Profiling and Debugging on a Blue Gene/Q system

Giusy Muscianisi – g.muscianisi@cineca.it
SuperComputing Applications and Innovation Department

Feb 05, 2013
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 release 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

mpixlc -c foo.c

mpixlf90 -openmp -o bar bar.f90

SCALASCA instrumentation command

scalasca -instrument mpixlc -c foo.c

skin mpixlf90 -openmp -o bar bar.f90
#!/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>
#!/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>
• Pure MPI:

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

```text
==> epik_<myexe>_2p256_sum
``` 

• MPI + OpenMP:

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

```text
==> epik_<myexe>_4p256x4_sum
```
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

- Topology controls toolbar (enable via 'Topology' menu)
- 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)

Questions:
- 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)
## Examination by text - summary.cube

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>4769825184</td>
<td>2340248.68</td>
<td>100.00</td>
<td>(summary) ALL</td>
</tr>
<tr>
<td>MPI</td>
<td>4593960</td>
<td>613242.18</td>
<td>26.20</td>
<td>(summary) MPI</td>
</tr>
<tr>
<td>COM</td>
<td>348528</td>
<td>212240.56</td>
<td>9.07</td>
<td>(summary) COM</td>
</tr>
<tr>
<td>USR</td>
<td>4764882648</td>
<td>1514021.98</td>
<td>64.69</td>
<td>(summary) USR</td>
</tr>
<tr>
<td>USR</td>
<td>2839795200</td>
<td>184264.39</td>
<td>7.87</td>
<td>PrimToChar</td>
</tr>
<tr>
<td>USR</td>
<td>188793792</td>
<td>34584.14</td>
<td>1.48</td>
<td>SoundSpeed2</td>
</tr>
<tr>
<td>USR</td>
<td>148475712</td>
<td>41108.12</td>
<td>1.76</td>
<td>Flux</td>
</tr>
<tr>
<td>USR</td>
<td>91241568</td>
<td>32795.35</td>
<td>1.40</td>
<td>PrimToCons</td>
</tr>
<tr>
<td>USR</td>
<td>82213824</td>
<td>41377.87</td>
<td>1.77</td>
<td>ConsToPrim</td>
</tr>
<tr>
<td>USR</td>
<td>74237856</td>
<td>26645.11</td>
<td>0.97</td>
<td>CT_StoreEMF</td>
</tr>
<tr>
<td>USR</td>
<td>74237856</td>
<td>370444.42</td>
<td>15.83</td>
<td>Roe_Solver</td>
</tr>
<tr>
<td>USR</td>
<td>40318080</td>
<td>21342.57</td>
<td>0.91</td>
<td>CTU_CT_Source</td>
</tr>
<tr>
<td>USR</td>
<td>40318080</td>
<td>6966.30</td>
<td>0.30</td>
<td>CheckPrimStates</td>
</tr>
<tr>
<td>USR</td>
<td>40318080</td>
<td>4172.12</td>
<td>0.18</td>
<td>PrimSource</td>
</tr>
<tr>
<td>USR</td>
<td>40318080</td>
<td>203264.59</td>
<td>8.69</td>
<td>CharTracingStep</td>
</tr>
<tr>
<td>USR</td>
<td>40318080</td>
<td>279464.91</td>
<td>11.94</td>
<td>States</td>
</tr>
<tr>
<td>USR</td>
<td>40318080</td>
<td>16575.16</td>
<td>0.71</td>
<td>CheckNaN</td>
</tr>
<tr>
<td>MPI</td>
<td>4343328</td>
<td>16560.02</td>
<td>0.71</td>
<td>MPI_Sendrecv</td>
</tr>
<tr>
<td>USR</td>
<td>1765632</td>
<td>335.38</td>
<td>0.01</td>
<td>Init</td>
</tr>
<tr>
<td>COM</td>
<td>241296</td>
<td>337.55</td>
<td>0.01</td>
<td>AL_Exchange_dim</td>
</tr>
<tr>
<td>USR</td>
<td>196608</td>
<td>22.27</td>
<td>0.00</td>
<td>Length_1</td>
</tr>
<tr>
<td>USR</td>
<td>196608</td>
<td>23.28</td>
<td>0.00</td>
<td>Length_3</td>
</tr>
<tr>
<td>USR</td>
<td>196608</td>
<td>22.77</td>
<td>0.00</td>
<td>Length_2</td>
</tr>
<tr>
<td>USR</td>
<td>194064</td>
<td>14.31</td>
<td>0.00</td>
<td>print1</td>
</tr>
<tr>
<td>USR</td>
<td>175296</td>
<td>26.61</td>
<td>0.00</td>
<td>SetIndexes</td>
</tr>
<tr>
<td>USR</td>
<td>131472</td>
<td>136.96</td>
<td>0.01</td>
<td>ResetState</td>
</tr>
</tbody>
</table>

