MPI and OpenMP
common bottlenecks and how to spot them

l.bellentani@cineca.it
Tutorial on Profiling @ MHPC - ICTP

Size matters

How much work 1 worker can do is limited by his speed.

A single worker can only move so fast.

Size matters

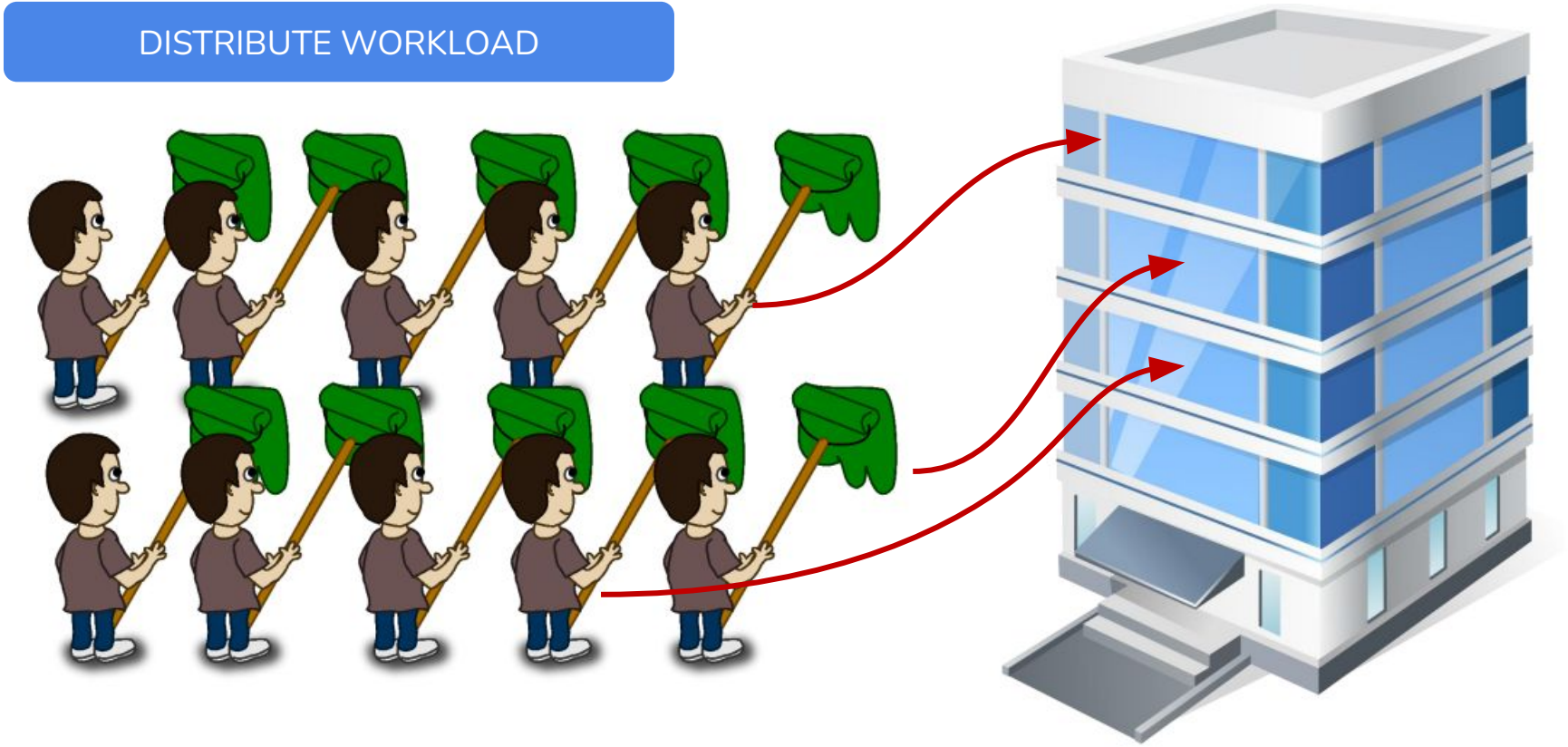
Even if we increase the size of his roller, he can only paint so fast.

We need more workers!

Union is strength

Union is strength

DISTRIBUTE WORKLOAD

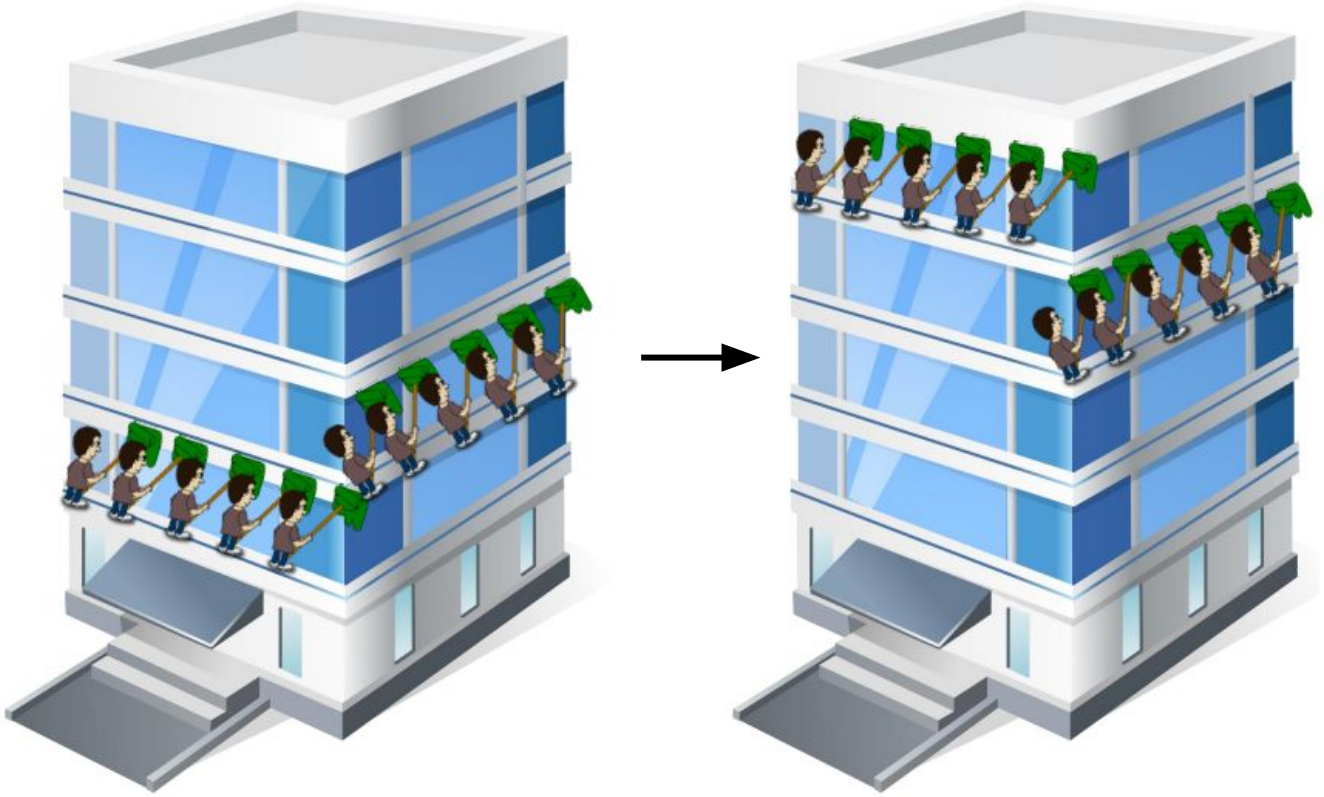


Union is strength

SYNCHRONIZE

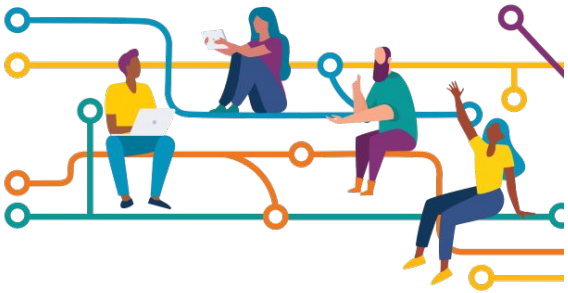
EXCHANGE INFORMATION

Union is strength



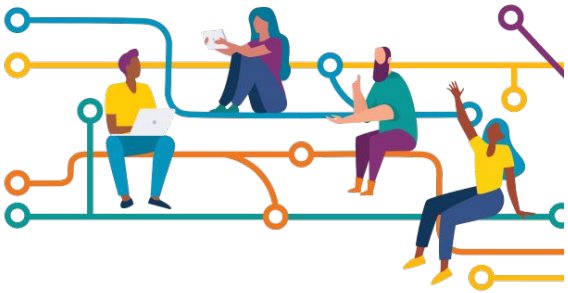
You need a good team management

Which are the main causes of bottlenecks for an MPI application?



You need a good team management

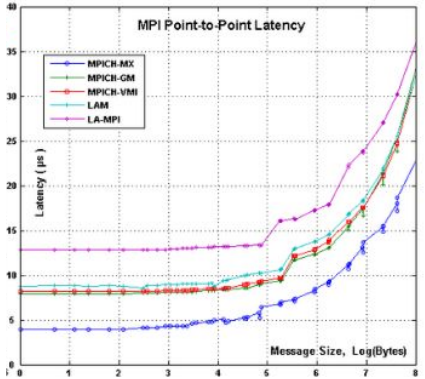
1. **DEPENDENCIES**
2. **MESSAGE TRANSFERS**
3. **WORKLOAD DISTRIBUTION**



Network metrics

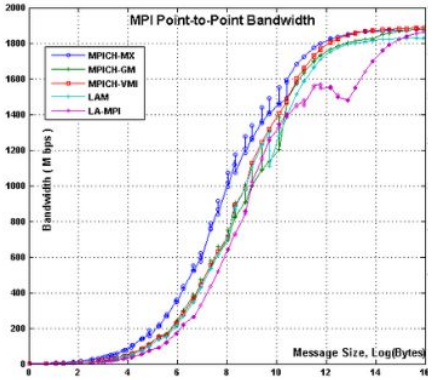
Latency :

The time it takes for a message of size 0 to get to a destination



Bandwidth :

The maximum rate at which data can flow over the network.

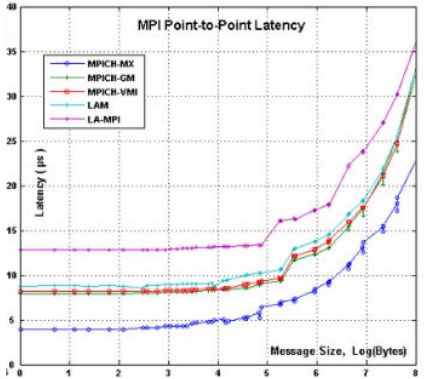


Network metrics

Latency :

The time it takes for a message of size 0 to get to a destination

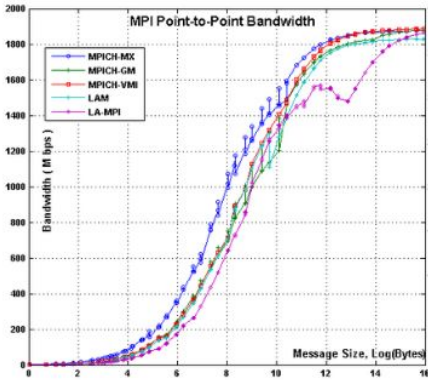
- Dominates the performance of small messages
- Better to bundle multiple small messages into larger one



Bandwidth :

The maximum rate at which data can flow over the network.

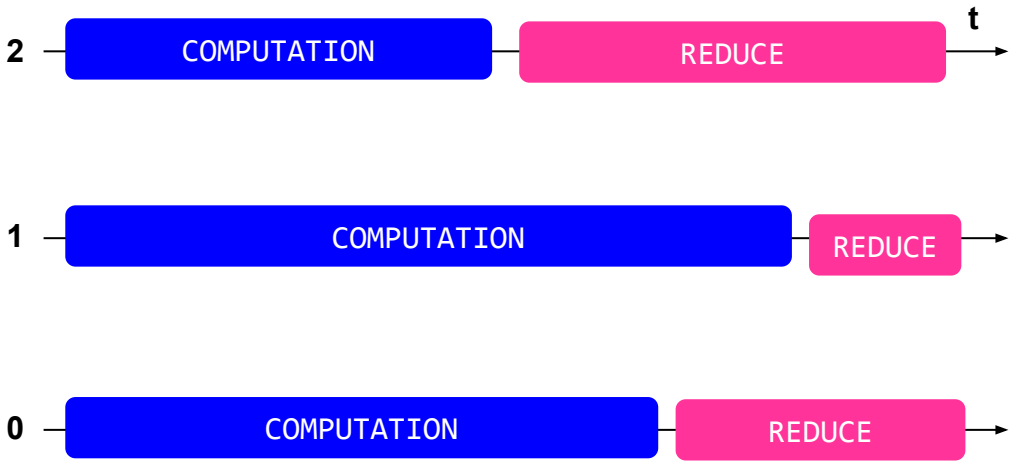
- Dominates the performance of large messages
- Improve network topology or mapping



$$T = L + (1/BW)*V$$

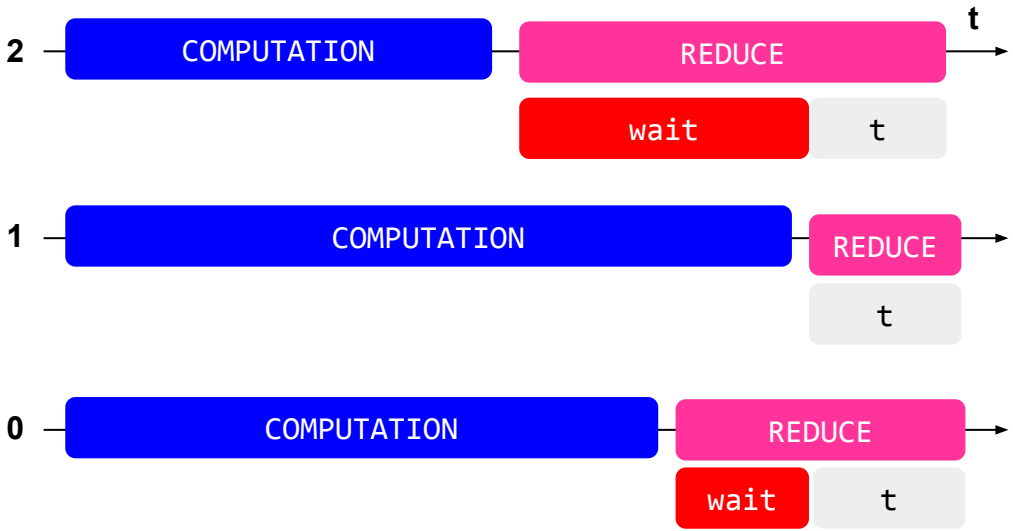
Workload imbalance

MPI procs might spend time in barriers inside MPI calls (“MPI wait time”)



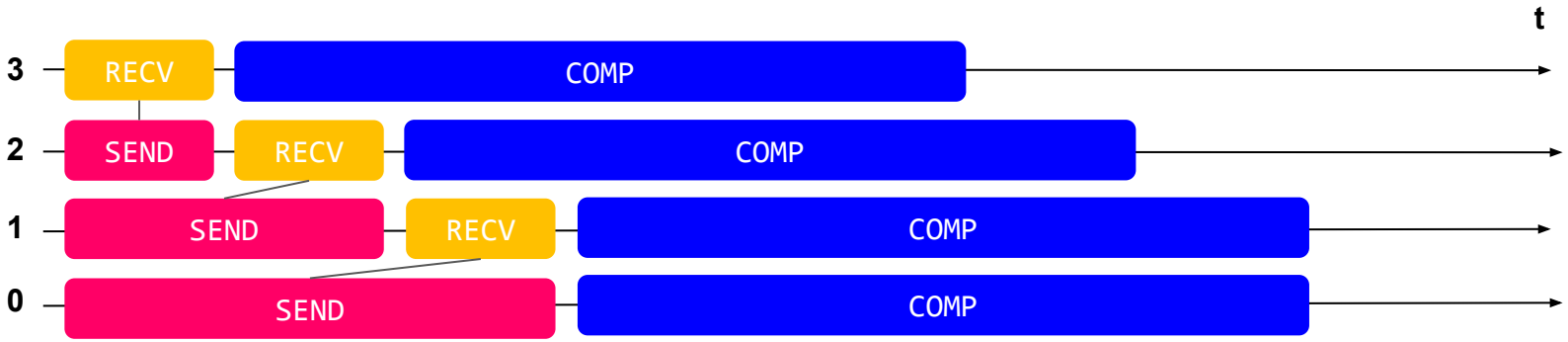
Workload imbalance

MPI procs might spend time in barriers inside MPI calls (“MPI wait time”)



