Parallel Programming in Python:
High-level Oomph

Jacob Wilkins

Scientific Computing Department, STFC

August 30, 2023

Overview

Introduction

Questions, questions, questions

- What is Python?

Questions, questions, questions

- What is Python?
- What is parallel?

Questions, questions, questions

- What is Python?
- What is parallel?
- How does it work?

Questions, questions, questions

- What is Python?
- What is parallel?
- How does it work?
- How to make the most of it.

Questions, questions, questions

- What is Python?
- What is parallel?
- How does it work?
- How to make the most of it.
- Not intended as introduction to parallel programming!

Short Snakes

- Python is a high-level interpreted programming language

Short Snakes

- Python is a high-level interpreted programming language
- Most versions run through C (CPython, not Cython)

Short Snakes

- Python is a high-level interpreted programming language
- Most versions run through C (CPython, not Cython)
- In general, slow!

Short Snakes

- Python is a high-level interpreted programming language
- Most versions run through C (CPython, not Cython)
- In general, slow!
- Use libraries to get real performance.

Snakes are fish now

- Python has a GIL

Snakes are fish now

- Python has a GIL
- Global-interpreter lock

Snakes are fish now

- Python has a GIL
- Global-interpreter lock
- No Python process can desynchronise from any other

Snakes are fish now

- Python has a GIL
- Global-interpreter lock
- No Python process can desynchronise from any other
- Efforts to remove GIL have been ongoing for years

Snakes are fish now

- Python has a GIL
- Global-interpreter lock
- No Python process can desynchronise from any other
- Efforts to remove GIL have been ongoing for years
- May finally be reaching fruition

Parallel

- What does it mean to do jobs in parallel?

IF	ID	EX	MEM	WB				
i	IF	ID	EX	MEM	WB			
t		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

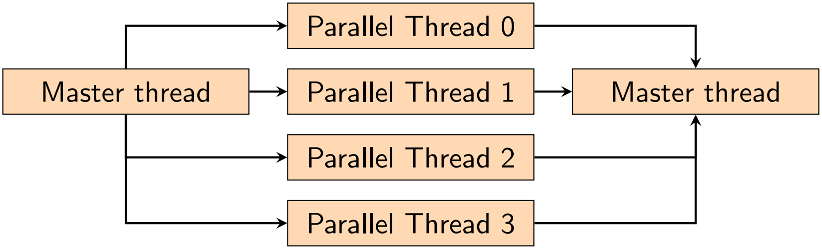
Parallel

- What does it mean to do jobs in parallel?

IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

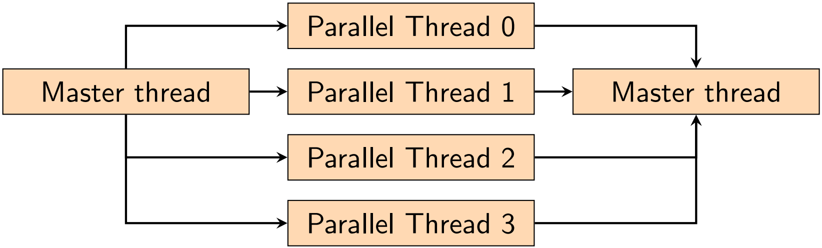
Fork-Join

- Fork-Join parallelism is temporary parallelism.



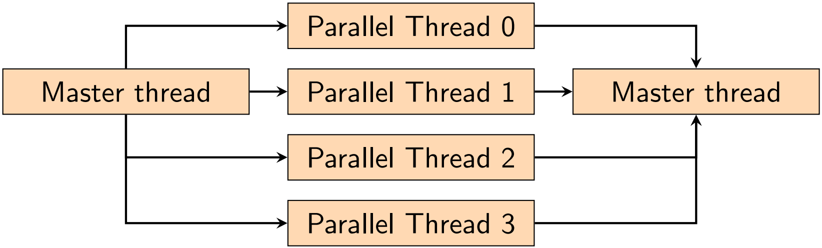
Fork-Join

- Fork-Join parallelism is temporary parallelism.
- Takes time to start/stop threads (sometimes leave them running).



