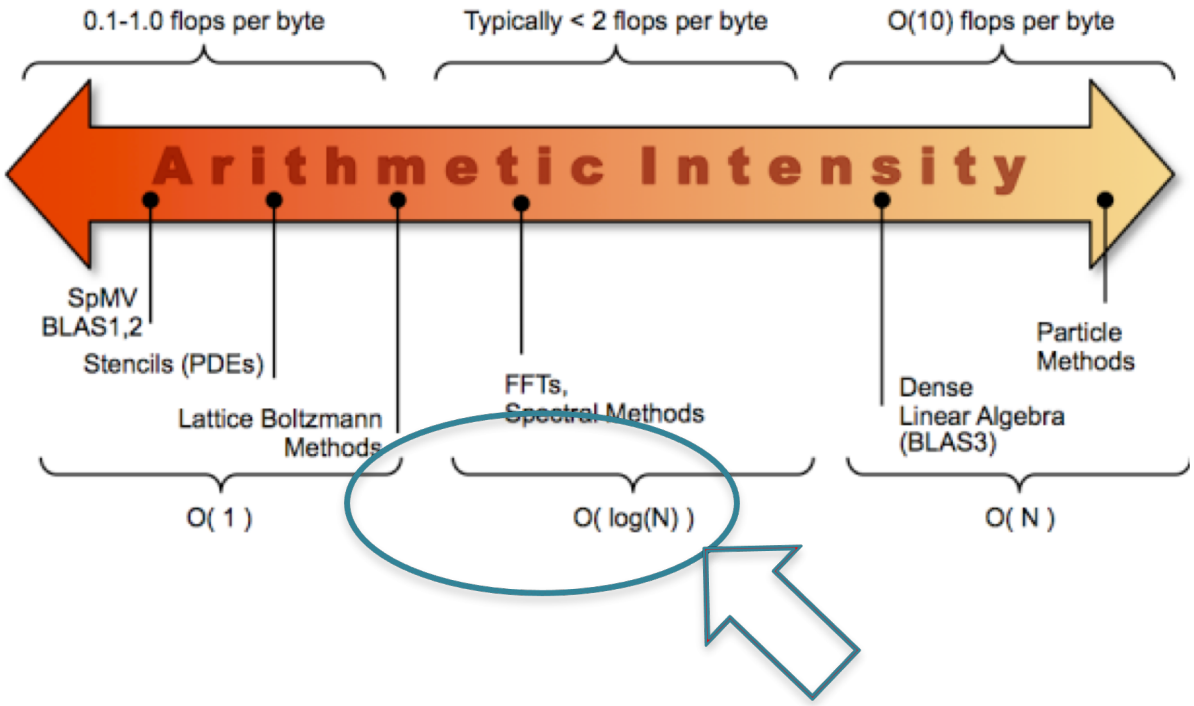
Parallel Sorting

Parallel computing - Parallel Algorithms

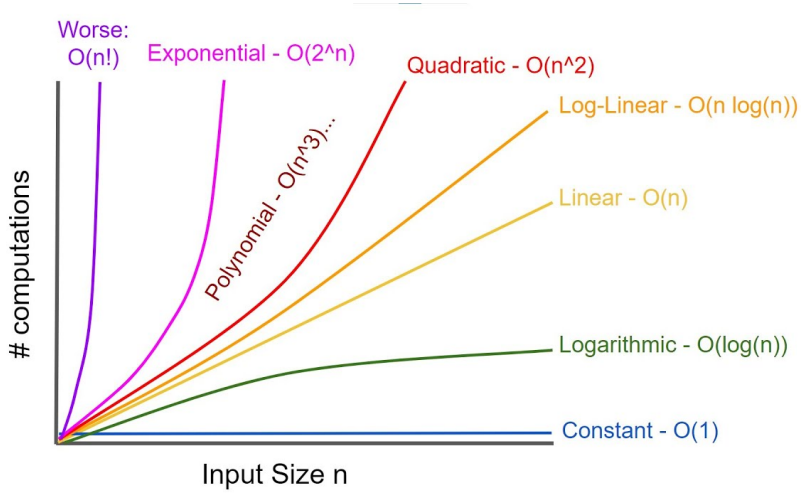
João Luís Sobral (jls@di …)

