Concurrency and the Real World

Richard Carlsson

Hardware and Software

Modern hardware

- The frequency race is long since over
  - Basically same clock speeds as 20 years ago
- Power consumption and heat is a big issue
  - Mobile devices – conserve battery
  - Data centers – power and cooling is expensive
- Multiple cores running on reasonable frequencies
  - Same amount of work done, less power and heat
  - But only if the work can be split up

Modern hardware

- RAM is cheap – but coherence doesn't scale
  - A modern memory system is a distributed database!
- Disks are really slow, even SSD
- Networks can be faster than disks
- Huge differences between local network and servers in the cloud

Heterogenous environments

- Modern systems contain lots of different machines
  - Load balancers, SSL termination, ...
  - Front end servers, database servers, authentication, ...
  - Logging, backup, monitoring, …
- Many or all of these may be virtual servers in a cloud in a data center in another country

Modern software

- To do more work, you need to split up your code to use multiple cores!
- Software is adapting very slowly to multicore
- Very dependent on the problem space

Utilizing multicore

- Desktop software only does it in special cases
  - Photo, video, sound processing, etc. are easy to parallellize – just split up the work into pieces
- Games: rendering, AI, physics, etc.
  - Still much to do – typically only a few main threads
  - Imagine running the AI for each NPC as a separate thread, or the physics for each object

Horizontal scaling

- Server software is easiest to parallellize
  - Requests from clients can be handled independently
  - Multiple servers behind a load balancer
  - One thread per client connection – possibly more
  - But the code that handles a request is usually sequential
    - If a single client submits a complicated request, it can still take a long time, no matter how much you scale horizontally

Vertical scaling

- Making the individual servers faster
  - Fast CPU
  - Fast disks and I/O hardware
  - Lots of RAM to keep files in cache
- Very expensive
  - Can’t afford many of them
- Only speeds up requests by a small linear factor
  - Mainly a way to buy time

Legacy software

- Legacy means that the people who wrote it have moved on to other things and nobody knows all the details of how it works and why
- A single company can have over a million lines of code – and it keeps working, but nobody understands the whole thing
- You rarely get to start completely from scratch
  - When you do, you don't think that your code might live for 10-20 years and will need to scale 1000 times

Modifying legacy code

- Parallelizing code can be hard even for a small self-contained algorithm
- It is 10 times harder if you don't fully understand what the code is doing
  - Side effects make code hard to understand/change
  - Code that relies on a shared memory space must be completely rewritten to run on separate machines
  - Functional code is easier to refactor

Understanding Concurrency

2023-05-15

What is time?

- You need to relearn how to think about time
  - It takes experience – you have to build a new intuition
- Clocks are mostly useless
- As in relativistic physics, there is no global ”right now”
- We can only know what has happened, not what is happening

Logical time

- Only messages and replies can be relied on
  - Logical time: discrete points, partial order
  - Causal relationships: if B, then A must have happened
- Even the smallest step in your program can take from a nanosecond to several hours
  - CPU scheduling, memory stalls
  - I/O and hardware driver delays

What is concurrency?

- If two things can *potentially happen at the same time*, they are said to be **concurrent**
  - Concurrent things do not have any order dependencies between them
  - Maybe they *usually* happen in a certain order, but if a delay happens somewhere, the order could change
  - Delays can occur at any point, and for any length of time

Never rely on timing

- The behaviour under real load will be different from in development and testing – if the program can fail, it will!
- You have to synchronize to ensure ordering
- ”Fixing” a timing problem by adding a pause is broken from start, and will come back to bite you later

What is parallelism?

