ECE 574 – Cluster Computing
Lecture 5

Vince Weaver

http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

11 September 2026

Announcements

- Homework #2 was posted
  Involves logging into server, running linpack while doing perf measurements, using slurm
- Haswell-EP was set up
- I'll be out of town Monday the 14th: no class
  Also no office hours Tuesday the 15th

Server Account Info

- Handed out account information in class
- Log in to weaver-lab. Be sure to use port 2131 or it will try to connect to the wrong machine. (Why?)
- Change your password first thing.

Server Guidelines

- Behave. No hacking / cracking / spamming / irc-bots / bitcoing-mining
  Also be responsible with disk usage, as I don't have disk quota set up.
- Also the disk isn't backed up so be careful when deleting files (using git locally might be a good option to avoid that)
- If you find a security bug, great! Let me know! Don't go deleting things or impersonating people or installing root kits, or other stuff.

Speedup / Parallel Efficiency Review

- Reminder
  - Speedup $= S_p = \frac{T_s}{T_p}$
    where p=# of processes (threads)
    T_s = execution time of sequential code
    T_p = execution time of parallel with p processes
    For ideal, $S_p = p$
  - Parallel Efficiency
    $E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$
    Ideal linear speedup E_p=1

Speedup / Parallel Efficiency Examples

- Examples where serial code takes 120s, p=2
  - $T_2 = 150s,\ S_p = \frac{120}{150} = 0.8, E_p = \frac{.8}{2} = .4$
  - $T_2 = 120s,\ S_p = \frac{120}{120} = 1, E_p = \frac{1}{2} = .5$
  - $T_2 = 60s,\ S_p = \frac{120}{60} = 2, E_p = \frac{2}{2} = 1$
  - $T_2 = 30s,\ S_p = \frac{120}{30} = 4, E_p = \frac{4}{2} = 2$

`time` Note on Time Measurements

- Ideally try to measure on an idle system
- Can turn off various features (ASLR, bind to cores)
- Even then, expect sometimes up to 5% variation run to run
- Ideally take *many* measurements and do things like calculate standard deviation
- Be wary making changes to your code and reporting speedups of under 10% because they might be noise

Hardware Performance Counters

- Registers that hold architectural performance counts
- Available on all modern CPUs
- Usually 2-8 of them, often 40-64 bits wide
- Possibly up to 100s of events available
- Have registers you set to enable, start, stop, read value, select event type
- Interface varies arch to arch, vendor to vendor, and even chip revisions

Performance Counters – Sampling

- Hardware interrupt can be triggered when counter overflows. Why?
  - If you read infrequently, could miss overflows and be off
  - Also useful for sampling.

Performance Counters – OS Interference

- Events measure everything happening in the system (including operating system)
- Can you measure events only belonging to your program?
  - Operating system can save/restore registers on context switch

Are counter results accurate?

- See my various papers
- Short answer is usually, but more obscure might not be
- Intel/AMD also tend to overcount on interrupts
- How would you validate the counters themselves? Exact assembly language program.
- Also chip companies care, but counter correctness is not enough to stop a chip from shipping. They might undocument (or errata) if you report a bug.

Linux Support

- `perf_event_open()` system call. Really complex, see the manpage.
- Old days was perfctr, then perfmon which required patching kernel.
- Slowly looked like was getting merged, but then out of nowhere Molnar introduced perf_event which got in quickly in 2.6.31 kernel
- Has issues but is mostly good enough these days.

perf tool

- perf tool comes with kernel
- Can be used for doing measurement
- Will give a demo, but you can do something like `perf stat ./xhpl`
- Might be disabled by default for security reasons, at least partly it is my fault.

PAPI

- Layer of abstraction.
- Want to use counters on all kinds of supercomputers without having to change for each?
- Also provides self-monitoring, can add “calipers” to your code to measure things.

Profiling

- Records summary information during execution
- Usually Low Overhead
- Implemented via:
  - **Sampling** (execution periodically interrupted and measures what is happening)
  - **Measurement** (extra code inserted to take readings)

Profiling Tools

- Low Overhead – Using hardware counters, such as perf
- Small Overhead – Using static instrumentation, such as gprof
- Large Overhead – Using dynamic binary instrumentation, such as valgrind callgrind
- Extreme Overhead – full system simulator

Compiler Profiling

- gprof
- gcc -pg
- Adds code to each function to track time spent in each function.
- Run program, gmon.out created. Run “gprof executable” on it.
- Adds overhead, not necessarily fine-tuned, only does time based measurements.
- Pro: available wherever gcc is.

DBI Profiling

- Valgrind / callgrind tool

Tracing

- When and where events of interest took place
- Shows when/where messages sent/received
- Records information on significant events
- Provides timestamps for events
- Trace files are typically *huge*
- When doing multi-processor or multi-machine tracing, hard to line up timestamps

Using Perf

perf tool

```
$ perf stat ./dgemm_naive 200
Will need 1280000 bytes of memory, Iterating 10 times

 Performance counter stats for './dgemm_naive 200':

       7239.152263      task-clock (msec)         #    0.992 CPUs utilized
               116      context-switches          #    0.016 K/sec
                 0      cpu-migrations            #    0.000 K/sec
               357      page-faults               #    0.049 K/sec
     6,513,184,942      cycles                    #    0.900 GHz
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
     2,592,685,475      instructions              #    0.40  insns per cycle
        91,797,411      branches                  #   12.681 M/sec
           974,817      branch-misses             #    1.06% of all branches

       7.299463710 seconds time elapsed
```

perf available events

- Many options. Can select events with `-e`
- Use `perf list` to list all available events
- Hundreds of events available on x86, not quite so many on ARM.
- Understanding the results often requires a certain knowledge of computer architecture.

