Understanding and Using Profiling Tools on Seaborg

Richard Gerber
NERSC User Services
ragerber@nersc.gov
510-486-6820
Overview

• What is Being Measured?
• POWER 3 Hardware Counters
• Available Tools
• Interpreting Output
• Theoretical Peak MFlops
• Simple Optimization Considerations
What is Being Measured?

• The Power 3 processor has counters in hardware on the chip.
  – E.g. cycles used, instructions completed, data moves to and from registers, floating point unit instructions executed.

• The tools discussed here read the hardware counters.

• These tools know nothing about MPI or other communication performance issues.
  – VAMPIR (http://hpcf.nersc.gov/software/tools/vampir.html)
  – tracempi (http://hpcf.nersc.gov/software/tools/sptools.html#trace_mpi)

• Xprofiler, gprof can give CPU time spent in functions
  – (http://hpcf.nersc.gov/software/ibm/xprofiler/)
Profiling Tools

- The tools discussed here are simple & basic ones that use the POWER 3 hardware counters to profile code.
- There are more sophisticated tools available, but have a steeper learning curve.
- See the PERC website for more:
  - [http://perc.nersc.gov/](http://perc.nersc.gov/)
- Also see the ACTS toolkit web site:
  - [http://acts.nersc.gov](http://acts.nersc.gov)
POWER 3 Hardware Counters

- Power 3 has 2 FPUs, each capable of an FMA
- Power 3 has 8 hardware counters
- 4 event sets (see `hpmcount -h`)

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Instructions Completed</th>
<th>TLB Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stores Completed</td>
<td>Loads Completed</td>
<td></td>
</tr>
<tr>
<td>FPU0 ops</td>
<td>FPU1 ops</td>
<td>FMAs executed</td>
</tr>
</tbody>
</table>

Default Event Set
Performance Profiling Tools
PAPI

- Standard application programming interface (API)
- Portable, don’t confuse with IBM low-level PMAPI interface
- User program can read hardware counters
- See
  - http://hpcf.nersc.gov/software/papi.html
The hpmcount Utility

• Easy to use; no need to recompile code…
• BUT, must compile with `-qarch=pwr3 (-O3+)`
• Minimal effect on code performance
• Profiles entire code
• Reads hardware counters at start and end of program
• Reports flip (floating point instruction) rate and many other quantities
How to Use hpmcount

• To profile serial code
  - `%hpmcount executable`

• To profile parallel code
  - `%poe hpmcount executable -nodes n -procs np`

• Reports performance numbers for each task
• Prints output to STDOUT (or use `--o filename`)
• Beware! These profile the `poe` command
  - `%hpmcount poe executable`
  - `%hpmcount executable` (if compiled with mp* compilers)
[Declarations]...
!********************************************************************
! Initialize variables
!********************************************************************

Z=0.0
CALL RANDOM_NUMBER(X)
CALL RANDOM_NUMBER(Y)

DO J=1,N
   DO K=1,N
      DO I=1,N
         Z(I,J) = Z(I,J) + X(I,K) * Y(K,J)
      END DO
   END DO
END DO

[Finish up] ...
% xlf90 -o xma_hpmcount -O2 -qarch=pwr3 ma_hpmcount.F
% hpmcount ./xma_hpmcount

hpmcount (V 2.4.3) summary
Total execution time (wall clock time): 4.200000 seconds

PM_CYC (Cycles) : 1578185168
PM_INST_CMPL (Instructions completed) : 3089493863
PM_TLB_MISS (TLB misses) : 506952
PM_ST_CMPL (Stores completed) : 513928729
PM_LD_CMPL (Loads completed) : 1025299897
PM_FPU0_CMPL (FPU 0 instructions) : 509249617
PM_FPU1_CMPL (FPU 1 instructions) : 10006677
PM_EXEC_FMA (FMAs executed) : 515946386

Utilization rate : 98.105 %
   TLB misses per cycle : 0.032 %
Avg number of loads per TLB miss : 2022.479
Load and store operations : 1539.229 M
MIPS : 599.819
Instructions per cycle : 1.632
HW Float points instructions per Cycle : 0.329
Floating point instructions + FMAs : 1035.203 M
Floating point instructions + FMA rate : 240.966 Mflip/s
FMA percentage : 99.680 %
Computation intensity : 0.673
The poe+ Utility

