MR2020: Coding for METOC

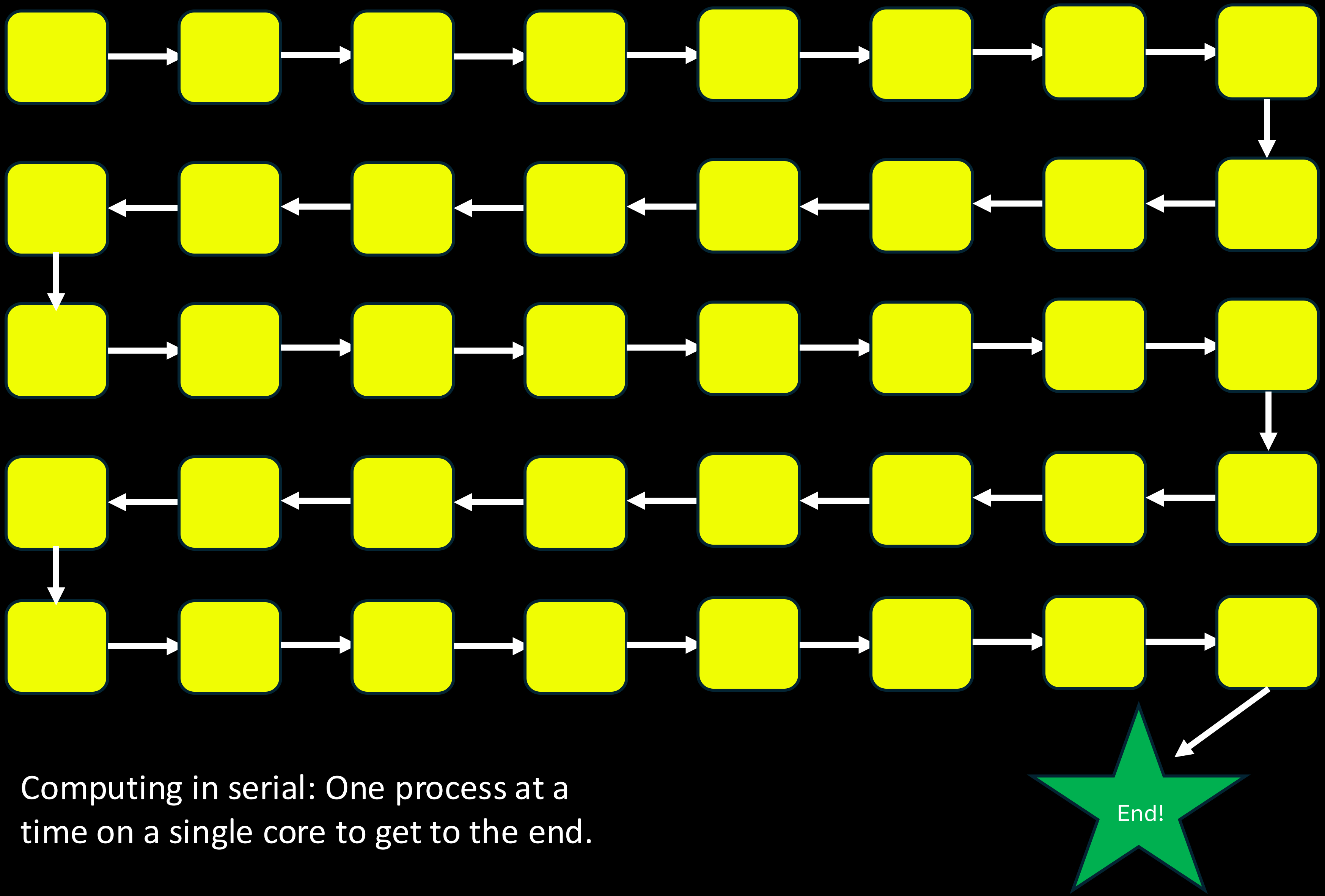
Module 10: Parallel Computing

Why execute code in parallel?

Modern CPUs and GPUs contain multiple cores, meaning that a single processing unit can execute multiple processes simultaneously.

