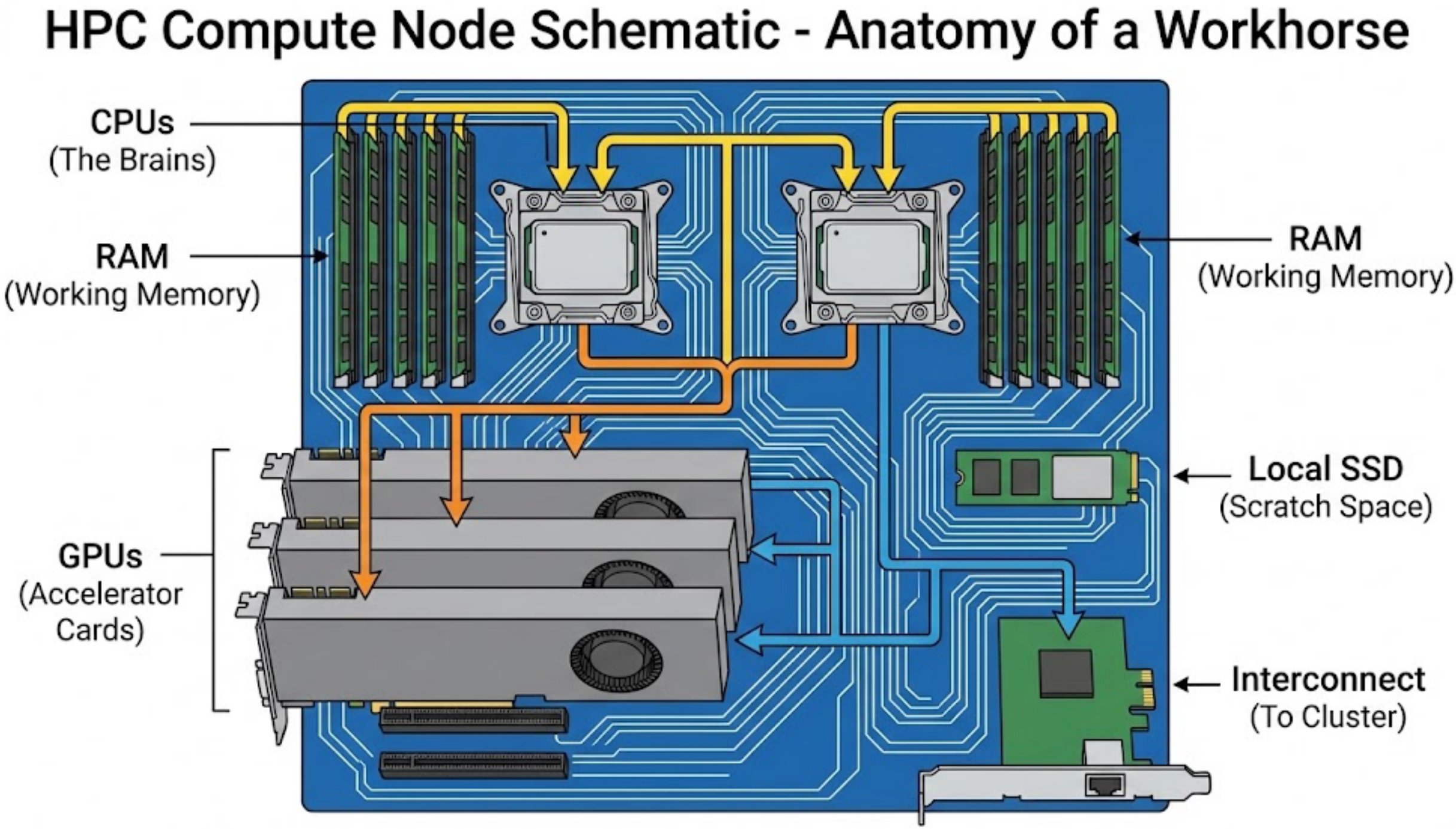
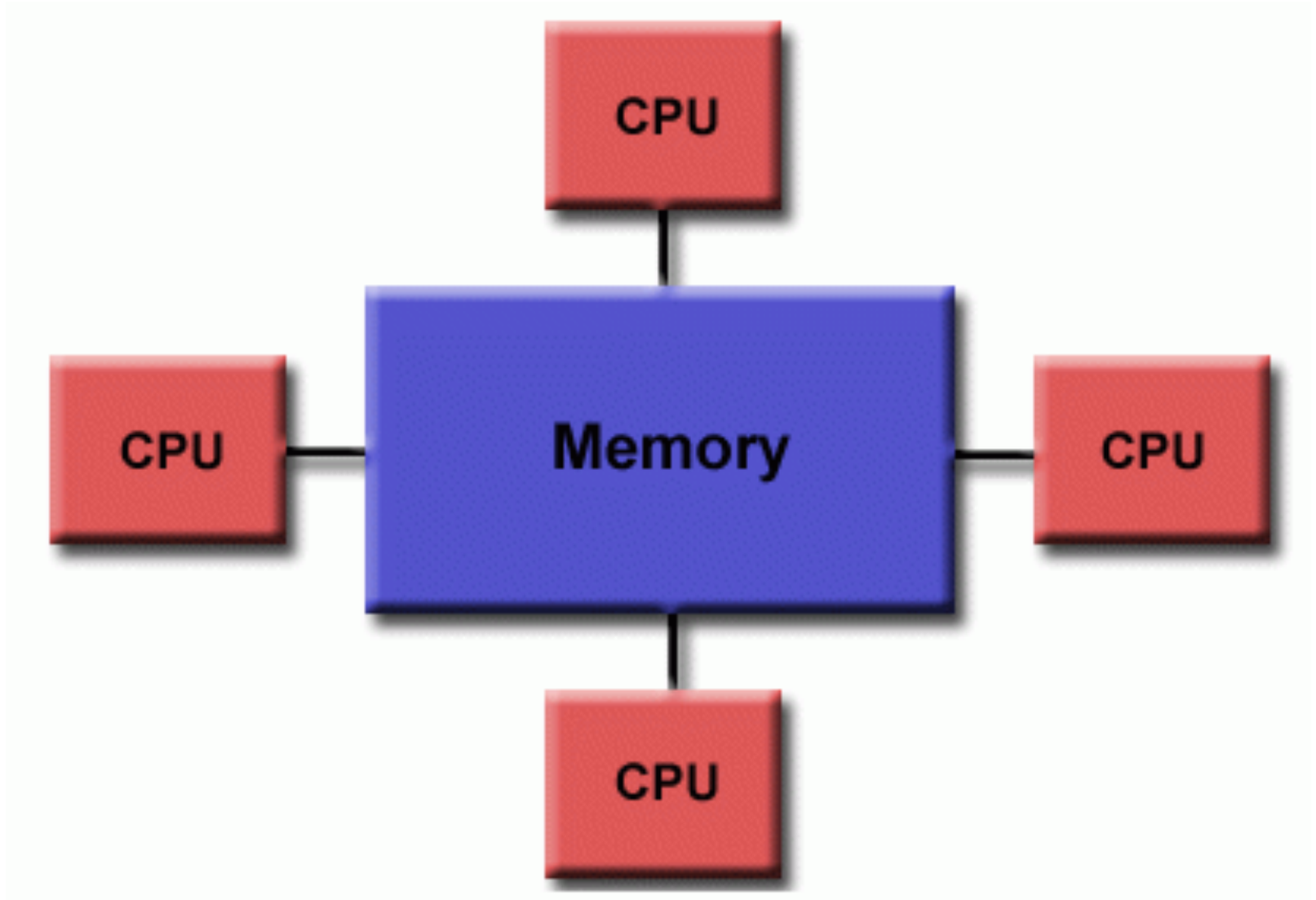
Lecture 10: Shared memory parallelism I