Nov/2022



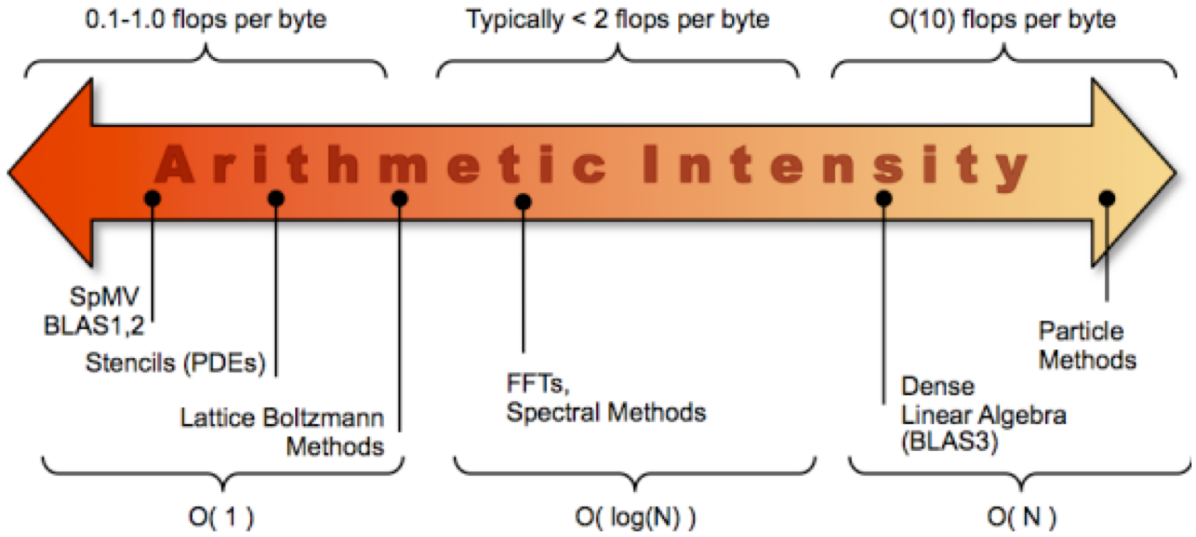
Parallel Algorithms

- **Traditional algorithm analysis:** Big O notation
  - Analysis of the number of operations per datum (n)
    - Matriz multiplication: ~~$O(n^3)$~~ $O(n^{\frac{3}{2}})$ (n=number of elements of C)
    - Sorting (n keys)
      - Bad sorting (burte force): $O(n^2)$
      - Better algorithm: $n \log_2(n)$
      - Best: ([k]n)
    - Insertion on a data structure (n insertions):
      - Sorted linked list $O(n^2)$
      - Binary tree: $O(n \log_2(n))$
      - Hash table O(n)



Parallel Algorithms

- **Arithmetic Intensity (operations per ~~byte loaded~~ data moves)**



- **Data storage**
  - Regular : vector, matrix
  - Irregular (pointer based): tree, graphs

