High Performance Computing in Julia

from the ground up.

Multithreading

Multithreading

- Each process can spin up multiple **threads** to enable concurrent processing
- Each thread has access to all the shared memory in a process
- Very **cheap** to spin up new threads (as opposed to starting a new process)
- If there are multiple cores available, each thread can be executed on a different core in **parallel**
- Shared memory introduces new challenges, namely **race conditions** which need to be addressed by **atomics**, **mutexes**, **semaphores** or **algorithm re-design**.

Race Conditions

- If two threads are trying to write & read from the **same block** of memory at the same time.
- Race conditions usually do not cause the program to crash, but often just produce the **wrong** results
- Typical examples:
  - Mutating an array or variable (i.e. a counter)
  - Appending to an array
  - Random number generation
- Functions/Operations that avoid race conditions are known as **thread-safe**

Mitigating Race Conditions

Atomics

Atomics

- Atomic operations are designed to be indivisible so that you can guarantee that the operations will happen sequentially

Atomics

Race Condition

```
using Base.Threads
function my_sum(numbers::Vector{Int})
    s = 0
    @threads for n in numbers
        s += n
    end
    return s
end
```

Thread-Safe (with Atomics)

```
function my_sum(numbers::Vector{Int})
    s = Atomic{Int}(0)
    @threads for n in numbers
        atomic_add!(s, n)
    end
    return s[]
end
```

Atomics

Advantages

- Fixes the race conditions
- Guarantees thread-safety if used correctly
- Can be used as part of the solution

Disadvantages

- Atomic operations are **much slower** than non-atomic counterparts
- Causes threads to **sleep** while waiting to write
- Can cause a higher slowdown with more threads
- Usually means that algorithm is badly designed

Atomics

Benchmarks

```
julia> @btime sum($numbers)
  46.328 ms (0 allocations: 0 bytes)
julia> @btime my_sum($numbers)
  1.144 s (26 allocations: 2.64 KiB)
julia> @btime my_sum_chunked($numbers)
  14.982 ms (27 allocations: 2.86 KiB)
```

25x

Performant Version (Chunked)

```julia
import Base.Iterators: partition
function my_sum_chunked(numbers::Vector{Int})
    s = Atomic{Int}(0)
    block_size = cld(length(numbers), nthreads())
    iter = collect(partition(numbers, block_size))
    @threads for nums in iter
        atomic_add!(s, sum(nums))
    end
    return s[]
end
```

Atomics

Benchmarks

```
julia> @btime sum($numbers)
  46.328 ms (0 allocations: 0 bytes)
julia> @btime my_sum($numbers)
  1.144 s (26 allocations: 2.64 KiB)
julia> @btime my_sum_chunked($numbers)
  14.982 ms (27 allocations: 2.86 KiB)
```

3x

Performant Version (Chunked)

```
import Base.Iterators: partition
function my_sum_chunked(numbers::Vector{Int})
    s = Atomic{Int}(0)
    block_size = cld(length(numbers), nthreads())
    iter = collect(partition(numbers, block_size))
    @threads for nums in iter
        atomic_add!(s, sum(nums))
    end
    return s[]
end
```

Atomics

Benchmarks

```
julia> @btime sum($numbers)
  46.328 ms (0 allocations: 0 bytes)
julia> @btime my_sum($numbers)
  1.144 s (26 allocations: 2.64 KiB)
julia> @btime my_sum_chunked($numbers)
  14.982 ms (27 allocations: 2.86 KiB)
```

Performant Version (Chunked)

```julia
import Base.Iterators: partition
function my_sum_chunked(numbers::Vector{Int})
    s = Atomic{Int}(0)
    block_size = cld(length(numbers), nthreads())
    iter = collect(partition(numbers, block_size))
    @threads for nums in iter
        atomic_add!(s, sum(nums))
    end
    return s[]
end
```

Mitigating Race Conditions

Mutexes and Semaphores

Mutexes vs Semaphores

Mutexes

- A mutex provides **mutual exclusion**, which means that only one worker can access a resource at any one time
- Can be implemented as a "**lock**" where it can be either unlocked or locked.

Semaphores

- A semaphore generalises the **mutex**
- Instead of one resource, this provides a "**pool**" of resources
- E.g. a pool of memory buffers
- Resources can be added back into the pool when usage is not required

Mutexes in Julia

- We use the `ReentrantLock()` to act as a **mutex**

```
function my_sum_mutex(numbers::Vector{Int})
    s = 0
    lk = ReentrantLock()
    @threads for n in numbers
        lock(lk) do
            s += n
        end
    end
    return s
end
```

Mutexes in Julia

- We use the `ReentrantLock()` to act as a **mutex**

```
function my_sum_mutex(numbers::Vector{Int})
    s = 0
    lk = ReentrantLock()
    @threads for n in numbers
        lock(lk) do
            s += n
        end
    end
    return s
end
```

The lock is a standalone variable

Mutexes in Julia

- We use the `ReentrantLock()` to act as a **mutex**

```
function my_sum_mutex(numbers::Vector{Int})
    s = 0
    lk = ReentrantLock()
    @threads for n in numbers
        lock(lk) do
            s += n
        end
    end
    return s
end
```

- The lock function attempts to **acquire** the mutex
- When acquired it will execute the code in the “do” block
- When finished executing, the thread **relinquishes** the lock so it can be acquired by another thread
- Threads will automatically **wait** to acquire a lock

Semaphores in Julia

- We can use a `Channel()` to act as a semaphore, which can be buffered or unbuffered.

```
function my_sum_channel(numbers::Vector{Int})
    num_buffers = 4
    pool = Channel{Int}(num_buffers)
    for i in 1:num_buffers
        put!(pool, 0)
    end
    @threads for n in numbers
        s = take!(pool)
        s += n
        put!(pool, s)
    end
    s = 0
    for i in 1:num_buffers
        s += take!(pool)
    end
    close(pool)
    return s
end
```

Semaphores in Julia

- We can use a `Channel()` to act as a semaphore, which can be buffered or unbuffered.

```
function my_sum_channel(numbers::Vector{Int})
    num_buffers = 4
    pool = Channel{Int}(num_buffers)
    for i in 1:num_buffers
        put!(pool, 0)
    end
    @threads for n in numbers
        s = take!(pool)
        s += n
        put!(pool, s)
    end
    s = 0
    for i in 1:num_buffers
        s += take!(pool)
    end
    close(pool)
    return s
end
```