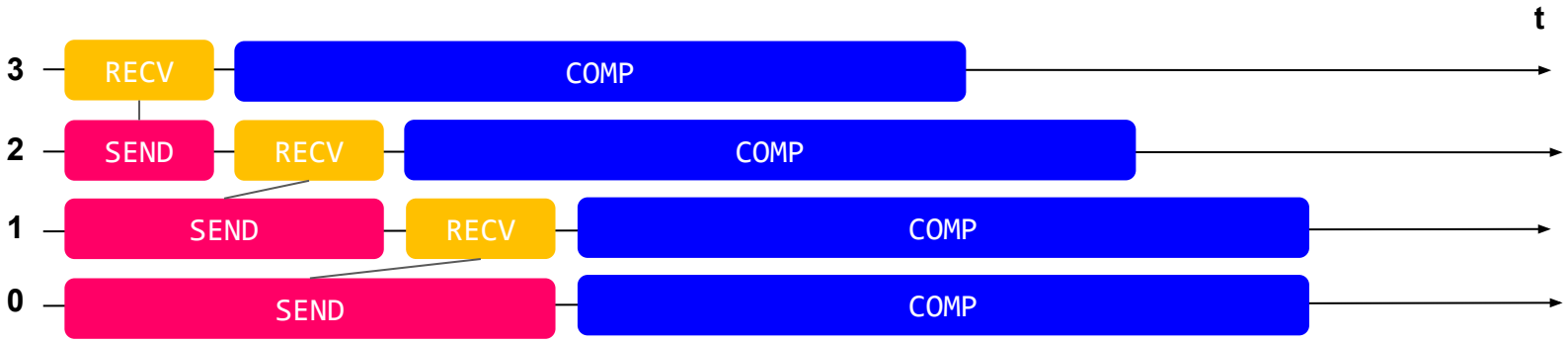
Dependencies

```
MPI_Recv(&token, 1, MPI_INT, rank - 1, 0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Send(&token, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Dependencies

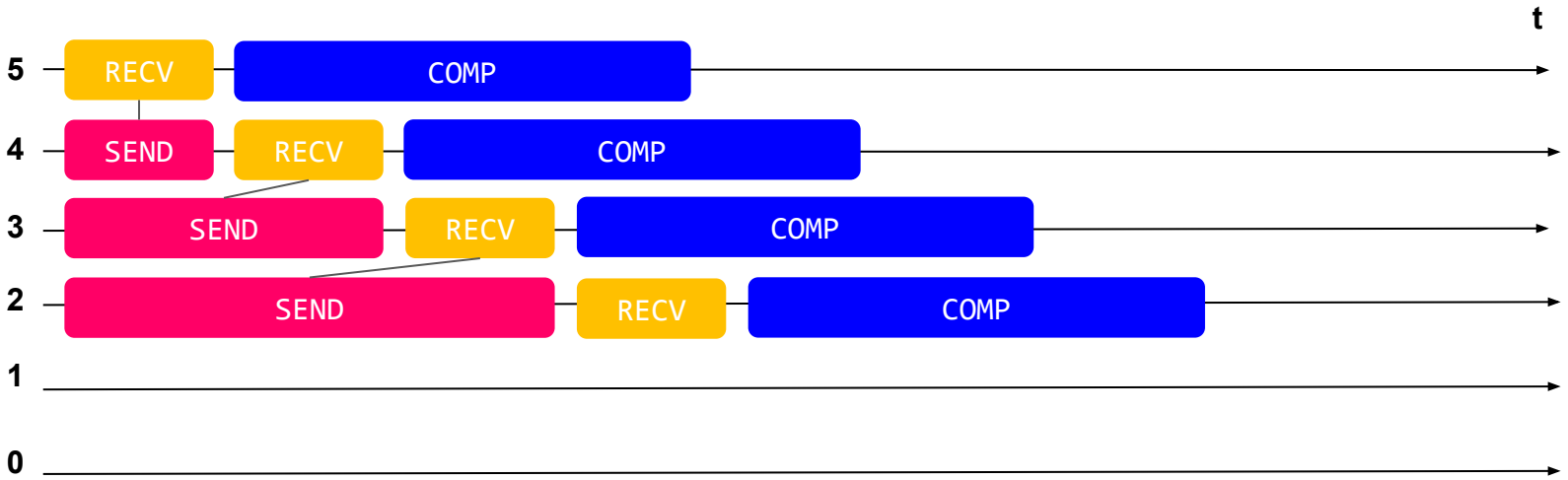
```
MPI_Recv(&token, 1, MPI_INT, rank - 1, 0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Send(&token, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



What happens if the number of MPI ranks increases?

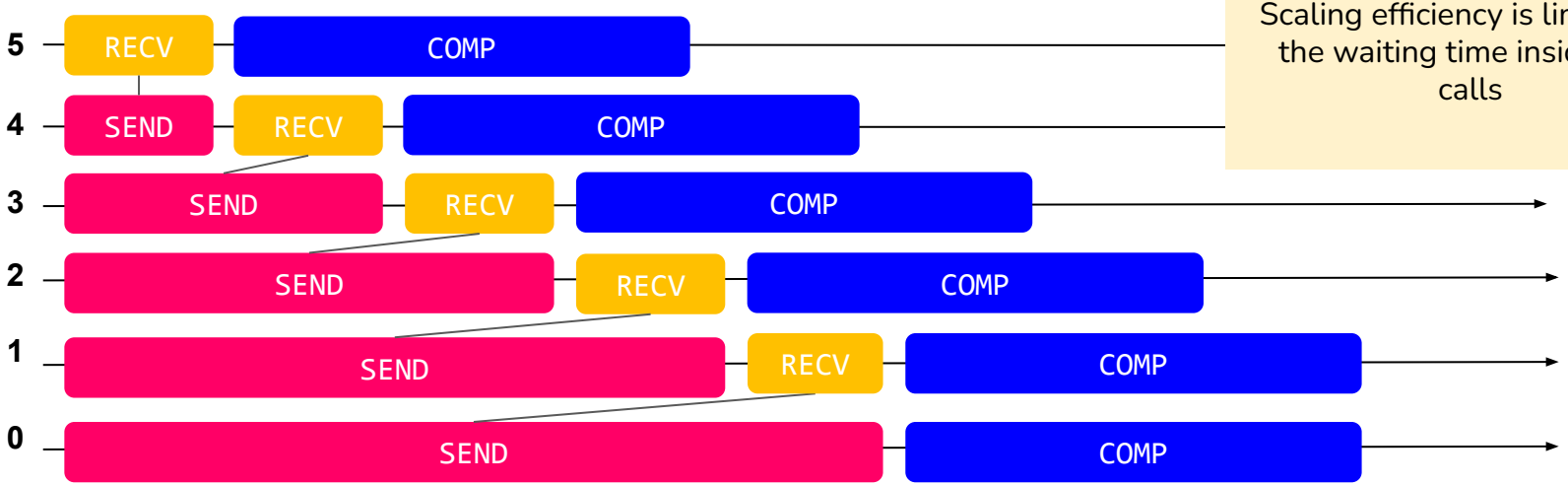
Dependencies

```
MPI_Recv(&token, 1, MPI_INT, rank - 1, 0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Send(&token, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



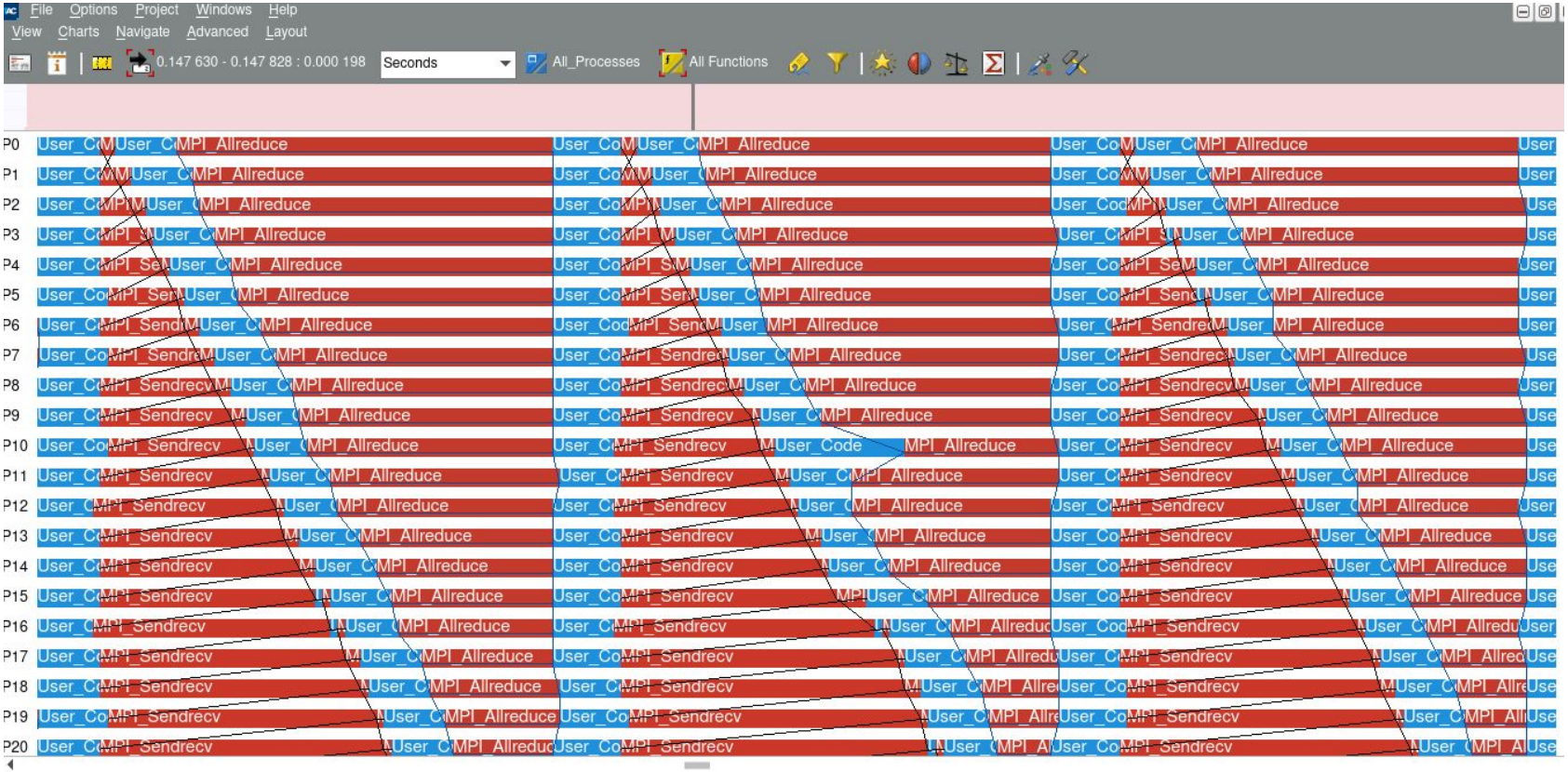
Dependencies

```
MPI_Recv(&token, 1, MPI_INT, rank - 1, 0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Send(&token, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



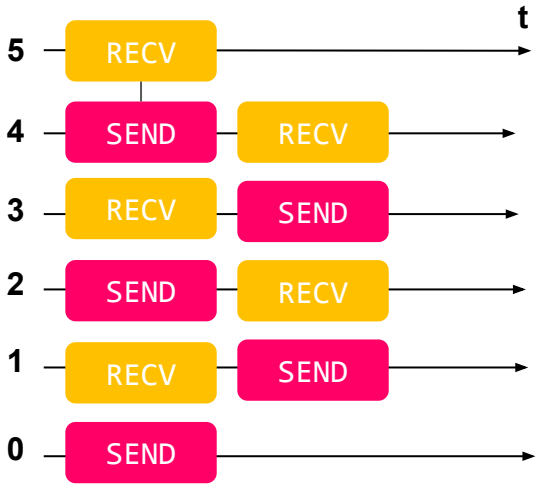
Scaling efficiency is limited by the waiting time inside MPI calls

Dependencies

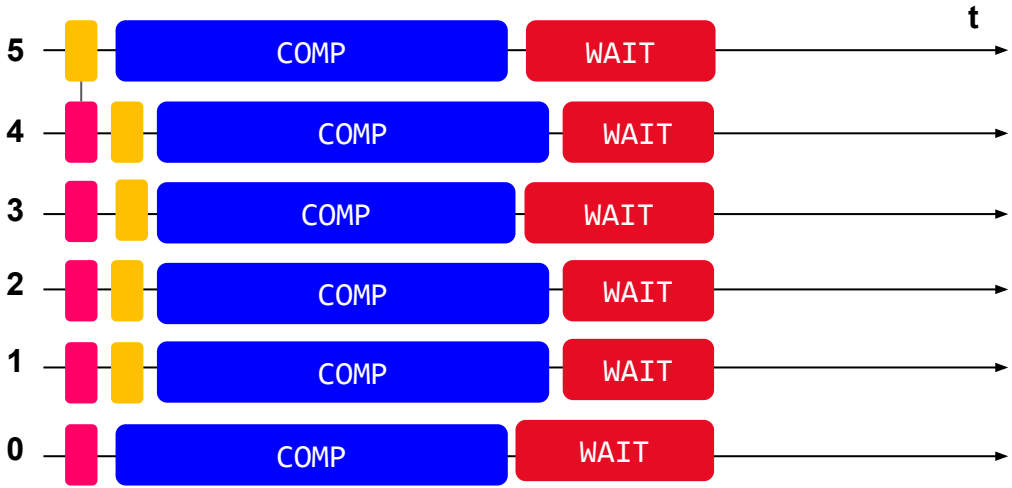


Dependencies

Consider using more efficient patterns or non-blocking operations



Overlap with GPU computing and or data movements



POP metrics for MPI

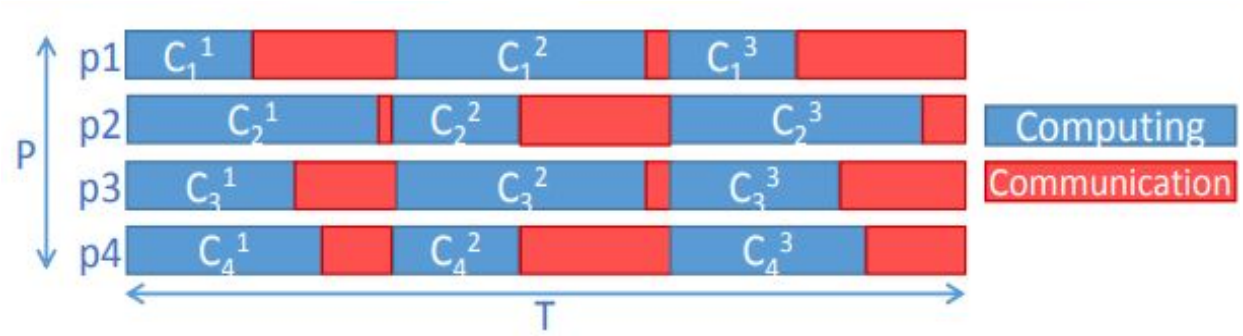
How to measure performance of an MPI application?

