Optimisation Tools
Overview

- Timers
- Profilers
- HPM Toolkit
Timers

- Wide range of timers available on the HPCx system

- Varying
  - precision
  - portability
  - language
  - ease of use
## Timers

<table>
<thead>
<tr>
<th>Timer</th>
<th>Usage</th>
<th>Wallclock/CPU</th>
<th>Resolution</th>
<th>Language</th>
<th>Portable</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>shell</td>
<td>both</td>
<td>centisecond</td>
<td>F, C/C++</td>
<td>yes</td>
</tr>
<tr>
<td>timex</td>
<td>shell</td>
<td>both</td>
<td>centisecond</td>
<td>F, C/C++</td>
<td>yes</td>
</tr>
<tr>
<td>gettimeofday</td>
<td>subroutine</td>
<td>wallclock</td>
<td>microsecond</td>
<td>C/C++</td>
<td>yes</td>
</tr>
<tr>
<td>read_real_time</td>
<td>subroutine</td>
<td>wallclock</td>
<td>nanosecond</td>
<td>C/C++</td>
<td>no</td>
</tr>
<tr>
<td>rtc</td>
<td>subroutine</td>
<td>wallclock</td>
<td>microsecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>irtc</td>
<td>subroutine</td>
<td>wallclock</td>
<td>nanosecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>dtime_</td>
<td>subroutine</td>
<td>CPU</td>
<td>centisecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>etime_</td>
<td>subroutine</td>
<td>CPU</td>
<td>centisecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>mclock</td>
<td>subroutine</td>
<td>CPU</td>
<td>centisecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>timef</td>
<td>subroutine</td>
<td>wallclock</td>
<td>millisecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>MPI_Wtime</td>
<td>subroutine</td>
<td>wallclock</td>
<td>microsecond</td>
<td>F, C/C++</td>
<td>yes</td>
</tr>
<tr>
<td>system_clock</td>
<td>subroutine</td>
<td>wallclock</td>
<td>centisecond</td>
<td>F</td>
<td>yes</td>
</tr>
</tbody>
</table>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

struct timeval start_time, end_time;

main() {
    int total;

    gettimeofday(&start_time, (struct timeval*)0);
    /* code to be timed */
    gettimeofday(&end_time, (struct timeval*)0);

    total = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
            (end_time.tv_usec-start_time.tv_usec);

    printf("Total time %d microseconds \n", total);
}
**system_clock - Fortran**

```fortran
integer :: clock0, clock1, clockmax, clockrate, ticks
real    :: secs

call system_clock(count_max=clockmax,
                   count_rate=clockrate)

call system_clock(clock0)
!    code to be timed

call system_clock(clock1)

ticks = clock1 - clock0
!    reset negative numbers

ticks = mod(ticks + clockmax, clockmax)
secs = float(ticks) / float(clockrate)

print*, 'Total time ', secs, ' seconds'
```
irtc - Fortran

- May have used irtc previously on the Cray

```fortran
integer(8) :: start, end, irtc

start = irtc()
! code to be timed
end = irtc()

print*,' Total time ', end - start, 'nanoseconds'
```
Profiler

• Investigate which parts of the program are responsible for most of the execution time
  - Program counter sampling
    • The program is interrupted at regular intervals (0.01s) and the location of the program counter (PC) is recorded
    • From these records the amount of time spent in each routine can be estimated
    • For stable results the runtime of the code has to be long compared to the sampling interval
  - Invocation counting
    • Compiler inserts a call to `mcount` each time a subroutine or function is invoked
Standard profiler: prof

- Compile and link your application with the option -p
  
xlf90 -qsuffix=f=f90 -p -c my_prog.f90
  
xlf90 -p -o my_prog.x myProg.o

- Run your application on the backend, using loadleveler
  
  - This will generate a file: mon.out

- prof is available on most UNIX systems

- Examine the output
  
  prof my_prog.x |more

- Remark: When compiling with mpxlf or mpxlc, the run will generate a file: mon.out.0, examine:
  
