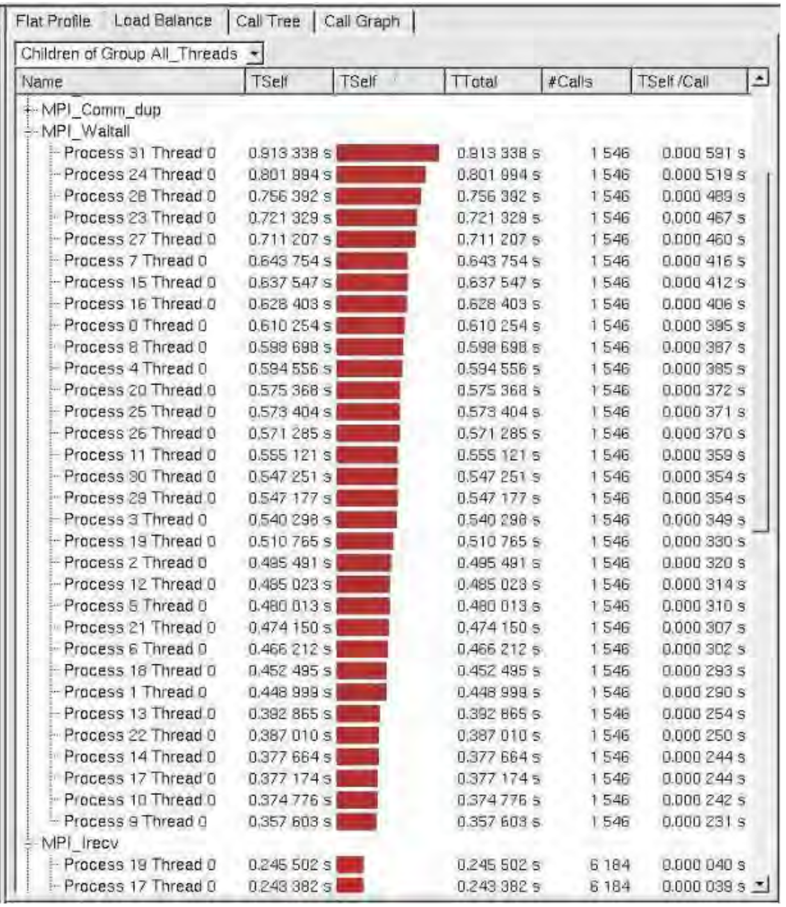
Statistics about functions



Flat Profile | Load Balance | Call Tree | Call Graph

Group All_Threads

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Threads					
PRECON	678.787 445 s	🟨	678.787 445 s	49 536	0.013 703 s
OMP_SYNC	580.473 344 s	🟩	580.473 344 s	296 320	0.001 959 s
MATMUL	410.463 131 s	🟪	410.463 131 s	49 280	0.008 329 s
SOLVER	328.400 819 s	🟪	2 169.146 934 s	128	2.565 631 s
User_Code	149.746 154 s	🟦	2 383.561 817 s	128	1.169 892 s
MPI_Bcast	94.227 914 s	🟫	94.227 914 s	37 248	0.002 530 s
ASSEMBLY	43.822 701 s		43.822 701 s	32	1.369 459 s
MPI_Barrier	24.222 499 s		24.222 499 s	49 312	0.000 491 s
MPI_Reduce	23.807 645 s		23.807 645 s	37 184	0.000 640 s
MPI_Waitall	17.607 615 s		17.607 615 s	49 472	0.000 356 s
MPI_Comm_dup	11.756 564 s		11.756 564 s	64	0.183 696 s
MPI_Isend	7.838 689 s		7.838 689 s	145 324	0.000 054 s
MPI_Wtime	7.490 313 s		7.490 313 s	136 192	0.000 055 s
MPI_Irecv	4.909 197 s		4.909 197 s	145 324	0.000 034 s
MPI_Finalize	0.006 288 s		0.006 288 s	32	0.000 197 s
MPI_Comm_size	0.001 205 s		0.001 205 s	64	0.000 019 s
MPI_Comm_rank	0.000 293 s		0.000 293 s	32	0.000 009 s