- Utilizing multiple **resources** in parallel
  - CPUs, GPUs, RAMs, data buses, interfaces, disks
- Concurrency is needed for parallelism
  - A completely non-concurrent program has nothing that can be done in parallel at any time
- Parallelism is not needed for concurrency
  - A concurrent program can use a single resource by interleaving the separate tasks in small portions

Resources and bottlenecks

- Whenever concurrent tasks want to use the same limited resource, you get a bottleneck
  - A narrow passage that everything has to go through
- No task can make progress until it has passed the bottleneck
  - Processes fighting over CPU or RAM resources
  - Processes fighting over disk access or network bandwidth
  - Threads killing each other's cache behaviour

Designing for concurrency

- Keep it simple!
  - Knowing that it works is always more important
- Think about your program as a set of services
  - Within each service, let each activity be a process
  - No process should know more than it needs to know
  - Always keep in mind that you may want to split out the different parts to run on separate machines
  - Avoid traditional ”object-oriented” design, which can lead to a spaghetti of shared data dependencies

Programming Techniques

2023-05-15

Fail fast and noisily

- The program should fail immediately on errors
  - The longer it tries to keep going, the harder it will be to figure out what really went wrong
- Write to proper logs, don't print to stdout!
- Check inputs at the borders of your program
  - Internally, assume that the inputs are correct
- Let it crash!
  - Don't try to detect errors and ”autocorrect” them – crash the program and let someone else restart it

Separate all the things

- Traditional optimizations try to do as much as possible at the same time, in a single thread
  - For example, loop over all items in an array and do a whole bunch of different things to each item
  - This makes it impossible to utilize multicore, since you cannot easily split the code into separate processes
- Doing several unrelated things in the same piece of code, just because you have the information available, is a really bad idea

Controlling your parallelism

- Handling a million requests in parallel is great. Making a million mistakes in parallel is bad!
  - Bugs, resource limits (file descriptors, memory)
  - If one worker fails, the others will probably fail too
    - For one thing, this can flood your logs
  - Circuit breaker pattern
- Make features configurable: runtime on/off switch

Tracking your parallelism

- An individual ”task”, such as a user’s login session, can involve calls to many different services in your network: database lookups, authorization checks, purchase systems, etc.
- Timestamps are useful but not enough to efficiently correlate events across systems.
- Include Pids or session IDs in all log messages so you can follow what a session has done, across services and across machines

Splitting shared resources

- When parallelizing, you will often find a few central things that all your processes need to access
  - Typical example: global counters (session IDs, etc.)
  - Performance bottleneck – everything gets serialized
  - Single point of failure – if it goes offline, nothing works
- ”No global lock ever goes unpunished”

Eliminate the sharing

- Must think outside the box
  - Give individual processes the resources to keep working independently for a while
    - Global server handing out number series to each front end server on demand, enough for hours or days
  - Does the application really require guaranteed uniqueness, or is very low probability of collisions enough? Use a hash!

Explicit sequencing

- You sometimes find that parts of your system need to be explicitly serialized to avoid race conditions or overload
- Often handled by database transactions, but you may need to do it all by yourself
  - E.g., send all such requests through a single server
- For example, to implement a circuit breaker.

Know what is important

- Which of your services *must* always be available to keep customers happy, and which are just ”extras”?
  - Don't let non-critical stuff bring your system down. Ensure that the system can keep going without it.
- Service Level Agreement
  - ”We promise that...” (usually one business to another)
  - E.g., ”99% of responses within 100ms”
  - Typically fines paid if SLA levels are violated

Think about the unthinkable

- Which errors are critical, and which are just ”bad” but can be fixed?
  - Sold a book that was out of stock? Mail customer about delay, order more books from publisher.
  - Two customers booking same room is pretty bad, but can be fixed if another hotel has a room.
  - Two planes on the same runway must simply never happen.

