CINECA

Tutorial on Profiling @ MHPC

Profiling High Performance Computing applications

12-14 FEB 2025

l.bellentani@cineca.it
m.celoria@cineca.it
s.orlandini@cineca.it

Why profiling?

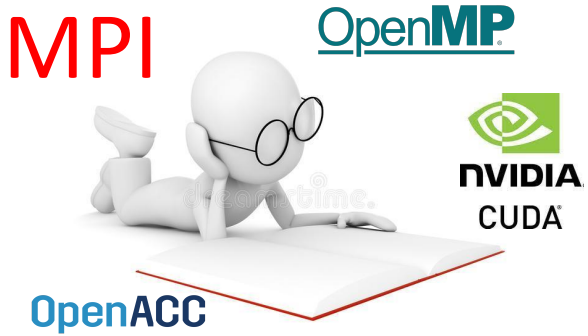
```
Starting main loop ...
 ... finished

 After       100 iterations, error is  0.7069E-02
 Time for    100 iterations was   15.26     seconds
 Each individual iteration took  0.1526     seconds
```

Why profiling?

```
Starting main loop ...
 ... finished

 After       100 iterations, error is  0.7069E-02
 Time for    100 iterations was   15.26     seconds
 Each individual iteration took  0.1526     seconds
```



Why profiling?

procs	2 MPI RANK
time (s)	7.8
eff	97%

Why profiling?

procs	2 MPI RANK	4 MPI RANK
time (s)	7.8	4,76
eff	97%	80%

Why profiling?

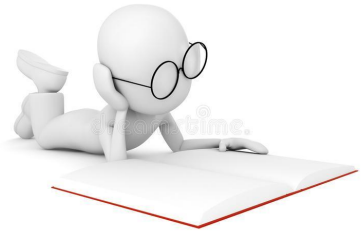
procs	2 MPI RANK	4 MPI RANK	16 MPI RANK
time (s)	7.8	4,76	1,58
eff	97%	80%	60%

Why profiling?

procs	2 MPI RANK	4 MPI RANK	16 MPI RANK	32 MPI RANK
time (s)	7.8	4,76	1,58	1,00
eff	97%	80%	60%	47%




Why profiling?



Why profiling?

VAMPIR

Tuning and Analysis Utilities

NVIDIA® Nsight™

Score-P

Scalable performance measurement infrastructure for parallel codes

Youth4work

intel

Intel VTune

Intel VTune

Why profiling?

VAMPIR

Tuning and Analysis Utilities

NVIDIA® Nsight™

Score-P

Scalable performance measurement infrastructure for parallel codes

Youth4work

intel

Intel VTune

Intel VTune

The Agenda

1. Fundamentals of profiling
    a. The glossary
    b. Profiling techniques and libraries
2. Parallel programming and common bottlenecks
    a. MPI and OpenMP metrics
    b. Score-P
3. GPU computing
    a. Profiling heterogenous codes with NSight Systems
    b. Analyze GPU metrics with NSight Compute

The Agenda

TODAY

UNDERSTAND THE IDEA OF MEASURING AN APPLICATION

IDENTIFY A SUITABLE MEASUREMENT TOOL ACCORDING TO YOUR PROGRAM

TOMORROW

READ THE USER GUIDE!

FOLLOW COURSES AND SEMINARS ON PROFILING AND SPECIFIC TOOLS

POP online training
https://pop-coe.eu/further-information/online-training

Vi-HPS training courses
https://www.vi-hps.org/training/index.html

Fundamentals

l.bellentani@cineca.it
Tutorials on profiling @ MHPC, ITAC

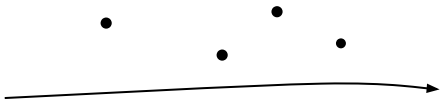
A matter of time

PROFILING:

*the act of measuring the properties of an application by **summarizing** a set of events **over the execution interval***

TRACING :

*recording the **stream of the events**, to provide additional information on **where and when** it takes place during the execution*



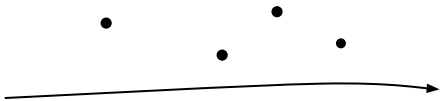
A matter of time

PROFILING:

*the act of measuring the properties of an application by **summarizing** a set of events **over the execution interval***

TRACING :

*recording the **stream of the events**, to provide additional information on **where and when** it takes place during the execution*



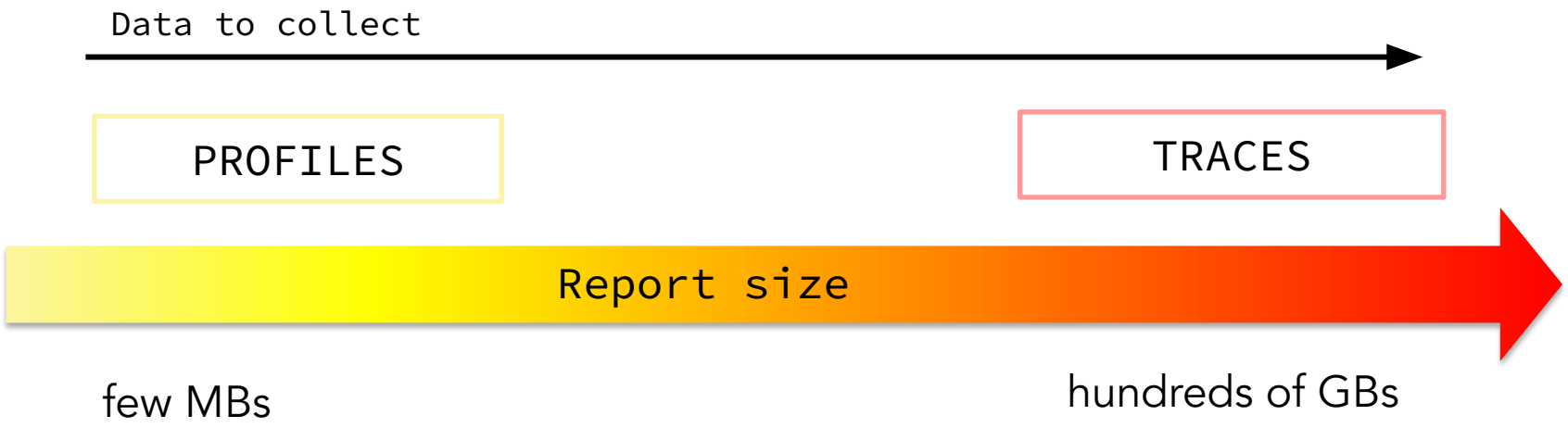
Results depend on the **test case** chosen !

Information is expensive

The **amount of events** monitored influences the report size

Information is expensive

The **amount of events** monitored influences the report size



What goes around comes around

Measurement **affects** performances

- **OVERHEAD**:
  - measurement takes time, memory, uses CPU, thus lowering performances
  - Increases with the amount of data to collect
- **PERTURBATION**:
  - measurement might hinder optimization strategies
- **ACCURACY**:
  - limited by timer and counter one, more detrimental on short routines

What goes around comes around

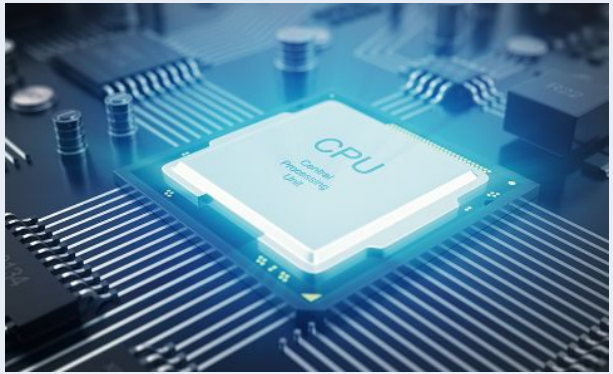
Measurement **affects** performances

Focus your measurement on a **subset of events**

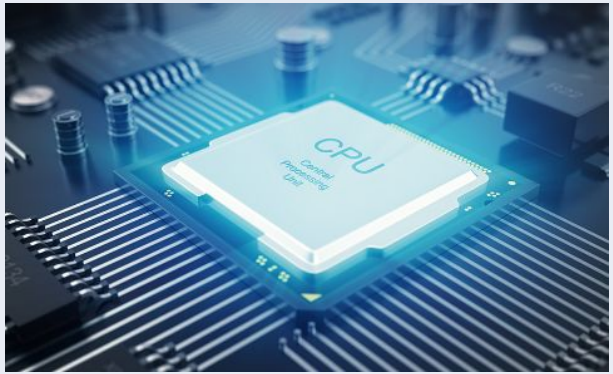
Identify a **suitable test case** for this set

Apply **filters**

What makes you stronger

PERFORMANCE FACTORS	
SEQUENTIAL	PARALLEL
 ?	

