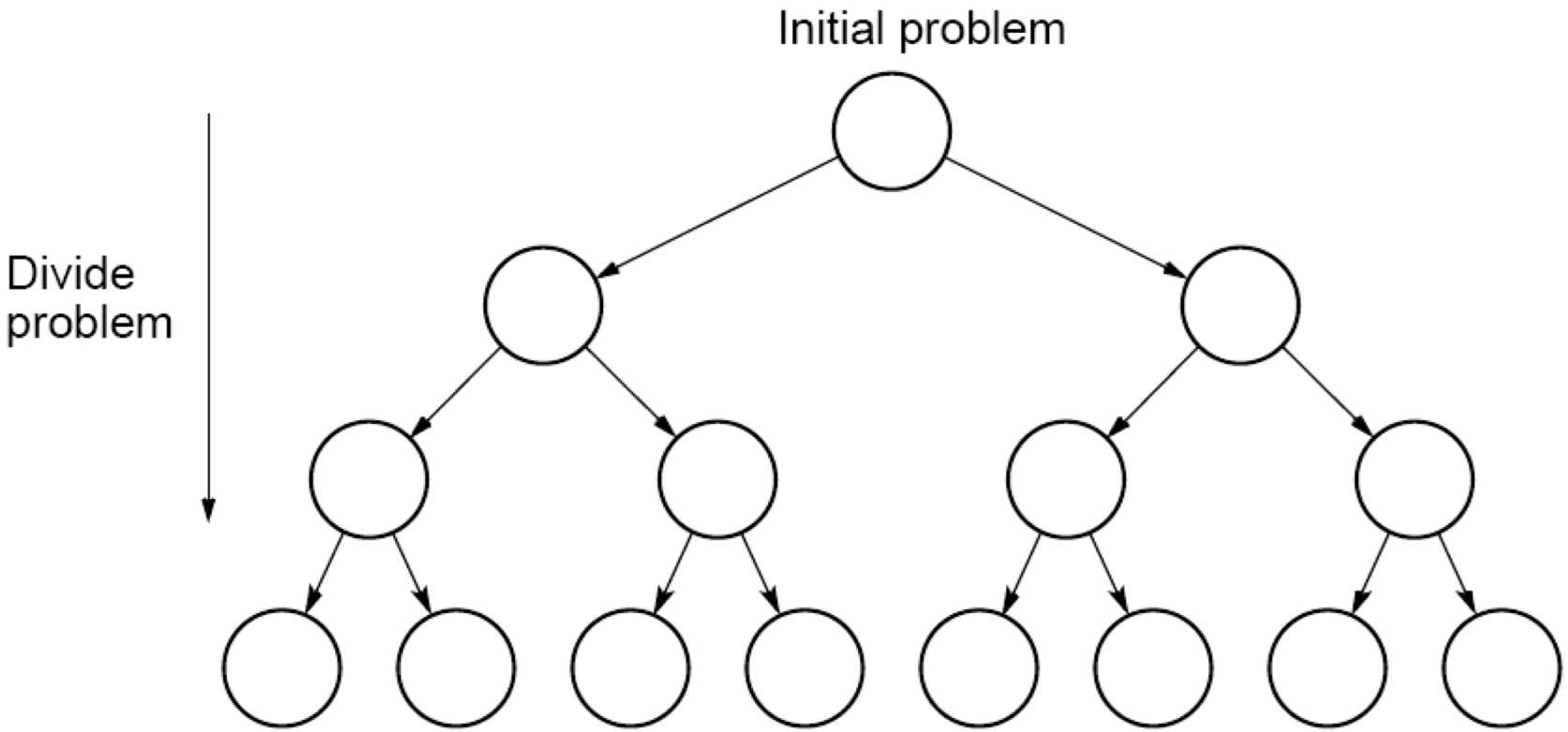
Partitioning and Divide-and-Conquer Strategies – Part I

CPSC 424/524
Lecture #10
October 31, 2018

Partitioning vs. Divide-and-Conquer

- Partitioning
  - Divides a problem into parts
  - Solves the individual parts
  - Combines the results
  - Parts may be completely dissimilar
- (Recursive) Divide-and-Conquer
  - Special case of partitioning
  - Partitioned sub-problems have same form as the original problem
  - Partitioning repeated recursively; often represented in tree form
  - Ideally, data usage for each sub-problem is localized

Recursive Partitioning



Examples

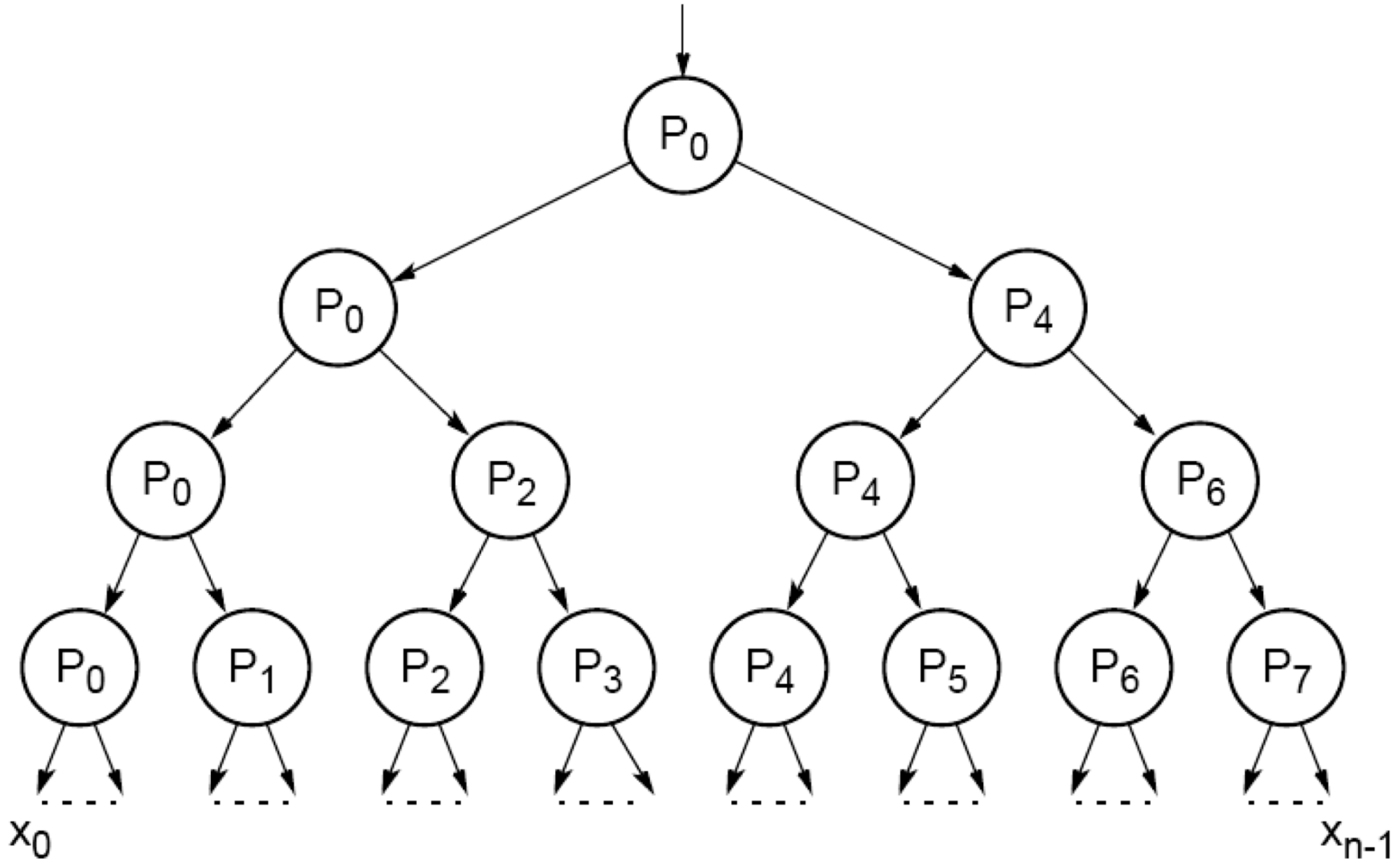
- Arithmetic operations on sequences of numbers (Reductions)
- Matrix multiplication in block form
- Sorting numbers (more later)
- Numerical Integration (more later)
- N-body problem (more later)
- Database searching
- Sequence comparison
- Grid computations (e.g. Partial differential equations)

