Performance analysis : Hands-on

- **time**
  - Wall/CPU
  - parallel context
- **gprof**
  - flat profile/call graph
  - self/inclusive
  - MPI context
- **VTune**
  - hotspots, per line profile
  - advanced metrics :
    - general exploration, snb-memory-access, concurrency …
  - parallel context

Performance analysis : Hands-on

- **Scalasca**
  - Load imbalance
  - PAPI counters
- **Vampir**
  - trace
- **A memory instrumentation**

Hands on Environment

The platform : Poincare

Architecture

- 92 nodes
- 2 Sandy Bridge x 8 cores
- 32 Go

Environnement

- Intel 13.0.1
- OpenMPI 1.6.3

Job & resources manager

- Today : Max **1** node / job

Hwloc : lstopo

- Compile on interactive nodes :

[mdlslx181]$ poincare

- Run on :

[poincareint01]$ llinteractif 1 callmds 6

The code : Poisson

- **Poisson – MPI @ IDRIS**
  - C / Fortran
- **Code reminder**
  - Stencil :

```
u_new[ix,iy] = c0 * (
      c1 * ( u[ix+1,iy] + u[ix-1,iy] )
    + c2 * ( u[ix,iy+1] + u[ix,iy-1] )
               - f[ix,iy] );
```

  - Boundary limits :

```
u = 0
```

  - Convergence criterion :

```
max | u[ix,iy] - u_new[ix,iy] | < eps
```

- **MPI**
  - Domain decomposition
  - Exchanging ghost cells

The code : Poisson (2)

- **Data size**

```
$ cat poisson.data
480
400
```

- **Validation :**
  - *compile on an interactive node :*

[poincareint01]$ make read

[poincareint01]$ make calcul_exact

  - *Run on a compute node*

[poincare001]$ make verification

…

BRAVO, Vous avez fini

Basics

time : Elapsed, CPU

Command lines :

```
$ time mpirun -np 1  ./poisson.mpi
```

Sequential results :

…

Convergence apres 913989 iterations en **425.560393** secs

… **MPI_Wtime : Macro instrumentation**

real 7m6.914s **Time to solution**

user 7m6.735s **Resources used**

sys 0m0.653s

time : MPI

Command lines :

```
$ time mpirun -np 16 ./poisson.mpi
```

MPI results :

```
…
Convergence apres 913989 iterations en 38.221655 secs
…
real   0m39.866s
user   10m27.603s
sys    0m1.614s
```

time : OpenMP

Command lines :

```
$ export OMP_NUM_THREADS=8
$ time mpirun -bind-to-socket -np 1 ./poisson.omp
```

OpenMP results :

```
…
Convergence apres 913989 iterations en 172.729974 secs
…
real   2m54.224s
user   22m32.978s
sys    0m31.832s
```

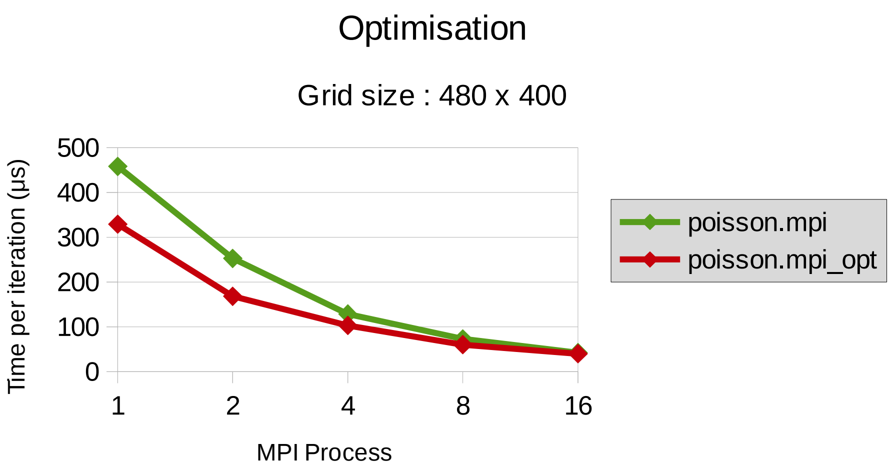
Resources

- **Binding :**

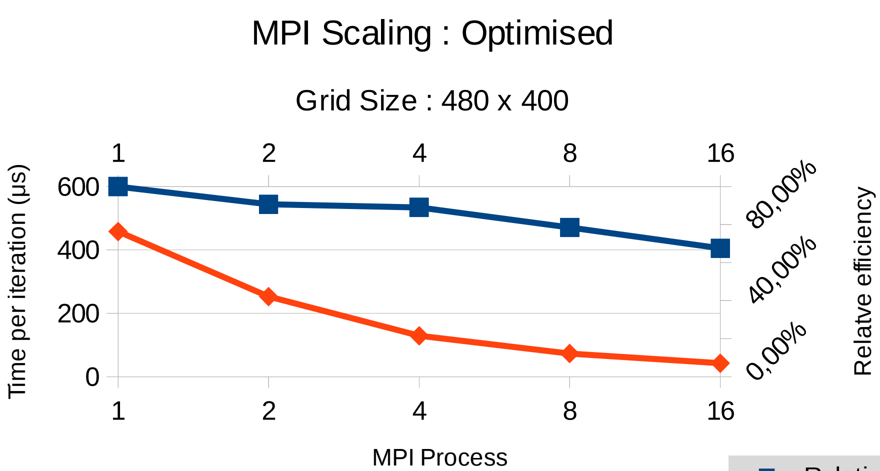
```
$ time mpirun -report-bindings -np 16 ./poisson.mpi 100000
Convergence apres 100000 iterations en 4.249197 secs
$ time mpirun -bind-to-none    -np 16 ./poisson.mpi 100000
Convergence apres 100000 iterations en 25.626133 secs
$ man mpirun / mpiexec … for the required option

$ export OMP_NUM_THREADS=8
$ time mpirun -report-bindings -np 1 ./poisson.omp 100000
$ time mpirun -bind-to-socket  -np 1 ./poisson.omp 100000
```

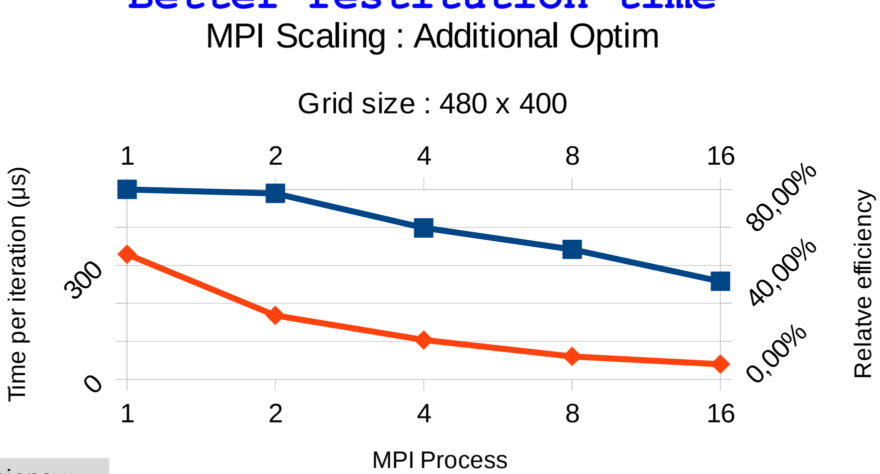
- **But not only :**
  - Process/thread distribution
  - Dedicated resources

Scaling metrics & Optimisation



Damaged scaling but ...
Better restitution time





gprof : Basics

- Widely available :
  - GNU, Intel, PGI …
- Regular code pattern, limit the number of iterations
  - Should be consolidated after optimisation
  - Measure reference on a limited number of iterations
- Edit make_inc to enable -pg option, then recompile
- **Command lines :**

```
$ mpirun -np 1 ./poisson 100000
Convergence apres 100000 iterations en 47.714439 secs
$ ls gmon.out
gmon.out
$ gprof poisson gmon.out
```

gprof : Flat profile

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 74.87     35.66    35.66   100000   356.60   356.60  calcul
 25.27     47.70    12.04   100000   120.37   120.37  erreur_globale
  0.00     47.70     0.00   100000     0.00     0.00  communication
```

Consolidate application behavior using an external tool

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00   47.70                 main [1]
               35.66    0.00  100000/100000      calcul [2]
               12.04    0.00  100000/100000      erreur_globale [3]
                0.00    0.00  100000/100000      communication [4]
                0.00    0.00       1/1           creation_topologie [5]
...
-----------------------------------------------
               35.66    0.00  100000/100000      main [1]
[2]     74.8   35.66    0.00  100000         calcul [2]
```

gprof : Call graph

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 74.87     35.66    35.66   100000   356.60   356.60  calcul
 25.27     47.70    12.04   100000   120.37   120.37  erreur_globale
  0.00     47.70     0.00   100000     0.00     0.00  communication

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00   47.70                 main [1]
               35.66    0.00 100000/100000       calcul [2]
               12.04    0.00 100000/100000       erreur_globale [3]
                0.00    0.00 100000/100000       communication [4]
                0.00    0.00      1/1            creation_topologie [5]
...
-----------------------------------------------
               35.66    0.00 100000/100000       main [1]
[2]     74.8   35.66    0.00 100000          calcul [2]
```

Addtionnal informations : gprof & MPI

- A per process profile :
  - Setting environment variable : **GMON_OUT_PREFIX**

- **Command lines :**

```
$ cat exec.sh
-----------------------------------------------------------
#!/bin/bash
# "mpirun -np 1 env|grep RANK"
export GMON_OUT_PREFIX='gmon.out-'${OMPI_COMM_WORLD_RANK}
./poisson
-----------------------------------------------------------
$ mpirun -np 2 ./exec.sh
$ ls gmon.out-*
gmon.out-0.18003
gmon.out-1.18004
$ gprof poisson gmon.out-0.18003
```

Vtune - Amplificator

VTune : Start

- Optimise the available sources :

```
$ mpirun -np 16 ./poisson
Convergence apres 913989 iterations en 1270.757420 secs
```

- Reminder :

