Parallel calculations

Maciej Marchwiany, PhD

https://www.linkedin.com/pulse/ai-powered-physics-how-ai-revolutionizing-scientific-discovery-cain-ddbwc

Plan

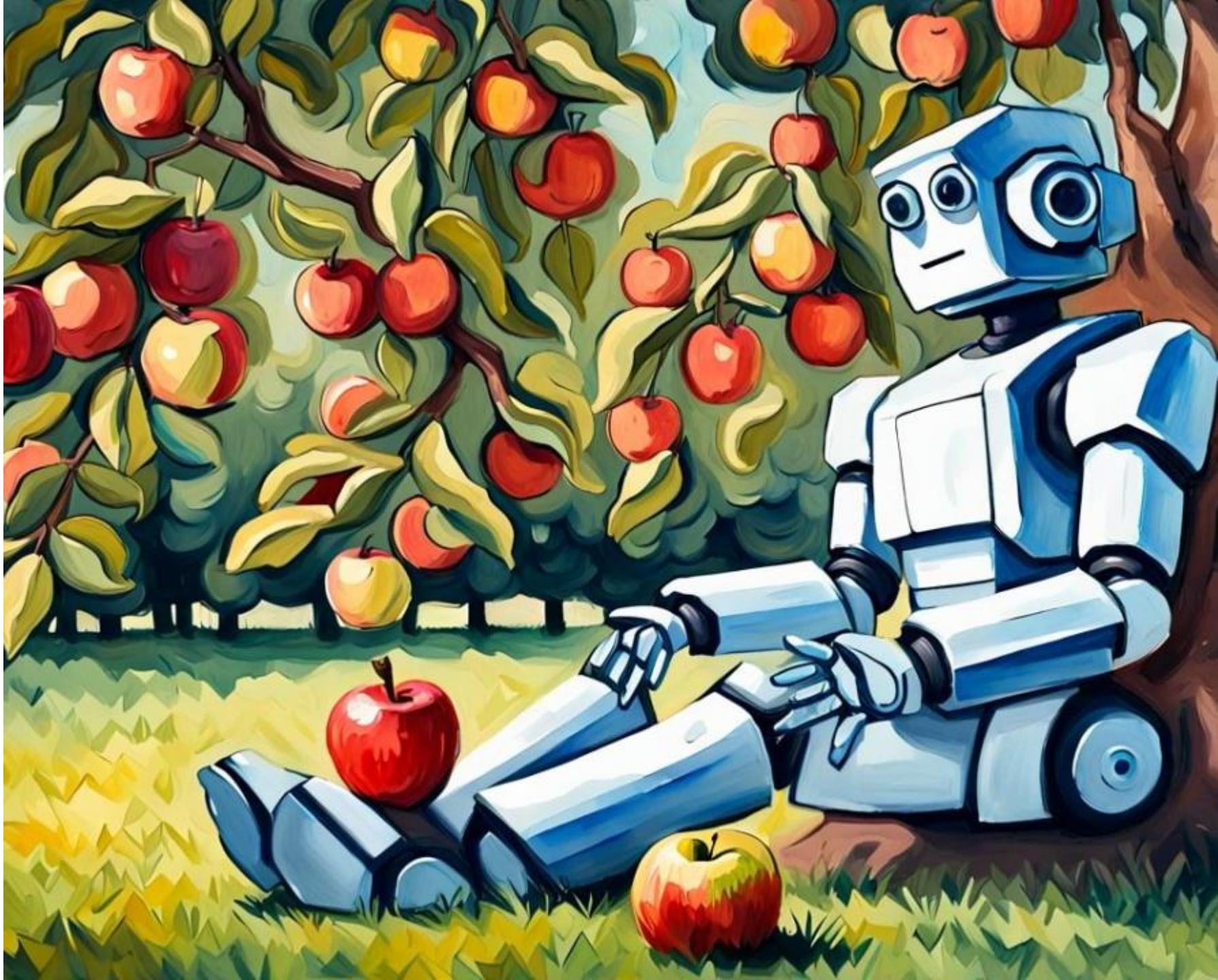
Memory models

Shared memory

Distributed memory

Limitations

Memory models



Cluster architecture

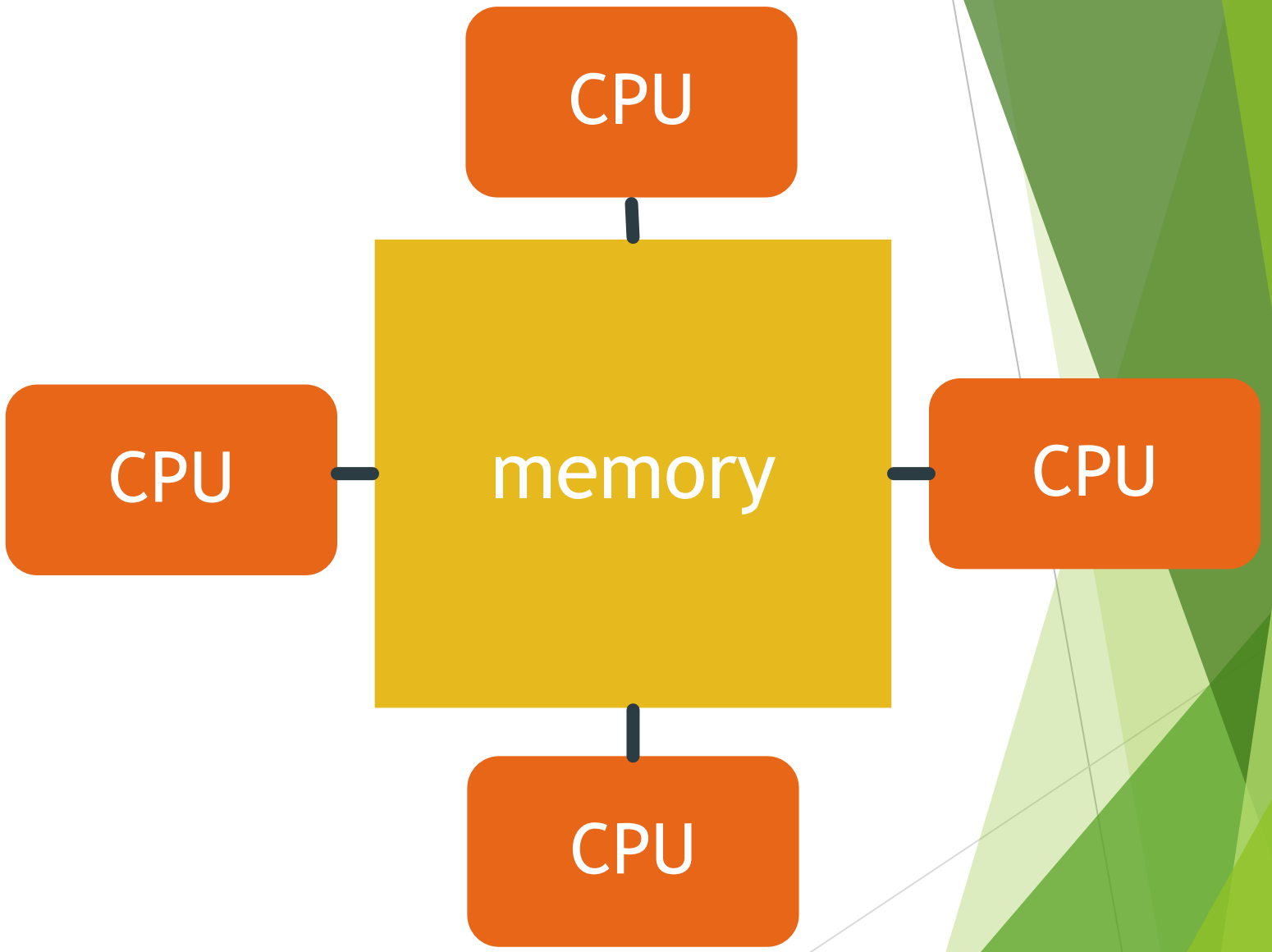
A computer cluster is built of:

- Nodes:
  - Head nodes
  - Compute nodes
  - Service nodes
- Interconection
- Shared storage