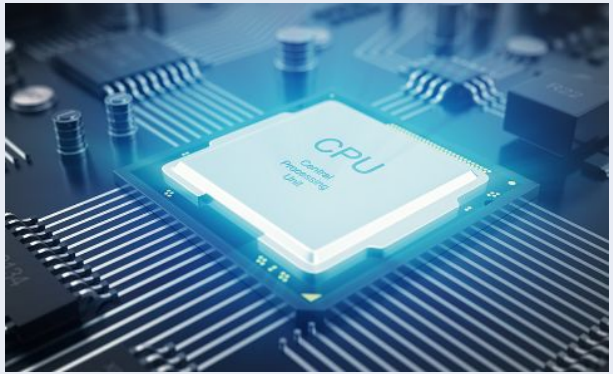
What makes you stronger

PERFORMANCE FACTORS	
SEQUENTIAL	PARALLEL
 Vectorization LX cache misses, hints #instruction / #cycles I/O volume	

What makes you stronger

PERFORMANCE FACTORS	
SEQUENTIAL	PARALLEL
 Vectorization LX cache misses, hints #instruction / #cycles I/O volume	 ?

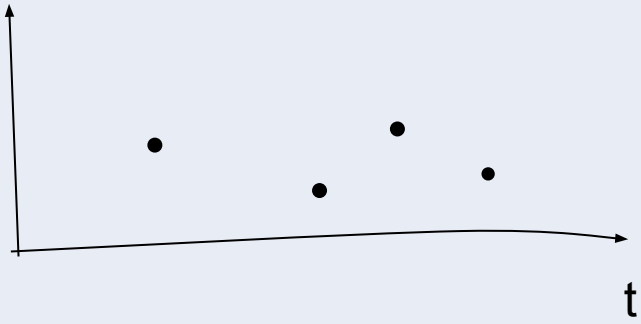
What makes you stronger

PERFORMANCE FACTORS	
SEQUENTIAL	PARALLEL
	
Vectorization LX cache misses, hints #instruction / #cycles I/O volume	Workload decomposition Communications Amount of serial work Synchronization

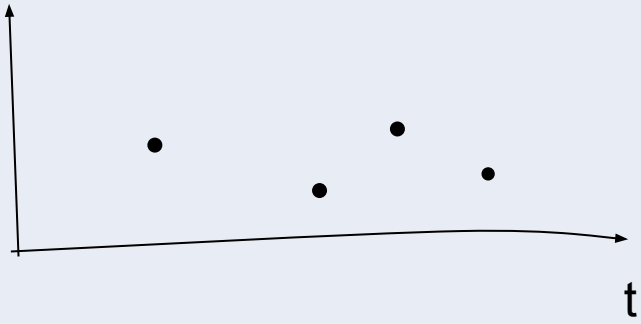
What makes you stronger

MAIN METRICS	
PROFILING	TRACING
	
?	

What makes you stronger

MAIN METRICS	
PROFILING	TRACING
 Time Counts (visits) Bytes transferred HW counters	 ?

What makes you stronger

MAIN METRICS	
PROFILING	TRACING
	
Time Counts (visits) Bytes transferred HW counters	Timestamp Location Event-type Communicator

It’s all about the right chart

Profiles Performance metrics are summarized over the program execution time

FLAT PROFILE

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.34	0.02	0.02	7208	0.00	0.00	open
16.67	0.03	0.01	244	0.04	0.12	offtime
16.67	0.04	0.01	8	1.25	1.25	memccpy
16.67	0.05	0.01	7	1.43	1.43	write
16.67	0.06	0.01				mcount
0.00	0.06	0.00	236	0.00	0.00	tzset
0.00	0.06	0.00	192	0.00	0.00	tolower
0.00	0.06	0.00	47	0.00	0.00	strlen
0.00	0.06	0.00	45	0.00	0.00	strchr
0.00	0.06	0.00	1	0.00	50.00	main
0.00	0.06	0.00	1	0.00	0.00	memcpy
0.00	0.06	0.00	1	0.00	10.11	print
0.00	0.06	0.00	1	0.00	0.00	profil
0.00	0.06	0.00	1	0.00	50.00	report

…

EXCLUSIVE vs INCLUSIVE time

?

It’s all about the right chart

Profiles Performance metrics are summarized over the program execution time

FLAT PROFILE

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
…
```

```
subroutine do_operation:
     <…>
     call open()
     <…>
end subroutine
```

It’s all about the right chart

Profiles Performance metrics are summarized over the program execution time

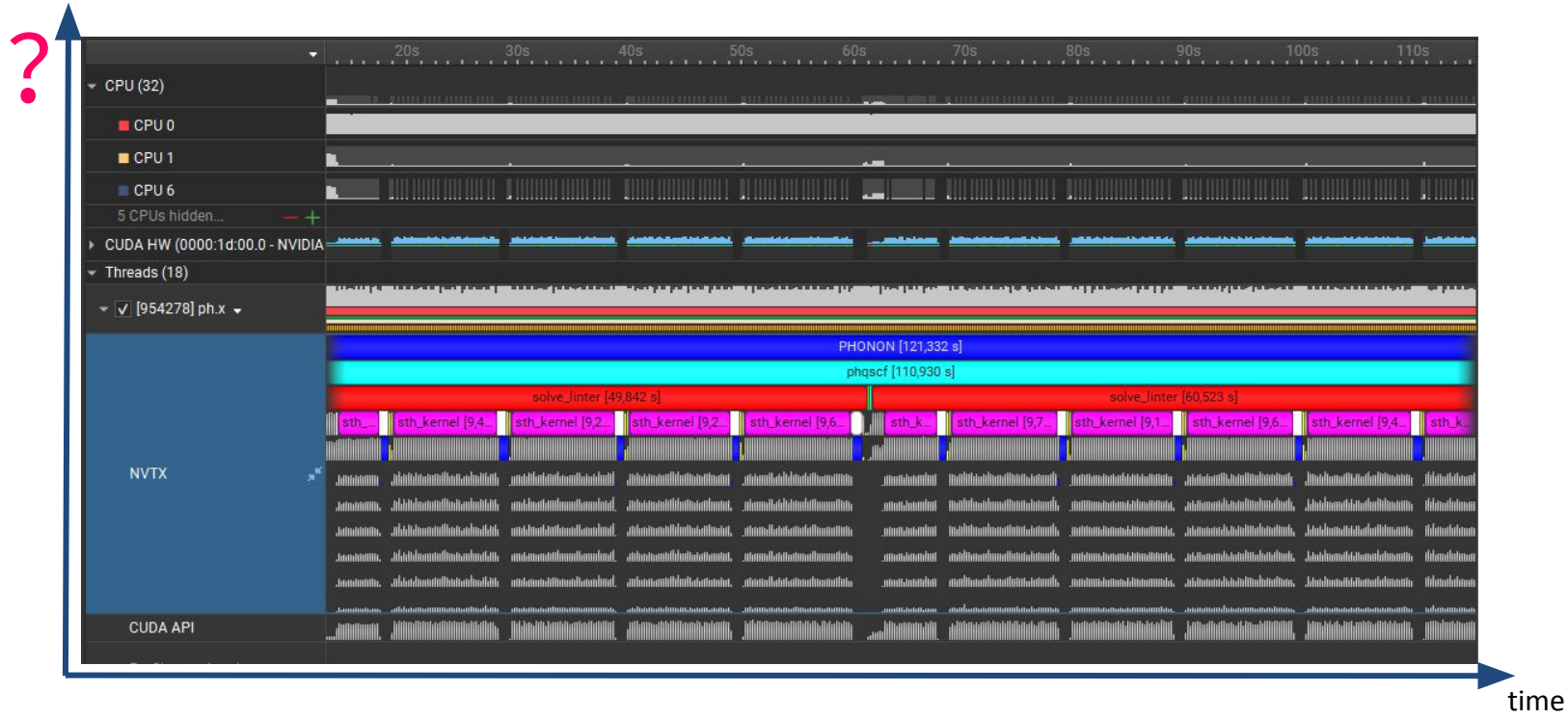
```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

