AN INTRODUCTION TO PARALLEL COMPUTING

CHAPTER 7: SORTING

BY FRED ANNEXSTEIN, PHD

7. SORTING

CHAPTER OBJECTIVES

In this chapter we study the problem of sorting in parallel. We consider sorting on a PRAM model as well as on distributed networks. We compare and contrast the computational complexity of a variety of approaches. By the end of this chapter you should be able to …..

1. Understand the difficulties and opportunities of parallelizing popular sequential sorting algorithms.
2. Understand the impact of sorting methods based on item comparisons versus non-comparison based methods.
3. Understand the construction and the complexity of sorting networks.
4. Understand and apply Knuth's 0/1 Sorting Principle to prove correctness

of sorting algorithms and sorting networks.
5. Understand direct sorting and mesh-based parallel sorting methods.
6. Understand Batcher's Bitonic sorting method and Radix sorting algorithms and their work and step complexity.

THE SORTING PROBLEM

Sorting algorithms are interesting to study in the context of parallel computing in that they provide a set of potential communication patterns needed in numerous applications.

When discussing the process of sorting records using computer algorithms we often make a distinction between comparison-based sorting versus other methods usually based on specified encoding.

Here to start, is an impractical but theoretically fast comparison-based parallel sorting algorithm called counting sort.

```
def countingSort(list):
    n = len(list)
    scatter = n * [0]
    result = n * [0]
```

```
# do for loop in parallel to count number items
# less than the i-th item

    for i in range(n):
        scatter[i] = len(compact(lessthan(list[i]),
                                          list))
        result[scatter[i]] = list[i]
    return result
```

Counting sort is a method that does O(n^2) work since every pair of items in the original list of length n, however the number of steps is very fast O(log n). The large amount of work done by this algorithm makes this an impractical algorithm.

Counting sort is comparison-based because it is based on a basic operation of lessthan of pairs of items.

Many parallel approaches to sorting algorithms use as basic operation a compare-and-exchange (CE). The basic CE operations to sort N elements is at least N log N. This N log N lower bound for sequential sorting is proved in many college Algorithms courses. Also, you should be aware that with compare and exchange, if we limit ourselves to only exchanging near neighbors than like counting sort we have complexity of O(n^2) work. This result is known as the Adjacent-key Sorting bound.

OPTIMAL AND PRACTICAL PARALLEL SORTING ALGORITHMS

We considered a simplified quick sort in Chapter 2 and argued that such an algorithm naturally lends itself to parallel implementation. However, it is only optimal on average, and has poor worst case performance.

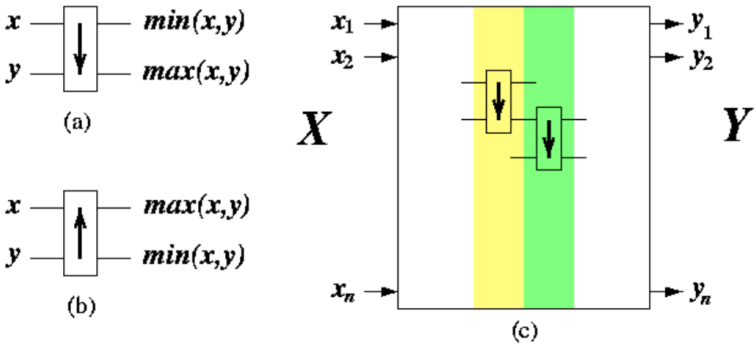
There do exist work and step optimal PRAM parallel sorting algorithms, however these are significantly more complex. For example, there is an algorithm known as Cole's Merge sort. This algorithm sorts n numbers in PRAM model using n processors in optimal time O(log n). Unlike Quicksort, a practical implementation of this algorithm is not known.

There exists a family of practical parallel algorithms based on a result known as Batcher's Bitonic-Merge (BBM). BBM has has efficient implementation, however the work and step complexity is sub-optimal. The work is O(n log^2 n) and the number of steps is O(log^2 n).

Near optimal mesh-based sorting algorithms exist. For example, given N=n^2 numbers N=nxn node 2D mesh, the numbers can be sorted on the mesh in time 3n. This is near optimal since we can see that the number of sorting steps must be at least the diameter of the mesh, which is 2n-2.

SORTING NETWORKS

Sorting networks are constructed from collections of parallel compare-exchange operations. In an exchange, pairs of processes, say Pi and Pj, send their data elements to each other. Process Pi keeps the minimum min{ai,aj}, and Pj keeps the maximum max{ai, aj}. Parallel CE-Network sorting algorithms can be written for either the PRAM model or distributed memory computers, and the later is often simply called a sorting network.



Formally, a sorting network is an end-to-end network composed of columns of comparator CE-nodes.