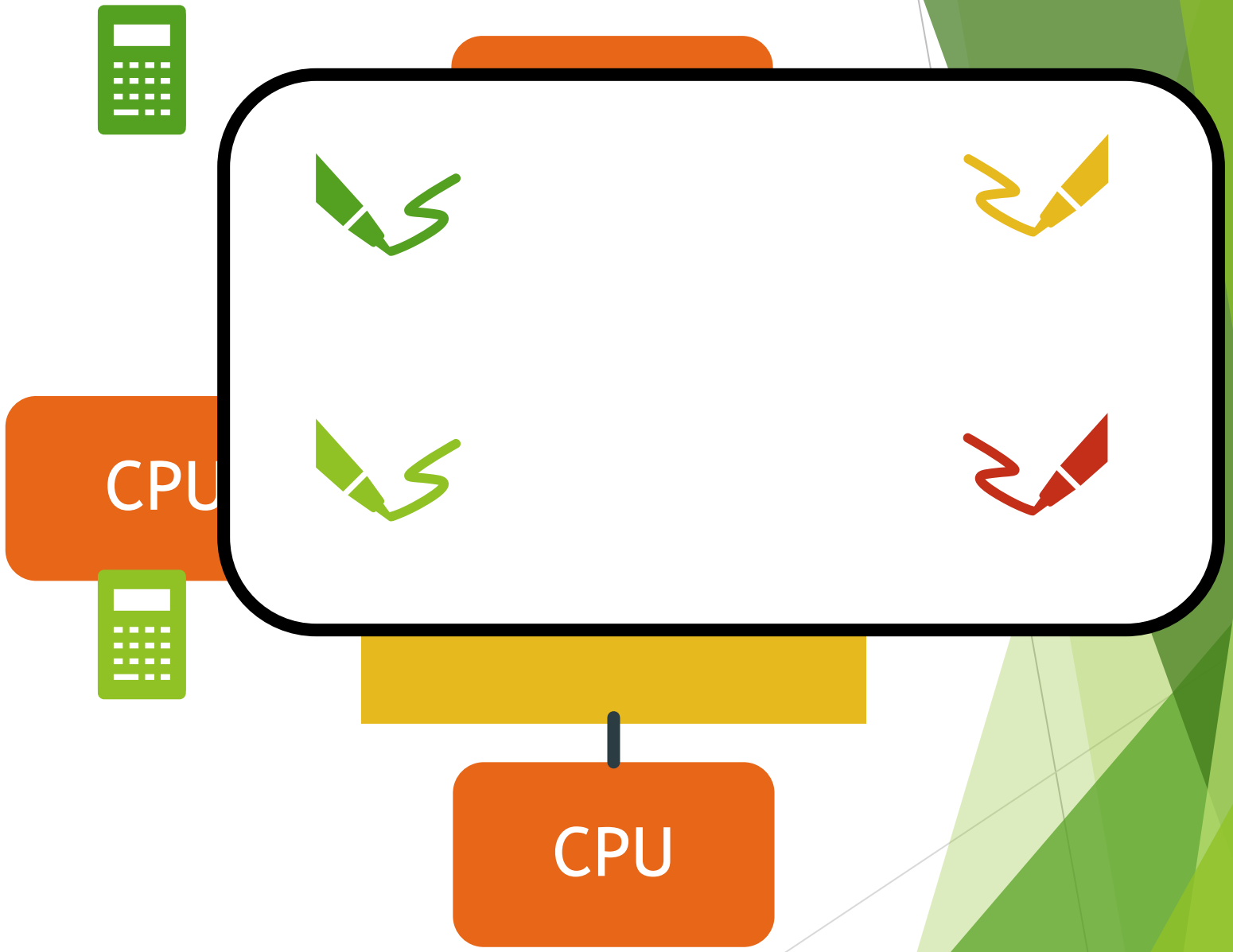
Memory models

- Distributed memory
- Shared memory
- Mixed memory

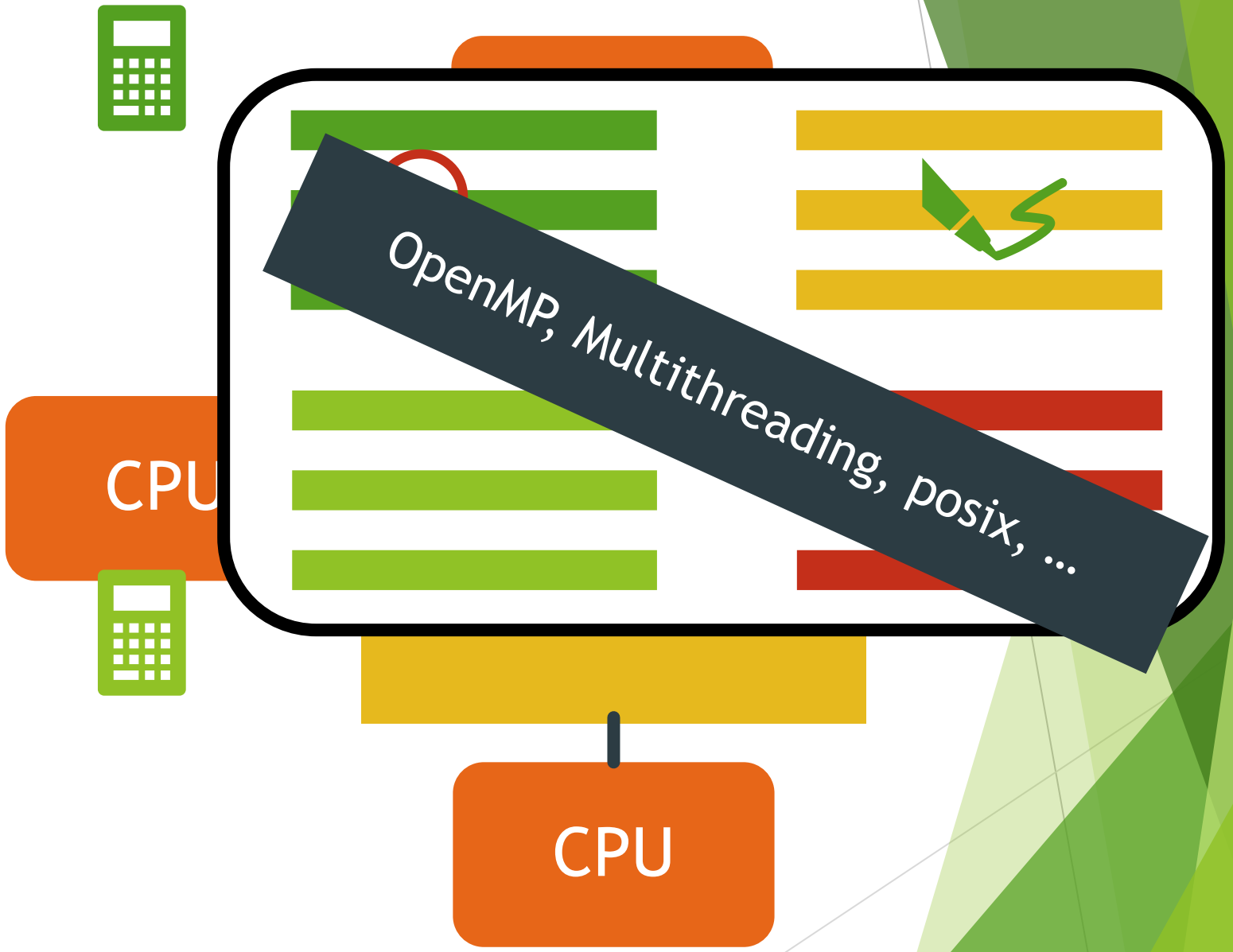
Shared memory



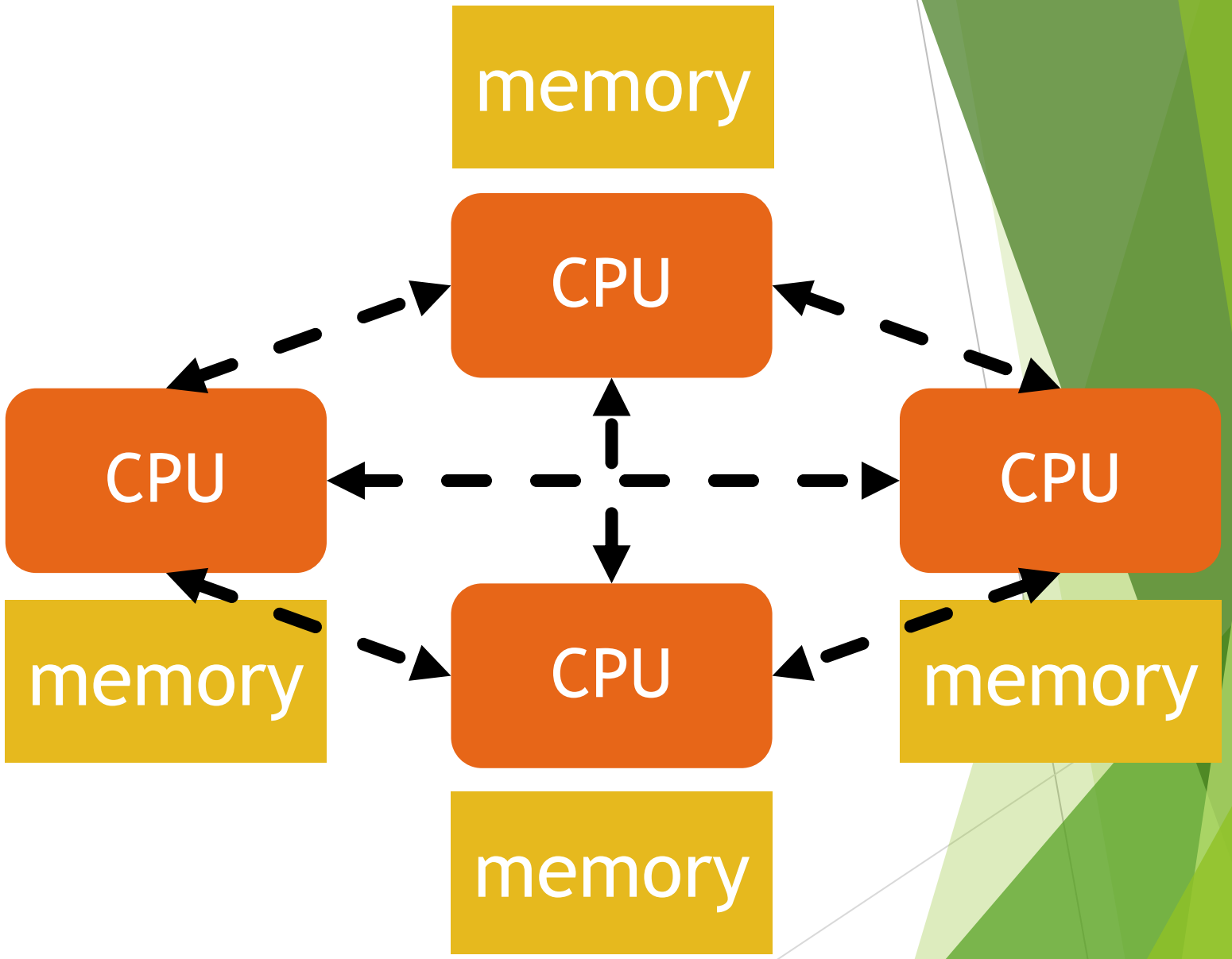
Shared memory



Shared memory

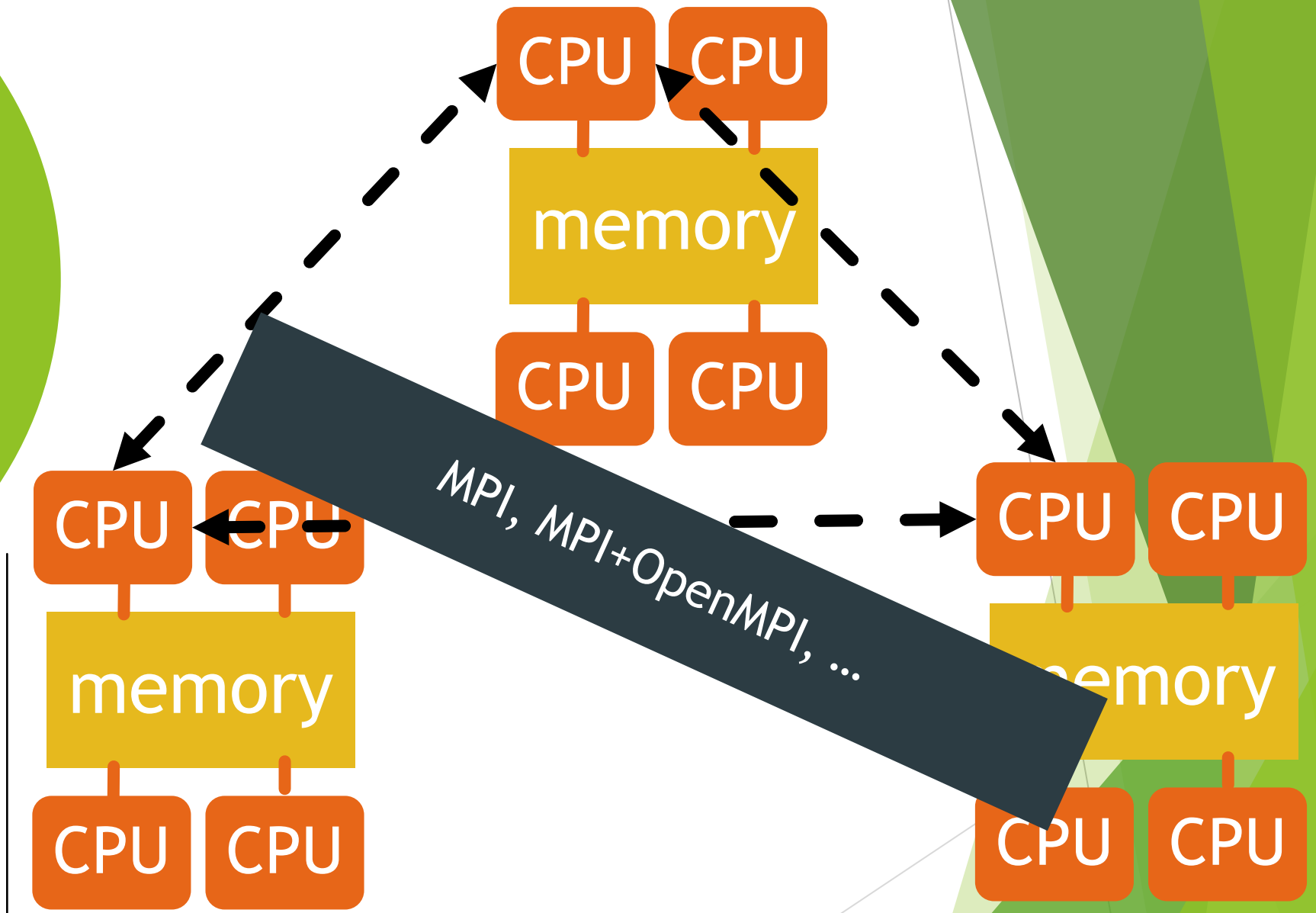


Distributed memory

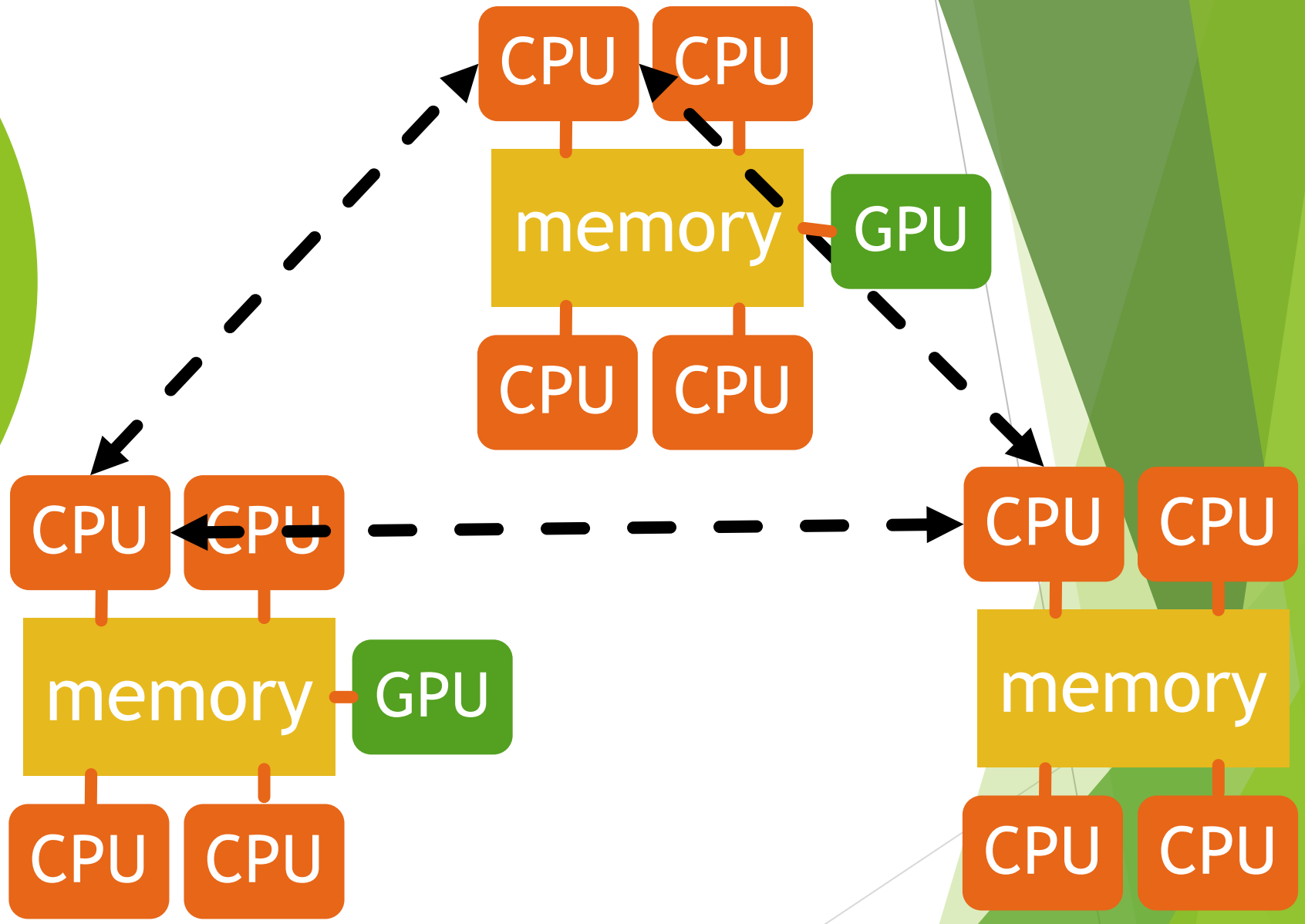


Distributed memory

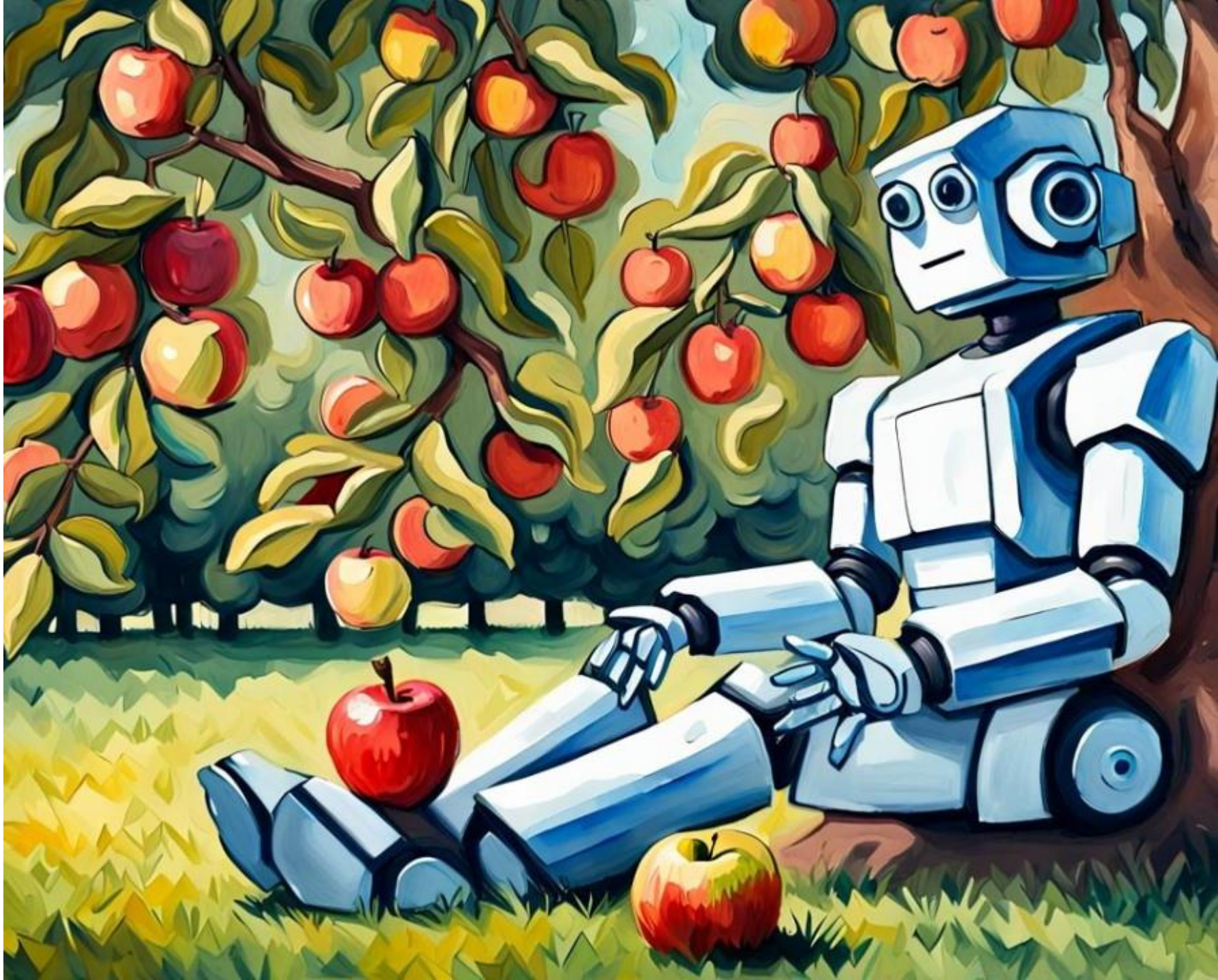
Mixed memory



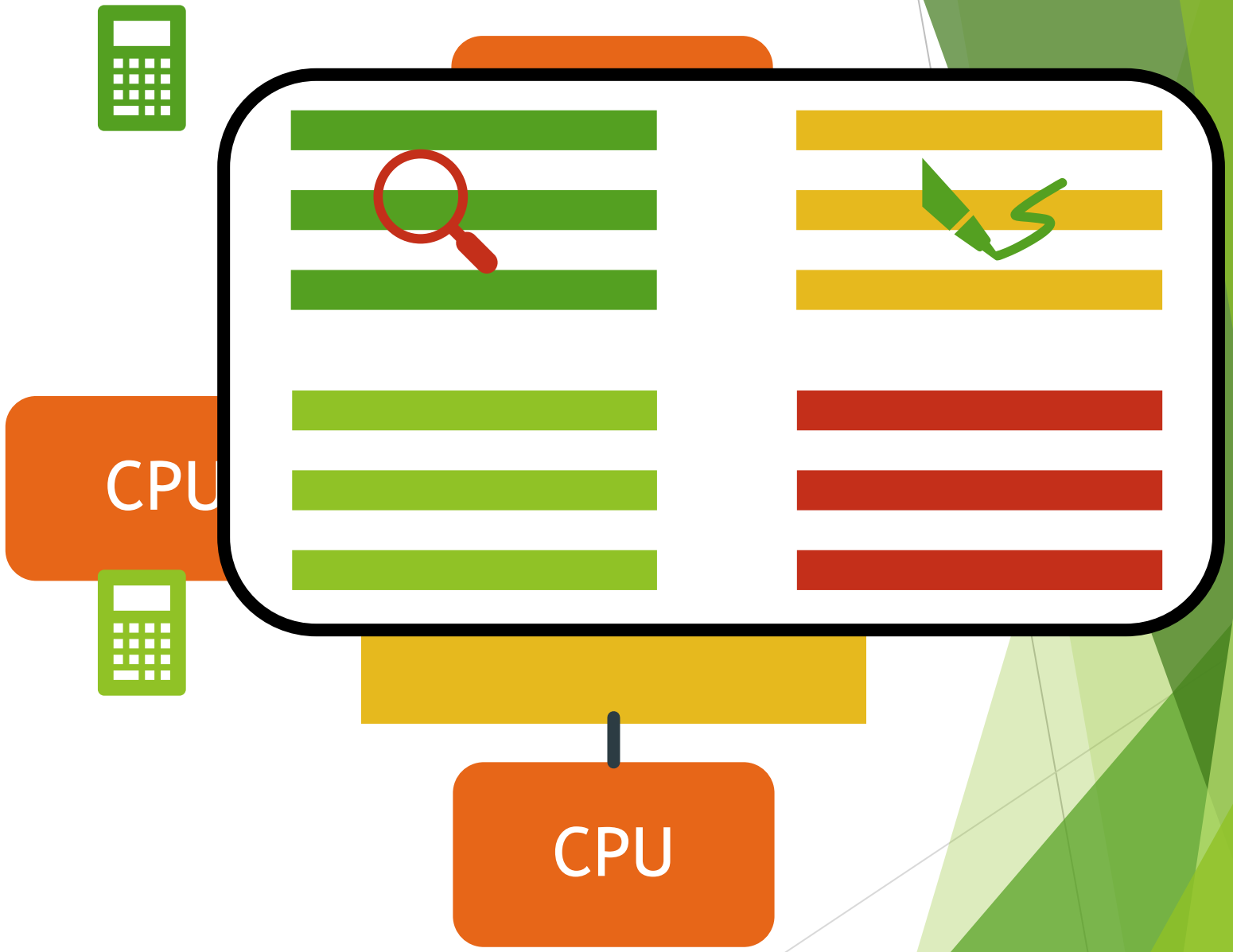
Mixed memory



Shared memory



Shared memory



Threads

A **thread** is the smallest unit of execution within a process. It represents a single sequence of instructions that can be executed independently but shares the same memory space and resources with other threads within the same process. Threads allow concurrent execution, meaning multiple threads can run simultaneously

- **Lightweight** → Threads use less memory compared to processes.
- **Faster Context Switching** → Switching between threads is quicker than between processes.
- **Shared Memory** → Threads within the same process share variables and resources.

- In Python, there are essentially **NO** real threads.

multiprocessing

The `multiprocessing` package in Python is a built-in module that allows for **parallel execution** of code by creating **multiple processes**. It enables true parallelism by utilizing **multiple CPU** cores, unlike threading, which is limited by the **Global Interpreter Lock** (GIL).

Global Interpreter Lock

The **Global Interpreter Lock (GIL)** is a mutex (lock) in **CPython** (the standard Python interpreter) that **allows only one thread to execute Python bytecode at a time**, even on multi-core processors.

This means that **multi-threading in Python does not achieve true parallelism** for CPU-bound tasks because the GIL forces threads to execute one at a time, limiting performance.

The GIL exists because Python's memory management (especially garbage collection) is **not thread-safe**. The GIL:

- **Simplifies memory management** by preventing race conditions
- **Ensures thread safety** without requiring complex locks

Hello world

```
import multiprocessing as mp

def worker():
 print(f"Hello from {mp.current_process().pid}")

p = mp.Process(target=worker)
p.start()  # Start the process
p.join()   # Wait for the process to finish
```

More processes

```
import multiprocessing as mp

def worker():
    print(f"Hello from {mp.current_process().pid}")

processes = []
for i in range(5):
    p = mp.Process(target=worker)
    processes.append(p)
    p.start()  # Start the process

for p in processes:
    p.join()   # Wait for the process to finish
```

```
Hello from 3149404
Hello from 3149405
Hello from 3149406
Hello from 3149407
Hello from 3149408
```

pool

```
import multiprocessing as mp

def worker(i):
