Parallelism the Old Way: Using MPI in Python with mpi4py

PyData London, 17 June 2022

Nicholas J. Radcliffe
Stochastic Solutions Limited
& Department of Mathematics, University of Edinburgh
& Smart Data Foundry
formerly: Edinburgh Parallel Computer Centre (EPCC)

https://github.com/stochasticsolutions/mpi4py-pydatalondon/blob/main/slides/mpi-pydata-london-2022.pdf

RESOURCES

- **CODE & SLIDES FROM THIS TUTORIAL**

  https://github.com/stochasticsolutions/mpi4py-pydatalondon

  ```
  git clone https://github.com/stochasticsolutions/mpi4py-pydatalondon.git
  ```

- **DOCUMENTATION FOR mpi4py**

  https://mpi4py.readthedocs.io

- **REWORKED (ENHANCED?) EXAMPLES FROM THE mpi4py TUTORIAL**

  https://github.com/stochasticsolutions/mpi4py-examples

  ```
  git clone https://github.com/stochasticsolutions/mpi4py-examples.git
  ```

PARALLEL PROCESSING
CONCURRENT PROCESSING
DISTRIBUTED PROCESSING
CLUSTERED COMPUTING
GRID COMPUTING

ASYNCHRONOUS
NON-BLOCKING
BUFFERING

PROCESSES
THREADS

AMDAHL'S LAW
SPEEDUP

SPMD
VECTORIZATION
DATA PARALLELISM
FUNCTIONAL PARALLELISM
PIPELINE PROCESSING
COMMUNICATING SEQUENTIAL PROCESSES (CSP)

DEADLOCK
LIVELOCK
MARSHALLING
LOCK • MUTEX
CONTENTION
COPY-ON-WRITE
QUEUES

MPP

GIL
ASYNCIO
ASYNC/AWAIT
PICKLING

SISD
SIMD
MIMD

(SYMMETRIC) MULTIPROCESSING
DISTRIBUTED MEMORY
SHARED MEMORY
SHARED NOTHING
MESSAGE-PASSING
NUMA

EMBARRASSINGLY
PARALLEL

MPI
OPEN-MP

CPUS
CORES
VIRTUAL CORES

FORKING
SPAWNING
BROADCAST
MAP-REDUCE
SCATTER-GATHER

Parallelism & Python

Parallelism & Python

PARALLEL PROCESSING
~~CONCURRENT PROCESSING~~
DISTRIBUTED PROCESSING
~~CLUSTERED COMPUTING~~
~~GRID COMPUTING~~

ASYNCHRONOUS
NON-BLOCKING
BUFFERING

PROCESSES
~~THREADS~~

AMDAHL'S LAW
SPEEDUP

SPMD
~~VECTORIZATION~~
DATA PARALLELISM
FUNCTIONAL PARALLELISM
~~PIPELINE PROCESSING~~
COMMUNICATING SEQUENTIAL PROCESSES (CSP)

DEADLOCK
LIVELOCK
MARSHALLING
~~LOCK • MUTEX~~
~~CONTENTION~~
~~COPY-ON-WRITE~~
~~QUEUES~~

MPP

GIL
~~ASYNCIO~~
~~ASYNC/AWAIT~~
PICKLING

SISD
SIMD
MIMD

(SYMMETRIC) MULTIPROCESSING
DISTRIBUTED MEMORY
SHARED MEMORY
SHARED NOTHING
MESSAGE-PASSING
~~NUMA~~

EMBARRASSINGLY PARALLEL

MPI
~~OPEN-MP~~

CPUS
CORES
~~VIRTUAL CORES~~

~~FORKING~~
SPAWNING
BROADCAST
~~MAP-REDUCE~~
SCATTER-GATHER

Parallel Computing

- Serial Computing: Do one thing (computation) at time, one after another
  - Sequential Computing
- Parallel Computing: Do several things at the same time
  - More than one processor coöperating to solve a single problem together

(Related to) Parallel Computing

- Clustered Computing/Compute Cluster:
  - Several computers, usually each with tight connectivity and own memory, connected together a bit less tightly; can be used for parallel computing, but communication between nodes in the cluster is more expensive than within a node.
- Concurrency: Executing several tasks in overlapping time periods, rather than sequentially.
  - Tasks may be unrelated, e.g. different user's programs; user programs & system processes
  - Can have concurrency on a single, serial processor
  - Think time-sharing; interrupts; threads

(Parallel) Computing Architectures

- SISD — Single Instruction, Single Data
- SIMD — Single Instruction, Multiple Data
- Vector Processing
- MIMD — Multiple Instruction, Multiple Data

SISD:
Single Instruction,
Single Data
("normal", serial/sequential execution)

Testing Primality

```python
import math

def is_prime(N):
    n = int(math.sqrt(N))  + 1
    if N < 5:
        return N in (2, 3)
    if N % 2 == 0:
        return False
    for i in range(3, n, 2):
        if N % i == 0:
            return False
    return True

print(' '.join(str(n) for n in range(1, 100) if is_prime(n)))
```

```
$ python is_prime.py
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
$
```

Testing Primality: `is_prime(29)`

```
def is_prime(N):
    n = int(math.sqrt(N))  + 1

    if N < 5:
        return N in (2, 3)

    if N % 2 == 0:
        return False

    for i in range(3, n, 2):
        if N % i == 0:
            return False




    return True
```

```
N = 29
n = int(math.sqrt(29)) + 1  #  6

if N < 5:                       # 6 > 5: False
    return N in (2, 3)

if N % 2 == 0:          # 29 % 2 = 1: False
    return False

for i in range(3, n, 2):  # i = 3
    if N % i == 0:  # 29 % 3 = 2: False
        return False

for i in range(3, n, 2):  # i = 5
    if N % i == 0:  # 29 % 5 = 4: False
        return False
# Fall out
return True
```


Stochastic Solutions

SIMD:
Single Instruction, Multiple Data

SIMD Machines

ICL DAP
(Distributed Array Processor)

Thinking Machine CM-200
"Connection Machine"

Stochastic Solutions

SIMD: Single Instruction, Multiple Data

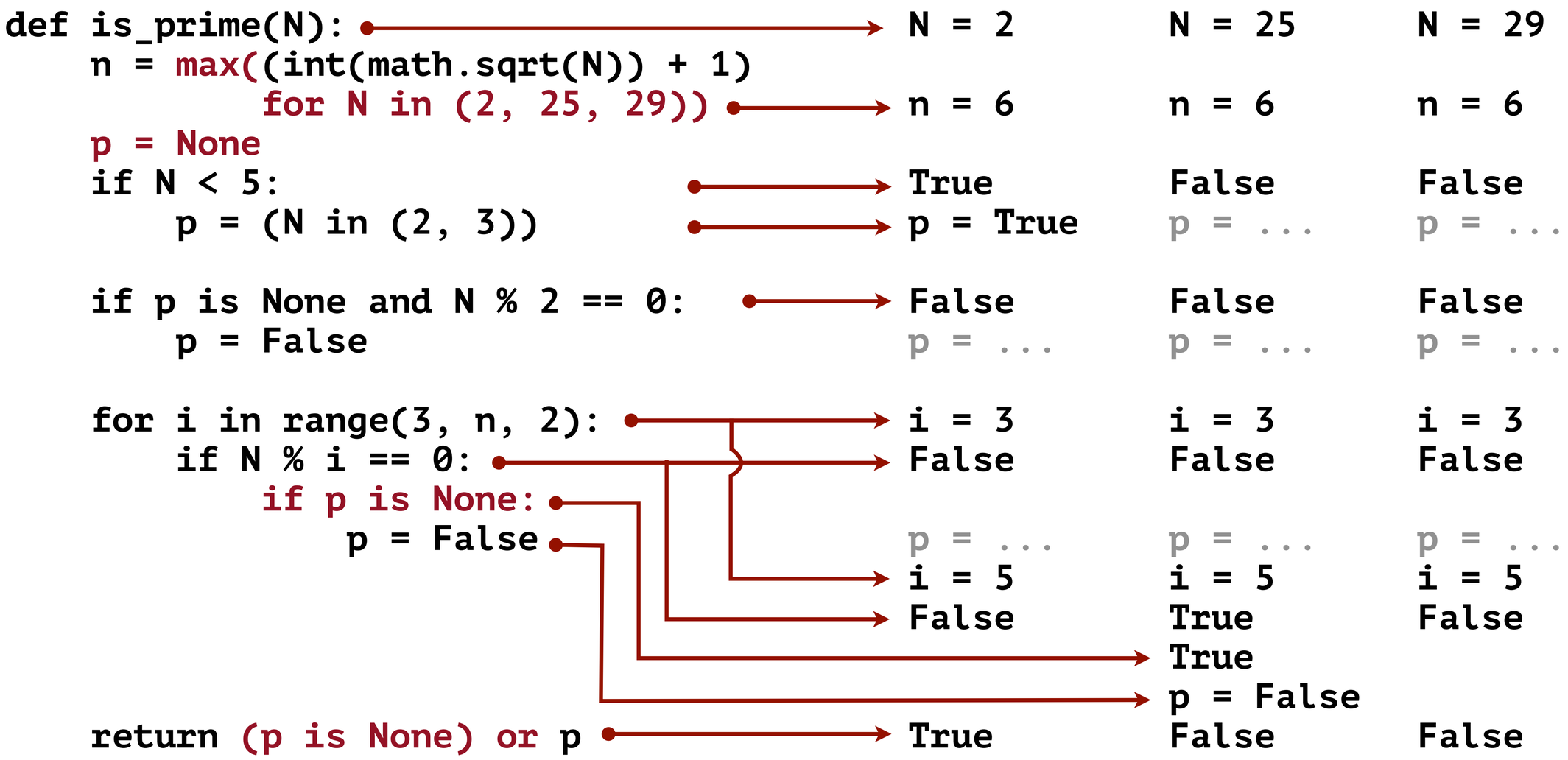
- Single program
- Single instruction pointer
- Executes on multiple processors, with different data, *in lockstep*
  - *All branches execute on all data*
  - *No early exit for loops/functions on some branches only*
  - *Ideal for graphics (GPUs; same operations on entire image), physical simulations etc.*

SIMD: Single Instruction, Multiple Data

Input	1	3	7	1	0	0	8	12
Multiply by 2	2	6	14	2	0	0	16	24
Add 3	5	9	17	5	3	3	19	27
Where > 10, negate elsewhere, double	10	18	-17	10	6	6	-19	-27
Cube	1000	5832	-4913	1000	216	216	-6859	19683

Stochastic Solutions

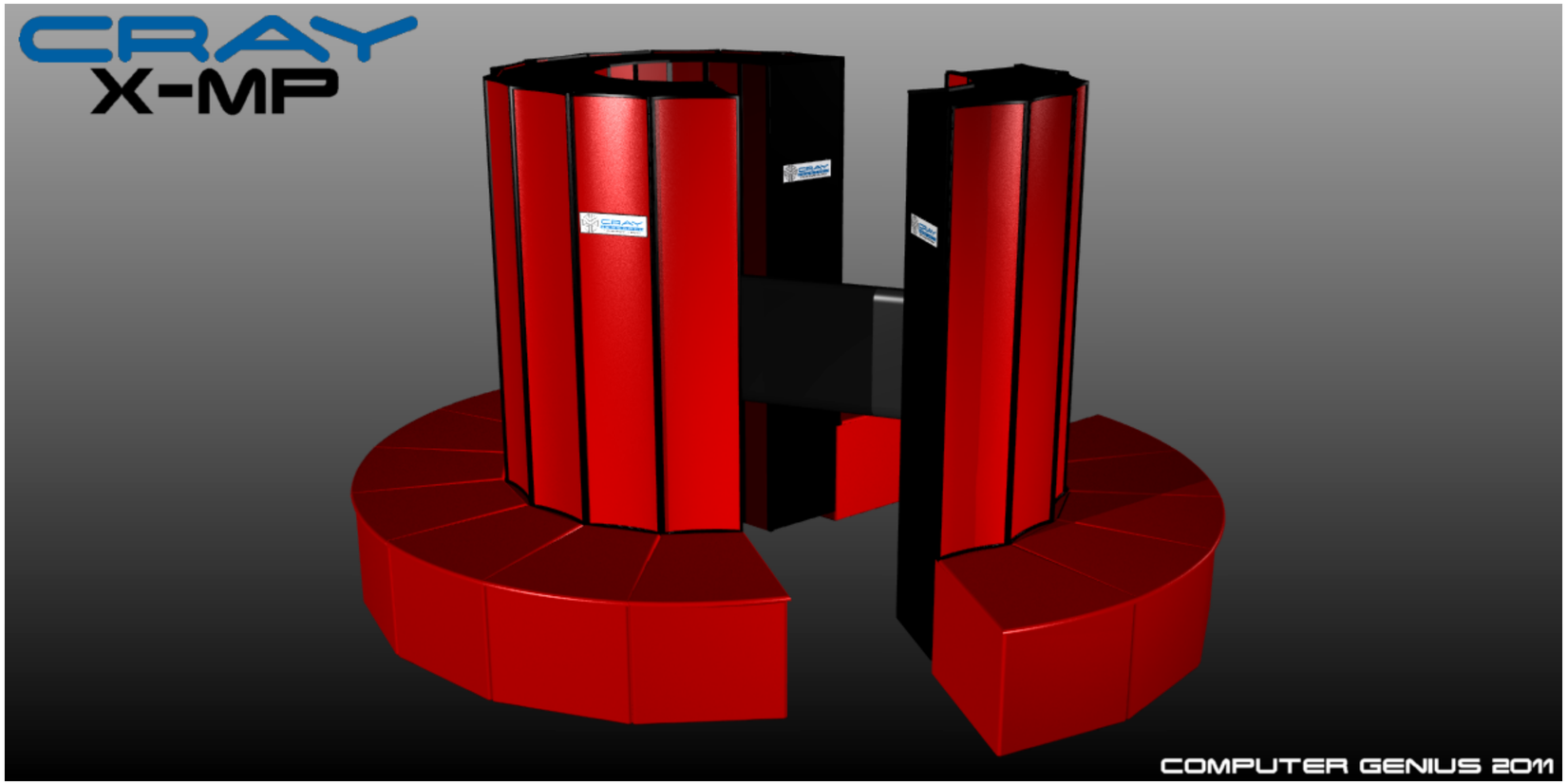
SIMD: Single Instruction, Multiple Data



Stochastic Solutions

Vector Processing
Multiple Data Passes
Through a Series of Execution Units

Vector Processors

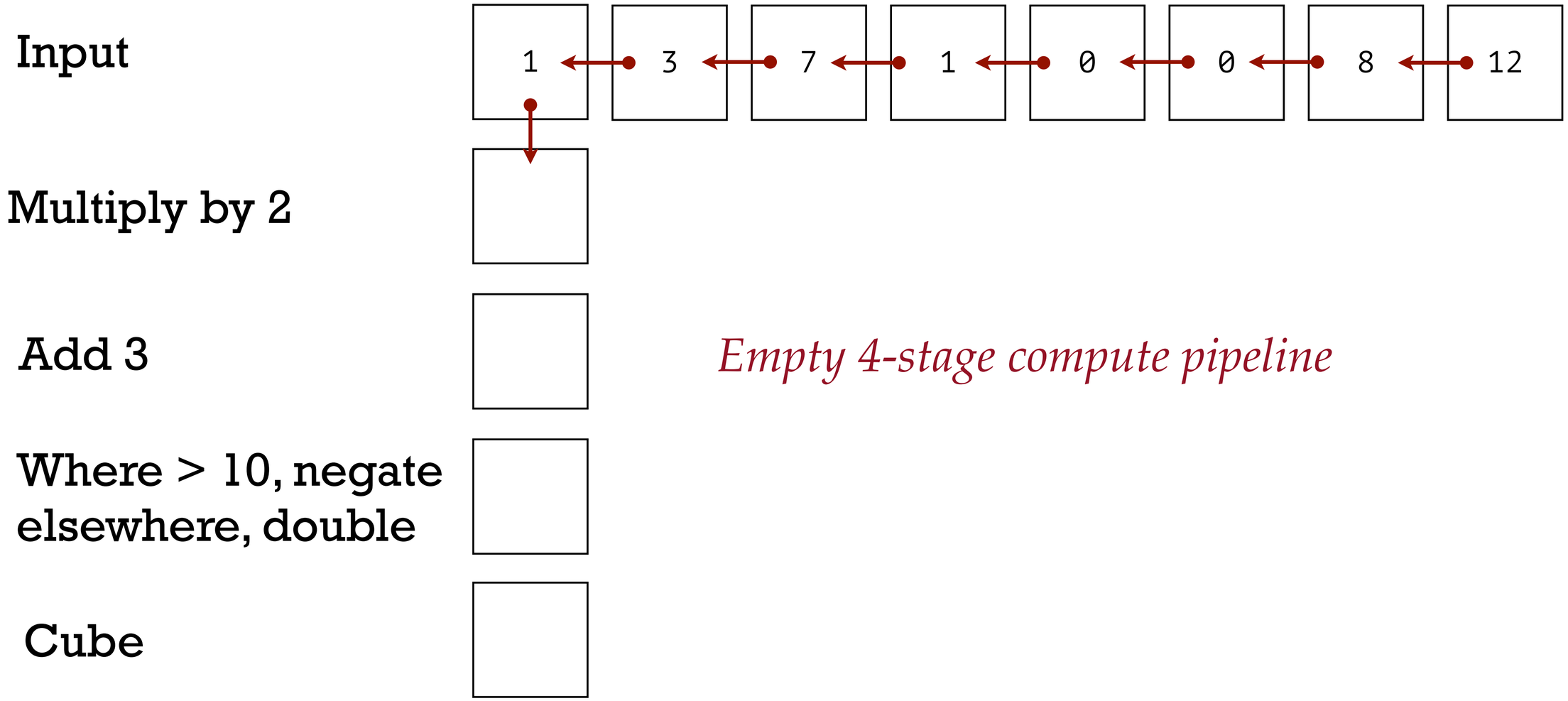


Stochastic Solutions

Vector Processing

- Single program
- Multiple execution units performing different functions (fixed or programmable)
- Data passes through the pipeline, moves through the execution units in turn
- Once the pipeline fills, an N-stage pipeline executes N instructions each clock cycle.

