
Intel software tools

Intel Parallel Studio

In the following slides we will give an overview of the tools available from the Intel Parallel Studio.

You can use these tools on the Cineca HPC machines using

```
[faffinit@r000u18l02 faffinit]$ module load intel/pe-xe-2016--binary
[faffinit@r000u18l02 faffinit]$ source $INTEL_HOME/parallel_studio_xe_2016.3.067/bin/psxevars.sh
mpsvars.sh: Warning: Hardware events collection is disabled by default. To enable it, run mpsvars.sh with --vtune (recommended) or
--papi option.
Copyright (C) 2009-2016 Intel Corporation. All rights reserved.
Intel(R) Inspector XE 2016 (build 460803)
Copyright (C) 2009-2016 Intel Corporation. All rights reserved.
Intel(R) VTune(TM) Amplifier XE 2016 (build 463186)
Copyright (C) 2009-2016 Intel Corporation. All rights reserved.
Intel(R) Advisor XE 2016 (build 463413)
[faffinit@r000u18l02 faffinit]$ █
```

Preliminarily...

Load compiler and Intel MPI:

```
module load intel  
module load intelmpi  
(if needed) module load mkl
```

Poisson

Check the Makefile:

```
[faffinit@r000u18l02 Poisson-C-0.6.1.4]$ cat Makefile
VERSION=0.6.1.4
#DEBUG=-g -debug inline-debug-info
DEBUG=-g
OMP=-qopenmp
#SIMD=-DSIMD
#ISA=-xAVX
#ISA=-xCORE-AVX2
#ISA=-xHOST
#ISA=-xMIC-AVX512
MPI=-DUSE_MPI
#REPORT=-qopt-report5
ADD_FLAGS=
DEFINES=-DDOUBLE -DUSE_MPI
```

Level-0 MPI profiling

```
export I_MPI_STATS=ipm  
mpirun -np 36 ./poisson.x
```

check stats.ipm

```
export I_MPI_STATS=1  
mpirun -np 36 ./poisson.x
```

check stats.txt

I_MPI_DEBUG

```
export I_MPI_DEBUG=4
```

```
[0] MPI startup(): Rank   Pid   Node name  Pin cpu
[0] MPI startup(): 0     11530  r045c01s03 0
[0] MPI startup(): 1     11531  r045c01s03 1
[0] MPI startup(): 2     11532  r045c01s03 2
[0] MPI startup(): 3     11533  r045c01s03 3
[0] MPI startup(): 4     11534  r045c01s03 4
[0] MPI startup(): 5     11535  r045c01s03 5
[0] MPI startup(): 6     11536  r045c01s03 6
[0] MPI startup(): 7     11537  r045c01s03 7
[0] MPI startup(): 8     11538  r045c01s03 8
[0] MPI startup(): 9     11539  r045c01s03 9
[0] MPI startup(): 10    11540  r045c01s03 10
[0] MPI startup(): 11    11541  r045c01s03 11
[0] MPI startup(): 12    11542  r045c01s03 12
[0] MPI startup(): 13    11543  r045c01s03 13
[0] MPI startup(): 14    11544  r045c01s03 14
[0] MPI startup(): 15    11545  r045c01s03 15
[0] MPI startup(): 16    11546  r045c01s03 16
[0] MPI startup(): 17    11547  r045c01s03 17
[0] MPI startup(): 18    11548  r045c01s03 18
[0] MPI startup(): 19    11549  r045c01s03 19
[0] MPI startup(): 20    11550  r045c01s03 20
[0] MPI startup(): 21    11551  r045c01s03 21
[0] MPI startup(): 22    11552  r045c01s03 22
[0] MPI startup(): 23    11553  r045c01s03 23
[0] MPI startup(): 24    11554  r045c01s03 24
```

Analysis of the Poisson code

Objectives:

- demonstrate features of Intel Trace Analyzer and Collector (ITAC) and VTune Amplifier
- find the root cause for suboptimal scaling
- show ways of tuning where the tools indicate suboptimal performance

Simple scaling analysis

Just give a look to the code running with different number of tasks/threads (up to 2 nodes, for example)

Speedup S is defined as $S[p]=T[1]/T[p]$

Efficiency E is defined as $E[p]=S[p]/p$

In the ideal case: $S[p]=p$ and $E[p]=1$

Before running..

Don't forget:

- to load the intel modules (and intelmpi)
- to source the psxe.vars
- check the resources you're running on with cpubinfo

cpuinfo

```
[faffinit@r000u17l01 ~]$ cpuinfo
Intel(R) processor family information utility, Version 5.1.3 Build 20160120 (build id: 14053)
Copyright (C) 2005-2016 Intel Corporation. All rights reserved.
```

```
==== Processor composition ====
Processor name   : Intel(R) Xeon(R)  E5-2695 v3
Packages(sockets) : 2
Cores           : 28
Processors(CPUs) : 56
Cores per package : 14
Threads per core  : 2
```

```
==== Processor identification ====
Processor      Thread Id.   Core Id.   Package Id.
0              0            0           0
1              0            1           0
2              0            2           0
3              0            3           0
4              0            4           0
5              0            5           0
6              0            6           0
7              0            8           0
8              0            9           0
9              0           10          0
10             0           11          0
11             0           12          0
12             0           13          0
13             0           14          0
14             0            0           1
```

Poisson solver

It's a standard problem (e.g. heat equation).

We will investigate a square 3600x3600 computational grid. It's on the edge of the bandwidth limitations.

Grid points will be distributed to MPI ranks to a 2D process grid. The cartesian process grid is a feature of the Poisson solver. Other programs can have different data distribution.

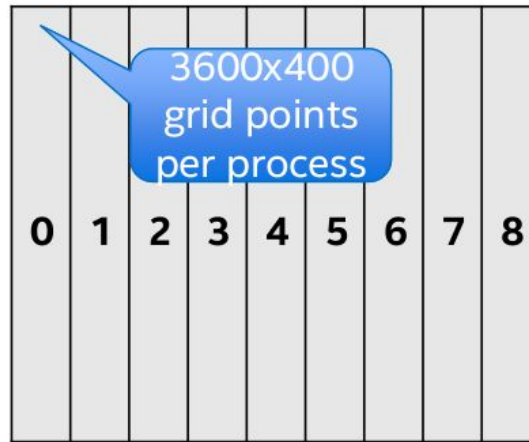
Optimal grid?

Which is the optimal choice for the grid partitioning?



1200x1200
grid points
per process

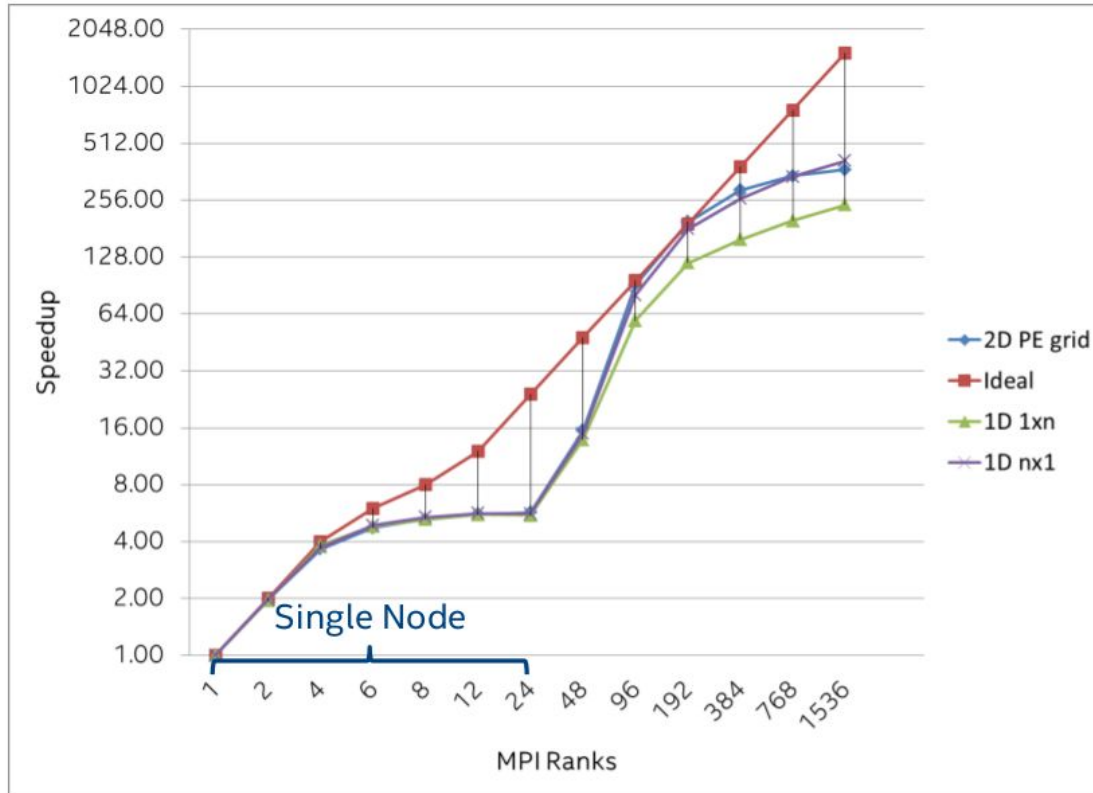
3x3 2D process grid



3600x400
grid points
per process

1x9 1D process grid

Grid and performances



Measure MPI times with ITAC

```
[faffinit@r000u17l01 ~]$ qsub -I -A cin_priorit -l select=1:ncpus=36:mpiprocs=36 -l walltime=15:00
qsub: waiting for job 63431.r000u17l01 to start
qsub: job 63431.r000u17l01 ready
```

```
[faffinit@r040c03s01 ~]$ module load intel intelmpi
[faffinit@r040c03s01 ~]$ source /cineca/prod/opt/compilers/intel/pe-xe-2016/binary/itac/9.1.2.024/bin/itacvars.sh
mpsvars.sh: Warning: Hardware events collection is disabled by default. To enable it, run mpsvars.sh with --vtune (
recommended) or --papi option.
[faffinit@r040c03s01 ~]$ cd /marconi_scratch/userinternal/faffinit/Poisson-C-0.6.1.4
[faffinit@r040c03s01 Poisson-C-0.6.1.4]$ mpirun -trace n 16 ./poisson.x
```

```
Residuum after 751 iterations = 0.004992
```

```
Compute time = 1.543324 [sec]
```

```
Perf          = 3.406284 [GFlops]
```

```
[0] Intel(R) Trace Collector INFO: Writing tracefile poisson.x.stf in /marconi_scratch/userinternal/faffinit/Poisso
n-C-0.6.1.4
```

Comments

- finding the right path in your case can be troublesome: if you can, use the which or the tab autocompletion...
- use `OMP_NUM_THREADS=1`
- by default, ITAC creates a .stf file for each process. It results in a high number of files. To overcome this problem use
`export VT_LOGFILE_FORMAT=STFSINGLE`
- after the data collection is completed, you will need a graphical interface to visualize with ITAC (-> use RCM)

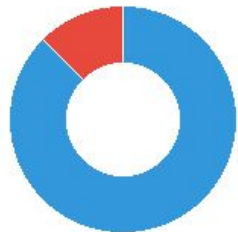
Summary: poisson.x.single.stf

Total time: 18.8 sec. Resources: 16 processes, 1 node.

[Continue >](#)

Ratio

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



- Serial Code - 16.4 sec 87.5 %
- MPI calls - 2.34 sec 12.4 %

Top MPI functions

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

MPI_Allreduce	■	0.722 sec (3.85 %)
MPI_Waitall	■	0.652 sec (3.47 %)
MPI_Isend	■	0.484 sec (2.58 %)
MPI_Irecv	■	0.43 sec (2.29 %)
MPI_Barrier	■	0.0331 sec (0.176 %)

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

To optimize node-level performance use:

- Intel® VTune™ Amplifier XE** for:
 - algorithmic level tuning with hotspots and threading efficiency analysis;
 - microarchitecture level tuning with general exploration and bandwidth analysis;
- Intel® Advisor** for:



1: /galileo/home/userinternal/faffinit/poisson.x.single.stf

View Charts Navigate Advanced Layout

0.000 000 - 1.173 042 : 1.173 042 Seconds All_Processes Major Function Groups

Flat Profile Load Balance Call Tree Call Graph

All_Processes

Name	TSelf	TSelf	TTotals	#Calls	TSelf /C
▲ All_Processes					
└ Group Application	16.4122 s		18.7541 s	16	1.0
└ Group MPI	2.34182 s		2.34182 s	228848	10.23:

Performance Issue	Duration (%)	Duration
Wait at Barrier	0.18%	34.038e-3 s
Show advanced...		

Description Affected Processes Source Locations (Root Cal...)

Select performance issue to see details.