    print(f"Hello from {mp.current_process().pid}")

with mp.Pool(processes=4) as pool:
# Create a pool of 4 worker processes
    jobs = [i for i in range(5)]
    results = pool.map(worker, jobs)
```

```
Hello from 3153811
Hello from 3153812
Hello from 3153813
Hello from 3153814
Hello from 3153811
```

Shared memory

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

num = Value('d', 0.0)
arr = Array('i', range(10))

p = Process(target=f, args=(num, arr))
p.start()
p.join()

print(num.value)
print(arr[:])
```

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Race condition

A **race condition** occurs in a concurrent system (e.g., when using multiple threads or processes) when the behavior of a program depends on the **order or timing** of events, such as which thread or process finishes first. If multiple threads or processes access shared data concurrently and at least one of them modifies it, the result can be unpredictable and may lead to incorrect behavior or bugs.

- **Concurrency**: Multiple threads or processes are working simultaneously.
- **Shared Resource**: Threads or processes access shared data or resources.
- **Timing/Order Dependent**: The final outcome depends on the non-deterministic ordering of thread/process execution.

Race condition

```
import multiprocessing as mp

counter = multiprocessing.Value('i', 0)

def increment(_):
    for _ in range(1000000):
        counter.value += 1

with mp.Pool(processes=4) as pool:
        pool.map(increment, range(4))

    print("Final counter value:", counter.value)
```

```
#5 runs:
Final counter value: 1200139
Final counter value: 1012196
Final counter value: 1532946
Final counter value: 1467751
Final counter value: 1369406
```

Locks

- A **Lock** is a synchronization mechanism that ensures that only one process can access a shared resource at a time, preventing race conditions.

Race condition

```
import multiprocessing as mp

counter = multiprocessing.Value('i', 0)
lock = mp.Lock()

def increment(_,lock):
    for _ in range(1000000):
        lock.acquire()
        counter.value += 1
        lock.release()

with mp.Pool(processes=4) as pool:
  pool.starmap(increment, [(i, lock) for i in range(4)])

print("Final counter value:", counter.value)
```

Like `map` but target function has multiple arguments

```
#5 runs:
Final counter value: 4000000
Final counter value: 4000000
Final counter value: 4000000
Final counter value: 4000000
Final counter value: 4000000
```

Multiprocessing is more

- Server proces (master process)
- Communication between processes
  - Queue
  - Pipe
- Lock on code
- Asynchronous map for pool
- Applay for pool

How to real use threads?

External libraries:

- NumPy
- Pandas
- SciPy
- SciKit-learn
- …

These libraries use **C extensions**, which release the GIL for heavy computations.

JIT compilers like PyPy