Perf Profiling

Automatically interrupts program and takes sample every X instructions.

- `perf record`
- `perf report`
- `perf annotate`

Skid

- Beware of “skid” in sampled results
- This is what happens when a complex processor cannot stop immediately, so the reported instruction might be off by a few instructions.
- Some processors do not have this problem. Recent Intel processors have special events that can compensate for this.

Performance Data Analysis

- **Manual Analysis**
  - Visualization, Interactive Exploration, Statistical Analysis
  - Examples: TAU, Vampir
- **Automatic Analysis**
  - Try to cope with huge amounts of data by automatic analysis
  - Examples: Paradyn, KOJAK, Scalasca, Perf-expert

Slurm

- https://slurm.schedmd.com/
- Slurm Workload Manager
  Simple Linux Utility for Resource Management
  Futurama Joke?
- Developed originally at LLNL
- Over 60% of top 500 use it (when?)

Setting Up Slurm

- Compiling / installing?
  Luckily Debian and such have packages
- Authentication – how does the job scheduler “log in” to each node to get the code running?
- slurm has something called “munge” that does authentication
- Setting up config file, a pain to get right
- Auto-starting the various servers at boot
- Fault tolerance

sinfo

provides info on the cluster

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug        up   infinite      1   idle haswell-ep
general*     up   infinite      1   idle haswell-ep
```

`srun`

start a job, but interactively

sbatch

submit job to job queue

```
#!/bin/bash

#SBATCH -p general # partition (queue)
#SBATCH -N 1  # number of nodes
#SBATCH -n 8  # number of cores
#SBATCH -t 0-2:00 # time (D-HH:MM)
#SBATCH -o slurm.%N.%j.out # STDOUT
#SBATCH -e slurm.%N.%j.err # STDERR
export OMP_NUM_THREADS=4
./xhpl
```

`sbatch` usage

- Specify in the shell script comments various parameters, sort of like command line parameters.
- `sbatch -N 24 - -ntasks-per-node=4 ./time_coarse.sh` To run on all 96 cores of pi-cluster
- Can set up to e-mail you when done (though only locally).

squeue

```
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
   63   general time_hpl ece574-0 PD       0:00      1 (Resources)
   64   general time_hpl ece574-0 PD       0:00      1 (Resources)
   65   general time_hpl ece574-0 PD       0:00      1 (Resources)
   62   general time_hpl ece574-0  R       0:14      1 haswell-ep
```

`scancel`

kills job

```
scancel 65
```

High Performance Linpack (HPL)

- HPL solves linear system of equations, Ax=b. LU factorization.

Building Linpack

- Download and install a BLAS. ATLAS? OpenBLAS? Intel?
  Compiler? intel? gcc? gfortran?
- Download and install MPI (we'll talk about that later). MPICH? OpenMPI? (these days I use OpenMPI)
- Download HPL. Current version 2.3?
  Modify a Makefile (not trivial) make sure links to proper BLAS. make arch=OpenBLAS
- Above step, might need to create a link from hpl in your home directory to actual location for reasons
- Creates a bin/OpenBLAS with default HPL.dat file

Running Linpack

- Run it `./xhpl` Or if on cluster `./mpirun -np 4 ./xhpl` or similar.
- Result won't be very good. Need to tune HPL.dat
- N is problem size. In general want this to fill RAM. Take RAM size, squareroot, round down. NxN matrix. Each N is 8 bytes for double precision.
- NB block size, can be tuned
- PxQ, if on cluster can specify machine grid to work on. Linpack works best with as square as possible.
- Fiddle with all the results until you get the highest.

Linpack Results on my Lab Computers

- `https://web.eece.maine.edu/~vweaver/group/machines.html`
- Selection of Machines
  - haswell-ep: 436 GFLOPS, 16/32 cores, 80GB, 2.13GFLOP/W
  - M1 ARM Mac laptop: 154 GFLOPS, 6 GFLOPS/W
  - power8: 195 GFLOPS, 8/64 cores, 32GB
  - pi-cluster: 15.4 GFLOPS, 96 cores, 24GB RAM, 0.166 GFLOP/W
  - pi-4B 2.02 GFLOPS/W
- First top500 list, June 1993. Top machine 1024 cores, 60 GFLOPS, 131kW
  Pi cluster would have been #7

Haswell-EP summary

- Theoretical: 16DP FLOP/cycle * 16 cores * 2.6GHz = 666 GFLOPS
- Linpack/OpenBLAS: 436 GFLOPS (65% of peak)
- HPCG: 0.7 GFLOPS (0.1% of peak)

Live Demo – Logging in

- Log in
- Run Linpack
- Try out “time”

Live Demo – Perf

- perf stat
- perf list
- perf record
- perf report
- perf annotate

Live Demo – System Status

- w
- top
- htop
- glances
- btop

Live Demo – Slurm

- sinfo
- squeue
- sbatch
- show sinfo on the pi-cluster