Massively Parallel Algorithms
Parallel Sorting

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de
Sorting using Spaghetti in $O(1)$ (?)

- Is $O(n)$ really the lower bound for sorting?
- Consider the following thought experiment:
  - B. For each number $x$ in the list, cut a spaghetto to length $x \rightarrow$ list = bundle of spaghettis & unary repr.
  - C. Hold the spaghettis loosely in your hand and tap them on the kitchen table $\rightarrow$ takes $O(1)$!
  - D. Lower your other hand from above until it meets with a spaghetto — this one is clearly the longest
  - E. Remove this spaghetto and insert it into the front of the output list
  - F. Repeat
- If we could use this mechanical computer, then sorting would be $O(1)$
Difficulties With Parallel Implementation of Standard Sequential Algorithms

- **Insertion sort:**
  - Considers only one element at a time

- **Quicksort:**
  - Yes, some parallelism at lower levels of the recursion tree
  - But, would need *median* as a pivot element → hard to find
  - Otherwise, random pivot element causes varying sub-array sizes

- **Heapsort:**
  - Only one element at a time
  - Heap (= recursive data structure) is difficult on mass.-parallel architecture

- **Radix sort:**
  - Yes, we've seen that already, works well
  - But, can handle only fixed-length numbers
Assumptions

- In this chapter, we will always assume that $n = 2^k$
- Elements can have any type, for which there is a comparison operator
Informal definition of comparator networks:
- Consist of a bundle of "wires"
- Each wire $i$ carries a data element $D_i$ (e.g., float) from left to right
- Two wires can be connected vertically by a comparator
  - If $D_i > D_j \land i < j$ (i.e., wrong order), then $D_i$ and $D_j$ are swapped by the comparator before they move on along the wires

Observation: every comparator network is data independent, i.e., the arrangement of comparators and the running time are always the same!

Goal: find a "small" comparator network that performs sorting for any input → sorting network
Example

One stage / step
The 0-1 Principle

- **Definition** (*monotone function*):
  Let $A, B$ be two sets with a total ordering relation, and let $f : A \to B$ be a mapping. $f$ is called **monotone** iff
  $$\forall a_1, a_2 \in A : a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2)$$

- **Lemma**:
  Let $f : A \to B$ be monotone. Then, $f$ and $\text{min}$ commute, i.e.
  $$\forall a_1, a_2 \in A : f(\text{min}(a_1, a_2)) = \text{min}(f(a_1), f(a_2))$$
  Analogously for the $\text{max}$.

- **Proof**:
  Case 1: $a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2)$
  $$\min(a_1, a_2) = a_1, \quad \text{min}(f(a_1), f(a_2)) = f(a_1)$$
  $$f(\text{min}(a_1, a_2)) = f(a_1) = \text{min}(f(a_1), f(a_2))$$
  Case 2: $a_2 < a_1 \to$ analog
- Extension of \( f : A \rightarrow B \) to sequences over \( A \) and \( B \), resp.:

\[
f(a_0, \ldots, a_n) = f(a_0), \ldots, f(a_n)
\]

- Lemma:
  Let \( f \) be a monotone mapping and \( \mathcal{N} \) a comparator network. Then \( \mathcal{N} \) and \( f \) commute, i.e.

\[
\forall n \forall a_0, \ldots, a_n : \mathcal{N}(f(a)) = f(\mathcal{N}(a))
\]
Proof:

Let $a = (a_0, \ldots, a_n)$ be a sequence

Notation: we write a comparator connecting wire $i$ and $j$ like so:

$$[i : j](a)$$

Now the following is true:

$$[i : j](f(a)) = [i : j](f(a_0), \ldots, f(a_n))$$

$$= (f(a_0), \ldots, \min_{i} f(a_i), f(a_j), \ldots, \max_{j} f(a_i), f(a_j), \ldots, f(a_n))$$

$$= (f(a_0), \ldots, f(\min(a_i, a_j)), \ldots, f(\max(a_i, a_j)), \ldots, f(a_n))$$

$$= f(a_0, \ldots, \min(a_i, a_j), \ldots, \max(a_i, a_j), \ldots, a_n)$$

$$= f([i : j](a))$$
Theorem (the 0-1 principle):

Let \( \mathcal{N} \) be a comparator network.

