Tools and techniques for optimization and debugging

Fabio Affinito
October 2015

Profiling

Why?

Parallel or serial codes are usually quite complex and it is difficult to understand what is the most time consuming part.

Profiling is a prerequisite to optimization.

You don't want to spend time to optimize a function where usually your application spends 0.0001% of the runtime!

Which?

There are a lot of different tools. Some of them are suitable for serial applications (they identify the most compute-intensive parts), some other for parallel computations (they identify conflicts, parallel bottlenecks, load unbalance, etc.).

There are free and proprietary tools. You can choice which to use depending on their availability (usually computer facilities offer licenses for many proprietary tools).

Time

Time is a CL tool available on every Linux/UNIX platform;

It provides time of execution and some other useful information;

It is extremely simple, but it can provide as well some insight on your system exploitation.

Time

```
time ./a.out

9.29user 6.19system 0:15.52elapsed 99%CPU (0avgtext+0avgdata 18753424maxresident)k
0inputs+0outputs (0major+78809minor)pagefaults 0swaps
```

User time: time spent by the CPU for the execution of the code
System time: time spent by the CPU for system calls
Elapsed time: time actually spent for the execution of your code
The percentuage of CPU used by the process.
Number of page faults
Number of swaps

Time

```
time ./a.out

9.29user 6.19system 0:15.52elapsed 99%CPU (0avgtext+0avgdata 18753424maxresident)k
0inputs+0outputs (0major+78809minor)pagefaults 0swaps
```

Looking at this example we can notice:
-User time is close to the system time
-CPU is used 99%
-There's no I/O
-No page faults
-18753424 is the total data area used (actually this is buggy, should 1/4)
-System time + CPU time = Elapsed time

Time

```
time ./a.out

9.29user 6.19system 0:15.52elapsed 99%CPU (0avgtext+0avgdata 18753424maxresident)k
0inputs+0outputs (0major+78809minor)pagefaults 0swaps
```

We rerun the same code, but reducing the number of alloc/dealloc operations (that require the execution of syscalls)

```
time ./a.out

2.28user 0.38system 0:02.67elapsed 99%CPU (0avgtext+0avgdata 9378352maxresident)k
0inputs+0outputs (0major+3153minor)pagefaults 0swaps
```

Time

If our application is multi-thread (i.e. parallel), we would expect that the CPU time will be a multiple of the elapsed time.

```
time ./a.out

12973.38user 1915.82system 20:55.80elapsed 1185%CPU (0avgtext+0avgdata
2597648maxresident)k
19608inputs+10649880outputs (147major+223489935minor)pagefaults 0swaps
```

Top

top provides a dynamic monitoring of every process running on a given machine (or node);

```
top - 20:32:08 up 170 days,  6:38,  0 users,  load average: 10.91, 10.98, 10.27
Tasks: 337 total,   9 running, 328 sleeping,   0 stopped,   0 zombie
%Cpu(s): 40.8 us, 26.5 sy,  0.0 ni, 32.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  13174488+total, 49884916 used, 81859968 free,     2048 buffers
KiB Swap: 32767996 total,    13772 used, 32754224 free.  8736308 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 8638 mcazzani  20   0 5822416 248532   5800 R 399.6  0.2   1572:21 l502.exe
28689 aesposit  20   0 33.839g 0.033t   1260 R 100.0 26.9 102:02.43 Tech_seq
 6405 mbruschi  20   0 15.347g 185016  27164 R 100.1  0.1  31:12.84 ridft_mpi
 6406 mbruschi  20   0 15.344g 184700  27396 R 100.1  0.1  31:11.91 ridft_mpi
 6407 mbruschi  20   0 15.339g 175916  27332 R  99.7  0.1  31:03.66 ridft_mpi
 6408 mbruschi  20   0 15.347g 185220  27340 R  99.7  0.1  31:11.14 ridft_mpi
 6409 mbruschi  20   0 15.400g 175600  27204 R  99.7  0.1  31:04.22 ridft_mpi
 6404 mbruschi  20   0 15.712g 1.197g  29168 R  76.1  1.0  29:38.73 ridft_mpi
10627 faffinit  20   0  123804   1848   1156 R   0.7  0.0   0:00.05 top
 1769 root      39  19       0      0      0 S   0.3  0.0   3031:42 kipmi0
 6385 mbruschi  20   0   27408   1172    980 S   0.3  0.0   0:01.53 mpid
 9372 root      20   0       0      0      0 S   0.3  0.0   0:00.74 kworker/u34:0
10527 root      20   0       0      0      0 S   0.3  0.0   0:00.11 kworker/u34:1
11028 root       0 -20 9788.5m 1.175g 108280 S   0.3  0.9 282:44.52 mmfsd
    1 root      20   0   55248   4248   2032 S   0.0  0.0   6:15.78 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:04.41 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   3:27.63 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
```

