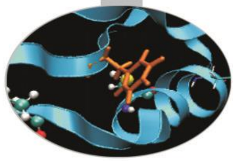
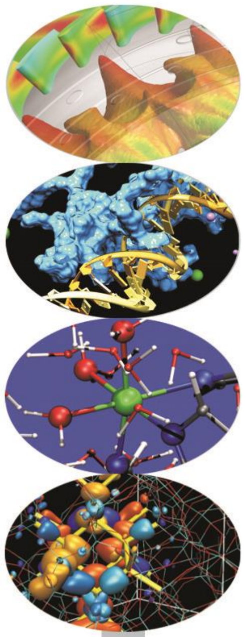


DEVELOPMENT TOOLS ON FERMI



Introduction to the FERMI Blue Gene/Q,

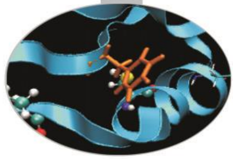
for users and developers

17 march 2014

a.marani@cineca.it

g.muscianisi@cineca.it

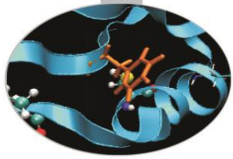
PROFILING & DEBUGGING ON FERMI



This talk will show the most important tools installed on FERMI for development purposes, i.e. debugging & profiling tools

FERMI isn't the proper cluster for development, since its particular architecture, so profiling and debugging on this machine may not be easy at all. However, something can still be done!

DISCLAIMER: This is NOT a lecture about how to use the tools presented, but about how to use them ON FERMI (compiling, running, etc.). For more informations about the tools general functionalities, please consult the documentation linked on the last slide.

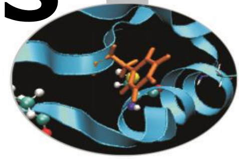


PART I

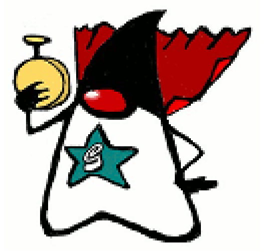
PROFILING ON FERMI

SCAI SuperComputing Applications and Innovation

FERMI PROFILING TOOLS



GPROF



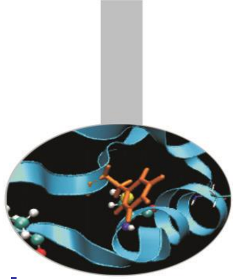
HPC toolkit

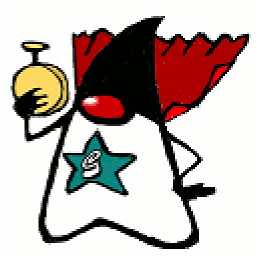
HPCToolkit

Scalasca

scalasca

GPROF





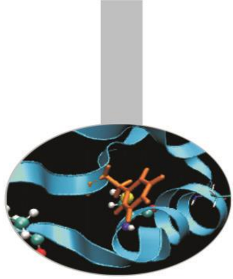
GNU Profiler – Gprof : The GNU profiler can be used to determine which parts of a program are taking most of the execution time.

Gprof can produce the following output styles:

- **Flat Profile**: The flat profile shows how much time was spent executing directly in each function.

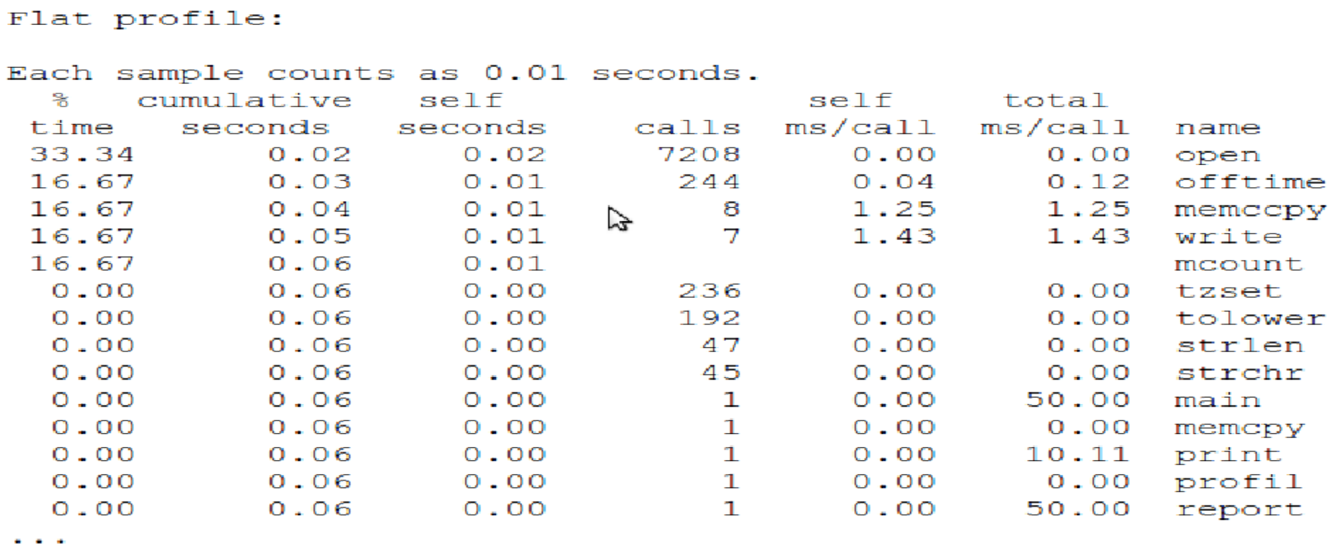
- **Call Graph**: The call graph shows which functions called which others, and how much time each function spent when its subroutine calls are included.

GPROF – FLAT PROFILE



The **flat profile** shows the total amount of time your program spent executing each function.

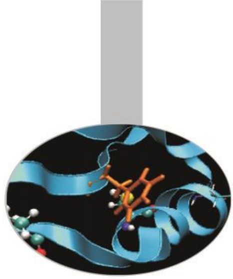
Note that if a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called.



