

Introduction to MPI

Topics to be covered

- MPI vs shared memory
- Initializing MPI
- MPI concepts -- communicators, processes, ranks
- MPI functions to manipulate these
- Timing functions
- Barriers and the *reduction* collective operation

Shared and distributed memory

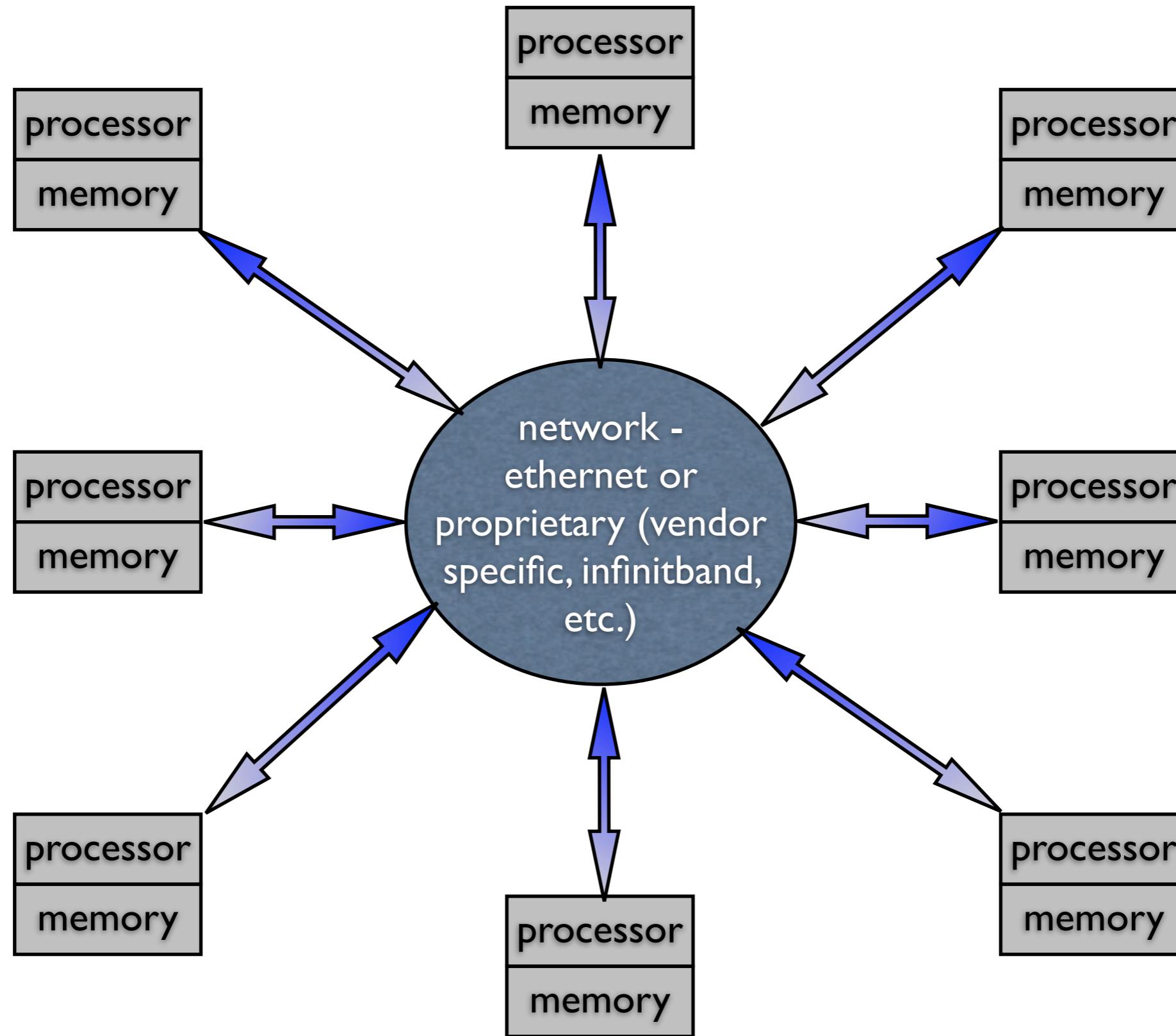
- **Shared memory**

- automatically maintained a consistent image of memory according to some memory model
- fine grained communication possible via loads, stores, and cache coherence
- model and multicore hardware support well aligned
- Programs can be converted piece-wise

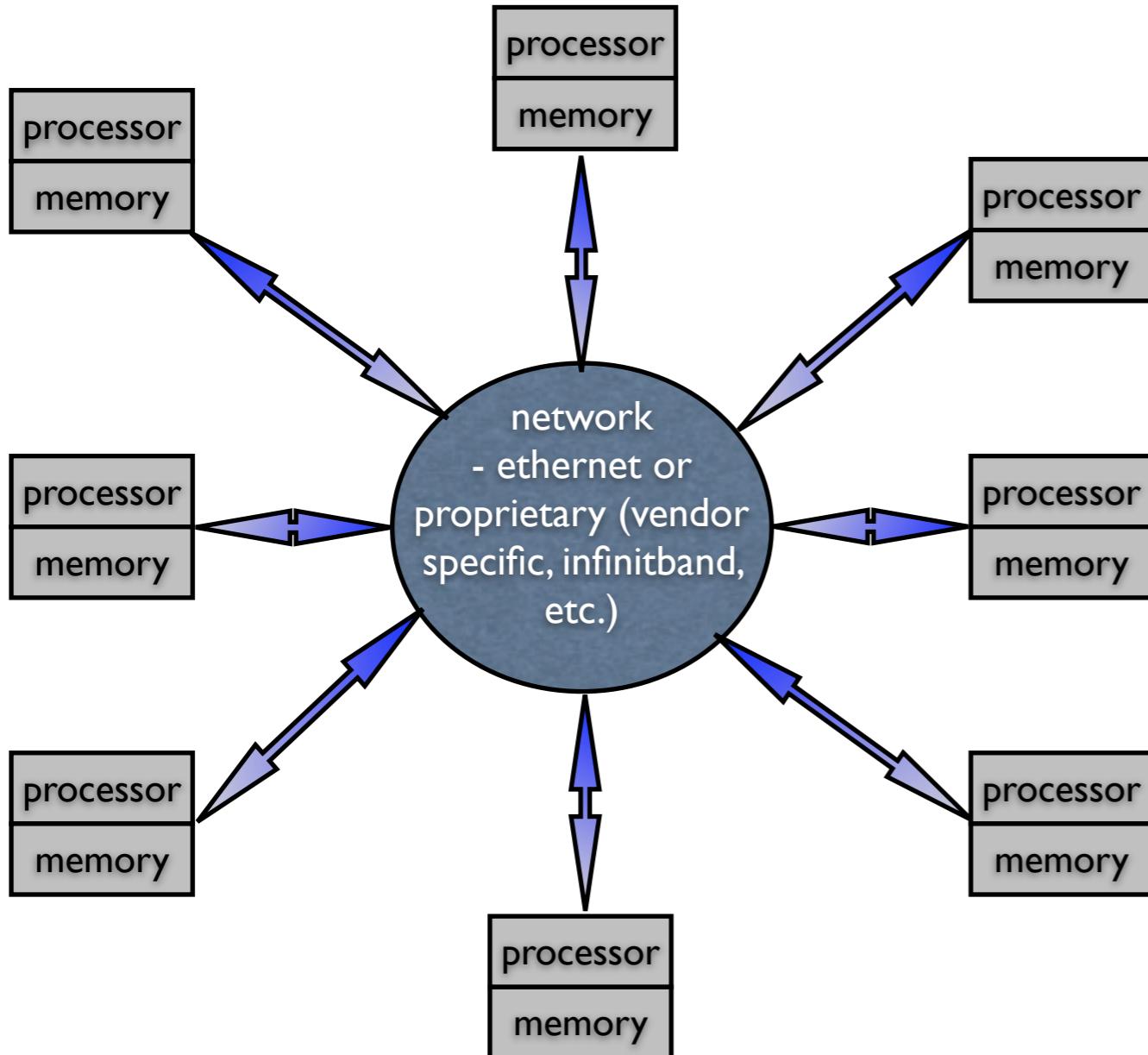
- **Distributed memory**

- Program executes as a collection of processes, all communication between processors explicitly specified by the programmer
- Fine grained communication in general too expensive -- programmer must aggregate communication
- Conversion of programs is all-or-nothing
- Cost scaling of machines is better than with shared memory -- well aligned with economics of commodity rack mounted blades

Message Passing

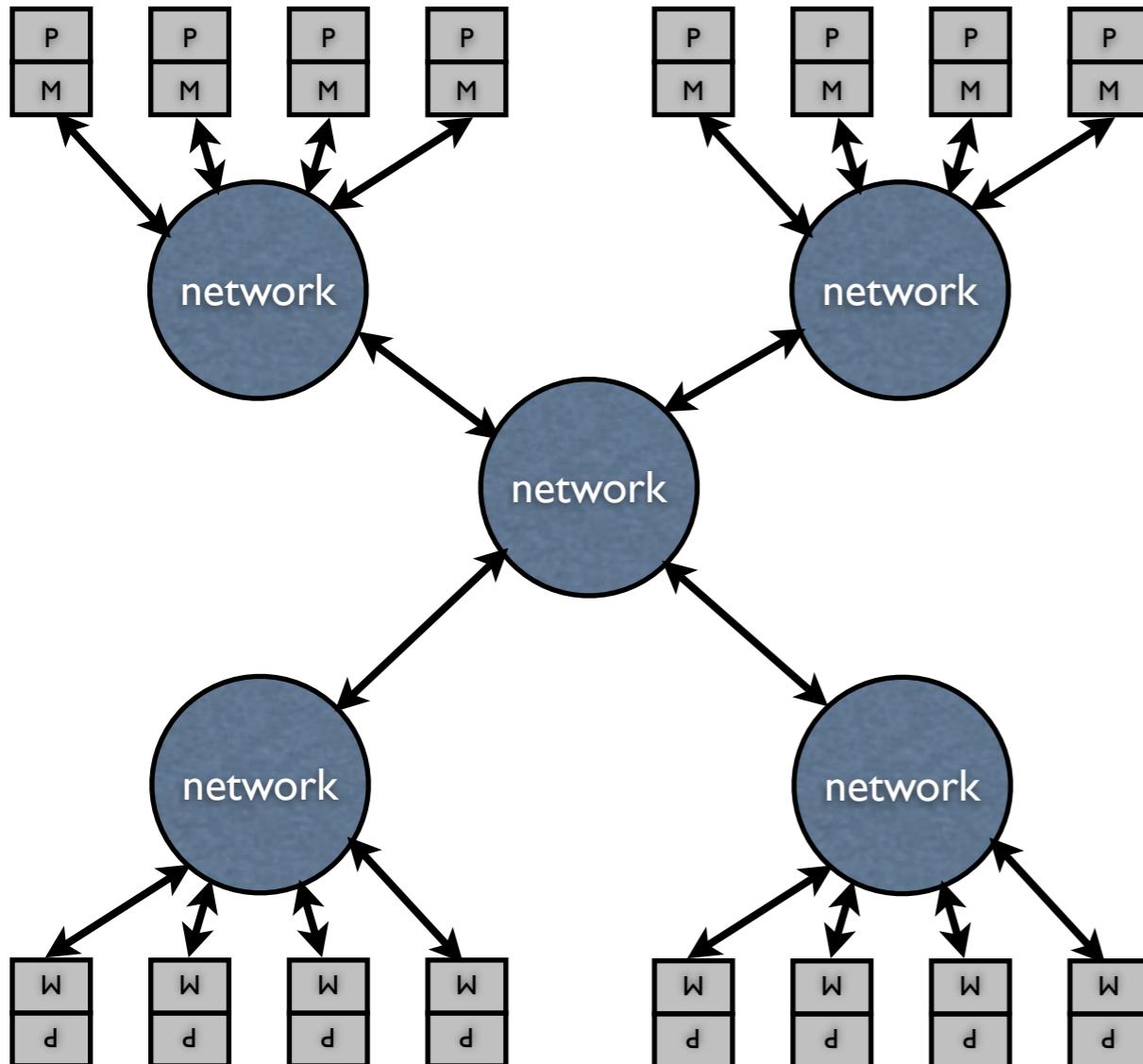


Message Passing Model



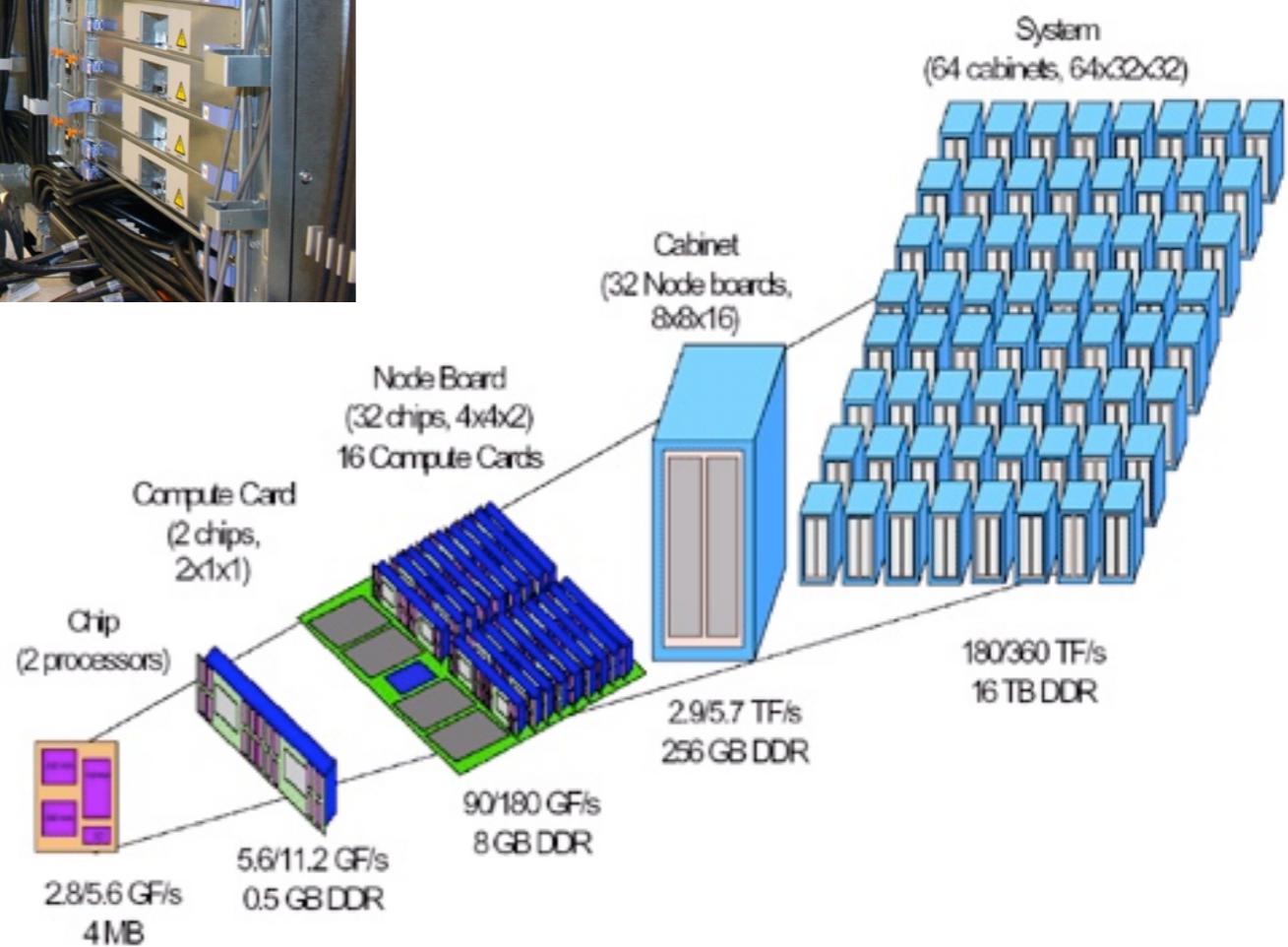
- This drawing implies that all processor are equidistant from one another
- This is often not the case -- the network topology and multicores make some processors closer than others
- programmers have to exploit this manually

Message Passing Model



- In reality, processes run on cores, and are closer to other processes on the same processor
- Across processors, some can be reached via a single hop on the network, others require multiple hops
- Not a big issue on small (several hundred processors), but it needs to be considered on large machines.

131,072 cores BG/L



Why use message passing

- Allows control over data layout, locality and communication -- very important on large machines
- Portable across all machines *including shared memory machines* -- it's a universal parallel programming model
- Easier to write deterministic programs
 - simplifies debugging
 - easier to understand programs
- Style needed for efficient messages can lead to better performance than shared memory programs, even on shared memory systems.

Why not use it?

