Notes on Parallel Sort

Parallel Computer Architecture and Programming

CMU 15-418/15-618, Spring 2015

Parallel sort API

```
void parallelSort(
    float *data, float *&sortedData,
    int procs, int procId,
    size_t dataSize, size_t &localSize ) {
  // Implement parallel sort algorithm as
  // described in assignment 3 handout.

  localSize = 0;
  return;
}
```

Inputs:

data: Input array (a[n/p])

procs: Number of processes (p)

procId: This process id (i)

dataSize: Aggregate data size (n)

localSize: Size of data on process i (~n/p)

Outputs:

sortedData: Sorted array (sorted)

localSize: Size of sorted data on process I

Important: set localSize to sortedData array size to pass the result checking, 0 to skip.

Parallel sort using MPI

Step 1: Choosing pivots to define buckets

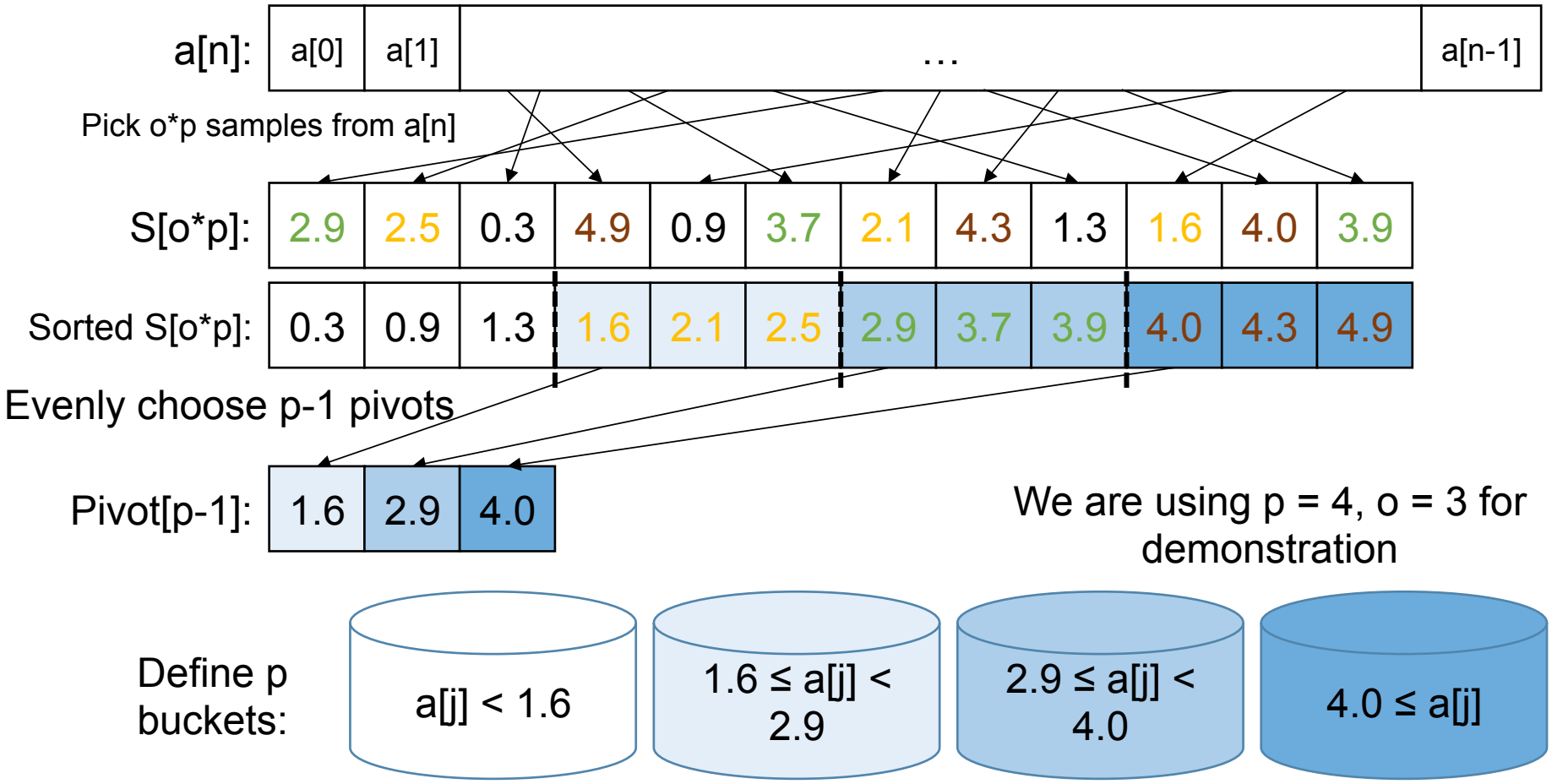
Step 2: Bucketing elements of the input array