Now, if \( \mathcal{N} \) sorts every sequence of 0's and 1's, then it also sorts every sequence of arbitrary elements!
- Proof (by contradiction):
  - Assumption: $\mathcal{N}$ sorts all 0-1 sequences, but does not sort sequence $a$
  - Then $\mathcal{N}(a) = b$ is not sorted correctly, i.e. $\exists k : b_k > b_{k+1}$
  - Define $f : A \rightarrow \{0,1\}$ as follows:
    \[
    f(c) = \begin{cases} 
    0, & c < b_k \\
    1, & c \geq b_k 
    \end{cases}
    \]
  - Now, the following holds:
    \[
    f(b) = f(\mathcal{N}(a)) = \mathcal{N}(f(a)) = \mathcal{N}(a')
    \]
    where $a'$ is a 0-1 sequence.
  - But: $f(b)$ is not sorted, because $f(b_k) = 1$ and $f(b_{k+1}) = 0$
  - Therefore, $\mathcal{N}(a')$ is not sorted as well, in other words, we have constructed a 0-1 sequence that is not sorted correctly by $\mathcal{N}$.
In the following, we'll always assume that the length $n$ of a sequence $a_0,\ldots,a_{n-1}$ is a power of 2, i.e., $n = 2^k$.

First of all, we define the sub-routine "odd-even merge":

```plaintext
oem( a_0,\ldots,a_{n-1} ):
precondition: a_0,\ldots,a_{n/2}-1 and a_{n/2},\ldots,a_{n-1} are both sorted
postcondition: a_0,\ldots,a_{n-1} is sorted
if n = 2:
    compare [a_0:a_1]  \hspace{1cm} (1)
if n > 2:
    \hat{a} \leftarrow a_0,a_2,\ldots,a_{n-2} \hspace{1cm} // = even sub-sequence
    \hat{a} \leftarrow a_1,a_3,\ldots,a_{n-1} \hspace{1cm} // = odd sub-sequence
    \bar{b} \leftarrow \text{oem}( \hat{a} )
    \hat{b} \leftarrow \text{oem}( \hat{a} )
    \text{copy } \bar{b} \rightarrow a_0,a_2,\ldots,a_{n-2}
    \text{copy } \hat{b} \rightarrow a_1,a_3,\ldots,a_{n-1}
    \text{for } i \in \{1,3,5,\ldots,n-3\}
        \text{compare } [a_i : a_{i+1}] \hspace{1cm} (2)
```


- **Proof of correctness:**
  - By induction and the 0-1-principle
  - Base case: $n = 2$
  - Induction step: $n = 2^k$, $k > 1$
  - Consider a 0-1-sequence $a_0,...,a_{n-1}$
  - Write it in two columns
  - Visualize $0 = \text{white}, 1 = \text{grey}$
  - Obviously: both $\bar{a}$ and $\hat{a}$ consist of two sorted halves $\rightarrow$ precondition of $\text{oem}$ is met
  - After line (2) we have this situation (the odd sub-sequence can have at most two 1's more than the even sub-sequence)
In loop (3), these comparisons are made, and there can be only 3 cases:

Afterwards, one of these two situations has been established:

- Result: the output sequence is sorted
- Conclusion: every 0-1-sequence (meeting the preconditions) is sorted correctly

Running time (sequ.): \( T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2} - 1 \in O(n \log n) \)
The complete general sorting-algorithm:

```python
oemSort(a_0, ..., a_{n-1}):
    if n = 1:
        return
    a_0, ..., a_{n/2-1} ← oemSort(a_0, ..., a_{n/2-1})
    a_{n/2}, ..., a_{n-1} ← oemSort(a_{n/2}, ..., a_{n-1})
    oem(a_0, ..., a_{n-1})
```