Flat Profile Load Balance Call Tree Call Graph

All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call	TTotal /Call	#Procs	TSelf /Proc	TTotal /Proc
▲ All Processes									
Group Application	16.4122 s		18.7541 s	16	1.02576 s	1.17213 s	16	1.02576 s	1.17213 s
MPI_Comm_size	226e-6 s		226e-6 s	16	14.125e-6 s	14.125e-6 s	16	14.125e-6 s	14.125e-6 s
MPI_Comm_rank	264e-6 s		264e-6 s	16	16.5e-6 s	16.5e-6 s	16	16.5e-6 s	16.5e-6 s
MPI_Finalize	19.994e-3 s		19.994e-3 s	16	1.24962e-3 s	1.24962e-3 s	16	1.24962e-3 s	1.24962e-3 s
MPI_Isend	483.588e-3 s		483.588e-3 s	84280	5.73787e-6 s	5.73787e-6 s	16	30.2242e-3 s	30.2242e-3 s
MPI_Irecv	430.316e-3 s		430.316e-3 s	84280	5.10579e-6 s	5.10579e-6 s	16	26.8947e-3 s	26.8947e-3 s
MPI_Wtime	409e-6 s		409e-6 s	32	12.7812e-6 s	12.7812e-6 s	16	25.5625e-6 s	25.5625e-6 s
MPI_Waitall	651.898e-3 s		651.898e-3 s	48160	13.5361e-6 s	13.5361e-6 s	16	40.7436e-3 s	40.7436e-3 s
MPI_Barrier	33.078e-3 s		33.078e-3 s	16	2.06737e-3 s	2.06737e-3 s	16	2.06737e-3 s	2.06737e-3 s
MPI_Allreduce	722.051e-3 s		722.051e-3 s	12032	60.0109e-6 s	60.0109e-6 s	16	45.1282e-3 s	45.1282e-3 s

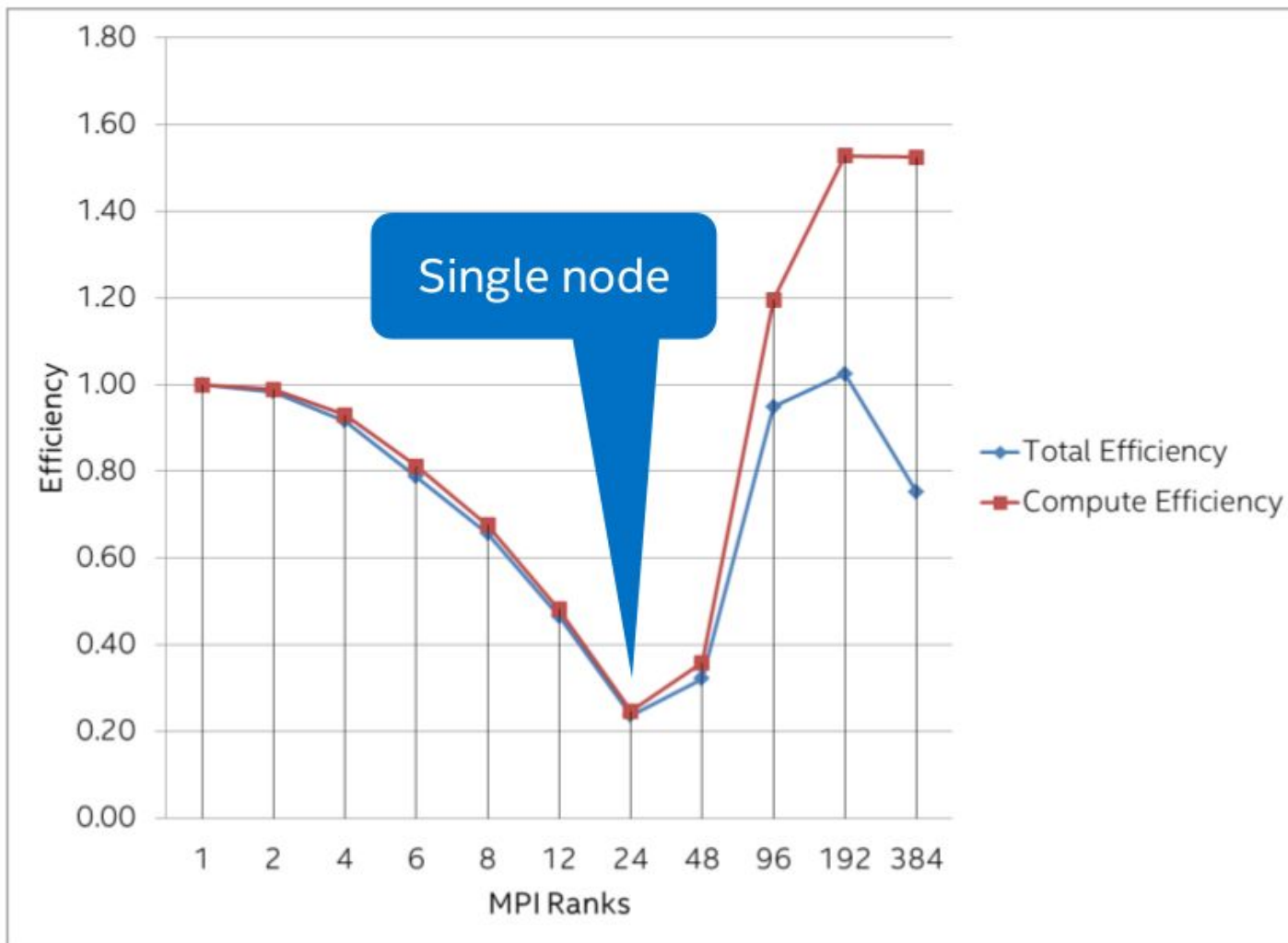
MPI times

The total wall clock run time $T[p]$, is given by the time spent in MPI plus time spent in computation:

$$T[p]=T_{\text{comp}}[p]+T_{\text{mpi}}[p]$$

Speedup and efficiency can be calculated for the compute time separately:

$$S_{\text{comp}}[p]=T_{\text{comp}}[1]/T_{\text{comp}}[p]=T[1]/T_{\text{comp}}[p]$$



Source of the MPI time

ITAC shows timing of all MPI routines used by an application

The timing of MPI routines may be due to network transfer times caused by interconnect bandwidth limitations (latencies)

The other possibility are waiting times caused by the algorithm: load imbalance or dependencies

A network model

- Latency L is defined as the transfer time for a 0 byte message
- Bandwidth BW is defined as the transfer rate for asymptotically large messages
- Message volume V is the data amount sent

The transfer time is:

$$T_{\text{trans}}[V] = L + (1/BW)*V$$

ITAC ideal network simulator

It is extremely complicated to simulate a realistic network!

An extreme case – the ideal network – may be simulated by setting all transfer times to 0. This would mean $L=0$ and $BW=\infty$ for the simple model

ITAC offers an ideal network simulation with transfer times set to zero. Compute times (non MPI) will stay the same

An existing real trace file is used as basis for the simulation

ITAC ideal network simulator

With a perfectly balanced algorithm the total MPI time will be vanishing in the ideal case

In most real cases the MPI time will just shrink but not vanish

The remaining part is due to waiting time e.g. when the receiver is starting to receive before the sender is ready to send

Start simulator with:

Advanced -> Idealization

1: /galileo/home/userinternal/faffinit/poisson.x.single.stf

View Charts Navigate Advanced Layout

0.00

Tagging...

All_Processes

MPI expanded in (Major Function Groups)

Filtering...

Reset Tagging/Filtering

Flat Profile

Load B

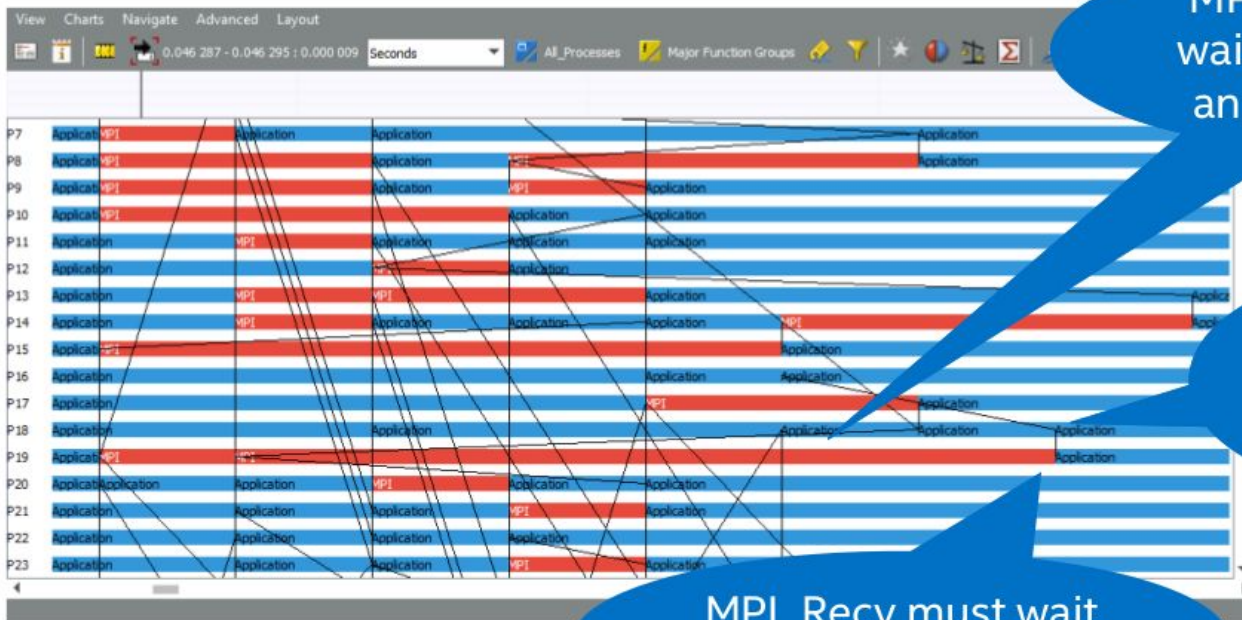
Command line for VTune Amplifier...

Command line for Advisor...

All_Processes

Name		#Calls	TSelf /Call	TTotal /Call	#Procs	TSelf /Proc	TTotal /Proc
All_Processes							
Group Application		16	1.02576 s	1.17213 s	16	1.02576 s	1.17213 s
MPI_Comm_size		16	14.125e-6 s	14.125e-6 s	16	14.125e-6 s	14.125e-6 s
MPI_Comm_rank		16	16.5e-6 s	16.5e-6 s	16	16.5e-6 s	16.5e-6 s
MPI_Finalize		16	1.24962e-3 s	1.24962e-3 s	16	1.24962e-3 s	1.24962e-3 s
MPI_Isend		84280	5.73787e-6 s	5.73787e-6 s	16	30.2242e-3 s	30.2242e-3 s
MPI_Irecv		84280	5.10579e-6 s	5.10579e-6 s	16	26.8947e-3 s	26.8947e-3 s
MPI_Wtime		32	12.7812e-6 s	12.7812e-6 s	16	25.5625e-6 s	25.5625e-6 s
MPI_Waitall		48160	13.5361e-6 s	13.5361e-6 s	16	40.7436e-3 s	40.7436e-3 s
MPI_Barrier		16	2.06737e-3 s	2.06737e-3 s	16	2.06737e-3 s	2.06737e-3 s
MPI_Allreduce		12032	60.0109e-6 s	60.0109e-6 s	16	45.1282e-3 s	45.1282e-3 s

Waiting time due to dependencies



MPI_Recv is pure waiting time inside an ideal trace file

MPI_Send time shrinks to 0

MPI_Recv must wait on MPI_Send call

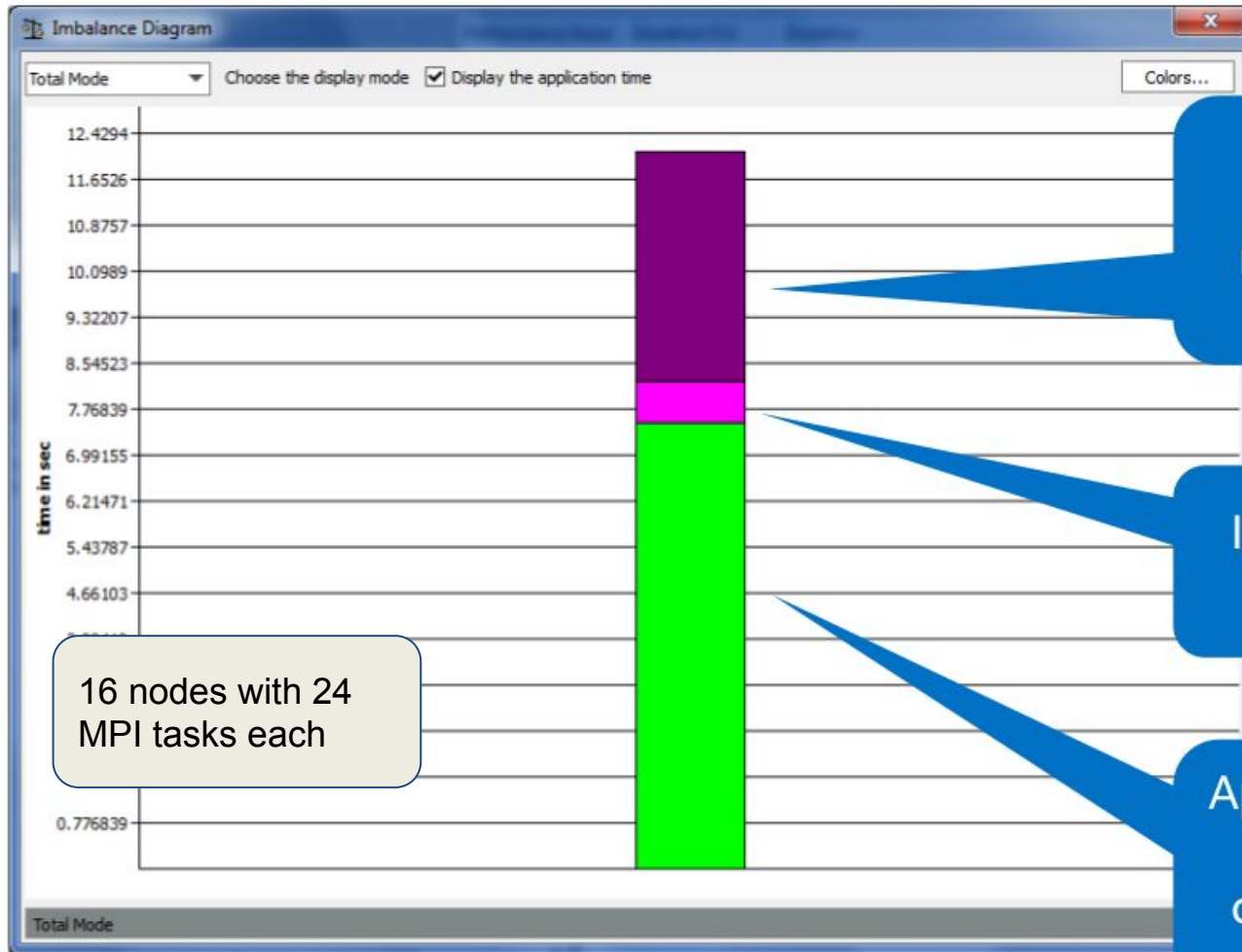
MPI time

The simulated MPI time for the ideal network may be regarded as the waiting time T_{wait} due to imbalance and dependencies:

$$T_{\text{mpi}} = T_{\text{transfer}} + T_{\text{wait}}$$

After generation of an ideal trace file the result can be displayed in the Imbalance Diagram:

Advanced Application -> Imbalance Diagram

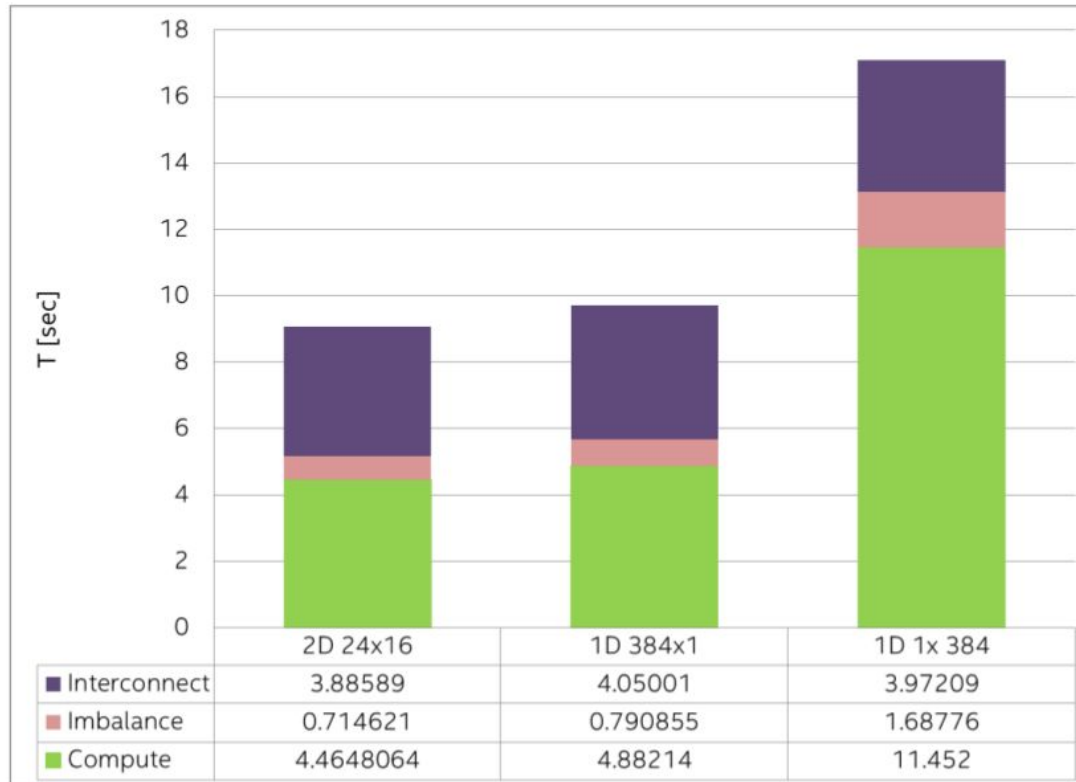


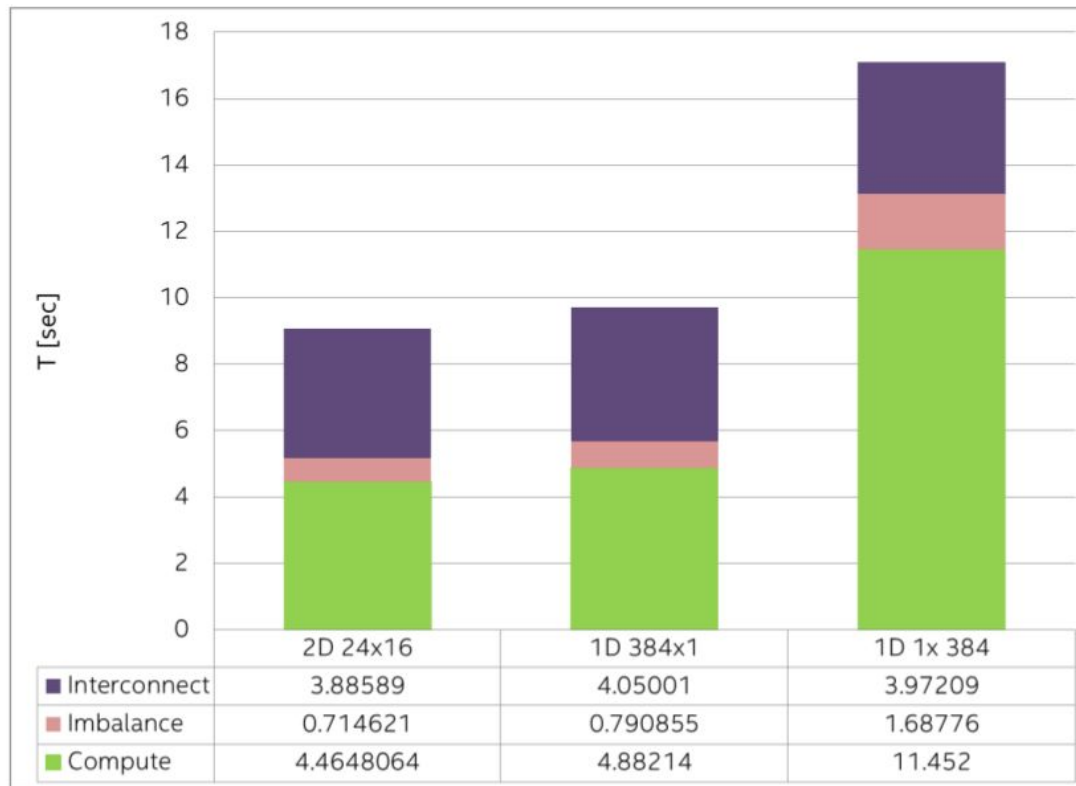
Imbalance diagram

The imbalance diagram displays the relation of transfer to wait time. Due to the result we can decide how to proceed with tuning:

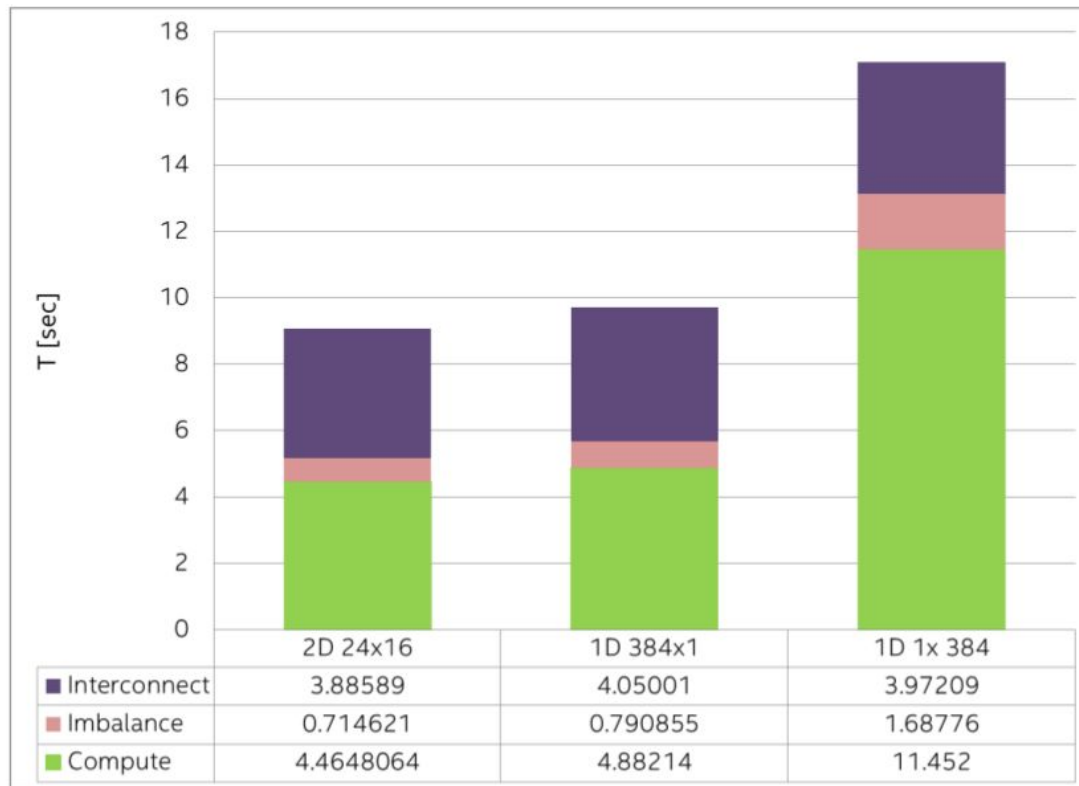
- Transfer time (Interconnect) dominates: the algorithm is balanced but we have to improve the network performance by e.g. different process placement or new network hardware
- Waiting time (Imbalance) dominates: the algorithm has to be revisited e.g. better load balancing. New network hardware or better process placement will not help!

Testing different grids

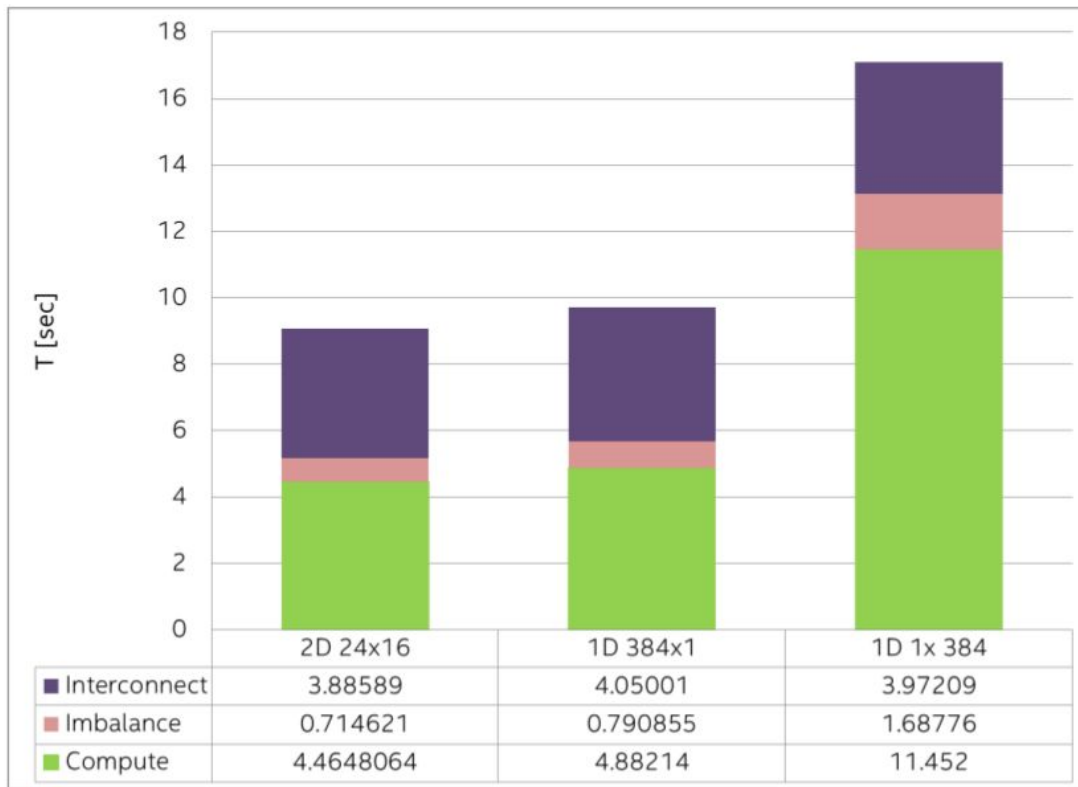




Compute time almost equal for 2D 24x16 and 1D 384x1 process grid. Row vectors are long enough. $3600/16 = 225$ for 24x16 process grid and 3600 grid points for 384x1 process grid



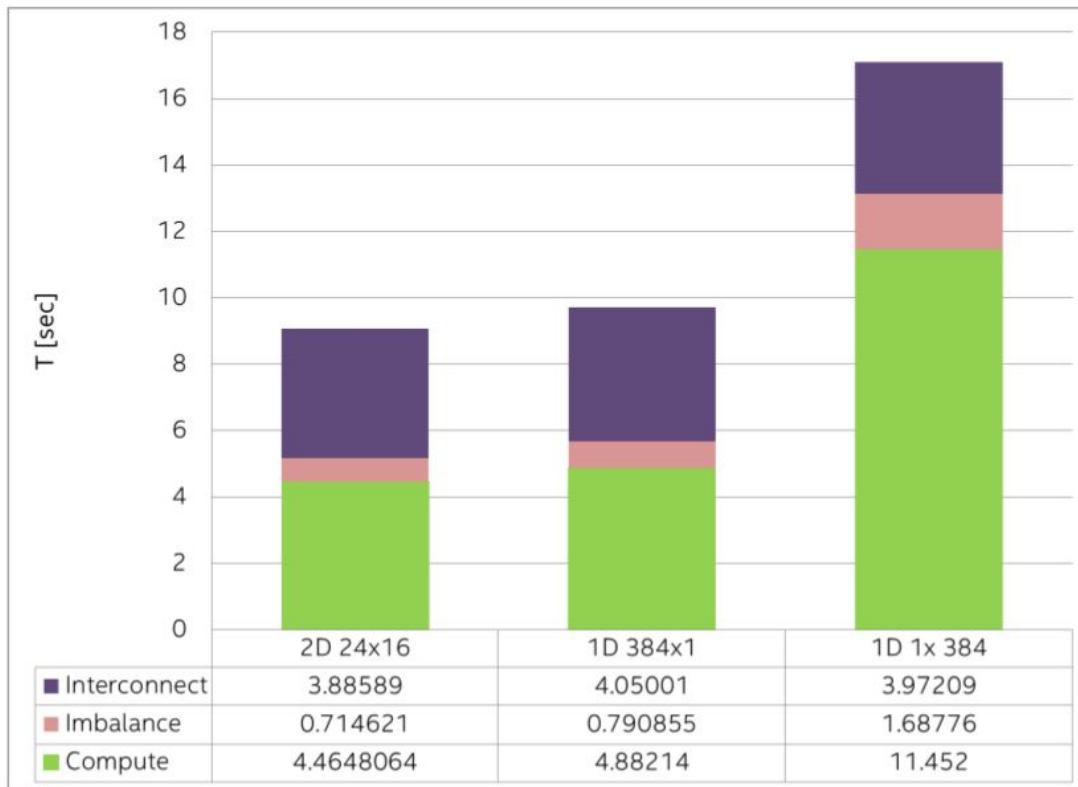
Compute time for 1x384 is almost 3X longer probably because of short vector length $3600/384 < 10$



Imbalance time best for 2D because process grid fits perfectly:
 $\text{local grid} = (3600/24) \times (3600/16) = 150 \times 225$ grid points



Imbalance time for 384x1 slightly worse because number of local grid point rows will vary between 10 and 9 ($3600/384 = 9.375$). See next slide(s) for a discussion about the measurement of imperfect data distribution



Imbalance time for 1x384 is even larger because of longer compute time. The imbalance stretches with compute time