Flat Profile | Load Balance | Call Tree | Call Graph

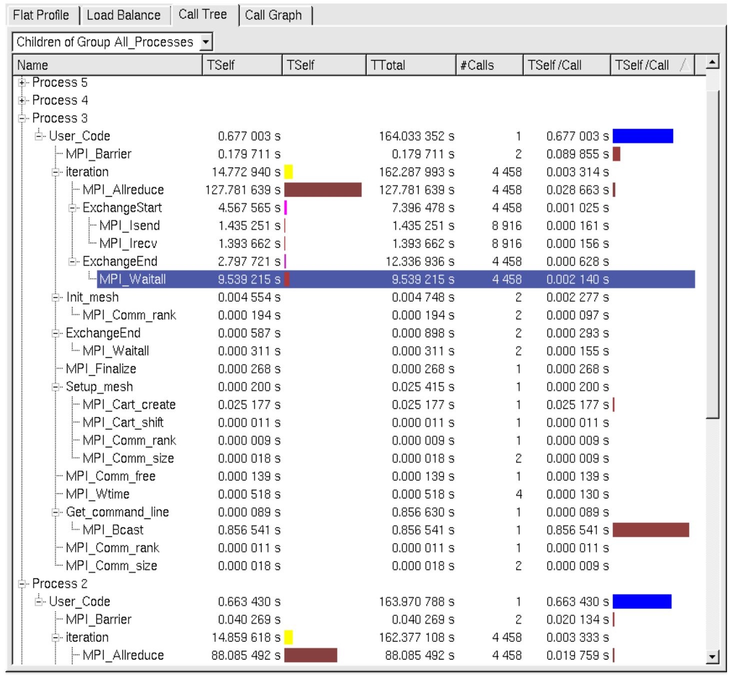
Children of Group All_Threads

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
MPI_Comm_dup					
MPI_Waitall					
Process 31 Thread 0	0.913 338 s	🟥	0.913 338 s	1 546	0.000 591 s
Process 24 Thread 0	0.801 994 s	🟥	0.801 994 s	1 546	0.000 519 s
Process 28 Thread 0	0.756 392 s	🟥	0.756 392 s	1 546	0.000 489 s
Process 23 Thread 0	0.721 329 s	🟥	0.721 328 s	1 546	0.000 467 s
Process 27 Thread 0	0.711 207 s	🟥	0.711 207 s	1 546	0.000 460 s
Process 7 Thread 0	0.643 754 s	🟥	0.643 754 s	1 546	0.000 416 s
Process 15 Thread 0	0.637 547 s	🟥	0.637 547 s	1 546	0.000 412 s
Process 16 Thread 0	0.628 403 s	🟥	0.628 403 s	1 546	0.000 406 s
Process 0 Thread 0	0.610 254 s	🟥	0.610 254 s	1 546	0.000 395 s
Process 8 Thread 0	0.598 698 s	🟥	0.598 698 s	1 546	0.000 387 s
Process 4 Thread 0	0.594 556 s	🟥	0.594 556 s	1 546	0.000 385 s
Process 20 Thread 0	0.575 368 s	🟥	0.575 368 s	1 546	0.000 372 s
Process 25 Thread 0	0.573 404 s	🟥	0.573 404 s	1 546	0.000 371 s
Process 26 Thread 0	0.571 285 s	🟥	0.571 285 s	1 546	0.000 370 s
Process 11 Thread 0	0.555 121 s	🟥	0.555 121 s	1 546	0.000 359 s
Process 30 Thread 0	0.547 251 s	🟥	0.547 251 s	1 546	0.000 354 s
Process 29 Thread 0	0.547 177 s	🟥	0.547 177 s	1 546	0.000 354 s
Process 3 Thread 0	0.540 298 s	🟥	0.540 298 s	1 546	0.000 349 s
Process 19 Thread 0	0.510 765 s	🟥	0.510 765 s	1 546	0.000 330 s
Process 2 Thread 0	0.495 491 s	🟥	0.495 491 s	1 546	0.000 320 s
Process 12 Thread 0	0.485 023 s	🟥	0.485 023 s	1 546	0.000 314 s
Process 5 Thread 0	0.480 013 s	🟥	0.480 013 s	1 546	0.000 310 s
Process 21 Thread 0	0.474 150 s	🟥	0.474 150 s	1 546	0.000 307 s
Process 6 Thread 0	0.466 212 s	🟥	0.466 212 s	1 546	0.000 302 s
Process 18 Thread 0	0.452 495 s	🟥	0.452 495 s	1 546	0.000 293 s
Process 1 Thread 0	0.448 999 s	🟥	0.448 999 s	1 546	0.000 290 s
Process 13 Thread 0	0.392 865 s	🟥	0.392 865 s	1 546	0.000 254 s
Process 22 Thread 0	0.387 010 s	🟥	0.387 010 s	1 546	0.000 250 s
Process 14 Thread 0	0.377 664 s	🟥	0.377 664 s	1 546	0.000 244 s
Process 17 Thread 0	0.377 174 s	🟥	0.377 174 s	1 546	0.000 244 s
Process 10 Thread 0	0.374 776 s	🟥	0.374 776 s	1 546	0.000 242 s
Process 9 Thread 0	0.357 603 s	🟥	0.357 603 s	1 546	0.000 231 s
MPI_Irecv					
Process 19 Thread 0	0.245 502 s	🟥	0.245 502 s	6 184	0.000 040 s
Process 17 Thread 0	0.243 382 s	🟥	0.243 382 s	6 184	0.000 039 s

Call Tree and Call Graph

Function statistics including calling hierarchy

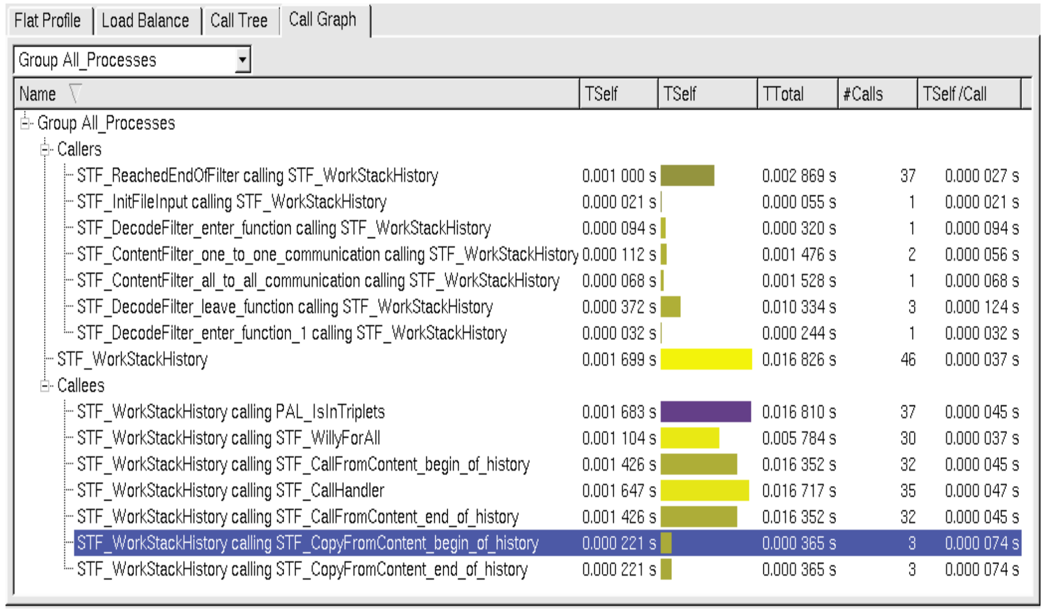
- Call Tree shows call stack
- Call Graph shows calling dependencies



Flat Profile | Load Balance | Call Tree | Call Graph

Children of Group All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call	TSelf /Call
Process 5						
Process 4						
Process 3						
User_Code	0.677 003 s		164.033 352 s	1	0.677 003 s	
MPI_Barrier	0.179 711 s		0.179 711 s	2	0.089 855 s	
iteration	14.772 940 s		162.287 993 s	4 458	0.003 314 s	
MPI_Allreduce	127.781 639 s		127.781 639 s	4 458	0.028 663 s	
ExchangeStart	4.567 565 s		7.396 478 s	4 458	0.001 025 s	
MPI_Isend	1.435 251 s		1.435 251 s	8 916	0.000 161 s	
MPI_Irecv	1.393 662 s		1.393 662 s	8 916	0.000 156 s	
ExchangeEnd	2.797 721 s		12.336 936 s	4 458	0.000 628 s	
MPI_Waitall	9.539 215 s		9.539 215 s	4 458	0.002 140 s	
Init_mesh	0.004 554 s		0.004 748 s	2	0.002 277 s	
MPI_Comm_rank	0.000 194 s		0.000 194 s	2	0.000 097 s	
ExchangeEnd	0.000 587 s		0.000 898 s	2	0.000 293 s	
MPI_Waitall	0.000 311 s		0.000 311 s	2	0.000 155 s	
MPI_Finalize	0.000 268 s		0.000 268 s	1	0.000 268 s	
Setup_mesh	0.000 200 s		0.025 415 s	1	0.000 200 s	
MPI_Cart_create	0.025 177 s		0.025 177 s	1	0.025 177 s	
MPI_Cart_shift	0.000 011 s		0.000 011 s	1	0.000 011 s	
MPI_Comm_rank	0.000 009 s		0.000 009 s	1	0.000 009 s	
MPI_Comm_size	0.000 018 s		0.000 018 s	2	0.000 009 s	
MPI_Comm_free	0.000 139 s		0.000 139 s	1	0.000 139 s	
MPI_Wtime	0.000 518 s		0.000 518 s	4	0.000 130 s	
Get_command_line	0.000 089 s		0.856 630 s	1	0.000 089 s	
MPI_Bcast	0.856 541 s		0.856 541 s	1	0.856 541 s	
MPI_Comm_rank	0.000 011 s		0.000 011 s	1	0.000 011 s	
MPI_Comm_size	0.000 018 s		0.000 018 s	2	0.000 009 s	
Process 2						
User_Code	0.663 430 s		163.970 788 s	1	0.663 430 s	
MPI_Barrier	0.040 269 s		0.040 269 s	2	0.020 134 s	
iteration	14.859 618 s		162.377 108 s	4 458	0.003 333 s	
MPI_Allreduce	88.085 492 s		88.085 492 s	4 458	0.019 759 s	



