Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Français
Հայերեն
Edit links

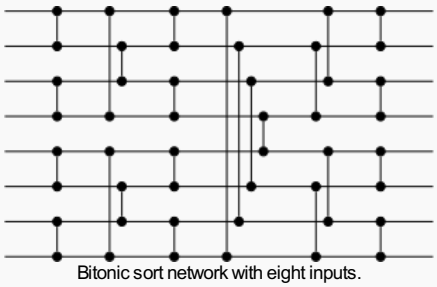
Bitonic sorter

From Wikipedia, the free encyclopedia

Bitonic mergesort is a parallel algorithm for sorting. It is also used as a construction method for building a sorting network. The algorithm was devised by Ken Batcher. The resulting sorting networks consist of $O(n \log(n)^2)$ comparators and have a delay of $O(\log(n)^2)$, where n is the number of items to be sorted.[1]

A sorted sequence is a monotonically non-decreasing (or non-increasing) sequence. A *bitonic* sequence is a sequence with $x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$ for some $k, 0 \leq k < n$, or a circular shift of such a sequence.

Bitonic sorter



Bitonic sort network with eight inputs.

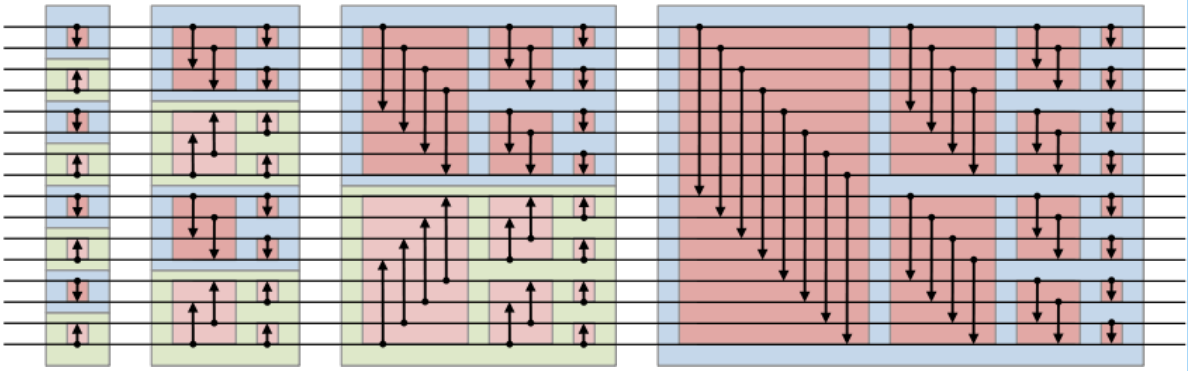
Class	Sorting algorithm
Data structure	Array
Worst case performance	$O(\log(n)^2)$ parallel time
Best case performance	$O(\log(n)^2)$ parallel time
Average case performance	$O(\log(n)^2)$ parallel time
Worst case space complexity	$O(n \log(n)^2)$ comparators

Contents [hide]

1 How the algorithm works
2 Example code
3 See also
4 References
5 External links

How the algorithm works [edit]

The following is a bitonic sorting network with 16 inputs:



The 16 numbers enter at the inputs at the left end, slide along each of the 16 horizontal wires, and exit at the outputs at the right end. The network is designed to sort the elements, with the largest number at the bottom.

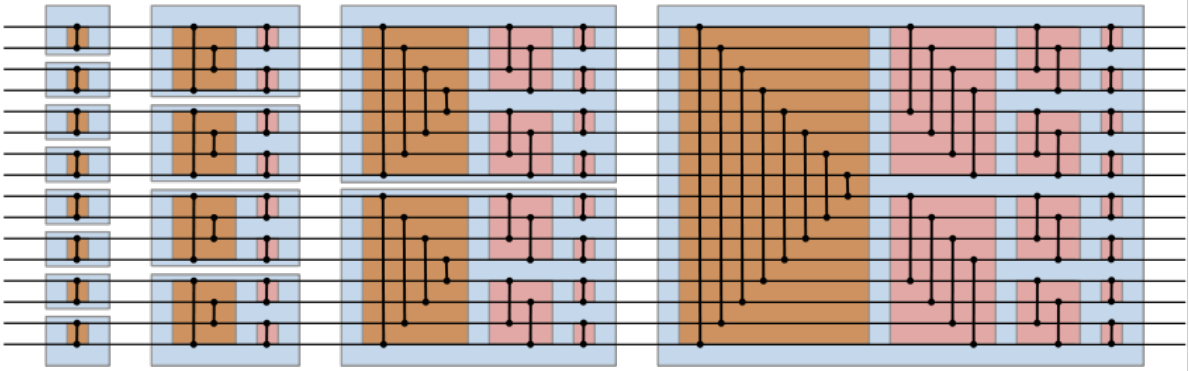
The arrows are comparators. Whenever two numbers reach the two ends of an arrow, they are compared to ensure that the arrow points toward the larger number. If they are out of order, they are swapped. The colored boxes are just for illustration and have no effect on the algorithm.

