Porting / Optimizing Codes for Summit Acceptance

Bob Walkup
walkup@us.ibm.com 914-945-1512

Programming environment : compilers, libraries, modules

Basic utilities : addr2line, objdump, readelf, ps, gstack, nvidia-smi

- Checking affinity and device assignments
- What to do when your code hangs : pdsh with gstack

Performance tools with examples (mostly with HACC)

- Elapsed-time timers
- Interrupt-based program sampling
- MPI profiling
- Hardware Counters

Compilers, libraries, and modules

IBM XL compilers – often provide an advantage for Power9 CPU code.
    Common option choice : -g -O3 -qhot -qarch=pwr9   (Fortran, C, C++)
    Assembly listings : -qlist (add -qsource for Fortran)
    Porting issues : using C-preprocessing for Fortran
        -WF,-DUSE_MPI ,   -qsuffix=cpp=f90 (not needed for .F, .F90)
      Fortran/C interface : XL default is no added underscore
        -qextname option for XL Fortran adds underscore
    For Summit : export OMPI_FC=xlf95_r (or higher language level)

GNU compilers : default is gcc-4.8.5 which does not support -mcpu=power9
        gcc/6.4.0 and gcc/8.1.0   have good power9 support

PGI compilers : required for OpenACC … check module avail for the latest

Mixing compilers : check env | grep OLCF_
You can set OLCF_ variables and add directories to your PATH manually
MPI include path : -I${OLCF_SPECTRUM_MPI_ROOT}/include

IBM ESSL : module add essl/6.1.0-1   … this adds XL libs to LD_LIBRARY_PATH
ESSL library contains entry points with and without an added underscore

NVIDIA cublas library : can get a good performance boost on Volta :
    zgemm(…) => cublasZgemm3m(…) is faster than cublasZgemm(...)

Malloc replacement libraries can sometimes help : jemalloc, tcmalloc

Basic Utilities – Tools for Survival

ssh to a compute node while your job is running : ensure correct resource assignments
Check affinity (top or ps command) and GPU assignments (nvidia-smi). Can use
sched_getcpu() to return cpu affinity inside your app.

```
#!/bin/bash    (this is the script that I use : psbind)
if [ -z "$1" ]; then
  echo syntax: psbind executable_file
  exit
fi
for i in $(pgrep $1); do ps -mo pid,tid,fname,user,psr,time,cmd -p $i;done
```

Check shared-libraries and any encoded rpath : readelf -d your.exe ; or ldd your.exe
Advanced : use the Linux patchelf command to modify the executable file

Your code crashes and you get a hex address : addr2line -e your.exe hex_address

You want to see the assembly code in your executable : objdump -d your.exe >dump.txt

You want a snapshot of the call-stack for a running process : gstack pid

Your code is hung :