- All or nothing program development - generally need to make the entire program parallel to make any part parallel
- Information needed for messages low-level and hard to program
- Subtle bugs in message passing code can lead to performance problems and deadlock
- Message passing code disrupts the flow of algorithms

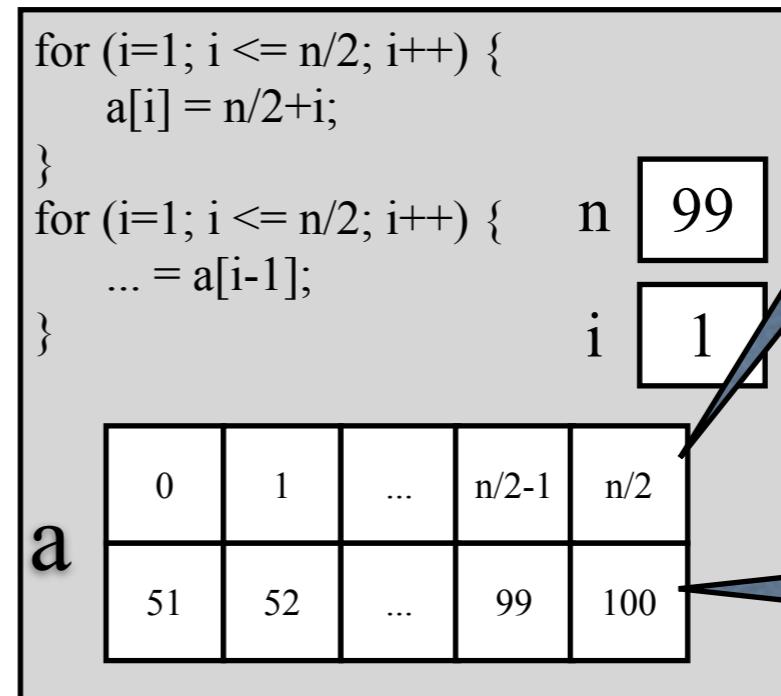
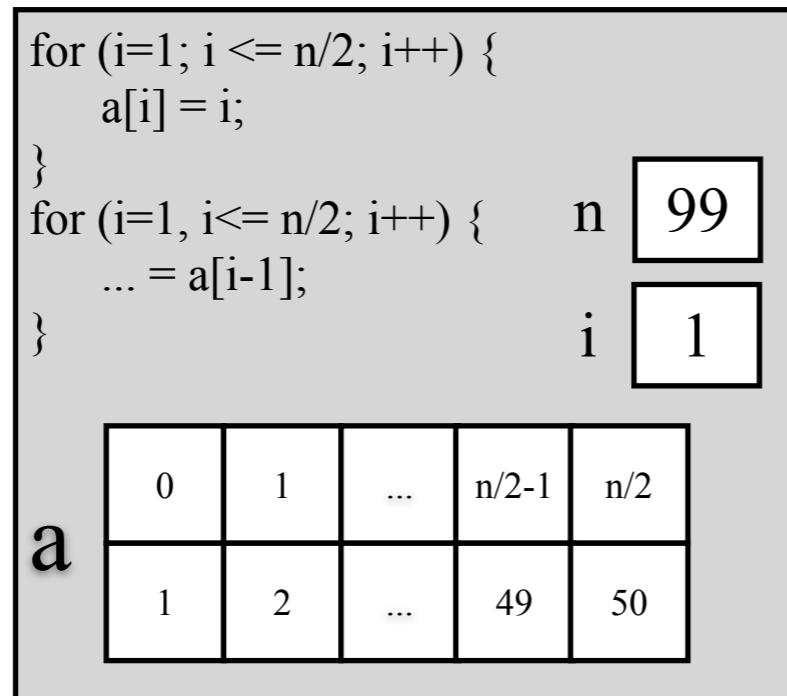
SPMD execution

- Single Program Multiple Data
- Multiple copies of the same program operating on different parts of the data (typically different sections of an array)
- Each program copy executes in a process
- Different processes can execute different paths through the program

SPMD execution

```
for (i=1; i <= n; i++) {  
    a[i] = i + 1;  
}  
for (i=1, i <= n; i++) {  
    ... = a[i-1];  
}
```

Note fixed loop bounds, subscripts and entries in "a" in figure below.



local index

global index

Work done by processes

- Each process has a unique rank or process id (often called *pid* in programs) that is set when program starts
- Is not changed during the execution of the program (however, see Naik, Moreira, et al. IBM DRMS project if you are really interested in this.)
- Each process has a unique identifier (often called *pid*) that is known to the program
- Typical program pattern is *compute* \Rightarrow *communicate* \Rightarrow *compute* ... \Rightarrow ... \Rightarrow *communicate*

Radix sort

- Radix sort works well to sort lists of numbers
- Will assume integers have values from 0 to 65,535
- Have $N \gg 65,535$ numbers to sort

Sequential program

```
for (i=0; i < 65535; i++) {  
    sorted[i] = 0;  
}
```

Want to convert to
SPMD message
passing code

```
for (i=0; i < n; i++) {  
    sorted[data[i]]++;  
}
```

Note that data input not
shown -- this can require
some thought

```
for (i=0; i<65535; i++) {  
    for (j=0; j < sort[i]; j++) {  
        fprintf("%i\n", i);  
    }  
}
```

Data often spread across
multiple files to accommodate
parallel I/O on large problems

SPMDizing the program

P0

```
data[0:N/4-1]  
i, j  
sorted[0:65353]
```

P1

```
data[N/4:2*N/4-1]  
i, j  
sorted[0:65353]
```

P3

```
data[2*N/4:3*N/4-1]  
i, j  
sorted[0:65353]
```

P2

```
data[3*N/4:N-1]  
i, j  
sorted[0:65353]
```

all processors execute this (replicated execution)

```
for (i=0; i < 65535; i++) {  
    sorted[i] = 0;  
}
```

each processor executes $N/4$ iterations (assume $N \bmod 4 = 0$)

```
for (i=0; i < N/4; i++) {  
    sorted[data[i]]++;  
}
```

this becomes a sum reduction over the sorted arrays on each processor,
i.e. communication. This code does not show that.

```
for (i=0; i<65535; i++) {  
    for (j=0; j < sort[i]; j++) {  
        fprintf("%i\n", i);  
    }  
}
```

global indices
shown, local is
[N/4:2*N/4-1]

Data management

P0

data[0:N/4-1]
i, j
sorted[0:65353]

P1

data[N/4:2*N/4-1]
i, j
sorted[0:65353]

P3

data[2*N/4:3*N/4-1]
i, j
sorted[0:65353]

P2

data[3*N/4:N-1]
i, j
sorted[0:65353]

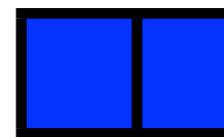
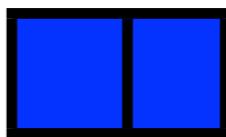
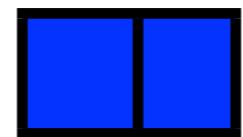
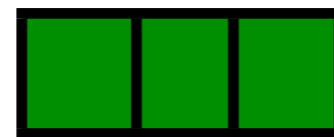
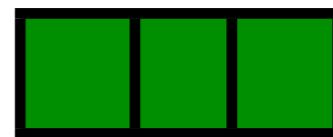
- All declared variables exist within each process
- There is a *global* and *local* logical index space for arrays
 - *globally*, data has N elements $0:N-1$
 - *locally*, each process has $N/4$ elements numbered $0:N/4-1$ (if $N \bmod 4 == 0$, otherwise $\lceil N/4 \rceil$ or $\lfloor N/4 \rfloor$ elements per processor)
 - The concatenation of the local partitions of data arrays forms the global array data
- The array data is *block* distributed over the processors

Data bounds for block

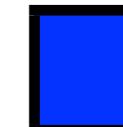
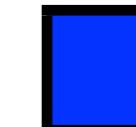
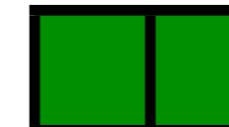
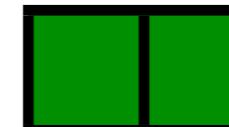
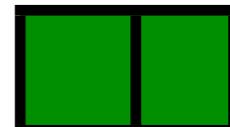
- Two “obvious” ways to compute
- Let n be the array size, P the number processors

First method

- Let P be the number of processes, n the number of array elements, $0 \leq p \leq P-1$ is a process id
- $r = n \bmod P$, $r = 0$, all blocks are the same size, otherwise, first r blocks have $\lceil n/P \rceil$ elements, last $n-r$ have $\lfloor n/P \rfloor$ elements
- First element on a process p is $p \lfloor n/P \rfloor + \min(p, r)$
- Last element on process p is $(p+1) \lfloor n/P \rfloor + \min(p+1, r) - 1$
- process with element i is $\min(\lfloor i/(\lfloor n/P \rfloor + 1) \rfloor, \lfloor (i-r)/\lfloor n/P \rfloor \rfloor)$
- Example -- 12 elements over 5 processors, $2 = 12 \bmod 5$



- Example -- 12 elements over 7 processors, $5 = 12 \bmod 7$

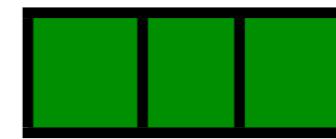
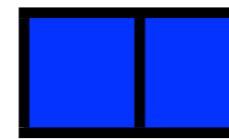
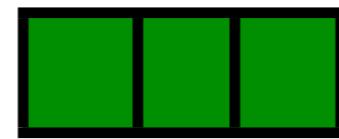
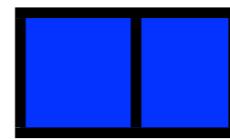
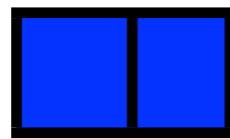


Second method

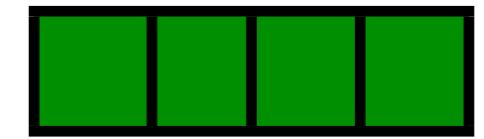
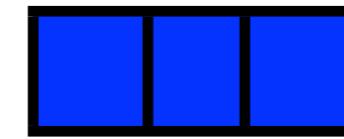
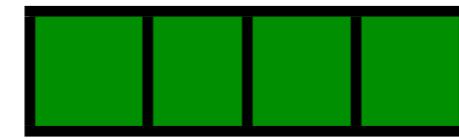
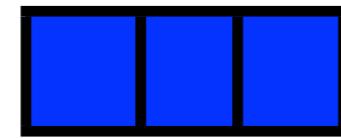
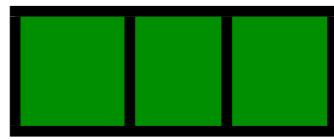
- First element controlled by process p is $\lfloor p \ n/P \rfloor$ (first element and first process id p is 0)
- Last element controlled by process p is one less than the first element controlled by process $p+1$

$$\lfloor (p+1) \ n/P \rfloor - 1$$

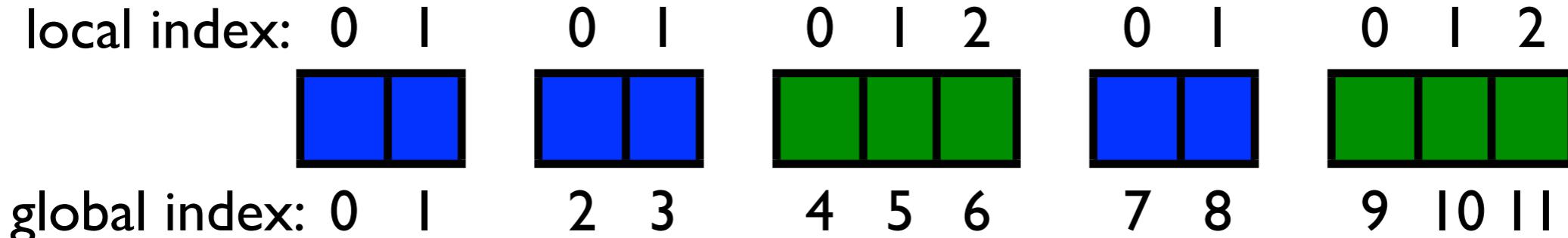
- Process controlling element i is $\lfloor (P(i+1)-1)/n \rfloor$
- Example -- 12 elements over 5 processors, $r = 2 = 12 \bmod 5$



- Example -- 17 elements over 5 processors, $r = 2 = 17 \bmod 5$

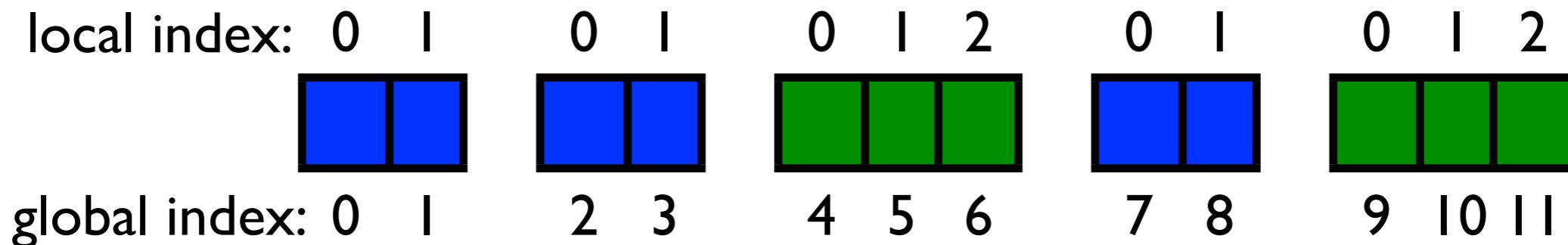


Global vs local indices



- Each part of an array within a process *must* be indexed as a local element of that array using the *local index*.
- Logically, each local element is a part of the global array, and within the problem domain has a *global index*
- It is the MPI *programmer's responsibility* (that means **you**) to maintain that mapping.

Use macros to access bounds



- Macros or functions can be used to compute these.
- Block lower bound: $LB(p, P, n) = (p * n / P)$
- Block upper bound: $UB(p, P, n) = LB(p + 1, P, n) - 1$
- Block size: $LB(p + 1, P, n) - LB(p, P, n)$
- Block owner: $Owner(i, P, N) = (P * (i + 1) - 1) / n$

Comparison of the two methods

Operations	First Method	Second Method
Low index	4	2
High index	6	4
Owner	7	4

Assumes floor is free (as it is with integer division although integer division itself may be expensive)

The *cyclic* distribution

P0	P1	P3	P2
data[0:N:4] i, j sorted[0:65353]	data[1:n:4] i, j sorted[0:65353]	data[2:N:4] i, j sorted[0:65353]	data[3:N:4] i, j sorted[0:65353]

- Let A be an array with N elements.
- Let the array be *cyclically distributed* over P processes
- Process p gets elements $p, p+P, p+2*P, p+3*P, \dots$
- In the above
 - process 0 gets elements 0, 4, 8, 12, ... of data
 - process 1 gets elements 1, 5, 9, 13, ... of data
 - process 2 gets elements 2, 6, 10, 14, ... of data
 - process 3 gets elements 3, 7, 11, 15, ... of data

The *block-cyclic* distribution

- Let A be an array with N elements
- Let the array be *block-cyclically distributed* over P processes, with blocksize B
- Block b , $b = 0 \dots$, on process p gets elements
$$b*B*P+p*B : b*B*P + (p+1)*B - 1$$
 elements
- With $P=4, B=3$
 - process 0 gets elements $[0:2], [12:14], [24:26]$ of data
 - process 1 gets elements $[3:5], [15:17], [27:29]$ of data
 - process 2 gets elements $[6:8], [18:20], [30:32]$ of data
 - process 3 gets elements $[9:11], [21:23], [33:35]$ of data

Converting the program to MPI: System initialization

P0

```
data[0:N/4-1]  
i, j  
sorted[0:65353]
```

P1

```
data[N/4:2*N/4-1]  
i, j  
sorted[0:65353]
```

P2

```
data[2*N/4:3*N/4-1]  
i, j  
sorted[0:65353]
```

P3

```
data[3*N/4:N-1]  
i, j  
sorted[0:65353]
```

```
#include <mpi.h> /* MPI library prototypes, etc. */  
#include <stdio.h>  
// all processors execute this (replicated execution)  
int main(int argc, char * argv[ ]) {  
    int pid; /* MPI process ID)  
    int numP; /* number of MPI processes */  
    int N;  
    extractArgv(&N, argv); // get N from the arg vector  
    int sorted[65536]; int data[N/4];  
    MPI_INIT(&argc, &argv);  
    for (i=0; i < 65535; i++) {  
        sorted[i] = 0;  
    }  
}
```

Converting the program to MPI: System initialization

P0

```
data[0:N/4-1]  
i, j  
sorted[0:65353]
```

P1

```
data[N/4:2*N/4-1]  
i, j  
sorted[0:65353]
```

P2

```
data[2*N/4:3*N/4-1]  
i, j  
sorted[0:65353]
```

P3

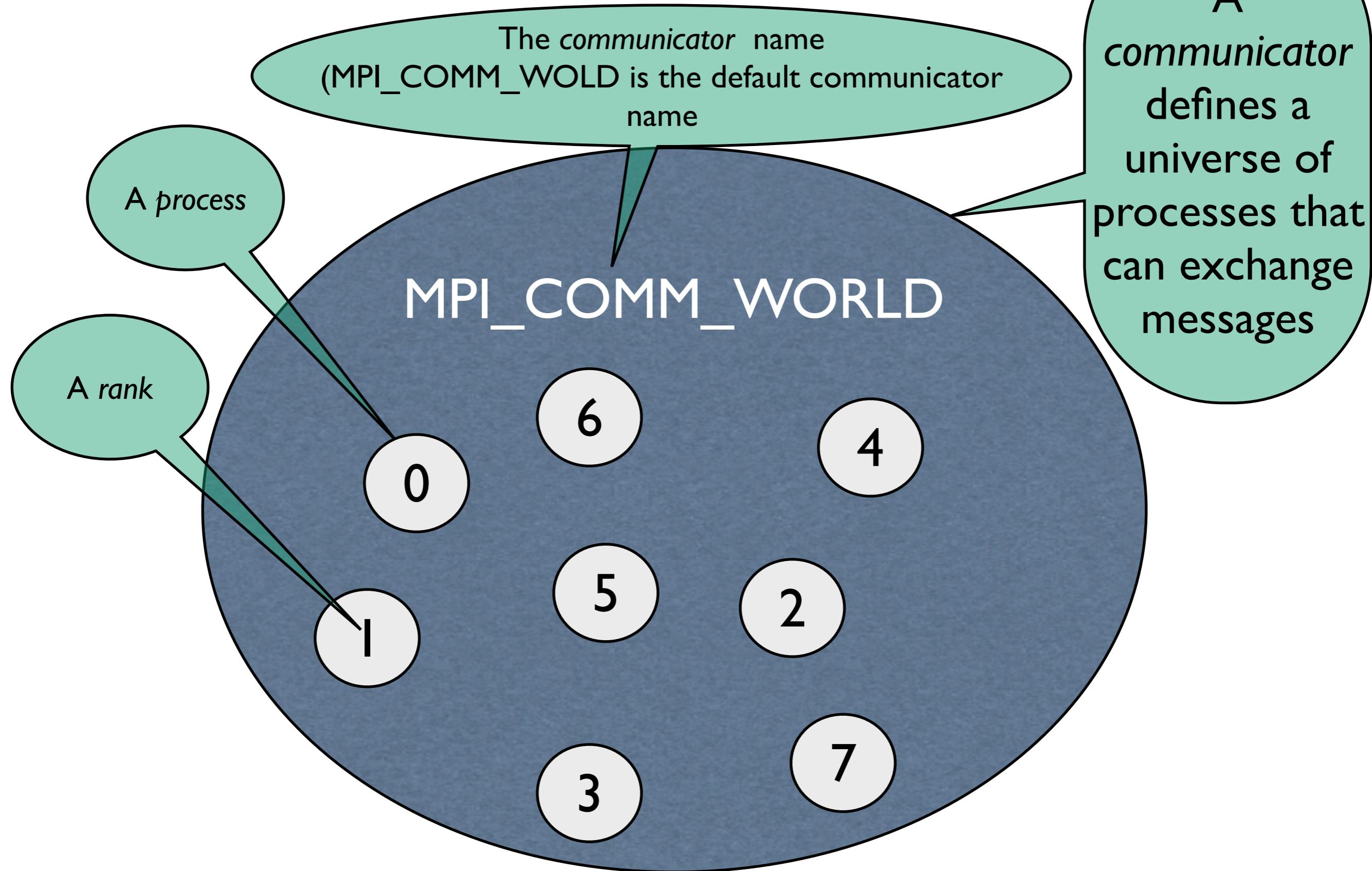
```
data[3*N/4:N-1]  
i, j  
sorted[0:65353]
```

```
#include <mpi.h> /* MPI library prototypes, etc. */  
#include <stdio.h>  
// all processors execute this (replicated execution)  
int main(int argc, char * argv[ ]) {  
    ...  
    int rc = MPI_INIT(&argc, &argv); // can be null  
    if (rc != MPI_SUCCESS) { // is MPI_ERR_OTHER  
        printf("Error in MPI INIT");  
        halt(rc);  
    }
```

MPI_INIT

- Initialize the MPI runtime
- Does not have to be the first executable statement in the program, but it *must be the first MPI call made*
- Initializes the default MPI *communicator* (`MPI_COMM_WORLD` which includes all processes)
- Reads standard files and environment variables to get information about the system the program will execute on
 - e.g. what machines executes the program?

