E4 COMPUTER ENGINEERING

WHEN PERFORMANCE MATTERS

PROFILING AND PERFORMANCE EVALUATION

Elisabetta Boella

Scandiano
November 28th, 2025

www.e4company.com

HOUSEKEEPING

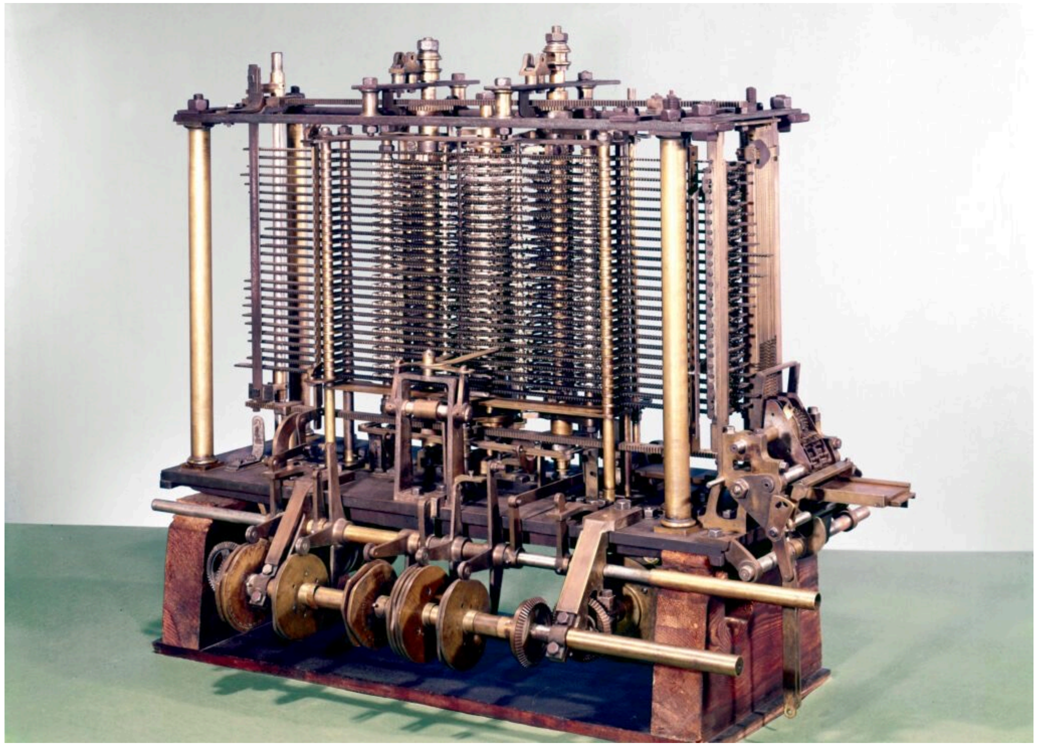
Material for today:

git clone

Further material and acknowledgement:

- https://gitlab.hpc.cineca.it/training/profiling-tutorial-mhpc
- https://github.com/EPCCed/archer2-scalasca-2021-07-27
- https://www.vi-hps.org/
- https://pop-coe.eu/

PERFORMANCE: AN OLD PROBLEM



The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.

