AMATH 483/583
High Performance Scientific Computing

Lecture 18:
Message Passing w/CSP/SPMD, MPI

Xu Tony Liu, PhD

Paul G. Allen School of Computer Science & Engineering

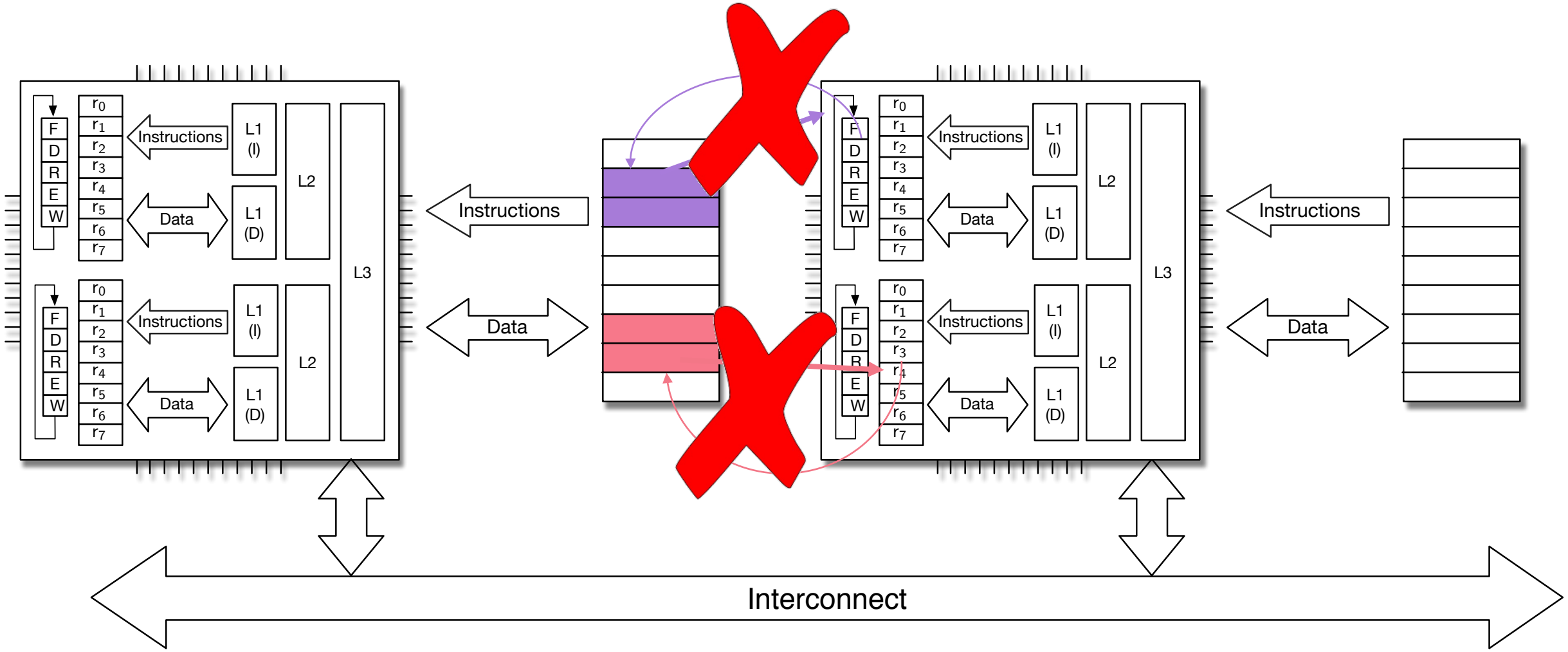
University of Washington

Seattle, WA

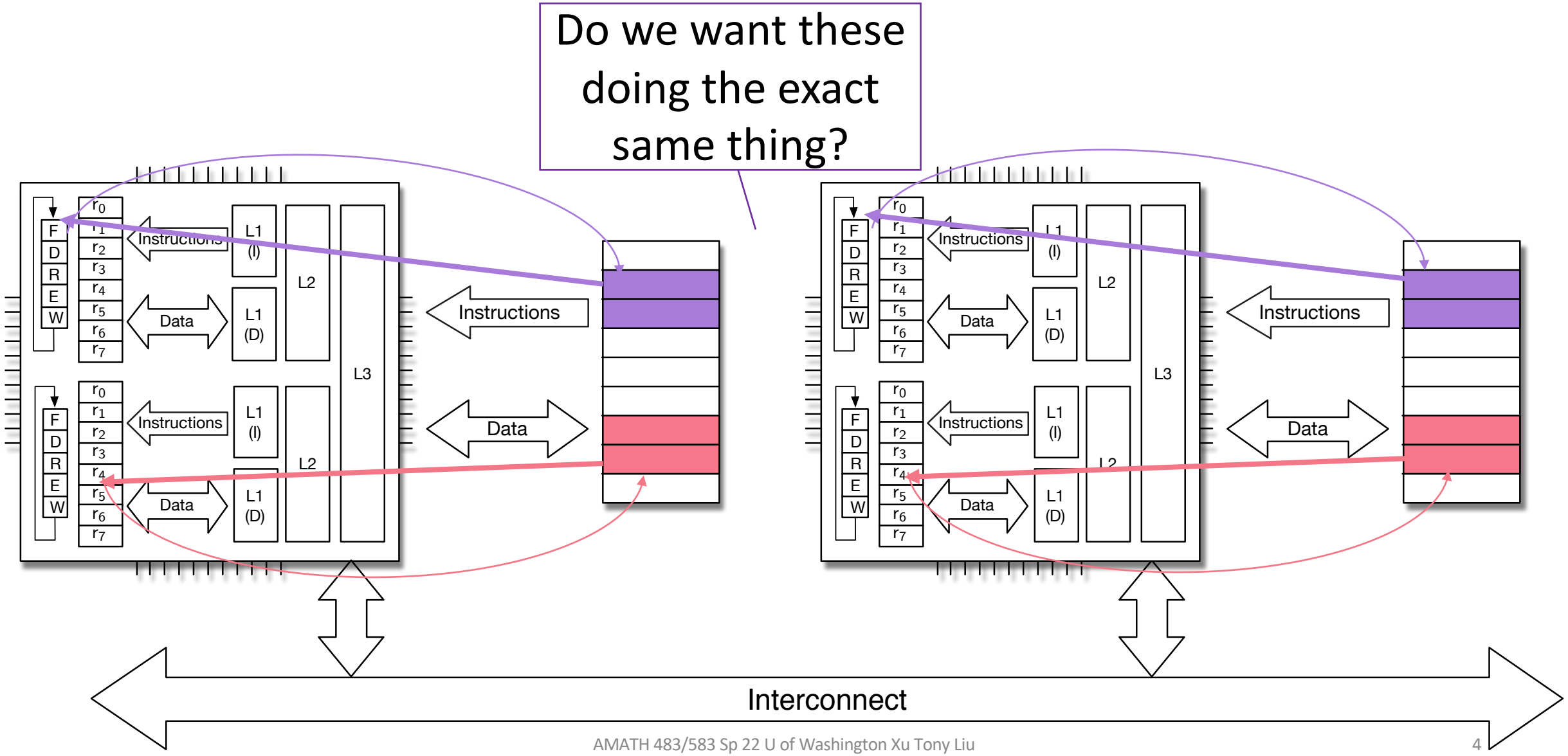
Overview

- SPMD / CSP recap
- MPI mental model
- Basic MPI
- Six Function MPI Point to Point Version
- Six Function MPI Collective Version
- Laplace’s equation on a regular grid

Distributed memory



Distributed memory

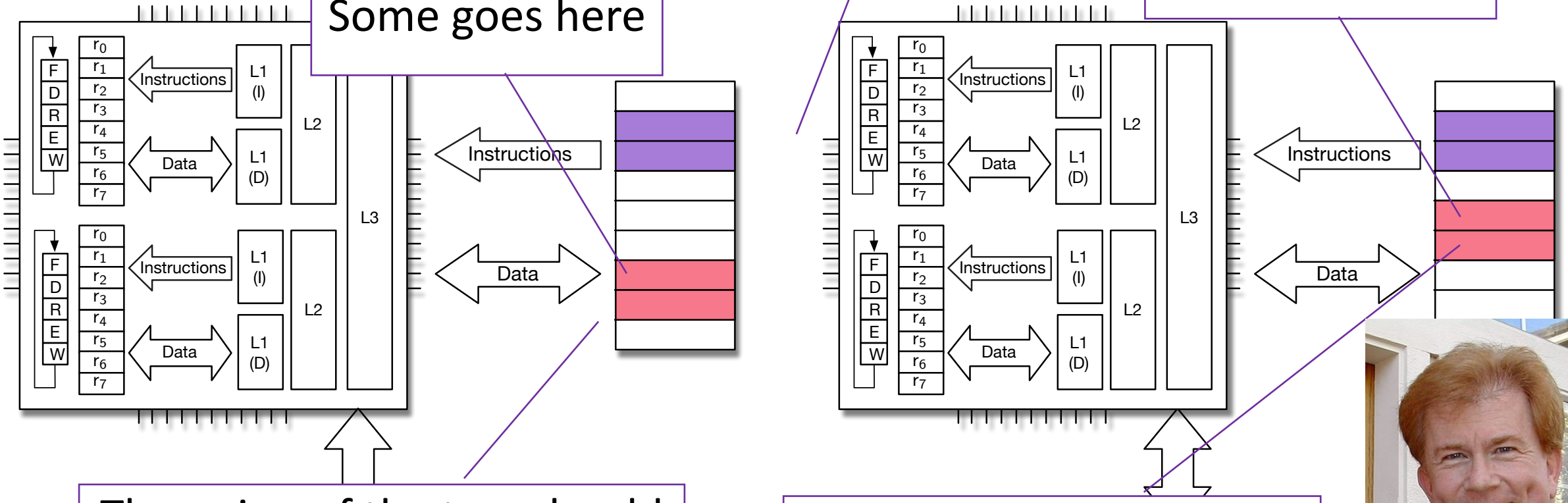


Distributed memory

But, Again. What do we keep? What do we not keep?

Some goes here

Some goes here

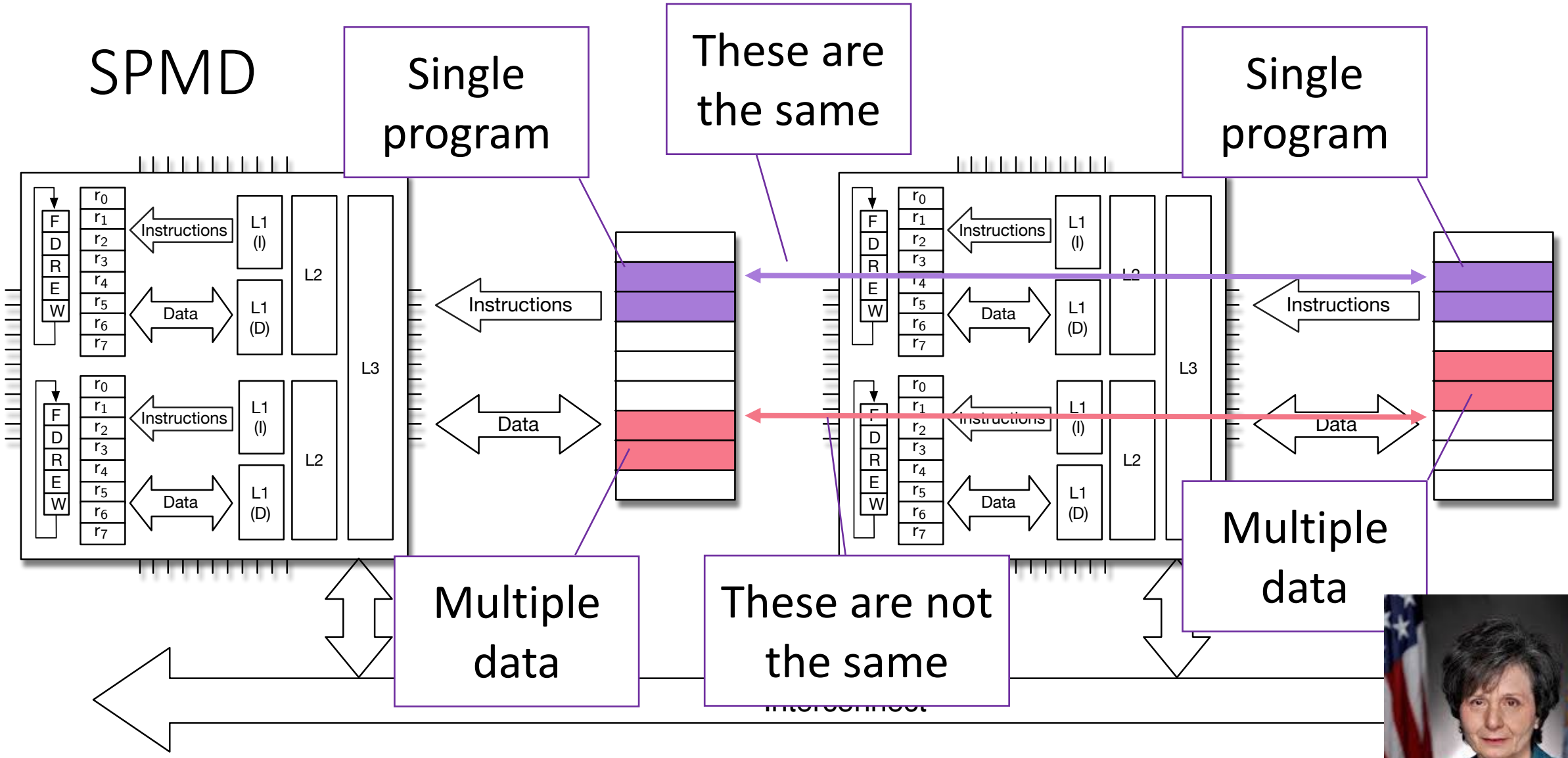


The union of the two should be the whole problem

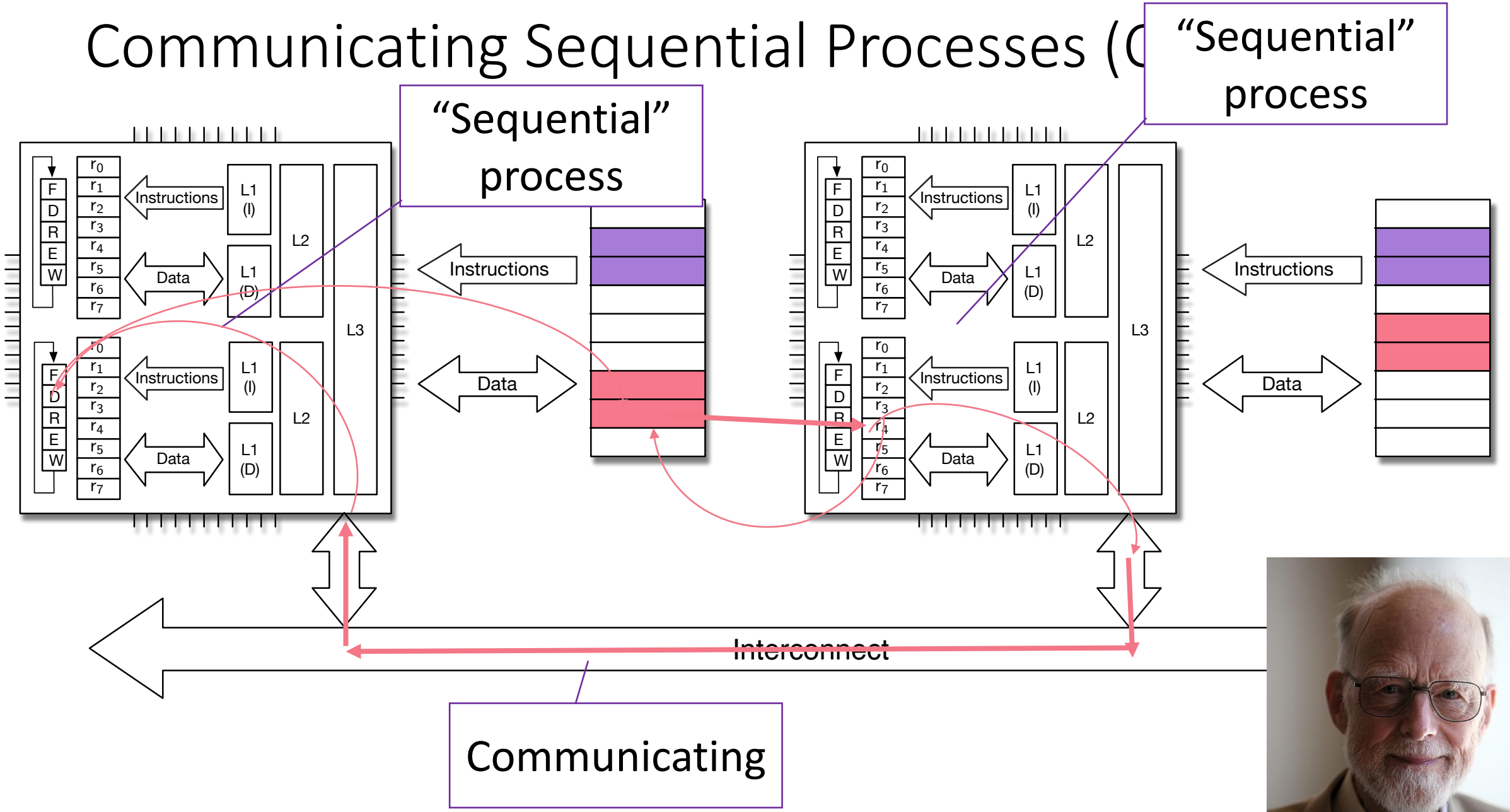
In

“Collectively exhaustive”

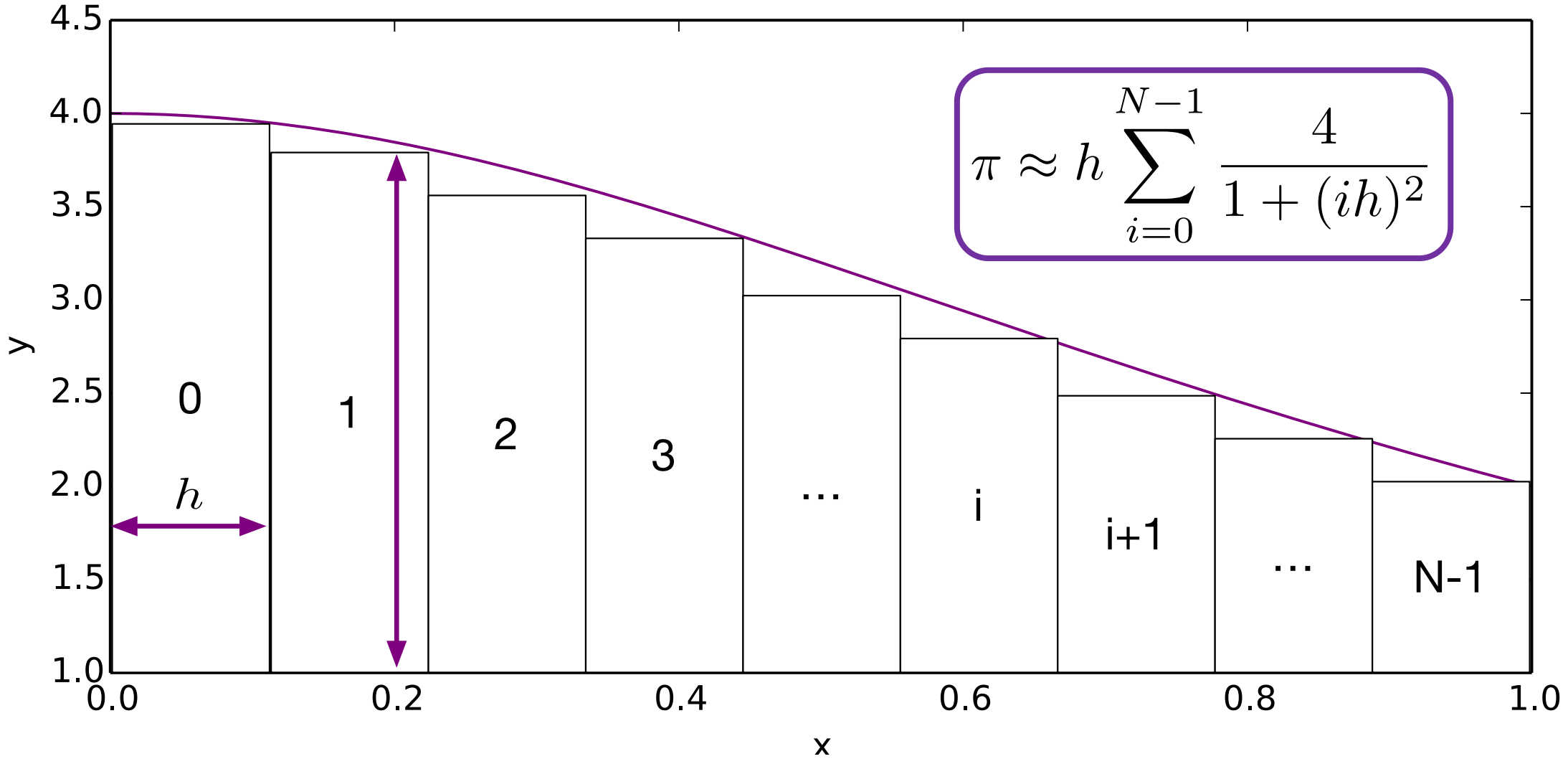
SPMD


Single program
These are the same
Single program
Multiple data
These are not the same
Multiple data
F
D
R
E
W
r_0
r_1
r_2
r_3
r_4
r_5
r_6
r_7
Instructions
Data
L1 (I)
L1 (D)
L2
L3
Instructions
Data
Instructions
Data
Interconnect

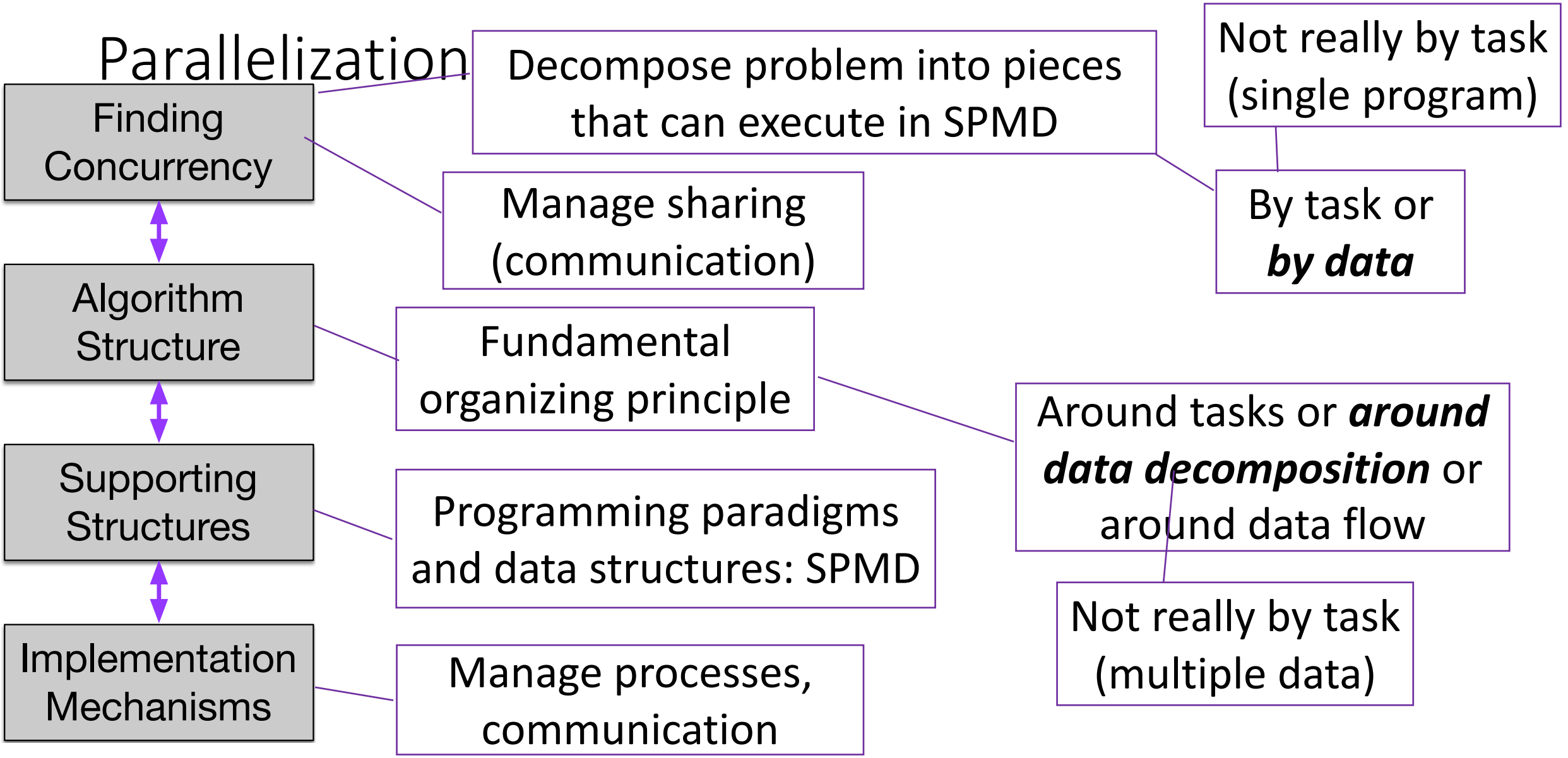
Communicating Sequential Processes (C



Numerical Quadrature



Parallelization



Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*(First ed.). Addison-Wesley Professional.

