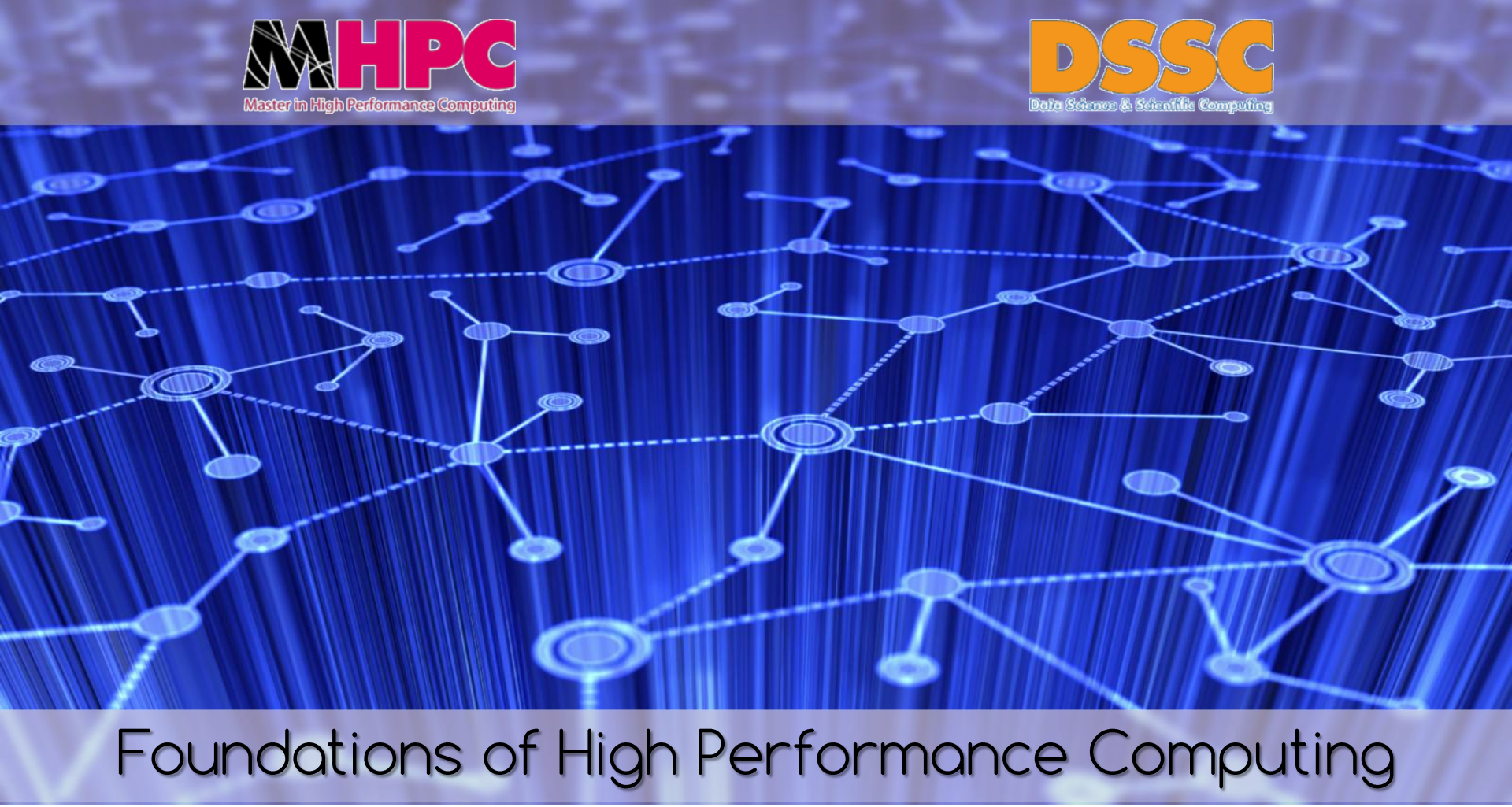


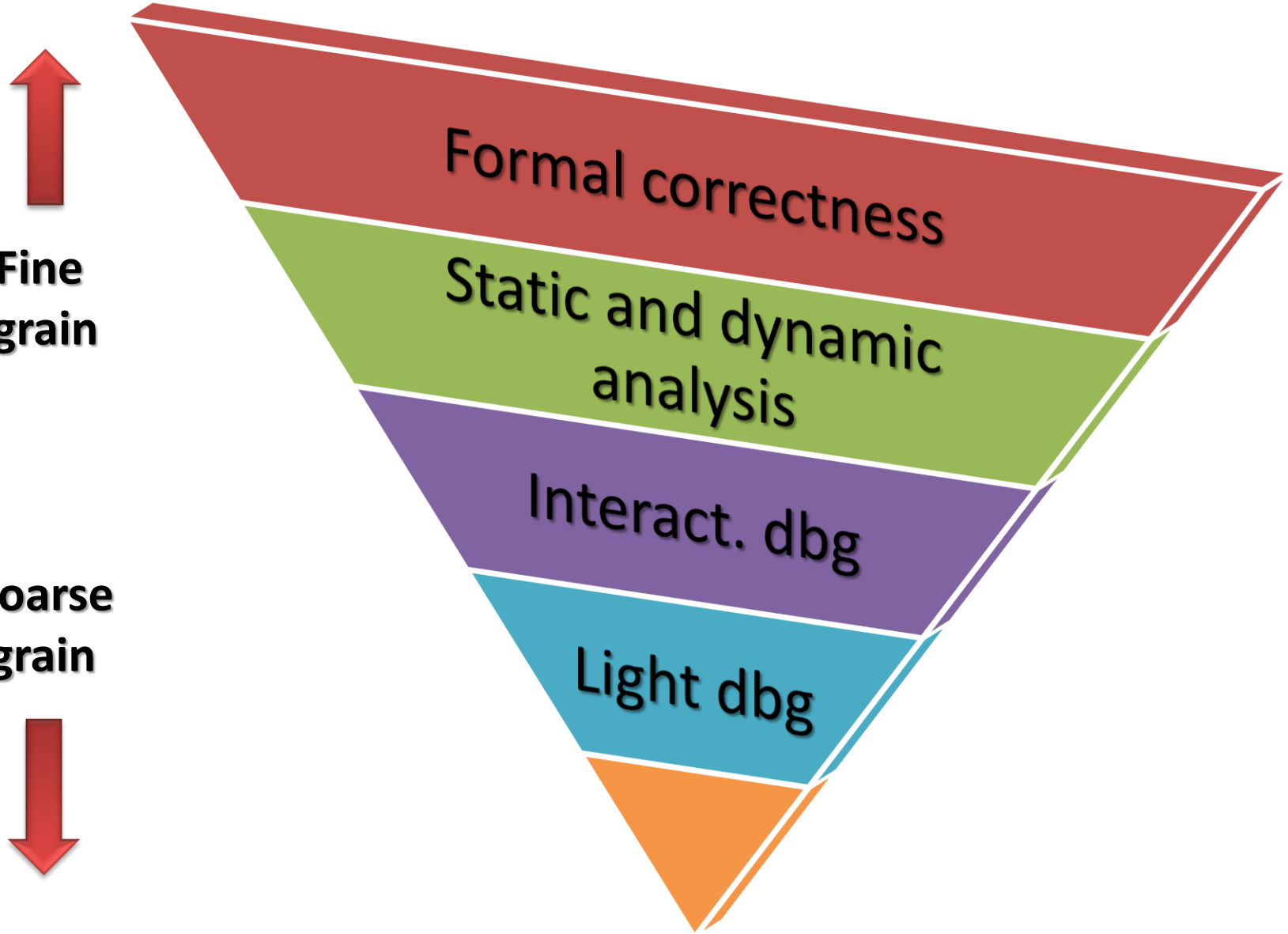
Parallel Programming

Parallel debugging and Profiling

Outline

1. **Debugging**, a quick recap
2. Using **gdb in parallel**
3. Open tools to profile, debug and understand your parallel code