```
$ mpirun -np 16 ./poisson.mpi
Convergence apres 913989 iterations en 38.221655 secs
```

- Reduce number of iterations to 10000 :

```
$ mpirun -np 1 ./poisson 10000
Convergence apres 10000 iterations en 38.011032 secs
```

```
$ mpirun -np 1 amplxe-cl -collect hotspots -r profil ./poisson
Convergence apres 10000 iterations en 38.109222 secs
$ amplxe-gui profil.0 &
```

https://software.intel.com/en-us/qualify-for-free-software/student

VTune : Analysis

Welcome | profil.0

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

Intel VTune Amplifier XE 2015

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Elapsed Time: 39.350s

Total Thread Count:	2
Overhead Time:	0s
Spin Time:	0s
CPU Time:	39.150s
Paused Time:	0s

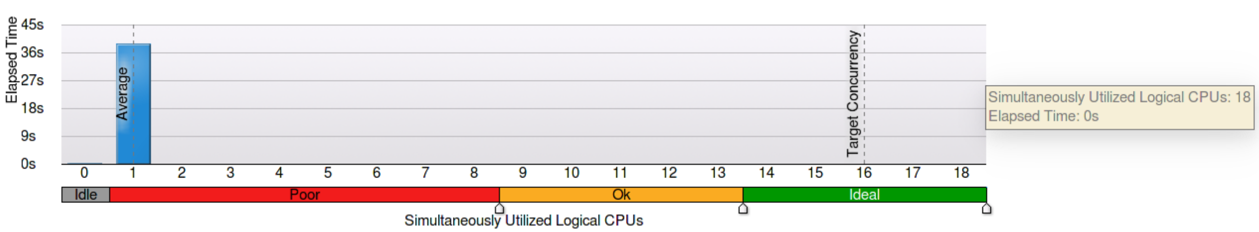
Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	CPU Time
calcul	35.220s
erreur_globale	2.770s
__psm_ep_close	1.000s
read	0.070s
PMPI_Init	0.020s
[Others]	0.070s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

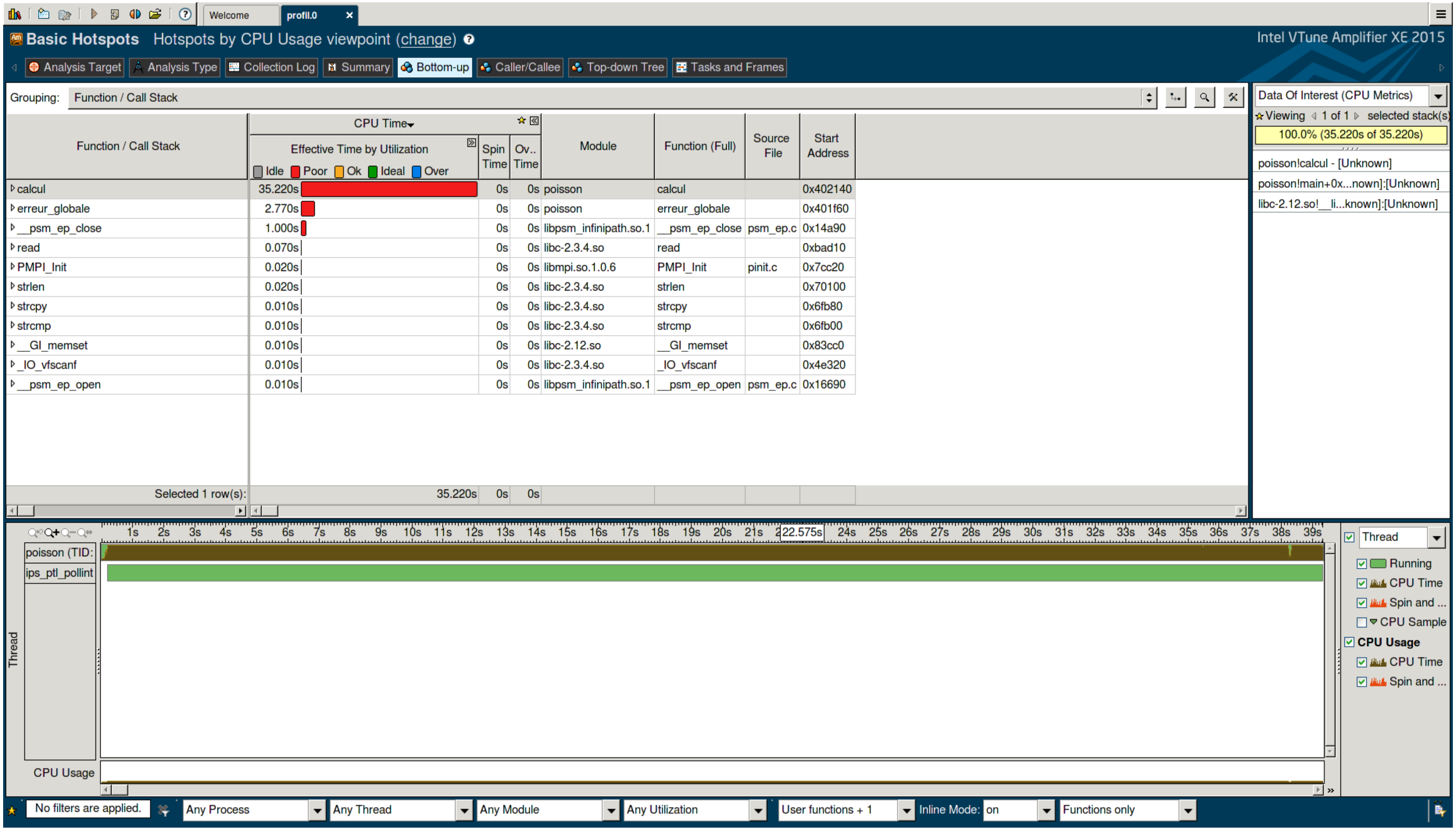


Collection and Platform Info

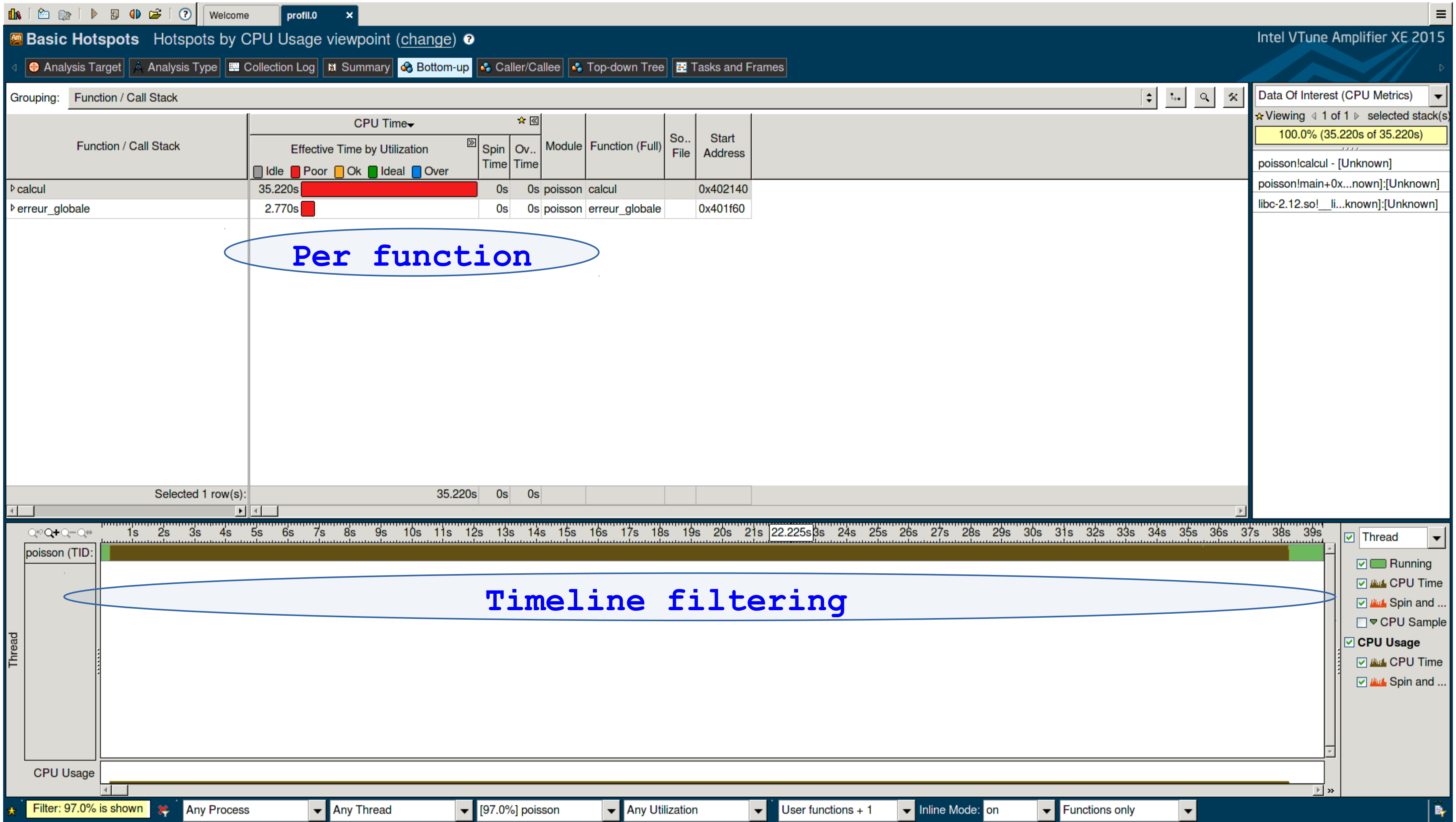
This section provides information about this collection, including result set size and collection platform data.

Application Command Line:	./poisson
Operating System:	2.6.32-431.el6.x86_64 CentOS release 6.5 (Final)
MPI Process Rank:	0
Computer Name:	poincare090-adm
Result Size:	2 MB

VTune : Profile

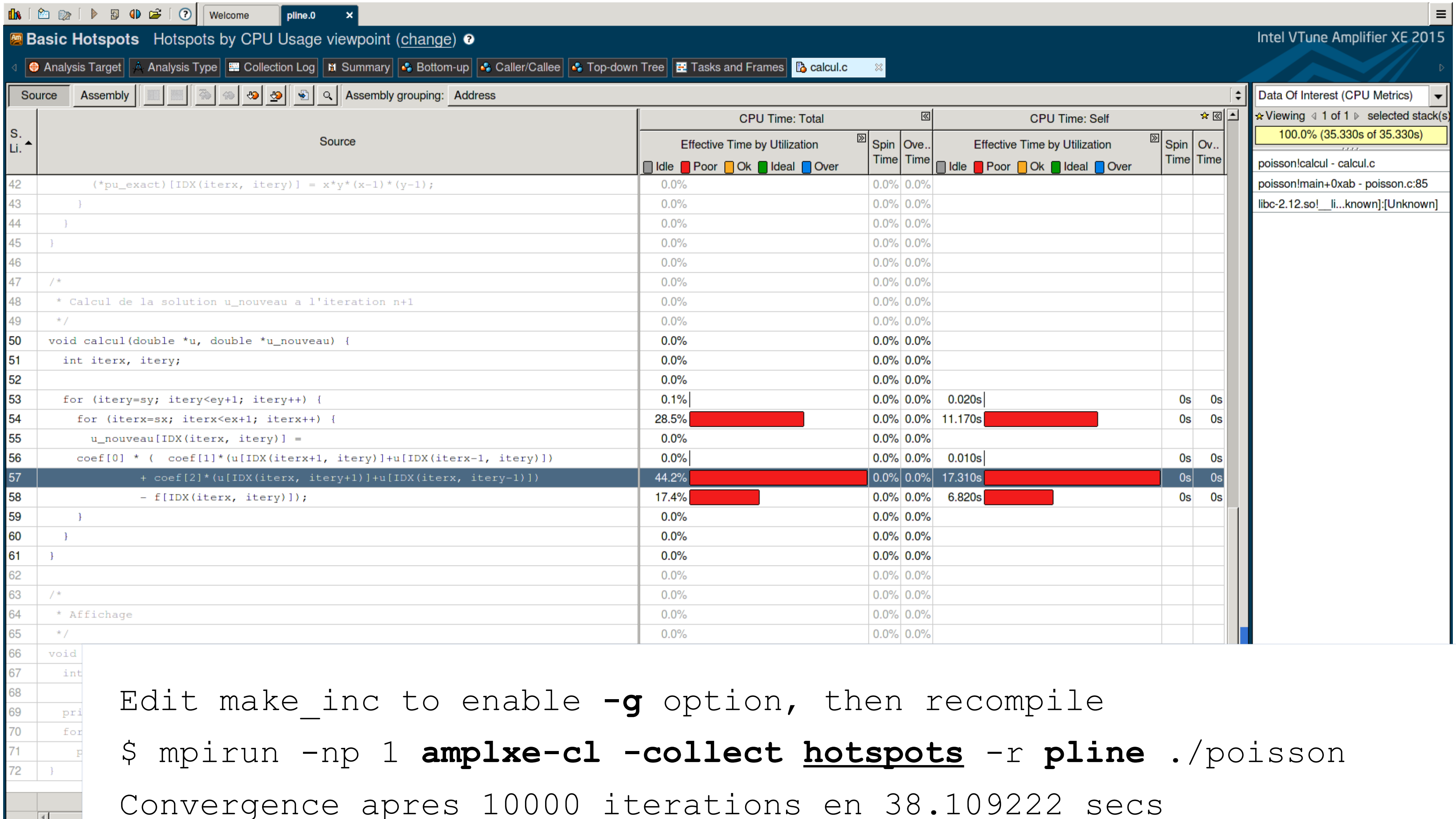


VTune : Data filtering



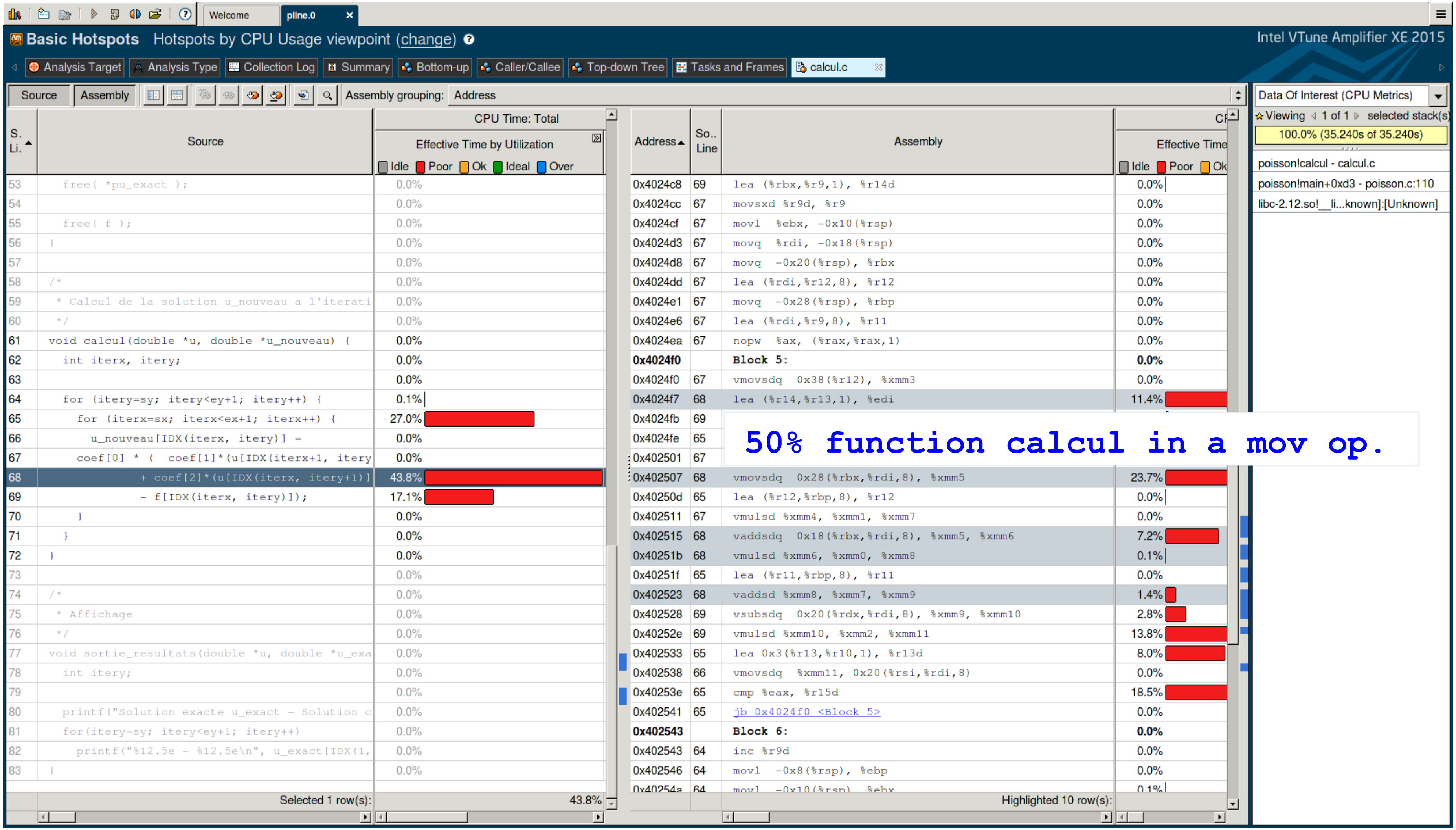
Application/MPI/system

VTune : Per line profile




