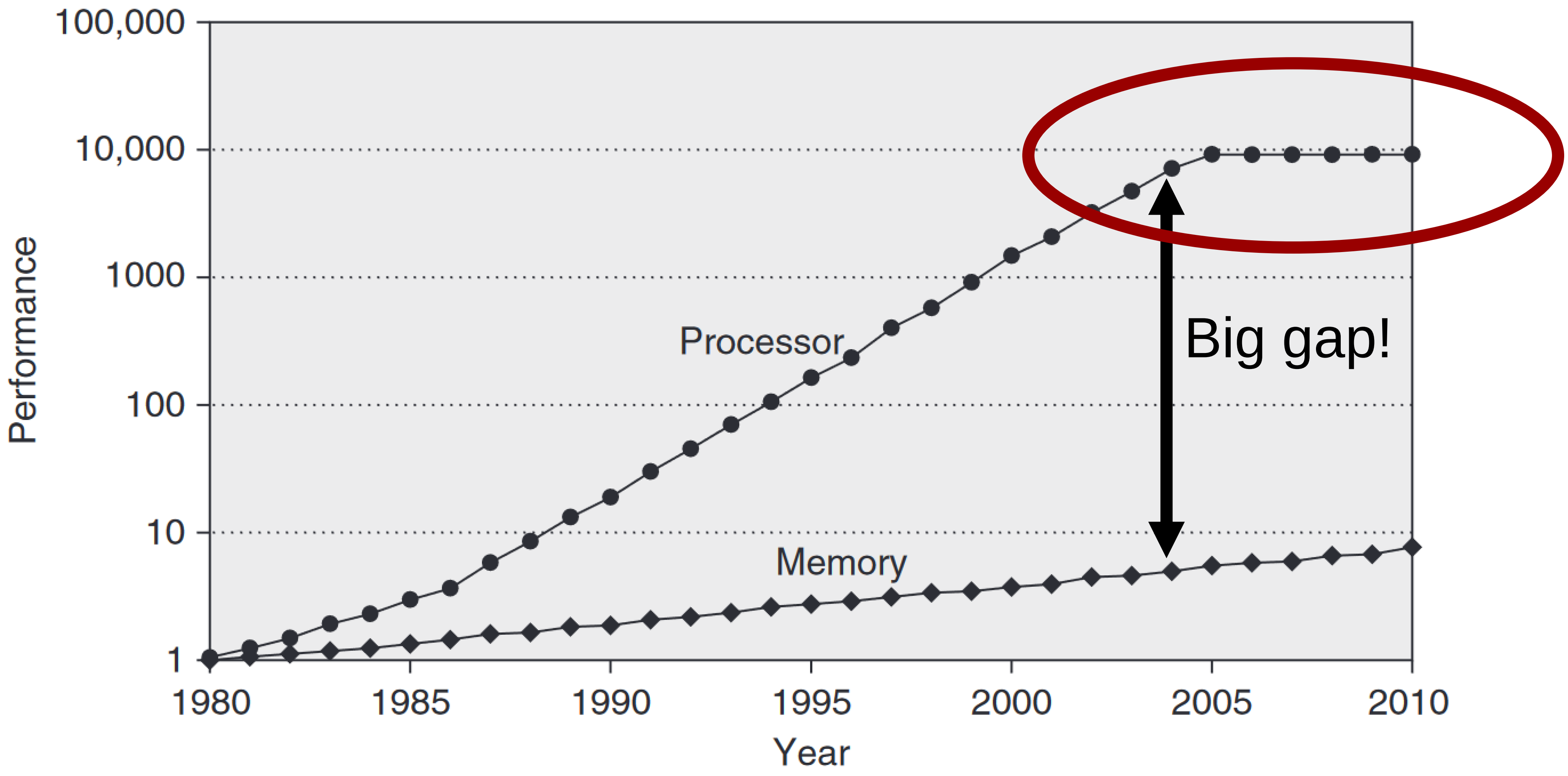
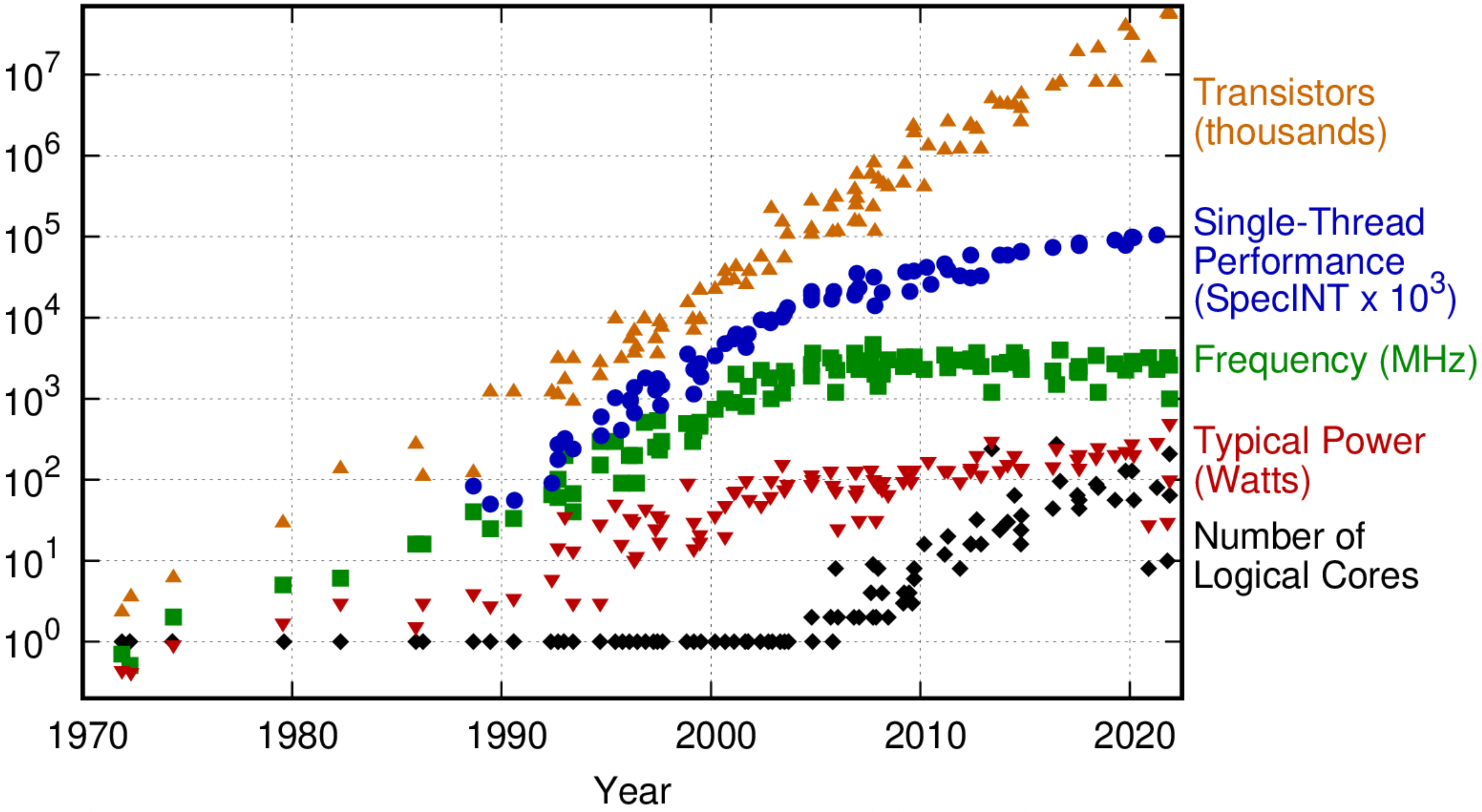
Week 5 – Parallelism Part 1

02613 Python and High-Performance Computing



Hennesy & Patterson, ”Computer Architecture: A Quantitative Approach”, 5th Ed.

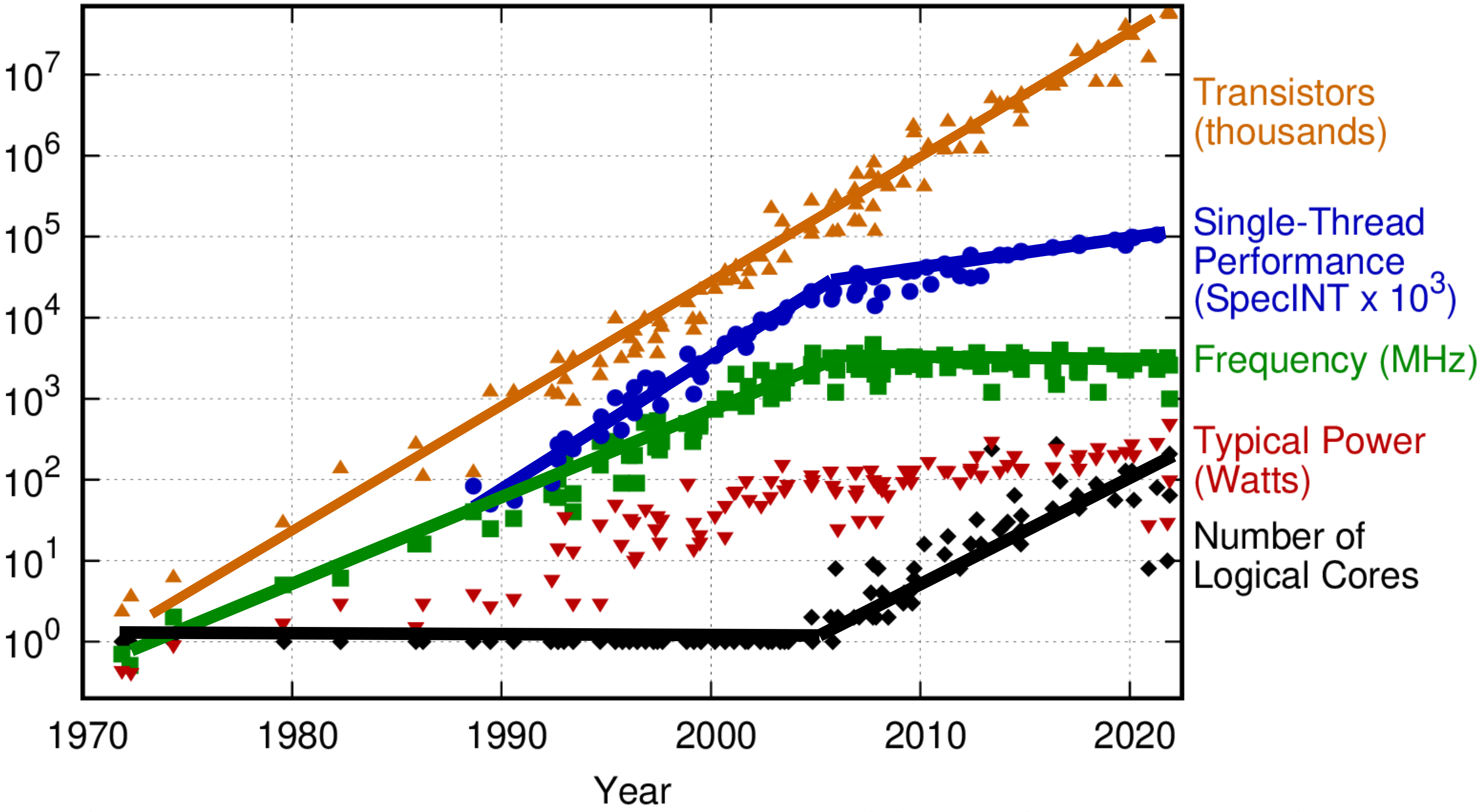
50 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

