Application Performance Snapshot

Performance Overview at Your Fingertips

Application Performance Snapshot (APS)

High-level overview of application performance

Merges previous Application Performance Snapshot with MPI Performance Snapshot

Identify primary optimization areas and next steps in analysis

Extremely easy to use

Informative, actionable data in clean HTML report

- Also includes recommendations for next steps in analysis

Detailed reports available via command line

Scales to large jobs

APS Usage

Setup Environment

$ source <APS_Install_dir>/apsvars.sh

Run Application

$ mpirun <mpi options> **aps** <application and args>

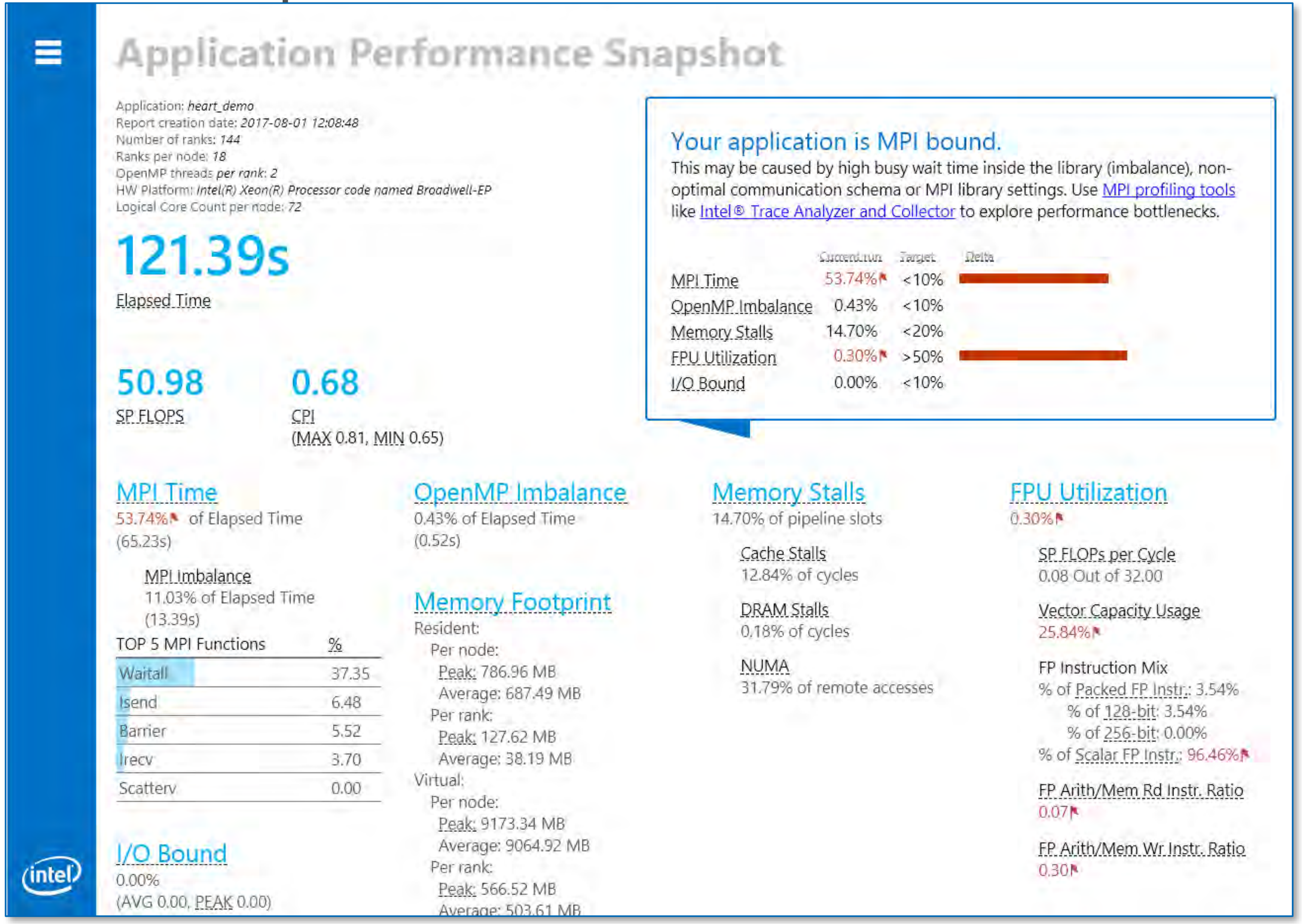
Generate Report on Results

$ aps --report <result folder>

Generate advanced CL reports on Results

$ aps-report -<option> <result folder>

APS HTML Report



Application Performance Snapshot

Application: *heart_demo*
Report creation date: *2017-08-01 12:08:48*
Number of ranks: *144*
Ranks per node: *18*
OpenMP threads per rank: *2*
HW Platform: *Intel(R) Xeon(R) Processor code named Broadwell-EP*
Logical Core Count per node: *72*

121.39s
Elapsed Time

50.98
SP FLOPS

0.68
CPI
(MAX 0.81, MIN 0.65)

Your application is MPI bound.
This may be caused by high busy wait time inside the library (imbalance), non-optimal communication schema or MPI library settings. Use MPI profiling tools like Intel® Trace Analyzer and Collector to explore performance bottlenecks.

	Current run	Target	Delta
MPI Time	53.74%	<10%	
OpenMP Imbalance	0.43%	<10%	
Memory Stalls	14.70%	<20%	
FPU Utilization	0.30%	>50%	
I/O Bound	0.00%	<10%	

MPI Time
53.74% of Elapsed Time
(65.23s)

MPI Imbalance
11.03% of Elapsed Time
(13.39s)

TOP 5 MPI Functions	%
Waitall	37.35
Isend	6.48
Barrier	5.52
Irecv	3.70
Scatterv	0.00

I/O Bound
0.00%
(AVG 0.00, PEAK 0.00)

OpenMP Imbalance
0.43% of Elapsed Time
(0.52s)

Memory Footprint
Resident:
Per node:
Peak: 786.96 MB
Average: 687.49 MB
Per rank:
Peak: 127.62 MB
Average: 38.19 MB
Virtual:
Per node:
Peak: 9173.34 MB
Average: 9064.92 MB
Per rank:
Peak: 566.52 MB
Average: 503.61 MB

Memory Stalls
14.70% of pipeline slots

Cache Stalls
12.84% of cycles

DRAM Stalls
0.18% of cycles

NUMA
31.79% of remote accesses

FPU Utilization
0.30%

SP FLOPs per Cycle
0.08 Out of 32.00

Vector Capacity Usage
25.84%

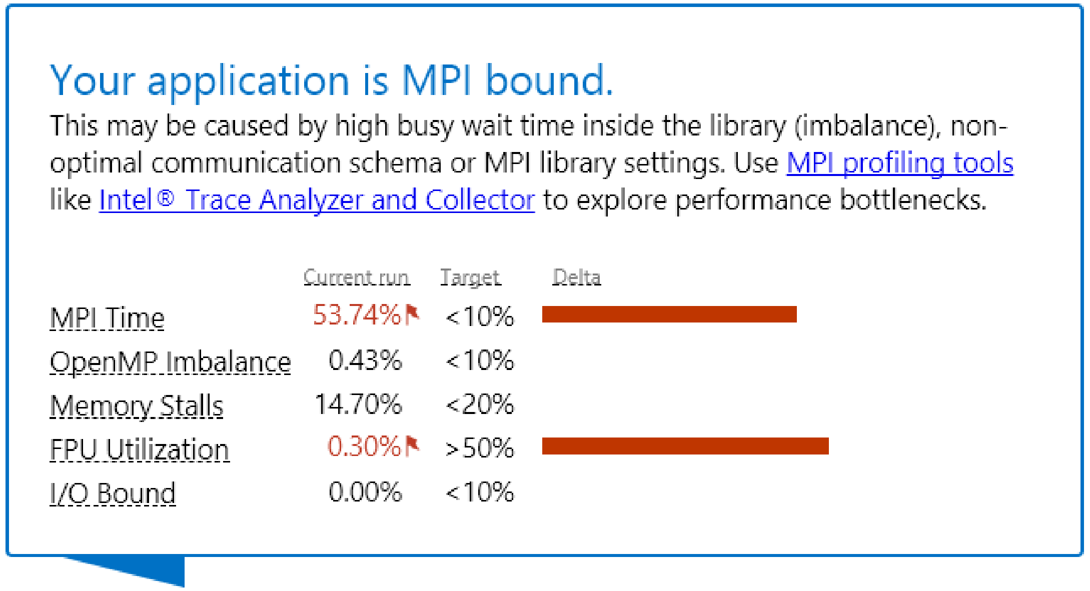
FP Instruction Mix
% of Packed FP Instr.: 3.54%
% of 128-bit: 3.54%
% of 256-bit: 0.00%
% of Scalar FP Instr.: 96.46%

