



中山大學
SUN YAT-SEN UNIVERSITY



多核程序设计与实践

应用举例：并行排序

陶钧

taoj23@mail.sysu.edu.cn

中山大学 数据科学与计算机学院
国家超级计算广州中心

- 排序算法定义及相关性质
- 插入排序、冒泡排序与相应排序网络
- 归并排序与双调排序
- 其他适合并行的排序算法



● 问题定义

- 给定一个由n个元素组成的序列
 - 可表示为数组或链表
 - 排序算法可能依赖于具体数据结构
 - 大多数算法都能使用数组
 - 部分算法不适合链表：如，快速排序，希尔排序（需要index）
 - $A=[2, 5, 4, 3, 6, 1, 3]$
- 将其中元素按一定顺序排列
 - 从序列中任选一对元素都是有序的，则序列为已排序的
 - $\text{sorted}(A)=[1, 2, 3, 3, 4, 5, 6]$

● 稳定性

– 稳定的排序算法中具有相同值的元素相对顺序**不会发生变化**

- 稳定: $\text{sorted}(\text{A})=[1, 2, \textcolor{red}{3}, \textcolor{blue}{3}, 4, 5, 6]$
- 不稳定: $\text{sorted}(\text{A})=[1, 2, \textcolor{blue}{3}, \textcolor{red}{3}, 4, 5, 6]$

– 稳定性的意义

- 对复杂对象的多个属性分别进行处理时，需要在排序中保持原有顺序
- 不稳定排序算法可以通过对元素值进行处理而符合稳定性要求

– 稳定的排序

- 冒泡排序、插入排序、基数排序、桶排序等

– 不稳定的排序

- 快速排序、希尔排序、选择排序等

● 数据驱动

- 排序算法所采取的每一步都依赖于此前进行的步骤
- 高度串行

● 数据无关

- 排序算法采取固定的步骤，不根据数据改变而改变
- 适用于并行算法
- 通常可以表示为排序网络
- 其他数据无关的排序举例：Bogo排序（不实用但高度并行化）
 - 随机生成元素排列，检查是否为有序
 - 不确定机在 $O(n)$ 时间内完成
 - 变种：Bozo排序

- 排序算法定义及相关性质
- 插入排序、冒泡排序与相应排序网络
- 归并排序与双调排序
- 其他适合并行的排序算法



● 插入排序

- 每次将一个新的元素插入到之前已排序的序列中
 - 如，对[2, 5, 4, 3, 6, 1]进行插入排序：
 - [2]→[2, 5]→[2, 4, 5]→[2, 3, 4, 5]→[2, 3, 4, 5, 6]→[1, 2, 3, 4, 5, 6]
 - 外层循环不变量：i次循环后，前i个元素为已排序
- 时间复杂度 $O(n^2)$ ，空间复杂度 $O(1)$ ，是否数据无关？

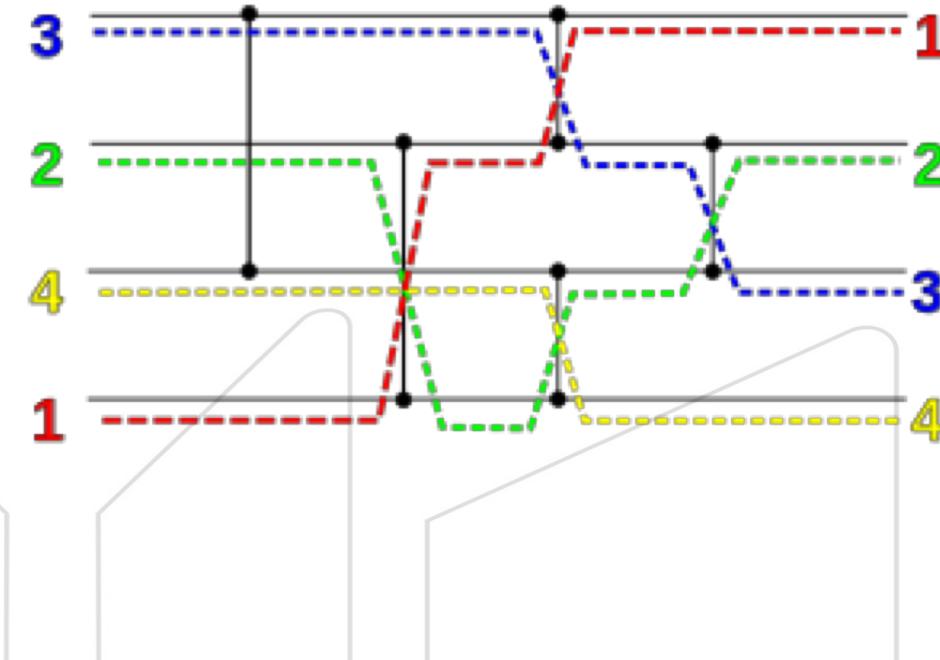
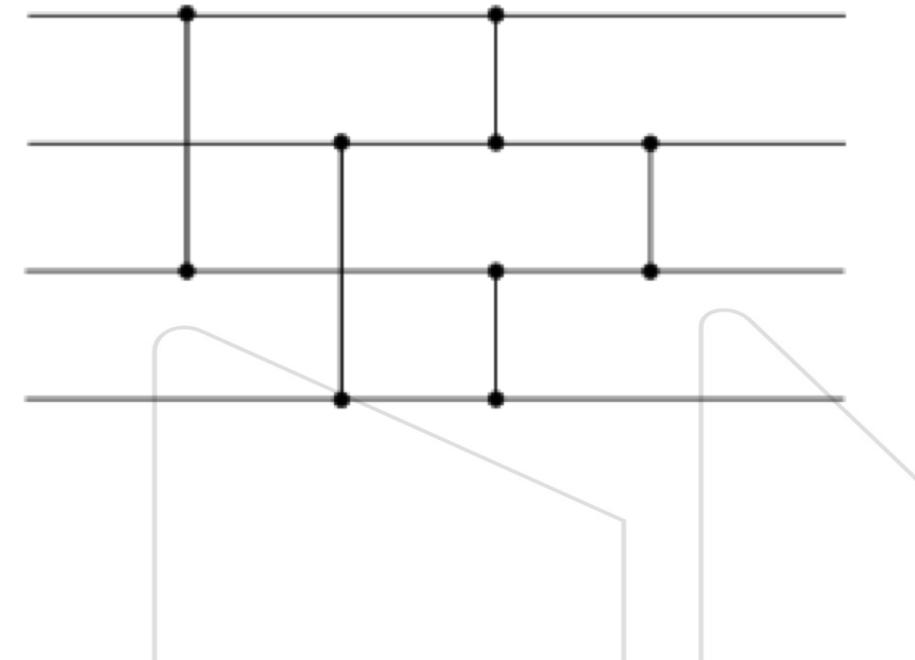
● 冒泡排序

- 每次检查相邻的两个元素是否有序，如无序则交换
 - 如，对[2, 5, 4, 3, 6, 1]进行插入排序（前两次外层循环）：
 - [2, 5, 4, 3, 6, 1]→[2, 4, 5, 3, 6, 1]→[2, 4, 3, 5, 6, 1]→[2, 4, 3, 5, 6, 1]→[2, 4, 3, 5, 1, 6]
 - [2, 4, 3, 5, 1, 6]→[2, 3, 4, 5, 1, 6]→[2, 3, 4, 5, 1, 6]→[2, 3, 4, 1, 5, 6]...
 - 外层循环不变量：i次循环后，后i个元素为已排序
- 时间复杂度 $O(n^2)$ ，空间复杂度 $O(1)$ ，是否数据无关？

● 排序网络

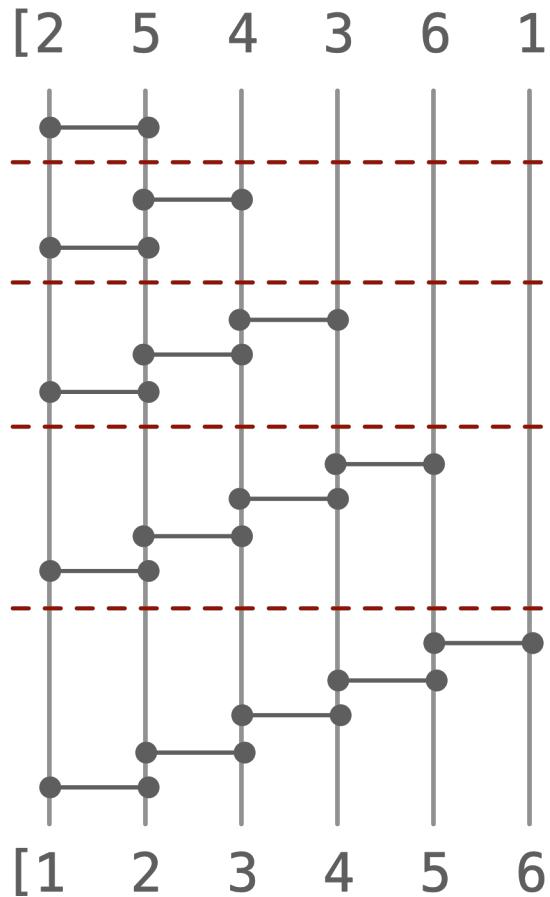
- 由一系列比较器组成的网络，对元素进行比较及交换，从而达到有序状态

- 具有固定的执行过程
- 现有数据无关排序算法多可表示为排序网络（如插入排序、冒泡排序）
- 排序网络举例：



● 插入排序的排序网络

– 每层中进行比较的序列长度加1



[2 5 4 3 6 1]

[2 4 5 3 6 1] → [2 4 5 3 6 1]