Correctness tools



Correctness tools

Formal correctness

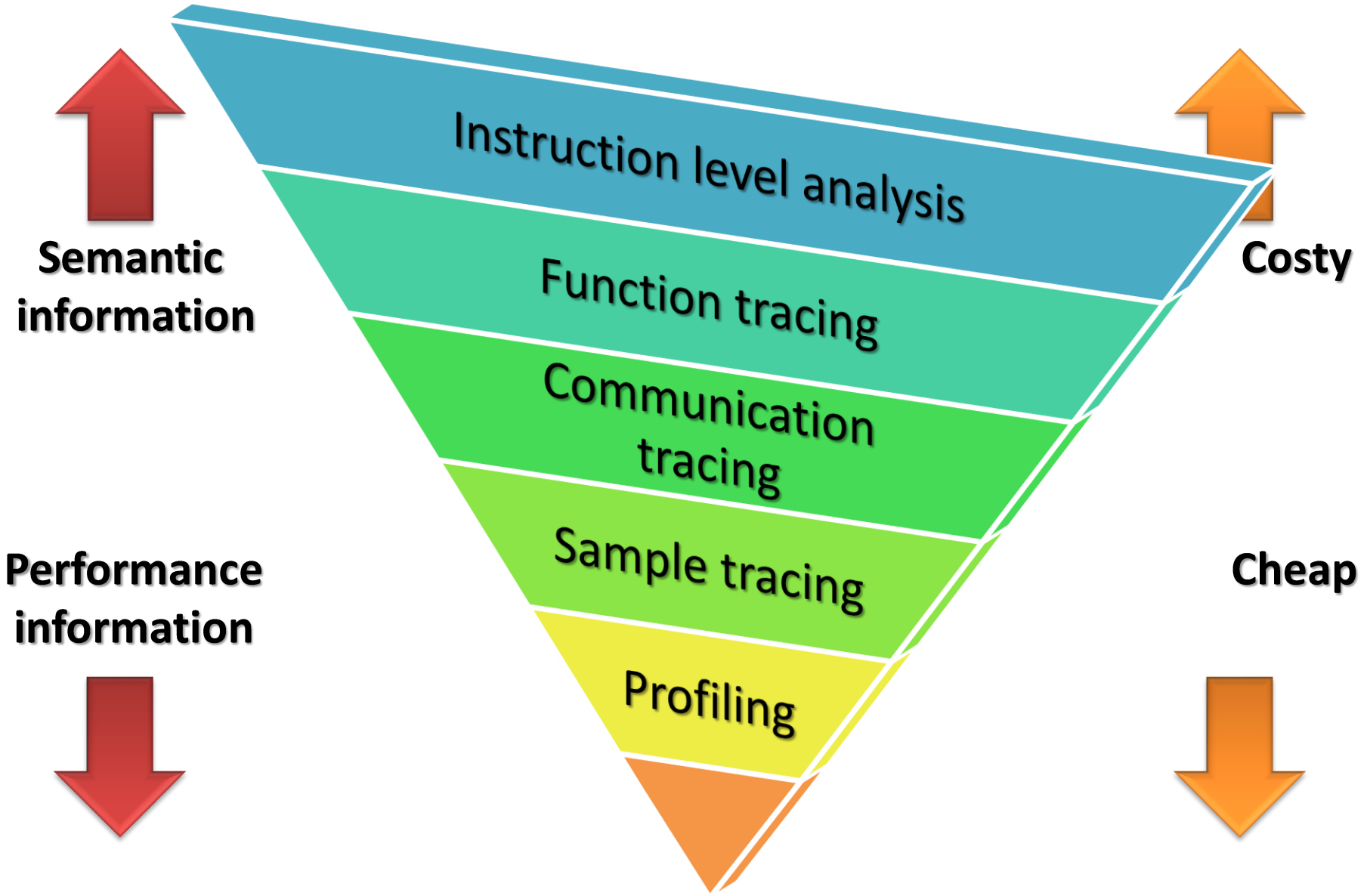
- Formal verification of your code
- Spots logical design errors in parallelism
- Spots possible deadlocks, race conditions, false sharing
- …

Dynamic analysis

Example:

- Valgrind memcheck → memory problems
- Valgrind helgrind → data race conditions

Performance tools



Performance tools

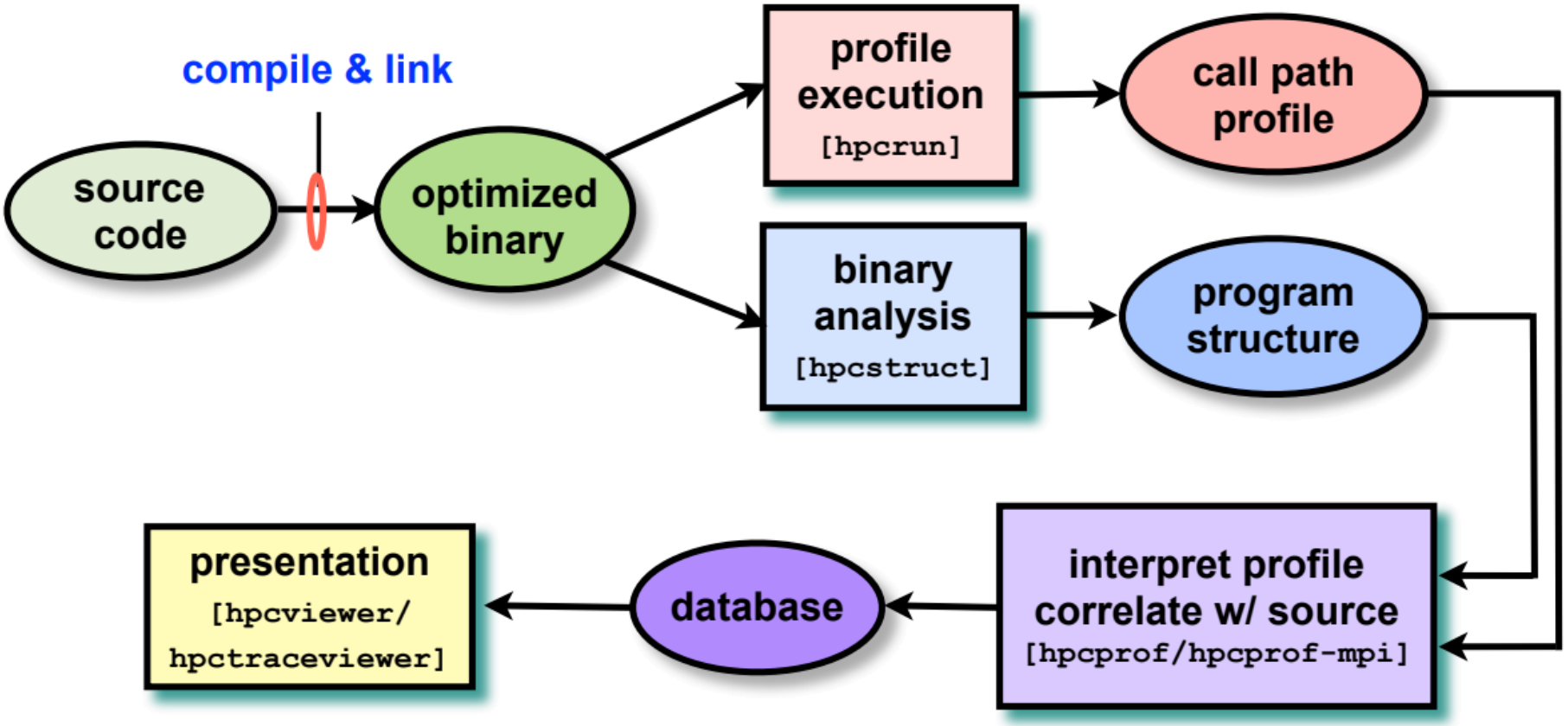
Instruction-level analysis

- Use binary instrumentation tools, like valgrind
- Measure **dynamic behaviour** of the code

focus on:

- Instruction kaleidoscope: FP, Int, branching, memory access
- FP per Byte →computational intensity
- Vectorization efficiency
- ICP / CPI
- Cache optimization
  - Misses
  - Reusing (memory intensity)
  - …

Performance tools



Debugging basics

Debugging is the process of finding and fixing flaws in software

Debugging basics : tools

1. Don't insert bugs

   Highly encouraged, but never works

2. Add print statements everywhere

   Highly discouraged, but sometimes it works

3. Command line based debuggers
   1. **gdb**: the gnu debugger
   2. **idb**: the intel debugger
4. Debuggers with GUI
   1. **ddd**: Gnu data display debugger
   2. **IDEs**: eclipse, NetBeans, emacs+gdb, …
   3. **DDT**: ARM/Allinea tool
   4. **TotalView**, …others…

gdb, your friend

You already know about gdb and its basic use.

RECAP: a **debugger** is a program that enables you to take the control of the execution of another program.

gdb is the GNU debugger.

gdb, your friend

- Crash inspections, through core files
- Function call stack
- Stepping through execution flow
- Automatic stopping at specific locations or conditions
- Watching for variables

- Use of GUI may be preferred when working locally
- On remote machines, it might be slower or graphics may not be available

Outline

1. Debugging, a quick recap
2. Using **gdb in parallel**
3. Open tools to profile, debug and understand your parallel code

Parallel debugging

The problem is much more complex.
The fundamental additional challenge is the simultaneous execution.

Shared memory paradigm: OpenMP, pthreads

- Multiple threads
- Shared vs private memory regions
- Race conditions

Message-passing paradigm: MPI

- Multiple independent processes (+ possible multithread)
- Communication
- Deadlocks

→ Commercial debuggers
→ Free debuggers, to some extent

gdb, multi-thread capability

```
void *ShowUp(void *thread_id)
{
  printf("Thread #%zd says: \" Hello World! \"\n", (size_t)thread_id);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NTHREADS];
   int  rc;
   size_t t;

   for(t = 0; t < NTHREADS; t++)
     {

       printf("Creating thread %zd\n", t);
       rc = pthread_create( &threads[t], NULL, ShowUp, (void *)t);

       if (rc)
           {
             printf("return code from pthread_create() is %d\n", rc);
             exit(-1);
           }
   }
   pthread_exit(NULL);
}
```

gdb, multi-thread capability

Let's have a try:

```
:~$ gcc -g -o my_threadprog  my_threadprog.c -lpthread

:~$ gdb ./my_threadprog
```

gdb, multi-thread capability

```
:~$ gdb ./my_threadprog
Reading symbols from my_threadprog...done.

(gdb) r

(gdb) Starting program: my_threadprog

[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Creating thread 0
[New Thread 0x7ffff77ef700 (LWP 24144)]
Thread #0 says: " Hello World! "
Creating thread 1
[New Thread 0x7ffff6fee700 (LWP 24145)]
Thread #1 says: " Hello World! "
Creating thread 2
[New Thread 0x7fffeffff700 (LWP 24146)]
Thread #2 says: " Hello World! "
Creating thread 3
[New Thread 0x7ffff65d6700 (LWP 24147)]
Thread #3 says: " Hello World! "
[Thread 0x7fffeffff700 (LWP 24146) exited]
[Thread 0x7ffff6fee700 (LWP 24145) exited]
[Thread 0x7ffff77ef700 (LWP 24144) exited]
[Thread 0x7ffff7fac700 (LWP 24140) exited]
[Inferior 1 (process 24140) exited normally]
(gdb) Quit
(gdb)
```

gdb, multi-thread capability

It is necessary to explicitly set up gdb for multi-thread debugging

```
(gdb) set pagination off
(gdb) target-async on
(gdb) non-stop on
```

In **`all-stop`** mode, whenever the execution stops, *all* the threads stop (wherever they are).
Whenever you restart the execution, *all* the threads re-start: however, gdb can not single-step all the threads in the steplock. Some threads may execute several instructions even if you single-stepped the thread under focus with *step* or *next* commands.