CALL GRAPH

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
```

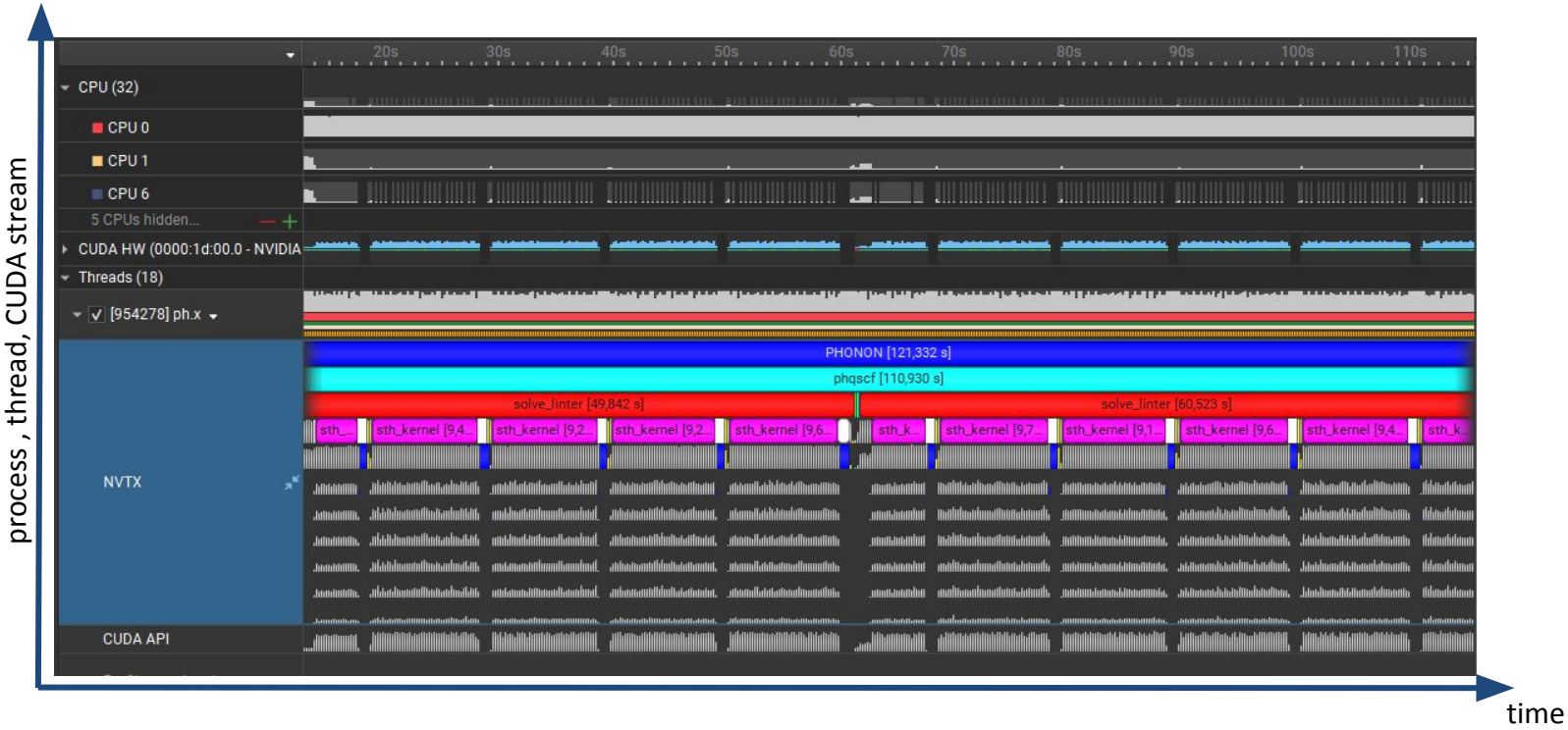
It’s all about the right chart

Traces Variation of the performance metric and distribution of the event over time



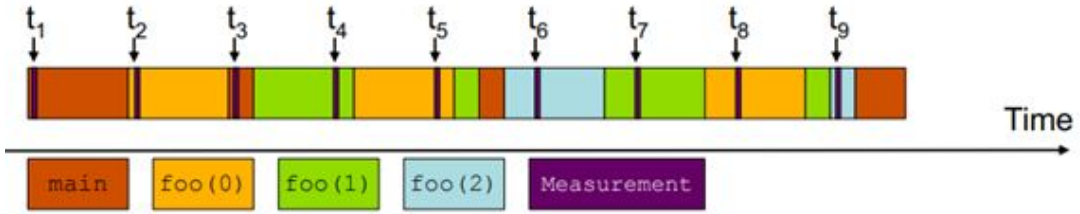
It’s all about the right chart

Traces Variation of the performance metric and distribution of the event over time



and the right measurement

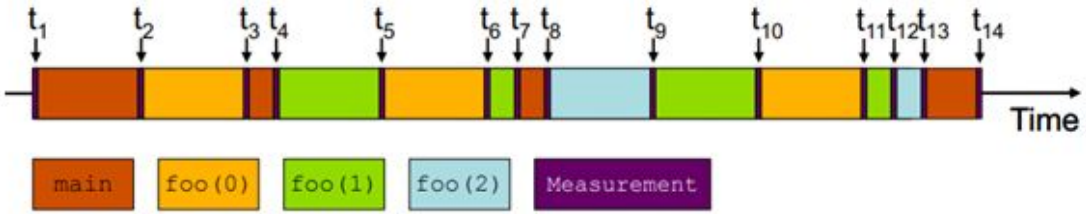
SAMPLING



- OS interrupts CPU to record currently-executed instruction at regular intervals
- Profiler correlates the record with the corresponding routine/line in the source code.
- Frequency of the routine recorded or code line is estimated statistically.

and the right measurement

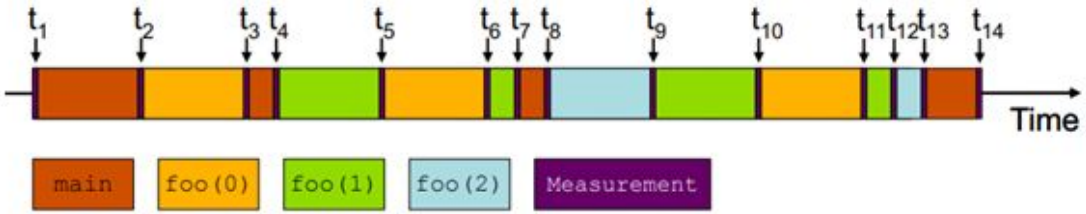
INSTRUMENTATION



- Special code is added at the beginning and the end of functions to measure an event.
- Two kinds:
  - *source-code modifiers* : intrusive, need to recompile the code
  - *binary profilers* : work at runtime, insert instrumentation at the first assembly instruction of each routine

and the right measurement

INSTRUMENTATION



- Special code is added at the beginning and the end of functions to measure an event.
- Two kinds:
  - *source-code modifiers* : intrusive, need to recompile the code +EFFORT, +INFO
  - *binary profilers* : work at runtime, insert instrumentation at the first assembly instruction of each routine -EFFORT, -INFO

How to choose?

PROS AND CONS	
SAMPLING	INSTRUMENTATION

overhead ?

irregular codes ?

How to choose?

PROS AND CONS	
SAMPLING	INSTRUMENTATION
✔ negligible overhead	✔ suitable for irregular codes
✔ no modification of source code	✔ provides the exact frequency
✔ suitable for frequent small ruotines	
✗ requires long runs and regular codes	✗ large overhead on short routines

How to choose?

PROS AND CONS	
SAMPLING	INSTRUMENTATION
✔ negligible overhead	✔ suitable for irregular codes
✔ no modification of source code	✔ provides the exact frequency
✔ suitable for frequent small ruotines	
✗ requires long runs and regular codes	✗ large overhead on short routines
compile with debug symbols !	

If I may suggest

Profiling is a starting point to understand on which events you should focus on

Traces are fundamental to understand how process, streams, threads interacts

An example from experience

My simulation uses MPI ranks to distribute independent calculations (“solve_linter”) in a loop

In my test case there are 216 independent solve_linter calculations

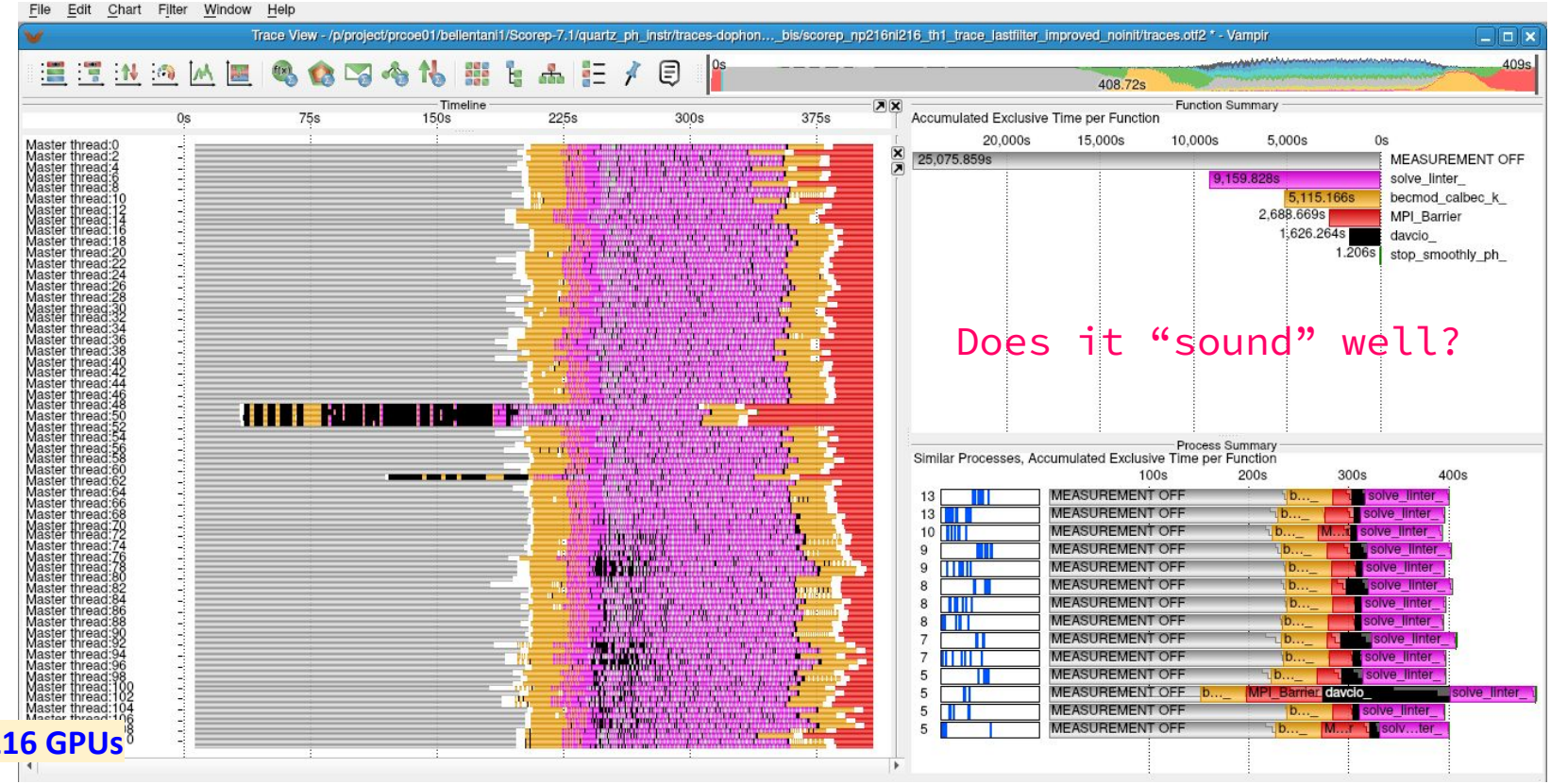
Each MPI ranks offloads computation to a GPUs

In principle, I could distribute solve_linter up to 216 GPUs with little communications. Is the code efficient ?

GPUS	1	24	108	216
EFFICIENCY	1	0.9	0.78	0.56

If I may suggest

If I may suggest



Libraries and softwares

l.bellentani@cineca.it
Tutorial on profiling @ MHPC

Where to start

- Does not require to re-compile the source code, negligible overhead
- **Wall/elapsed time** vs **CPU time**.

WALL TIME : difference from the start to the end of a program execution, for all threads or tasks

CPU TIME : time that a CPU was used for processing instructions of a computer program

time for UNIX/Linux shells