Example: Summing a sequence of numbers

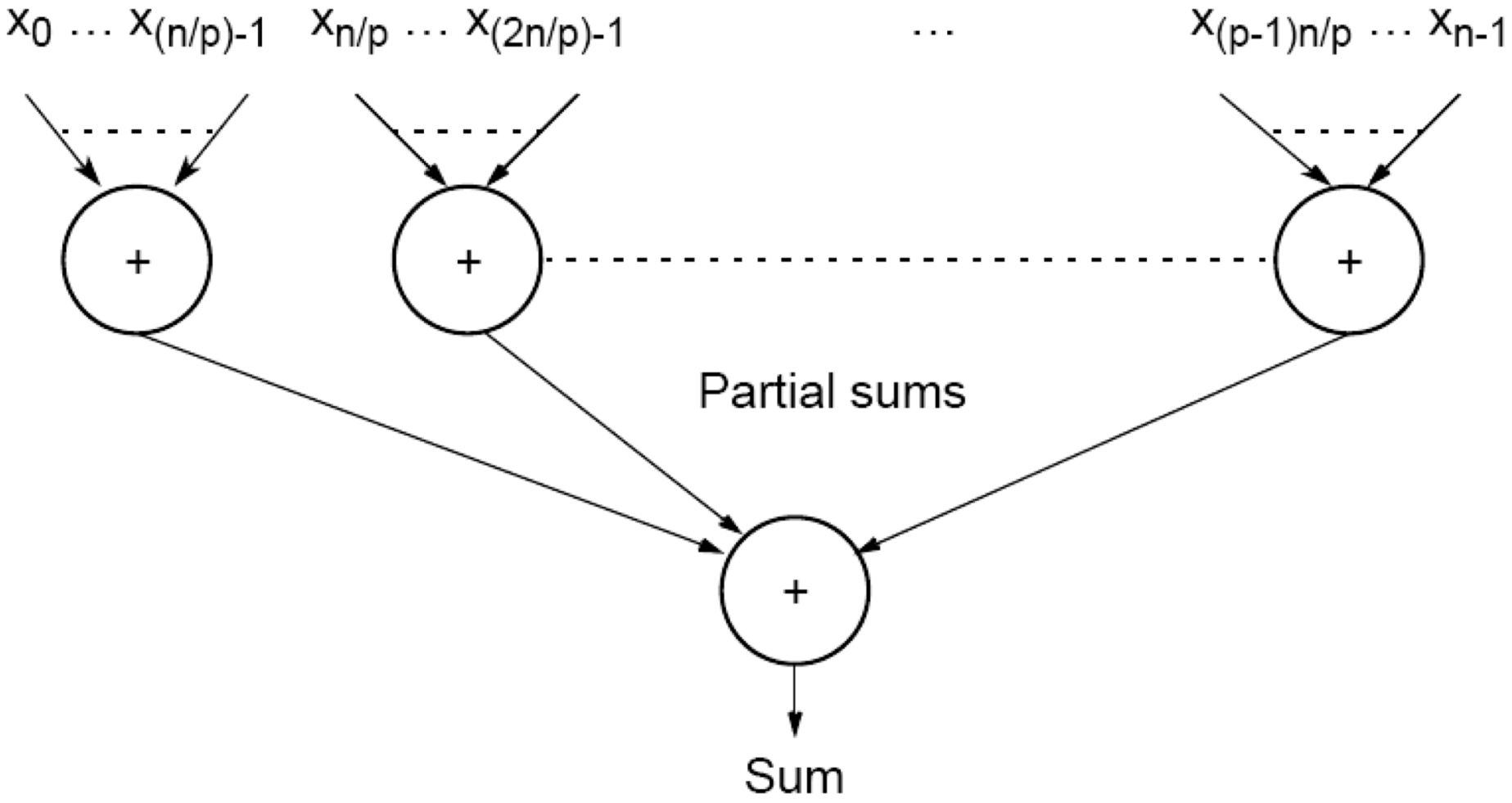
$$S = \sum_{i=0}^{n-1} x_i$$

Partitioning the Sequence

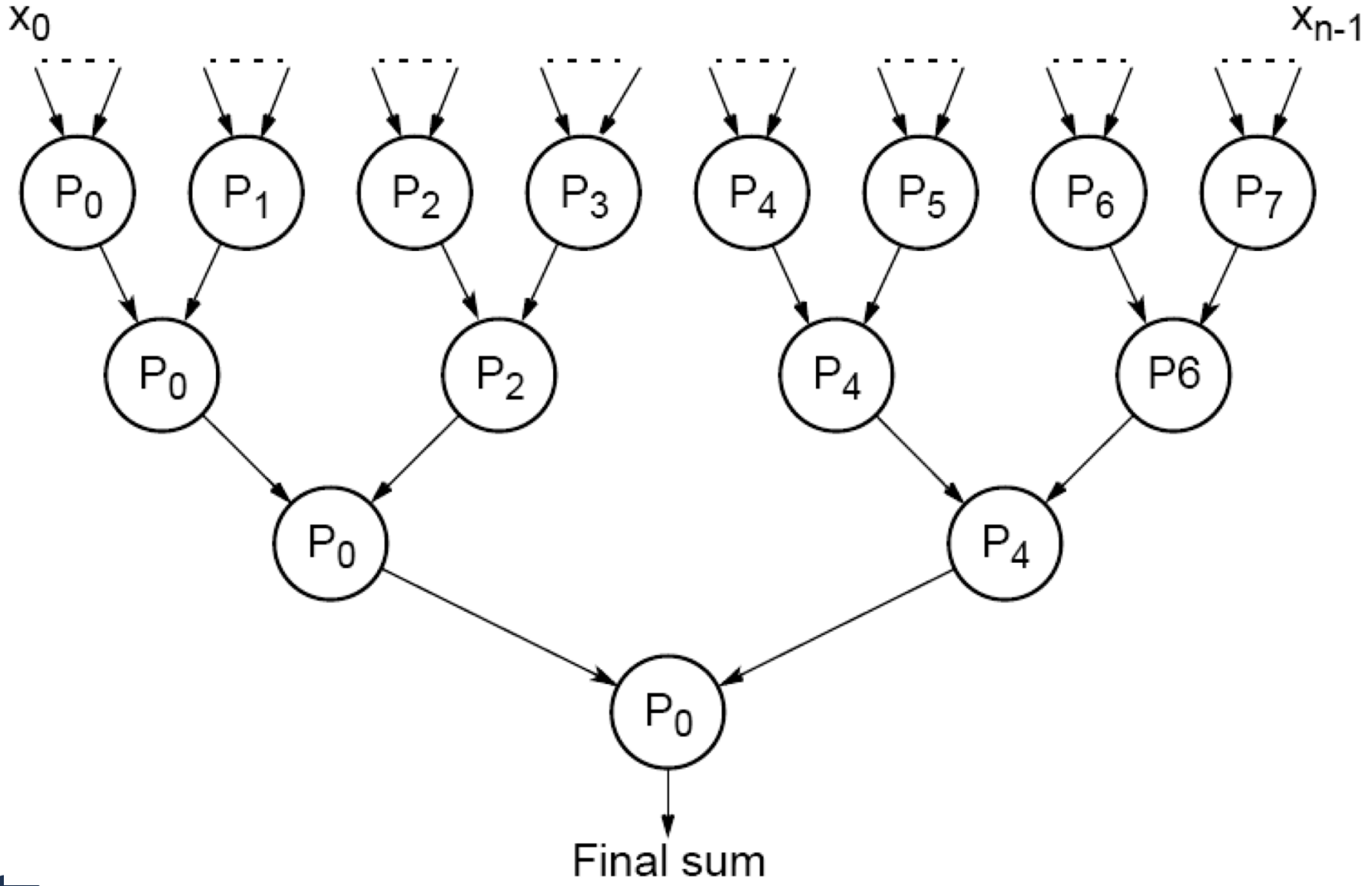
Original Sequence: $x_0, x_1, x_2, \ldots, x_{n-1}$



Summing the Partitioned Sequence



Carrying Out the Summation



Sorting Algorithms

Many sorting algorithms can be parallelized using divide-and-conquer.

Examples:

- Bucket sort

- Quicksort

  Start with n numbers (assume no duplicates)

  Select one at random, say x_s

  Split the problem into two parts:

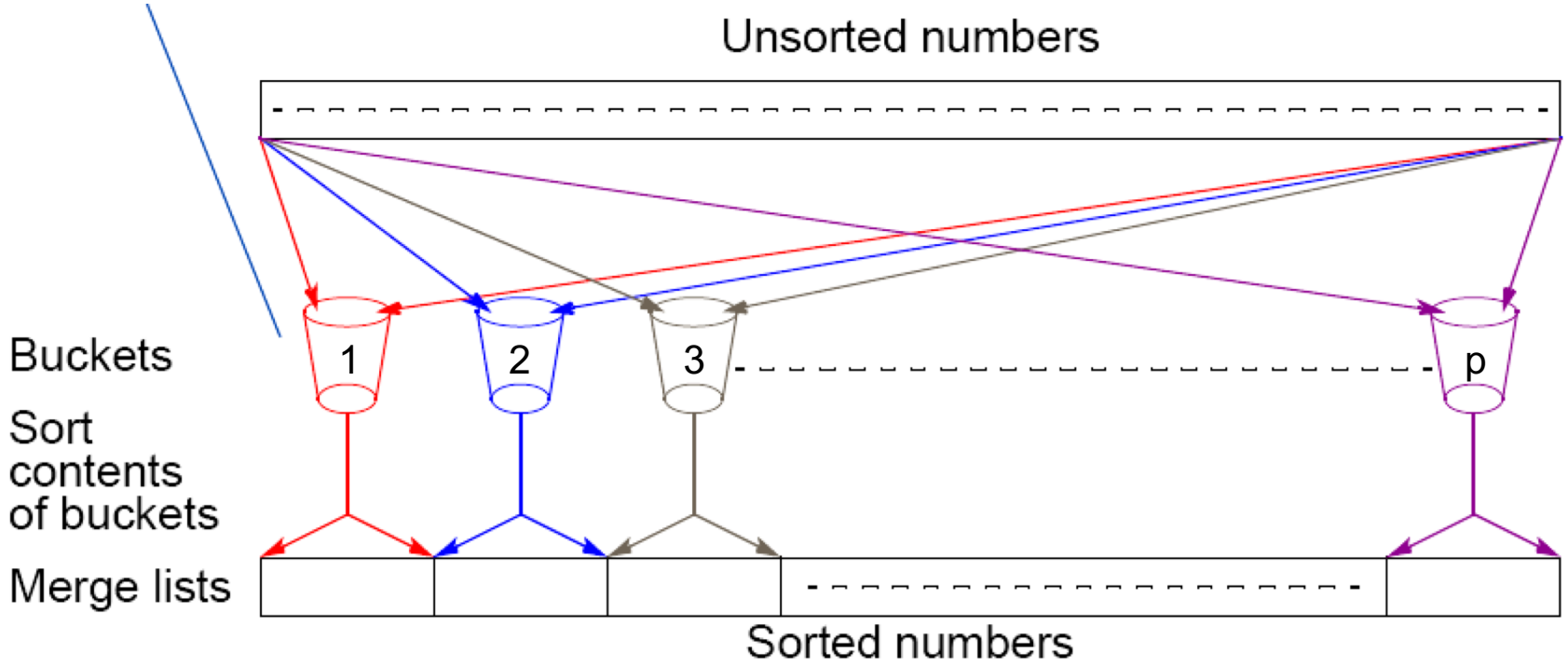
  $S_<$: those numbers < x_s

  $S_>$: those numbers > x_s

  Repeat until the subproblems are small enough

Bucket Sort

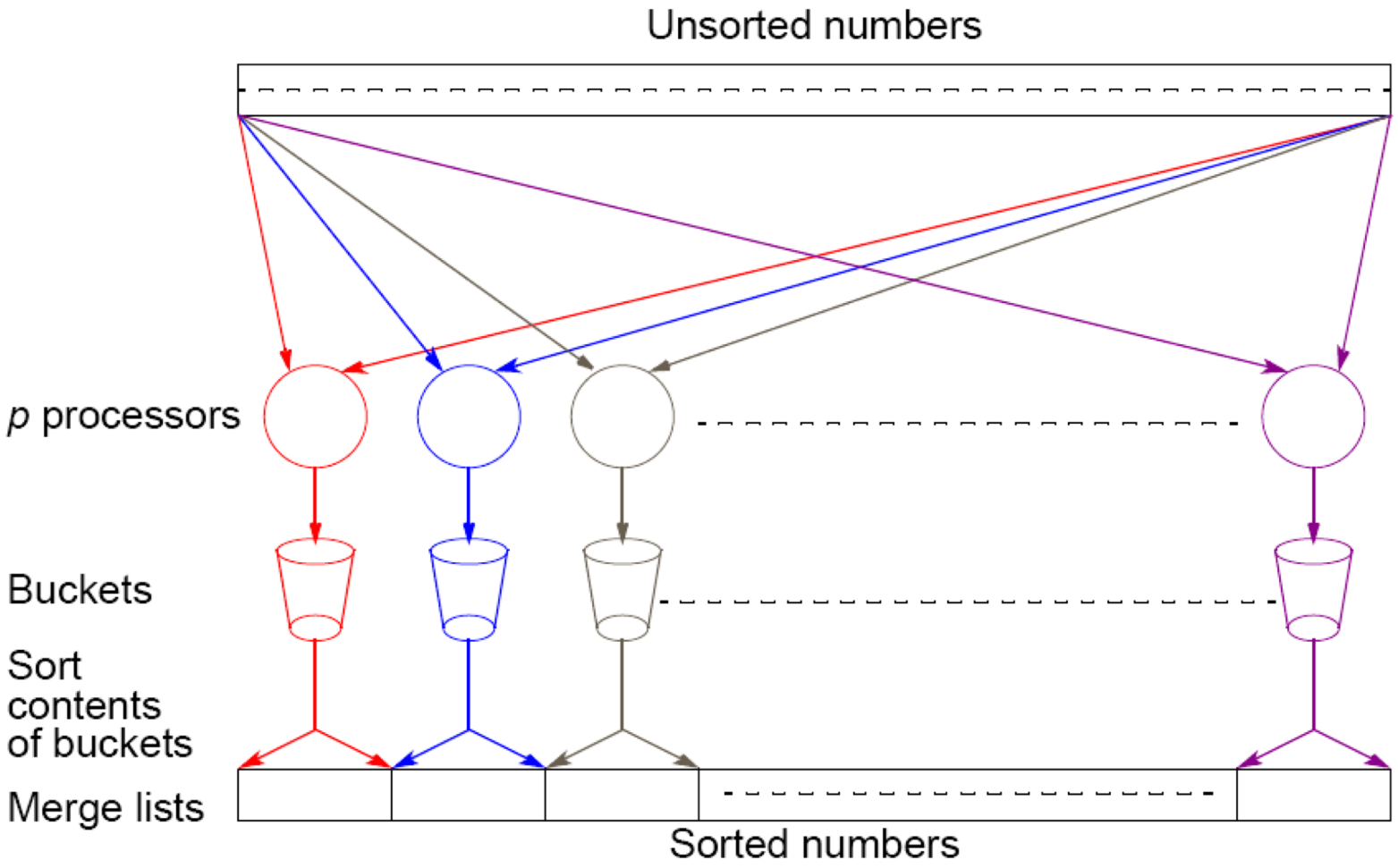
One “bucket” assigned to hold numbers that fall within each region.
Numbers in each bucket sorted using a sequential sorting algorithm.



Sequential sorting time: O(p·(n/p)·log(n/p)) for p buckets (best case).
Works well if the original numbers uniformly distributed across a known interval, say 0 to M-1.

Simple Parallel Version of Bucket Sort

Simple Approach: Assign one processor for each bucket.
(Each process looks at all numbers in first phase.)