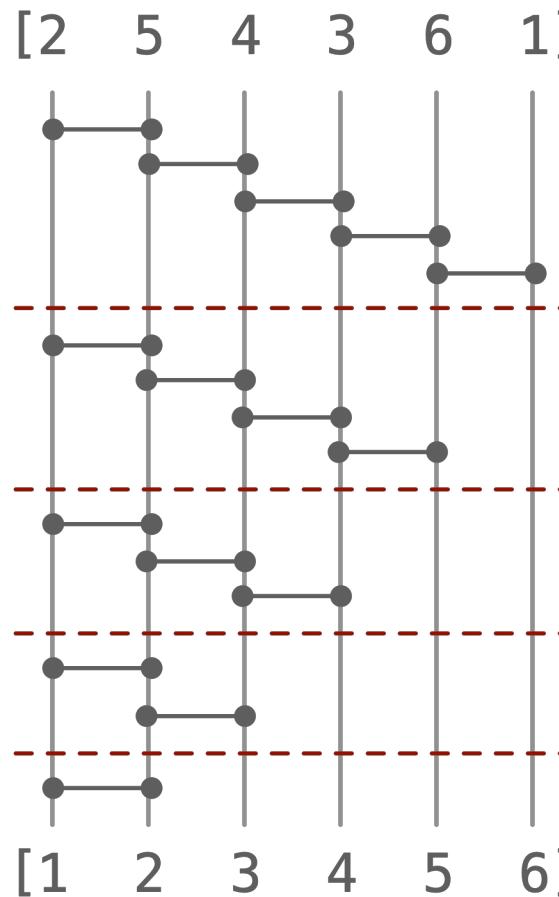
[2 4 3 5 6 1] → [2 3 4 5 6 1] → [2 3 4 5 6 1]

[2 3 4 5 6 1] → [2 3 4 5 6 1] → [2 3 4 5 6 1] → [2 3 4 5 6 1]

[2 3 4 5 1 6] → [2 3 4 1 5 6] → [2 3 1 4 5 6] → [2 1 3 4 5 6] → [1 2 3 4 5 6]

● 冒泡排序的排序网络

– 每层中进行比较的序列长度减1

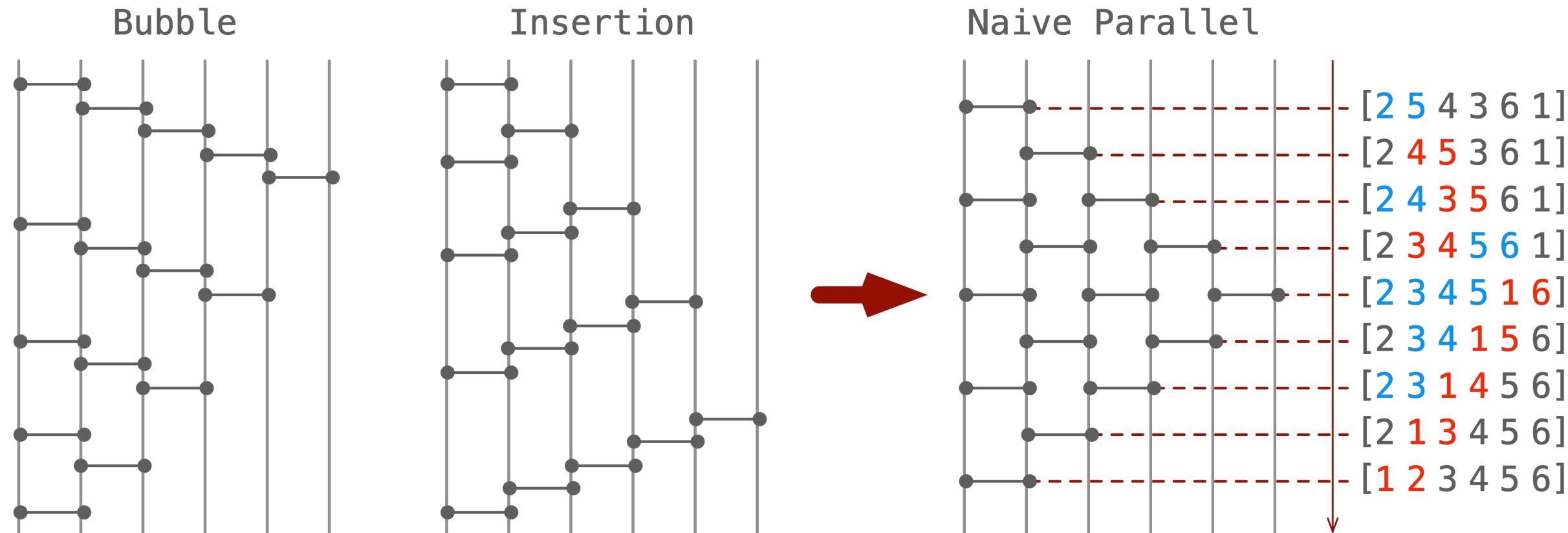


[2 5 4 3 6 1] → [2 4 5 3 6 1] → [2 4 3 5 6 1] → [2 4 3 5 6 1] → [2 4 3 5 1 6]
[2 4 3 5 1 6] → [2 3 4 5 1 6] → [2 3 4 5 1 6] → [2 3 4 1 5 6]
[2 3 4 1 5 6] → [2 3 4 1 5 6] → [2 3 1 4 5 6]
[2 3 1 4 5 6] → [2 1 3 4 5 6]
[1 2 3 4 5 6]

● 并行排序网络

– 插入排序与冒泡排序的网络高度相似！

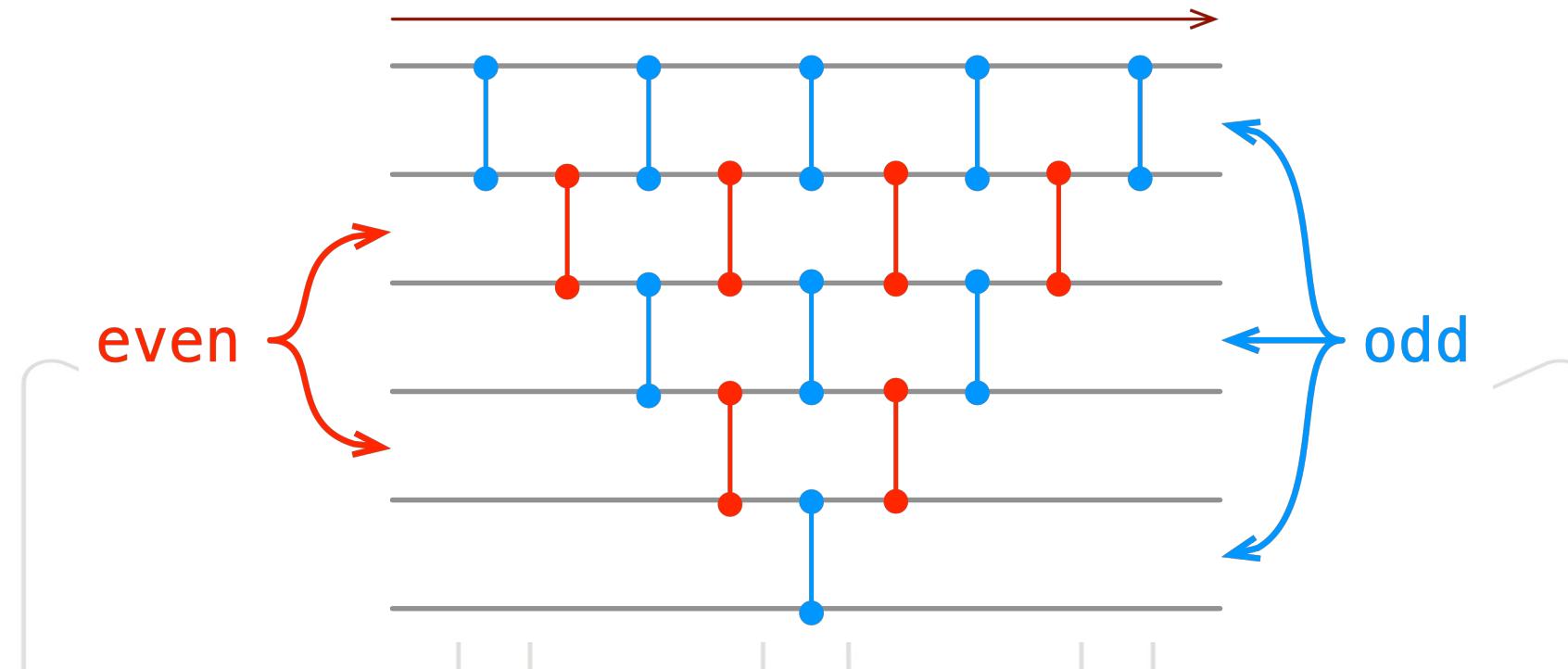
- 在不改变原有执行顺序的情况下，通过同时执行作用于操作可并行
- 总运算量不变： $n(n - 1)/2$ 次比较
- 时间复杂度： $O(n^2)$