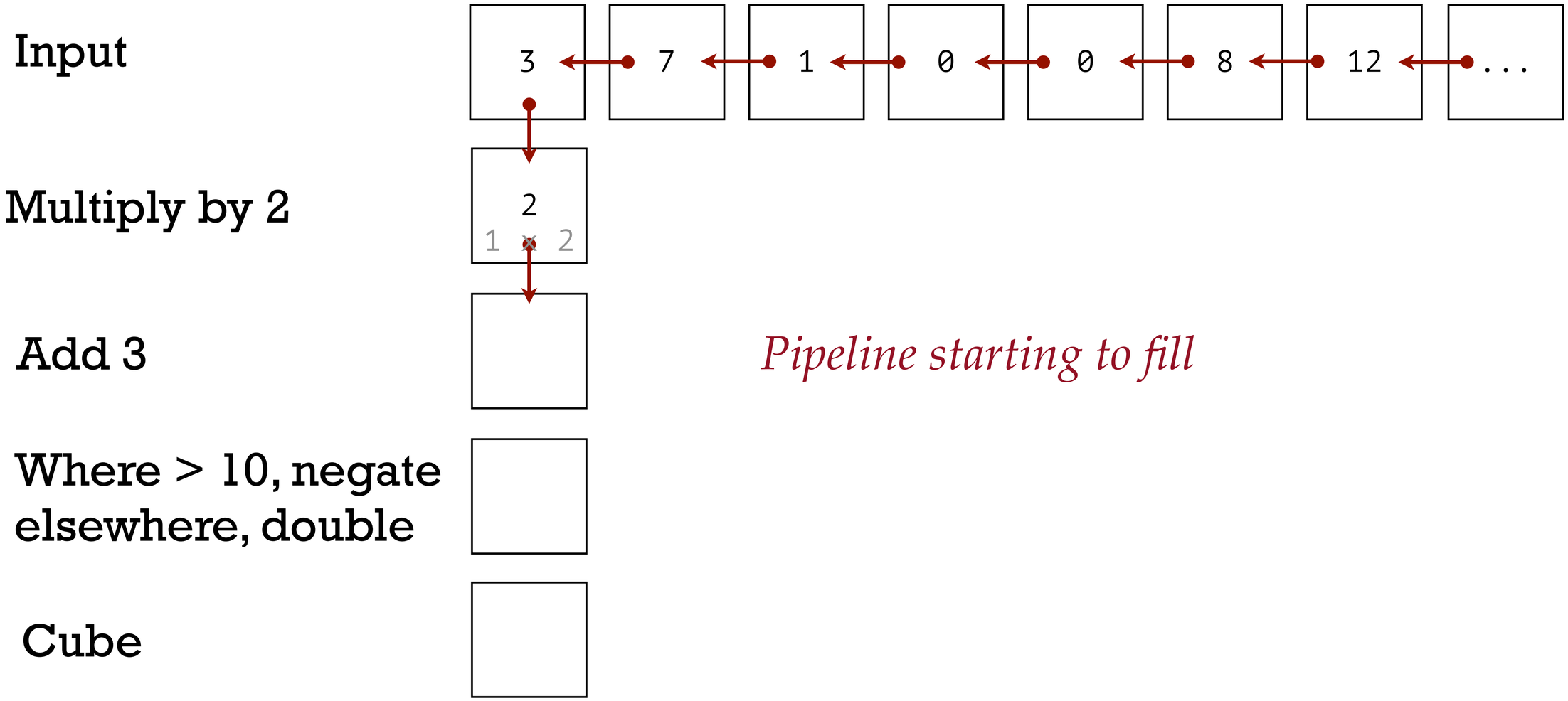
Vector Processing



Empty 4-stage compute pipeline

Stochastic Solutions

Vector Processing



Stochastic Solutions

Vector Processing

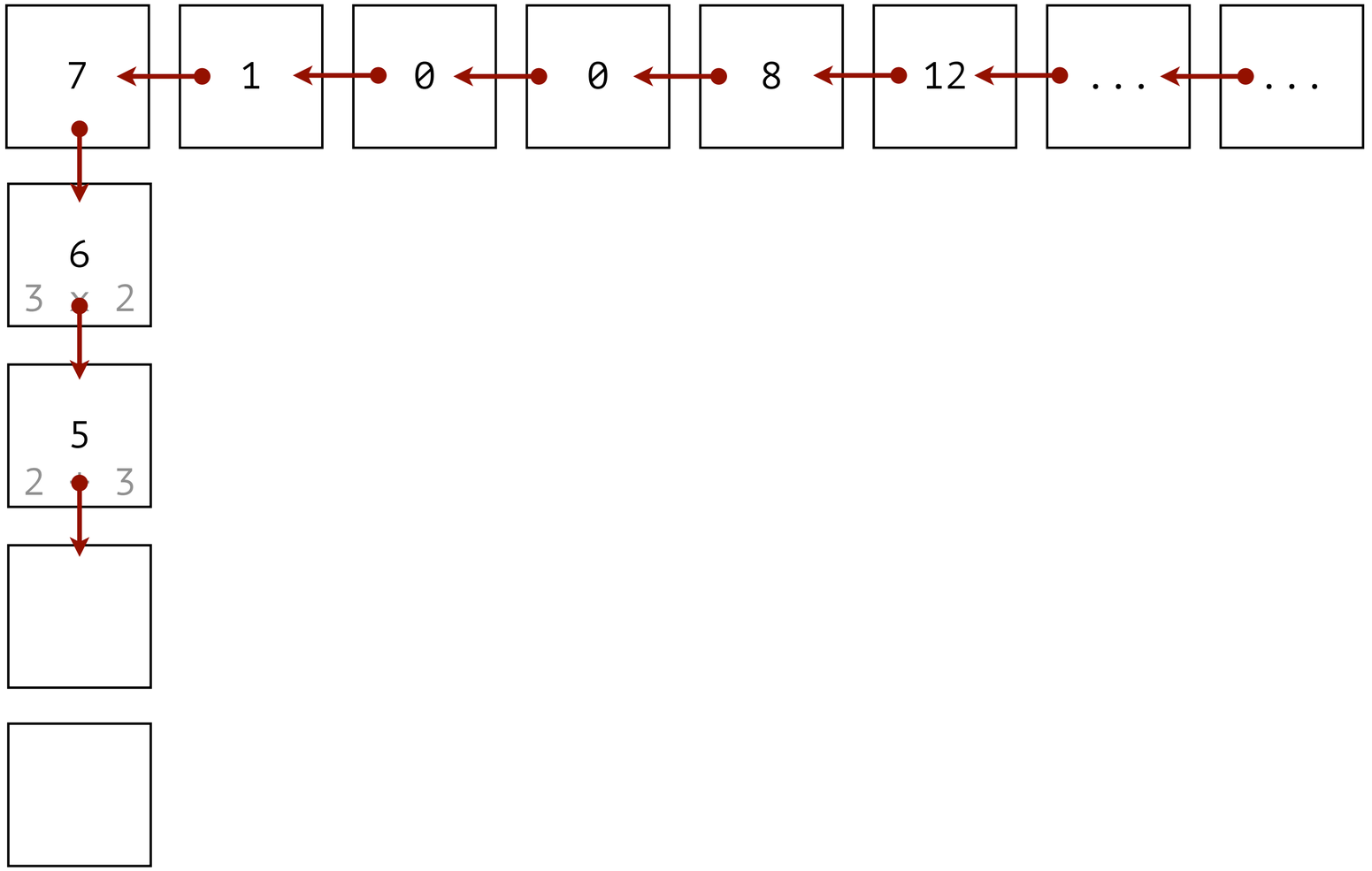
Input

Multiply by 2

Add 3

Where > 10, negate elsewhere, double

Cube



Stochastic Solutions

Vector Processing

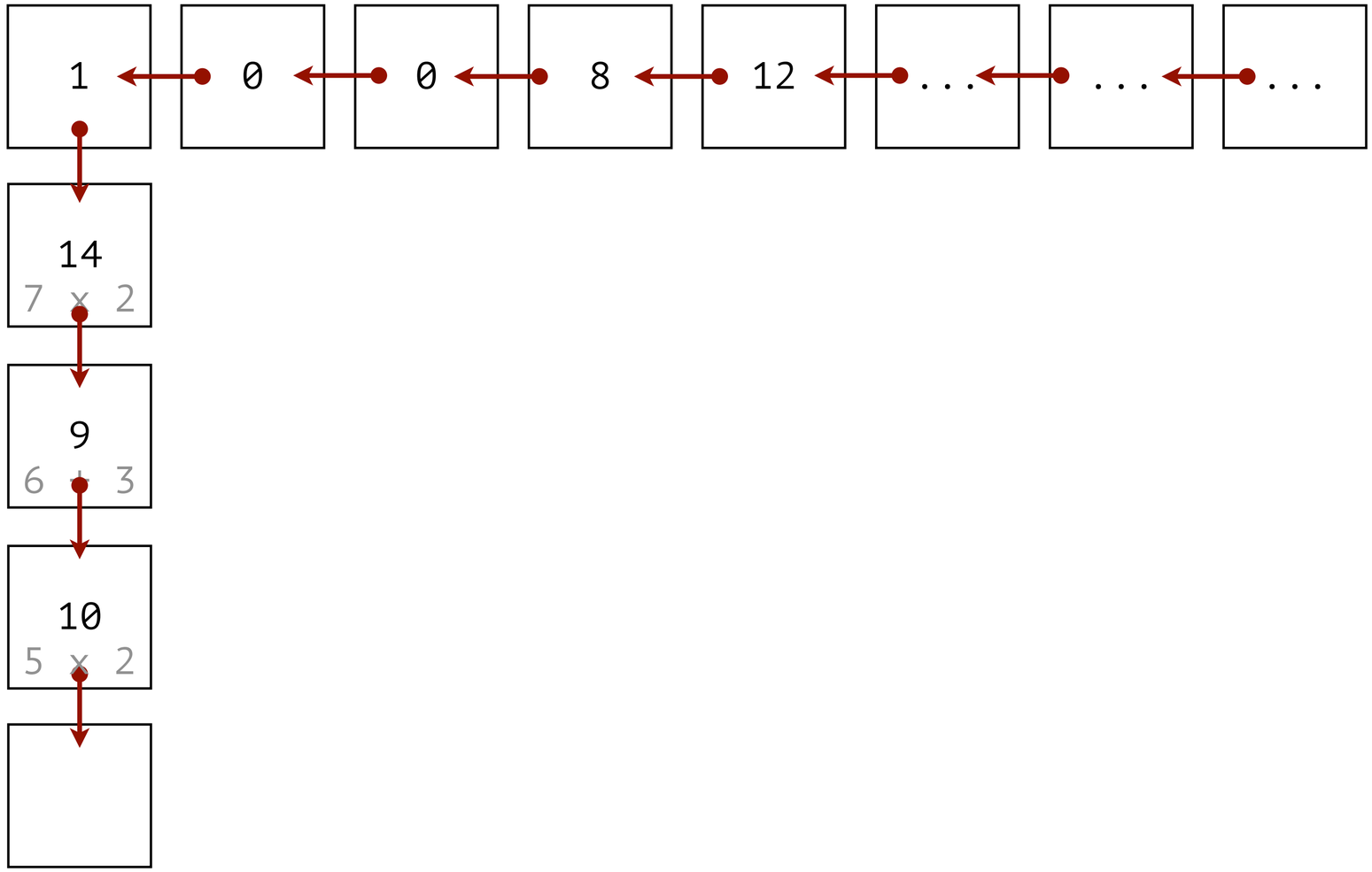
Input

Multiply by 2

Add 3

Where > 10, negate elsewhere, double

Cube



Stochastic Solutions

Vector Processing

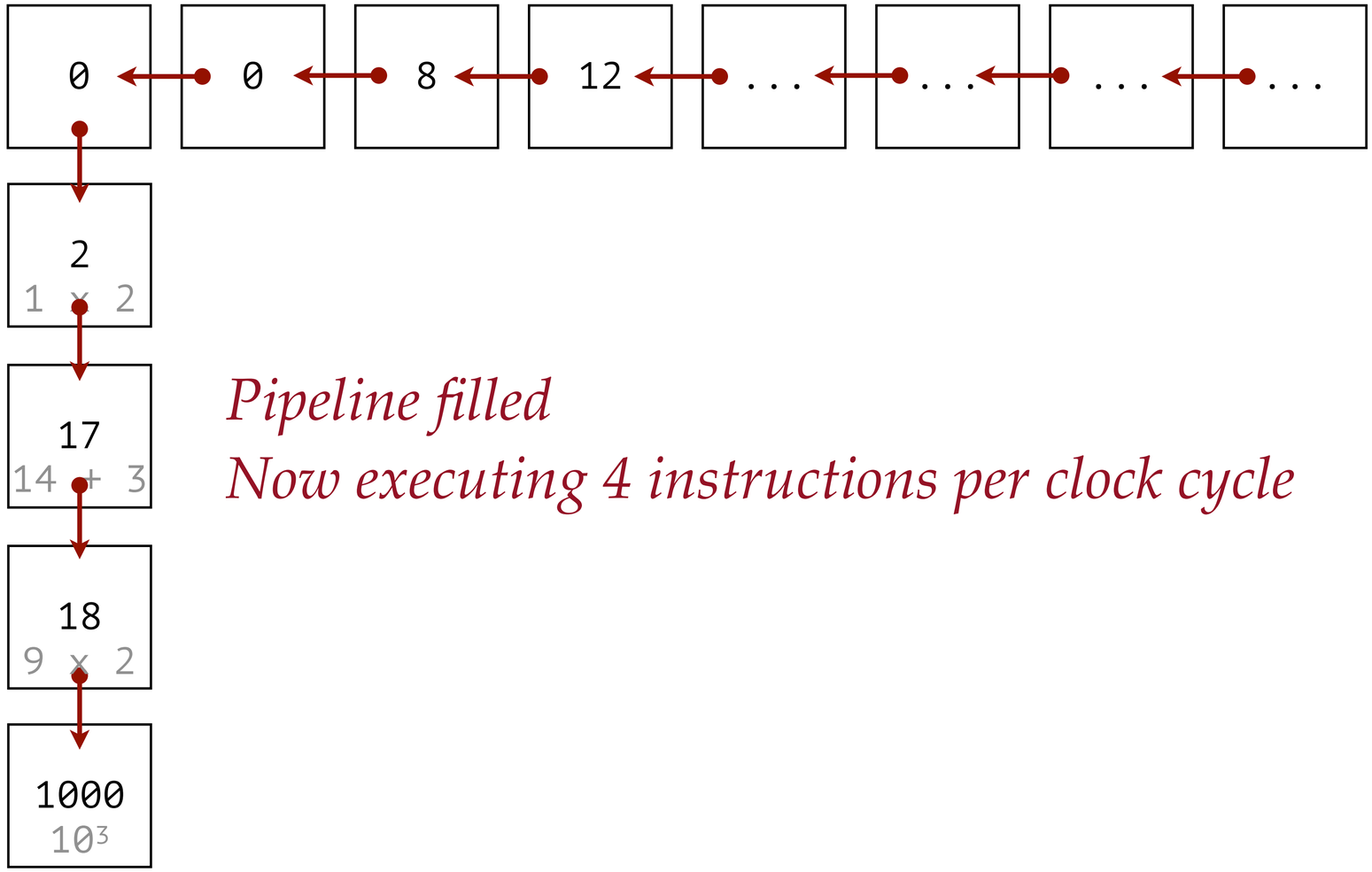
Input

Multiply by 2

Add 3

Where > 10, negate elsewhere, double

Cube

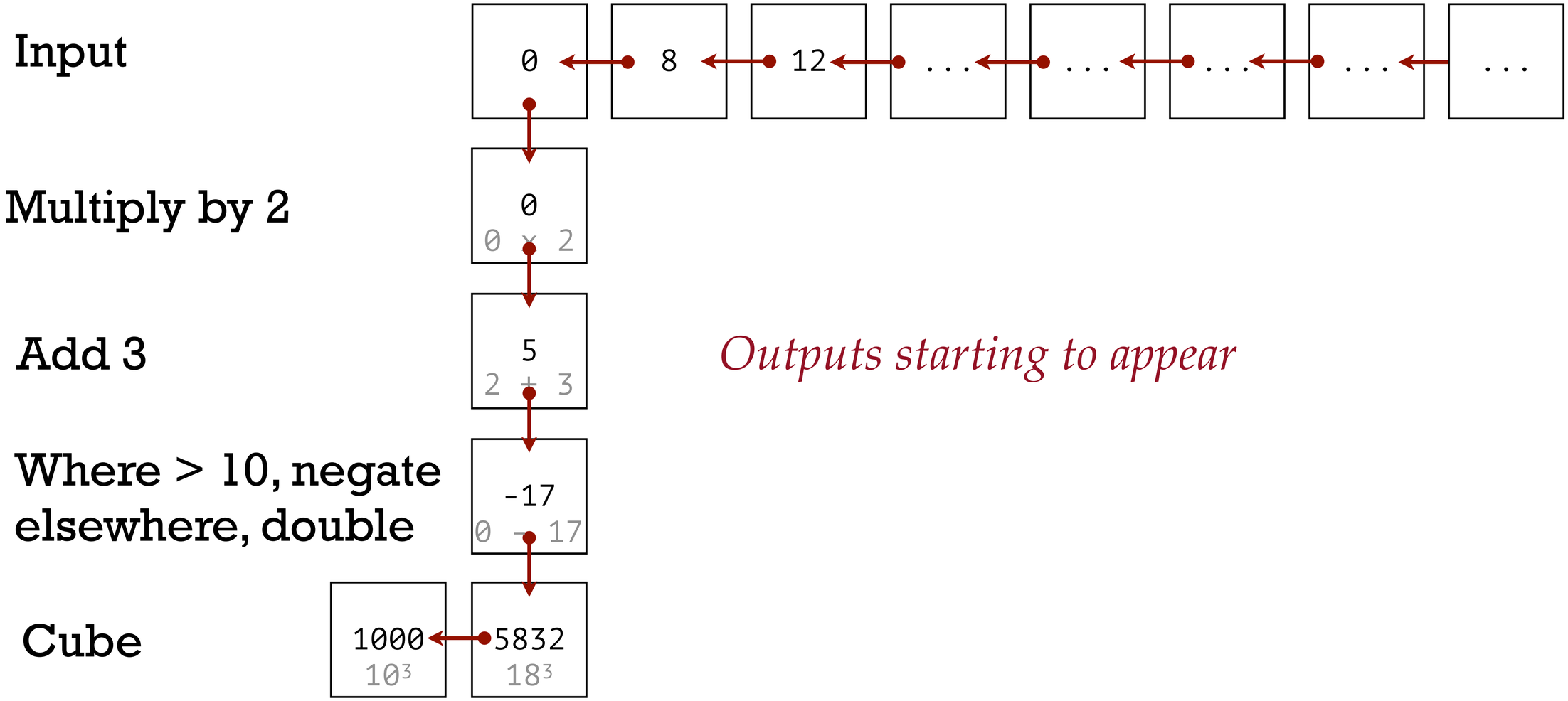


Pipeline filled

Now executing 4 instructions per clock cycle

Stochastic Solutions

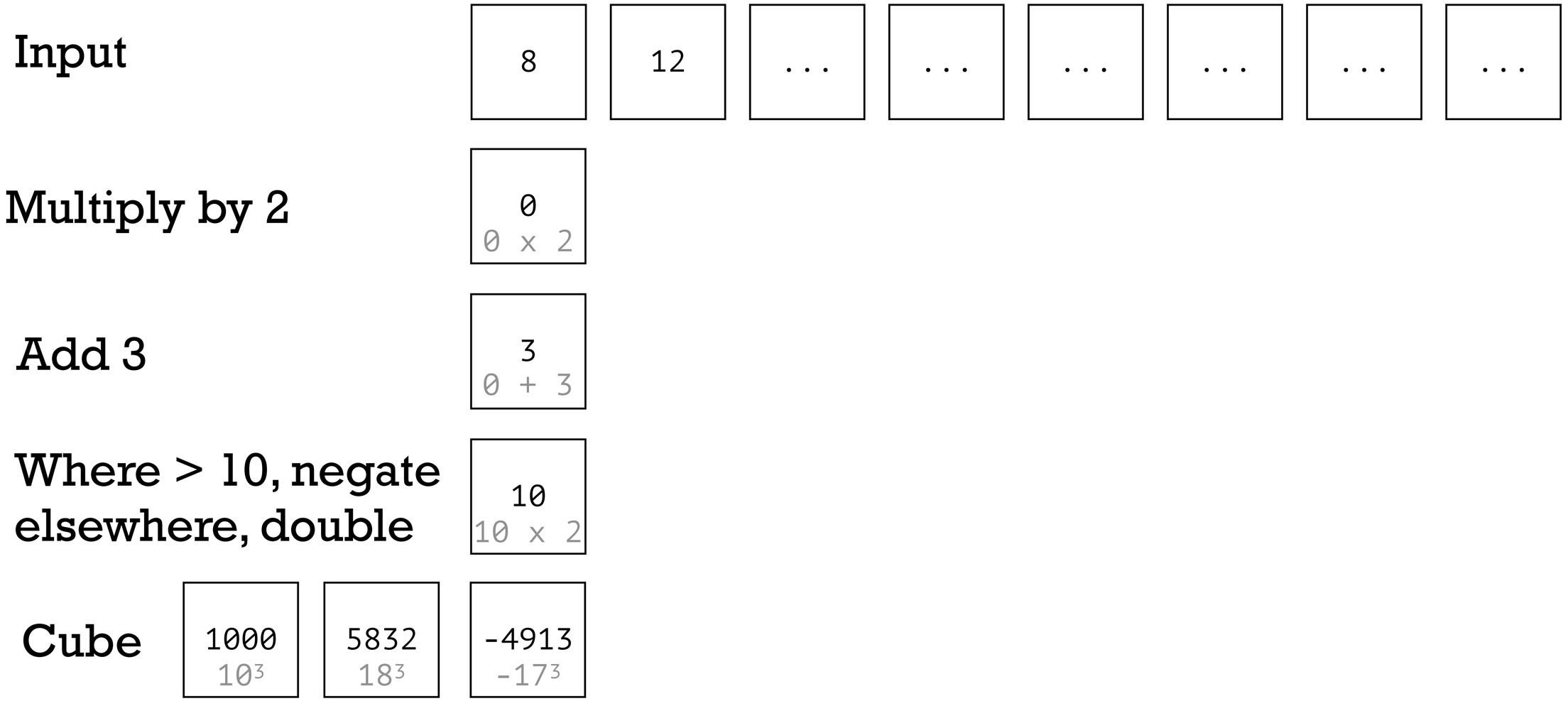
Vector Processing



Outputs starting to appear

Stochastic Solutions

Vector Processing



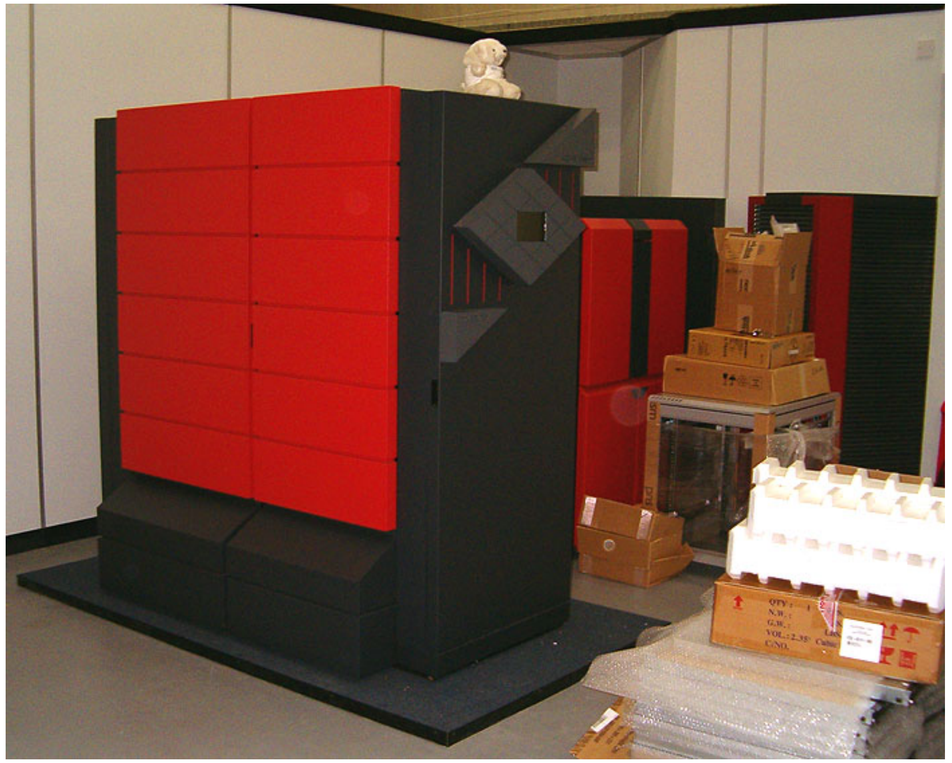
Stochastic Solutions

MIMD
Multiple Instruction, Multiple Data

This is what we'll be concentrating on

MIMD Processors

Meiko Computing Surface



Cray T3D

MIMD: Multiple Instruction, Multiple Data

- A (potentially) different program on each processor
  - in practice, very often the same program; so-called *Single Program, Multiple Data* (SPMD)
- Each processor has own instruction pointer: they run independently apart from any communications, shared resources etc.