Finding Concurrency

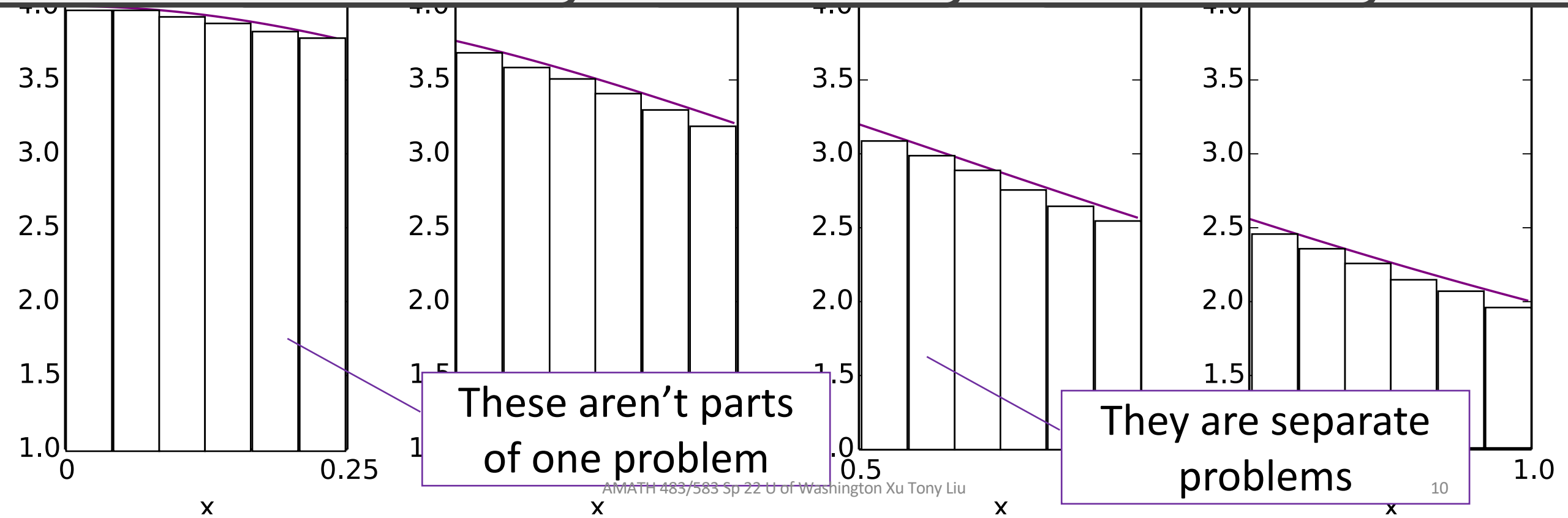
```
for (int i = begin; i < end; ++i) {
  pi += h * 4.0 / (1 + i*h*i*h);
}
```

```
int i = 0; i < N/4; ++i) {
+= h * 4.0 / (1 + i*h*i*h);
```

```
4; i < N/2; ++i) {
/ (1 + i*h*i*h);
```

```
N/2; i < 3*N/4; ++i) {
0 / (1 + i*h*i*h);
```

```
N/4; i < N
/ (1 + i*h
```



These aren't parts of one problem

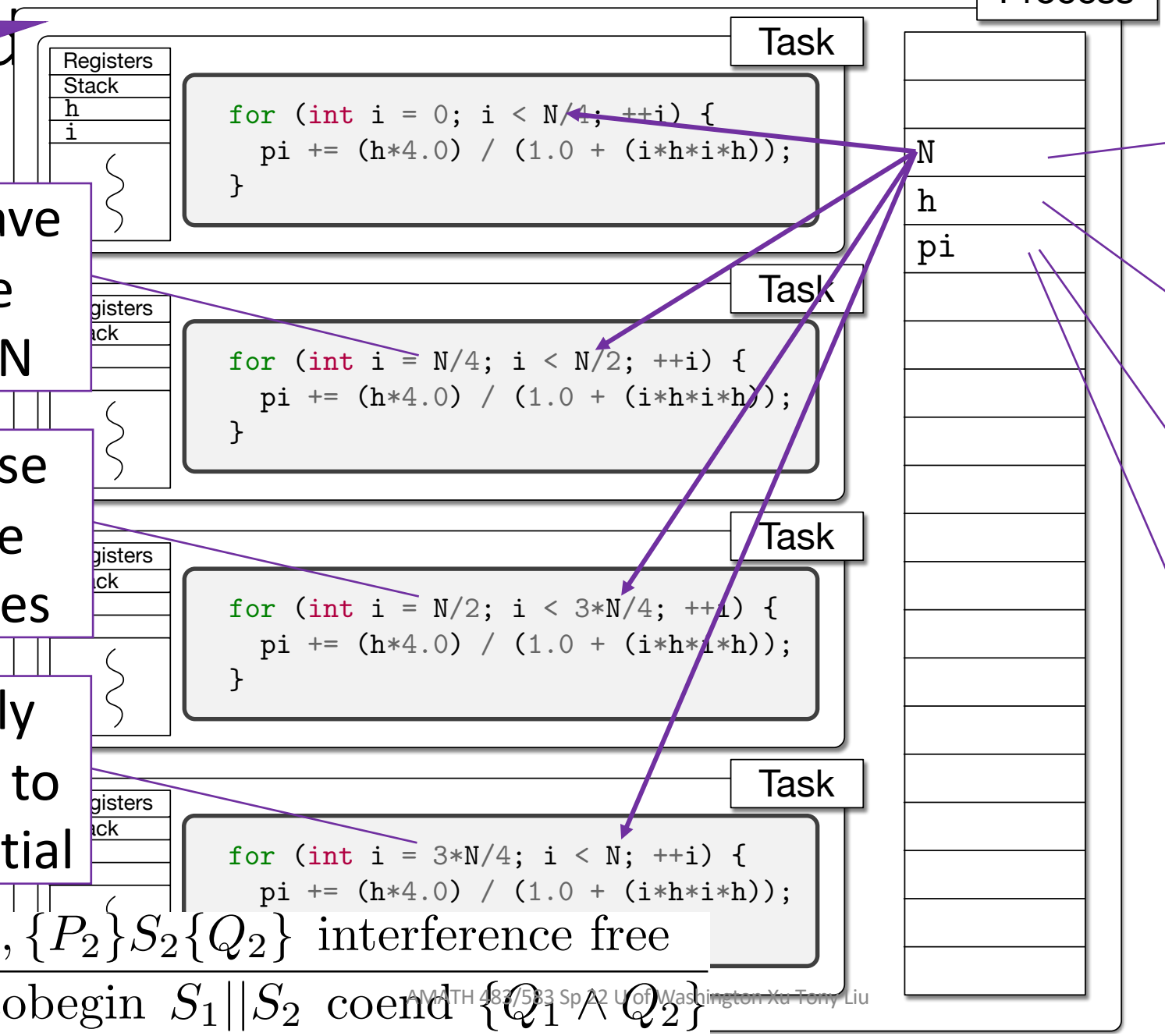
They are separate problems

Very Important Slide!!

Threads have the same value for N

And if these are all the same values

It is exactly equivalent to the sequential



Because they are reading ***the same*** N

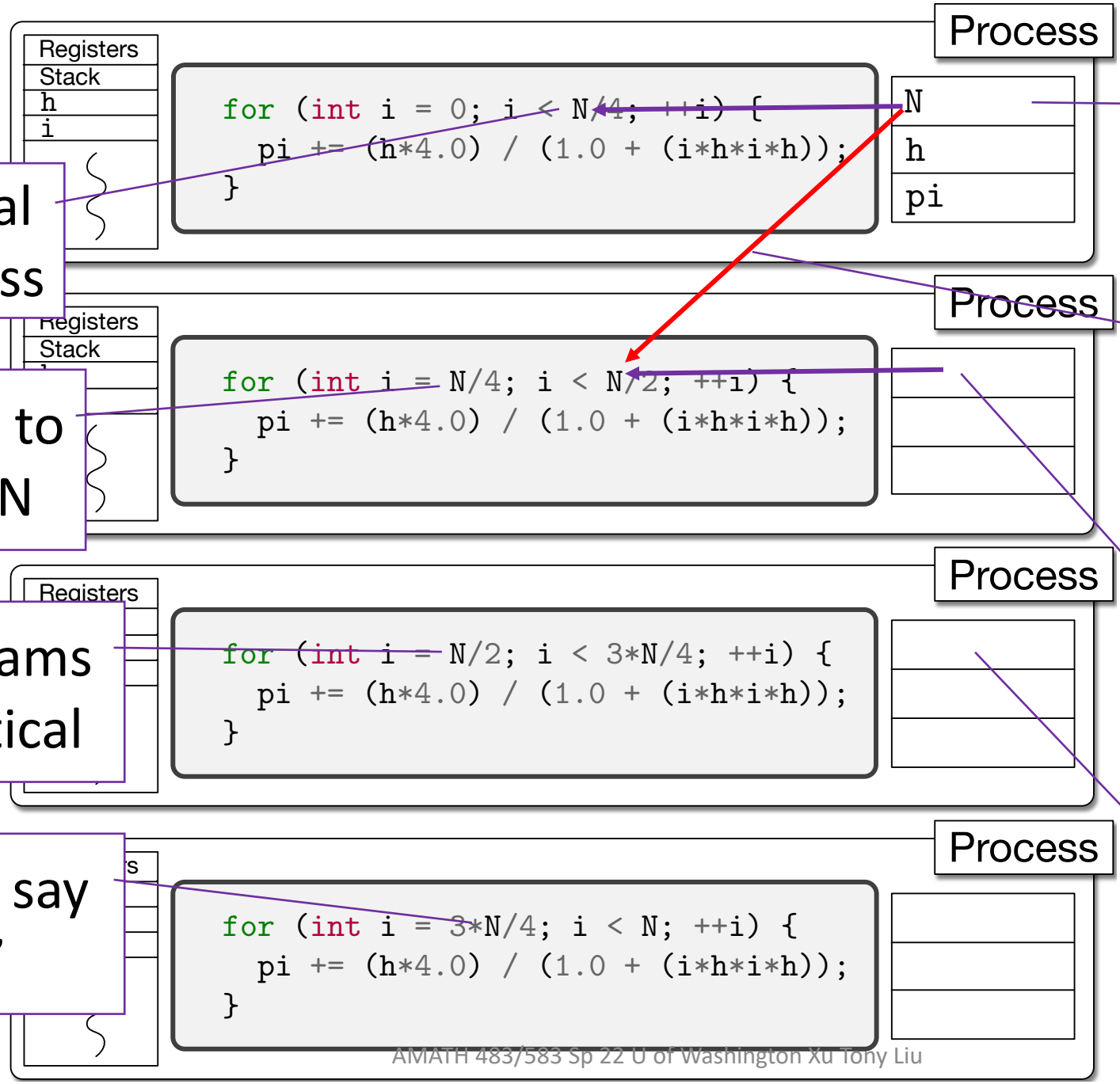
Similarly h

Similarly pi

At least for reading

(Have to deal with race when writing)

$$\frac{\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\} \text{ interference free}}{\{P_1\} \wedge \{P_2\} \text{ cobegin } S_1 || S_2 \text{ coend } \{Q_1 \wedge Q_2\}}$$



This N is local to this process

But we need to read ***some*** N

These programs are all identical

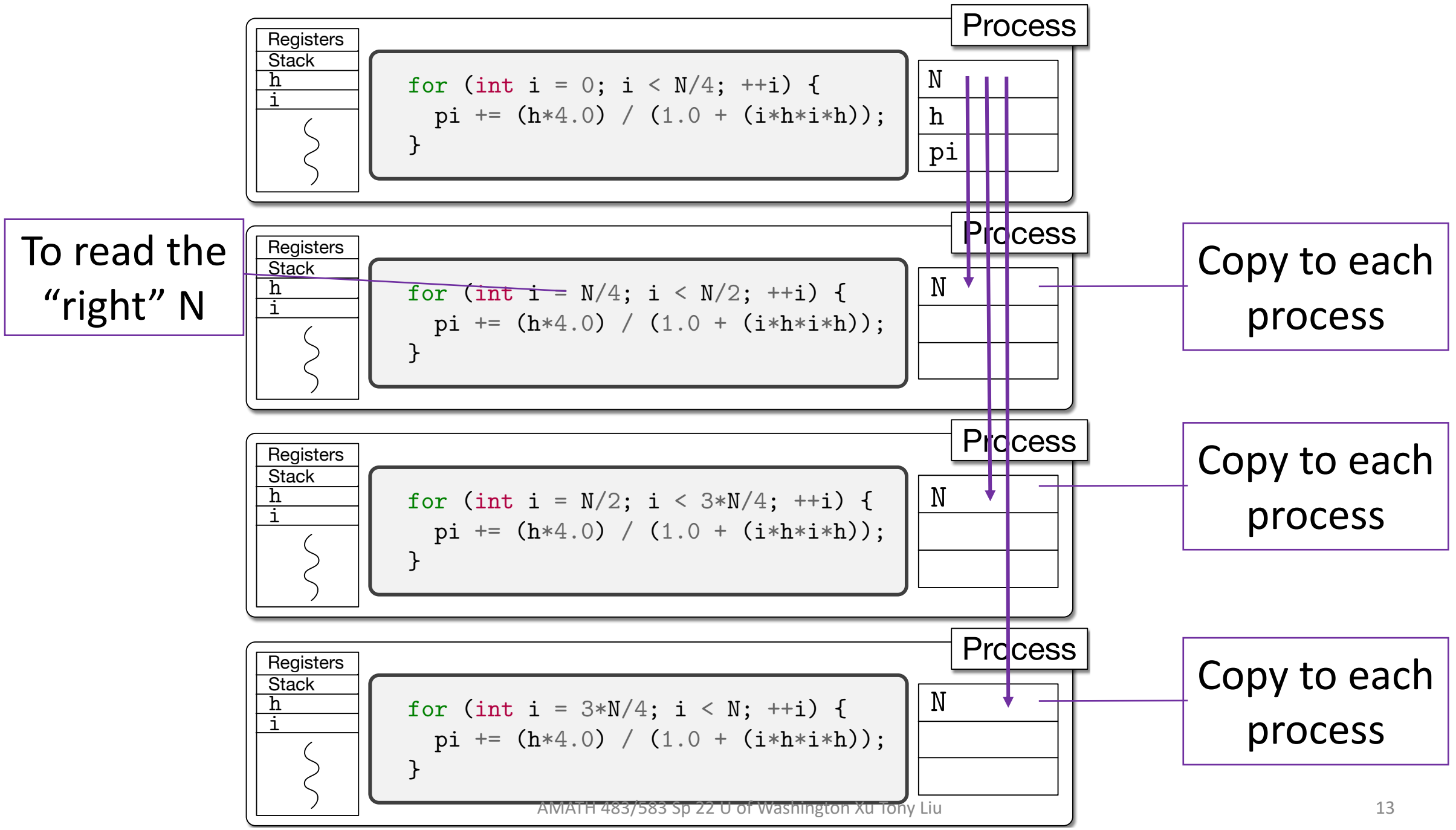
And they all say "read N"

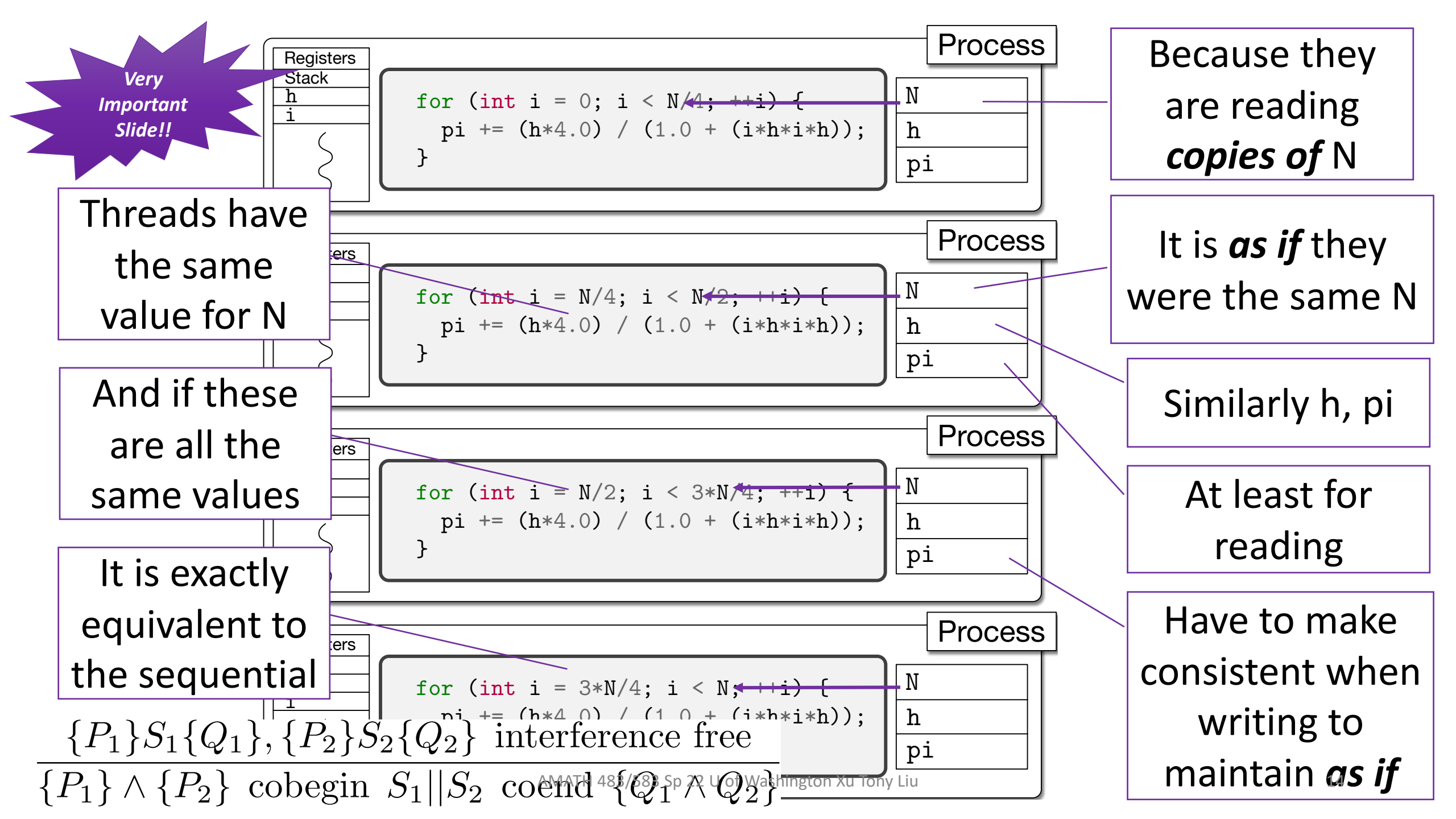
This N is local to this process

Can ***not*** read from another process memory

(In some sense there is an N here)

How do we get the right ***value*** for N here?


Process
Registers
Stack
h
i
for (int i = 0; i < N/4; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
h
pi
Process
Registers
Stack
h
i
for (int i = N/4; i < N/2; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
Process
Registers
Stack
h
i
for (int i = N/2; i < 3*N/4; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
Process
Registers
Stack
h
i
for (int i = 3*N/4; i < N; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
To read the "right" N
Copy to each process
Copy to each process
Copy to each process


Very Important Slide!!
Registers
Stack
h
i
Process
for (int i = 0; i < N/4; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
h
pi
Process
for (int i = N/4; i < N/2; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
h
pi
Process
for (int i = N/2; i < 3*N/4; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
h
pi
Process
for (int i = 3*N/4; i < N; ++i) {
N
h
pi
Threads have the same value for N
And if these are all the same values
It is exactly equivalent to the sequential
Because they are reading copies of N
It is as if they were the same N
Similarly h, pi
At least for reading
Have to make consistent when writing to maintain as if
$\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\}$ interference free
$\{P_1\} \wedge \{P_2\}$ cobegin $S_1 || S_2$ coend $\{Q_1 \wedge Q_2\}$

MPI

Get our id and number of other nodes

id 0 gets N

This pattern is ubiquitous

id 0 shares N

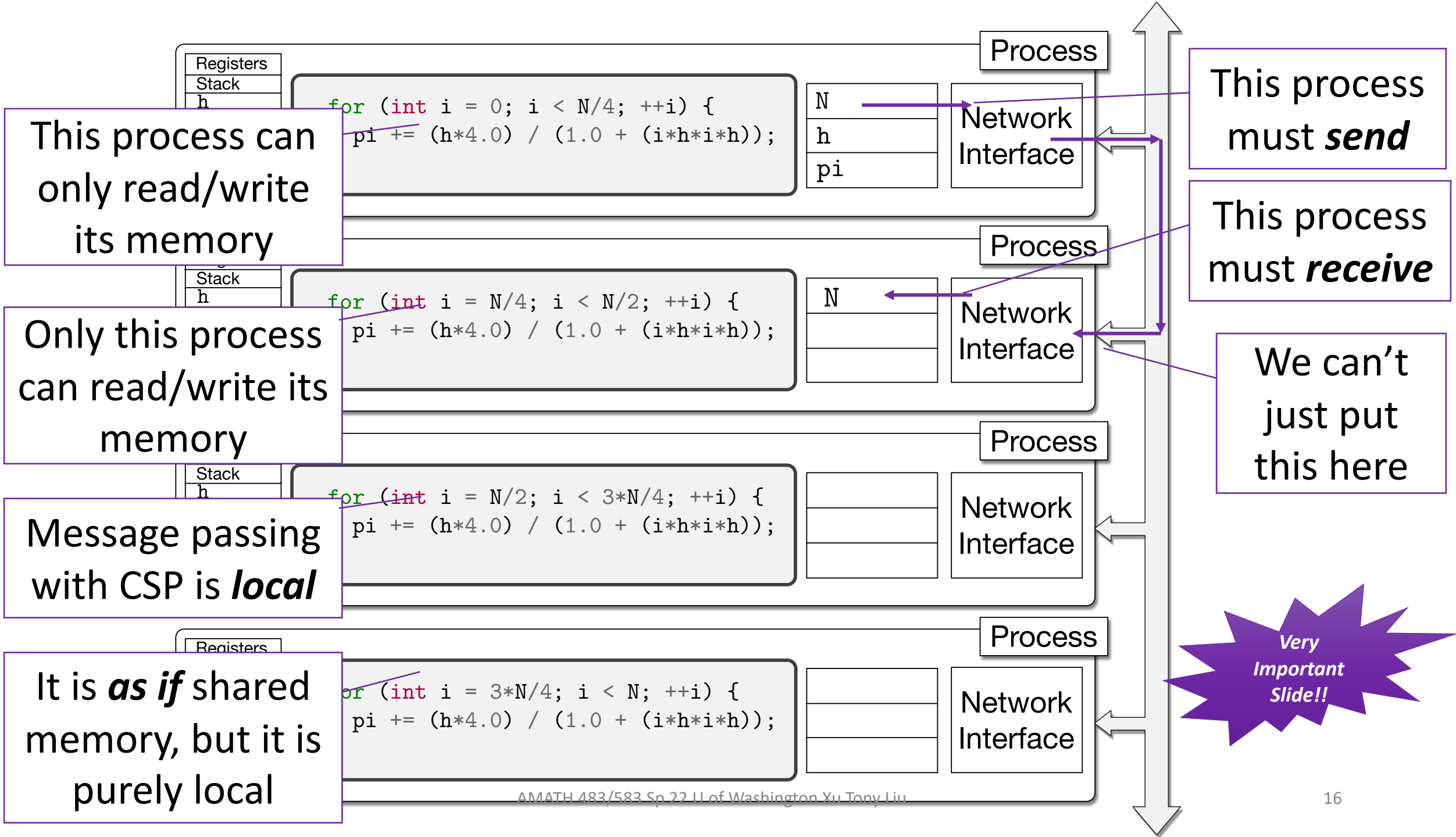
Everyone has same N

Everyone computes their own partial pi

id 0 collects all partials, adds them, and prints

```
int main(int argc, char* argv[]) {
  size_t intervals = 1024 * 1024;

  MPI::Init();

  int myrank = MPI::COMM_WORLD.Get_rank();
  int mysize = MPI::COMM_WORLD.Get_size();

  if (0 == myrank) {
    if (argc >= 2) intervals = std::atol(argv[1]);
  }

  MPI::COMM_WORLD.Bcast(&intervals, 1, MPI::UNSIGNED_LONG, 0);

  size_t blocksize = intervals / mysize;
  size_t begin     = blocksize * myrank;
  size_t end       = blocksize * (myrank + 1);
  double h         = 1.0 / ((double)intervals);

  double pi        = 0.0;
  for (size_t i = begin; i < end; ++i) {
    pi += 4.0 / (1.0 + (i * h * i * h));

  MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE, MPI::SUM, 0);

  if (0 == myrank) {
    std::cout << "pi is approximately " << pi << std::endl;
  }

  MPI::Finalize();

  return 0;
}
```

id 0 is root


Registers
Stack
h
for (int i = 0; i < N/4; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
N
h
pi
Network Interface
Process
Stack
h
for (int i = N/4; i < N/2; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
N
Network Interface
Process
Stack
h
for (int i = N/2; i < 3*N/4; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
Network Interface
Process
Registers
for (int i = 3*N/4; i < N; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
Network Interface
Process
This process can only read/write its memory
Only this process can read/write its memory
Message passing with CSP is local
It is as if shared memory, but it is purely local
This process must send
This process must receive
We can't just put this here
Very Important Slide!!

