

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

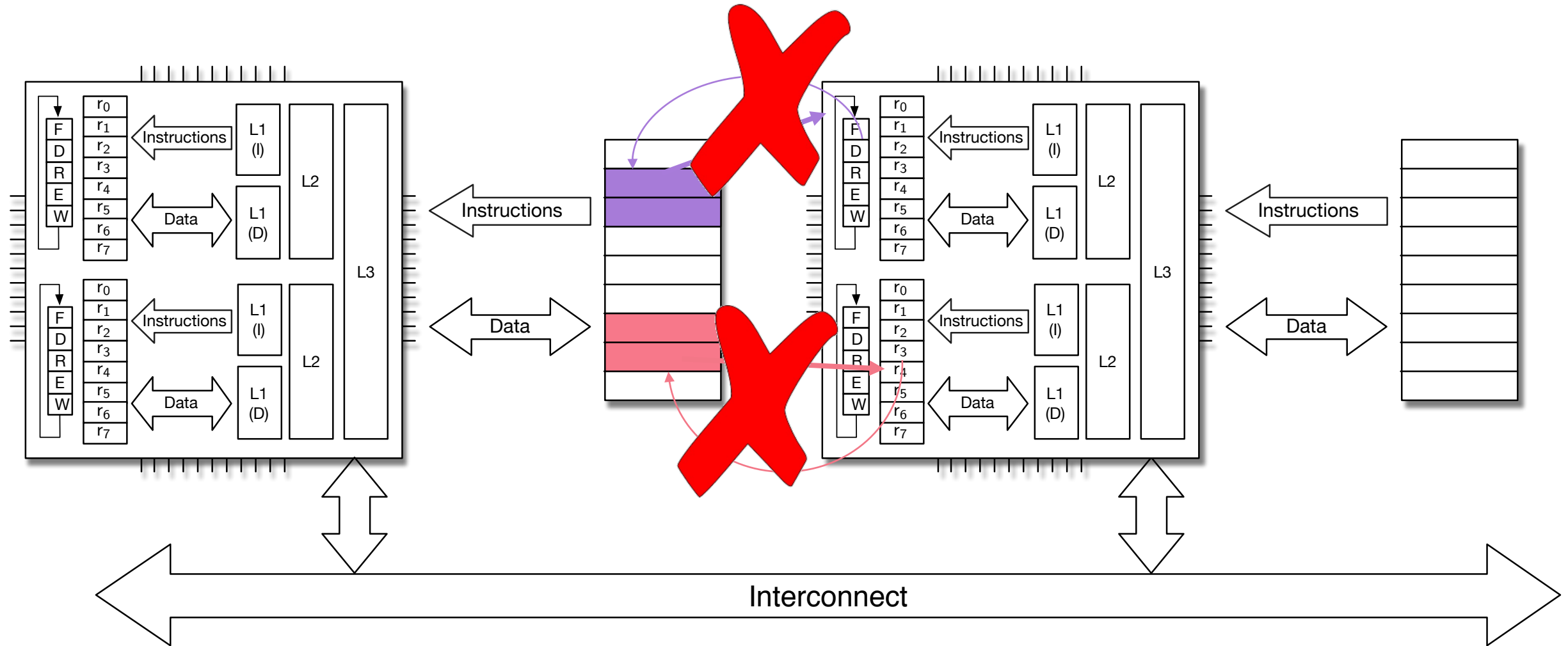
University of Washington

Seattle, WA

# Overview

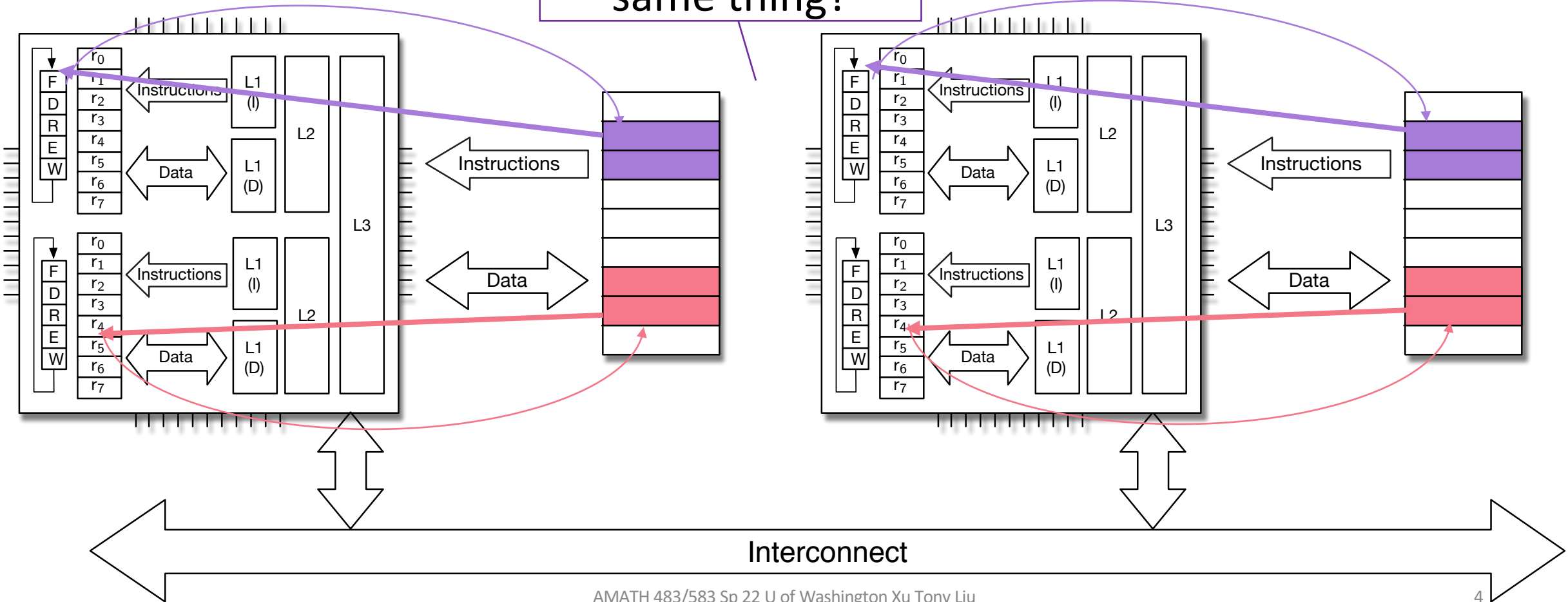
- SPMD / CSP recap
- MPI mental model
- Basic MPI
- Six Function MPI Point to Point Version
- Six Function MPI Collective Version
- Laplace's equation on a regular grid

# Distributed memory



# Distributed memory

Do we want these doing the exact same thing?

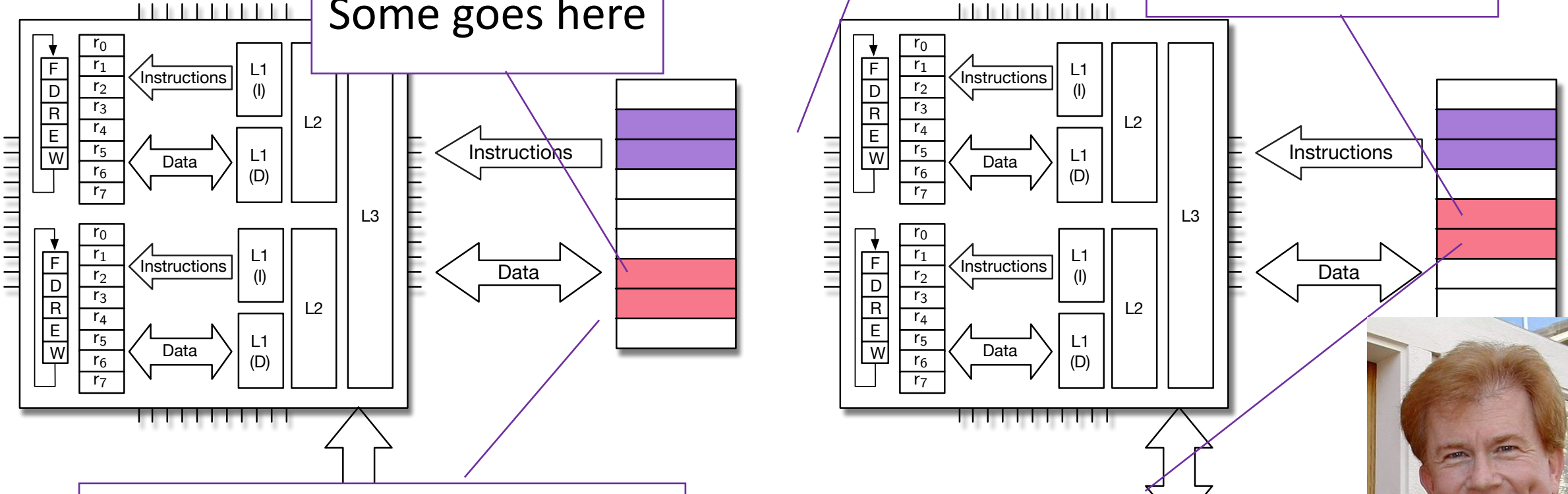


# Distributed memory

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

Some goes here

Some goes here

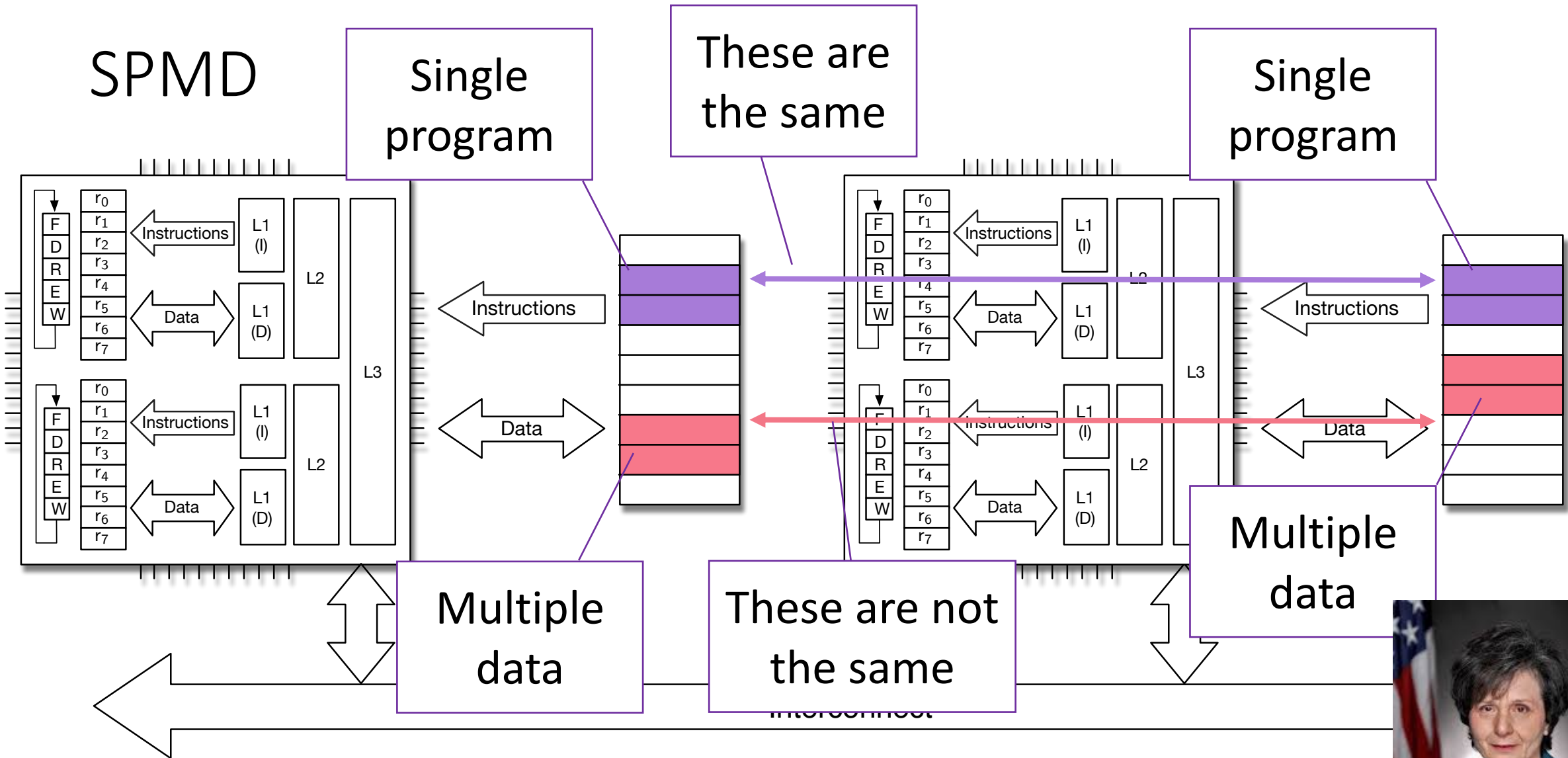


The union of the two should be the whole problem

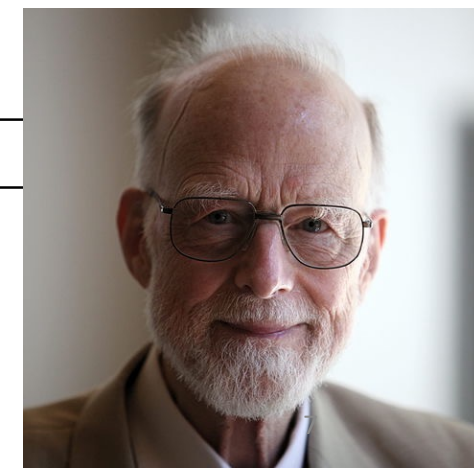
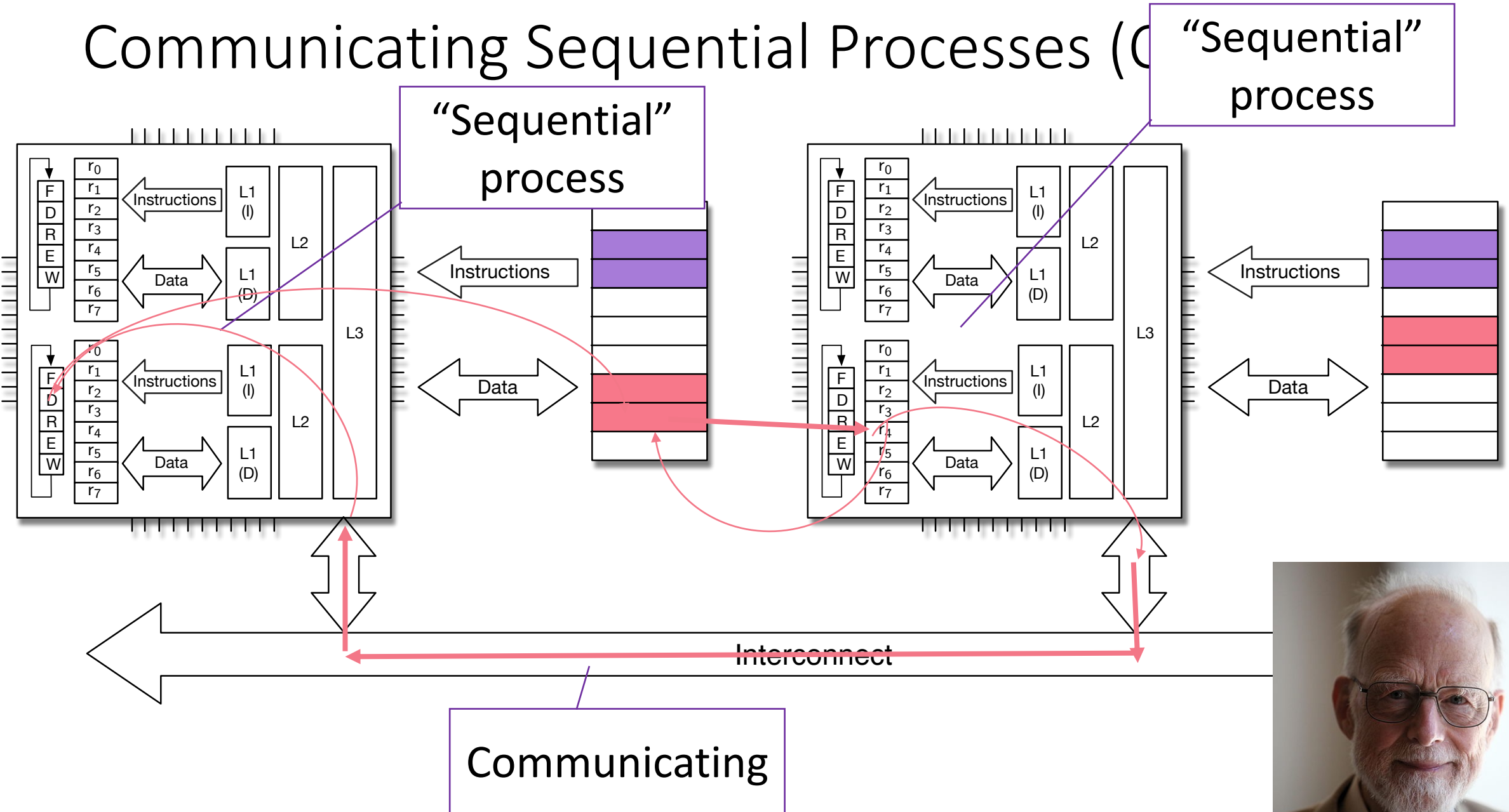
“Collectively exhaustive”



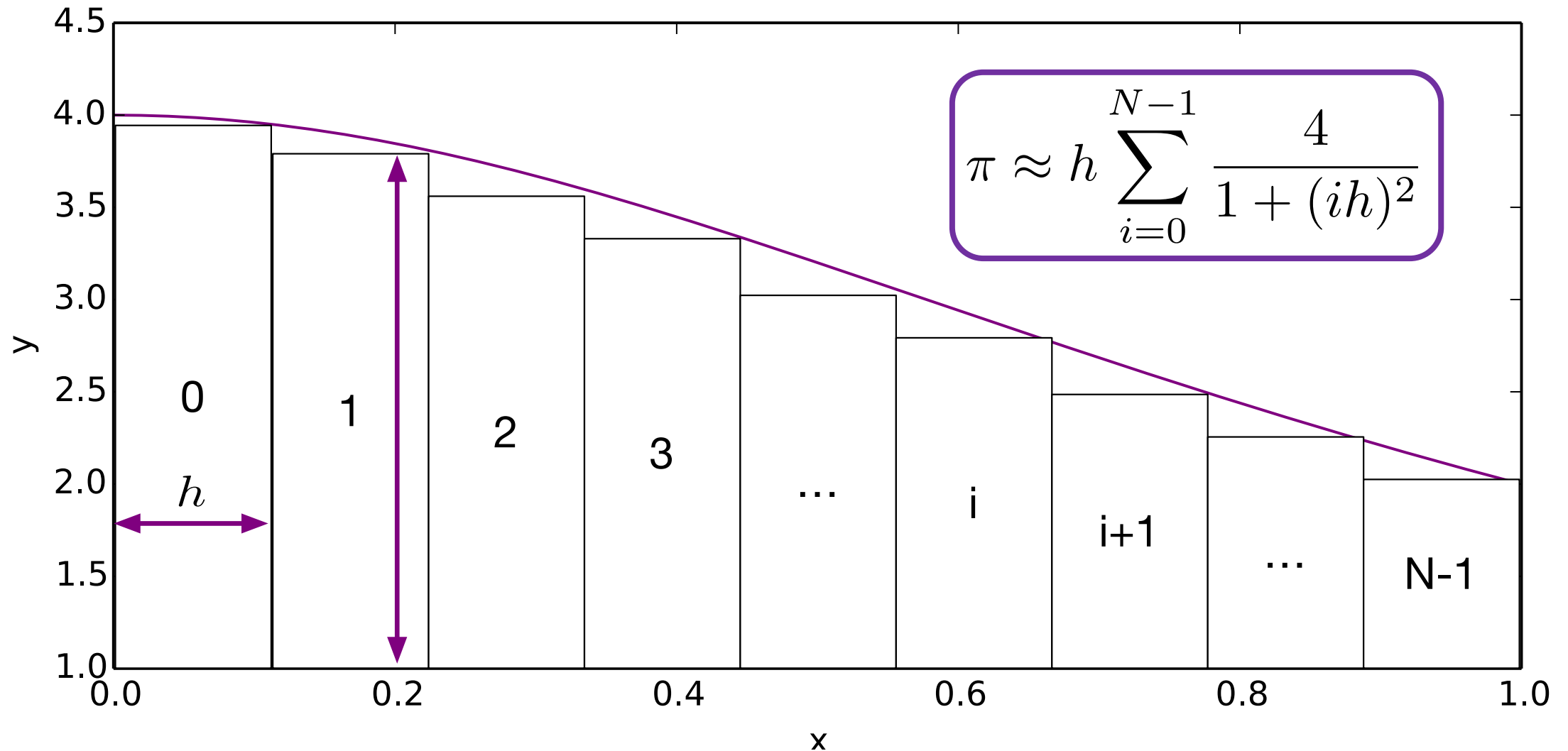
# SPMD



# Communicating Sequential Processes (CSP)



# Numerical Quadrature





# Parallelization

Finding  
Concurrency

Decompose problem into pieces  
that can execute in SPMD

Not really by task  
(single program)