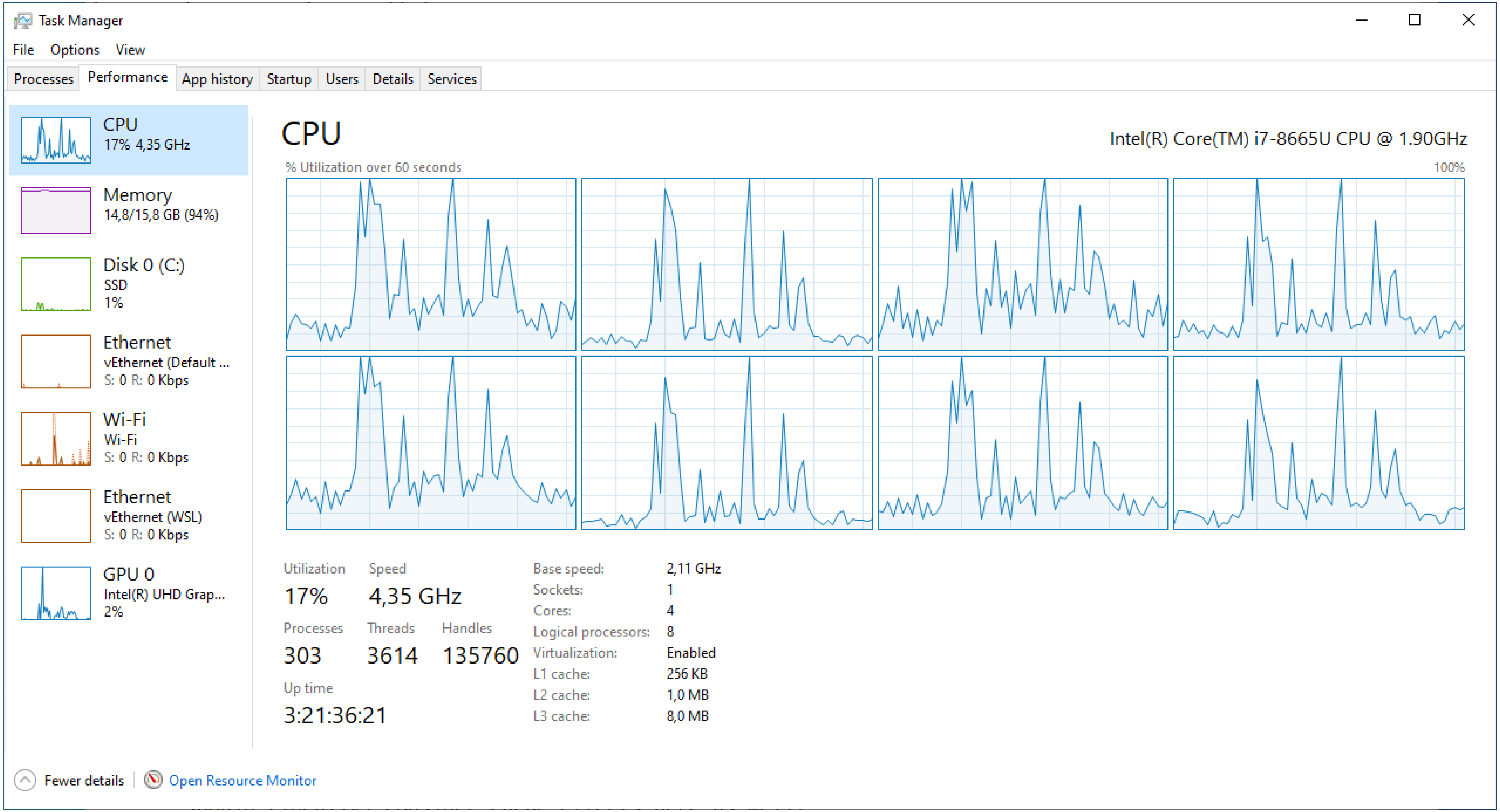
50 Years of Microprocessor Trend Data

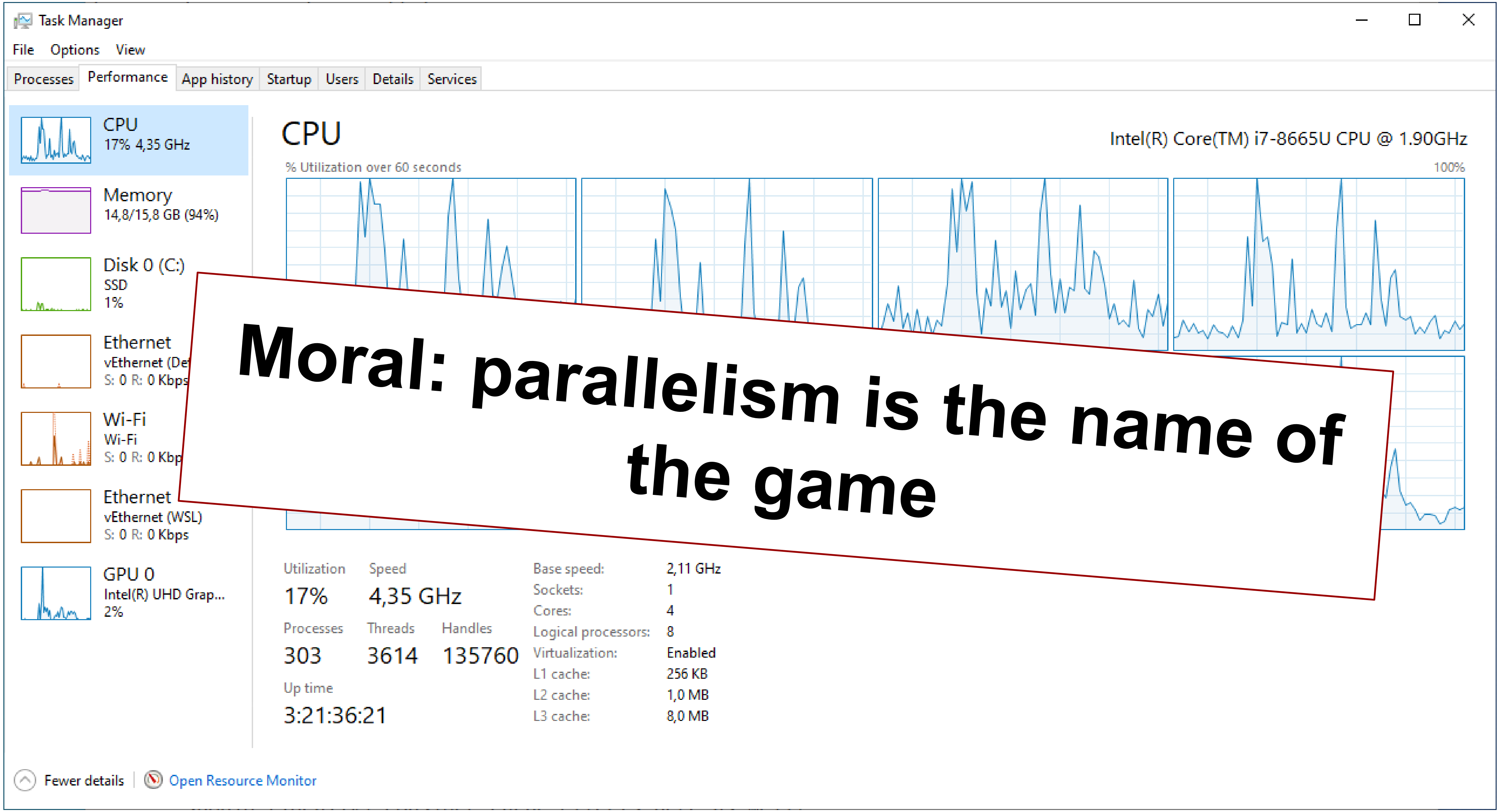


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

```
1  [                0.0%]   13 [|               0.7%]   25 [                0.0%]   37 [|               0.7%]
2  [                0.0%]   14 [                0.0%]   26 [                0.0%]   38 [                0.0%]
3  [                0.0%]   15 [                0.0%]   27 [                0.0%]   39 [                0.0%]
4  [                0.0%]   16 [                0.0%]   28 [                0.0%]   40 [|               0.7%]
5  [                0.0%]   17 [                0.0%]   29 [                0.0%]   41 [                0.0%]
6  [                0.0%]   18 [|               0.7%]   30 [                0.0%]   42 [                0.0%]
7  [                0.0%]   19 [||              2.0%]   31 [                0.0%]   43 [                0.0%]
8  [                0.0%]   20 [                0.0%]   32 [                0.0%]   44 [                0.0%]
9  [                0.0%]   21 [                0.0%]   33 [                0.0%]   45 [|               0.7%]
10 [                0.0%]   22 [                0.0%]   34 [                0.0%]   46 [                0.0%]
11 [                0.0%]   23 [                0.0%]   35 [                0.0%]   47 [|               0.7%]
12 [                0.0%]   24 [                0.0%]   36 [                0.0%]   48 [                0.0%]
Mem[||||||                                4.03G/377G]   Tasks: 95, 195 thr; 1 running
Swp[                                       0K/20.0G]   Load average: 0.24 0.16 0.09
                                                       Uptime: 27 days, 11:44:00
```


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
CPU
17% 4,35 GHz
Memory
14,8/15,8 GB (94%)
Disk 0 (C:)
SSD
1%
Ethernet
vEthernet (Default ...
S: 0 R: 0 Kbps
Wi-Fi
Wi-Fi
S: 0 R: 0 Kbps
Ethernet
vEthernet (WSL)
S: 0 R: 0 Kbps
GPU 0
Intel(R) UHD Grap...
2%
CPU
Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz
% Utilization over 60 seconds
100%
Utilization
17%
Speed
4,35 GHz
Processes
303
Threads
3614
Handles
135760
Up time
3:21:36:21
Base speed: 2,11 GHz
Sockets: 1
Cores: 4
Logical processors: 8
Virtualization: Enabled
L1 cache: 256 KB
L2 cache: 1,0 MB
L3 cache: 8,0 MB
Fewer details
Open Resource Monitor



Moral: parallelism is the name of the game

Today

1. Parallelism in general

2. Parallelism in Python

3. Parallelism in practice (i.e., how to make it scale)

Parallelism

What is parallelism?

When several operations are being done *simultaneously*

Parallelism

$$x = \overbrace{\boxed{4}\boxed{9}\boxed{1}\boxed{2}\boxed{3}\boxed{7}\boxed{1}\boxed{8}\boxed{5}\boxed{2}\boxed{0}\boxed{4}\boxed{6}\boxed{7}\boxed{1}\boxed{3}\boxed{2}\boxed{9}\boxed{2}\boxed{5}}^{N}$$

```
x * 2
```

Normally N time units

Parallelism

N

x = 4 9 1 2 3 7 1 8 5 2 0 4 6 7 1 3 2 9 2 5

Task 1 Task 2

x * 2

Normally N time units

2 Processors N / 2 time units

Parallelism

$$x = \overbrace{\underbrace{\boxed{4}\boxed{9}\boxed{1}\boxed{2}\boxed{3}}_{\text{Task 1}}\underbrace{\boxed{7}\boxed{1}\boxed{8}\boxed{5}\boxed{2}}_{\text{Task 2}}\underbrace{\boxed{0}\boxed{4}\boxed{6}\boxed{7}\boxed{1}}_{\text{Task 3}}\underbrace{\boxed{3}\boxed{2}\boxed{9}\boxed{2}\boxed{5}}_{\text{Task 4}}}^{N}$$

```
x * 2
```

Normally	N time units
2 Processors	N / 2 time units
4 Processors	N / 4 time units

Embarrassingly parallel:

Every unit of work is independent.
I.e., everything can be parallelized.

Parallelism

N

x =	4	9	1	2	3	7	1	8	5	2	0	4	6	7	1	3	2	9	2	5

Task 1 Task 2 Task 3 Task 4

`x * 2`

Normally	N time units	1x
2 Processors	N / 2 time units	2x
4 Processors	N / 4 time units	4x

Speed-up:

$$\boldsymbol{S(n) = \frac{T(1)}{T(n)}}$$

Wall-clock time

Parallelism in Python

Multi-threading

&

Multi-processing

Parallelism in Python

GIL says NO!!!

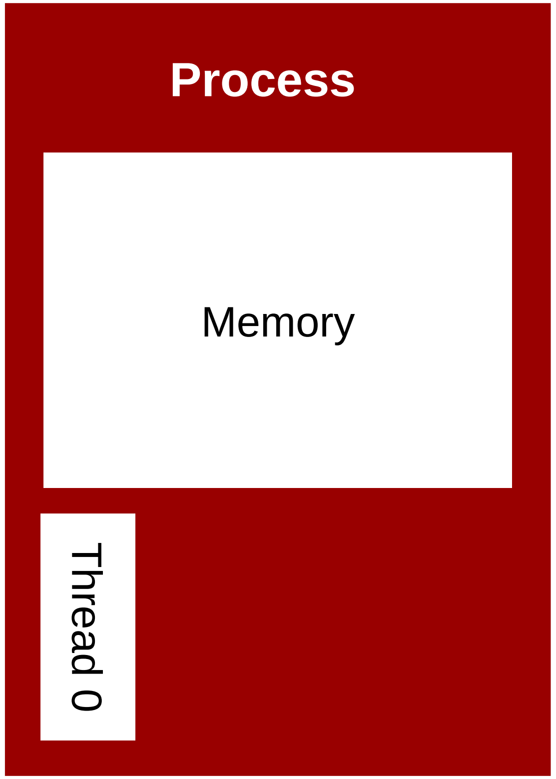
…or?

~~Multi-threading~~

&

Multi-processing

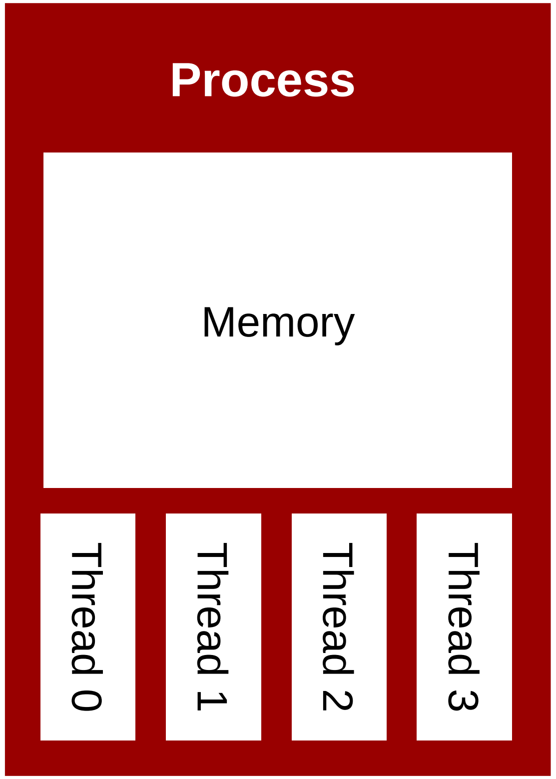
Threads and processes



Process = everything needed to execute a program.

- Memory address space
- Handles to objects
- Etc.
- At least one **thread** of execution

Threads and processes



Process = everything needed to execute a program.

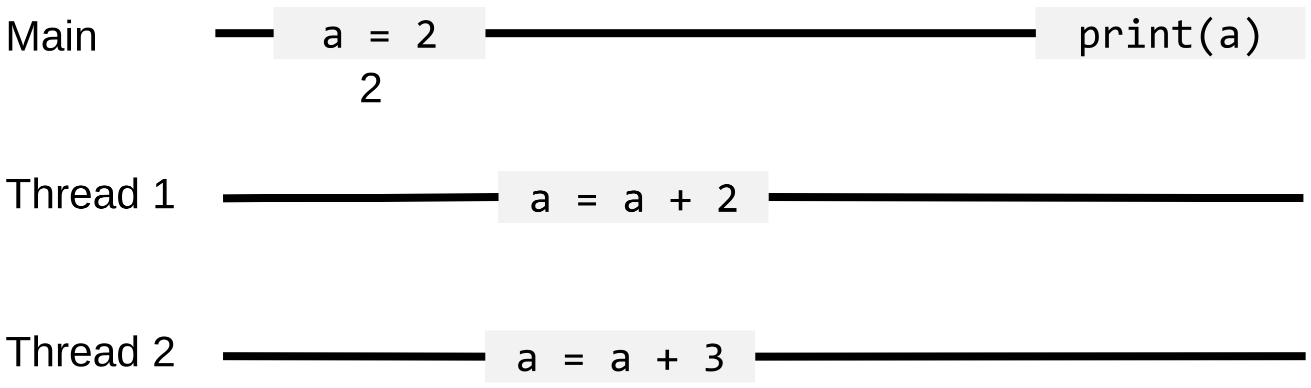
- Memory address space
- Handles to objects
- Etc.
- At least one **thread** of execution

Thread = entity *within* a process to be scheduled for execution.

- Thread local storage and state
- Access to process memory ← **Shared Memory**
- Multiple processers = multiple threads can run in parallel

Threads and processes

```
<< Main >>
a = 2

<< Thread 1 >>
a = a + 2

<< Thread 2 >>
a = a + 3

<< Main >>
wait_for_threads()
print(a)
```

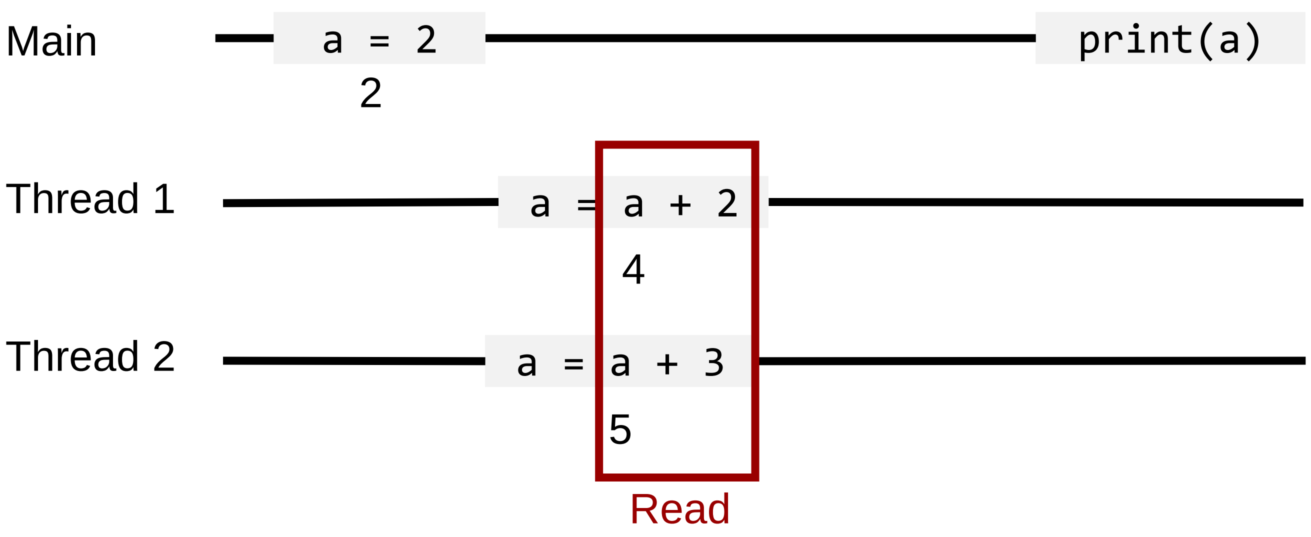


What does this print?

Example from: https://python.land/python-concurrency/the-python-gil

Threads and processes

```
<< Main >>
a = 2

<< Thread 1 >>
a = a + 2

<< Thread 2 >>
a = a + 3

<< Main >>
wait_for_threads()
print(a)
```

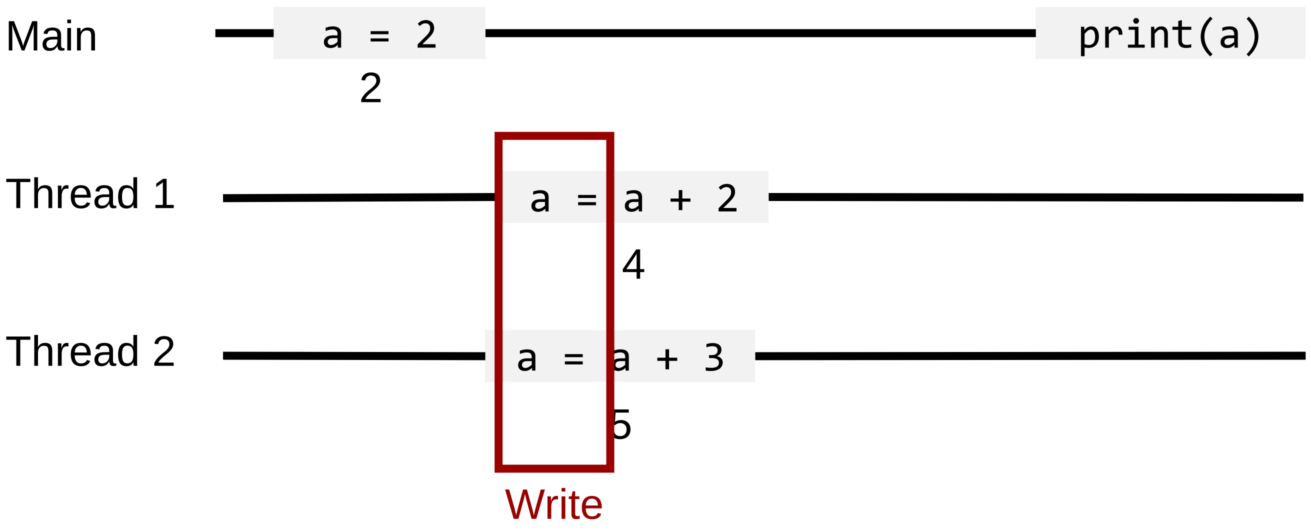


What does this print?

Example from: https://python.land/python-concurrency/the-python-gil

Threads and processes

```
<< Main >>
a = 2

<< Thread 1 >>
a = a + 2

<< Thread 2 >>
a = a + 3

<< Main >>
wait_for_threads()
print(a)
```



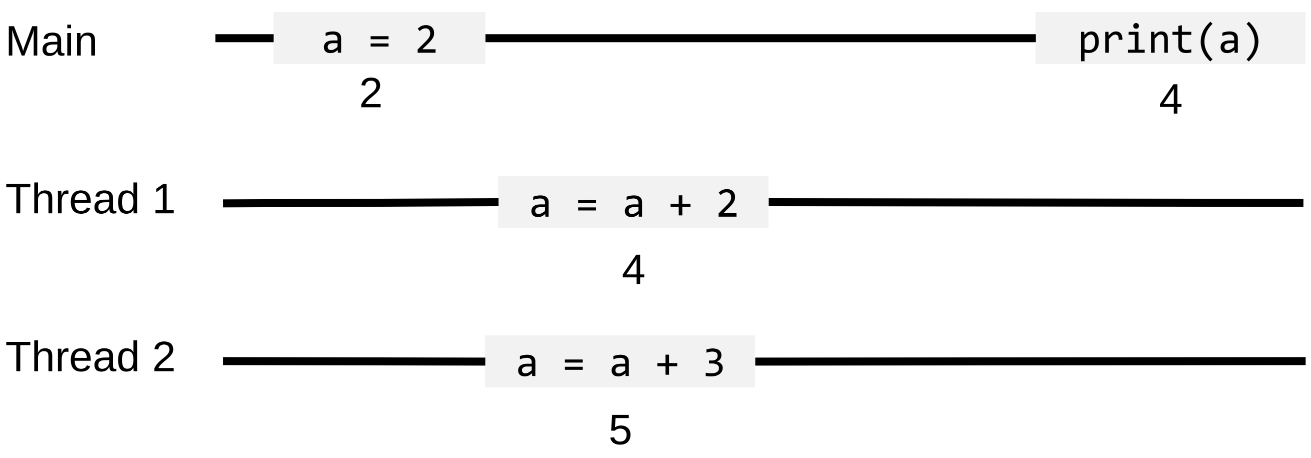
What does this print?