Charles Babbage
1791 - 1871

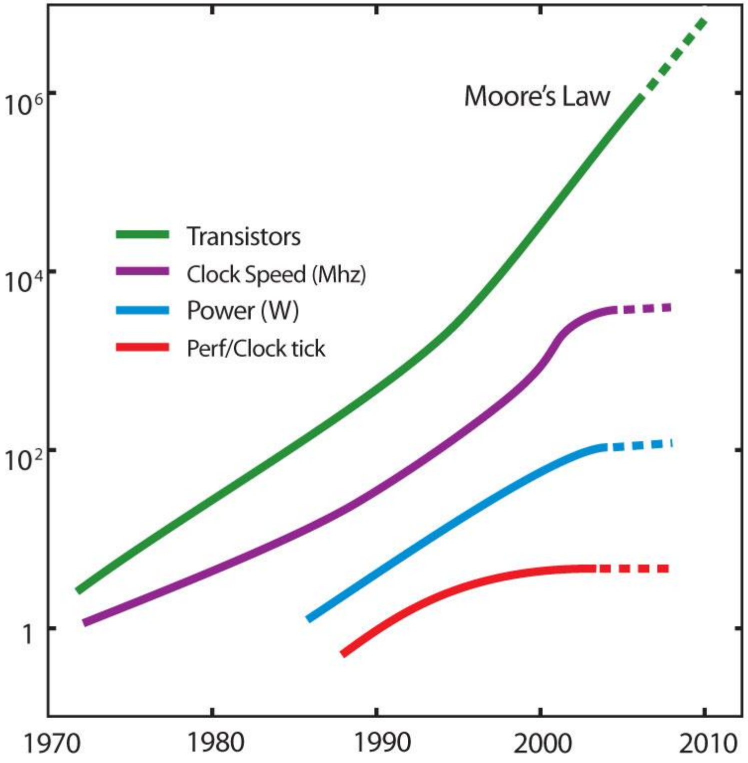
EVERYTHING COUNTS FOR PERFORMANCE

- Clock speed: how many instructions the processor can execute in a unit of time
- Floating point unit: how many operands can be operated on and what type of operations can the CPU execute at once
- Memory latency: how long does it take to fetch one piece of data from memory
- Memory bandwidth: how much data can be loaded up at any one time
- I/O to storage: how quickly can we access persisten data

Scientific codes can be

- Compute bound
- Memory bound (memory latency bound or memory bandwidth bound)
- IO bound

PERFORMANCE GAINS CAN NO LONGER BE EXPECTED JUST BY WAITING FOR NEW CHIPS



Clock rate no longer increases
Performance gains only through more parallelism

Optimising applications is complex
Optimising parallel applications is even more complex

Every doubling of scale reveals a new bottleneck

OPTIMISATIONS PERFORMED BY THE COMPILER

- Loop interchange

```
for( int i = 0; i< N; i++)
      for( int j=0; j<N; j++){
            matrix[i][j] = i*j;}
```

```
for( int j = 0; j< N; j++)
      for( int i=0; i<N; i++){
            matrix[i][j] = i*j;}
```

- Loop unrolling

```
sum = 0.0
do i = 1, n
      sum = sum + array(i)
end do
```

```
sum = 0.0
do i = 1, n, 4
      sum = sum + array(i) + array(i+1) + array(i+2) +  array(i+3)
end do
```

- Fast mathematical operators

OPTIMISATIONS PERFORMED BY THE COMPILER

- Function inlining

```
int main(){
      int x=10;
      cout << “ square value “ << pow(x) << endl;
      return 0;
}
int pow(int value){
      return value*value;
}
```

```
int main(){
      int x=10;
      cout << “ square value “ << x*x << endl;
      return 0;
}
```

Flag **-finline-limit=n** at compilation time + **inline** attribute (only C/C++)

- Operation reordering for cache reuse

WHEN TO USE COMPILER OPTIMISATIONS?

Always! Because they give beteer performance

If you are debugging your code, then it is better to turn optimisations off

Every compiler has its own flags for optimisation

GNU compiler **–O0, -O1, -O2, -O3, -Ofast** (**-O3** enables auto vectorisation; it can also be enabled at lower levels with **-ftree-vectorize)**

Beware that optimisations can change the order of calculation and since floating point arithmetic is not exact arithmetic, your code can produce slightly different results. Always check that your results are still "correct".