```
Edit make_inc to enable -g option, then recompile
$ mpirun -np 1 amplxe-cl -collect hotspots -r pline ./poisson
Convergence apres 10000 iterations en 38.109222 secs
$ amplxe-gui pline.0 &
```

VTune : Line & Assembly

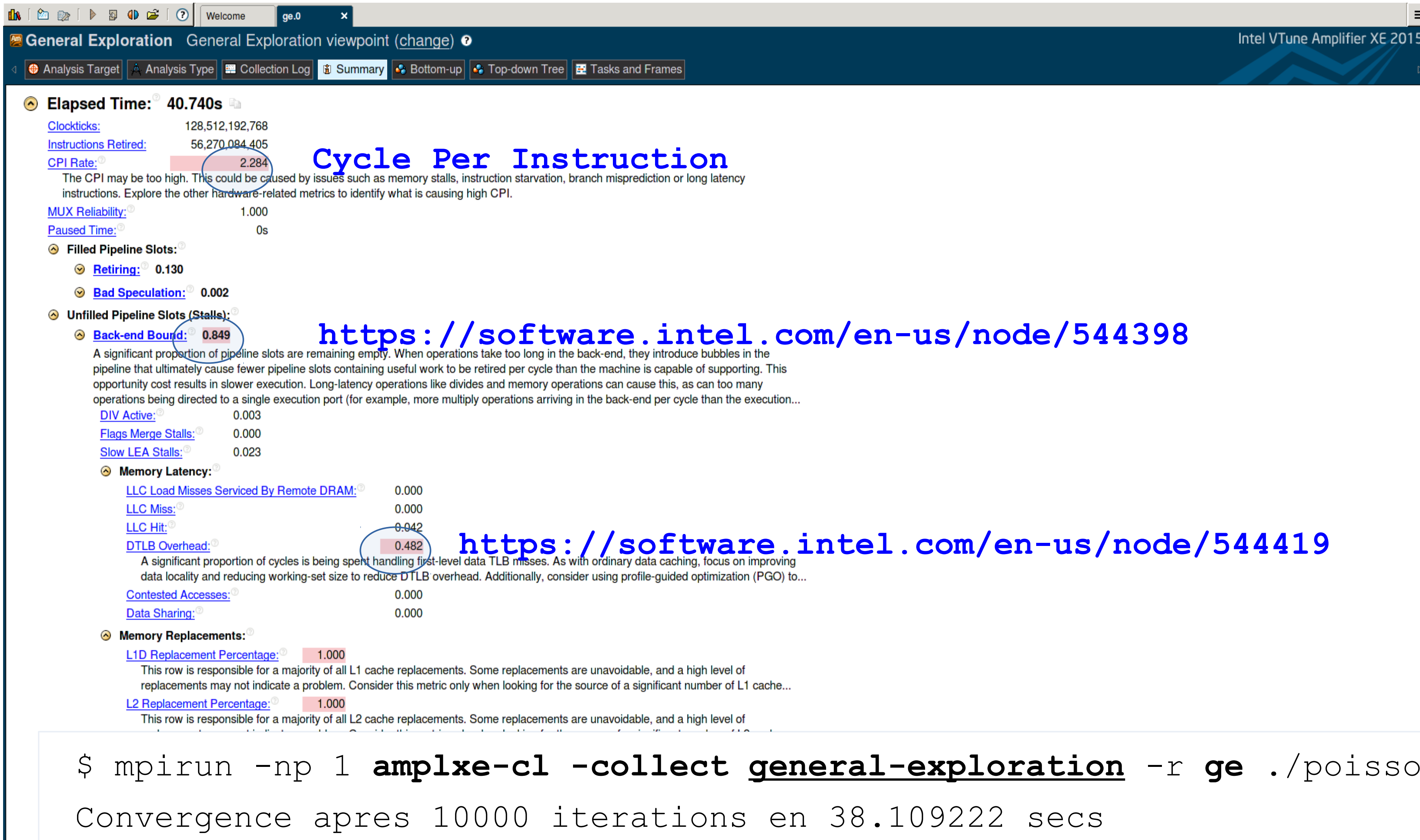


50% function calcul in a mov op.

Addtionnal informations : Command line profile