Algorithm  
Structure

Manage sharing  
(communication)

By task or  
*by data*

Supporting  
Structures

Fundamental  
organizing principle

Around tasks or *around  
data decomposition* or  
around data flow

Implementation  
Mechanisms

Programming paradigms  
and data structures: SPMD

Not really by task  
(multiple data)

Manage processes,  
communication

# Finding Concurrency

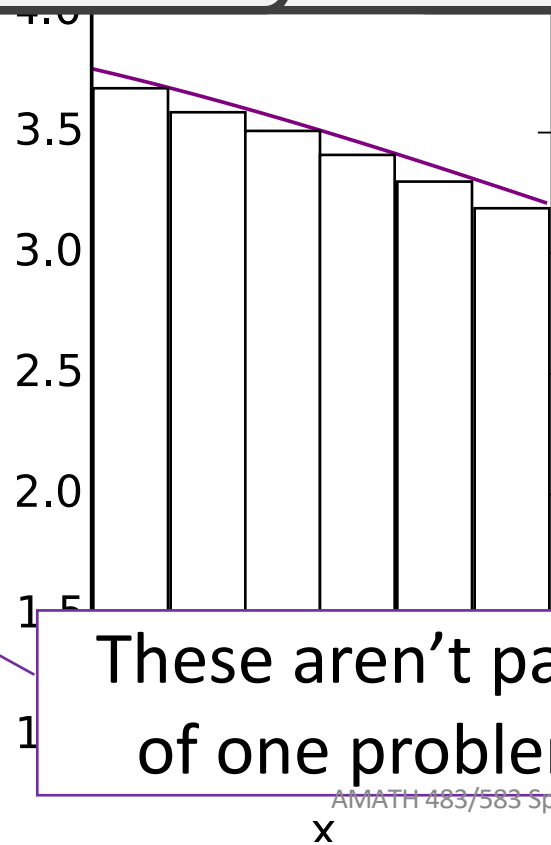
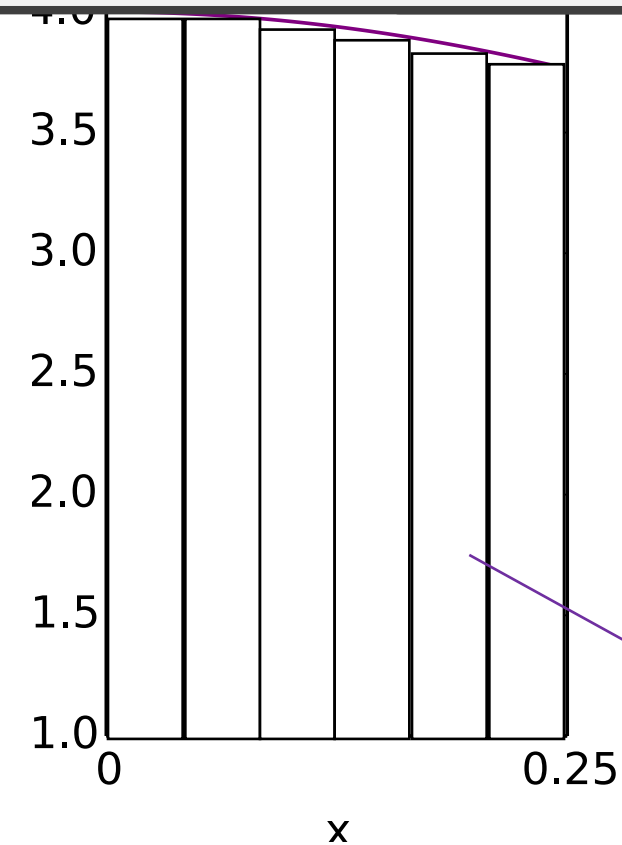
```
for (int i = begin; i < end; ++i) {  
    pi += h * 4.0 / (1 + i*h*i*h);  
}
```

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

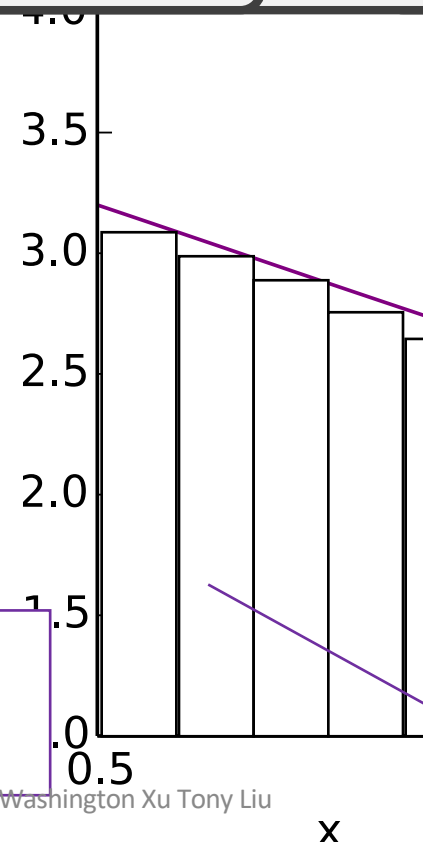
```
int i = N/4; i < N/2; ++i) {  
    pi += h * 4.0 / (1 + i*h*i*h);  
}
```

```
int i = N/2; i < 3*N/4; ++i) {  
    pi += h * 4.0 / (1 + i*h*i*h);  
}
```

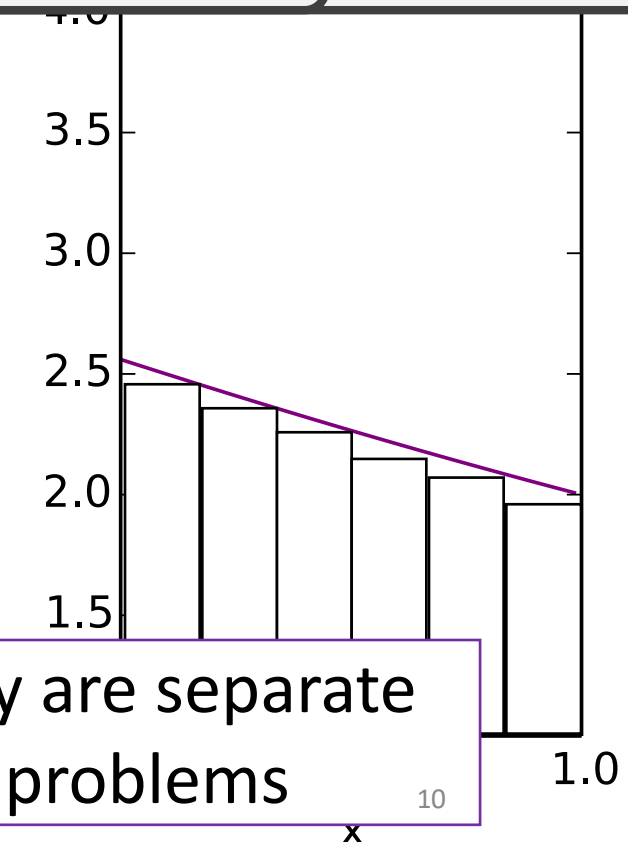
```
int i = 3*N/4; i < N; ++i) {  
    pi += h * 4.0 / (1 + i*h*i*h);  
}
```



These aren't parts of one problem

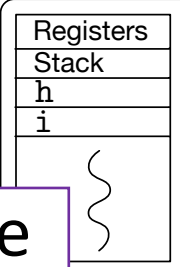


They are separate problems



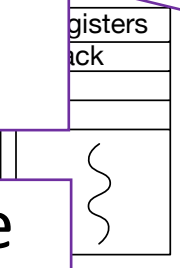
Process

Task



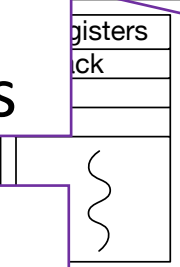
```
for (int i = 0; i < N/4; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```

Task



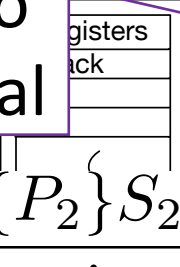
```
for (int i = N/4; i < N/2; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```

Task

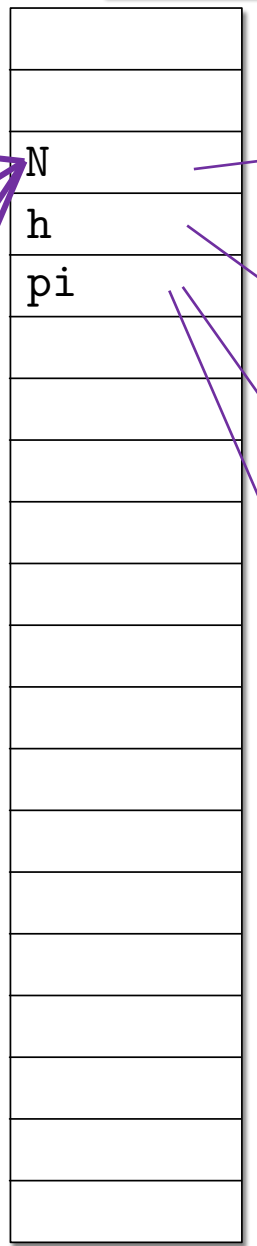


```
for (int i = N/2; i < 3*N/4; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```

Task



```
for (int i = 3*N/4; i < N; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```



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

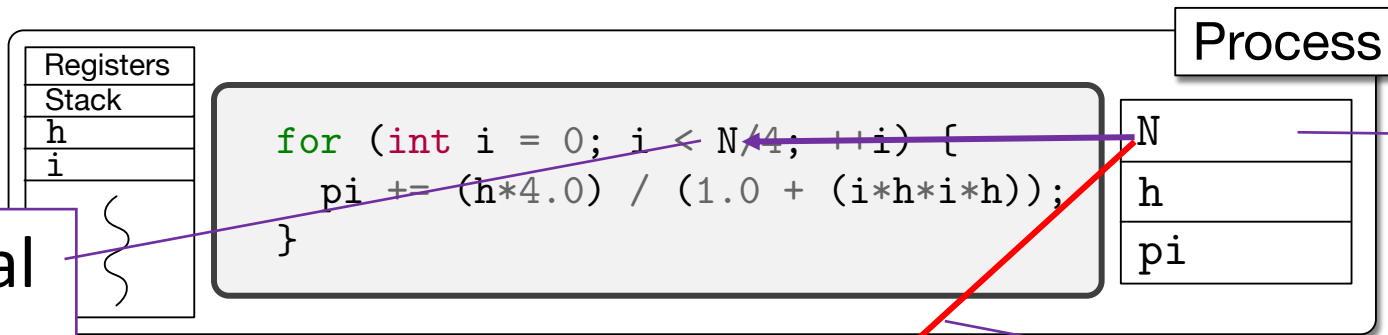
Similarly pi

At least for reading

(Have to deal with race when writing)

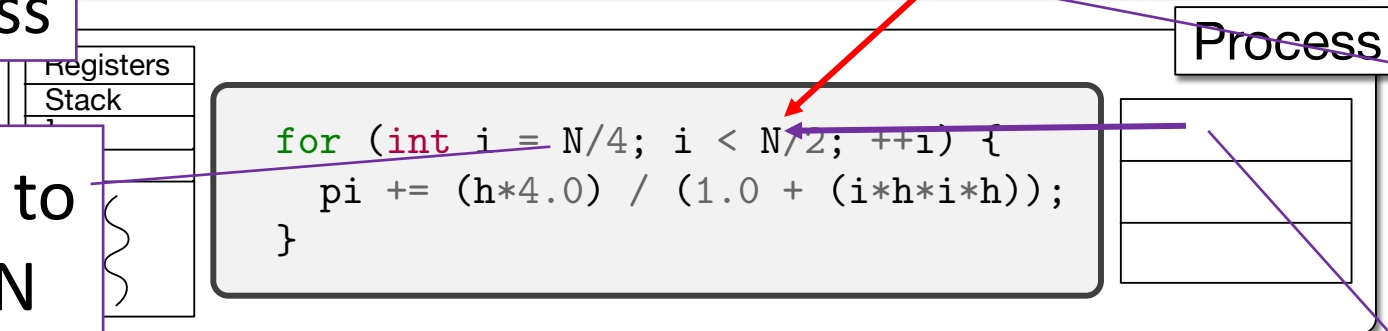
$\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\}$  interference free

$\{P_1\} \wedge \{P_2\}$  cobegin  $S_1 || S_2$  coend  $\{Q_1 \wedge Q_2\}$



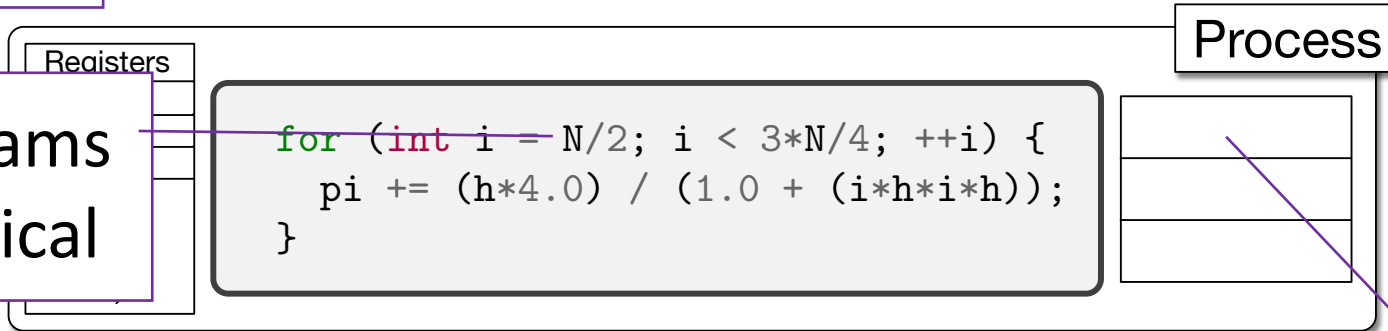
This N is local to this process

This N is local to this process



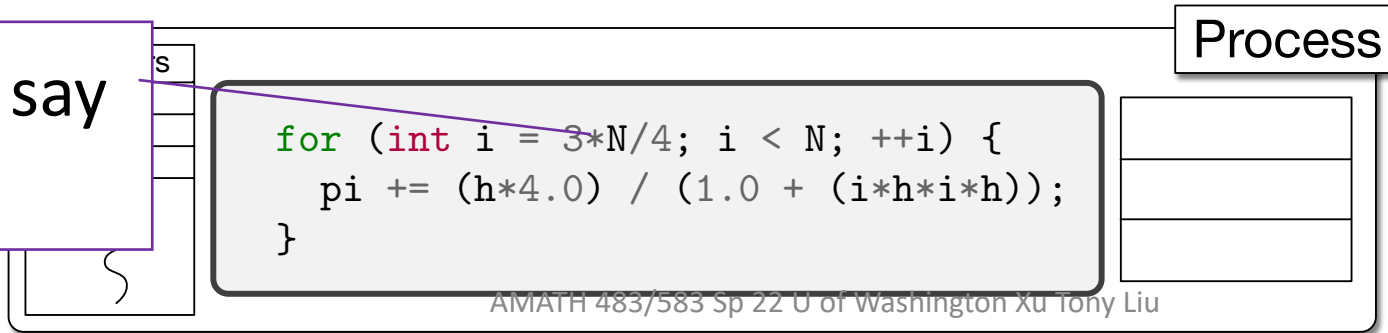
But we need to read *some* N

Can *not* read from another process memory



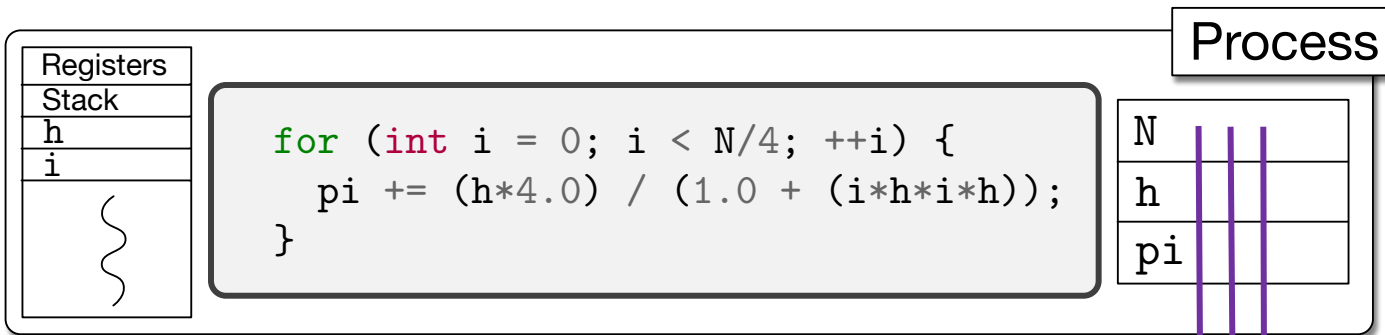
These programs are all identical

(In some sense there is an N here)

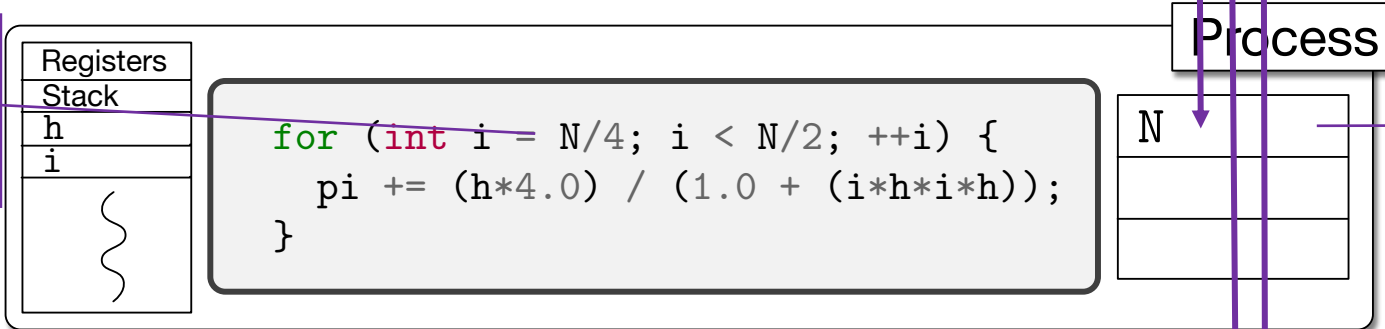


And they all say "read N"

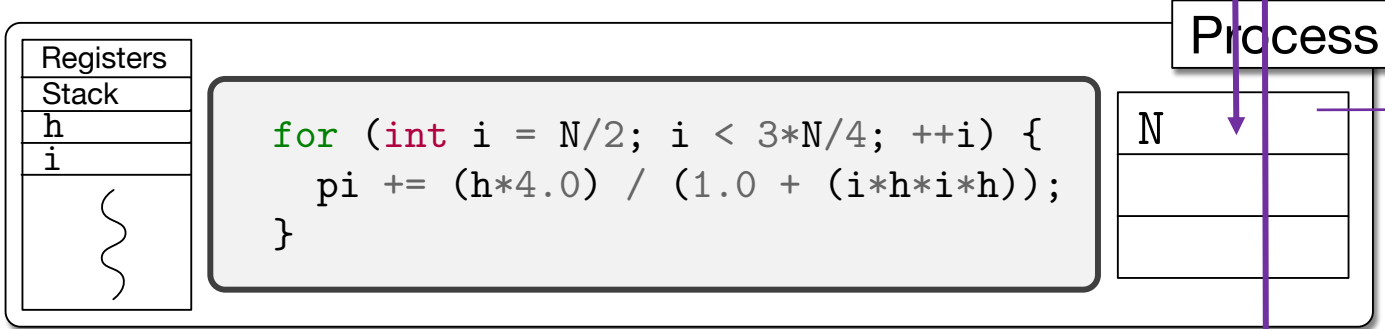
How do we get the right *value* for N here?



