# ECE 574 – Cluster Computing Lecture 3

Vince Weaver

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

vincent.weaver@maine.edu

28 January 2024

## Announcements

- HW#1 was graded, you should have gotten an e-mail
- If you did not do HW#1 for any reason (for example, added the class last-minute) let me know, it it still possible to do the assignment. Homeworks are 50% in this class so you don't want to miss any if you can avoid it.

## Notes from Last Time

- Forgot to mention student cluster competition at SC
- Note, if you truly have an interesting parallel project as a student can probably find a place willing to run it for you. The ASC would gladly do this in the old days, not sure if new setup will without some sort of account to charge
- Ethics about working on nuclear weapons?
- H100/A100 GPUs on campus

## Floating point Review

- Whole point of FLOPs
- Easy to picture math on 2s complement integers 8/16/32/64 bit
- If you need fractional parts, you can do fixed point at the expense of size of biggest integer
- Fixed point uses the exact same hardware as regular math, the only real change is you have to shift things back down after multiplies/divides
- To do math on very large/small numbers you probably

## need floating point

## Place Value

- How does it work in base 10?
  $1234.56 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1} + 6 \times 10^{-2}$
- You can do the same thing in binary (base2)
  $1010.10 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2}$
  This is 10.5 in decimal
- You can do this for arbitrary bases.
  You have to keep track of the decimal or “radix” point

## Floating Point

- Tradeoff between range and precision
- Can express very large or very small numbers, but there are gaps with numbers that can't be represented

## IEEE 754 Floating Point

- Standard from 1985
- Before this computers often had custom, incompatible floating point
- Prior to the 1990s was often an optional feature, possibly a separate FPU chip
- What can you do if you lack floating point unit? Emulate in software?

## Floating Point Layout

- $(-1)^{sign}(1.fraction) \times 2^{exponent-bias}$
- bias offsets so that the value is always positive (makes for faster comparisons, no messing with twos complement)

## Floating Point Sizes

- Single precision (32 bit) (float in C)
  Sign=1, Exponent=8, Fraction=23, Bias=127
- Double precision (64 bit) (double in C)
  Sign=1, Exponent=11, Fraction=52, Bias=1023
- Half precision (16 bit)
  Sign=1, Exponent=5, fraction=10, Bias=15
- Intel x86 (80 bits)
- Why have smaller sizes? They take up less room and are faster.

## Floating Point Range

- Any integer with absolute value less than 24 bits can be expressed losslessly in float
- Any integer with absolute value less than 53 bits can be expressed losslessly in double

## Binary FP to Decimal

- You have the value `0xc1ff0000`. If it's a 32-bit floating point value, what is the decimal equivalent?
- `1100 0001 1111 1111 0000 0000 0000 0000`
- Sign bit is 1 (negative)
- Exponent is 8 bits, `1000 0011` which is 131
- Fraction is `1111 1110 0000 0000 0000 000` so 0.1111 111 $1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128 = 0.9921875$
- $f = (-1)^S \times (1 + fraction) \times 2^{exponent-127}$

- $f = (-1)^{-1} \times (1 + 0.9921875) \times 2^{131-127}$
- $f = -1 \times 1.9921875 \times 2^{4}$
- $f = -31.875$

## Convert decimal FP to binary

- Want to convert 6.125 to binary
- **Sign=0**
- Need to divide or multiply by 2 until it is greater than one but less than 2
- $6.125/2 = 3.0625$
- $3.0625/2 = 1.53125$
- so $6.125 = 1.53125 * 2^2$
- so **exponent = 2+BIAS = 2+127=129 = 1000 0001**

- convert fraction (.53125) to binary
- repeatedly multiply fraction by 2
  - 0.53125*2 = 1.0625 **1**
  - 0.625*2= 0.125 **0**
  - 0.125*2= 0.25 **0**
  - 0.25*2 = 0.5 **0**
  - 0.5*2 = 1.0 **1**
  - so **fraction = 100 0100 0000 0000 0000 0000**
- so 32-bit float = 0100 0000 1100 0100 0000 0000 0000 0000
- 0x40C4 0000

## Floating Point Conversion Examples

How to do this? Memcpy? unions? Flirting with undefined behavior?

```
int main(int argc, char **argv) {

        float f;
        unsigned int x;

        x=0x40c40000;
        memcpy(&f,&x,sizeof(float));
        printf("%x is %g\n",x,f);
        f=;
        memcpy(&x,&f,sizeof(float));
        printf("%x is %g\n",x,f);
        return 0;
}
```

