Sorting Algorithms

Jingke Li

Portland State University

Sorting Categorizations

- *Based on Data Location* —
  - *Internal Sort* — source data and final result are both stored in a processor's main memory.
  - *External Sort* — source data and final result are both stored in hard disks (too large to fit into main memory); data is brought into memory for processing a portion at a time.
- *Based on Data Information* —
  - *Comparison-based sorting* — without assuming any knowledge of data distribution. Most algorithms are for this case; a well-known lower bound is $\ln(n!)$.
  - *Distribution-based sorting* — with knowledge of data distribution, often can do better than $O(n \log n)$. *Example:* bucket sort.
  - *Adaptive sorting* — without assuming any knowledge of data distribution, but try to discover patterns from data and use the info in sorting. *Example:* sample sort.

Sorting Algorithms

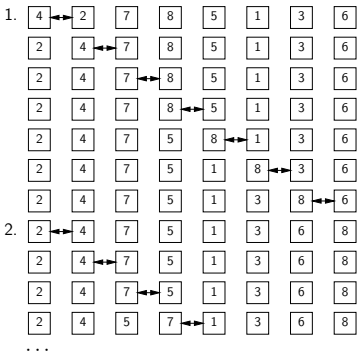
- Many sequential algorithms exist:
  - Bubble sort, Selection sort, Quicksort
  - Bucket sort, Radix sort
  - Mergesort, Timsort

  Most of these algorithms can be parallelized.
- Some can be extended to special parallel versions:
  - Odd-even sort, Shellsort, Shearsort (based bubble sort)
  - Hyper-quicksort (based on quicksort)
- Some are naturally parallel algorithms:
  - Sample sort
  - Bitonic mergesort

Bubble Sort

- Scan the array $n-1$ times, with the range reduced by one for each scan.
- In each scan, compare two adjacent array elements and swap them if necessary so that the smaller is on the left and the larger is on the right.

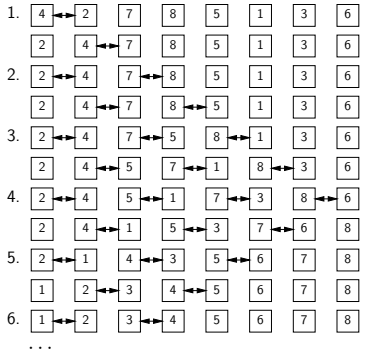
Complexity: $O(n^2)$



Parallel Bubble Sort

- The "bubbling" action of one number could start before the previous number has finished so long as it does not overtake the previous bubbling action.

Complexity: $2n$

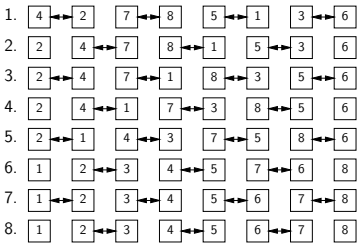


Odd-Even Sort

An improved parallel bubble sort. It operates in two alternating phases.

- *Even phase* — Even-numbered processes exchange numbers with their right neighbor.
- *Odd phase* — Odd-numbered processes exchange numbers with their right neighbor.

Complexity: n steps

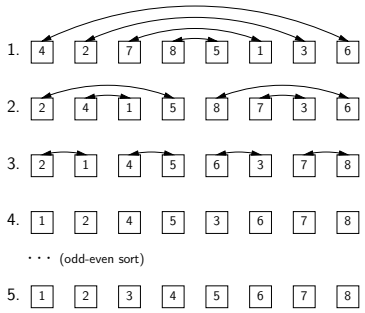


Now all processes have work to do from the beginning.

Shellsort

A further improvement over odd-even sort (provided long-distance communication is efficiently supported).

- *Phase 1* — $\log n$ steps are taken. In each step, processes are paired by "distances". The distance of pairing reduces by half in each step.
- *Phase 2* — Runs the odd-even sort; stops as soon as no exchange happens.



Complexity: $\leq n$ steps

Shearsort

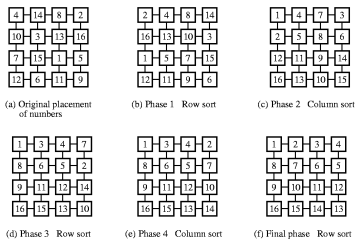
A 2D-mesh based variant of odd-even sort.

