

GPU Computing

Laurea Magistrale in Informatica - AA 2018/19

Docente **G. Grossi**

Lezione 8 – Parallel patterns: scan & sorting

Sommario

- ✓ Parallel patterns: scan e sorting
- ✓ Definizione di Scan e ottimizzazione
- ✓ QuickSort, mergeSort e bitonicSort
- ✓ Implementazioni degli algoritmi di sorting e simulazioni

Modelli di programmazione parallela

- ✓ Sequential models

- ✓ von Neumann (RAM) model

- ✓ Parallel model

- A **parallel computer** is simple a collection of processors interconnected in some manner to coordinate activities and exchange data
 - **Models** that can be used as general frameworks for describing and analyzing parallel algorithms
 - **Simplicity:** description, analysis, architecture independence
 - **Implementability:** able to be realized, reflect performance

- ✓ Three common parallel models

- ✓ **Directed acyclic graphs, shared-memory, network**

DAG & shared memory

DAG

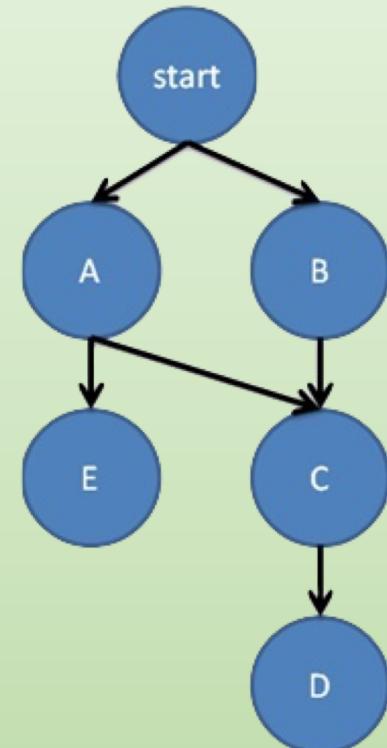
- ✓ Captures data flow parallelism
- ✓ Nodes represent operations to be performed
 - Inputs are nodes with no incoming arcs
 - Output are nodes with no outgoing arcs
 - Think of nodes as tasks
- ✓ Arcs are paths for flow of data results
- ✓ DAG represents the operations of the algorithm
- ✓ and implies precedent constraints on their order

SHARED MEMORY MODELS

- ✓ Parallel extension of RAM model (PRAM)
 - Memory size is infinite
 - Number of processors is unbounded
 - Processors communicate via the memory
 - Every processor accesses any memory location in 1 cycle
 - Synchronous
 - All processors execute same algorithm synchronously
 - READ phase
 - COMPUTE phase
 - WRITE phase
 - Some subset of the processors can stay idle
 - Asynchronous

Parallel pattern design

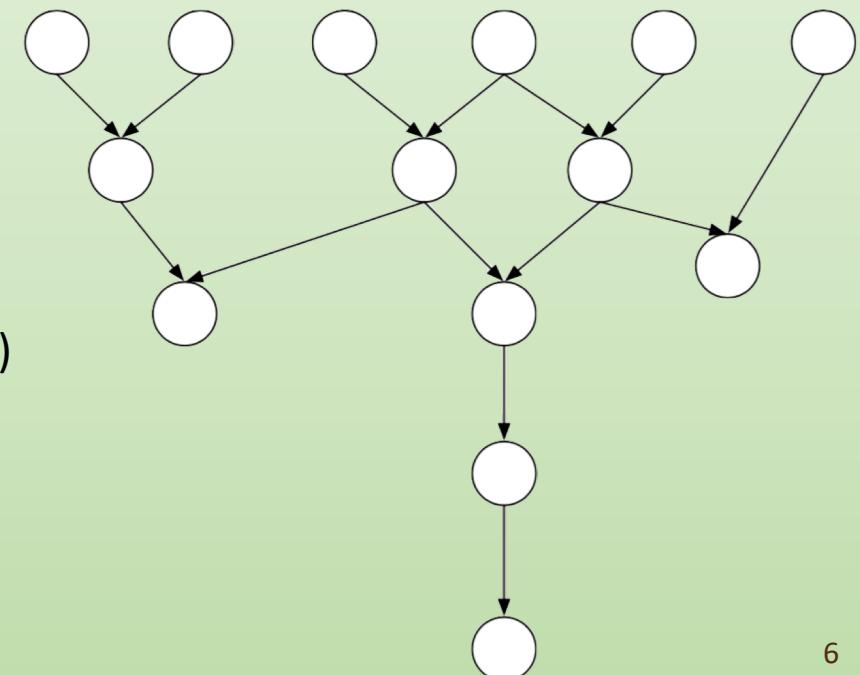
- ✓ Possiamo **decomporre** una computazione ‘complessa’ in **task** di più piccola taglia
- ✓ Lo scopo è determinare quali task si possono eseguire in parallelo
- ✓ Il trade-off è spesso tra: **molti piccoli task vs pochi grandi task** (granularità della computazione)
- ✓ **Task-dependency graph (DG)**
 - **grafo orientato aciclico** (condizione perché la computazione sia feasible)
 - i **nodi** sono i **task**
 - esiste un **arco** tra il **task A** e il **task B** se B deve essere eseguito dopo A
- ✓ **Massimo grado di concorrenza:** massimo numero di task che possono essere eseguiti in parallelo
- ✓ **Grado medio di concorrenza:** il numero medio di task che possono essere eseguiti in parallelo
- ✓ **NOTA:** Il grado medio potrebbe essere più utile nel descrivere il parallelismo effettivo