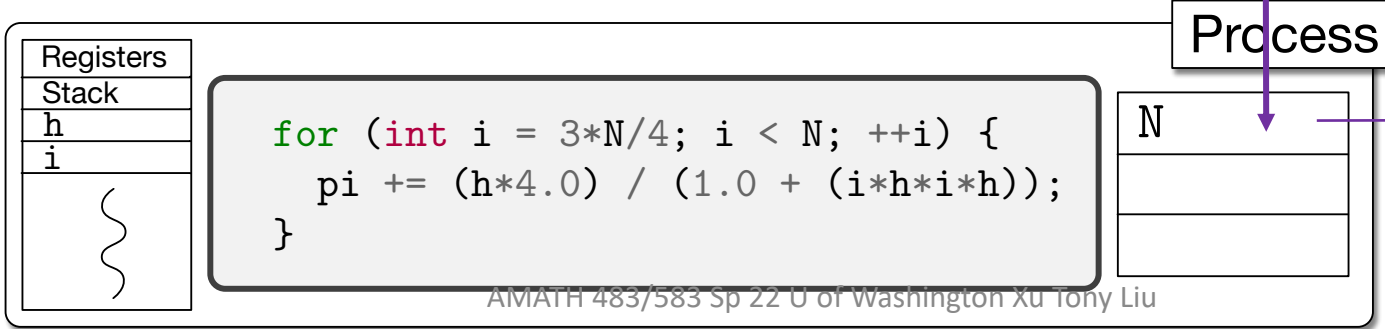
To read the "right" N



Copy to each process

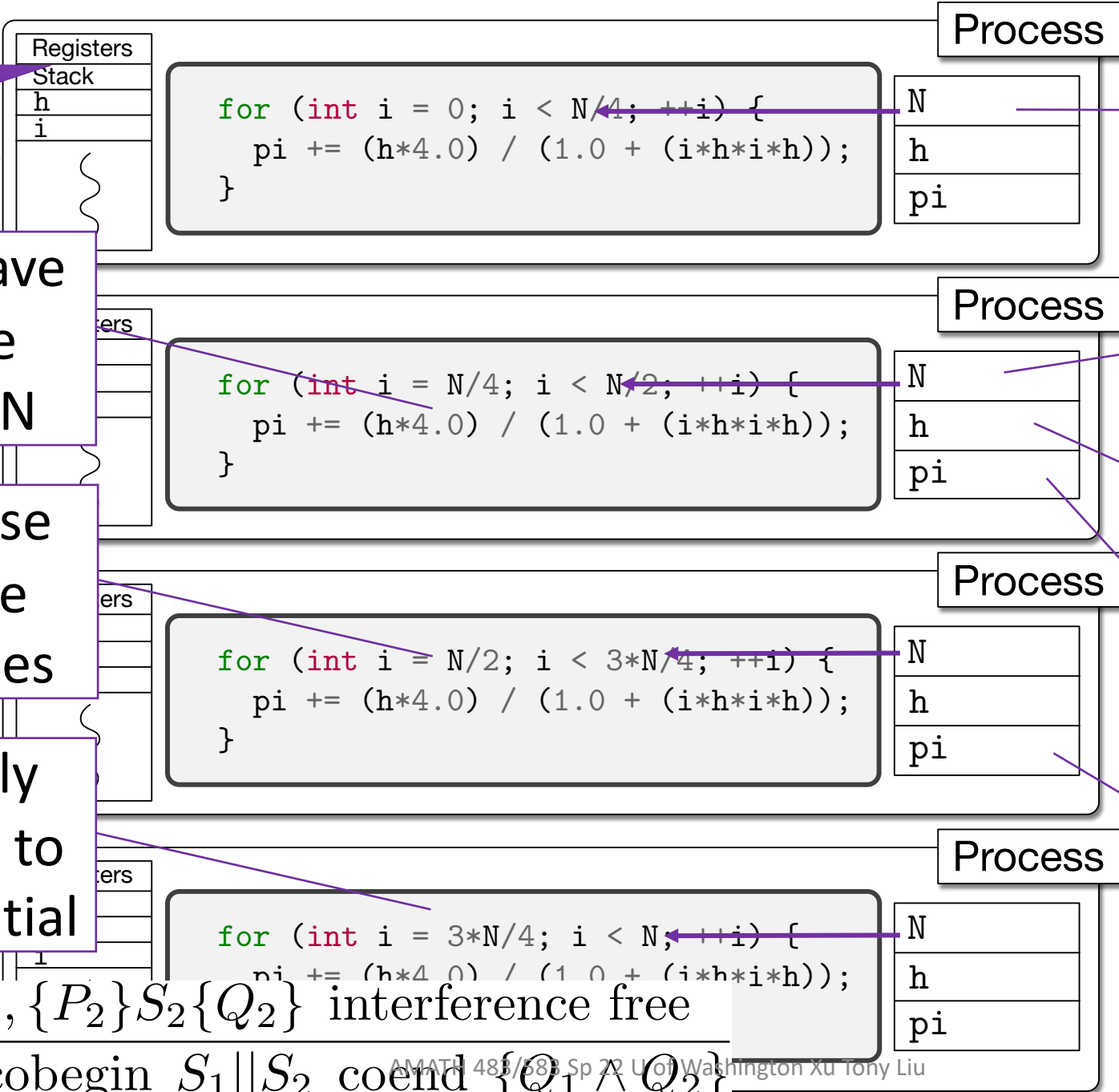


Copy to each process



Copy to each process

Very Important Slide!!



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***

Threads have the same value for `N`

And if these are all the same values

It is exactly equivalent to the sequential

$\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\}$  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;
}
```

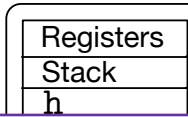
id 0 is root

This process can only read/write its memory

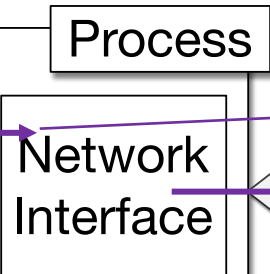
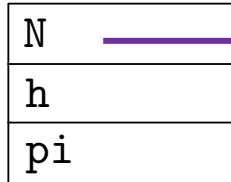
Only this process can read/write its memory

Message passing with CSP is **local**

It is **as if** shared memory, but it is purely local



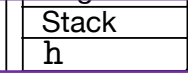
```
for (int i = 0; i < N/4; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```



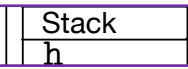
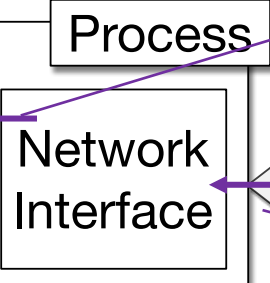
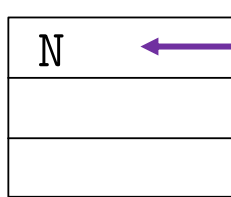
This process must **send**

This process must **receive**

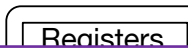
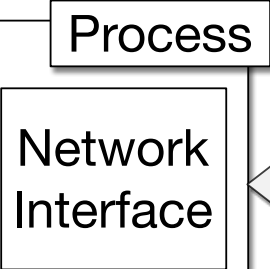
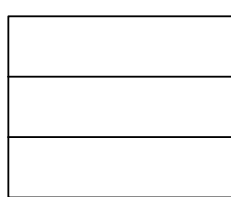
We can't just put this here



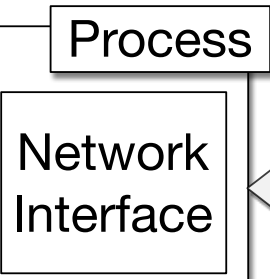
```
for (int i = N/4; i < N/2; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```



```
for (int i = N/2; i < 3*N/4; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```



```
for (int i = 3*N/4; i < N; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```





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;
}
```

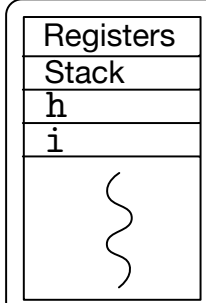
All copies call this communication function

And intervals gets copied to all processes

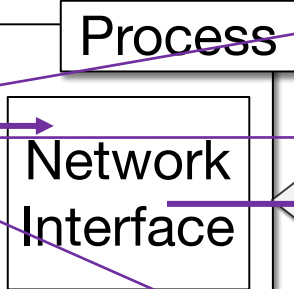
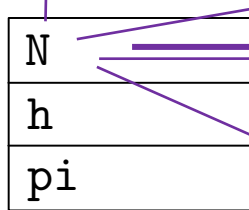
Only because each process calls this

First question: What are we sending?

“N” only has meaning in source code

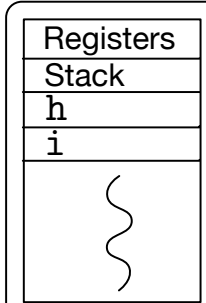


```
for (int i = 0; i < N/4; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```

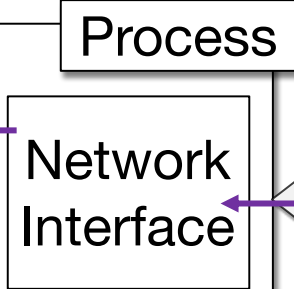
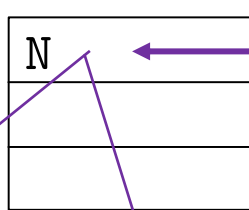


We are sending the **value** of “N”

The bits in the memory location



```
for (int i = N/4; i < N/2; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```



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

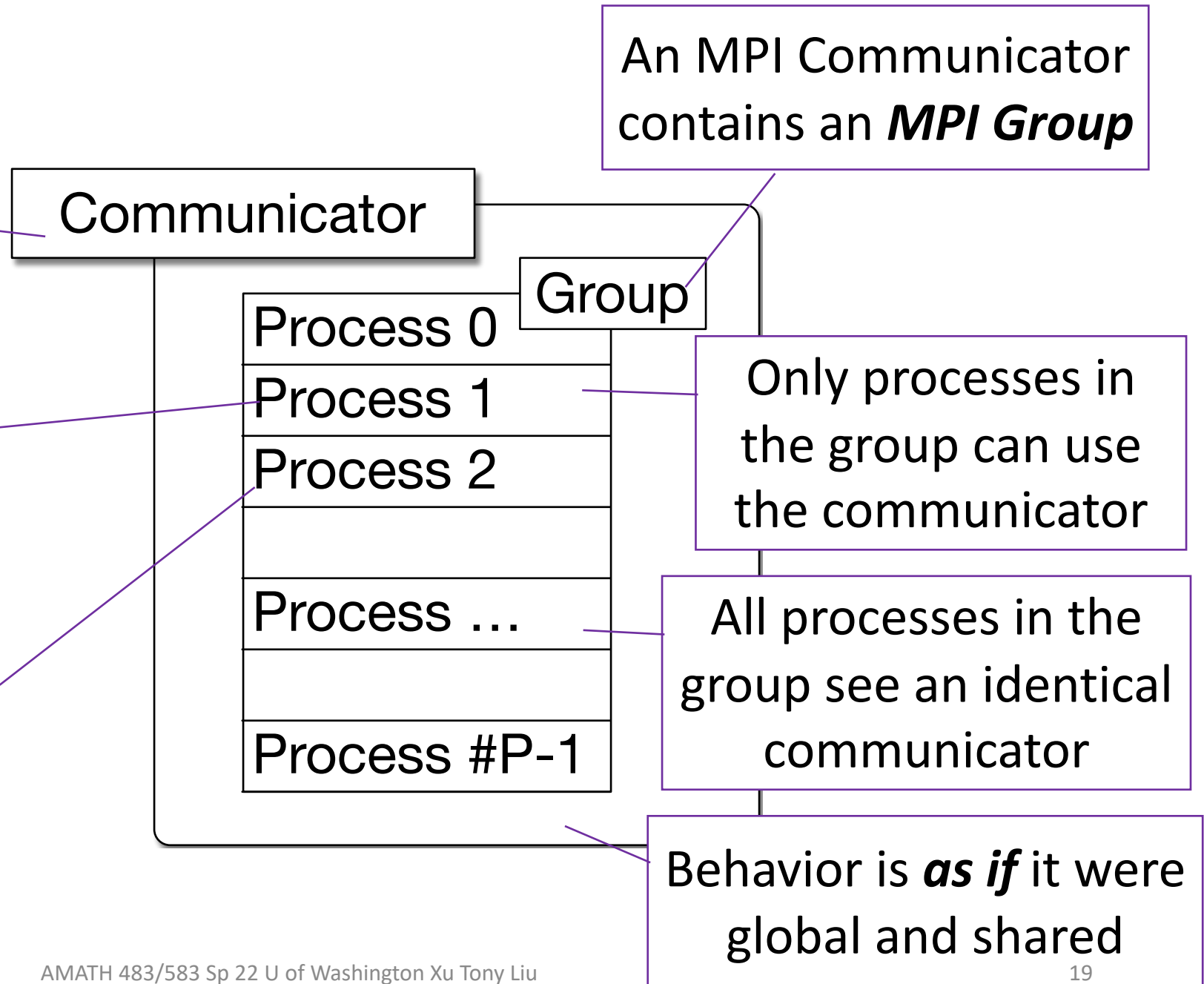
And its location only makes sense locally

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 use the index of a process in the group to identify other processes



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

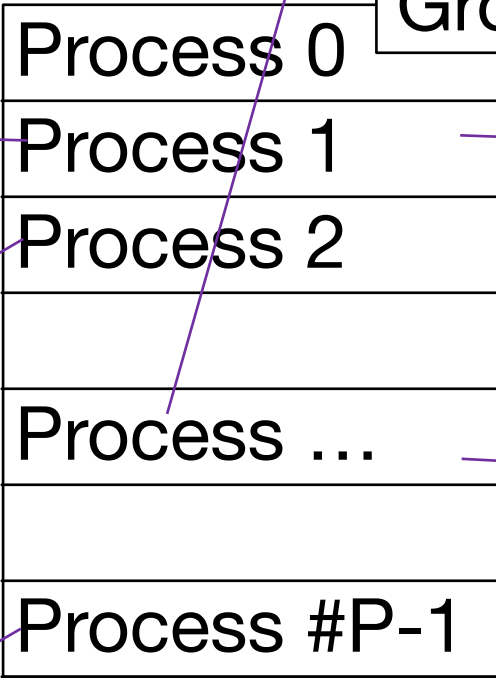
An MPI Communicator contains an **MPI Group**

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

Communicator

Group

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



Only processes in the group can use the communicator

We use 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

Member function  
of a communicator

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

Communicator used  
for this message

Recipient

Message tag

Sender is implicit  
(the process that  
called this function)

Message  
envelope

# MPI\_Recv

Member function  
of a communicator

Overloaded function  
returns status

```
#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
```

Communicator used  
for this message

Sender

Message tag

Message  
envelope

Receiver is implicit  
(the process that  
called this function)

# MPI\_Send and MPI\_Recv

?

Contents

```
#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
```

Match these for  
message  
delivery

NB: SPMD

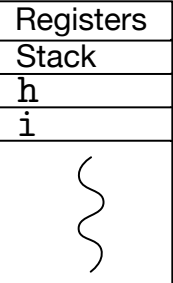
# Message Contents

Contents

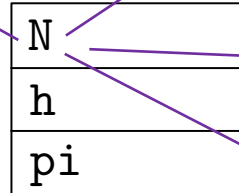
```
#include <mpi.h>
void Comm::Send(const void* buf, int count, const Datatype& datatype,
               int dest, int tag) const
```

In the program  
this is a value

In the computer  
this is just bits

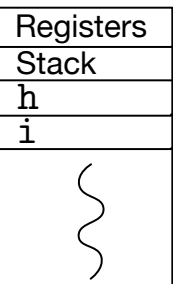


```
for (int i = 0; i < N/4; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```



Network  
Interface

The value represented  
by bits is not defined



```
for (int i = N/4; i < N/2; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```



Network  
Interface

Only makes sense in a  
given process / CPU



# Message Contents

Contents

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

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

The screenshot shows the open-mpi.org website. The left sidebar contains navigation links such as 'v1.2 (ancient)', 'Source Code Access', 'Bug Tracking', 'Regression Testing', 'Version Information', 'Sub-Projects', 'Hardware Locality', 'Network Locality', 'MPI Testing Tool', 'Open MPI User Docs', 'Open Tool for Parameter Optimization', 'PMIx', 'Community', 'Mailing Lists', 'Getting Help/Support', 'Contribute', 'Mirrors', 'Contact', and 'License'. The main content area is titled 'MPI API (section 3 man pages)' and displays a grid of links to various MPI functions, including MPI\_Abort, MPI\_Accumulate, MPI\_Add\_error\_class, MPI\_Add\_error\_code, MPI\_Add\_error\_string, MPI\_Address, MPI\_Allgather, MPI\_Allgatherv, MPI\_Alloc\_mem, MPI\_Allreduce, MPI\_Alltoall, MPI\_Alltoallv, MPI\_Alltoallw, MPI\_Attr\_delete, MPI\_Attr\_get, MPI\_Attr\_put, MPI\_Barrier, MPI\_Bcast, MPI\_Bsend, MPI\_Bsend\_init, MPI\_Buffer\_attach, MPI\_Buffer\_detach, MPI\_Cancel, MPI\_Cart\_coords, MPI\_Cart\_create, MPI\_Cart\_get, MPI\_Cart\_map, MPI\_Cart\_rank, MPI\_Cart\_shift, MPI\_Cart\_sub, MPI\_Cartdim\_get, MPI\_Close\_port, MPI\_Comm\_accept, MPI\_Comm\_c2f, MPI\_Comm\_call\_errhandler, MPI\_Comm\_compare, MPI\_Comm\_connect, MPI\_Comm\_create, MPI\_Comm\_create\_errhandler, MPI\_Comm\_create\_group, MPI\_Comm\_create\_keyval, MPI\_Comm\_delete\_attr, MPI\_Comm\_disconnect, MPI\_Comm\_dup, MPI\_Comm\_dup\_with\_info, MPI\_Comm\_f2c, MPI\_Comm\_free, MPI\_Comm\_free\_keyval, MPI\_Comm\_get\_attr, MPI\_Comm\_get\_errhandler, MPI\_Comm\_get\_info, MPI\_Comm\_get\_name, MPI\_File\_get\_errhandler, MPI\_File\_get\_group, MPI\_File\_get\_info, MPI\_File\_get\_position, MPI\_File\_get\_position\_shared, MPI\_File\_get\_size, MPI\_File\_get\_type\_extnt, MPI\_File\_get\_view, MPI\_File\_iread, MPI\_File\_iread\_at, MPI\_File\_iread\_shared, MPI\_File\_irewrite, MPI\_File\_irewrite\_at, MPI\_File\_irewrite\_shared, MPI\_File\_open, MPI\_File\_preallocate, MPI\_File\_read, MPI\_File\_read\_all, MPI\_File\_read\_all\_begin, MPI\_File\_read\_all\_end, MPI\_File\_read\_at, MPI\_File\_read\_at\_all, MPI\_File\_read\_at\_all\_begin, MPI\_File\_read\_at\_all\_end, MPI\_File\_read\_ordered, MPI\_File\_read\_ordered\_begin, MPI\_File\_read\_ordered\_end, MPI\_File\_read\_shared, MPI\_File\_seek, MPI\_File\_seek\_shared, MPI\_File\_set\_atomicity, MPI\_File\_set\_errhandler, MPI\_File\_set\_info, MPI\_File\_set\_size, MPI\_File\_set\_view, MPI\_File\_sync, MPI\_File\_write, MPI\_File\_write\_all, MPI\_File\_write\_all\_begin, MPI\_File\_write\_all\_end, MPI\_File\_write\_at, MPI\_File\_write\_at\_all, MPI\_File\_write\_at\_all\_begin, MPI\_File\_write\_at\_all\_end, MPI\_File\_write\_ordered, MPI\_File\_write\_ordered\_begin, MPI\_File\_write\_ordered\_end, MPI\_File\_write\_shared, MPI\_Finalize, MPI\_Finalized, MPI\_Free\_mem, MPI\_Gather, MPI\_Gatherv, MPI\_Iexscan, MPI\_Igather, MPI\_Igatherv, MPI\_Improbe, MPI\_Imrecv, MPI\_Ineighbor\_allgather, MPI\_Ineighbor\_allgatherv, MPI\_Ineighbor\_alltoall, MPI\_Ineighbor\_alltoallv, MPI\_Ineighbor\_alltoallw, MPI\_Info\_c2f, MPI\_Info\_create, MPI\_Info\_delete, MPI\_Info\_dup, MPI\_Info\_env, MPI\_Info\_f2c, MPI\_Info\_free, MPI\_Info\_get, MPI\_Info\_get\_nkeys, MPI\_Info\_get\_nthkey, MPI\_Info\_get\_valuelen, MPI\_Info\_set, MPI\_Init, MPI\_Init\_thread, MPI\_Initialized, MPI\_Intercomm\_create, MPI\_Intercomm\_merge, MPI\_Iprobe, MPI\_Irecv, MPI\_Ireduce, MPI\_Ireduce\_scatter, MPI\_Ireduce\_scatter\_block, MPI\_Irsend, MPI\_Is\_thread\_main, MPI\_Iscan, MPI\_Iscatter, MPI\_Iscatterv, MPI\_Isend, MPI\_Issend, MPI\_Keyval\_create, MPI\_Keyval\_free, MPI\_Lookup\_name, MPI\_Mprobe, MPI\_Mrecv, MPI\_Neighbor\_allgather, MPI\_Neighbor\_allgatherv, MPI\_Neighbor\_alltoall, MPI\_Neighbor\_alltoallv, MPI\_Neighbor\_alltoallw, MPI\_Op\_c2f, MPI\_Op\_create, MPI\_Op\_f2c, MPI\_Op\_free, MPI\_Status\_f2c, MPI\_Status\_set\_cancelled, MPI\_Status\_set\_elements, MPI\_Status\_set\_elements\_x, MPI\_Test, MPI\_Test\_cancelled, MPI\_Testall, MPI\_Testany, MPI\_Testsome, MPI\_Topo\_test, MPI\_Type\_c2f, MPI\_Type\_commit, MPI\_Type\_contiguous, MPI\_Type\_create\_darray, MPI\_Type\_create\_f90\_complex, MPI\_Type\_create\_f90\_integer, MPI\_Type\_create\_f90\_real, MPI\_Type\_create\_hindexed, MPI\_Type\_create\_hindexed\_block, MPI\_Type\_create\_hvector, MPI\_Type\_create\_indexed\_block, MPI\_Type\_create\_keyval, MPI\_Type\_create\_resized, MPI\_Type\_create\_struct, MPI\_Type\_create\_subarray, MPI\_Type\_delete\_attr, MPI\_Type\_dup, MPI\_Type\_extent, MPI\_Type\_f2c, MPI\_Type\_free, MPI\_Type\_free\_keyval, MPI\_Type\_get\_attr, MPI\_Type\_get\_contents, MPI\_Type\_get\_envelope, MPI\_Type\_get\_extent, MPI\_Type\_get\_extent\_x, MPI\_Type\_get\_name, MPI\_Type\_get\_true\_extent, MPI\_Type\_get\_true\_extent\_x, MPI\_Type\_hindexed, MPI\_Type\_hvector, MPI\_Type\_indexed, MPI\_Type\_lb, MPI\_Type\_match\_size, MPI\_Type\_set\_attr, MPI\_Type\_set\_name, MPI\_Type\_size, MPI\_Type\_size\_x, MPI\_Type\_struct, MPI\_Type\_ub, MPI\_Type\_vector, MPI\_Unpack, MPI\_Unpack\_external