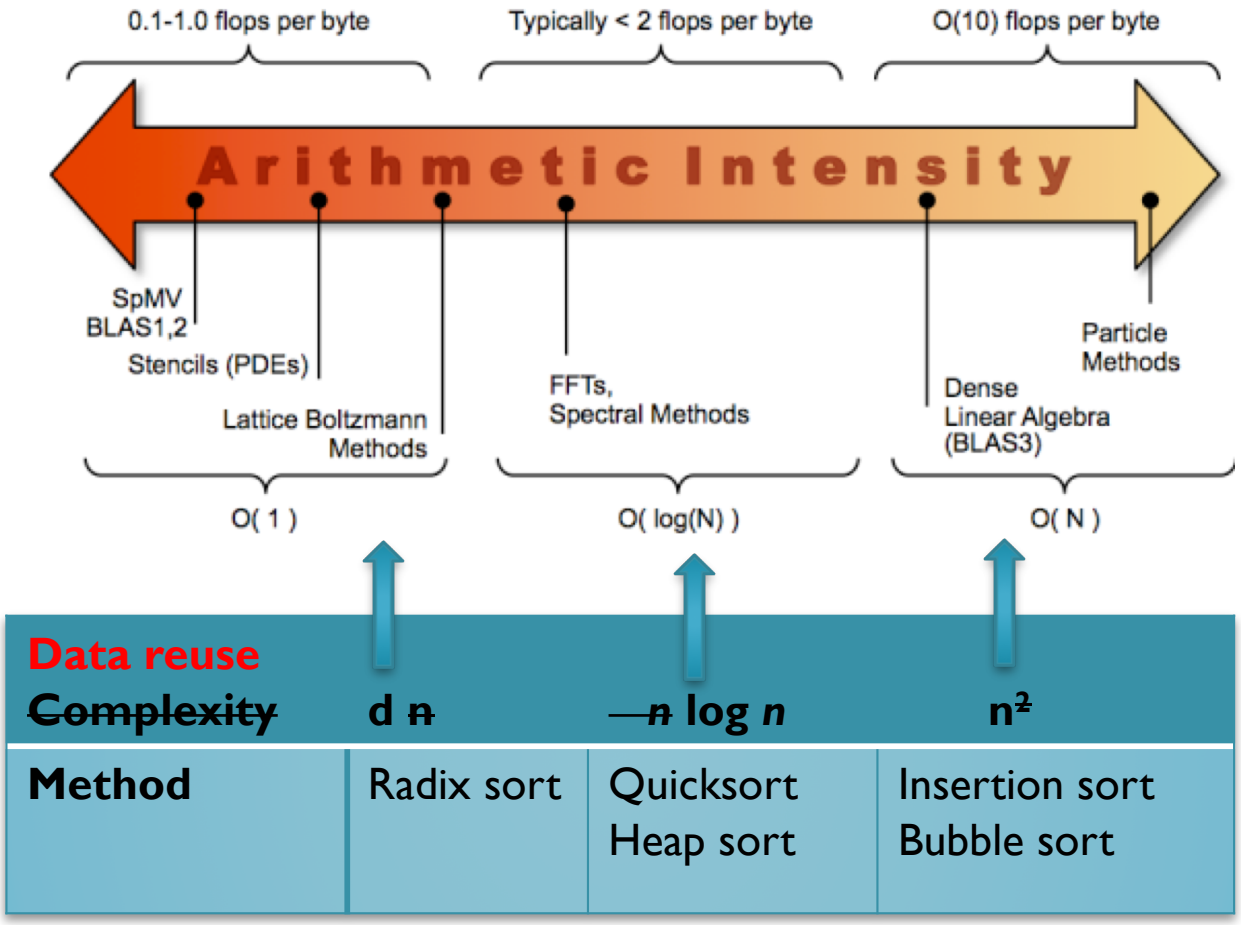
Data accesses
- Regular / irregular

Parallel Sorting

- [Sequential] sorting algorithms

Method	Complexity (average)	Description
Insertion sort	n^2	Insert elements into a sorted list
Bubble sort	n^2	Compare (and swap) successive elements
Quicksort	$n \log n$	Recursively sort elements less/greater than a given pivot
Merge sort	$n \log n$	Successively merge sorted sub-lists starting from lists with one element
Heap sort	$n \log n$	Insert elements into a binary heap
Radix sort	n d	Sort elements digit by digit (d) (two variants: MSD and LSD)

