Lecture 14

Collective Communication

Announcements

- Project Progress report, due next Weds 11/28

Today's lecture

- Collective Communication algorithms
- Sorting

Collective communication

- Collective operations are called by **all** processes within a communicator
- Basic collectives seen so far
  - Broadcast: distribute data from a designated root process to all the others
  - Reduce: combine data from all processes returning the result to the root process
  - Will revisit these
- Other Useful collectives
  - Scatter/gather
  - All to all
  - Allgather
- Diverse applications
  - Fast Fourier Transform
  - Sorting

Underlying assumptions

- Fast interconnect structure
  - All nodes are equidistant
  - Single-ported, bidirectional links
- Communication time is $\alpha + \beta n$ in the absence of contention
  - Determined by bandwidth β^{-1} for long messages
  - Dominated by latency α for short messages

Inside MPI-CH

- Tree like algorithm to broadcast the message to blocks of processes, and a linear algorithm to broadcast the message within each block
- Block size may be configured at installation time
- If there is hardware support (e.g. Blue Gene), then it is given responsibility to carry out the broadcast
- Polyalgorithms apply different algorithms to different cases, i.e. long vs. short messages, different machine configurations
- We'll use hypercube algorithms to simplify the special cases when $P=2^k$, k an integer

Details of the algorithms

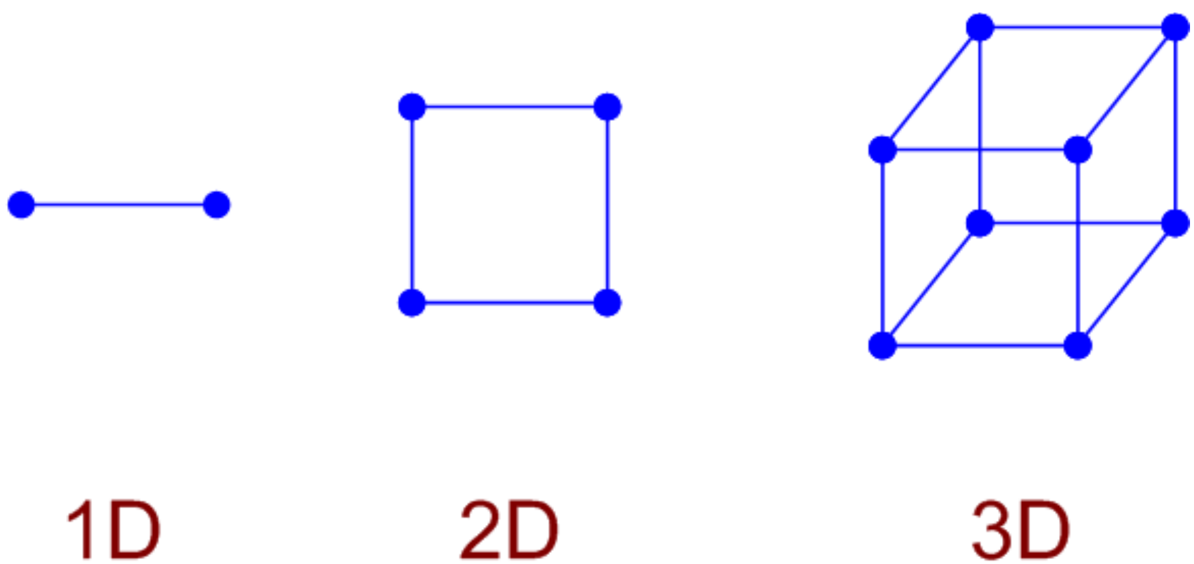
- Broadcast
- AllReduce
- Scatter/gather
- Allgather
- All to all

Broadcast

- The root process transmits of *m* pieces of data to all the *p-1* other processors
- Spanning tree algorithms are often used
- We'll look at a similar algorithm with logarithmic running time: the *hypercube algorithm*
- With the linear ring algorithm this processor performs $p-1$ sends of length m
  - Cost is $(p-1)(\alpha + \beta m)$

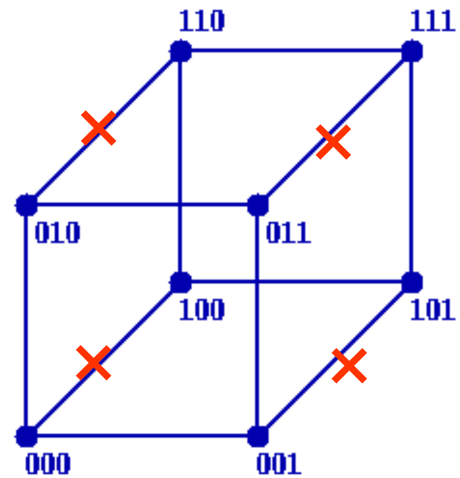
Sidebar: what is a hypercube?

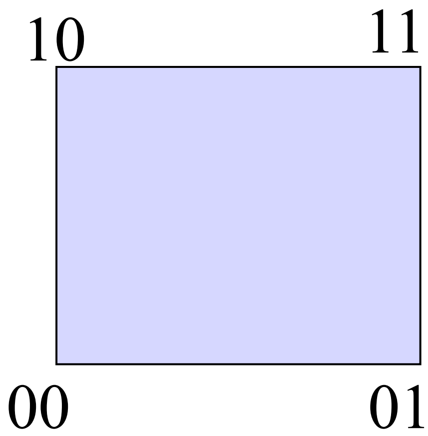
- A hypercube is a d-dimensional graph with 2^d nodes
- A 0-cube is a single node, 1-cube is a line connecting two points, 2-cube is a square, etc
- Each node has d neighbors



Properties of hypercubes

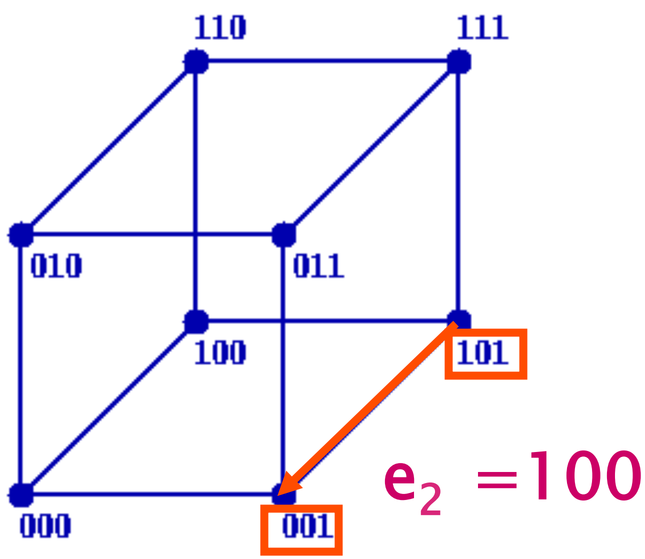
- A hypercube with p nodes has lg(p) dimensions
- *Inductive construction*: we may construct a d-cube from two (d-1) dimensional cubes
- **Diameter:** What is the maximum distance between any 2 nodes?
- **Bisection bandwidth:** How many cut edges (mincut)