gprof

gprof is an open source profiler provided by the GNU toolchain

The analysis provided by gprof is more deep with respect to the time command:

- it is at the function/subroutine grain level
- it has a very low "impact" on the real performances
- it provides information about the graph of dependencies inside our code

gprof

gprof makes use of both “sampling” and “instrumentation”

sampling = it checks in fixed intervals the time execution and advancement of the code

instrumentation = it adds instructions to the original code, in order to track the execution of such parts of code

gprof

To use gprof, you need to compile the program with the –pg flag

Then you run your code normally and at the end you check the measures with

```
gcc mycode.c -pg -o myexe
./myexe

gprof myexe
```

gprof

If the execution ends without problem a gmon.out file is generated (and eventually overwritten).

Flat profile

gprof can produce a flat profile. Let's see a simple example starting from a code:

We would expect that the function b is 4 times more long than function a

```
#include <stdio.h>
int a(void) {
  int i=0,g=0;
  while(i++<100000){
    g+=i;
  } return g;
}
int b(void) {
  int i=0,g=0;
  while(i++<400000){
    g+=i;
  }return g;
}
int main(int argc, char** argv){
  int iterations;
  if (argc != 2){
    printf("Usage %s <No of
Iterations>\n", argv[0]);
  exit(-1);
  }
  else
  iterations  = atoi(argv[1]);
  printf("No  of  iterations  =
%d\n",  iterations);
  while(iterations--){
    a();
    b();
  }
}
```

Flat profile

```
/usr/bin/time ./Main\ example.exe 10000
No of iterations = 10000
3.22user 0.00system 0:03.23elapsed 99%CPU (0avgtext+0avgdata 1760maxresident)k
0inputs+0outputs (0major+131minor)pagefaults 0swaps
gcc -O Main\ example.c -o Main\ example_gprof.exe -pg
[lanucara@louis ~]$ /usr/bin/time ./Main\ example_gprof.exe 10000
No of iterations = 10000
3.33user 0.00system 0:03.34elapsed 99%CPU (0avgtext+0avgdata 2064maxresident)k
0inputs+8outputs (0major+150minor)pagefaults 0swaps
gprof ./Main\ example_gprof.exe > Main\ example.gprof
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative self             self    total
time    seconds   seconds calls us/call us/call name
81.43   2.73      2.73   10000 272.78  272.78  b
19.60   3.38      0.66   10000  65.67   65.67  a
```

Flat profile

- time in %
- cumulative time spent by function and ancestors (in sec)
- time spent by the function (in sec)
- number of function calls
- average time for every function call (us)
- average time cumulative per function call and children functions
- function name

```
Flat profile:
Each sample counts as 0.01 seconds.
 %   cumulative self          self    total
time    seconds   seconds calls us/call us/call name
81.43    2.73      2.73   10000 272.78  272.78  b
19.60    3.38      0.66   10000  65.67   65.67  a
```

Flat profile

We want to introduce a new function:

and we put it inside b()

```
int cinsideb(int d) {
  {
  }
  return d;
}
```

```
int b(void) {
  int i=0,g=0;
  while(i++<400000){
    g+=cinsideb(i);
  }return g;
}
```

Flat profile

Let's check the new flat profile

```
Flat profile:
Each sample counts as 0.01 seconds.
  %  cumulative  self            self   total
time  seconds    seconds calls us/call us/call name
44.72  3.28       3.28   10000 327.78  604.55  b
37.76  6.05       2.77 4000000000 0.00 0.00    cinsideb
18.53  7.40       1.36   10000 135.86  135.86  a
```

Tree profile

In addition to the flat profile, the tree profile provides information about the relation caller/callee.