```
python script.py  # CPython
pypy3 script.py   # PyPy
```

Do not use PyPy with C-based packages

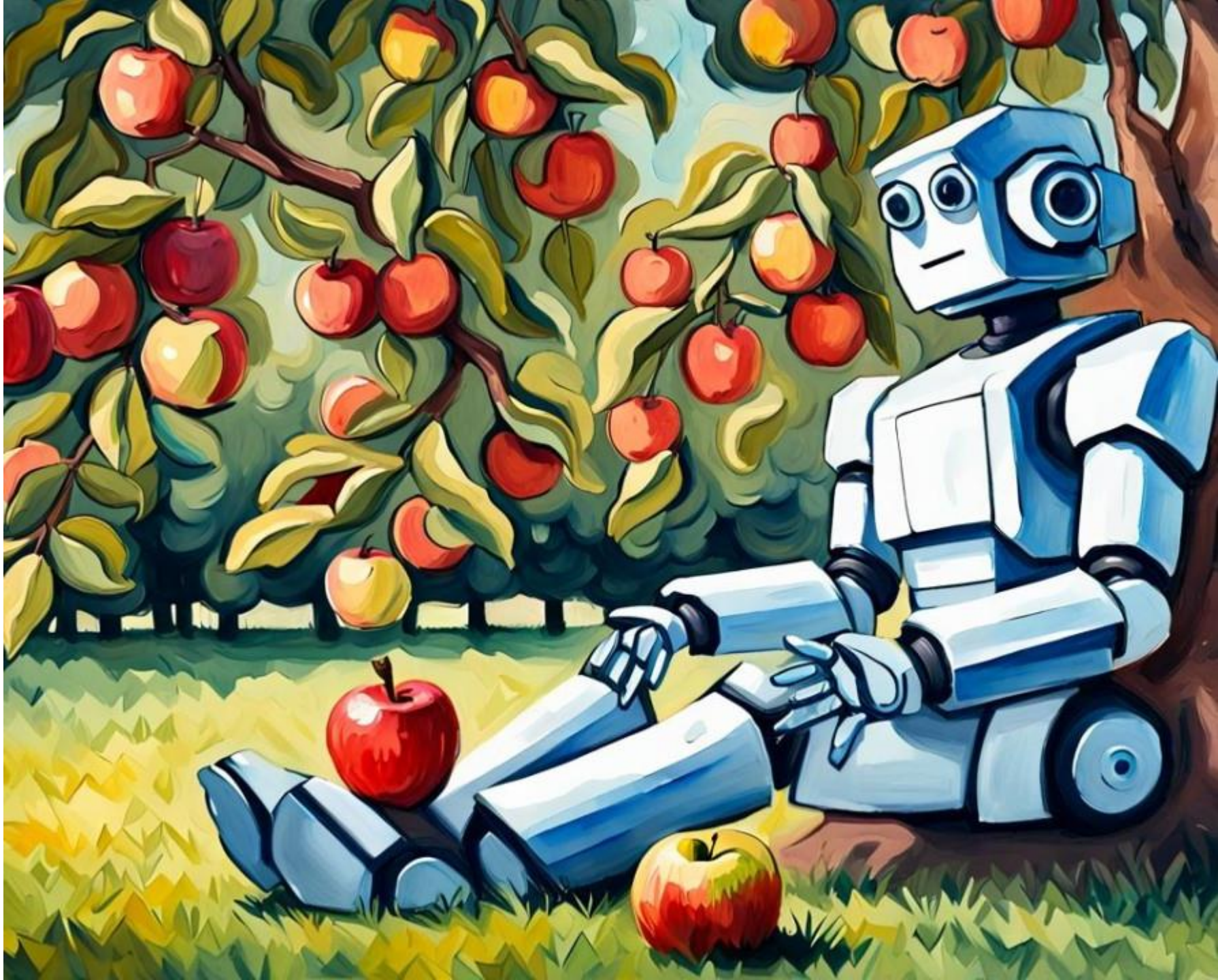
`threading`

Hello world

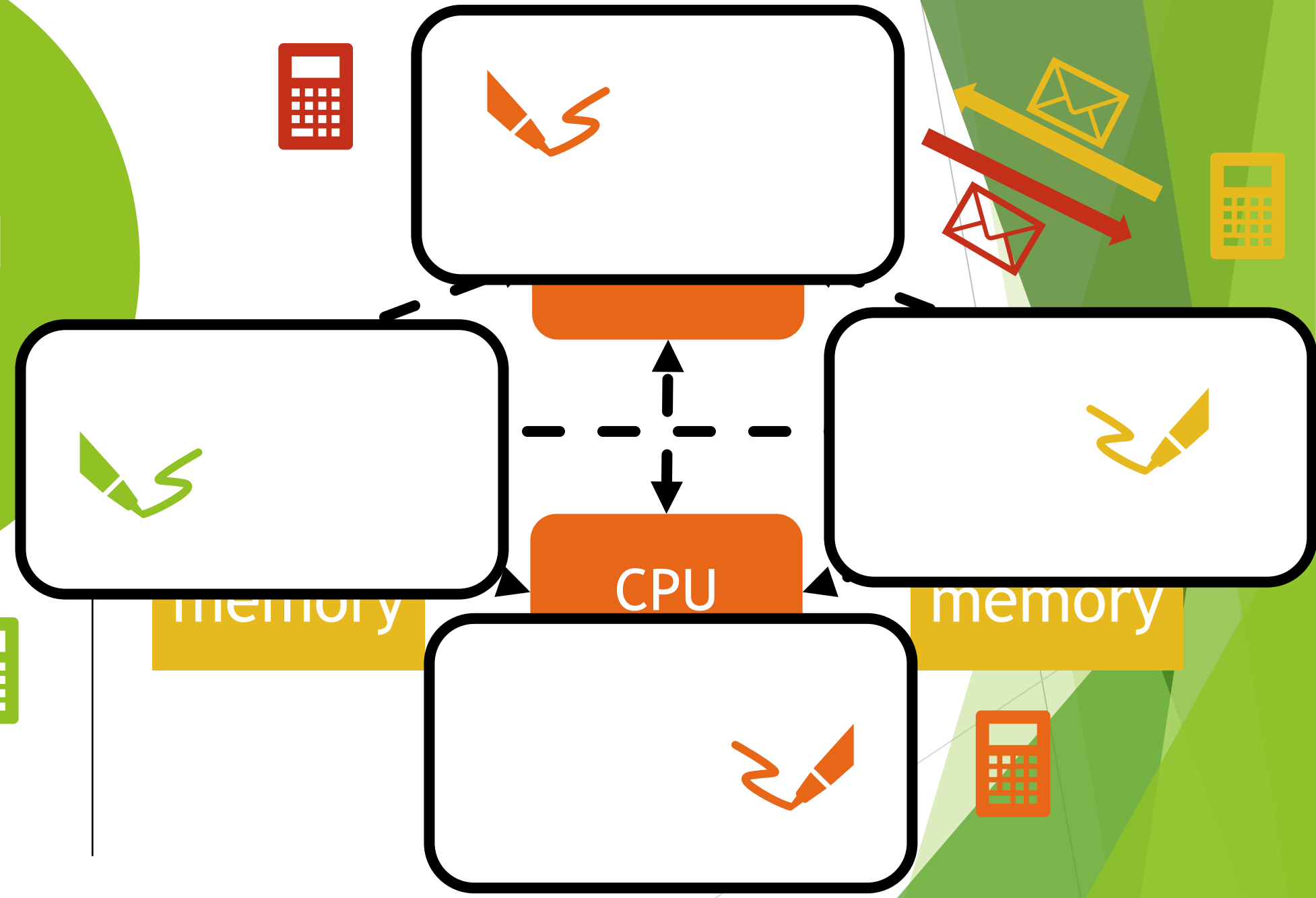
```
import threading
def hello():
        print(f"Hello {threading.current_thread().name}!")

if __name__ == "__main__":
    threads = []
    for i in range(5):
        t = threading.Thread(target=hello, name=f"Thread-{i}")
        threads.append(t)
        t.start
    for t in threads:
        t.join()
```

```
Hello Thread-0!
Hello Thread-1!
Hello Thread-2!
Hello Thread-3!
Hello Thread-4!
```

Distributed memory



Distributed memory



MPI

MPI (Message Passing Interface) is a standardized and portable **message-passing system** designed for **parallel computing.** It allows multiple processes (running on the same or different machines) to communicate and coordinate their tasks.

- Used in high-performance computing (HPC)
- Enables distributed computing across multiple CPUs/nodes
- Supports data sharing via message passing
- Works with languages like C, C++, Fortran, and Python (mpi4py)

When to use MPI?

- **Running on multiple machines** - MPI is designed for distributed systems where computations span across multiple nodes.
- **Running a large-scale paralel processing** - MPI efficiently distributes work across hundreds or thousands of processors.
- **CPU-intensive task** - MPI scales well for heavy computations that need parallel execution.
- **Big memory footprint** - MPI allows processes to work independently without shared memory.
- **When scalability is critical** - MPI works well for thousands of cores with minimal overhead.

How MPI works?

MPI programs run as **multiple independent processes** rather than threads.
These processes:

- Run in parallel on multiple CPUs or machines
- Communicate using message passing (send/receive)
- Have no shared memory, so they must explicitly exchange data

Launching an MPI Program

```
mpiexec -n 4 python my_mpi_script.py
```

Process

A **process** is an independent program in execution, with its own memory space, system resources, and execution context. It represents the highest level of execution unit in an operating system, and each process operates in isolation from others, ensuring that one process cannot directly access the memory or resources of another. Processes can run concurrently on different CPU cores, enabling true parallelism for CPU-bound tasks. Unlike threads, processes do not share memory space, which means inter-process communication (IPC) is required for sharing data between processes. Processes are suitable for tasks that require isolation or extensive CPU processing, as they do not suffer from issues like the Global Interpreter Lock (GIL) that affects threads.

Core concepts

- **Rank** - Each process has a unique rank (ID) for –n 4 ranks = 0,1,2,3
- **Size** - Total number of MPI processes running
- **Communicator** - Defines a group of processes that can communicate (MPI.COMM_WORLD)

Communication:

- Point-to-Point Communication - Direct messaging between two processes
  - send,
  - recv.
- Collective Communication - Group operations between multiple processes (>2)
  - bcast,
  - scatter,
  - gather,
  - reduce.

Hello world

```
from mpi4py import MPI

# Get global communicator
comm = MPI.COMM_WORLD
# Get process ID (0, 1, 2...)
rank = comm.Get_rank()
# Total number of processes
size = comm.Get_size()

print(f"Hello from process {rank} of {size}")
```

```
mpiexec -n 4 python hello_mpi.py
```

```
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 2 of 4
Hello from process 3 of 4
```

point2point

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:  # Process 0 sends data
    data = "Hello from rank 0"
    comm.send(data, dest=1)
    print("Process 0 sent data.")
elif rank == 1:  # Process 1 receives data
    received_data = comm.recv(source=0)
    print(f"Process 1 received: {received_data}")
```

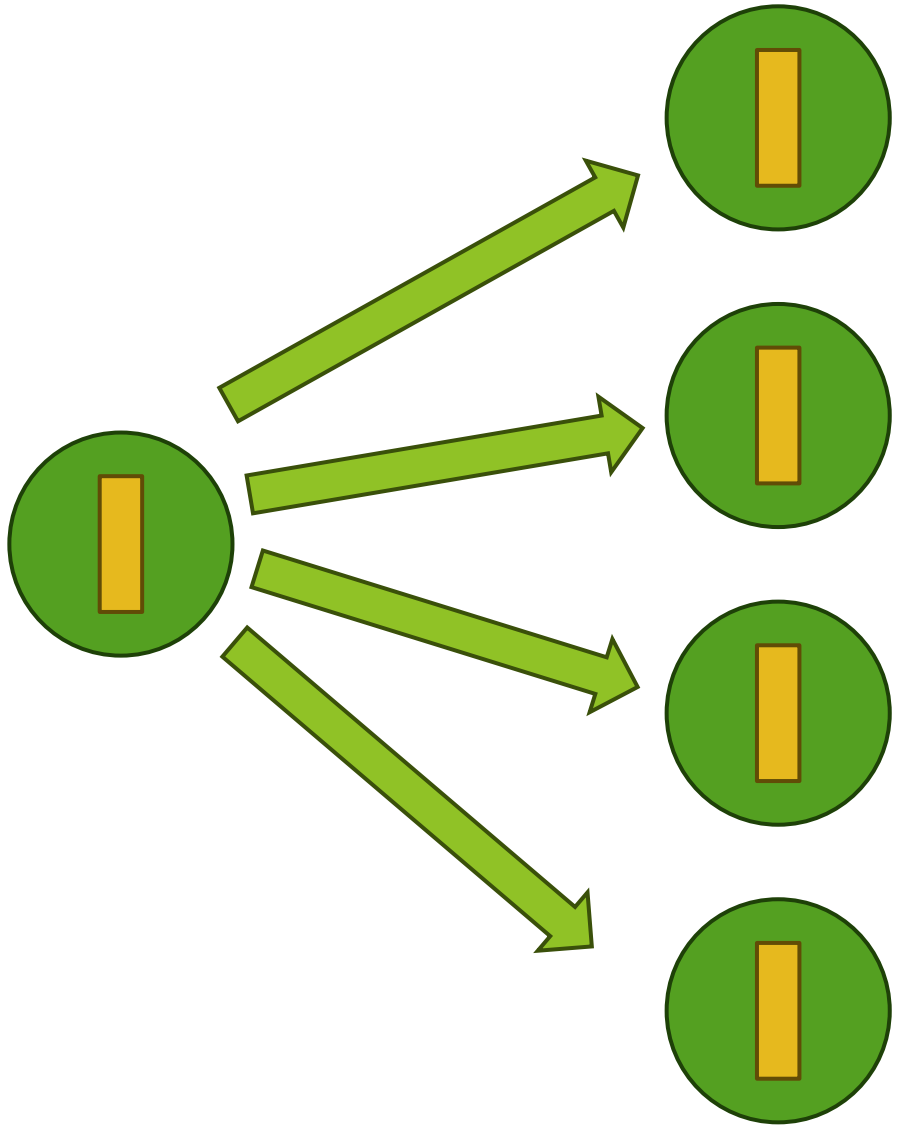
```
mpiexec -n 2 python send_receive.py
```

```
Process 0 sent data.
Process 1 received: Hello from rank 0
```

collective

- Broadcast (bcast): Root process sends the same data to all other processes.
- Scatter (scatter): Root process splits data and sends different parts to each process.
- Gather (gather): Each process sends its data to the root process, which collects everything.
- Reduce (reduce): Each process sends a value, and the root process applies an operation (SUM, MAX, etc.).