Parallel Sorting



Parallel Sorting

- Locality: Radix MSD vs Radix LSD
  - Both are based on key-indexed sorting
    - Example for 3-digit Strings (figure: 1st digit step on MSD)

1. Count frequencies of each letter using key as index
2. Compute frequency cumulates
3. Access cumulates using key as index to find record positions.
4. Copy back into original array

count frequencies →
compute cumulates →
move records →
copy back →

```
int N = a.length;
int[] count = new int[R];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int k = 1; k < 256; k++)
   count[k] += count[k-1];

for (int i = 0; i < N; i++)
   temp[count[a[i]++]] = a[i]

for (int i = 0; i < N; i++)
   a[i] = temp[i];
```

a[]

0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

count[]

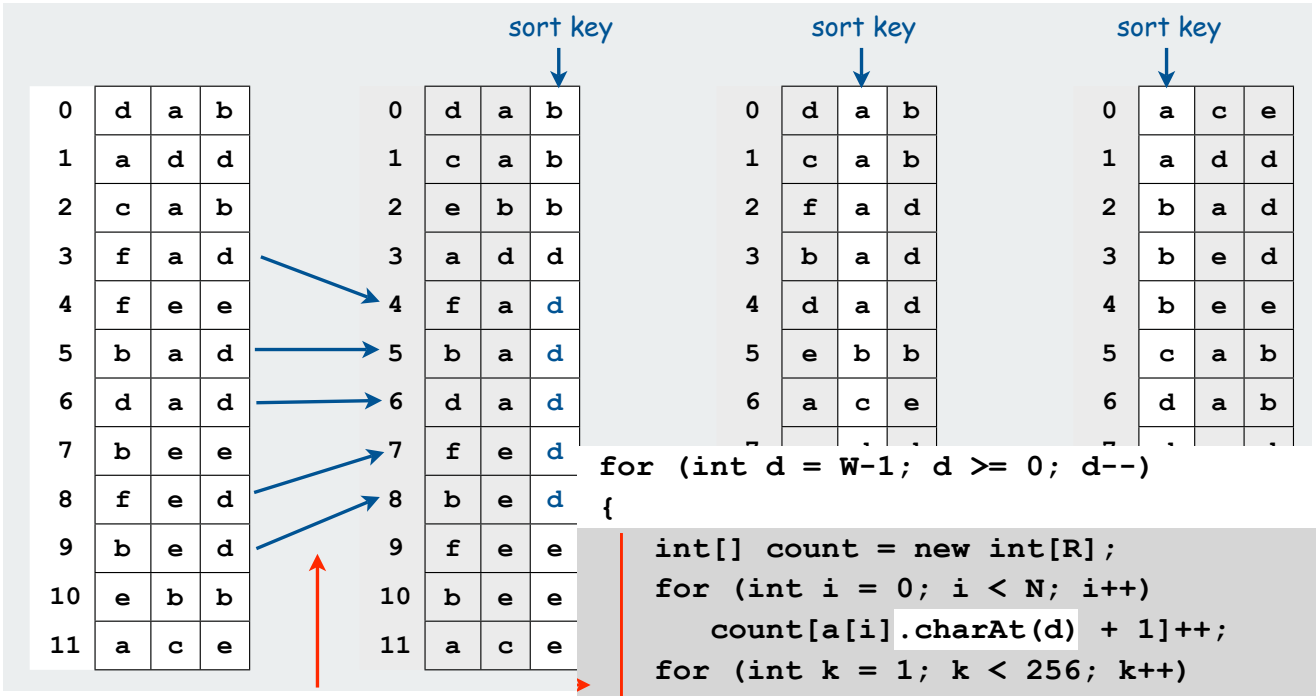
a	2
b	5
c	6
d	8
e	9
f	12

temp[]

