Homework 1 Tips

Week 2 – Part 2

Amittai Aviram – 15 September 2020

Python Resources

Threads and Processes

- Threads
  - `import threading`
  - User-level threads—scheduled by the Python runtime
  - Do not map onto the machine's CPU cores—all threads are executed on a single core
  - Share the processes memory—global variables, etc.
  - Enable the process to get work done while one or another thread waits for resources
  - Best for *network-bound* applications
- Processes
  - import multiprocessing
  - Kernel-level processes—scheduled by the OS
  - The OS can schedule them on available CPU cores (like other processes)
  - Shared data must be communicated between processes
  - The OS incurs the cost of setting up *context* for each new process
  - Enable tasks to be performed *at the same time*
  - Best for *compute-bound* applications

Similar API

The Basics

```
from threading import Thread
from typing import List

def foo(x: int, y: int, z: List[int]):
    z.append(x * y)
    z.append(x + y)

if __name__ == '__main__':
    result = []
    thread_0 = Thread(target=foo, args=(17, 19, result)
    thread_0.start()
    thread_0.join()
    print(result)
```

```
from multiprocessing import Array, Process

def foo(x: int, y: int, z: Array):
    z[0] = (x * y)
    z[1] = (x + y)

if __name__ == '__main__':
    result = Array('i', 2)
    process_0 = Process(target=foo, args=(17, 19, result)
    process_0.start()
    process_0.join()
    print(result[:])
```

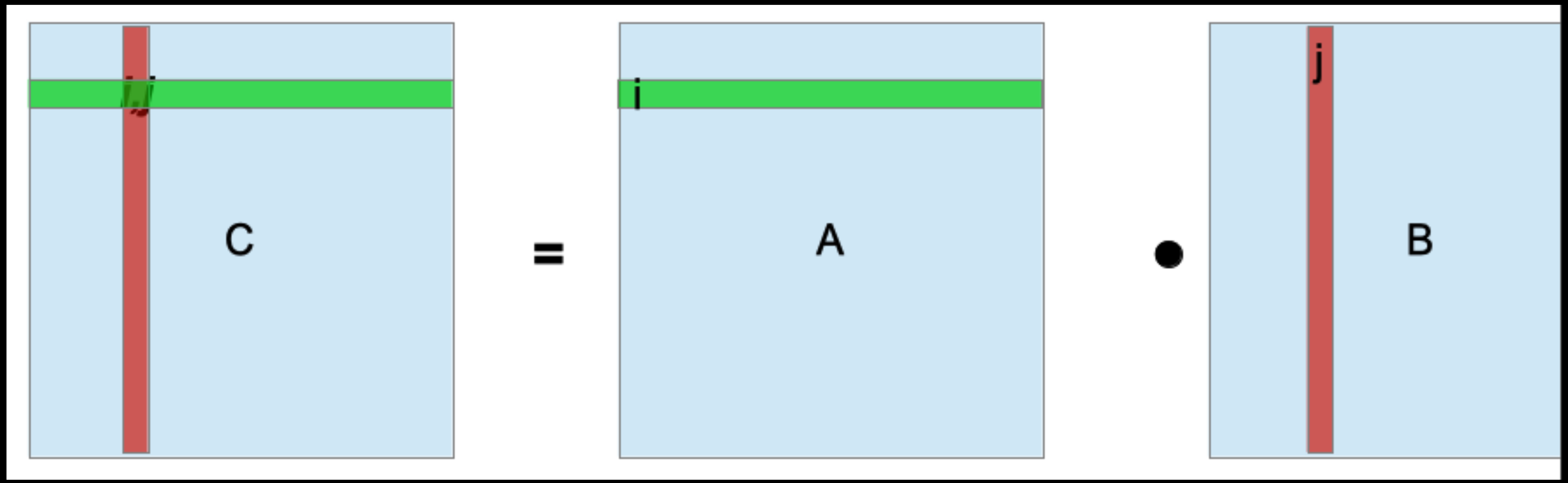
Synchronization and Communication

See Python Documentation

- Communication
  - Queue (both threading and multiprocessing—distinct but similar)
  - Pipe (multiprocessing)
- Synchonization
  - Lock
  - Condition (i.e., condition variable)
  - Semaphore
  - …