When we run this “parallel program” we aren’t running a parallel program

We are running multiple copies of this sequential program

All copies execute exactly this same code (not in lock step)

```
int main(int argc, char* argv[]) {
  size_t intervals = 1024 * 1024;

  MPI::Init();

  int myrank = MPI::COMM_WORLD.Get_rank();
  int mysize = MPI::COMM_WORLD.Get_size();

  if (0 == myrank) if (argc >= 2) intervals = std::atol(argv[1]);

  MPI::COMM_WORLD.Bcast(&intervals, 1, MPI::UNSIGNED_LONG, 0);

  size_t blocksize = intervals / mysize;
  size_t begin     = blocksize * myrank;
  size_t end       = blocksize * (myrank + 1);
  double h         = 1.0 / ((double)intervals);

  double pi        = 0.0;
  for (size_t i = begin; i < end; ++i)
    pi += 4.0 / (1.0 + (i * h * i * h));

  MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE, MPI::SUM, 0);

  if (0 == myrank) std::cout << "pi is approximately " << pi << std::endl;

  MPI::Finalize();

  return 0;
}
```

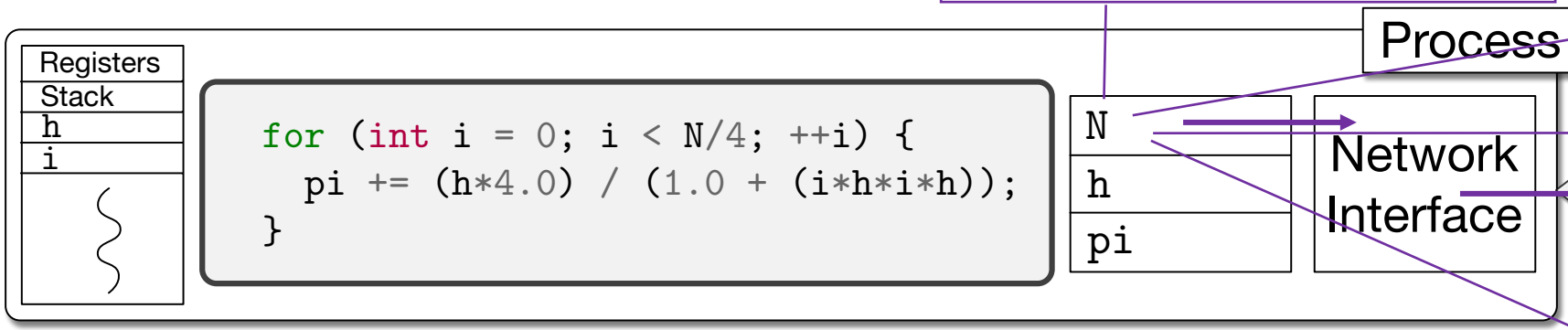
All copies call this communication function

And `intervals` gets copied to all processes

Only because each process calls this

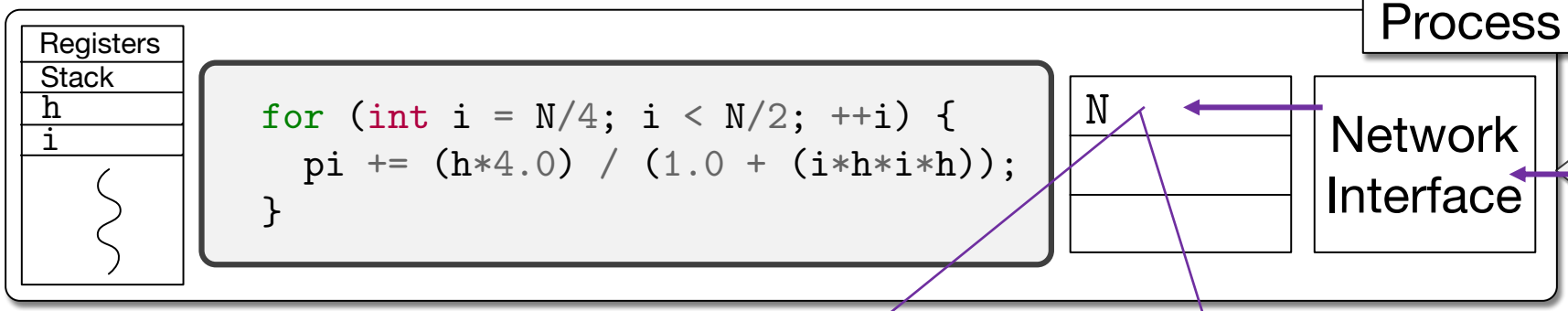
First question: What are we sending?

"N" only has meaning in source code



We are sending the ***value*** of "N"

The bits in the memory location



And we need to be able to id other processes

Second question: Where are we sending the bits?

How do we say "N@other_process"?

"N" only has meaning in source code

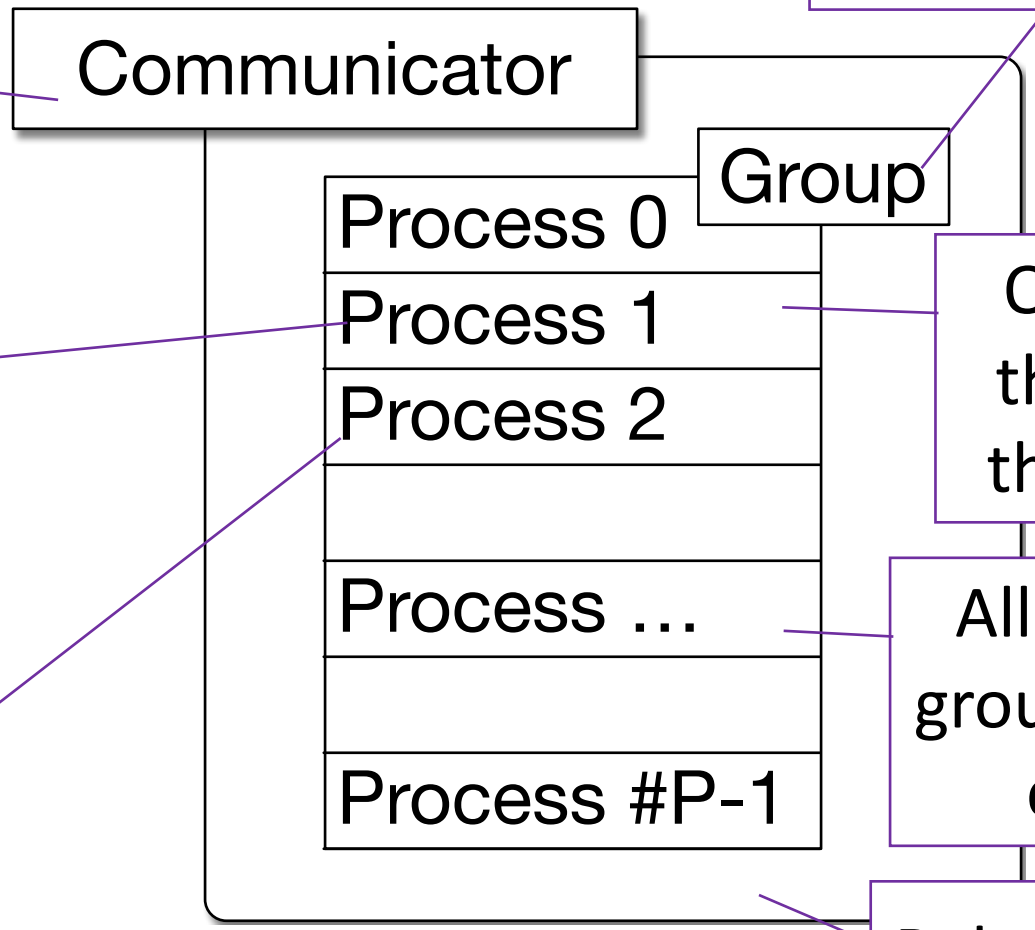
And its location only makes sense locally

But sender and receiver can agree on an alias

All MPI communication takes place in the context of an ***MPI Communicator***

An MPI Group translates from rank in the group to actual process

We us the index of a process in the group to identify other processes



An MPI Communicator contains an ***MPI Group***

Only processes in the group can use the communicator

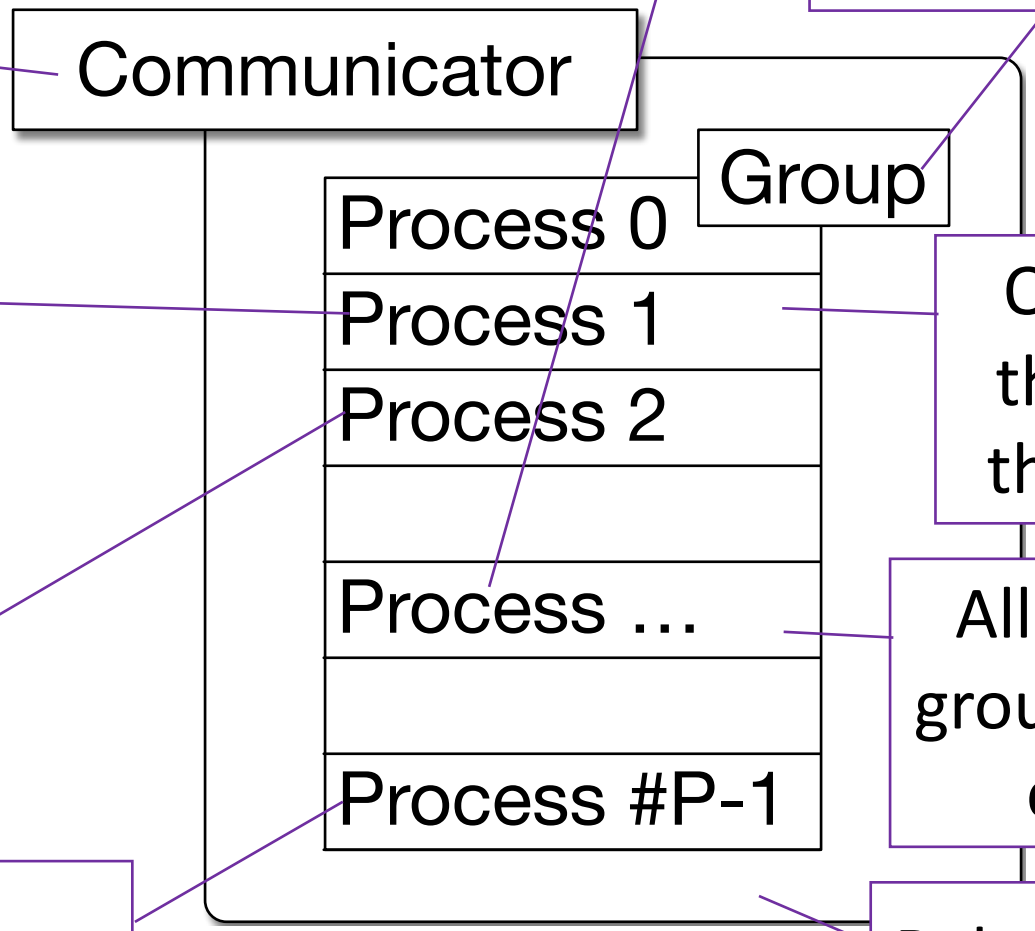
All processes in the group see an identical communicator

Behavior is ***as if*** it were global and shared

All MPI communication takes place in the context of an ***MPI Communicator***

Processes can query for size and for their own rank in group

An MPI Communicator contains an ***MPI Group***



An MPI Group translates from ***rank*** in the group to actual process

Only processes in the group can use the communicator

We us the index (***rank***) of a process in the group to identify other processes

All processes in the group see an identical communicator

The ***size*** of a communicator is the size of the group

Behavior is ***as if*** it were global and shared

MPI_Send

```
#include <mpi.h>
void Comm::Send(const void* buf, int count, const Datatype& datatype,
 ↪  int dest, int tag) const
```

Member function of a communicator

Communicator used for this message

Recipient

Message tag

Sender is implicit (the process that called this function

Message envelope

MPI_Recv

```
#include <mpi.h>
void Comm::Recv(void* buf, int count, const Datatype& datatype,
 ↪  int source, int tag, Status& status) const

void Comm::Recv(void* buf, int count, const Datatype& datatype,
 ↪  int source, int tag) const
```

Member function of a communicator

Overloaded function returns status

Communicator used for this message

Sender

Message tag

Message envelope

Receiver is implicit (the process that called this function

MPI_Send and MPI_Recv

```
#include <mpi.h>
void Comm::Send(const void* buf, int count, const Datatype& datatype,
 ↪  int dest, int tag) const
```

```
#include <mpi.h>
void Comm::Recv(void* buf, int count, const Datatype& datatype,
 ↪  int source, int tag, Status& status) const

void Comm::Recv(void* buf, int count, const Datatype& datatype,
 ↪  int source, int tag) const
```

?

Contents

Match these for message delivery

NB: SPMD

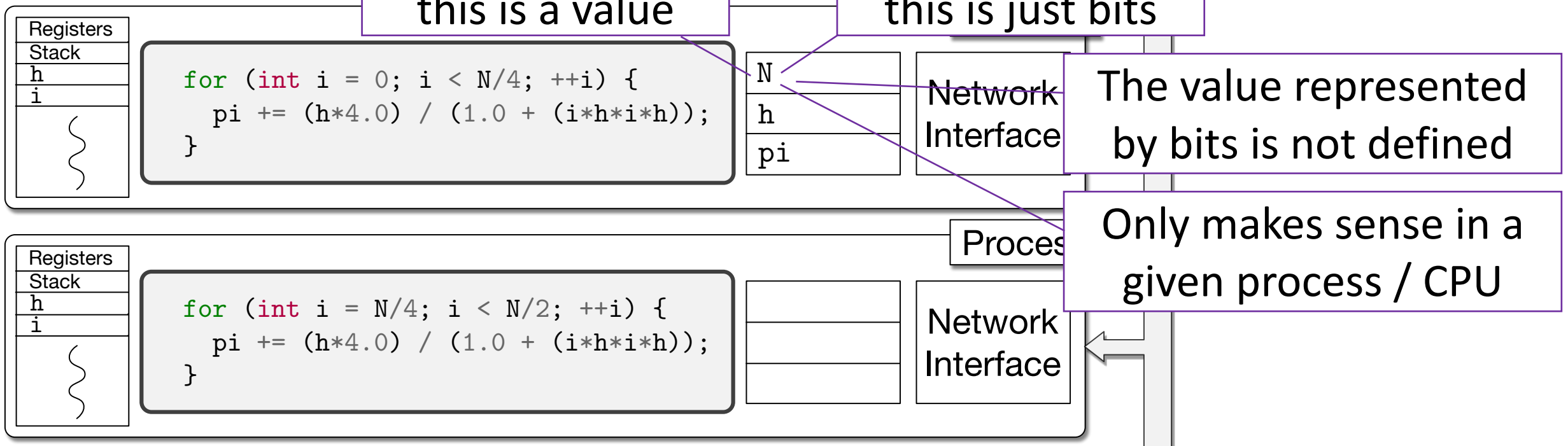
Message Contents

```
#include <mpi.h>
void Comm::Send(const void* buf, int count, const Datatype& datatype,
 ↪  int dest, int tag) const
```

Contents

In the program this is a value

In the computer this is just bits



The value represented by bits is not defined

Only makes sense in a given process / CPU

Message Contents

```
#include <mpi.h>
void Comm::Send(const void* buf, int count, const Datatype& datatype,
 ↪  int dest, int tag) const
```

Contents

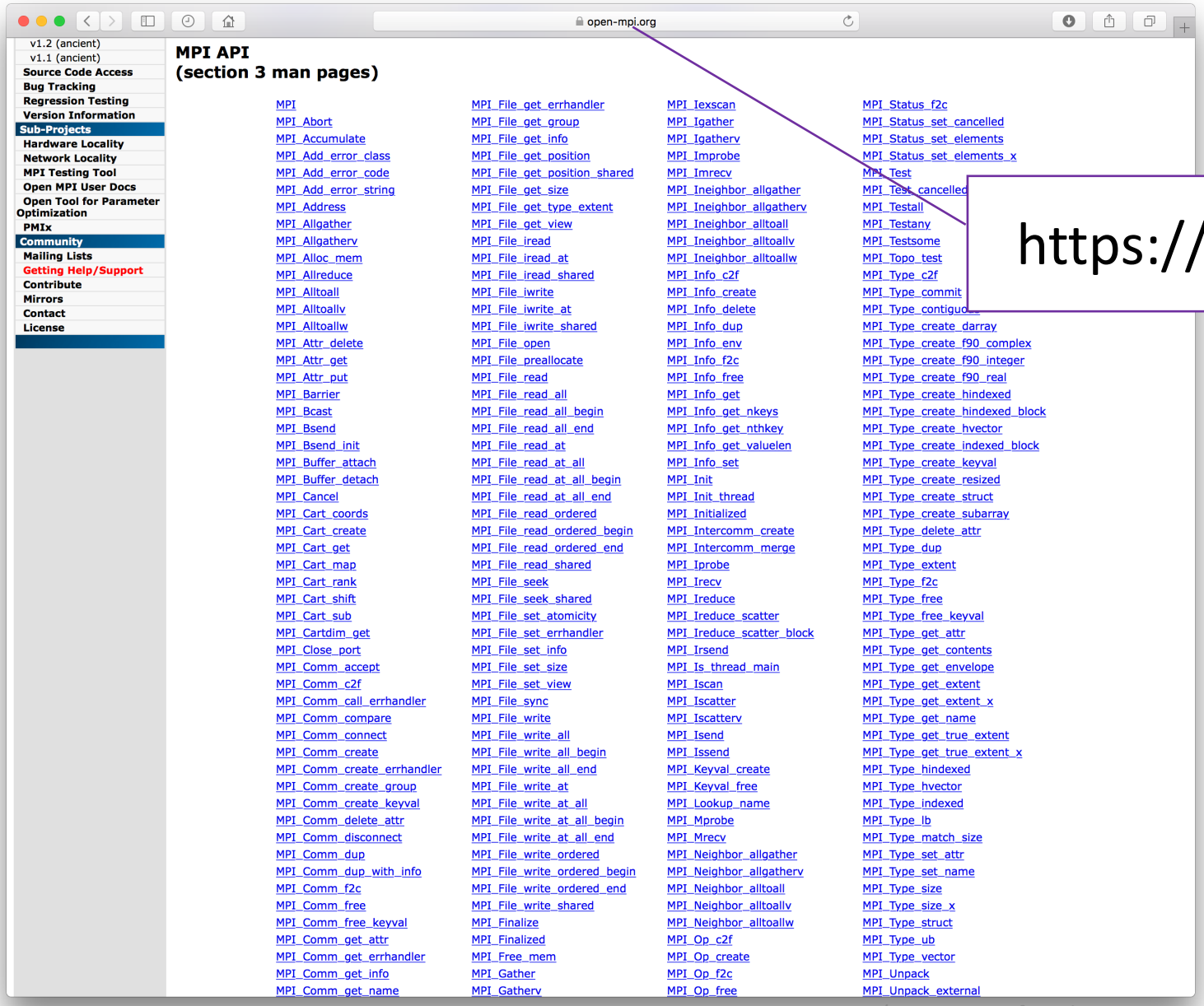
The location in memory of the bits we want to send

How many of the elements are in the message

How to interpret the bits as a data element

Note that contents are not part of envelope

Documentation of All MPI Functions



https://www.open-mpi.org/doc/v1.8/

Six Function MPI (Point to Point)

```
#include <mpi.h>

void MPI::Init(int& argc, char**& argv)
void MPI::Init()

int MPI::Comm::Get_size() const

int MPI::Comm::Get_rank() const

void MPI::Comm::Send(const void* buf, int count, const Datatype&
 ↪  datatype, int dest, int tag) const

void MPI::Comm::Recv(void* buf, int count, const Datatype& datatype,
 ↪  int source, int tag, Status& status) const
void MPI::Comm::Recv(void* buf, int count, const Datatype& datatype,
 ↪  int source, int tag) const

void MPI::Finalize()
```

Initialize MPI environment

Get size of communicator

Get rank of communicator

Send

Receive

Shut down and leave MPI environment

Aside

```c
#include <mpi.h>

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
 ↪  int source, int tag, MPI_Comm comm, MPI_Status *status)
```

MPI has C bindings for all functions

```
INCLUDE 'mpif.h'

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS,
 ↪  IERROR)
    <type>     BUF(*)
    INTEGER     COUNT, DATATYPE, SOURCE, TAG, COMM
    INTEGER     STATUS(MPI_STATUS_SIZE), IERROR
```

And Fortran bindings

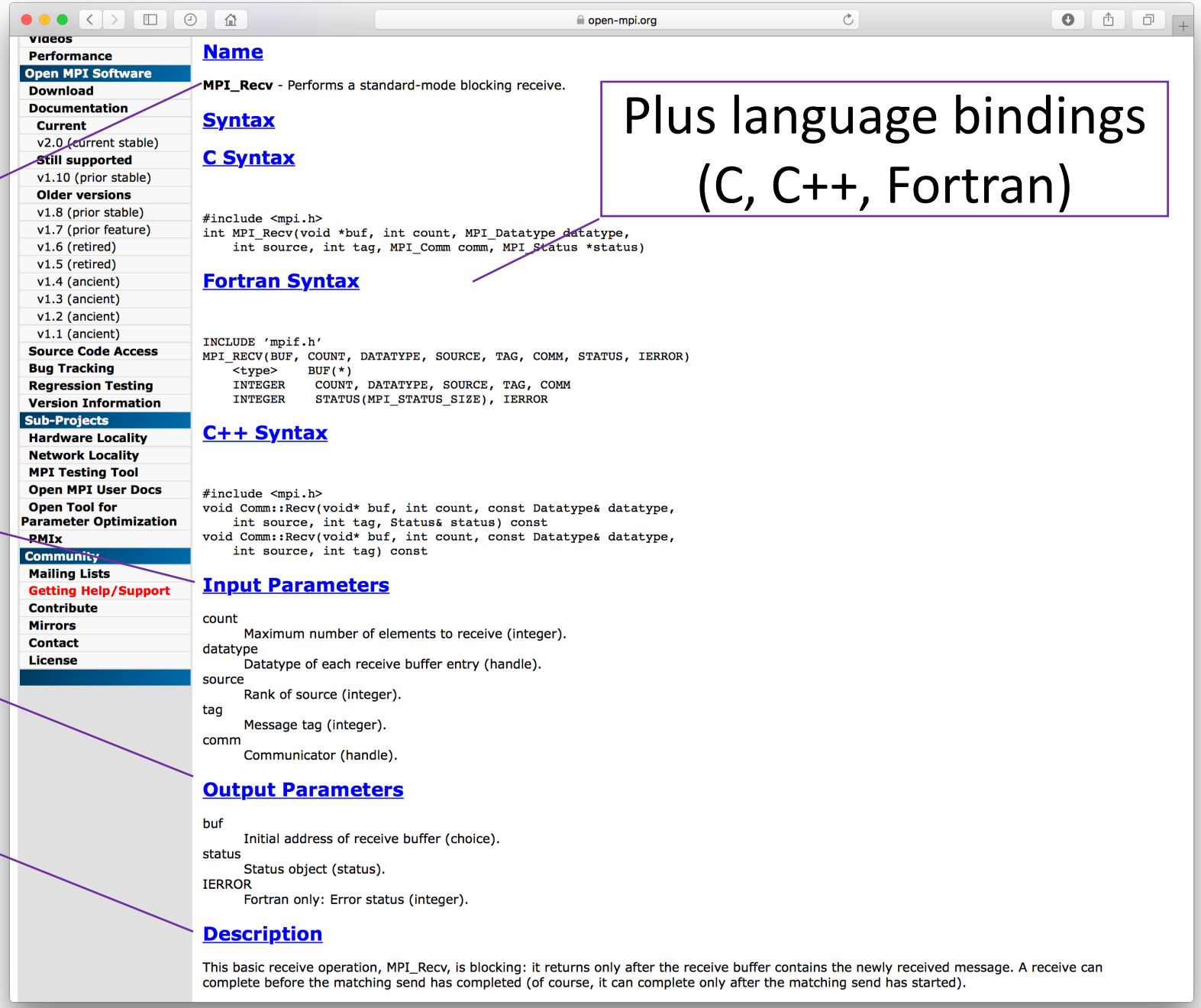
MPI Functions

Functions are defined independently of any language

Including parameters

Including parameters

And semantics



Plus language bindings (C, C++, Fortran)

