AMATH 483/583
High Performance Scientific Computing

Lecture 19:
Advanced Message Passing, Collectives, Performance Models, Eigenfaces

Xu Tony Liu, PhD

Paul G. Allen School of Computer Science & Engineering

University of Washington

Seattle, WA

Administrative

- Fill out course evaluations!
- Final assignment is out, due Friday midnight June 10^{th}
- No physical office hours at LEW 315
- Zoom office hours instead
- Link will be posted through announcement

Top500 As of May 30th, 2022 (top500.org)

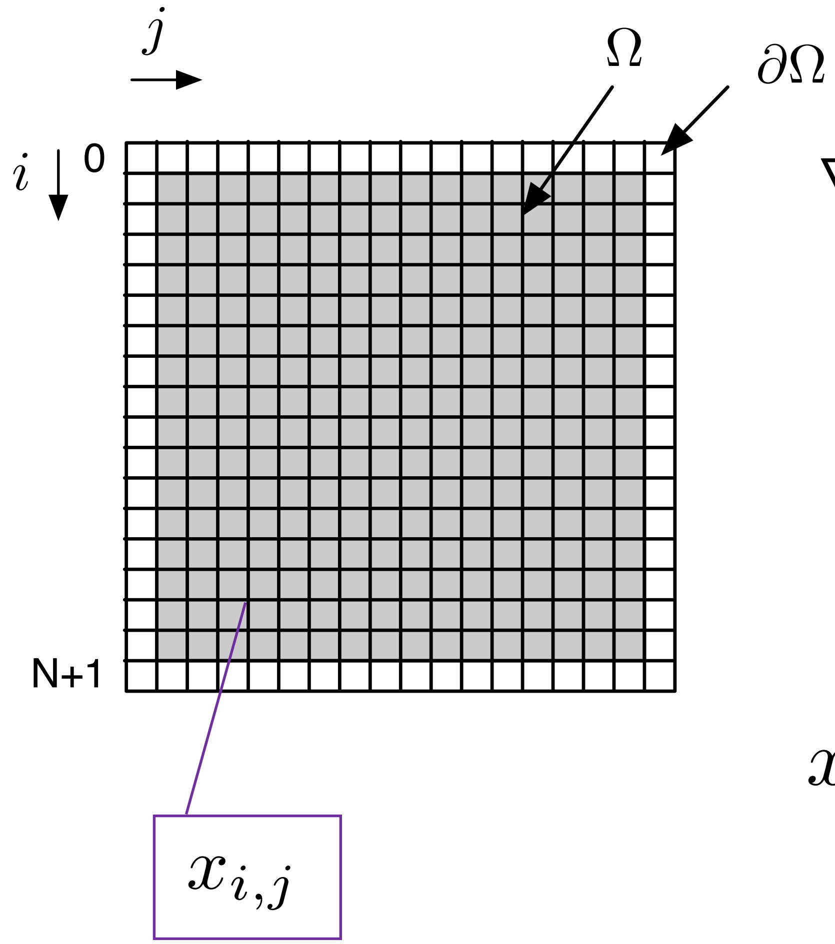
- ORNL's Frontier First to Break the Exaflop Ceiling!
  - HPE Cray EX architecture
  - 1.102 Exaflop/s
  - 8,730,112 total AMD EPYC 64C 2GHz processors
  - AMD Instinct™ 250X accelerators
  - Slingshot-11 interconnect

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

Outline

- Previously
  - Laplace’s equation on a regular grid
- Non-blocking operations
- Collectives
- Performance models
- Eigenfaces

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 solution

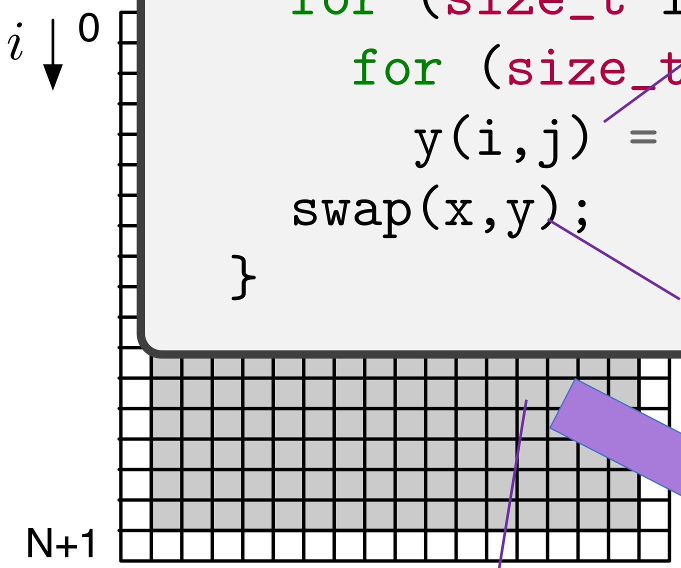
```
while (! converged())
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-
  swap(x,y);
}
```

Approximation at iteration k+1

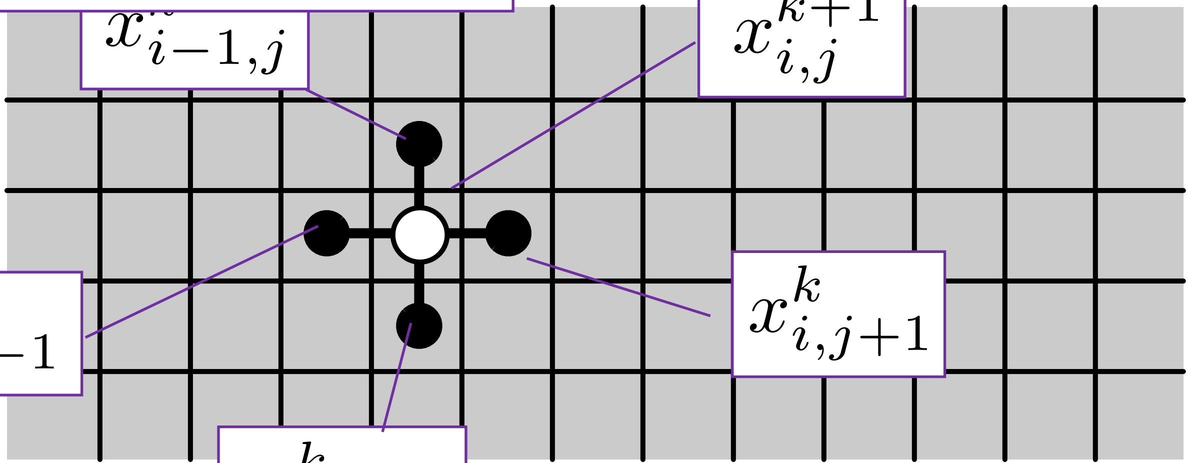
Average of approximation at iteration k

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

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



$x_{i,j}$



$x^{k}_{i-1,j}$

$x^{k+1}_{i,j}$

$x^{k}_{i,j-1}$

$x^{k}_{i,j+1}$

$x^{k}_{i+1,j}$

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;
};
```

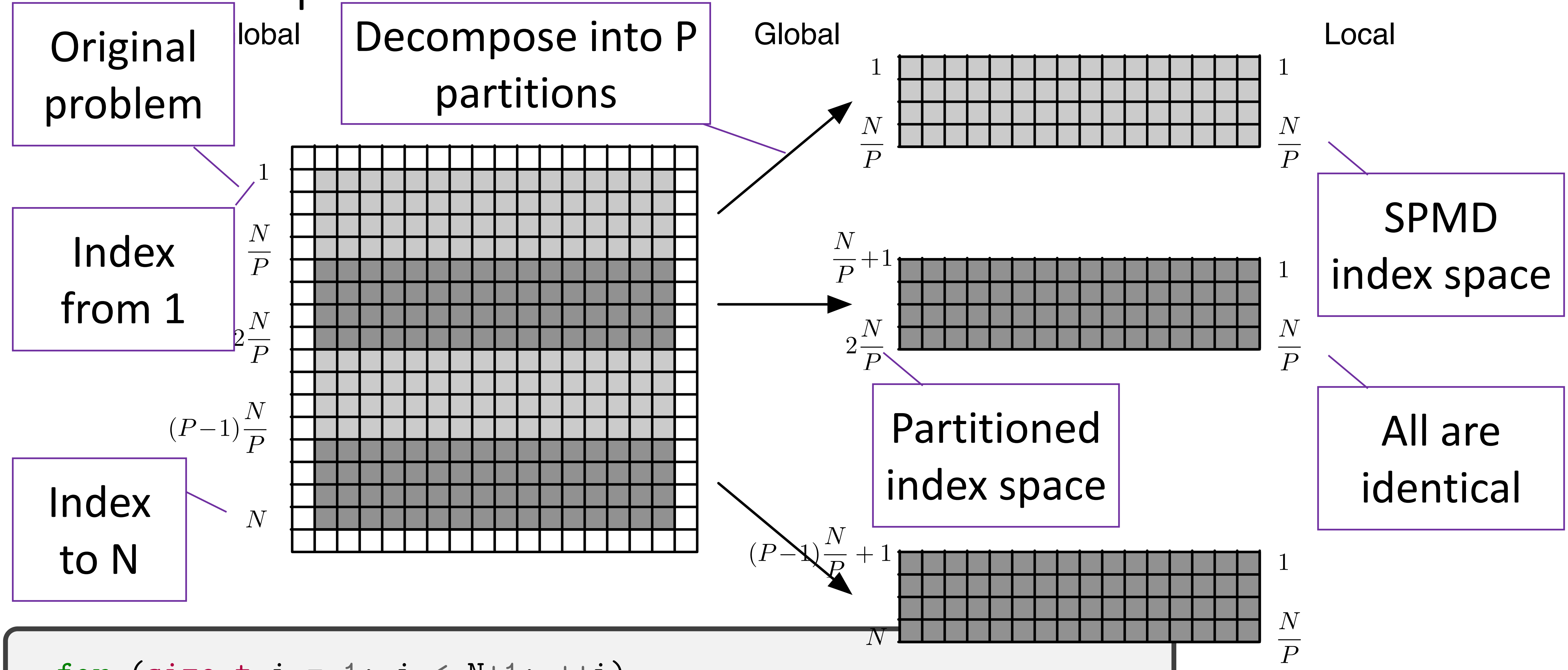
Grid is a 2D array

Constructor

Accessor

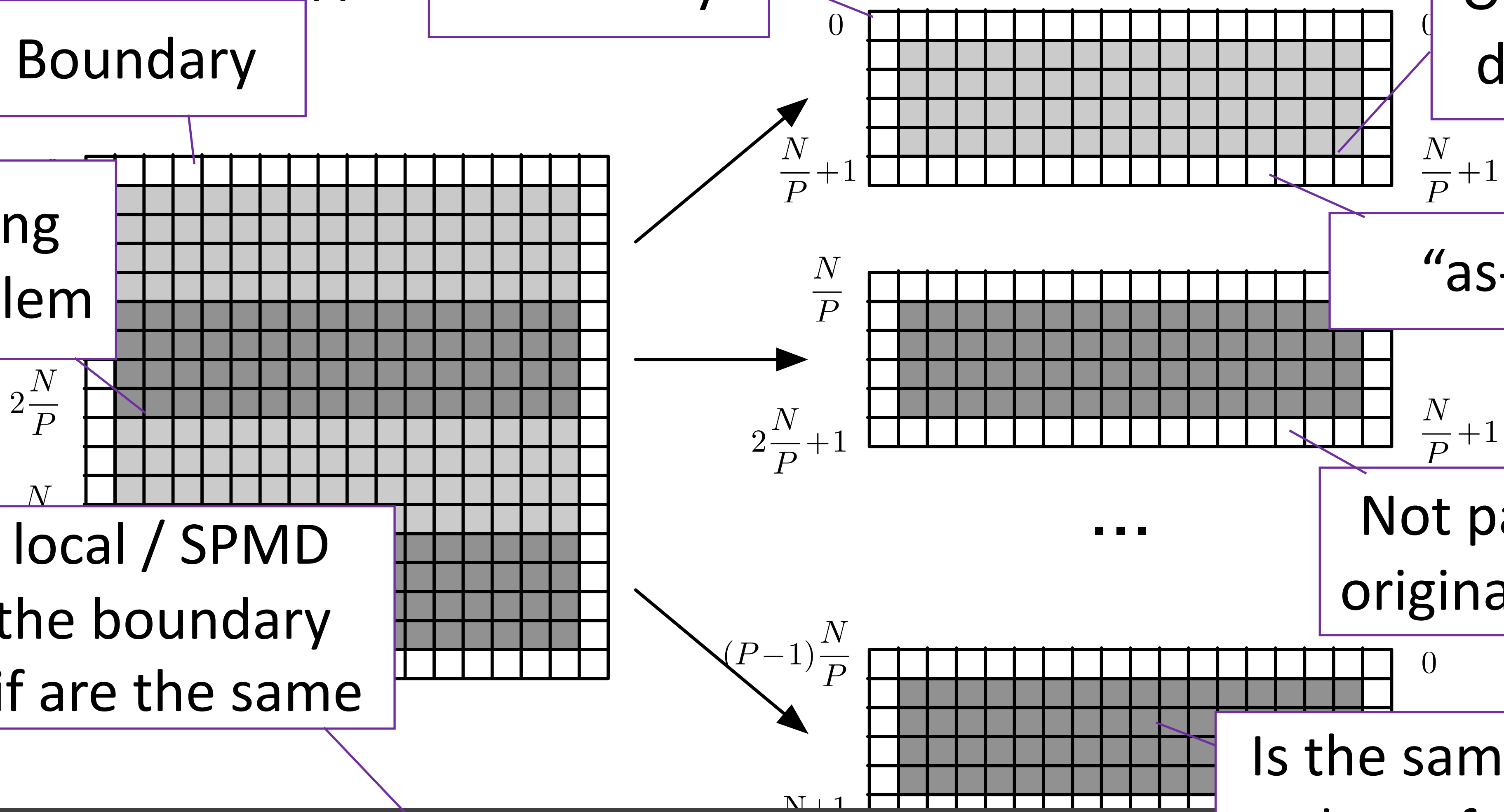
Storage

Decomposition




```
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



Boundary

Boundary

So solving this problem

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

One crucial difference

"as-if"

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

Always write y

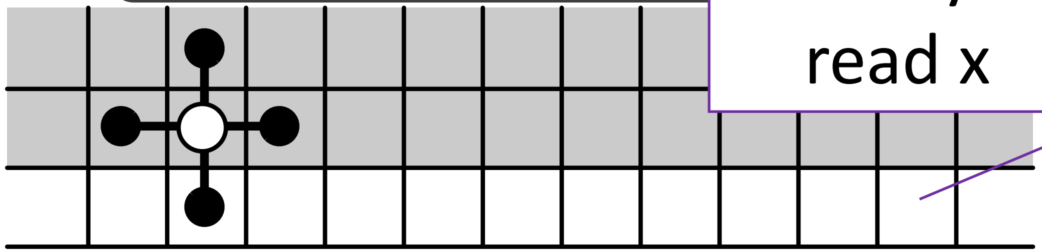
```
  (! converged()) {
    for (size_t i = 1; i < N+1; ++i)
      for (size_t j = 1; j < N+1; ++j)
        y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
    swap(x,y);
  }
```

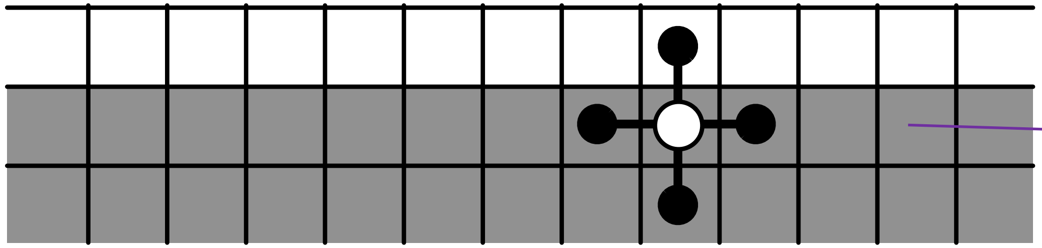
This is the entire program

Always read x

Not changed during an iteration

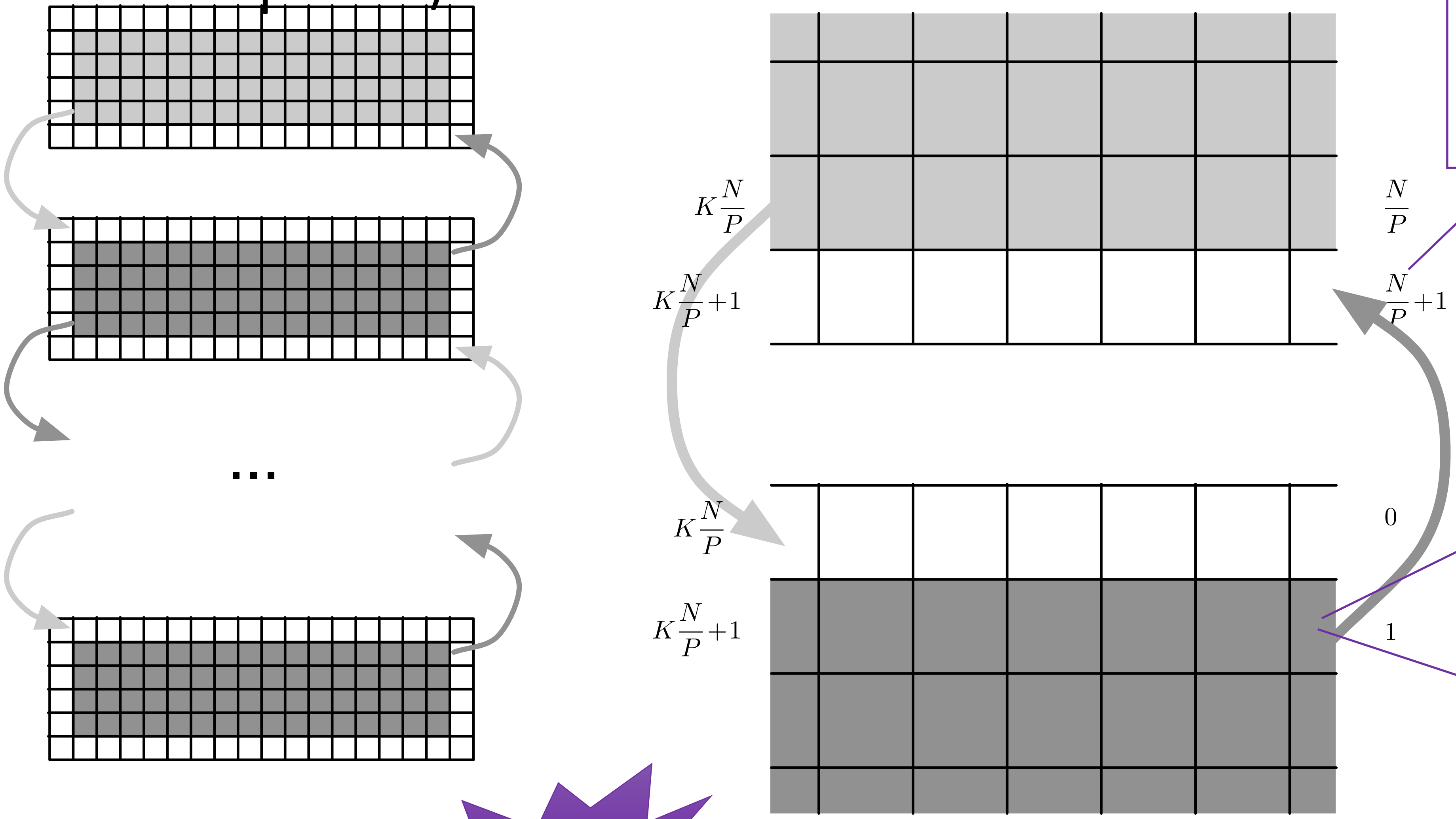
Rows need to be as-if only during iteration