CEE 690

Back to our node schematic



Shared memory: All CPUs see the same bucket



How would you leverage this architecture when developing software?

Baseline

```python
import time
import numpy as np

def calculate_mean(data):
    val = 0
    count = 0
    for i in range(data.shape[0]):
        for j in range(data.shape[1]):
            val += data[i,j]
            count += 1
    return val/count

#Create data
niter = 2
np.random.seed(1)
data = np.random.randn(5000,5000)

start_p = time.time()
for i in range(niter):
    res_p = calculate_mean(data)
time_p = (time.time() - start_p)/niter
print(f"Result:    {res_p} | Time: {time_p:.4f}s")
result = calculate_mean(data)
```

```
(cee690) nc153 $ srun -n 1 python serial.py
Result:    0.00012655884641268577 | Time: 5.2289s
```

Python threads

```python
import threading

def calculate_mean_thread(data, imin, imax, results, index):
    val = 0
    count = 0
    for i in range(imin,imax):
        for j in range(data.shape[1]):
            val += data[i,j]
            count += 1
    results[index] = (val, count)

# Create data
np.random.seed(1)
data = np.random.randn(4000, 4000)

n_threads = 4
chunk = data.shape[0] // n_threads
results = [None] * n_threads
threads = []

start = time.time()
for i in range(n_threads):
    imin = i*chunk
    imax = (i+1)*chunk
    t = threading.Thread(target=calculate_mean_thread,
                         args=(data, imin, imax, results, i))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

```
(cee690) nc153 $ srun -n 1 python threads.py
4 threads
Result:    0.00012655884641268866 | Time: 4.8178s
```

Python threads

```
import threading

def calculate_mean_thread(data, imin, imax, results, index):
    val = 0
    count = 0
    for i in range(imin,imax):
        for j in range(data.shape[1]):
            val += data[i,j]
            count += 1
    results[index] = (val, count)

# Cre
np.ra
data

n_thr
chunk = data.shape[0] // n_threads
results = [None] * n_threads
threads = []

start = time.time()
for i in range(n_threads):
    imin = i*chunk
    imax = (i+1)*chunk
    t = threading.Thread(target=calculate_mean_thread,
                         args=(data, imin, imax, results, i))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

But why didn't I get ~4x speed-up?

```
(cee690) nc153 $ srun -n 1 python threads.py
4 threads
Result:    0.00012655884641268866 | Time: 4.8178s
```

Python threads (flip i,j)

```python
import threading

def calculate_mean_thread(data, imin, imax, results, index):
    val = 0
    count = 0
    for i in range(imin,imax):
        for j in range(data.shape[1]):
            val += data[j,i]
            count += 1
    results[index] = (val, count)

# Create data
np.random.seed(1)
data = np.random.randn(4000, 4000)

n_threads = 4
chunk = data.shape[0] // n_threads
results = [None] * n_threads
threads = []

start = time.time()
for i in range(n_threads):
    imin = i*chunk
    imax = (i+1)*chunk
    t = threading.Thread(target=calculate_mean_thread,
                         args=(data, imin, imax, results, i))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

- How you read your data from memory matters…

```
(cee690) nc153 $ srun -n 1 python threads.py
4 threads
Result:    0.00012655884641267235 | Time: 6.4842s
```

Global Interpreter Lock (GIL)

1. **Single thread execution** - The GIL turns your Python script into a hot potato game
2. **Memory safety** - Meant to ensure that not two threads are writing to the same place in memory
3. **I/O loophole** - The GIL is released when performing I/O operations.
4. **Math bottleneck** - The threads spend their time fighting over who gets to do their part.
5. **C-extension freedom** - Libraries like Numpy and Numba can release the GIL when compiled.


Numba

Numba (with the GIL)

```python
@numba.jit(nopython=False, parallel=True, forceobj=True)
def calculate_mean(data):
    val = 0
    count = 0
    for i in numba.prange(data.shape[0]):
        for j in numba.prange(data.shape[1]):
            val += data[i,j]
            count += 1
    return val/count

#Create data
np.random.seed(1)
data = np.random.randn(4000,4000)

# 1. The compilation step
calculate_mean(data)

# 2. Define the number of threads
numba.set_num_threads(4)
print(f"Using {numba.get_num_threads()}",flush=True)
start_time = time.time()

# 3. The execution
final_result = calculate_mean(data)
```

```
(cee690) nc153 $ srun -n 1 python numba_python_true.py
Using 4
Result: 0.00012655884641268 57
Time: 7.73373294 seconds
```

Numba (without the GIL)

```python
@numba.jit(nopython=True, parallel=True)
def calculate_mean(data):
    val = 0
    count = 0
    for i in numba.prange(data.shape[0]):
        for j in numba.prange(data.shape[1]):
            val += data[i,j]
            count += 1
    return val/count

#Create data
np.random.seed(1)
data = np.random.randn(4000,4000)

# 1. The compilation step
calculate_mean(data)

# 2. Define the number of threads
numba.set_num_threads(4)
print(f"Using {numba.get_num_threads()}",flush=True)
start_time = time.time()

# 3. The execution
final_result = calculate_mean(data)

duration = time.time() - start_time
```

```
Running with 4 threads
Result:    0.0001265588464126886 | Time: 0.0087s
```

Numba (without the GIL)

```python
@numba.jit(nopython=True, parallel=True)
def calculate_mean(data):
    val = 0
    count = 0
    for i in numba.prange(data.shape[0]):
        for j in numba.prange(data.shape[1]):
            val += data[i,j]
            count += 1
    return val/count
```

But how much of that is simply because we used the “Just in time compiler”?