The MPI environment



Converting the program to MPI

P0

```
data[0:N/4-1]  
i, j  
sorted[0:65353]
```

P1

```
data[N/4:2*N/4-1]  
i, j  
sorted[0:65353]
```

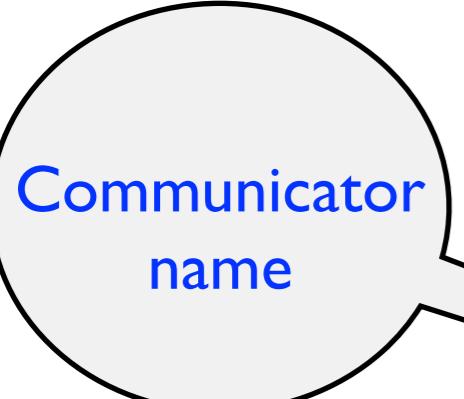
P2

```
data[2*N/4:3*N/4-1]  
i, j  
sorted[0:65353]
```

P3

```
data[3*N/4:N-1]  
i, j  
sorted[0:65353]
```

```
#include <mpi.h> /* MPI library prototypes, etc. */  
#include <stdio.h>  
/* all processors execute this (replicated execution)  
int main(int argc, char * argv[ ]) {  
    int pid; /* MPI process ID)  
    int numP; /* number of MPI processes */  
    int N;  
    extractArgv(&N, argv);  
    int sorted[65536]; int data[N/4];  
    MPI_INIT(&argc, &argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &numP);  
    for (i=0; i < 65535; i++) {  
        sorted[i] = 0;  
    }  
}
```



cheat!
should
malloc

get
number of
processors

Converting the program to MPI

P0

```
data[0:N/4-1]  
i, j  
sorted[0:65353]
```

P1

```
data[N/4:2*N/4-1]  
i, j  
sorted[0:65353]
```

P2

```
data[2*N/4:3*N/4-1]  
i, j  
sorted[0:65353]
```

P3

```
data[3*N/4:N-1]  
i, j  
sorted[0:65353]
```

```
#include <mpi.h> /* MPI library prototypes, etc. */  
#include <stdio.h>  
/* all processors execute this (replicated execution)  
int main(int argc, char * argv[ ]) {  
    int pid; /* MPI process ID */  
    int numP; /* number of MPI processes */  
    int N;  
    extractArgv(&N, argv);  
    int sorted[65536]; int data[*N/4]; MPI_INIT(&argc, &argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &numP);  
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);  
    for (i=0; i < 65535; i++) {  
        sorted[i] = 0;  
    } }
```

Communicator
name

arg to get
rank (i.e. pid) of
this processor

Converting the program to MPI

P0

```
data[0:N/4-1]  
i, j  
sorted[0:65353]
```

P1

```
data[N/4:2*N/4-1]  
i, j  
sorted[0:65353]
```

P2

```
data[2*N/4:3*N/4-1]  
i, j  
sorted[0:65353]
```

P3

```
data[3*N/4:N-1]  
i, j  
sorted[0:65353]
```

```
#include <mpi.h> /* MPI library prototypes, etc. */  
#include <stdio.h>  
/* all processors execute this (replicated execution)  
int main(int argc, char * argv[ ]) {  
    int pid; /* MPI process ID */  
    int numP; /* number of MPI processes */  
    int N;  
    extractArgv(&N, argv);  
    int sorted[65536]; int data[*N/4]; MPI_INIT(&argc, &argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &numP);  
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);  
    for (i=0; i < 65535; i++) {  
        sorted[i] = 0;  
    }  
    MPI_Finalize();  
}
```

The *last* MPI
function called

MPI_Finalize frees
system resources
associated with MPI

Time to do something useful

P0

```
data[0:N/4-I]  
i, j  
sorted[0:65353]
```

P1

```
data[N/4:2*N/4-I]  
i, j  
sorted[0:65353]
```

P2

```
data[2*N/4:3*N/4-I]  
i, j  
sorted[0:65353]
```

P3

```
data[3*N/4:N-I]  
i, j  
sorted[0:65353]
```

```
#include <mpi.h> /* MPI library prototypes, etc. */  
#include <stdio.h>  
/* all processors execute this (replicated execution)  
int main(int argc, char * argv[ ]) {  
    int pid; /* MPI process ID */  
    int numP; /* number of MPI processes */  
    int N;  
    extractArgv(&N, argv);  
    int sorted[65536]; int data[*N/4];  
    MPI_INIT(&argc, &argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &numP);  
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);  
    for (i=0; i < 65535; i++) {  
        sorted[i] = 0;  
    }  
    sort(data, sort, pid, numP);  
    MPI_Finalize( );
```

)

The serial code

P0

```
data[0:N/4-1]  
i, j  
sorted[0:65353]
```

P1

```
data[N/4:2*N/4-1]  
i, j  
sorted[0:65353]
```

P2

```
data[2*N/4:3*N/4-1]  
i, j  
sorted[0:65353]
```

P3

```
data[3*N/4:N-1]  
i, j  
sorted[0:65353]
```

```
void sort (sort[ ], data[ ], int pid, int numP) {  
    for (i=0; i < N/4; i++) { // parallelism comes from here!  
        sorted[data[i]]++;  
    }  
    // sorted results available here ...  
}
```

If above is done in parallel, need to get results from all processes before printing them

```
for (i=0; i<65535; i++) {  
    for (j=0; j < sort[i]; j++) {  
        fprintf("%i\n", i);  
    }  
}
```

MPI_Reduce(...)

- Does a reduction like the reduce clause in OpenMP, only it uses messages.

address
of the first
element to
be reduced

address of the first result
element

number
reduction
elements/
results

type of data
being reduced

```
MPI_Reduce(void *opnd, void *result, int count, MPI_Datatype type,  
          MPI_Operator op, int root, MPI_Comm comm);
```

reduction
operation

rank of the
process getting
the result

the
communicator over
which the reduction is
performed

MPI_Datatype

- Defined as constants in the mpi.h header file
- Types supported are

MPI_CHAR

MPI_DOUBLE

MPI_FLOAT

MPI_INT

MPI_LONG

MPI_LONG_DOUBLE

MPI_SHORT

MPI_UNSIGNED_CHAR

MPI_UNSIGNED

MPI_UNSIGNED_LONG

MPI_UNSIGNED_SHORT

MPI_Op

- Defined as constants in the mpi.h header file
- Types supported are

MPI_BAND

MPI_EXOR

MPI_LAND

MPI_LXOR

MPI_MAXLOC

MPI_MINLOC

MPI_SUM

MPI_BOR

MPI_BXOR

MPI_LOR

MPI_MAX

MPI_MIN

MPI_PROD

MPI_Reduce(...)

- Does a reduction like the reduce clause in OpenMP, only it uses messages.

use *result
as in and out
buffer

address of the first
result element

number
reduction
elements/
results

type of data
being reduced

```
MPI_Reduce(MPI_IN_PLACE, void *result, int count, MPI_Datatype type,  
MPI_Operator op, int root, MPI_Comm comm);
```

reduction
operation

rank of
process getting
the result

the
communicator

Example of reduction

sorted, p=0

3	5	2	9	8	11	20	4
---	---	---	---	---	----	----	---

sorted, p=1

8	3	6	8	38	5	27	6
---	---	---	---	----	---	----	---

sorted, p=2

1	0	9	0	2	1	2	40
---	---	---	---	---	---	---	----

sorted, p=3

13	15	12	19	18	21	42	3
----	----	----	----	----	----	----	---

sorted, p=0

25	23	39	36	64	38	91	53
----	----	----	----	----	----	----	----

```
MPI_Reduce(MPI_IN_PLACE, sorted, 8, MPI_INT,  
MPI_SUM, 0, MPI_COMM_WORLD);
```

Add the reduction

P0

data[0:N/4-1]
i, j
sorted[0:65353]

P1

data[N/4:2*N/4-1]
i, j
sorted[0:65353]

P2

data[2*N/4:3*N/4-1]
i, j
sorted[0:65353]

P3

data[3*N/4:N-1]
i, j
sorted[0:65353]

```
void sort (sort[ ], data[ ], int pid, int numP) {  
    for (i=0; i < N; i++) {  
        sorted[data[i]]++;  
    }  
    // can merge all of the “sorted” arrays here  
    if (pid == 0) {  
        MPI_Reduce(MPI_IN_PLACE, sorted, 8, MPI_INT,  
                   MPI_SUM, 0, MPI_COMM_WORLD);  
    } else {  
        MPI_Reduce(sorted, (void *) null, 8, MPI_INT,  
                   MPI_SUM, 0, MPI_COMM_WORLD);  
    }
```

Alternatively, could allocate a buffer for final sorted result. Buffer would be the same size as sorted.

Notes on Reduce

- There is a result for each element of the source array across all processors
- The result ends up on only one processor (allreduce sends the result to all processors)

Determining program performance

- MPI_Barrier - barrier synchronization
- MPI_Wtick - returns the clock resolution in seconds
- MPI_Wtime - current time

```
int main(int argc, char * argv[ ]) {  
    ...  
    double elapsed;  
    int pid; /* MPI process ID)  
    int numP; /* number of MPI processes */  
    int N;  
    extractArgv(&N, argv);  
    for (i=0; i < 65535; i++) {  
        sorted[i] = 0;  
    }  
    MPI_Barrier( );  
    elapsed = -MPI_Wtime();  
    sort(data, sort, pid, numP);  
    elapsed += MPI_Wtime();  
    if (pid == 0) printSort(final);  
    MPI_Finalize( );  
}
```

Determining program performance

```
int main(int argc, char * argv[ ]) {  
    ...  
    double elapsed;  
    int pid; /* MPI process ID)  
    int numP; /* number of MPI processes */  
    int N;  
    extractArgv(&N, argv); for (i=0; i < 65535; i++) {  
        sorted[i] = 0;  
    }  
    MPI_Barrier();  
    elapsed = -MPI_Wtime();  
    sort(data, sort, pid, numP);  
    elapsed += MPI_Wtime();  
    if (pid == 0) printSort(final, elapsed);  
    MPI_Finalize();  
}
```

Holds the elapsed time

wait for all processors to finish initialization

negative of start time

plus finish time gives elapsed time

Wtick() returns a *double* that holds the number of seconds between clock ticks - 10^{-3} is milliseconds

`Wtick()` gives the clock resolution

`MPI_WTick` returns the resolution of `MPI_WTime` in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks.

```
double tick = MPI_WTick();
```

Thus, a millisecond resolution timer will return 10^{-3}

Sieve of Erosthenes

- Look at block allocations
- Performance tuning
- MPI_Bcast function

Finding prime numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

To find primes

- 1.start with two, mark all multiples
- 2.find the next unmarked u -- it is a prime
- 3.mark all multiples of u between k^2 and n until $k^2 > n$
- 4.repeat 2 & 3 until finished

Finding prime numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

To find primes

3 is prime

mark all multiples of 3 > 9

Finding prime numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

To find primes

5 is prime

mark all multiples of 5 > 25

Finding prime numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

To find primes

7 is prime

mark all multiples of 7 >
49

Finding prime numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

To find primes

11 is prime

mark all multiples of 11
> 121

Finding prime numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

To find primes

~~1, 2, 3, 5, 7, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89 and 97~~ are prime.

1 is not prime by definition

Want to parallelize this

- Because we are message passing, obvious thing to look at it domain decomposition, i.e. how can we break up the domain being operated on over multiple processors
 - partition data across processors
 - associate tasks with data
- In general, try to find fundamental operations and associate them with data

What is (are) the fundamental operation(s)?

- Marking of the multiples of the last prime found
- if v a multiple of k
then $v \bmod k == 0$
- *min-reduction* to find the next prime (i.e. smallest unmarked value)
across all processes
- broadcast the value to all tasks

```
forall (v = k; v < n+1; v++) {  
    if (v mod k != 0) a[v] = 1;  
}
```

To make this efficient

- Combine as many tasks as possible onto a single process
- Make the amount of work done by each process similar, i.e. *load balance*
- Make the communication between tasks efficient

Combining work/ partitioning data

- Because processes work on data that they own, the two problems are tightly inter-related.
- Cyclic distributions have the property that for all elements i on some process p , $i \bmod p = c$ where c is some integer value
 - Although cyclic usually gives better load balance, it doesn't in this case
 - Lesson -- don't apply rules-of-thumb blindly
- Block, in this case, gives a better load balance
 - computation of indices will be harder

Interplay of decomposition and implementation

- Decomposition affects how we design the implementation
- More abstract issues of parallelization can affect the implementation
- In the current algorithm, let Φ be the highest possible prime
- At most, only first $\sqrt{\Phi}$ values may be used to mark off (sieve) other primes
- if P processes, n elements to a process, then if

$$n/P > \sqrt{\Phi}$$

only elements in $p=0$ will be used to sieve. This means we only need to look for lowest unmarked elements in $p=0$ and only $p=0$ needs to send this out. **Saves a min-reduce for every prime.**

Use of block partitioning affects marking

- Can mark $j, j+k, j+2k, \dots$ where j is the first prime in the block
- Using the parallel method described in earlier psuedo-code, would need to use an expensive mod

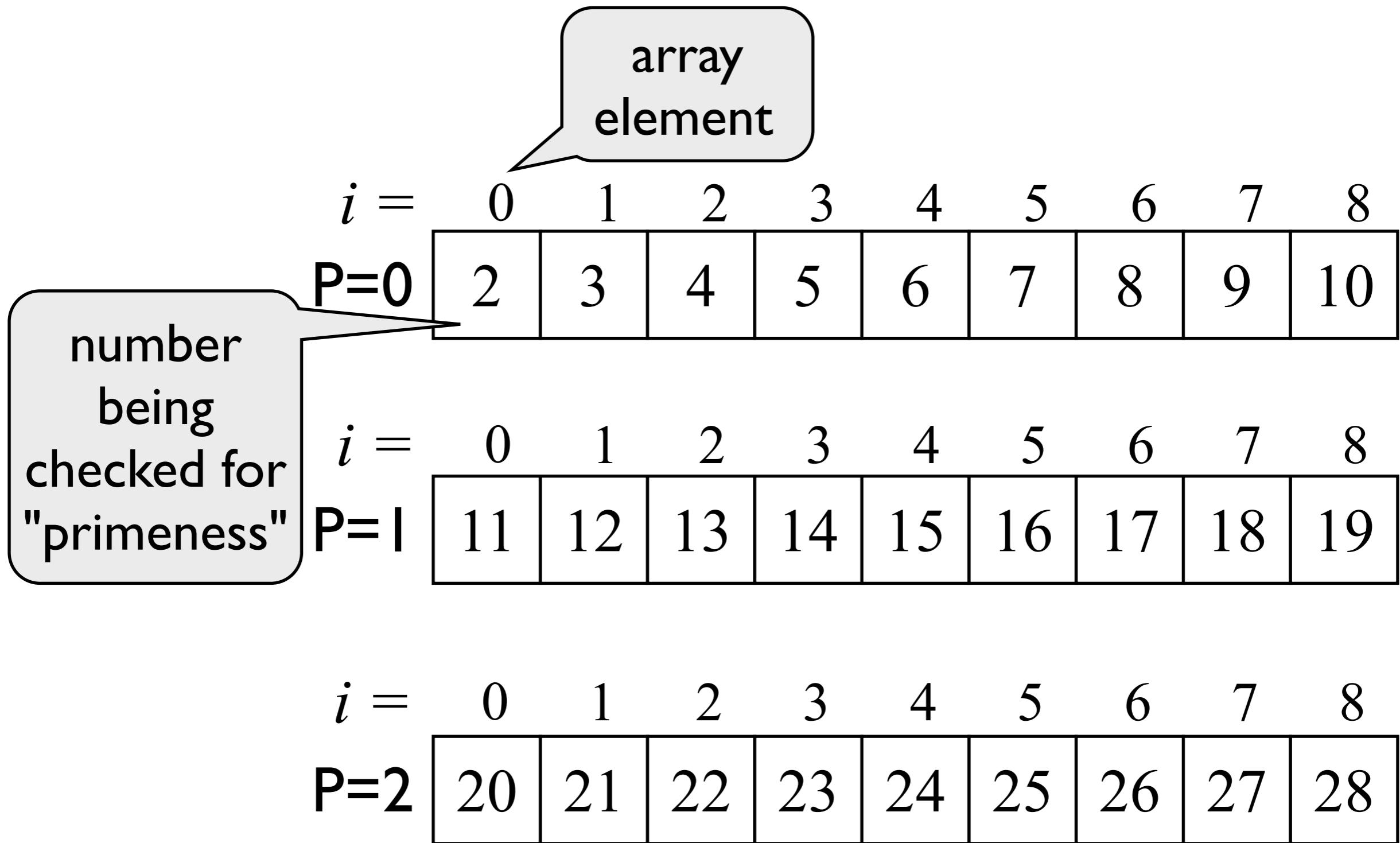
for all e in the block

if $e \bmod k = 0$, mark e

Sketch of the algorithm

1. Create list of possible primes
2. On each process, set $k = 2$
3. Repeat
 - 3.1. On each process, mark all multiples of k
 - 3.2. On process 0, find smallest unmarked number u , set $k=u$
 - 3.3. On process 0, *broadcast* k to all processes
4. Until $k^2 > \Phi$ (the highest possible prime)
5. Perform a sum reduction to determine the number of primes

Data layout, primes up to 28



Algorithm 1 /4

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include "MyMPI.h"
#define MIN(a,b) ((a)<(b)?(a):(b))
```

standard
stuff

bounds
macros, etc.