```
# shell keyword, see help time, works for pipelines
time ./matrixmul.exe
real    0m7.357s
user    0m7.335s
sys     0m0.020s

# executable, see man time
/usr/bin/time ./matrixmul.exe
7.30user 0.00system 0:07.31elapsed 99%CPU
(0avgtext+0avgdata 22912maxresident)k
0inputs+128outputs (0major+435minor)pagefaults 0swaps

/usr/bin/time sleep 10
0.00user 0.00system 0:10.00elapsed 0%CPU (0avgtext+0avgdata
3136maxresident)k
0inputs+0outputs (0major+70minor)pagefaults 0swaps
```

date for scripts

```
#!/bin/bash
start_time=$(date +"%s")
echo date
./matrixmul.exe
end_time=$(date +"%s")
walltime=$(($end_time-$start_time))
echo "walltime (s) is :" $walltime
```

A top! command for parallel programs

“top” command

- Infos about each process running on the present node, including MPI processes.
- Per-process wall time time, memory, CPU usage.

! CPU > 100% with OpenMP threads

```
%Cpu(s):  3.2 us,  0.0 sy,  0.0 ni, 96.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem : 322140.5 total, 213124.9 free,  82320.4 used,  26695.2 buff/cache
MiB Swap:   4095.9 total,   4095.9 free,      0.0 used. 238316.8 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
3335658 lbellen1  20   0   26560   9728   4928 R   1.0   0.0   0:00.14 top
3335621 lbellen1  20   0   32.2g  15.8g  35072 R 100.0   5.0   0:25.78 matrixmul_mpi.e
3335622 lbellen1  20   0   16.2g  12.6g  35520 R 100.0   4.0   0:25.79 matrixmul_mpi.e
3335623 lbellen1  20   0   16.2g  12.6g  35648 R 100.0   4.0   0:25.77 matrixmul_mpi.e
3335624 lbellen1  20   0   16.2g  12.6g  35712 R 100.0   4.0   0:25.79 matrixmul_mpi.e
  12317 root      20   0 2559616  62656  26496 S   1.3   0.0  61:43.04 mmsysmon.py
```

```
mpirun -np 4 ./mpimatrix.exe
```

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
3338873 lbellen1  20   0   16.9g 174144   8064 R 797.4   0.1   0:51.24 matrixmul_ope
3338884 lbellen1  20   0   26560   9728   4928 R   1.0   0.0   0:00.08 top
 996562 root      20   0  275904  49472  19392 S   0.7   0.0  17:38.73 python2
     11 root      20   0       0      0      0 I   0.3   0.0   7:56.92 rcu_sched
   5352 root      20   0  832000  57472  29312 S   0.3   0.0  24:58.93 fail2ban-server
  17849 root       0 -20   70.1g   5.6g   1.4g S   0.3   1.8 101:31.69 mmfsd
```

```
export OMP_NUM_THREADS=8
```

How long does it take?