<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

```
#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

**Name**  
MPI\_Recv - Performs a standard-mode blocking receive.

**Syntax**  
**C Syntax**

```
#include <mpi.h>
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
```

**Fortran Syntax**

```
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
```

**C++ Syntax**

```
#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
```

**Input Parameters**

- count: Maximum number of elements to receive (integer).
- datatype: Datatype of each receive buffer entry (handle).
- source: Rank of source (integer).
- tag: Message tag (integer).
- comm: Communicator (handle).

**Output Parameters**

- buf: Initial address of receive buffer (choice).
- status: Status object (status).
- IERROR: Fortran only: Error status (integer).

**Description**

This basic receive operation, MPI\_Recv, is blocking: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

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

# Hello MPI World

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

Include mpi.h

NB: namespace MPI

```
int main() {
```

Initialize

```
    MPI::Init();
```

Get size of  
communicator

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

Get rank of  
calling process

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

```
    MPI::Finalize();
```

Finalize

```
    return 0;
}
```

# Hello MPI World

```
#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

```
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, 321);  
    MPI::COMM_WORLD.Recv(&ballreceived, 1, MPI::INT, 1, 321);  
    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

```
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;  
}
```

Only process 0  
sends this

# 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;
} AMATH 483/583 Sp 22 U of Washington Xu Tony Liu
```

# 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

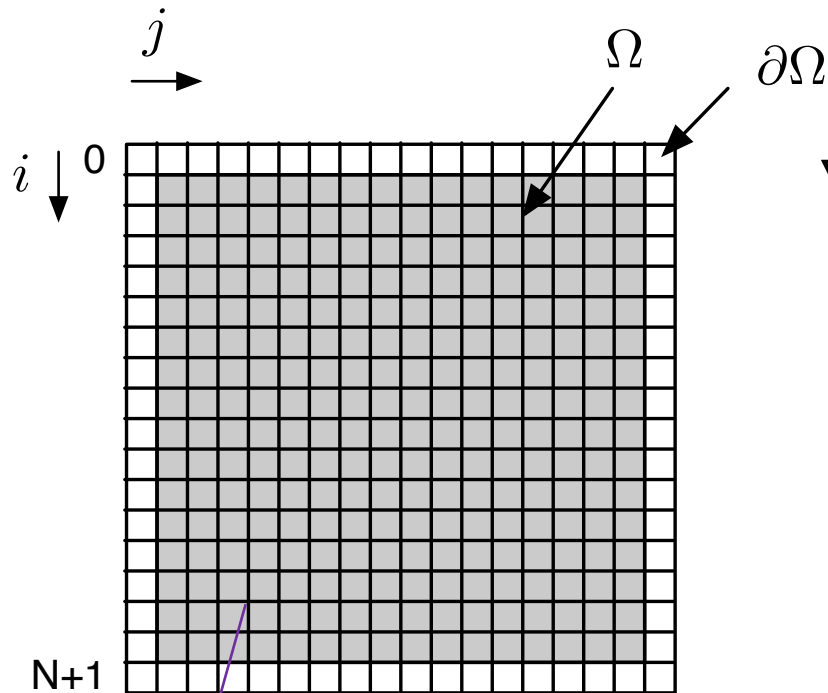
Once all nodes  
have called it

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



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

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



Discretization



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

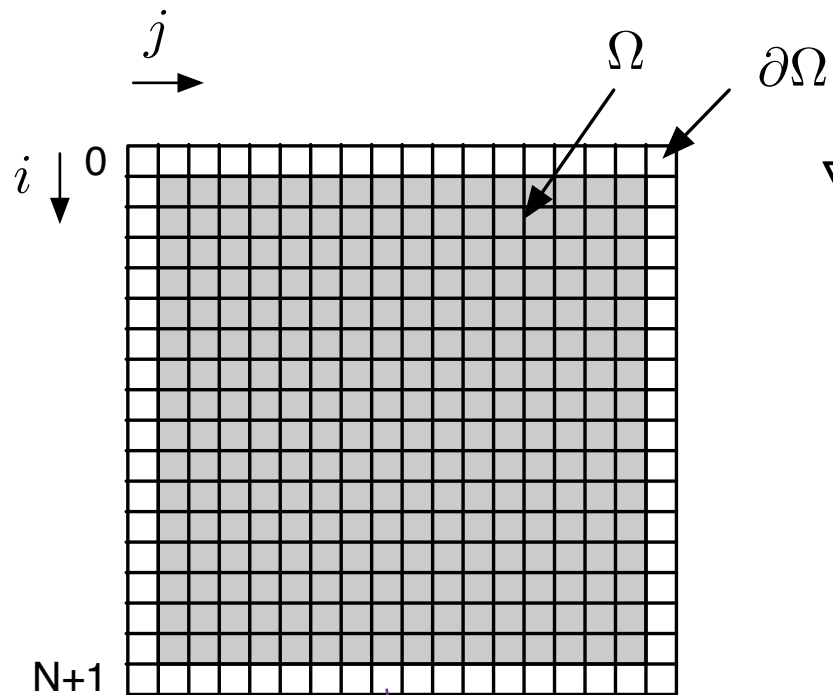
$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1}) / 4$$

$x_{i,j}$

The value of each point on the grid

The average of its neighbors

# Laplace's Equation on a Regular Grid



Why isn't 0 the solution?

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

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

Discret  $\downarrow$  The boundary is non-zero

$\uparrow$  Non-zeros in here due to boundary

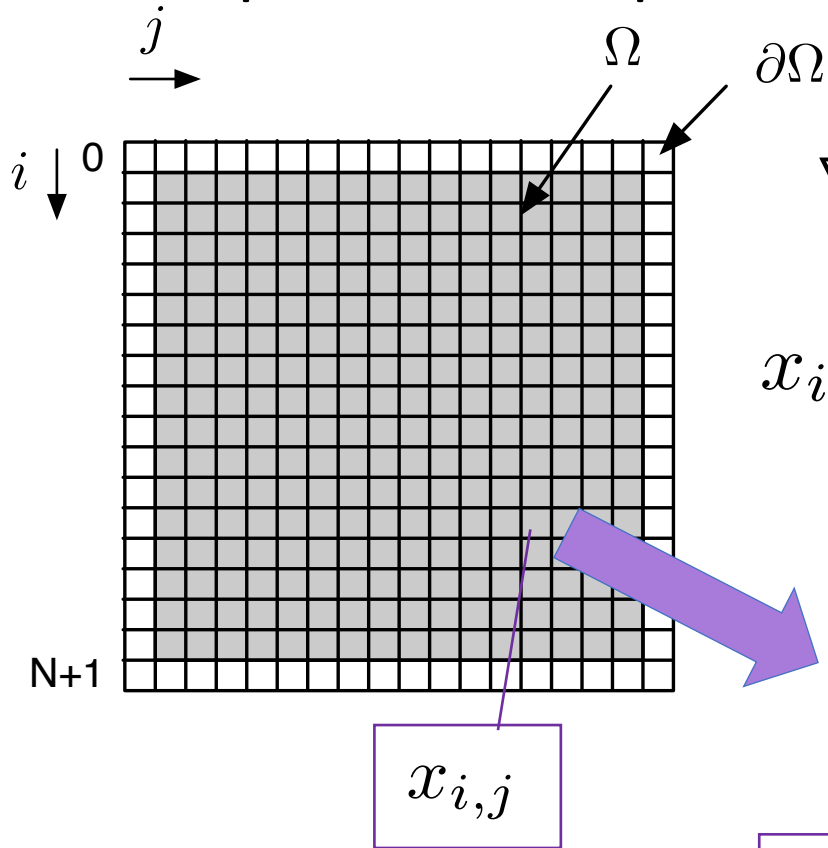
The boundary is non-zero

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

The value of each point on the grid

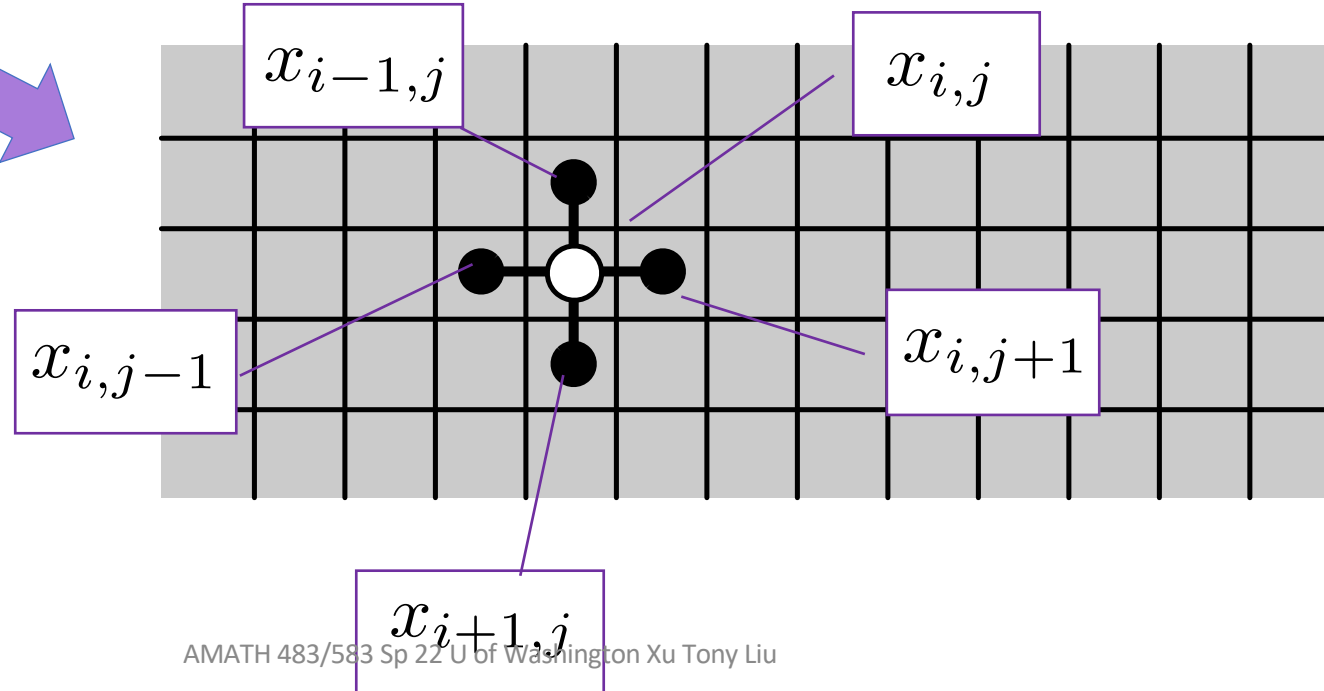
The average of its neighbors

# Laplace's Equation on a Regular Grid

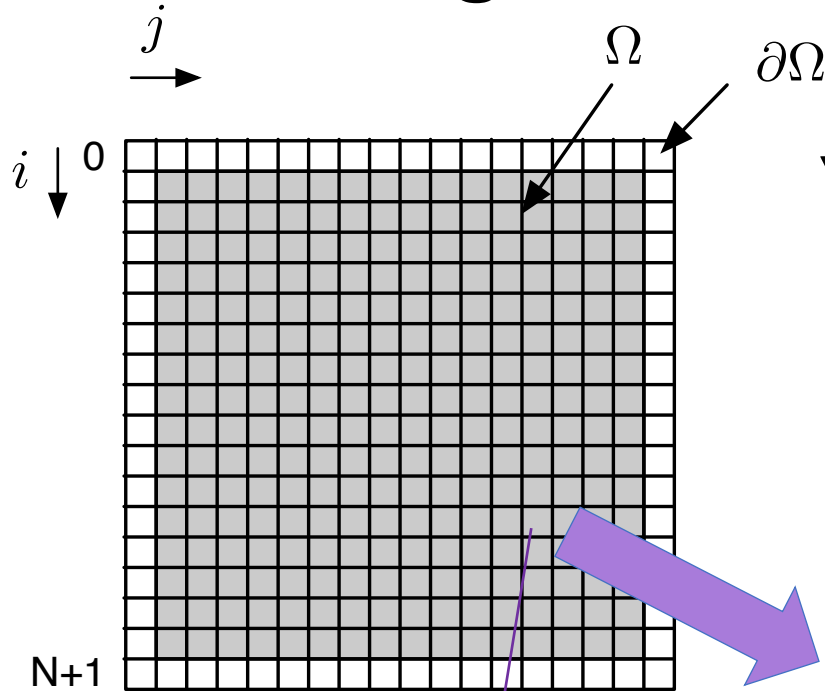


$$\begin{aligned}\nabla^2\phi &= 0 \quad \text{on } \Omega \\ \phi &= f \quad \text{on } \partial\Omega\end{aligned}$$

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



# Iterating for a solution



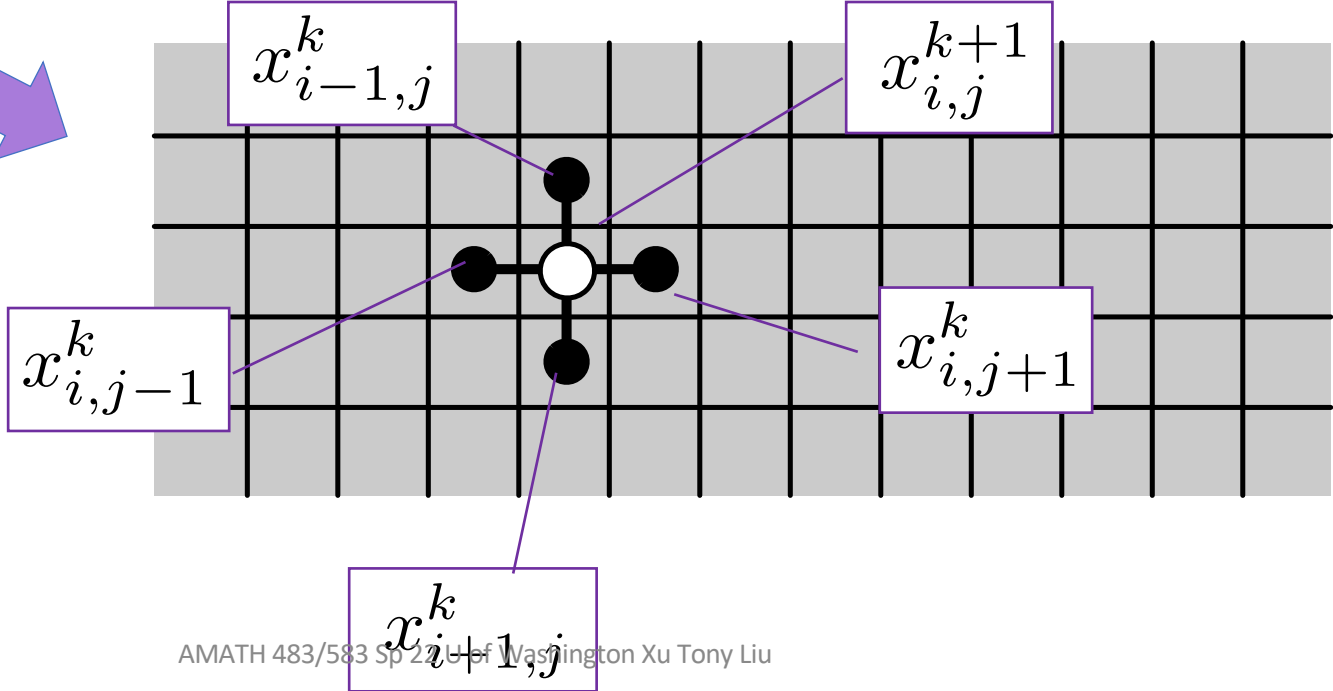
$$\begin{aligned} \nabla^2 \phi &= 0 && \text{on } \Omega \\ \phi &= f && \text{on } \partial\Omega \end{aligned}$$

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

Approximation at iteration  $k+1$

Average of approximation at iteration  $k$

$x_{i,j}$





# Iterating for a solution

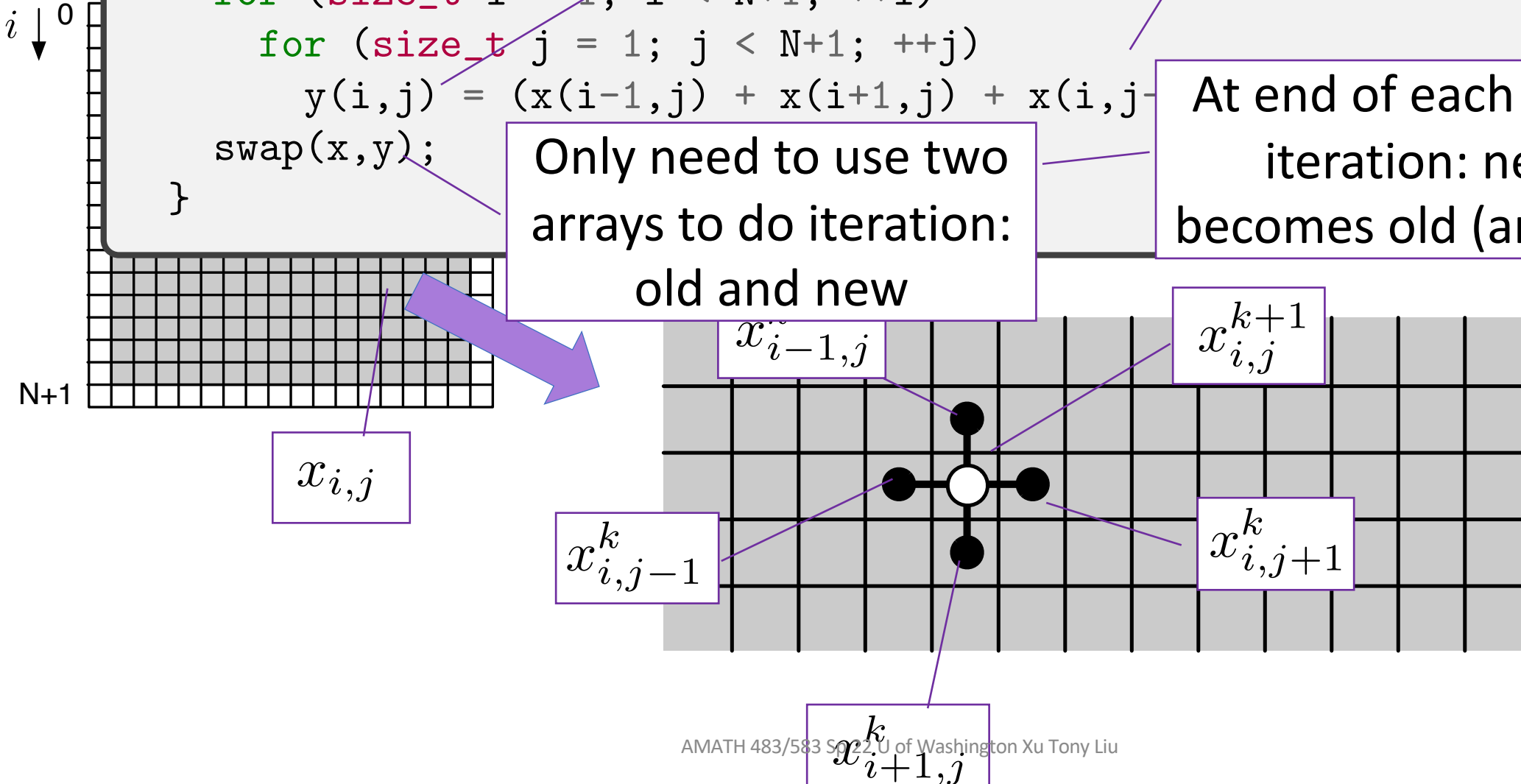
```
while (! converged()) {  
  for (size_t i = 1; i < N+1; ++i)  
    for (size_t j = 1; j < N+1; ++j)  
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1)) / 4;  
  swap(x,y);  
}
```