```
int main (int argc, char *argv[])
{
    ...
    MPI_Init (&argc, &argv);
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    if (argc != 2) {
        if (!id) printf ("Command line: %s <m>\n", argv[0]);
        MPI_Finalize(); exit (1);
    }
}
```

setup,
check args,
etc.

Algorithm, 2/4

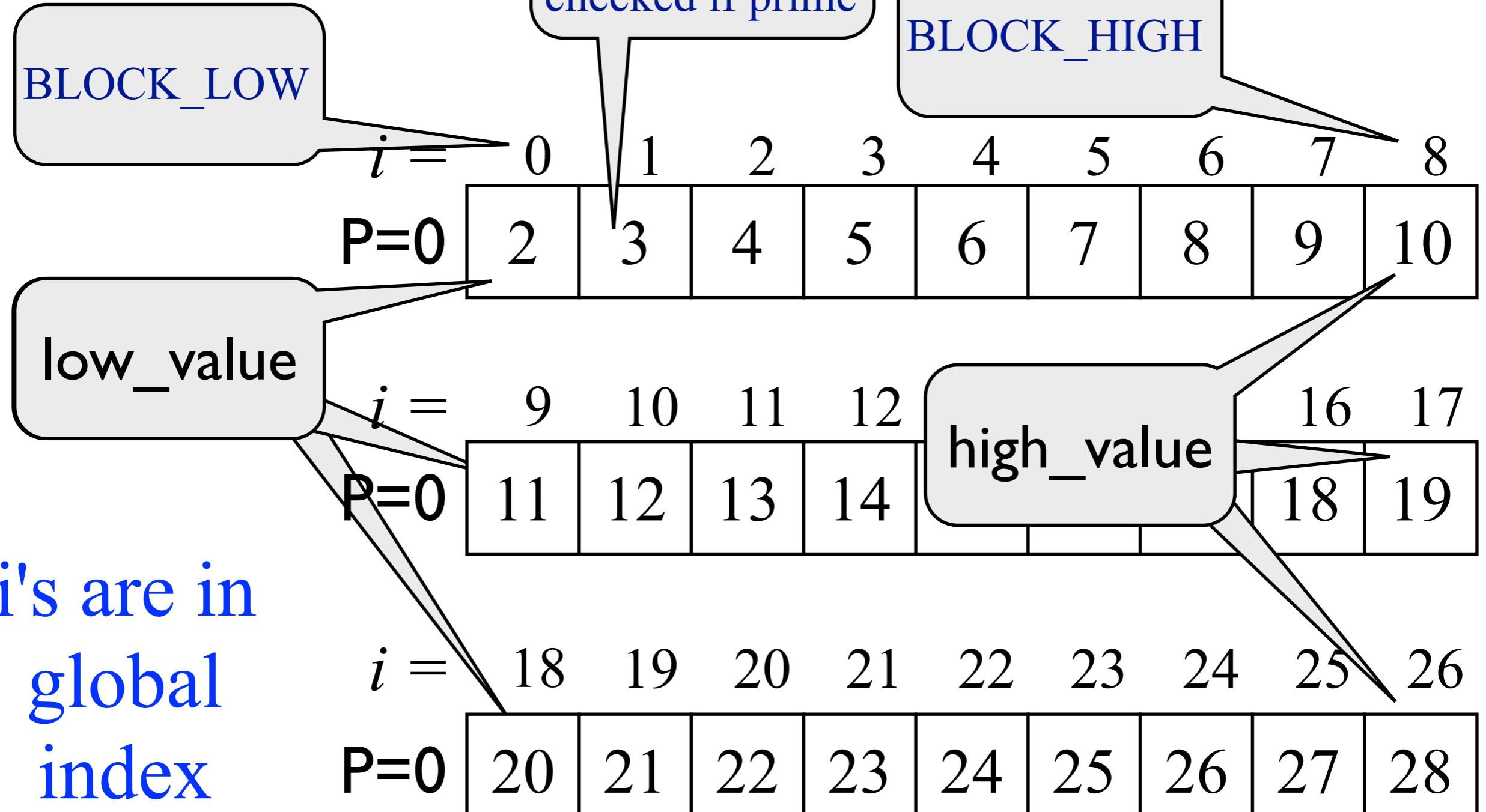
```
n = atoi(argv[1]);
low_value = 2 + BLOCK_LOW(id,p,n-1);
high_value = 2 + BLOCK_HIGH(id,p,n-1);
size = BLOCK_SIZE(id,p,n-1);
proc0_size = (n-1)/p;
if ((2 + proc0_size) < (int) sqrt((double) n)) {
    if (!id) printf ("Too many processes\n");
    MPI_Finalize();
    exit (1);
}
marked = (char *) malloc (size);
if (marked == NULL) {
    printf ("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit (1);
}
```

Get min and max possible prime on p in global space

Figure out if too many processes for $\sqrt{\Phi}$ candidates on $p=0$

allocate array to use to mark primes

values for P=0, similar for other processes



i's are in
global
index
space

Algorithm 3/4

```
for (i = 0; i < size; i++) marked[i] = 0; // initialize marking array
if (!id) index = 0; // p=0 action, find first prime
prime = 2;
do { // prime = 2 first time through, sent by bcast on later iterations
    if (prime * prime > low_value) // find first value to mark
        first = prime * prime - low_value; // first item in this block
    else {
        if (!(low_value % prime)) first = 0; // first element divisible by prime
        else first = prime - (low_value % prime);
    }
    for (i = first; i < size; i += prime) marked[i] = 1; // mark every kth item
    if (!id) { // p=0 action, find next prime by finding unmarked element
        while (marked[++index]);
        prime = index + 2;
    }
    MPI_Bcast (&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
} while (prime * prime <= n);
```

First prime

index = 0
prime = 2

$2 * 2 > \textcolor{red}{2}$ local $i = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8$

$first = 2 * 2 - 2$ P=0 $\boxed{2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10}$

$first = \textcolor{red}{2}$

$2 * 2 > \textcolor{red}{2}$	$first = 2 * 2 - 2$	$first = \textcolor{red}{2}$	local $i = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8$	P=0	$\boxed{2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10}$
------------------------------	---------------------	------------------------------	---	-----	--

$not 2 * 2 > 11$ local $i = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8$

$11 \% 2 == 1$

$first = 2 - (11 \% 2)$ P=0 $\boxed{11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19}$

$first = 1$

$not 2 * 2 > 11$	$11 \% 2 == 1$	$first = 2 - (11 \% 2)$	local $i = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8$	P=0	$\boxed{11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19}$
------------------	----------------	-------------------------	---	-----	--

$not 2 * 2 > 20$ local = $0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8$

$20 \% 2 == 0$

$first = 0$ P=0 $\boxed{20 \quad 21 \quad 22 \quad 23 \quad 24 \quad 25 \quad 26 \quad 27 \quad 28}$

$not 2 * 2 > 20$	$20 \% 2 == 0$	$first = 0$	local = $0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8$	P=0	$\boxed{20 \quad 21 \quad 22 \quad 23 \quad 24 \quad 25 \quad 26 \quad 27 \quad 28}$
------------------	----------------	-------------	---	-----	--

third prime

index = 3
prime = 5

$5 * 5 > 2$	$first = 5 * 5 - 2$	$first = 23$	$local\ i = 0$	1	2	3	4	5	6	7	8	
			P=0	2	3	4	5	6	7	8	9	10

$5 * 5 > 11$	$first = 5 * 5 - 11$	$first = 16$	$local\ i = 0$	1	2	3	4	5	6	7	8	
			P=0	11	12	13	14	15	16	17	18	19

$5 * 5 > 20$	$first = 5 * 5 - 20$	$first = 5$	$local\ i = 0$	1	2	3	4	5	6	7	8	
			P=0	20	21	22	23	24	25	26	27	28

Mark every prime elements starting with *first*

index = 0
prime = 2

$2 * 2 > 4$ local *i* = 0 1 2 3 4 5 6 7 8
first = $2 * 2 - 2$ P=0 2 3 4 5 6 7 8 9 10
first = 2

$2 * 2 > 4$	local <i>i</i> = 0 1 2 3 4 5 6 7 8	first = 2 * 2 - 2	P=0	2 3 4 5 6 7 8 9 10
-------------	------------------------------------	-------------------	-----	--------------------

not $2 * 2 > 11$ local *i* = 0 1 2 3 4 5 6 7 8
11 % 2 == 1
first = $2 - (11 \% 2)$ P=0 11 12 13 14 15 16 17 18 19
first = 1

not $2 * 2 > 11$	local <i>i</i> = 0 1 2 3 4 5 6 7 8	11 % 2 == 1	first = $2 - (11 \% 2)$	P=0 11 12 13 14 15 16 17 18 19
------------------	------------------------------------	-------------	-------------------------	--------------------------------

not $2 * 2 > 20$ local = 0 1 2 3 4 5 6 7 8
20 % 2 == 0
first = 0 P=0 20 21 22 23 24 25 26 27 28

not $2 * 2 > 20$	local = 0 1 2 3 4 5 6 7 8	20 % 2 == 0	first = 0	P=0 20 21 22 23 24 25 26 27 28
------------------	---------------------------	-------------	-----------	--------------------------------

Algorithm 4/4

```
// on each processor count the number of primes, then reduce this total
count = 0;
for (i = 0; i < size; i++)
    if (!marked[i]) count++;
MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
            0, MPI_COMM_WORLD);
elapsed_time += MPI_Wtime();
if (!id) {
    printf ("%d primes are less than or equal to %d\n",
           global_count, n);
    printf ("Total elapsed time: %10.6f\n", elapsed_time);
}
MPI_Finalize ();
return 0;
}
```

Algorithm 4/4