40c40000 is 6.125 c1ff0000 is -31.875

## Special Values

- Zero ... cannot represent in standard form (why? because it has to be 1.x in the mantissa). Special value, all zeros. Positive and Negative zeros.
- Pos / Neg infinity: exponent bits all 1s, mantissa all 0s
- NaN (not a number).
  Exponent all 1s, mantissa non-zero
  Things like 0/0, sqrt(-1), log(-1). Two types: QNaN (quiet) which does not cause an exception, and SNaN (signaling) that cause an exception

## Overflow and Underflow

- What's the smallest number you can represent? Smallest exponent 0b00000001, fraction 0000, so
  $(-1)^S \times (1+0) \times 2^{1-127} = \pm 1.18 \times 10^{-38}$
- What's the largest number?
  Maximum exponent 0b11111110 and mantissa all 1s
  $(-1)^S \times (1+1-2^{-23}) \times 2^{254-127} = \pm 3.40 \times 10^{38}$
- What is underflow? Too small but not zero?
- Overflow, too large to be represented

## Subnormal Numbers

- Numbers between smallest and zero
- If exponent is 0, treat leading digit in mantissa as 0 instead of 1
- Can get down to $1.45 \times 10^{-45}$

## Floating Point Comparison

- `float f = (5.0 - 1.0/7.0) + (1.0/7.0);`
- `if (f==5.0) printf("Five exactly.\n")`
- May work may not. Best way is to have some error (epsilon) and compare if absolute value less than a number.

## Special Comparison

- +/- 0 compare equal
- Every NaN not equal to anything, not even itself
- All numbers are greater than -inf but smaller than inf

## Floating Point Rounding Rules

- Complex mess, can cause interesting issues
- Especially as in floating point there are extra bits usually kept around for accuracy, but they are rounded off when written out to memory and you have to fit exactly in 32 or 64 bits
- IEEE-754
  - Round to nearest

    What do we do if *exactly* in between?

    round to even

guard bit, round bit, sticky bits

- ◦ Round toward zero (truncate)
- ◦ Round to $+$inf (round up)
- ◦ Round toward -inf (round down)

## Floating Point Addition

- Shift smaller fraction to match larger one
- add or subtract based on sign bits
- normalize the sum
- round to appropriate bits
- detect overflow and underflow
- do example
- decimal, $5.2 * 10^0 + 9.5 * 10^1$
  - 0.52*10**1
  - 9.50*10**1

- $10.02$
- $1.002 * 10^2$

## Floating Point Multiplication

- Identify the sign
- add the exponents
- multiply the fractions (including leading hidden one)
- normalize the results

## Transcendentals?

- sin(), cos(), etc
- Lookup tables?
- Newton's approximation
- Taylor series expansion
  For example, $cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - ...$
- cordic – (for when multiplies are slow) lookups, shift, add

## FP4

- 4-bit floating point
- Can be very quick on modern GPUs
- Matches well to AI workloads
- E2M1 +/- (0, 0.5, 1, 1.5, 2, 3, 4, 6)
  can also replace 4/6 with NaN / inf
- E3M0

## Introduction to Performance Analysis

## What is Performance?

- Getting results as quickly as possible?
- Getting *correct* results as quickly as possible?
- What about Budget?
- What about Development Time?
- What about Hardware Usage?
- What about Power Consumption?