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

Compute / Communicate



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

With their "as-if" values

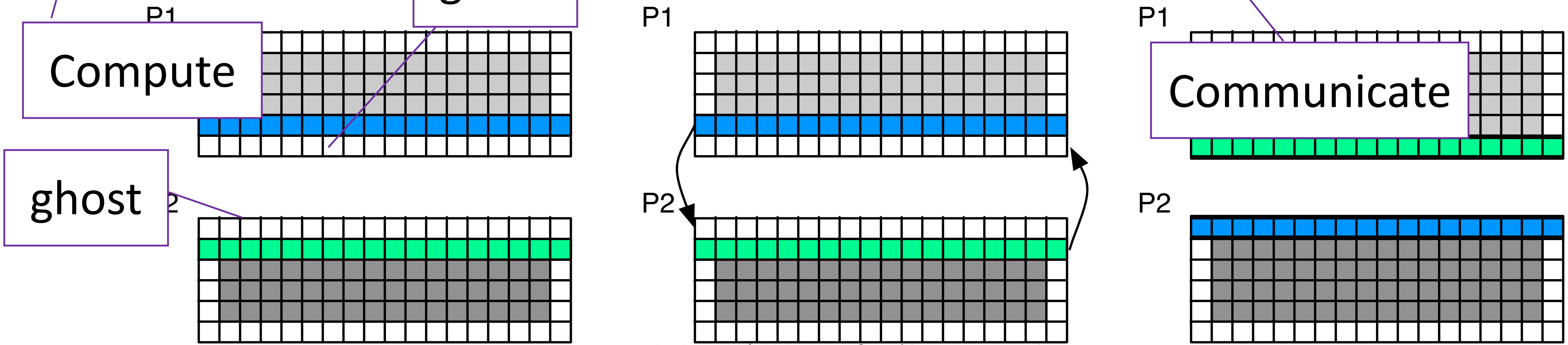
Before they are read at the next outer iteration

Very Important Slide!!

Compute / Communicate

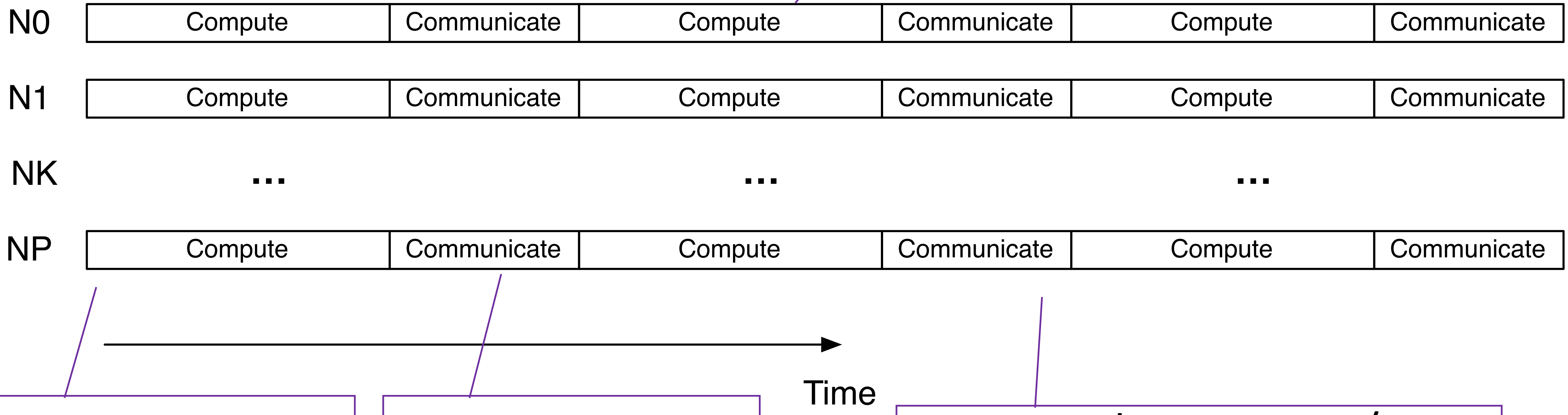
```
while (! converged()) {
  for (size_t i = 1; i < N+1; ++i)
    for (size_t j = 1; j < N+1; ++j)
      y(i,j) = (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))/4.0;
  swap(x,y);
  make_as_if(x); // Communicate ghost cells
}
```

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



Compute / Communicate

"Bulk Synchronous Parallel" (BSP)

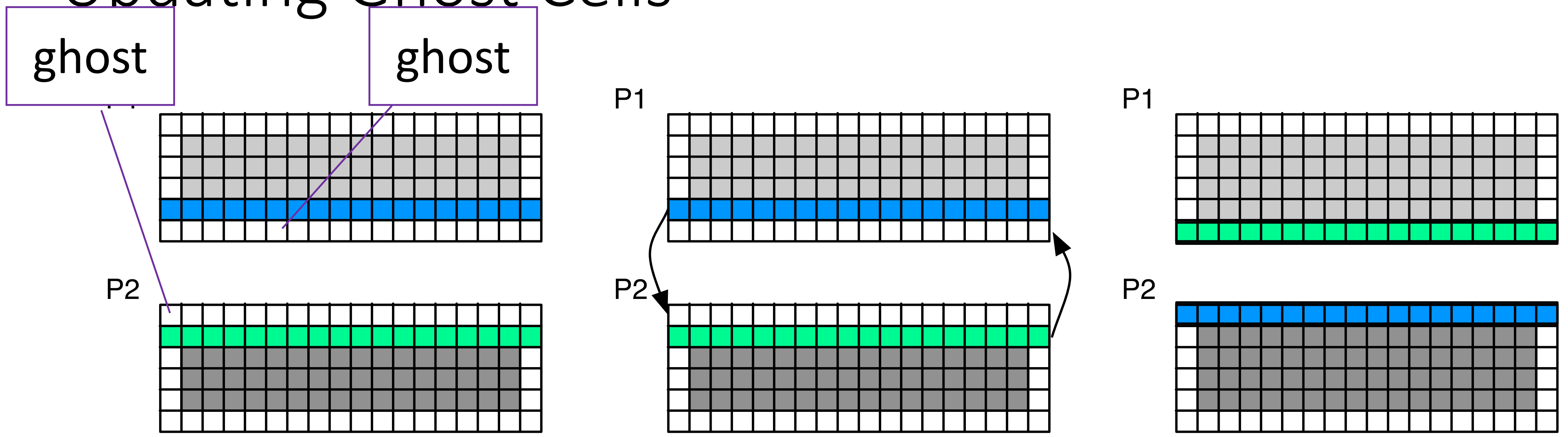


This is an almost universal pattern

Processors are still only loosely coupled

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

Updating Ghost Cells

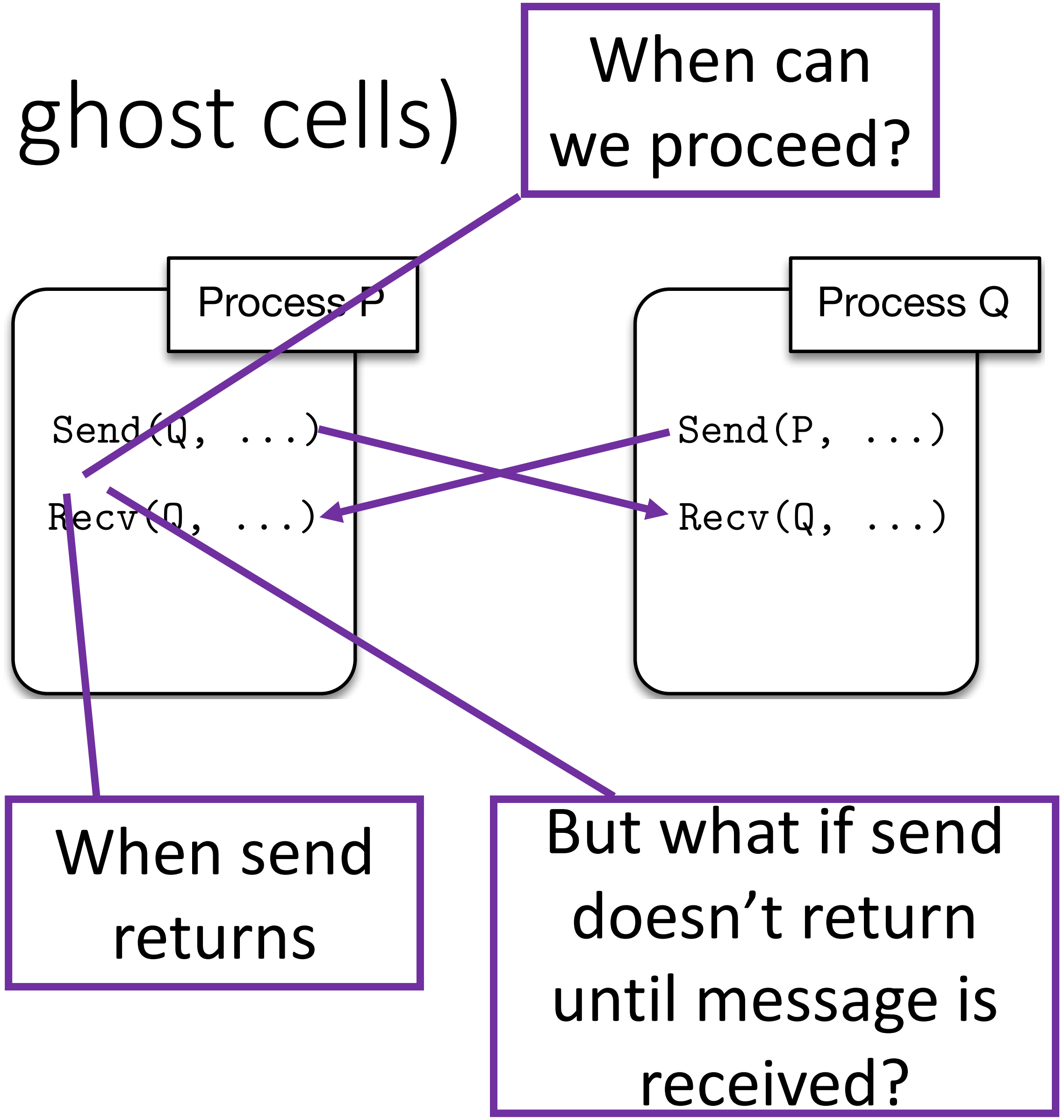



```
MPI_Send( ... );   // to upper neighbor
MPI_Send( ... );   // to lower neighbor
MPI_Recv( ... );   // from lower neighbor
MPI_Recv( ... );   // from upper neighbor
```

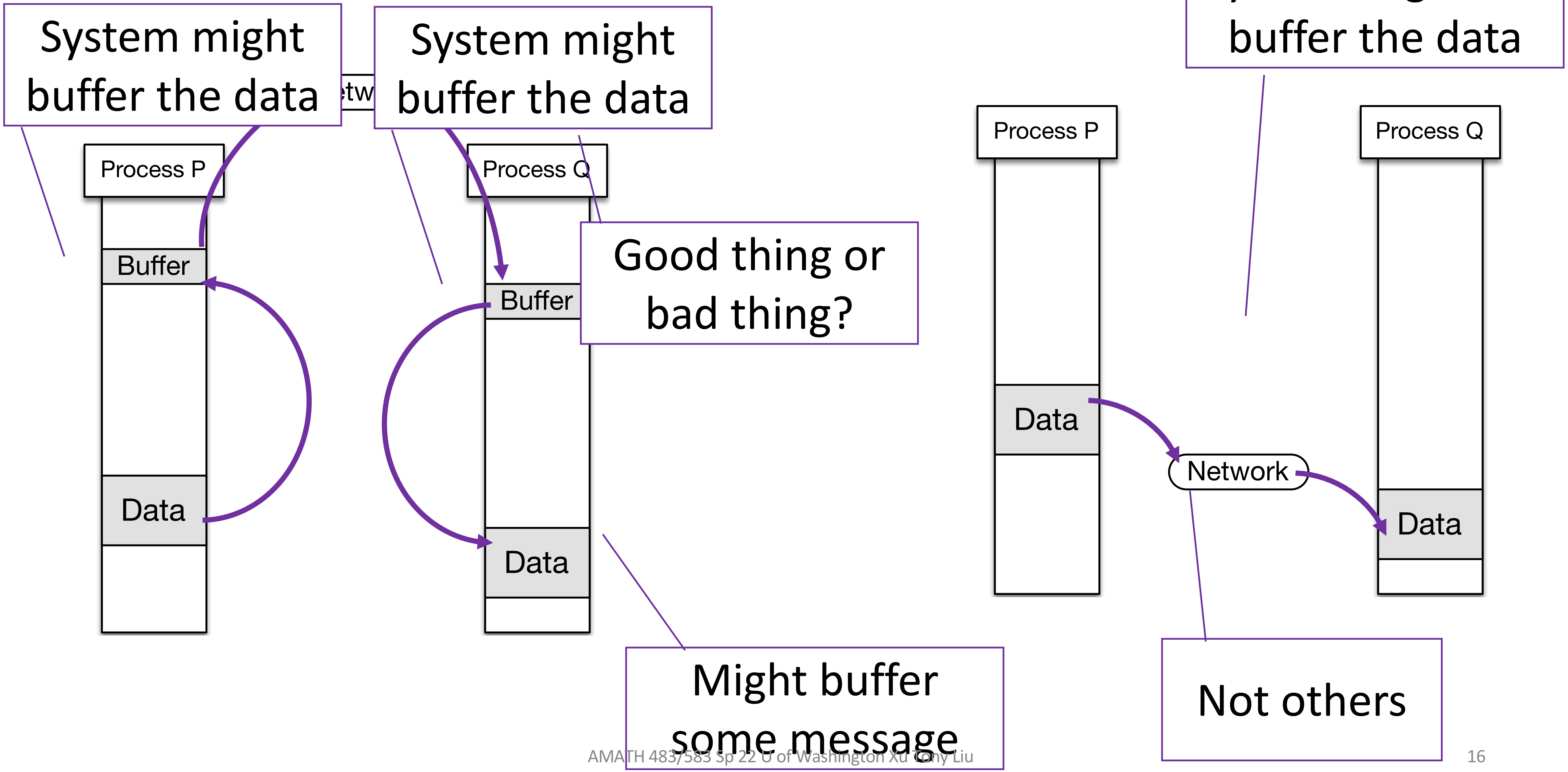
Works?

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 t ?