```
// on each processor count the number of primes, then reduce this total
count = 0;
for (i = 0; i < size; i++)
    if (!marked[i]) count++;
MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
            0, MPI_COMM_WORLD);
elapsed_time += MPI_Wtime();
if (!id) {
    printf ("%d primes are less than or equal to %d\n",
           global_count, n);
    printf ("Total elapsed time: %10.6f\n", elapsed_time);
}
MPI_Finalize ();
return 0;
}
```

moved out, not controlled by the *for*

Finding the global count of primes

index = 0

prime = 2

global_count = 1 + 4 + 2

count = 1 P=0

2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	----

count = 4 P=0

11	12	13	14	15	16	17	18	19
----	----	----	----	----	----	----	----	----

count = 2 P=0

20	21	22	23	24	25	26	27	28
----	----	----	----	----	----	----	----	----

Other MPI environment management routines

- `MPI_Abort(comm, errorcode)`
 - Aborts all processors associated with communicator *comm*
- `MPI_Get_processor_name(&name, &length)`
 - MPI version of `gethostname`, but what it returns is implementation dependent. `gethostname` may be more portable.
- `MPI_Initialized(&flag)`
 - Returns true if `MPI_Init` has been called, false otherwise

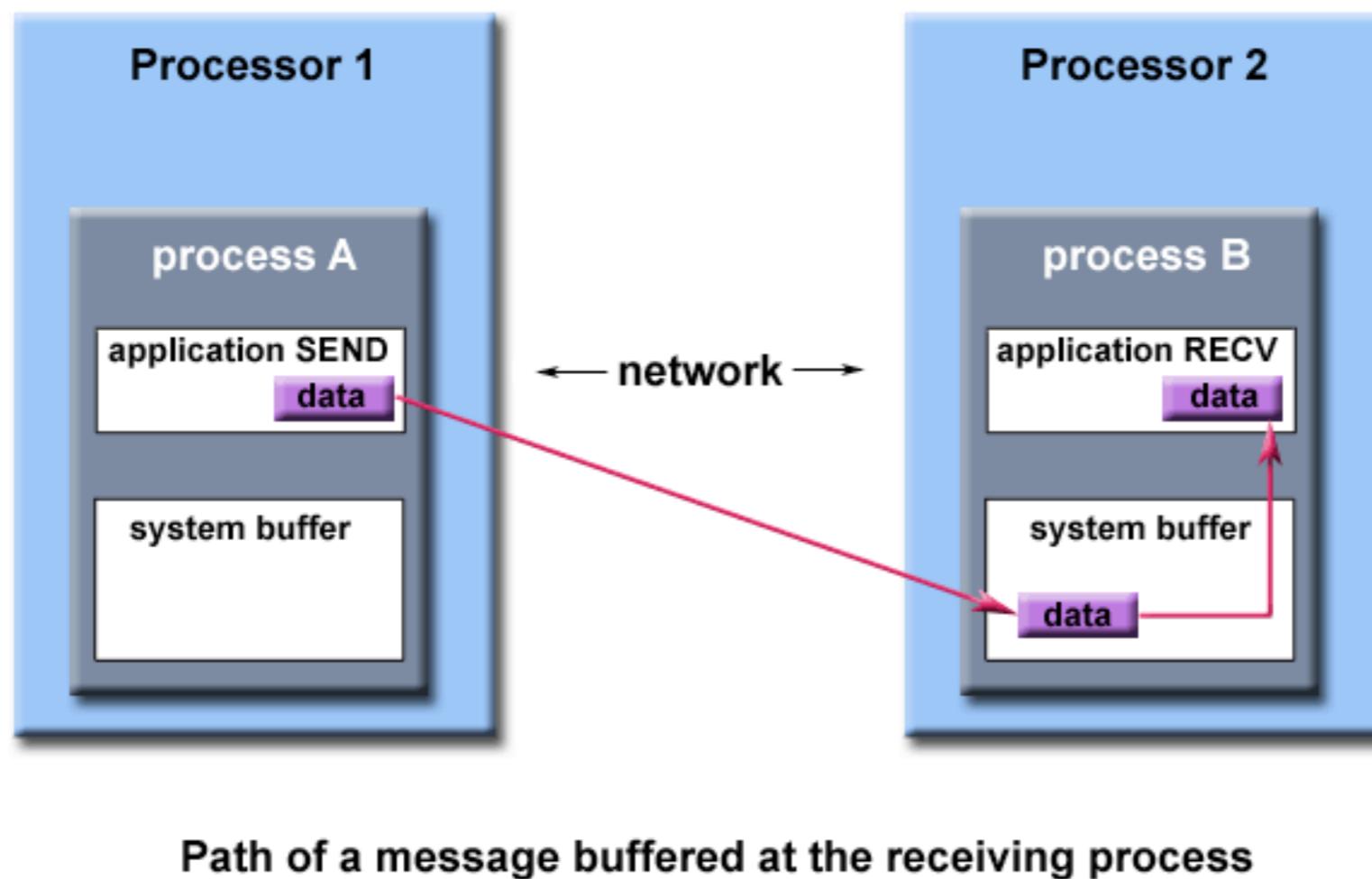
point-to-point communication

- Most MPI communication is between a pair of processors
 - send/receive transmits data from the sending process to the receiving process
- MPI point-to-point communication has many flavors:
 - Synchronous send
 - Blocking send / blocking receive
 - Non-blocking send / non-blocking receive
 - Buffered send
 - Combined send/receive
 - "Ready" send
- All types of sends can be paired with all types of receive

Buffering

What happens when

- A send occurs before the receiving process is ready for the data
- The data from multiple sends arrive at the receiving task which can only accept one at a time



System buffer space

Not part of the standard -- an “implementation detail”

- Managed and controlled by the MPI library
- Finite
- Not well documented -- size maybe a function of install parameters, consequences of running out not well defined
- Both sends and receives can be buffered
- Can help performance by allowing asynchronous send/recvs
- Can hurt performance because of memory copies
- Program variables are called *application buffers* in MPI-speak

Blocking and non-blocking point-to-point communication

Blocking

- Most point-to-point routines have a blocking and non-blocking mode
- A blocking send call returns only when it is safe to modify/reuse the application buffer. Basically the data in the application buffer has been copied into a system buffer or sent.
- Blocking send can be synchronous, which means call to send returns when data is safely delivered to the recv process
- Blocking send can be asynchronous by using a send buffer
- A blocking receive call returns when sent data has arrived and is ready to use

Non-blocking

- Non-blocking send and receive calls behave similarly and return almost immediately.
- Non-blocking operations request the MPI library to perform the operation when it is able. It cannot be predicted when the action will occur.
- You should not modify any application buffer (*program variable*) used in non-blocking communication until the operation has finished. *Wait* calls are available to test this.
- Non-blocking communication allows overlap of computation with communication to achieve higher performance

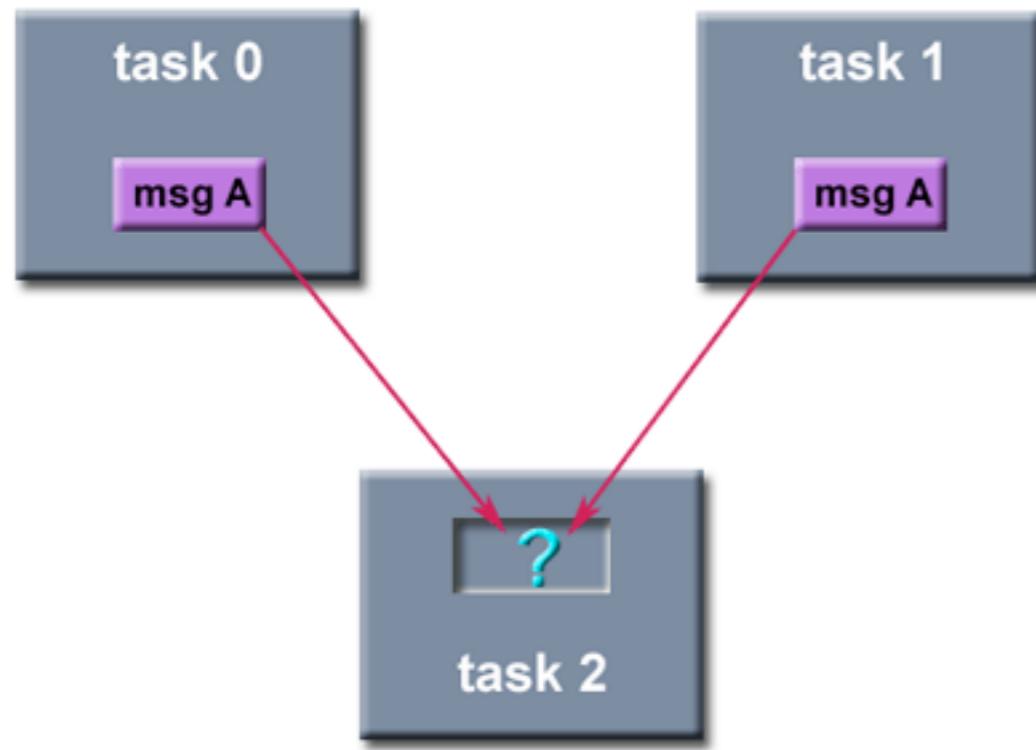
Synchronous and buffered sends and receives

- synchronous send operations block until the receiver begins to receive the data
- buffered send operations allow specification of a buffer used to hold data (this buffer is not the buffer that is the variable being sent or received)
 - allows user to get around system imposed buffer limits
 - for programs needing large buffers, provides portability
 - One buffer/process allowed
 - synchronous and buffered can be *matched*

Ordering of messages and fairness

- Messages received in-order
- If a sender sends two messages, (m_1 and m_2) to the same destination, and both match the same receive, m_1 will be received before m_2 .
- If a receiver posts two receives (r_1 and r_2), and both are looking for the same messages, r_1 will receive a message before r_2 .
- *Operation starvation* is possible
 - $task0$ sends a message to $task2$. $task3$ sends a message that also matches $task2$'s receive. Only one of the sends will complete if the receive is only executed once.
 - It is the programmer's job to ensure this doesn't happen

Operation starvation



Only one of the *sends* will complete.

Networks are generally not deterministic, cannot be predicted whose message will arrive at *task2* first, and which will complete.

Basic sends and receives

I forms are non-blocking

- `MPI_send(buffer, count, type, dest, tag, comm)`
- `MPI_Isend(buffer, count, type, dest, tag, comm, request)`
- `MPI_Recv(buffer, count, type, source, tag, comm, status)`
- `MPI_Irecv(buffer, count, type, source, tag, comm, request)`

Basic sends/recv arguments (*I* forms are non-blocking)

- `MPI_send(buffer, count, type, dest, tag, comm)`
- `MPI_Isend(buffer, count, type, dest, tag, comm, request)`
- `MPI_Recv(buffer, count, type, source, tag, comm, status)`
- `MPI_Irecv(buffer, count, type, source, tag, comm, request)`
- buffer: pointer to the data to be sent or where received (a program variable)
- count: number of data elements (*not bytes!*) to be sent
- type: an MPI_Type (e.g., `MPI_INT`)
- tag: the message type, any unsigned integer 0 - 32767.
- comm: sender and receiver communicator

Basic send/recv arguments

- `MPI_send(buffer, count, type, dest, tag, comm)`
- `MPI_Isend(buffer, count, type, dest, tag, comm, request)`
- `MPI_Recv(buffer, count, type, source, tag, comm, status)`
- `MPI_Irecv(buffer, count, type, source, comm, request)`
- dest: rank of the receiving process
- source: rank of the sending process
- request: for non-blocking operations, a handle to an `MPI_Request` structure for the operation to allow `wait` type commands to know what send/recv they are waiting on
- status: the source and tag of the received message. This is a pointer to the structure of type `MPI_Status` with fields `MPI_SOURCE` and `MPI_TAG`.

Blocking send/recv/etc.

MPI_Send: returns after *buf* is free to be reused. Can use a system buffer but not required, and can be implemented by a system send.

MPI_Recv: returns after the requested data is in *buf*.

MPI_Ssend: blocks sender until the application buffer is free and the receiver process started receiving the message

MPI_Bsend: permits the programmer to allocate buffer space instead of relying on system defaults. Otherwise like *MPI_Send*.

MPI_Buffer_attach (&buffer,size): allocate a message buffer with the specified size

MPI_Buffer_detach (&buffer,size): frees the specified buffer

MPI_Rsend: blocking ready send, copies directly to the receive application space buffer, but the receive must be posted before being invoked.

MPI_Sendrecv: performs a blocking send and a blocking receive. Processes can swap without deadlock

Wait and probe

MPI_Wait (&request, &status): wait until the operation specified by *request* (specified in an *Isend/Irecv* finishes)

MPI_Waitany (count, &array_of_requests, &index,&status): wait for any blocking operations specified in *&array_of_requests* to finish

MPI_Waitall (count, &array_of_requests, &array_of_statuses): wait for all blocking operations specified in *&array_of_requests* to finish

MPI_Waitsome (incount, &array_of_requests, &outcount, &array_of_offsets, &array_of_statuses): wait for at least one request to finish, the number is returned in *outcount*.

MPI_Probe (source, tag, comm, &status): performs a blocking test but doesn't require a corresponding receive to be posted.

Example of blocking send/recv

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int numtasks, rank, dest, source, rc, count, tag=1;
    char inmsg, outmsg='x';
    MPI_Status Stat; // status structure

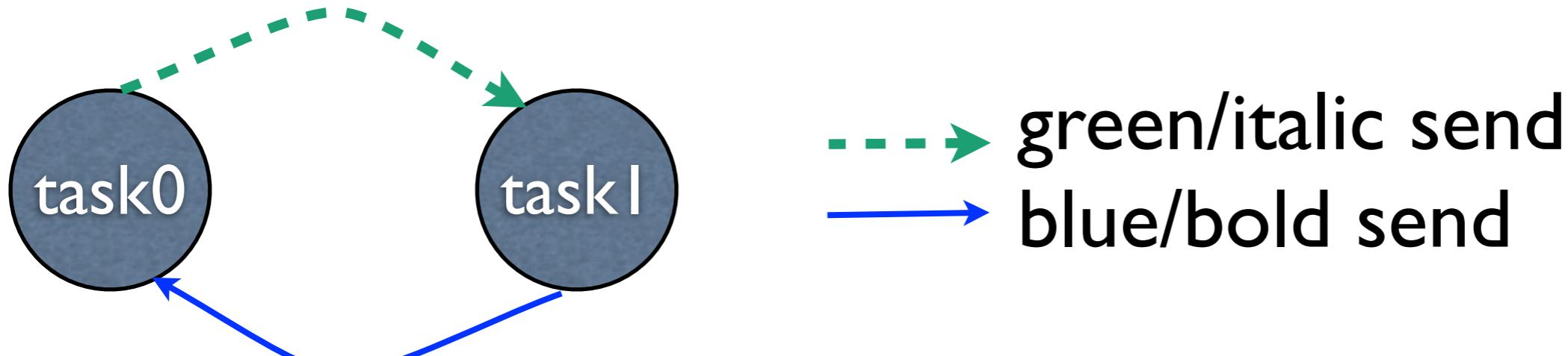
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Example of blocking send/recv

```
if (rank == 0) {  
    dest = 1;  
    source = 1;  
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);  
} else if (rank == 1) {  
    dest = 0;  
    source = 0;  
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);  
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
}  
  
rc = MPI_Get_count(&Stat, MPI_CHAR, &count); // returns # of type received  
printf("Task %d: Received %d char(s) from task %d with tag %d \n",  
    rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);  
  
MPI_Finalize();  
}
```

Example of blocking send/recv

```
if (rank == 0) {  
    dest = 1;  
    source = 1;  
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);  
} else if (rank == 1) {  
    dest = 0;  
    source = 0;  
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);  
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
}  
}
```



Non-blocking operations

- **MPI_Isend, MPI_Irecv, MPI_Issend, Ibsend, Irsend:** similar to *MPI_Send, MPI_Recv, MPI_Ssend, Bsend, Rsend* except that a `Test` or `Wait` must be used to determine that the operation has completed and the buffer may be read (in the case of a `recv`) or written (in the case of a `send`).
- `MPI_Test (&request, &flag,&status)`
- `MPI_Testany (count, &array_of_requests, &index, &flag, &status)`
- `MPI_Testall (count,&array_of_requests,&flag, &array_of_statuses)`
- `MPI_Testsome (incount, &array_of_requests, &outcount, &array_of_offsets, &array_of_statuses)`
- Like the wait operations, but do not block

Non-blocking example

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];
{
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Non-blocking example

```
prev = rank-1;  
next = rank+1;  
if (rank == 0) prev = numtasks - 1;  
if (rank == (numtasks - 1)) next = 0;
```

```
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);  
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
```

```
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);  
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
```

```
{ do some work }
```

```
MPI_Waitall(4, reqs, stats);  
MPI_Finalize();  
}
```

Nearest neighbor exchange
in a ring topology

Collective communication routines

- Use these when communicating among processes with a well defined pattern
- Some can be used to allow all processes to communicate
- Some perform computation during the communication (reductions)
- Involve all processes in the specified communicator, even if a particular processor has no data to send
- Can only be used with MPI predefined types, not derived *types*.
- The programmer has to make sure all processes participate in the collective operation

All processors participate in the collective operation

```
if (pid % 2) {  
    MPI_Reduce(..., MPI_COMM_WORLD);  
}
```

This program will deadlock, as the MPI_Reduce will wait forever for even processes to begin executing it.

If you want to only involve odd processes, add them to a new communicator.

Groups and communicators

- Two terms used in MPI documentation are *groups* and *communicators*.
- A *communicator* is a group of processes that can communicate with each other
- A *group* is an ordered set of processes
- Programmers can view groups and communicators as being identical

Collective routines

MPI_Barrier (comm): tasks block upon reaching the barrier until every task in the group has reached it

MPI_Bcast (&buffer,count,datatype,root,comm): process *root* sends a copy of its data to every other processor. Should be $\log_2(\text{comm_size})$ operation.

**MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,
recvcnt,recvtype,root,comm)**: distributes a unique message from *root* to every process in the group.

**MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype,
root, comm)**: opposite of scatter, every process in the group sends a unique message to the *root*.

**MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf,
recvcount,recvtype,comm)**: each tasks performs a one-to-all broadcast to every other process in the group These are concatenated together in the *recvbuf*.

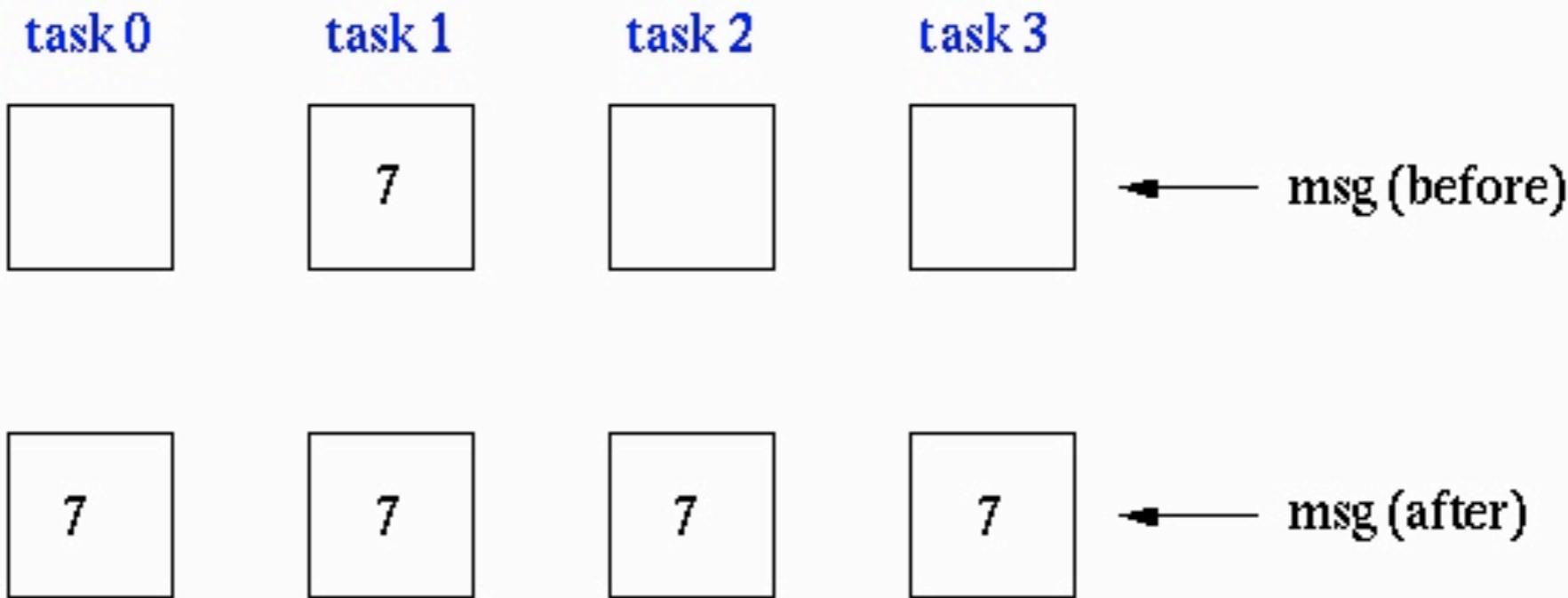
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm): performs a reduction using operation *op* and places the result into *recvbuf* on the *root* process.

MPI_Bcast

MPI_Bcast

Broadcasts a message to all other processes of that group

```
count = 1;  
source = 1;           broadcast originates in task 1  
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

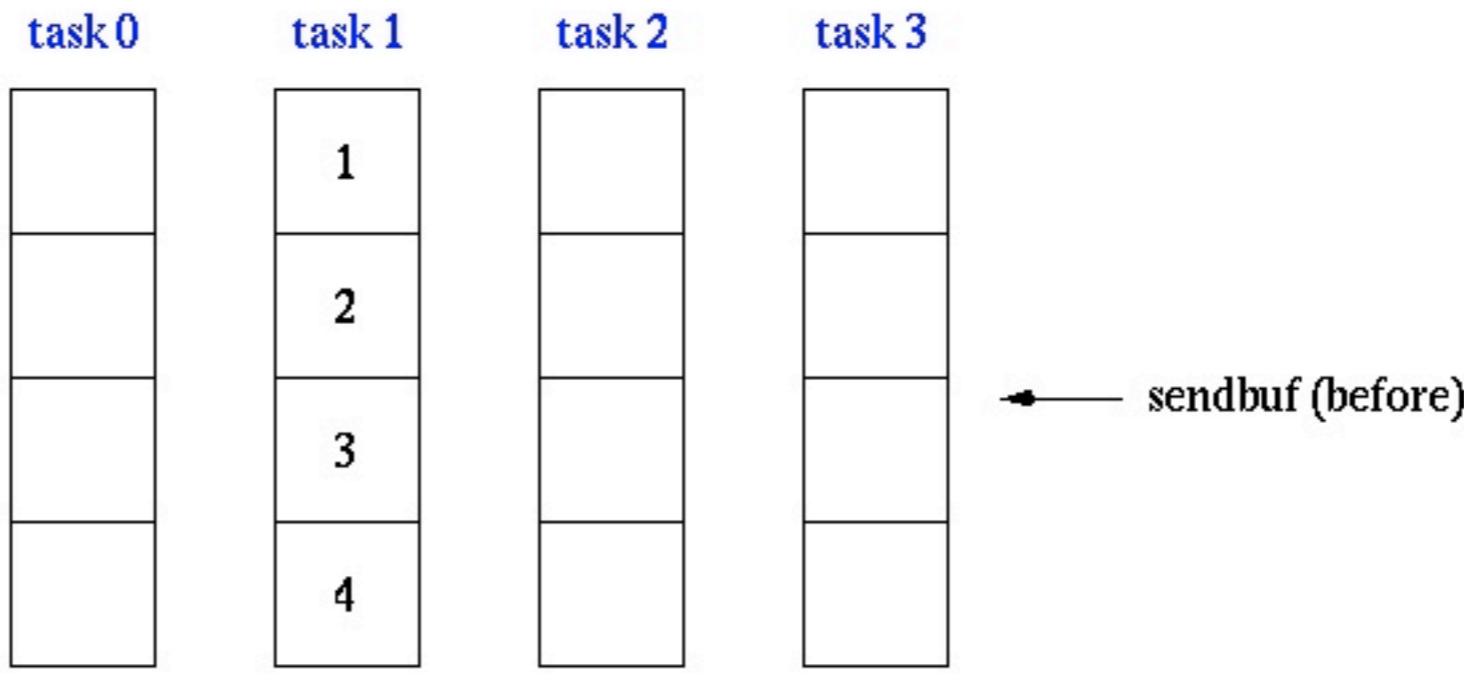


MPI Scatter

MPI_Scatter

Sends data from one task to all other tasks in a group

```
sendcnt = 1;  
recvcnt = 1;  
src = 1;           task 1 contains the message to be scattered  
MPI_Scatter(sendbuf, sendcnt, MPI_INT,  
            recvbuf, recvcnt, MPI_INT,  
            src, MPI_COMM_WORLD);
```