- Wrap subroutines or regions of the source code with timing functions (*instrumentation*) to detect the most time-consuming portion of the code.

```
CALL start_clock()
[source code to measure...]
CALL stop_clock()
```

- Intrusive method, introduces an overhead

SERIAL APPLICATIONS

- Fortran 77 : etime(), dtime()
- Fortran 90 : cputime(), system_clock(), date_and_time()
- C/C++: clock()

Careful with multithreading

SYSTEM_CLOCK([COUNT, COUNT_RATE, COUNT_MAX])

```
REAL       :: walltime, rate
INTEGER :: c1,c2,cr,cm ! counts, resolution, capacity

! (1) initialize the system clock
 CALL system_clock(count_rate=cr)
 rate = REAL(cr)
! (2) measure time
 CALL system_clock(c1)
!$omp parallel do collapse(3)
 DO j=1,numy ; DO i=1,numx ; DO k=1,numz
             c(i,j)=c(i,j)+a(i,k)*b(k,j)
 END DO ; END DO ; END DO
!$omp end parallel do
 CALL system_clock(c2)
```

```
walltime : 3.83899999

real    0m3.867s
user    0m15.169s
sys     0m0.000s
```

CPU_TIME(TIME)

```
REAL       :: cputime, t1, t2

! (1) measure time
 CALL cpu_time(t1)

 DO j=1,numy ; DO i=1,numx ; DO k=1,numz
             c(i,j)=c(i,j)+a(i,k)*b(k,j)
 END DO ; END DO ; END DO

 CALL cpu_time(t2)
 cputime = t2 - t1
```

```
cpu time : 14.8512011

real    0m3.764s
user    0m14.863s
sys     0m0.010s
```

```
export OMP_NUM_THREADS=4
```

Careful with multithreading

SYSTEM_CLOCK([COUNT, COUNT_RATE, COUNT_MAX])

```
REAL       :: walltime, rate
INTEGER :: c1,c2,cr,cm ! counts, resolution, capacity

! (1) initialize the system clock
 CALL system_clock(count_rate=cr)
 rate = REAL(cr)
! (2) measure time
 CALL system_clock(c1)
!$omp parallel do collapse(3)
 DO j=1,numy ; DO i=1,numx ; DO k=1,numz
             c(i,j)=c(i,j)+a(i,k)*b(k,j)
 END DO ; END DO ; END DO
!$omp end parallel do
 CALL system_clock(c2)
```

```
walltime : 3.83899999

real    0m3.867s
user    0m15.169s
sys     0m0.000s
```

CPU_TIME(TIME)

```
REAL       :: cputime, t1, t2

! (1) measure time
 CALL cpu_time(t1)
!$omp parallel do collapse(3)
 DO j=1,numy ; DO i=1,numx ; DO k=1,numz
             c(i,j)=c(i,j)+a(i,k)*b(k,j)
 END DO ; END DO ; END DO
!$omp end parallel do
 CALL cpu_time(t2)
 cputime = t2 - t1
```

! cpu_time is sensitive to the number of threads

```
export OMP_NUM_THREADS=4
```

```
cpu time : 14.8512011

real    0m3.764s
user    0m14.863s
sys     0m0.010s
```

APIs from libraries might be better

PARALLEL APPLICATIONS

- OpenMP : omp_get_wtime()
- MPI : MPI_wtime()

DOUBLE PRECISION **omp_get_wtime()**

```
REAL*8     ::  t1, t2
!$omp parallel private(t1,t2)
 t1 = omp_get_wtime()
!$omp do collapse(3)
 DO j=1,numy ; DO i=1,numx ; DO k=1,numz
             c(i,j)=c(i,j)+a(i,k)*b(k,j)
 END DO ; END DO ; END DO
!$omp end do
t2 = omp_get_wtime()
!$omp end parallel
walltime = pt2 - pt1
```

DOUBLE PRECISION **MPI_WTIME()**

```
USE MPI
[…]
REAL*8     ::  pt1, pt2
[…]
 pt1 = MPI_WTIME()
 DO j=1,numy ; DO i=1,numx ; DO k=1,numz
             c(i,j)=c(i,j)+a(i,k)*b(k,j)
 END DO ; END DO ; END DO
pt2 = MPI_WTIME()
walltime = pt2 - pt1
```

Let’s dig deeper

Hardware counters special-purpose registers built into modern microprocessors to store the counts of hardware-related activities

- Used to compute derived metrics such as instruction per cycle, load imbalances in MPI
- More accurate, low overhead

Let's dig deeper

Hardware counters special-purpose registers built into modern microprocessors to store the counts of hardware-related activities

- ✳ Used to compute derived metrics such as instruction per cycle, load imbalances in MPI
- ✳ More accurate, low overhead

PAPI: high-level portable interface providing access to hardware counters.

- Memory hierarchy access events [PAPI_LX_DCM, X=1,2,3 ...]
- Cycle/instruction counts [**PAPI_TOT_CYC, PAPI_TOT_INS**, PAPI_VEC_INS, ...]
- Pipeline status [PAPI_MEM_SCY, ...]

- ✳ PAPI APIs are callable from your C/FORTRAN code to start/stop/read event counters.
- ✳ Integrated with many profiling tools, e.g. Vtune, Scalasca and TAU.

The PAPI library

```
$ papi_avail

Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------------
PAPI version             : 6.0.0.1
Operating system         : Linux 4.18.0-147.51.2.el8_1.ppc64le
Vendor string and code   : IBM (3, 0x3)
Model string and code    : 8335-GTG (0, 0x0)
CPU revision             : 2.000000
CPU Max MHz              : 3800
CPU Min MHz              : 2300
Total cores              : 128
SMT threads per core     : 4
Cores per socket         : 16
Sockets                  : 2
Cores per NUMA region    : 128
NUMA regions             : 1
Running in a VM          : no
Number Hardware Counters : 5
Max Multiplex Counters   : 384
Fast counter read (rdpmc): no
--------------------------------------------------------------------------------

================================================================================
  PAPI Preset Events
================================================================================
    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   Yes  Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   No   Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  Yes   Yes  Level 3 data cache misses
```

The PMPI library

- In the MPI standard, each function can be called with MPI_ (weak symbol) or PMPI_ prefix (strong symbol), e.g. MPI_Send() <=> PMPI_Send()
- Interposition of custom MPI_ routines to profile MPI calls:
  - communication performed with PMPI_ routine
  - extra operations to collect information

```
static int totalbytes = 0 ;
static double totalTime = 0.0 ;
int MPI_Send(const void*start,int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
{
     double t_start = MPI_Wtime() ;
     int size ;
     int result = PMPI_Send(start, count, datatype, dest, tag, comm);
     totaltime += MPI_Wtime() - tstart ;
     MPI_Type_size(datatype, &size) ;
     totalbytes += count*size
     return result
}
```

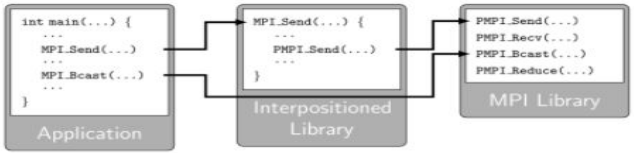
cumulates time

cumulates message size

The PMPI library

- **Library interposition** adopted by different profiler tools, such as Vtune, Scalasca, Tau: these **intercept MPI calls,** perform message-passing with PMPI_ and collect profiling data

```
profile.0.0.0
-------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
-------------------------------------------------------------------------------
100.0        1,784        2,867           1      175226    2867083 .TAU application
 21.7          622          622           1           1     622124 MPI_Init_thread()
  3.8          110          110       18028           0          6 MPI_Alltoall()
  2.5           72           72       23294           0          3 MPI_Recv()
  2.5           70           70       31286           0          2 MPI_Bcast()
  2.0           57           57        7526           0          8 MPI_Barrier()
  1.6           45           45       11213           0          4 MPI_Allreduce()
  0.8           22           22         705           0         32 MPI_Comm_split()
```




```
USER EVENTS: profile.0.0.0
-------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
-------------------------------------------------------------------------------
 1.121E+04      8E+05          4       1328  1.832E+04  Message size for all-reduce
 1.808E+04  1.749E+05          4       3631       8142  Message size for all-to-all
 3.129E+04      8E+05          4        626  1.254E+04  Message size for broadcast
      1984       5184          8       1864       1876  Message size for reduce
-------------------------------------------------------------------------------
```