bcast

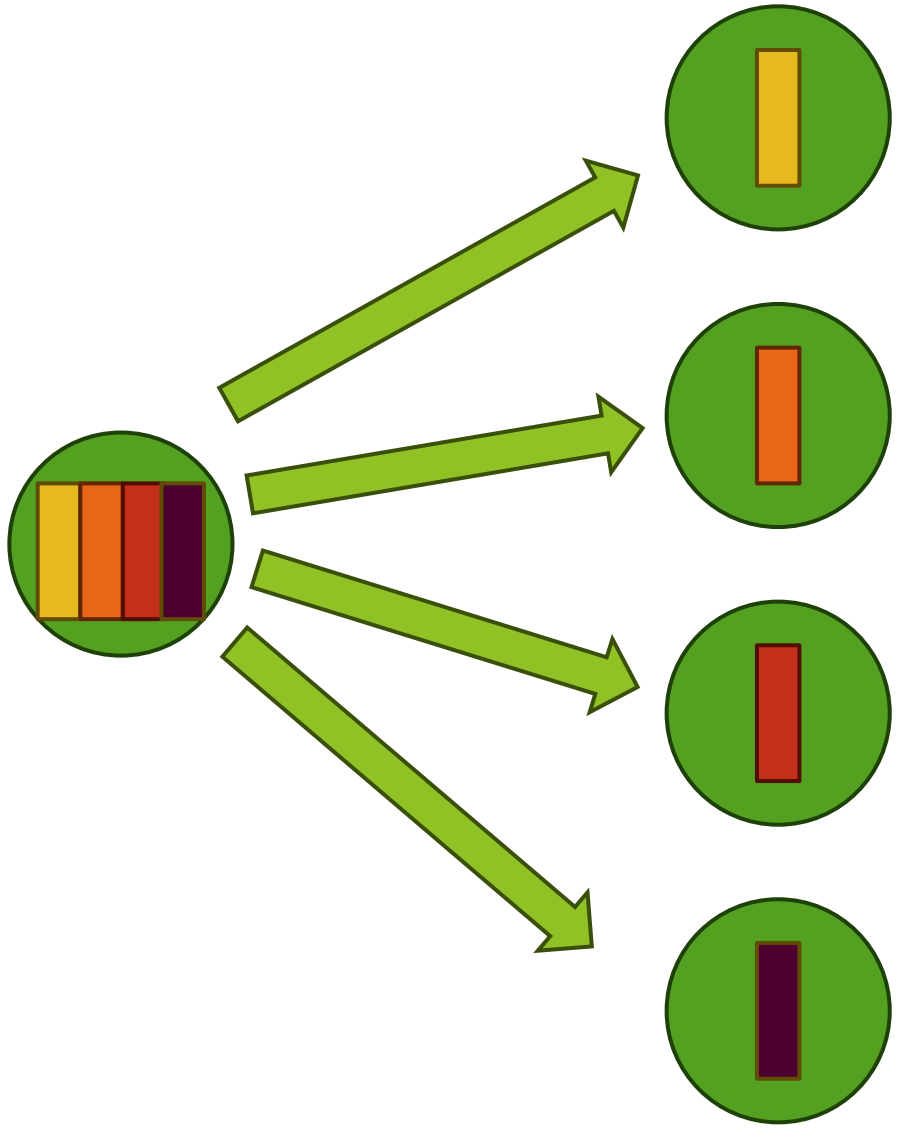


bcast

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Only process 0 has the data initially
data = "Hello, MPI!" if rank == 0 else None

# Broadcasting data from process 0 to all others
data = comm.bcast(data, root=0)

print(f"Process {rank} received: {data}")
```

```
Process 0 received: Hello, MPI!
Process 1 received: Hello, MPI!
Process 2 received: Hello, MPI!
Process 3 received: Hello, MPI!
```

scatter

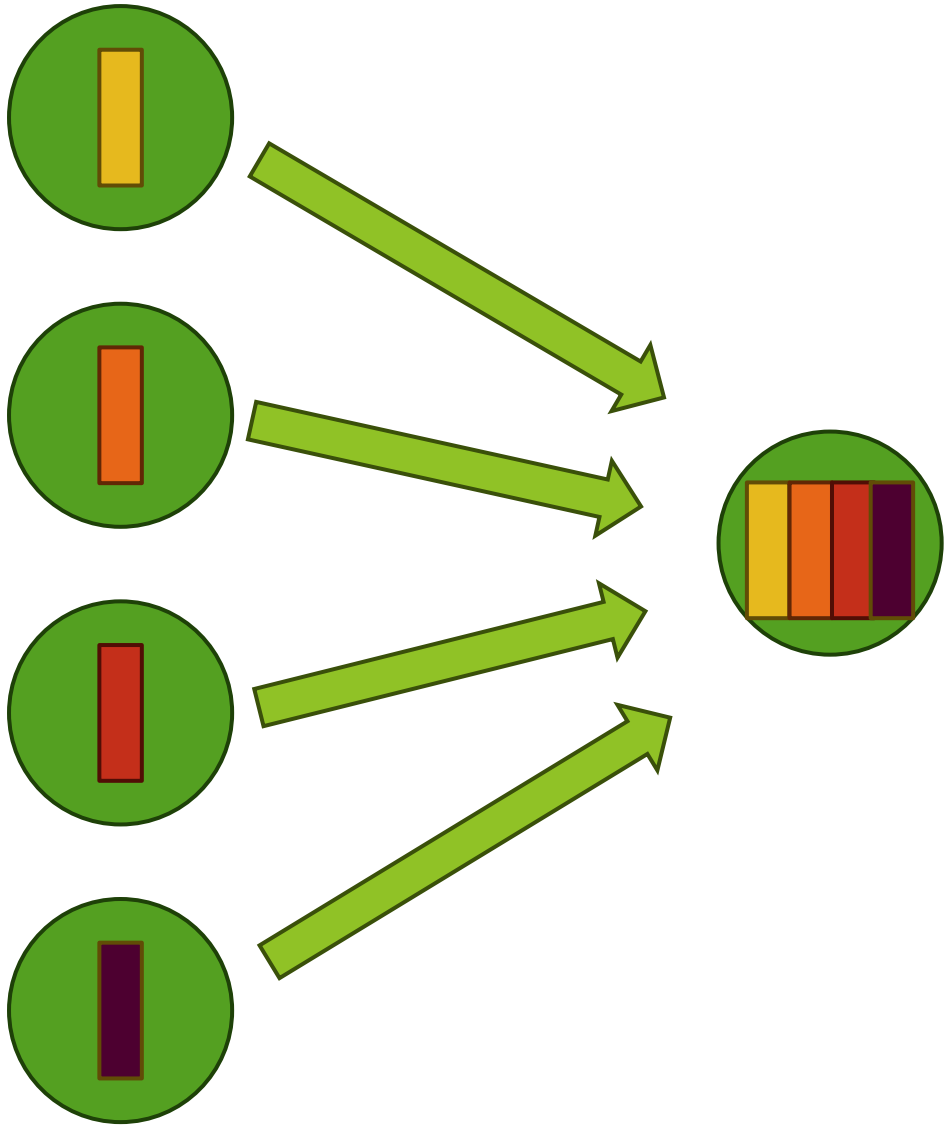


scatter

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Process 0 creates a list of data to distribute
if rank == 0:
    data = [i * 10 for i in range(size)]
else:
    data = None

# Scatter distributes one value per process
recv_data = comm.scatter(data, root=0)

print(f"Process {rank} received: {recv_data}")
```

```
Process 0 received: 0
Process 1 received: 10
Process 2 received: 20
Process 3 received: 30
```

gather

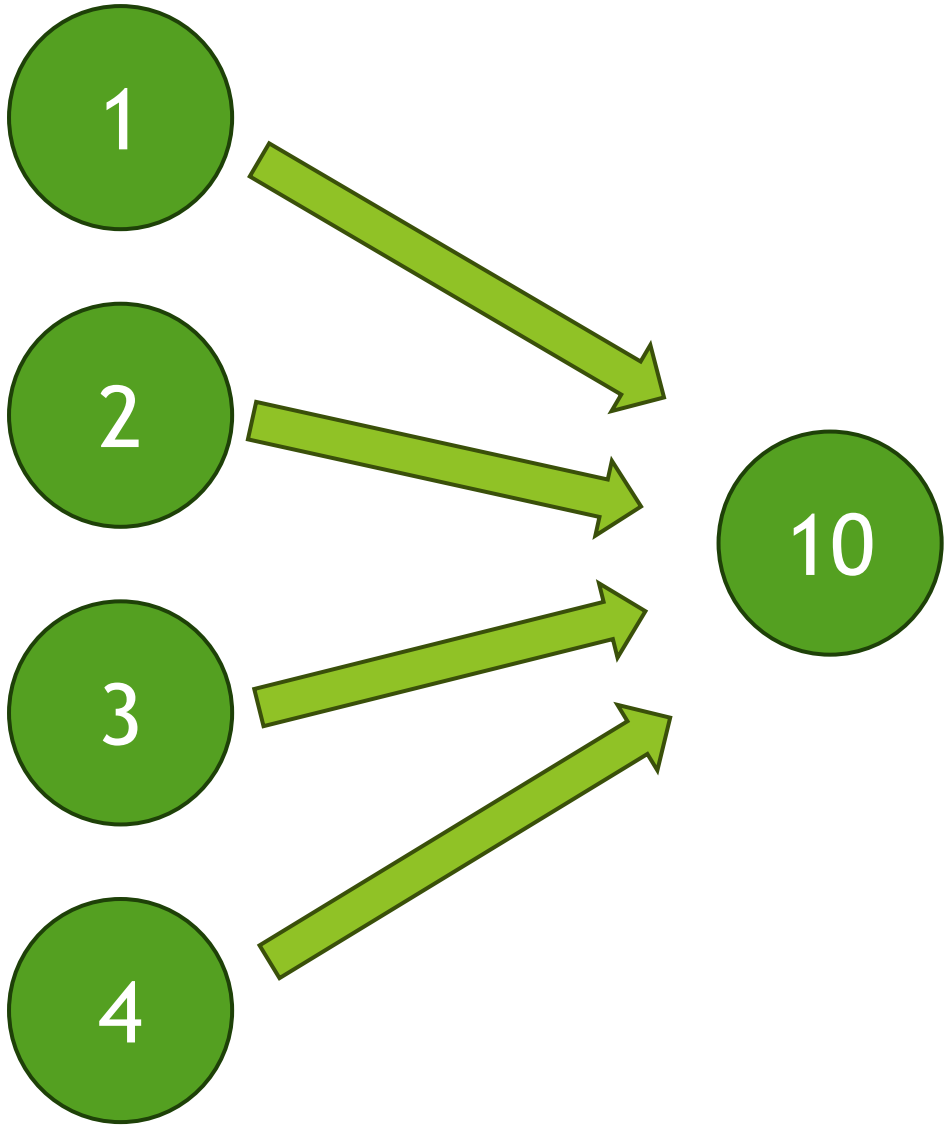


gather

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Each process generates a value
send_data = rank * 2

# Gather collects values at root process (rank 0)
gathered_data = comm.gather(send_data, root=0)

if rank == 0:
    print(f"Gathered data at root: {gathered_data}")
```

```
Gathered data at root: [0, 2, 4, 6]
```

reduce



reduce

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Each process contributes a value
send_value = rank + 1

# Reduce operation (SUM all values)
sum_result = comm.reduce(send_value, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Total sum after Reduce: {sum_r
```

```
Total sum after Reduce: 10
```

