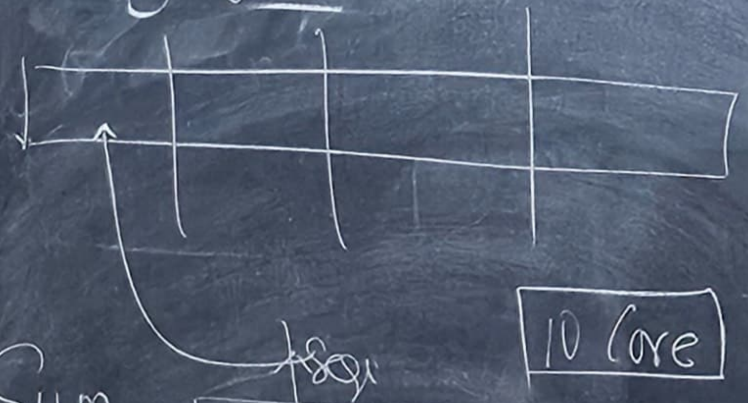
Array of int



Sum

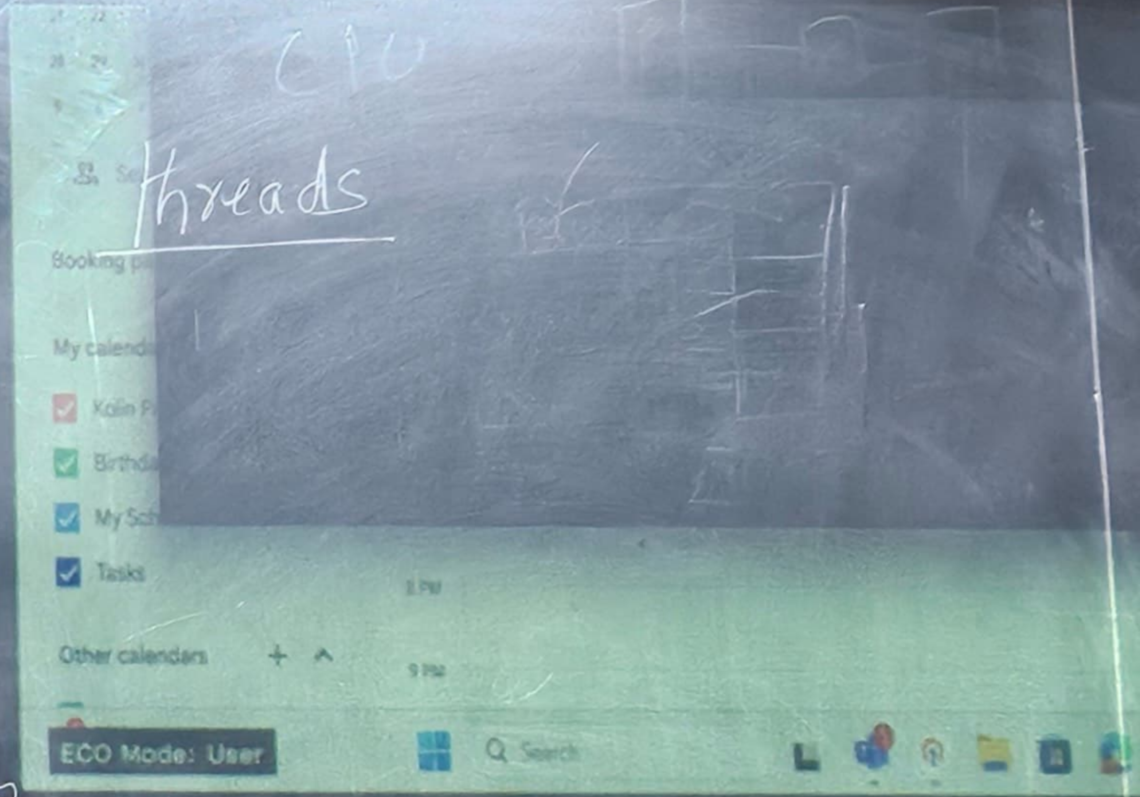
10 Core

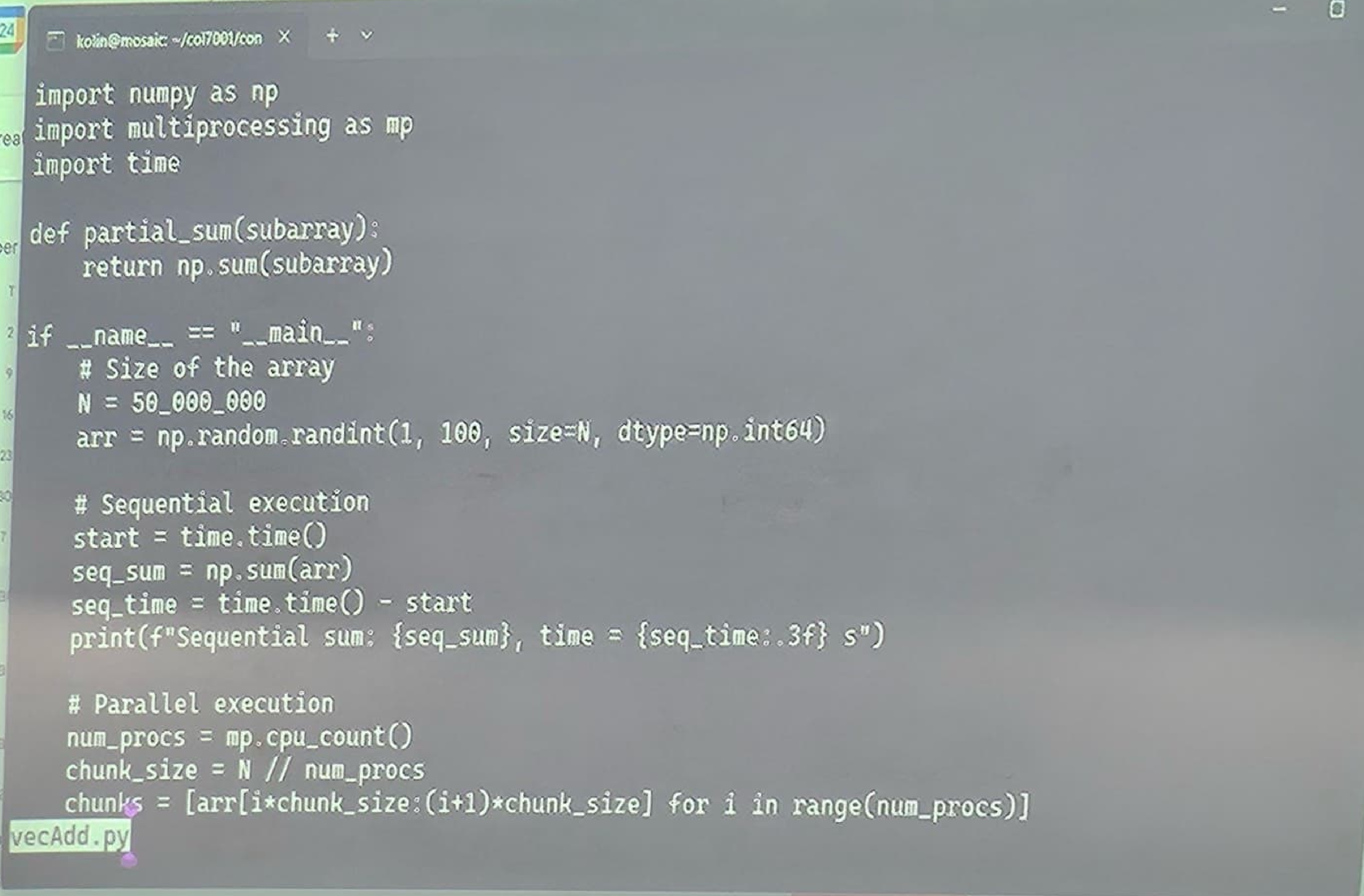
Partial Sum on Small Chunks

Reduct^n of Partial Sums

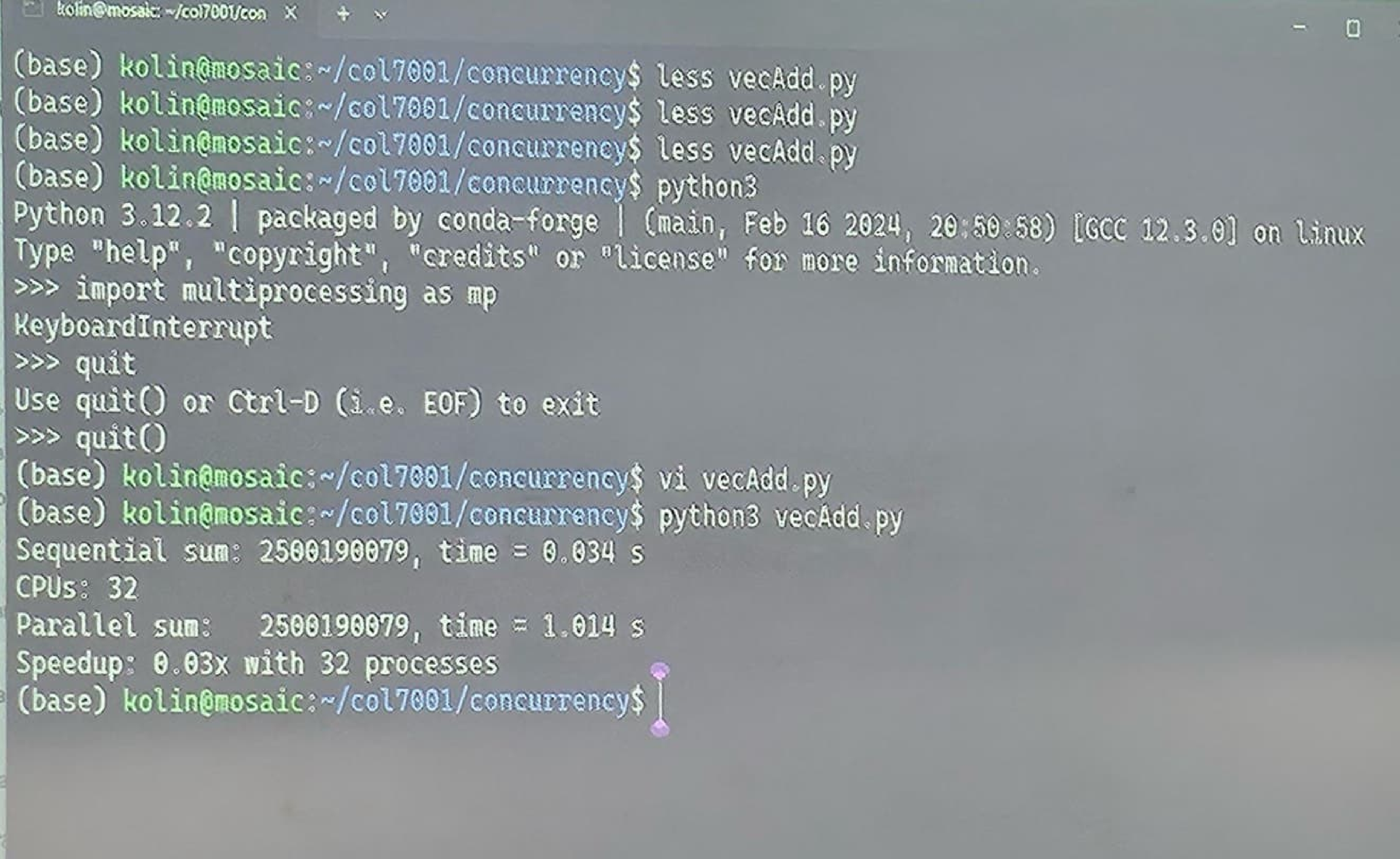
CPU

Threads

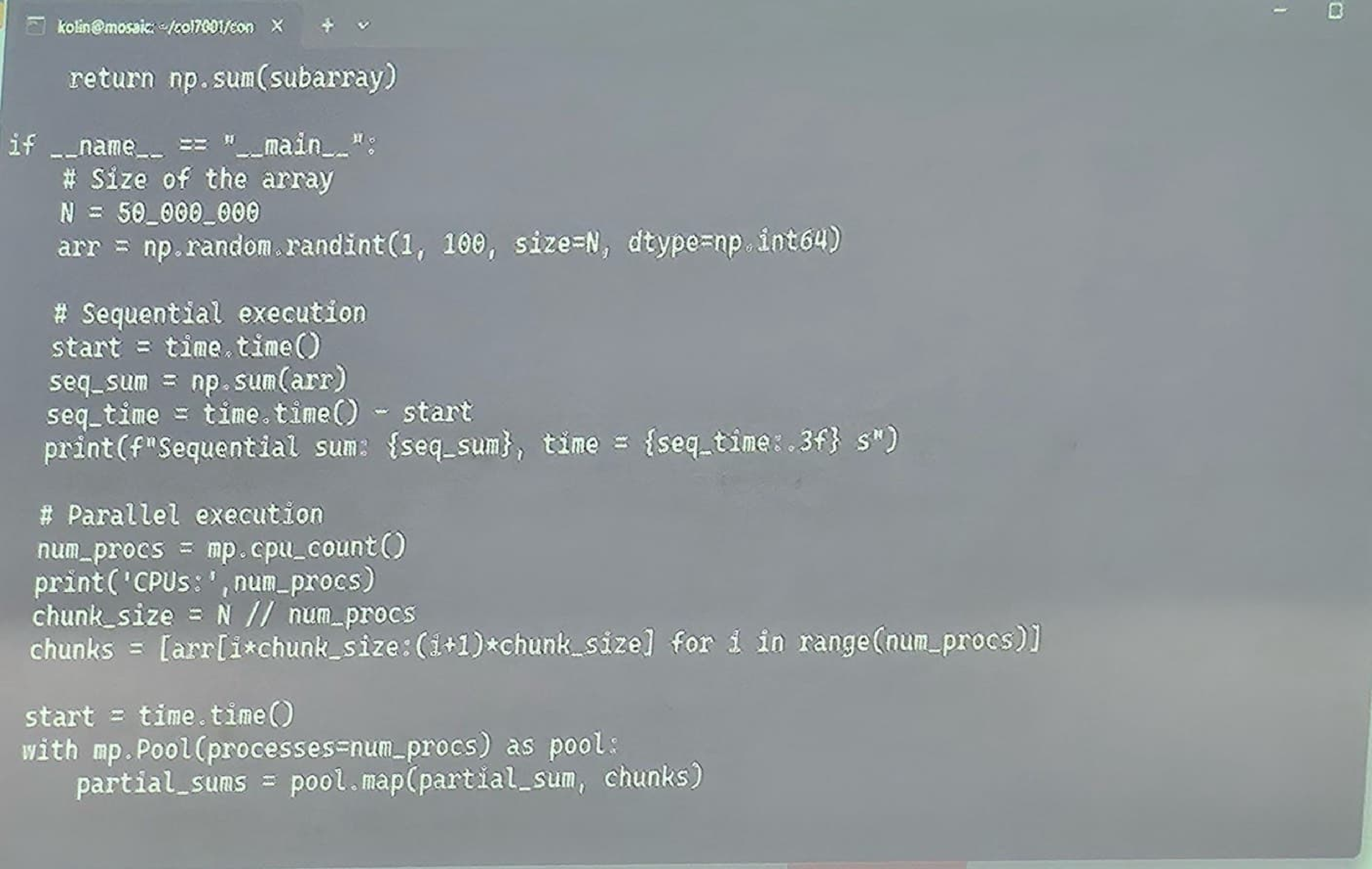




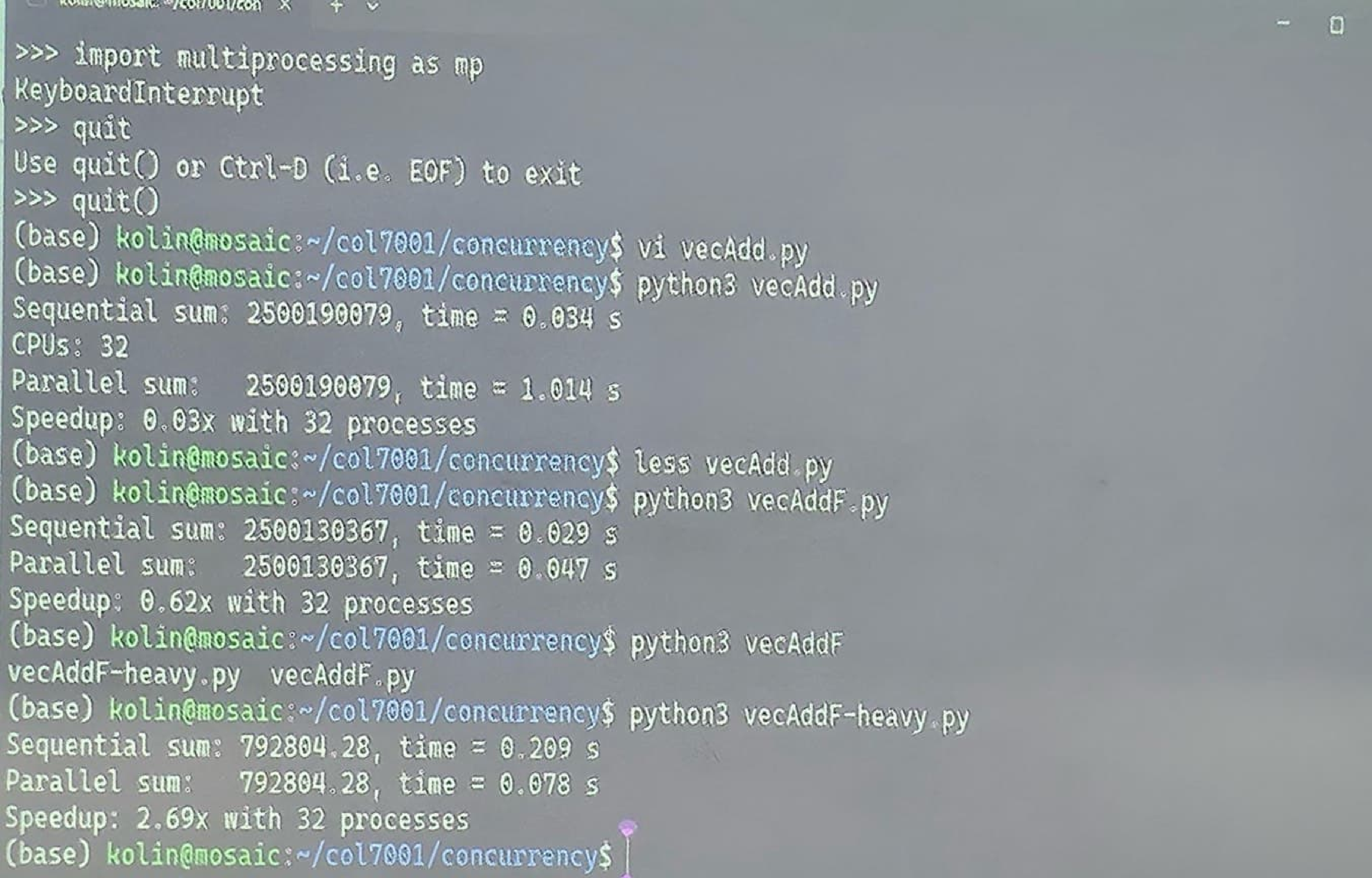
```
import numpy as np
import multiprocessing as mp
import time

def partial_sum(subarray):
    return np.sum(subarray)

if __name__ == "__main__":
    # Size of the array
    N = 50_000_000
    arr = np.random.randint(1, 100, size=N, dtype=np.int64)

    # Sequential execution
    start = time.time()
    seq_sum = np.sum(arr)
    seq_time = time.time() - start
    print(f"Sequential sum: {seq_sum}, time = {seq_time:.3f} s")

    # Parallel execution
    num_procs = mp.cpu_count()
    chunk_size = N // num_procs
    chunks = [arr[i*chunk_size:(i+1)*chunk_size] for i in range(num_procs)]
vecAdd.py
```



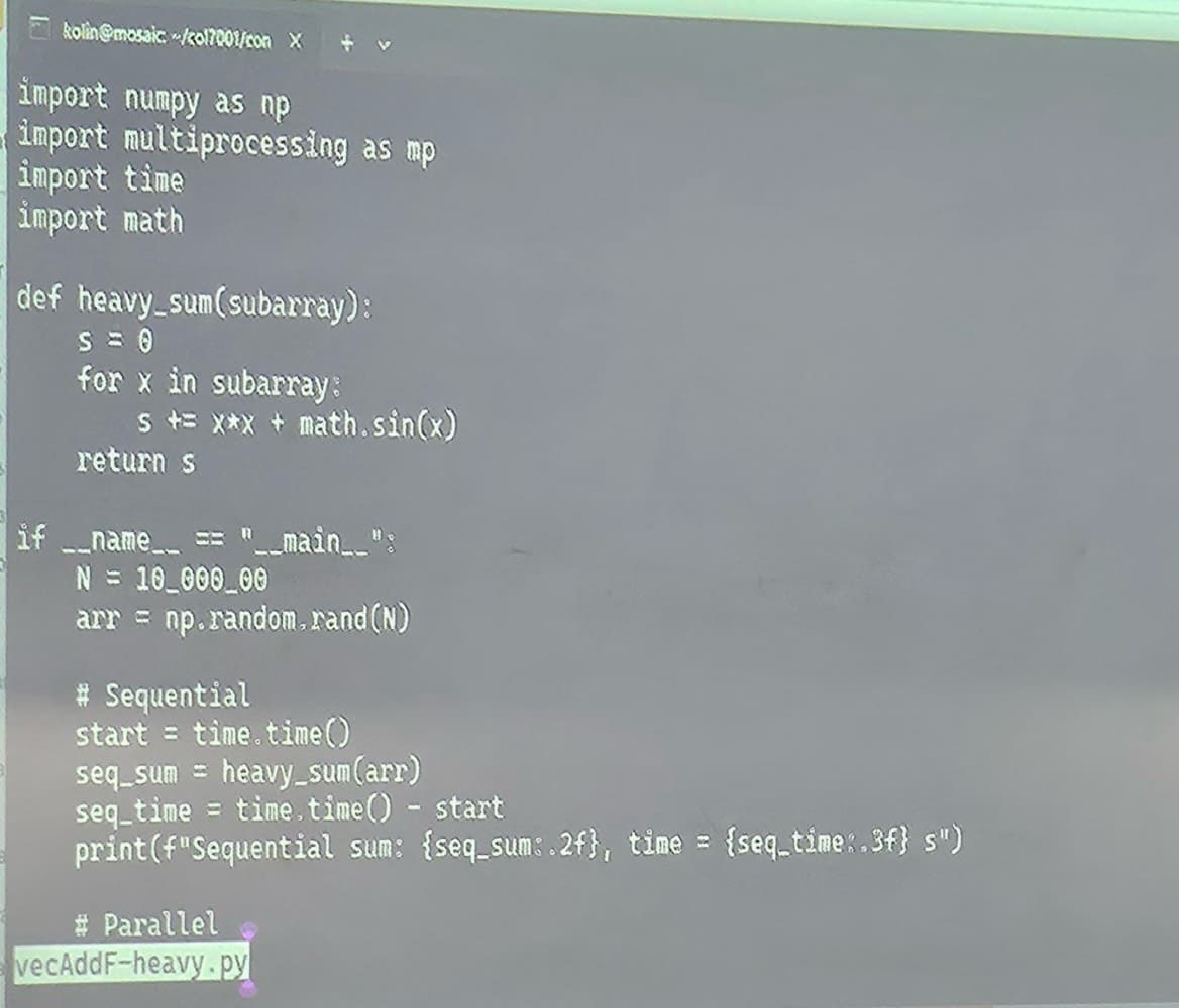
```
(base) kolin@mosaic:~/col7001/concurrency$ less vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ less vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ less vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ python3