Flat Profile | Load Balance | Call Tree | Call Graph

Group All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Callers					
STF_ReachedEndOfFilter calling STF_WorkStackHistory	0.001 000 s		0.002 869 s	37	0.000 027 s
STF_InitFileInput calling STF_WorkStackHistory	0.000 021 s		0.000 055 s	1	0.000 021 s
STF_DecodeFilter_enter_function calling STF_WorkStackHistory	0.000 094 s		0.000 320 s	1	0.000 094 s
STF_ContentFilter_one_to_one_communication calling STF_WorkStackHistory	0.000 112 s		0.001 476 s	2	0.000 056 s
STF_ContentFilter_all_to_all_communication calling STF_WorkStackHistory	0.000 068 s		0.001 528 s	1	0.000 068 s
STF_DecodeFilter_leave_function calling STF_WorkStackHistory	0.000 372 s		0.010 334 s	3	0.000 124 s
STF_DecodeFilter_enter_function_1 calling STF_WorkStackHistory	0.000 032 s		0.000 244 s	1	0.000 032 s
STF_WorkStackHistory	0.001 699 s		0.016 826 s	46	0.000 037 s
Callees					
STF_WorkStackHistory calling PAL_IsInTriplets	0.001 683 s		0.016 810 s	37	0.000 045 s
STF_WorkStackHistory calling STF_WillyForAll	0.001 104 s		0.005 784 s	30	0.000 037 s
STF_WorkStackHistory calling STF_CallFromContent_begin_of_history	0.001 426 s		0.016 352 s	32	0.000 045 s
STF_WorkStackHistory calling STF_CallHandler	0.001 647 s		0.016 717 s	35	0.000 047 s
STF_WorkStackHistory calling STF_CallFromContent_end_of_history	0.001 426 s		0.016 352 s	32	0.000 045 s
STF_WorkStackHistory calling STF_CopyFromContent_begin_of_history	0.000 221 s		0.000 365 s	3	0.000 074 s
STF_WorkStackHistory calling STF_CopyFromContent_end_of_history	0.000 221 s		0.000 365 s	3	0.000 074 s

Communication Profiles

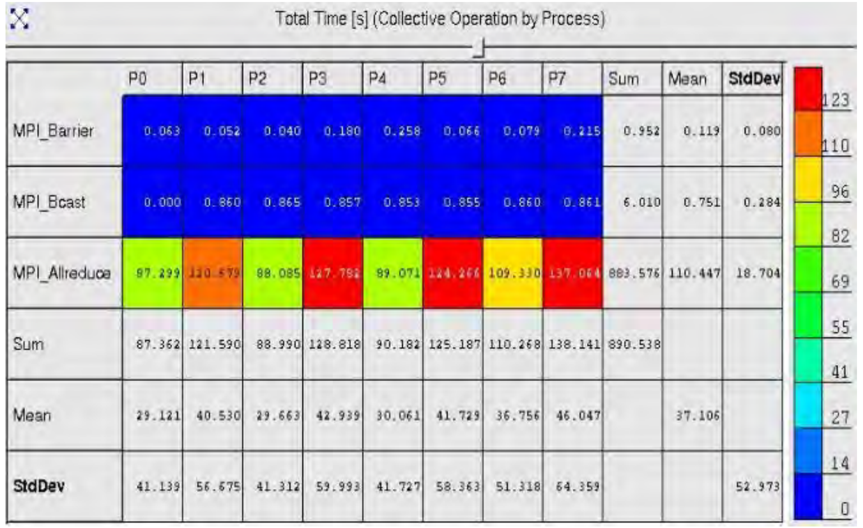
Statistics about point-to-point or collective communication

Matrix supports grouping by attributes in each dimension

- Sender, Receiver, Data volume per msg, Tag, Communicator, Type

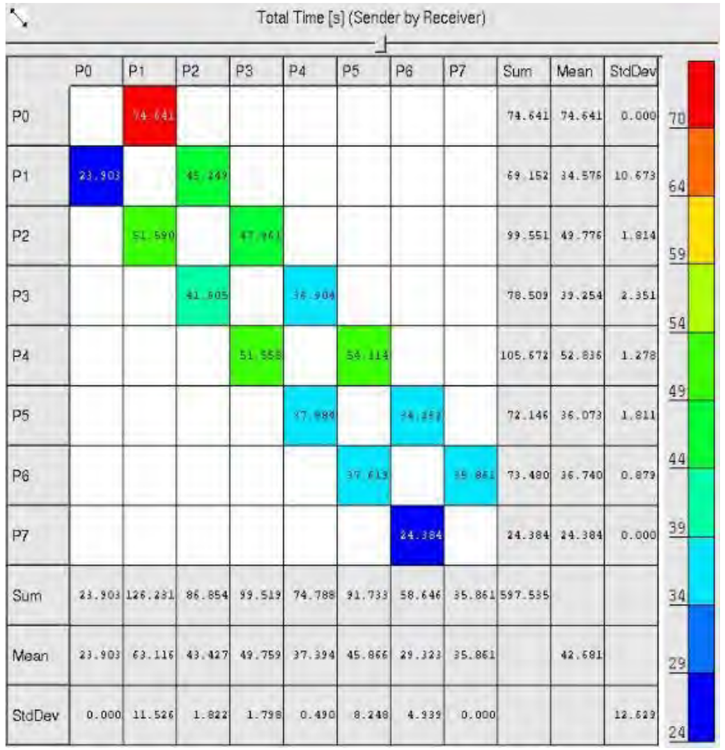
Available attributes

- Count, Bytes transferred, Time, Transfer rate



Total Time [s] (Collective Operation by Process)

	P0	P1	P2	P3	P4	P5	P6	P7	Sum	Mean	StdDev
MPI_Barrier	0.063	0.052	0.040	0.180	0.258	0.066	0.079	0.215	0.952	0.119	0.080
MPI_Bcast	0.000	0.860	0.865	0.857	0.853	0.855	0.860	0.861	6.010	0.751	0.284
MPI_Allreduce	87.299	120.679	88.085	127.782	89.071	124.266	109.330	137.064	883.576	110.447	18.704
Sum	87.362	121.590	88.990	128.818	90.182	125.187	110.268	138.141	890.538		
Mean	29.121	40.530	29.663	42.939	30.061	41.729	36.756	46.047		37.106	
StdDev	41.139	56.675	41.312	59.993	41.727	58.363	51.318	64.359			52.973

