

AMATH 483/583

High Performance Scientific Computing

Lecture 18:

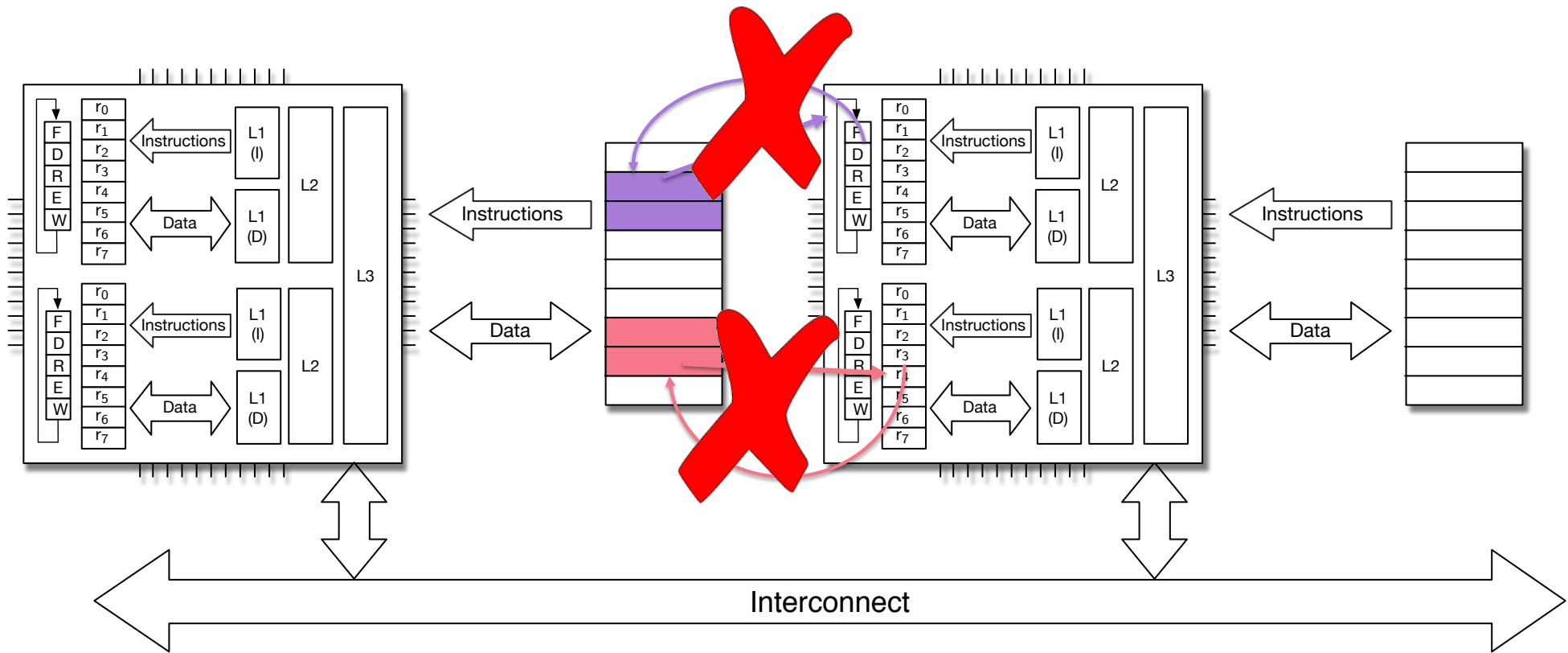
Message Passing w/CSP/SPMD, MPI

Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA

Overview

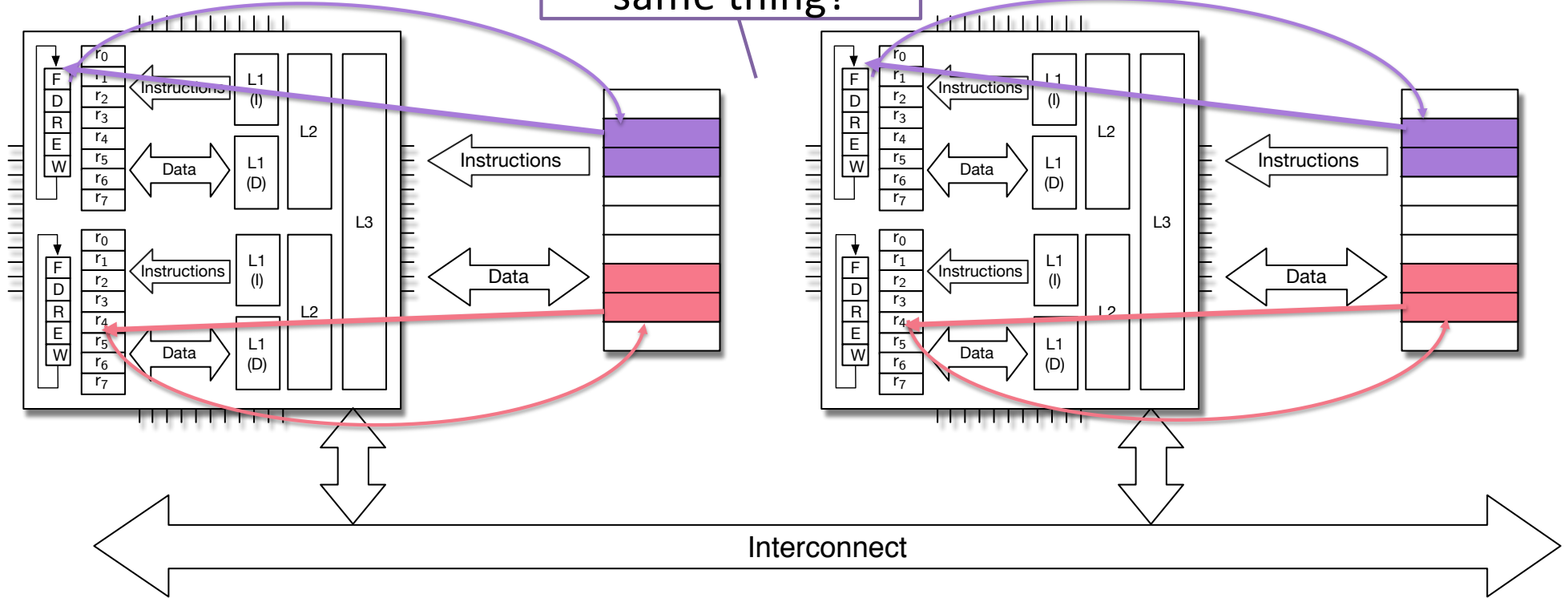
- SPMD / CSP recap
- MPI mental model recap
- Basic MPI recap
- Laplace's equation on a regular grid

Distributed memory



Distributed memory

Do we want these doing the exact same thing?

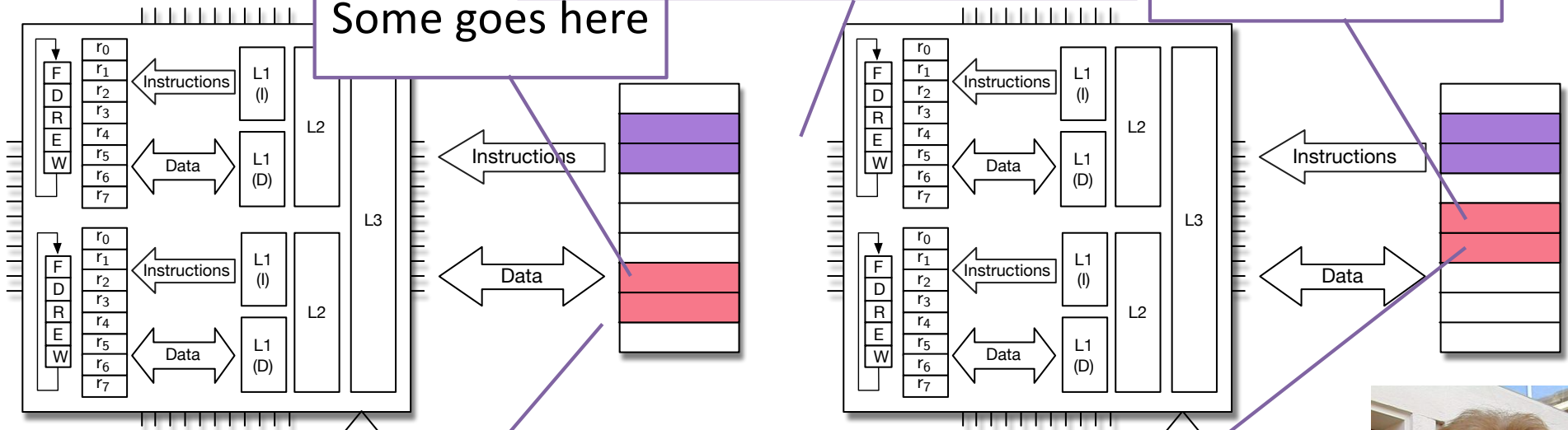


Distributed memory

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

Some goes here

Some goes here

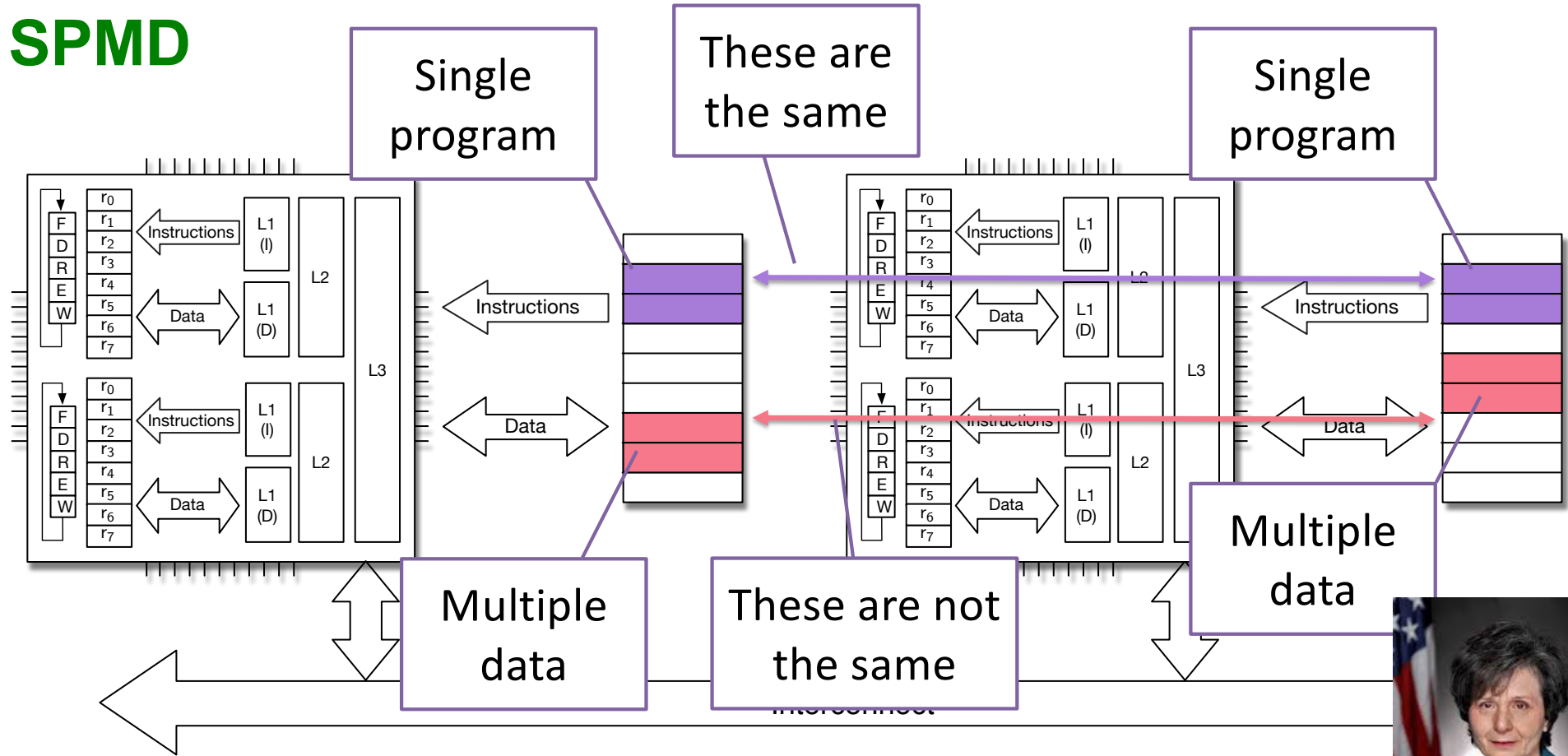


The union of the two should be the whole problem

“Collectively exhaustive”



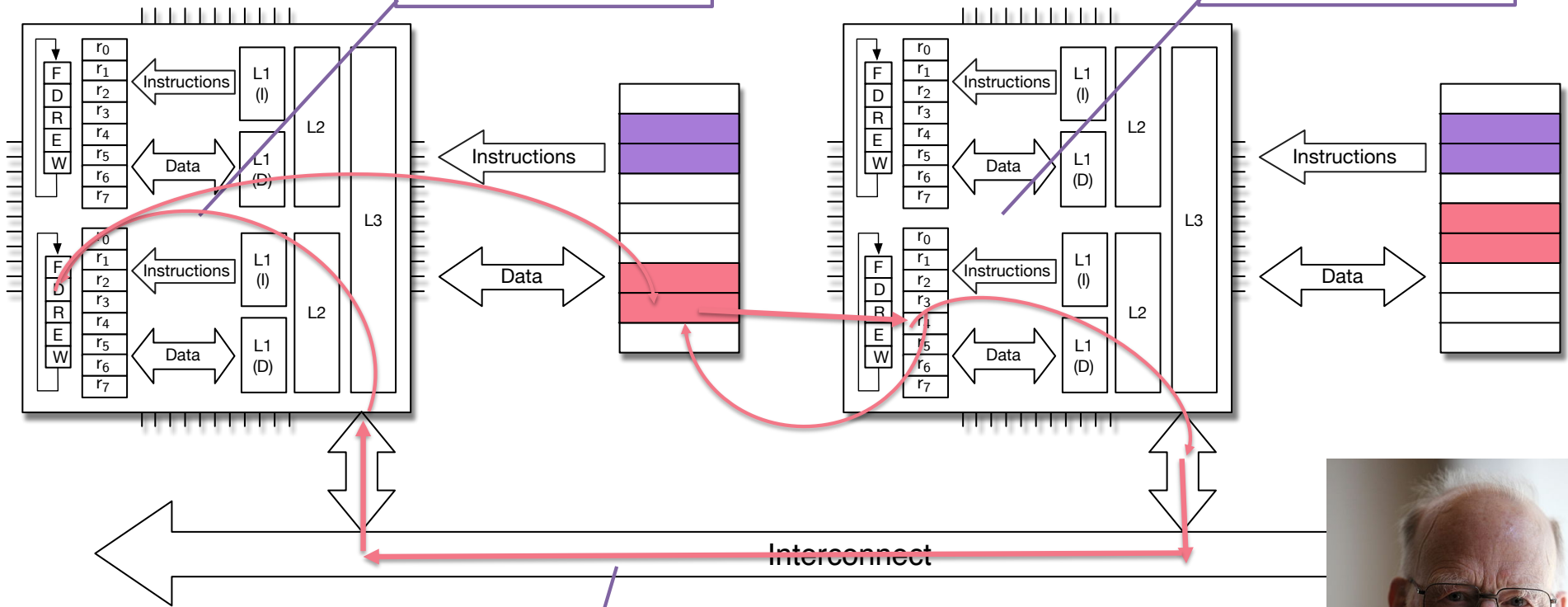
SPMD



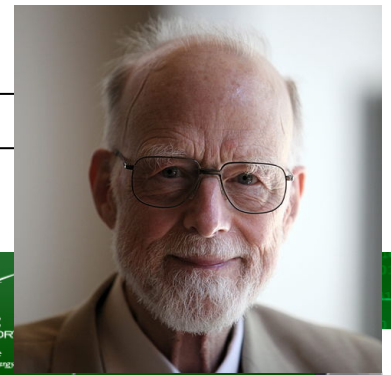
Communicating Sequential Processes (CSP)