- *Odd phase* — Each row of numbers is sorted independently, in alternative directions:
  - *Even rows* — The smallest number of each column is placed at the rightmost end and largest number at the leftmost end.
  - *Odd rows* — The smallest number of each column is placed at the leftmost end and the largest number at the rightmost end.
- *Even phase* — Each column of numbers is sorted independently, placing the smallest number of each column at the top and the largest number at the bottom.

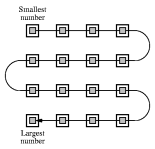
After $\log n + 1$ phases, the numbers are sorted with a snakelike placement in the mesh.

Complexity: $\sqrt{n}(\log n + 1)$ steps on a $\sqrt{n} \times \sqrt{n}$ mesh.

A Shearsort Example



Final result is in a snakelike placement:



- Sorting operations can be constrained in a single dimension if transposition is used.

Selection Sort

An improvement over bubble sort on the number of swaps.

- Scan the array $n - 1$ times, with range reduced by one for each scan.
- In each scan, the largest element in the range is found and swapped to the end of the range.

Complexity: $O(n^2)$

```
void selection_sort(int *array, int low, int high) {
  if (low >= high)
    return;
  for (int i = low; i < high; i++) {
    int jmax = low;
    for (int j = low+1; j <= high - (i-low); j++)
      if (array[j] > array[jmax])
        jmax = j;
    swap(&array[jmax], &array[high - (i-low)]);
  }
}
```


Parallel Selection Sort (?)

- Assume finding max can be done in $O(\log n)$ steps. Still the algorithm would need $O(n)$ iterations.
- This would give a total of $O(n \log n)$ time for a parallel version, which is not competitive.

Conclusion: A better sequential algorithm may not translate to a better parallel algorithm.

Quicksort

Arguably the best sequential sorting algorithm for random unknown data.

- Select a pivot value. Use the pivot to divide the array into two sections. In one section, all elements are smaller than the pivot, and in the other section, all elements are larger than the pivot.
- Recursively quicksort the two sections.
- The whole sorting process is done in-place, no additional buffers are needed.

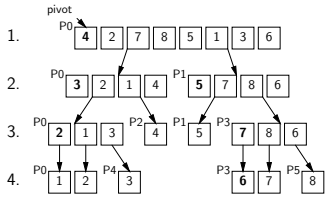
Complexity: $O(n \log n)$ on average, $O(n^2)$ in the worst case

Simple improvement for data that are not totally random — use sampling to select the pivot, *e.g.*

```
pivot = average(a[low], a[(low+high)/2], array[high]);
```

Parallel Quicksort

Assign recursive calls to processes.



Complexity: $O(\log n)$ with n processes (*average case*)

Drawback: At the beginning, only one process is active.

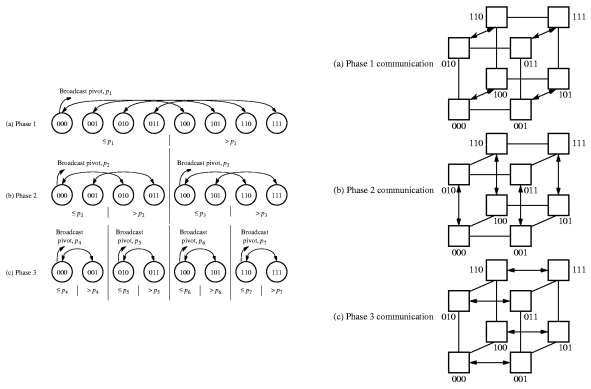
Hyper-Quicksort

A load-balanced version of parallel quicksort. Motivated from the hypercube topology, but works on other topologies as well.

1. Divide unsorted data evenly among processes.
2. Every process sorts its data with a fast sequential algorithm.
3. One process finds its median, and broadcasts to others.
4. Pair off processes; for each pair, data are exchanged — one process keeps small values, the other keeps large values.
5. Processes are now in two groups – "large" and "small"; apply the steps recursively to the two groups.

Advantage: Every process has work to do from beginning to end.

Hyper-Quicksort (cont.)



Bucket Sort

Conceptually, it is an extension to quicksort — Instead of partitioning an array into two sections, why not partition it into *more* sections?

However, it needs to assume that the range of array elements' values is known.

