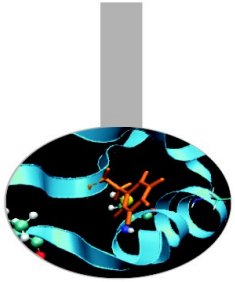
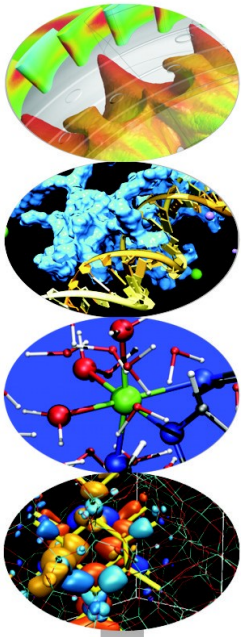
CINECA SCAI
SuperComputing Applications and Innovation



Profiling and Debugging on a Blue Gene/Q system

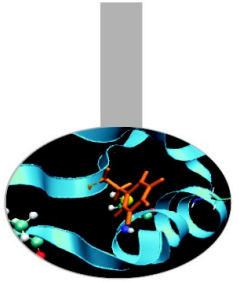


Giusy Muscianisi – g.muscianisi@cineca.it

SuperComputing Applications and Innovation Department

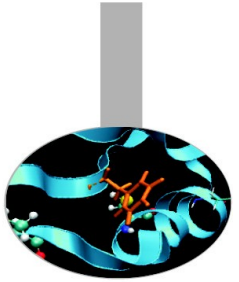
Feb 05, 2013

CINECA

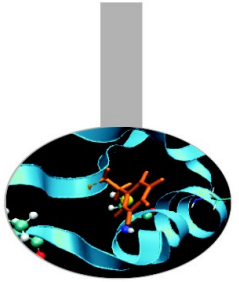


Outline

- **Profiling**
  - Scalasca
  - IBM® High Performance Computing (HPC) Toolkit
  - GNU Profiler – Gprof
- **Debugging**
  - GDB
  - addr2line
  - Totalview

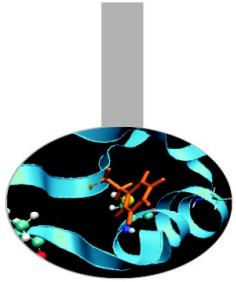


PROFILING



Outline

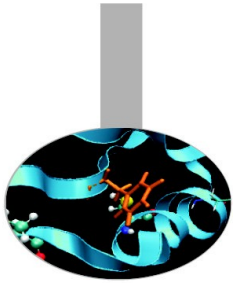
- Profiling
  - Scalasca
  - IBM® High Performance Computing (HPC) Toolkit
  - GNU Profiler – Gprof
- Debugging
  - GDB
  - addr2line
  - Totalview



Scalasca

- SCalable performance Analysis of LArge SCale Applications

- Developed by Juelich Supercomputer Centre
- Toolset for performance analysis of parallel applications on a large scale
- Manage programs MPI, OpenMP, MPI+OpenMP
- Latest releast 1.4.2, available on FERMI
- www.scalasca.org
- http://www2.fz-juelich.de/jsc/datapool/scalasca/UserGuide.pdf

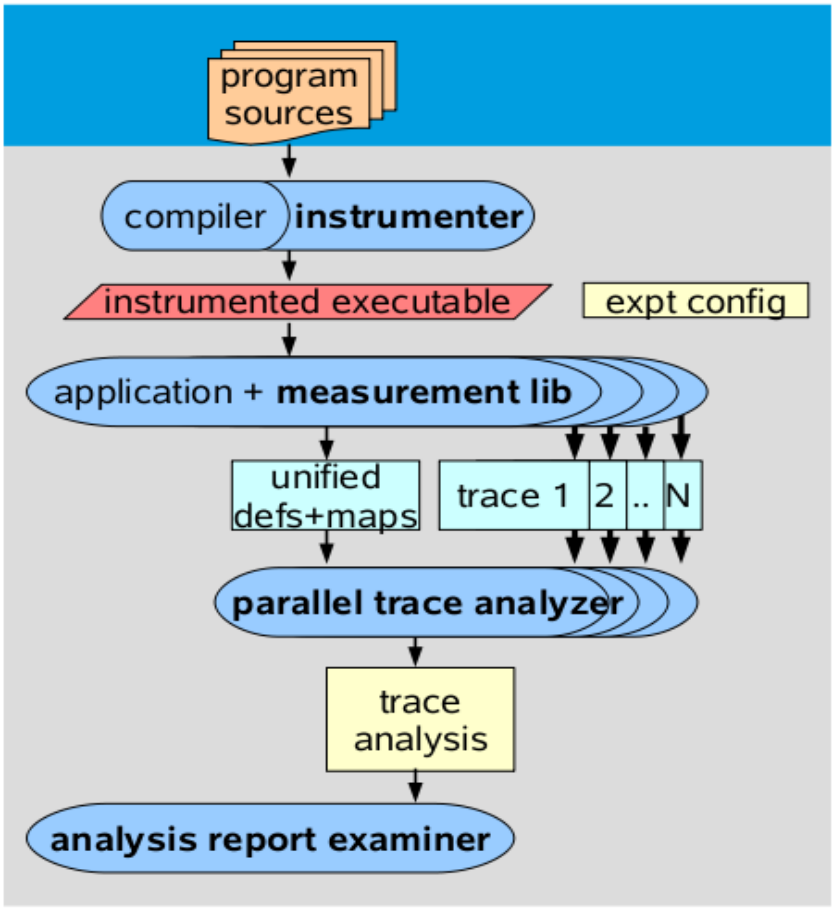


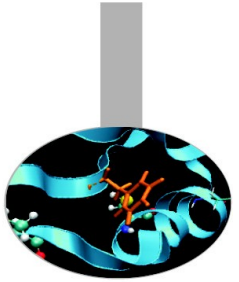
Scalasca

Event tracing

During the measurement there is a buffer for each thread/process

Final collect of the results





Scalasca -- How to use

- prepare application objects and executable for measurement (automatic instrumentation)

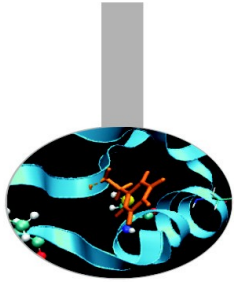
  `scalasca -instrument <compile-or-link-command>`

- run application under control of measurement system

  `scalasca -analyze <application-launch-command>`

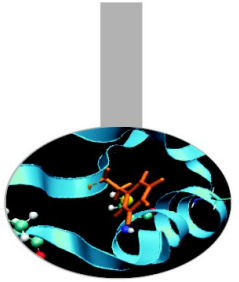
- post-process & explore measurement analysis report

  `scalasca -examine <experiment-archive|report>`



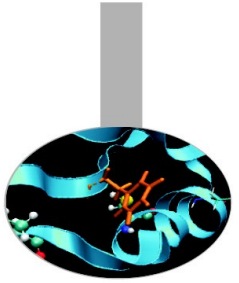
Instrumentation (default)

Original command	SCALASCA instrumentation command
`mpixlc -c foo.c`	`scalasca -instrument mpixlc -c foo.c`
`mpixlf90 -openmp -o bar bar.f90`	`skin mpixlf90 -openmp -o bar bar.f90`



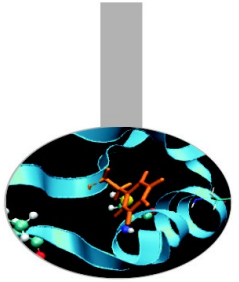
Analysis -- Pure MPI

```
#!/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

module load bgq-xl/1.0
module load scalasca/1.4.2
scalasca -analyze runjob --np 256 --ranks-per-node 2 --exe
<my_exe>
```



Analysis -- MPI+OpenMP

```
#!/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

module load bgq-xl/1.0
module load scalasca/1.4.2
scalasca -analyze runjob --np 256 --ranks-per-node 4 -envs
OMP_NUM_THREADS=4 --exe <my_exe>
```



Archive with log files 1/2

- Pure MPI:

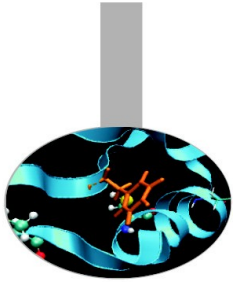
```
scalasca -analyze runjob --np 256 --ranks-per-node
2 --exe <my_exe>
```

  ==> epik_<myexe>_2p256_sum

- MPI + OpenMP:

```
scalasca -analyze runjob --np 256 --ranks-per-node
4 -envs OMP_NUM_THREADS=4 --exe <my_exe>
```

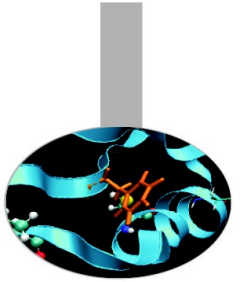
  ==> epik_<myexe>_4p256x4_sum



Archive with log files 2/2

In each epik archive there are the following files:

epik.conf	Measurement configuration when the experiment was collected
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



Log informations in the STDERR

At the beginning of the run:

S=C=A=N: Scalasca 1.4.2 runtime summarization

S=C=A=N: ./epik_pluto_1p16_sum experiment archive

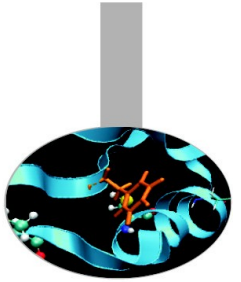
S=C=A=N: Mon Nov 26 20:33:07 2012: Collect start

/bgsys/drivers/ppcfloor/bin/runjob --np 16 --ranks-per-node 1 --envs EPK_TITLE=pluto_1p16_sum --envs EPK_LDIR=. --envs EPK_SUMMARY=1 --envs EPK_TRACE=0 : ./pluto -show-dec

At the end of the run:

S=C=A=N: Mon Nov 26 20:36:23 2012: Collect done (status=0) 196s

S=C=A=N: ./epik_pluto_1p16_sum complete.