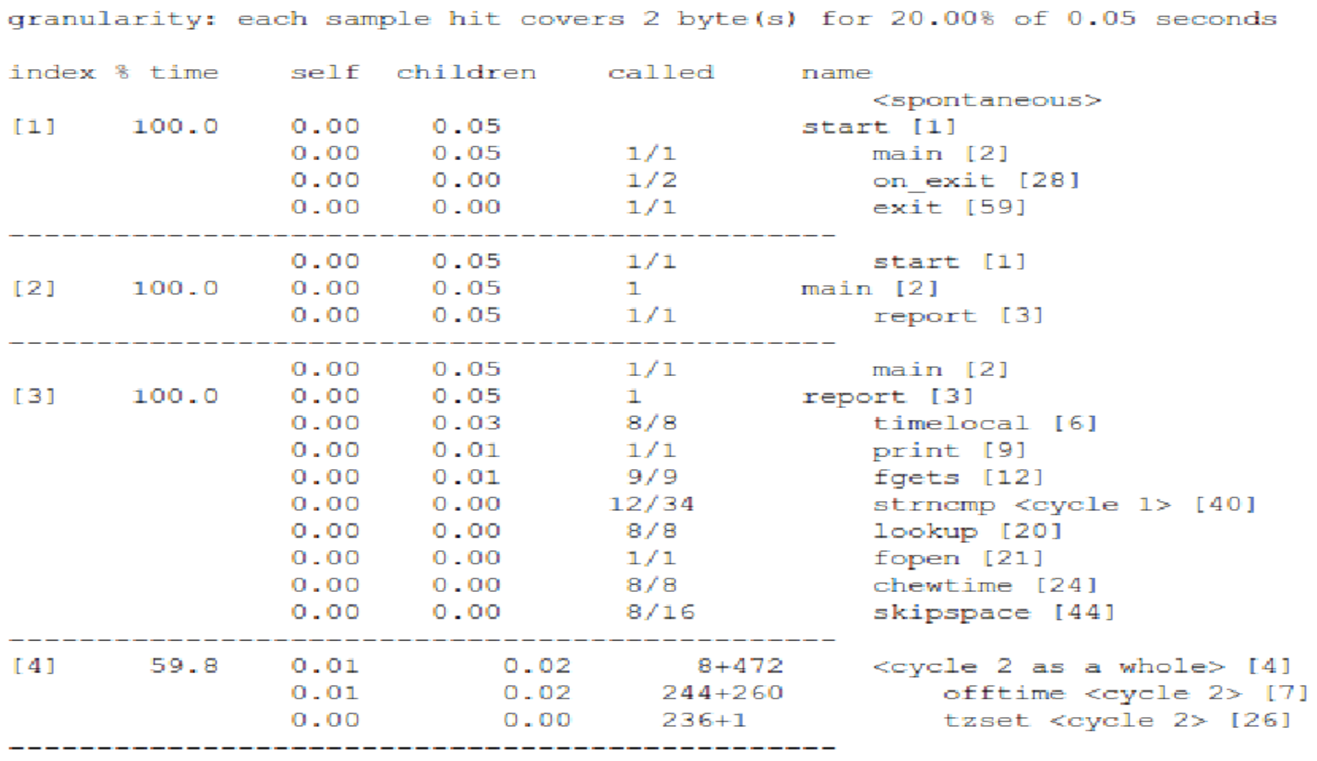
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

GPROF – CALL GRAPH



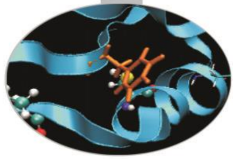
The **call graph** shows how much time was spent in each function and its children. With this you may spot functions that may not have used much time by themselves, but called other functions that did use unusual amounts of time



```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole> [4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

COMPILE, RUN & ANALYZE

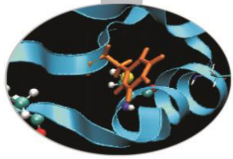


1. Compile and link the program with options: **-g -pg -qfullpath**
-g is for activating debugging specifics, **-pg** is for profiling ones
-qfullpath is for displaying the fullpath of the source code files in gprof output

2. Run the executable with a jobscript as usual

3. Several files will be generated, named gmon.out.<*MPI rank*>.
Those are binary files and therefore can't be read with normal text editors.
Convert them to text files with the command:
gprof test_gprof.exe gmon.out.0 > gprof.0.txt

DEALING WITH TASKS AND THREADS



By default, gmon.out files are generated only for ranks 0 - 31. You can customize the number of profiled ranks by setting the environment variable BG_GMON_RANK_SUBSET:

BG_GMON_RANK_SUBSET = N -- Generates the gmon.out file for rank N only

BG_GMON_RANK_SUBSET = N:M -- Generates gmon.out files for from rank N to M

BG_GMON_RANK_SUBSET = N:M:S -- Generates gmon.out files from rank from N to M, skipping S. For example, 0:16:8 generates gmon.out.0, gmon.out.8 and gmon.out.16

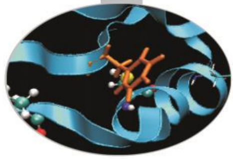
By default, thread profiling is not enabled. To enable it, set this environmental variable:

BG_GMON_START_THREAD_TIMERS = all -- enables the SIGPROF timer on all threads

BG_GMON_START_THREAD_TIMERS = nocomm -- enables the SIGPROF timer on all threads, except for the ones created to support MPI

- Add a call to the **gmon_start_all_thread_timers()** function to the program, from the main thread
- Add a call to the **gmon_thread_timer(int start)** function on the thread to be profiled: 1 to start, 0 to stop

IBM® HPC TOOLKIT



HPC toolkit is a collection of tools created for analyzing performance of parallel applications written in C or Fortran on BG/Q systems.

The tools provided by the toolkit provide profile analysis that can be categorized in four groups of interest:

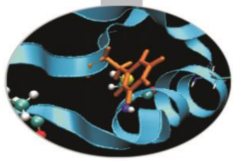
Hardware Performance Monitor (HPM): measurement of cache misses, number of floating point instructions executed, branch prediction counts, …

MPI profiling: tracing of MPI calls, communication patterns observation, measurement of the time spent in each MPI function and the size of the MPI messages

OpenMP profiling: informations about the time spent in OpenMP constructs, overhead in OpenMP constructs, balancing of the workload across OpenMP threads

I/O profiling: informations about I/O calls made in the application, understanding of the I/O performance of the application and to identifying possible I/O performance problems in the application (not treated in this presentation)

HPM LIBRARIES



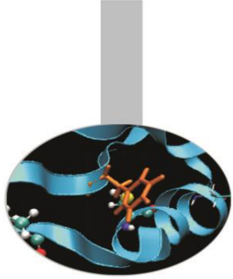
To profile with HPM libraries, you have to add to the source code the proper functions/routines in charge of event countering. It is possible to choose from a list of sets of hardware counter events to focus on a specific performance area.

The main functions are:

- **hpmInit()** for initializing the instrumentation library.
- **hpmTerminate()** for generating the reports and performance data files and shutting down the HPM environment.
- **hpmStart()** for identifying the start of a section of code in which hardware performance counter events will be counted.
- **hpmStop()** for identifying the end of the instrumented section.

hpmStart and **hpmStop** can be inserted as desidered, but they must be executed in pairs.
The section identifier label is passed as the parameter to the **hpmStart** and the matching **hpmStop** function.

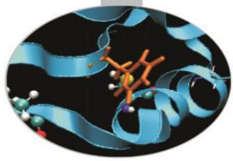
HPM LIBRARIES EXAMPLE



```c
#include <hpm.h>
int main(int argc, char *argv[ ]){
float x;
hpmInit();
x=10.0;
  hpmStart("Instrumented section 1");
    for(int i=0; i<100000; i++){
      x=x/1.001;
    }
  hpmStop("Instrumented section 1");
...
  hpmStart("Instrumented section 2");
    /* other computation */
    ...
  hpmStop("Instrumented section 2");
hpmTerminate();
}
```

```fortran
#include "f_hpm.h"
integer i
real*4 x
call f_hpminit();
x=10.0
   call f_hpmstart('Instrumented section 1', 22)
      do i=1,00000
        x=x/1.001
      enddo
   call f_hpmstop('Instrumented section 1', 22)
...
   call f_hpmstart('Instrumented section 2', 22)
!     other computation
      ...
   call f_hpmstop('Instrumented section 2', 22)
call f_hpmterminate()
end program
```

COMPILE & RUN



1. Set environment variables: run the setup script

    cd /bgsys/ibmhpc/ppedev.hpct
    ./env_sh   (for sh, bash,ksh shell)
    source snv_csh   (for csh shell)

2. Compile with **-g** and statically link HPM libraries.

    non-threaded application:

```
mpixlc myprog.c -o myprog -I/bgsys/ibmhpc/ppedev.hpct/include/ \
-L/bgsys/drivers/ppcflor/bgpm/lib/ \
-L/bgsys/ibmhpc/ppedev.hpct/lib64 -lhpc -lbgpm
```

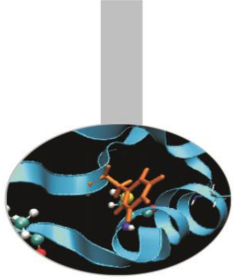
    threaded application:

```
 mpixlc_r myprog.c -o myprog_r -I/bgsys/ibmhpc/ppedev.hpct/include/ \
-L/bgsys/drivers/ppcflor/bgpm/lib/ \
-L/bgsys/ibmhpc/ppedev.hpct/lib64 -lhpc_r -lbgpm -qsmp=omp
```

3. Run the application as usual.

WARNING:
HPM libraries collect information and compute summaries during run time. Because of this, there may be **overhead** if instrumentation sections are inserted inside inner loops which are executed many times.

PERFORMANCE DATA FILES



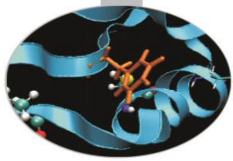
HPM will generate a performance data file for each rank, named *hpmCounts_<rank>.txt*
There is also a .viz file for visualization with Peekperf (more on this later).

Some environmental variables can control the generation of the HPM files:

HPM_IO_BATCH = set it to **yes** to reduce the number of output simultaneously opened by HPM in order to reduce file system impact
HPM_OUTPUT_PROCESS = set it to **all** if you want that all the MPI task write the performance data files; set it to **root** if you want that only root processor writes the performance data file.
HPM_SCOPE (non-threaded version) = set it to **node** to aggregate at node level the sum of the data file produced; set it to **process** if you want the each task produces a performance data file.

Default:
HPM_ASC_OUTPUT = no
HPM_VIZ_OUTPUT = yes
HPM_IO_BATCH = no
HPM_OUTPUT_PROCESS = all
HPM_SCOPE = process

MPI PROFILING LIBRARIES



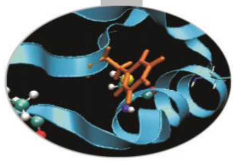
The **libmpitrace** library is used for profiling the MPI function calls, by creating a trace of them;

when an application is linked with such library, it intercepts the MPI calls in the application, using the Profiled MPI (PMPI) interface defined by the MPI standard, and obtains the needed profiling and trace informations.

The library also provides a set of functions that

- can be used to control how profiling and trace data is collected
- can be used to customize the trace data (see the manual linked at the documentation slide)

COMPILE & RUN



1. Set environment variables: run the setup script

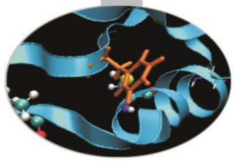
```
cd /bgsys/ibmhpc/ppedev.hpct
./env_sh        (for sh, bash,ksh shell)
source snv_csh    (for csh shell)
```

2. Compile with **-g** and statically link libmpitrace library.

```
mpixlc myprog.c -o myprog \
-I/bgsys/ibmhpc/ppedev.hpct/include/ \
-L/bgsys/ibmhpc/ppedev.hpct/lib64 -lmpitrace
```

3. Run the application as usual.

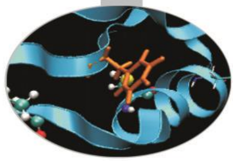
PERFORMANCE DATA FILES



A data file for each rank will be generated: *mpi_profile_<world_id>_<world_rank>.txt*
There is also a .viz file for visualization with Peekperf (more on this later).
world_id is the MPI world id; world_rank is the MPI task rank of the task that generated the file.
The file *single_trace_<world_id>* contains trace data, and can be visualized with Peekperf.

Default settings:
- number of trace event collected per task = 30000 . (MAX_TRACE_EVENTS)
- only 4 output files will be generated: for task 0, and for tasks having maximum, minimum and median total MPI communication time. (OUTPUT_ALL_RANKS)
- all the MPI calls after MPI_Init() are traced. (TRACE_ALL_EVENTS)
- max 256 MPI tasks are traced (MAX_TRACE_RANK, TRACE_ALL_TASKS)

OPENMP PROFILING LIBRARIES



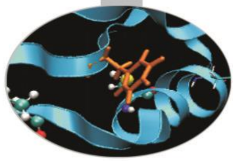
The openMP profiling libraries are used for analyzing performance problems in an OpenMP application.

They help in determining if the OpenMP application properly structures its processing for achieving the best performance.

They obtain informations about:

- time spent in OpenMP constructs in the application
- overhead in OpenMP constructs
- how is the workload balanced across OpenMP threads in the application

COMPILE & RUN



1. Set environment variables: run the setup script

```
cd /bgsys/ibmhpc/ppedev.hpct
./env_sh        (for sh, bash,ksh shell)
source snv_csh    (for csh shell)
```

2. Compile with **-g** and statically link openMP profiling libraries.

```
        mpixlc myprog.c -o myprog -qsmp=omp \
-L/bgsys/ibm_compilers/prod/opt/ibmcmp/xlsmp/bg/3.1/bglib64/ \
-lxlsmp_pomp -L/bgsys/ibmhpc/ppedev.hpct/lib64 -lpompprof_probe \
-lm -g
```

3. Run the application as usual.

A data file for each MPI rank will be generated:
popenmp_prof_<*rank*>
There is also a .viz file for visualization with Peekperf

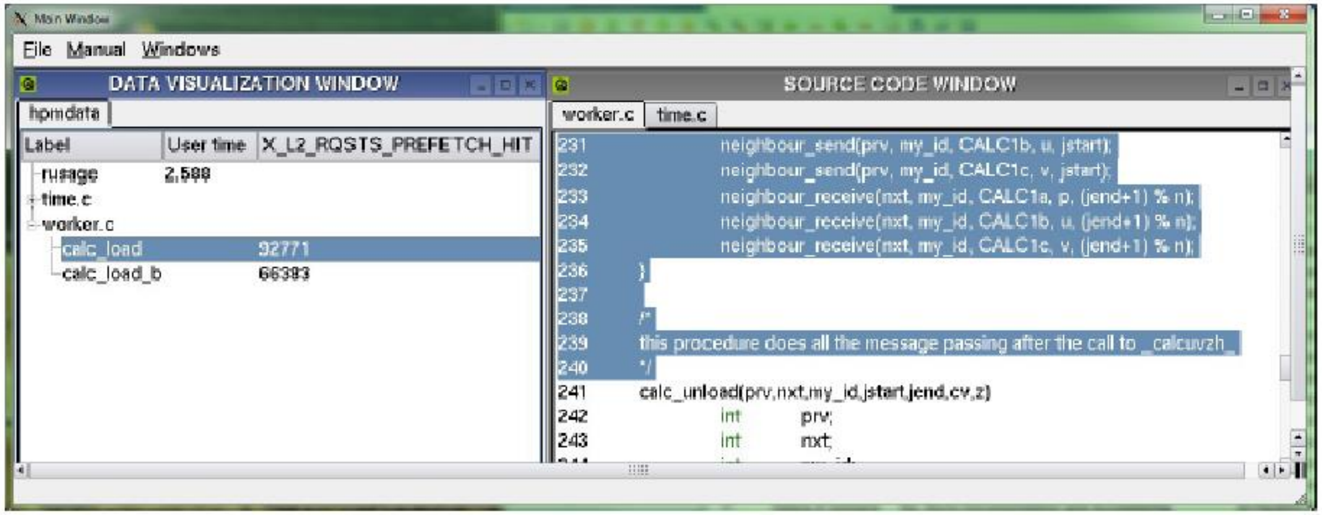
PEEKPERF

Peekperf provides a GUI interface to view application performance data.

It allows to visualize and analyze the collected performance data, collected in the in the visualization (.viz) files from the various instrumentation libraries.

If more than one visualization file is specified, peekperf combines the data from them and displays the result.
It also provides filtering and sorting capabilities to help you analyze the data.

/bgsys/ibmhpc/ppedev.hpct/bin/peekperf



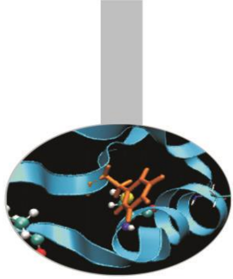
Label	User time	X_L2_RQSTS_PREFETCH_HIT
rusage	2.588	
time.c		
worker.c		
calc_load		32771
calc_load_b		66393

