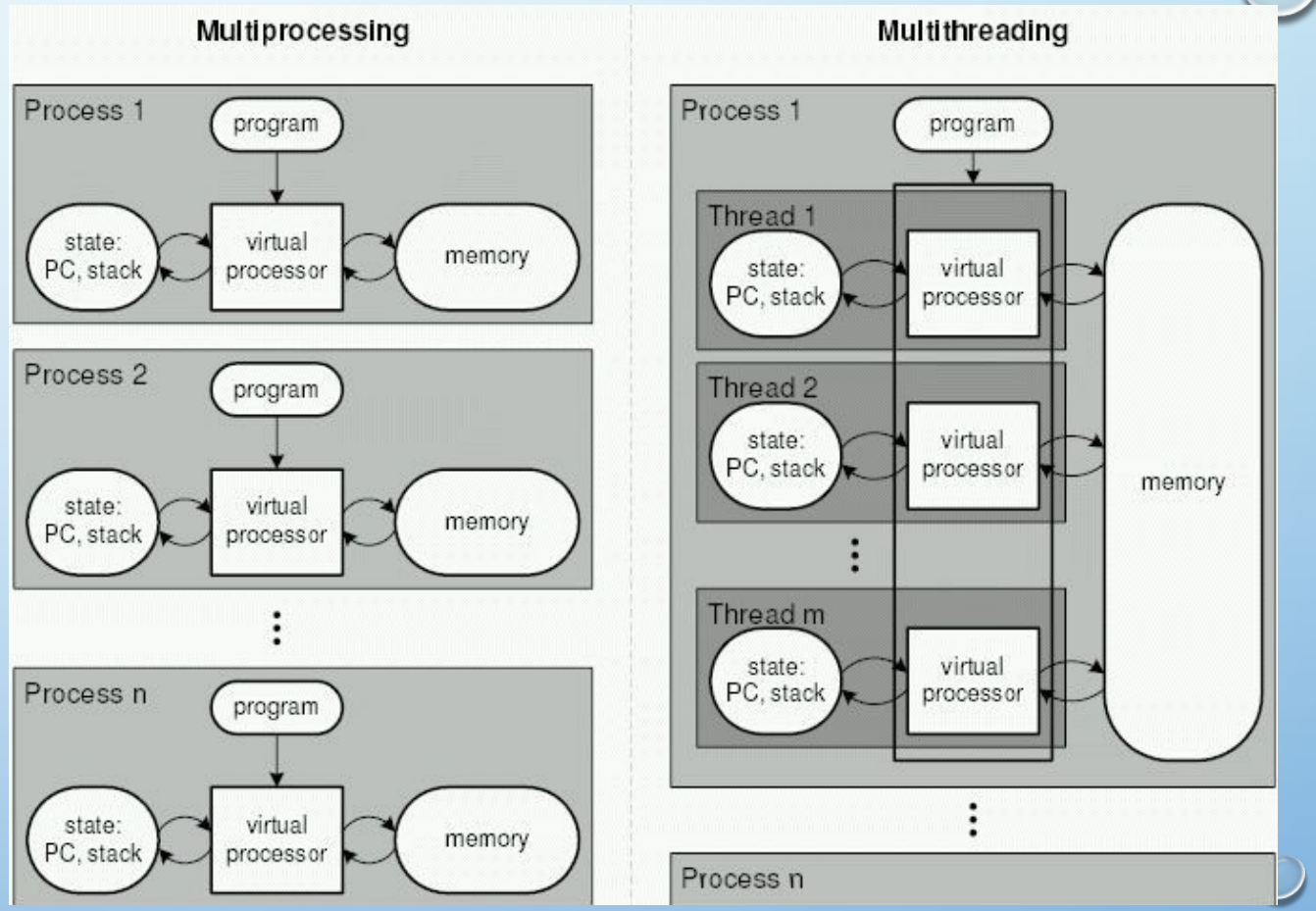
LECTURE 7 PYTHON MULTIPROCESSING

MULTIPROCESSING

Python Multiprocessing and Threading

- Python threads share the process context
- Python multiprocessing runs with completely independent processes (new copies of python interpreter).



Python Multiprocessing

- Python multiprocessing using a built-in multiprocessing module to manage processes
- Multiprocessing is similar to threading in the code
  1. Create an instance of the **multiprocessing.Process** class.
  2. Specify the name of the function as the starting point of the new process via the "**target**" argument.
  3. Call the **start()** function.
  4. We can explicitly wait for the new process to finish executing by calling the **join()**
- Run lect7 /RunFunctionProcess.py, you can see the main process and the task process are progressing concurrently.
  - Use 'ps –u username' to see that there are two python process running.

```
from time import sleep
from multiprocessing import Process

# a function that blocks for a moment
def task():
    sleep(1)
    # display a message
    print('task Process running for 1 second')
    sleep(10)
    print('task Process running for 11 second')

# create a Process
p = Process(target=task)
p.start()  # duplicate the main process
# wait for the child process to finish
print('Main thread after task process starts...')
p.join()
```