Create a pool of resources to use with a maximum capacity of 4

Semaphores in Julia

- We can use a `Channel()` to act as a semaphore, which can be buffered or unbuffered.

```julia
function my_sum_channel(numbers::Vector{Int})
    num_buffers = 4
    pool = Channel{Int}(num_buffers)
    for i in 1:num_buffers
        put!(pool, 0)
    end
    @threads for n in numbers
        s = take!(pool)
        s += n
        put!(pool, s)
    end
    s = 0
    for i in 1:num_buffers
        s += take!(pool)
    end
    close(pool)
    return s
end
```

Acquire one of the resources from the pool

Semaphores in Julia

- We can use a `Channel()` to act as a semaphore, which can be buffered or unbuffered.

```
function my_sum_channel(numbers::Vector{Int})
    num_buffers = 4
    pool = Channel{Int}(num_buffers)
    for i in 1:num_buffers
        put!(pool, 0)
    end
    @threads for n in numbers
        s = take!(pool)
        s += n
        put!(pool, s)
    end
    s = 0
    for i in 1:num_buffers
        s += take!(pool)
    end
    close(pool)
    return s
end
```

Put it back in the pool when finished

Semaphores in Julia

- We can use a `Channel()` to act as a semaphore, which can be buffered or unbuffered.

```
function my_sum_channel(numbers::Vector{Int})
    num_buffers = 4
    pool = Channel{Int}(num_buffers)
    for i in 1:num_buffers
        put!(pool, 0)
    end
    @threads for n in numbers
        s = take!(pool)
        s += n
        put!(pool, s)
    end
    s = 0
    for i in 1:num_buffers
        s += take!(pool)
    end
    close(pool)
    return s
end
```

Combine the resources together and close the pool

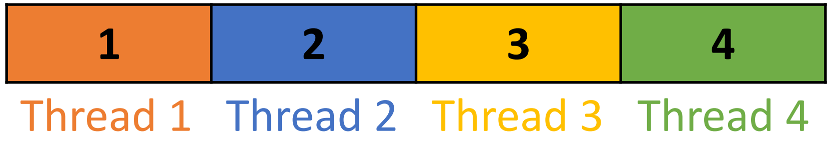
Mitigating Race Conditions

Separate Memory Per Thread

Separate Memory Per Thread

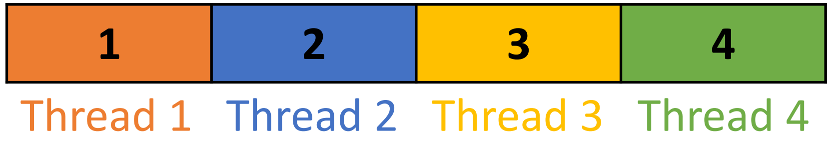
```
function est_pi_mc_threaded(n)
    n_cs = zeros(typeof(n), Threads.nthreads())
    Threads.@threads for _ in 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        if r2 <= 1
            n_cs[Threads.threadid()] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

Separate Memory Per Thread



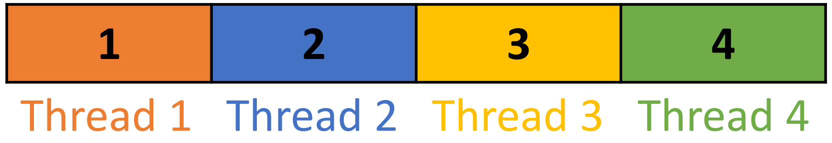

```julia
function est_pi_mc_threaded(n)
    n_cs = zeros(typeof(n), Threads.nthreads())
    Threads.@threads for _ in 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        if r2 <= 1
            n_cs[Threads.threadid()] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