“Sequential”
process

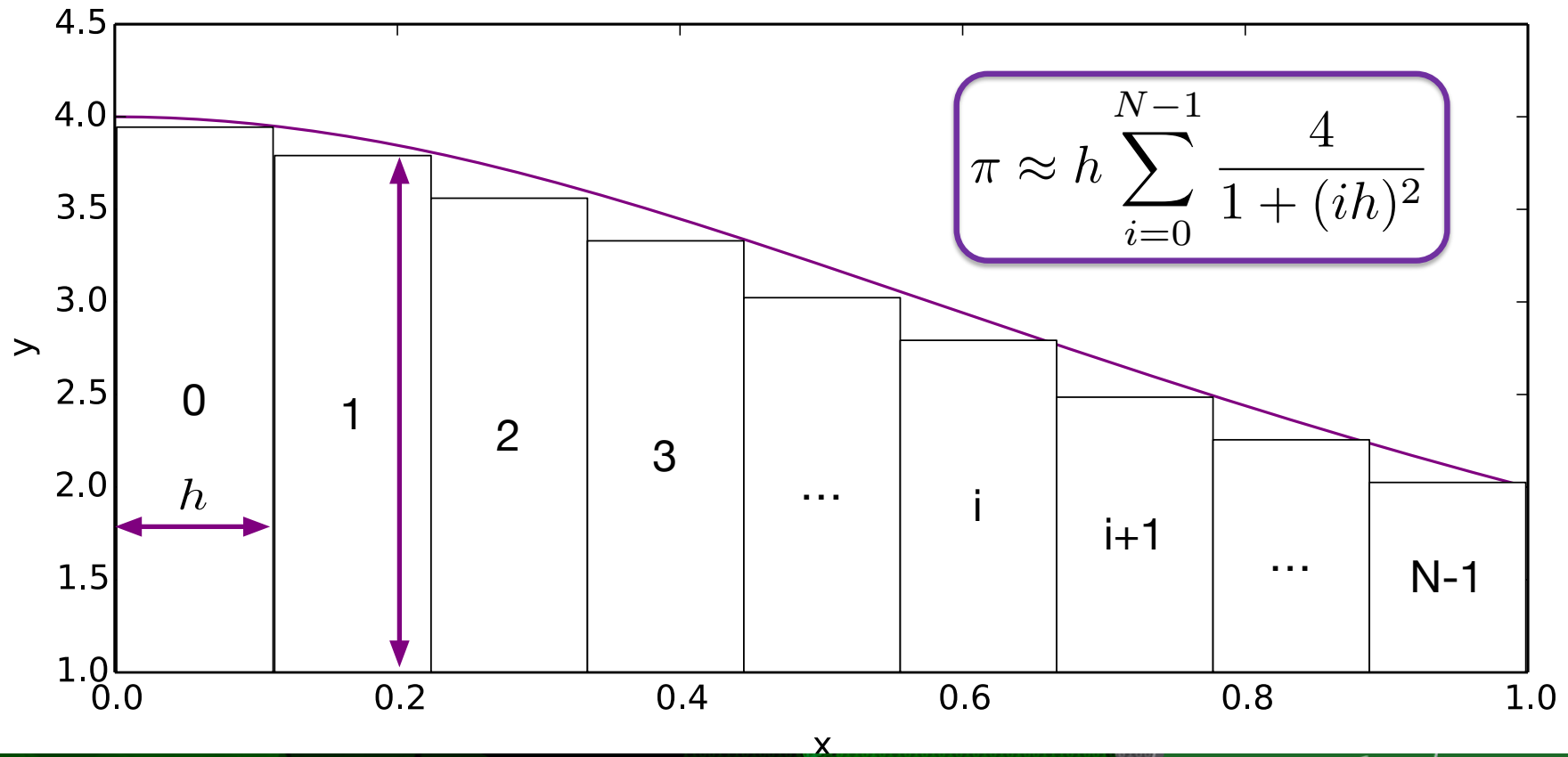
“Sequential”
process



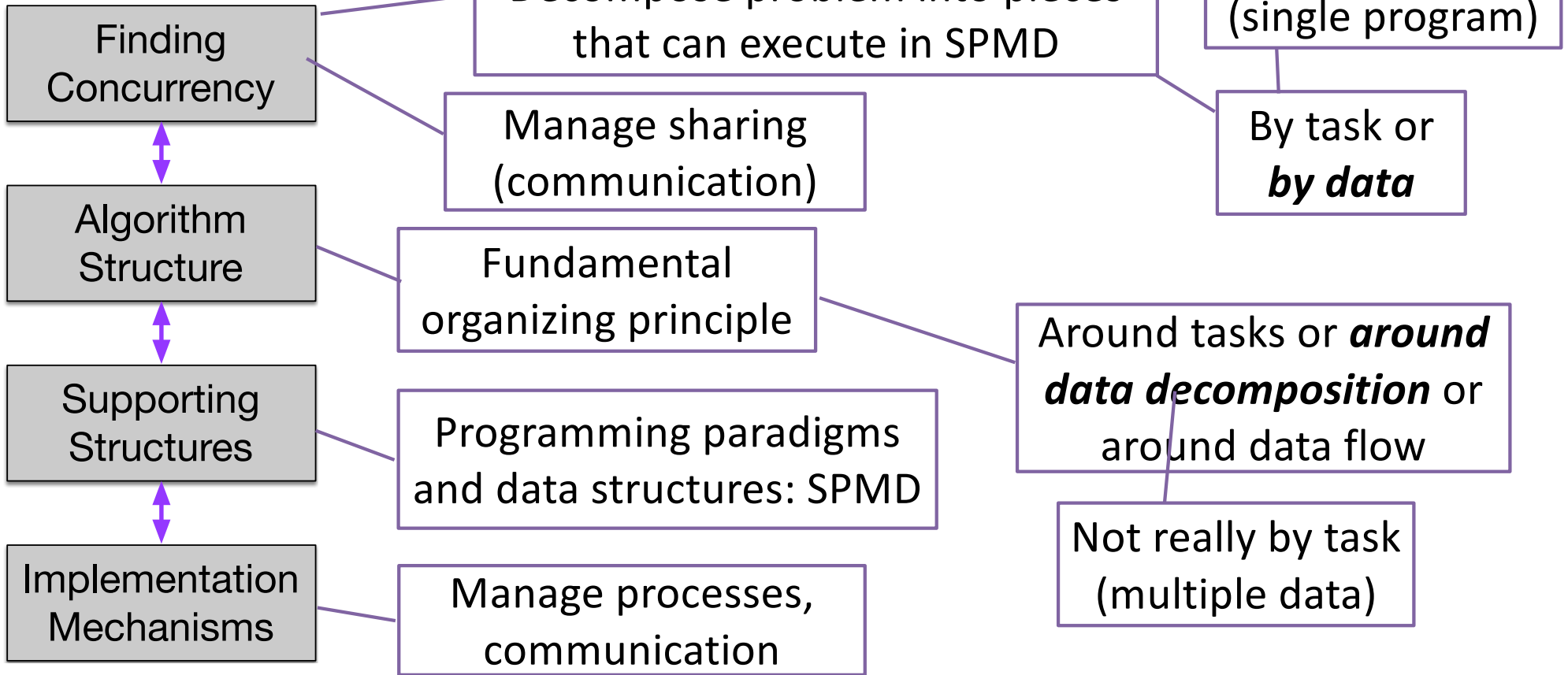
Communicating



Numerical Quadrature



Parallelization strategy: SPMD version



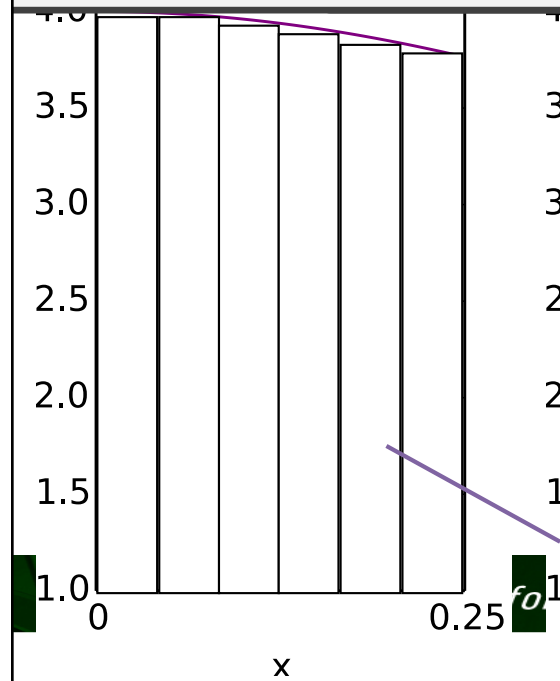
Finding Concurrency

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

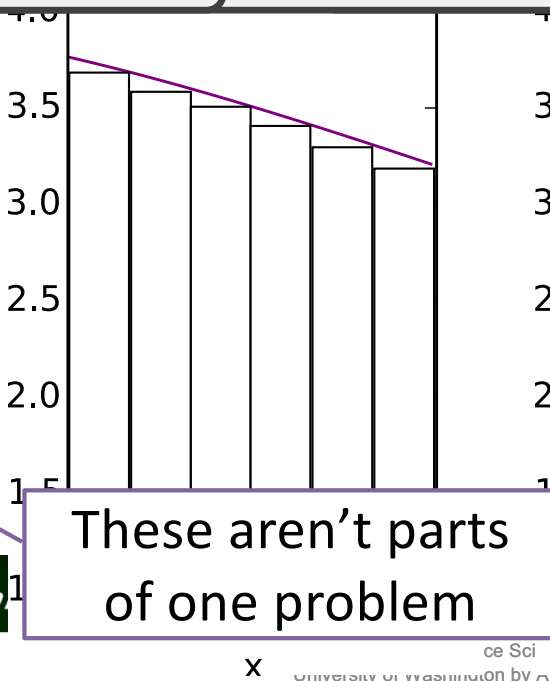
```
int i = 0; i < N/4; ++i) {  
    += h * 4.0 / (1 + i*h*i*h);
```

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

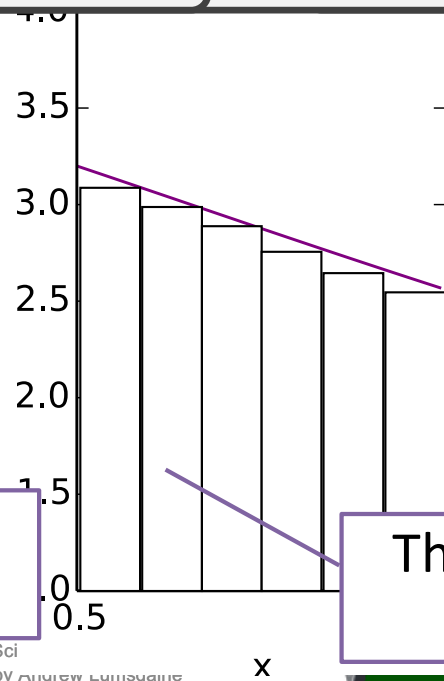
```
1/2; i < 3*N/4; ++i) {  
    0 / (1 + i*h*i*h);  
N/4; i < N  
    / (1 + i*
```



These aren't parts of one problem



They are separate problems



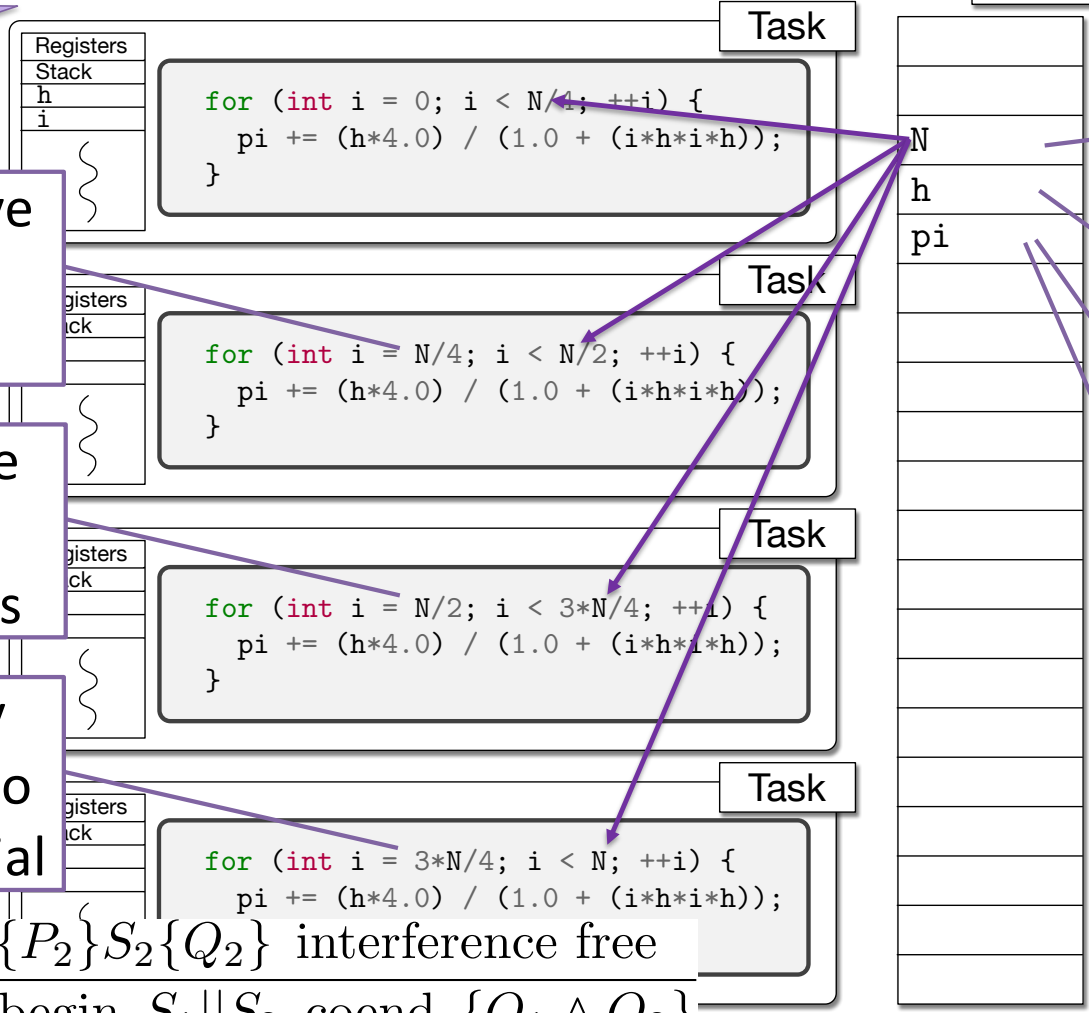
Shared Memory Parallelism

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

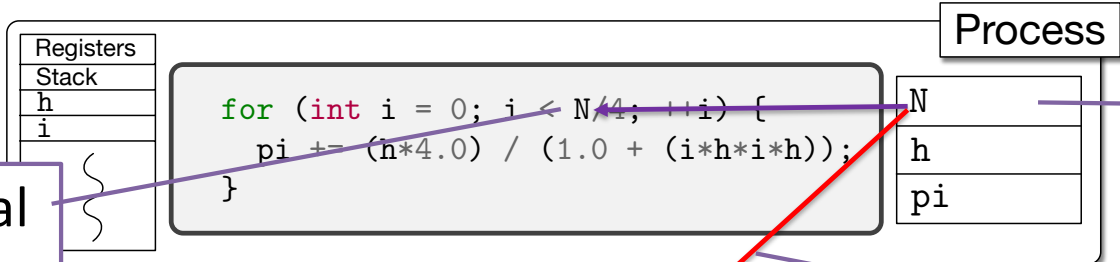
Similarly pi

At least for reading

(Have to deal with race when writing)