```python
numba.set_num_threads(4)
print(f"Using {numba.get_num_threads()}",flush=True)
start_time = time.time()

# 3. The execution
final_result = calculate_mean(data)

duration = time.time() - start_time
```

```
Running with 4 threads
Result:    0.0001265588464126886 | Time: 0.0087s
```

Numba (number of threads)

```
(cee690) nc153 $ srun -n 1 python numba_python_false.py
Running with 1 threads
Result:    0.00012655884641268573 | Time: 0.0249s
Running with 2 threads
Result:    0.00012655884641269993 | Time: 0.0129s
Running with 4 threads
Result:    0.00012655884641268863 | Time: 0.0087s
Running with 8 threads
Result:    0.00012655884641269273 | Time: 0.0069s
Running with 16 threads
Result:    0.00012655884641270173 | Time: 0.0072s
```

The vast majority of the speed up is coming from the just-in-time compiling. Think about where you should prioritize your coding…

The results keep changing!

```
(cee690) nc153 $ srun -n 1 python numba_python_false.py
Running with 1 threads
Result:    0.00012655884641268857 | Time: 0.0249s
Running with 2 threads
Result:    0.00012655884641269999 | Time: 0.0129s
Running with 4 threads
Result:    0.00012655884641268886 | Time: 0.0087s
Running with 8 threads
Result:    0.00012655884641269273 | Time: 0.0069s
Running with 16 threads
Result:    0.00012655884641270173 | Time: 0.0072s
```

But why?

Floating point non-associativity

$$(a + b) + c \approx a + (b + c)$$

```
a = 1e16
b = -1e16
c = 1.0

print('A:',(a + b) + c)
print('B:',a + (b + c))
```

- The order of operations matters when adding floating points.
- When we parallelize, we are inherently changing the order of operations.
- To minimize, try to add up numbers of similar orders of magnitude first.

```
(cee690) nc153 $ python nonassociativity.py
A: 1.0
B: 0.0
```

The end result is that your software will give different results depending on how many threads you use.

Floating point non-associativity

$$(a + b) + c \approx a + (b + c)$$

- The order of operations matters when adding floating points.
- When we parallelize, we are inherently changing the order of magnitude first.

```
a = 1e16
```

What is working under the hood to do most of the threading Numba?

```
(cee690) nc153 $ python nonassociativity.py
A: 1.0
B: 0.0
```

The end result is that your software will give different results depending on how many threads you use.

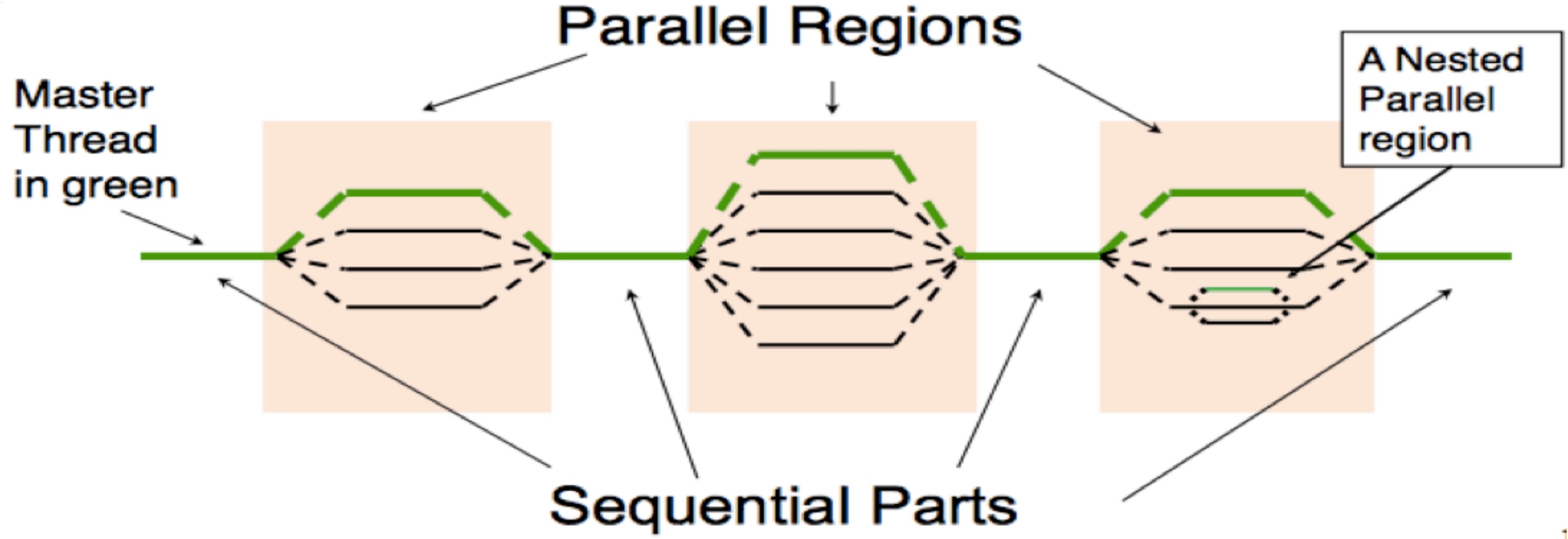
OpenMP®

Industry standard for shared memory parallelism

What is OpenMP?

- **“Directive” Philosophy** - It allows you to annotate existing serial code. Simplest path into parallelization
- **Fork/Join model—** Main thread forks into many threads when it hits a parallel region and then joins when complete
- **Shared vs private memory —** All the threads can share the same memory; however, there is a place for private variables for each thread as well
- **Reductions —** Operation to combine “thread/local sums” into total sums
- **Synchronization and race conditions —** Special care needs to be taken that two threads don’t write to the exact same memory location at once

OpenMP flow


Parallel Regions
Master
Thread
in green
A Nested
Parallel
region
Sequential Parts

PyOMP example