Python 3.12.2 | packaged by conda-forge | (main, Feb 16 2024, 20:50:58) [GCC 12.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import multiprocessing as mp
KeyboardInterrupt
>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
>>> quit()
(base) kolin@mosaic:~/col7001/concurrency$ vi vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAdd.py
Sequential sum: 2500190079, time = 0.034 s
CPUs: 32
Parallel sum:   2500190079, time = 1.014 s
Speedup: 0.03x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$
```



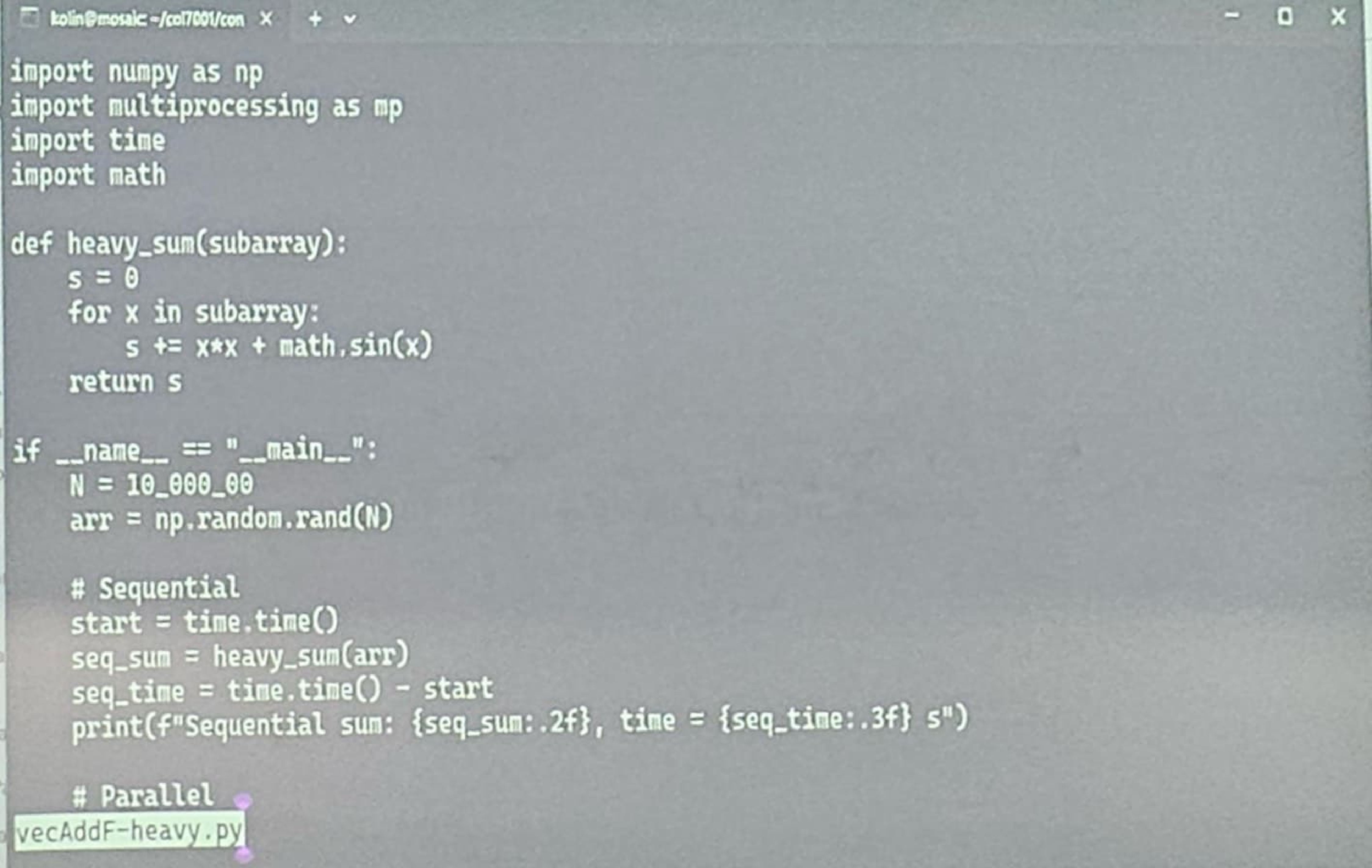
```
    return np.sum(subarray)

if __name__ == "__main__":
    # Size of the array
    N = 50_000_000
    arr = np.random.randint(1, 100, size=N, dtype=np.int64)

    # Sequential execution
    start = time.time()
    seq_sum = np.sum(arr)
    seq_time = time.time() - start
    print(f"Sequential sum: {seq_sum}, time = {seq_time:.3f} s")

    # Parallel execution
    num_procs = mp.cpu_count()
    print('CPUs:',num_procs)
    chunk_size = N // num_procs
    chunks = [arr[i*chunk_size:(i+1)*chunk_size] for i in range(num_procs)]

    start = time.time()
    with mp.Pool(processes=num_procs) as pool:
        partial_sums = pool.map(partial_sum, chunks)
```