```
$ amplxe-cl -report hotspots -r profil.0
Function         Module                    CPU Time:Self
calcul           poisson                          35.220
erreur_globale   poisson                           2.770
__psm_ep_close   libpsm_infinipath.so.1            1.000
read             libc-2.3.4.so                     0.070
PMPI_Init        libmpi.so.1.0.6                   0.020
strlen           libc-2.3.4.so                     0.020
strcpy           libc-2.3.4.so                     0.010
__GI_memset      libc-2.12.so                      0.010
_IO_vfscanf      libc-2.3.4.so                     0.010
__psm_ep_open    libpsm_infinipath.so.1            0.010
```

VTune : Advanced Metrics




```
$ mpirun -np 1 amplxe-cl -collect general-exploration -r ge ./poisson
Convergence apres 10000 iterations en 38.109222 secs
$ amplxe-gui ge.0 &
```

Advanced Metrics : Back-End Bound

Search document...

Intel® VTune™ Amplifier XE 2016 and Intel® VTune™ Amplifier 2016 for Systems Help (Linux*, OS X*)

- Home
- Legal Information
- › Introducing the Intel® VTune™ Amplifier
- › Getting Started with Intel® VTune™ Amplifier
- › Tuning Methodology
- › Performance Analysis User's Guide
- › Energy Analysis User's Guide
- › Troubleshooting

Back-End Bound

Metric Description

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the back-end of the pipeline. Back-end metrics describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.

Parent topic: CPU Metrics Reference

- Memory Bandwidth
- Port Utilization

See Also

Reference for Performance Metrics
Interpreting General Exploration Data

Prev Average Loop Trip Count
Next Memory Bandwidth

For more complete information about compiler optimizations, see our Optimization Notice.

Advanced Metrics : DTLB

intel Developer Zone

Join Today › Log in

Development › Tools › Resources ›

powered by Google

Search document...

Intel® VTune™ Amplifier XE 2016 and Intel® VTune™ Amplifier 2016 for Systems Help (Linux*, OS X*)

DTLB Overhead

Metric Description

In x86 architectures, mappings between virtual and physical memory are facilitated by a page table, which is kept in memory. To minimize references to this table, recently-used portions of the page table are cached in a hierarchy of 'translation look-aside buffers', or TLBs, which are consulted on every virtual address translation. As with data caches, the farther a request has to go to be satisfied, the worse the performance impact. This metric estimates the performance penalty paid for missing the first-level data TLB (DTLB) that includes hitting in the second-level data TLB (STLB) as well as performing a hardware page walk on an STLB miss.

Possible Issues

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Parent topic: L1 Bound

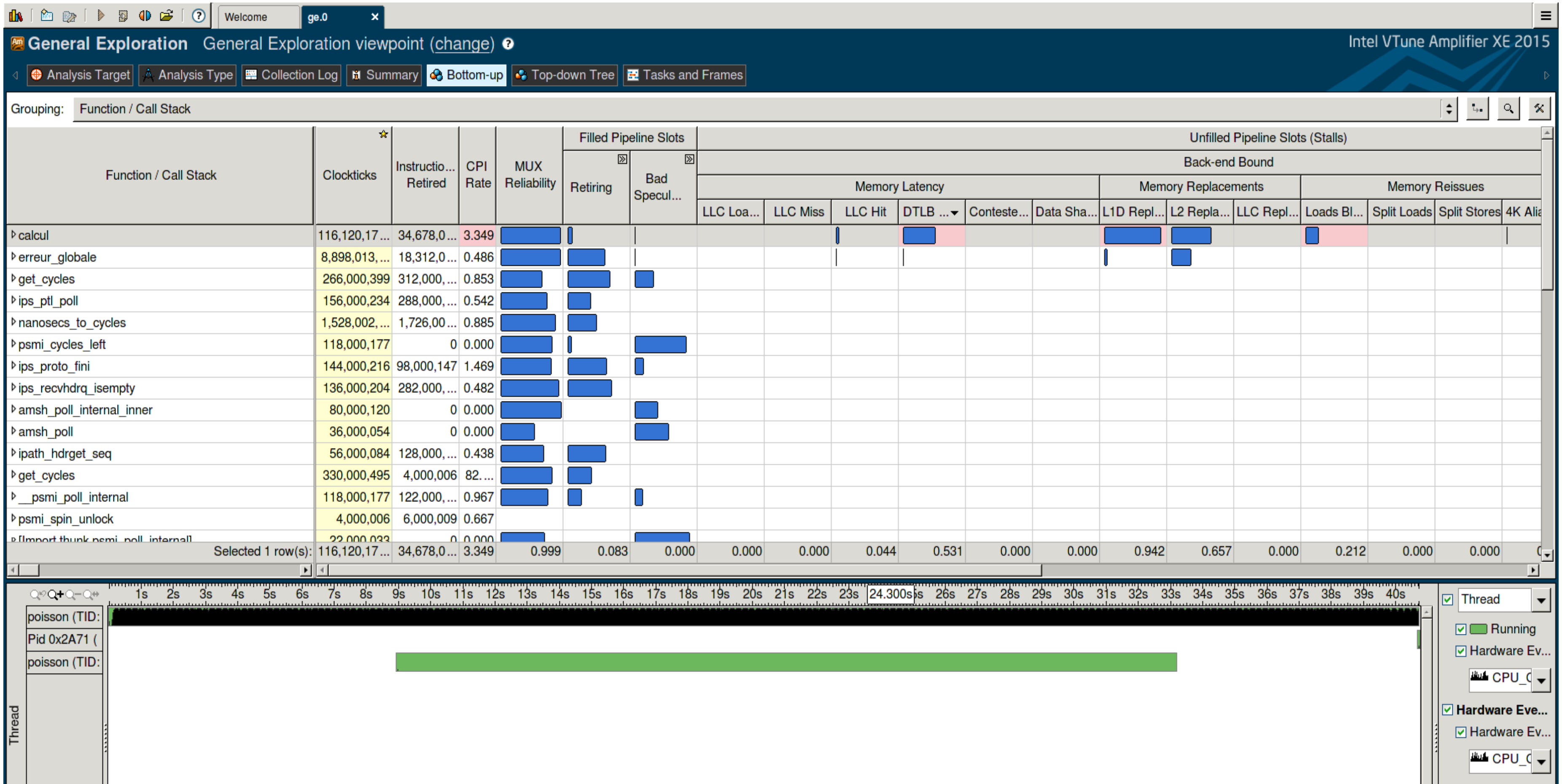
See Also

Reference for Performance Metrics

Prev 4K Aliasing

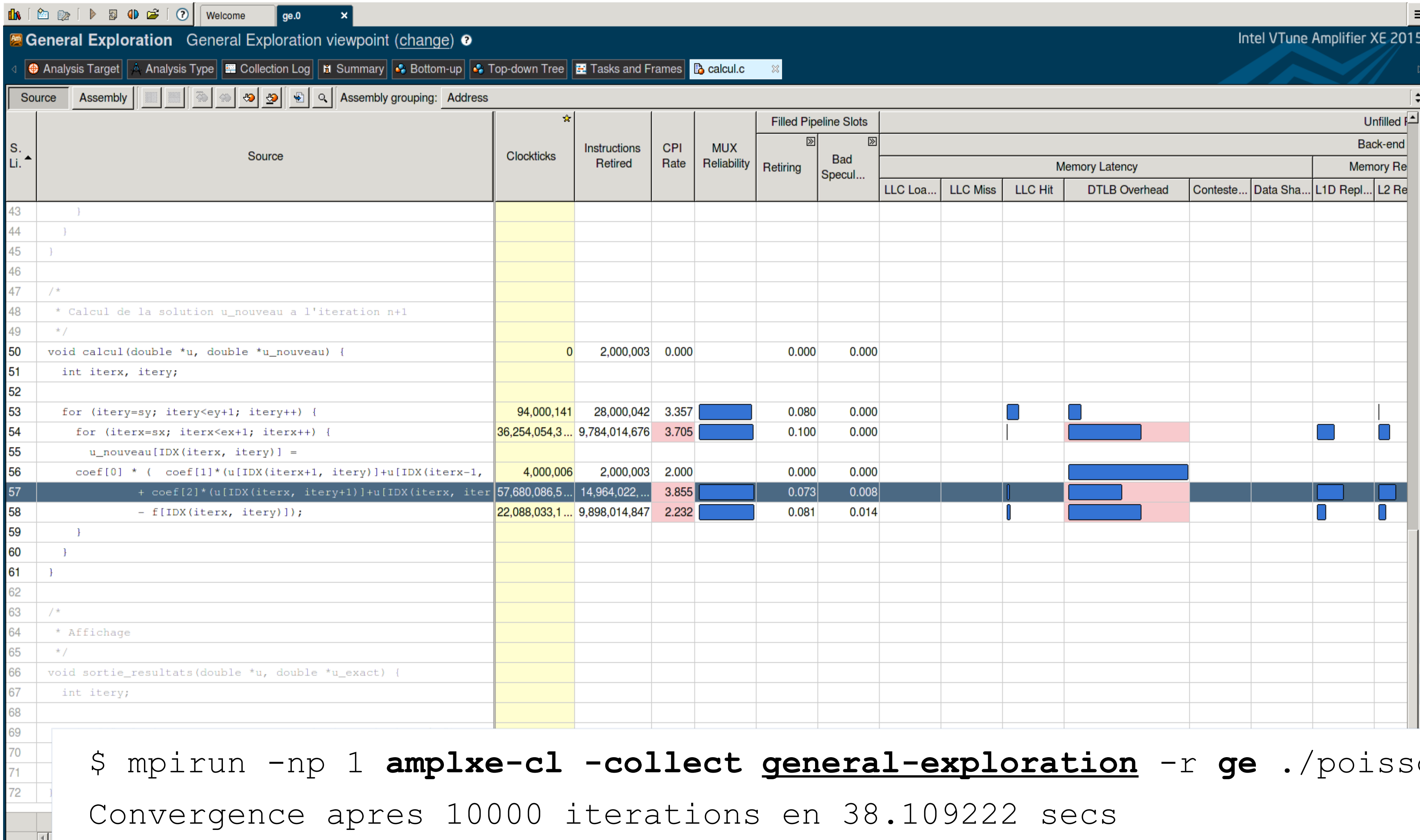
Next FB Full

Advanced Metrics : Flat profile