Example from: https://python.land/python-concurrency/the-python-gil

Threads and processes

```
<< Main >>
a = 2

<< Thread 1 >>
a = a + 2

<< Thread 2 >>
a = a + 3

<< Main >>
wait_for_threads()
print(a)
```

What does this print?

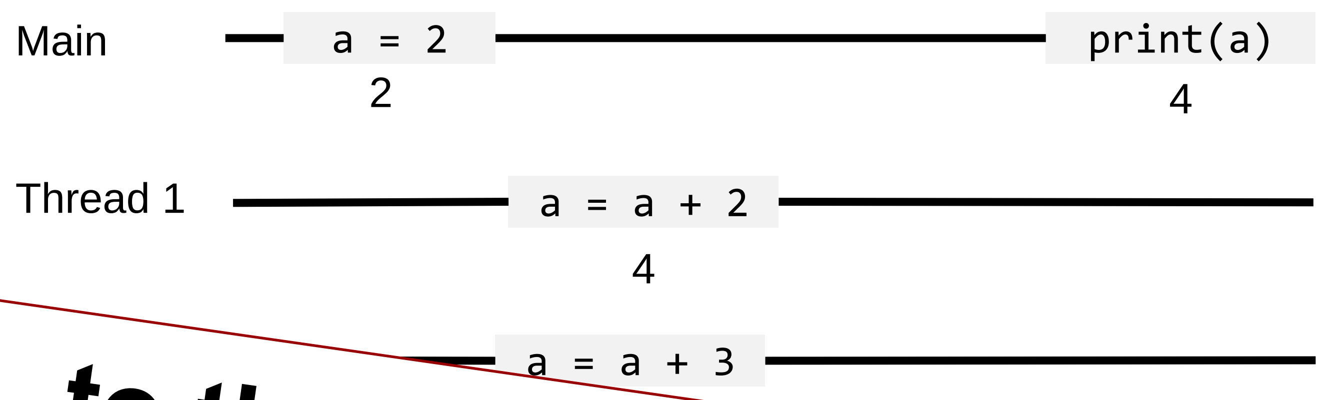


Race condition!

Example from: https://python.land/python-concurrency/the-python-gil

Threads and processes

```
<< Main >>
a = 2

<< Thread 1 >>
a = a + 2

<< Thread 2 >>
a = a + 3

<< Main >>
wait_for_threads()
print(a)
```



What does this print?

Race conditio

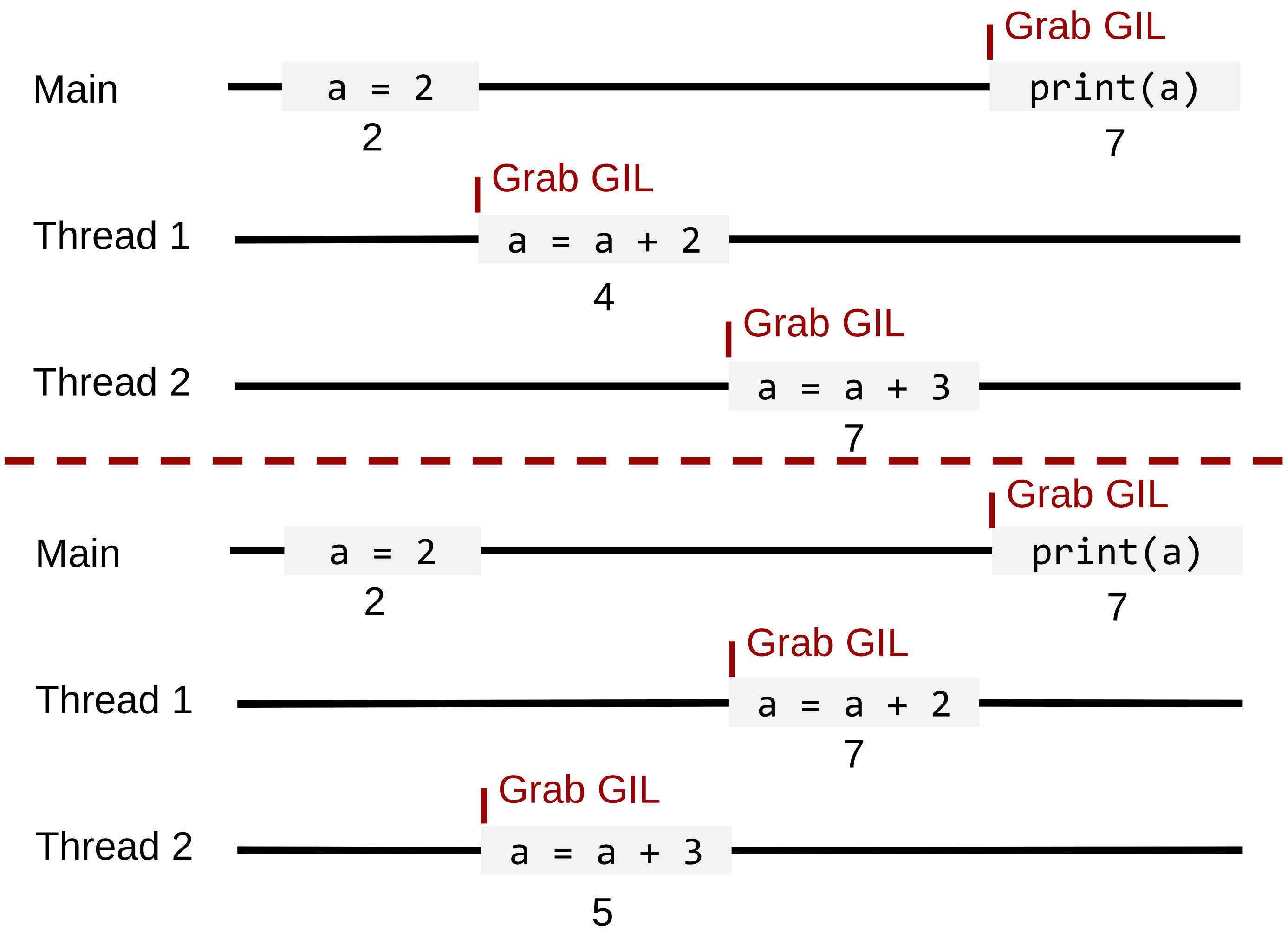
GIL to the rescue!

Example from: https://python.land/python-concurrency/the-python-gil

GIL: Global Interpreter Lock

```
<< Main >>
a = 2

<< Thread 1 >>
a = a + 2

<< Thread 2 >>
a = a + 3

<< Main >>
wait_for_threads()
print(a)
```

What does this print?



Example from: https://python.land/python-concurrency/the-python-gil

GIL: Global Interpreter Lock

Pro:

- Ensures no race conditions – convenient!

Con:

- Compute heavy multi-threaded = impossible

Some operations manually release the GIL:

- File reading
- I/O
- NumPy

Multi-threading

```python
import numpy as np


arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

for a in arr_x10:
    np.sum(a)
```

sums.py

```
$ time python sums.py  # Single thread
```

Multi-threading

```python
import numpy as np


arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

for a in arr_x10:
    np.sum(a)
```

sums.py

```
$ time python sums.py  # Single thread

real    0m3.634s
user    0m3.573s
sys     0m0.036s
```

Wall-clock time

CPU time

Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(2) as pool:
    pool.map(np.sum, arr_x10)
```

sums.py

```
$ time python sums.py  # Single thread

real    0m3.634s
user    0m3.573s
sys     0m0.036s
$ time python sums.py  # 2 threads
```

Wall-clock time

CPU time

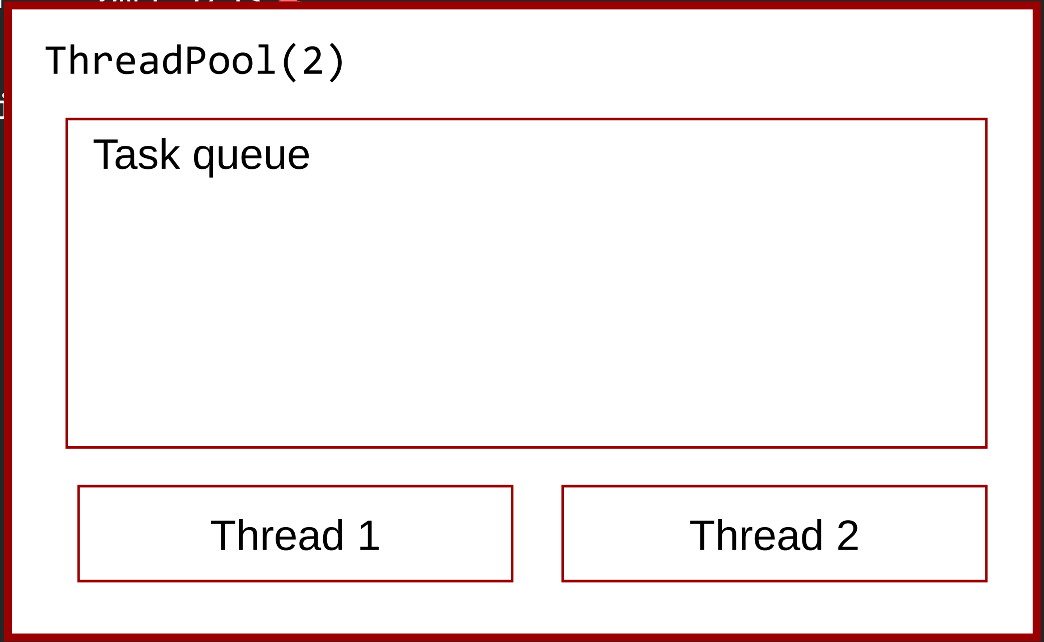
Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(2) as pool:
    pool.map(np.sum, arr_x10)
```

sums.py

```
$ time python sums.py  # Single thread

real    0m3.634s
user    0m3.573s
sys
$ ti
```

Wall-clock time



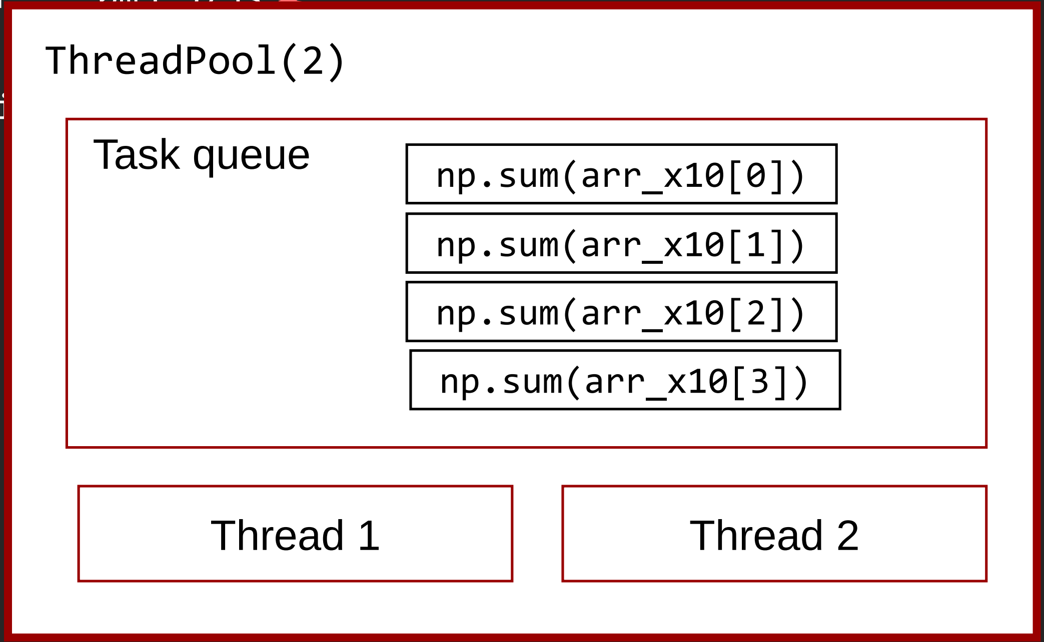
Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(2) as pool:
    pool.map(np.sum, arr_x10)
```

sums.py

```
$ time python sums.py  # Single thread

real    0m3.634s
user    0m3.573s
sys
$ ti
```

Wall-clock time



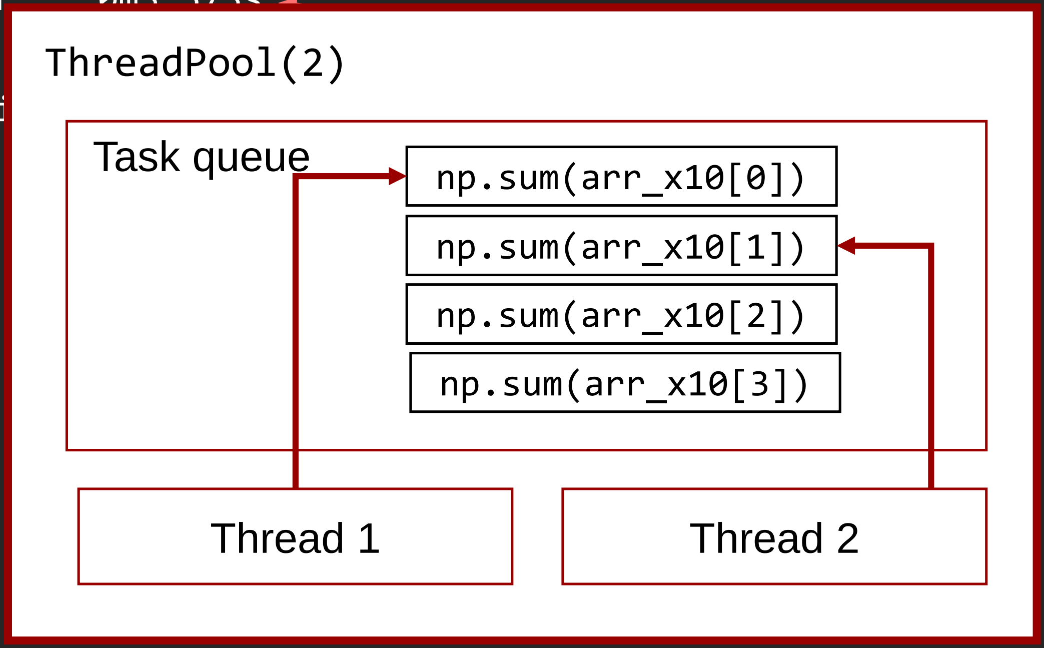
Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(2) as pool:
    pool.map(np.sum, arr_x10)
```

sums.py

```
$ time python sums.py  # Single thread

real    0m3.634s
user    0m3.573s
sys
$ ti
```

Wall-clock time



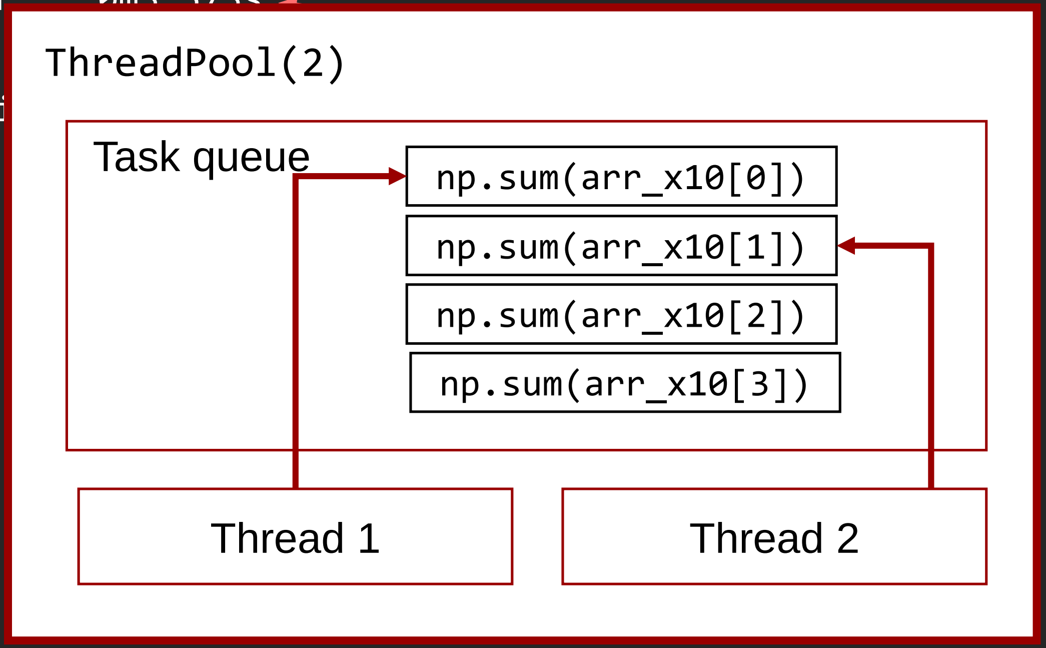
Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(2) as pool:
    for a in arr_x10:
        # Add to task queue
        pool.apply_async(np.sum, (a,))
    pool.close()  # No more tasks
    pool.join()   # Wait for completion
```

sums.py

```
$ time python sums.py  # Single thread

real    0m3.634s
user    0m3.573s
sys
$ ti
```

Wall-clock time



Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(2) as pool:
    pool.map(np.sum, arr_x10)
```

sums.py

```
$ time python sums.py  # Single thread

real    0m3.634s
user    0m3.573s
sys     0m0.036s
$ time python sums.py  # 2 threads

real    0m1.873s
user    0m3.179s
sys     0m0.047s
```

Wall-clock time

CPU time

Speed-up:
3.624 / 1.873 = 1.94

Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(4) as pool:
    pool.map(np.sum, arr_x10)
```

sums.py