- Choose a parameter m, and partition the elements' value range evenly into m sections; create m empty buckets, one for each section.
- Distribute array elements into the m buckets.
- Sort each bucket with any sorting algorithm (typically, quicksort).
- Concatenate the sorted buckets to form the resulting array.

Complexity: $O(n + n\log(n/m))$ (on average)

– For sorting a random distribution of integers, and if $m \approx n$, then the sorting time is $O(n)$.

Example: Bucket Sort

Given a value range of 0-999. Create 10 buckets, [0-99], [100-199], etc.

- Input data:

```
231 043  531 101 750 218 955 440 387 210 187 532 376 604 021 797
424 181  123 862 519 894 911 015 327 431 849 246 756 048 027 152
```

- Distribute numbers to buckets:

[0xx]	[1xx]	[2xx]	[3xx]	[4xx]	[5xx]	[6xx]	[7xx]	[8xx]	[9xx]
043	101	231	387	440	531	604	750	862	955
021	187	218	376	424	532		797	894	911
015	181	210	327	431	519		756	849	
048	123	246							
027	152								

- Sort the buckets:

015	101	210	327	424	519	604	750	849	911
021	123	218	376	431	531		756	862	955
027	152	231	387	440	532		797	894	
043	181	246							
048	187								

- Result:

```
015 021 027 043 048 101 123 152 181 187 210 218 231 246 327 376
387 424 431 440 519 531 532 604 750 756 797 849 862 894 911 955
```


Parallel Bucket Sort

Assume the original numbers are distributed across a known interval M.

1. The interval of the input distribution is divided into p equal-size regions, where p is the number of processes.
2. Divide the unsorted data evenly among processes.
3. Every process sorts its data into p buckets, one for each region.
4. Every process sends bucket i to process P_i; and receives $p-1$ buckets from other processes.
5. Every process merges p buckets into a single sorted list.

Advantages: Every process has work to do from start to end.

Problems: Only works when the distribution of the data is known.

Sample Sort

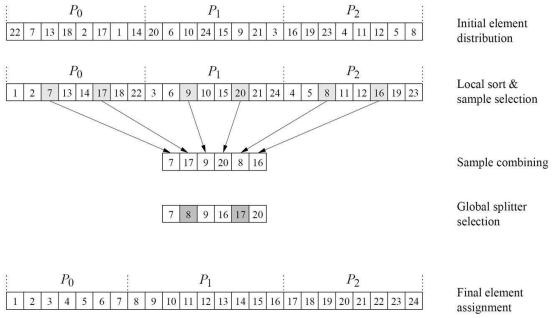
An attempt to combine bucket sort and hyper-quicksort.

1. Divide the unsorted data evenly among processes.
2. Every process sorts its data with a fast sequential algorithm.
3. Every process collects $p-1$ samples at equal-interval points
4. All the samples are collected to a single process and are sorted there.
5. $p-1$ values are selected from equal-interval points in the sorted list, and are broadcast to all processes.
6. Every process divides its data to p partitions using the $p-1$ "splitter" values it receives; and send partition j to process j.
7. Every process merges p copies of partition.

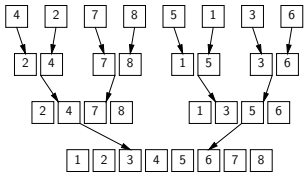
Observations:

- Every process has work to do from beginning to end.
- Suitable for MIMD multicomputers.

A Sample Sort Example



Mergesort



- Pairs of numbers are combined (merged) into sorted list of two numbers.
- Pairs of these lists of four numbers are merged into sorted lists of eight numbers.
- This is continued until a fully sorted list is obtained.

Parallel Mergesort:

Works like an inverse of parallel quicksort.

Complexity: $O(\log n)$ with n processes (*average case*)

Timsort

A practical adaptive mergesort developed by Tim Peters in 2002. (It's Python's standard sorting algorithm since version 2.3.)

Key Idea: Taking advantage of partially sorted segments in the input.

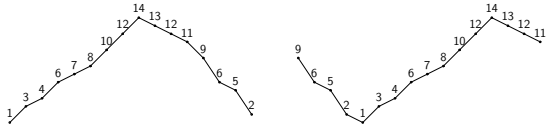
- Finding "runs" — A run is a sequence of adjacent numbers that is either "ascending" (*e.g.* `1,2,2,3,6`) or "strictly descending", (*e.g.* `8,5,3,2`).

