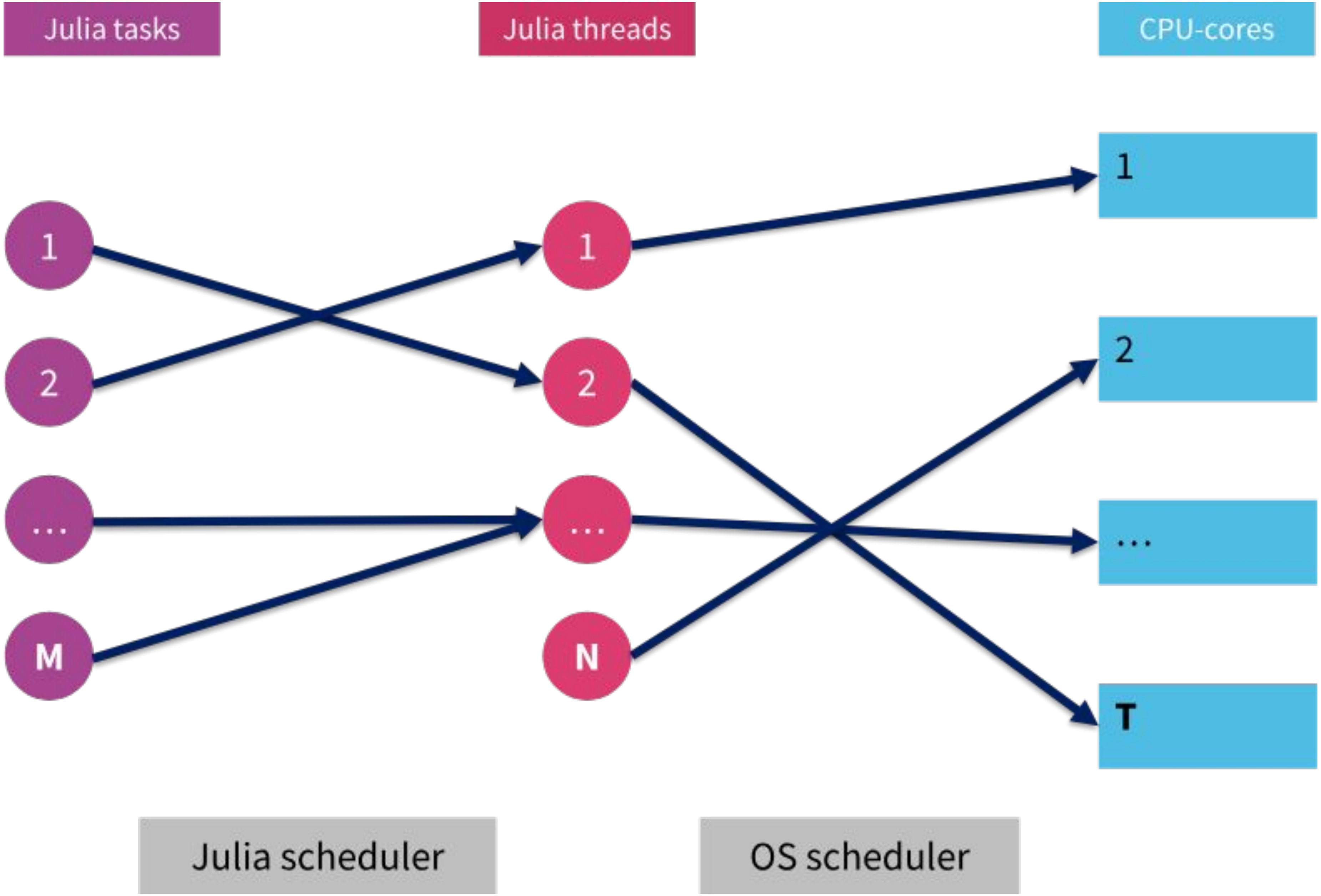
MULTITHREADING IN JULIA

Mosè Giordano (ARC, UCL)

with material by Carsten Bauer

Task-Based Multithreading



Task-Based Multithreading (cont.)

Generally speaking, the user should think about **tasks** and not threads.

- The scheduler is controlling on which thread a task will eventually run.
- It might even dynamically [**migrate tasks**] between threads.