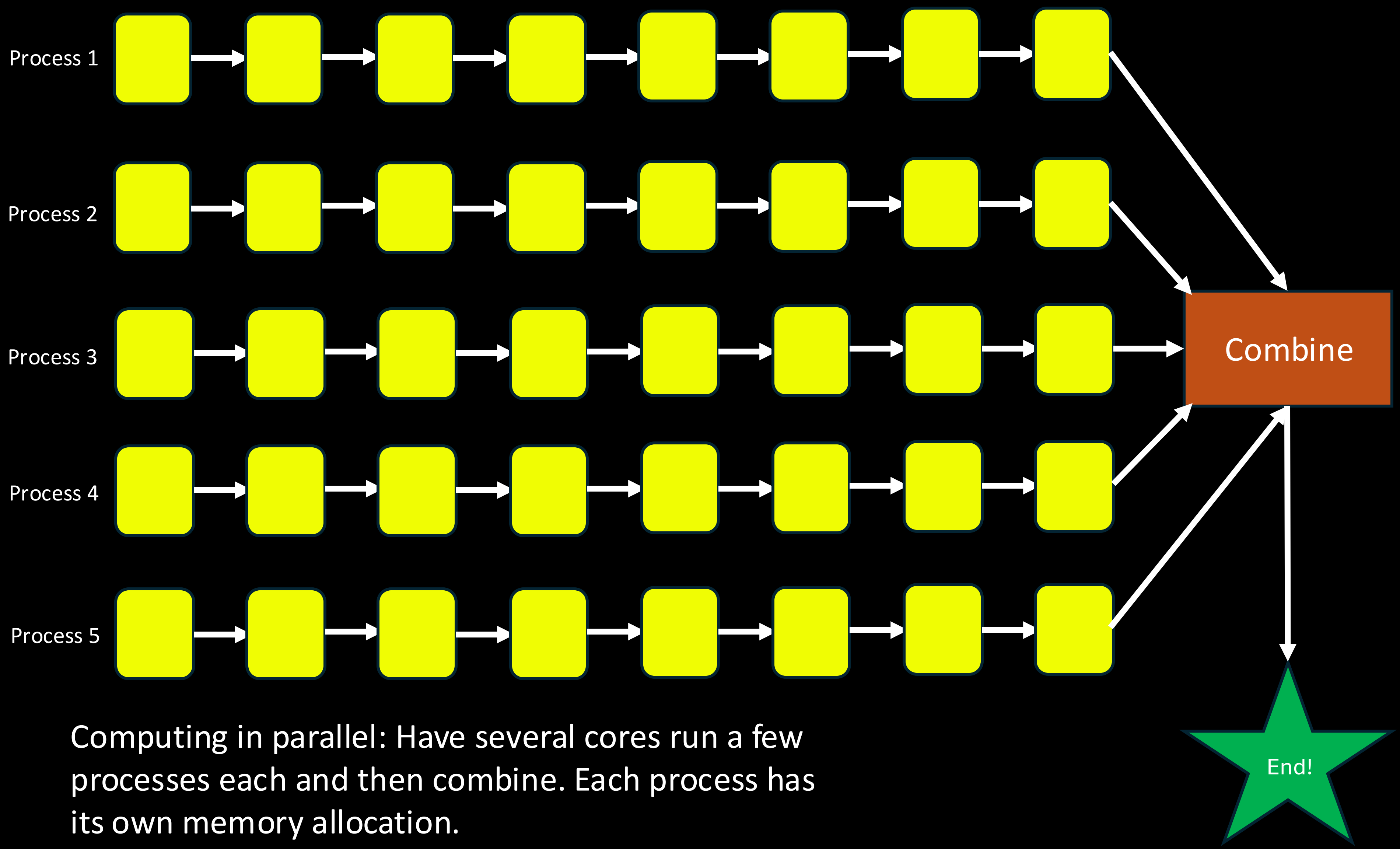
For operations in your code that are repeated *and are independent of one another*, we can use Python’s native libraries for running several operations at the same time instead of one after another. For intensive jobs, this can result in significant speed ups.

Speeding up repeated tasks



Computing in serial: One process at a time on a single core to get to the end.

Speeding up repeated tasks



Computing in parallel: Have several cores run a few processes each and then combine. Each process has its own memory allocation.

Methods in Multiprocessing

Process: Represents an individual process. These can be manually started. Can be started, joined, and terminated.

Pool: Creates and manages a pool of worker processes. Simplifies parallel execution with methods like map, apply, starmap. Does not require starting individual processes.

Queue: Allows processes to communicate by sending and receiving messages (First In, First Out; i.e., FIFO).

Pipe: Establishes a two-way communication channel between processes.