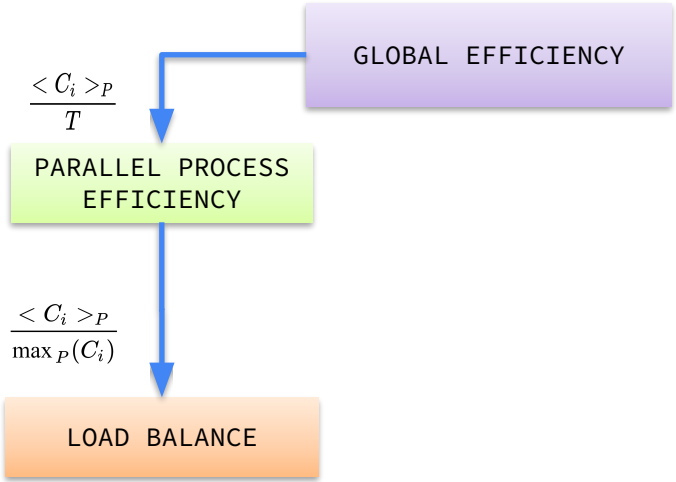
https://pop-coe.eu/

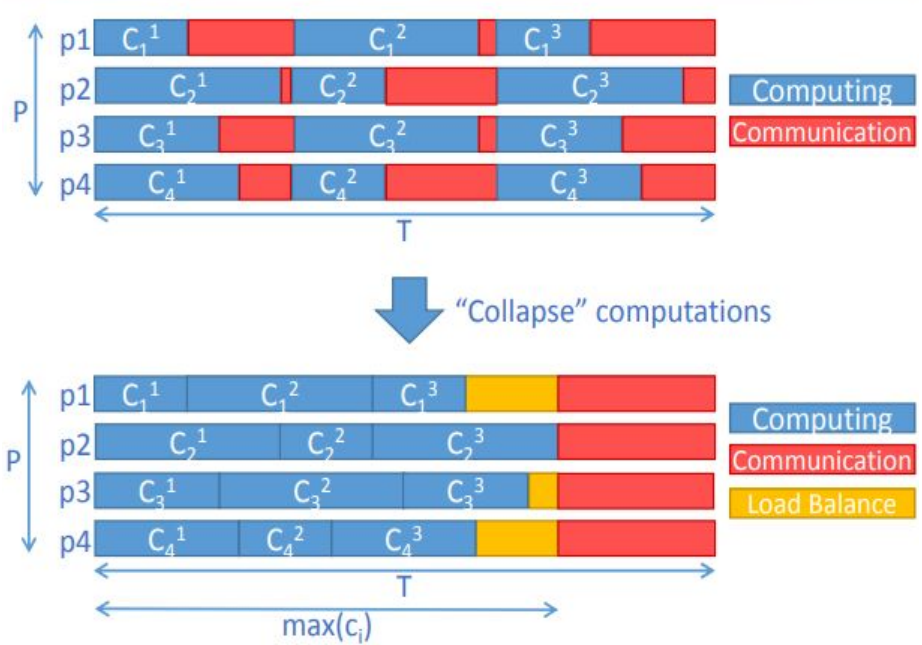
GLOBAL EFFICIENCY

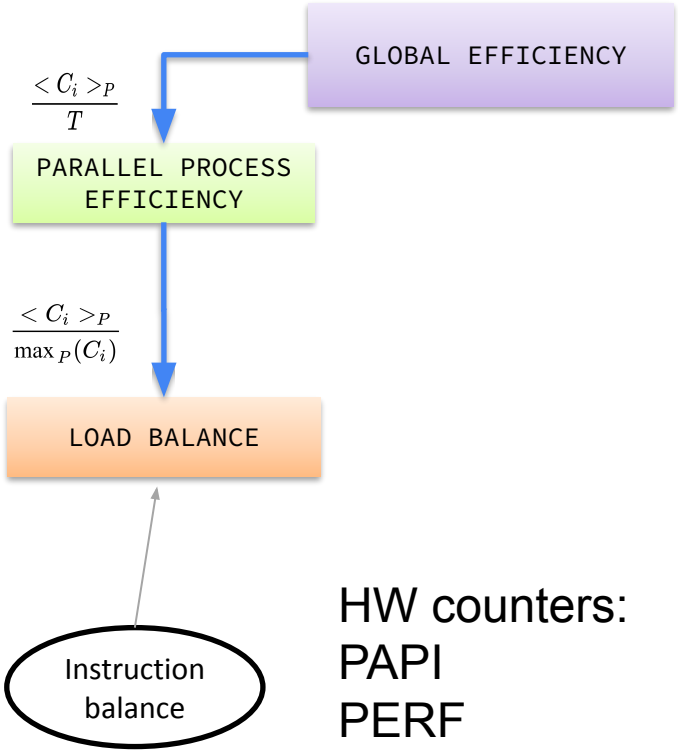
$\frac{< C_i >_P}{T}$

PARALLEL PROCESS EFFICIENCY

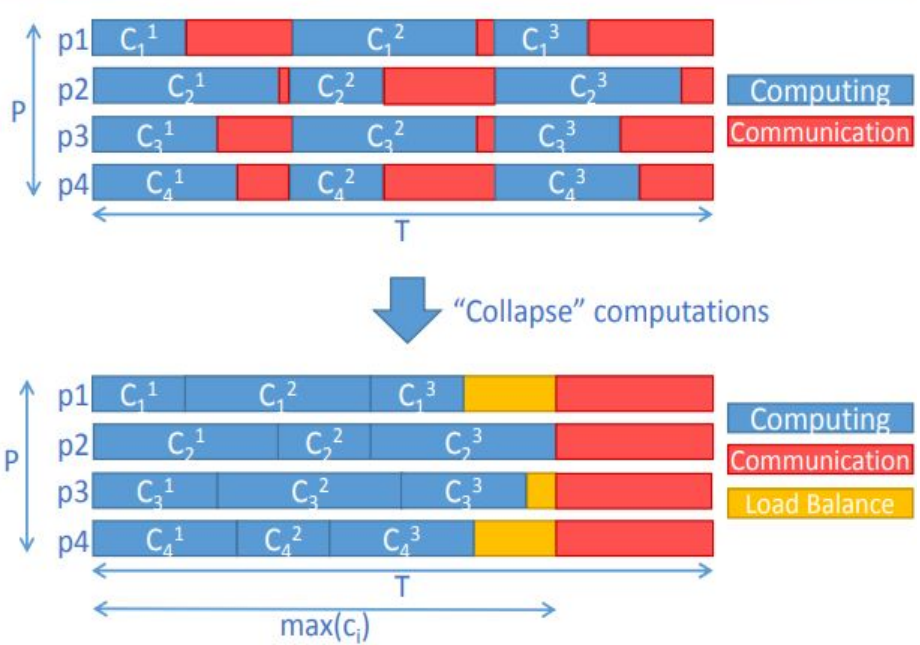


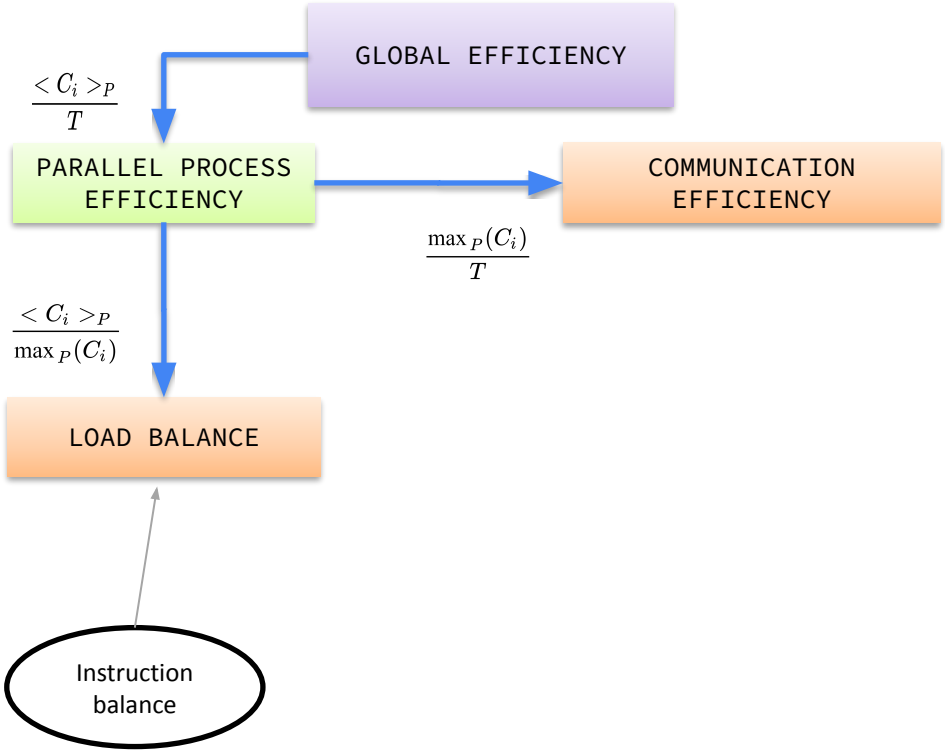

GLOBAL EFFICIENCY
$\frac{< C_i >_P}{T}$
PARALLEL PROCESS EFFICIENCY
$\frac{< C_i >_P}{\max_P(C_i)}$
LOAD BALANCE

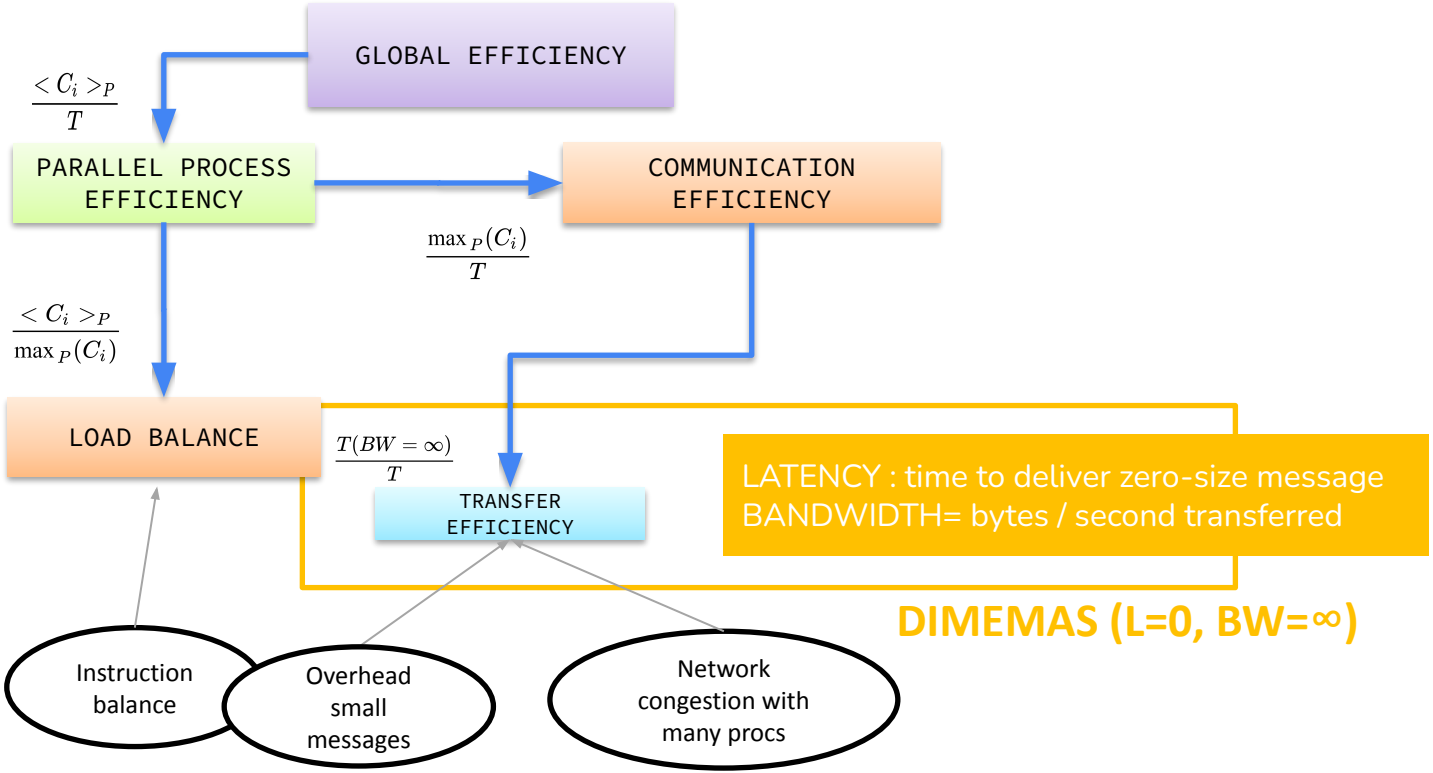

P
p1
p2
p3
p4
C_1^1 C_1^2 C_1^3
C_2^1 C_2^2 C_2^3
C_3^1 C_3^2 C_3^3
C_4^1 C_4^2 C_4^3
T
Computing
Communication
"Collapse" computations
Load Balance
max(c_i)