```
$ time python sums.py  # Single thread

real    0m3.634s
user    0m3.573s
sys     0m0.036s
$ time python sums.py  # 2 threads

real    0m1.873s
user    0m3.179s
sys     0m0.047s
$ time python sums.py  # 4 threads

real    0m1.136s
user    0m3.282s
sys     0m0.055s
```

Wall-clock time

CPU time

Speed-up:
3.624 / 1.873 = 1.94

Speed-up:
3.624 / 1.136 = 3.20

Multi-threading

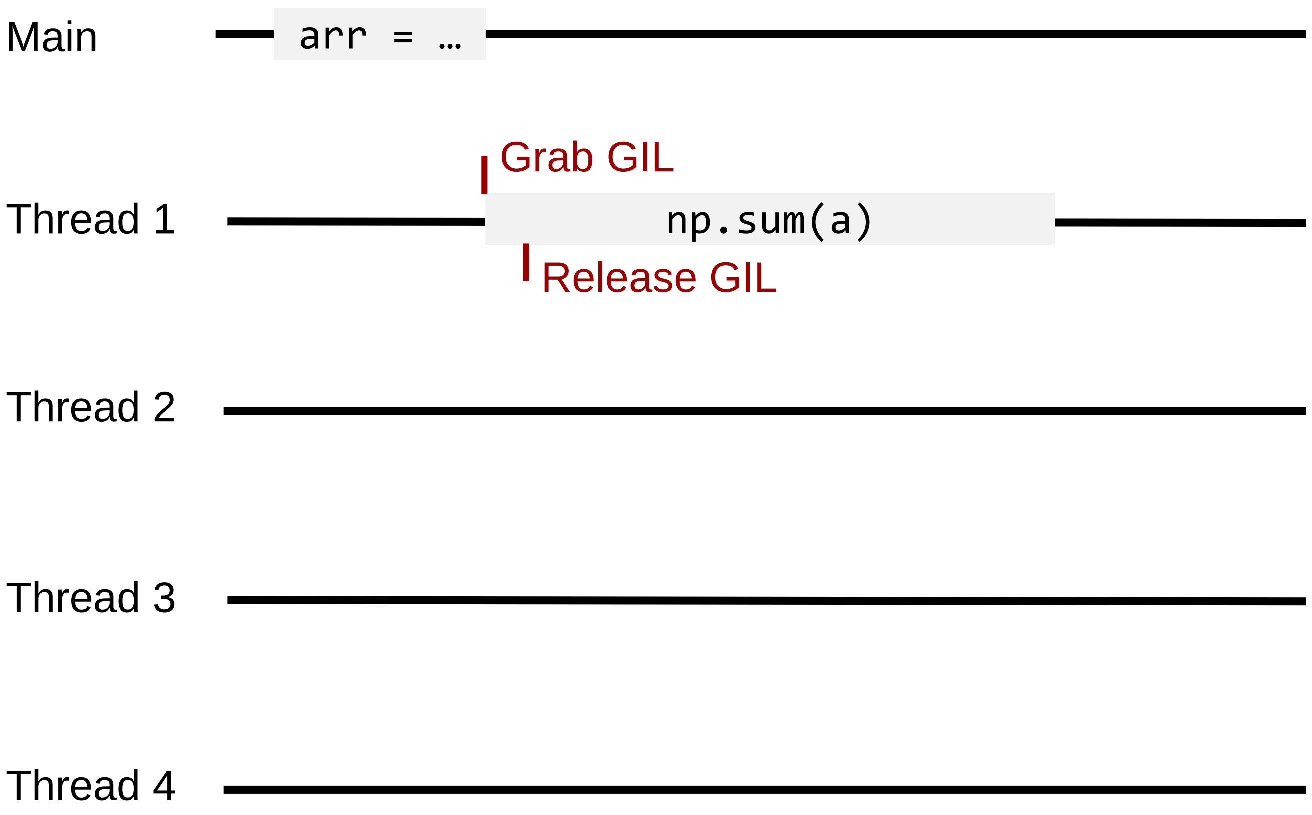
```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(4) as pool:
    pool.map(np.sum, arr_x10)
```

sums.py

Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(4) as pool:
    pool.map(np.sum, arr_x10)
```

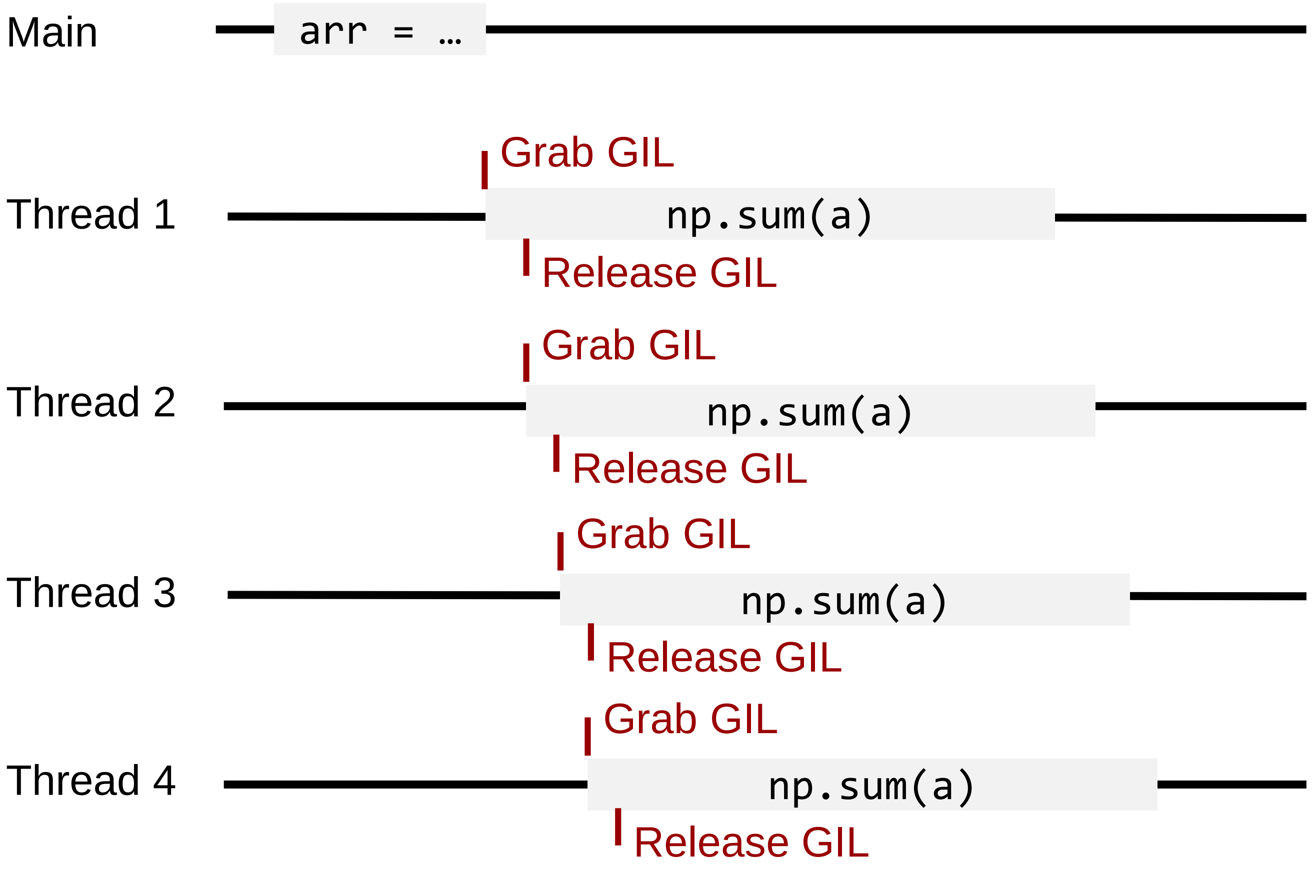
sums.py



Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(4) as pool:
    pool.map(np.sum, arr_x10)
```

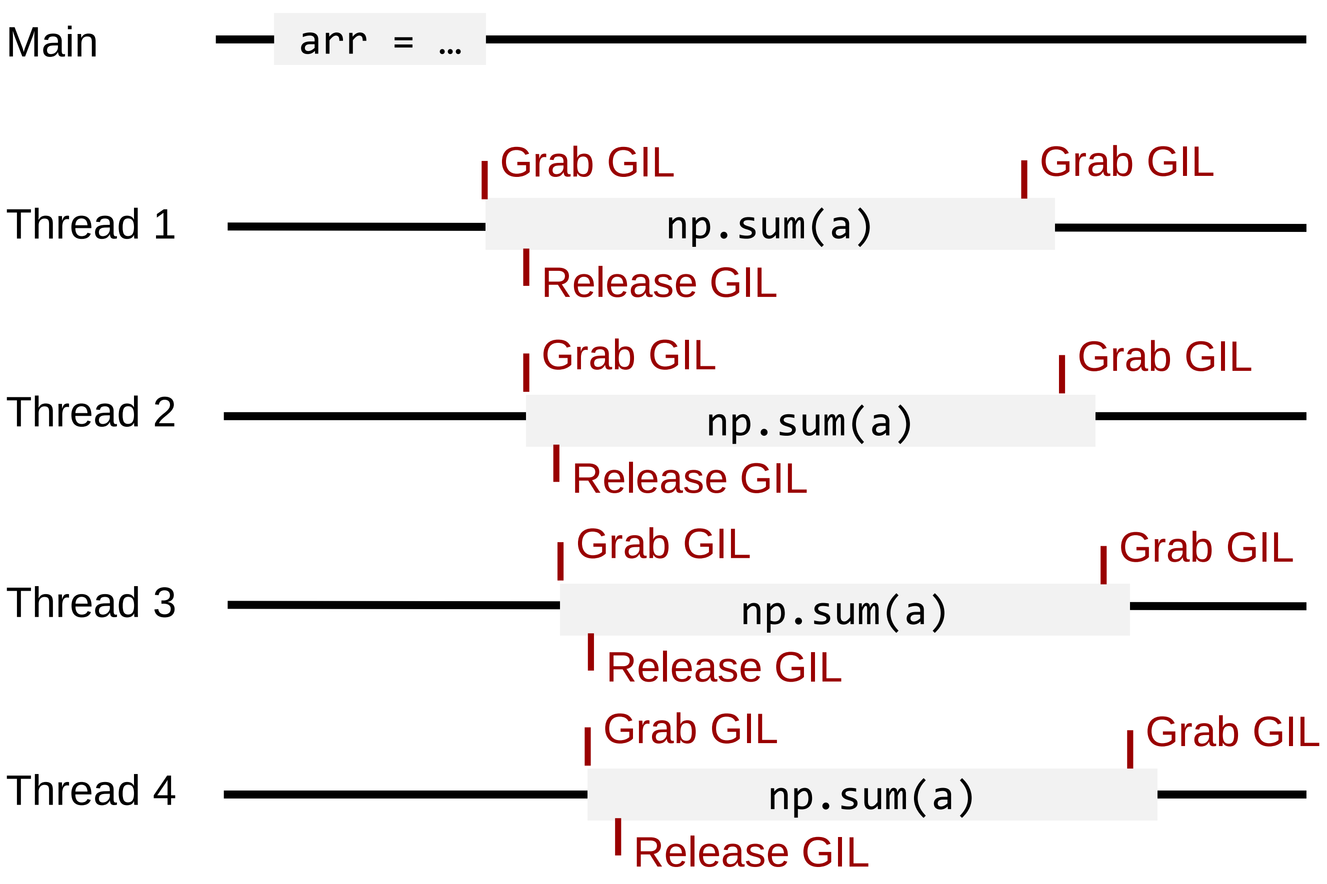
sums.py



Multi-threading

```
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(4) as pool:
    pool.map(np.sum, arr_x10)
```

sums.py



Multi-threading

```python
import numpy as np


arr = [0] * 10_000_000
arr_x10 = [arr] * 10

def manual_sum(arr):
    s = 0
    for a in arr:
        s += a
    return s

for a in arr_x10:
    manual_sum(a)
```

sums.py

```
$ time python sums.py # Single thread

real    0m1.916s
user    0m1.800s
sys     0m0.069s
```

Wall-clock time

CPU time

Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = [0] * 10_000_000
arr_x10 = [arr] * 10

def manual_sum(arr):
    s = 0
    for a in arr:
        s += a
    return s

with ThreadPool(2) as pool:
    pool.map(sum, arr_x10)
```

sums.py

```
$ time python sums.py # Single thread

real    0m1.916s
user    0m1.800s
sys     0m0.069s
$ time python sums.py # 2 threads

real    0m1.855s
user    0m1.790s
sys     0m0.048s
```

Wall-clock time

CPU time

Multi-threading

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = [0] * 10_000_000
arr_x10 = [arr] * 10

def manual_sum(arr):
    s = 0
    for a in arr:
        s += a
    return s

with ThreadPool(4) as pool:
    pool.map(sum, arr_x10)
```

sums.py

```
$ time python sums.py # Single thread

real    0m1.916s
user    0m1.800s
sys     0m0.069s
$ time python sums.py # 2 threads

real    0m1.855s
user    0m1.790s
sys     0m0.048s
$ time python sums.py # 4 threads

real    0m3.254s
user    0m3.157s
sys     0m0.075s
```

Wall-clock time

CPU time

Multi-threading

```
import numpy as np
from multiprocessing.pool import ThreadPool

arr = [0] * 10_000_000
arr_x10 =

def manu
    s =
    for

    return s

with ThreadPool(4) as pool:
    pool.map(sum, arr_x10)
```

sums.py

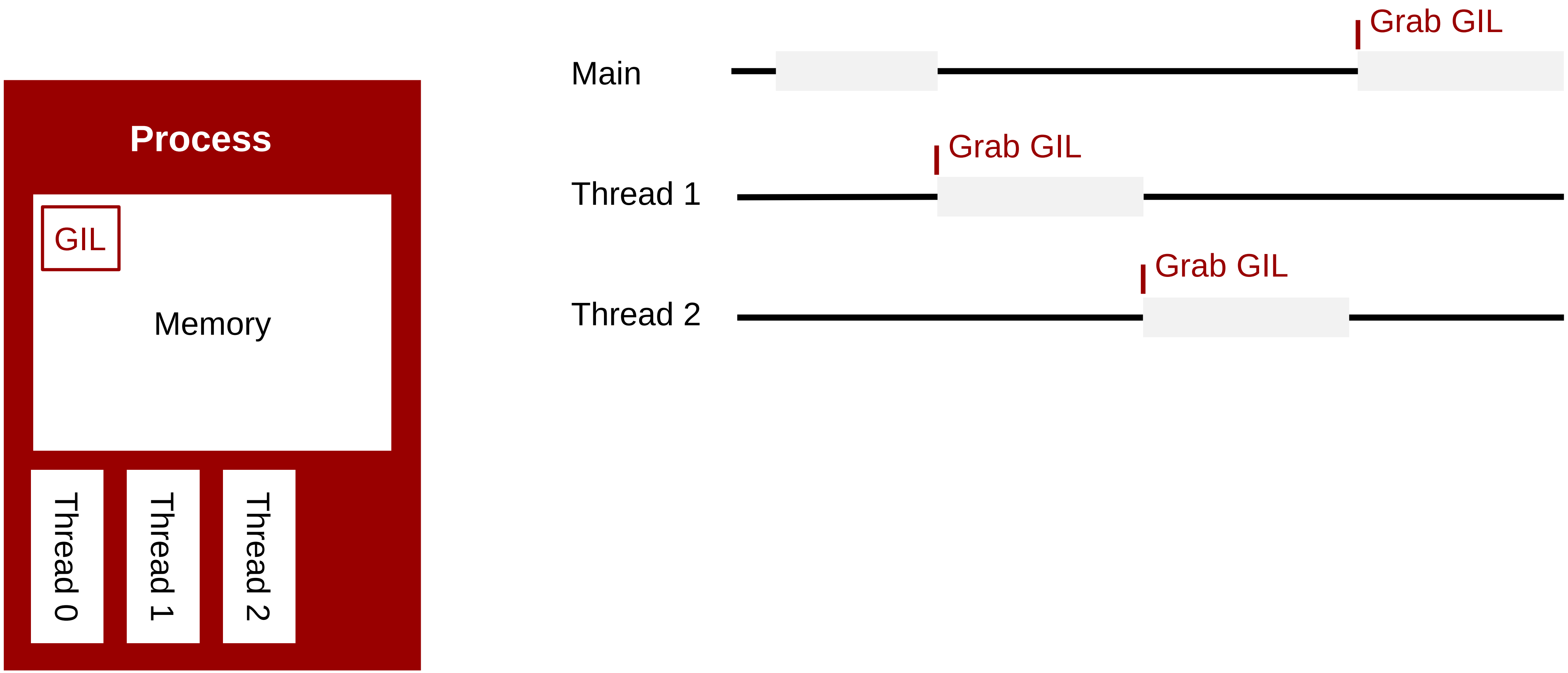
```
$ time python sums.py # Single thread

real    0m1.916s
user    0m1.800s
sys     0m0.069s
             .py # 2 threads

real    0m3.254s
user    0m3.157s
sys     0m0.075s
```

Wall-clock time

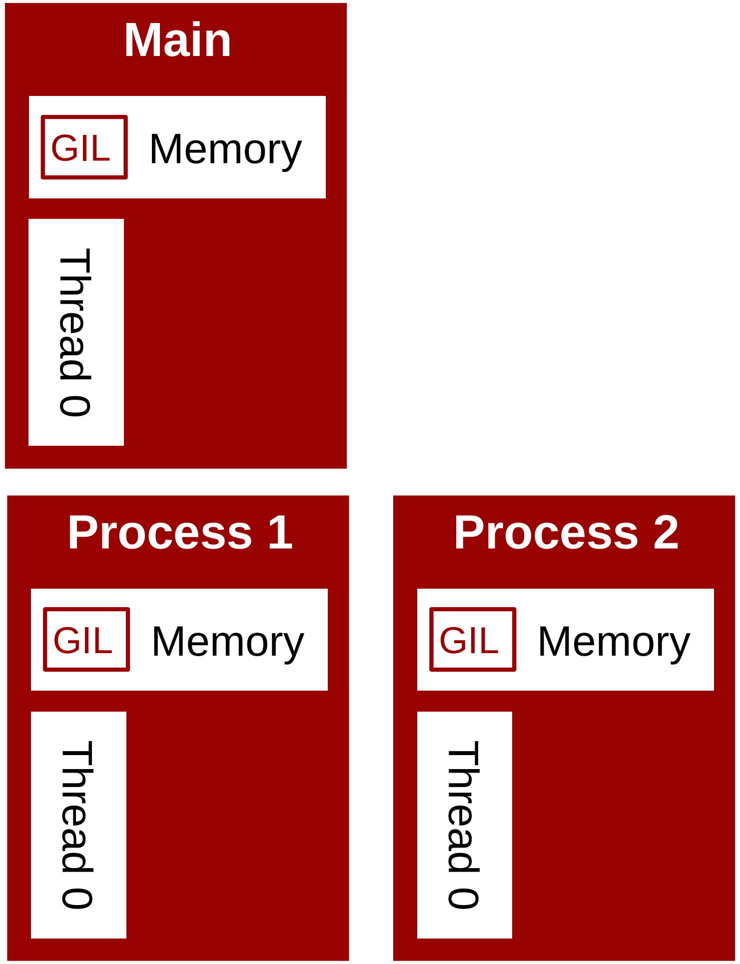
CPU time

Moral: multi-threading *only* works if threads almost never hold the GIL

Multi-processing



Multi-processing



Pro:

Processes can always run in parallel

Cons:

Processes have more overhead

No shared memory – must explicitly **copy**

Multi-processing

