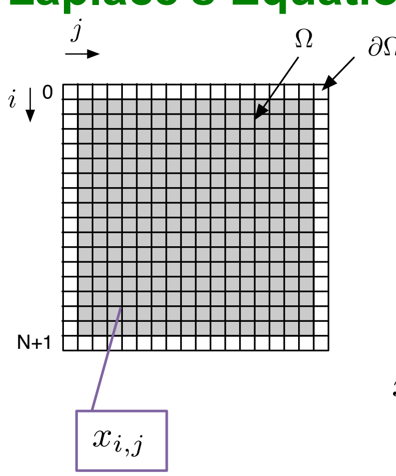
AMATH 483/583
High Performance Scientific Computing
Lecture 19:
Advanced Message Passing, Collectives, Performance Models

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

Laplace's Equation on a Regular Grid



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

Discretization

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

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

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

The value of each point on the grid

The average of its neighbors

Iterating for a soluti

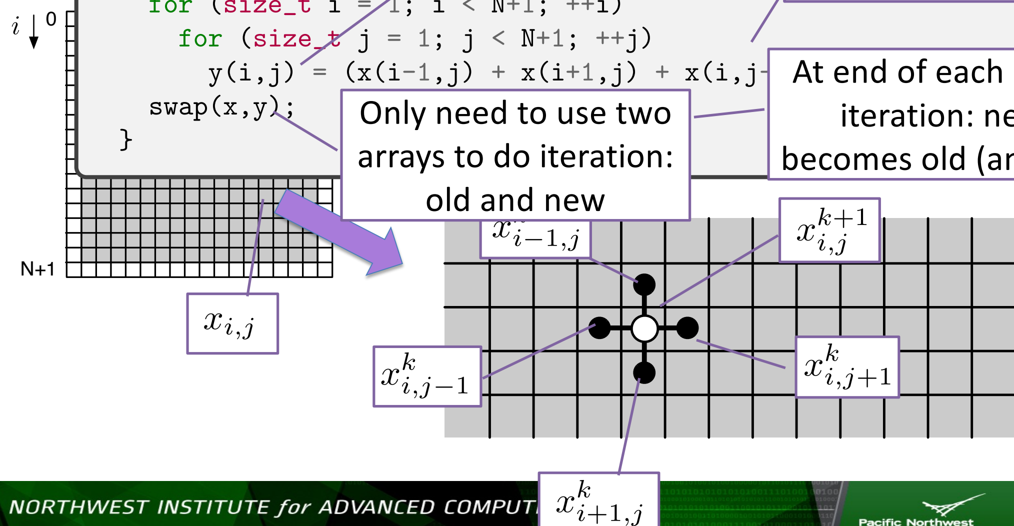
```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-
  swap(x,y);
}
```

Approximation at iteration k+1

Average of approximation at iteration k

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

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



class Grid

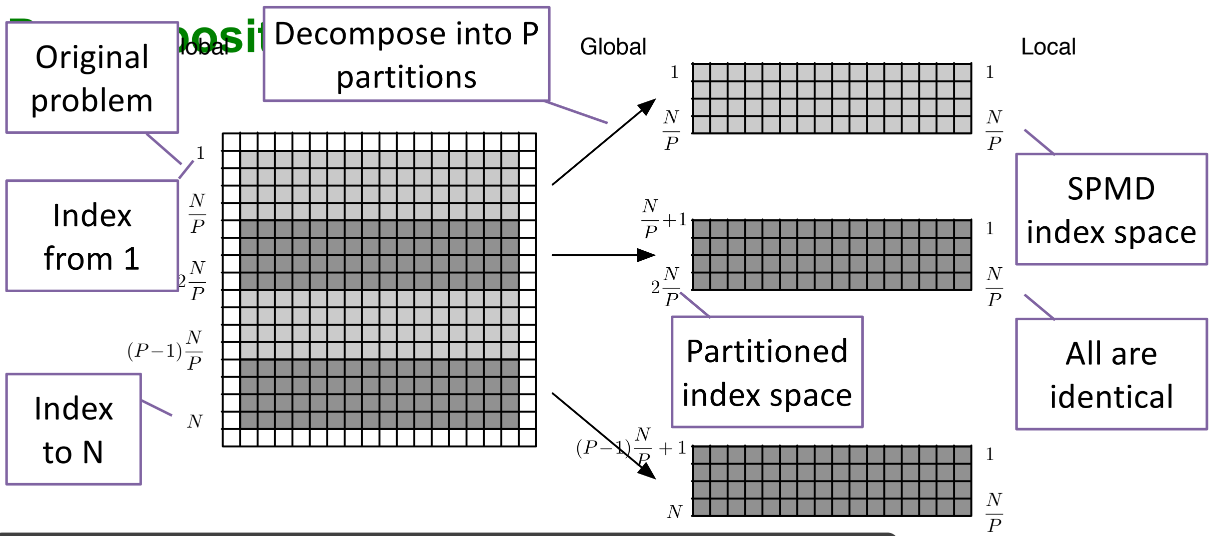
```
class Grid {

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

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

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

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

Grid is a 2D array

Constructor

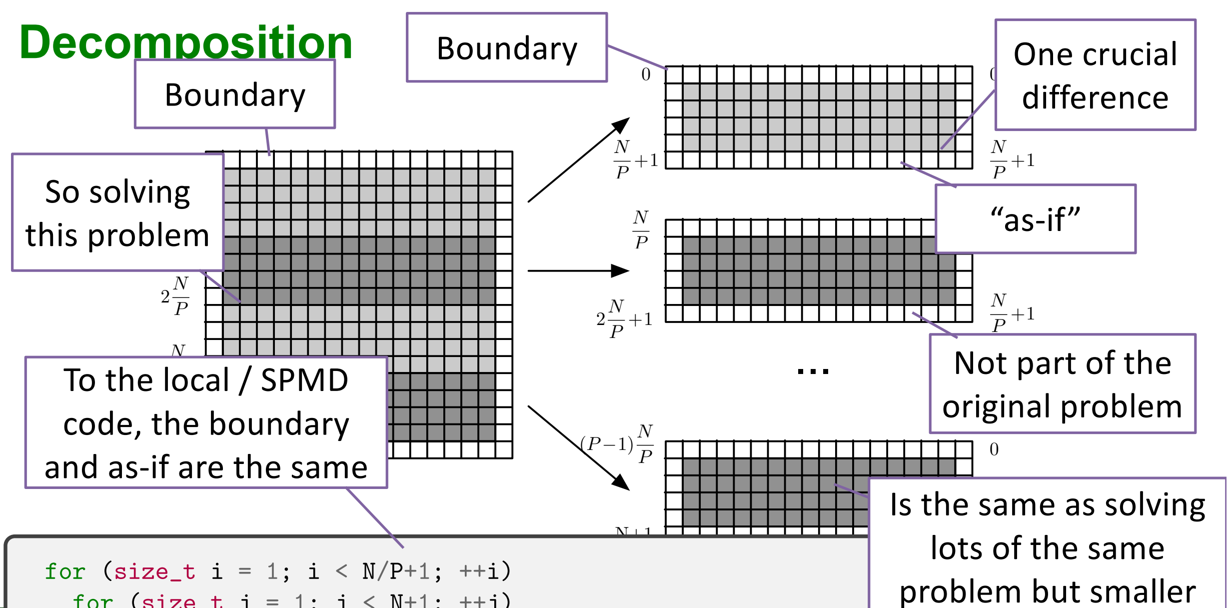
Accessor

Storage




```
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;
```


Decomposition




```
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;
```

```
  (! 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);
}
```

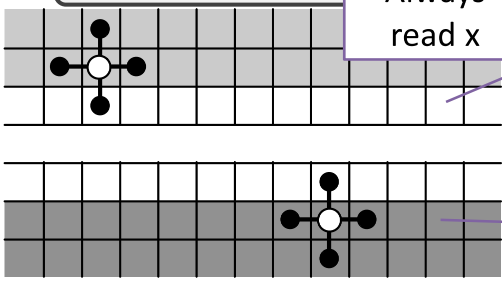
Always write y

This is the entire program

Always read x

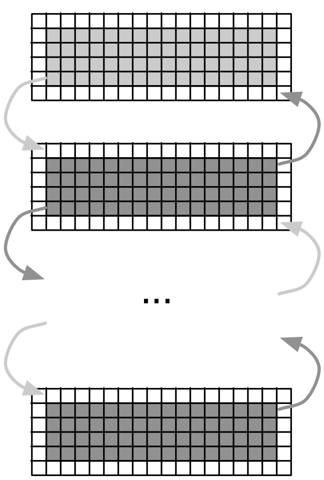
Not changed during an iteration

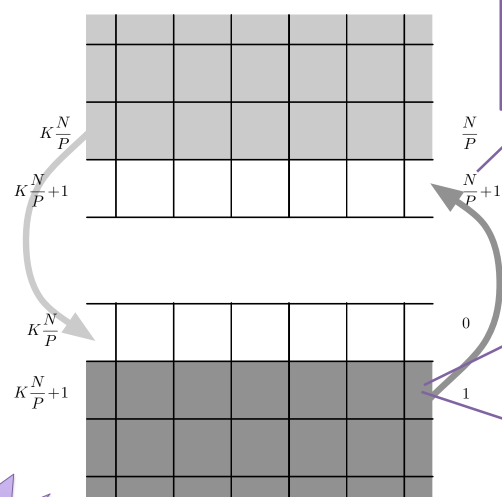
Rows need to be as-if only during iteration



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

Compute / Communicate





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

With their "as-if" values

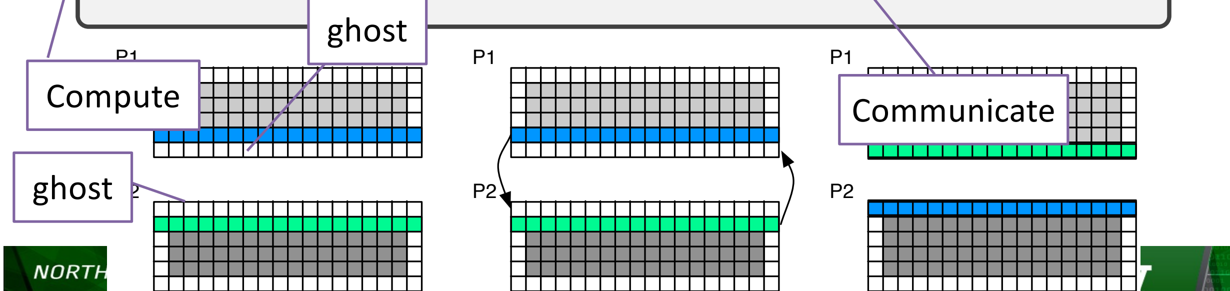
Before they are read at the next outer iteration

Very Important Slide!!

Compute / Communicate

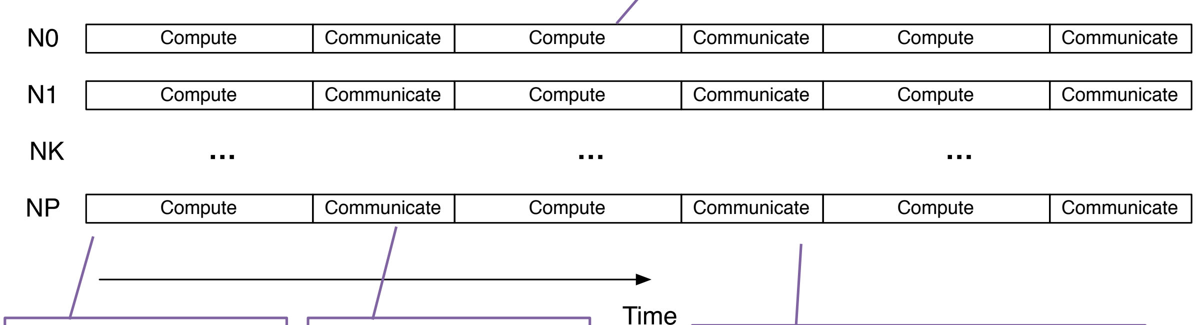
```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
  swap(x,y);
  make_as_if(x); // Communicate ghost cells
}
```

Standard terminology for as-if boundary is “ghost cell” or “halo”



Compute / Communicate

“Bulk Synchronous Parallel” (BSP)

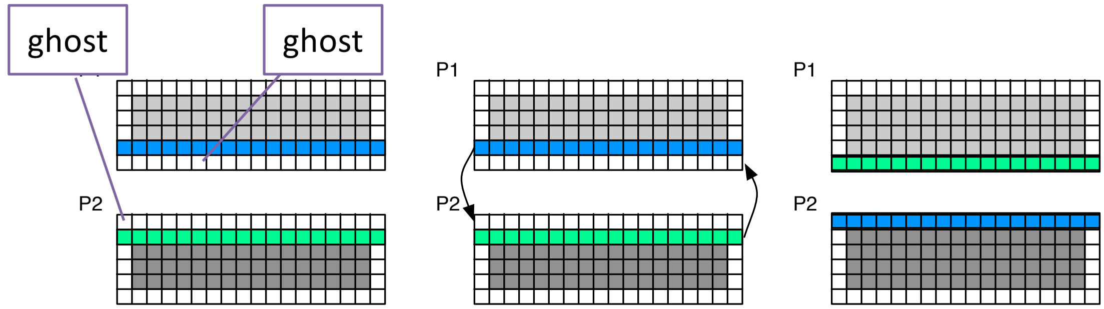


This is an almost universal pattern

Processors are still only loosely coupled

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

Updating Ghost Cells

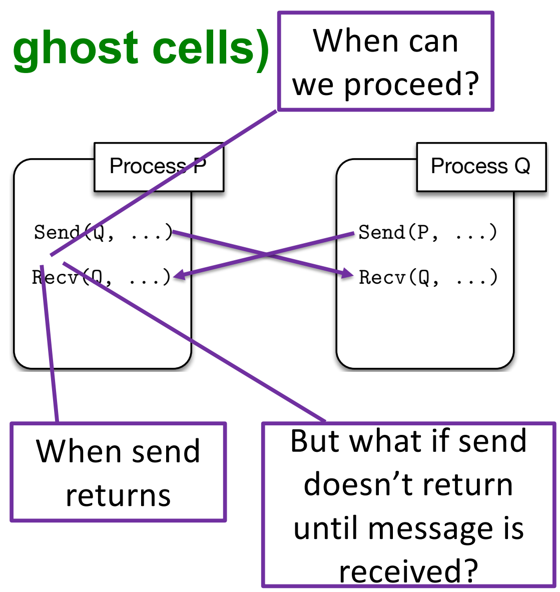



```
MPI_Send( ... );   // to upper neighbor
MPI_Send( ... );   // to lower neighbor
MPI_Recv( ... );   // from lower neighbor
MPI_Recv( ... );   // from upper neighbor
```

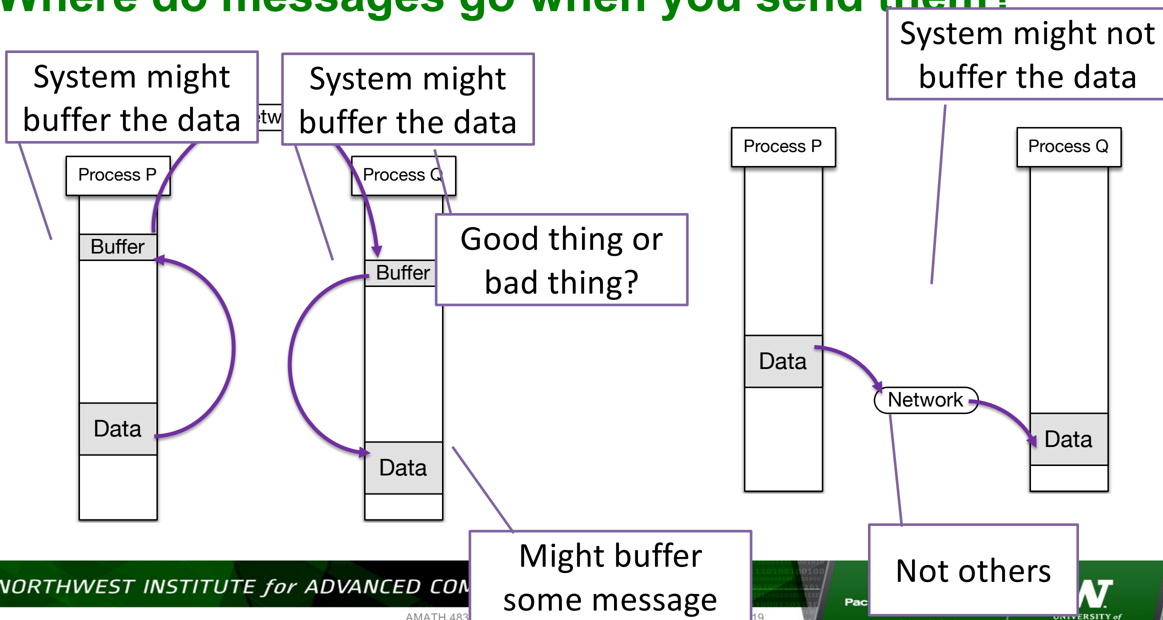
Works?

Exchanging halos (updating ghost cells)

- What happens with this set of operations?
- Have we seen this before?
- Behavior depends on implementation of Send (not its semantics)
  - Size of message (use of eager vs rendezvous protocol)
  - System dependent
  - Most MPI implementations have diagnostics for this



Where do messages go when you send them?



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)

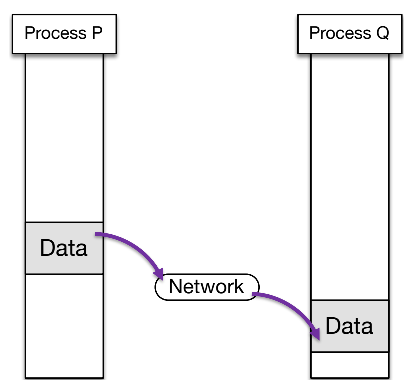
MPI_Recv