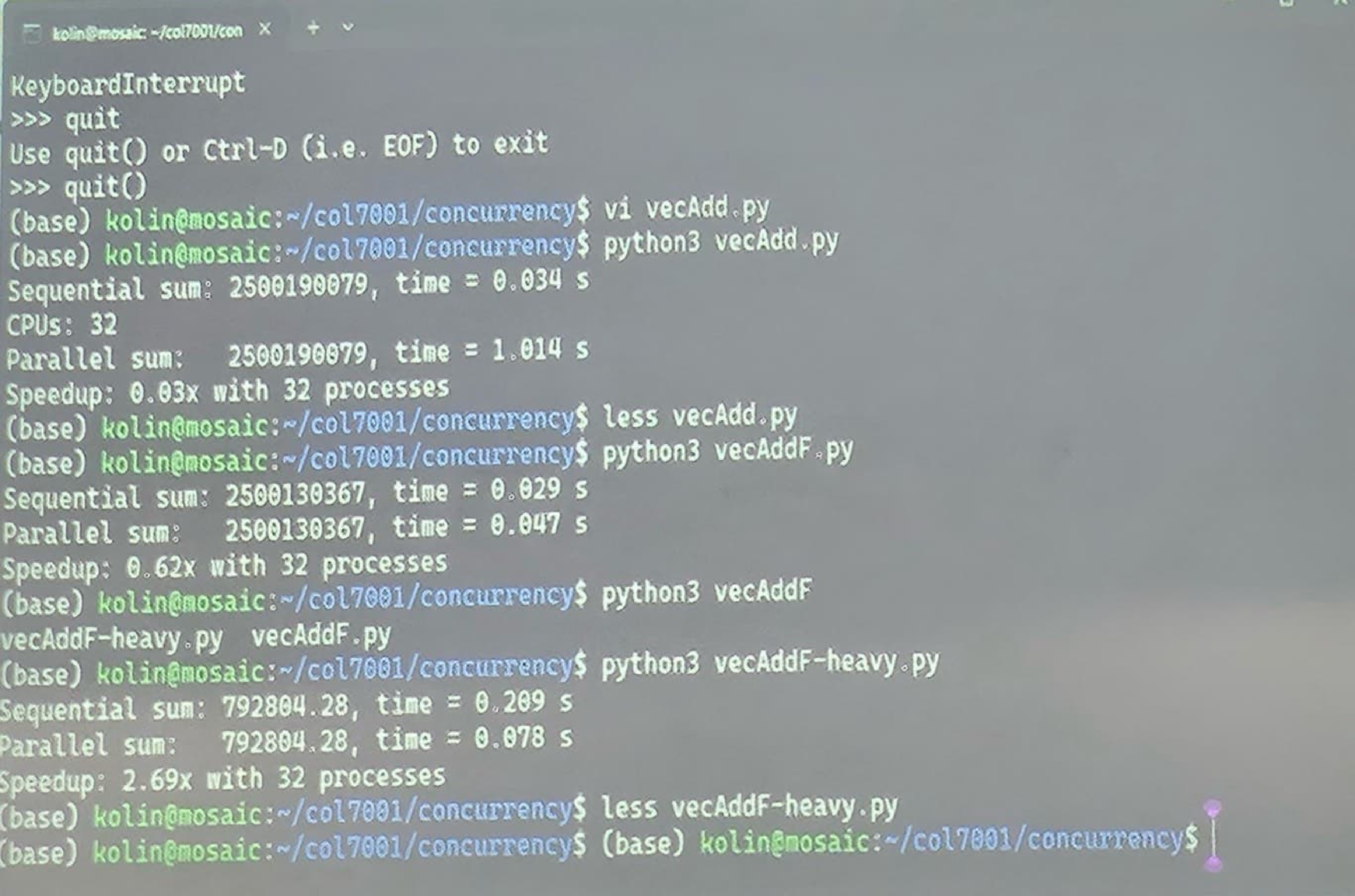
```
>>> import multiprocessing as mp
KeyboardInterrupt
>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
>>> quit()
(base) kolin@mosaic:~/col7001/concurrency$ vi vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAdd.py
Sequential sum: 2500190079, time = 0.034 s
CPUs: 32
Parallel sum:   2500190079, time = 1.014 s
Speedup: 0.03x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ less vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF.py
Sequential sum: 2500130367, time = 0.029 s
Parallel sum:   2500130367, time = 0.047 s
Speedup: 0.62x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF
vecAddF-heavy.py  vecAddF.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF-heavy.py
Sequential sum: 792804.28, time = 0.209 s
Parallel sum:   792804.28, time = 0.078 s
Speedup: 2.69x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$
```



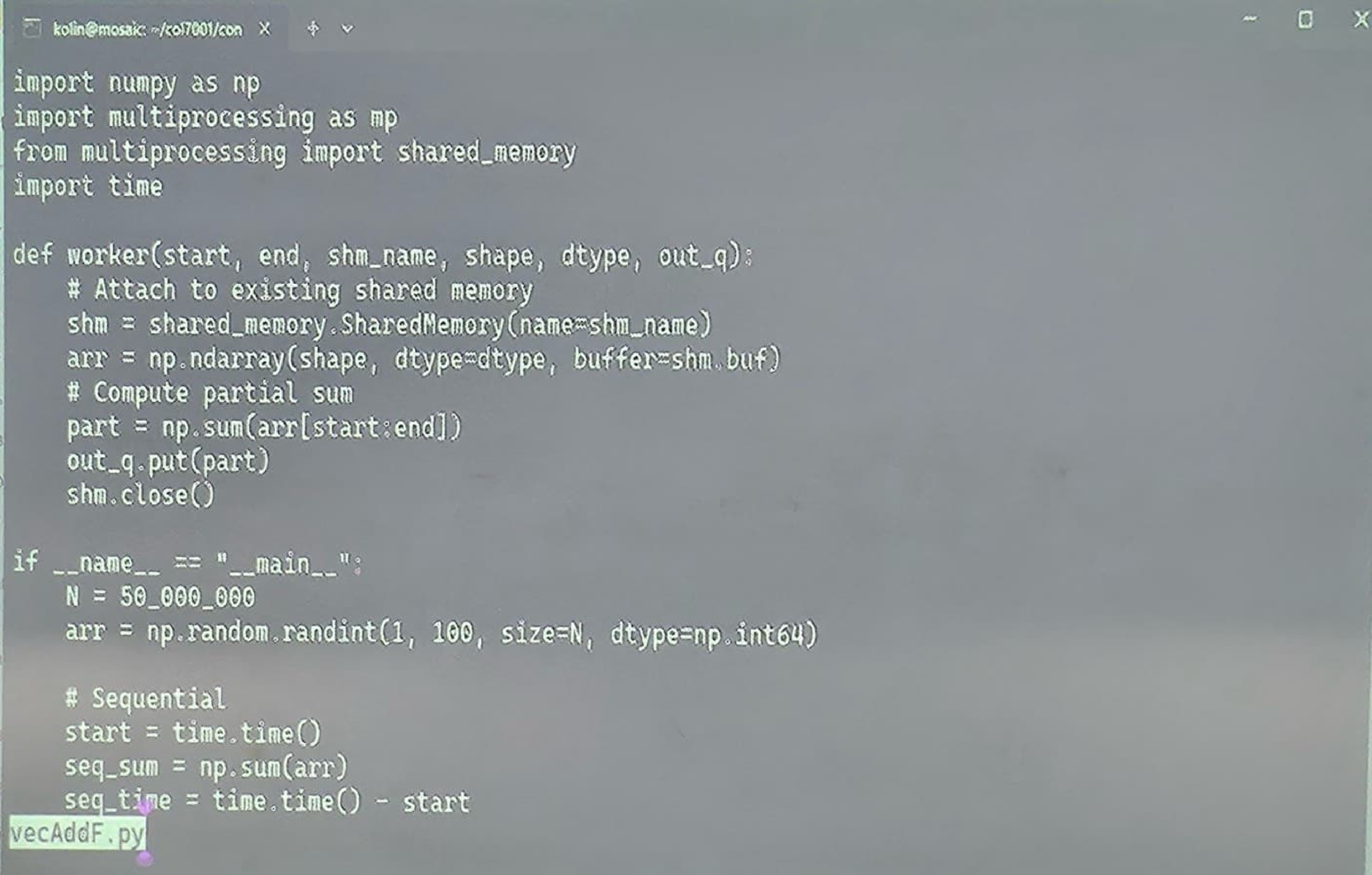
```
import numpy as np
import multiprocessing as mp
import time
import math

def heavy_sum(subarray):
    s = 0
    for x in subarray:
        s += x*x + math.sin(x)
    return s

if __name__ == "__main__":
    N = 10_000_00
    arr = np.random.rand(N)

    # Sequential
    start = time.time()
    seq_sum = heavy_sum(arr)
    seq_time = time.time() - start
    print(f"Sequential sum: {seq_sum:.2f}, time = {seq_time:.3f} s")

    # Parallel
vecAddF-heavy.py
```



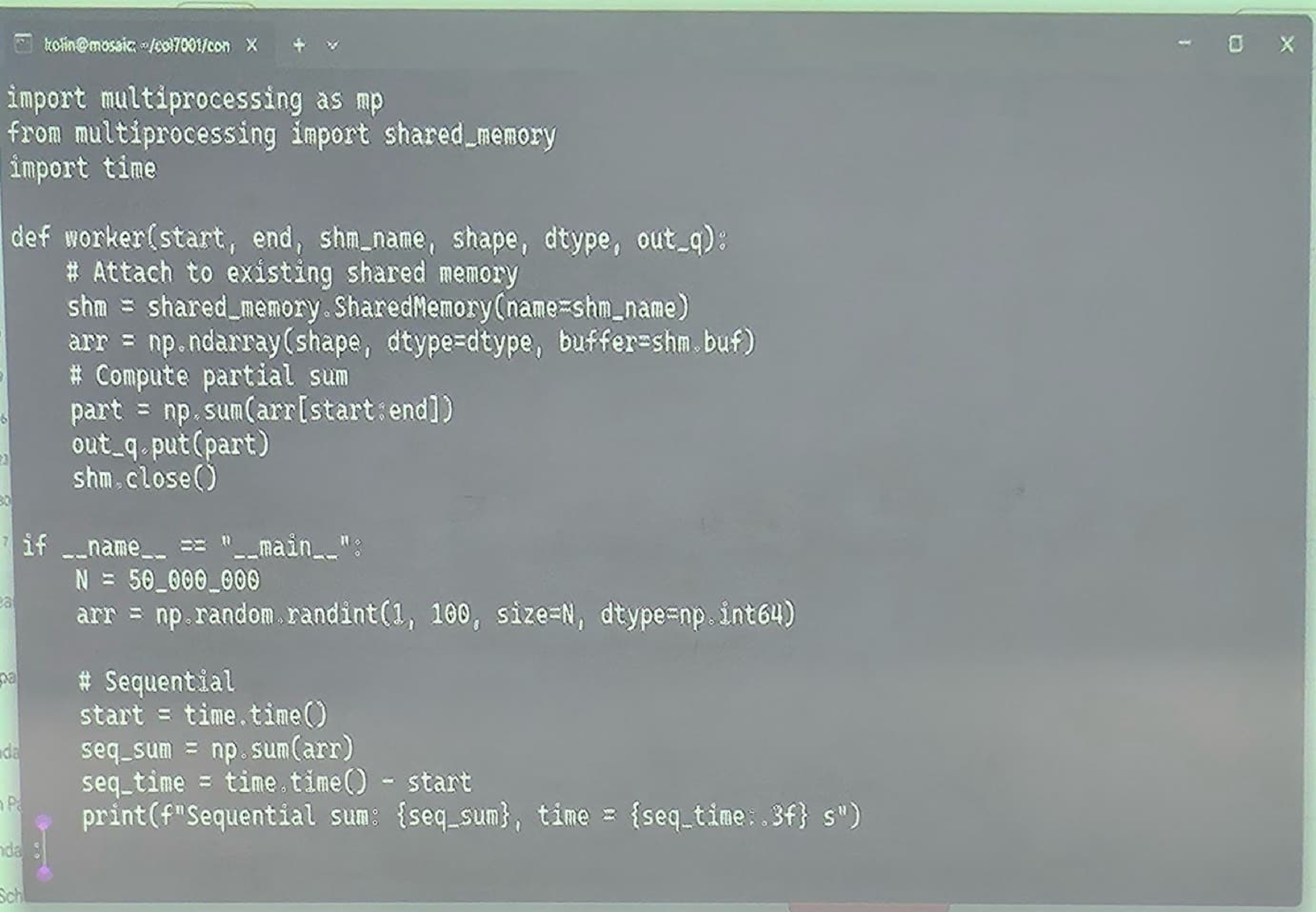
```
import numpy as np
import multiprocessing as mp
import time
import math

def heavy_sum(subarray):
    s = 0
    for x in subarray:
        s += x*x + math.sin(x)
    return s

if __name__ == "__main__":
    N = 10_000_00
    arr = np.random.rand(N)

    # Sequential
    start = time.time()
    seq_sum = heavy_sum(arr)
    seq_time = time.time() - start
    print(f"Sequential sum: {seq_sum:.2f}, time = {seq_time:.3f} s")

    # Parallel
vecAddF-heavy.py
```