PROCESSOR 1

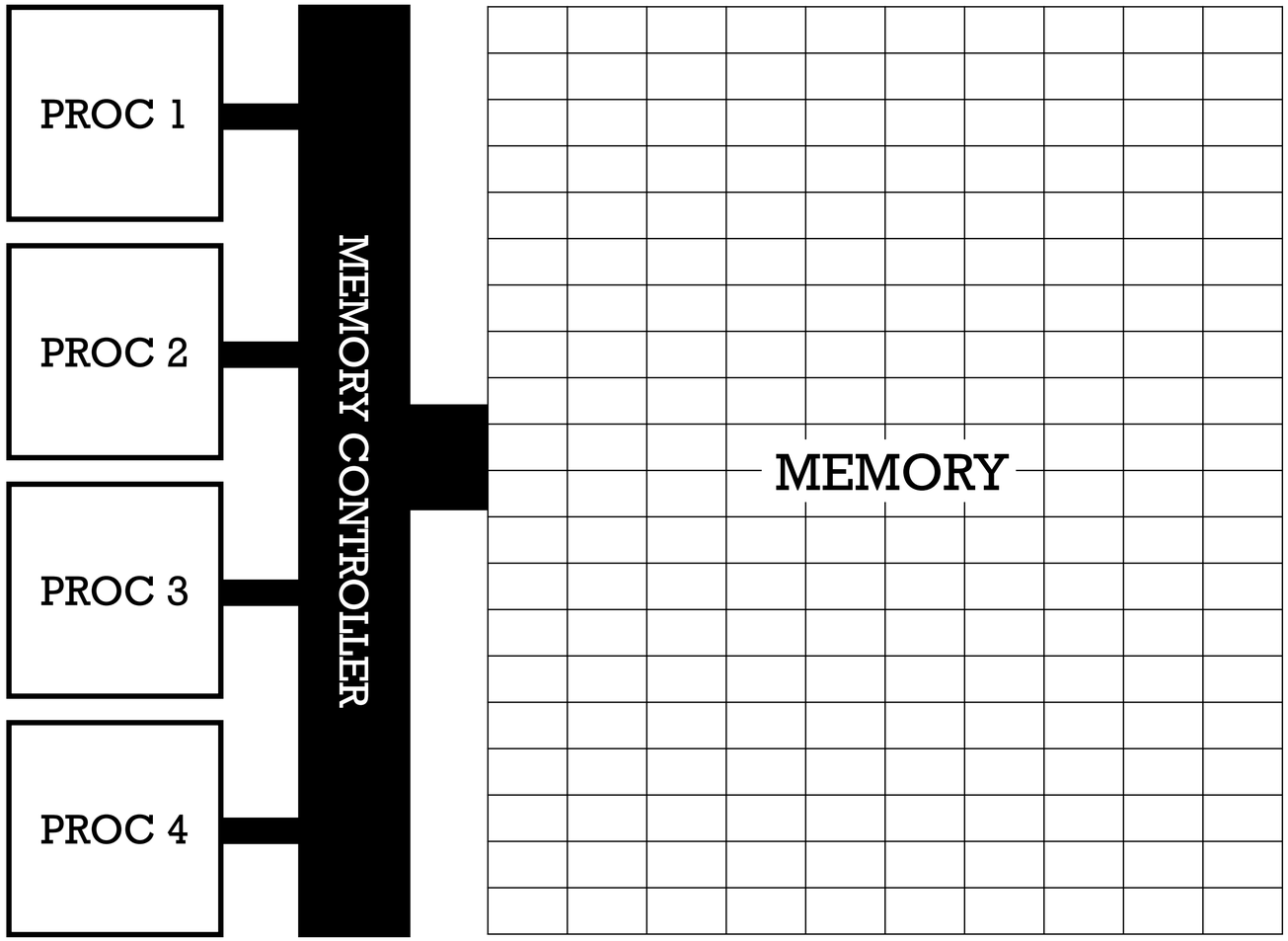
```
is_prime(29):
N = 29
n = int(math.sqrt(29)) + 1  # 6

if N < 5:          # 6 > 5: False
    return N in (2, 3)

if N % 2 == 0:  # 29 % 2 = 1: False
    return False

for i in range(3, n, 2):  # i = 3
    if N % i == 0:        # False
        return False

for i in range(3, n, 2):  # i = 5
    if N % i == 0:        # False
        return False
# Fall out
return True

is_prime(31):

    ⋮
```

PROCESSOR 2 ...

```
is_prime(4):
N = 4
n = int(math.sqrt(4)) + 1  # 3

if N < 5:          # 4 < 5: True
    return N in (2, 3)  # return False

is_prime(6):
N = 4
n = int(math.sqrt(4)) + 1  # 3

if N < 5:          # 6 < 5: False
    return N in (2, 3)  # False
if N % 2 == 0:  # 6% 2 = 0: True
    return False

is_prime(8):
N = 8
n = int(math.sqrt(4)) + 1  # 3

    ⋮
```

MIMD Architectures: Shared Memory *vs.* Distributed Memory

Shared Memory



PROC = PROCESSOR *or* CORE *or* CPU

All processors access
a single pool of memory

If access speeds are equal for
all processors, known as
"symmetric multiprocessing"

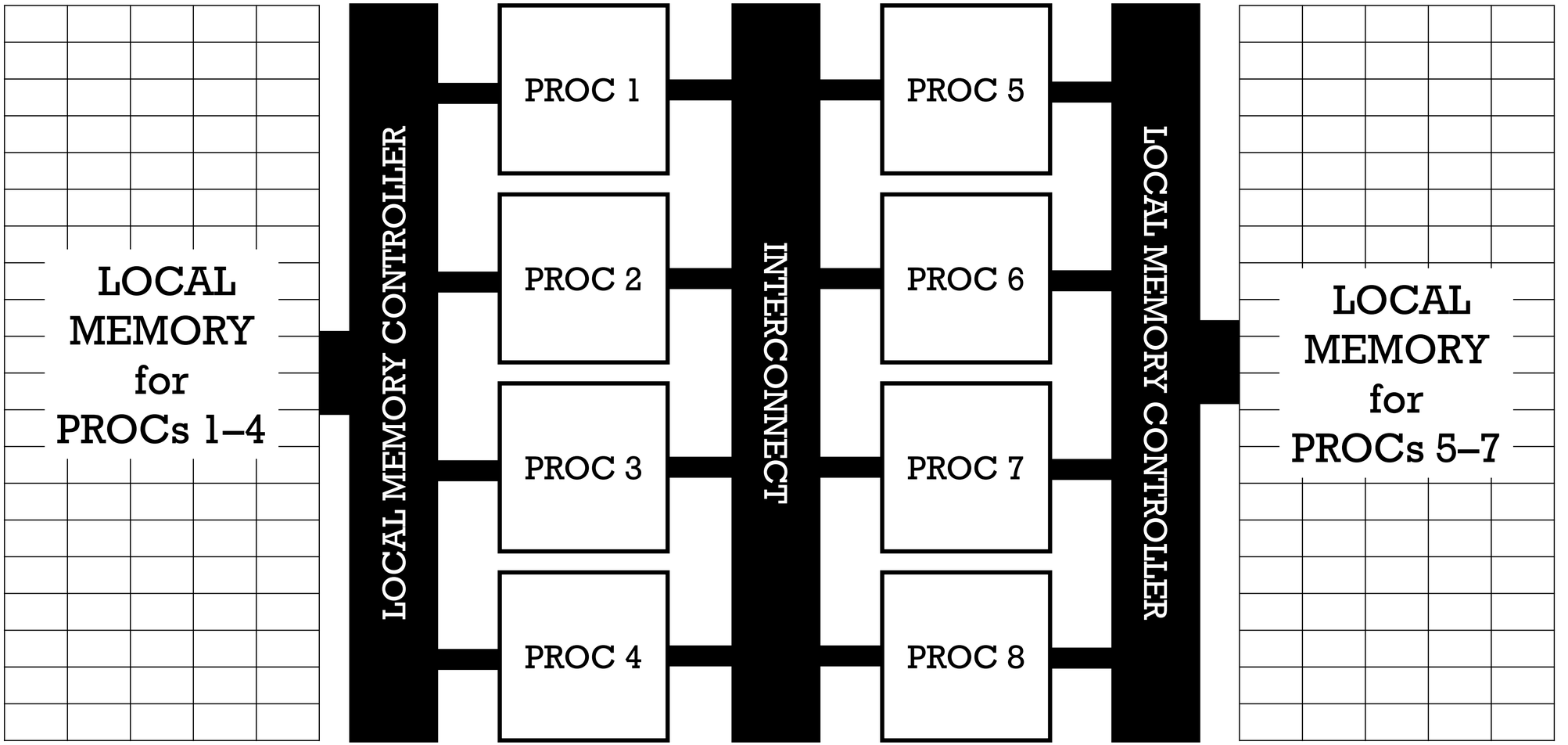
PROS:

- Conceptually Simple
- Direct access to anything

CONS:

- Contention
- Need to manage writes
- Locks etc.
- Programs only work on shared memory machines