Matrix Multiplication

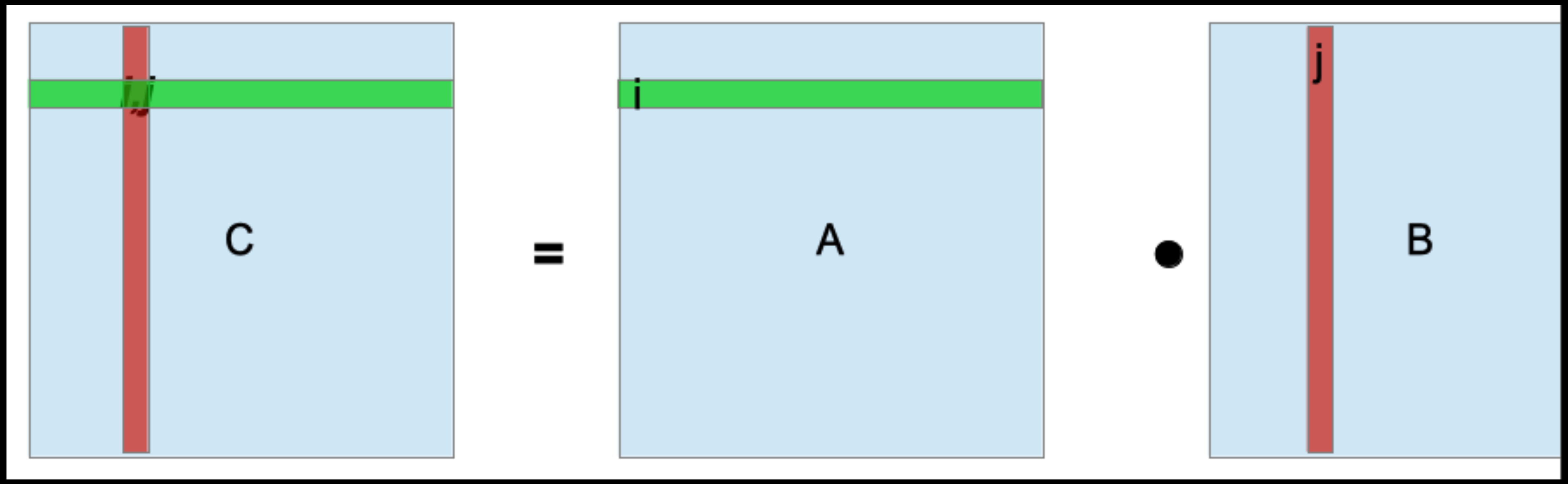
Basic Algorithm – Square Matrices




```
# SQUARE MATRICES
# C = A * B
# A, B, and C are n x n lists of lists.
# C is initialized to 0.
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
```

Matrix Multiplication

Basic Algorithm – Generalized




```
# GENERAL CASE
# C = A * B
# A, B, and C are lists of lists.
# C is initialized to 0.
# A: m rows by n columns — m == len(A)
# B: n rows by o columns — n == len(B) == len(A[0])
# C: m rows by o columns — o == len(B[0])
for i in range(len(A)):
    for j in range(len(B[0])):
        for k in range(len(B)):
            C[i][j] += A[i][k] * B[k][j]
```

Using One-Dimensional Arrays

To Supoprt Shared Memory

```
# C = A * B
# A, B, and C are multiprocessing Array objects.
# C is initialized to 0.
# A, B, and C all represent square n x n matrices.
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i * n + j] += A[i * n + k] * B[k * n + j]
```

```
# C = A * B
# A, B, and C are multiprocessing Array objects.
# C is initialized to 0.
# A:  length m * n
# B: length n * o
# C: length m * o
for i in range(m):
    for j in range(o):
        for k in range(n):
            C[i * o + j] += A[i * n + k] * B[k * o + j]
```