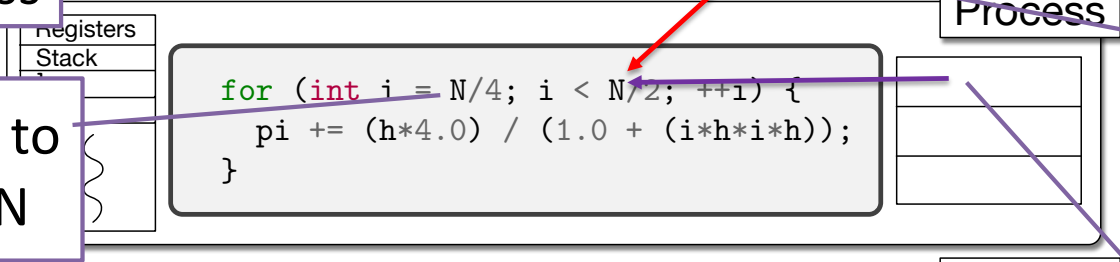
$\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\}$ interference free

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



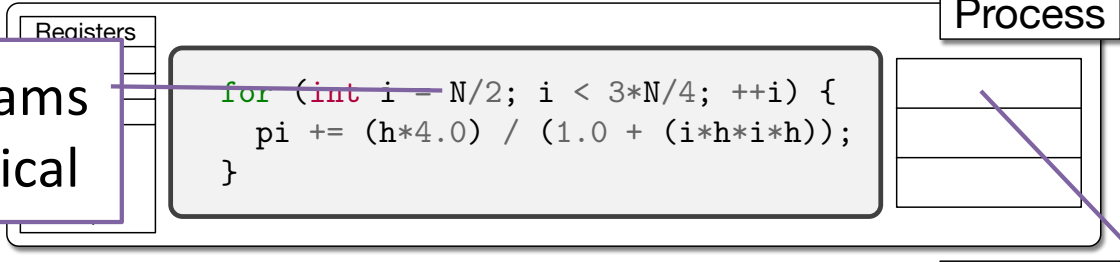
This N is local to this process

This N is local to this process



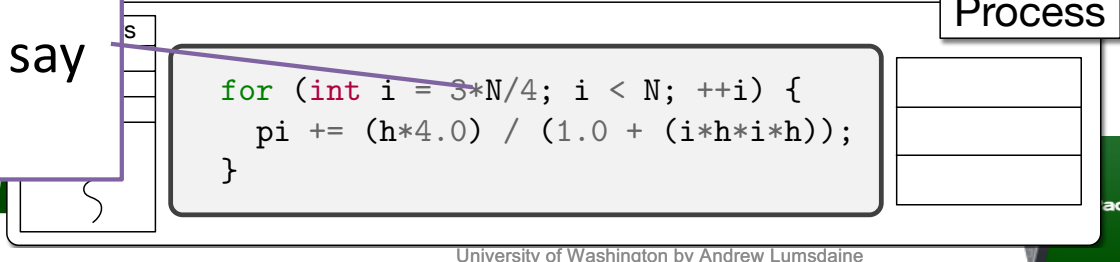
But we need to read *some* N

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?

Registers
Stack
h
i
⋮

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

Process		
N	↓	↓
h	↓	↓
pi	↓	↓
	↓	↓

To read the "right" N

Registers
Stack
h
i
⋮

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

Process		
N	↓	↓
	↓	↓
	↓	↓
	↓	↓

Copy to each 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));
}
```

Process		
N	↓	↓
	↓	↓
	↓	↓
	↓	↓

Copy to each process

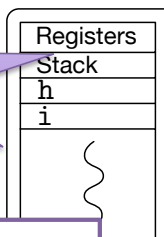
Registers
Stack
h
i
⋮

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

Process		
N	↓	↓
	↓	↓
	↓	↓

Copy to each process

Very Important Slide!!



```

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

```

Process
N
h
pi

Because they are reading ***copies of N***

Threads have the same value for N

```

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

```

Process
N
h
pi

It is ***as if*** they were the same N

And if these are all the same values

```

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

```

Process
N
h
pi

Similarly h, pi

It is exactly equivalent to the sequential

```

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

```

Process
N
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));
    }

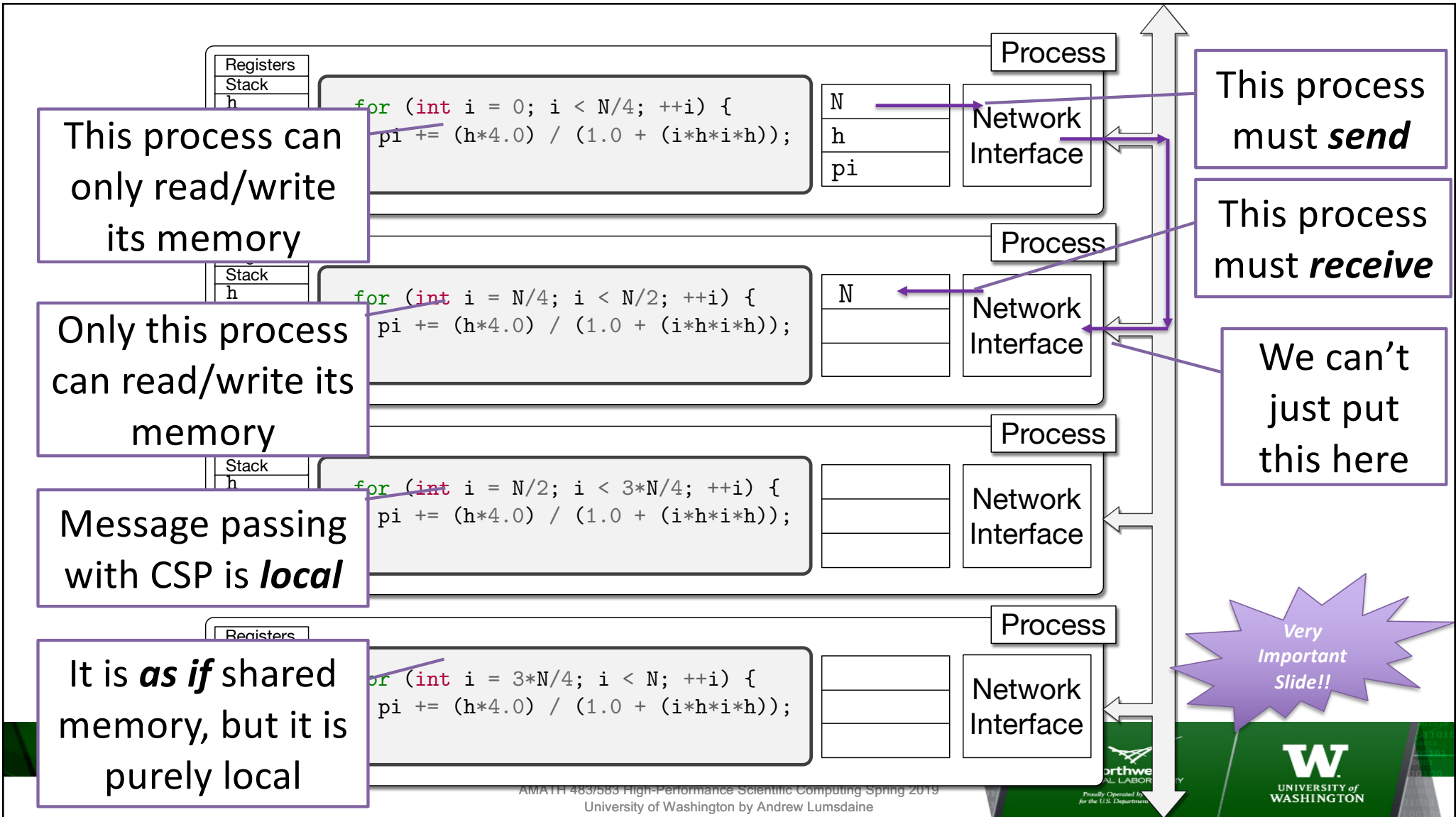
    MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE, MPI::SUM, 0);

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

    MPI::Finalize();

    return 0;
}
```

id 0 is root



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;
}
```

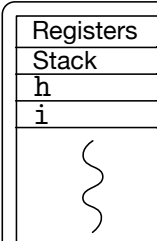
All copies call this communication function

And intervals gets copied to all processes

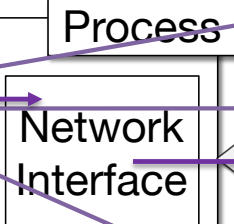
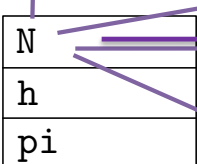
Only because each process calls this

First question: What are we sending?

“N” only has meaning in source code

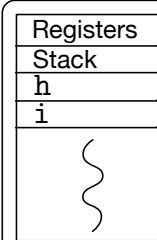


```
for (int i = 0; i < N/4; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```

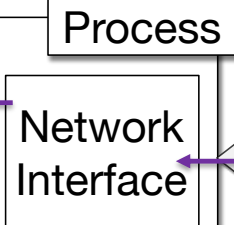


We are sending the *value* of “N”

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

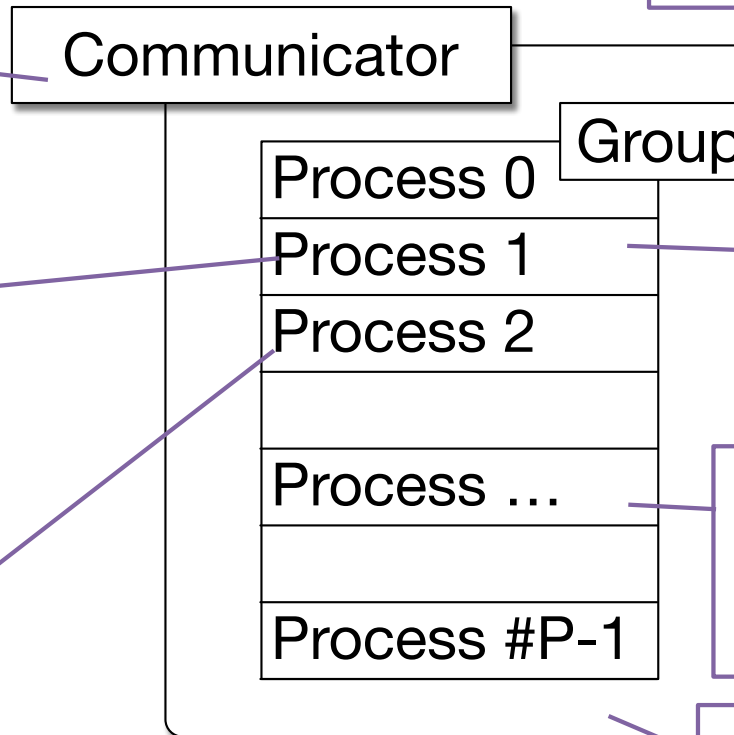
And its location only makes sense locally

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



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**

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

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

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

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

Communicator

An MPI Communicator contains an **MPI Group**

Group

Process 0

Process 1

Process 2

Process ...

Process #P-1

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

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

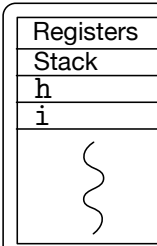
Message Contents

Contents

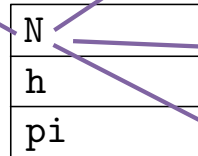
```
#include <mpi.h>
void Comm::Send(const void* buf, int count, const Datatype& datatype,
  ↪ int dest, int tag) const
```

In the program
this is a value

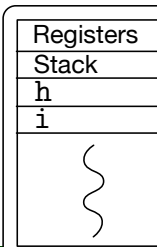
In the computer
this is just bits



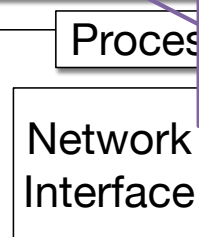
```
for (int i = 0; i < N/4; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```



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));
}
```



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

MPI API (section 3 man pages)