Tree profile

```
                     Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.14% of 7.40 seconds

index % time    self  children    called     name
                                                                    <spontaneous>
[1]    100.0    0.00    7.40                 main [1]
                      3.28    2.77   10000/10000        b [2]
                      1.36    0.00   10000/10000        a [4]
-----------------------------------------------
                      3.28    2.77   10000/10000        main [1]
[2]     81.7     3.28    2.77   10000           b [2]
                      2.77    0.00 4000000000/4000000000     cinsideb [3]
-----------------------------------------------
                      2.77    0.00 4000000000/4000000000     b [2]
[3]     37.4     2.77    0.00 4000000000          cinsideb [3]
-----------------------------------------------
                      1.36    0.00   10000/10000        main [1]
[4]     18.3     1.36    0.00   10000           a [4]
-----------------------------------------------
...
```

gprof: limitations

- gprof sometimes doesn’t provide data about library functions (cfr. MKL, etc.)
- gprof has a quite coarse granularity: it doesn’t dig into a function (that in some cases can be also very large..)
- sometimes the overhead due to gprof can be very relevant (always compare execution times with and without gprof)
- measured times comparable to the “sampling time” are not reliable

Temporize

Sometimes, it can be necessary to manually insert code in our application in order to measure what is the time really spent by a given function.

There are a lot of ad-hoc functions or language primitives. For example:

- etime(), dtime() for Fortran77
- cputime(), system_clock(), date_and_time() for Fortran90
- clock() for C/C++
- etc

Temporize

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
clock_t time1, time2;
double dub_time;
int main(){
int i,j,k,nn=1000;
double c[nn][nn],a[nn][nn],b[nn][nn];
...
time1 = clock();
for (i = 0; i < nn; i++)
for (k = 0; k < nn; k++)
for (j = 0; j < nn; j ++)
c[i][j] = c[i][j] + a[i][k]*b[k][j];
time2 = clock();
dub_time = (time2 - time1)/(double) CLOCKS_PER_SEC;
printf("Time -----------------> %lf \n", dub_time);
...
return 0;
}
```

Temporize

```
real(8)::a(1000,1000),b(1000,1000),c(1000,1000)
real(8) ::t1,t2
integer :: time_array(8)
a=0;b=0;c=0;n=1000
...
...
call date_and_time(values=time_array)
t1 = 3600.*time_array(5)+60.*time_array(6)+time_array(7)+time_array(8)/1000.
do j = 1,n
do k = 1,n
do i = 1,n
c(i,j) = c(i,j) + a(i,k)*b(k,j)
enddo
enddo
enddo
call date_and_time(values=time_array)
t2 = 3600.*time_array(5)+60.*time_array(6)+time_array(7)+time_array(8)/1000.
write(6,*) t2-t1
...
...
end
```

PAPI

PAPI = Performance Application Programming Interface is a set of function (APIs) designed in order to profile a code at a very fine level.

One of the aim of the PAPI is the portability, i.e. the possibilty of being ran on most actual architectures (x86, GPUs, Intel MIC, etc.)

PAPI can access the hardware counters: special-purpose registers that provide informations about the CPU behavior

PAPI

PAPI provides 2 levels of interface:

- High level interface: a library that provides informations about a given set of events (PAPI Preset Events)

- Low level interface: it provides information more specific about the hardware. It is much more complex and difficult to use.

PAPI events

Most interesting events (among the PAPI Preset Events) are:

- PAPI_TOT_CYC: total number of CPU cycles
- PAPI_TOT_INS: number of completed instructions
- PAPI_FP_INS: number of floating point instructions
- PAPI_L1_DCA: accesses in L1 cache
- PAPI_L1_DCM: cache misses in L1
- PAPI_SR_INS: number of store instructions
- PAPI_TLB_DM: TLB misses
- PAPI_BR_MSP: conditional branches mispredicted

PAPI example

```
#include <stdio.h>
#include <stdlib.h>
#include "papi.h"