Separate Memory Per Thread



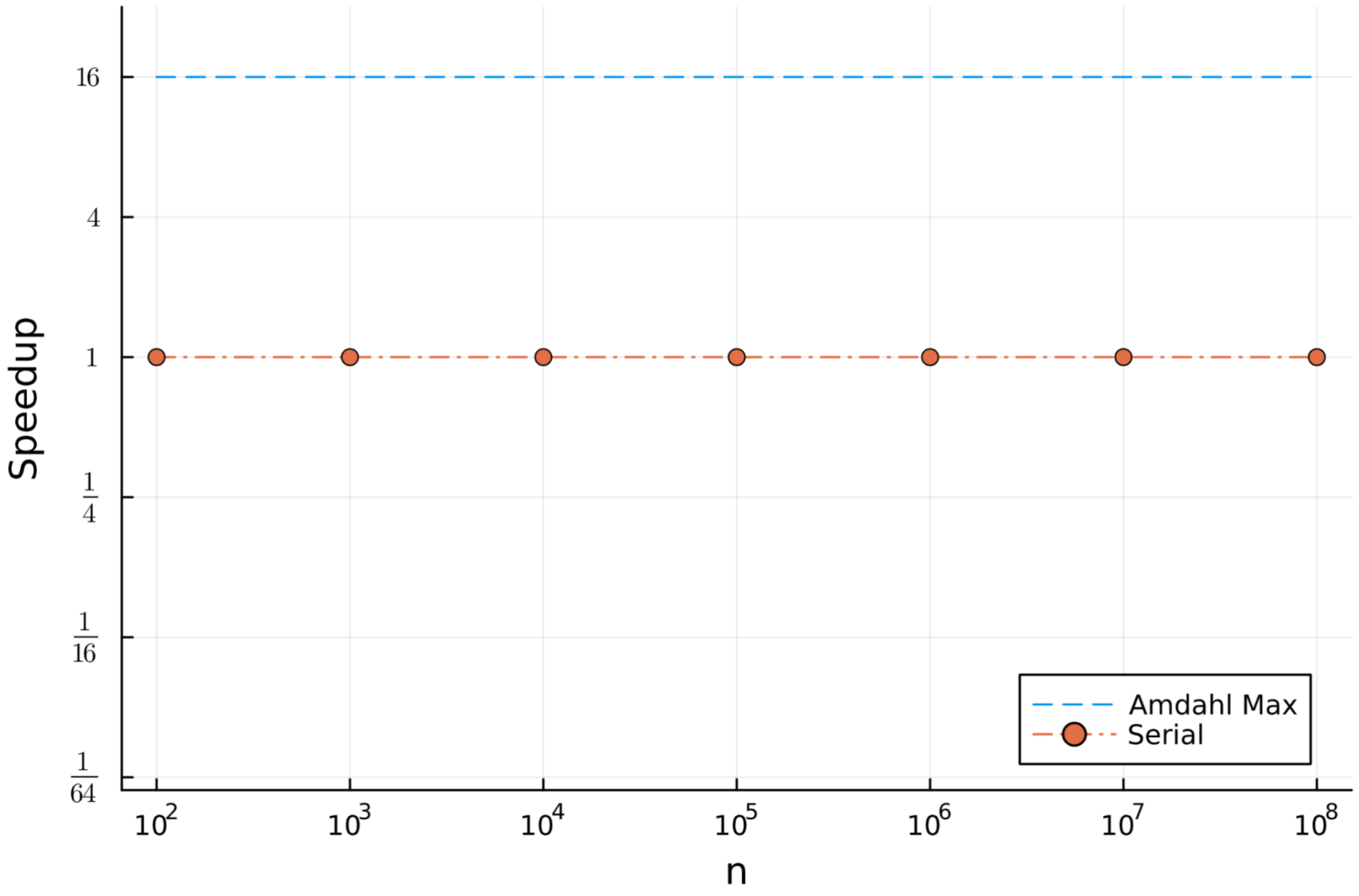

```julia
function est_pi_mc_threaded(n)
    n_cs = zeros(typeof(n), Threads.nthreads())
    Threads.@threads for _ in 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        if r2 <= 1
            n_cs[Threads.threadid()] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

Separate Memory Per Thread

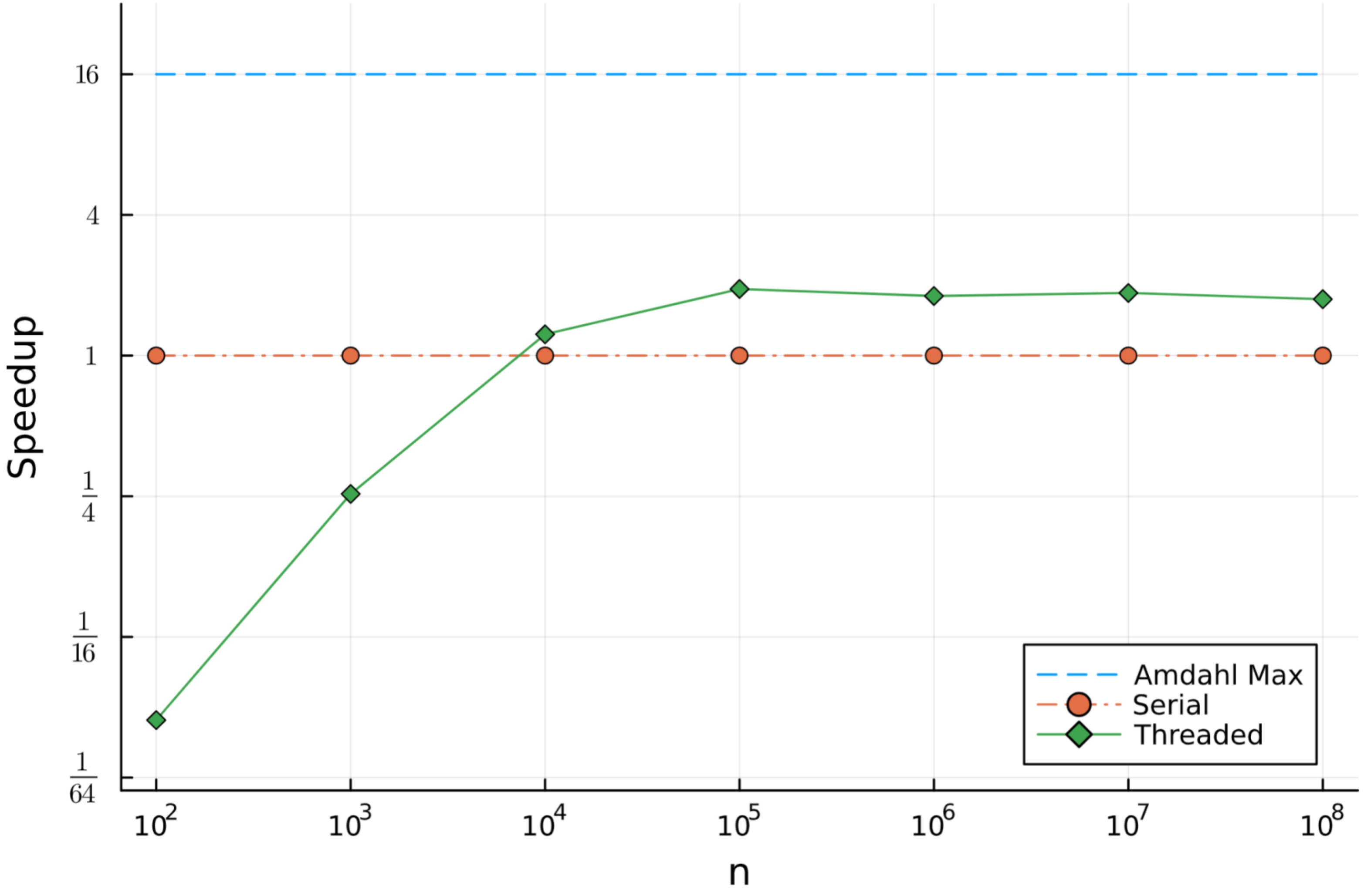



```
function est_pi_mc_threaded(n)
    n_cs = zeros(typeof(n), Threads.nthreads())
    Threads.@threads for _ in 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        if r2 <= 1
            n_cs[Threads.threadid()] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

