DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

CCRT TRAINING

DEBUGGING & PROFILING TOOLS

Loris LUCIDO

Ziad SULTAN

Day 1: Debugging

1. User environment
2. Debugging: Overview
3. Compiler options
4. Debugging with GDB
5. Graphical debugging with DDT
6. Memory debugging

Day 2: Profiling

1. Profiling: overview
2. Simple code profiling
3. ScoreP & Vampir
4. Profiling with Vtune

DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

CCRT TRAINING
DEBUGGING TOOLS

February 24-25 2016

USER ENVIRONMENT

Summary

User Environment

ENVIRONMENT

Basics

- Remote access:

```
ssh login@cobalt.ccc.cea.fr
ssh -Y login@cobalt.ccc.cea.fr
```

- Command line documentation:

```
machine.info
```

- Online documentation:

```
https://www-ccrt.ccc.cea.fr
```

- Shells:
  - Default and highly recommended shell is bash
  - Other available shells: ksh, csh, tcsh, zsh

- Text editors: vi, vim, gedit, emacs, etc.

Available file systems

- Reachable from all clusters of the center:
  - $HOME:
    - Data saved
    - Quota: 3 GB
  - $CCCWORKDIR:
    - Not purged, not saved
    - Quota: 1TB and 500 000 inodes
  - $CCCSTOREDIR:
    - For data archiving for large files. Time for data access can be long
    - Quota: 100 000 files with size range of 1GB - 1TB
- Local to each clusters:
  - $SCRATCHDIR:
    - For intermediate/temporary data issued from computation (can then be migrated to $CCCSTOREDIR if needed)
    - Periodically purged
- Check your data usage on file systems with ccc_quota command

Module command

- module available
  - Display a list of available products
- module load / module unload
  - Set / unset the product environment
- module show <product/version>
  - Display the product description
- module list
  - Display a list of currently loaded modules

User environment

JOB SUBMISSION AND MONITORING

Basics

- Do not run computations on login nodes. Submit them through a batch scheduler (SLURM)
- To submit a job, a submission file must be created
- This file is a shell script containing submission directives (execution time, number of processors requested, etc.) as well as commands needed to launch computation
- With a specific command, you can submit the job to the resource manager which will place it in the appropriate queue (as requested with the directives) until computational resources are available
- When resources are available, the job is launched and the commands found in the submission script are executed

Useful commands

- ccc_msub
  - submit a job
- ccc_mdel
  - delete a job
- ccc_mprun
  - parallel launch interface (srun)
- ccc_mstat / ccc_macct
  - retrieve and display information about a job
- ccc_mpp
  - visualize the state of jobs
- ccc_mpinfo
  - partition information
- ccc_mqinfo
  - QoS information

Job submission

- Submission modes: interactive and batch
  - Batch (highly recommended)

```
ccc_msub script.sh
```

    - The script submitted calls ccc_mprun
  - Interactive (through a Slurm reservation)

```
ccc_mprun -n 2 -q broadwell ./a.out
ccc_mprun -Xfirst -n 1 -c 28 ddt
```

- Use a compute node interactively

```
ccc_mprun -q broadwell -n 1 -x -s
```

- In any case, the hours used are taken from your project

Submission options

- Options are passed to ccc_msub when submitting:

```
-$ ccc_msub -q broadwell -n 1 script.sh
-$ ccc_msub -h
```

- Options can also be passed in the submission script with the #MSUB directives

- WARNING: the ccc_msub options overrule the #MSUB directives

Submission options

- **#MSUB -r name**
  - Job name
- **#MSUB -o std_out / #MSUB -e err_out**
  - Standard or error output
- **#MSUB -n nprocs**
  - Number of processes to start (default = 1)
- **#MSUB -c ncores**
  - Number of cores to allocate per process (default = 1)
- **#MSUB -A ccrt0038**
  - Project ID. Important if you are part of several projects
- **#MSUB -x**
  - Request for exclusive usage of allocated nodes
- **#MSUB -X**
  - Enable X11 forwarding
- Complete up to date documentation: ccc_msub -h

Job submission

Example of a submission script for a serial job

```
#!/bin/bash
#MSUB -r MyJob                  # Request name
#MSUB -n 1                      # Number of tasks to use
#MSUB -T 600                    # Elapsed time limit in seconds
#MSUB -o example_%I.o           # Standard output. %I is the job id
#MSUB -e example_%I.e           # Error output. %I is the job id
#MSUB -A raxxxx                 # Project ID
#MSUB -q broadwell              # Choosing standard nodes
#MSUB -@ noreply@cea.fr:end     # Mail notification at the end of execution

set -x
cd ${BRIDGE_MSUB_PWD}    # contains the directory where the script was submitted

ccc_mprun ./a.out
```

Job submission

Example of a submission script for an MPI job

```
#!/bin/bash
#MSUB -r MyJob                  # Request name
#MSUB -n 32                     # Number of tasks to use
#MSUB -T 600                    # Elapsed time limit in seconds
#MSUB -o example_%I.o           # Standard output. %I is the job id
#MSUB -e example_%I.e           # Error output. %I is the job id
#MSUB -A raxxxx                 # Project ID
#MSUB -q broadwell              # Choosing standard nodes

set -x
cd ${BRIDGE_MSUB_PWD}    # contains the directory where the script was submitted

ccc_mprun ./a.out
```

Job submission

- Example of a submission script for an OpenMP job

```
#!/bin/bash
#MSUB -r MyJob                  # Request name
#MSUB -n 1                      # Number of tasks to use
#MSUB -c 7                      # Number of cores per task to use
#MSUB -T 600                    # Elapsed time limit in seconds
#MSUB -o example_%I.o           # Standard output. %I is the job id
#MSUB -e example_%I.e           # Error output. %I is the job id
#MSUB -A raxxxx                 # Project ID
#MSUB -q broadwell              # Choosing standard nodes

set -x
cd ${BRIDGE_MSUB_PWD}    # contains the directory where the script was submitted
export OMP_NUM_THREADS=7

ccc_mprun ./a.out
```

Job submission

- Example of a submission script for a hybrid MPI/OpenMP job

```
#!/bin/bash
#MSUB -r MyJob_ParaHyb              # Request name
#MSUB -n 4                          # Number of tasks to use
#MSUB -c 7                          # Number of cores per task to use
#MSUB -T 600                        # Elapsed time limit in seconds
#MSUB -o example_%I.o               # Standard output. %I is the job id
#MSUB -e example_%I.e               # Error output. %I is the job id
#MSUB -q standard                   # Choosing standard nodes
#MSUB -A paxxxx                     # Project ID

set –x
cd ${BRIDGE_MSUB_PWD}
export OMP_NUM_THREADS=7

ccc_mprun ./a.out
```

Monitoring

- The job ID

```
$ ccc_msub myscript.sh
  Submitted Batch Session 605375
```

- Display launched jobs

```
$ ccc_mpp -u $USER
```

USER	ACCOUNT	BATCHID	NCPU	QUEUE	PRIORITY	STATE	RLIM	RUN/START	SUSP	OLD	NAME	NODES/REASON
ccrttp01	ccrt0035	605425	1024	normal	299996	PEN	10.0m	-	-	5.0s	simple_test	Resources
ccrttp01	ccrt0035	605378	64	normal	299996	PEN	10.0m	-	-	1.6m	simple_test	Reservation
ccrttp01	ccrt0035	**605375**	64	normal	299996	RUN	10.0m	8.1m	-	8.0m	simple_test	cobalt[1363,1365-1366,1368]

- Delete one or several jobs with ccc_mdel

```
$ ccc_mdel 605425 605378
```

Monitoring*

Display job summary with ccc_macct

```
$ ccc_macct 3601806
Jobid    : 3601806
Jobname  : test
User     : user
Account  : groupt@skylake
Limits   : time = 00:02:00 , memory/task = Unknown
Date     : submit = 15/03/2019 11:13:34 , start = 15/03/2019 11:13:34 , end = 15/03/2019 11:13:35
Execution : partition = skylake , QoS = test , Comment = none
Resources : ncpus = 2 , nnodes = 2
  Nodes=cobalt[3136,3143]

Memory / step
-------------
                    Resident Size (Mo)                    Virtual Size (Go)
JobID          Max     (Node:Task)      AveTask    Max  (Node:Task)          AveTask
-----------    ----------------------  -------   ------------------------  -------

Accounting / step
-----------------

JobID         JobName             Ntasks  Ncpus Nnodes  Layout      Elapsed          User       System     Eff
State
------------  ------------        ------  ----- ------  -------     -------          -------    ------     ---  -----
3601806       test                    -     2     2       -     00:00:01   (100%)        -          -      -  -
```

DEBUGGING: OVERVIEW

Summary

1. User environment
2. **Debugging: Overview**
   1. **Common bugs**
   2. **Debugging techniques**
3. Compiler options
4. Debugging with GDB
5. Graphical debugging with DDT
6. Memory debugging with Valgrind

Introduction

- Software and architectures are becoming generally more complex
- “Bugs” = mistake, failure, exception, crash, error,…
- Bugs can be introduced at any point during development
- Bugs become more and more difficult to find and fix

Common bugs

- Arithmetic bugs
  - Division by zero
  - Arithmetic overflow or underflow
  - Loss of arithmetic precision due to rounding or numerically unstable algorithms. This may be triggered by compiler optimization
- Logic bugs
  - Infinite loops and infinite recursion
  - Off by one error, counting one too many or too few when looping
- Syntax bugs
  - Use of the wrong operator. For example, x=5 instead of x==5
  - Often warned by the compiler; in many languages, deliberately guarded against by language syntax

Common bugs

- Resource bugs
  - Null pointer dereferencing
  - Using an uninitialized variable
  - Access violations
  - Resource leaks, where a finite system resource (such as memory or file handles) become exhausted by repeated allocation without release
  - Buffer overflow, in which a program tries to store data past the end of allocated storage
  - Double free error

Common bugs

- MPI bugs
  - Logical errors regarding the distribution of data across processes
  - Communication deadlocks: sends without receives, collective communication shared by all but one processes, etc
  - Errors due to the order of operations in Allreduce calls
  - Lack of synchronization: read a buffer before the communication is completed
- Multi-threading programming bugs
  - Deadlock, where task A can't continue until task B finishes, but at the same time, task B can't continue until task A finishes
  - Race condition, where the computer does not perform tasks in the order the programmer intended
  - Concurrency errors in critical sections, mutual exclusions and other features of concurrent processing

Debugging

- Main difficulty is finding the origin of the faulty behavior in the source code

- Steps for debugging
  - Reproduce the error
  - Simplify the test case and the code as much as possible
  - Close in on the suspected area of the bug
  - Monitor the code step by step
  - Check critical values and behavior

- Most kinds of debugging will have a relevant effect on execution time and memory usage

Debugging techniques

- Print debugging
  - Check the flow of execution of a process thanks to print statements
- Use debugger
  - Follow the execution of a process with a debugger
  - Set breakpoints, check variables, etc
- Forensic debugging
  - Analyze the call stack at the moment of the crash if available
  - If a core dump is generated, analyze it

Print debugging

- Using printf / print
  - In order to print critical values
  - In order to evaluate where the program crashes

- Best practice: use conditions to enable/disable debug easily
  - Add conditions around all calls to print

```
#ifdef DEBUG
printf("Starting xx computation");
#endif
```

  - Define conditional macros
  - Enable debugging information by compiling with

```
$ icc -DDEBUG
$ ifort -fpp -DDEBUG
```

Print debugging: Using macros

Usage of the macro in a code:

```
#ifdef DEBUG
#define debug_print(message, ...) \
fprintf(stdout, "(debug %s l.%d) " message "\n", \
__FILE__, __LINE__, ##__VA_ARGS__)
#else
#define debug_print(message, ...)
#endif  /*DEBUG*/

int main(int argc, char *argv[]){
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);

    debug_print("input information:");
    debug_print("input x = %d",x);
    debug_print("input y = %d",y);
```

Expected output:

```
$ icc -DDEBUG -o crash crash.c
$ ./crash 2 9
  (debug crash.c 1.29) input information:
  (debug crash.c 1.30) input x = 2
  (debug crash.c 1.31) input y = 9
```

Print debugging: Using macros

Usage of the macro in a code:

```
#ifdef DEBUG
#define debug_print(message,value)            &
write(*,'("(debug ", A ," l.",I4,") ",A,A)')  &
__FILE__,__LINE__,message,value
#else
#define debug_print(message)
#endif

program main
  character   :: x,y
  call getarg(1,x)
  call getarg(2,y)

  debug_print("input x = ",x)
  debug_print("input y = ",y)
```

Expected output:

```
$ ifort -fpp -DDEBUG -o crash crash.f90
$ ./crash 2 9
   (debug crash.f90 1.16) input x = 2
   (debug crash.f90 1.17) input y = 9
```

COMPILER OPTIONS

Summary

1. User environment
2. Debugging: Overview
3. **Compiler options**
4. Debugging with GDB
5. Graphical debugging with DDT
6. Memory debugging with Valgrind

Compiler options

- Default options and syntax depend on the chosen compiler
- Level of optimization may change with the compiler versions
- The following presentation is valid for the Intel 17 compilers
- To get a complete list and description of compiler options, use the man pages:

```
$ man ifort
$ man icc
```

Basic optimization options

- -O0
  - Disables all optimization
    Math expressions will be evaluated in the same order in which they are written
- -O1
  - Optimizes for maximum speed, but disables some optimizations which increase code size for a small speed benefit. Also disables software pipelining and global code scheduling
- -O2
  - Enables optimization. This is the default option
- -O3
  - Enables -O2 plus more aggressive optimizations that may not improve performance for all programs

Debugging options

- -g
  - Produces symbolic debug information in object file
  - Implies -O0 when no other optimization option is set explicitly
- -debug
  - Enables generation of debugging information
- -traceback
  - Tells the compiler to generate extra information to provide source file traceback information when an error occurs at runtime

Debugging and optimization

- Default optimization with debug information is -O0
- A code may be much slower when compiled with -O0 instead of -O2
- It is possible to combine -g and -O2 or -O3
- Some errors may only appear when compiling with higher optimizations

Debug options

- -ftrapuv
  - Initializes stack local variables to an unusual value to help error detections
  - Makes it easier to identify errors caused by improper initialization

- -check-uninit (C/C++) / -check uninit (Fortran)
  - Enables runtime checking for uninitialized variables

Debug options (Fortran only)

- -check bounds
  - Enables checking for array subscript and character substring expressions
- -check pointers
  - Enables checking for certain disassociated or uninitialized pointers or unallocated allocatable objects
- -check all

Floating point precision

- 3 conflicting objectives of a good application:
  - Accuracy
    - get a result that is close to the exact calculation
  - Reproducibility
    - produce consistent results
      - From one run to the next
      - From one set of build options to another
      - From one compiler to another
      - From one processor or operating system to another
  - Performance

Floating point precision

- The order of operations matters:

```
2^60 - (2^60 + 1) = 0
(2^60 - 2^60) + 1 = 1
```

- Optimizations or compiler versions may change the order of operations

Floating point precision

- -fp-model
  - precise
    - Enables value-safe optimizations on floating-point data
  - except
    - Enables floating-point exception semantics
  - strict
    - Enables precise and except. This is the strictest floating-point model
  - fast [=1|2]
    - Enables more aggressive optimizations on floating-point data
    - -fp-model fast=1 is the default option

Floating point precision

- Intel MKL (>=Composer XE 2017) provides conditional numerically reproducible results
- Setting the MKL_CBWR environment variable ensures consistency of MKL calls
  - On every Intel or Intel compatible CPUs:
    - MKL_CBWR=COMPATIBLE
  - Only on Intel CPUs, depending on the type of instructions supported:
    - MKL_CBWR=SSE2
    - MKL_CBWR=SSE4_2
    - MKL_CBWR=AVX
    - MKL_CBWR=AVX2
    - MKL_CBWR=AVX512
- Setting this option may affect the performance of the application

Floating point precision

- Denormalized values: numbers smaller than the floating point precision

- -ftz (-no-ftz)
  - Flushes denormalized results to zero
  - Default option for all optimization levels, except O0
  - Could alter the numerical behavior of your program

Floating point exceptions

- Floating point exceptions:
  - Overflow
    - result exceeds largest finite number
  - Underflow
    - result is smaller than the minimum range of normalized values
  - Divide-by-zero
    - divisor is zero and dividend is a finite nonzero number
  - Invalid
    - operation produces NaN value
    - eg:

```
0/0 ; log(negative) ; 0*(-/+∞) ; +∞+-∞;
```

- The behavior of the program when these exceptions occur depends on the compiler options

Floating point exceptions

- Specify the floating-point exception handling level for the main routine
  - Fortran: -fpe[0|1|3]
  - C/C++: -fp-trap=[mode,…]
- Specify the floating-point exception handling level for all routines
  - Fortran: -fpe-all[0|1|3]
  - C/C++: -fp-trap-all=[mode,…]

Floating point exceptions (Fortran)

- -fpe0:
  - The overflow, the divide-by-zero, and the invalid floating-point exceptions will make the program print an error message and abort
  - If a floating-point underflow occurs, the result is set to zero and execution continues
- -fpe1:
  - The overflow, the divide-by-zero, and the invalid floating-point exceptions will produce exceptional values (NaN and signed Infinities) and execution continues
  - If a floating-point underflow occurs, the result is set to zero and execution continues
- -fpe3 (default):
  - The overflow, the divide-by-zero, and the invalid floating-point exceptions will produce exceptional values (NaN and signed Infinities) and execution continues
  - Floating underflow is gradual: denormalized values are produced until the result becomes 0

Floating point exceptions (C/C++)

-fp-trap=divzero	Enables the trap for division by zero
-fp-trap=inexact	Enables the trap for inexact result
-fp-trap= invalid	Enables the trap for invalid operation
-fp-trap=overflow	Enables the trap for overflow
-fp-trap=underflow	Enables the trap for underflow
-fp-trap=denormal	Enables the trap for denormalized values
-fp-trap= common	Sets the most commonly used IEEE traps: division by zero, invalid operation, and overflow
all	
none (default)	

DEBUGGING WITH GDB

Summary

1. User environment
2. Debugging: Overview
3. Compiler options
4. **Debugging with GDB**
   1. **Start a program with GDB**
   2. **Useful commands**
   3. **Alternative uses of GDB**
   4. **DDD**
1. Graphical debugging with ddt
2. Memory debugging with valgrind

Description

- GDB is the GNU DeBugger
- Available on most UNIX systems
- Useful to easily:
  - Start and stop a program
  - Check and modify variables at runtime
  - Run the code step by step
- Most effective if the code is compiled with -g

Debugging with GDB

STARTING A PROGRAM WITH GDB

Starting a program with GDB

- Simple “crash” code:

```
1     #include <stdio.h>
2
3     void display(int value){
4         printf("%d\n",value);
5     }
6
7     void divint(int a, int b){
8         int c = a / b;
9         display(c);
10    }
11
12    int main(int argc, char *argv[]){
13        int M[10];
14        int i=0;
15        for(i=0; i<10; i++) M[i]=9-i;
16        int x = atoi(argv[1]);
17        int y = atoi(argv[2]);
18        divint(M[x],M[y]);
19        return 0;
20    }
```

Starting a program with GDB

- Generates crash if division by zero:

```
$ ./crash 1 9
    Floating point exception (core dumped)
```

- Recompile the code with debug information

```
$  icc -g crash.c -o crash
```

Starting a program with gdb

- Start gdb: gdb <program>

  $ gdb crash

  GNU gdb (GDB) bullx Linux (7.2-50.bl6.Bull.1.20120306)
  Copyright (C) 2010 Free Software Foundation, Inc.
  …
  Reading symbols from /…/Formation_gdb/crash...done.
  (gdb)

- Once the gdb session has started, launch the program and pass arguments: run <args>

  (gdb) run 1 9

  Starting program: /…/Formation_gdb/crash 1 9

  Program received signal SIGFPE, Arithmetic exception.
  0x004005a7 in divint (a=8, b=0) at crash.c:8
  8 int c = a / b;

Debugging with GDB

USEFUL COMMANDS

Reading the stack

- When the program stops, the first thing to do is analyze the call stack. That is the list of all the function calls, arguments and variables met in the program

- Each level of calls is called a “stack frame”

- To get a brief list of all stack frames: backtrace

(gdb) backtrace

#0 0x004005a7 in divint (a=8, b=0) at crash.c:8

#1 0x00400678 in main (argc=3, argv=0x7fffffffbd08) at crash.c:18

Reading the stack

- There are different possibilities to navigate in the call stack
  - frame <n>
    - Moves from one stack frame to another, and prints the stack frame you select
    - Without an argument, frame prints the current stack frame
  - up <n> / down <n>
    - Move n frames up/down the stack
    - Default for n is 1

Setting breakpoints

- A breakpoint makes your program stop whenever a certain point in the program is reached
- Breakpoints are set in the GDB session, before running the code
- Setting a breakpoint for a specific line in a specific file
  - (gdb) break crash.c:18
    - Breakpoint 1 at 0x400649: file crash.c, line 18.
- Setting a breakpoint for a specific function
  - (gdb) break crash.c:display
    - Breakpoint 2 at 0x40056f: file crash.c, line 4.

Managing breakpoints

- Commands to manage breakpoints
  - info breakpoints
    - Display currently set breakpoints
  - delete <Num>
    - Delete specified breakpoints
    - Num is the value given when the breakpoint is set
  - disable <Num> / enable <Num>
    - Disable/Enable specified breakpoints
  - Without arguments, default behavior is to delete/disable all breakpoints

Running step by step

- After the program stops at the breakpoint
  - continue
    - resume program execution normally
  - step <count>
    - continue running until next source line count times
    - default is 1
  - next <count>
    - continue running until the next source line in the same stack frame
    - function calls are executed without stopping

Check variables with GDB

- Display the value of a variable
  - print <variable>
- Change the value of a variable
  - set <variable = newvalue>

```
(gdb) run 0 5
      Starting program: /…/Formation_gdb/crash 0 5
      Breakpoint 1, main (argc=3, argv=0x7fffffffbd08) at crash.c:18
      18              divint(M[x],M[y]);
(gdb) print M
      $1 = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
(gdb) print x
      $2 = 0
(gdb) print M[x]
      $3 = 9
(gdb) set x=1
(gdb) print M[x]
      $4 = 8
```

Check source file

- Display part of the source file
  - list <linenum>
    - displays lines centered around “linenum” in the current file
  - list <function>
    - displays lines centered around the specified function

```
(gdb) bt
#0  0x00000000004005af in divint (a=9, b=0) at crash.c:19
#1  0x0000000000400680 in main (argc=3, argv=0x7fff64b676a8) at
crash.c:30

(gdb) list crash.c:30
25        int i=0;
26        for(i=0; i<10; i++) M[i]=9-i;
27        int x = atoi(argv[1]);
28        int y = atoi(argv[2]);
29
30        divint(M[x],M[y]);
31        return 0;
32    }
```

Debugging with GDB

ALTERNATIVE USES OF GDB

Attach to a running process

- While a process is running, it is possible to run GDB to check what it is doing
  - From the command line
    - gdb -p <process_id>
    - stopping gdb will detach the process
  - From a GDB session
    - (gdb) attach <process_id>
    - (gdb) detach
- Attaching to a process will pause the program
- Detaching from a process will make it resume

Generate and read a core file

- When some program crashes it automatically generates a core file

- To allow the production of core files, change the default limits of the system with

```
$ ulimit -c unlimited

$ ./crash 0 9
Floating point exception (core dumped)
```

- Core files are named crash-<pid>.core

Generate and read a core file

- To generate a core file of a running program, use gcore

- You just need the pid of the process

```
$ gcore <pid>
```

Generate and read a core file

- A core file contains the backtrace and callstack of the corresponding process

- The core file can be analyzed with GDB just like a running process

```
$ gdb ./crash *.core
```

Debugging with GDB

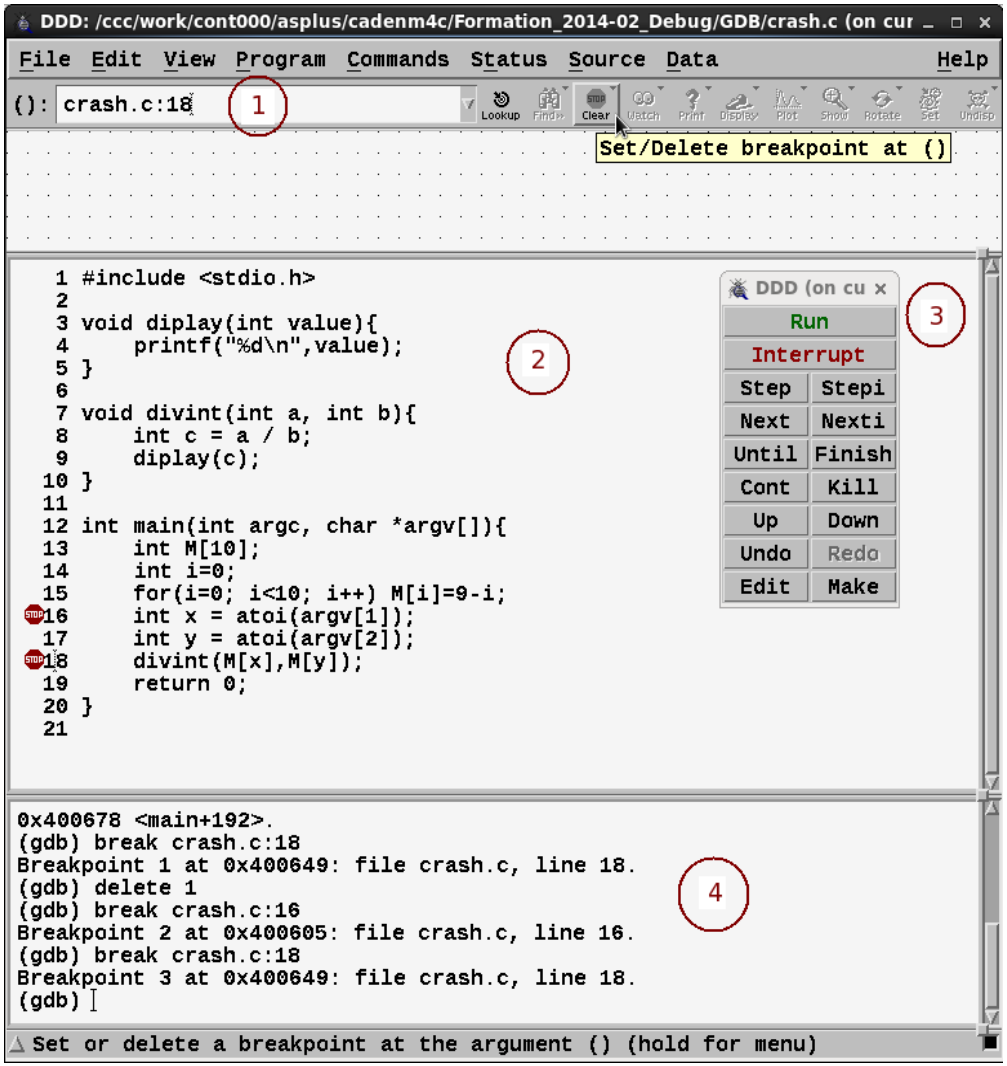
DDD

DDD

- DDD (Data Display Debugger) is a graphical front-end for GDB
- Allows exactly the same operations as command line gdb
- Launch:

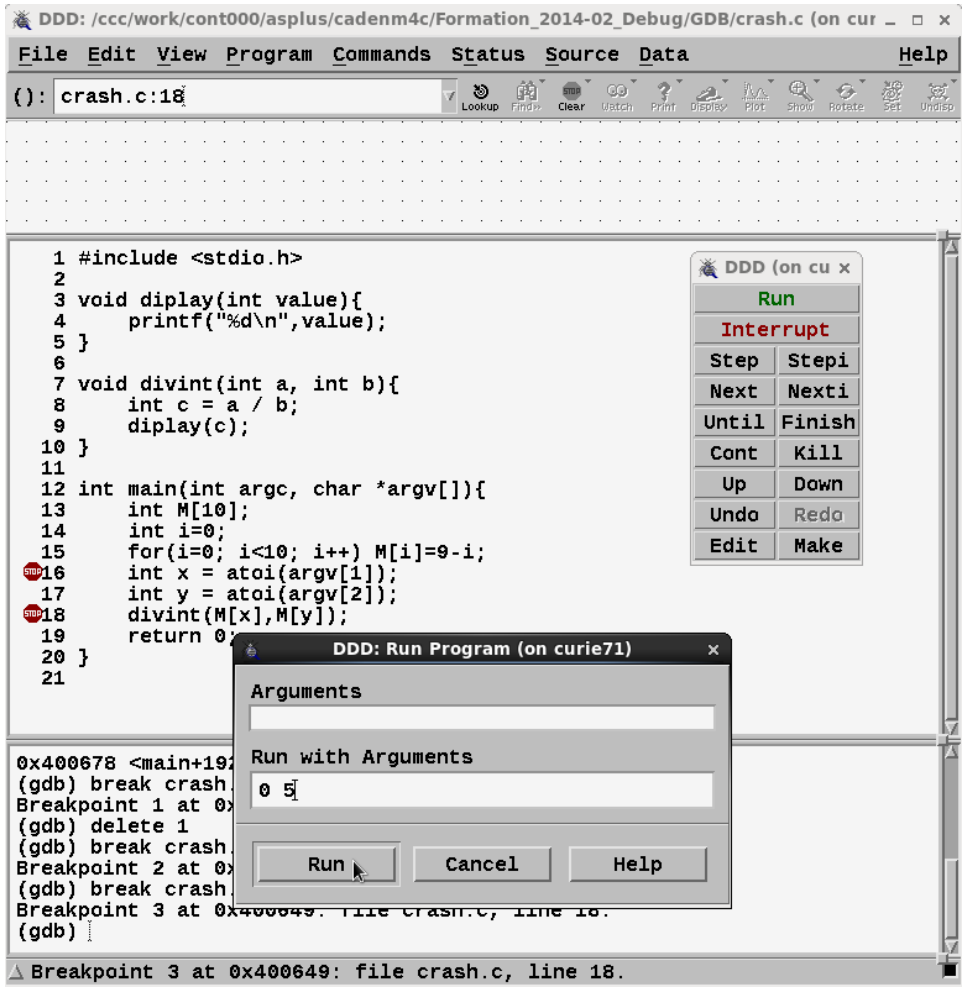
```
$ module load ddd
$ ddd <program>
```

DDD



- 1 - Argument field. Displays currently selected line, function or variable
- 2 - Source code
- 3 - Command tool. All the available commands to run the program
- 4 - Debugger console. Displays corresponding GDB commands

DDD



- To start the program and pass arguments
  - Program -> Run…

or

  - F2

ALTERNATIVES COMMAND LINE DEBUGGING

Find process ID

- top

```
$ top -u $USER
```

```
top - 14:26:42 up 37 days,  4:58, 74 users,  load average: 3.61, 3.34, 3.64
Tasks: 1381 total,   4 running, 1323 sleeping,  42 stopped,  12 zombie
Cpu(s):  7.1%us,  1.3%sy,  0.0%ni, 89.0%id,  2.6%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  132003416k total, 61344516k used, 70658900k free,   159680k buffers
Swap: 16384756k total,      640k used, 16384116k free, 28117928k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26312 cadenm4c  20   0 12780  448  344 R 99.6  0.0   3:49.16 crash
 6261 cadenm4c  20   0 20116 2360 1020 R  2.0  0.0   0:00.92 top
28120 cadenm4c  20   0  123m  18m 1484 S  0.0  0.0   0:00.16 sshd_user
28121 cadenm4c  20   0  110m 2212 1644 S  0.0  0.0   0:00.15 bash
```

- ps

```
$ ps
```

```
  PID TTY          TIME CMD
14198 pts/20   00:00:00 ps
26312 pts/20   00:05:20 crash
28121 pts/20   00:00:00 bash
```

Gstack

- gstack prints a stack trace of a running process
  - gstack <pid>

```
$ gstack 26312

#0  0x0000000000400576 in display ()
#1  0x00000000004005c4 in divint ()
#2  0x0000000000400686 in main ()
```

- It does not interrupt the process itself
- TIP: If the code was compiled with -g, use the command “addr2line” to find the matching file and line:

```
$ addr2line -e ./crash 0x00000000004005c4
…/Formation_gdb/crash.c:7
```

Strace

- strace prints all system calls of a running process
  - strace -p <pid>
    - attaches to an already running process
  - strace ./prog.exe
    - launches a program while printing system calls
- Useful options:
  - strace -c ./prog.exe
    - reports a summary on program exit
  - strace -e trace=<list,of,calls>
    - only prints specific system calls. Can be one of the calls listed in the summary or file, process, network, signal, ipc, desc
- See “man strace” for more details

Parallel jobs

- Tools available for parallel jobs submitted with ccc_msub :
  - clustack slurmjobid:<jobid>
    - gives an aggregated view of the stacks of the processes associated to the job
  - ccc_mstat -b <jobid>
    - shows all the processes related to the job <jobid> on the main compute node

GRAPHICAL DEBUGGING WITH DDT

Summary

1. User environment
2. Debugging: Overview
3. Compiler options
4. Debugging with GDB
5. **Graphical debugging with DDT**
   1. **Launching DDT**
   2. **Controlling program execution**
   3. **View variables and data**
   4. **Breakpoints, watchpoints, tracepoints**
   5. **Memory debugging**
6. Memory debugging with Valgrind

Overview

- Part of Arm-Forge
- Scalable, graphical debugger by Allinea (now part of ARM)
- Works for:
  - Single process and multi-threaded software
  - OpenMP
  - MPI
  - Hybrid codes such as MPI + OpenMP, or MPI + CUDA
- Works for most languages including C, C++, Fortran, HMPP, CUDA, etc.

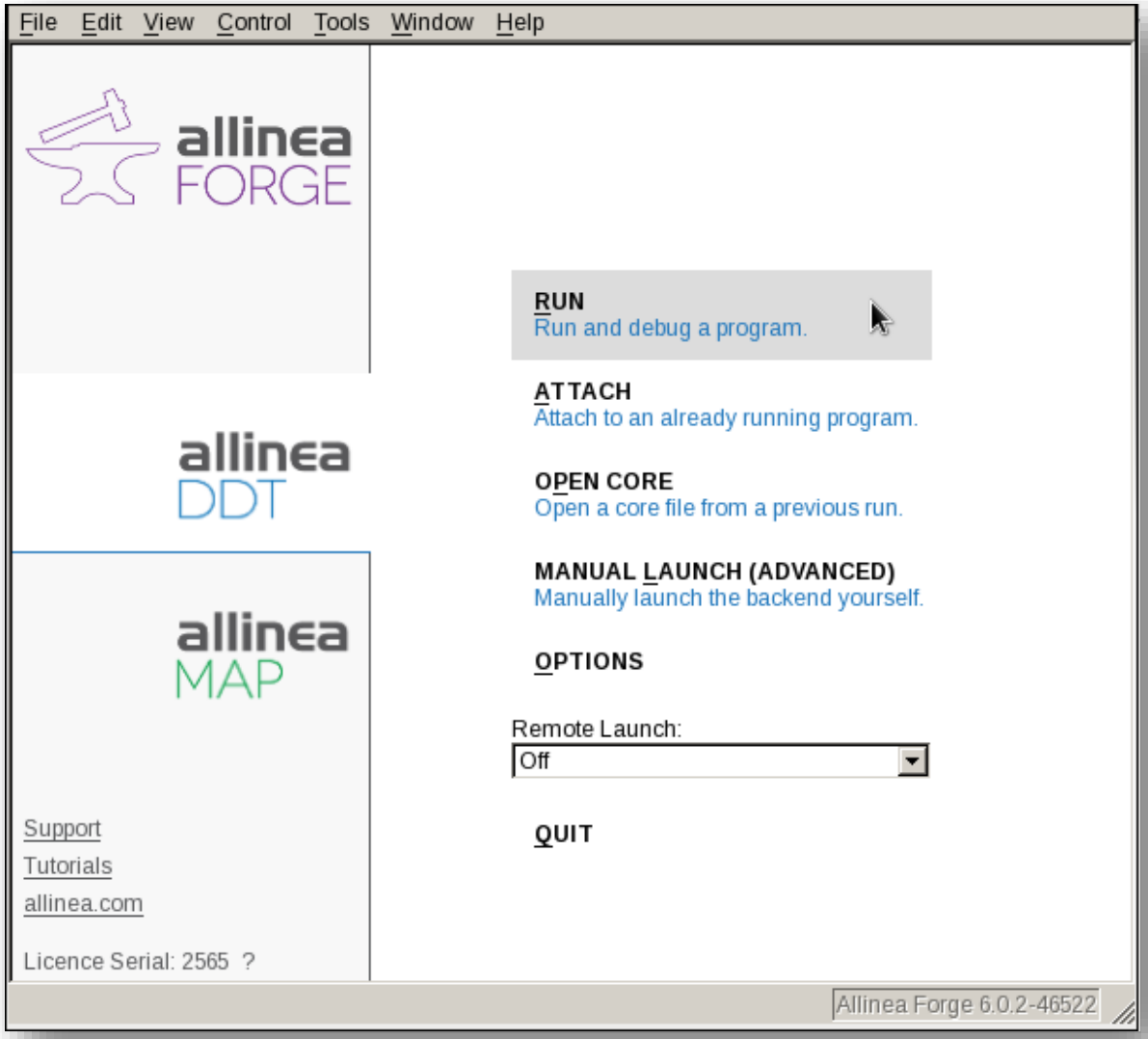
DDT

LAUNCHING DDT

Submission script

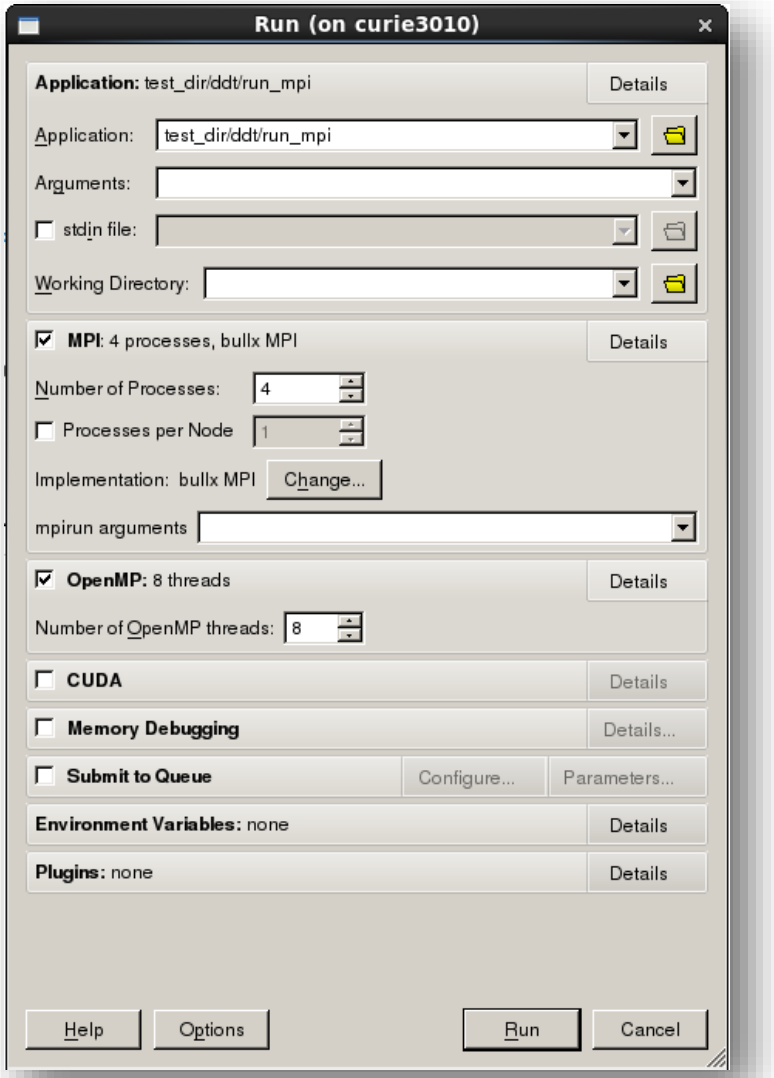
```
#!/bin/bash
#MSUB -r Test_DDT              # Task name
#MSUB -n 4                     # Number of tasks
#MSUB -X                       # Enable X11 forwarding
#MSUB -q broadwell             # Use broadwell partition

module load arm-forge

##
# ddt -n 4 ./test_ddt
##
# ddt -start -n 4 ./test_ddt  # skips the configuration
##
# ddt
##
```