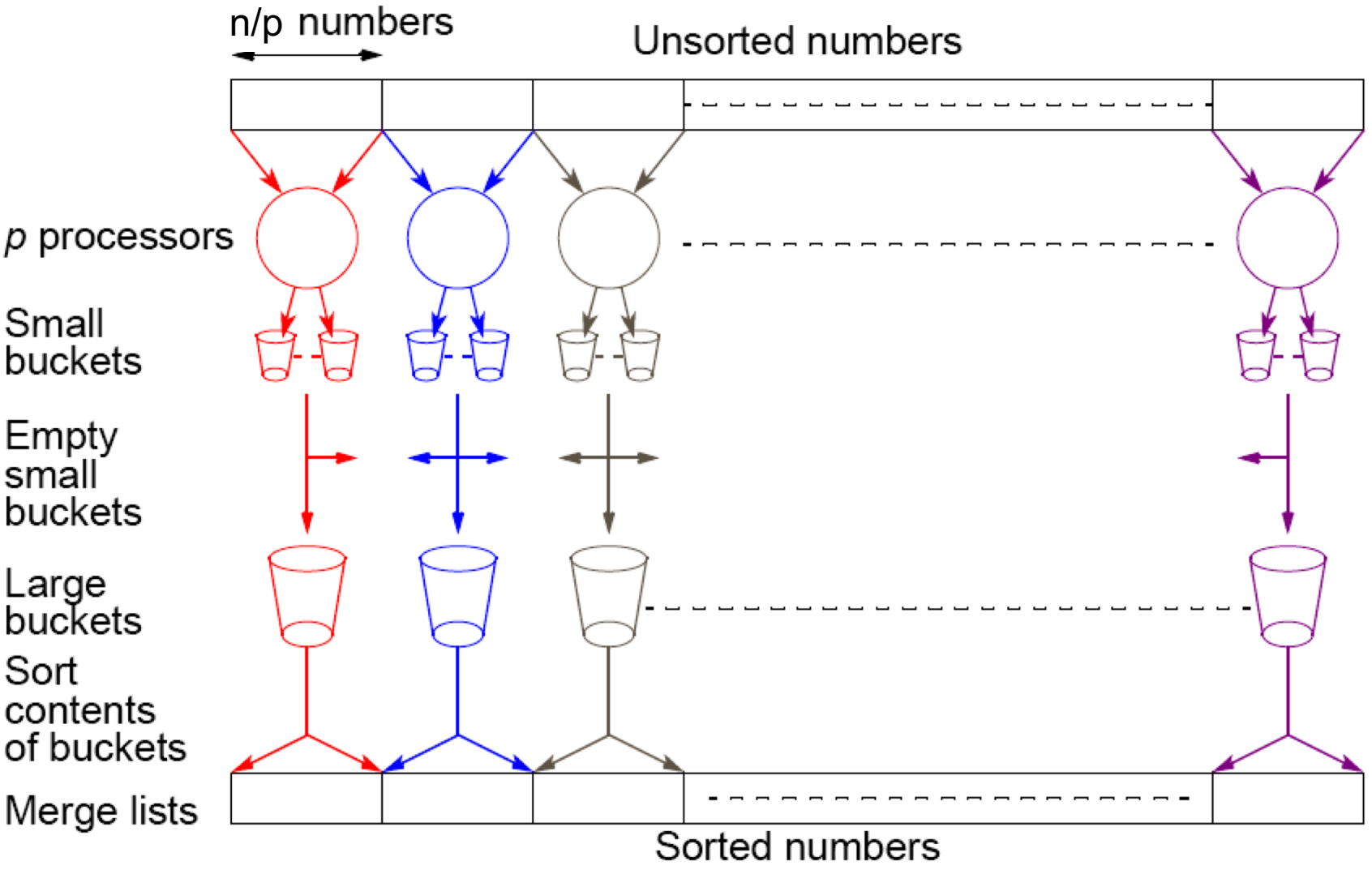


Better Parallelization

Using *p* processors to sort *n* numbers:

- Partition sequence range into *p* disjoint intervals, 1 per processor
- Give each processor ~*n/p* (randomly selected) numbers
- Each processor "deals" its numbers into local buckets:
  - A "large" bucket for its own interval (which it keeps)
  - *p-1* "small" buckets for other intervals
    (which will be sent to the processors responsible for those intervals)
- Processors exchange the small buckets
- Each processor sorts all the numbers it has
- Master merges results
  - Easy, since sorted buckets correspond to disjoint, ordered intervals