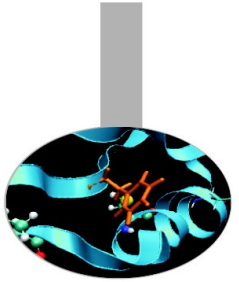
Log informations in the STDOUT

At the beginning of the run:

[00000]EPIK: Created new measurement archive ./epik_pluto_1p16_sum
[00000]EPIK: Activated ./epik_pluto_1p16_sum [NO TRACE] (0.048s)
[00000]EPIK: MPI-2.2 initialized 16 ranks

At the end of the run:

[00000]EPIK: Closing experiment ./epik_pluto_1p16_sum
[00000]EPIK: Largest definitions buffer 34547 bytes
[00000]EPIK: 218 unique paths (218 max paths, 7 max frames, 0 unknowns)
[00000]EPIK: cpath[25]: rid=14 ppid=12 order=188813.183310 OutOfOrder
[00000]EPIK: Unifying... done (0.125s)
[00000]EPIK: Collating... done (0.371s)
[00000]EPIK: Closed experiment ./epik_pluto_1p16_sum (0.507s)
maxHeap(*)=1.617/25.832MB



Examination

scalasca -examine epik_<myexe>_<resources>_sum

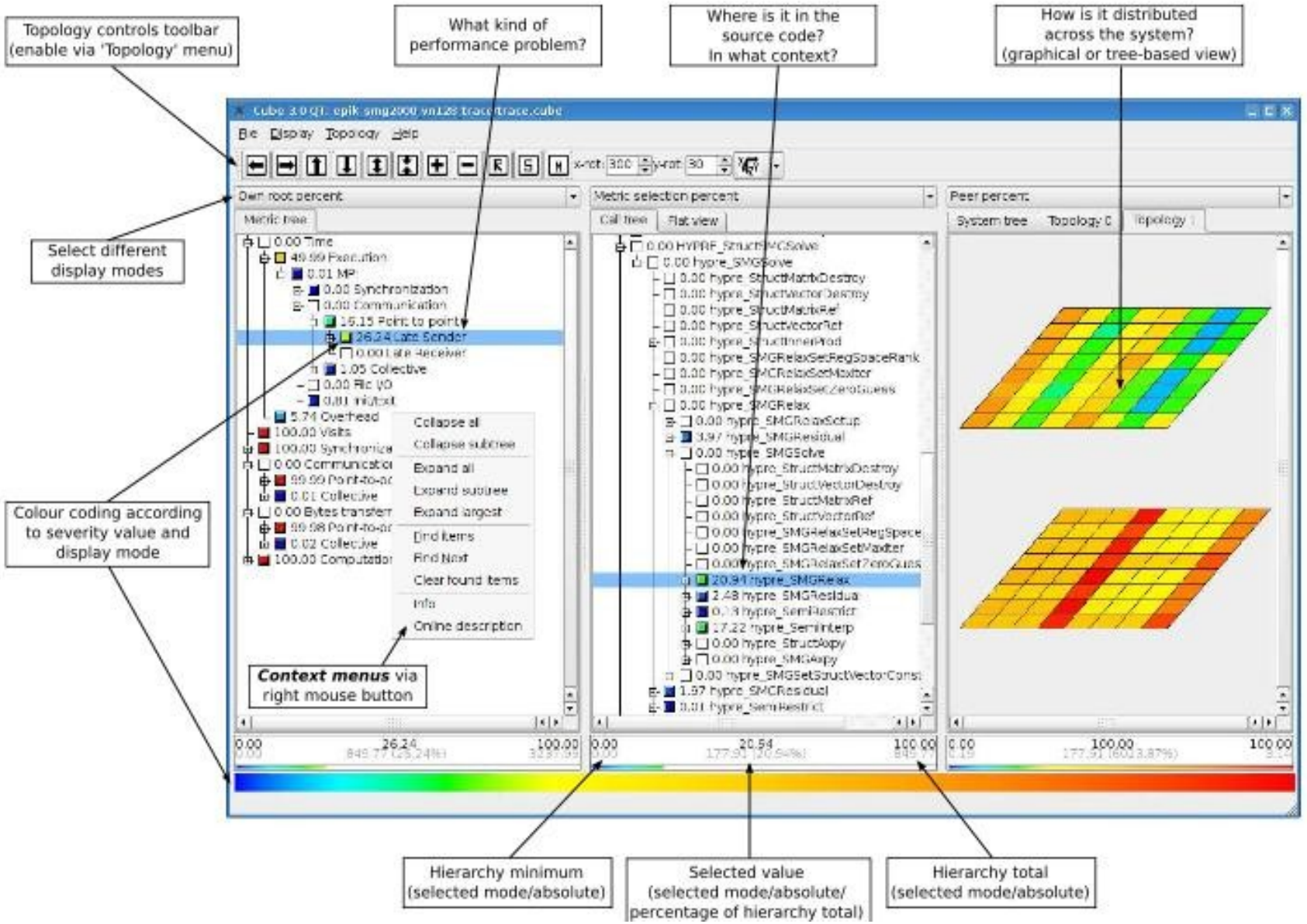
to examinate the report by GUI

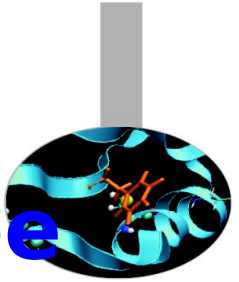
scalasca -examine **-s** epik_<myexe>_<resources>_sum

to examinate the report by textual score output

- The file **epik.score** will be added in the epik_ directory

Examination by GUI





Examination by text - summary.cube

```
flt  type      max_tbc        time      % region
     ANY   4769825184   2340248.68  100.00 (summary) ALL
     MPI      4593960    613242.18   26.20 (summary) MPI
     COM       348528    212240.56    9.07 (summary) COM
     USR   4764882648   1514021.98   64.69 (summary) USR

     USR   2839795200    184264.39    7.87 PrimToChar
     USR    946598400    186935.74    7.99 PrimEigenvectors
     USR    188793792     34584.14    1.48 SoundSpeed2
     USR    148475712     41108.12    1.76 Flux
     USR     91241568     32795.35    1.40 PrimToCons
     USR     82213824     41377.87    1.77 ConsToPrim
     USR     74237856     36365.90    1.55 RightHandSide
     USR     74237856     22645.11    0.97 CT_StoreEMF
     USR     74237856    370444.42   15.83 Roe_Solver
     USR     40318080     21342.57    0.91 CTU_CT_Source
     USR     40318080      6966.30    0.30 CheckPrimStates
     USR     40318080      4172.12    0.18 PrimSource
     USR     40318080    203264.59    8.69 CharTracingStep
     USR     40318080    279464.91   11.94 States
     USR     40318080     16575.16    0.71 CheckNaN
     MPI      4343328     16560.02    0.71 MPI_Sendrecv
     USR      1765632       335.38    0.01 Init
     COM       241296       337.55    0.01 AL_Exchange_dim
     USR       196608        22.27    0.00 Length_1
     USR       196608        23.28    0.00 Length_3
     USR       196608        22.77    0.00 Length_2
     USR       194064        14.31    0.00 print1
     USR       175296        26.61    0.00 SetIndexes
     USR       131472       136.96    0.01 ResetState
...
```

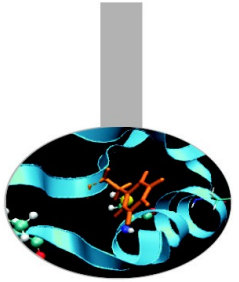
ANY / ALL = provide aggregate information for all measured routines

MPI = refers to function calls to the MPI library

OMP = either to OpenMP regions or calls to the OpenMP API

COM = User-program routines on paths that directly or indirectly call MPI or OpenMP provide valuable context for understanding the communication and synchronization behaviour of the parallel execution

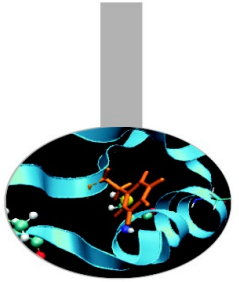
USR = User-programroutines that are involved with purely local computation



Display of results

Results are displayed using three coupled tree browser showing:

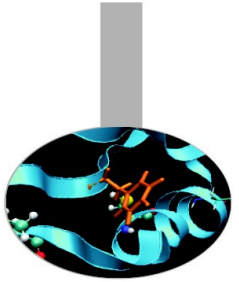
- Metrics (i.e. Performance properties/problems)
- Call-tree or flat region profile
- System location



Metrics 1/2

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.

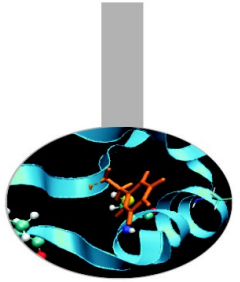
http://www2.fz-juelich.de/jsc/datapool/scalasca/scalasca_patterns-1.4.html



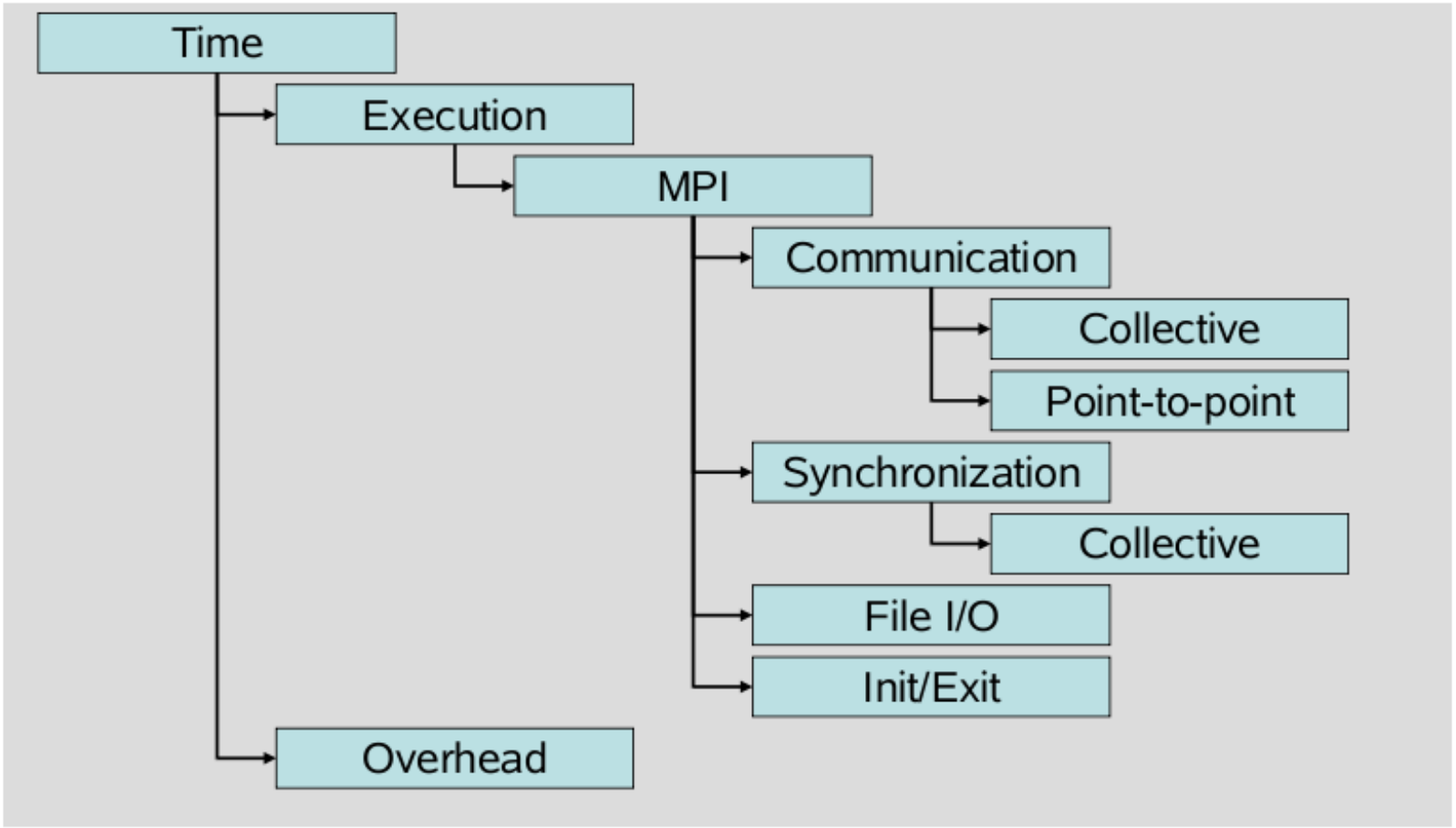
Metrics 2/2

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.