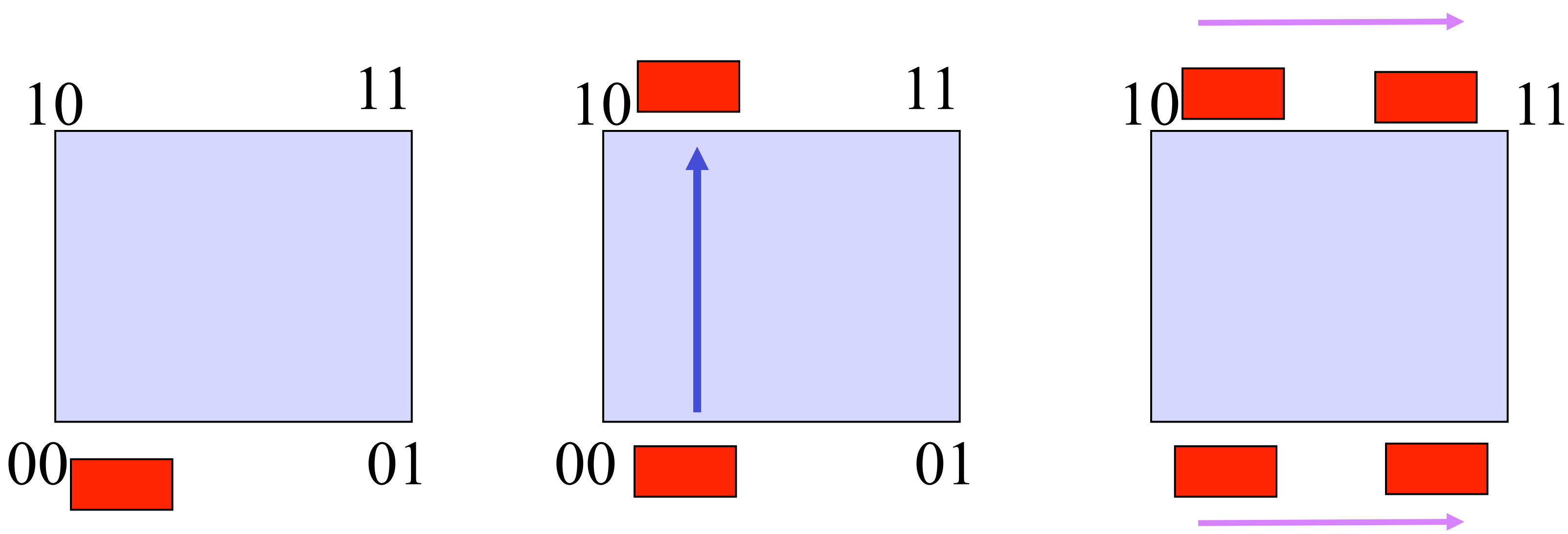
Bookkeeping

- Label nodes with a binary reflected grey code
  http://www.nist.gov/dads/HTML/graycode.html
- Neighboring labels differ in exactly one bit position $001 = 101 \otimes e_2, \quad e_2 = 100$



Hypercube broadcast algorithm with p=4

- Processor 0 is the root, sends its data to its hypercube "buddy" on processor 2 (10)
- Proc 0 & 2 send data to respective buddies



Details of the algorithms

- Broadcast
- AllReduce
- Scatter/gather
- Allgather
- All to all

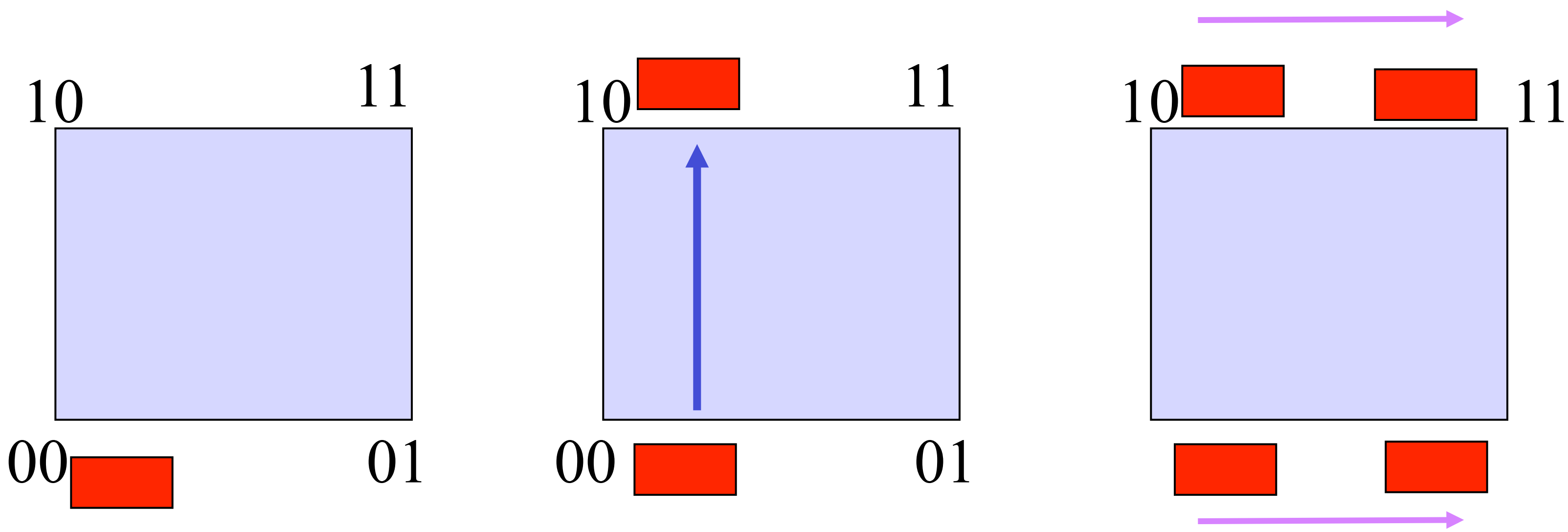
Reduction

- We may use the hypercube algorithm to perform reductions as well as broadcasts
- Another variant of reduction provides all processes with a copy of the reduced result

  `Allreduce( )`
- Equivalent to a `Reduce` + `Bcast`
- A clever algorithm performs an `Allreduce` in one phase rather than having perform separate reduce and broadcast phases

Allreduce

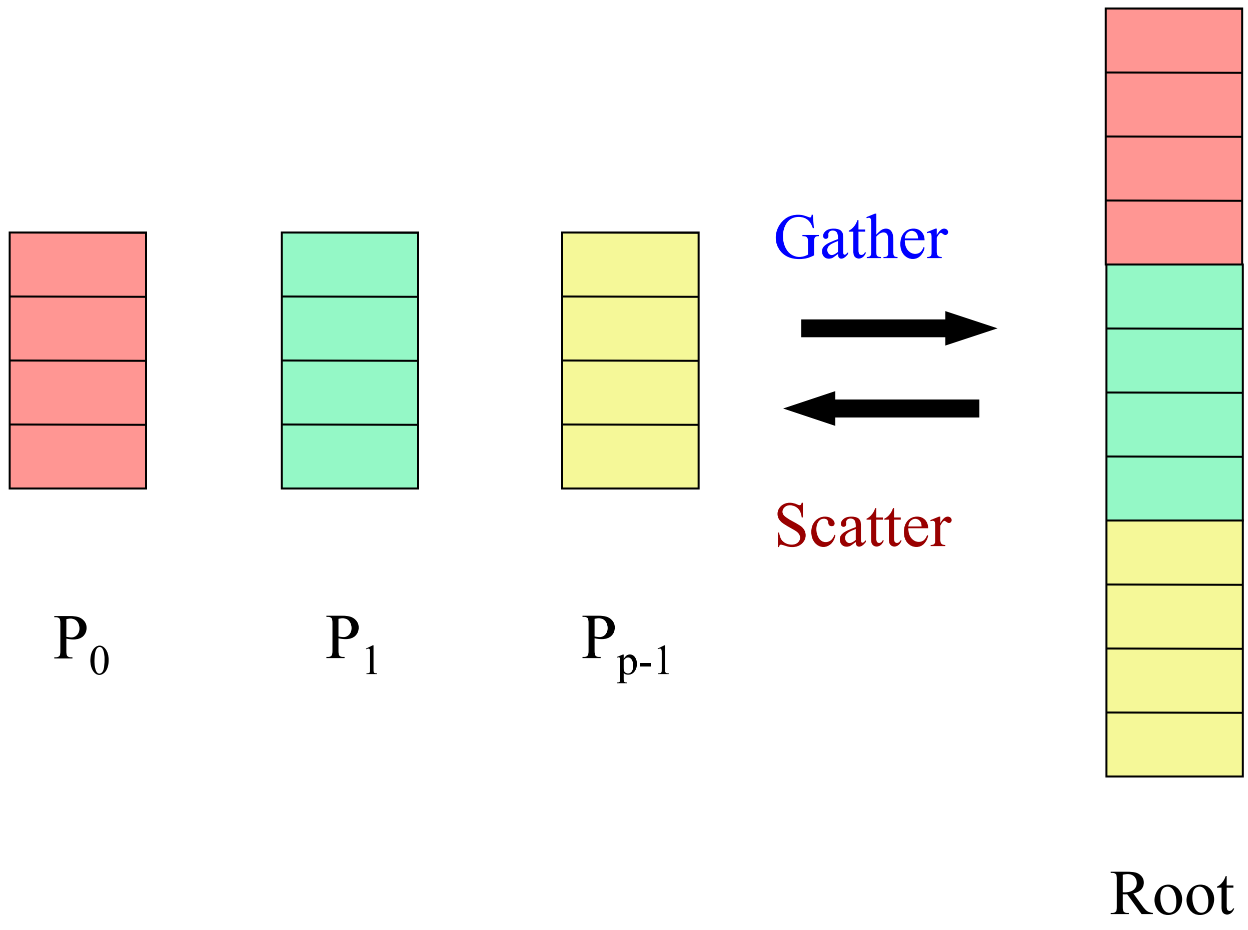
- Can take advantage of duplex connections