Step 3: Redistributing elements

Step 4: Final local sort

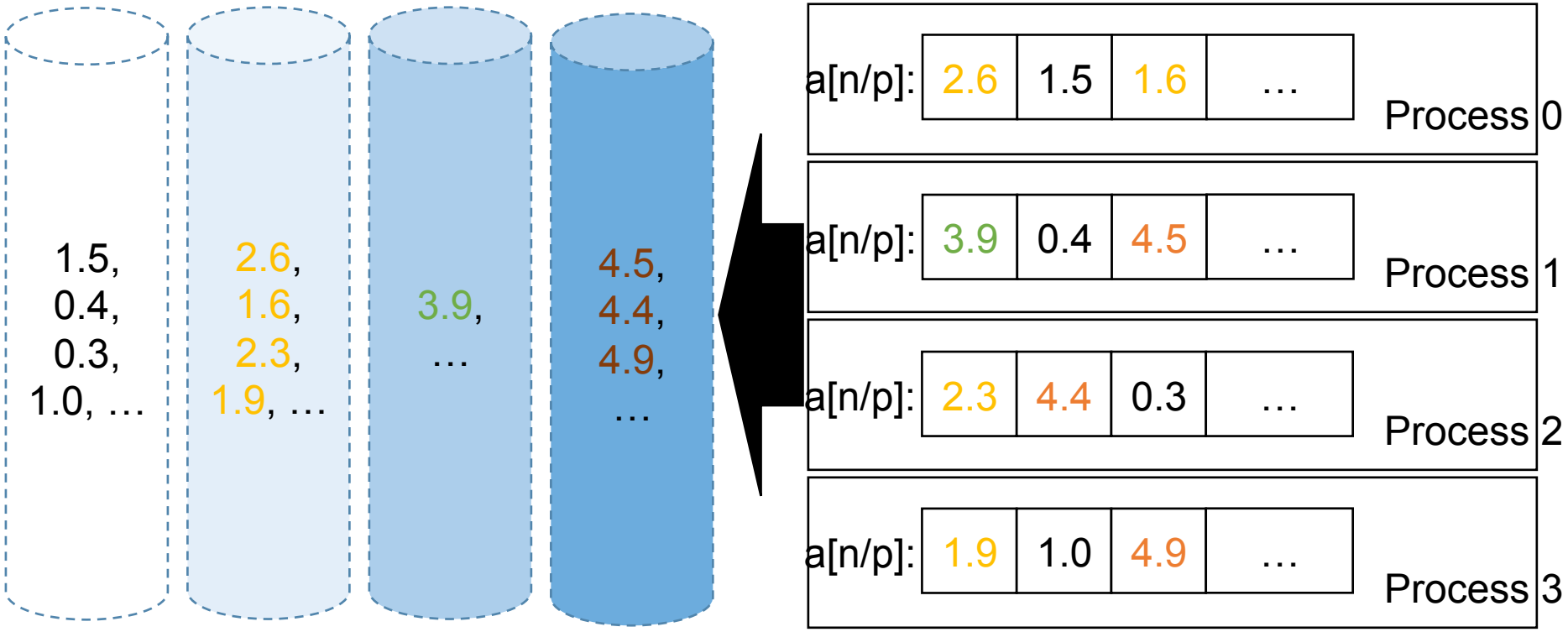
Warning: This is only a sketch of the algorithm, not implementation (Think of how you will implement this with MPI)