```
#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
```

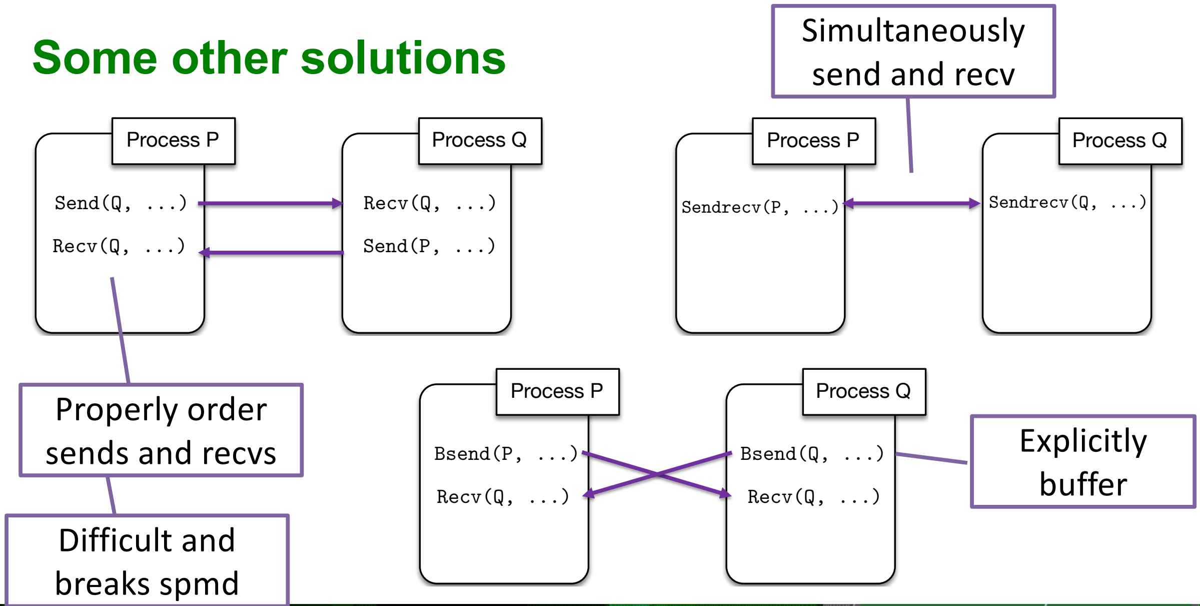
- Blocking receive
- Semantics: Blocks until message is received. On return from call, buffer will have message data

Unbuffered Communication

- Buffering can be avoided
- But we need to make sure it is safe to touch message data
  - Block until it is safe
  - Return before transfer is complete and wait/test later

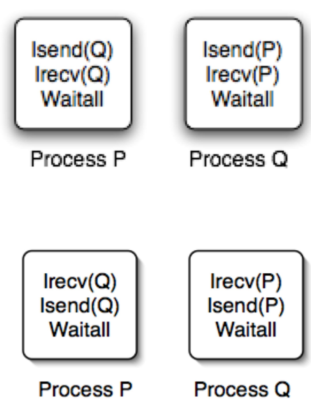


Some other solutions

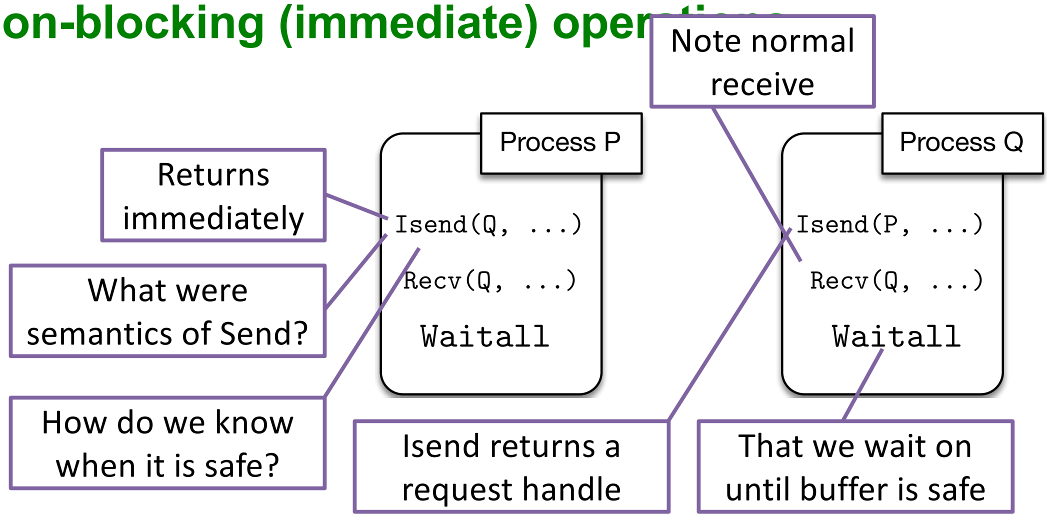


Non-Blocking Operations

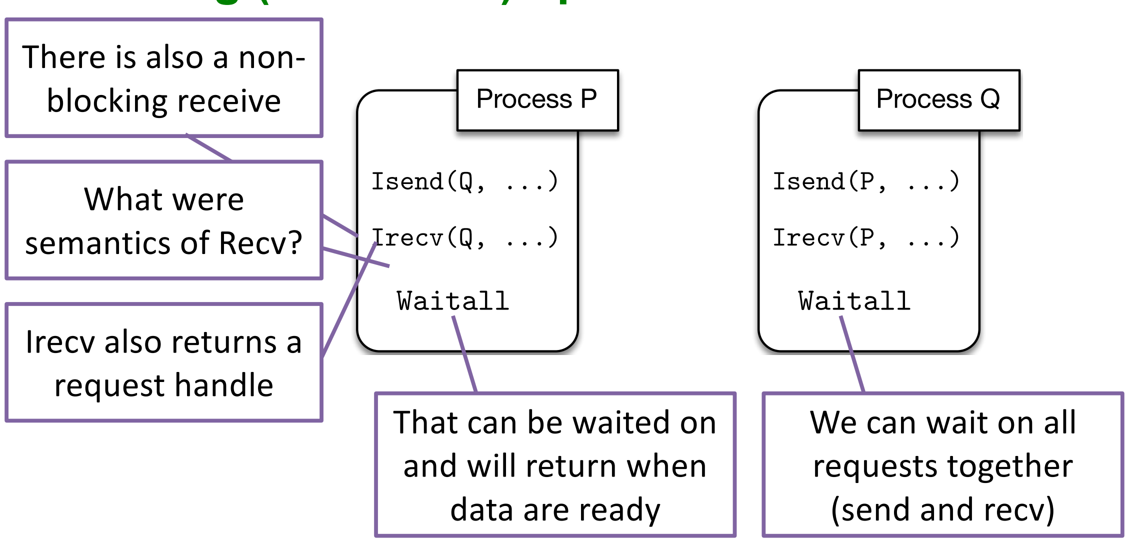
- Non-blocking operations (send and receive) return immediately
- Return "request handles" that can be tested or waited on
- Where progress is made (and where communication happens) is implementation specific



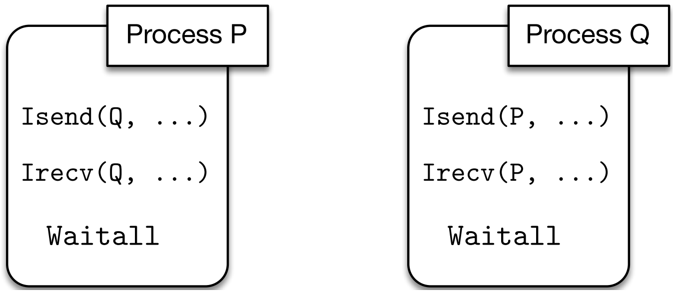
Non-blocking (immediate) operations



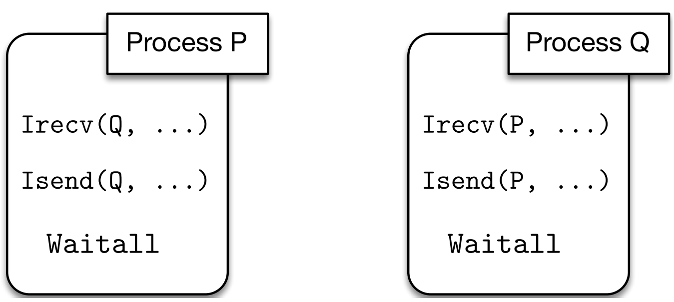
Non-blocking (immediate) operations



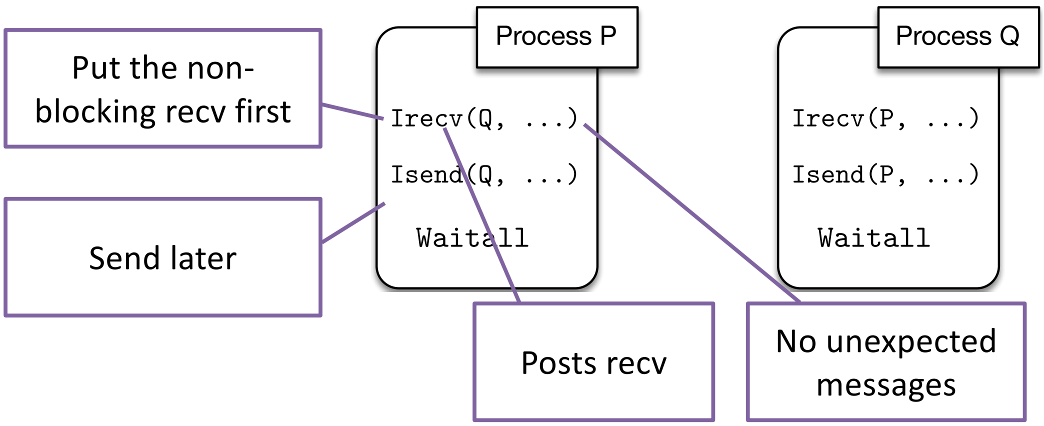
Before



After



After



Bindings for non-blocking receive

```
Request Comm::Isend(const void* buf, int count, const
↪ Datatype& datatype, int dest, int tag) const
```

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

Communication completion: Wait

```
void Request::Wait(Status& status)
void Request::Wait()
```

```
static void Request::Waitall(int count, Request
↪  array_of_requests[], Status array_of_statuses[])
static void Request::Waitall(int count, Request
↪  array_of_requests[])
```

```
static int Request::Waitany(int count, Request
↪  array_of_requests[], Status& status)
static int Request::Waitany(int count, Request
↪  array_of_requests[])
```


Communication completion: Test

```
bool Request::Test(Status& status)
bool Request::Test()
```

```
static bool Request::Testall(int count, Request
↪  array_of_requests[], Status array_of_statuses[])
static bool Request::Testall(int count, Request
↪  array_of_requests[])
```

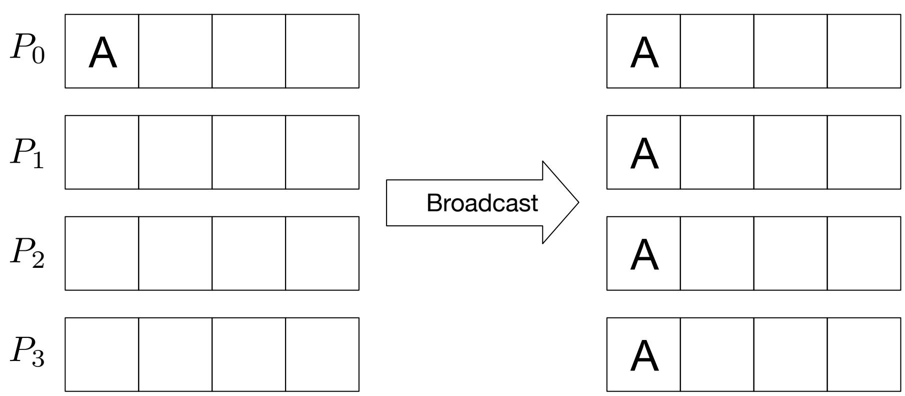
```
static bool Request::Testany(int count, Request
↪  array_of_requests[], int& index, Status& status)
static bool Request::Testany(int count, Request
↪  array_of_requests[], int& index)
```

Collectives

- Collective operations are called by all processes in a communicator.
- **MPI_BCAST** distributes data from one process (the root) to all others in a communicator
- **MPI_REDUCE** combines data from all processes in communicator and returns it to one process
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency

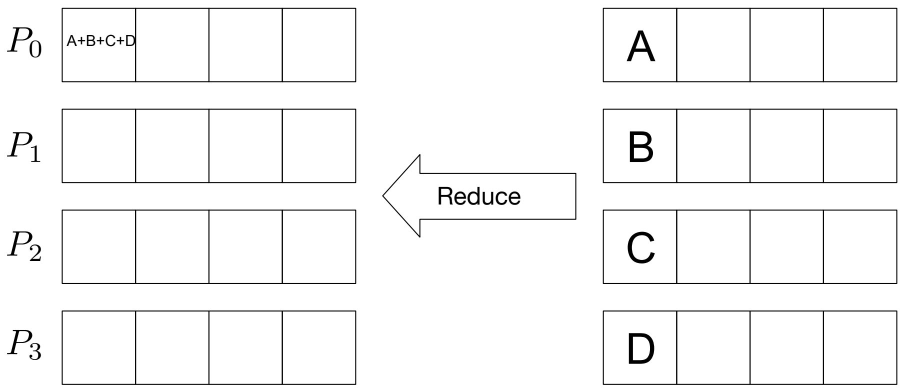
Bcast

```
void MPI::Comm::Bcast(void* buffer, int count, const MPI::Datatype& datatype,
↪  int root) const = 0
```



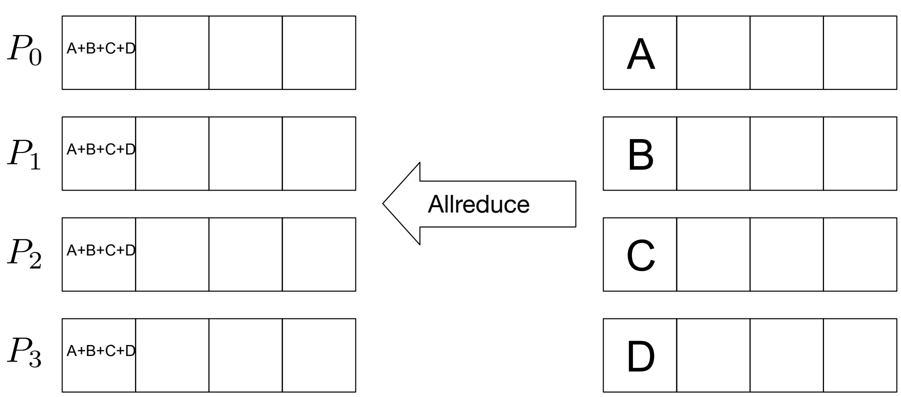
Reduce

```
void MPI::Intracomm::Reduce(const void* sendbuf, void* recvbuf, int count,
↪  const MPI::Datatype& datatype, const MPI::Op& op, int root) const
```



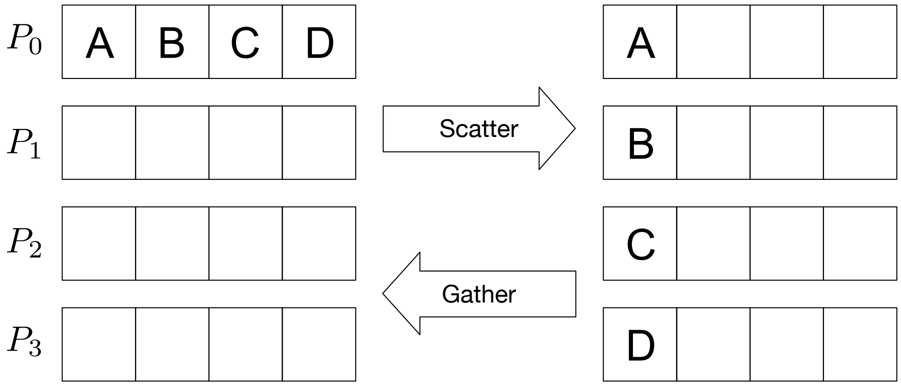
Allreduce

```
void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count, const
↪  MPI::Datatype& datatype, const MPI::Op& op) const=0
```



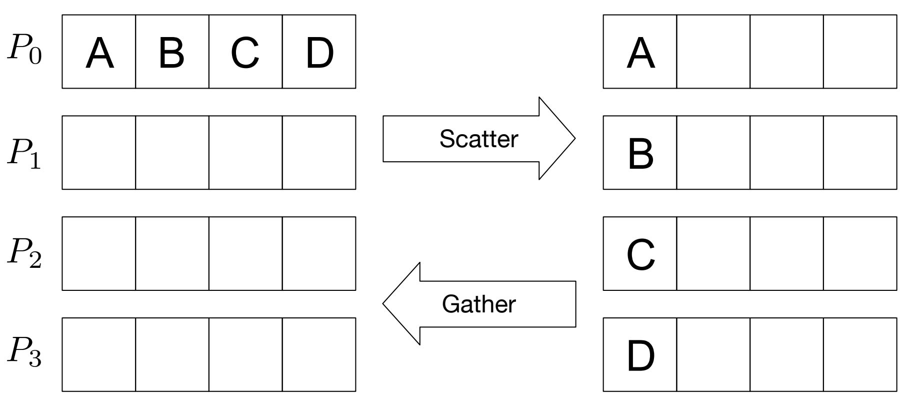
Scatter/Gather

```
void MPI::Comm::Scatter(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype,
↪  void* recvbuf, int recvcount, const MPI::Datatype& recvtype, int root) const
```



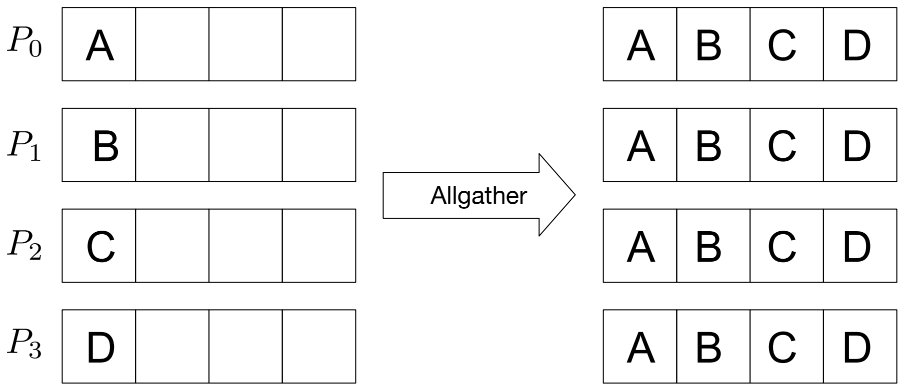
Scatter/Gather