Details of the algorithms

- Broadcast
- AllReduce
- Scatter/gather
- Allgather
- All to all

Scatter/Gather



Scatter

- Simple linear algorithm
  - Root processor sends a chunk of data to all others
  - Reasonable for long messages

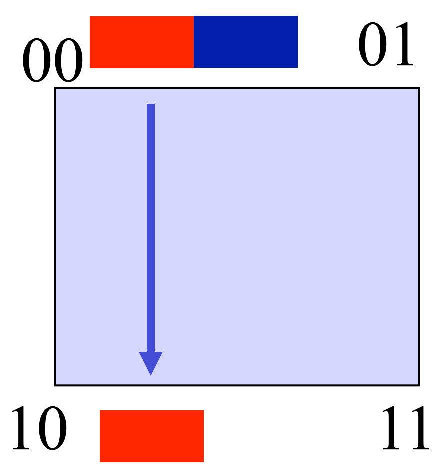
$$(p-1)\alpha + \frac{p-1}{p} n\beta$$

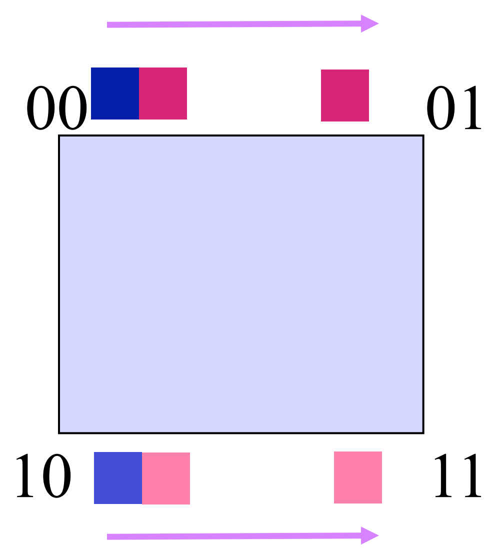
- Similar approach taken for Reduce and Gather
- For short messages, we need to reduce the complexity of the latency (α) term

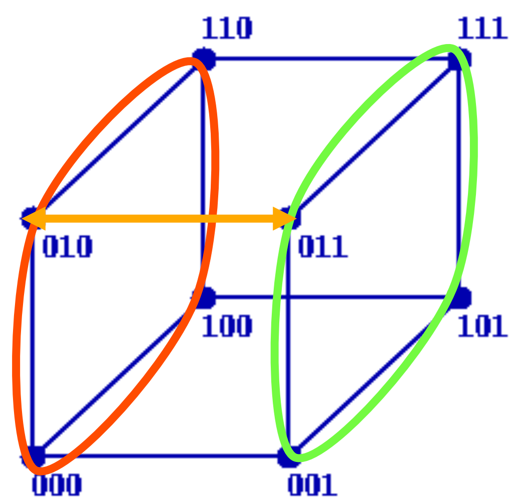
Minimum spanning tree algorithm

- Recursive hypercube-like algorithm with ⌈log P⌉ steps
  - Root sends half its data to process (root + p/2) **mod** p
  - Each receiver acts as a root for corresponding half of the processes
  - MST: organize communication along edges of a minimum-spanning tree covering the nodes
- Requires O(n/2) temp buffer space on intermediate nodes
- Running time:

$$\lceil \lg P \rceil \alpha + \frac{p-1}{p} n\beta$$





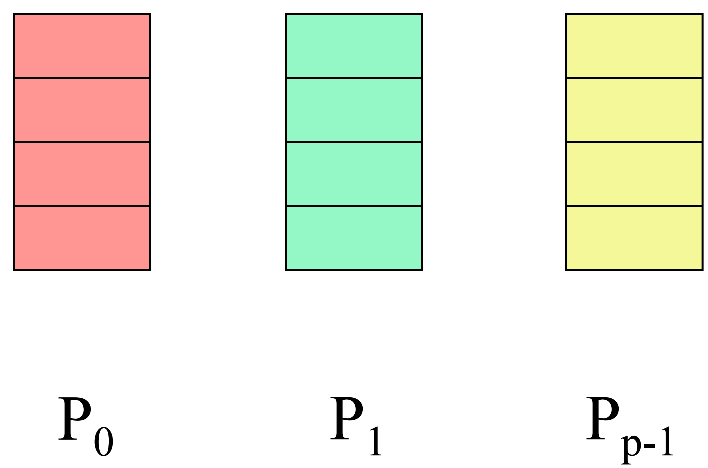


Details of the algorithms

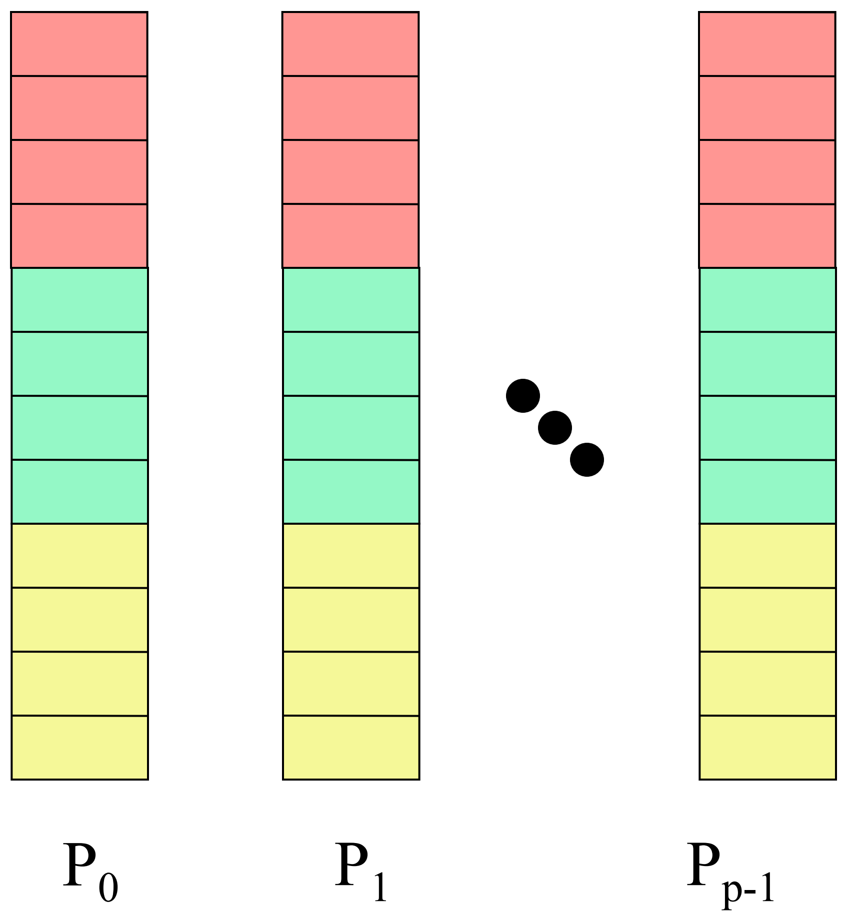
- Broadcast
- AllReduce
- Scatter/gather
- Allgather
- All to all