An unsorted input sequence placed on input wires of the leftmost column passes through the network so that it becomes a sorted output sequence at the rightmost column.

The amount of work a network does is defined as the number of comparators. The number of steps of the network, sometimes called the depth of the network, is defined as the longest chain of comparators from the left column to the right column.

Oblivious sorting is term used to describe algorithms that map to sorting networks since they have a control flow that is independent of specific properties of the data and only change flow based on outcomes of comparisons. Sorting networks are by their definition oblivious sorting algorithms.

KNUTH'S 0-1 SORTING PRINCIPLE

Oblivious sorting algorithms are easy to design and analyze due to a result known as *Knuth's 0-1 Sorting Principle*. This result

states that if a sorting network (or any oblivious sorting method) works correctly for every input sequence of 0's and 1's, then it also works correctly on any input taken from any linearly ordered set. Numbers naturally are linearly ordered. This sorting principle is extremely useful for proving correctness of oblivious sorting algorithms.

Knuth's 0-1 Sorting Principle: If a comparison network with N inputs sorts all 2^N possible sequences of 0's and 1's correctly, then it sorts all input sequences of arbitrary numbers correctly.

NETWORK-BASED DIRECT SORTING

Given a network of n processing nodes, each with one input number. We can impose a total ordering on nodes by numbering them P1,P2,…,Pn. Usually we insist that such orderings are based on adjacency so that adjacent nodes in the ordering are adjacent in the network. The direct sorting problem is equivalent to finding a unique permutation of input data such that each node Pi holds smaller number than Pi+1, using only exchanges of numbers between logically

adjacent nodes. 2D Meshes lend themselves to several total orderings for direct sorting, such as row-major, column-major, and snake-like orderings.

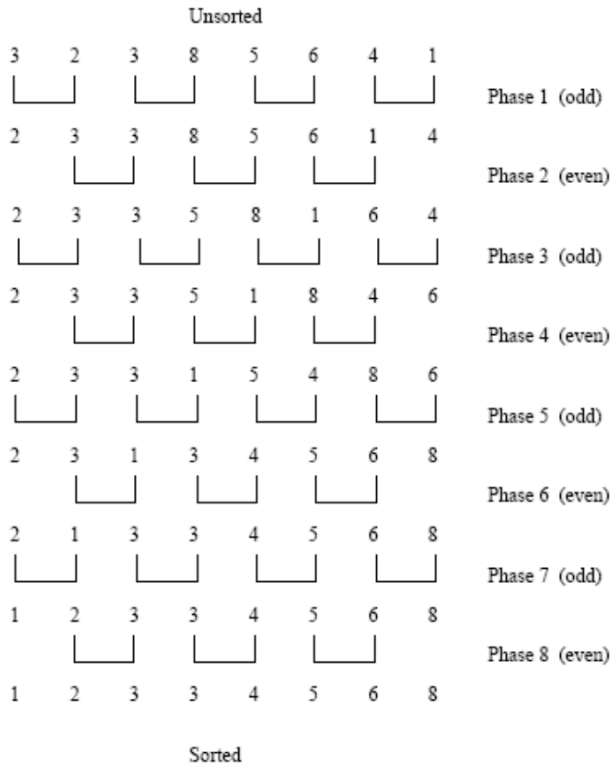
MESH-BASED OBLIVIOUS SORTING

Even-odd transposition sorting is a direct sorting method on 1-D meshes is also called parallel bubble-sort. It takes precisely n steps to sort n numbers on a 1D-Mesh array with n nodes. Note that this number of steps is within 1 step of optimal since the diameter of the network in n-1. The work of the algorithm is thus O(n^2).

The idea of the algorithm is similar in some sense to bubble sort by alternating Compare and Exchange CE-operations between odd-even and even-odd adjacent pairs of nodes.

```
Let x1,.., xn be a sequence of numbers to be sorted
in ascending order.

for j=1…[n/2] do_sequentially
    begin
       for i=1,3,..,2[n/2]-1  do_in_parallel
            if xi > xi+1 then CE(xi,xi+1);
       for i=2,4,..,2[(n-1)/2] do_in_parallel
            if xi > xi+1 then CE(xi,xi+1);
    end
```