Run a program

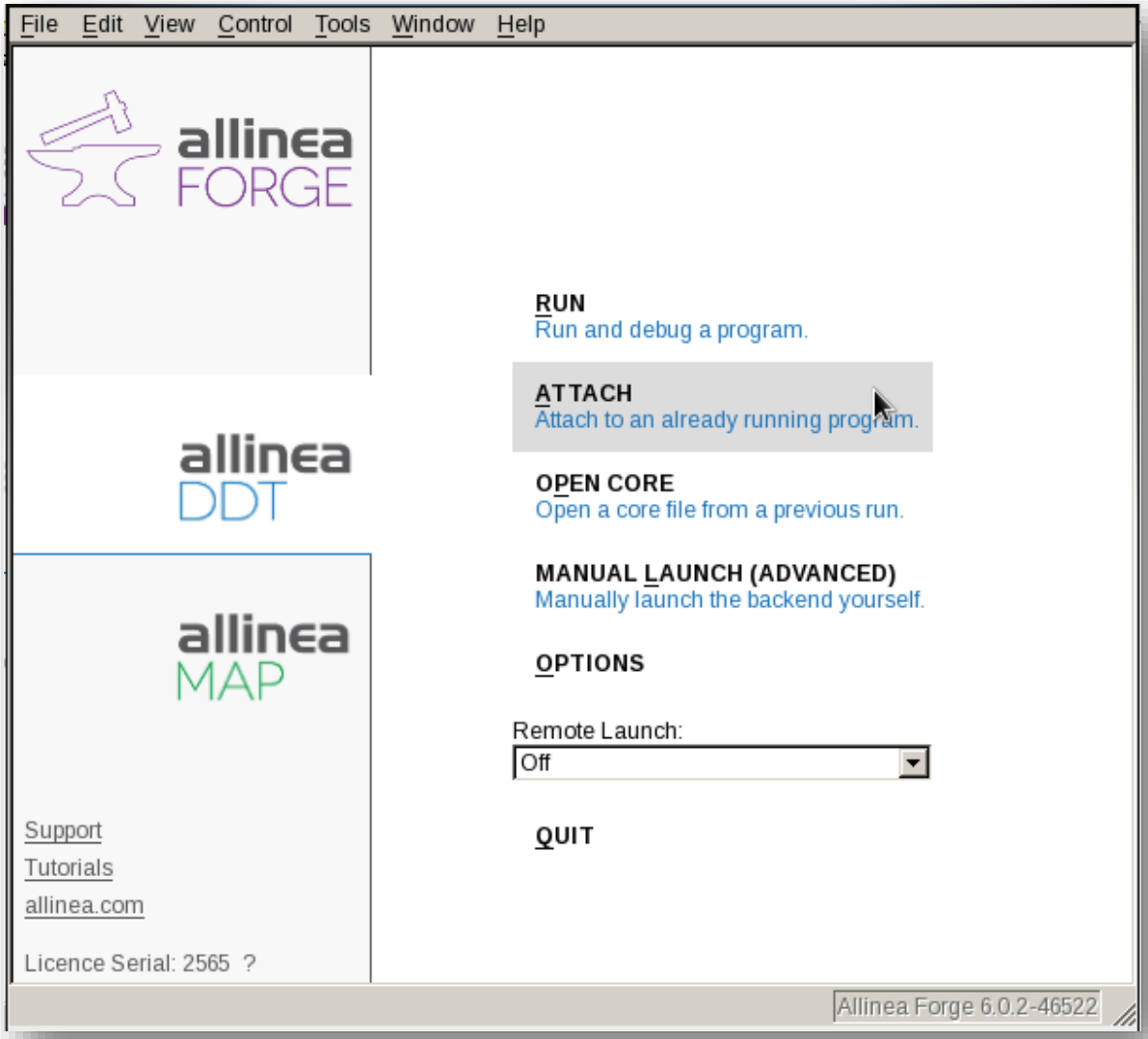


Run a program

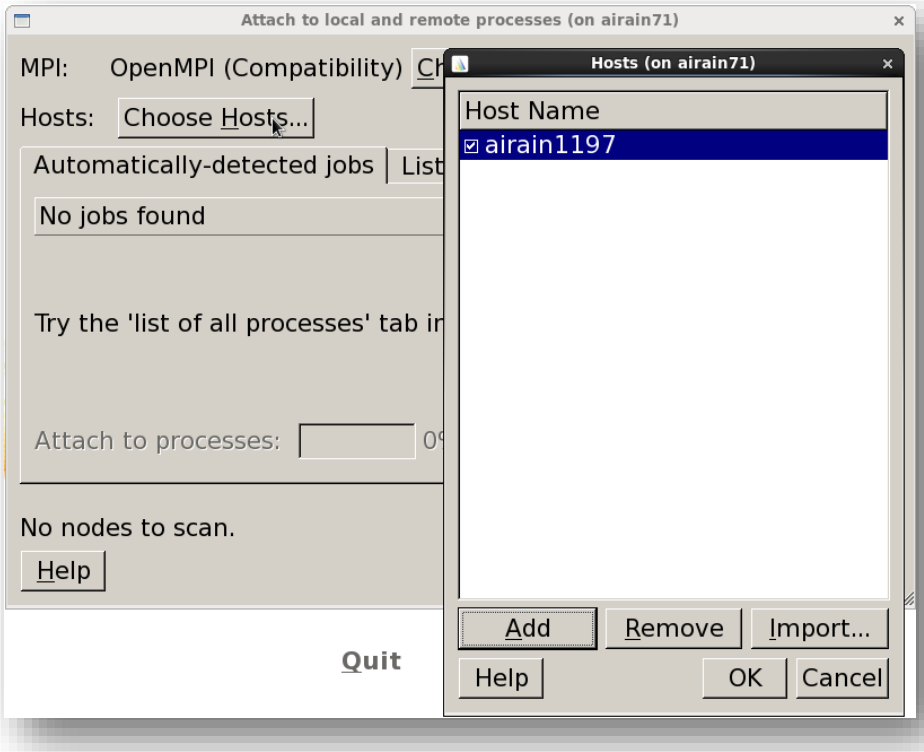
- Choose executable and arguments
- If it is an MPI code:
  - select the number of processes
  - select the MPI implementation (Slurm generic)
- Choose number of OpenMP threads if it is an OpenMP code
- Memory Debugging: check if you need it



Attach to a running program



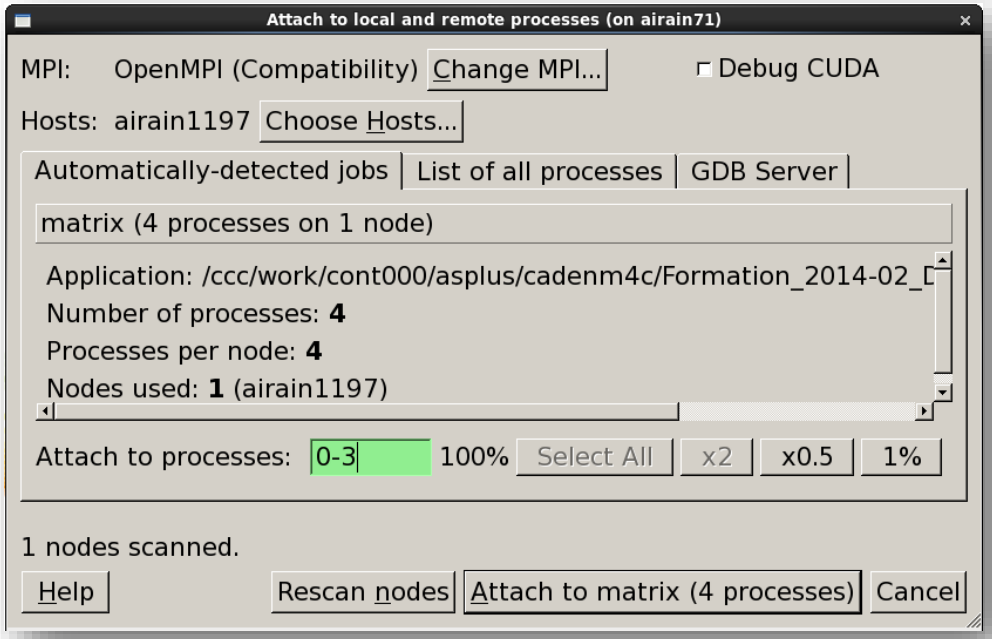
Attach to running process



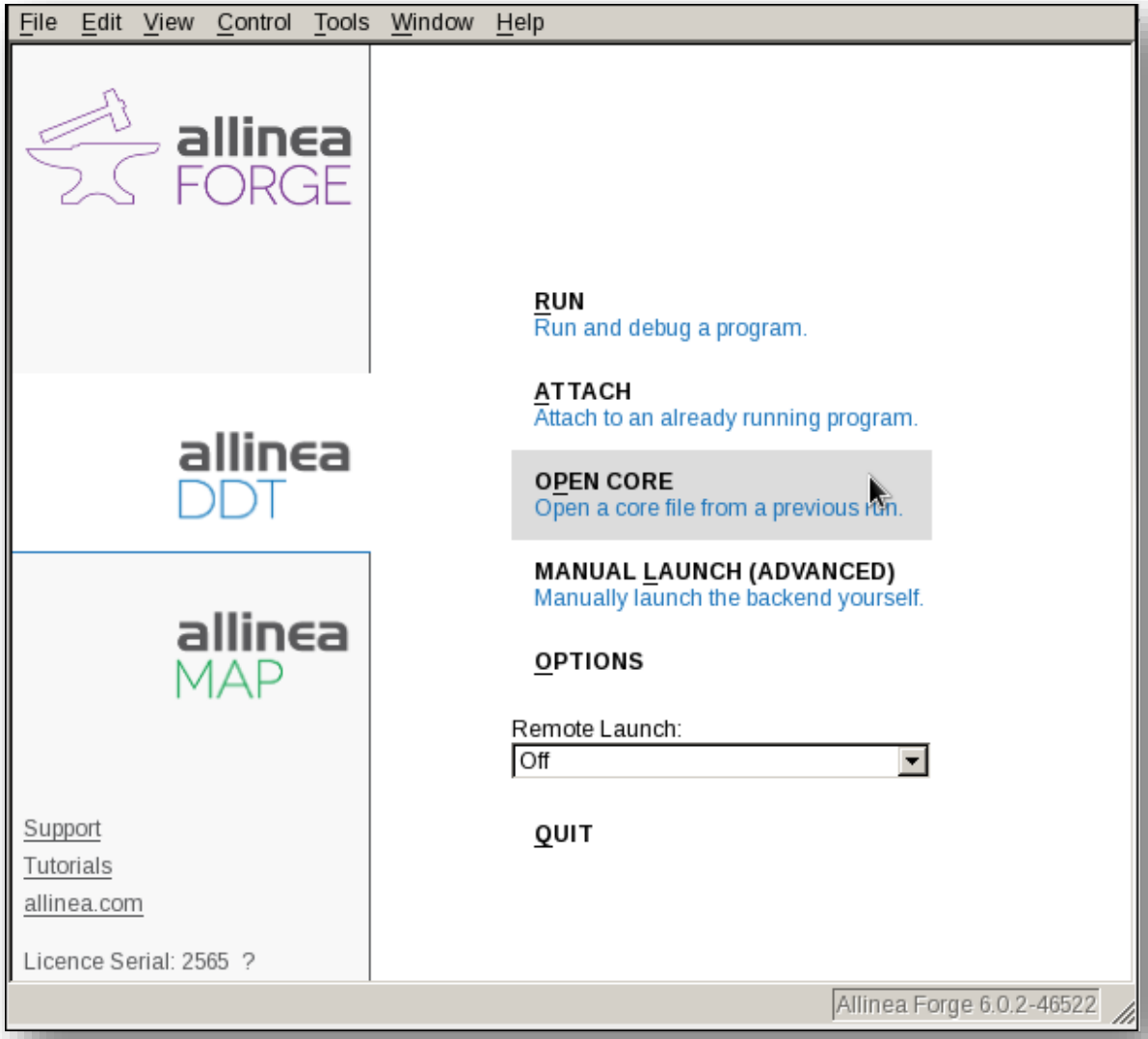
- If a job is running on Cobalt, it can easily be detected
- Click on *Choose Host*
- Add one of the hosts your job is running on (given in ccc_macct "jobid")

Attach to running process

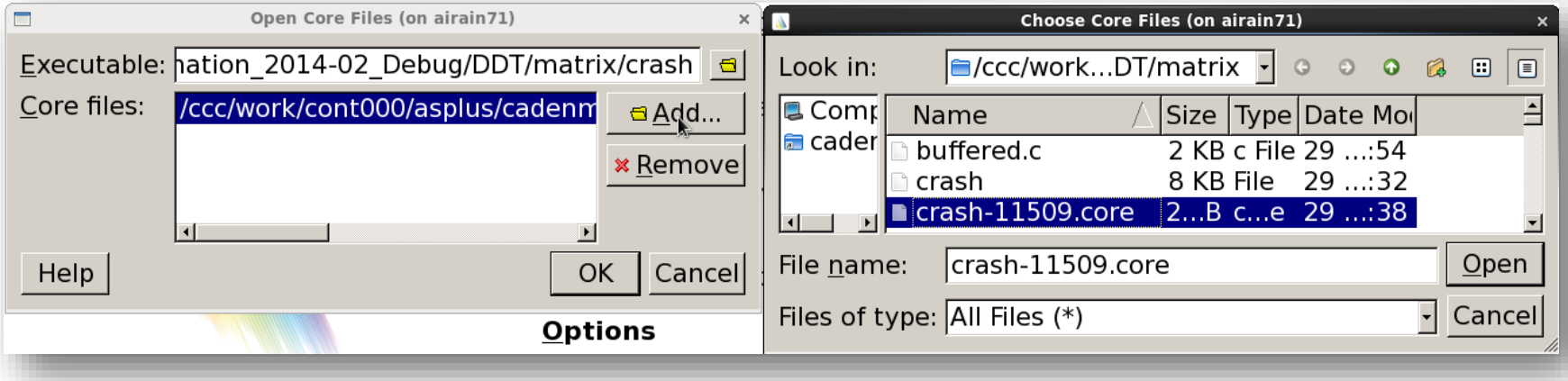
- Check if the detected process matches your job
- Choose number of processes to attach to
  - Warning: If your job is running on many processes, do not attach to all of them
- This will pause the program and open the usual DDT window



Open a core file



Open core files



- Select the program to debug
- Add the core files obtained with this program
- It is only “post mortem” debugging. You cannot play/pause/step

DDT

CONTROLLING PROGRAM EXECUTION

Overview

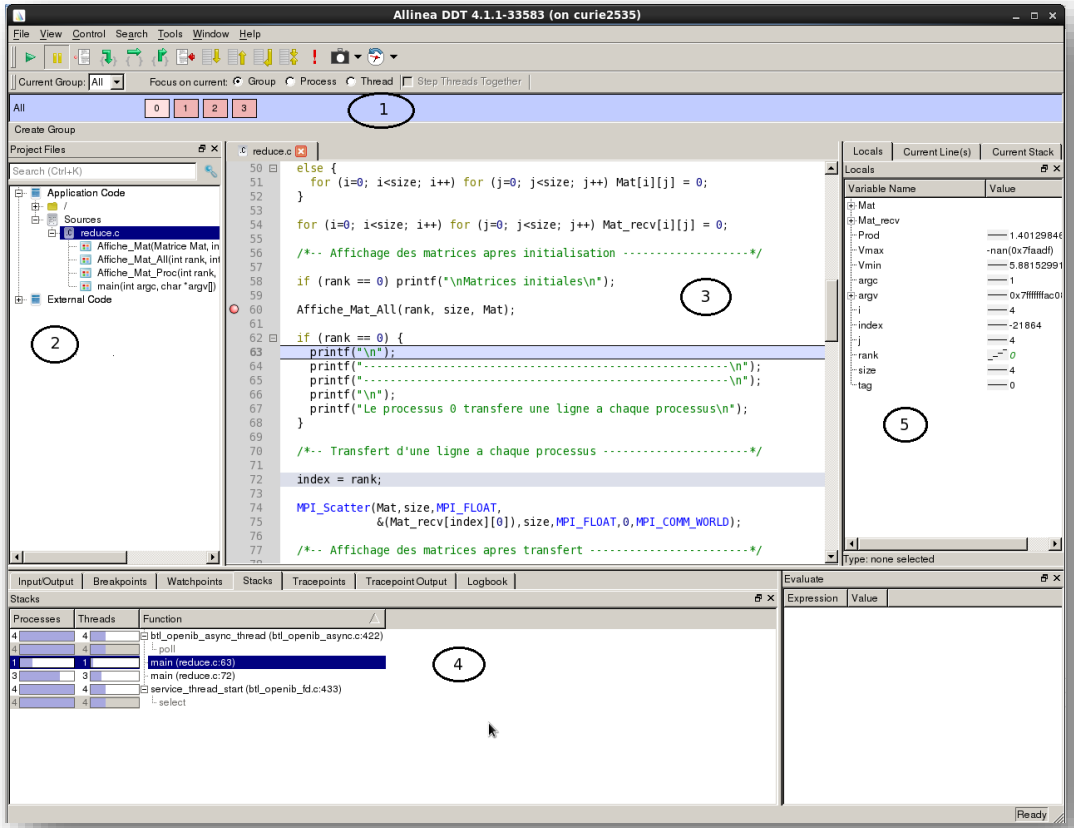
1) Process group and thread control

2) Source files and functions

3) Source code display

4) Parallel stack, I/O and breakpoints

5) Variables and stack of current process



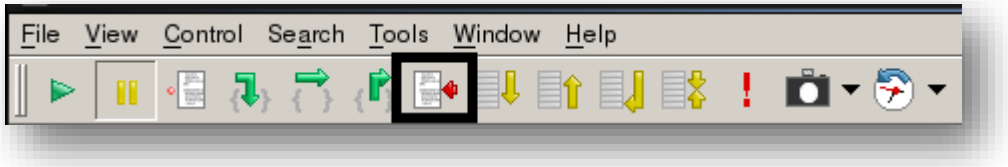
Controlling program execution

- Play/Continue
- Pause
  - all processes in the current group

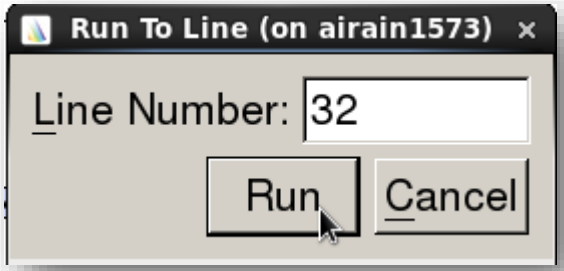
- Step into
  - Continue until next line or function
- Step over
  - Continue until next line in the same stack frame
- Step out
  - End function call and stop at first line of the frame above

Controlling program execution

- Run to line
  - Through control bar



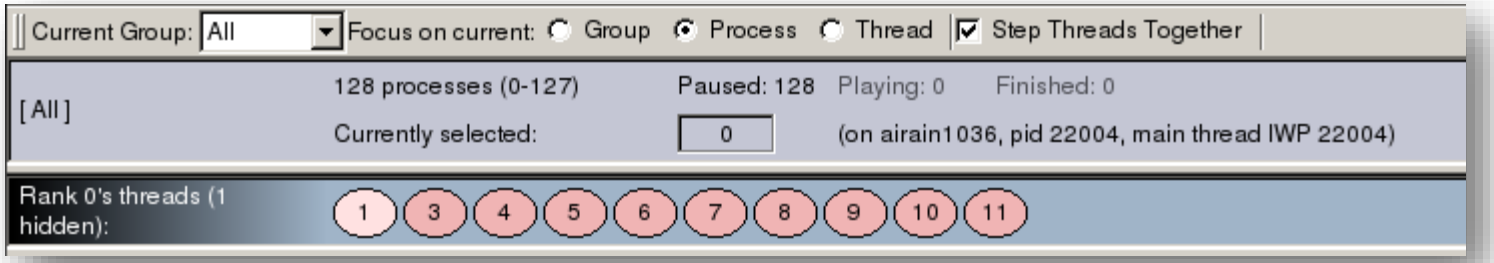
  - And then choose line in current file to run to



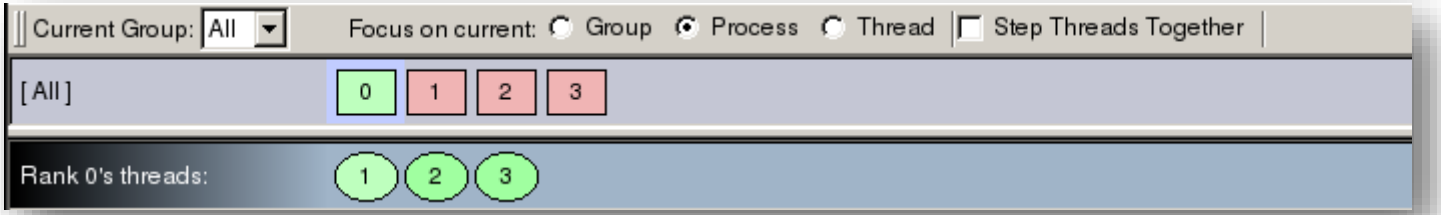
  - Or with *right-click > run to here* on source line directly

Select processes and tasks

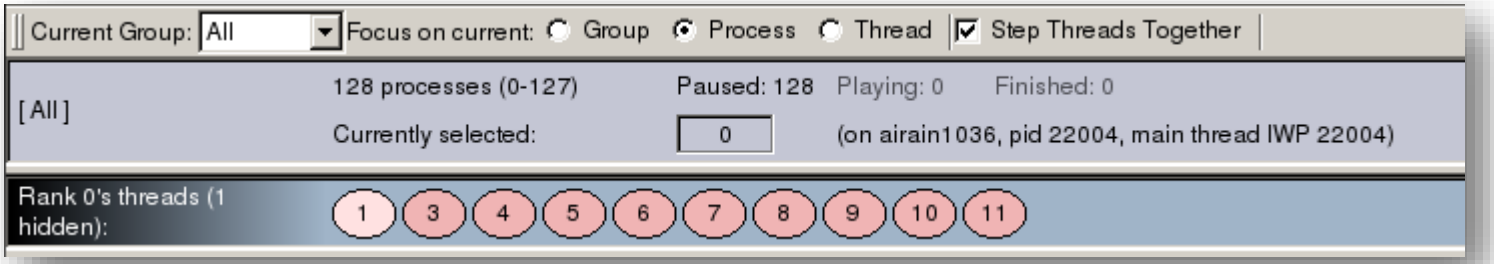
Summary view (default for nprocess > 32)



Detailed view (default for nprocess < 32)



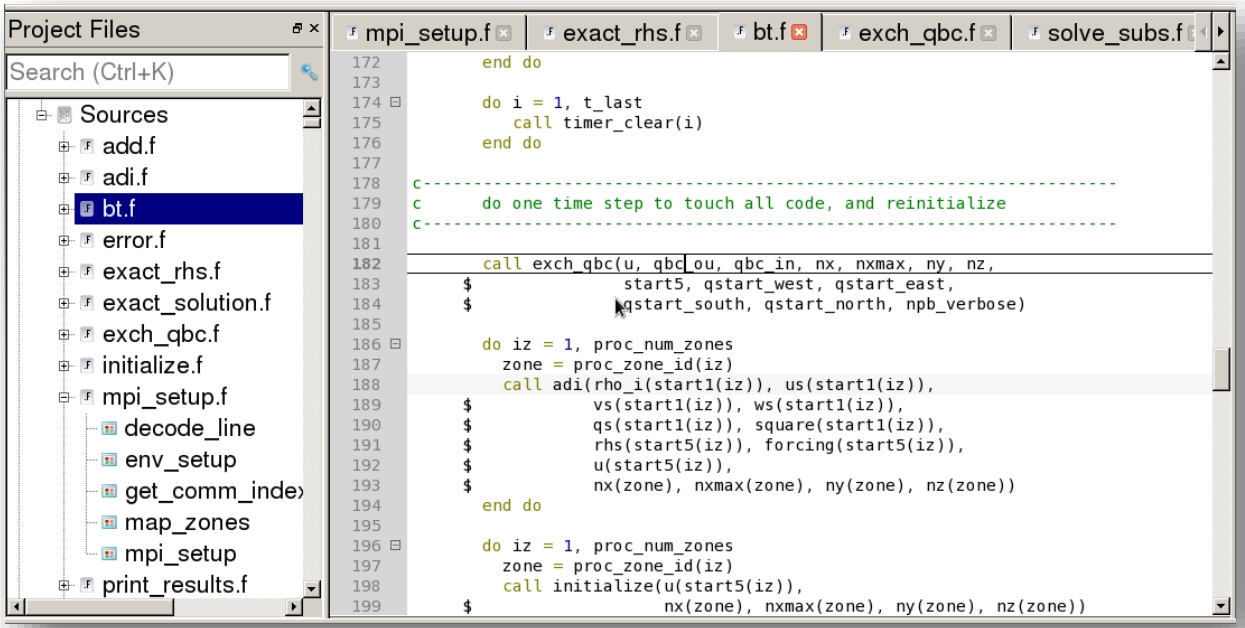
Select processes and tasks



- Shows
  - Number of processes in the group
  - State of each process

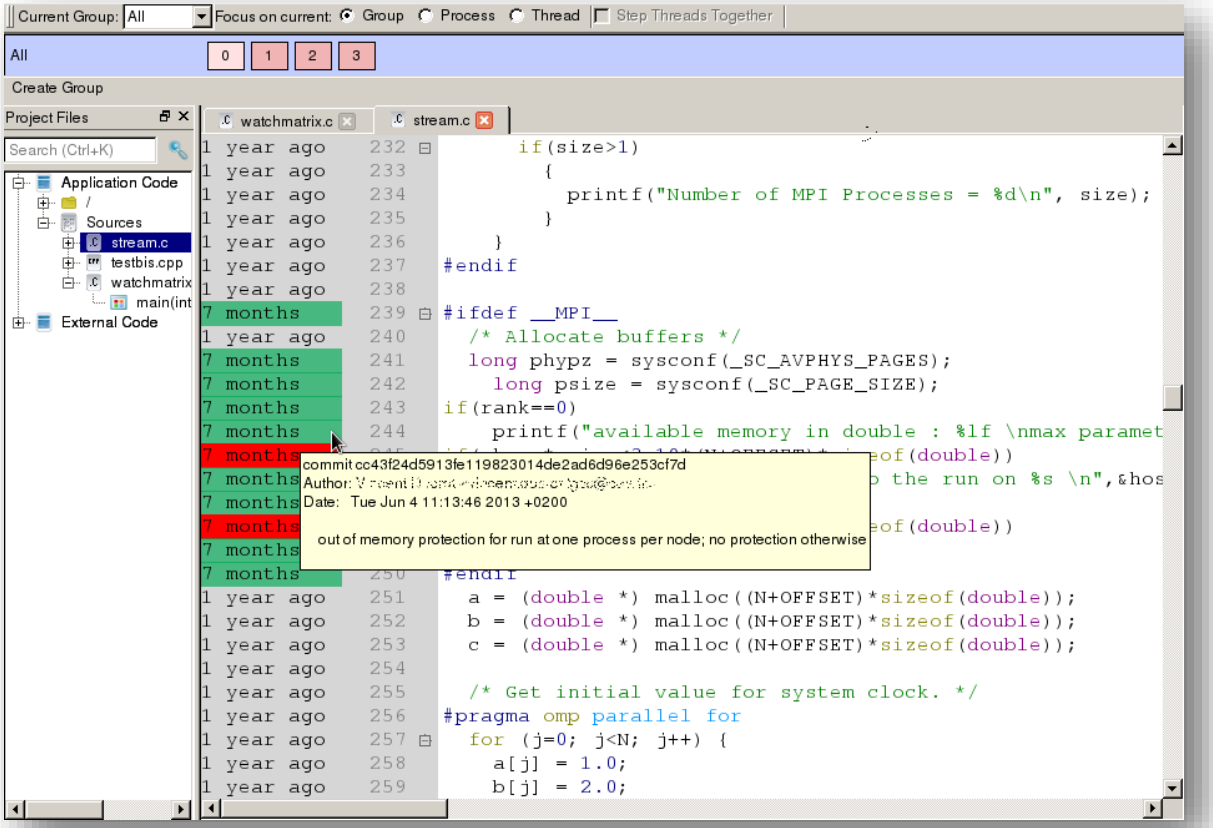
- Focus control
  - Allows to focus on group, process or thread
  - Will influence stepping, starting/pausing, breakpoints, etc
  - Influences display of source, stack trace, etc

Display source



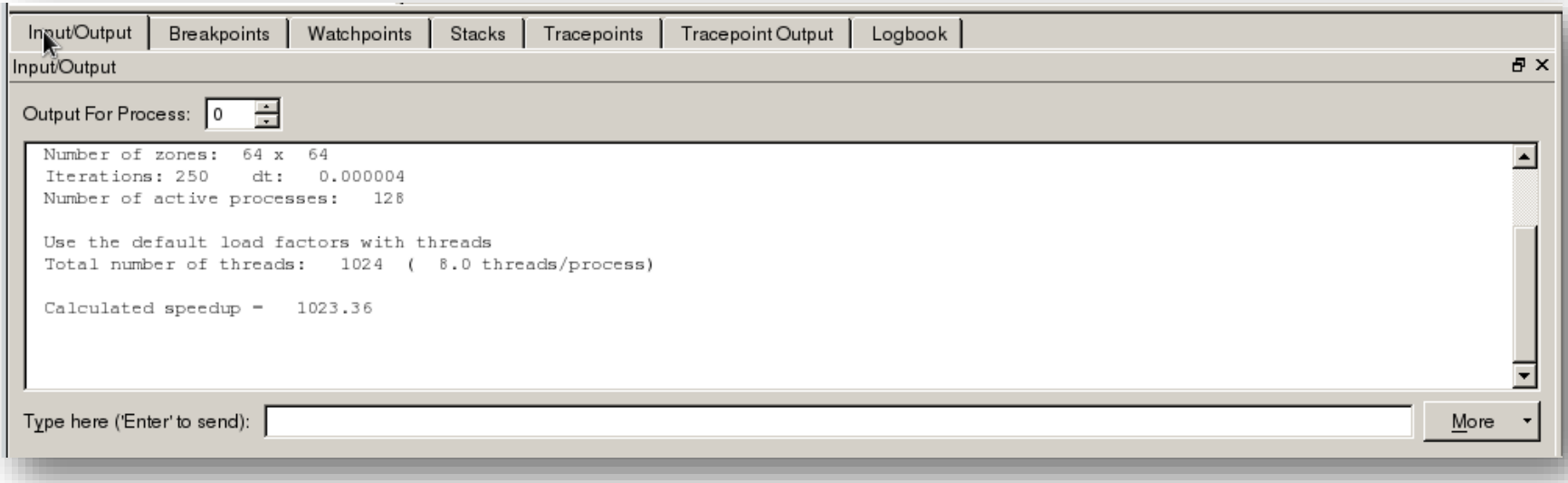
- When stopping a program, the corresponding source file is automatically opened
- The position of the current process or thread is highlighted in the source
- When starting DDT, starting line is automatically detected to “main” or “mpi_init”

Display source



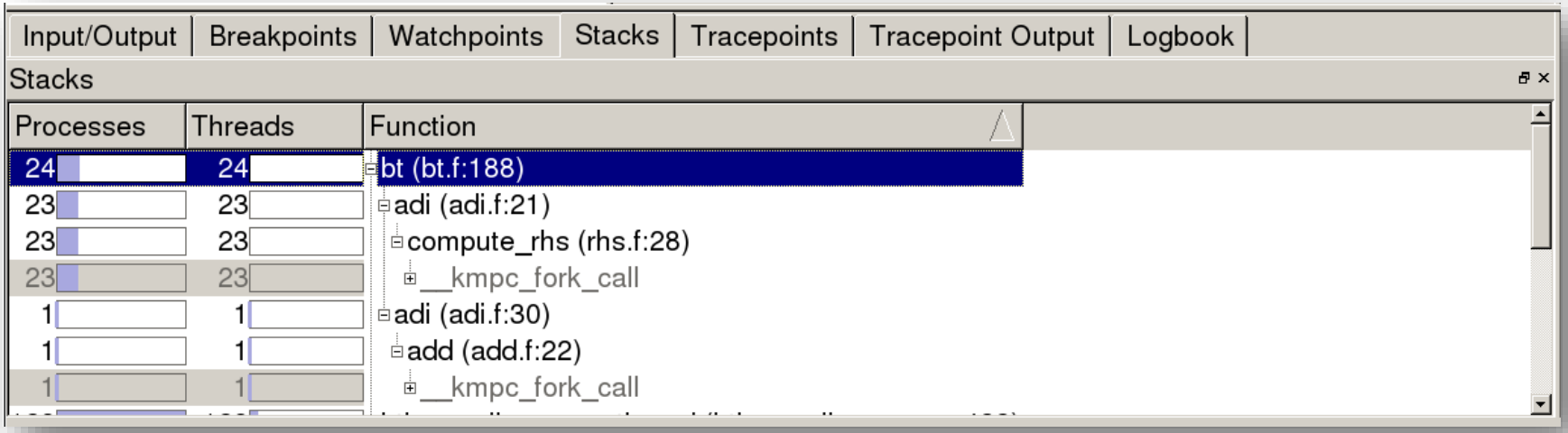
- Possibility to see line-by-line information from Git, Mercurial or Subversion next to source files
- In the menu, select *View > Version Control Information*

Input / Output



- Display standard output and error messages
- Input data during a session
- Possibility to filter the output on a specified process
- Save to a file with a right click

Call stack



- Parallel stack frames for currently selected group or processes
- Go up and down in the call stack
- The line in the source corresponding to the selected stack frame is displayed

Call stack

- The control bar allows an easy exploration of the stack when the program is paused

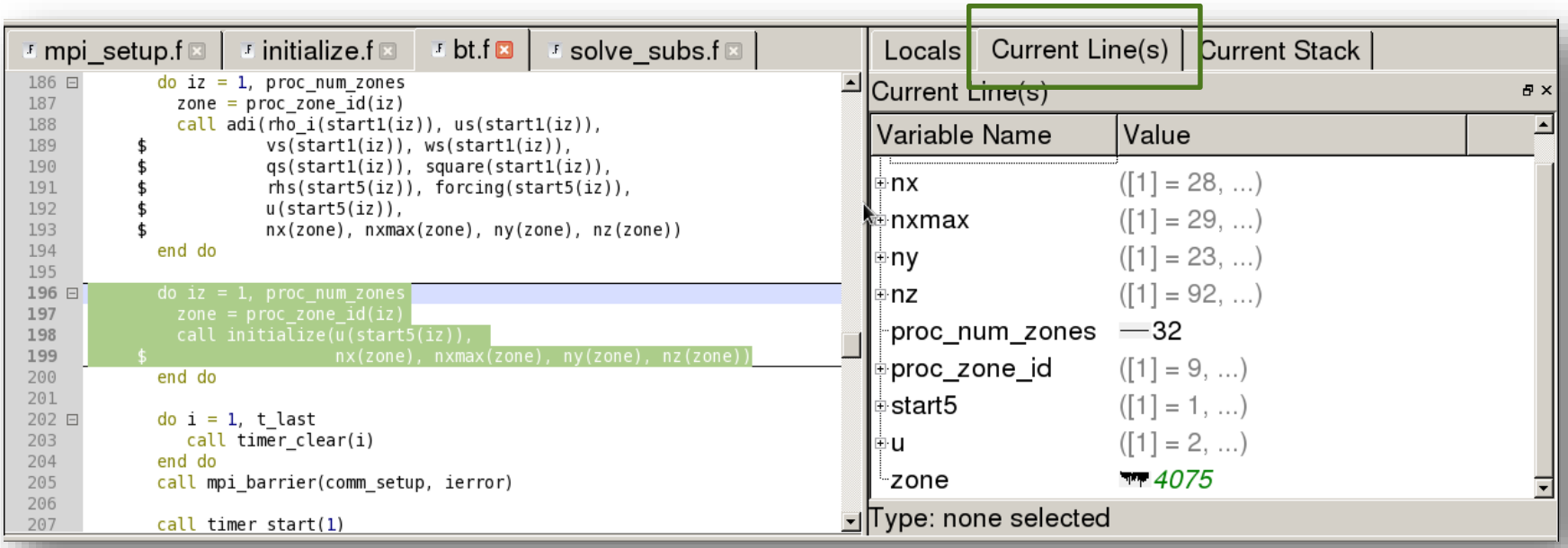
- Move to upper/lower frame
- Move to bottom frame
- Align stack frame with current

DDT

VIEWING VARIABLES AND DATA

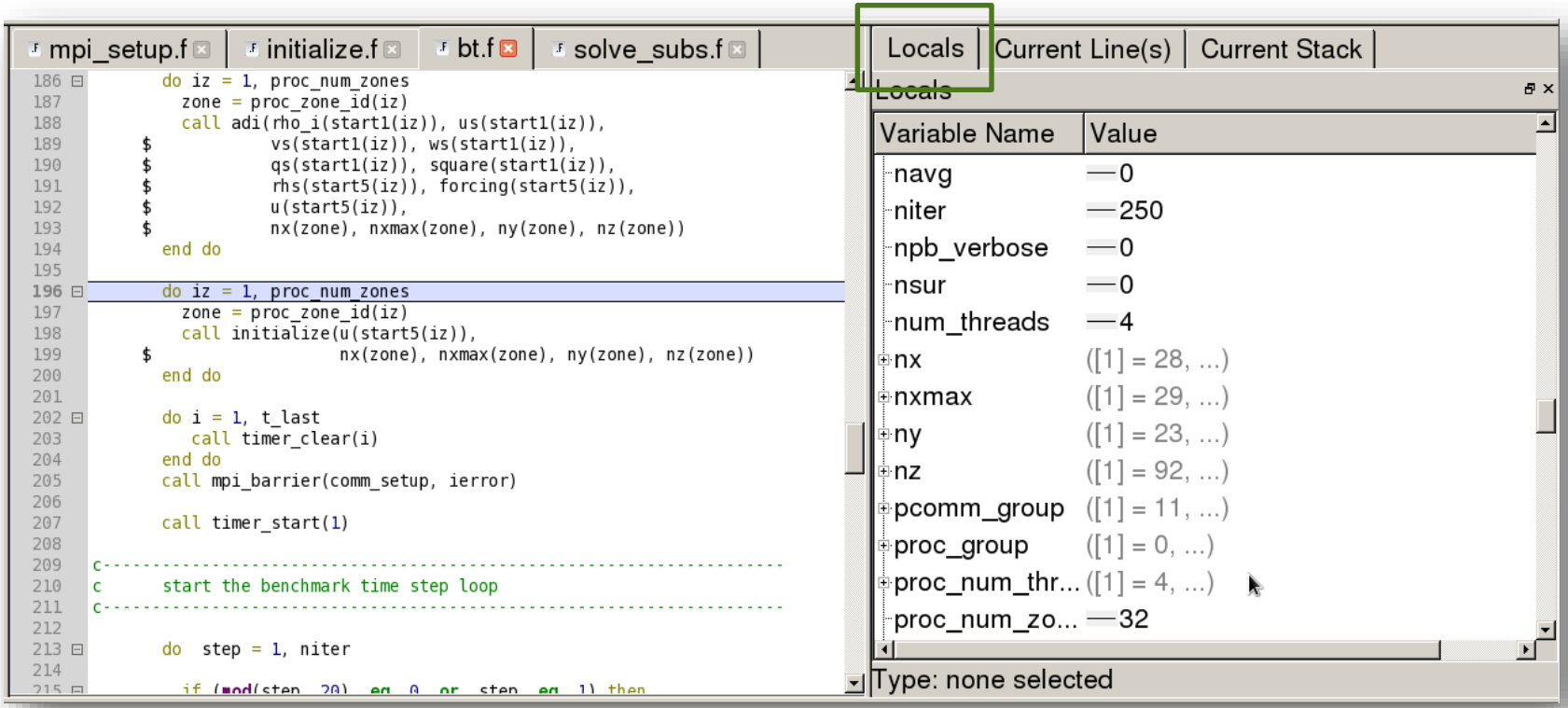
Current line

- Variables referenced on the currently selected lines
- Possibility to select several lines
- Spark lines give an idea of the value of the variable across processes



