Parallel Sorting Algorithms

Ricardo Rocha and Fernando Silva

Computer Science Department
Faculty of Sciences
University of Porto

Parallel Computing 2015/2016

(Slides based on the book 'Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. B. Wilkinson, M. Allen, Prentice Hall')

Sorting in Parallel

Why?

- Sorting, or rearranging a list of numbers into increasing (decreasing) order, is a **fundamental operation** that appears in many applications

Potential speedup?

- Best sequential sorting algorithms (mergesort and quicksort) have (respectively worst-case and average) time complexity of $\mathcal{O}(n \log(n))$
- The best we can aim with a parallel sorting algorithm using n processing units is thus a time complexity of $\mathcal{O}(n \log(n))/n = \mathcal{O}(\log(n))$
- But, in general, a realistic $\mathcal{O}(\log(n))$ algorithm with n processing units is a goal that is not easy to achieve with comparasion-based sorting algorithms

Compare-and-Exchange

An operation that forms the basis of several classical sequential sorting algorithms is the **compare-and-exchange (or compare-and-swap)** operation.

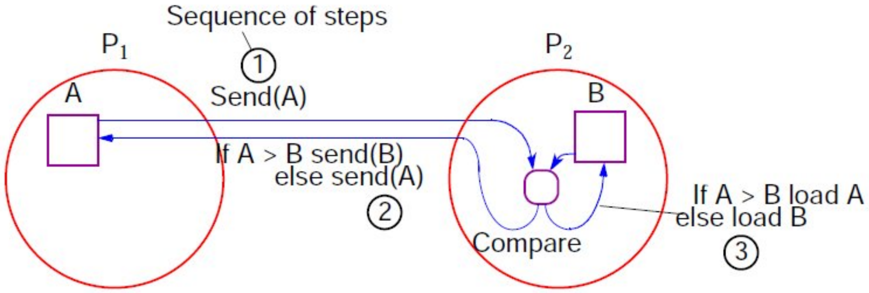
In a compare-and-exchange operation, two numbers, say A and B, are compared and if they are not ordered, they are exchanged. Otherwise, they remain unchanged.

```
if (A > B) { // sorting in increasing order
  temp = A;
  A = B;
  B = temp;
}
```

Question: how can compare-and-exchange be done in parallel?

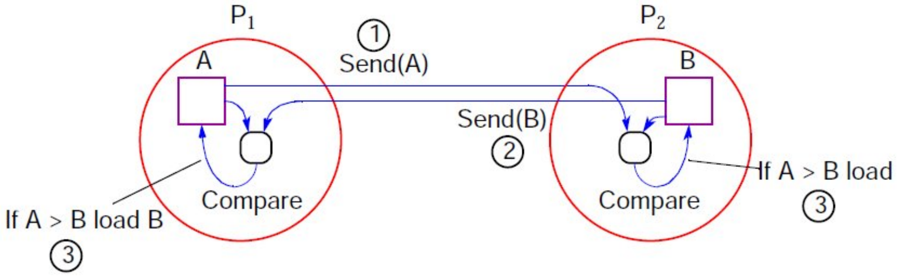
Parallel Compare-and-Exchange

Version 1 – P_1 sends A to P_2, which then compares A and B and sends back to P_1 the $min(A, B)$.



Parallel Compare-and-Exchange

Version 2 – P_1 sends A to P_2 and P_2 sends B to P_1, then both perform comparisons and P_1 keeps the $min(A, B)$ and P_2 keeps the $max(A, B)$.



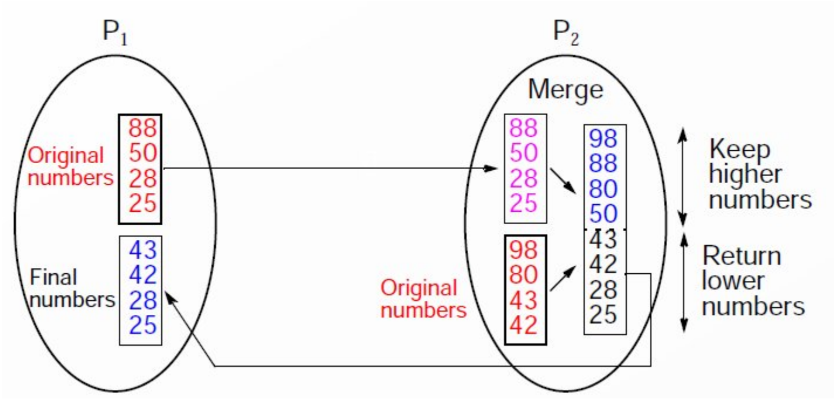
Data Partitioning

So far, we have assumed that there is one processing unit for each number, but normally there would be many more numbers (n) than processing units (p) and, in such cases, a list of n/p numbers would be assigned to each processing unit.