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

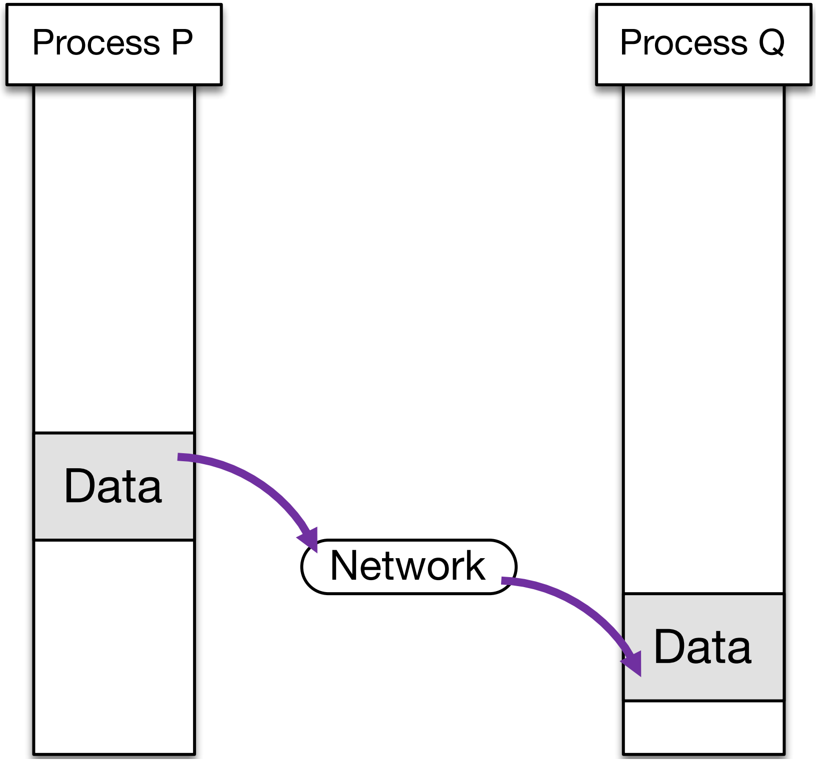
```
#include <mpi.h>
void Comm::Recv(void* buf, int count, const Datatype& datatype,
↪  int source, int tag, Status& status) const

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

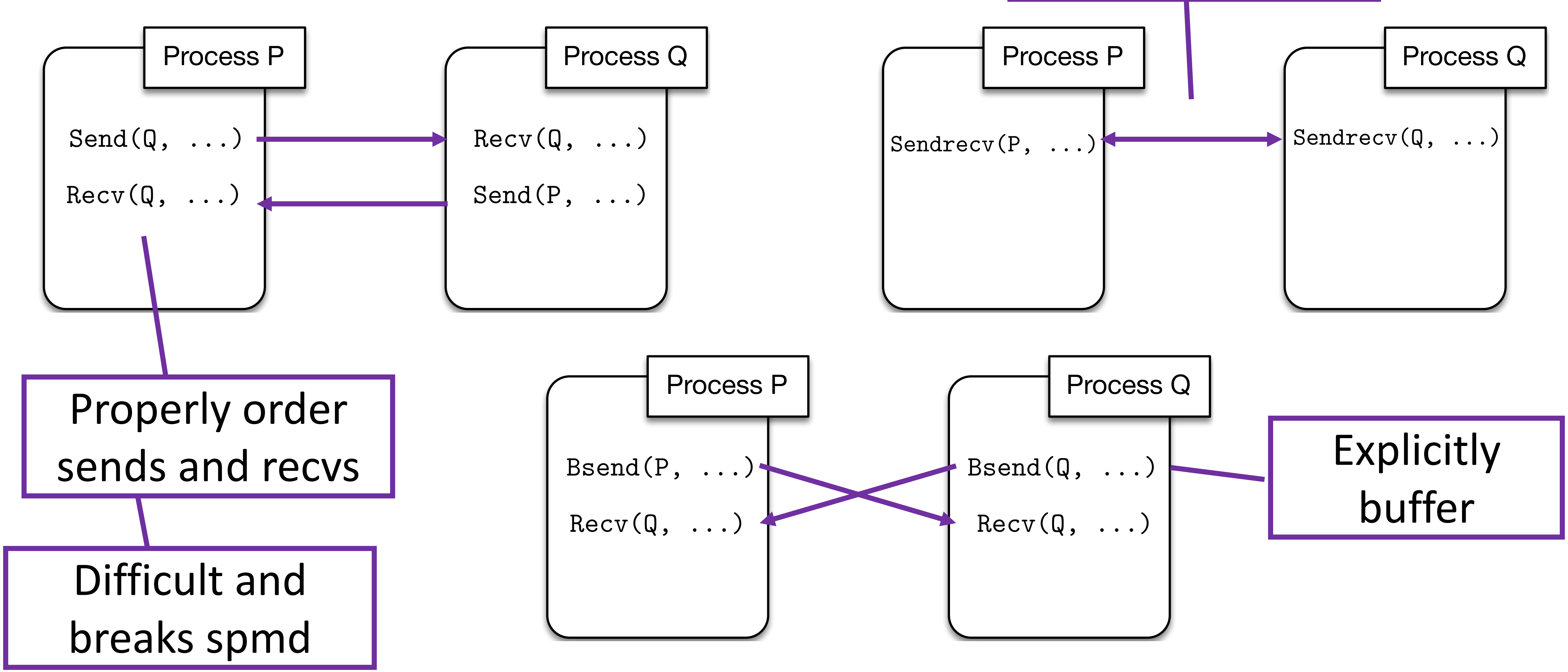
- Blocking receive
- Semantics: Blocks until message is received. On return from call, buffer will have message data

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

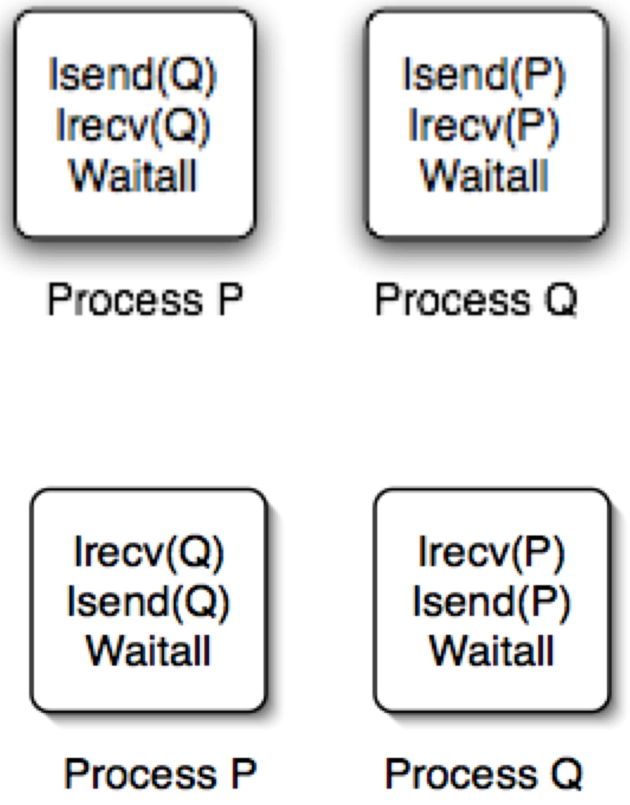


Some other solutions

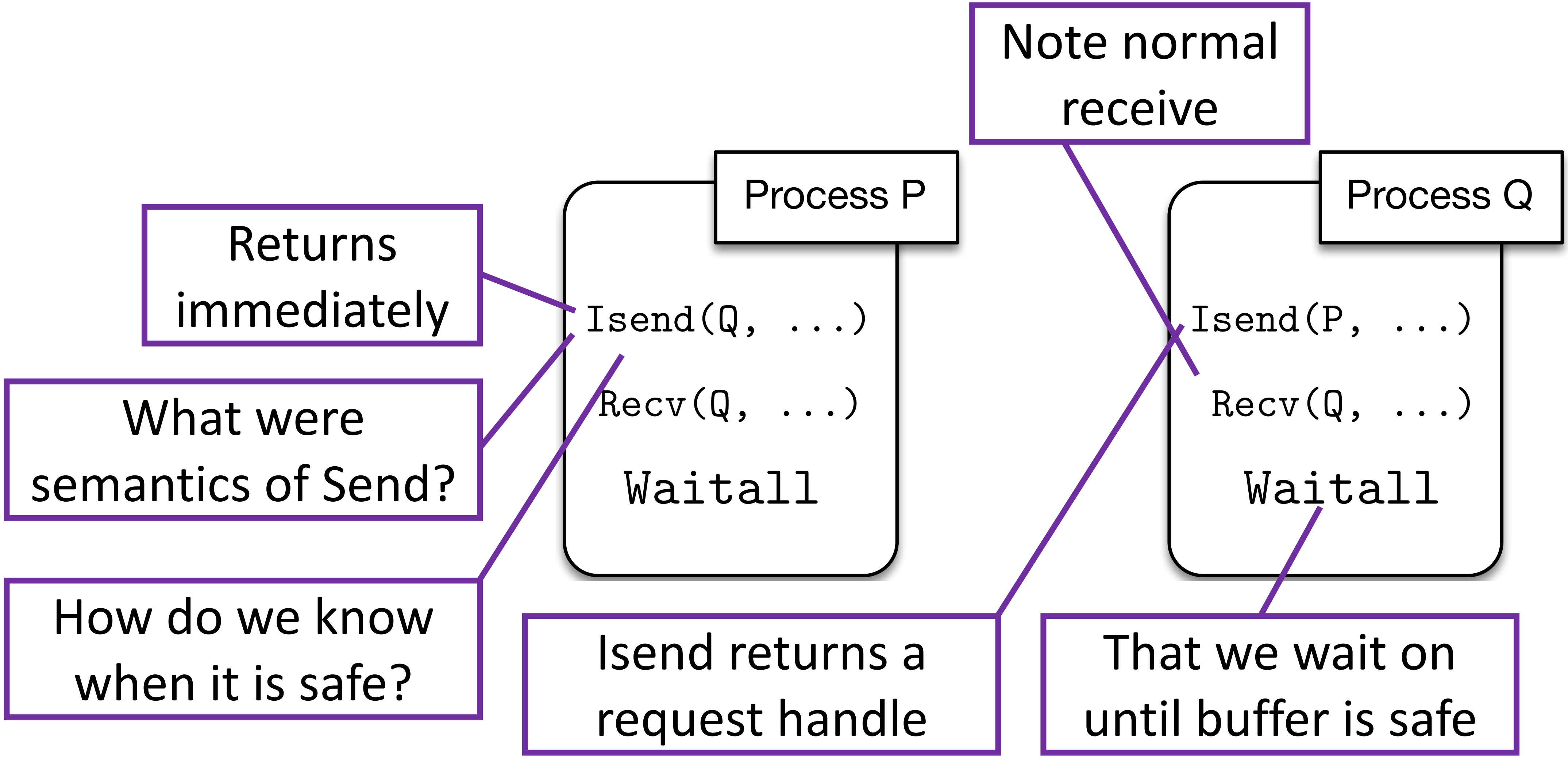


Non-Blocking Operations

- 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

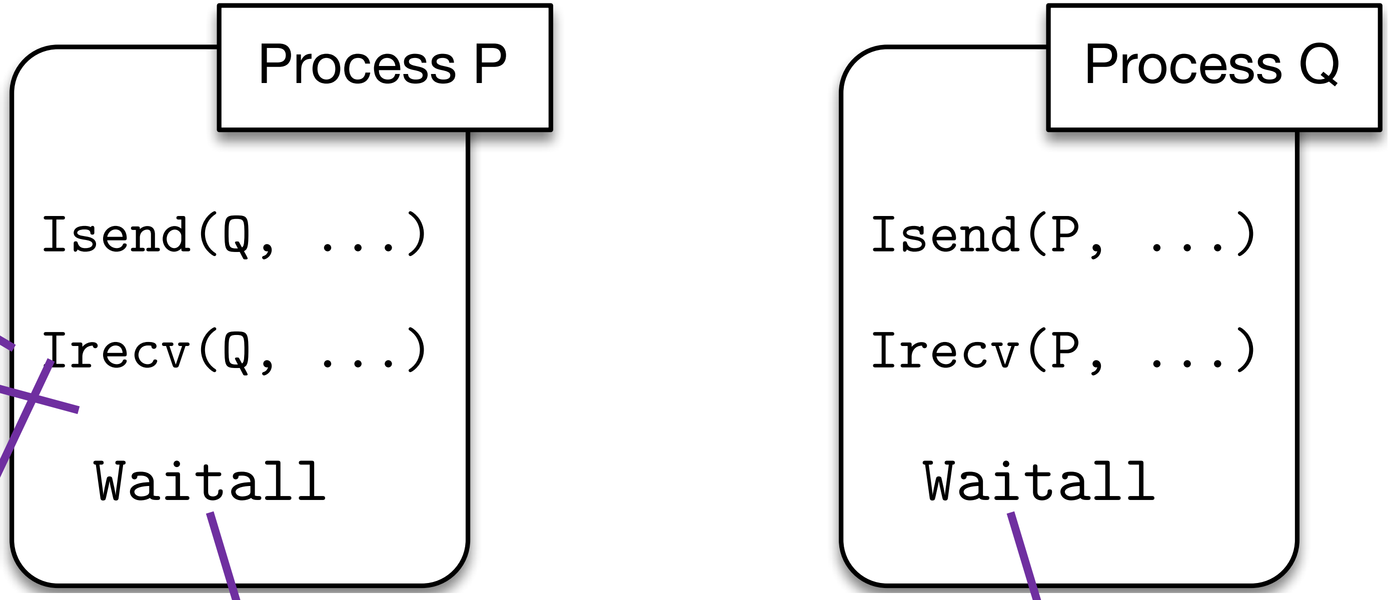


Non-blocking (immediate) operations

There is also a non-blocking receive

What were semantics of Recv?

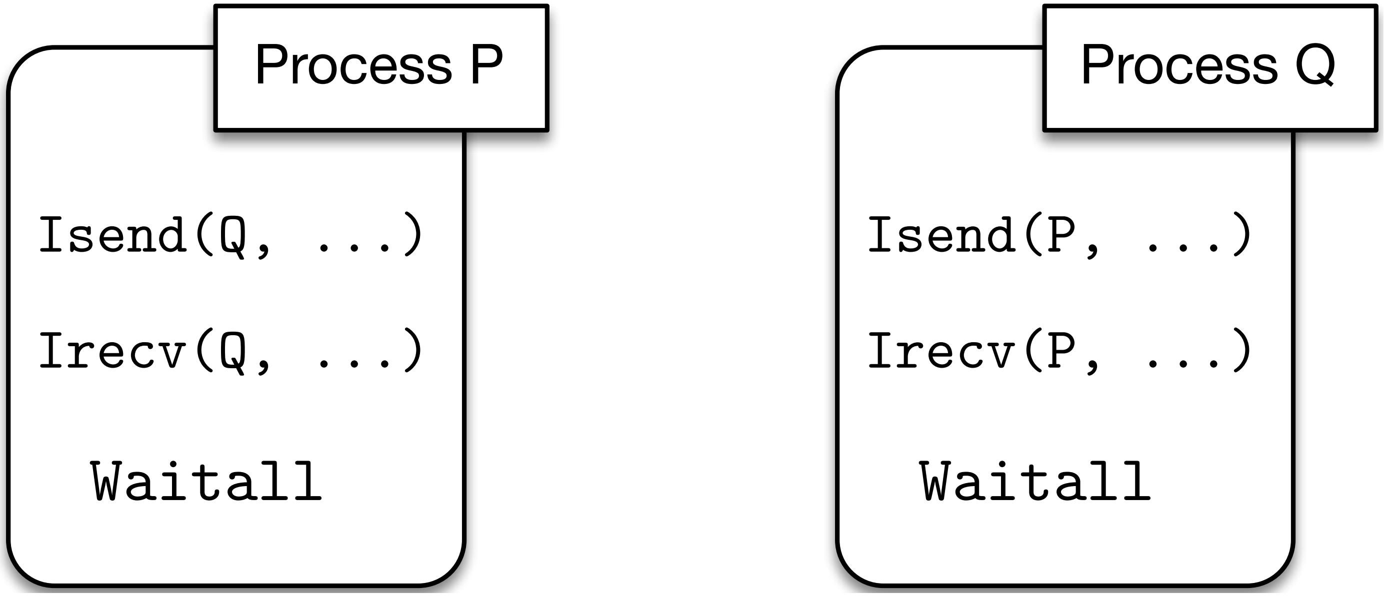
Irecv also returns a request handle



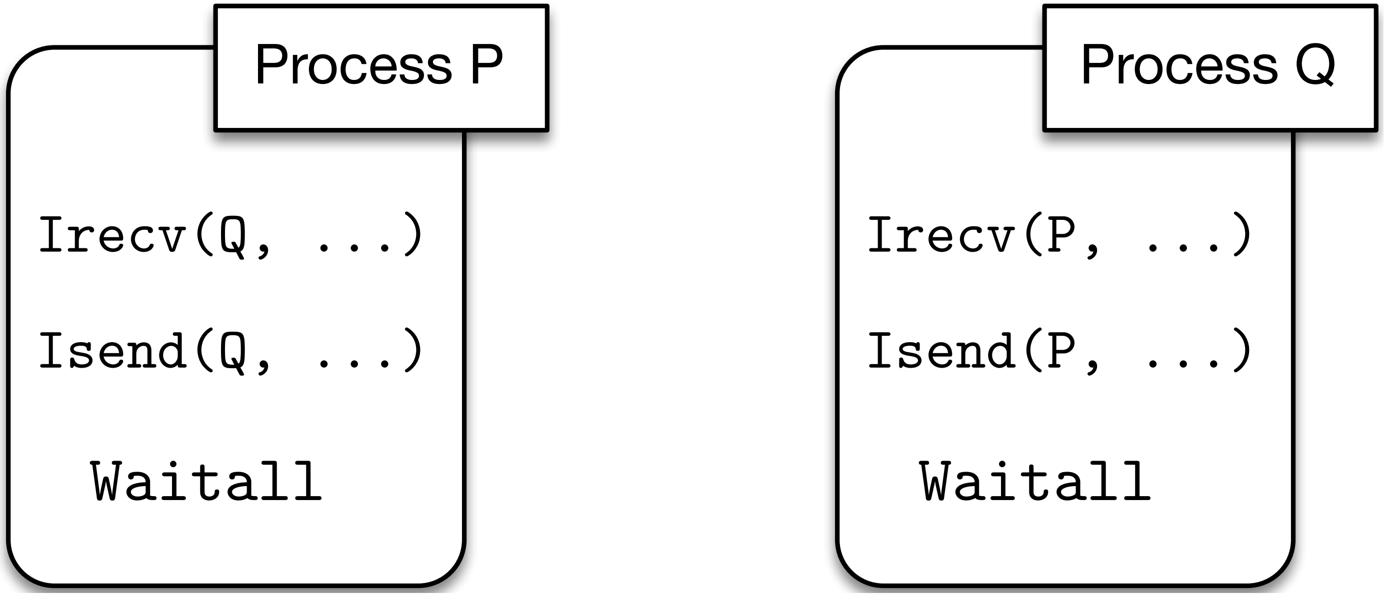
That can be waited on and will return when data are ready

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

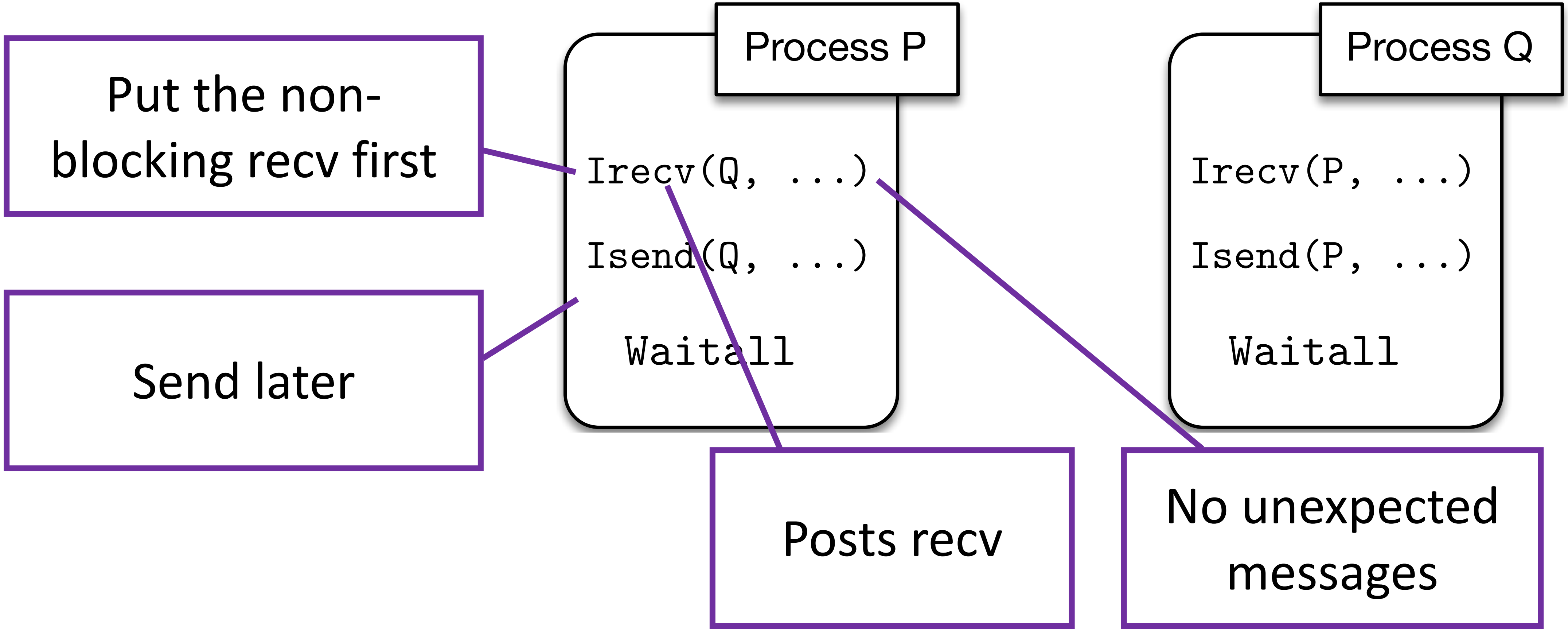
Before


Process P
Isend(Q, ...)
Irecv(Q, ...)
Waitall
Process Q
Isend(P, ...)
Irecv(P, ...)
Waitall

After


Process P
Irecv(Q, ...)
Isend(Q, ...)
Waitall
Process Q
Irecv(P, ...)
Isend(P, ...)
Waitall

After



Put the non-blocking recv first
Send later
Process P
Irecv(Q, ...)
Isend(Q, ...)
Waitall
Process Q
Irecv(P, ...)
Isend(P, ...)
Waitall
Posts recv
No unexpected messages

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[])
```