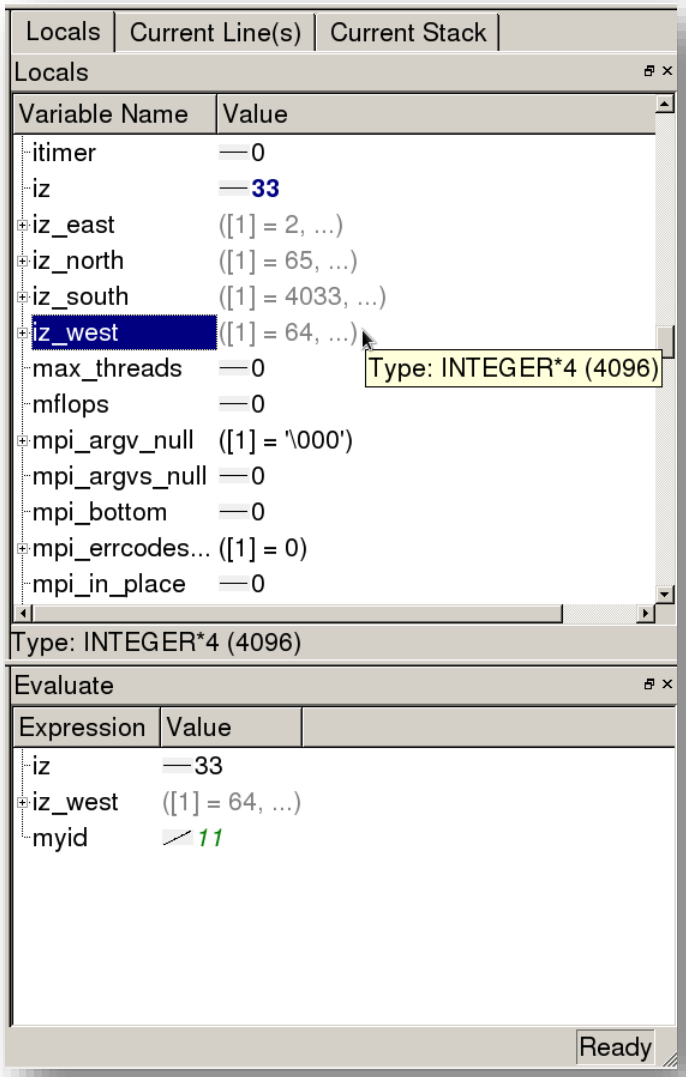
Local variables

- All the variables for the current stack frame
- Colors of variables change if their values change



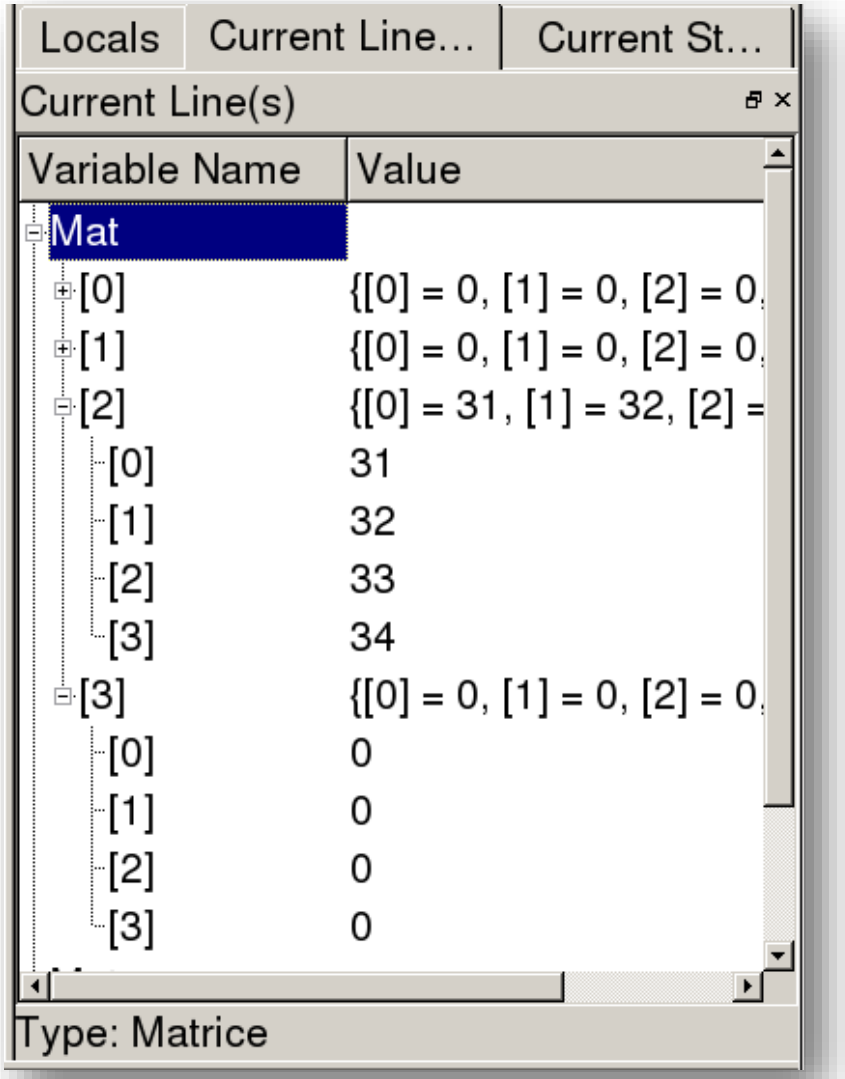
Keep track of specific variables

- You can choose variables to display in the Evaluate Window
- Select those values with *right-click > Evaluate value* option or by dragging them directly
- It will then be evaluated in whichever stack frame, thread or process you select
- Values in this window can be edited with *right-click > Edit value*



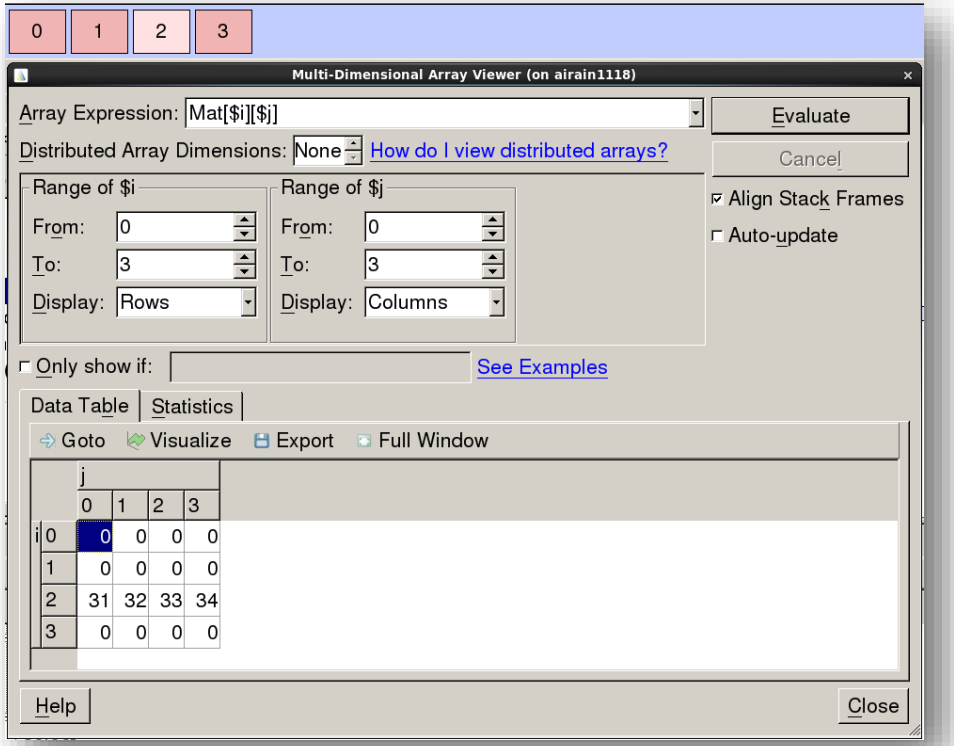
View array data

- Arrays are displayed as expandable values
- It is possible to expand them to view the content of each cell
- There is a viewer to display arrays more easily
- On a variable in the Source Code, Locals, Current Line(s) or Evaluate views do: *right-click > View Array*



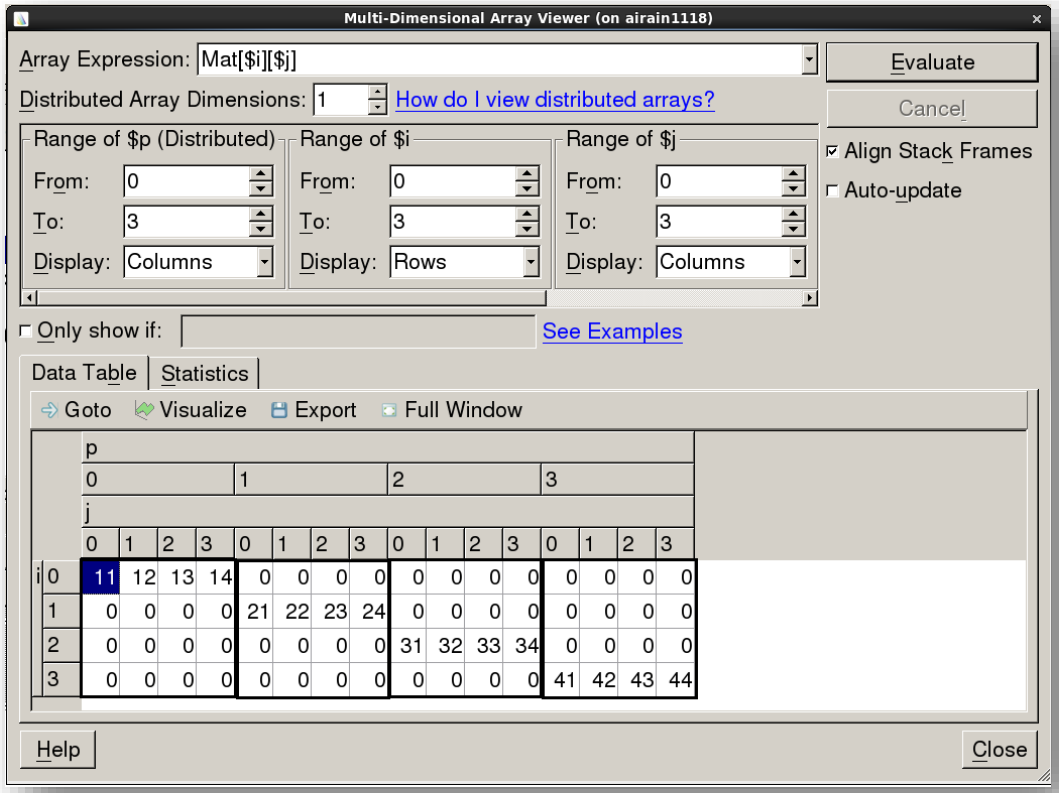
Multi dimensional array viewer

- Array Expression
  - If not set automatically, choose metavariables
    - [$i] [$j] for C/C++
    - ($i , $j) for Fortran
  - Choose range of those metavariables
  - Choose if metavariables are shown as row or column
- Filter displayed values with the “Only show if” condition



Multi dimensional array viewer

- Distributed arrays: Arrays distributed across processes as local arrays
- Add number of distribution dimensions. They will appear as new meta-variables



DDT

BREAKPOINTS, TRACEPOINTS, WATCHPOINTS

Breakpoints

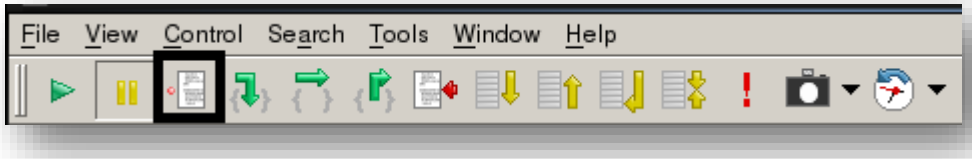
- One or several processes will be automatically paused when the execution reaches a breakpoint
- DDT will then ask you what to do. Usually, options are:
  - continue
  - pause
  - pause all
- Some default breakpoints will stop your program under certain conditions. For instance, if an abort or fatal error is detected
  - default breakpoints are modified through the menu *Control>Default Breakpoints*
- Users can define their own breakpoints

Setting a breakpoint

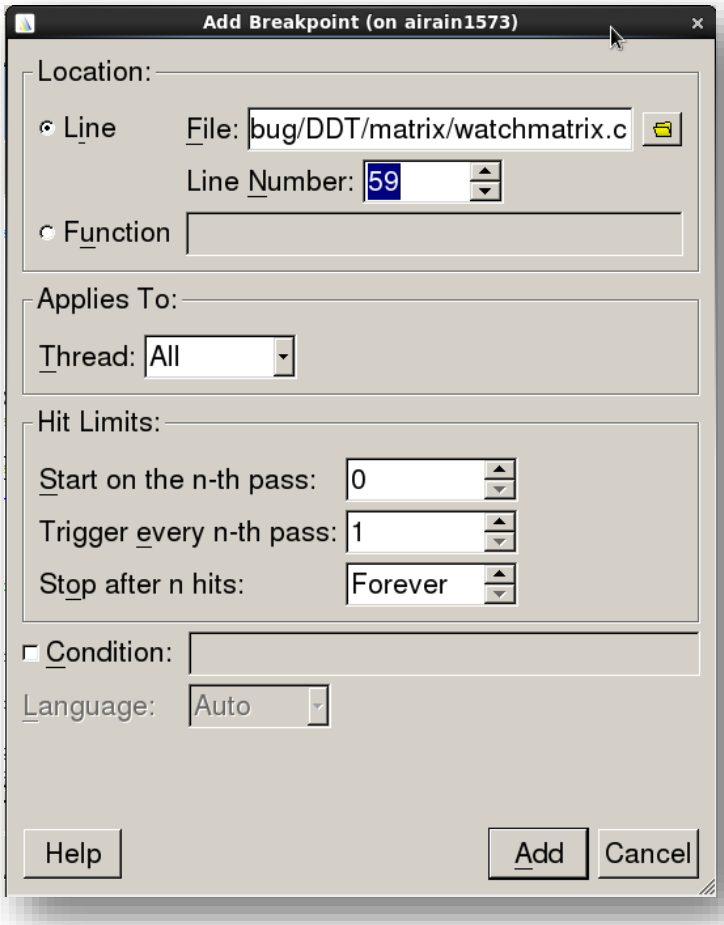
- Adding a breakpoint through the source code
  - *Right-click > Add breakpoint* on the line
  - Just click in the margin left to the line number
- It will add a breakpoint for every member of the currently selected group

Setting a breakpoint

- Adding a breakpoint through the control bar

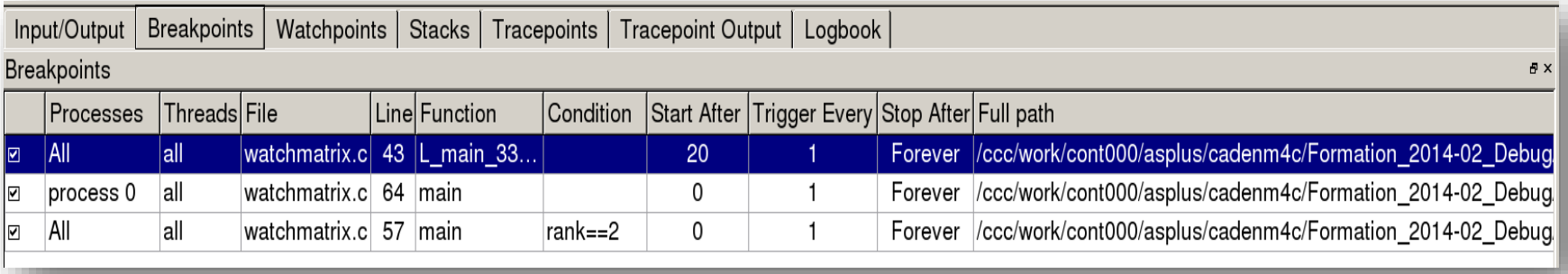


- It will open the *Add Breakpoint* window



- You can choose:
  - the file / line
  - function
  - what threads it applies to
  - conditions and limits

Managing breakpoints



	Processes	Threads	File	Line	Function	Condition	Start After	Trigger Every	Stop After	Full path
☑	All	all	watchmatrix.c	43	L_main_33...		20	1	Forever	/ccc/work/cont000/asplus/cadenm4c/Formation_2014-02_Debug
☑	process 0	all	watchmatrix.c	64	main		0	1	Forever	/ccc/work/cont000/asplus/cadenm4c/Formation_2014-02_Debug
☑	All	all	watchmatrix.c	57	main	rank==2	0	1	Forever	/ccc/work/cont000/asplus/cadenm4c/Formation_2014-02_Debug

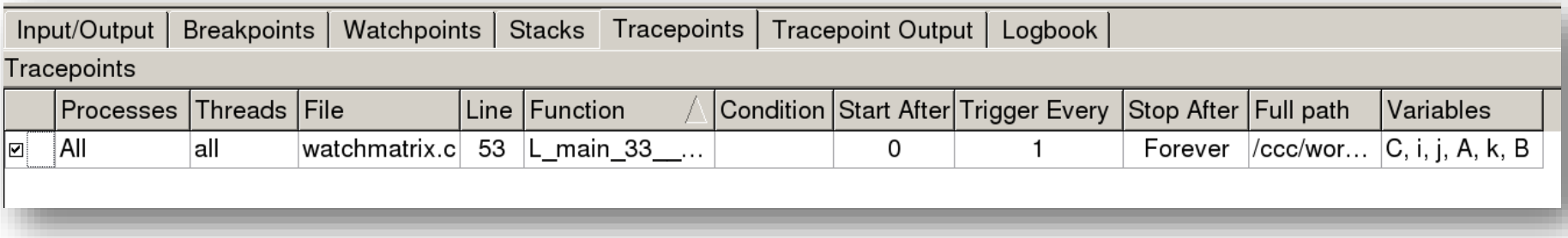
- Breakpoint are summarized in the lower window
- From there, managing breakpoints is easy
  - Enable/Disable
  - Edit
  - Remove

Tracepoints

- Tracepoints allow you to see the evolution of variables without stopping the program
- Whenever a thread reaches a tracepoint it will print to the Input/Output view:
  - the file and line number of the tracepoint
  - the selected variables values for the selected line
- Tracepoints can lead to considerable resource consumption. If it is set in a loop, you should set conditions to filter the generated data

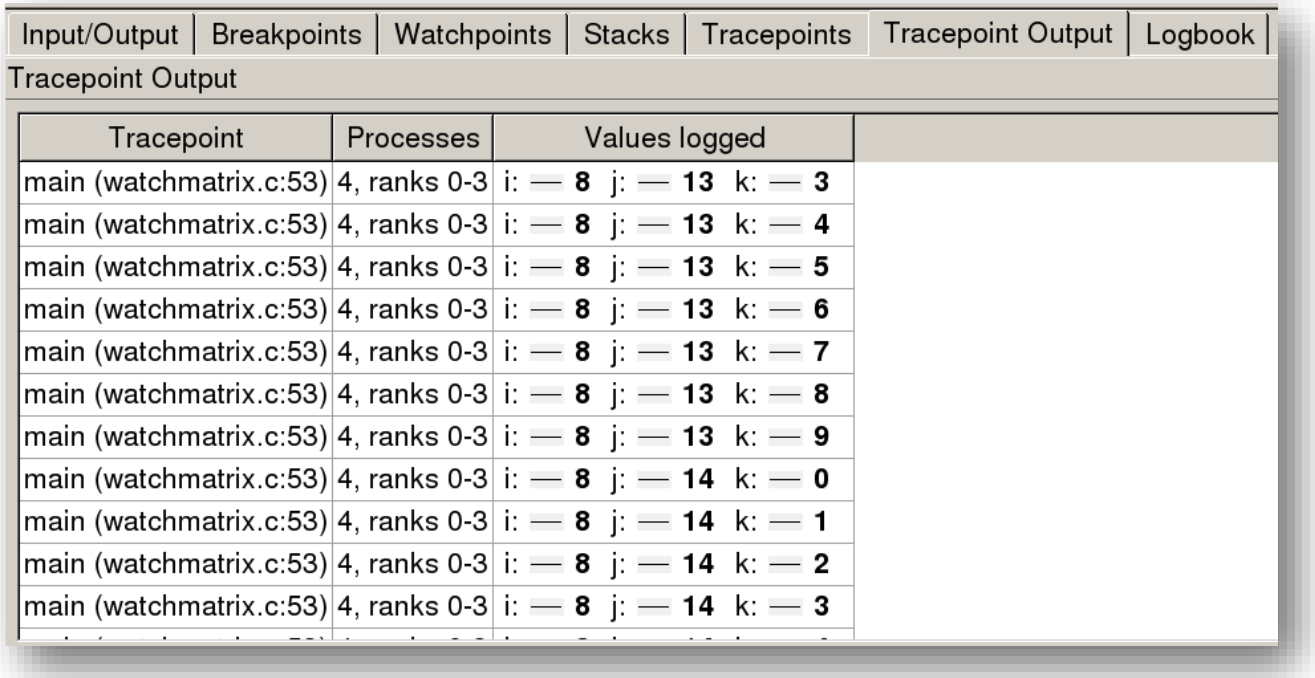
Setting a Tracepoint

- Adding a tracepoint through the source code
  - *Right-click > Add tracepoint for all* on the line
  - It will choose by default all the variables of the selected line



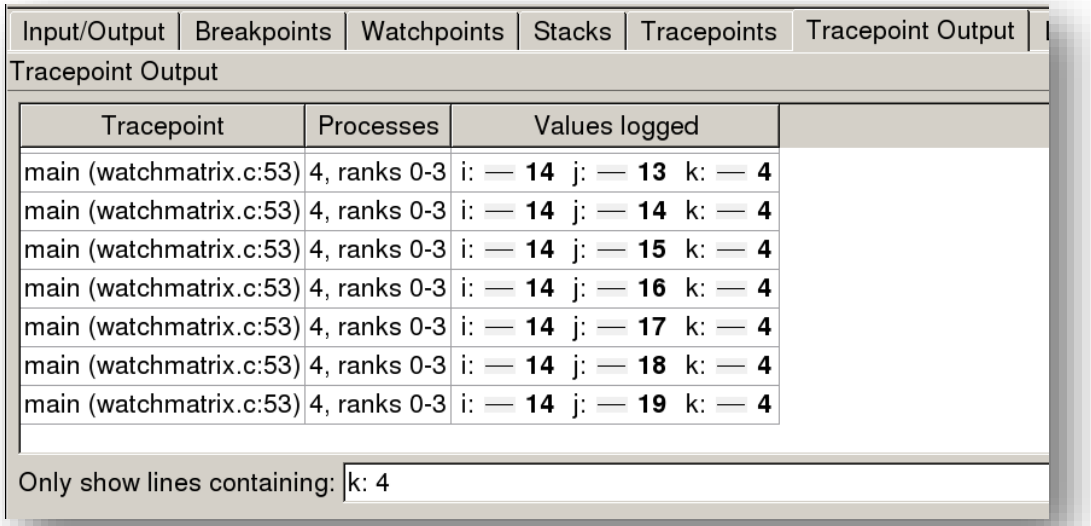
- *Right-click>Edit* on a tracepoint to edit parameters

Reading tracepoint output



- In the *Tracepoint Output* window, the selected values are displayed
- When tracepoints are passed by multiple processes within a short interval, the outputs are merged
- It is possible to save a HTML version of the output with *right-click>Save as HTML*

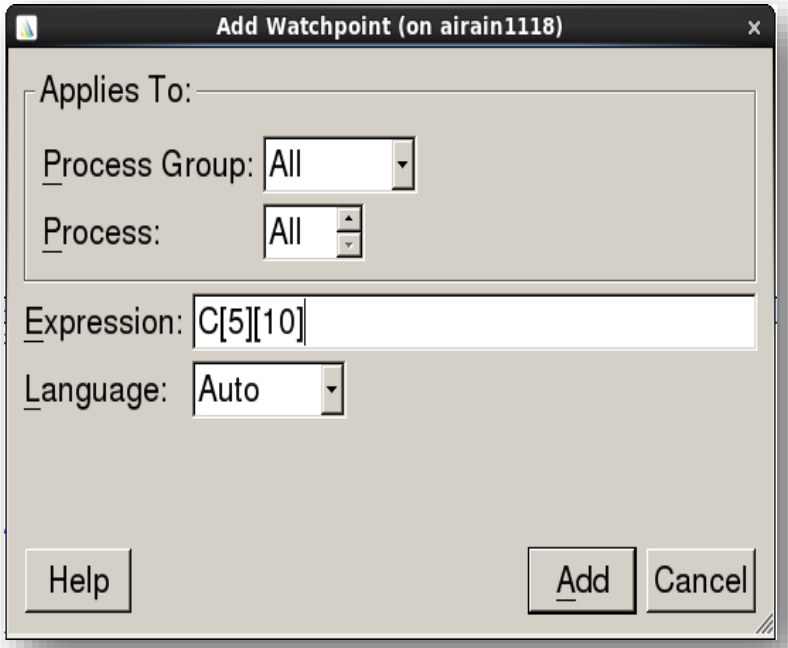
Reading tracepoint output



- To filter the tracepoint output, use the *Only show lines containing* box to:
  - search a pattern
  - search a value (a space is needed after the ":")

Watchpoints

- A watchpoint on a variable or expression will cause DDT to stop every time it changes
- Created by
  - right-clicking on the Watchpoints view and selecting the Add Watchpoint menu item
  - On any variable from the Local Variables, Current Line or Evaluate views, *right-click>Add watchpoint*



Watchpoints

- Program execution will stop as soon as the selected variable is modified
- DDT displays the old and new values

DDT

MEMORY DEBUGGING

Memory debugging

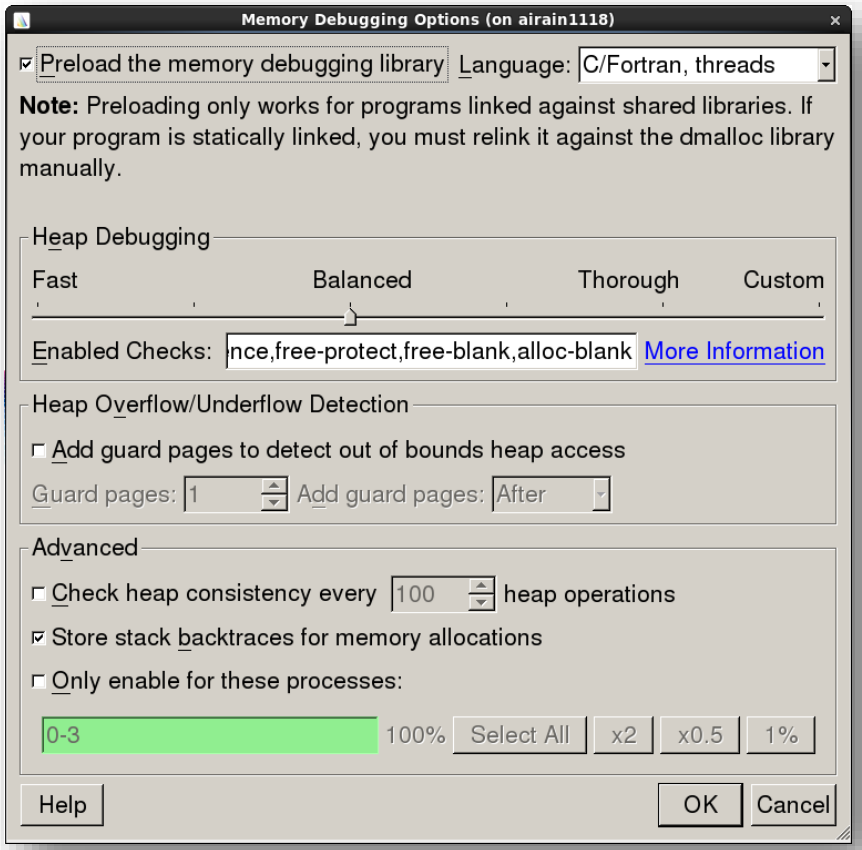
- DDT can intercept calls to the system memory allocation library, record memory usage, monitor correct usage of the library

- Helps detecting classical problems:
  - Memory exhaustion due to memory leaks
  - Random crashes caused by access to out of bound memory
  - Deallocation of invalid pointers

- Enabled by checking « Memory Debugging » when starting a run

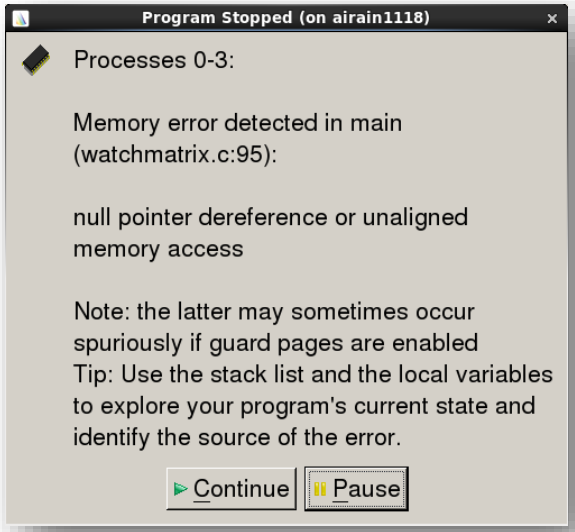
Memory debugging configuration

- Click on *Details* to configure the memory debugging
- *Preload the memory debugging library* is only possible if your code uses shared libraries
- The options for memory debugging have an overhead.



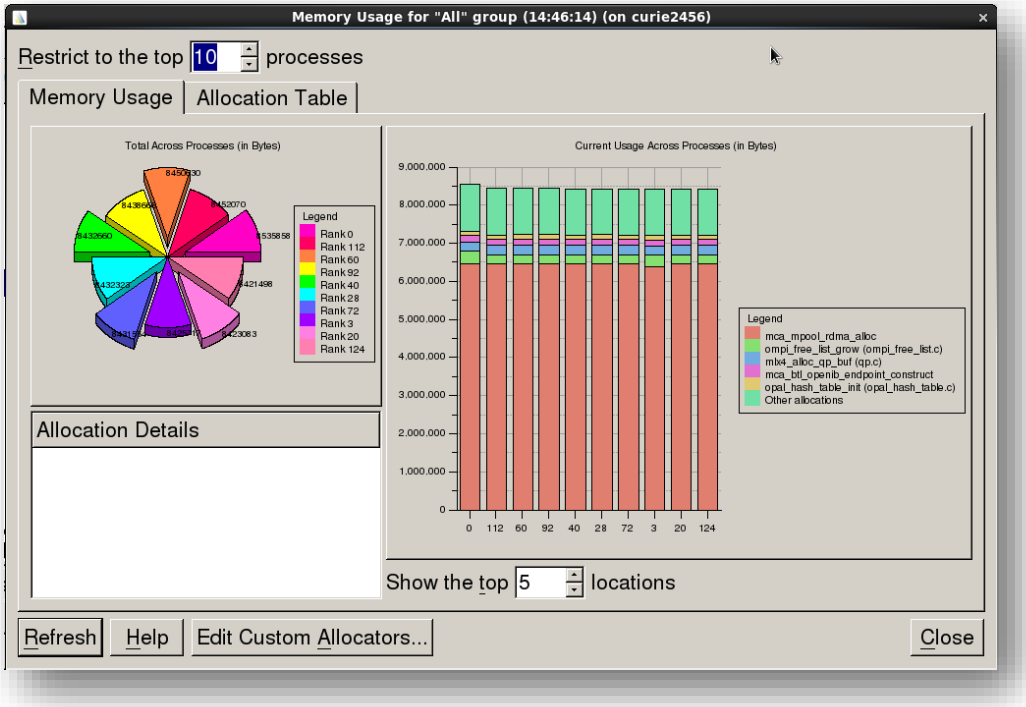
Memory debugging

- Each time an error is reported by the memory debugging library
  - Program stops
  - A short description of the detected problem is given
  - DDT asks what to do (Pause/Continue)



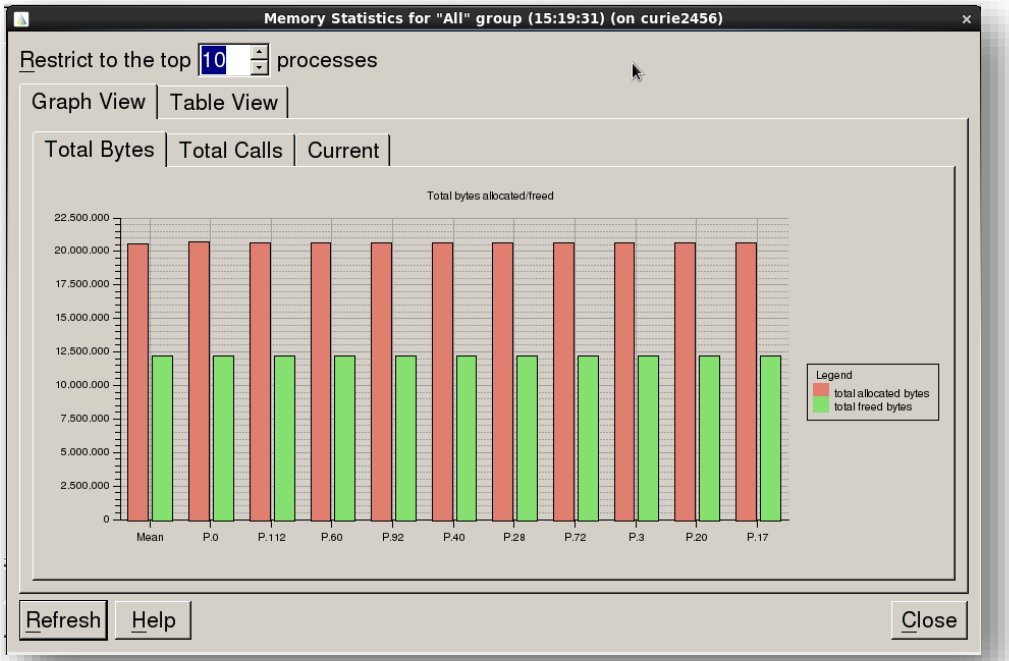
Current memory usage

- At any point during execution, it is possible to check the memory usage of each process
- From the menu: *Tools>Current Memory Usage*
- Only the most consuming processes are displayed



Current memory usage

- It is also possible to get statistics about allocations vs deallocation
- From the menu: *Tools>Overall Memory Stats*
- Useful to detect memory leaks



Conclusion

- A quickstart manual is available on the ccrt website:

  `https://www-ccrt.ccc.cea.fr/docs/Allinea_DDT_QuickStart.pdf`

- Complete documentation from the GUI: “Help > User Guide”

- DDT offers many possibilities regarding parallel debugging

- It should be scalable

- But keep in mind it is always easier to debug on small problems. Try to reduce your code as much as possible

MEMORY DEBUGGING WITH VALGRIND

Summary

1. User environment
2. Debugging: Overview
3. Compiler options
4. Debugging with GDB
5. Graphical debugging with DDT
6. **Memory debugging**
   1. **Tools to check memory consumption**
   2. **Valgrind and Memcheck**
   3. **Valgrind for parallel programs**
   4. **Other available debuggers**

Memory debugging

TOOLS TO CHECK MEMORY CONSUMPTION

Check memory consumption

- top

```
top - 14:26:42 up 37 days,  4:58, 74 users,  load average: 3.61, 3.34, 3.64
Tasks: 1381 total,   4 running, 1323 sleeping,  42 stopped,  12 zombie
Cpu(s):  7.1%us,  1.3%sy,  0.0%ni, 89.0%id,  2.6%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  132003416k total, 61344516k used, 70658900k free,   159680k buffers
Swap: 16384756k total,      640k used, 16384116k free, 28117928k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26312 cadenm4c  20   0 12780  448  344 R 99.6  0.0   3:49.16 crash
 6261 cadenm4c  20   0 20116 2360 1020 R  2.0  0.0   0:00.92 top
28120 cadenm4c  20   0  123m  18m 1484 S  0.0  0.0   0:00.16 sshd_user
28121 cadenm4c  20   0  110m 2212 1644 S  0.0  0.0   0:00.15 bash
```

  - VIRT: virtual memory used by the task
  - RES: physical (resident) memory used by a task
  - SHR: shared memory used by a task
  - %MEM: task's currently used share of available physical memory

- free
  - Displays amount of free and used memory in the system

- pmap <pid>
  - Reports memory map of a process

Check memory consumption

- SLURM
  - ccc_mremain <JobID>
    - Shows the remaining available memory for a parallel application
  - ccc_macct <JobID>

```
Memory / step
--------------
                        Resident Size (Mo)                      Virtual Size (Go)
JobID           Max     (Node:Task)         AveTask    Max  (Node:Task)             AveTask
-----------     ------------------------  -------    ---------------------------  -------
885395.batch      18 (airain1019  :   0)       0      1.00 (airain1019  :   0)    0.00
885395.0        1852 (airain1035  :   8)       0      4.00 (airain1019  :   0)    0.00
```

- These commands only give a punctual memory consumption
- No information about the part of code allocating and using the memory

Memory debugging

VALGRIND

Overview

- Valgrind is an instrumentation framework for building dynamic analysis tools
- It comes with a set of tools for debugging and profiling
- Here are some of the tools available:
  - **Memcheck** - a memory error detector (Default)
  - **Cachegrind** - a cache and branch-prediction profiler
  - **Callgrind** - a call-graph generating cache profiler
  - **Helgrind** - a thread error detector
  - **Massif** - a heap profiler

Memcheck

- Memcheck is the default and most current use of Valgrind

- In theory, no need to recompile, relink or modify the code but it works better with -g -O0

- To start memcheck

```
$ valgrind --tools=memcheck ./Exec <Args>
or just
$ valgrind ./Exec <Args>
```

Memcheck

- Valgrind/Memcheck options:
  - --leak-check=no|**summary**|full search for memory leaks at exit?
  - --leak-resolution=low|med|**high** differentiation of leak stack traces
  - --show-reachable=**no**|yes show reachable blocks in leak check?
  - --show-possibly-lost=no|**yes** show possibly lost blocks in leak check?
  - --undef-value-errors=no|**yes** check for undefined value errors
  - --track-origins=**no**|yes show origins of undefined values? [no]
- Options can be read from
  - ~/.valgrindrc
  - $VALGRIND_OPTS
  - ./.valgrindrc

Memcheck

- Different kinds of errors detected:
  - Reading/writing freed memory or incorrect memory areas
  - Uninitialized values
  - Incorrect freeing of memory, such as double freeing heap blocks
  - Misuse of functions for memory allocations: new(), malloc(), free(), deallocate(), etc.
  - Memory leaks - unintentional memory consumption

Most common errors

- Heap block overrun
  - Try to write a value out of bounds of the allocated array

```
==22860== Invalid write of size 4
==22860==    at 0x4005DD: func1 (test1.c:12)
==22860==    by 0x40061E: main (test1.c:20)
==22860==  Address 0x4c11068 is 0 bytes after a block of size 40 alloc'd
==22860==    at 0x4A05FDE: malloc (vg_replace_malloc.c:236)
==22860==    by 0x4005B0: func1 (test1.c:9)
==22860==    by 0x40061E: main (test1.c:20)
```

  - Try to read a value in an incorrect memory area

```
==15157== Invalid read of size 4
==15157==    at 0x4005FE: func1 (test1.c:16)
==15157==    by 0x40060E: main (test1.c:22)
==15157==  Address 0x4c1107c is not stack'd, malloc'd or (recently)
free'd
```

Most common errors

- Use of an uninitialized value

```
==3552== Conditional jump or move depends on uninitialized value(s)
==3552==    at 0x31D10459C7: vfprintf (in /lib64/libc-2.12.so)
==3552==    by 0x31D104EAC9: printf (in /lib64/libc-2.12.so)
==3552==    by 0x400619: func1 (test1.c:17)
==3552==    by 0x400628: main (test1.c:23)
```

- Launching Valgrind with the option *--track-origins=yes* will give more information about the uninitialized values:

```
==5564==  Uninitialized value was created by a heap allocation
==5564==    at 0x4A05FDE: malloc (vg_replace_malloc.c:236)
==5564==    by 0x4005B0: func1 (test1.c:9)
==5564==    by 0x400628: main (test1.c:23)
```

- This error only appears if the uninitialized value is being interpreted, for example by a print

Most common errors

- Overlap in copy
  - Copying data on overlapping blocks may result in undefined behavior
  - Even if they do not always generate errors, Valgrind signals them with the following message

```
==25191== Source and destination overlap in memcpy(0x4c100b0, 0x4c100b8, 10)
==25191==    at 0x4A07F24: memcpy (mc_replace_strmem.c:628)
==25191==    by 0x4006D0: func1 (test1.c:20)
==25191==    by 0x4006FB: main (test1.c:29)
```

- Invalid free: when heap memory with the wrong function or twice

```
==26169== Invalid free() / delete / delete[]
==26169==    at 0x4A0595D: free (vg_replace_malloc.c:366)
==26169==    by 0x40069D: main (test1.c:28)
==26169==  Address 0x4c10040 is 0 bytes inside a block of size 40 free'd
==26169==    at 0x4A0595D: free (vg_replace_malloc.c:366)
==26169==    by 0x400691: main (test1.c:27)
```

Most common errors

- Memory leak
  - The summary at the end of its output can help detecting memory leaks

```
==15157== HEAP SUMMARY:
==15157==     in use at exit: 40 bytes in 1 blocks
==15157==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==15157==
==15157== LEAK SUMMARY:
==15157==    definitely lost: 40 bytes in 1 blocks
==15157==    indirectly lost: 0 bytes in 0 blocks
==15157==      possibly lost: 0 bytes in 0 blocks
==15157==    still reachable: 0 bytes in 0 blocks
==15157==         suppressed: 0 bytes in 0 blocks
==15157== Rerun with --leak-check=full to see details of leaked memory
```

  - To identify the origin of the leak, rerun valgrind with the option --leak-check=full

```
==25483== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==25483==    at 0x4A05FDE: malloc (vg_replace_malloc.c:236)
==25483==    by 0x4005B0: func1 (test1.c:9)
==25483==    by 0x40061E: main (test1.c:20)
```

Create suppression files

- As the function complexity grows, Valgrind may detect more and more false positive
- Making the output harder to analyze
- Errors that are known to be false positives may be suppressed thanks to suppression files
- The installation contains a default suppression file:

```
$VALGRIND_LIBDIR/valgrind/default.supp
```

- You can create custom files

Create suppression files

- To select what to put in the suppression file, start your program under Valgrind with the option *--gen-suppressions=all*

- For each detected error, it will display the corresponding syntax to add in your suppression file

```
==5981== Source and destination overlap in memcpy(0xce8b780, 0xce8b788, 10)
==5981==    at 0x4A091D0: memcpy (mc_replace_strmem.c:878)
==5981==    by 0x400AF9: func1 (test1.c:22)
==5981==    by 0x400B5E: main (test1.c:36)
==5981==

{
   <insert_a_suppression_name_here>
   Memcheck:Overlap
   fun:memcpy
   fun:func1
   fun:main
}
```

Copy this paragraph in your file "my_valgrind.supp"

Use suppression files

- To use the suppression file created, start valgrind with the option *--suppressions*

```
valgrind --suppressions=./my_valgrind.supp ./Exec <Args>
```

- Launching valgrind with the verbose option *-v* will list the error suppressed during the run

```
--98282-- used_suppression:      1 Suppression - My_Close
--98282-- used_suppression:      1 Suppression - My_Write
```

Memory debugging

VALGRIND FOR PARALLEL PROGRAMS

Run MPI programs with Valgrind

- To launch your MPI program with Valgrind in a submission script:

```
ccc_mprun <mpi_opts> valgrind <vg_opts> ./Exec <Args>
```

- This default run will generate information for each process
- It will also probably display lots of detected memory errors on MPI libs
- There are ways to make the output more readable

Manage output

- By default, all the tasks write their output to the same file. It can be difficult to read
- To differentiate the tasks generating the messages
  - ccc_mprun **-l** valgrind …
    - Adds a label identifying the task before each output
    - Allows to filter the output after the run
  - ccc_mprun -E "-e *valgrind_%j_%t.e*" valgrind …
    - Creates one error file per task
    - %j: jobid - %t: taskid
  - ccc_mprun valgrind --log-file=*valgrind_%q{FOO}* …
    - All the valgrind output is redirected to the specified file
    - %q allows to differentiate the output thanks to an environment variable. For example, *SLURM_PROCID* for MPI jobs on Cobalt.