0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Parallel Sorting

- Locality: Radix LSD (e.g. string with 3 digits)
  - D steps through the full data: DxN data moves




```
for (int d = W-1; d >= 0; d--)
{
   int[] count = new int[R];
   for (int i = 0; i < N; i++)
      count[a[i].charAt(d) + 1]++;
   for (int k = 1; k < 256; k++)
      count[k] += count[k-1];
   for (int i = 0; i < N; i++)
      temp[count[a[i].charAt(d)]++] = a[i];
   for (int i = 0; i < N; i++)
      a[i] = temp[i];
}
```

count frequencies

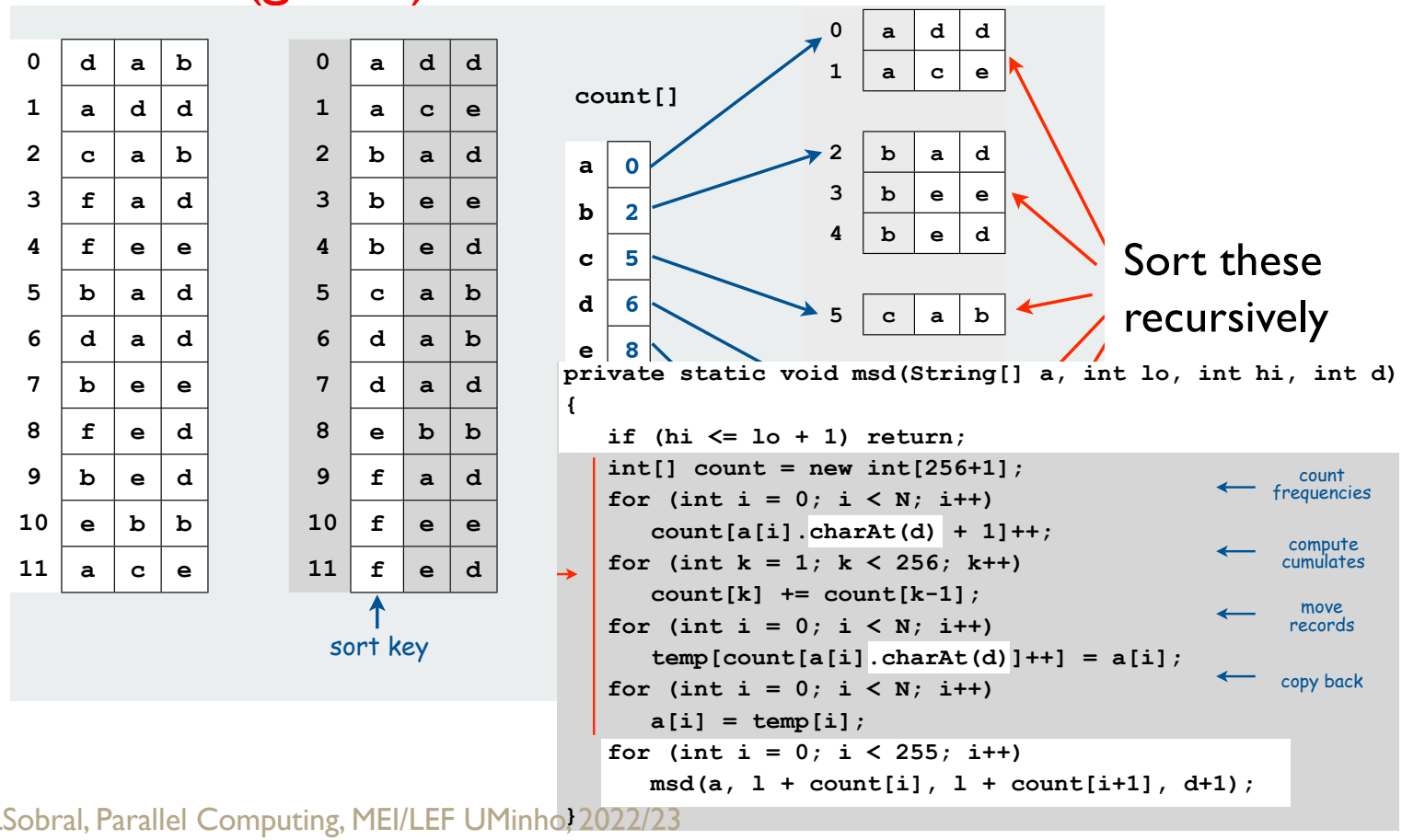
