Engineering Robust Server Software

Scalability
Intro To Scalability

• What does scalability mean?
• **What does scalability mean?**
  - How does performance change with **resources**?
• What does scalability mean?
  • How does performance change with resources?
  • How does performance change with load?
Scalability Terms

- **Scale Out**: Add more nodes
  - More computers
- **Scale Up**: Add more stuff in each node
  - More processors in one node
- **Strong Scaling**: How does time change for fixed problem size?
  - Do 100M requests, add more cores -> speedup?
- **Weak Scaling**: How does time change for fixed (problem size/core)?
  - Do (100*N)M requests, with N cores -> speedup?
Amdahl's Law

Speedup (N) = \frac{S + P}{S + \frac{P}{N}}
Amdahl's Law

\[
\text{Speedup (N)} = \frac{S + P}{S + \frac{P}{N}} = \frac{6 + 4}{6 + \frac{4}{N}}
\]

Serial Portion: \( S = 6 \)

Parallel Portion: \( P = 4 \)
Amdahl's Law

\[
\text{Speedup (N)} = \frac{S + \frac{P}{N}}{S + \frac{P}{N}} = \frac{6 + \frac{4}{2}}{6 + \frac{4}{2}} = \frac{10}{8}
\]

Serial Portion: \( S = 6 \)
Parallel Portion: \( P = 4 \)

- \( 10/8 = 1.25x \) speedup = 25% increase in throughput.
- \( 8/10 = 0.8x \) = 20% reduction in latency
Amdahl's Law

\[
\text{Speedup (N)} = \frac{S + P}{S + \frac{P}{N}} = \frac{6 + 4}{6 + \frac{4}{4}} = \frac{10}{7}
\]

Serial Portion: \(S = 6\)

Parallel Portion: \(P = 4\)

- \(10/7 = 1.42x\) speedup = 42% increase in throughput.
- \(7/10 = 0.7x\) = 30% reduction in latency
Amdahl's Law

\[
\text{Speedup (N)} = \frac{S + P}{S + \frac{P}{N}} = \frac{6 + 4}{6 + \frac{4}{\infty}} = \frac{10}{6}
\]

- Serial Portion: \(S = 6\)
- Parallel Portion: \(P = 4\)

\[
\frac{10}{6} = 1.67x \text{ speedup} = 67\% \text{ increase in throughput.}
\]

\[
\frac{6}{10} = 0.6x = 40\% \text{ reduction in latency}
\]
Amdahl's Law

• Anne Bracy: "Don't try to speed up brushing your teeth"
  • What does she mean?

Brushing your teeth doesn’t take long, so you can’t save much total time by optimizing it.
Why Not Perfect Scalability?

- Why don't we get \((Nx)\) speedup with \(N\) cores?
- What prevents ideal speedups?
Impediments to Scalability

- Shared Hardware
  - Functional Units
  - Caches
  - Memory Bandwidth
  - IO Bandwidth
  - ...
- Data Movement
  - From one core to another
- Blocking
  - Locks (and other synchronization)
  - Blocking IO

Let's talk about these for now
A core has 2 threads (2-way SMT)
- Also private L1 + L2 caches (not shown)
Hypothetical System

4 cores share a **last-level cache (LLC)**, e.g., L3 cache
- Connected by on chip interconnect
Hypothetical System

We have a 2 socket node
- Has 2 chips
- DRAM
- Also some IO devices (not shown)
And a Real Example (IBM POWER9 CPU)
Hypothetical System

We have 2 nodes
Hypothetical System

Suppose we have 2 requests: where best to run them?
Hypothetical System

Different threads on same core?
Different cores on same chip?
Hypothetical System

Different chips on same node?
Hypothetical System

Different nodes?
How To Control Placement?

• **Within a node:** `sched_setaffinity`
  • Set mask of CPUs that a thread can run on
  • SMT contexts have different CPU identifiers
  • In pthreads, library wrapper: `pthread_setaffinity_np`

• **Across nodes:** it depends…
  • Daemons running on each node? Direct requests to them
  • Startup/end new services? Software management
  • Load balancing becomes important here
Tradeoff: Contention vs Locality

Increasing Communication Latency

- Same Core
- Same Chip
- Same Node
- Different Node

Increasing Contention

- Trade off:
  - Contend for shared resources?
  - Longer/slower communication?
Tradeoff: Contention vs Locality

Increasing Communication Latency

- **Same Core**
  - Loads + Stores
  - Same Cache
  - 1s of cycles

- **Same Chip**
  - Loads + Stores
  - On Chip Coherence
  - 10s of cycles

- **Same Node**
  - Loads + Stores
  - Off Chip Coherence
  - 100s cycles

- **Different Node**
  - IO Operations
  - Network
  - Ks-Ms of cycles
• Non Uniform Memory Access (NUMA—technically, ccNUMA)
  • Memory latency differs depending on physical address
  • migrate_pages, mbind: control physical memory placement
Tradeoff: Contention vs Locality

- Same Core
- Same Chip
- Same Node
- Different Node

Increasing Contention
Re-examine Our Hypothetical System
Tradeoff: Contention vs Locality

- External network bandwidth
- Datacenter cooling
- Memory bandwidth
- Chip<-> chip bandwidth
- IO bandwidth
- On chip bandwidth
- LLC capacity
- On chip cooling
- L1/L2 capacity
- Functional Units

Same Core  |  Same Chip  |  Same Node  |  Different Node

Increasing Contention
Interactions Between Resource Contention

- Suppose two threads need and are sensitive to:
  - LLC Capacity
  - Memory bandwidth
- What happens when we run them together?
  - Contention for LLC -> more cache misses
    - Slows down program, but also…
    - Increases memory bandwidth demands
      - Which we already need and are contending for :(
  - Interactions can make contention even worse!
    - Is there a flip side?
Improved Utilization

• Can improve utilization of resources
  • One thread executes while another stalls
  • One thread uses FUs that the other does not need
  • Pair large cache footprint with small cache footprint
  • Shared code/data: one copy in cache
Strategy: How to improve Performance/Scalability

• So how do we improve things?
  • Profile our system! Understand what is slow and why
  • Remember: Amdal's law!
Remember this graph?