AllGather

- Equivalent to a gather followed by a broadcast
- All processors accumulate a chunk of data from all the others



AllGather



Allgather

- Use the all to all recursive doubling algorithm
- For P a power of two, running time is

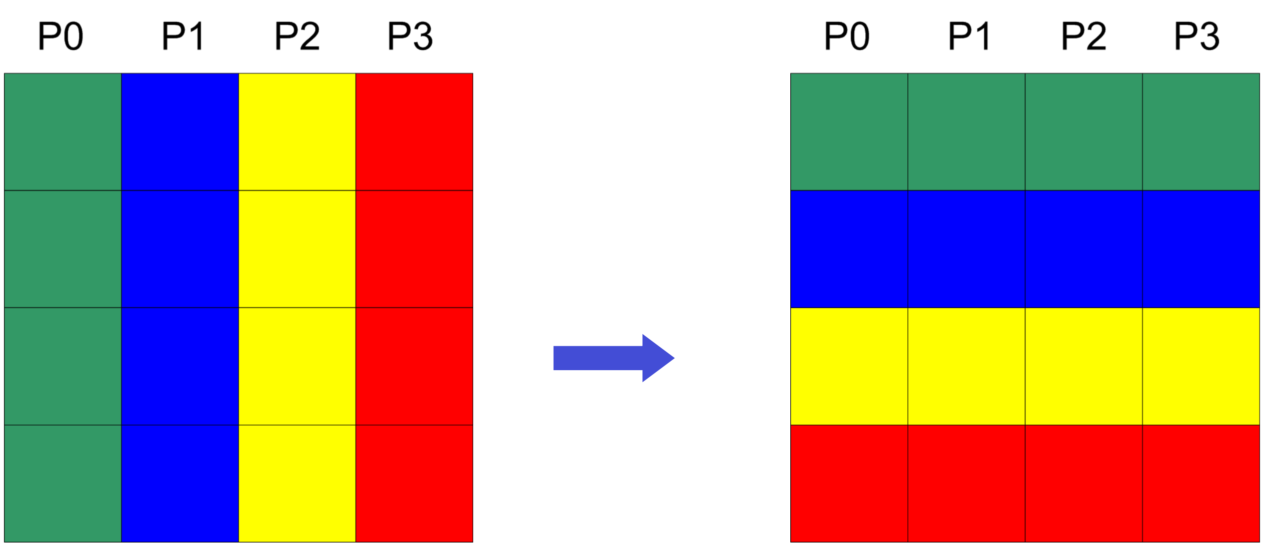
$$\lceil \lg P \rceil \alpha + \frac{p-1}{p} n\beta$$

Details of the algorithms

- Broadcast
- AllReduce
- Scatter/gather
- Allgather
- All to all

All to all

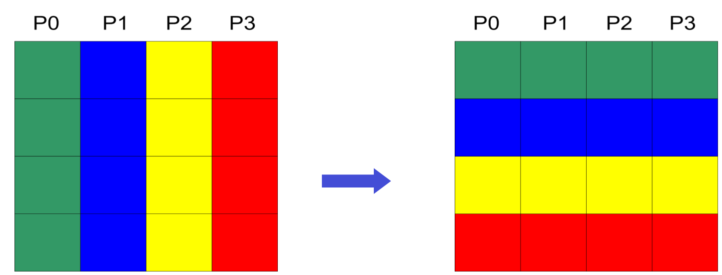
- Also called *total exchange* or *personalized communication:* a transpose
- Each process sends a different chunk of data to each of the other processes
- Used in sorting and the Fast Fourier Transform



Exchange algorithm

- n elements / processor (n total elements)
- p - 1 step algorithm
  - Each processor exchanges n/p elements with each of the others
  - In step i, process k exchanges with processes $k \pm$ i

```
for i = 1 to p-1
   src  =  (rank - i + p)  mod p
   dest =  (rank + i     )  mod p
   sendrecv( from src to dest )
end for
```



- Good algorithm for long messages
- Running time:

$$(p-1)\alpha \ + \ (p-1)\frac{n}{p}\beta \ \approx \ n\beta$$

Recursive doubling for short messages

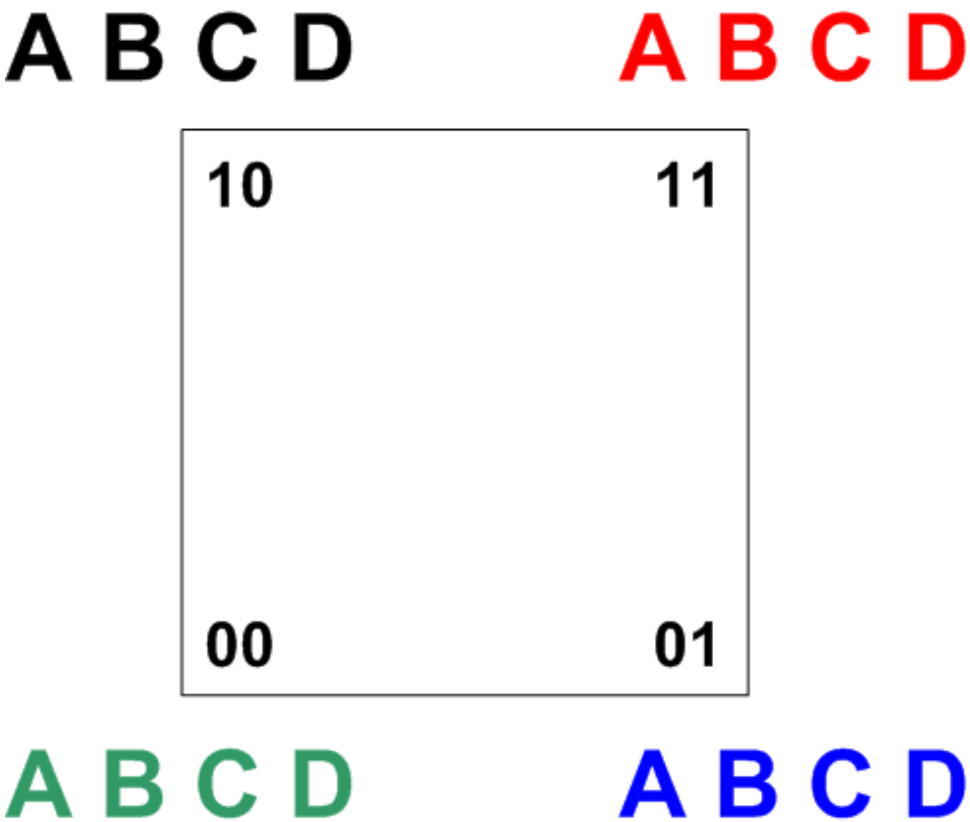
- In each of $\lceil \log p \rceil$ phases all nodes exchange ½ their accumulated data with the others
- Only P/2 messages are sent at any one time

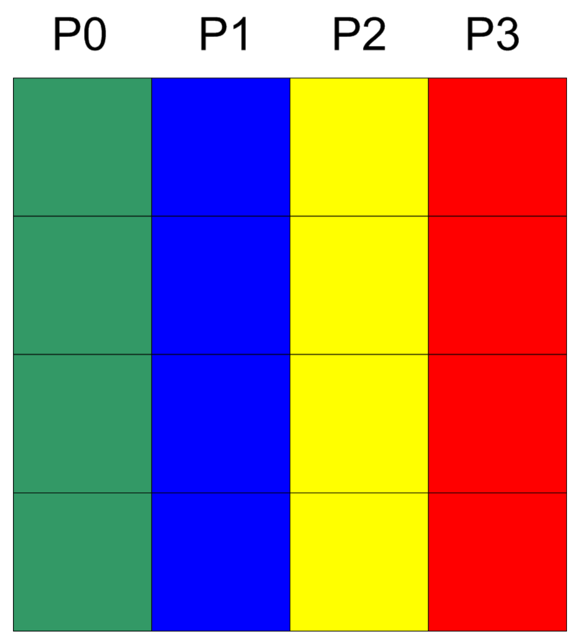
```
D = 1
while (D < p)
   Exchange & accumulate data with rank ⊗ D
   Left shift D by 1
end while
```