compute cumulates

move records

copy back

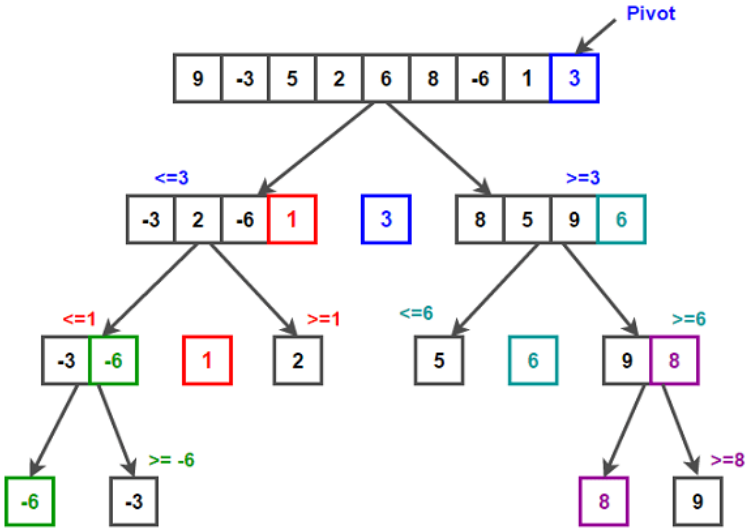
Parallel Sorting

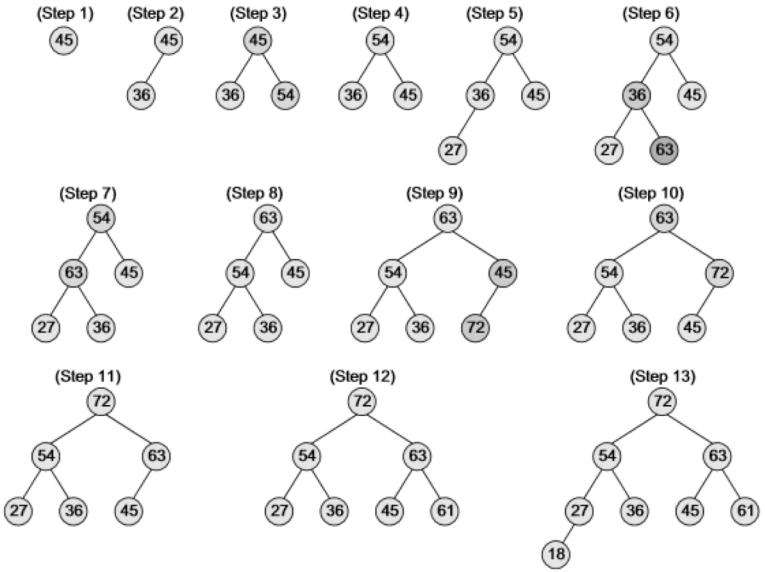
- Locality: Radix MSD(e.g. string with 3 digits)
  - Partition data in K-sets- 1 step through the full data:
    1xN (global) + local data moves



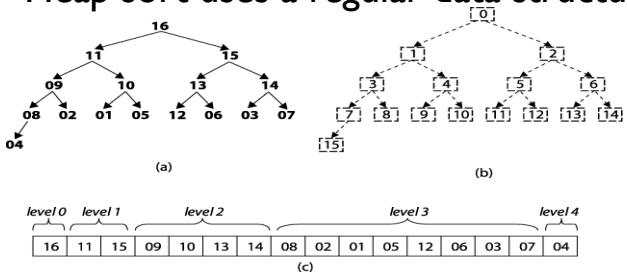
Parallel Sorting

- Locality: quick-sort vs heap-sort
  - Regular VS irregular data references





Heap sort uses a regular data structure



Both perform *n x log n* steps through the data but heapsort shows more irregular accesses to data indexes