Python Multiprocessing

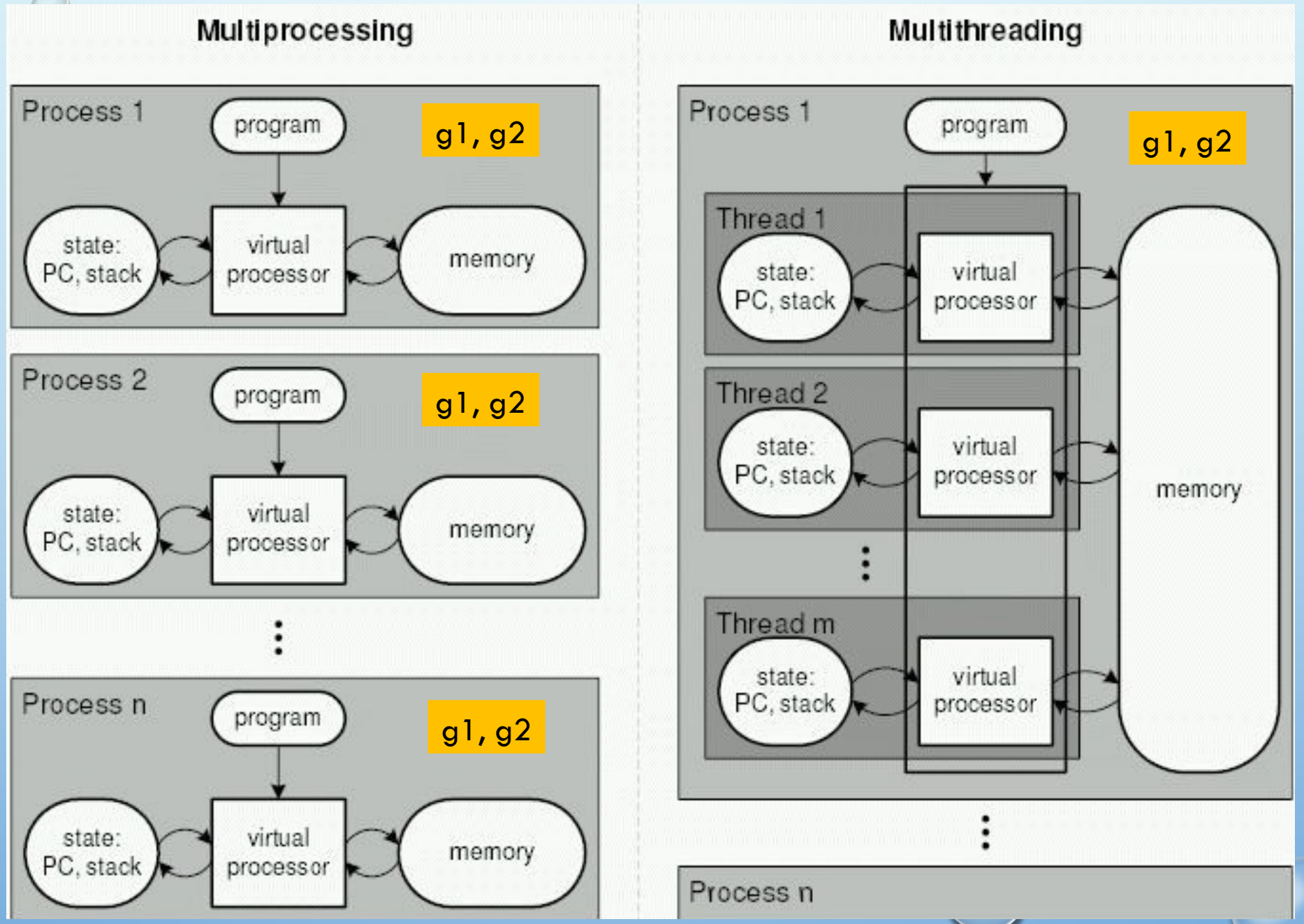
- To run a function in a processes with parameters:
  1. Create an instance of the **multiprocessing.Process** class.
  2. Specify the name of the function via the "**target**" argument.
     1. Function has parameters
  3. Specify the arguments in order that the function expects them via the "**args**"
  4. Call the **start()** function.
  5. We can explicitly wait for the new thread to finish executing by calling the **join()**
- See lect7/RunFunctionProcessWArgs.py

```
……
def task(arg1, arg2):
    # display a message
    print("arg1=",arg1)
    print("arg2=",arg2)

# create a thread
p = Process(target=task, args=("One",2))
```

Difference between Python Thread and Process from the programmer's perspective

- With threads, global variables in different threads are the same variables – the variable updated in one thread will be seen by all threads
  - What about local variables in a function?
- With processes:
  - The value of all global variables are duplicated when a new process is started.
  - After a process is started, variables in different processes are independent (unrelated)
- See lect7/variables_process.py and lect7_variables_thread.py


Multiprocessing
Process 1
program
g1, g2
state:
PC, stack
virtual
processor
memory
Process 2
program
g1, g2
state:
PC, stack
virtual
processor
memory
Process n
program
g1, g2
state:
PC, stack
virtual
processor
memory
Multithreading
Process 1
program
g1, g2
Thread 1
state:
PC, stack
virtual
processor
Thread 2
state:
PC, stack
virtual
processor
Thread m
state:
PC, stack
virtual
processor
memory
Process n

Potential speedup for CPU bound and IO bound applications with multiprocessing

- See lect7/iowithprocesses.py and lect7/cpuwithprocesses.py
  - Multiprocessing in Python can truly take advantage of the multiple cores in a process, and is parallel processing on a multicore computer.