  prof my_prog.x -m mon.out.0 |more
### Sample output from prof

#### Relative time

<table>
<thead>
<tr>
<th>Name</th>
<th>%Time</th>
<th>Seconds</th>
<th>Cumsecs</th>
<th>#Calls</th>
<th>msec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>__image_work_MOD_ma</td>
<td>64.5</td>
<td>36.02</td>
<td>36.02</td>
<td>2000</td>
<td>18.010</td>
</tr>
<tr>
<td>__image_work_MOD_re</td>
<td>31.2</td>
<td>17.40</td>
<td>53.42</td>
<td>2000</td>
<td>8.700</td>
</tr>
<tr>
<td>__hpf_random_modul</td>
<td>1.2</td>
<td>0.66</td>
<td>54.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FormatControl</td>
<td>0.8</td>
<td>0.46</td>
<td>54.54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgmwrite</td>
<td>0.4</td>
<td>0.24</td>
<td>54.78</td>
<td>1</td>
<td>240.</td>
</tr>
<tr>
<td>_xlfWriteFmt</td>
<td>0.4</td>
<td>0.24</td>
<td>55.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WriteUnit</td>
<td>0.4</td>
<td>0.20</td>
<td>55.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memmove</td>
<td>0.3</td>
<td>0.18</td>
<td>55.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IOWrite</td>
<td>0.3</td>
<td>0.16</td>
<td>55.56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FmtLongIntToDec</td>
<td>0.1</td>
<td>0.06</td>
<td>55.62</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_fill</td>
<td>0.1</td>
<td>0.06</td>
<td>55.68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__image_work_MOD_ed</td>
<td>0.1</td>
<td>0.05</td>
<td>55.73</td>
<td>1</td>
<td>50.</td>
</tr>
<tr>
<td>__image_mpi_comm_MO</td>
<td>0.1</td>
<td>0.05</td>
<td>55.78</td>
<td>1</td>
<td>50.</td>
</tr>
<tr>
<td>_xlfWriteFmtArray</td>
<td>0.1</td>
<td>0.04</td>
<td>55.82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__mcount</td>
<td>0.0</td>
<td>0.01</td>
<td>55.83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__hpf_random_modul</td>
<td>0.0</td>
<td>0.01</td>
<td>55.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__image_main</td>
<td>0.0</td>
<td>0.00</td>
<td>55.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__image_work_MOD_pr</td>
<td>0.0</td>
<td>0.00</td>
<td>55.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__image_work_MOD_up</td>
<td>0.0</td>
<td>0.00</td>
<td>55.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__time_tool_MOD_now</td>
<td>0.0</td>
<td>0.00</td>
<td>55.84</td>
<td>2000</td>
<td>0.</td>
</tr>
<tr>
<td>__locale_init</td>
<td>0.0</td>
<td>0.00</td>
<td>55.84</td>
<td>6001</td>
<td>0.</td>
</tr>
<tr>
<td>__time_init</td>
<td>0.0</td>
<td>0.00</td>
<td>55.84</td>
<td>6001</td>
<td>0.</td>
</tr>
</tbody>
</table>

#### Absolute time

- **Average time/call**
- **Number of Calls**
- **Name of function or subroutine**
- **Relative time**
- **Absolute time**
 Comments on prof

• If mcount appears high on the list
  - Significant time spent in subroutine calls
  - Write longer routines or recompile with --qinline

• Profiling code with inlining needs care
  - Time spent in inlined routines will be attributed to mother routine

• Flat profile of prof provides no information on mother routine
  - Would be useful for routines called from different places
  - To obtain call tree use gprof or xprofiler, see below
Call graph profiling with gprof

- Compile and link your application with the option -p
  `xlf90 -qsuffix=f=f90 -pg -c my_prog.f90`
  `xlf90 -pg -o my_prog.x my_prog.o`

- Note: -pg is not -p -g

- Run your application on the backend, using loadleveler
  - This will generate a file: gmon.out

- Examine the output
  `gprof my_prog.x gmon.out |more`
Sample output from gprof

<table>
<thead>
<tr>
<th>index</th>
<th>%time</th>
<th>self descendents</th>
<th>called/total</th>
<th>parents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>96.0</td>
<td>0.00</td>
<td>53.41</td>
<td>1/1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>53.41</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>53.10</td>
<td>2000/2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.04</td>
<td>0.22</td>
<td>1/1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.05</td>
<td>0.00</td>
<td>1/1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>6001/6001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
</tr>
</tbody>
</table>