## Motivation for HPC Optimization

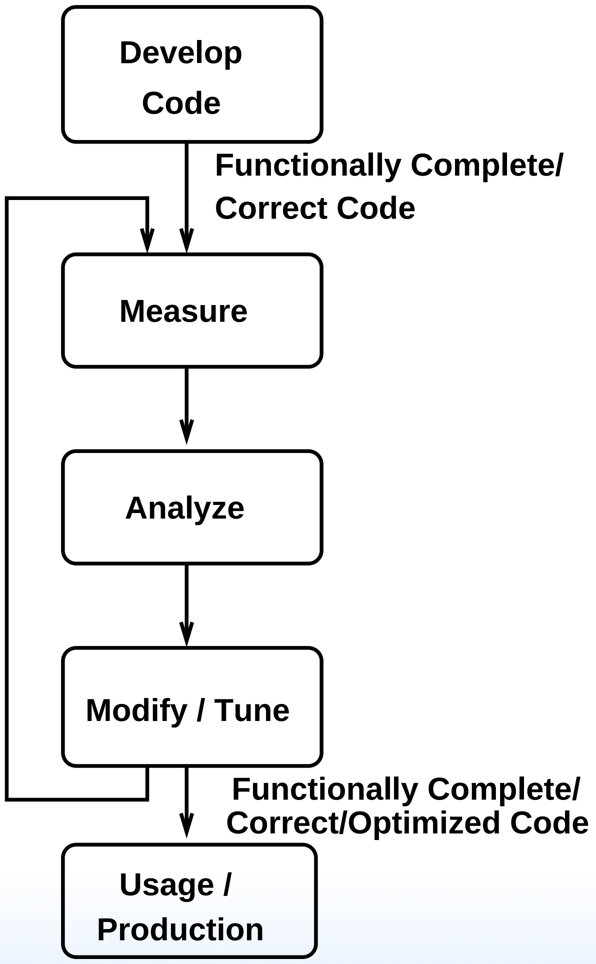
**HPC environments are expensive:**

- Procurement costs: ~$40 million
- Operational costs: ~$5 million/year
- Electricity costs: 1 MW / year ~$1 million
- Air Conditioning costs: ??

## Know Your Limitation

- CPU Constrained
- Memory Constrained (Memory Wall)
- I/O Constrained
- Thermal Constrained
- Energy Constrained

## Performance Optimization Cycle


Develop Code
Functionally Complete/ Correct Code
Measure
Analyze
Modify / Tune
Functionally Complete/ Correct/Optimized Code
Usage / Production

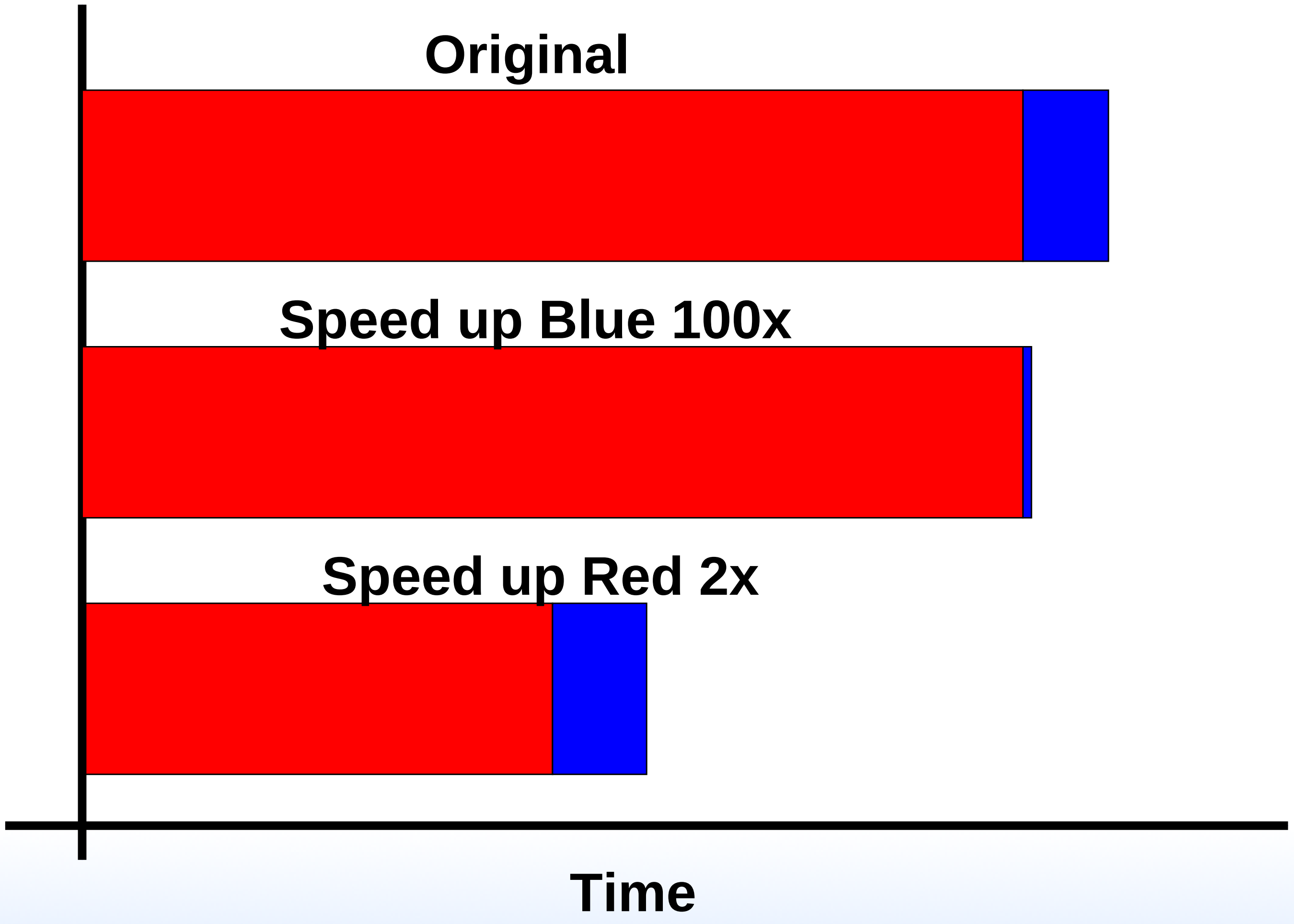
## Wisdom from Knuth

“We should forget about small efficiencies, say about 97% of the time:

**premature optimization is the root of all evil**.

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified” — Donald Knuth

## Amdahl's Law



## 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.

- Good metric for serial code

## Scalability

- How a workload behaves as more processors are added
- Parallel efficiency: $E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$

  p=number of processes (threads)

  $T_s$ is execution time of serial code

  $T_p$ is execution time with p processes
- Linear scaling, ideal: $S_p = p$, $E_p = 100\%$
- Real world it's usually less. Why?
- 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 (linear speedup)
- *NOTE* can still be some amount of strong scaling even if it's not linear!
- 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

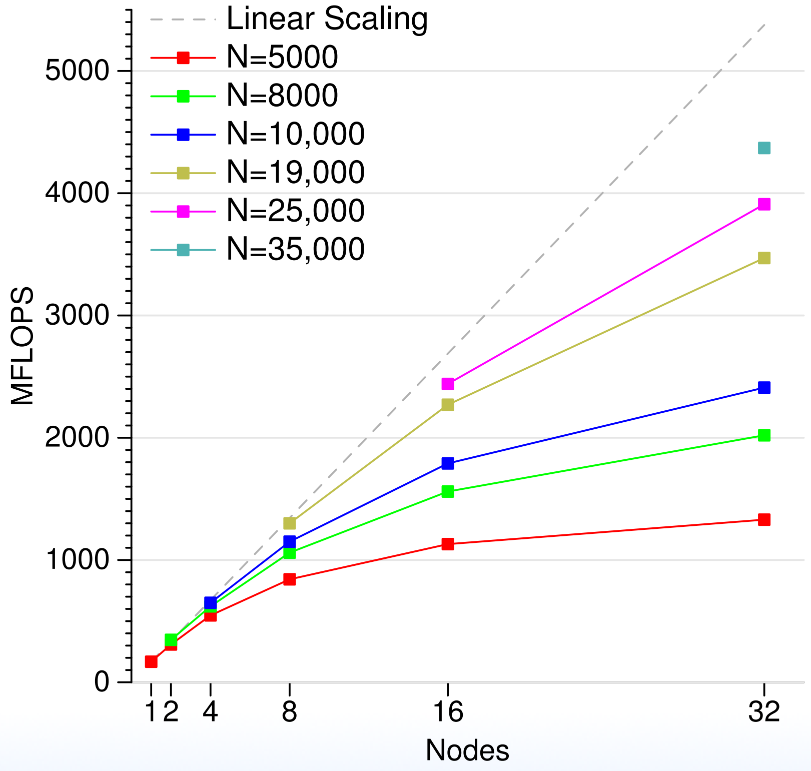
- Amdahl's Law limits things, as more cores don't help serial code
- Improve by throwing CPUs at the problem.

## 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.
- Gustafson's Law (rough paraphrase)
  No matter how much you parallelize your code, there will be serial sections that just can't be made parallel
- Improve by more memory, or faster communication

## Scaling Example

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



There is some strong scaling but it quickly fades after 8 nodes.
There is much more prominent weak scaling, in order to approach a linear speedup you have to increase the workload as you add cores.
This is most likely due to the very slow network connections in this particular cluster.

## Common Performance Analysis Methods

- Aggregate/Overall Measurements
  - Wall clock time
  - Hardware Performance Counters
- Profiling
- Tracing

## Where Performance Info Comes From

- User Level (instrumentation)
  Add timing measurements to own code
- Kernel Level (kernel metrics)
  Kernel tracks metrics on context switch
- Hardware Level (performance counters)
  CPU hardware tracks performance independent of software

## 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

## `time` Corner Cases

- Can REAL be much larger than USER?
  Related: Must USER+SYS = REAL?
  On a heavily loaded multitasking system your program might only get a fraction on the CPU power
- 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?

## `time` Related Note on 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
- Other useful thing, hardware interrupt can be triggered

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

- Pure user events, how can you make sure only belongs to your process?
  Operating system can save/restore registers on context switch