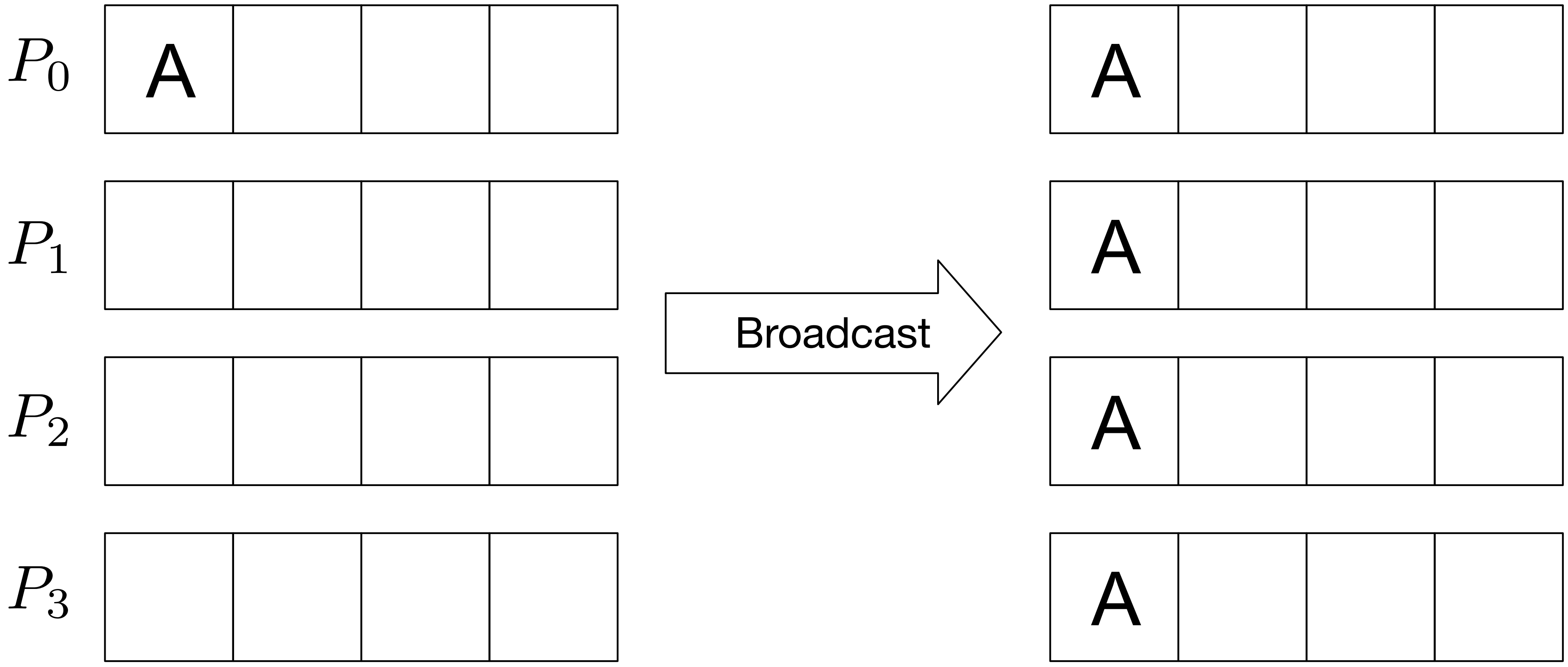
```
static bool Request::Testany(int count, Request
↪  array_of_requests[], int& index, Status& status)
static bool Request::Testany(int count, Request
↪  array_of_requests[], int& index)
```