- By default, `hpmcount` writes separate output for each parallel task.
- `poe+` is a utility written by NERSC to gather & summarize `hpmcount` output for parallel programs.
- `poe+` combines all `hpmcount` output and outputs one summary report to STDOUT.
How to Use poe+

- `%poe+ executable -nodes n -procs np`
  - Prints aggregate number to STDOUT

- **Do not do these!**
  - `hpmcount poe+ executable ...
  - `hpmcount executable` (if compiled with mp* compiler)

- See `man poe+` on Seaborg

- In a batch script, just use this on the command line
  - `poe+ executable`
### % poe+ ./xma_hpmcount -nodes 1 -procs 16

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average execution time (wall clock time)</td>
<td>4.46998 seconds</td>
</tr>
<tr>
<td>Total maximum resident set size</td>
<td>120 Mbytes</td>
</tr>
<tr>
<td>PM_CYC (Cycles)</td>
<td>25173734104</td>
</tr>
<tr>
<td>PM_INST_CMPPL (Instructions completed)</td>
<td>41229695424</td>
</tr>
<tr>
<td>PM_TLB_MISS (TLB misses)</td>
<td>8113100</td>
</tr>
<tr>
<td>PM_ST_CMPPL (Stores completed)</td>
<td>8222872708</td>
</tr>
<tr>
<td>PM_LD_CMPPL (Loads completed)</td>
<td>16404831574</td>
</tr>
<tr>
<td>PM_FPU0_CMPPL (FPU 0 instructions)</td>
<td>8125215690</td>
</tr>
<tr>
<td>PM_FPU1_CMPPL (FPU 1 instructions)</td>
<td>182898872</td>
</tr>
<tr>
<td>PM_EXEC_FMA (FMAs executed)</td>
<td>8255207322</td>
</tr>
<tr>
<td>Utilization rate</td>
<td>84.0550625 %</td>
</tr>
<tr>
<td>Avg number of loads per TLB miss</td>
<td>2022.0178125</td>
</tr>
<tr>
<td>Load and store operations</td>
<td>24627.712 M</td>
</tr>
<tr>
<td>Avg instructions per load/store</td>
<td>1.84</td>
</tr>
<tr>
<td>MIPS</td>
<td>9134.331</td>
</tr>
<tr>
<td>Instructions per cycle</td>
<td>1.63775</td>
</tr>
<tr>
<td>HW Float points instructions per Cycle</td>
<td>0.3300625</td>
</tr>
<tr>
<td>Total Floating point instructions + FMAs</td>
<td>16563.28 M</td>
</tr>
<tr>
<td>Total Float point instructions + FMA rate</td>
<td>3669.55 Mflip/s (= 408 / task)</td>
</tr>
<tr>
<td>Average FMA percentage</td>
<td>99.68 %</td>
</tr>
<tr>
<td>Average computation intensity</td>
<td>0.673</td>
</tr>
</tbody>
</table>
Using HPMLIB

- HPM library can be used to instrument code sections
- Embed calls into source code
  - Fortran, C, C++
- Access through the `hpmtoolkit` module
  - `%module load hpmtoolkit`
- Compile with `$HPMTOOLKIT` env variable
  - `%xlf -qarch=pwr3 -O2 source.F \ $HPMTOOLKIT`
- Execute program normally
- Output written to files; separate ones for each task
HPMLIB Functions

• Include files
  – Fortran: f_hpmlib.h
  – C: libhpm.h

• Initialize library
  – Fortran: f_hpminit(taskID, progName)
  – C: hpmInit(taskID, progName)

• Start Counter
  – Fortran: f_hpmstart(id,label)
  – C: hpmStart(id,label)
HPMLIB Functions II

• Stop Counter
  – Fortran: `f_hpmstop(id)`
  – C: `hpmStop(id)`

• Finalize library when finished
  – Fortran: `f_hpmterminate(taskID, progName)`
  – C: `hpmTerminate(taskID, progName)`

• You can have multiple, overlapping counter stops/starts in your code
[Declarations]...

\[ Z = 0.0 \]
CALL RANDOM_NUMBER(X)
CALL RANDOM_NUMBER(Y)

!************************************************************************
! Initialize HPM Performance Library and Start Counter
!************************************************************************