Asymmetric Shared Memory



Fast access to local memory

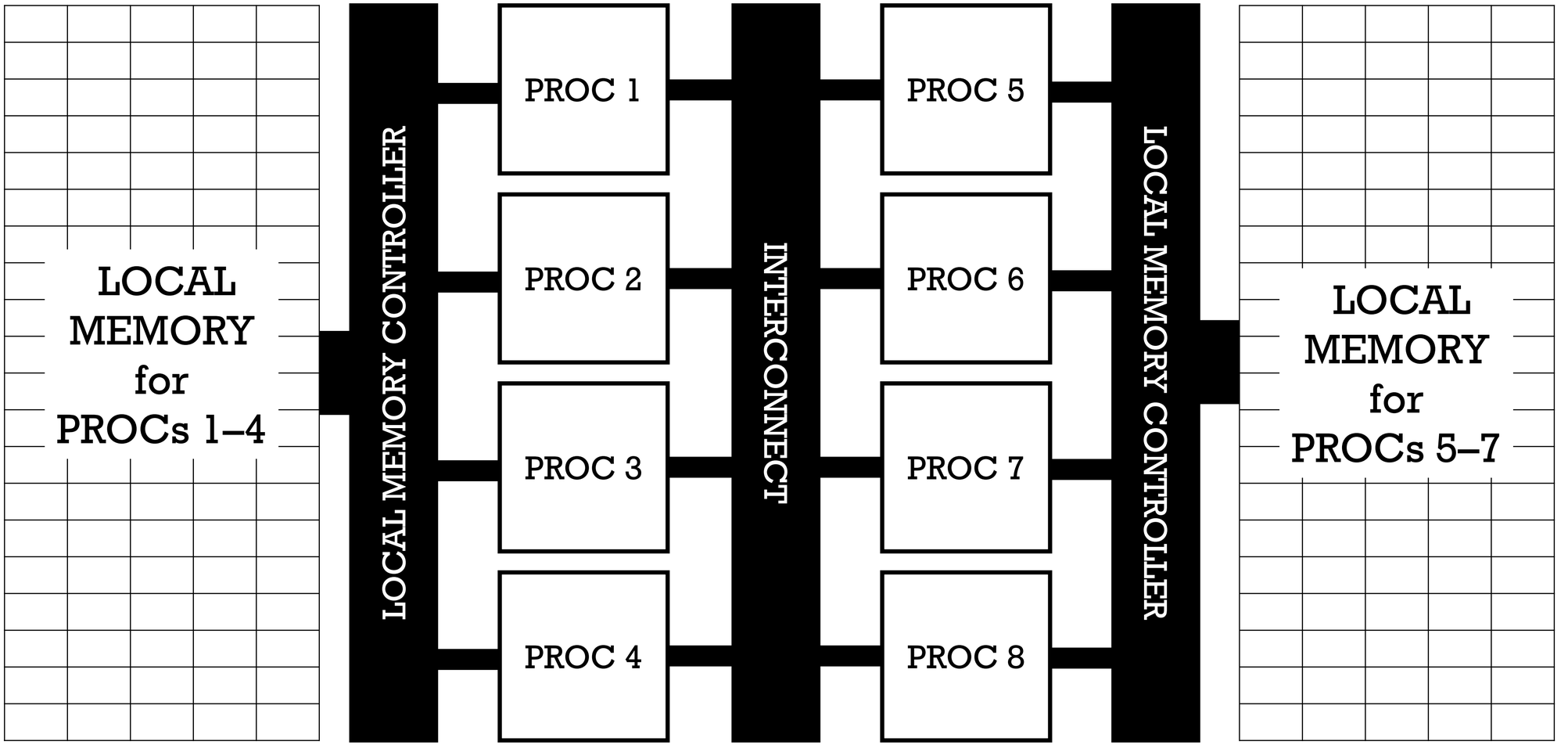
Slower access to non-local memory through interconnect

Diagram shows conceptual (not physical) architecture

"Non-Uniform Memory Architecture" (NUMA)

Stochastic Solutions

Relationship to Clustered/Distributed Computing

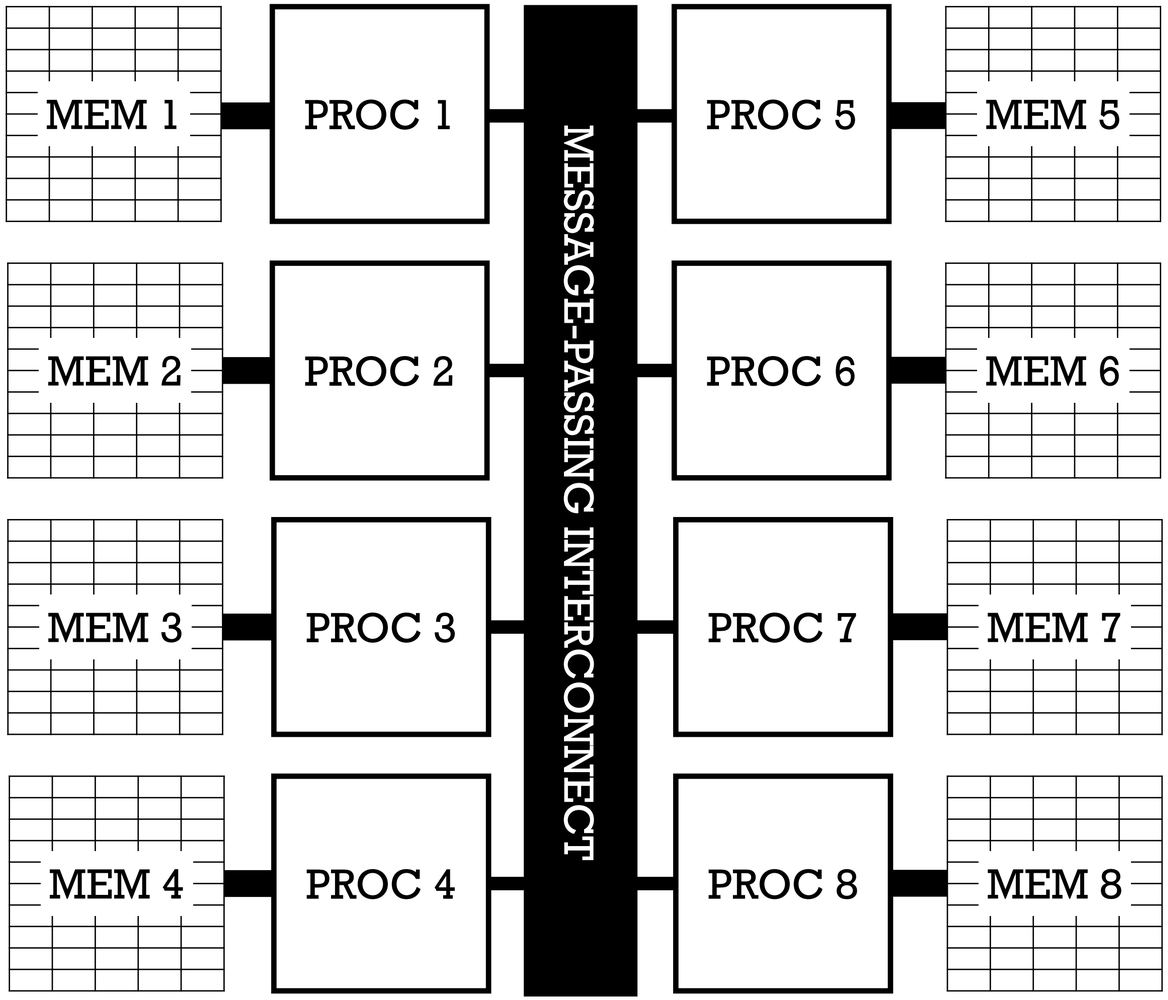


Replace Interconnect with local area network (intranet): Example of Compute Cluster

Replace Interconnect with internet: Example of Distributed Computing

Stochastic Solutions

Distributed Memory ("Shared Nothing")



PROC = PROCESSOR *or* CORE *or* CPU

Each processor has own memory, which is the only memory it can access

Each processor also has access to a messaging system, allowing it to send and receive messages (data) to and from any other processor

PROS:

- No contention
- Can be used on distributed and shared-memory systems
- Very scalable

CONS:

- Arguably conceptually harder than shared memory
- Involves message passing/data copying & latency
- Potential for deadlock etc.

MPI : The Message Passing Interface

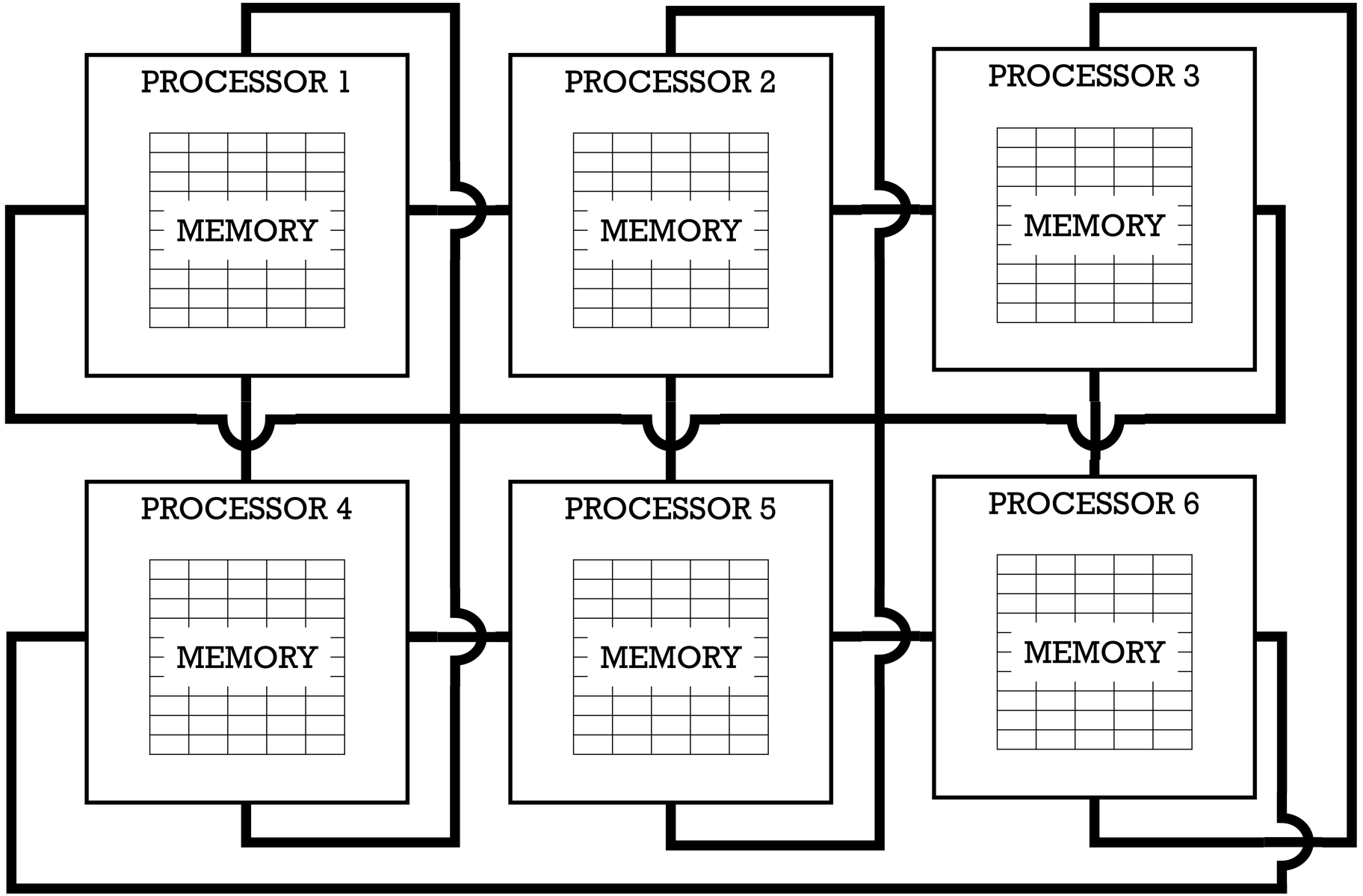
MPICH

- High-performance general-purpose message passing interface
- Extremely mature; bindings in all important languages
- The vast majority of large-scale scientific parallel computing uses MPI (I believe)
- Perfect for distributed-memory parallel computing; also remarkably good for shared-memory systems where message-passing becomes local buffer copying
- Two main implementations: OPEN MPI and MPICH; not much to choose between them

Stochastic Solutions

Historical Interlude

Transputers (historical interlude)



Lego-like systems

Each processor had own memory and 4 communications links

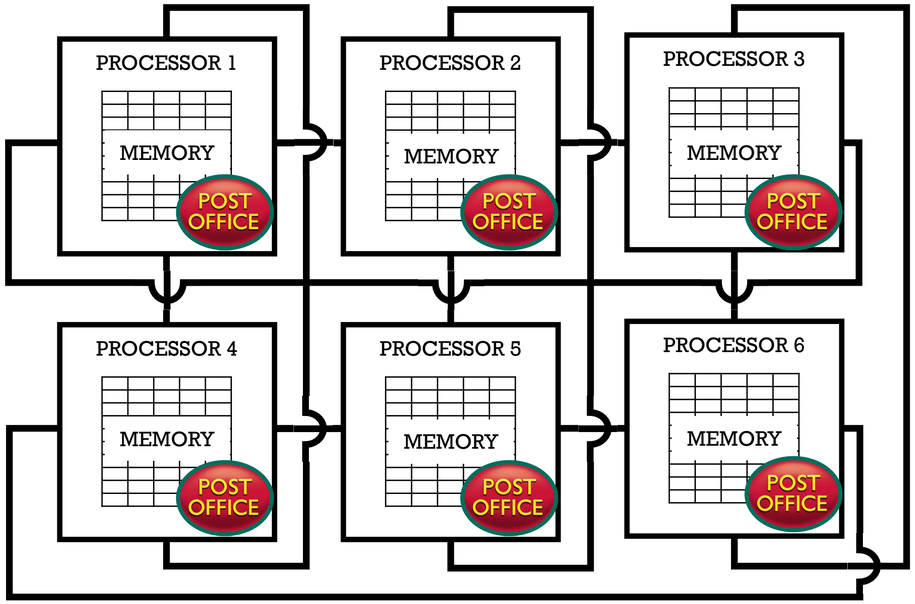
Could only communicate with the processors to which it was connected.

Could choose topology (connections) in software before booting each processor

Here 1 is connected to 2, 3, & 4, but not 5 or 6

Stochastic Solutions

A Partial Edinburgh-Centric History of MPI



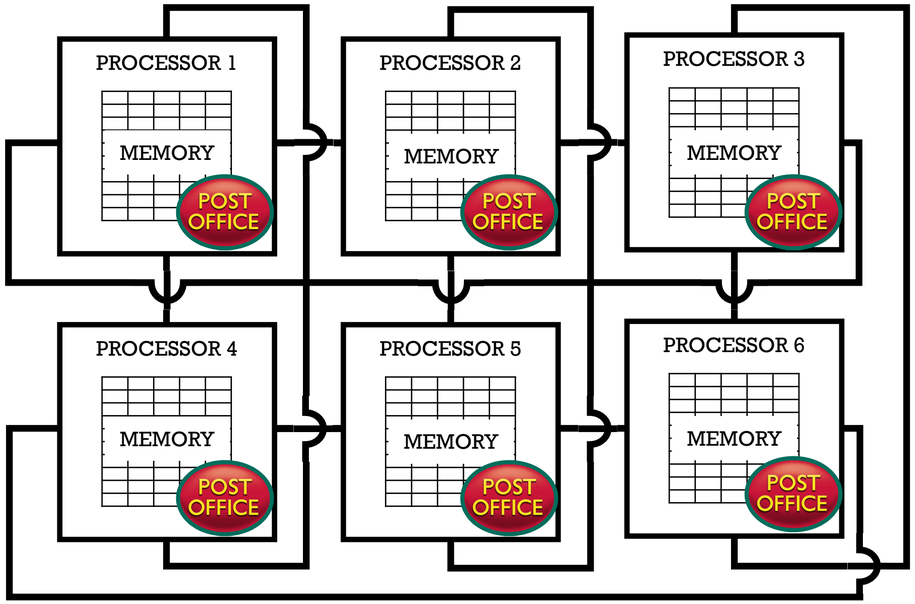
- In the beginning, people chose a topology and wrote code to that topology—often using occam, an inherently parallel language based on Tony Hoare's *Communication Sequential Processes* (CSP) model of parallelism
- c. 1988, Mike Norman, at the Edinburgh Concurrent Supercomputer Project (ECSP) developed TITCH: the Topology-Independent Transputer Communications Harness.
- TITCH was "a post-office on every processor (Transputer)":

- Every processor got an address, and instead of only being able to communicate with your neighbours, you just told TITCH the address you wanted to send to and it delivered it
- TITCH required 7 buffered copies to work successfully

= TITCH

Stochastic Solutions

A Partial Edinburgh-Centric History of MPI

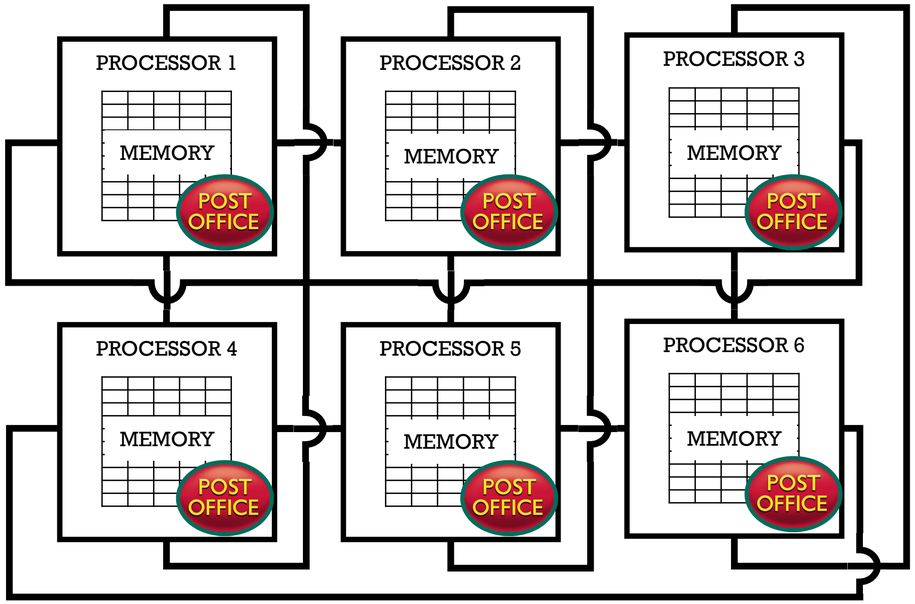


- TITCH begat Tiny. c. 1989, Lyndon Clarke re-implemented TITCH in C
- Tiny was blazingly fast and required far fewer buffers (2?)