```
void MPI::Comm::Gather(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype,
↪  void* recvbuf, int recvcount, const MPI::Datatype& recvtype, int root, const = 0
```



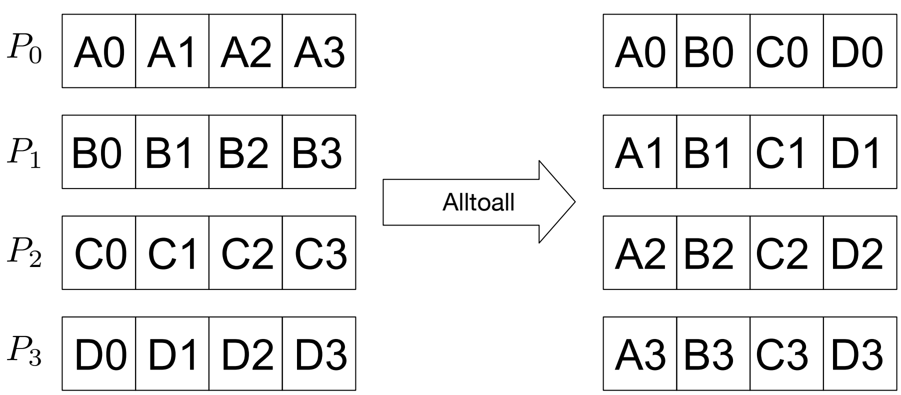
Allgather

```
void MPI::Comm::Allgather(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype,
 ↪  void* recvbuf, int recvcount, const MPI::Datatype& recvtype) const = 0
```



Alltoall

```
void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype,
 ↪  void* recvbuf, int recvcount, const MPI::Datatype& recvtype)
```

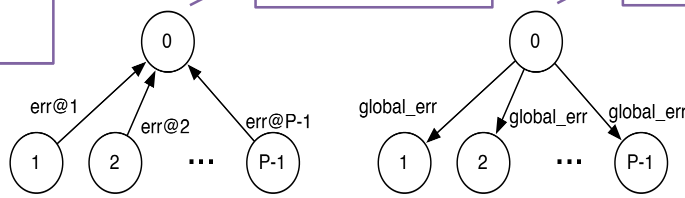


All Reduce

All data is sent to root

How many sends?

How many receives?



Cost? Scalability?

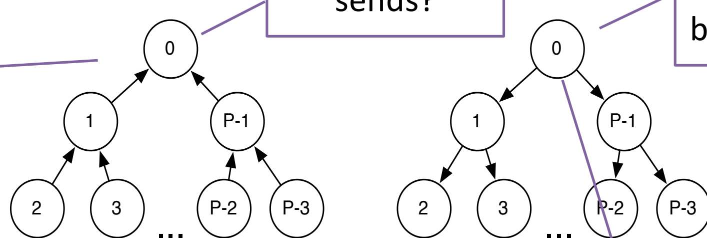
Root sends back out to all

All Reduce

All data is sent to root

How many sends?

Root sends back out to all

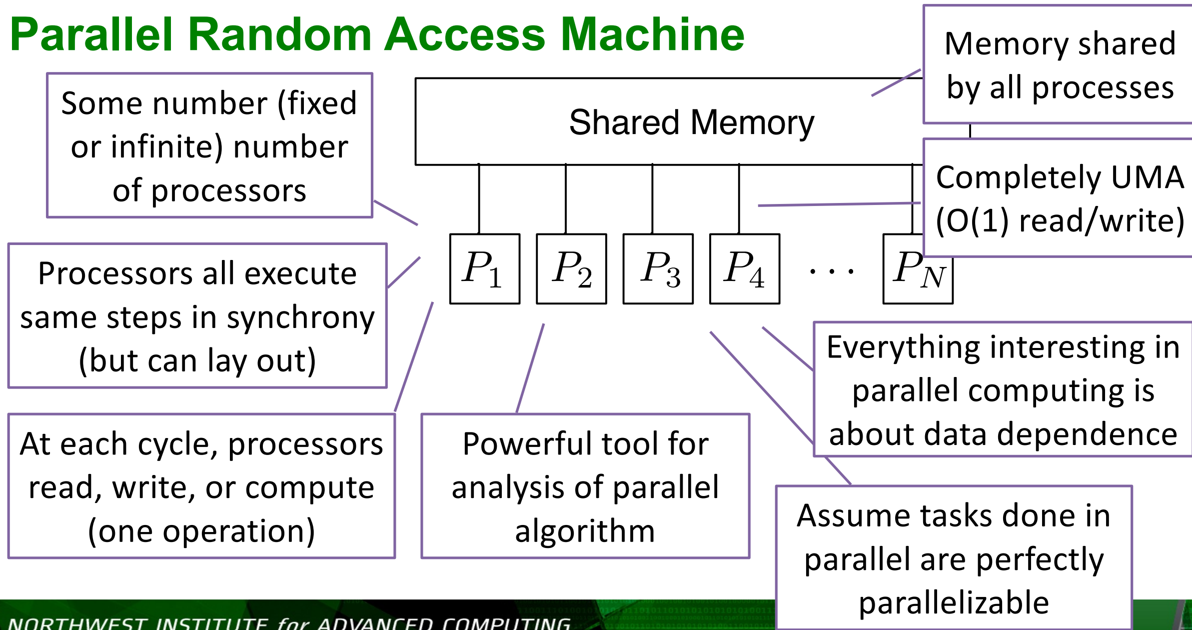


Cost? Scalability?

What is the best ordering?

How many receives?

Parallel Random Access Machine



PRAM cont.

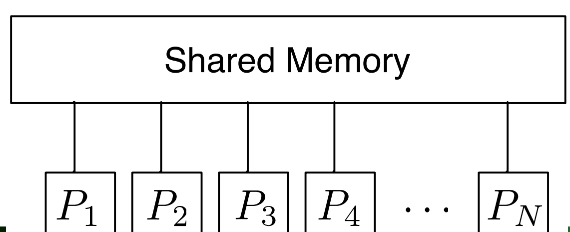
Reads and writes need to be ordered

Writes need to be ordered

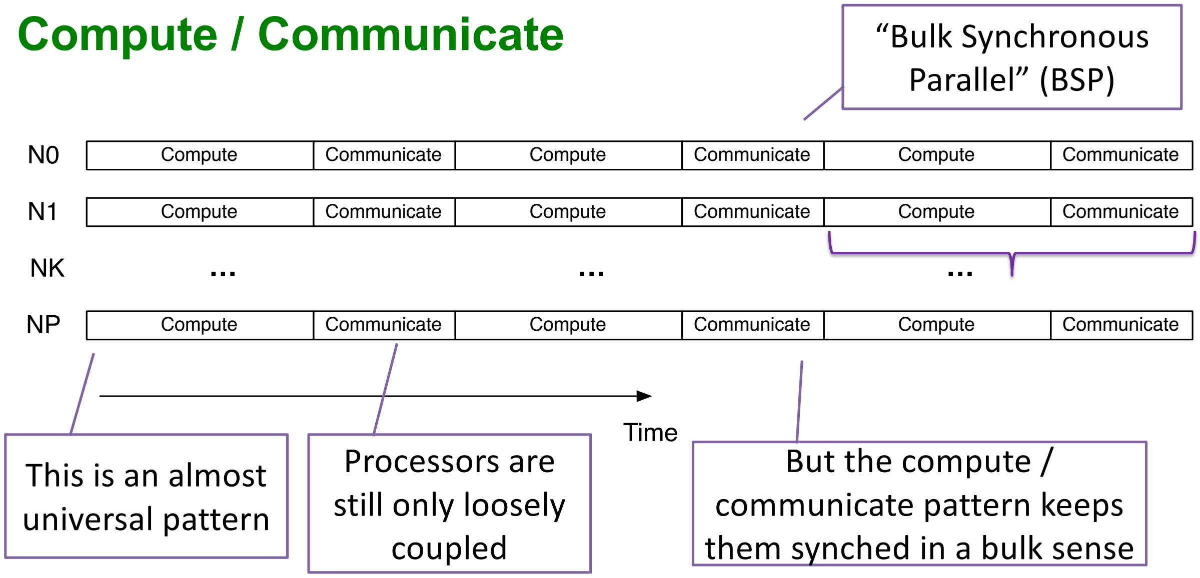
Reads need to be ordered

Nothing needs to be ordered

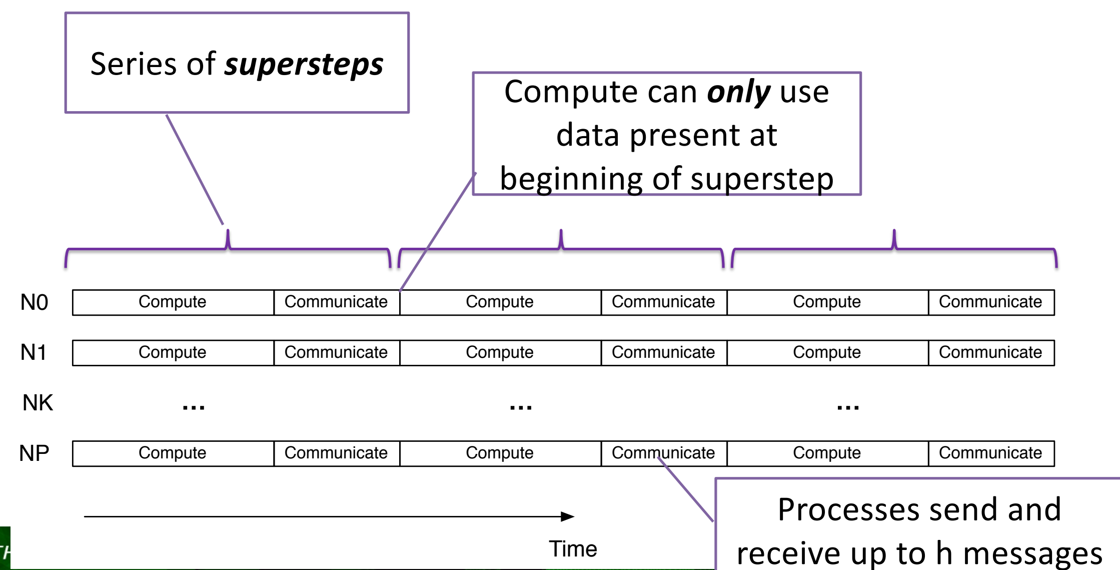
- Several types of PRAM
  - EREW - Exclusive Read Exclusive Write
  - CREW - Concurrent Read Exclusive Write
  - ERCW - Exclusive Read Concurrent Write
  - CRCW - Concurrent Read Concurrent Write
- Stronger models can be emulated by weaker models



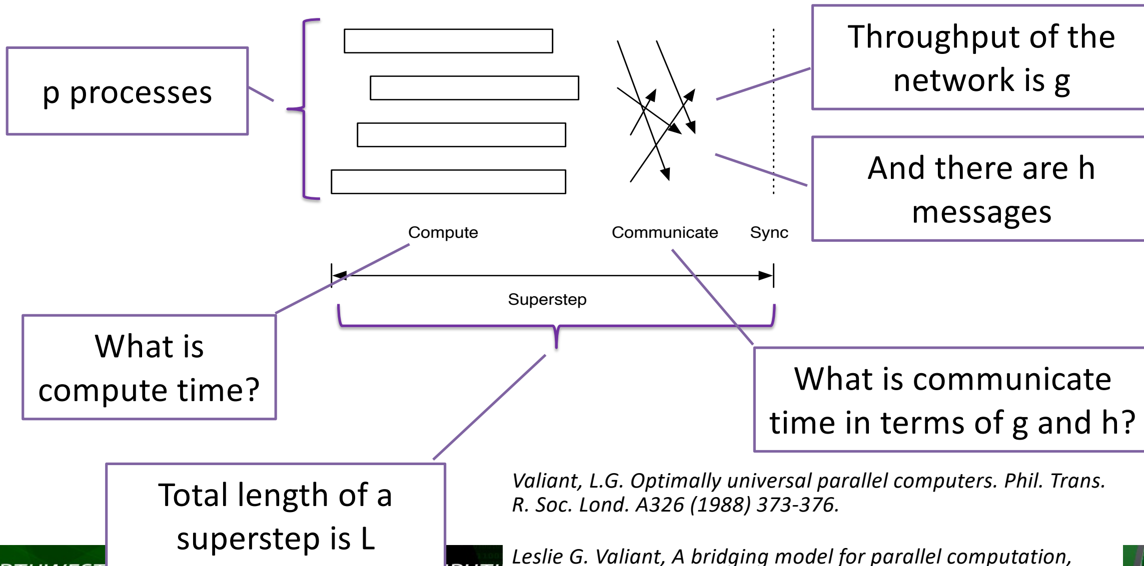
Compute / Communicate



Bulk Synchronous Parallel (BSP)



BSP cont.



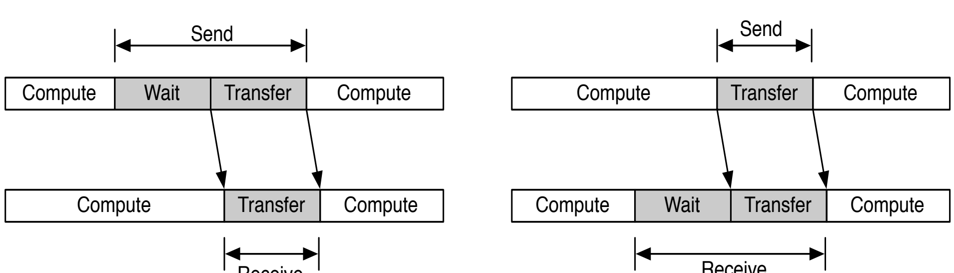
Valiant, L.G. Optimally universal parallel computers. Phil. Trans. R. Soc. Lond. A326 (1988) 373-376.

Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, v.33 n.8, p.103-111, Aug. 1990

Performance Model

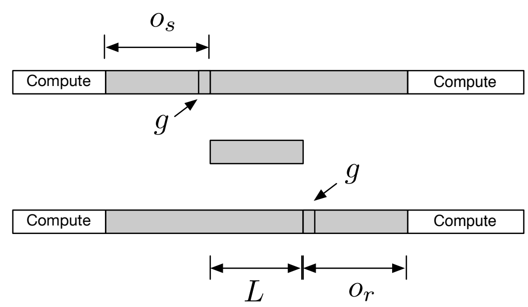
$$T_{communicate} = T_{latency} + T_{bandwidth} = T_L + r_{nic} \cdot Size$$

$$Speedup = \frac{T_{seq}}{T_{parallel}} = \frac{T_{seq}}{T_{compute} + T_{bandwidth} + T_{latency}}$$



LogP

- Parameters (measured in processor cycles)
  - *L* - upper bound on *latency* for a single message
  - *o* - overhead to transmit or receive a message
  - *g* - minimum *gap* between consecutive messages
  - *P* - number of processors



- *Finite capacity* constraint
  - At most $\lceil L/g \rceil$ messages can be in transit from or to any given processor at one time
  - Processors that attempt to exceed this limit stall until the message can be sent

LogP

- Sending single message

  $$T = 2o + L$$

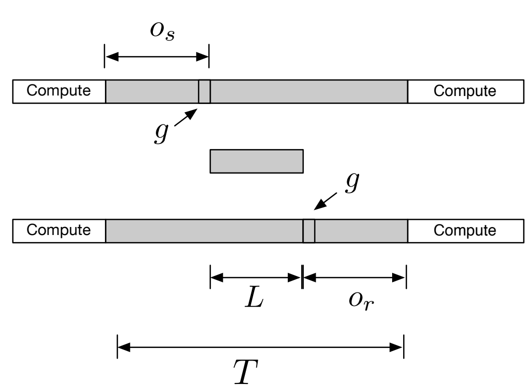
- Ping-pong round trip

  $$T = 4o + 2L$$

- N messages in a row

  $$T = L + (n-1)\max(g, o) + 2o$$

Why?



LogP cont.

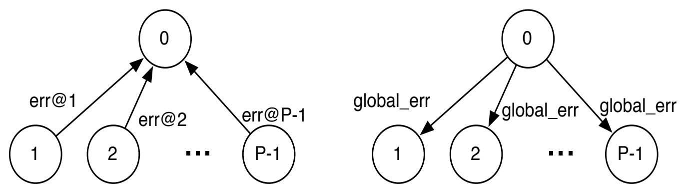
- More coarse grained than PRAM
  - PRAM = LogP with (L = 0, o = 0, g = 0)
- More fine grained than BSP
- Allows more precise scheduling of communication
  - Reading a remote memory location
    - BSP - next superstep, L cycles
    - LogP - $2L + 4o$ cycles
- No special synchronization hardware
- Parameters can be experimentally determined for a given machine/architecture
- No special treatment for long messages

Applications: Reduce

- PRAM
  - EREW/CREW
    - Binary tree - O(log n)
  - CRCW
    - Arbitrary succeed
      - Binary tree - O(log n)
    - Arbitrary operation
      - All procs write one memory location - O(1)
- BSP
  - O(log n) supersteps
  - L = time to read two memory locations and write one

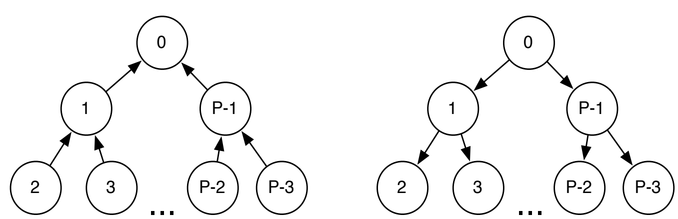
LogP Analysis

- Linear reduce
  - o for each processor to send it's value to the root
  - (P-1)o + L for the root to receive them
  - o + (P-1)*max{g,o} + L



LogP Analysis

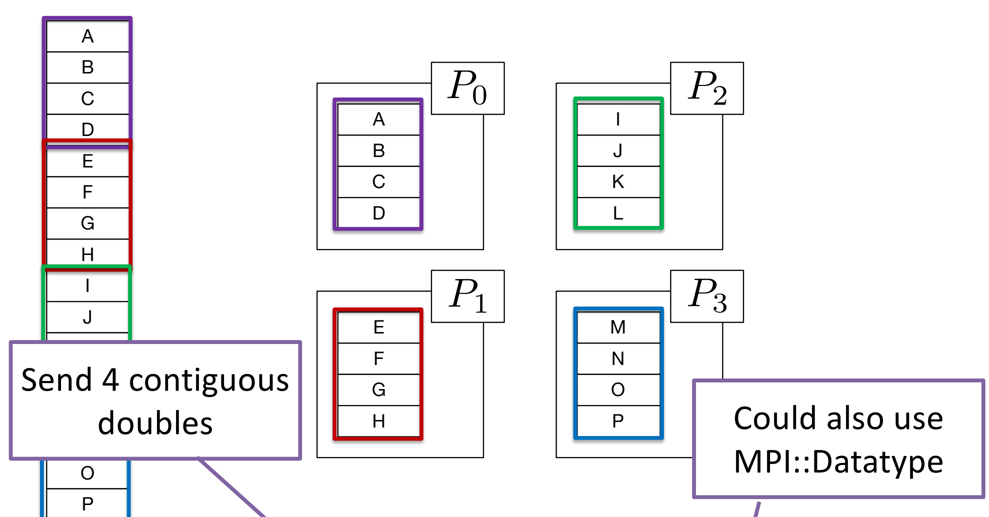
- Binary tree
  - o for each leaf proc to send its value to it's parent
  - o + max{g,o} + L + o for each non-leaf processor to receive values from each of its children and send the result to its parent
  - o + (log P)(o + max{g,o} + L + o)



Parallel Matrix-Matrix Multiply

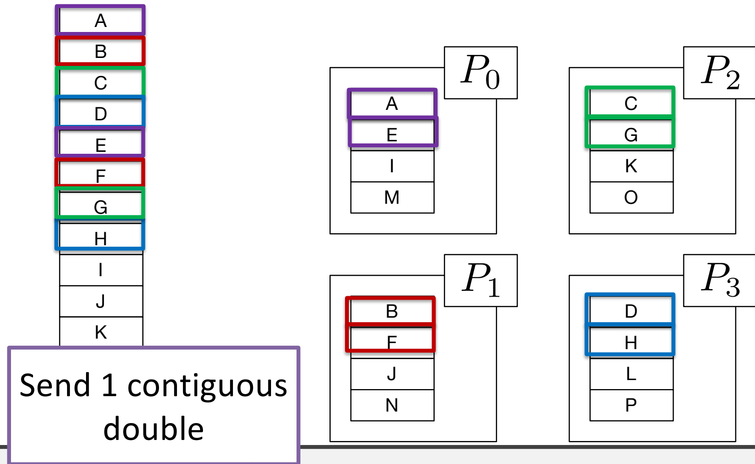
- Use block algorithm
- Partition matrix into blocks
- Assign blocks to processors
- Orchestrate communication and computation
- ***Owner computes***

Block Partitioning



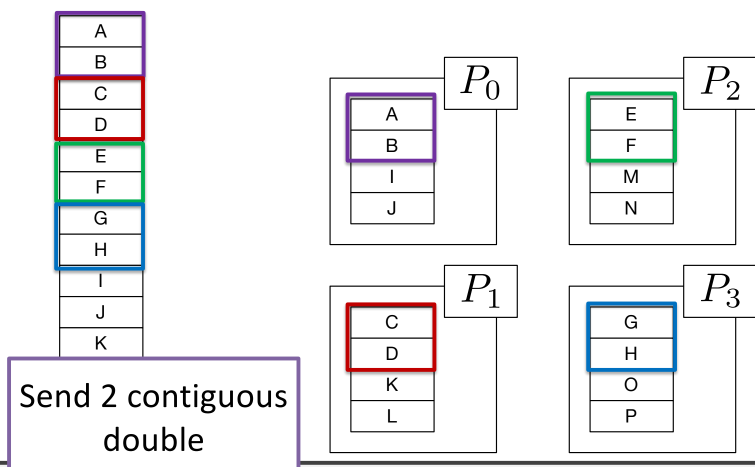