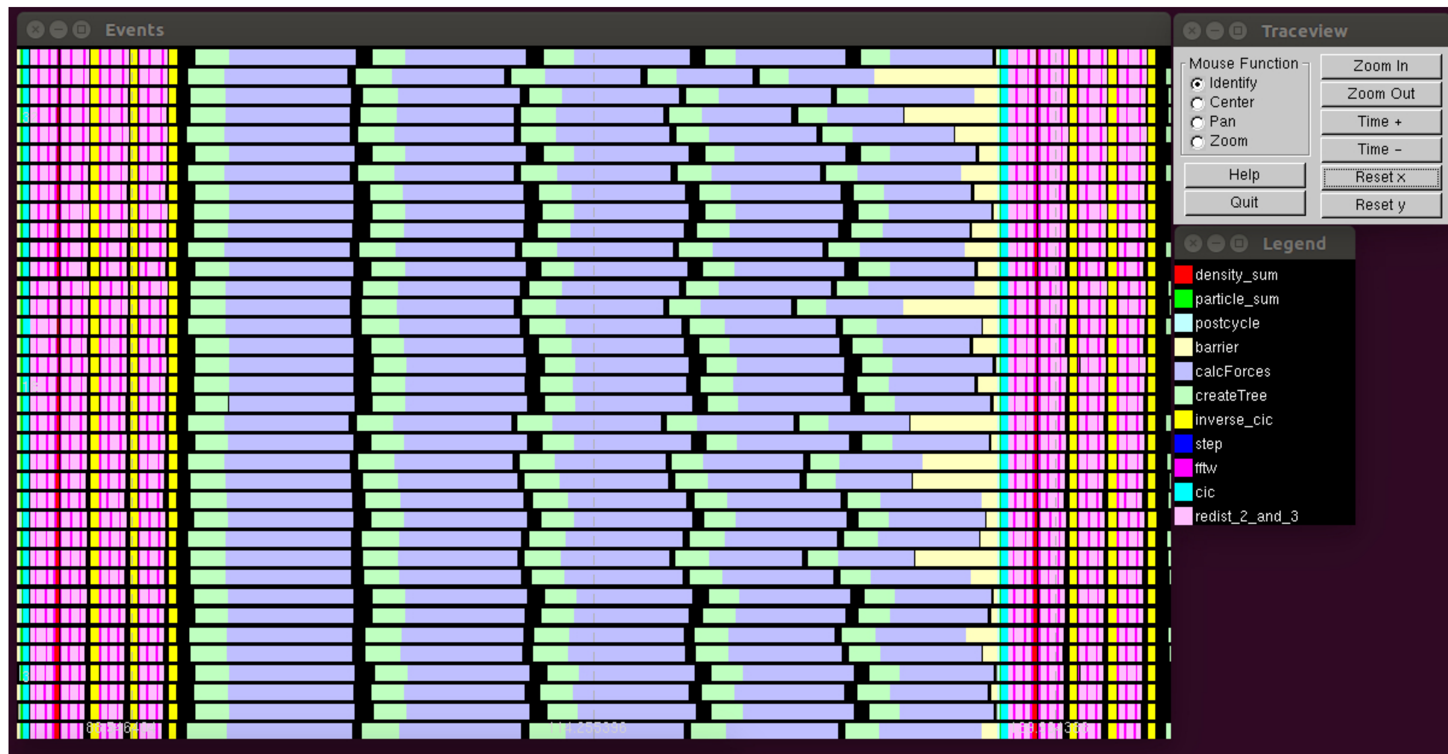
```
make a host file : cat ~/.lsbatch/*.jobid.hostfile | uniq >hf  ; delete the first line (launch node)
 export WCOLL=/path/to/hf   (this defines the host list for the pdsh command)
 pdsh -f 144 /path/to/getstacks.sh   (launches the getstacks.sh script, 144 nodes at a time)
 #!/bin/bash    (this is the script that I use, getstacks.sh … put in your executable file name)
 host=`hostname -s`
 for pid in $( pgrep your_executable_file )
    do /usr/bin/gstack $pid >>/path/to/$host.stacks
 done
```

Use grep and "wc" judiciously … need to find any outliers.

Elapsed-time Timers : Instrument your code

I use : gettimeofday(…) ; microsecond granularity, sub-microsecond overhead
Timer library : Timer_Beg("code_block"); …; Timer_End("code_block");
Can collect timing summary, or save event records for graphical display.

Timing summary:	#calls	avg(sec)	min(sec)	minRank	max(sec)	maxRank
step	1	173.1	173.0	0	173.1	17
cic	4	1.5	1.5	33	1.5	14
redist_2_and_3	80	18.5	15.8	0	20.6	35
fftw	48	5.4	5.3	31	5.6	1
inverse_cic	12	5.1	5.0	33	5.4	11
createTree	15	27.2	25.8	34	30.0	0
calcForces	15	92.7	83.6	7	97.6	26



X-axis = time
Y-axis = MPI rank

Time-line display can be revealing.

HACC 1-node 36 MPI ranks.

Can scale to thousands of ranks.

Interrupt-based Program Sampling - Finding Hot Spots

Instead of gprof, I use hardware counters to trigger interrupts, via PAPI.

Advantages : can collect data on shared-libraries, not just program text
can optionally sample using any hardware counter event

(1) generate profile data : add (or modify) one line to your job script
export OMPI_LD_PRELOAD_POSTPEND=/path/to/libhpmprof.so
On Summit : /ccs/home/walkup/mpitrace/spectrum_mpi/libhpmprof.so
Ensure that -g is included as a compiler option, for statement-level info.

(2) analyze the data : bfdprof your.exe hpm_histogram.jobid.rank >profile.jobid.rank
Or : annotate_objdump your.exe hpm_histogram.jobid.rank > dump.jobid.rank
On Summit : /ccs/home/walkup/mpitrace/bin has bfdprof , annotate_objdump

Caveats :

(a) Some elements of the system software are not interrupt safe. Enabling interrupts can cause codes to crash. There are not always workarounds.

(b) Certain coding styles are not amenable to statement or function-level profiling. Example : templated C++ where every source line has many function calls, where all operators are defined in header files that are included everywhere.