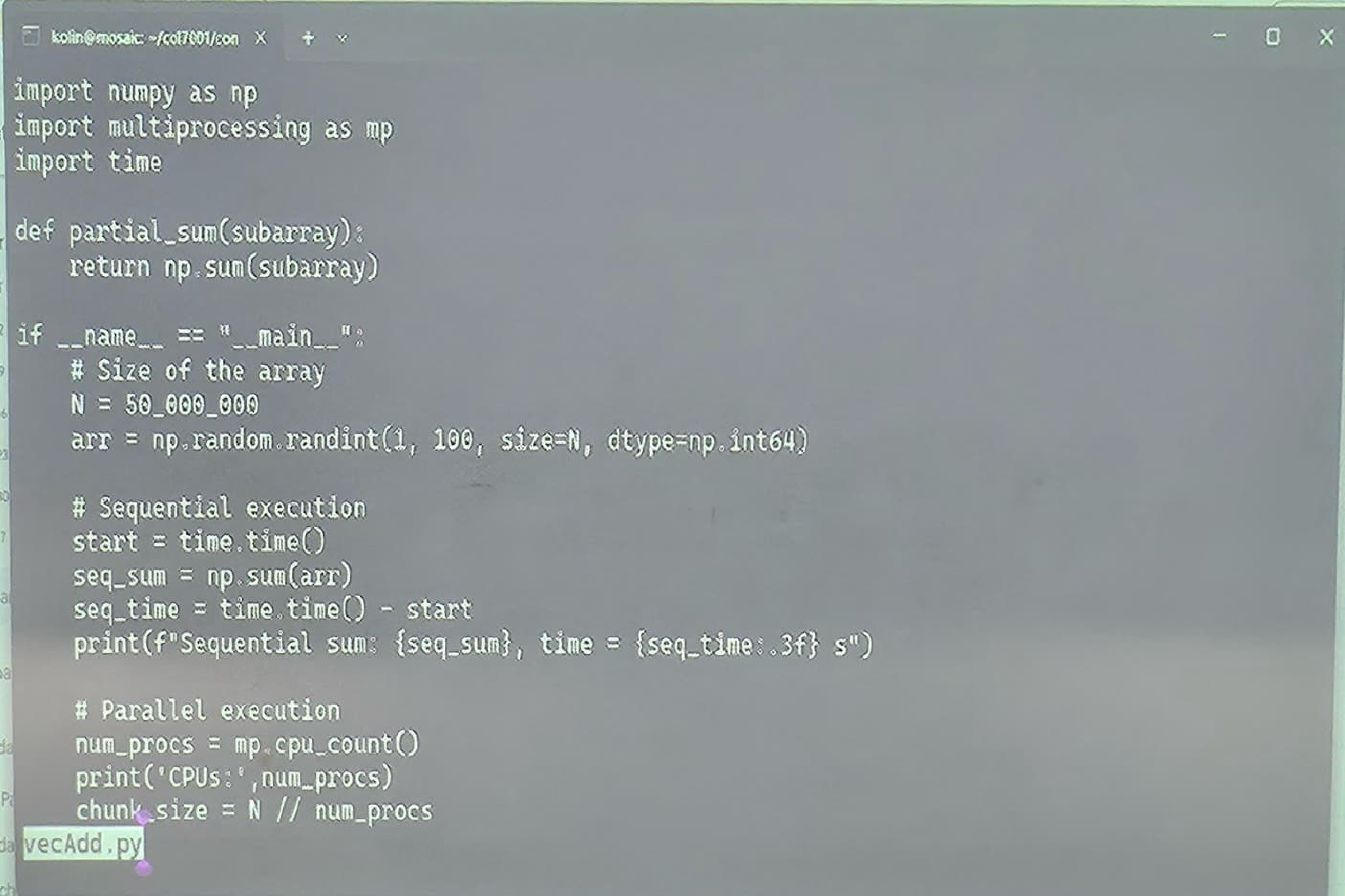
```
KeyboardInterrupt
>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
>>> quit()
(base) kolin@mosaic:~/col7001/concurrency$ vi vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAdd.py
Sequential sum: 2500190079, time = 0.034 s
CPUs: 32
Parallel sum:   2500190079, time = 1.014 s
Speedup: 0.03x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ less vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF.py
Sequential sum: 2500130367, time = 0.029 s
Parallel sum:   2500130367, time = 0.047 s
Speedup: 0.62x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF
vecAddF-heavy.py  vecAddF.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF-heavy.py
Sequential sum: 792804.28, time = 0.209 s
Parallel sum:   792804.28, time = 0.078 s
Speedup: 2.69x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ less vecAddF-heavy.py
(base) kolin@mosaic:~/col7001/concurrency$ (base) kolin@mosaic:~/col7001/concurrency$
```



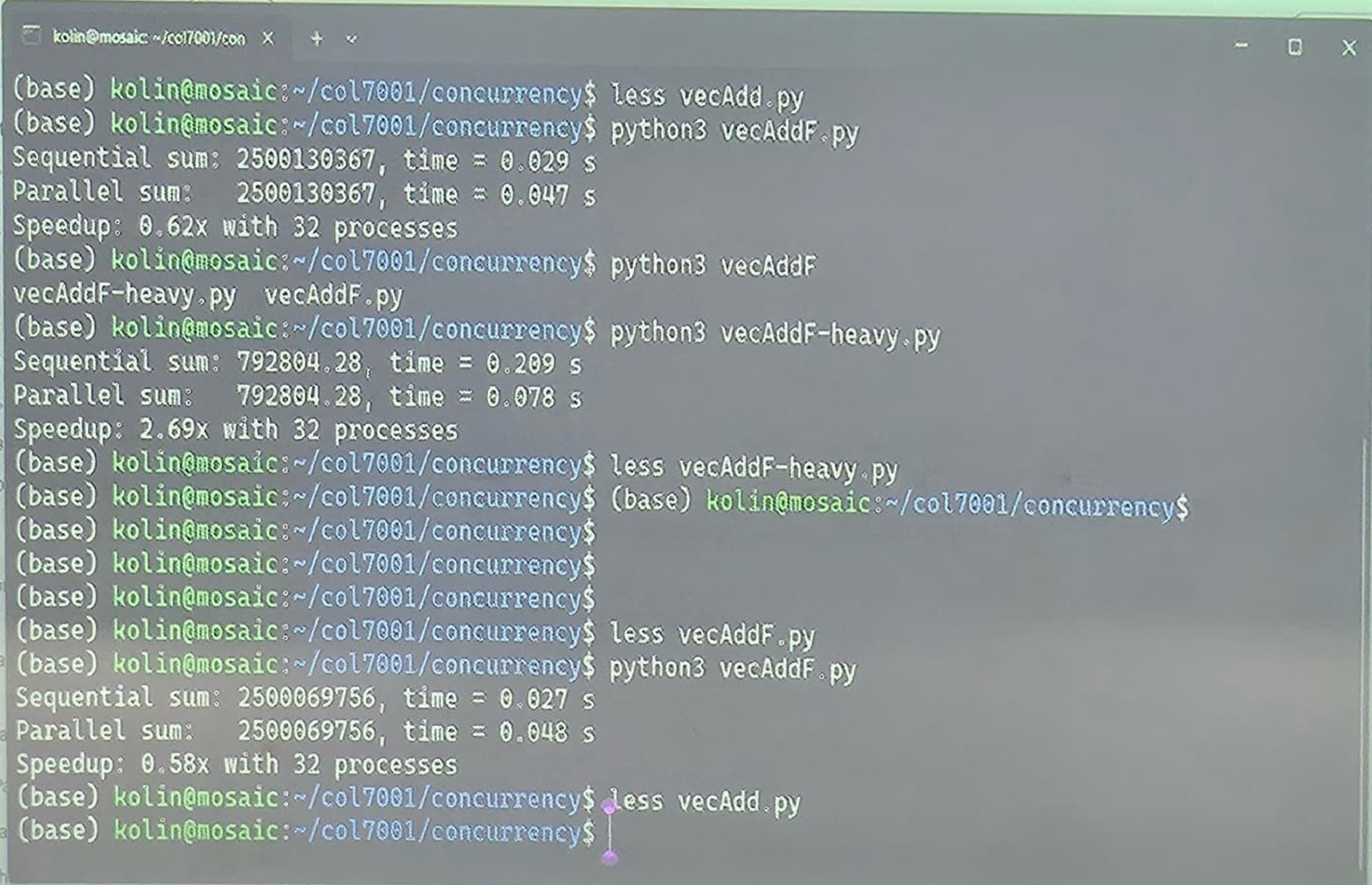
```
import numpy as np
import multiprocessing as mp
from multiprocessing import shared_memory
import time

def worker(start, end, shm_name, shape, dtype, out_q):
    # Attach to existing shared memory
    shm = shared_memory.SharedMemory(name=shm_name)
    arr = np.ndarray(shape, dtype=dtype, buffer=shm.buf)
    # Compute partial sum
    part = np.sum(arr[start:end])
    out_q.put(part)
    shm.close()

if __name__ == "__main__":
    N = 50_000_000
    arr = np.random.randint(1, 100, size=N, dtype=np.int64)

    # Sequential
    start = time.time()
    seq_sum = np.sum(arr)
    seq_time = time.time() - start
vecAddF.py
```



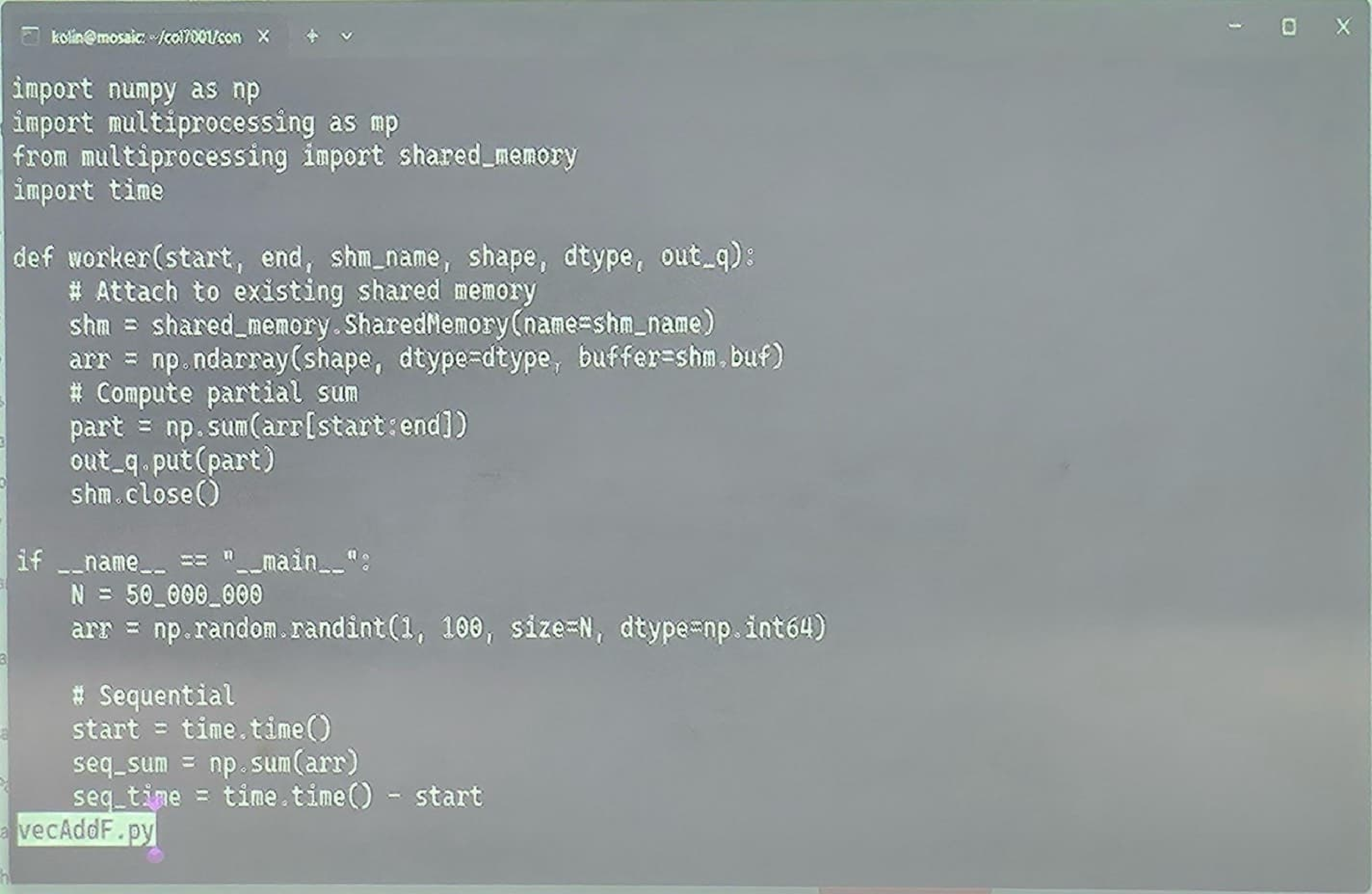
```
import multiprocessing as mp
from multiprocessing import shared_memory
import time

def worker(start, end, shm_name, shape, dtype, out_q):
    # Attach to existing shared memory
    shm = shared_memory.SharedMemory(name=shm_name)
    arr = np.ndarray(shape, dtype=dtype, buffer=shm.buf)
    # Compute partial sum
    part = np.sum(arr[start:end])
    out_q.put(part)
    shm.close()

if __name__ == "__main__":
    N = 50_000_000
    arr = np.random.randint(1, 100, size=N, dtype=np.int64)

    # Sequential
    start = time.time()
    seq_sum = np.sum(arr)
    seq_time = time.time() - start
    print(f"Sequential sum: {seq_sum}, time = {seq_time:.3f} s")
```