```
MPI::COMM_WORLD.Scatter(&x(0), 4, MPI::DOUBLE, &x(0), 4, MPI::DOUBLE, 0);
```

Cyclic Partitioning




```
for (size_t i = 0; i < 4; ++i) {
  MPI::COMM_WORLD.Scatter(&x(i*4), 1, MPI::DOUBLE, &x(i*4), 1, MPI::DOUBLE, 0);
}
```

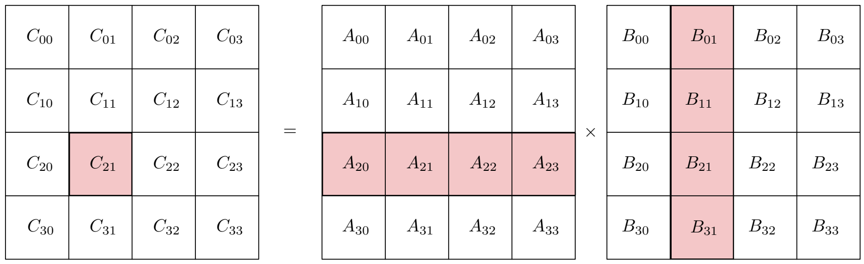

Block Cyclic Partitioning




```
for (size_t i = 0; i < 2; ++i) {
  MPI::COMM_WORLD.Scatter(&x(i*8), 2, MPI::DOUBLE, &x(i*8), 2, MPI::DOUBLE, 0);
}
```

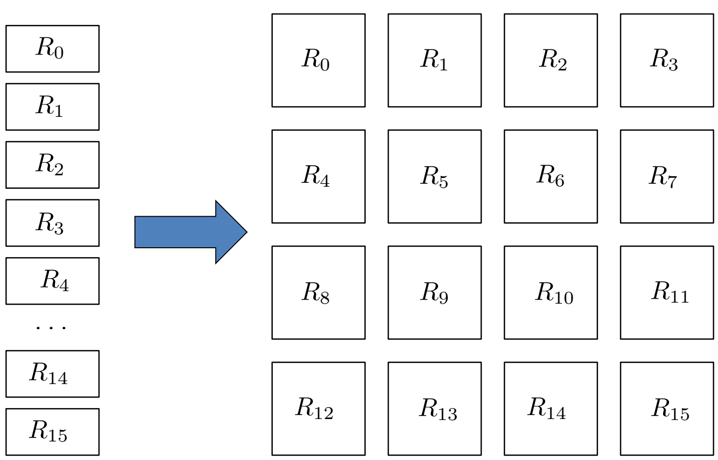
Block Matrix-Matrix Product

$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

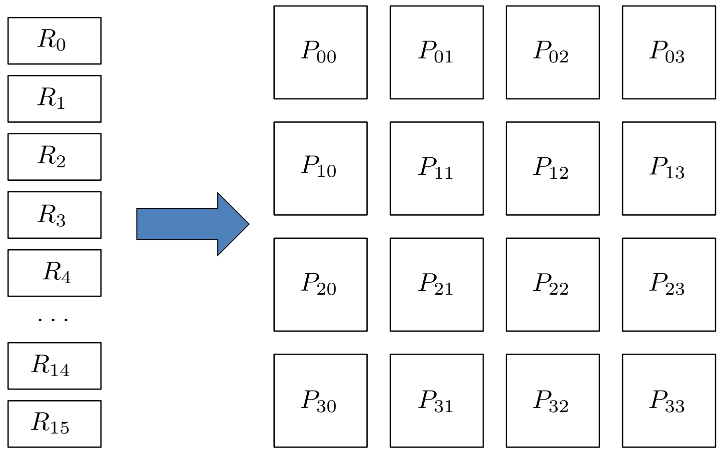


$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

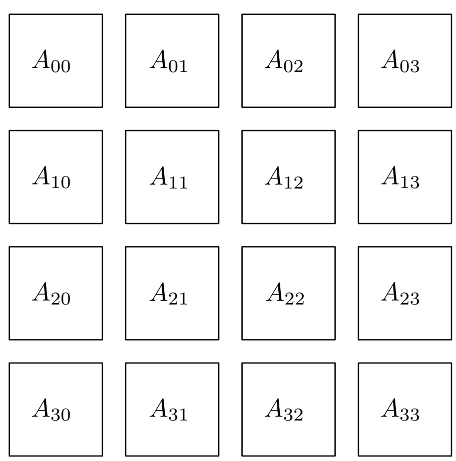
Processor Grid



Processor Grid



Matrix Block Partitioning



Matrix Block Partitioning

A_{00} B_{00}	A_{01} B_{01}	A_{02} B_{02}	A_{03} B_{03}
A_{10} B_{10}	A_{11} B_{11}	A_{12} B_{12}	A_{13} B_{13}
A_{20} B_{20}	A_{21} B_{21}	A_{22} B_{22}	A_{23} B_{23}
A_{30} B_{30}	A_{31} B_{31}	A_{32} B_{32}	A_{33} B_{33}

Matrix Block Partitioning

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

Matrix Block Partitioning

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

$C_{IJ} = \sum_K A_{IK} B_{KJ}$ (Owner computes)

$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

Matrix Block Partitioning

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

$C_{IJ} = \sum_K A_{IK} B_{KJ}$ (Owner computes)

$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

Matrix Block Partitioning

$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$ (Owner computes)

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

Matrix Block Partitioning

$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$ (Owner computes)

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

Matrix Block Partitioning

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

$C_{IJ} = \sum_K A_{IK} B_{KJ}$ (Owner computes)

$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

Matrix Block Partitioning

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{10} B_{10}	C_{11} A_{11} B_{11}	C_{12} A_{12} B_{12}	C_{13} A_{13} B_{13}
C_{20} A_{20} B_{20}	C_{21} A_{21} B_{21}	C_{22} A_{22} B_{22}	C_{23} A_{23} B_{23}
C_{30} A_{30} B_{30}	C_{31} A_{31} B_{31}	C_{32} A_{32} B_{32}	C_{33} A_{33} B_{33}

$C_{IJ} = \sum_K A_{IK} B_{KJ}$ (Owner computes)

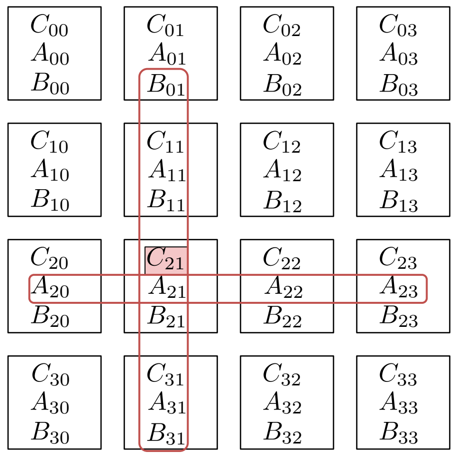
- At each step K, arrange for

  $A_{I,(I+J+K)}$ $B_{(I+J+K),J}$

  to be on processor I,J

$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

Cannon's Algorithm



$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

$$C_{IJ} = \sum_K A_{IK}B_{KJ}$$

- At each step K, arrange for

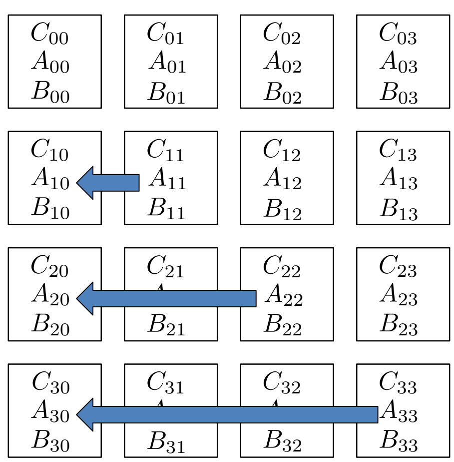
  $A_{I,(I+J+K)}$ $\qquad$ $B_{(I+J+K),J}$

  to be on processor I,J

- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

Cannon's Algorithm: Setup (K = 0)



$$C_{IJ} = \sum_K A_{IK}B_{KJ}$$

- At each step K, arrange for

  $A_{I,(I+J+K)}$ $\qquad$ $B_{(I+J+K),J}$

  to be on processor I,J

- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

Cannon's Algorithm: Setup (K = 0)

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{01}	C_{02} A_{02} B_{02}	C_{03} A_{03} B_{03}
C_{10} A_{11} B_{10}	C_{11} A_{12} B_{11}	C_{12} A_{13} B_{12}	C_{13} A_{10} B_{13}
C_{20} A_{22} B_{20}	C_{21} A_{23} B_{21}	C_{22} A_{20} B_{22}	C_{23} A_{21} B_{23}
C_{30} A_{33} B_{30}	C_{31} A_{30} B_{31}	C_{32} A_{31} B_{32}	C_{33} A_{32} B_{33}

$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

- At each step K, arrange for

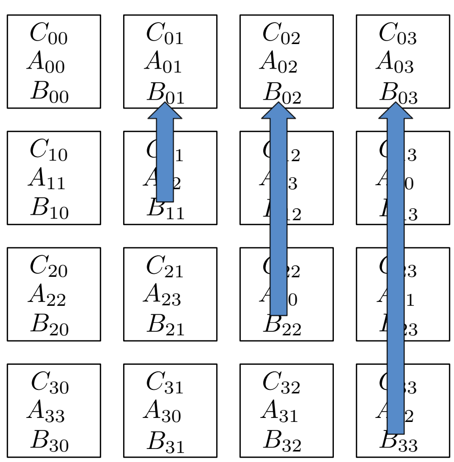
  $A_{I,(I+J+K)}$ $B_{(I+J+K),J}$

  to be on processor I,J

- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

Cannon's Algorithm: Setup (K = 0)



$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

- At each step K, arrange for

  $A_{I,(I+J+K)}$ $B_{(I+J+K),J}$

  to be on processor I,J

- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

Cannon's Algorithm: Setup

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{11}	C_{02} A_{02} B_{22}	C_{03} A_{03} B_{33}
C_{10} A_{11} B_{10}	C_{11} A_{12} B_{21}	C_{12} A_{13} B_{32}	C_{13} A_{10} B_{03}
C_{20} A_{22} B_{20}	C_{21} A_{23} B_{31}	C_{22} A_{20} B_{02}	C_{23} A_{21} B_{13}
C_{30} A_{33} B_{30}	C_{31} A_{30} B_{01}	C_{32} A_{31} B_{12}	C_{33} A_{32} B_{23}

$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

- At each step K, arrange for

  $A_{I,(I+J+K)}$ $\quad$ $B_{(I+J+K),J}$

  to be on processor I,J
- Compute

  $$C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$$

Cannon's Algorithm: K = 0

C_{00} A_{00} B_{00}	C_{01} A_{01} B_{11}	C_{02} A_{02} B_{22}	C_{03} A_{03} B_{33}
C_{10} A_{11} B_{10}	C_{11} A_{12} B_{21}	C_{12} A_{13} B_{32}	C_{13} A_{10} B_{03}
C_{20} A_{22} B_{20}	C_{21} A_{23} B_{31}	C_{22} A_{20} B_{02}	C_{23} A_{21} B_{13}
C_{30} A_{33} B_{30}	C_{31} A_{30} B_{01}	C_{32} A_{31} B_{12}	C_{33} A_{32} B_{23}

$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

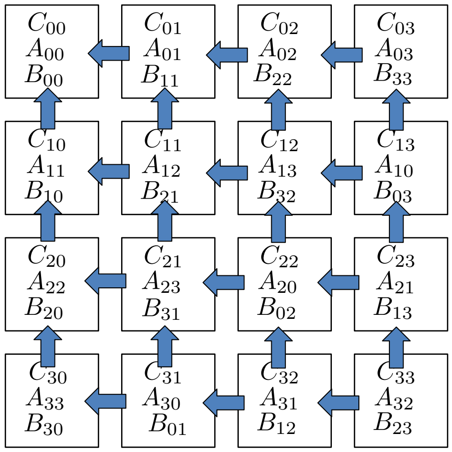
- At each step K, arrange for

  $A_{I,(I+J+K)}$ $\quad$ $B_{(I+J+K),J}$

  to be on processor I,J
- Compute

  $$C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$$

Cannon's Algorithm: K = 1



$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

- At each step K, arrange for

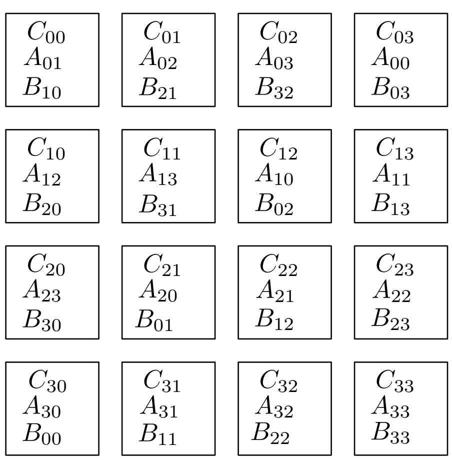
  $A_{I,(I+J+K)}$ $B_{(I+J+K),J}$

  to be on processor I,J

- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

Cannon's Algorithm: K = 1



$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

- At each step K, arrange for

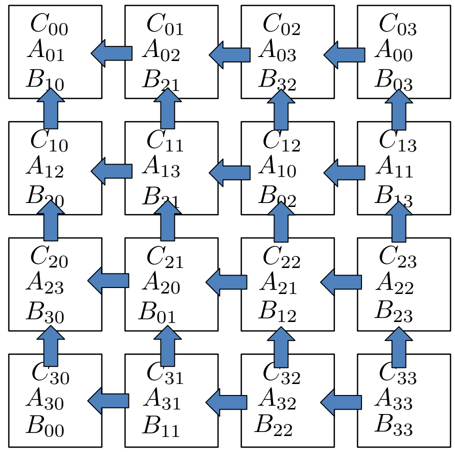
  $A_{I,(I+J+K)}$ $B_{(I+J+K),J}$

  to be on processor I,J

- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

Cannon's Algorithm: K = 2



$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

- At each step K, arrange for

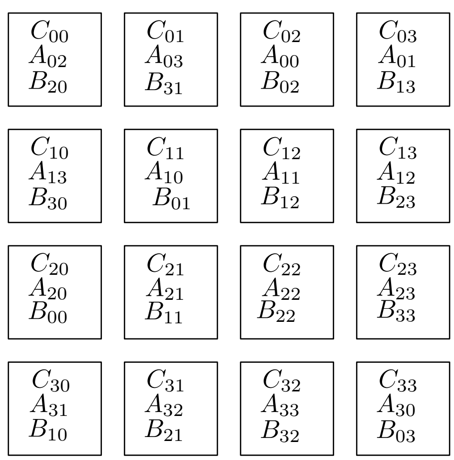
  $A_{I,(I+J+K)}$ $B_{(I+J+K),J}$

  to be on processor I,J

- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

Cannon's Algorithm: K = 2



$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

- At each step K, arrange for

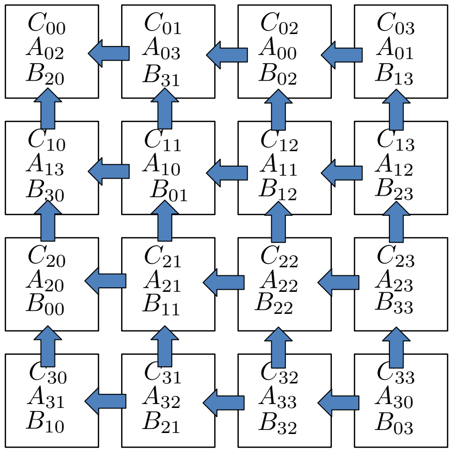
  $A_{I,(I+J+K)}$ $B_{(I+J+K),J}$

  to be on processor I,J

- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

Cannon's Algorithm: K = 3



$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

- At each step K, arrange for

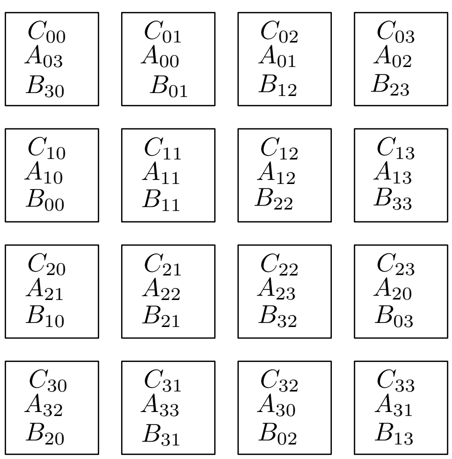
  $A_{I,(I+J+K)}$ $B_{(I+J+K),J}$

  to be on processor I,J

- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

Cannon's Algorithm: K = 3



$$C_{IJ} = \sum_K A_{IK} B_{KJ}$$

- At each step K, arrange for

  $A_{I,(I+J+K)}$ $B_{(I+J+K),J}$

  to be on processor I,J

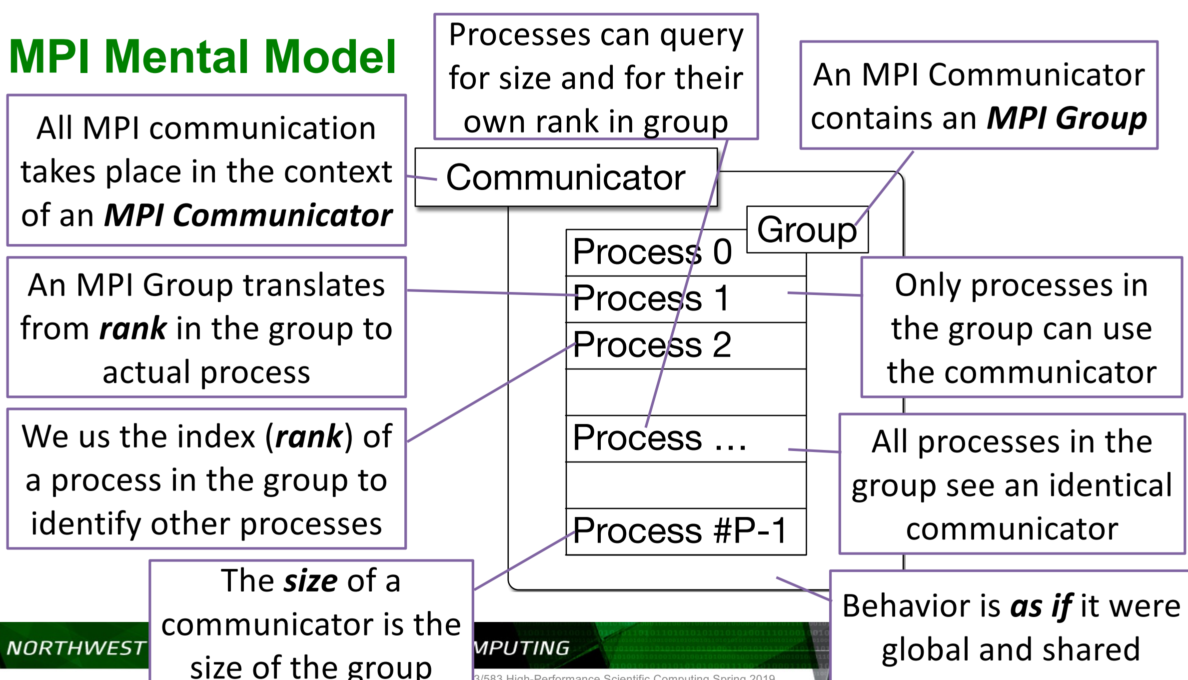
- Compute

  $C_{IJ} \mathrel{+}= A_{I,(I+J+K)} \times B_{(I+J+K),J}$

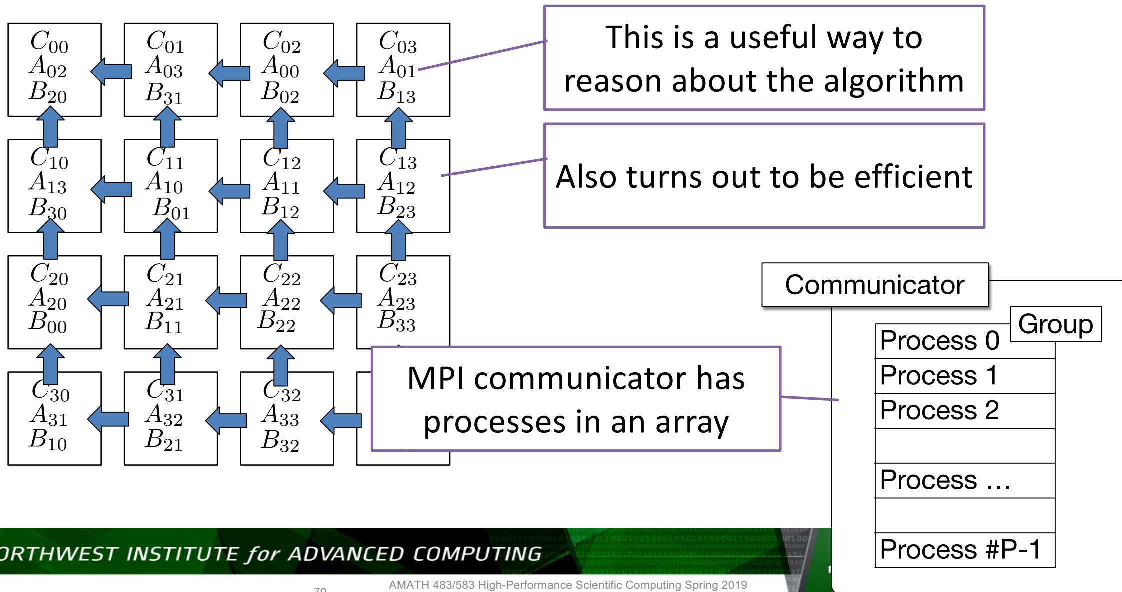
Implementation

- Two-D decomposition of matrices A, B, C
- Move A and B to starting positions
- Local matrix-matrix product
- Shift left
- Shift up
- Move A and B back to initial distributions

MPI Mental Model

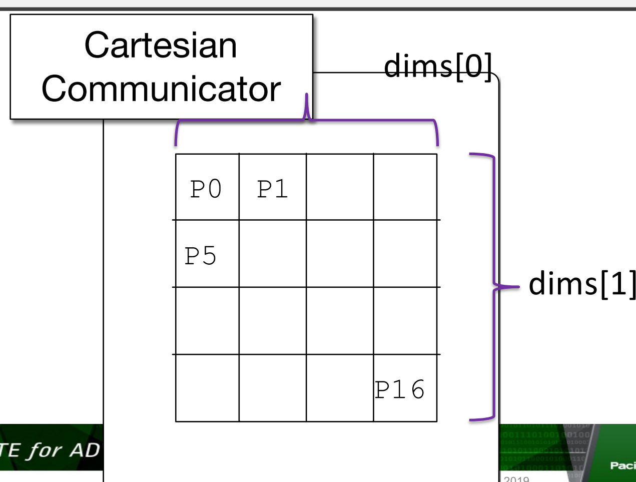


Shifting North, East, West, South



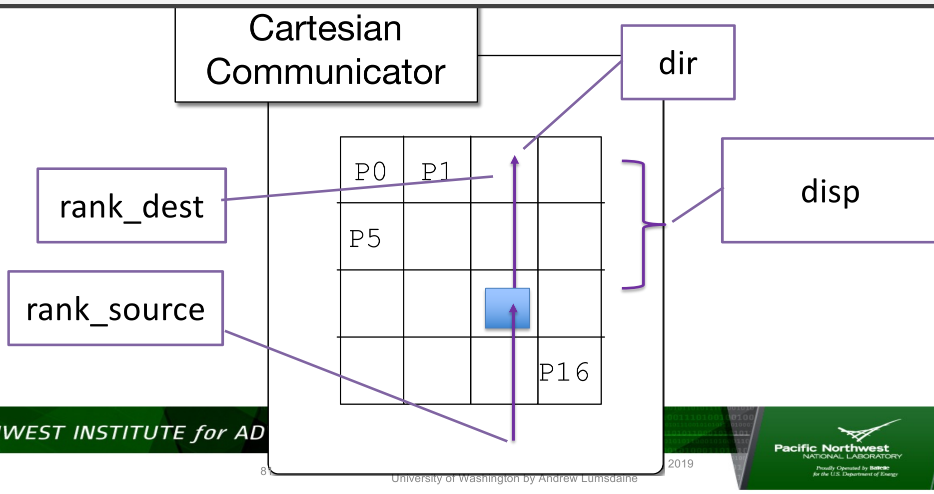
Cartesian Communicator