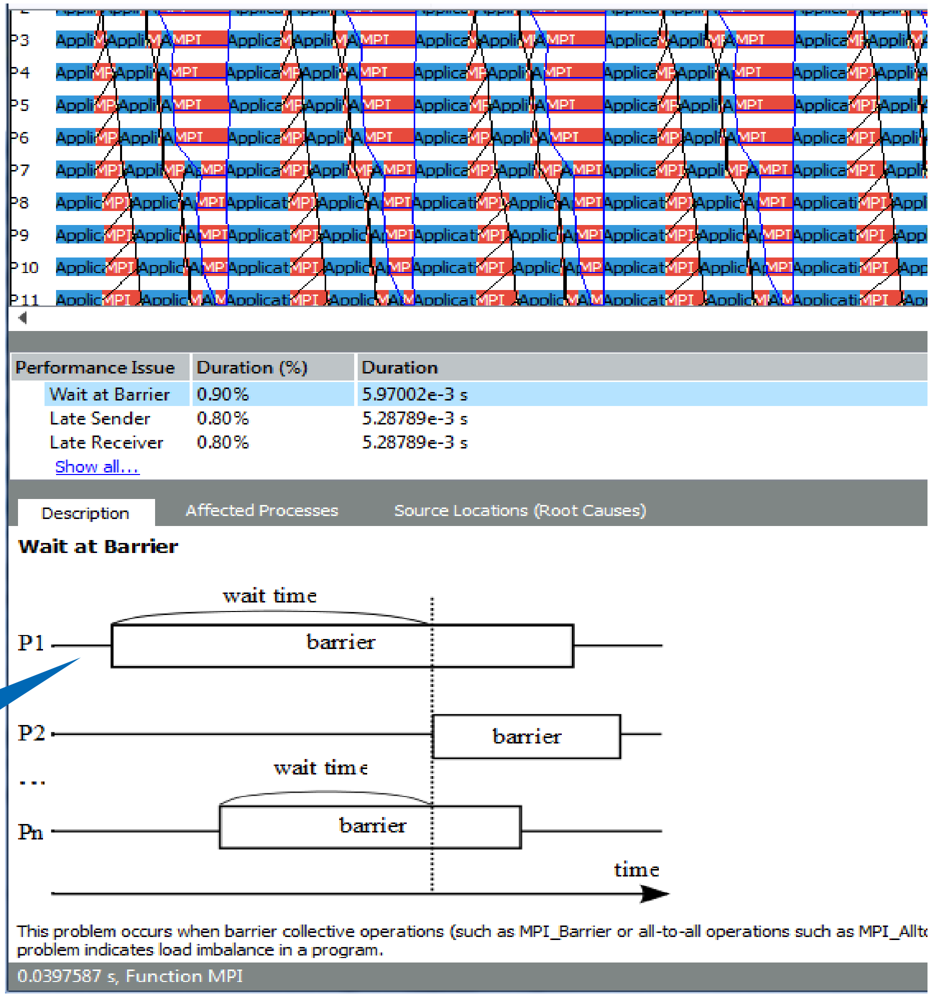


Total Time [s] (Sender by Receiver)

	P0	P1	P2	P3	P4	P5	P6	P7	Sum	Mean	StdDev
P0		74.641							74.641	74.641	0.000
P1	23.903		45.249						69.152	34.576	10.673
P2		51.590		47.961					99.551	49.776	1.814
P3			41.605		36.904				78.509	39.254	2.351
P4				51.558		54.114			105.672	52.836	1.278
P5					37.884		34.262		72.146	36.073	1.811
P6						37.619		35.861	73.480	36.740	0.879
P7							24.384		24.384	24.384	0.000
Sum	23.903	126.231	86.854	99.519	74.788	91.733	58.646	35.861	597.535		
Mean	23.903	63.116	43.427	49.759	37.394	45.866	29.323	35.861		42.681	
StdDev	0.000	11.526	1.822	1.798	0.490	8.248	4.939	0.000			12.623

MPI Performance Assistant

- Automatic Performance Assistant
- Detect common MPI performance issues
- Automated tips on potential solutions



Performance Issue	Duration (%)	Duration
Wait at Barrier	0.90%	5.97002e-3 s
Late Sender	0.80%	5.28789e-3 s
Late Receiver	0.80%	5.28789e-3 s

Show all...

Description | Affected Processes | Source Locations (Root Causes)

Wait at Barrier

This problem occurs when barrier collective operations (such as MPI_Barrier or all-to-all operations such as MPI_Allt
problem indicates load imbalance in a program.

0.0397587 s, Function MPI

Automatically detect performance issues and their impact on runtime

Checking MPI Application Correctness

Runtime Correctness Checks

Integration with Debuggers

MPI Correctness Checking

Solves two problems:

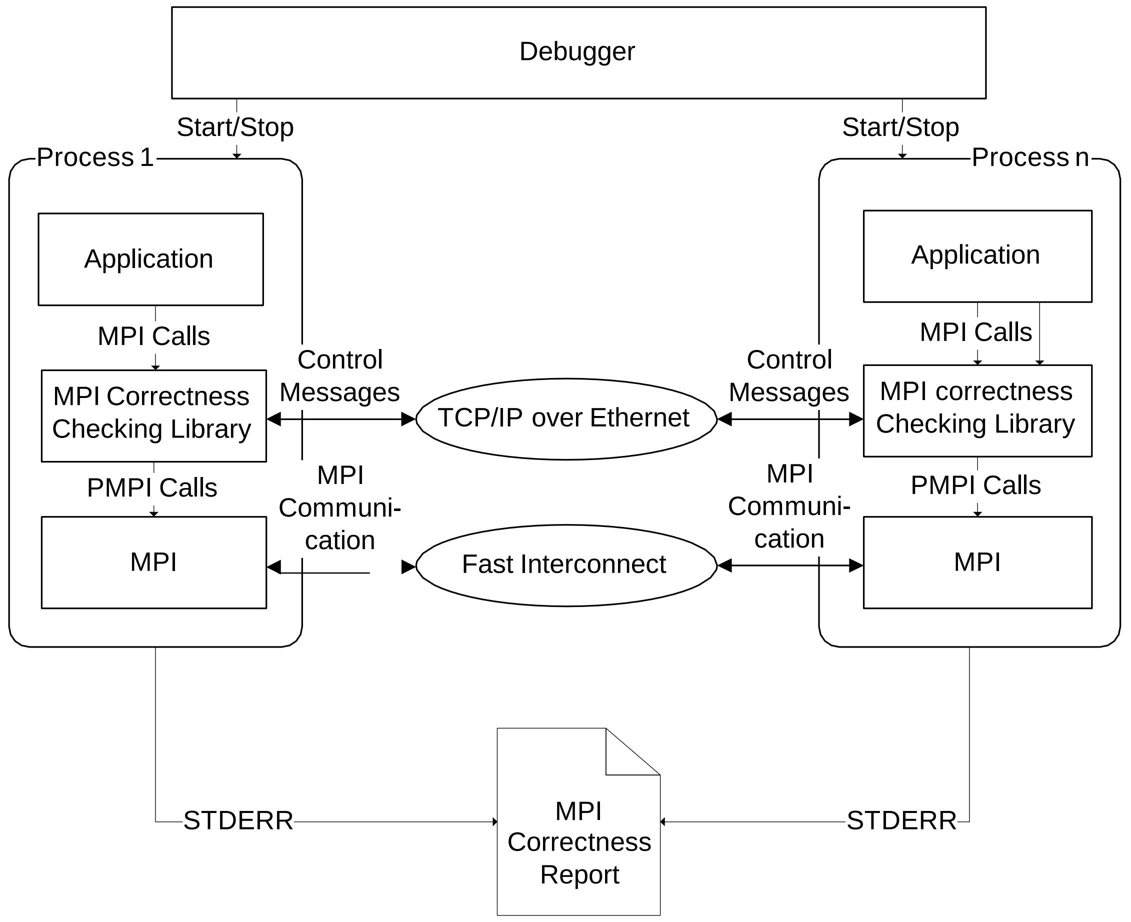
- Finding programming mistakes which need to be fixed by the application developer
- Detecting errors in the execution environment