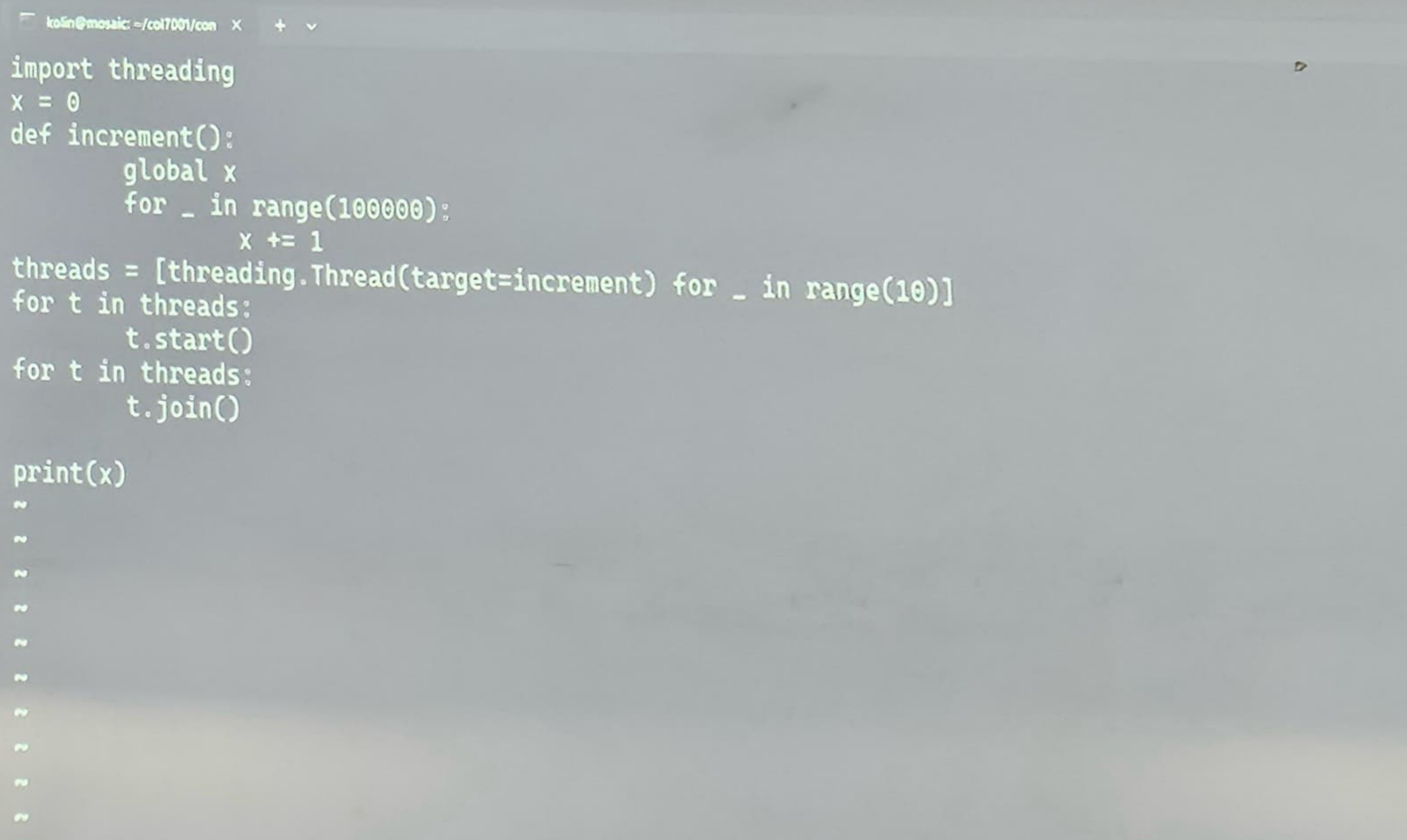
```
import numpy as np
import multiprocessing as mp
import time

def partial_sum(subarray):
    return np.sum(subarray)

if __name__ == "__main__":
    # Size of the array
    N = 50_000_000
    arr = np.random.randint(1, 100, size=N, dtype=np.int64)

    # Sequential execution
    start = time.time()
    seq_sum = np.sum(arr)
    seq_time = time.time() - start
    print(f"Sequential sum: {seq_sum}, time = {seq_time:.3f} s")

    # Parallel execution
    num_procs = mp.cpu_count()
    print('CPUs:',num_procs)
    chunk_size = N // num_procs
vecAdd.py
```



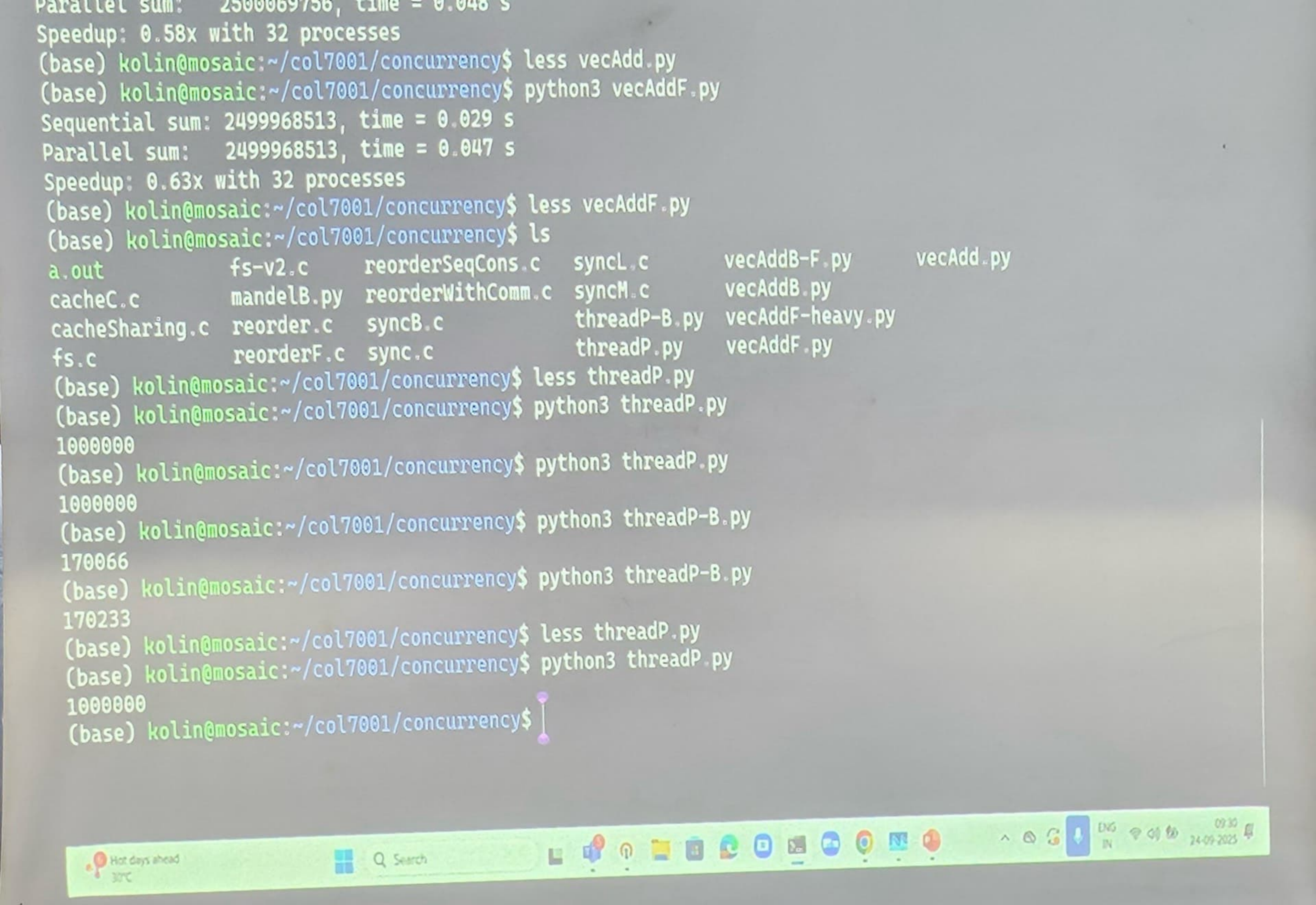
```
(base) kolin@mosaic:~/col7001/concurrency$ less vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF.py
Sequential sum: 2500130367, time = 0.029 s
Parallel sum:   2500130367, time = 0.047 s
Speedup: 0.62x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF
vecAddF-heavy.py  vecAddF.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF-heavy.py
Sequential sum: 792804.28, time = 0.209 s
Parallel sum:   792804.28, time = 0.078 s
Speedup: 2.69x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ less vecAddF-heavy.py
(base) kolin@mosaic:~/col7001/concurrency$ (base) kolin@mosaic:~/col7001/concurrency$
(base) kolin@mosaic:~/col7001/concurrency$
(base) kolin@mosaic:~/col7001/concurrency$
(base) kolin@mosaic:~/col7001/concurrency$
(base) kolin@mosaic:~/col7001/concurrency$ less vecAddF.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF.py
Sequential sum: 2500069756, time = 0.027 s
Parallel sum:   2500069756, time = 0.048 s
Speedup: 0.58x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ less vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$
```



```
import numpy as np
import multiprocessing as mp
from multiprocessing import shared_memory
import time

def worker(start, end, shm_name, shape, dtype, out_q):
    # Attach to existing shared memory
    shm = shared_memory.SharedMemory(name=shm_name)
    arr = np.ndarray(shape, dtype=dtype, buffer=shm.buf)
    # Compute partial sum
    part = np.sum(arr[start:end])
    out_q.put(part)
    shm.close()

if __name__ == "__main__":
    N = 50_000_000
    arr = np.random.randint(1, 100, size=N, dtype=np.int64)

    # Sequential
    start = time.time()
    seq_sum = np.sum(arr)
    seq_time = time.time() - start
vecAddF.py
```



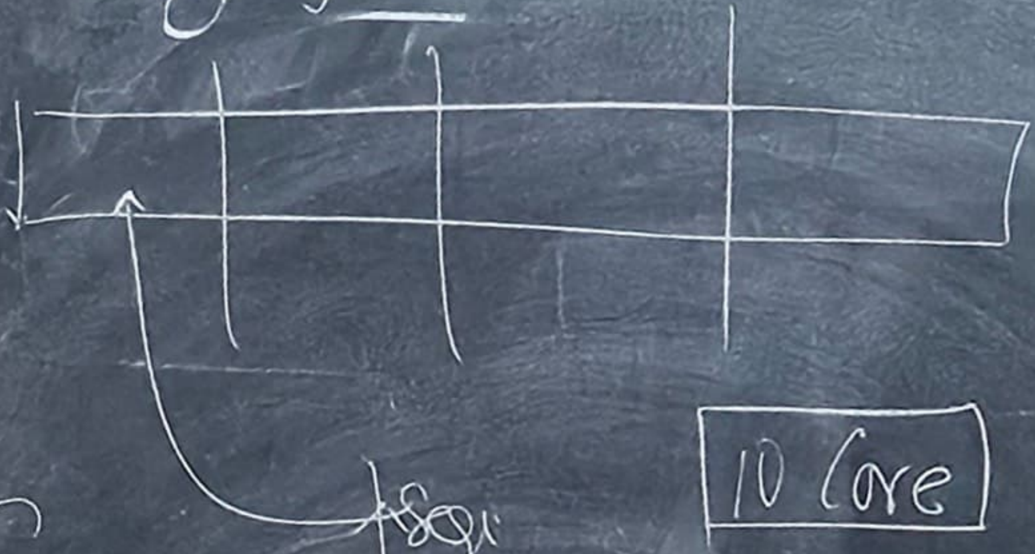
```
import threading
x = 0
def increment():
        global x
        for _ in range(100000):
                x += 1
threads = [threading.Thread(target=increment) for _ in range(10)]
for t in threads:
        t.start()
for t in threads:
        t.join()

print(x)
~
~
~
~
~
~
~
~
~
~
```



```
Parallel sum:   2500069756, time = 0.048 s
Speedup: 0.58x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ less vecAdd.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 vecAddF.py
Sequential sum: 2499968513, time = 0.029 s
Parallel sum:   2499968513, time = 0.047 s
Speedup: 0.63x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ less vecAddF.py
(base) kolin@mosaic:~/col7001/concurrency$ ls
a.out           fs-v2.c    reorderSeqCons.c   syncL.c        vecAddB-F.py      vecAdd.py
cacheC.c        mandelB.py reorderWithComm.c  syncM.c        vecAddB.py
cacheSharing.c  reorder.c  syncB.c            threadP-B.py   vecAddF-heavy.py
fs.c            reorderF.c sync.c             threadP.py     vecAddF.py
(base) kolin@mosaic:~/col7001/concurrency$ less threadP.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP.py
1000000
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP.py
1000000
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP-B.py
170066
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP-B.py
170233
(base) kolin@mosaic:~/col7001/concurrency$ less threadP.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP.py
1000000
(base) kolin@mosaic:~/col7001/concurrency$
```