Inter Process Communication with Python Multiprocessing

- Now we can create two or more processes in a python program.
- Normally, processes are completely independent. But in order for multiple processes to solve problem together, they need to communicate (e.g sharing result, objects, etc). This is where inter-process communication mechanisms come into play!
- Inter-process communication mechanisms:
  - Common IPC mechanisms include semaphore, lock, shared memory, pipe, message queue.
  - The multiprocessing module has these primitives. See https://docs.python.org/3/library/multiprocessing.html

IPC: Message Queue

- Message queue is a common IPC (inter-process communication mechanisms)
  - A queue is a data structure that allows adding and removing items following the FIFO order
  - A message queue is a FIFO queue that allows **different processes** to add and remove items.
    - The first item added is the first item retrieved, but may or may not be by a different process.

Message Queue

- Python provides the message queue in multiprocessing.Queue class
  - queue = multiprocessing.Queue() # create a message queue
  - queue.put(item) # add an item to a queue
  - item = queue.get() # get an item from the queue
  - size = queue.qsize() # number of items in the queue
  - queue.empty(): # check if a message queue is empty
- See lect7/message_queue.py

Things to be careful when using message queue

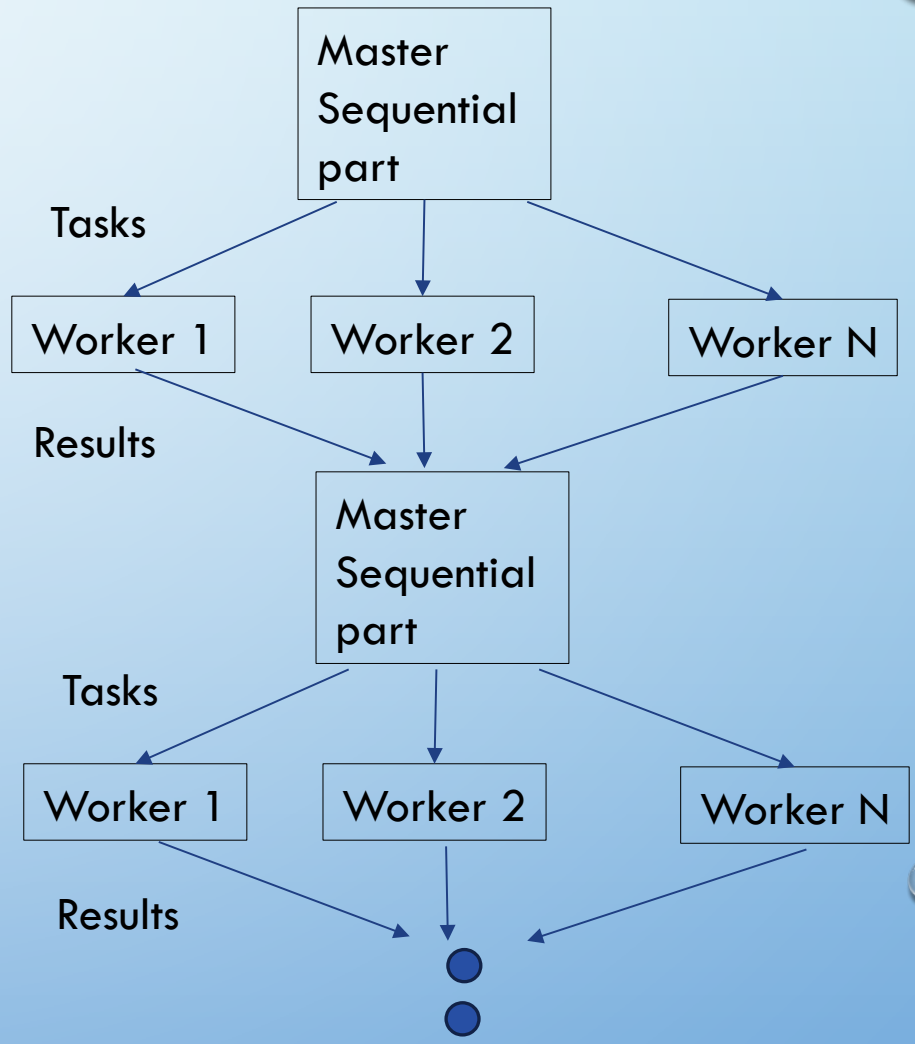
- All processes that have access to a message queue can read and write to it.
  - What are these processes in a python code?
  - If the code allows multiple processes to write/read the queue at the same time, you need to understand the implication of the non-deterministic read/write of the queue.
  - Example: What is the problem with lect7/message_queue1.py?
  - Fixes: lect7/message_queue2.py

Things to be careful when using message queue

- Each message queue has a limited amount of buffer memory
  - After an item is placed in the queue, but has not been consumed, the item must be buffered by the operating system in the buffer memory for the message queue.
  - This consumes precious physical/system memory, and can be very limited.
  - The default size of buffer memory for each message queue is quite small, usually in tens of kilobytes.
  - Such a limitation applies to all IPC mechanisms.
  - Implication to programmer?
    - If a process write more than the buffer size, the process will be blocked (stuck in the put call).
    - Make sure when one process doing put to a queue, there exists another process doing get from the queue
    - Do not put too much data to a queue in each operation.
- See lect7/message_queue_limit.py