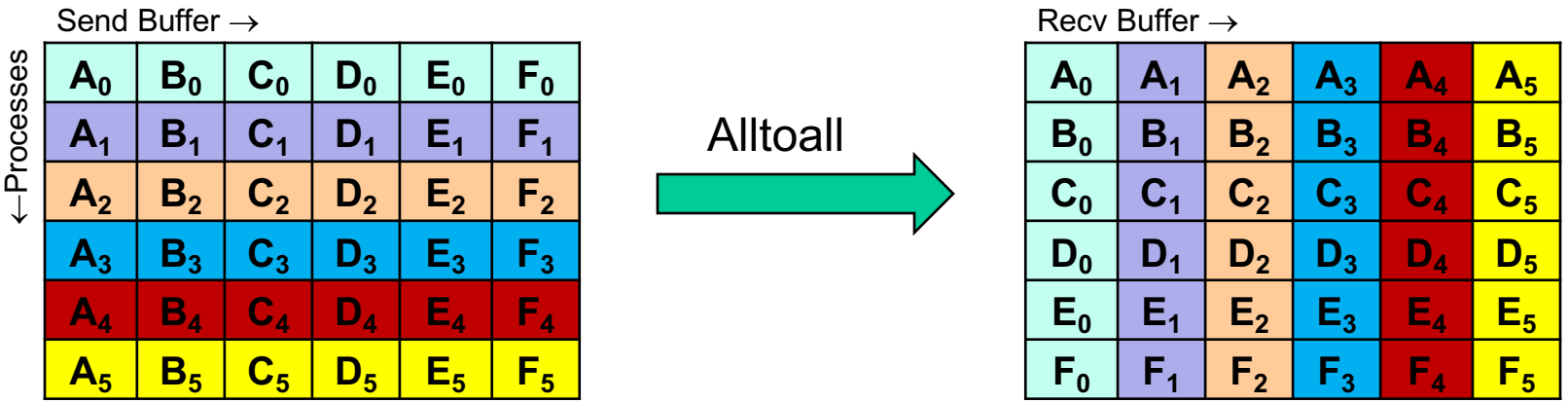
Parallel Bucket Sort

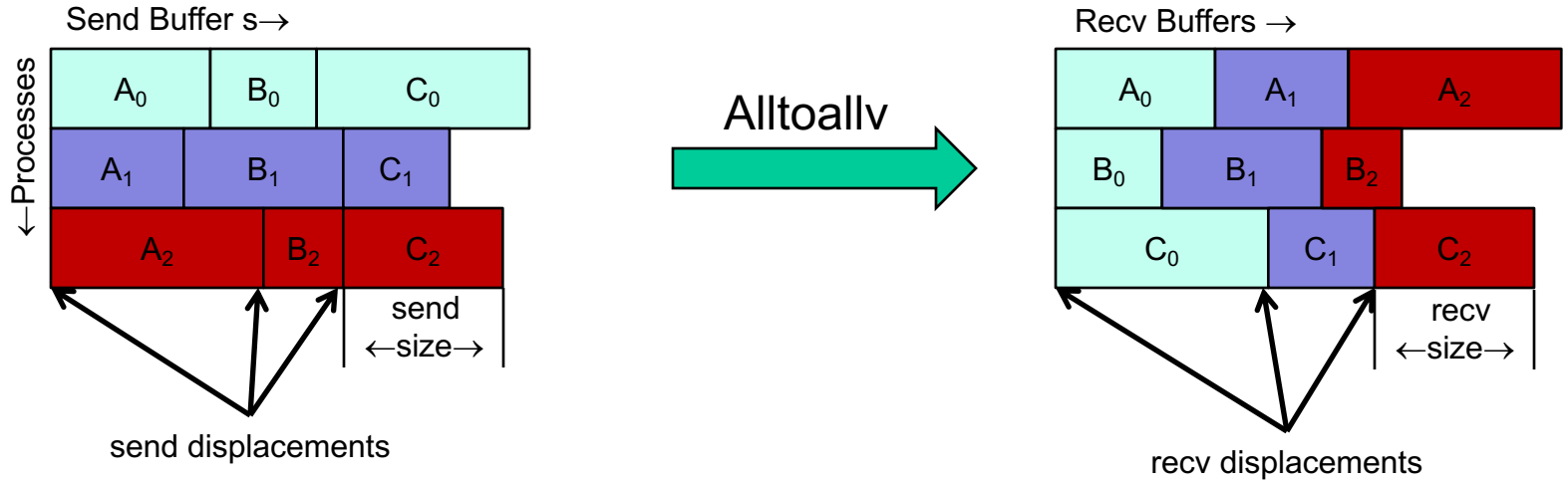


Generalized Gather/Scatter-like Operations

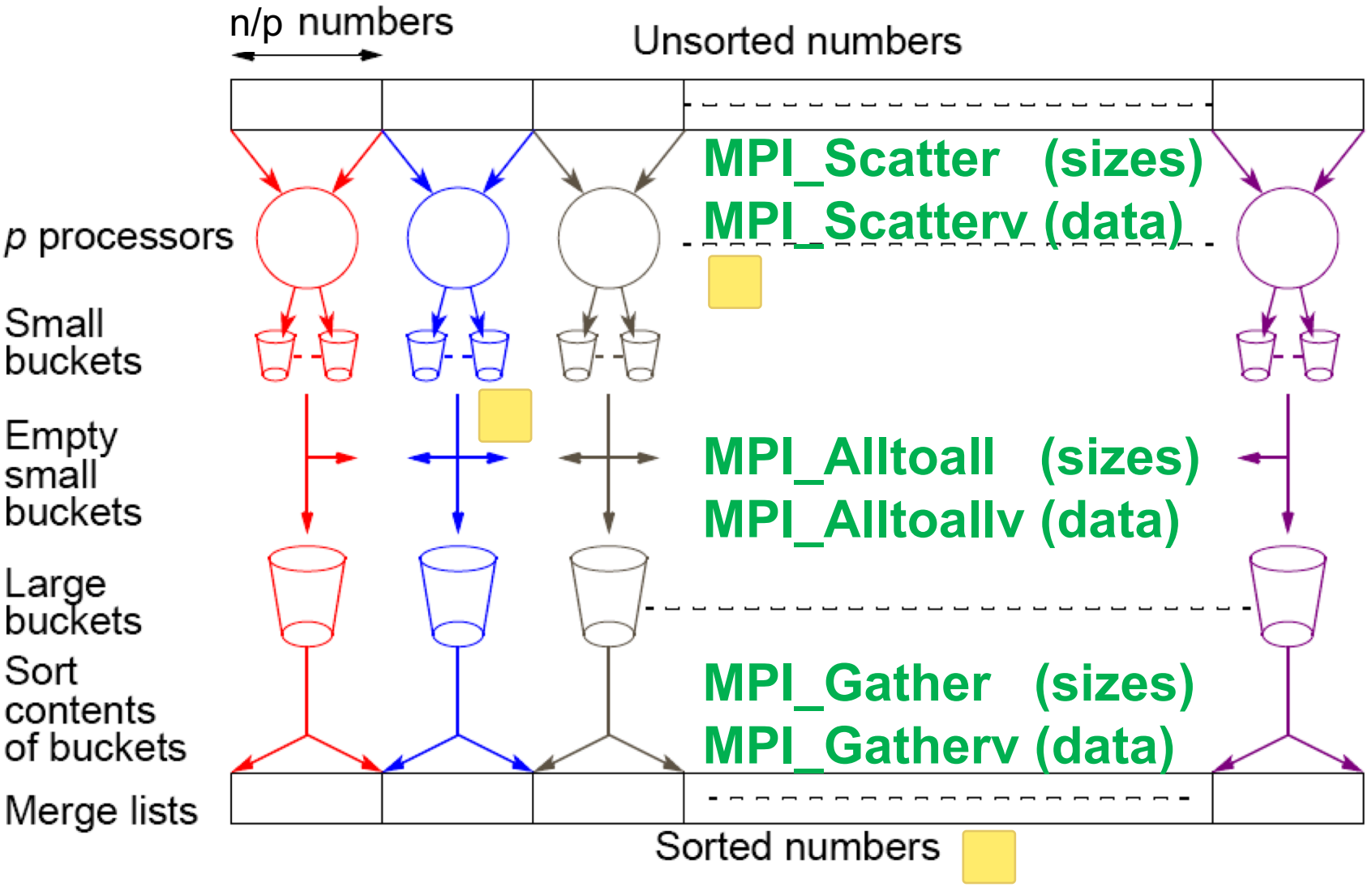
- Vector versions: Permit varying segment sizes & displacements
  - `MPI_Scatterv`
  - `MPI_Gatherv`
  - `MPI_Allgatherv`
  - `MPI_Alltoallv`
- "W" Version: Extends "V" version with varying data types
  - `MPI_Alltoallw`
  - Can be specialized to act like "W" versions of other collectives
- Intent is to allow flexibility in the location of the data on the send and/or recv side of the operations. (Obviously useful in cases like bucket sort.)