#define NUM_EVENTS 2
#define THRESHOLD 10000
#define ERROR_RETURN(retval) { fprintf(stderr, "Error %d %s:line %d: \n",
retval,__FILE__,__LINE__);  exit(retval); }
...
/* stupid codes to be monitored */
void computation_add()
   ....
int main()
{
   int Events[2] = {PAPI_TOT_INS, PAPI_TOT_CYC};
   long long values[NUM_EVENTS];
...
if ( (retval = PAPI_start_counters(Events, NUM_EVENTS)) != PAPI_OK)
      ERROR_RETURN(retval);
   printf("\nCounter Started: \n");
if ( (retval=PAPI_read_counters(values, NUM_EVENTS)) != PAPI_OK)
     ERROR_RETURN(retval);
   printf("Read successfully\n");
   computation_add();
if ( (retval=PAPI_stop_counters(values, NUM_EVENTS)) != PAPI_OK)
     ERROR_RETURN(retval);
   printf("Stop successfully\n");
   printf("The total instructions executed for addition are %lld \n",values[0]);
   printf("The total cycles used are %lld \n", values[1] );
}
```

PAPI high level functions

PAPI also provides a set of useful high level functions:

- PAPI_num_counters : number of available hw counters
- PAPI_flips : floating point instruction rate
- PAPI_flops : floating point operation rate
- PAPI_ipc : instructions per cycle
- PAPI_read_counters : read and reset the counters
- PAPI_start_counters : start counting hw events
- PAPI_stop_counters : stop counters and return the count

SCALASCA

- gprof and PAPI provide information about the serial performance of a given application.
- We can use also gprof in order to profile a parallel application, but the results are often very difficult to understand.
- SCALASCA is a tool developed by F. Wolf and coworkers in the JSC and it is a good tool to check the scalability and efficiency of parallel software, also when going on a large scale.
- Open source and available at www.scalasca.org

SCALASCA

It provides 2 different analysis:

- "Summary" provides a fine level profiling but in an "aggregate" way
- "Tracing" is a profiling more local to a process. It provides much more information but it can be expensive in terms of storage

SCALASCA

Profiling with SCALASCA needs 3 steps:

1) compilation and instrumentation of the code

```
scalasca -instrument mpiifort -openmp mycode.f90 -o myapp.x
```

1) execution

```
scalasca -analyze mpirun -np 1024 ./myapp.x
```

2) analysis

```
scalasca -examine epik_XXXXX
```

SCALASCA

Profiling with SCALASCA needs 3 steps:

1) compilation and instrumentation of the code

```
skin -instrument mpiifort -openmp mycode.f90 -o myapp.x
```

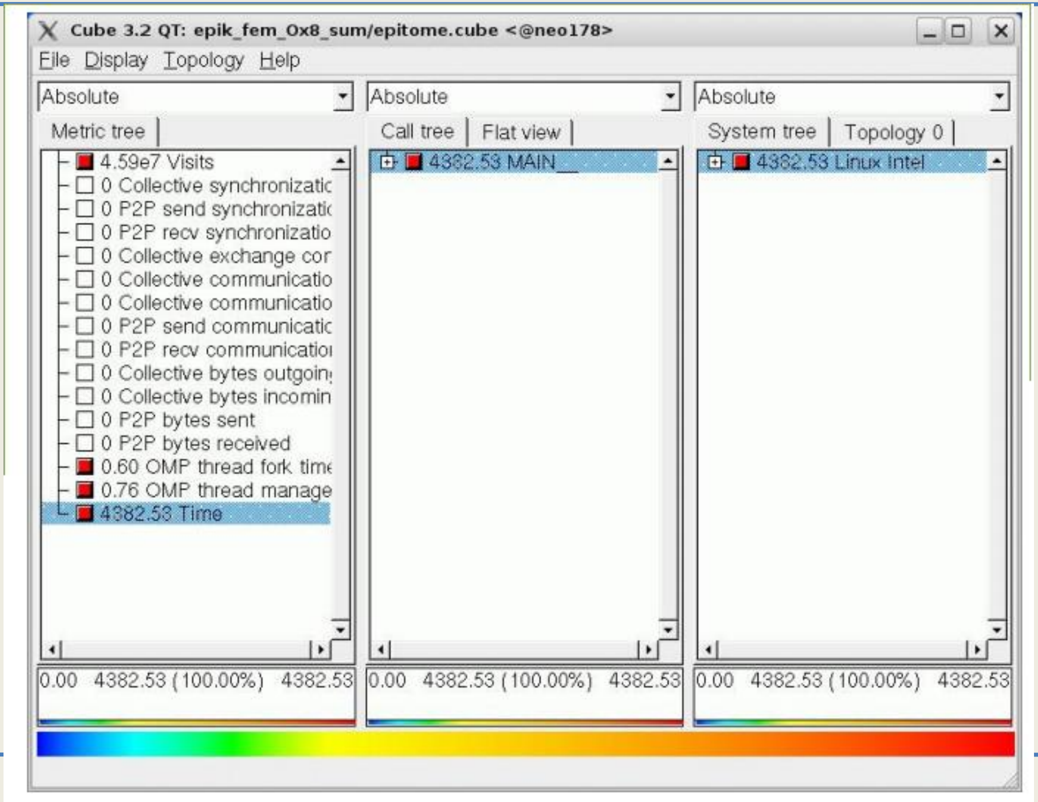
1) execution