= Tiny

Stochastic Solutions

A Partial Edinburgh-Centric History of MPI



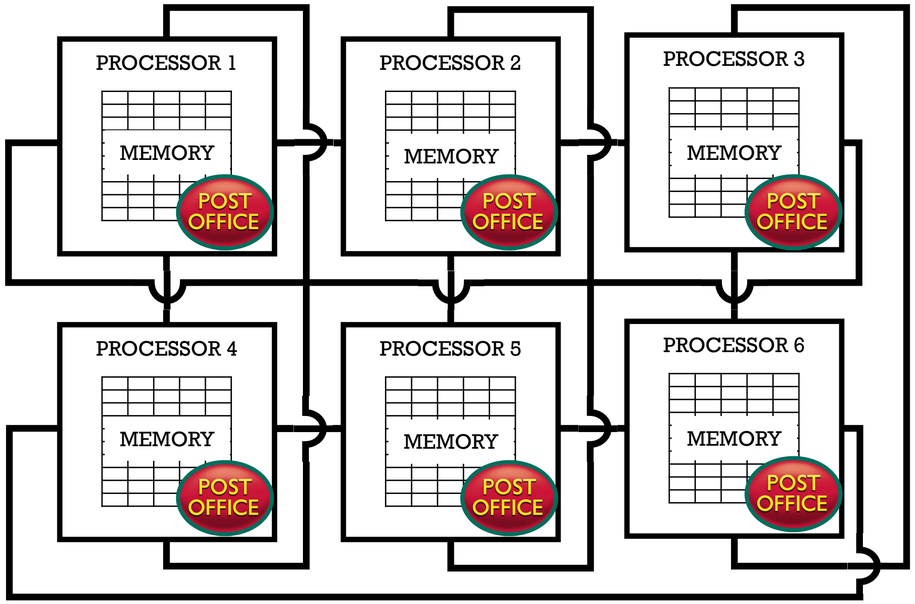
- In 1990, the EPCC was formed—the Edinburgh Parallel Computing Centre
- Tiny begat CHIMP: (Common High-level Interface to Message Passing)

- CHIMP was a lot like Tiny, also written in C, but with bindings for both C and FORTRAN.

POST OFFICE = CHIMP

Stochastic Solutions

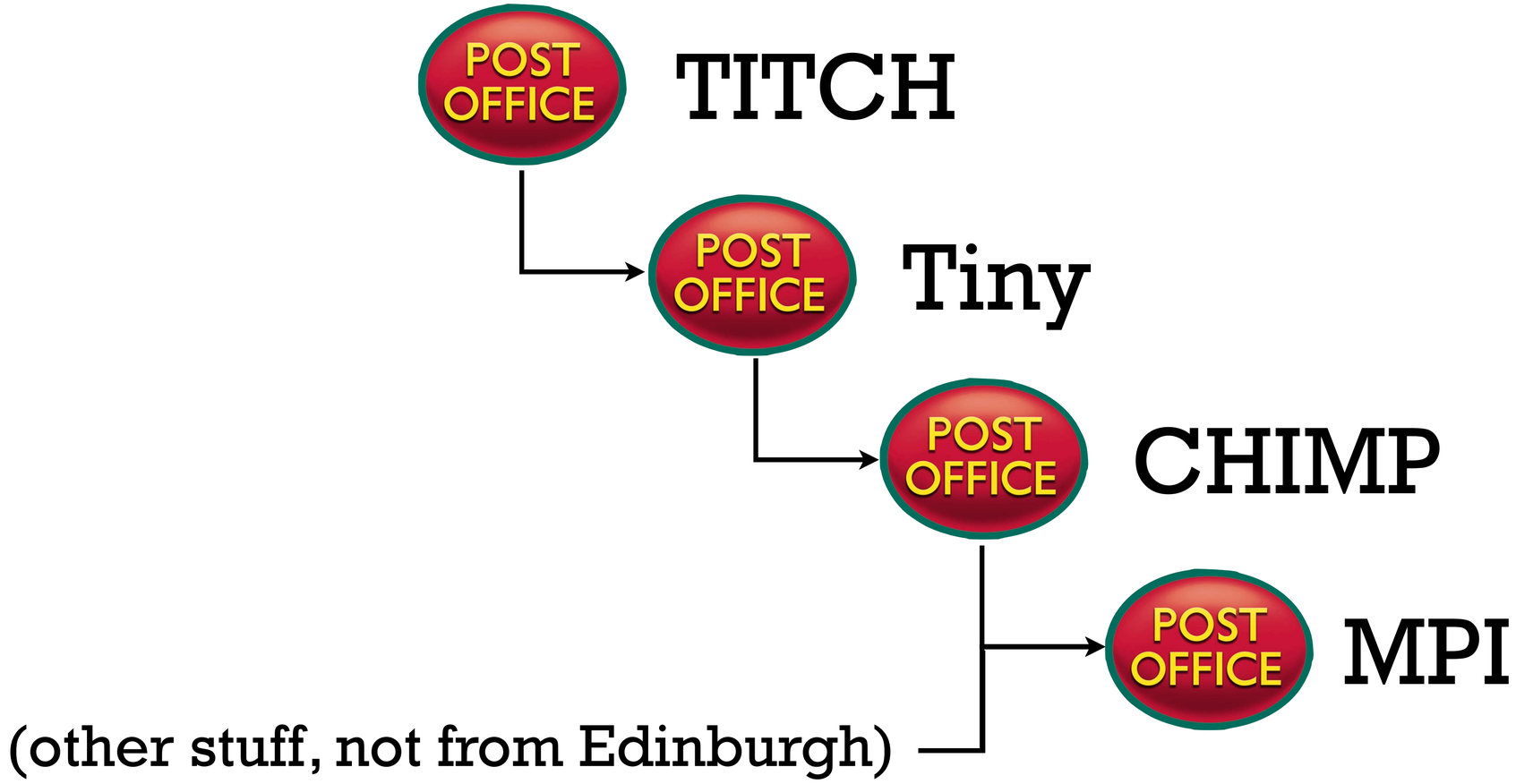
A Partial Edinburgh-Centric History of MPI



- In 1991, CHIMP was one of a number of inputs to MPI.
- A proposal for MPI 1.0 was published by Jack Dongerra (Tennessee), Tony Hey (Southampton) and David Walker (QMC).

POST OFFICE = MPI

A Partial Edinburgh-Centric History of MPI



Stochastic Solutions

mpiexec

- The standard way to run SPMD programs with MPI — that is, the case where same the program will run on each processor is to use the mpiexec command, which is installed with mpi4py, e.g.

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

- For example, I use:

```
#!/bin/sh
OMPI_MCA_btl=self,tcp time mpiexec -n `sysctl -n
hw.physicalcpu` python3 -m mpi4py ptest.py $*
```

Times the run

Allows MPI to exit cleanly on Macs, which it sometimes otherwise fails to do

Retrieves the number of processors on the machine

```
"""
0-send-receive.py

Run with:

   mpiexec -n 2 python 0-send-receive.py

"""
from mpi4py import MPI

comm = MPI.COMM_WORLD                # The communication channel ("Intercom")

rank = comm.Get_rank()               # The process(or) ID

if rank == 0:
    msg = 'Dear 1, Hi! Love 0.'      # (not in scope in other branch)
    comm.send(msg, dest=1)
    print(f'Processor {rank} sent message to processor 1: {repr(msg)}')
elif rank == 1:
    msg = comm.recv(source=0)        # Can omit the source to receive from anyone
    print(f'Processor {rank} received message from 0: {repr(msg)}')
```

```
$ mpiexec -n 2 python 0-send-receive.py
Processor 0 sent message to processor 1: 'Dear 1, Hi! Love 0.'
Processor 1 received message from 0: 'Dear 1, Hi! Love 0.'
$
```

MPI Progams: Message Passing

`Intercom.send` is a simple BLOCKING send command for picklable Python objects

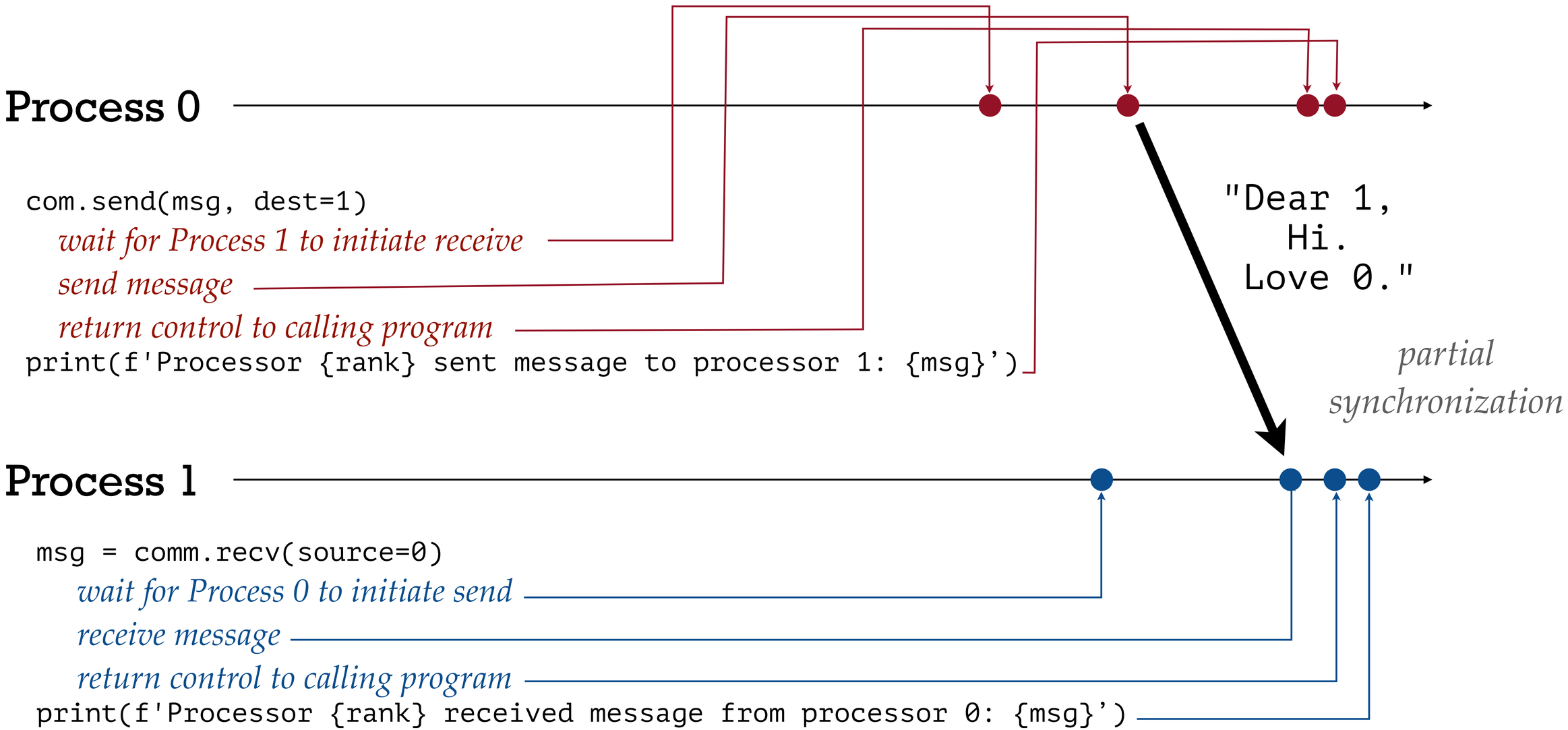
`Intercom.receive` is a simple BLOCKING receive command for picklable Python objects

Every send needs a matching receive

Type and run this
Try removing `source=0` *in* `comm.recv` *and confirm it still works*

Stochastic Solutions

The send/receive Timeline

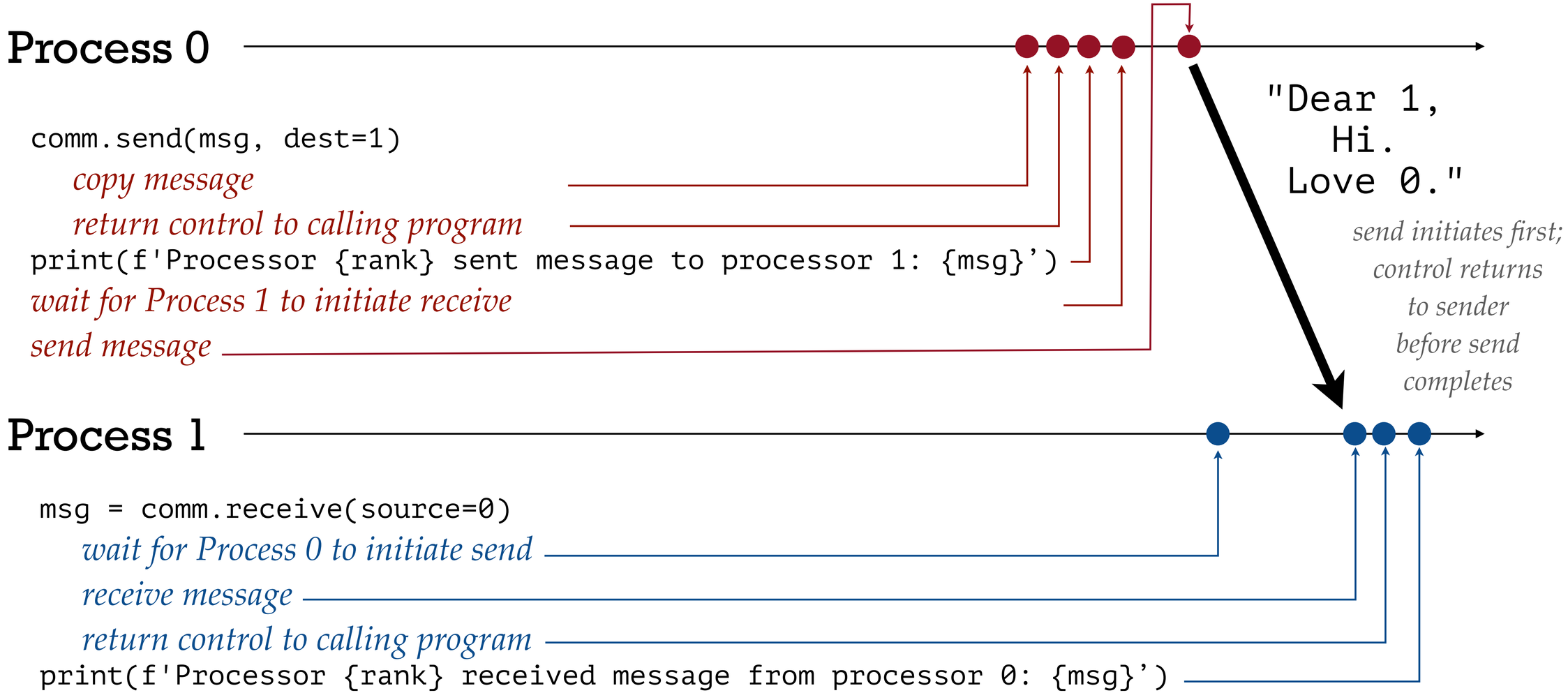


Stochastic Solutions

The send/receive Timeline

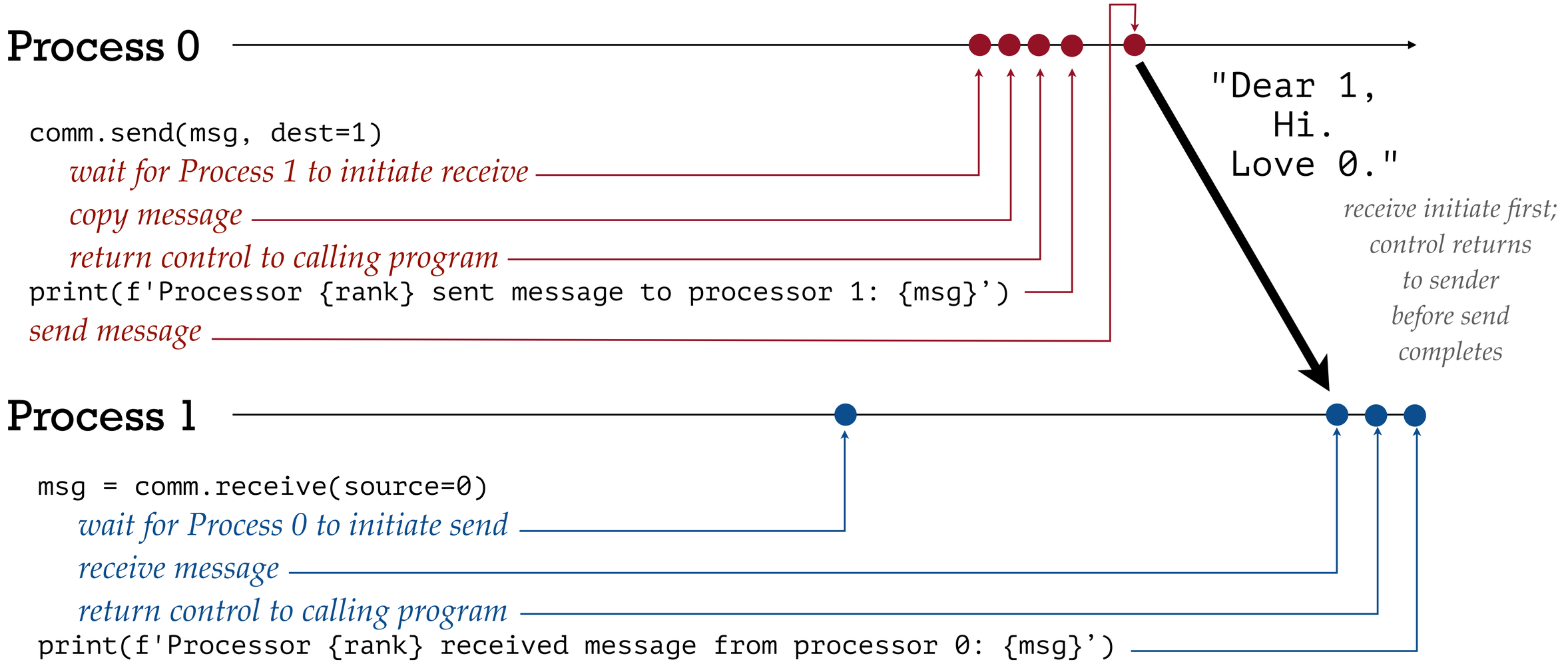
- With blocking sends:
  - The call blocks *until it is safe for you to modify/delete the object you have sent/are sending.*
  - MPI may copy your object and then return control or may wait until it has been sent. YOU DO NOT KNOW.
- With blocking receives:
  - The call blocks until the receive is complete.
  - When your program regains control, it is likely (but not certain) the sending program has also regained control