6.6s

<table>
<thead>
<tr>
<th>index</th>
<th>%time</th>
<th>self descendents</th>
<th>called/total</th>
<th>parents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>96.0</td>
<td>0.00</td>
<td>53.41</td>
<td>1/1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>53.41</td>
<td>1</td>
</tr>
</tbody>
</table>

6.6s

<table>
<thead>
<tr>
<th>index</th>
<th>%time</th>
<th>self descendents</th>
<th>called/total</th>
<th>parents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>95.5</td>
<td>0.00</td>
<td>53.10</td>
<td>2000/2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>53.10</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>36.06</td>
<td>0.00</td>
<td>2000/2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.04</td>
<td>0.00</td>
<td>2000/2000</td>
</tr>
</tbody>
</table>

6.6s

<table>
<thead>
<tr>
<th>index</th>
<th>%time</th>
<th>self descendents</th>
<th>called/total</th>
<th>parents</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>64.8</td>
<td>36.06</td>
<td>0.00</td>
<td>2000/2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>36.06</td>
<td>0.00</td>
<td>2000</td>
</tr>
</tbody>
</table>
IBM specific profiler: xprofiler

- **xprofiler** reads the same input data as **gprof**
  - Compile and link with `-pg` and run to obtain `gmon.out`

- **Key differences to gprof**
  - Graphical user interface (gui)
  - Profiling at source statement level
    - This needs compiler option `-g` in addition to `-pg`
    - Needs care when interpreting results

- **Starting xprofiler**
  - `xprofiler my_prog.x gmon.out &`
The xprofiler main window

- **xprofiler starts with its main window**
- **Often extremely busy**
  - Includes calls to libraries
- **For not too complex applications choose:**
  - **Filter**
  - **Hide all Library Calls**
Hidden library calls

- After hiding the library calls, you are left with a cluster box, containing your executable

- To un-cluster:
  - Filter
  - Uncluster Functions
Call tree of your application

- Functions and subroutines are represented as green “function boxes”
  - Width: time spend in function and daughters
  - Height: time spend in function only
- Calls are represented by blue “call arcs”
Getting details of a function

- A left-click on a function box reveals the information box.
- The information box details:
  - Name of function or subroutine
  - Its index
  - Time spent in itself and daughters
  - Time spent in itself
  - Number of invocations
Getting details of a call arc

- A left-click on a call arc reveals its information box

- The information box details:
  - Name of caller
  - Name of the callee
  - Their indices
  - Number of times the caller called the callee

```
<Press rightmost mouse button for Arc Menu>
```
Most time consuming functions

- To profile complex applications, it is advisable to start with the most time consuming functions
  - Filter
  - Filter by CPU Time ...
  - Select the number of functions you want to see
- The example picture is when selecting the two high scorers from the previous example
Getting the function menu

• Using the function menu we can build up a call tree
• A click with the right mouse button on the green function box brings up the “function menu”
• To add the calling routine to the tree choose:
  • Immediate Parents
Building up a call tree

- This will add the calling routine

- In our example the two routines of the previous screen are daughters of the same routine. xprofiler puts arcs from the caller to both of them
Building up to the main routine

- This process might be iterated to build up to the main routine

- The option “All Paths To” from the function menu (right click) build this at once

- If this leads to a busy picture, use “Undo” from the Filter menu
Profiling options

• The Function Menu offers
  - Disassembler code profiling
  - Source code profiling

• For source code profiling, application needs to be compiled with the –g option
  - This disables –qin1ine, bad for performance of OO-codes (e.g. C++)

• The results of both options have to be read with care
  - It often picks the wrong instruction/line in the vicinity of the hot spot
### Disassembler code profiling