Irrelevant MPI messages

- There are lots of memory issues detected in MPI calls
- For instance, in mpi_init:

```
==81277==    by 0x8231614: mca_pml_ob1_add_procs (pml_ob1.c:326)
==81277==    by 0x4C5D3F2: ompi_mpi_init (ompi_mpi_init.c:761)
==81277==    by 0x4C72BEF: PMPI_Init (pinit.c:84)
==81277==    by 0x400C50: main (buffered.c:31)
```

- If the Valgrind installed has MPI support, there should be a wrapper available to suppress those messages: libmpiwrap-amd64-linux.so

Memory debugging

OTHER AVAILABLE MEMORY DEBUGGERS

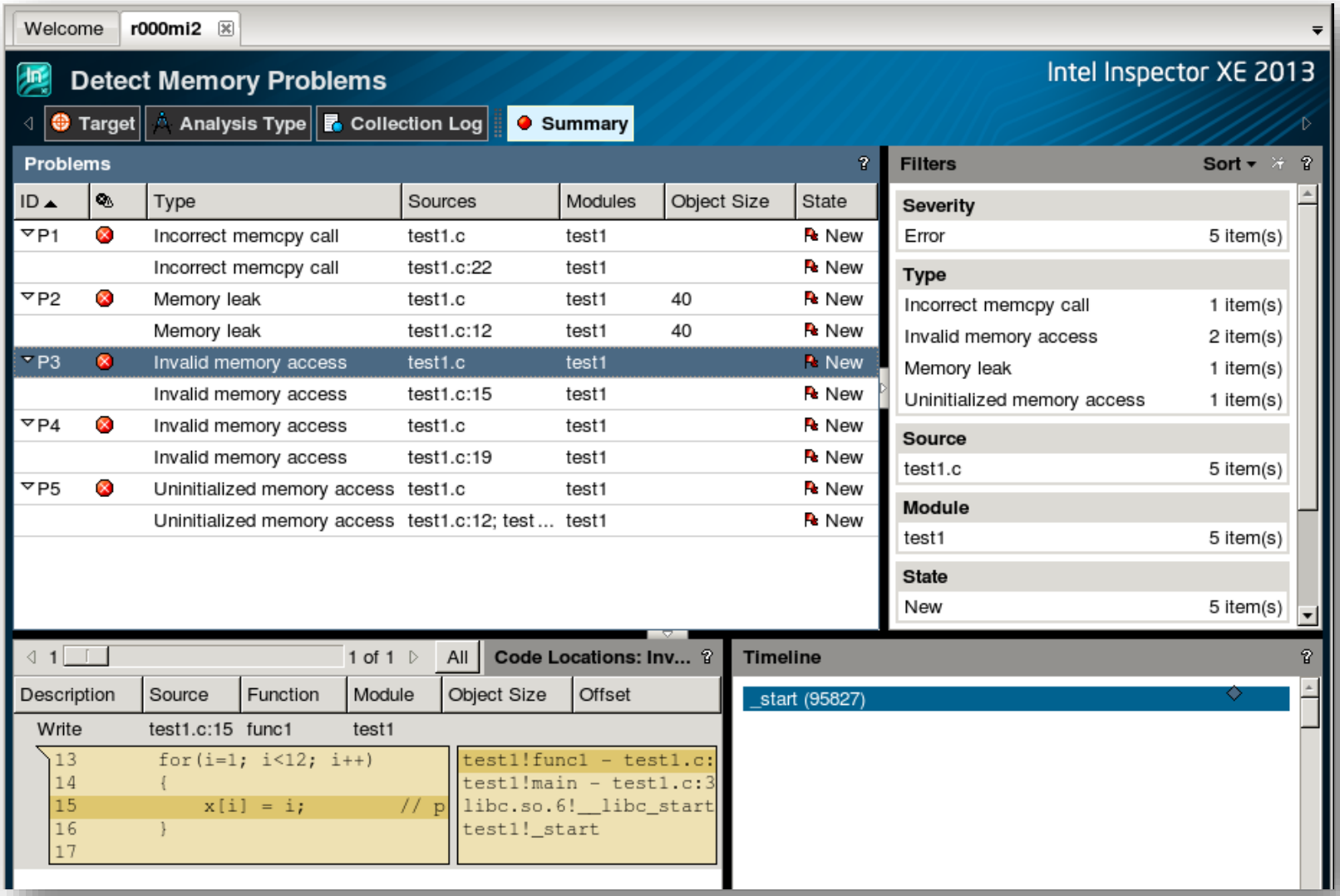
Electric fence

- Electric Fence is a library that detects accesses to invalid memory areas
- It will generate segfault each time those errors are detected
- To run your code with Electric fence:

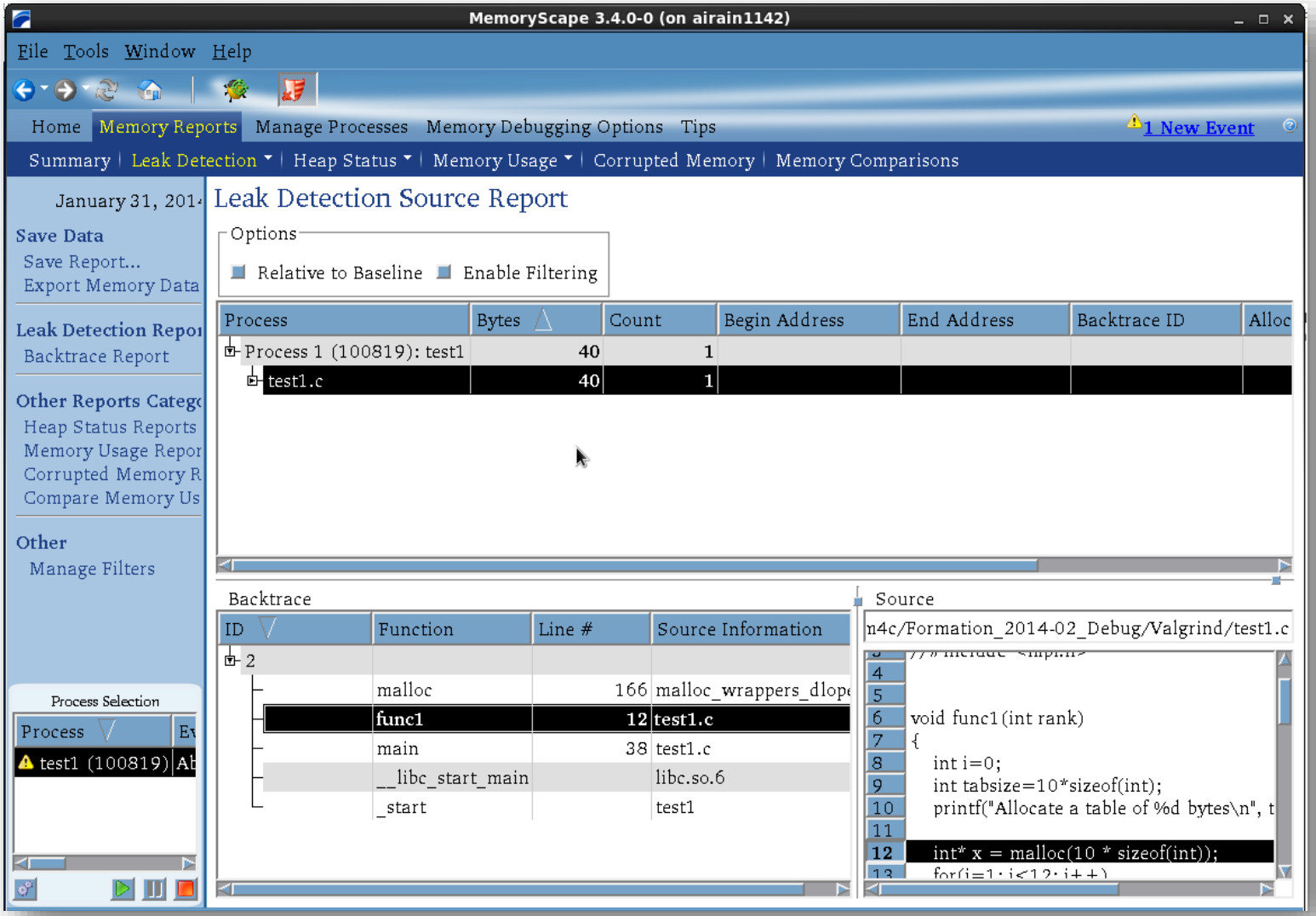
```
module load electricfence
LD_PRELOAD=$EF_PRELOAD ./Exec
```

- Every call to ccc_mprun automatically adds this LD_PRELOAD variable, thanks to the BRIDGE_MPRUN_PRELOAD variable that is set by the electricfence module.
- Useful when combined with a debugger

Intel Inspector



Totalview MemoryScape



Conclusion

- There are lots of tools for debugging
- Some are more appropriate than others depending on the code
- There are more available debugging tools installed on the computing center: Totalview, Inspector, memleax,…
  - To display them all:

```
$ module search debugger
```

- Reminder: Always simplify your code as much as possible before debugging

DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

CCRT TRAINING
PROFILING TOOLS

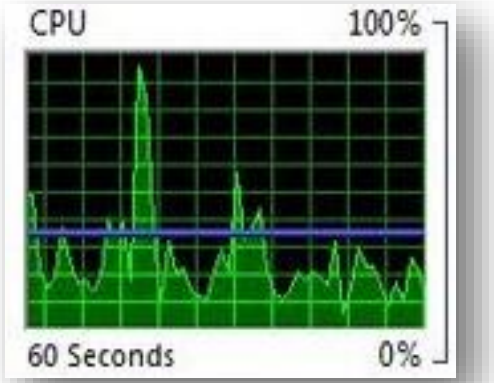
Summary

1. Profiling: overview
2. Simple code profiling
3. Score-P & Vampir
4. Profiling with Vtune

PROFILING: OVERVIEW

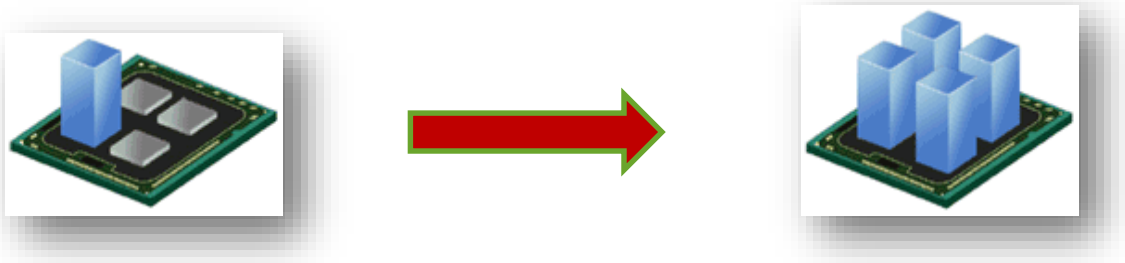
Profiling: General statements

- General issues:
  - Sequential
    - Algorithm
    - Data structures (cache misses, inhibit compiler optimizations…)
    - I/O
    - …



Profiling: General statements

- General issues:
  - Sequential to parallel
    - Algorithm
    - Multithreading
    - Partitioning
    - Communication
    - Synchronization
    - I/O



Profiling: General statements

- General issues:
  - Parallel to massively parallel
    - Algorithm
    - Multithreading
    - Partitioning ++
    - Communication ++
    - Synchronization ++
    - I/O ++

Profiling: General statements

- Performance issues/bottleneck
  - Compute (low InstructionPerCycle, …)
  - Memory (cache, locality, volume, false sharing…)
  - Communications (Frequency, volume, synchronization, …)
  - Inputs/Outputs (volume, not suited to parallel file system, …)
  - Overheads (threading strategy, …)
  - Load balancing (decomposition, scheduling, …)
  - Scalability

Profiling: General statements

- Solution:
  - Measure
  - Find hotspots and bottlenecks
  - Understand the causes
  - Modify
  - Compare
  - Validate

Profiling the code!

Profiling: General statements

- Achieved by instrumenting source code or binary
- Different technics to collect data (not exclusive!)
  - “Hand-made” instrumentation (code + external)
  - Event-based
  - Statistical (sampling)
  - Instrumented
- Best strategy: none. Depends on the issue you face or what you are looking for

Profiling: General statements

- What can be collected
  - Durations
  - Performance counters
  - Call stack/graph
  - Communications (numbers, size)
  - Memory throughput, memory accesses, cache misses
  - Latencies
  - Electrical consumption
  - Etc.

Profiling: printf & print

- "Hand-made" profiling
  - C and Fortran functions: printf and print
    - Output information you consider relevant
    - Limit their use to profiling/debug sessions (preprocessor directives)

```
double* tab;
long long int size;
tab = (double*) calloc(SIZE, sizeof(double));

#ifdef PROF
     size=SIZE*sizeof(double);
     printf("Size of allocation: %ld MB\n",
            size/1024/1024);
#endif
```

Profiling: printf & print

- "Hand-made" profiling
  - C and Fortran functions: printf and print
    - Test usage

```
$ icc -DPROF -o exe mycode.c
$ ifort -fpp -DPROF -o exe mycode.f90
$ ccc_mprun exe

Size of allocation: 7629 MB
```

    - Production usage

```
$ icc -o exe mycode.c
$ ifort -o exe mycode.f90
$ ccc_mprun exe
```

Profiling: C functions

- "Hand-made" profiling
  - C functions:
    - **time**: Get the current calendar time

```
time_t time (time_t* timer);
```

    - **clock**: Returns the processor time consumed by the program

```
clock_t clock (void);
```

    - **Gettimeofday**: Get the time of day

```
int gettimeofday(timeval *tp, NULL)
```

    - …

Profiling: C functions

- "Hand-made" profiling
  - gettimeofday example:

```
double* tab;
#ifdef PROF
    struct timeval tim;
    gettimeofday(&tim, NULL);
    double t1=tim.tv_sec+(tim.tv_usec/1000000.0);
#endif
tab = (double*) calloc(SIZE, sizeof(double));

#ifdef PROF
    gettimeofday(&tim, NULL);
    double t2=tim.tv_sec+(tim.tv_usec/1000000.0);
    printf("%.6lf seconds elapsed\n", t2-t1);
#endif
```

Profiling: Fortran functions

- "Hand-made" profiling
  - Fortran functions:
    - **cpu_time**: CPU elapsed time in seconds

```
CPU_TIME(X)
```

    - **system_clock**: Determines the COUNT of a processor clock

```
CALL SYSTEM_CLOCK([COUNT, COUNT_RATE, COUNT_MAX])
```

Profiling: Fortran functions

- "Hand-made" profiling
  - system_clock example:

```
#ifdef PROF
     CALL SYSTEM_CLOCK(COUNT_RATE=nb_periodes_sec,
                                    COUNT_MAX=nb_periodes_max)
     CALL SYSTEM_CLOCK(COUNT=nb_periodes_initial)
#endif
     ! put code to time here
#ifdef PROF
     CALL SYSTEM_CLOCK(COUNT=nb_periodes_final)
     nb_periodes = nb_periodes_final - nb_periodes_initial
     IF (nb_periodes_final < nb_periodes_initial) &
          nb_periodes = nb_periodes + nb_periodes_max
     elapsed_time = REAL(nb_periodes) / nb_periodes_sec
#ifdef PROF
```

Profiling: MPI functions

- "Hand-made" profiling
  - MPI calls:
    - **MPI_Wtime**: Returns an elapsed time on the calling processor

```
double MPI_Wtime(void)          // C
DOUBLE PRECISION MPI_WTIME()    ! Fortran
```

    - **MPI_Wtick**: Returns the resolution of MPI_WTIME in seconds

```
double MPI_Wtick(void)          // C
DOUBLE PRECISION MPI_WTICK()    ! Fortran
```

Profiling: MPI functions

- "Hand-made" profiling
  - MPI_Wtime example:

```
MPI_Init(&argc,&argv);
double t1 = MPI_Wtime();
sleep(1);
double t2 = MPI_Wtime();
printf("MPI_Wtime measured a 1 second sleep to be:
           %1.2lf\n", t2-t1);
MPI_Finalize();
```

Profiling: OpenMP functions

- "Hand-made" profiling
  - OpenMP Call:
    - **omp_get_wtime**: Elapsed wall clock time in seconds

```
double omp_get_wtime(void);                    // C
double precision function omp_get_wtime()      ! Fortran
```

    - **omp_get_wtick**: Gets the timer precision, i.e., the number of seconds between two successive clock ticks

```
double omp_get_wtick(void);                    // C
double precision function omp_get_wtick()      ! Fortran
```

Profiling: OpenMP functions

- "Hand-made" profiling
  - omp_get_wtime example:

```
double t1,t2;
#pragma omp parallel shared(a,b,c) private(i,tid,t1,t2)
{
     tid = omp_get_thread_num();
     printf("Thread %d starting...\n",tid);
     t1 = omp_get_wtime();
     #pragma omp for
     for (i=0; i<N; i++)
            c[i] = a[i] + b[i];
     t2 = omp_get_wtime();
     printf("Thread %d: omp_get_wtime measured %lf s to
            complete the loop\n",tid,t2-t1);
}
```

Profiling: Tools

- Specific profiling tools
  - Using profiling tools is an efficient way to spot, understand and solve performance issues.
  - Advantageously replace (or at least complete) manual timings and prints
  - Many of them exist and can be used depending on what you want to investigate
  - Many of them are at your disposal at the TGCC and others can be added on demand

Profiling: Tools

- Profilers outputs
  - Profile: Statistical summary of observed events
    - Highlights hotspots in source code
    - Returns metrics (number of calls, time spent in calls, performance counters, etc.)
    - Returns derived metrics (instructions per cycle, flops, throughputs, etc.)
  - Trace: A stream of recorded events
    - More adapted to highly parallel code
    - Outputs a trace file which can be large
    - Graphical overview of execution (synchronization issues, …)

Profiling: Tools

- Event-based collection
  - Events can occur
    - On a hardware device (clock cycle, cache miss)
    - On the system
    - On the network
    - In an application (send/receive functions)
  - A profiler can accumulate and merge these data to get a performance report
    - By a sampling method
    - By an instrumenting method

Profiling: Tools

- Statistical collection
  - Probe data from different sources (hardware counters, program counters, etc) are retrieved at regular intervals using operating system interrupts
  - Less accurate
  - Overhead reduction
  - Often produce a good picture of the application as it is not intrusive and has low side effects (additional memory allocation, instructions)

Profiling: Tools

- Instrumented collection
  - Adds instruction to collect the required information
  - Accurate
  - Higher overhead
  - Can be automatic or manual
  - Can be at source level or at runtime
  - May require a re-compilation
  - Can not instrument system or third party libraries

Profiling: Hardware counters

- Hardware counters are a set of special-purpose registers built into microprocessors to store the counts of hardware-related activities within computer systems
- They are very reliable to conduct low-level performance analysis or tuning
- They provide detailed performance information related to CPU's functional units
  - Cache miss / load / request
  - Number of instructions
  - Number of cycles
  - Branch instructions
  - Etc.

Profiling: Hardware counters

- We usually use Papi to collect hardware counters
- To see a list of available counters:

```
module load papi
papi_avail
```

```
 Name         Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
…
```

- Papi counter works with many profilers such as ScoreP or IPM.

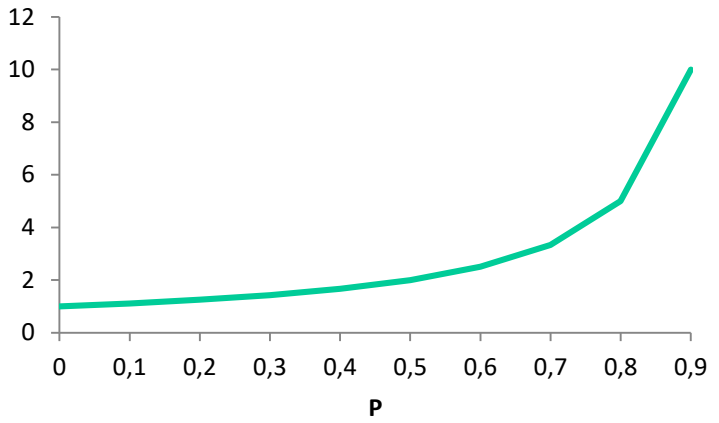
Profiling: General statements

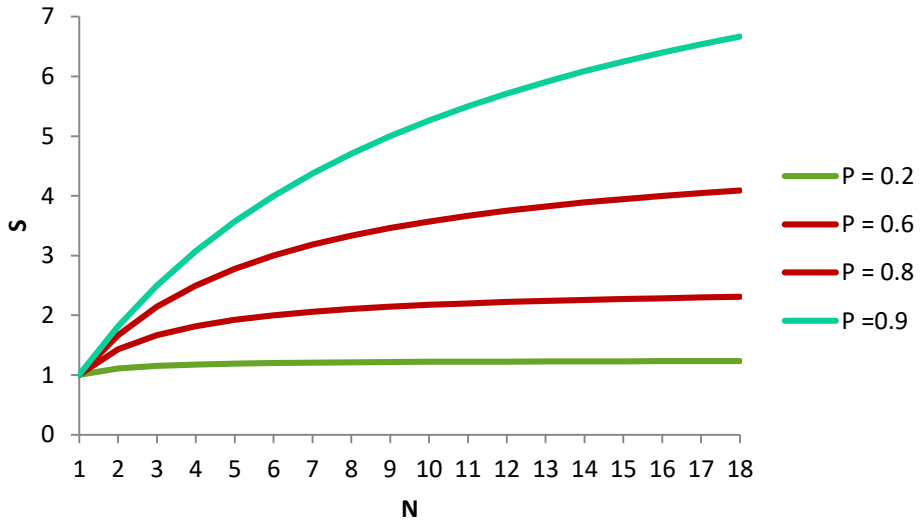
- Amdahl's law
  - The speedup is limited by the parallel fraction of the program P:

$$Speedup = \frac{1}{1-P}$$

  - When introducing the number of processors N, and the sequential section of the code S, the law becomes:

$$Speedup = \frac{1}{\frac{P}{N} + S}$$

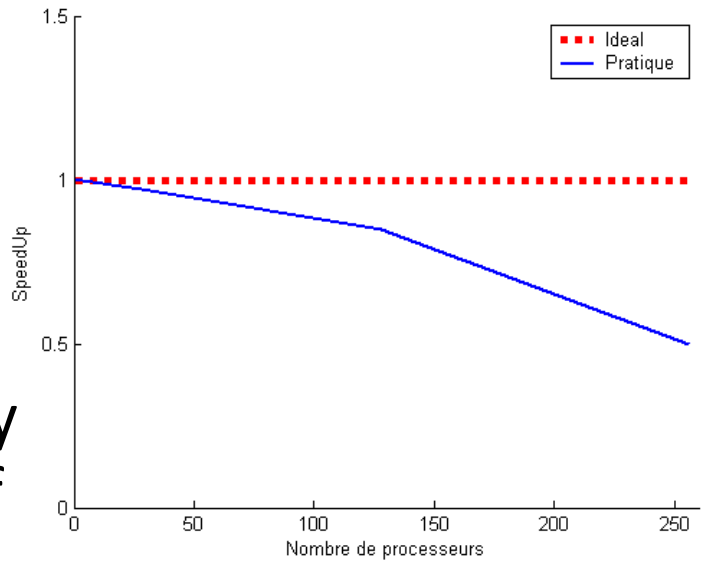




Strong scalability

Profiling: General statements

- Gustafson's law
  - Sets an upper limit to the speed up that can be reached using several processors to parallelize a code in a data parallel context
  - For a defined domain size for each process, scalability is ideal if execution time of the program remains the same as for a one process job



Weak scalability

Profiling: General statements

- The Pareto principle:
  - 80 % of the time is spent in 20 % of the code
  - 80 % of the speed up reached in 20 % of the programmer's effort
  - 80 % of the errors are located in 20 % of the code
  - Etc.

No need to spend huge amount of time to get interesting results

- Mixing tools helps finding and understanding performance problems

Profiling tools

	Supported programming model				Collectible events						Tool support		Type of profiling		
Name	MPI	OpenMP	Cuda	SIMD	Comm	I/O	Call graph	Hardware counters	Memory Usage	Cache Usage	Collection	GUI	Sampling	Tracing	Instrumentation
Advisor	✗	✓	✗	✓	✗	✗	✓	✗	✗	✗	✓	✓	✓	✗	✗
Allinea MAP	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✗
Cube	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✓	✓	✗	✓
Darshan	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗
Gprof	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✓	✓
HPCToolkit	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗
Igprof	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓	✗
IPM	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✗	✓	✗	✗
Paraver	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✓
PAPI	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✓	✗	✓	✗	✓
ScoreP	✓	✓	✓	✗	✓	✗	✓	✓	✗	✗	✓	✗	✓	✓	✓
Tau	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✗
ThreadSpotter	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗
Cachegrind	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗
Callgrind	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗	✗
Massif	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✗
Vampir	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✓
Vtune	✗	✓	✗	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✗	✗

SIMPLE AND LIGHT WEIGHT CODE PROFILING TOOLS

Summary

1. Profiling: overview

2. **Simple code profiling**
   1. **Gprof**
   2. **IPM**
   3. **Darshan**
   4. **MAP**

3. Score-P & Vampir

4. Profiling with Vtune

Simple and light weight code profiling tools

GPROF

Usage

- gprof produces an execution profile of C and Fortran programs
- A simple way to get a call graph and the amount of CPU time spent in each routine
- Steps to run your code under gprof:
  - Recompile your code with the option -pg
  - Run the code. It will generate the file “gmon.out”
  - Open it with

```
$ gprof ./Exec gmon.out
```

Usage

- The output is displayed in 2 main forms: flat profile and call graph
  - The flat profile shows how much time your program spent in each function, and how many times that function was called.
  - The call graph shows, for each function, which functions called it, which other functions it called, and how many times.

Flat profile

- To display only the flat profile:

```
$ gprof -p -b ./Exec gmon.out
```

- To display the flat profile of one specific routine:

```
$ gprof -p<routine> -b ./Exec gmon.out
```

- Lists the following information:
  - Time: Percentage of time spent in this routine
  - Cumulative: Total time of this routine plus the functions above this one in this table
  - Self: Time spent in this routine
  - Calls: Total number of times this function was called
  - Self ms/call: The average time spent in this function per call
  - Total ms/call: The average time spent in this function and its descendants per call

Flat profile

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ks/call  Ks/call  name
 25.02   4677.46  4677.46 3525238040     0.00     0.00  binvcrhs_
 15.95   7658.33  2980.87     8064     0.00     0.00  compute_rhs_
 14.82  10428.70  2770.37     8032     0.00     0.00  y_solve_
 14.48  13134.87  2706.16     8032     0.00     0.00  x_solve_
 14.00  15752.13  2617.26     8032     0.00     0.00  z_solve_
 11.95  17986.08  2233.95 4072549953     0.00     0.00  matmul_sub_
  2.05  18369.07   382.98 3515509281     0.00     0.00  matvec_sub_
  0.83  18523.30   154.23     8032     0.00     0.00  add_
  0.28  18575.78    52.48 74544003     0.00     0.00  lhsinit_
  0.28  18627.85    52.07 61795415     0.00     0.00  binvrhs_
  0.13  18652.23    24.39 28660458     0.00     0.00  exact_solution_
  0.09  18669.77    17.54    32128     0.00     0.00  copy_x_face_
  0.05  18678.83     9.06    31877     0.00     0.00  copy_y_face_
  0.05  18687.73     8.90       32     0.00     0.00  exact_rhs_
  0.03  18692.48     4.75       64     0.00     0.00  initialize_
  0.00  18693.18     0.70       32     0.00     0.00  rhs_norm_
```

Call graph

- To display only the call graph:

```
$ gprof -q -b ./Exec gmon.out
```

- To display the call graph of one specific routine:

```
$ gprof -q<routine> -b./Exec gmon.out
```

- Lists the following information:
  - %time: The percentage of the total time that was spent in this function and its children
  - Self: The total amount of time spent in this function. For the parent processes
  - Children: The total amount of time propagated into this function by its children
  - Called: The number of times the function was called

Call graph

```
-----------------------------------------------
                0.03 18616.02    8032/8032        MAIN__ [1]
[3]     99.6    0.03 18616.02    8032         adi_ [3]
             2770.37 2606.51    8032/8032        y_solve_ [4]
             2706.16 2462.64    8032/8032        x_solve_ [5]
             2617.26 2329.79    8032/8032        z_solve_ [6]
             2969.04    0.00    8032/8064        compute_rhs_ [8]
              154.23    0.00    8032/8032        add_ [11]
-----------------------------------------------
             2770.37 2606.51    8032/8032        adi_ [3]
[4]     28.8 2770.37 2606.51    8032         y_solve_ [4]
             1641.76    0.00 1237334872/3525238040     binvcrhs_ [7]
              788.66    0.00 1437742646/4072549953     matmul_sub_ [9]
              131.86    0.00 1210335776/3515509281     matvec_sub_ [10]
               22.74    0.00 26984199/61795415     binvrhs_ [13]
               21.50    0.00 30543377/74544003     lhsinit_ [12]
-----------------------------------------------
-----------------------------------------------
             1481.57    0.00 1116609151/3525238040     z_solve_ [6]
             1554.13    0.00 1171294017/3525238040     x_solve_ [5]
             1641.76    0.00 1237334872/3525238040     y_solve_ [4]
[7]     25.0 4677.46    0.00 3525238040         binvcrhs_ [7]
-----------------------------------------------
```

Simple and light weight code profiling tools

IPM

Overview

- IPM is a portable profiling infrastructure for MPI codes
- It is very scalable, has low overhead and is extremely easy to use
- Provides performance and resource usage information for communications, computation, and IO
- Requires no source code modification or specific compiler options

Usage

- To enable IPM dynamically, just load the IPM module and add make sure to use ccc_mprun:

```
module load ipm
ccc_mprun ./test
```

- Every call to ccc_mprun appends IPM_PRELOAD to LD_PRELOAD, so that the code is profiled automatically.

- IPM generates a profile trace in an xml file and a profile summary in the standard output file

Standard output

```
##IPMv0.983##############################################################################
#
# command : /usr/local/gromacs-4.6.6/bin/mdrun_mpi -stepout 25000 -deffnm input -rdd 1.4
# host    : curie6162/x86_64_Linux         mpi_tasks : 16 on 8 nodes
# start   : 02/24/15/11:33:43              wallclock : 182.508046 sec
# stop    : 02/24/15/11:36:45              %comm     : 5.72
# gbytes  : 8.06393e+00 total              gflop/sec : 1.40031e+00 total
#
##############################################################################
# region  : *       [ntasks] =     16
#
#                          [total]         <avg>           min           max
# entries                       16             1             1             1
# wallclock                2920.13       182.508       182.508       182.508
# user                     22693.8       1418.36       1416.83       1420.18
# system                   251.176       15.6985        13.043       17.2944
# mpi                      166.887       10.4304       6.66851       11.6311
# %comm                                  5.71506       3.65382       6.37291
# gflop/sec                1.40031     0.0875197     0.0873254     0.0876713
# gbytes                   8.06393      0.503996      0.494202      0.637413
#
# PAPI_FP_OPS           9.93193e+11   6.20746e+10   6.15695e+10   6.29053e+10
# PAPI_FP_INS           9.93193e+11   6.20746e+10   6.15695e+10   6.29053e+10
# PAPI_DP_OPS           2.55569e+11    1.5973e+10   1.59376e+10   1.60007e+10
# PAPI_VEC_DP           5.75289e+07   3.59556e+06   3.46105e+06    3.7458e+06
#
#                           [time]       [calls]       <%mpi>      <%wall>
# MPI_Sendrecv              94.8594    1.6057e+06         56.84         3.25
# MPI_Waitall               40.5859        895472         24.32         1.39
# MPI_Bcast                 23.3455         32016         13.99         0.80
# MPI_Isend                 2.16025   1.70196e+06          1.29         0.07
# MPI_Gatherv               1.64405            48          0.99         0.06
# MPI_Scatter               1.49911            16          0.90         0.05
# MPI_Scatterv              1.16537            48          0.70         0.04
# MPI_Irecv                0.668721   1.70196e+06          0.40         0.02
# MPI_Allreduce             0.64684         11328          0.39         0.02
# MPI_Reduce               0.282225         22464          0.17         0.01
# MPI_Send                0.0112959          4456          0.01         0.00
# MPI_Recv                0.0108482          4456          0.01         0.00
# MPI_Gather             0.00499536           316          0.00         0.00
# MPI_Barrier            0.00260794            32          0.00         0.00
# MPI_Comm_size          2.13857e-05            80          0.00         0.00
# MPI_Comm_rank          1.37705e-05            80          0.00         0.00
##############################################################################
```

Job execution summary

Memory usage and time summary

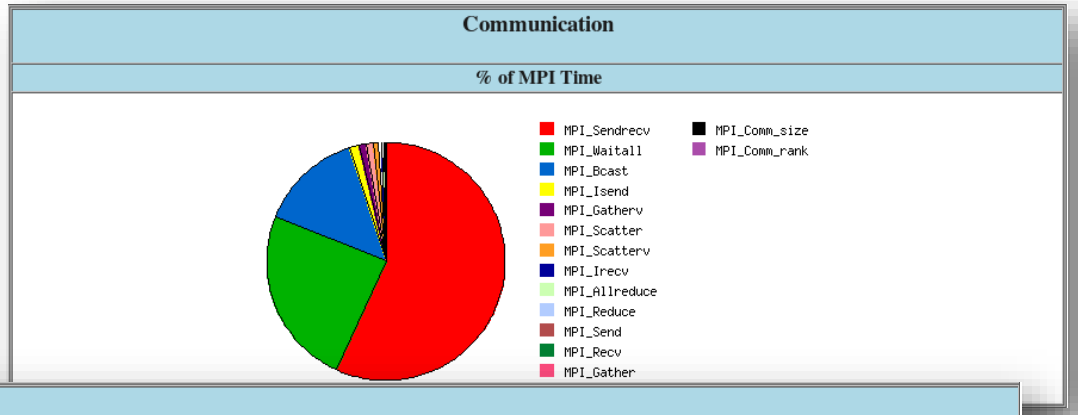
Time spent in MPI calls

Post process

- The XML file can be used to generate an html page with the command:

```
$ ipm_parse -html XML_File
```

- Will generate a graphical html page representing the results obtained



Communication Event Statistics (100.00% detail, -8.0975e-06 error)

	Buffer Size	Ncalls	Total Time	Min Time	Max Time	%MPI	%Wall
MPI_Bcast	1064	16	11.965	7.974e-01	8.023e-01	7.17	0.41
MPI_Bcast	200	32	6.109	1.677e-05	4.387e-01	3.66	0.21
MPI_Waitall	960	58120	4.666	1.125e-07	9.134e-01	2.80	0.16
MPI_Waitall	600	74002	4.022	1.083e-07	9.077e-01	2.41	0.14
MPI_Waitall	840	60637	3.859	1.155e-07	9.100e-01	2.31	0.13
MPI_Sendrecv	8	271184	3.573	4.680e-07	9.283e-03	2.14	0.12
MPI_Waitall	1080	49778	3.429	1.141e-07	9.094e-01	2.05	0.12

Simple and light weight code profiling tools

DARSHAN

Overview

- Darshan is a light weight IO profiling tool capable of profiling POSIX IO, MPI IO and HDF5 IO
- It is very scalable, has low overhead and is extremely easy to use
- Requires no source code modification or specific compiler options

Usage

- Here are the two steps to profile a code with Darshan:
  - Add the Darshan library in LD_PRELOAD. This is done automatically by ccc_mprun when the module is loaded
  - Specify where you want the Darshan trace to be created by exporting the variable DARSHAN_LOG_PATH

Usage

Submission script example

