ECE 574 – Cluster Computing
Lecture 3

Vince Weaver

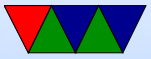
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

8 September 2015

Announcements

- None this week.

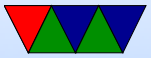


Speedup

- Speedup is the improvement in latency (time to run)

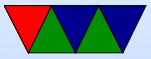
$$S = \frac{t_{old}}{t_{new}}$$

So if originally took 10s, new took 5s, then speedup=2.



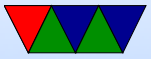
Scalability

- How a workload behaves as more processors are added
- Parallel efficiency: $E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$
- Linear scaling, ideal: $S_p = p$
- Super-linear scaling – possible but unusual



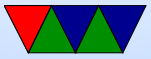
Strong vs Weak Scaling

- Strong Scaling –for fixed program size, how does adding more processors help
- Weak Scaling – how does adding processors help with the same per-processor workload

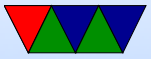


Strong Scaling

- Have a problem of a certain size, want it to get done faster.
- Ideally with problem size N, with 2 cores it runs twice as fast as with 1 core.
- Often processor bound; adding more processing helps, as communication doesn't dominate
- Hard to achieve for large number of nodes, as many

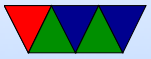


algorithms communication costs get larger the more nodes involved



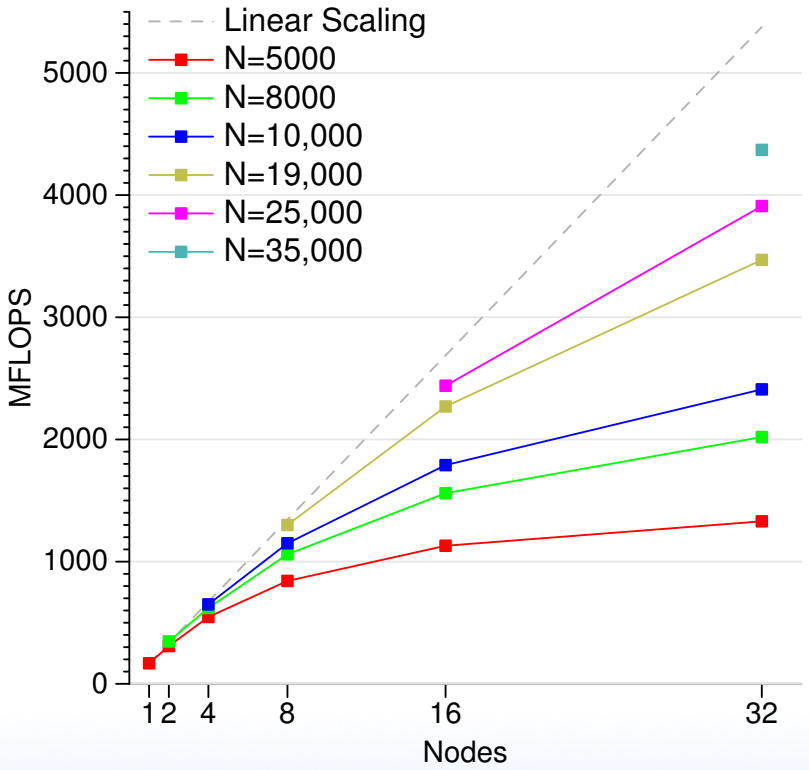
Weak Scaling

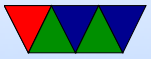
- Have a problem, want to increase problem size without slowing down.
- Ideally with problem size N with 1 core, a problem of size 2*n just as fast with 2 cores.
- Often memory or communication bound.



Scaling Example

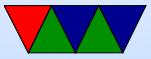
LINPACK on Rasp-pi cluster. What kind of scaling is here?





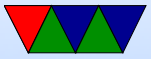
Weak scaling. To get linear speedup need to increase problem size.

If it were strong scaling, the individual colored lines would increase rather than dropping off.



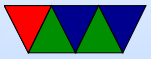
Where Performance Info Comes From

- User Level (instrumentation)
- Kernel Level (kernel metrics)
- Hardware Level (performance counters)

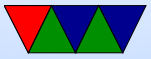


Types of Performance Info

- Aggregate counts – total counts of events that happen
- Profiles – periodic snapshots of program behavior, often providing statistical representations of where program hotspots are
- Traces – detailed logs of program behavior over time



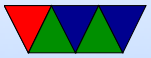
Gathering Aggregate Counts



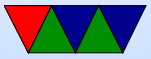
Measuring runtime – using `time`

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

real        0m7.360s
user        0m7.330s
sys        0m0.000s
```

- Real – wallclock time
- User – time the program is actually running (how calculated)
- Sys – time spent in the kernel

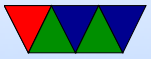


- Must USER+SYS = REAL? Not necessarily (what if other things using the kenrel)
- Can USER be greater than REAL? yes, if multiprocessor
- Is the time command deterministic?
  No. Lots of noise in a system. Can write whole papers on why.
- Which do you use in speedup calculations?

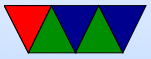


perf tool

```
$ 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 cyc
       91,797,411      branches                  #   12.681 M/sec
          974,817      branch-misses             #    1.06% of all branch

       7.299463710 seconds time elapsed
```

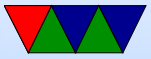


- Many options. Can select events with `-e`
- Use `perf list` to list all available events
- Hundreds of events available on x86, not quite so many on ARM.
- Understanding the results often requires a certain knowledge of computer architecture.



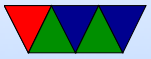
Profiling

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



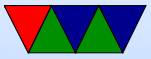
Profiling Tools

- 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



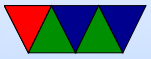
Compiler Profiling

- 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.

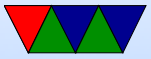
- Pro: available wherever gcc is.



Perf Profiling

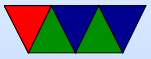
Automatically interrupts program and takes sample every X instructions.

- `perf record`
- `perf annotate`

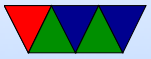


Tracing

- 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



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