```
Cartcomm Intracomm.Create_cart(int ndims, int dims[], const bool periods[],
    bool reorder) const
```



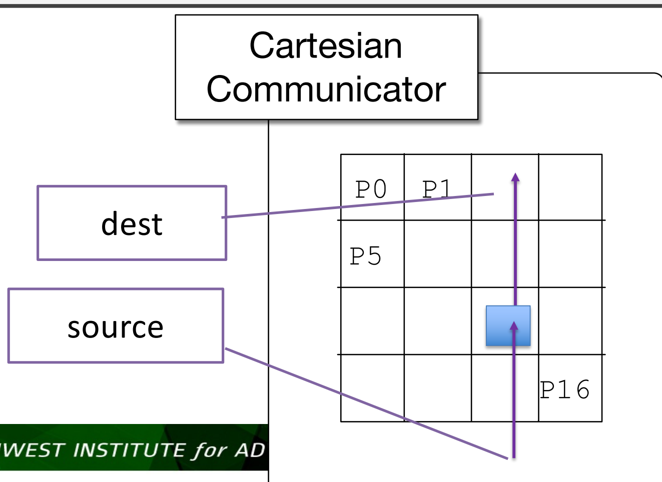
Cartesian Communicator

```
void Cartcomm::Shift(int direction, int disp, int& rank_source,
↪  int& rank_dest) const
```



Cartesian Communicator

```
void Comm::Sendrecv_replace(void* buf, int count, const Datatype& datatype,
↪  int dest, int sendtag, int source, int recvtag) const
```



Implementation

```cpp
void cannonMultiplyMV(const Matrix& A, const Matrix& B, Matrix& C) {
  size_t mysize = MPI::COMM_WORLD.Get_size();

  // Set up grid topology and a grid (Cartesian) communicator
  int dims[2] = { (int) std::sqrt(mysize), (int) std::sqrt(mysize) };
  bool periods[2] = { true, true };

  MPI::Cartcomm gridComm = MPI::COMM_WORLD.Create_cart(2, dims, periods, true);
  size_t myrank = gridComm.Get_rank();

  int mycoords[2];
  gridComm.Get_coords(myrank, 2, mycoords);

  int northRank, eastRank, westRank, southRank;
  gridComm.Shift(0, -1, westRank,  eastRank);
  gridComm.Shift(1, -1, southRank, northRank);

  // Move A and B where they need to be to start
  int shiftSource, shiftDest;
  gridComm.Shift(0, -mycoords[0], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&A(0,0)), A.numRows()*A.numCols(),
                            MPI::DOUBLE, shiftDest, 314, shiftSource, 314);

  gridComm.Shift(1, -mycoords[1], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&B(0,0)), B.numRows()*B.numCols(),
                            MPI::DOUBLE, shiftDest, 314, shiftSource, 315);


  // Main loop
  for (int k = 0; k < dims[0]; ++k) {
    hoistedCopyBlockedTiledMultiply2x2(A, B, C); // Local block matmat

    gridComm.Sendrecv_replace(const_cast<double*>(&A(0,0)), A.numRows()*A.numCols(),
                              MPI::DOUBLE, westRank,  316, eastRank,  316);
    gridComm.Sendrecv_replace(const_cast<double*>(&B(0,0)), B.numRows()*B.numCols(),
                              MPI::DOUBLE, northRank, 317, southRank, 317);
  }

  // Restore A and B to initial distribution
  gridComm.Shift(0, +mycoords[0], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&A(0,0)), A.numRows()*A.numCols(),
                            MPI::DOUBLE, shiftDest, 318, shiftSource, 318);

  gridComm.Shift(1, +mycoords[1], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&B(0,0)), B.numRows()*B.numCols(),
                            MPI::DOUBLE, shiftDest, 319, shiftSource, 319);

  gridComm.Free();
}
```


Implementation

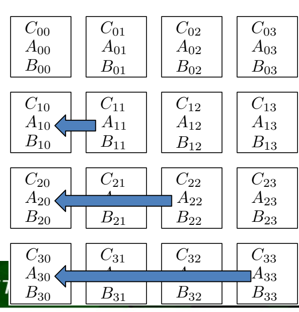
```cpp
void cannonMultiplyMV(const Matrix& A, const Matrix& B, Matrix& C) {
  size_t mysize = MPI::COMM_WORLD.Get_size();

  // Set up grid topology and a grid (Cartesian) communicator
  int dims[2] = { (int) std::sqrt(mysize), (int) std::sqrt(mysize) };
  bool periods[2] = { true, true };

  MPI::Cartcomm gridComm = MPI::COMM_WORLD.Create_cart(2, dims, periods, true);
  size_t myrank = gridComm.Get_rank();

  int mycoords[2];
  gridComm.Get_coords(myrank, 2, mycoords);

  int northRank, eastRank, westRank, southRank;
  gridComm.Shift(0, -1, westRank,  eastRank);
  gridComm.Shift(1, -1, southRank, northRank);

  // Move A and B where they need to be to start
  int shiftSource, shiftDest;
  gridComm.Shift(0, -mycoords[0], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&A(0,0)), A.numRows()*A.numCols(),
                            MPI::DOUBLE, shiftDest, 314, shiftSource, 314);

  gridComm.Shift(1, -mycoords[1], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&B(0,0)), B.numRows()*B.numCols(),
                            MPI::DOUBLE, shiftDest, 314, shiftSource, 315);


  // Main loop
  for (int k = 0; k < dims[0]; ++k) {
    hoistedCopyBlockedTiledMultiply2x2(A, B, C); // Local block matmat

    gridComm.Sendrecv_replace(const_cast<double*>(&A(0,0)), A.numRows()*A.numCols(),
                              MPI::DOUBLE, westRank,  316, eastRank,  316);
    gridComm.Sendrecv_replace(const_cast<double*>(&B(0,0)), B.numRows()*B.numCols(),
                              MPI::DOUBLE, northRank, 317, southRank, 317);
  }

  // Restore A and B to initial distribution
  gridComm.Shift(0, +mycoords[0], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&A(0,0)), A.numRows()*A.numCols(),
                            MPI::DOUBLE, shiftDest, 318, shiftSource, 318);

  gridComm.Shift(1, +mycoords[1], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&B(0,0)), B.numRows()*B.numCols(),
                            MPI::DOUBLE, shiftDest, 319, shiftSource, 319);

  gridComm.Free();
}
```

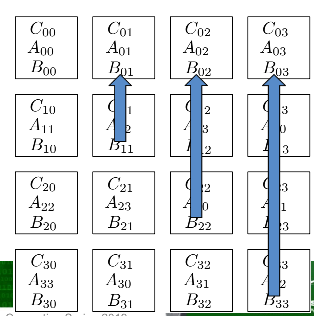
Implementation

```
void cannonMultiplyMV(const Matrix& A, const Matrix& B, Matrix& C) {
  size_t mysize = MPI::COMM_WORLD.Get_size();

  // Set up grid topology and a grid (Cartesian) communicator
  int dims[2] = { (int) std::sqrt(mysize), (int) std::sqrt(mysize) };
  bool periods[2] = { true, true };

  MPI::Cartcomm gridComm = MPI::COMM_WORLD.Create_cart(2, dims, periods, true);
  size_t myrank = gridComm.Get_rank();

  int mycoords[2];
  gridComm.Get_coords(myrank, 2, mycoords);

  int northRank, eastRank, westRank, southRank;
  gridComm.Shift(0, -1, westRank,  eastRank);
  gridComm.Shift(1, -1, southRank, northRank);
```


Implementation

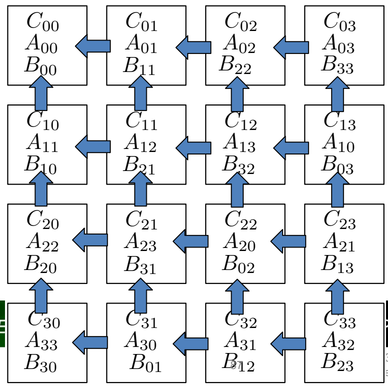
```