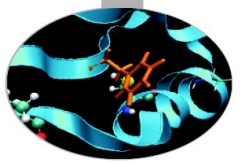
http://www2.fz-juelich.de/jsc/datapool/scalasca/scalasca_patterns-1.4.html



Metrics – Time, pure MPI code 1/2

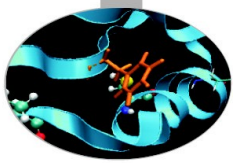


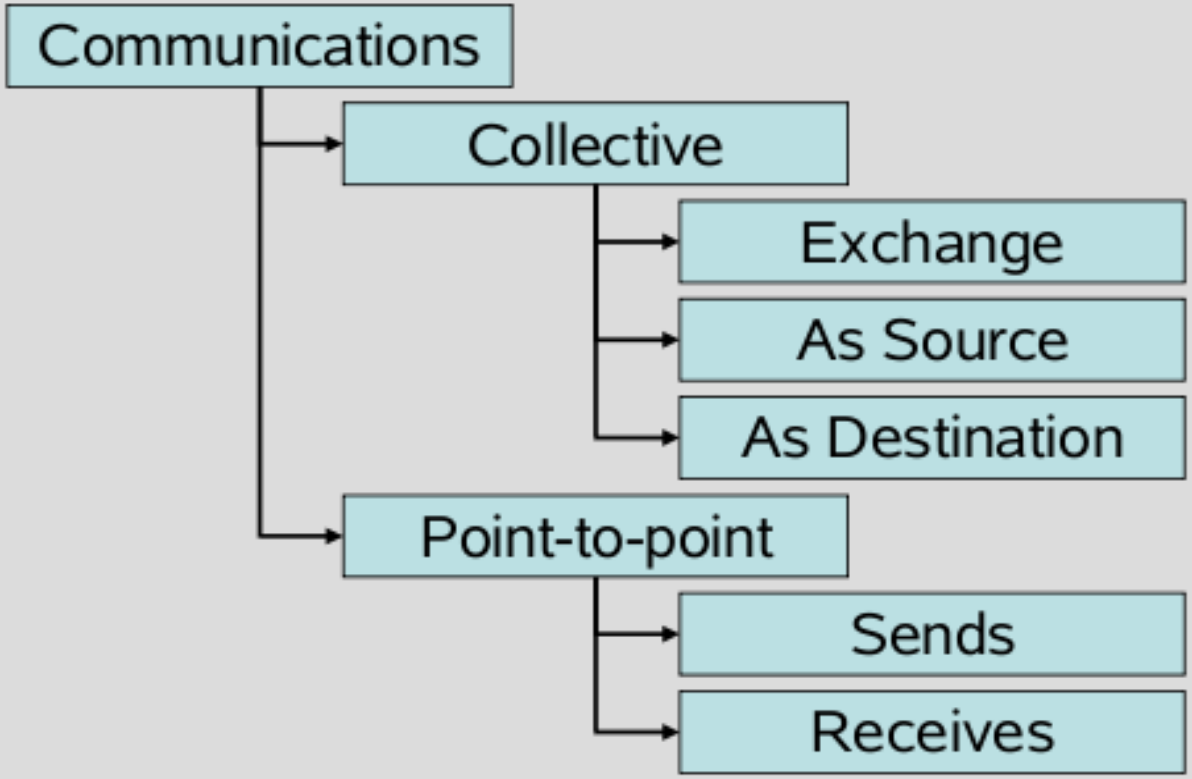
Metrics – Time, pure MPI code 2/2



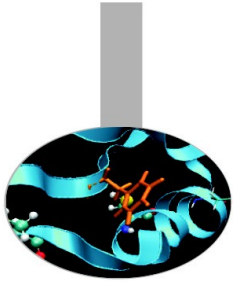
Time	Total CPU allocation time
Execution	Execution time without overhead
Overhead	Time spent in tasks related to measurement (not including dilation from instrumentation!)
MPI	Time spent in pre-instrumented MPI functions
Communication	Time spent in MPI communication calls, subdivided into collective and point-to-point
Synchronization	Time spent in calls to `MPI_Barrier()`
File I/O	Time spent in MPI file I/O functions
Init/Exit	Time spent in `MPI_Init()` and `MPI_Finalize()`

Pure MPI code – Communications

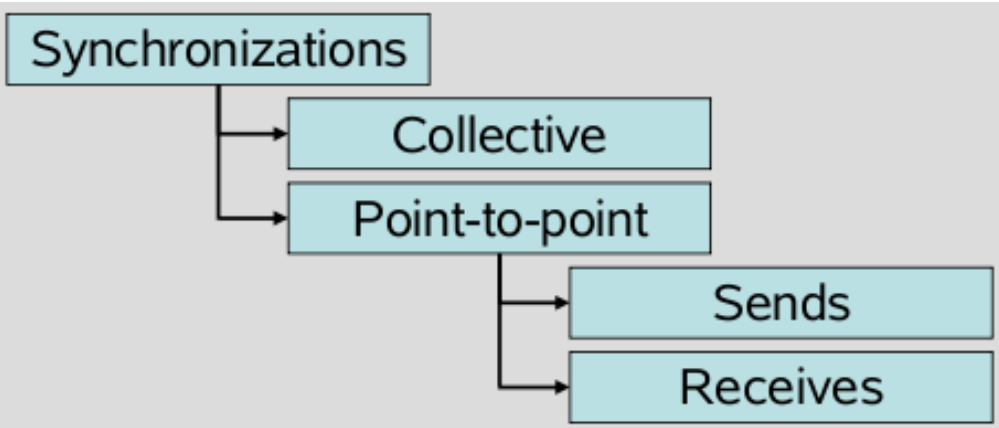




- Provides the number of calls to an MPI communication function of the corresponding class
- Zero-sized message transfers are considered *synchronization*!

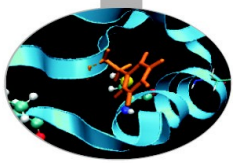


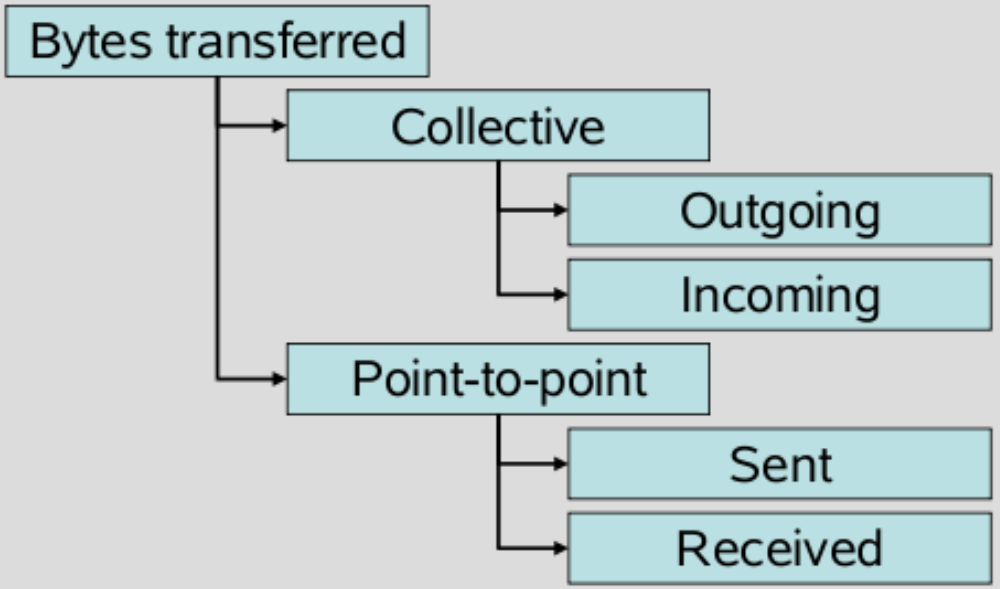
Pure MPI code – Synchronizations



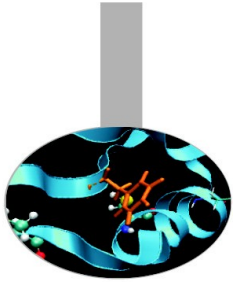
- Provides the number of calls to an MPI synchronization function of the corresponding class
- MPI synchronizations include zero-sized message transfers!