SOME USEFUL COMPILER FLAGS

	GNU (gcc, g++, gfortran)	Intel (icx, icpx, ifx)	NVHPC (nvc, nvc++, nvfortran)
Listing	-fdump-tree-all	-opt-report3	-Minfo
Vectorisation	-O3 or –tree-vectorize	-O2 and above	-O3 and -Mvect
Loop unrolling	-mavx	-x	-Mvect
Floating point optimization	-f[no-]fast-math or -funsafe-math-optimizations	-fast and -Ofast	-fast
Inter-procedural optimisation	-flto	-ipo	-ipo
Aggressive optimisation	-O3	-fast	-fast
Debugging	-g	-g	-g
Architecture	-march, -mtune, -mcpu	-mtune	-tp
OpenMP	-fopenmp	-qopenmp	-mp

EXECUTION TIME: BASIC LINUX FUNCTIONS

```
time ./a.out
real 0m10.695s
user 0m0.001s
sys 0m0.006s
```

REAL : walltime
USER : cpu time in user space
SYS : cpu time in OS

IO → SYS time
NETWORK → WALLTIME

In a script

```
start_time=$(date +"%s")
...
end_time=$(date +"%s")
walltime=$(($end_time-$start_time))
echo "walltime $walltime"
```

MEASURE PERFORMANCE WITHIN THE CODE FOR BETTER ACCURACY

The programmer must be aware though that these methods are intrusive, and introduce overheads to the code

Serial code

- Fortran90

cputime(), system_clock(), data_and_time()

- c/c++

clock()

```
#include <time.h>
clock_t time1, time2;
double dub_time;
...
time1 = clock();
for (i = 0; i < nn; i++)
  for (k = 0; k < nn; k++)
    for (j = 0; j < nn; j ++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
time2 = clock();
dub_time = (time2 - time1)/(double) CLOCKS_PER_SEC;
printf("Time -----------------> %lf \n", dub_time);
```

Parallel code

- MPI

  double MPI_Wtime()

- OpenMP

  double omp_get_wtime()

EVERYTHING AND MORE COUNTS FOR PARALLEL PERFORMANCE

- Partitioning/decomposition
- Communications
- Multithreading
- Synchronisation

…And measuring time alone sometimes is not sufficient to understand what is going on