```
#!/bin/bash
#MSUB -r MyJob_Para              # Request name
#MSUB -n 32                      # Number of tasks to use
#MSUB -T 1800                    # Elapsed time limit in seconds
#MSUB -o example_%I.o            # Standard output.
#MSUB -e example_%I.e            # Error output. %I is the job ID
#MSUB -q broadwell               # Queue

module load darshan
export DARSHAN_LOG_PATH=$PWD

ccc_mprun ./prog.exe
```

Post process

- Once the job has finished, a trace will be generated in the specified directory

```
<USERNAME>_<BINARY_NAME>_<JOB_ID>_<..>.darshan.gz.
```

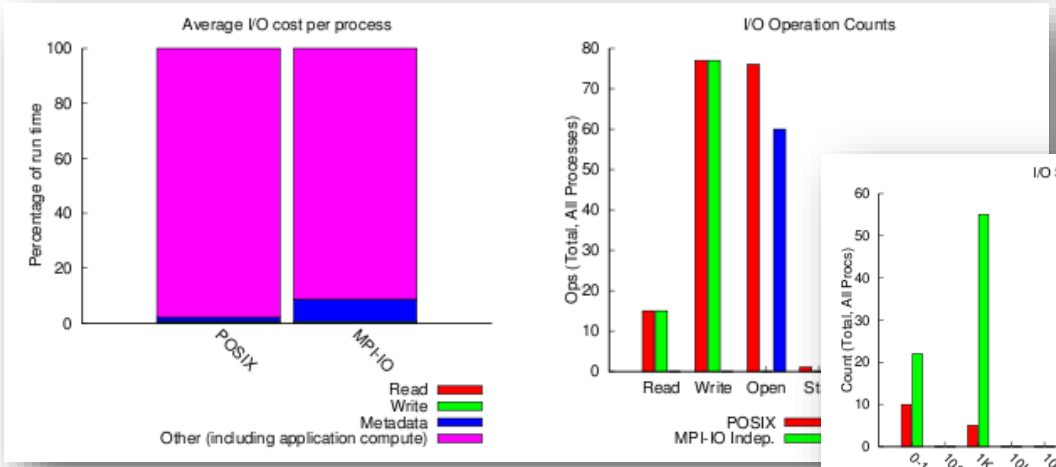
- There are several scripts available to analyze the trace:
  - darshan-parser
    - Gives a full, human readable dump of all information contained in a log file
    - Lists the different files accessed and the type of file systems used

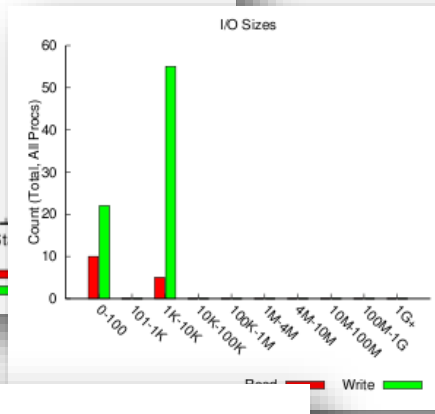
```
$ darshan-parser *.darshan.gz > example_output.txt
```

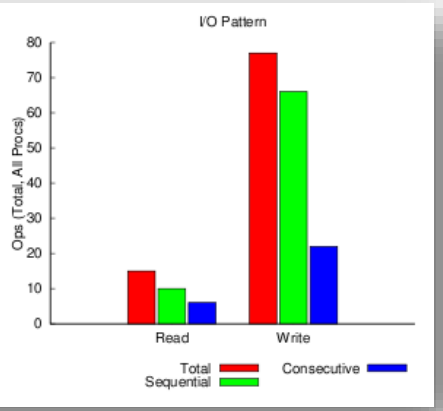
Post process

- darshan-job-summary.pl
  - Generates a graphical summary of the I/O activity for the job

```
$ darshan-job-summary.pl *.darshan.gz
```







Most Common Access Sizes

access size	count
7128	48
4	32
7920	12

File Count Summary

type	number of files	avg. size	max size
total opened	13	33K	36K
read-only files	1	36K	36K
write-only files	11	36K	36K
read/write files	0	0	0
created files	11	36K	36K

Post process

- darshan-summary-per-file.sh
  - Similar to darshan-job-summary.pl except that it produces a separate pdf summary for every file accessed by the application
  - The summaries will be written in the directory specified as argument

```
$ darshan-summary-per-file.sh *.darshan.gz output_dir
```

Simple and light weight code profiling tools

MAP

Overview

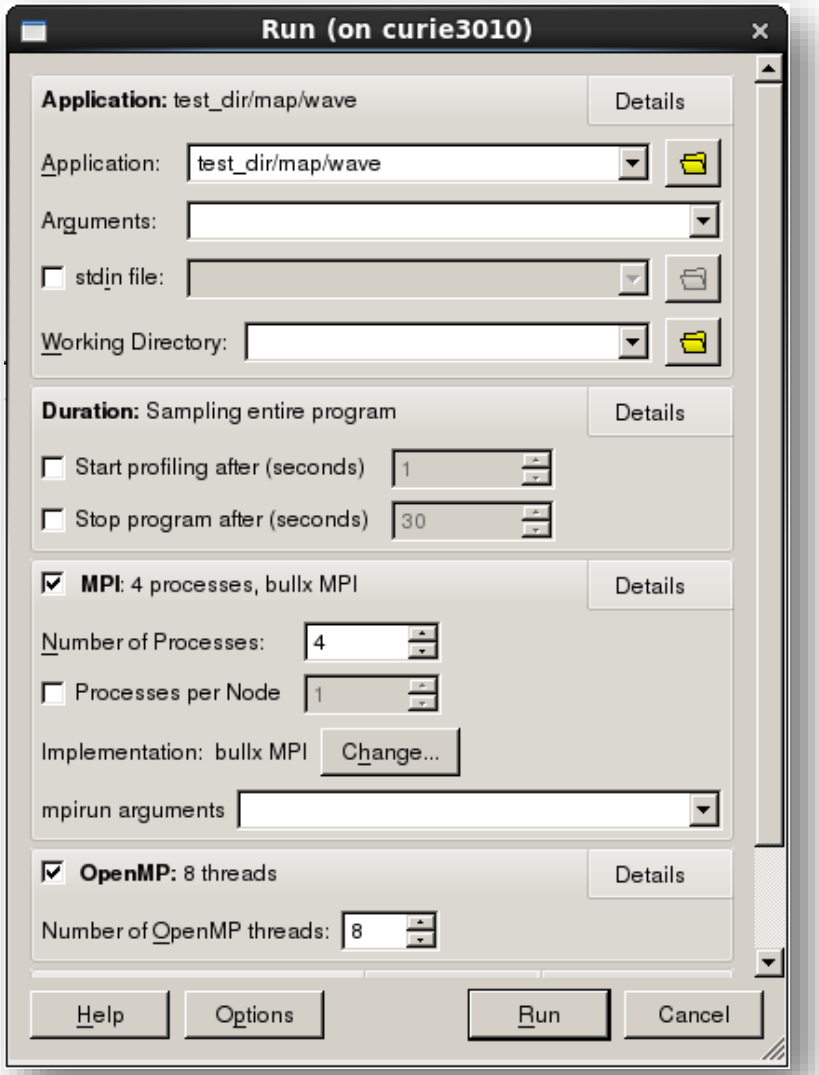
- Part of ARM-Forge
- Highly scalable and easy to use
- Preconfigured set of metrics showing:
  - memory usage
  - floating-point calculations
  - percentage of vectorization
  - MPI usage
  - …

Launch method

- Direct launch
  - In a submission script

```
module load arm-forge
map ./prog
```

  - Will open a start window
  - Change parameters if necessary and hit “Run”



Launch method

- Launch offline
  - In a submission script

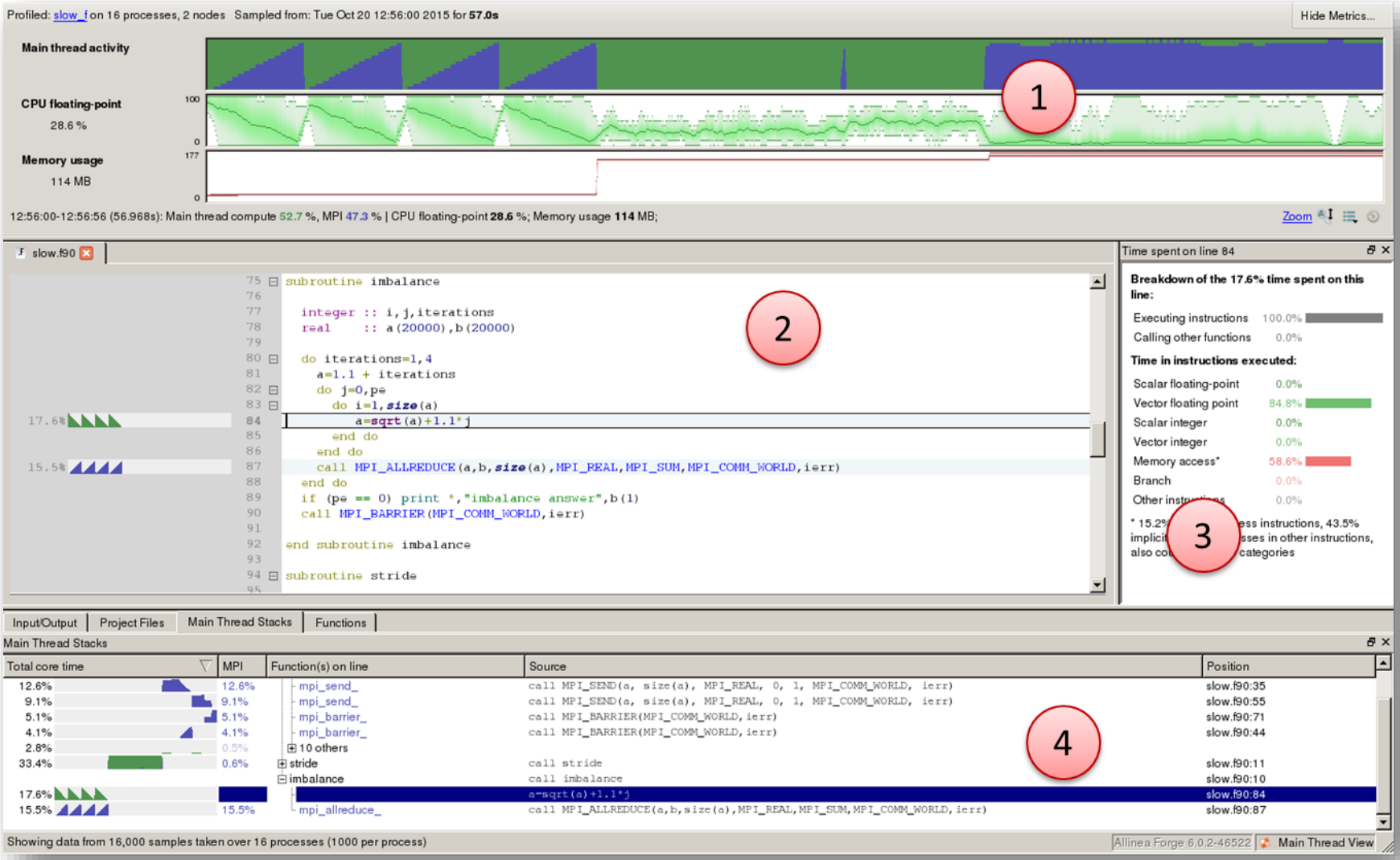
```
module load arm-forge
map --profile ./prog
```

  - Will not open any window but run the code and generate a “*.map” file

  - Open with map:

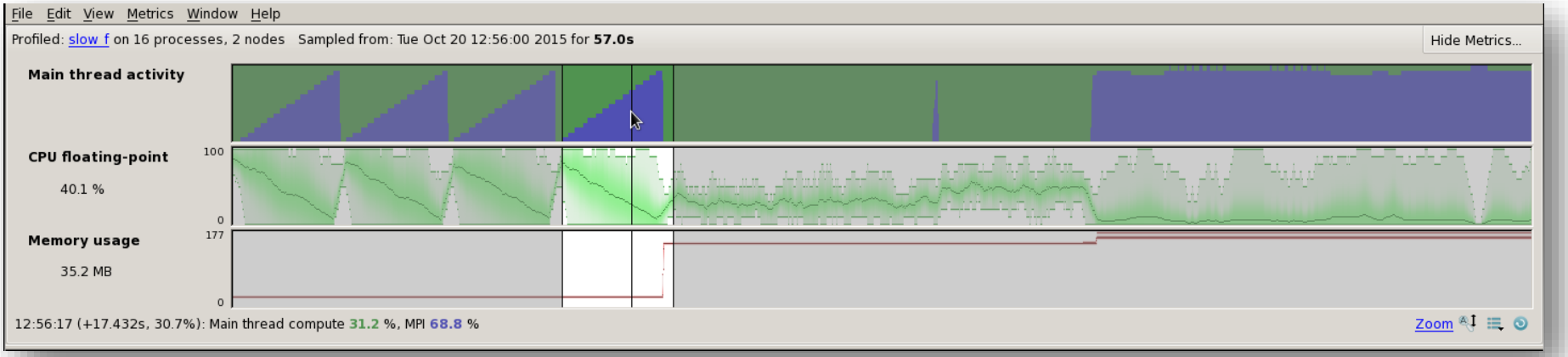
```
$ module load arm-forge
$ map <outputfile>.map
```

MAP: General presentation

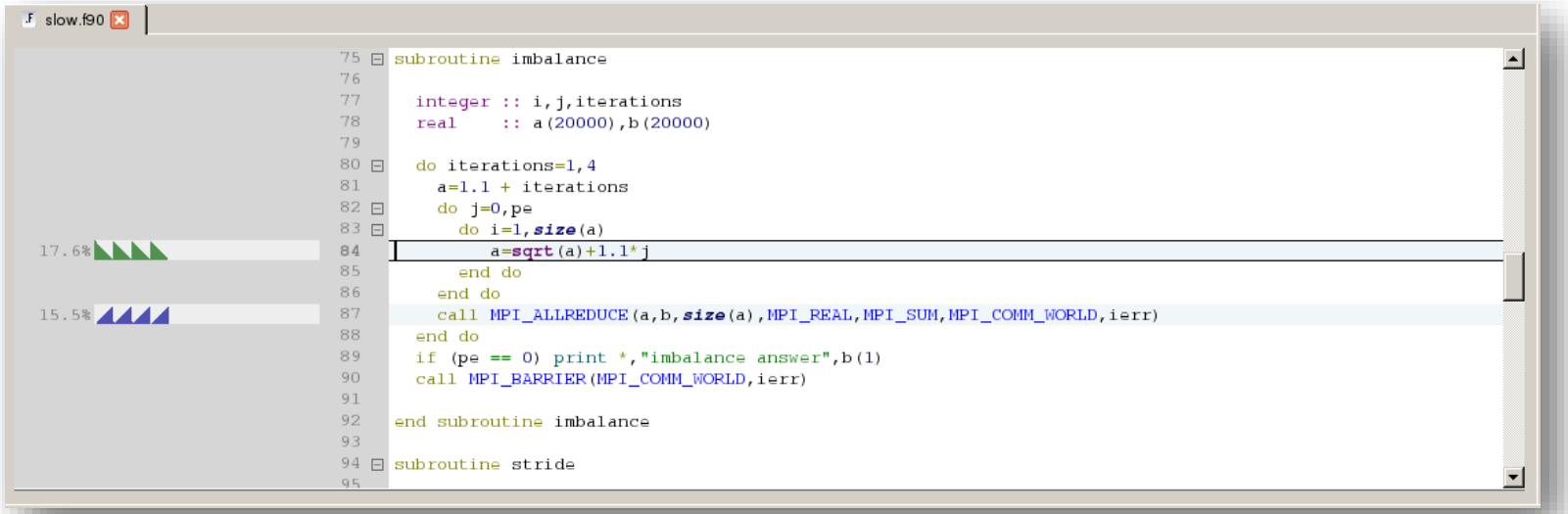


Metric view

- Displays different metrics over a timeline
- To change displayed metrics, go to « Menu > Metrics »
- Use the timeline to zoom on a specific region
- Activity color code:
  - Blue -> MPI
  - Dark green -> Main threaded computation
  - Light green -> OpenMP computation
  - Grey -> OpenMP overhead
  - Orange -> I/O

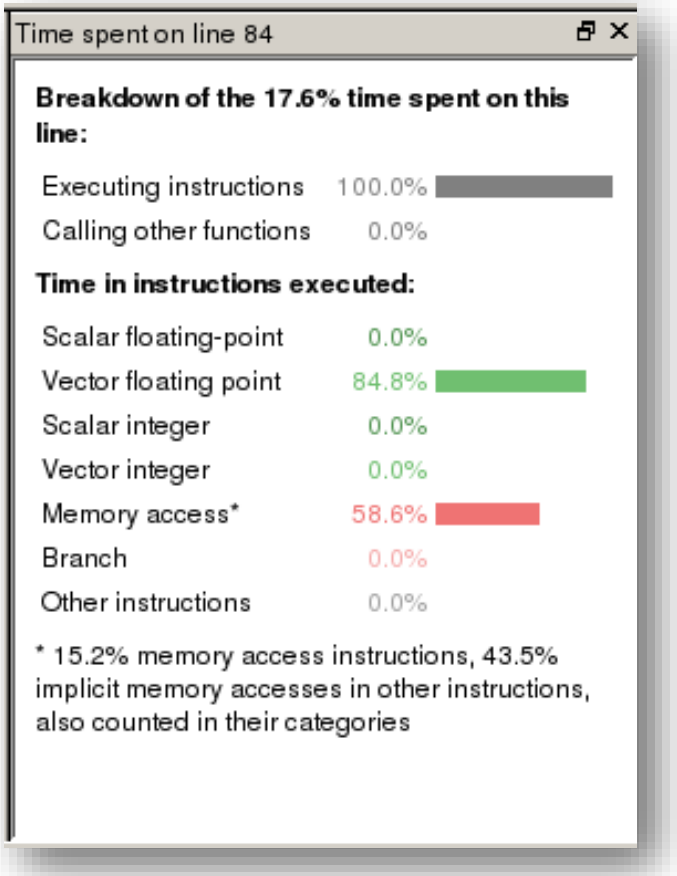


Source code view



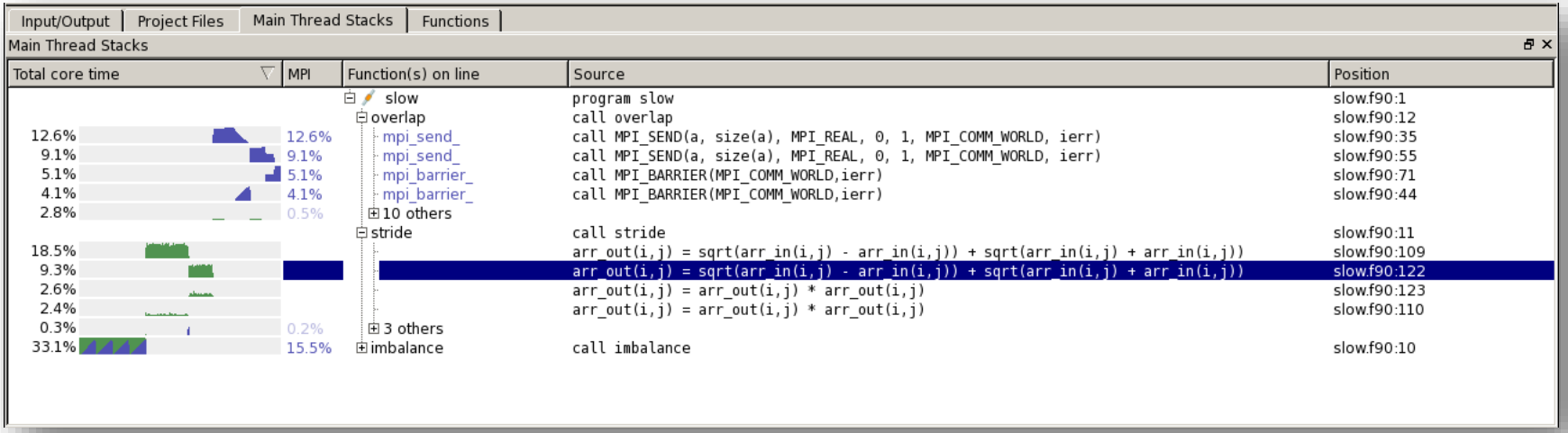
- Displays the source code
- The sparkline charts next to each line of source code shows the inclusive time spent on that line
- A zoom on the metric timeline will update the sparkline charts

Selected lines view



- Get detailed information on one or more selected lines of code
- First section gives an overview of how much time was spent executing instructions
- Second section details the CPU instruction metrics for the selected line.

Stack view



- A top-down view of the functions called in your program
- Clicking on any line of the Stacks view jumps the Source Code view to show that line of code

SCORE-P AND VAMPIR

Summary

1. Profiling: overview

2. Simple code profiling

3. **Score-P & Vampir**
   1. **Score-P simple profiling**
   2. **Score-P trace**
   3. **Vampir**

4. Profiling with Vtune

Score-P and Vampir

SCORE-P: SIMPLE PROFILING

Presentation

- Score-P: the Scalable Performance Measurement Infrastructure for Parallel Codes
- Provides:
  - A measurement infrastructure for profiling, event trace recording, and online analysis
  - The Opari2 instrumenter as a common infrastructure for a number of analysis tools like Periscope, Scalasca, Vampir, and Tau
- Supports:
  - The new Open Trace Format version 2 (OTF2) for tracing data
  - Several programming models: MPI, OpenMP, Pthreads, Cuda, OpenCL

Score-P: Usage

- 3 basic steps:
  - Instrumentation
    - The code needs to be instrumented, which means it has to be recompiled with Score-P
    - There are different instrumentation techniques, manual or automatic
  - Measurement collection
    - When the instrumented code runs, it will collect profiling data
    - It is possible to configure Score-P with a set of environment variables
  - Analysis
    - The resulting profiling data can be opened by several existing tools
    - For example Cube, Vampir, Tau

Score-P: Instrumentation

- The “scorep” command will instrument the code automatically. All that is needed is to recompile the code with it
  - Load scorep

```
$ module load scorep
```

  - Compile

```
$ scorep mpicc -g -O3 -openmp prog.c -o prog
$ scorep mpif90 -g -O3 -openmp prog.f90 -o prog
```

- In a Makefile, the compiler is usually defined by the CC variable. You may simply override it with

```
$ make CC="scorep mpicc"
$ make CF="scorep mpif90"
```

Score-P: Collection

- Once the code has been instrumented with Score-P, running it will generate profile data

```
module load scorep
ccc_mprun ./prog
```

- A directory is created when running the code. It will contain several files, depending on the set of parameters.

- To set the collection parameters, Score-P provides a set of environment variables

Score-P: Environment variables

- Here are some useful variables:
  - SCOREP_ENABLE_TRACING: Enable tracing, default is 0
  - SCOREP_ENABLE_PROFILING: Enable simple profiling, default is 1
  - SCOREP_TOTAL_MEMORY: Total memory in bytes for the measurement system
  - SCOREP_EXPERIMENT_DIRECTORY: Name of the experiment directory
  - SCOREP_FILTERING_FILE: A file name which contain the filter rules

- To have a full description of available variables:

```
$  scorep-info config-vars --full
```

- To have a list of currently set variables:

```
$  scorep-backend-info config-vars
```

Recording performance metrics: Resource Usage

- It is also possible to record resource usage counters
- Uses Unix system call getrusage to provide information about:
  - consumed resources
  - operating system events (user/system time)
  - received signals
  - number of page faults
  - …
- See all available counters

```
$ man getrusage
```

- To activate those counters, use the environment variable

```
export SCOREP_METRIC_RUSAGE=ru_maxrss,ru_utime,ru_nswap
```

Recording performance metrics: PAPI

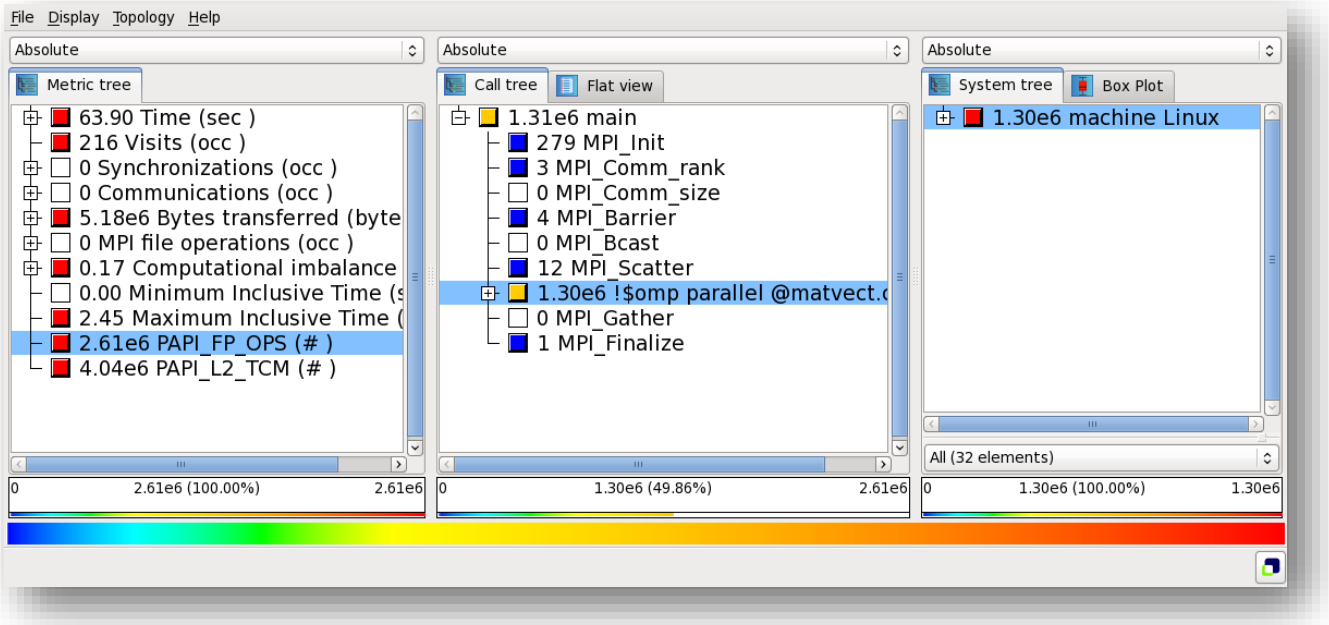
- It is possible to add hardware event counters with PAPI
- Types of counters:
  - Number of operations
  - Level 1, 2 or 3 cache misses, cache hits
  - Conditional branches
  - …
- To see all the available counters on the current configuration

```
$ papi_avail
```

- To activate those counters, use the environment variable

```
export SCOREP_METRIC_PAPI=PAPI_FP_OPS,PAPI_L2_TCM
```

Recording performance metrics: PAPI



- The selected PAPI counters appear in the left browser
- Warning: adding counters can make the collection dramatically longer

Score-P: Collection output

- Let us say we set the "SCOREP_EXPERIMENT_DIRECTORY" variable to "output"

- For a basic run with the default arguments, here is the output you should get:

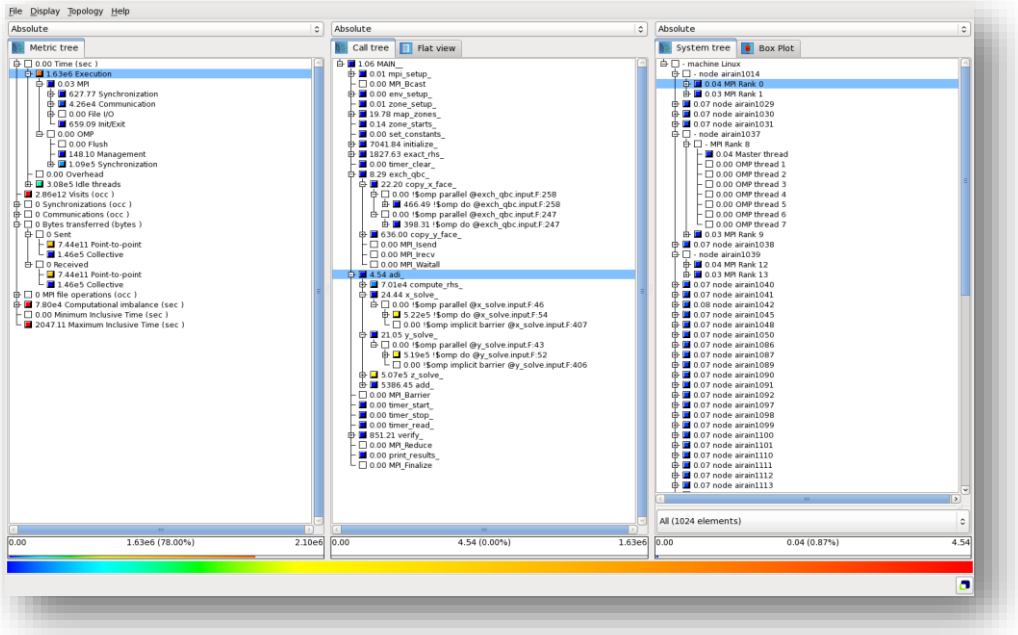
```
output/
|-- profile.cubex
`-- scorep.cfg
```

- scorep.cfg contains the parameters that were set during the run and profile.cubex is the profile data to be opened with the Cube GUI

Display the profile with Cube 4

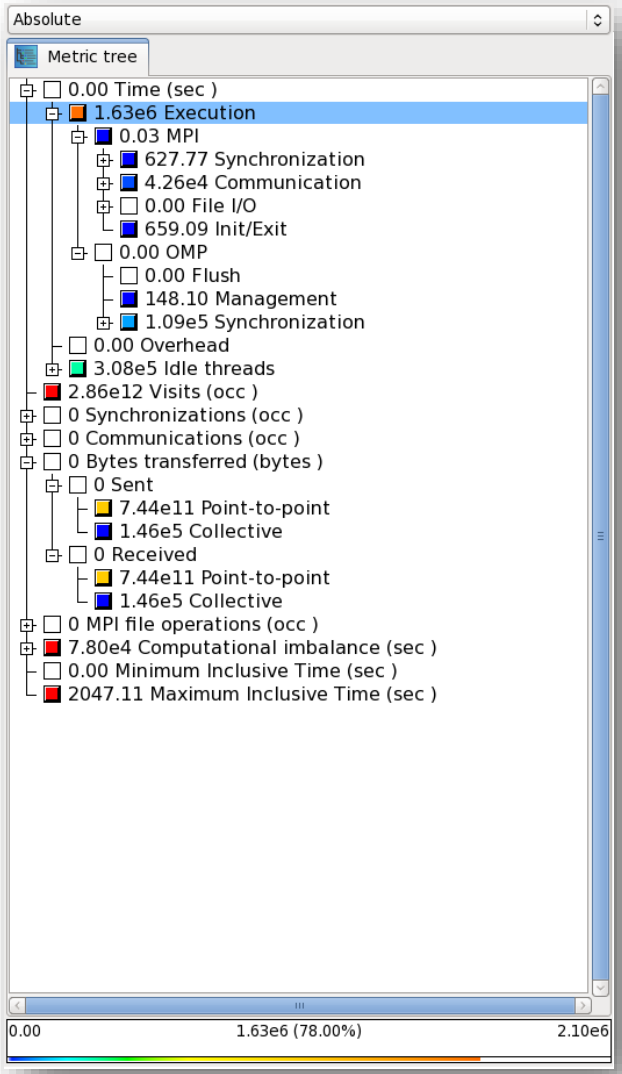
- Start graphic interface:

```
$ module load scorep
$ cube output/*.cubex
```



- 3 coupled browsers
- Colors enable the easy identification of regions of interest
- Numerical values enable precise, individual comparison

Display the profile with Cube 4



- The metric dimension
- Choose which part of the problem is measured
- To get documentation on one of the metrics:
  - right click on the metric
  - “online description”

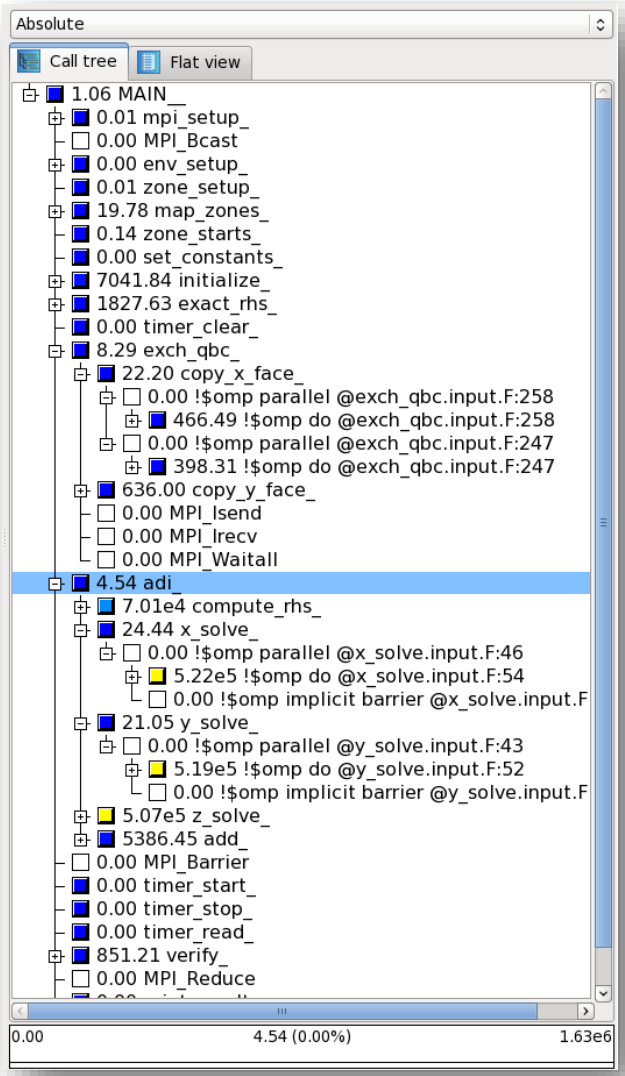
Display the profile with Cube 4

- The metrics can be organized in different ways

- That can be set by the cube function “cube_remap2” and with a specification file available in the share directory of the cube installation

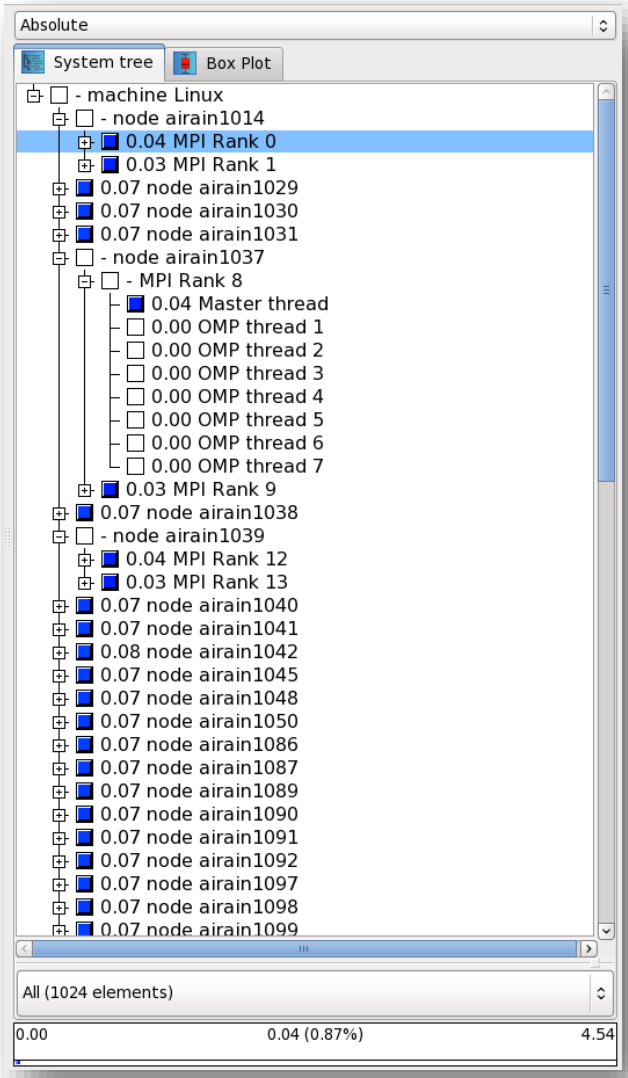
```
cube_remap2 -r $CUBE_ROOT/share/scorep.spec \
-o output/summary.cubex output/profile.cubex
```

Display the profile with Cube 4



- The program dimension
- Call tree
- Identify which part of the code is related to the selected metric

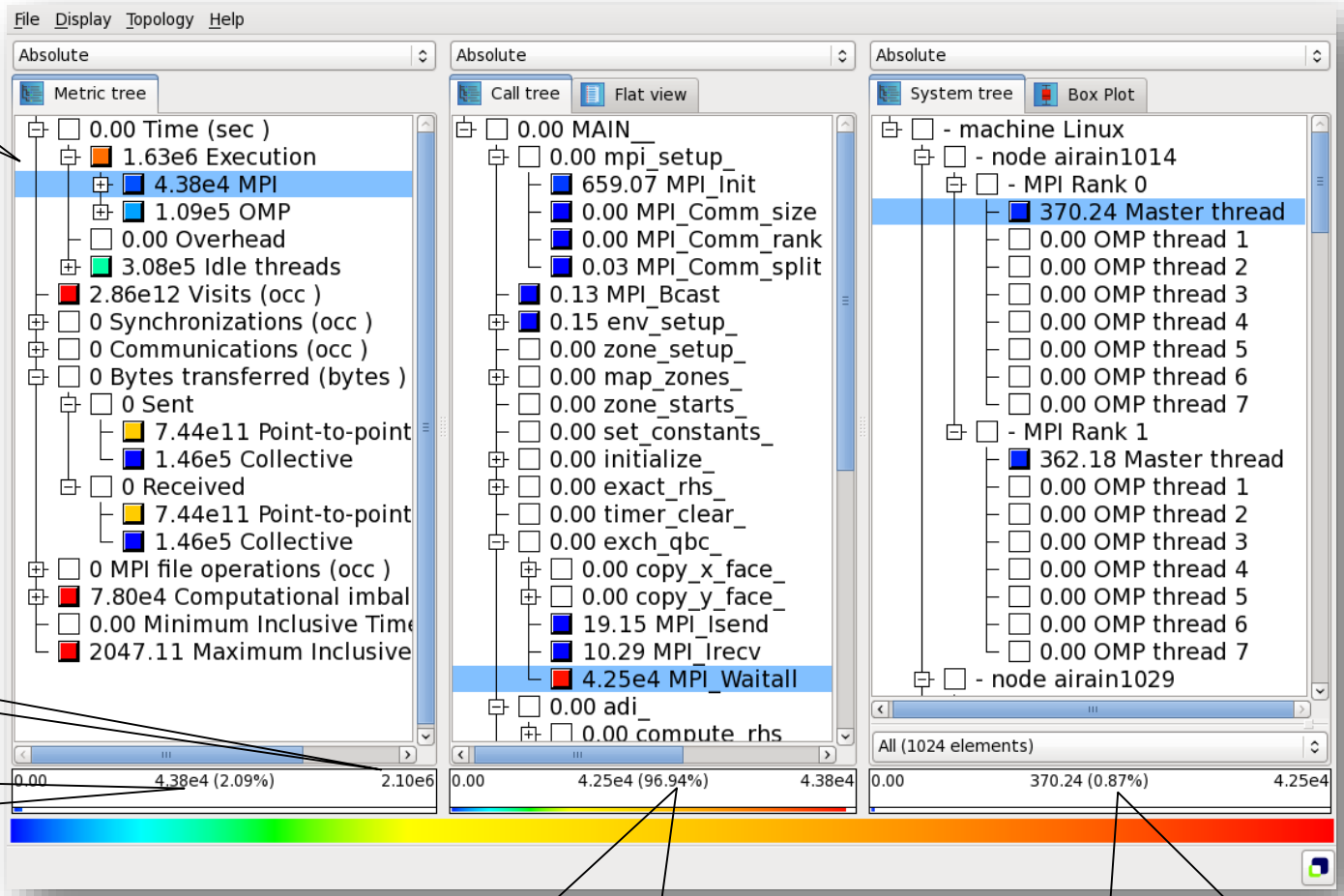
Display the profile with Cube 4



- The system dimension
- Shows how it is distributed across the system
  - Machines
  - Nodes
  - Processes (MPI)
  - Threads (OpenMP)

Example: Time spent in MPI calls

Select all MPI related events



Total value considered

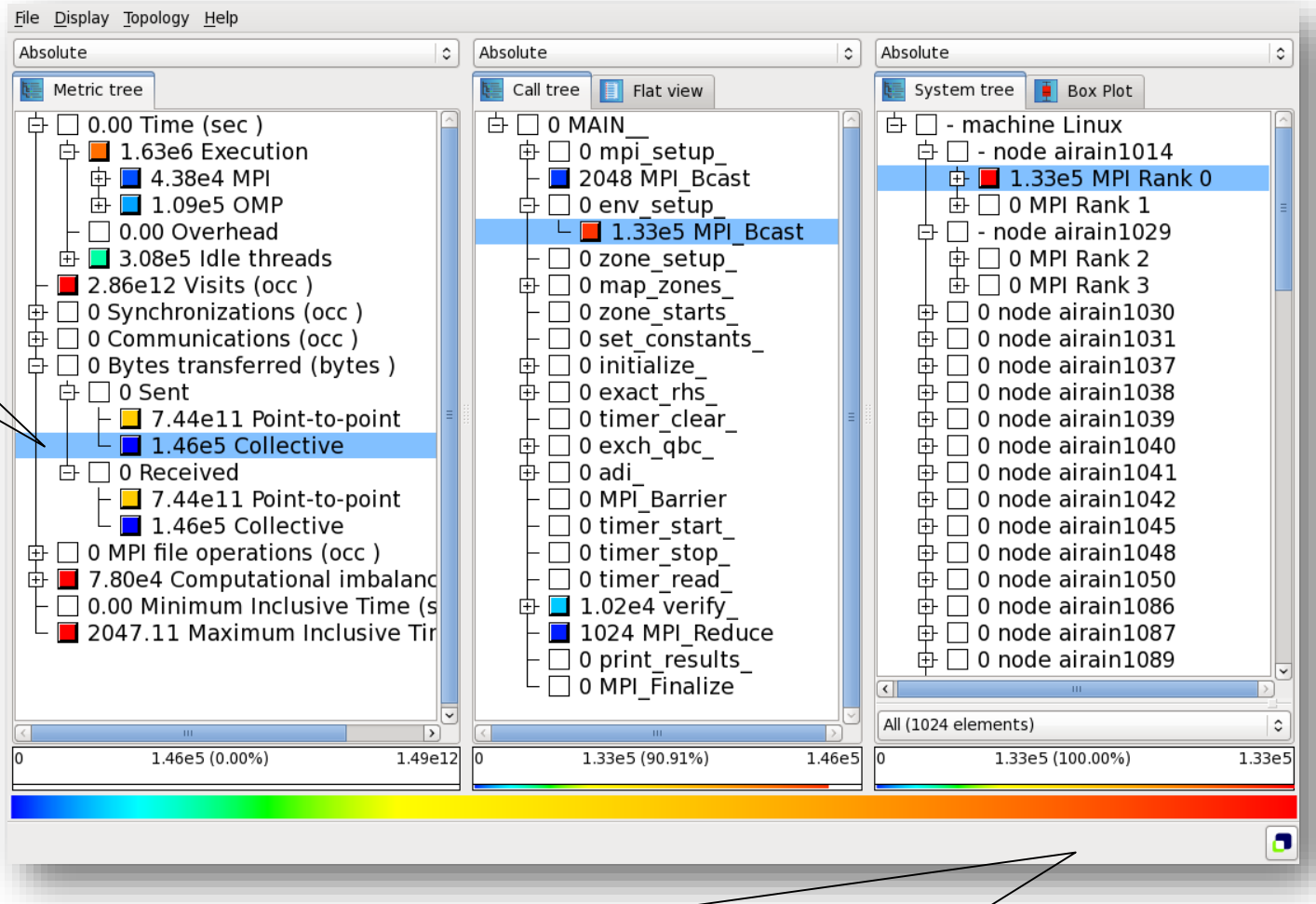
Percentage of whole execution time

Percentage of time spent in MPI_Waitall compared to whole MPI time

Percentage of MPI_Waitall time spent for rank 0

Example: Data transfer

Measure number of bytes sent through collective communications



We selected the MPI_Bcast calls: All bytes sent by Bcast are sent by process 0

Cube: Documentation

- http://www.scalasca.org/software/cube-4.x/documentation.html

Score-P and Vampir

SCORE-P: TRACE

Textual report

- To get some basic information, it is possible to print a textual report:

```
-bash-4.1 $ scorep-score [-c nb_metrics] -r
output/profile.cubex
```

```
Estimated aggregate size of event trace:                   68TB
Estimated requirements for largest trace buffer (max_buf): 543GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       543GB
(hint: When tracing set SCOREP_TOTAL_MEMORY=543GB 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 582,836,099,072 2,864,183,151,627 1564100.36   100.0           0.55  ALL
         USR 582,721,235,044 2,863,727,813,377  698484.77    44.7           0.24  USR
         OMP     109,060,352       433,913,856  847161.56    54.2        1952.37  OMP
         MPI       3,416,590         6,973,834   18317.02     1.2        2626.54  MPI
         COM       2,935,270        14,450,560     137.02     0.0           9.48  COM

         USR 190,897,220,696   937,960,767,488  182246.59    11.7           0.19  matvec_sub_
         USR 190,897,220,696   937,960,767,488  291905.05    18.7           0.31  binvcrhs_
         USR 190,897,220,696   937,960,767,488  212406.87    13.6           0.23  matmul_sub_
         USR   4,368,270,504    21,491,941,376    4011.69     0.3           0.19  exact_solution_
         USR   2,973,212,970    14,154,825,728    3170.98     0.2           0.22  binvrhs_
         USR   2,973,212,970    14,154,825,728    4725.91     0.3           0.33  lhsinit_
         USR       8,920,106        43,914,368       8.08     0.0           0.18  get_comm_index_
         OMP       6,554,112        16,449,536       6.51     0.0           0.40  !$omp parallel @exch_qbc.f:258
         OMP       6,554,112        16,449,536       6.38     0.0           0.39  !$omp parallel @exch_qbc.f:218
         OMP       6,554,112        16,449,536       6.50     0.0           0.40  !$omp parallel @exch_qbc.f:247
         OMP       6,554,112        16,449,536       6.38     0.0           0.39  !$omp parallel @exch_qbc.f:207
```

Textual report

- Description of different regions
  - **ALL**: Includes all functions of the application
  - **OMP**: This group contains all regions that represent an OpenMP construct
  - **MPI**: This group contains all MPI functions
  - **COM**: This group contains all functions, implemented by the user, that appear on a call path to an MPI function or an OpenMP construct
  - **USR**: This group contains all user functions that do not appear on a call path to an OpenMP construct or MPI function
- Most part of computation spent in region “USR”

Textual report

- Estimated size needed if we wanted to get a whole trace

```
Estimated aggregate size of event trace:                   68TB
Estimated requirements for largest trace buffer (max_buf): 543GB
```

- Using performance metrics will also dramatically increase the trace size
  - For example, with 3 metrics to record:

```
Estimated aggregate size of event trace:                   251TB
Estimated requirements for largest trace buffer (max_buf): 2005GB
```

- It is not possible to run the trace collection as it is

- We will have to filter the trace or the instrument the code manually

Filtering: Create a trace filter

- It is possible to filter the trace collection thanks to a text file

- Here is the basic syntax of a filter file:

```
SCOREP_REGION_NAMES_BEGIN
    EXCLUDE   foo
              bar
    INCLUDE   test1
              test2
SCOREP_REGION_NAMES_END
```

Filtering: Create a trace filter

- For example, if some very costly functions do not contain any interesting calls, you may exclude them:

```
$ cat filter
SCOREP_REGION_NAMES_BEGIN
    EXCLUDE binvcrhs_
            matmul_sub_
            matvec_sub_
            exact_solution_
            binvrhs_
            lhsinit_
            timer_
SCOREP_REGION_NAMES_END
```

Filtering: Create a trace filter

- Check the effect of that filter on previously taken summary with scorep-score:

```
-bash-4.1 $ scorep-score -f filter.txt -r output/profile.cubex
```

```
Estimated aggregate size of event trace:                   15GB
Estimated requirements for largest trace buffer (max_buf): 119MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       135MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=135MB 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 582,836,099,072 2,864,183,151,627 1564100.36   100.0           0.55  ALL
 -       USR 582,721,235,044 2,863,727,813,377  698484.77    44.7           0.24  USR
 -       OMP     109,060,352       433,913,856  847161.56    54.2        1952.37  OMP
 -       MPI       3,416,590         6,973,834   18317.02     1.2        2626.54  MPI
 -       COM       2,935,270        14,450,560     137.02     0.0           9.48  COM

 *       ALL     124,333,072       499,256,331  865633.26    55.3        1733.85  ALL-FLT
 +       FLT 582,712,314,184 2,863,683,895,296  698467.09    44.7           0.24  FLT
 -       OMP     109,060,352       433,913,856  847161.56    54.2        1952.37  OMP-FLT
 *       USR       8,920,886        43,918,081      17.67     0.0           0.40  USR-FLT
 -       MPI       3,416,590         6,973,834   18317.02     1.2        2626.54  MPI-FLT
 *       COM       2,935,270        14,450,560     137.02     0.0           9.48  COM-FLT

 +       USR 190,897,220,696   937,960,767,488  182246.59    11.7           0.19  matvec_sub_
 +       USR 190,897,220,696   937,960,767,488  291905.05    18.7           0.31  binvcrhs_
 +       USR 190,897,220,696   937,960,767,488  212406.87    13.6           0.23  matmul_sub_
 +       USR   4,368,270,504    21,491,941,376    4011.69     0.3           0.19  exact_solution_
 +       USR   2,973,212,970    14,154,825,728    3170.98     0.2           0.22  binvrhs_
 +       USR   2,973,212,970    14,154,825,728    4725.91     0.3           0.33  lhsinit_
 -       USR       8,920,106        43,914,368       8.08     0.0           0.18  get_comm_index_
 -       OMP       6,554,112        16,449,536       6.51     0.0           0.40  !$omp parallel @exch_qbc.f:258
 -       OMP       6,554,112        16,449,536       6.38     0.0           0.39  !$omp parallel @exch_qbc.f:218
 -       OMP       6,554,112        16,449,536       6.50     0.0           0.40  !$omp parallel @exch_qbc.f:247
 -       OMP       6,554,112        16,449,536       6.38     0.0           0.39  !$omp parallel @exch_qbc.f:207
```

Manual instrumentation

- It is possible to instrument the code manually to chose which part of the code to instrument
- Macros are available to do this easily
- In order to be taken into account, the manually instrumented code should be compiled with the option “--user”

```
$ scorep --user mpicc -g -O3 prog.c -o prog
$ scorep --user mpif90 -g -O3 prog.f90 -o prog
```

Manually define interesting regions

- #define SCOREP_USER_REGION_DEFINE (handle)
  - defines a user region handle in a local context

- #define SCOREP_USER_REGION_BEGIN (handle, name, type)
  - marks the beginning of the user defined region
  - handle: unique handle defined previously by SCOREP_USER_REGION_DEFINE
  - name: a string containing the name of the new region. The name should be unique
  - type: specifies the type of the region. Possible values are
    - SCOREP_USER_REGION_TYPE_COMMON
    - SCOREP_USER_REGION_TYPE_FUNCTION
    - SCOREP_USER_REGION_TYPE_LOOP
    - SCOREP_USER_REGION_TYPE_DYNAMIC
    - SCOREP_USER_REGION_TYPE_PHASE

- #define SCOREP_USER_REGION_END (handle)
  - marks the end of the user defined region

Manually define interesting regions

Fortran

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

subroutine test

  SCOREP_USER_REGION_DEFINE(
  my_handle)

   ! more declarations

  SCOREP_USER_REGION_BEGIN(my_handle
  ,"region1",
  SCOREP_USER_REGION_TYPE_COMMON)

   ! do something

  SCOREP_USER_REGION_END(my_handle)

end subroutine foo
```

C/C++

```
#include <scorep/SCOREP_User.h>

void test()
{
  SCOREP_USER_REGION_DEFINE(
  my_handle)

      // more declarations

  SCOREP_USER_REGION_BEGIN(my_handle
  ,"region1",
  SCOREP_USER_REGION_TYPE_COMMON )

      // do something

  SCOREP_USER_REGION_END(my_handle)

}
```

Switch recording off

- If there are large sections of the code that do not need to be analyzed, it is possible to switch recording off
  - #define SCOREP_RECORDING_OFF ( )
  - #define SCOREP_RECORDING_ON ( )
- It will reduce the trace size

Switch recording off

Fortran

```
subroutine test

  SCOREP_RECORDING_OFF()

  ! do something

  SCOREP_RECORDING_ON()

end subroutine test
```

C/C++

```
void test()
{
  SCOREP_RECORDING_OFF()

  // do something

  SCOREP_RECORDING_ON()

}
```

Filtering: Manually instrumented regions

- It is possible to filter the trace acquisition thanks to manually instrumented regions
- Just create a text file with the names of the regions to consider when analyzing the code

```
-bash-4.1 $ cat ./region_select
region1
region2
```

- The filter is activated through the environment variable SCOREP_SELECTIVE_CONFIG_FILE

```
export SCOREP_SELECTIVE_CONFIG_FILE=./region_select
```

Filter MPI events

- For MPI events, it is possible to easily select the groups of events to trace with SCOREP_MPI_ENABLE_GROUPS

- For instance, to collect information on point-to-point communications only, just export the following variable:

```
export SCOREP_MPI_ENABLE_GROUPS="P2P"
```

Filter MPI events

Token	Module
ALL	Activate all available modules
DEFAULT	CG, COLL, ENV, IO, P2P, RMA, TOPO, XNONBLOCK
CG	Communicators and groups (MPI_Comm_rank , MPI_Group_size, …)
COLL	Collective communication (MPI_Gather, MPI_Barrier, …)
ENV	Environmental management (MPI_Init, MPI_Finalize, …)
ERR	Error handlers (MPI_Errhandler_create, …)
EXT	External interfaces (MPI_Abort, MPI_Get_processor_name, MPI_Wtime, …)
IO	I/O (MPI_File_close, MPI_File_read, …)
MISC	Miscellaneous
P2P	Point-to-point communication (MPI_Send, MPI_Recv, MPI_Wait …)
RMA	One-sided communication
SPAWN	Process management interface (aka Spawn)
TOPO	Topology communicators
TYPE	MPI Datatypes (MPI_Type_struct, …)
XNONBLOCK	Extended non-blocking communication events
XREQTEST	Test events for tests of uncompleted requests

Generating a trace

- Once the estimated trace size is low enough to be considered, you can run a trace collection with:

```
export SCOREP_FILTERING_FILE=filter.txt
export SCOREP_ENABLE_TRACING=1
ccc_mprun ./prog
```

- This time, an OTF2 trace file will be created in the output directory
- This format may be opened with Vampir for instance

Score-P: Documentation

- https://silc.zih.tu-dresden.de/scorep-current/pdf/scorep.pdf
- https://silc.zih.tu-dresden.de/scorep-current/html/index.html

Score-P and Vampir

VISUALIZING DATA WITH VAMPIR

Presentation

- Easy to use performance analysis framework for parallel programs
  - Graphical data representation enables detailed understanding of dynamic processes on massively parallel systems
  - In-depth event based analysis of parallel run-time behavior and interprocess communication
  - Identification of performance problems and bottlenecks

Presentation

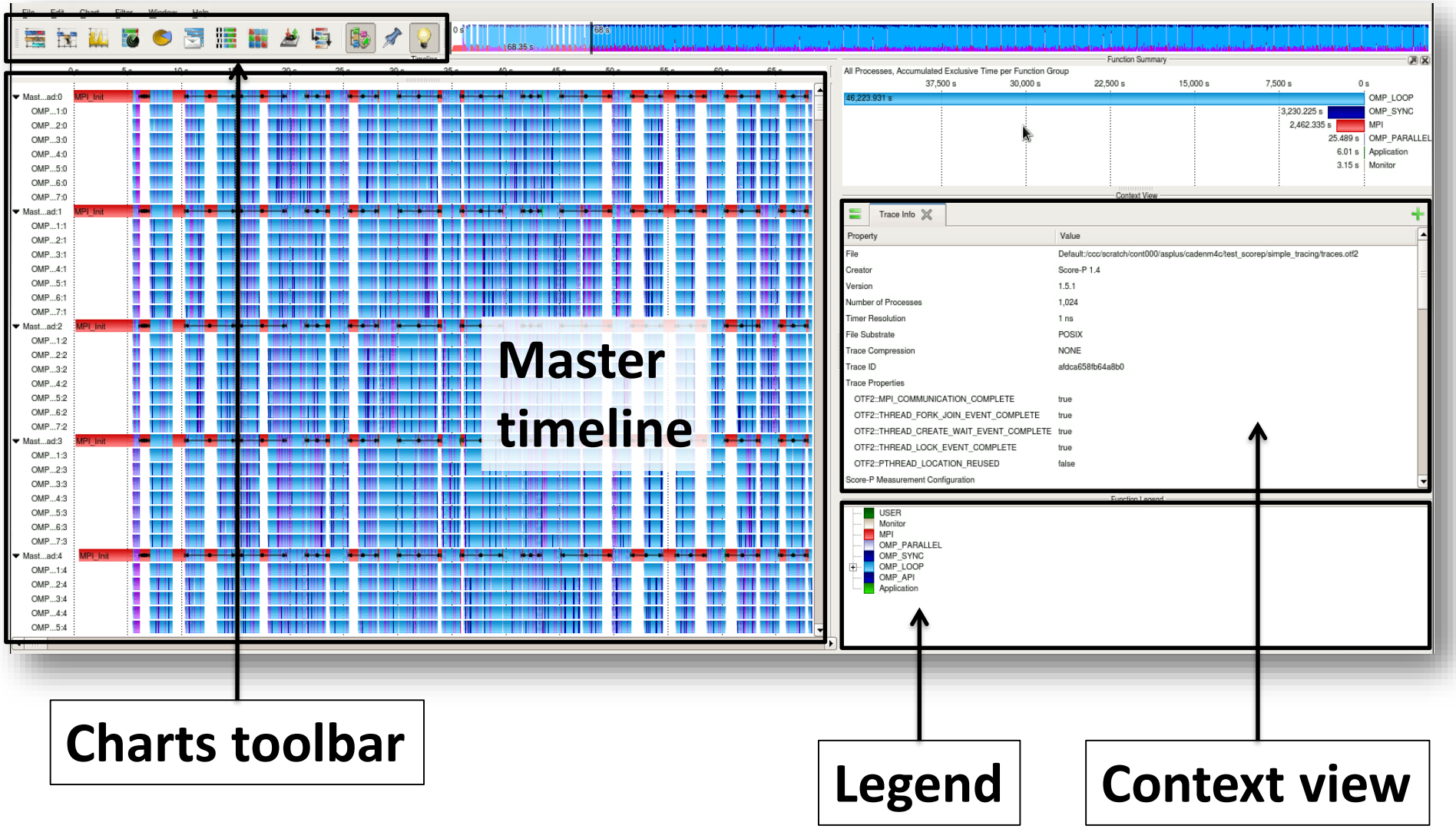
- Vampir focuses on extending the **graphical presentation** of performance data
  - Powerful zooming and scrolling in all displays
  - Adaptive statistics for user selected time ranges
  - Filtering of processes, functions, messages, collective operations
  - Hierarchical grouping of threads, processes, and nodes
  - Support of source code locations
  - Integrated snapshot and printing for publishing
  - Customizable displays

Vampir usage: visualize the trace

- OTF traces generated by Score-P may be opened by Vampir
- You must connect with X11 forwarding to launch the GUI (`ssh login@host -Y`) but it's advised to use HPCDRIVE
- To visualize the trace, launch vampir on the `OTF` files:

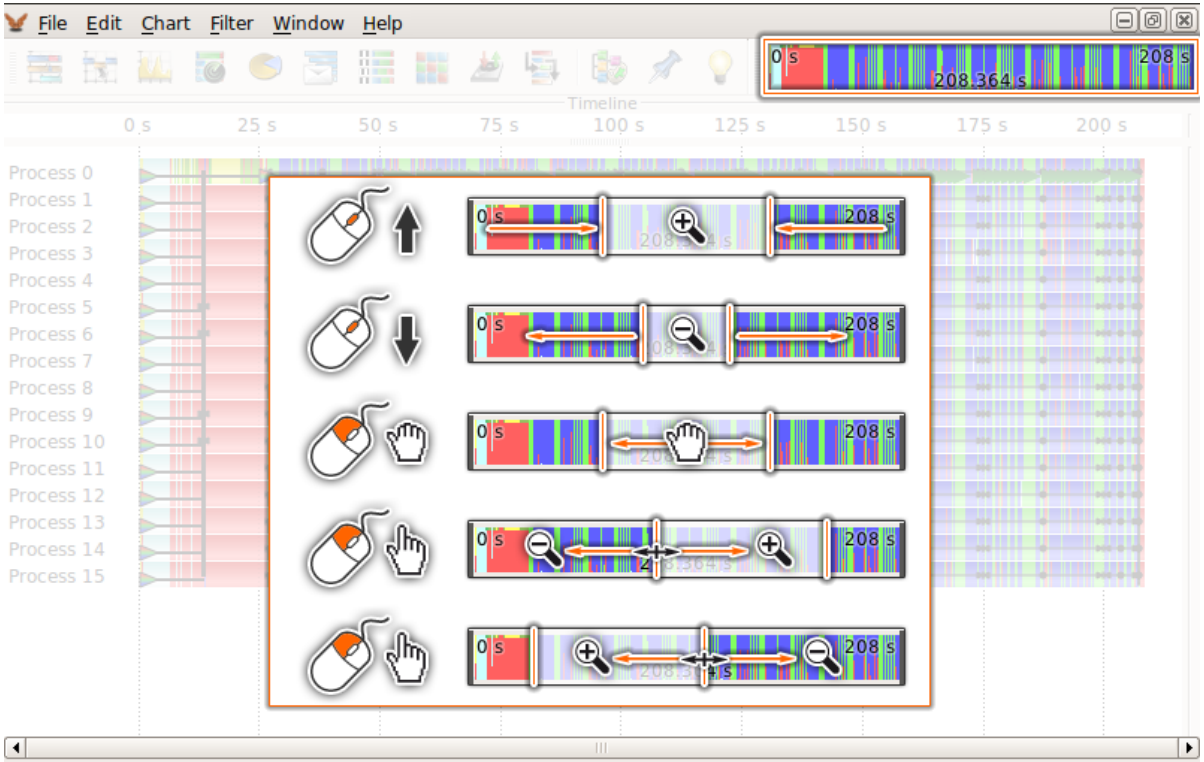
```
$ module load vampir
$ vampir <result_dir>/prog.otf2
```

Vampir GUI


Master timeline
Charts toolbar
Legend
Context view
Function Summary
All Processes, Accumulated Exclusive Time per Function Group
OMP_LOOP
OMP_SYNC
MPI
OMP_PARALLEL
Application
Monitor
Context View
Trace Info
Property
Value
File
Creator
Version
Number of Processes
Timer Resolution
File Substrate
Trace Compression
Trace ID
Trace Properties
Score-P Measurement Configuration
USER
Monitor
MPI
OMP_PARALLEL
OMP_SYNC
OMP_LOOP
OMP_API
Application

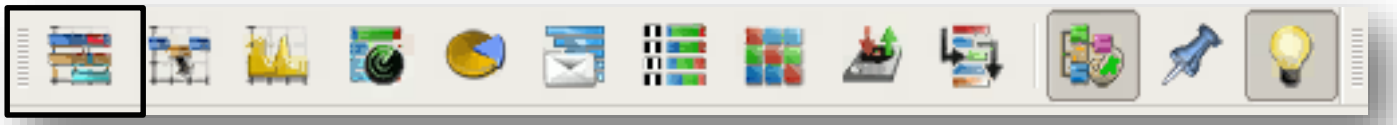
Vampir GUI

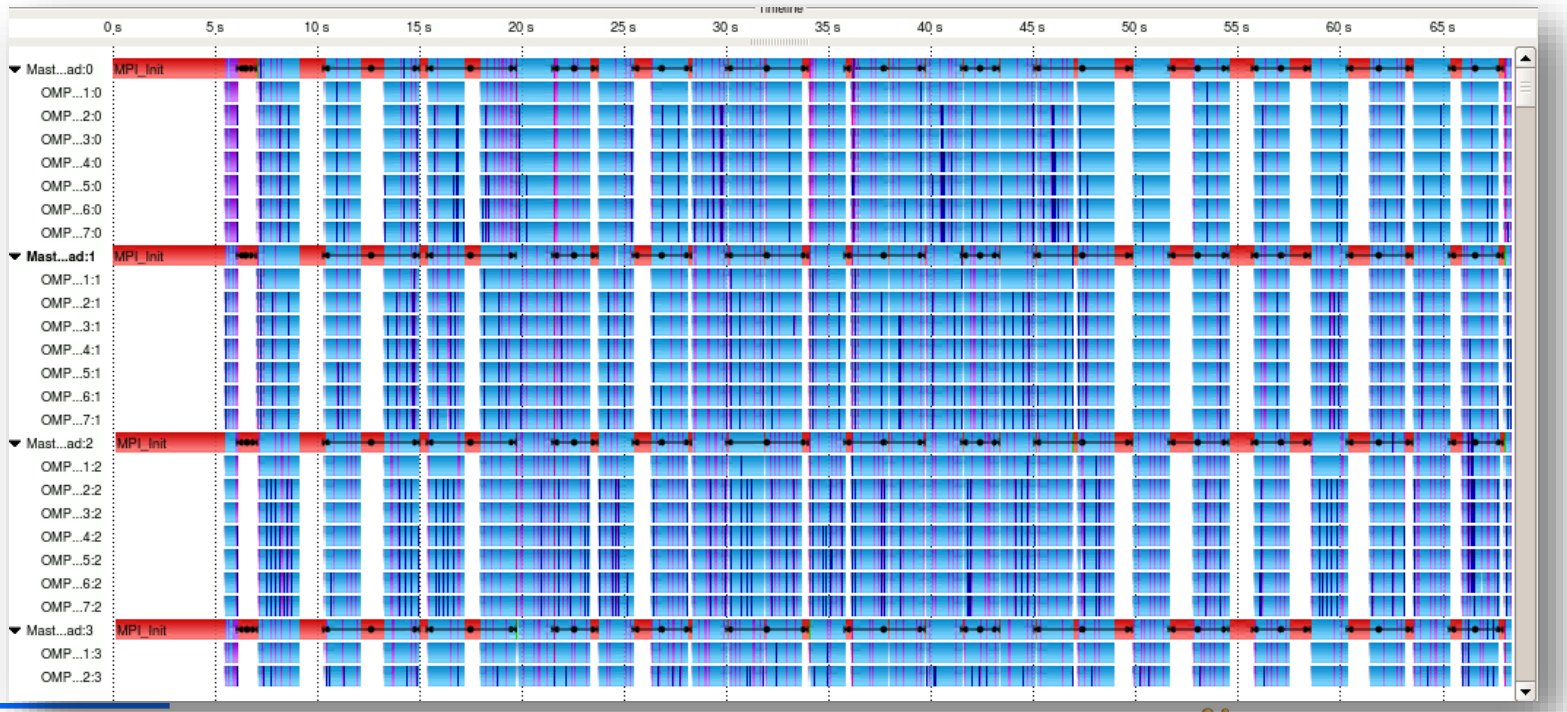
- Zoom toolbar
  - Used to zoom and navigate easily in the timeline
  - Only the selected part of the timeline will be displayed in the next charts



Vampir GUI

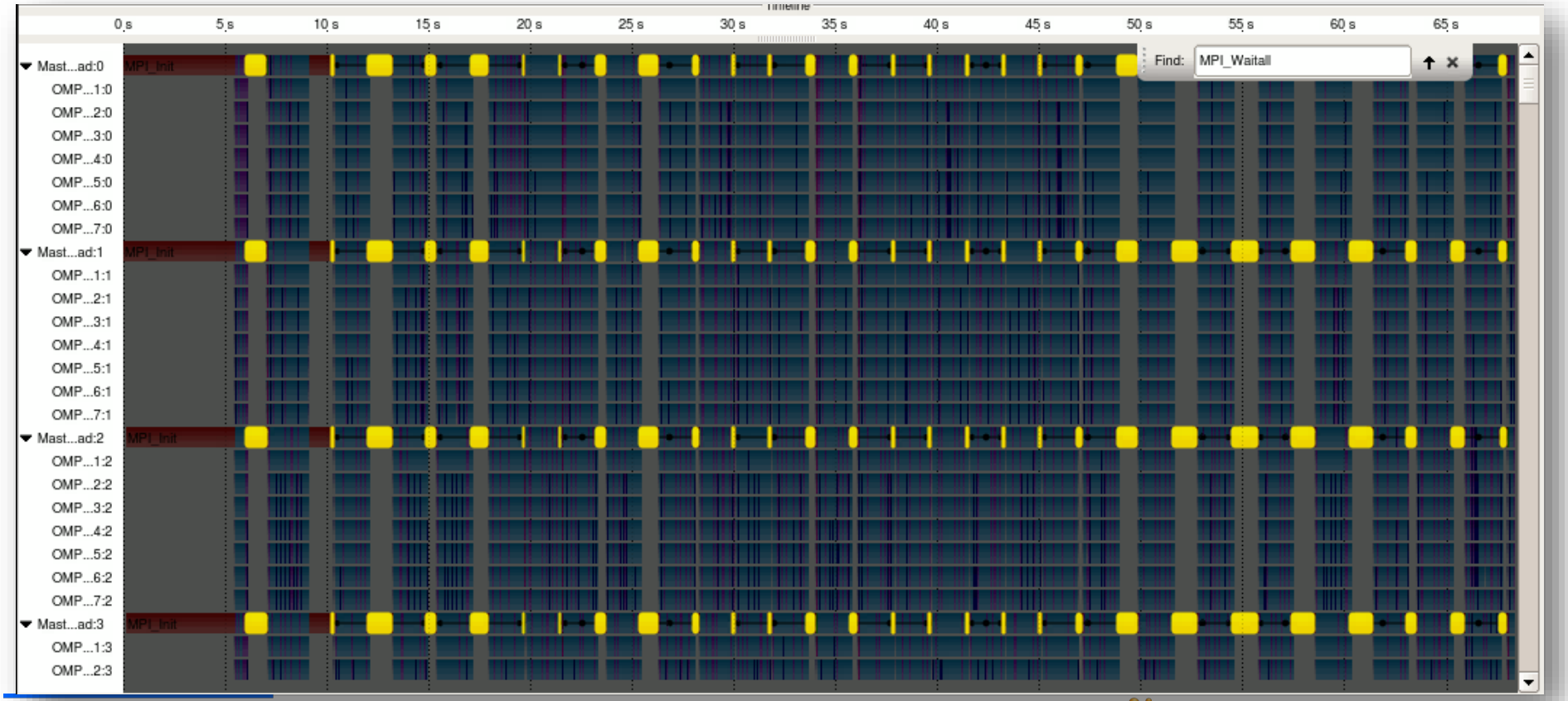
- Master Timeline
  - Each row represents one MPI or OpenMP process





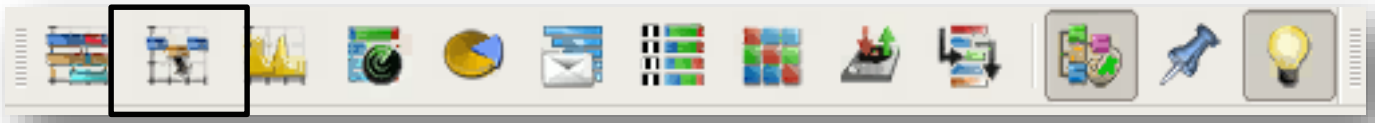
Vampir GUI

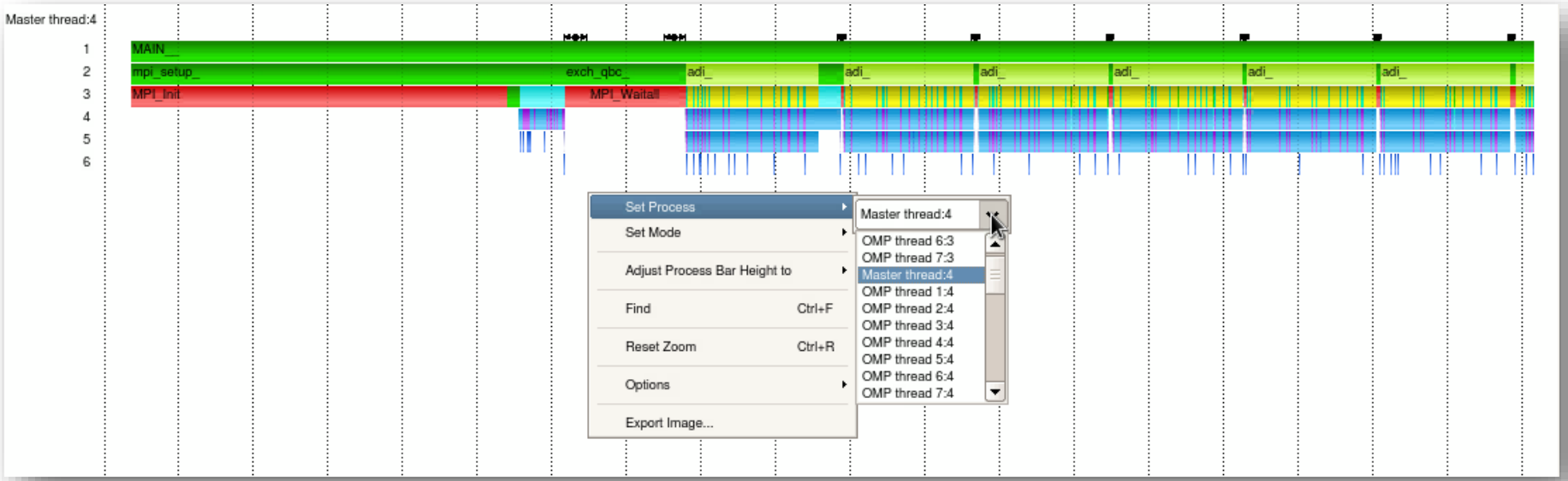
- Master Timeline: Search function
  - It is possible to highlight a specific type of function in the timeline
  - Activate the search with “CTRL+F” or with “right-click > Find…”



Vampir GUI

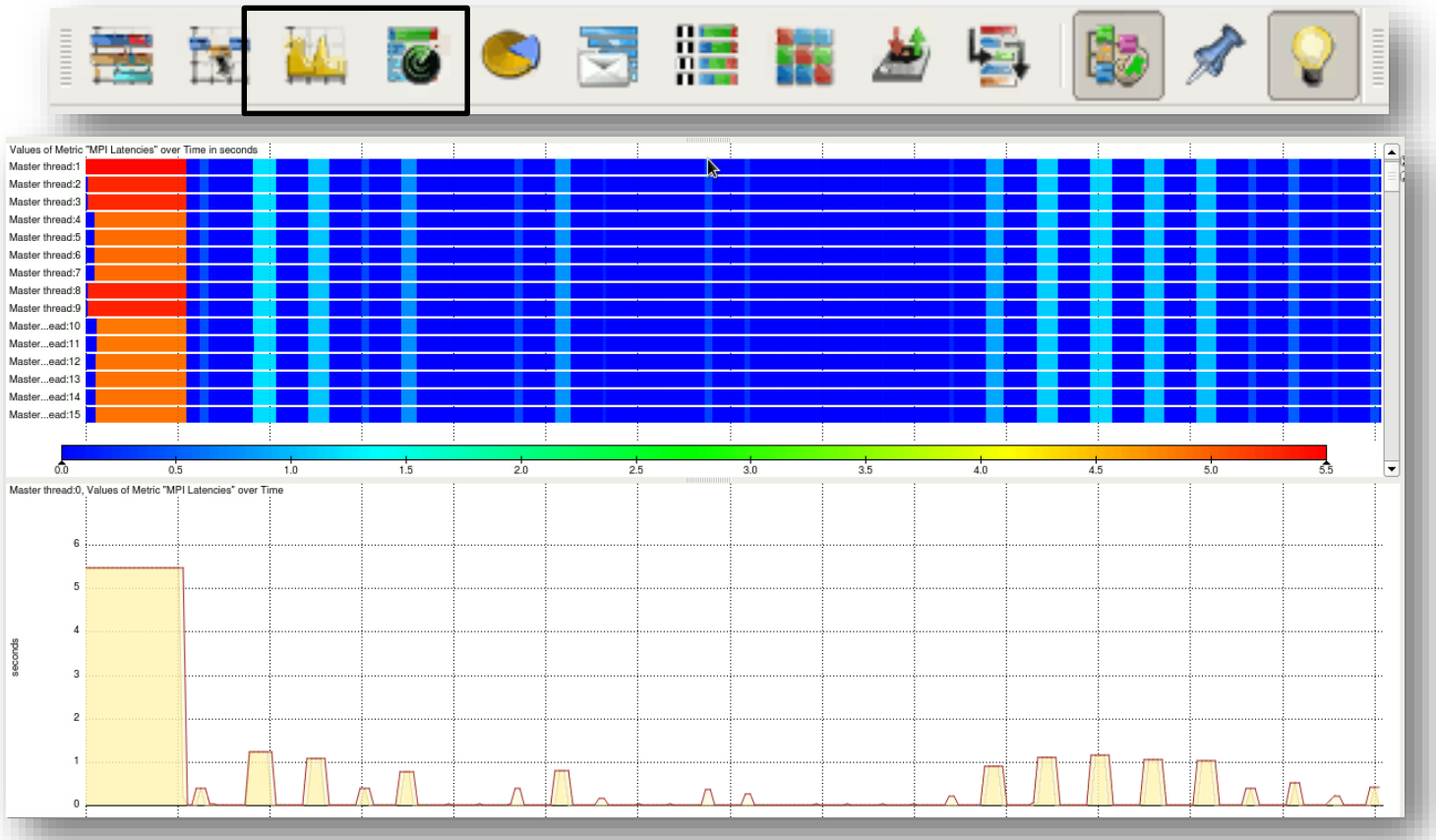
- Process Timeline
  - Represents the different levels of function calls for one selected process





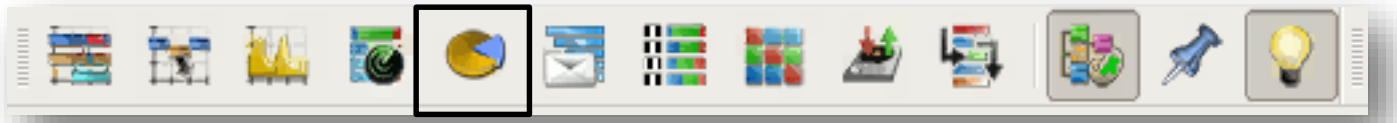
Vampir GUI

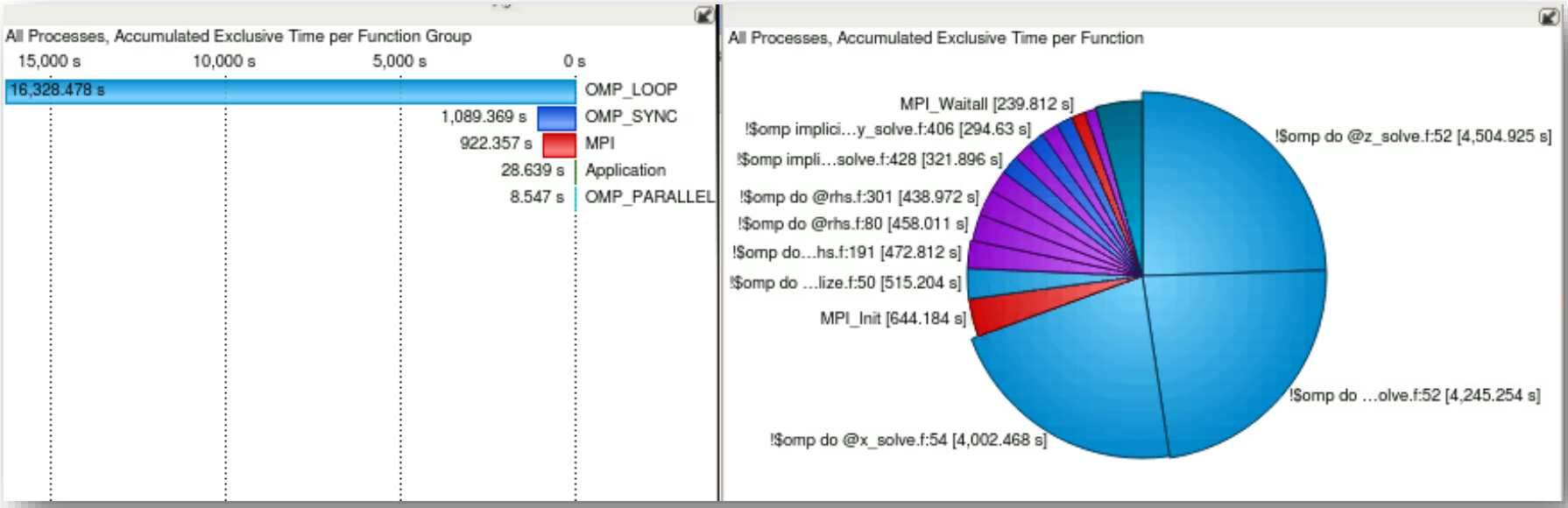
- Counter data timeline and performance radar
  - Shows the selected counter values