The CUPTI library

CUDA Profiling Tools Interface (CUPTI) enables the creation of profiling and tracing tools that target CUDA applications

- CUDA API instrumentation, OpenACC
- NVLink metrics
- Kernel performance counters

Table 1. Description of CUPTI APIs

CUPTI API	Feature Description
Activity	Asynchronously record CUDA activities, e.g. CUDA API, Kernel, memory copy
Callback	CUDA event callback mechanism to notify subscriber that a specific CUDA event executed e.g. "Entering CUDA runtime memory copy"
Event	Collect kernel performance counters for a kernel execution
Metric	Collect kernel performance metrics for a kernel execution
Profiling	Collect performance metrics for a range of execution
PC Sampling	Collect continuous mode PC Sampling data without serializing kernel execution
Checkpoint	Provides support for automatically saving and restoring the functional state of the CUDA device

Handmade might not be the best idea

- Handmade profiling might be tedious and introduces overhead
- Software development tools are designed to analyze the performance of your applications **with reduced overhead**
- Performance data typically include
  - Time spent in subroutines and functions (MPI, CUDA or code-specific)
  - Number of calls
  - Memory usage, allocations/deallocations, leaks
  - Load balancing, thread usage
  - I/O performances (bandwidth, bytes read/written)
  - Communication pattern (peculiar of traces)

They know better

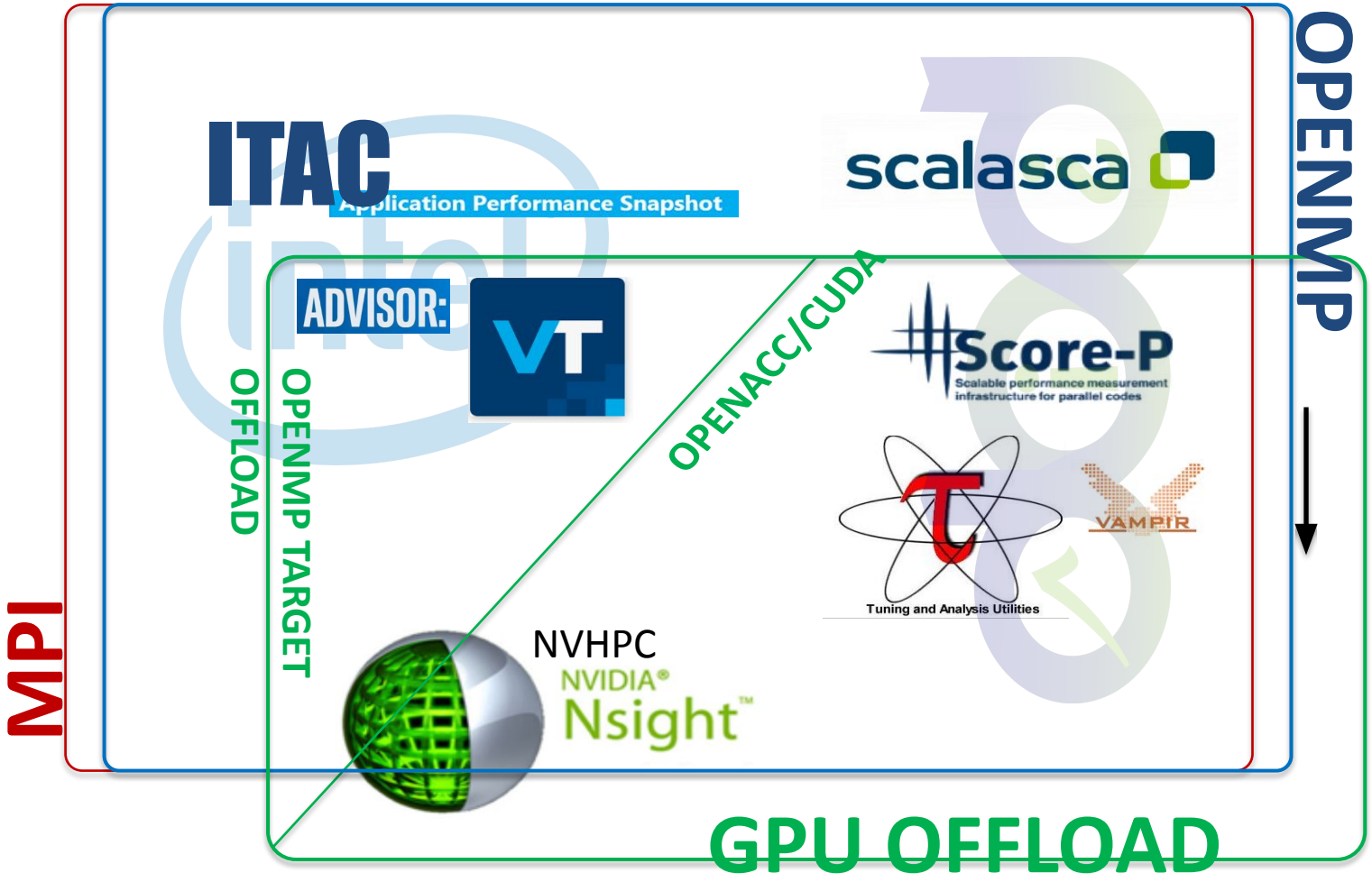
Tool name	Scope	Features
Scalasca-Score-P	Profiling and tracing of MPI and multithreaded -[OpenACC, CUDA,GPU kernels]	Installed on Marconi, G100 and Leonardo
Intel Trace Analyser and Collector (ITAC)	Quick tool to trace intel-compiled apps	Intel licensed profiler and tracing (Marconi, G100)
Intel Vtune Amplifier	Detailed profiling for intel applications	Intel licensed profiler (Marconi, G100, Leonardo)
Extrae/Paraver	General purpose tracing tool	Can be installed on CINECA machines
Valgrind	Memory debugging	Installed on CINECA machines
Tau	Profiling and tracing, PAPI	Can be installed on CINECA machines
Vampir	Tracing	Under license
Darshan	I/O	Can be installed on CINECA's machines
Nsight System	Traces of GPU-accelerated applications	CLI available on Leonardo, G100

Not always straightforward

!

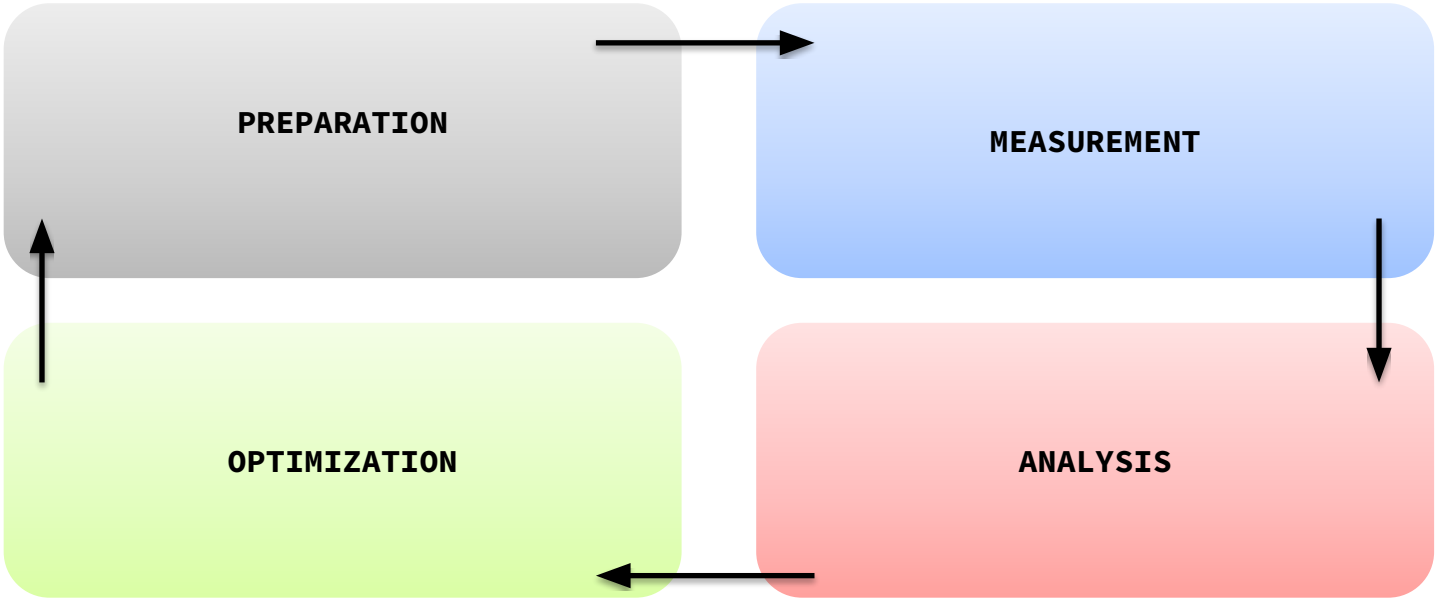
usually require to be installed with the **same compiler** and **MPI implementation** of the program to be profiled

!