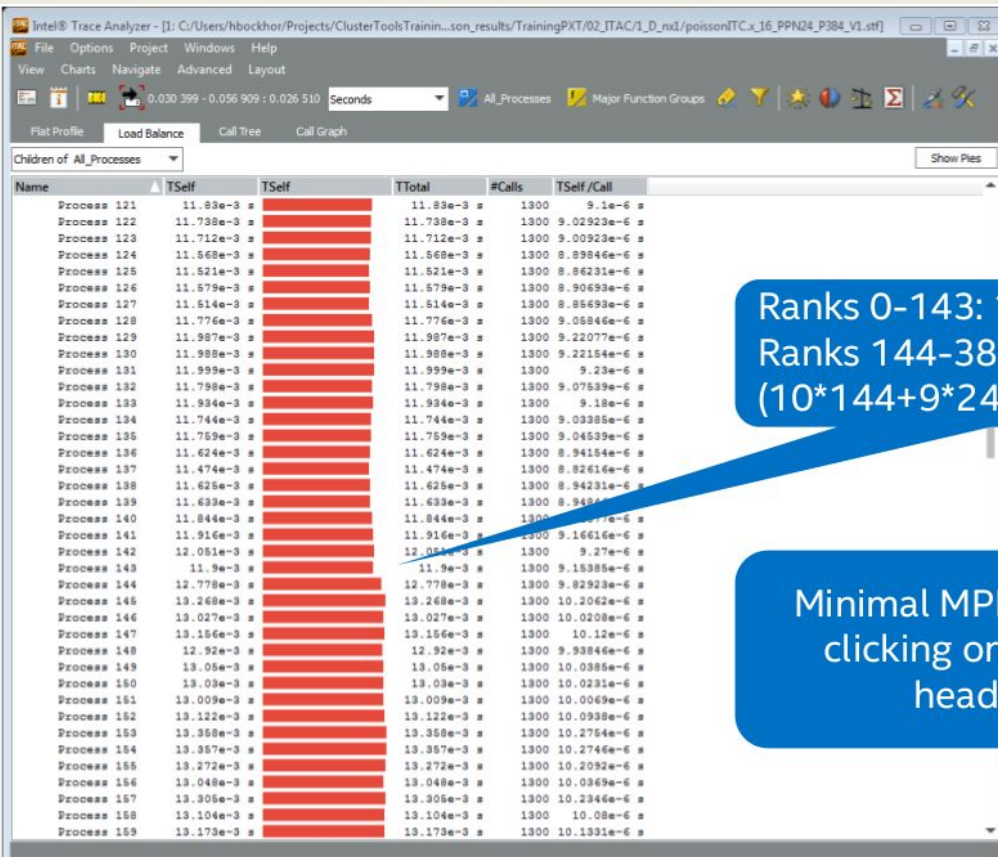
Global load imbalance

A portion of the waiting time is normally due to Global Load Imbalance. The Global Load Imbalance is measured by determining the maximum per rank and average compute time over all processes:

$$\begin{aligned} T_{\text{load}} &= T_{\text{compute_max}} - T_{\text{compute}} \\ &= T_{\text{mpi}} - T_{\text{mpi_min}} \end{aligned}$$

T_{load} is the time we may win by achieving a perfect load balance. It should be lower than the previously calculated MPI time for an ideal network (= $T_{\text{wait}} = \text{Imbalance}/p$)

Load imbalance: MPI for 1D 384x1



Ranks 0-143: 10x3600 points
Ranks 144-383: 9x3600 points
(10*144+9*240) = 3600

Minimal MPI can be found by clicking on TSelf (Column header) → sort

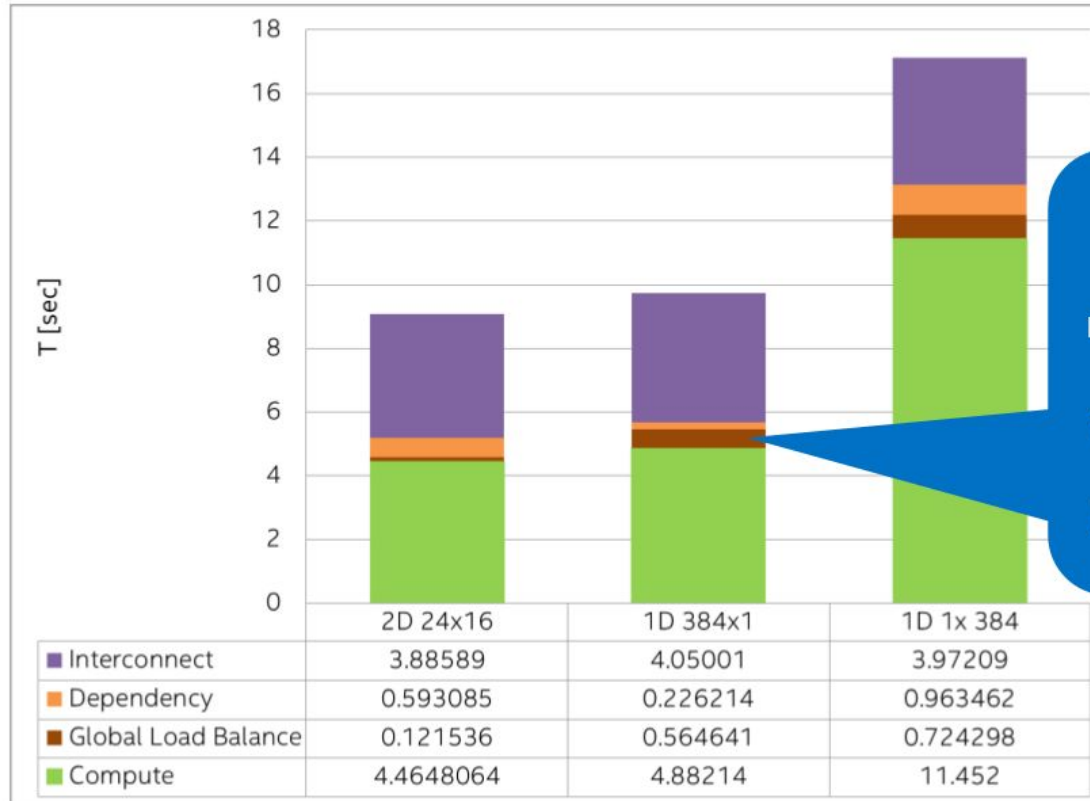
Split of timings

We have now all components of our split of timings:

$$\begin{aligned} T &= T_{\text{compute}} + T_{\text{mpi}} \\ &= T_{\text{compute}} + T_{\text{trans}} + T_{\text{wait}} \\ &= T_{\text{compute}} + T_{\text{trans}} + T_{\text{load}} + T_{\text{depend}} \end{aligned}$$

The Imbalance diagram shows only the second line but we might additionally compute T_{load} and T_{depend} for a deeper analysis. T_{depend} is called Dependency time. This is just the rest of the imbalance time T_{wait} that is not due to the Global Load Imbalance.

Refined imbalance diagram



Global Load Imbalance is more severe for 1D. This is consistent with the different local grid sizes!

Message passing profile

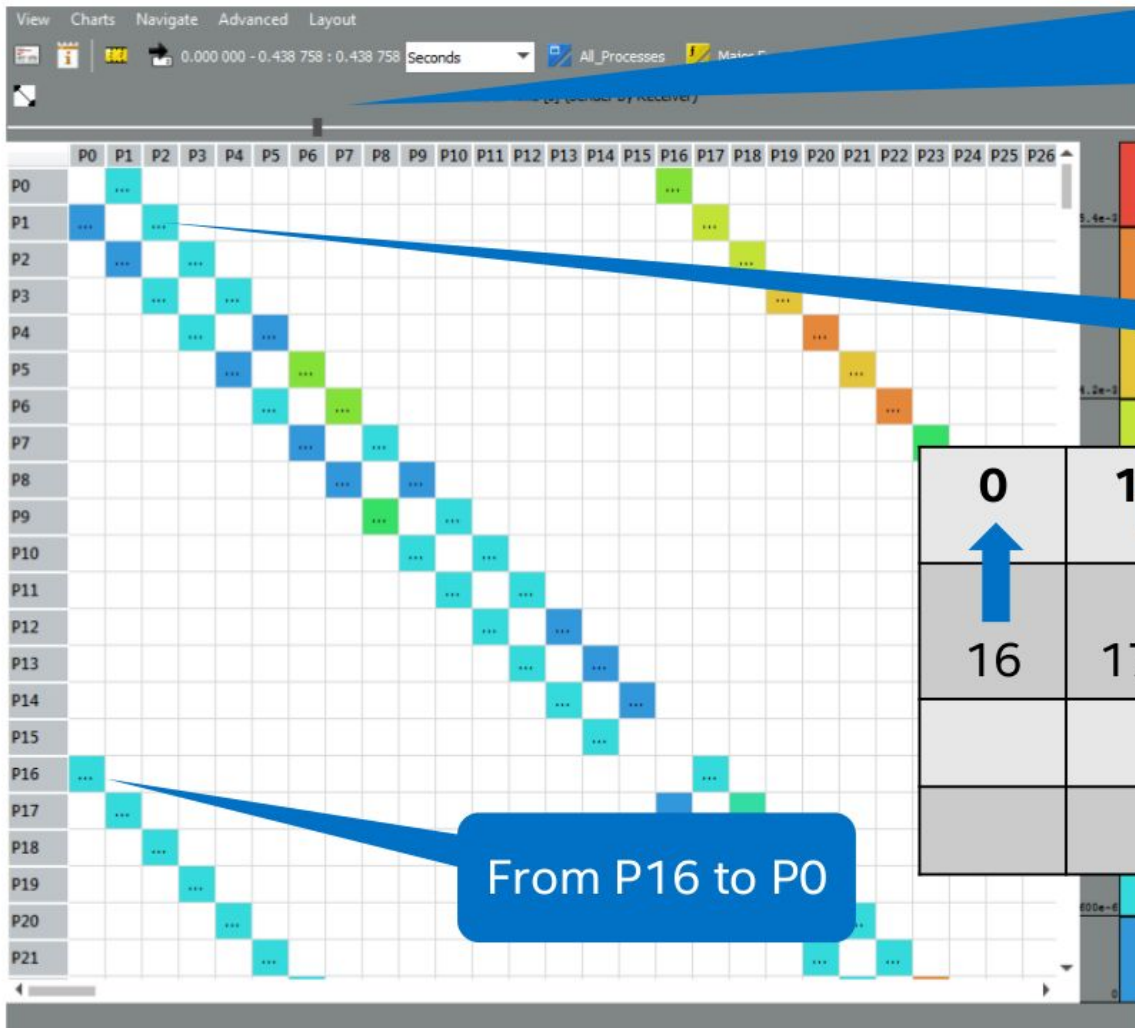
Message passing profile displays various characteristics of message passing in a sender/receiver Matrix

Charts -> Message Profile

The Matrix element N,M corresponds to the message passing characteristics from rank N to rank M. Change these attributes by:

Right Click -> Attribute to show

Characteristics are: total message volume, message passing time, max, min, average rate and count



Use slider for changing the size of cells or:
 Message Profile Settings → Automatic Cell Size

Messages from P1 to P2

From P16 to P0

0	1	2	...	14	15
16	17	18	...	30	31
			...		

Process aggregation

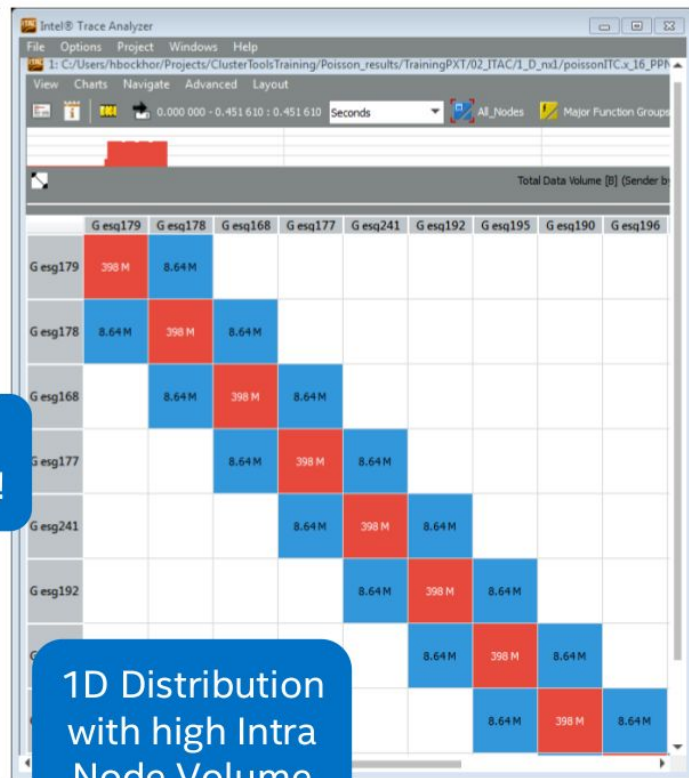
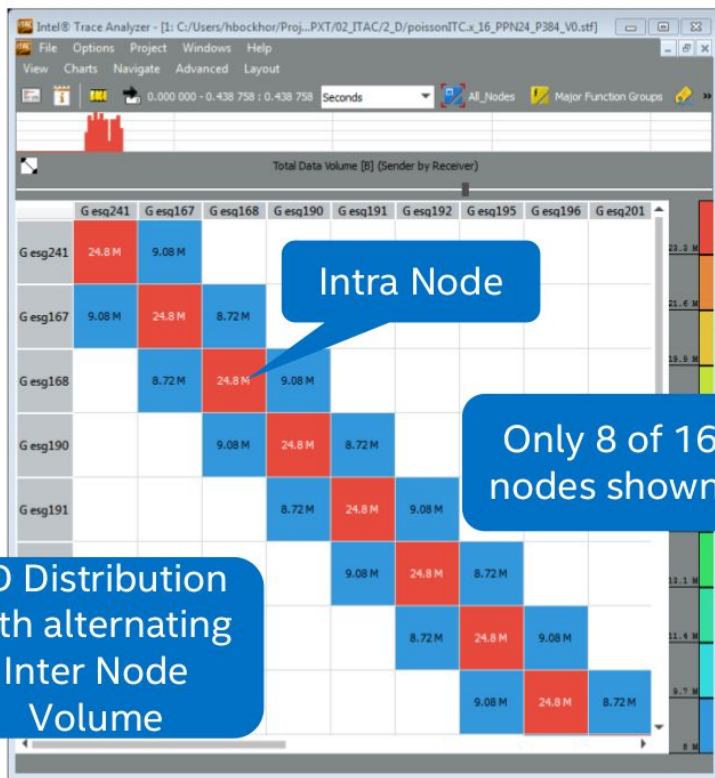
For 16 nodes (384 ranks on IVB) the total Message Passing profile is not very handy