When dealing with lists of numbers, the **operation of merging two sorted lists** is a common operation in sorting algorithms.

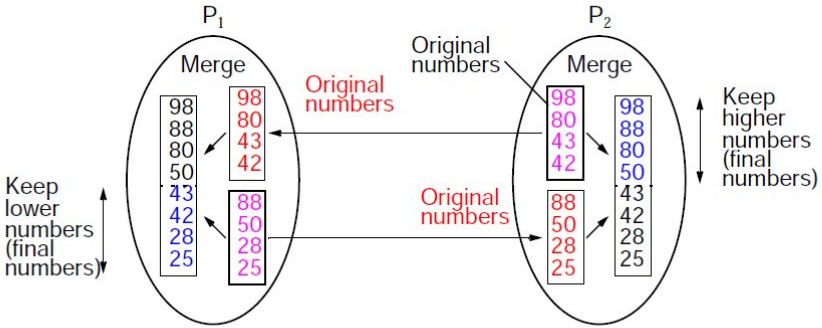
Parallel Merging

Version 1 – P_1 sends its list to P_2, which then performs the merge operation and sends back to P_1 the lower half of the merged list.



Parallel Merging

Version 2 – both processing units exchange their lists, then both perform the merge operation and P_1 keeps the lower half of the merged list and P_2 keeps the higher half of the merged list.



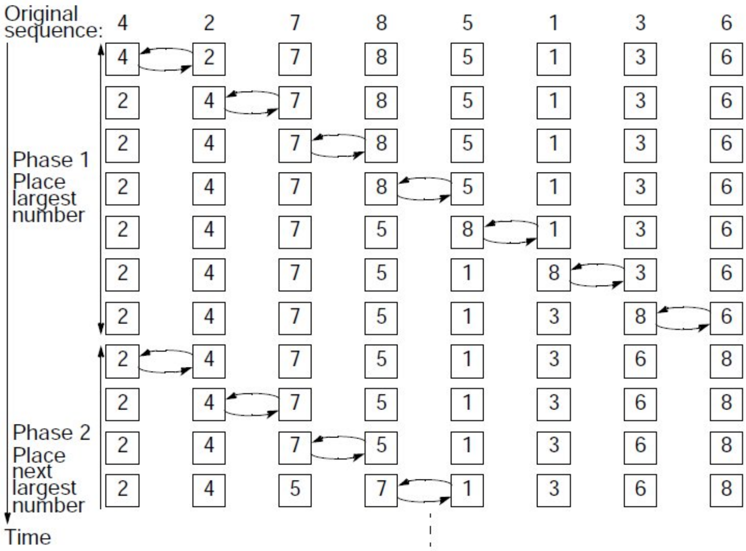
Bubble Sort

In bubble sort, the largest number is first moved to the very end of the list by a series of compare-and-exchange operations, starting at the opposite end. The procedure repeats, stopping just before the previously positioned largest number, to get the next-largest number. In this way, **the larger numbers move (like a bubble) toward the end of the list**.

```
for (i = N - 1; i > 0; i--)
  for (j = 0; j < i; j++) {
    k = j + 1;
    if (a[j] > a[k]) {
      temp = a[j];
      a[j] = a[k];
      a[k] = temp;
    }
  }
```

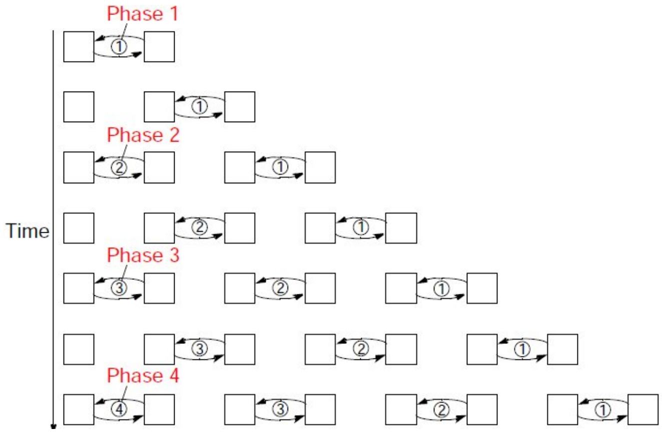
The total number of compare-and-exchange operations is $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, which corresponds to a time complexity of $\mathcal{O}(n^2)$.

Bubble Sort


Original sequence: 4 2 7 8 5 1 3 6
4 2 7 8 5 1 3 6
2 4 7 8 5 1 3 6
2 4 7 8 5 1 3 6
2 4 7 8 5 1 3 6
2 4 7 5 8 1 3 6
2 4 7 5 1 8 3 6
2 4 7 5 1 3 8 6
2 4 7 5 1 3 6 8
2 4 7 5 1 3 6 8
2 4 7 5 1 3 6 8
2 4 5 7 1 3 6 8
Phase 1
Place largest number
Phase 2
Place next largest number
Time

Parallel Bubble Sort

A possible idea is to **run multiple iterations in a pipeline fashion**, i.e., start the bubbling action of the next iteration before the preceding iteration has finished in such a way that it does not overtakes it.