Pure MPI code – Bytes transferred

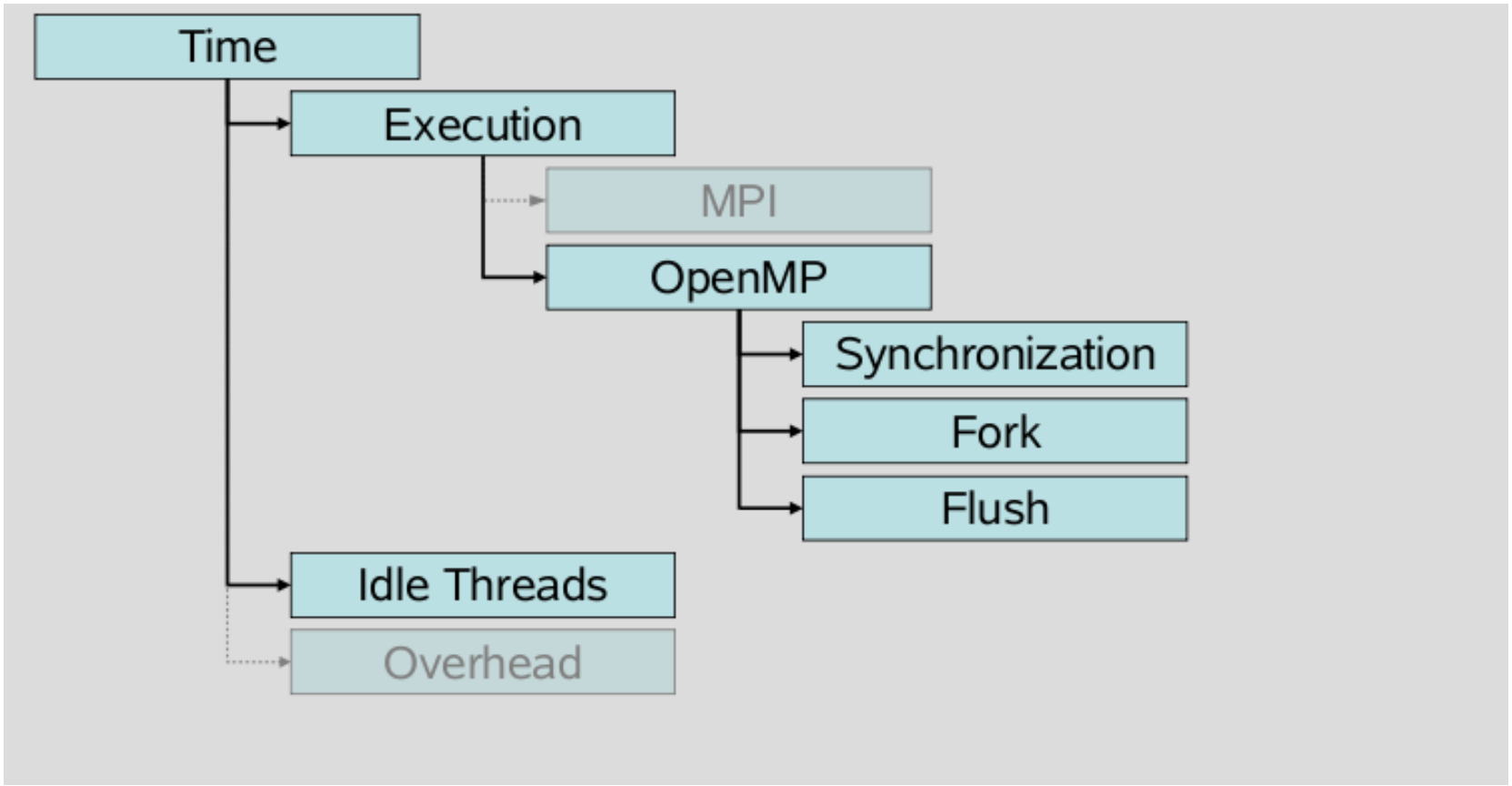


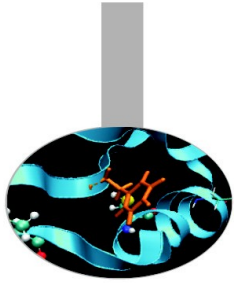


- Provides the number of bytes transferred by an MPI communication function of the corresponding class



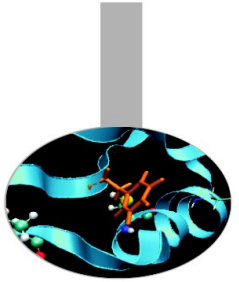
Metrics – Time, MPI-OpenMP code





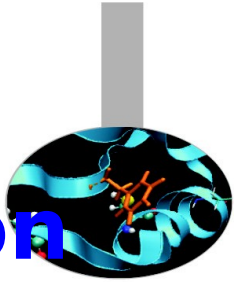
Time, OpenMP part of the code

OpenMP	Time spent for all OpenMP-related tasks
Synchronization	Time spent synchronizing OpenMP threads
Fork	Time spent by master thread to create thread teams
Flush	Time spent in OpenMP flush directives
Idle Threads	Time spent idle on CPUs reserved for slave threads



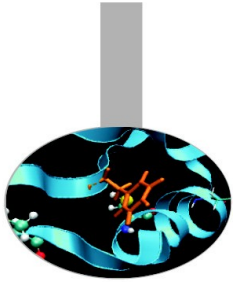
Hardware counters measurement

- Hardware counter measurement is disabled by default
- Can be enabled using
  - the environment variable EPK_METRICS in the jobscript (`scalasca -analyze`)
  - `scalasca -analyze -m <metric_name> runjob ….`
- Set EPK_METRICS to a colon-separated list of counter names, or a predefined platform-specific group
- Metric names can be chosen from the list contained in file $SCALASCA_HOME/doc/METRICS.SPEC



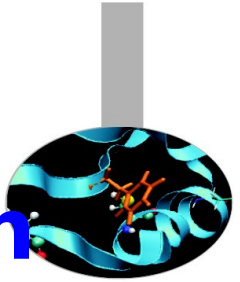
Manual source-code instrumentation

- Region or phase annotations manually inserted in source file can augmented or substiture automatic instrumentation, and can improve the structure of analysis reports to make them more readly comprehensible
- 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).



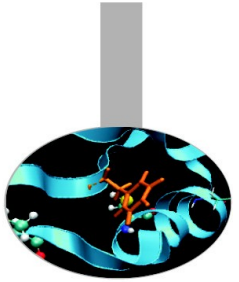
User instrumentation API -- C/C++

```
#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;
}
```



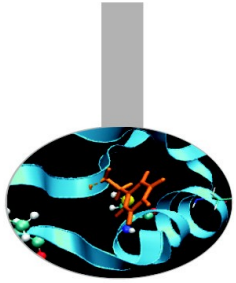
User instrumentation API -- Fortran

```
#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
end subroutine bar
```



Outline

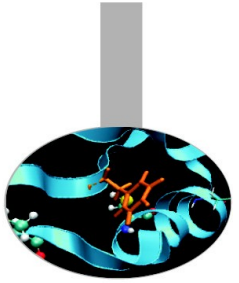
- Profiling
  - Scalasca
  - IBM® High Performance Computing (HPC) Toolkit
  - GNU Profiler – Gprof
- Debugging
  - GDB
  - addr2line
  - Totalview



IBM® HPC Toolkit

Collection of tools to analyze performance of parallel applications written in C or Fortran on BG/Q systems.

- **Hardware Performance Monitor (HPM)**: measurement for cache misses, number of floating point instructions executed, branch prediction counts, and so on.
- **MPI profiling**: tracing of MPI calls, to observe the communication patterns, to measure both the time spent in each MPI function and the size of the MPI messages.
- **OpenMP profiling**: informations on the time spent in OpenMP constructs, overhead in OpenMP constructs, how workload is balanced across OpenMP threads.
- **I/O profiling**: informations about I/O calls made in the application, to understand application I/O performance and to identify possible I/O performance problems in the application. (available in December 2012).

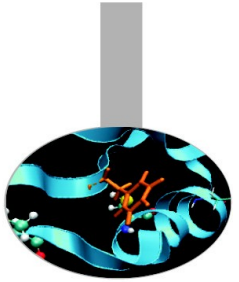


IBM® HPC Toolkit – peekperf GUI

peekperf GUI:

Provides a GUI interface to view application performance data.

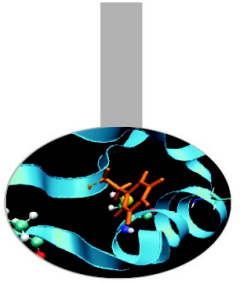
- allow to visualize and analyze the collected performance data.
- can display the data in the visualization (.viz) files from the various instrumentation libraries.
- if more than one visualization file is specified, peekperf combines the data from them for display.
- provides filtering and sorting capabilities to help you analyze the data.



Reference

Reference guide of IBM ®HPC Toolkit

https://www.ibm.com/developerworks/wikis/download/attachments/91226643/hpct_guide_bgq_V1.1.1.0.pdf



HPM Libraries

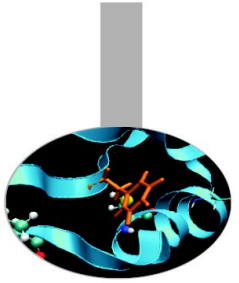
Hardware Performance Monitor (HPM)

Access hardware performance counters to analyze the performance of the application.

It is possible to choose from a list of sets of hardware counter events to focus on a specific performance area.

Available libraries (both for C and Fortran):

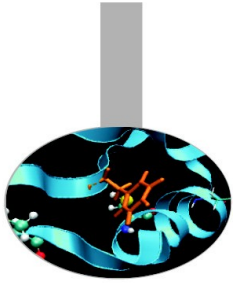
- **libhpc** for linking with non-threaded applications.
- **libhpc_r** for linking with threaded (OpenMP) applications.



HPM library API (C and Fortran version)

- **hpmInit()** for initializing the instrumentation library. The first HPM function called by the application.
- **hpmTerminate()** for generating the reports and performance data files and shutting down the HPM environment. The last HPM function called by the application.
- **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.

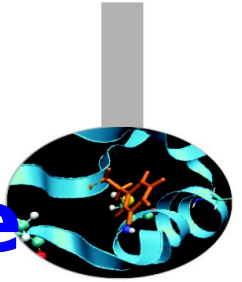
The **hpmStart** and **hpmStop** functions can be inserted as desidered, they must be executed in pairs.
The section identifier label is passed as the parameter to the **hpmStart** and matching **hpmStop** function.



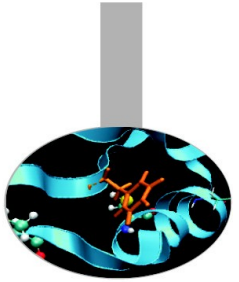
HPM library API -- C example

```
#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();
}
```

HPM library API -- Fortran example



```
#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 1', 22)

call f_hpmterminate()
end program
```



Compiling, Linking, Running

- 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

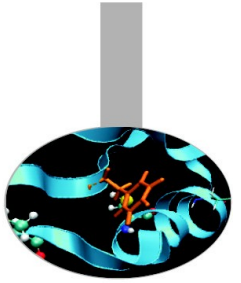
- Compile with **-g**.
- Statically link HPM libraries.
  - non-threated application:

```
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
```