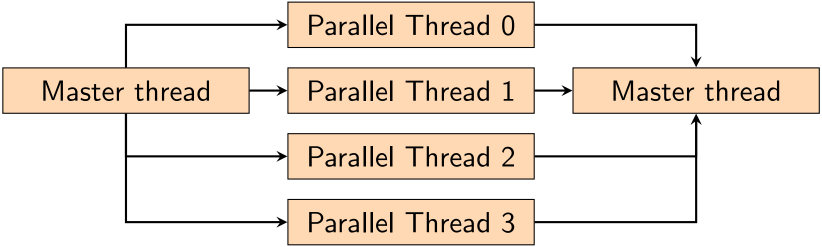
Fork-Join

- Fork-Join parallelism is temporary parallelism.
- Takes time to start/stop threads (sometimes leave them running).
- Often simpler to deal with for small regions.



Fork-Join

- Fork-Join parallelism is temporary parallelism.
- Takes time to start/stop threads (sometimes leave them running).
- Often simpler to deal with for small regions.
- Often less efficient.



Independent

- Independent parallelism has full independent threads.

Independent

- Independent parallelism has full independent threads.
- Always parallel, need ground-up supporting structure.

Independent

- Independent parallelism has full independent threads.
- Always parallel, need ground-up supporting structure.
- Not in sync, running own copies of codes.

Independent

- Independent parallelism has full independent threads.
- Always parallel, need ground-up supporting structure.
- Not in sync, running own copies of codes.
- Can be periodically forcibly synchronised.

Shared Memory

- All processes see the same memory

Shared Memory

- All processes see the same memory
- All processes can change the same memory

Shared Memory

- All processes see the same memory
- All processes can change the same memory
- Race conditions are a risk

Shared Memory

- All processes see the same memory
- All processes can change the same memory
- Race conditions are a risk
- Race condition is where 2 threads try to access same memory at once

Distributed Memory

- All processors have their own memory

Distributed Memory

- All processors have their own memory
- Need to specify which memory to transfer

Distributed Memory

- All processors have their own memory
- Need to specify which memory to transfer
- Some systems allow memory windows

Basic Optimisation

Using Generators

- Avoid computing things you aren't going to use

Using Generators

- Avoid computing things you aren't going to use
- Generators allow infinite loops

Using Generators

- Avoid computing things you aren't going to use
- Generators allow infinite loops
- Functions only calculated as needed

Using Generators

- Avoid computing things you aren't going to use
- Generators allow infinite loops
- Functions only calculated as needed
- Maps and filters can apply sequentially

Feeling lazy

```
def my_simple_counter(n: int = 0):
    """Count infinitely"""
    while true:
        yield n
        n = n+1

for i in my_simple_counter(15):
    print(i) # 15, 16, 17, ...
```

Using C

- Simple comprehensions run in C

Using C

- Simple comprehensions run in C
- Efficient clean use of these can speed up code

Using C

- Simple comprehensions run in C
- Efficient clean use of these can speed up code
- Also implement filters and transformations

No comprende

```python
# List comprehensions
[3*x for x in range(10) if x % 2]
# Set comprehensions
{3*x for x in range(10) if x % 2}
# Dict comprehensions
{key: 3*val for key, val in zip("abcdefghij",
    range(10)) if val % 2}
# Generator comprehension
(3*x for x in range(10) if x % 2)
# Using generator comprehensions
tuple(3*x for x in range(10) if x % 2)
"\t".join(str(3*x) for x in range(10) if x % 2)
```

Vectorisation

Numpy

- Numpy provides free parallelism out of the gate

```
import numpy as np

x = np.fromiter(range(1000))
print(x + np.ones(1000))

y = np.sin(x)
```

Numpy

- Numpy provides free parallelism out of the gate
- Numpy operations on arrays vectorised in C

```
import numpy as np

x = np.fromiter(range(1000))
print(x + np.ones(1000))

y = np.sin(x)
```

Numpy

- Numpy provides free parallelism out of the gate
- Numpy operations on arrays vectorised in C
- Numpy ufuncs operate efficiently over arrays

```
import numpy as np

x = np.fromiter(range(1000))
print(x + np.ones(1000))

y = np.sin(x)
```