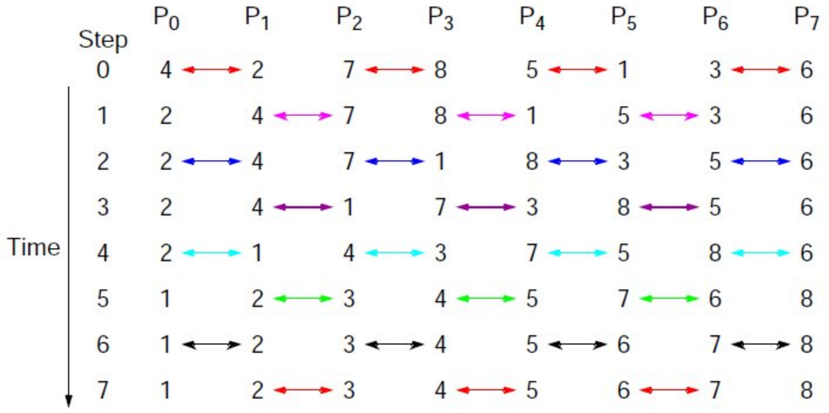
Odd-Even Transposition Sort

Odd-even transposition sort is a variant of bubble sort which operates in two alternating phases:

- **Even Phase:** even processes exchange values with right neighbors ($P_0 \leftrightarrow P_1$, $P_2 \leftrightarrow P_3$, ...)
- **Odd Phase:** odd processes exchange values with right neighbors ($P_1 \leftrightarrow P_2$, $P_3 \leftrightarrow P_4$, ...)

For sequential programming, odd-even transposition sort has no particular advantage over normal bubble sort. However, its parallel implementation corresponds to a time complexity of $\mathcal{O}(n)$.

Odd-Even Transposition Sort

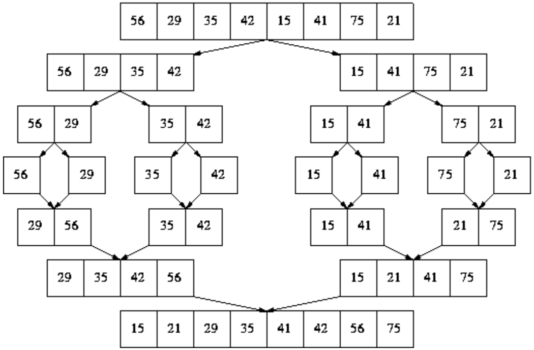


Odd-Even Transposition Sort

```
rank = process_id();
A = initial_value();
for (i = 0; i < N; i++) {
  if (i % 2 == 0) {                          // even phase
    if (rank % 2 == 0) {                     // even process
      recv(B, rank + 1); send(A, rank + 1);
      A = min(A,B);
    } else {                                 // odd process
      send(A, rank - 1); recv(B, rank - 1);
      A = max(A,B);
    }
  } else if (rank > 0 && rank < N - 1) { // odd phase
    if (rank % 2 == 0) {                     // even process
      recv(B, rank - 1); send(A, rank - 1);
      A = max(A,B);
    } else {                                 // odd process
      send(A, rank + 1); recv(B, rank + 1);
      A = min(A,B);
    }
}
```

Mergesort

Mergesort is a classical sorting algorithm using a **divide-and-conquer** approach. The initial unsorted list is first divided in half, each half sublist is then applied the same division method until individual elements are obtained. Pairs of adjacent elements/sublists are then **merged into sorted sublists** until the one fully merged and sorted list is obtained.



Mergesort

Computations only occur when merging the sublists. In the worst case, it takes $2s - 1$ steps to merge two sorted sublists of size s. If we have $m = \frac{n}{s}$ sorted sublists in a merging step, it takes

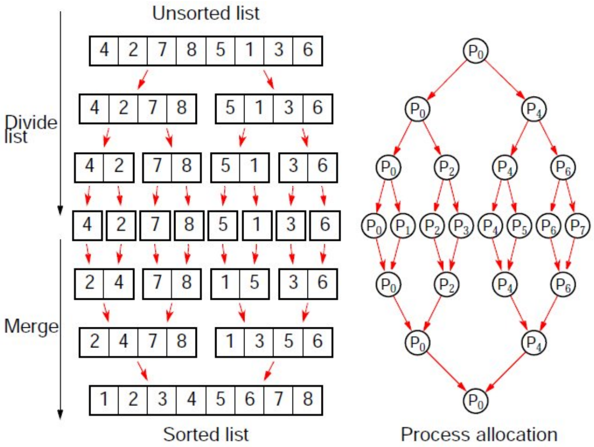
$$\frac{m}{2}(2s - 1) = ms - \frac{m}{2} = n - \frac{m}{2}$$

steps to merge all sublists (two by two).