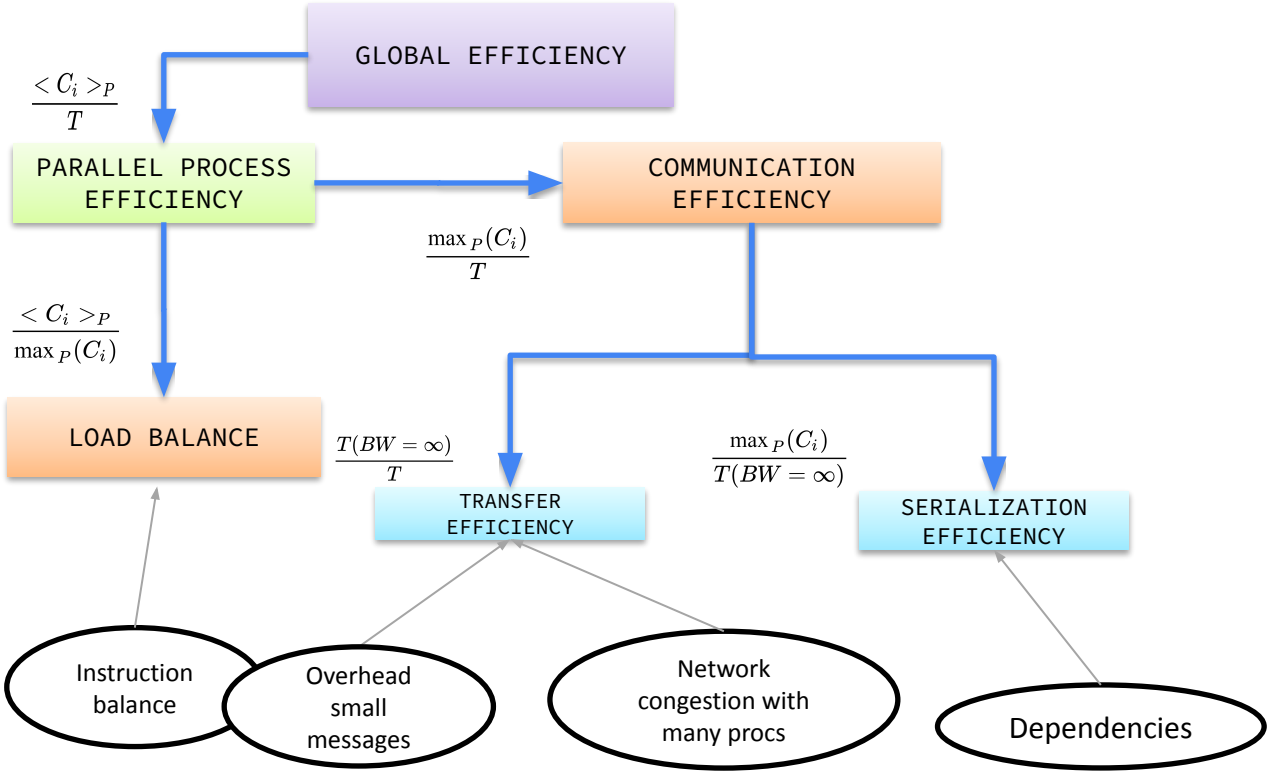


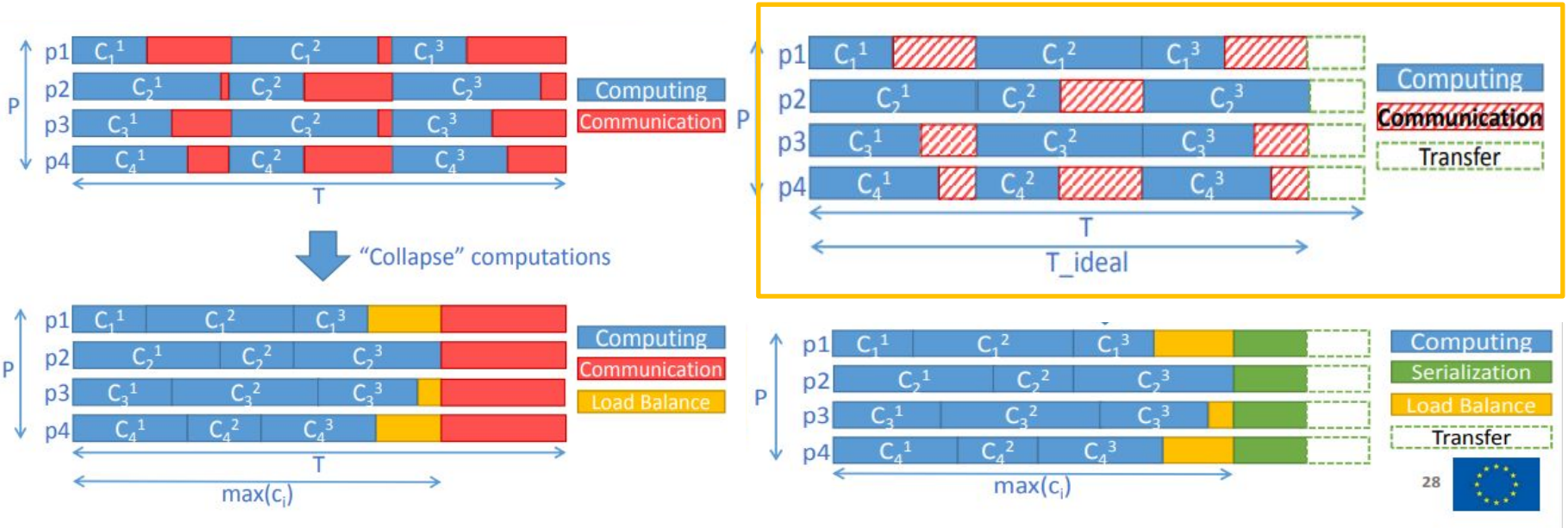
HW counters:
PAPI
PERF
LIKWID

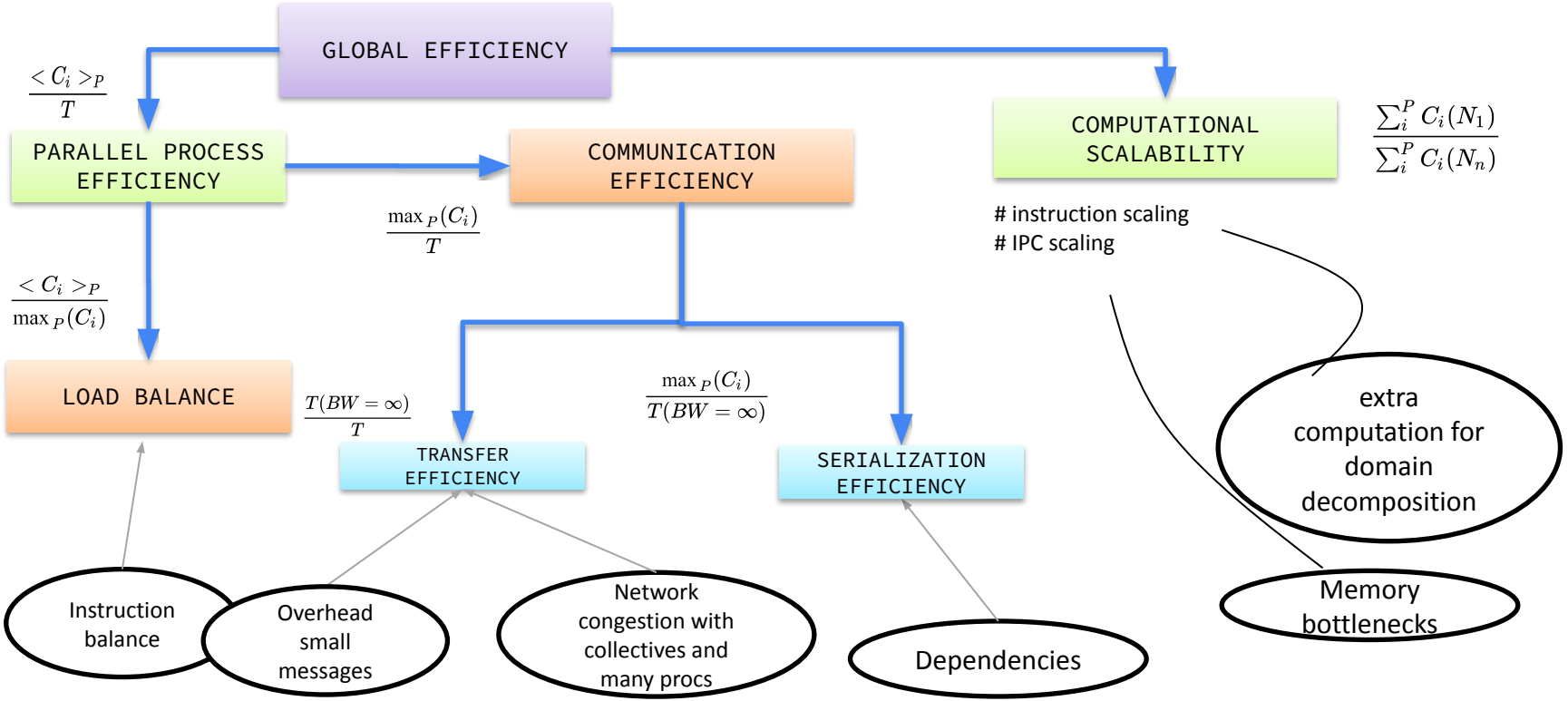



GLOBAL EFFICIENCY
$\frac{< C_i >_P}{T}$
PARALLEL PROCESS EFFICIENCY
COMMUNICATION EFFICIENCY
$\frac{\max_P(C_i)}{T}$
$\frac{< C_i >_P}{\max_P(C_i)}$
LOAD BALANCE
Instruction balance


GLOBAL EFFICIENCY
$\frac{< C_i >_P}{T}$
PARALLEL PROCESS EFFICIENCY
COMMUNICATION EFFICIENCY
$\frac{\max_P(C_i)}{T}$
$\frac{< C_i >_P}{\max_P(C_i)}$
LOAD BALANCE
$\frac{T(BW=\infty)}{T}$
TRANSFER EFFICIENCY
LATENCY : time to deliver zero-size message
BANDWIDTH= bytes / second transferred
DIMEMAS (L=0, BW=∞)
Instruction balance
Overhead small messages
Network congestion with many procs


GLOBAL EFFICIENCY
$\frac{< C_i >_P}{T}$
PARALLEL PROCESS EFFICIENCY
COMMUNICATION EFFICIENCY
$\frac{\max_P(C_i)}{T}$
$\frac{< C_i >_P}{\max_P(C_i)}$
LOAD BALANCE
$\frac{T(BW=\infty)}{T}$
$\frac{\max_P(C_i)}{T(BW=\infty)}$
TRANSFER EFFICIENCY
SERIALIZATION EFFICIENCY
Instruction balance
Overhead small messages
Network congestion with many procs
Dependencies


p1
p2
p3
p4
P
T
Computing
Communication
"Collapse" computations
max(c_i)
T_ideal
Transfer
Serialization
Load Balance


GLOBAL EFFICIENCY
$\frac{< C_i >_P}{T}$
PARALLEL PROCESS EFFICIENCY
COMMUNICATION EFFICIENCY
COMPUTATIONAL SCALABILITY
$\frac{\sum_i^P C_i(N_1)}{\sum_i^P C_i(N_n)}$
instruction scaling
IPC scaling
$\frac{\max_P(C_i)}{T}$
$\frac{< C_i >_P}{\max_P(C_i)}$
LOAD BALANCE
$\frac{T(BW=\infty)}{T}$
$\frac{\max_P(C_i)}{T(BW=\infty)}$
TRANSFER EFFICIENCY
SERIALIZATION EFFICIENCY
extra computation for domain decomposition
Memory bottlenecks
Instruction balance
Overhead small messages
Network congestion with collectives and many procs
Dependencies

POP metrics AT SCALE

! See how these number change with the number of MPI ranks to identify what limits mostly efficiency in scaling

Number of cores	48	96	192	384	768
Global Efficiency	0.93	0.94	0.93	0.84	0.76
↳ Parallel Efficiency	0.93	0.91	0.87	0.77	0.68
↳ Load balance	0.99	0.98	0.98	0.97	0.95
↳ Communication Efficiency	0.94	0.92	0.89	0.79	0.72
↳ Serialisation	0.95	0.94	0.92	0.85	0.81
↳ Transfer efficiency	0.99	0.99	0.97	0.94	0.89
↳ Computational Scaling	1.00	1.03	1.07	1.09	1.12
↳ Instruction Scaling	1.00	0.99	0.97	0.95	0.92
↳ IPC Scaling	1.00	1.05	1.10	1.18	1.27

! Score-P plugin for this

POP metrics for OpenMP

Which could be the main causes of bottlenecks for an OpenMP application?

POP metrics for OpenMP

Which could be the main causes of bottlenecks for an OpenMP application?

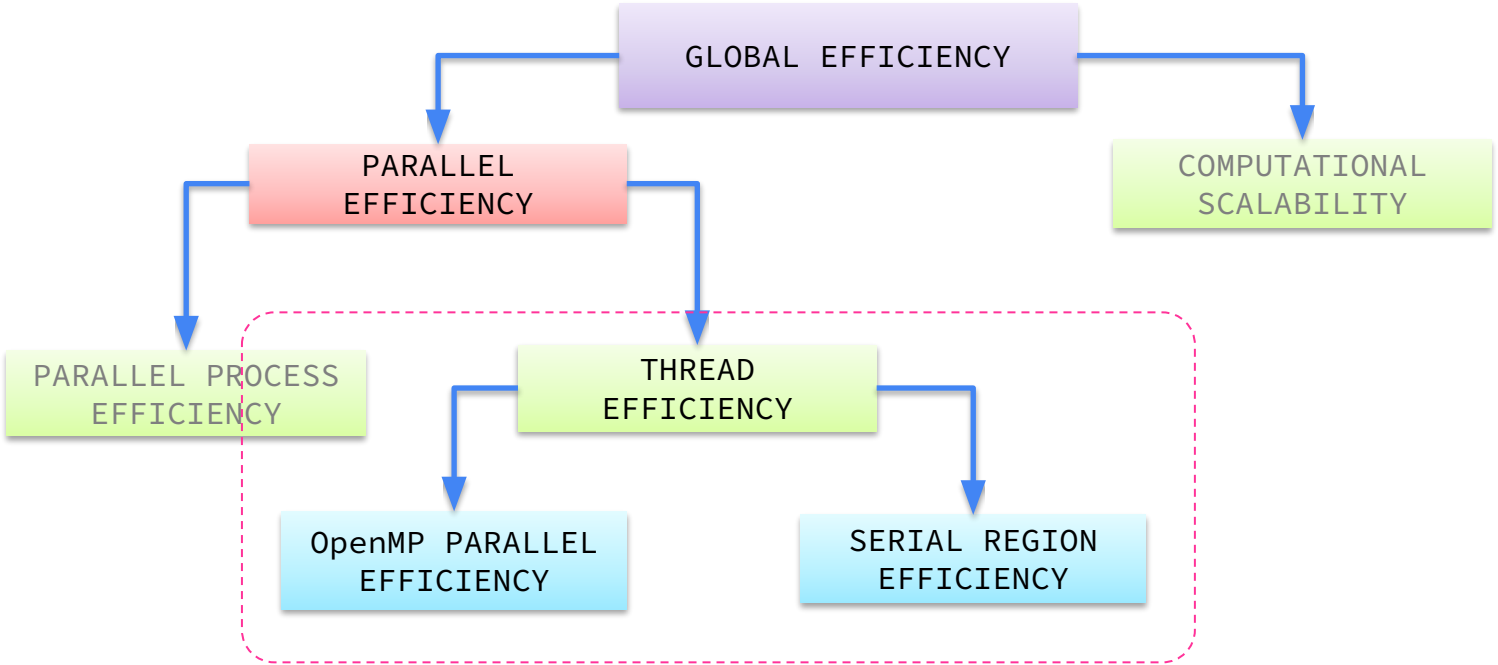
Serialization:

- time spent in computation outside OpenMP

  Are there costly serial region which might exploit OpenMP?

OpenMP parallel efficiency

- Load balance - *Improve scheduling, avoid if*
- Overhead of forking - *Extend parallel regions across multiple loops*
- Synchronizations
- Reduction, Atomic



https://pop-coe.eu/sites/default/files/pop_files/pop_hybrid_metrics_additive_handout.pdf

Profiling softwares
Scalasca and Score-P

l.bellentani@cineca.it
Tutorial on Profiling @ MHPC - ICTP

Scalasca

- *Scalable performance analysis of large scale applications*, developed at Jülich Supercomputing Centre.
- Supports analysis at large scale by exploiting available processors and memory
- Ideal tool to investigate synchronization issues and workload imbalance in communication-intensive applications with thousands of cores.
- **MPI, OpenMP**, user-defined routine instrumentation
- Supports HWC metrics via PAPI.

→ **Score-P** for source-code instrumentation

→ **CUBE** / **Vampir** for profile and traces visualization (but ITAC works as well!)

scalasca

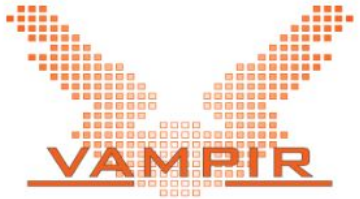
https://dl.acm.org/doi/abs/10.1016/j.parco.2009.02.003