```python
import numpy as np
import numba
from numba.openmp import njit, omp_set_num_threads, omp_get_num_threads, openmp_context, omp_get_thread_num

@njit
def calculate_mean_pyomp(data):
    """
    Using the Numba-OpenMP extension syntax.
    Note: We do NOT use parallel=True in the decorator because
    the 'with openmp' block handles the parallelism explicitly.
    """
    val = 0.0
    count = 0
    rows, cols = data.shape

    # 1. Parallel Region (Spawn threads)
    with openmp_context("parallel shared(data, val, count)"):

        # Get rank and number of threads
        rank = omp_get_thread_num()
        size = omp_get_num_threads()

        # Every thread gets its own private local counters
        local_val = 0.0
        local_count = 0

        # 2. Work-Sharing Block (Calculate local sums)
        for i in range(rows):
            if i%size != rank:continue
            for j in range(cols):
                local_val += data[i, j]
                local_count += 1

        # 3. Separate Reduction Block (The merge)
        # 'critical' ensures only ONE thread at a time can run this block
        with openmp_context("critical"):
            val += local_val
            count += local_count

    return val / count
```

```
(cee690) nc153 $ srun -n 1 python pyomp_long.py
Running with 1 threads
Result:    0.00012655884641268864 | Time: 0.0243s
Running with 2 threads
Result:    0.00012655884641269994 | Time: 0.0121s
Running with 4 threads
Result:    0.00012655884641268864 | Time: 0.0062s
Running with 8 threads
Result:    0.00012655884641270994 | Time: 0.0034s
Running with 16 threads
Result:    0.00012655884641269634 | Time: 0.0043s
```

PyOMP directives

`with openmp("parallel"):`	Create a team of threads. Execute a parallel region
`with openmp("for"):`	Use inside a parallel region. Split up a loop across the team.
`with openmp("parallel for"):`	A combined construct. Same a **`parallel`** followed by a **`for`**.
`with openmp ("single"):`	One thread does the work. Others wait for it to finish
`with openmp("task"):`	Create an explicit task for work within the construct.
`with openmp("taskwait"):`	Wait for all tasks in the current task to complete.
`with openmp("barrier"):`	All threads arrive at a barrier before any proceed.
`with openmp("critical"):`	Mutual exclusion. One thread at a time executes code
`schedule(static [,chunk])`	Map blocks of loop iterations across the team. Use with **`for`**.
`reduction(op:list)`	Combine values with op across the team. Used with **`for`**
`private(list)`	Make a local copy of variables for each thread. Use with **`parallel`**, **`for`** or **`task`**.
`firstprivate(list)`	**`private`**, but initialize with original value. Use with **`parallel`**, **`for`** or **`task`**
`shared(list)`	Variables shared between threads. Use with **`parallel`**, **`for`** or **`task`**.
`default(none)`	Force definition of variables as **`private`** or **`shared`**.
`omp_get_num_threads()`	Return the number of threads in a team
`omp_get_thread_num()`	Return an ID from 0 to the number of threads minus one
`omp_set_num_threads(int)`	Set the number of threads to request for parallel regions
`omp_get_wtime()`	Return a snapshot of the wall clock time.

PyOMP example simplified