No, you func

```
import numpy as np

def add_two(x):
    "Add 2 to x"
    return x + 2

#              frompyfunc(func, nin, nout)
my_ufunc = np.frompyfunc(add_two, 1, 1)
print(my_ufunc(np.array([3, 4, 5]))) # => [5 6 7]
```

Ufunc powers

- `__call__(*args, **kwargs)`
  Call self as a function.
- `accumulate(array[, axis, dtype, out])`
  Accumulate the result of applying the operator to all elements.
- `at(a, indices[, b])`
  Performs unbuffered in place operation on operand 'a' for elements specified by 'indices'.
- `outer(A, B, /, **kwargs)`
  Apply the ufunc op to all pairs (a, b) with a in A and b in B.
- `reduce(array[, axis, dtype, out, keepdims, ...])`
  Reduces array's dimension by one, by applying ufunc along one axis.
- `reduceat(array, indices[, axis, dtype, out])`
  Performs a (local) reduce with specified slices over a single axis.

Don't vectorise

- Don't use `numpy.vectorize`
  The vectorize function is provided primarily for convenience, not for performance. The implementation is essentially a for loop.
  – Numpy Docs

https://numpy.org/doc/stable/reference/generated/numpy.vectorize.html

Threading

Threads in briefs

- Fork-join, shared memory model

Threads in briefs

- Fork-join, shared memory model
- Throttled by the GIL

Threads in briefs

- Fork-join, shared memory model
- Throttled by the GIL
- Mostly designed for background tasks where the main loop shouldn't freeze

Threads in briefs

- Fork-join, shared memory model
- Throttled by the GIL
- Mostly designed for background tasks where the main loop shouldn't freeze
- Think GUIs, IO operations, etc.

Threads in briefs

```python
from threading import Thread

# Declare the job and the arguments
p = Thread(target=f, args=('bob',))
# Starts a thread running the job - Fork
p.start()
# Wait until finished
p.join()
```

Threads in briefs

- In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation).
  – Python Docs

https://docs.python.org/3/library/threading.html

In Letters

- More like writing a note to yourself
- Do the job when the processor has free time

Be aware

- Need to be aware of “thread-safe” methods.
- Queues are useful for managing threads
- Threads are also handy for spawning other processes.

ASync

async

- In language native co-routines

async

- In language native co-routines
- Ask now, get later

async

- In language native co-routines
- Ask now, get later
- Similar to `threading`, but subtly different

async/await

- Two main keywords `async`, `await`
- Also involves the `asyncio` package.

```
import asyncio

async def sleepy():
    print("Yawn")
    asyncio.sleep(1)
    print("Awake")

# Runs in 1 second
asyncio.gather(sleepy(), sleepy(), sleepy())

# Takes all 3
for _ in range(3):
    sleepy()
```

async/await

- `async` attaches to `def`, `for` and `with`

```
async def g(x):
    return 2*x

async def f(x):
    ans = await g(x)
    return ans
```

async/await

- `async` attaches to `def`, `for` and `with`
- `async` declares that object is to be scheduled in background.

```
async def g(x):
    return 2*x

async def f(x):
    ans = await g(x)
    return ans
```

async/await

- `async` attaches to `def`, `for` and `with`
- `async` declares that object is to be scheduled in background.
- `await` attaches to `async` functions

```
async def g(x):
    return 2*x

async def f(x):
    ans = await g(x)
    return ans
```

async/await

- `async` attaches to `def`, `for` and `with`
- `async` declares that object is to be scheduled in background.
- `await` attaches to `async` functions
- `await` says to pause execution until answer returned

```
async def g(x):
    return 2*x

async def f(x):
    ans = await g(x)
    return ans
```

await as gen

- Can help to think in terms of generators
- `async` declares object whose value is not computed yet
- `await` acts like `yield from` for `async`

Async objects

- Can declare object as `awaitable`
- Add `__await__` method
- Must return iterator object

```
class MyObject:
    def __init__(self):
        ...

    def __await__(self):
        ...
        return iterator_object
```

Async generator

- Like normal generators
- Can be iterated over with `async for`
- **N.B.** this is not parallelism, merely allowing other operations to borrow the processor.

```
import asyncio

async def async_generator(n=0):
    while true:
        yield n
        n = n + 1
        asyncio.sleep(1)

async for i in async_generator(3):
    print(i)
```

Multiprocessing

Finally Parallel!