Step 1: Choosing pivots to define buckets



a[n]: Input array S[o*p]: Sample array o: Oversample n = dataSize p = procs
Tip for o: our reference solution use o = 12 * lg(n)

Step 2: Bucketing elements of the input array

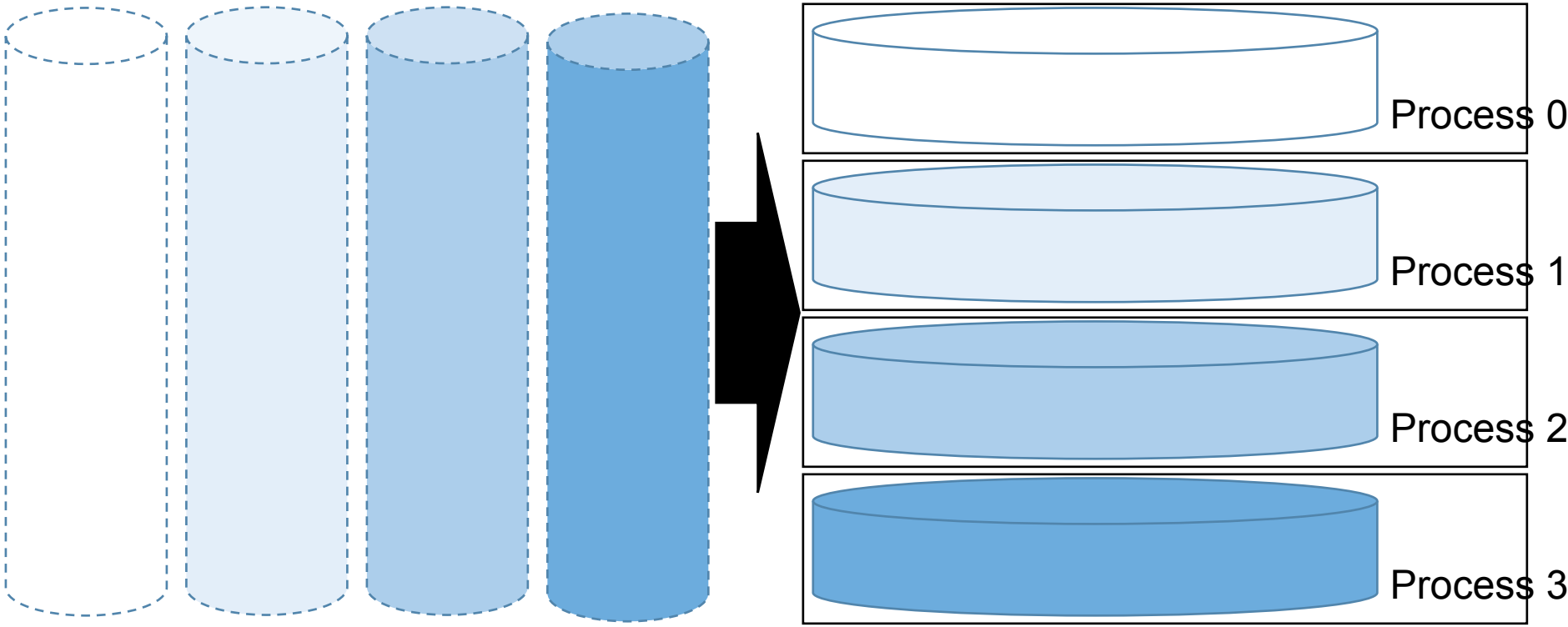


Buckets defined by pivots in step 1

Input arrays in each process's address space

Put all the elements into their corresponding bucket (as defined in step 1)
Note that all processes have to agree on their bucket definition