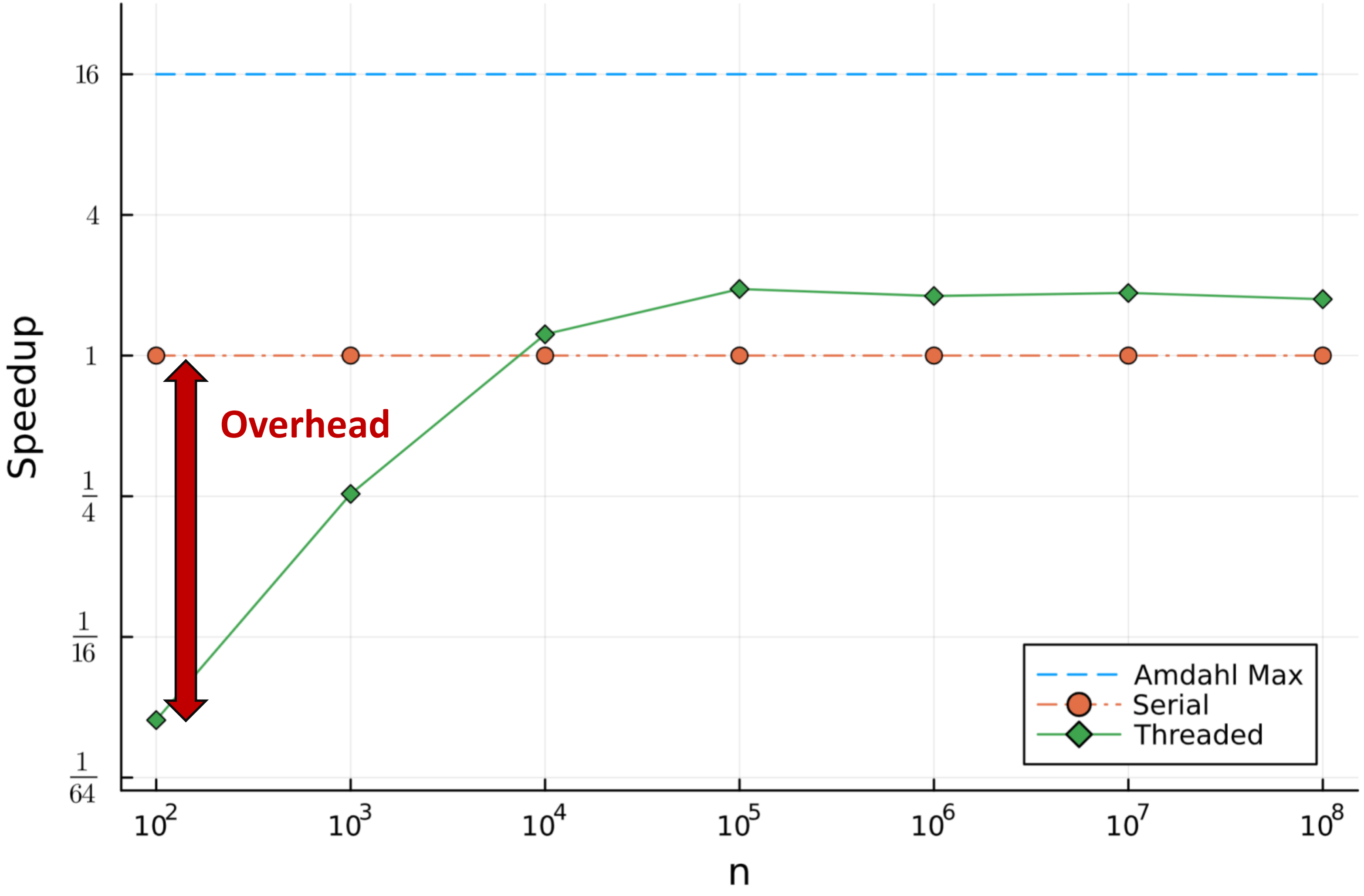
Threaded Performance (separate memory)



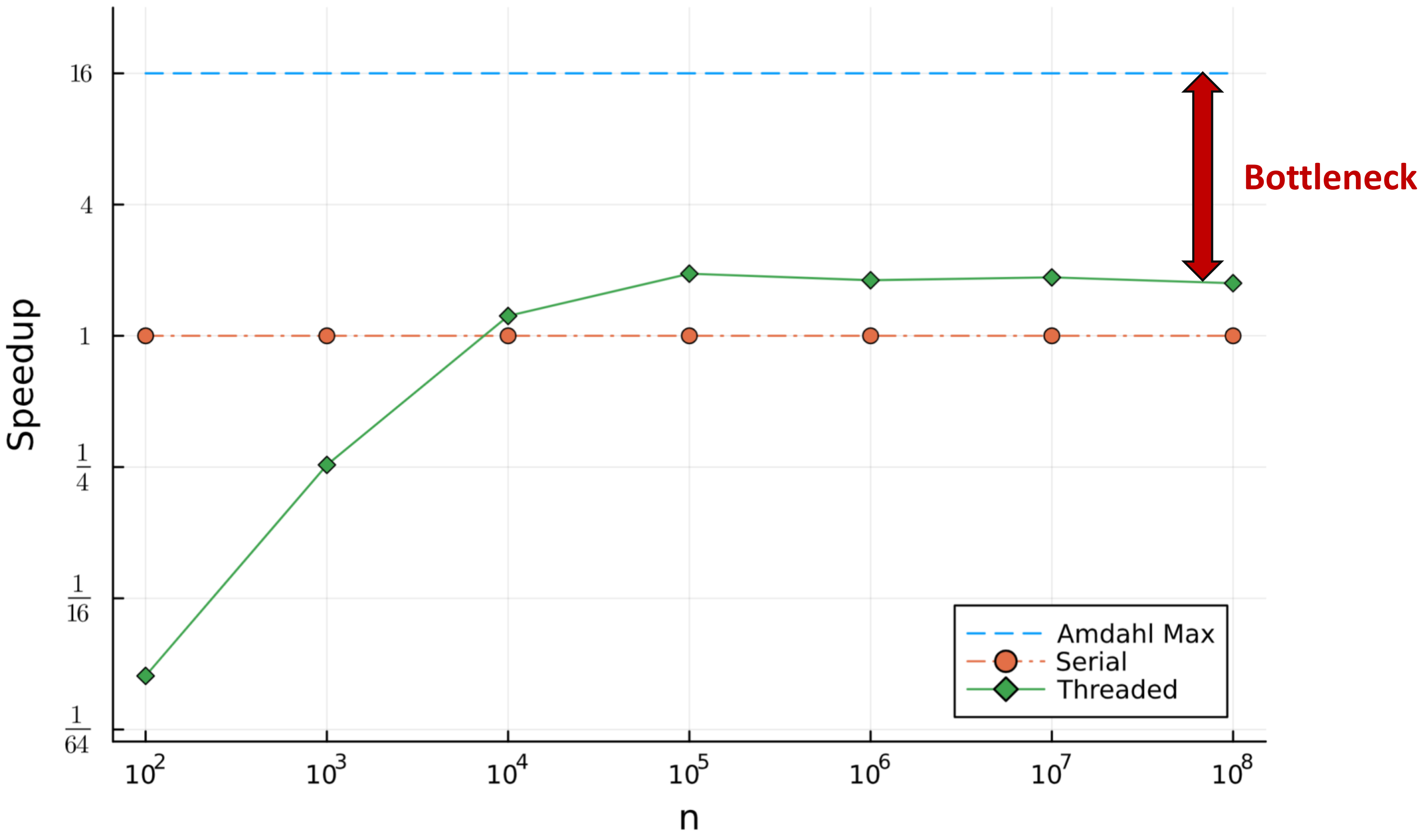
Threaded Performance (separate memory)