MPI	MPI_File_get_errhandler	MPI_texecscan	MPI_Status_f2c
MPI_Abort	MPI_File_get_group	MPI_tgather	MPI_Status_set_cancelled
MPI_Accumulate	MPI_File_get_info	MPI_tgatherv	MPI_Status_set_elements
MPI_Add_error_class	MPI_File_get_position	MPI_timprobe	MPI_Status_set_elements_x
MPI_Add_error_code	MPI_File_get_position_shared	MPI_timrecv	MPI_Test
MPI_Add_error_string	MPI_File_get_size	MPI_tneighbor_allgather	MPI_Test_cancelled
MPI_Address	MPI_File_get_type_extent	MPI_tneighbor_allgatherv	MPI_Testall
MPI_Allgather	MPI_File_get_view	MPI_tneighbor_alltoall	MPI_Testany
MPI_Allgatherv	MPI_File_read	MPI_tneighbor_alltoallv	MPI_Testsome
MPI_Alloc_mem	MPI_File_read_at	MPI_tneighbor_alltoallv	MPI_Topo_test
MPI_Allreduce	MPI_File_read_at_shared	MPI_Info_c2f	MPI_Type_c2f
MPI_Alltoall	MPI_File_read_shared	MPI_Info_create	MPI_Type_commit
MPI_Alltoallv	MPI_File_readv	MPI_Info_delete	MPI_Type_contiguous
MPI_Alltoallw	MPI_File_readv_shared	MPI_Info_dup	MPI_Type_create_darray
MPI_Attr_delete	MPI_File_open	MPI_Info_env	MPI_Type_create_f90_complex
MPI_Attr_get	MPI_File_preallocate	MPI_Info_f2c	MPI_Type_create_f90_integer
MPI_Attr_put	MPI_File_read	MPI_Info_free	MPI_Type_create_f90_real
MPI_Barrier	MPI_File_read_all	MPI_Info_get	MPI_Type_create_hindexed
MPI_Bcast	MPI_File_read_all_begin	MPI_Info_get_nkeys	MPI_Type_create_hindexed_block
MPI_Bsend	MPI_File_read_all_end	MPI_Info_get_nthkey	MPI_Type_create_hvector
MPI_Bsend_init	MPI_File_read_at	MPI_Info_get_valuelen	MPI_Type_create_indexed_block
MPI_Buffer_attach	MPI_File_read_at_all	MPI_Info_set	MPI_Type_create_keyval
MPI_Buffer_detach	MPI_File_read_at_all_begin	MPI_Init	MPI_Type_create_resized
MPI_Cancel	MPI_File_read_at_all_end	MPI_Init_thread	MPI_Type_create_struct
MPI_Cart_coords	MPI_File_read_ordered	MPI_Initialized	MPI_Type_create_subarray
MPI_Cart_create	MPI_File_read_ordered_begin	MPI_Intercomm_create	MPI_Type_delete_attr
MPI_Cart_get	MPI_File_read_ordered_end	MPI_Intercomm_merge	MPI_Type_dup
MPI_Cart_map	MPI_File_read_shared	MPI_Iprobe	MPI_Type_extent
MPI_Cart_rank	MPI_File_seek	MPI_Irecv	MPI_Type_f2c
MPI_Cart_shift	MPI_File_seek_shared	MPI_Ireduce	MPI_Type_free
MPI_Cart_sub	MPI_File_set_atomicsity	MPI_Ireduce_scatter	MPI_Type_free_keyval
MPI_Cartdim_get	MPI_File_set_errhandler	MPI_Ireduce_scatter_block	MPI_Type_get_attr
MPI_Close_port	MPI_File_set_info	MPI_Irsend	MPI_Type_get_contents
MPI_Comm_accept	MPI_File_set_size	MPI_Is_thread_main	MPI_Type_get_envelope
MPI_Comm_c2f	MPI_File_set_view	MPI_Iscan	MPI_Type_get_extent
MPI_Comm_call_errhandler	MPI_File_sync	MPI_Iscatter	MPI_Type_get_extent_x
MPI_Comm_compare	MPI_File_write	MPI_Iscatterv	MPI_Type_get_name
MPI_Comm_connect	MPI_File_write_all	MPI_Issend	MPI_Type_get_true_extent
MPI_Comm_create	MPI_File_write_all_begin	MPI_Issend	MPI_Type_get_true_extent_x
MPI_Comm_create_errhandler	MPI_File_write_all_end	MPI_Keyval_create	MPI_Type_hindexed
MPI_Comm_create_group	MPI_File_write_at	MPI_Keyval_free	MPI_Type_hvector
MPI_Comm_create_keyval	MPI_File_write_at_all	MPI_Lookup_name	MPI_Type_indexed
MPI_Comm_delete_attr	MPI_File_write_at_all_begin	MPI_Mprobe	MPI_Type_lb
MPI_Comm_disconnect	MPI_File_write_at_all_end	MPI_Mrecv	MPI_Type_match_size
MPI_Comm_dup	MPI_File_write_ordered	MPI_Neighbor_allgather	MPI_Type_set_attr
MPI_Comm_dup_with_info	MPI_File_write_ordered_begin	MPI_Neighbor_allgatherv	MPI_Type_set_name
MPI_Comm_f2c	MPI_File_write_ordered_end	MPI_Neighbor_alltoall	MPI_Type_size
MPI_Comm_free	MPI_File_write_shared	MPI_Neighbor_alltoallv	MPI_Type_size_x
MPI_Comm_free_keyval	MPI_Finalize	MPI_Neighbor_alltoallv	MPI_Type_struct
MPI_Comm_get_attr	MPI_Finalized	MPI_Op_c2f	MPI_Type_ub
MPI_Comm_get_errhandler	MPI_Free_mem	MPI_Op_create	MPI_Type_vector
MPI_Comm_get_info	MPI_Gather	MPI_Op_f2c	MPI_Unpack
MPI_Comm_get_name	MPI_Gather	MPI_Op_free	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>

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

```
#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 to the community (researchers, vendors, et al) to provide implementations that conform to the standard specification
- High-quality implementations have useful extensions

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

Only process 0
sends this

Only process 0
receives this

Only process 1
receives this

Only process 1
sends this

```
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)

```
#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

Ping

Only process 0
sends this

Only process 0
receives this

Only process 1
receives this

Only process 1
sends this

```
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;  
}
```

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 Fun

Broadcast values
to all nodes

All nodes to
exactly this

Collect results
from all nodes

```
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()
```


MPI Bcast

Have to 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 have
to call this

But no sender or
receiver per se

MPI Reduce

Have to 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

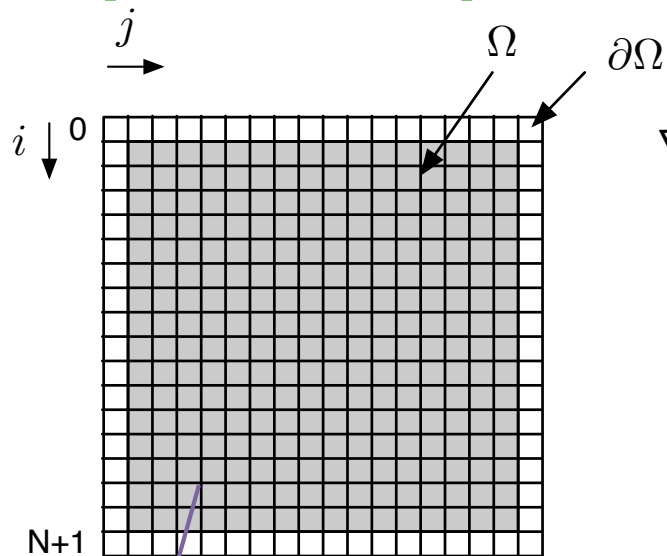
Once all nodes have called it

All nodes have to 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



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

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

Discretization

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

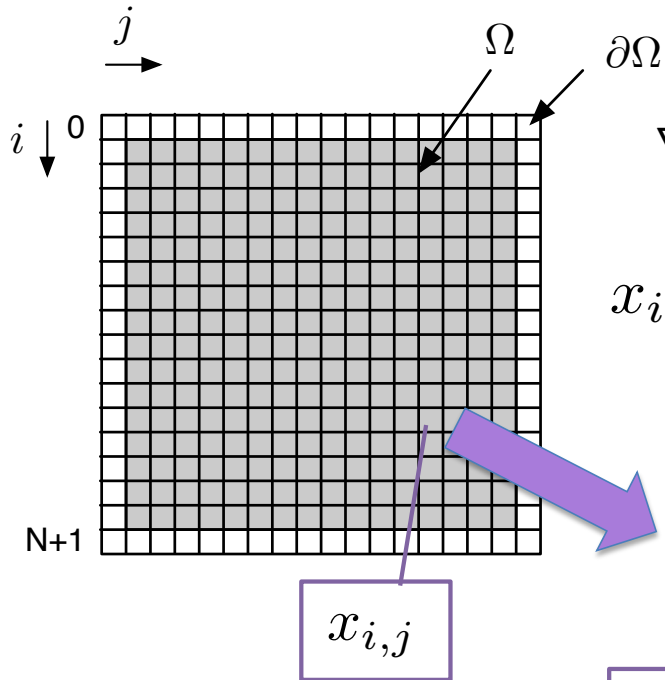
$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4$$

$x_{i,j}$

The value of each point on the grid

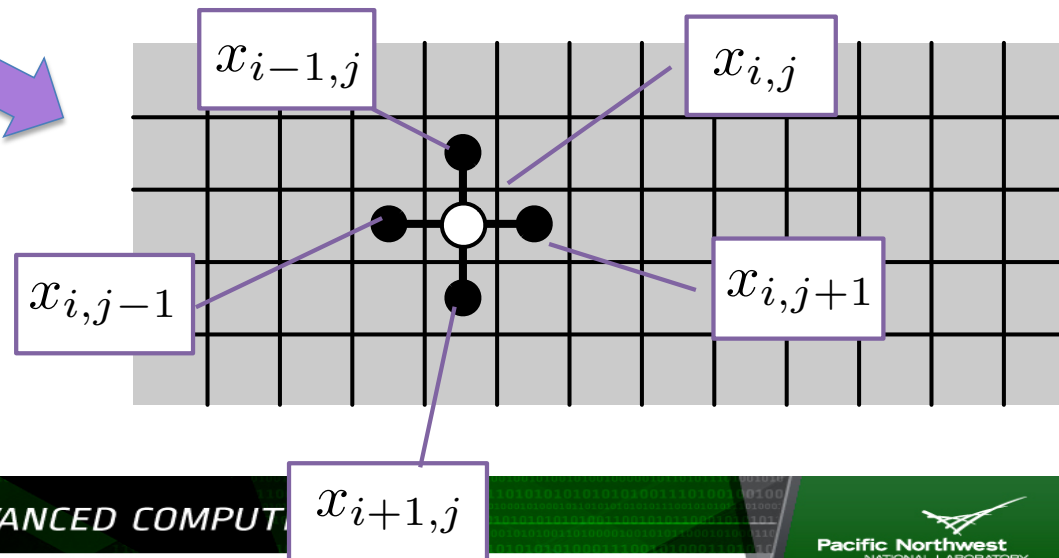
The average of its neighbors

Laplace's Equation on a Regular Grid

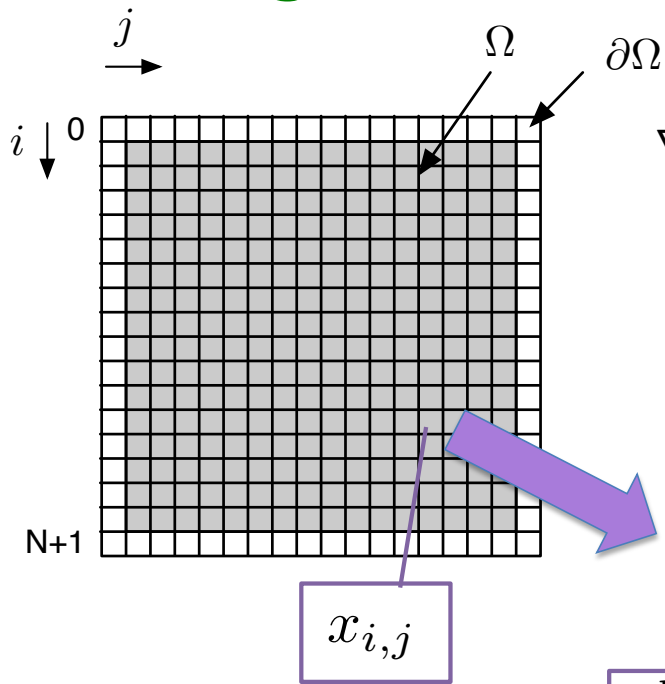


$$\begin{aligned}\nabla^2\phi &= 0 \quad \text{on } \Omega \\ \phi &= f \quad \text{on } \partial\Omega\end{aligned}$$

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

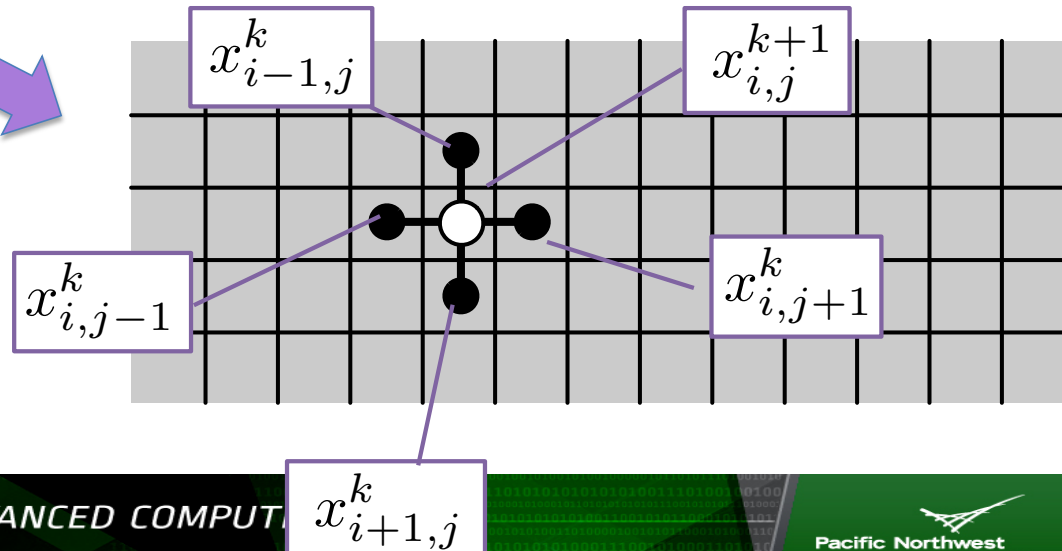


Iterating for a solution



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

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



Approximation at iteration $k+1$

Average of approximation at iteration k

Iterating for a solution

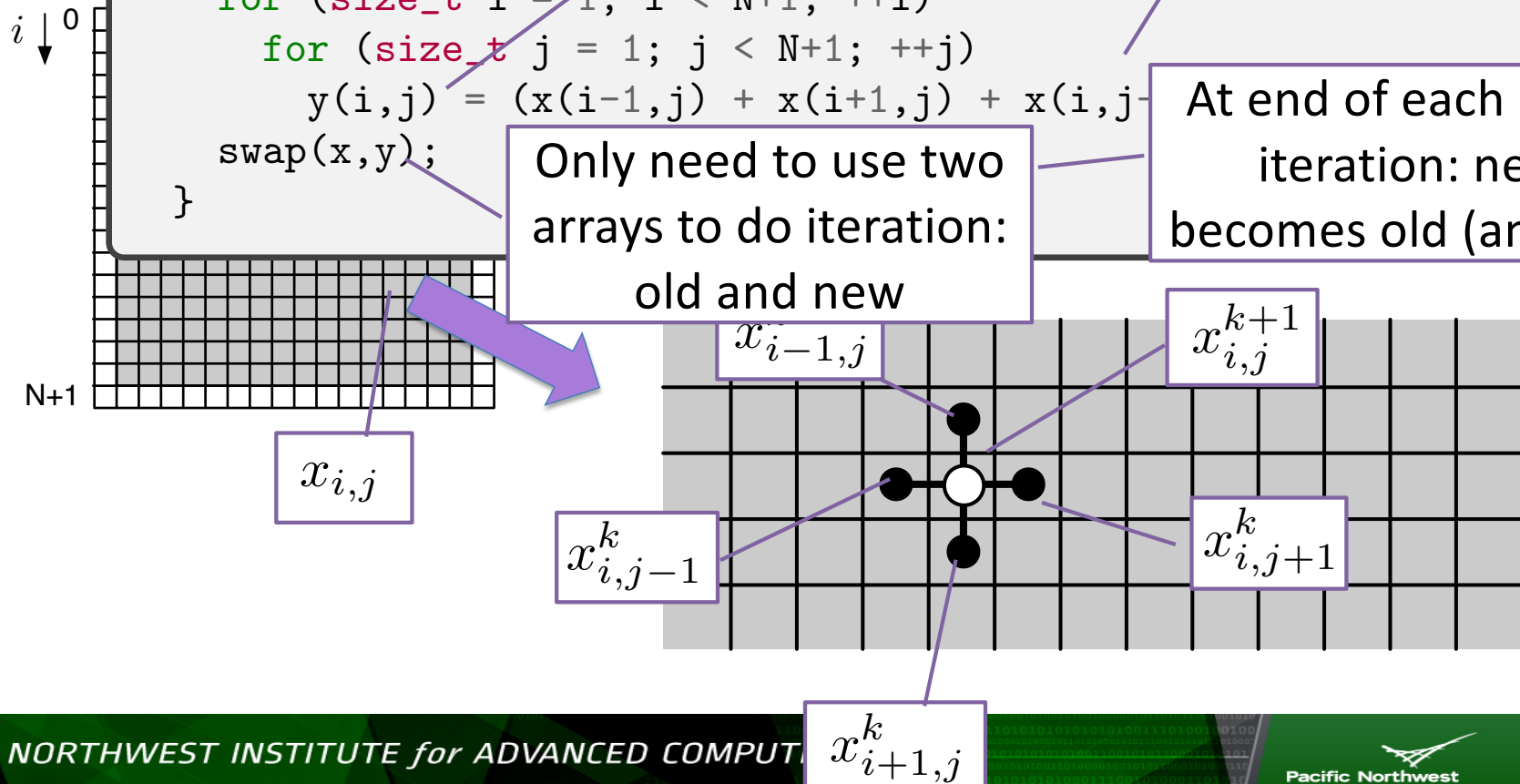
```
while (! converged())  
  for (size_t i = 1; i < N+1; ++i)  
    for (size_t j = 1; j < N+1; ++j)  
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1)) / 4;  
      swap(x,y);  
}
```

Approximation at iteration k+1

Average of approximation at iteration k

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

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



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

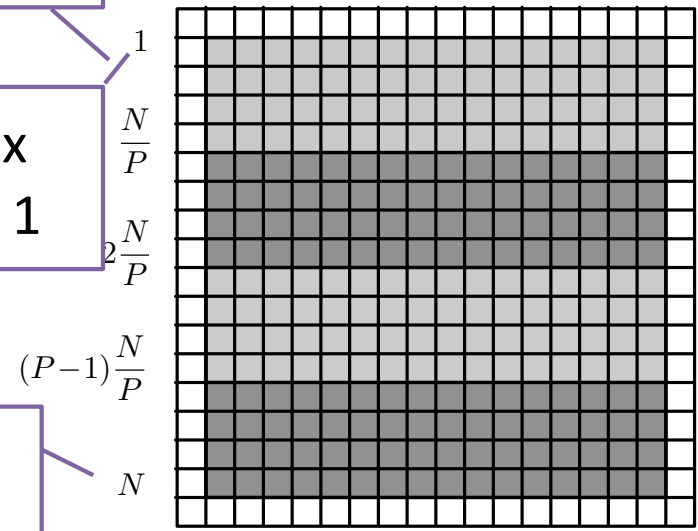
```
int jacobi(Grid& X0, Grid& X1, size_t max_iters, double tol) {
    for (size_t iter = 0; iter < max_iters; ++iter) {
        double rnorm = jacobiStep(X0, X1);
        if (rnorm < tol) return 0;
        swap(X0, X1);
    }
    return -1;
}
```

Original problem

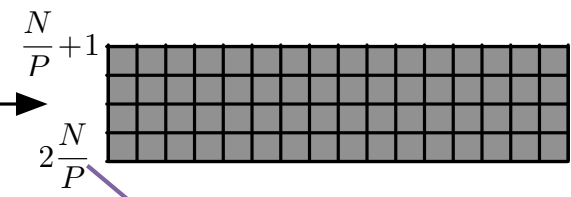
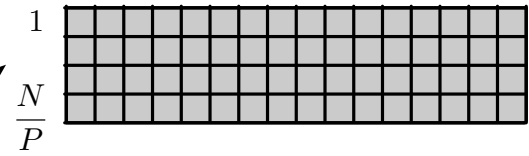
Index from 1

Index to N

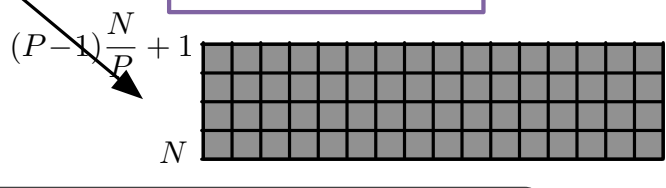
Decompose into P partitions



Global



Partitioned index space



Local

SPMD index space

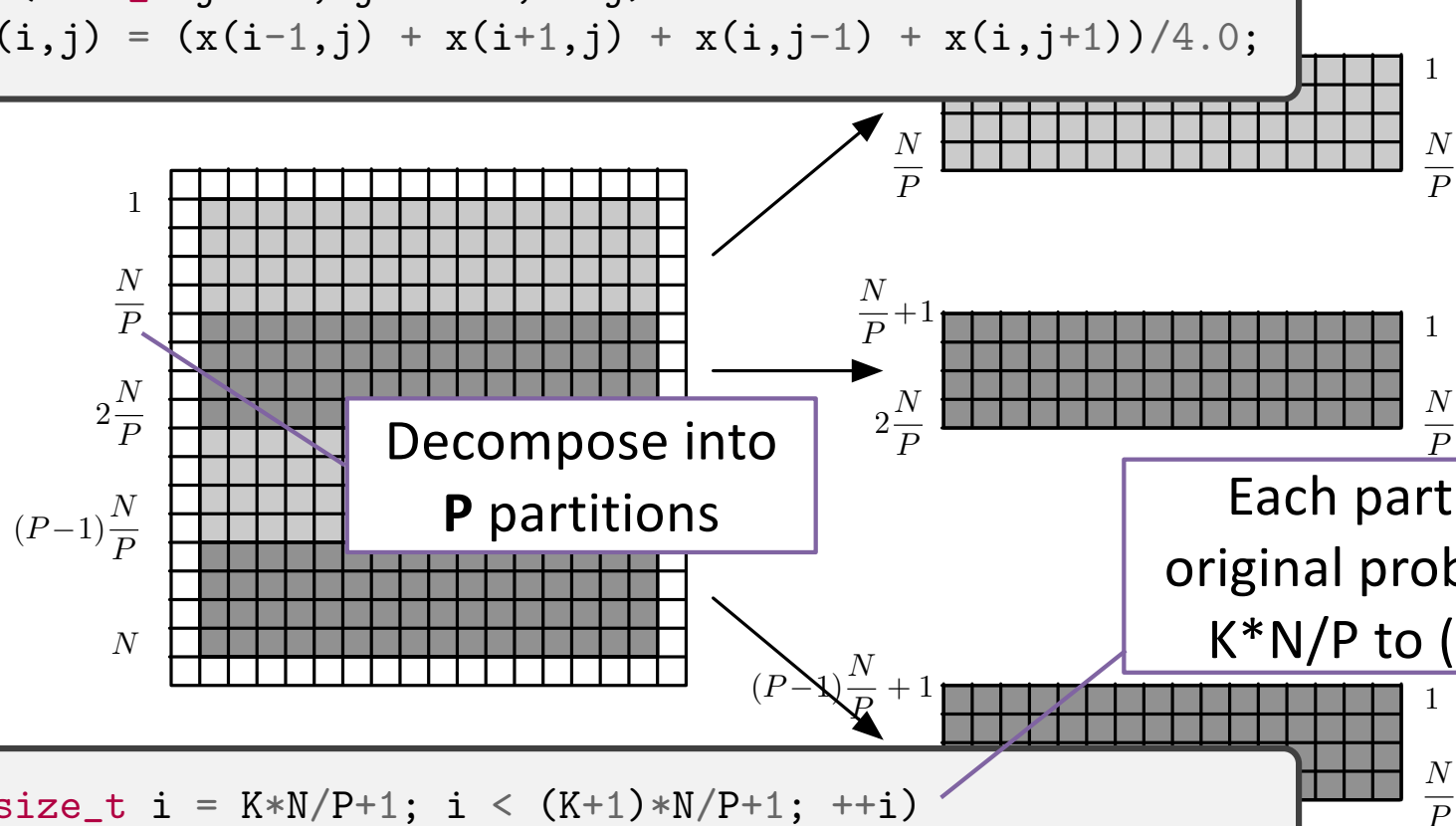
All are identical