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

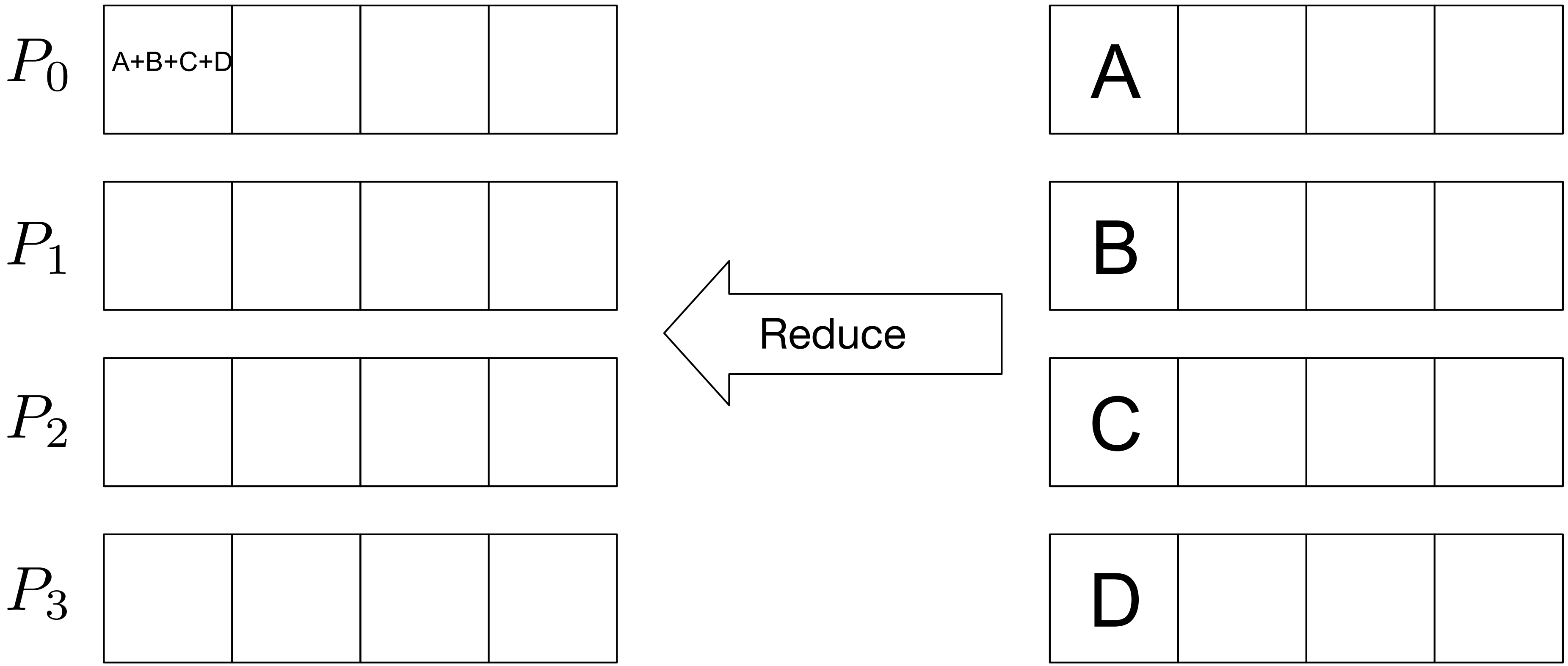
Bcast

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



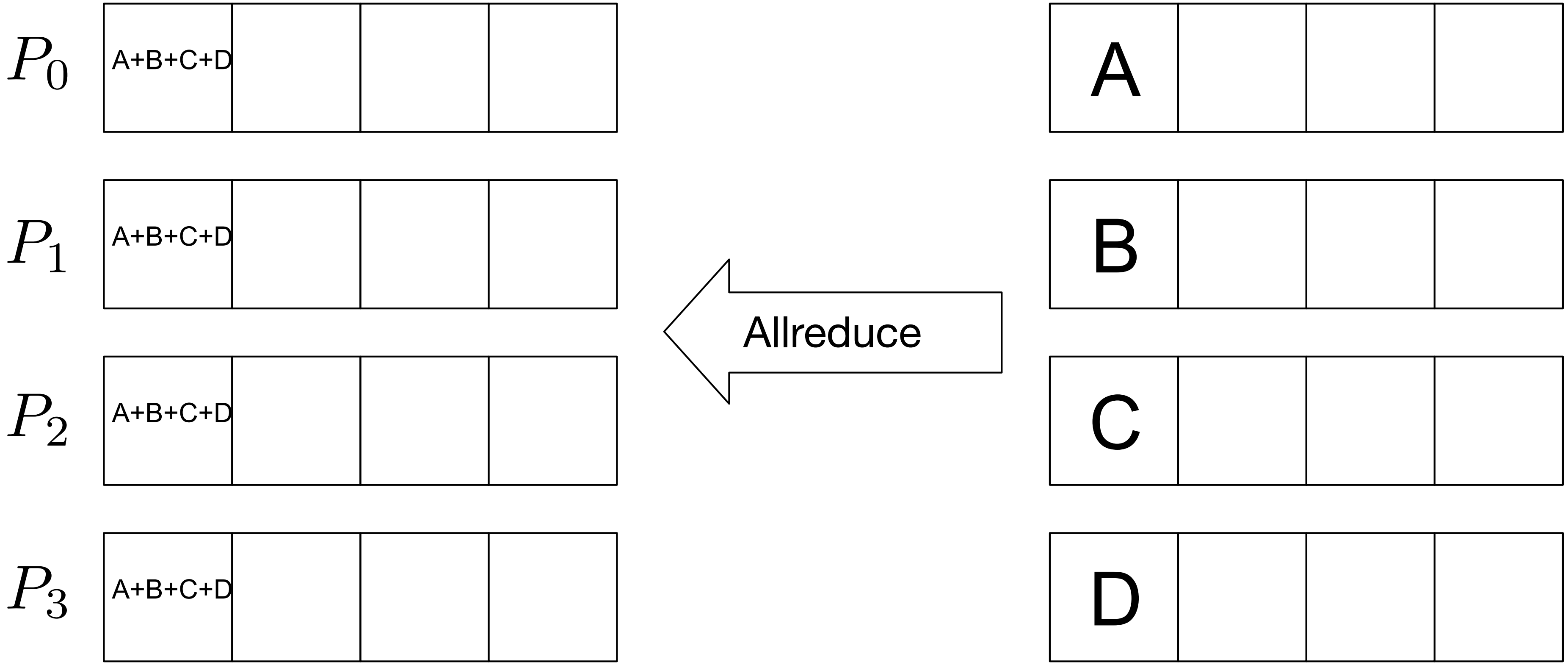
Reduce

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



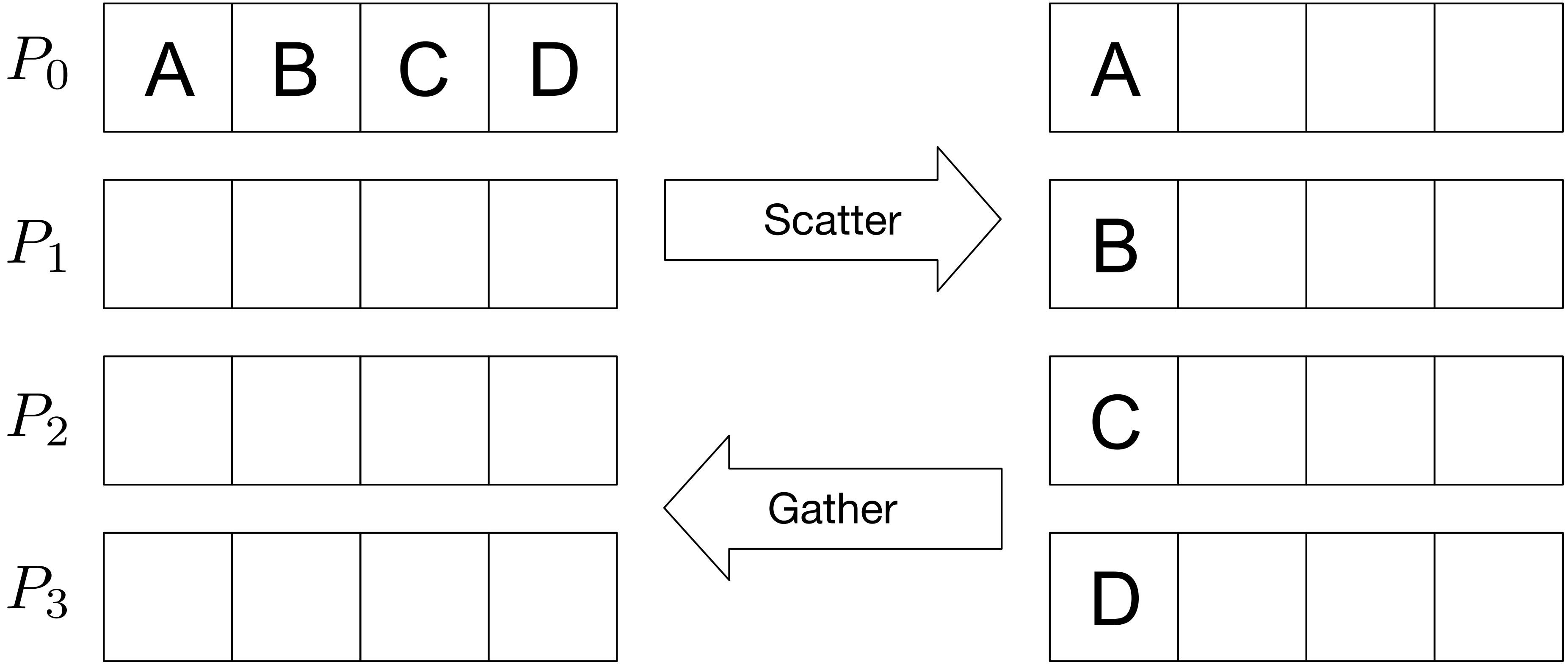
Allreduce

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



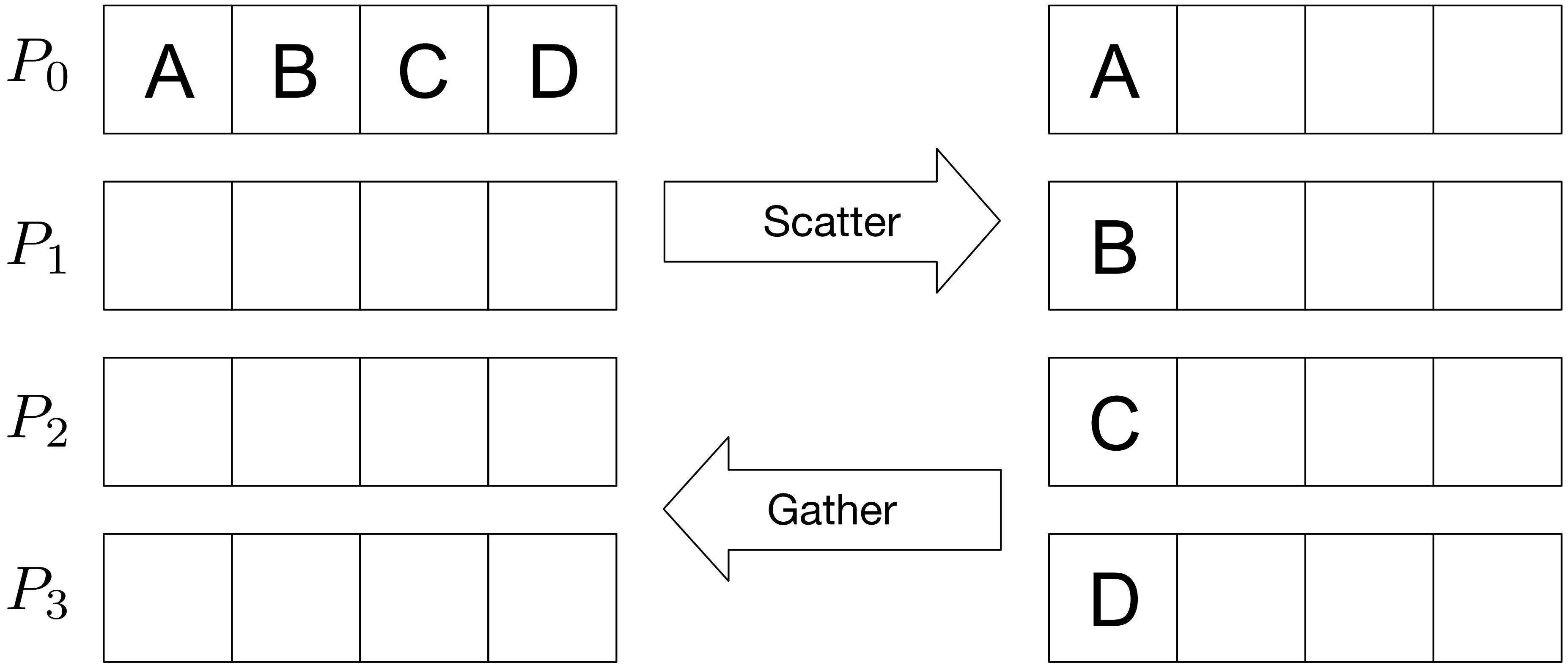
Scatter/Gather

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



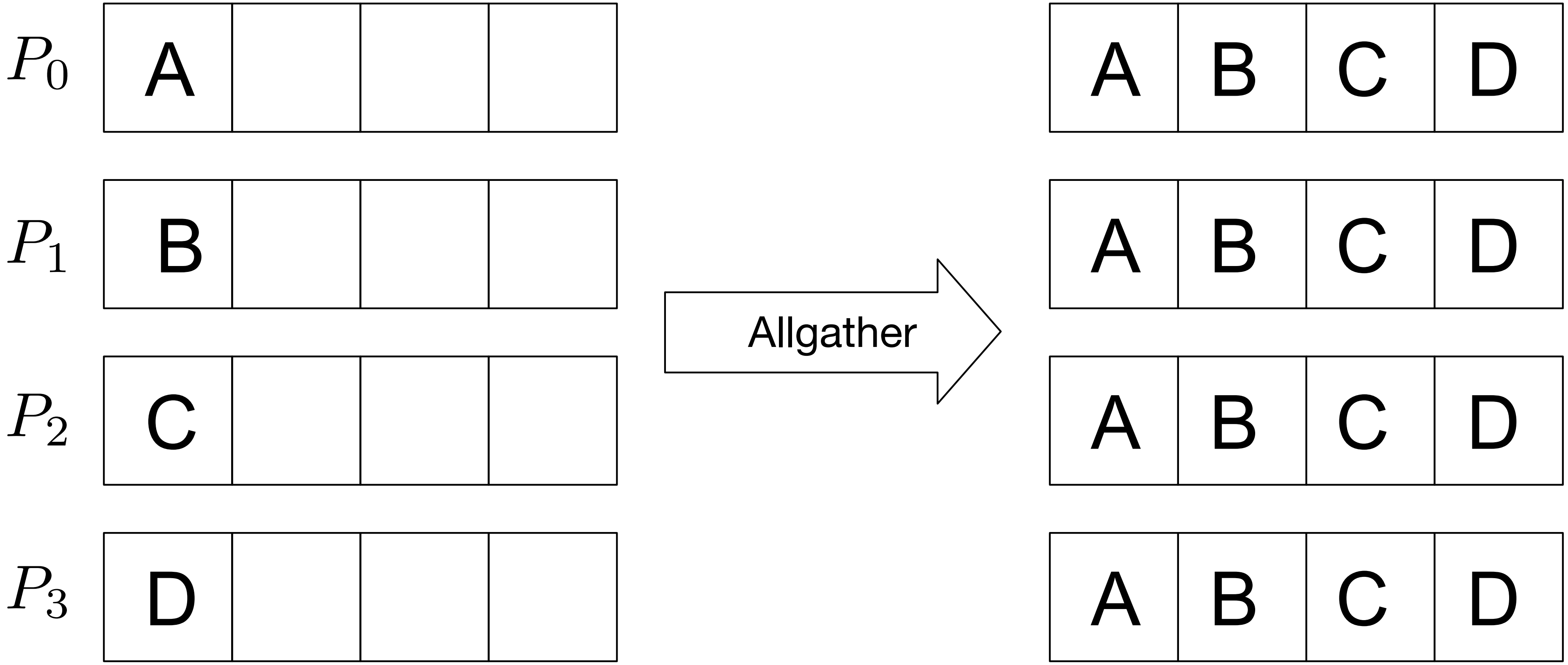
Scatter/Gather

```
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
```



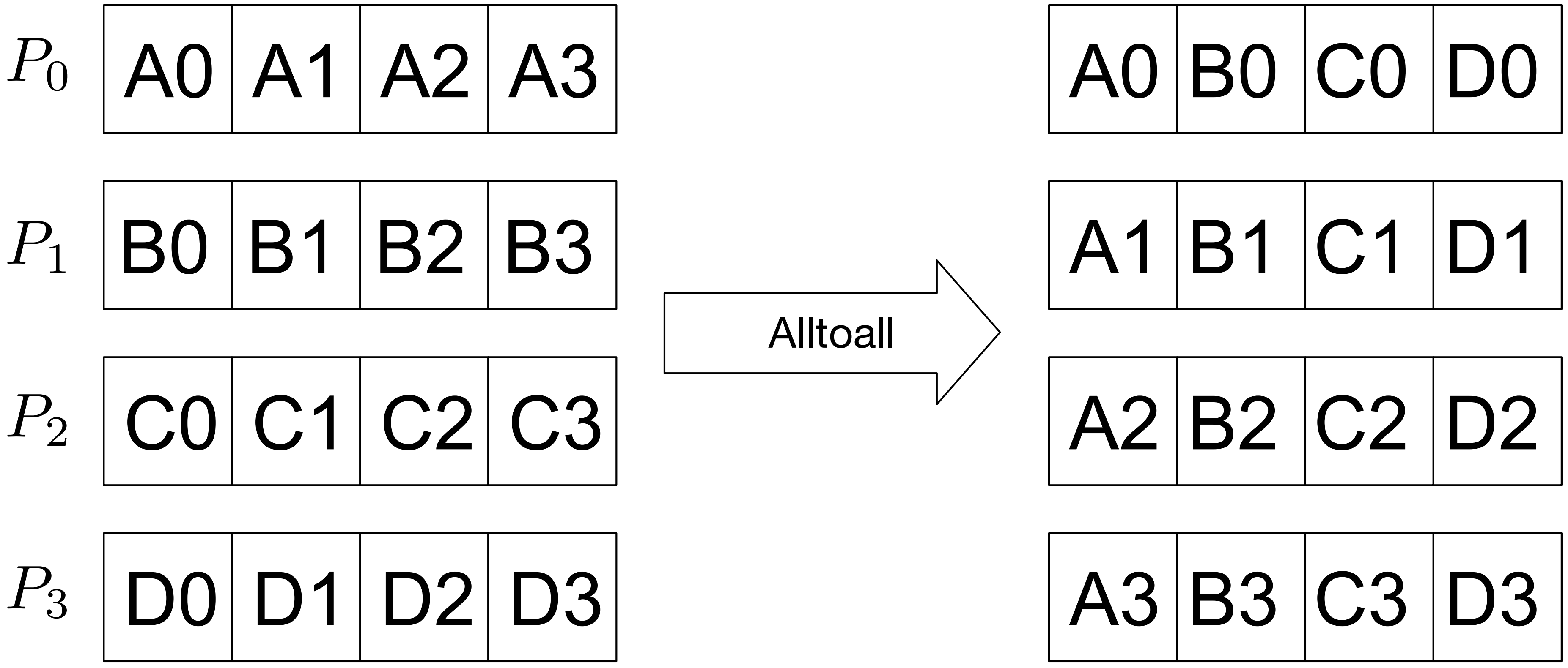
Allgather

```
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)
```

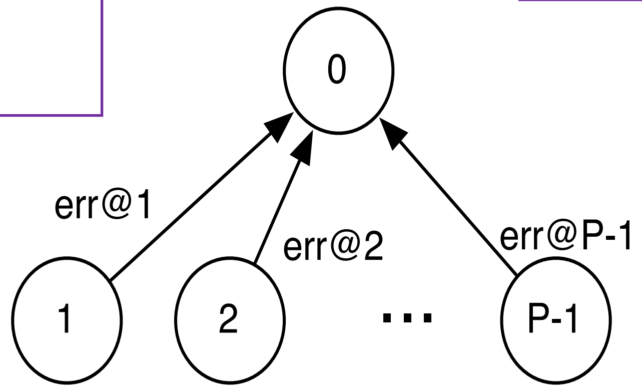


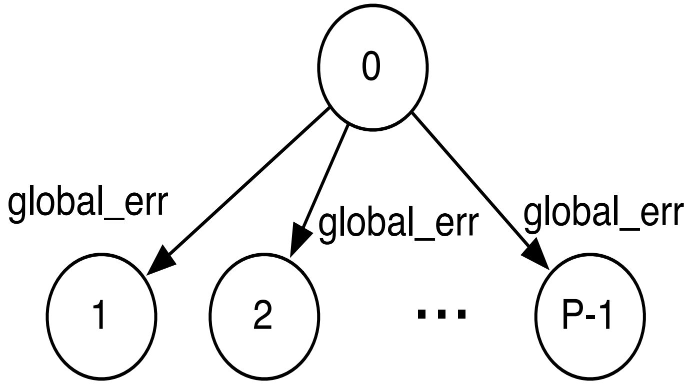
All Reduce

All data is sent to root

How many sends?

How many receives?

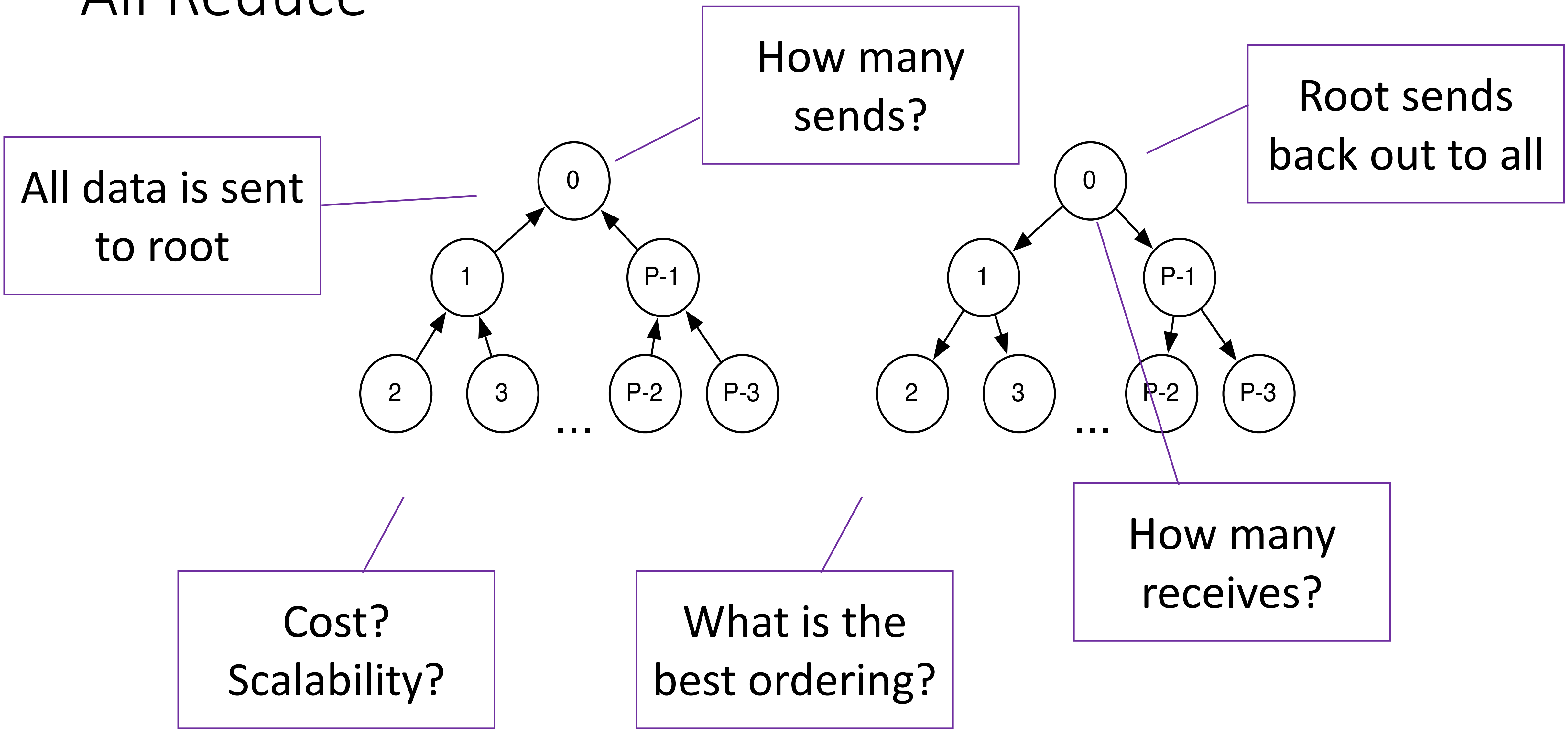




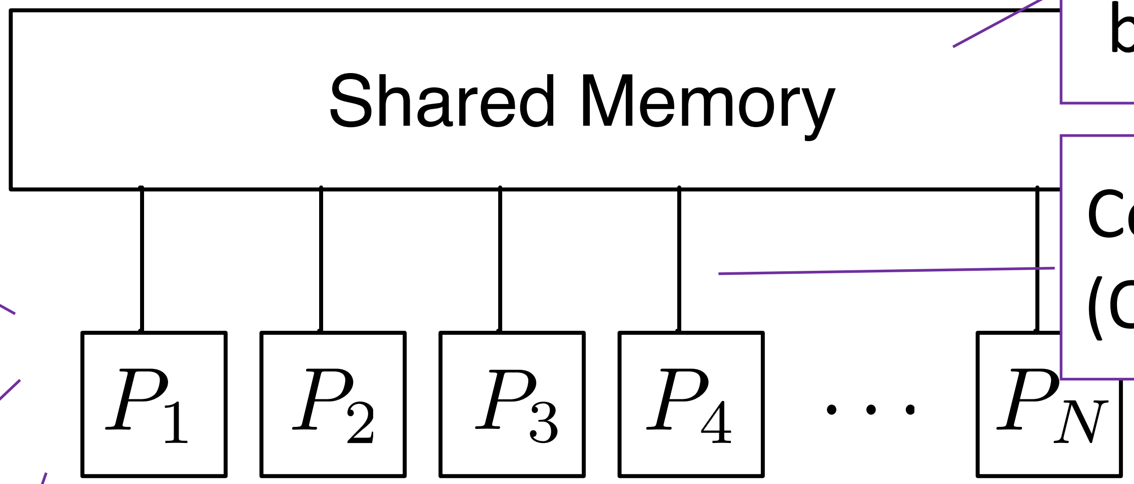
Cost? Scalability?

Root sends back out to all

All Reduce


All data is sent to root
How many sends?
Root sends back out to all
0
1
P-1
2
3
...
P-2
P-3
0
1
P-1
2
3
...
P-2
P-3
How many receives?
Cost? Scalability?
What is the best ordering?

Parallel Random Access Machine



Memory shared by all processes

Some number (fixed or infinite) number of processors

Completely UMA (O(1) read/write)

Processors all execute same steps in synchrony (but can lay out)

Everything interesting in parallel computing is about data dependence

At each cycle, processors read, write, or compute (one operation)

Powerful tool for analysis of parallel algorithm

Assume tasks done in parallel are perfectly parallelizable

PRAM cont.

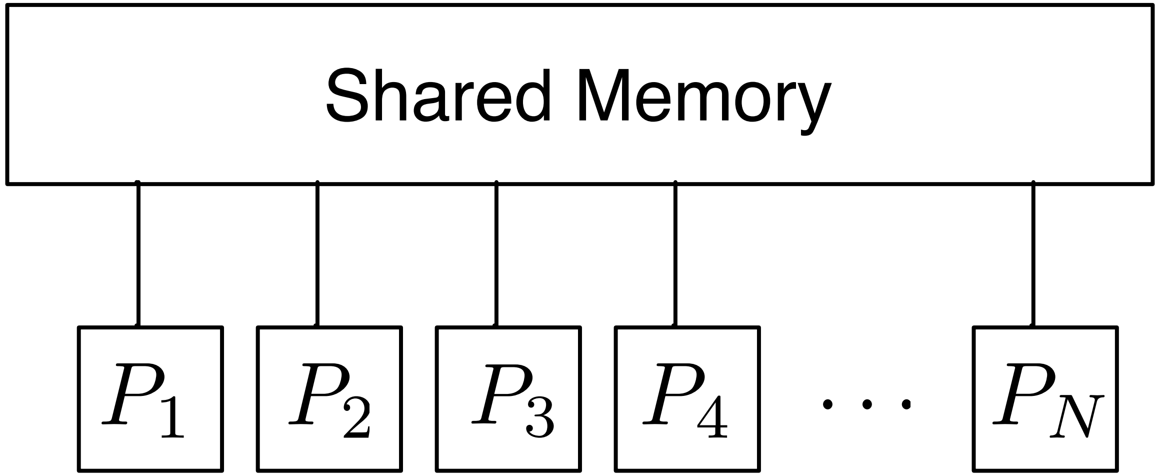
- Several types of PRAM
  - EREW - Exclusive Read Exclusive Write
  - CREW - Concurrent Read Exclusive Write
  - ERCW - Exclusive Read Concurrent Write
  - CRCW - Concurrent Read Concurrent Write
- Stronger models can be emulated by weaker models