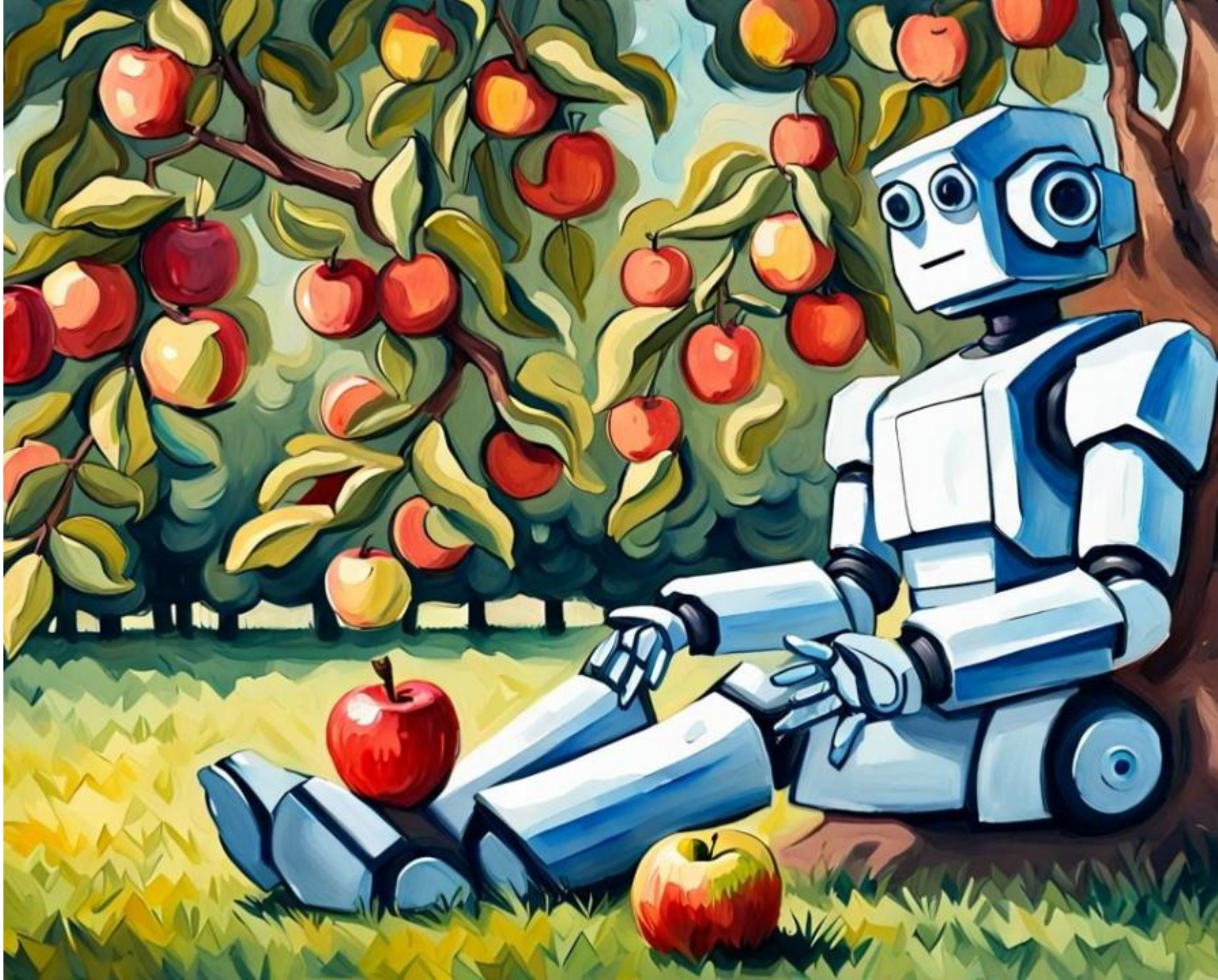
```
Sum of [1, 2, 3, 4] = 10
```

- MPI.SUM
- MPI.PROD
- MPI.MIN
- MPI.MAX
- MPI.LAND
- MPI.LOR
- MPI.LXOR
- MPI.BAND
- MPI.BOR
- MPI.BXOR

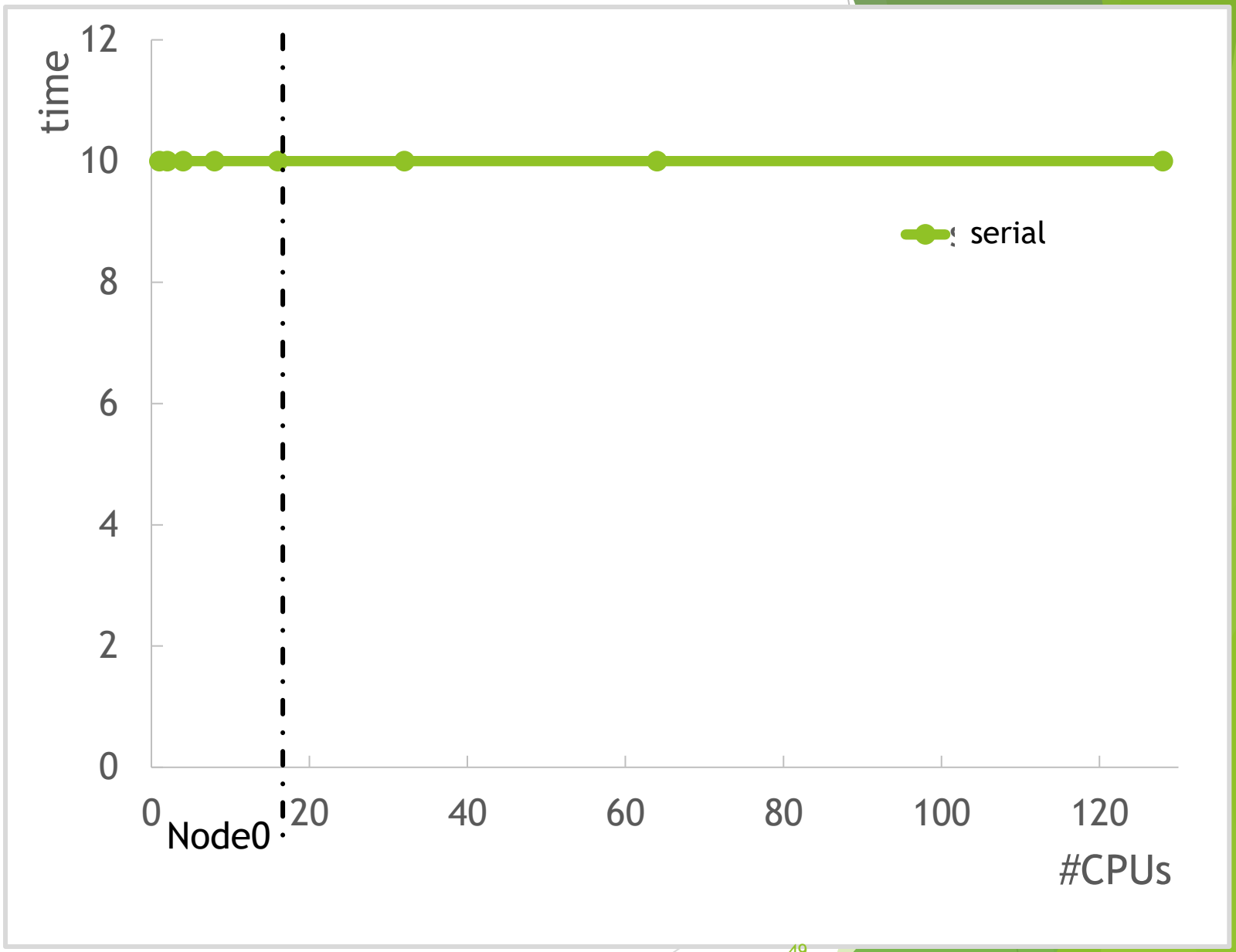
MPI is more

- Communication
  - Blocking
  - Non-blocking
- Synchronization
- Communication
  - Point2point
  - Collective
  - One-sided
- Groups and Communicators
- Custom data types
- MPI I/O