This graph hides a lot of important detail
Remember this graph?

Breaking it down shows WHERE to focus our optimization efforts.
Strategy: How to improve Performance/Scalability

• So how do we improve things?
  • **Profile our system!** Understand what is slow and why
  • Remember: Ahmdal's law!

• After making a change, what do we do?
  • Measure impact: did we make things better? How much?
Tactics: How to improve Performance/Scalability

- So what can we do to improve the slow code?
  - Optimize code to improve its performance
  - Transform code to improve resource usage (e.g. cache space)
  - Pair threads with complementary resource usage

- Sounds complicated?
  - Learn more about hardware (e.g., ECE 552)
  - Take Performance/Optimization/Parallelism (ECE 565)
Impediments to Scalability

• Shared Hardware
  • Functional Units
  • Caches
  • Memory Bandwidth
  • IO Bandwidth
  • ...

• Data Movement
  • From one core to another

• Blocking
  • Blocking IO
  • Locks (and other synchronization)

Let's talk about this next
Never Block

• Critical principle: never block
  • Why not?
• Can't we just throw more threads at it?
  • One thread per request (or even a few per request)
  • Just block whenever you want
• Nice in theory, but has overheads
  • Context switching takes time
  • Switching threads reduces temporal locality
    • Threads not blocked? May thrash if too many
  • Threads use resources
Non-Blocking IO

- IO operations often block (we never want to block)
  - Can use non-blocking IO
- Set FD to non-blocking using fcntl:
  - `int x = fcntl(fd, F_GETFL, 0);`
  - `x |= O_NONBLOCK;`
  - `fcntl(fd, F_SETFL, x);`
- Now reads/writes/etc won't block
  - Just return immediately if can't perform IO immediately
  - Note: not magic
    - ONLY means that IO operation returns without waiting
Non-Blocking IO: Continued

```c
int x = read(fd, buffer, size);
if (x < 0) {
    if (errno == EAGAIN) {
        // no data available
    }
    else {
        // error
    }
}
```
Non-Blocking IO: Continued

```c
while (size > 0) {
    int x = read(fd, buffer, size);
    if (x < 0) {
        if (errno == EAGAIN) {
            //no data available
        }
        else {
            //error
        }
    } else {
        buffer += x;
        size -= x;
    }
}
```

What if we just wrap this up in a while loop?
Non-Blocking IO: Continued

```c
while (size > 0) {
    int x = read(fd, buffer, size);
    if (x < 0) {
        if (errno == EAGAIN) {
            // no data available
        }
        else {
            // error
        }
    }
    else {
        buffer += x;
        size -= x;
    }
}
```

What if we just wrap this up in a while loop?

Now we just made this blocking!

We are just doing the blocking ourselves...
Busy Wait

- This approach is **worse** than blocking IO
  - Why?
- Busy waiting
  - Code is "actively" doing nothing
  - Keeping CPU busy, consuming power, contending with other threads
- Blocking IO:
  - At least OS will put thread to sleep while it waits
So What Do We Do?

• Need to do **something else** while we wait
  • Like what?
• It depends….  
  • On what?
• On what our server does
• On what the demands on it are
• On the model of parallelism we are using
  • Who can name some models of parallelism? [AoP Ch 28 review]

Data parallelism, Pipeline parallelism, Task parallelism, etc.
Pipeline Parallelism

- When would this be appropriate?
- What do our IO threads do for "something else"?
Pipeline Parallelism

• When appropriate: Can keep IO thread(s) busy
  • Heavy IO to perform
  • Might have one thread do reads and writes

• What is "something else"?
  • Other IO requests
  • Do whichever one is ready to be done

• Making hundreds of read/write calls to see which succeeds = inefficient
  • Use poll or select
Pipeline Parallelism

- What can you say about data movement in this model? There’s lots of it 😑
- What can you say about load balance? It’s pretty easy to balance (just profile it) 😊
Another Option

• Could have one thread work on many requests
  
  while(1) {
    Accept new requests
    Do any available reads/writes
    Do any available compute
  }

• What can you say about data movement in this model?  
  There’s less of it 😊

• What can you say about load balance?  
  It’s harder: a thread can end up with imbalanced tasks 😒
A Slight Variant

• Slightly different inner loop:
  
  while(1) {
    Accept new requests into a shared pool
    For each request with anything to do
      Do any available IO for that request
      Do any compute for that request
  }

• What can you say about data movement in this model?
  In between 😊 (We do compute on the IO we just did, so less movement there)

• What can you say about load balance?
  Good, all threads can hit all request 😊
Summary

- **Scalability**: how performance changes with resources and load
- **Amdahl's law**: You can only speed up the parallelizable part of code
- **Impediments to scalability**:
  - Shared hardware (tradeoff in contention vs locality)
  - Communication
  - Blocking
- **Improving scalability**
  - **Strategy**: **Profile** (measure performance) to find bottleneck code
  - **Tactics**:
    - Better algorithms/data structures (improve big-O), better locality (cache optimizations), etc.
    - Blocking: try to minimize it with non-blocking modes when appropriate