Reads and writes need to be ordered

Writes need to be ordered

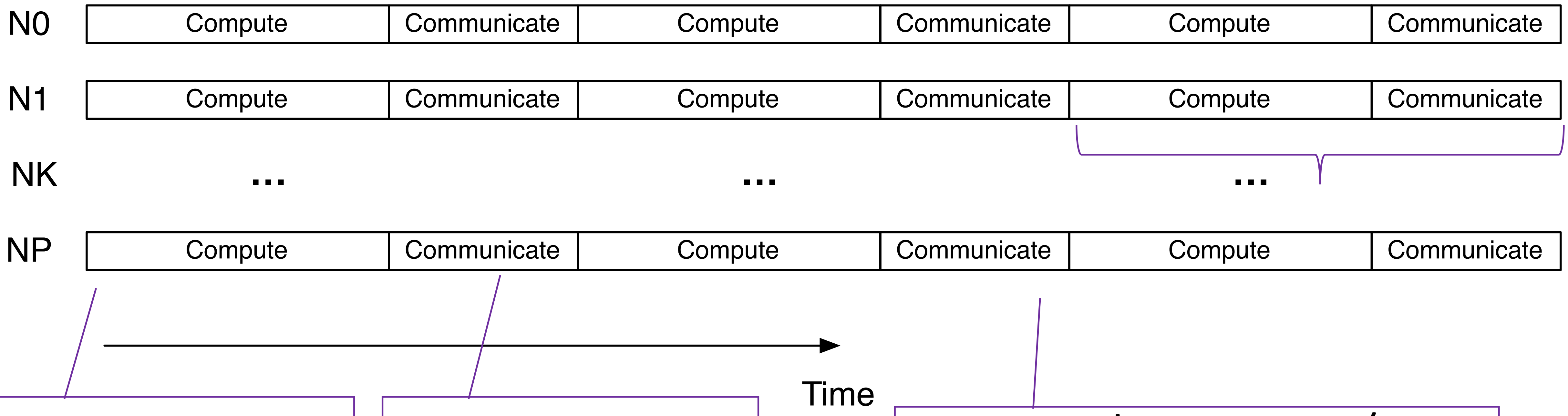
Reads need to be ordered

Nothing needs to be ordered



Compute / Communicate

"Bulk Synchronous Parallel" (BSP)



This is an almost universal pattern

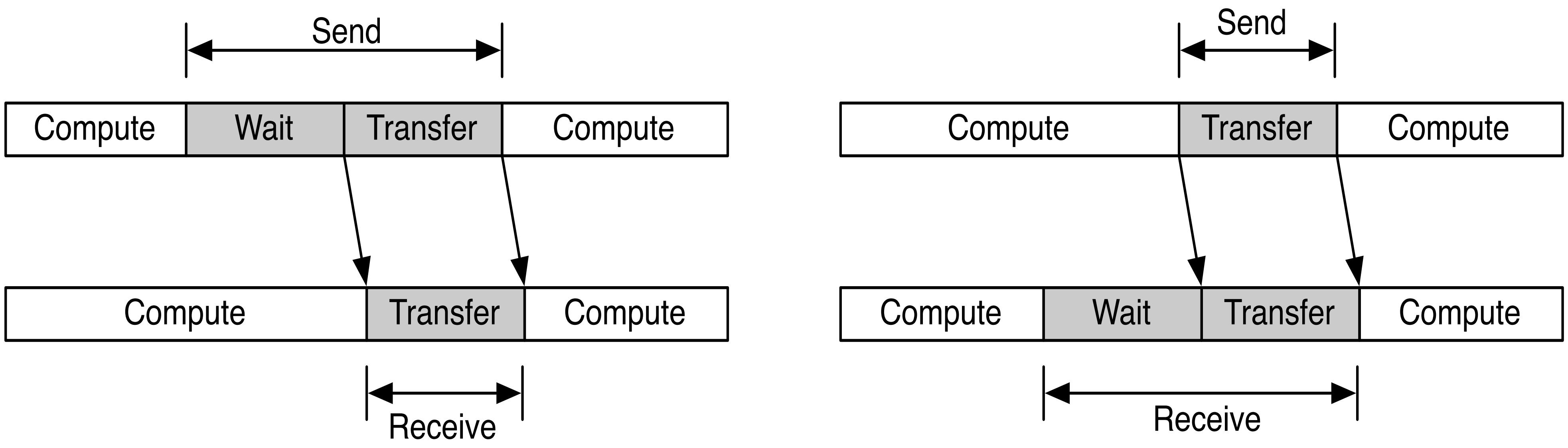
Processors are still only loosely coupled

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

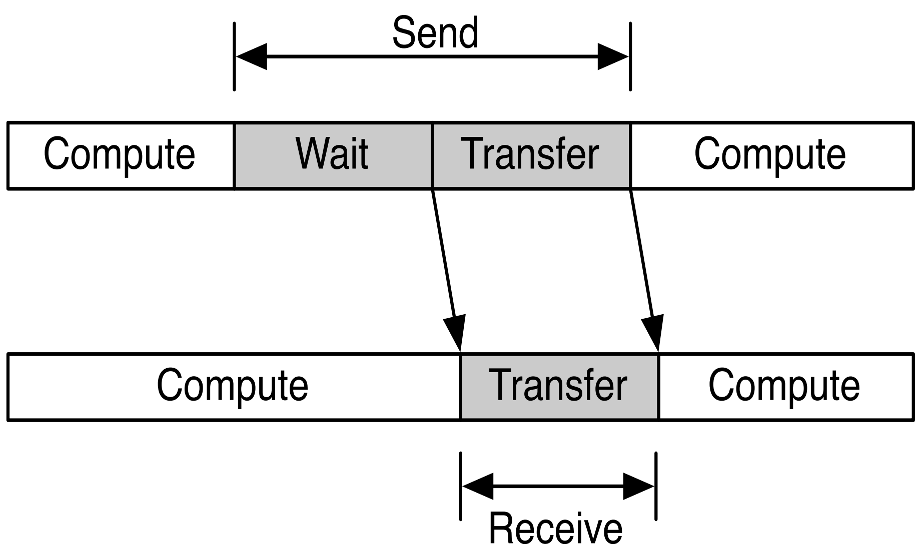
Performance Model

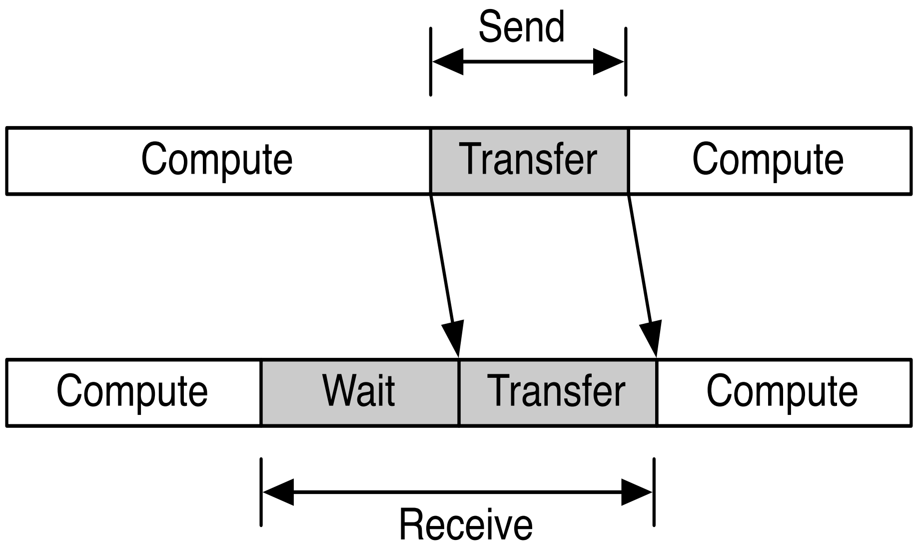
$$T_{communicate} = T_{latency} + T_{bandwidth} = T_L + r_{nic} \cdot Size$$

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



Synchronous vs Asynchronous

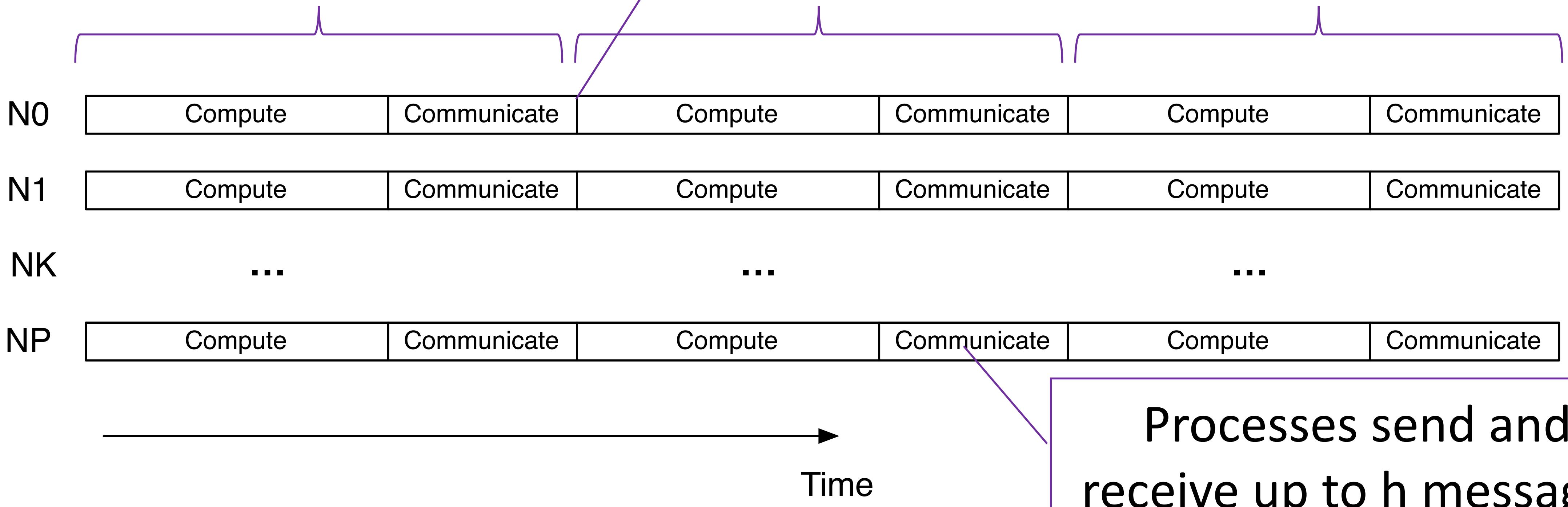




Bulk Synchronous Parallel (BSP)

Series of ***supersteps***

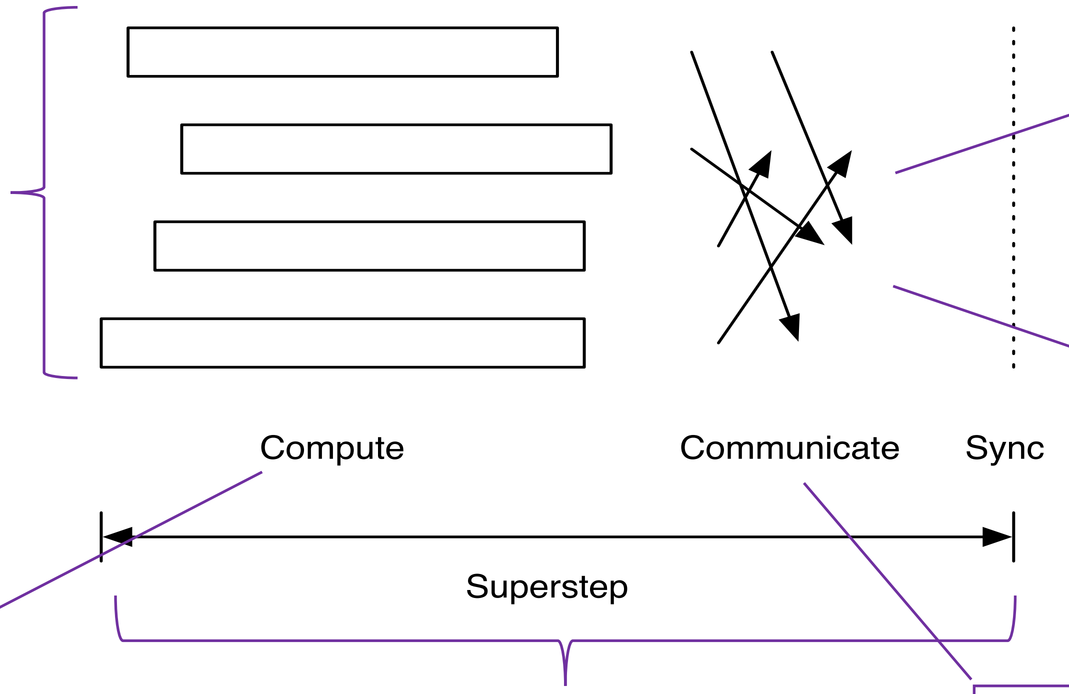
Compute can ***only*** use data present at beginning of superstep



Processes send and receive up to h messages

BSP cont.

p processes



Throughput of the network is g

And there are h messages

What is compute time?

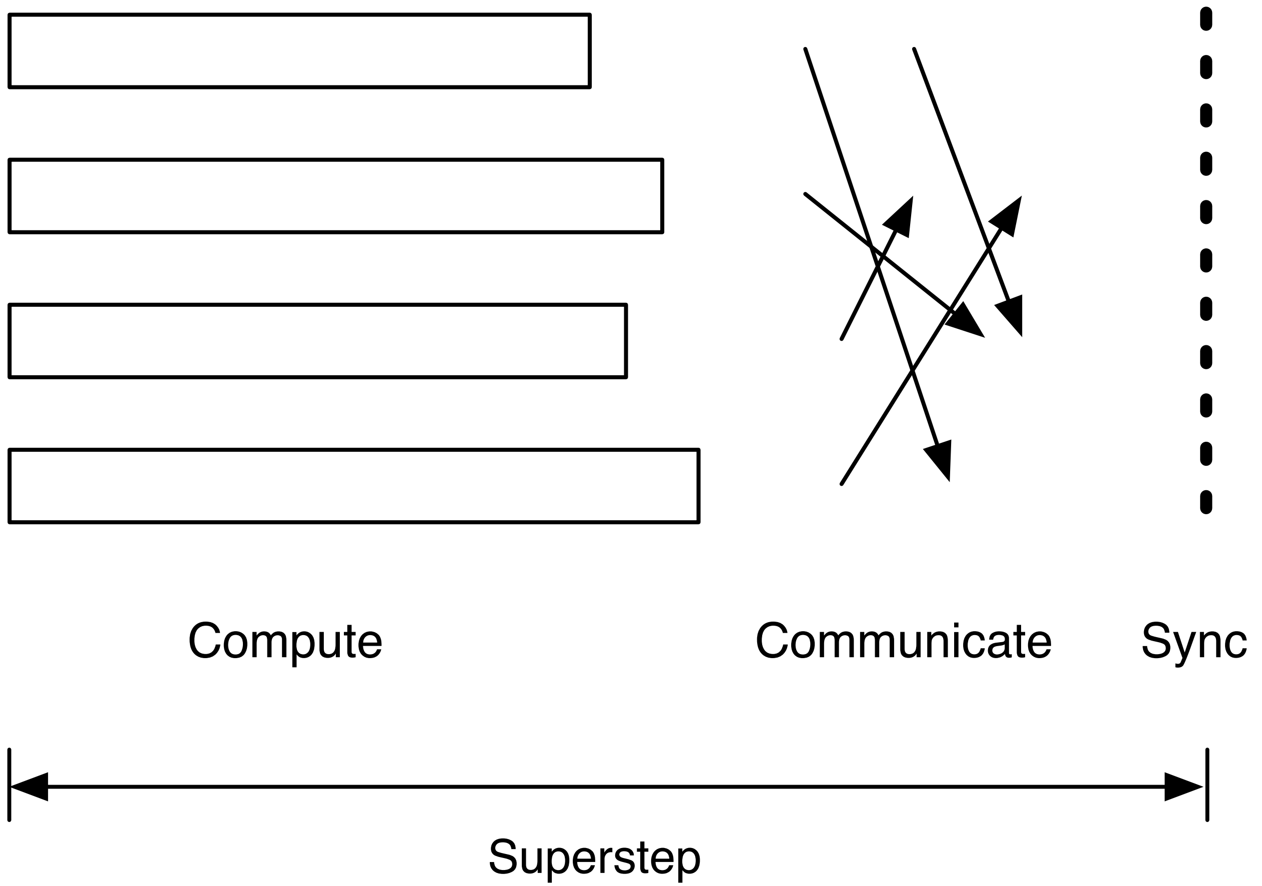
What is communicate time in terms of g and h?

Total length of a superstep is L

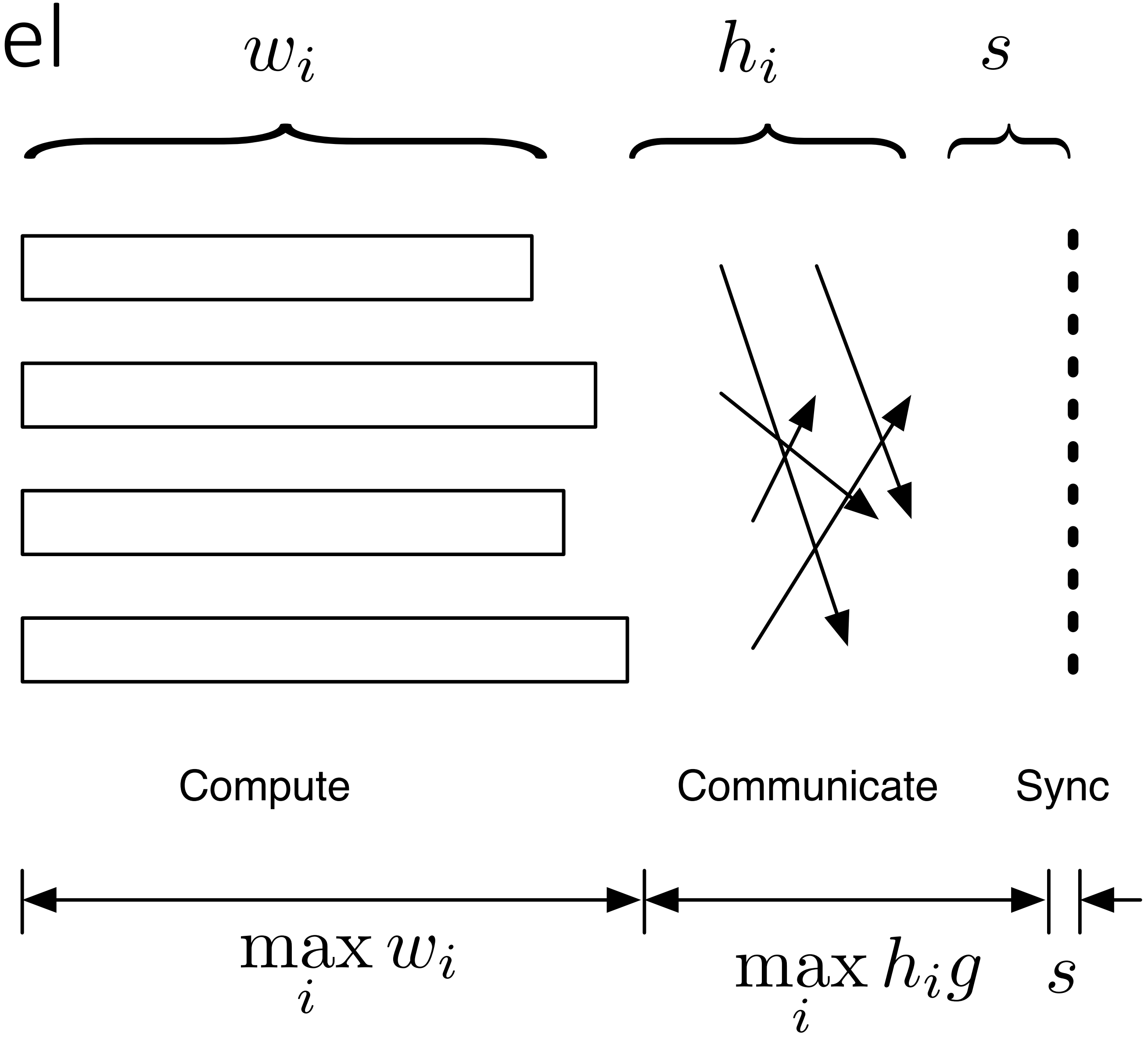
Valiant, L.G. Optimally universal parallel computers. Phil. Trans. R. Soc. Lond. A326 (1988) 373-376.