CALL f_hpminit(0,"ma.F")
CALL f_hpmstart(1,"matrix-matrix multiply")

DO J=1,N
   DO K=1,N
      DO I=1,N
         \[ Z(I,J) = Z(I,J) + X(I,K) \times Y(K,J) \]
      END DO
   END DO
END DO

!************************************************************************
! Stop Counter and Finalize HPM
!************************************************************************

CALL f_hpmstop(1)
CALL f_hpmterminate(0)

[Finish up] ...
HMPlib Example Output

% module load hpmtoolkit
% xlf90 -o xma_hpmlib -O2 -qarch=pwr3 ma.F \
$HPMTOOLKIT
% ./xma_hpmlib

libHPM output in perfhpm0000.67880

libhpm (Version 2.4.2) summary - running on POWER3-II
Total execution time of instrumented code (wall time): 4.185484 seconds

Instrumented section: 1 - Label: matrix-matrix multiply - process: 0
Wall Clock Time: 4.18512 seconds
Total time in user mode: 4.16946747484786 seconds

PM_FPU0_CMPL (FPU 0 instructions) : 505166645
PM_FPU1_CMPL (FPU 1 instructions) : 6834038
PM_EXEC_FMA (FMAs executed) : 512000683

MIPS : 610.707
Instructions per cycle : 1.637
HW Float points instructions per Cycle : 0.327
Floating point instructions + FMAs : 1024.001 M
Float point instructions + FMA rate : 243.856 Mflip/s
FMA percentage : 100.000 %
Computation intensity : 0.666
The hpmviz tool

• The hpmviz tool has a GUI to help browse HPMLib output
• Part of the hpmtoolkit module
• After running a code with HPMLIB calls, a *.viz file is also produced for each task.
• Usage:
  – `%hpmviz filename1.viz filename2.viz ...
  – Eg.
    • `%hpmviz hpm0000_ma.F_67880.viz

NERSC NUG Training 5/30/03
hpmviz Screen Shot 1

File

<table>
<thead>
<tr>
<th>Label</th>
<th>Count</th>
<th>ExcSec</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix-matrix multiply 1</td>
<td>4.188</td>
<td></td>
</tr>
</tbody>
</table>

Z = 0.0
CALL RANDOM_NUMBER(X)
CALL RANDOM_NUMBER(Y)

! Initialize HPM Performance Library and Start Counter
******************************************************************************

CALL f_hpminit(0,"ma.f")

CALL f_hpmstart(1,"matrix-matrix multiply")

******************************************************************************

!! Theoretical peak: Examine Inner Loop
1 Store
2 Loads
1 FMA (= 2 Flops)

!! Theoretical Peak = (375 MHz)*(2 FPUs)*(2 Flops)/(3 Cycles for Load/Store)

= 500 MFlops/sec

NERSC NUG Training 5/30/03
Right clicking on the Label line in the previous slide brings up a detail window.
Interpreting Output and Metrics
Floating Point Measures

• **PM_FPU0_CMPL (FPU 0 instructions)**
  
• **PM_FPU1_CMPL (FPU 1 instructions)**
  
  – The POWER3 processor has two Floating Point Units (FPU) which operate in parallel.
  
  – Each FPU can start a new instruction at every cycle.
  
  – This is the number of floating point instructions (add, multiply, subtract, divide, FMA) that have been executed by each FPU.

• **PM_EXEC_FMA (FMAs executed)**
  
  – The POWER3 can execute a computation of the form \( x = s \times a + b \) with one instruction. The is known as a Floating point Multiply & Add (FMA).
Total Flop Rate

- **Float point instructions + FMA rate**
  - Float point instructions + FMAs gives the floating point operations. As a performance measure, the two are added together since an FMA instruction yields 2 Flops.
  - The rate gives the code’s Mflops/s.
  - The POWER3 has a peak rate of 1500 Mflops/s. (375 MHz clock x 2 FPUs x 2Flops/FMA instruction)
  - Our example: 241 Mflops/s.
Memory Access

• **Average number of loads per TLB miss**
  – Memory addresses that are in the Translation Lookaside Buffer can be accessed quickly.
    – Each time a TLB miss occurs, a new page (4KB, 512 8-byte elements) is brought into the buffer.
    – A value of ~500 means each element is accessed ~1 time while the page is in the buffer.
    – A small value indicates that needed data is stored in widely separated places in memory and a redesign of data structures may help performance significantly.
  – Our example: 2022
Cache Hits