- 使用排序网络将插入排序与冒泡排序并行化

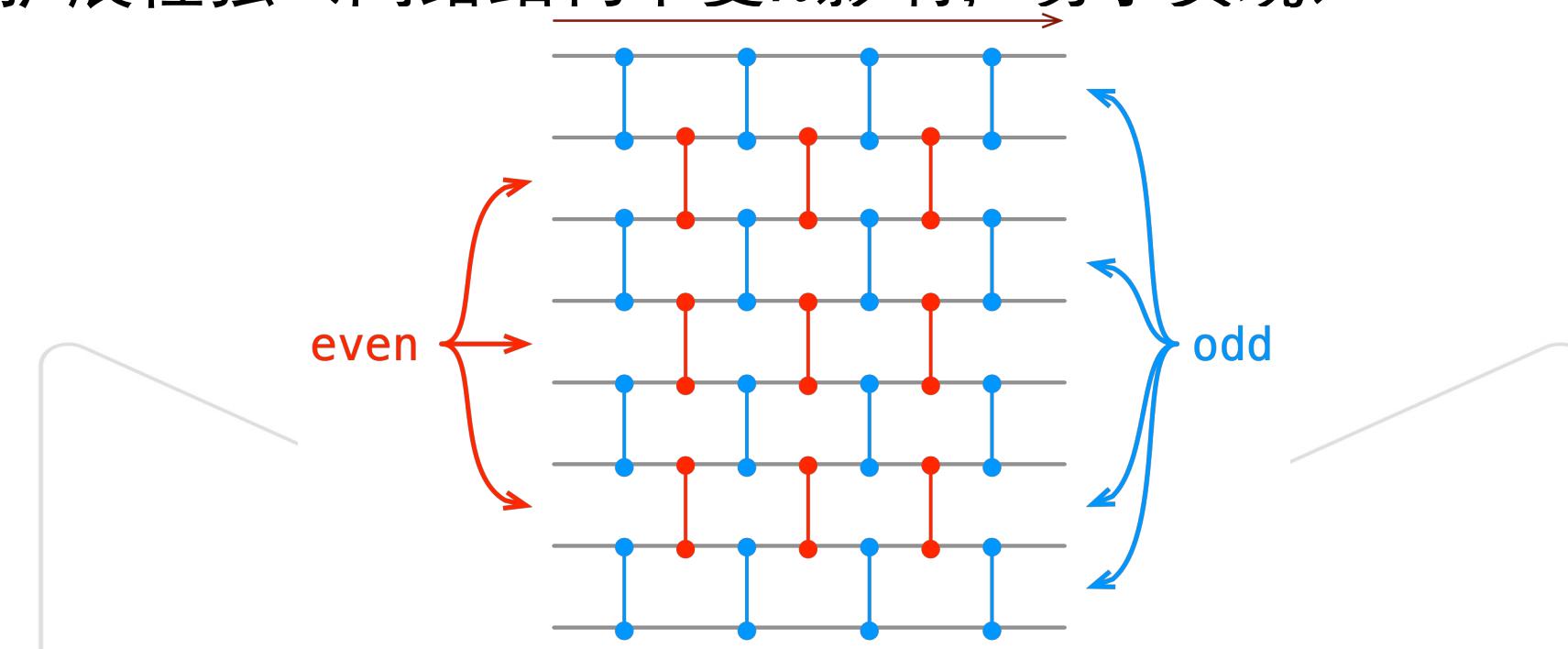
- 插入排序与冒泡排序的网络高度相似！

- 在不改变原有执行顺序的情况下，通过同时执行作用于操作可并行
 - 总运算量不变： $n(n - 1)/2$ 次比较
 - 时间复杂度： $O(n^2)$
 - 共 $2n - 3$ 层，其中第*i*层处理*i*/2对元素（全为奇数对，或全为偶数对）
 - 假设运算核心足够多的情况下，其并行所需时间 $2n - 3$



● 奇偶移项排序网络 (odd-even transposition sort)

- 由插入排序及冒泡排序改进的排序网络深度为 $2n - 3$ 层，在起始及结束阶段并没有充分利用计算资源
- 改进：将三角形网络改为正方形网络
 - 深度从 $2n - 3$ 降至 $n - 1$ （为什么 $n - 1$ 层已经足够？）
- 优点：空间局部性强（所有比较都作用于相邻元素上）
可扩展性强（网络结构不受 n 影响，易于实现）



● 奇偶移项排序网络 (odd-even transposition sort)

- 网络深度为 $n - 1$ (为什么 $n - 1$ 层已经足够?)
 - 直观地看: n 层已经足够将任意元素移动到序列中的任意位置
 - 从序列一端移动到另一端需要 $n - 1$ 次操作
 - 从运算量看: $n - 1$ 层中总计进行 $O(n^2)$ 次比较运算

0-1-principle

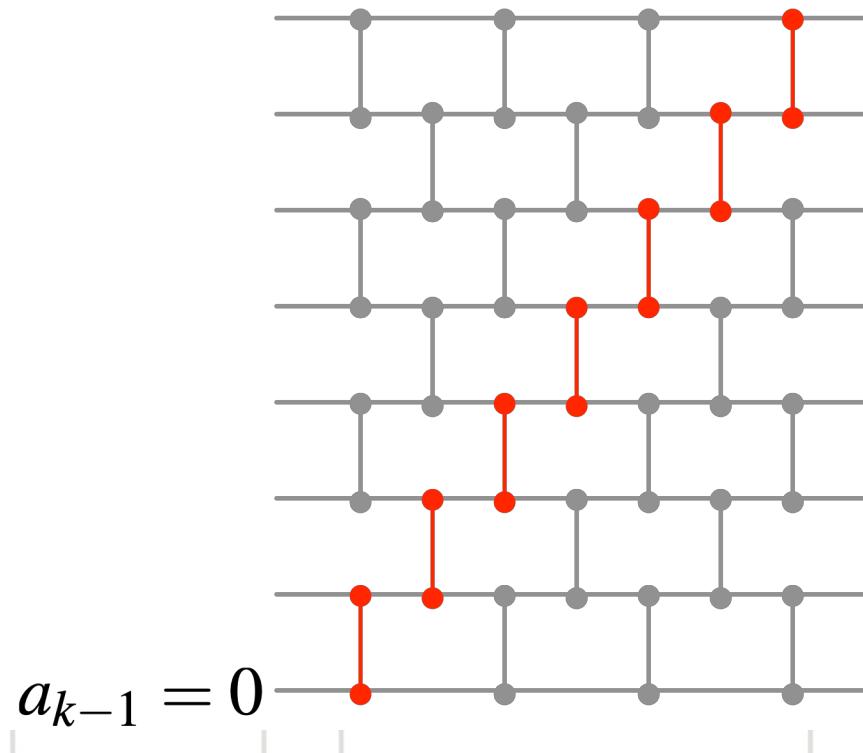
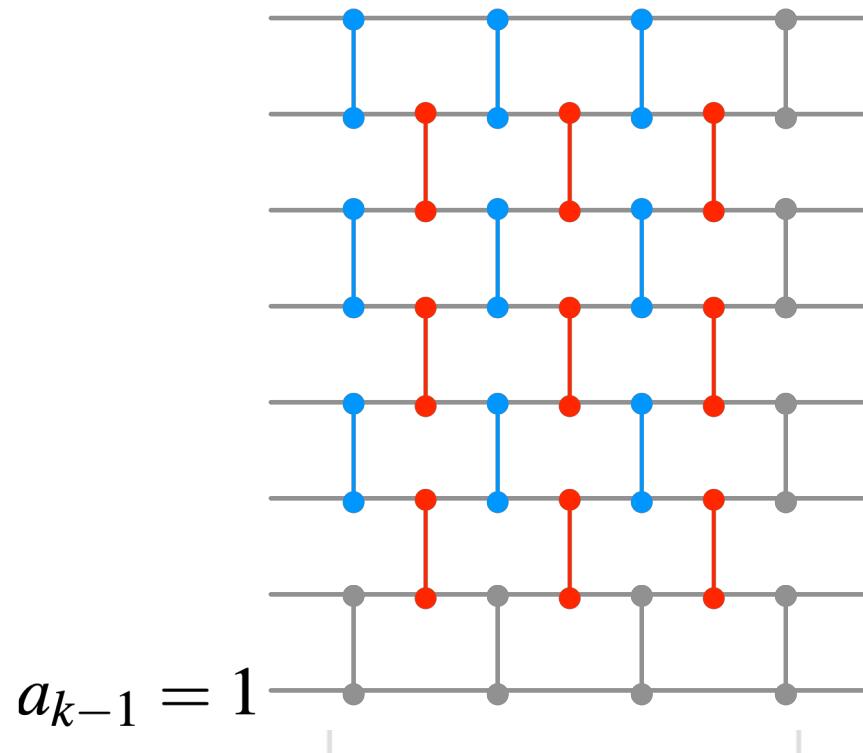
由Donald Ervin Knuth提出: 一个比较网络如果能对长度为 2^n 的任意0-1序列进行排序, 则该比较网络为排序网络 (能对任意数字组成的序列进行正确的排序)。



● 奇偶移项排序网络 (odd-even transposition sort)

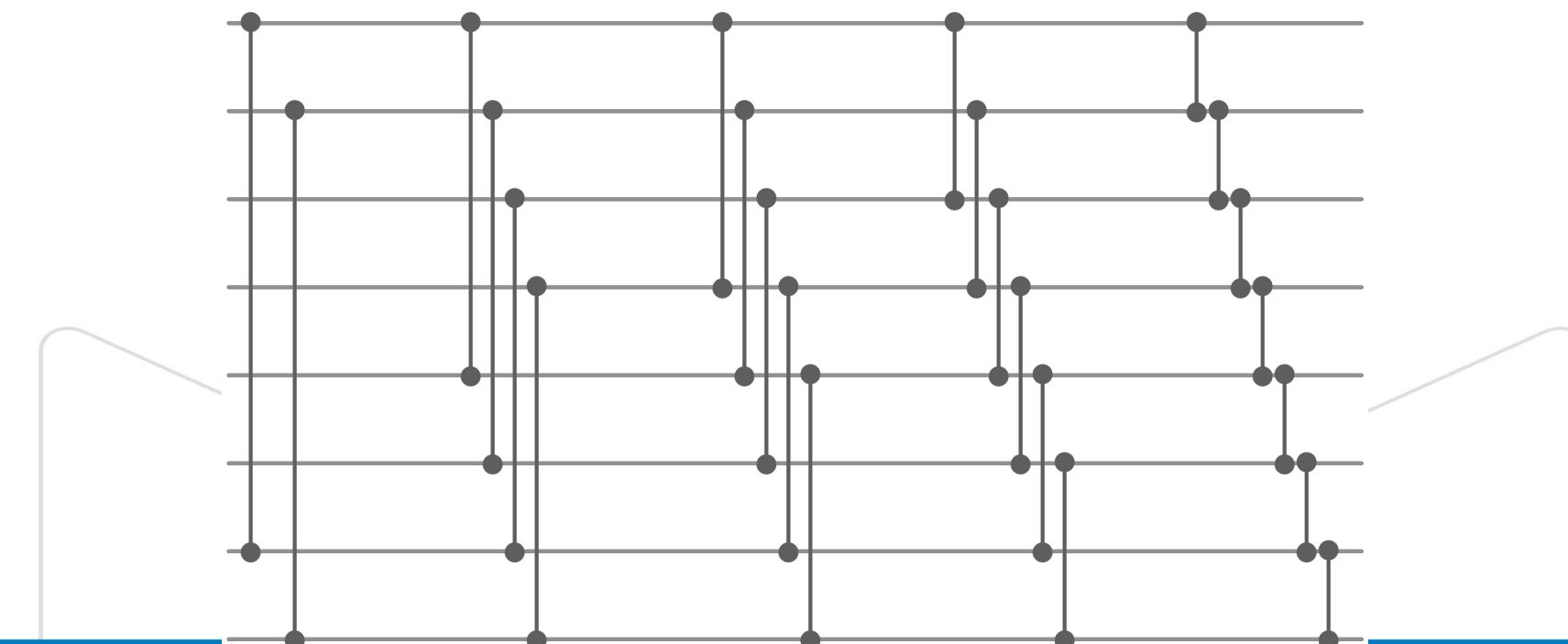
– 网络深度为 $n - 1$ (为什么 $n - 1$ 层已经足够?)