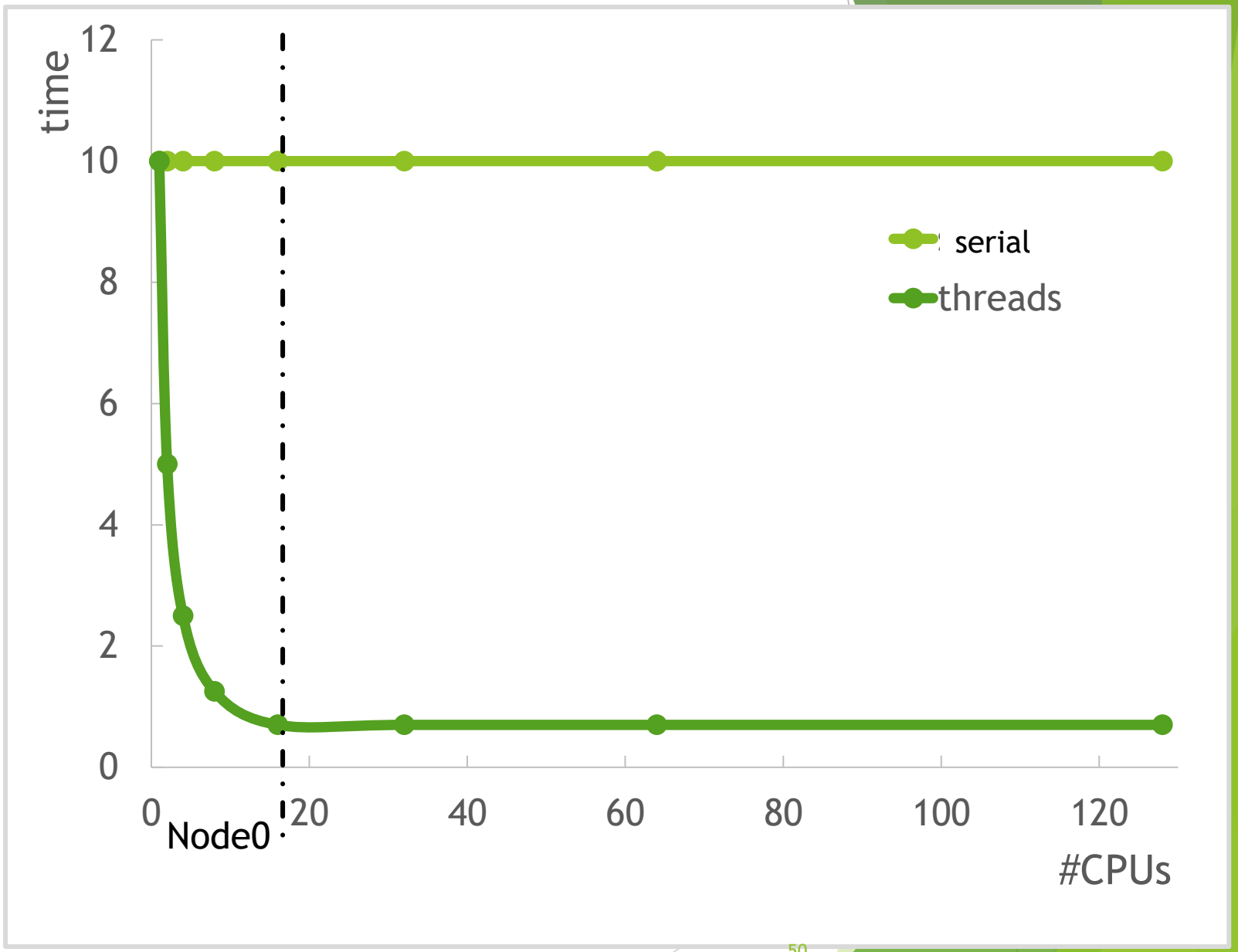
Limitations



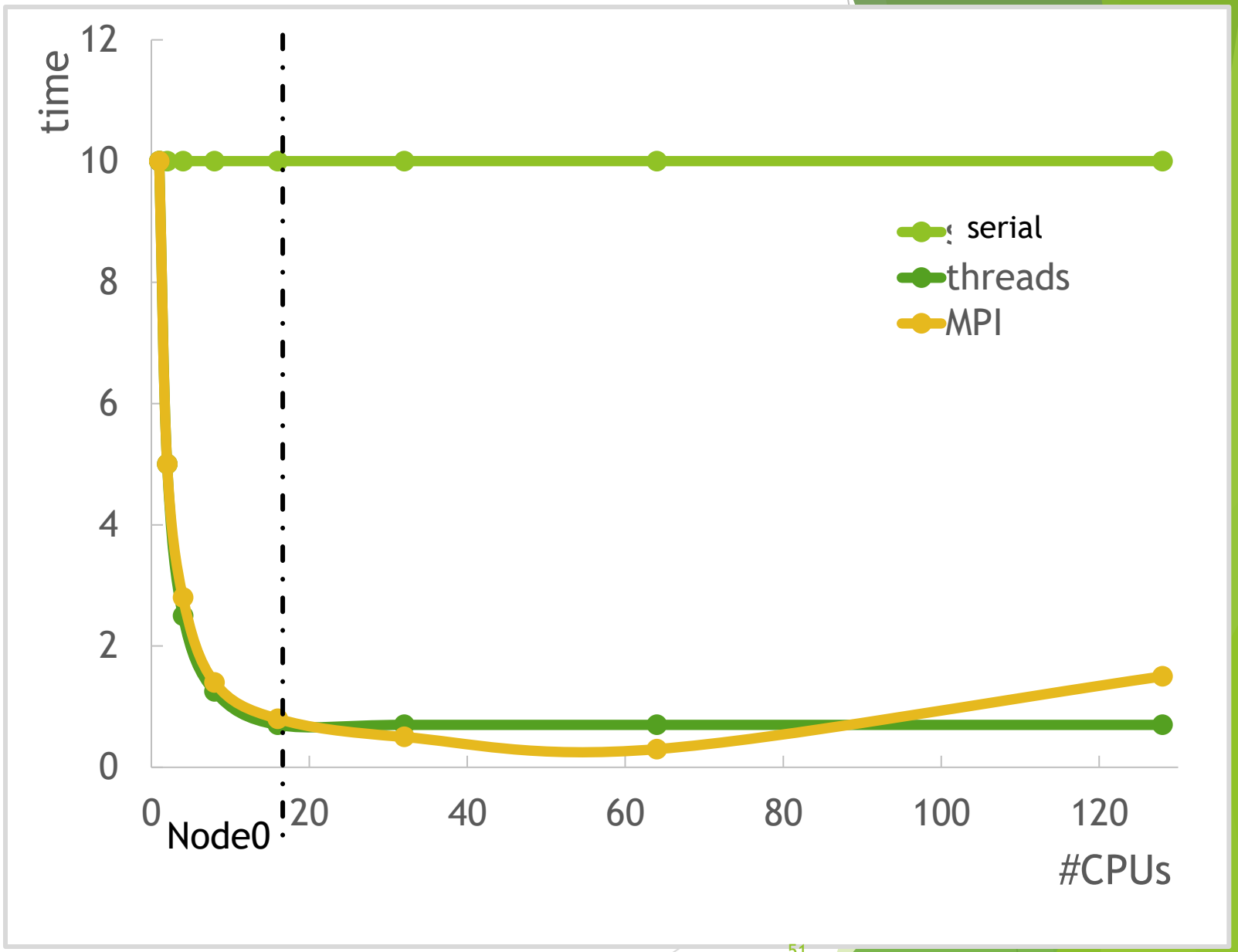
Suboptimal calculations



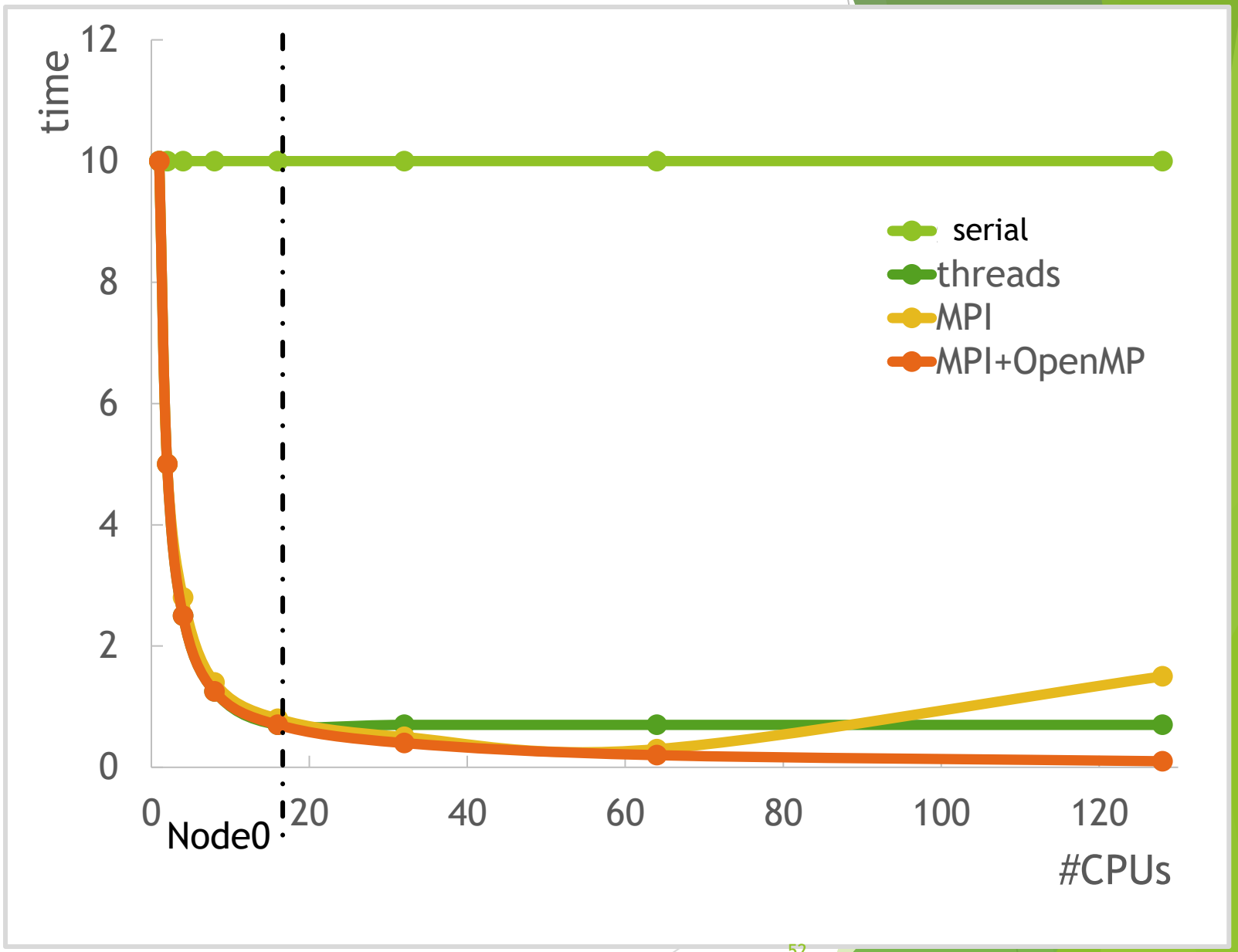
Suboptimal calculations



Suboptimal calculations

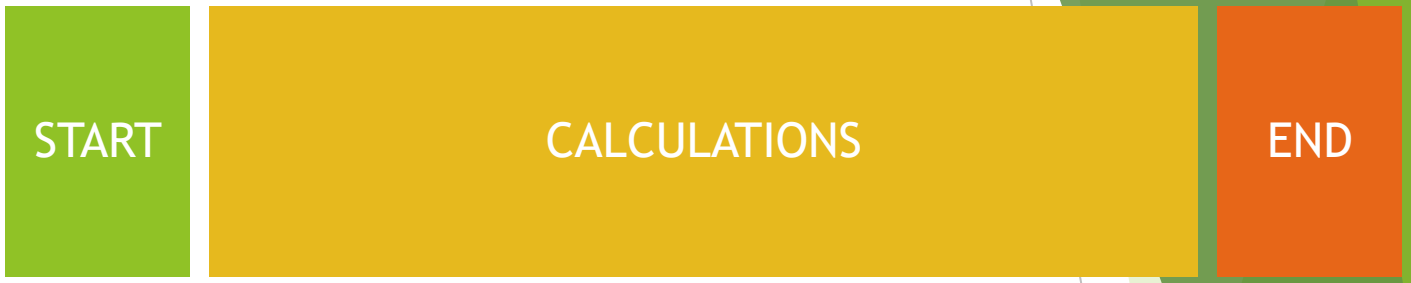


Suboptimal calculations

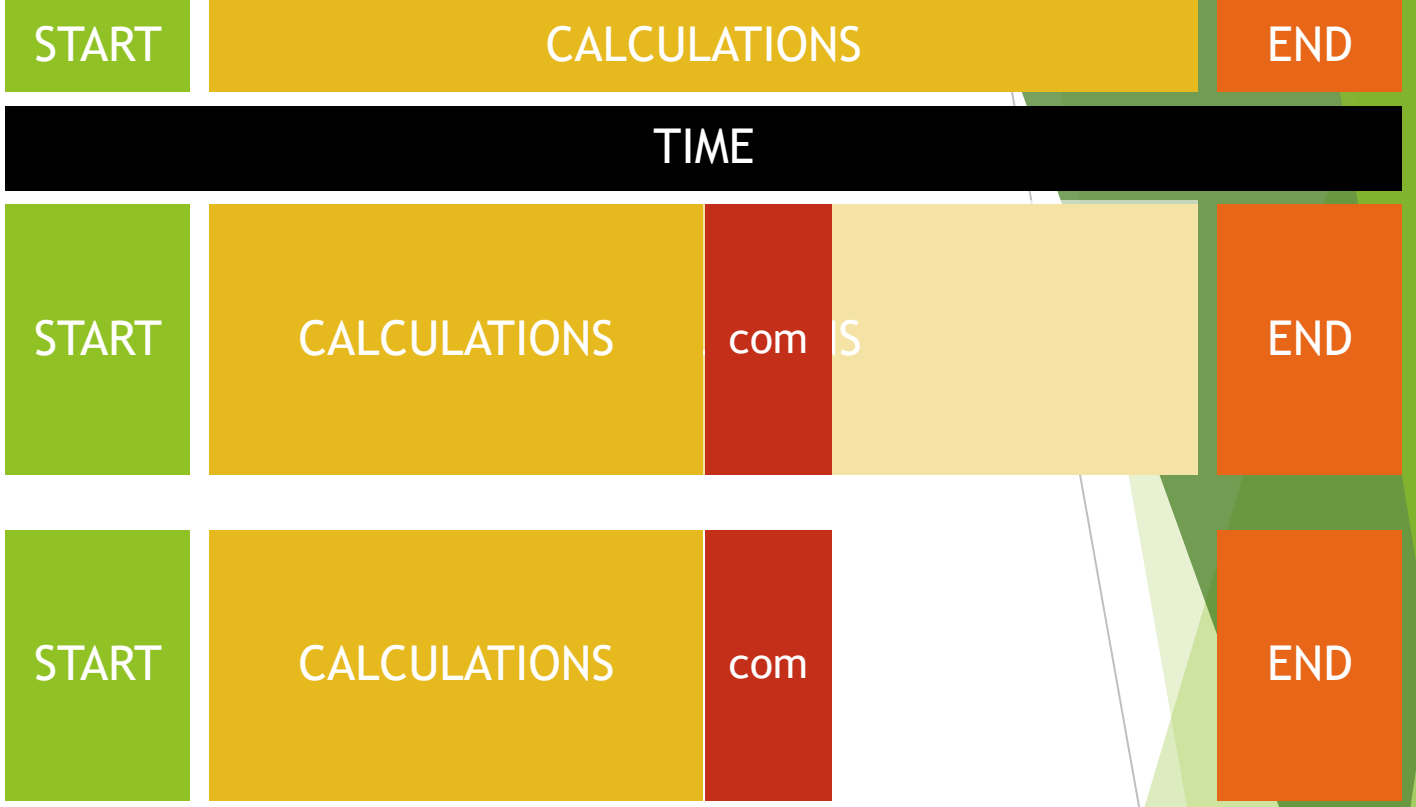


Time of calculations

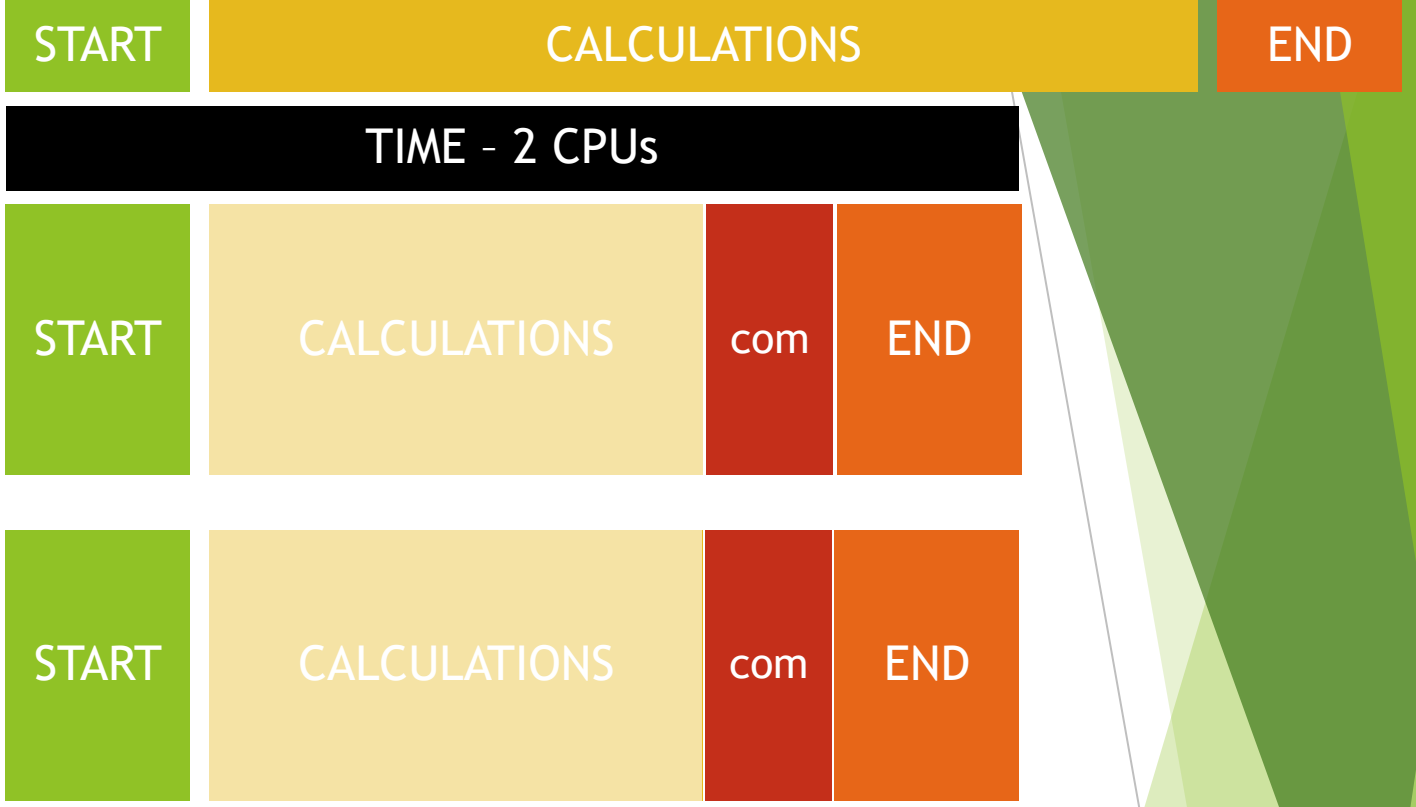
$$T_{calculations} = T_{START} + T_{CALCULATIONS} + T_{END}$$