```
$ mpirun -np 1 amplxe-cl -collect general-exploration -r ge ./poisson
Convergence apres 10000 iterations en 38.109222 secs
$ amplxe-gui ge.0 &
```

Advanced Metrics : Per line profile




```
$ mpirun -np 1 amplxe-cl -collect general-exploration -r ge ./poisson
Convergence apres 10000 iterations en 38.109222 secs
$ amplxe-gui ge.0 &
```

Sequential optimisations

- Hotspot #1 :
- Hotspot #2 :
- Can we go further ?
  - Hotspot #3
  - And further ?

Sequential optimisations

- Hotspot #1 :
  - Stencil : DTLB
    - → Invert loops in **calcul**
- Hotspot #2 :
  - Convergence criterion : Vectorisable
    - → delete **#pragma -novector**
- Can we go further ?
  - Hotspot #3 : Back on stencil
    - – using **-no-vec** compiler option : no impact on **calcul**
    - – stencil vectorisable
      - → add **#pragma simd** on the internal loop
  - Does it worth to call **erreur_globale** for each iteration ?

Addtionnal informations : MPI Context

- Hotspots :

```
$ mpirun -np 2 amplxe-cl -collect hotspots -r pmpi ./poisson
$ ls pmpi.*/*.amplxe
pmpi.0/pmpi.0.amplxe
pmpi.1/pmpi.1.amplxe
```

Per MPI processus profile

- Advanced metrics through a dedicated driver :

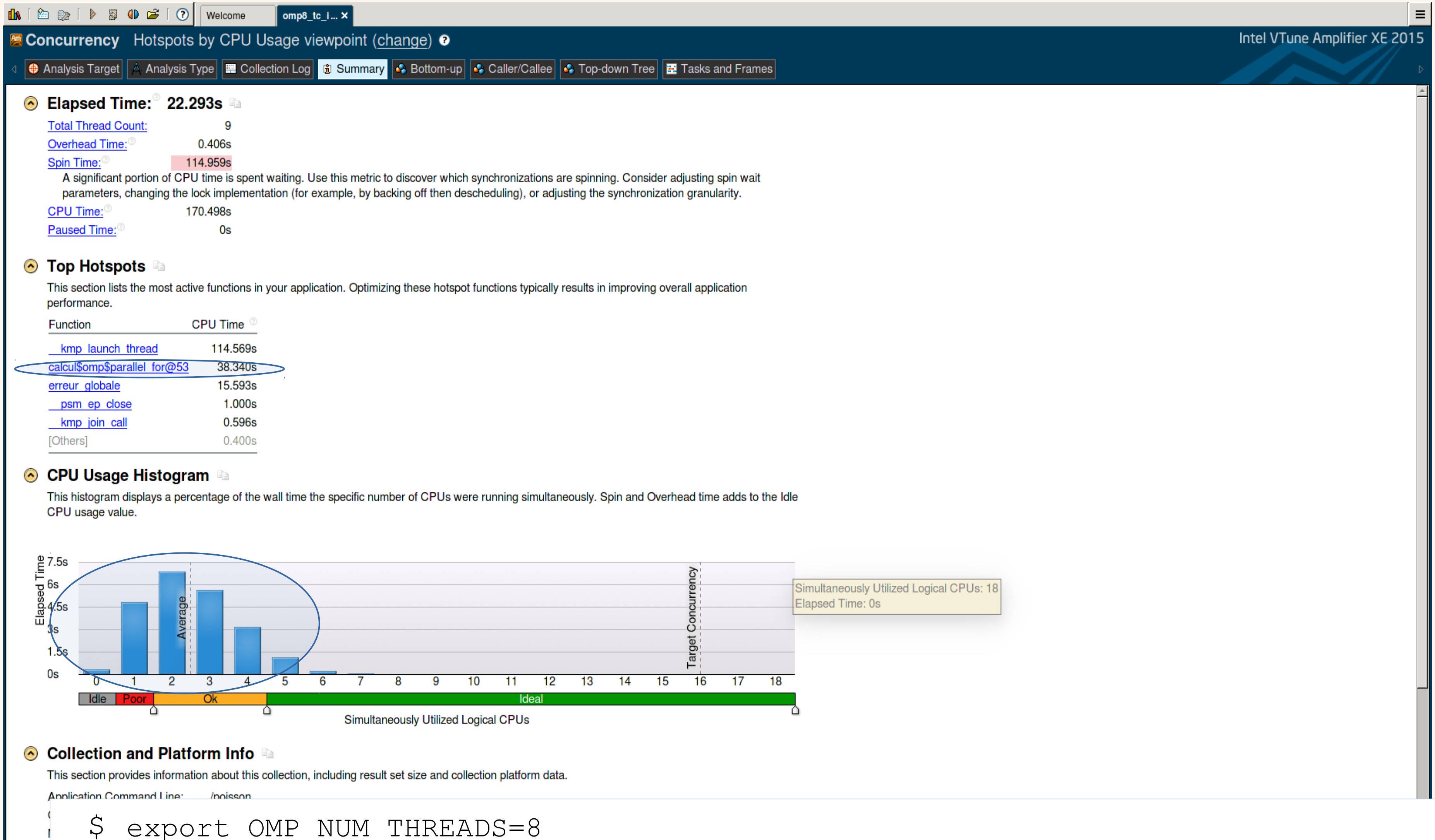
```
$ mpirun -np 2 amplxe-cl -collect general-exploration -r gempi ./poisson
amplxe: Error: PMU resource(s) currently being used by another profiling
  tool or process.
  amplxe: Collection failed.
  amplxe: Internal Error
```

A single collect per CPU

- MPMD like mode :

```
$ mpirun -np 1 amplxe-cl -collect general-exploration -r gempi ./poisson
    : -np 1 ./poisson
```

Addtionnal informations : Thread concurrency