```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = [0] * 10_000_000
arr_x10 = [arr] * 10

def manual_sum(arr):
    s = 0
    for a in arr:
        s += a
    return s

with ThreadPool(2) as pool:
    pool.map(manual_sum, arr_x10)
```

sums.py

```
$ time python sums.py # Single thread

real    0m1.793s
user    0m1.716s
sys     0m0.048s
```

Multi-processing

```python
import numpy as np
from multiprocessing.pool import Pool

arr = [0] * 10_000_000
arr_x10 = [arr] * 10

def manual_sum(arr):
    s = 0
    for a in arr:
        s += a
    return s

with Pool(2) as pool:
    pool.map(manual_sum, arr_x10)
```

sums.py

```
$ time python sums.py # Single thread

real    0m1.793s
user    0m1.716s
sys     0m0.048s
```

Multi-processing

```python
import numpy as np
from multiprocessing.pool import Pool

arr = [0] * 10_000_000
arr_x10 = [arr] * 10

def manual_sum(arr):
    s = 0
    for a in arr:
        s += a
    return s

with Pool(1) as pool:
    pool.map(manual_sum, arr_x10)
```

sums.py

```
$ time python sums.py # Single thread

real    0m1.793s
user    0m1.716s
sys     0m0.048s
$ time python sums.py # 1 worker process

real    0m2.410s
user    0m2.315s
sys     0m0.172s
```

Overhead!

Multi-processing

```python
import numpy as np
from multiprocessing.pool import Pool

arr = [0] * 10_000_000
arr_x10 = [arr] * 10

def manual_sum(arr):
    s = 0
    for a in arr:
        s += a
    return s

with Pool(2) as pool:
    pool.map(manual_sum, arr_x10)
```

sums.py

```
$ time python sums.py # Single thread

real    0m1.793s
user    0m1.716s
sys     0m0.048s
$ time python sums.py # 1 worker process

real    0m2.410s
user    0m2.315s
sys     0m0.172s

$ time python sums.py # 2 worker processes
real    0m1.620s
user    0m2.299s
sys     0m0.187s
```

Speed-up:
2.410 / 1.620 = 1.49

Multi-processing

```
import numpy as np
from multiprocessing.pool import Pool

arr = [0] * 10_000_000
arr_x10 =

def manu
    s =
    for

    return s

with Pool(2) as pool:
    pool.map(manual_sum, arr_x10)
```

sums.py

```
$ time python sums.py # Single thread

real    0m1.793s
user    0m1.716s
sys     0m0.048s
                  py # 1 worker process

$ time python sums.py
real    0m1.620s
user    0m2.299s
sys     0m0.187s
```

Speed-up:
2.410 / 1.620 = 1.49

Moral: multi-processing has a lot of overhead but does work

Quiz time!

menti.com

Mentimeter

Parallelism in practice

how to make it scale

I just spent 2 days making it parallel, why the &#@! isn't it faster!?

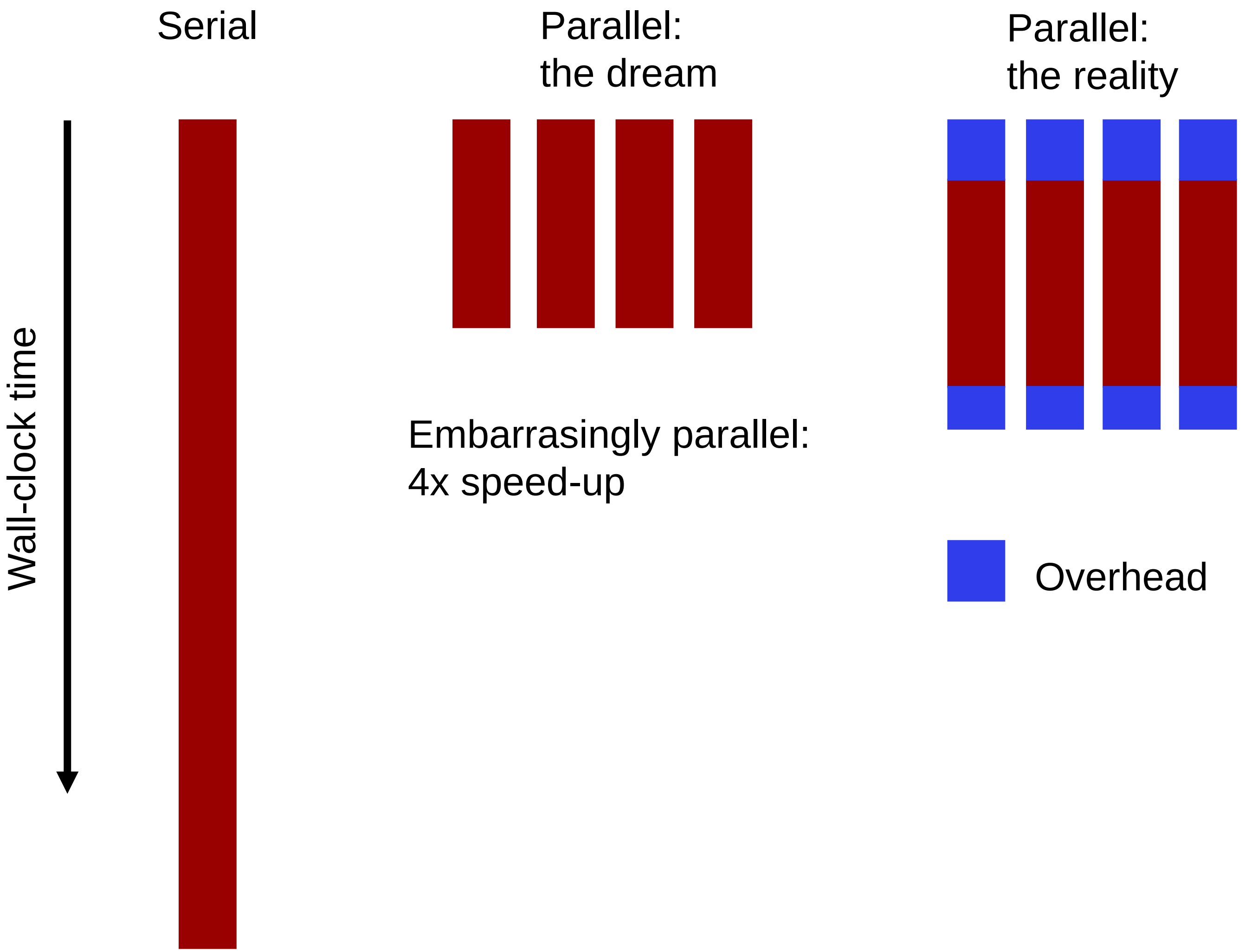
1. Parallelization overhead

2. Too much communication

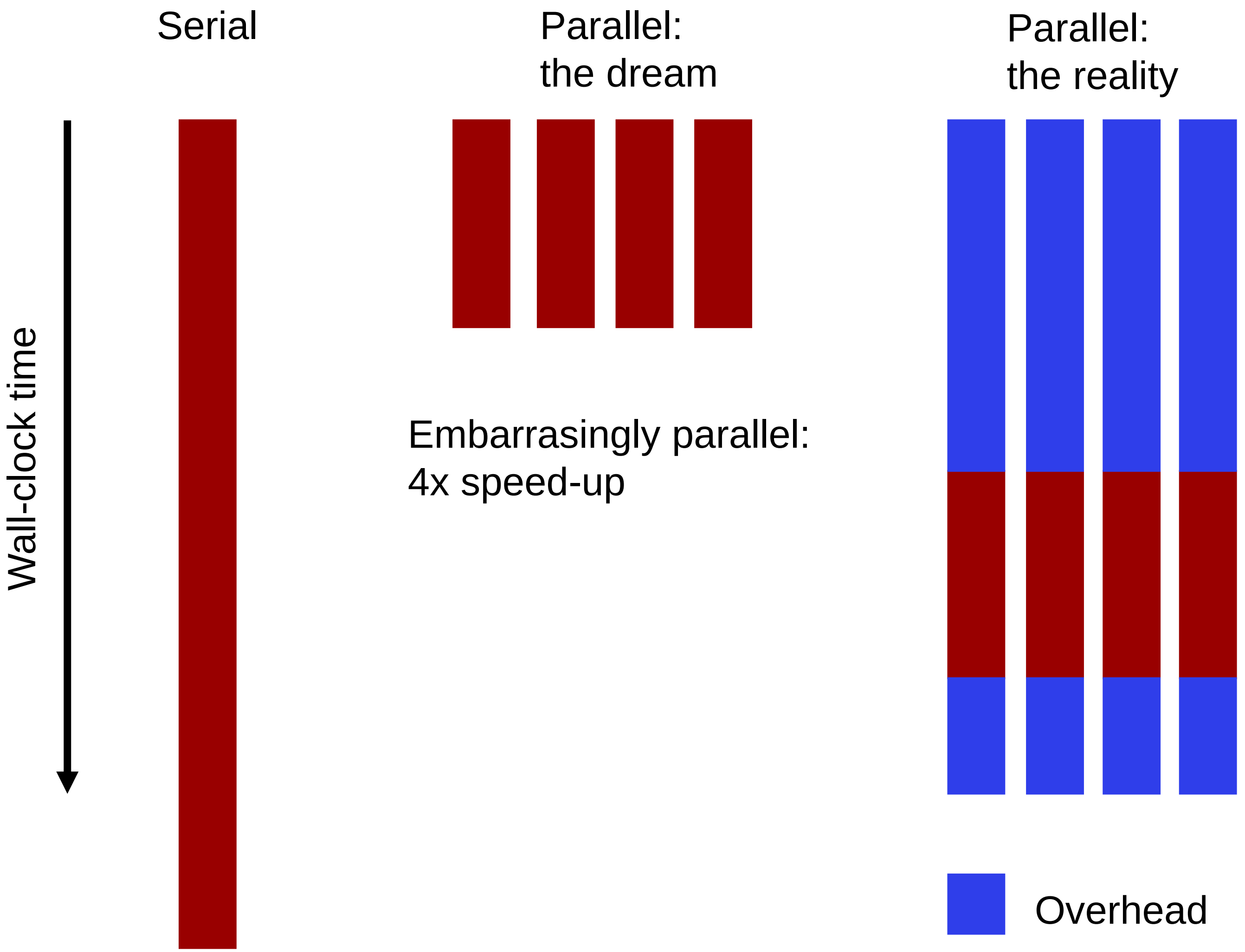
3. Load balancing

4. Amdahl's law

Parallelization overhead



Parallelization overhead



Problem:
Parallelization overhead is larger than runtime

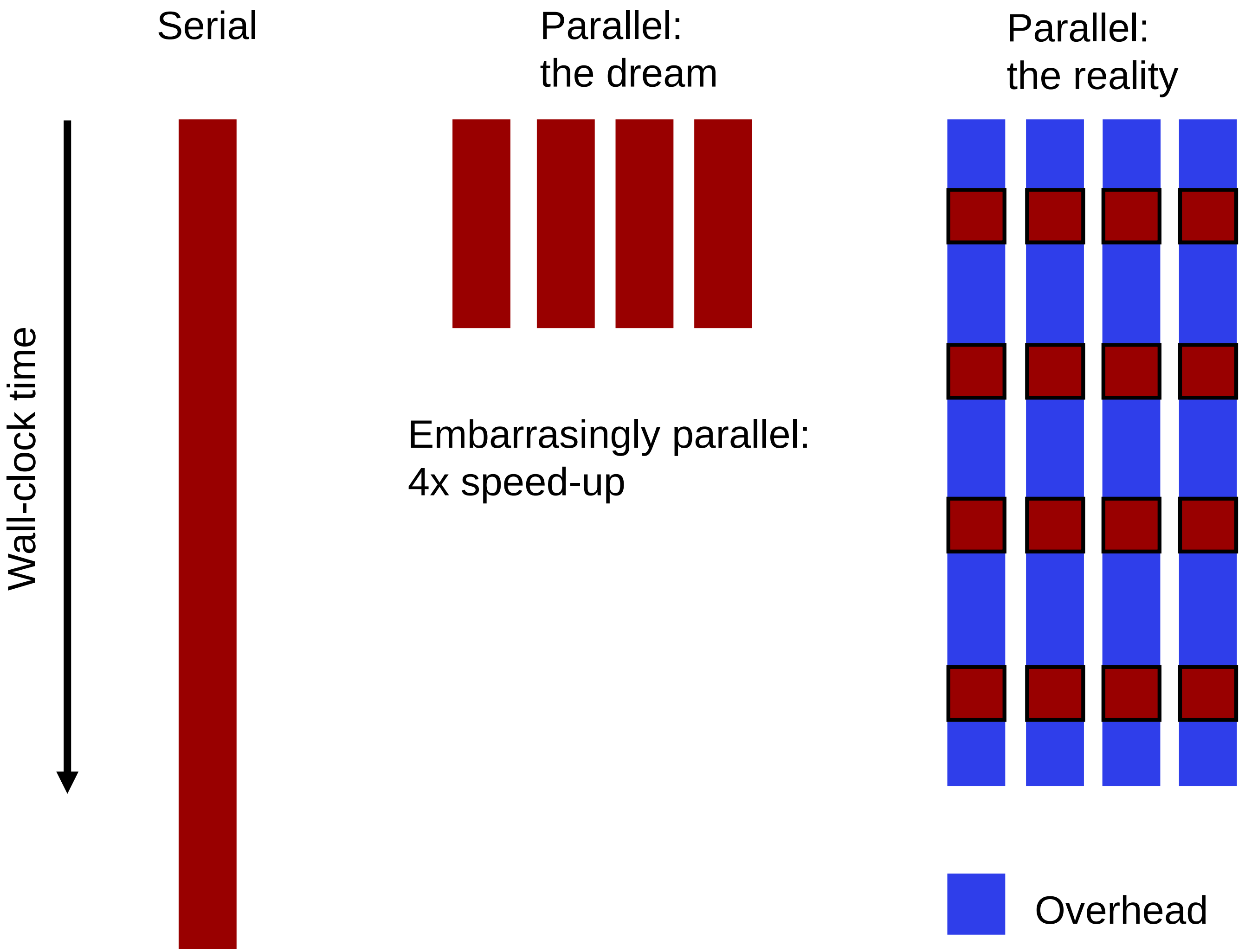
Solution:

Sorry! ☹

Escape to C

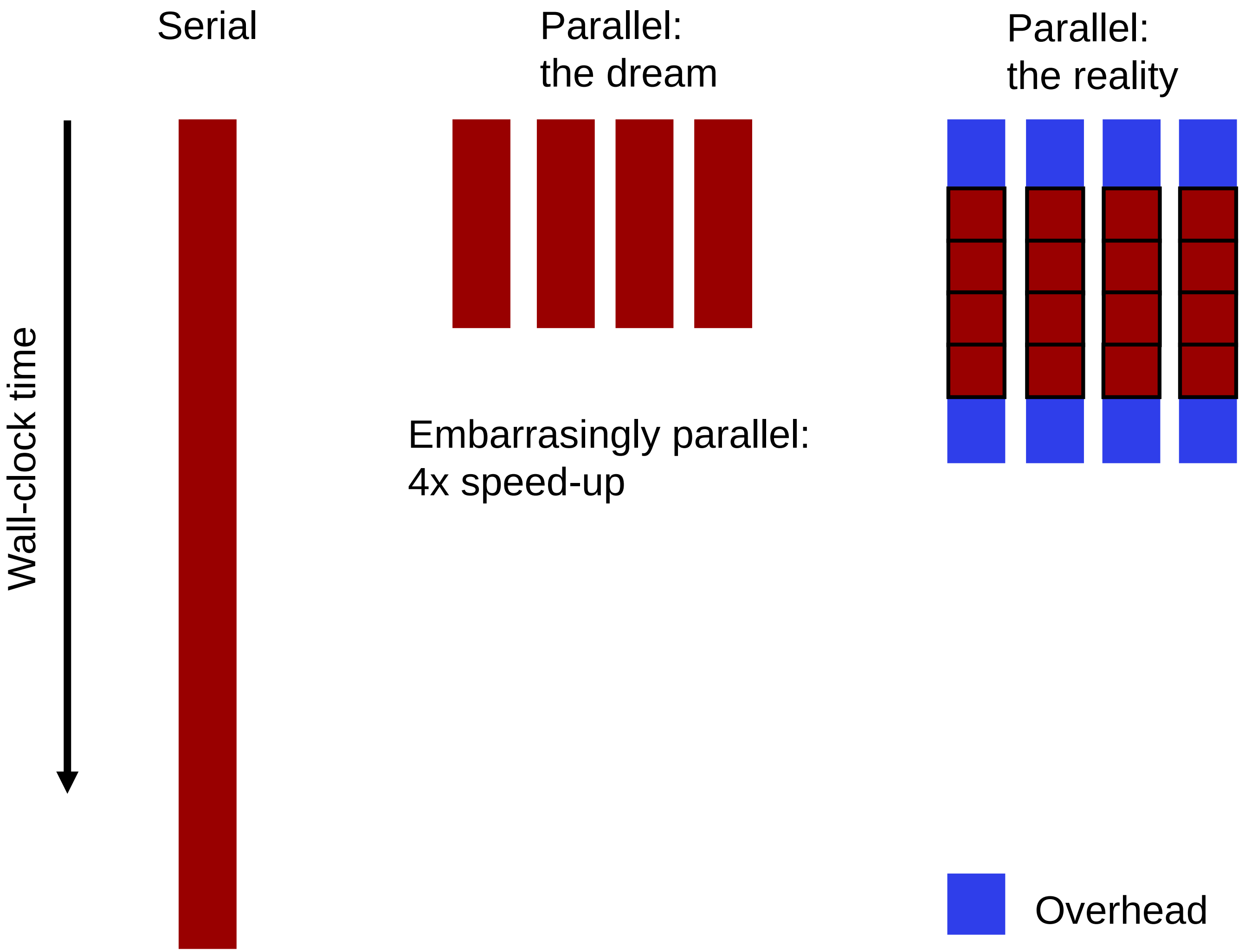
Find a Python package

Parallelization overhead



Problem:
Parallelization overhead is larger than runtime

Parallelization overhead



Problem:
Parallelization overhead is larger than runtime

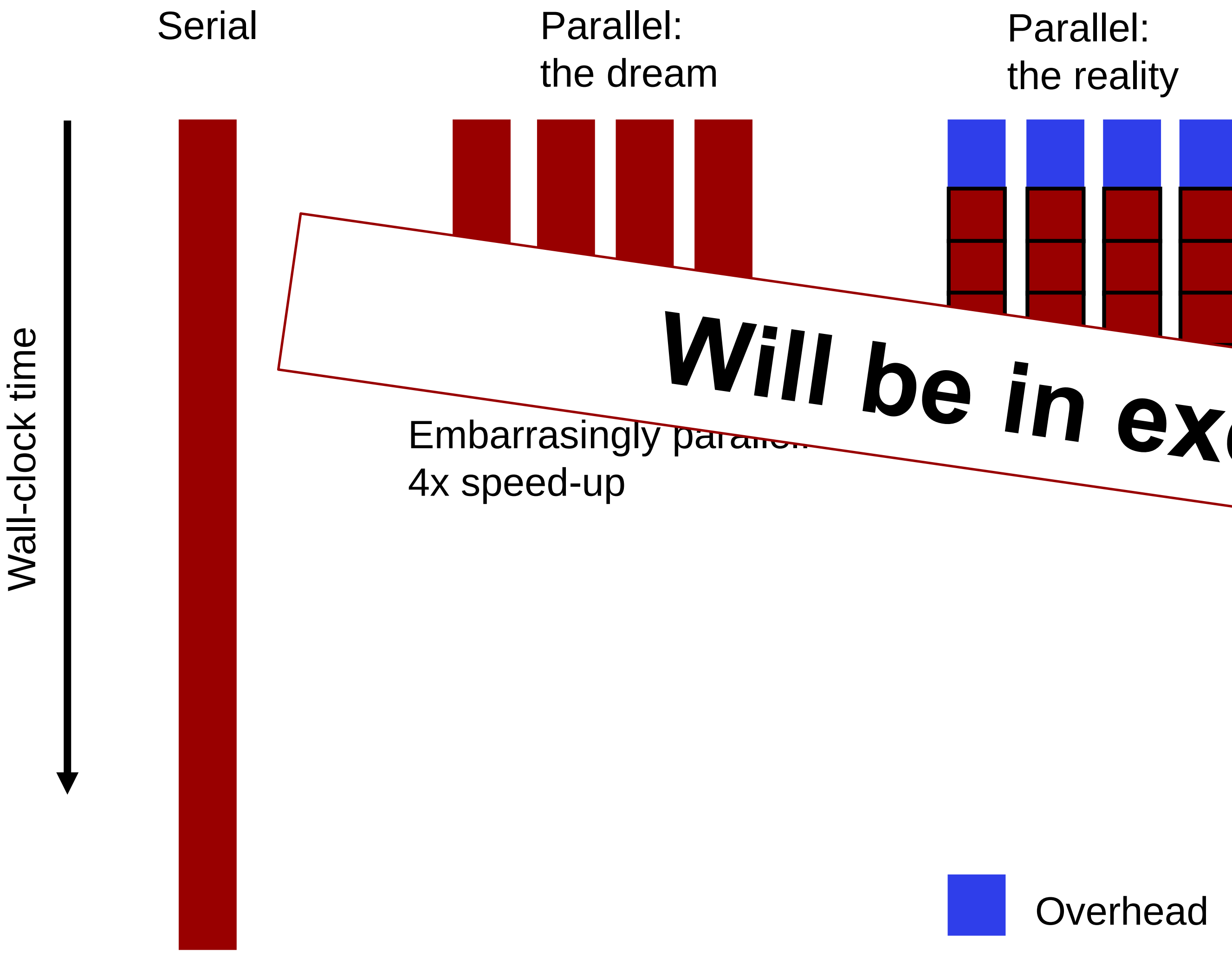
Solution:
Chunking