Alltoall and Alltoallv





Alternative Bucket Sort

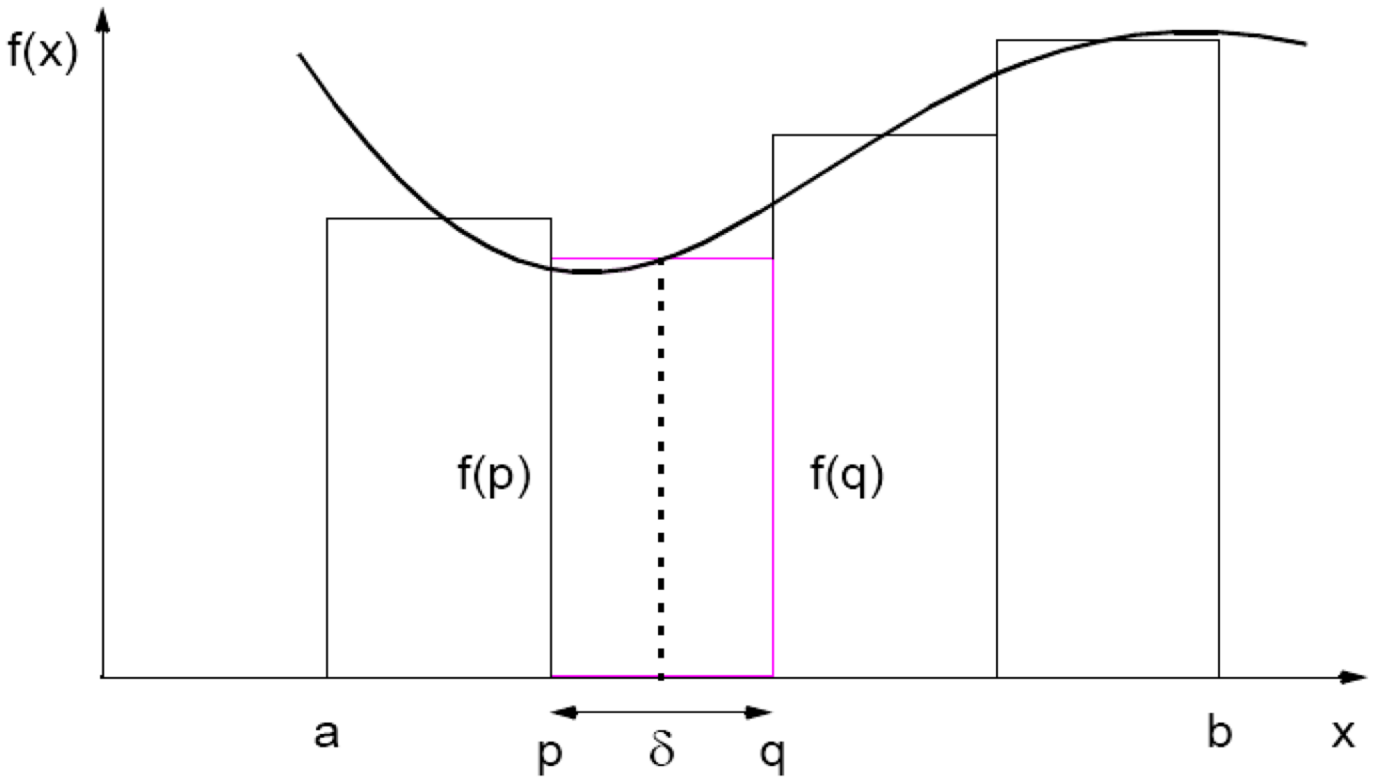


Collectives for Bucket Sort

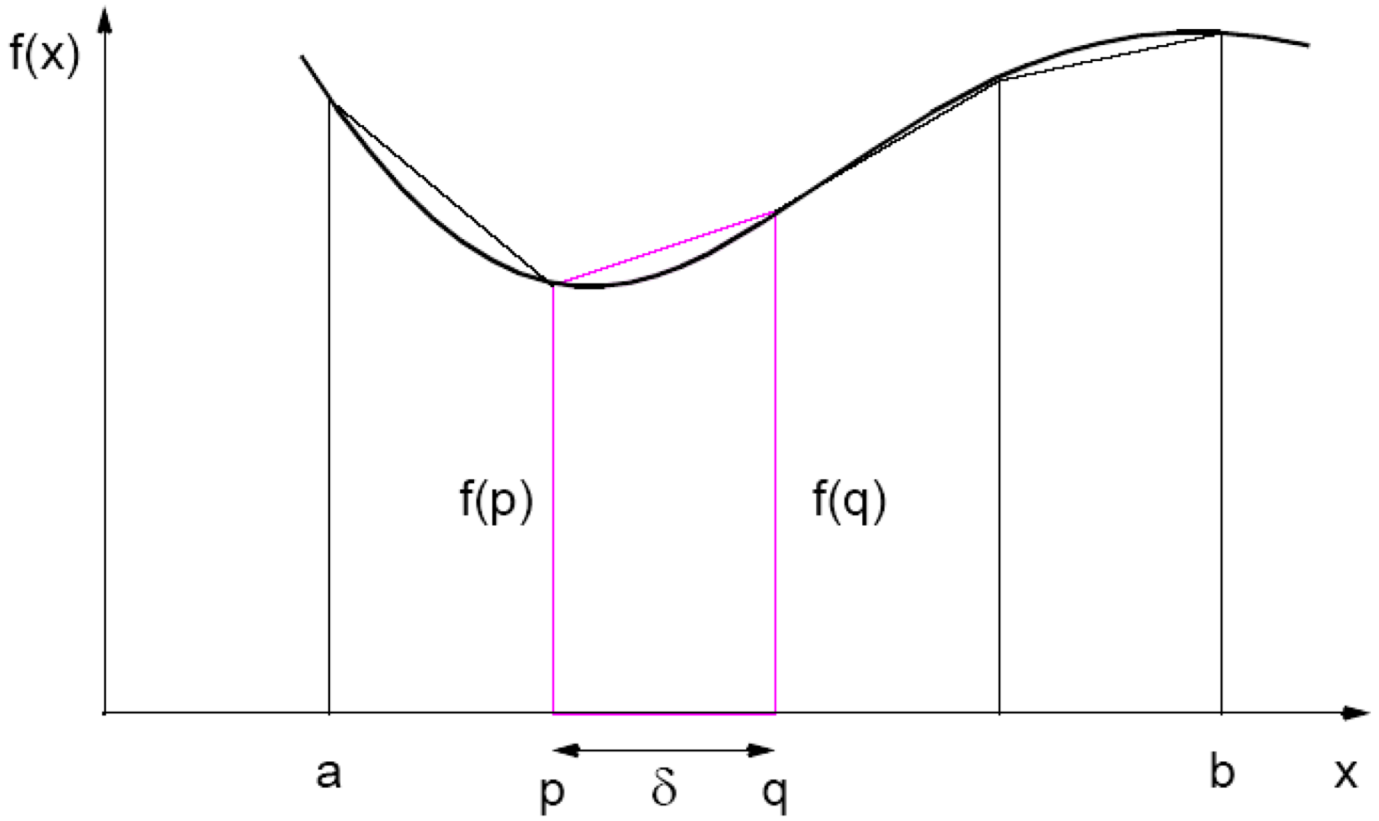
```
int size, *recvbuf, *sendbuf;
int sendsizes[p], senddisps[p], recvsizes[p], recvdisps[p];
    . . .
//sendsizes[i] = count of data items to be sent to process i
//                (comes from local partitioning)
//senddisps[i] = displacement in sendbuf of data for process i
//                (computed based on sendsizes)

MPI_Alltoall(sendsizes, 1, MPI_INT, recvsizes, 1, MPI_INT,
             MPI_COMM_WORLD);

//recvdisps[i] is displacement in recvbuf for data from proc i
for (i=1,recvdisps[0]=0; i<p; i++)
    recvdisps[i] = recvdisps[i-1] + recvsizes[i-1];

MPI_Alltoallv(sendbuf, sendsizes, senddisps, MPI_INT,
              recvbuf, recvsizes, recvdisps, MPI_INT,
              MPI_COMM_WORLD);
```

Numerical integration using midpoint rule

Each region calculated using an approximation given by rectangles:

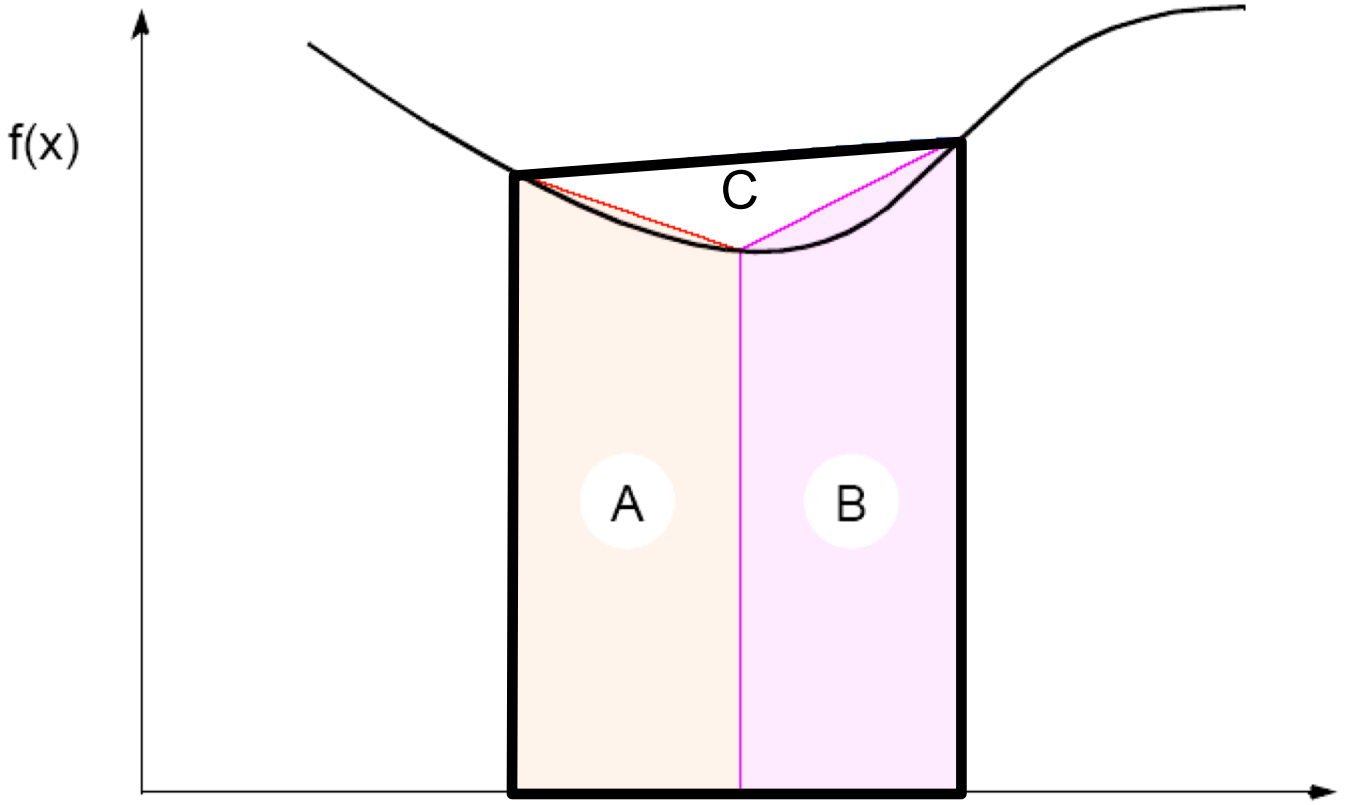


Numerical integration using trapezoidal rule



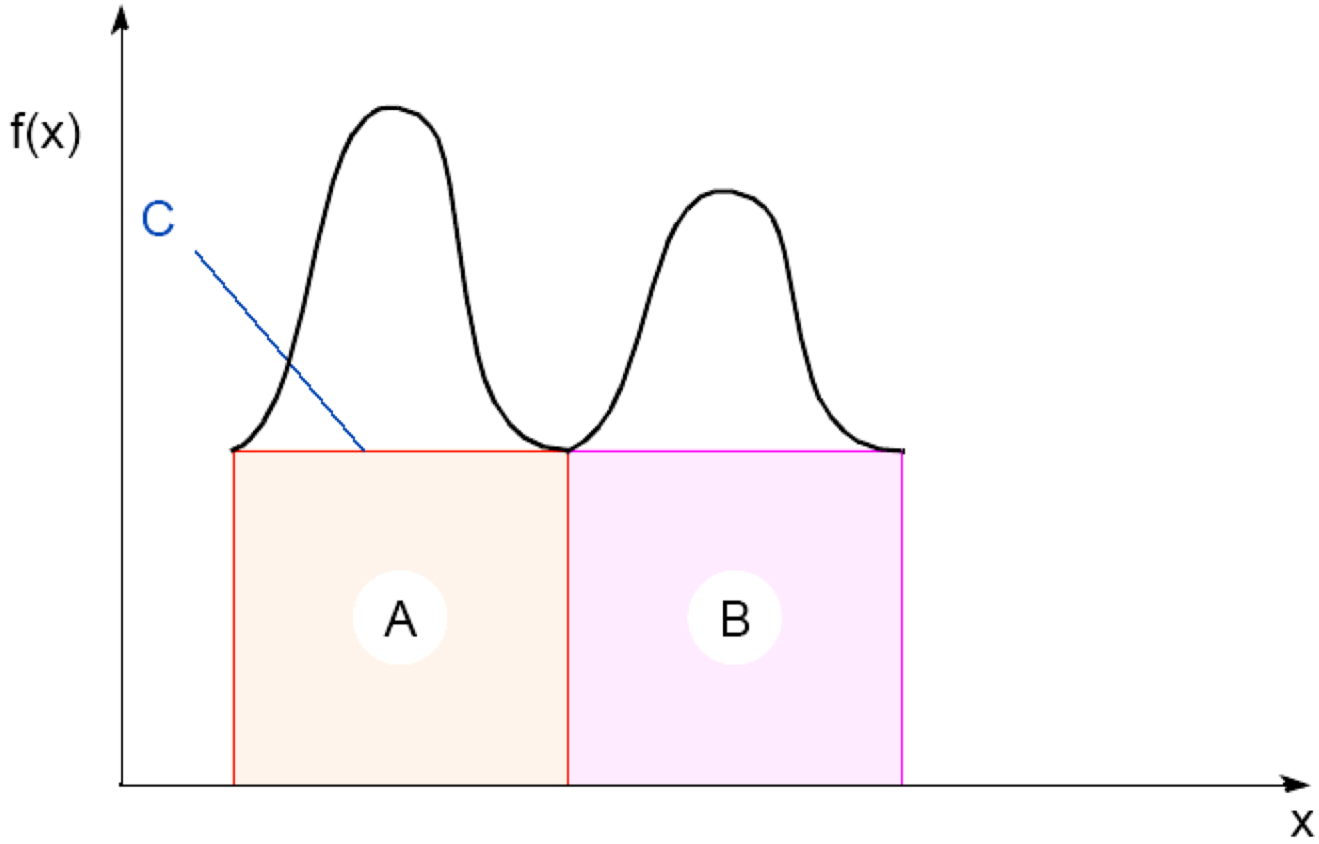
Adaptive Quadrature

Solution adapts to shape of curve. Use three areas, *A*, *B*, and *C* (from the previous interval). Computation terminated when *A* + *B* sufficiently close to *C*.



Termination Issues

Some care might be needed in choosing when to terminate.



Might cause early termination, as two large regions are the same (i.e., $C = A+B$).