Advantages:

- **High-level abstraction**: one can spawn many tasks (>> number of threads)
- **nestability** / **composability** (especially important for libraries)

Disadvantages:

- Dynamic scheduling **overhead** ~1-10 μs (not ideal for *very* quick tasks)
- **Uncertain** and potentially suboptimal task → thread assignment
  - Can get in the way when performance engineering because
    - **Scheduler** has limited information (e.g. about the system topology, NUMA-unaware)
    - **Profiling tools** often don't know anything about tasks but monitor threads (or even CPU-cores) instead (e.g. LIKWID)

Multithreading in Julia

- Currently the **number of Julia threads** must be set at startup and can't be changed during runtime. This limitation *may* be lifted in the future.
- In addition to task-based parallelism, you can use **multithreaded `for` loops**, choosing between **different schedulers**:
  - **dynamic** (the default): composable and easy to use
  - **greedy**: composable and good for unbalanced workloads, slightly more overhead
  - **static**: not composable, lower overhead, task migration is disabled

Spawning Parallel Tasks

```
julia> using Base.Threads

julia> @time t = @spawn begin # `@spawn` returns right away
           sleep(3)
           3+3
       end
  0.000051 seconds (35 allocations: 2.320 KiB)
Task (runnable, started) @0x00007f69a3e01460

julia> @time fetch(t) # `fetch` waits for the task to finish
  3.004165 seconds (1 allocation: 16 bytes)
6
```

Example: multithreaded `map`

```
julia> using LinearAlgebra, BenchmarkTools

julia> BLAS.set_num_threads(1) # Fix number of BLAS threads

julia> function tmap(fn, itr)
           # for each i ∈ itr, spawn a task to compute fn(i)
           tasks = map(i -> @spawn(fn(i)), itr)
           # fetch and return all the results
           return fetch.(tasks)
       end;

julia> M = [rand(200,200) for i in 1:8];
```

Example: multithreaded `map` (cont.)

```
julia> tmap(svdvals, M)
8-element Vector{Vector{Float64}}:
[...]

julia> nthreads()
4

julia> @btime map(svdvals, $M) samples=10 evals=3;
 13.024 ms (106 allocations: 3.37 MiB)

julia> @btime tmap(svdvals, $M) samples=10 evals=3;
 3.754 ms (155 allocations: 3.38 MiB)
```

Example: multithreaded `for` loop

```
julia> using ChunkSplitters, Base.Threads, BenchmarkTools

julia> function sum_threads(fn, data; nchunks=nthreads())
           psums = zeros(eltype(data), nchunks)
           @threads for (c, elements) in enumerate(chunks(data; n=nchunks))
               psums[c] = sum(fn, elements)
           end
           return sum(psums)
       end;
```

Example: multithreaded `for` loop (cont.)

```
julia> v = randn(10_000_000);

julia> @btime sum(sin, $v)
  80.191 ms (0 allocations: 0 bytes)
-670.7508590568106

julia> @btime sum_threads(sin, $v)
  21.854 ms (24 allocations: 2.31 KiB)
-670.7508590568106
```

Example: multithreaded `for` loop (cont.)

```
julia> function sum_map_spawn(fn, data; nchunks=nthreads())
           ts = map(chunks(data, n=nchunks)) do elements
               @spawn sum(fn, elements)
           end
           return sum(fetch.(ts))
       end;

julia> @btime sum_map_spawn(sin, $v)
 21.783 ms (48 allocations: 3.00 KiB)
-670.7508590568106
```

Example: multithreaded `for` loop (cont.)

```
julia> using OhMyThreads: @tasks

julia> function sum_tasks(fn, data; nchunks=nthreads())
           psums = zeros(eltype(data), nchunks)
           @tasks for (c, elements) in enumerate(chunks(data; n=nchunks))
               psums[c] = sum(fn, elements)
           end
           return sum(psums)
       end;

julia> @btime sum_tasks(sin, $v)
 21.967 ms (32 allocations: 2.59 KiB)
-670.7508590568106
```

Useful Tips

Julia-specific:

- **Reduce memory (heap) allocations** in hot loops, especially if multi-threaded: stop-the-world garbage collector stops all threads
- Apart from few exceptions (e.g. FFTW), most external libraries **don't compose** with Julia parallelism: may need to control their threads

Non Julia-specific:

- Be careful **not to oversubscribe** the system, especially when combined with distributed computing
- **Thread pinning** prevents Julia threads from jumping between cores

Additional tools:

- OhMyThreads.jl: user-friendly tools for task-based multithreading
- ChunkSplitters.jl: easy chunking of multithreaded workloads
- ThreadsX.jl: parallelized Base functions
- ThreadPinning.jl: fine-grained control to pin Julia threads