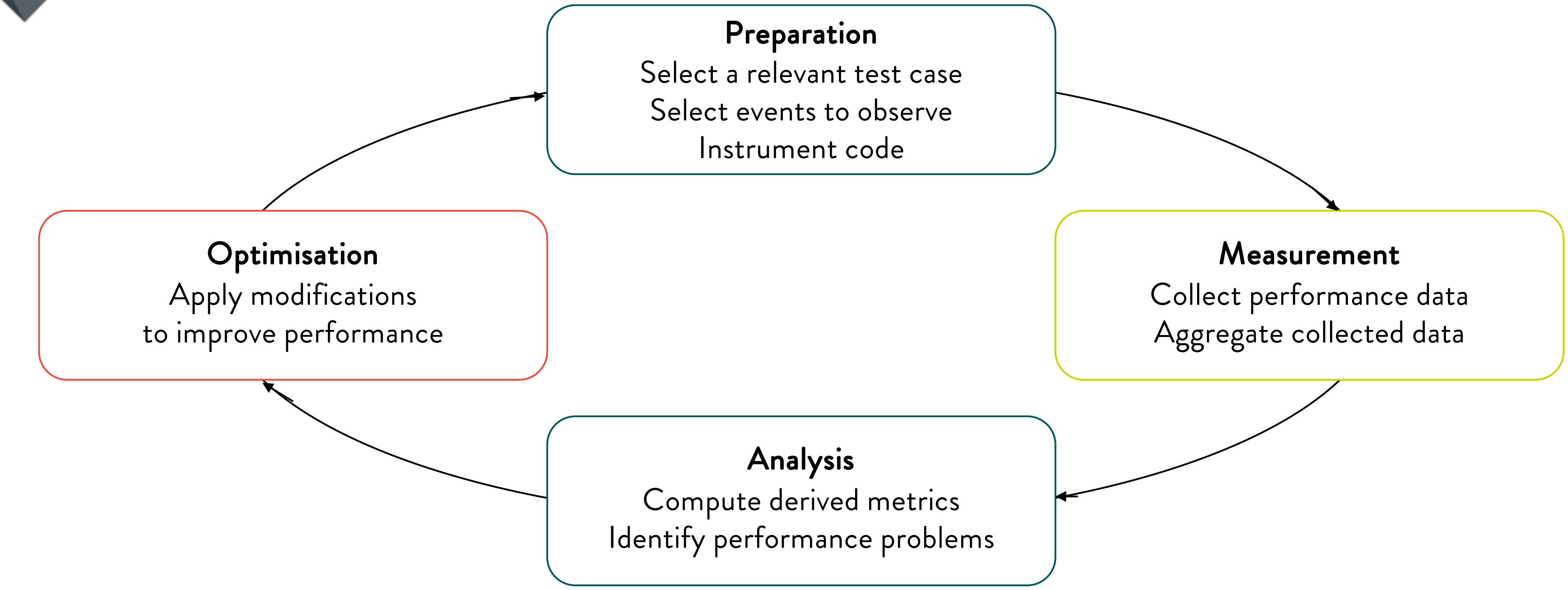
+

Inserting time commands in the source is tedious and not without overheads.
There may also be problems of portability between architectures and compilers.

BETTER TO USE A PROFILING TOOL

- gprof → works for serial code
- Intel VTune
- Extrae
- Vampir
- NVIDIA Nsight → works well for NVIDIA GPU
- **Score-P**

THE PERFORMANCE ENGINEERING WORKFLOW



Preparation

Select a relevant test case

Select events to observe

Instrument code

Measurement

Collect performance data

Aggregate collected data

Analysis

Compute derived metrics

Identify performance problems

Optimisation

Apply modifications to improve performance

TIP: APPLY THE 80/20 RULE

- Programs typically spend 80% of their time in 20% of the code
- Programmers typically spend 20% of their effort to get 80% of the total speedup possible for the application (e.g. know when to stop)
- Do not optimise things that do not matter
- “If you optimise everything, you will always be unhappy” (D. E. Knuth)

WHAT ARE MEASURABLE METRICS?

- Count of how often an event occurs
  e.g. number of MPI point-to-point messages sent
- Duration of some interbal
  e.g. the time spent these send calls
- Size of some parameter
  e.g. number of bytes transmitted by these calls
- Examples: execution time, number of function calls, CPU cycles per instruction, FLOPS

A NOTE ABOUT EXECUTION TIME

- Wall-clock time
  Includes waiting time: I/O, memory, other system activities
  In time-sharing environments also the time consumed by other applications

- CPU time
  Time spent by the CPU to execute the application
  Does not include time the program was context-switched out
  Does not include inherent waiting time (e.g., I/O)
  Portability? What is user, what is system time?

- Problem: Execution time is non-deterministic
  Use mean or minimum of several runs

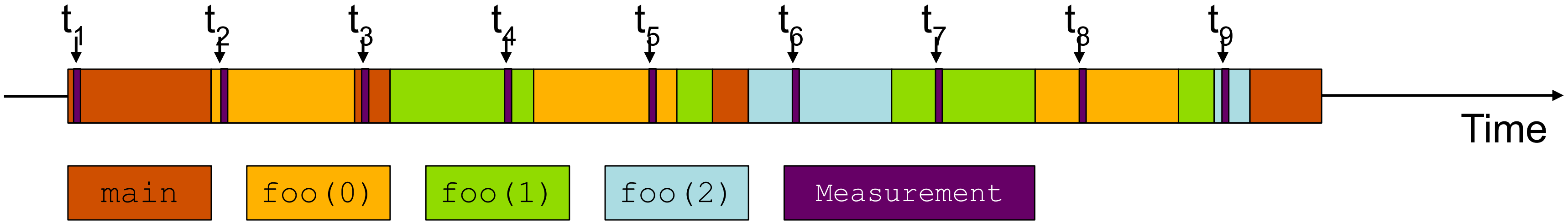
PROFILING OPTIONS

- Type of measurements
  - Sampling
  - Code instrumentation

- Type of performance data recorded
  - Profiling/Runtime summarisation → summarisation of events over execution interval
  - Tracing → chronologically ordered sequence of event records