Interrupt-based Program Sampling :: HACC function-level data

Output from bfdprof :

```
Using executable file : ../omp4/cpu/ibm/hacc_tpm
     histogram file : hpm_histogram.315735.0.
     14574 hits at 1087 program-text locations,
      5946 hits outside the program-text section,
     20520 hits total.
HPM sampling using event = perf::cycles, threshold = 34500000.

##########################
Shared library profile:
##########################
      tics   sharedlib
--------------------------
      3334   /usr/lib64/nvidia/libcuda.so.1
      1023   /opt/mellanox/hcoll/lib/libhcoll.so.1
       889   /lib64/libpthread.so.0
       499   /lib64/libc.so.6
…
########################################
Function-level profile (exclusive data):
########################################
    tics   function-name
--------------------------
    4523  00000017.plt_call.fmaxf@@GLIBC_2.17
    3342  00000017.plt_call.fminf@@GLIBC_2.17
    2005  RCBForceTree<1>::createRCBForceSubtree
    1174  IPRA.$redistribute_2_and_3
    1050  RCBForceTree<1>::calcInternodeForce
     564  cm
     507  Particles::inverse_cic
     371  Particles::resortParticles
```

This section is similar to the gprof flat-profile.

expensive function calls fminf, fmaxf … why ??

Program Sampling : HACC continued

Center-of-mass calculation : statement-level profile … output from bfdprof

```
  tics |
       |   for (int i = 1; i < count; ++i) {
     87|     xmin0 = fminf(xmin0, xx[i]);
    114|     xmax0 = fmaxf(xmax0, xx[i]);
    106|     xmin1 = fminf(xmin1, yy[i]);
     74|     xmax1 = fmaxf(xmax1, yy[i]);
    112|     xmin2 = fminf(xmin2, zz[i]);
     69|     xmax2 = fmaxf(xmax2, zz[i]);
       |
       |     w = mass[i];
       |     x += w*xx[i];
      1|     y += w*yy[i];
       |     z += w*zz[i];
      1|     m += w;
       |   }
```

six function calls per particle
want vector inline assembly.

Assembly listing using -qlist XL compiler option : showing the function calls inside the loop

```
209|                                        CL.1:
210| 0000D0 lfsx      7F38BC2E   1     LFS        vs25=...
210| 0000D4 xxlor     F059CC90   1     LRVS       vs2=vs25
210| 0000D8 bl        48000001   0     CALLN      vs1,vs14-vs31=fminf..
210| 0000DC ori       60000000   1
211| 0000E0 xxlor     F059CC90   1     LRVS       vs2=vs25
210| 0000E4 xxlor     F3010C90   1     LRVS       vs24=vs1
211| 0000E8 xxlor     F03EF490   1     LRVS       vs1=vs30
211| 0000EC bl        48000001   0     CALLN      vs1,vs14-vs31=fmaxf..
```

Program Sampling : HACC continued

Help the compiler : #define FMINF(a,b) (((a)<(b))?(a):(b))

Annotated objdump … now with inline vector assembly in the center-of-mass loop

```
  tics |
     2|00000000100ac6d0 <cm+0x470> xvcvspdp vs14,vs6
     4|00000000100ac6d4 <cm+0x474> xvcvspdp vs63,vs8
     3|00000000100ac6d8 <cm+0x478> xxmrghw vs6,vs31,vs31
   148|00000000100ac6dc <cm+0x47c> addi    r12,r12,32
    52|00000000100ac6e0 <cm+0x480> addi    r31,r31,32
      |00000000100ac6e4 <cm+0x484> xxmrghw vs8,vs28,vs28
    62|00000000100ac6e8 <cm+0x488> addi    r30,r30,32
      |00000000100ac6ec <cm+0x48c> xvcvspdp vs62,vs9
      |00000000100ac6f0 <cm+0x490> xxmrglw vs9,vs28,vs28
    13|00000000100ac6f4 <cm+0x494> xvmulsp vs31,vs42,vs47
    26|00000000100ac6f8 <cm+0x498> lxv     vs42,-32768(r31)
  ...
     8|00000000100ac7ec <cm+0x58c> xvadddp vs35,vs35,vs62
    19|00000000100ac7f0 <cm+0x590> lxv     vs47,-32768(r11)
     2|00000000100ac7f4 <cm+0x594> xvcvspdp vs13,vs21
     2|00000000100ac7f8 <cm+0x598> xxmrglw vs21,vs31,vs31
     1|00000000100ac7fc <cm+0x59c> xvadddp vs36,vs36,vs14
     2|00000000100ac800 <cm+0x5a0> xvadddp vs39,vs39,vs58
     4|00000000100ac804 <cm+0x5a4> xvadddp vs37,vs37,vs63
      |00000000100ac808 <cm+0x5a8> bdnz    00000000100ac6d0 <cm+0x470>
```

addr2line -e hacc_tpm 0x100ac6f4   translate address to source file/line number

Power9 assembly reference : search for power9 isa assembly
https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0