Two aspects:

- Error Detection – done automatically by the tool
- Error Analysis – manually by the user based on:
  - Information provided about an error
  - Knowledge of source code, system, …

How Correctness Checking Works

- All checks are done at runtime in MPI wrappers
- Detected problems are reported on stderr immediately in textual format
- A debugger can be used to investigate the problem at the moment when it is found



Categories of Checks

- Local checks: isolated to single process
  - Unexpected process termination
  - Buffer handling
  - Request and data type management
  - Parameter errors found by MPI

- Global checks: all processes
  - Global checks for collectives and p2p ops
    - Data type mismatches
    - Corrupted data transmission
    - Pending messages
    - Deadlocks (hard & potential)
  - Global checks for collectives – one report per operation
    - Operation, size, reduction operation, root mismatch
    - Parameter error
    - Mismatched MPI_Comm_free()

Severity of Checks

Levels of severity:

- *Warnings*: application can continue
- *Error*: application can continue but almost certainly not as intended
- *Fatal error*: application must be aborted

Some checks may find both warnings and errors

- Example: CALL_FAILED check due to invalid parameter
  - Invalid parameter in MPI_Send() => msg cannot be sent => *error*
  - Invalid parameter in MPI_Request_free() => resource leak => *warning*

Correctness Checking on Command Line

Command line option via –check_mpi flag for Intel MPI Library:

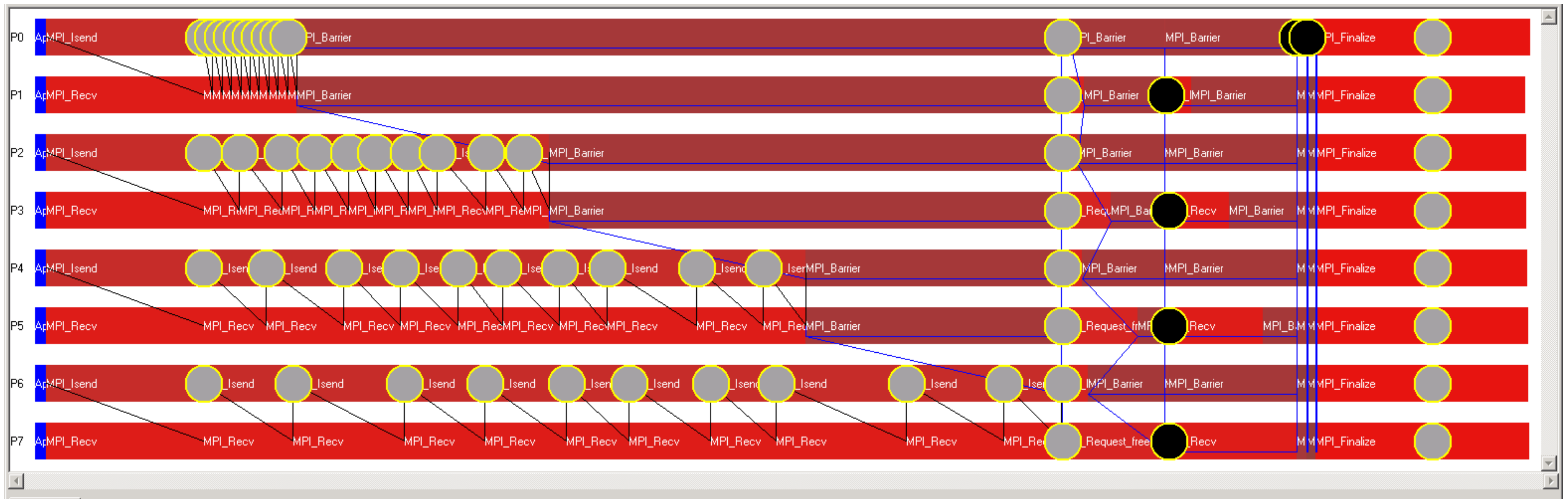
```
$ mpirun –check_mpi -n 2 overlap
[…]
[0] WARNING: LOCAL:MEMORY:OVERLAP: warning
[0] WARNING:    New send buffer overlaps with currently active send buffer at address 0x7fbfffec10.
[0] WARNING:    Control over active buffer was transferred to MPI at:
[0] WARNING:      MPI_Isend(*buf=0x7fbfffec10, count=4, datatype=MPI_INT, dest=0, tag=103,
comm=COMM_SELF [0], *request=0x508980)
[0] WARNING:      overlap.c:104
[0] WARNING:    Control over new buffer is about to be transferred to MPI at:
[0] WARNING:      MPI_Isend(*buf=0x7fbfffec10, count=4, datatype=MPI_INT, dest=0, tag=104,
comm=COMM_SELF [0], *request=0x508984)
[0] WARNING:      overlap.c:105
```

Correctness Checking in GUI

Enable correctness checking info to be added to the tracefile:

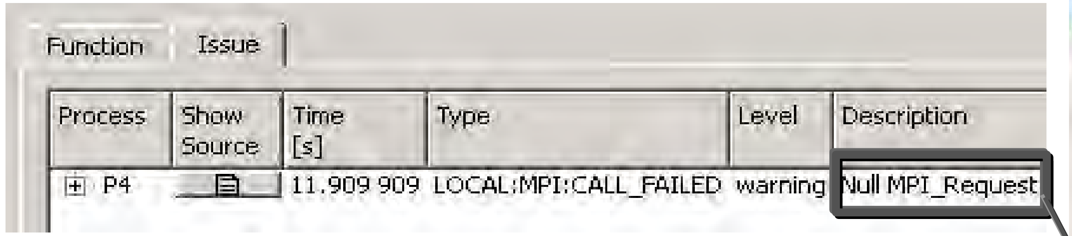
- Enable VT_CHECK_TRACING environment variable:

```
$ mpirun –check_mpi –genv VT_CHECK_TRACING on –n 4 ./a.out
```