FP Arith/Mem Rd Instr. Ratio
0.07

FP Arith/Mem Wr Instr. Ratio
0.30

HTML Report Breakdown - Overview

- Overview shows all areas and relative impact on code performance
- Provides recommendation for next step in performance analysis



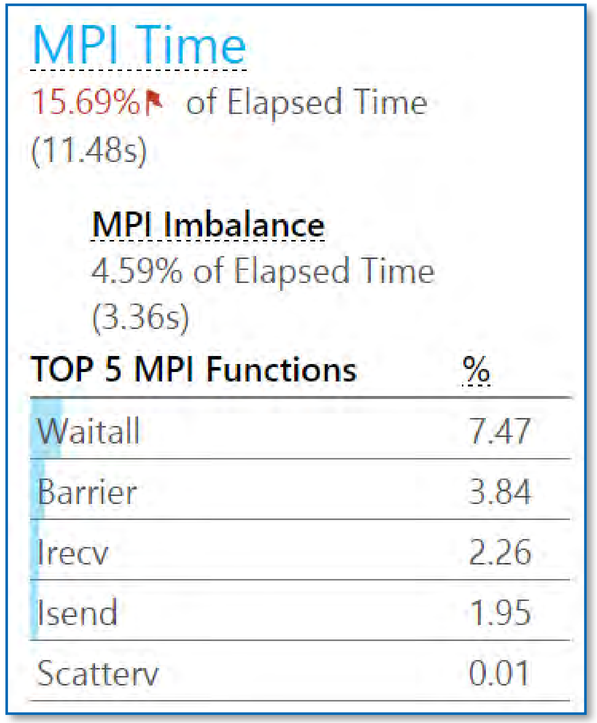
Your application is MPI bound.

This may be caused by high busy wait time inside the library (imbalance), non-optimal communication schema or MPI library settings. Use MPI profiling tools like Intel® Trace Analyzer and Collector to explore performance bottlenecks.

	Current run	Target	Delta
MPI Time	53.74% ⚑	<10%	🟥
OpenMP Imbalance	0.43%	<10%	
Memory Stalls	14.70%	<20%	
FPU Utilization	0.30% ⚑	>50%	🟥
I/O Bound	0.00%	<10%	

HTML Report Breakdown – Parallel Runtimes

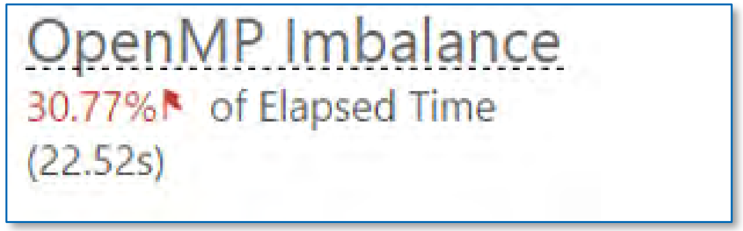
- MPI Time
  - How much time was spent in MPI calls
  - Average by ranks with % of Elapsed time
  - Available for MPICH-based MPIs
- MPI Imbalance
  - Unproductive time spent in MPI library waiting for data
  - Available for Intel MPI
- OpenMP Imbalance
  - Time spent at OpenMP Synchronization Barriers normalized by number of threads
  - Available for Intel OpenMP



MPI Time
15.69% of Elapsed Time
(11.48s)

MPI Imbalance
4.59% of Elapsed Time
(3.36s)

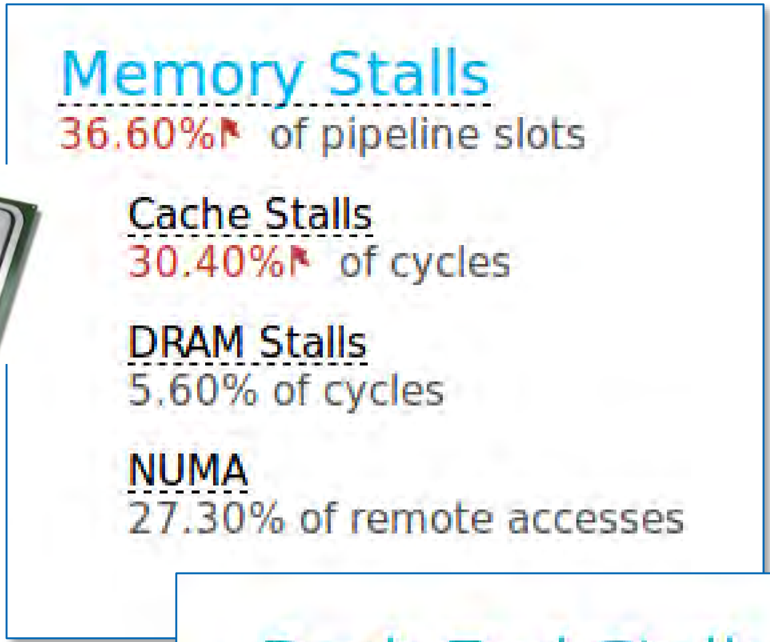
TOP 5 MPI Functions	%
Waitall	7.47
Barrier	3.84
Irecv	2.26
Isend	1.95
Scatterv	0.01



OpenMP Imbalance
30.77% of Elapsed Time
(22.52s)

HTML Report Breakdown - Memory

- Memory stalls measurement with breakdown by cache and DRAM
- NUMA ratio
- KNL: back-end stalls with L2-demand access efficiency

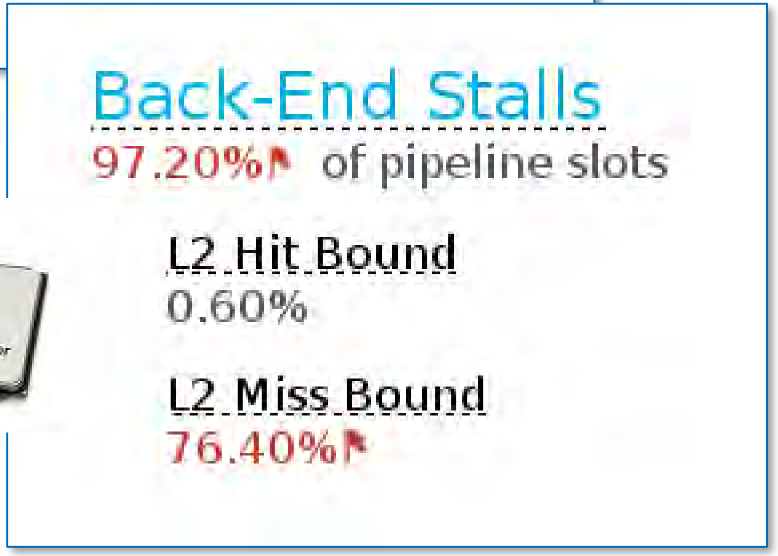


Memory Stalls
36.60% of pipeline slots

Cache Stalls
30.40% of cycles

DRAM Stalls
5.60% of cycles

NUMA
27.30% of remote accesses



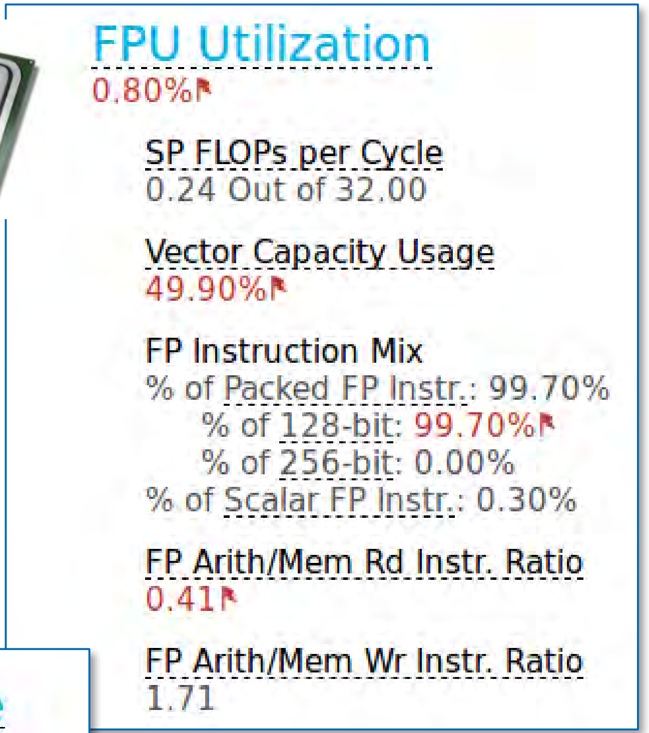
Back-End Stalls
97.20% of pipeline slots