```
pool.map(f, arr,
         chunksize=N)
```

Manual with `apply_async`

Can chunk both data and work!

Parallelization overhead



Will be in exercises

Problem:
Parallelization overhead is larger than runtime

Solution:

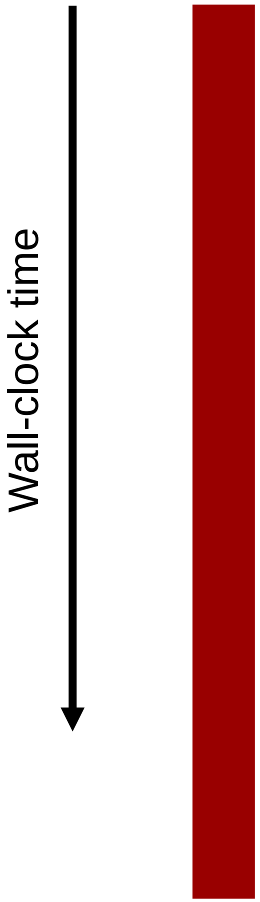
```
pool.map(
         chunksize=N)
```

Manual with `apply_async`

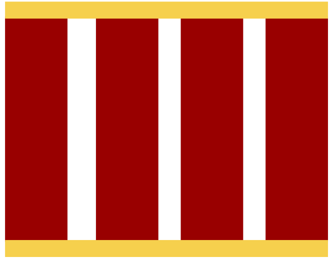
Can chunk both data and work!

Too much communication

Serial



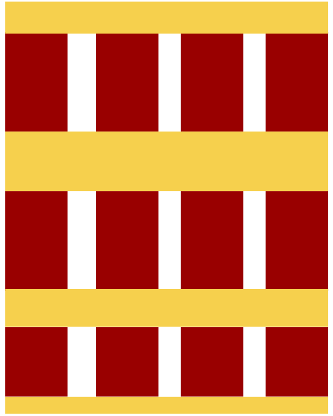
Parallel:
the dream



Embarrasingly parallel:
~4x speed-up

Only communicate at ends

Parallel:
the reality



Communication

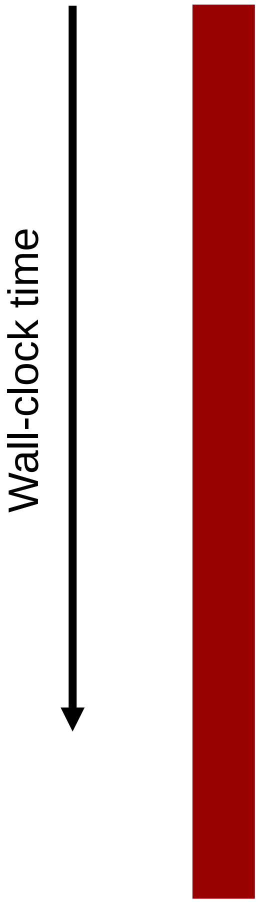
Problem:
Communication adds overhead

Examples:

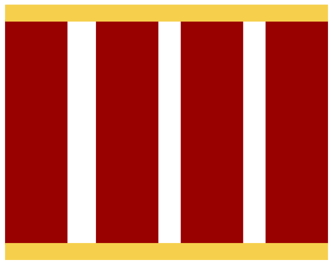
- Training neural nets on multiple GPUs
- Solving PDEs on large grids with iterative methods
- Particle simulations with attraction (from book)

Too much communication

Serial



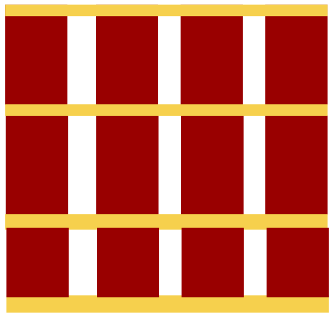
Parallel:
the dream



Embarrasingly parallel:
~4x speed-up

Only communicate at ends

Parallel:
the reality

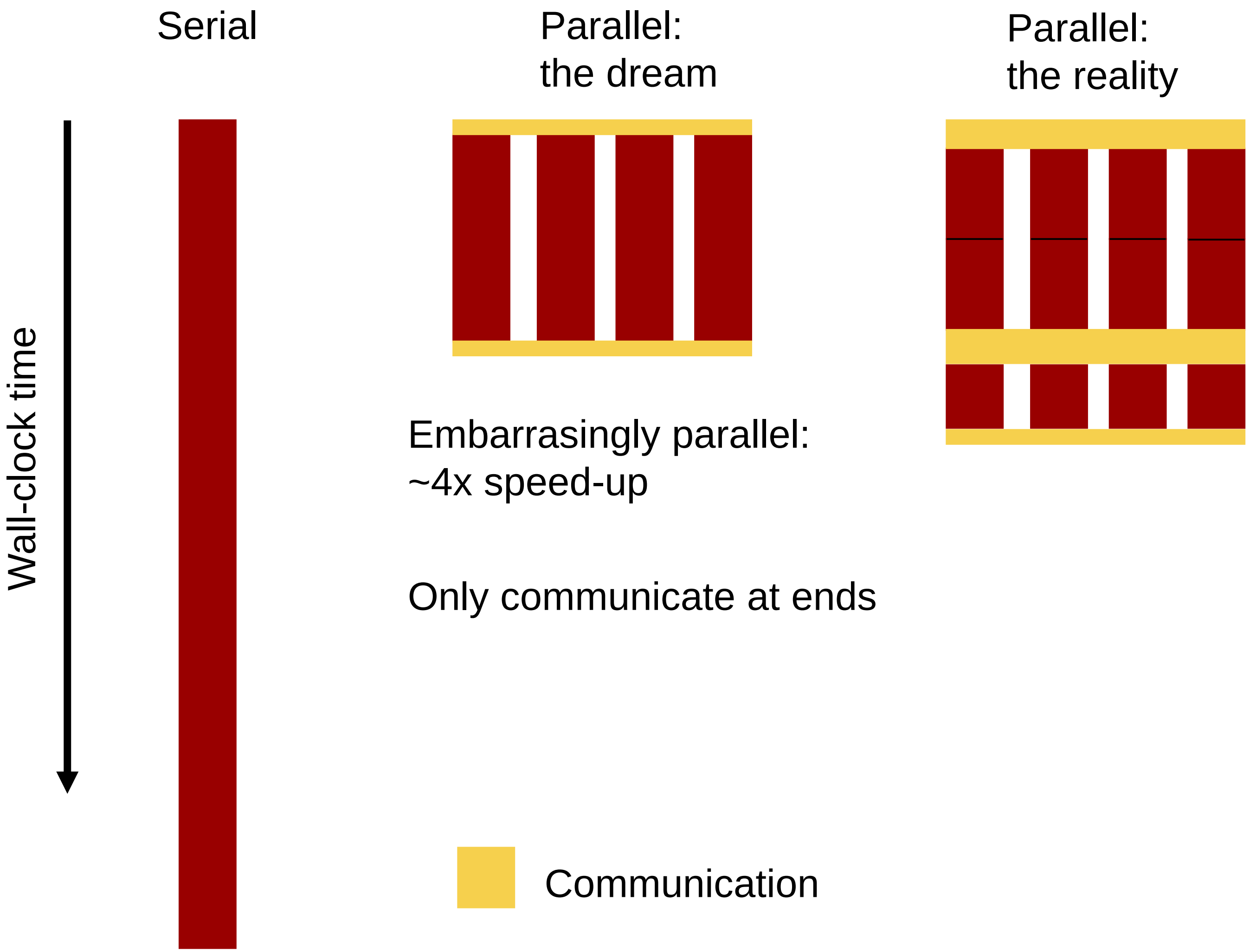


Communication

Problem:
Communication adds overhead

Solution:
Send less data

Too much communication



Embarrasingly parallel:
~4x speed-up

Only communicate at ends

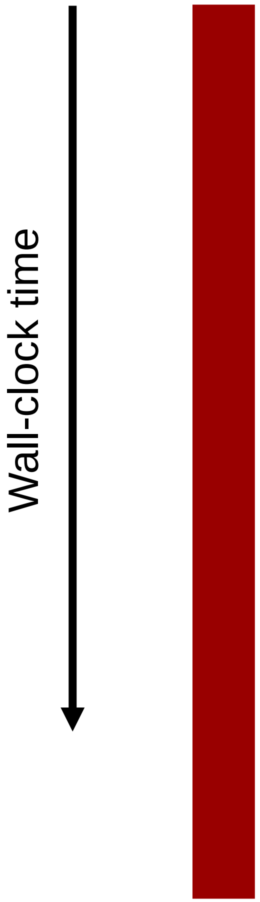
Problem:
Communication adds overhead

Solution:
Send less data

Chunk work / skip communication
WARNING: May *<u>not</u>* be correct!

Load balancing

Serial

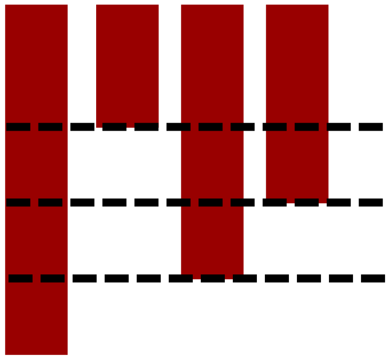


Parallel:
the dream

Embarrasingly parallel:
4x speed-up

All threads finish at
the same time

Parallel:
the reality



1 idle, 3 running

2 idle, 2 running

3 idle, 1 running

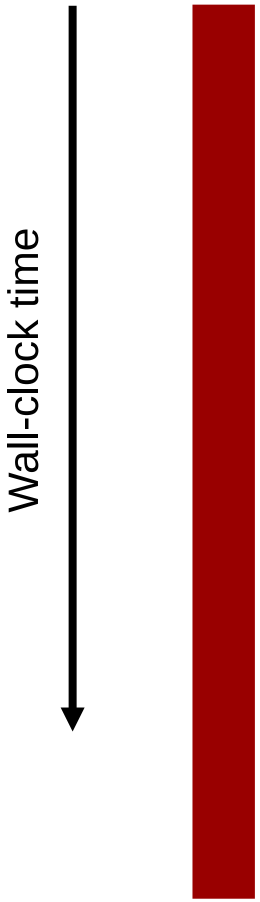
Problem:
Work is unevenly distributed

Typical for **static scheduling**

= Divide all work up front and then start running

Load balancing

Serial

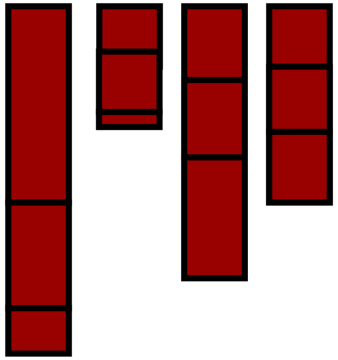


Parallel:
the dream

Embarrasingly parallel:
4x speed-up

All threads finish at
the same time

Parallel:
the reality



Typical for **static scheduling**

= Divide all work up front and then
start running

Problem:
Work is unevenly distributed

Solution

Use **dynamic scheduling** and smaller chunks

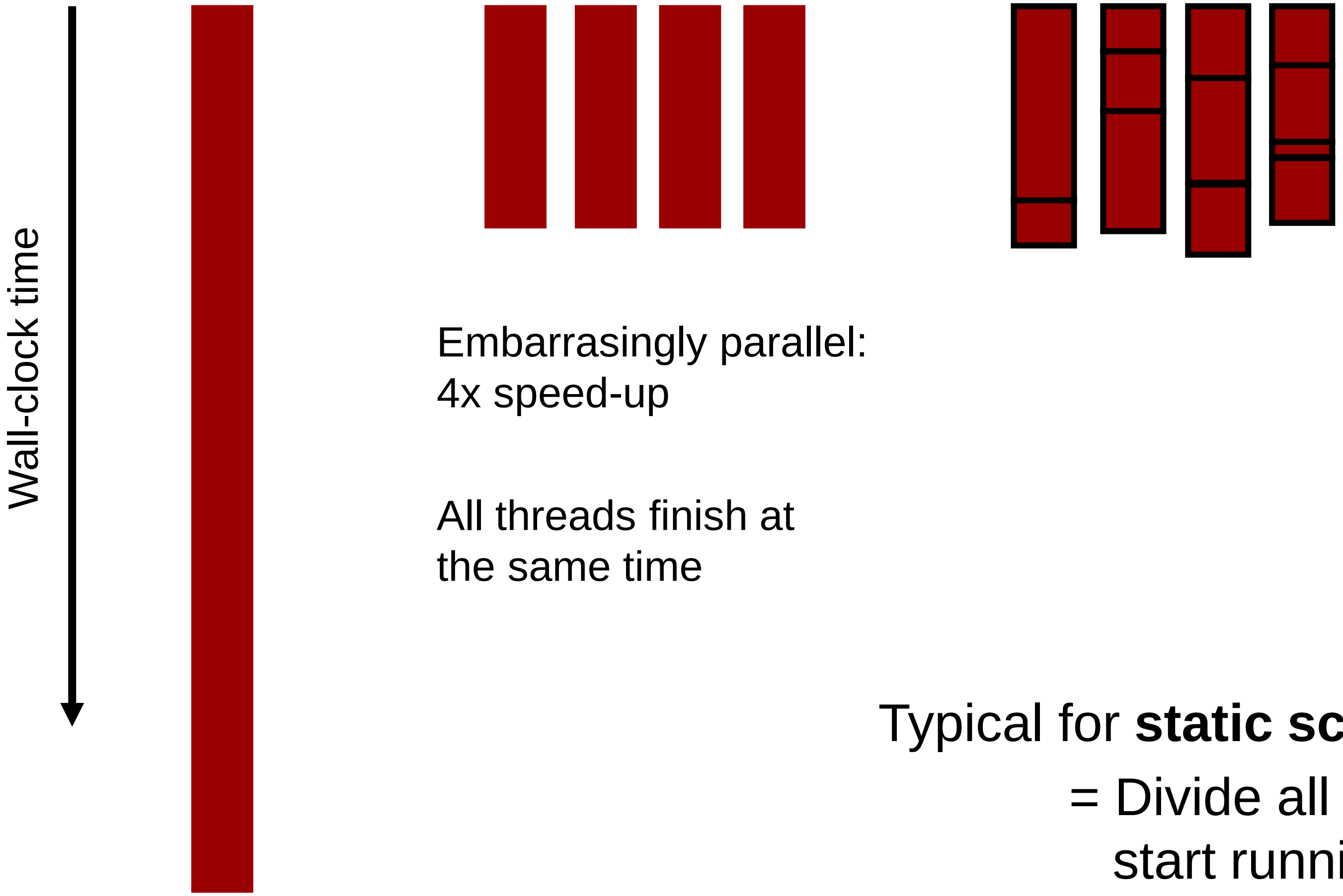
Already built-in with Python’s Pool objects!

Load balancing

Serial

Parallel:
the dream

Parallel:
the reality



Embarrasingly parallel:
4x speed-up

All threads finish at
the same time

Problem:
Work is unevenly distributed

Solution

Use **dynamic scheduling** and smaller chunks

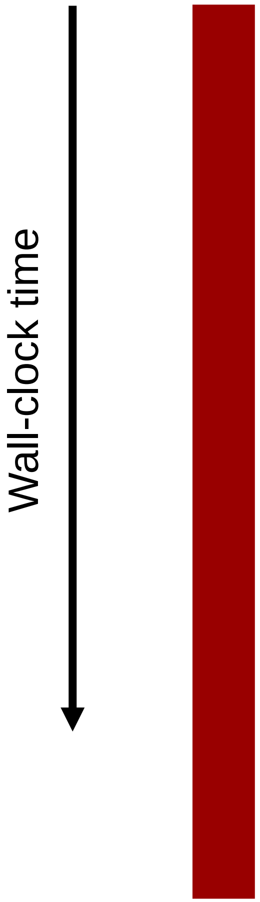
Already built-in with Python's Pool objects!

Typical for **static scheduling**

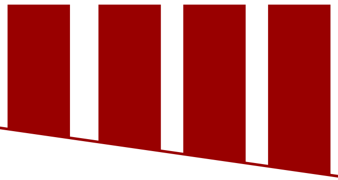
= Divide all work up front and then start running

Load balancing

Serial



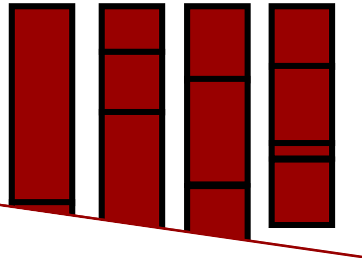
Parallel:
the dream



Embarrasingly paral
4x speed-up

All threads finish at
the same time

Parallel:
the reality



Will be in exercises

Problem:
Work is unevenly distributed

Solution
Use **dynamic scheduling** and
with
objects!

Typical for **static scheduling**
= Divide all work up front and then
start running

Amdahl's law

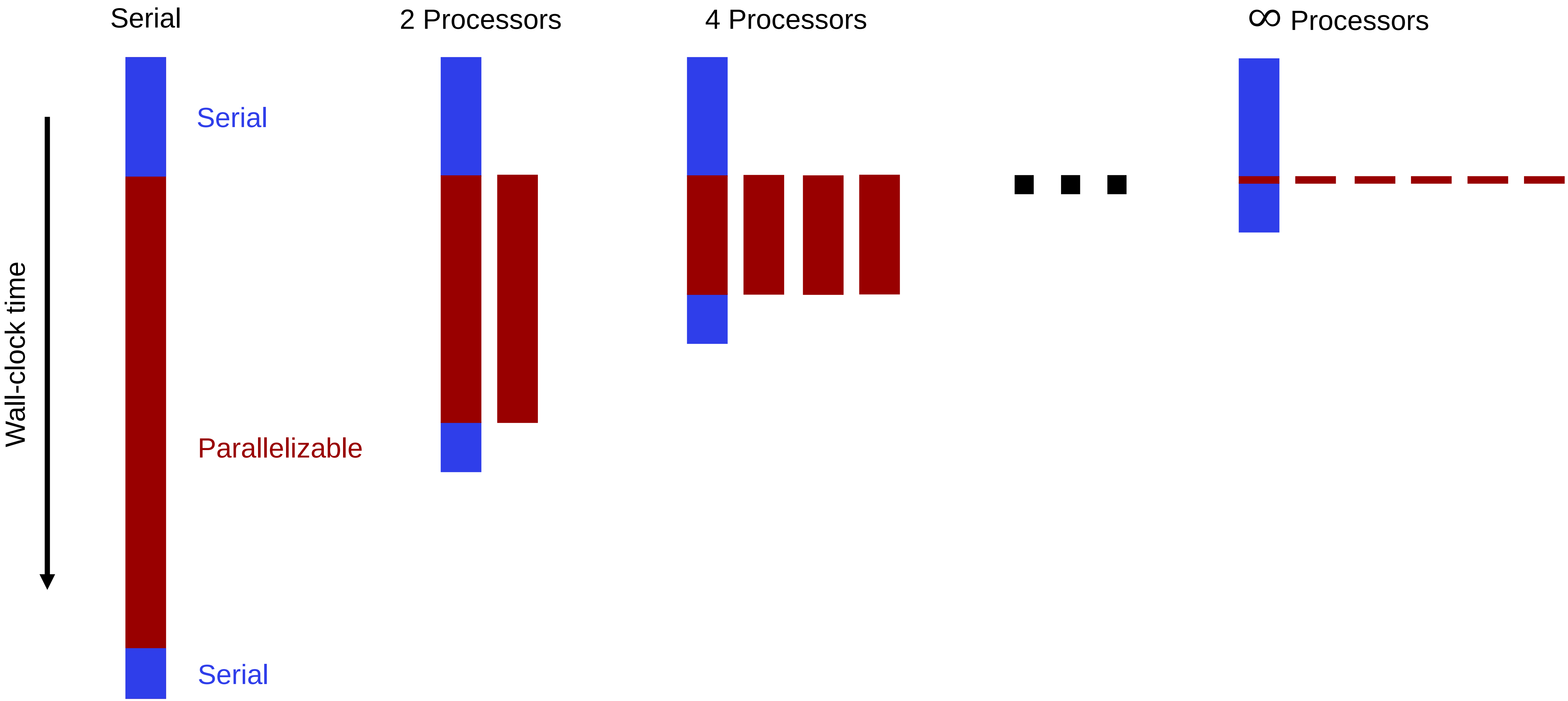
```python
import numpy as np
from multiprocessing.pool import ThreadPool