Master-worker design pattern in parallel programming

- There are different ways to organize processes in a multi-process program
- Master-worker paradigm is a common design pattern in multiprocessing (or parallel computing in general)
  - Master: usually the main thread/process, performs the sequential part of the program, does bookkeeping, and controls the workers (distributes works among worker processes).
  - Workers: get tasks from the master and perform heavy duty computation tasks in parallel



Computing PI with multiprocessing

$$PI = \lim_{n \to \infty} \frac{4.0}{1 + \frac{i - 0.5}{n} \times \frac{i - 0.5}{n}}$$

```
sum = 0.0
for i in range(1, N+1):
    x = (float(i) - 0.5) / float(N)
    sum = sum + 4.0/(1.0 + x * x)

myApproxPI = sum / float(N)
```

- See lect7/pi.py
  - When n is large the time to compute the approximation can be quite large.
  - High level idea to do this with multiprocessing: chop the loop into chunks and start multiple worker processes. Each worker will perform a chunk of the loop in parallel with other workers that perform other chunks of the loop.
    - Each worker computes a sub-range of the iteration space range(1, N+1)

Computing PI with multiprocessing: the worker

```
sum = 0.0
for i in range(1, N+1):
    x = (float(i) - 0.5) / float(N)
    sum = sum + 4.0/(1.0 + x * x)

myApproxPI = sum / float(N)
```

```
def worker(id) :
    loop until no more work to do
        # get lowerBound and upperBound from the master
        ....
        partialSum = 0.0
        for i in range(lowerBound, upperBound):
            x = (float(i) - 0.5) / float(N)
            partisalSum = partisalSum + 4.0/(1.0 + x * x)
        # Send partitialSum to the master
        ...
```

- The master:
  - Sets up workers and IPC with workers and start the workers
  - Chops the range (1, N+1) into chunks and pass chunks to workers
  - Get partial results from workers and add them up
  - Report the final result.

Using message queue for IPC

```
def worker(id) :
    loop until no more work to do
        # get lowerBound and upperBound from the master
        ....
        partialSum = 0.0
        for i in range(lowerBound, upperBound):
            x = (float(i) - 0.5) / float(N)
            partisalSum = partisalSum + 4.0/(1.0 + x * x)
        # Send partitialSum to the master
        …
```

```
def worker() :
    while True:
        # get lowerBound and upperBound from the master
        lowerBound, upperBound = tQueue.get()
        if lowerBound < 0 or upperBound < 0:
            exit()
        partialSum = 0.0
        for i in range(lowerBound, upperBound):
            x = (float(i) - 0.5) / float(N)
            partisalSum = partisalSum + 4.0/(1.0 + x * x)
        # Send partitialSum to the master
        rQueue.put(partisalSum)
```

- Use two message queues
  - One (tQueue) for distributing tasks – multiple reads situation. OK, it does not matter which worker computes which part of the loop.
  - One (rQueue) for getting results
- Note that using just one queue can be problematic.

The master

```
# setup the queue, this must happen before start worker, why?
tQueue = multiprocessing.Queue()
rQueue = multiprocessing.Queue()
#setup and start the workers
p = list()
for i in range(0, nprocs):
    p.append(multiprocessing.Process(target=worker))

for i in range(0, nprocs):
    p[i].start()
# Distributed the tasks
chunk = 10000  # this is an important parameter for performance
if chunk > N // nprocs + 1:
    chunk = N // nprocs + 1

tasks = N // chunk
for i in range(0, tasks):
    tQueue.put((i*chunk, (i+1) * chunk))
if (tasks * chunk != N):
    tQueue.put((tasks*chunk, N))
    tasks = tasks + 1
```

The master (continue)

```
# make sure child processes can exit
for i in range(0, nprocs):
    tQueue.put((-1, -1)) # to make the workers stop

myPI = 0.0
# get partial sum from workers
for i in range(0, tasks):
    partialSum = rQueue.get()
    myPI += partialSum

myPI = myPI / float(N)
```

- See lect7/pi_mw.py

Build a list of prime numbers with multiprocessing

- Task:
  - Input N
  - Build a sorted list of all prime numbers whose values are no more than N (can be exactly N).

- See lect7/primes.py
  - Building the list of prime numbers has dependence
  - Computing larger primes depends on knowing smaller primes.
    - ❖ Cannot naively chops the space and distributes to different workers

Build a list of prime numbers with multiprocessing

- Task:
  - Input N
  - Build a sorted list of all prime numbers whose values are no more than N (can be exactly N).