L2 Hit Bound
0.60%

L2 Miss Bound
76.40%

HTML Report Breakdown – Vectorization

- FPU Utilization based on HW-event statistics with
  - Breakdown by vector/scalar instructions
  - Floating point vs memory instruction ratio
- KNL: SIMD Instr. per Cycle
  - Scalar vs. vectorized instructions



FPU Utilization
0.80%

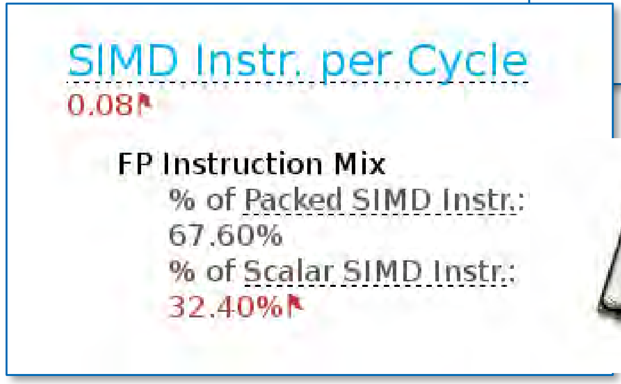
SP FLOPs per Cycle
0.24 Out of 32.00

Vector Capacity Usage
49.90%

FP Instruction Mix
% of Packed FP Instr.: 99.70%
% of 128-bit: 99.70%
% of 256-bit: 0.00%
% of Scalar FP Instr.: 0.30%

FP Arith/Mem Rd Instr. Ratio
0.41

FP Arith/Mem Wr Instr. Ratio
1.71



SIMD Instr. per Cycle
0.08

FP Instruction Mix
% of Packed SIMD Instr.:
67.60%
% of Scalar SIMD Instr.:
32.40%

Intel® VTune™ Profiler

Profiling MPI and Hybrid MPI+Threads Applications

Using Intel® VTune™ Profiler on MPI programs

- Run VTune underneath MPI
  - Results are grouped into one result per node
    - <result folder>.<node name>
  - Within result, ranks indicate rank number

```
$ mpirun <mpi args> amplxe-cl <vtune args> -- <application and args>
```

Easier Multi-Rank Analysis of MPI + OpenMP

Tune hybrid parallelism using ITAC + VTune Profiler

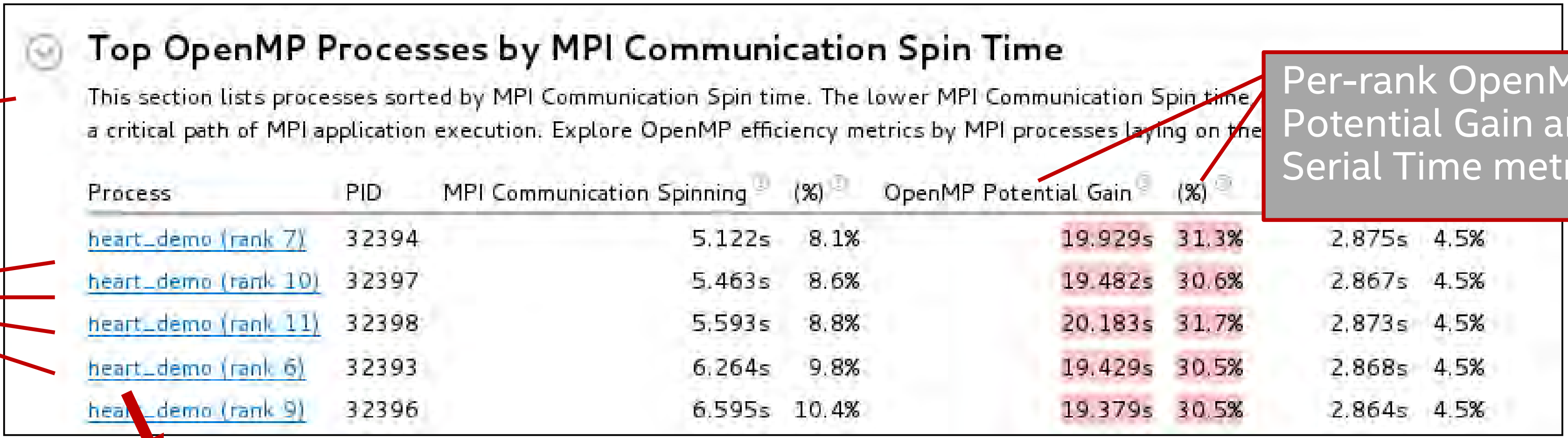
Tune OpenMP performance of high impact ranks in VTune Profiler

Ranks sorted by MPI Communication Spins – ranks on the critical path are on the top

Process names link to OpenMP metrics

Detailed OpenMP metrics per MPI ranks

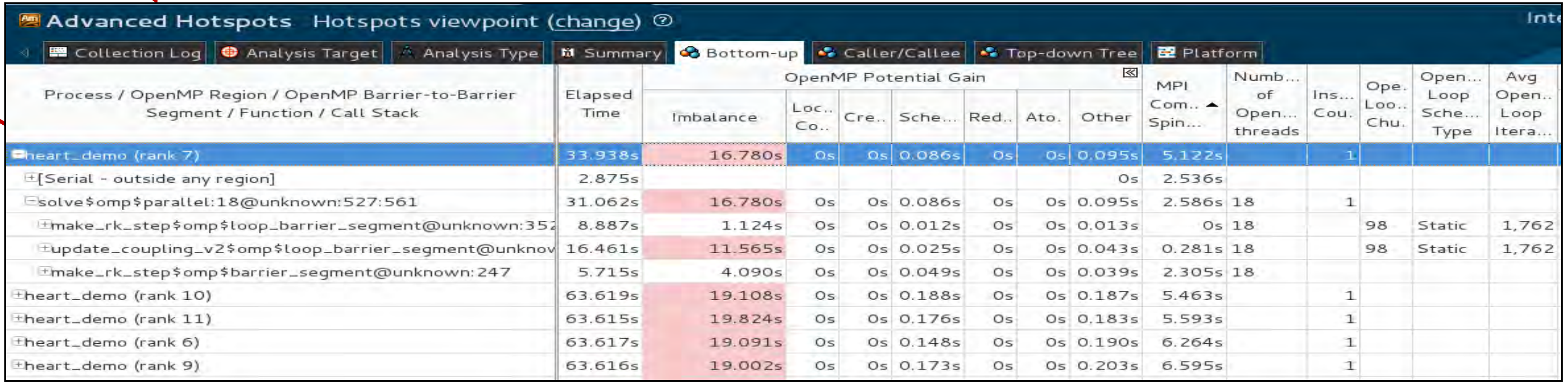
Per-rank OpenMP Potential Gain and Serial Time metrics



Top OpenMP Processes by MPI Communication Spin Time

This section lists processes sorted by MPI Communication Spin time. The lower MPI Communication Spin time a critical path of MPI application execution. Explore OpenMP efficiency metrics by MPI processes laying on the

Process	PID	MPI Communication Spinning	(%)	OpenMP Potential Gain	(%)		
heart_demo (rank 7)	32394	5.122s	8.1%	19.929s	31.3%	2.875s	4.5%
heart_demo (rank 10)	32397	5.463s	8.6%	19.482s	30.6%	2.867s	4.5%
heart_demo (rank 11)	32398	5.593s	8.8%	20.183s	31.7%	2.873s	4.5%
heart_demo (rank 6)	32393	6.264s	9.8%	19.429s	30.5%	2.868s	4.5%
heart_demo (rank 9)	32396	6.595s	10.4%	19.379s	30.5%	2.864s	4.5%



Process / OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	Elapsed Time	OpenMP Potential Gain							MPI Com.. Spin...	Numb... of Open... threads	Ins... Cou.	Ope. Loo.. Chu.	Open... Loop Sche... Type	Avg Open.. Loop Itera...
		Imbalance	Loc.. Co..	Cre..	Sche...	Red..	Ato.	Other						
heart_demo (rank 7)	33.938s	16.780s	0s	0s	0.086s	0s	0s	0.095s	5.122s		1			
[Serial - outside any region]	2.875s							0s	2.536s					
solve$omp$parallel:18@unknown:527:561	31.062s	16.780s	0s	0s	0.086s	0s	0s	0.095s	2.586s	18	1			
make_rk_step$omp$loop_barrier_segment@unknown:352	8.887s	1.124s	0s	0s	0.012s	0s	0s	0.013s	0s	18		98	Static	1,762
update_coupling_v2$omp$loop_barrier_segment@unknow	16.461s	11.565s	0s	0s	0.025s	0s	0s	0.043s	0.281s	18		98	Static	1,762
make_rk_step$omp$barrier_segment@unknown:247	5.715s	4.090s	0s	0s	0.049s	0s	0s	0.039s	2.305s	18				
heart_demo (rank 10)	63.619s	19.108s	0s	0s	0.188s	0s	0s	0.187s	5.463s		1			
heart_demo (rank 11)	63.615s	19.824s	0s	0s	0.176s	0s	0s	0.183s	5.593s		1			
heart_demo (rank 6)	63.617s	19.091s	0s	0s	0.148s	0s	0s	0.190s	6.264s		1			
heart_demo (rank 9)	63.616s	19.002s	0s	0s	0.173s	0s	0s	0.203s	6.595s		1			