ITAC
Application Performance Snapshot
scalasca
ADVISOR:
VT
Score-P
Scalable performance measurement infrastructure for parallel codes
VAMPIR
Tuning and Analysis Utilities
NVHPC
NVIDIA®
Nsight™
MPI
OPENMP TARGET OFFLOAD
OPENACC/CUDA
OPENMP
GPU OFFLOAD

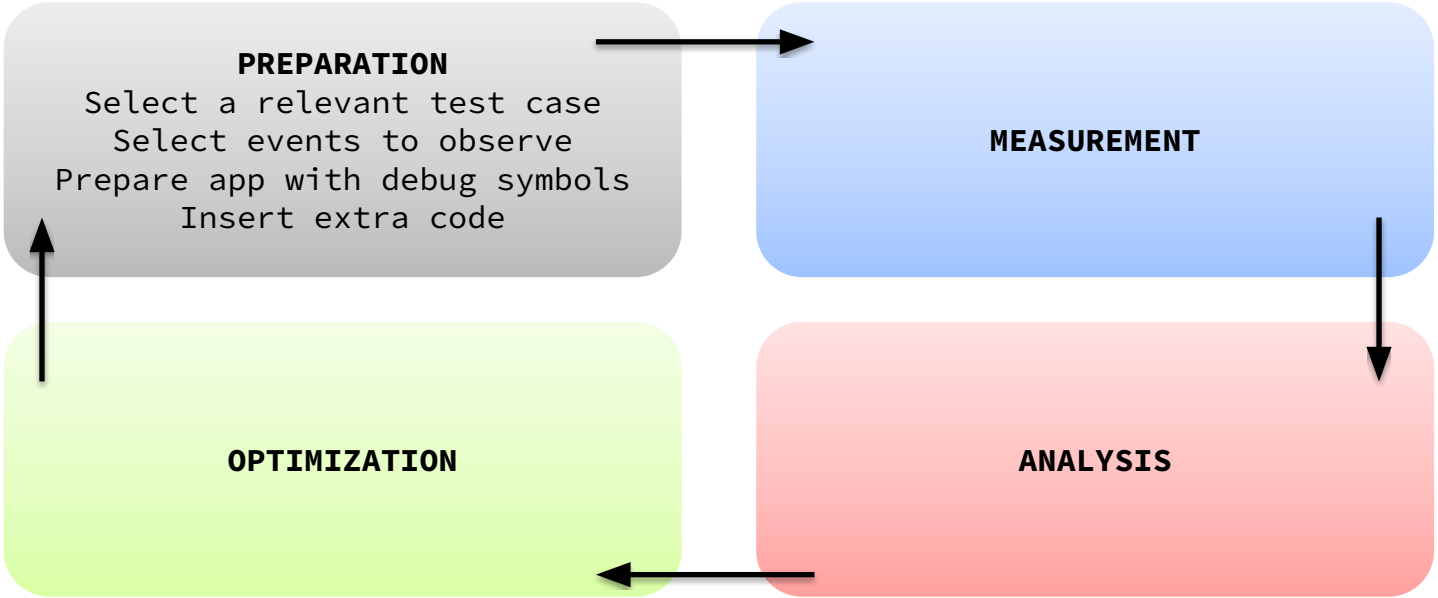
A never-ending story

VI-HPS



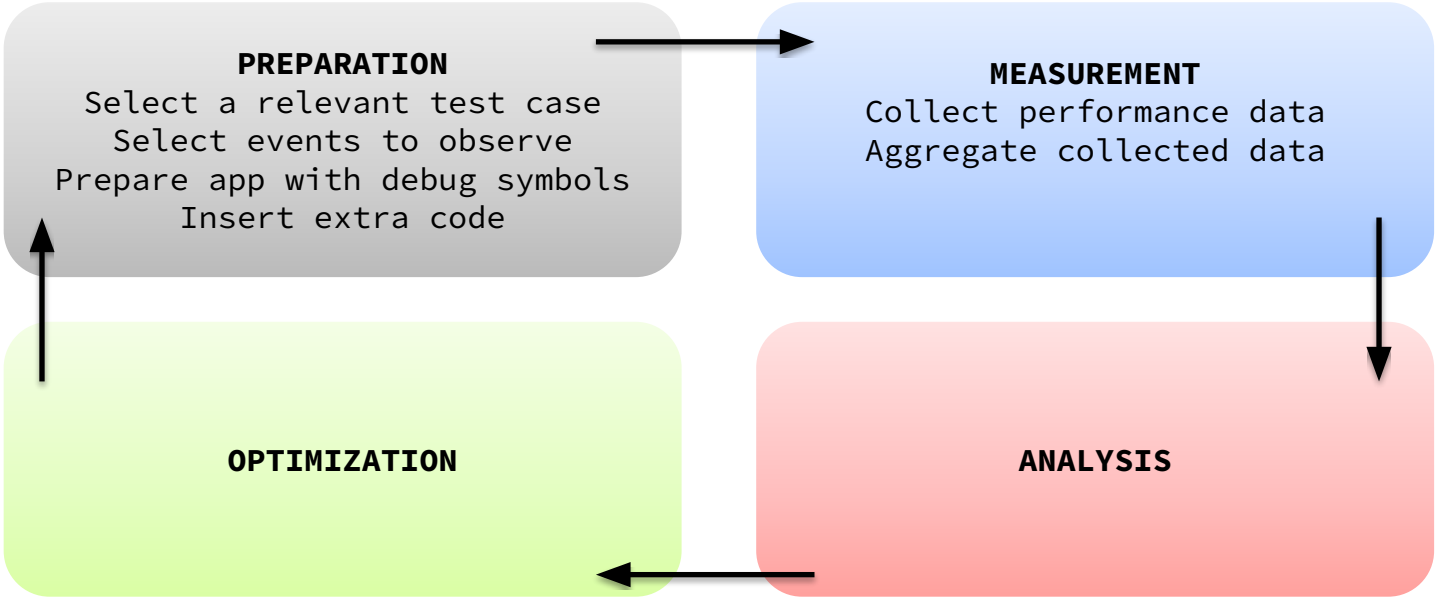
A never-ending story

VI-HPS



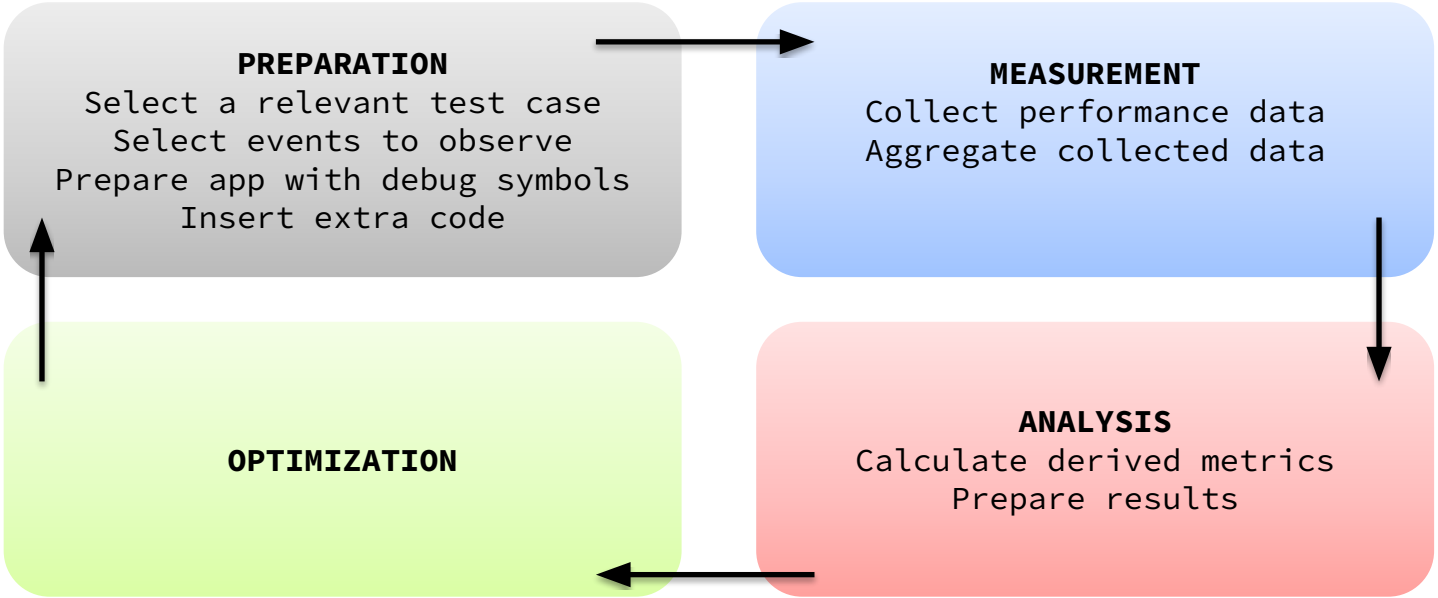
A never-ending story

VI-HPS



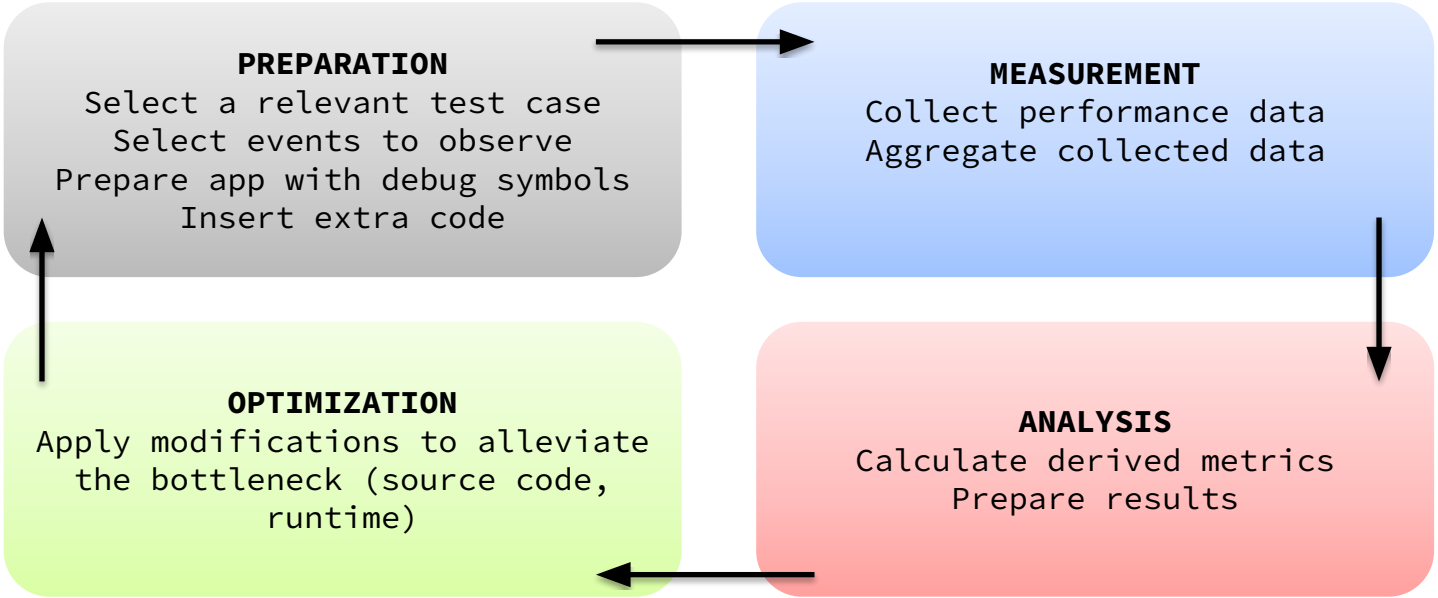
A never-ending story

VI-HPS



A never-ending story

VI-HPS



GIT CLONE HERE

https://gitlab.hpc.cineca.it/training/profiling-tutorial-mhpc

GNUprof

Hybrid of instrumentation and time-based sampling

- EVENTS: user function instrumentation
- METRICS: exclusive time, inclusive time, counts, time/call
- PROFILES: flat profile and basic call graph

PHASE	COMMANDS
Instrumentation: probes are added to your source code at compile time	gfortran **–pg** source.f90 –o app.exe
Measurement: a daemon collects event metrics while the application is running; a binary file (gmon.out) is generated	./app.exe
Postprocessing: turns binary profiles in human-readable tables	**gprof** app.exe gmon.out

compile and run with ./bin/cfd_serial.exe 128 30

GNUprof

FLAT PROFILE : Summary over time and call paths

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.88    175.32   175.32     2000    87.66    87.66  __core_MOD_evolve
  0.05    175.40     0.08        2    40.00    40.00  __utilities_MOD_average
  0.05    175.48     0.08        1    80.00    80.00  __setup_MOD_generate_field
  0.03    175.53     0.05        1    50.00    50.00  __utilities_MOD_copy_fields
  0.00    175.53     0.00     2000     0.00     0.00  __utilities_MOD_swap_fields
  0.00    175.53     0.00       41     0.00     0.00  __io_MOD_write_field
  0.00    175.53     0.00       41     0.00     0.00  __pngwriter_MOD_save_png
  0.00    175.53     0.00        2     0.00     0.00  __heat_MOD_set_field_dimensions
  0.00    175.53     0.00        1     0.00     0.00  __setup_MOD_finalize