  // Move A and B where they need to be to start
  int shiftSource, shiftDest;
  gridComm.Shift(0, -mycoords[0], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&A(0,0)), A.numRows()*A.numCols(),
                            MPI::DOUBLE, shiftDest, 314, shiftSource, 314);

  gridComm.Shift(1, -mycoords[1], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&B(0,0)), B.numRows()*B.numCols(),
                            MPI::DOUBLE, shiftDest, 314, shiftSource, 315);

```





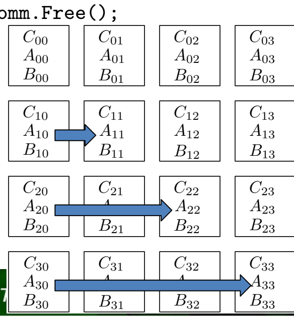
Implementation

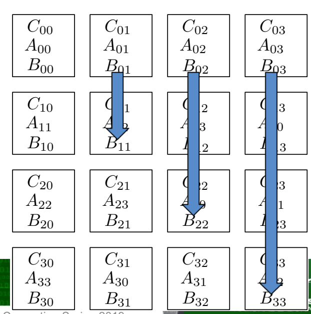
```
// Main loop
for (int k = 0; k < dims[0]; ++k) {
  hoistedCopyBlockedTiledMultiply2x2(A, B, C); // Local block matmat

  gridComm.Sendrecv_replace(const_cast<double*>(&A(0,0)), A.numRows()*A.numCols(),
                            MPI::DOUBLE, westRank,  316, eastRank,  316);
  gridComm.Sendrecv_replace(const_cast<double*>(&B(0,0)), B.numRows()*A.numCols(),
                            MPI::DOUBLE, northRank, 317, southRank, 317);
}
```



Implementation

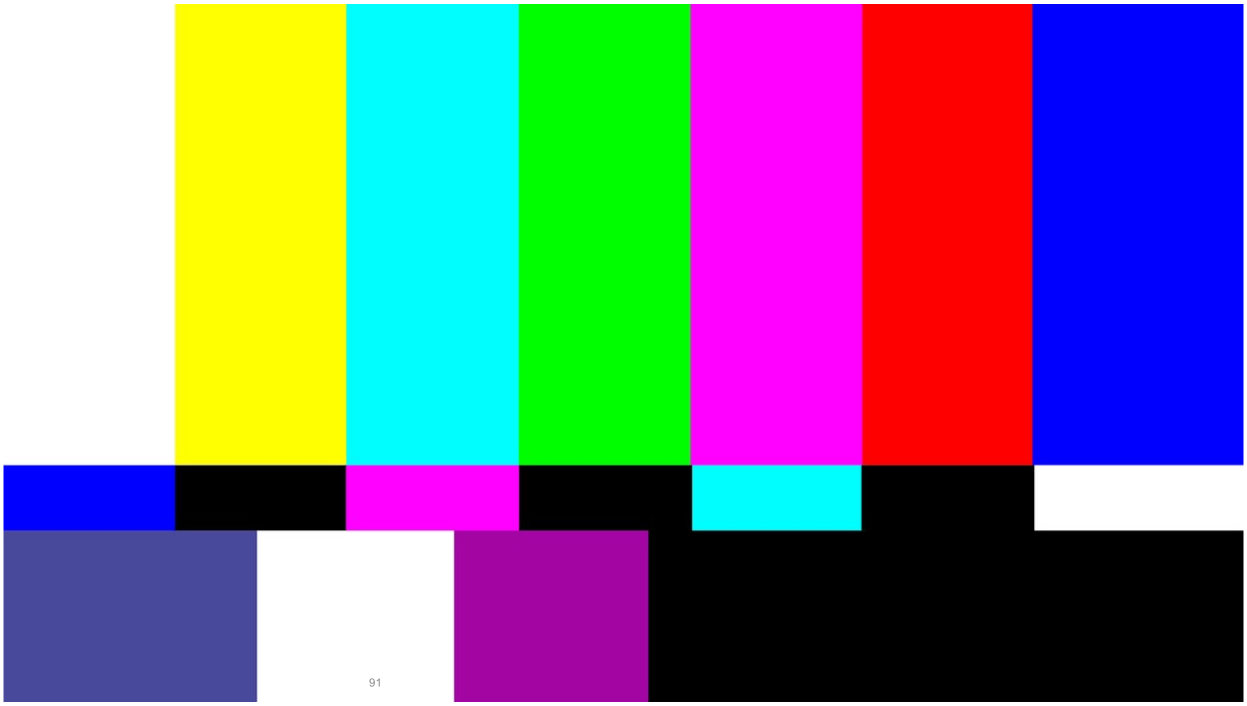
```
  // Restore A and B to initial distribution
  gridComm.Shift(0, +mycoords[0], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&A(0,0)), A.numRows()*A.numCols(),
                            MPI::DOUBLE, shiftDest, 318, shiftSource, 318);

  gridComm.Shift(1, +mycoords[1], shiftSource, shiftDest);
  gridComm.Sendrecv_replace(const_cast<double*>(&B(0,0)), B.numRows()*B.numCols(),
                            MPI::DOUBLE, shiftDest, 319, shiftSource, 319);

  gridComm.Free();
}
```



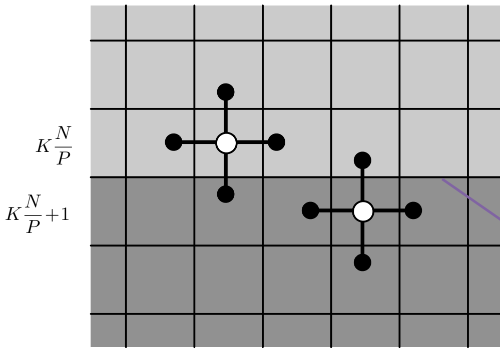


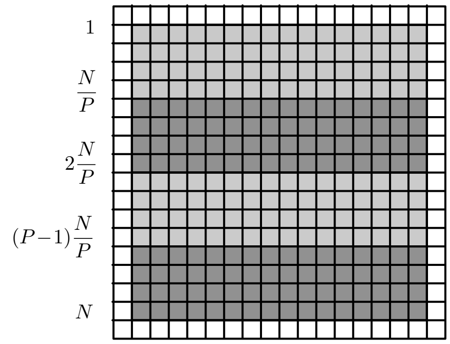
Thank You!

Creative Commons BY-NC-SA 4.0 License



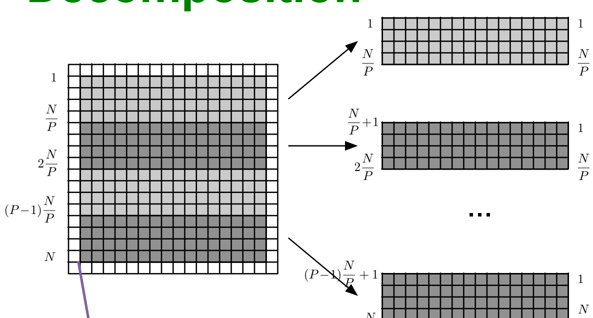
Decomposition

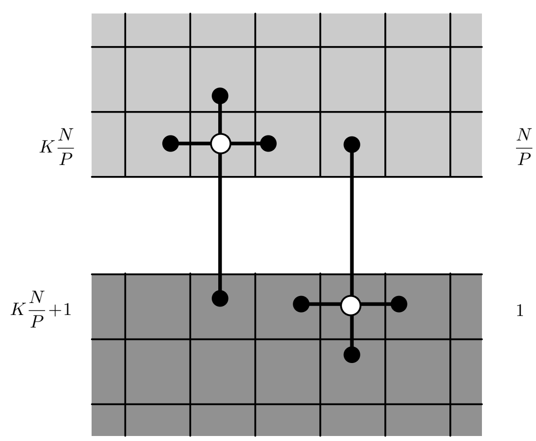




Data dependencies for stencil crosses partition boundary in original problem

Decomposition



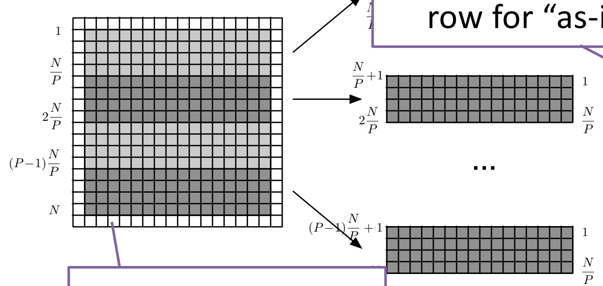


Where did the boundary go

When we partitioned?

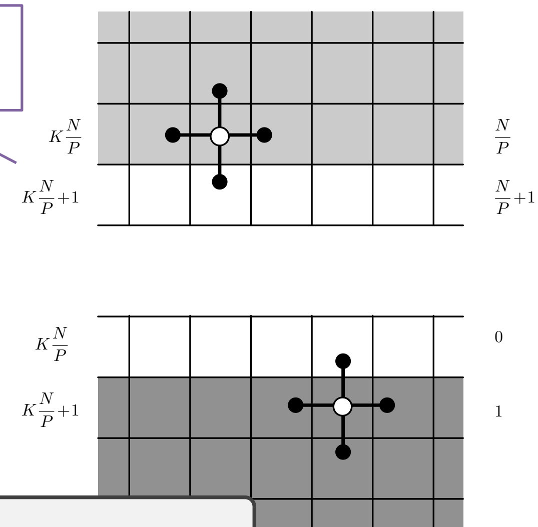
```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition



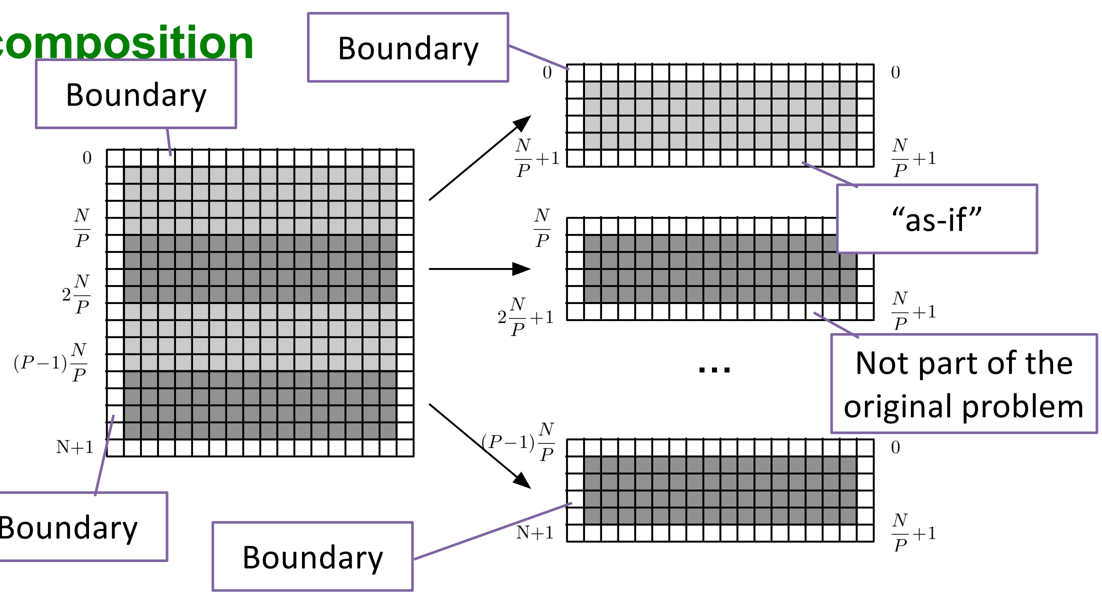
We need an extra row for "as-if"

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

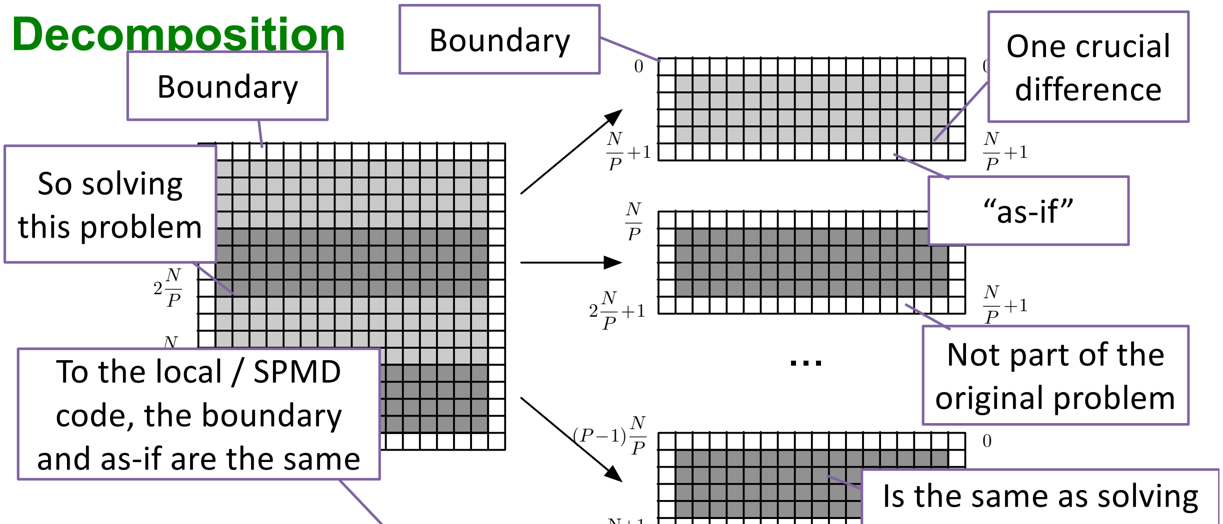



```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```


Decomposition

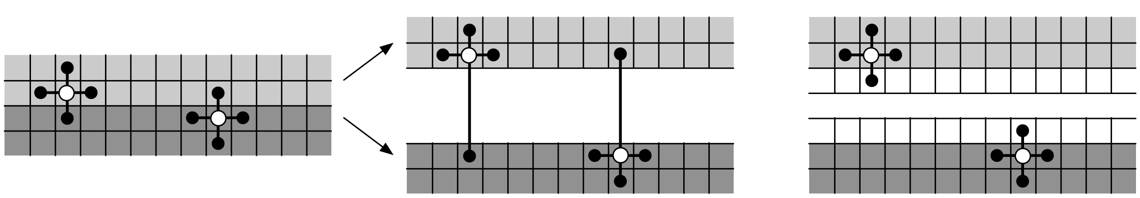


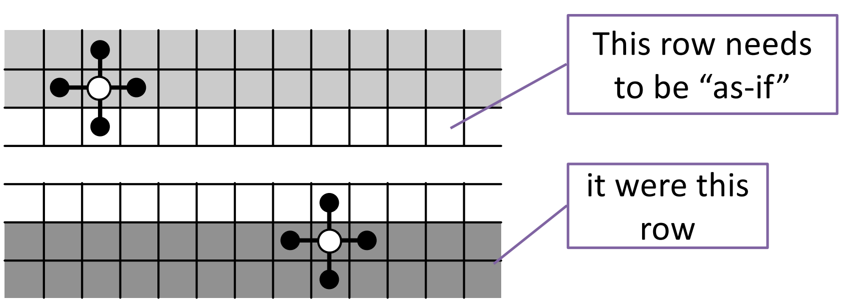
Decomposition



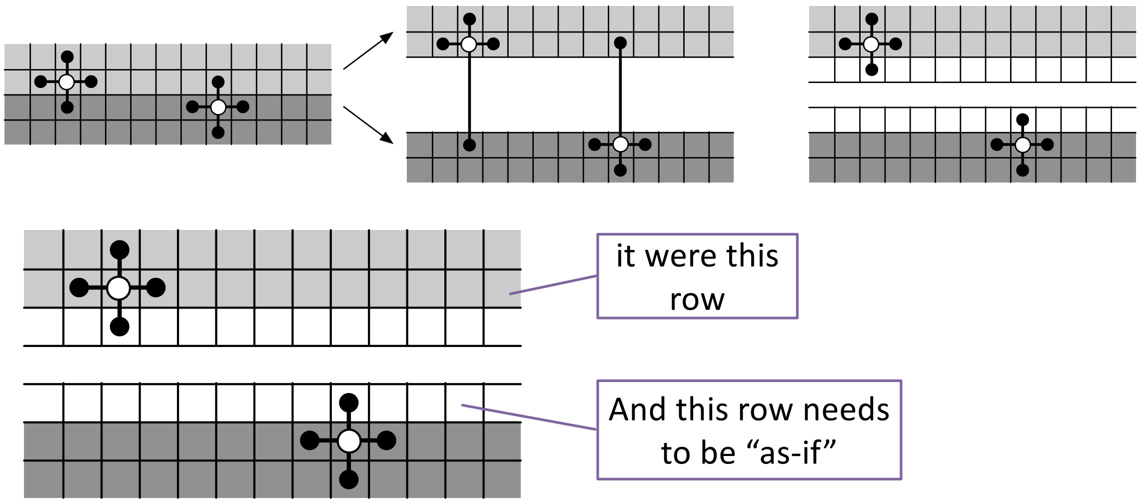