Threaded Performance (separate memory)



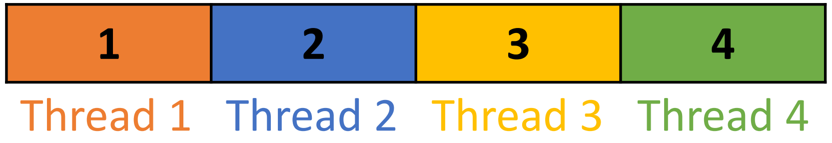
Threaded Performance (separate memory)



False Sharing

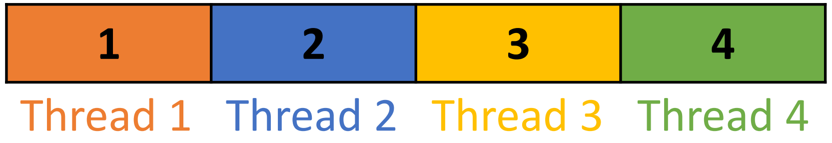
- False sharing is a performance degrading bug, which can occur in shared-memory multithreading code
- The CPU cache collects **contiguous chunks** of memory called **cache lines**
- As elements of an array are stored contiguously (i.e. next to each other), adjacent elements are usually sharing a **cache line**
- Each CPU core has its own L1 cache, which stores the cache line
- If **one** CPU core modifies the cache line, it is **invalidated** across **all** CPU caches
- This will force a **reload** of the cache from memory **despite** it not being logically required

False Sharing Experiment




```
function est_pi_mc_threaded(n)
    n_cs = zeros(typeof(n), Threads.nthreads())
    Threads.@threads for _ in 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        if r2 <= 1
            n_cs[Threads.threadid()] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

False Sharing Experiment

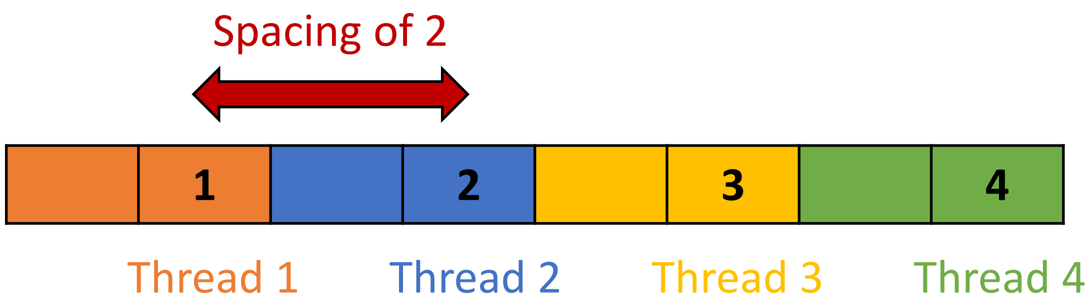



```julia
function est_pi_mc_threaded_spaced(n, spacing=1)
    n_cs = zeros(typeof(n), Threads.nthreads()*spacing)
    Threads.@threads for _ in 1:n
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        r2 = x*x+y*y
        if r2 <= 1
            n_cs[Threads.threadid()*spacing] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

False Sharing Experiment

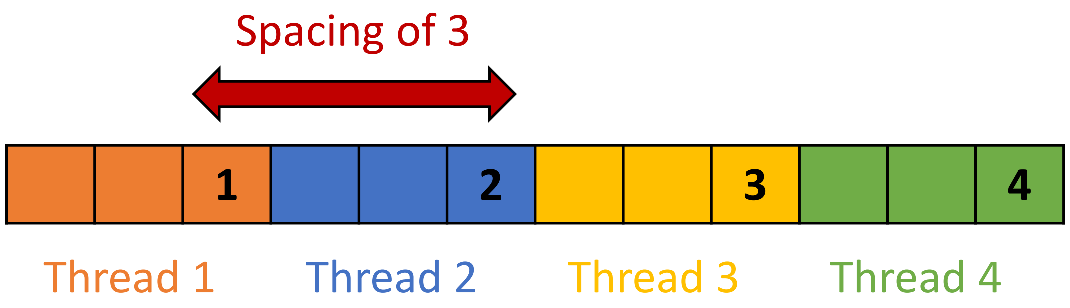

```
function est_pi_mc_threaded_spaced(n, spacing=1)
    n_cs = zeros(typeof(n), Threads.nthreads()*spacing)
    Threads.@threads for _ in 1:n
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        r2 = x*x+y*y
        if r2 <= 1
            n_cs[Threads.threadid()*spacing] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

False Sharing Experiment



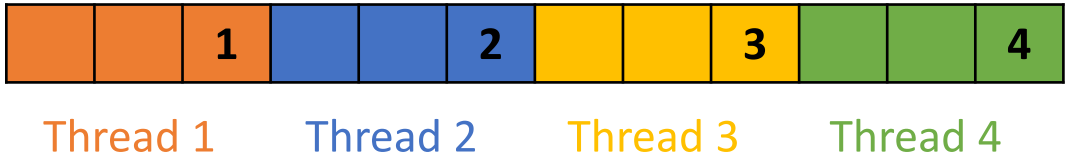

```julia
function est_pi_mc_threaded_spaced(n, spacing=1)
    n_cs = zeros(typeof(n), Threads.nthreads()*spacing)
    Threads.@threads for _ in 1:n
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        r2 = x*x+y*y
        if r2 <= 1
            n_cs[Threads.threadid()*spacing] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