- 使用 0-1-principle 及数学归纳法证明
- 当 $n = 2$ 时, 显然 $n - 1 = 1$ 层已经足够
- 假设当 $n = k - 1$ 时, 深度为 $k - 2$ 的网络为排序网络, 由下图可知在添加一层后, 深度为 $k - 1$ 的网络为对 $n = k$ 数组的排序网络



● 奇偶移项排序网络 (odd-even transposition sort)

- 存在问题：每次只交换相邻元素，对于移动距离较长的元素，需要进行多次交换
 - 必须有 $n - 1$ 步才能保证排序完成（从一端至另一端）
- 希尔排序 (shellsort) : 时间复杂度 $O(n \log^2 n)$
 - 步长变化的插入排序，可用排序网络实现



- 排序算法定义及相关性质
- 插入排序、冒泡排序与相应排序网络
- 归并排序与双调排序
- 其他适合并行的排序算法



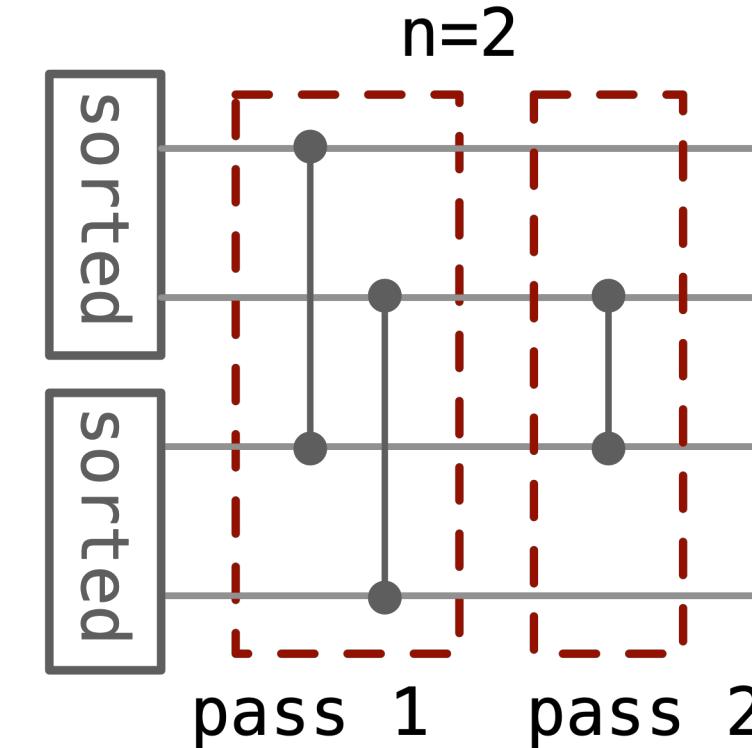
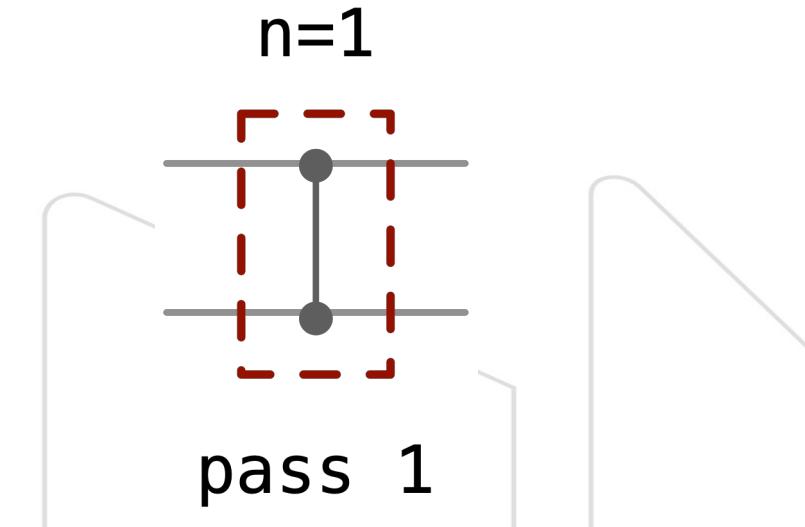
- 如何将层数从 $O(n)$ 进一步降低?

- 之前在并行模式中最常见的模式：分治 (divide and conquer)

- $O(\log n)$

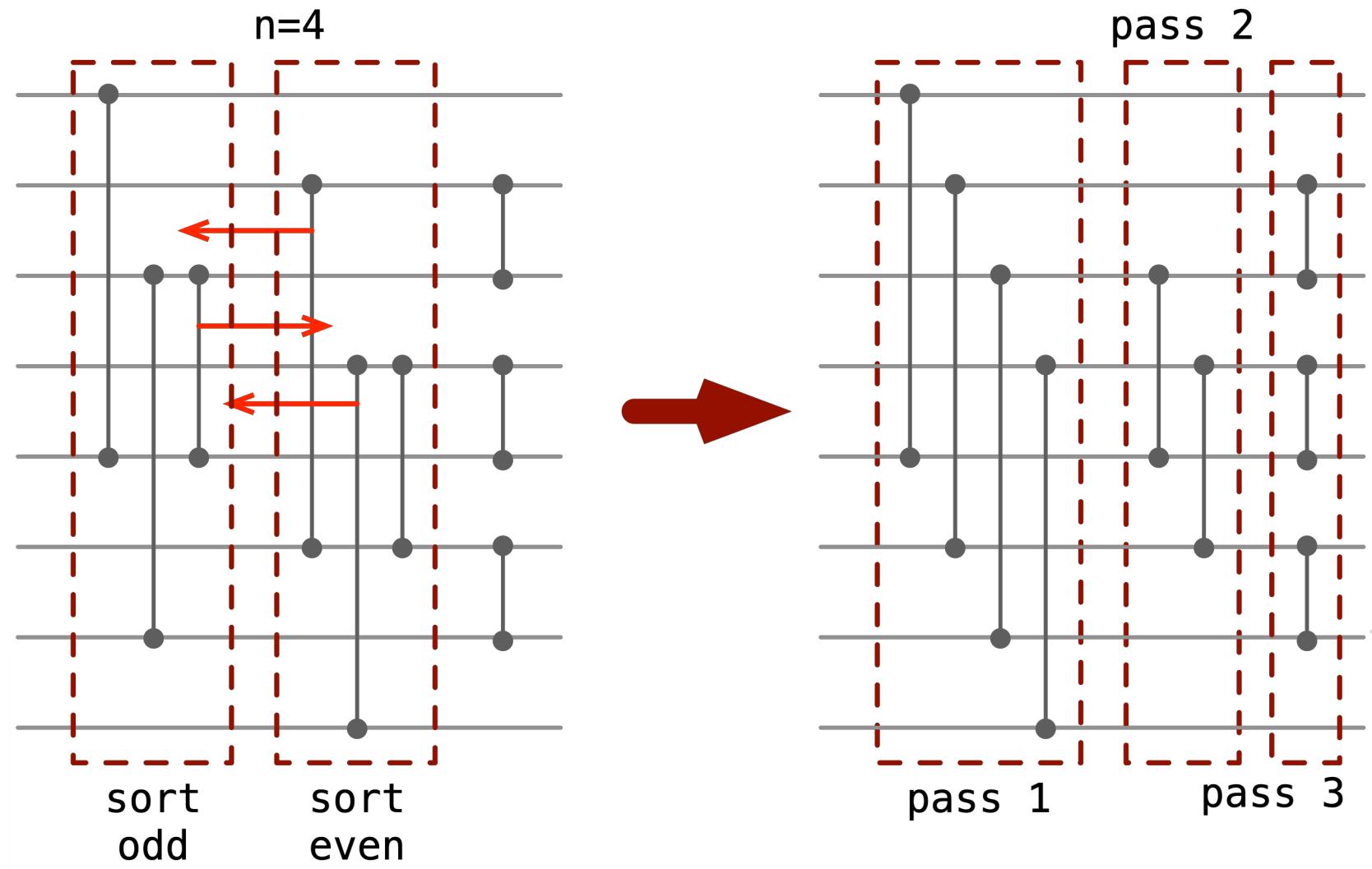
- 奇偶归并排序网络 (odd-even merge sorting network)

- 将两个分别排序的数列 $[a_0, \dots, a_{n-1}]$ 及 $[a_n, \dots, a_{2n-1}]$ 合并
 - 每次合并需要 $\log n$ 层
 - 总工作量为 $Q(n \log^2 n + \log n)$