```
                neighbour_send(prv, my_id, CALC1b, u, jstart);
                neighbour_send(prv, my_id, CALC1c, v, jstart);
                neighbour_receive(nxt, my_id, CALC1a, p, (jend+1) % n);
                neighbour_receive(nxt, my_id, CALC1b, u, (jend+1) % n);
                neighbour_receive(nxt, my_id, CALC1c, v, (jend+1) % n);
        }

        /*
        this procedure does all the message passing after the call to _calcuvzh_
        */
        calc_unload(prv,nxt,my_id,jstart,jend,cv,z)
                int         prv;
                int         nxt;
```

Figure 14: Peekperf viewing hardware performance counter data

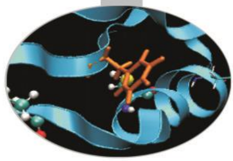
SCALASCA



SCALASCA (SCalable performance Analysis of LArge SCale Applications) is a software tool that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior. The analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes.

Like with HPC toolkit libraries, Scalasca takes advantage of a process of **event tracing**: for each thread/task is defined a buffer that measures the number of calls to functions, the time spent on each routine and so on. Final results are collected at the end.

PROFILING WITH SCALASCA



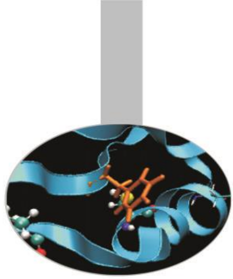
In order to profile an application with Scalasca, you have to compile and execute your application with the proper setting. Follow these instructions:

1. *module load autoload scalasca* (autoload the bgq-xl compiler)

2. Compile with *scalasca -instrument* (or its alias "skin"):
skin mpixlf90 -openmp -o bar bar.f90
Notice that there are no other specific flags that have to be added

3. Execute in a job script with *scalasca -analyze* (before the runjob command)

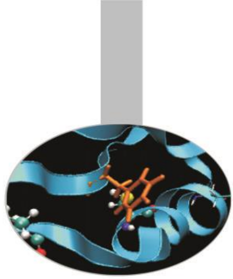
SCALASCA JOB SCRIPT EXAMPLE



```
#!/bin/bash
# @ job_name = myjob.$(jobid)
# @ output = $(job_name).out
# @ error = $(job_name).err
# @ environment = COPY_ALL
# @ job_type = bluegene
# @ wall_clock_limit = 1:00:00
# @ bg_size = 128
# @ account_no = <Account number>
# @ notification = always
# @ notify_user = <valid email address>
# @ queue