- See lect7/primes.py
  - Building the list of prime numbers has dependence
  - Computing larger primes depends on knowing smaller primes.
    - ❖ Cannot naively chops the space and distributes to different workers

```
def isPrime(n):
    i = 0
    bound = int(math.sqrt(n) + 1)
    while primes[i] <= bound:
        if n % primes[i] == 0:
            return False
        i = i + 1
    return True


for i in range(6, N+1):
    if isPrime(i):
        primes.append(i)
```

Build a list of prime numbers with multiprocessing

- Separate the loop into two loops
  - Do the small loop sequentially by the main process
  - Parallelize the bigger loop
  - Task distribution method can be similar to that used in pi_mw.py
    - ❖ Workers need to add sentinels in the result data to inform the master that their computation is done.
    - ❖ Master receives primes not sorted – need to sort after receiving all data
- See primes_mw_sort.py

```
def isPrime(n):
    i = 0
    bound = int(math.sqrt(n) + 1)
    while primes[i] <= bound:
        if n % primes[i] == 0:
            return False
        i = i + 1
    return True

m = math.sqrt(N+2)
for i in range(6, m):
    if isPrime(i):
        primes.append(i)

for i in range(m, N+1)
    if isPrime(i):
        primes.append(i)
```

Parallel programming design pattern 2: Domain decomposition

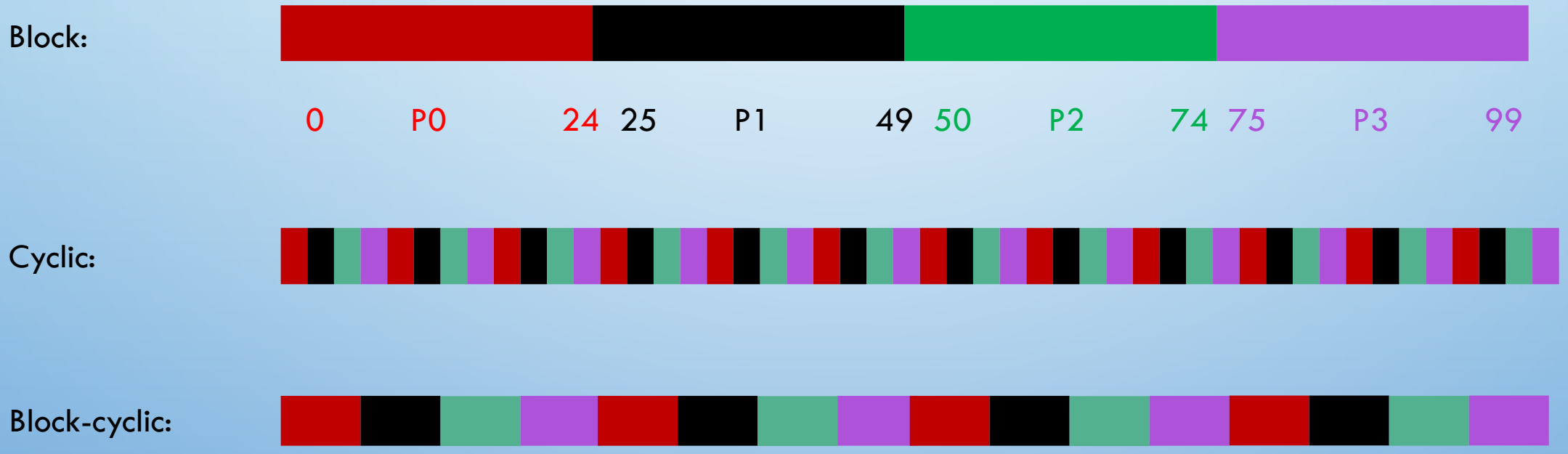
- Large simulations (e.g., climate modeling, fluid flow, heat transfer) often model the spatial world as grid or mesh and solving equations over the grid.
  - The finer the grid, the more accurate the model
  - Solving the model sequentially may take too much time
- Domain decomposition (partition the grid into sub-domains, and have each process performing the computation within one sub-domain) is a widely used technical to achieve parallel computing for such problems
  - Distribute workload (computation and data) across multiple processors
    - ❖ Speed up computation
    - ❖ Enable solving larger problems (the global grid is often too large to fit in the memory of one machine)

Domain Decomposition of 1D domain

- Example: Consider domain decomposition of 1D-domain of size 100 among 4 processes.

domain decomposition of 1D-domain

- In general, the domain can be partitioned in three different ways, block, cyclic, and block-cyclic.



Partitioning multi-dimensional domain

- With multiple dimensions, one can choose which dimension(s) to partition.
  - What is the best way to partition? Simplify and minimize the communication!

Steps in developing parallel program using domain decomposition (from a sequential code)

1. Break up the global domain into sub-domains. Assign each sub-domain to a process
2. Provide a "map" of all domains to each process (each process knows who "owns" which data).
3. Orchestra the computation
    1. Insert the communication and synchronization calls when necessary
    2. Modify the code (e.g. mapping local index to global index) to find the domain portion for each process, and only compute the domain portion

- Note 1: In most applications, the computation for each domain requires data from another domain, resulting in communication!
    - The communication requirement is what decides whether a partitioning is a good partitioning.
- Note 2: two variables: myrank and nprocs are often used to decide which sub-domain to work on for each process