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

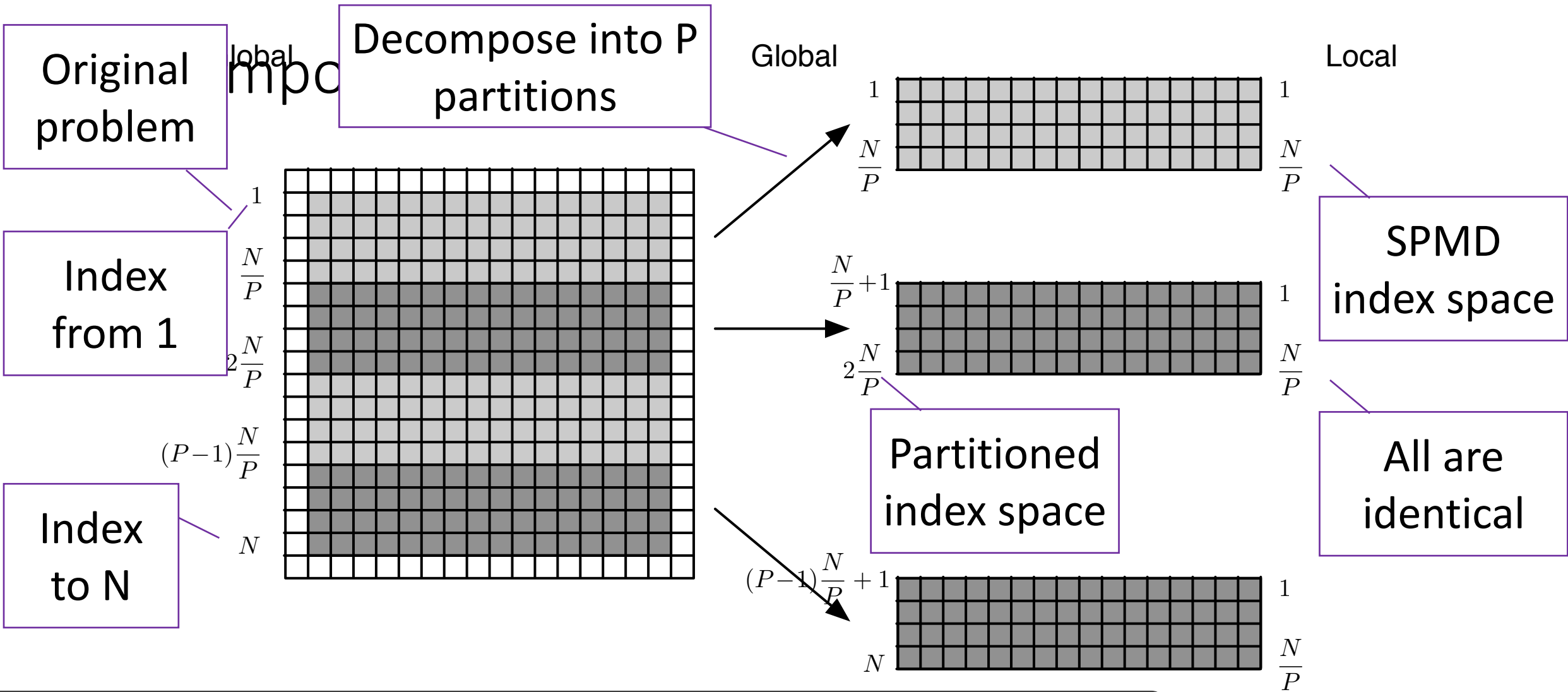
```
double jacobiStep(const Grid& x, Grid& y) {
    assert(x.numX() == y.numX() && x.numY() == y.numY());
    double rnorm = 0.0;

    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;

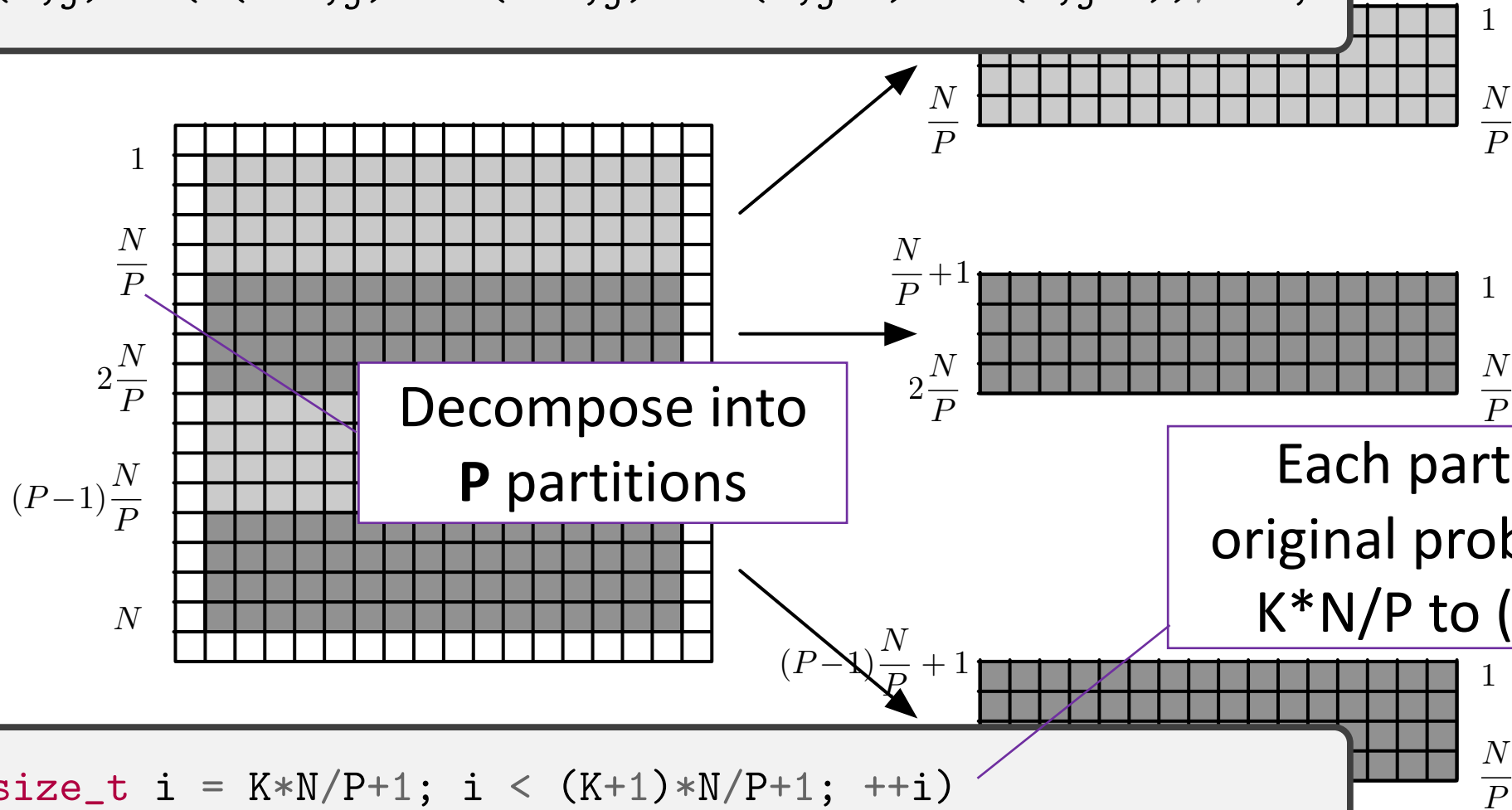
```

```

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

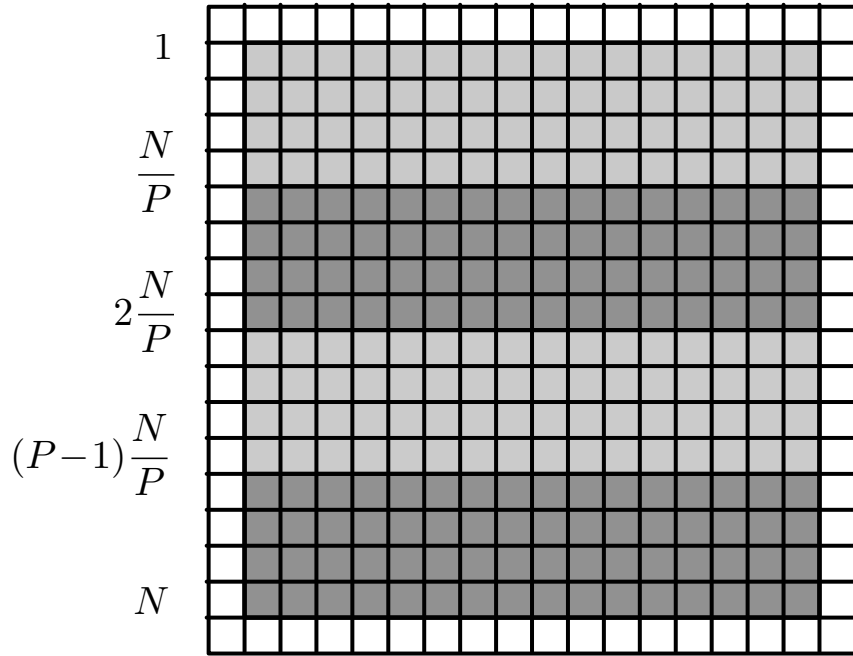


```

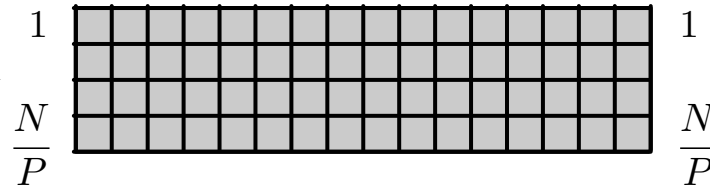
for (size_t i = K*N/P+1; i < (K+1)*N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;

```

# Global Decomposition

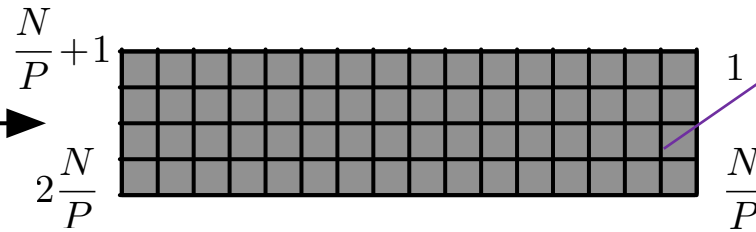


Global



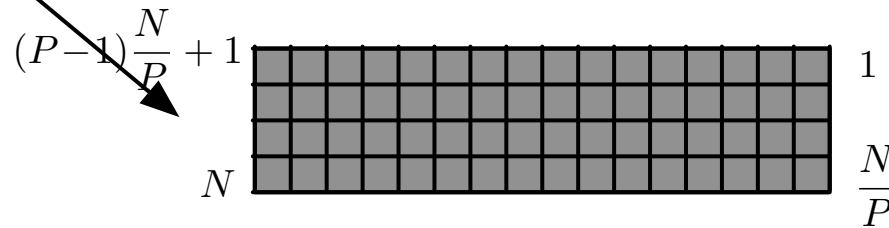
Local

We aren't storing entire array on each node



We are storing only N/P rows

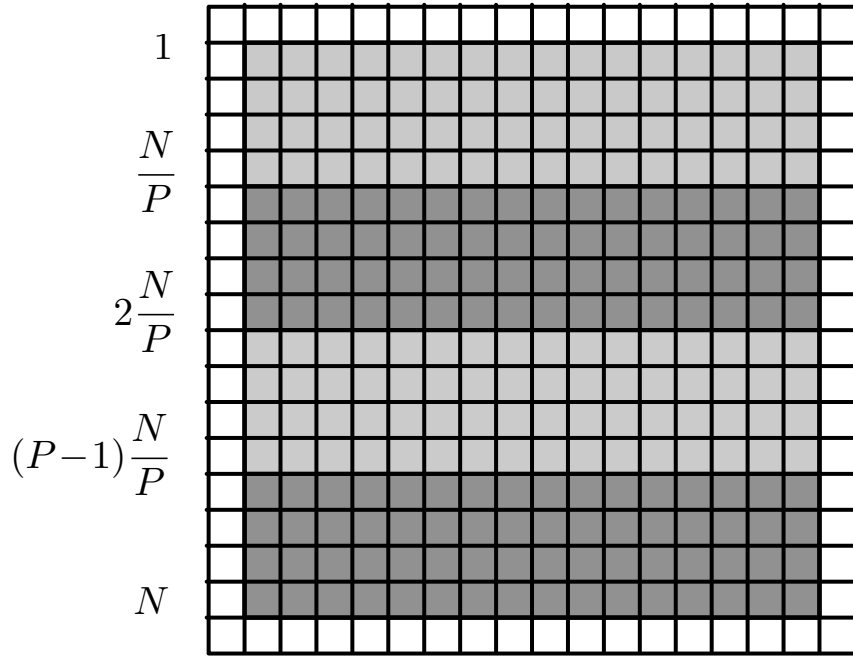
...



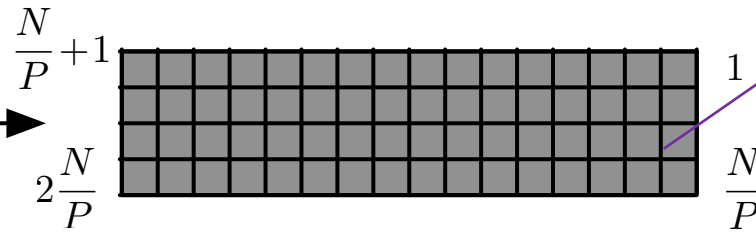
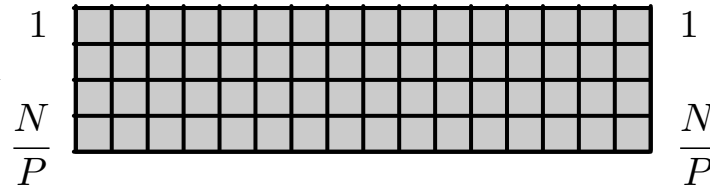
We need to iterate 1 to N

```
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;
```

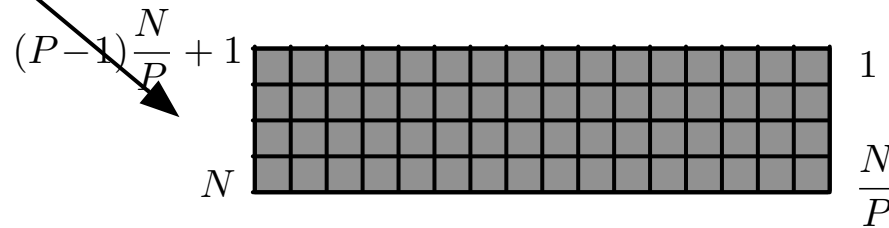
# Global Decomposition



Global



...



Local

We aren't storing entire array on each node

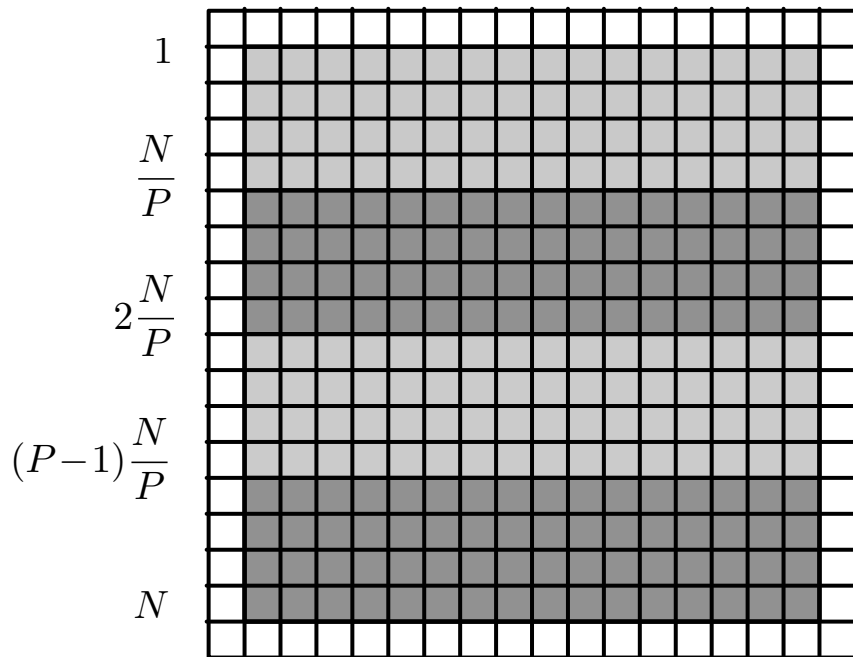
We are storing only  $N/P$  rows

We need to iterate 1 to  $N$

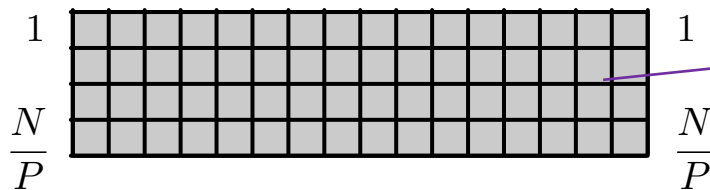
```
for (size_t i = K*N/P+1; i < (K+1)*N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```



# Global Decomposition

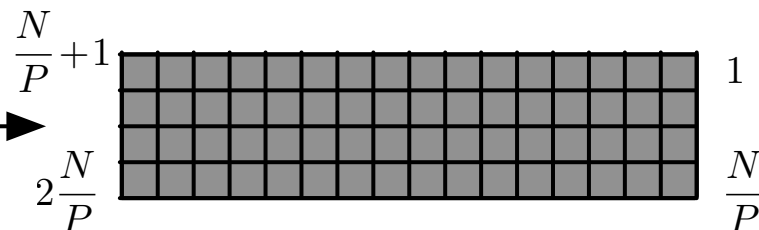


Global

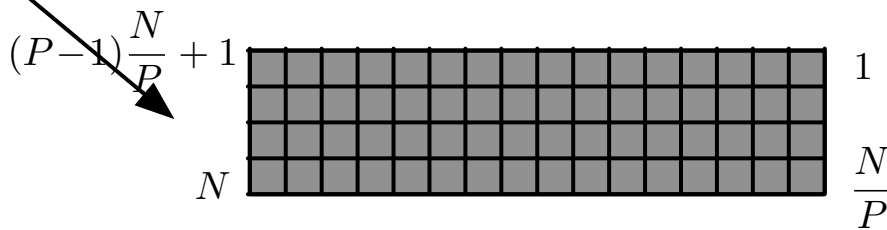


Local

Any obvious problems?



...