**Ticks/Instructions**

<table>
<thead>
<tr>
<th>Address</th>
<th>No. Ticks</th>
<th>Instruction</th>
<th>Assembler Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>10002584</td>
<td>1</td>
<td>C1A7FFFC lfs</td>
<td>13,0xffffc(7)</td>
</tr>
<tr>
<td>10002588</td>
<td>3</td>
<td>D3E6FFF0 stfs</td>
<td>31,0xffff0(6)</td>
</tr>
<tr>
<td>1000258C</td>
<td></td>
<td>E0FDE02A fadds</td>
<td>0x1f,0x1d,0x1c</td>
</tr>
<tr>
<td>10002590</td>
<td>257</td>
<td>C3C7CB74 lfs</td>
<td>30,0xcb74(7)</td>
</tr>
<tr>
<td>10002594</td>
<td></td>
<td>EF830032 fmuls</td>
<td>0x1c,0x3,0</td>
</tr>
<tr>
<td>10002598</td>
<td></td>
<td>C367E5C0 lfs</td>
<td>27,0xe5c0(7)</td>
</tr>
<tr>
<td>1000259C</td>
<td></td>
<td>C347E5C8 lfs</td>
<td>26,0xe5c8(7)</td>
</tr>
<tr>
<td>100025A0</td>
<td>211</td>
<td>EFA53028 fsbs</td>
<td>0x1d,0x5,0x6</td>
</tr>
<tr>
<td>100025A4</td>
<td></td>
<td>C0680008 lfs</td>
<td>3,0x8(8)</td>
</tr>
<tr>
<td>100025A8</td>
<td></td>
<td>C0A70000 lfs</td>
<td>5,0(7)</td>
</tr>
<tr>
<td>100025AC</td>
<td></td>
<td>ECC1382A fadds</td>
<td>0x8,0x1,0x7</td>
</tr>
<tr>
<td>100025B0</td>
<td>726</td>
<td>ECE2202A fadds</td>
<td>0x7,0x2,0x4</td>
</tr>
<tr>
<td>100025B4</td>
<td></td>
<td>EC99C02A fadds</td>
<td>0x4,0x19,0x18</td>
</tr>
<tr>
<td>100025B8</td>
<td></td>
<td>39080010 cal</td>
<td>8,0x10(8)</td>
</tr>
<tr>
<td>100025BC</td>
<td></td>
<td>C047CB78 lfs</td>
<td>2,0xcb78(7)</td>
</tr>
<tr>
<td>100025C0</td>
<td>225</td>
<td>FC20C090 fmr</td>
<td>1,24</td>
</tr>
<tr>
<td>100025C4</td>
<td></td>
<td>ED080032 fmuls</td>
<td>0x8,0x8,0</td>
</tr>
<tr>
<td>100025C8</td>
<td></td>
<td>D126FFFF stfs</td>
<td>9,0xffff4(6)</td>
</tr>
<tr>
<td>100025CC</td>
<td></td>
<td>C127E5CC lfs</td>
<td>9,0xe5cc(7)</td>
</tr>
<tr>
<td>100025D0</td>
<td>276</td>
<td>38E70010 cal</td>
<td>7,0x10(7)</td>
</tr>
<tr>
<td>100025D4</td>
<td></td>
<td>ED6B6028 fsbs</td>
<td>0xb,0xb,0xc</td>
</tr>
</tbody>
</table>

**xprofiler ticked fast floating point instructions (e.g. fadds) instead to the slow loads (lfs)**
The report menu

- **Flat Profile** similar to prof
- **Call Graph Profile** similar to gprof
- **Function Index**
- **Function Call Summary**, which lists the routines by their call counts
- **Library Statistics**, a flat profile listing the time spend in the different libraries
The HPM Toolkit

- Hardware event counters were originally invented to assist the development of the processor hardware.

- They proved valuable tools for the performance optimisation of applications.

- On HPCx the Hardware Performance Monitor Toolkit is available to read the counters:
  - Use HPMCOUNT for a global analysis.
  - Use LIBHPM to analyse code segments (instrumentation).
  - HPMVIZ is a simple GUI to read the output from LIBHPM.
Event sets

- The individual events of the Toolkit are grouped into 61 sets

- Within a single run only one event set can be investigated

- To enquire about the raw counters available within the event sets use
  
  /usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount -l

- Event sets may contain derived metrics
Floating point operations

- Use event set **60** (default) to count:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float divide (hardware)</td>
<td></td>
</tr>
<tr>
<td>Fused multiply-add</td>
<td></td>
</tr>
<tr>
<td>Operation count on FPU</td>
<td></td>
</tr>
<tr>
<td>Float stores</td>
<td></td>
</tr>
<tr>
<td>Number of instructions</td>
<td></td>
</tr>
<tr>
<td>Float loads</td>
<td></td>
</tr>
</tbody>
</table>