Program Sampling : HACC improved

Output from bfdprof :

```
Using executable file : ../omp4/cpu/ibm/hacc_tpm,
      histogram file : hpm_histogram.315616.0.
      7130 hits at 1147 program-text locations,
      5952 hits outside the program-text section,
     13082 hits total.
HPM sampling using event = perf::cycles, threshold = 34500000.

##########################
Shared library profile:
##########################
      tics   sharedlib
--------------------------
      3329   /usr/lib64/nvidia/libcuda.so.1
      1025   /opt/mellanox/hcoll/lib/libhcoll.so.1
       853   /lib64/libpthread.so.0
       534   /lib64/libc.so.6
...
########################################
Function-level profile (exclusive data):
########################################
    tics   function-name
--------------------------
    2296  RCBForceTree<1>::createRCBForceSubtree
    1191  IPRA.$redistribute_2_and_3
    1092  RCBForceTree<1>::calcInternodeForce
     646  cm
     508  Particles::inverse_cic
     377  Particles::resortParticles
```

2x reduction in time spent doing CPU computation.

HACC timing summaries … before and after tuning

Before tuning :

Timing summary:	#calls	avg(sec)	min(sec)	minRank	max(sec)	maxRank
step	1	248.8	248.7	0	248.7	18
cic	4	1.5	1.5	33	1.5	14
redist_2_and_3	80	18.5	15.6	0	20.6	35
fftw	48	5.4	5.3	30	5.7	0
inverse_cic	12	5.2	5.1	33	5.4	11
createTree	15	103.6	102.6	20	105.5	14
calcForces	15	91.1	74.9	8	96.4	17

After tuning :

Timing summary:	#calls	avg(sec)	min(sec)	minRank	max(sec)	maxRank
step	1	173.1	173.1	0	173.1	17
cic	4	1.5	1.5	33	1.5	14
redist_2_and_3	80	18.5	15.8	0	20.6	35
fftw	48	5.4	5.3	31	5.6	1
inverse_cic	12	5.2	5.1	33	5.4	11
createTree	15	27.2	25.8	34	30.1	0
CalcForces	15	92.7	83.6	7	97.6	26

More than 3x improvement in the time for createTree … now getting a much better benefit from the GPUs.

HACC : OpenACC and/or OpenMP for pair-force evaluation

The OpenMP offload code is shown below … OpenACC is very similar

```
#pragma omp target teams distribute map(to: data1[0:npts1]) \
        map(from: data2[0:npts2]) num_teams(count) thread_limit(224)
   for (int i = 0; i < count; ++i) {
     float vxi = 0.0f, vyi = 0.0f, vzi = 0.0f;
     float * xx    = data1;  ...; float * vx = data2; ... // set pointers
     float xxi = xx[i], yyi = yy[i], zzi = zz[i];
     float mifc = mass[i]*fcoeff;
#pragma omp parallel for reduction(+:vxi,vyi,vzi)
     for (int j = 0; j < count1; ++j) {
       float dx = xx1[j] - xxi;
       float dy = yy1[j] - yyi;
       float dz = zz1[j] - zzi;
       float d2 = dx*dx + dy*dy + dz*dz;
       if (d2 < fsrrmax2) {
         float rtemp = (d2 + rsm2)*(d2 + rsm2)*(d2 + rsm2);
         float poly = ma0 + d2*(ma1 + d2*(ma2 + d2*(ma3 + d2*(ma4 + d2*ma5))));
         float f_over_r = mifc*mass1[j]*(1.0f/sqrtf(rtemp) - poly);
         vxi += f_over_r*dx;
         vyi += f_over_r*dy;
         vzi += f_over_r*dz;
       }
     }
     vx[i] = vxi;
     vy[i] = vyi;
     vz[i] = vzi;
   }
```

OpenACC (PGI compiler) provided slightly faster GPU kernel times. OpenMP (XL compiler) provided faster CPU code, resulting in better overall performance.

