CS 5220: Performance basics

David Bindel

2017-08-24

Starting on the Soap Box

- The goal is right enough, fast enough — not flop/s.
- Performance is not all that matters.
  - Portability, readability, debuggability matter too!
  - Want to make intelligent trade-offs.
- The road to good performance starts with a single core.
  - Even single-core performance is hard.
  - Helps to build on well-engineered libraries.
- Parallel efficiency is hard!
  - p processors $\neq$ speedup of p
  - Different algorithms parallelize differently.
  - Speed vs a naive, untuned serial algorithm is cheating!

The Cost of Computing

Consider a simple serial code:

```
1     // Accumulate C += A*B for n-by-n matrices
2     for (i = 0; i < n; ++i)
3       for (j = 0; j < n; ++j)
4         for (k = 0; k < n; ++k)
5           C[i+j*n] += A[i+k*n] * B[k+j*n];
```

Simplest model:

1. Dominant cost is $2n^3$ flops (adds and multiplies)
2. One flop per clock cycle
3. Expected time is

$$\text{Time (s)} \approx \frac{2n^3 \text{ flops}}{2.4 \cdot 10^9 \text{ cycle/s} \times 1 \text{ flop/cycle}}$$

Problem: Model assumptions are wrong!

The Cost of Computing

Dominant cost is $2n^3$ flops (adds and multiplies)?

- Dominant cost is often memory traffic!
- Special case of a *communication cost*
- Two pieces to cost of fetching data

  Latency Time from operation start to first result (s)

  Bandwidth Rate at which data arrives (bytes/s)
- Usually latency $\gg$ bandwidth^{-1} $\gg$ time per flop
- Latency to L3 cache is 10s of ns, DRAM is 3–4$\times$ slower
- Partial solution: caches (to discuss next time)

See: Latency numbers every programmer should know

The Cost of Computing

One flop per clock cycle? For cluster CPU cores:

$$2\frac{\text{flops}}{\text{FMA}} \times 4\frac{\text{FMA}}{\text{vector FMA}} \times 2\frac{\text{vector FMA}}{\text{cycle}} = 16\frac{\text{flops}}{\text{cycle}}$$

Theoretical peak (one core) is

$$\text{Time (s)} \approx \frac{2n^3 \text{ flops}}{2.4 \cdot 10^9 \text{ cycle/s} \times 16 \text{ flop/cycle}}$$

Makes DRAM latency look even worse! DRAM latency $\sim$ 100 ns:

$$100 \text{ ns} \times 2.4\frac{\text{cycle}}{\text{ns}} \times 16\frac{\text{flops}}{\text{cycle}} = 3840 \text{ flops}$$

The Cost of Computing

Theoretical peak for matrix-matrix product (one core) is

$$\text{Time (s)} \approx \frac{2n^3 \text{ flops}}{2.4 \cdot 10^9 \text{ cycle/s} \times 16 \text{ flop/cycle}}$$

For 12 core node, theoretical peak is $12\times$ faster.

- But lose orders of magnitude if too many memory refs
- And getting full vectorization is also not easy!
- We'll talk more about (single-core) arch next week

The Cost of Computing

Sanity check: What is the theoretical peak of a Xeon Phi 5110P accelerator?

Wikipedia to the rescue!

The Cost of Computing

What to take away from this performance modeling example?

- Start with a simple model
  - Simplest model is asymptotic complexity (e.g. $O(n^3)$ flops)
  - Counting *every* detail just complicates life
  - But we want enough detail to predict something
- Watch out for hidden costs
  - Flops are not the only cost!
  - Memory/communication costs are often killers
  - Integer computation may play a role as well
- Account for instruction-level parallelism, too!

And we haven't even talked about more than one core yet!

The Cost of (Parallel) Computing

Simple model:

- Serial task takes time T (or $T(n)$)
- Deploy p processors
- Parallel time is $T(n)/p$

... and you should be suspicious by now!

The Cost of (Parallel) Computing

Why is parallel time not T/p?

- **Overheads:** Communication, synchronization, extra computation and memory overheads
- **Intrinsically serial** work
- **Idle time** due to synchronization
- **Contention** for resources

We will talk about all of these in more detail.

Quantifying Parallel Performance

- Starting point: good *serial* performance
- Scaling study: compare parallel to serial time as a function of number of processors (p)

$$\text{Speedup} = \frac{\text{Serial time}}{\text{Parallel time}}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{p}$$

- Ideally, speedup = p. Usually, speedup $< p$.
- Barriers to perfect speedup
  - Serial work (Amdahl's law)
  - Parallel overheads (communication, synchronization)

Amdahl's Law

Parallel scaling study where some serial code remains:

$$
\begin{aligned}
p &= \text{number of processors} \\
s &= \text{fraction of work that is serial} \\
t_s &= \text{serial time} \\
t_p &= \text{parallel time} \geq st_s + (1-s)t_s/p
\end{aligned}
$$

Amdahl's law:

$$
\text{Speedup} = \frac{t_s}{t_p} = \frac{1}{s + (1-s)/p} < \frac{1}{s}
$$

So 1% serial work $\implies$ max speedup $< 100\times$, regardless of p.

Strong and weak scaling

Ahmdahl looks bad! But two types of scaling studies:

Strong scaling Fix problem size, vary p

Weak scaling Fix work per processor, vary p

For weak scaling, study *scaled speedup*

$$S(p) = \frac{T_{\text{serial}}(n(p))}{T_{\text{parallel}}(n(p), p)}$$

Gustafson's Law:

$$S(p) \leq p - \alpha(p - 1)$$

where α is the fraction of work that is serial.

Pleasing Parallelism

A task is “pleasingly parallel” (aka “embarrassingly parallel”) if it requires very little coordination, for example:

- Monte Carlo computations with many independent trials
- Big data computations mapping many data items independently

Result is “high-throughput” computing – easy to get impressive speedups! Says nothing about hard-to-parallelize tasks.

Dependencies

Main pain point: *dependency* between computations

```
1    a = f(x)
2    b = g(x)
3    c = h(a,b)
```

Compute `a` and `b` in parallel, but finish both before `c`!
Limits amount of parallel work available.

This is a true dependency (read-after-write). Also have false dependencies (write-after-read and write-after-write) that can be dealt with more easily.

Granularity

- Coordination is expensive — including parallel start/stop!
- Need to do enough work to amortize parallel costs
- Not enough to have parallel work, need big chunks!
- How big the chunks must be depends on the machine.

Patterns and Benchmarks

If your task is not pleasingly parallel, you ask:

- What is the best performance I reasonably expect?
- How do I get that performance?

Look at examples somewhat like yours – a *parallel pattern* – and maybe seek an informative benchmark. Better yet: reduce to a previously well-solved problem (build on tuned *kernels*).

NB: Easy to pick uninformative benchmarks and go astray.