- Fork-join, shared memory model

```
from multiprocessing import Process

def f(name):
    print('hello', name)

# Declare the job and the arguments
p = Process(target=f, args=('bob',))
# Starts a thread running the job
p.start()
# Wait until finished
p.join()
```

Finally Parallel!

- Fork-join, shared memory model
- Actually sidesteps the GIL – Finally parallel

```python
from multiprocessing import Process

def f(name):
    print('hello', name)

# Declare the job and the arguments
p = Process(target=f, args=('bob',))
# Starts a thread running the job
p.start()
# Wait until finished
p.join()
```

Finally Parallel!

- Fork-join, shared memory model
- Actually sidesteps the GIL – Finally parallel
- Support for task based parallelism

```python
from multiprocessing import Process

def f(name):
    print('hello', name)

# Declare the job and the arguments
p = Process(target=f, args=('bob',))
# Starts a thread running the job
p.start()
# Wait until finished
p.join()
```

Multiprocessing shared memory

- Support for shared memory processing

```
from multiprocessing import Process, Value, Array

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

# Double
num = Value('d', 0.0)
# Integer
arr = Array('i', [4, 5, 6])
p = Process(target=f, args=(num, arr))
p.start()
p.join()
```

Swimming in threads

- Support for iterative parallelism

```
from multiprocessing import Pool

def add_two(x):
    return x + 2

# start 4 worker processes
with Pool(processes=4) as pool:

    x = pool.map(add_two, [4, 5, 6]) # 6, 7, 8
    y = pool.starmap(add, [(1, 2), (3, 4)] # 3, 7
```

Swimming in threads

- Support for iterative parallelism
- Threads are grouped into pools

```
from multiprocessing import Pool

def add_two(x):
    return x + 2

# start 4 worker processes
with Pool(processes=4) as pool:

    x = pool.map(add_two, [4, 5, 6]) # 6, 7, 8
    y = pool.starmap(add, [(1, 2), (3, 4)] # 3, 7
```

Swimming in threads

- Support for iterative parallelism
- Threads are grouped into pools
- Simplest way is to parallelise over array

```
from multiprocessing import Pool

def add_two(x):
    return x + 2

# start 4 worker processes
with Pool(processes=4) as pool:

    x = pool.map(add_two, [4, 5, 6]) # 6, 7, 8
    y = pool.starmap(add, [(1, 2), (3, 4)] # 3, 7
```

MPI4Py

MPI-ython

- MPI is a standard for parallel computing

```
from mpi4py import MPI

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

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

MPI-ython

- MPI is a standard for parallel computing
- Originally designed to enable parallel computing in languages such as C and Fortran

```
from mpi4py import MPI

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

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

MPI-ython

- MPI is a standard for parallel computing
- Originally designed to enable parallel computing in languages such as C and Fortran
- Since it exists in C, is available in Python

```python
from mpi4py import MPI

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

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

- MPI is a standard for parallel computing
- Originally designed to enable parallel computing in languages such as C and Fortran
- Since it exists in C, is available in Python
- `mpi4py` is that interface to underlying C

```python
from mpi4py import MPI

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

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

MPI-ython

- Independent, distributed memory model

```
>> mpirun -np 4 python parallel_python.py
Hello from process 3 of 4
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 2 of 4
```

MPI-ython

- Independent, distributed memory model
- Always parallel

```
>> mpirun -np 4 python parallel_python.py
 Hello from process 3 of 4
 Hello from process 0 of 4
 Hello from process 1 of 4
 Hello from process 2 of 4
```

MPI-ython

- Independent, distributed memory model
- Always parallel
  - Requires you to change how you think from the ground up

```
>> mpirun -np 4 python parallel_python.py
Hello from process 3 of 4
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 2 of 4
```