  - threated 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
```

- Run the application as usual.



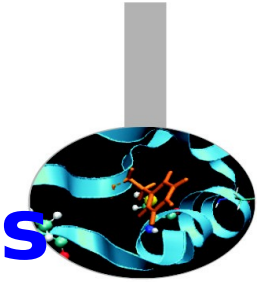
Performance Data Files Naming

The name of the performance data files generated by HPM during hpmTerminate() are:

Name	Type
hpmCounts_<rank>.txt	ASCII
hpmCounts_<rank>.viz	XML for viewing with peekperf

Default:

It will be generated a number of files equal to the number of the MPI tasks involved in the application.



Controlling Performance Data Files

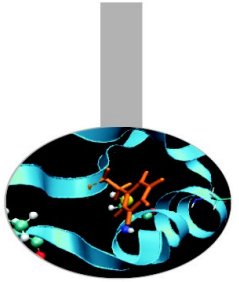
HPM_IO_BATCH = set it to **yes** to reduce the number of output simultaneously opened by HPM in order to reduce file system impact

HPM_OUTPUT_PROCESS = set it to **all** if you want that all the MPI task write performance data files; set it to **root** if you want that only root processor writes 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).

Default:

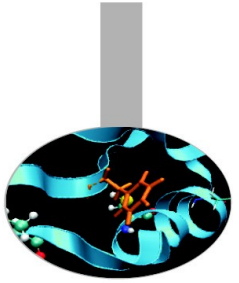
HPM_ASC_OUTPUT = no
HPM_VIZ_OUTPUT = yes
HPM_IO_BATCH = no
HPM_OUTPUT_PROCESS = all
HPM_SCOPE = process



Hardware Counter Event Sets

HPM_EVENT_SET : Environment variable to choose the hardware count events that you want to monitor

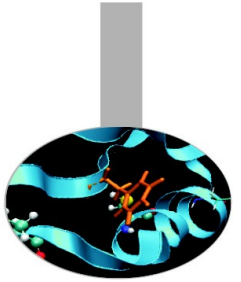
Event Set Number	Description
-1	default setting, corresponds to a basic set of non multiplexed counter
0	multiplexed set that provides information about total cycles, instructions and LSU events
1	multiplexed set to explore branch prediction
2	multiplexed set that presents data about the floating point instruction mix
3	multiplexed set with a mix of different counters
4	multiplexed set for stream pre-fetching events.
5	multiplexed set to investigate pipelining characteristics



Note about HPM library

- HPM libraries collect information and compute summaries during run time.

- Because of this, there could be **overhead** if instrumentation sections are inserted inside inner loops which are executed many times.



Viewing Hardware Performance Counter Data

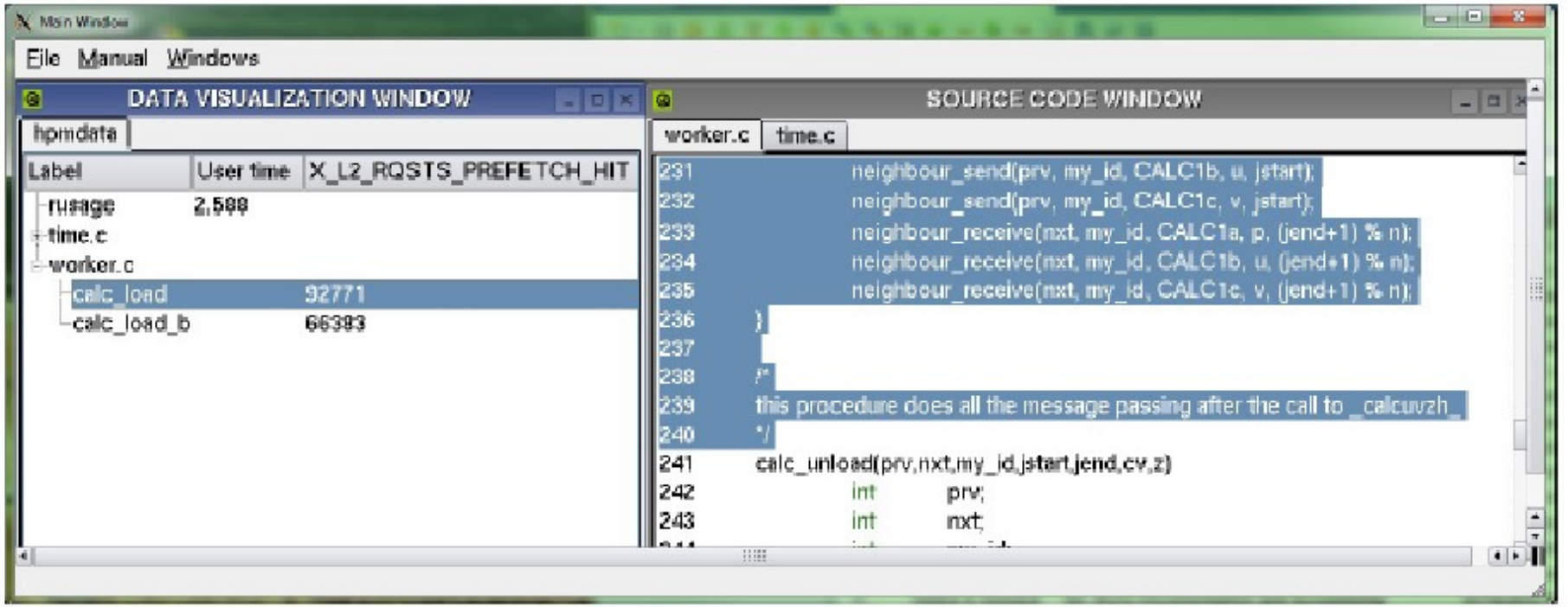
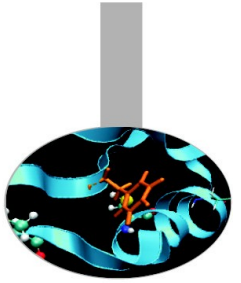


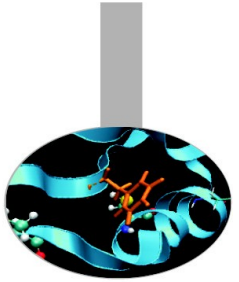
Figure 14: Peekperf viewing hardware performance counter data



MPI Profiling Library

libmpitrace library:

- linked to a MPI application, profiles the MPI function calls, or creates a trace of those MPI calls;
- when an application is linked with such library, the library intercepts the MPI calls in the application, using the Profiled MPI (PMPI) interface defined by the MPI standard, and obtains the profiling and trace information it needs;
- 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



Compiling, Linking, Running

- Set environment variables: run the setup script

  > cd /bgsys/ibmhpc/ppedev.hpct

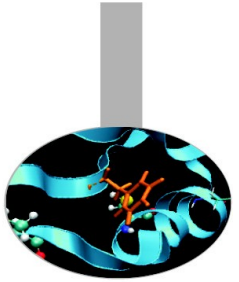
  > . ./env_sh ! for sh, bash,ksh shell

  > source snv_csh ! for csh shell

- Compile with **-g**.
- Statically link **libmpitrace** library.

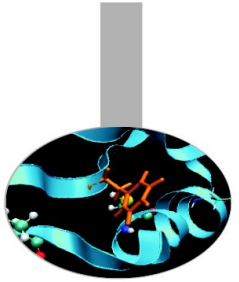
```
mpixlc myprog.c -o myprog
-I/bgsys/ibmhpc/ppedev.hpct/include/ \
-L/bgsys/ibmhpc/ppedev.hpct/lib64 -lmpitrace
```

- Run the application as usual.



Performance Data File Naming

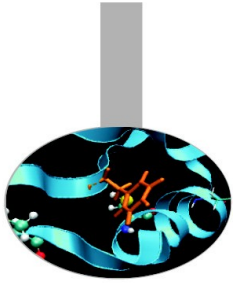
- **mpi_profile_world_id_world_rank** :
  - ***world_id*** is the MPI world id;
  - ***world_rank*** is the MPI task rank of the task that generated the file;
  - If the application doesn't use dynamic tasking, ***world_id*** will be 0 .
- **mpi_profile_world_id_world_rank.viz** :
  - visualization data that can be viewed using peekperf.
- **single_trace_world_id** :
  - trace file containing trace data;
  - can be viewed using peekperf.



Controlling Profiling and Tracing

Default settings:

- number of trace event collected per task = 30000 . (MAX_TRACE_EVENTS).
- it will be gerated only 4 output files: for task 0, and for task 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).



Controlling Profiling and Tracing

MAX_TRACE_EVENTS = max num of trace event collected per task.

MAX_TRACE_RANK = MPI task rank of the highest rank process that has MPI trace events collected. Default is 256.

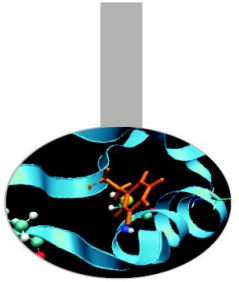
MT_BASIC_TRACE = specifies whether the MAX_TRACE_RANK environment variable is checked. If MT_BASIC_TRACE is set to yes, then MAX_TRACE_RANK is ignored and the trace is generated with less overhead. If MT_BASIC_TRACE is not set, then the setting of MAX_TRACE_RANK is honored.

OUTPUT_ALL_RANKS = Set to yes to generate trace file for all MPI tasks (not only the default 4 trace files).

TRACE_ALL_EVENTS = Set to no if you want that the collection of MPI trace events is controlled by MT_trace_start() and MT_trace_stop().

TRACE_ALL_TASKS = Set to yes to generate MPI trace files for all MPI tasks in the application.

TRACEBACK_LEVEL = Specifies the number of levels to walk back in the function call stack when recording the address of an MPI call. Defaul is 0



Additional Trace Controls

Trace selected sections of an MPI code by bracketing areas of interest with calls

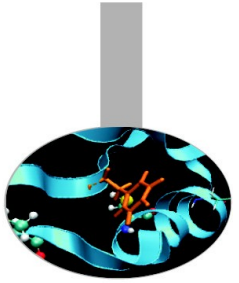
MT_trace_start()

MT_trace_stop()

MT_trace_event(int id)

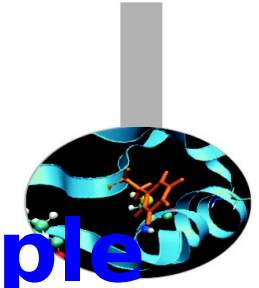
MT_output_trace(int task)

MT_output_text(void)



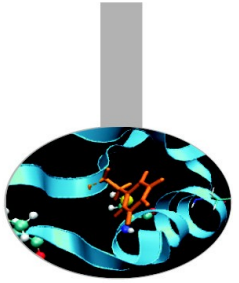
MPI profiling library -- C example

```
#include <mpt.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    MT_trace_start();
    /* MPI communication region of interest */
    MT_trace_stop();