Hello MPI World

```cpp
#include <iostream>
#include <mpi.h>

int main() {

  MPI::Init();

  int mysize = MPI::COMM_WORLD.Get_size();
  int myrank = MPI::COMM_WORLD.Get_rank();

  std::cout << "Hello World!";
  std::cout << "I am " << myrank << " of " << mysize << std::endl;

  MPI::Finalize();

  return 0;
}
```

Include mpi.h

NB: namespace MPI

Initialize

Get size of communicator

Get rank of calling process

Finalize

Hello MPI World

```cpp
#include <iostream>
#include <mpi.h>

int main() {

  MPI::Init();

  int mysize = MPI::COMM_WORLD.Get_size();
  int myrank = MPI::COMM_WORLD.Get_rank();

  std::cout << "Hello World!";
  std::cout << "I am " << myrank << " of " << mysize << std::endl;

  MPI::Finalize();

  return 0;
}
```

MPI defines that a default communicator exists after MPI::Init()

Recall that send, receive, etc all referred to a communicator

Creating other communicators is done by program

Named MPI::COMM_WORLD

MPI::COMM_WORLD is usually sufficient for many programs

Compiling and Running

```
$ mpic++ hello.cpp
```

Usually we use a compiler wrapper set up for local development environment

```
$ mpirun -np 4 ./a.out
```

Launch 4 copies of a.out

```
Hello World! I am 0 of 4
Hello World! I am 3 of 4
Hello World! I am 1 of 4
Hello World! I am 2 of 4
```

Output (printed from all processes since this was local on my laptop)

Compiling and Running

```
$ mpic++ hello.cpp
```

Where did compiler come from? mpi.h? The actual MPI functions?

```
$ mpirun -np 4 ./a.out
```

Where did mpirun come from

- MPI is just a library interface specification (with language bindings)
- It is up the community (researchers, vendors, et al) to provide implementations the conform to the standard specification
- High-quality implementations have useful extensions

Open MPI, MPICH, Intel MPI

Ping Pong

```cpp
int main() {

  MPI::Init();

  int myrank = MPI::COMM_WORLD.Get_rank();
  int mysize = MPI::COMM_WORLD.Get_size();

  int ballsent = 42, ballreceived = 0;
  MPI::COMM_WORLD.Send(&ballsent, 1, MPI::INT, 1, 321);
  MPI::COMM_WORLD.Recv(&ballsent, 1, MPI::INT, 0, 321);

  MPI::COMM_WORLD.Send(&ballreceived, 1, MPI::INT, 0, 321);
  MPI::COMM_WORLD.Recv(&ballreceived, 1, MPI::INT, 1, 321);
  std::cout << "Received " << ballreceived << std::endl;

  MPI::Finalize();

  return 0;
}
```

Ping Pong

```
$ mpic++ pingpong.cpp

$ mpirun -np 2 ./a.out

Received 42
... ^C ....  Process terminated
```

Ping Pong – What Went Wrong?

```
int main() {

  MPI::Init();

  int myrank = MPI::COMM_WORLD.Get_rank();
  int mysize = MPI::COMM_WORLD.Get_size();

  int ballsent = 42, ballreceived = 0;
  MPI::COMM_WORLD.Send(&ballsent, 1, MPI::INT, 1, 321);
  MPI::COMM_WORLD.Recv(&ballsent, 1, MPI::INT, 0, 321);

  MPI::COMM_WORLD.Send(&ballreceived, 1, MPI::INT, 0, 3
  MPI::COMM_WORLD.Recv(&ballreceived, 1, MPI::INT, 1, 3
  std::cout << "Received " << ballreceived << std::endl

  MPI::Finalize();

  return 0;
}
```

All processes run this same program

Both processes send this

And try to receive

Ping Pong 2.0

Only process 0 receives this

Only process 1 receives this

Only process 1 sends this

Only process 0 sends this

```
int main() {

  MPI::Init();

  int myrank = MPI::COMM_WORLD.Get_rank();
  int mysize = MPI::COMM_WORLD.Get_size();

  int ballsent = 42, ballreceived = 0;
  if (0 == myrank) {
    MPI::COMM_WORLD.Send(&ballsent, 1, MPI::INT, 1, 321);
    MPI::COMM_WORLD.Recv(&ballreceived, 1, MPI::INT, 1, 321);
    std::cout << "Received " << ballreceived << std::endl;
  }
  if (1 == myrank) {
    MPI::COMM_WORLD.Recv(&ballreceived, 1, MPI::INT, 0, 321);
    MPI::COMM_WORLD.Send(&ballsent, 1, MPI::INT, 0, 321);
  }

  MPI::Finalize();

  return 0;
}
```

Ping Pong 2.0

```
$ mpic++ pingpong.cpp

$ mpirun -np 2 ./a.out

Received 42

$
```

Ping Pong 2.0

```
$ mpic++ pingpong.cpp

$ mpirun -np 8 ./a.out

Received 42

$
```

Six Function MPI Point to Point Version

```
#include <mpi.h>

void MPI::Init(int& argc, char**& argv)
void MPI::Init()

int MPI::Comm::Get_size() const

int MPI::Comm::Get_rank() const

void MPI::Comm::Send(const void* buf, int count, const Datatype&
↪  datatype, int dest, int tag) const

void MPI::Comm::Recv(void* buf, int count, const Datatype& datatype,
↪  int source, int tag, Status& status) const
void MPI::Comm::Recv(void* buf, int count, const Datatype& datatype,
↪  int source, int tag) const

void MPI::Finalize()
```

The Other Six Function MPI

Broadcast values to all nodes

All nodes do exactly this

Collect results from all nodes

What is this?

```
int main(int argc, char* argv[]) {
  size_t intervals = 1024 * 1024;

  MPI::Init();

  int myrank = MPI::COMM_WORLD.Get_rank();
  int mysize = MPI::COMM_WORLD.Get_size();

  if (0 == myrank) if (argc >= 2) intervals = std::atol(argv[1]);

  MPI::COMM_WORLD.Bcast(&intervals, 1, MPI::UNSIGNED_LONG, 0);

  size_t blocksize = intervals / mysize;
  size_t begin     = blocksize * myrank;
  size_t end       = blocksize * (myrank + 1);
  double h         = 1.0 / ((double)intervals);

  double pi        = 0.0;
  for (size_t i = begin; i < end; ++i)
    pi += 4.0 / (1.0 + (i * h * i * h));

  MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE, MPI::SUM, 0);

  if (0 == myrank) std::cout << "pi is approximately " << pi << std::endl;

  MPI::Finalize();

  return 0;
}
```

Six Function MPI Collective Version

```
#include <mpi.h>

void MPI::Init(int& argc, char**& argv)
void MPI::Init()

int MPI::Comm::Get_size() const
int MPI::Comm::Get_rank() const

void MPI::Comm::Bcast(void *buf, int count, const Datatype& datatype,
 ↪  int root);
void MPI::Comm::Reduce(void *buf, int count, const Datatype&
 ↪  datatype, const Op& op, int root);

void MPI::Finalize()
```

Initialize MPI environment

Get size of communicator

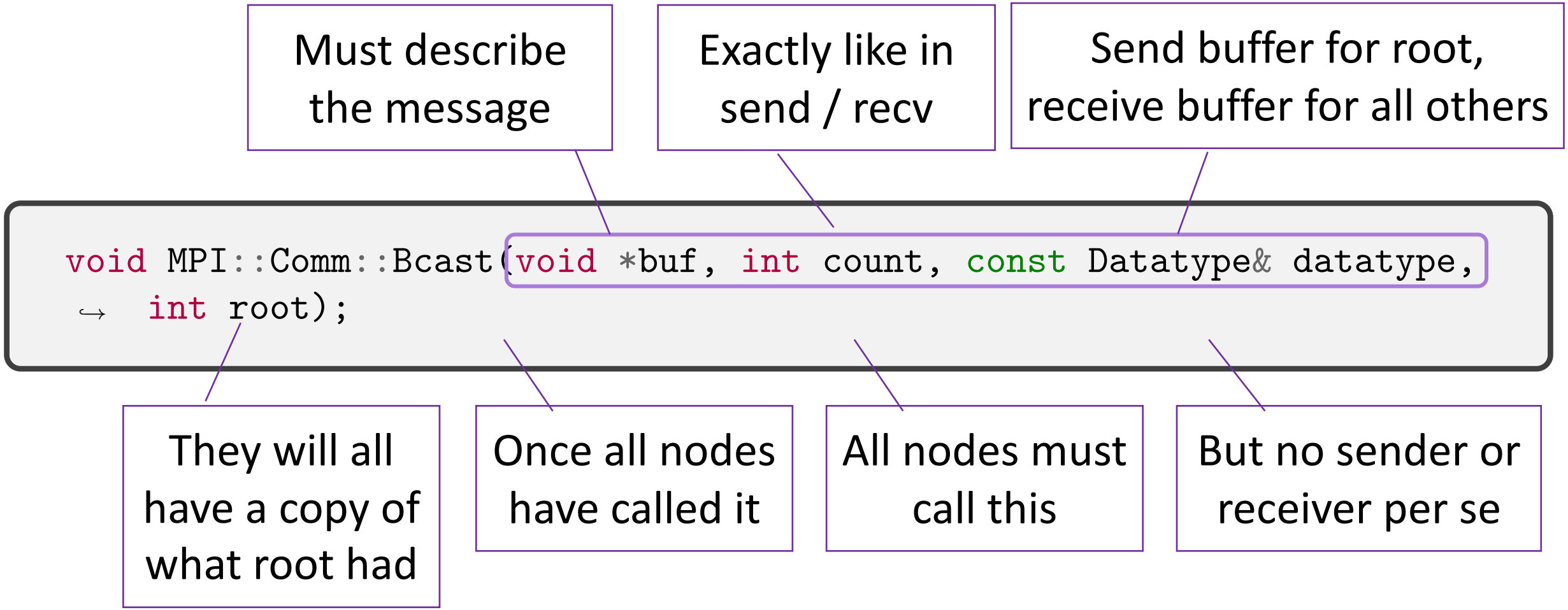
Get rank of communicator

Broadcast values to all nodes

Collect results from all nodes

Shut down and leave MPI environment

MPI Bcast


Must describe the message
Exactly like in send / recv
Send buffer for root, receive buffer for all others
void MPI::Comm::Bcast(void *buf, int count, const Datatype& datatype,
↪ int root);
They will all have a copy of what root had
Once all nodes have called it
All nodes must call this
But no sender or receiver per se

MPI Reduce

Must describe the message

Exactly like in send / recv

Receive buffer for root, send buffer for all others

```
void MPI::Comm::Reduce(void *buf, int count, const Datatype& datatype,
 ↪  const Op& op, int root);
```

Reduced with this operation

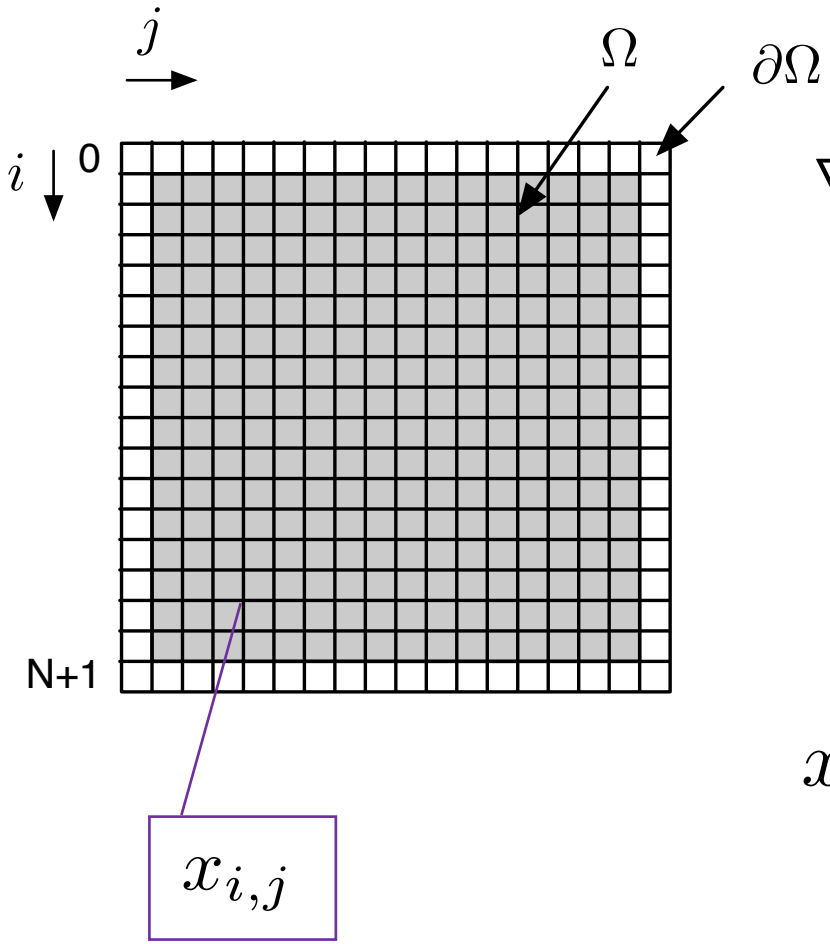
Once all nodes have called it

All nodes must call this

But no sender or receiver per se

Root will have a value reduced from all the others

Laplace's Equation on a Regular Grid



$$\begin{aligned} \nabla^2\phi &= 0 \quad \text{on } \Omega \\ \phi &= f \quad \text{on } \partial\Omega \end{aligned}$$

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ & & -1 & \cdots & -1 & 4 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \end{bmatrix}$$

Discretization

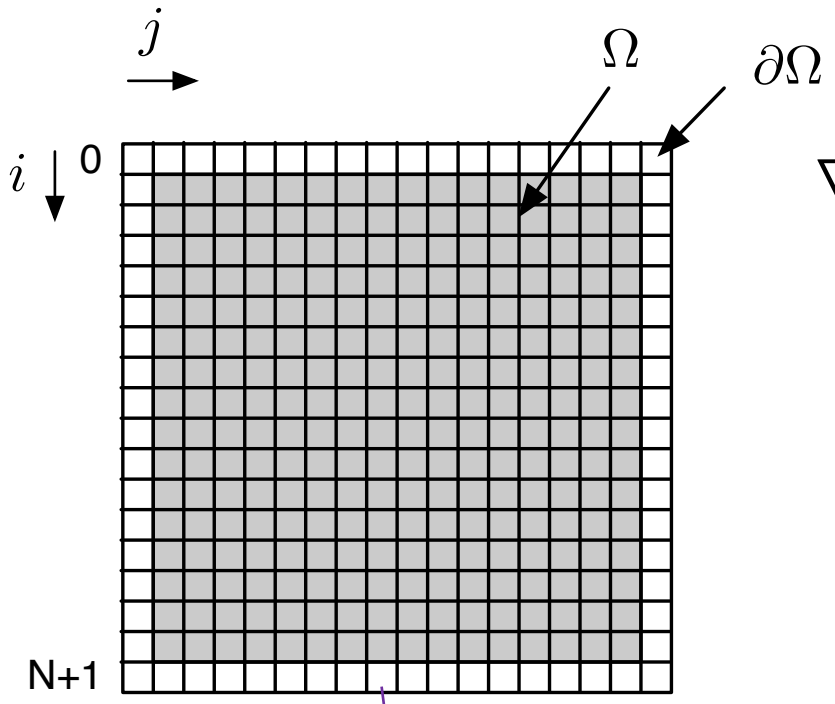
$$x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1} - 4x_{i,j} = 0$$

$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4$$

The value of each point on the grid

The average of its neighbors

Laplace's Equation on a Regular Grid



The boundary is non-zero

Why isn't 0 the solution?

$$\begin{aligned} \nabla^2\phi &= 0 \quad \text{on } \Omega \\ \phi &= f \quad \text{on } \partial\Omega \end{aligned}$$

Discret

The boundary is non-zero

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ & & -1 & \cdots & -1 & 4 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \end{bmatrix}$$

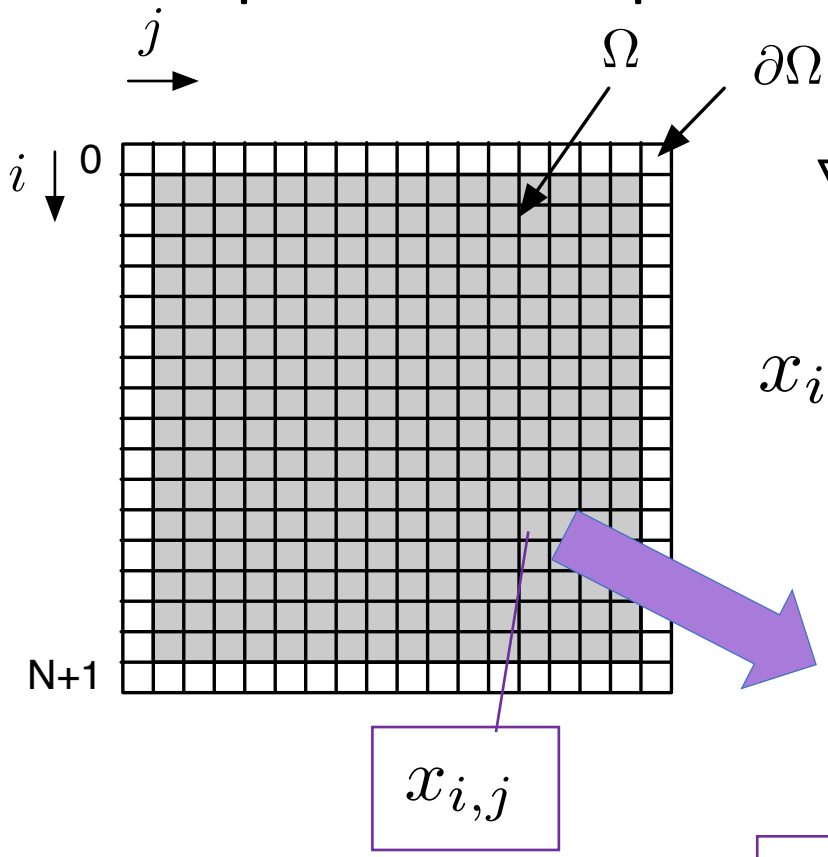
Non-zeros in here due to boundary

$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4$$

The value of each point on the grid

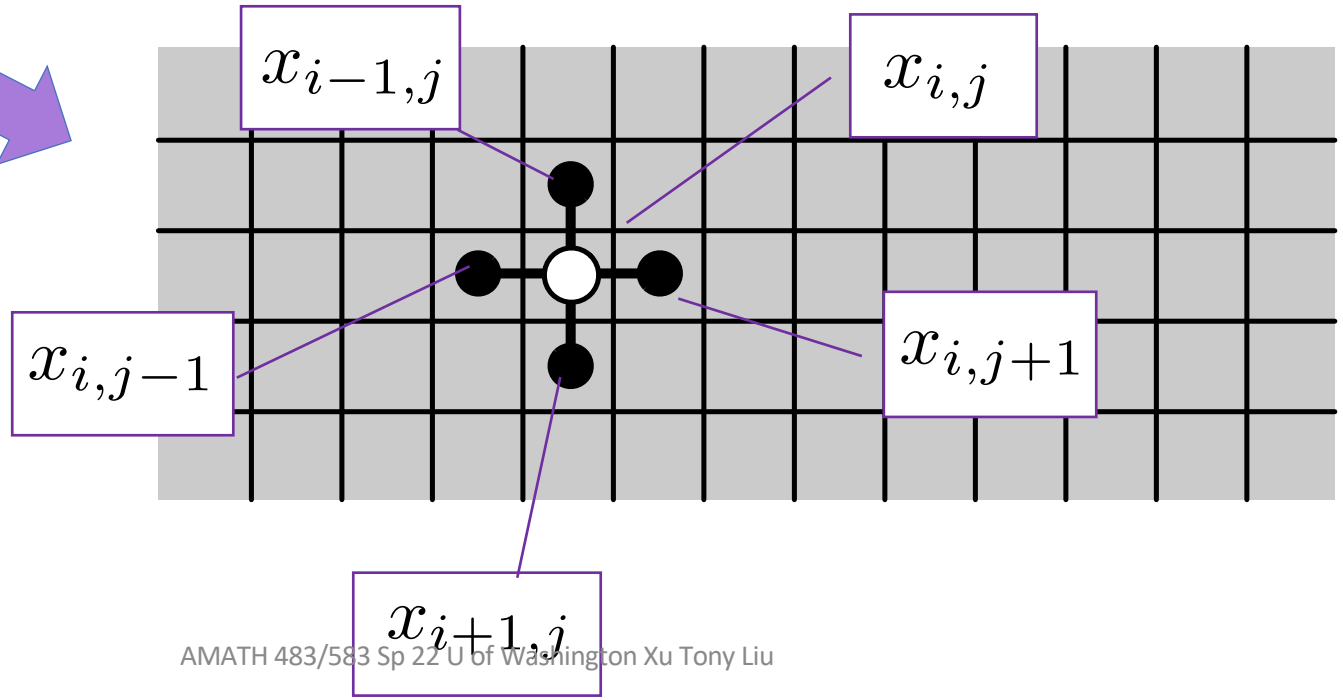
The average of its neighbors

Laplace's Equation on a Regular Grid

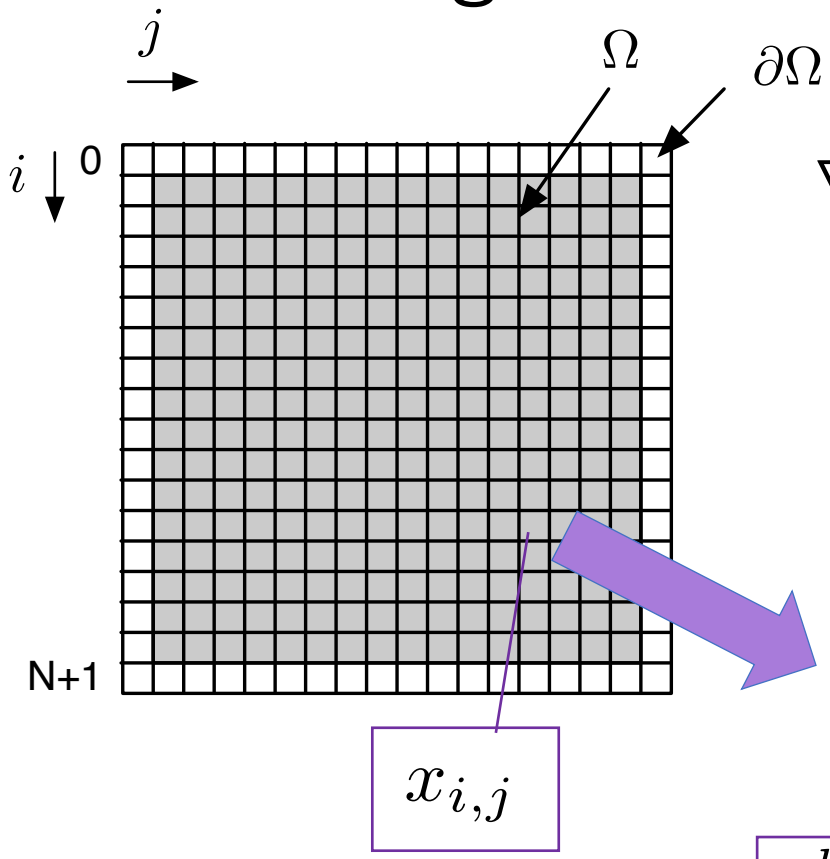


$$\begin{aligned} \nabla^2\phi &= 0 \quad \text{on } \Omega \\ \phi &= f \quad \text{on } \partial\Omega \end{aligned}$$