Intel® Inspector

- **Analysis Process**
- Launch Intel® Inspector
  - Use mpirun
  - List your app as a parameter
- Results organized by MPI rank
- Review results
  - Graphical user interface
  - Command line report

Find errors earlier when they are less expensive to fix

Using Intel® Inspector with MPI

Use the command-line tool under the MPI run script to gather report data

```
$ mpirun -n 4 inspxe-cl -r my_result -collect mi1 -- ./test
```

Argument Sets can be used for more control

- Only collect data on certain ranks
- Different collections or options on different ranks

A unique results directory is created for each analyzed MPI rank

Launch the GUI and view the results for each rank

Hands-On

Build and Run a Simple MPI Program

Simple MPI Program - Setup

- Intel MPI provides a simple example in C, C++, Fortran, fortran90 and Java under the directory <impi_install_dir>/test/
- Copy the file to your working directory :

-
  ```
  $ cp /opt/intel/impi/<version>/test/test.c .
  ```
- Ensure your environment is set to use the Intel MPI compiler wrappers:

-
  ```
  $ source /opt/intel/impi/<version>/bin64/mpivars.sh
  ```
- You can check that the environment is set properly by checking if the variable I_MPI_ROOT is set:

-
  ```
  $ echo $I_MPI_ROOT
  ```

Simple MPI Program – Build and Run

- Compile the test program using the provided wrapper:
- `$ mpiicc ./test.c –o test.x`
- Note that you can see exactly what the compiler wrapper will use by typing:
- `$ mpiicc –show`
- Run the program on a single node. How many cores are available?
- `$ mpirun ./test.x`
- Run the program on at least two nodes, and set **`I_MPI_DEBUG=4`** to check on which node and core each MPI rank run.

Simple MPI Program – Review Questions

- How do compiler scripts match to languages, e.g which compile script is used for Fortran and which one for C?
- What is the difference between mpicc and mpiicc?
- Does mpiifort/mpiicc always take the ifort/icc compiler contained in the same package? Can you force mpiifort to take a different version?
- What is the strategy Intel MPI Library uses to fill up nodes? What is the default process pinning when we have less ranks than physical core?
- How would tasks be distributed if we had 8 physical cores per node, 2 nodes, and we only used 12 MPI ranks total?

Hands-On

ITAC Trace of Simple MPI Program

Simple MPI Trace - Setup

- Intel MPI provides a simple example in C, C++, Fortran, fortran90 and Java under the directory <impi_install_dir>/test/
- Copy the file to your working directory :

-
  ```
  $ cp /opt/intel/impi/<version>/test/test.c .
  ```
- Ensure your environment is set to use Intel MPI and ITAC:

-
  ```
  $ source /opt/intel/impi/<version>/bin64/mpivars.sh
  ```

-
  ```
  $ source /opt/intel/itac/<version>/bin/itacvars.sh
  ```
- You can check that the environment is set properly by checking if the variables I_MPI_ROOT and VT_ROOT are set.

Simple MPI Trace – Build and Run (I)

- Compile the test program using the provided wrapper:

-
  ```
  $ mpiicc ./test.c -o test_1.x
  ```
- Run the program with a small number of tasks (2 or 4) and collect the trace:

-
  ```
  $ mpirun -np 2 -trace ./test_1.x
  ```
-

Simple MPI Trace – Build and Run (II)

- Copy the executable to change its name:
-
```
$ cp ./test_1.x ./test_2.x
```
- Set **`LD_PRELOAD`** to the **`libVT.so`** location and run with a different number of tasks but without the –trace flag:
-
```
$ export LD_PRELOAD=$VT_SLIB_DIR/libVT.so
```
-
```
$ mpirun -np 4 ./test_2.x
```
-

Simple MPI Trace – Build and Run (II)

- Compile the test program using the provided wrapper and the –trace flag:

```
$ mpiicc –trace ./test.c –o test_3.x
```

- This time, make sure we collect to a single trace file:

```
$ export VT_LOGFILE_FORMAT=STFSINGLE
```

- Run the program again:

```
$ mpirun –np 2 ./test_3.x
```

Simple MPI Trace – Visualize Traces

- There are several ways of viewing your results using the ITAC GUI
  - From the system that collected the data – using an X window
  - Copying the trace files to a local Linux* or Windows system
- If you have an X-Window or VNC connection running:
-

```
$ traceanalyzer test_3.x.stf
```

- If you moved the file to a local system just doble-click the stf file
- The Summary opens first – what is the most expensive MPI call?
- Proceed to the next window, which rank uses the most MPI time? [Hint: Load Balance tab, look at “Children of all processes”]
- Make sure to check out the “Event Timeline” within the “Charts” menu

Hands-On

Produce APS Report with Simple MPI Program

Simple MPI APS report - Setup

- Intel MPI provides a simple example in C, C++, Fortran, fortran90 and Java under the directory <impi_install_dir>/test/
- Copy the file to your working directory :
-
```
$ cp /opt/intel/impi/<version>/test/test.c .
```
- Ensure your environment is set to use Intel MPI and APS:
-
```
$ source /opt/intel/impi/<version>/bin64/mpivars.sh
```
-
```
$ source /opt/intel/performance_snapshots/apsvars.sh
```
- You can check that the environment is set properly by checking if the variable I_MPI_ROOT.

Simple MPI APS report - Build and Run

- Compile the test program using the provided wrapper:

-
  ```
  $ mpiicc ./test.c –o test.x
  ```
- Run the program with a small number of tasks (2 or 4) and using APS:

-
  ```
  $ mpirun –np 4 aps ./test.x
  ```
- Notice that a new directory named **`aps_report_<yyymmdd>`** has been generated
- Notice also that instructions are given regarding how to generate a full report
-

Simple MPI APS report - Build and Run

- Generate a text summary report from the command line:
- `$ aps-report -s ./aps_results_<yyymmdd>`
- Generate an html report following the instructions provided:
- `$ aps --report=<working_dir>/aps_result_<yyymmdd>`
- This will generate the text summary again by default
- Open the html report with any browser and check that all the expected sections are included.

Hands-On

Simple MPI VTune Example

Simple MPI VTune™ Example - Setup

- Obtain the example tarball from the website:
- **`https://mantevo.org/downloads/miniFE_ref_2.0.html`**
- Ensure your environment is set to use Intel MPI and VTune™ Profiler:

```
$ source /opt/intel/impi/<version>/intel64/bin/mpivars.sh
$ source /opt/intel/vtune/amplxe-vars.sh
```

- You can check that the environment is set properly by checking if the variables **`I_MPI_ROOT`** and **`VTUNE_PROFILER_2020_DIR`** are set.

Simple MPI VTune™ Example - Build

- Untar the example:
- `$ tar xzvf sources/miniFE-2.0_ref.tgz`
- Change to the new directory:
- `$ cd miniFE-2.0_ref/src`
- Build the executable:
- `$ make CC=mpiicc CXX=mpiicpc`
- Copy the generated executable **`miniFE.x`** to the top-level directory:
- `$ cp ./miniFE.x ../`

Simple MPI VTune™ Example - Run

- Now change directory to the top-level directory and perform an hpc-performance collection:

```
▪   $ cd ../
▪   $ mpirun -np 4 amplxe-cl -collect hpc-performance \
▪       -r hpc -- ./miniFE.x nx=100
```