### Notes
- **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-program routines 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
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Total CPU allocation time</td>
</tr>
<tr>
<td>Visits</td>
<td>Number of times a routine/region was executed</td>
</tr>
<tr>
<td>Synchronizations</td>
<td>Total number of MPI synchronization operations that were executed</td>
</tr>
<tr>
<td>Communications</td>
<td>The total number of MPI communication operations, excluding calls transferring no data (which are considered Synchronizations)</td>
</tr>
<tr>
<td>Bytes transferred</td>
<td>The total number of bytes that were sent and received in MPI communication operations. It depends on the MPI internal implementation.</td>
</tr>
</tbody>
</table>

[http://www2.fz-juelich.de/jsc/datapool/scalasca/scalasca_patterns-1.4.html](http://www2.fz-juelich.de/jsc/datapool/scalasca/scalasca_patterns-1.4.html)
## Metrics 2/2

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI file operations</strong></td>
<td>Number of MPI file operations of any type.</td>
</tr>
<tr>
<td><strong>MPI file bytes transferred</strong></td>
<td>Number of bytes read or written in MPI file operations of any type.</td>
</tr>
<tr>
<td><strong>Computational imbalance</strong></td>
<td>This simple heuristic allows to identify computational load imbalances and is calculated for each (call-path, process/thread) pair.</td>
</tr>
</tbody>
</table>

http://www2.fz-juelich.de/jsc/datapool/scalasca/scalasca_patterns-1.4.html
Metrics - Time, pure MPI code 1/2
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>Total CPU allocation time</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>Execution time without overhead</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>Time spent in tasks related to measurement (not including dilation from instrumentation!)</td>
</tr>
<tr>
<td><strong>MPI</strong></td>
<td>Time spent in pre-instrumented MPI functions</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td>Time spent in MPI communication calls, subdivided into collective and point-to-point</td>
</tr>
<tr>
<td><strong>Synchronization</strong></td>
<td>Time spent in calls to MPI_Barrier()</td>
</tr>
<tr>
<td><strong>File I/O</strong></td>
<td>Time spent in MPI file I/O functions</td>
</tr>
<tr>
<td><strong>Init/Exit</strong></td>
<td>Time spent in MPI_Init() and MPI_Finalize()</td>
</tr>
</tbody>
</table>
### Pure MPI code - Communications

- **Collective**
  - Exchange
  - As Source
  - As Destination

- **Point-to-point**
  - Sends
  - Receives

- 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

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td>Time spent for all OpenMP-related tasks</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Time spent synchronizing OpenMP threads</td>
</tr>
<tr>
<td>Fork</td>
<td>Time spent by master thread to create thread teams</td>
</tr>
<tr>
<td>Flush</td>
<td>Time spent in OpenMP flush directives</td>
</tr>
<tr>
<td>Idle Threads</td>
<td>Time spent idle on CPUs reserved for slave threads</td>
</tr>
</tbody>
</table>
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 substitute automatic instrumentation, and can improve the structure of analysis reports to make them more readily 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 desirable 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++

```c
#include "epik_user.h"
...
void foo(){
    ...
    // local declarations
    ...
    // more declarations
    EPIK_FUNC_START();
    ...
    // executable statements
    if(...){
        EPIK_FUNC_END();
        return;
    }
    ...
    // executable statements
    } 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

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 desired, 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

```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();
}
```
HPM library API -- Fortran example