```
scan -analyze mpirun -np 1024 ./myapp.x
```

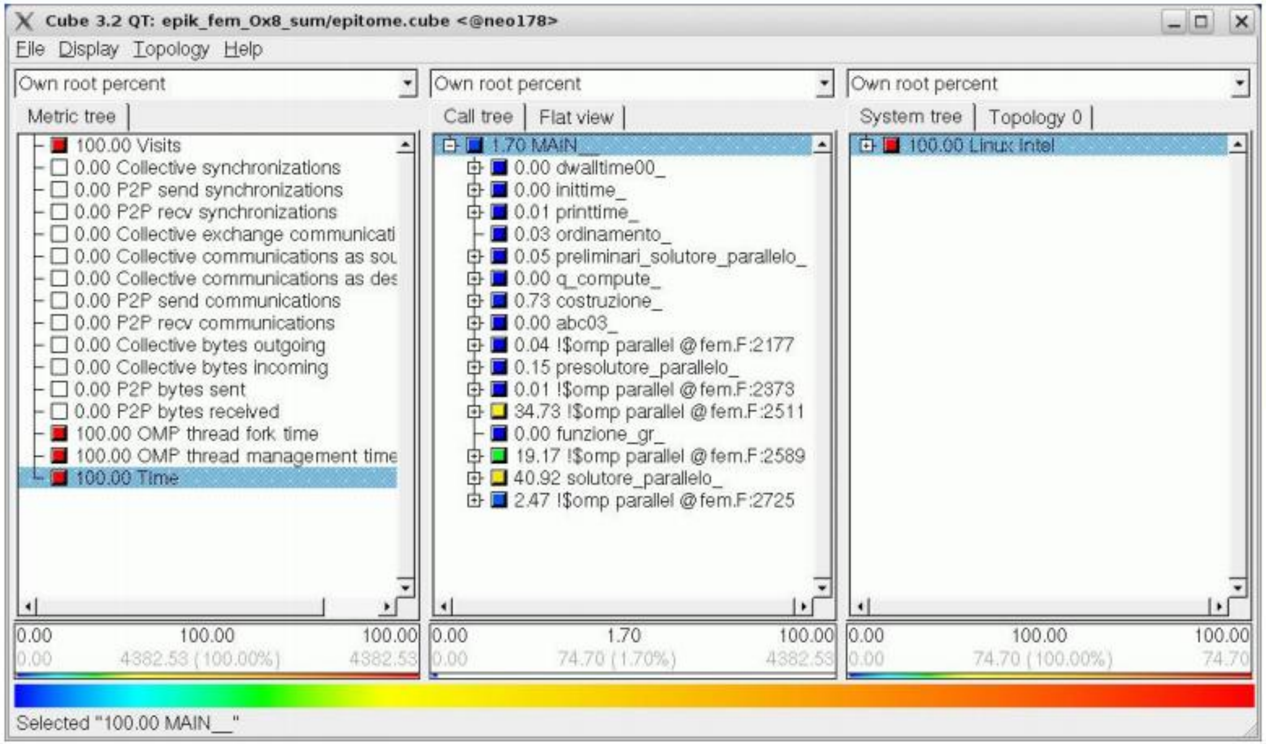
2) analysis