arr = np.zeros((1024, 1024, 1024))
arr_x10 = [arr, arr, arr, arr, arr,
           arr, arr, arr, arr, arr]

with ThreadPool(4) as pool:
    pool.map(np.sum, arr_x10)
```

Setup data Serial

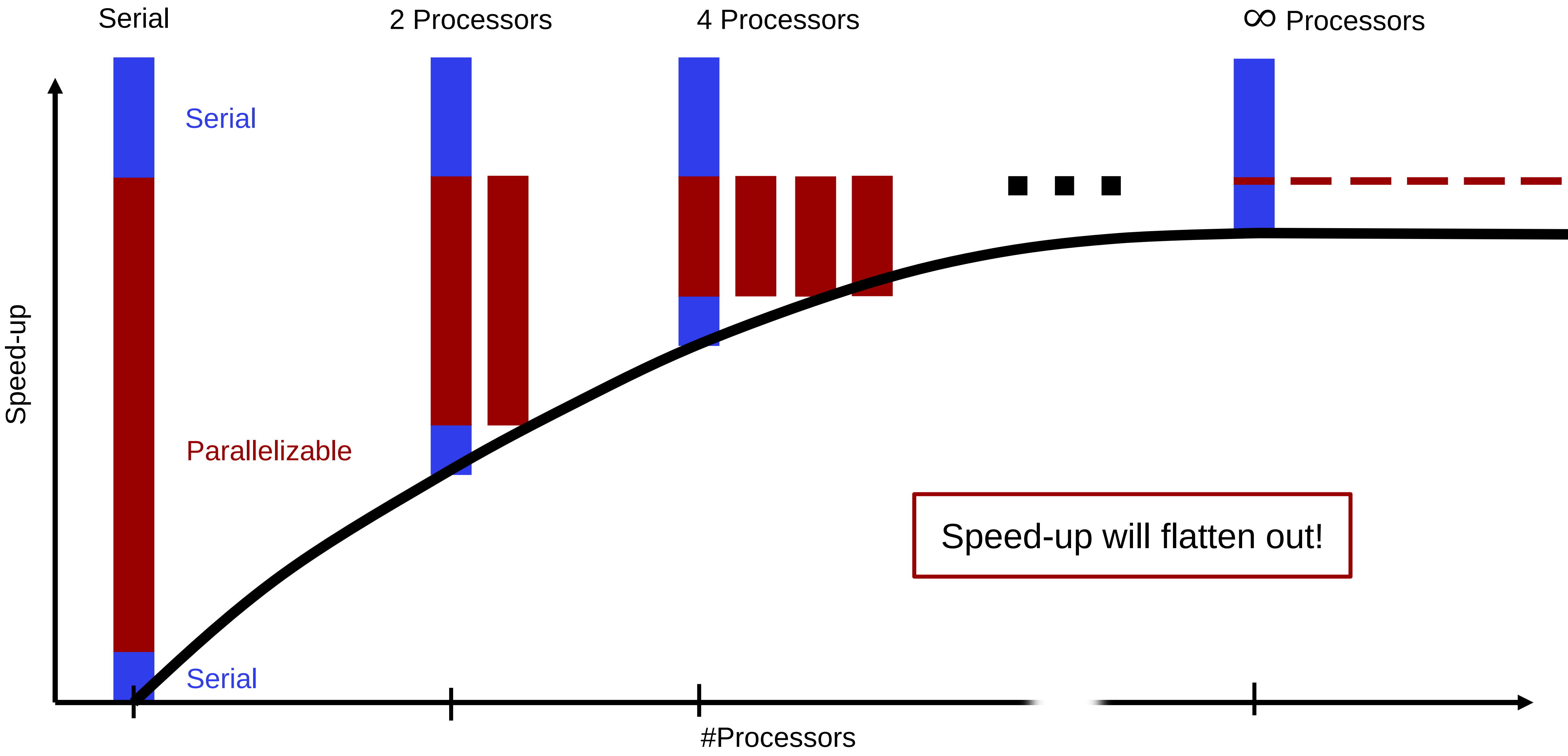
Compute Parallel

sums.py

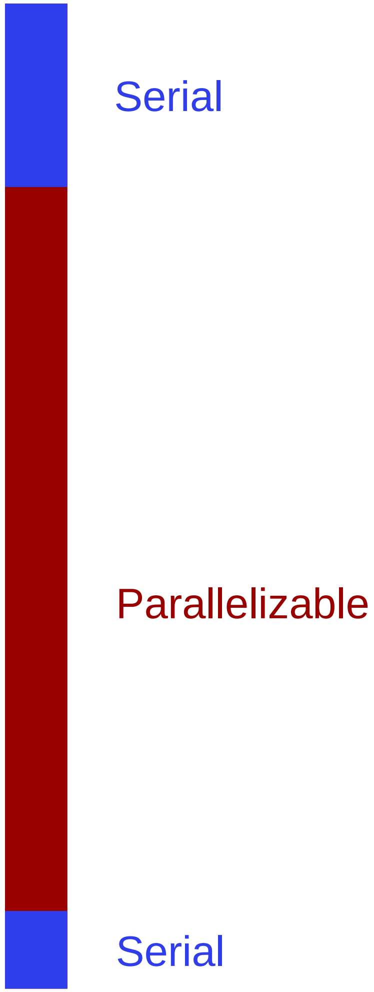
Amdahl's law


Serial
2 Processors
4 Processors
∞ Processors
Wall-clock time
Serial
Parallelizable
Serial

Amdahl’s law


Serial
2 Processors
4 Processors
∞ Processors
Serial
Parallelizable
Serial
Speed-up
#Processors
Speed-up will flatten out!

Amdahl’s law



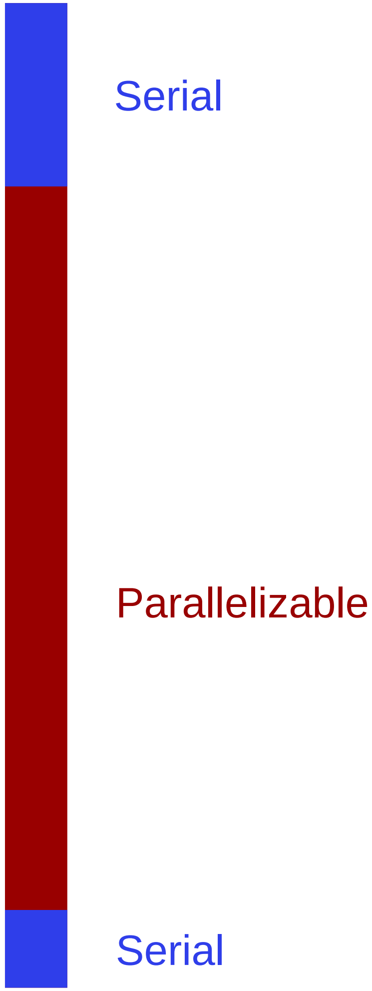
Assume our program has a ‘parallel fraction’ F

Serial execution time: $T(1) = (1 - F) * T(1) + F * T(1)$

Time on p processors: $T(p) = (1 - F) * T(1) + \frac{F}{p} * T(1)$

Speed-up: $S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{(1-F)*T(1) + \frac{F}{p}*T(1)}$

Amdahl’s law



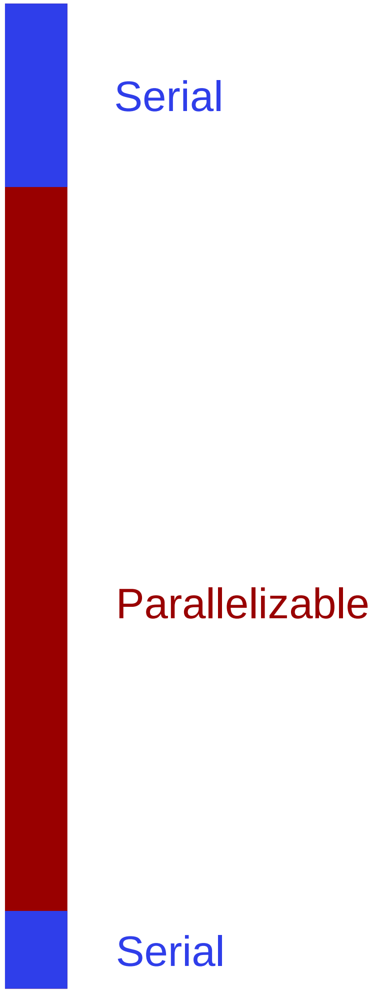
Assume our program has a ‘parallel fraction’ F

Serial execution time: $$T(1) = (1 - F) * T(1) + F * T(1)$$

Time on p processors: $$T(p) = (1 - F) * T(1) + \frac{F}{p} * T(1)$$

Amdahl’s law: $$S(p) = \frac{T(1)}{T(p)} = \frac{1}{(1-F) + \frac{F}{p}}$$

Amdahl's law



Assume our program has a 'parallel fraction' F

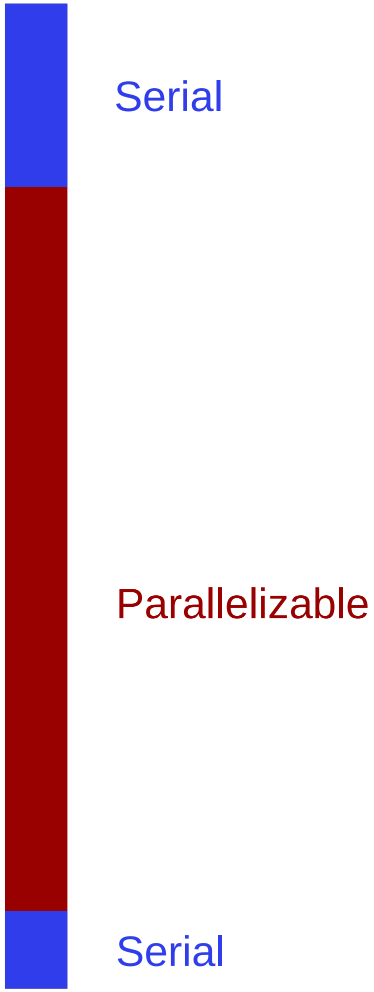
Serial execution time: $$T(1) = (1 - F) * T(1) + F * T(1)$$

Time on p processors: $$T(p) = (1 - F) * T(1) + \frac{F}{p} * T(1)$$

Amdahl's law: $$S(p) = \frac{T(1)}{T(p)} = \frac{1}{(1-F) + \frac{F}{p}}$$

For 'serial fraction' $B = 1 - F$: $$= \frac{1}{B + \frac{1 - B}{p}}$$

Amdahl's law



Assume our program has a 'parallel fraction' F

Serial execution time: $$T(1) = (1 - F) * T(1) + F * T(1)$$