GIL

Array of int

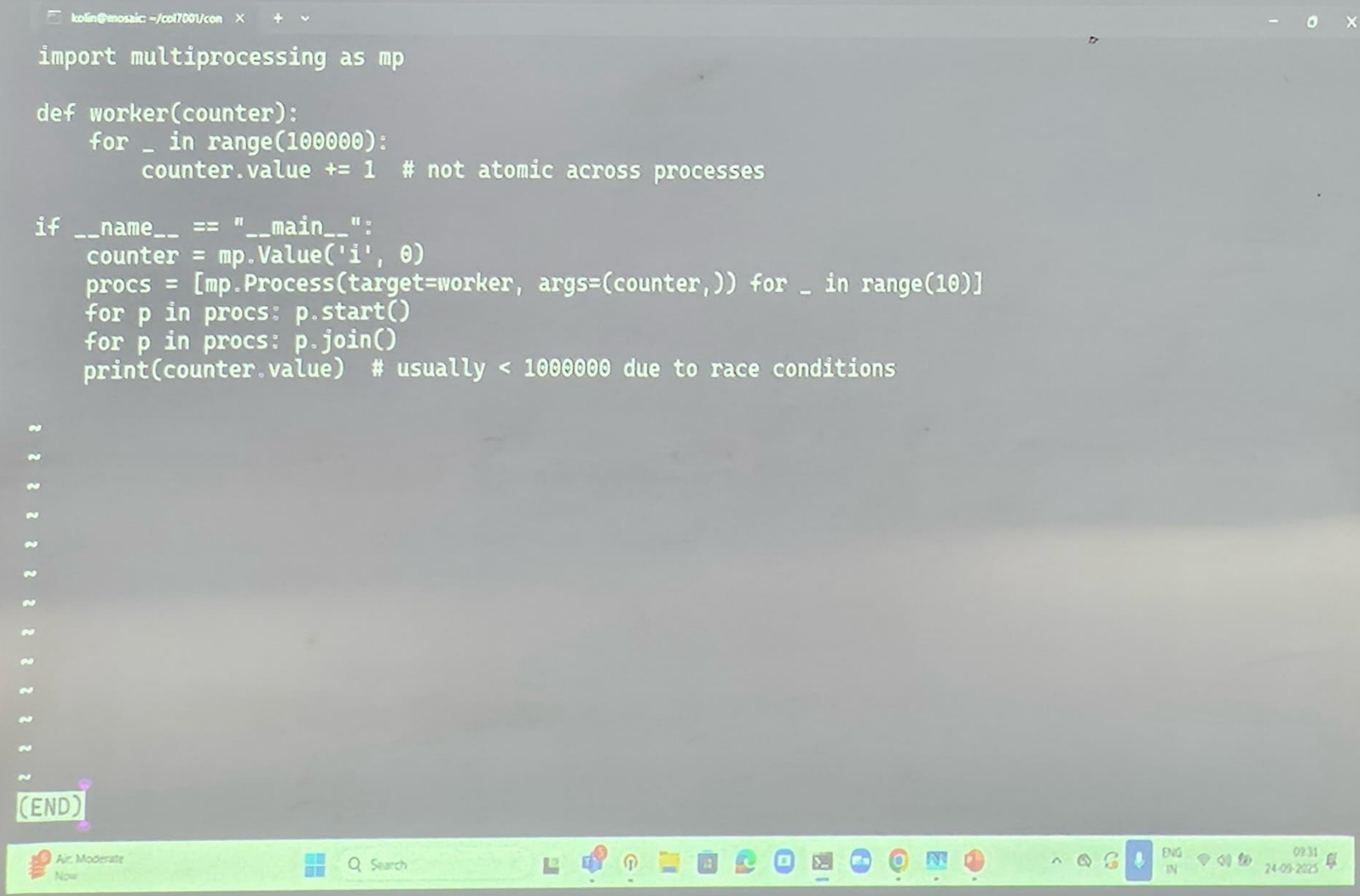


10 Core

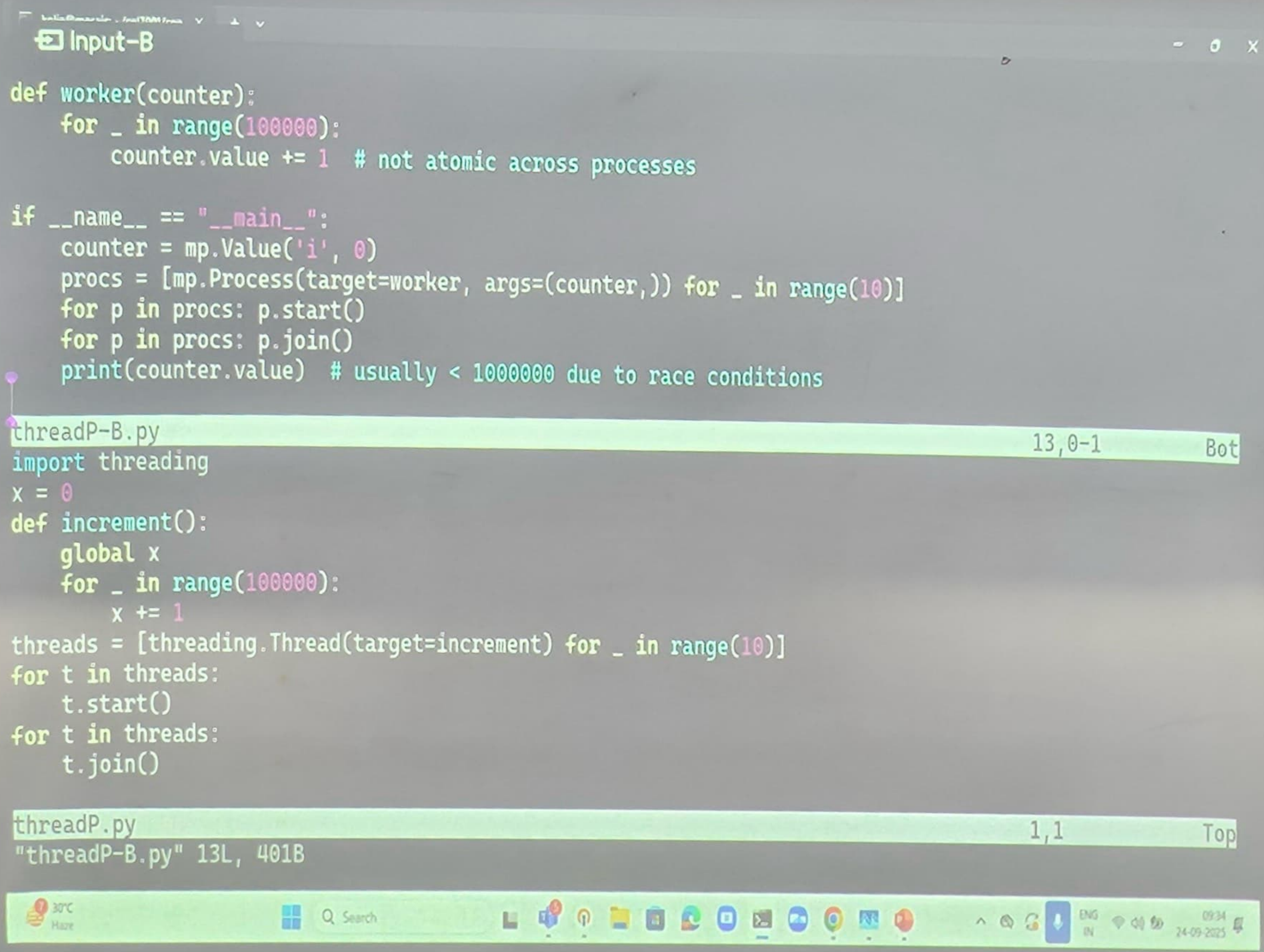
Sum

Partial Sum on Small Chunks

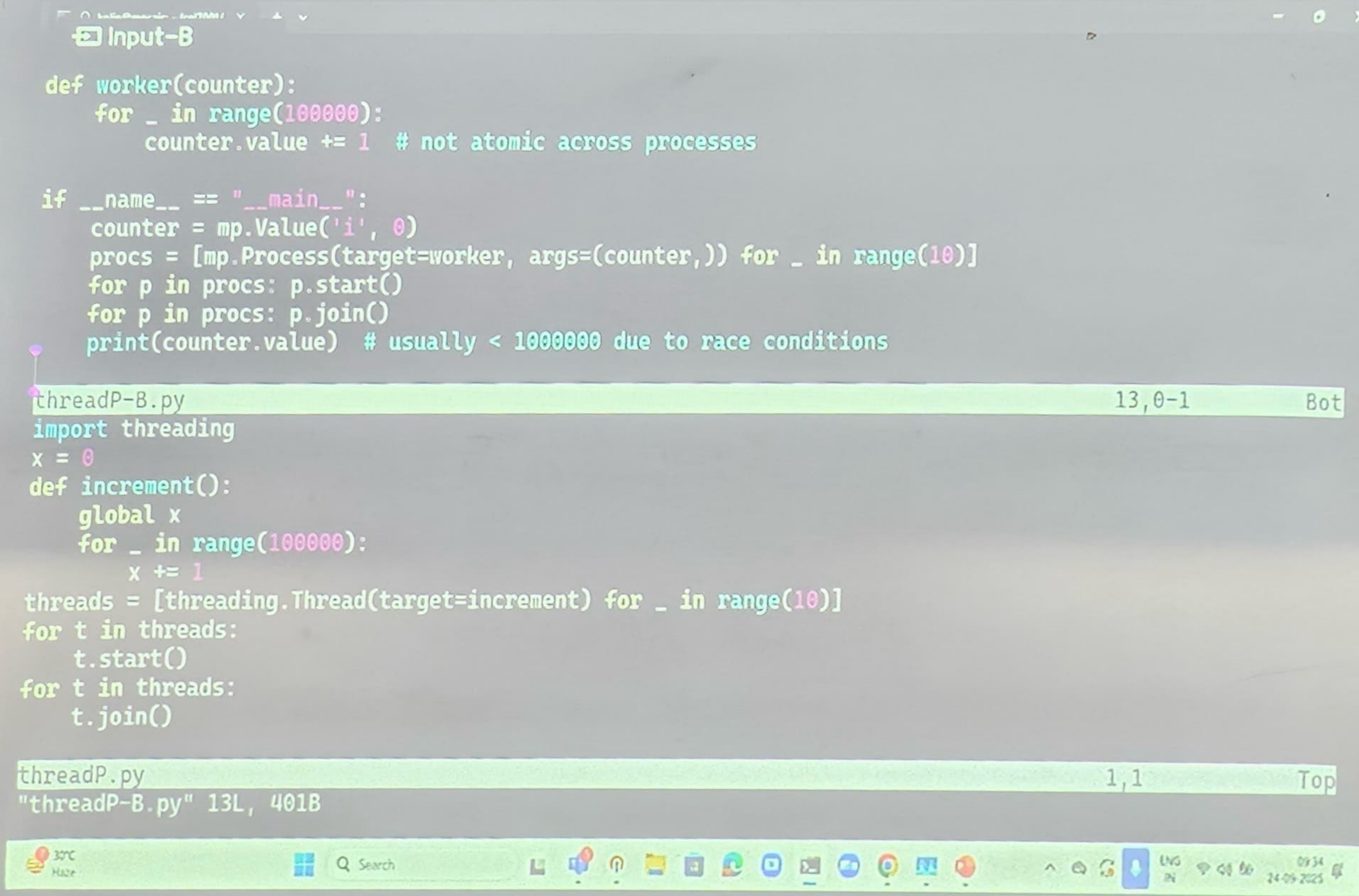
Reduct^{n} of Partial Sums



```
import multiprocessing as mp

def worker(counter):
    for _ in range(100000):
        counter.value += 1  # not atomic across processes

if __name__ == "__main__":
    counter = mp.Value('i', 0)
    procs = [mp.Process(target=worker, args=(counter,)) for _ in range(10)]
    for p in procs: p.start()
    for p in procs: p.join()
    print(counter.value)  # usually < 1000000 due to race conditions

~
~
~
~
~
~
~
~
~
~
~
~
~
(END)
```



```
def worker(counter):
    for _ in range(100000):
        counter.value += 1  # not atomic across processes

if __name__ == "__main__":
    counter = mp.Value('i', 0)
    procs = [mp.Process(target=worker, args=(counter,)) for _ in range(10)]
    for p in procs: p.start()
    for p in procs: p.join()
    print(counter.value)  # usually < 1000000 due to race conditions

threadP-B.py                                                     13,0-1         Bot
import threading
x = 0
def increment():
    global x
    for _ in range(100000):
        x += 1
threads = [threading.Thread(target=increment) for _ in range(10)]
for t in threads:
    t.start()
for t in threads:
    t.join()

threadP.py                                                       1,1            Top
"threadP-B.py" 13L, 401B
```