```
square -examine epik_XXXXX
```

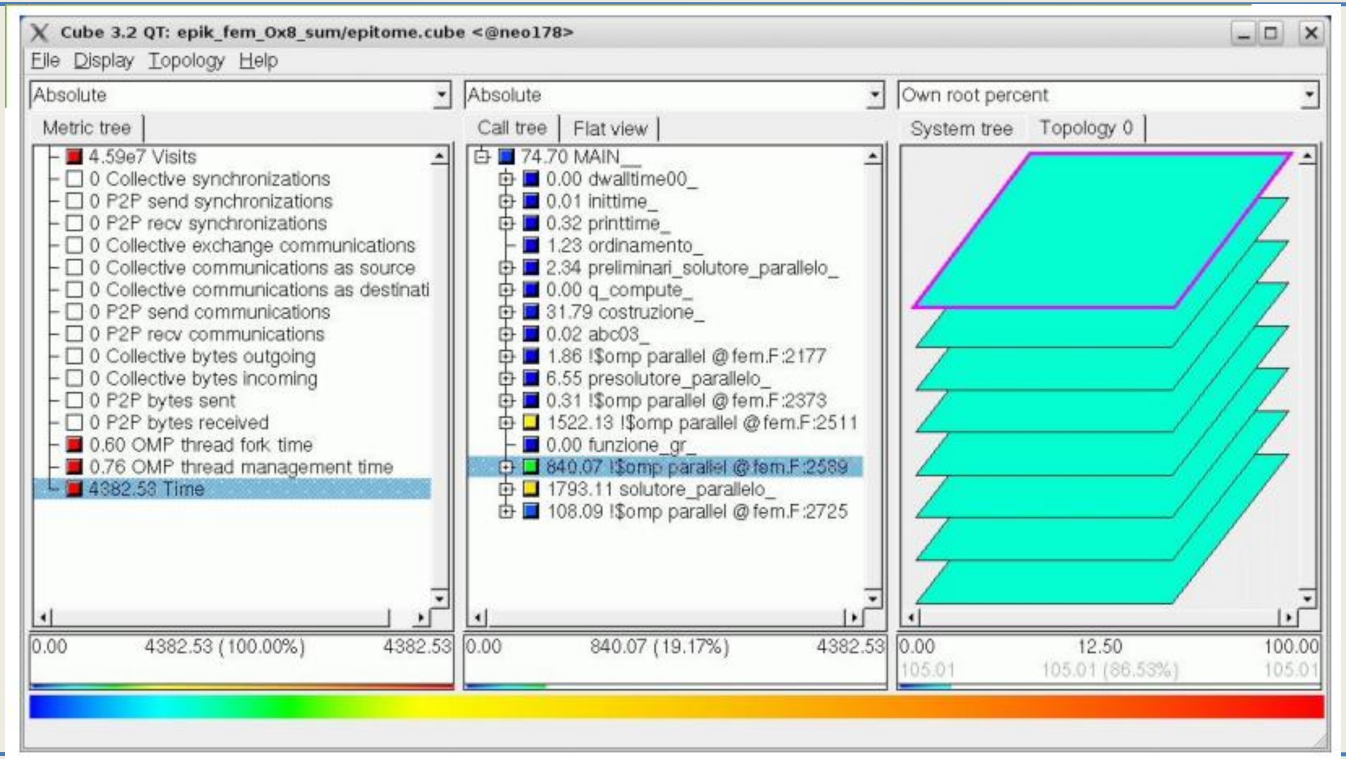
SCALASCA - Cube



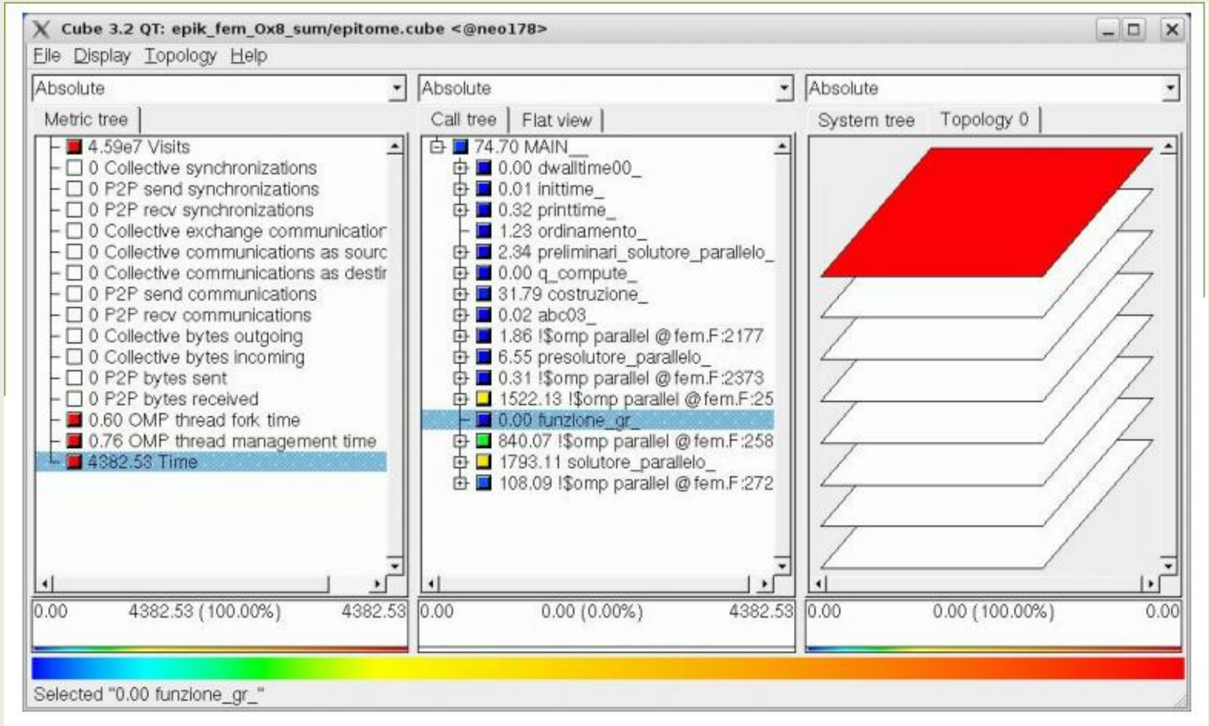
SCALASCA - Cube



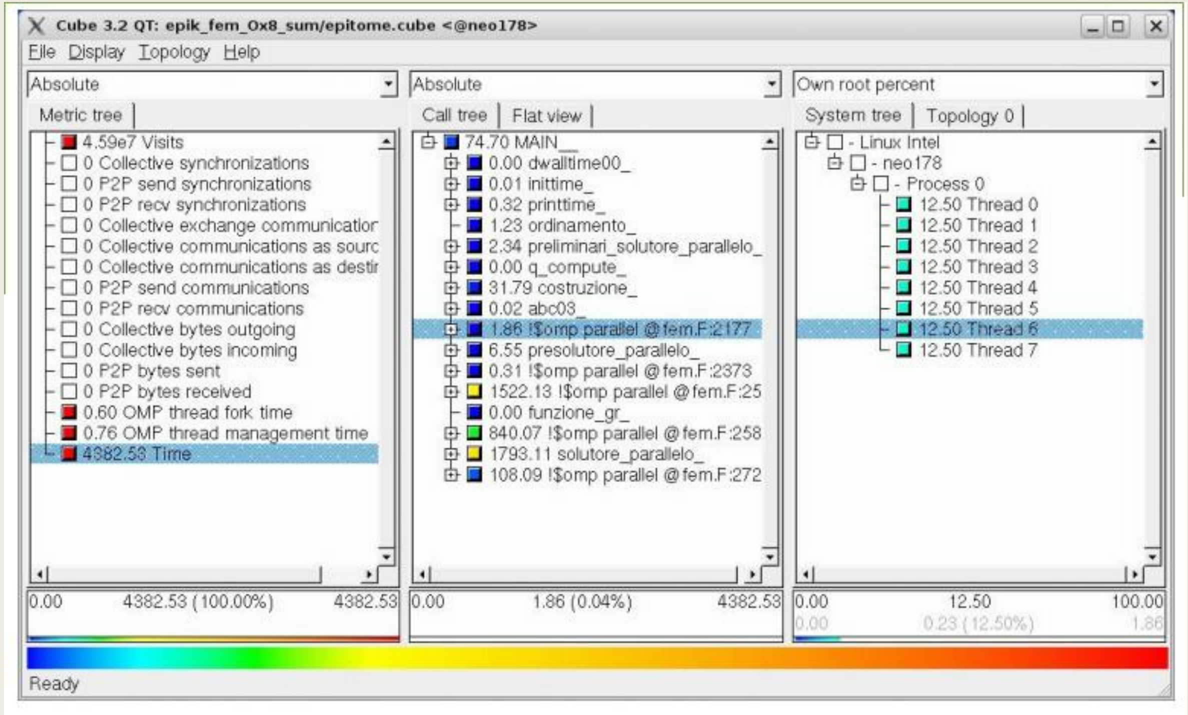
SCALASCA - Cube



