# Algorithms and Applications

# Evaluating Algorithms

## Cost

*Processor-time* product or *cost* (or *work*) of a computation can be defined as

Cost = execution time × total number of processors used

Cost of a sequential computation simply its execution time, $t_s$.

Cost of a parallel computation is $t_p \times n$. Parallel execution time, $t_p$, is given by $t_s/S(n)$.

Hence, the cost of a parallel computationgiven by

$$\text{Cost} = \frac{t_s n}{S(n)} = \frac{t_s}{E}$$

# Cost-Optimal Parallel Algorithm

One in which the cost to solve a problem on a multiprocessor is proportional to the cost (i.e., execution time) on a single processor system.

Can be used to compare algorithms.

# Parallel Algorithm Time Complexity

Can derive the time complexity of a parallel algorithm in a similar manner as for a sequential algorithm by counting the steps in the algorithm (worst case) .

Following from the definition of cost-optimal algorithm

$$\text{(Cost) optimal parallel time complexity} = \frac{\text{sequential time complexity}}{\text{number of processors}}$$

But this does not take into account communication overhead. In textbook, calculated computation and communication separately.

Areas done in textbook:

- Sorting Algorithms
- Numerical Algorithms
- Image Processing
- Searching and Optimization

Chapter 9

# Sorting Algorithms

- rearranging a list of numbers into increasing (strictly non-decreasing) order.

# Potential Speedup

O($n \log n$) optimal for any sequential sorting algorithm without using special properties of the numbers.

Best we can expect based upon a sequential sorting algorithm using $n$ processors is

$$\text{Optimal parallel time complexity} = \frac{O(n \log n)}{n} = O(\log n)$$

Has been obtained but the constant hidden in the order notation extremely large.
Also an algorithm exists for an $n$-processor hypercube using random operations.

But, in general, a realistic O($\log n$) algorithm with $n$ processors not be easy to achieve.

# Sorting Algorithms Reviewed

- Rank sort
  (to show that an non-optimal sequential algorithm may in fact be a good parallel algorithm

- Compare and exchange operations
  (to show the effect of duplicated operations can lead to erroneous results)

- Bubble sort and odd-even transposition sort
- Two dimensional sorting - Shearsort (with use of transposition)
- Parallel Mergesort
- Parallel Quicksort
- Odd-even Mergesort
- Bitonic Mergesort

# Rank Sort

The number of numbers that are smaller than each selected number is counted. This count provides the position of selected number in sorted list; that is, its "rank."

First `a[0]` is read and compared with each of the other numbers, `a[1]` ... `a[n-1]`, recording the number of numbers less than `a[0]`.Suppose this number is *x*. This is the index of the location in the final sorted list. The number `a[0]` is copied into the final sorted list `b[0]` ... `b[n-1]`, at location `b[x]`. Actions repeated with the other numbers.

Overall sequential sorting time complexity of $O(n^2)$ (not exactly a good sequential sorting algorithm!).

# Sequential Code

```
for (i = 0; i < n; i++) {          /* for each number */
    x = 0;
    for (j = 0; j < n; j++)  /* count number less than it */
      if (a[i] > a[j]) x++;
    b[x] = a[i];       /* copy number into correct place */
}
```

This code will fail if duplicates exist in the sequence of numbers.

# Parallel Code
## Using *n* Processors

One processor allocated to each number. Finds final index in O(*n*) steps. With all processors operating in parallel, parallel time complexity O(*n*).

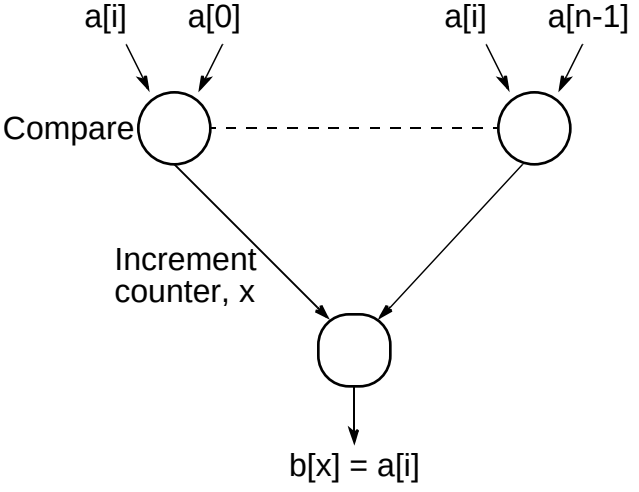
In **`forall`** notation, the code would look like

```
forall (i = 0; i < n; i++) {/* for each no in parallel*/
  x = 0;
  for (j = 0; j < n; j++)/* count number less than it */
    if (a[i] > a[j]) x++;
  b[x] = a[i];        /* copy no into correct place */
}
```

Parallel time complexity, O(*n*), better than any sequential sorting algorithm. Can do even better if we have more processors.

# Using $n^2$ Processors

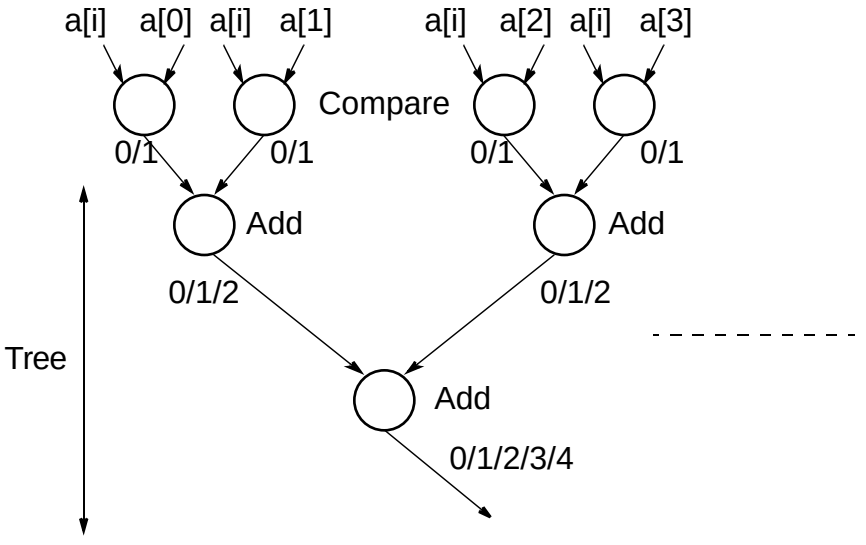
Comparing one number with the other numbers in list using multiple processors:




$n$ – 1 processors used to find rank of one number. With $n$ numbers, ($n$ – 1)$n$ processors or (almost) $n^2$ processors needed. Incrementing the counter done sequentially and requires maximum of $n$ steps.