$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4$$



Iterating for a solution

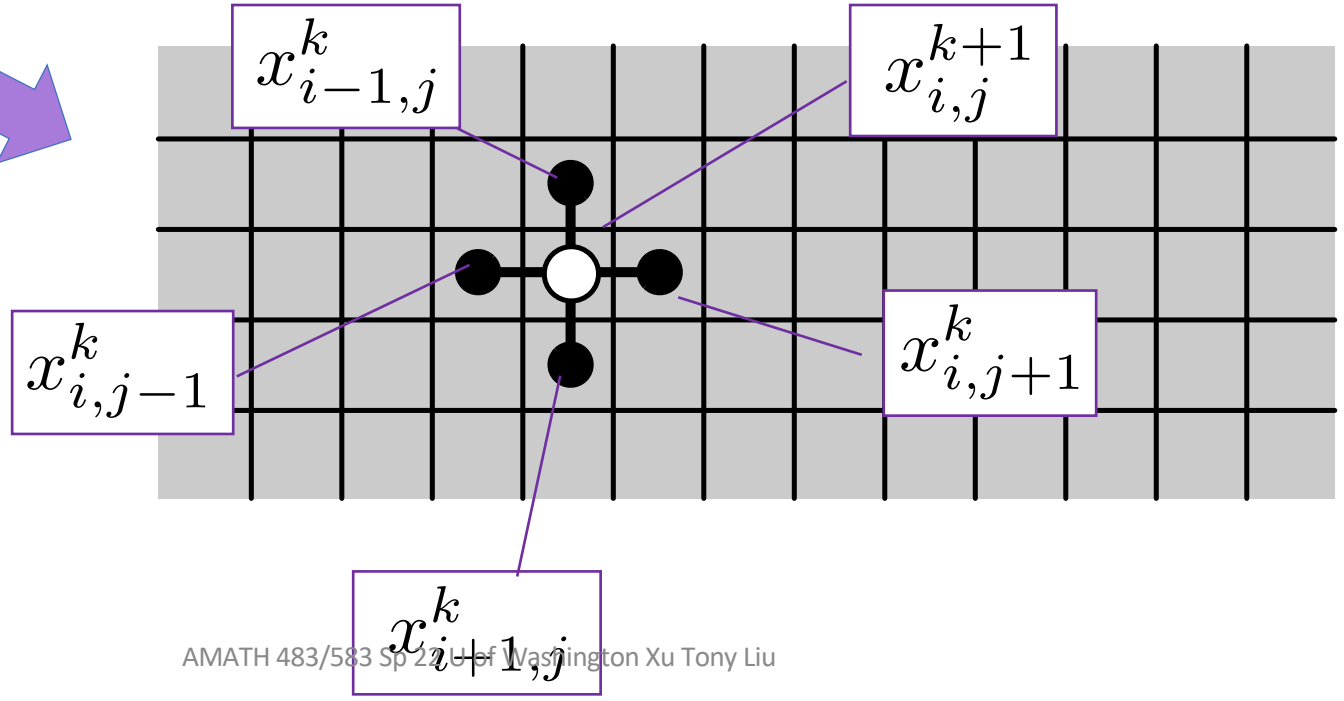


$$\begin{aligned} \nabla^2\phi &= 0 \quad \text{on } \Omega \\ \phi &= f \quad \text{on } \partial\Omega \end{aligned}$$

Approximation at iteration $k+1$

Average of approximation at iteration k

$$x_{i,j}^{k+1} = (x_{i-1,j}^{k} + x_{i+1,j}^{k} + x_{i,j-1}^{k} + x_{i,j+1}^{k})/4$$



Iterating for a solution

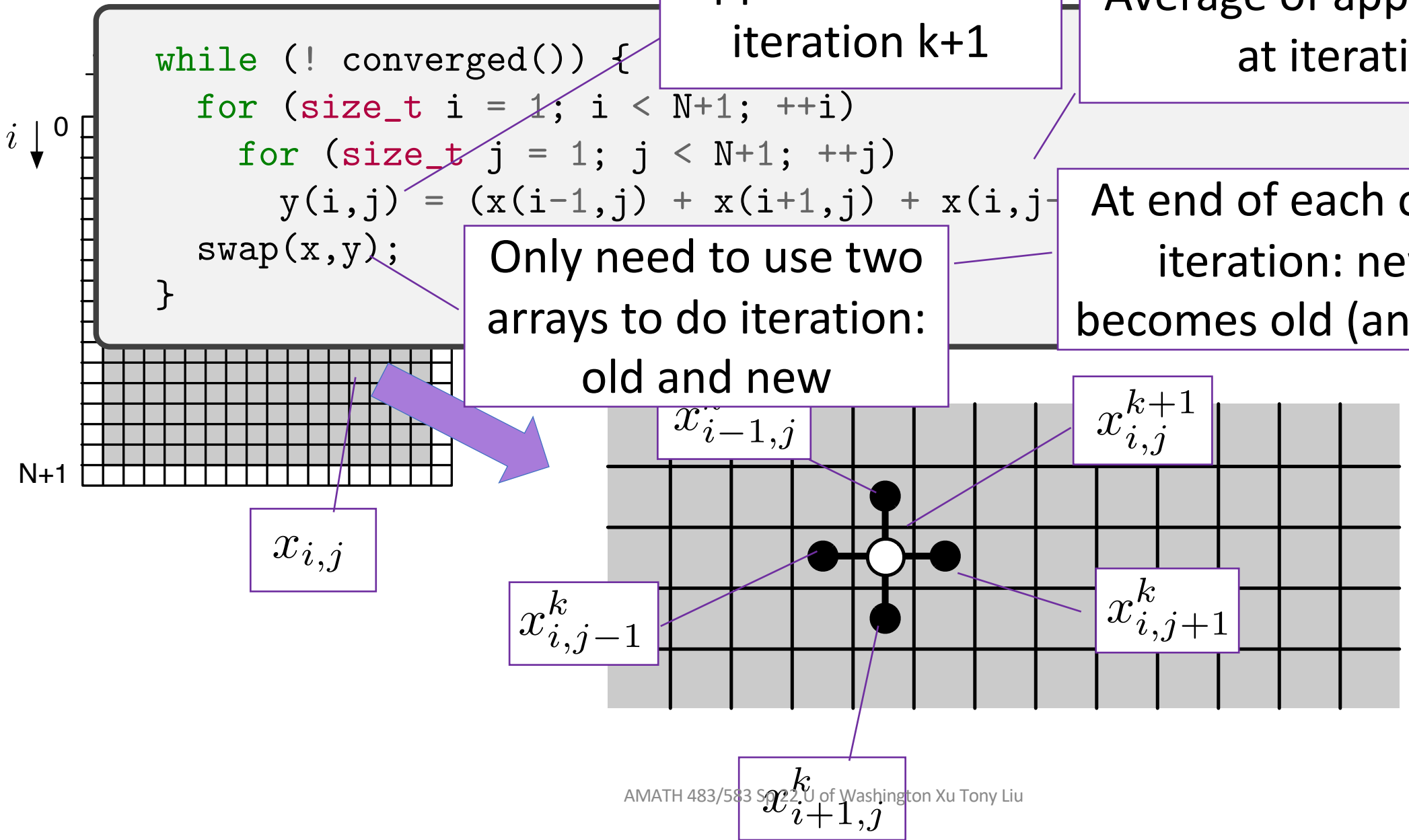
```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-
  swap(x,y);
}
```

Approximation at iteration k+1

Average of approximation at iteration k

At end of each outer iteration: new becomes old (and v.v.)

Only need to use two arrays to do iteration: old and new



class Grid

```
class Grid {

public:
  explicit Grid(size_t x, size_t y)                    :
      xPoints(x+2), yPoints(y+2), arrayData(xPoints*yPoints) {}

        double &operator()(size_t i, size_t j)
            { return arrayData[i*yPoints + j]; }
  const double &operator()(size_t i, size_t j) const
            { return arrayData[i*yPoints + j]; }

  size_t numX() const { return xPoints; }
  size_t numY() const { return yPoints; }

private:
  size_t xPoints, yPoints;
  std::vector<double> arrayData;
};
```

Grid is a 2D array

Constructor

Accessor

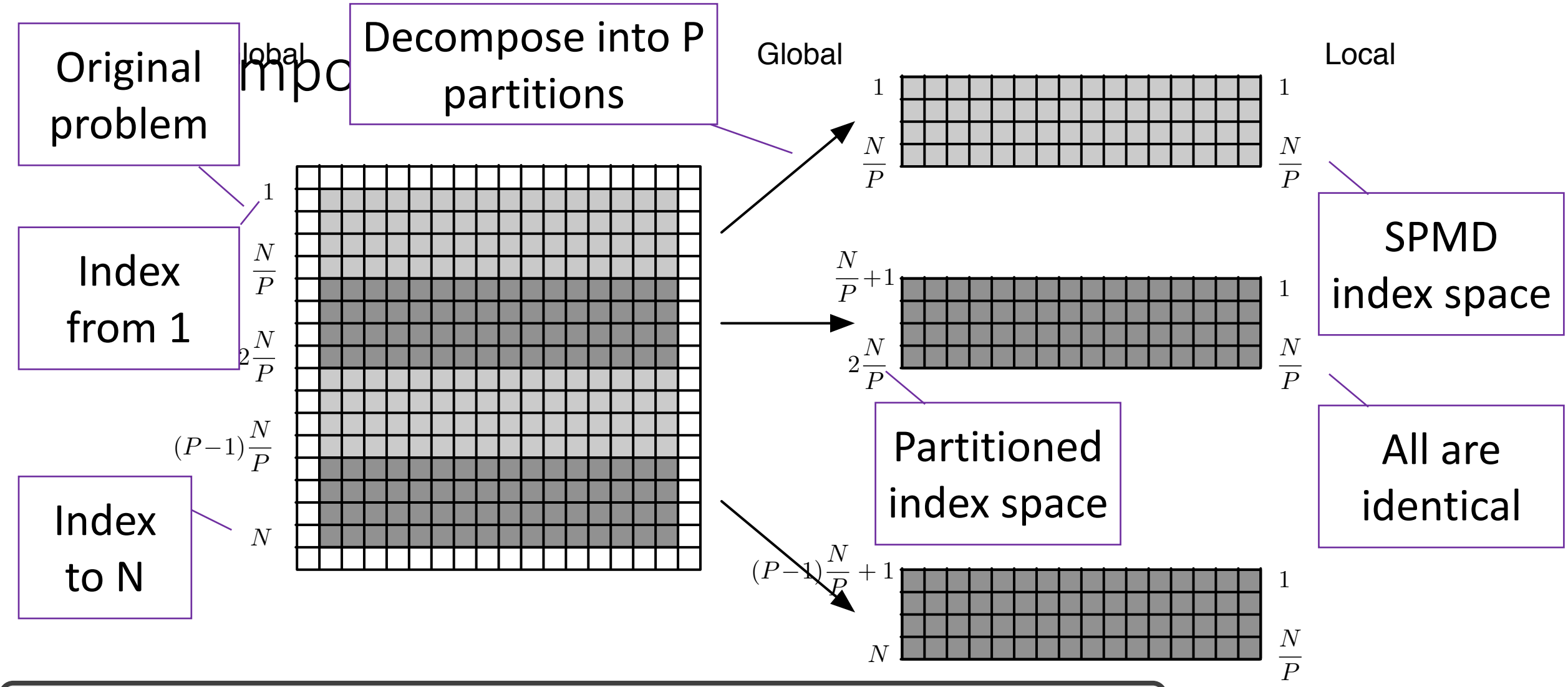
Storage

Main Sequential Jacobi Sweep

```
double jacobiStep(const Grid& x, Grid& y) {
  assert(x.numX() == y.numX() && x.numY() == y.numY());
  double rnorm = 0.0;

  for (size_t i = 1; i < x.numX()-1; ++i) {
    for (size_t j = 1; j < x.numY()-1; ++j) {
      y(i, j) = (x(i-1, j) + x(i+1, j) + x(i, j-1) + x(i, j+1))/4.0;
      rnorm += (y(i, j) - x(i, j)) * (y(i, j) - x(i, j));
    }
  }

  return std::sqrt(rnorm);
}
```

Sequential Jacobi Solver

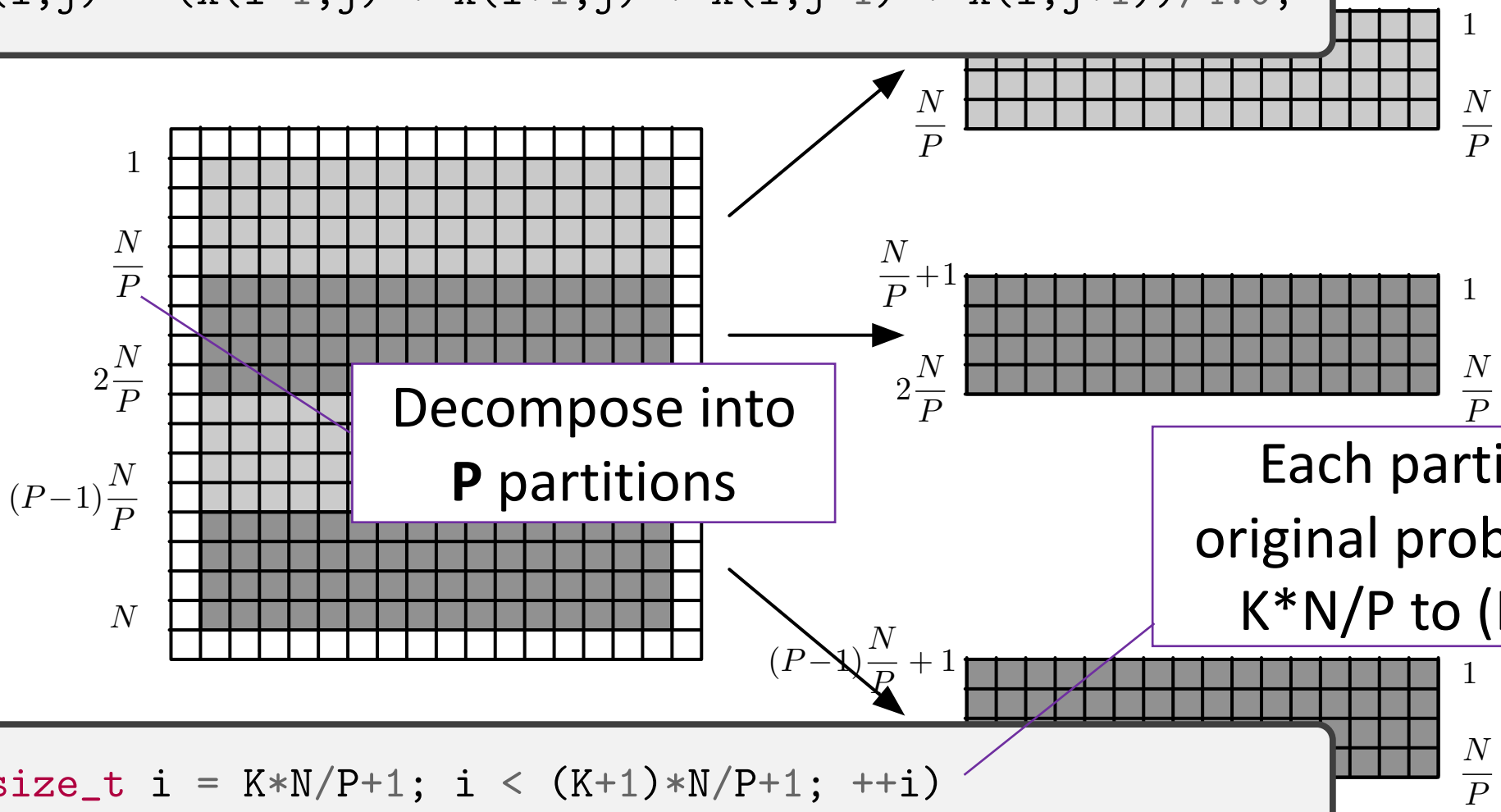
```
int jacobi(Grid& X0, Grid& X1, size_t max_iters, double tol) {
  for (size_t iter = 0; iter < max_iters; ++iter) {
    double rnorm = jacobiStep(X0, X1);
    if (rnorm < tol) return 0;
    swap(X0, X1);
  }
  return -1;
}
```




```
for (size_t i = 1; i < N+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

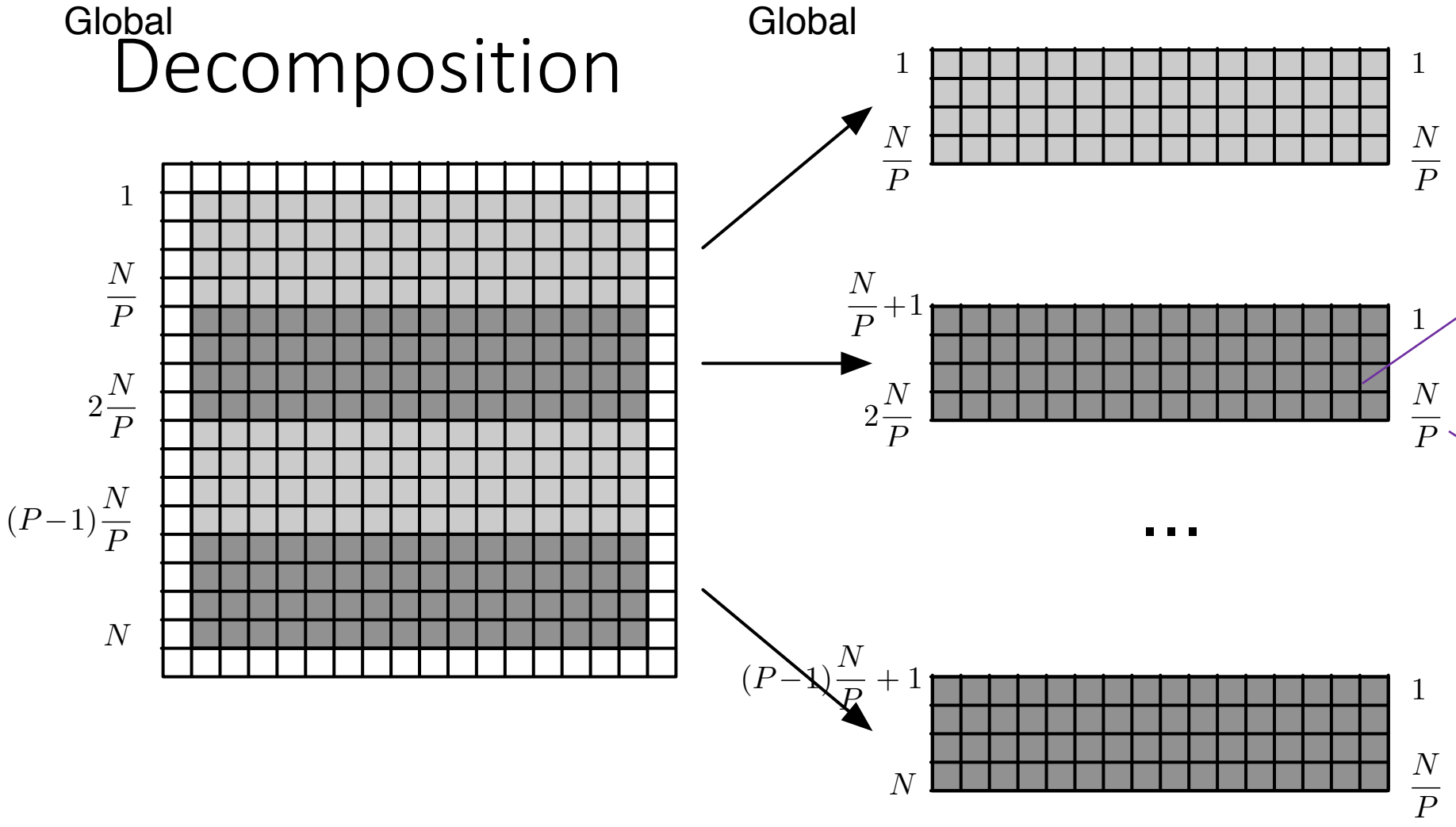
```
for (size_t i = 1; i < N+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Local




```
for (size_t i = K*N/P+1; i < (K+1)*N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition



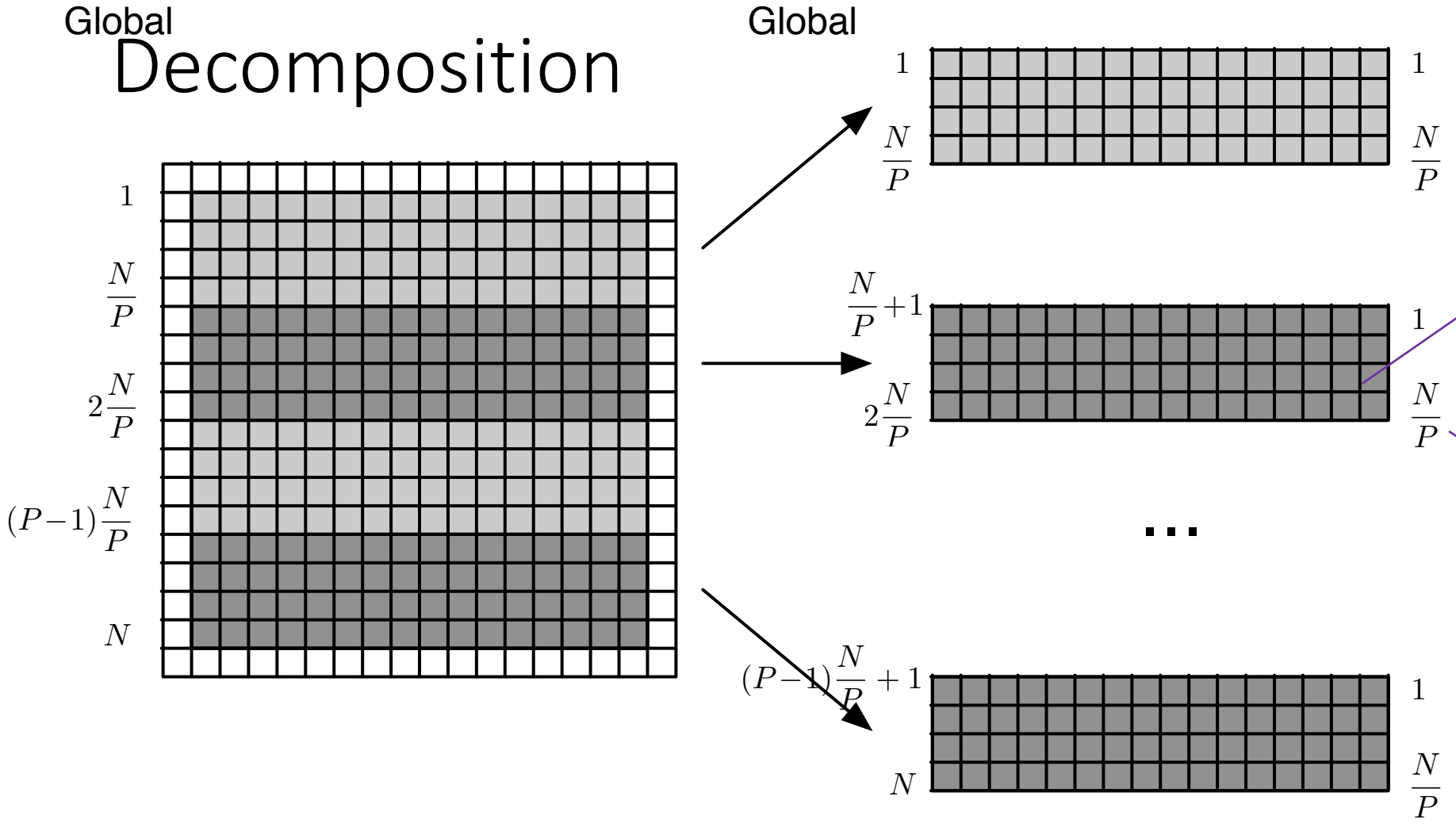
We aren't storing entire array on each node

We are storing only N/P rows

We need to iterate 1 to N

```
for (size_t i = K*N/P+1; i < (K+1)*N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition



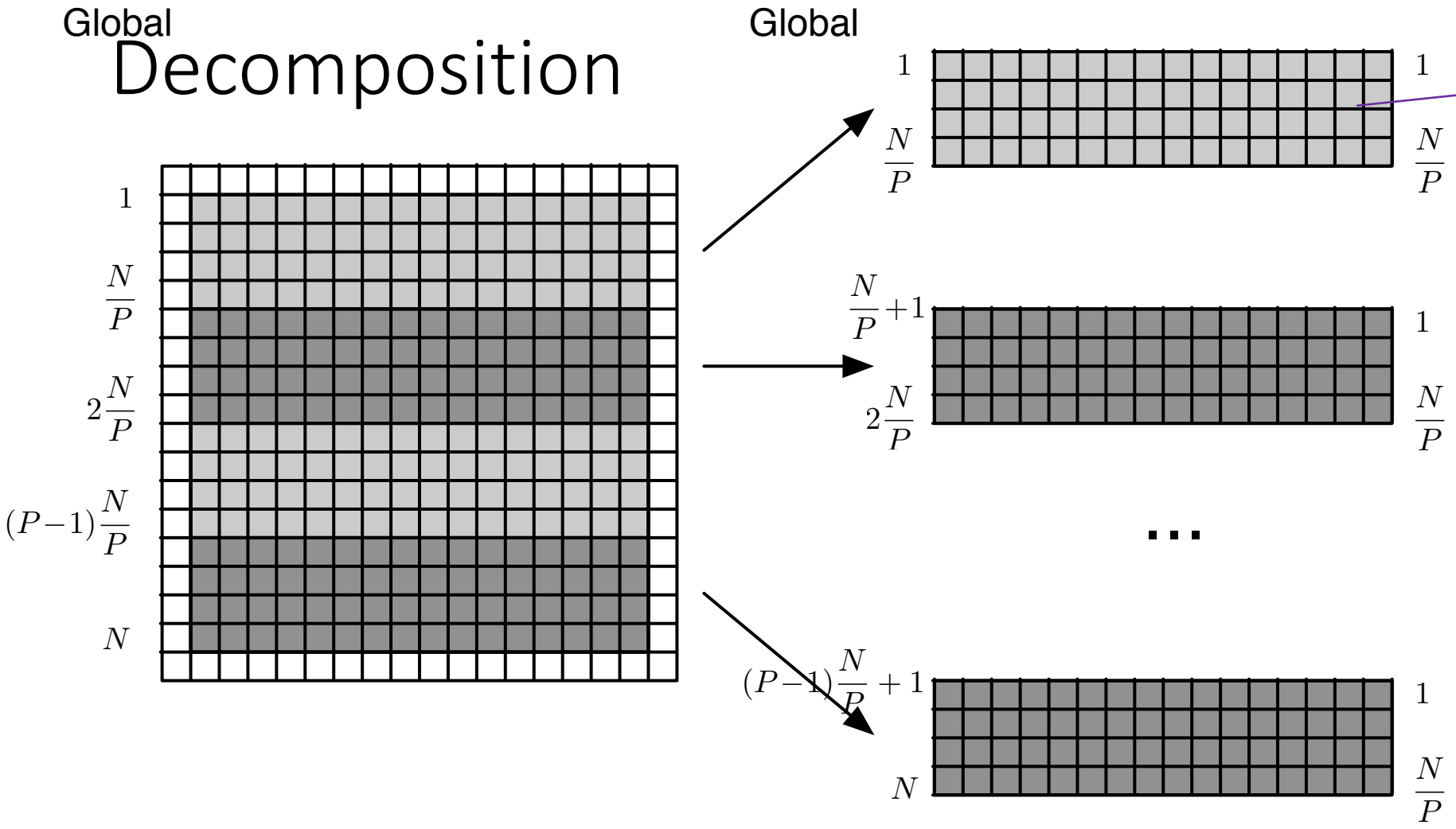
We aren't storing entire array on each node

We are storing only N/P rows

We need to iterate 1 to N

```
for (size_t i = K*N/P+1; i < (K+1)*N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition

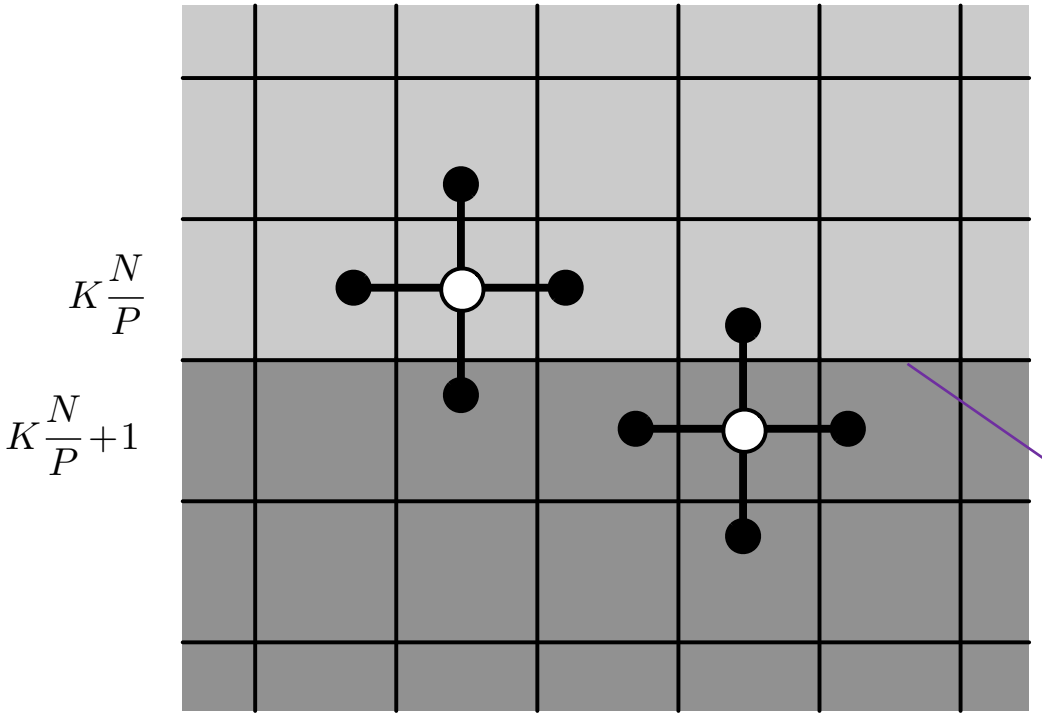


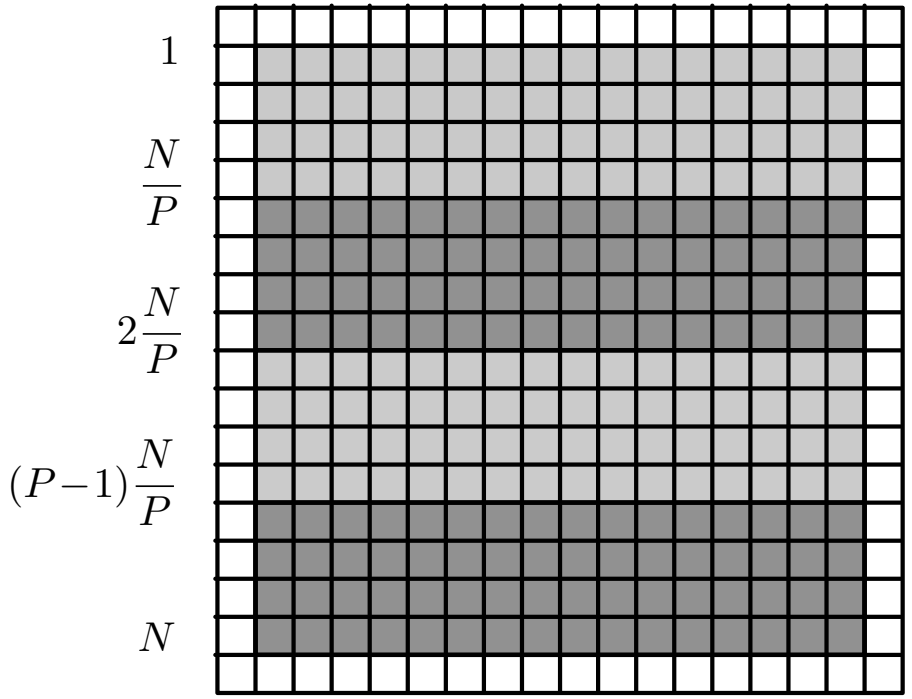
Any obvious problems?

We need to iterate 1 to N

```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition



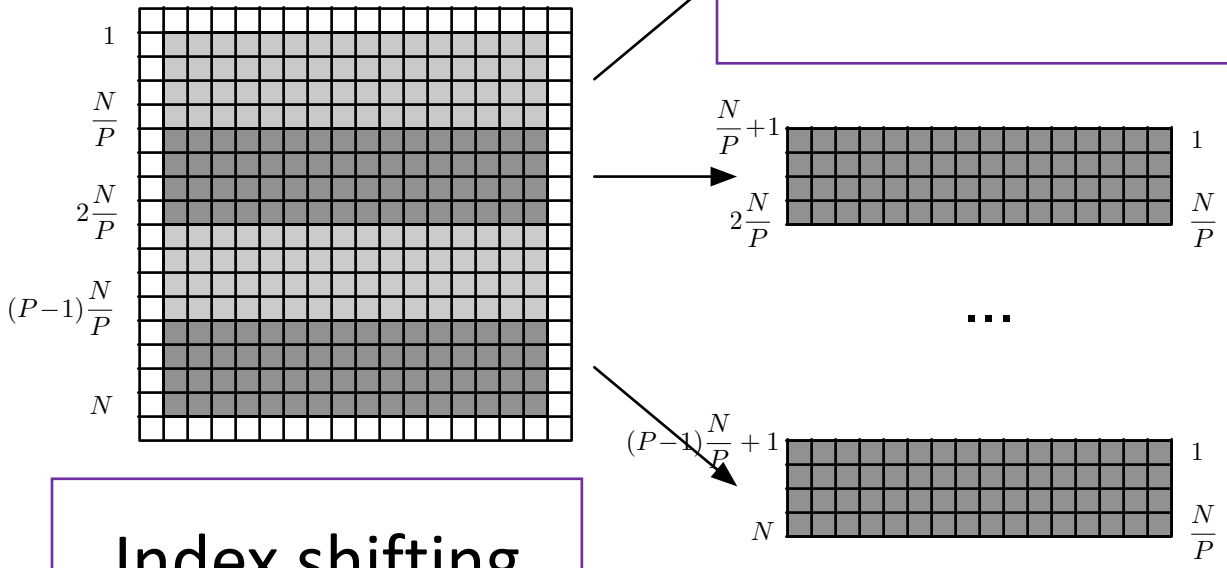


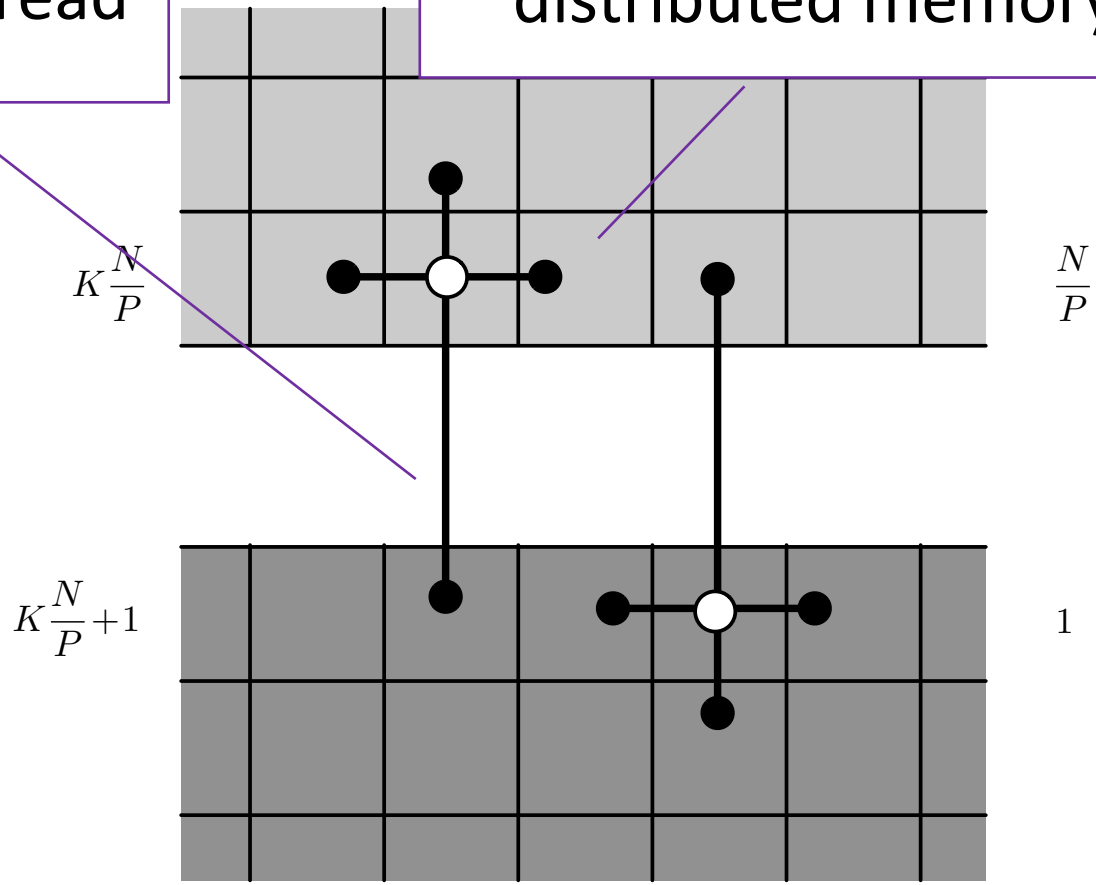
Data dependencies for stencil crosses partition boundary in original problem

Decompositi

This is not a valid read

Which is a problem in distributed memory





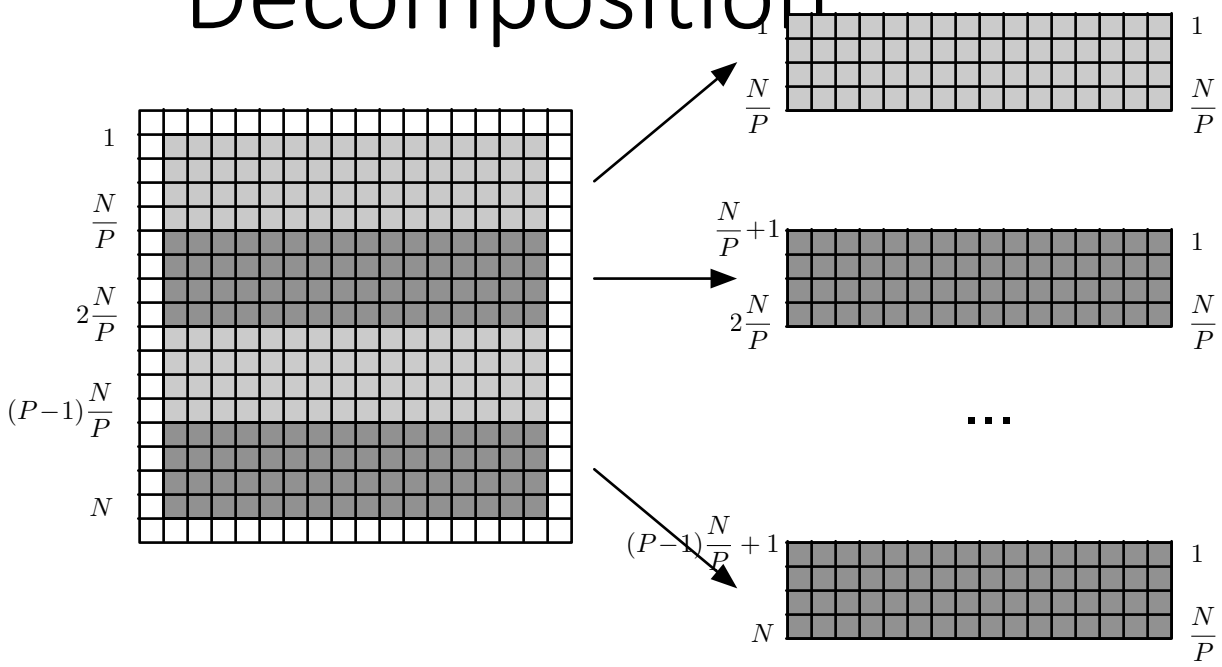
Index shifting doesn't help

This is just a program

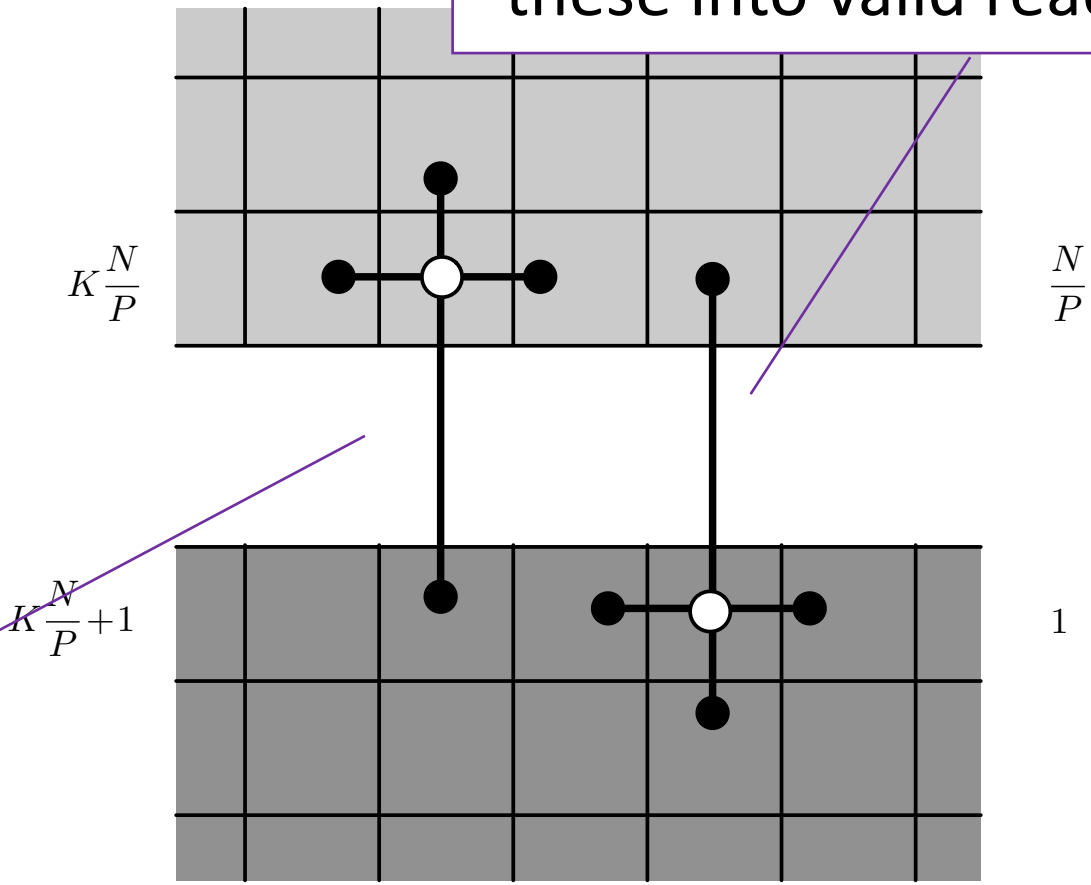
```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4
```