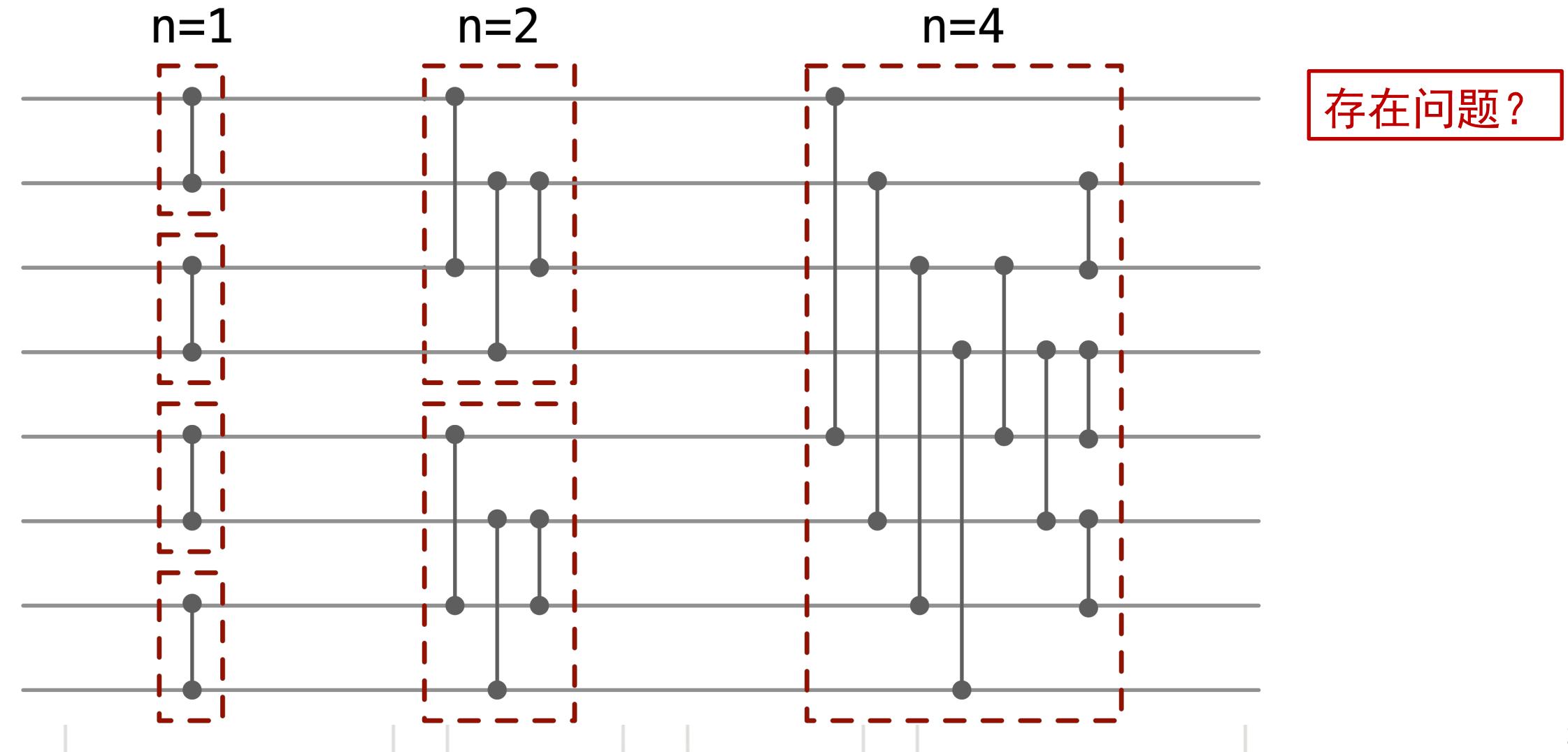
- 奇偶归并排序网络 (odd-even merge sorting network)

- 合并两个长度为4的序列



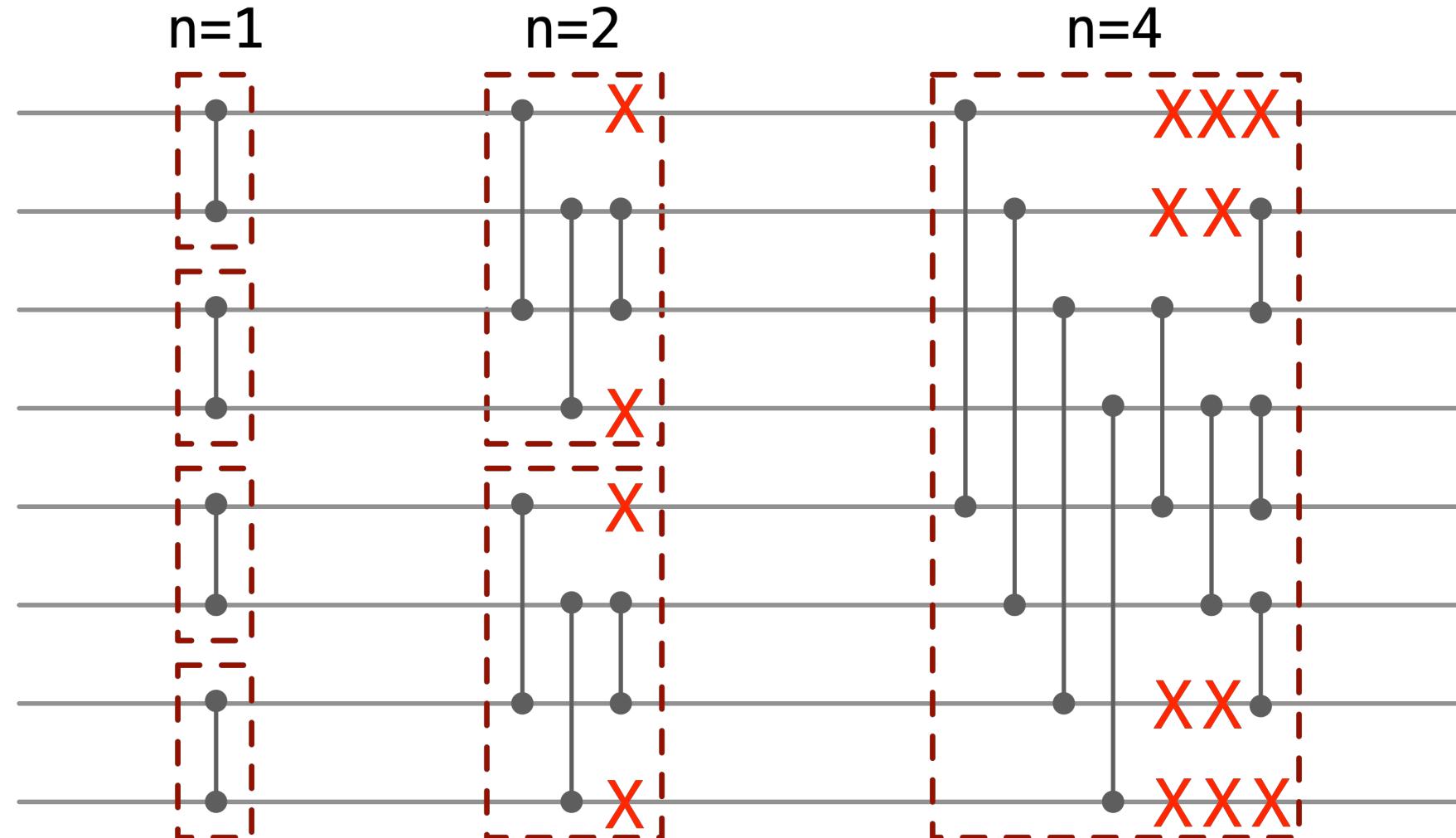
- 奇偶归并排序网络 (odd-even merge sorting network)

- 对长度为8的数组排序全过程



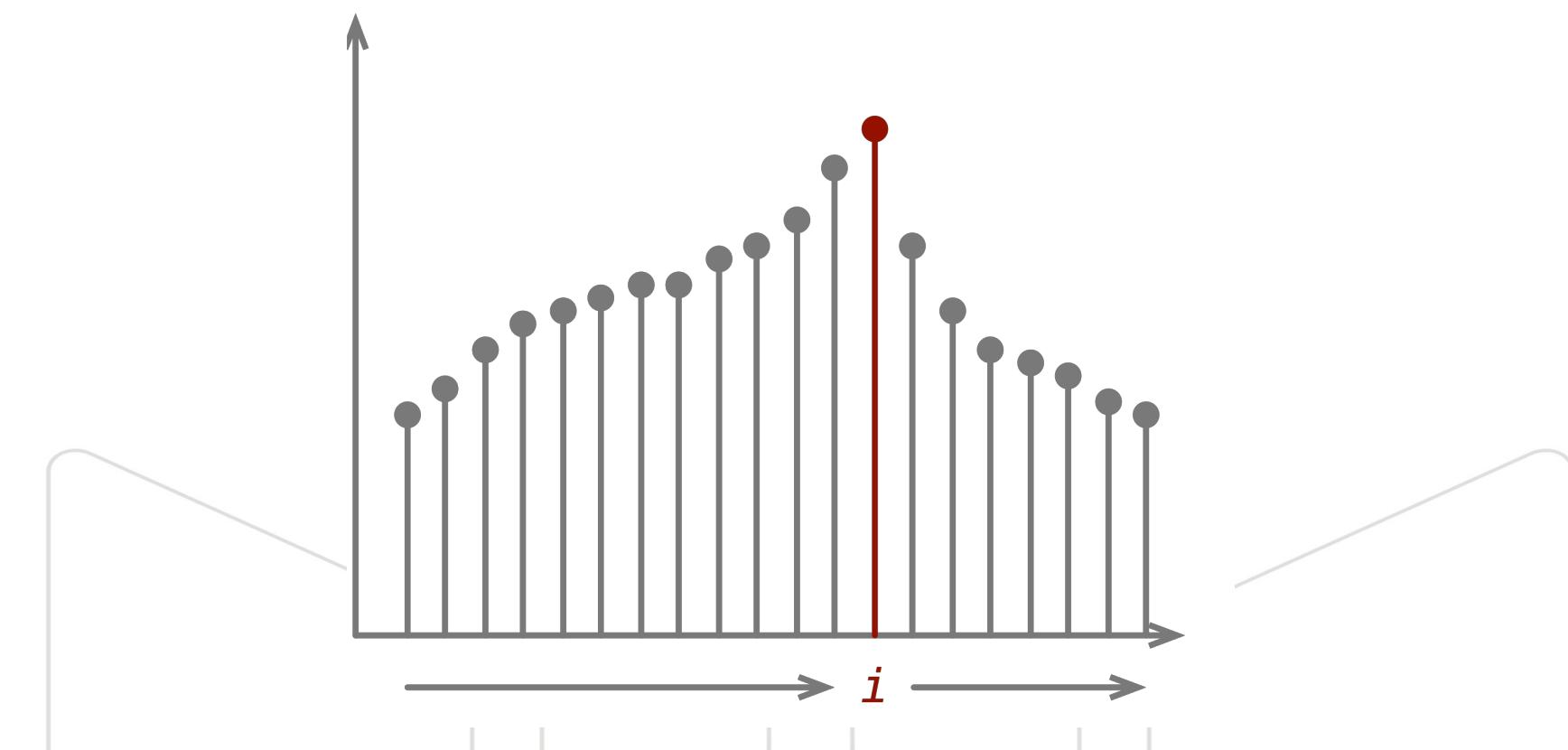
- 奇偶归并排序网络 (odd-even merge sorting network)

- 存在问题：不规则的数据访问模式；每一层运算量不一致（负载不均衡）；难于编程



- 双调排序网络 (bitonic sorting network)