https://dl.acm.org/doi/abs/10.1016/j.parco.2009.02.003

SCALABLE

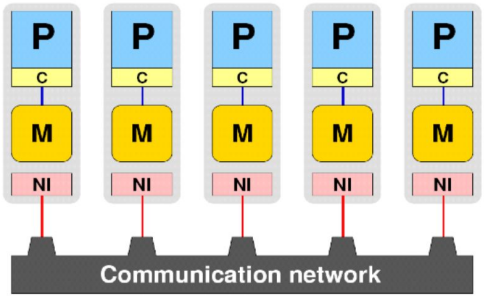
PERFORMANCE ANALYSIS

OF LARGE SCALE APPLICATIONS

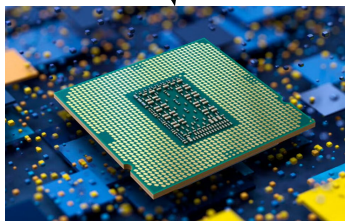


https://dl.acm.org/doi/abs/10.1016/j.parco.2009.02.003

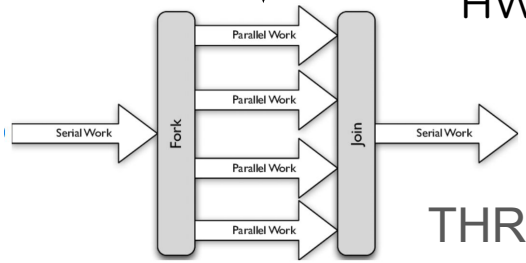
scalasca



MPI



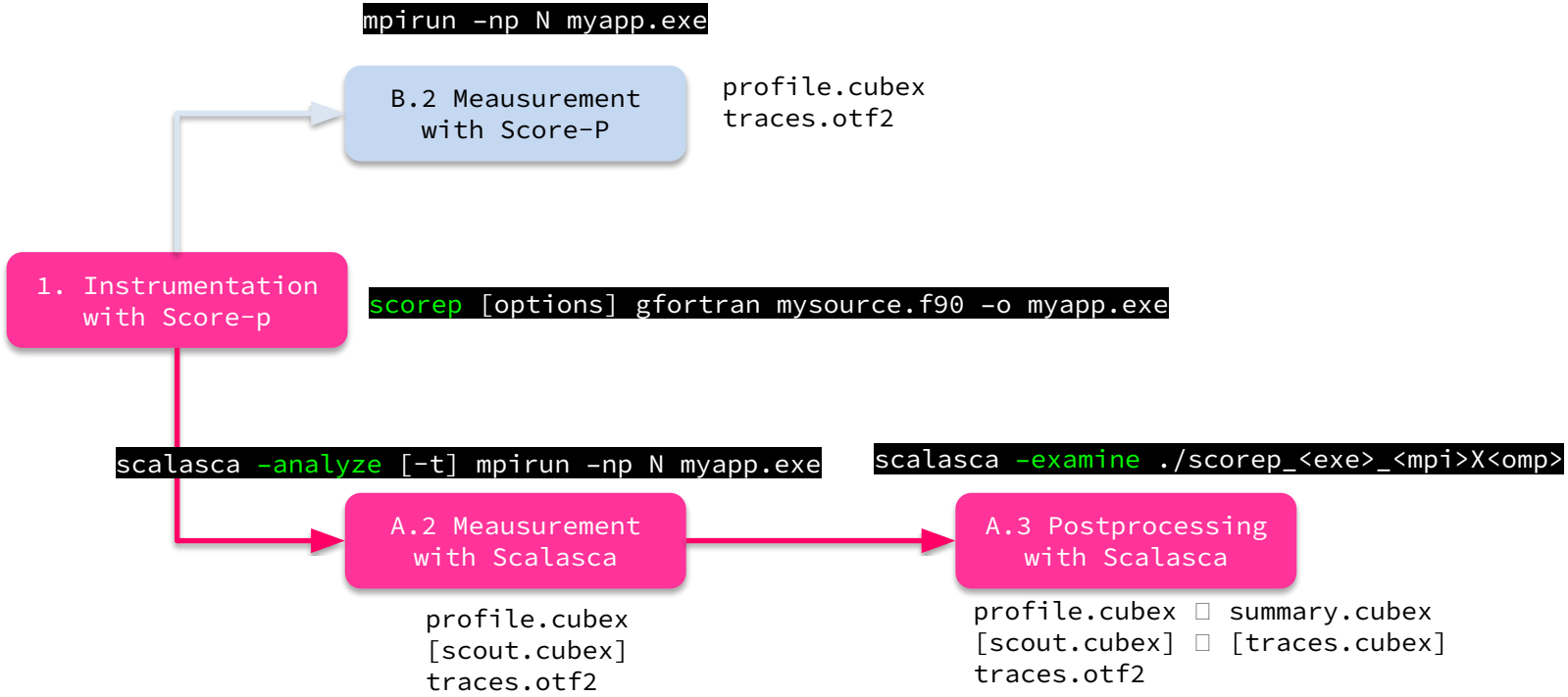
HW COUNTERS



THREADS

USER

```
subroutine example(x, y)
use e_fct
real, dimension(100) :: x, y
where (log_sqr(x) < 0.5)
  y = x * x
end where
if (any(log_sqr(x) > 10.)) then
  print *, '... something ...'
endif
end program
```


mpirun -np N myapp.exe
B.2 Meausurement with Score-P
profile.cubex
traces.otf2
1. Instrumentation with Score-p
scorep [options] gfortran mysource.f90 -o myapp.exe
scalasca -analyze [-t] mpirun -np N myapp.exe
scalasca -examine ./scorep_<exe>_<mpi>X<omp>
A.2 Meausurement with Scalasca
A.3 Postprocessing with Scalasca
profile.cubex
[scout.cubex]
traces.otf2
profile.cubex □ summary.cubex
[scout.cubex] □ [traces.cubex]
traces.otf2

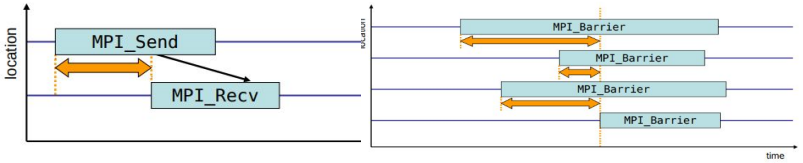
MPI APIs | OPENMP | USR ROUTINES

MAIN METRICS

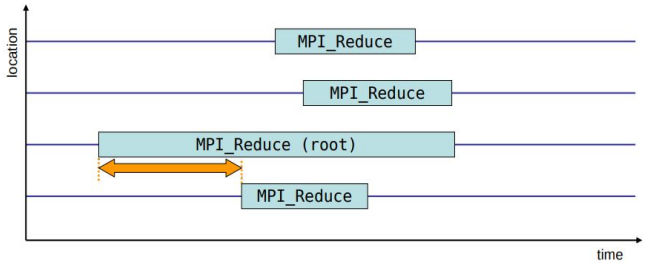
- Time
  - Computation
  - MPI
    - › Synchronization
    - › Management
    - › Communications
    - › MPI I/O
  - OpenMP
- Counts
- Bytes transferred : PtoP vs collective
- MPI io bytes
- + Hardware counters [PAPI]
- + timestamps, locations, communicators (traces)

DERIVED METRICS

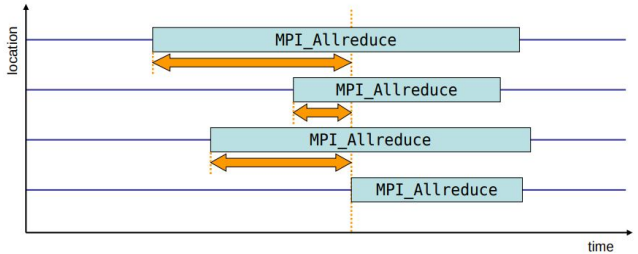
- MPI and Thread efficiency of POP assessment (plugin)
- Workload balance and synchronization
  - Computational imbalance
  - Idle threads
  - Late receiver, wait at barrier ...

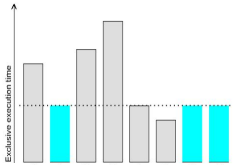


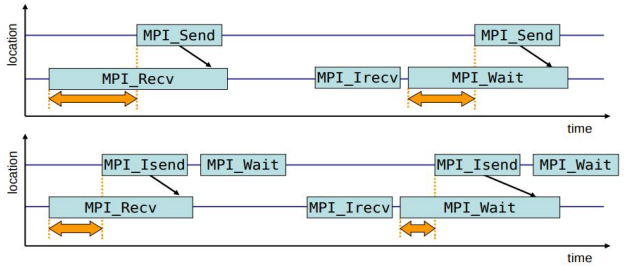
LATE RECEIVER WAIT AT BARRIER

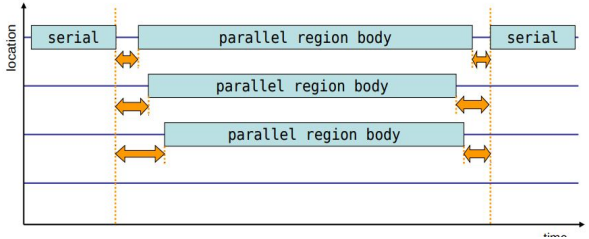


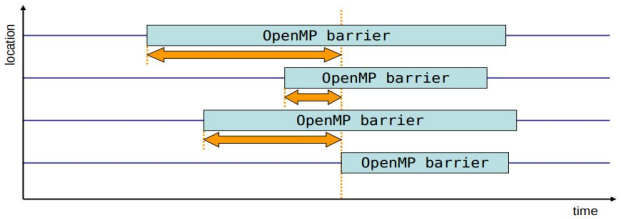
https://www.vi-hps.org/cms/upload/material/tw-original/vi-hps-tw-original-Scalasca_Patterns.pdf

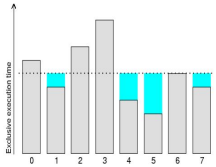




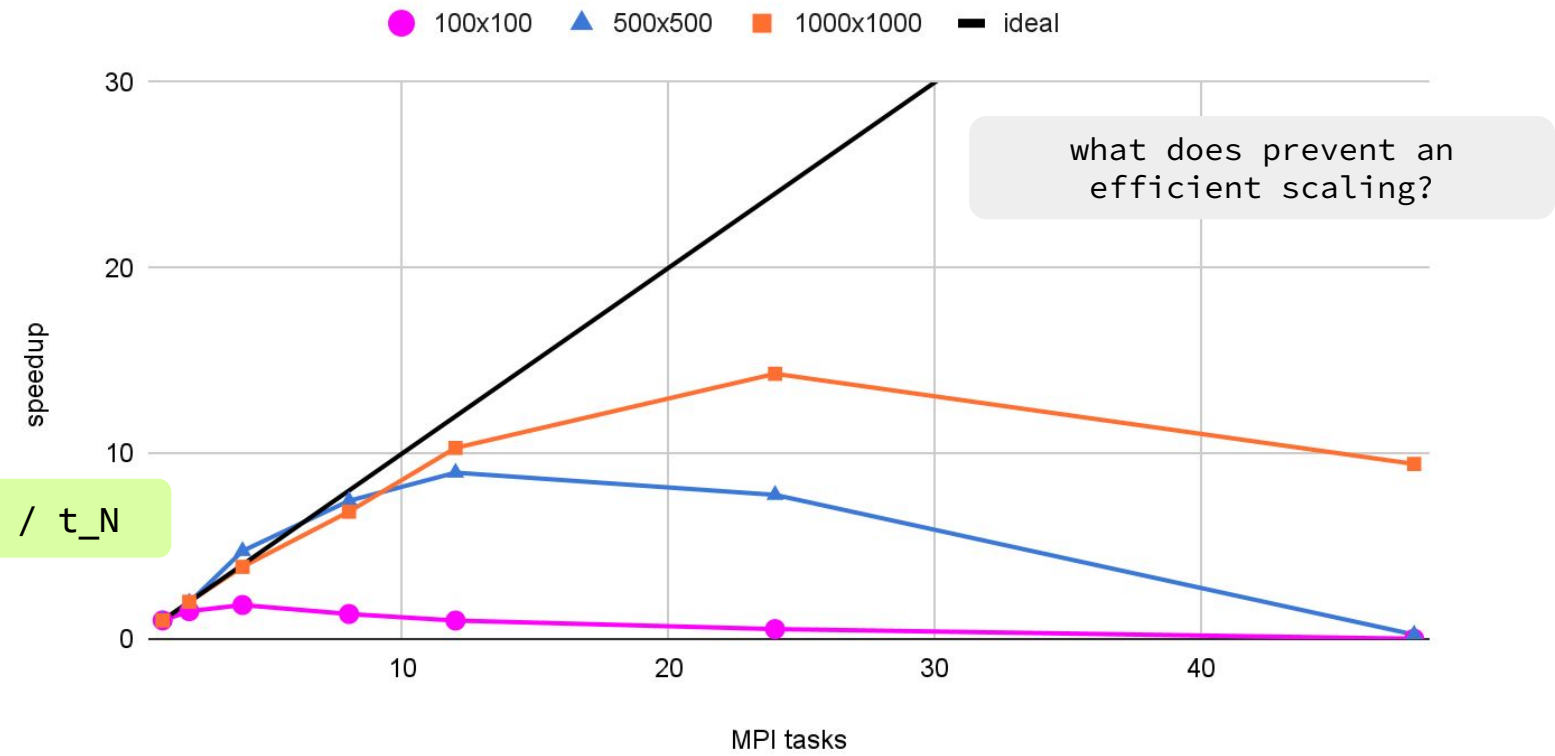








JACOBI EXAMPLE



what does prevent an efficient scaling?

t_1 / t_N

Scalasca

✳ INSTRUMENTATION : add prefix at compile and linking stage

```
scorep [--mpp=mpi --user --openmp --verbose ...] mpif90 source.f90 -o myapp.exe
```

- The options define the events you want to instrument
- If using CMake or autotools the commands differ a bit, check the user guide

✳ MEASUREMENT : add prefix to mpirun, -t for traces

```
scalasca -analyze [ -t ]  mpirun -np NP ./myapp.exe
```

- Results in folder “scorep_myapp_<NP>X<M>”
- **-t** option for traces
- Never trace at first!

INSTRUMENTATION AND MEASUREMENT

1. Before tracing, check the memory required according to the events traced with

```
scorep-score scorep_cfd_14_trace/profile.cubex
```

```
Estimated aggregate size of event trace:                   1181MB
Estimated requirements for largest trace buffer (max_buf): 85MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       87MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=87MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type max_buf[B]     visits time[s] time[%] time/visit[us]  region
         ALL 88,407,491 51,464,585  291.42   100.0           5.66  ALL
         USR 88,128,408 51,408,225   59.01    20.2           1.15  USR
         MPI    254,960     42,290  192.44    66.0        4550.40  MPI
         COM     24,096     14,056   39.63    13.6        2819.40  COM
      SCOREP         51         14    0.35     0.1       24757.11  SCOREP
```

From here you can also infer the amount of time spent in MPI, or USR routines

INSTRUMENTATION AND MEASUREMENT

2. With **-r** option you can also get a flat profile of the events traced (MPI, user)

```
scorep-score -r scorep_cfd_14_trace/profile.cubex
```

```
USR 66,060,288 38,535,168    3.68     1.3          0.10  cfdio_mp_colfunc_
USR 22,020,096 12,845,056    5.09     1.7          0.40  cfdio_mp_hue2rgb_
MPI    188,188     24,024    0.52     0.2         21.47  MPI_Sendrecv
MPI     66,066     14,014    0.81     0.3         57.96  MPI_Allreduce
MPI     60,593      2,028    0.17     0.1         81.59  MPI_Recv
MPI     59,059      2,002    0.02     0.0          8.03  MPI_Send
COM     24,024     14,014    0.07     0.0          5.14  boundary_mp_haloswap_
USR     24,000     14,000   29.97    10.3       2140.95  jacobi_mp_jacobistep_
USR     24,000     14,000   20.26     7.0       1447.21  jacobi_mp_deltasq_
```

These are ordered according to buffer size, helpful to consider events to filter out.

For example here the first routines related to IO at the end of the run require large buffers, and are not interesting for our analysis.

INSTRUMENTATION AND MEASUREMENT

3. Filter events

```
cat filter.file
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
        cfdio_mp_colfunc_
        cfdio_mp_hue2rgb_
SCOREP_REGION_NAMES_END
```

And check the estimated buffer with filter applied

```
scorep-score -f filter.file scorep_cfd_14_trace/profile.cubex

Estimated aggregate size of event trace:                   5MB
Estimated requirements for largest trace buffer (max_buf): 320kB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       4097kB
```

If satisfied, repeat the measurement with **`scalasca analyse -f filter.file`**

POSTPROCESSING AND VISUALIZATION

✳ POSTPROCESSING : derive metrics with Scalasca

```
scalasca -examine  [options] scorep_myapp_NPXM
```

- derives metrics from the fundamental ones
- opens the profile in CUBE

✳ VISUALIZATION : open *.cubex with cubegui

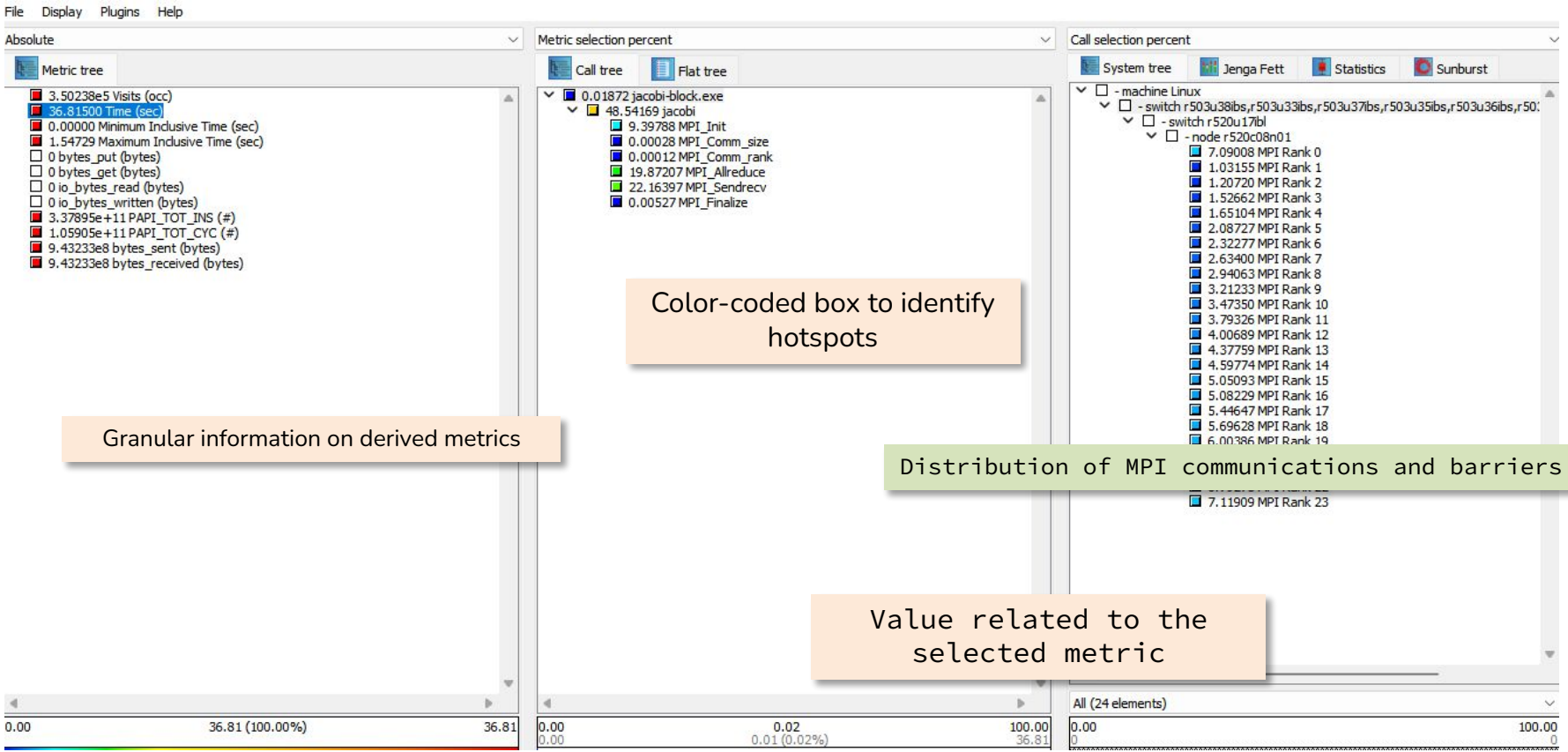
```
cube scorep_myapp_NPXM/summary.cubex
```