Persistent Storage

Most systems need a database

- You will need to keep track of stuff
- Database = global shared memory
  - All of the code assumes it can access all of the data
  - Often leads to bad code with lots of dependencies
- Databases usually imply transactions
  - Transactions imply bottlenecks (locks or collisions)
  - You have to weigh consistency against bottlenecks

File systems are databases

- Don’t think you can get around database limitations by “simply using the file system”
- Hierarchical, good at “file-ish” things, bad at other things, usually no transactions
- Ultimately, has the same technical limitations as any other database
- If you need a database, use a suitable one

Availability and Consistency

- **Availability**
  - Your system can keep working and provide an answer even if a subset of the servers have failed
    - Obviously not possible with only a single server
- **Consistency**
  - It doesn’t matter which of your servers (of those that that still work) that you ask – they will all provide the same answer
    - Implies that the servers must talk to each other about what they do, so they can have the same picture
    - **Partitioning** means the servers get cut off from each other

Databases and distribution

- To guarantee **availability**, you need *more than one machine*, and *more than one copy of your data*
  - Any machine can die at any time (power failure, hardware problems, overheating, etc.)
  - Disks can go offline, die, or get corrupted
    - Having backups does not mean availability – it can take hours to restore a system from backup tapes

The CAP Theorem

- Consistency, Availability, Partition tolerance
  - You can't have all 3 at the same time - pick 2 (CP/AP)
  - You *need* availability, or you could be out of business
  - You *will* get partitioning; in a real system, networks fail
- CP: Drop availability in case of partitioning
  - Customers will have to wait until it gets fixed
- AP: Drop consistency in case of partitioning
  - Inconsistencies must be resolved afterwards

Consistency is overrated

- Transactions need to (mostly) go away if you want availability and performance
  - Eventual consistency – allow temporary inconsistency, and resolve it when the system is fully connected
  - No simple rules (yet) for writing transactionless systems
- CRDTs (conflict-free replicated data types) may be the future
  - Still a lot of work to be done in that area
  - Not like your usual lists and hash tables

Erlang – Built for Concurrency

Erlang design philosophy

- Isolation of processes, no shared memory, message passing, error handling via links
  - A process can't corrupt the state of another process
- Mostly functional programming, few side effects (messages), easy to reason about
- Service oriented software design
  - Divide your program into small components, no shared state

Erlang and multicore

- The Erlang philosophy is a perfect fit for multicore programming
  - No need for locks or synchronized sections
  - No shared memory – a process can run anywhere
- Erlang gives you built-in support for multicore, but can also run the same concurrent code on a single-core machine

Erlang and clusters

- Erlang lets you run code on separate machines over the network just as easily as on a single machine
- Generally, without rewriting any of the code

When to use Erlang

- If your program has a lot of concurrency
- If you communicate with other computers
  - E.g., supervising and controlling other systems
- If you want to get something working quickly
- If you need to parse binary formats
- If your code needs to keep running forever

Why not use ... instead?

- Virding's Law
  - ”Any sufficiently complicated concurrent program in another language contains an ad hoc informally-specified bug-ridden slow implementation of half of Erlang.”
- Erlang lets you get close enough to the best possible performance in a very short time
  - The code is clean and easy to modify further

When Erlang is not a good fit

- If your program is not really concurrent
  - Unless you simply like Erlang as a kind of Lisp/Scheme with pattern matching, list comprehensions, bit syntax, tracing, etc.
- If you really need shared memory
  - For example high-performance number crunching
- If you mainly work with strings
  - Erlang is not a replacement for Perl or Python

Final words

Concurrency is hard – or not

- Most people aren't used to thinking about concurrent programming – it takes practice
- On the other hand, as a human you deal with concurrency every day
  - Drinking coffee and answering email while talking to someone; not bumping into people in the corridor; watching TV and eating cheetos while texting a friend…
  - If it works for people, it can work for computers

Play around with Erlang!

- Erlang is great for getting a feeling for concurrency
- It lets you quickly try out things that would take a lot of time to do in most other languages

We live in exciting times

- Few textbook examples of how to solve typical problems without resorting to transactions
- Huge demand for systems that scale well and remain available despite partial failures
- Lots of questions and few hard answers

Know what you’re doing

- Most situations need a combination of engineering and domain specific workarounds
- High performance parallel (scientific) computing, distributed web services, games, AI and robotics, etc., all have very different requirements. Pick a language that suits the problem.

The End