`non-stop` mode means that when you stop a thread, all the other ones continue running until they finish or they reach some breakpoint that you pre-defined

→ live demo

PARALLEL DEBUGGING

A good approach:

- Write a (possibly good) code natively parallel but able to run in serial, which means with 1 MPI task
- Profile, debug and optimize that code first
- If multi-threaded, deal with thread sync / races with 1 MPI task
- Deal with communications, synchronization and race/deadlock conditions on a *small* number of MPI tasks
- Profile, debug and optimize communications on a *small* number of MPI tasks
- Try a full-size run

gdb, with MPI / 1

It is still possible to use gdb directly, called from mpirun:

```
:~$ mpirun -np <NP> -e gdb ./program
```

Because gdb has multi-thread capability.
However, depending on your system that may not properly work with MPI.

gdb, with MPI / 2

The simplest way to use gdb with a parallel program is :

```
:~$ mpirun -np <NP> xterm -hold -e gdb ./program
```

Which launches <NP> xterm windows with running gdb processes in which you can run

```
(gdb) run <arg_1> <arg_2> … <arg_n>
```

..**most likely *this* will not work**, because HPC environments are hostile to *X* for several reasons.

gdb, with MPI / 3

Another handy possibility is to open as many connections as processes on different terminals on your local machine, and *attach* gdb to the already running MPI processes

```
:~$ mpirun -np <NP> ./program
```

Followed by:

```
:~$ gdb -p <PID_of_MPI_task_n>
```

For each MPI task you want to follow.

gdb, with MPI / *3*

You still have **2 issues**:

1. ***Where*** to run gdb, if `xterm` is not available and you do not want to use it in multi-thread mode ?

   You may consider using `screen` (practical example in few minutes)

2. ***How*** the MPI tasks should be convinced to wait for gdb to step in ?

   → *Next slides*

gdb, in parallel / 4

NOTE

A possible issue for **attacching** gdb to a running process is that you **may not have the capability to do that** on a Linux system.

Look in the `/proc/sys/kernel/yama/ptrace_scope` file…

gdb, in parallel / 4

NOTE

A pos hat
you m em.

Look i ..

```
0 ("classic ptrace permissions")
       No additional restrictions on operations that perform
       PTRACE_MODE_ATTACH checks (beyond those imposed by the
       commoncap and other LSMs).

       The use of PTRACE_TRACEME is unchanged.

1 ("restricted ptrace") [default value]
       When performing an operation that requires a
       PTRACE_MODE_ATTACH check, the calling process must either have
       the CAP_SYS_PTRACE capability in the user namespace of the
       target process or it must have a predefined relationship with
       the target process.  By default, the predefined relationship
       is that the target process must be a descendant of the caller.

       A target process can employ the prctl(2) PR_SET_PTRACER
       operation to declare an additional PID that is allowed to
       perform PTRACE_MODE_ATTACH operations on the target.  See the
       kernel source file Documentation/admin-guide/LSM/Yama.rst (or
       Documentation/security/Yama.txt before Linux 4.13) for further
       details.

       The use of PTRACE_TRACEME is unchanged.

2 ("admin-only attach")
       Only processes with the CAP_SYS_PTRACE capability in the user
       namespace of the target process may perform PTRACE_MODE_ATTACH
       operations or trace children that employ PTRACE_TRACEME.

3 ("no attach")
       No process may perform PTRACE_MODE_ATTACH operations or trace
       children that employ PTRACE_TRACEME.
```

gdb, in parallel / 4

NOTE

Solutions:

1. Get the capability
2. As root type:
   `echo 0 > /proc/sys/kernel/yama/ptrace_scope`
3. Set the `kernel.yama.ptrace_scope` variable in the file `/etc/sysctl.d/10-ptrace.conf` to `0`

The last solution turns off the security measure permanently, it is not a good idea (at least on a facility)

gdb, in parallel / 5

We are left with the problem of *attaching* the gdb to a running process (or several running processes).

How can we do that ?

There is a classical trick, that requires to insert some small additional code in your program

```
int wait = 1

while(wait)
   sleep(1);
```

The MPI processes will wait indefinitely until you do not change the value of wait *from inside dbg attached to each process.*

gdb, in parallel / 6

Let's say that you MPI program start with:

```
int main(int argc, char **argv)
{
  int Me, Size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD,
&Me);
  MPI_Comm_size(MPI_COMM_WORLD,
&Size);

   …
```

and that you insert the following code snippets right after it →

Attaching gdb…

```
#ifdef DEBUGGER
  int   Wait = 1;
  pid_t my_pid;
  char  my_host_name[200];

  gethostname(my_host_name, 200);
  my_pid = getpid();

  for(int i = 0; i < Size; i++)
   {
     if(i == Me)
        printf("task with PID %d on host %s is waiting\n", my_pid, my_host_name);
      MPI_Barrier(MPI_COMM_WORLD);
   }

  while(Wait)
   sleep(1);

  MPI_Barrier(MPI_COMM_WORLD);
#endif
```

Attaching gdb…

Some more flexibility

```
char *env_ptr;
 if( ( (env_ptr = getenv("DEBUG_THIS")) != NULL) &&
    (strncasecmp(env_ptr, "YES", 3) == 0) )
  {
    int   Wait = 1;
    pid_t my_pid;
    char  my_host_name[200];

    gethostname(my_host_name, 200);
    my_pid = getpid();

    < ... >
    while(Wait)
       sleep(3);

    MPI_Barrier(MPI_COMM_WORLD);
  }
```