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

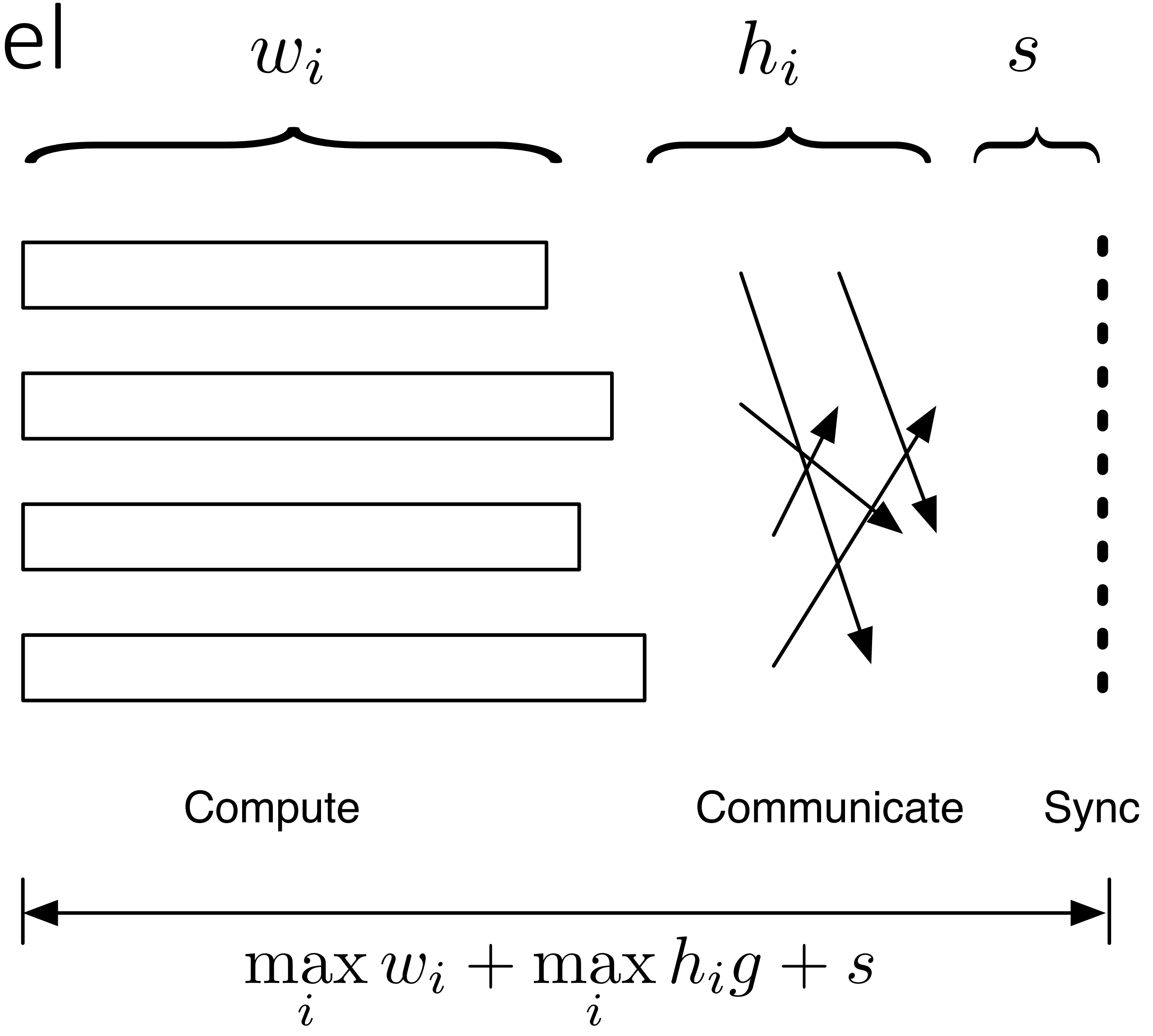
BSP Model



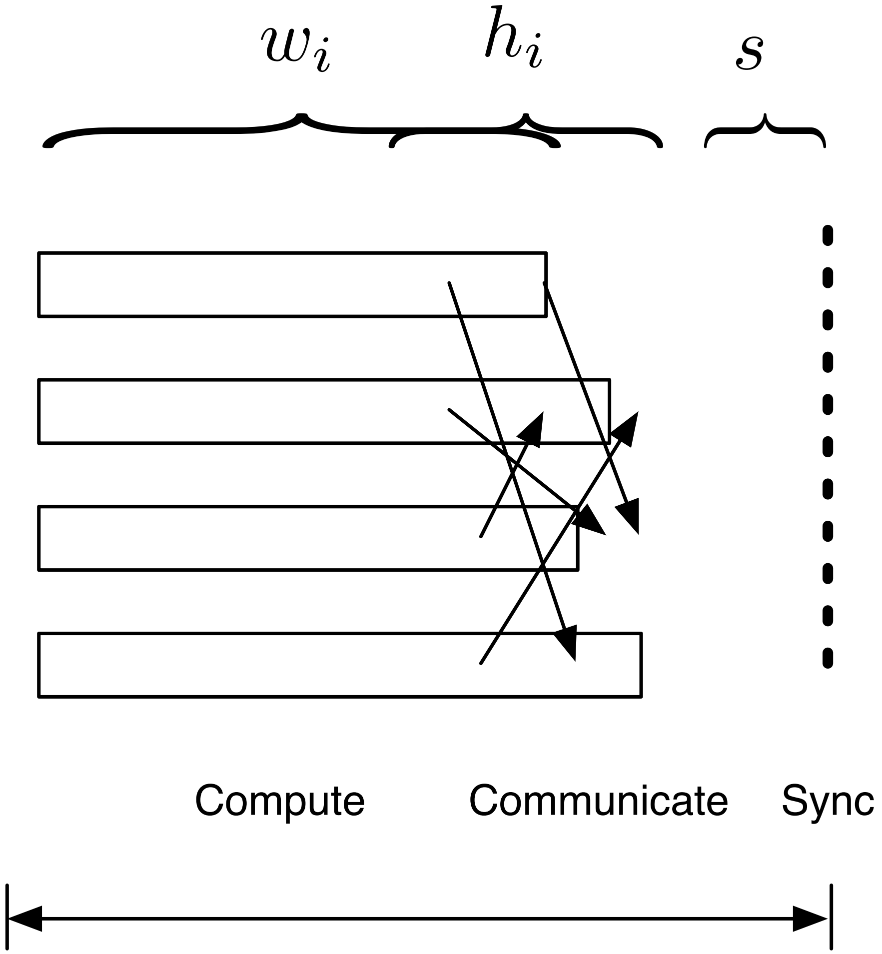
BSP Model



BSP Model



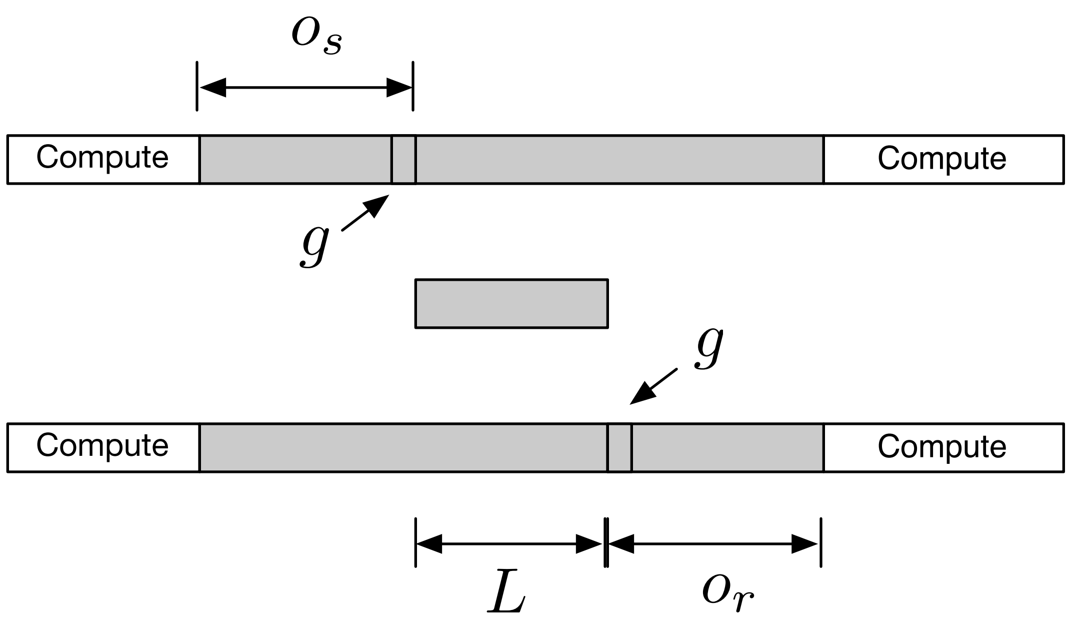
BSP with asynchronous communication



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

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

- Send single message

$$T = 2o + L$$

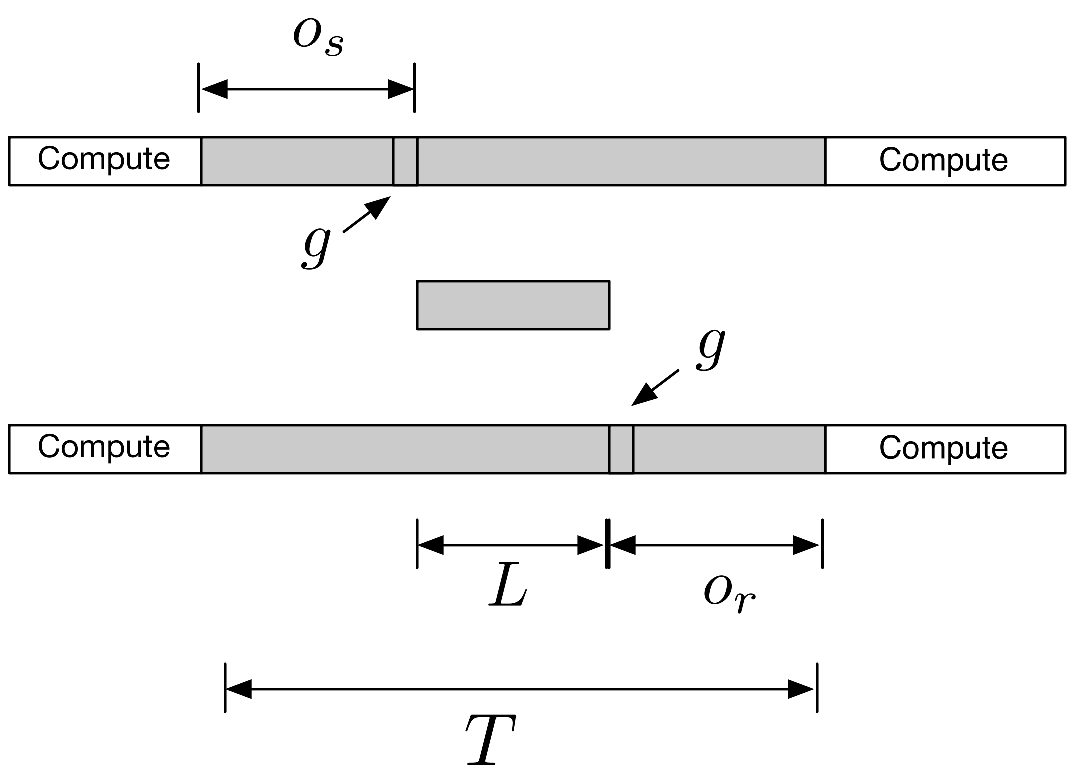
- Ping-pong round trip

$$T = 4o + 2L$$

- N messages in a row

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

Why?



LogP cont.

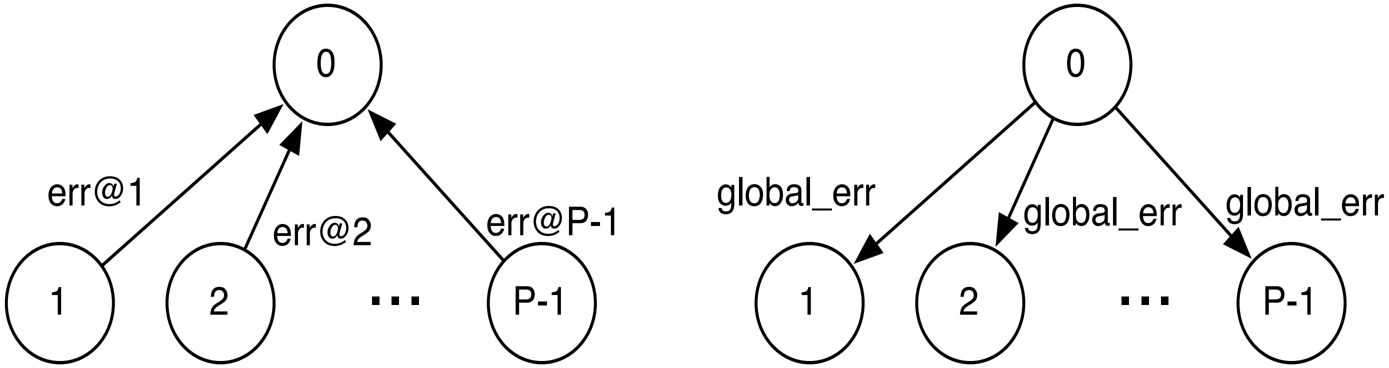
- Allows more precise scheduling of communication
  - Reading a remote memory location
    - BSP - next superstep, *L* cycles
    - LogP - *2L + 4o* cycles
- No special synchronization hardware
- Parameters can be experimentally determined for a given machine/architecture
- No special treatment for long messages

Applications: Reduce

- BSP
  - O(log n) supersteps
  - L = time to read two memory locations and write one

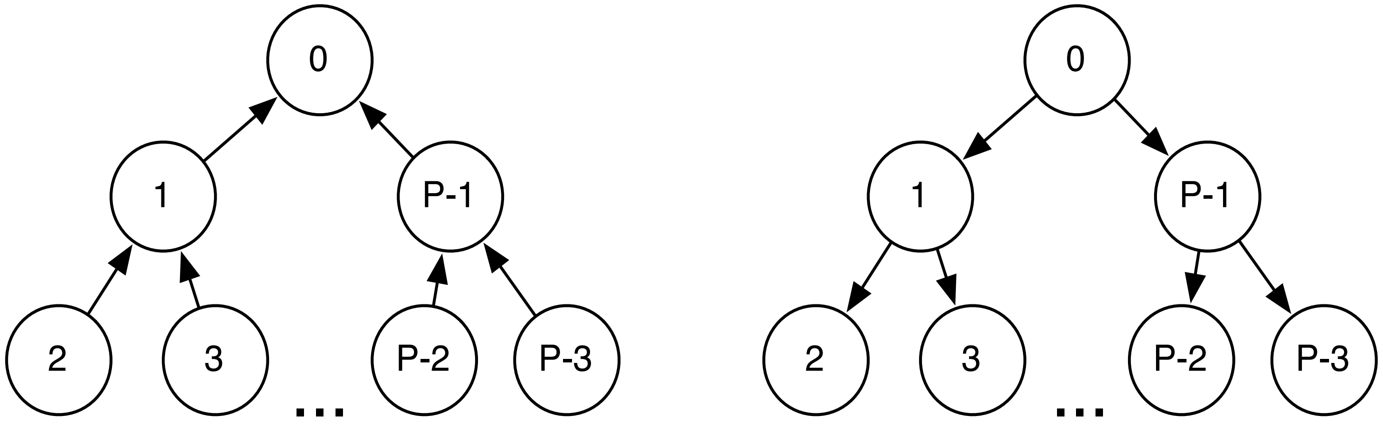
LogP Analysis

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

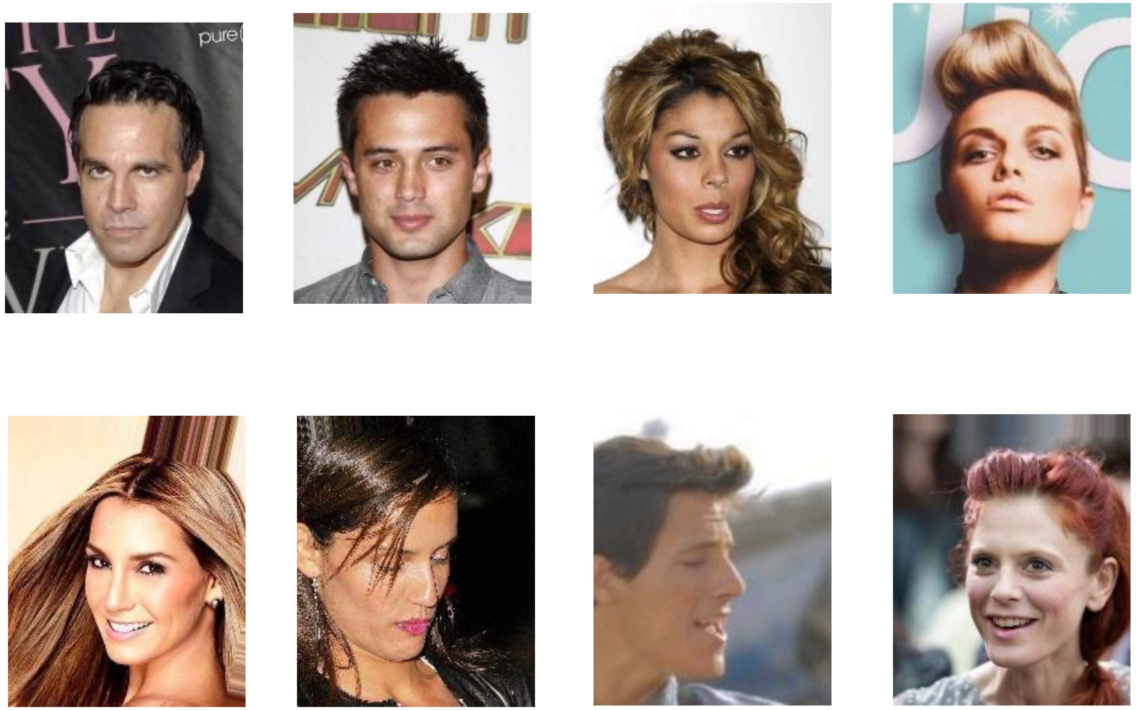


LogP Analysis

- Binary tree
  - o for each leaf processor to send its value to its 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)