# Reduction in Number of Steps

Tree to reduce number of steps involved in incrementing counter:




O(log$n$) algorithm with $n^2$ processors.
Processor efficiency relatively low.

# Parallel Rank Sort Conclusions

Easy to do as each number can be considered in isolation.

Rank sort can sort in:

$O(n)$ with $n$ processors

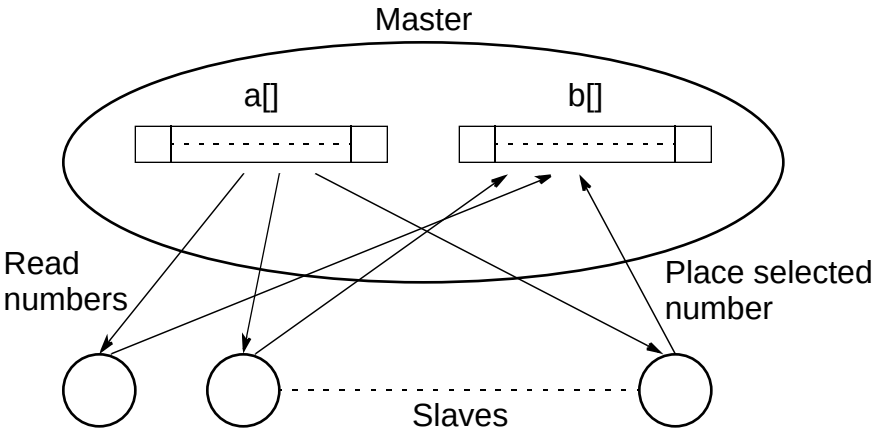
or

$O(\log n)$ using $n^2$ processors.

In practical applications, using $n^2$ processors prohibitive.

Theoretically possible to reduce time complexity to $O(1)$ by considering all increment operations as happening in parallel since they are independent of each other.

# Message Passing Parallel Rank Sort
## Master-Slave Approach




Requires shared access to list of numbers. Master process responds to request for numbers from slaves. Algorithm better for shared memory

# Compare-and-Exchange Sorting Algorithms

## Compare and Exchange

Form the basis of several, if not most, classical sequential sorting algorithms.

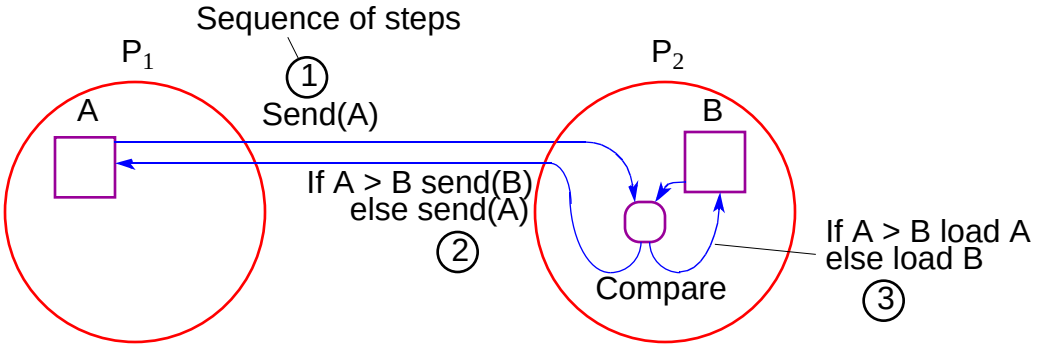
Two numbers, say *A* and *B*, are compared. If *A* > *B*, *A* and *B* are exchanged, i.e.:

```
if (A > B) {
  temp = A;
  A = B;
  B = temp;
}
```

# Message-Passing Compare and Exchange

## Version 1

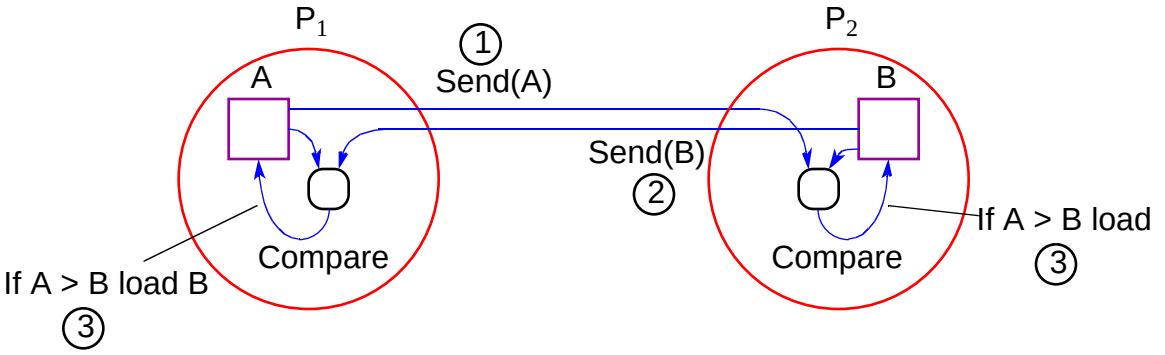
$P_1$ sends $A$ to $P_2$, which compares $A$ and $B$ and sends back $B$ to $P_1$ if $A$ is larger than $B$ (otherwise it sends back $A$ to $P_1$):



# Alternative Message Passing Method

## Version 2

For $P_1$ to send $A$ to $P_2$ and $P_2$ to send $B$ to $P_1$. Then both processes perform compare operations. $P_1$ keeps the larger of $A$ and $B$ and $P_2$ keeps the smaller of $A$ and $B$:



# Note on Precision of Duplicated Computations

Previous code assumes that the `if` condition, `A > B`, will return the same Boolean answer in both processors.