Running time (sequ.): \( T(n) \in O(n \log^2 n) \)
Mapping the Recursion on a Massively-Parallel Architecture

- Load data onto the GPU (global memory)
- The CPU executes the following controlling program:

```python
def oemSort(n):
    if n = 1 → return oemSort( n/2 )
oem( n, 1 )
```

```python
def oem(n, stride):
    if n = 2:
        launch oemBaseCaseKernel(stride)
        // launches n parallel threads
    else:
        oem( n/2, stride*2 )
        launch oemRecursionKernel(stride)
```

- With the stride parameter, we can achieve sorting "in situ"
The kernel for line (3) of the original function oem():

```python
@cuda.jit
def oemRecursionKernel(stride):
    tid = cuda.threadIdx.x
    n = cuda.gridDim.x
    for tid in range(tid, n, stride):
        a_i = SortData[tid]
        a_j = SortData[tid + stride]
        if tid/stride is even:
            output max(a_i, a_j)
        else:
            output min(a_i, a_j)
```

As usual, $tid = \text{thread ID} = 0, ..., n-1$
Kernel for line (1) of the function oem():

```plaintext
doemBaseCaseKernel ( stride ):
  i = tid                      // tid = thread ID
  if tid/stride is even:       // are we on even/odd side?
    j = i + stride
  else:
    j = i - stride
  a0 ← SortData[i]            // SortData = global array
  a1 ← SortData[j]
  if on even side:
    SortData[i] = min(a0,a1)   // write output back
  else:
    SortData[i] = max(a0,a1)
```

Reminder: this kernel is executed in parallel for each index \( tid = 0, \ldots, n-1 \) in a stream.
- Depth complexity:
  \[ \frac{1}{2} \log^2 n + \frac{1}{2} \log n \]

- E.g., for \(2^{20}\) elements this are 210 passes
Bitonic Sorting

- Definition "bitonic sequence":
  A sequence of numbers $a_0, \ldots, a_{n-1}$ is bitonic \iff there is an index $i$ such that
  - $a_0, \ldots, a_i$ is monotonically increasing, \textit{and}
  - $a_{i+1}, \ldots, a_{n-1}$ is monotonically decreasing;
  OR
  if there is a cyclic shift of this sequence such that this is the case.

- Because of the latter "OR", we understand \textit{all index arithmetic} in the following \textit{modulo} $n$, and/or we assume in the following that the sequence(s) have been cyclically shifted as described above.
Examples of bitonic sequences:

- 0 2 4 8 10 9 7 5 3 ; 2 4 8 10 9 7 5 3 0 ; 4 8 10 9 7 5 3 0 2 ; ...
- 10 12 14 20 95 90 60 40
  35 23 18 0 3 5 8 9
- 1 2 3 4 5
- []
- 00000111110000 ;
  11111000000111111 ;
  1111100000 ; 000011111

These sequences are **NOT** bitonic sequences:

- 1 2 3 1 2 3
- 1 2 3 0 1 2
- **Visual representation of bitonic sequences:**

  ![Visual representation of bitonic sequences](image1)

- **Because of the "modulo" index arithmetic, we can also visualize them on a circle or cylinder:**

  - Clearly, bitonic sequences have exactly two inflection points.
Properties of Bitonic Sequences

- Any sub-sequence of a bitonic sequence is a bitonic sequence
  - More precisely, assume $a_0, ..., a_{n-1}$ is bitonic and we have indices
    $0 \leq i_1 \leq i_2 \leq ... \leq i_m < n$
  - Then, $a_{i_0}, a_{i_1}, ..., a_{i_m}$ is bitonic, too