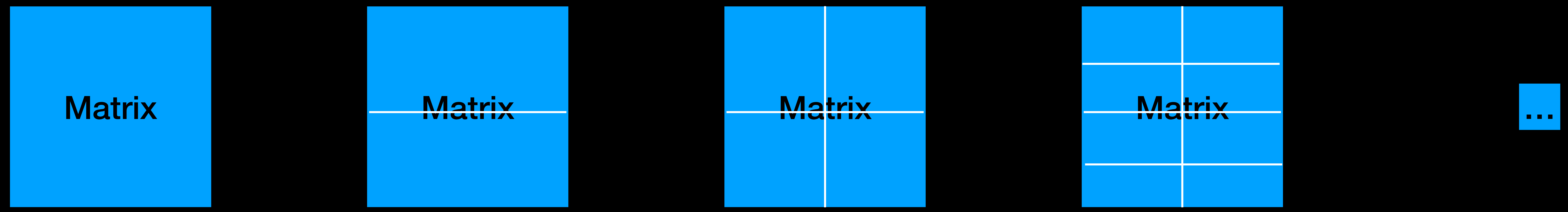
Dividing Up the Work
Partitioning Data

- Are the data independent?
  - How do you know?
- Is any communication between processes necessary?
- How do you divide the input data up evenly among the workers?
  - 2?
  - 4?
  - 8?

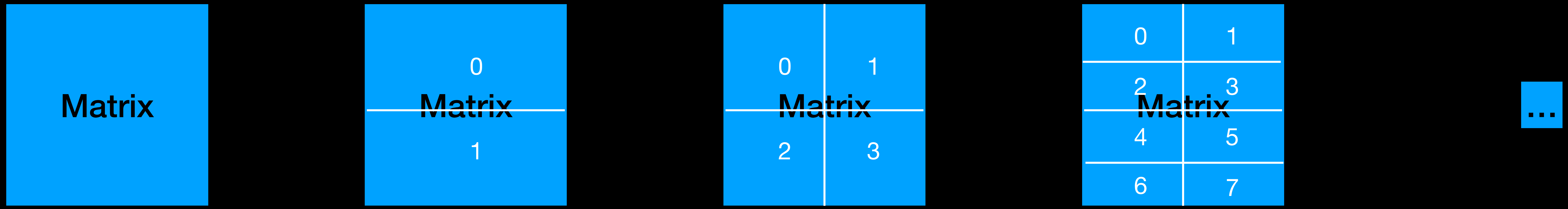
Thing about it before we move on!