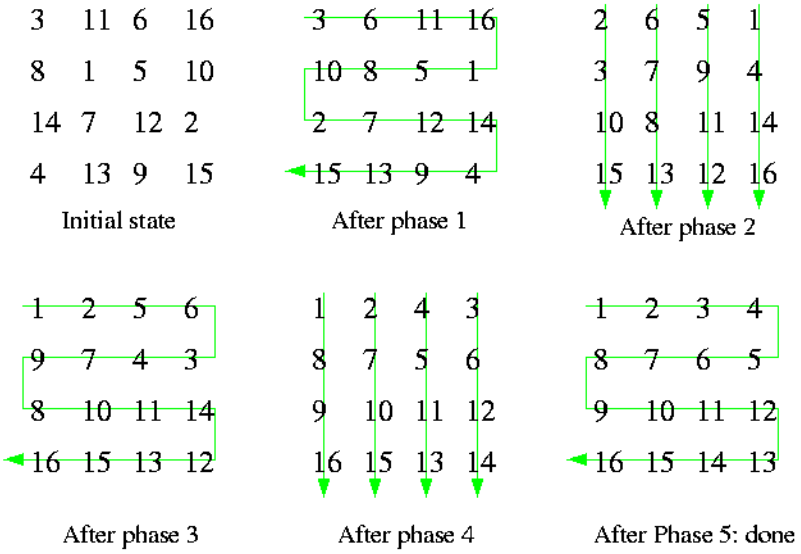
SHEARSORT ON 2-D MESHES

The simplest 2-D mesh sorting algorithm is called ShearSort and it is based on the even-odd transposition sorting discussed above. The algorithm works for any 2-D mesh as follows. Consider a mesh M(n,m), with n rows and m columns. The algorithm proceeds in several alternating phases with one phase sorting all the rows in Snake-like

order and the other phase sorting all columns in increasing order.

```
for i=1,..., 2log n+1 do_sequentially
  if (i is odd):
    SORT_ALL_ROWS_SNAKELIKE in parallel (Phase 1,3,5)
  else:
    SORT_ALL_COLUMNS in parallel (Phase 2,4)
```

Here is an example run of Shearsort on a 4x4 mesh with 5 total phases.



The general result is that ShearSort requires [log n]+1 row phases and [log n] column phases to sort n^2 numbers on M(n,n) in a snakelike order. One row or one column phase takes O(n) computational and communication steps. The proof of this result will follow from the application of the 0-1 Sorting Principle. Assume you are given any zero-one n×n matrix. It may contain rows of three distinct kinds.

1) all-one rows containing only 1's,

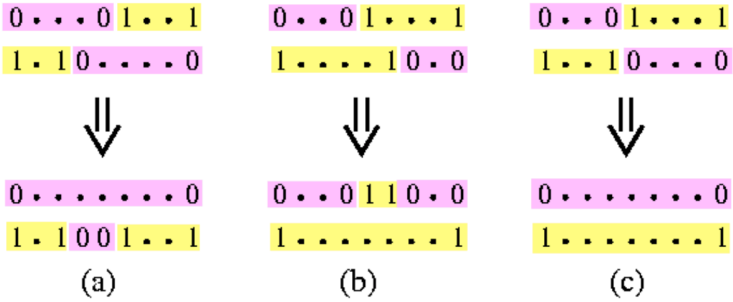
2) all-zero rows containing only 0's,

3) so called dirty rows containing both 0's and 1's.

Initially, the input matrix can contain n dirty rows in the worst case. The final matrix sorted in snakelike order contains at most one dirty row.

It follows that each pair of phases, one row and one column phase, will reduce the number of dirty rows by at least one half in total number. To see this, we can do a simple case analysis. Consider all dirty rows after one row phase. One half of them are sorted rightward, whereas the other half are in reverse order. If we consider pairs of leftward and rightward rows, there can exist pairs of

at most three different types, as shown in the following figure.



CAPTION: Three kinds of pairs of dirty rows. After applying one column phase, one dirty row disappears in cases (a) and (b), and both dirty rows disappear in case (c).

Hence, after 2log n phases, at most one dirty row remains and one more row sort will complete the sorting.

Unfortunately, ShearSort is not an optimal algorithm. Improvements based on decreasing number of dirty rows help only a little. There is a more complex, but asymptotically optimal 2-D mesh sorting algorithm which runs in linear 3n number of steps.

BITONIC MERGESORT NETWORK

A bitonic sorting network sorts n elements in Θ(log^2 n) depth (time) and Θ(n log n) work (size). A bitonic sequence has two tones - increasing and decreasing, or vice versa. Any cyclic rotation of such a sequence is also considered bitonic.

⟨1,2,4,7,6,0⟩ is a bitonic sequence, because it first increases and then decreases.

⟨8,9,2,1,0,4⟩ is another bitonic sequence, because it is a cyclic shift of the sequence ⟨0,4,8,9,2,1⟩ .

Original sequence	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
1st Split	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
2nd Split	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3rd Split	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
4th Split	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

The key kernel of a biotic merge sort network is the rearrangement of a bitonic sequence into a sorted sequence.