- Optimal running time for short messages

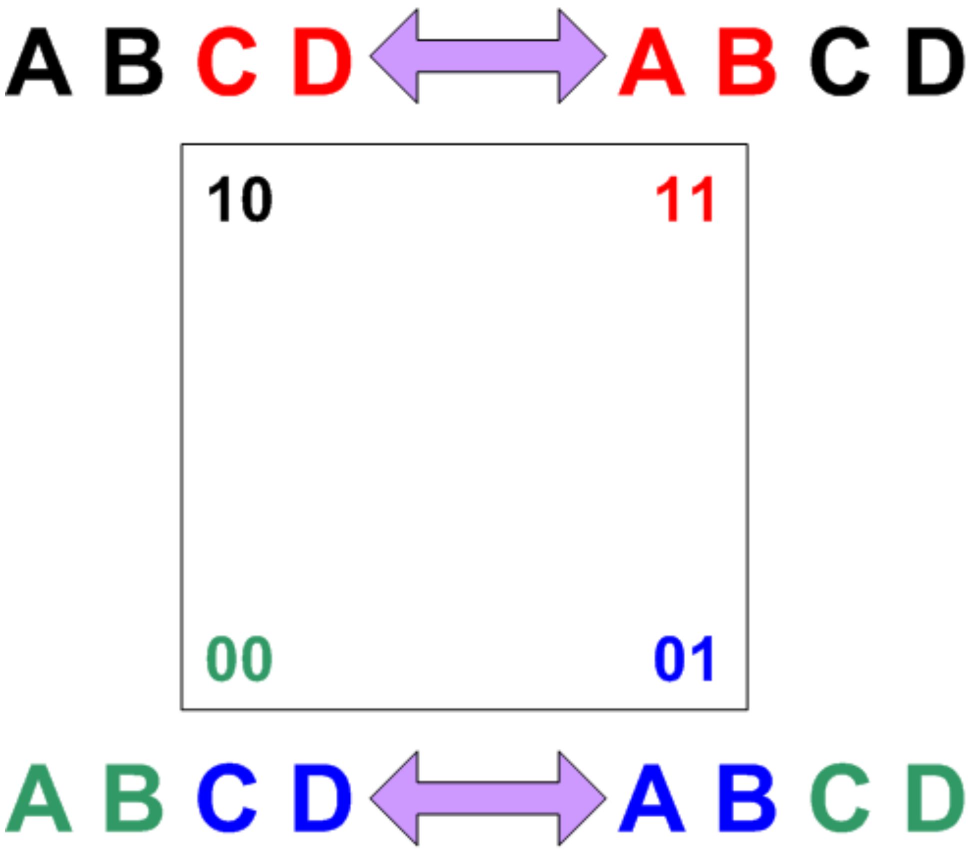
$$\lceil \lg P \rceil \alpha + nP\beta \approx \lceil \lg P \rceil \alpha$$