Parallel Sorting

- Locality of reference in sorting algorithms

Method	Locality of reference	Improvements
Quicksort	Good spatial locality + bad temporal locality on initial stages	Initial set partioning using k keys
Mergesort	Good spatial locality + bad temporal locality on final merge stages	Single merge when data exceeds cache size
Heap sort	Bad	Cache aware trees + d-fan-out
Insertion sort	Bad	
Radix sort	Good when MSD first (only when processing LSDs)	Reduce the number passes through data

Parallel Sorting

- Parallelism in sorting algorithms

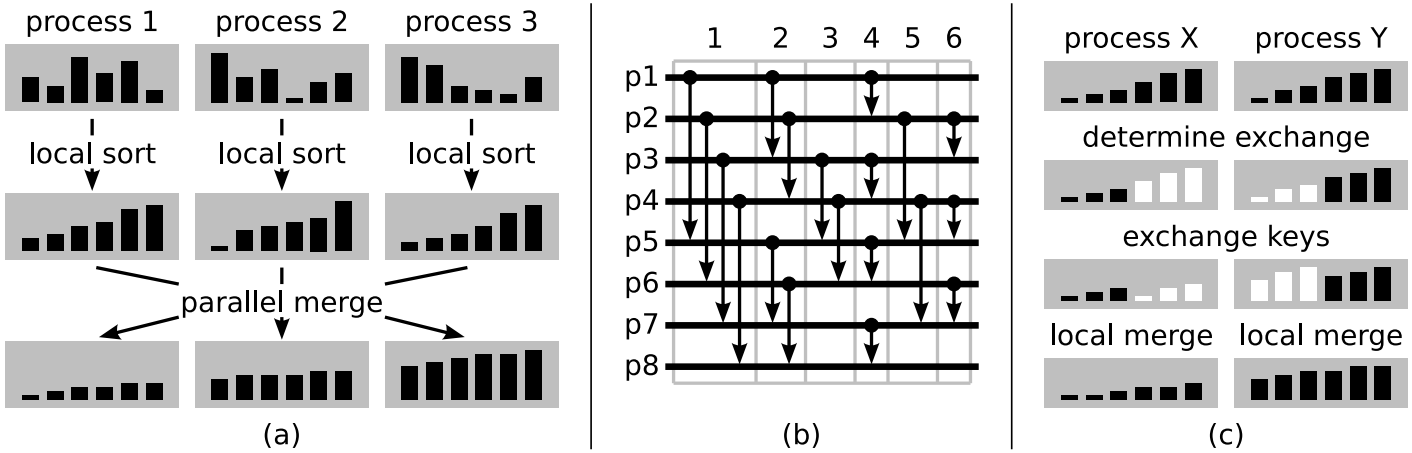
Method	
Quicksort	Sort sub-lists in parallel / start with p lists
Merge-sort	Merge p lists in parallel
Heap sort	???
Insertion sort	???
Radix sort	Sort set of digits in parallel

Parallel Sorting (on distributed memory)

- Design issues:
  - Keys are initially distributed over processors
    - Intermediate stage for other parallel algorithms
  - Data properties
    - Partially-sorted data? (not in the scope of this lecture)
  - Exploitable parallelism
    - Merger-based
    - Splitter-based
  - Efficiency considerations for parallelism
    - Data movements [across processors]
    - Load balancing
    - Avoid idle time (e.g. sequential phases)