Equivalent to



`MPI_Send(sendbuf+i*sendcount*extent(sendtype), sendcount, sendtype, i, ...)`

`MPI_Recv(recvbuf, recvcount, recvtype, i, sendcount, sendtype, i, ...)`

MPI_Gather

MPI_Gather

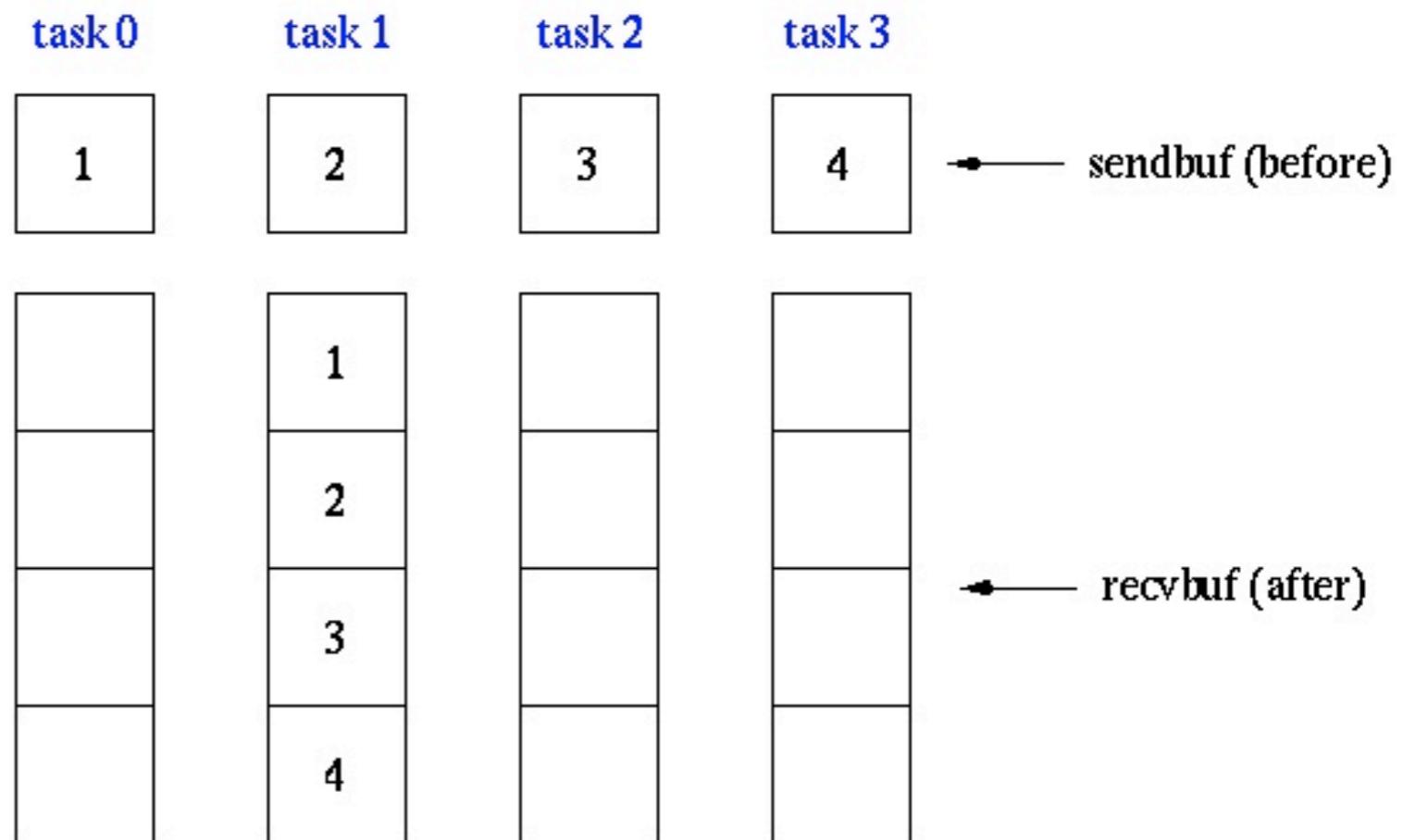
Gathers together values from a group of processes

```
sendcnt = 1;  
recvcnt = 1;  
src = 1;           messages will be gathered in task 1  
MPI_Gather(sendbuf, sendcnt, MPI_INT,  
            recvbuf, recvcnt, MPI_INT,  
            src, MPI_COMM_WORLD);
```

MPI_Send(sendbuf, sendcount,
 sendtype, root, ...)

MPI_Recv(recvbuf+
 i*recvcount*
 extent(recvtype),
 recvcount,
 recvtype, i, ...)

With the results of each
recv stored in rank order of
the sending process



MPI_Allgather

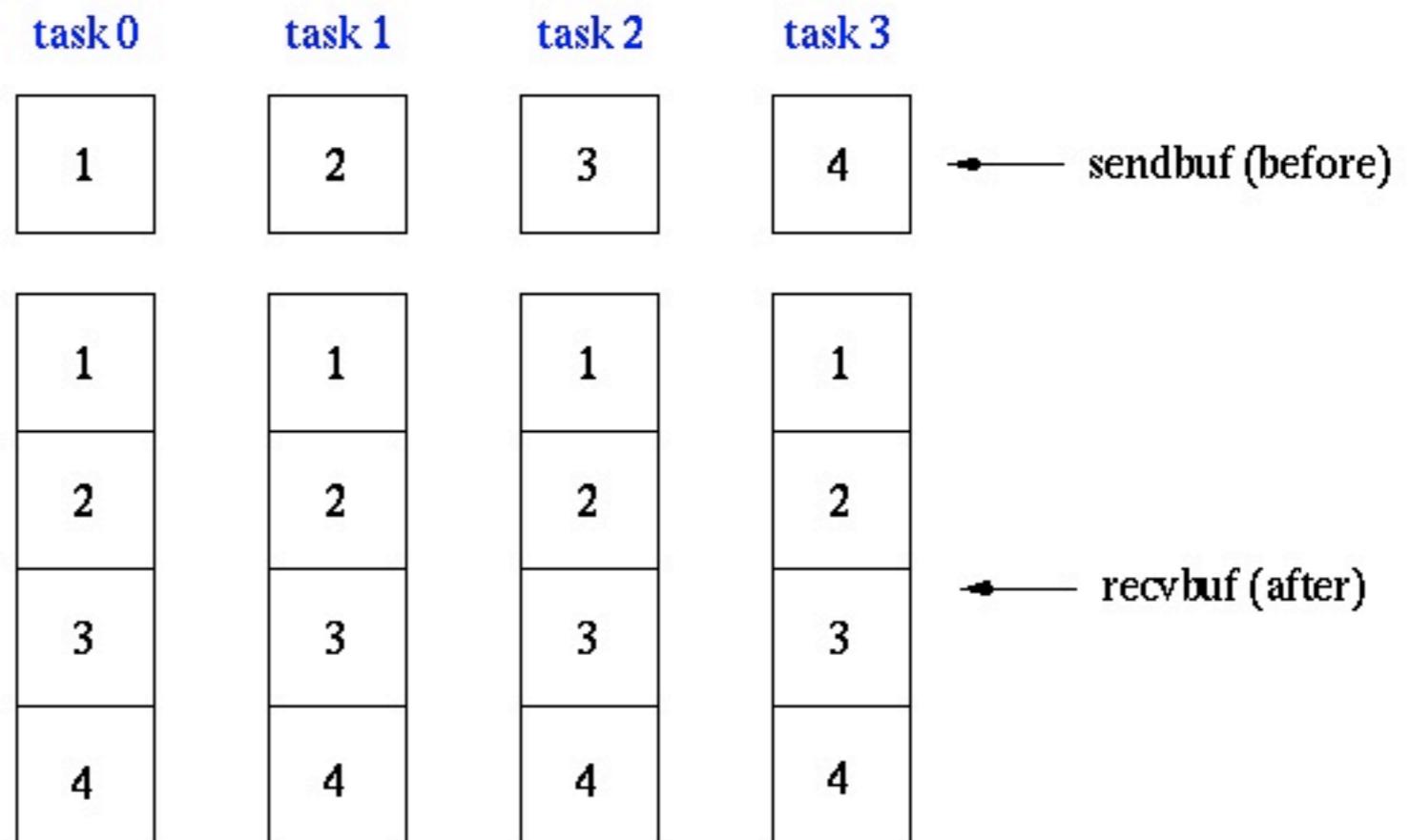
MPI_Allgather

Gathers together values from a group of processes and distributes to all

`sendcnt = 1;
recvcnt = 1;`

`MPI_Allgather(sendbuf, sendcnt, MPI_INT,
recvbuf, recvcnt, MPI_INT,
MPI_COMM_WORLD);`

An gather with
every process
being a target.



MPI_Reduce

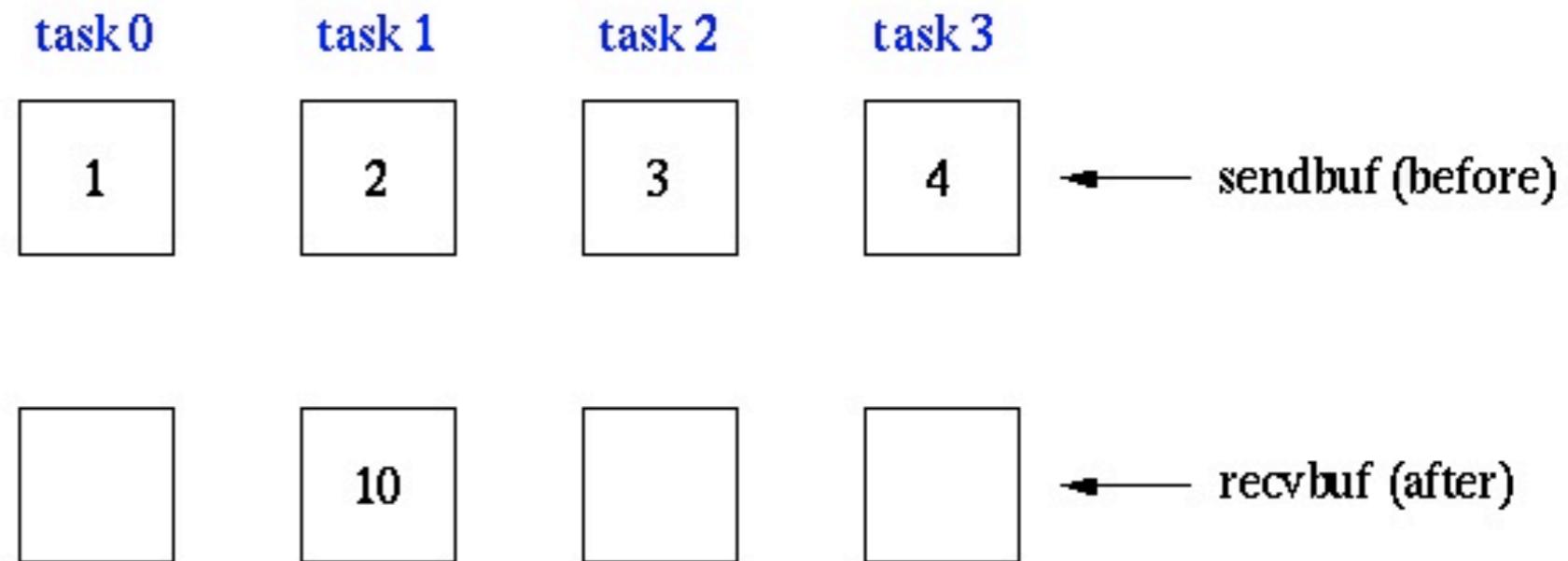
Also see MPI introductory slides.

You can form your own reduction function using MPI_Op_create

MPI_Reduce

Perform and associate reduction operation across all tasks in the group and place the result in one task

```
count = 1;  
dest = 1;                                result will be placed in task 1  
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,  
           dest, MPI_COMM_WORLD);
```



MPI_Op_create

```
#include "mpi.h"
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op )
```

pointer
to the user
defined
function

true if
commutative, false
otherwise

Handle
to refer to the
function wherever
an MPI_Op is
needed

More operations

MPI_Allreduce (&sendbuf, &recvbuf, count, datatype, op, comm): functionally equivalent to an *MPI_Reduce* followed by an *MPI_Bcast*. Faster on most hardware than the combination of these.

MPI_Reduce_scatter(&sendbuf, &recvbuf, recvcount, datatype, op, comm): Does an element-wise reduce on the vector in *sendbuf* of length *recvcount*. The vector is then split into disjoint segments and spread across the tasks. Equivalent to an *MPI_Reduce* followed by an *MPI_Scatter* operation.

MPI_Alltoall(&sendbuf, sendcount, sendtype, &recvbuf, recvnt, recvtype, comm): Each task in the group performs a scatter with the results concatenated on each process in task rank order.

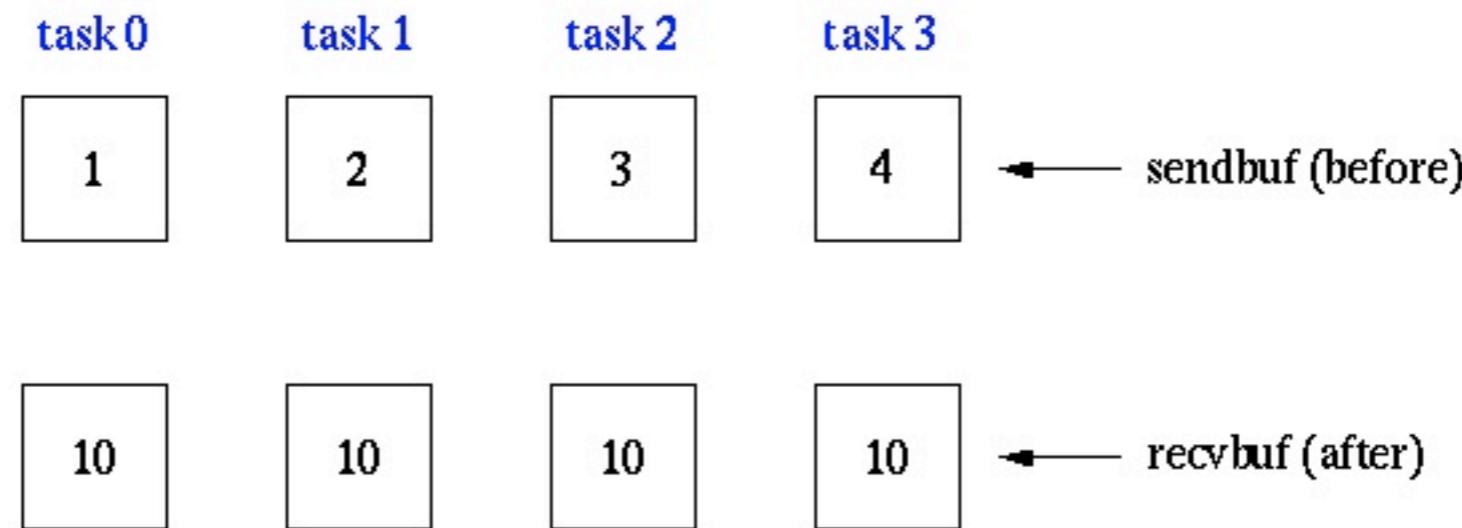
MPI_Scan(&sendbuf, &recvbuf, count, datatype, op, comm): performs the partial sums on each processor that would result from doing an in-order reduction across the processors in rank order.