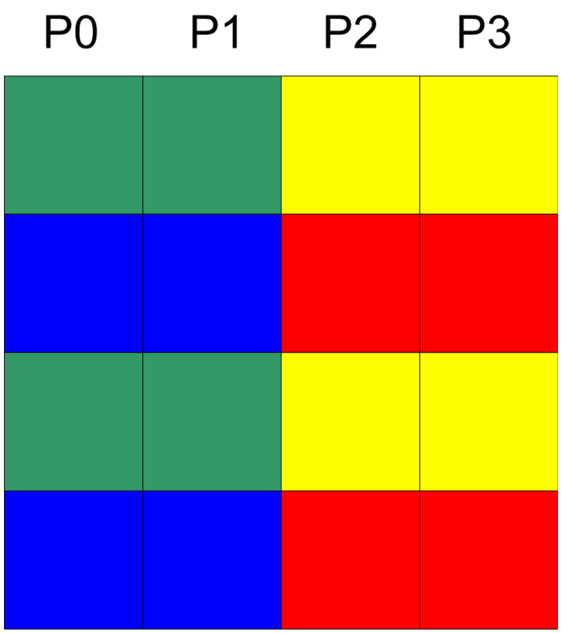
Flow of information



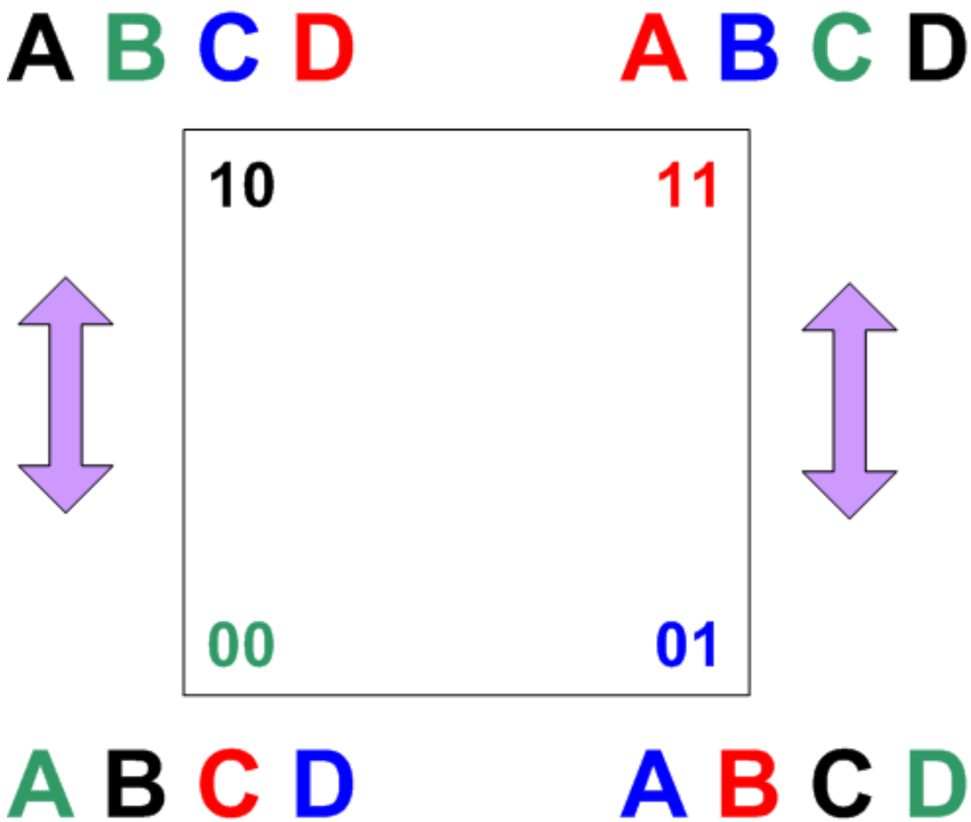


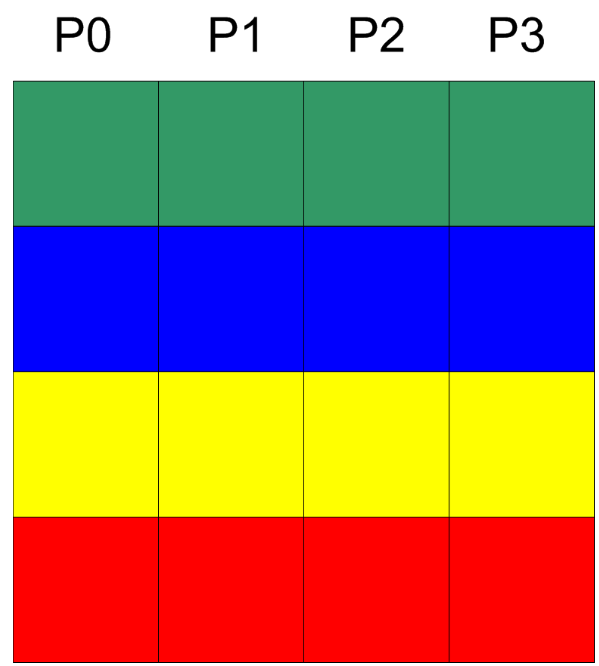
Flow of information





Flow of information





Summarizing all to all

- Short messages $\lceil \lg P \rceil \alpha$

- Long messages $\dfrac{P-1}{P} n\beta$

“Vector” All to ALl

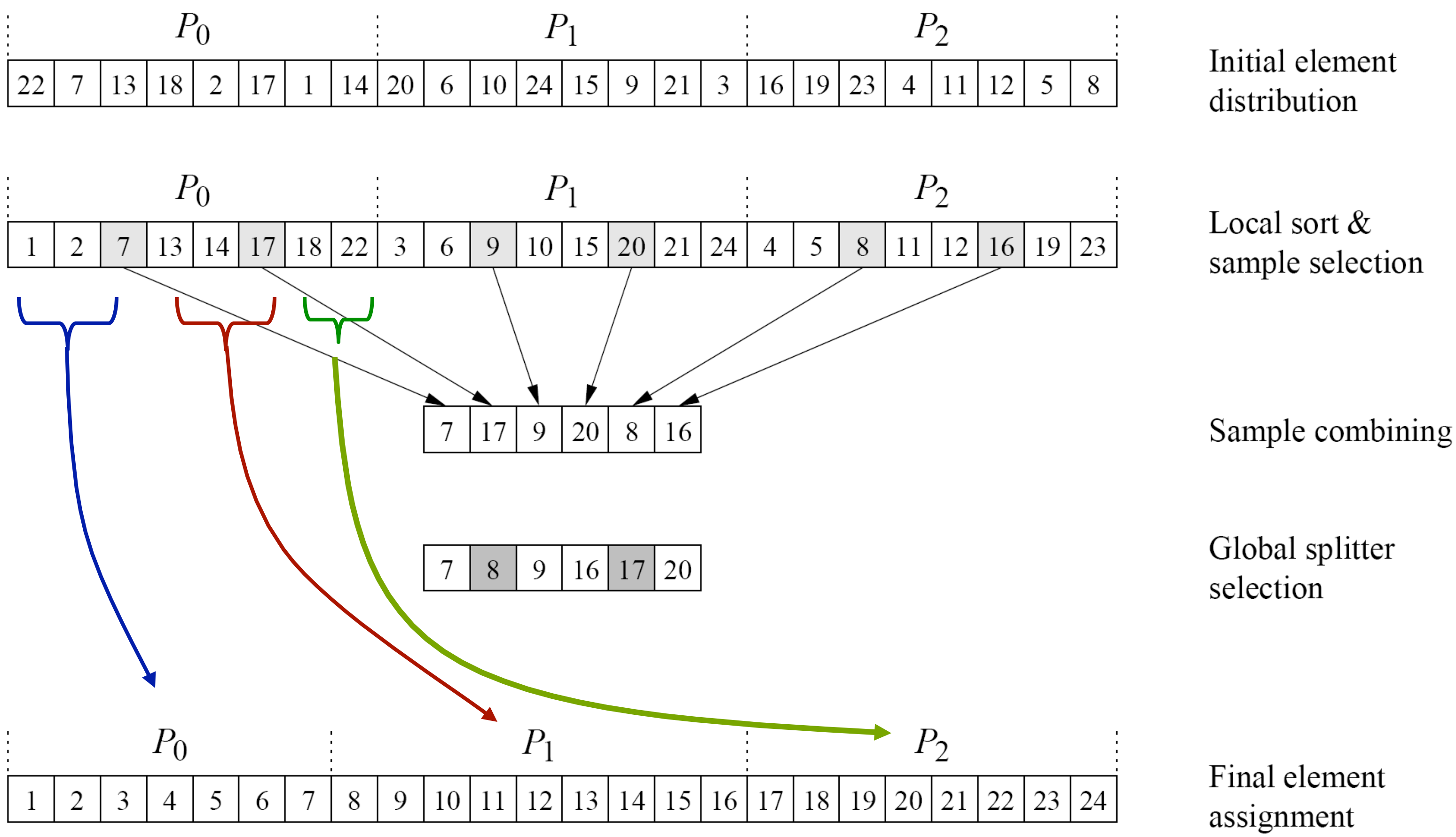
- Generalize all-to-all, gather, etc.
- Processes supply varying length datum
- Vector all-to-all

  MPI_Alltoallv (

      void *sendbuf, int sendcounts[], int sDispl [],
      MPI_Datatype sendtype,
      void* recvbuf,   int recvcnts[], int rDispl[],
      MPI_Datatype recvtype, MPI_Comm comm)

- Used in sample sort (coming)

Alltoallv used in sample sort



Introduction to Parallel Computing, 2nd Ed,, A.Grama, A.l Gupta, G. Karypis, and V. Kumar, Addison-Wesley, 2003.

Details of the algorithms

- Broadcast
- AllReduce
- Scatter/gather
- Allgather
- All to all
- Revisiting Broadcast

Revisiting Broadcast

- P may not be a power of 2
- We use a binomial tree algorithm
- We'll use the hypercube algorithm to illustrate the special case of $P=2^k$
- Hypercube algorithm is efficient for short messages
- We use a different algorithm for long messages

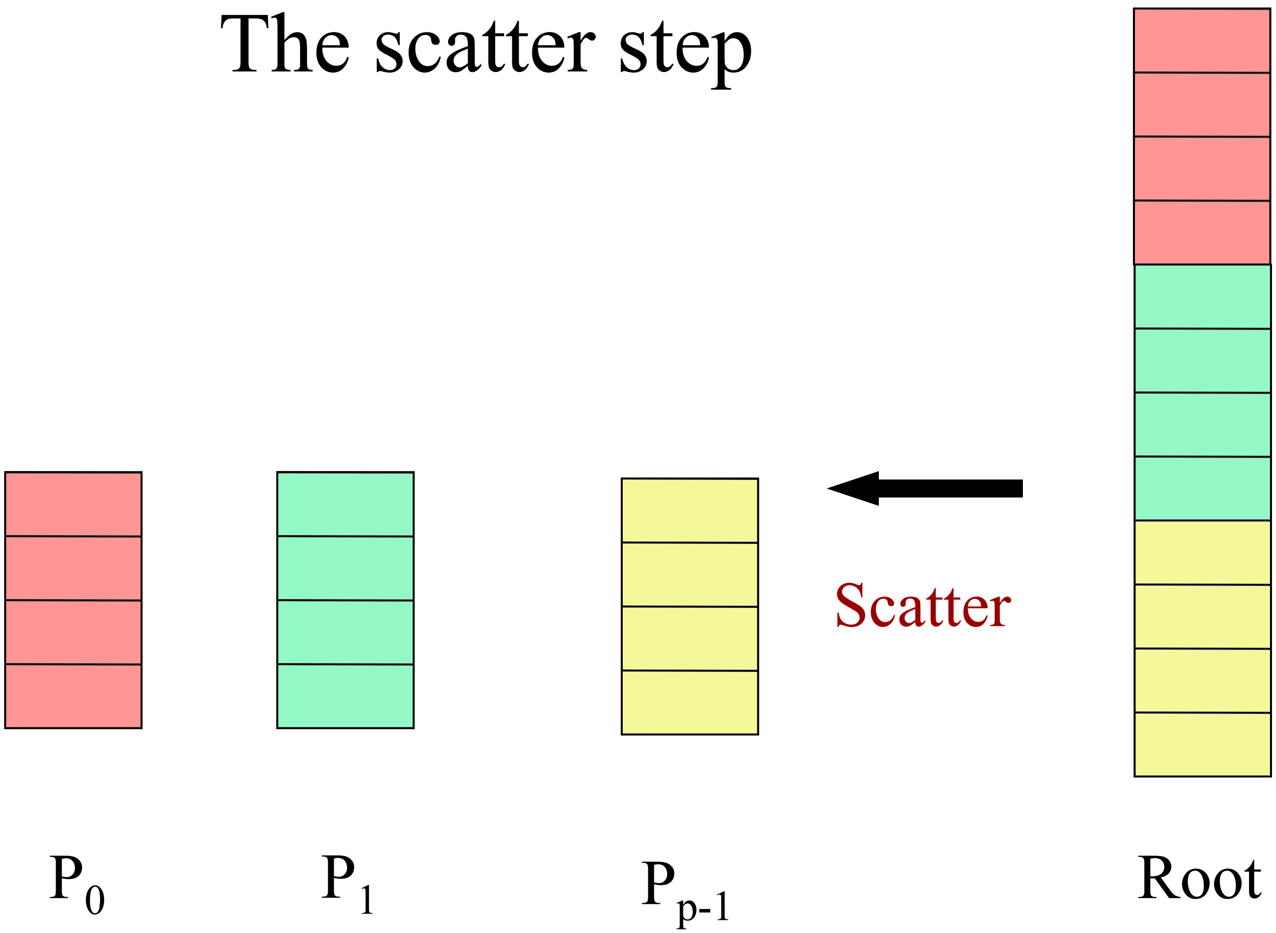
Strategy for long messages

- Based van de Geijn's strategy
- Scatter the data
  - Divide the data to be broadcast into pieces, and fill the machine with the pieces
- Do an Allgather
  - Now that everyone has a part of the entire result, collect on all processors
- Faster than MST algorithm for long messages