Chunking

Tiling

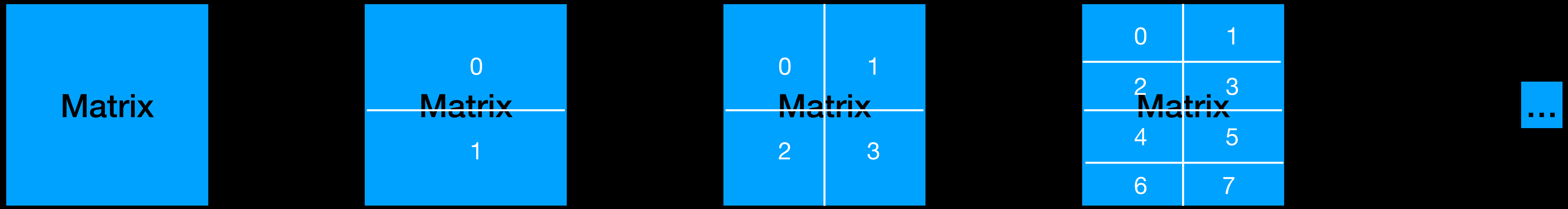


Chunking
Tiling



Chunking

Tiling



$p = 2^0$
rows: 1
columns: 1

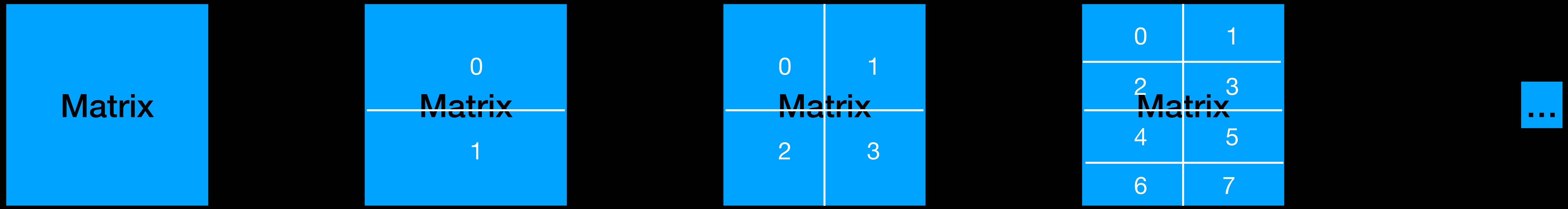
$p = 2^1$
rows: 2
columns: 1

$p = 2^2$
rows: 2
columns: 2

$p = 2^3$
rows: 4
columns: 2

Chunking

Tiling



$p = 2^0$
rows: 1
columns: 1

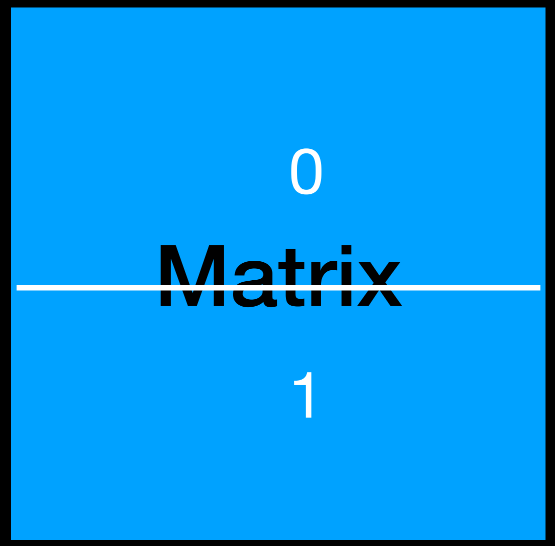
$p = 2^1$
rows: 2 (dim/2)
columns: 1 (dim)

$p = 2^2$
rows: 2 (dim/2)
columns: 2 (dim/2)

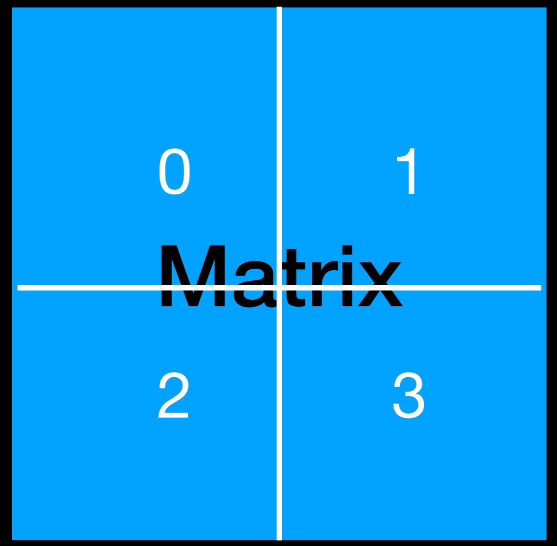
$p = 2^3$
rows: 4 (dim/2)
columns: 2 (dim/4)

Chunking
Tiling

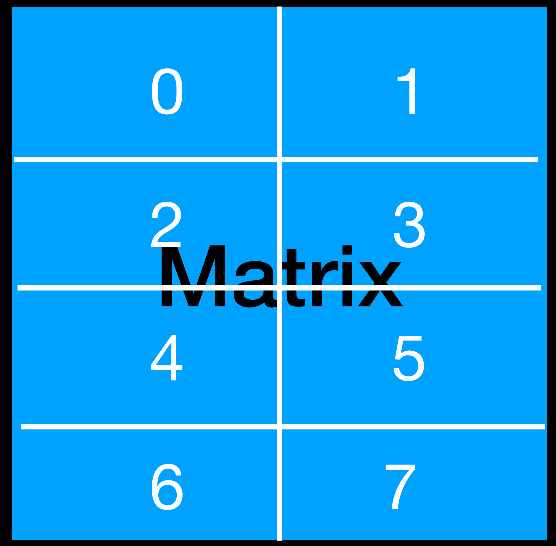
$p = 2^0$
rows: 1
columns: 1



$p = 2^1$
rows: 2 (dim/2)
columns: 1 (dim)
vertical starts:
0: 0 1: v_chunk
horizontal starts:
0, 1: 0