1. Scan array to find or create runs of a certain minimum size:
   - If a run of size $\geq$ `MINRUN` exists at current position, normalize it to ascending form (perform in-place reversal if needed);
   - otherwise, create a run by sorting `MINRUN` elements from current position.
2. Use a stack to keep track of runs' positions and sizes:
   - The interval info of runs are pushed onto the stack, one at a time.
   - The size of stack is carefully maintained to be $O(\log n)$.
3. Merge the runs from the stack in order.

Bitonic Mergesort

A fast parallel sorting algorithm — $O(\log^2 n)$ with n processes.

Bitonic Sequence: "Two sorted lists concatenated together, with or without a circular shift."

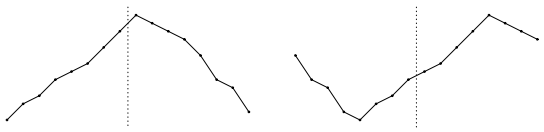


Formal Definition:

A bitonic sequence is a sequence $a_0, \cdots, a_{n-1}$ such that $a_0 \leq \cdots \leq a_k \geq \cdots \geq a_{n-1}$ for some $k, 0 \leq k < n$, or a circular shift of such a sequence.

A Key Property of Bitonic Sequence

Partition an (even-numbered) bitonic sequence into two equal halves:

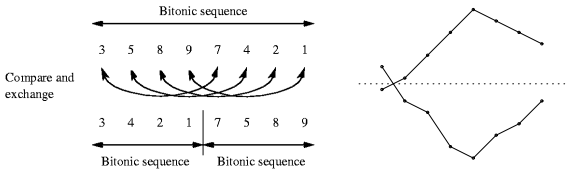


Overlay the two halves on top of each other:

Key Observation: The two subsequences cross each other at most once.

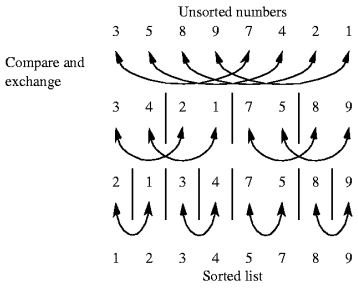
Sort a Bitonic Sequence

Given an n number bitonic sequence, if we perform a compare-and-exchange operation on a_i with $a_{n/2+i}$ for all i $(0 \le i < n/2)$, such that the smaller number goes to the left and the larger number goes to the right, we get two bitonic sequences, one consists of "small" values, the other "large" values. For example:



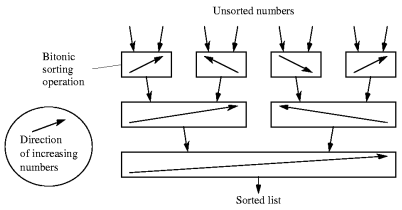
Sort a Bitonic Sequence (cont.)

Recursively performing compare-and-exchange operations to subsequences will eventually create bitonic sequences consisting of one number each, and overall a fully sorted list.

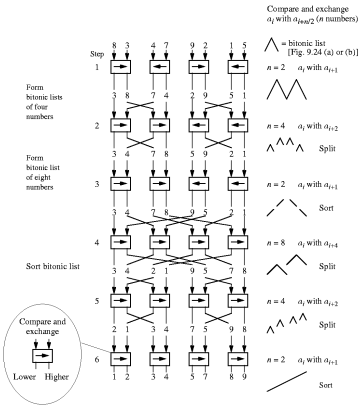


Bitonic Sort Algorithm

1. Form $n/2$ trivial bitonic sequences (each consists of two numbers).
2. Sort these bitonic sequences; the result are $n/4$ bitonic sequences.
3. Repeat until a single sorted sequence is obtained.



Bitonic Sort Example



Complexity Analysis

Let $M(n)$ be the time to merge two $n/2$-number bitonic sequences into a single sorted sequence, and $T(n)$ be the time to sort n numbers following the bitonic sort algorithm.

$$
\begin{aligned}
M(n) &= \log n \\
T(n) &= T(n/2) + M(n) \\
&= T(n/4) + M(n/2) + M(n) \\
&= 1 + \cdots + M(n/2) + M(n) \\
&= 1 + \cdots + (\log n - 1) + \log n \\
&= \log n(\log n + 1)/2 \\
&= O(\log^2 n)
\end{aligned}
$$

Theoretically, it is one of the best parallel sorting algorithms.