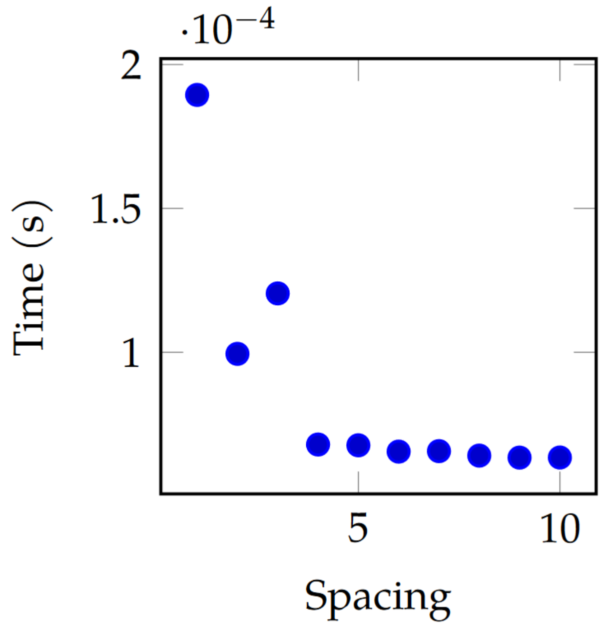
False Sharing Experiment




```julia
function est_pi_mc_threaded_spaced(n, spacing=1)
    n_cs = zeros(typeof(n), Threads.nthreads()*spacing)
    Threads.@threads for _ in 1:n
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        r2 = x*x+y*y
        if r2 <= 1
            n_cs[Threads.threadid()*spacing] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

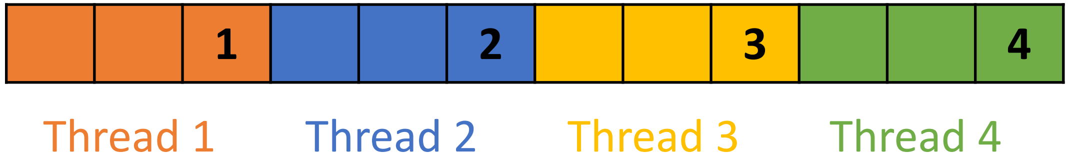
False Sharing Experiment

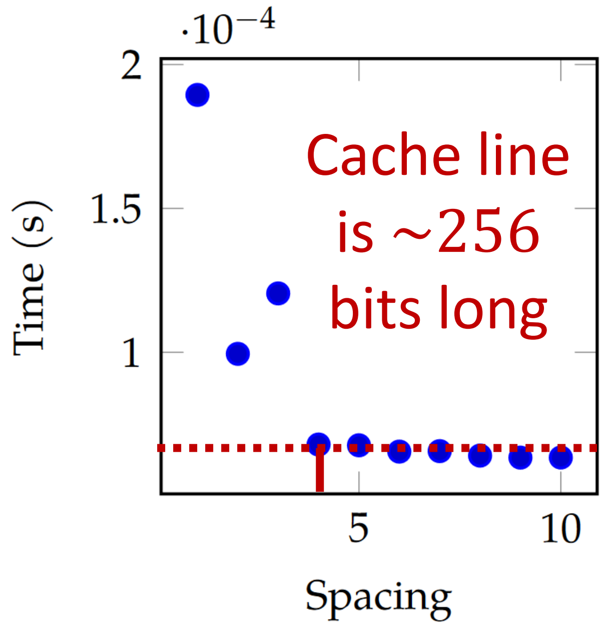





```julia
function est_pi_mc_threaded_spaced(n, spacing=1)
    n_cs = zeros(typeof(n), Threads.nthreads()*spacing)
    Threads.@threads for _ in 1:n
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        r2 = x*x+y*y
        if r2 <= 1
            n_cs[Threads.threadid()*spacing] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

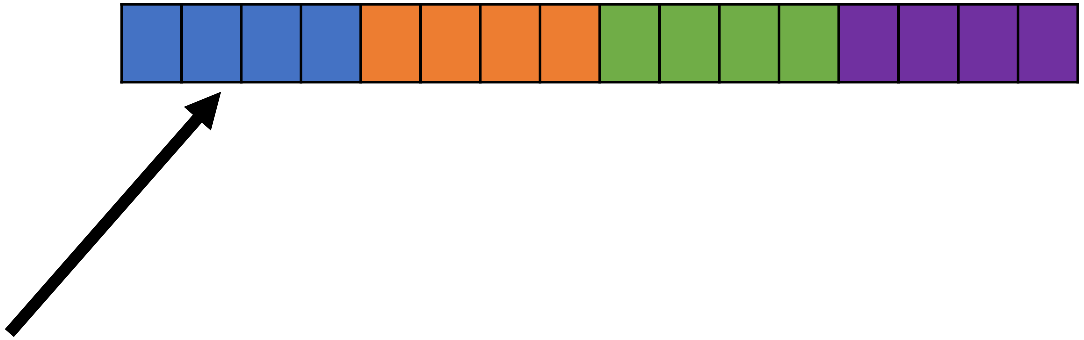
False Sharing Experiment





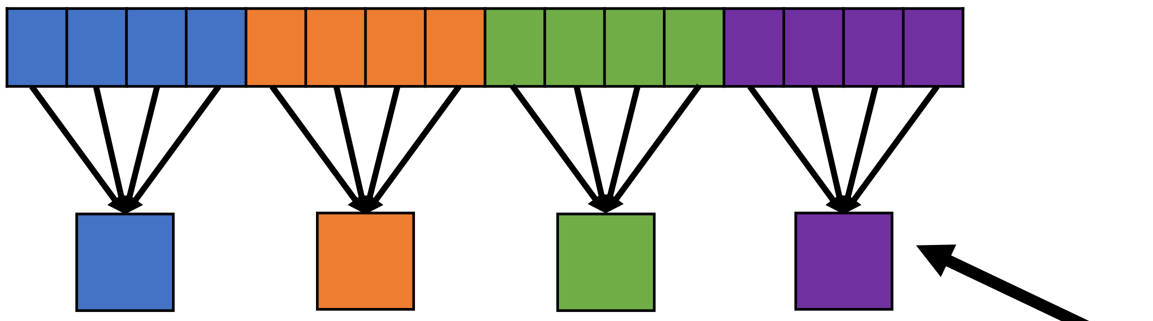

```julia
function est_pi_mc_threaded_spaced(n, spacing=1)
    n_cs = zeros(typeof(n), Threads.nthreads()*spacing)
    Threads.@threads for _ in 1:n
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        r2 = x*x+y*y
        if r2 <= 1
            n_cs[Threads.threadid()*spacing] += 1
        end
    end
    n_c = sum(n_cs)
    return 4 * n_c / n
end
```

Monte-Carlo Simulation (Estimating π)