Step 3: Redistributing elements

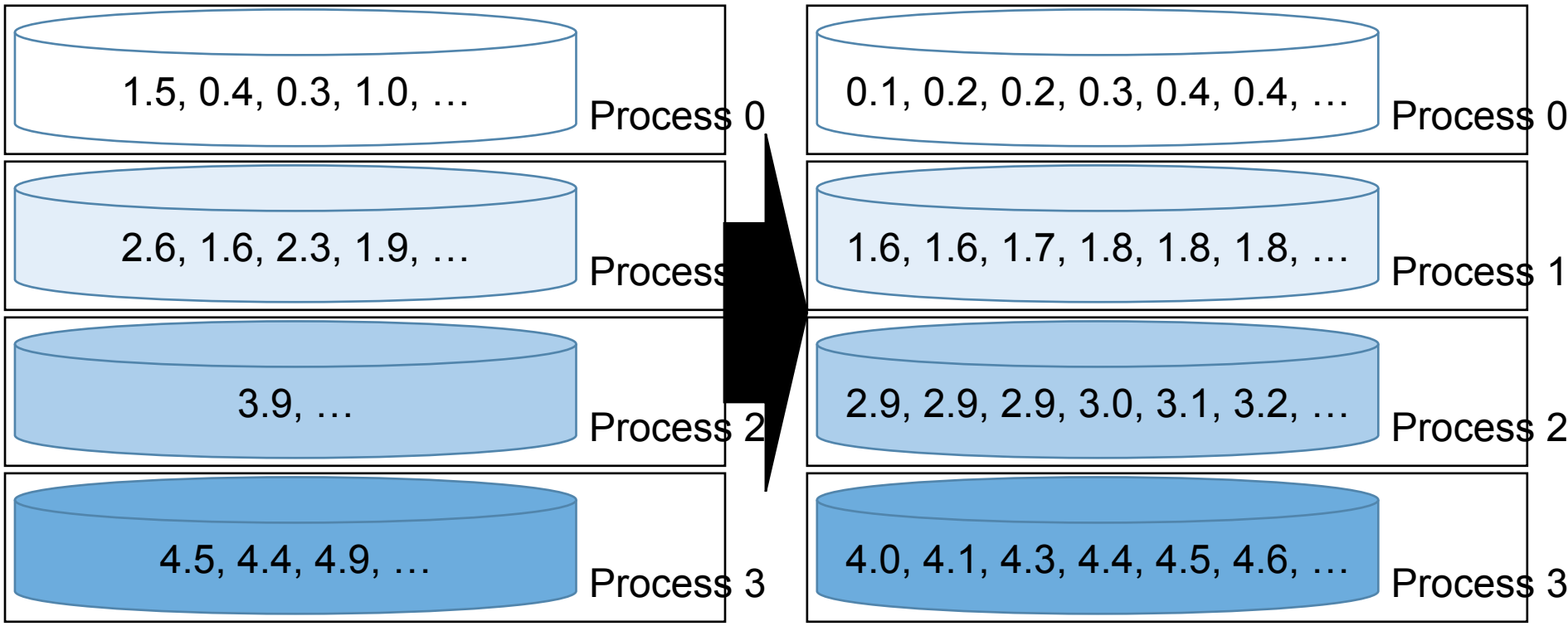


Virtual buckets from step 2

Buckets after redistribution

Redistribute the elements such that elements on each process are now separate, i.e., elements on process i < elements on process j

Step 4: Final local sort



Unsorted buckets from step 3

Sorted buckets after step 4

Sequentially sort each bucket using a fast sequential sort algorithm
The distributed array is now sorted!