Domain decomposition example

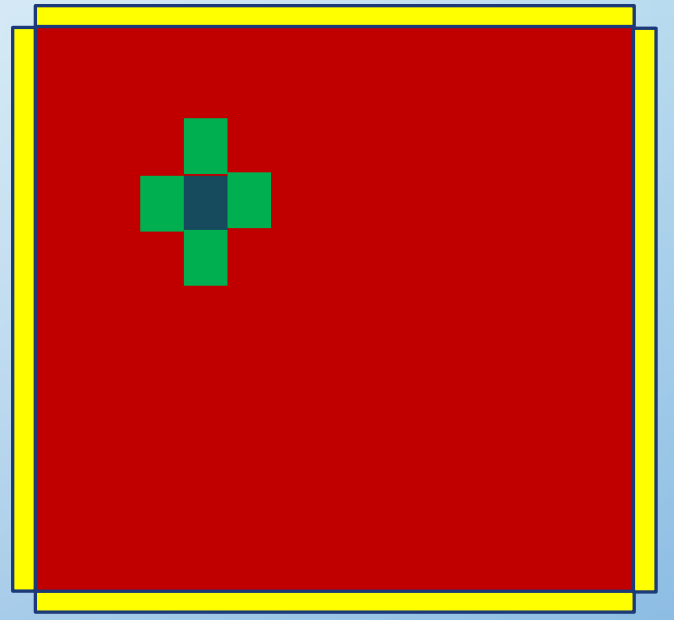
- Heat distribution problem
  - Given a 2D rectangular metal plate, we know the temperatures at the boundary, we want to find out the temperatures spreading across the whole plate at thermal equilibrium.
    - We can model the plate as a 2D grid such as 10x10 or 1000x1000 cells, we want to find out the temperature at every interior grid point.
    - The physics to solve this equation is the Laplace Equation: +
      - At equilibrium, each interior point's temperature is the **average of its four neighboring points** (up, down, left, right)
- The Jacobi Method is a classical iterative algorithm to solve this type of problem

Jacobi method for heat distribution

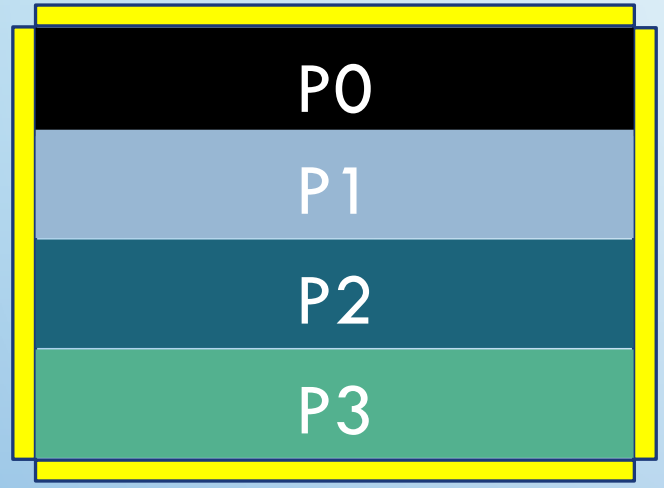
- Step 1: initialize the boundary values and values for interior points
- Step 2:
  - Iterate
    - ❖ At each iteration for every point (i, j), update to a new temperature:
    - ❖ Until the maximum difference between and is less than a threshold

Example: Jacobi method (jacobi.py)

```
for some number of timesteps/iterations {
   for (i=0; i<n; i++ )
         for( j=1, j<n, j++ )
               temp[i][j] = 0.25 *
                     ( grid[i-1][j] + grid[i+1][j]
                     grid[i][j-1] + grid[i][j+1] );
   for( i=0; i<n; i++ )
         for( j=1; j<n; j++ )
               grid[i][j] = temp[i][j];
}
```



Domain deposition Step 1: Partition the domain

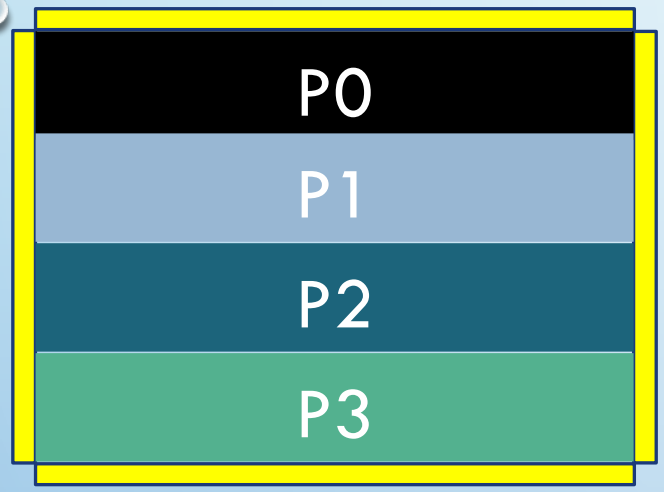


Given the global domain grid[0..N+1, 0..N+1], grid[0][_], grid[_][0], grid[_][N+1], grid[N+1][_] are the boundary condition that does not need to be updated.

How to decide which sub-domain each process should perform the computation?
We need to derive the range from myrank, nprocs, and N: which range should process 0 compute?