Attaching gdb…

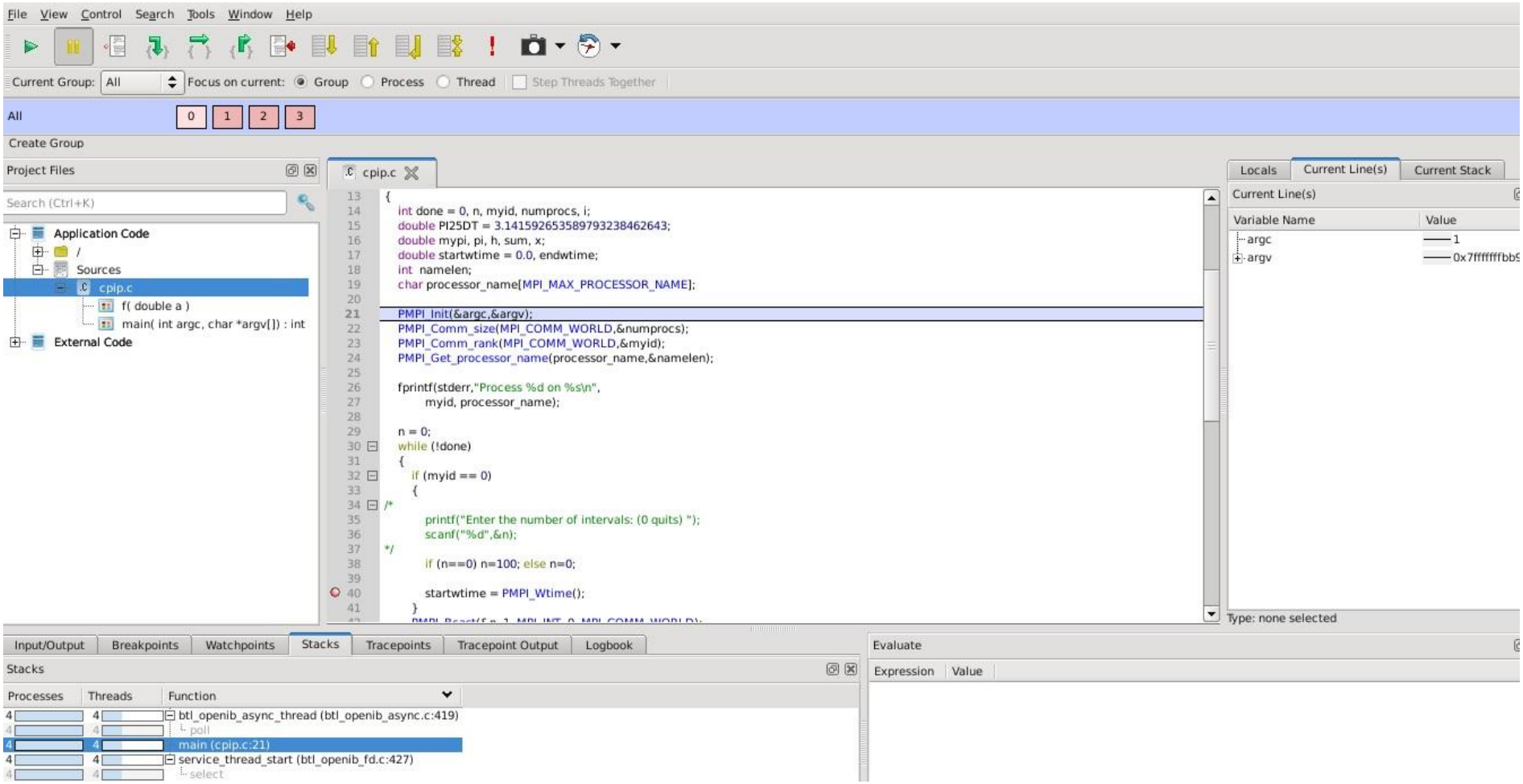
Even more flexibility

```
char *env_ptr;
 if( ( (env_ptr = getenv("DEBUG_THIS")) != NULL) &&
    (strncasecmp(env_ptr, "YES", 3) == 0) )
  {
    int   Wait = 1;
    pid_t my_pid;
    char  my_host_name[200];

    gethostname(my_host_name, 200);
    my_pid = getpid();
    < ... >
    if(me == 0)
      while(Wait)
         sleep(3);

    MPI_Barrier(MPI_COMM_WORLD);
  }
```

All the MPI tasks but the 0th will wait at the barrier. This way, you can avoid to attach to *all* the tasks and unlock only the 0th .

What if you have do debug on 100k processes ?

For instance, use ARM/Allinea DDT…



Outline

1. Debugging, a quick recap
2. Using gdb in parallel
3. Open tools to profile, debug and understand your parallel code

The parallel ecosystem

Understanding the details of a large parallel/multi-threaded code is not an easy task.

BUGS

- Bottlenecks
- Inefficiencies
- Deadlocks
- Race conditions
- Errors in memory addressing

All usually have subtle dependencies on the specific run you perform

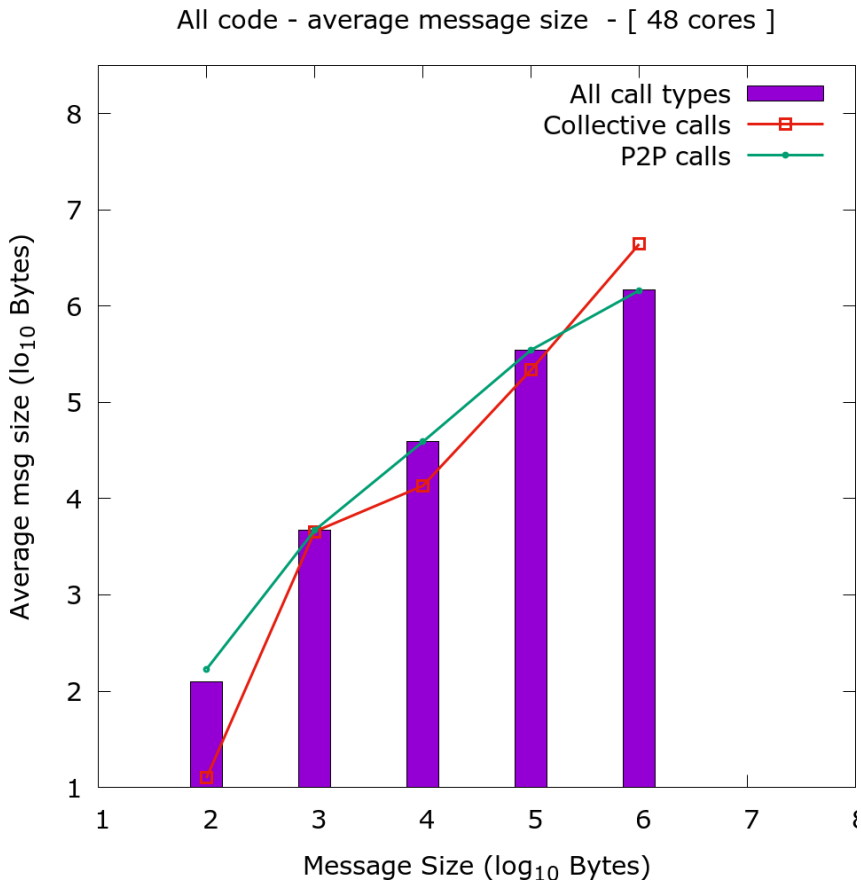
(those that do not are more easily found and fixed)