```
$ export OMP_NUM_THREADS=8
$ mpirun -bind-to-socket -np 1 amplxe-cl -collect concurrency -r omp ./poisson
```

Addtionnal informations : Memory counters

Welcome | snb_line.0

Memory Access Hardware Event Counts viewpoint (change)

Intel VTune Amplifier XE 2015

Analysis Target | Analysis Type | Collection Log | Summary | PMU Events | Caller/Callee | Top-down Tree | Tasks and Frames

Elapsed Time: 47.620s

CPU Time: 47.251s
Paused Time: 0s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample
CPU_CLK_UNHALTED.REF_TSC	122,852,184,278	61,426	2000003
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	0	0	20011
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	0	0	20011
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	0	0	20011
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	334,810,044	1,116	100003
MEM_LOAD_UOPS_RETIRED.HIT_LFB_PS	16,260,187,791	54,199	100003
MEM_LOAD_UOPS_RETIRED.L1_HIT_PS	109,656,164,484	18,276	2000003
MEM_LOAD_UOPS_RETIRED.L2_HIT_PS	257,707,731	859	100003
MEM_LOAD_UOPS_RETIRED.LLC_MISS	0	0	100007
MEM_UOPS_RETIRED.ALL_LOADS_PS	127,062,190,593	21,177	2000003
MEM_UOPS_RETIRED.ALL_STORES_PS	19,668,029,502	3,278	2000003

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

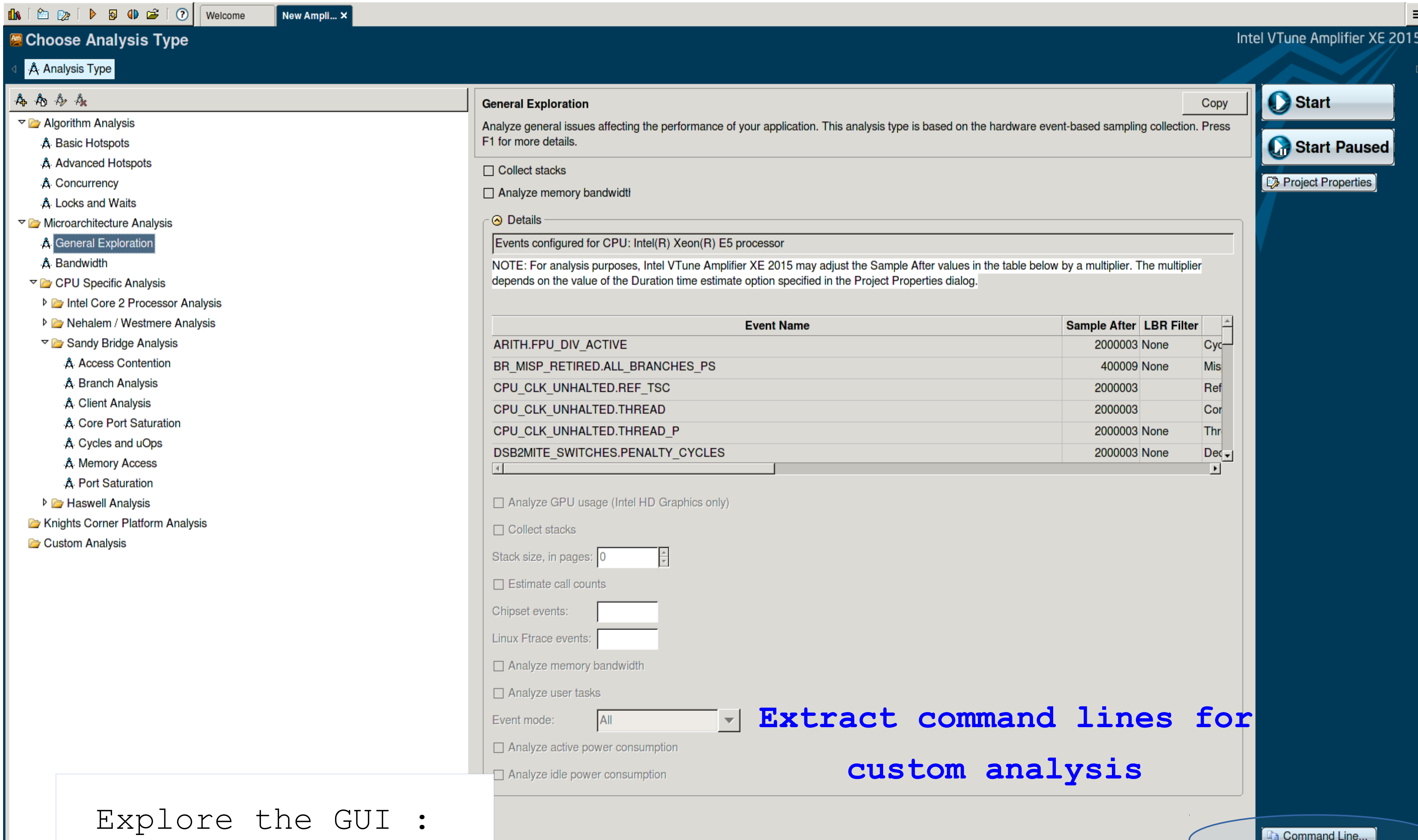
Application Command Line: ./poisson
User Name: jderouillat
Operating System: 2.6.32-431.el6.x86_64 CentOS release 6.5 (Final)
MPI Process Rank: 0
Computer Name: poincare005-adm
Result Size: 28 MB
Collection start time: 14:17:01 09/06/2016 UTC
Collection stop time: 14:17:48 09/06/2016 UTC

CPU

Name: Intel(R) Xeon(R) E5 processor
Frequency: 2,6 GHz
Logical CPU Count: 16

```
$ mpirun -bind-to-socket -np 1 amplxe-cl -collect snb-memory-access -r mem ./poissor
```

And more ...



Extract command lines for custom analysis

Explore the GUI :

$ amplxe-gui &

Scorep, Scalasca, Vampir

Scorep : instrumentation, profiling and tracing

- Scorep : Scalable Performance Measurement Infrastructure for Parallel Codes

- Objectives
  - Automatic instrumentation at compilation
  - Generation of call-path profiles and event traces
  - Recording time, number of visits, exchanged data volumes, hardware counters, ...
  - A wide range of associated softwares for results analyse and visualization (Scalasca, Cube, Vampir, TAU, ...)
  - Supports MPI, OpenMP, basic CUDA.

- Pros & Cons

  + Open Source, in continuous development, well supported

  + Scalable, portable on most HPC platforms

  + Compatible with the main compilers

  - Introduces an overhead in time and memory, especially for the trace collection. (classic tradeoff of the instrumentation)

  - Some open issues and remaining case-specific bugs (bur refer to advantage 1)

Scorep exercise : instrumentation and first profile

- Environment - on Poincare :

  ```
  module use /gpfslocal/pub/vihps/UNITE/local
  module load UNITE
  module load scorep/2.0.2-intel13-openmpi163 scalasca/2.3.1-intel13-openmpi163
  ```

- Instrumentation : by prefixing the compilation and linking commands

  ```
  Ex: mpif90 -> scorep mpif90
      gcc -> scorep gcc
  ```

- Profile generation : automatic by running a normal execution
  - → Generates a folder scorep-yyyymmdd_hhmm_#### containing a profile.cubex file

- First analysis : using scorep-score command :

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

  Gives a summary analysis with the main informations

- Second analysis : usins Scalasca analysis and Cube interface

```
scalasca --examine scorep-########/
```

opens detailed results in Cube graphical interface

Scorep-score : Summary Analysis Result Scoring

```
-bash-4.1$ scorep-score scorep-20160613_1739_7420448901032141/profile.cubex

Estimated aggregate size of event trace:                    6GB
Estimated requirements for largest trace buffer (max_buf): 376MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):        378MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=378MB 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 393,966,035 116,461,281 1968.70   100.0          16.90  ALL
         MPI 328,456,523  72,788,288 1129.39    57.4          15.52  MPI
         COM  43,673,016  29,115,344  271.79    13.8           9.34  COM
         USR  21,836,496  14,557,649  567.51    28.8          38.98  USR
```

Region/callpath classification :

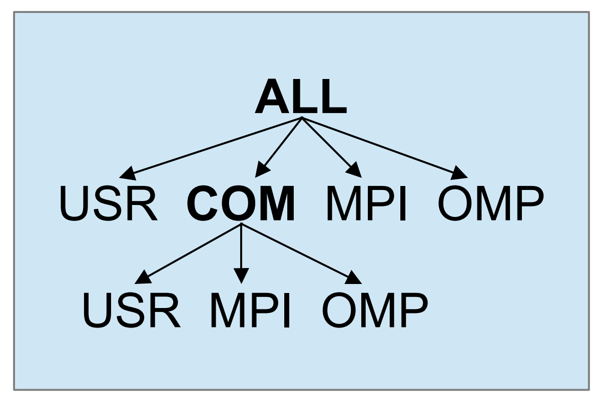
MPI : pure MPI library functions

OMP : pure OpenMP functions/refions

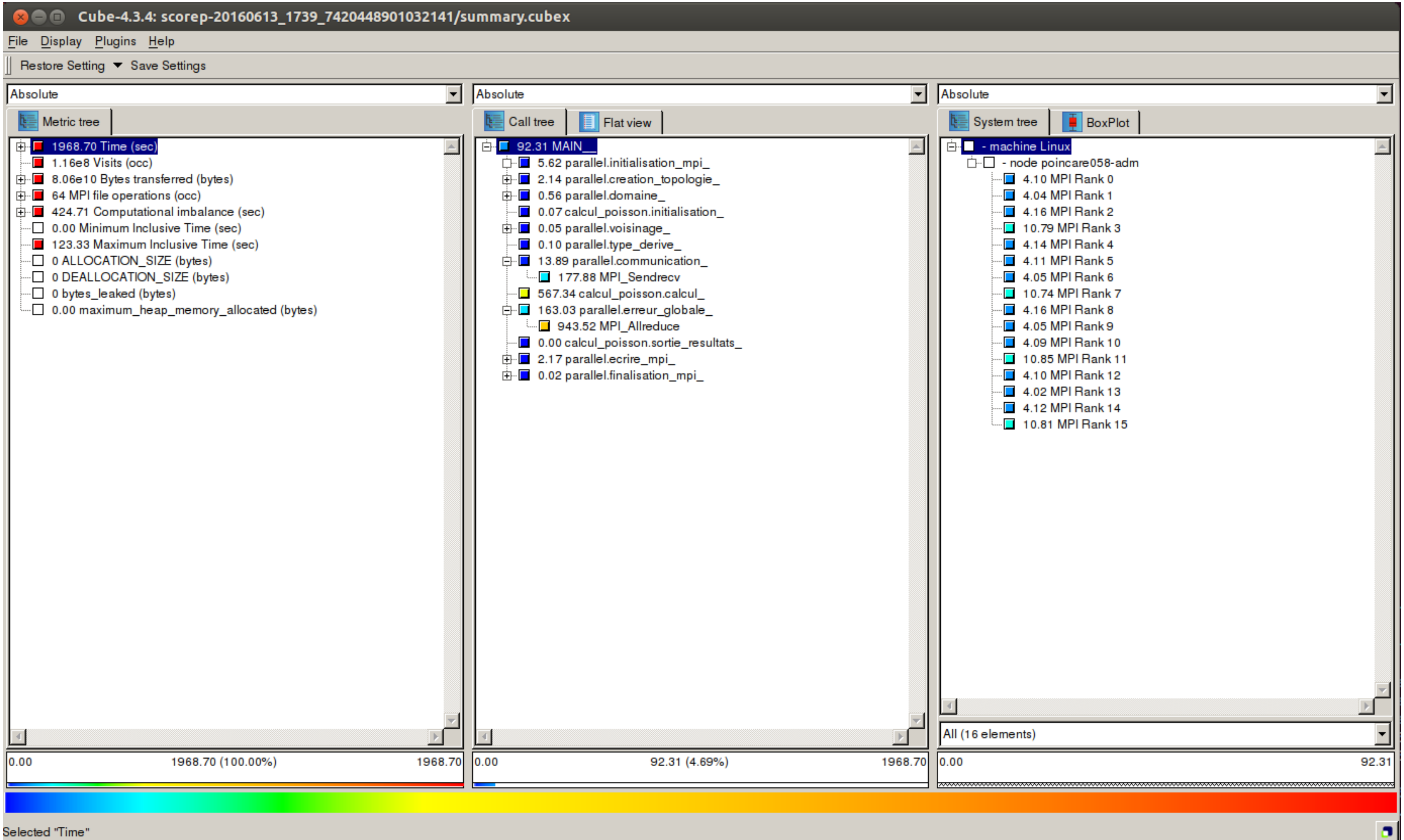
USR : user-level source local computation

COM : 'combined' USR + OMP/MPI

ANY/ALL : aggregation of all regions



Cube interface (through scalasca --examine)



Scorep : Trace Collection (1)

- Tracing give more insight into the code behavior
- It often generates too much collected data, resulting in huge time and memory overheads
- First step : estimate this cost
  - The scorep scoring gives an estimation of the needed memory buffer and trace files size.
  - -r option shows all regions
- External example : code X running on Curie on 288 processes