Vampir GUI

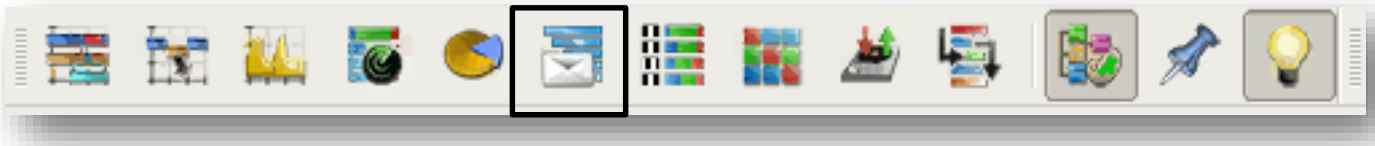
- Function Summary
  - Gives an overview of the accumulated time consumption across all function groups and functions

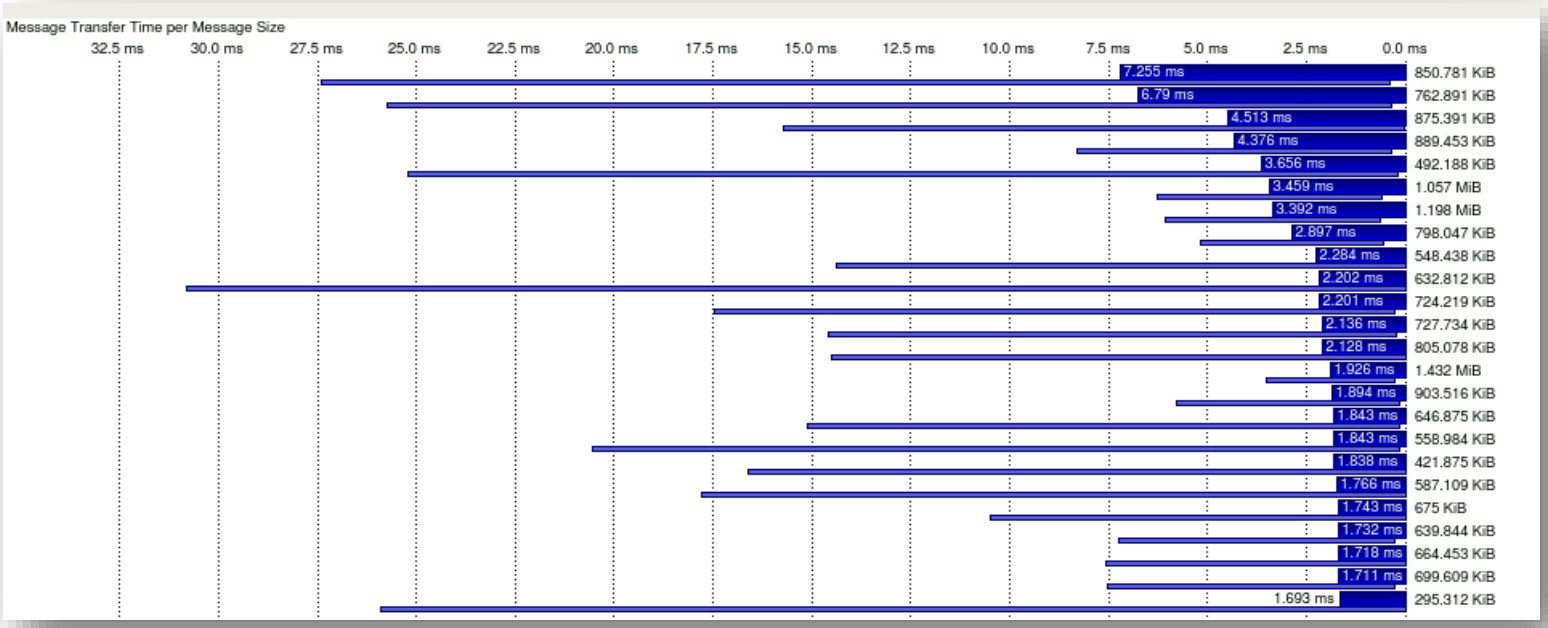




Vampir GUI

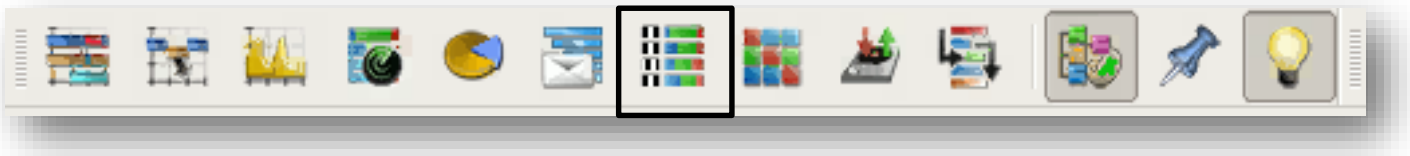
- The Message Summary
  - Shows an overview of all messages grouped by certain characteristics
  - The metric used can be switched between Aggregated Message Volume, Message Size, Number of Messages, and Message Transfer Rate

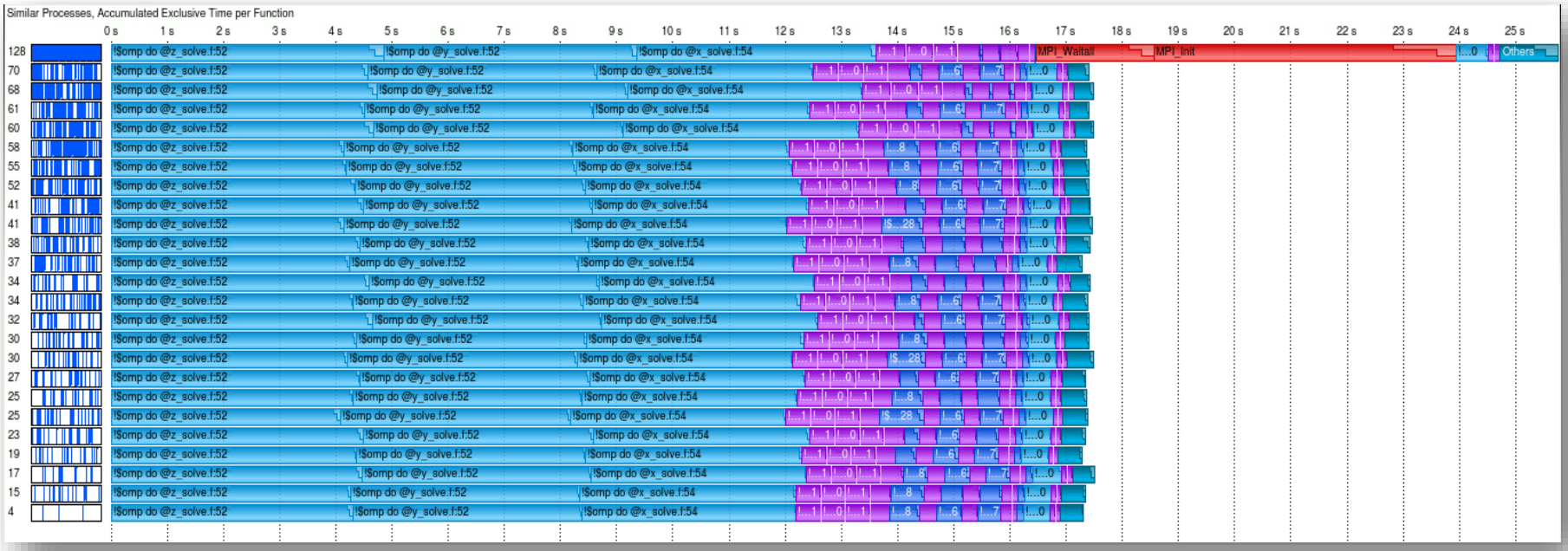




Vampir GUI

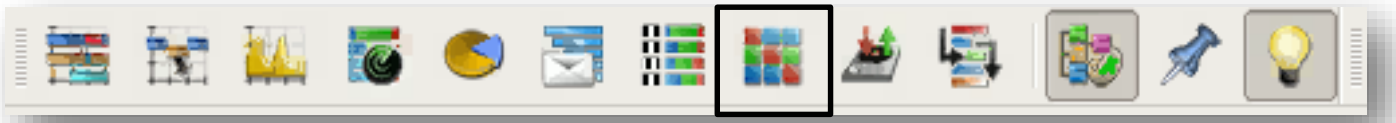
- The Process Summary
  - Shows the information for every process independently

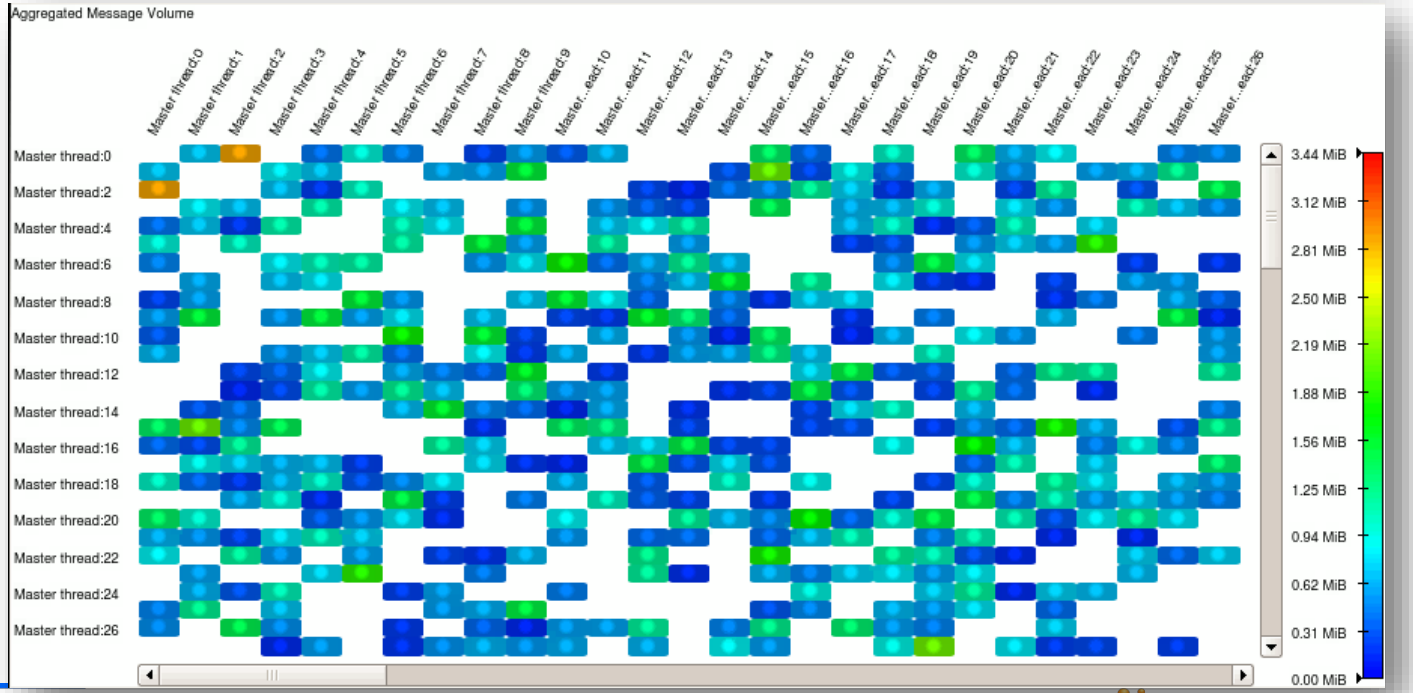




Vampir GUI

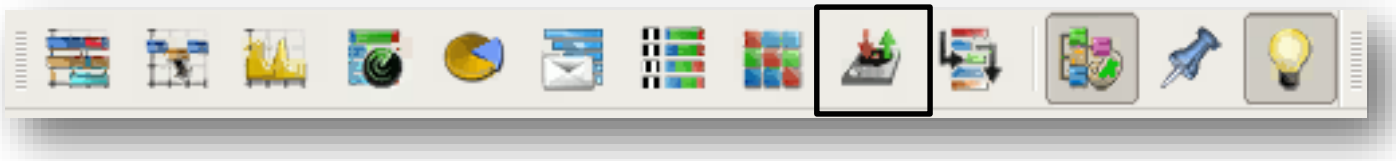
- The Communication Matrix
  - Shows information about messages sent between processes
  - Also possible to change the metric

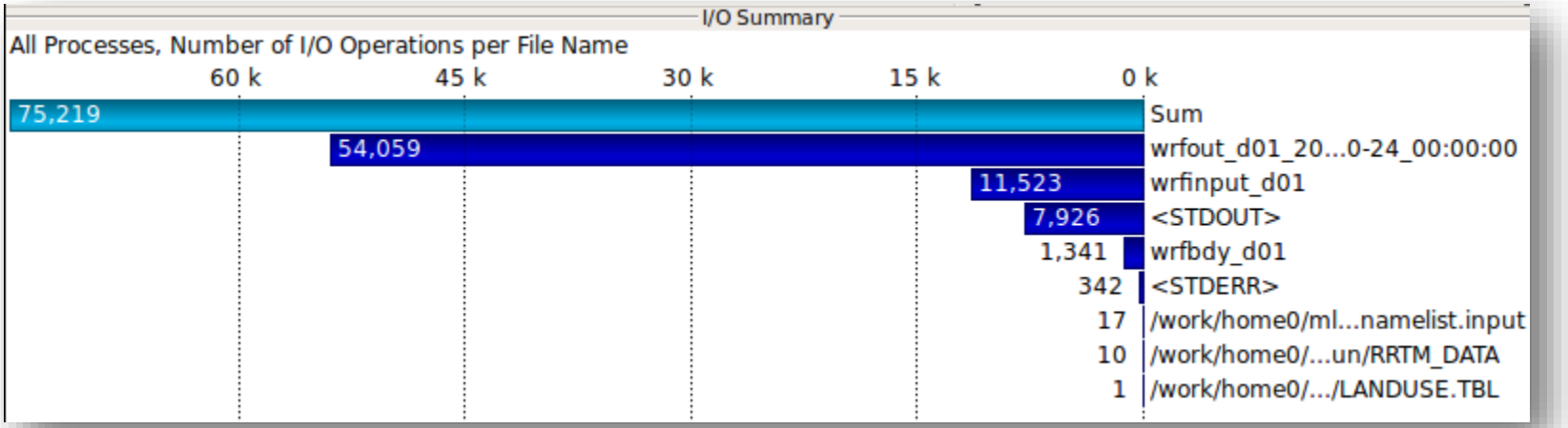




Vampir GUI

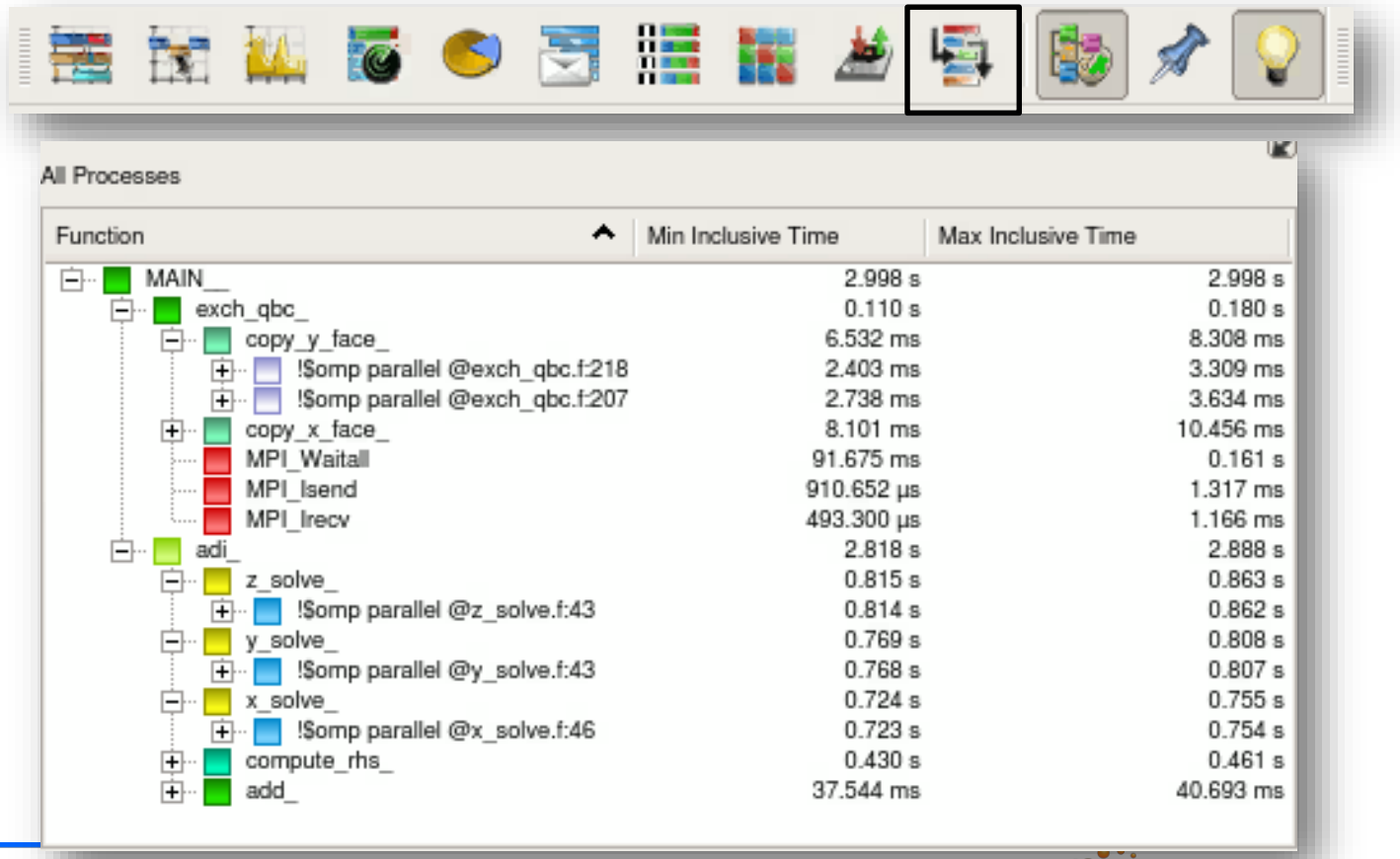
- The I/O Summary
  - Gives an overview of the input-/output operations recorded in the trace file





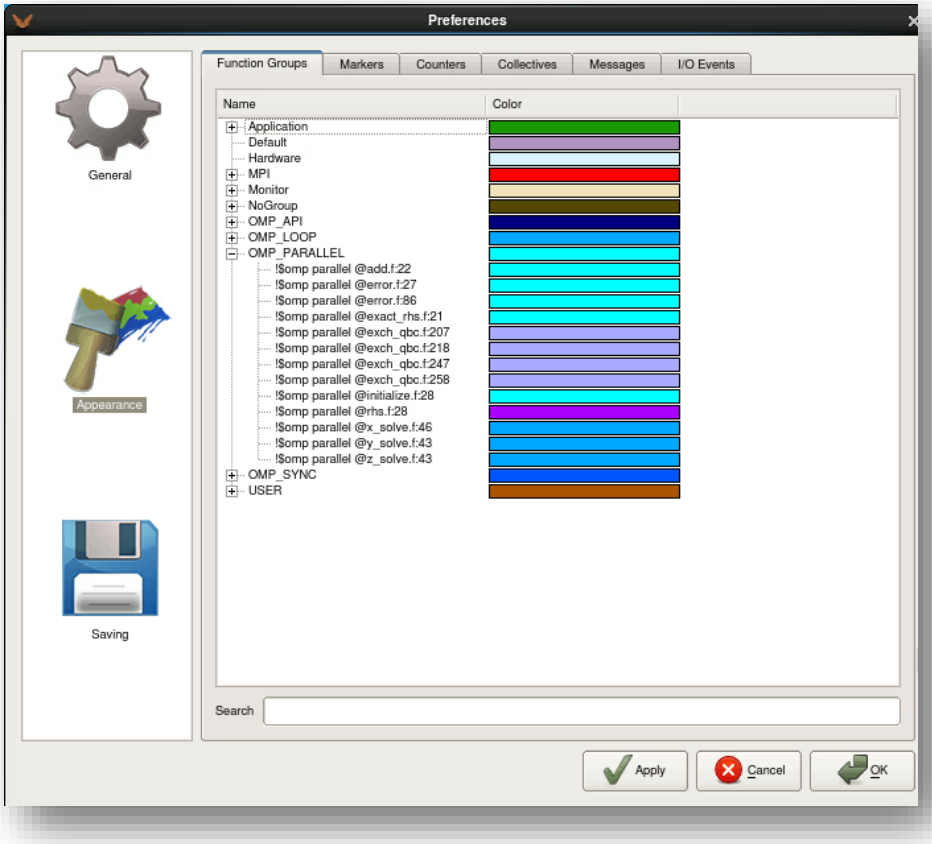
Vampir GUI

- The Call Tree
  - Illustrates the invocation hierarchy of all monitored functions in a tree representation



Customization: Color

- It is possible to adapt the display of different components in “File>Preferences”
- You can change the color of a specific MPI call or function group to make it more noticeable
- You can also adapt the display of markers, counters, collectives, messages, and I/O events



Customization: Filter

- It is possible to filter the information to display with the “filter” menu
- Filters are available to act on
  - Processes
    - process group, communicators, process hierarchy, representative process
  - Messages
    - communicators, tags
  - Functions
    - name, duration, number of invocations
  - Collective operations
    - communicators, collective operations
  - I/O events
    - I/O groups, file names, operation types

Vampir usage: visualize large traces

- Vampir often requires a large amount of resources
- Do not run it on login nodes

- For big traces, the GUI can become very slow and impossible to use
- 2 solutions:
  - HPCDrive
  - VampirServer

Vampir usage: visualize large traces

- Vampirserver is a parallel program which uses CPU computing to accelerate Vampir visualization
- First you have to submit a job for the server:

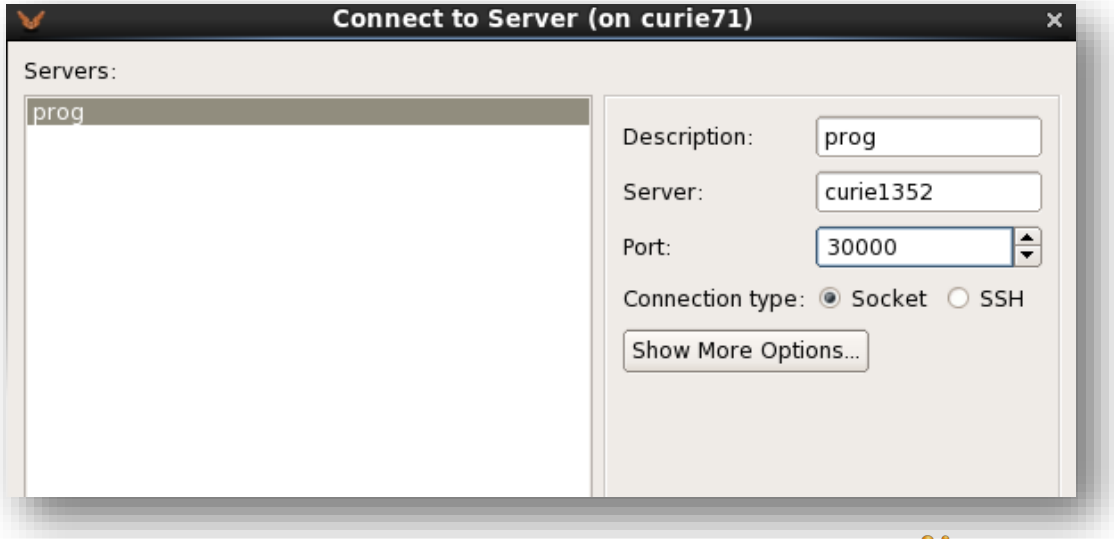
```
#!/bin/bash
#MSUB -r vampirserver                # Request name
#MSUB -n 32                          # Number of tasks to use
#MSUB -o vampirserver_%I.o           # Standard output.
#MSUB -e vampirserver_%I.e           # Error output.
#MSUB -q broadwell                   # Queue

module load vampir
vampirserver start -n $((BRIDGE_MSUB_NPROC-1))

sleep 7200
```

Vampir usage: visualize large traces

- Then you can retrieve its address and port:

```
$ ccc_mpp
USER        ACCOUNT    BATCHID  NCPU   QUEUE      PRIORITY   STATE  RLIM   RUN/START     SUSP   OLD    NAME             NODES
toto        genXXX     234481   32     large      210332     RUN    30.0m  1.3m          -      1.3m   vampirserver     curie1352
$ ccc_mpeek 234481
Found license file: /usr/local/vampir-7.5/bin/lic.dat
Running 31 analysis processes...
Server listens on: curie1352:30000
```

- Finally you can launch Vampir, choose “Open remote…” and connect to the server



Going further

- Vampir tutorial (website)
  - http://www.vampir.eu/tutorial

PROFILING WITH VTUNE

Summary

1. Profiling: overview

2. Simple code profiling

3. Score-P & Vampir

4. **Profiling with Vtune**
   1. **Broadwell architecture**
   2. **Vtune amplifier**

Profiling with VTune

BROADWELL ARCHITECTURE

Broadwell Architecture

- The Broadwell partition of Cobalt contains
- Intel Xeon E5-2680 v4 CPUs
  - 14 cores
  - 2,4 to 3,3 GHz
  - 35 MB LLC
  - 2x 9,6 GT/s QPI links
  - 4x DDR4 2400 channels

Broadwell Architecture

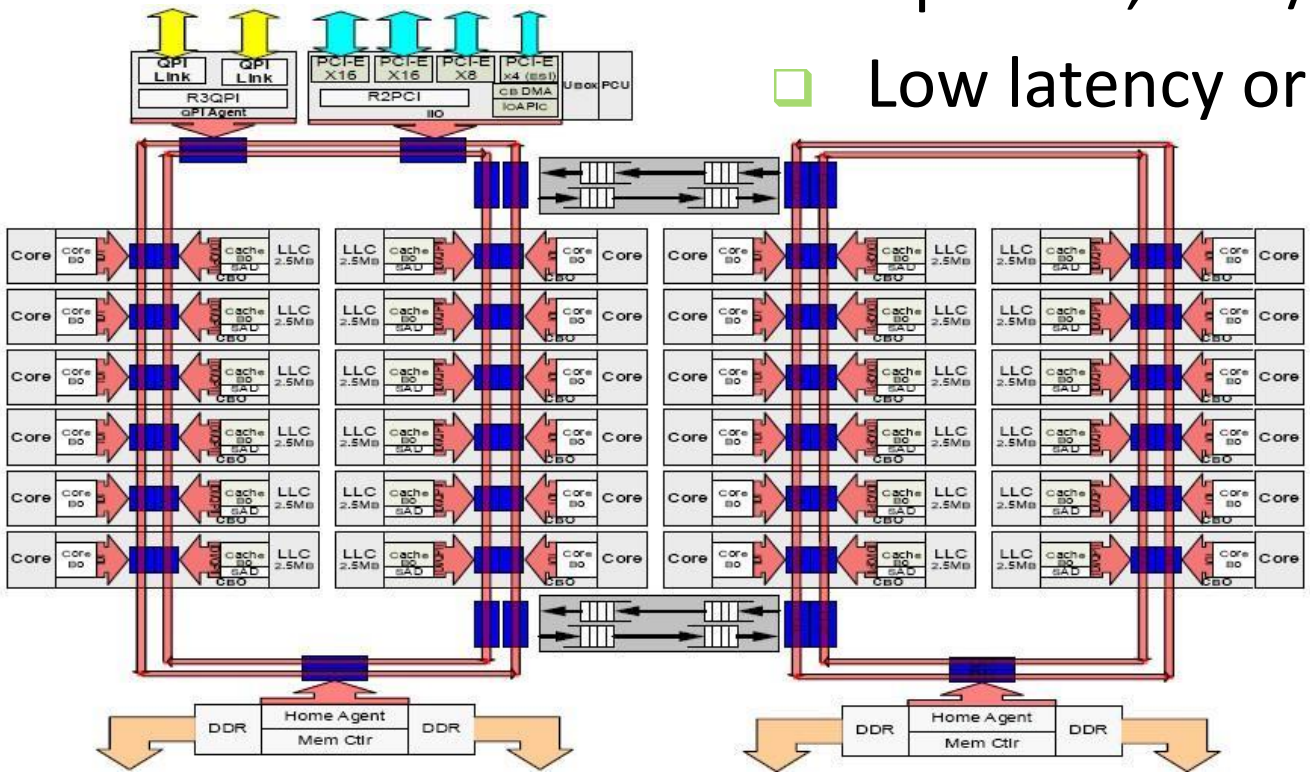
- Uncore: describes functions shared by the cores within a socket
  - Memory controller
  - PCI-E controller
  - QPI
  - Power Management unit
- Core
  - Out-of-order computation unit
  - Vector units
  - Caches

Communication ring

- 2 bidirectional & independent rings joined by 2 interconnects
- Ring bandwidth not communicated any more. It used to be 32 bytes width (half a cache line) on Sandy Bridge
- One cycle for each hop
- 5 cycles to go through ring interconnects
- A core does not use the ring to access data in its “local” cache.

Integrated memory controller

- Two dual channels controllers per socket
  - 4 memory channels per socket
  - Up to 76,8 GB/s
  - Low latency oriented

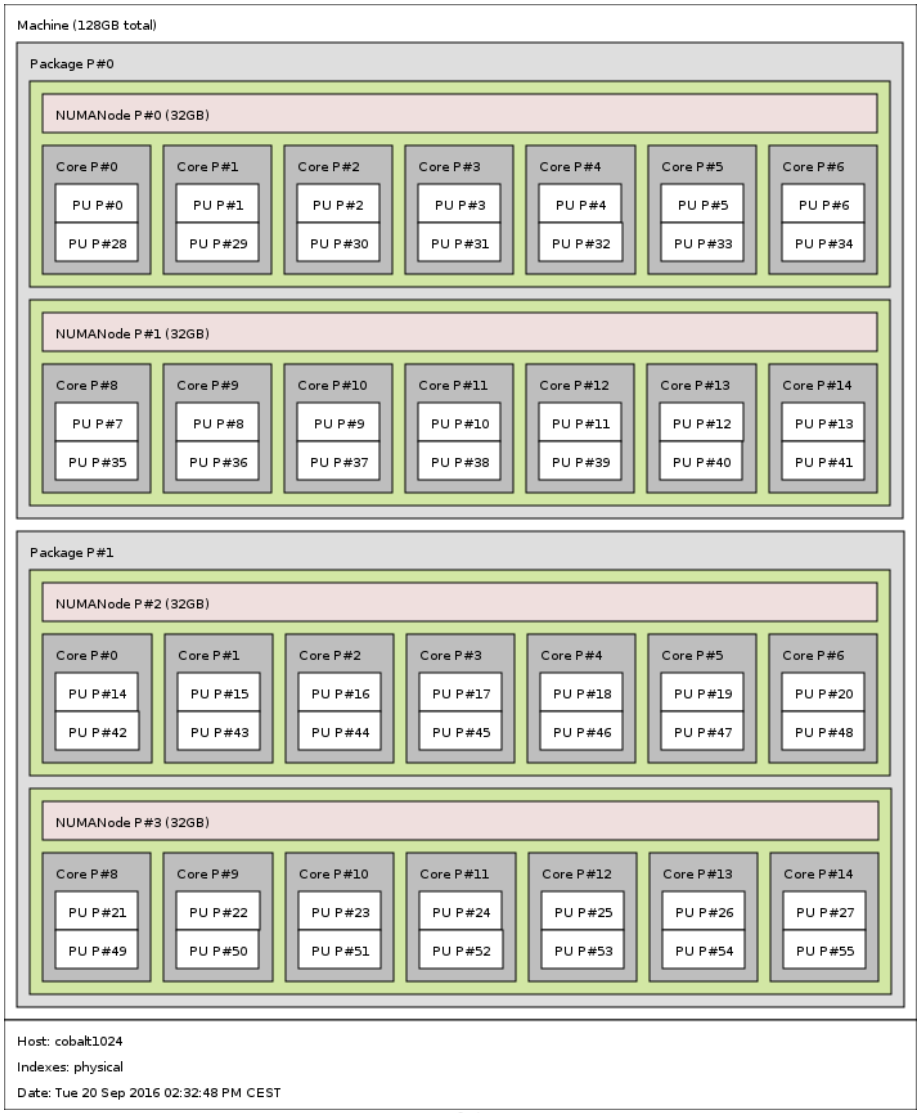


Selected Snoop Mode

- Cobalt nodes are set in “Cluster On Die” cache coherency mode
  - COD mode splits one CPU in two NUMA spaces
  - Broadwell nodes can be considered as 4 CPUs with 7 cores each
  - Can decrease last level cache access latency and local memory access latency

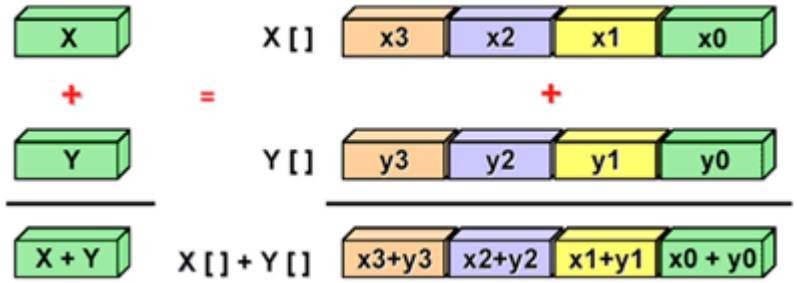
Cobalt configuration

Cobalt broadwell node with COD



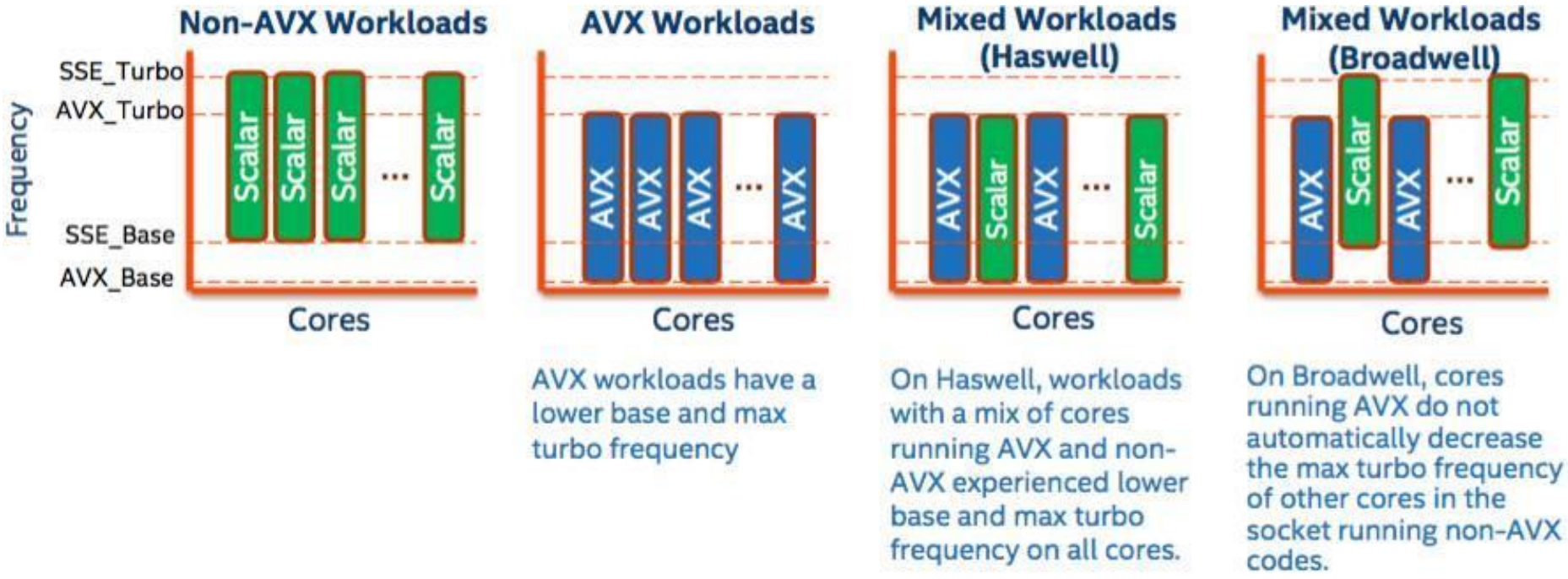
Core architecture

- Core
  - Independent execution unit
  - 2 FMA vector units
  - 38,4 Gflops DP at 2.4 GHz
- Vector unit
  - 256 bits SIMD (real & int)
  - FMA 3 (Fused Multiply Add)
    - C += A * B
    - C = A + C*B
  - Programming model
    - Intrinsic
    - Instructions generated by compiler



Turbo Boost

- Frequency range : 2,4 – 3,3 GHz
- AVX frequency range : 1,9 – 2,7 GHz

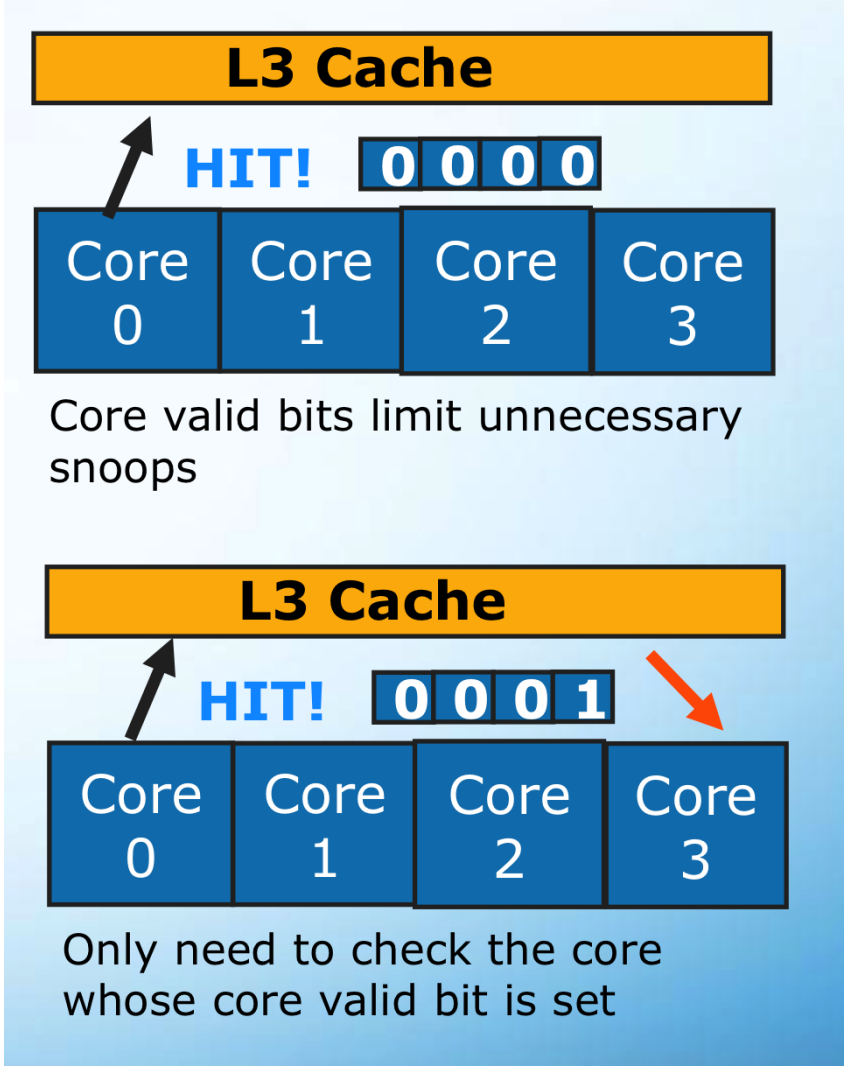


Cache

- Coherent inclusive cache on 3 levels
  - Data contained in L1/L2 is also contained in L3
- L3 cache shared by the cores inside a socket
  - Shared variables problem
  - False sharing
  - High efficiency