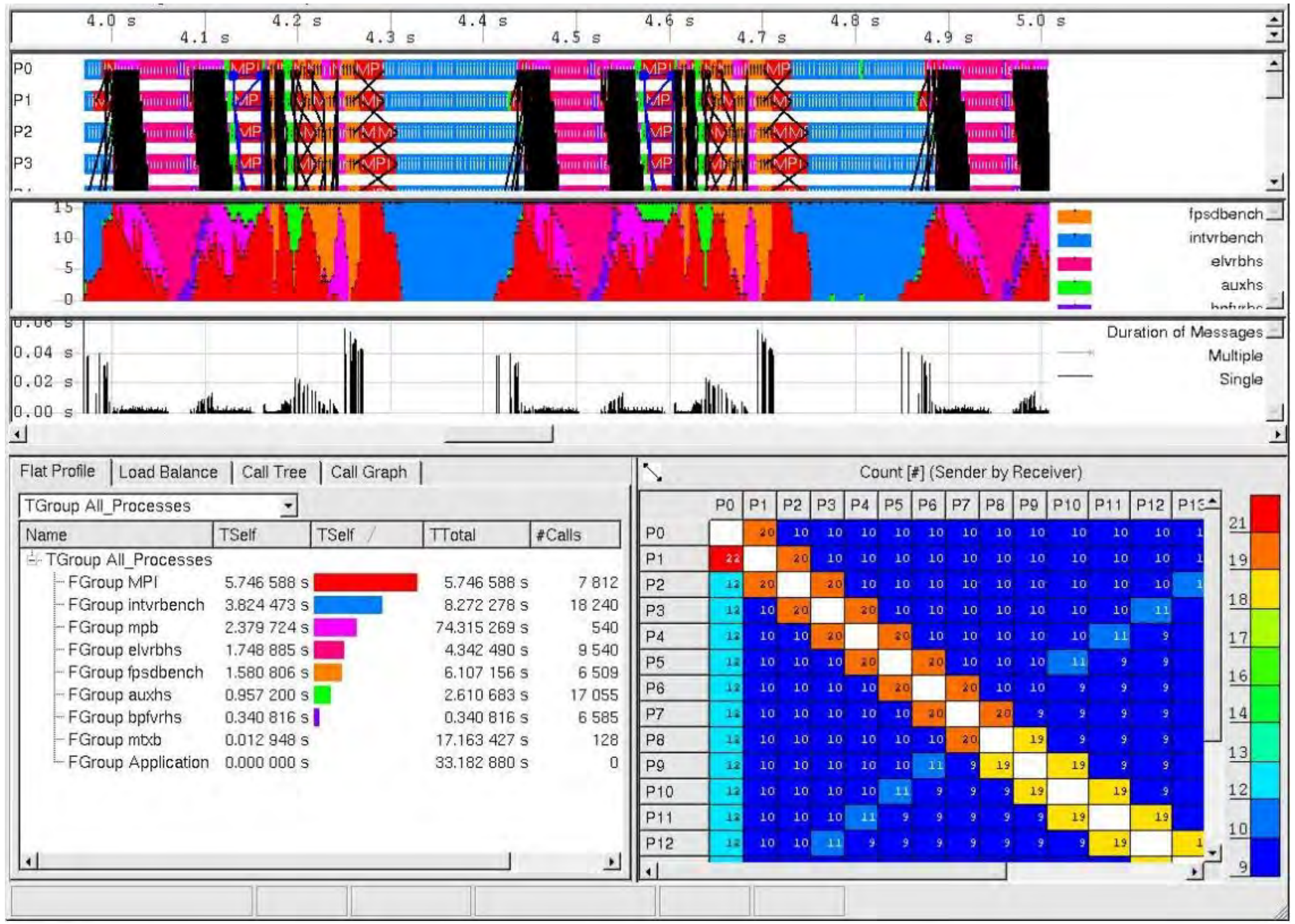
- Open newly generated results in the GUI:

```
▪   $ amplxe-gui ./hpc.<hostname>
```

- What is the MPI imbalance for this job?
- Can you identify the slowest MPI rank? [Hint: Bottom-up, Grouping by Process/Function/Threads/Call Stack]

Additional ITAC Capabilities

Zooming



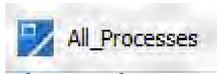
Name	TSelf	TSelf	TTotal	#Calls
TGroup All_Processes				
FGroup MPI	5.746 588 s	🟥	5.746 588 s	7 812
FGroup intvrbench	3.824 473 s	🟦	8.272 278 s	18 240
FGroup mpb	2.379 724 s	🟪	74.315 269 s	540
FGroup elvrbhs	1.748 885 s	🟥	4.342 490 s	9 540
FGroup fpsdbench	1.580 806 s	🟧	6.107 156 s	6 509
FGroup auxhs	0.957 200 s	🟩	2.610 683 s	17 055
FGroup bpfvrhs	0.340 816 s	🟪	0.340 816 s	6 585
FGroup mtxb	0.012 948 s		17.163 427 s	128
FGroup Application	0.000 000 s		33.182 880 s	0

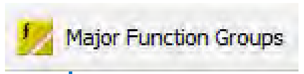
Count [#] (Sender by Receiver)

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
P0		20	10	10	10	10	10	10	10	10	10	10	10	1
P1	22		20	10	10	10	10	10	10	10	10	10	10	1
P2	12	20		20	10	10	10	10	10	10	10	10	10	1
P3	12	10	20		20	10	10	10	10	10	10	10	11	
P4	12	10	10	20		20	10	10	10	10	10	11	9	
P5	12	10	10	10	20		20	10	10	10	11	9	9	
P6	12	10	10	10	10	20		20	10	10	9	9	9	
P7	12	10	10	10	10	10	20		20	9	9	9	9	
P8	12	10	10	10	10	10	10	20		19	9	9	9	
P9	12	10	10	10	10	10	11	9	19		19	9	9	
P10	12	10	10	10	10	11	9	9	9	19		19	9	
P11	12	10	10	10	11	9	9	9	9	9	19		19	
P12	12	10	10	11	9	9	9	9	9	9	9	19		1

Grouping and Aggregation

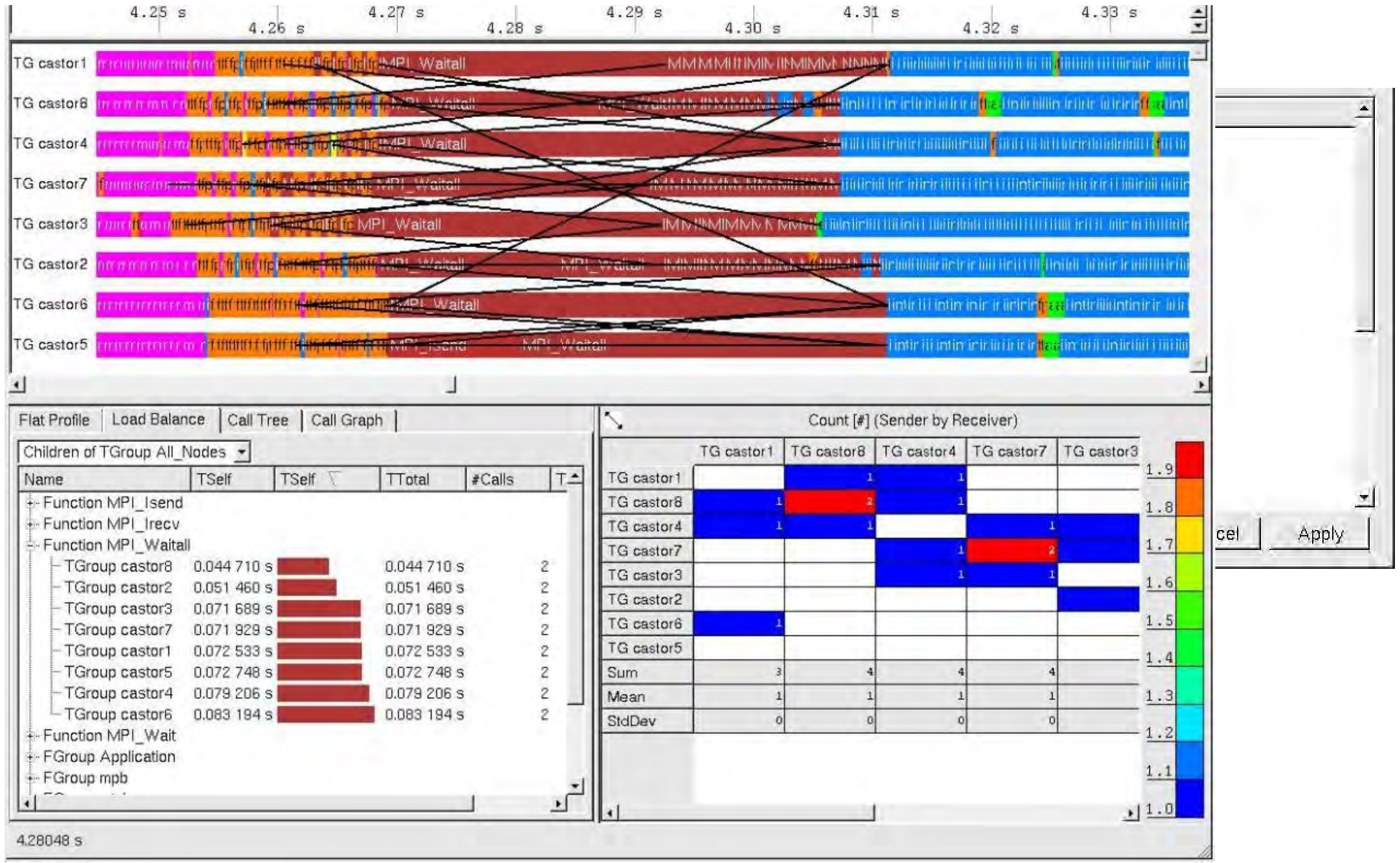
- Allow analysis on different levels of detail by aggregating data upon group-definitions
- Functions and threads can be grouped hierarchically
  - Process Groups and Function Groups