Cammini critici

- ✓ **Nodo iniziale:** senza archi entranti
- ✓ **Nodo finale:** senza archi uscenti
- ✓ **Cammini critico:** il più lungo cammino orientato tra il nodo iniziale e il nodo finale
- ✓ **Lunghezza di un cammino critico:** la somma dei pesi degli archi del cammino
- ✓ **Grado medio di concorrenza:** Lavoro totale / lunghezza del cammino critico

- **Massimo grado di concorrenza = 6**
- **Lunghezza di un cammino critico = 5**
- **Lavoro totale = 14** (assumendo ogni task abbia costo unitario)
- **Grado medio di concorrenza = $14/5 = 2.8$**



Scalabilità e Speed-up

La scalabilità si riferisce in genere a due cose:

1. **parallelismo** che si ha quando problemi di grandi dimensioni contengono molte operazioni indipendenti
2. **speedup** che si ottiene con l'uso di molti processori producendo risultati in tempi rapidi

Lo **speedup** S di una implementazione parallela rispetto a quella sequenziale è definita come il rapporto tra il tempo di esecuzione con 1 processore e il tempo di esecuzione con p processori:

$$S_p = \frac{t_1}{t_p}$$

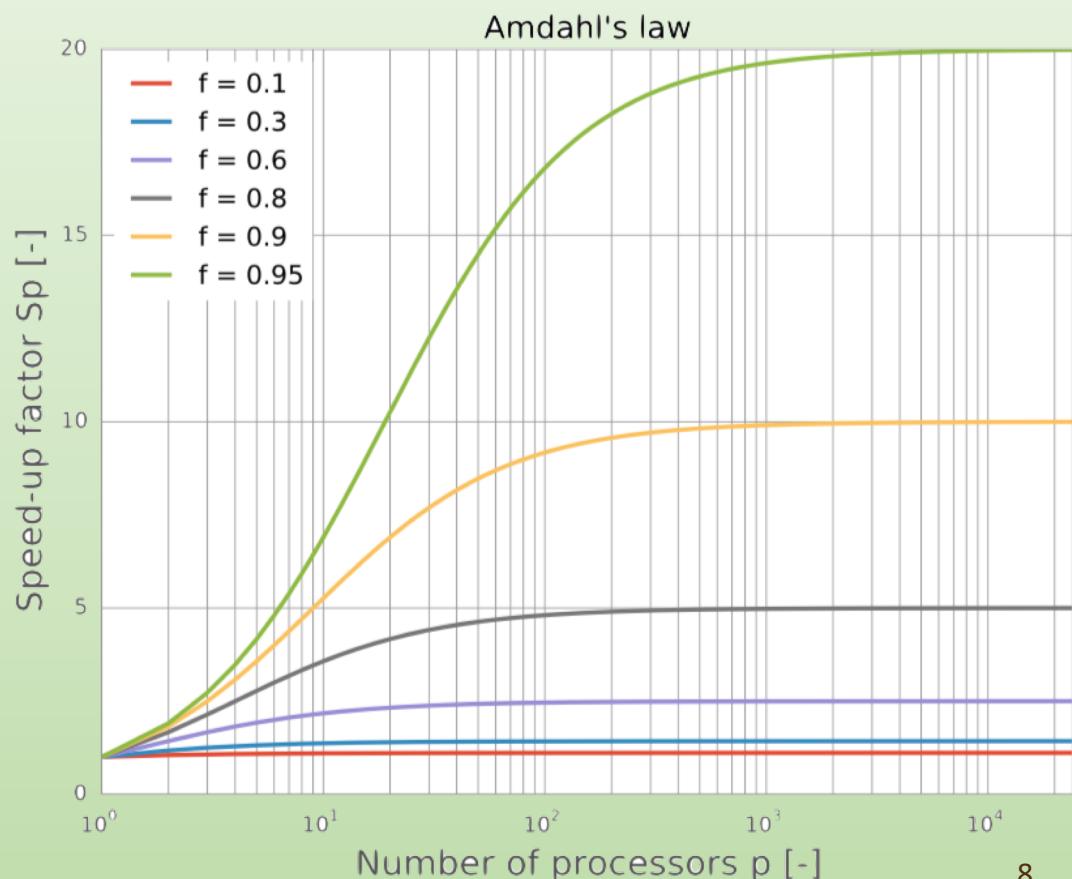