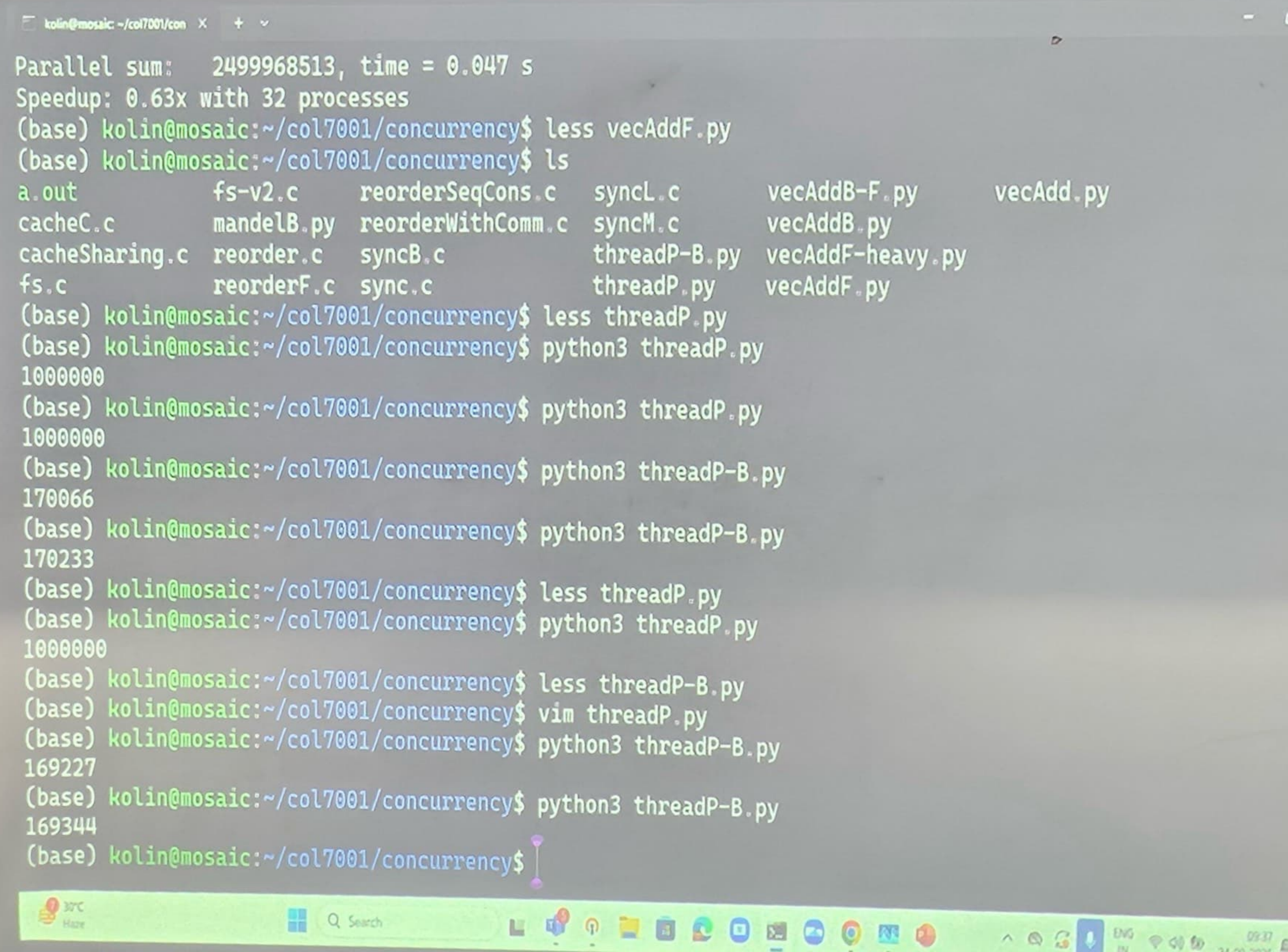


```
def worker(counter):
    for _ in range(100000):
        counter.value += 1  # not atomic across processes

if __name__ == "__main__":
    counter = mp.Value('i', 0)
    procs = [mp.Process(target=worker, args=(counter,)) for _ in range(10)]
    for p in procs: p.start()
    for p in procs: p.join()
    print(counter.value)  # usually < 1000000 due to race conditions
```

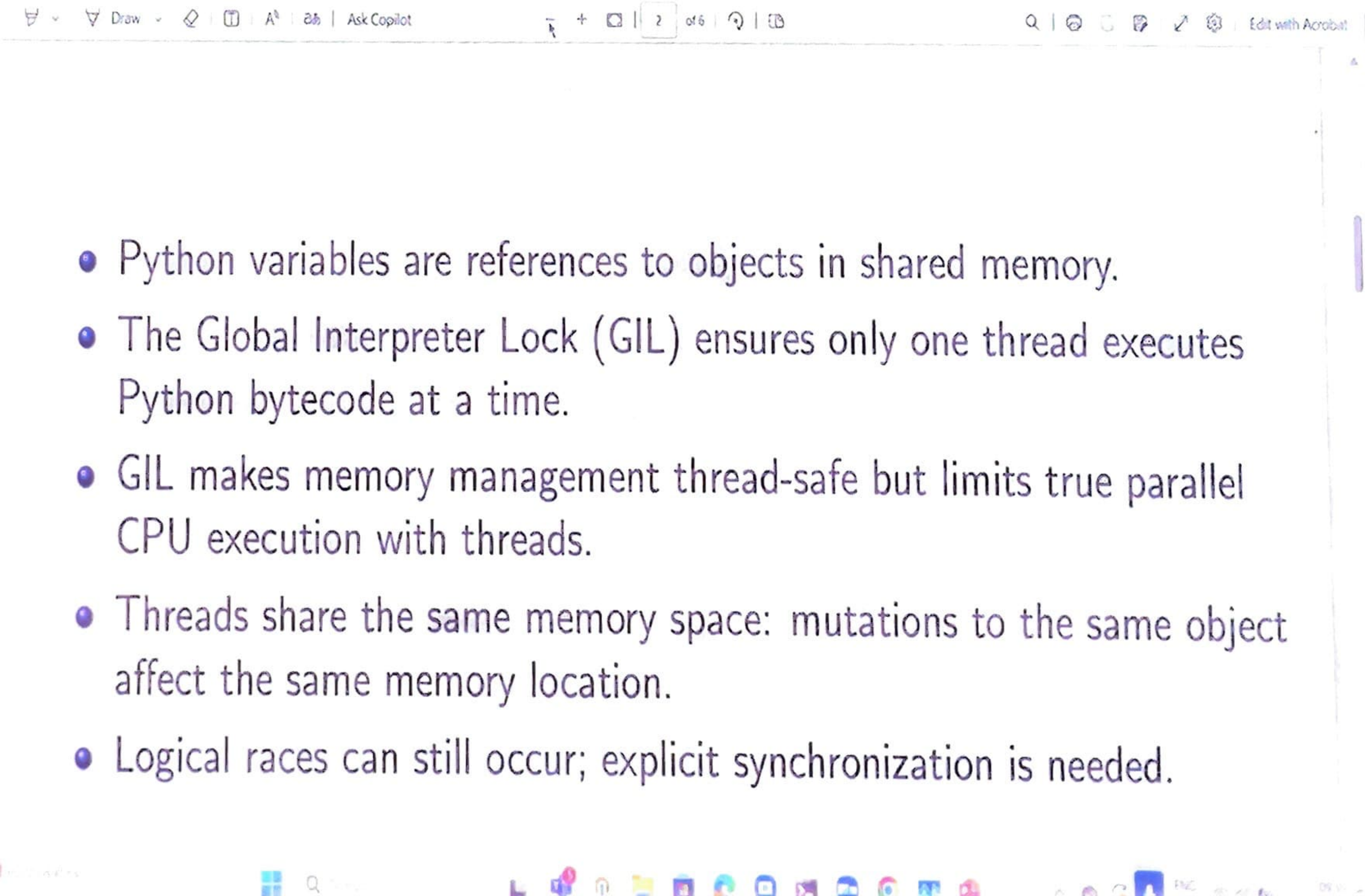
```
threadP-B.py                                                    13,0-1         Bot
```

```
import threading
x = 0
def increment():
    global x
    for _ in range(100000):
        x += 1
threads = [threading.Thread(target=increment) for _ in range(10)]
for t in threads:
    t.start()
for t in threads:
    t.join()
```

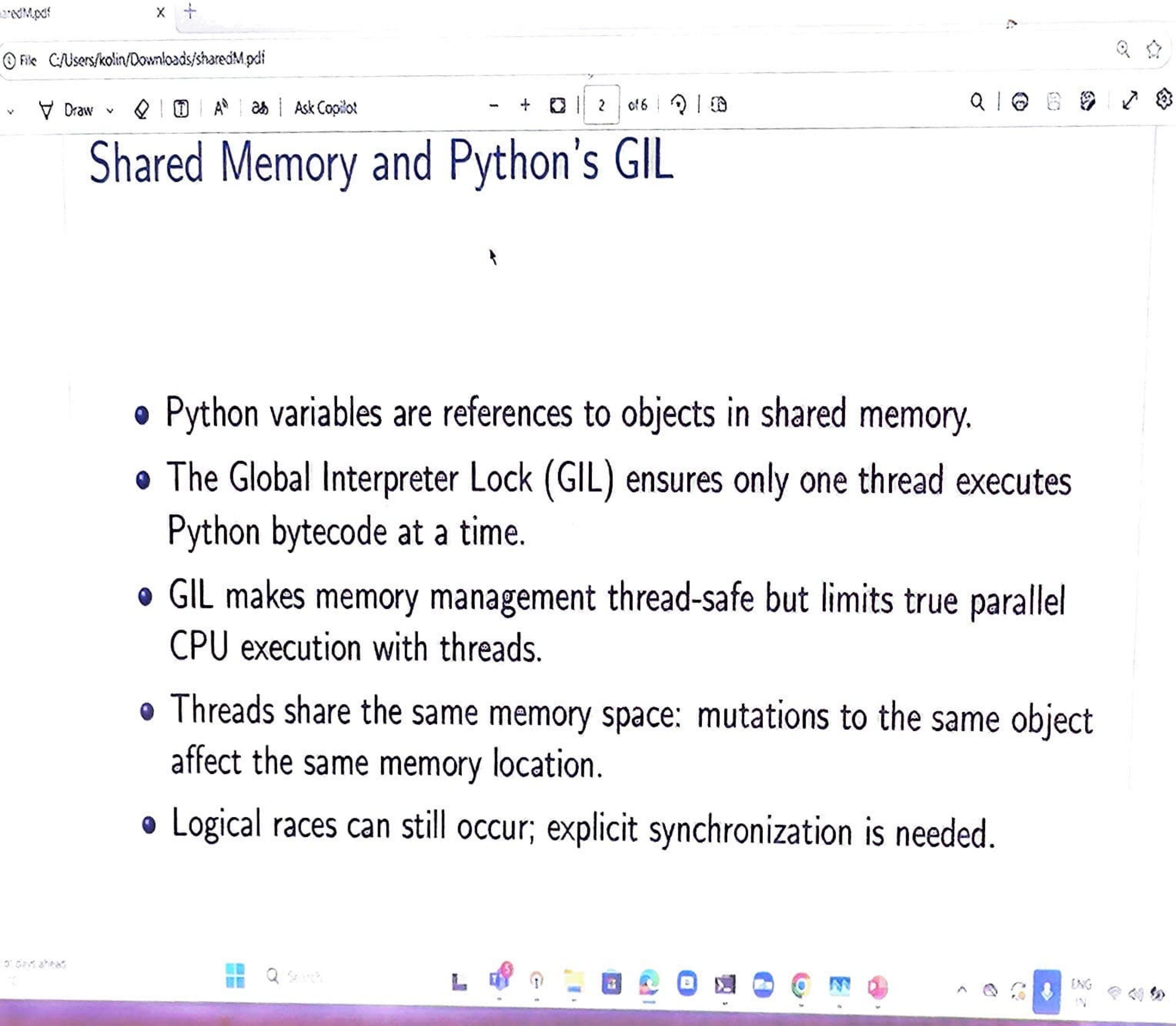
```
threadP.py                                                      1,1            Top
"threadP-B.py" 13L, 401B
```



```
Parallel sum:   2499968513, time = 0.047 s
Speedup: 0.63x with 32 processes
(base) kolin@mosaic:~/col7001/concurrency$ less vecAddF.py
(base) kolin@mosaic:~/col7001/concurrency$ ls
a.out            fs-v2.c     reorderSeqCons.c   syncL.c        vecAddB-F.py       vecAdd.py
cacheC.c         mandelB.py  reorderWithComm.c  syncM.c        vecAddB.py
cacheSharing.c   reorder.c   syncB.c            threadP-B.py   vecAddF-heavy.py
fs.c             reorderF.c  sync.c             threadP.py     vecAddF.py
(base) kolin@mosaic:~/col7001/concurrency$ less threadP.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP.py
1000000
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP.py
1000000
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP-B.py
170066
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP-B.py
170233
(base) kolin@mosaic:~/col7001/concurrency$ less threadP.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP.py
1000000
(base) kolin@mosaic:~/col7001/concurrency$ less threadP-B.py
(base) kolin@mosaic:~/col7001/concurrency$ vim threadP.py
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP-B.py
169227
(base) kolin@mosaic:~/col7001/concurrency$ python3 threadP-B.py
169344
(base) kolin@mosaic:~/col7001/concurrency$
```
]