- Arbitrary nesting is supported
  - Functions/threads on the same level as groups
  - User can define his/her own groups
- Aggregation is part of View-definition
  - All charts in a View adapt to requested grouping
  - All charts support aggregation

Aggregation Example



Name	TSelf	TSelf	TTotal	#Calls
Function MPI_Isend				
Function MPI_Irecv				
Function MPI_Waitall				
TGroup castor8	0.044 710 s		0.044 710 s	2
TGroup castor2	0.051 460 s		0.051 460 s	2
TGroup castor3	0.071 689 s		0.071 689 s	2
TGroup castor7	0.071 929 s		0.071 929 s	2
TGroup castor1	0.072 533 s		0.072 533 s	2
TGroup castor5	0.072 748 s		0.072 748 s	2
TGroup castor4	0.079 206 s		0.079 206 s	2
TGroup castor6	0.083 194 s		0.083 194 s	2
Function MPI_Wait				
FGroup Application				
FGroup mpb				

Count [#] (Sender by Receiver)

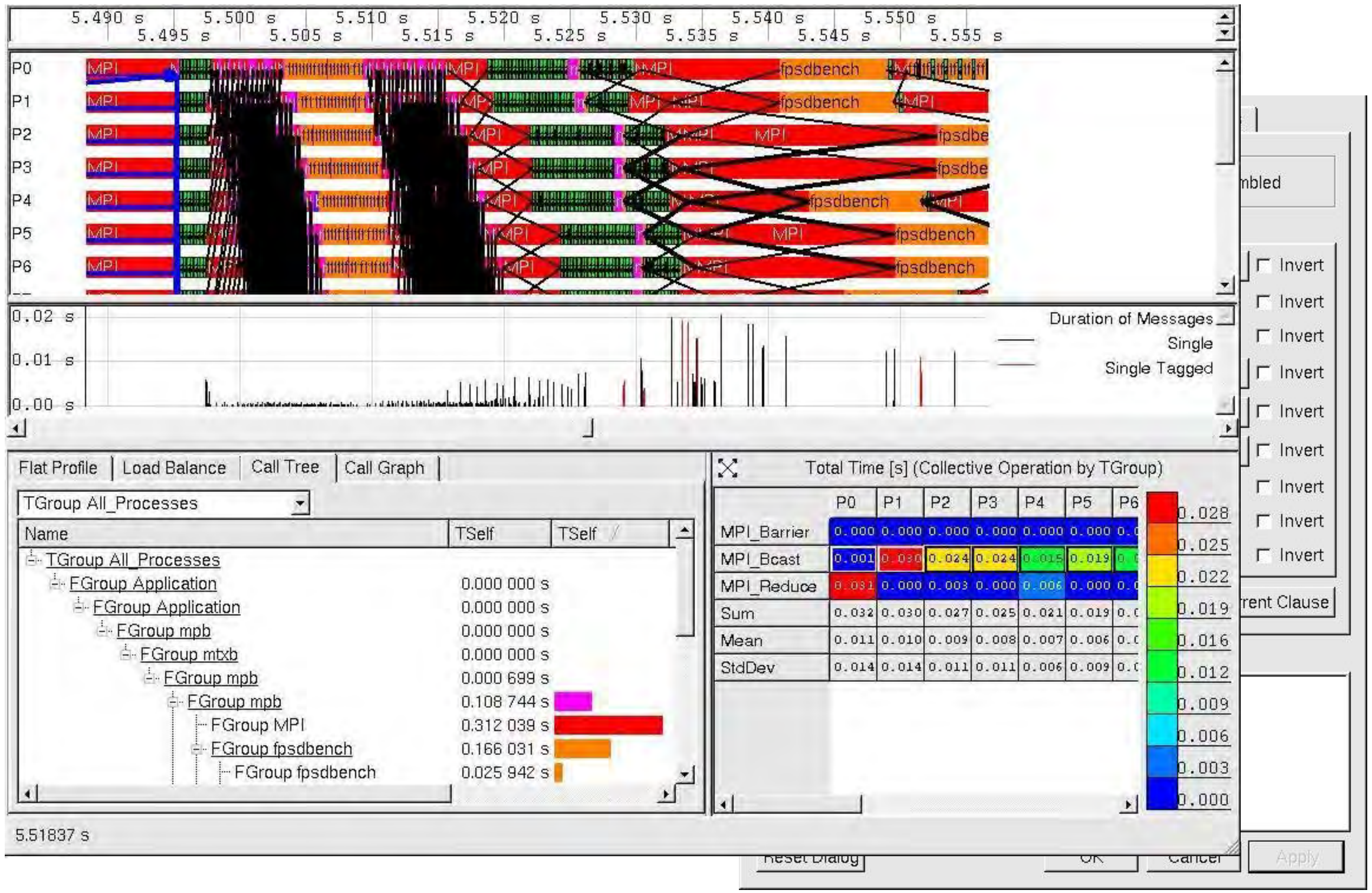
	TG castor1	TG castor8	TG castor4	TG castor7	TG castor3
TG castor1		1	1		
TG castor8	1	2	1		
TG castor4	1	1		1	🟦
TG castor7			1	2	🟦
TG castor3			1	1	
TG castor2					🟦
TG castor6	1				
TG castor5					
Sum	3	4	4	4	
Mean	1	1	1	1	
StdDev	0	0	0	0	

1.9	🟥
1.8	🟧
1.7	🟨
1.6	🟩
1.5	🟩
1.4	🟩
1.3	🟩
1.2	🟦
1.1	🟦
1.0	🟦

Tagging and Filtering

- Help concentrating on relevant parts
- Avoid getting lost in huge amounts of trace data
- Define a set of interesting data
  - E.g. all occurrences of function x
  - E.g. all messages with tag y on communicator z
- Combine several filters:
  Intersection, Union, Complement
- Apply it
  - Tagging: Highlight messages
  - Filtering: Suppress all non-matching events

Tagging Example



TGroup All_Processes

Name	TSelf	TSelf
TGroup All_Processes		
FGroup Application	0.000 000 s	
FGroup Application	0.000 000 s	
FGroup mpb	0.000 000 s	
FGroup mtxb	0.000 000 s	
FGroup mpb	0.000 699 s	
FGroup mpb	0.108 744 s	
FGroup MPI	0.312 039 s	
FGroup fpsdbench	0.166 031 s	
FGroup fpsdbench	0.025 942 s	