```
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;

```



Decompose into
P partitions

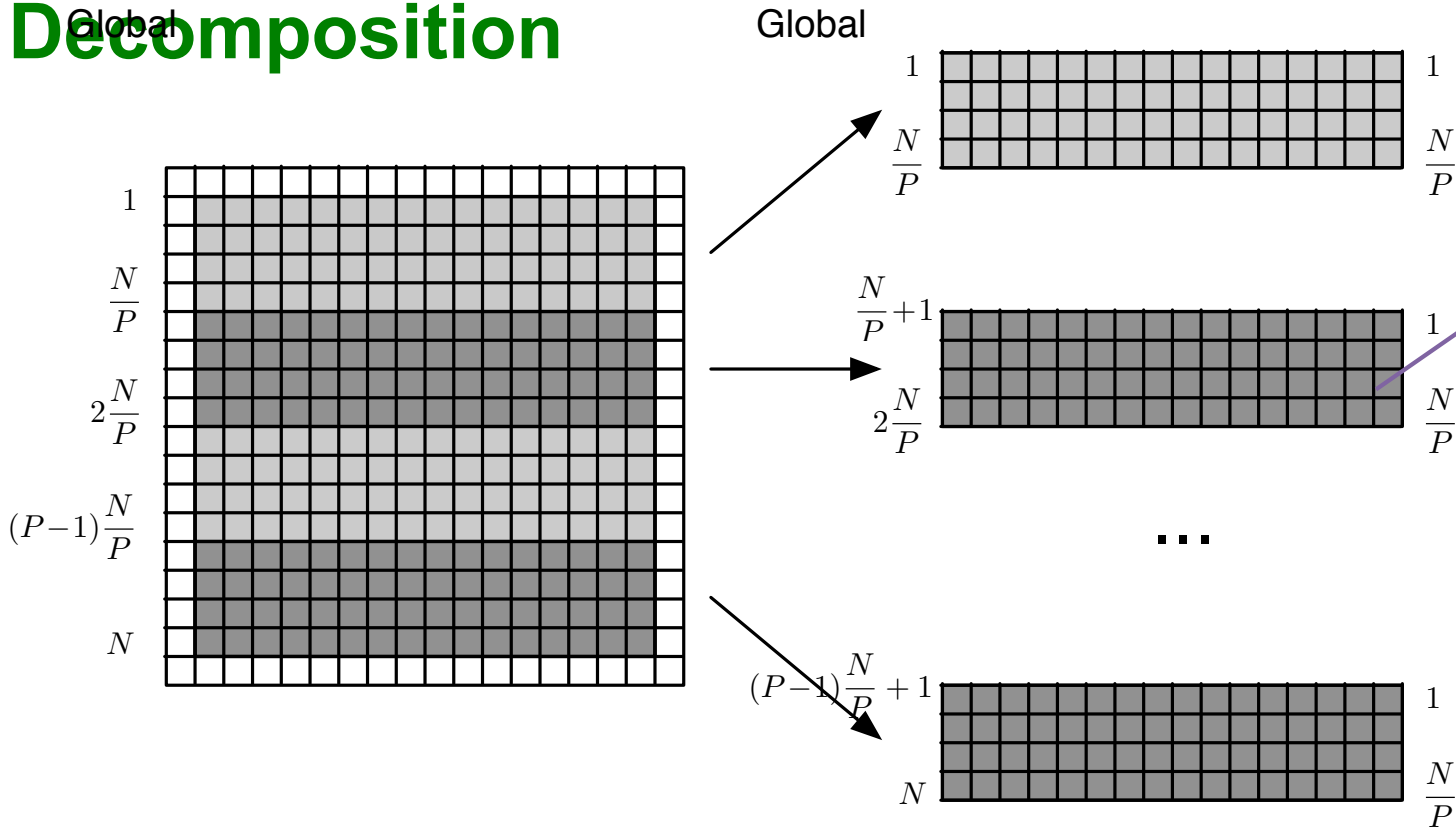
Each partition **K** in
original problem covers
 $K*N/P$ to $(K+1)*N/P$

```

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



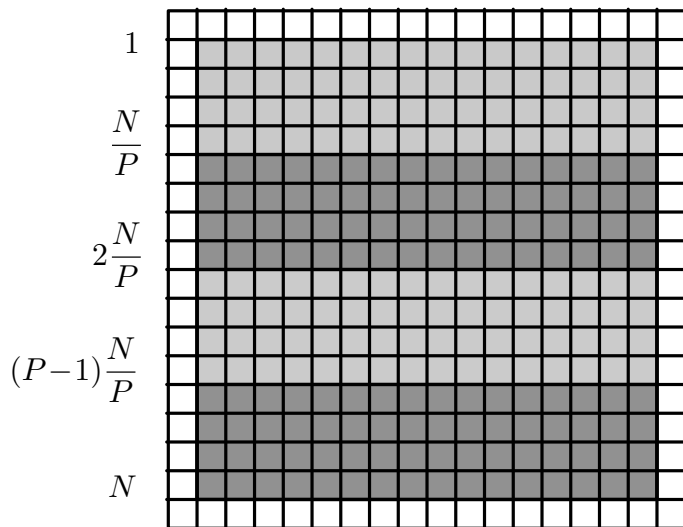
We aren't storing entire array on each node

We are storing only N/P rows

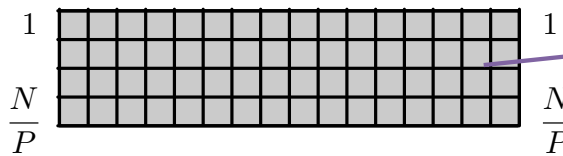
We need to iterate 1 to N

```
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;
```

Global Decomposition

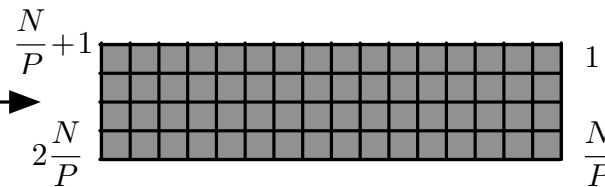


Global

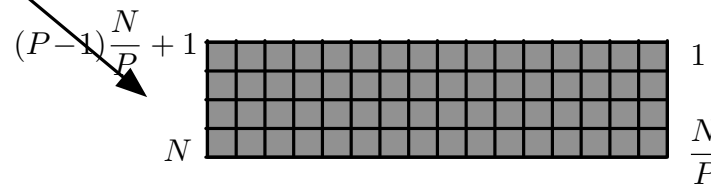


Local

Any obvious problems?