● Errors ○ Warnings

Viewing Source Code

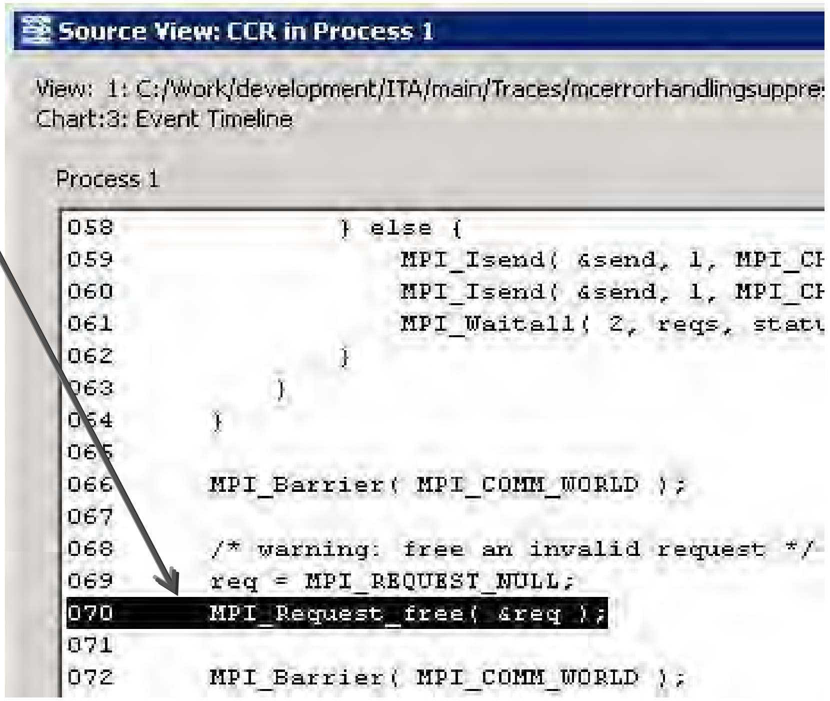


Function | Issue

Process	Show Source	Time [s]	Type	Level	Description
⊞ P4	🗎	11.909 909	LOCAL:MPI:CALL_FAILED	warning	Null MPI_Request

Warnings indicate potential problems that could cause unexpected behavior (e.g., incomplete message requests, overwriting a send/receive buffer, potential deadlock, etc.).

Errors indicate problems that violate the MPI standard or definitely cause behavior not intended by the programmer (e.g., incomplete collectives, API errors, corrupting a send/receive buffer, deadlock, etc.).



Source View: CCR in Process 1

View: 1: C:/Work/development/ITA/main/Traces/mcerrorhandlingsuppre
Chart:3: Event Timeline

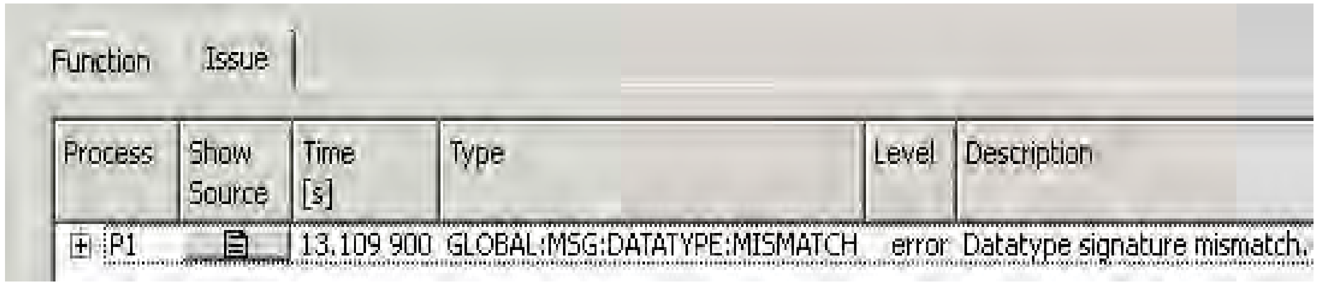
Process 1

```
            } else {
                MPI_Isend( &send, 1, MPI_CH
                MPI_Isend( &send, 1, MPI_CH
                MPI_Waitall( 2, reqs, statu
            }
        }
    }

    MPI_Barrier( MPI_COMM_WORLD );

    /* warning: free an invalid request */
    req = MPI_REQUEST_NULL;
    MPI_Request_free( &req );

    MPI_Barrier( MPI_COMM_WORLD );
```



Function | Issue

Process	Show Source	Time [s]	Type	Level	Description
⊞ P1	🗎	13.109 900	GLOBAL:MSG:DATATYPE:MISMATCH	error	Datatype signature mismatch.

Debugger Integration

Debugger must be in control of application before error is found

A breakpoint must be set in MessageCheckingBreakpoint()

Documentation contains instructions for automating this process for TotalView*, gdb, and idb.

Trace of a Simple MPI Program

Demo

Related Tools

Intel® MPI Benchmarks

Intel® Cluster Checker

Intel® MPI Benchmarks

- Standard benchmarks with OSI-compatible CPL license
  - Enables testing of interconnects, systems, and MPI implementations
  - Comprehensive set of MPI kernels that provide performance measurements for:
    - Point-to-point message-passing
    - Global data movement and computation routines
    - One-sided communications
    - File I/O
    - Supports MPI-1.x, MPI-2.x, and MPI-3.x standards

- What's New:
- Introduction of new benchmarks
  - Measure cumulative bandwidth and message rate values

The Intel® MPI Benchmarks provide a simple and easy way to measure MPI performance on your cluster

Use an Extensive Diagnostic Toolset for High Performance Compute Clusters—Intel® Cluster Checker (for Linux*)

- Ensure Cluster Systems Health
- Expert system approach providing cluster systems expertise - verifies system health: find issues, offers suggested actions
- Provides extensible framework, API for integrated support
- Check 100+ characteristics that may affect operation & performance – improve uptime & productivity

- New in 2019 Update 5 Release: Output & Features Improve Usability & Capabilities
  - New default test with faster execution
  - New predefined user/admin specific tests and in-depth analysis
  - Improved summary output on nodes and issue, details in log files
  - Troubleshooting tests on prerequisites for Intel® MPI Library
  - Support for the latest Intel processors (Intel® Xeon® Platinum 9200 Processor Family)
  - BIOS checking capability for administrators, using 'syscfg' utility

For application developers, cluster architects & users, & system administrators

Online Resources

Intel® MPI Library product page

- www.intel.com/go/mpi

Intel® Trace Analyzer and Collector product page

- www.intel.com/go/traceanalyzer

Intel® Clusters and HPC Technology forums

- http://software.intel.com/en-us/forums/intel-clusters-and-hpc-technology

Intel® MPI Library Tuning Files

- https://software.intel.com/en-us/articles/replacing-tuning-configuration-files-in-intel-mpi-library

Intel® Cluster Checker

- https://software.intel.com/content/www/us/en/develop/tools/cluster-checker.html