There are a number of environment variable to customize your measurement
https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/scorepmeasurementconfig.html

export SCOREP_PAPI_METRIC → PAPI_TOT_INS, PAPI_TOT_CYC, ...

export SCOREP_TOTAL_MEMORY → to increase the buffer memory against buffer flushes

GUI


File
Display
Plugins
Help
Absolute
Metric tree
3.50238e5 Visits (occ)
36.81500 Time (sec)
0.00000 Minimum Inclusive Time (sec)
1.54729 Maximum Inclusive Time (sec)
0 bytes_put (bytes)
0 bytes_get (bytes)
0 io_bytes_read (bytes)
0 io_bytes_written (bytes)
3.37895e+11 PAPI_TOT_INS (#)
1.05905e+11 PAPI_TOT_CYC (#)
9.43233e8 bytes_sent (bytes)
9.43233e8 bytes_received (bytes)
0.00
36.81 (100.00%)
36.81
Metric selection percent
Call tree
Flat tree
0.01872 jacobi-block.exe
48.54169 jacobi
9.39788 MPI_Init
0.00028 MPI_Comm_size
0.00012 MPI_Comm_rank
19.87207 MPI_Allreduce
22.16397 MPI_Sendrecv
0.00527 MPI_Finalize
0.00
0.02
100.00
0.00
0.01 (0.02%)
36.81
Call selection percent
System tree
Jenga Fett
Statistics
Sunburst
- machine Linux
- switch r503u38ibs,r503u33ibs,r503u37ibs,r503u35ibs,r503u36ibs,r50
- switch r520u17ibl
- node r520c08n01
7.09008 MPI Rank 0
1.03155 MPI Rank 1
1.20720 MPI Rank 2
1.52662 MPI Rank 3
1.65104 MPI Rank 4
2.08727 MPI Rank 5
2.32277 MPI Rank 6
2.63400 MPI Rank 7
2.94063 MPI Rank 8
3.21233 MPI Rank 9
3.47350 MPI Rank 10
3.79326 MPI Rank 11
4.00689 MPI Rank 12
4.37759 MPI Rank 13
4.59774 MPI Rank 14
5.05093 MPI Rank 15
5.08229 MPI Rank 16
5.44647 MPI Rank 17
5.69628 MPI Rank 18
7.11909 MPI Rank 23
All (24 elements)
0.00
100.00
0
0
Color-coded box to identify hotspots
Granular information on derived metrics
Distribution of MPI communications and barriers
Value related to the selected metric

MANUAL INSTRUMENTATION

- Score-P automatically instruments the user-defined routines in the source code
- Additional instrumentation is possible to label source code regions as with a name

1. Include the scorep file with definitions after “implicit none”

```
#include "scorep/SCOREP_User.inc"
```

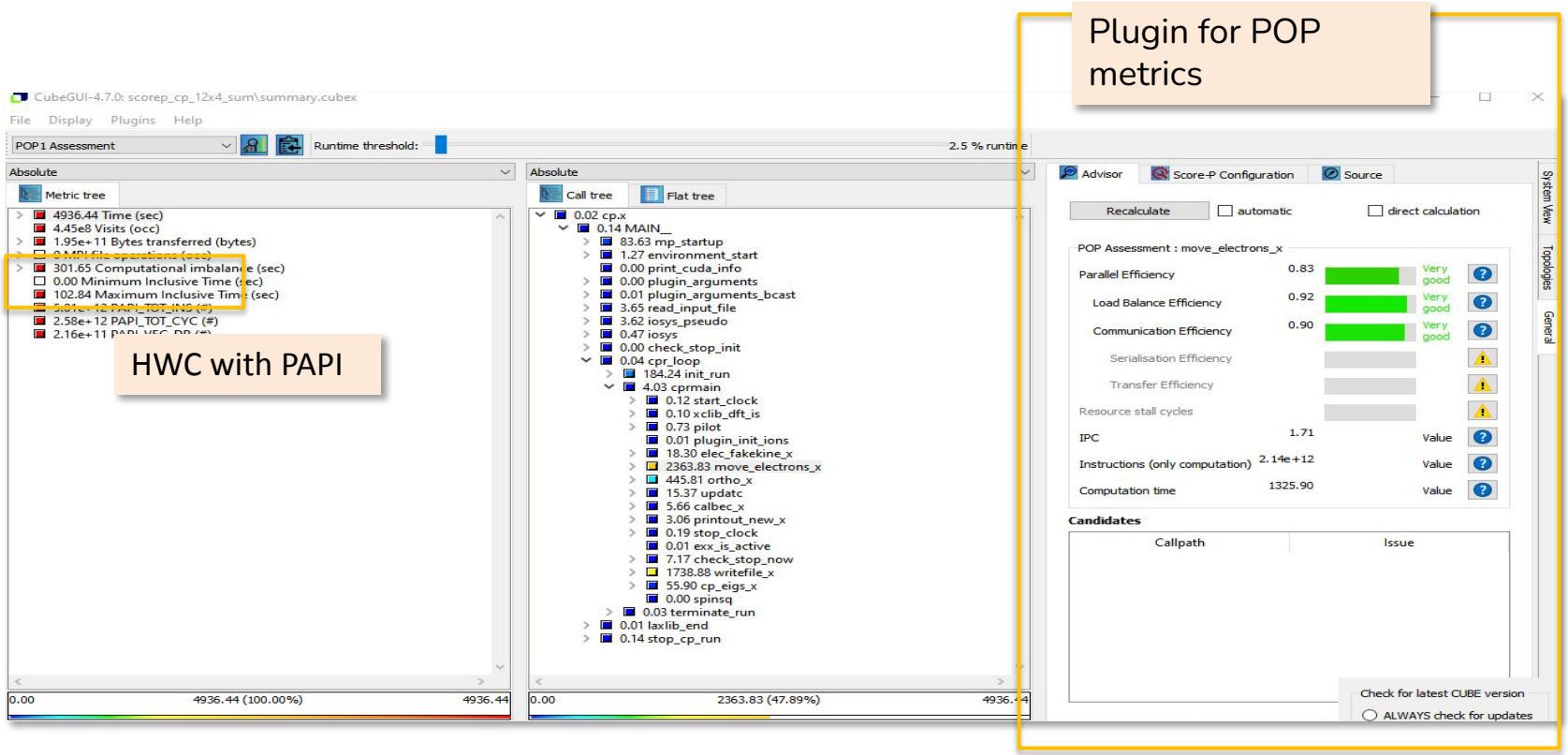
2. Initialize an handle

```
SCOREP_USER_REGION_DEFINE( myhandle1 )
```

3. Add APIs to start and stop the regions

```
#start the region to name as “block”
SCOREP_USER_REGION_BEGIN( myhandle, "block", SCOREP_USER_REGION_TYPE_COMMON )
...
#stop the region to name as “block”
SCOREP_USER_REGION_END( myhandle )
```

GUI

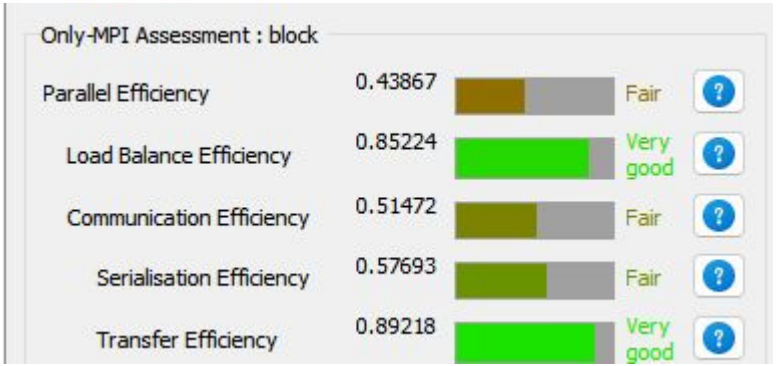

Plugin for POP metrics
CubeGUI-4.7.0: scorep_cp_12x4_sum\summary.cubex
File Display Plugins Help
POP1 Assessment
Runtime threshold:
2.5 % runtime
Absolute
Metric tree
4936.44 Time (sec)
4.45e8 Visits (occ)
1.95e+11 Bytes transferred (bytes)
301.65 Computational imbalance (sec)
0.00 Minimum Inclusive Time (sec)
102.84 Maximum Inclusive Time (sec)
2.58e+12 PAPI_TOT_CYC (#)
HWC with PAPI
0.00
4936.44 (100.00%)
4936.44
Absolute
Call tree
Flat tree
0.02 cp.x
0.14 MAIN__
83.63 mp_startup
1.27 environment_start
0.00 print_cuda_info
0.00 plugin_arguments
0.01 plugin_arguments_bcast
3.65 read_input_file
3.62 iosys_pseudo
0.47 iosys
0.00 check_stop_init
0.04 cpr_loop
184.24 init_run
4.03 cprmain
0.12 start_clock
0.10 xclib_dft_is
0.73 pilot
0.01 plugin_init_ions
18.30 elec_fakekine_x
2363.83 move_electrons_x
445.81 ortho_x
15.37 updatc
5.66 calbec_x
3.06 printout_new_x
0.19 stop_clock
0.01 exx_is_active
7.17 check_stop_now
1738.88 writefile_x
55.90 cp_eigs_x
0.00 spinsq
0.03 terminate_run
0.01 laxlib_end
0.14 stop_cp_run
0.00
2363.83 (47.89%)
Advisor
Score-P Configuration
Source
Recalculate
automatic
direct calculation
POP Assessment : move_electrons_x
Parallel Efficiency 0.83 Very good
Load Balance Efficiency 0.92 Very good
Communication Efficiency 0.90 Very good
Serialisation Efficiency
Transfer Efficiency
Resource stall cycles
IPC 1.71 Value
Instructions (only computation) 2.14e+12 Value
Computation time 1325.90 Value
Candidates
Callpath
Issue
System View
Topologies
General
Check for latest CUBE version
ALWAYS check for updates

POP ASSESSMENT PLUGIN

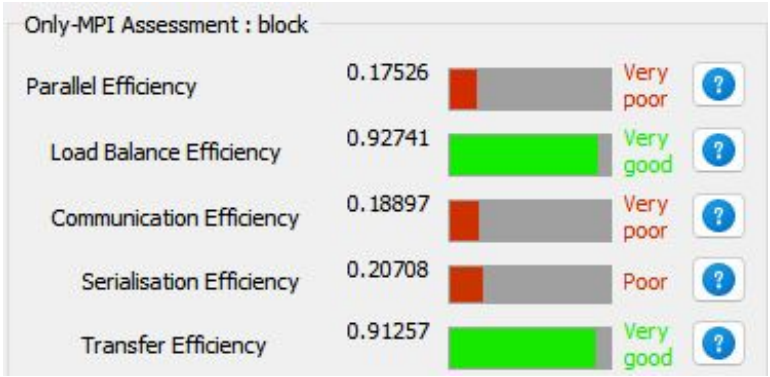
try different algorithm?

4

12



24



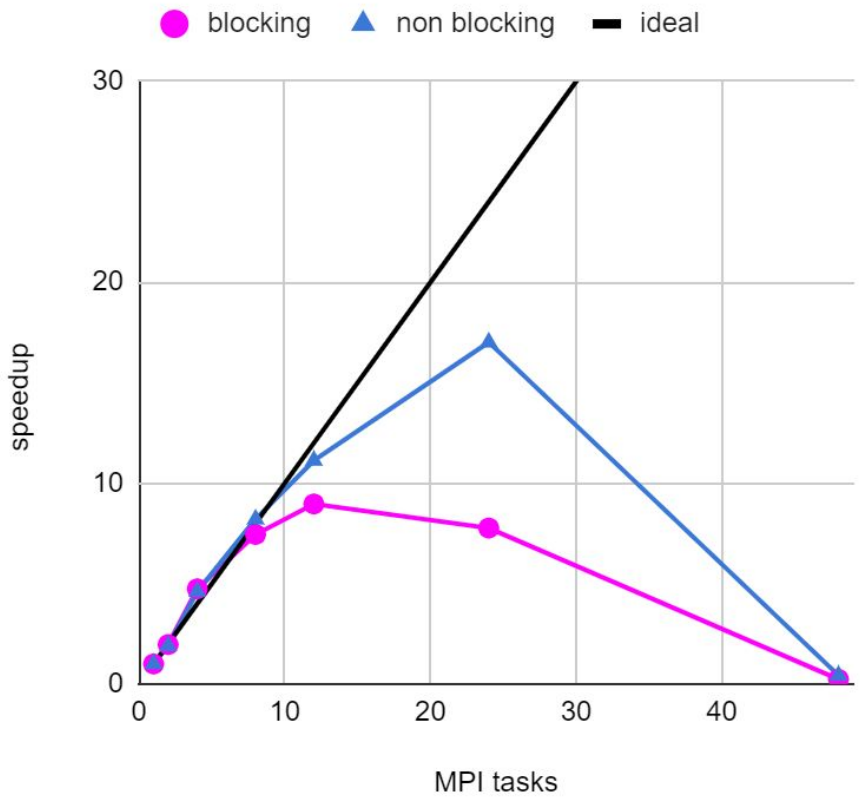
48

JACOBI EXAMPLE

try a non-blocking communication

```
[lbellen1@login02 2-itac]$ diff isendirecv/Fortran/jacobi-mpi-sendrecv.f90 sendrecv/Fortran/jacobi-mpi-sendrecv.f90
101,104c101,104
<         call mpi_isend(grid(1,1), ny, MPI_DOUBLE, myrank-1, 0, MPI_COMM_WORLD,requests(1),  ierr)
<         call mpi_irecv(grid(1,0), ny, MPI_DOUBLE, myrank-1, 0, MPI_COMM_WORLD, requests(2), ierr)
<         !call mpi_sendrecv(grid(1,1),ny,MPI_DOUBLE,myrank-1,0,grid(1,0),ny,&
<         !                  MPI_DOUBLE,myrank-1,0,MPI_COMM_WORLD,status,ierr)
---
>         !call mpi_isend(grid(1,1), ny, MPI_DOUBLE, myrank-1, 0, MPI_COMM_WORLD,requests(1),  ierr)
>         !call mpi_irecv(grid(1,0), ny, MPI_DOUBLE, myrank-1, 0, MPI_COMM_WORLD, requests(2), ierr)
>         call mpi_sendrecv(grid(1,1),ny,MPI_DOUBLE,myrank-1,0,grid(1,0),ny,&
>                           MPI_DOUBLE,myrank-1,0,MPI_COMM_WORLD,status,ierr)
111,114c111,114
<         call mpi_isend(grid(1,local_nx), ny, MPI_DOUBLE, myrank+1, 0, MPI_COMM_WORLD, requests(3),  ierr)
<         call mpi_irecv(grid(1,local_nx+1), ny, MPI_DOUBLE, myrank+1, 0, MPI_COMM_WORLD,requests(4),  ierr)
<         !call mpi_sendrecv(grid(1,local_nx), ny, MPI_DOUBLE, myrank+1, 0,grid(1,local_nx+1),&
<         !            ny, MPI_DOUBLE, myrank+1, 0, MPI_COMM_WORLD, status,ierr)
---
>         !call mpi_isend(grid(1,local_nx), ny, MPI_DOUBLE, myrank+1, 0, MPI_COMM_WORLD, requests(3),  ierr)
>         !call mpi_irecv(grid(1,local_nx+1), ny, MPI_DOUBLE, myrank+1, 0, MPI_COMM_WORLD,requests(4),  ierr)
>         call mpi_sendrecv(grid(1,local_nx), ny, MPI_DOUBLE, myrank+1, 0,grid(1,local_nx+1),&
>                     ny, MPI_DOUBLE, myrank+1, 0, MPI_COMM_WORLD, status,ierr)
117c117
<      call mpi_waitall(4, requests, MPI_STATUSES_IGNORE, ierr)
---
> !     call mpi_waitall(4, requests, MPI_STATUSES_IGNORE, ierr)
```