Since in total there are $\log(n)$ merging steps, this corresponds to a time complexity of $\mathcal{O}(n \log(n))$.

Parallel Mergesort

The idea is to **take advantage of the tree structure of the algorithm** to assign work to processes.



Parallel Mergesort

If we ignore communication time, computations still only occur when merging the sublists. But now, in the worst case, it takes $2s - 1$ steps to merge all sublists (two by two) of size s in a merging step.

Since in total there are $\log(n)$ merging steps, it takes

$$\sum_{i=1}^{\log(n)} (2^i - 1)$$

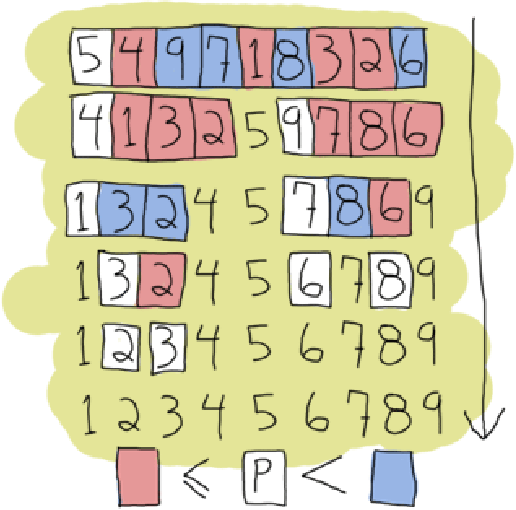
steps to obtain the final sorted list in a parallel implementation, which corresponds to a time complexity of $\mathcal{O}(n)$.

Quicksort

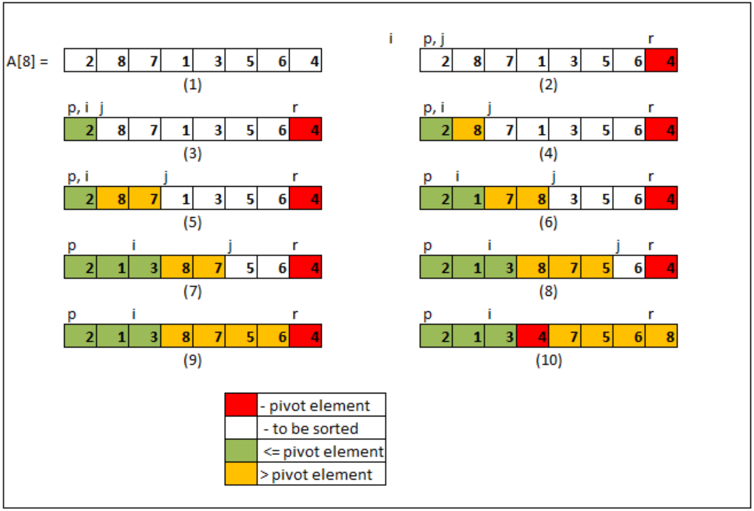
Quicksort is a popular sorting algorithm also using a **divide-and-conquer** approach. The initial unsorted list is first divided into two sublists in such a way that **all elements in the first sublist are smaller than all the elements in the second sublist**. This is achieved by selecting one element, called a **pivot**, against which every other element is compared (the pivot could be any element in the list, but often the first or last elements are chosen). Each sublist is then applied the same division method until individual elements are obtained. With proper ordering of the sublists, the final sorted list is then obtained.

On average, the quicksort algorithm shows a time complexity of $\mathcal{O}(n\log(n))$ and, in the worst case, it shows a time complexity of $\mathcal{O}(n^2)$, though this behavior is rare.

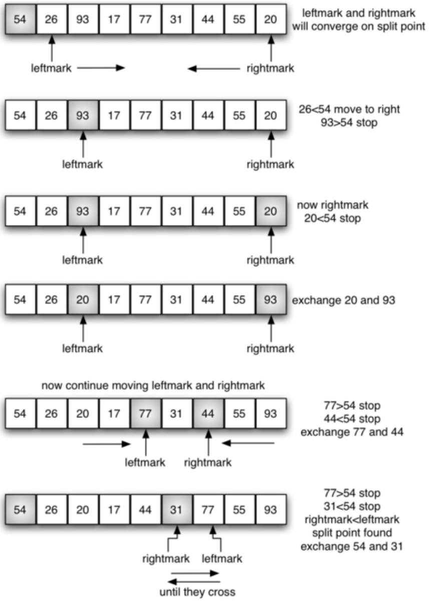
Quicksort


5 4 9 7 1 8 3 2 6
4 1 3 2 5 9 7 8 6
1 3 2 4 5 7 8 6 9
1 3 2 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
≤ P <