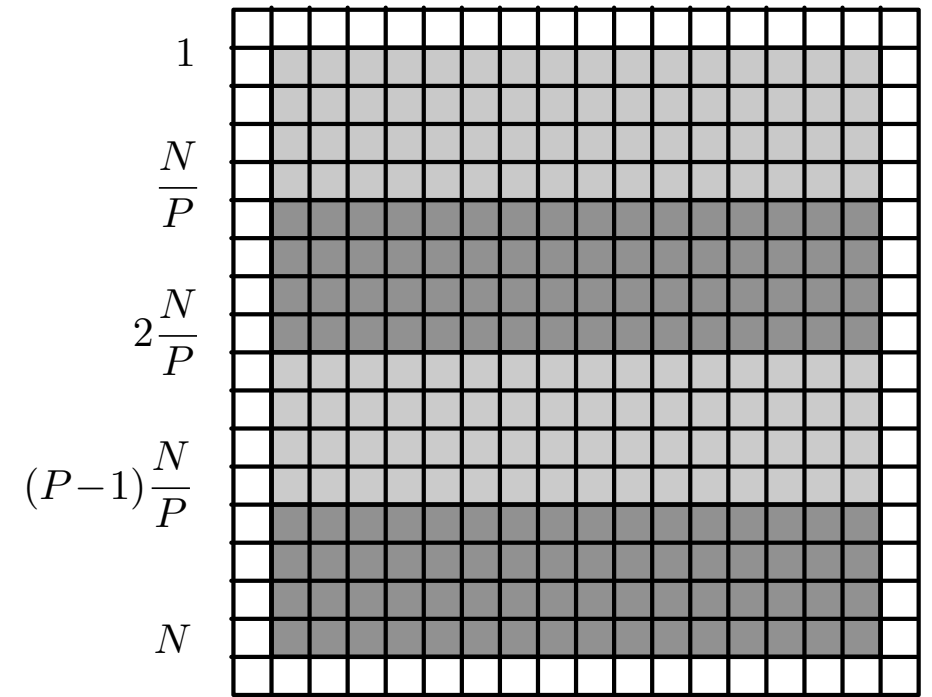
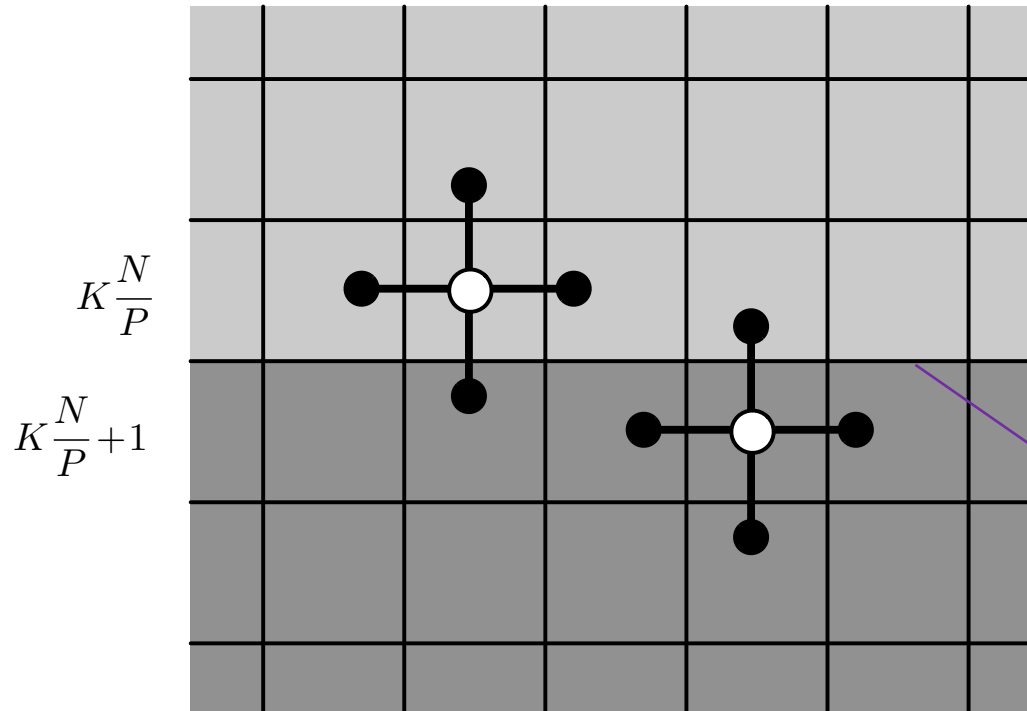


We need to iterate 1 to N

```

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

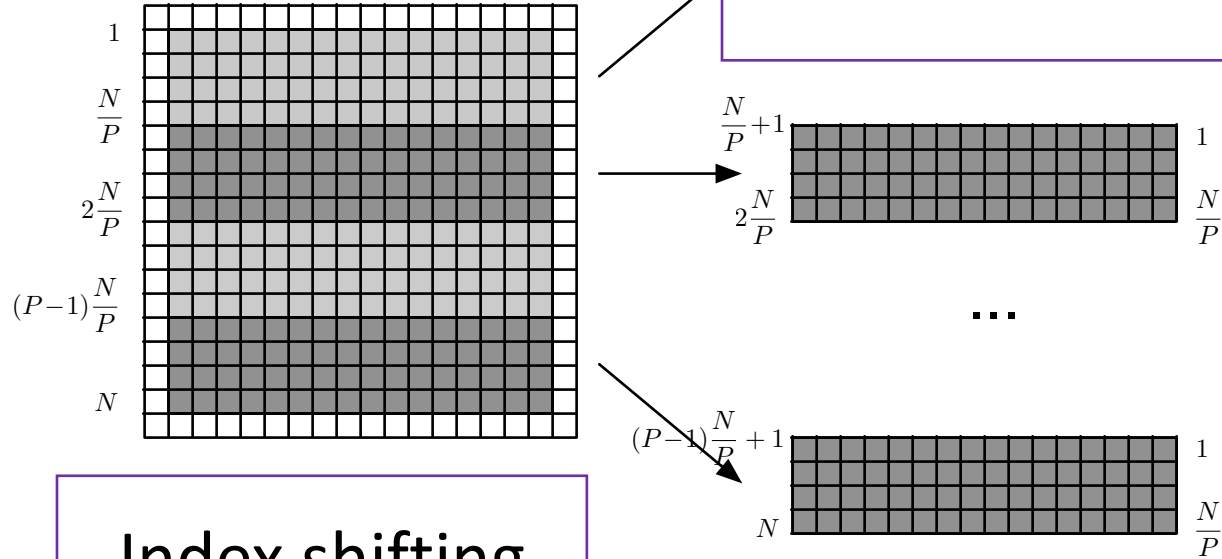


Data dependencies for stencil crosses partition boundary in original problem

# Decomposition

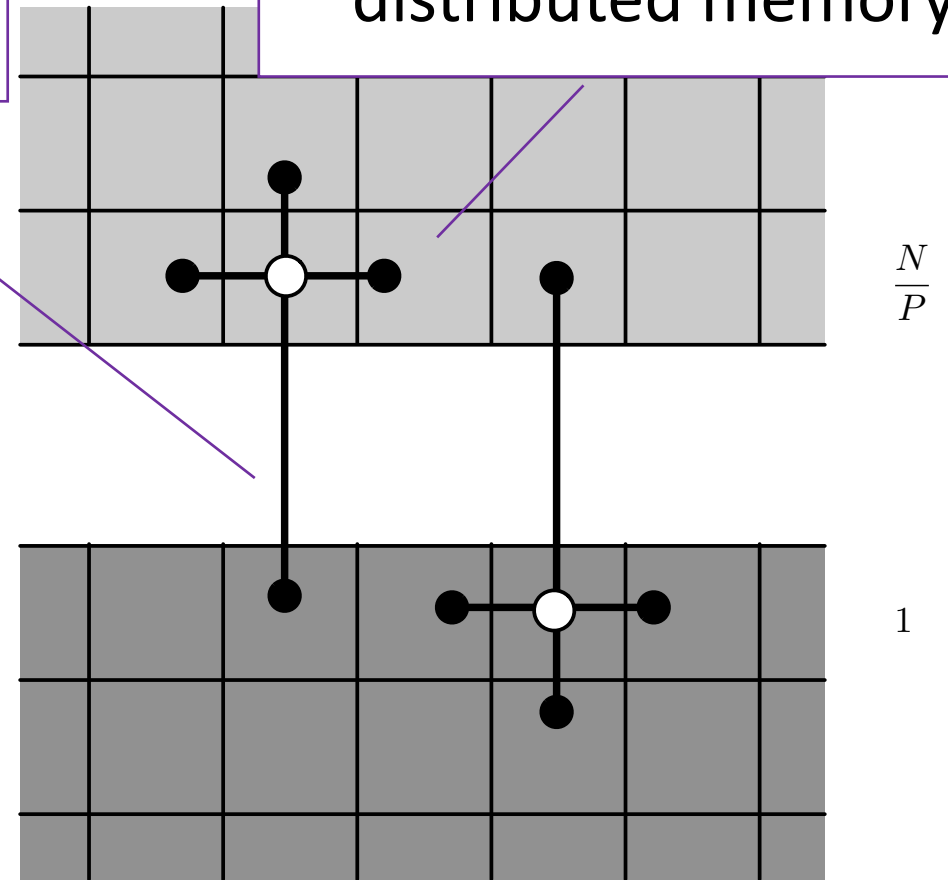
This is not a valid read

Which is a problem in distributed memory



$K \frac{N}{P}$

$K \frac{N}{P} + 1$



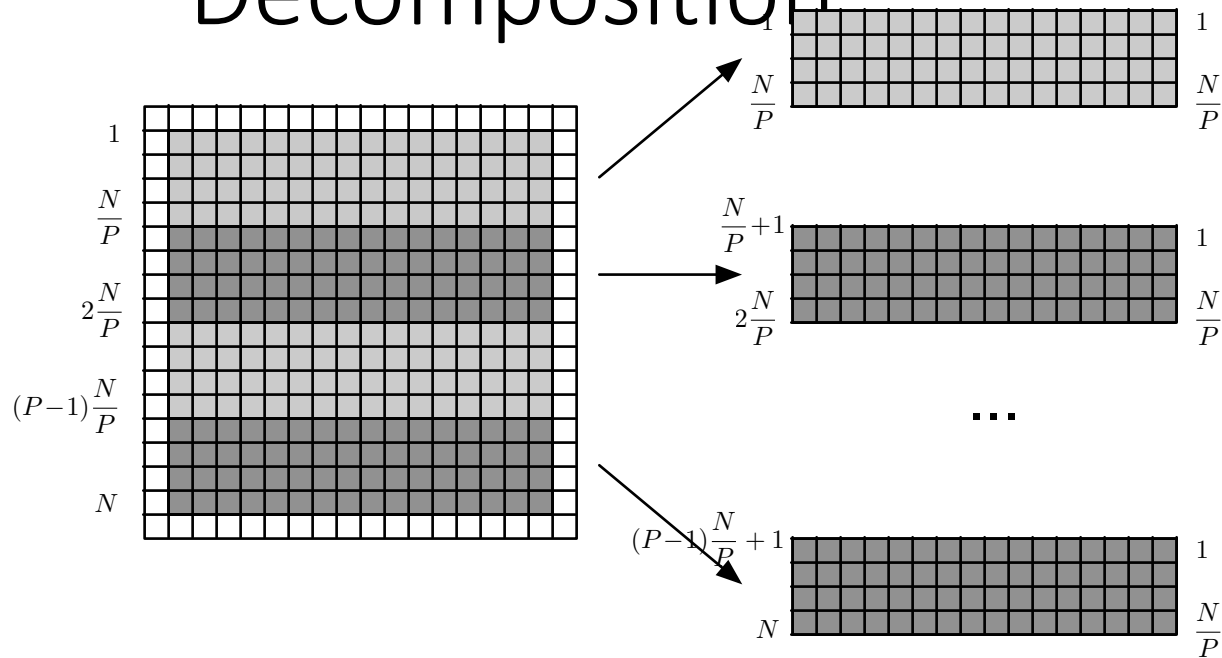
Index shifting doesn't help

This is just a program

It reads/writes local data, just like any other

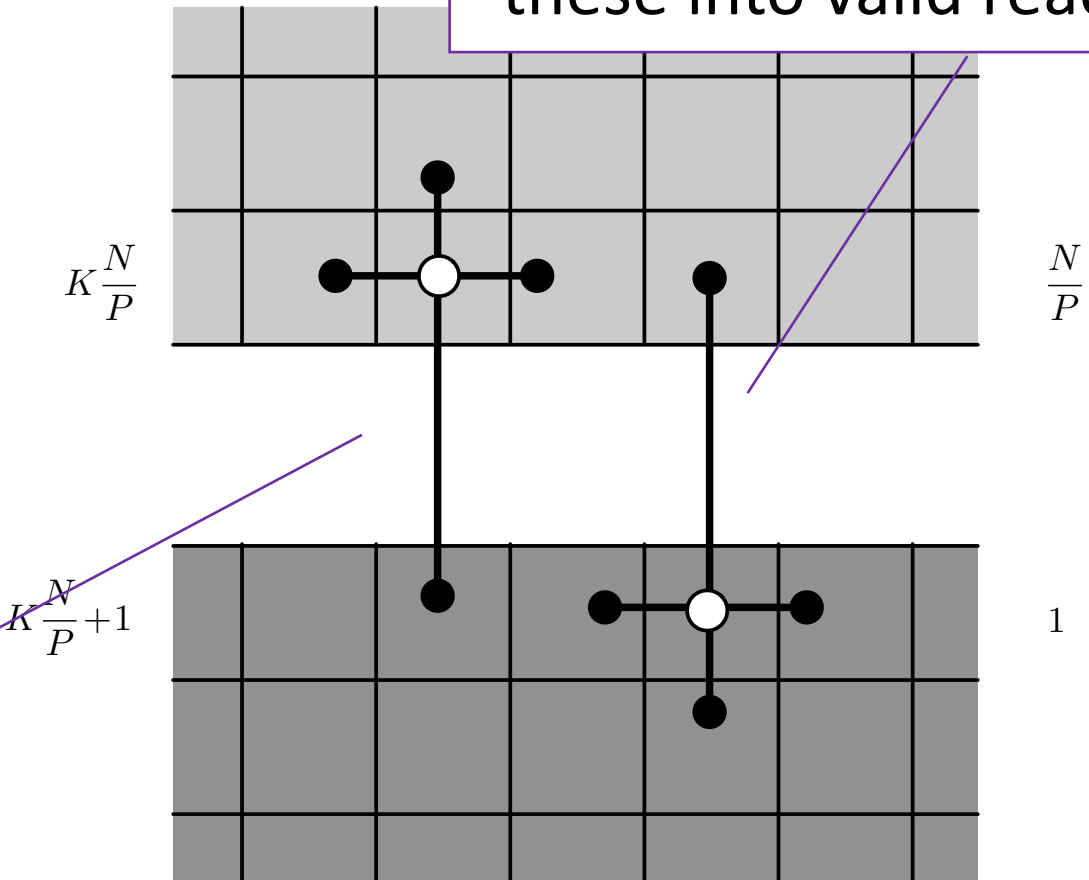
```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4
```

# Decomposition



And preserve the "as-if" property

We need to make these into valid reads

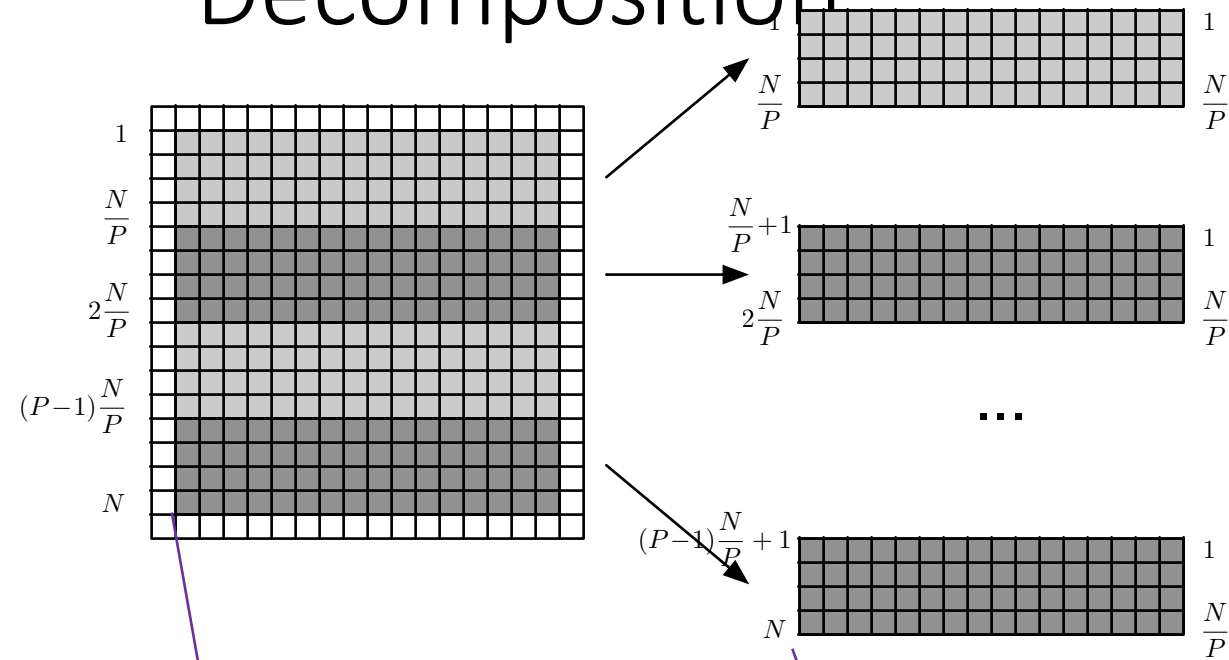


```

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;

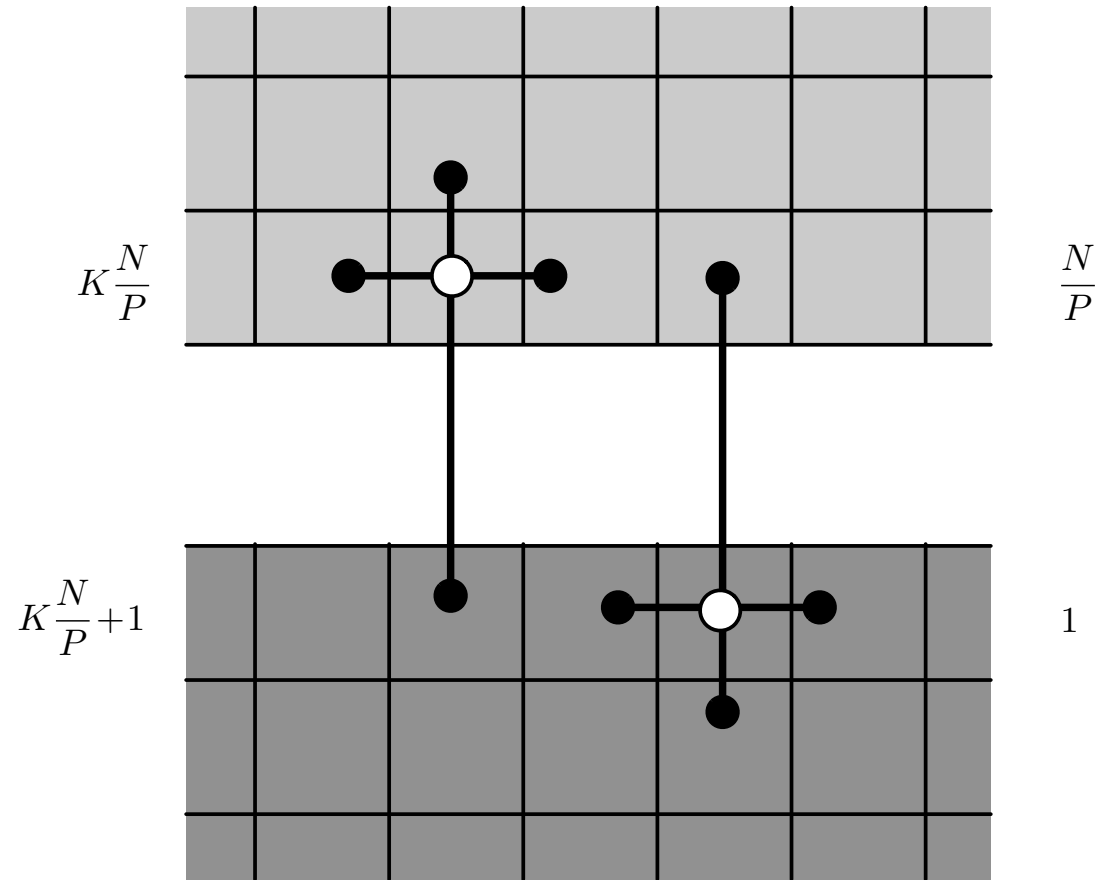
```

# Decomposition



Where did the boundary go

When we partitioned?



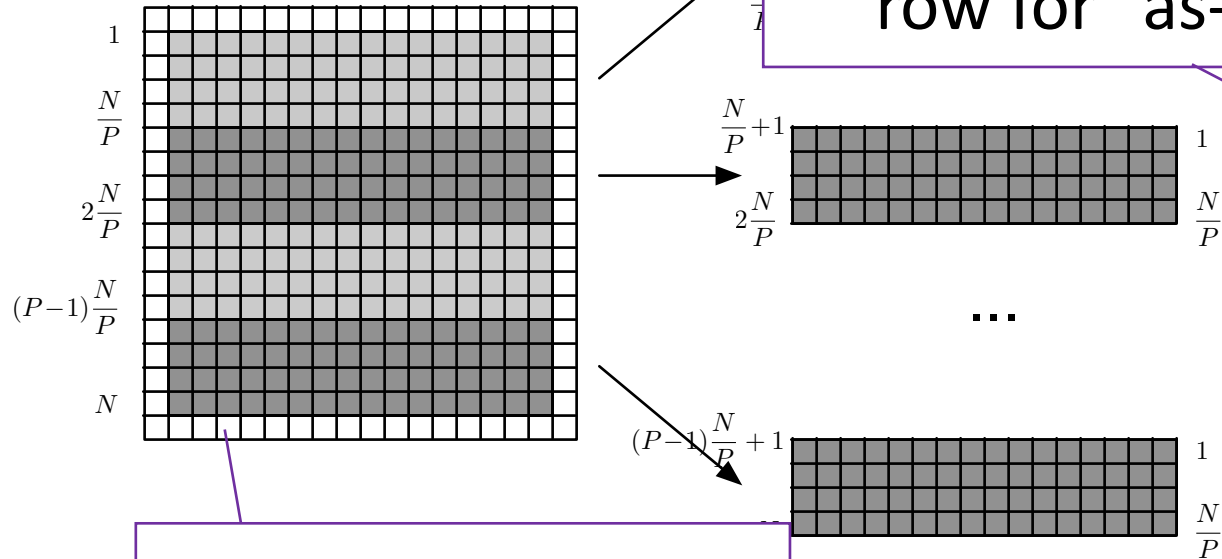
```