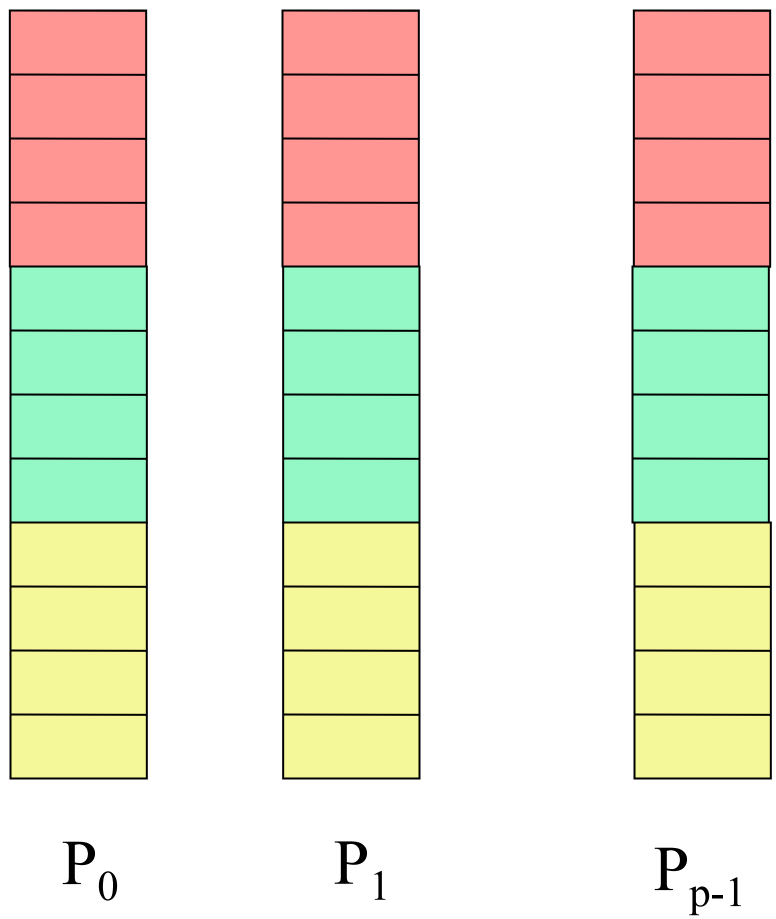
$$2\frac{p-1}{p}n\beta << \lceil \lg p \rceil n\beta$$

Algorithm for long messages

The scatter step



Algorithm for long messages



AllGather step

Today's lecture

- Collective Communication algorithms
- Sorting

Rank sorting

- Compute the rank of each input value
- Move each value in sorted position according to its rank
- On an ideal parallel computer, the forall loops parallelize perfectly

```
forall  i=0:n-1, j=0:n-1
     if ( x[i] > x[j] ) then rank[i] += 1 end if
forall  i=0:n-1
     y[rank[i]] = x[i]
```

In search of a fast and practical sort

- Rank sorting is impractical on real hardware
- Let’s borrow the concept: compute the processor owner for each key
- Communicate data in sorted order in one step
- But how do we know which processor is the owner?
- Depends on the distribution of keys

Bucket sort

- Divide key space into equal subranges and associate a bucket with each subrange
- Unsorted input data distributed evenly over processors
- Each processor maintains p local buckets
  - Assigns each key to a local bucket: $\lfloor p \times key/(K_{max}-1) \rfloor$
  - Routes the buckets to the correct owner
    (each local bucket has ~ **n/p^2** elements)
  - Sorts all incoming data into a single bucket

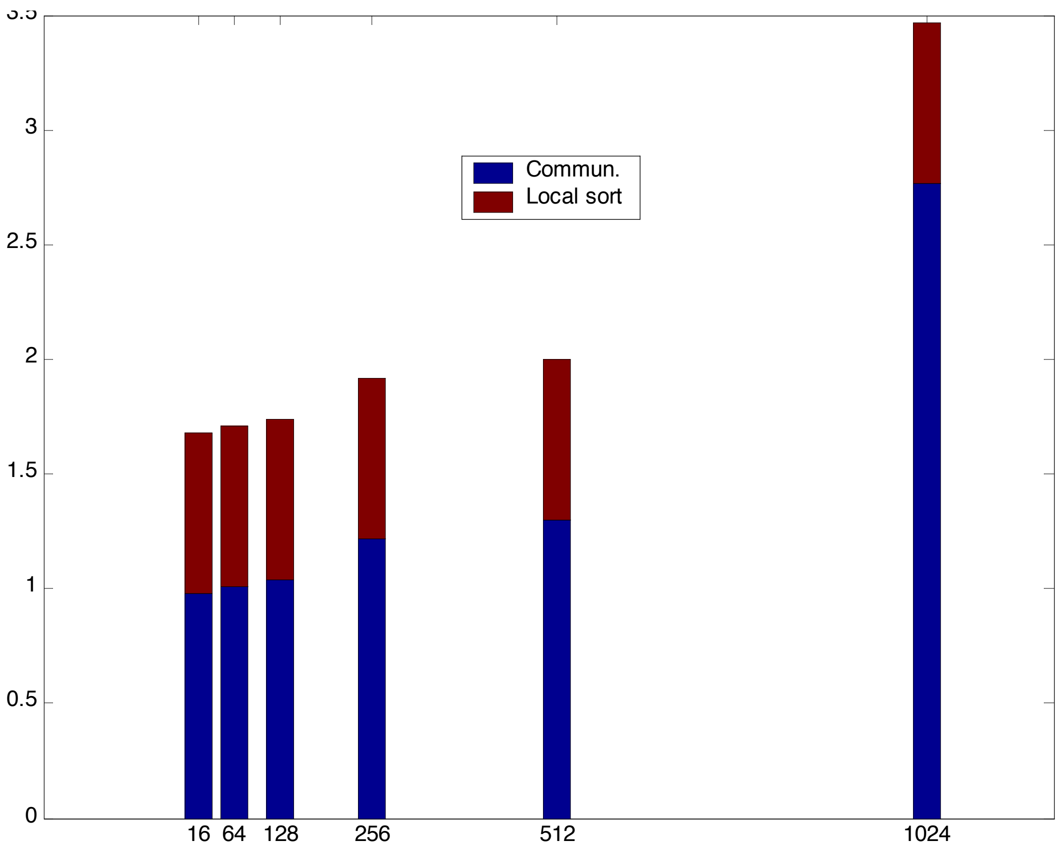
Wikipedia

Running time

- Assume that the keys are distributed uniformly over 0 to K_{max}-1
- Local bucket assignment: O(n/p)
- Route each local bucket to the correct owner
  All to all: O(n)
- Local sorting : O(n/p)
  - Radix sort
  - www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Radix

Scaling study

- IBM SP3 system: 16-way servers w/ Power 3 CPUs
- Weak scaling : 1M points per processor



Local sort: quicksort
O(n/p log(n/p))