Quicksort: Lomuto Partition Scheme

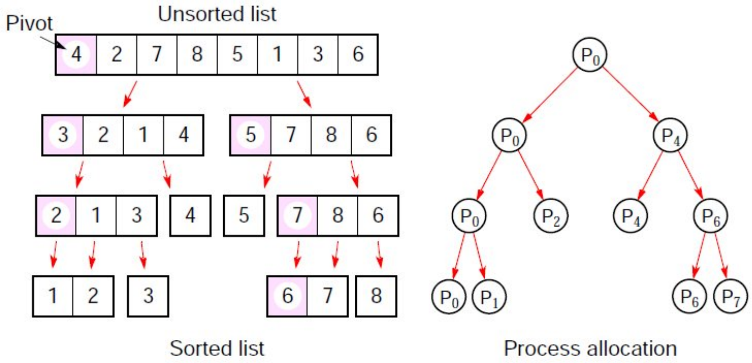


Quicksort: Hoare Partition Scheme



Parallel Quicksort

As for mergesort, the idea is to **take advantage of the tree structure of the algorithm** to assign work to processes.



Parallel Quicksort

Allocating processes in a tree structure leads to two fundamental problems:

- In general, the **partition tree is not perfectly balanced** (selecting good pivot candidates is crucial for efficiency)
- The process of assigning work to processes seriously **limits the efficient usage of the available processes** (the initial partition only involves one process, then the second partition involves two processes, then four processes, and so on)

Again, if we ignore communication time and consider that pivot selection is ideal, i.e., creating sublists of equal size, then it takes

$$\sum_{i=0}^{\log(n)} \frac{n}{2^i} \approx 2n$$

steps to obtain the final sorted list in a parallel implementation, which corresponds to a time complexity of $\mathcal{O}(n)$. The worst-case pivot selection degenerates to a time complexity of $\mathcal{O}(n^2)$.

Odd-Even Merge

Odd-even merge is a merging algorithm for sorted lists which can **merge two sorted lists into one sorted list**. It works as follows:

- Let $A = [a_0, \ldots, a_{n-1}]$ and $B = [b_0, \ldots, b_{n-1}]$ be two sorted lists
- Consider $E(A) = [a_0, a_2, \ldots, a_{n-2}]$, $E(B) = [b_0, b_2, \ldots, b_{n-2}]$ and $O(A) = [a_1, a_3, \ldots, a_{n-1}]$, $O(B) = [b_1, b_3, \ldots, b_{n-1}]$ as the sublists with the elements of A and B in even and odd positions, respectively
- Recursively merge $E(A)$ with $E(B)$ to obtain C
- Recursively merge $O(A)$ with $O(B)$ to obtain D
- Interleave C with D to get an almost-sorted list E
- Rearrange unordered neighbors (using compare-and-exchange operations) to completely sort E

Odd-Even Merge

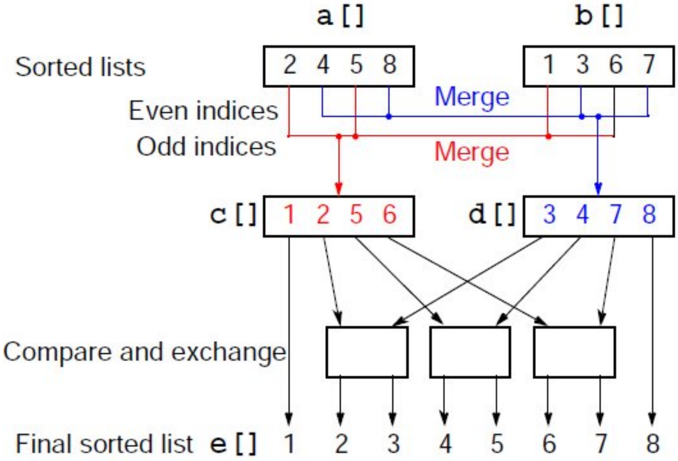
Let $A = [2, 4, 5, 8]$ and $B = [1, 3, 6, 7]$ then:

- $E(A) = [2, 5]$ and $O(A) = [4, 8]$
- $E(B) = [1, 6]$ and $O(B) = [3, 7]$

Odd-even merge algorithm:

- Recursively merge $E(A)$ with $E(B)$ to obtain C
  $C = odd\_even\_merge([2, 5], [1, 6]) = [1, 2, 5, 6]$
- Recursively merge $O(A)$ with $O(B)$ to obtain D
  $D = odd\_even\_merge([4, 8], [3, 7]) = [3, 4, 7, 8]$
- Interleave C with D to obtain E
  $E = [1, \quad 2, 3, \quad 5, 4, \quad 6, 7, \quad 8]$
- Rearrange unordered neighbors (c, d) as $(min(c, d), max(c, d))$
  $E = [1, 2, 3, 4, 5, 6, 7, 8]$

Odd-Even Merge



Odd-Even Merge

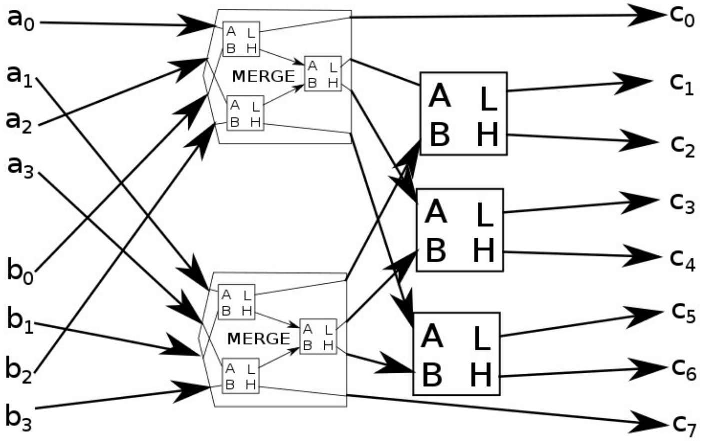
In summary, merging two sorted lists of size n requires:

- Two recursive calls to $odd\_even\_merge()$ with each call merging two sorted sublists of size $\frac{n}{2}$
- $n - 1$ compare-and-exchange operations

For example, the call to $odd\_even\_merge([2, 4, 5, 8], [1, 3, 6, 7])$ leads to:

- One call to $odd\_even\_merge([2, 5], [1, 6])$
- Another call to $odd\_even\_merge([4, 8], [3, 7])$
- 3 compare-and-exchange operations

4x4 Odd-Even Merge



Parallel Odd-Even Merge

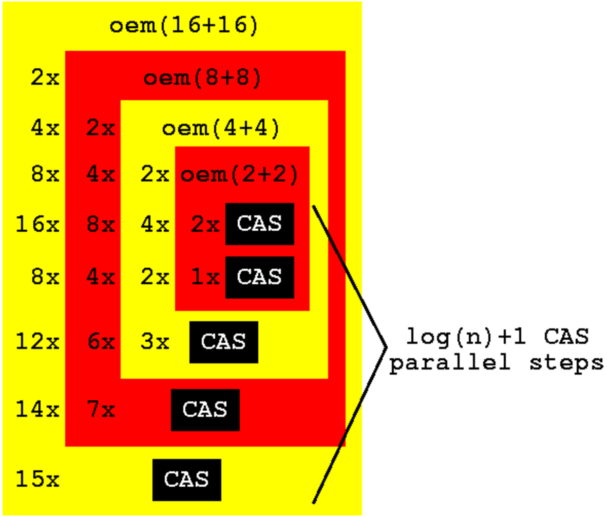
For each *odd_even_merge*() call with lists of size n, we can:

- Run in parallel the two recursive calls to *odd_even_merge*() for merging sorted sublists of size $\frac{n}{2}$
- Run in parallel the $n - 1$ compare-and-exchange operations

Since in total there are $\log(n) - 1$ recursive merging steps, it takes $\log(n) + 1$ parallel steps to obtain the final sorted list in a parallel implementation, which corresponds to a time complexity of $\mathcal{O}(\log(n))$:

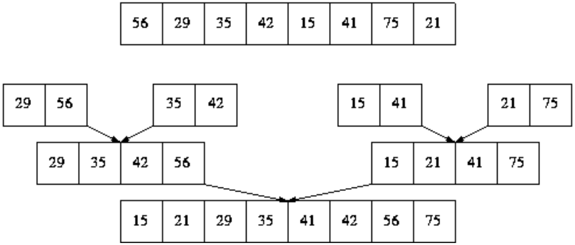
- 2 parallel steps for the compare-and-exchange operations of the last recursive calls to *odd_even_merge*() (sublists of size 2)
- $\log(n) - 1$ parallel steps for the remaining compare-and-exchange operations

Parallel Odd-Even Merge



Odd-Even Mergesort

Odd-even mergesort is a parallel sorting algorithm based on the **recursive application of the odd-even merge algorithm** that merges sorted sublists bottom up – starting with sublists of size 2 and merging them into bigger sublists – until the final sorted list is obtained.



How many steps does the algorithm?
Since in total there are $\log(n)$ sorting steps and each step (odd-even merge algorithm) corresponds to a time complexity of $\mathcal{O}(\log(n))$, the odd-even mergesort time complexity is thus $\mathcal{O}(\log(n)^2)$.