- Type of performance analysis
  - Online
  - Postmortem

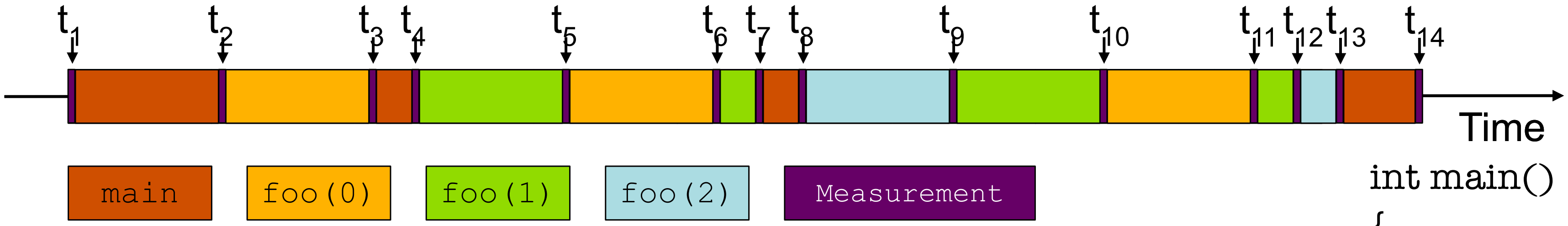
SAMPLING



- 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.
- Works with unmodified executables.
- Negligible overhead
- Require long runs

```
int main()
{
  int i;
  for (i=0; i < 3; i++)
          foo(i);
  return 0;
}
void foo(int i)
{
  if (i > 0)
          foo(i - 1);
}
```