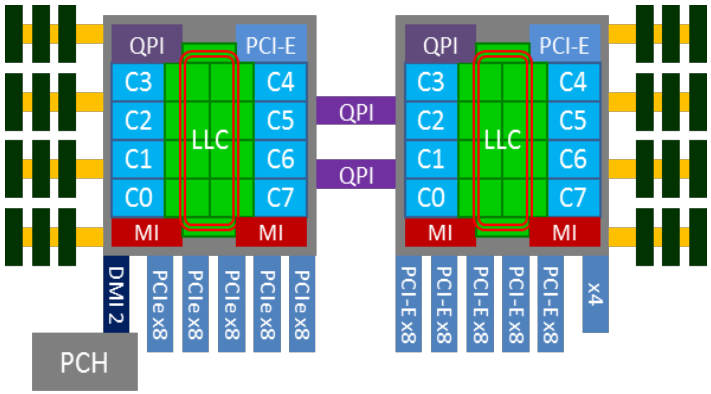
Cache levels managment

- A counter for each cache line
- Each bit represents the presence of data in L1/L2 cache of a core
- No check is performed for cores that do not contain the line in their cache
- If a bit is set to 1, data is read-only for other cores



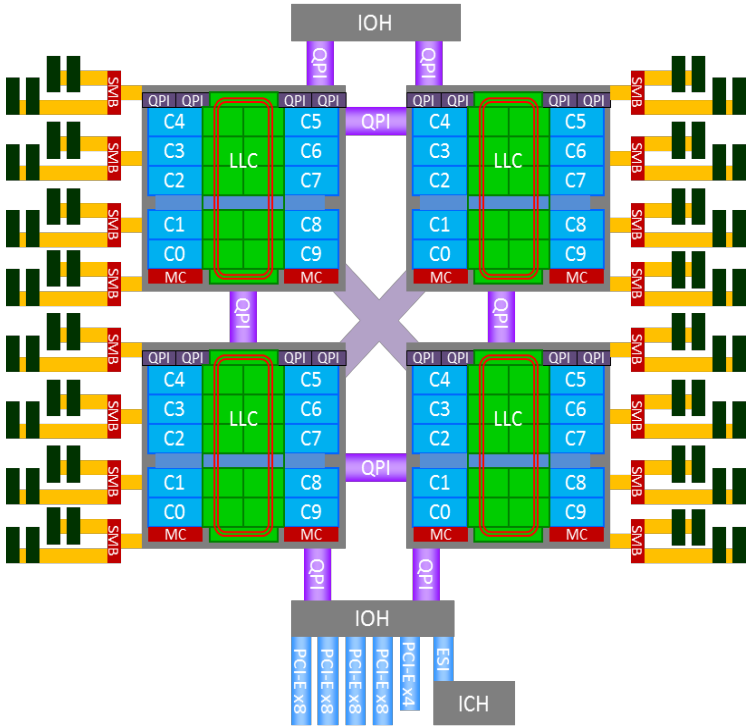
QPI

- QuickPath Interconnect
- Low latency fast interconnection to link CPUs
- Bandwidth
  - 6.4 GT/s on Nehalem => 12.8 GB/s in each direction
  - 8 GT/s on Sandy Bridge => 16 GB/s in each direction
  - 9,6 GT/s on Broadwell => 19,2 GB/s in each direction



NUMA System

- Non Uniform Memory Access
- Although the OS consider memory as a whole, it is physically spread over different locations
- Memory is local to socket
  - Higher bandwidth
  - Lower latency
- Remote memory
  - Higher latency
  - Access through QPI link



NUMA: remote access

- CPU0 needs a data which is neither in cache nor in local memory
- CPU0 asks data to CPU1 through QPI link
- CPU1 looks for the data into caches and local RAM
- Data is sent through QPI

Profiling with VTune

INTEL VTUNE AMPLIFIER

Aim

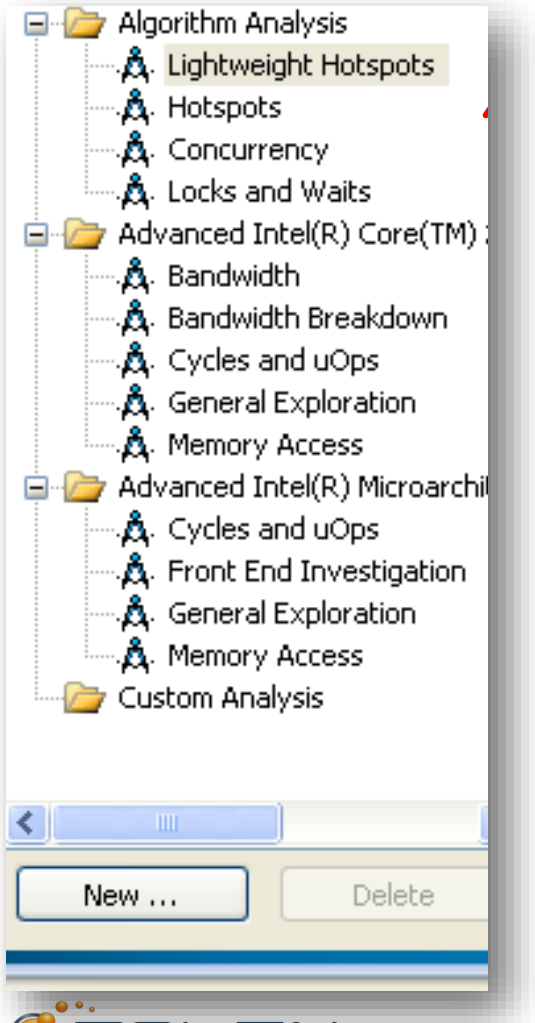
- Analyze the behavior of an application during its execution
  - Based on event sampling, instrumentation of functions and call stack sampling
  - Facilitate the quest for optimizable spots in a code
  - Highlights possible optimizations

Use

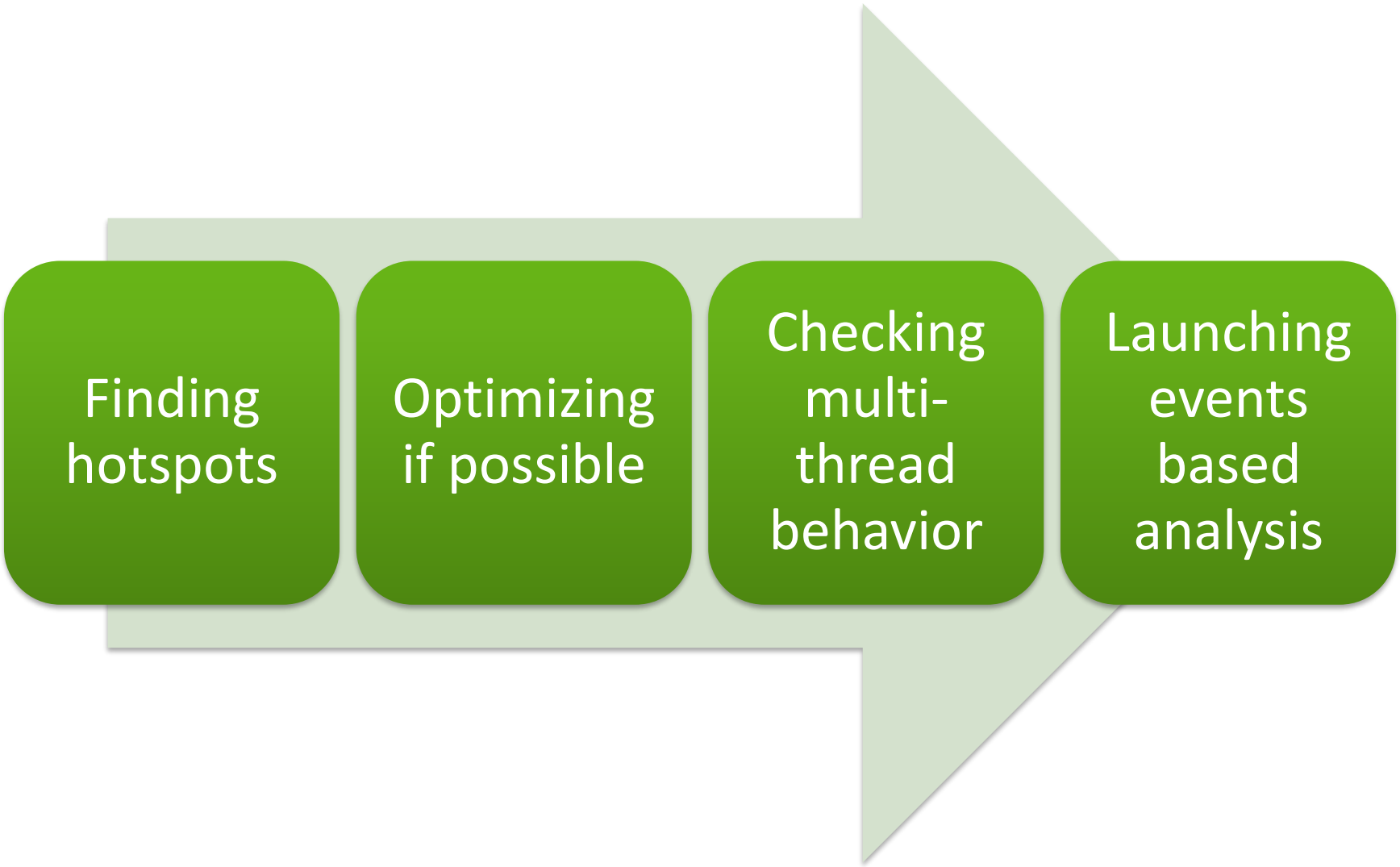
- Non expert programmers
  - Direct access to Help menu and counter explanations
  - Predefined analysis models
  - Quick and efficient results without need of advanced architecture knowledge
- Expert programmers
  - Availability of every hardware counters for different Intel CPU architectures
  - Possibility to make personalized analysis to find out specific issues

Analysis

- Different analysis depending on what you are looking for
  - Hotspots
  - Memory
  - Parallelism
  - Microarchitecture specific analysis
- A high number of predefined analysis profiles are available in the interface



Usual methodology



Working with Vtune

- There are 2 steps to work with Vtune
  - Launch a collection on your code
    - Through a submission script
    - amplxe-cl
    - It will generate a file '*.amplxe'
  - Analyze the result with the graphical interface
    - Always on computer nodes
    - amplxe-gui
    - From there, you can open the *.amplxe file

Launching a collection

- Usage:
  - Long run
  - Several/many processes
  - For later analysis

- Pick the collection you need:

```
$ amplxe-cl -help collect
#Syntax example: amplxe-cl –collect hotspots ./mybinary
```

Launching a collection

- For serial or multithreaded codes, launch amplxe-cl
  - In submission script:

```
module load vtune
amplxe-cl -collect hotspots ./mybinary
```

- For MPI codes, launching a collection through an appfile

```
$ cat appfile
  1 amplxe-cl -collect hotspots ./mybinary
  3 ./mybinary
```

  - And in the submission script:

```
ccc_mprun -f appfile
```

Use the graphical interface

- Analyzing the results
  - Show gprof like output:

```
$ amplxe-cl -report gprof-cc -result-dir outputdir -format text
```

  - Use graphical interface
    - Never on the login node!
    - Using HPCDrive (highly recommended)

```
$ amplxe-gui
```

    - Batch mode:

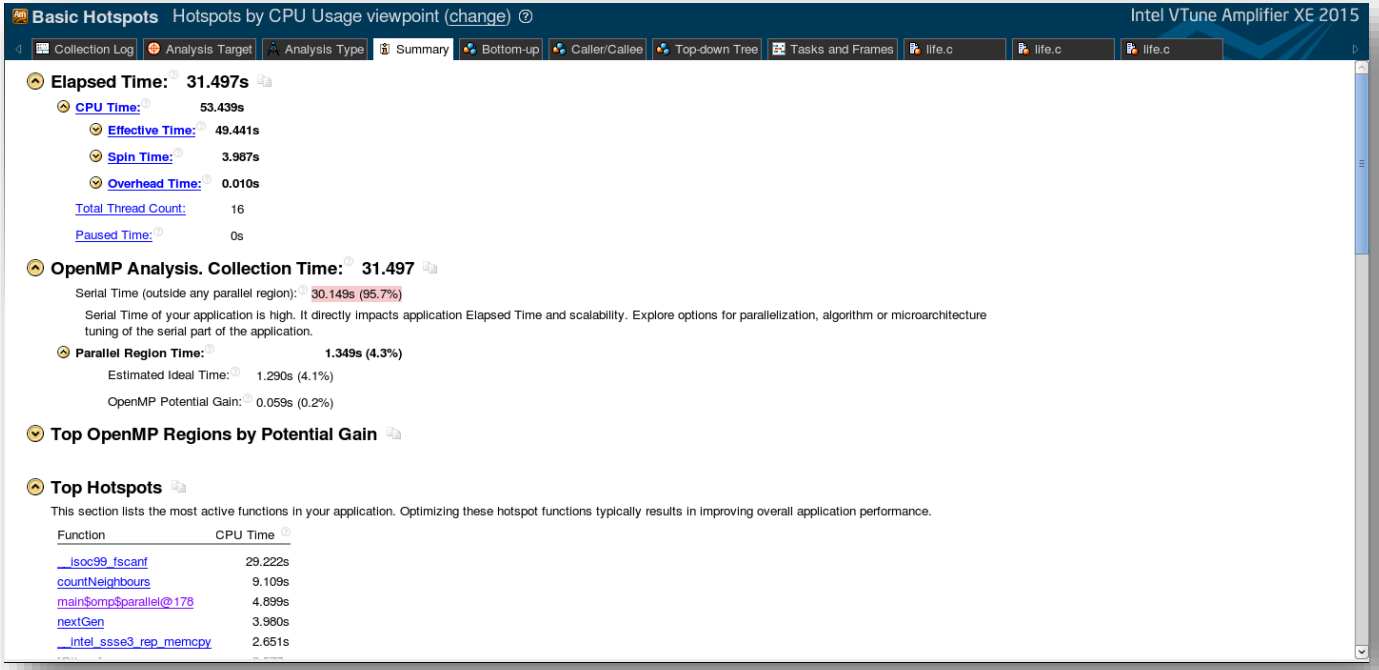
```
ccc_mprun -Xfirst amplxe-gui
```

    - Interactively:

```
$ ccc_mprun -Xfirst -p broadwell -x amplxe-gui
```

Help

- After each analysis, a summary is displayed
- Hardware counters are highlighted if they are out of certain range
- « Easy to understand »: values are extracted/computed from hardware counters



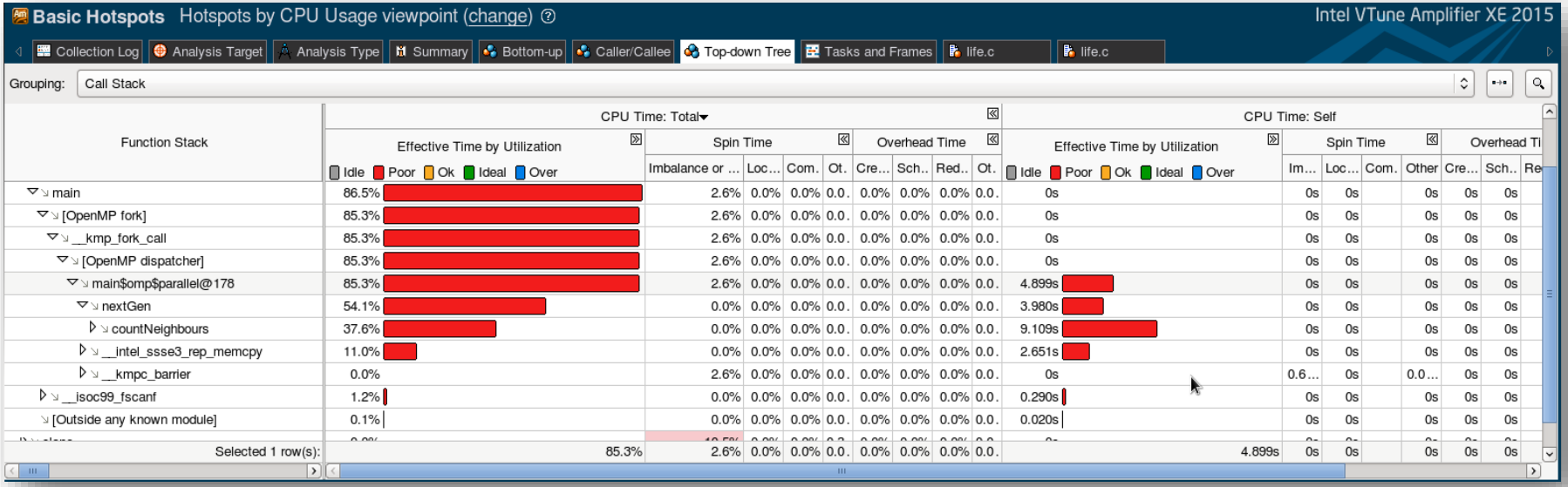
Hotspots

- Call stack trace: let you know the time spent in each part of the code

- Examples:
  - Check that time is spent on the right task
  - Avoid optimizing a function with low computational cost

Finding hotspots

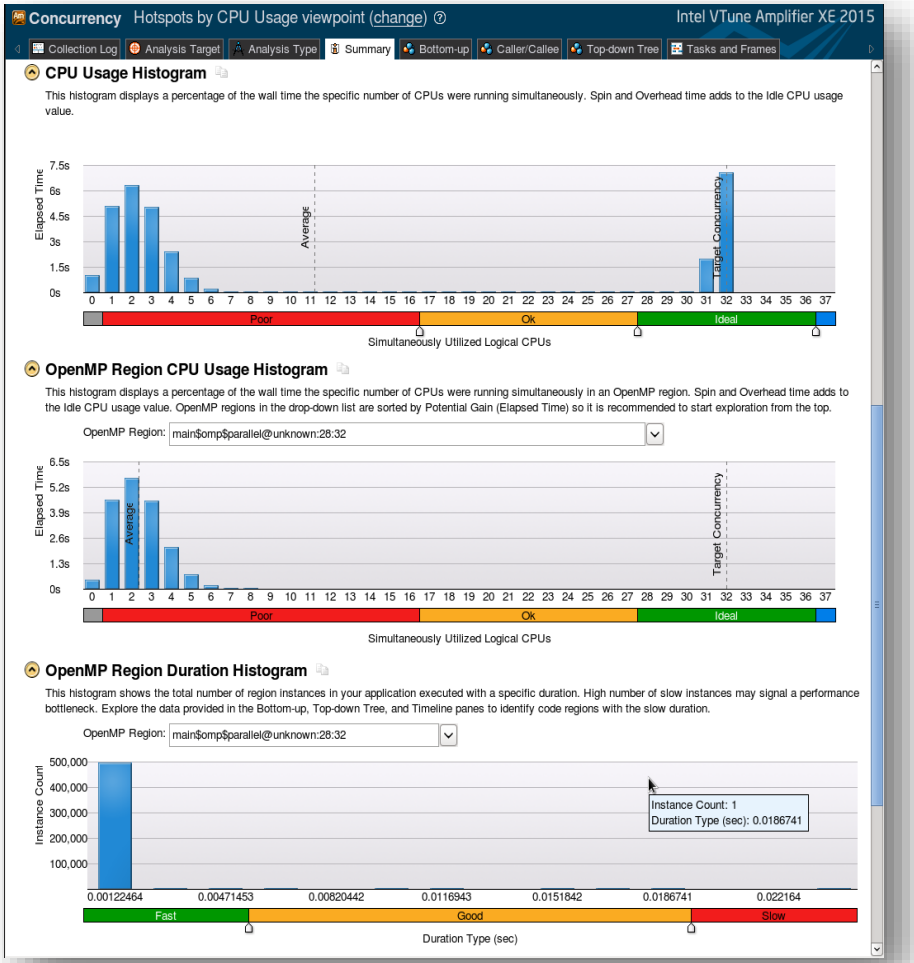
- Use the *call stack* display and the *top-down tree* display



- Check that time spent match with the task that needs to be carried out

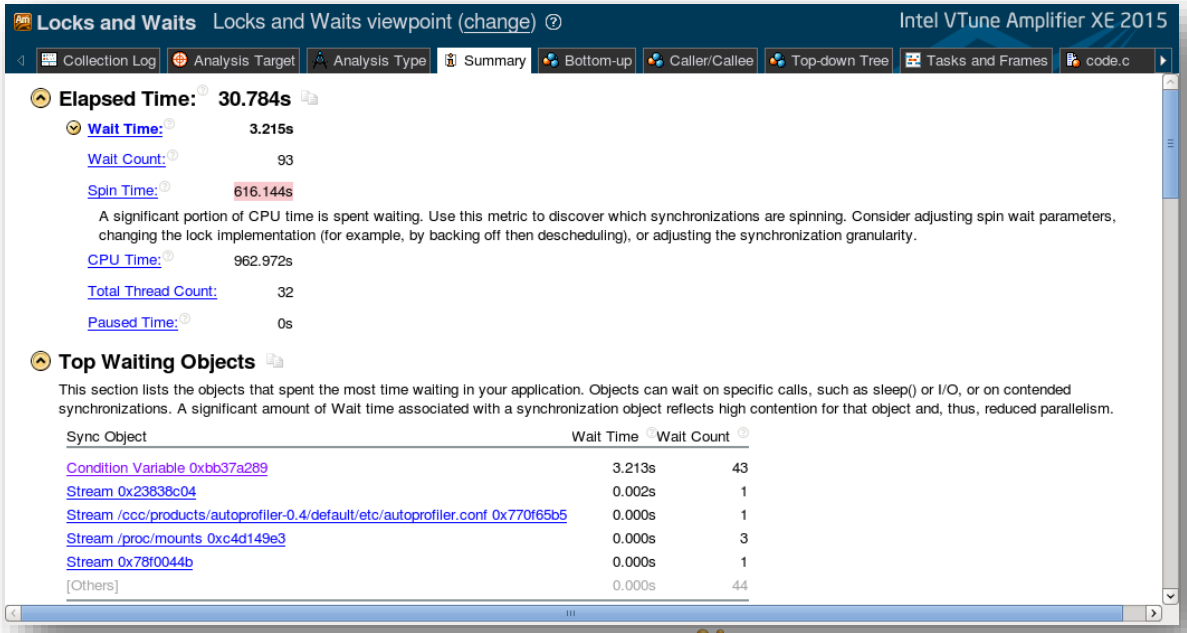
Multi-threaded analysis

- Concurrency
  - Gives a precise idea of the level of parallelism reached
  - Allows to quickly find the functions which have poor parallel execution
  - Intuitive interpretation



Multi-threaded analysis

- Locks-and-waits
  - Find the reason of poor multi-threaded scalability
  - Check that waits are necessary
  - Know the reasons of each wait
    - I/O
    - Synchronizations
    - Threading API

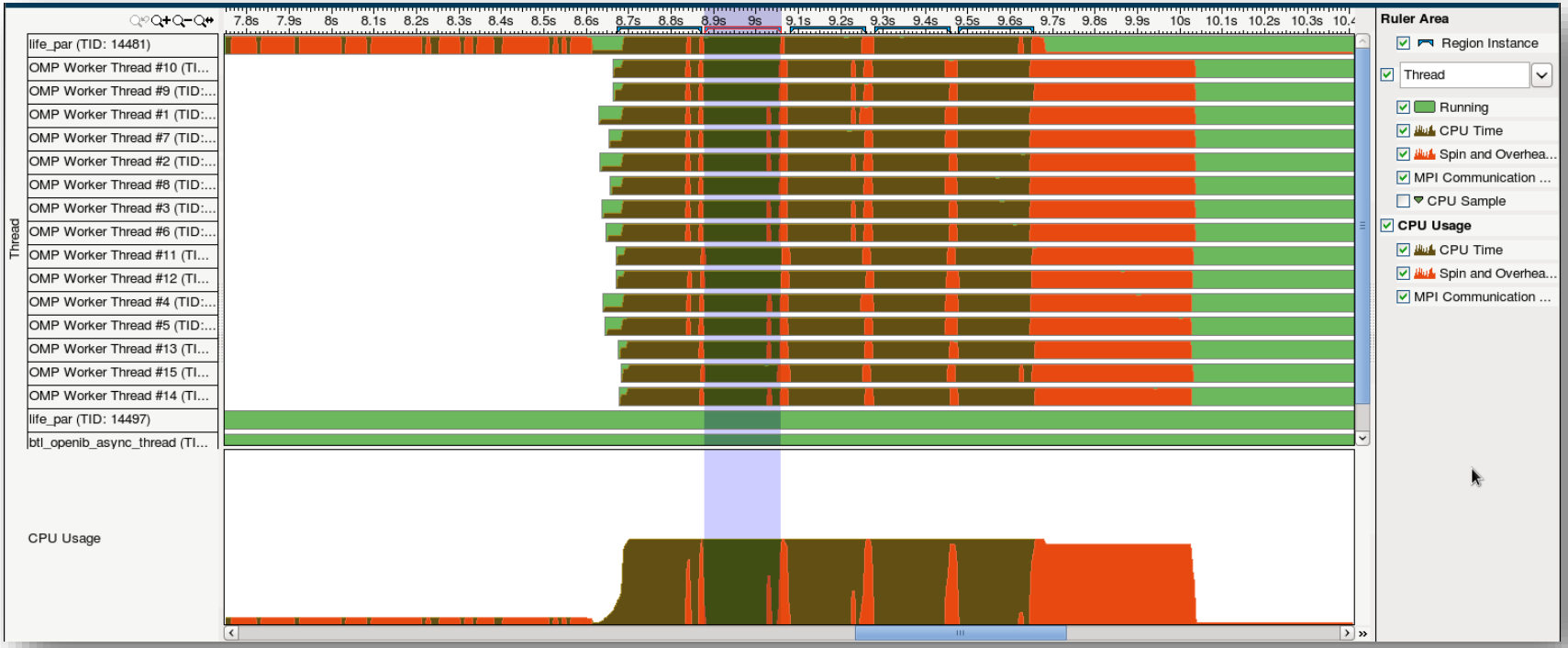


Counters and hardware samples analysis

- Architecture dependent
- More precise but requires higher expertise
- Allows to find:
  - Cache misses
  - Memory access problems
  - Branch miss-prediction
  - FP performances
  - Etc.

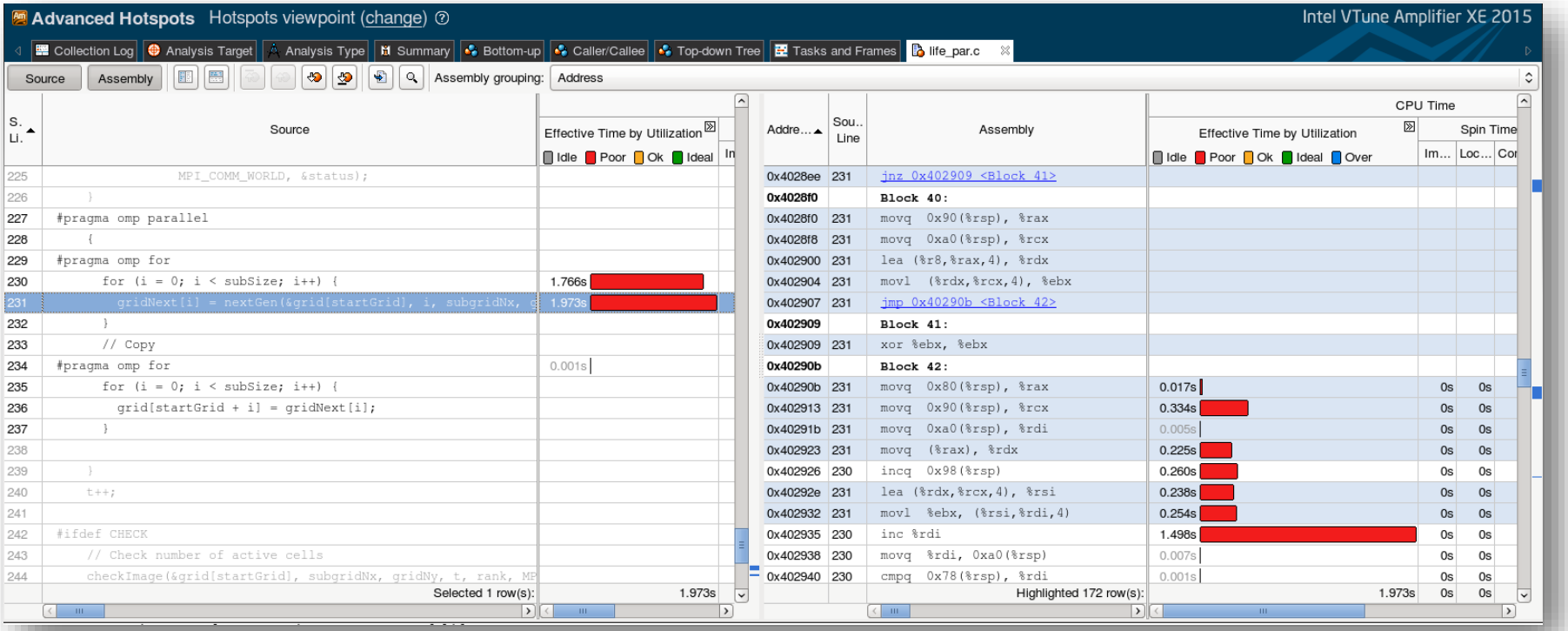
Going into details

- A timeline shows the activity of each thread
- It allows to:
  - zoom and filter on selected sections
  - choose metrics to display



Going into details

- The code / assembly can be visualized simultaneously with related captured counters



CONCLUSION

Remaining profilers

- To complete this training, we encourage you to have a look at other tools that are worth being considered
  - igprof (memory profiling)
  - threadspotter
  - HPC Toolkit
  - TAU (automatic code instrumentation)
  - Paraver (Vampir equivalent)
  - Intel Advisor/Inspector (predict and improve parallelism)
  - Valgrind (Cachegrind, Callgrind…)

- To check all the available profilers:

```
module search profiler
```

Igprof

- IgProf is a simple nice tool for measuring and analyzing application memory and performance characteristics

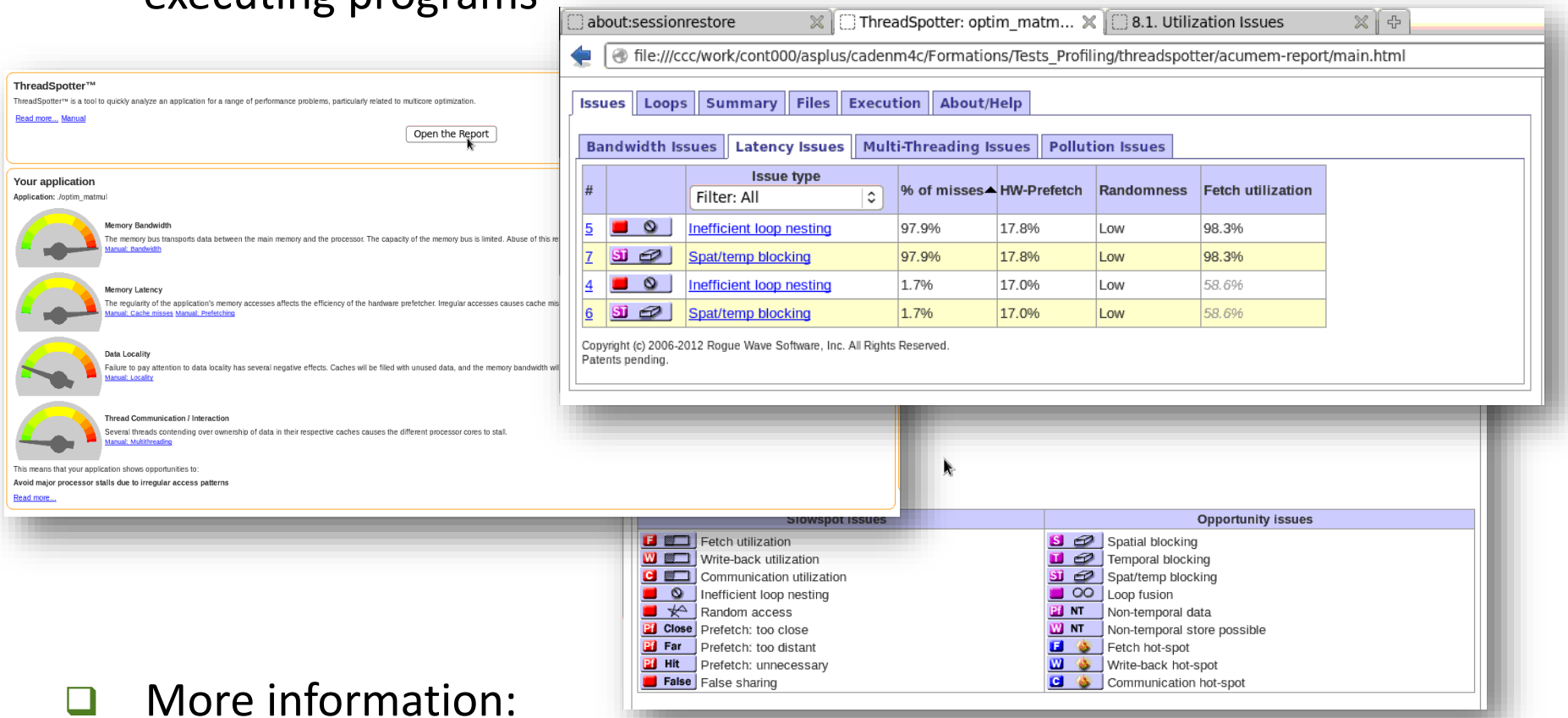
```
module load igprof
igprof -d -mp -z -o igprof.mp.gz myApp >& igtest.mp.log
```

```
Counter: MEM_TOTAL

-------------------------------------------------------------------------------
Flat profile (cumulative >= 1%)

% total     Total     Calls  Function
  100.0     2'416         3  <spontaneous> [1]
  100.0     2'416         3  _start [2]
  100.0     2'416         3  __libc_start_main [3]
  100.0     2'416         3  main [4]
   99.3     2'400         2  MAIN__ [5]
   99.3     2'400         2  for_alloc_allocatable [6]
   82.8     2'000         1  main_IP_b_ [7]
    0.7        16         1  for_rtl_init_ [8]


-------------------------------------------------------------------------------
Flat profile (self >= 0.01%)

% total      Self     Calls  Function
  99.34     2'400         2  for_alloc_allocatable [6]
   0.66        16         1  for__get_vm [9]
   0.00         0         0  <spontaneous> [1]
```

- More information:
  http://igprof.org/index.html

Threadspotter

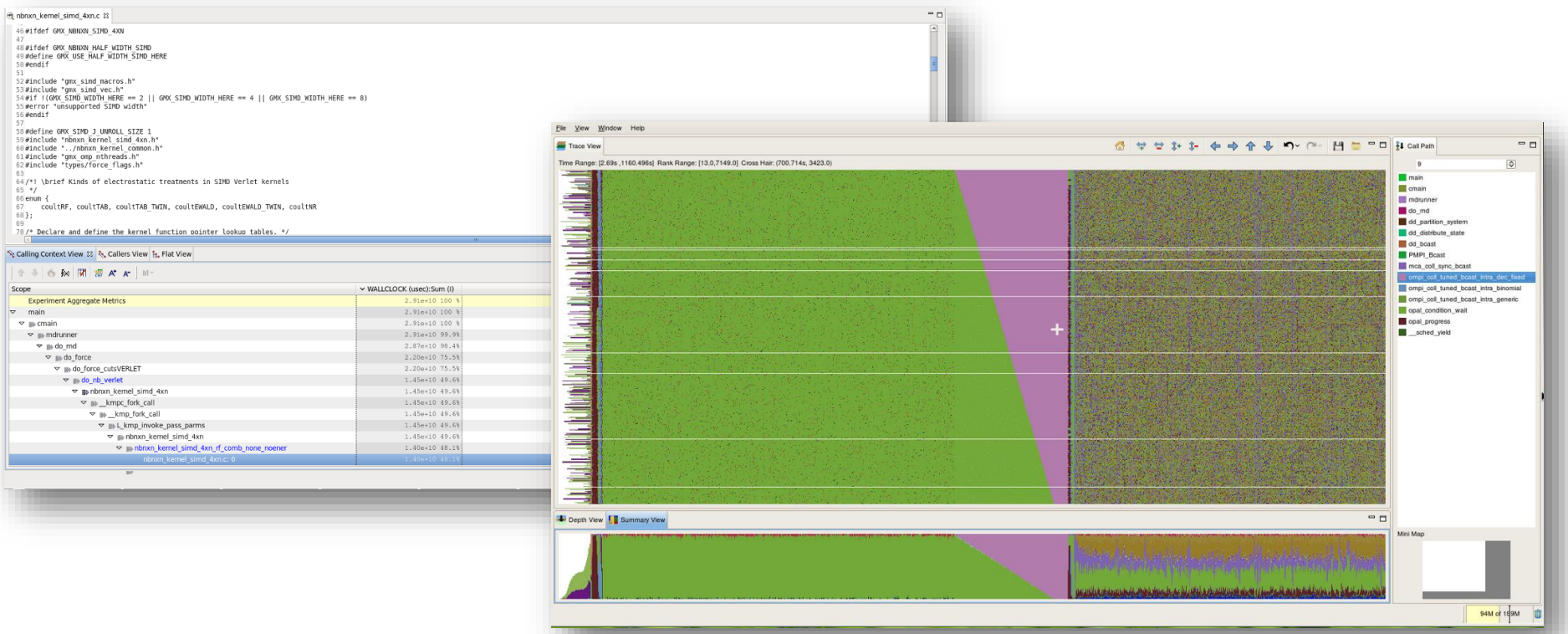
- ThreadSpotter efficiently monitors the memory fingerprint of executing programs



- More information:
  http://www.paratools.com/threadspotter

HPC Toolkit

- HPCToolkit is an integrated suite of tools for measurement and analysis of program performance



- More information:
  http://hpctoolkit.org/documentation.html

Conclusion

- We have been through a set of tools, useful in both debugging and profiling fields. Depending on the kind of code, one tool or another may be most effective and relevant
- The challenge is now on your side
- Being comfortable with these tools takes from little to average time but the investment deserves to be done
- From this knowledge, you will design scalable and future proof codes for present and coming computing architectures

Conclusion

- Good knowledge of tools implies:
  - Efficient debug and profiling
  - Better understanding of system behaviors and architectures which implies better conception of your applications
  - Good results within a small amount of time

Reminder

1. Additional trainings:
    1. **MPI & OpenMP**
    2. **Debugging & Profiling**
    3. **Advanced Optimization & Parallelism**
2. Command line information:
    1. **machine.info**
    2. **evince /ccc/products/cdc_docs/pdf/cobalt.info.pdf**
3. Web site:
    1. **www-ccrt.ccc.cea.fr**
    2. **www-ccrt.ccc.cea.fr/fr/tgcc/User_documentation.html**
4. Support:
    1. **hotline.tgcc@cea.fr**