We may fuse the communication to compute node level. In this case 384 ranks are fused to 16 compute nodes:

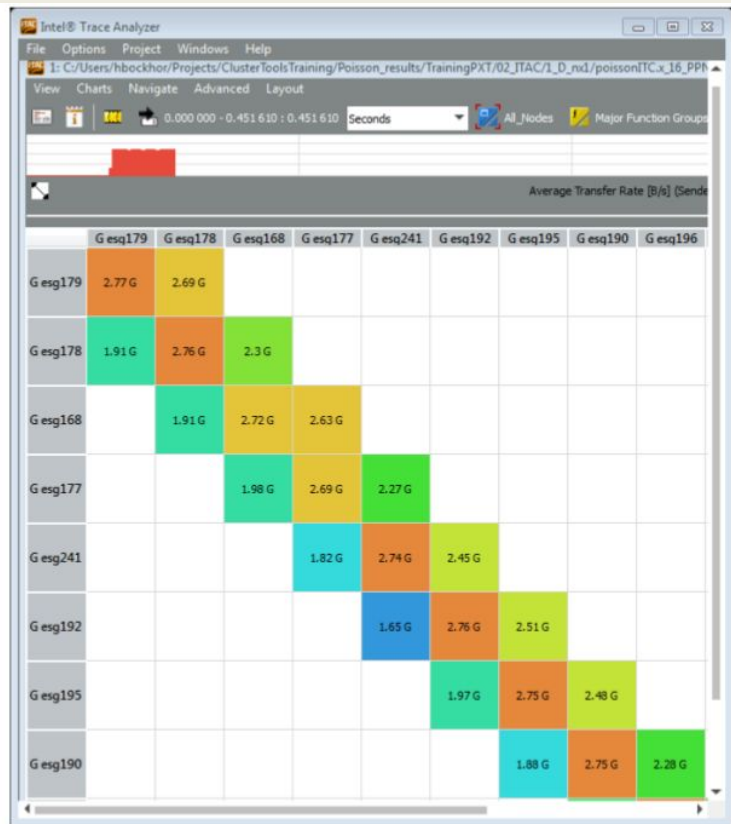
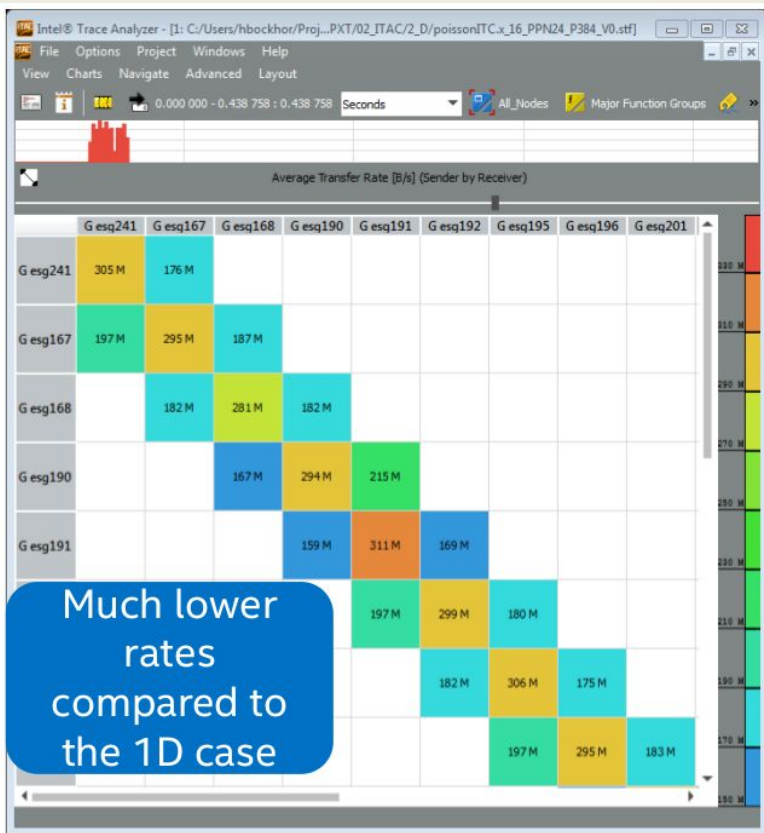
Advanced Process -> Aggregation

This will pop up a new window: check All_Nodes and apply

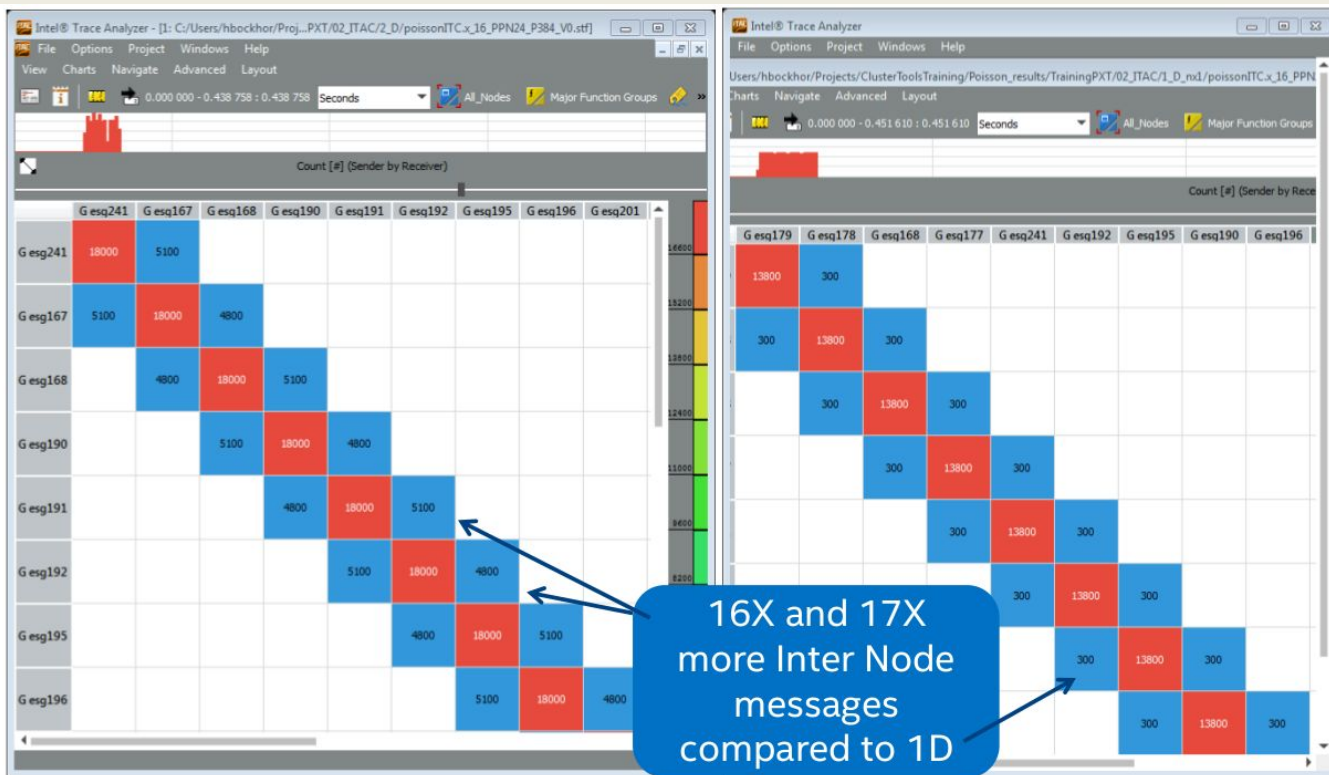
Total volume 2D vs 1D distribution



Average rate: 2D vs 1D distribution



Number of messages: 2D vs 1D dist



Message profile: observations

Inter node communication has about the same volume in the 2D case but 16x more messages are sent

There is just a single inter node message per boundary exchange in the 1D case (3 exchanges per iteration times 100 iterations == 300 messages)

Communication rate drops so much in the quadratic 2D case that the total transfer time (Imbalance diagram) is almost equal for both configurations

Optimization hints

A compromise between quadratic and 1D processor grid may be more appropriate here like 48x8 or 96x4. This will reduce the number of inter node messages and raise the bandwidth for each message

The default rank to node mapping is just linear. This leads to alternating communication patterns (see following slides)

A better mapping can be achieved by putting all ranks of a rectangular sub process grid onto a single node. The following slides explain the ranks to node mapping

Default mapping for 24x16 grid

0	1	2	...	7	8	...	14	15
16	17	18	...	23	24	...	30	31
32	33	34	...	39	40	...	46	47
48	49	50	...	55	56	...	62	63
64	65	66	...	71	72	...	78	79
80	81	82	...	87	88	...	94	95
96	...							

- Node #0
- Node #1
- Node #2
- Node #3

24 ranks per node! One rank per core

Additional horizontal exchange. 16 or 17 boundary lines between two nodes

Defining a 16x24 process grid may be better – why?

Optimal mapping

0	...	3	4	...	7	8	...	11	12	...	15
16	...	19	20	...	23	24	...	27	28	...	31
32	...	35	36	...	39	40	...	43	44	...	47
48	...	51	52	...	55	56	...	59	60	...	63
64	...	67	68	...	71	72	...	75	76	...	79
80	...	83	84	...	87	88	...	91	92	...	95
96	...	99	100	...							

Node #0

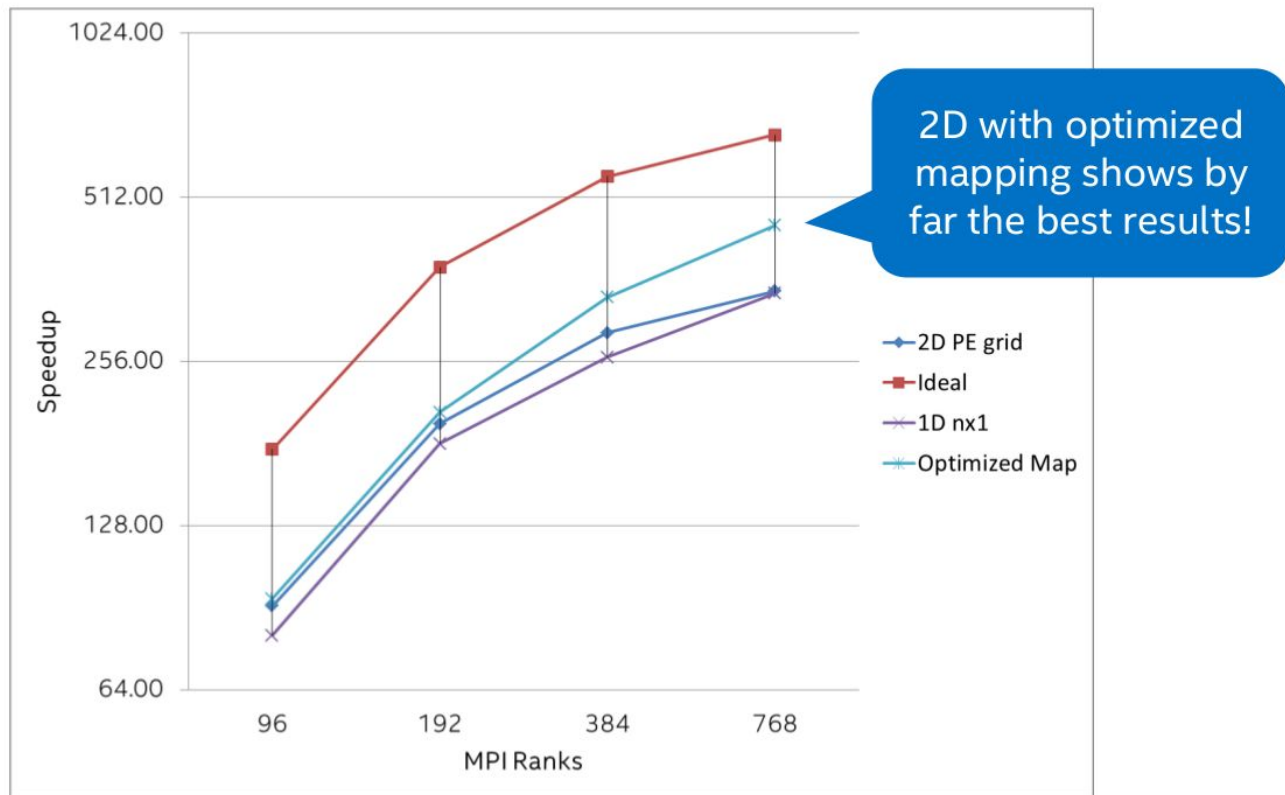
Node #1

Node #2

Node #3

This 6x4 pattern can be repeated for all nodes. The number of processor boundary lines between nodes are: 4 (vertical) and 6 (horizontal)