```python
import numba
from numba.openmp import njit, omp_set_num_threads, omp_get_num_threads, openmp_context

@njit
def calculate_mean_pyomp(data):
    """
    Using the Numba-OpenMP extension syntax.
    Note: We do NOT use parallel=True in the decorator because
    the 'with openmp' block handles the parallelism explicitly.
    """
    val = 0.0
    count = 0
    rows, cols = data.shape

    # 1. Parallel Region (Spawn threads)
    with openmp_context("parallel shared(data, val, count)"):

        # Every thread gets its own private local counters
        local_val = 0.0
        local_count = 0

        # 2. Work-Sharing Block (Calculate local sums)
        # We use 'private(j)' to ensure inner loop safety
        with openmp_context("for private(j)"):
            for i in range(rows):
                for j in range(cols):
                    local_val += data[i, j]
                    local_count += 1

        # 3. Separate Reduction Block (The merge)
        # 'critical' ensures only ONE thread at a time can run this block
        with openmp_context("critical"):
            val += local_val
            count += local_count

    return val / count
```

```
(cee690) nc153 $ srun -n 1 python pyomp_long.py
Running with 1 threads
Result:    0.00012655884641268864 | Time: 0.0243s
Running with 2 threads
Result:    0.00012655884641269999 | Time: 0.0121s
Running with 4 threads
Result:    0.00012655884641268864 | Time: 0.0062s
Running with 8 threads
Result:    0.00012655884641270994 | Time: 0.0034s
Running with 16 threads
Result:    0.00012655884641269634 | Time: 0.0043s
```

PyOMP example simplified

```python
import numba
from numba.openmp import njit, omp_set_num_threads, omp_get_num_threads, openmp_context

@njit
def calculate_mean_pyomp(data):
    """
    Using the Numba-OpenMP extension syntax.
    Note: We do NOT use parallel=True in the decorator because
    the 'with openmp' block handles the parallelism explicitly.
    """
    val = 0.0
    count = 0
    rows, cols = data.shape

    # 1. Parallel Region (Spawn threads)
    with openmp_context("parallel shared(data, val, count)"):

        # Every thread gets its own private local counters
        local_val = 0.0
        local_count = 0

        # 2. Work-Sharing Block (Calculate local sums)
        # We use 'private(j)' to ensure inner loop safety
        with openmp_context("for private(j)"):
            for i in range(rows):
                for j in range(cols):
                    local_val += data[i, j]
                    local_count += 1

        # 3. Separate Reduction Block (The merge)
        # 'critical' ensures only ONE thread at a time can run this block
        with openmp_context("critical"):
            val += local_val
            count += local_count

    return val / count
```

```
(cee690) nc153 $ srun -n 1 python pyomp_long.py
Running with 1 threads
Result:    0.00012655884641268864 | Time: 0.0062s
Running with 8 threads
Result:    0.00012655884641270994 | Time: 0.0034s
Running with 16 threads
Result:    0.00012655884641269634 | Time: 0.0043s
```

PyOMP automatically schedules/ assigns rows to threads

And simplified again

```python
import numba
from numba.openmp import njit, omp_set_num_threads, omp_get_num_threads, openmp_context

@njit
def calculate_mean_pyomp(data):
    """
    Using the Numba-OpenMP extension syntax.
    Note: We do NOT use parallel=True in the decorator because
    the 'with openmp' block handles the parallelism explicitly.
    """
    val = 0.0
    count = 0
    rows, cols = data.shape

    # 1. Parallel Region (Spawn threads)
    with openmp_context("parallel shared(data, val, count) reduction(+:val, count)"):

        # 2. Work-Sharing Block (Calculate local sums)
        # We use 'private(j)' to ensure inner loop safety
        with openmp_context("for private(j)"):
            for i in range(rows):
                for j in range(cols):
                    val += data[i, j]
                    count += 1

    return val / count
```

```
(cee690) nc153 $ srun -n 1 python pyomp_short.py
Running with 1 threads
Result:    0.00012655884641268857 | Time: 0.0290s
Running with 2 threads
Result:    0.00012655884641269999 | Time: 0.0154s
Running with 4 threads
Result:    0.00012655884641268886 | Time: 0.0081s
Running with 8 threads
Result:    0.00012655884641270997 | Time: 0.0050s
Running with 16 threads
Result:    0.00012655884641269634 | Time: 0.0052s
```

And again