Manager: Enables sharing data (e.g., lists, dictionaries) between processes.

Common Start Methods

fork: (Default on Unix) Child process is a copy of the parent.

spawn: (Default on Windows & macOS) Starts with a fresh Python interpreter.

Processes vs. Threads

Definition

•Processes:

- Independent execution units with their own memory space.
- Created using multiprocessing in Python.
- Suitable for CPU-bound tasks.

•Threads:

- Lightweight units of execution within a process, sharing the same memory space.
- Created using threading in Python.
- Suitable for I/O-bound tasks.

2. Memory & Resources

•Processes:

- **Isolated memory**: Each process has its own memory space.
- **More overhead**: Due to isolation, processes are heavier on system resources.

•Threads:

- **Shared memory**: All threads in a process share the same memory.
- **Less overhead**: Threads are lighter and more efficient in terms of resource usage.

3. Communication

•Processes:

- **Inter-Process Communication (IPC)**: Requires mechanisms like Queue, Pipe, or Manager for data sharing.

•Threads:

- **Shared data**: Directly access shared memory, but must manage synchronization (e.g., using Lock).

4. Concurrency

•Processes:

- **True parallelism**: Multiple processes can run simultaneously on multi-core CPUs.
- **Global Interpreter Lock (GIL)**: Not affected, so ideal for CPU-bound tasks.

•Threads:

- **Concurrent execution**: Threads can run concurrently but are subject to GIL in Python, limiting true parallelism.
- **Best for I/O-bound tasks**: Where waiting on I/O allows other threads to run.

5. Use Cases

•Processes:

- **CPU-bound tasks**: Heavy computations, simulations, and data processing.

•Threads:

Starting a Multiprocessing Pool

```
from multiprocessing import Pool, set_start_method
from time import time

def square(x):
        return x * x

if __name__ == "__main__":
        # Explicitly set the start method to 'spawn'
        # Important for some Macs and Windows!
        set_start_method('spawn', force=True)

        t0 = time()
        with Pool(processes=4) as pool:
                results = pool.map(square, range(1000))
        t1 = time()
        msg = 'Parallel compute time: ' + str(t1-t0) + ' seconds.'
        print(msg)
        print(results[:10])


        # Now do this as a for-loop
        t0 = time()
        results = []
        for i in range(1000):
                results.append(square(i))
        t1 = time()
        msg = 'Serial compute time: ' + str(t1-t0) + ' seconds.'
        print(msg)
        print(results[:10])
```

Not required for parallel computing. Using this to compare time to run with and without parallelized code.

Multiprocessing is the name of the module to load, and Pool is the method you will often use to run code in parallel.

set_start method is called below and is required on a Mac or Windows machine but not on a Linux machine.

Starting a Multiprocessing Pool

```
from multiprocessing import Pool, set_start_method
from time import time

def square(x):
    return x * x

if __name__ == "__main__":
    # Explicitly set the start method to 'spawn'
    # Important for some Macs and Windows!
    set_start_method('spawn', force=True)

    t0 = time()
    with Pool(processes=4) as pool:
        results = pool.map(square, range(1000))
    t1 = time()
    msg = 'Parallel compute time: ' + str(t1-t0) + ' seconds.'
    print(msg)
    print(results[:10])


    # Now do this as a for-loop
    t0 = time()
    results = []
    for i in range(1000):
        results.append(square(i))
    t1 = time()
    msg = 'Serial compute time: ' + str(t1-t0) + ' seconds.'
    print(msg)
    print(results[:10])
```

set_start method is called here and is required on a Mac or Windows machine but not on a Linux machine. The line should be written as shown. ChatGPT may not include this unless you specify that you are running on a Mac/Windows and need to include this line.

Starting a Multiprocessing Pool

```
from multiprocessing import Pool, set_start_method
from time import time

def square(x):
    return x * x

if __name__ == "__main__":
    # Explicitly set the start method to 'spawn'
    # Important for some Macs and Windows!
    set_start_method('spawn', force=True)

    t0 = time()
    with Pool(processes=4) as pool:
        results = pool.map(square, range(1000))
    t1 = time()
    msg = 'Parallel compute time: ' + str(t1-t0) + ' seconds.'
    print(msg)
    print(results[:10])


    # Now do this as a for-loop
    t0 = time()
    results = []
    for i in range(1000):
        results.append(square(i))
    t1 = time()
    msg = 'Serial compute time: ' + str(t1-t0) + ' seconds.'
    print(msg)
    print(results[:10])
```

with is a Python keyword! It is useful for resource management. This line will temporarily set up a Pool (named ‘pool’) This example will set up 4 individual processes that can run in parallel.