Impact of mapping



Detailed visualization of MPI programs

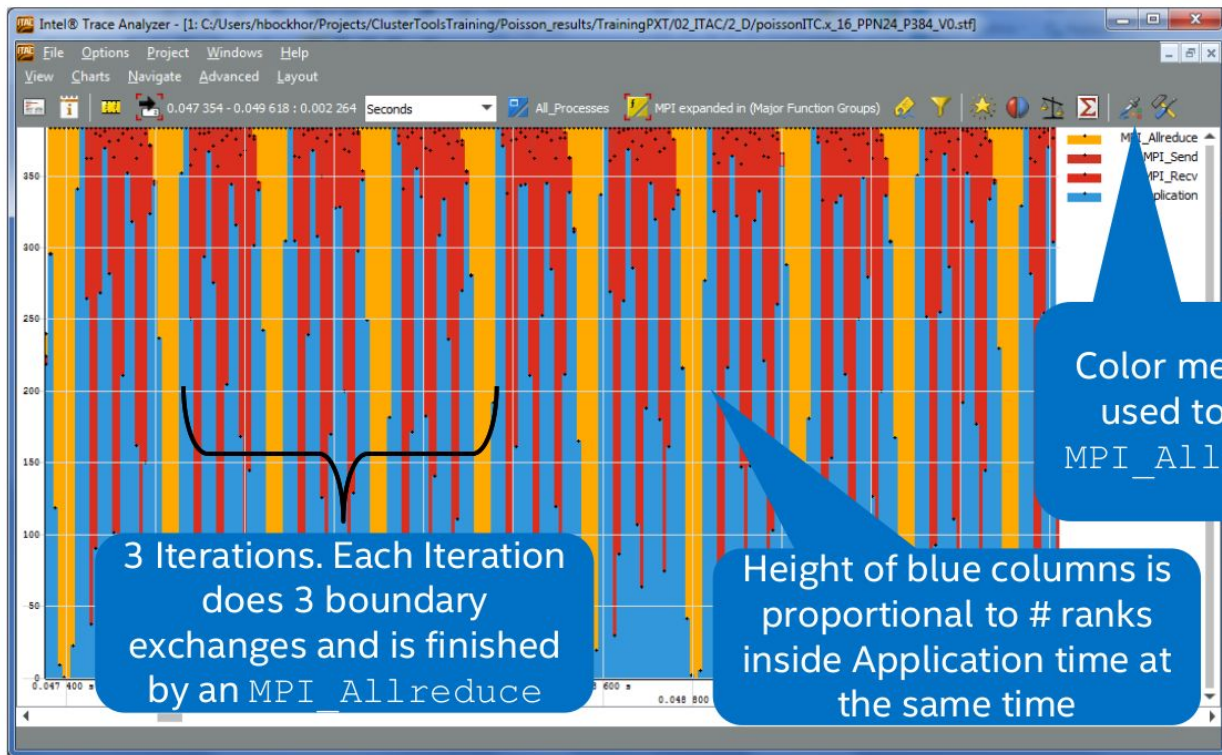
After some global evaluations we may dive now into the MPI algorithm by showing the temporal evolution with ITAC

Most programs consist of recurring patterns like iterations or different phases: initialization, computation and I/O

Quantitative timeline shows nicely coarse patterns:

Charts -> Quantitative Timeline

Quantitative timeline



Event timeline

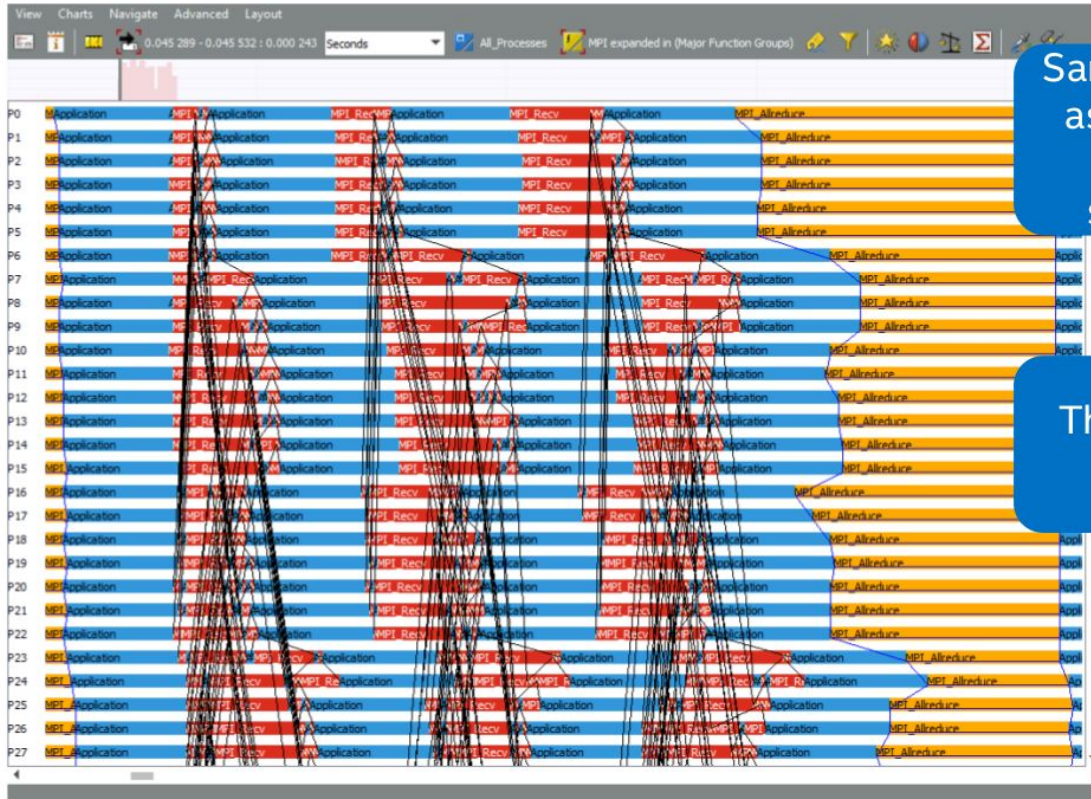
After identification of basic patterns we may now change to the more detailed Event Timeline

Event timeline is the most important Chart in ITAC

Temporal development reveals root causes of dependencies due to suboptimal implementations

Charts -> Event Timeline

Single iteration



Same configuration as used before in the mapping section: 24x16

This is the default mapping!

Boundary exchange in ideal networks

MPI times in the ideal network case are due to global load imbalances and dependencies

Dependencies are e.g. due to order of blocking sends and receives

The current naive implementation of the boundary exchange uses blocking sends and receives: `MPI_Send`, `MPI_Recv`

The Ideal network simulation helps to clearly identify dependencies

Boundary exchange in ideal networks



An MPI_send could have started already here!

This Send (P5 → P6) may be started long before

Optimization hints

Some of the dependencies may be resolved by using `MPI_Isend` and `MPI_Irecv` with an `MPI_Waitall()` in the end

In a first step we may just exchange the blocking `Sends/Recv`s by the immediate routines and place a `MPI_Waitall()` at the end.

Data copies of boundary arrays have to be done after the wait routine

In a second step we may optimize the order of MPI routines and data copies. Some requests may be ended by a separate `MPI_Wait()`

Comparing ITAC traces

Compare before and after optimization e.g. compare boundary exchange with blocking Send/Receive to non blocking Send/Receive

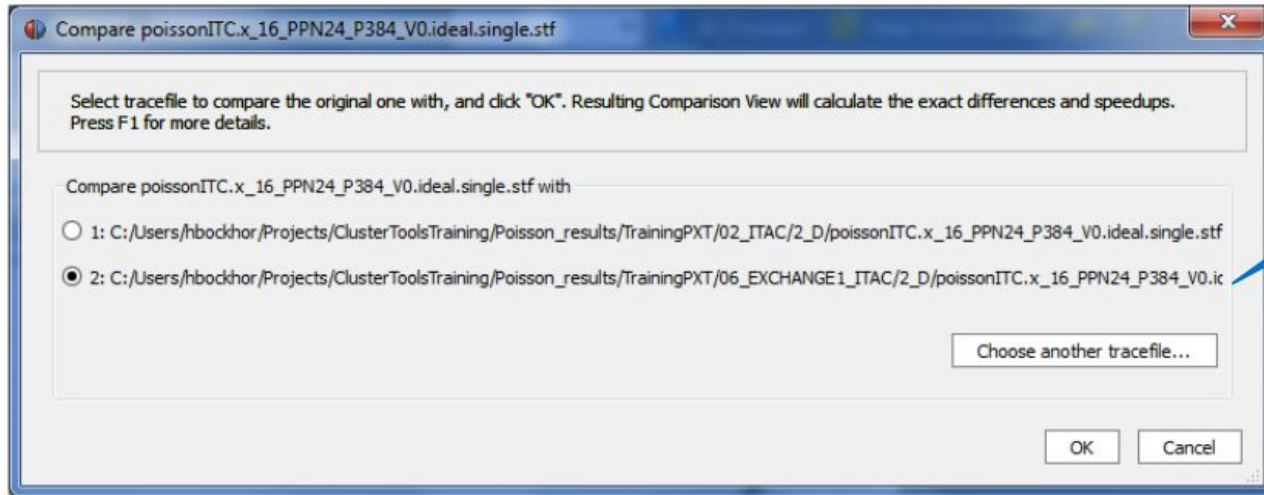
Further potential comparison scenarios:

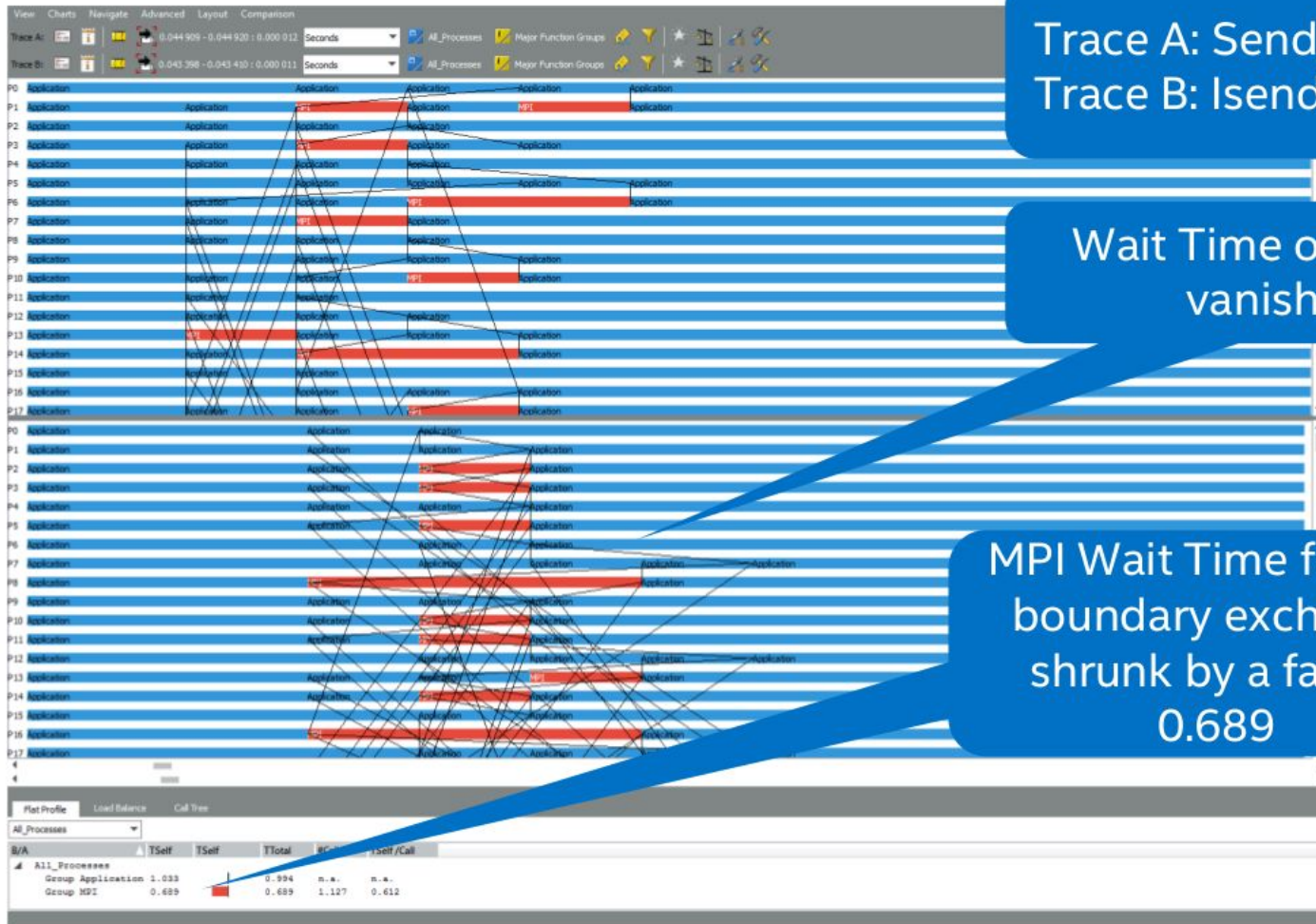
- Compare ideal to real trace
- Compare different number of ranks
- Compare different mappings

Comparing ITAC traces

Open tab: View → Compare

Open another
file for a
comparison





Trace A: Send/Recv
Trace B: lsend/lrecv/waitall

Wait Time on P6 has vanished!

MPI Wait Time for this boundary exchange shrunk by a factor 0.689

Instrumentation of user functions

So far, we only see MPI routines and Application time inside ITAC traces

Navigation becomes far more easy when adding user functions

For evaluation of the impact of optimization we may want to see the timing of the boundary exchange including all its MPI calls

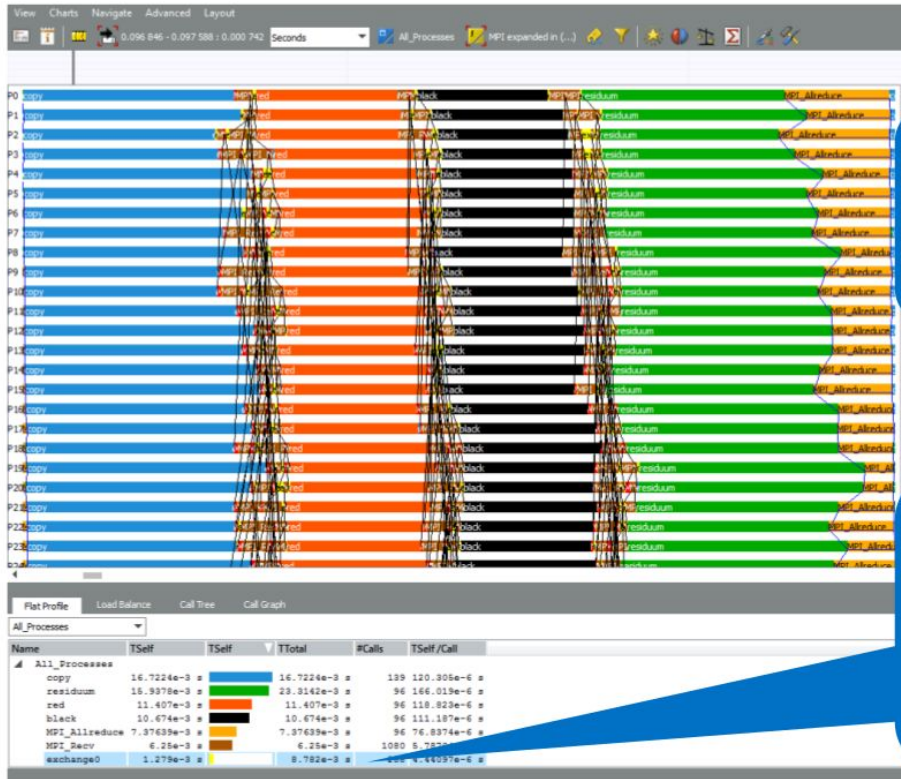
ITAC compiler instrumentation

All source files or just the files of interest may be compiled with the *-tcollect* flag (Intel compiler only)

The executable has to be linked using this flag, as well

As an alternative (different compiler or code blocks that are not a function) we might consider to use the ITAC API functions for instrumentation.