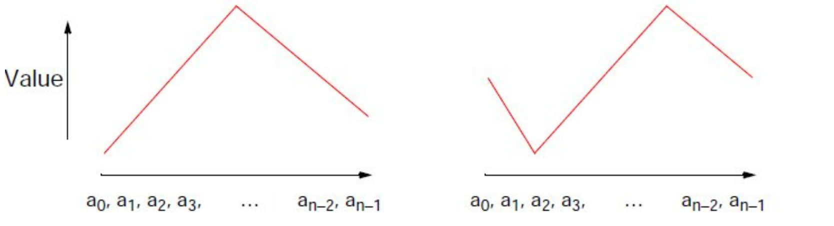
Bitonic Mergesort

The basis of bitonic mergesort is the **bitonic sequence**, a list having specific properties that are exploited by the sorting algorithm.

A sequence is considered bitonic if it contains **two sequences, one increasing and one decreasing**, such that:

$$a_1 < a_2 < \ldots < a_{i-1} < a_i > a_{i+1} > a_{i+2} > \ldots > a_n$$

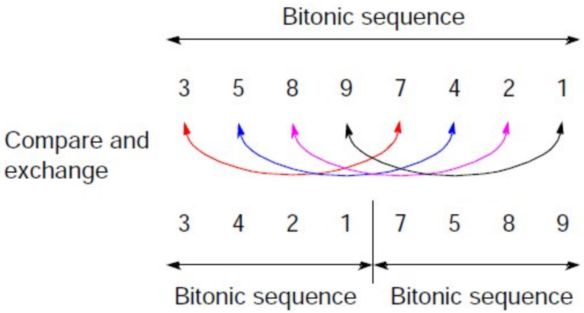
for some value i ($0 \leq i \leq n$). A sequence is also considered bitonic if the property is attained by shifting the numbers cyclically (left or right).


Value
a0, a1, a2, a3, ... an−2, an−1
a0, a1, a2, a3, ... an−2, an−1

(a) Single maximum (b) Single maximum and single minimum

Bitonic Mergesort

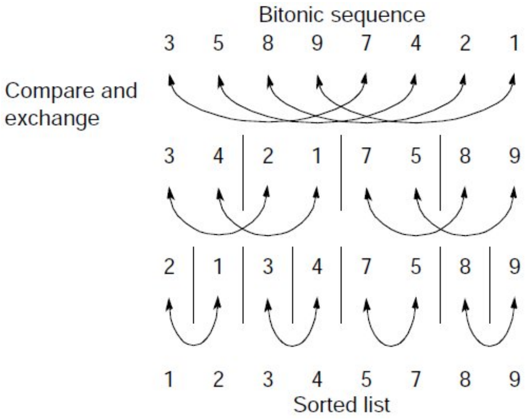
A **special characteristic** of bitonic sequences is that if we perform a compare-and-exchange operation with elements a_i and $a_{i+n/2}$ for all i $(0 \leq i \leq n/2)$ in a sequence of size n, we obtain **two bitonic sequences in which all the values in one sequence are smaller than the values of the other**.



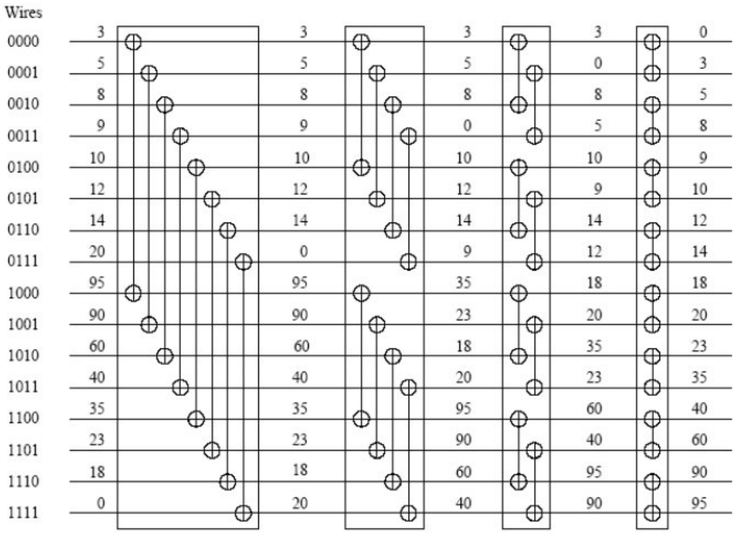
Bitonic Mergesort

In addition to both sequences being bitonic sequences, **all values in the left sequence are less than all the values in the right sequence**.

Hence, if we **apply recursively these compare-and-exchange operations** to a given bitonic sequence, we will get a **sorted sequence**.