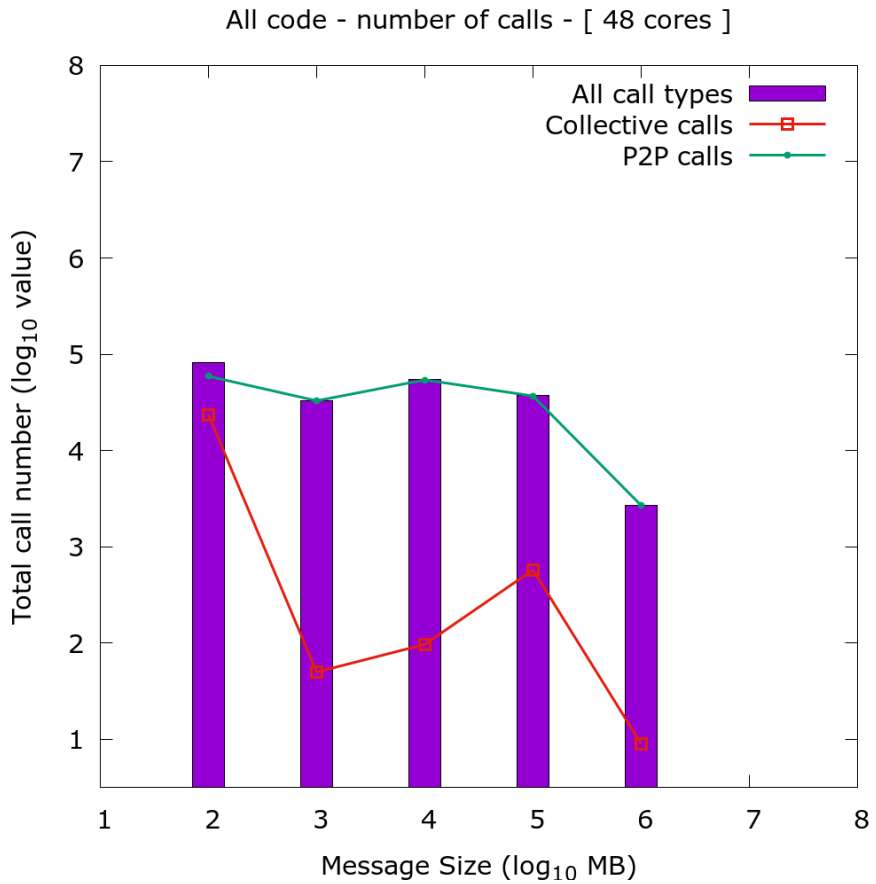
OPTMIZATION

- Cache inefficiencies
- Loop optimizations
- Performance counters

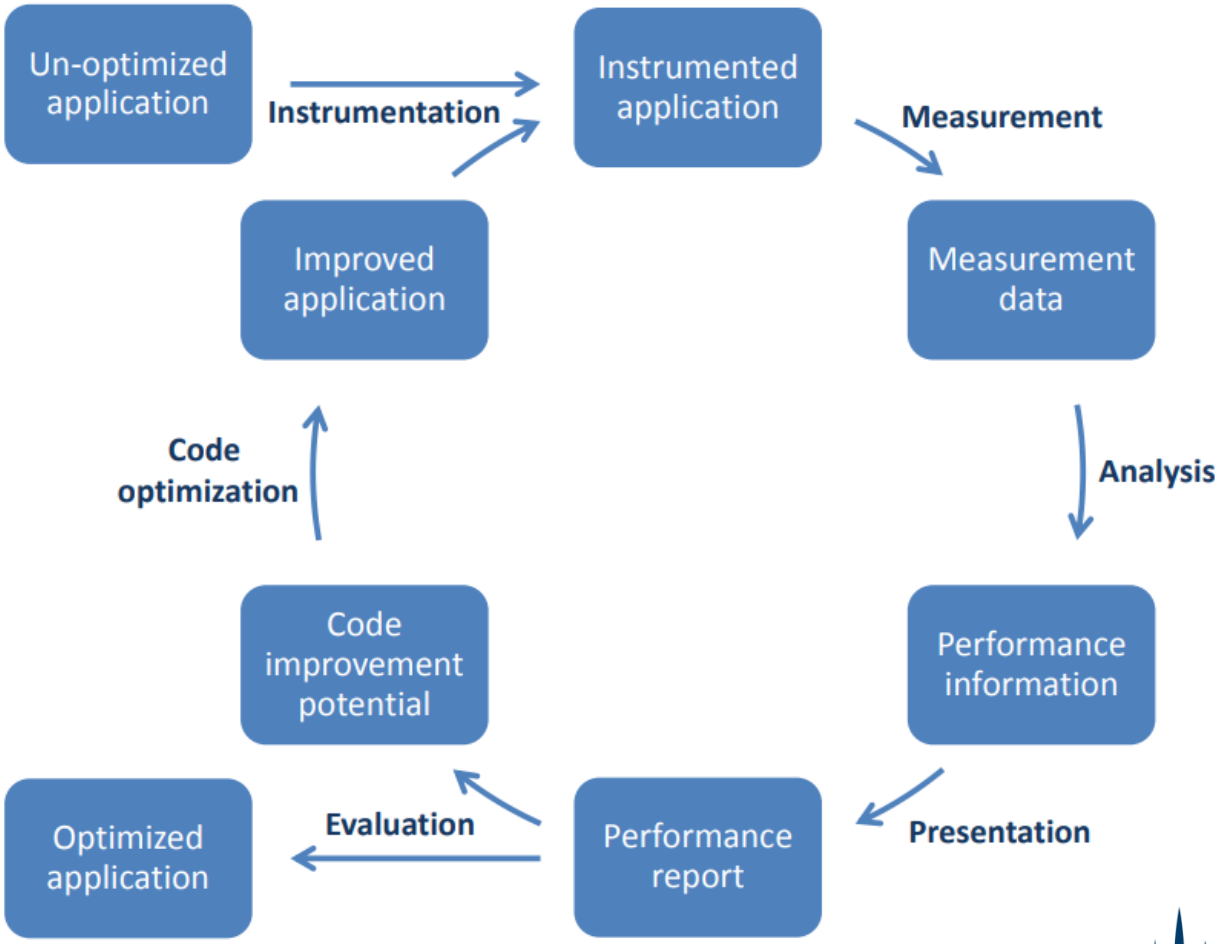
Very tough to trace and profile by hands
(sometimes still unavoidable)