export OMP_NUM_THREADS=4
module load autoload scalasca/1.4.2
scalasca -analyze runjob --np 256 --ranks-per-node 4 --env-all --exe <my_exe>
```

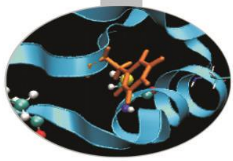
EPIK ARCHIVE



After the execution, a folder named ***epik_<myexe>_<resources> _sum*** will be created. It contains the following files:

epik.conf	Measurement configuration of the execution
epik.log	Output of the instrumented program and measurement system
epik.path	Callpath-tree recorded by the measurement system
epitome.cube	Intermediate analysis report of the runtime summarization system
summary.cube[.gz]	Post-processed analysis report of runtime summarization

EXAMINE RESULTS



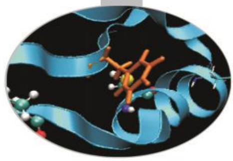
The content of the epik folder can be examinated via the *scalasca -examine* command:

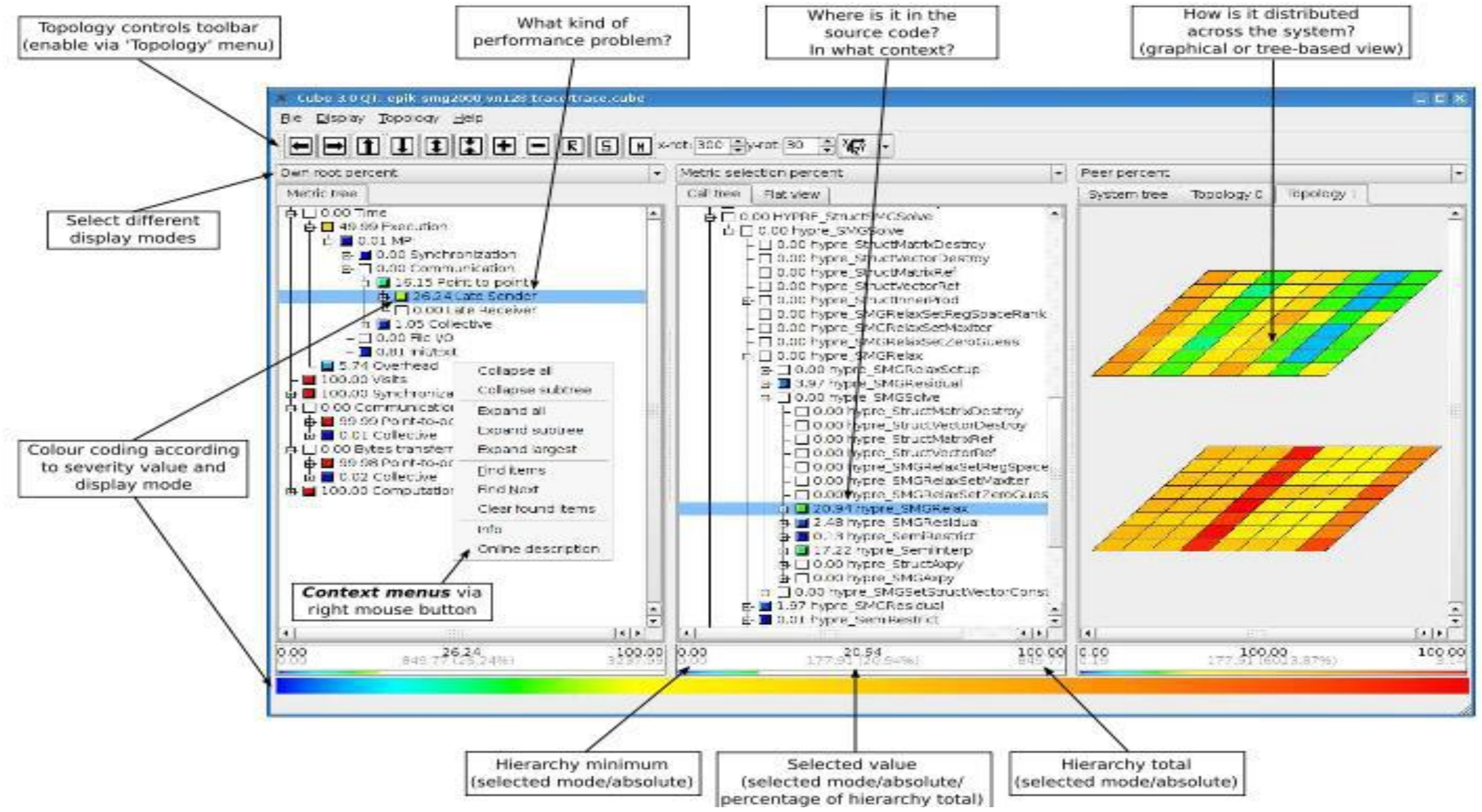
scalasca -examine epik_<myexe>_<resources>_sum
to examinate the report by GUI

scalasca -examine **-s** *epik_<myexe>_<resources>_sum*
to examinate the report by textual score output
(the file **epik.score** will be added in the epik directory)

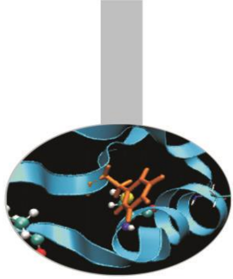
Results are displayed using three coupled tree browser showing:
- Metrics (i.e. Performance properties/problems)
- Call-tree or flat region profile
- System location

EXAMINATION BY GUI



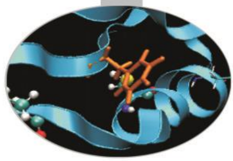

Topology controls toolbar (enable via 'Topology' menu)
What kind of performance problem?
Where is it in the source code? In what context?
How is it distributed across the system? (graphical or tree-based view)
Select different display modes
Colour coding according to severity value and display mode
Context menus via right mouse button
Hierarchy minimum (selected mode/absolute)
Selected value (selected mode/absolute/ percentage of hierarchy total)
Hierarchy total (selected mode/absolute)
Cube 3.0 QT: epik_smg2000_vn128_trace/trace.cube
File Display Topology Help
x-rot: 300 y-rot: 30
Own root percent
Metric selection percent
Peer percent
Metric tree
Call tree Flat view
System tree Topology 0 Topology 1
0.00 Time
49.99 Execution
0.01 MPI
0.00 Synchronization
0.00 Communication
15.15 Point to point
26.24 Late Sender
0.00 Late Receiver
1.05 Collective
0.00 File I/O
0.01 Init/Exit
5.74 Overhead
100.00 Visits
100.00 Synchronizations
0.00 Communications
99.99 Point-to-point
0.01 Collective
0.00 Bytes transferred
99.98 Point-to-point
0.02 Collective
100.00 Computation
Collapse all
Collapse subtree
Expand all
Expand subtree
Expand largest
Find items
Find Next
Clear found items
Info
Online description
0.00 HYPRE_StructSMGSolve
0.00 hypre_SMGSolve
0.00 hypre_StructMatrixDestroy
0.00 hypre_StructVectorDestroy
0.00 hypre_StructMatrixRef
0.00 hypre_StructVectorRef
0.00 hypre_StructInnerProd
0.00 hypre_SMGRelaxSetRegSpaceRank
0.00 hypre_SMGRelaxSetMaxIter
0.00 hypre_SMGRelaxSetZeroGuess
0.00 hypre_SMGRelax
0.00 hypre_SMGRelaxSetup
3.97 hypre_SMGResidual
0.00 hypre_SMGSolve
0.00 hypre_StructMatrixDestroy
0.00 hypre_StructVectorDestroy
0.00 hypre_StructMatrixRef
0.00 hypre_StructVectorRef
0.00 hypre_SMGRelaxSetRegSpace
0.00 hypre_SMGRelaxSetMaxIter
0.00 hypre_SMGRelaxSetZeroGuess
20.94 hypre_SMGRelax
2.48 hypre_SMGResidual
0.18 hypre_SemiRestrict
17.22 hypre_SemiInterp
0.00 hypre_StructCopy
0.00 hypre_SMGAxpy
0.00 hypre_SMGSetStructVectorConst
1.97 hypre_SMGResidual
0.01 hypre_SemiRestrict
0.00 26.24 100.00
849.77 (26.24%)
0.00 20.94 100.00
177.91 (20.94%)
0.00 100.00 100.00
177.91 (603.87%)

METRICS



Time	Total CPU allocation time
Visits	Number of times a routine/region was executed
Synchronizations	Total number of MPI synchronization operations that were executed
Communications	The total number of MPI communication operations, excluding calls transferring no data (which are considered Synchronizations)
Bytes transferred	The total number of bytes that were sent and received in MPI communication operations. It depends on the MPI internal implementation.
MPI file operations	Number of MPI file operations of any type.
MPI file bytes transferred	Number of bytes read or written in MPI file operations of any type.
Computational imbalance	This simple heuristic allows to identify computational load imbalances and is calculated for each (call-path, process/thread) pair.

MANUAL SOURCE CODE INSTRUMENTATION

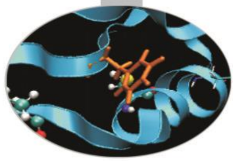


Region or phase annotations manually inserted in source file can augment or substitute automatic instrumentation, and can improve the structure of the analysis reports to make them more comprehensible to read

These annotations can be used to mark any sequence or block of statements, such as functions, phases, loop nests, etc., and can be nested, provided that every enter has matching exit

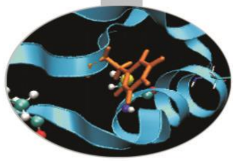
If automatic compiler instrumentation is not used, it is typically desiderable to manually instrument at least the **main** function/program and perhaps its major phases (e.g. initialization, core/body, finalization).

MANUAL INSTRUMENTATION EXAMPLE