```python
import numba
from numba.openmp import njit, omp_set_num_threads, omp_get_num_threads, openmp_context

@njit
def calculate_mean_pyomp(data):
    """
    Using the Numba-OpenMP extension syntax.
    Note: We do NOT use parallel=True in the decorator because
    the 'with openmp' block handles the parallelism explicitly.
    """
    val = 0.0
    count = 0
    rows, cols = data.shape

    # 1. Parallel Region (Spawn threads)
    with openmp_context("parallel for private(j) shared(data, val, count) reduction(+:val, count)"):
        for i in range(rows):
            for j in range(cols):
                val += data[i, j]
                count += 1

    return val / count
```

```
(cee690) nc153 $ srun -n 1 python pyomp_short.py
Running with 1 threads
Result:    -7.970853033679433e-05 | Time: 0.1187s
Running with 2 threads
Result:    -7.970853033679742e-05 | Time: 0.0718s
Running with 4 threads
Result:    -7.97085303367978e-05 | Time: 0.0353s
Running with 8 threads
Result:    -7.970853033680114e-05 | Time: 0.0194s
Running with 16 threads
Result:    -7.970853033679374e-05 | Time: 0.0233s
```

Back to our most simple case

```python
import numba
from numba.openmp import njit, omp_set_num_threads, omp_get_num_threads, openmp_context

@njit
def calculate_mean_pyomp(data):
    """
    Using the Numba-OpenMP extension syntax.
    Note: We do NOT use parallel=True in the decorator because
    the 'with openmp' block handles the parallelism explicitly.
    """
    val = 0.0
    count = 0
    rows, cols = data.shape

    # 1. Parallel Region (Spawn threads)
    with openmp_context("parallel shared(data, val, count) reduction(+:val, count)"):

        # 2. Work-Sharing Block (Calculate local sums)
        # We use 'private(j)' to ensure inner loop safety
        with openmp_context("for private(j)"):
            for i in range(rows):
                for j in range(cols):
                    val += data[i, j]
                    count += 1

    return val / count
```

```
(cee690) nc153 $ srun -n 1 python pyomp_short.py
Running with 1 threads
Result:    0.00012655884641268857 | Time: 0.0290s
Running with 2 threads
Result:    0.00012655884641269999 | Time: 0.0154s
Running with 4 threads
Result:    0.00012655884641268886 | Time: 0.0081s
Running with 8 threads
Result:    0.00012655884641270997 | Time: 0.0050s
Running with 16 threads
Result:    0.00012655884641269634 | Time: 0.0052s
```

Back to our most simple case

```python
import numba
from numba.openmp import njit, omp_set_num_threads, omp_get_num_threads, openmp_context

@njit
def calculate_mean_pyomp(data):
    """
    Using the Numba-OpenMP extension syntax.
    Note: We do NOT use parallel=True in the decorator because
    the 'with openmp' block handles the parallelism explicitly.
    """
```

This just starts to look like Numba…

```python
                for j in range(cols):
                    val += data[i, j]
                    count += 1

    return val / count
```

```
(cee690) nc153 $ srun -n 1 python pyomp_short.py
Running with 1 threads
Result:    0.00012655884641268857 | Time: 0.0290s
Running with 2 threads
Result:    0.00012655884641269999 | Time: 0.0154s
Running with 4 threads
Result:    0.00012655884641268886 | Time: 0.0081s
Running with 8 threads
Result:    0.00012655884641270997 | Time: 0.0050s
Running with 16 threads
Result:    0.00012655884641269634 | Time: 0.0052s
```

Detour: Race conditions

```python
import numpy as np
from numba.openmp import njit, openmp_context, omp_set_num_threads

@njit
def trigger_race_condition(iterations):
    # This is our shared variable
    counter = 0

    # We spawn threads but provide NO protection (no reduction, no critical)
    with openmp_context("parallel shared(counter)"):
        with openmp_context("for"):
            for i in range(iterations):
                counter += 1

    return counter

# Setup
omp_set_num_threads(16)
iters = 1_000_000
expected = iters * 4

# Run the experiment
result = trigger_race_condition(iters)

print(f"Expected Result: {expected}")
print(f"Actual Result:   {result}")
print(f"Lost Increments: {expected - result}")
```