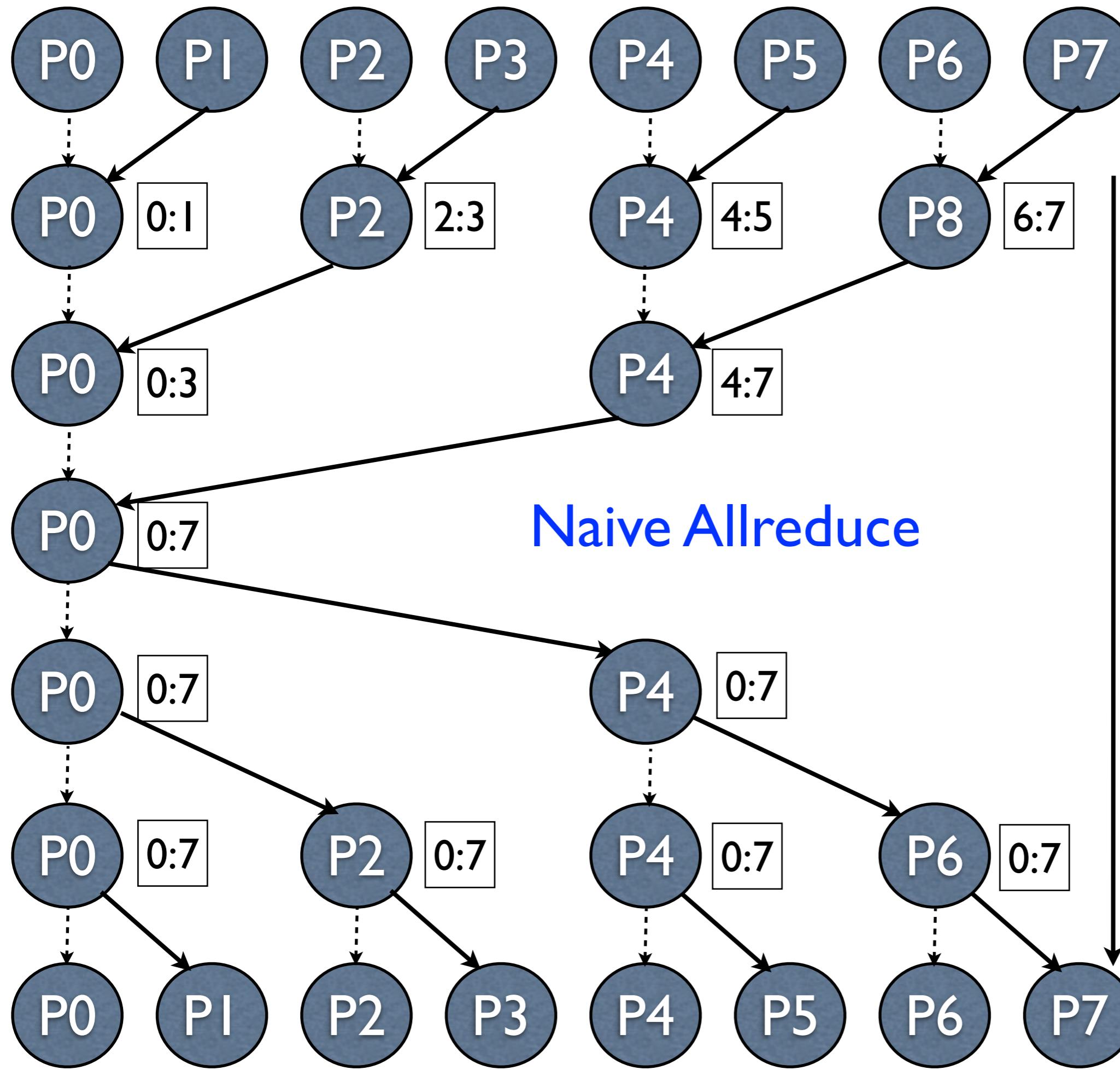
MPI_Allreduce

MPI_Allreduce

Perform and associate reduction operation across all tasks in the group and place the result in all tasks

```
count = 1;  
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,  
              MPI_COMM_WORLD);
```

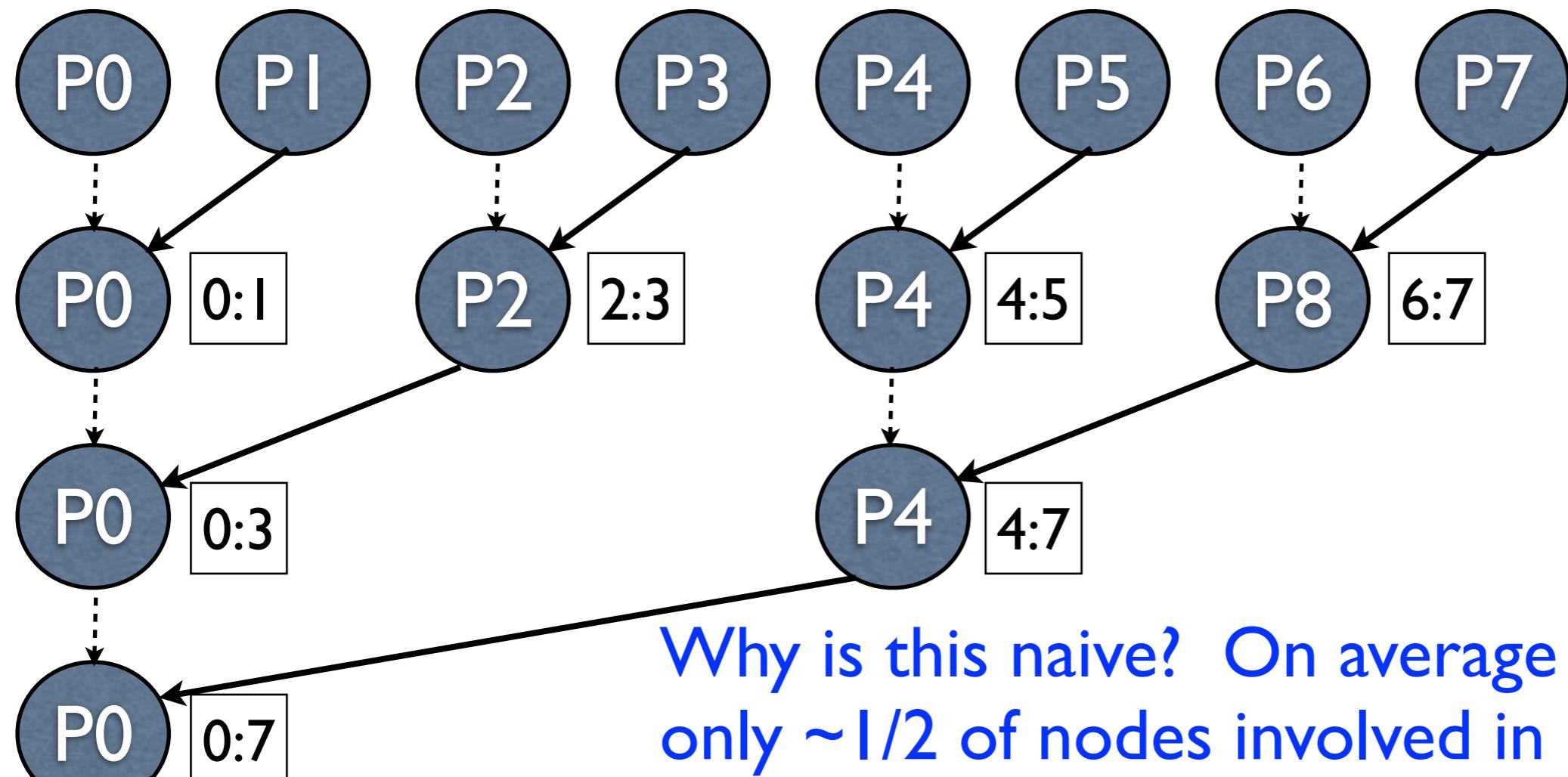




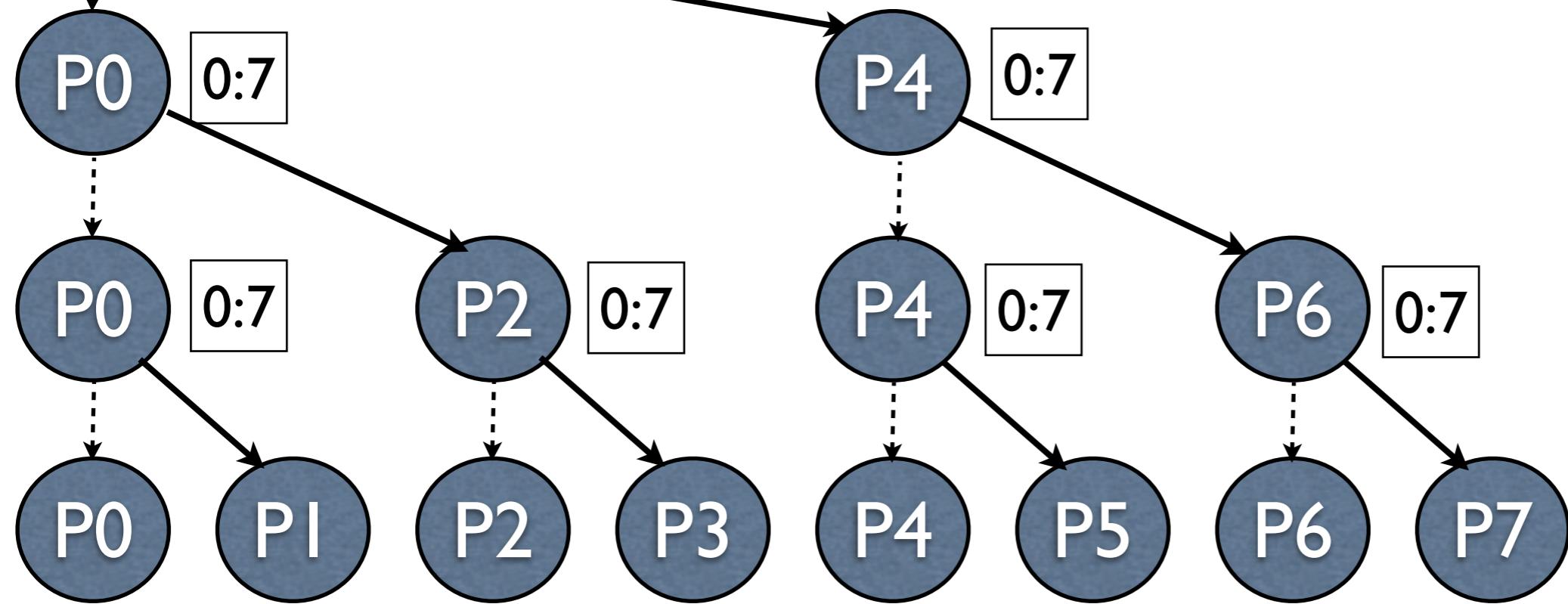
Naive Allreduce

$2 * \log_2(|P|)$
steps

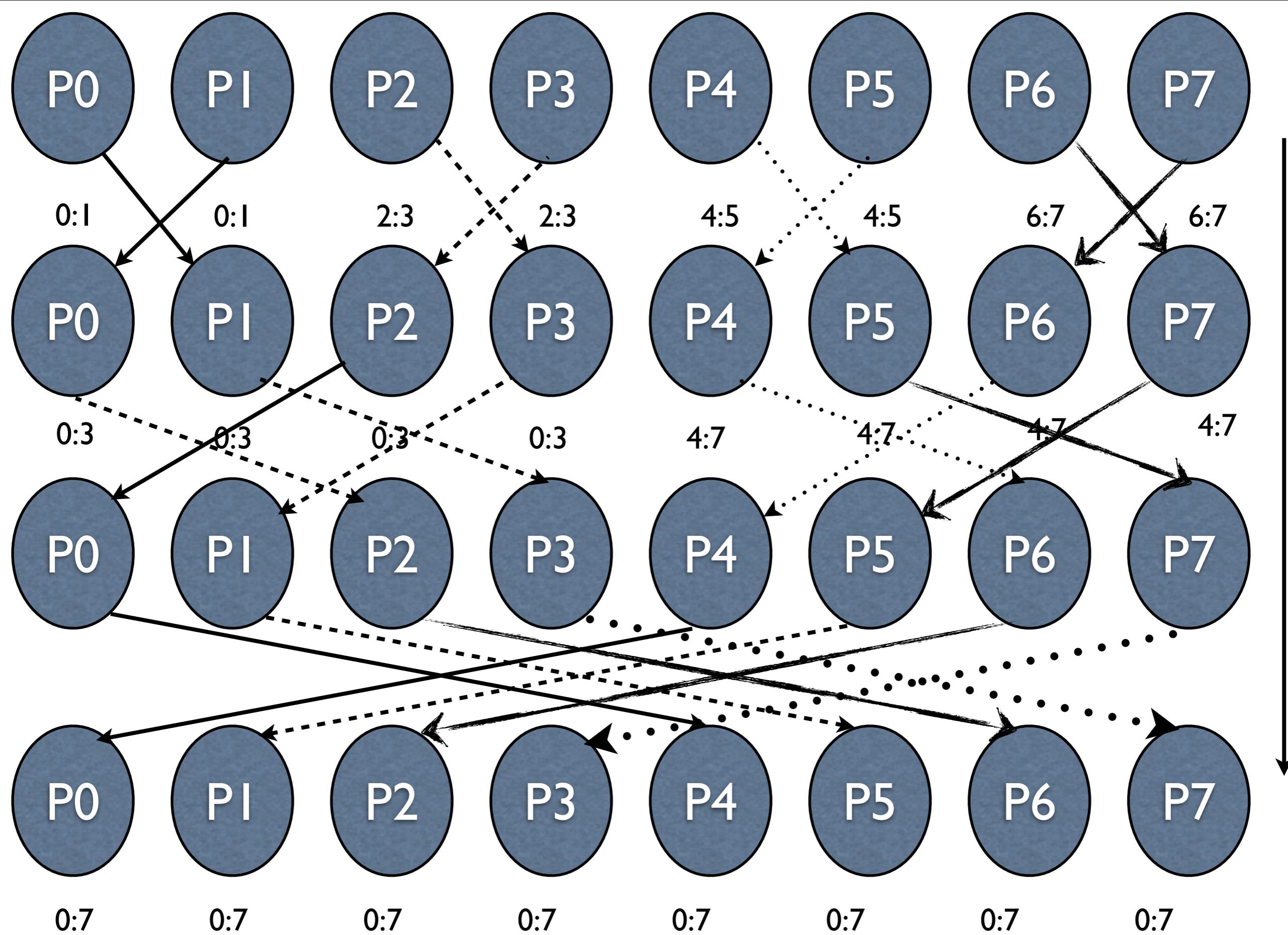
all have $0:7$



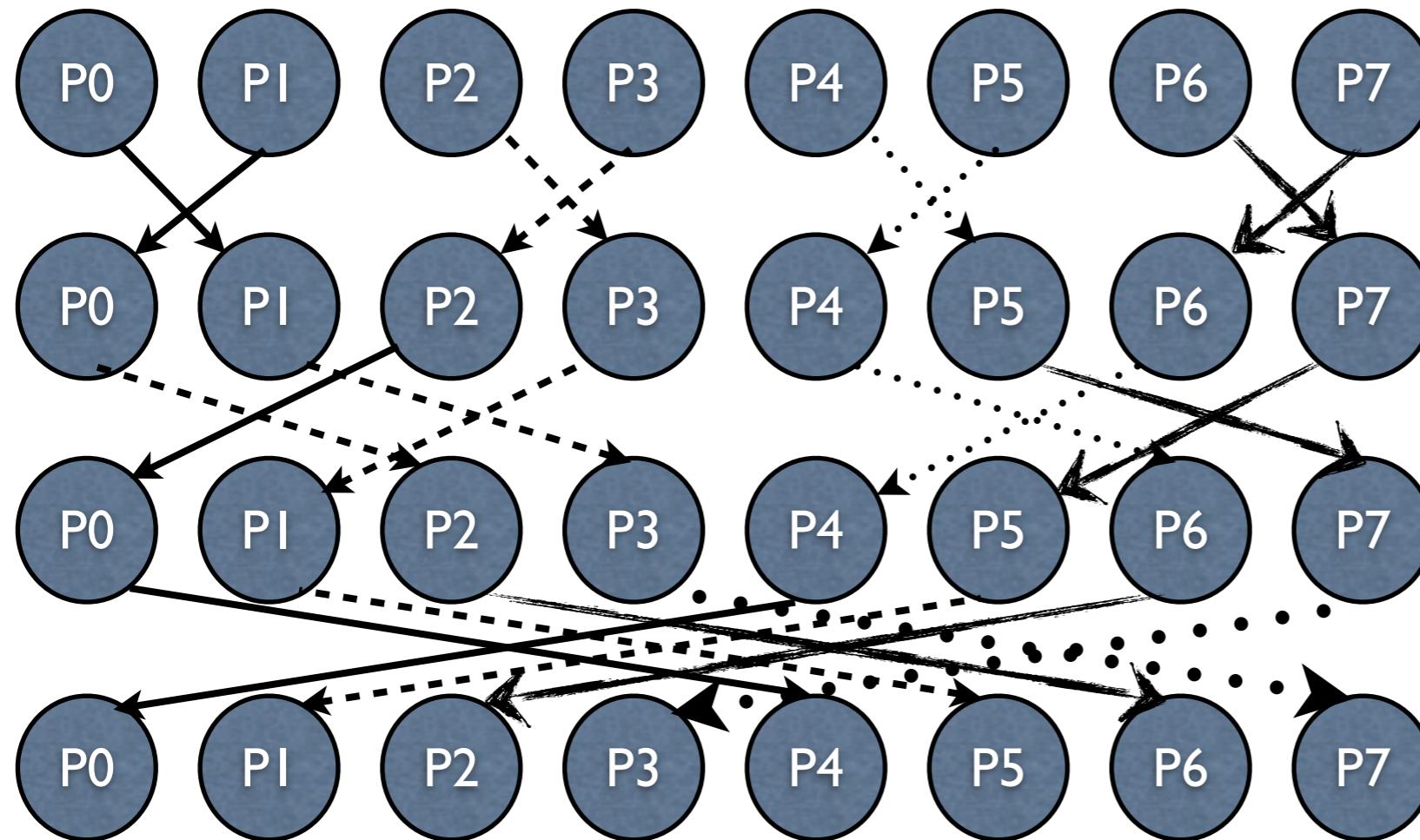
Why is this naive? On average
only $\sim 1/2$ of nodes involved in
communication each step



all have 0:7



$\log_2(|P|)$ steps



All processors all busy each step.

Note that the bandwidth requirements of the network change

The faster algorithm relies on current network interface cards being at least dual ported. Each node in the system can simultaneously send and receive a message.