- ✓ Lo **speedup** S per una dato problema con **frazione parallelizzabile** f usando p **processori** può essere modellato con la legge di **Amdahl**
- ✓ Si assume che un certo carico di lavoro abbia **uguale tempo di computazione** su ogni processore e si abbia a complemento almeno una parte sequenziale di peso $(1 - f)$
- ✓ si assume che la **frazione parallelia** f sia **indipendente** rispetto alla taglia del problema

$$S_p = \frac{t_1}{t_p} = \frac{t_1}{\frac{t_1}{p}(f + (1 - f)p)}$$

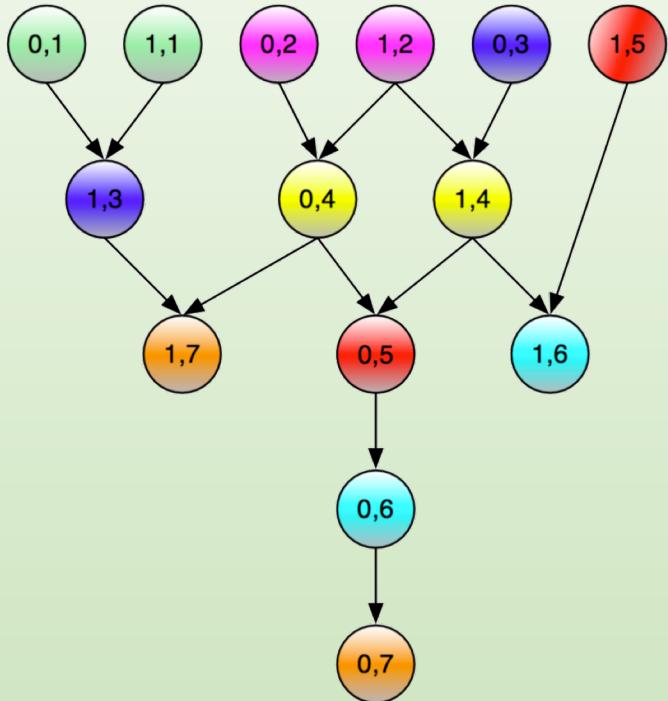
Amdahl's law - significato

- ✓ Ahmdahl's law mostra che il potenziale **speedup** è definito dalla frazione parallela del codice
- ✓ Ahmdahl's law assume che ***p*** and ***f*** siano **indipendenti** e questo non è necessariamente vero
- ✓ Il miglior **numero di processori** da usare nella parallelizzazione di un algoritmo dipende essenzialmente dalla **natura dello stesso**
- ✓ E' ovvio che **oltre un certo numero**, aggiungere processori non porta a significativi aumenti di prestazioni
- ✓ Lo **speedup non è scalabile indefinitamente** ma fortemente limitato superiormente come si vede in fig. o nella formula sopra
- ✓ Questo mette un **freno alle ambizioni** verso il **parallelismo massivo**