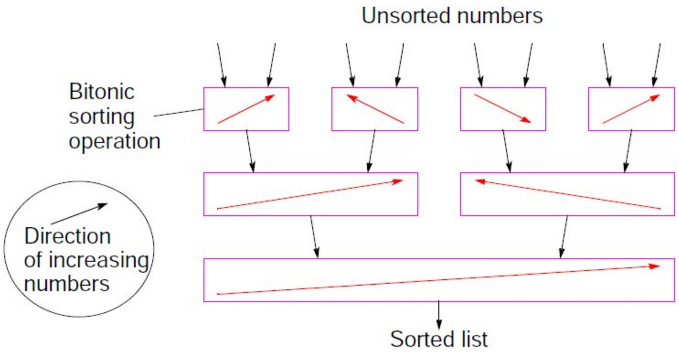


Bitonic Mergesort

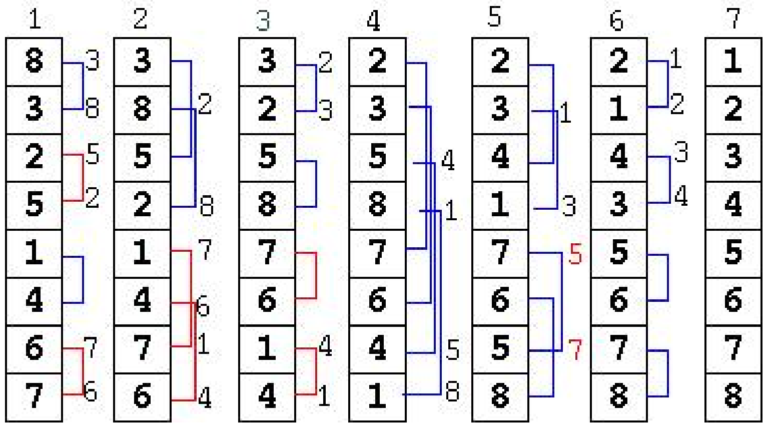


Bitonic Mergesort

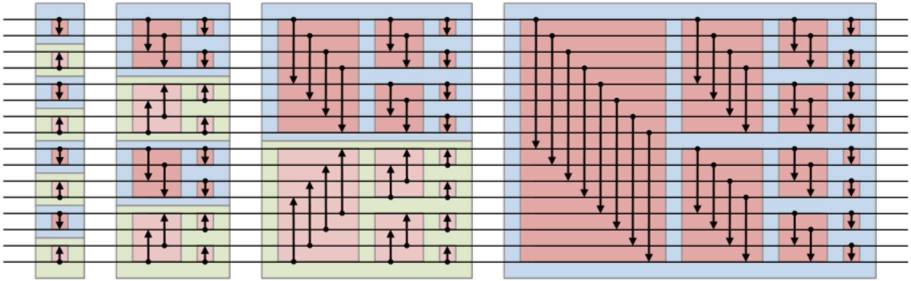
To sort an unsorted sequence, we **first transform it in a bitonic sequence**. Starting from adjacent pairs of values of the given unsorted sequence, bitonic sequences are created and then **recursively merged into (twice the size) larger bitonic sequences**. At the end, a single bitonic sequence is obtained, which is then sorted (as described before) into the final increasing sequence.



Bitonic Mergesort



Bitonic Mergesort



Bitonic Mergesort

The bitonic mergesort algorithm can be split into k phases (with $n = 2^k$):

- **Phase 1:** convert adjacent pair of values into increasing/decreasing sequences, i.e., into 4-bit bitonic sequences
- **Phases 2 to k-1:** sort each m-bit bitonic sequence into increasing/decreasing sequences and merge adjacent sequences into a $2m$-bit bitonic sequence, until reaching a n-bit bitonic sequence
- **Phase k:** sort the n-bit bitonic sequence

Given an unsorted list of size $n = 2^k$, there are k phases, with each phase i taking i parallel steps. Therefore, in total it takes

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2} = \frac{\log(n)(\log(n)+1)}{2}$$

parallel steps to obtain the final sorted list in a parallel implementation, which corresponds to a time complexity of $\mathcal{O}(\log(n)^2)$.

Time Complexity Summary

Algorithm	Sequential		Parallel	
Bubble Sort	$\mathcal{O}(n^2)$		$\mathcal{O}(n)$	*
Mergesort	$\mathcal{O}(n\log(n))$		$\mathcal{O}(n)$	
Quicksort	$\mathcal{O}(n\log(n))$	**	$\mathcal{O}(n)$	***
Odd-Even Mergesort	–		$\mathcal{O}(\log(n)^2)$	****
Bitonic Mergesort	–		$\mathcal{O}(\log(n)^2)$	

* odd-even transposition sort

** average time complexity

*** ideal pivot selection

**** recursive odd-even merge