- The \(-sN\) option to hpmcount specifies a different statistics set
- \(-s2\) will include L1 data cache hit rate
- Power 3 has a 64K L1 data cache
- 98.895% for our example
- See [http://hpcf.nersc.gov/software/ibm/hpmcount/HPM_README.html](http://hpcf.nersc.gov/software/ibm/hpmcount/HPM_README.html) for more options and descriptions.
• The Power 3 can execute multiple instructions in parallel

• MIPS
  – The average number of instructions completed per second, in millions.
  – Our example: 600

• Instructions per cycle
  – Well-tuned codes may reach more than 2 instructions per cycle
  – Our example: 1.632
Computation Intensity

- The ratio of load+store operations to floating point operations
- To get best performance for FP codes, this metric should be <1
- Our example: 0.673
Low-Effort Optimization
Simple Optimization Considerations

• Try to keep data in L1, L2 caches
  – L1 data cache size: 64 KB
  – L2 data cache size: 8192 KB
• Use stride one memory access in inner loops
• Use compiler options
• Maximize: FP ops / (Load+Store ops)
• Unroll loops
• Use PESSL & ESSL whenever possible; they are highly tuned
Stride 1 Array Access

• Consider previous example, but exchange DO loop nesting (swap I, J)
• Inner loop no longer accessed sequentially in memory (Fortran)
• Mflops/s goes 245 -> 11.

```fortran
DO I=1,N
  DO K=1,N
    DO J=1,N
      Z(I,J) = Z(I,J) + X(I,K) * Y(K,J)
    END DO
  END DO
END DO
```
• Effects of different compiler optimization levels on original code
  – No optimization: 23 Mflips/s
  – -O2: 243 Mflips/s
  – -O3: 396 Mflips/s
  – -O4: 750 Mflips/s

• NERSC recommends
  – -O3 -qarch=pwr3 -qtune=pwr3 -qstrict
  – See http://hpcf.nersc.gov/computers/SP/options.html
Max. Flops/Load+Stores

• The POWER 3 can perform 2 Flips or 1 register Load/Store per cycle
• Flips and Load/Stores can overlap
• Try to have code perform many Flips per Load/Store
• For simple loops, we can calculate a theoretical peak performance
Theoretical Peak for a Loop

• How to calculate theoretical peak performance for a simple loop
  – Look at the inner loop only
  – Count the number of FMAs & unpaired +, -, *, & the number of divides*18 = No. Cycles for Flops
  – Count the number of loads and stores that depend on the inner loop index = No. Cycles for load/stores
  – No. of cycles needed for loop = max(No. cycles for Flips, No. cycles for Loads+Stores)
Theoretical Peak Cont’d

• Count the number of FP operators in the loop; one for each +, -, *, /

• Mflops/s = (375 MHz) * (2 FPUs) * (No. FP operators) / (Cycles needed for loop)

• Example
  – 1 store (X) + 2 loads (Y,Z(J) ) = 3 cycles
  – 1 FMA + 1 FP mult = 2 cycles
  – 3 FP operators
  – Theoretical Pk = (375 MHz)*(2 FPUs) * (3Flops) / (3 Cycles) = 750 Mflops

```fortran
DO I=1,N
    DO J=1,N
        X(J,I) = A + Y(I,J)*Z(J) * Z(I)
    END DO
END DO
```
• Our previous example code has a theoretical peak of 500 Mflops.

• Compiling with –O2 yields 245 Mflops

• Only enough “work” to keep 1 FPU busy

```fortran
!**************************************************************************
! Theoretical peak: Examine Inner Loop
! 1 Store
! 2 Loads
! 1 FMA (= 2 Flops)
! Theoretical Peak = (375 MHz)*(2 FPUs)*(2 Flops)/(3 Cycles for Load/Store)
! = 500 MFlops/sec
!**************************************************************************

DO J=1,N
  DO K=1,N
    DO I=1,N
      Z(I,J) = Z(I,J) + X(I,K) * Y(K,J)
    END DO
  END DO
END DO
```
Unrolling Loops