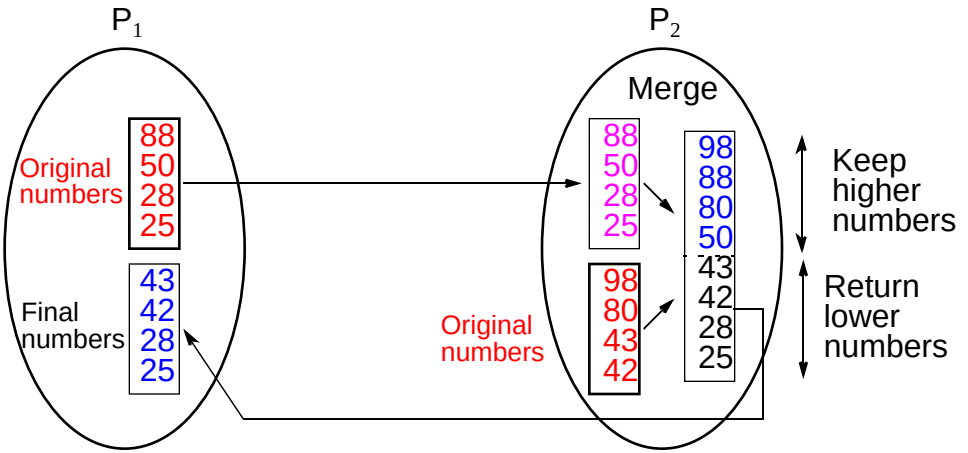
Different processors operating at different precision could conceivably produce different answers if real numbers are being compared.

This situation applies to anywhere computations are duplicated in different processors to reduce message passing, or to make the code SPMD.

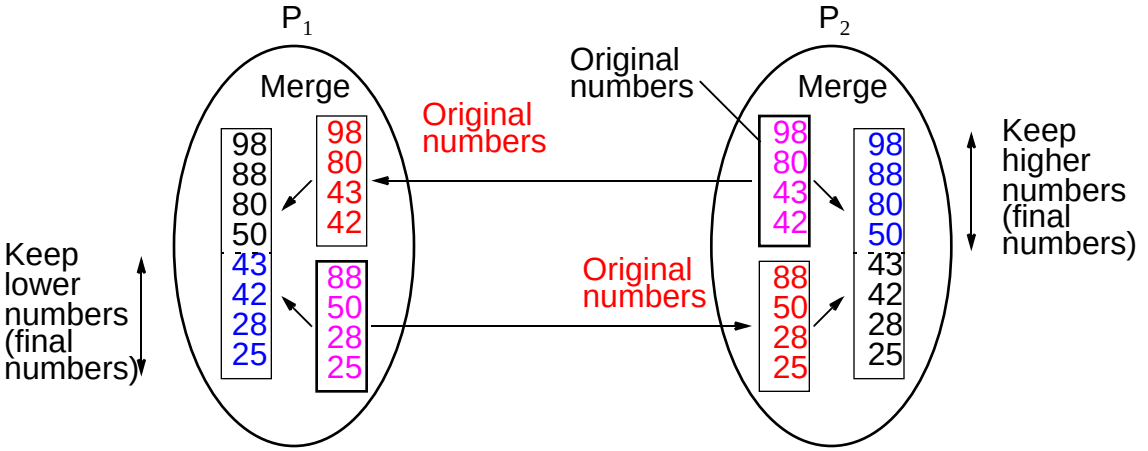
# Data Partitioning

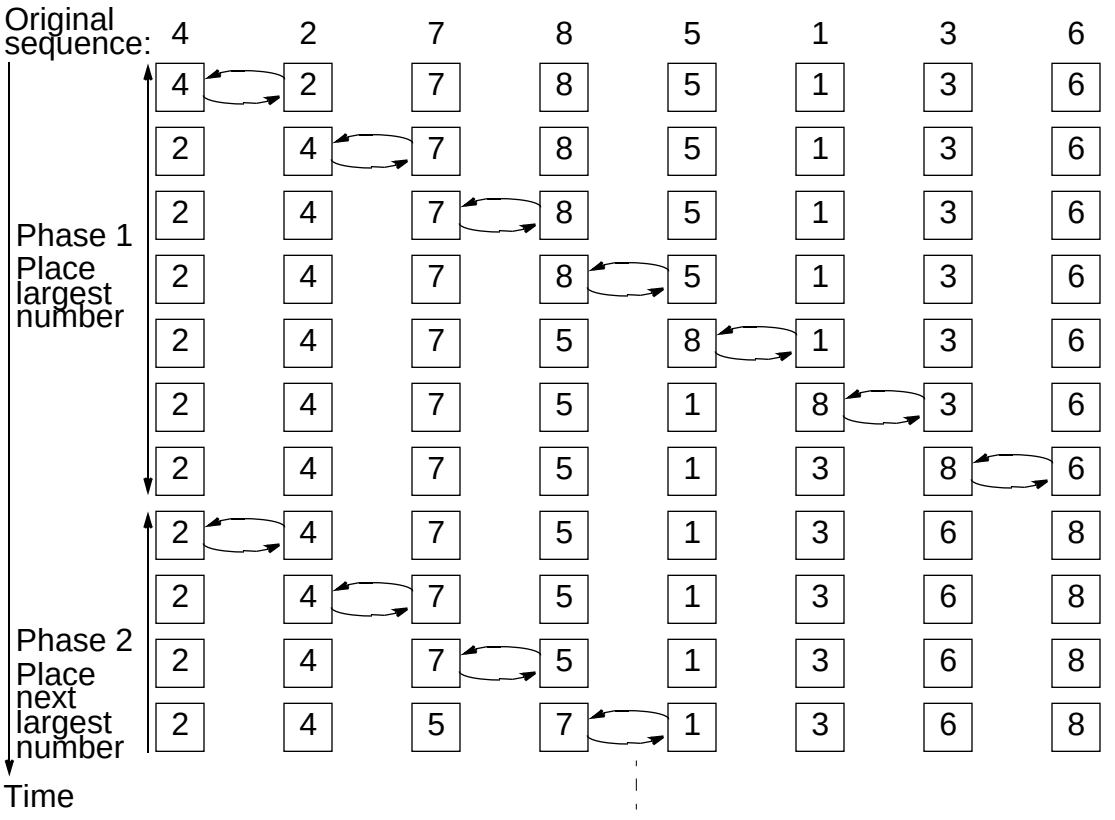
## (Version 1)

*p* processors and *n* numbers. *n*/*p* numbers assigned to each processor:



# Merging Two Sublists — Version 2





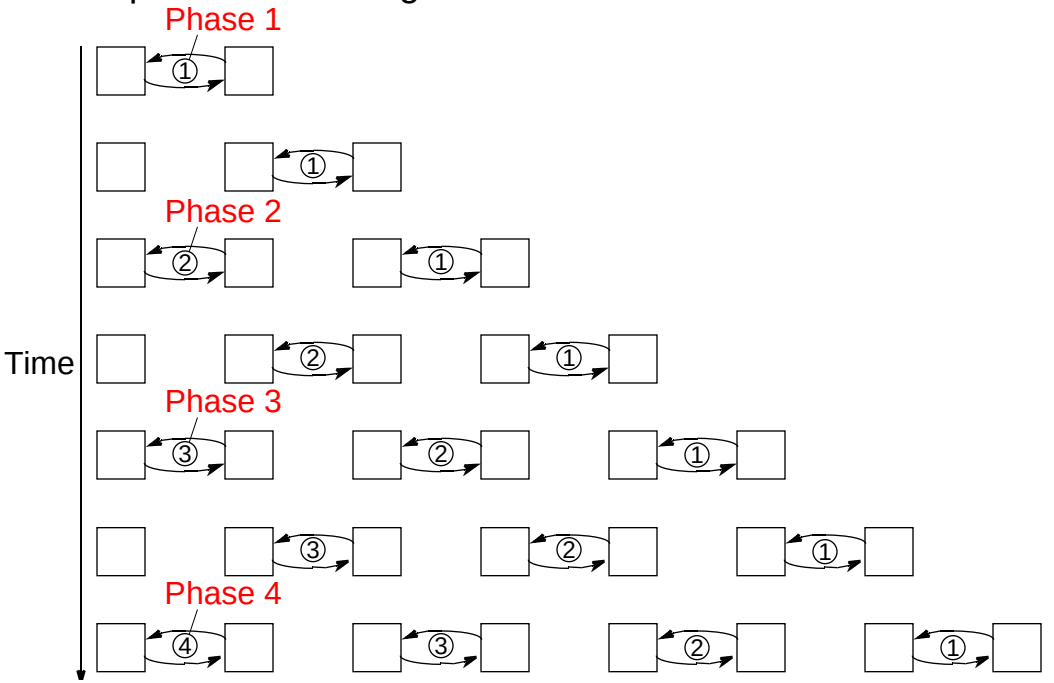
# Time Complexity

$$\text{Number of compare and exchange operations} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

which indicates a time complexity of $O(n^2)$ given that a single compare-and-exchange operation has a constant complexity, $O(1)$.

# Parallel Bubble Sort

Iteration could start before previous iteration finished if does not overtake previous bubbling action:



# Odd-Even (Transposition) Sort

Variation of bubble sort.

Operates in two alternating phases, *even* phase and *odd* phase.

## Even phase

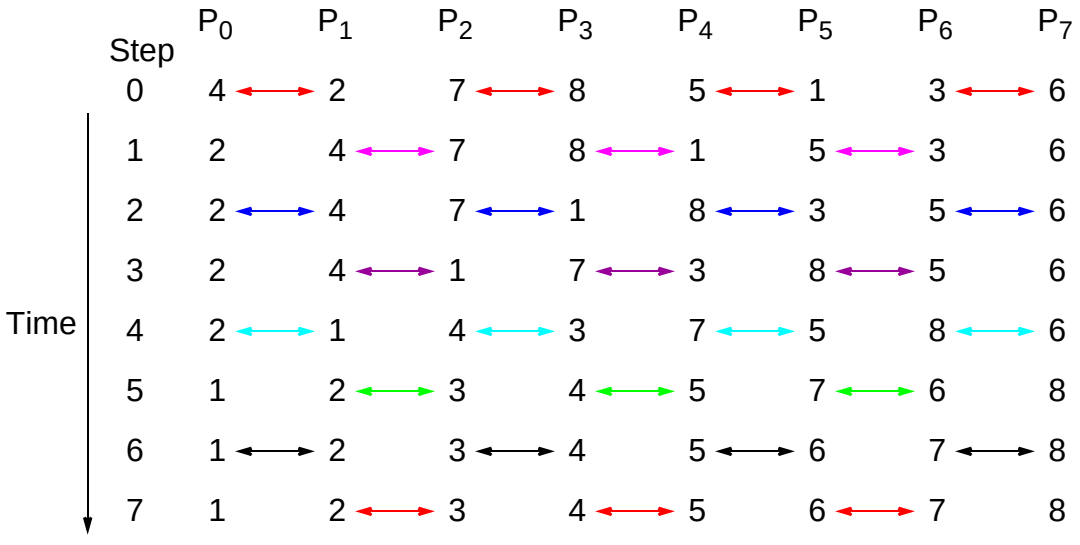
Even-numbered processes exchange numbers with their right neighbor.

## Odd phase

Odd-numbered processes exchange numbers with their right neighbor.

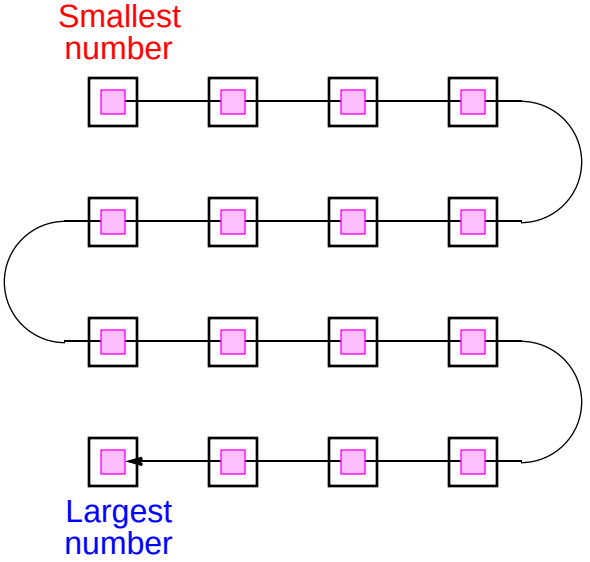
# Odd-Even Transposition Sort

## Sorting eight numbers



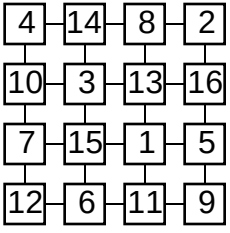
# Two-Dimensional Sorting

The layout of a sorted sequence on a mesh could be row by row or *snakelike*. Snakelike:

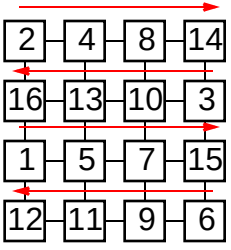


# Shearsort

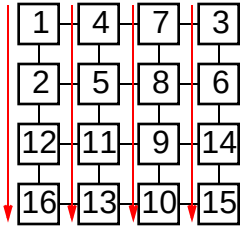
Alternate row and column sorting until list fully sorted. Row sorting alternative directions to get snake-like sorting:



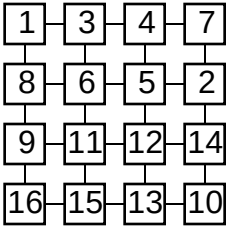

(a) Original placement of numbers



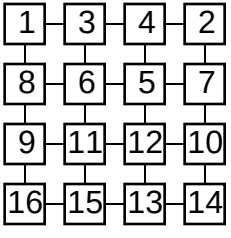

(b) Phase 1 — Row sort



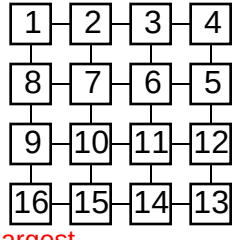

(c) Phase 2 — Column sort




(d) Phase 3 — Row sort




(e) Phase 4 — Column sort



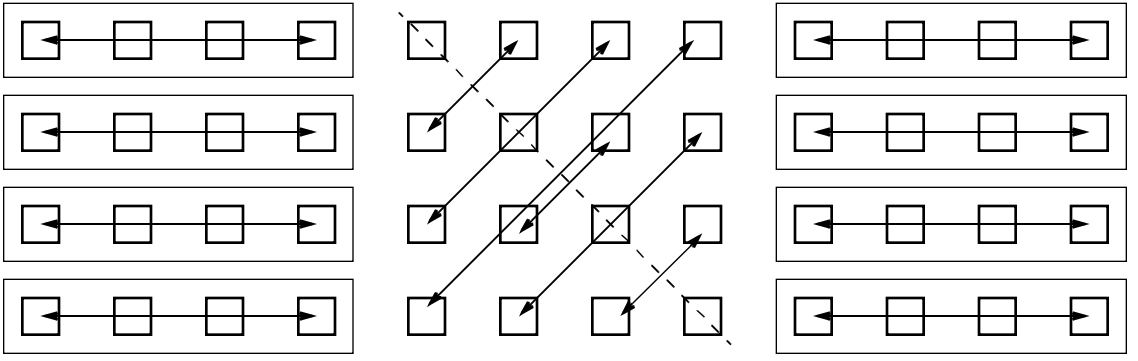

(f) Final phase — Row sort

# Shearsort

Requires $\sqrt{n}(\log n + 1)$ steps for $n$ numbers on a $\sqrt{n} \times \sqrt{n}$ mesh.

# Using Transposition

Causes the elements in each column to be in positions in a row.
Can be placed between the row operations and column operations:



(a) Operations between elements in rows

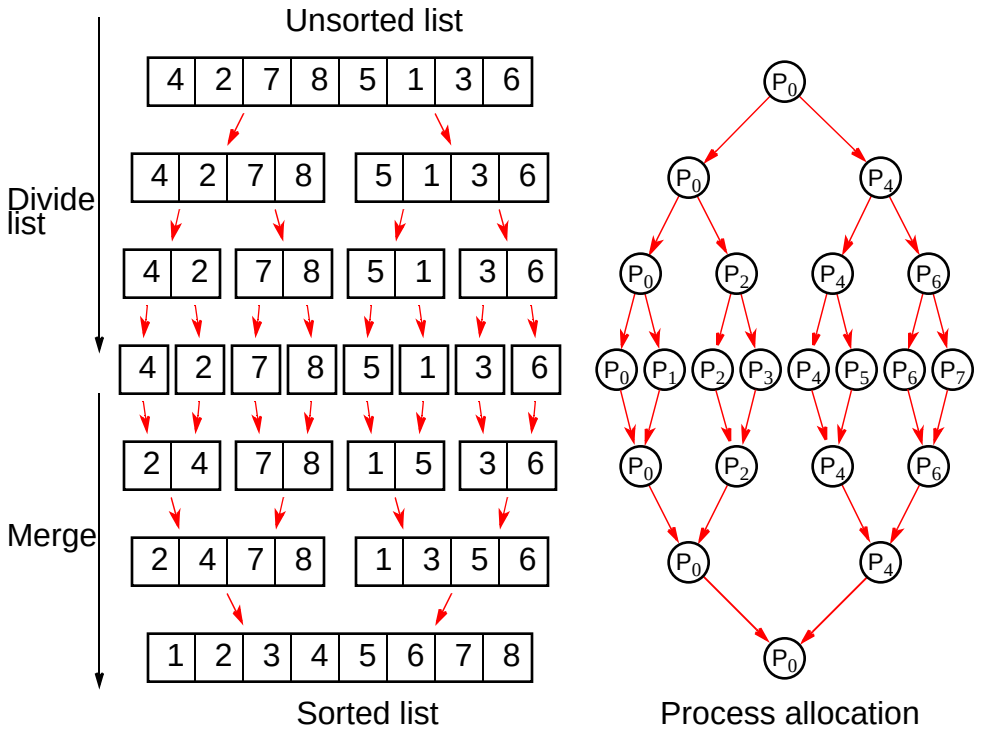
(b) Transpose operation

(c) Operations between elements in rows (originally columns)

Transposition can be achieved with $\sqrt{n}(\sqrt{n} - 1)$ communications ($O(n)$). An *all-to-all* routine could be reduce this.