Another possible send/receive Timeline



Stochastic Solutions

Another possible send/receive Timeline



The send/receive Timeline

- In general, blocking communication introduces idle time, which is detremental to performance, but makes code easier to reason about.
- Blocking also often creates the conditions for deadlock: you will often need some non-blocking communications to avoid this.
- The choice as to whether to block is independent on the two sides.

```
"""
1-deadlock-risk.py

Run with:

   mpiexec -n 2 python 1-deadlock-risk.py

"""
from mpi4py import MPI

comm = MPI.COMM_WORLD             # The communication channel ("Intercom")

me = comm.Get_rank()              # The process(or) ID
other = 1 - me

sent_msg = f'Dear {other}, Hi! Love {me}.'
comm.send(sent_msg, dest=other)
received_msg = comm.recv(source=other)
print(f'Processor {me} sent message to processor {other}: {repr(sent_msg)}')
print(f'Processor {me} received message from {other}: {repr(received_msg)}')
```

DEADLOCK

This program exhibits a very unsafe communication pattern and might deadlock.

Each processor does a blocking send, then a blocking receive.

Copy your program, and modify as above, as `deadlock-risk.py`.
Then run it (as long as you know how to kill processes on your system.)

Stochastic Solutions

Deadlock

- In fact, there is every chance this program won't actually deadlock
- This is because our message is small (19 characters) and most MPI implementations, by default, copy small messages when using `Intercom.send`.

So while you (might) expect

PROCESS 0
comm.send(msg, dest=1)
wait until Process 1 receives
PROCESS 1
comm.send(msg, dest=1)
wait until Process 0 receives

} *Deadlock!*

What often actually happens is:

PROCESS 0
comm.send(msg1, dest=1)
MPI copies msg1
msg2 = recv(source=1)
MPI does the send & receive: works

PROCESS 1
comm.send(msg1, dest=0)
MPI copies msg1
msg2 = recv(source=0)
MPI does the send & receive: works

No Deadlock!

```
"""
2-deadlock.py

Run with:

   mpiexec -n 2 python 2-deadlock.py

"""
from mpi4py import MPI

comm = MPI.COMM_WORLD             # The communication channel ("Intercom")

me = comm.Get_rank()              # The process(or) ID
other = 1 - me

sent_msg = f'Dear {other}, Hi! Love {me}.'
L = len(sent_msg)

for n in range(1, 16):
    comm.send(sent_msg * (2 ** n), dest=other)
    received_msg = comm.recv(source=other)
    print(f'Processor {me} sent message of length {len(sent_msg)} '
          f'to processor {other}, starting starting{repr(sent_msg)}')
    print(f'Processor {me} received message of length {len(received_msg)} '
          f'from {other}, starting: {repr(received_msg[:L])}')
```

Forced Deadlock:

Keep doubling the size until it deadlocks

Copy `1-deadlock-risk.py`*, and modify as above, as* `2-deadlock.py`*.*
Then run it (as long as you know how to kill processes on your system.)

```
$ mpiexec -n 2 python 2-deadlock.py
Processor 0 sent message of length 19 to processor 1, starting starting'Dear 1, Hi! Love 0.'
Processor 0 received message of length 38 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 0 sent message of length 19 to processor 1, starting starting'Dear 1, Hi! Love 0.'
Processor 0 received message of length 76 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 0 sent message of length 19 to processor 1, starting starting'Dear 1, Hi! Love 0.'
Processor 0 received message of length 152 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
Processor 1 received message of length 38 from 0, starting: 'Dear 1, Hi! Love 0.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
Processor 1 received message of length 76 from 0, starting: 'Dear 1, Hi! Love 0.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
Processor 1 received message of length 152 from 0, starting: 'Dear 1, Hi! Love 0.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
Processor 1 received message of length 304 from 0, starting: 'Dear 1, Hi! Love 0.'
Processor 0 sent message of length 19 to processor 1, starting starting'Dear 1, Hi! Love 0.'
Processor 0 received message of length 304 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 0 sent message of length 19 to processor 1, starting starting'Dear 1, Hi! Love 0.'
Processor 0 received message of length 608 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 0 sent message of length 19 to processor 1, starting starting'Dear 1, Hi! Love 0.'
Processor 0 received message of length 1216 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 0 sent message of length 19 to processor 1, starting starting'Dear 1, Hi! Love 0.'
Processor 0 received message of length 2432 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
Processor 1 received message of length 608 from 0, starting: 'Dear 1, Hi! Love 0.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
Processor 1 received message of length 1216 from 0, starting: 'Dear 1, Hi! Love 0.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
Processor 1 received message of length 2432 from 0, starting: 'Dear 1, Hi! Love 0.'
```

Forced Deadlock:

The last successful receive on each processor was a string of length 2432; then there was deadlock

It is possible (likely?) that this MPI copies objects under 4K, not over.

It might be different on your machine

Stochastic Solutions

Non-Blocking Communications

- `intercom.send` and `intercom.recv` have non-blocking (`i`) counterparts that can return control, with a *request handle*, to the calling code
  - before objects sent with `isend` can safely be altered, and,
  - before messages requested with `irecv` have actually been received
- After issuing one or more non-blocking sends or receives, a correct MPI program must `wait` on the request handle or handles.

NON-BLOCKING SEND

```
req = comm.isend(data, dest=1)
#  do other stuff
#  don't overwrite data!
req.wait()
#  now safe to overwrite  data
```

NON-BLOCKING RECEIVE

```
req = comm.irecv()
#  do other stuff
data = req.wait()
#  data has been received and can be used
```

Blocking & Non-Blocking Communications Can Be Mixed

A *non-blocking* receive (`irecv`) can receive a message from a *blocking* send (`send`).

A *blocking* receive (`recv`) can receive a message from a *non-blocking* send (`isend`).

COMPATIBILITY	`recv`	`irecv`
`send`	✔	✔
`isend`	✔	✔

NOTE (or be sad)

The implementation of `Intercom.irecv` in `mpi4py` requires a buffer to be passed if the message is "large" (≳ 32K (kilobytes))

https://bitbucket.org/mpi4py/mpi4py/issues/65/mpi_err_truncate-message-truncated-when

Comments (7)

Author/maintainer of mpi4py

Lisandro Dalcin

The implementation of `irecv()` for large messages requires users to pass a buffer-like object large enough to receive the pickled stream. This is not documented (as most of mpi4py), and even non-obvious and unpythonic, but if you have some good knowledge about MPI you will will understands its limitations for implementing `irecv()` for pickled streams.

NON-BLOCKING RECEIVE WITH BUFFER

```
buf = bytearray(1 << 20)  # 1MB receive buffer
req = comm.irecv(buf, source=1)
#  do other stuff; data not received
data = req.wait()
#  data has been received
```


Stochastic Solutions

BUFFERS & PICKLING

- You can use `send`, `isend`, `recv`, and `irecv`, with any picklable Python object
- pickling is a Python implementation of marshalling or serialisation: the process of turning a structured object into a flat stream of data (bytes, in Python) that can be sent.
- json.dumps is a simple example of marshalling: a structured OBJECT is turned into a string, e.g.
- self-referential objects may not be picklable
- only the buffer size matters: it can always be a bytearray, because the pickled data is always a byte stream

```
{
    'a': 1,
    'b': 2
}
```



```
'{"a":1,"b":2}'
```

Stochastic Solutions

```
"""
3-no-deadlock.py

Run with:

   mpiexec -n 2 python 3-deadlock.py

"""
from mpi4py import MPI

comm = MPI.COMM_WORLD             # The communication channel ("Intercom")
BUF_SIZE = 1 << 20                # 1MB receive buffer

me = comm.Get_rank()              # The process(or) ID
other = 1 - me

sent_msg = f'Dear {other}, Hi! Love {me}.'
L = len(sent_msg)

buf = bytearray(BUF_SIZE)
for n in range(1, 16):
    rreq = comm.irecv(buf, source=other)  # re-used buffer
    sreq = comm.isend(sent_msg * (2 ** n), dest=other)
    received_msg = rreq.wait()
    print(f'Processor {me} sent message of length {len(sent_msg)} '
          f'to processor {other}, starting starting{repr(sent_msg)}')
    print(f'Processor {me} received message of length {len(received_msg)} '
          f'from {other}, starting: {repr(received_msg[:L])}')
    sreq.wait()
```

Deadlock-Free Guaranteed™ Implementation

Copy `2-deadlock.py` *to* `3-no-deadlock.py` *and modify as above.*

Then run it.

Note that we wait on both the receive (to get get the data) and the send (to avoid memory leaks & worse).

```
$ mpiexec -n 2 python 3-no-deadlock.py > out
$ head -5 out
Processor 0 sent message of length 19 to processor 1, starting starting'Dear 1, Hi! Love 0.'
Processor 0 received message of length 38 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
Processor 1 received message of length 38 from 0, starting: 'Dear 1, Hi! Love 0.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
$ tail -5 out
Processor 0 received message of length 311296 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 0 sent message of length 19 to processor 1, starting starting'Dear 1, Hi! Love 0.'
Processor 0 received message of length 622592 from 1, starting: 'Dear 0, Hi! Love 1.'
Processor 1 sent message of length 19 to processor 0, starting starting'Dear 0, Hi! Love 1.'
Processor 1 received message of length 622592 from 0, starting: 'Dear 1, Hi! Love 0.'
$
```

All runs fine as written.

If I don't pass the buffer:

```
mpi4py.MPI.Exception: MPI_ERR_TRUNCATE: message truncated
```

Buffers & numpy Arrays

- Python is slow*
- But many of its libraries, especially `numpy`, are pretty fast.
- It makes little sense to parallelise slow code, so optimise first
- MPI has first-class support for raw buffers, which include `numpy` arrays.

Intercom Method	**Python Object**	**Buffer**
blocking send	`send`	`Send`
non-blocking send	`isend`	`Isend`
blocking receive	`recv`	`Recv`
non-blocking receive	`irecv`	`Irecv`

* Please don't tell me all the senses in which it isn't slow.

Buffers & numpy Arrays

- It is normally recommended, when sending buffers, that the user specify the type of the values in the buffer as an MPI type, and usually the number of items in the buffer as well.
- mpi4py does include some automatic type inference for numpy arrays, but this is limited to C-types (*"all C/C99-native signed/unsigned integral types and single/double precision real/complex floating types"*)
- Examples available at:

  https://github.com/stochasticsolutions/mpi4py-examples

```
git clone https://github.com/stochasticsolutions/mpi4py-examples.git
```

Data Parallelism vs. Functional Parallelism

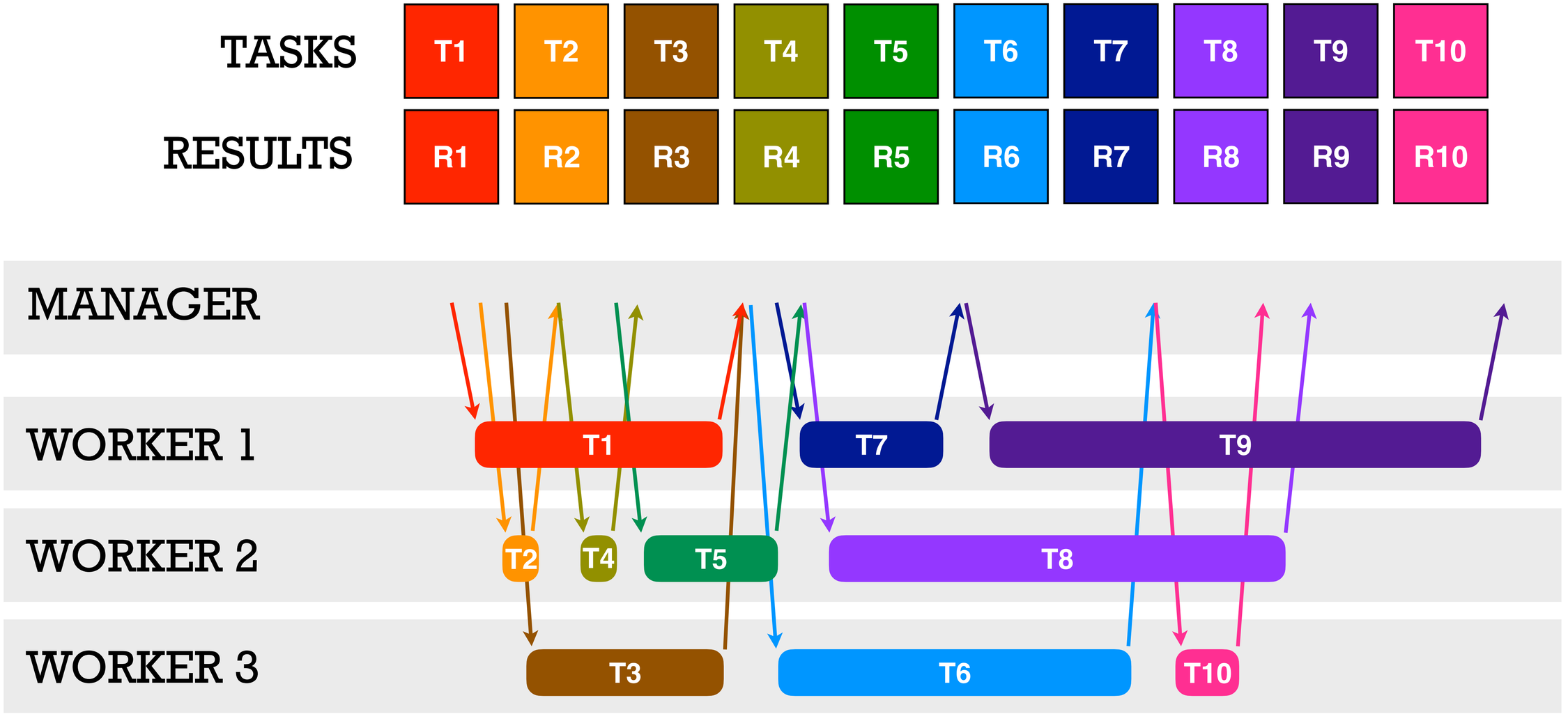
- With functional parallelism, different processors (processes) perform different function. It is not uncommon to have some functional decomposition of a problem in parallel computing (e.g. a manager process and a set of workers), but it can be hard to scale if you have different functions for different processors.
- Data parallelism is the dominant pattern: here each processor (or at least, most processors) are allocated data for different parts of a problem, but all carry out similar (or identical) processing on the parts.
- With SPMD (single program, multiple data), usually either all processors run the same code, or a simple switch at the top chooses between a small number of roles for different processors

Task Farm

Task Farm

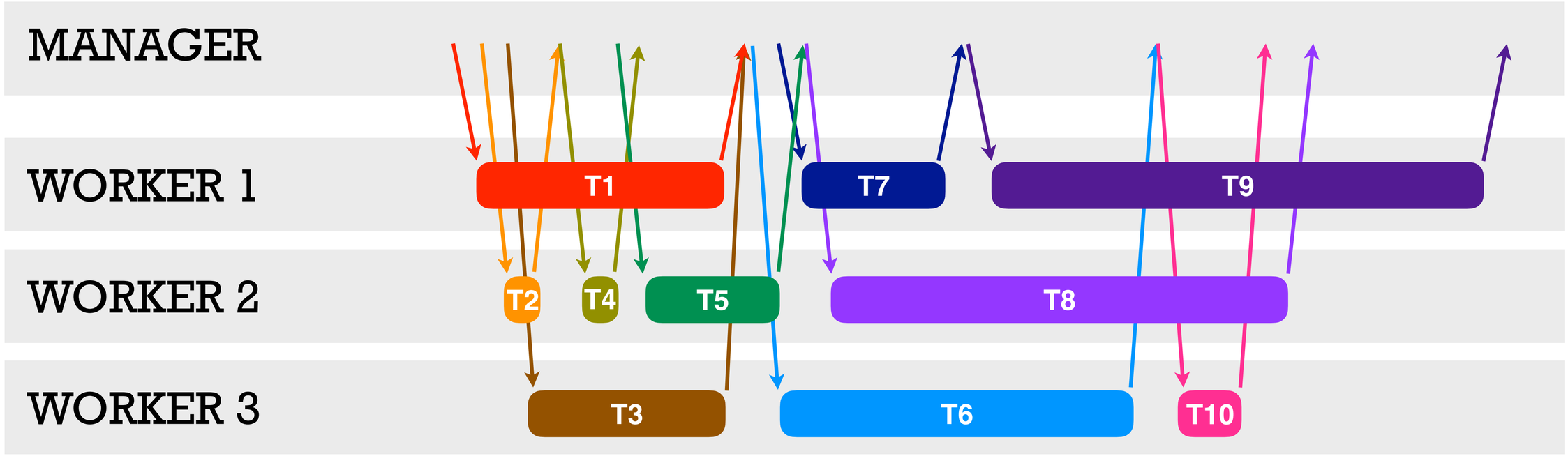
- Simplest general-purpose parallel paradigm
- One process(or) has (is given) a collection of tasks to be performed
  - Controller, Manager, Main, "Master", Distributor, Queen, …
- All other processes act as workers ("slaves", …)
- Manager allocates tasks to each worker then collects results (if any) and allocates further tasks ask workers become free, until all tasks have been performed
- Usually all tasks are independent, each worker need only communicate with the manager.