Note that the line ends with a colon and code belonging to with must be indented.

Starting a Multiprocessing Pool

```
from multiprocessing import Pool, set_start_method
from time import time

def square(x):
        return x * x

if __name__ == "__main__":
        # Explicitly set the start method to 'spawn'
        # Important for some Macs and Windows!
        set_start_method('spawn', force=True)

        t0 = time()
        with Pool(processes=4) as pool:
                results = pool.map(square, range(1000))
        t1 = time()
        msg = 'Parallel compute time: ' + str(t1-t0) + ' seconds.'
        print(msg)
        print(results[:10])


        # Now do this as a for-loop
        t0 = time()
        results = []
        for i in range(1000):
                results.append(square(i))
        t1 = time()
        msg = 'Serial compute time: ' + str(t1-t0) + ' seconds.'
        print(msg)
        print(results[:10])
```

This line applies the method map to the object pool.

How does this work? pool houses 4 processes, so this line applies range(1000) as inputs to the function square. Instead of doing this one element at a time, it does so 4 elements at a time.

Starting a Multiprocessing Pool

```
from multiprocessing import Pool, set_start_method
from time import time

def square(x):
        return x * x

if __name__ == "__main__":
        # Explicitly set the start method to 'spawn'
        # Important for some Macs and Windows!
        set_start_method('spawn', force=True)

        t0 = time()
        with Pool(processes=4) as pool:
                results = pool.map(square, range(1000))
        t1 = time()
        msg = 'Parallel compute time: ' + str(t1-t0) + ' seconds.'
        print(msg)
        print(results[:10])
```

```
        # Now do this as a for-loop
        t0 = time()
        results = []
        for i in range(1000):
                results.append(square(i))
        t1 = time()
        msg = 'Serial compute time: ' + str(t1-t0) + ' seconds.'
        print(msg)
        print(results[:10])
```

How does the time required for this calculation differ if we run the same thing serially in a for-loop? What happens if we change the 1000 to something big like 10000000?

```
import random
from multiprocessing import Pool, cpu_count, set_start_method

def monte_carlo_pi_part(num_samples):
    count_inside_circle = 0
    for _ in range(num_samples):
        x, y = random.uniform(-1, 1), random.uniform(-1, 1)
        if x*x + y*y <= 1:
            count_inside_circle += 1
    return count_inside_circle

def estimate_pi(total_samples):
    # Determine the number of processes and samples per process
    num_processes = 2  # Maximum value is cpu_count()
    samples_per_process = total_samples // num_processes
    with Pool(num_processes) as pool:
        # Perform the Monte Carlo simulation in parallel
        counts = pool.map(monte_carlo_pi_part, [samples_per_process] * num_processes)
    # Aggregate results from all processes
    total_count_inside_circle = sum(counts)
    return (4.0 * total_count_inside_circle) / total_samples

if __name__ == "__main__":
    total_samples = 1000

    set_start_method('spawn', force=True)
    print("Estimating Pi with Monte Carlo simulation...")
    t0 = time()
    estimated_pi = estimate_pi(total_samples)
    t1 = time()

    print('Time required: ' + str(t1-t0) + ' seconds')
    print(f"Estimated Pi: {estimated_pi}")
```

How do changing num_processes and total_samples impact run time?

Parallelizing functions with multiple inputs

Many times you want to execute a function for which either

a) There are multiple input values that are paired together, and both are different each time the function is called.

Use starmap.

b) You have multiple inputs, and all but one remain the same each time the function is called.

Use partial from functools.

Using starmap.

```
from multiprocessing import Pool

# Define a function that takes two arguments
def add(x, y):
    return x + y

if __name__ == "__main__":
    # List of argument pairs
    inputs = [(1, 2), (3, 4), (5, 6), (7, 8)]

    # Create a pool of 4 worker processes
    with Pool(processes=4) as pool:
        # Use starmap to apply the 'add' function to each
        # pair of inputs in parallel
        results = pool.starmap(add, inputs)
        # Print the results
        print(results)
```

Both input variables (x and y) change as a pair.