```
#include “epik_user.h”
…
void foo(){
        … … // local declarations
        … … // more declarations
        EPIK_FUNC_START();
        … … // executable statements
        if(...){
                EPIK_FUNC_END();
                return;
        } else {
                EPIK_USER_REG (r_name,
                    “region”);
                EPIK_USER_START (r_name);
                … …
                … …
                EPIK_USER_END (r_name);
        }
        … … // executable statements;
        EPIK_FUNC_END();
        return;
```

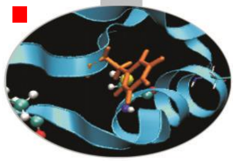
```
#include “epik_user.inc”
…
subroutine bar()
        EPIK_FUNC_REG(“bar”)
        EPIK_USER_REG (r_name,
             “region”)
        … … ! local declarations
        EPIK_FUNC_START();
        … … ! executable statements
        if(...) then
                EPIK_FUNC_END()
                return
        else
                EPIK_USER_START (r_name)
                … …
                … …
                EPIK_USER_END (r_name)
        endif
        … … ! executable statements
        EPIK_FUNC_END()
        return
```



PART II

DEBUGGING ON FERMI

DEBUGGING ON FERMI…



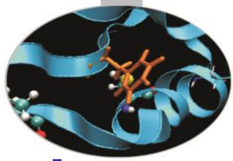
Debugging on FERMI is **no** easy task!

Error messages are often vague, and core files may be rather incomprehensible…

However, there are some useful tools that can help on the task!

Before that, let’s see some general advice for the setting of a debug session

COMPILING FOR A DEBUG SESSION



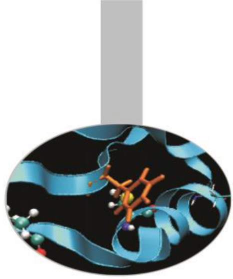
3 flags are required for compiling a program that can be analyzed by debugging tools:

-g : integrates debugging symbols on your code, making them "human readable" when analyzed from debuggers

-O0 : avoids any optimization on your code, making it execute the instructions in the exact order they're implemented

-qfullpath : Causes the full name of all source files to be added to the debug informations

OTHER USEFUL FLAGS



-qcheck   Helps detecting some array-bound violations, aborting with SIGTRAP at runtime

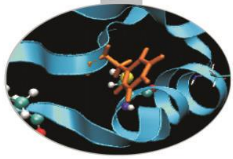
-qflttrap   Helps detecting some floating-point exceptions, aborting with SIGTRAP at runtime

-qhalt=<sev>   Stops compilation if encountering an error of the specified lever of severity

-qformat (C/C++)   Warns of possible problems with I/O format specification (printf,scanf…)

-qkeepparm   ensures that function parameters are stored on the stack even if the application is optimized.

FERMI DEBUGGING TOOLS



GDB

addr2line

Totalview

GDB

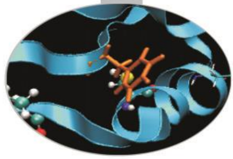
On FERMI, GDB is available both for front-end and back-end applications

Front-end: *gdb <exe>*
Back-end: */bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gdb <exe>*
Back-end shortcut: *module load gdb*
be-gdb <exe>

It is possible to make a post-mortem analysis of the **binary** core files generated by the job:
module load gdb
be-gdb <exe> <corefile>

To generate binary core files, add the following envs to runjob:
--envs BG_COREDUMPONEXIT=1
*--envs BG_COREDUMPBINARY=**
'*' means "all the processes". It is possible to indicate which ranks generate their core by specifying their number

GDB – REMOTE ACCESS



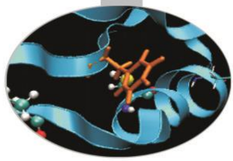
The Blue Gene/Q system includes support for using GDB real-time with applications running on compute nodes.

IBM provides a simple debug server called gdbserver. Each running instance of GDB is associated with one process or rank (also called GDB client).

Each instance of a GDB client can connect to and debug one process. To debug multiple processes at the same time, run multiple GDB tools at the same time. A maximum of four GDB tools can be run on one job.

…so, how to do that?

USING GDB ON RUNNING APPLICATIONS



1) First of all, submit your job as usual;

llsubmit <jobscript>

2) Then, get your job ID;

llq -u $USER

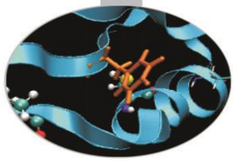
3) Load the GDB module; it contains the shortcut for the back-end GDB and the script for the environment setting

module load gdb

4) Launch the “gdb-setup” script. It will print the instructions for the next steps

gdb-setup -j <jobid> -r <rank #>

USING GDB ON RUNNING APPLICATIONS



5) Launch GDB! (back-end version);

be-gdb ./myexe

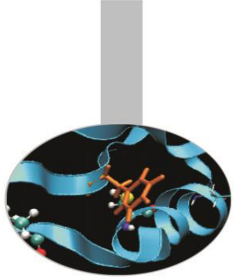
6) Connect remotely to your job process (the value of <*IP address*> was printed on the screen after step 4);

(gdb) target remote <IP address>:10000

7) Start debugging!!!

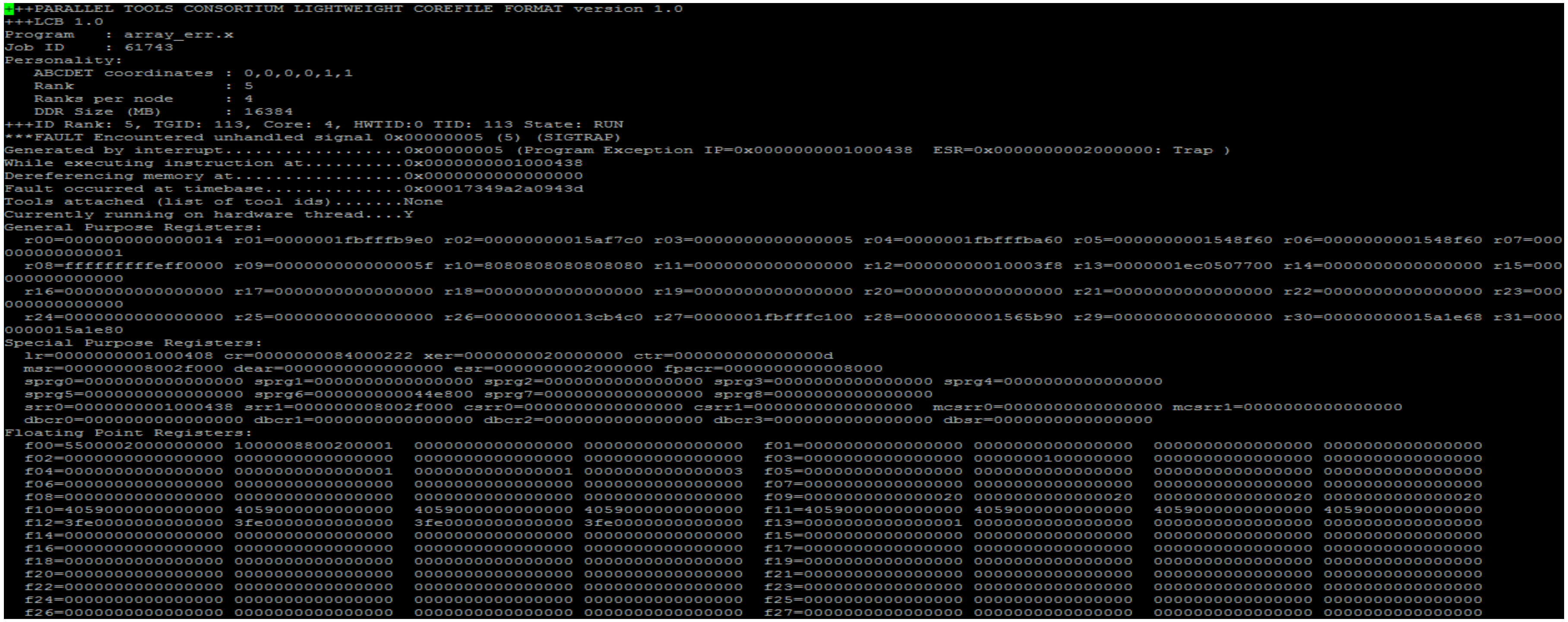
(Although you aren’t completely free…for example, command ‘run’ does not work)

ADDR2LINE



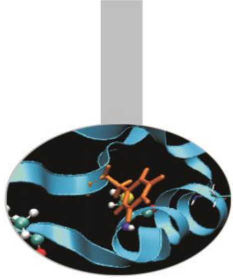
If nothing is specified, an unsuccesful job generates a text core file for the processes that caused the crash…

…however, those core files are all but easily readable!