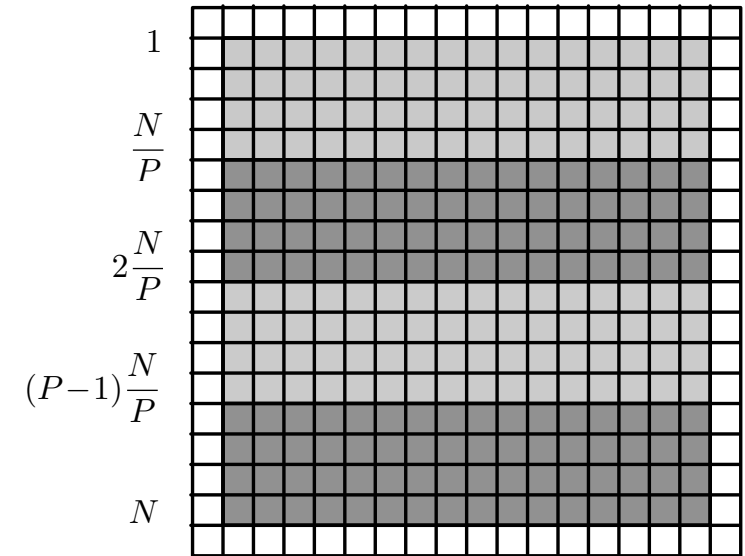
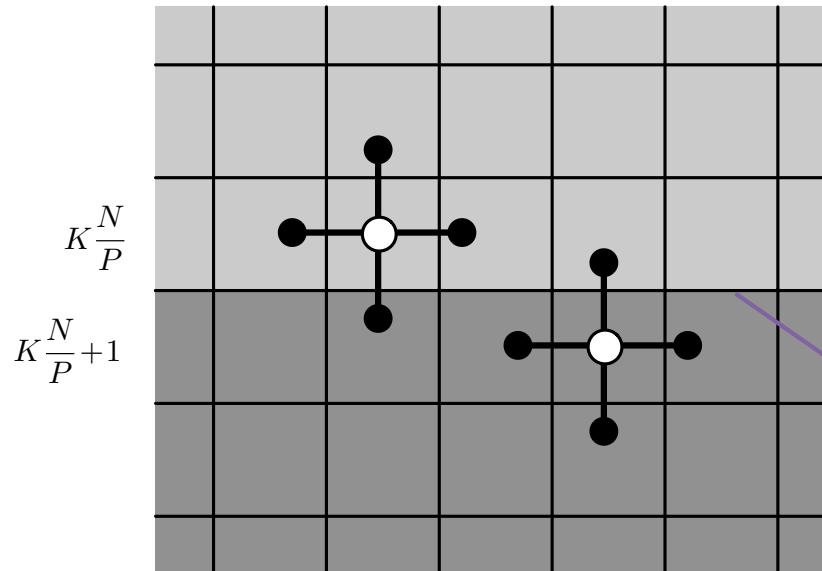
...



We need to iterate 1 to N

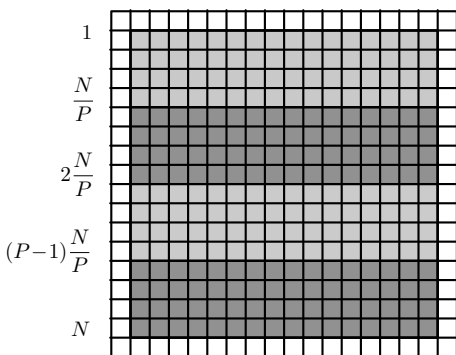
```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition

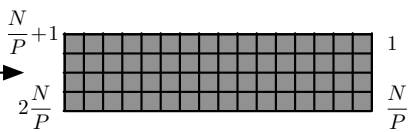


Data dependencies for stencil crosses partition boundary in original problem

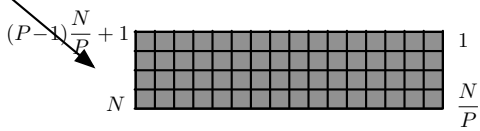
Decomposition



This is not a valid read



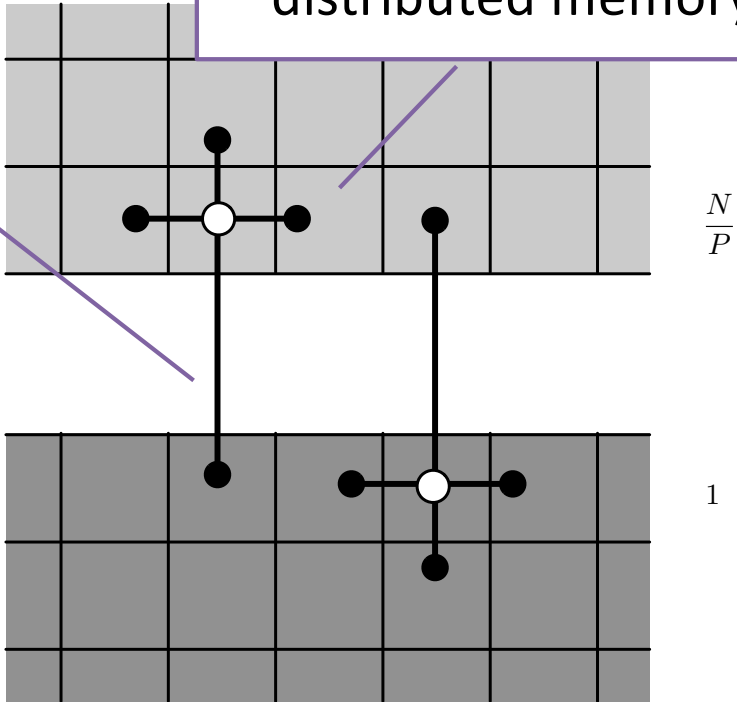
...



Index shifting doesn't help

This is just a program

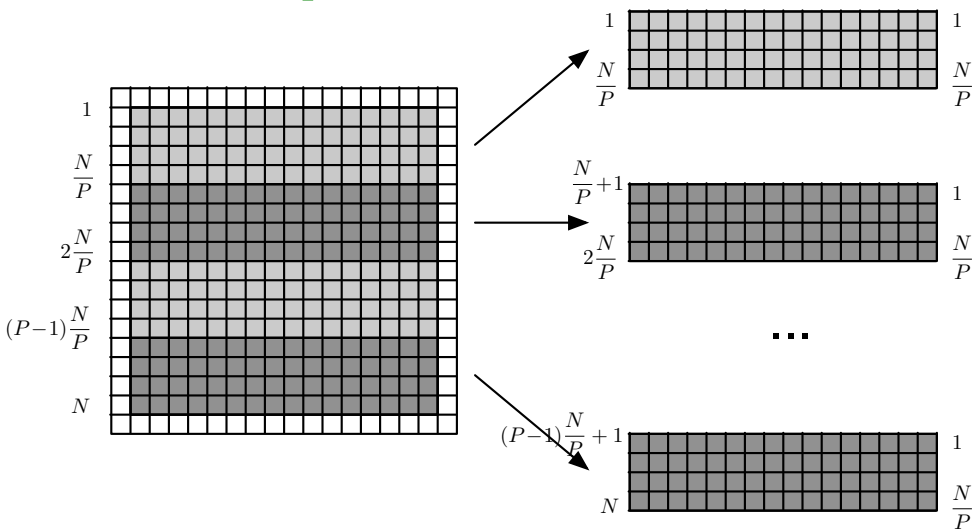
Which is a problem in distributed memory



```
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
```

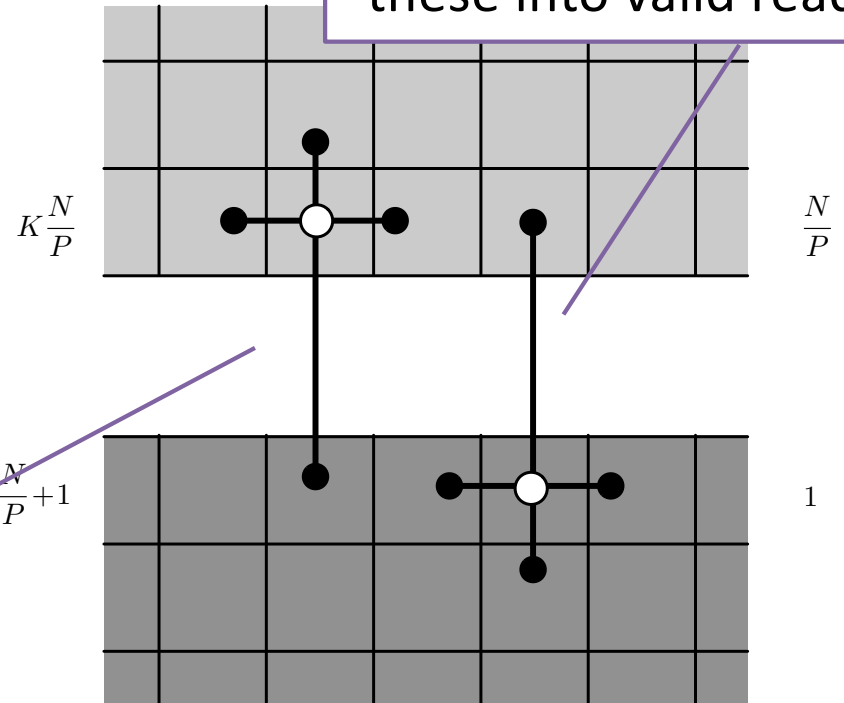
It reads/writes local data, just like any other

Decomposition



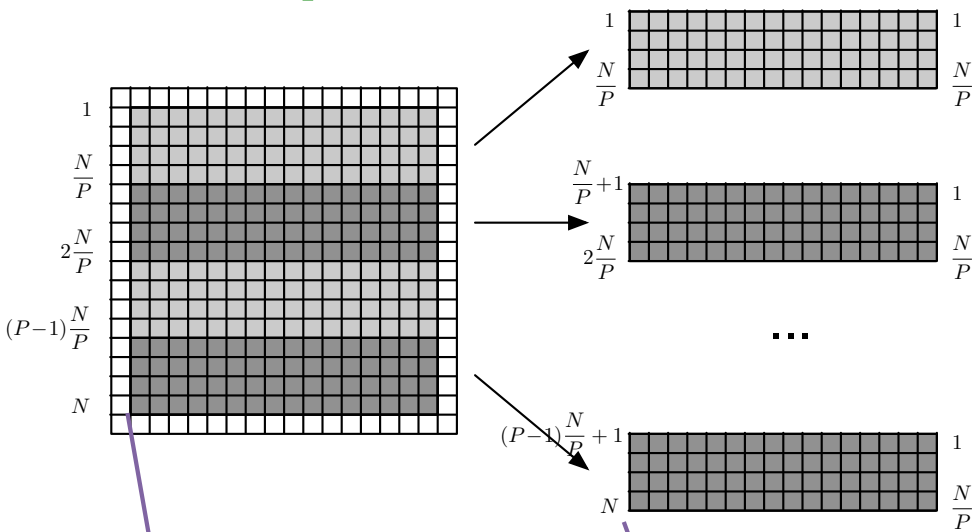
And preserve the
"as-if" property

We need to make
these into valid reads



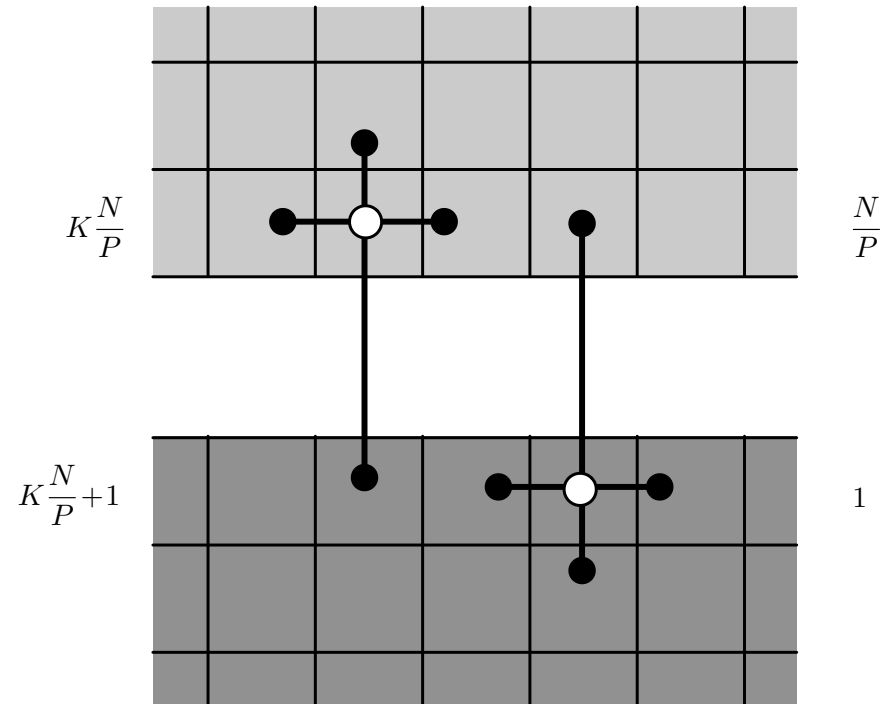
```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition



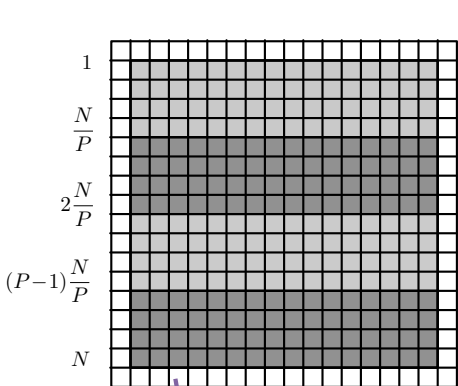
Where did the boundary go

When we partitioned?