Backup

Environment Propagation

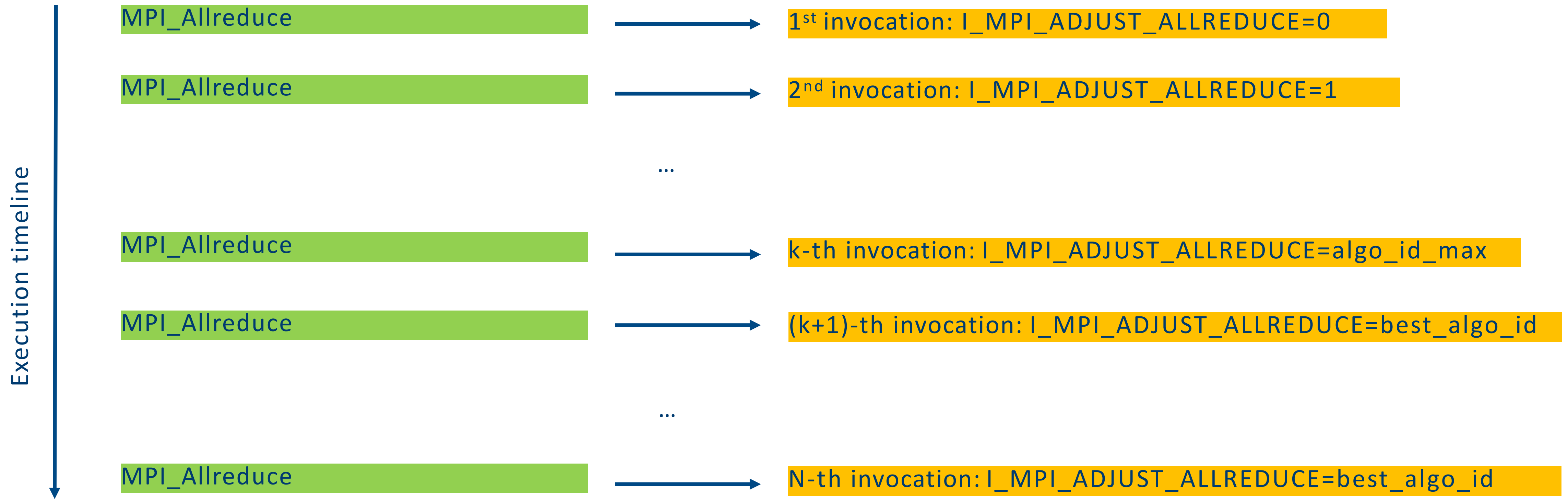
- Use –[g]env[*] to control environment propagation
  - Adding g propagates to all ranks, otherwise only to ranks in current argument set
- -env <variable> <value> Set <variable> to <value>
- -envuser All user environment variables, with a few exceptions (Default)
- -envall All environment variables
- -envnone No environment variables
- -envlist <variable list> Only the listed variables

Autotuner Detail

Intel® MPI library tuning approaches

	mpitune	mpitune /fast tuner	autotune r
Micro benchmark tuning	🟩	🟩	🟨
Application tuning	🟥	🟨	🟩
Easy of use	🟥	🟥	🟩
Cluster time	🟥	🟨	🟩
Adoption to environment	🟥	🟥	🟩

Intel® MPI Library 2019 autotuner tuning flow



MPI_Allreduce → 1st invocation: I_MPI_ADJUST_ALLREDUCE=0

MPI_Allreduce → 2nd invocation: I_MPI_ADJUST_ALLREDUCE=1

…

MPI_Allreduce → k-th invocation: I_MPI_ADJUST_ALLREDUCE=algo_id_max

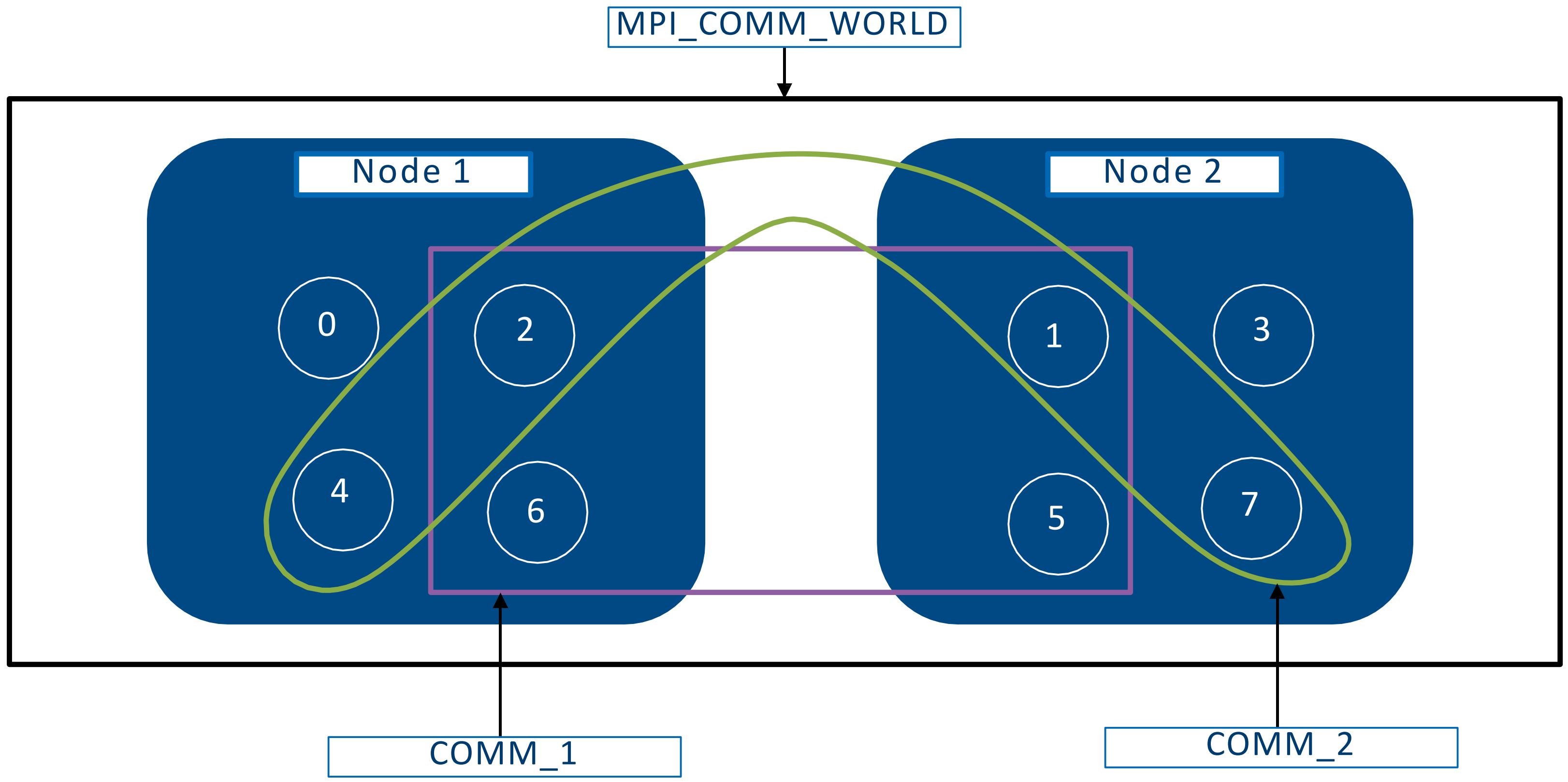
MPI_Allreduce → (k+1)-th invocation: I_MPI_ADJUST_ALLREDUCE=best_algo_id

…

MPI_Allreduce → N-th invocation: I_MPI_ADJUST_ALLREDUCE=best_algo_id

- No extra calls. Pure **application driven** tuning
- The procedure is performed for each message size and for each communicator

Autotuner communicator specific tuning



Each communicator has its own tuning. (E.g. COMM_1 and COMM_2 have independent tuning)

Get started with autotuner

Step 1 – Enable autotuner and store results (store is optional):

```
$ export I_MPI_TUNING_MODE=auto

$ export I_MPI_TUNING_BIN_DUMP=./tuning_results.dat

$ mpirun -n 96 -ppn 48 IMB-MPI1 allreduce -iter 1000,800 -time 4800
```

Step 2 – Use the results of autotuner for consecutive launches (optional):

```
$ export I_MPI_TUNING_BIN=./tuning_results.dat

$ mpirun -n 96 -ppn 48 IMB-MPI1 allreduce -iter 1000,800 -time 4800
```

NOTE: You may adjust number of tuning iterations (minimal overhead/maximum precision balance) and use autotuner with every application run without results storing.