“Unrolling” loops provides more work to keep the CPU and FPUs busy

This loop:
DO I=1,N
   X(I) = X(I) + Z(I) * Y(J)
END DO
Can be unrolled to something like
DO I=1,N,4
   X(I) = X(I) + Z(I) * Y(J)
   X(I+1) = X(I+1) + Z(I+1) * Y(J)
   X(I+2) = X(I+2) + Z(I+2) * Y(J)
   X(I+3) = X(I+3) + Z(I+3) * Y(J)
END DO

-O3 optimization flag will unroll inner loops
Unrolling Outer Loops

- Unrolling outer loops by hand may help
- With –O2 the following gets 572 Mflops; FPU1 and FPU0 do equal work

```fortran
!********************************************************************
! Theoretical peak: Examine Inner Loop
!                       4 Store
!                       5 Loads
!                       4 FMA (= 8 Flops)
! Theoretical Peak = (375 MHz)*(2 FPUs)*(8 Flops)/(9 Cycles for Load/Store)
!                  = 667 MFlops/sec
!********************************************************************

DO J=1,N,4
   DO K=1,N
      DO I=1,N
         Z(I,J) = Z(I,J) + X(I,K) * Y(K,J)
         Z(I,J+1) = Z(I,J+1) + X(I,K) * Y(K,J+1)
         Z(I,J+2) = Z(I,J+2) + X(I,K) * Y(K,J+2)
         Z(I,J+3) = Z(I,J+3) + X(I,K) * Y(K,J+3)
      END DO
   END DO
END DO
```
ESSL is Highly Optimized

- ESSL & PESSL provide highly optimized routines
- Matrix-Matrix multiply routine DGEMM gives 1,300 Mflops or 87% of theoretical peak.
- Mflops/s for various techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>idim - Row/Column Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Fortran Source</td>
<td>695</td>
</tr>
<tr>
<td>C Source</td>
<td>692</td>
</tr>
<tr>
<td>matmul (default)</td>
<td>424</td>
</tr>
<tr>
<td>matmul (w/ essl)</td>
<td>1176</td>
</tr>
<tr>
<td>dgemm (-lessl)</td>
<td>1299</td>
</tr>
</tbody>
</table>
Real-World Example

- User wanted to get a high percentage of the POWER 3’s 1500 Mflop peak
- An look at the loop shows that he can’t

Real-world example (Load/Store dominated):

```
!*******************************************************************************
! Loads: 4; Stores 1
! Flops: 1 FP Mul
! Theoretical Peak:
! (375 MHz)*(2 FPUs)*(1 Flop)/(5 Cycles) = 150 MFlops
! Measured: 57 MFlops
!*******************************************************************************

   do 56 k=1,kmax
       do 55 i=28,209
           uvect(i,k) = uvect(index1(i),k) * uvect(index2(i),k)
      55 continue
   56 continue
```
• Unrolling the outer loop increases performance

```
do 56 k=1,kmax,4
   do 55 i=28,209
      uvect(i,k) = uvect(index1(i),k) * uvect(index2(i),k)
      uvect(i,k+1) = uvect(index1(i),k+1) * uvect(index2(i),k+1)
      uvect(i,k+2) = uvect(index1(i),k+2) * uvect(index2(i),k+2)
      uvect(i,k+3) = uvect(index1(i),k+3) * uvect(index2(i),k+3)
   55  continue
56  continue```

---

!******************************************************************************
!Theoretical Peak:
!  Loads: 10
!  Stores: 4
!  Flops: 4 FP Mult
!Theoretical Peak:
!  (375 MHz)*(2 FPUs)*(4 Flop)/(14 Cycles) = 214 MFlops
!Measured: 110 MFlops
!******************************************************************************
Summary

• Utilities to measure performance
  – hpmcount
  – poe+
  – hpmlib

• The compiler can do a lot of optimization, but you can help

• Performance metrics can help you tune your code, but be aware of their limitations
Where to Get More Information

• NERSC Website: http://hpcf.nersc.gov
• PAPI
  – http://hpcf.nersc.gov/software/tools/papi.html
• hpmcount, poe+
  – http://hpcf.nersc.gov/software/ibm/hpmcount/
  – http://hpcf.nersc.gov/software/ibm/hpmcount/counter.html
• hpmlib
  – http://hpcf.nersc.gov/software/ibm/hpmcount/HPM_README.html
• Compilers, general NERSC SP info
  – http://hpcf.nersc.gov/computers/SP/