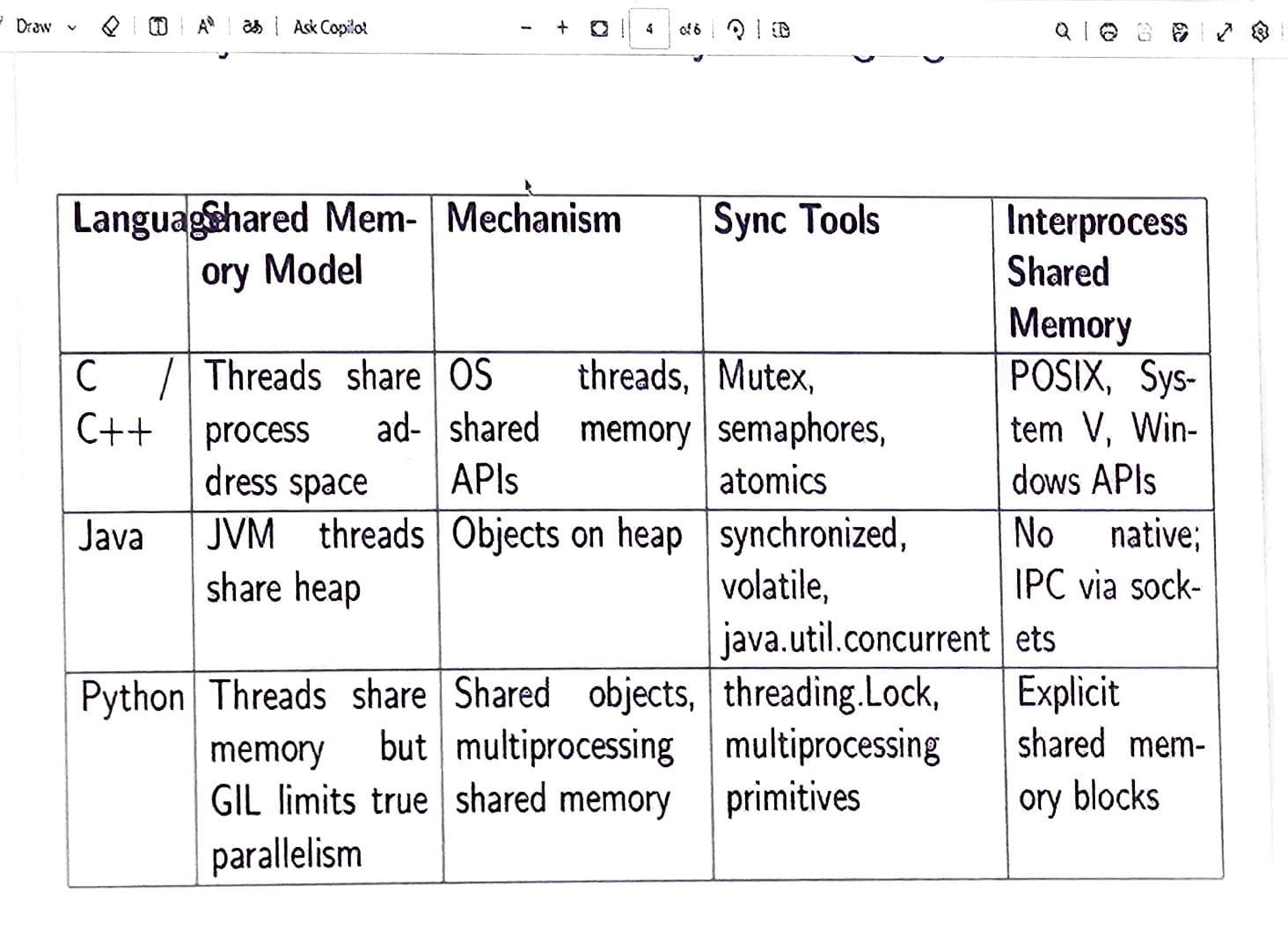
- Python variables are references to objects in shared memory.
- The Global Interpreter Lock (GIL) ensures only one thread executes Python bytecode at a time.
- GIL makes memory management thread-safe but limits true parallel CPU execution with threads.
- Threads share the same memory space: mutations to the same object affect the same memory location.
- Logical races can still occur; explicit synchronization is needed.



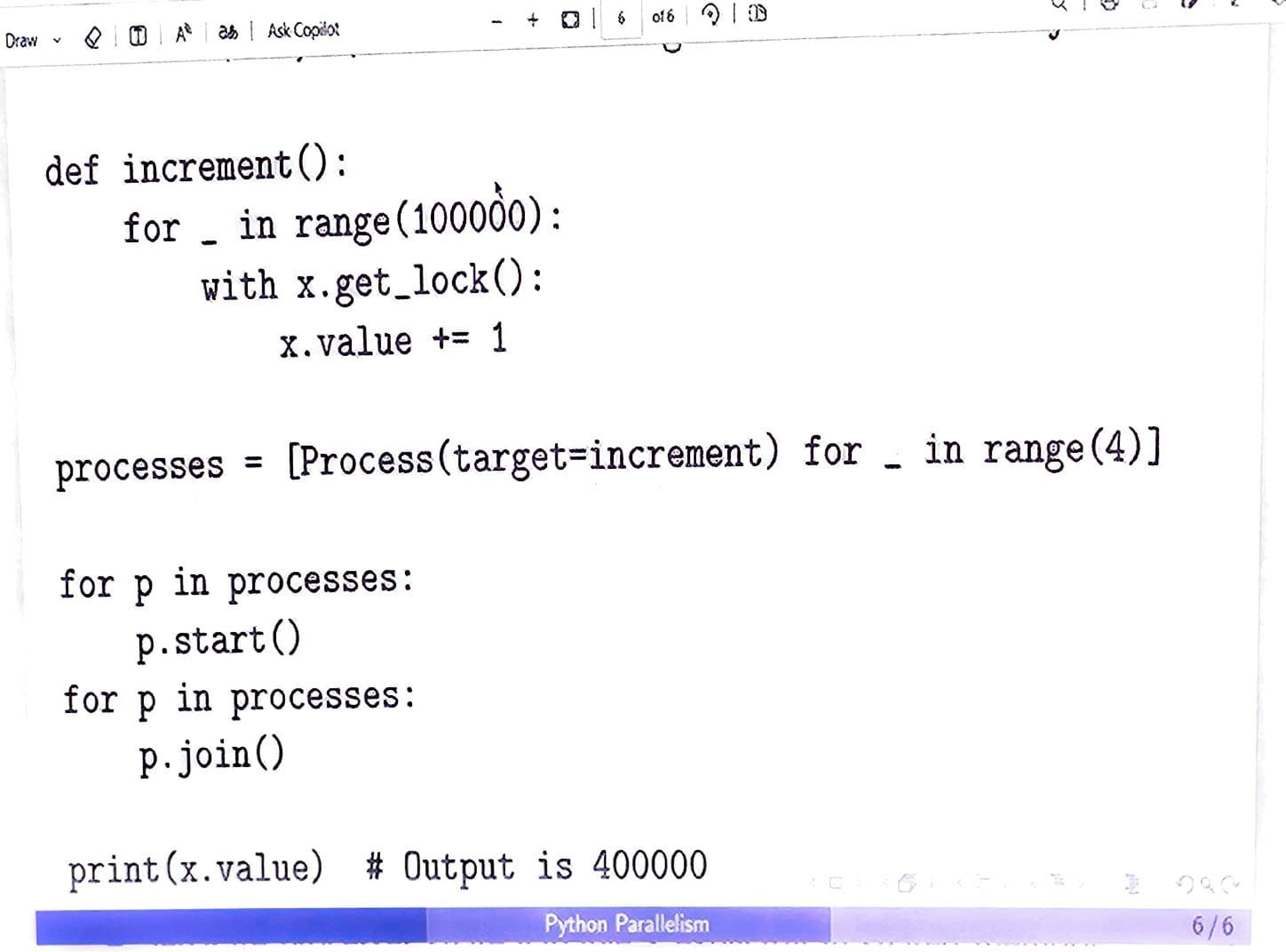
Shared Memory and Python's GIL

- Python variables are references to objects in shared memory.
- The Global Interpreter Lock (GIL) ensures only one thread executes Python bytecode at a time.
- GIL makes memory management thread-safe but limits true parallel CPU execution with threads.
- Threads share the same memory space: mutations to the same object affect the same memory location.
- Logical races can still occur; explicit synchronization is needed.

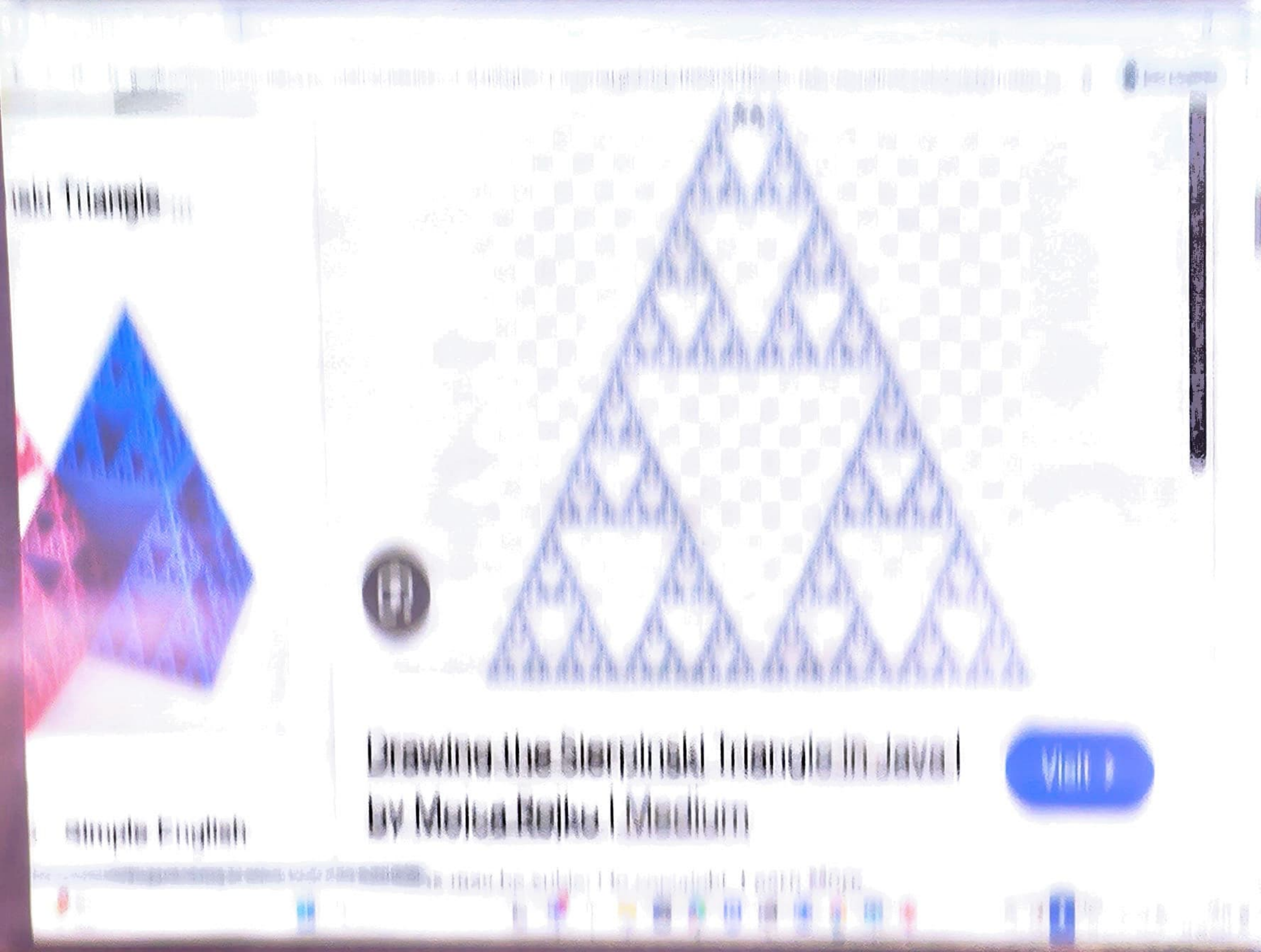
- Python's `multiprocessing` module creates separate processes with independent memory spaces.
- Shared memory must be explicitly allocated (e.g., via `multiprocessing.shared_memory`) for interprocess communication.
- Enables true parallelism by bypassing the GIL limitation.
- Processes do not share Python objects implicitly; communication is explicit.



Language	Shared Memory Model	Mechanism	Sync Tools	Interprocess Shared Memory
C / C++	Threads share process address space	OS threads, shared memory APIs	Mutex, semaphores, atomics	POSIX, System V, Windows APIs
Java	JVM threads share heap	Objects on heap	synchronized, volatile, java.util.concurrent	No native; IPC via sockets
Python	Threads share memory but GIL limits true parallelism	Shared objects, multiprocessing shared memory	threading.Lock, multiprocessing primitives	Explicit shared memory blocks



```
def increment():
    for _ in range(100000):
        with x.get_lock():
            x.value += 1

processes = [Process(target=increment) for _ in range(4)]

for p in processes:
    p.start()
for p in processes:
    p.join()

print(x.value)  # Output is 400000
```



Drawing the Sierpinski Triangle in Java | by [illegible] | Medium

Images may be subject to copyright. Learn More