INSTRUMENTATION

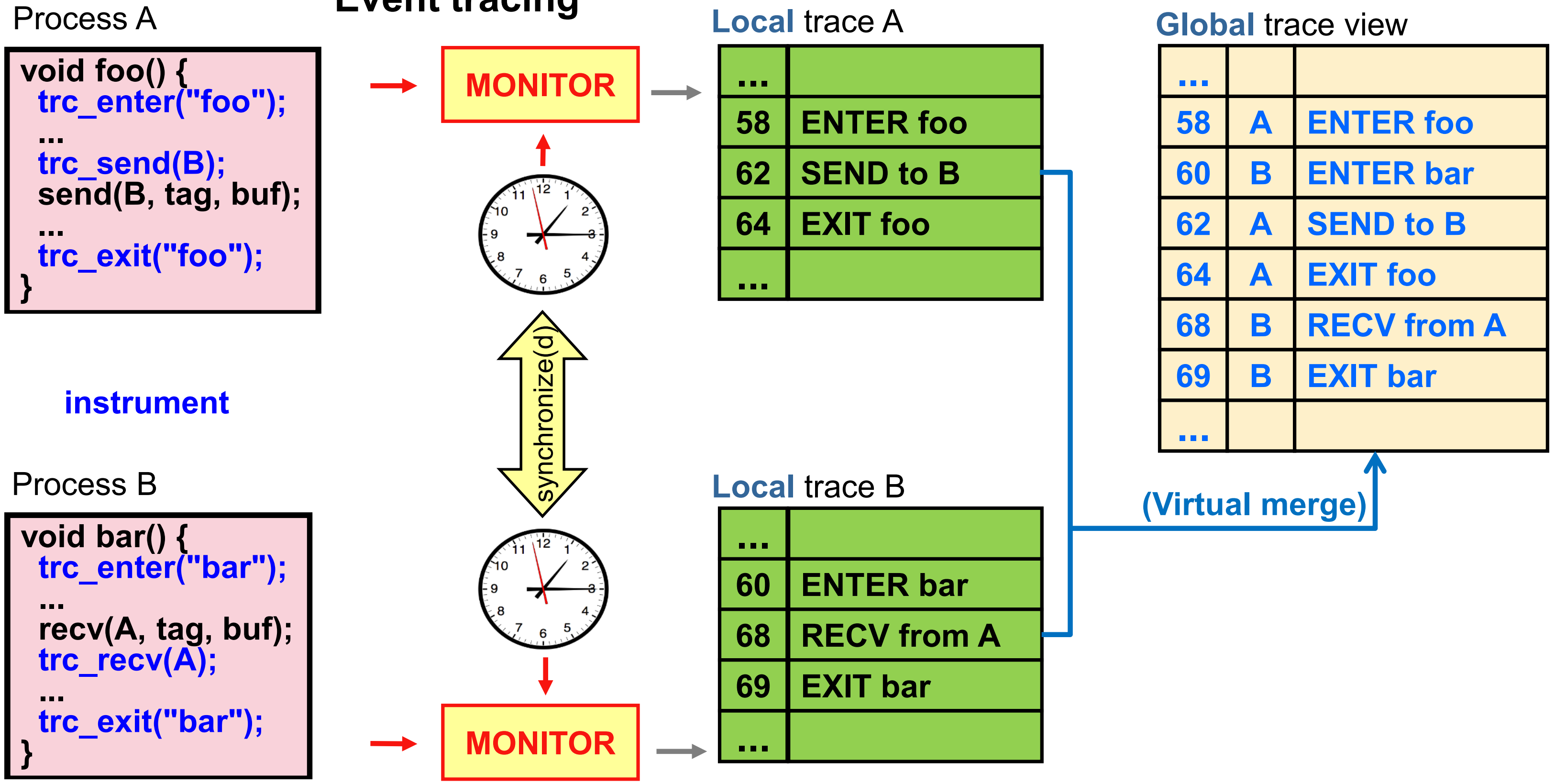


- Special code is added around the region we want to measure
- Provide very detailed information
- Require modifying source code and recompiling it
- Risk for large overheads on short subroutines

```
int main()
{
  int i;
  enter ("main");
  for (i=0; i < 3; i++)
           foo(i);
  leave ("main");
  return 0;
}
void foo(int i)
{
  enter("foo");
  if (i > 0)
           foo(i - 1);
  leave("foo");
}
```

TRACING

Event tracing



Process A

```
void foo() {
  trc_enter("foo");
  ...
  trc_send(B);
  send(B, tag, buf);
  ...
  trc_exit("foo");
}
```

instrument

Process B

```
void bar() {
  trc_enter("bar");
  ...
  recv(A, tag, buf);
  trc_recv(A);
  ...
  trc_exit("bar");
}
```

Local trace A

...	
58	ENTER foo
62	SEND to B
64	EXIT foo
...	

Local trace B

...	
60	ENTER bar
68	RECV from A
69	EXIT bar
...	

Global trace view

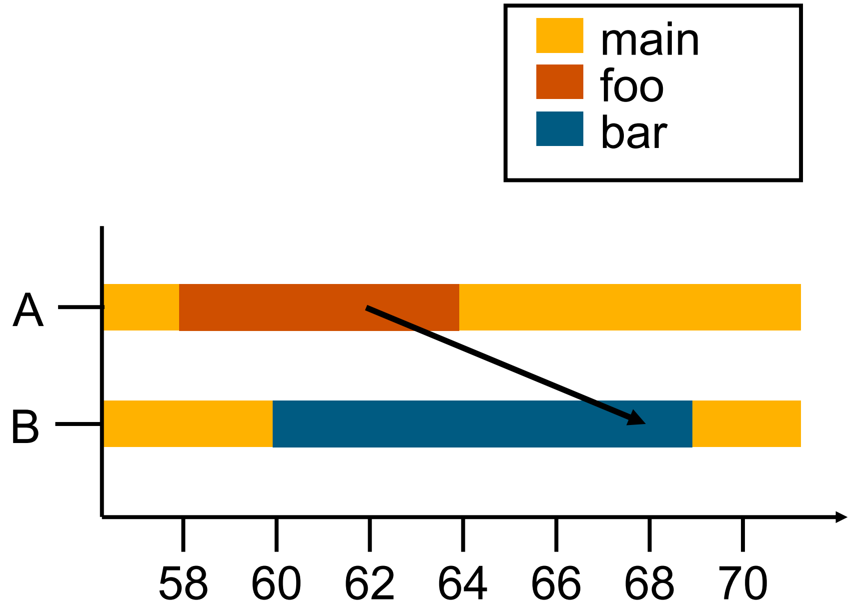
...		
58	A	ENTER foo
60	B	ENTER bar
62	A	SEND to B
64	A	EXIT foo
68	B	RECV from A
69	B	EXIT bar
...		