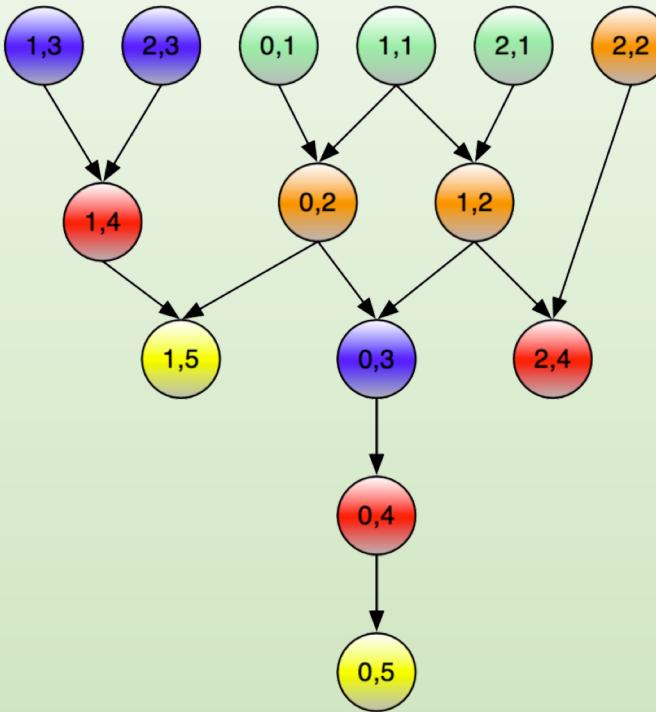
$$S_p = \frac{1}{(1-f) + \frac{f}{p}} < \frac{1}{1-f}, \quad \forall p$$



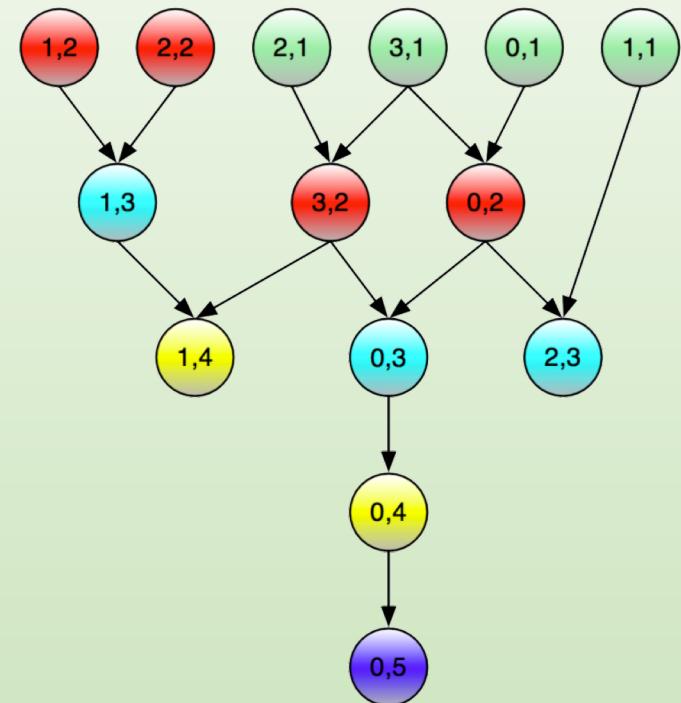
Esempi - speedup



processori = 2
Speedup = $14 / 7 = 2$



processori = 3
Speedup = $14 / 5 = 2.8$



processori = 4
Speedup = $14 / 5 = 2.8$

Esempio: decomposizione LU

Decomposizione matriciale mediante matrici triangolari LU (Upper e Lower)

$$\begin{matrix} A \\ \left[\begin{matrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{matrix} \right] \end{matrix} = \begin{matrix} L \\ \left[\begin{matrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{matrix} \right] \end{matrix} \times \begin{matrix} U \\ \left[\begin{matrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{matrix} \right] \end{matrix}$$