Task Farm



Stochastic Solutions

Task Farm



- As can be seen here, having a large task allocated late means that worker will tend to finish significantly after the others.
- If tasks sizes are known, and the task list is fixed, it's generally better to allocate larger tasks first, and later tasks later.

```
if __name__ == '__main__':

    def cube(n):
        return n * n * n

    def report(tasks, results):
        for task_id, task in sorted(results.items()):
            (n,), _ = tasks[task_id][:2]
            print(f'{n}^3 = {results[task_id]}')

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

    if rank == 0:  # Task Farm Manager; always ID 0
        tasks = [((i,), {}) for i in range(100)]
        tf = TaskFarm(comm, tasks)
        results = tf.go()
        tf.stop_workers()
        report(tasks, results)

    else:  # Task Worker
        f = cube
        worker(comm, rank, f, verbose=VERBOSE)
```

Task Farm: Example

Use for a trivial task:
cubing first hundred numbers

Each task is a pair 2-tuple consisting of the positional and keywords args to the function the worker will execute. We will use the `cube` function, which takes a single integer argument. So our list of tasks is:

```
[((0,), {}),
 ((1,), {}),
        ⋮
 ((99,), {}))]
```

`cube(*(n, ), **{})` ⟼ `cube(n)`

Note that after initialising the Task Farm in the manager process, we must `go()`; and when finished we call `tf.stop_workers()` to shut them down.

```python
def worker(comm, worker_id, f, verbose=VERBOSE):
    """
    Worker process for taskfarm. (Run only on the workers.)
    Processes jobs using the function f until it receives STOP
    as the task_id.

    Args:
        comm: the MPI comm object
        worker_id: this worker's ID (rank, as returned by comm.Get_rank())
        f: the function to use to process jobs

    Returns:
        None (after all work is done!)
    """
    msg = comm.recv(source=MANAGER_ID)  # blocking

    (task_id, args, kw) = msg
    while task_id != STOP:
        result = f(*args, **kw)
        comm.send((worker_id, task_id, result), dest=MANAGER_ID)
        (task_id, args, kw) = comm.recv(source=MANAGER_ID)
    if verbose:
        print(f'Worker {worker_id} received STOP; stopping.')
```

Task Farm Worker

The Worker is given a communicator, an ID, and a function to use to perform tasks.

It first does a blocking receive, waiting for a task from the manager.

The task will be a 3-tuple positional arguments (args) and the keyword args to pass to the function to perform the task.

If the unpacked `task_id` is `STOP`, this means there is no more work, and the `Worker` stops. Otherwise, it repeatedly does the task by calling `f` with the `args` & `kw`, then returns the `result` to the manager, implicitly requesting a new task, which may be `STOP`, which it receives.

Task Farm

Basic initialisation

```
class TaskFarm:
    """
    MPI-based Task Farm, Manager Process (run only on PID 0)

    Args:
        comm: the MPI comm object
        tasks: the list of tasks. Each task is a tuple, (f, args, kwargs)
               where f is a function to perform the task,
               args is the list of positional arguments for the function
               and kwargs is the list of keyword arguments for the function
        verbose: Optional boolean to control verbosity
                 Defaults to VERBOSE.
    """
    def __init__(self, comm, tasks, verbose=VERBOSE):
        self.comm = comm
        self.tasks = tasks or []
        self.verbose = verbose

        self.n_procs = comm.Get_size()
        self.n_results_outstanding = 0
        self.n_tasks = len(self.tasks)
        self.requests = [None] * self.n_procs  # holds request handles
        self.results = {}   # holds results, keyed on task_id
```

When we send a task (non-blocking) to a worker, we store the handle in `self.requests` in position `worker_id`. Since workers are numbered from `1`, `self.requests[0]` is unused. This allows us to wait on the request later to tidy up.

```
    def go(self):
        """
        Run the task farm with the tasks already specified.

        Return:
            Results, keyed on task_id, which is the index of the
            task in the task list.
        """
        # Send each worker an initial task, assuming there are enough
        n_initial = self.send_initial_tasks()

        # Collect each result and send a replacement task until all sent
        for task_id in range(n_initial, self.n_tasks):
            worker = self.collect_result()
            self.allot(task_id, worker)

        # Collect remaining results
        while self.n_results_outstanding > 0:
            worker = self.collect_result()

        return self.results
```

Task Farm: Go

We send each worker a task, if there are enough. (Otherwise we just send out all tasks)

Then we run the main loop, collecting a result, storing it, and sending the next, until we run out of tasks.

```
def send_initial_tasks(self):
    """
    Send initial task to each worker

    Args:
        report (bool): If set to True, this will report the
                       initial allocation of tasks

    Returns:
        Number of tasks allocated (int)
    """
    n_workers = self.n_procs - 1
    n_initial = min(n_workers, self.n_tasks)
    for task_id in range(0, n_initial):
        self.allot(task_id, worker=task_id + 1)

    if self.verbose:
        print(f'{n_initial} tasks allocated.')

    return n_initial
```

Task Farm: Allot Initial Tasks

If there are fewer tasks than workers, we send them all out; otherwise we send one to each worker.

We return the number we allotted.

```
def allot(self, task_id, worker):
    """
    Allot task n to the worker specified.
      - Picks the task with task_id
      - sends to a the nominated worker
      - Stores the request, keyed on worker, so it can be
        called after receipt.

    Args:
        task_id: the number of the task to be allocated
        worker: the ID of the task to be allocated
    """
    args, kw = self.tasks[task_id]
    task = (task_id, args, kw)
    req = self.comm.isend(task, dest=worker)  # non-blocking
    if self.verbose:
        print(f'Allotted task {task_id} to worker {worker}.')
    self.n_results_outstanding += 1
    self.requests[worker] = req  # store request handle for later await
```

Task Farm: Allot a Task

A task is just a 3-tuple of the `task_id` and the function arguments (positional, `args`, and keyword `kw`).

We send the task without blocking, and save the request handle in `self.requests`.

We keep track of the number of outstanding results for the later phase when all tasks have been allotted.

```
def collect_result(self):
    """
    Collects the next result from any worker then:
      - Records the result in self.results (keyed on task_id)
      - Waits on the initial request (which should be complete)

    Args:
        None

    Returns:
        - The ID of the worker that returned a result.
    """
    (worker, task_id, result) = self.comm.recv()  # receive
    self.results[task_id] = result
    if self.verbose:
        print(f'Received result of task {task_id} from worker {worker}.')

    req = self.requests[worker]
    self.n_results_outstanding -= 1
    self.requests[worker] = None
    if req:
        req.wait()  # must be done now: the result has been returned
    return worker
```

Task Farm: Collect Results

The Worker process returns its ID, the task ID and the result..

We store the result in self.results, get the handle from the request we used to send the task and wait on it.

We keep track of the number of requests outstanding and clear this workers entry in self.requests as we have now waited on it.

We return the worker ID so the caller knows who to allocate a task, or to STOP, next.

```
    def stop_workers(self):
        """
        Send stop task for all workers
        """
        # Wait for all the workers to have stopped
        for worker_id in range(1, self.n_procs):
            self.stop_worker(worker_id)

    def stop_worker(self, worker_id):
        """
        Send stop task to nominated worker
        """
        self.comm.send((STOP, (), {}), worker_id)
```

Task Farm: Stopping Workers

We stop a Worker simply by sending it a task with `task_id` set to `STOP`.

The other two tuple members aren't used, but we choose and empty tuple and empty dictionary match the normal `args` and `kw` parameters.

Task Farm: Header

```
"""
task farm

Run example with:

   mpiexec -n 4 python taskfarm.py

replacing 4 with however many processors you have/want to use
"""
from mpi4py import MPI

MANAGER_ID = 0  # Manager is always worker (rank) zero under mpiexec
VERBOSE = False
STOP = -1
```

Run the Task Farm


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

Change setting of `VERBOSE=False` at the top to `True`. Run again. Scroll back up to see the Task Farm describing what it's doing.

In your own time (i.e. later):

1. Extract the code under `if __name__ == '__main__':` to separate file (with necessary imports etc.) and run again
2. Write a few functions (or better, plug in functions you already have) and try using the Task Farm for real, timing and measuring speedup
3. Experiment with re-ordering tasks from largest to smallest and see how much difference it makes
4. Use it for real!

Indicative Timings from Miró Tests

SERIAL
No Task Farm

```
Ran 1979 tests in 186.659s
OK (skipped=17)
Number of sessions created: 998
All 1,979 tests (23,052 assertions) ran OK.

real    3m22.979s
```

3m23s = 203 seconds
(1x)

PARALLEL
Weighted
Task Farm,
8 processors,
7 workers

```
Total test time: 181.71 seconds.

Job completed after a total of 26.8064 seconds.
       40.59 real       229.49 user        34.74 sys

real    0m40.632s
```

41 seconds
203 / 41 = 5.1x

PARALLEL
Weighted
Task Farm,
4 processors,
3 workers

```
Total test time: 147.06 seconds.

Job completed after a total of 50.4334 seconds.
       64.78 real       188.02 user        42.01 sys

real    1m4.812s
```

1m5s = 65 seconds
203/65 = 3.1x

Mac Studio, 20 procs. c.10 secs, 1 proc c. 90 secs

Amdahl's Law & Maximum Speedups

Nine women can't make a baby in one month

— Fred Brooks, The Mythical Man Month (book)

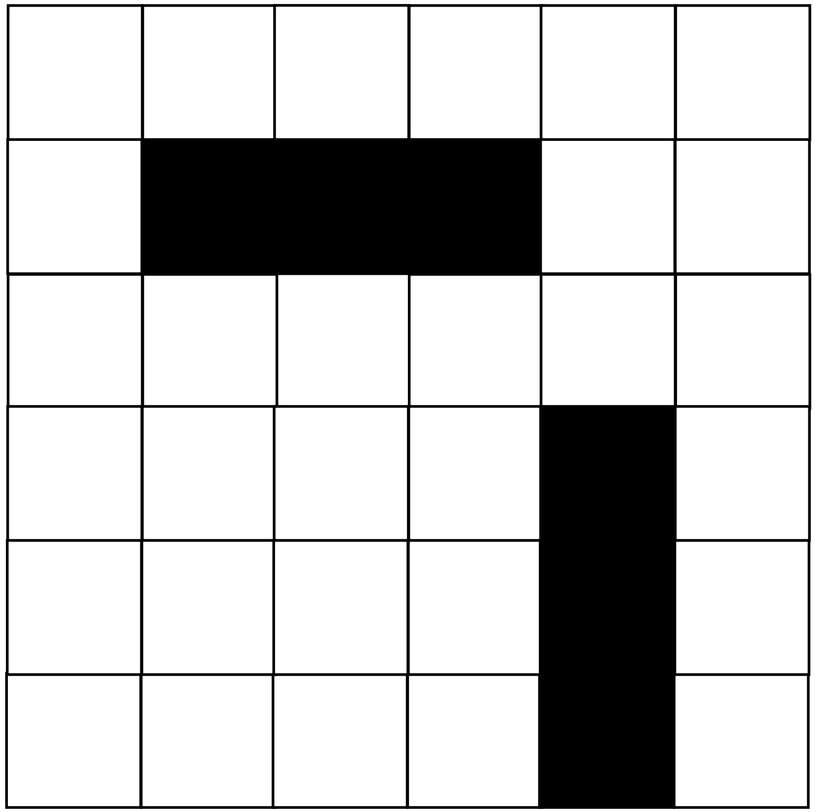
Amdahl's Law says the maximum possible speed-up is limited by the longest sequential task. If you have a 1-minute task, whatever happens in parallel, it's going to take at least 1 minute to complete the computation.

Grid Decomposition

Grid Decomposition

- There is some grid or mesh of data over which calculations take place—the atmosphere, the ocean, a metal bar, an aircraft wing, an oil well, a cellular automaton or whatever—in some number of dimensions,, usually over a number of time steps. (This is almost all physical simulations.)
- The computation is local—computing the value at any point on the grid/mesh at each point in time only requires data from that point in the grid/mesh and a few neighbours
- With grid decomposition, each processor is allocated a portion of the grid, which is decomposed in (usually) some of the dimensions
- At each step, the "interior" of each processors grid/mesh can be computed with only local data, but messages must be passed between neighbours to get information about their state before update.

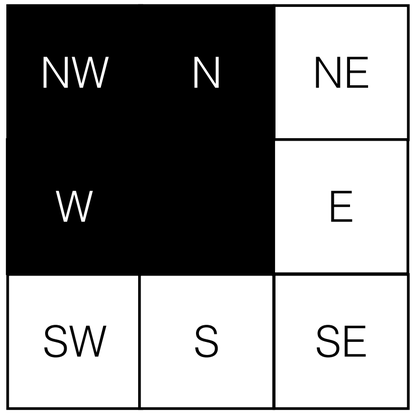
e.g. Game of Life (a cellular automaton)



Grid of cells that can be alive (black/full) or dead (white/empty)

At each timestep T, use a (local) update rule to determine whether the cell will be alive or dead at the next timestep T+1

e.g. Game of Life (a cellular automaton)



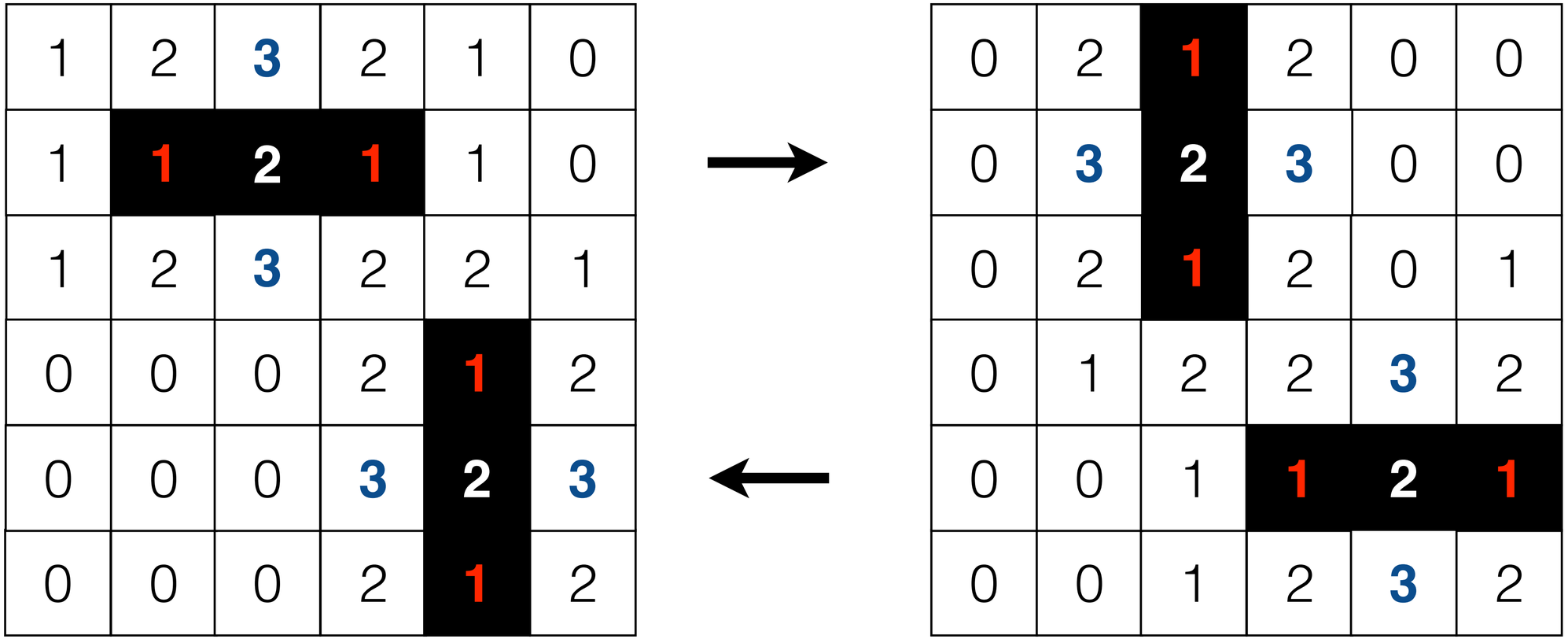
Update Rule:
Count number n of live neighbours (0–8):

