Concurrency and parallelism

Tasks, threads, channels, and what does not transfer from the BEAM

Nothing here resembles the BEAM. Shared mutable memory, real OS threads, cooperative scheduling, no process isolation, no supervision, no mailboxes bound to addresses. Data races are possible and silent.

Four mechanisms, for four different problems:

Mechanism	For	Memory	Parallel?
`@async` / `Task`	I/O concurrency	shared	no (concurrent only)
`Threads.@spawn` / `@threads`	CPU work	shared	yes
`Distributed` (`addprocs`)	multi-process / multi-machine	separate	yes
`@simd`, vectorisation	inner loops	—	instruction-level

Tasks (coroutines)

Green threads, cooperatively scheduled. They yield at I/O, `sleep`, `wait`, channel operations — never preemptively. A tight CPU loop in a task blocks everything on that thread.

```
t = @async begin
    sleep(1)
    "done"
end

istaskdone(t); istaskstarted(t); istaskfailed(t)
wait(t)              # block until finished, rethrow on failure
fetch(t)             # wait and return the value
```

Concurrent I/O:

```
tasks = [@async download(u) for u in urls]
results = fetch.(tasks)
```

trap: an exception in a task is stored, not raised. If you never `fetch` or `wait`, the failure is invisible. Always collect your tasks.

```
@sync begin              # waits for all tasks spawned inside, and surfaces errors
    for u in urls
        @async process(u)
    end
end
```

`@sync` is the structured-concurrency primitive; use it rather than a bare list of `@async` calls whenever you can.

Threads

Start Julia with threads or you get one:

```
julia -t auto            # or JULIA_NUM_THREADS=auto
```

```
Threads.nthreads()
Threads.threadid()       # do NOT use this to index into per-thread state; see below
```

`@threads` — static partition of a loop across threads. Best for uniform work over an indexable range.

```
Threads.@threads for i in eachindex(v)
    v[i] = expensive(i)          # distinct indices per iteration: safe
end
```

`@spawn` — dynamic scheduling, composable, nests properly. Prefer it for uneven work and for anything that itself spawns.

```
tasks = map(chunks) do c
    Threads.@spawn process(c)
end
results = fetch.(tasks)

# divide and conquer
function psum(v)
    length(v) < 10_000 && return sum(v)
    mid = length(v) ÷ 2
    left = Threads.@spawn psum(@view v[1:mid])
    right = psum(@view v[mid+1:end])
    fetch(left) + right
end
```

`OhMyThreads.jl` wraps the common patterns (`tmap`, `treduce`, `tforeach`, chunking, scheduling policies) and is worth using instead of hand-rolling.

Data races

Real, silent, and yours to prevent.

```
# BROKEN: concurrent read-modify-write on a shared scalar
total = 0
Threads.@threads for i in 1:10^6
    total += i          # lost updates
end
```

Fixes, in order of preference:

```
# 1. Per-task partial results, combined at the end. Fastest, no contention.
partials = map(Iterators.partition(1:10^6, 10^4)) do chunk
    Threads.@spawn sum(chunk)
end
total = sum(fetch.(partials))

# 2. Atomics, for counters
counter = Threads.Atomic{Int}(0)
Threads.atomic_add!(counter, 1)
counter[]

# 3. Locks, for anything more complex
const L = ReentrantLock()
Threads.@threads for x in items
    y = compute(x)                # do the work OUTSIDE the lock
    lock(L) do
        push!(shared, y)
    end
end
```

Don't index per-thread buffers by `threadid()`. With dynamic scheduling a task can migrate between threads mid-execution, so `buffers[threadid()]` is not safe. Use one buffer per task (per chunk) instead.

Writing to *distinct* indices of a shared array from different threads is safe. Growing a shared `Vector` with `push!` from multiple threads is not.

Channels

Typed, bounded queues. Closest thing to a mailbox, but they are objects you pass around, not addresses attached to a process.

```
ch = Channel{Int}(32)          # buffered, capacity 32
ch = Channel{Int}(0)           # unbuffered: put! blocks until a take!
put!(ch, 1)
x = take!(ch)
```

```
isready(ch); close(ch)

for x in ch                    # iterates until the channel is closed
    process(x)
end
```

Producer bound to a task (closes automatically when the task ends):

```
producer = Channel{Int}(10) do ch
    for i in 1:100
        put!(ch, i)
    end
end
collect(producer)
```

Worker pool:

```
jobs = Channel{Int}(100)
results = Channel{Int}(100)

for _ in 1:Threads.nthreads()
    Threads.@spawn for j in jobs
        put!(results, expensive(j))
    end
end

foreach(j -> put!(jobs, j), 1:1000)
close(jobs)
```

Taking from a closed, empty channel throws `InvalidStateException`; that's how consumers know to stop, and it's what the `for x in ch` form handles for you.

Distributed

Separate processes, separate heaps, message passing over sockets. For multi-machine work or for escaping the GC on shared-memory workloads.

```
using Distributed
addprocs(4)                    # local workers
addprocs([("host", 4)])        # over SSH
nworkers(); workers()

@everywhere using MyPkg        # load on every worker
@everywhere f(x) = x^2

r = remotecall_fetch(f, 2, 10)  # run f(10) on worker 2
pmap(f, 1:100)                 # parallel map, dynamic load balancing
@distributed (+) for i in 1:1000    # parallel reduction
    f(i)
end

using SharedArrays             # shared memory on one machine
S = SharedArray{Float64}(1000)
```

`Distributed` is a stdlib but no longer the default recommendation for single-machine parallelism — threads are cheaper. Use it for clusters, or when you need process isolation.

Async I/O details

```
sleep(0.1)                # yields
yield()                   # explicitly give up the scheduler
Timer(cb, 1; interval=5)  # periodic callback
@async while true; ...; sleep(60); end     # a background loop
```

```
# blocking a task on an external event
c = Condition()
notify(c, value)
wait(c)

Base.Event()              # one-shot, thread-safe
Semaphore(n); acquire(s); release(s)
```

`ReentrantLock`, `Semaphore`, `Condition`, `Event`, `Threads.Atomic` are the whole synchronisation toolkit.

What you don't get, coming from Elixir

- No supervision trees. If a task dies, nothing restarts it; write your own retry loop.
- No process isolation. A task that corrupts shared state corrupts it for everyone.
- No preemption. A CPU-bound task without `yield` starves its thread's other tasks (though other threads keep running).
- No `receive` with pattern matching. Channels are typed queues; you match on the value yourself.
- No transparent distribution. `Distributed` requires explicit `@everywhere` code loading and explicit placement.
- GC is stop-the-world across all threads, so latency has a floor that the BEAM doesn't have.

What you do get: genuine shared-memory parallelism over large arrays with no copying, which is the trade the language is making.

Debugging concurrency

```
julia -t 4 --check-bounds=yes

@sync                     # surface exceptions you'd otherwise lose
istaskfailed(t) && fetch(t)
```

There is no race detector in the standard toolchain. `ThreadSanitizer` builds exist but are not routine. The practical defence is structure: prefer pure functions returning values over shared mutation, use `@sync` and `fetch` everywhere, and keep the mutable region small enough to reason about.