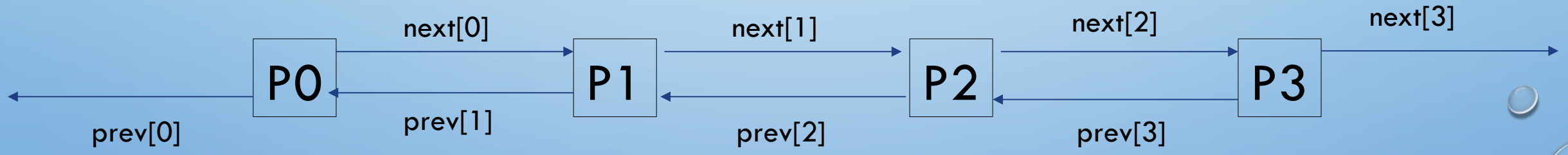
See jacobi_mp_v0.py

Domain deposition Step 2: Orchestra the computation



jacobi_mp_v0.py is incorrect, there are several issues: The first one is that the diff computed at each iteration is local to each process, we need to find the maximum diff across all processes!

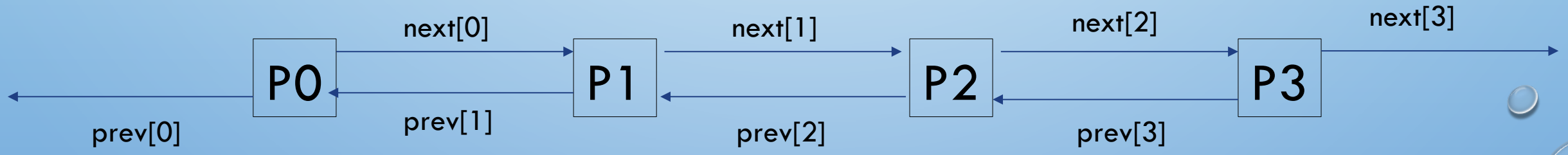
Need to setup message queues and find the correct diff
See jacobi_mp_v0.py



Domain deposition Step 2: Orchestra the computation

jacobi_mp_v1.py
Set up the message queue:

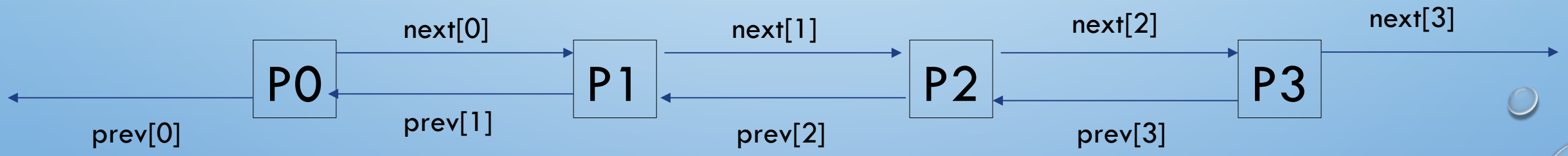
```
next = []
prev = []
for i in range(nprocs):
    next.append(multiprocessing.Queue())
    prev.append(multiprocessing.Queue())
```



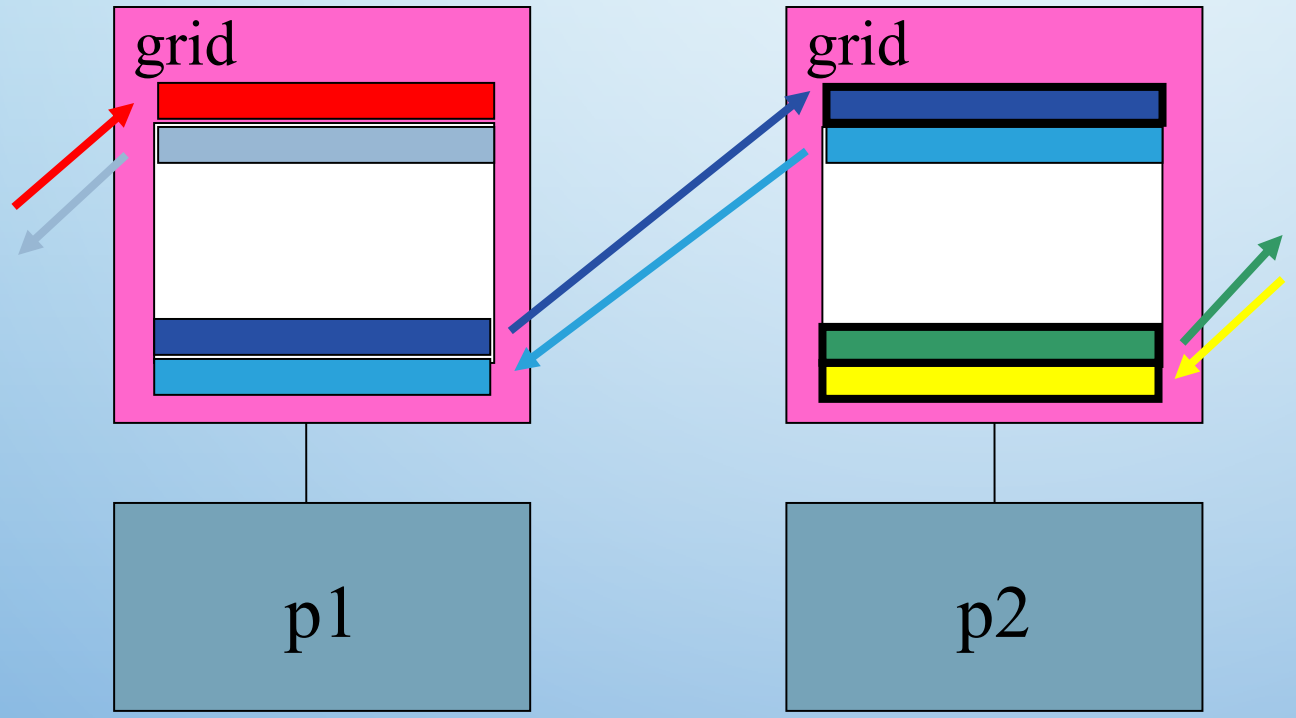
Domain deposition Step 2: Orchestra the computation

```
Compute the global diff (part 1)
        if (myId == nprocs -1) :
            prev[myId].put(maxdiff1)
        elif (myId == 0) :
            tmp = prev[myId+1].get()
            maxdiff1 = tmp if tmp > maxdiff1 else maxdiff1
        else : # this is for all processes from 1 to nprocs-2
            tmp = prev[myId+1].get()
            maxdiff1 = tmp if tmp > maxdiff1 else maxdiff1
            prev[myId].put(maxdiff1)
```

```
Compute the global diff (part 2)
        if (myId ==0) :
            next[myId].put(maxdiff1)
        elif (myId == nprocs -1):
            maxdiff1 = next[myId-1].get()
        else : # all in between
            maxdiff1 = next[myId-1].get()
            next[myId].put(maxdiff1)
```