A small example of a communication profiling



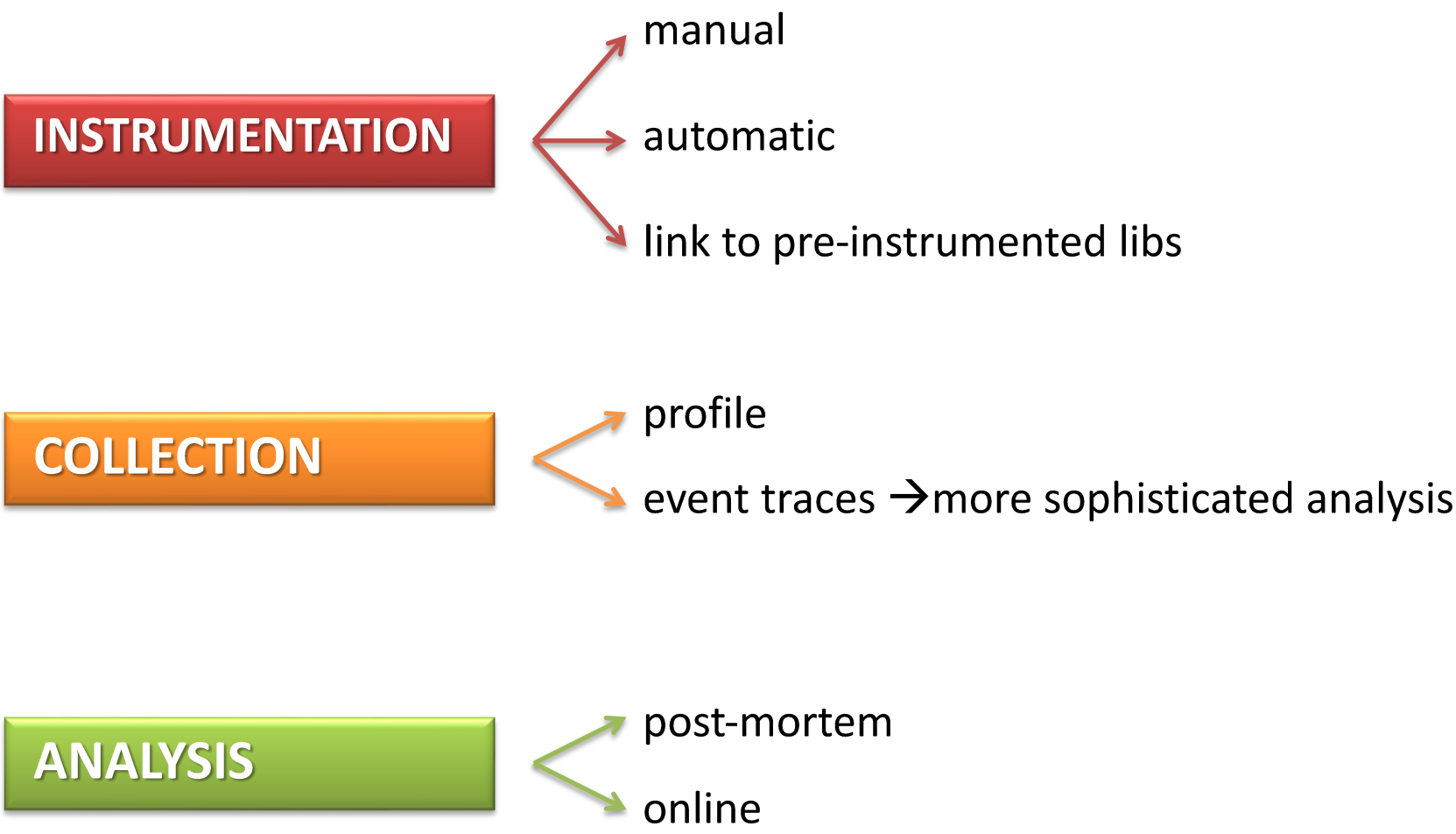


The parallel ecosystem



Score-P
Scalable performance measurement infrastructure for parallel codes

The parallel ecosystem



The parallel ecosystem

There are several very sophisticated open tools, that are mainly outcomes of project research on high-productivity supercomputing.

Many of them are clustered in “ecosystems”, and several of those ecosystems, or some of their components, can talk each other

MpiP

Score-P
Scalable performance measurement infrastructure for parallel codes

scalasca

Virtual Inst – High productivity supercomp.

HPCToolkit

Rice University

Open|SpeedShop

Krell Inst., LANL, LLNL, SNL

PTF

Technische Universität München

PAPI

The parallel ecosystem

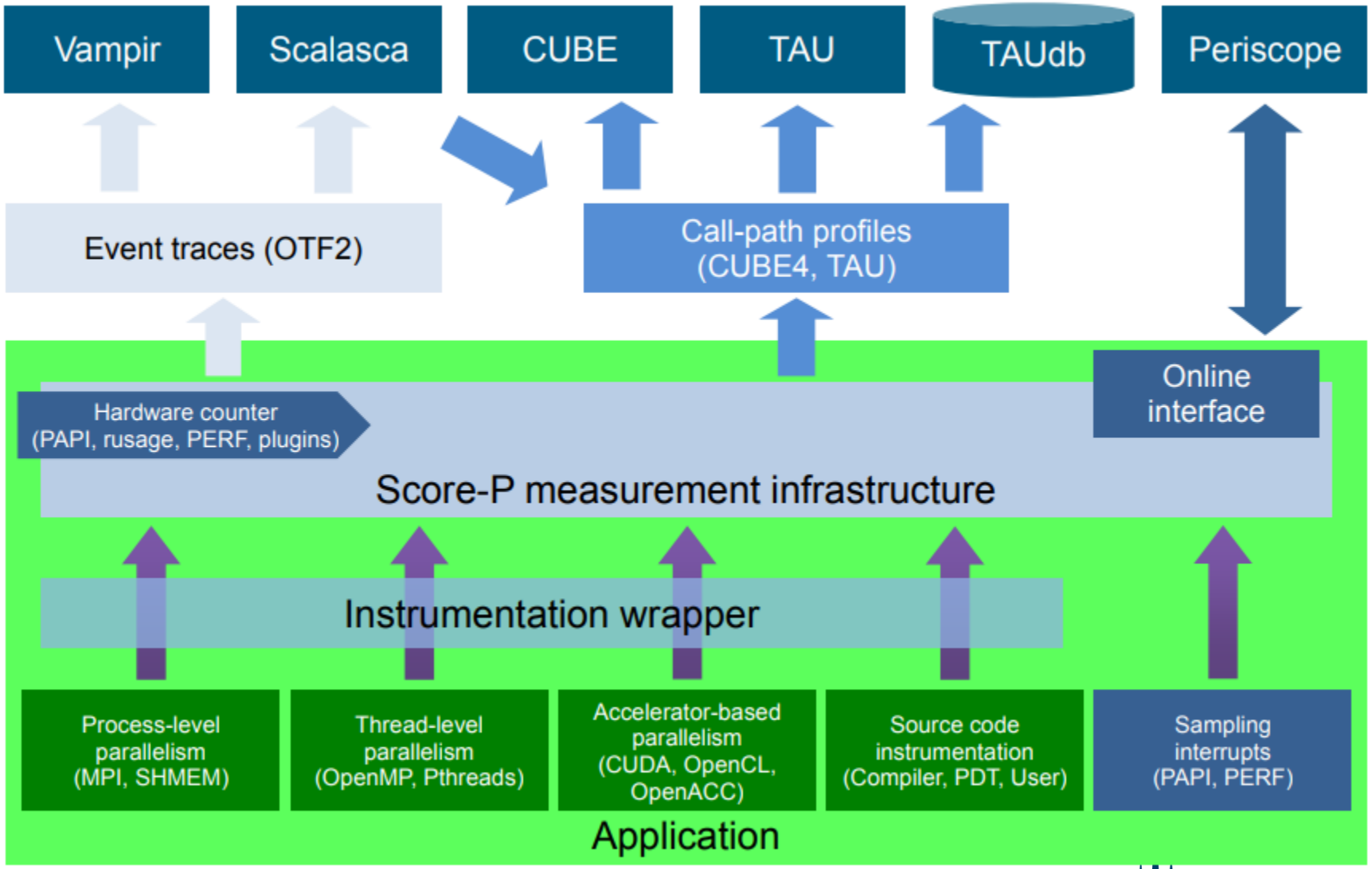
Quick and effective MPI usage profiling: mpiP

```
:~$ mpicc -g prog.c -o prog -lmpiP -lm -lbfd -liberty -lunwind
:~$ mpirun -np <NP> ./prog
```