- If $a_0, ..., a_{n-1}$ is bitonic, then $a_{n-1}, ..., a_0$ is bitonic, too
- (If we mirror a bitonic sequence "upside down", then the new sequence is bitonic, too)

- A bitonic sequence has exactly one local(!) minimum and one local maximum
Some Notions and Definitions

- More precise graphical notation of a comparator:

- Definition rotation operator:
  Let \( a = (a_0, \ldots, a_{n-1}) \), and \( j \in [1, n-1] \).
  We define the rotation operator \( R_j \) acting on \( a \) as
  \[
  R_j a = (a_j, a_{j+1}, \ldots, a_{j+n-1})
  \]
Definition \textbf{L / U operator}:

\[ L a = ( \min(a_0, a_{\frac{n}{2}}), \ldots, \min(a_{\frac{n}{2}-1}, a_{n-1}) ) \]
\[ U a = ( \max(a_0, a_{\frac{n}{2}}), \ldots, \max(a_{\frac{n}{2}-1}, a_{n-1}) ) \]

Lemma:

The L/U operators are \textit{rotation invariant}, i.e.

\[ L a = R_{-j} L R_j a, \quad \text{and} \quad U a = R_{-j} U R_j a. \]

(Remember that indices are always meant mod \( n \))

Proof:

- We need to show that \( R_j L a = L R_j a \)
- This is trivially the case:
  \[ LR_j a = ( \min(a_j, a_{j+\frac{n}{2}}), \ldots, \min(a_{\frac{n}{2}-1}, a_{n-1}), \ldots, \min(a_{j-1}, a_{j-1+\frac{n}{2}}) ) = \ldots \]
Definition half-cleaner:
A network that takes \( a \) as input and outputs \( (L_a, U_a) \) is called a half-cleaner.

The network that realizes a half-cleaner:

Because of the rotation invariance, we can depict a half-cleaner on a circle:
- It always produces \( L_a \) and \( U_a \), no matter how \( a \) is rotated around the circle!
Theorem 1:
Given a bitonic input sequence \( a \), the output of a half-cleaner has the following properties:

1. \( L_a \) and \( U_a \) are bitonic, too;
2. \( \max\{L_a\} \leq \min\{U_a\} \)
Proof

- The half-cleaner does the following:
  1. Shift (only conceptually) the right half of \( a \) over to the left
  2. Take the point-wise min/max → \( L_a, U_a \)
  3. Shift \( U_a \) back to the right
- Because \( a \) is bitonic, there can be only one *cross-over point*
- By construction, both \( L_a \) and \( U_a \) must have length \( n/2 \)
- Property 1 follows from the sub-sequence property
The Bitonic Merger

- The half-cleaner is the basic (and only) building block for the bitonic sorting network!
- The recursive definition of a bitonic merger $\text{BM}^\uparrow(n)$:
  - Input: bitonic sequence of length $n$
  - Output: sorted sequence in ascending order
- Analogously, we can define $\text{BM}^\downarrow(n)$
Visualization of the Workings of a Bitonic Merger
Mapping to Massively Parallel Architecture

- We have \( n = 2^k \) many "lanes" = threads
- At each step, each thread needs to figure out its partner for compare/exchange
- This can be done by considering the ID of each process (in binary):
  - At step \( j, j = 1, \ldots, k \) : partner ID = ID obtained by reversing bit \( (k-j) \) of own ID
- Example:

\[
\begin{array}{cccccccc}
000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
| & ^ & ^ & ^ & \\
|______| & | & |
| k-3 & | & |
|___________| & | \\
| k-2 & | \\
|_____________________| \\
k-1
\]
The Bitonic Sorter

- The recursive definition of a bitonic sorter $BS^\uparrow(n)$:

\[ BS^\uparrow(n) = \begin{cases} a_0, & \text{if } n = 1 \\ BS^\uparrow(n/2) \cdot BM^\uparrow(n/2), & \text{otherwise} \end{cases} \]