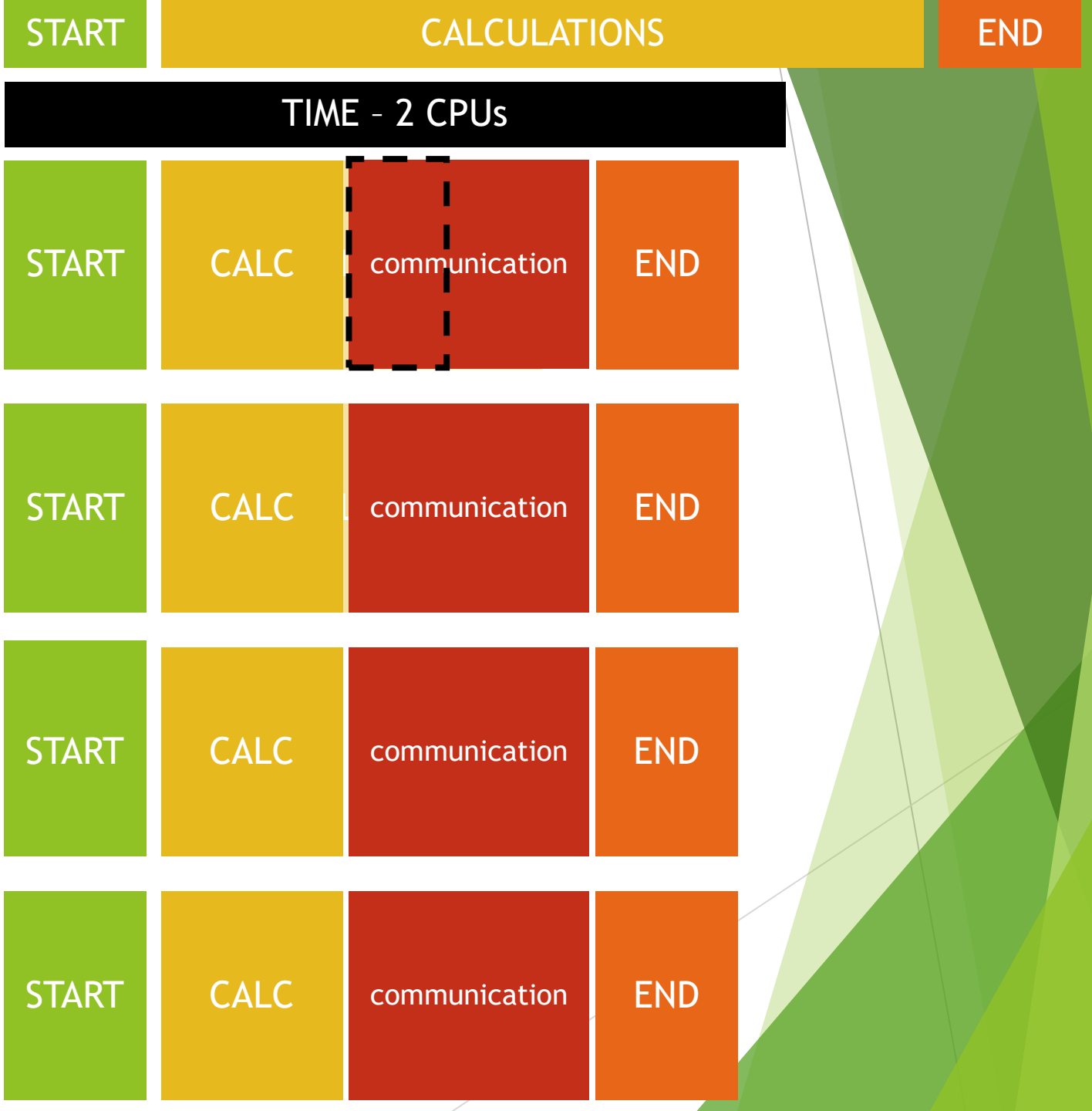
Time of calculations



Time of calculations



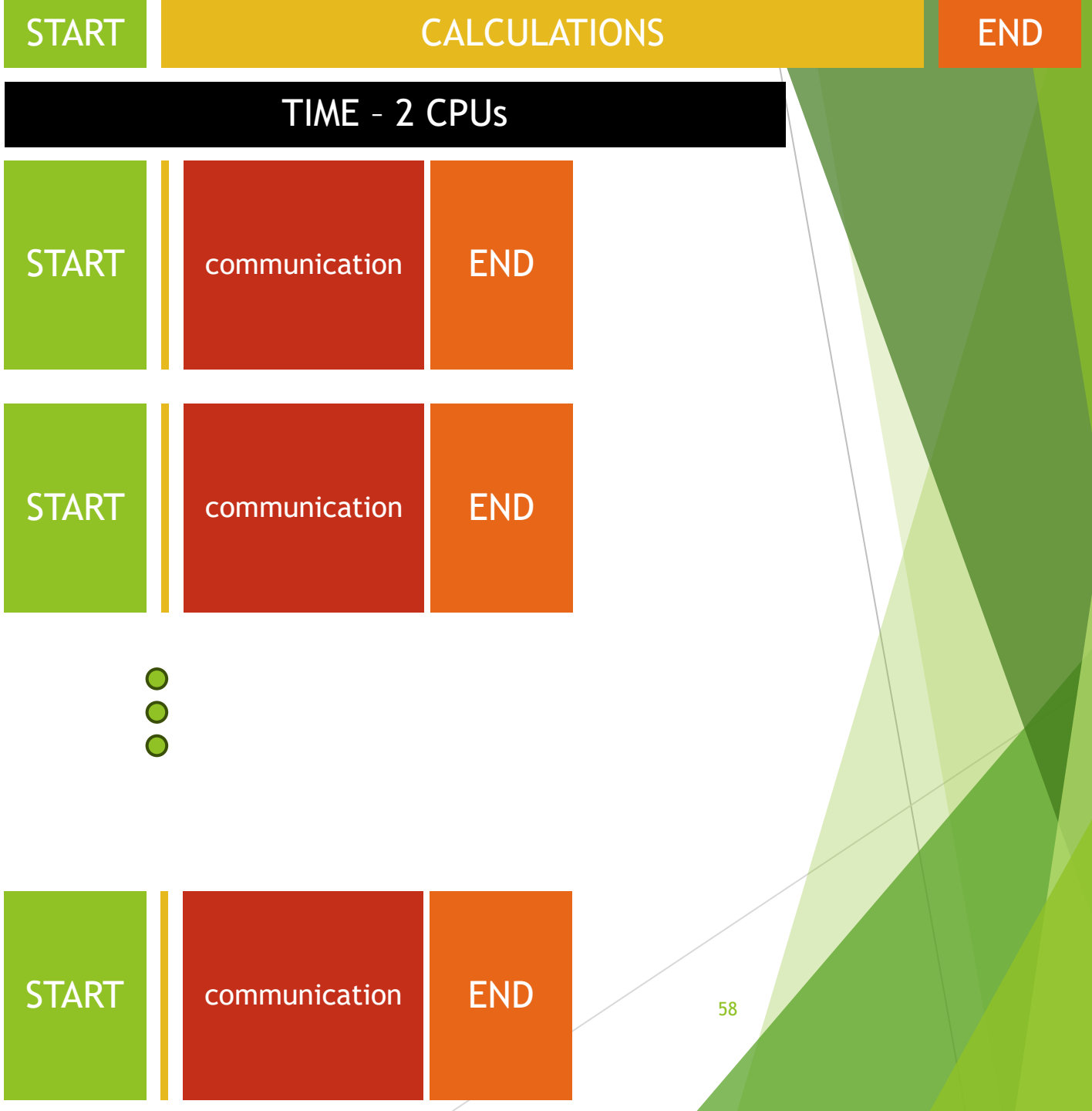
Time of calculations



Parallel integral

t =T/9

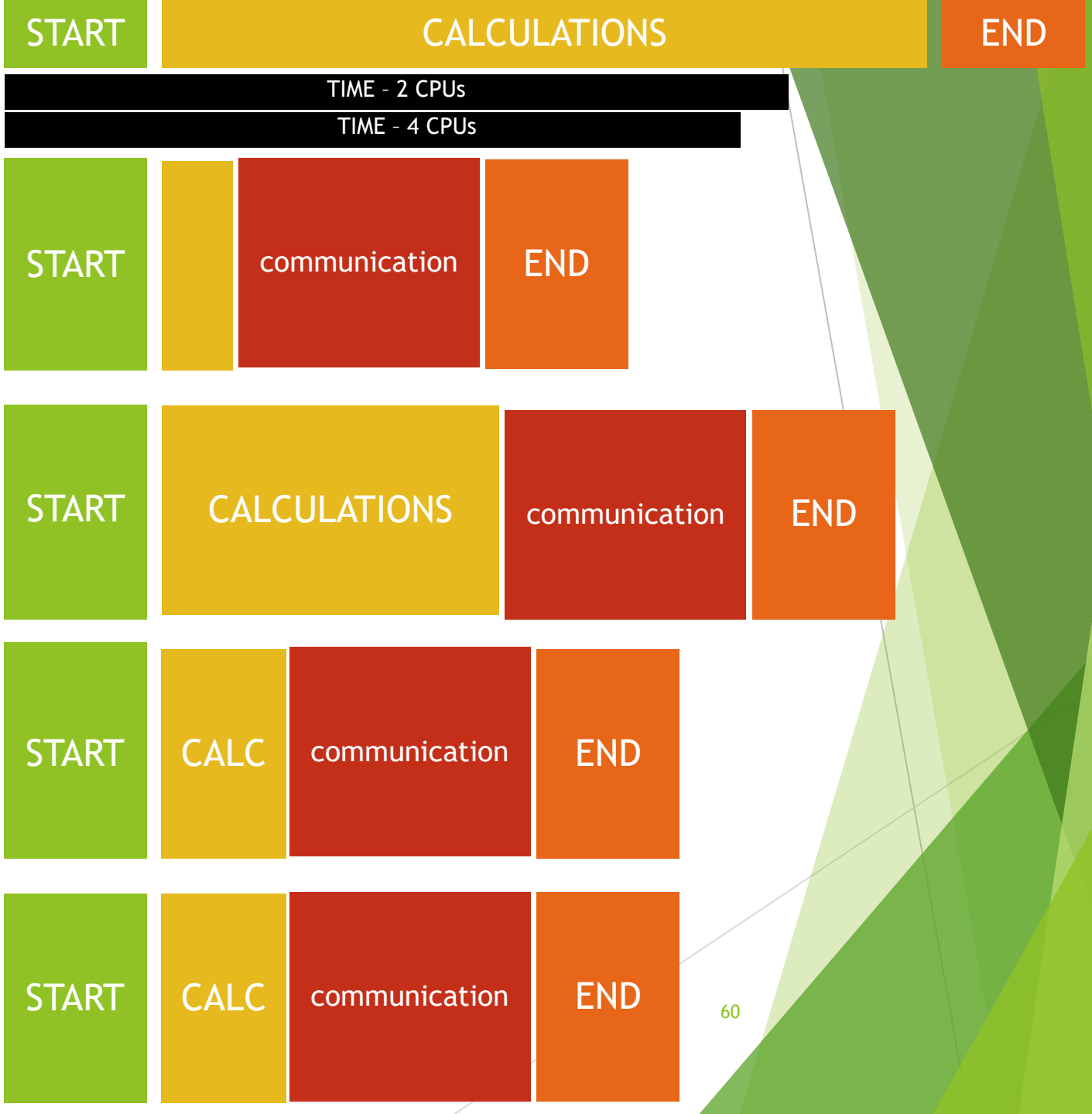
Time of calculations



Parallel integral

t =T/9

Time of calculations



Scaling blockers

- Work cannot be split into smaller pieces.
- Unbalanced workload splitting.
- Communication cost.