To this end, let s = ⟨a0,a1,…, a(n-1)⟩ be a bitonic sequence such that

a0 ≤ a1 ≤ ⋯ ≤ a(n/2-1) and

a(n/2) ≥ a(n/2+1) ≥ ⋯ ≥ a(n-1).

Consider the following min and max subsequences of s:

s1 = ⟨
min{a0,a(n/2)},
min{a1,a(n/2+1)},…,
min{a(n/2-1),a(n-1)}⟩

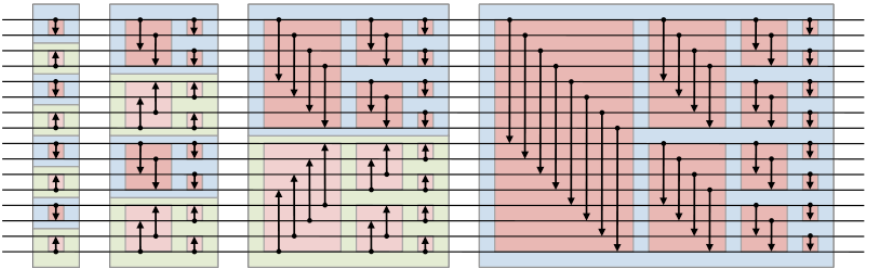
s2 = ⟨
max{a0,a(n/2)},
max{a1,a(n/2+1)},…,
max{a(n/2-1),a(n-1)}⟩

Note that s1 and s2 are both bitonic and each element of s1 is less than every element in s2.

We can apply the procedure recursively on s1 and s2 to get the sorted sequence.

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

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



```
def bitonic_compare_swap(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]
```

The following figure shows the bionic merging a 16-element bitonic sequence through a series of log16 =4 bitonic splits.

We can easily build a sorting network to implement this bitonic merge algorithm, and such a network is called a bitonic merging network. The network contains log n columns. Each column contains n/2 comparators and performs one step of the bitonic merge.

We denote a bitonic merging network with n inputs by the notation ⊕BM[n]. Flipping these comparators, that is by replacing each of the plus ⊕ comparators by negative Θ comparators will result in a decreasing output sequence. Such a network is denoted by ΘBM[n].

To show the correctness of this sorting network we can again apply Knuth's 0/1 Sorting Principle in the following argument.

We assume that the length of the sequence is n = 2^k a power of 2. By induction, assume that the network sorts n/2 = 2^(k-1) length 0-1 sequences. The result is of n-sequence with two tones is always of the form 0000…01111…1100000…0

Now perform n/2 Compare/Swap operations. We claim that either the first half is all 0's and the second half is bitonic, or the first half is bitonic and the second half is all 0s. Therefore, it is sufficient to apply the same construction recursively on the two halves. From this the correctness of the sorting network follows by the induction principle!

Summarizing the results above, we have provided the design of a sorting network that is called bionic merge sort network and is specified recursively. We also emphasize that the design of this network yields an oblivious algorithm which easily maps to GPUs. Recall that an oblivious sorting algorithm is defined as one that will make the same sequence of compare-exchanges regardless of the input distribution.

The figure above is an illustration of a bionic sorting network for n=16 inputs. Notice how the left half of the network is a sub-network that produces a bitonic sequence with the first half increasing and second half decreasing.

RADIX SORTING

Radix sorting is a high performance non-comparison sorting algorithm that can be easily adapted from a sequential algorithm to run in parallel.

Radix sort generally only works on numbers (ints or floats) and relies on the numerical representation of number. For example, we will focus on k-bit binary numbers. The algorithms starts with a focus on the lowest order bit and moves one bit at time towards the most significant bit. At each stage the set of numbers is split into 2 sets—a lower set representing those numbers with 0 bit and upper set representing numbers with a 1 bit in focus. We can easily generalize radix sort to other numerical representations by agglomerating several bits, say b at a time, thus producing a sequencing of 2^b sets in each round.

By using a technique similar to compacting covered in Chapter 6, we can scan to determine a location for each element in each set.

The parallel implementation of radix sort uses the basic idea to construct a running frequency on each pass of how many of each "digit" there are. With scan this we know position and where to send the number. For example, we can use

1) Scan to determine number of occurrences of each digit

2) Determine relative offset of each digits. For example [0 0 1 1 0 0 1] -> [0 1 0 1 2 3 2]

3) Determine the final scatter location for each element and write it there.

The overall work complexity of radix sorting is O(n k b), and can be determined as follows. There are k rounds in the algorithm and each round does O(n) constant time operations. All these rounds are independent. Now we only need to compact a pair of arrays (or 2^b arrays in general for b-bit digits). Hence, the step complexity is O(k log n), and can be determined by k rounds where each round has at most log n steps to complete the compaction operation.