Time on p processors: $$T(p) = (1 - F) * T(1) + \frac{F}{p} * T(1)$$

Amdahl's law: $$S(p) = \frac{T(1)}{T(p)} = \frac{1}{(1-F) + \frac{F}{p}}$$

Limit: $$S(\infty) = \frac{T(1)}{T(\infty)} = \frac{1}{1-F} = \frac{1}{B}$$

Amdahl's law

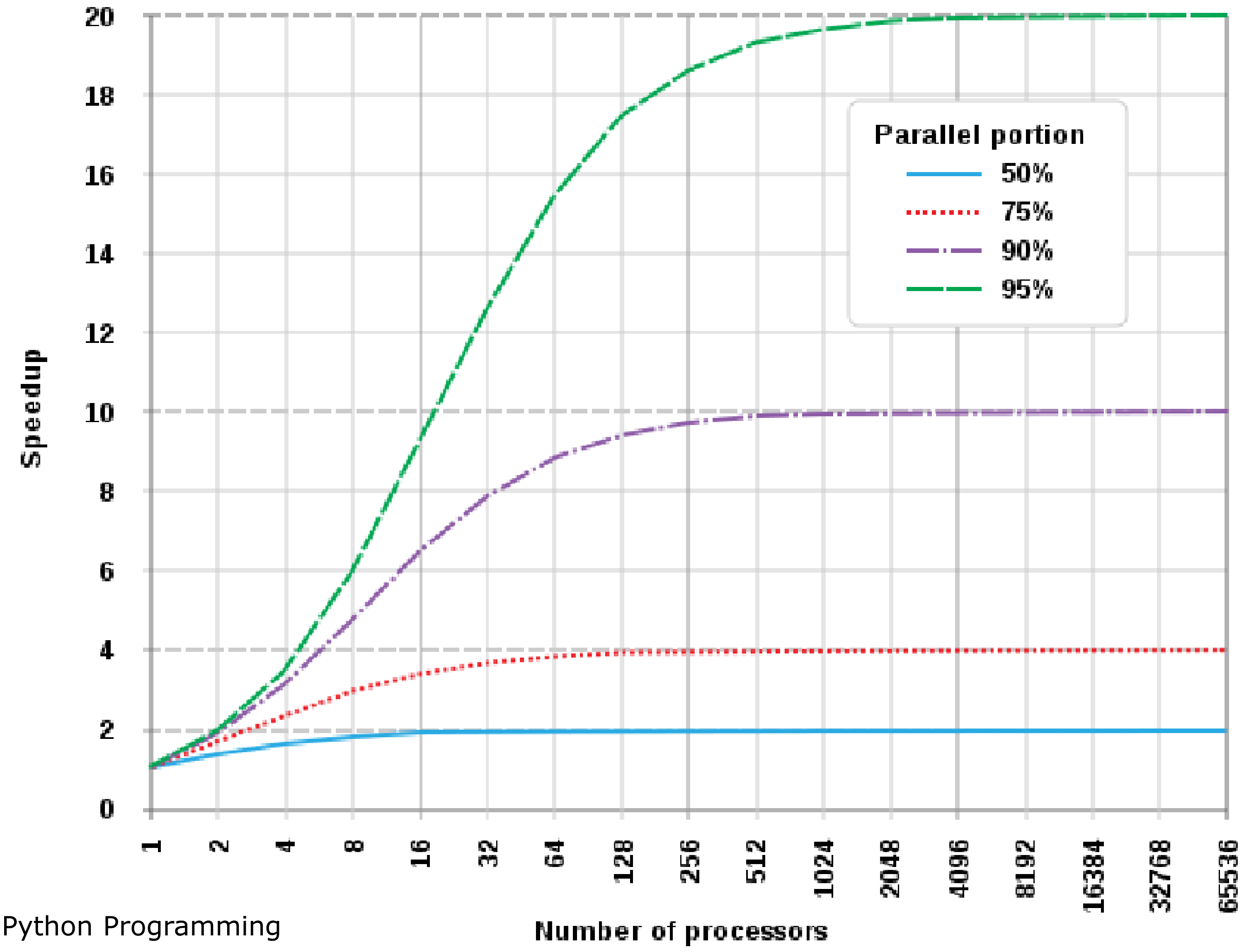


Figure from Advanced Python Programming

Amdahl's law

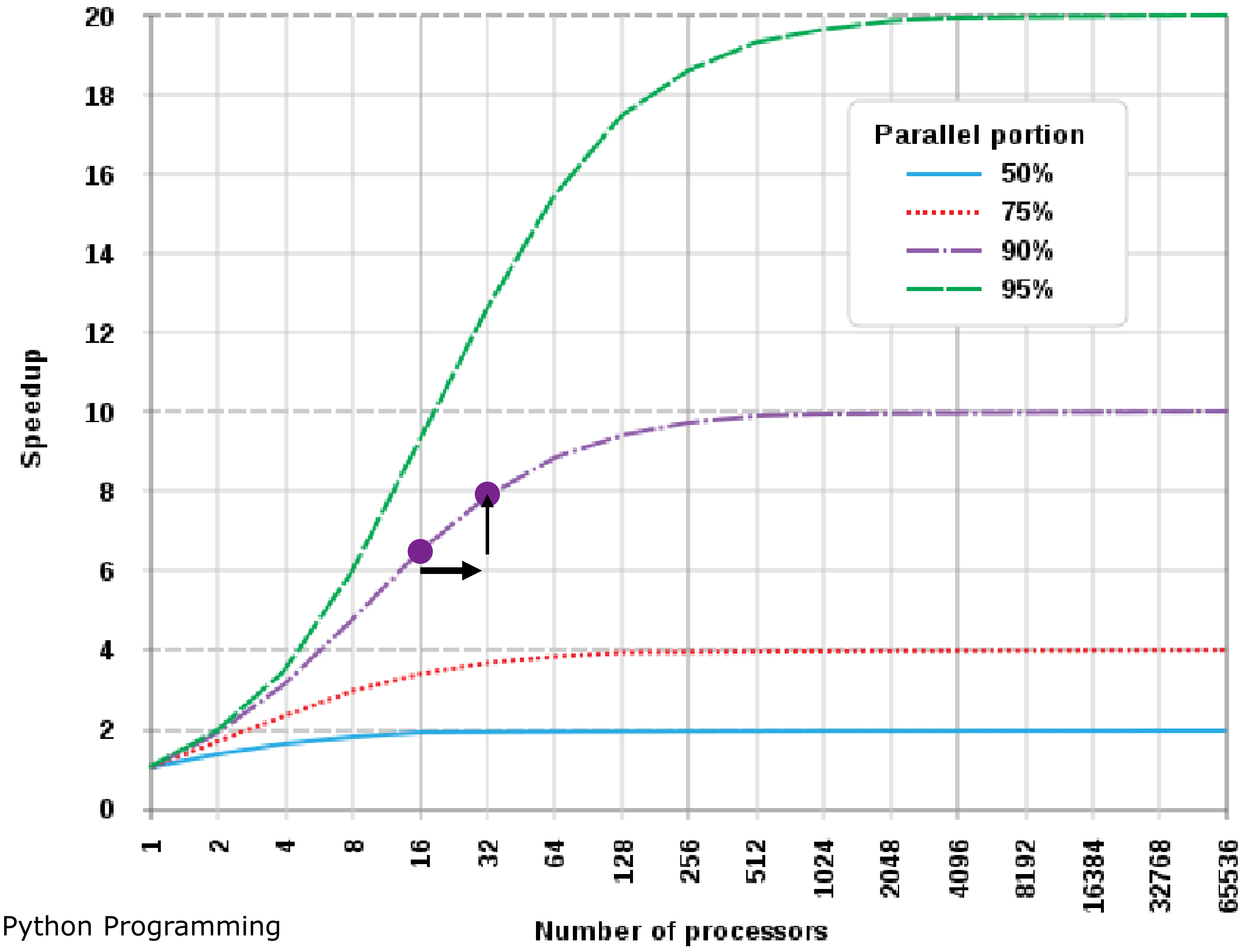


Figure from Advanced Python Programming

Amdahl's law

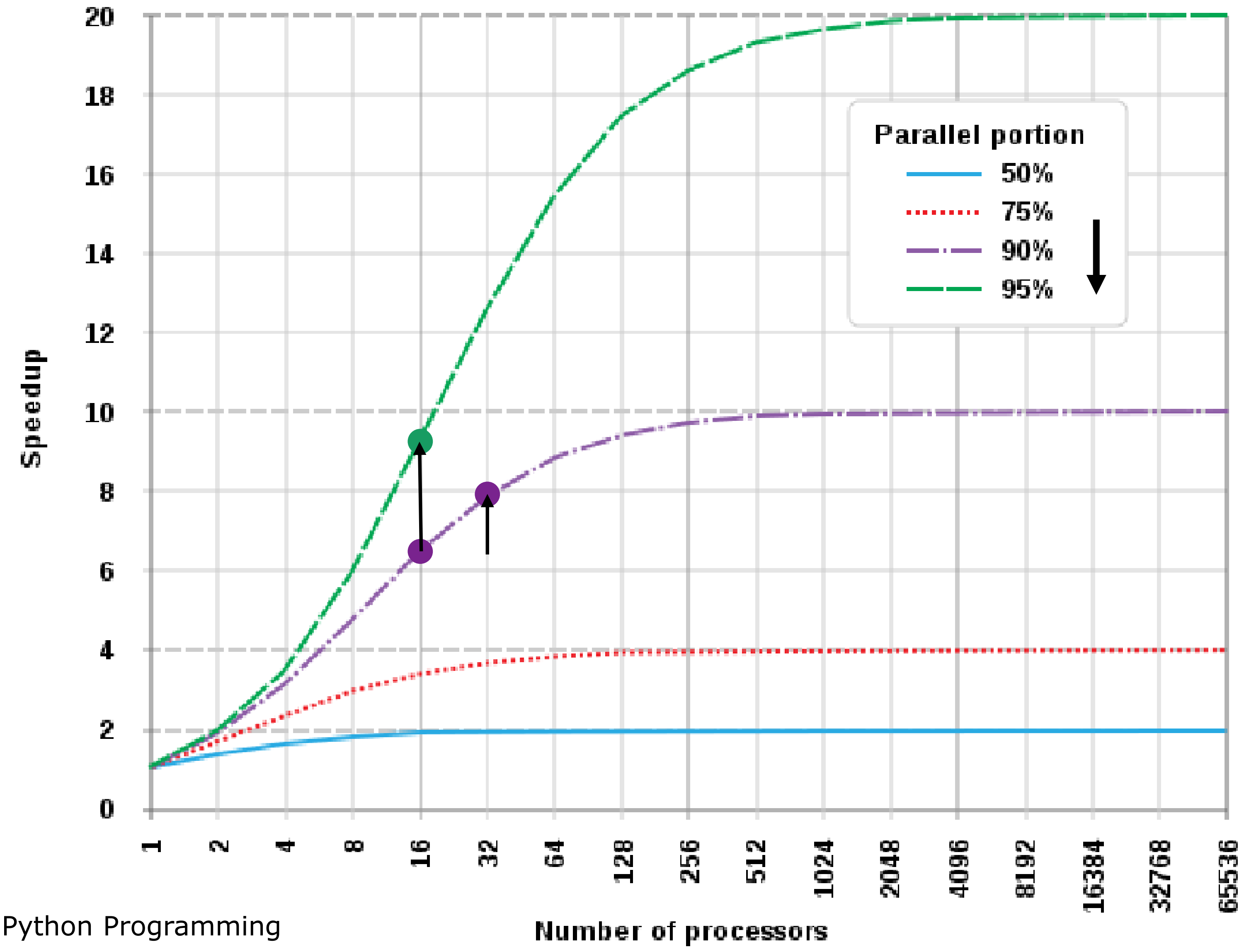


Figure from Advanced Python Programming

Amdahl's law

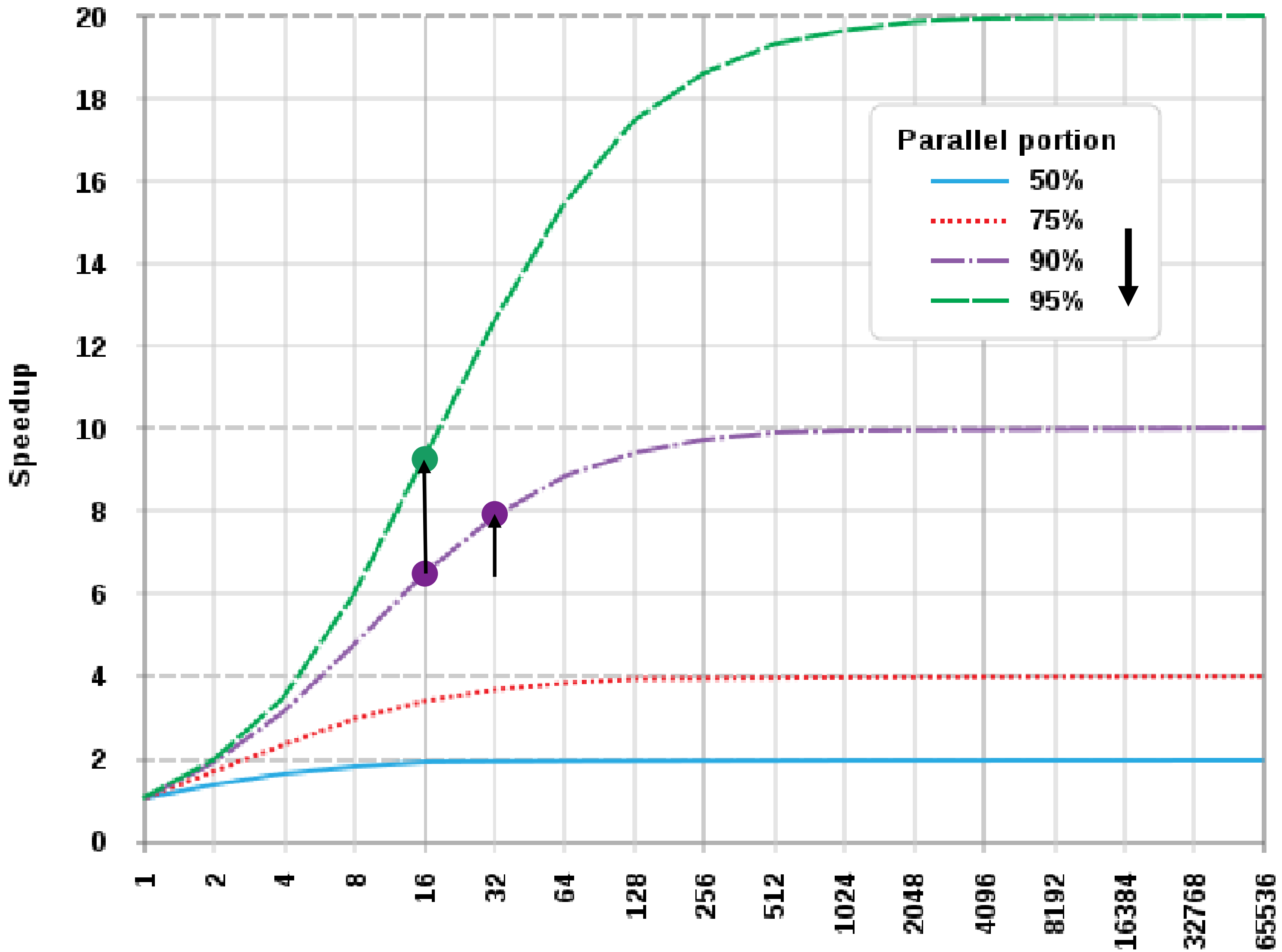


Figure from Advanced Python Programming

$$$$$

Quiz time!

menti.com

Mentimeter

Takeaways

- Parallelization is the name of the game now!
- Multi-threading *can* work, but multi-processing is the general way
- Efficient parallel programs are not trivial – no free lunch!
- Amdahl’s law tell us the limits

Today's exercise

Run stuff in parallel and efficiently

- Play with multiprocessing
- Play with chunking
- Play with scheduling
- Play with Amdahl’s law
- **NOTE:** These exercises are also for next week.

Useful concepts

Amdahl’s law

Parallel fraction F. Serial fraction $B = 1 - F$

$$S(p) = \frac{1}{(1-F) + \frac{F}{p}} = \frac{1}{B + \frac{1-B}{p}}$$

$$S(\infty) = \frac{1}{1-F} = \frac{1}{B}$$

Python Multiprocessing Pool

```
from multiprocessing.pool import Pool
pool = Pool(n_processes)

for part in data:
    pool.apply_async(function, part)

pool.map(function, data)
```

Change to work node

linuxsh

Submit job script

bsub < submit.sh

Job status

bstat / bjobs

Check job output

bpeek / bpeek <JOBID>

Kill job

bkill <JOBID>

Time command for Python script

time python script.py