# Parallelizing Mergesort

Using tree allocation of processes



# Analysis

## Sequential

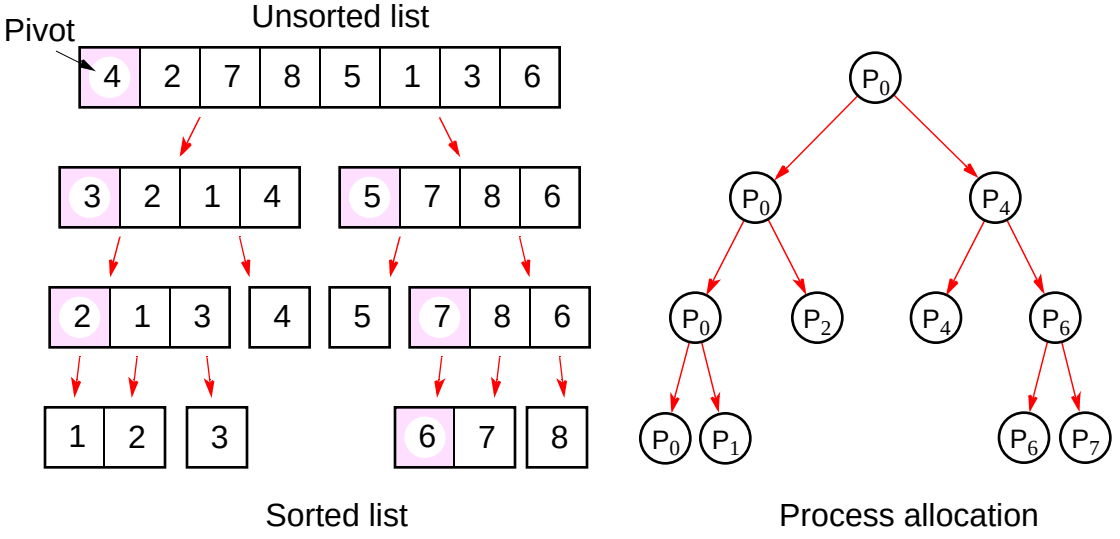
Sequential time complexity is $O(n \log n)$.

## Parallel

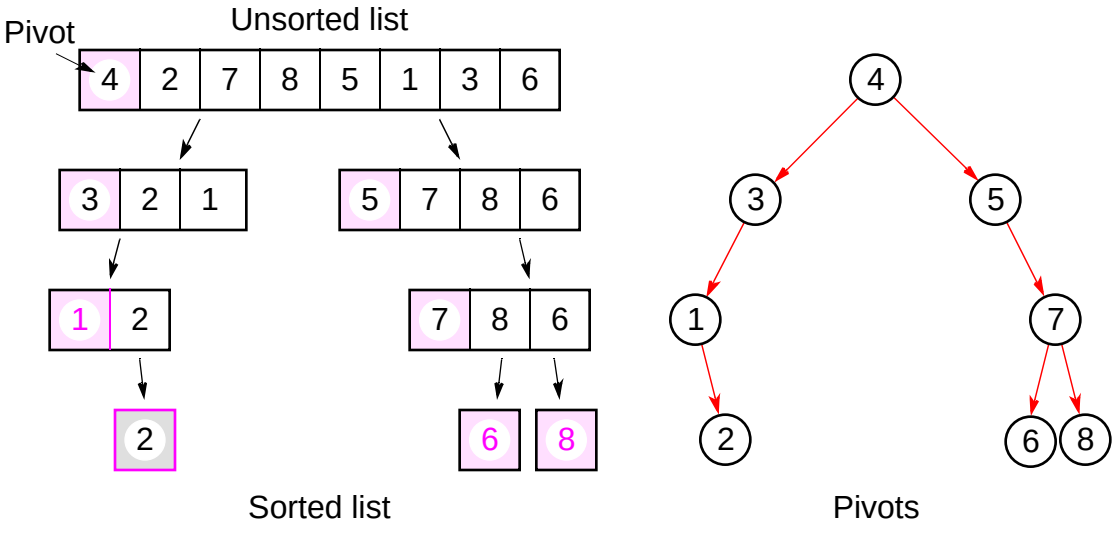
2 log *n* steps in the parallel version but each step may need to perform more than one basic operation, depending upon the number of numbers being processed - see text.

# Parallelizing Quicksort

Using tree allocation of processes



# With the pivot being withheld in processes:



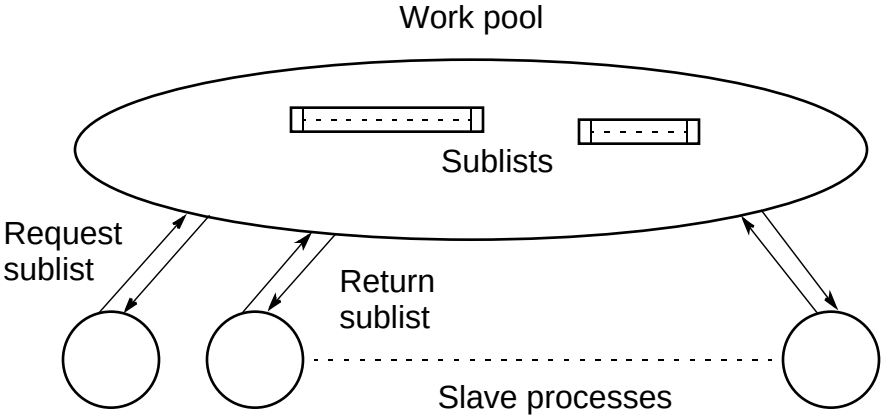
# Analysis

Fundamental problem with all tree constructions – initial division done by a single processor, which will seriously limit speed.

Tree in quicksort will not, in general, be perfectly balanced Pivot selection very important to make quicksort operate fast.

# Work Pool Implementation of Quicksort

First, work pool holds initial unsorted list. Given to first processor which divides list into two parts. One part returned to work pool to be given to another processor, while the other part operated upon again.



Neither Mergesort nor Quicksort parallelize very well as the processor efficiency is low (see book for analysis).

Quicksort also can be very unbalanced. Can use load balancing techniques

Parallel hypercube versions of quicksort in textbook - however hypercubes not now of much interest.

# Batcher's Parallel Sorting Algorithms

- Odd-even Mergesort
- Bitonic Mergesort

Originally derived in terms of switching networks.

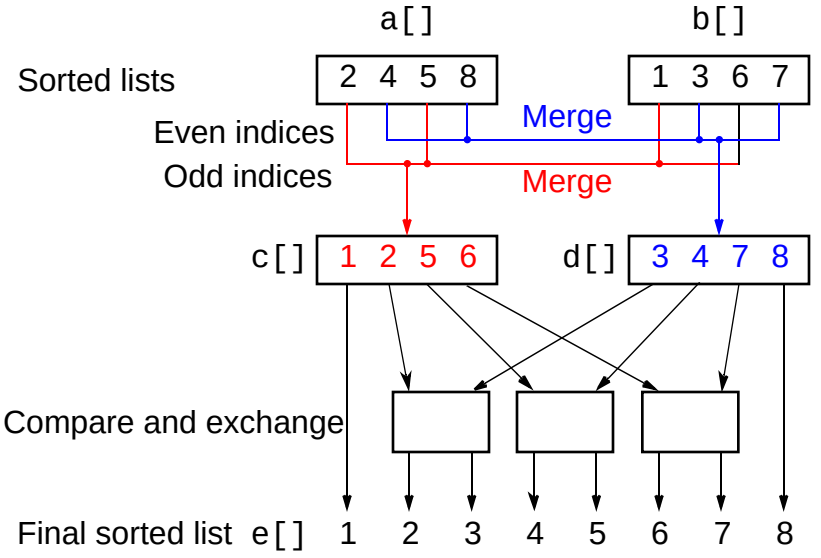
Both are well balanced and have parallel time complexity of $O(\log^2 n)$ with n processors.