Name This Famous Person

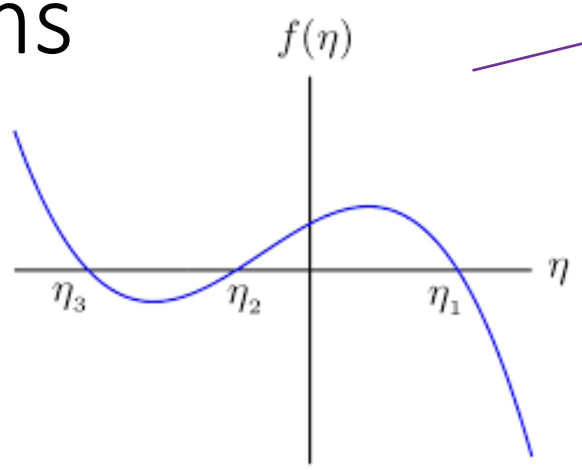


Name This Famous Person

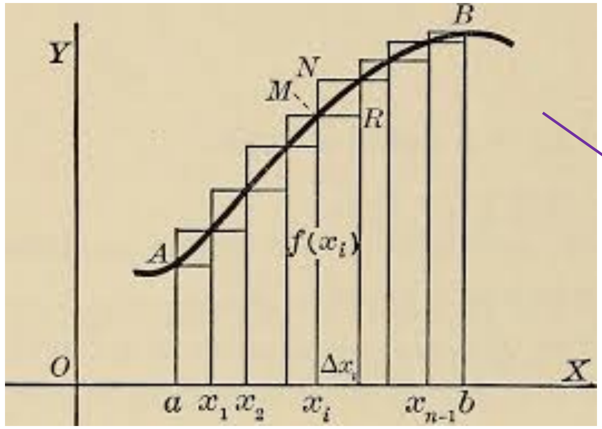
Fundamental Theorems

$$N = \prod_{i=0}^{m} x_i$$

Arithmetic: Every number is a product of primes



Algebra: Every polynomial has a root



$$\frac{\mathrm{d}}{\mathrm{dt}} \int_0^t f(t)dt = \int_0^t \left[\frac{\mathrm{d}}{\mathrm{dt}} f(t)\right] = f(t)$$

Calculus: The integral of the derivative is the derivative of the integral

Software engineering: We can solve any problem by introducing an extra level of indirection

David Wheeler via Butler Lampson via Andrew Koenig

Linear Systems

$$e_i = (0, 0, \dots 1 \dots 0)$$

$$x = \sum_{i=0}^{\dim X} \alpha_i y_i$$

Every linear space has a basis

Any element in the space can be expressed as weighted sums of members of the basis

Nice orthonormal basis

$$x = (x_0, x_1 ... x_{N-1})$$

$$\hat{x} = (\alpha_0, \alpha_1, \dots \alpha_{N-1})$$

The same element has multiple representations

Linear Systems

$$e_i = (0, 0, \ldots 1 \ldots 0)$$

$$x = \sum_{i=0}^{\dim X} \alpha_i y_i$$

$$x = (x_0, x_1 ... x_{N-1})$$

What is x^?

$$x = \alpha_0(1, 0, \ldots, 0) + \alpha_1(0, 1, 0, \ldots, 0) + \ldots + \alpha_{N-1}(0, 0, \ldots, 1)$$

$$\hat{x} = (\alpha_0, \alpha_1, \ldots \alpha_{N-1})$$

Which is equal to?

Transforming From One Representation to Another

$$x = (x_0, x_1 ... x_{N-1}) \Rightarrow \hat{x} = (\alpha_0, \alpha_1, \ldots \alpha_{N-1})$$

$$x = \sum_{i=0}^{\dim X} \alpha_i y_i \Rightarrow x = \alpha_0 y_0 + \alpha_1 y_1 + \ldots + \alpha_{N-1} y_{N-1}$$

$$Y = [y_0, y_1, \ldots, y_{N-1}]$$

What is this?

Transforming From One Representation to Another

$$\begin{bmatrix} y_{0,0} & y_{0,1} & \cdots & y_{0,N-1} \\ y_{1,0} & y_{1,1} & \cdots & y_{1,N-1} \\ & & \cdots & \\ y_{N-1,0} & y_{N-1,1} & \cdots & y_{N-1,N-1} \end{bmatrix}$$

!

$$Y\alpha = x$$

Conditions?

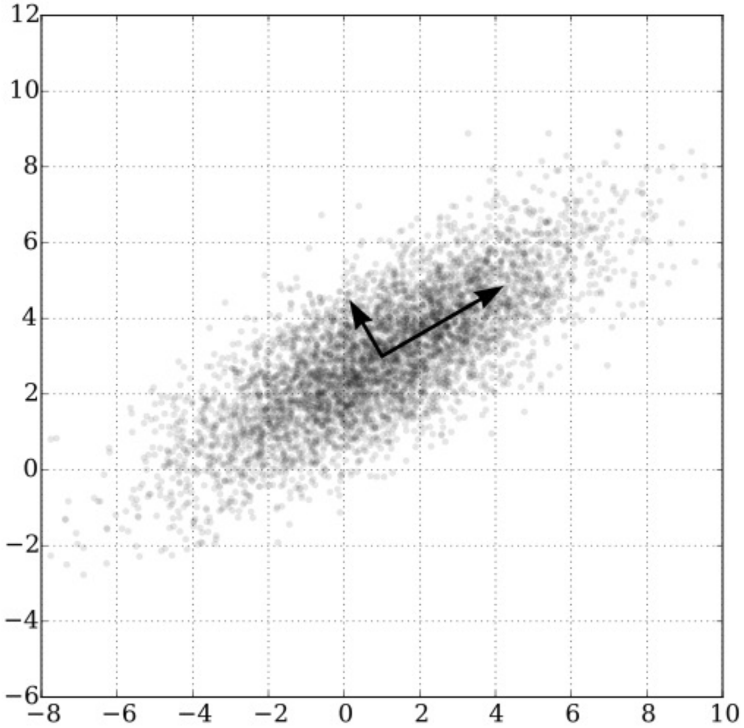
$$\begin{bmatrix} y_{0,0} & y_{0,1} & \cdots & y_{0,N-1} \\ y_{1,0} & y_{1,1} & \cdots & y_{1,N-1} \\ & & \cdots & \\ y_{N-1,0} & y_{N-1,1} & \cdots & y_{N-1,N-1} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \cdots \\ \alpha_{N-1} \end{bmatrix}$$

$$\alpha = Y^{-1}x$$

$$\begin{bmatrix} y_{0,0} & y_{0,1} & \cdots & y_{0,N-1} \\ y_{1,0} & y_{1,1} & \cdots & y_{1,N-1} \\ & & \cdots & \\ y_{N-1,0} & y_{N-1,1} & \cdots & y_{N-1,N-1} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \cdots \\ \alpha_{N-1} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_{N-1} \end{bmatrix}$$

Principal Components

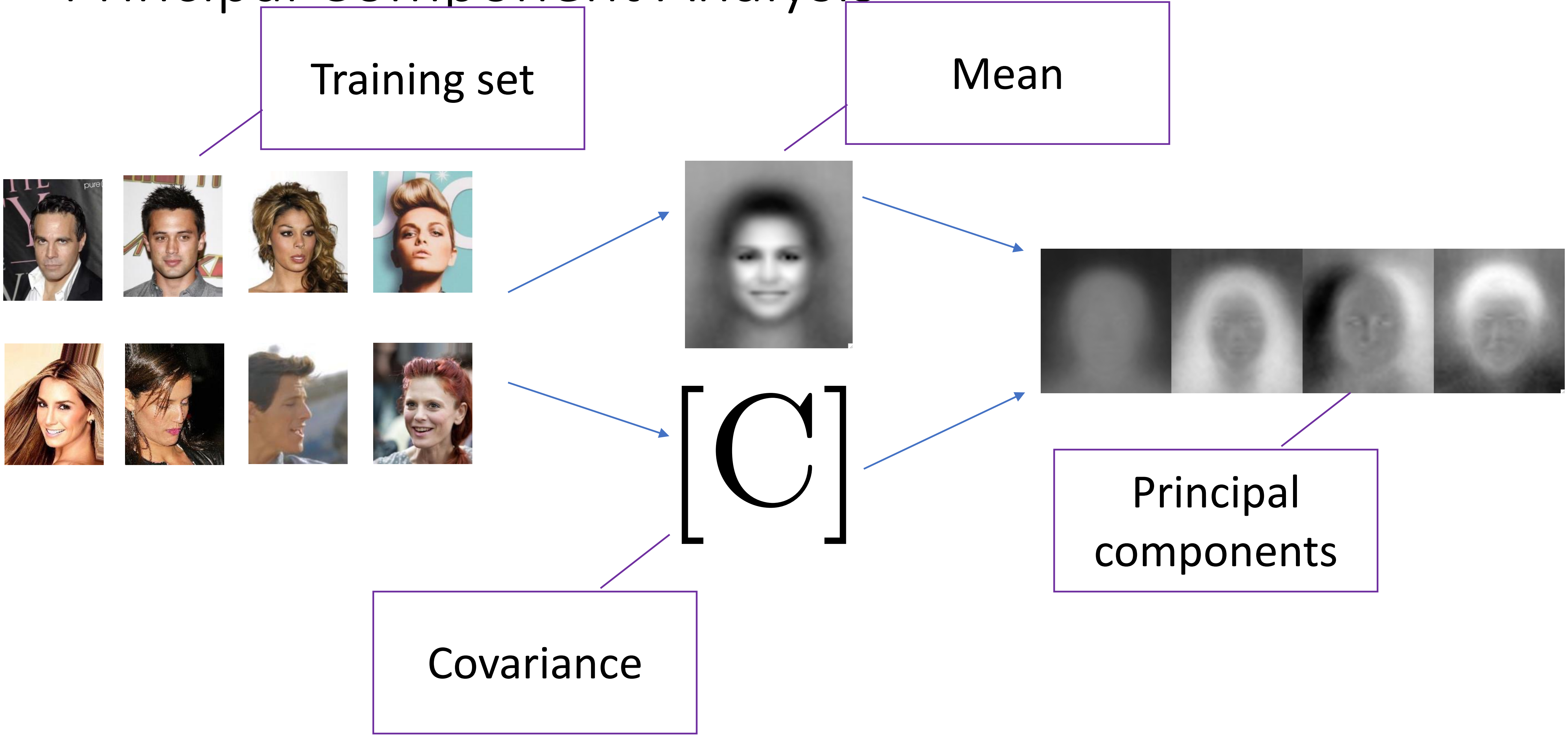
- Given a set of data, what is the best basis for representing elements of that set



Principal Components Analysis

- We are given a training set X of faces
- We want to find an orthonormal basis that can form an alternate representation of faces with as few dimensions as possible
  - Axes are the “principal components”
  - First axis captures as much of the data set as possible
  - Next axis captures as much of the data that isn’t captured by first
  - And so on
- We can represent any face with linear combination of the principal components

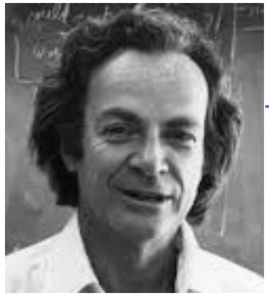
Principal Component Analysis



Principal Component Analysis

Project face onto principal components

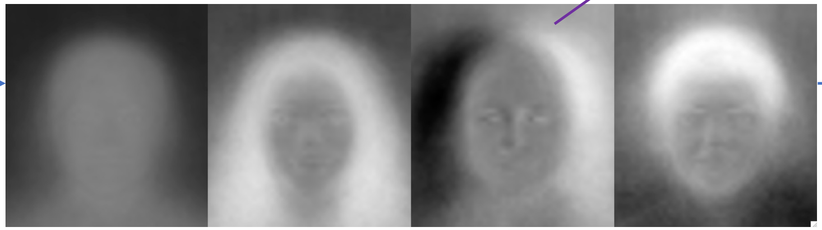
Representation in feature space

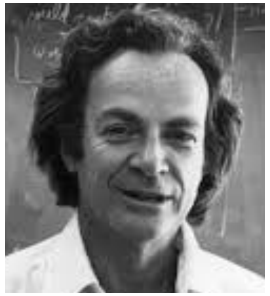


$$\{\phi_0, \phi_1, \ldots, \phi_{N-1}\}$$

How do we compute these?

$$\{\phi_0, \phi_1, \ldots, \phi_{N-1}\}$$





Linear combination of principal components

Recreate original face

Computing Principal Components

Let $\{f_0, f_1, \dots f_M\}$ be as set of faces. Each face $f_i \in \mathbb{R}^N$ where N is the total number of pixels in a face image and the elements of each f_i all have the same correspond to the pixels in face image i (without loss of generality, take lexicographical ordering). Let $\mathbf{F} \in \mathbb{R}^{M \times N}$ be a matrix in which every column i consists of f_i.

Let $\Phi \in \mathbb{R}^{N \times N}$ be an orthonormal matrix where each column is a feature vector ϕ_i. For a given face f_i we let $\tilde{f}_i = \sum_{j=0}^{K-1} \alpha_{i,j} \phi_j$ and define Φ such that for each K the difference between f_i and $\tilde{f}_i$ is minimized for $i = 0, 1, \dots, N-1$.

Let's start with the case of $K = 0$. Then each $\tilde{f}_i = \alpha_i \phi_0$. The best approximation of f_i will be the projection of f_i onto ϕ_0, i.e., $\alpha_i = \langle f_i, \phi_0 \rangle$.

Computing Principal Components

The sum of squares difference between all of the f_i and their projection onto ϕ_0 is

$$\sum_{i=0}^{N-1} \|f_i - \langle f_i, \phi_0\rangle \phi_0\|_2^2$$

The best choice for ϕ_0 is thus the one that minimizes this expression, i.e.,

$$\phi_0 = \operatorname{argmin} \sum_{i=0}^{N-1} \|f_i - \langle f_i, \phi_0\rangle \phi_0\|_2^2 = \operatorname{argmin} \sum_{i=0}^{N-1} \langle f_i - \langle f_i, \phi_0\rangle \phi_0, f_i - \langle f_i, \phi_0\rangle \phi_0\rangle$$

$$= \operatorname{argmin} \sum_{i=0}^{N-1} \left(\langle f_i, f_i\rangle - \langle f_i, \langle f_i, \phi_0\rangle \phi_0\rangle\right)$$

Computing Principal Components

$$\phi_0 = \operatorname{argmax} \sum_{i=0}^{N-1} \langle f_i, \langle f_i, \phi_0 \rangle \phi_0 \rangle = \operatorname{argmax} \sum_{i=0}^{N-1} \langle \langle \phi_0, f_i \rangle f_i, \phi_0 \rangle$$

$$= \operatorname{argmax} \sum_{i=0}^{N-1} \phi_0^T f_i f_i^T \phi_0 = \operatorname{argmax} \left[\phi_0^T \left(\sum_{i=0}^{N-1} f_i f_i^T \right) \phi_0 \right]$$

$$= \operatorname{argmax} \left(\langle \phi_0, C\phi_0 \rangle \right)$$

Covariance Matrix

Rayleigh Quotient

Constrained Optimization

Lagrangian

$$L(\phi_0) = \langle \phi_0, C\phi_0 \rangle - \sigma_0 \left(1 - \langle \phi_0, \phi_0 \rangle\right)$$

Maximize

$$\langle \phi_0, C\phi_0 \rangle$$

Subject to

$$\langle \phi_0, \phi_0 \rangle = 1$$

$$\nabla L(\phi_0) = 0$$

Will be maximized where gradient is zero

$$\nabla L(\phi_0) = C\phi_0 - \sigma_0\phi_0 = 0$$

Gradient of the Lagrangian

$$C\phi_0 = \sigma_0\phi_0$$

Largest eigenvalue

Corresponding eigenvector

Eigenfaces

$$f \;=\; \mathrm{read\_face}();$$

$$\phi = U^T f$$

$$\phi[K : N] = 0$$

$$f' = U\phi$$

Example

Our Code

- Read in faces data
- Compute mean of all faces
- Subtract mean from every face
- Compute covariance matrix C
- Compute eigendecomposition of matrix C
- Write out eigenface images

$$f = \text{read\_face}();$$

$$\phi = U^T f$$

$$\phi[K : N] = 0$$

$$f' = U\phi$$

Most computationally expensive step

We want to parallelize this

Thank You!

Creative Commons BY-NC-SA 4.0 License