Algorithm from *Optimization of Collective Reduction Operations*, Rolf Rabenseifner, International Conference on Computational Science, 2004

MPI Reduce scatter

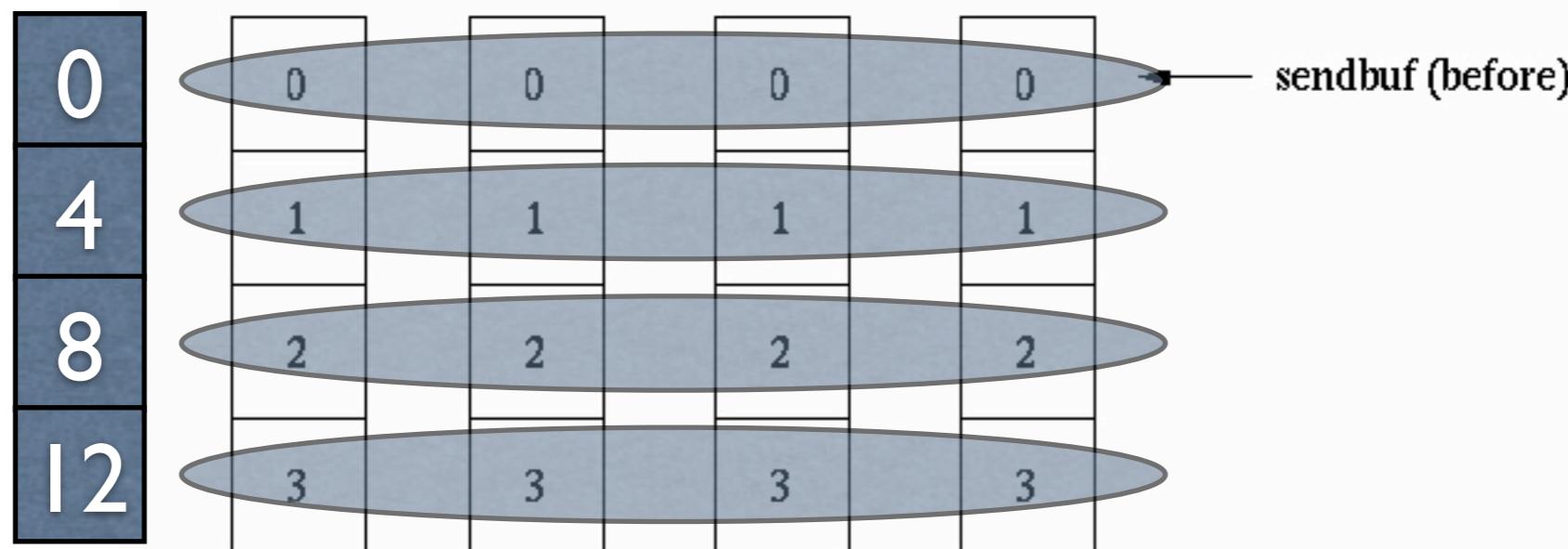
MPI_Reduce_scatter

Perform reduction operation on vector elements across all tasks in the group, then distribute segments of result vector to tasks

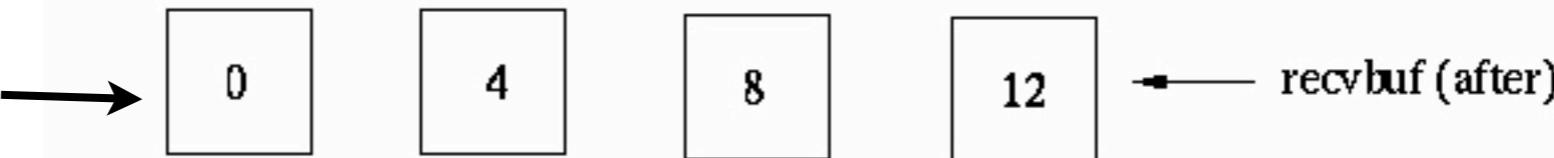
```
recvcount = 1;  
MPI_Reduce_scatter(sendbuf, recvbuf, recvcount, MPI_INT, MPI_SUM,  
                    MPI_COMM_WORLD);
```

reduce

result



result of scattering
the reduce result



MPI_Alltoall

MPI_Alltoall

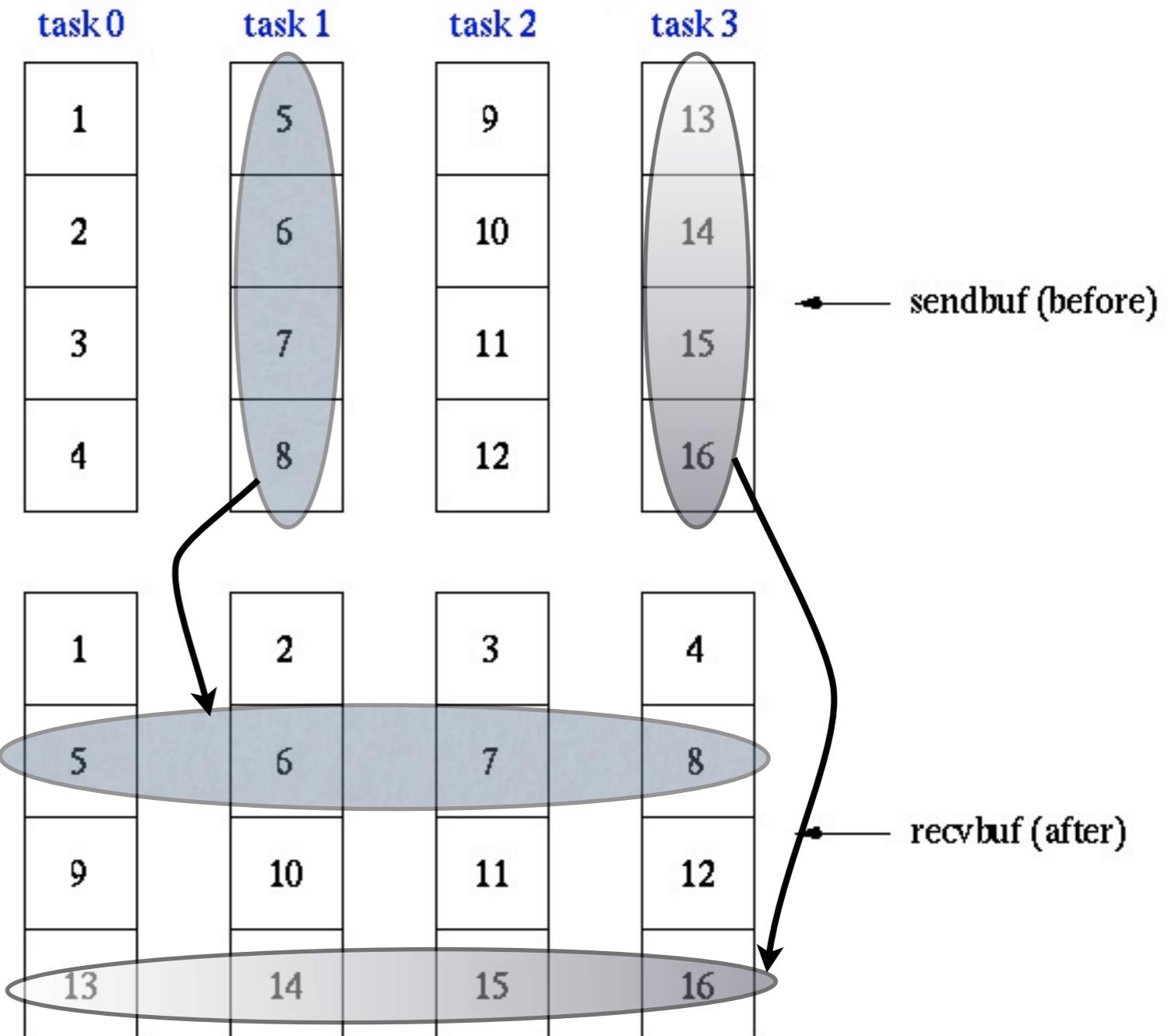
Sends data from all to all processes. Each process performs a scatter operation.

sendcnt = 1;
recvcnt = 1;

`MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
recvbuf, recvcnt, MPI_INT,
MPI_COMM_WORLD);`

Each process performs a scatter of its elements to all other processes.

Received data is concatenated in sender rank order

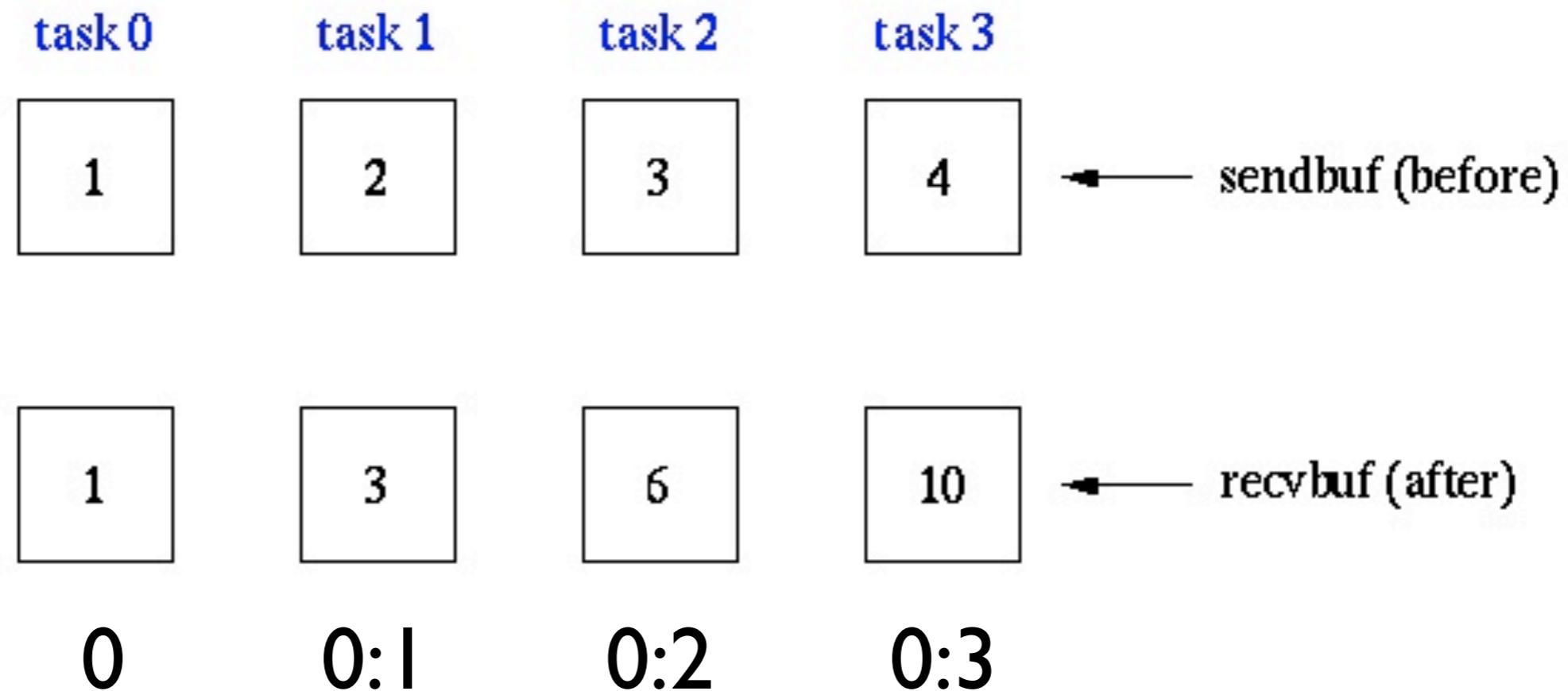


MPI_Scan

MPI_Scan

Computes the scan (partial reductions) of data
on a collection of processes

```
count = 1;  
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,  
         MPI_COMM_WORLD);
```



Group and communicator

- Remember that
 - A *communicator* is a group of processes that can communicate with each other
 - A *group* is an ordered set of processes
- Programmers can view groups and communicators as being the same thing
- group routines are used in collecting processes to form communicator.

Why groups and communicators?

- Allow programmer to organize tasks by functions
- Enable collective communication operations
- Allow user-defined *virtual topologies* to be formed
- Enable manageable communication by enabling synchronization

Properties

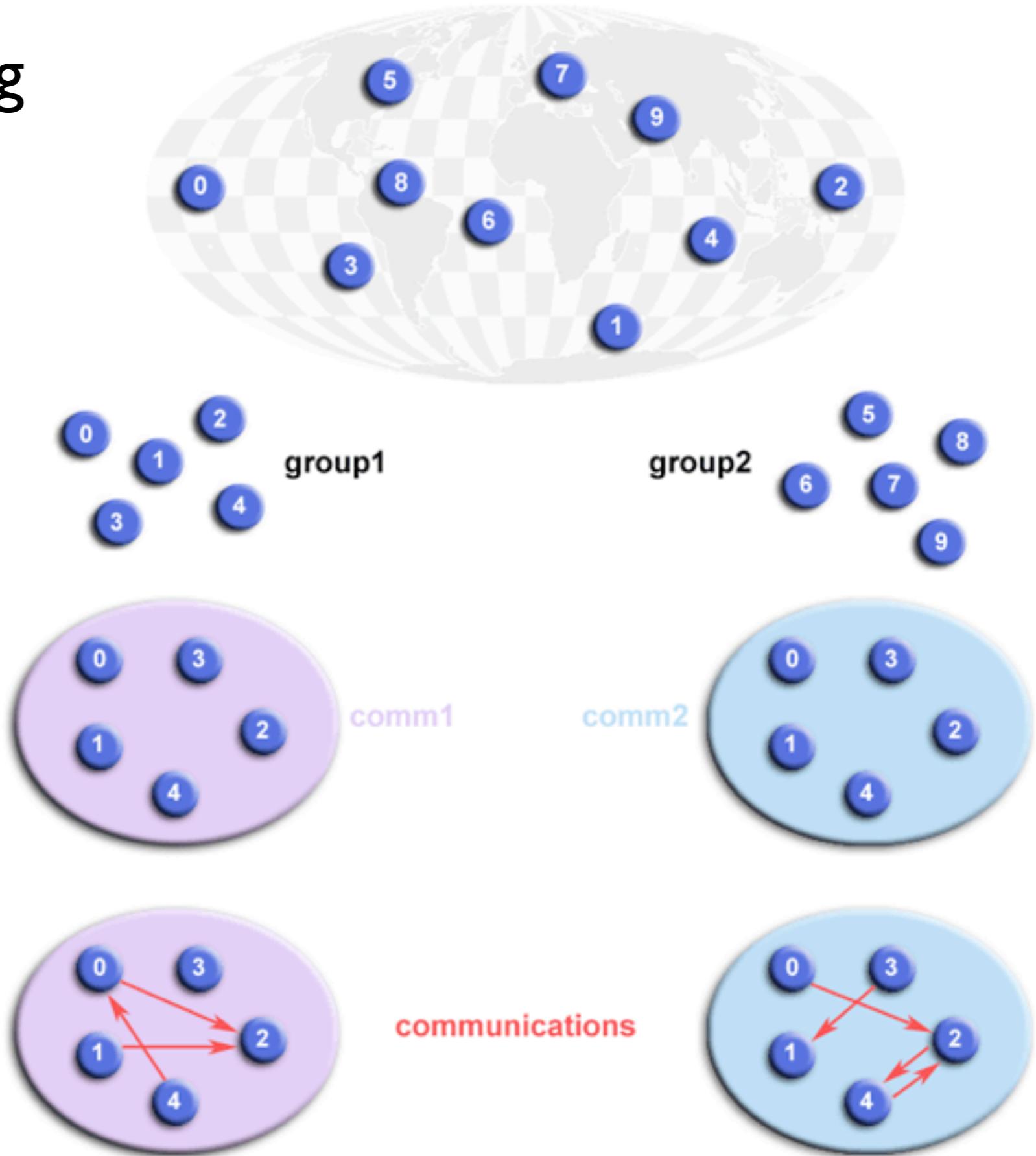
- Groups/communicators are dynamic, i.e. they can be created and destroyed
- Processes can be in many groups, and will have a unique, possibly different, rank in each group
- MPI provides 40+ routines for managing groups and communicators! Mercifully, we will not cover them all.

Tasks these 40+ routines can perform

Extract handle of a global group a communicator using
`MPI_Comm_group`

- Form new group as a subset of another group using
`MPI_Group_incl`
- Create new communicator for a group using
`MPI_Comm_create`
- Determine a processor's rank in a communicator using
`MPI_Comm_rank`
- Communicate among the processors of a group
- When finished, free communicators and groups using
`MPI_Comm_free` and `MPI_Group_free`

Relationships among communicators and groups.



```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

int main(argc,argv)
int argc;
char *argv[];
{
    int      rank, new_rank, sendbuf[recvbuf], numtasks,
            ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
    MPI_Group orig_group, new_group;
    MPI_Comm  new_comm;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    if (numtasks != NPROCS) {
        printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
        MPI_Finalize();
        exit(0);
    }
}
```

Handle for
MPI_COMM_WORLD

Handle for a
new group

Handle for a new
communicator

Get the
number of tasks and
the rank of
MPI_COMM_WORLD

sanity check code

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8
```

Variables to hold information about the new group this will be in. Note that since this is an SPMD program, if we do this statically we need information for all groups the process can be in, not just the one that it is in.

```
int main(argc,argv)
int argc;
char *argv[];
{
    int rank, new_rank, sendbuf, recvbuf, numtasks,
        ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
```

```
MPI_Group orig_group, new_group;
MPI_Comm new_comm;
```

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtas
```

Hold the ranks of processors in (in MPI_COMM_WORLD) of processes in each of the two new groups.

```
if (numtasks != NPROCS) {
    printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
    MPI_Finalize();
    exit(0);
}
```

```
sendbuf = rank;
```

get handle for
MPI_COMM_WORLD

```
/* Extract the original group handle */  
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

```
/* Divide tasks into two distinct groups based upon rank */
```

```
if (rank < NPROCS/2) {
```

```
    MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
```

```
}
```

```
else {
```

```
    MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
```

```
}
```

```
/* Create new new communicator and then perform collective communications */
```

```
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
```

```
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
```

```
MPI_Group_rank (new_group, &new_rank);
```

```
printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);
```

```
MPI_Finalize();
```

```
}
```

Each process executes one of these statements.

Based on its number, becomes a member of one of the new groups.

```
sendbuf = rank;
```

```
/* Extract the original group handle */
```

```
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

```
/* Divide tasks into two distinct groups based upon rank */
```

```
if (rank < NPROCS/2) {
```

```
    MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group),  
}
```

```
else {
```

```
    MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);  
}
```

```
/* Create new new communicator and then perform collective communications */
```

```
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
```

```
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
```

```
MPI_Group_rank (new_group, &new_rank);
```

```
printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);
```

```
MPI_Finalize();
```

```
}
```

Create a
communicator from the
group formed above

Perform collective
communication within the
group

Get the
processes rank
within the new
group