MPI Profile Data : use the MPI profiling interface => interposition library

There are many similar tools, example mpiP … find one that you like and use it.

I use libmpitrace.so … ships with Spectrum MPI

${OLCF_SPECTRUM_MPI_ROOT}/lib/libmpitrace.so
Or latest : /ccs/home/walkup/mpitrace/spectrum_mpi/libmpitrace.so
Add one line to your job script :
export OMPI_LD_PRELOAD_POSTPEND=/path/to/libmpitrace.so
Normally get data for rank 0 and ranks with min, median, and max times in MPI

```
Data for MPI rank 0 of 6480:
Times and statistics from summary_start() to summary_stop().
-----------------------------------------------------------
MPI Routine              #calls     avg. bytes      time(sec)
-----------------------------------------------------------
MPI_Comm_rank                 3            0.0          0.000
MPI_Comm_size                 3            0.0          0.000
MPI_Isend                  1784     31215583.9          0.015
MPI_Irecv                  1784     31215583.9          1.881
MPI_Wait                   2944            0.0         92.653
MPI_Waitall                 312            0.0         57.871
MPI_Barrier                   3            0.0          0.000
MPI_Allreduce                15            8.0         13.272
-----------------------------------------------------------
total communication time = 165.692 seconds.
total elapsed time       = 630.499 seconds.
user cpu time            = 503.543 seconds.
system time              = 124.274 seconds.
max resident set size    = 6147.938 MiB.
```

On Summit adaptive routing is enabled by default : check env | grep PAMI_

PAMI_IBV_ENABLE_OOO_AR=1
PAMI_IBV_QP_SERVICE_LEVEL=8

Adaptive routing applies for messages larger than the eager limit = 64KB, which can be adjusted via runtime arguments or env variables.

Data for HACC, 180 nodes, Sierra, 36 MPI ranks/node
BW estimate : 2 * 1784 * 3.1216e7 * 36e-9 / 152.5 = 26 GB/sec exch bw per node

MPI Profile Data : useful information when things go wrong

```
Data for MPI rank 0 of 20480:
Times and statistics from summary_start() to summary_stop().
…

Communication summary for all tasks:

  minimum communication time = 35.036 sec for task 11947
  median  communication time = 216.645 sec for task 6714
  maximum communication time = 242.055 sec for task 4827

Histogram of times spent in MPI
    time-bin    #ranks
      35.036         4
      49.823        11
      64.610        12
      79.397         8
      94.185         5
     108.972         0
     123.759         0
     138.546         0
     153.333         0
     168.120         0
     182.907      4464
     197.694      3176
     212.481      5636
     227.268      5141
     242.055      2023
```

The profile data includes host and cpu assignments.

```
MPI timing summary for all ranks:
taskid     hostname    cpu    comm(s)  elapsed(s)   size(MiB)
     0   sierra1210      0     239.74      269.15      48.31
     1   sierra1210      4     237.64      269.15      48.74
     2   sierra1210      8     237.27      269.15      51.90
...
 11918   sierra3247    160     189.61      269.15      33.17
 11919   sierra3247    164     189.32      269.15      34.05
 11920   sierra3248      0      74.26      269.15      26.17
 11921   sierra3248      1      68.52      269.15      12.12
 11922   sierra3248      2      82.43      269.15      20.66
 11923   sierra3248      3      75.46      269.15      24.18
 11924   sierra3248      4      68.72      269.15      26.64
...
 11960   sierra3249      0     192.39      269.15      34.80
 11961   sierra3249      4     192.37      269.15      33.20
 11962   sierra3249      8     197.14      269.15      34.83
```

The distribution of time in MPI indicates trouble : nearly all ranks are waiting on a few, with a large gap in the timing histogram. In this case there were 40 ranks per node, and all 40 ranks on node sierra3248 were slow.

This job requested “smt1”, but node sierra3248 was provided in “smt4” mode. Notice the cpu assignments for node sierra3248 : multiple ranks share the same core, hence bad performance.

MPI profiling with libmpitrace.so : tips

Overhead is ~0.1 microsec per call … normally (but not always) negligible.

Use a threadsafe version if your code makes independent, concurrent calls to MPI from multiple threads. In this case the time reported in MPI is cumulative over threads, and is bounded by number-of-threads times the elapsed time.

Outputs are named mpi_profile.jobid.rank … so if you have multiple MPI job-steps in the same LSF job, you should save the profile data in a separate directory for that step (requires adding a few lines between steps to your job script).