Environment Variables. Main flow control

I_MPI_TUNING_MODE=<auto|auto:application|auto:cluster> (**disabled** by default)

I_MPI_TUNING_AUTO_ITER_NUM=<number> Tuning iterations number (**1** by default).

I_MPI_TUNING_AUTO_SYNC=<0|1> Call internal barrier on every tuning iteration (**disabled** by default)

I_MPI_TUNING_AUTO_WARMUP_ITER_NUM=<number> Warmup iterations number (**1** by default).

NOTE: Assume that there are around 30 algorithms to be iterated. E.g. Application has 10000 invocations of MPI_Allreduce 8KB. For full tuning cycle I_MPI_TUNING_AUTO_ITER_NUM may be in 30 to 300 (if there is no warmup part) range. High value is recommended for the best precision. Iteration number for large messages may depend on I_MPI_TUNING_AUTO_ITER_POLICY_THRESHOLD.

I_MPI_TUNING_AUTO_SYNC is highly recommended for tuning file store scenario.

Environment Variables. Tuning scope and storage control

I_MPI_TUNING_AUTO_COMM_LIST=<comm_id_1, … , comm_id_k> List of communicators to be tuned (all communicators by default)

I_MPI_TUNING_AUTO_COMM_USER=<0|1> Enable user defined comm_id through MPI_Info object. (**disabled** by default)

I_MPI_TUNING_AUTO_COMM_DEFAULT=<0|1> Default/universal comm_ids. (**disabled** by default)

I_MPI_TUNING_AUTO_STORAGE_SIZE=<size> Max per-communicator tuning storage size (**512KB** by default)

NOTE: You may use Intel® VTune™ Profiler's Application Performance Snapshot for per communicator MPI cost analysis and narrow tuning scope.

I_MPI_TUNING_AUTO_COMM_DEFAULT disables comm_id check (allows to get universal tuning)

Intel® VTune™ Profiler's Application Performance Snapshot (APS) per communicator analysis

1. Source apsvars.sh:

```
$ source <path_to_aps>/apsvars.sh
```

2. Gather APS statistics:

```
$ export MPS_STAT_LEVEL=5
$ export APS_COLLECT_COMM_IDS=1
$ mpirun -n 4 -ppn 2 aps IMB-MPI1 allreduce -iter 1000,800
```

3. Generate an APS report:

```
$ aps-report aps_result_20190228/ -lFE
```

https://software.intel.com/sites/products/snapshots/application-snapshot/

Available with Intel® VTune™ Profiler's Application Performance Snapshot Update 4

Intel® VTune™ Profiler's Application Performance Snapshot (APS) per communicator analysis

4. Get the results:

```
| Communicators used in the application
|-------------------------------------------------------------------------------------
| Communicator Id      Communicator Size     Time (Rank Average)(sec) Ranks
|-------------------------------------------------------------------------------------
  4611686018431582688   4                    1.80 (0.45)              0,1,2,3
|-------------------------------------------------------------------------------------
  4611686018431582208   4                    0.59 (0.15)              0,1,2,3
|-------------------------------------------------------------------------------------
```

Intel® VTune™ Profiler's Application Performance Snapshot (APS) Interoperability

5. Specify communicators to be tuned:

```
$ export I_MPI_TUNING_AUTO_COMM_LIST=4611686018431582688
$ export I_MPI_TUNING_MODE=auto
$ mpirun -n 96 -ppn 48 IMB-MPI1 allreduce -iter 1000,800 -time 4800
```

NOTE: I_MPI_TUNING_AUTO_ITER_POLICY may impact tuning cycle for large messages. Please check that you have enough application level invocations

HANDLING HETEROGENEOUS JOBS

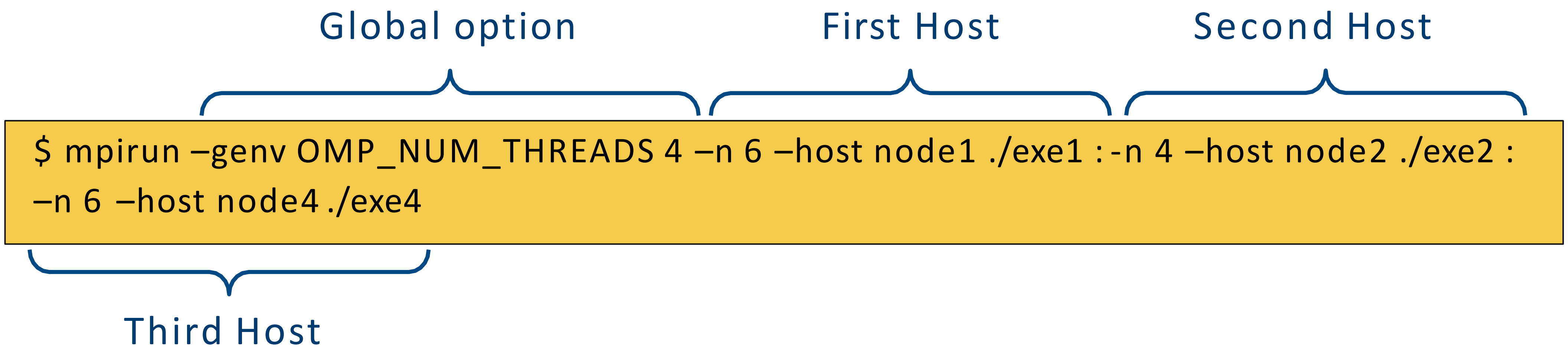
Global Options vs. Local Options

- Global Options are applied to all ranks
  - -ppn, -genv, …
- Local Options are applied to a subset of ranks
  - -n, -host, -env, …
- WARNING: Some options can be set as local options via environment variable, but must be consistent across job
  - Collective algorithms
  - Fabric selection and parameters

Configuration Files and Argument Sets

- Arguments Sets are used on the command line
- Configuration Files are pulled from the file specified by *–configfile <configfile>*
- Global arguments appear first (first line, or at beginning of first argument set)
- Local arguments for each argument set next
- Separated by : on command line (don't separate globals), new line in configfile
- Can be used to run heterogeneous binaries, different arguments for each binary, different environment variables, etc.
- All ranks combined in order specified into one job

Command Line Argument Set



```
$ mpirun –genv OMP_NUM_THREADS 4 –n 6 –host node1 ./exe1 : -n 4 –host node2 ./exe2 :
–n 6 –host node4 ./exe4
```

- Host 1 runs “exe1” on “node1” using 6 MPI tasks and 4 threads per MPI task
- No limit to number of different host or executables
- For high numbers of hosts a configuration file is more convenient…

Configuration File

- Configuration file allows flexibility and automation
- Notice commented out line – simple to change host assignment

```
$ cat theconfigfile
-genv OMP_NUM_THREADS 4
-n 6 –host node1 ./exe1
-n 4 –host node2 ./exe2
# -n 4 –host dead_node3 ./exe3
-n 6 –host node4 ./exe4
```

- Launching job is straightforward

```
$ mpirun –configfile theconfigfile
```