TRACING: EXAMPLE OF VISUALISATION

Global trace view

...		
58	A	ENTER foo
60	B	ENTER bar
62	A	SEND to B
64	A	EXIT foo
68	B	RECV from A
69	B	EXIT bar
...		

Post-Mortem Analysis



GPROF

- GNU profiler gprof
- Open-source
- Uses time Based Sampling: at intervals the “program counter” is interrogated to decide at which point in the code the execution has arrived.

```
gcc/g++/gfortran –pg source_code -o exectubale.exe
./executable.exe
gprof ./executable.exe gmon.out
```

PROFILING WITH GPROF

1. inspect the Makefile
2. compile with make
3. run the instrumented binary with input parameters 128 30 and postprocess the binary profile

1. Which is the most time-consuming routine ?
2. How much time is spent in MAIN_ exclusively ?
3. Are there routines contributing to _MAIN with negligible exclusive time ?
4. Are there routines with more than one calling path ?
5. Repeat by increasing the optimization flag. Are there changes in the flat summary ?

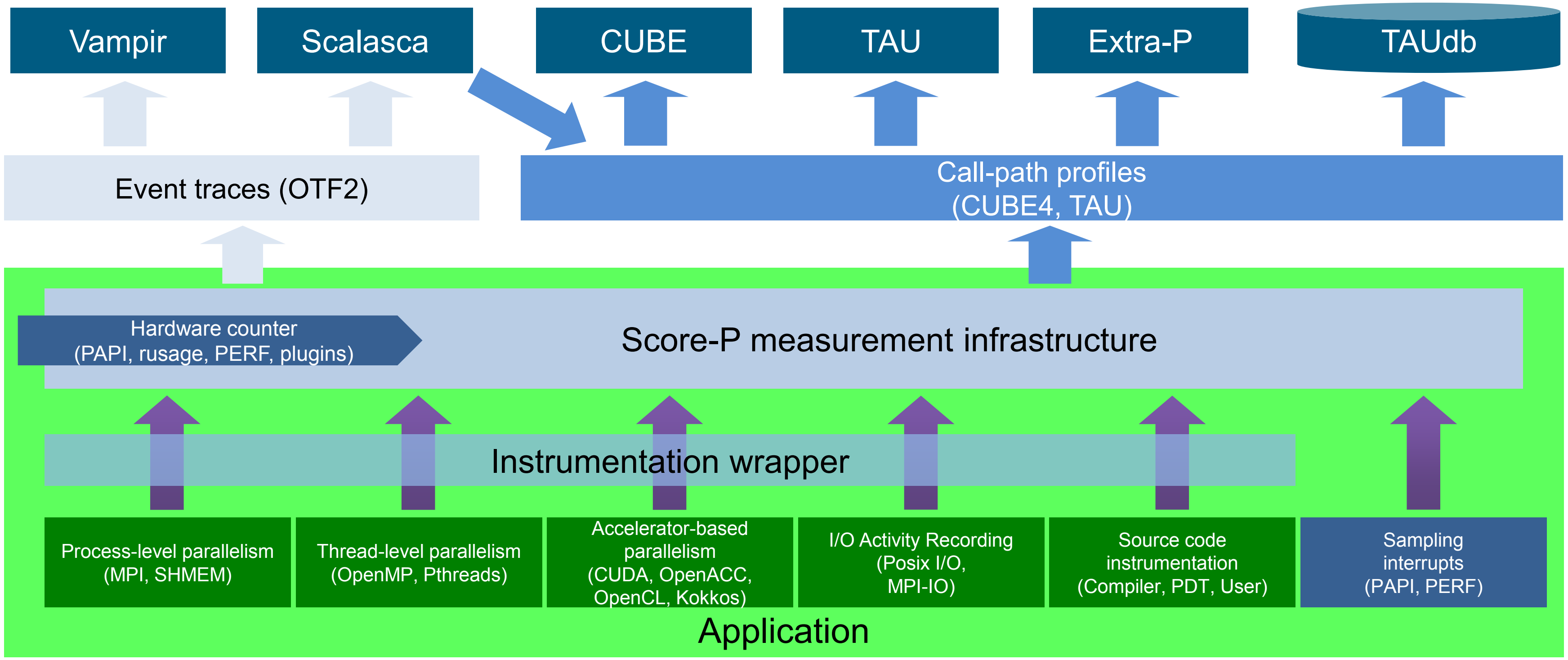
Adapted from https://gitlab.hpc.cineca.it/training/profiling-tutorial-mhpc