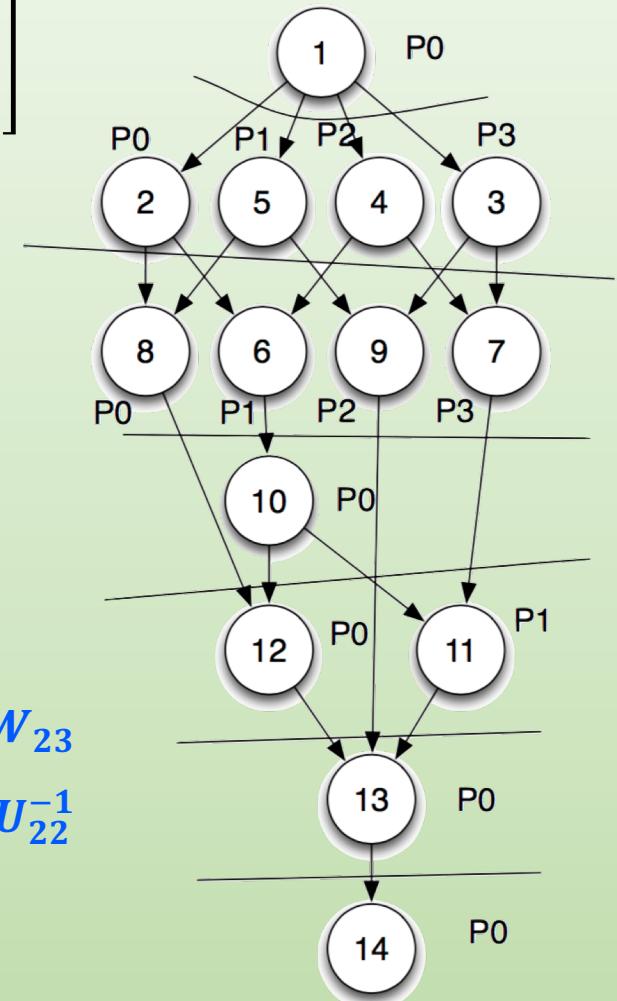
Da: $A_{11} = L_{11}U_{11}$ $\rightarrow \left\{ \begin{matrix} L_{11} \text{ e } U_{11} \end{matrix} \right.$

$$\begin{matrix} A_{21} = L_{21}U_{11} \\ A_{31} = L_{31}U_{11} \end{matrix} \rightarrow \left\{ \begin{matrix} L_{21} = A_{21}U_{11}^{-1} & L_{31} = A_{31}U_{11}^{-1} \\ U_{12} = A_{21}L_{11}^{-1} & U_{13} = A_{31}L_{11}^{-1} \end{matrix} \right.$$

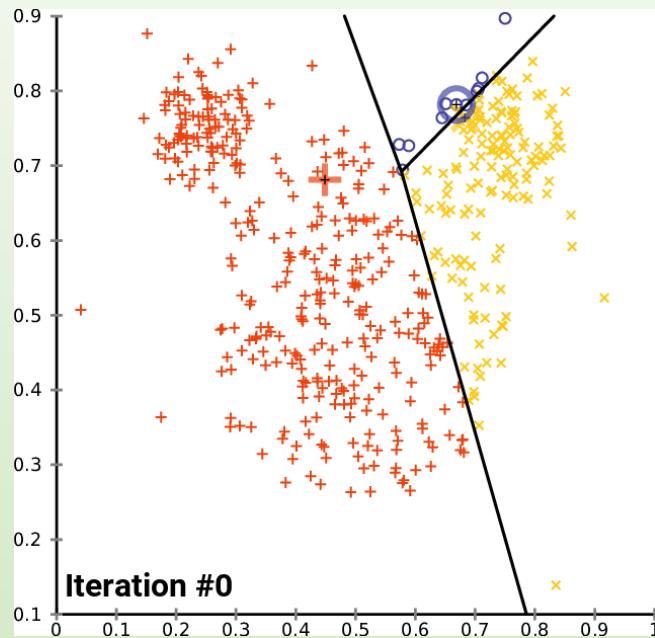
$$A_{22} = L_{21}U_{12} + L_{22}U_{22} \rightarrow \left\{ \begin{matrix} L_{22} \text{ e } U_{22} \end{matrix} \right.$$