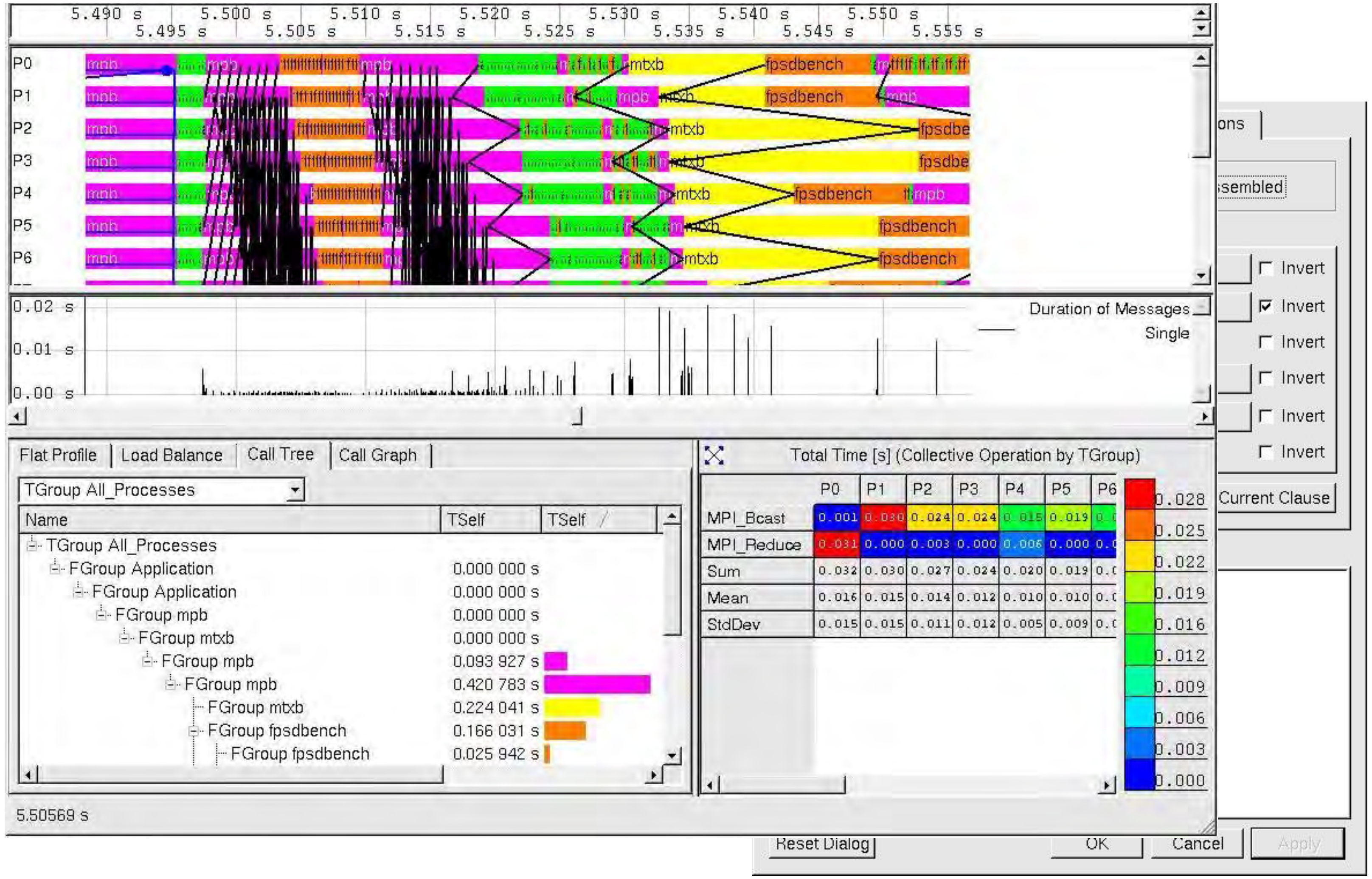
Total Time [s] (Collective Operation by TGroup)

	P0	P1	P2	P3	P4	P5	P6
MPI_Barrier	0.000	0.000	0.000	0.000	0.000	0.000	0.0
MPI_Bcast	0.001	0.030	0.024	0.024	0.015	0.019	0.0
MPI_Reduce	0.031	0.000	0.003	0.000	0.006	0.000	0.0
Sum	0.032	0.030	0.027	0.025	0.021	0.019	0.0
Mean	0.011	0.010	0.009	0.008	0.007	0.006	0.0
StdDev	0.014	0.014	0.011	0.011	0.006	0.009	0.0

0.028
0.025
0.022
0.019
0.016
0.012
0.009
0.006
0.003
0.000

5.51837 s

Filtering Example



Name	TSelf	TSelf
TGroup All_Processes		
FGroup Application	0.000 000 s	
FGroup Application	0.000 000 s	
FGroup mpb	0.000 000 s	
FGroup mtxb	0.000 000 s	
FGroup mpb	0.093 927 s	
FGroup mpb	0.420 783 s	
FGroup mtxb	0.224 041 s	
FGroup fpsdbench	0.166 031 s	
FGroup fpsdbench	0.025 942 s	

Total Time [s] (Collective Operation by TGroup)

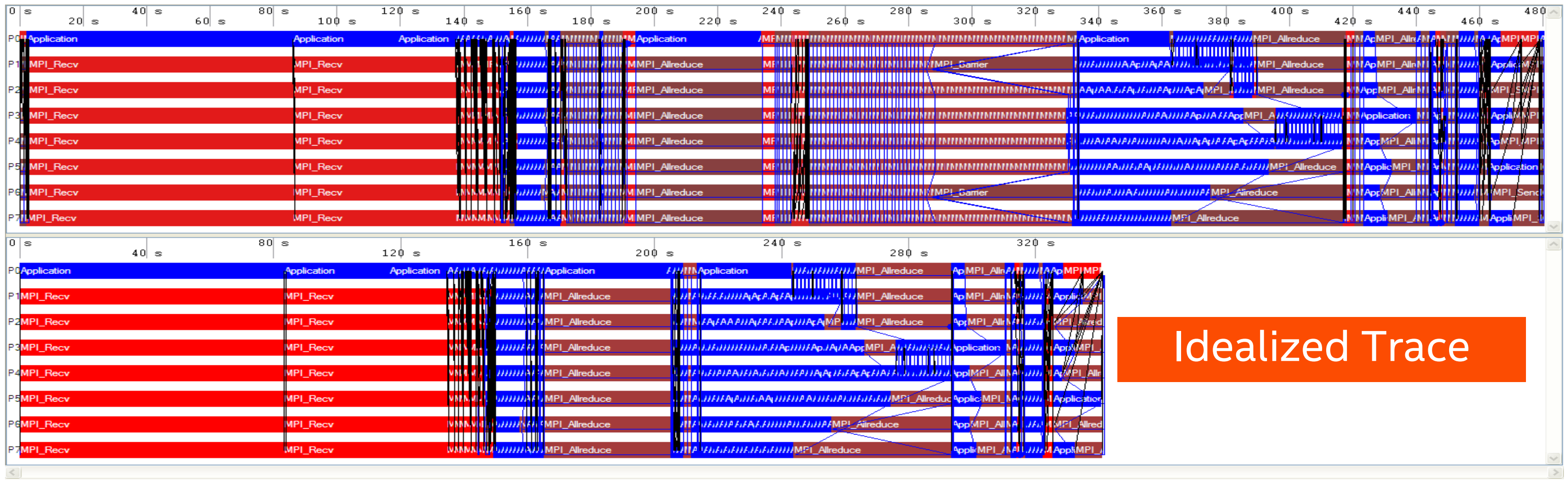
	P0	P1	P2	P3	P4	P5	P6
MPI_Bcast	0.001	0.030	0.024	0.024	0.015	0.019	0.0
MPI_Reduce	0.031	0.000	0.003	0.000	0.006	0.000	0.0
Sum	0.032	0.030	0.027	0.024	0.020	0.019	0.0
Mean	0.016	0.015	0.014	0.012	0.010	0.010	0.0
StdDev	0.015	0.015	0.011	0.012	0.005	0.009	0.0

🟥	0.028
🟧	0.025
🟨	0.022
🟩	0.019
🟩	0.016
🟩	0.012
🟦	0.009
🟦	0.006
🟦	0.003
🟦	0.000

Ideal Interconnect Simulator (Idealizer)

- Helps to figure out application's imbalance simulating its behavior in the "ideal communication environment"

Actual trace

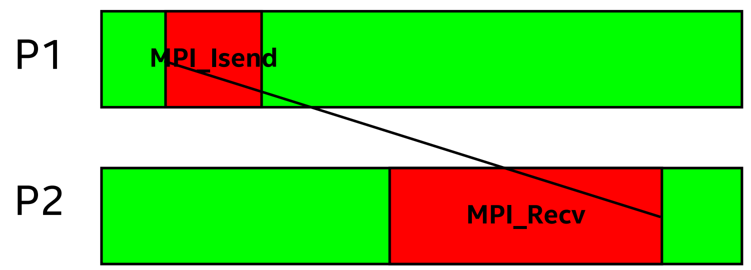


Idealized Trace

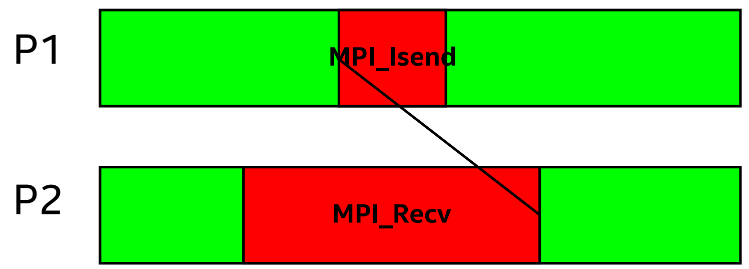
Easy way to identify application bottlenecks