Orchestra the computation: Boundary elements



- Processes 0, 1, 2 send lower row to Processes 1,2 3.
- Processes 1, 2, 3 receiver upper row from processes 0, 1, 2
- Process 1, 2, 3 send the upper row to processes 0, 1, 2
- Processes 0, 1, 2 receive the lower row from processes 1, 2,3

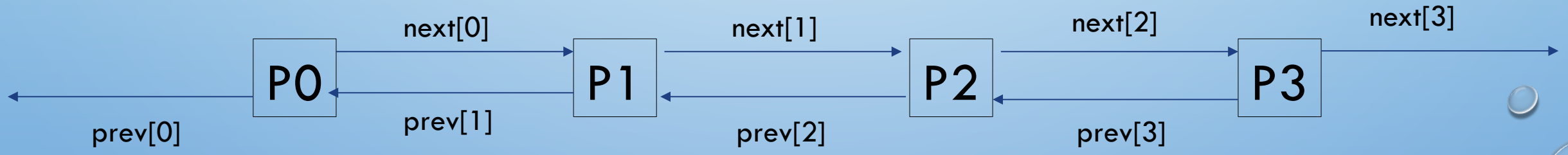
Domain deposition Step 2: Orchestra the computation
See jacobi_mp_v2.py

Communicate boundary elements (send south)

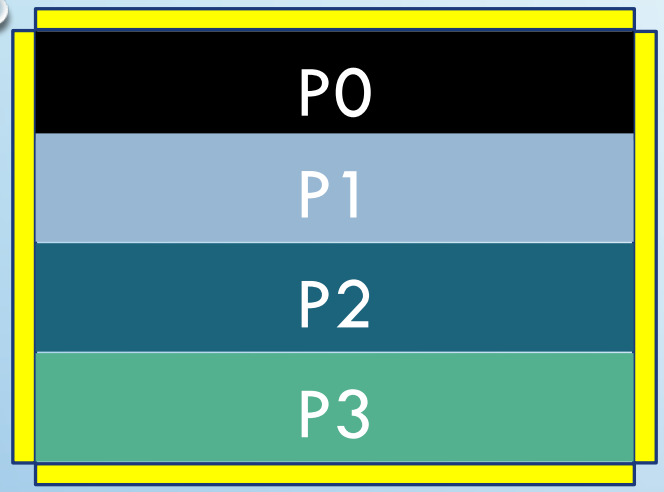
```
if (myId == nprocs -1):
    row = next[myId-1].get()
    for i in range(N+2):
        x[myId*blockSize][i] = row[i]
elif (myId == 0):
    next[myId].put(x[(myId+1)*blockSize])
else:
    row = next[myId-1].get()
    for i in range(N+2):
        x[myId*blockSize][i] = row[i]
    next[myId].put(x[(myId+1)*blockSize])
```

Communicate boundary elements (send north)

```
if (myId == nprocs -1):
        row = next[myId-1].get()
        for i in range(N+2):
            x[myId*blockSize][i] = row[i]
    elif (myId == 0):
        next[myId].put(x[(myId+1)*blockSize])
    else:
        row = next[myId-1].get()
        for i in range(N+2):
            x[myId*blockSize][i] = row[i]
        next[myId].put(x[(myId+1)*blockSize])
```



Parallel jacobi: output the results



The correct results of the grid are distributed in different processes. What to do the output the results?

All processes except P0 send their data to P0
P0 outputs its own data, and then receives data from each Process (P1, P2, P3) and output. You can create another list of message queues to achieve this.

See jacobi_mp_v3.py

Parallel Jacobi: Reduce the memory footprint

- In jacobi_mp_v3.py, each process allocates space for the global array.
- In a production software, each process should only allocate space for a portion of the global grid.
  - Need to establish the mapping between local grid indices to global grid indices (derives from myrank, nprocs, and N).