  0.00    175.53     0.00        1     0.00   130.00  __setup_MOD_initialize
```

% time: time spent in each function (%)
calls: number of calls
cumulative seconds: time spent up to each functions (s)
self seconds: time spent in this function only (s)
self ms/call: average time spent in function per call
total ms/call: average time in function + descendent per call

GNUprof

CALL GRAPH : Results are resolved on the calling path

```
^L
                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.01% of 175.53 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00  175.53                 MAIN__ [1]
              175.32    0.00    2000/2000        __core_MOD_evolve [2]
                0.00    0.13       1/1           __setup_MOD_initialize [3]
                0.08    0.00       2/2           __utilities_MOD_average [4]
                0.00    0.00    2000/2000        __utilities_MOD_swap_fields [16]
                0.00    0.00      41/41          __io_MOD_write_field [17]
                0.00    0.00       1/1           __setup_MOD_finalize [20]
-----------------------------------------------
              175.32    0.00    2000/2000        MAIN__ [1]
[2]     99.9  175.32    0.00    2000           __core_MOD_evolve [2]
-----------------------------------------------
                0.00    0.13       1/1           MAIN__ [1]
[3]      0.1    0.00    0.13       1           __setup_MOD_initialize [3]
                0.08    0.00       1/1           __setup_MOD_generate_field [5]
                0.05    0.00       1/1           __utilities_MOD_copy_fields [6]
                0.00    0.00       2/2           __heat_MOD_set_field_dimensions [19]
-----------------------------------------------
                0.08    0.00       2/2           MAIN__ [1]
[4]      0.0    0.08    0.00       2           __utilities_MOD_average [4]
-----------------------------------------------
```

Parent (children) routines listed above (below) each routine

Self-called routines marked with a [+]

! Performance overhead can be large

! MPI and external routines are not instrumented