N + W + S + E + NE + NW + SW + SE = 3 here

If cell is alive (black),
remain alive if n is 2 or 3;
otherwise die

If cell is not alive (white/empty),
bring it to life only if n is 3

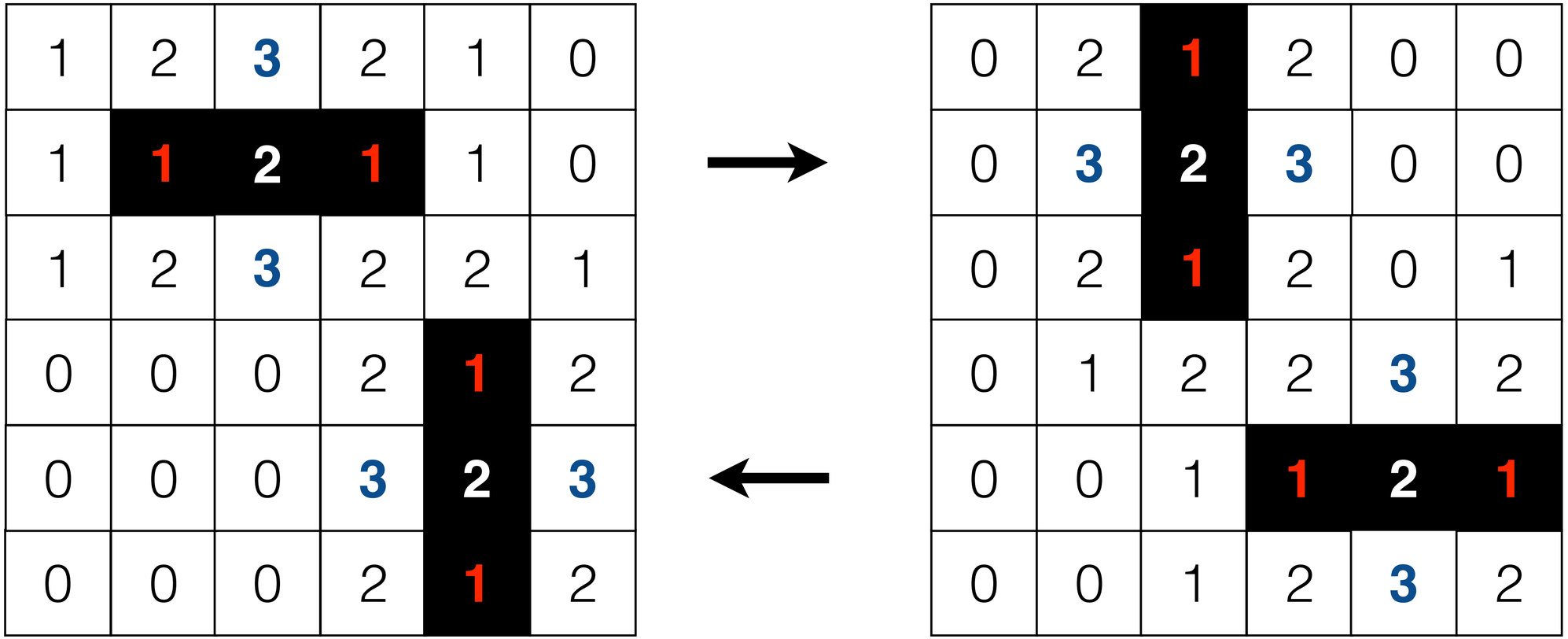
e.g. Game of Life (a cellular automaton)



Here have assumed edge of grid is empty:
you can wrap it around for a Toroidal world

Stochastic Solutions

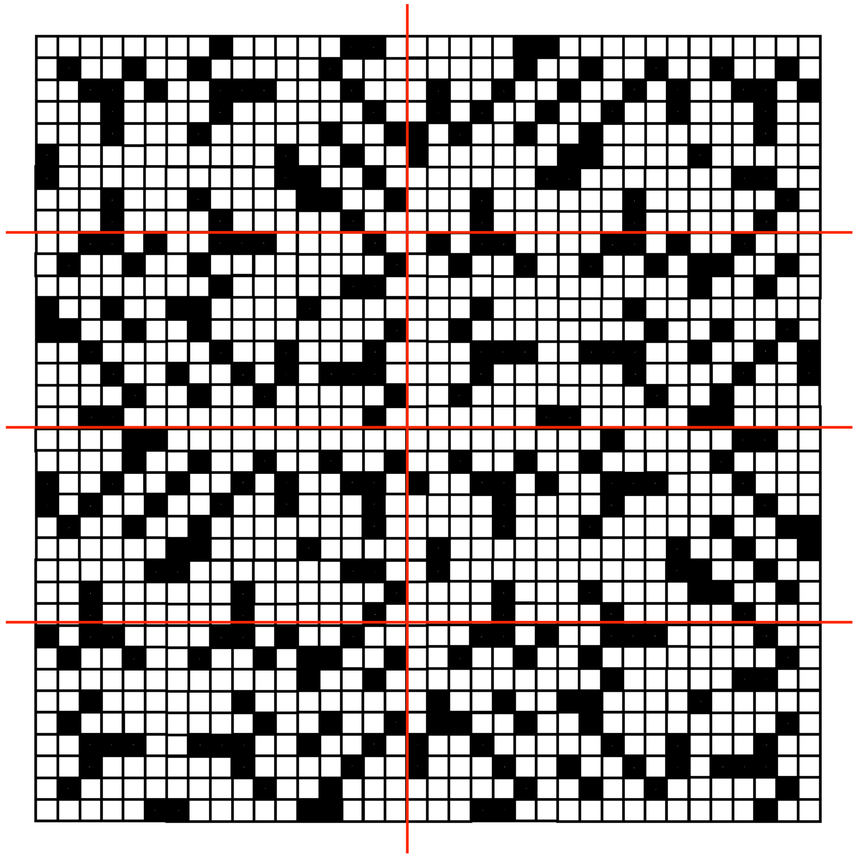
e.g. Game of Life (a cellular automaton)



Here have assumed edge of grid is empty:
you can wrap it around for a Toroidal world

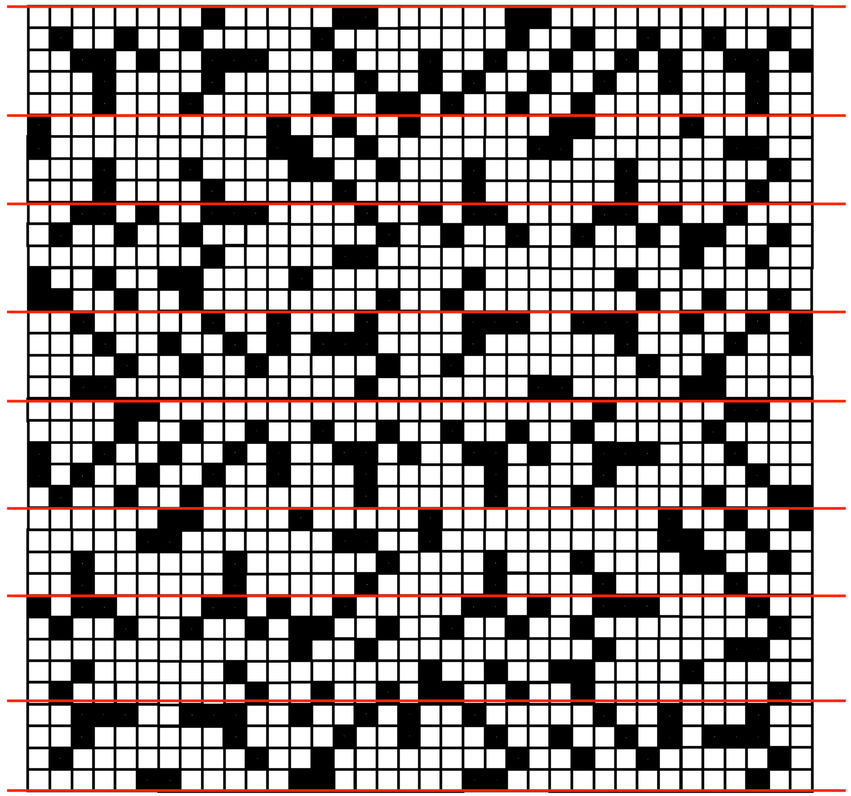
Stochastic Solutions

e.g. Game of Life (a cellular automaton)



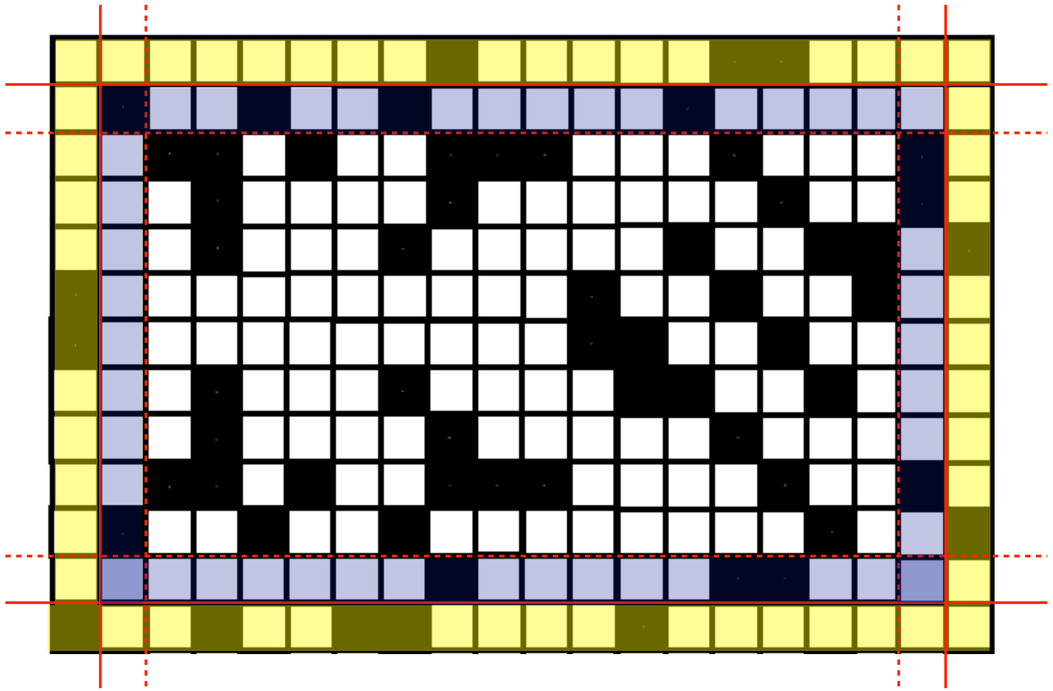
- Can try to decompose in the same number of dimensions as system
- But can also decompose in fewer dimensions
- Depending on grid dimensions, either might allow more even decomposition without resorting to irregular boundaries
- Higher dimension: more interior, but also more neighbours
- Sometimes 1D decomposition is enough

e.g. Game of Life (a cellular automaton)



- Can try to decompose in the same number of dimensions as system
- But can also decompose in fewer dimensions
- Depending on grid dimensions, either might allow more even decomposition without resorting to irregular boundaries
- Higher dimension: more interior, but also more neighbours
- Sometimes 1D decomposition is enough

e.g. Game of Life (a cellular automaton)



Set up non-blocking receives on all neighbours for their boundaries (yellow)

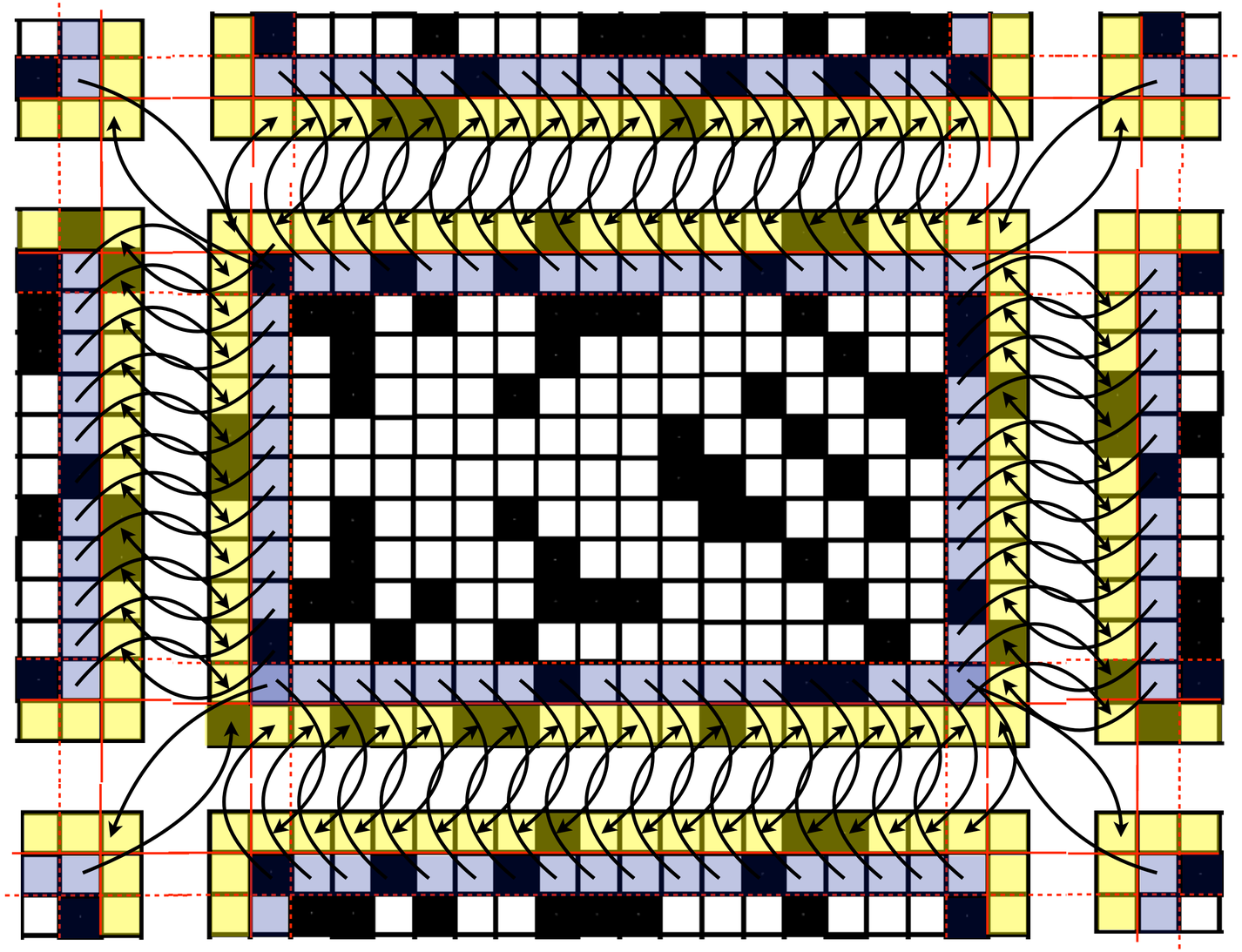
Set up non-blocking sends of your boundaries (blue) to each neighbour

Update the interior (B&W)

Wait on all comms

Update your boundaries

e.g. Game of Life (a cellular automaton)



Set up non-blocking receives on all neighbours for their boundaries (yellow)

Set up non-blocking sends of your boundaries (blue) to each neighbour

Update the interior (B&W)

Wait on all comms

Update your boundaries

Game of Life Exercise

- The repo includes `game_of_life.py`, which is a modified version of a simple `Tkinter` implementation by Philip Norton, described here
- Try some/all of the following

1. Separate the rendering and the updating of the grid
2. Use a simple functional decomposition with one worker doing the update and one doing the computation
3. Try using a task farm to compute blocks, sending boundaries back to the manager and letting it send them for the next step where they need to go. Does this work
4. Reimplement Game of Life using numpy. It should be much faster if you do it right. (Hint: don't iterate over the numpy arrays!)
5. Now do a proper grid decomposition, exchanging boundaries with neighbours, in using Isend, IRecv for the comms. Worry about updating the display later.
6. Write tests to make sure you parallel version is equivalent. To facilitate this, allow the initial configuration of the board to be loaded from file.

Other things we Haven't Covered

- Dynamic Process Management
  - It's possible to run an MPI program directly with Python instead of using mpiexec and the the Spawn a set of other processes. In this case, your communication link will be an MPI.Intracom, rather than an MPI.Intercom. It behaves slightly differently
- Broadcast: send same data to all workers
- Scatter/Gather:
  - Scatter: divide data up and send some to each process (including self)
  - Gather: gather partial results from each process (including self) and combine
- Map/Reduce: Like scatter/gather, but with reducing operation rather than (preserving) combine operation.
- All left as exercises for the reader. All covered in mp4py docs and examples

Stochastic
Solutions


stochasticsolutions.com • smartdatafoundry.com


https://github.com/stochasticsolutions/mpi4py-pydatalondon


njr@StochasticSolutions.com


https://linkedin.com/in/njradcliffe


@njr0 @tdda0 @StochasticSolns @SmartDataFdry

Zero, not letter "Oh"

https://github.com/stochasticsolutions/mpi4py-pydatalondon/blob/main/slides/mpi-pydata-london-2022.pdf