SCORE-P

Score-P

- Infrastructure for instrumentation and performance measurements
- Instrumented application can be used to produce several results:
  - Call-path profiling: CUBE4 data format used for data exchange
  - Event-based tracing: OTF2 data format used for data exchange
- Support the main parallel paradigms (MPI, OpenMP, CUDA, OpenACC, …)
- Open-source
- Portable and scalable to all major HPC systems
- Developed by the Virtual Institute – High Productivity Supercomputing
  https://www.vi-hps.org/about/about.html
  One of the main partner is Forschungszentrum Jülich, Germany

SCORE-P INFRASTRUCTURE



INSTALLING SCORE-P

```
module purge
module load nvidia/nvhpc/25.7
wget https://perftools.pages.jsc.fz-juelich.de/cicd/scorep/tags/scorep-9.3/scorep-9.3.tar.gz
tar –xvf scorep-9.3.tar.gz
./configure \
--prefix=<path_to_install>\
--with-machine-name=ngnode02 --with-nocross-compiler-suite=nvhpc \
--with-libcudart-include=/opt/share/sdk/nvidia/Linux_aarch64/25.7/cuda/12.9/include \
--with-libcudart-lib=/opt/share/sdk/nvidia/Linux_aarch64/25.7/cuda/12.9/lib64 \
--with-shmem=no --with-libgotcha=download
make
make install
```

Note `path_to_install` should be something like
`/home/usera/prod/opt/applications/scorep/9.3/Linux_aarch64/nvhpc-25.7`

Now create a module (example to adapt in the repo)

TESTING SCORE-P 1/2

- We will test SCORE-P with the BT-MZ NAS benchmark (MPI + OpenMP)
- Original source: http://www.nas.nasa.gov/Software/NPB
- Available in the GitHub repo (NPB3.3-MZ-MPI)
- Test solves a discretized version of the unsteady, compressible Navier-Stokes equations in three spatial dimensions
- Test performs 200 time-steps on a regular 3-dimensional grid
- Code implemented in 20 or so Fortran77 source modules

TESTING SCORE-P 2/2

- Step 1: Test the code without Score-P
  - Inspect config/make.def
  - Build the test case with `make bt-mz CLASS=C NPROCS=8`
  - Adapt jobscript and launch or launch it interactively, but export OMP options
  - Use 8 MPI tasks and 6 OMP threads (bind OMP threads to core)
  - Inspect results

- Step 2: Test the code with Score-P
  - Load the scorep module
  - Change the compilation option in config/make.def to use Score-P
  - MPIF77 = scorep --user mpifort o MPI77 = scorep-mpifort
  - Build the test case with `make clean && make bt-mz CLASS=C NPROCS=8`
  - Check possible Score-P measurements with `scorep-info config-vars --full`
  - Enable Score-P measurements (for now only directory where to store results)
  - Launch jobscript or launch code interactively, but export relevant environmental variables

ANALYSING RESULTS WITH CUBE

Install CUBE on your local machine
https://www.scalasca.org/scalasca/software/cube-4.x/
(Use the binary if possible)

Check the directory created by running the code with SCORE-P. It contains:

- A brief content overview (MANIFEST.md)
- A record of the measurement configuration (scorep.cfg)
- The analysis report that was collated after measurement (profile.cubex)

Inspect the first two files with cat
Download the third file to be open with CUBE