Building Blocks: Elementary Messages

Early Send /
Late Receive

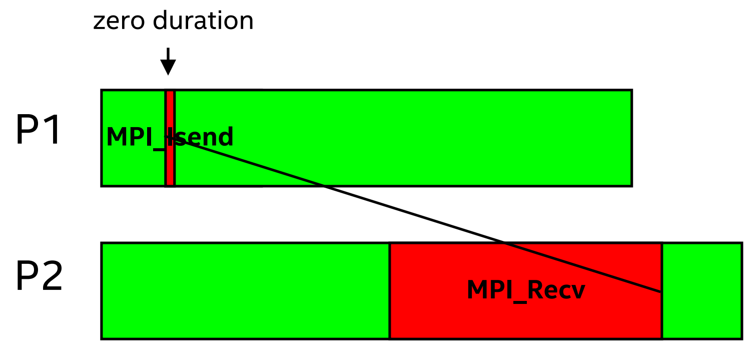


Late Send /
Early Receive

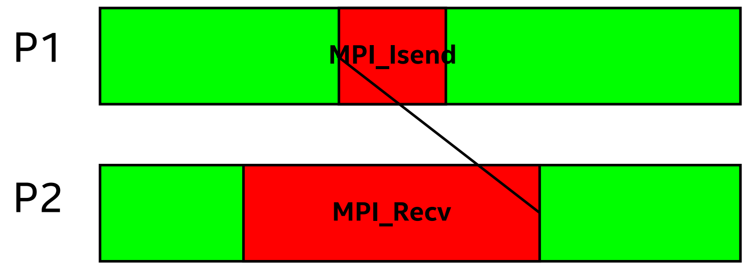


Building Blocks: Elementary Messages

Early Send / Late Receive

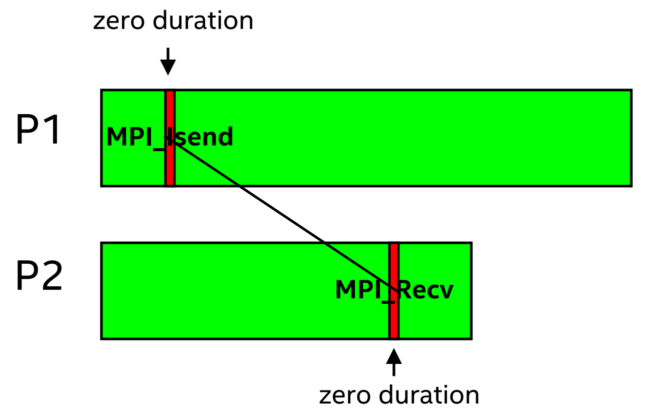


Late Send / Early Receive

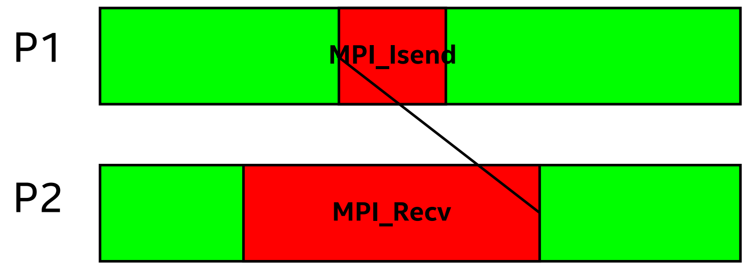


Building Blocks: Elementary Messages

Early Send / Late Receive

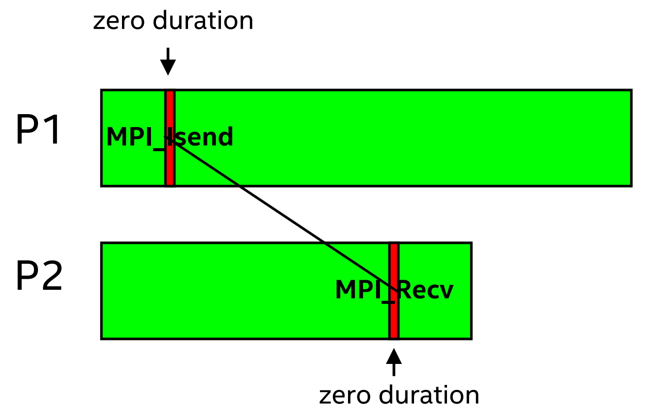


Late Send / Early Receive

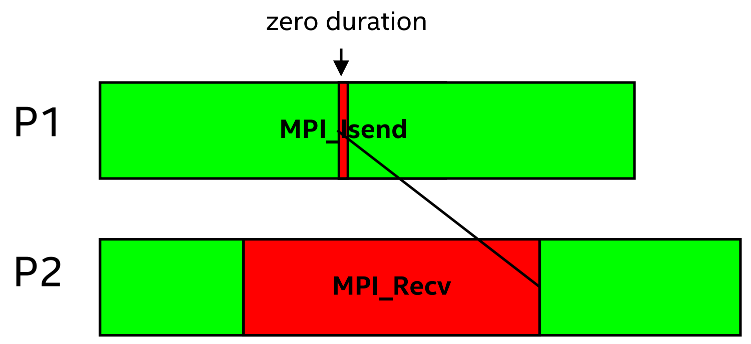


Building Blocks: Elementary Messages

Early Send /
Late Receive

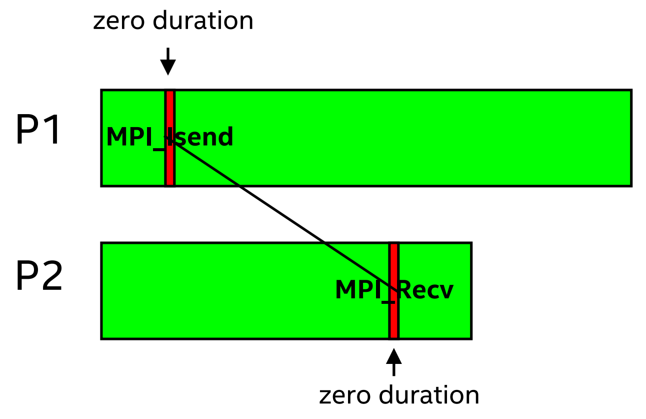


Late Send /
Early Receive

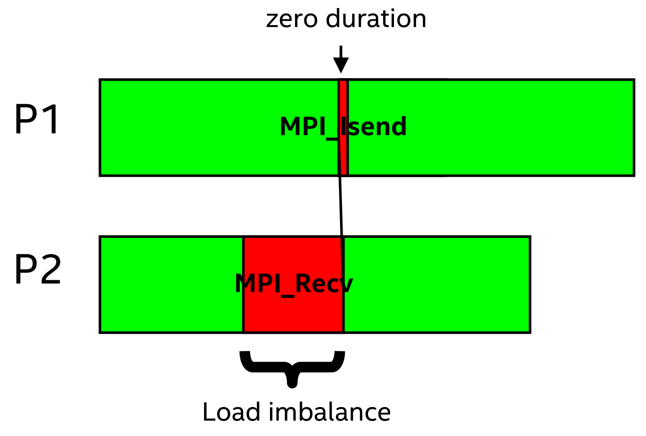


Building Blocks: Elementary Messages

Early Send / Late Receive



Late Send / Early Receive



Tuning Methods

- Library Tuning (algorithms, fabric parameters)
  - mpitune
  - MPI statistics
- Application Tuning (load balance, MPI/threaded/serial performance)
  - Intel® Trace Analyzer and Collector
  - Intel® VTune™ Profiler XE
  - Application Performance Snapshot

Library Tuning: mpitune

Use the automatic tuning facility to tune the Intel® MPI Library for your cluster or application (done once, may take a long time)

Modes (see mpitune –h for options)

- Cluster-wide tuning
- Application-specific tuning

```
mpitune …
```

```
mpitune –application \"mpirun –n 32 ./exe\" …
```

Creates options settings which are used with the –tune flag

```
mpirun –tune …
```

Lightweight Statistics

- Set I_MPI_STATS to a non-zero integer to gather MPI communication statistics (max. 10)
- Change scope with I_MPI_STATS_SCOPE
- Define output file with I_MPI_STATS_FILE
- Bin results using I_MPI_STATS_BUCKET

Lightweight Statistics

```
▪   $ export I_MPI_STATS=3
▪   $ export I_MPI_SCOPE=coll
▪   $ mpirun -np 8 ./exe
▪
▪   ...

▪   Communication Activity by actual args
▪   Collectives
▪   Operation        Context  Algo  Comm size    Message size      Calls  Cost(%)
▪   --------------------------------------------------------------------------------
▪   Allreduce
▪   1                    248     1          8              64          1     0.27
▪   Barrier
▪   1                      0     2          8               0          1     0.00
▪   Bcast
▪   1                      0     8          8              32          1     0.00
▪   2                      0     8          8              72          1     0.00
▪   3                      0     8          8              56          1     0.01
▪   Reduce
▪   1                    248     1          8               8          1     0.00
▪   2                    248     1          8              72          1     0.32
▪   ================================================================================
```

intel®