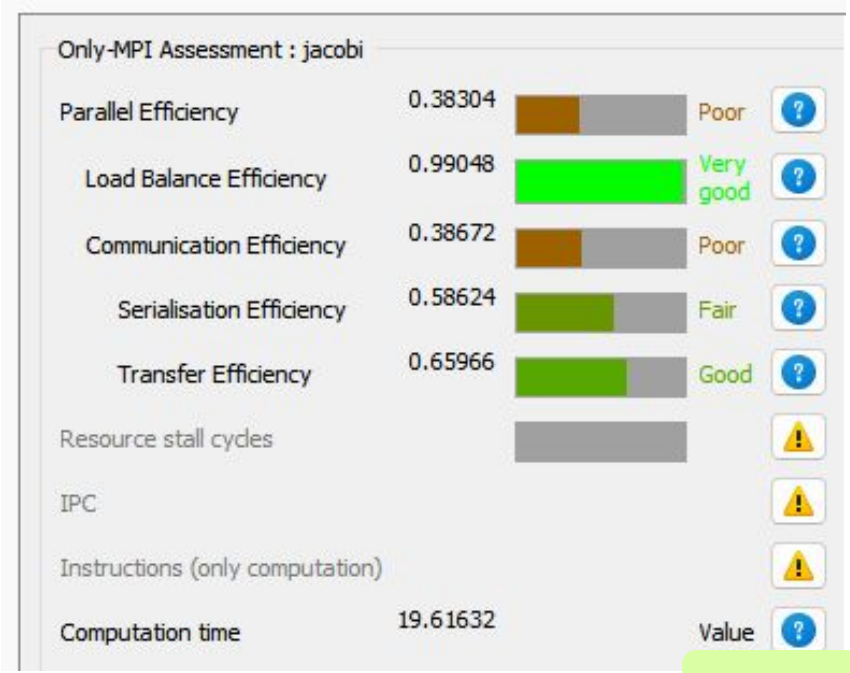
JACOBI EXAMPLE



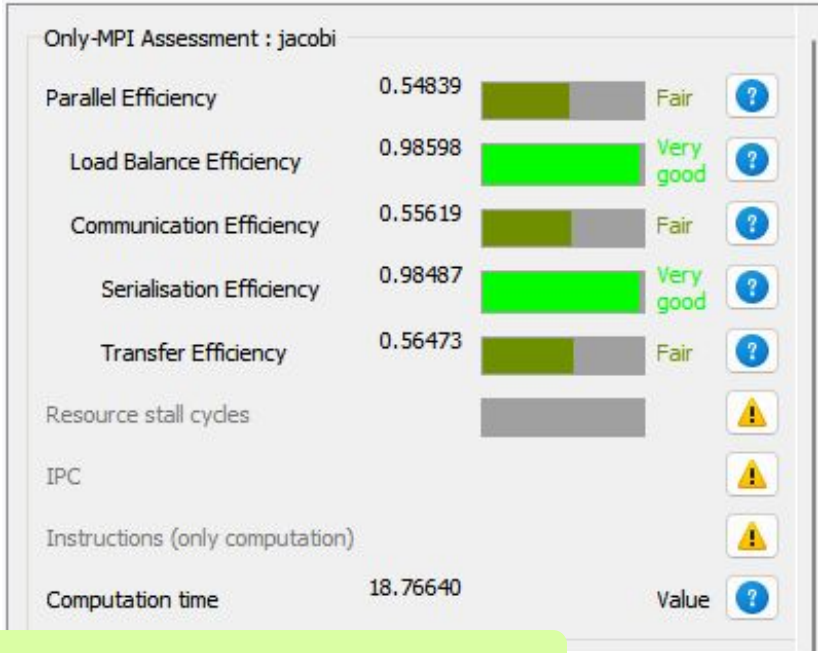
non blocking improves scalability

POP ASSESSMENT PLUGIN

blocking



non blocking



non blocking reduces dependencies

ITAC

Source-code instrumentation providing summary and **Trace** Collector and Analyzer for communication hotspots

- **MPI** and **OpenMP** time and imbalance
- **USR routines** instrumentation
- **Ideal traces** to measure interconnect vs waiting time

```
module load intel-oneapi-mpi intel-oneapi-compilers intel-oneapi-itac
mpiifort -trace [-tcollect] mycode.f90
srun ./mycode → *.stf
traceanalyzer *.stf
```

ITAC

Summary: jacobi-mpi-instr.exe.stf

Total time: **8** sec. Resources: **24** 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 - 2.01 sec	25.1 %
OpenMP - 0 sec	0 %
MPI calls - 5.99 sec	74.8 %

☐ *Exclude from total time*

Top MPI functions

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

MOST TIME-CONSUMING MPI CALLS

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:

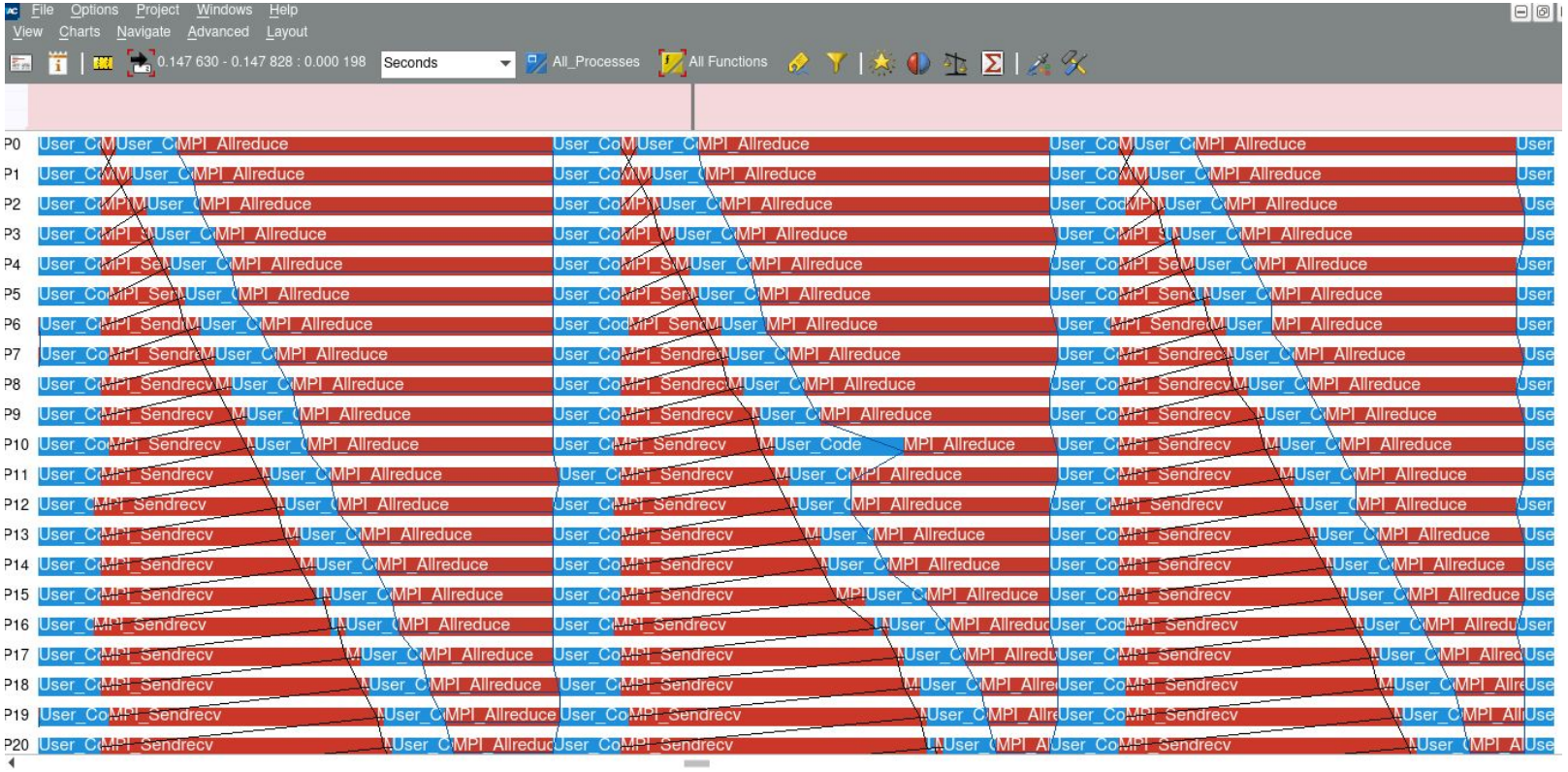
- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use:
Intel® VTune™ Profiler for:
- algorithmic level tuning with hpc-performance and threading efficiency analysis;
- microarchitecture level tuning with general exploration and bandwidth analysis;

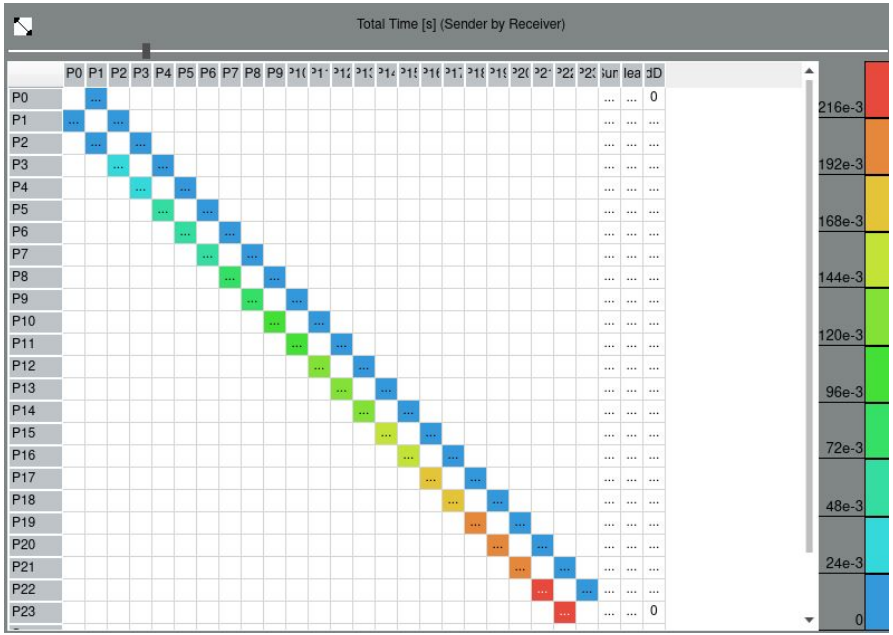
Intel® Advisor for:
- vectorization optimization and thread prototyping.

For more information, see documentation for the respective tool:
Analyzing MPI applications with Intel® VTune™ Profiler
Analyzing MPI applications with Intel® Advisor