```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```


As-If

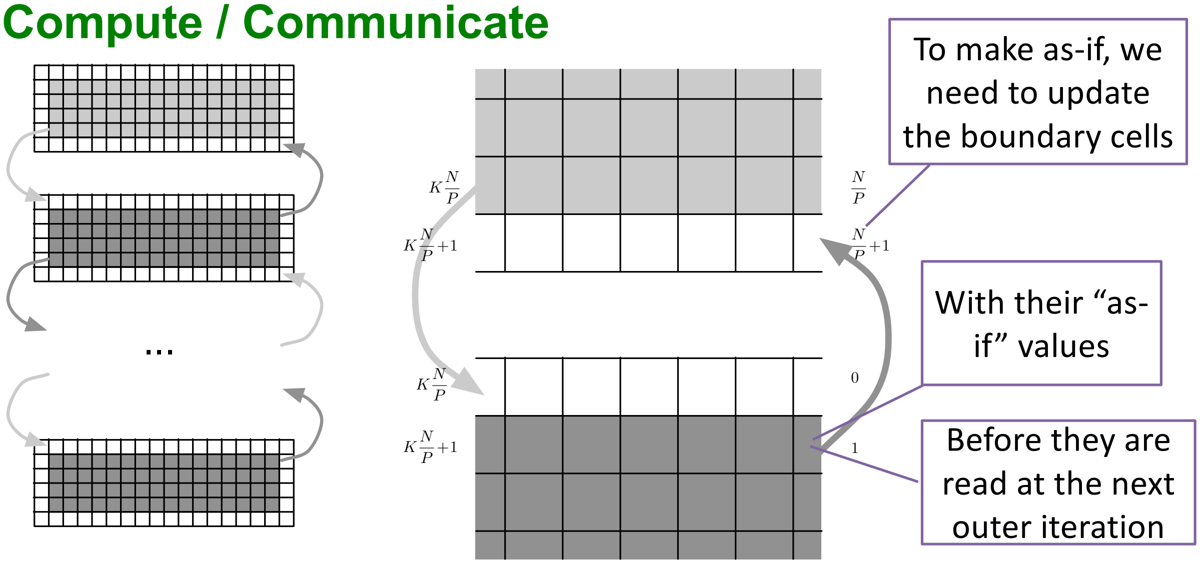




As-If



Compute / Communicate



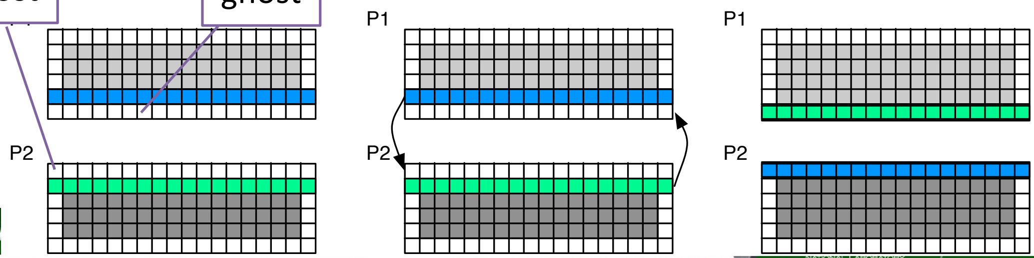
Compute / Communicate

```
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

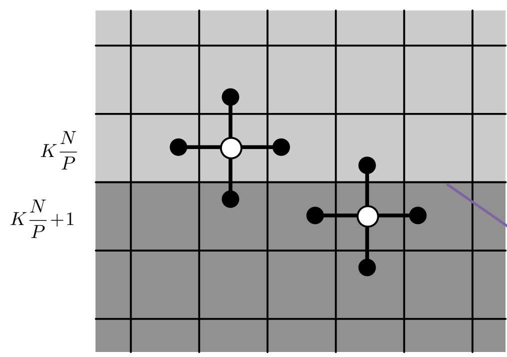
```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
  swap(x,y);
  make_as_if(x); // Communicate ghost cells
}
```

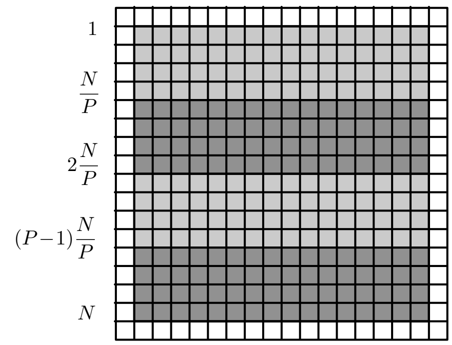
Compute

This is an almost universal pattern

Communicate

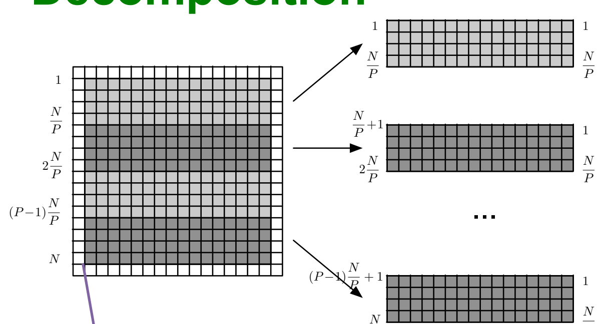
Decomposition





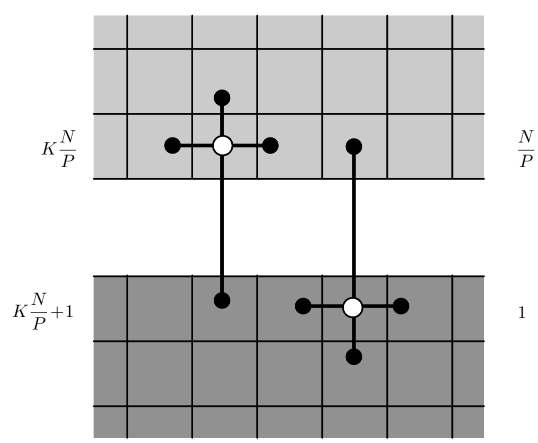
Data dependencies for stencil crosses partition boundary in original problem

Decomposition



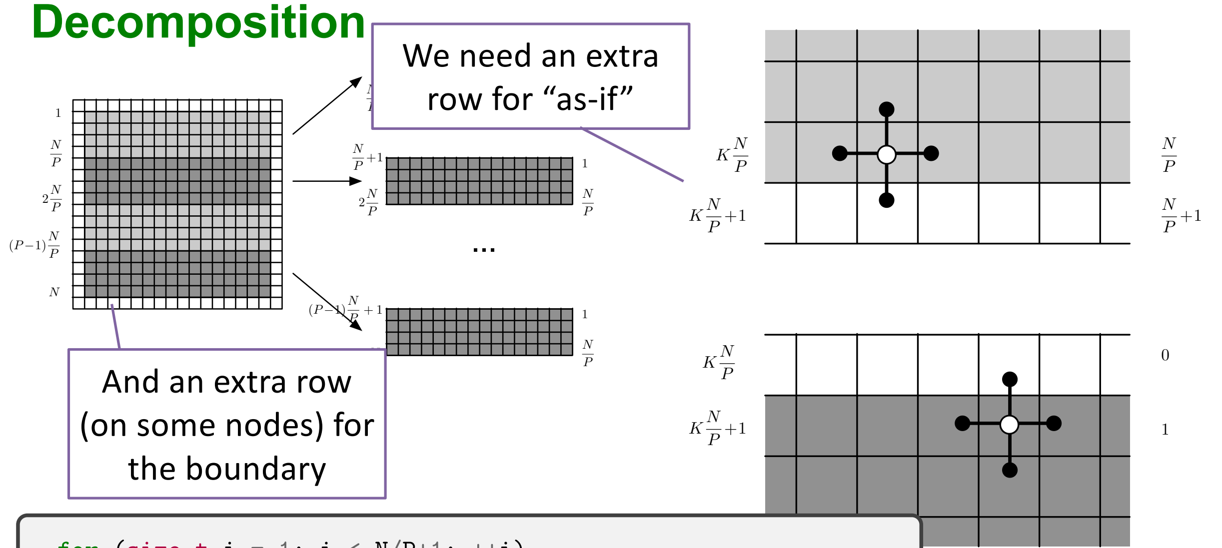
Where did the boundary go

When we partitioned?



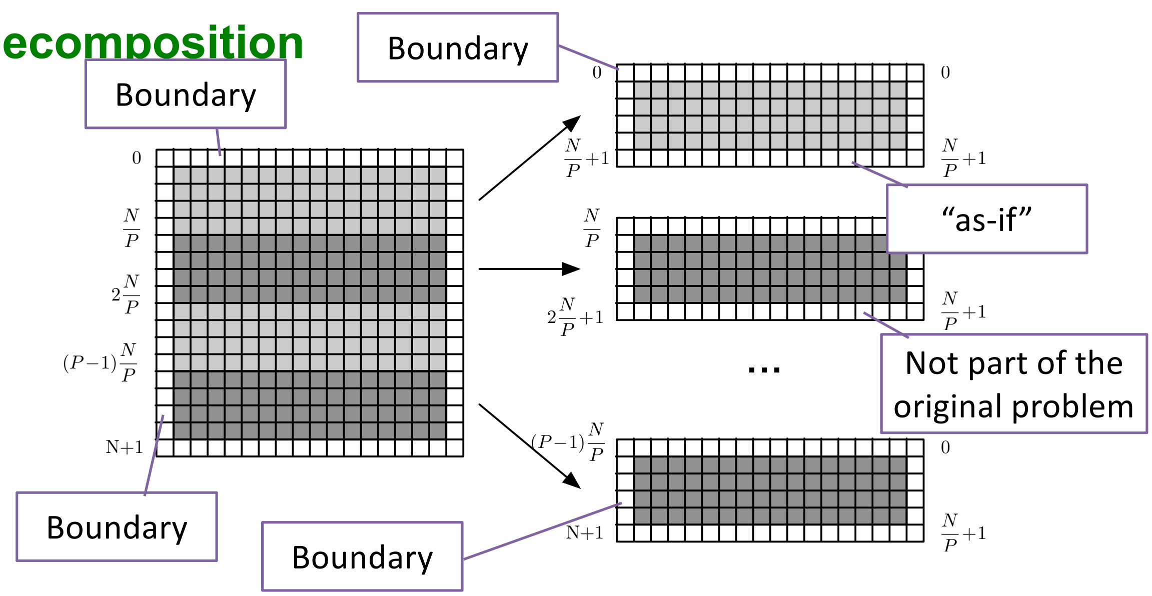

```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```

Decomposition

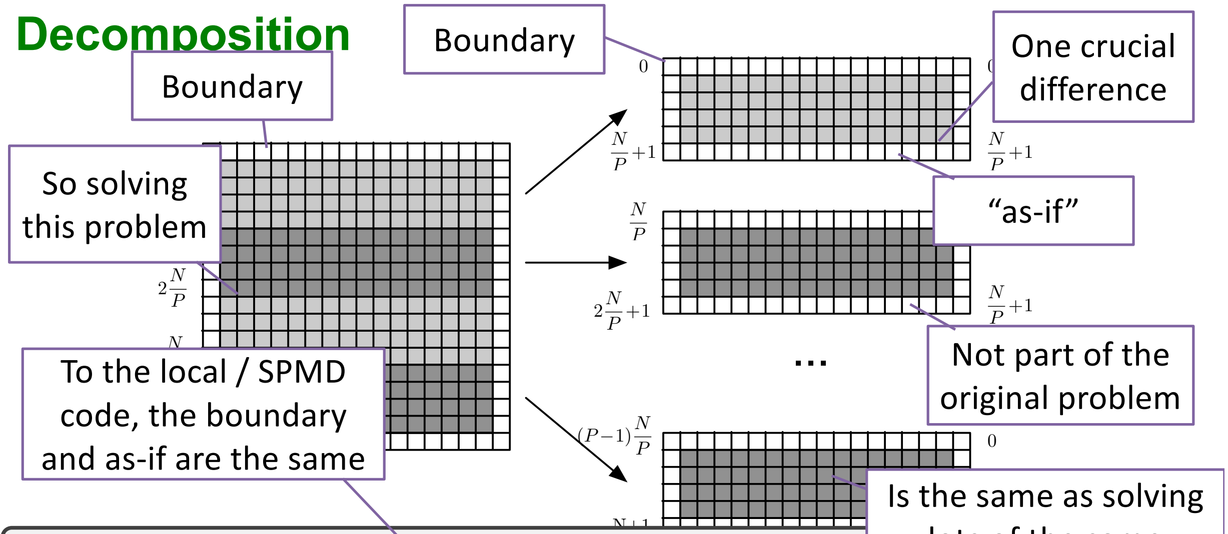



```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```


Decomposition

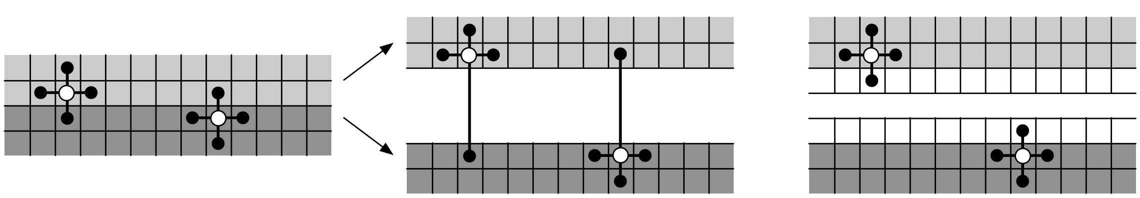


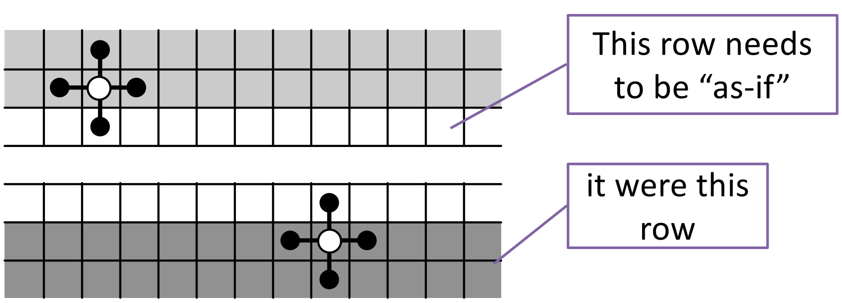
Decomposition



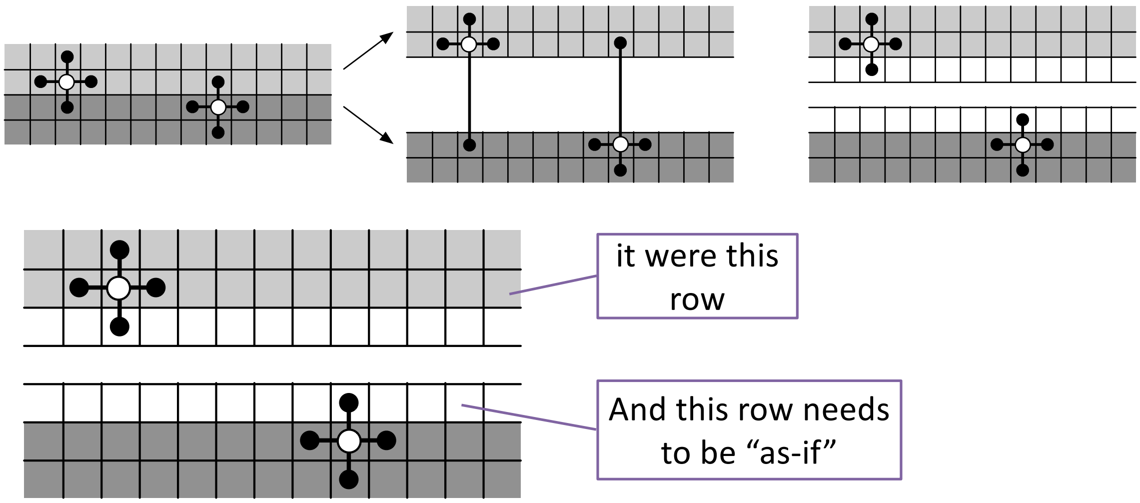

```
for (size_t i = 1; i < N/P+1; ++i)
  for (size_t j = 1; j < N+1; ++j)
    y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
```


As-If

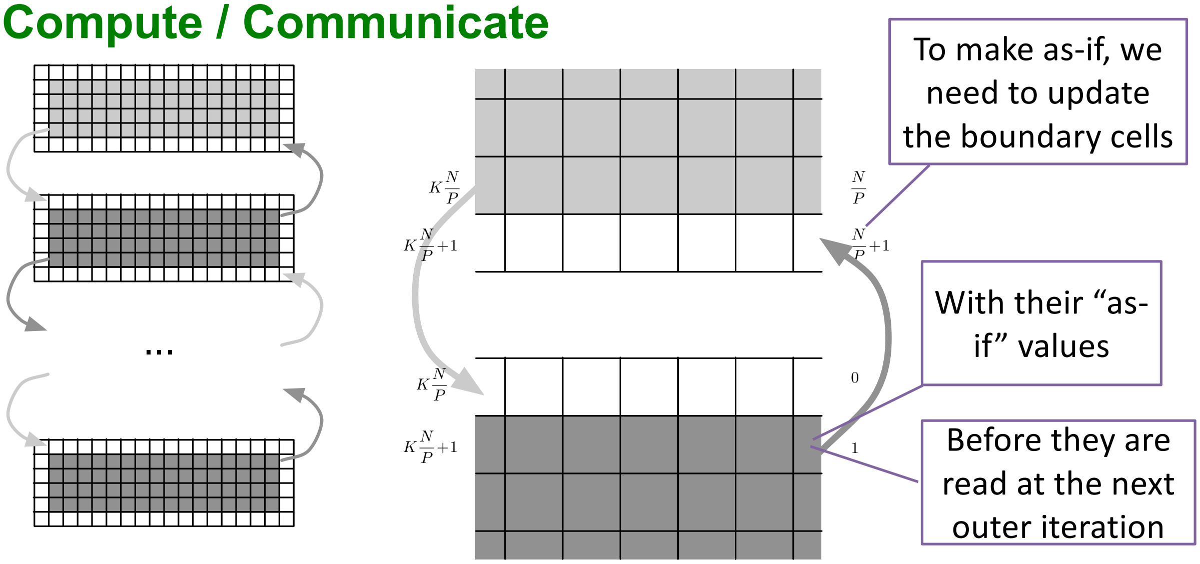




As-If



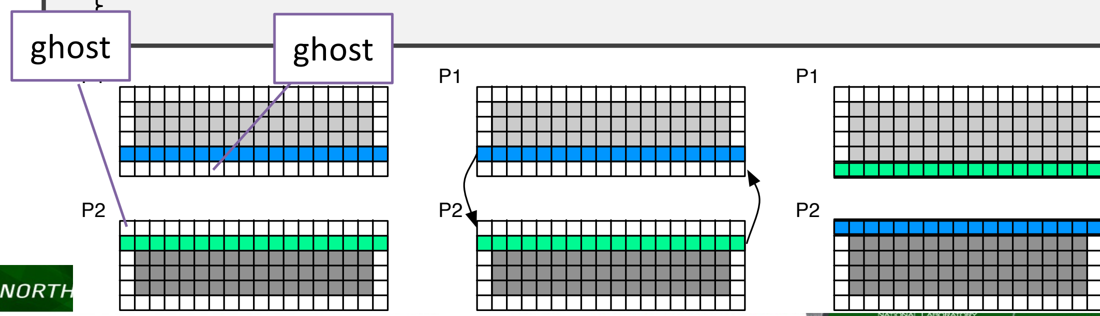
Compute / Communicate



Compute / Communicate

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

```
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 / Communicate

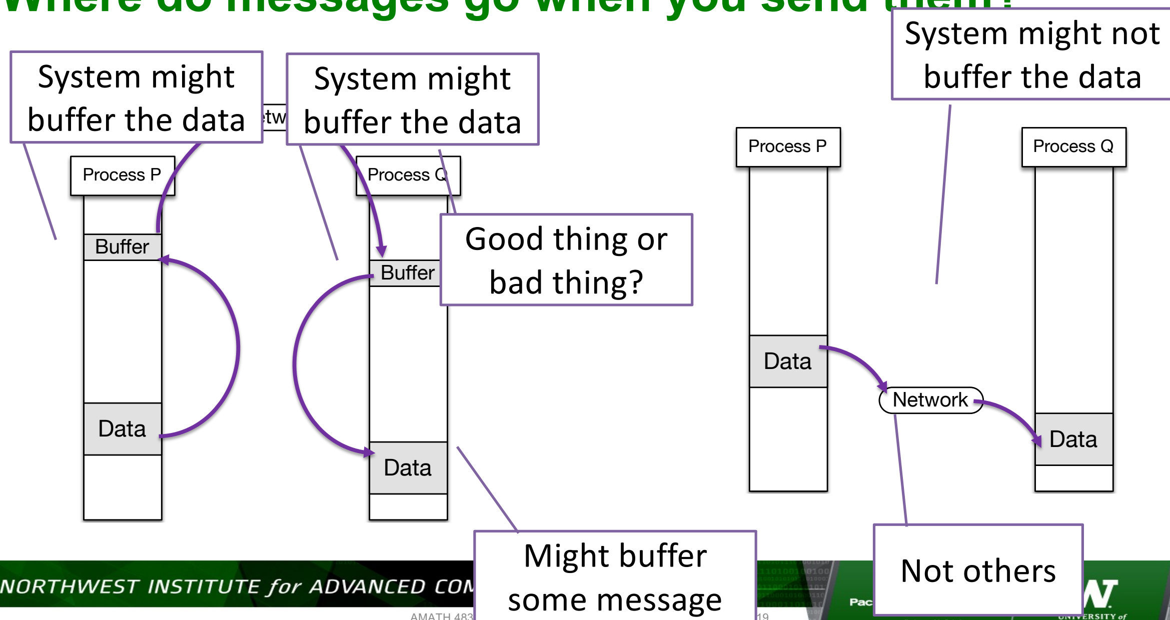
```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
  swap(x,y);
  make_as_if(x); // Communicate ghost cells
}
```

Compute

This is an almost universal pattern

Communicate

Where do messages go when you send them?