To limit MPI profile data to a given code-region :

```
MPI_Pcontrol(1); // start profiling  ;  for Fortran : call mpi_pcontrol(1)
…
MPI_Pcontrol(0); // stop profiling ;   for Fortran : call mpi_pcontrol(0)
```

There are many options controlled by env variables … examples :

```
export TRACE_SEND_PATTERN=yes     to record pt-to-pt message pattern
export SAVE_LIST=0,100,200        to save data for specific ranks
```

Can optionally turn on event tracing … but that topic requires more discussion.

Advanced Topic : Hardware Counters

I use PAPI ; git clone https://bitbucket.org/icl/papi.git
configure … --with-CPU=POWER9
can optionally enable CUDA : --with-components=cuda

Note : as of Dec. 1, 2018, Summit has papi-5.5.1 but it is broken … reports no counters

I use a thin layer on top of PAPI , with simple stop / start around code sections :

C++ : extern “C” void HPM_Start(const char *); C : void HPM_Start(const char *);
C/C++ : HPM_Start(“label”); … do work …; HPM_Stop(“label”);
Fortran : call hpm_start(‘label’); … do work …; call hpm_stop(‘label’);

If you want cumulative counts for the whole job, no code instrumentation is needed.

export OMPI_LD_PRELOAD_POSTPEND=/path/to/libmpihpm.so

On Summit : libmpihpm.so is in /ccs/home/walkup/mpitrace/spectrum_mpi
Current build has Power9 counters only … not CUDA
Can use MPI_Pcontrol(1); …; MPI_Pcontrol(0); for calipers

Power9 supports counting up to six events concurrently without multiplexing.

papi_native_avail will list all native Power9 events.

I currently have a only a few pre-defined groups of counters for Power9 …
/ccs/home/walkup/mpitrace/src/p9_groups.h … export HPM_GROUP=7

Instead of groups, you can define your own list of events via HPM_EVENT_LIST

Power9 Hardware Counter Examples

Data for HACC, just over the time-step loop (HPM_GROUP=0)

```
pcontrol, call count = 1, avg time = 173.957, max time = 173.957 :
  -- Counter values for processes in this reporting group ----
  min-val  min-rank  max-val  max-rank  avg-value   label
  5.65e+10      13   5.74e+10      19   5.675e+10 (PM_FLOP_CMPL) Floating Point instructions completed
  9.65e+10       7   9.91e+10       4   9.800e+10 (PM_ST_FIN) Store instructions finished
  2.27e+11       1   2.53e+11      26   2.386e+11 (PM_LSU_FIN) LSU instructions finished
  2.15e+11      31   2.50e+11       1   2.284e+11 (PM_CMPLU_STALL) Completion stall cycles
  4.82e+11       1   5.85e+11      26   5.207e+11 (PM_RUN_INST_CMPL) Run instructions completed
  4.51e+11      12   4.71e+11      16   4.575e+11 (PM_RUN_CYC) Run cycles
```

Effective average frequency : 4.575e+11/173.96 = 2.63 GHz
Instructions completed per cycle per core : 1.14
Percent of instructions that do floating-pint work : 12.4%

Data for a simple streaming load kernel (HPM_GROUP=7), 42 ranks/node

```
load, call count = 1, avg time = 2.827, max time = 2.848 :
  -- Counter values for processes in this reporting group ----
  min-val  min-rank  max-val  max-rank  avg-value   label
  4.99e+05      36   5.72e+05      20   5.344e+05 (PM_MEM_READ) Reads from memory from this thread
  1.56e+08       9   1.56e+08      36   1.557e+08 (PM_MEM_PREF) Prefetches from memory for this thread
  1.25e+09      26   1.25e+09      29   1.250e+09 (PM_2FLOP_CMPL) VSU two-flop instructions completed
  1.25e+09      35   1.25e+09      21   1.250e+09 (PM_LSU_FIN) LSU instructions finished
  2.81e+09      26   2.81e+09      29   2.812e+09 (PM_RUN_INST_CMPL) Run instructions completed
  9.63e+09      20   9.78e+09      35   9.705e+09 (PM_RUN_CYC) Run cycles
```

42*(5.344e+05 + 1.557e+08)*128e-9/2.848 = 294.9 GB/sec   load bandwidth per node
Counters distinguish between “demand loads” and data that is prefetched.
For PM_MEM_READ and PM_MEM_PREF, each count indicates a 128-byte transfer