```
(cee690) nc153 $ srun -n 1 python pyomp_race_condition.py
Expected Result: 4000000
Actual Result:   812500
Lost Increments: 3187500
```

Detour: Race conditions

```python
import numpy as np
from numba.openmp import njit, openmp_context, omp_set_num_threads

@njit
def trigger_race_condition(iterations):
    # This is our shared variable
    counter = 0

    # We spawn threads but provide NO protection (no reduction, no critical)
    with openmp_context("parallel shared(counter)"):
```

If are going to use “raw” OpenMP then you need to know what you are doing.

```python
# Run the experiment
result = trigger_race_condition(iters)

print(f"Expected Result: {expected}")
print(f"Actual Result:   {result}")
print(f"Lost Increments: {expected - result}")
```

```
(cee690) nc153 $ srun -n 1 python pyomp_race_condition.py
Expected Result: 4000000
Actual Result:   812500
Lost Increments: 3187500
```

Know your directives…

with openmp("parallel"):	Create a team of threads. Execute a parallel region
with openmp("for"):	Use inside a parallel region. Split up a loop across the team.
with openmp("parallel for"):	A combined construct. Same a **parallel** followed by a **for**.
with openmp ("single"):	One thread does the work. Others wait for it to finish
with openmp("task"):	Create an explicit task for work within the construct.
with openmp("taskwait"):	Wait for all tasks in the current task to complete.
with openmp("barrier"):	All threads arrive at a barrier before any proceed.
with openmp("critical"):	Mutual exclusion. One thread at a time executes code
schedule(static [,chunk])	Map blocks of loop iterations across the team. Use with **for**.
reduction(op:list)	Combine values with op across the team. Used with **for**
private(list)	Make a local copy of variables for each thread. Use with **parallel**, **for** or **task**.
firstprivate(list)	**private**, but initialize with original value. Use with **parallel**, **for** or **task**
shared(list)	Variables shared between threads. Use with **parallel**, **for** or **task**.
default(none)	Force definition of variables as **private** or **shared**.
omp_get_num_threads()	Return the number of threads in a team
omp_get_thread_num()	Return an ID from 0 to the number of threads minus one
omp_set_num_threads(int)	Set the number of threads to request for parallel regions
omp_get_wtime()	Return a snapshot of the wall clock time.

Unless you are feeling adventurous…
just use Numba

```python
@numba.jit(nopython=True, parallel=True)
def calculate_mean(data):
    val = 0
    count = 0
    for i in numba.prange(data.shape[0]):
        for j in numba.prange(data.shape[1]):
            val += data[i,j]
            count += 1
    return val/count

#Create data
np.random.seed(1)
data = np.random.randn(4000,4000)

# 1. The compilation step
calculate_mean(data)

# 2. Define the number of threads
numba.set_num_threads(4)
print(f"Using {numba.get_num_threads()}",flush=True)
start_time = time.time()

# 3. The execution
final_result = calculate_mean(data)

duration = time.time() - start_time
```

Threading in Numpy?

```python
import os
import time
# Set these BEFORE importing numpy
nthreads = 16
os.environ["OMP_NUM_THREADS"] = f"{nthreads}"
os.environ["MKL_NUM_THREADS"] = f"{nthreads}"
os.environ["OPENBLAS_NUM_THREADS"] = f"{nthreads}"

import numpy as np

#Create data
np.random.seed(1)
A = np.random.randn(5000,5000)
B = np.random.randn(5000,5000)

start_time = time.time()

# 3. The execution
final_result = np.dot(A,B)

duration = time.time() - start_time

print(f"{nthreads} threads")
print(f"Time: {duration:.8f} seconds")
```

```
(cee690) nc153 $ srun -n 1 python numpy_threads.py
1 threads
Time: 2.23773479 seconds
(cee690) nc153 $ srun -n 1 python numpy_threads.py
2 threads
Time: 1.10966587 seconds
(cee690) nc153 $ srun -n 1 python numpy_threads.py
4 threads
Time: 0.56612849 seconds
(cee690) nc153 $ srun -n 1 python numpy_threads.py
8 threads
Time: 0.40759492 seconds
(cee690) nc153 $ srun -n 1 python numpy_threads.py
16 threads
Time: 0.50479555 seconds
```

Numpy will by default use the number of threads available (which you can also specify). Since it is compiled in C, it can release the GIL