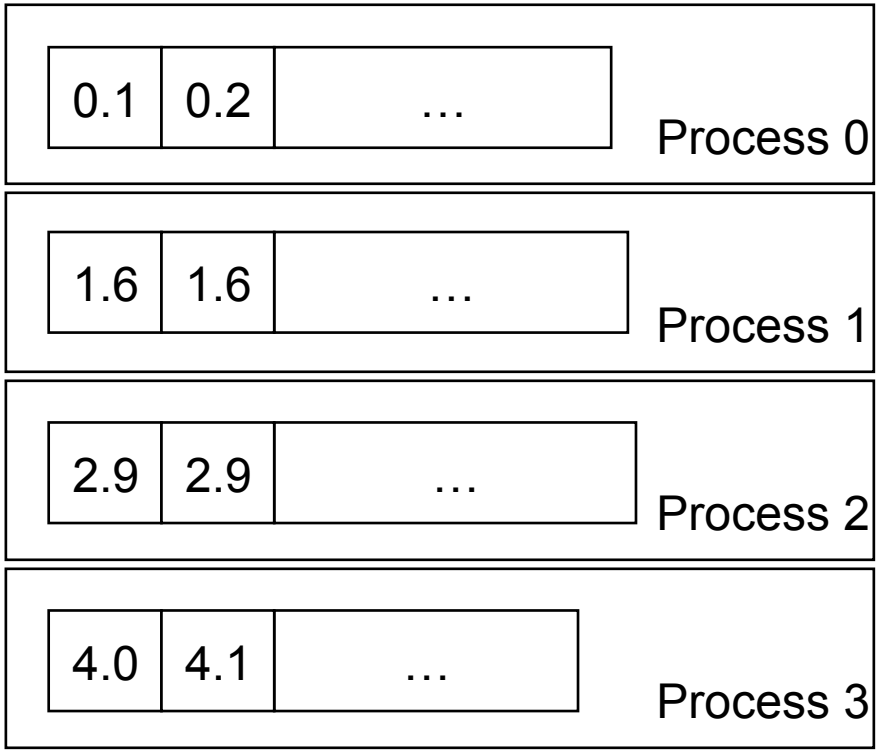
Step 4: Final local sort

Notes for the final step:

Buckets should not overlap so that all elements on process i should be less than elements on process j.

Bucket size on each process can be different, but,

Update localSize to the bucket size on each process!



Sorted buckets from step 4

Helper functions

```
void printArr(const char* arrName, int *arr, size_t size, int procId);
void printArr(const char* arrName, float *arr, size_t size, int procId);
```

e.g.,`printArr("pivot", pivot, procs-1, procId);`

Helps you debug your program, can be easily turned off by uncommenting `#define NO_DEBUG` in parallelSort.h

```
void randomSample(float *data, size_t dataSize,
                  float *sample, size_t sampleSize) {
  for (size_t i=0; i<sampleSize; i++) {
    sample[i] = data[rand()%dataSize];
  }
}
```

e.g. `randomSample( data, localSize, sample, 12*log(dataSize) );`

Uniform-randomly pick samples from data and put in sample array

Useful STL functions

std::sort(first, last)

e.g.
```
sort(data, data + localSize);
```

Comments: a very decent sequential sort

std::inplace_merge(first, middle, last)

e.g.
```
inplace_merge(data, data + 5, data + 10);
```

Comments: merge two sorted arrays between
(1) first to middle-1, and
(2) middle to last-1

std::lower_bound(first, middle, val)

e.g.
```
int bucketId = lower_bound(pivot, pivot+procs-1, data[i]) - pivot;
```

Comments: useful to find buckets for each elements

Examples can be found in src/stlSort.cpp

References: http://www.cplusplus.com/

Challenges

Choose a pivot that can divide the workload evenly.

Experiment your code with different inputs we provided: norm, exp, bad1

How to deal with different input patterns?

What are the inputs that can break your sampling scheme?

Thought experiment:

What if the input array is an integer array?

What are the new challenges induced by integer array?