- The derived metrics include:

<table>
<thead>
<tr>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point instructions + FMAs (flips)</td>
</tr>
<tr>
<td>Flip rates (Wall clock and User time) <em><strong>VERY USEFUL</strong></em></td>
</tr>
<tr>
<td>FMA percentage</td>
</tr>
<tr>
<td>Flips/load-store</td>
</tr>
</tbody>
</table>
TLB and level 1 cache

- **Use event set 56 to count:**

<table>
<thead>
<tr>
<th>Data TLB misses</th>
<th>Instruction TLB misses</th>
<th>L1 cache load misses</th>
<th>L1 cache store misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cycles</td>
<td>Number of instructions</td>
<td>L1 data store references</td>
<td>L1 data load references</td>
</tr>
</tbody>
</table>

- **The derived metrics include:**

<table>
<thead>
<tr>
<th>TLB miss rate (cycles and loads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction rates (loads, cycles, wall clock)</td>
</tr>
</tbody>
</table>

- **Rem:** Due to prefetching and multiple load units on the Power4 chip, the L2 traffic cannot be calculated from L1 misses. The derived metrics referring to the L2 will be renamed in future releases.
Memory, L2 and L3 cache

• **Nomenclature of cache levels and locations:**

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 2</td>
<td>Level 2 on same chip</td>
</tr>
<tr>
<td>Level 2.5</td>
<td>Level 2 on different chip, same MCM</td>
</tr>
<tr>
<td>Level 2.75</td>
<td>Level 2 on different MCM, inaccessible (phase 1)</td>
</tr>
<tr>
<td>Level 3</td>
<td>Level 3 on same MCM</td>
</tr>
<tr>
<td>Level 3.5</td>
<td>Level 3 on different MCM, inaccessible (phase 1)</td>
</tr>
</tbody>
</table>

• **Rem:** For parallel codes, if you use shared memory (e.g. OpenMP) or shared memory MPI inside the logical partition, you will note level 2.5 traffic.
Memory, L2 and L3 (continued)

• Use event set 5 to count:

<table>
<thead>
<tr>
<th>Loads from memory</th>
<th>Loads from L3</th>
<th>Loads from L3.5</th>
<th>Loads from L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads fr. L2.5</td>
<td>Loads fr. L2.5</td>
<td>Loads L2.75</td>
<td>Loads L2.75</td>
</tr>
<tr>
<td>'read only'</td>
<td>'exclusive'</td>
<td>'read only'</td>
<td>'exclusive'</td>
</tr>
</tbody>
</table>

• The derived metrics include:

| Total loads from L2 and L3 |
| Traffic, bandwidth and miss rates for L2, L3 and Memory |

• Rem: Interpretation needs care, see streams example below
Global analysis with HPMCOUNT

- HPMCOUNT investigates the performance of the entire application
  - Pros:
    - Very easy to use
    - No need to modify the source
  - Cons:
    - No information on which part of the code performs poorly
    - Several rates are calculated with respect to Wallclock time. Due to overheads these rates can be severely distorted when running short test jobs
- Executable:
  /usr/local/packages/actc/hpmtk/pwr4/bin/hpmcount
Using HPMCOUNT

• For code compiled for serial execution (xlf, xlc)
  
  `<path>/hpmcount -g <set> my_prog.x`

• For code compiled for MPI (e.g. mpxlf, mpxlc)
  
  `poe <path>/hpmcount -g <set> my_prog.x`

• **Pitfall:** poe and the order of poe and `<path>/hpmcount` is crucial. Otherwise HPMCOUNT examines the performance of poe.
Options of HPMCOUNT

- **Getting help**
  - `-h`

- **Specifying an output file (default standard out)**
  - `-o <outfile>`

- **Specifying the event set (default set 60)**
  - `-g <set>`

- **Listing the available event sets**
  - `-l`
Investigating code segments

- Use LIBHPM to investigate the performance if individual code segments

- Can exclude the effect from input, output and initialisation routines, allowing for reliable results from short test runs

- This requires instrumentation of the codes with a few simple routines
Header files

• Every file containing any of the LIBHPM instrumentation calls needs to contain a header file:

  - FORTRAN
    
    #include “f_hpm.h”
    • All FORTRAN instrumentation carries an “f_” prefix
    • LIBHPM is not standard FORTRAN and needs a preprocessor
    • Some of the commands are case sensitive!