It reads/write local data, just like any other

Decomposition



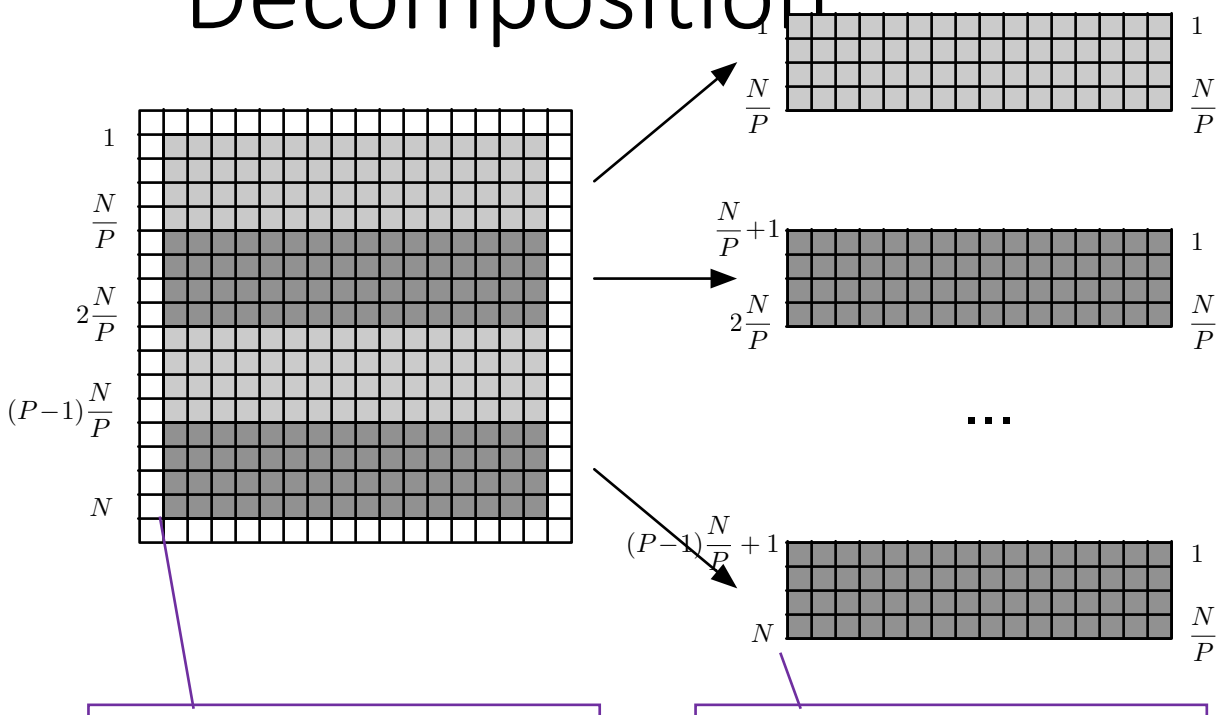
We need to make these into valid reads



And preserve the "as-if" property

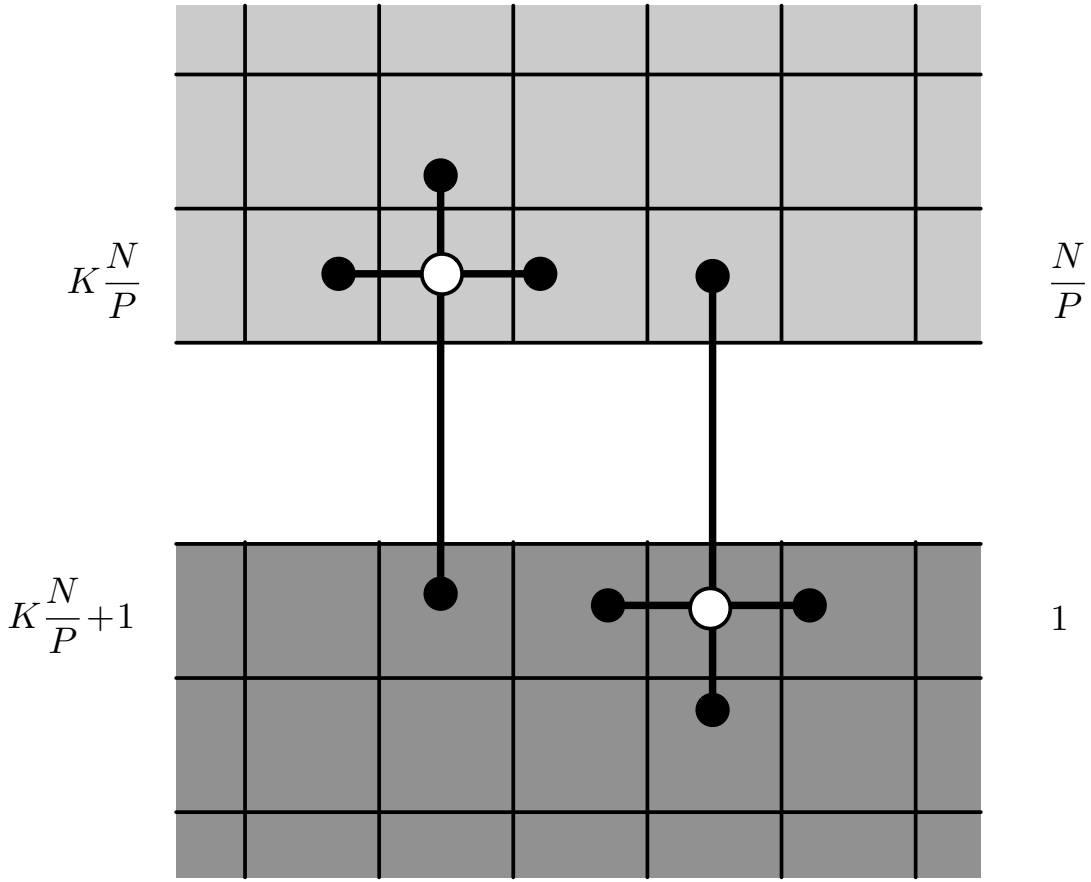
```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition



Where did the boundary go

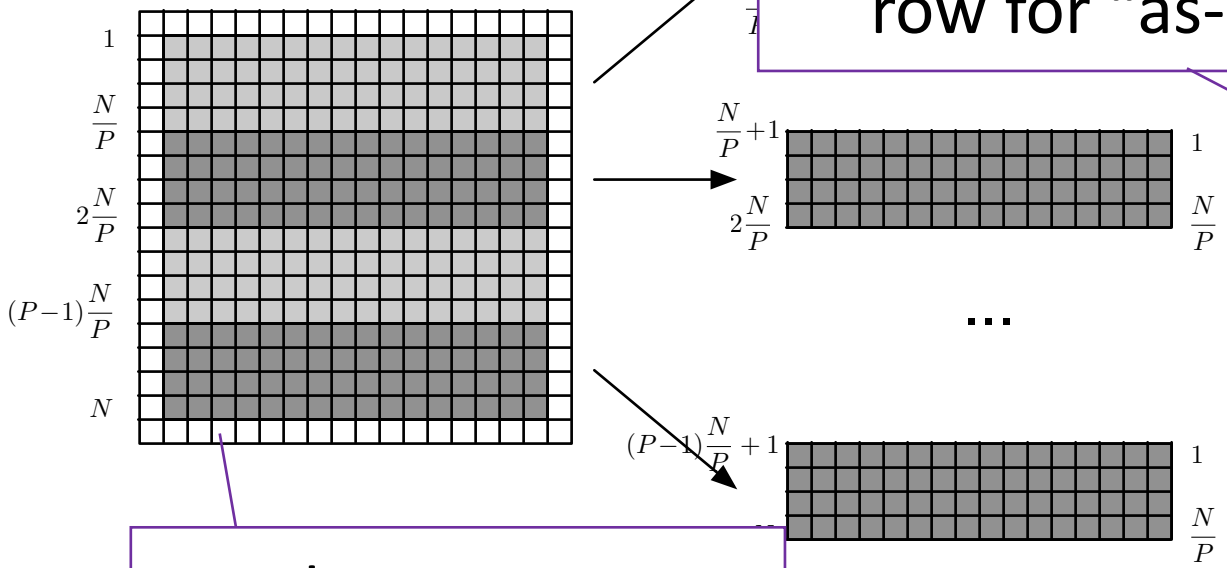
When we partitioned?

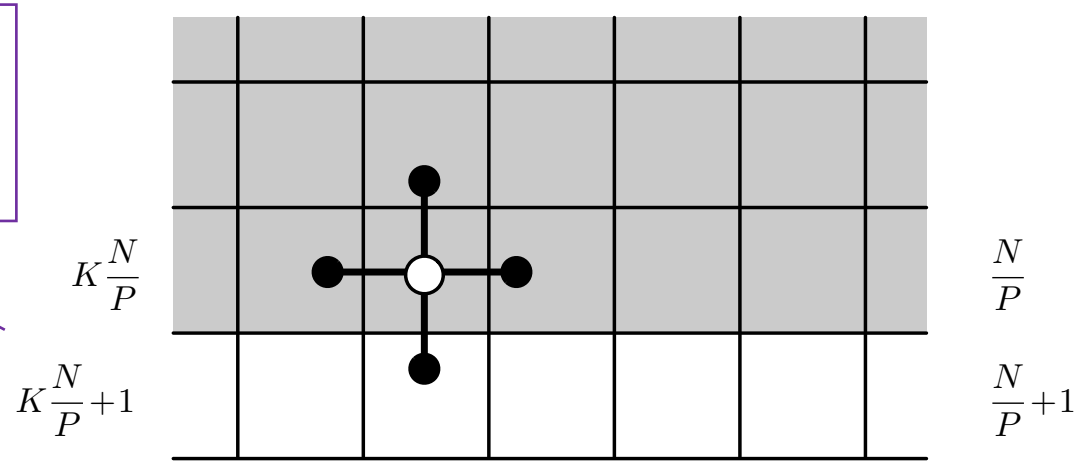


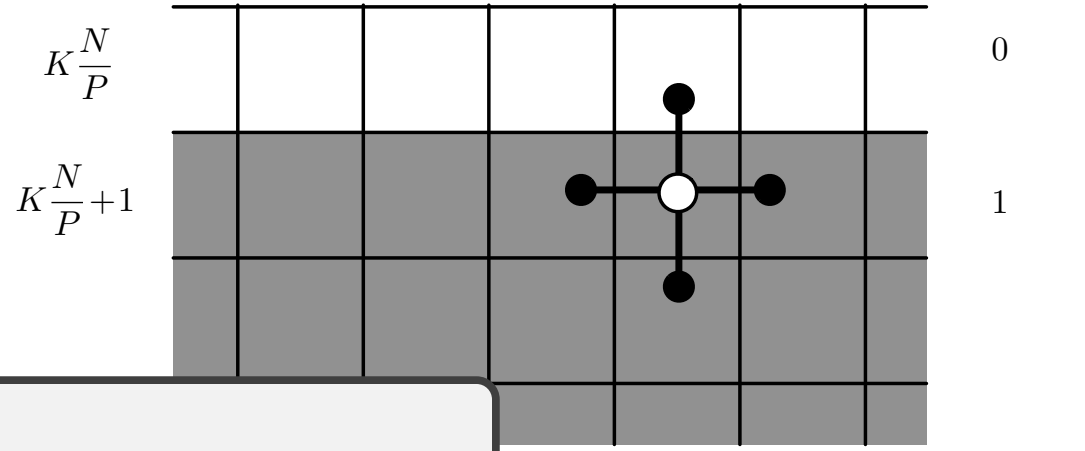

```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decompositic

We need an extra row for “as-if”



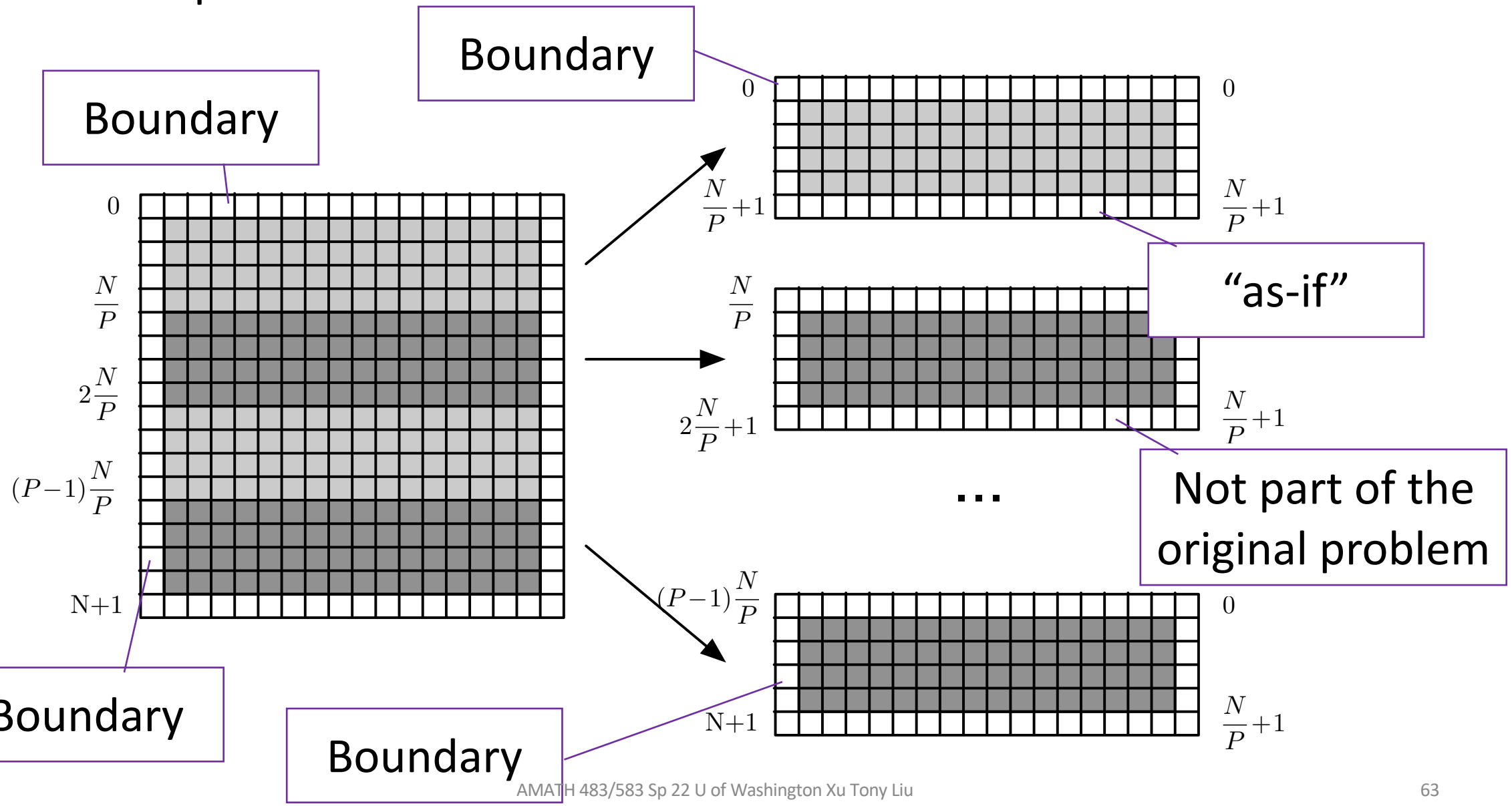




And an extra row (on some nodes) for the boundary

```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition



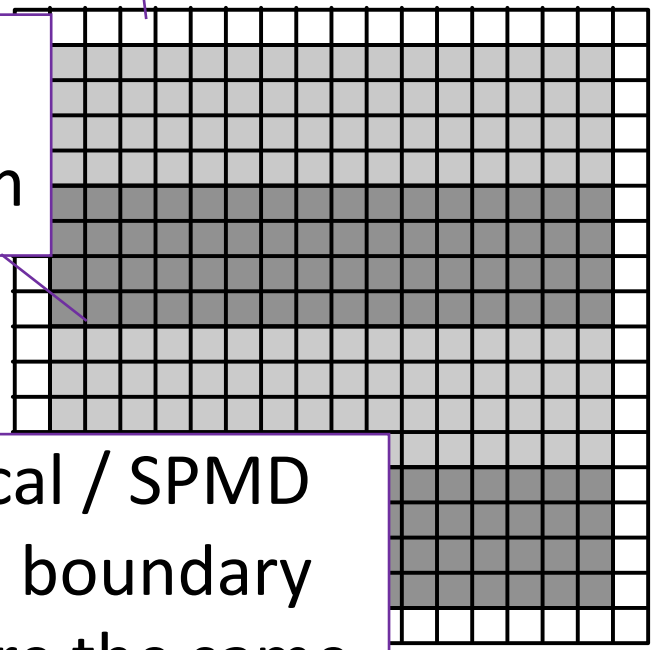
Decomposition

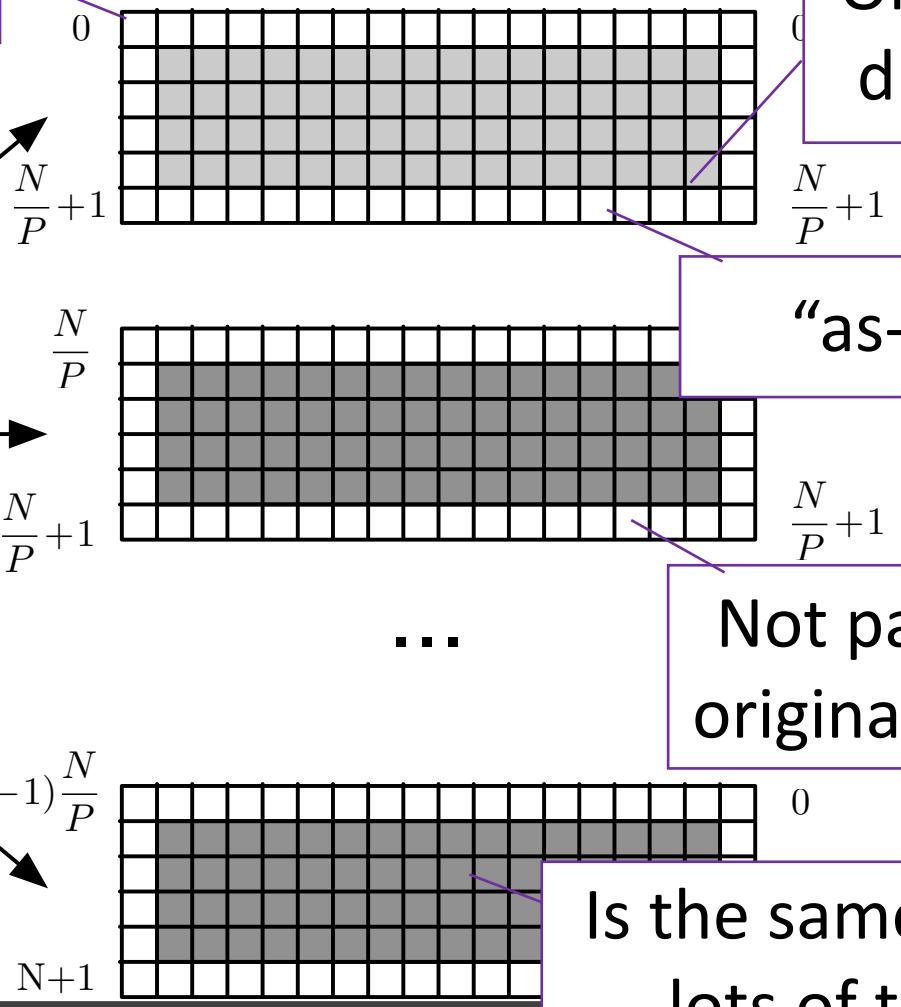
Boundary

Boundary

One crucial difference

So solving this problem





"as-if"

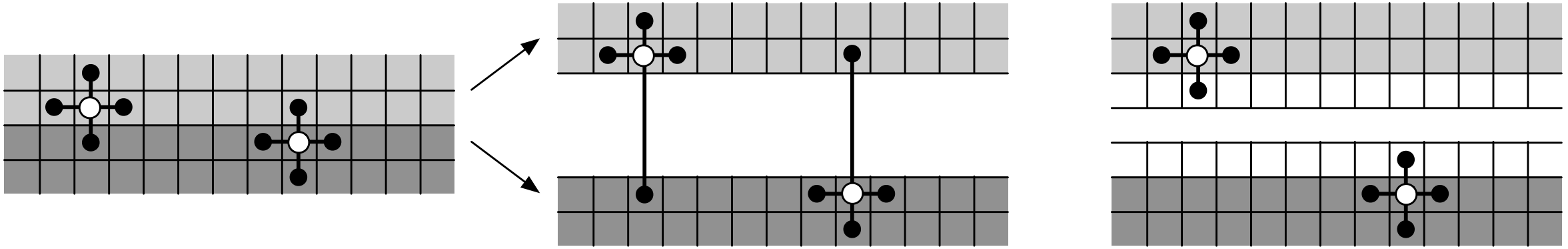
Not part of the original problem

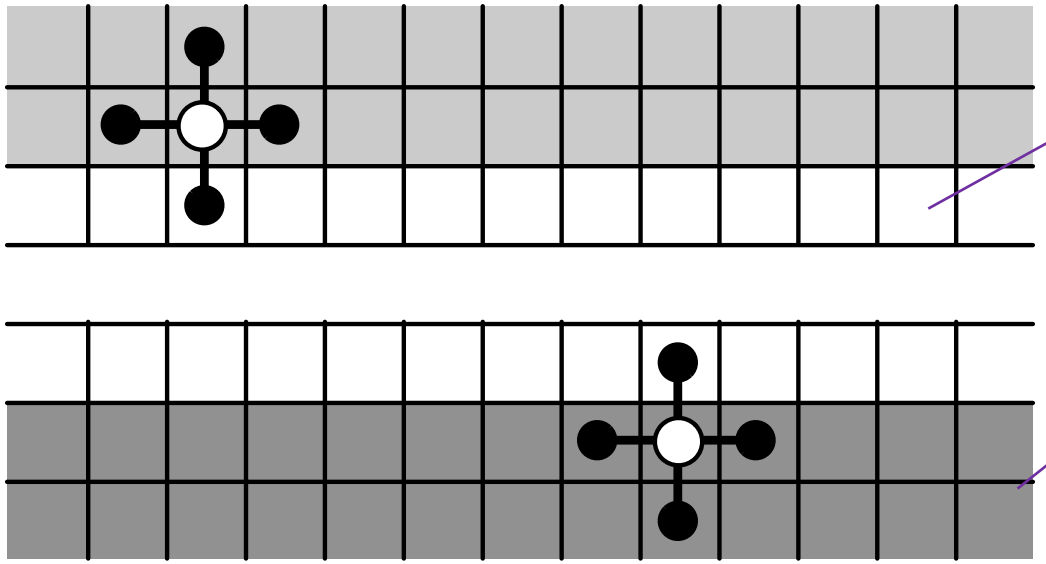
To the local / SPMD code, the boundary and as-if are the same

Is the same as solving lots of the same problem but smaller

```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

As-If

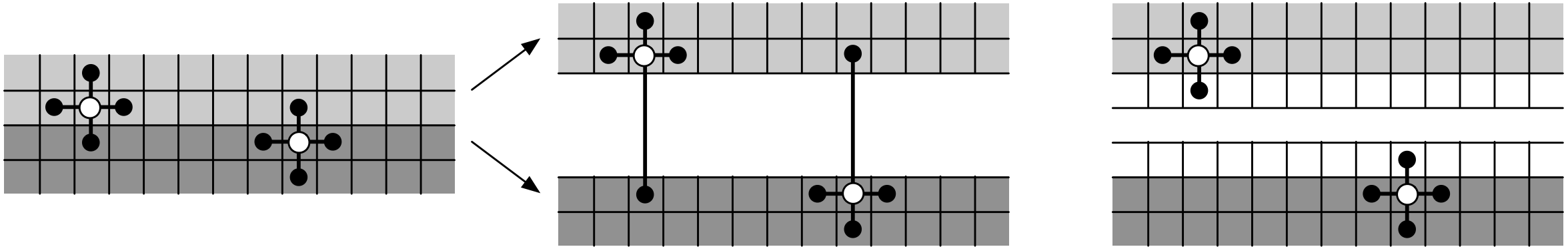


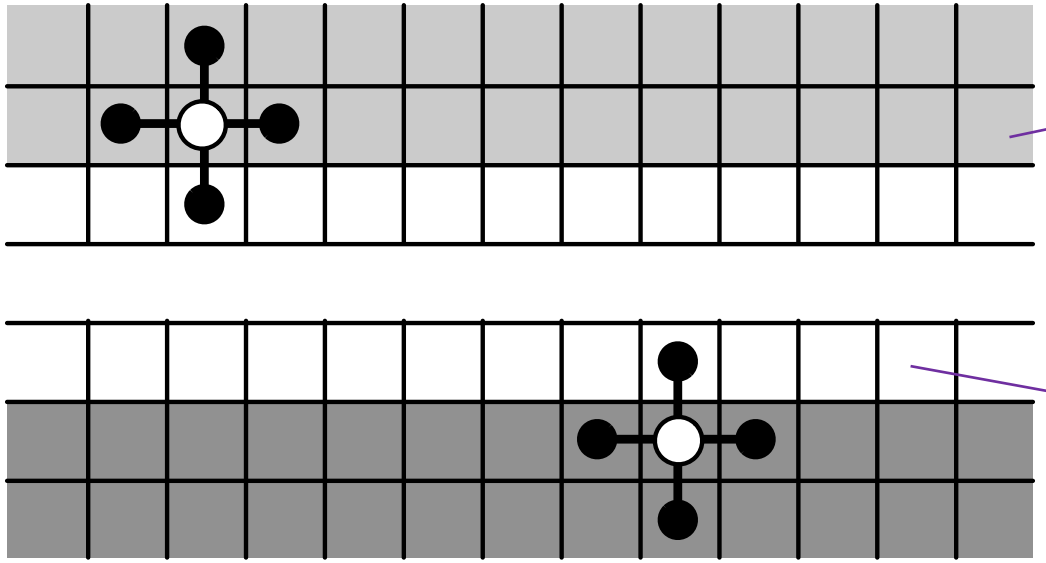


This row needs to be “as-if”

it were this row

As-If



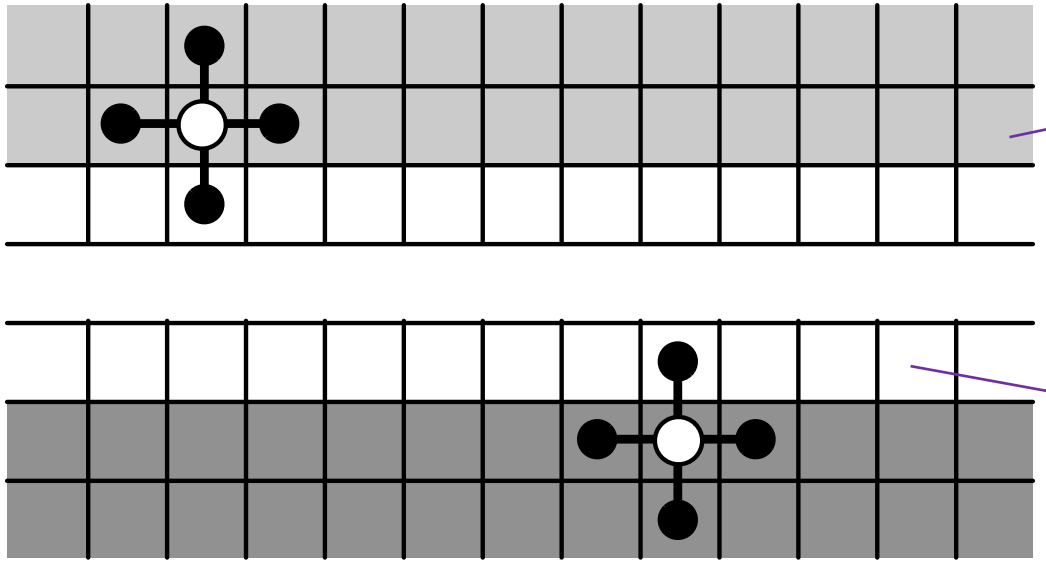


it were this row

And this row needs to be “as-if”

As-If

```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```



it were this row

This is the computation

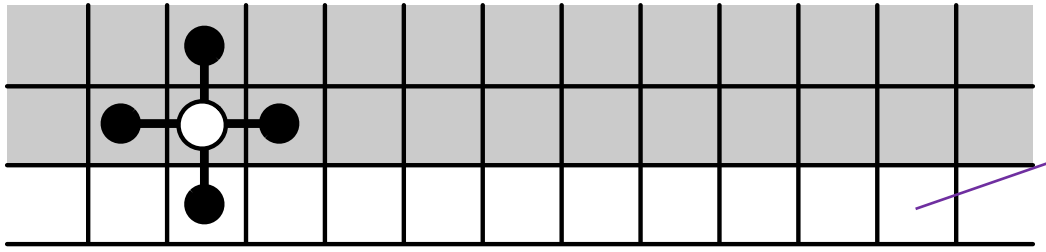
And this row needs to be “as-if”

As-If

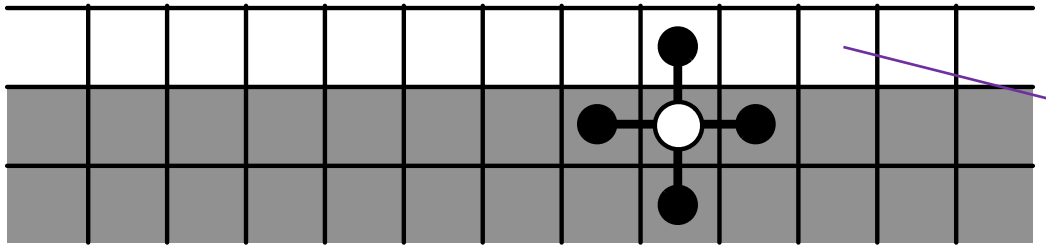
```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

This is the computation

Only when it is read



Note these are not changed during an iteration



Does this row ***always*** have to have the same value as the other row?