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;

```

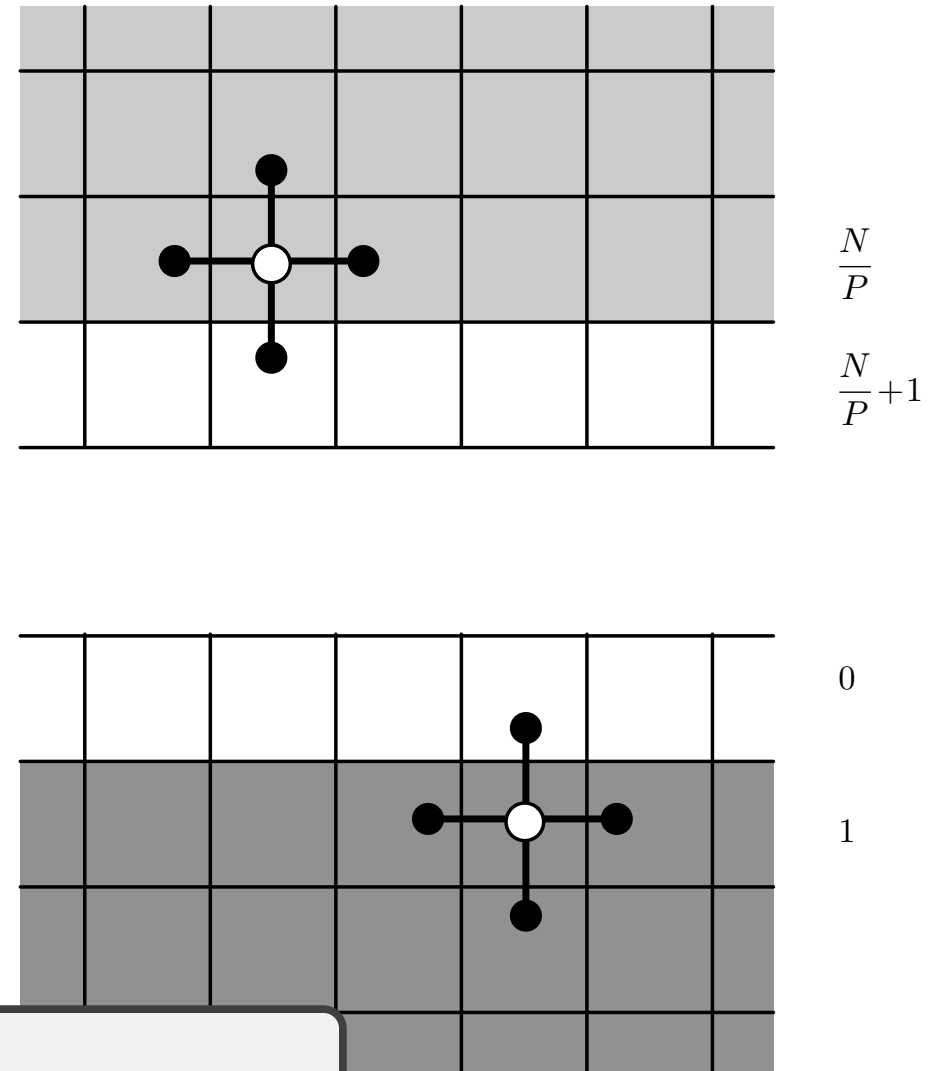
# Decomposition

We need an extra row for "as-if"



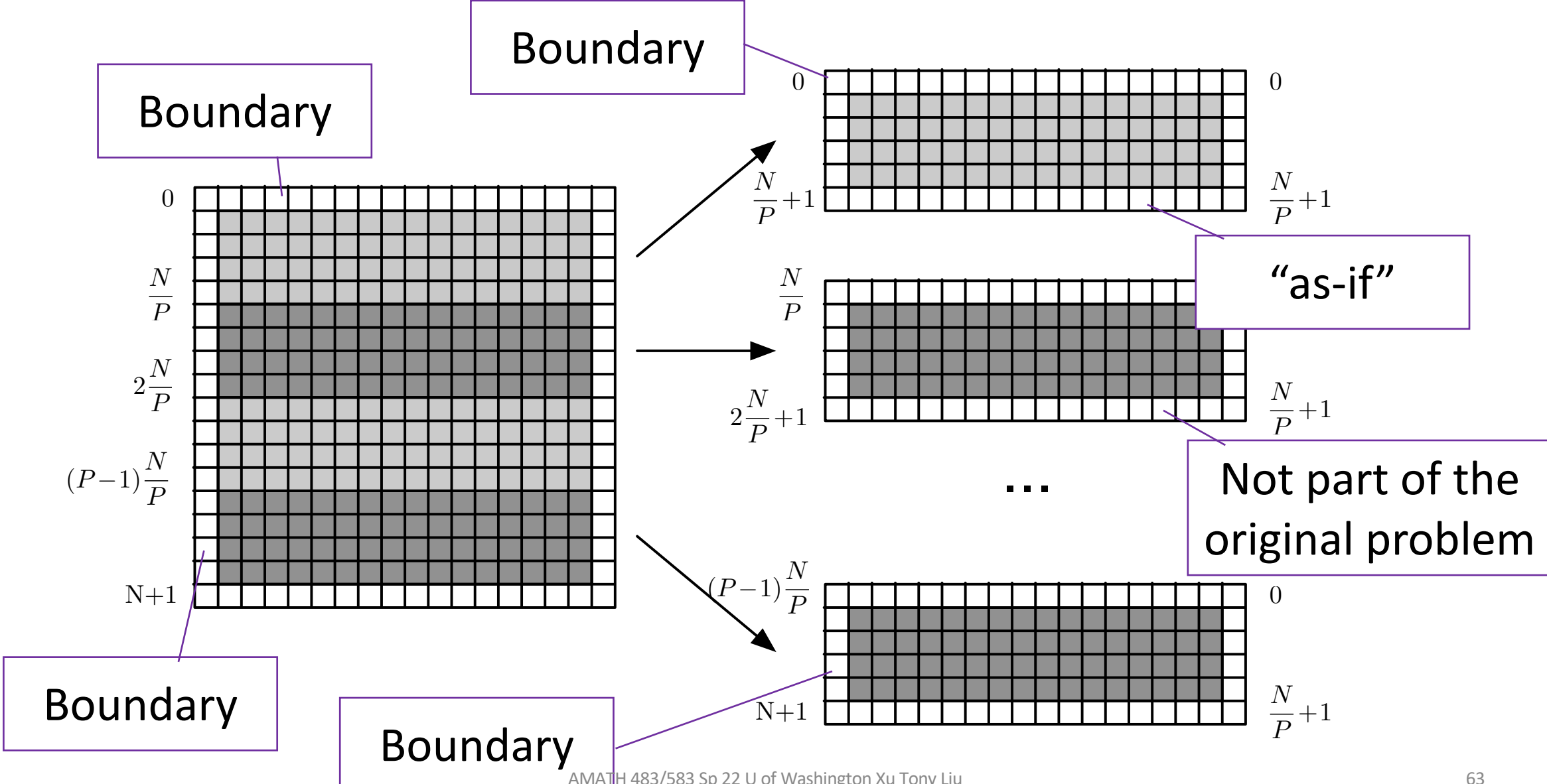
And an extra row (on some nodes) for the boundary

$K \frac{N}{P}$   
 $K \frac{N}{P} + 1$



```
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



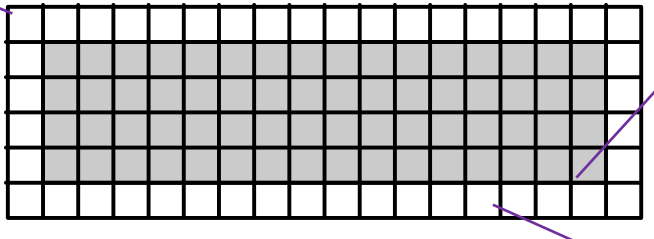
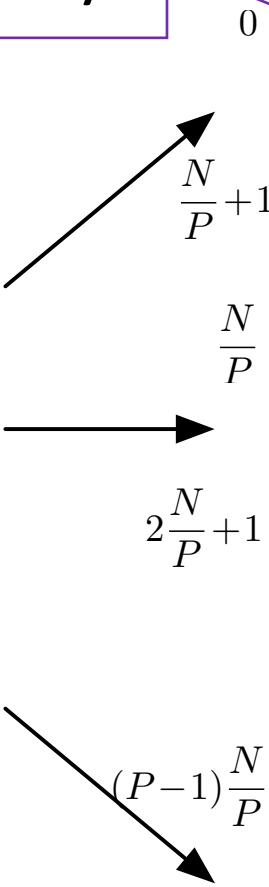
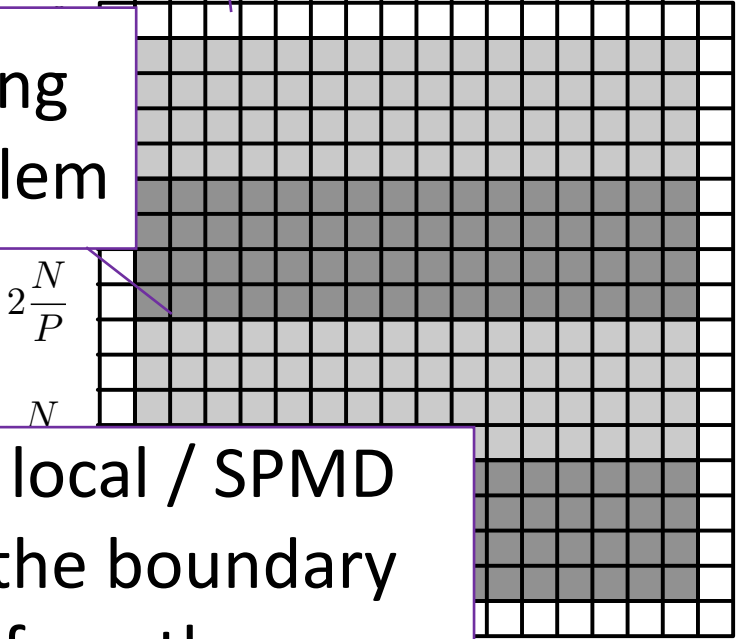
# Decomposition

Boundary

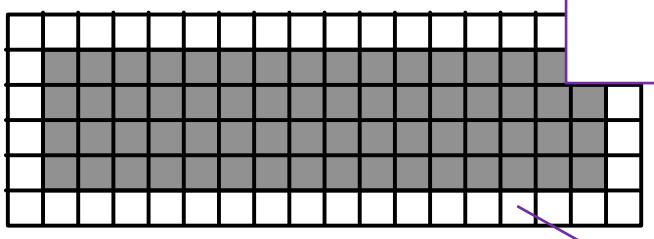
Boundary

One crucial difference

So solving this problem

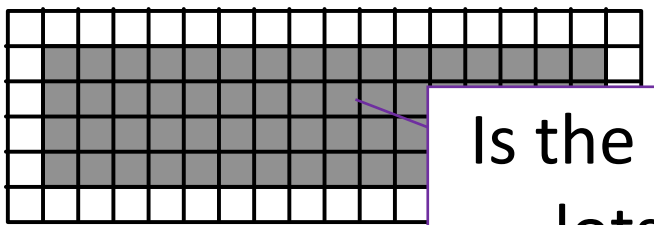


“as-if”



Not part of the original problem

...



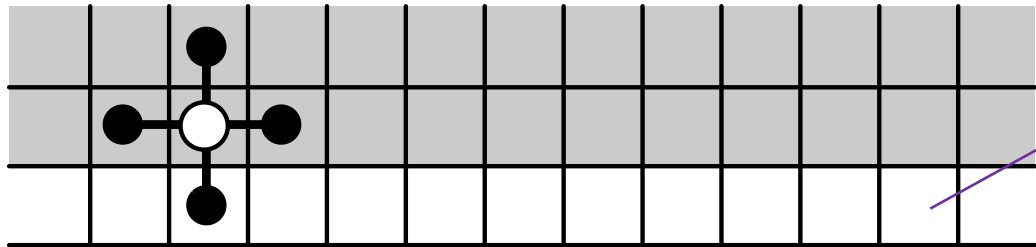
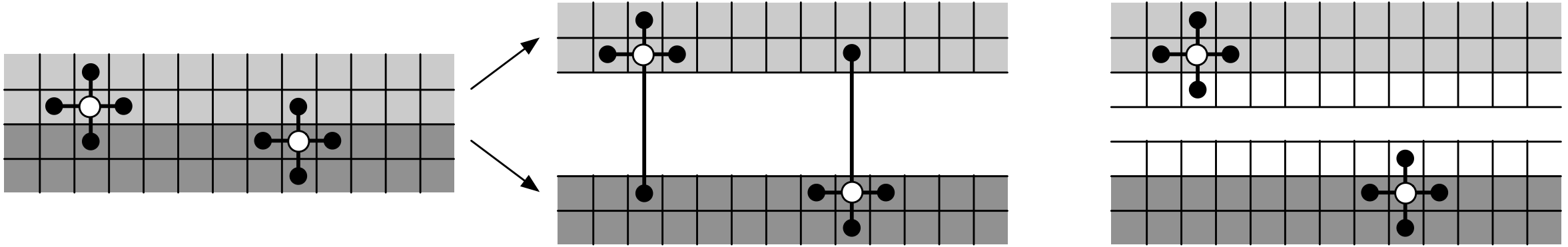
Is the same as solving lots of the same problem but smaller

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

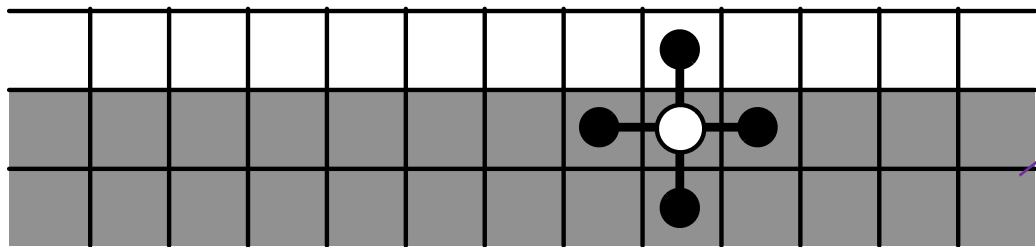
```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```



# As-If

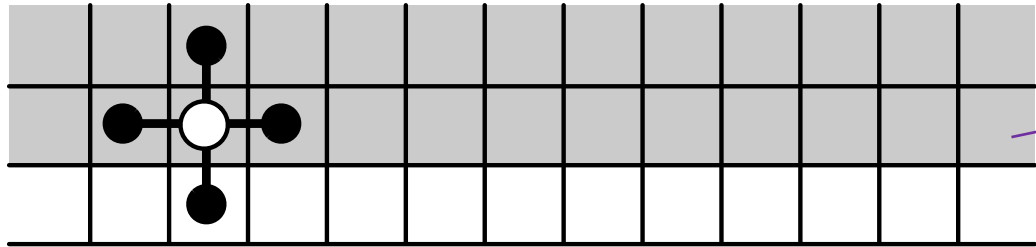
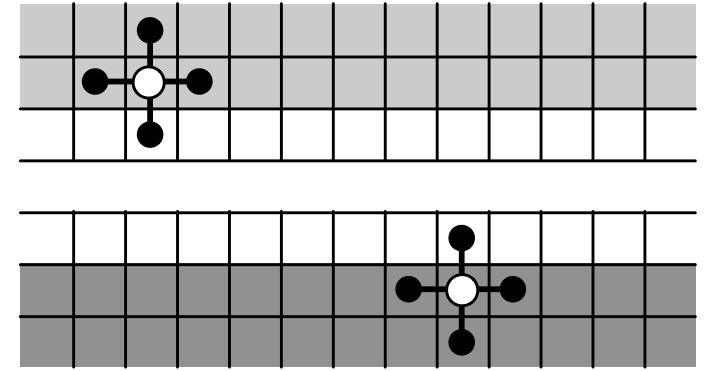
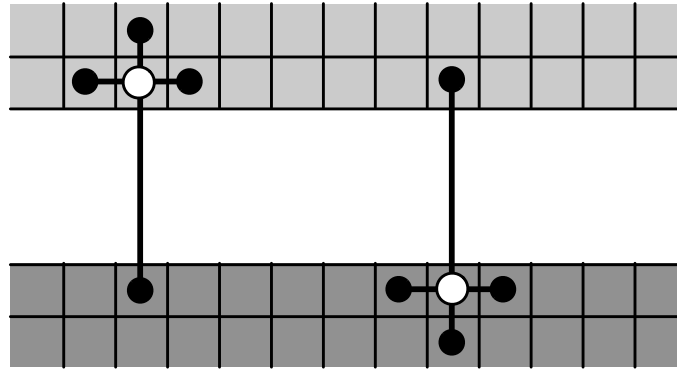
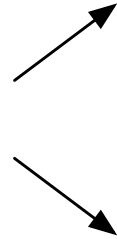
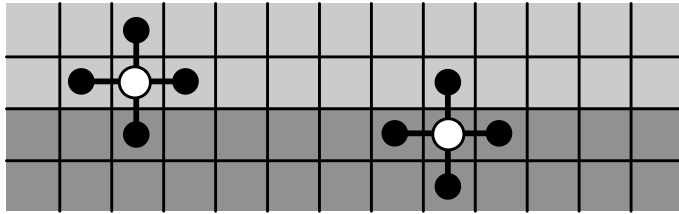


This row needs to be "as-if"

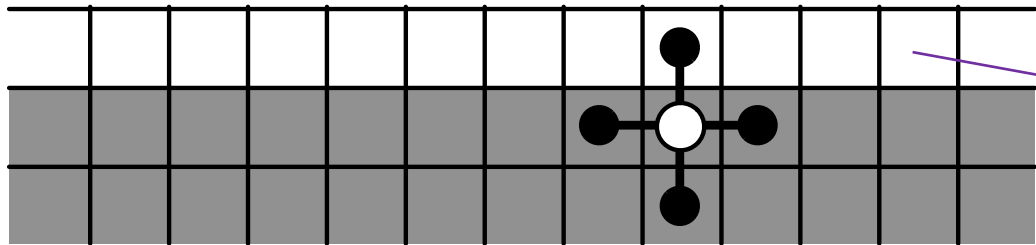


it were this row

# As-If



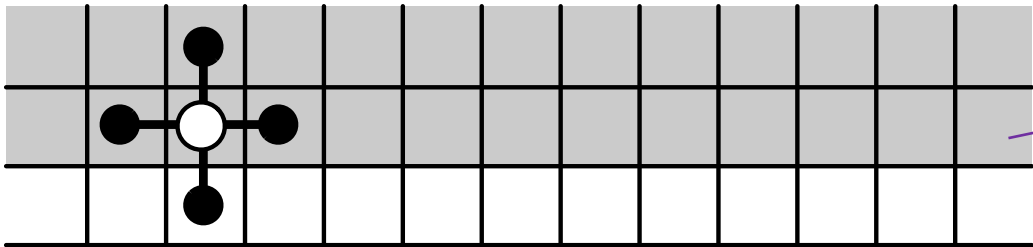
it were this row



And this row needs to be "as-if"

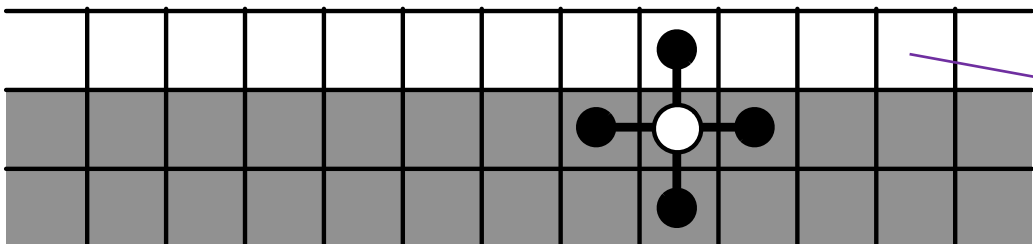
# As-If

```
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

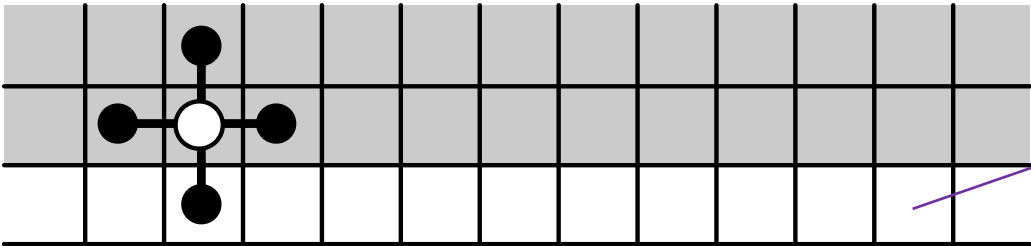


And this row needs  
to be "as-if"

# As-If

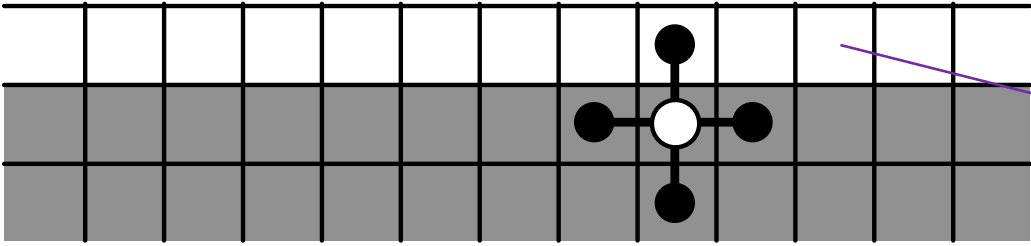
This is the computation

```
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;
```