```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 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-threaded application:
    ```
    mpixlc myprog.c -o myprog -I/bgsys/ibmhpc/ppedev.hpct/include/ \ 
    -L/bgsys/drivers/ppcfior/bgpm/lib/ \ 
    -L/bgsys/ibmhpc/ppedev.hpct/lib64 -lhpc -lbglm
    ```
  - threaded application:
    ```
    mpixlc_r myprog.c -o myprog_r
    -I/bgsys/ibmhpc/ppedev.hpct/include/ \ 
    -L/bgsys/drivers/ppcfior/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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpmCounts_&lt;rank&gt;.txt</td>
<td>ASCII</td>
</tr>
<tr>
<td>hpmCounts_&lt;rank&gt;.viz</td>
<td>XML for viewing with peekperf</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Event Set Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>default setting, corresponds to a basic set of non multiplexed counter</td>
</tr>
<tr>
<td>0</td>
<td>multiplexed set that provides information about total cycles, instructions and LSU events</td>
</tr>
<tr>
<td>1</td>
<td>multiplexed set to explore branch prediction</td>
</tr>
<tr>
<td>2</td>
<td>multiplexed set that presents data about the floating point instruction mix</td>
</tr>
<tr>
<td>3</td>
<td>multiplexed set with a mix of different counters</td>
</tr>
<tr>
<td>4</td>
<td>multiplexed set for stream pre-fetching events.</td>
</tr>
<tr>
<td>5</td>
<td>multiplexed set to investigate pipelining characteristics</td>
</tr>
</tbody>
</table>
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 generated 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_TRACEEVENTS** = 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. Default 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

```c
#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

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT_get_mpi_counts</td>
<td>How many times an MPI function is called</td>
</tr>
<tr>
<td>MT_get_mpi_bytes</td>
<td>Total number of bytes that are transferred by all calls to a specific MPI function</td>
</tr>
<tr>
<td>MT_get_mpi_time</td>
<td>Cumulative amount of time spent in all calls to a specific MPI function</td>
</tr>
<tr>
<td>MT_get_mpi_name</td>
<td>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</td>
</tr>
<tr>
<td>MT_get_time</td>
<td>Elapsed time since MPI_Init was called</td>
</tr>
<tr>
<td>MT_get_elapsed_time</td>
<td>Elapsed time between calls to MPI_Init and MPI_Finalize</td>
</tr>
</tbody>
</table>
## MPI Profiling Utility Functions 2/2

Functions used to obtain informations about the execution of the application

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT_get_environments</td>
<td>Obtains information about the MPI execution environment</td>
</tr>
<tr>
<td>MT_get_allresults</td>
<td>Obtains statistical information about a specific MPI function call</td>
</tr>
<tr>
<td>MT_get_tracebufferinfo</td>
<td>Size and current usage of the internal MPI trace buffer that is used by the IBM HPC Toolkit</td>
</tr>
<tr>
<td>MT_get_calleraddress</td>
<td>Address of the caller of a currently-active MPI function</td>
</tr>
<tr>
<td>MT_get_callerinfo</td>
<td>Source file and line number information for an MPI function call, using the address that is obtained by calling MT_get_calleraddress</td>
</tr>
</tbody>
</table>
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.
The name of the performance data files generated by OpenMP Profiling library are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>pompprof_&lt;rank&gt;</td>
<td>ASCII</td>
</tr>
<tr>
<td>pompprof_&lt;rank&gt;.viz</td>
<td>XML for viewing with peekperf</td>
</tr>
</tbody>
</table>
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

/gbsys/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:

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.34</td>
<td>0.02</td>
<td>0.02</td>
<td>7208</td>
<td>0.00</td>
<td>open</td>
</tr>
<tr>
<td>16.67</td>
<td>0.03</td>
<td>0.01</td>
<td>244</td>
<td>0.04</td>
<td>offtime</td>
</tr>
<tr>
<td>16.67</td>
<td>0.04</td>
<td>0.01</td>
<td>8</td>
<td>1.25</td>
<td>memccpy</td>
</tr>
<tr>
<td>16.67</td>
<td>0.05</td>
<td>0.01</td>
<td>7</td>
<td>1.43</td>
<td>write</td>
</tr>
<tr>
<td>16.67</td>
<td>0.06</td>
<td>0.01</td>
<td>236</td>
<td>0.00</td>
<td>mcount</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>192</td>
<td>0.00</td>
<td>tzset</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>47</td>
<td>0.00</td>
<td>toLOWER</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>45</td>
<td>0.00</td>
<td>strlen</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>mcount</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>memccpy</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>print</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>profi</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>report</td>
</tr>
</tbody>
</table>
```
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

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

• 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

-00 : 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

-qfltttrap 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

addr2line

Totalview
• 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 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 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-bqg-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 unsuccessful job generates a text core file for the processes that caused the crash...
...however, those core files are all but easily readable!

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
0000001fffffff5ac0  00000000000000001c
0000001fffffff5bc0  000000000018b2678
0000001fffffff5c60  00000000015046d0
0000001fffffff5d00  00000000015738a8
0000001fffffff5e00  00000000015734ec
0000001fffffff5f00  000000000151a4d4
0000001fffffff6000  00000000015001c8
---STACK
```

In particular, “Saved Link Reg” column is the one we need!
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 \Rightarrow 0x018b2678
To replace the first eight 0s with 0x:

0000000018b2678 => 0x018b2678

there is an 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 release 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.
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:

```bash
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:

```bash
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.
An easier (and maybe safer) way to use Totalview is Totalview RDC (Remote Display Client), 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!!
KEEP CALM AND USE THE DEFAULT