As-If

```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
  swap(x,y);
}
```

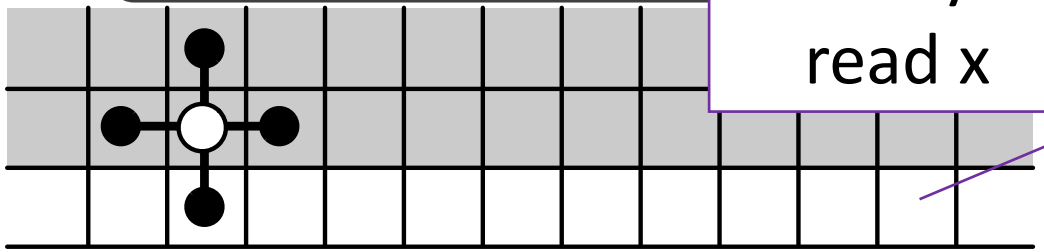
This is the entire program

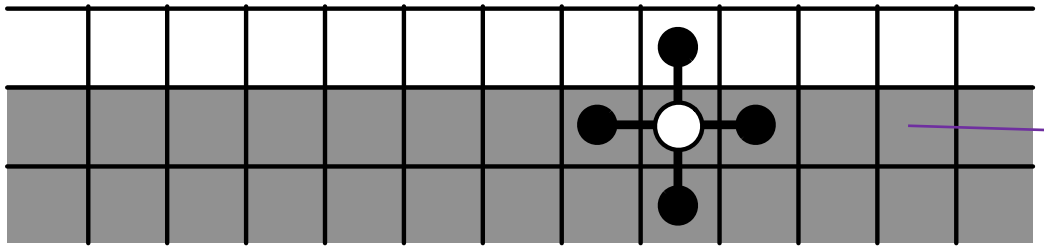
Always write y

Always read x

Not changed during an iteration

Rows need to be as-if only during iteration





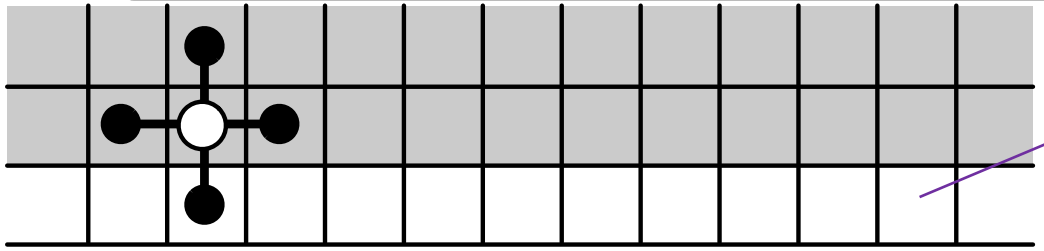
This changes only on every outer iteration (on the swap())

As-If

Here is where we need to make as-if true

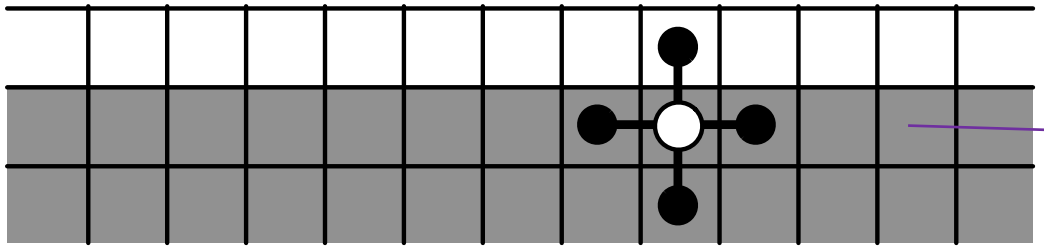
This is the entire program

```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
  swap(x,y);
}
```



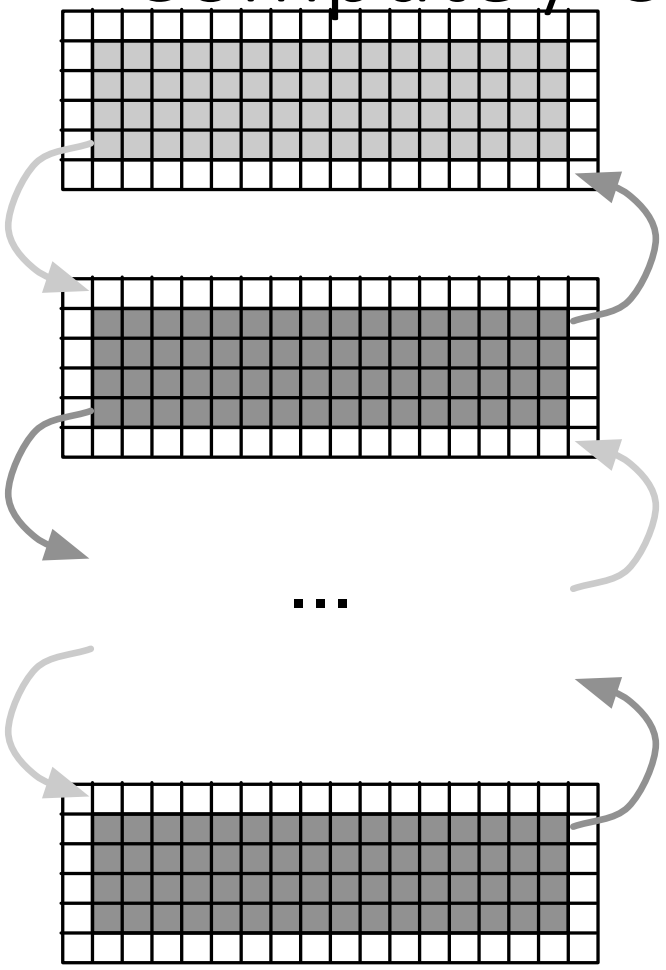
Not changed during an iteration

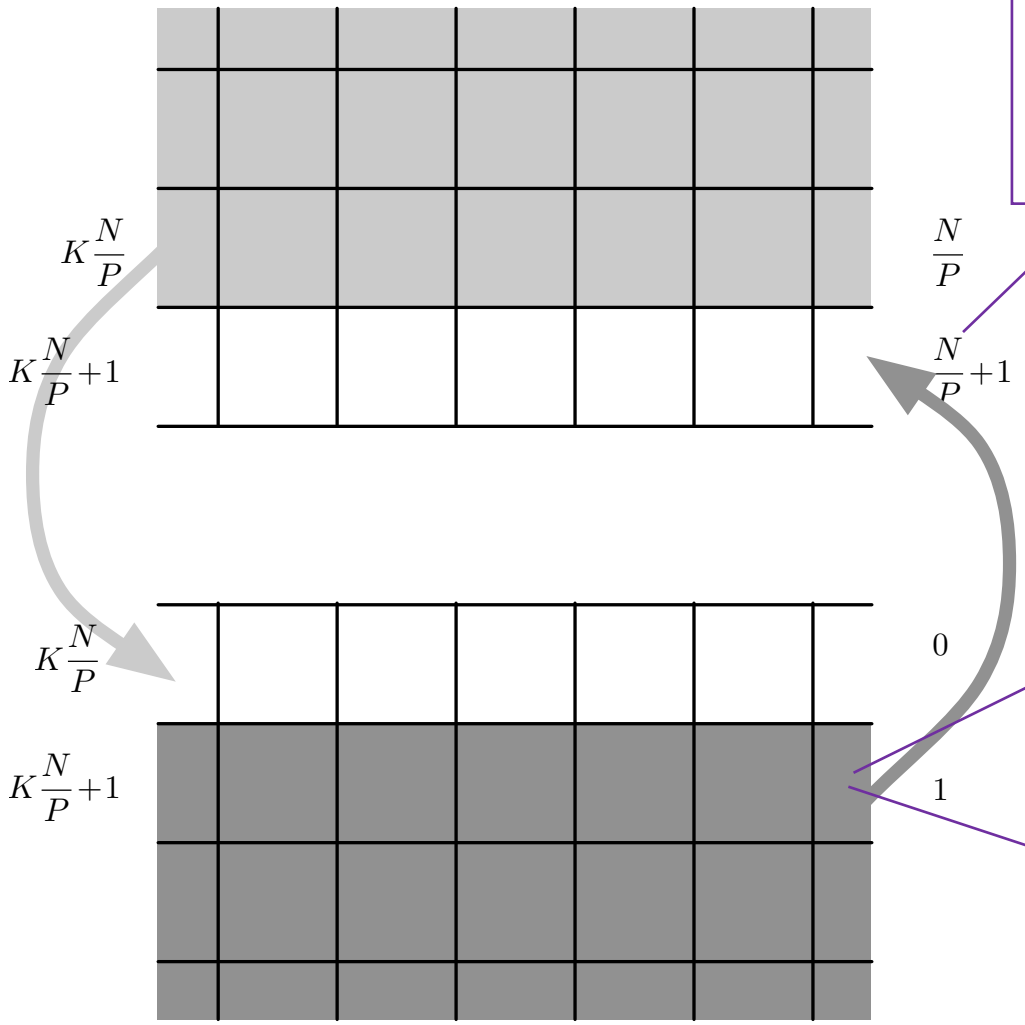
Rows need to be as-if only during iteration



This changes only on every outer iteration (on the swap())

Compute / Communicate





To make as-if, we need to update the boundary cells

With their "as-if" values

Before they are read at the next outer iteration

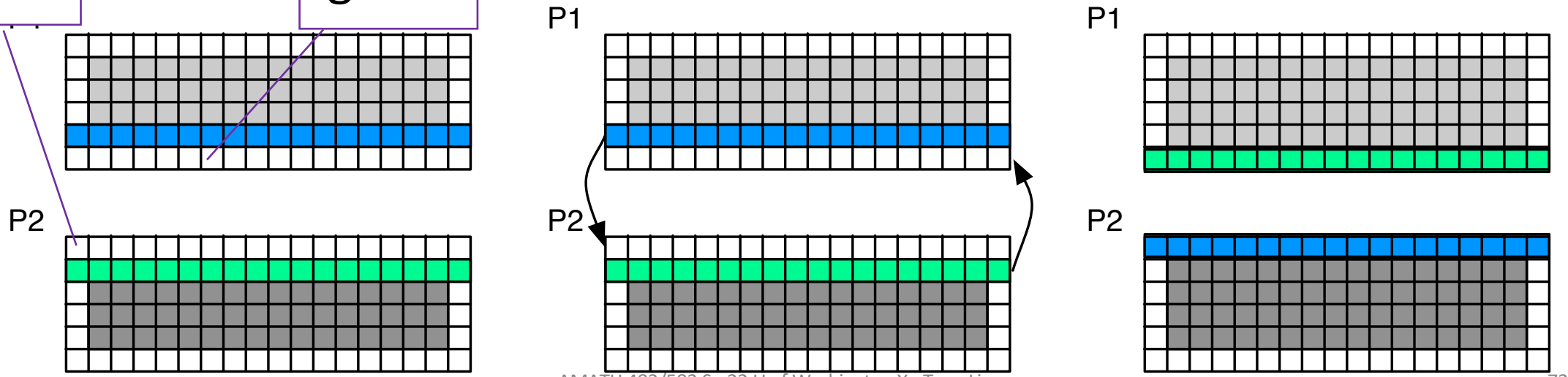
Compute / Communicate

```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
  swap(x,y);
  make_as_if(x); // Communicate ghost cells
}
```

Standard terminology for as-if boundary is "ghost cell"

ghost

ghost



Compute / Communicate

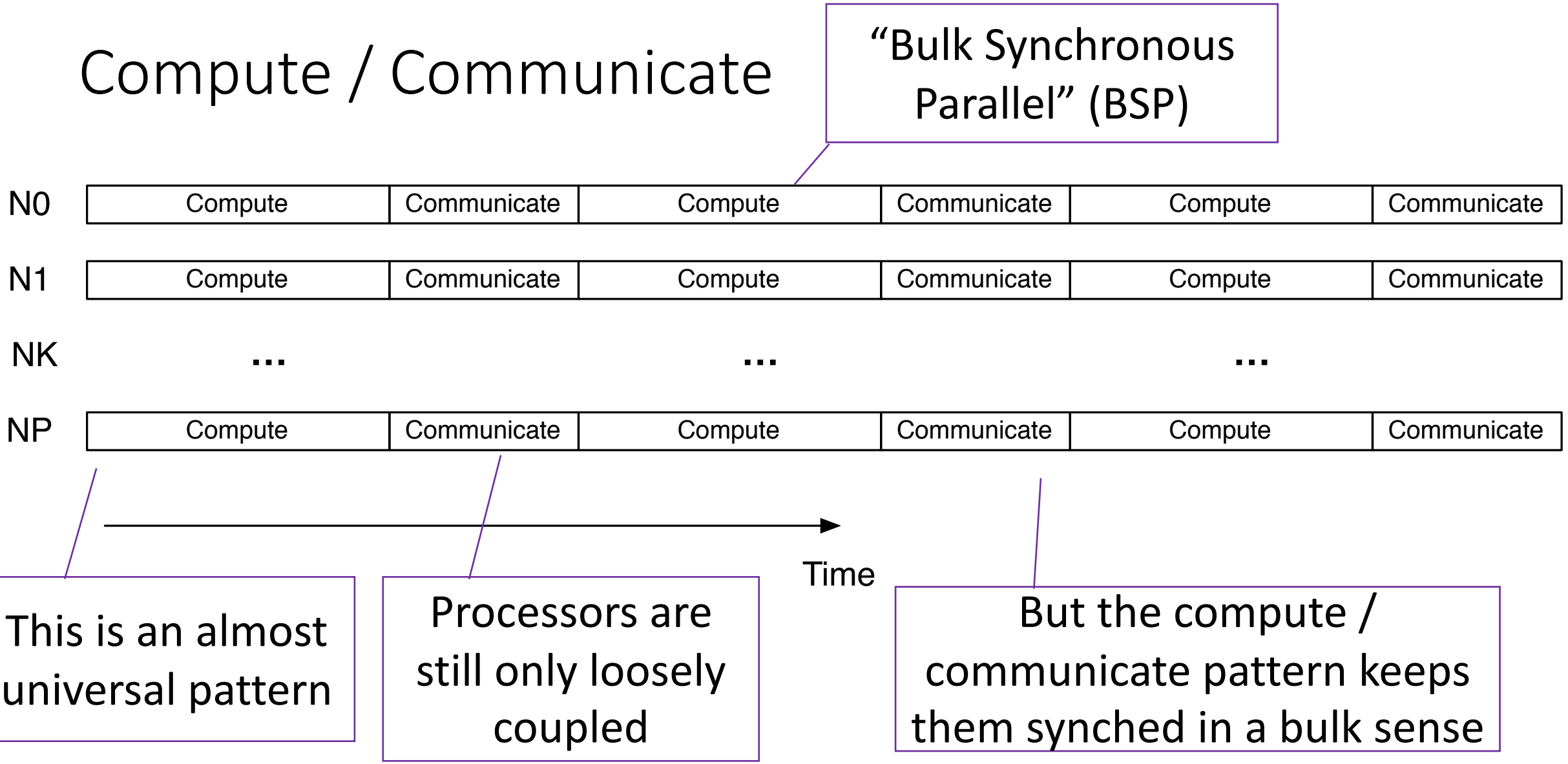
```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
  swap(x,y);
  make_as_if(x); // Communicate ghost cells
}
```

Compute

This is an almost universal pattern

Communicate

Compute / Communicate



Parallel Jacobi Solver

```
int jacobi(Grid& X0, Grid& X1, size_t max_iters, double tol) {
  for (size_t iter = 0; iter < max_iters; ++iter) {
    double rnorm = jacobiStep(X0, X1);
    if (rnorm < tol) return 0;
    swap(X0, X1);
  }
  return -1;
}
```

As-if: This needs to happen on all nodes all_reduce instead of reduce

As-if: Update ghost cells

Parallel Jacobi Step

```
double jacobiStep(const Grid& x, Grid& y) {
  assert(x.numX() == y.numX() && x.numY() == y.numY());
  double rnorm = 0.0;

  for (size_t i = 1; i < x.numX()-1; ++i) {
    for (size_t j = 1; j < x.numY()-1; ++j) {
      y(i, j) = (x(i-1, j) + x(i+1, j) + x(i, j-1) + x(i, j+1))/4.0;
      rnorm += (y(i, j) - x(i, j)) * (y(i, j) - x(i, j));
    }
  }

  return std::sqrt(rnorm);
}
```

MPI Allreduce

Very expensive!

Just like reduce

All use sendbuf to send, all use recvbuf to recv

But when call is completed, ***ALL*** nodes have reduced value

```
void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf,
    int count, const MPI::Datatype& datatype, const MPI::Op& op)
```

```
void MPI::Comm::Reduce(void *buf, int count, const Datatype& datatype,
↪  const Op& op, int root);
```

Root uses buffer to receive, all others use buffer to send

ONLY Root will have a value reduced from all the others

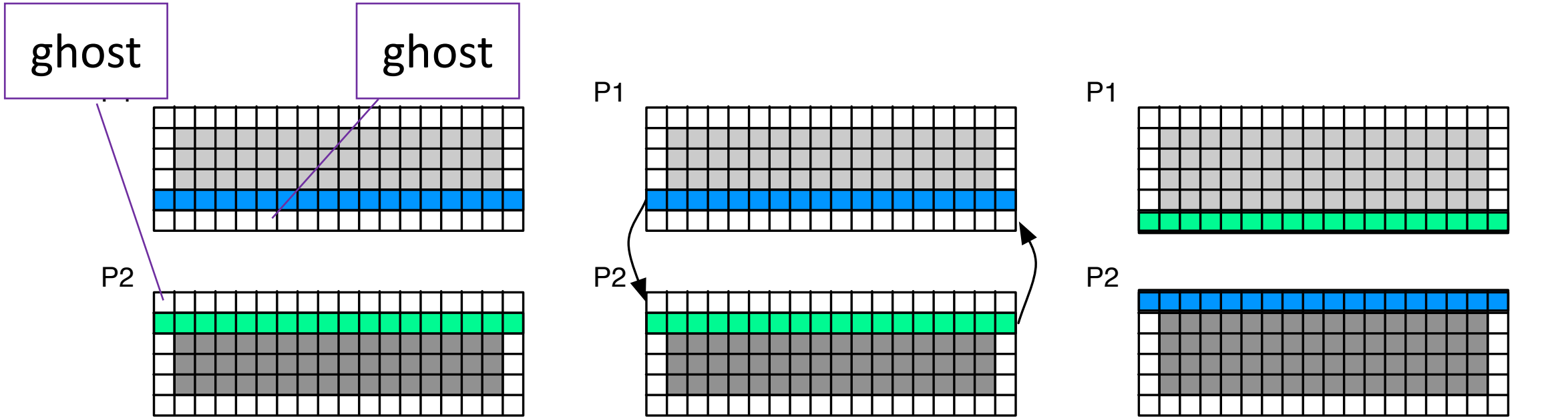
Parallel Jacobi Solver

```
int jacobi(Grid& X0, Grid& X1, size_t max_iters, double tol) {
  for (size_t iter = 0; iter < max_iters; ++iter) {
    double lnorm = jacobiStep(X0, X1);
    double rnorm = 0.0;
    MPI_COMM_WORLD.Allreduce(&rnorm, &lnorm, 1, MPI::DOUBLE, MPI::SUM);
    if (rnorm < tol) return 0;
    swap(X0, X1);
    update_ghosts(X0);
  }
  return -1;
}
```

As-if: This needs to happen on all nodes all_reduce instead of reduce

As-if: Update ghost cells

Updating Ghost Cells

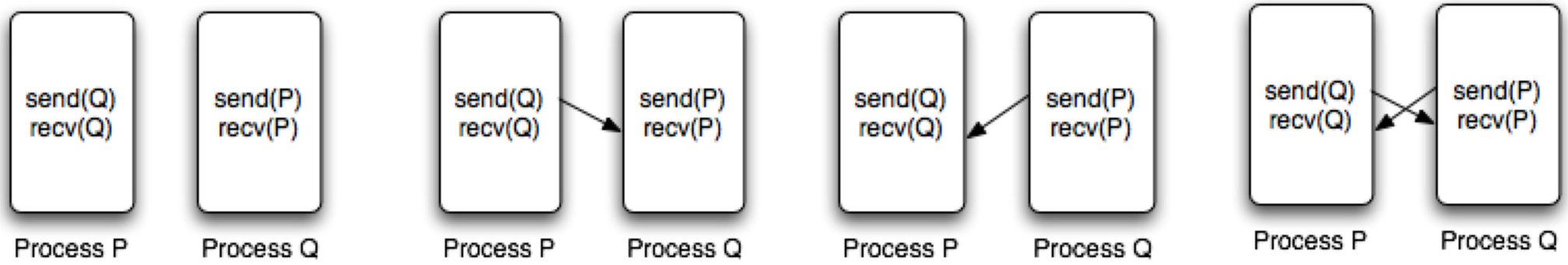



```
MPI_Send( ... );    // to upper neighbor
MPI_Send( ... );    // to lower neighbor
MPI_Send( ... );    // from lower neighbor
MPI_Send( ... );    // from upper neighbor
```

Works?

Updating ghost cells

- What happens with following sequence of communication operations?



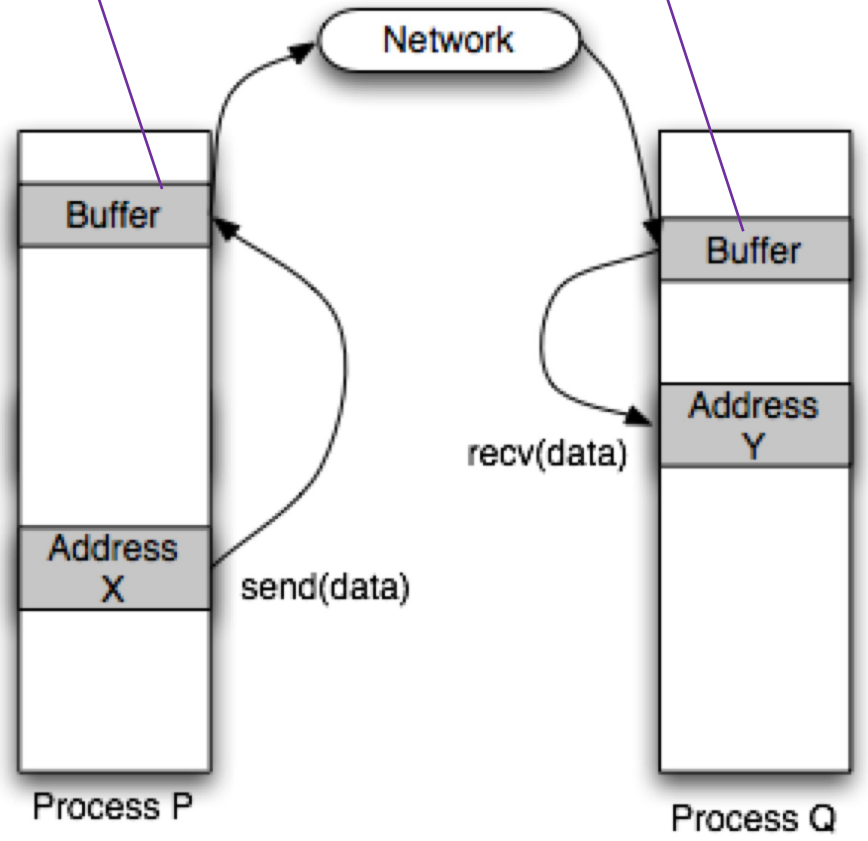
- Have we seen this before?
- Behavior depends on availability (and size) of buffering
  - System dependent
  - MPI implementation (LAM, Open MPI, MPICH) have diagnostics for this

Where does data go when you send it?

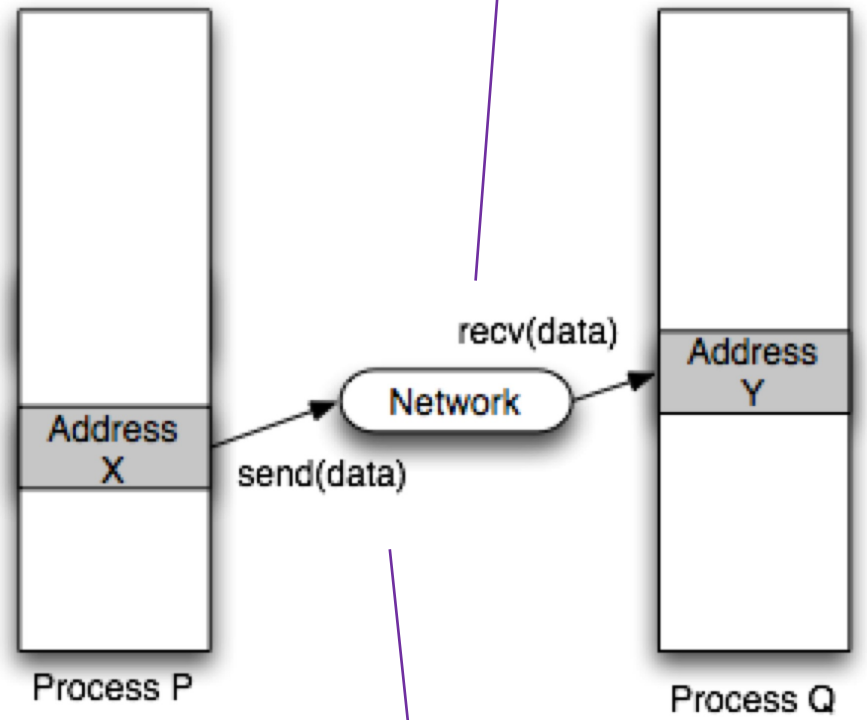
System might buffer the data

System might buffer the data

System might not buffer the data



Good thing or bad thing?



Might buffer some message

Not others

MPI_Send

```
#include <mpi.h>
void Comm::Send(const void* buf, int count, const Datatype& datatype,
 ↪  int dest, int tag) const
```

- MPI_Send is sometimes called a “blocking send”
- Semantics (from the standard): Send MPI_Send returns, it is safe to reuse the buffer
- So it only blocks until buffer is safe to reuse
- (Recall we can only specify local semantics)

MPI_Recv

```
#include <mpi.h>
void Comm::Recv(void* buf, int count, const Datatype& datatype,
 ↪  int source, int tag, Status& status) const

void Comm::Recv(void* buf, int count, const Datatype& datatype,
 ↪  int source, int tag) const
```

- Blocking receive
- Semantics: Blocks until message is received. On return from call, buffer will have message data

Summary

- As-if is the most important principle in parallelization (correctness first)
- SPMD has high degree of self-similarity – solving global problem is same as solving local problem – communication enforces as-if
- Ubiquitous compute / communicate cycle

Thank You!

Creative Commons BY-NC-SA 4.0 License