    /* MPI communication region of no interest */

    MPI_Finalize();
}
```



MPI profiling library -- Fortran example

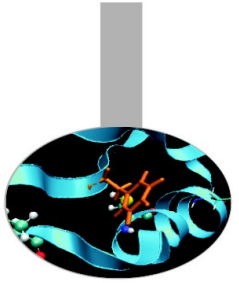
```
program main
include 'mpif.h'

call mpi_init()

call mt_trace_start()
! MPI communication region of interest
call mt_trace_stop()
! MPI communication region of no interest

call mpi_finalize()

end program
```



MPI Profiling Utility Functions 1/2

Functions used to obtain informations about the execution of the application

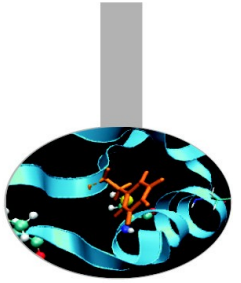
Function	Purpose
MT_get_mpi_counts	How many times an MPI function is called
MT_get_mpi_bytes	Total number of bytes that are transferred by all calls to a specific MPI function
MT_get_mpi_time	Cumulative amount of time spent in all calls to a specific MPI function
MT_get_mpi_name	Obtains the name of an MPI function, given the internal ID that is used by the IBM HPC Toolkit to refer to this MPI function
MT_get_time	Elapsed time since MPI_Init was called
MT_get_elapsed_time	Elapsed time between calls to MPI_Init and MPI_Finalize



MPI Profiling Utility Functions 2/2

Functions used to obtain informations about the execution of the application

Function	Purpose
MT_get_environments	Obtains information about the MPI execution environment
MT_get_allresults	Obtains statistical information about a specific MPI function call
MT_get_tracebufferinfo	Size and current usage of the internal MPI trace buffer that is used by the IBM HPC Toolkit
MT_get_calleraddress	Address of the caller of a currently-active MPI function
MT_get_callerinfo	Source file and line number information for an MPI function call, using the address that is obtained by calling MT_get_calleraddress



Viewing MPI Profiling Data

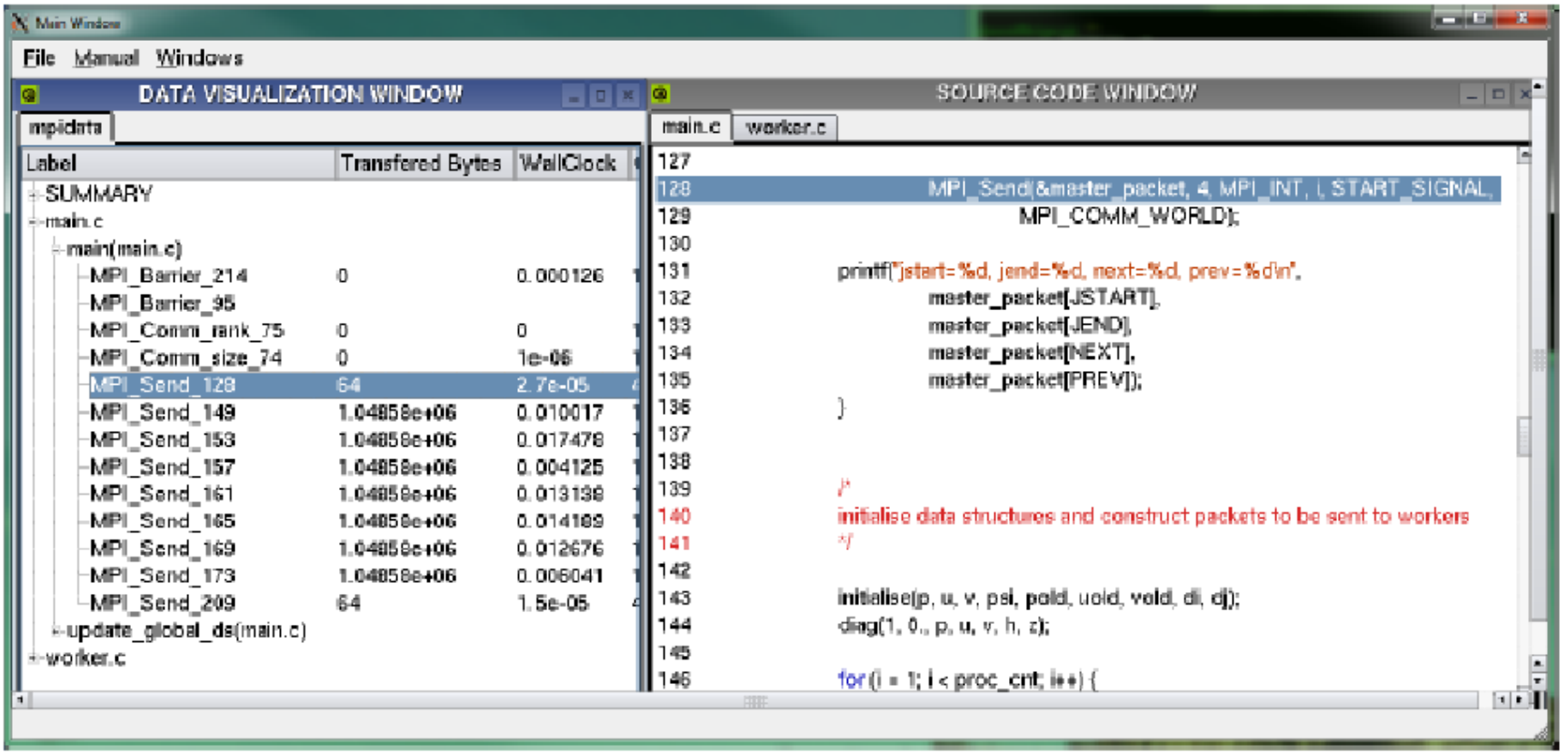
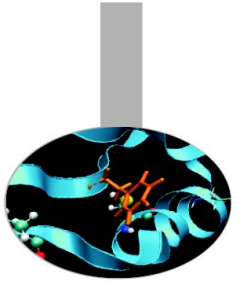


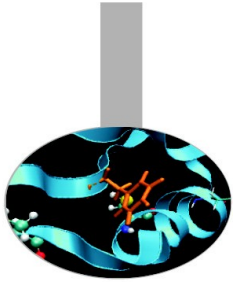
Figure 16: Peekperf MPI profiling data visualization window



OpenMP Profiling Library

OpenMP Profiling library can be used:

- to analyze performance problems in an OpenMP application
- to help in determining if the OpenMP application investigated properly structures its processing for best performance
- to obtain information about
  - time spent in OpenMP constructs in the application
  - overhead in OpenMP constructs
  - information about how workload is balanced across OpenMP threads in the application



Compiling, Linking, Running

- Set environment variables: run the setup script

  > cd /bgsys/ibmhpc/ppedev.hpct

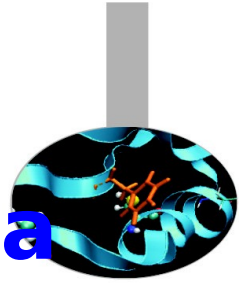
  > . ./env_sh ! for sh, bash,ksh shell

  > source snv_csh ! for csh shell

- Compile with **-g**.
- Statically link with **libxlsmp_pomp** and **libpompprof_probe** 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
```

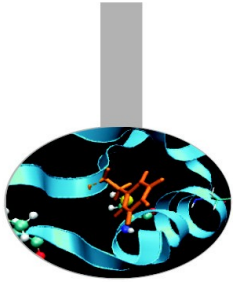
- Run the application as usual.



OpenMP Profiling Performance Data

When running your application after linking with OpenMP profiling, performance measurements such as time in OpenMP thread, time in master thread, computation time and load imbalance percentage are collected for the threads running in individual regions and loops.

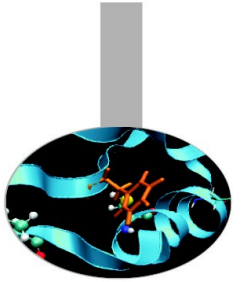
Sections of your application where measurements were gathered are labeled with type of OpenMP construct and the starting line number of the construct.



Performance Data Files Naming

The name of the performance data files generated by OpenMP Profiling library are:

Name	Type
pompprof_<rank>	ASCII
pompprof_<rank>.viz	XML for viewing with peekperf



Viewing OpenMP Profiling Data

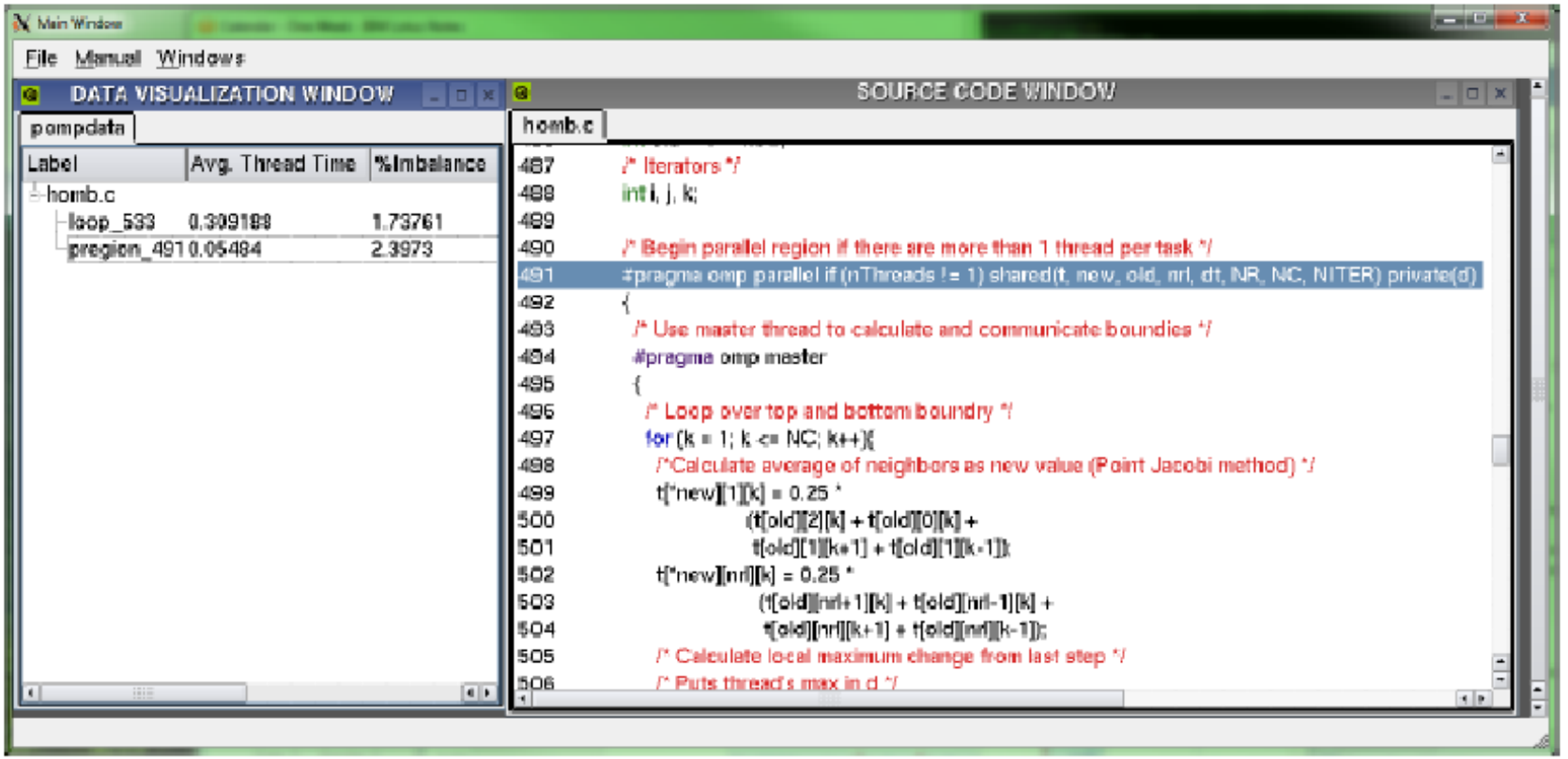
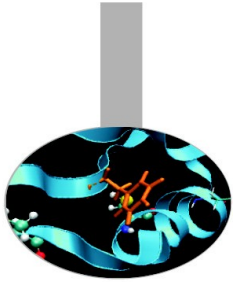
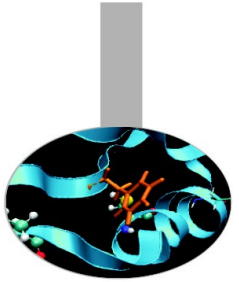