```
me@curie90 $ scorep-score -r scorep-20150701_1639_1289782540682648/profile.cube
Estimated aggregate size of event trace:                   15TB
Estimated requirements for largest trace buffer (max_buf): 64GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       64GB
(hint: When tracing set SCOREP_TOTAL_MEMORY=64GB 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 68,445,407,406 684,452,054,099 308648.07   100.0           0.45  ALL
         USR 68,436,752,472 684,392,045,176 118937.03    38.5           0.17  USR
         MPI      8,701,920      59,381,659  76915.79    24.9        1295.28  MPI
         COM         52,272         627,264 112795.25    36.5      179821.02  COM

         USR 68,344,181,664 683,280,328,752 117688.27    38.1           0.17  mod_kernel.kernel_
         USR     92,665,344   1,111,120,000    230.54     0.1           0.21  mod_intgr.binsearch_
         MPI      2,463,231       8,195,263     26.98     0.0           3.29  MPI_Isend
         MPI      2,325,231       8,195,268     18.44     0.0           2.25  MPI_Irecv
         MPI      2,197,704      26,271,152      4.55     0.0           0.17  MPI_Cart_rank
         MPI      1,452,744      14,908,466  40192.02    13.0        2695.92  MPI_Waitall
         MPI        407,360       1,543,680  30548.99     9.9       19789.72  MPI_Sendrecv
         MPI         71,750           1,435      0.22     0.0         152.44  MPI_Ssend
         USR         46,128         430,000      0.21     0.0           0.48  mod_intgr.sumrk3_
            (...)
```

Tracing requires 15TB on disk
64GB buffer on each process

Scorep : Trace Collection (2)

Observations on the overhead :

- Some USR regions, like kernek functions, called too many times, are problematic for scorep, generating too much overhead due to their instrumentation
- Filtering them out of the collection can improve the quality of the profile collection, and is necessary for the trace collection
- Other ways exist to reduce requirements. Ex :reducing the number of iteration

Filtering : by using a filter file

- Scorep scoring with -f option allows a new estimation without re-running

Example : code X

```
me@curie90 $ cat scorep.filt
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    mod_kernel.kernel*
    mod_intgr.binsear*
SCOREP_REGION_NAMES_END

me@curie90 $ scorep-score -f scorep.filt scorep-20150701_1639_12897825   48/profile.cubex