User functions



Instrumented user functions like exchange0 can improve analysis of the MPI algorithm

Exchange routine is on bottom of the list. But the total time TTotal also contains all MPI functions. This time exceeds the Allreduce time

Intel VTune Amplifier XE

We used ITAC for the analysis of the message passing algorithm

We already saw that computation performance saturated on a single node

With this tool we may have a closer look to the processor performance and program structure

VTune Amplifier XE based analysis can be started and performed by its GUI. Together with MPI on a Cluster which probably prefers batch usage, we will use the command line interface

Hotspot analysis

This is the most basic analysis type to start an investigation

The analysis will present hotspots of the calculation for a chosen MPI rank. Timings go down to source lines or assembly code

The Call Stack provides information about how the function is called and how much time is due to this branch

Collecting data

This analysis may be conducted for each of all 384 ranks but probably we may concentrate on a single rank first:

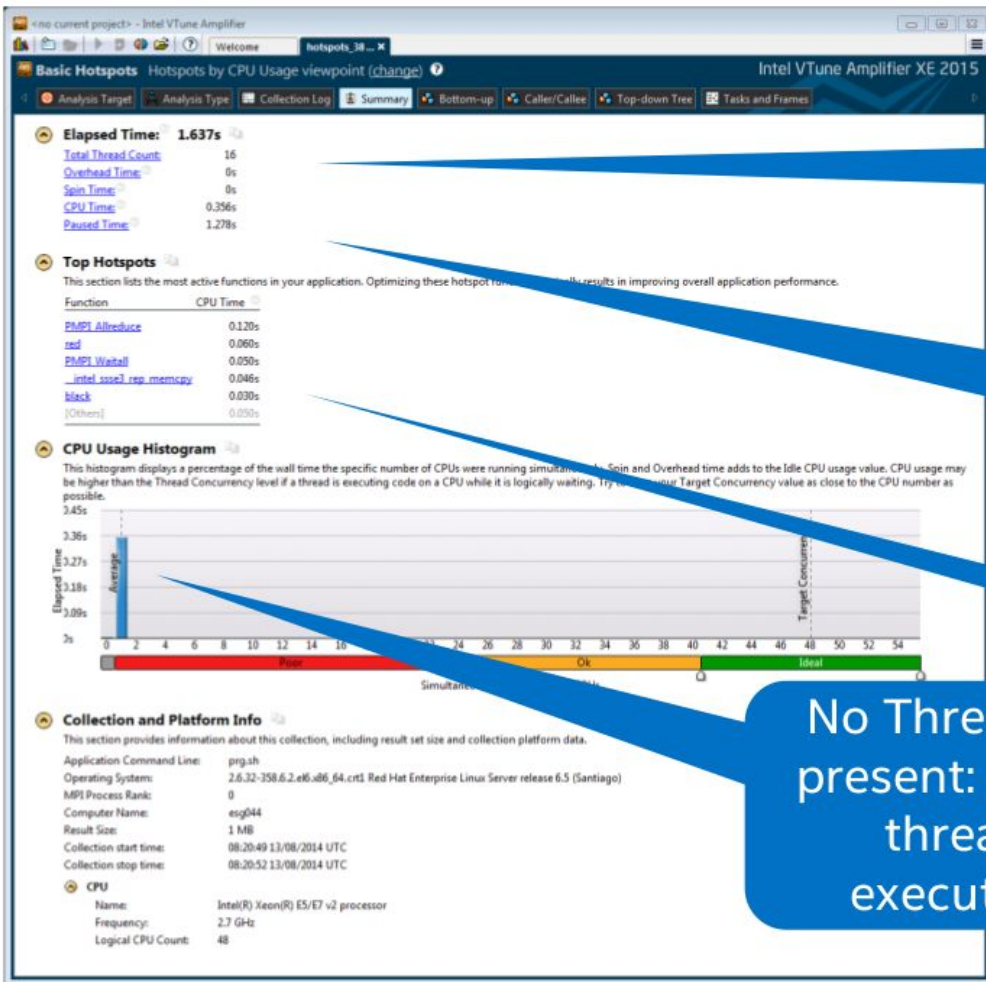
```
mpirun -n 1 amplxe-cl --result-dir hotspots \  
      --collect hotspots -- poisson.x :\  
-n 383 poisson.x
```

Hotspot analysis is performed on rank 0 and results are stored in directory hotspots.0. All other ranks run poisson.x without analysis. More complex selection of ranks are possible building groups of ranks doing analysis or not

Collecting data

A new syntax:

```
mpirun -n 1 amplxe-cl --result-dir hotspots \  
      --collect hotspots -- poisson.x :\  
-n 383 poisson.x
```

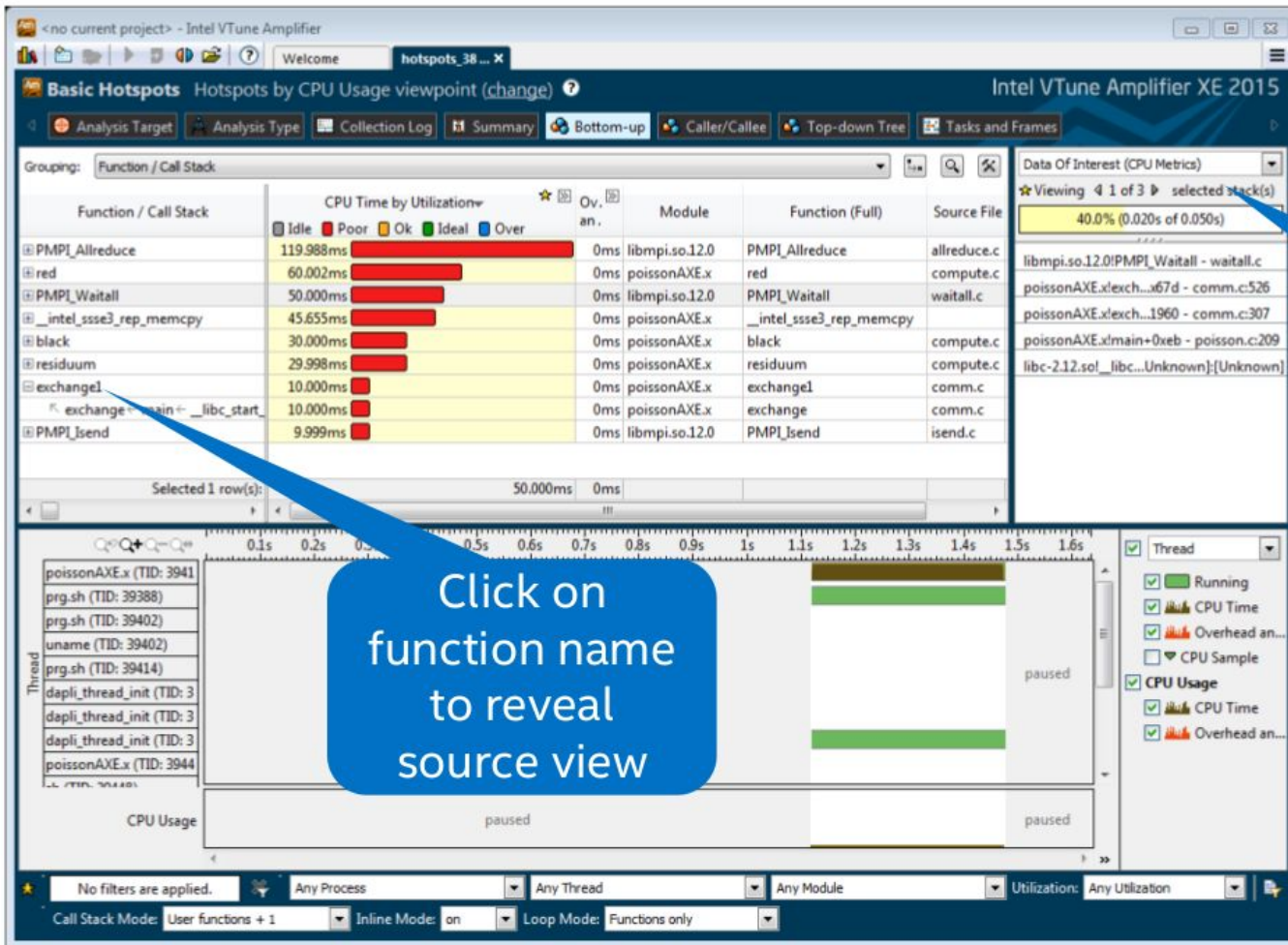


Ignore Thread count.
It is just a single MPI
process

Analysis only for
iterations. Rest of app.
Is paused by using
API functions

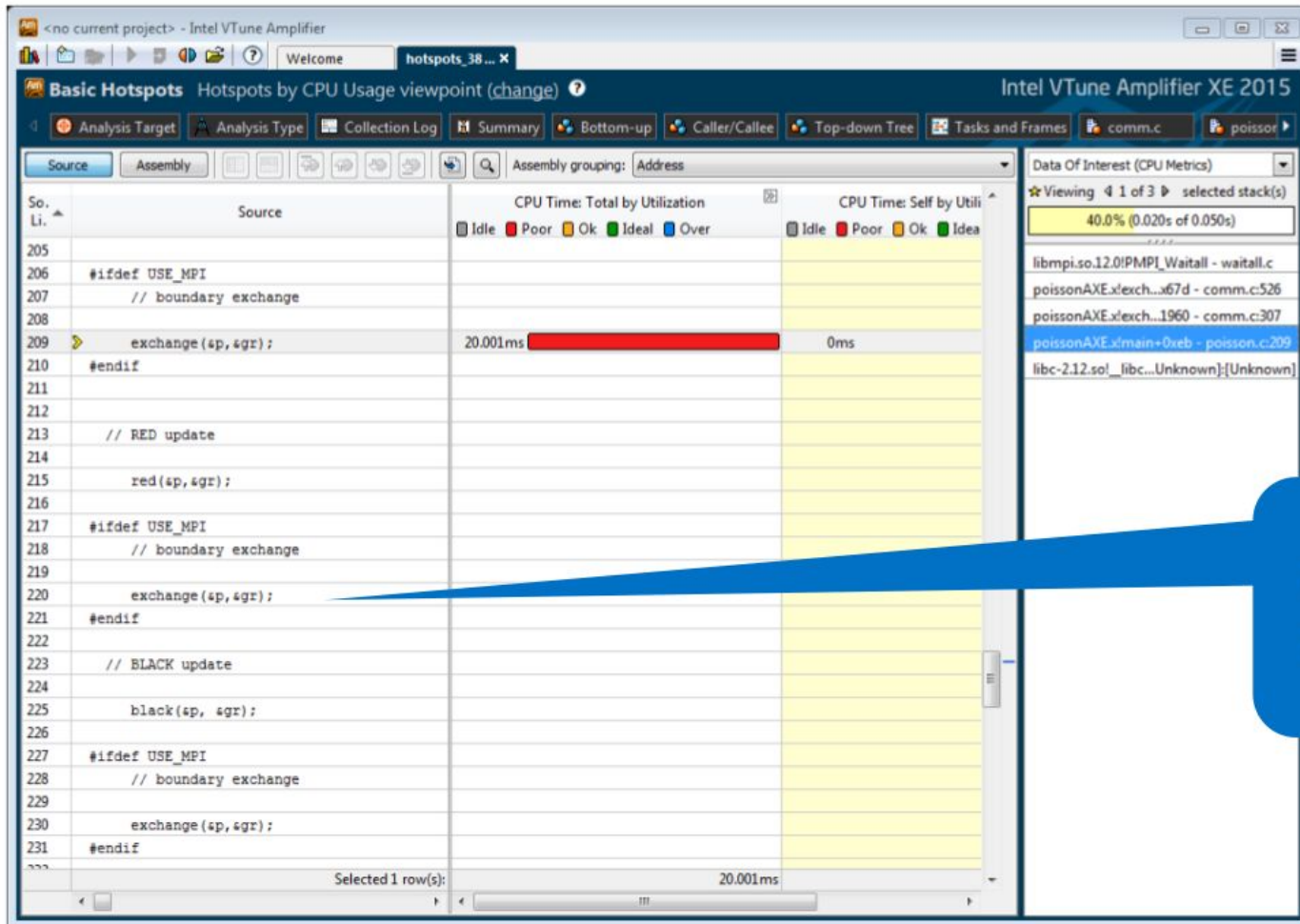
No Threading
present: single
thread
execution

memcpy is called by
copy routine



Click on function name to reveal source view

3 different stacks for MPI_Waitall. Source line of call to exchange in poisson.c is shown. Exchange is called 3 times!



Click on first stack and poisson.c line: shows first call to exchange at line 209

Second call at line 220 shows up by selecting another stack

Advanced hotspots

Hotspot identification using directly the Performance monitoring Unit (PMU) . Needs special drivers realized by kernel modules (root rights necessary for installation)

Exchange **hotspots** by **advanced-hotspots** in previous command line

Instructions retired is the basic indicator for processor utilization.

Maximum is 4 simultaneous instructions per clock-tick.

The output shows CPI: clock-ticks per Instruction. 4 simultaneous

The screenshot shows the Intel VTune Amplifier XE 2015 interface. The main window displays performance metrics for an application. The 'Elapsed Time' is 1.378s. The 'Instructions Retired' is 1,377,000,000. The 'CPI Rate' is 0.686. The 'CPU Frequency Ratio' is 0.999. The 'Paused Time' is 1.023s. The 'Overhead Time' is 0s. The 'Spin Time' is 0s.