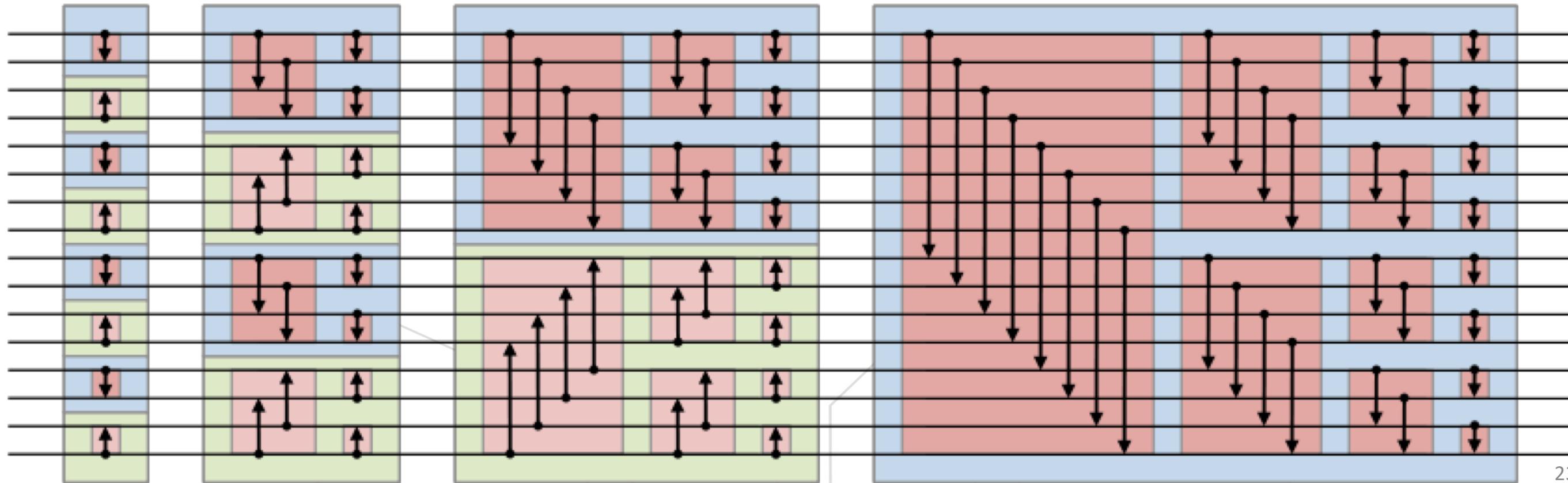
- 通过合并操作将输入合并为较大的双调序列
- 双调序列：对于序列 $[a_0, \dots, a_{n-1}]$ 存在*i*使得 $[a_0, \dots, a_i]$ 为单调递增序列，且 $[a_{i+1}, \dots, a_{n-1}]$ 为单调递减序列



● 双调排序网络 (bitonic sorting network)

- 第 i 步将一个长度为 2^i 的双调序列（两个长度为 2^{i-1} 的单调序列）变为一个长度为 2^i 的单调序列（相邻两个单调序列组成一个长度为 2^{i+1} 的双调序列）

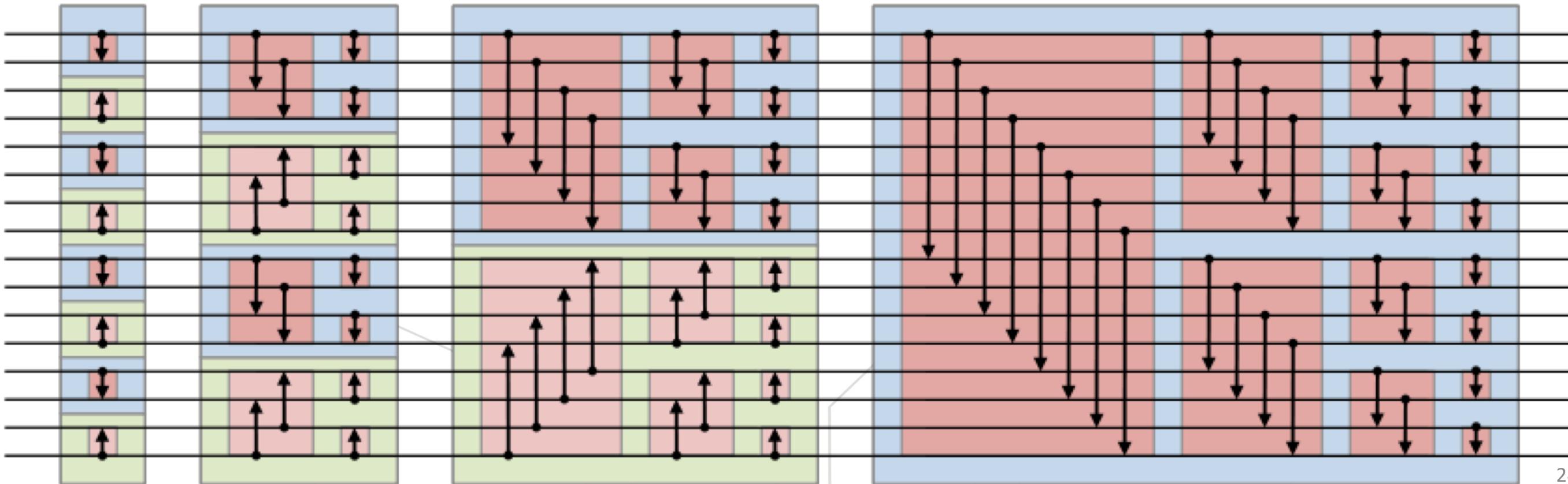
图片来自 Wikipedia



- 双调排序网络 (bitonic sorting network)

- 第 $i = 2^k$ 步完成，每步执行 $\frac{n}{2}$ 次比较
- 便于CUDA实现：规则的访问模式；负载均衡
- 当 $2^k > \text{block_size}$ 时，需要同步

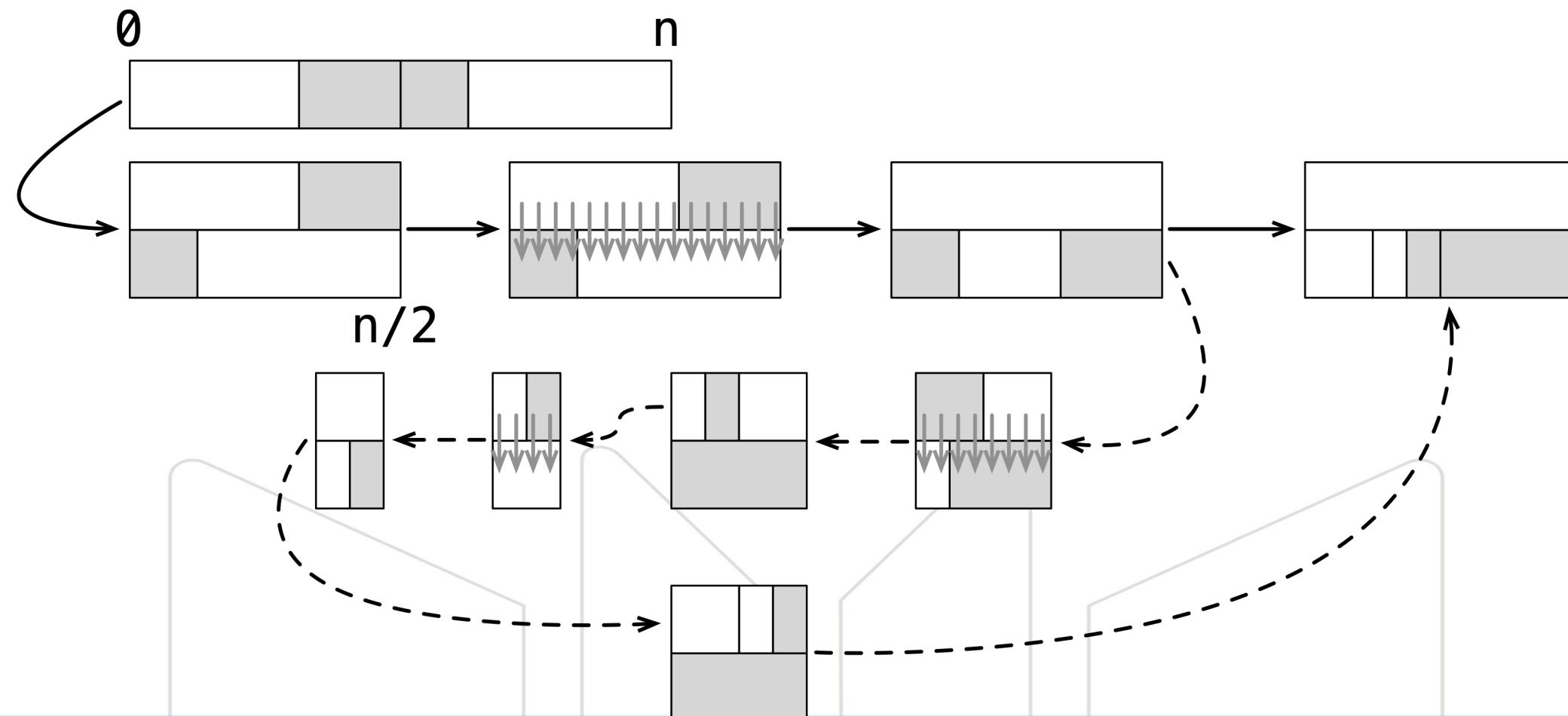
图片来自Wikipedia



● 双调排序网络 (bitonic sorting network)