This will produce a complete report of MPI API usage in your program.

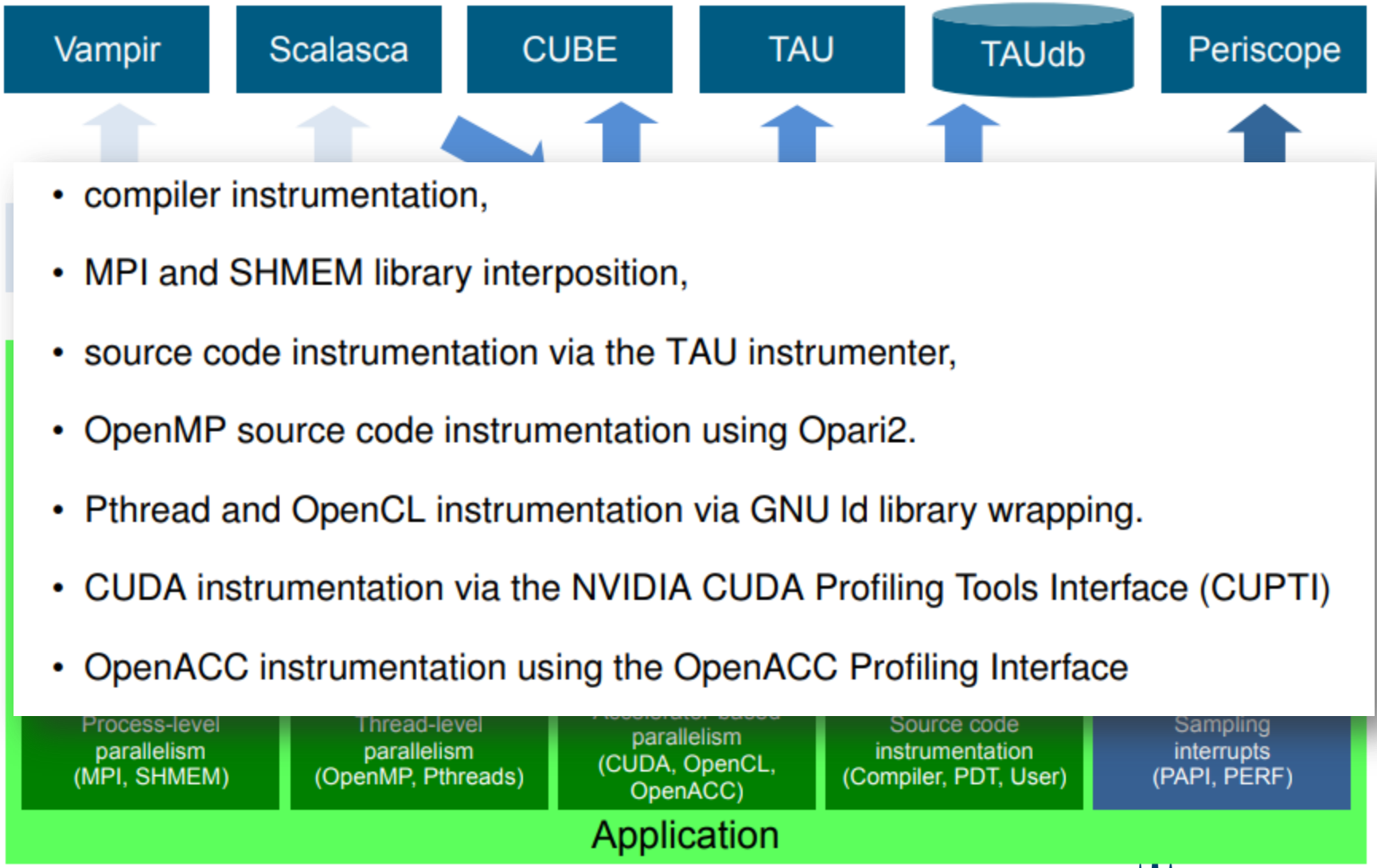
You can control, to some extent, the regions in which the profiling is active from inside your code and the type of report you want

The parallel ecosystem



Score-P
Scalable performance measurement infrastructure for parallel codes

The parallel ecosystem



- compiler instrumentation,
- MPI and SHMEM library interposition,
- source code instrumentation via the TAU instrumenter,
- OpenMP source code instrumentation using Opari2.
- Pthread and OpenCL instrumentation via GNU ld library wrapping.
- CUDA instrumentation via the NVIDIA CUDA Profiling Tools Interface (CUPTI)
- OpenACC instrumentation using the OpenACC Profiling Interface

Score-P
Scalable performance measurement infrastructure for parallel codes

The parallel ecosystem

Score-P offers you a quite flexible measurement infrastructure