Note these are not changed during an iteration

Only when it is read



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);  
}
```

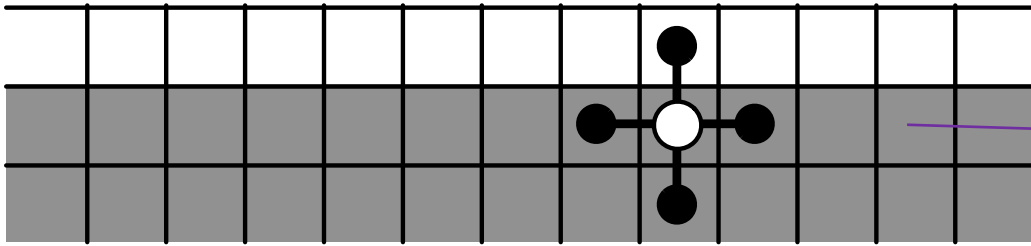
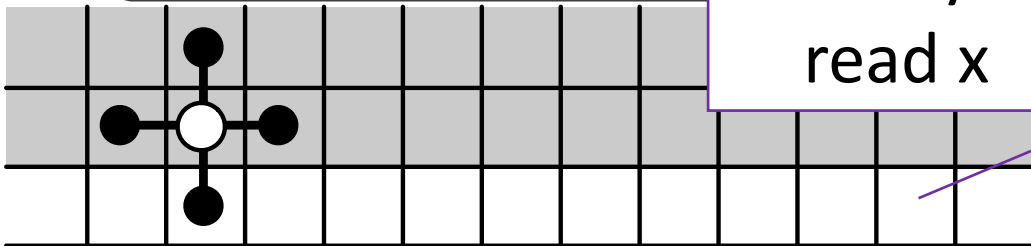
This is the entire program

Always write y

Always read x

Not changed during an iteration

Rows need to be as-if only during iteration



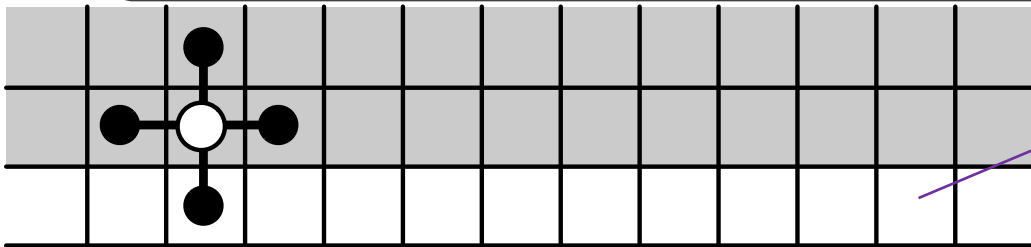
This changes only on every outer iteration (on the swap())

# As-If

Here is where we need to make as-if true

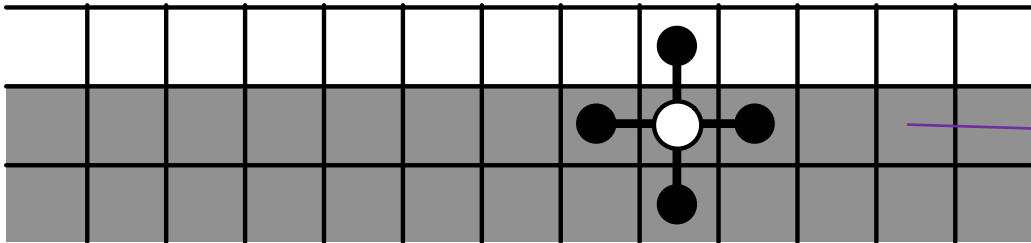
This is the entire program

```
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);  
}
```



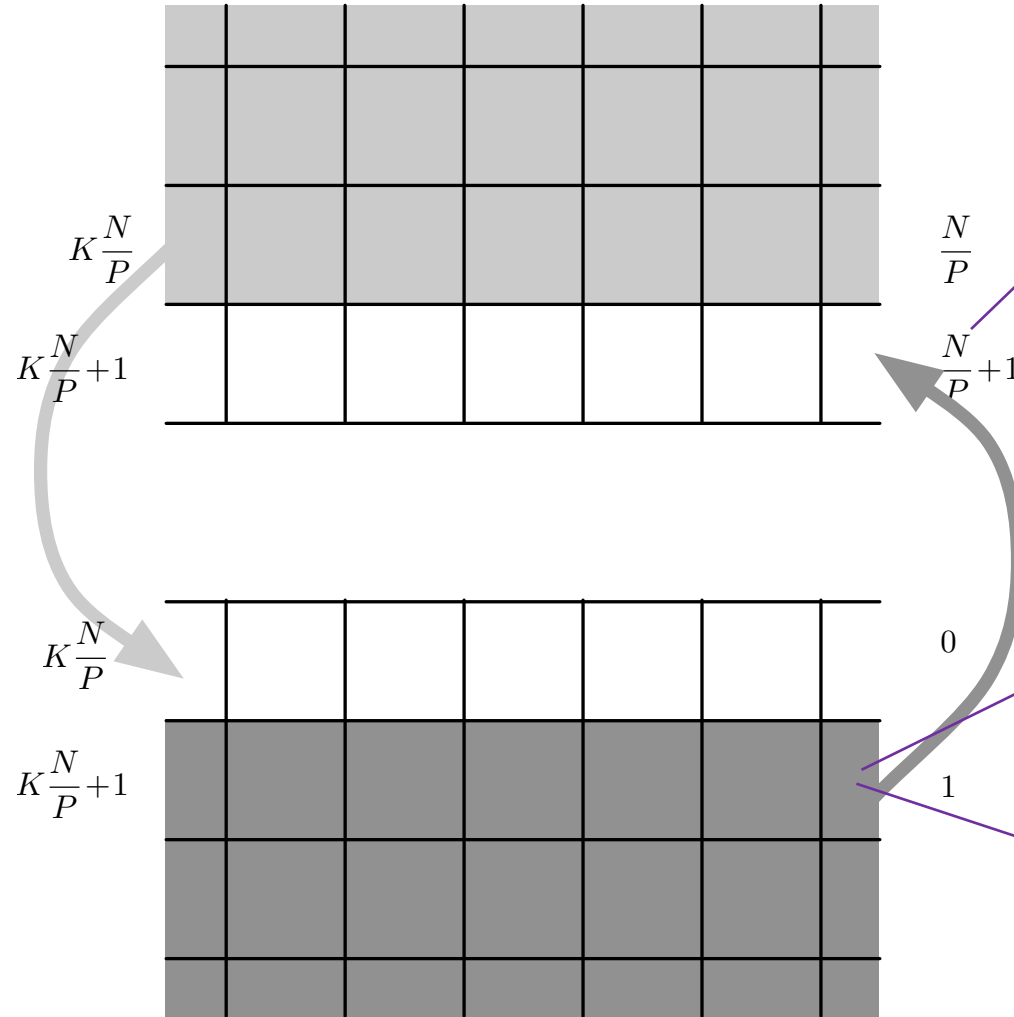
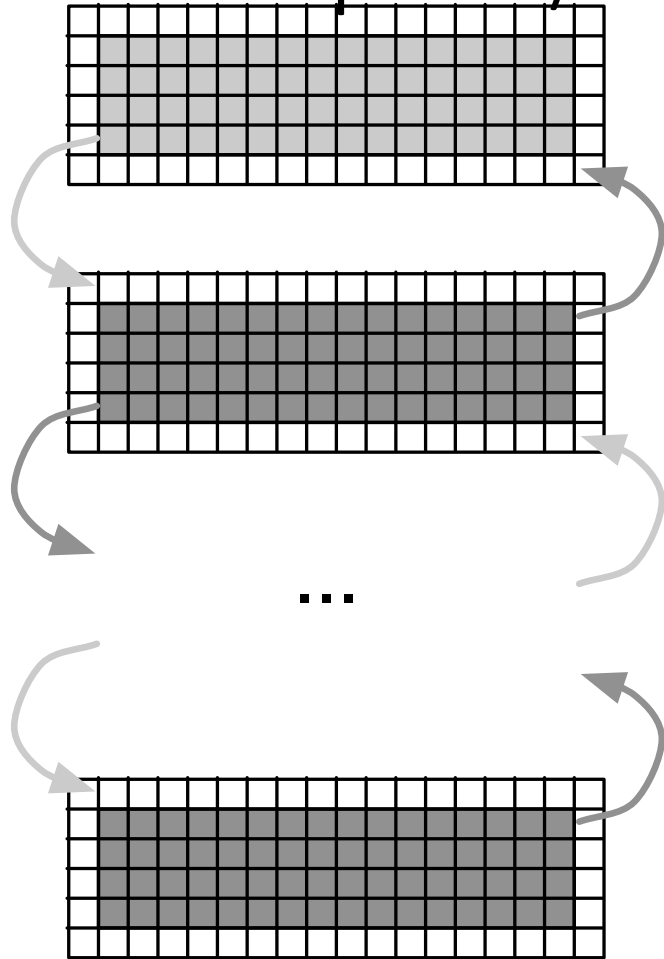
Not changed during an iteration

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

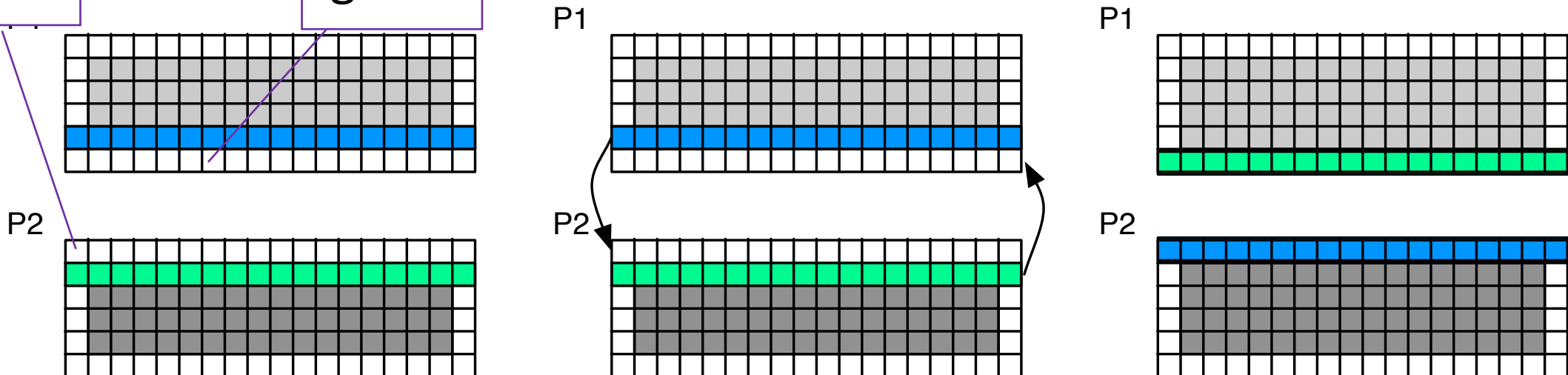
# Compute / Communicate

```
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  
}
```

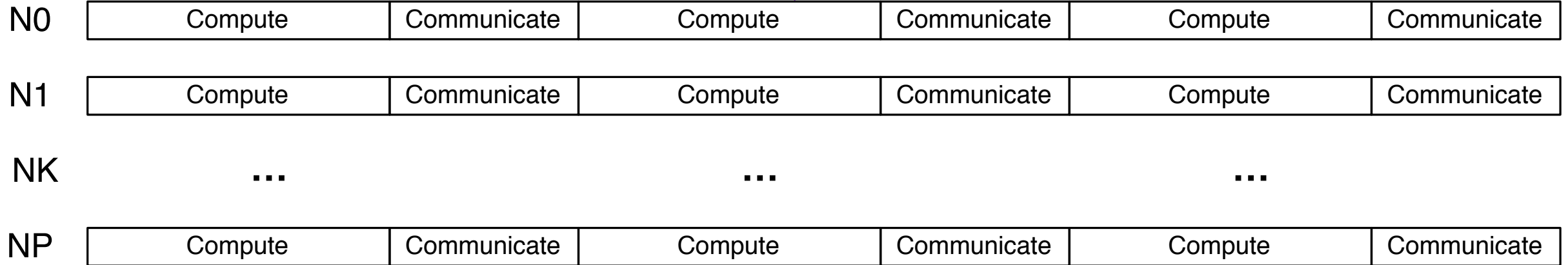
Compute

This is an almost  
universal pattern

Communicate

# Compute / Communicate

“Bulk Synchronous Parallel” (BSP)



This is an almost universal pattern

Processors are still only loosely coupled

But the compute / communicate pattern keeps them synched in a bulk sense

Time

# 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

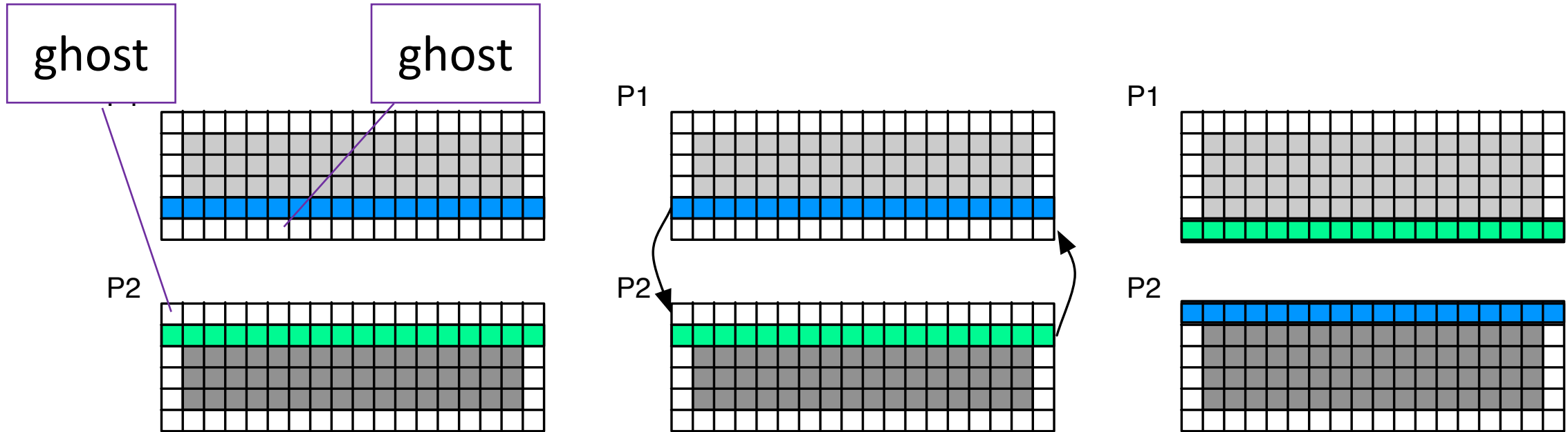
# Parallel Jacobi Solver

```
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

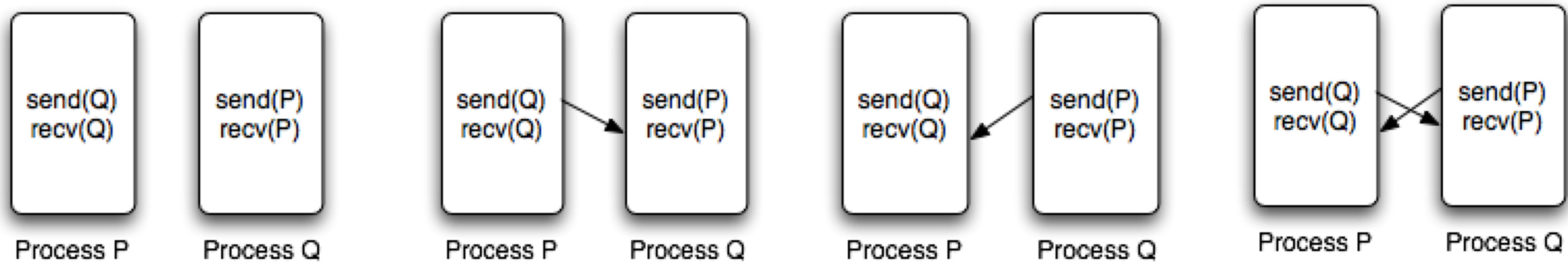


```
MPI_Send( ... ); // to upper neighbor  
MPI_Send( ... ); // to lower neighbor  
MPI_Send( ... ); // from lower neighbor  
MPI_Send( ... ); // from upper neighbor
```

Works?

# Updating ghost cells

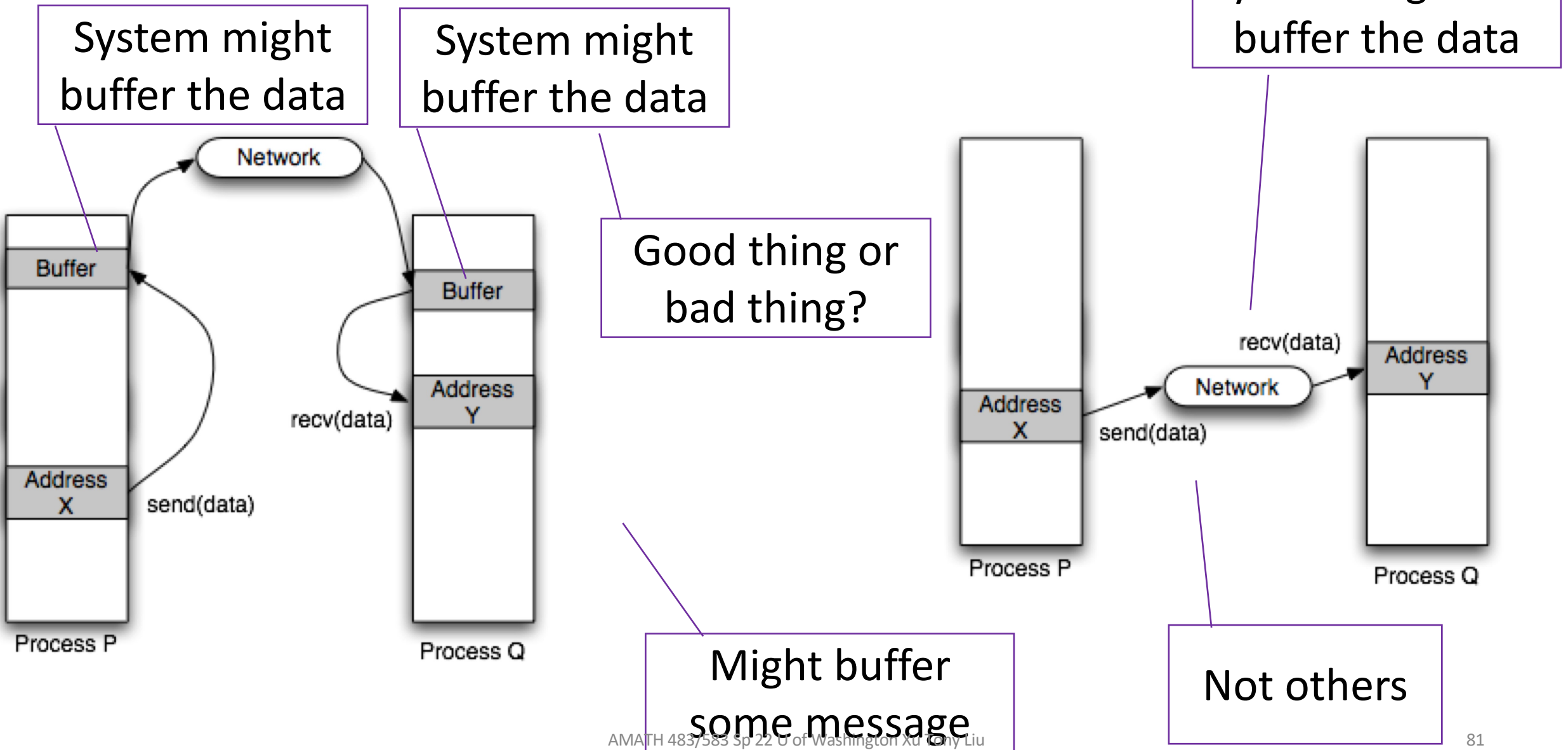
- What happens with following sequence of communication operations?



- 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?



# 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
```

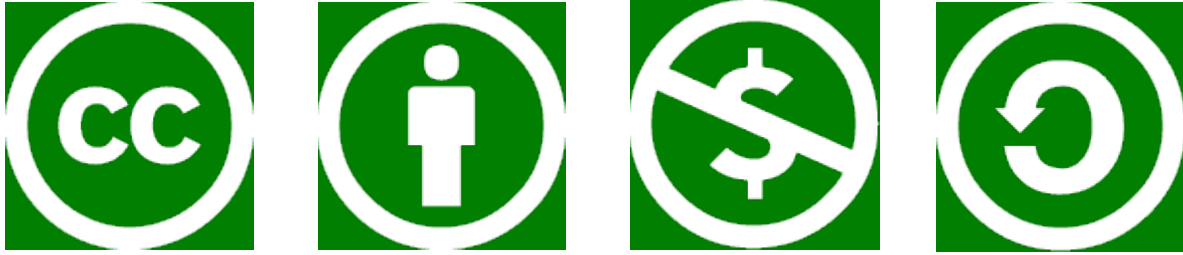
- Blocking receive
- Semantics: Blocks until message is received. On return from call, buffer will have message data

# 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



© Andrew Lumsdaine, 2017-2022

Except where otherwise noted, this work is licensed under

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