Every red box has the same structure: each input in the top half is compared to the corresponding input in the bottom half, with all arrows pointing down (dark red) or all up (light red). If the inputs happen to form a bitonic sequence, then the output will form two bitonic sequences. The top half of the output will be bitonic, and the bottom half will be bitonic, with every element of the top half less than or equal to every element of the bottom half (for dark red) or vice versa (for light red). This theorem is not obvious, but can be verified by carefully considering all the cases of how the various inputs might compare, using the zero-one principle.

The red boxes combine to form blue and green boxes. Every such box has the same structure: a red box is applied to the entire input sequence, then to each half of the result, then to each half of each of those results, and so on. All arrows point down (blue) or all point up (green). This structure is known as a butterfly network. If the input to this box happens to be bitonic, then the output will be completely sorted in increasing order (blue) or decreasing order (green). If a number enters the blue or green box, then the first red box will sort it into the correct half of the list. It will then pass through a smaller red box that sorts it into the correct quarter of the list within that half. This continues until it is sorted into exactly the correct position. Therefore, the output of the green or blue box will be completely sorted.

The green and blue boxes combine to form the entire sorting network. For any arbitrary sequence of inputs, it will sort them correctly, with the largest at the bottom. The output of each green or blue box will be a sorted sequence, so the output of each pair of adjacent lists will be bitonic, because the top one is blue and the bottom one is green. Each column of blue and green boxes takes N sorted sequences and concatenates them in pairs to form N/2 bitonic sequences, which are then sorted by the boxes in that column to form N/2 sorted sequences. This process starts with each input considered to be a sorted list of one element, and continues through all the columns until the last merges them into a single, sorted list. Because the last stage was blue, this final list will have the largest element at the bottom.

Each green box performs the same operation as a blue box, but with the sort in the opposite direction. So, each green box could be replaced by a blue box followed by a crossover where all the wires move to the opposite position. This would allow all the arrows to point the same direction, but would prevent the horizontal lines from being straight. However, a similar crossover could be placed to the right of the bottom half of the outputs from any red block, and the sort would still work correctly, because the reverse of a bitonic sequence is still bitonic. If a red box then has a crossover before and after it, it can be rearranged internally so the two crossovers cancel, so the wires become straight again. Therefore, the following diagram is equivalent to the one above, where each green box has become a blue plus a crossover, and each orange box is a red box that absorbed two such crossovers:



The arrowheads are not drawn, because every comparator sorts in the same direction. The blue and red blocks perform the same operations as before. The orange blocks are equivalent to red blocks where the sequence order is reversed for the bottom half of its inputs and the bottom half of its outputs. This is the most common representation of a bitonic sorting network

Example code [edit]

The following is an implementation of the bitonic mergesort sorting algorithm in Python. The input is a boolean value *up*, and a list *x* of length a power of 2. The output is a sorted list that is ascending if *up* is true, and decreasing otherwise.

```
def bitonic_sort(up, x):
    if len(x) <= 1:
        return x
    else:
        first = bitonic_sort(True, x[:len(x) / 2])
        second = bitonic_sort(False, x[len(x) / 2:])
        return bitonic_merge(up, first + second)

def bitonic_merge(up, x):
    # assume input x is bitonic, and sorted list is returned
    if len(x) == 1:
        return x
    else:
        bitonic_compare(up, x)
        first = bitonic_merge(up, x[:len(x) / 2])
        second = bitonic_merge(up, x[len(x) / 2:])
        return first + second

def bitonic_compare(up, x):
    dist = len(x) / 2
    for i in range(dist):
        if (x[i] > x[i + dist]) == up:
            x[i], x[i + dist] = x[i + dist], x[i] #swap
```

```
>>> bitonic_sort(True, [10, 30, 11, 20, 4, 330, 21, 110])
[4, 10, 11, 20, 21, 30, 110, 330]
>>> bitonic_sort(False, [10, 30, 11, 20, 4, 330, 21, 110])
[330, 110, 30, 21, 20, 11, 10, 4]
```

See also [edit]

- Batcher odd–even mergesort

References [edit]

1. ^ Bitonic sorting network for n not a power of 2

External links [edit]

- A discussion of this algorithm
- Reference code at NIST
- Tutorial with animated pictures and working code
- Experimental Analysis of Parallel Sorting Algorithms

v · t · e	Sorting algorithms [hide]
Theory	Computational complexity theory · Big O notation · Total order · Lists · Inplacement · Stability · Comparison sort · Adaptive sort · Sorting network · Integer sorting
Exchange sorts	Bubble sort · Cocktail sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Stooge sort · Bogosort
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort · Strand sort
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsort · Counting sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort
Concurrent sorts	**Bitonic sorter** · Batcher odd–even mergesort · Pairwise sorting network
Hybrid sorts	Block sort · Timsort · Introsort · Spreadsort · JSort
Other	Topological sorting · Pancake sorting · Spaghetti sort

Categories: Sorting algorithms