# Odd-Even Mergesort

## Odd-Even Merge Algorithm

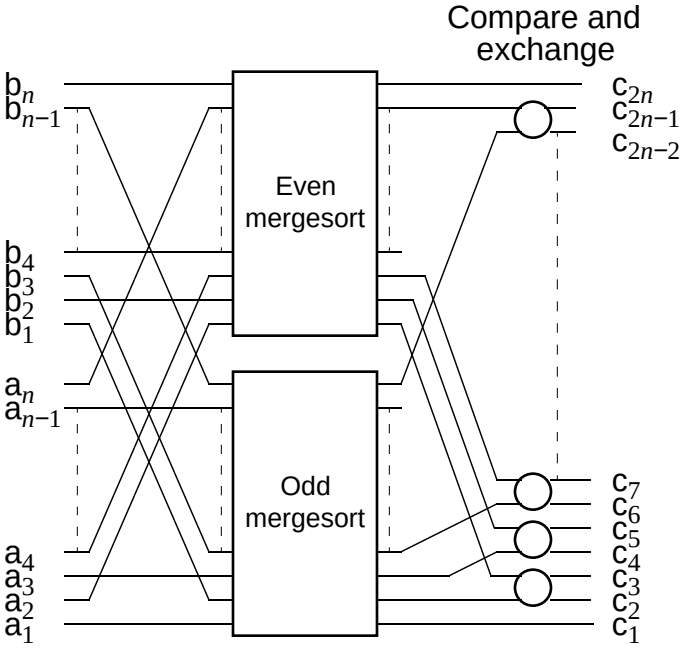
Start with odd-even merge algorithm which will merge two *sorted* lists into one sorted list. Given two sorted lists $a_1$, $a_2$, $a_3$, …, $a_n$ and $b_1$, $b_2$, $b_3$, …, $b_n$ (where $n$ is a power of 2)

# Odd-Even Merging of Two Sorted Lists



# Odd-Even Mergesort

Apply odd-even merging recursively



# Bitonic Mergesort

## Bitonic Sequence

A monotonic increasing sequence is a sequence of increasing numbers.

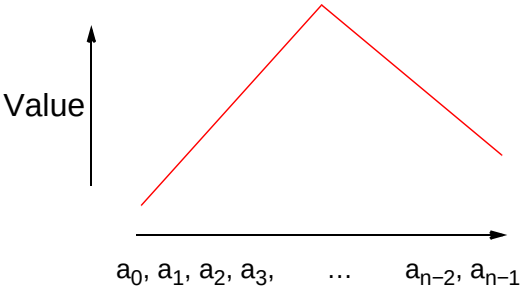
A *bitonic sequence* has two sequences, one increasing and one decreasing. e.g.

$$a_0 < a_1 < a_2, a_3, \ldots, a_{i-1} < a_i > a_{i+1}, \ldots, a_{n-2} > a_{n-1}$$

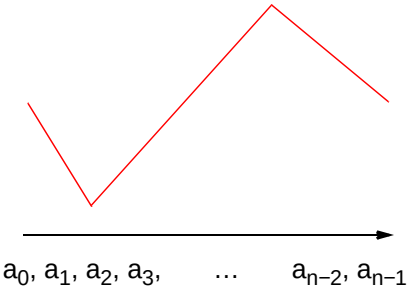
for some value of $i$ ($0 \leq i < n$).

A sequence is also bitonic if the preceding can be achieved by shifting the numbers cyclically (left or right).

# Bitonic Sequences




(a) Single maximum




(b) Single maximum and single minimum

# "Special" Characteristic of Bitonic Sequences

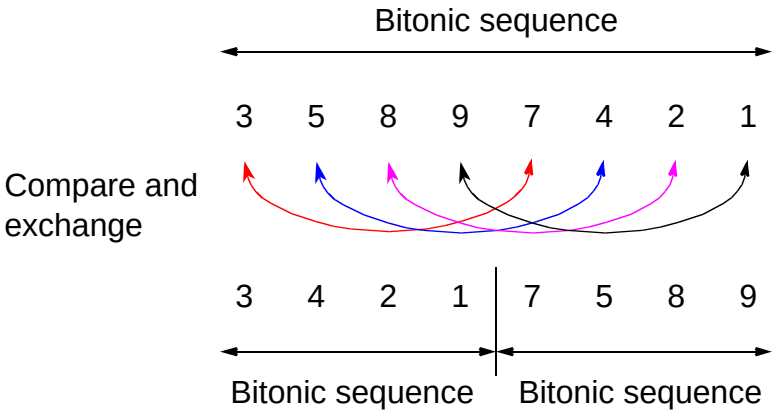
If we perform a compare-and-exchange operation on $a_i$ with $a_{i+n/2}$ for all $i$ , where there are $n$ numbers in the sequence, get **TWO** bitonic sequences, where the numbers in one sequence are **all less than the numbers in the other sequence**.

# Example - Creating two bitonic sequences from one bitonic sequence

Starting with the bitonic sequence

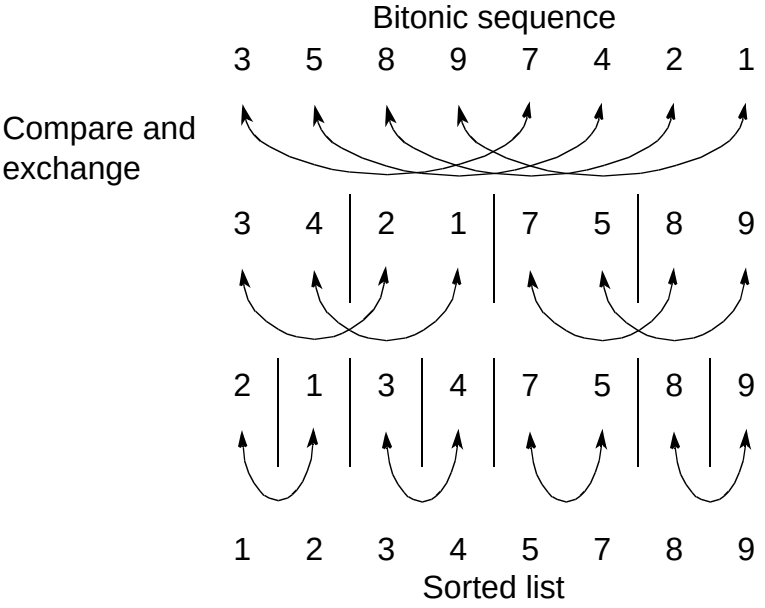
3, 5, 8, 9, 7, 4, 2, 1

we get:



# Sorting a bitonic sequence

Compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right. Given a bitonic sequence, recursively performing operations will sort the list.