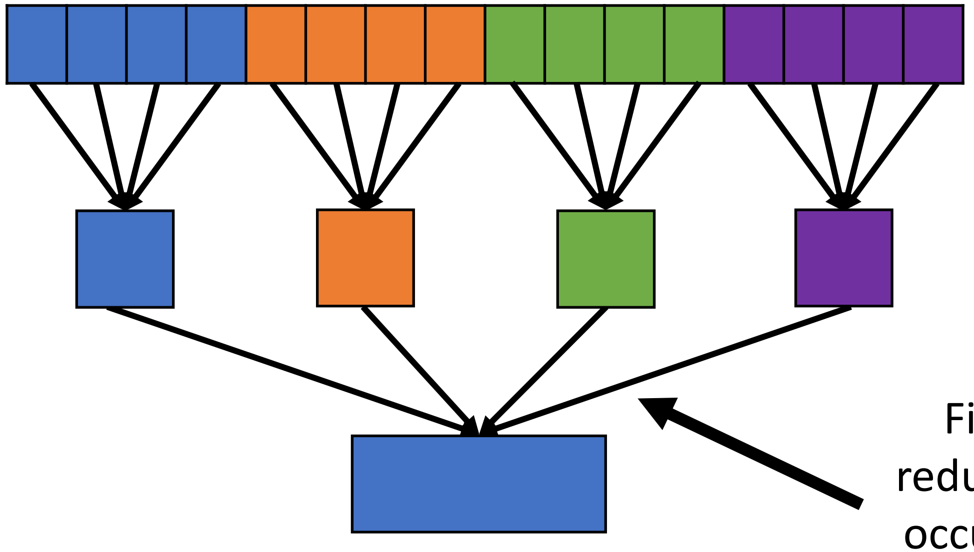
Partition workload
into chunks for
each thread

Monte-Carlo Simulation (Estimating π)



Each chunk
reduces its
own result

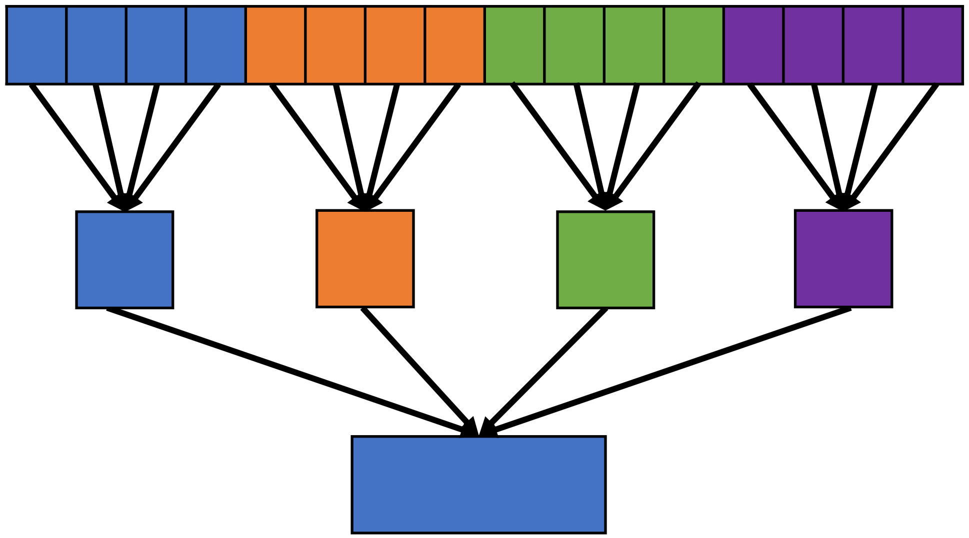
Monte-Carlo Simulation (Estimating π)



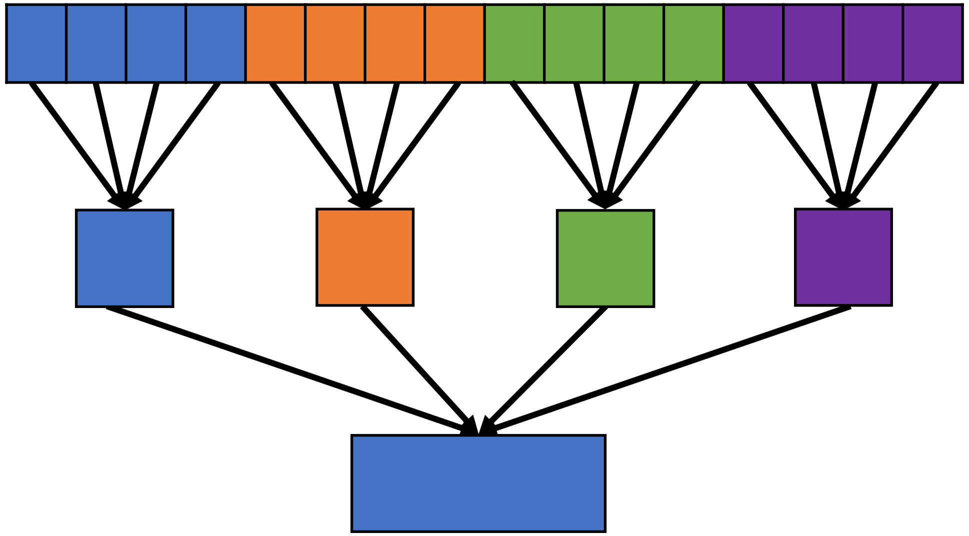
Final
reduction
occurs in
constant time

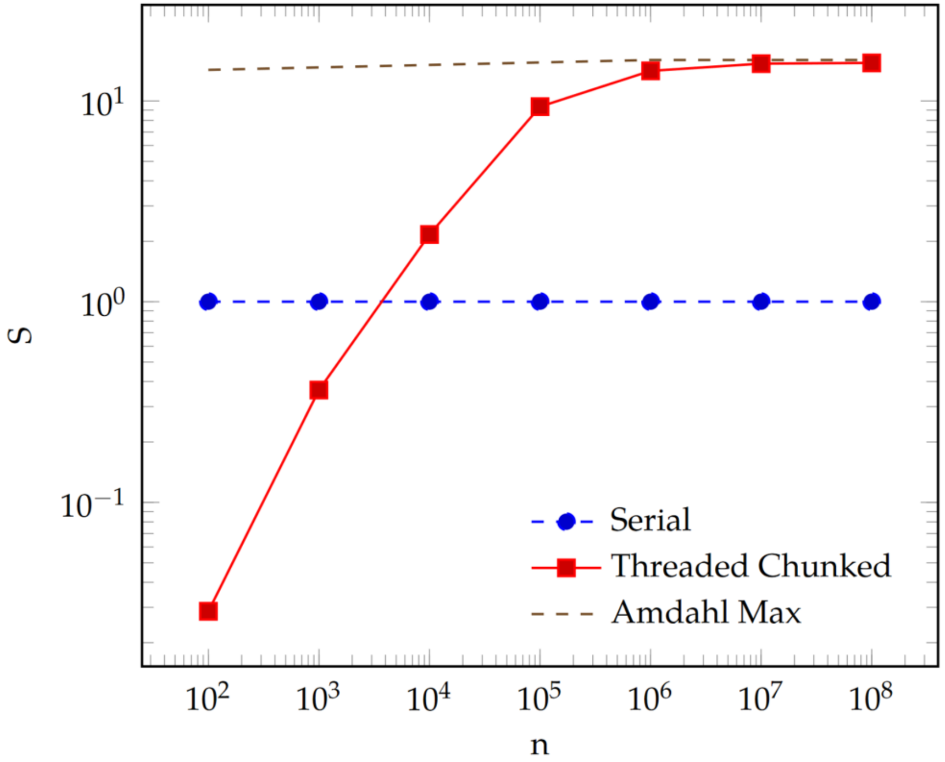
Monte-Carlo Simulation (Estimating π)

```
function is_dart_hit()
    x = rand() * 2 - 1
    y = rand() * 2 - 1
    return (x^2 + y^2 <= 1)
end
function est_pi_mc_threaded_chunked(n)
    n_total = Atomic{Int}()
    block_size = cld(n, nthreads())
    @threads for t in 1:nthreads()
        n_c = mapreduce(x->is_dart_hit(), +, 1:block_size)
        atomic_add!(n_total, n_c)
    end
    return 4 * n_total[] / n
end
```