$p = 2^2$
rows: 2 (dim/2)
columns: 2 (dim/2)
vertical starts:
0, 1: 0
2, 3: v_chunk
horizontal starts:
0, 2: 0
1, 3: h_chunk



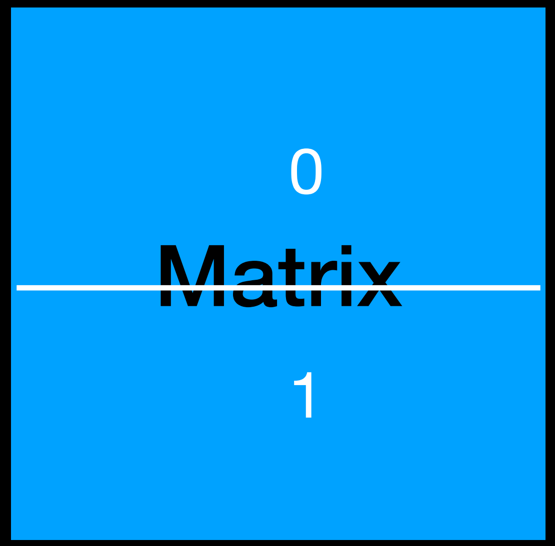
$p = 2^3$
rows: 4 (dim/2)
columns: 2 (dim/4)
vertical starts:
0, 1: 0
2, 3: v_chunk
4, 5: 2 * v_chunk
6, 7: 3 * v_chunk
horizontal starts:
0, 2, 4, 6: 0
1, 3, 5, 7: h_chunk

…

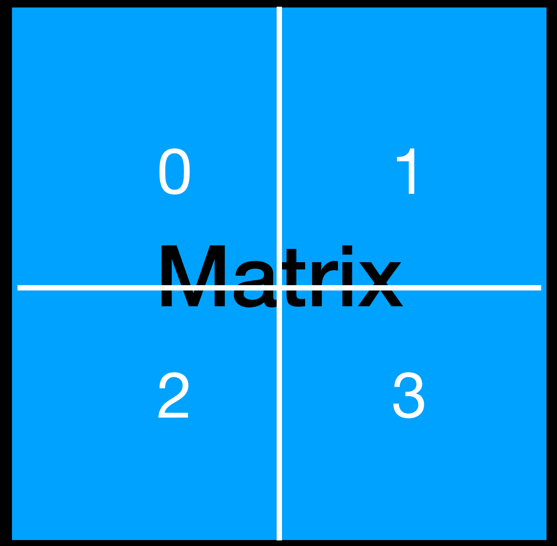
Chunking

Tiling

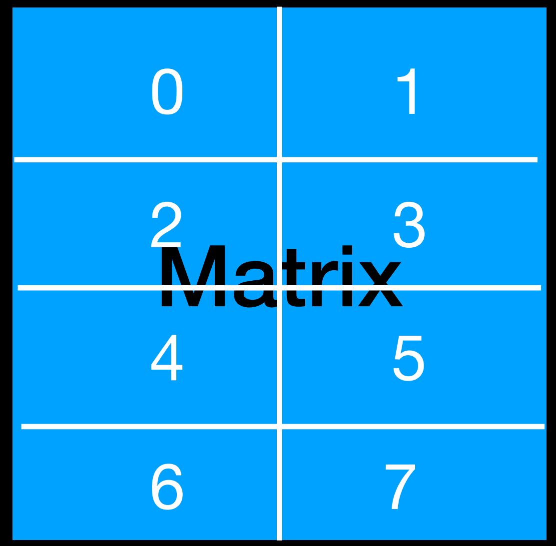
$p = 2^0$
rows: 1
columns: 1



$p = 2^1$
rows: 2 (dim/2)
columns: 1 (dim)
vertical starts:
0: 0 * v_chunk
1: 1 * v_chunk
horizontal starts:
0, 1: 0 * h_chunk