MPI-ython

- Independent, distributed memory model
- Always parallel
  - Requires you to change how you think from the ground up
- Effective up to thousands of cores in principle

```
>> mpirun -np 4 python parallel_python.py
Hello from process 3 of 4
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 2 of 4
```

MPI-ython

- Independent, distributed memory model
- Always parallel
  - Requires you to change how you think from the ground up
- Effective up to thousands of cores in principle
- Each process has its own memory pool

```
>> mpirun -np 4 python parallel_python.py
Hello from process 3 of 4
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 2 of 4
```

MPI-ython

- Independent, distributed memory model
- Always parallel
  - Requires you to change how you think from the ground up
- Effective up to thousands of cores in principle
- Each process has its own memory pool
- Need to send all data to other processes manually and keep track yourself.

```
>> mpirun -np 4 python parallel_python.py
 Hello from process 3 of 4
 Hello from process 0 of 4
 Hello from process 1 of 4
 Hello from process 2 of 4
```

Communicators

- Fundamental object in MPI4py is the communicator.
- Communicator groups and labels processors.
- Enables transfer of data between them.
- `mpi4py.MPI.COMM_WORLD` is default.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD

# Create new communicator for procs (1,2),(3,4)
colour = int(comm.rank < 3) # Group to be in
key = comm.rank              # Rank order on comm
new_comm = comm.split(colour, key)

# Create a 2x2 non-periodic grid of processors
comm.Create_cart([2,2], periods=[False, False])
```

What's the point(-to-point)?

- Point-to-point communications specify source and destination
- Collective communications specify source
- Collectives send/receive from all in group
- MPI4py compatible with numpy
- Cartesian communicators allow N-D destinations

```
# Send from rank -> rank + 1 (next in series)
comm.send(3, dest=(rank+1)%size)

# Split array and send equal portions to each
    processor in group
recv = np.zeros(50)
send = np.array(100)
send[0:50] = 1; send[50:] = 2;
comm.Scatter(send, recv, root=0)
```

MPI Differences

- Don't need to handle typing explicitly (uses pickle)
- Requests from asynchronous comms are objects
- `lowercase` methods transfer objects
- `Titlecase` methods transfer buffers (lower-level)

```
comm.bcast(data, root=0)
comm.Bcast((data, MPI.DOUBLE), root=0)

req = comm.isend(data, rank+1)
req.test()
req.wait()
```

MPI Features

- Works with latest MPI features:

MPI Features

- Works with latest MPI features:
  - MPI-IO

MPI Features

- Works with latest MPI features:
  - MPI-IO
  - One-sided communications (memory windows)

MPI Features

- Works with latest MPI features:
    - MPI-IO
    - One-sided communications (memory windows)
    - GPU Aware MPI

MPI Features

- Works with latest MPI features:
  - MPI-IO
  - One-sided communications (memory windows)
  - GPU Aware MPI
- Some Python specific:

MPI Features

- Works with latest MPI features:
  - MPI-IO
  - One-sided communications (memory windows)
  - GPU Aware MPI
- Some Python specific:
  - MPI Sub-Process spawning

MPI Features

- Works with latest MPI features:
  - MPI-IO
  - One-sided communications (memory windows)
  - GPU Aware MPI
- Some Python specific:
  - MPI Sub-Process spawning
  - SWIG

MPI Features

- Works with latest MPI features:
  - MPI-IO
  - One-sided communications (memory windows)
  - GPU Aware MPI
- Some Python specific:
  - MPI Sub-Process spawning
  - SWIG
  - F2Py

Tensor Computational Graph Libraries

TCGMs

- Provide high-level interfaces for complicated mathematical operations

TCGMs

- Provide high-level interfaces for complicated mathematical operations
- Will often compile (JIT) into other languages

TCGMs

- Provide high-level interfaces for complicated mathematical operations
- Will often compile (JIT) into other languages
- Often include support for GPUs

TCGMs

- Aesara (previously Theano)

TCGMs

- Aesara (previously Theano)
- PyTorch

TCGMs

- Aesara (previously Theano)
- PyTorch
- JAX(ish)

TCGMs

- Aesara (previously Theano)
- PyTorch
- JAX(ish)
- Tensorflow

TCGMs

- Aesara (previously Theano)
- PyTorch
- JAX(ish)
- Tensorflow
- Going to take a look at Aesara

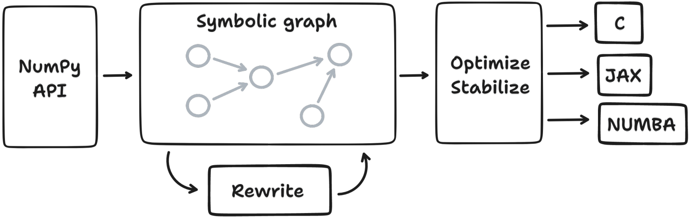
Aesara

- Aesara works by defining a sequence of functions on anonymous typed arguments
- Can then dispatch those functions as needed to different backends
- Designed to be generic and able to operate generally