ITAC - TRACE VISUALIZATION



ITAC - MESSAGE PROFILE

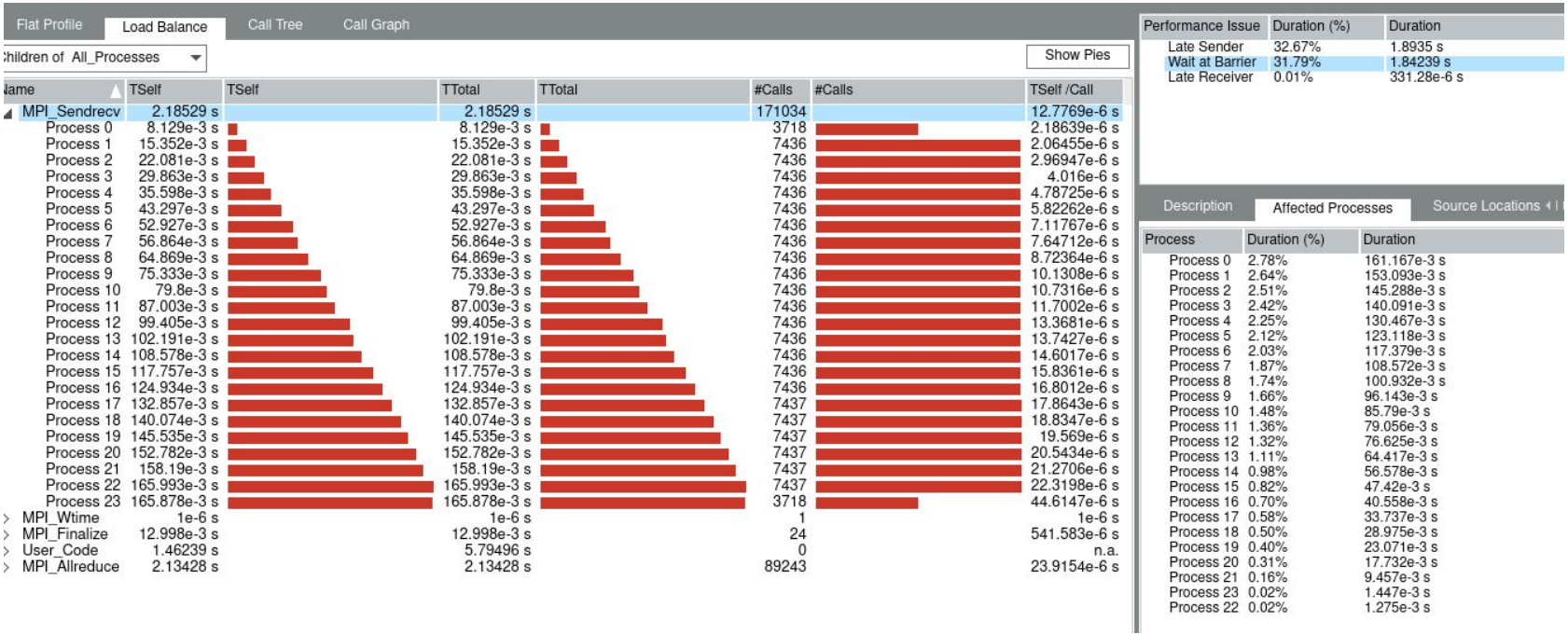


TIME

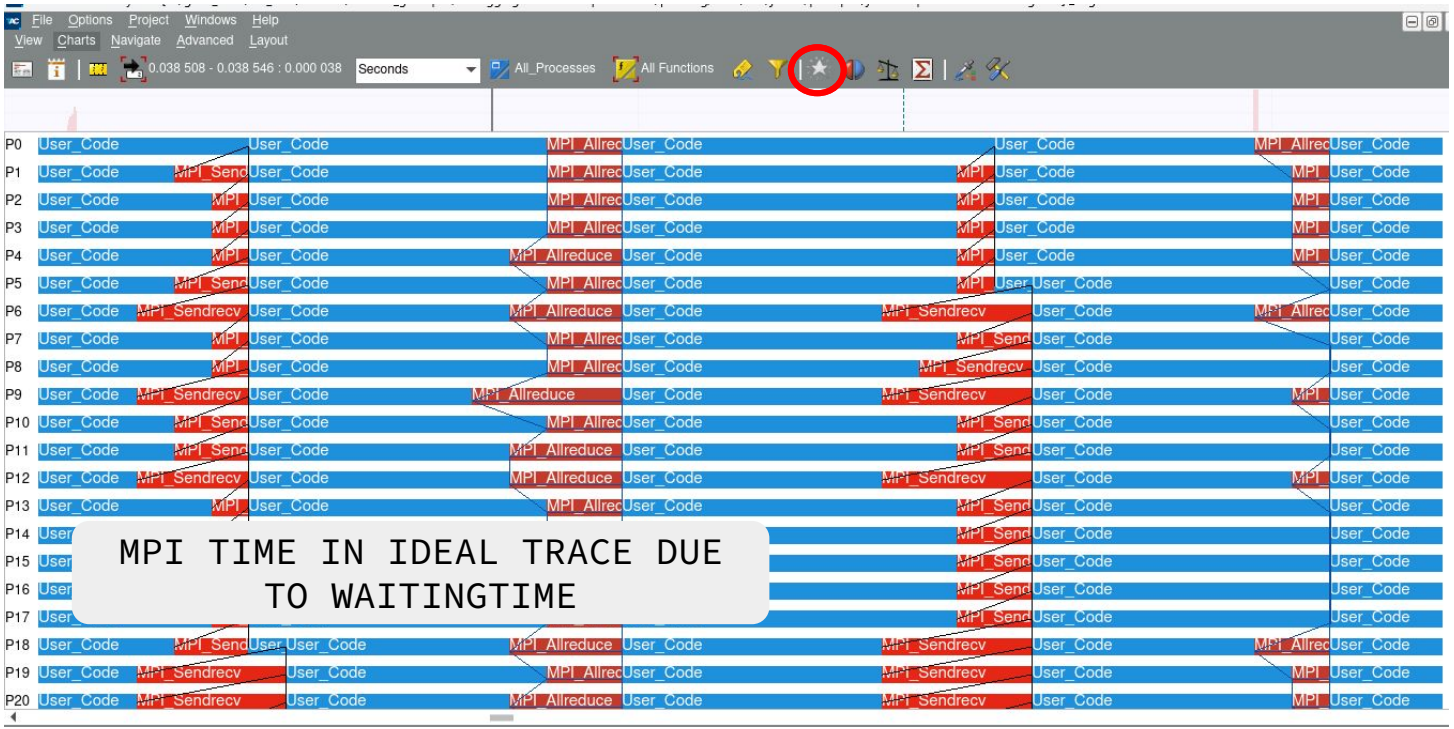
MESSAGE VOLUME

AVERAGE TRANSFER TIME

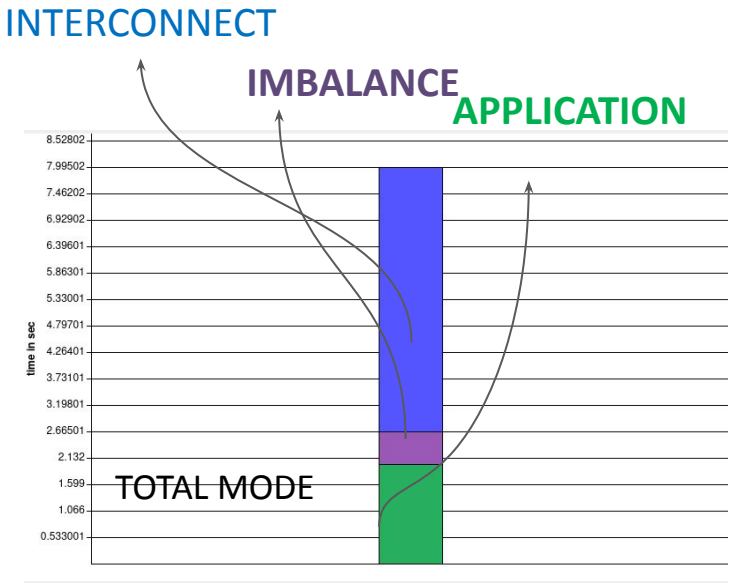
ITAC - FUNCTION PROFILE



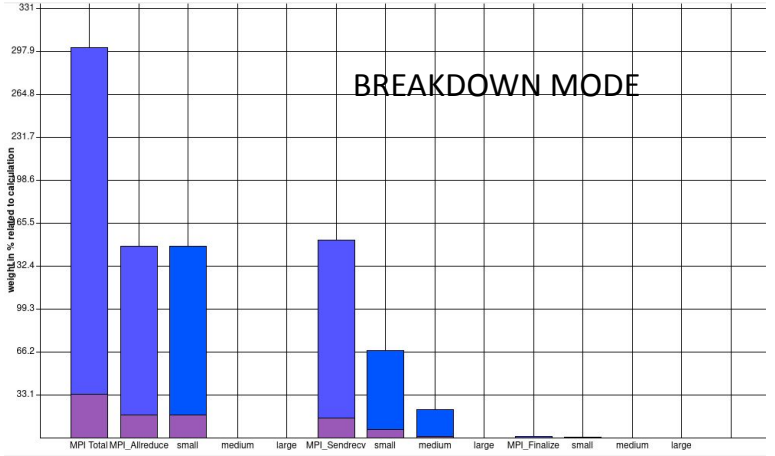
ITAC - IDEAL TRACES



ITAC - IMBALANCE DIAGRAM




```
Advanced > Event timeline
Advanced > Imbalance Diagram
```



Interconnet → time spent waiting
Imbalance → load balance
time transfer → T_real - T_ideal

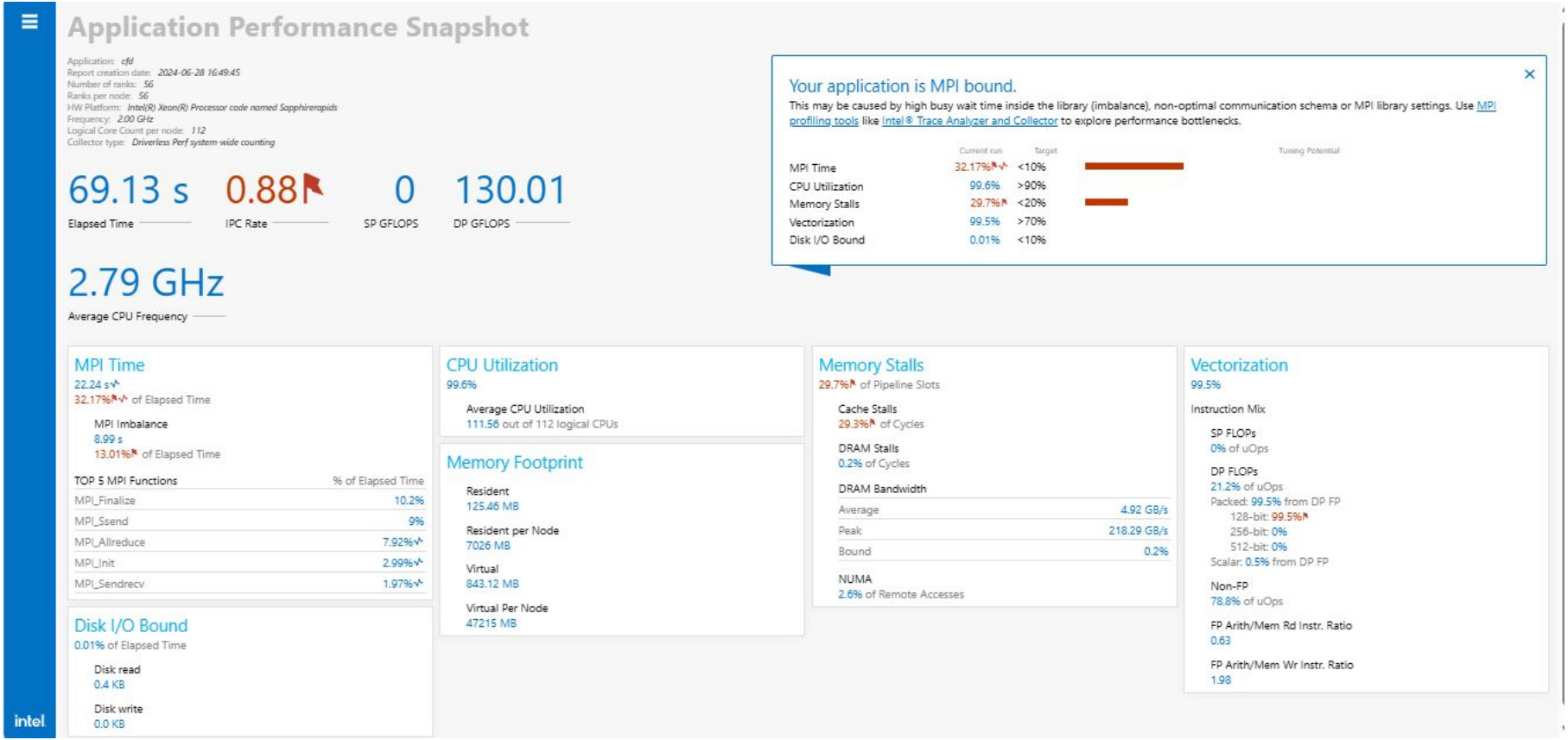
APPLICATION PERFORMANCE SNAPSHOT

- Does not requires source code instrumentation
- Provides summaries about MPI, OpenMP and memory
- Provides hints on the bottleneck, suggest analysis with VTUNE

```
mpirun <options> aps <my-app>

aps --report aps_result_<date>
```

APPLICATION PERFORMANCE SNAPSHOT



VTUNE

- Does not requires source code instrumentation
- Analyze **CPU** and **(Intel) GPU** performances
- Supports **MPI, OpenMP** and **OpenMP target** codes
- Provides **traces** with for CPU and GPU utilization, memory access and cache BW

```
vtune -collect <collection-name> ./matrix
```

- performance-summary
- hotspot
- memory-access
- **hpc-performance**

VTUNE hpc-performance collection

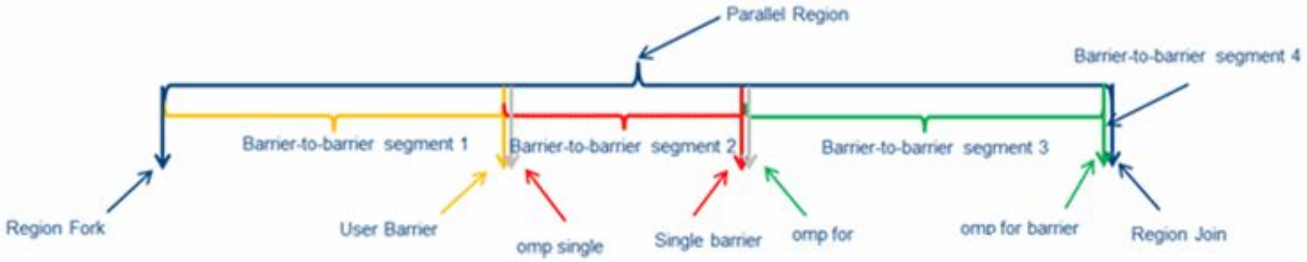
```
!$omp parallel shared ( n ) private ( i, j, prime )
!$omp do reduction ( + : total )
do i = 2, n
    prime = 1
    do j = 2, i - 1
      if ( mod ( i, j ) == 0 ) then
        prime = 0
        exit
      end if
    end do
    total = total + prime
end do
!$omp end do
!$omp end parallel
```

```
ifort -g -qopenmp -O2 -parallel-source-info=2 prime_openmp.f90 -o prime_openmp.exe
vtune -collect hpc-performance ./prime_openmp.exe
```

VTUNE hpc-performance collection

Provides the time spent in the implicit barriers

```
#pragma omp parallel
{
    .....
#pragma omp barrier
#pragma omp single
    {
        ....
    }
#pragma omp for
    {
        ...
    }
}
```



VTUNE hpc-performance collection

HPC Performance Characterization

INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up

Elapsed Time: 15.247s

SP GFLOPS:	0.000
DP GFLOPS:	0.000
x87 GFLOPS:	4.988
CPI Rate:	0.899
Average CPU Frequency:	3.1 GHz
Total Thread Count:	12

Effective CPU Utilization: 13.4%

Average Effective CPU Utilization: 6.436 out of 48

Serial Time (outside parallel regions): 0.101s (0.7%)

Parallel Region Time: 15.146s (99.3%)

Estimated Ideal Time: 8.229s (54.0%)

OpenMP Potential Gain: 6.917s (45.4%)

Top OpenMP Regions by Potential Gain

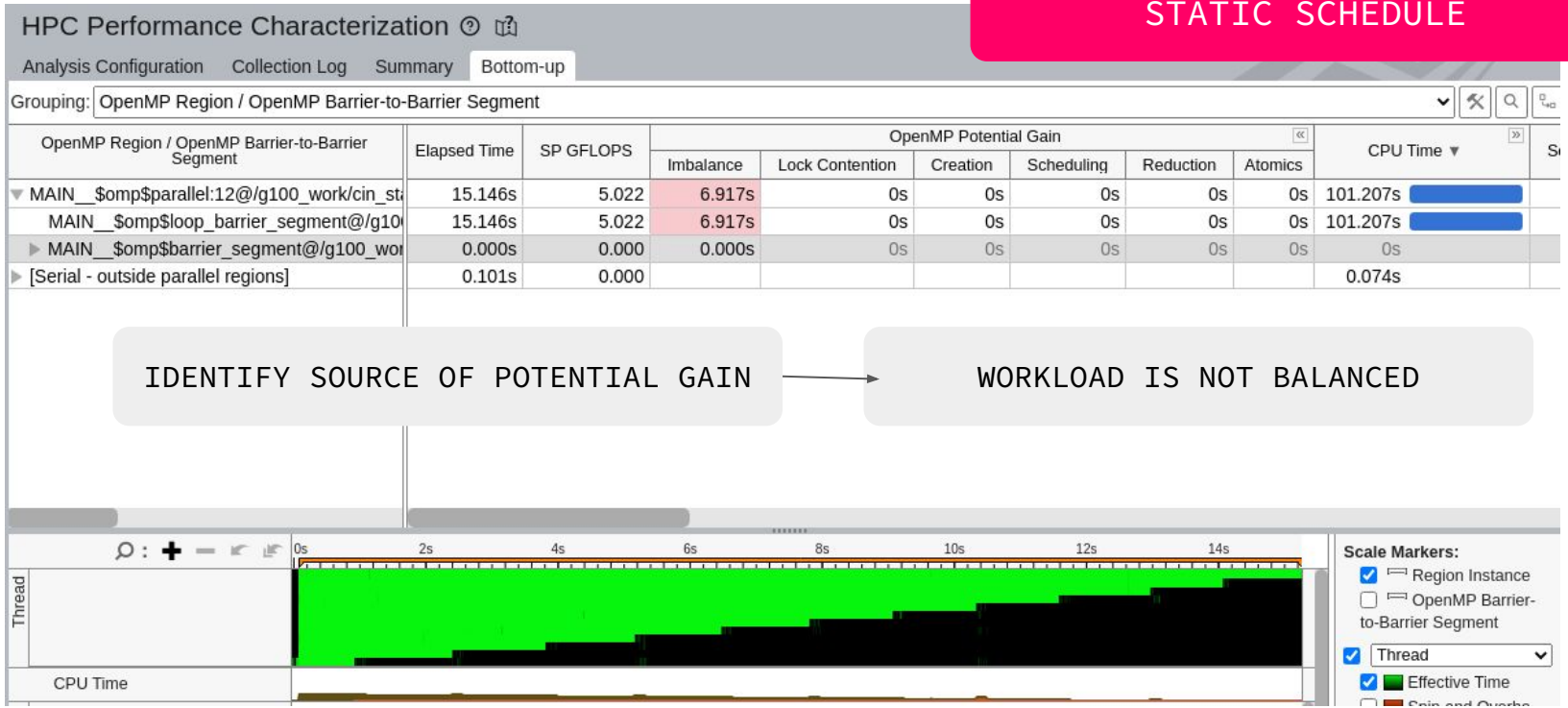
This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
MAIN__ompparallel:12@/g100_work/cin_staff/lbellen1/courses_genrepos/debugging-and-code-optimization/profiling/2023/4-vtune/prime_openmp.f90:174:195	6.917s	45.4%	15.146s

*N/A is applied to non-summable metrics.

VTUNE hpc-performance collection

STATIC SCHEDULE



VTUNE hpc-performance collection

DYNAMIC SCHEDULE