$p = 2^2$
rows: 2 (dim/2)
columns: 2 (dim/2)
vertical starts:
0, 1: 0 * v_chunk
2, 3: 1 * v_chunk
horizontal starts:
0, 2: 0 * h_chunk
1, 3: 1 * h_chunk



$p = 2^3$
rows: 4 (dim/2)
columns: 2 (dim/4)
vertical starts:
0, 1: 0 * v_chunk
2, 3: 1 * v_chunk
4, 5: 2 * v_chunk
6, 7: 3 * v_chunk
horizontal starts:
0, 2, 4, 6: 0 * h_chunk
1, 3, 5, 7: 1 * h_chunk

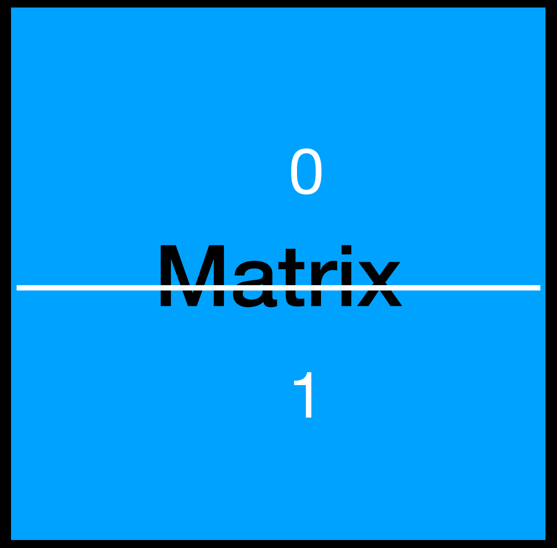
…

Find the general pattern for vertical and horizontal starts (and ends) based on worker ID.
Test on $p = 16 = 2^4$.

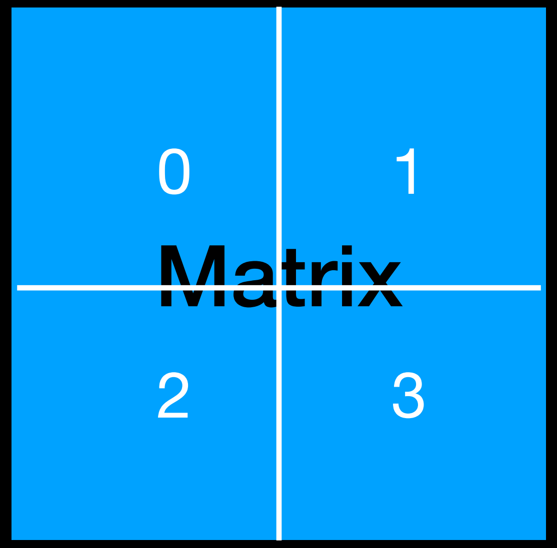
Chunking

Tiling

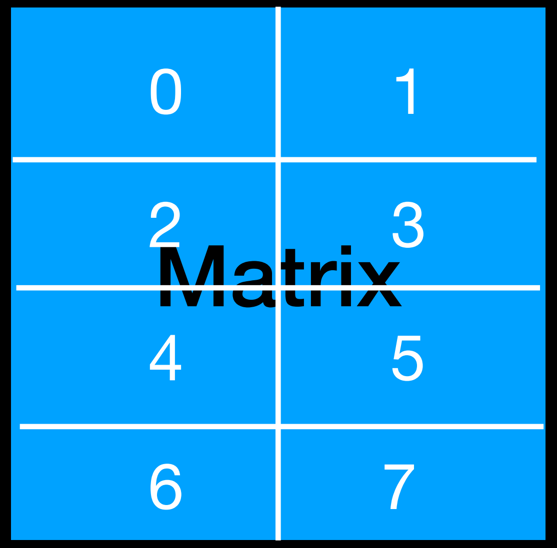
$p = 2^0$
rows: 1
columns: 1



$p = 2^1$
rows: 2 (dim/2)
columns: 1 (dim)
vertical starts:
(pid // columns) * v_chunk
horizontal starts:
(pid % columns) * h_chunk



$p = 2^2$
rows: 2 (dim/2)
columns: 2 (dim/2)
vertical starts:
(pid // columns) * v_chunk
horizontal starts:
(pid % columns) * h_chunk