Figure 21: Peekperf OpenMP data visualization window



Outline

- Profiling
  - Scalasca
  - IBM® High Performance Computing (HPC) Toolkit
  - GNU Profiler – Gprof
- Debugging
  - GDB
  - addr2line
  - Totalview



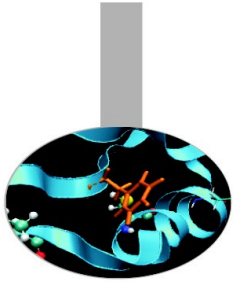
GNU Profiler – Gprof

/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gprof

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

gprof can produce several different 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 used when its subroutine calls are included.



Gprof – Flat profile

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

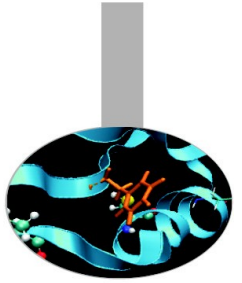
Note that if a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total 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

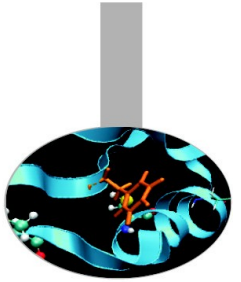
...



Gprof – Call Graph

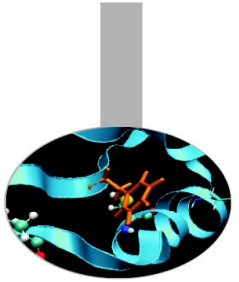
The **call graph** shows how much time was spent in each function and its children. From this information, you can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time.

```
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]
-----------------------------------------------
```



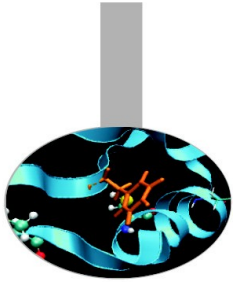
Compiling, Running, Output files

- Compile and link the program with options: **-g -pg -qfullpath**
- Profiling files in execution directory
  - **gmon.out.<MPI Rank>** = binary files, not readable
  - The number of files depends on environment variable
    - 1 Profiling File / Process: The default setting is to generate gmon.out files only for profiling data collected on ranks 0 - 31.
    - **BG_GMON_RANK_SUBSET = N** -- Only generate the gmon.out file for rank N.
    - **BG_GMON_RANK_SUBSET = N:M** -- Generate gmon.out files for all ranks from N to M.
    - **BG_GMON_RANK_SUBSET = N:M:S** -- Generate gmon.out files for all ranks from N to M. Skip S; 0:16:8 generates gmon.out.0, gmon.out.8, gmon.out.16
- Output files interpretation
  - gprof <executable> gmon.out.<MPI Rank> > gprof.out.<MPI Rank>

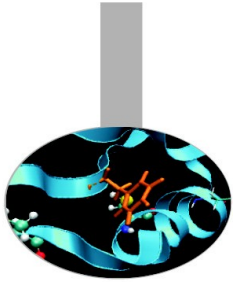


Using GNU profiling – Threads

- The base GNU toolchain does not provide support for profiling on threads
- Profiling threads
  - **BG_GMON_START_THREAD_TIMERS**
    - Set this environment variable to “all” to enable the SIGPROF timer on all threads created with the pthread_create() function.
    - "nocomm" to enable the SIGPROF timer on all threads except the extra threads that are 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 from the thread to be profiled: 1 to start, 0 to stop



DEBUGGING

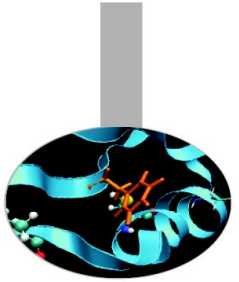


Debugging on FERMI…

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

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

- 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



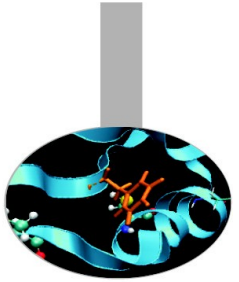
Compiling for a debug session

Three 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

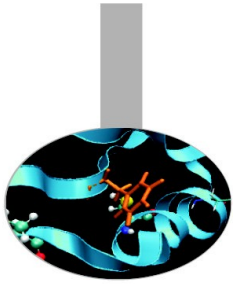
-qcheck Helps detecting some array-bound violations, aborting with SIGTRAP at runtime

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

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



FERMI compiling tools

GDB

GDB
The GNU Project Debugger

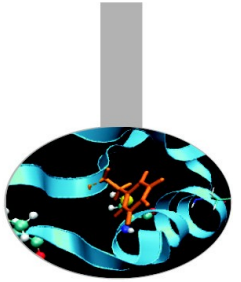
addr2line

Totalview

TOTALVIEW
TECHNOLOGIES

GDB

GDB
The GNU Project Debugger



- 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>`

- It is possible to make a post-mortem analysis of the **binary** core files generated by the job

  `/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gdb <exe> <corefile>`

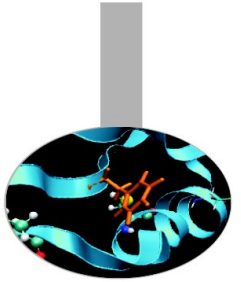
- To generate binary core filed, add the following envs to runjob:

  `--envs BG_COREDUMPONEXIT=1`

  `--envs BG_COREDUMPBINARY=*`

  '*' means "all the processes". It is possible to indicate which ranks generate its core by specifying its number

GDB – remote access

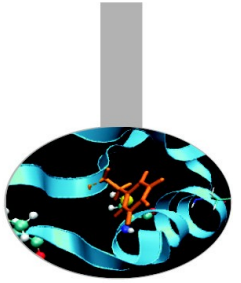


The Blue Gene/Q system includes support for using GDB real-time with applications running on compute nodes.

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 4 GDB tools can be run on one job.

...so, how to do that?



Using GDB on running applications

1) Submit your job as usual

```
llsubmit <jobscript>
```

2) Get your job ID

```
llq -u $USER
```

3) Use it for getting the BG Job ID

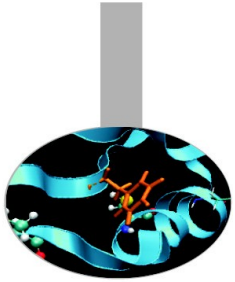
```
llq -l <jobID> | grep “Job Id”
```

4) Start the gdb-server tool

```
start_tool --tool
/bgsys/drivers/ppcfloor/ramdisk/distrofs/cios/sbin/gdbtool
--args “--rank=<rank #> --listen_port=10000” --id <BG Job ID>
```

5) Get the IP address for your process

```
dump_proctable --id <BG Job ID> --rank <rank #> --host sn01-io
```



Using GDB on running applications

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

```
/bgsys/drivers/ppcfloor/gnu-linux/bin/
powerpc64-bgq-linux-gdb ./myexe
```

7) Connect remotely to your job process;

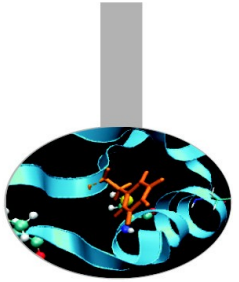
```
(gdb) target remote <IP address>:10000
```

8) Start debugging!!!

```
where   --->   up to #
```

Although you aren't completely free...for example, command 'run' does not work

addr2line

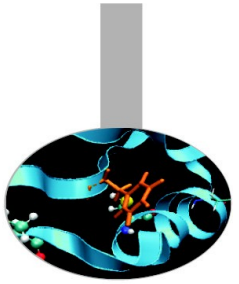


If nothing is specified, an unsuccesful job generates a text core file for the processes that caused the crash...
...however, those core files are all but easily readable!

```
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.0
+++LCB 1.0
Program   : deadlock.exe
Job ID    : 96550
Personality:
   ABCDET coordinates : 0,0,0,0,0,3
   Rank               : 3
   Ranks per node     : 4
   DDR Size (MB)      : 16384
+++ID Rank: 3, TGID: 337, Core: 12, HWTID:0 TID: 337 State: RUN
***FAULT Encountered unhandled signal 0x00000009 (9) (???)
While executing instruction at..........0x00000000011f009c
Dereferencing memory at.................0x0000000000000000
Tools attached (list of tool ids).......None
Currently running on hardware thread....Y
General Purpose Registers:
  r00=00000000010dbef8 r01=0000001fffff9860 r02=00000000015b2cc0 r03=0000000000000000 r04=0000000000000001 r05=0000001fffff98d0
r06=0000000000000000 r07=0000001fffff95a0
  r08=0000000001649160 r09=0000000300900020 r10=0000000000000000 r11=0000001f00a00020 r12=0000000024000222 r13=0000001f00707700
r14=0000000000000000 r15=0000000000000000
  r16=0000000000000000 r17=0000000000000000 r18=0000000000000000 r19=0000000000000000 r20=0000000000000001 r21=0000000000000000
r22=0000001f00728848 r23=0000000000000001
  r24=0004000000000000 r25=0000000000000000 r26=00000000015f8ff8 r27=0000000000000001 r28=0000000000000000 r29=0000000000000000
r30=0000000000000000 r31=0000001f007326e0
Special Purpose Registers:
  lr=00000000011f0130 cr=0000000044004222 xer=0000000000000000 ctr=000000000102a7a4
  msr=000000008002f000 dear=0000000000000000 esr=0000000000000000 fpscr=0000000000004000
  sprg0=0000000000000000 sprg1=0000000000000000 sprg2=0000000000000000 sprg3=0000000000000000 sprg4=0000000000000000
  sprg5=0000000000000000 sprg6=000000000056e200 sprg7=0000000000000000 sprg8=0000000000000000
  srr0=00000000011f009c 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 1000008800200019  0000000000000000 0000000000000000  f01=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f02=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f03=0000000000000000 0000000000000000  0000000000000000 0000000100000000
```