# Sorting

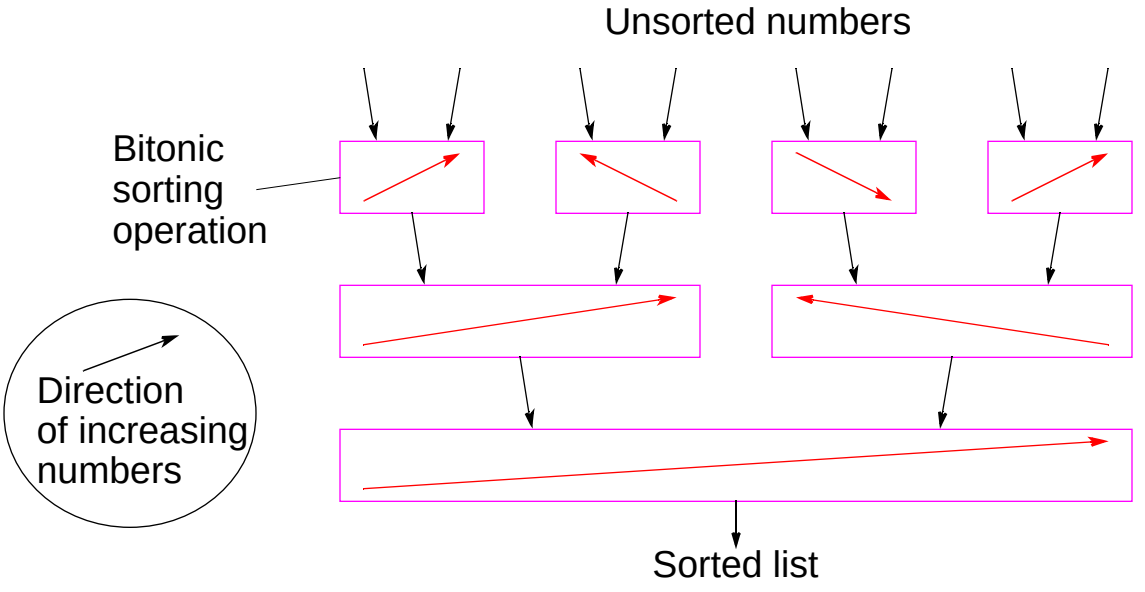
To sort an unordered sequence, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers.

By a compare-and-exchange operation, pairs of adjacent numbers are formed into increasing sequences and decreasing sequences, pairs of which form a bitonic sequence of twice the size of each of the original sequences.

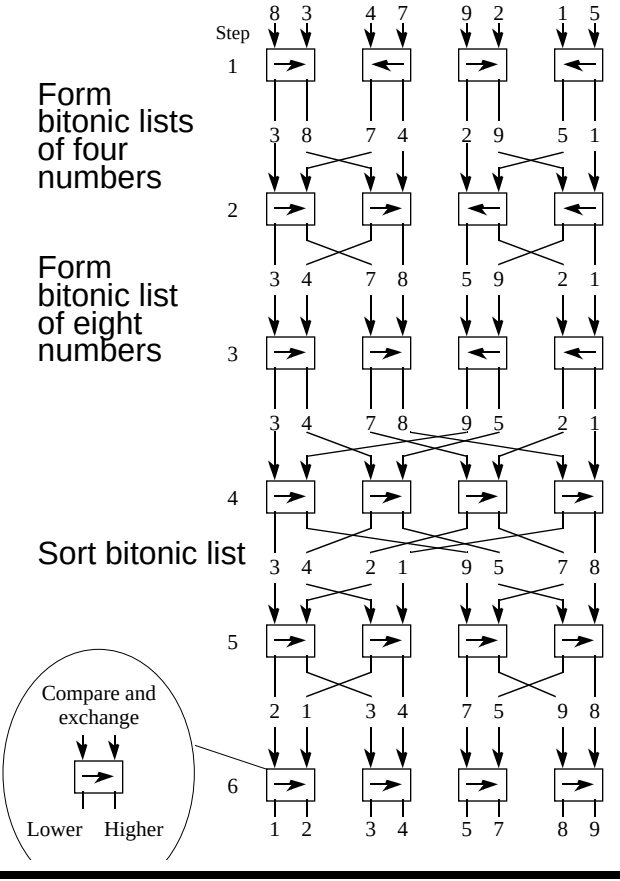
By repeating this process, bitonic sequences of larger and larger lengths are obtained.

In the final step, a single bitonic sequence is sorted into a single increasing sequence.

# Bitonic Mergesort



# Bitonic Mergesort on Eight Numbers



# Phases

The six steps (for eight numbers) are divided into three phases:

Phase 1 (Step 1) Convert pairs of numbers into increasing/ decreasing sequences and hence into 4-bit bitonic sequences.

Phase 2 (Steps 2/3) Split each 4-bit bitonic sequence into two 2-bit bitonic sequences, higher sequences at center.

Sort each 4-bit bitonic sequence increasing/ decreasing sequences and merge into 8-bit bitonic sequence.

Phase 3 (Steps 4/5/6) Sort 8-bit bitonic sequence

# Number of Steps

In general, with $n = 2^k$, there are $k$ phases, each of 1, 2, 3, …, $k$ steps. Hence the total number of steps is given by

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

# Sorting Conclusions

Computational time complexity using n processors

- Ranksort O(n)
- Odd-even transposition sort- O(n)
- Parallel mergesort - O(n) but unbalanced processor load and communication
- Parallel quicksort - O(n) but unbalanced processor load, and communication can generate to $O(n^2)$
- Odd-even Mergesort and Bitonic Mergesort $O(\log^2 n)$

Bitonic mergesort has been a popular choice for a parallel sorting.