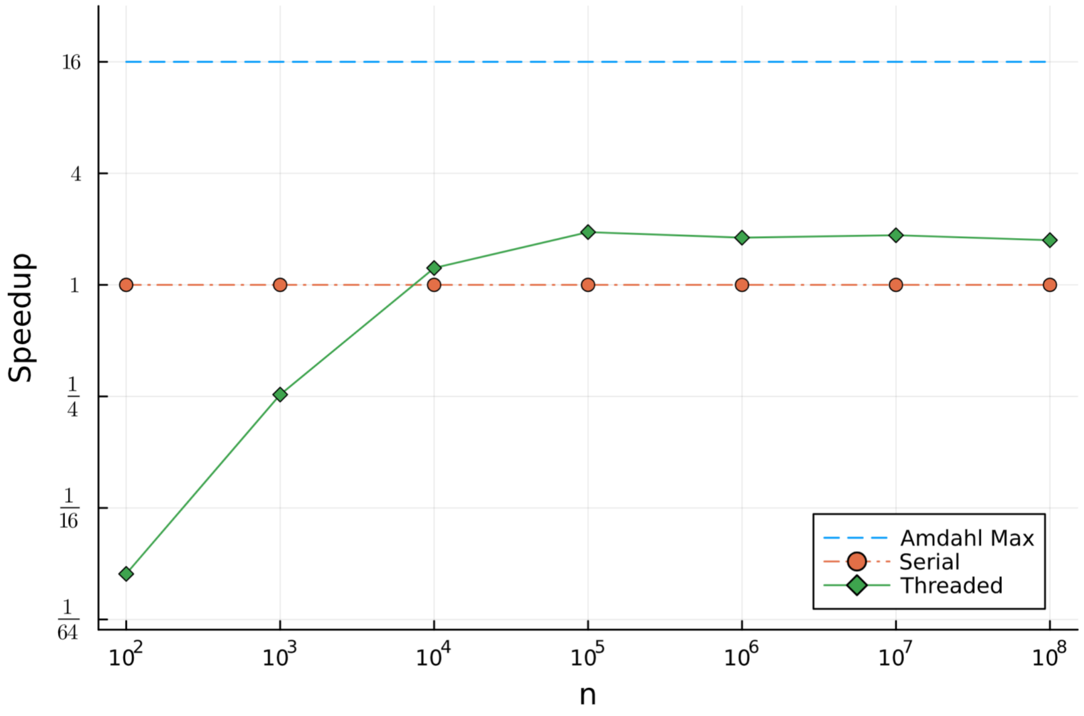


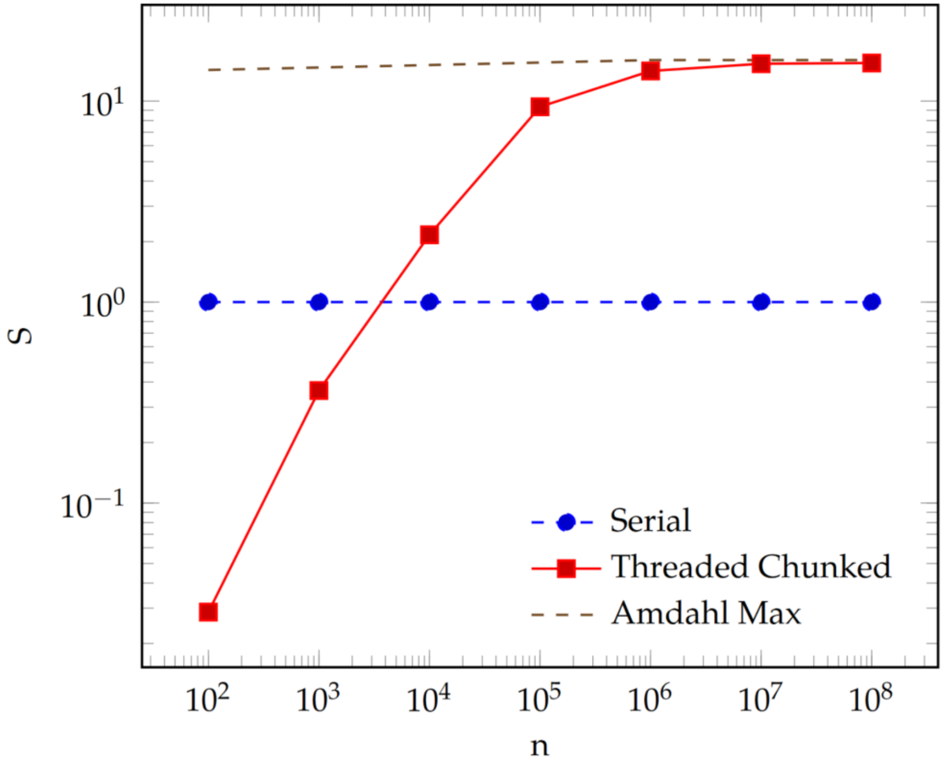
Monte-Carlo Simulation (Estimating π)





Monte-Carlo Simulation (Estimating π)





Workshop

Assignment Link:

https://classroom.github.com/a/HqKUZUwc

Task:

Q1) Fix a race condition

Q2) Create a DAG for the dependices of a calculation and parallelise it with "**Threads.@spawn**" and "**fetch**"

Q3) Parallelise the N-body force calculation