```
import aesara
from aesara import tensor as at

# Declare two symbolic floating-point scalars
a = at.dscalar("a")
b = at.dscalar("b")

# Create a simple example expression
c = a + b

# Convert the expression into a callable object
    that takes `(a, b)`
# values as input and computes the value of `c`.
f_c = aesara.function([a, b], c)

assert f_c(1.5, 2.5) == 4.0
```

Aesara

- Builds a graph of operations
- Attempts to optimise and reduce graph
- Compiles resultant graph returning accessible function
- Compiled function dispatched to “processor”
- “Processor” may be CPU, GPU, TPU, or more



https://github.com/aesara-devs/aesara/

TCGM

- In principle saves you the work of: Optimisation, Ordering, Dispatching

```
d = a/a + (M + a).dot(v)
aesara.dprint(d)
# Elemwise{add,no_inplace} [id A] ''
#  |InplaceDimShuffle{x} [id B] ''
#  | |Elemwise{true_divide,no_inplace} [id C] ''
#  |   |a [id D]
#  |   |a [id D]
#  |dot [id E] ''
#    |Elemwise{add,no_inplace} [id F] ''
#    | |M [id G]
#    | |InplaceDimShuffle{x,x} [id H] ''
#    |   |a [id D]
#    |v [id I]
```

TCGM

- In principle saves you the work of: Optimisation, Ordering, Dispatching

```
f_d = aesara.function([a, v, M], d)
# 'a/a' -> '1' and the dot product is replaced
    with a BLAS function
# Elemwise{Add}[(0, 1)] [id A] ''   5
#  |TensorConstant{(1,) of 1.0} [id B]
#  |CGemv{inplace} [id C] ''   4
#    |AllocEmpty{dtype='float64'} [id D] ''   3
#    | |Shape_i{0} [id E] ''   2
#    |   |M [id F]
#    |TensorConstant{1.0} [id G]
#    |Elemwise{add,no_inplace} [id H] ''   1
#    | |M [id F]
#    | |InplaceDimShuffle{x,x} [id I] ''   0
#    |   |a [id J]
#    |v [id K]
#    |TensorConstant{0.0} [id L]
```

TCGM

- Because these have access to (or are) the graph
- Can manipulate the graph constructively
- Can compute gradients or inverses

```
x = at.dmatrix('x')
# Logistic equation
s = at.sum(1 / (1 + at.exp(-x)))
gs = at.grad(s, x)

# Now have a function for derivative of logistic
dlogistic = aesara.function([x], gs)
```

Summary

Summary

- Numpy – Parallel(ish)

Summary

- Numpy – Parallel(ish)
- Threading – Not parallel

Summary

- Numpy – Parallel(ish)
- Threading – Not parallel
- Async – Not parallel

Summary

- Numpy – Parallel(ish)
- Threading – Not parallel
- Async – Not parallel
- Multiprocessing – Parallel

Summary

- Numpy – Parallel(ish)
- Threading – Not parallel
- Async – Not parallel
- Multiprocessing – Parallel
- MPI4Py – Parallel

Summary

- Numpy – Parallel(ish)
- Threading – Not parallel
- Async – Not parallel
- Multiprocessing – Parallel
- MPI4Py – Parallel
- TCGM – Maybe parallel

Worst MD

- If you want to have a play there is a really bad Python MD code available at: https://github.com/oerc0122/worst_md/blob/main/md.py