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

Core files



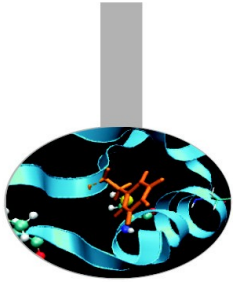
Blue Gene core files are lightweight text files

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 1/2



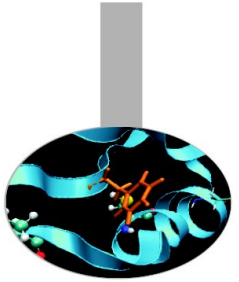
From the core file output, save only the addresses in the Saved Link Reg column:

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

Replace the first eight 0s with 0x:

00000000018b2678 => 0x018b2678

using addr2line 2/2



To replace the first eight 0s with 0x:

00000000018b2678 => 0x018b2678

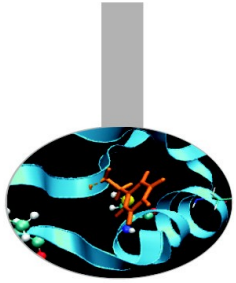
there is a easy way:

```
module load superc
a2l-tanslate core.<num>
```

The file with all the addresses **core.<num>.t0** will be created

Lauch addr2line:

```
addr2line –e ./myexe 0x018b2678
addr2line –e ./myexe < core.<num>.t0
```

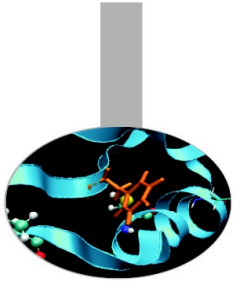


Totalview

- 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.
- Latest releast 8.11.0, available on FERMI
- Running a Totalview execution in back-end can be a bit tricky, as it requires connection from FERMI to your local machine via ssh tunneling to VNC server.

TOTALVIEW TECHNOLOGIES

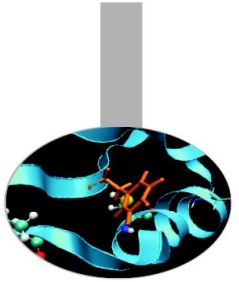
Using Totalview: preliminaries



In order to use Totalview, first you need to have downloaded and installed VNCviewer on your local machine.
(http://www.realvnc.com/download/viewer/)

Windows users will also find useful Cygwin, a Linux-like environment for Windows. During installation, be sure to select “openSSH” from the list of available packages.
(http://cygwin.com/setup.exe)

Once all the required softwares are installed, we are ready to start preparing our Totalview session!



Using Totalview: preparation

1) On FERMI, load tightvnc module;

```
module load tightvnc
```

2) Execute the script vncserver_wrapper;

```
vncserver_wrapper
```

3) Instructions will appear. Copy/paste to your local machine (Cygwin shell if Windows) this line from those instructions:

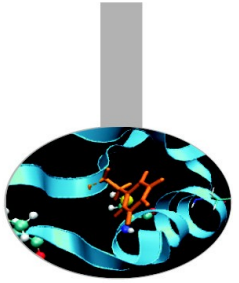
```
ssh -L 59xx:localhost:59xx -L 58xx:localhost:58xx –N
<username>@login<no>.fermi.cineca.it
```

where **xx** is your VNC display number, and **<no>** is the number of the front-end node you're logged into (01, 02, 07 or 08)

4) Open VNCViewer. On Linux, use another local shell and type:

```
vncviewer localhost:xx
```

On Windows, double click on VNCviewer icon and write localhost:xx when asked for the server. Type your VNC password (or choose it, if it's your first visit)



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 the above slide (you'll find the correct **DISPLAY** name to export in vncserver_wrapper instructions)

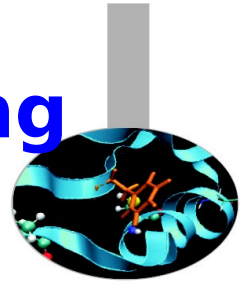
6) Totalview execution line (inside your LoadLeveler script) will be as follows:

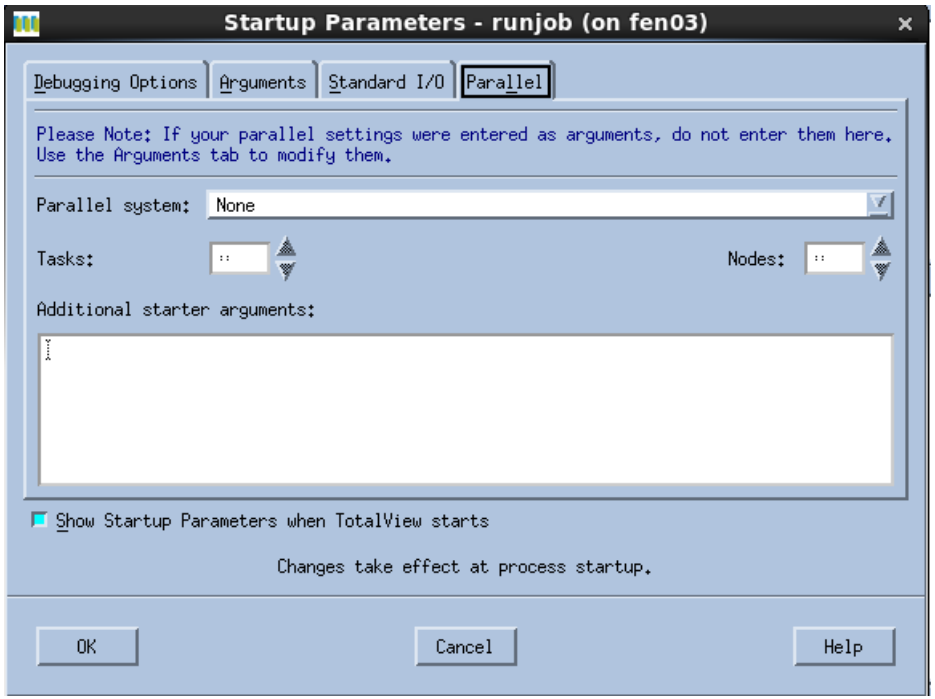
```
totalview runjob -a <runjob arguments…>
```

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

Closing Totalview will also kill the job.

Using Totalview: start debugging



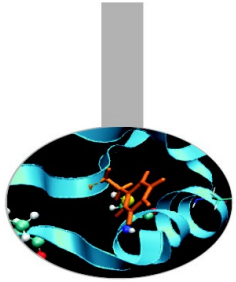


Select “BlueGene” as a parallel system, and a number of tasks and nodes according to the arguments you gave to runjob during submission phase.

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

User Guide for Totalview:

```
module load totalview
module show totalview

$MANPATH :
/cineca/prod/tools/totalview/8.11.0-0/binary/toolworks/totalview.8.11.
0-0/doc/pdf
```

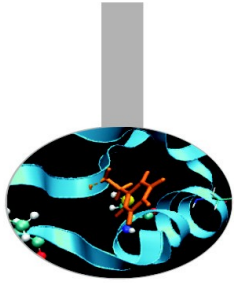


Using Totalview: licenses

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

You can visualize the **usage status of the licenses** by typing the command:

```
module load totalview

lmstat -c $LM_LICENSE_FILE -a
```



Out from Totalview

When you've finished using Totalview, please follow this procedure in order to close the session safely:

1) Close VNCviewer on your local machine;

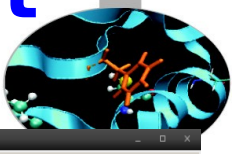
2) Kill the VNCserver on FERMI:

```
vncserver kill :x
```

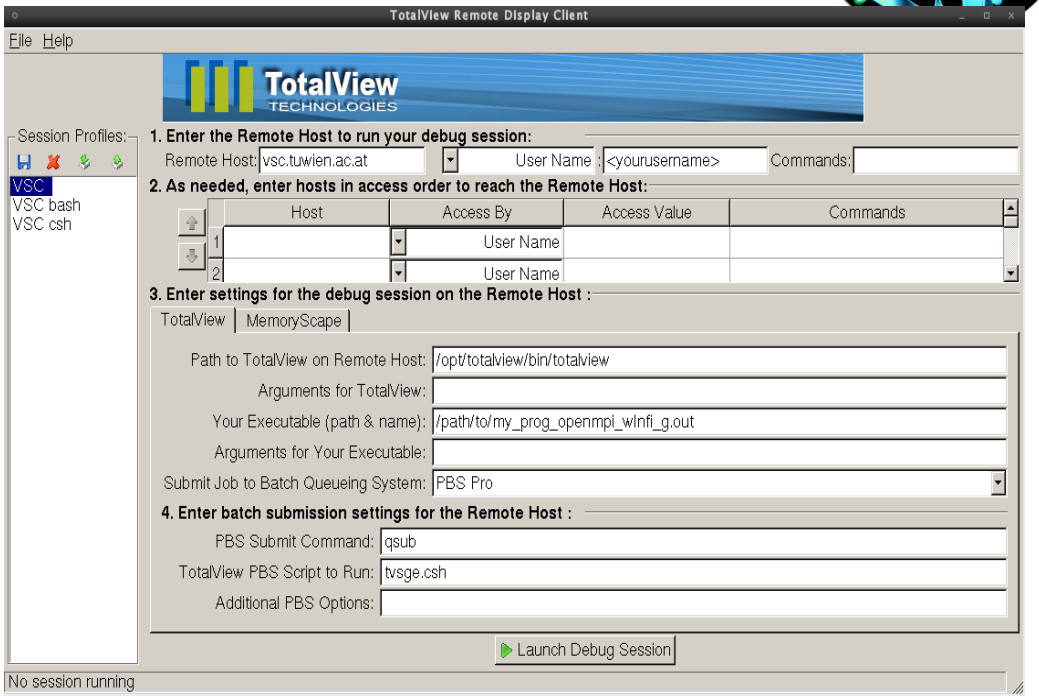
x is the usual VNC display number, without the initial 0 (if present);

3) On your first local shell, close the ssh tunneling connection with CTRL+C.

Totalview Remote Display Client

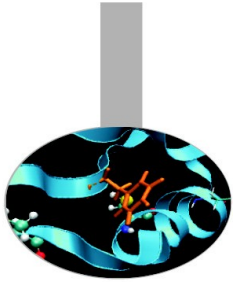


An easier (and maybe safer) way to use Totalview is Totalview **RDC** (**R**emote **D**isplay **C**lient), a simple tool that helps with submitting a job already setted with the proper characteristics (and with no VNC involved)



RDC procedure isn't fully operative yet, since we encountered some firewall issues that lead to different behaviours depending on the single workstation settings.

Our System Administrators are looking into it.
Connecting with RDC will be soon a possibility!!




CINECA
KEEP
CALM
AND
USE THE
DEFAULT