  - C
    
    #include “libhpm.h”
Initialising and closing the tool

- Initialise the tool with
  
  \[
  f\_hpminit(\text{taskid}, \text{name})
  \]
  
  \[
  \text{hpmInit(\text{taskid}, \text{name})}
  \]
  
  - \text{taskid}: identifies the MPI-task (put 0 for serial)
  - \text{name}: will become part of the *.viz file

- Close the tool with
  
  \[
  f\_hpmterminate(\text{taskid})
  \]
  
  \[
  \text{hpmTerminate(\text{taskid})}
  \]
  
  - \text{taskid}: put same as for the hpmInit call

- Forgetting \text{hpmTerminate}: no output file
Starting and stopping the counters

- Start the counters with
  - `f_hpmstart(instid, label)`
  - `hpmStart(instid, label)`
    - `instid`: unique number identifying the section, $0 \leq \text{instid} \leq 100$ (default)
    - `label`: string
- Stop the counters with
  - `f_hpmstop(instid)`
  - `hpmStop(instid)`
    - `instid`: unique number identifying the section, has to match the argument of `hpmStart`
- Use `hpmTstart` and `hpmTstop` in threaded regions
Comments on hpmstart & hpmstop

• Measured sections can be nested

• Both calls hpmstart & hpmstop take about 10µs
  - Instrumented regions have to be longer than several 100µs for the rates quoted among the derived metrics to become reliable
A simple example in C

#include <stdio.h>
#include <stdlib.h>

#include "libhpm.h"

main(int argc, char *argv[]){
    hpmInit(0, "hpm_hello");
    hpmStart(1, "print_count");
    printf("Hello world!\n");
    hpmStop(1);
    hpmTerminate(0);
}

• Example measures the performance of printf

• The measured region gets an identifier 1 and the label "print_cont"

• The name of *.viz file will include the string “hpm_hello”
Compiling instrumented FORTRAN

- To invoke the preprocessor on HPCx use the -qsuffix option of xlf
  -qsuffix=cpp=f
  -qsuffix=cpp=f90

- Example: Compiling FORTRAN90
  xlf90_r -qsuffix=cpp=f90 \
  -o my_prog.x my_prog.f90 \
  -I/usr/local/packages/actc/hpmtk/include \n  -L/usr/local/packages/actc/hpmtk/pwr4/lib \n  -lhpm -lpmapi

- For OpenMP replace -lhpm with -lhpm_r, add -qsmp=omp
Compiling instrumented C

- Compiling C
  
  \texttt{xlc\_r -o my\_prog.x my\_prog.c \-L/usr/local/packages/actc/hpmtk/include \-L/usr/local/packages/actc/hpmtk/pwr4/lib \-lhpm -lpmapi -lm}

- For OpenMP replace \texttt{-lhpm} with \texttt{-lhpm\_r} and add \texttt{-qsmp=omp}
Environment variables

- **Use HPM_EVENT_SET** to select the event set:
  
  ```
  export HPM_EVENT_SET=5
  ```

- **HPM_OUTPUT_NAME** changes the name of the output files. It overwrites the argument of hpmInit. To get output files which include the event set specify the following in your script:
  
  ```
  export HPM_EVENT_SET=5
  export HPM_OUTPUT_NAME=hpm_set${HPM_EVENT_SET}
  ```
Visualisation of the output

• LIBHPM generates two output files per MPI-task
  1. *.hpm
  2. *.viz

• Both files contain the same data

• The *.hpm file is a plain ASCII file. Investigate with UNIX tools such as more, grep, awk

• Use HPMVIZ to read the *.viz file
Example for an *.hpm file
• HPMVIZ is a simple visualisation tools to investigate the output from LIBHPM

• Starting HPMVIZ
  
  /usr/local/packages/actc/hpmtk/pwr4/hpmviz &

• You may list the *.viz file(s) before the &
HPMVIZ main window

List of instrumented sections
• Left click for source code
• Right click for metrics window

Source code display
HPMVIZ file window

- Use the file window for loading *.viz files
HPMVIZ file window detailed view

- Click top right hand corner to get detailed view
- Allows sorting with respect to date (very useful with parallel jobs)
HPMVIZ metrics window

- Right click on section in main window
- Set precision (double or single)
- Highlights results it thinks are good or poor

Good result

Poor results
### HPMVIZ metrics options

- By default, the metrics window of HPMVIZ shows the derived metrics of the group only.