All-to-allv
O(n)

Worst case behavior

- What is the worst case?
- Mapping of keys to processors based on knowledge of K_{max}
- If keys are in range [0,Q-1] …
  … processor *k* has keys in the range [k*Q/P : (k+1)*Q/P]
- For $Q=2^{30}$, P=64, each processor gets 2^{24} = 16 M elements
- What if keys $\in [0, 2^{24}-1] \subset [0, 2^{30}-1]$?
- But if they keys are distributed non-uniformly, we need more information to ensure that the keys (and communication) are balanced over the processors
- Sample sort is an algorithm that collects such information and improves worst case behavior

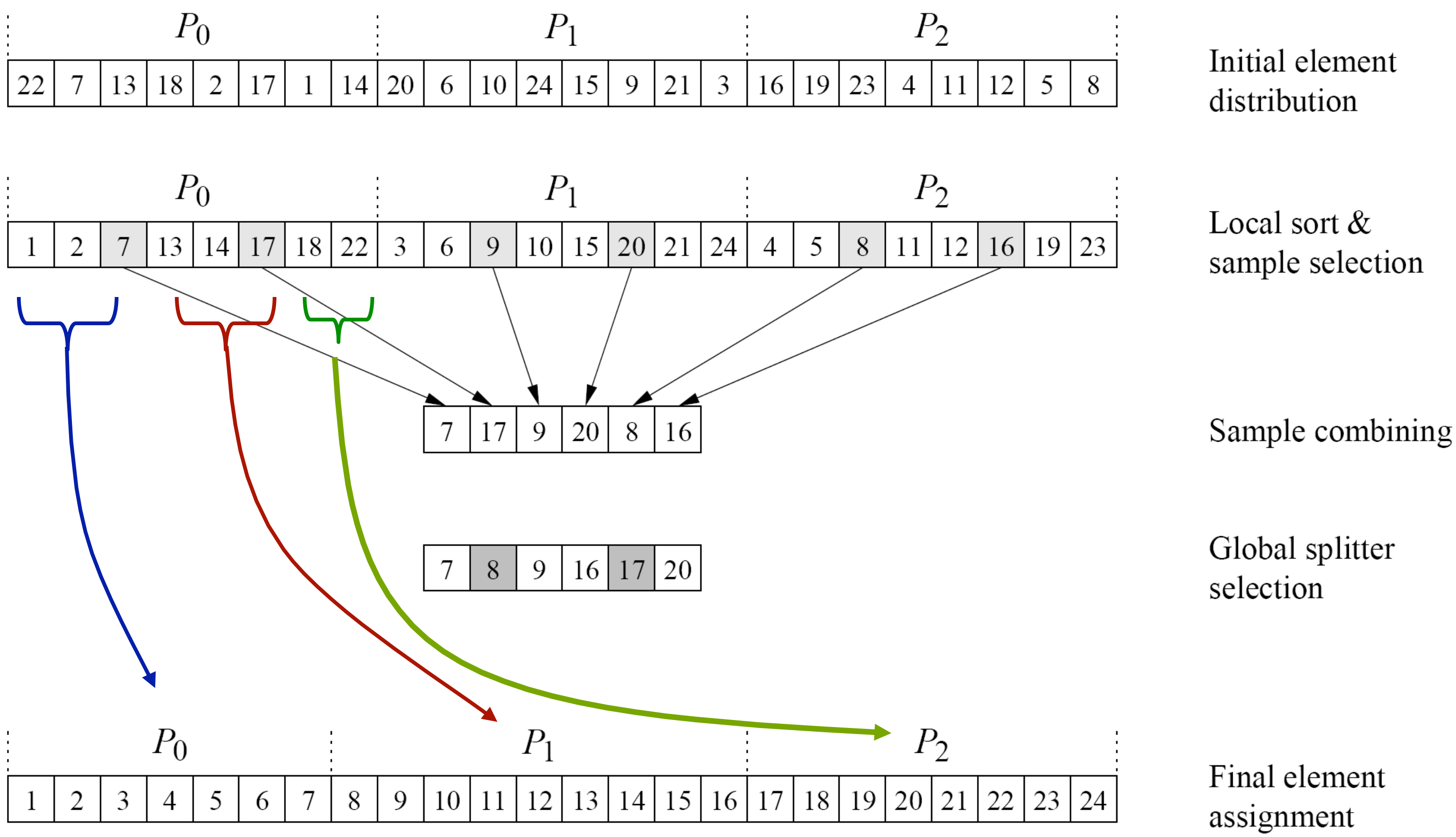
Improving on bucket sort

- *Sample sort* remedies the problem
- "Parallel Sorting by Regular Sampling." H. Shi and J. Schaeffer. J. Parallel and Distributed Computing, 14:361-372, 1992
- "Parallel Algorithms for Personalized Communication and Sorting With an Experimental Study."
  D. R. Helman, D.A. Bader, and J. JáJá,
  Proc. SPAA: Annual ACM Symp. on Parallel Algorithms and Architectures (1996)
  http://www.umiacs.umd.edu/research/EXPAR/papers/spaa96.html

The idea behind sample sort

- Use a heuristic to estimate the distribution of the global key range over the *p* processors processor so that…
- …each processor gets about the same number of keys
- Sample the keys to determine a set of *p-1* **splitters** that partition the key space into *p* disjoint intervals
  [sample size parameter: s]
- Each interval is assigned a unique processor mapped to a bucket
- Once each processor knows the splitters, it can distribute its keys to the others accordingly
- Processors sort incoming keys

Alltoallv used in sample sort



Introduction to Parallel Computing, 2nd Ed,, A.Grama, A.l Gupta, G. Karypis, and V. Kumar, Addison-Wesley, 2003.

Splitter selection: regular sampling

- After sorting local keys, each processor chooses *p* evenly spaced samples
- Each processor "deals" its sorted data into one of *p* bins
    - The k^{th} item is placed into position $\lceil k/p \rceil$ of bin k **mod** p
    - When done, each sends bin j to processor j
- This is like a transpose with block sizes = n/p^2
- Each processor receives p sorted subsequences
- Processor *p-1* determines the splitters
    - It samples each sorted subsequence, taking every $(kn/(p^2s))^{th}$ element $(1 \leq k \leq s-1)$, where $p \leq s \leq n/p^2$
    - Merges the sampled sequences, and collects p-1 regularly spaced splitters
    - Broadcasts the splitters to all processors
- Processors route (exchange) sorted subsequences according to the splitters (transpose)
- The data are unshuffled

Performance

- Assuming $n \geq p^3$ and $p \leq s \leq n/p^2$
- Running time is $\approx O((n/p) \lg n)$
- With high probability …
  no processor holds more than $(n/p + n/s - p)$ elements
- Duplicates d do not impact performance unless $d = O(n/p)$
- Tradeoff: increasing s …
  - Spreads the final distribution more evenly over the processors
  - Increases the cost of determining the splitters
- For some inputs, communication patterns can be highly irregular with some pairs of processors communicating more heavily than others
- This imbalance degrades communication performance

The collective calls

- Processes transmit varying amounts of information to the other processes
- This is an MPI_Alltoallv

(SKeys, send_counts, send_displace, MPI_INT, RKeys, recv_counts, recv_displace, MPI_INT, MPI_COMM_WORLD)

- Prior to making this call, all processes must cooperate to determine how much information they will exchange
  - The *send list* describes the number of keys to send to each process k, and the offset in the local array
  - The *receive list* describes the number of incoming keys for each process k and the offset into the local array

Determining the send and receive lists

- After sorting, each process scans its local keys from left to right, marking where the splitters divide the keys, in terms of send counts
- Perform an all to all to transpose these send counts into receive counts

```
MPI_Alltoall(send_counts, 1, MPI_INT,
             recv_counts, 1, MPI_INT,MPI_COMM_WORLD)
```

- A simple loop determines the displacements

```
for (p=1; p < nodes; p++){
    s_displ[p] = s_displ[p-1] + send_counts[p-1];
    r_displ[p] = r_displ[p-1] + rend_counts[p-1];
}
```

Fin