The 'Top Hotspots' section lists the most active functions in the application. The functions and their CPU times are:

Function	CPU Time
residuum	0.058s
MPIDI_CH3I_Progress	0.055s
red	0.053s
__intel_ssse3_rep_memcpy	0.048s
black	0.047s
[Others]	0.091s

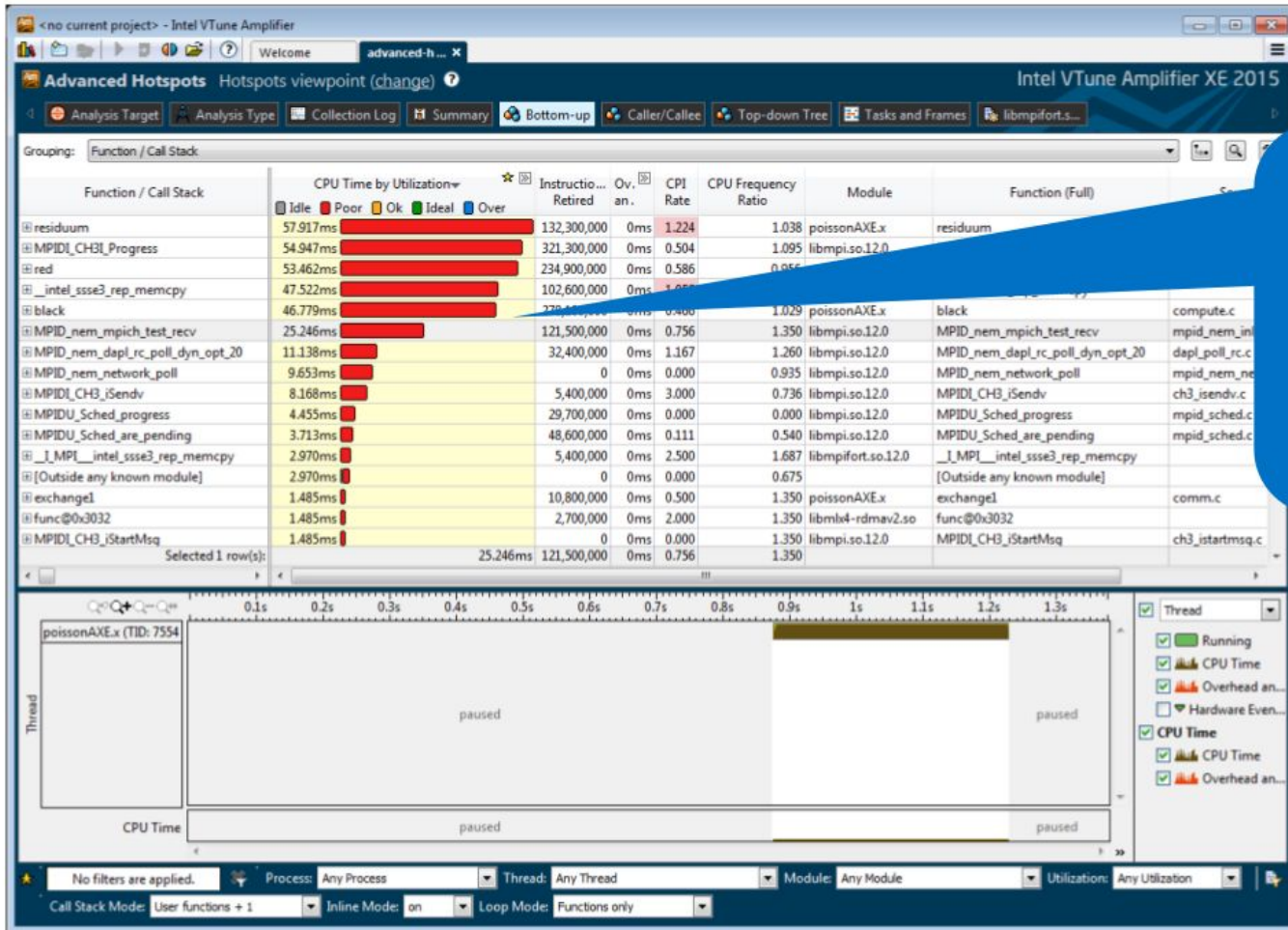
The 'CPU Usage Histogram' section is also visible. The 'Collection and Platform Info' section provides information about the collection, including result set size and collection platform data.

The 'CPU' section shows the processor details:

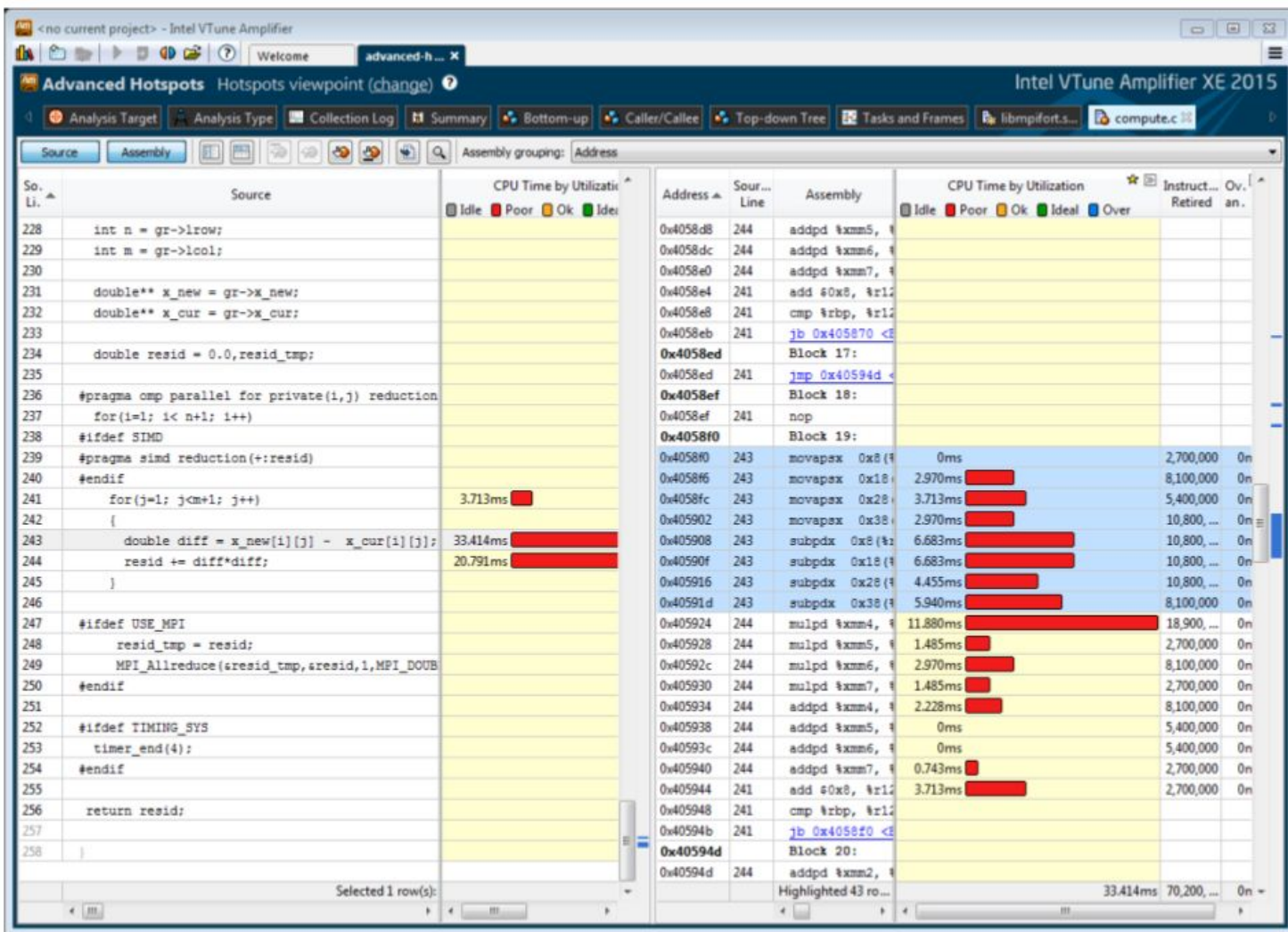
Name:	Intel(R) Xeon(R) E5/E7 v2 processor
Frequency:	2.7 GHz
Logical CPU Count:	48

Instructions Retired: completed Instructions
 CPI Rate: Clock Ticks per Instruction
 CPU Frequency Ratio: >1 : Turbo boost!

Second routine comes from MPI – Progress Engine
 More MPI internal functions shown



Order of Hotspot functions changes due to: 1. better time resolution 2. Internal MPI functions are displayed



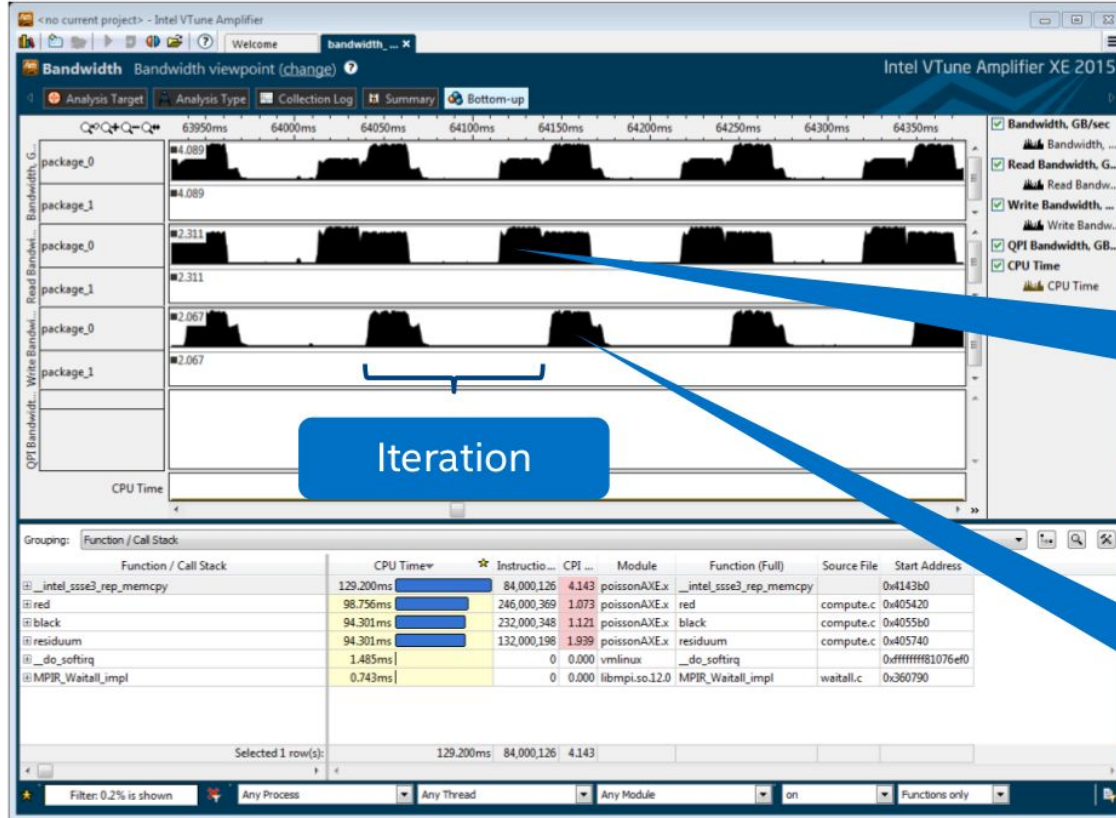
Bandwidth analysis

The speedup curve for a single node shows saturation for more than 12 ranks per node (24 cores per node in total)

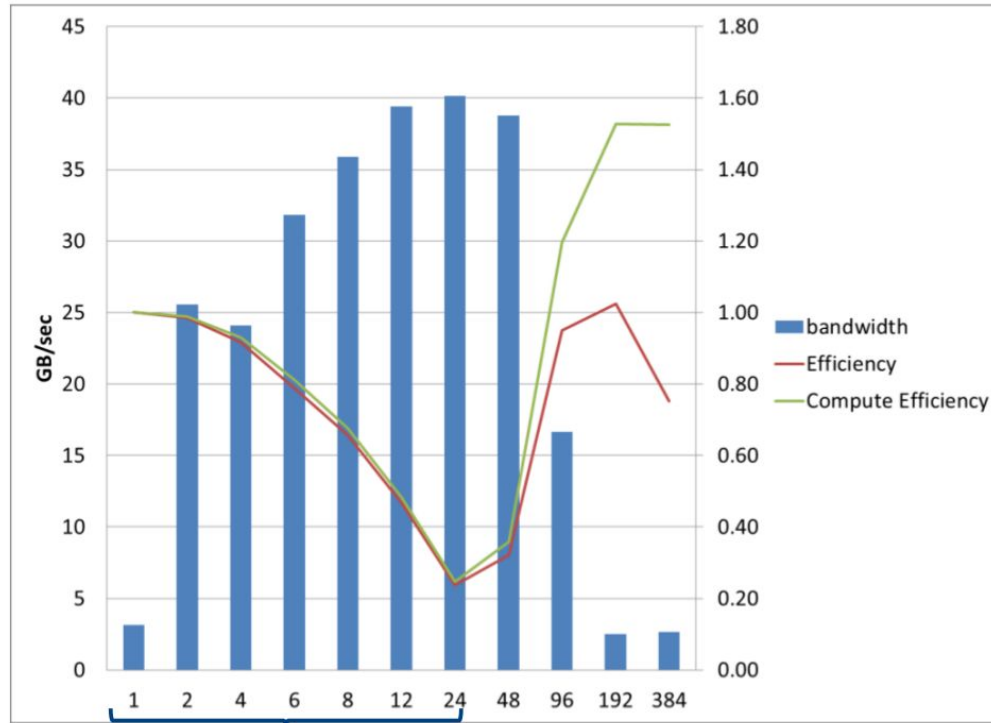
Intel® VTune Amplifier XE provides a Bandwidth analysis for proving this assumption

We concentrate on total bandwidth which can be related to the bandwidth that is delivered by the STREAM benchmark (~80GB/s on IVB dual Socket)

BW: Bottom-up sequential run



Efficiency vs BW



Optimization hints

Bandwidth can be reduced by combining copy and residuum routine.

This is possible because residuum is at the end and copy at the beginning of a new iteration

Bandwidth reduction may only have an impact in the bandwidth limited regime that we observe for this grid size only for less than 4 nodes

Prefetching of data may also improve performance in the copy and reduction loop

A blocked loop structure for the iteration loop may also improve data reuse

Summary

Some methodologies were presented for performing a MPI analysis

ITAC offers interesting new features like simulation of ideal traces and the computation of transfer and waiting time

Intel® VTune™ Amplifier XE analyzes the compute part of the application. Bandwidth analysis is useful for many HPC applications