– 正确性证明: 0-1-principle

- 双调的部分每次缩减为原先范围的一半，最终将两个双调序列合并为一个双调序列



- 排序算法定义及相关性质
- 插入排序、冒泡排序与相应排序网络
- 归并排序与双调排序
- 其他适合并行的排序算法



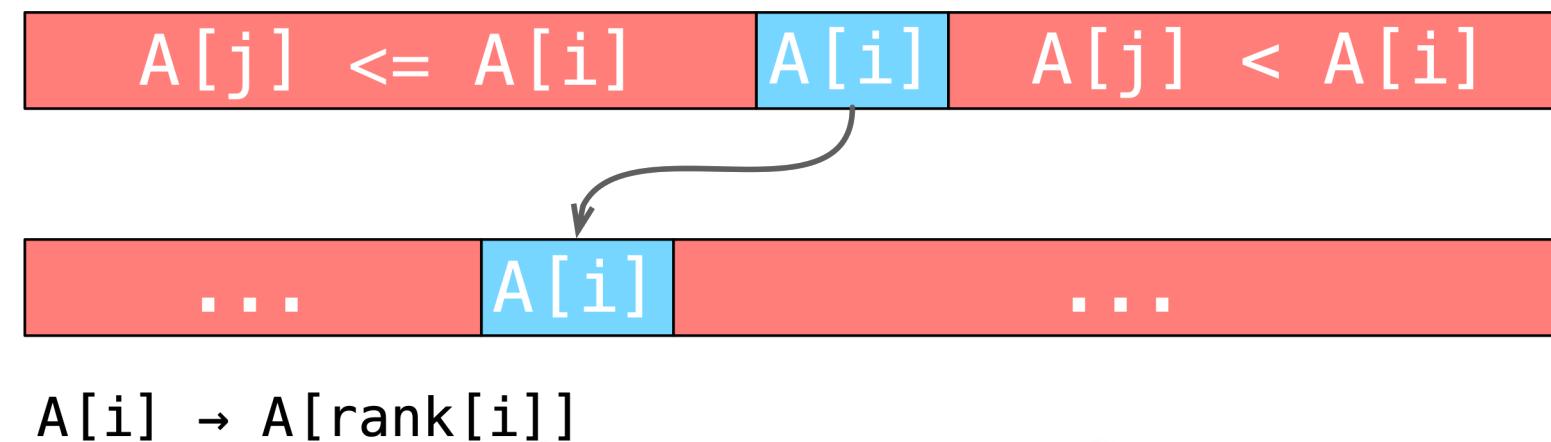
● 计数排序 (counting sort)

- 第一步：计算在A[i]左边的元素个数
 - 第二步：将A[i]置于相应位置

```

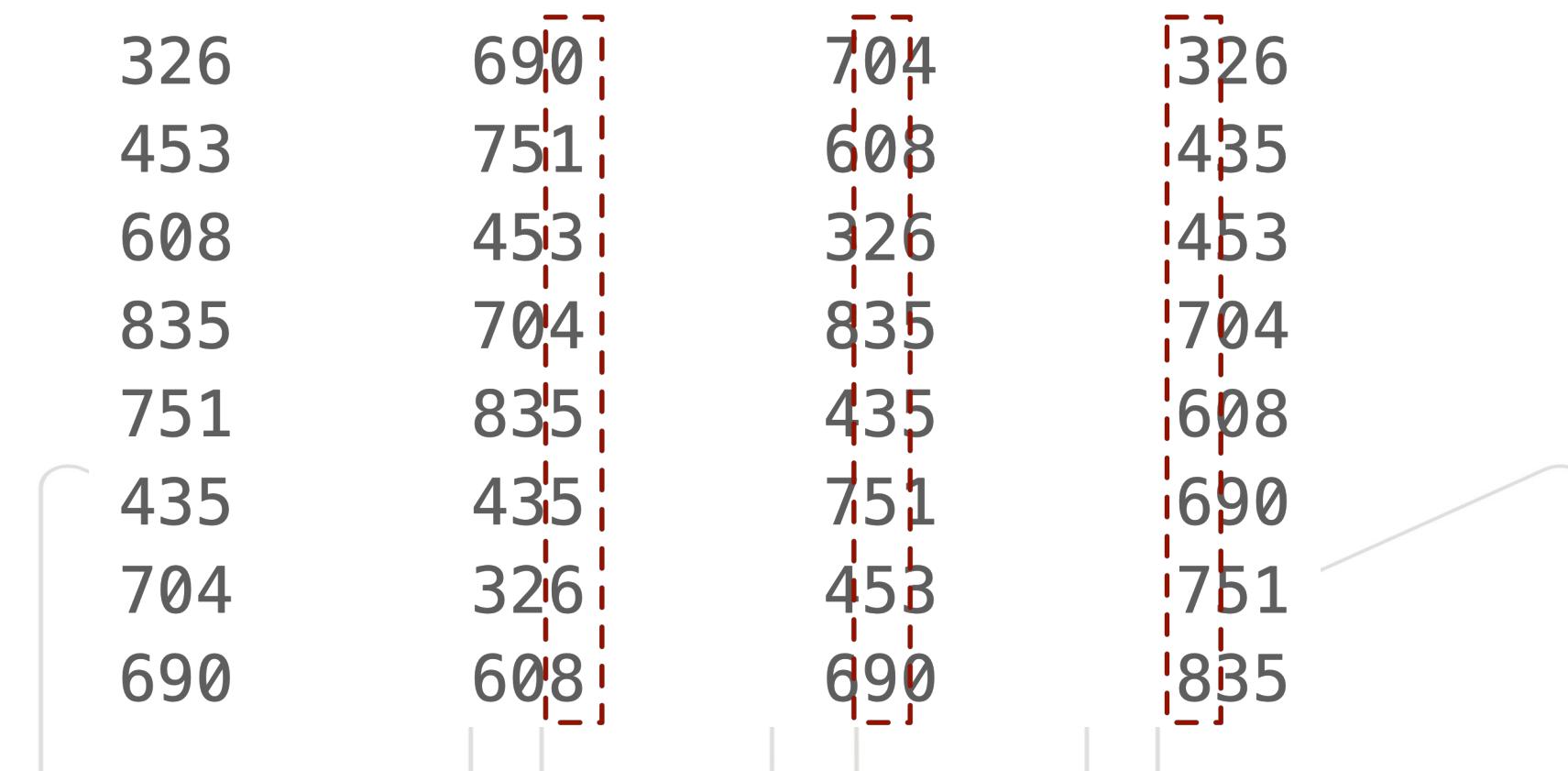
rank[i] = count ( j < i where A[j] <= A[i] )
          +count ( j > i where A[j] < A[i] )

```



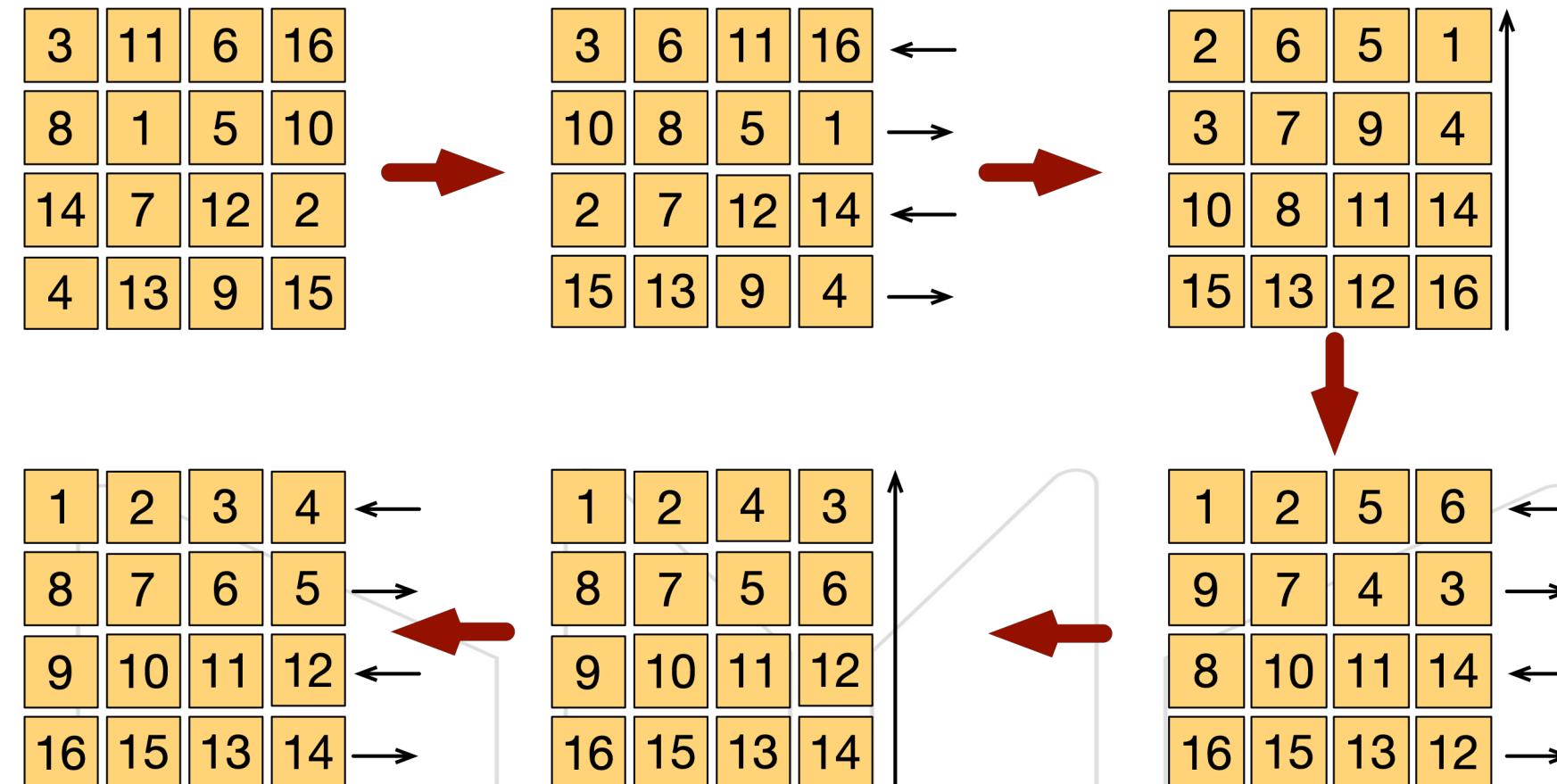
● 基数排序 (radix sort)

- 按位排序 (可从最高位或最低位开始)
- 对每一位排序时需要稳定的排序
- 可使用binary counting sort对二进制的每一位分别排序实现



● Shear sort

- 将数组沿“蛇形”放置在矩阵中
- 交替在行、列两个方向上进行排序
- 需要进行 $\log n + 1$ 次



● 排序网络

- 数据无关的排序架构
- 易于并行化
- 排序过程中需考虑内存访问模式

● 其他易于并行化的排序算法

- Shear sort
- 分治法：快速排序、归并排序
- 计数排序、基数排序



- Sorting network and the 0-1-principle

- <http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/sortieren.htm>
 - <http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/nulleinsen.htm>
 - <http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oetsen.htm>
 - https://en.wikipedia.org/wiki/Sorting_network#Zero-one_principle