Estimated aggregate size of event trace:                   2176MB
Estimated requirements for largest trace buffer (max_buf): 9MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       11MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=11MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)
      (...)
```

Trace files reduced to 2 GB

Buffer needed on each process to avoid intermediate flushes < 15M

Scorep : Trace Collection (3) - Exercise

- Run the trace collection : you need to
  - Define the following variables :
    - SCAN_ANALYZE_OPTS=--time-correct
    - SCOREP_FILTERING_FILE=#the name of the filter file if needed#
    - SCOREP_TOTAL_MEMORY=#the required memory estimated by scorep-score#
  - Prefix the execution line with 'scalasca --analyse -t'

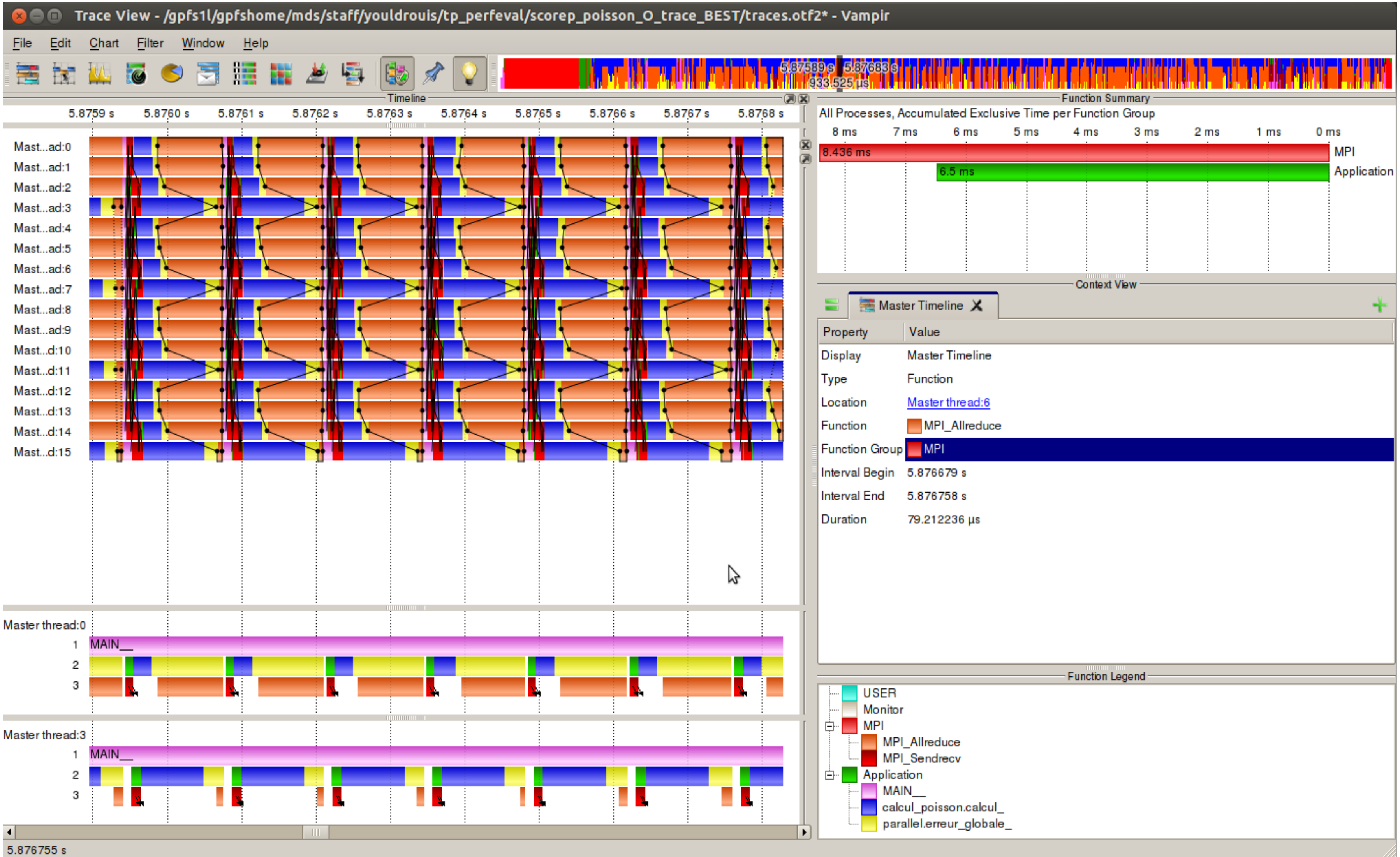
- First analysis : on Cube interface (scalasca --examine scalasca_trace_folder)

-> additional metrics such as Late Sender / Late Receiver cases quantification

- Second analysis : visualization with Vampir
  - Load vampir module
  - Open traces.otf2 file in the Vampir interface
      - » vampir scorep_poisson_O_trace/traces.otf2 &
  - Zoom in on the timeline, explore
  - Color different regions for better visualization
  - Check the 'Process timeline', the 'Communication matrix', and other charts available in the charts menu

— Got some idea about the inbalance reasons?

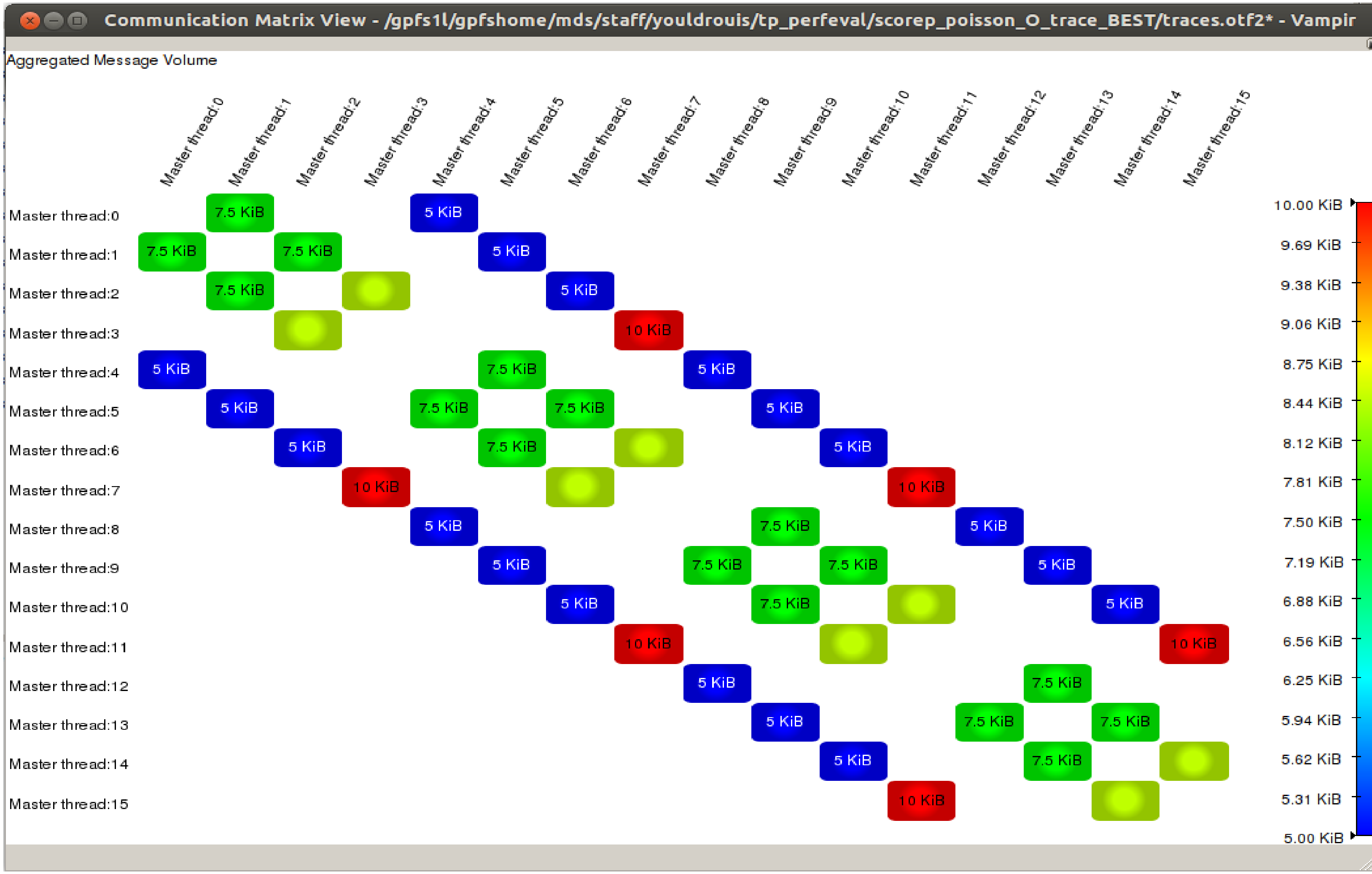
Trace visualisation with Vampir



Running the trace collection needs :

- Prefixing the execution command with « scalasca -t »

Trace visualisation with Vampir (2)



Running the trace collection needs :

- Prefixing the execution command with « scalasca -t »

PAPI : Performance Application Programming Interface

- Processors are equiped with a big number of counters
- PAPI is a portable library that gives easy access to these counters

- PAPI can be used as :
  - libraries for advanced use with C, Fortran, ...
  - through end-user perf tools providing easy collection and visualization

- Can provide insight into :
  - Memory, L1, L2, L3 Cache and tlb behaviors
  - Cycles, Instructions, Pipeline stalls, Branch behaviors
  - Floating point operations, ...

- Downsides :
  - Only 4 to 6 counters can be accessed at the same time
  - Simultaneous access to 2 available counters can be problematic (check with papi_command_line)
  - The availability of each counter depends on the hardware (papi_avail)
  - Some hardware counters are erroneous (ex : FP ops on Sandy Bridge)
  - Increases size and buffer requirement of the profile or trace collection

Importance of data access performance

- Data access

Number of cycles per data access depending on its location

Location	Access
L1	~4 cycles
L2	~10 cycles
L3	~40-65 cycles
memory	60ns (150-200 cycles)

Source http://software.intel.com/en-us/forums/topic/287236

- Bad data access has a huge effect on the performance
- Causes instruction pipelines stalls, thus reduction of completed instructions per cycle.

PAPI counters collection with Scorep - exercise

- Load papi module papi/5.3.0_intel13
- Choose set of PAPI counters to collect
  - Browse available preset counters with papi_avail
  - Check compatibility of a chosen set with papi_command_line
    - Ex : papi_command_line PAPI_L1_DCM PAPI_L2_DCM PAPI_TOT_CYC

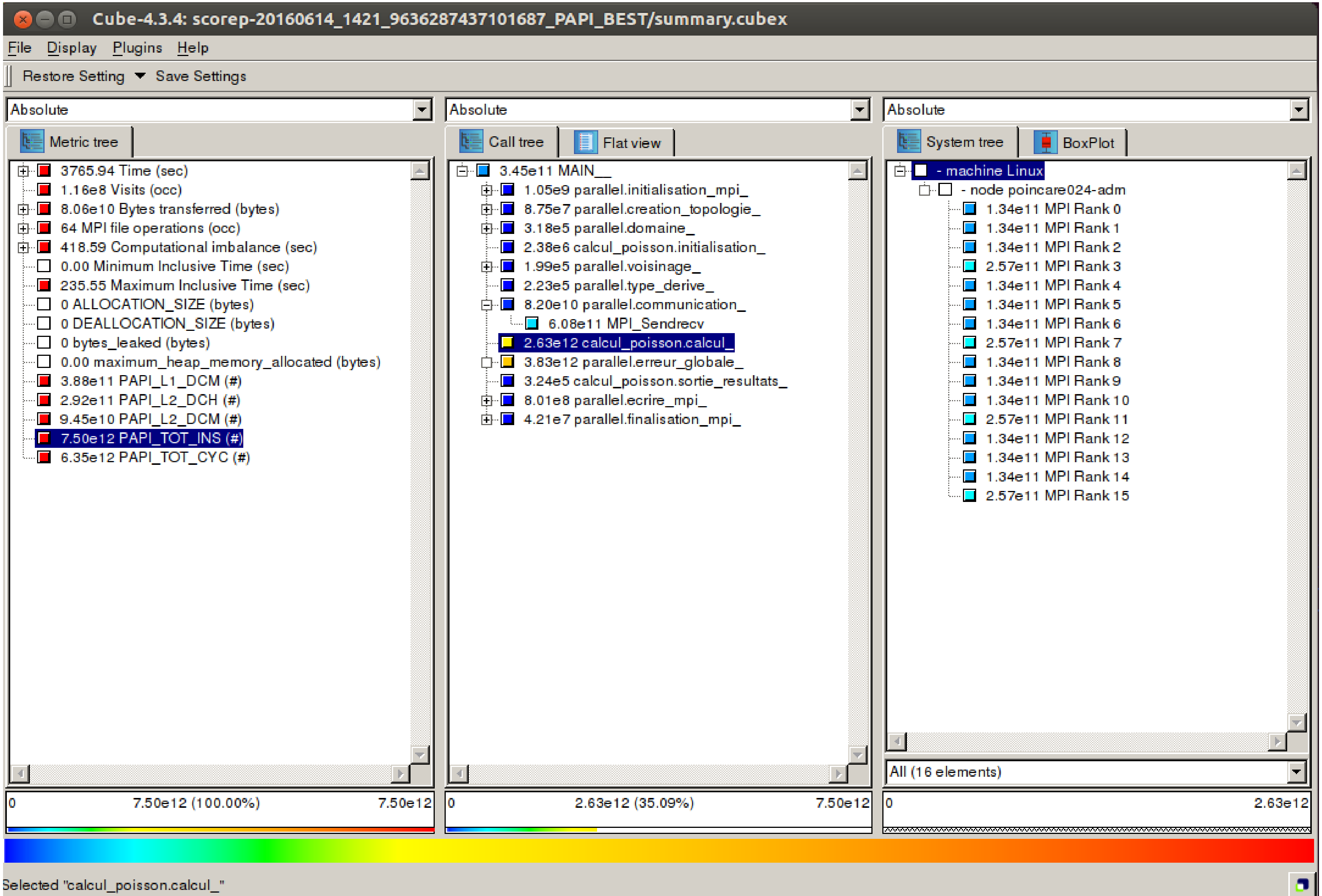
- Run profile collection :
  - Define the SCOREP_METRIC_PAPI variable prior to the execution line:
    - Ex : export SCOREP_METRIC_PAPI=list of counters separated with commas

- Open the profile in Cube (scalasca --examine)

- Run trace collection :
  - Check first the requirements with scorep-score **-c** number_of_papi_metrics
  - Adapt your run to the requirements
  - Define SCOREP_METRIC_PAPI

- Open the trace in Vampir. Visualize the perf counters in the 'Counter Data Timeline' and the 'Performance Radar' charts

PAPI – Scorep/Scalasca results



PAPI metrics in Scalasca trace – Visualization with Vampir

Memory

Memory instrumentation

- Ecosystem light :
  - Initiative @ :
    - a memory trace included in EZTrace
    - Exascale Lab : MALT (a MALloc Tracker)
    - MdlS : LIBMTM (Modeling and Tracing Memory Library)
  - *Memory bandwidth estimation in Allinea Perf Reports*

- Linux function : getrusage

```
#include <sys/resource.h>
{
    struct rusage r_usage;
    getrusage(RUSAGE_SELF,&r_usage);
    // r_usage.ru_maxrss
}
```

Max value only

- Read /proc/pid/status

```
$ grep VmRss /proc/pid/status
```

Memory instrumentation

- Edit make_inc to enable -D_INST_MEM option, then recompile

```
$ mpirun -np 1 ./poisson 1
…
Taille du domaine : ntx=480 nty=400
…
Start --- Rang 0 - Memory usage: 24448 Kbytes
Alloc --- Rang 0 - Memory usage: 30512 Kbytes
…
Convergence apres 1 iterations en 0.003954 secs
Clean --- Rang 0 - Memory usage: 24736 Kbytes
```

- Grid footprint :
  - 480 x 400 x 8 = 1500 Kbytes / grid
  - x 4 grids

Memory instrumentation

- Edit make_inc to enable -D_INST_MEM option, then recompile

```
$ mpirun -np 1 ./poisson 1
…
Taille du domaine : ntx=480 nty=400
…
Start --- Rang 0 - Memory usage: 24448 Kbytes
Alloc --- Rang 0 - Memory usage: 30512 Kbytes
…
Convergence apres 1 iterations en 0.003954 secs
Clean --- Rang 0 - Memory usage: 24736 Kbytes
```

- **Memory leaks**
- **MPI ? Load imbalance**