SCALASCA - Cube

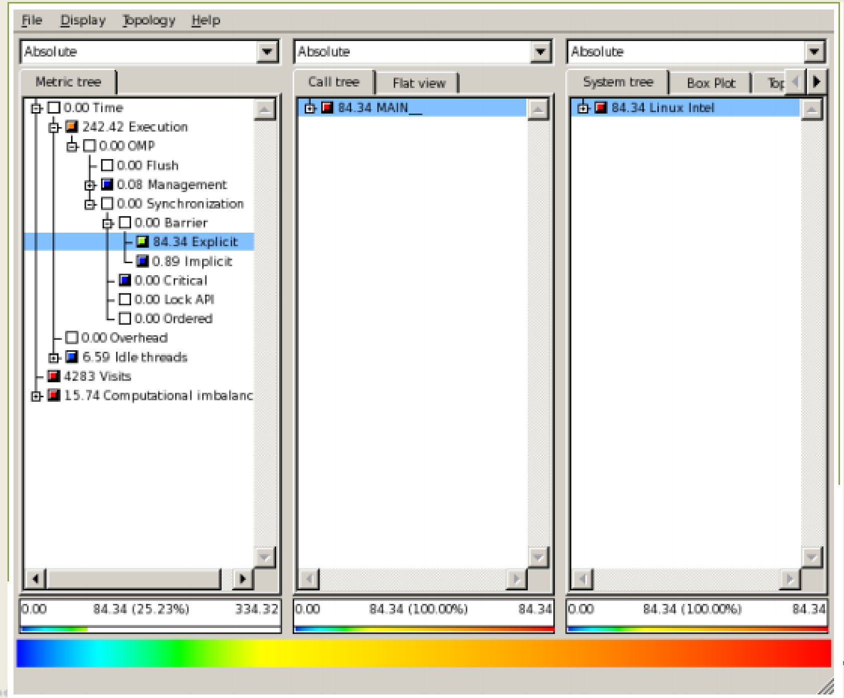


SCALASCA - Cube

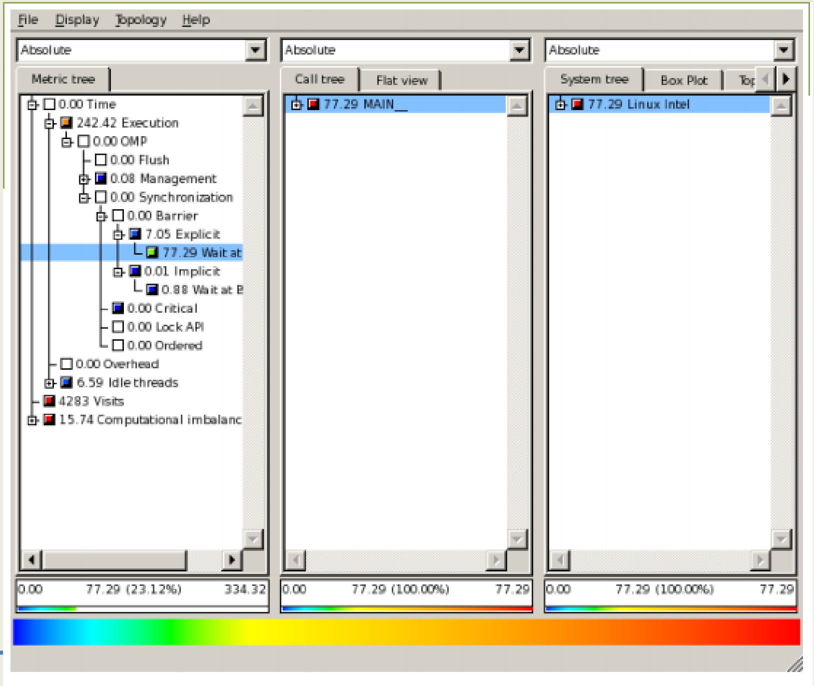


SCALASCA - Cube

Summary mode



Tracing mode



Conclusions

- Profiling is a necessary preliminar steps before the optimization
- Optimize the serial code before
- One single tool is not enough
- One single data set is not enough
- Consider the overhead induced by the profiler
- Use tools made available from your HPC centre