- Sorting

- https://en.wikipedia.org/wiki/Sorting_algorithm

- Parallel Sorting

- https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_sorting.htm

- A great tutorial considering shell sort as a variant of bubble sort (odd-even)

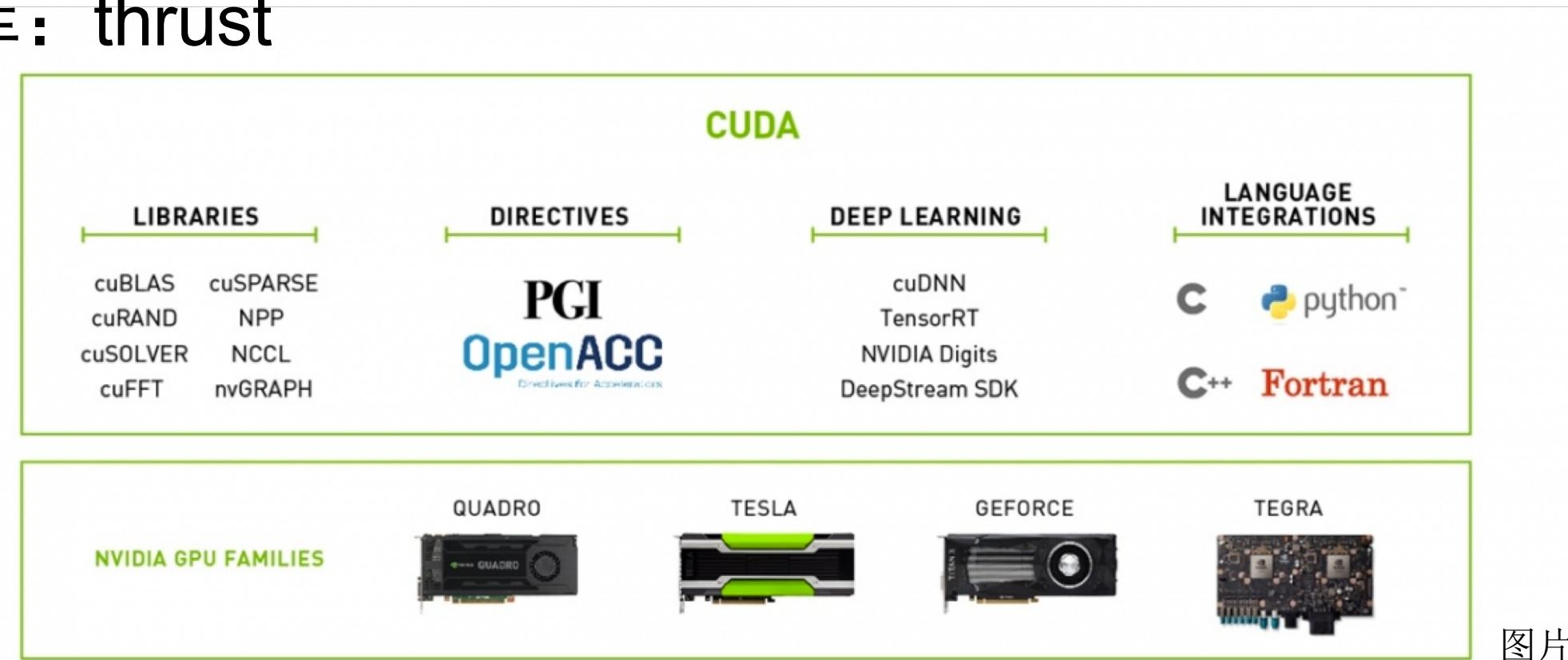
- <http://parallelcomp.uw.hu/ch09lev1sec3.html>

- Enumeration, odd-even, parallel merge, hyper quick sort

- https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_sorting.htm

- CUDA已经实现了大量常用函数

- 线性代数相关: cuBLAS、cuSPARSE、cuSOLVER
- 快速傅里叶变换: cuFFT
- 深度学习: cuDNN、TensorRT
- 模板库: thrust



图片来自NVIDIA

● Thrust

- CUDA使用的模板库
 - CUDA Toolkit中自带
- 与C++标准模板库（STL）在语法及设计上都十分相似
 - C++程序员可无障碍使用
 - 与CUDA C完全兼容
- 实现了许多并行程序中的基本组成模块
 - 并行扫描， 并行归约， 并行排序
 - 稳定性、性能、精度高
- 高度抽象， 隐藏CUDA细节
 - 核函数调用、内存分配、释放、拷贝等



图片来自NVIDIA

● Thrust容器类

- vector容器
 - 主机内存: host_vector
 - 设备内存: device_vector
- 其他容器
 - queue
 - priority_queue
 - list
 - stack
 - set
 - multiset
 - bitset
 - map

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

int main() {
    //create a vector on the host
    thrust::host_vector<int> h_vec(10);

    //create a vector on the device
    thrust::device_vector<int> d_vec = h_vec;

    //device data manipulated directly from host
    for (int i = 0; i < 10; i++)
        d_vec[i] = i;

    //vector memory automatically released
    return 0;
}
```

● Thrust迭代器

- 指向容器中的一个元素
- 操作与指针相似
 - 使用“*”取值时需要显示类型转换

```
thrust::device_vector<int>::iterator begin = d_vec.begin();
thrust::device_vector<int>::iterator end = d_vec.end();
printf("d_vec at begin=%d", (int)*begin);
begin++; //move on a single position
printf("d_vec at begin++=%d", (int)*begin);
*end = 88;
printf("d_vec at end=%d", (int)*end);
```

输出：

```
d_vec at begin=0
d_vec at begin++=1
d_vec at end=88
```

● Thrust迭代器

- 可转换为raw pointer
 - 可于核函数中使用

```
int * d_ptr = thrust::raw_pointer_cast(begin);
int * d_ptr = thrust::raw_pointer_cast(begin[0]);

kernel<<<...>>>(d_ptr);
```

- Raw points可以在Thrust中使用
 - 但用法与vector不完全相同

```
int* d_ptr;
cudaMalloc((void**)&d_ptr, N);

thrust::device_ptr<int> d_vec = thrust::device_pointer_cast(d_ptr);
//or
thrust::device_ptr<int> d_vec = thrust::device_ptr<int>(d_ptr)

cudaFree(d_ptr);
```

● Thrust中实现的算法

- Transformations
 - 使用指定函数作用于数组中每一个元素
 - 可以指定数组的一部分
- Reduction
 - 使用指定的二元操作将数组中的元素归约至一个元素
- Scan (prefix sum)
 - inclusive及exclusive扫描（参见课件7， 并行编程模式）
- Sort
 - 可对一组keys序列或一组 $\langle \text{key}, \text{value} \rangle$ 对组成的序列进行排序
- Binary search
 - 使用二分搜索在数组中查找特定值

● Thrust中实现的算法

– Transformations举例

```
thrust::copy(d_vec.begin(), d_vec.begin() + 10, d_vec_cpy.begin());  
  
thrust::fill(d_vec.begin(), d_vec.begin() + 10, 0);  
  
thrust::generate(d_vec.begin(), d_vec.begin() + 10, rand);  
  
//rand is a predefined Thrust generator  
thrust::generate(d_vec.begin(), d_vec.begin() + 10, rand);  
  
// fill d_vec with {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  
thrust::sequence(d_vec.begin(), d_vec.begin() + 10);  
  
//all occurrences of the value 1 are replaced with the value 10  
thrust::replace(d_vec.begin(), d_vec.end(), 1, 10);
```

- Thrust中实现的算法
 - 使用自定义Transformations

```
thrust::device_vector<int> d_vec(10);
thrust::device_vector<int> d_vec_out(10);

//fill d_vec with {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
d_vec = thrust::sequence(d_vec.begin(), d_vec.begin() + 10);

//declare a custom operator
struct add_5{
    _host_ _device_ int operator()(int a){
        return a + 5;
    }
};

add_5 func;

//apply custom transformation
thrust::transform(d_vec.begin(), d_vec.end(), d_vec_out.begin(), func);
//d_vec is now {5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
```

● Thrust中实现的算法

– 扫描举例

- 结果可保存至原数组，也可以另外指明输出数组

```
thrust::device_vector<int> d_vec(10);
thrust::device_vector<int> d_vec_out(10);

//fill d_vec with {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
thrust::sequence(d_vec.begin(), d_vec.begin() + 10);

//inclusive scan to output vector
thrust::inclusive_scan(d_vec.begin(), d_vec.end(), d_vec_out.begin());

//inclusive scan in place
thrust::inclusive_scan(d_vec.begin(), d_vec.end(), d_vec.begin());
```

● Thrust中实现的算法

– 排序举例

- 同样可以指明输出数组

```
thrust::device_vector<int> d_vec(10);
thrust::device_vector<int> d_vec_out(10);

//generate random data
thrust::generate(d_vec.begin(), d_vec.end(), rand);
//in-place sort
thrust::sort(d_vec.begin(), d_vec.end());
//sort to output vector
thrust::sort(d_vec.begin(), d_vec.end(), d_vec_out.begin());

//generate random keys
thrust::generate(d_keys.begin(), d_keys.end(), rand);
//generate random values
thrust::generate(d_values.begin(), d_values.end(), rand);
//sort by keys
thrust::sort_by_key(d_keys.begin(), d_keys.end(), d_values.begin());
```

● Thrust中实现的算法

- Thrust算法合并：减少不必要的函数调用

```
struct absolute{
    __host__ __device__ int operator()(int a){
        return a < 0 ? -a : a ; }
};

absolute func;

//custom transformation to calculate absolute value
thrust::transform(d_vec.begin(), d_vec.end(), d_vec.begin(), func);
//apply reduction, maximum binary associate operator
int result = thrust::reduce(d_vec.begin(), d_vec.end(), 0, thrust::maximum<int>());
```

```
//apply transform reduction maximum binary associate operator
int result = thrust::transform_reduce(d_vec.begin(), d_vec.end(), func, 0,
thrust::maximum<int>());
```

Questions?