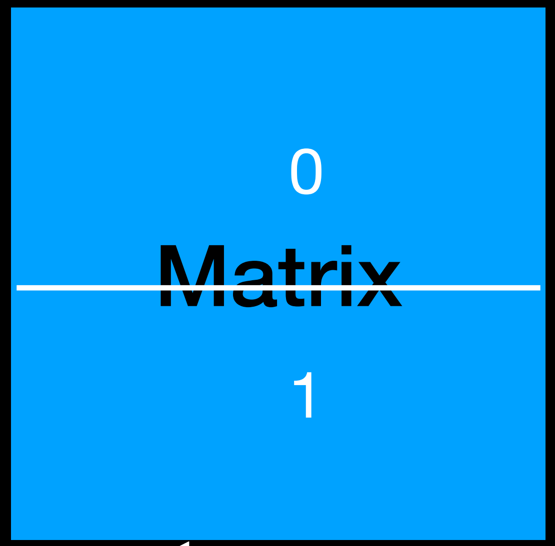
$p = 2^3$
rows: 4 (dim/2)
columns: 2 (dim/4)
vertical starts:
(pid // columns) * v_chunk
horizontal starts:
(pid % columns) * h_chunk

// = integer division
1 // 2 = 0
2 // 2 = 1
3 // 2 = 1

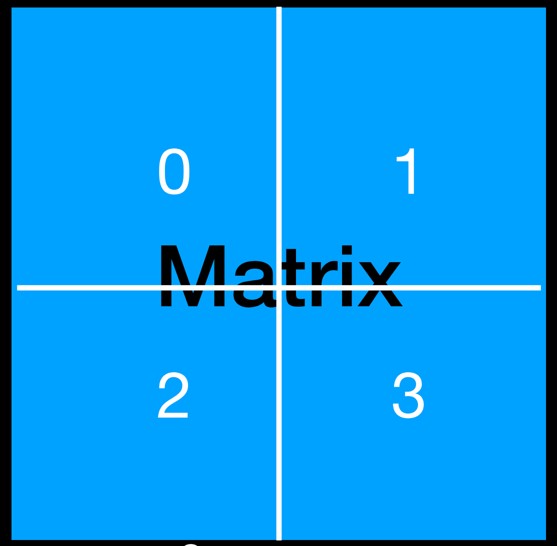
Chunking

Tiling

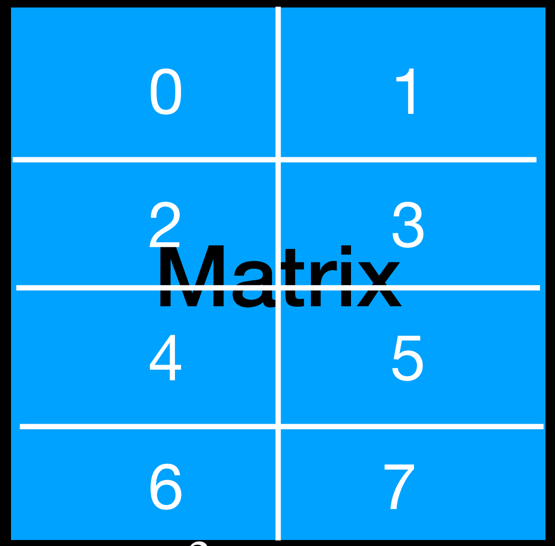
$p = 2^0$
rows: 1
columns: 1



$p = 2^1$
rows: 2 $(2^{q//2+1})$
columns: 1 $(2^{q//2})$
vertical starts:
(pid // columns) * v_chunk
horizontal starts:
(pid % columns) * h_chunk



$p = 2^2$
rows: 2 $(2^{q//2})$
columns: 2 $(2^{q//2})$
vertical starts:
(pid // columns) * v_chunk
horizontal starts:
(pid % columns) * h_chunk



$p = 2^3$
rows: 4 $(2^{q//2+1})$
columns: 2 $(2^{q//2})$
vertical starts:
(pid // columns) * v_chunk
horizontal starts:
(pid % columns) * h_chunk

// = integer division
1 // 2 = 0
2 // 2 = 1
3 // 2 = 1