Using starmap.

```
from multiprocessing import Pool

# Define a function that takes two arguments
def add(x, y):
    return x + y

if __name__ == "__main__":
    # List of argument pairs
    inputs = [(1, 2), (3, 4), (5, 6), (7, 8)]

    # Create a pool of 4 worker processes
    with Pool(processes=4) as pool:
        # Use starmap to apply the 'add' function to each
        # pair of inputs in parallel
        results = pool.starmap(add, inputs)
        # Print the results
        print(results)
```

Inputs to starmap are the function name and one input variable.

Using starmap.

```
from multiprocessing import Pool

# Define a function that takes two arguments
def add(x, y):
    return x + y

if __name__ == "__main__":
    # List of argument pairs
    inputs = [(1, 2), (3, 4), (5, 6), (7, 8)]

    # Create a pool of 4 worker processes
    with Pool(processes=4) as pool:
        # Use starmap to apply the 'add' function to each
        # pair of inputs in parallel
        results = pool.starmap(add, inputs)
        # Print the results
        print(results)
```

Inputs are defined here as a sequence of tuples. For each tuple, the first number gets mapped to x and the second number gets mapped to y.

Using partial.

```
from multiprocessing import Pool
from functools import partial

# Function to calculate the area of a rectangle
def calculate_area(width, height):
    return width * height

if __name__ == "__main__":
    # Height is constant
    constant_height = 10

    # List of varying widths
    widths = [2, 4, 6, 8, 10]

    # Partially apply the height constant using `partial`
    calculate_area_with_height = partial(calculate_area, height=constant_height)

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
        # Use `map` to apply the function to the list of widths
        results = pool.map(calculate_area_with_height, widths)
        # Print the results
        print(results)
```

We still have two input variables, but only one (width) changes.

Using partial.

```
from multiprocessing import Pool
from functools import partial

# Function to calculate the area of a rectangle
def calculate_area(width, height):
    return width * height

if __name__ == "__main__":
    # Height is constant
    constant_height = 10

    # List of varying widths
    widths = [2, 4, 6, 8, 10]

    # Partially apply the height constant using `partial`
    calculate_area_with_height = partial(calculate_area, height=constant_height)

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
        # Use `map` to apply the function to the list of widths
        results = pool.map(calculate_area_with_height, widths)
        # Print the results
        print(results)
```

Since map can only accept a single input variable, and we don’t want to copy constant_height to memory for each width, we need to create a bridge function using partial.

Using partial.

```
from multiprocessing import Pool
from functools import partial

# Function to calculate the area of a rectangle
def calculate_area(width, height):
        return width * height

if __name__ == "__main__":
        # Height is constant
        constant_height = 10

        # List of varying widths
        widths = [2, 4, 6, 8, 10]

        # Partially apply the height constant using `partial`
        calculate_area_with_height = partial(calculate_area, height=constant_height)

        # Create a pool of worker processes
        with Pool(processes=4) as pool:
                # Use `map` to apply the function to the list of widths
                results = pool.map(calculate_area_with_height, widths)
                # Print the results
                print(results)
```

Call the function you want to execute in parallel, and then list all of the constant variables (can be more than one) separated by commas.

Using partial.

```
from multiprocessing import Pool
from functools import partial

# Function to calculate the area of a rectangle
def calculate_area(width, height):
    return width * height

if __name__ == "__main__":
    # Height is constant
    constant_height = 10

    # List of varying widths
    widths = [2, 4, 6, 8, 10]

    # Partially apply the height constant using `partial`
    calculate_area_with_height = partial(calculate_area, height=constant_height)

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
        # Use `map` to apply the function to the list of widths
        results = pool.map(calculate_area_with_height, widths)
        # Print the results
        print(results)
```

The constant input variables take the format
function local name = global name

Using partial.

```
from multiprocessing import Pool
from functools import partial

# Function to calculate the area of a rectangle
def calculate_area(width, height):
    return width * height

if __name__ == "__main__":
    # Height is constant
    constant_height = 10

    # List of varying widths
    widths = [2, 4, 6, 8, 10]

    # Partially apply the height constant using `partial`
    calculate_area_with_height = partial(calculate_area, height=constant_height)

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
        # Use `map` to apply the function to the list of widths
        results = pool.map(calculate_area_with_height, widths)
        # Print the results
        print(results)
```

Finally, call pool.map, passing the bridge function name and the iterable variable (widths) as inputs.