```
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.0
+++LCB 1.0
Program     : array_err.x
Job ID      : 61743
Personality:
   ABCDET coordinates : 0,0,0,0,1,1
   Rank               : 5
   Ranks per node     : 4
   DDR Size (MB)      : 16384
+++ID Rank: 5, TGID: 113, Core: 4, HWTID:0 TID: 113 State: RUN
***FAULT Encountered unhandled signal 0x00000005 (5) (SIGTRAP)
Generated by interrupt..................0x00000005 (Program Exception IP=0x0000000001000438  ESR=0x0000000002000000: Trap )
While executing instruction at..........0x0000000001000438
Dereferencing memory at.................0x0000000000000000
Fault occurred at timebase..............0x00017349a2a0943d
Tools attached (list of tool ids).......None
Currently running on hardware thread....Y
General Purpose Registers:
  r00=0000000000000014 r01=0000001fbfffb9e0 r02=00000000015af7c0 r03=0000000000000005 r04=0000001fbfffba60 r05=0000000001548f60 r06=0000000001548f60 r07=000
000000000001
  r08=fffffffffeff0000 r09=000000000000005f r10=8080808080808080 r11=0000000000000000 r12=00000000010003f8 r13=0000001ec0507700 r14=0000000000000000 r15=000
000000000000
  r16=0000000000000000 r17=0000000000000000 r18=0000000000000000 r19=0000000000000000 r20=0000000000000000 r21=0000000000000000 r22=0000000000000000 r23=000
000000000000
  r24=0000000000000000 r25=0000000000000000 r26=00000000013cb4c0 r27=0000001fbfffc100 r28=0000000001565b90 r29=0000000000000000 r30=00000000015a1e68 r31=000
0000015a1e80
Special Purpose Registers:
  lr=0000000001000408 cr=0000000084000222 xer=0000000020000000 ctr=000000000000000d
  msr=000000008002f000 dear=0000000000000000 esr=0000000002000000 fpscr=0000000000008000
  sprg0=0000000000000000 sprg1=0000000000000000 sprg2=0000000000000000 sprg3=0000000000000000 sprg4=0000000000000000
  sprg5=0000000000000000 sprg6=000000000044e800 sprg7=0000000000000000 sprg8=0000000000000000
  srr0=0000000001000438 srr1=000000008002f000 csrr0=0000000000000000 csrr1=0000000000000000  mcsrr0=0000000000000000 mcsrr1=0000000000000000
  dbcr0=0000000000000000 dbcr1=0000000000000000 dbcr2=0000000000000000 dbcr3=0000000000000000 dbsr=0000000000000000
Floating Point Registers:
  f00=5500002000000000 1000008800200001  0000000000000000 0000000000000000   f01=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f02=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f03=0000000000000000 0000000100000000  0000000000000000 0000000000000000
  f04=0000000000000000 0000000000000001  0000000000000001 0000000000000003   f05=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f06=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f07=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f08=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f09=0000000000000020 0000000000000020  0000000000000020 0000000000000020
  f10=4059000000000000 4059000000000000  4059000000000000 4059000000000000   f11=4059000000000000 4059000000000000  4059000000000000 4059000000000000
  f12=3fe0000000000000 3fe0000000000000  3fe0000000000000 3fe0000000000000   f13=0000000000000001 0000000000000000  0000000000000000 0000000000000000
  f14=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f15=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f16=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f17=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f18=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f19=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f20=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f21=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f22=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f23=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f24=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f25=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f26=0000000000000000 0000000000000000  0000000000000000 0000000000000000   f27=0000000000000000 0000000000000000  0000000000000000 0000000000000000
```

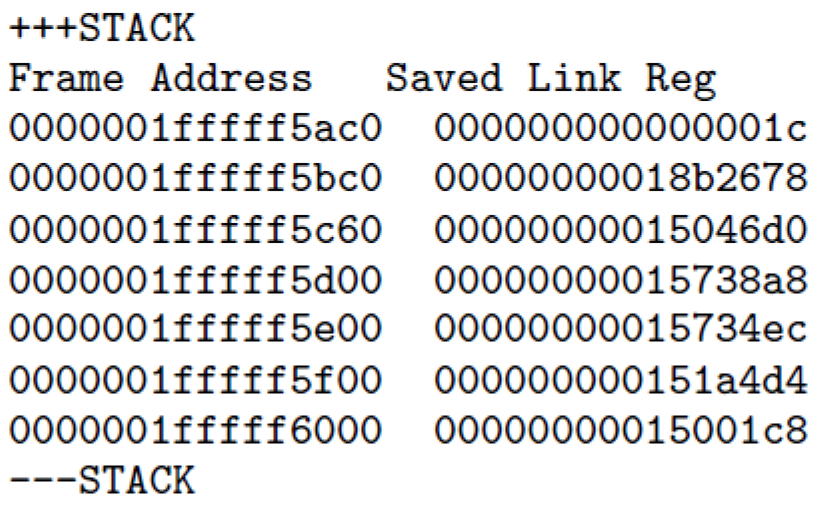
addr2line is an utility that allows to get from this file informations about where the job crashed



CORE FILES

Blue Gene core files are lightweight text files

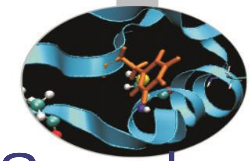
Hexadecimal addresses in section STACK describe function call chain until program exception. It’s the section delimited by tags: +++STACK / ---STACK



```
+++STACK
Frame Address     Saved Link Reg
0000001fffff5ac0  000000000000001c
0000001fffff5bc0  00000000018b2678
0000001fffff5c60  00000000015046d0
0000001fffff5d00  00000000015738a8
0000001fffff5e00  00000000015734ec
0000001fffff5f00  000000000151a4d4
0000001fffff6000  00000000015001c8
---STACK
```

In particular, “Saved Link Reg” column is the one we need!

USING ADDR2LINE



From the core file output, save only the addresses in the Saved Link Reg column:

```
000000000000001c
00000000018b2678
00000000015046d0
00000000015738a8
00000000015734ec
000000000151a4d4
00000000015001c8
```

Replace the first eight 0s with 0x:

```
00000000018b2678 => 0x018b2678
```

If you load the module “superc”, a simple script called “a2l-translate” is capable of doing the replacement for you:

module load superc

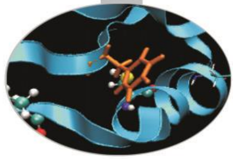
a2l-translate corefile

Lauch addr2line:

addr2line -e ./myexe 0x018b2678

addr2line -e ./myexe < addresses.txt

TOTALVIEW

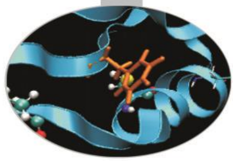


TotalView is a GUI-based source code defect analysis tool that gives you control over processes and thread execution and visibility into program state and variables.

It allows you to debug one or many processes and/or threads with complete control over program execution.

It is by far the most versatile and user-friendly debugger on our clusters!!

REMOTE CONNECTION MANAGER (RCM)

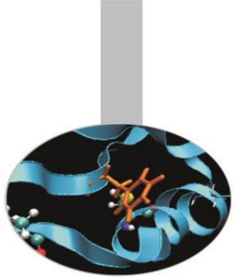


Launching Totalview on FERMI is complicated, since it involves working with a GUI during a working execution. Thus you need to see what's inside the computing nodes, that aren't very "graphical-friendly"

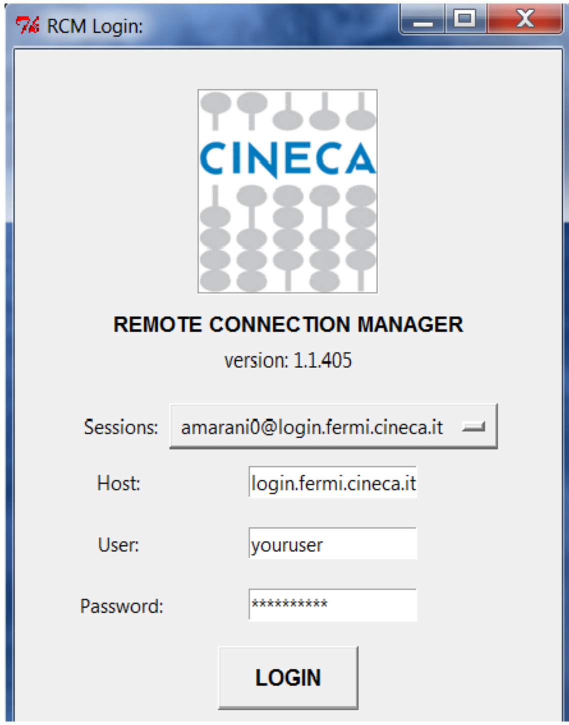
There are some workarounds for this issue, one of them involves estabilishing a VNC connection and use it for SSH tunneling to your local workstation

The easiest, however, is to use the **Remote Connection Manager** devolped by CINECA!!

USING TOTALVIEW: PREPARATION



1) Download the version of RCM that suits your Operative System: http://www.hpc.cineca.it/services/remote-visualisation


RCM Login:

REMOTE CONNECTION MANAGER

version: 1.1.405

Sessions: amarani0@login.fermi.cineca.it

Host: login.fermi.cineca.it

User: youruser

Password: **********

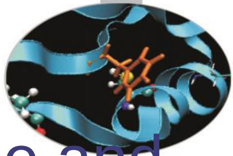
LOGIN

2) For Windows users, an X server like Xming may be needed: http://sourceforge.net/projects/xming/ . Download and launch it.

3) Launch RCM and fill the boxes as in the picture (type the credentials you use for accessing FERMI)

4) Select “new display”. A job (budget-free) in the special “visual” queue will create a remote display for 12 hours

USING TOTALVIEW: JOB SCRIPT SETTING

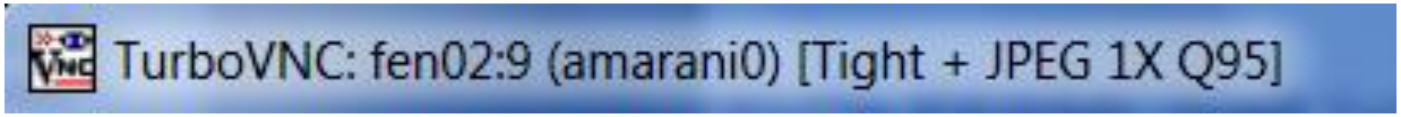


5) Inside your job script, you have to load the proper module and export the DISPLAY environment variable:

module load totalview

export DISPLAY=fen<no>:xx

where *xx* and <*no*> are as you can see in the name of the remote display window (as in the picture)



TurboVNC: fen02:9 (amarani0) [Tight + JPEG 1X Q95]

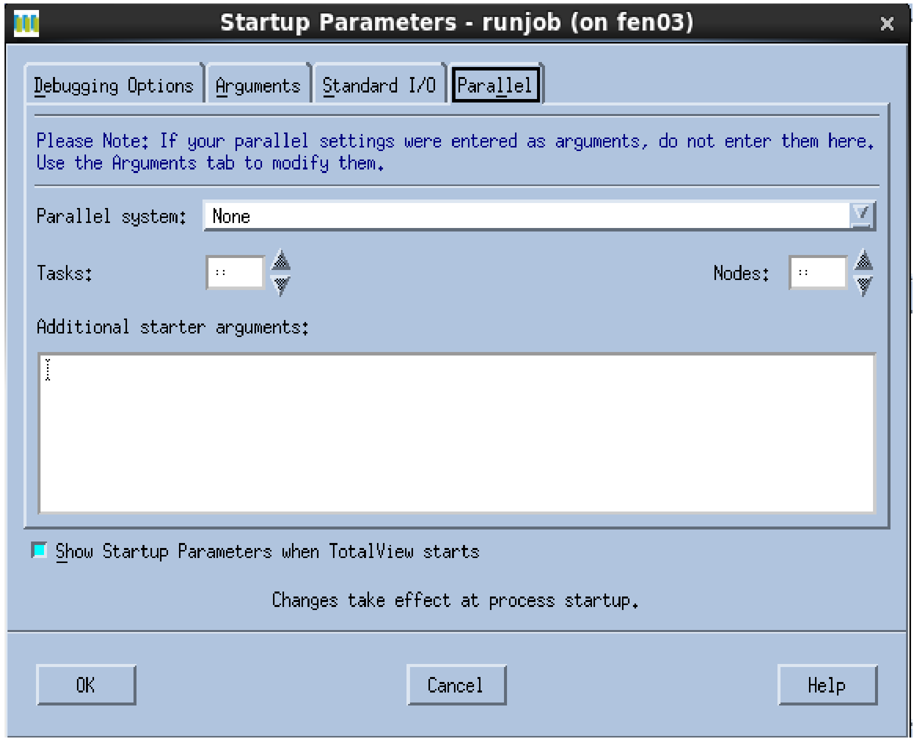
6) Totalview execution line (inside your LoadLeveler script) will be as follows:

totalview runjob -a <runjob arguments: --np, --exe, --args…>

7) Launch the job. When it will start running, you will find a Totalview window opened on your remote display! Closing Totalview will also kill the job.

SuperComputing Applications and Innovation

Using Totalview: start debugging


Startup Parameters - runjob (on fen03)

Please Note: If your parallel settings were entered as arguments, do not enter them here. Use the Arguments tab to modify them.

Parallel system: None

Tasks:

Nodes:

Additional starter arguments:

Show Startup Parameters when TotalView starts

Changes take effect at process startup.

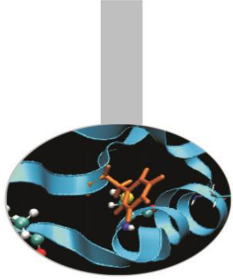
Select “BlueGene” as a parallel system, and a number of tasks and nodes according to the arguments you gave to runjob during submission phase.

Click “Go” (the green arrow) on the next screen and your application will start running.

WARNING: due to license issues, you are NOT allowed to run Totalview sessions with more than 1024 tasks simultaneously!!!

WARNING: The BG/Q version of Totalview doesn’t implement Replay Engine yet.

DOCUMENTATION



PROFILING:

GPROF
http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html
HPC TOOLKIT (ON BG\Q)
http://community.hartree.stfc.ac.uk/access/content/group/admin/HPC%20Training/BG_Q%20training%20course%20February%202013/Reference/hpct_guide_bgq_V1.1.1.0.pdf
SCALASCA
http://apps.fz-juelich.de/scalasca/releases/scalasca/1.4/docs/UserGuide.pdf

DEBUGGING: http://www.hpc.cineca.it/sites/default/files/Debug%20guide_0.pdf

GDB
https://sourceware.org/gdb/current/onlinedocs/gdb/
TOTALVIEW
http://www.roguewave.com/portals/0/products/totalview-family/totalview/docs/8.13/html/index.html#page/User_Guides/totalviewug-title.html

The course "Introduction to HPC Scientific Programming: tools and techniques" may also prove useful!
http://www.hpc.cineca.it/content/hpc-scientific-programming