Parallel Sorting

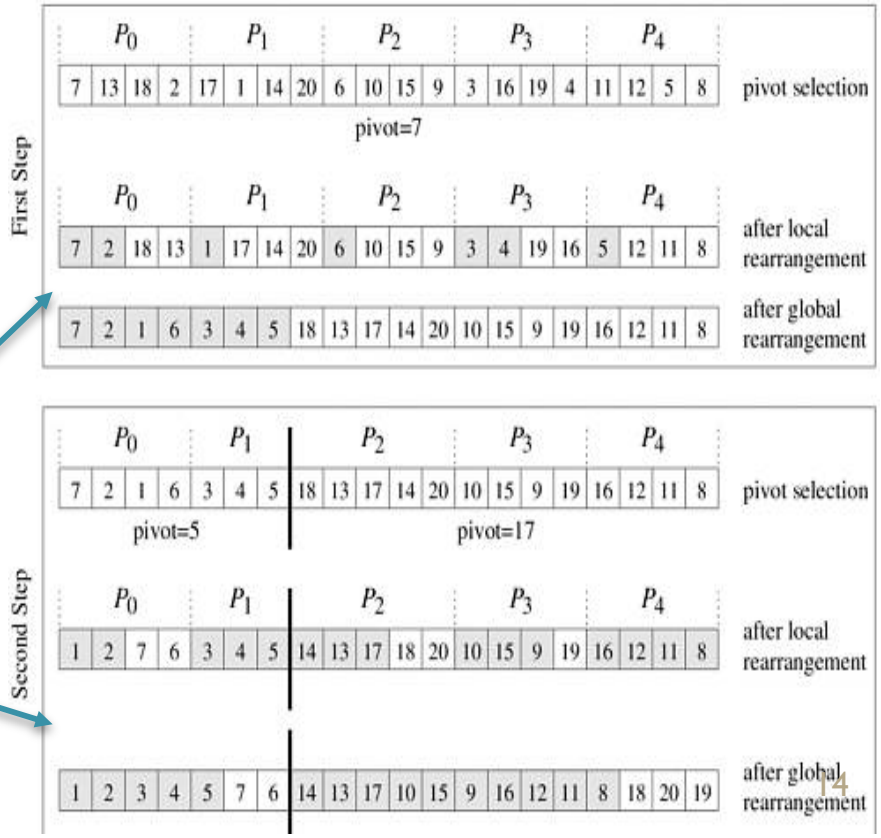
- Parallel Merge-Sort
  - Locally sort each set
  - Exchange sets among processors



  - Only effective when n/p ~1
  - Extensive data movements: when n/p>>1

Parallel Sorting

- Parallel quicksort (simplified)
  - Master selects and broadcasts *pivot* key
  - Each process locally splits using the *pivot*
    - Each processes holds *smaller* and *greater* partitions
  - Divide processors into *smaller* and *greater* sets
    - Send data to one processor on the other set
  - Repeat the processes until #sets = #p
  - Locally sort on each process p
  - Complexity:
    - Requires log p communication steps



Parallel Sorting

- Parallel radix sort
  - Each processor is responsible by a subset of digit values
  - Sort and count the number of digit values
  - All-reduce the total number of digits
  - Send keys to the processor responsible for each digit range
  - Repeat for the next digit
  - Complexity:
    - LSD – #D (number of digits) communication steps
    - MSD – One communication step

Parallel Sorting

- Parallelism in sorting algorithms
  - Sampling based
    - Split data into P sets using p-1 splitters
    - Each processor acts upon a local set
    - Minimizes data movements
  - Sampling alternatives
    - Regular sampling (p*(p-1) keys)
      - Not effective for large p
    - Random sampling
    - Histogram sampling

Parallel Sorting by Regular Sampling

1. Divide the set into *p* disjoint sets and locally order each set
   - Applies a local QuickSort
   - Selects *p-1* local samples that uniformly divide each set into p subsets
2. Order *p*(p-1)* samples and select best *p-1* pivot keys
3. Partition each set using the p-1 pivot keys
4. Merge *p*p* sets
   - Processor *i* merges the *i* partition