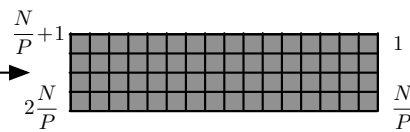


```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

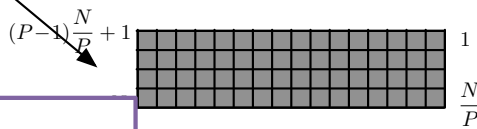
Decomposition



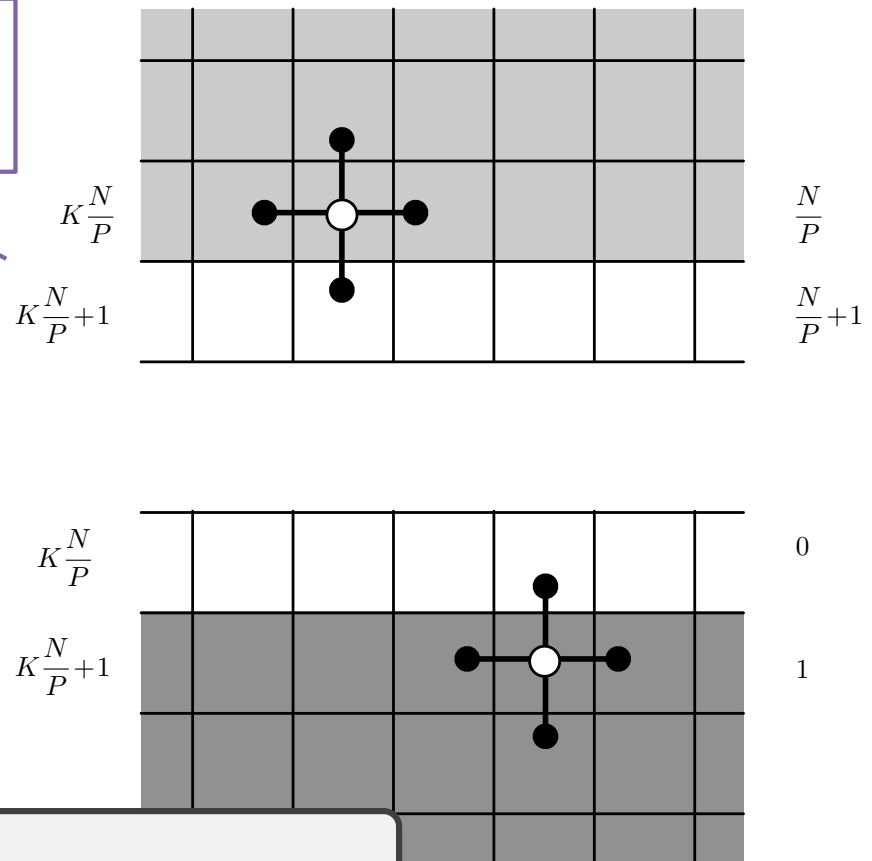
We need an extra row for "as-if"



...

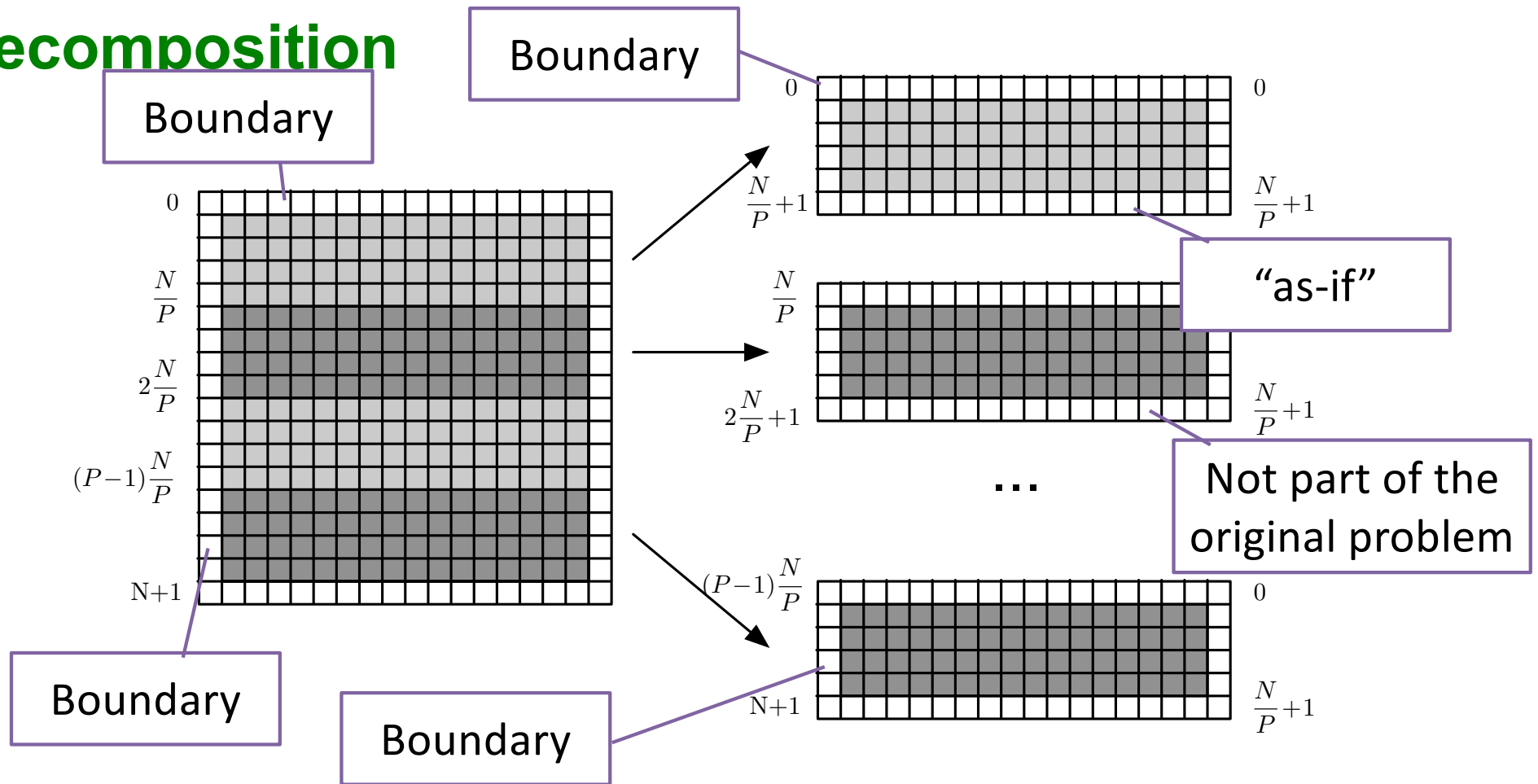


And an extra row (on some nodes) for the boundary



```
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”

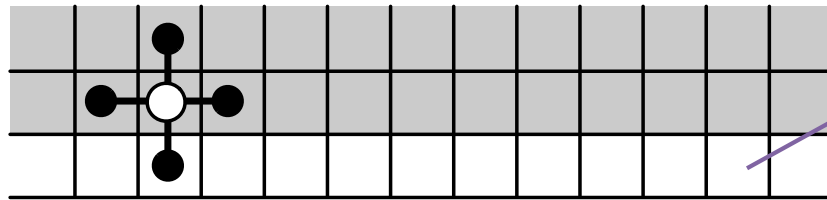
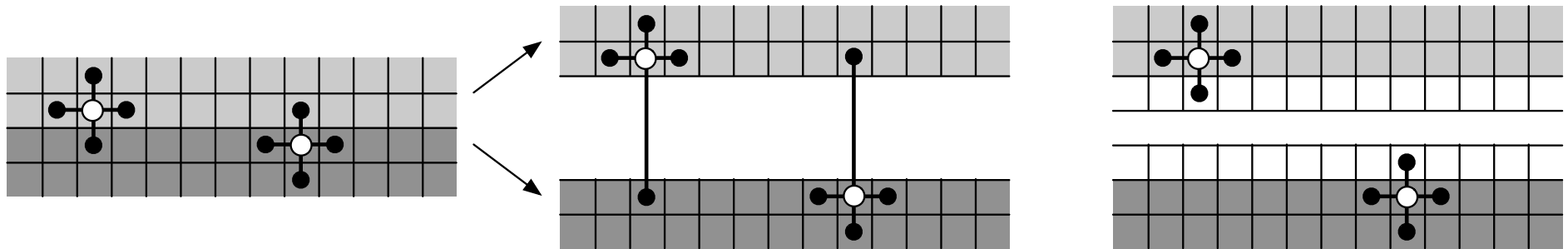
To the local / SPMD code, the boundary and as-if are the same

Not part of the original problem

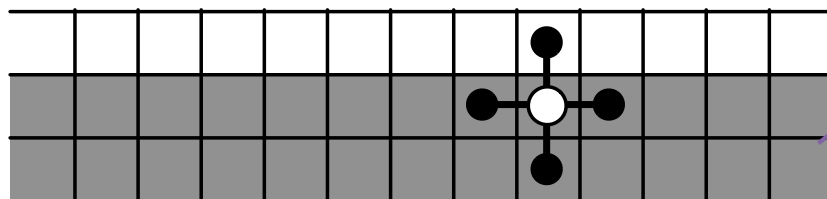
Is the same as solving lots of the same problem but smaller

```
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

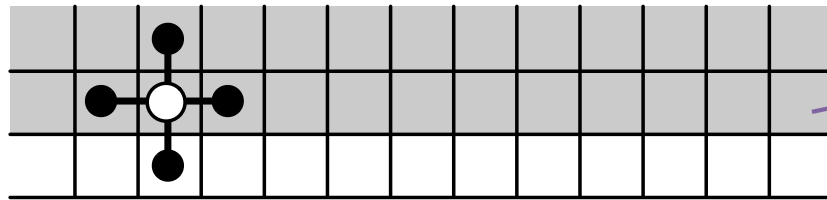
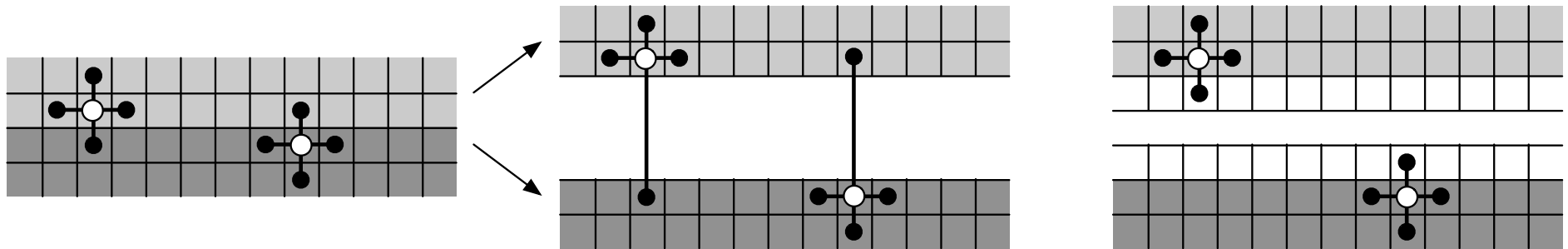


This row needs to be "as-if"

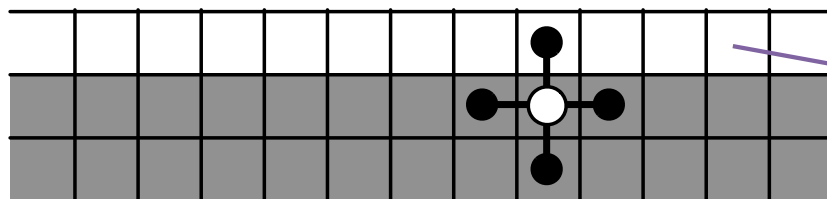


it were this row

As-If



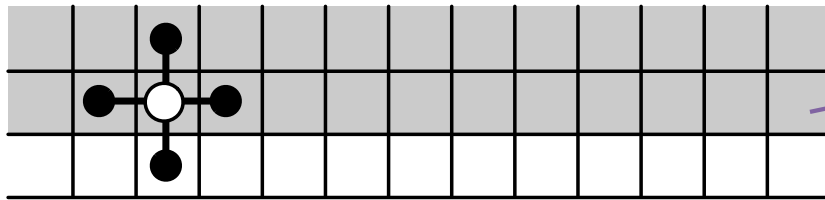
it were this row



And this row needs to be "as-if"

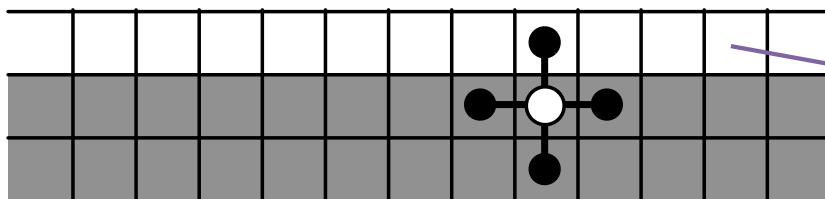
As-If

```
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

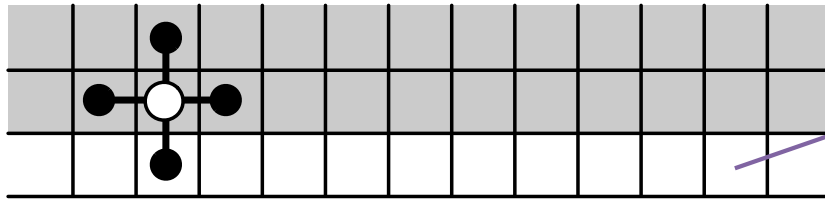


And this row needs
to be "as-if"

As-If

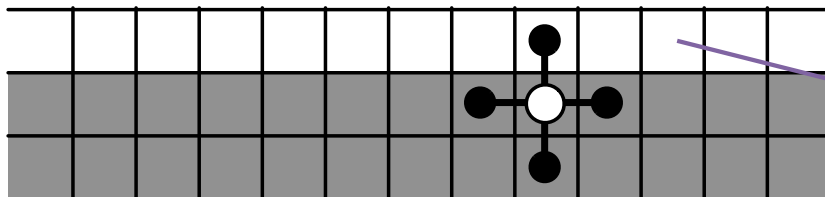
```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

This is the computation



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?

Always write y

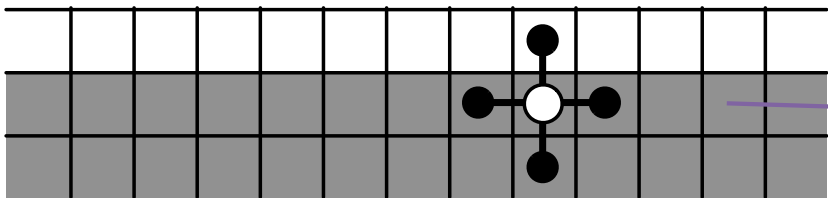
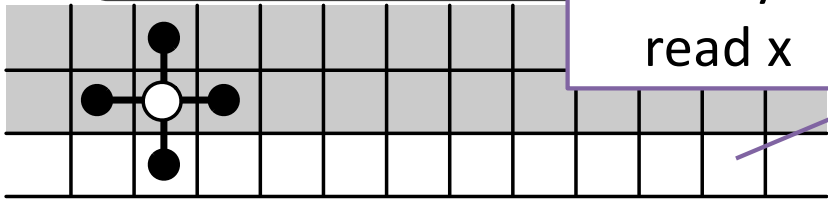
```
(! converged()) {  
  for (size_t i = 1; i < N+1; ++i)  
    for (size_t j = 1; j < N+1; ++j)  
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;  
  swap(x,y);  
}
```

This is the entire program

Always read x

Not changed during an iteration

Rows need to be as-if only during iteration



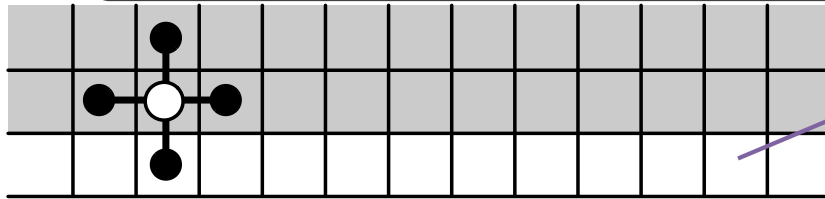
This changes only on every outer iteration (on the swap())

As-If