- Use the Metrics Options pull down to activate raw counters and deactivate derived metrics as needed.

<table>
<thead>
<tr>
<th>Metrics Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
</tr>
<tr>
<td>ExcSec</td>
</tr>
<tr>
<td>IncSec</td>
</tr>
<tr>
<td>PM_FPU_FDIV</td>
</tr>
<tr>
<td>PM_FPU_FMA</td>
</tr>
<tr>
<td>PM_FPU0_FIN</td>
</tr>
<tr>
<td>PM_FPU1_FIN</td>
</tr>
<tr>
<td>PM_CYC</td>
</tr>
<tr>
<td>PM_FPU_STF</td>
</tr>
<tr>
<td>PM_INST_CMPL</td>
</tr>
<tr>
<td>PM_LSU_LDF</td>
</tr>
<tr>
<td>Usec</td>
</tr>
<tr>
<td>User</td>
</tr>
<tr>
<td>(M)LS</td>
</tr>
<tr>
<td>I/LS</td>
</tr>
<tr>
<td>MIPS</td>
</tr>
<tr>
<td>IpC</td>
</tr>
<tr>
<td>hwmIp/c</td>
</tr>
<tr>
<td>MFlops</td>
</tr>
<tr>
<td>Mflop/s</td>
</tr>
<tr>
<td>MFlop/Us</td>
</tr>
<tr>
<td>FMA %</td>
</tr>
<tr>
<td>Cmp int</td>
</tr>
</tbody>
</table>
Case study:
Streams benchmark copy

<table>
<thead>
<tr>
<th></th>
<th>L2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array (doubles)</td>
<td>17E3</td>
<td>70E6</td>
</tr>
<tr>
<td>Stores (set 60)</td>
<td>17E3</td>
<td>70E6</td>
</tr>
<tr>
<td>Loads (set 60)</td>
<td>17E3</td>
<td>70E6</td>
</tr>
<tr>
<td>L1-Store (set 56)</td>
<td>17E3</td>
<td>70E6</td>
</tr>
<tr>
<td>L1-Load (set 56)</td>
<td>17E3</td>
<td>70E6</td>
</tr>
<tr>
<td>L1-S-miss (set 56)</td>
<td>17E3</td>
<td>70E6</td>
</tr>
<tr>
<td>L1-L-miss (set 56)</td>
<td>0.15E3</td>
<td>0.66E6</td>
</tr>
<tr>
<td>Loads from L2 (5)</td>
<td>1.1E3</td>
<td>3.5E6</td>
</tr>
<tr>
<td>Loads from L3 (5)</td>
<td>0</td>
<td>130</td>
</tr>
<tr>
<td>Loads from Mem (5)</td>
<td>0</td>
<td>0.9E6</td>
</tr>
</tbody>
</table>

Fitting into L2: no loads from L3 and Memory

Code fitting on Memory: most loads still from L2 (prefetching)
2nd Case Study: Sum of an array

- No simple relation between L1 misses and loads
  - Prefetching
  - Two LSU

- Stride larger than two cache lines: all loads triggered by L1 misses and data loaded from memory
Further reading

• General
  - User Guide to the HPCx Service
    www.hpcx.ac.uk/support/introduction

• xprofiler
  - IBM Parallel Environment for AIX
    Operation and use, Volume 2
    www.hpcx.ac.uk/support/documentation/IBMdocuments/a2274261.pdf

• HPM Toolkit
  - Hardware Performance Monitor (HPM) Toolkit
    www.hpcx.ac.uk/support/documentation/IBMdocuments/HPM.html
  - Using the Hardware Performance Monitor Toolkit on HPCx
    www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0307_choose.html