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)

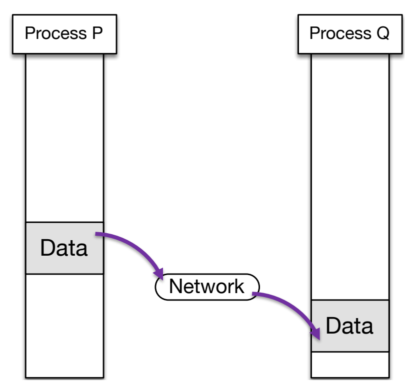
MPI_Recv

```
#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
```

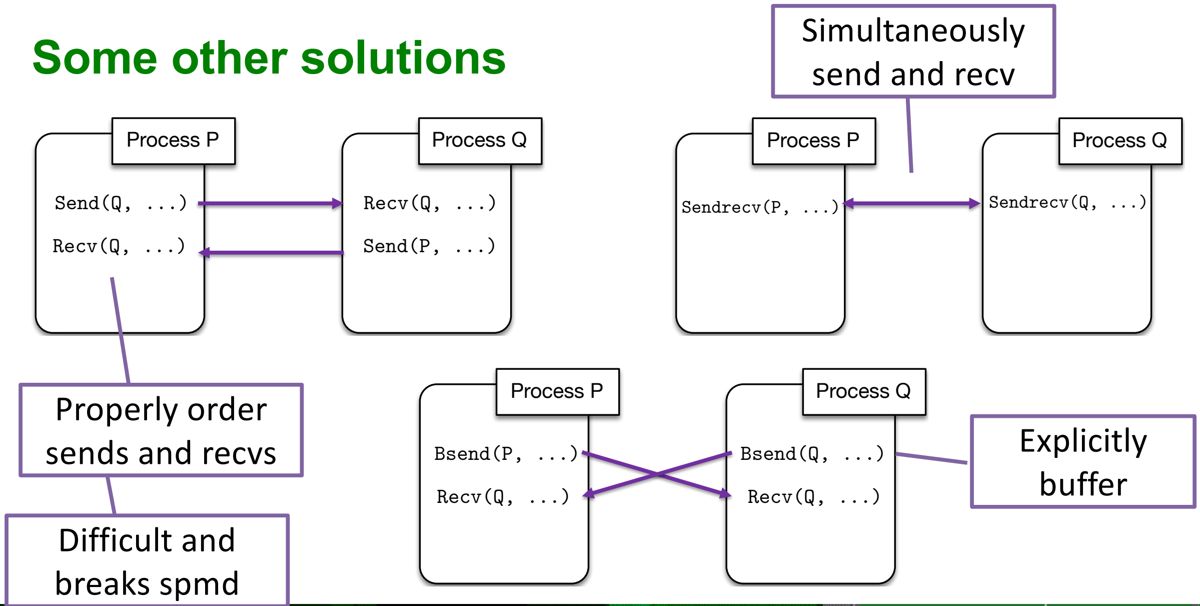
- Blocking receive
- Semantics: Blocks until message is received. On return from call, buffer will have message data

Unbuffered Communication

- Buffering can be avoided
- But we need to make sure it is safe to touch message data
  – Block until it is safe
  – Return before transfer is complete and wait/test later

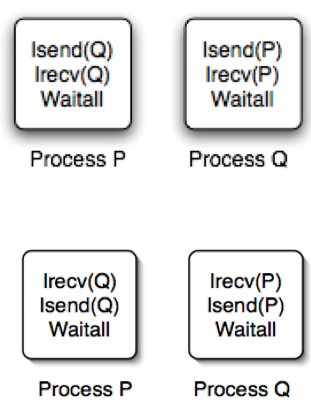


Some other solutions



Non-Blocking Operations

- Non-blocking operations (send and receive) return immediately
- Return "request handles" that can be tested or waited on
- Where progress is made (and where communication happens) is implementation specific



Non-blocking (immediate) operations

Note normal receive

Process P

```
Isend(Q, ...)
Recv(Q, ...)
Waitall
```

Process Q

```
Isend(P, ...)
Recv(Q, ...)
Waitall
```

Returns immediately

What were semantics of Send?

How do we know when it is safe?

Isend returns a request handle

That we wait on until buffer is safe

Non-blocking (immediate) operations

There is also a non-blocking receive

What were semantics of Recv?

Irecv also returns a request handle

Process P

```
Isend(Q, ...)
Irecv(Q, ...)
Waitall
```

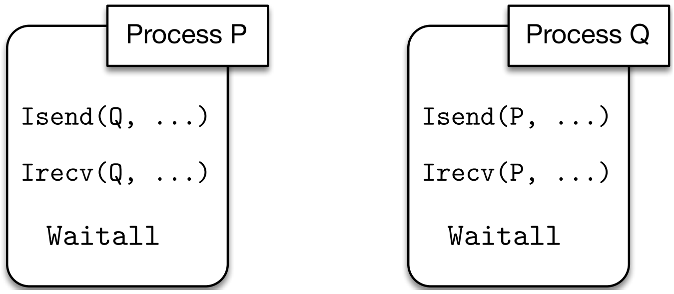
Process Q

```
Isend(P, ...)
Irecv(P, ...)
Waitall
```

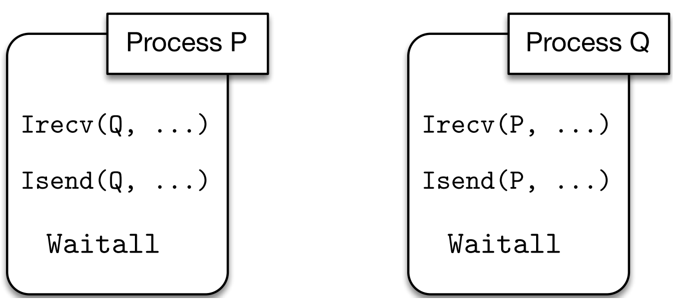
That can be waited on and will return when data are ready

We can wait on all requests together (send and recv)

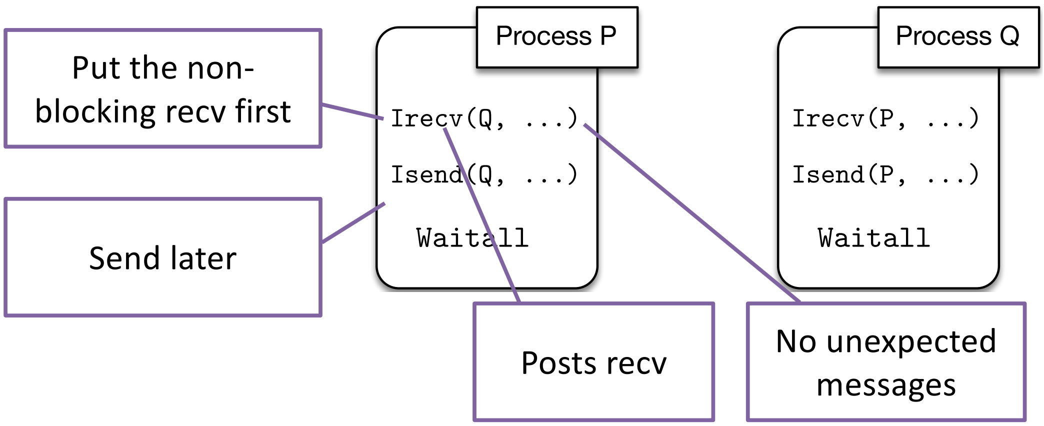
Before



After



After



Bindings for non-blocking receive

```
Request Comm::Isend(const void* buf, int count, const
↪ Datatype& datatype, int dest, int tag) const
```

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

Communication completion: Wait

```
void Request::Wait(Status& status)
void Request::Wait()
```

```
static void Request::Waitall(int count, Request
↪  array_of_requests[], Status array_of_statuses[])
static void Request::Waitall(int count, Request
↪  array_of_requests[])
```

```
static int Request::Waitany(int count, Request
↪  array_of_requests[], Status& status)
static int Request::Waitany(int count, Request
↪  array_of_requests[])
```

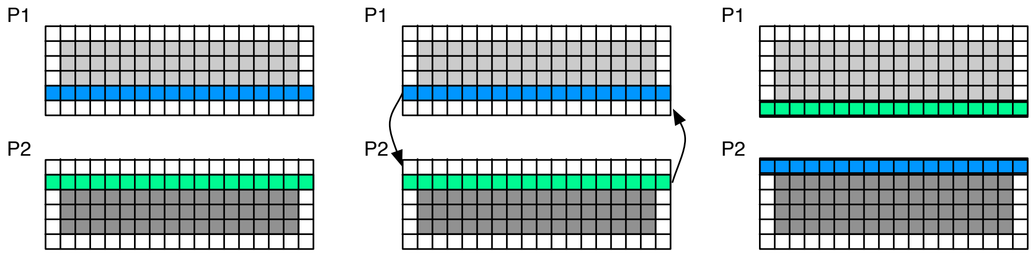

Communication completion: Test

```
bool Request::Test(Status& status)
bool Request::Test()
```

```
static bool Request::Testall(int count, Request
↪  array_of_requests[], Status array_of_statuses[])
static bool Request::Testall(int count, Request
↪  array_of_requests[])
```

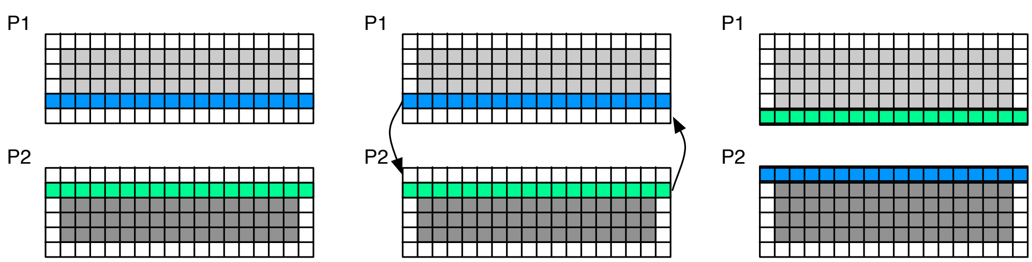
```
static bool Request::Testany(int count, Request
↪  array_of_requests[], int& index, Status& status)
static bool Request::Testany(int count, Request
↪  array_of_requests[], int& index)
```

Updating Ghost Cells




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


Updating Ghost Cells

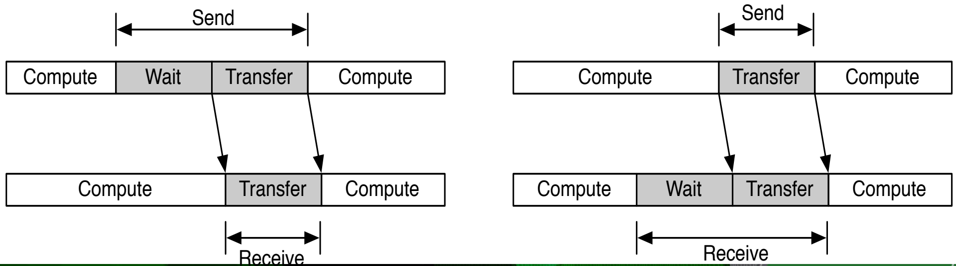



```
MPI_Irecv( ... );    // from lower neighbor
MPI_Irecv( ... );    // from upper neighbor
MPI_Isend( ... );    // to upper neighbor
MPI_Isend( ... );    // to lower neighbor
MPI_Waitall( ... ); // wait for completion
```

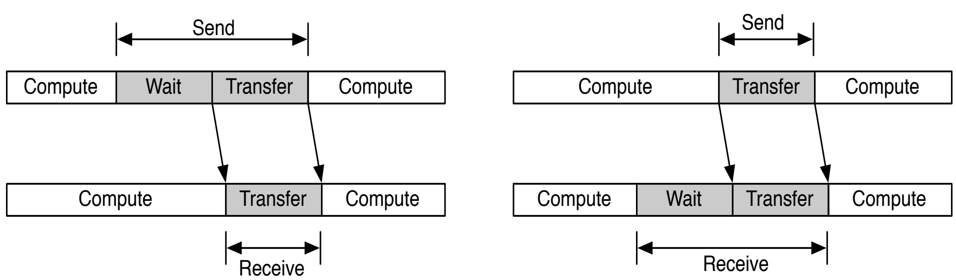
Performance Model

$$T_{communicate} = T_{latency} + T_{bandwidth} = T_L + r_{nic} \cdot Size$$

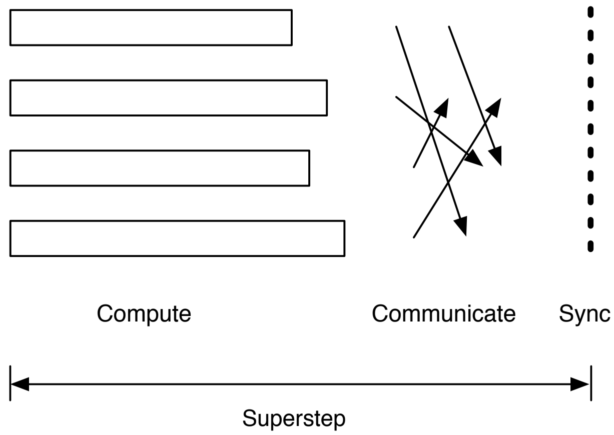
$$Speedup = \frac{T_{seq}}{T_{parallel}} = \frac{T_{seq}}{T_{compute} + T_{bandwidth} + T_{latency}}$$



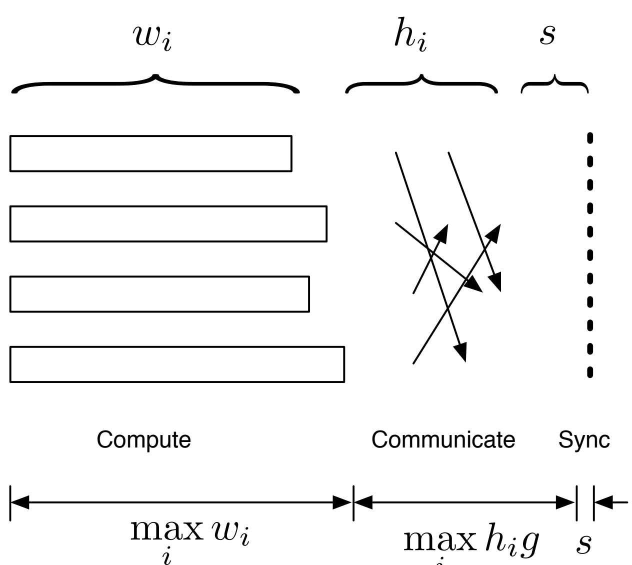
Synchronous vs Asynchronous



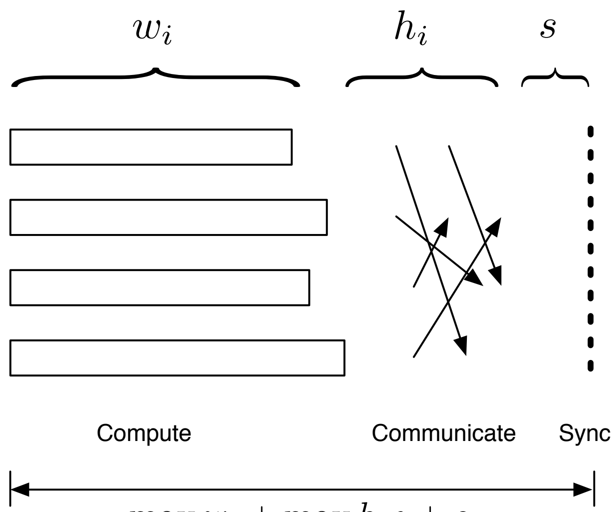
BSP Model



BSP Model

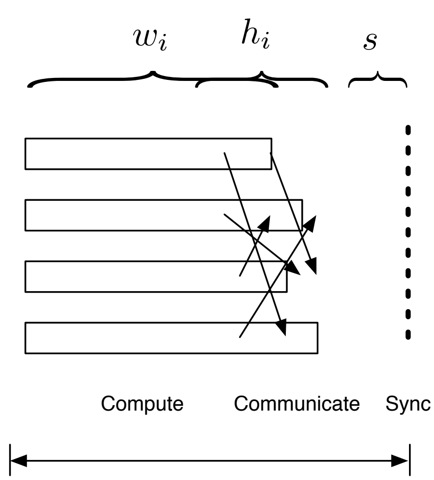


BSP Model



$$\max_i w_i + \max_i h_i g + s$$

BSP with asynchronous communication



$$\max_i (w_i, h_i g) + s$$

LogP

- Parameters (measured in processor cycles)
  - L - upper bound on latency for a single message
  - o - overhead to transmit or receive a message
  - g - minimum gap between consecutive messages
  - P - number of processors
- Finite capacity constraint
  - At most $\lceil L/g \rceil$ messages can be in transit from or to any given processor at one time
  - Processors that attempt to exceed this limit stall until the message can be sent

LogP cont.

- More coarse grained than PRAM
  - PRAM = LogP with (L = 0, o = 0, g = 0)
- More fine grained than BSP
- Allows more precise scheduling of communication
  - Reading a remote memory location
    - BSP - next superstep, L cycles
    - LogP - 2L + 4o cycles
- No special synchronization hardware
- Parameters can be experimentally determined for a given machine/architecture
- No special treatment for long messages

LogGP

- Actual machines have special hardware to handle long messages
  - DMA channel to network interface
- G parameter captures the bandwidth achieved for long messages
  - G = Gap per byte for long messages defined as the time per byte
  - 1/G = per processor bandwidth for long messages
- Implicit w parameter to define size of small messages
- Sending a k-byte message between two processors
  - LogP
    - $\lceil k/w \rceil$ messages
    - $o + \lceil (k-1)/w \rceil * \max\{g,o\} + L + o$ cycles.
  - LogGP
    - $o + (k-1)G + L + o$

Applications: Reduce

- PRAM
  - EREW/CREW
    - Binary tree - O(log n)
  - CRCW
    - Arbitrary succeed
      - Binary tree - O(log n)
    - Arbitrary operation
      - All procs write one memory location - O(1)
- BSP
  - O(log n) supersteps
  - L = time to read two memory locations and write one

Applications: Reduce

- LogP
  - Linear reduce
    - o for each processor to send it's value to the root
    - (P-1)o + L for the root to receive them
    - o + (P-1)*max{g,o} + L
  - Binary tree
    - o for each leaf proc to send it's value to it's parent
    - o + max{g,o} + L + o for each non-leaf processor to receive values from each of its children and send the result to its parent
    - o + (log P)(o + max{g,o} + L + o)

Thank You!

Creative Commons BY-NC-SA 4.0 License