Parallel Programming

Janis Hutz
https://janishutz.com

August 21, 2025

1 Formulas

Amdahl's Law **Formula 1.1**

Let W_{ser} be non-parallelizable work, W_{par} parallelizable work, $T_1 = W_{ser} + W_{par}$ the processing time on one processor and T_p the time taken on p processors.
From this we can get an upper bound for speed-up, given by $T_p \geq W_{ser} + \frac{W_{par}}{P}$. Then S_p denotes the speed-up for p processors, given by

$$S_p \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{P}}$$

If now f is the fraction of work that is non-parallelizable, then $W_{ser} = fT_1$ and $W_{par} = (1-f)T_1$ and we get

$$S_p \leq \frac{1}{f + \frac{1-f}{P}}$$

On the other hand, Gustafson's law is an optimistic take on Amdahl's law. So, where Amdahl's law asks for doing the same work faster, Gustafson's law asks for more work to be done in the same time

Gustafson's law **Formula 1.2**

Again, f denotes the non-parallelizable work, p denotes the number of processors and T_1 denotes time for one processor, where T_L is the constant time assumed

$$W = p(1-f)T_L + fT_L$$
$$S_p \leq \frac{T_1}{T_P} = f + p(1-f)$$

- t_{max} := Time for longest stage in pipeline
- **Definition 1.1:** *(Latency)* L Sum of all stages in pipeline
- **Definition 1.2:** *(Balanced Pipeline)* All stages take equally long
- **Definition 1.3:** *(Throughput)* $\frac{1}{t_{max}}$

Time for i iterations $i \cdot t_{max} + (L - t_{max}) = (i-1)t_{max} + L$

2 Java BS

`ExecutorService` is not suited to Divide & Conquer (or any other non-flat structure), we can give it `Callable` (need to implement `call` method, which returns) or a `Runnable` (need to implement `run` method, doesn't return). We can create an `ExecutorService` using `java.util.concurrent.Executors.newFixedThreadPool(int threads)` and we can add tasks using `ex.submit(Task task)` When submitting `Callable` to `ExecutorService`, a `Future` is returned (= Promise).

For Divide & Conquer, use `java.util.concurrent.ForkJoinPool`, to which we submit `RecursiveTask` (returns) or `RecursiveAction` (doesn't return). `RecursiveTask` and `RecursiveAction` support the following methods:

- `compute()` runs the new task in the current thread
- `fork()` runs the task in a new thread
- `join()` waits for the task to finish (like the `join()` of Threads)

To start execution, run `pool.invoke(Task task)` on the `ForkJoinPool`

3 Locking

For synchronization, we also have `volatile`, which does guarantee that all threads see the value immediately (as the variable is written back to memory immediately and not stored in each Thread's internal cache) and it enforces memory consistency, i.e. instructions are not reordered around such variables. It however is not atomic and does not guarantee that two consequent accesses (e.g. read and increment) could be reordered / interfered with, just as if we used a normal variable

A sequentially consistent memory model enforces that actions of threads become visible in program order.

If we need atomic operations, they are available in `java.util.concurrent.atomic`, e.g. `AtomicInteger`. These variables provide the following methods: `get()`, `set(E newValue)`, `compareAndSet(E expect, E newValue)` (CAS operation, which sets if the value of the variable is equal to `expect`) and `getAndSet(E newValue)` (Updates & returns old value).

We also have the TAS (Test And Set) operation, which is, like CAS, provided by Hardware and allows us to test if value is 0 to set it to 1 and get a return of `true` if it was zero.

For TAS based locks, use exponential back-off (wait exponentially longer between accesses)

3.1 Monitors

Inside `synchronized` blocks, we can use `wait()` (releases lock and waits to be woken again), `notify()` (wakes up *random* Thread) and `notifyAll()` (wakes up all threads). The `wait()` should *always* be inside a for loop, as the condition could be incorrect when the thread is woken.

If we however want to manually acquire locks with the Java Lock interface, we need to use `Conditions`, which we can obtain using `lock.newCondition()`. They offer `await()`, `signal()` and `signalAll()`, which all work similar to their `synchronized` counterparts.

IMPORTANT Always use `try-catch-finally` blocks around locked elements to ensure lock is released again to avoid deadlocks

Finally, some concepts for locking:

- Coarse grained locking: One lock for entire structure, very safe, but very slow
- Fine grained locking: In lists, every element has lock, lock previous and current element, to move through list, lock next, then release previous, move to next. For a list, we then need to lock (*number of elements*) +1 (for head) for traversal and for insert at the end a further one time for tail.
- Optimistic synchronization: Traverse list without locking, then lock when updating / reading. Much faster, but need to traverse twice and not starvation free. We just need to lock the predecessor and tail to insert and for a contains operation, predecessor and current node.

Monitor locks are reentrant locks in Java.

A `static synchronized` method locks the whole class, not just the instance of it.

3.2 Lock-free programming

- Wait-Free ⇒ Lock-free
- Wait-Free ⇒ Starvation-free
- Lock-Free ⇒ Deadlock-free
- Starvation-Free ⇒ Deadlock-free
- Starvation-Free ⇒ Livelock-free
- Deadlock-Free AND fair ⇒ Starvation-free

To program lock-free, use hardware concurrency features like TAS & CAS

3.3 ABA-Problem

Occurs if a thread fails to recognize that a variable's value was *temporarily* changed (and the changed back to the original), thus not noticing state change

Solutions: DCAS (Double Compare And Set, not available on most platforms), GC (Garbage Collection, very slow), Pointer-Tagging (Only delays problem, but practical), Hazard Pointers (Before reading, pointer marked as hazard), Transactional Memory

4 Consistency / Linearisability

Between Invocation and Response states, method in pending state.

Linearization Each method should appear to take effect *immediately*. When deciding whether or not something is linearizable, decide if there is an order of commit such that the desired effects happen. A commit can happen at any point during a function's life-cycle and the same applies to a read / dequeue, etc.

History Complete sequence of invocations & responses. History linearisable if it can be extended to another one by adding ≥ 0 responses that took effect, or discard ≥ 0 pending invocations that have not taken effect (yet). It is sequentially consistent, if we can add ≥ 0 pending responses, or same as before (Linearizability implies Sequential Consistency)

We can check if a history is linearizable, if we can find linearization points such that the responses are correct.

It is sequentially consistent, if we can find a sequential execution order (non-interleaved calls) such that the history is valid. We are allowed to move operations of other threads in between to make the result correct, but we are not allowed to change the order of operations in a thread. We may only reorder if the operations are not overlapping.

The below history:

```
A: r.write(2)
A: r:void
A: r.write(1)
A: r:void
B: r.read()
B: r:2
```

Can be rewritten as:

```
A: r.write(2)
A: r:void
B: r.read()
B: r:2
A: r.write(1)
A: r:void
```

And is thus sequentially consistent. (The history is thus also sequential, as actions between multiple threads are not interleaved)

A more detailed explanation:

- For Linearisability, we need to respect the program order and the real-time ordering of the method calls (i.e. we can't move them around)
- For sequential consistency, we only need operations done by a single thread to respect program order. Ordering across threads not needed.

5 Consensus

n threads should agree on picking one element of e.g. a list. Consensus number is largest number of threads such that the consensus problem can be solved. **Ex 5.1:** Atomic reg: 1; CompareAndSwap: ∞; wait-free FIFO queue: 2; TAS & getAndSet: 2

6 Transactional Memory

Atomic by definition, programmer defines atomic code sections. Issue: Still not standardized / WIP

7 Message Passing Interface

Used to send messages between threads. Other threads can choose when to handle, if at all