```
while (! converged)
  for (size_t i = 0; i < N; ++i)
    for (size_t j = 0; j < N; ++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);
}
```

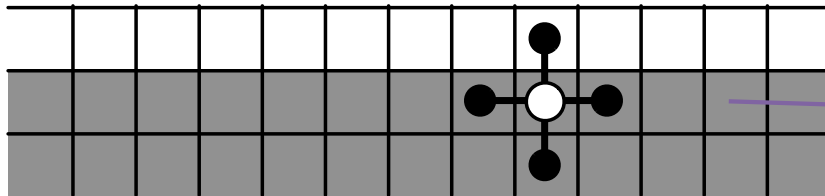
Here is where we need to make as-if true

This is the entire program



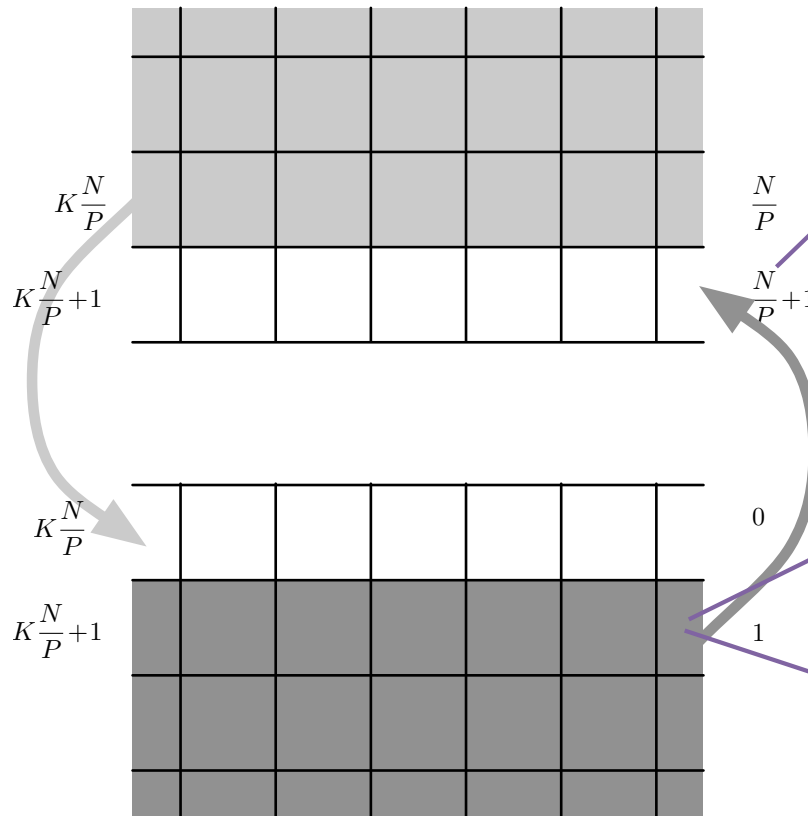
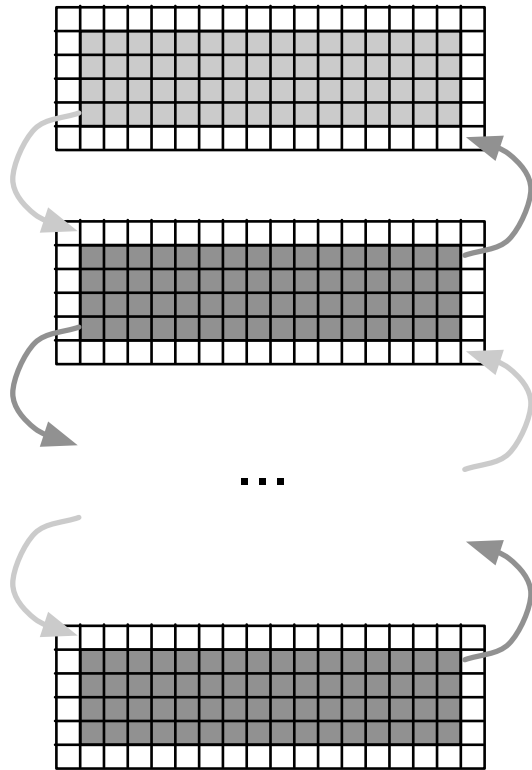
Not changed during an iteration

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

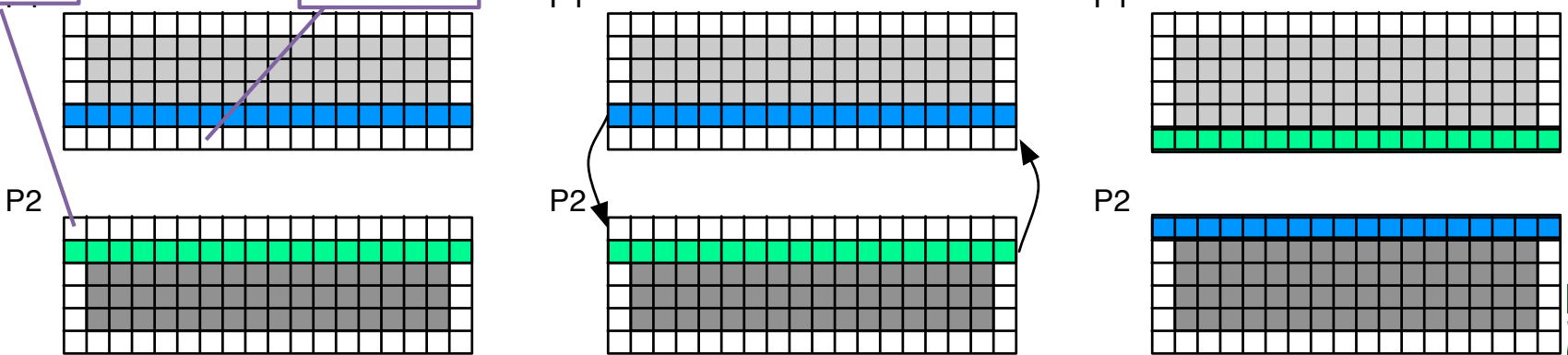
Before they are read at the next outer iteration

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



NORTH

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  
}
```

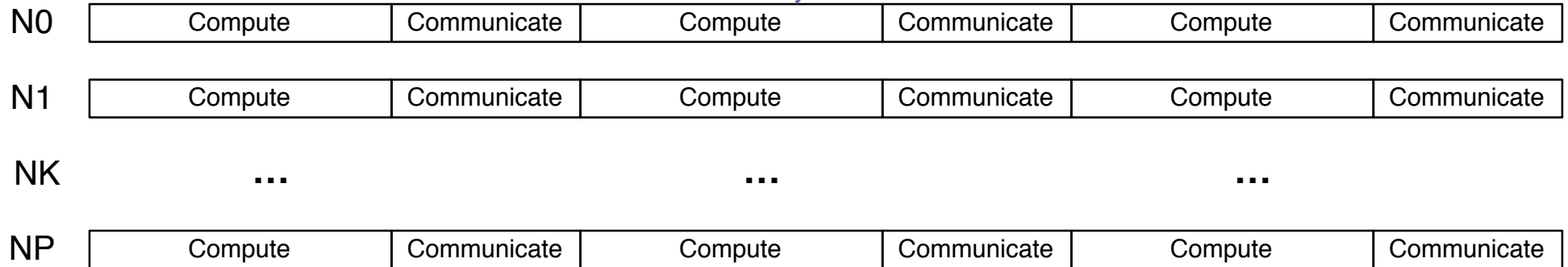
Compute

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

Time

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

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

Just like
reduce

So there are
two buffers

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)
```

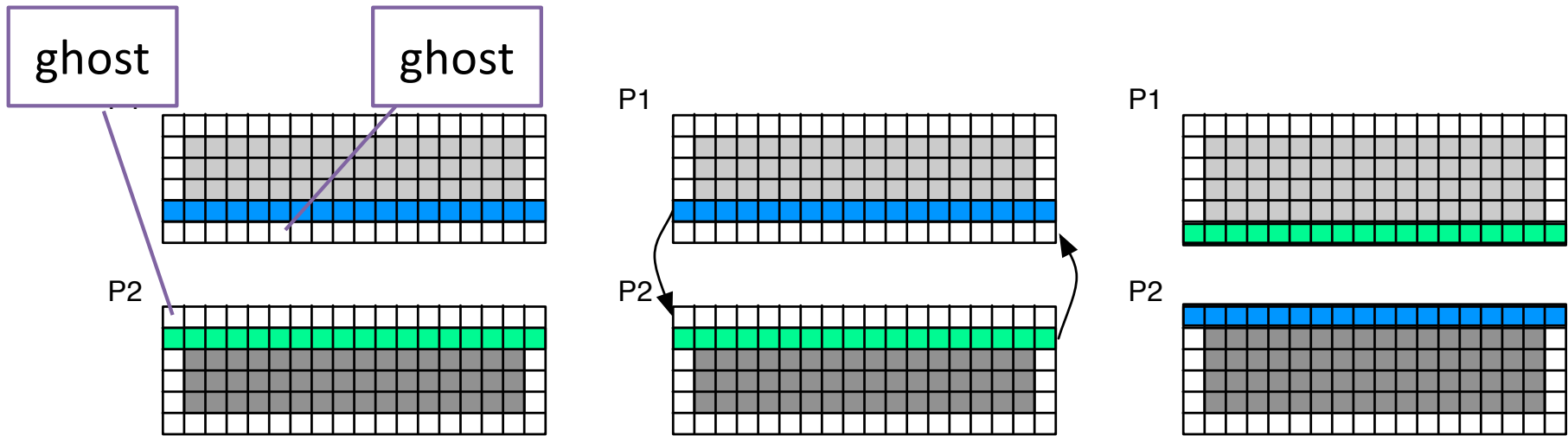
Parallel Jacobi Solver

```
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



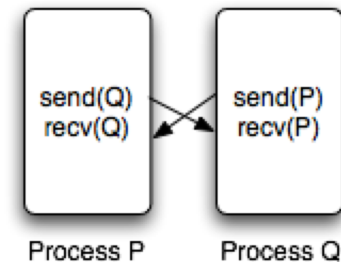
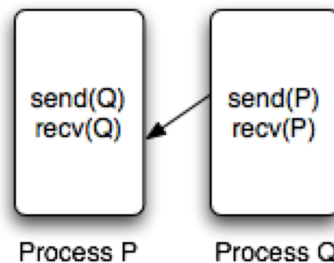
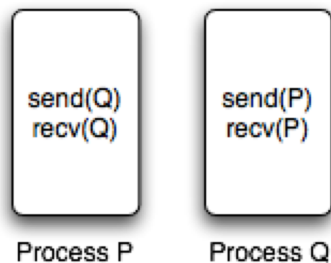
```

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

Works?

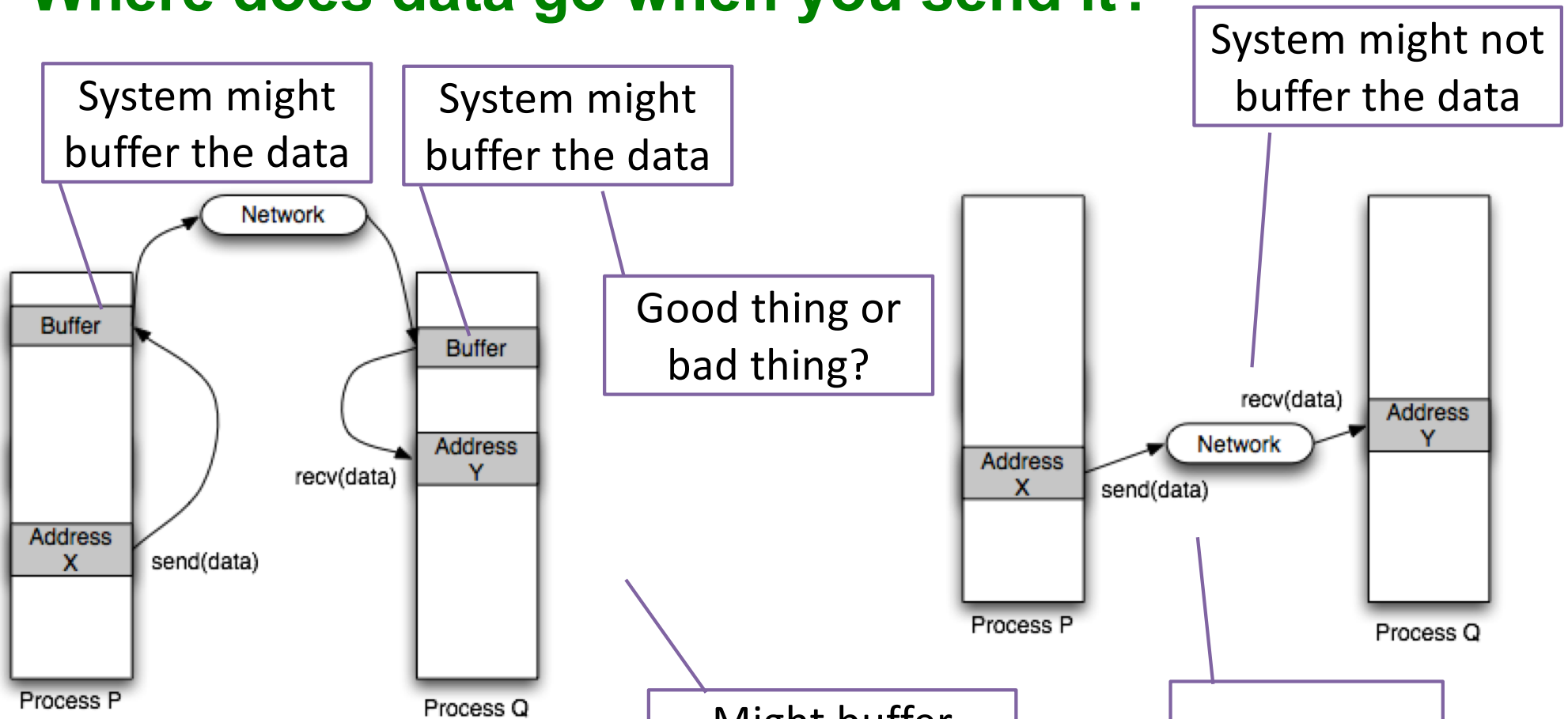
Updating ghost cells

- 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

Next

- Performance models (LogP and BSP)
- Summary of collectives, datatypes, non-blocking operations
- Finish up CSP Jacobi iteration
- Briefly discuss amath583 cluster login, questions for docker portion

- Stay tuned

Thank You!

NORTHWEST INSTITUTE for ADVANCED COMPUTING

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine


Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle
for the U.S. Department of Energy


UNIVERSITY of
WASHINGTON

Creative Commons BY-NC-SA 4.0 License



© Andrew Lumsdaine, 2017-2018

Except where otherwise noted, this work is licensed under

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