$$\begin{matrix} A_{23} = L_{21}U_{13} + L_{22}U_{23} \\ A_{32} = L_{31}U_{12} + L_{32}U_{22} \end{matrix} \rightarrow \begin{matrix} W_{23} = A_{23} - L_{21}U_{13} = L_{22}U_{23} \\ W_{32} = A_{32} - L_{31}U_{12} = L_{32}U_{22} \end{matrix} \rightarrow \left\{ \begin{matrix} U_{23} = L_{22}^{-1}W_{23} \\ L_{32} = W_{32}U_{22}^{-1} \end{matrix} \right.$$

$$A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33} \rightarrow \left\{ \begin{matrix} L_{33} \text{ e } U_{33} \end{matrix} \right.$$



Algoritmo di clustering k-means



1. Initialize **cluster centroids** $\mu_1, \mu_2, \dots, \mu_k \in \mathbb{R}^n$ randomly.
2. Repeat until convergence: {

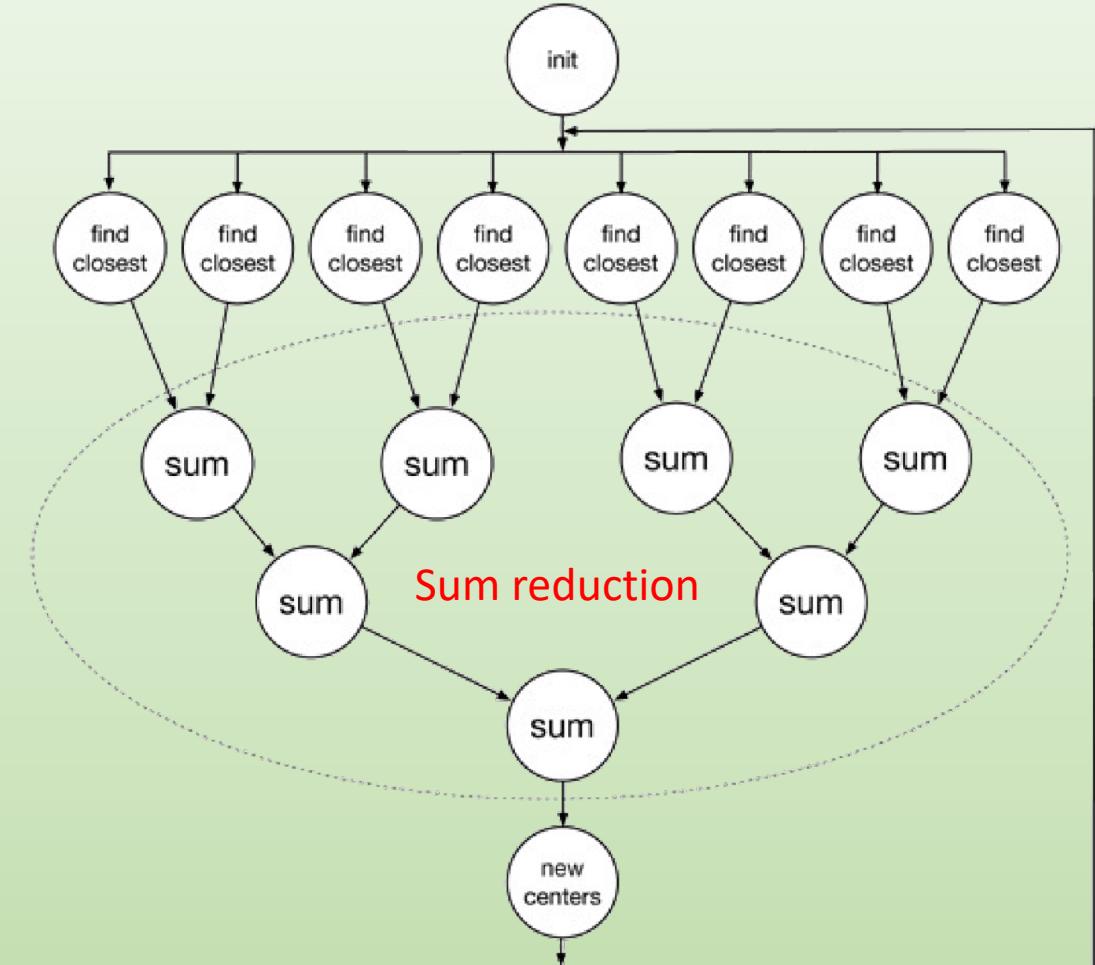
For every i , set

$$c^{(i)} := \arg \min_j \|x^{(i)} - \mu_j\|^2.$$

For each j , set

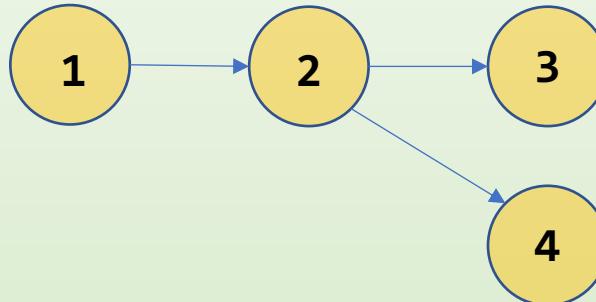
$$\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$$

}



DG a granularità più fine

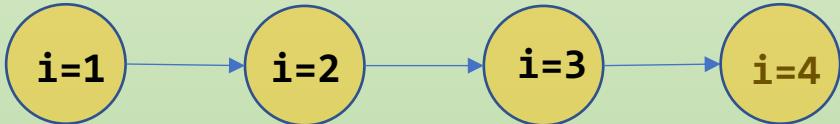
```
1: a <- 5  
2: b <- 3 + a  
3: c <- 2 * b - 1  
4: d <- b + 1
```



```
for i < 0 to n - 1 do  
    c[i] <- a[i] + b[i]  
end
```



```
for i < 1 to n - 1 do  
    c[i] <- c[i-1] + 2  
end
```



Parallel pattern: scan (prefix-sum)

Definizione:

L'operazione **parallel scan** (chiamata anche **prefix-sum**) considera un operatore binario associativo \oplus e un array di n elementi $[a_0, a_1, \dots, a_{n-1}]$ e restituisce l'array $b = [a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-1})]$

Esempio (taglio del sandwich):

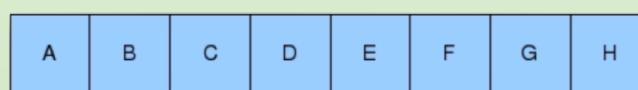
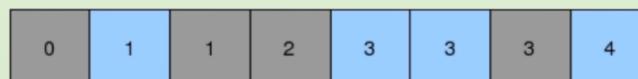
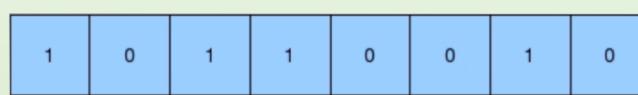
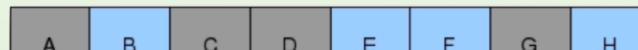
- ✓ Assumiamo di avere un sandwich molto lungo (diciamo 1 mt!) da distribuire tra 10 persone e che ognuna ne desidera (in cm) la quantità data nell'array **[3 5 2 7 28 4 3 0 8 1] ...**
- ✓ Qual è il modo più veloce per tagliare il sandwich? Dopo la distribuzione, quanto ne rimane?
- ✓ **Metodo 1:** taglio sequenziale delle sezioni richieste da ognuno: 3 cm al primo, 5 cm al secondo, etc
- ✓ **Metodo 2:** calcolo di prefix scan e taglio parallelo **[3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 cm avanzati)**

Scan: applicazioni

- ✓ Sorting: radix sort e quicksort
- ✓ String comparison
- ✓ Lexical analysis
- ✓ Stream compaction
- ✓ Polynomial evaluation
- ✓ Solving recurrences
- ✓ Tree operations
- ✓ Histograms
- ✓ Assigning space in farmers market
- ✓ Allocating memory to parallel threads
- ✓ Allocating memory buffer for communication channels

Esempi di applicazioni di scan

Stream compaction



Prefix sum

Run-length decoding in parallelo

4W 3B

0

4

2W

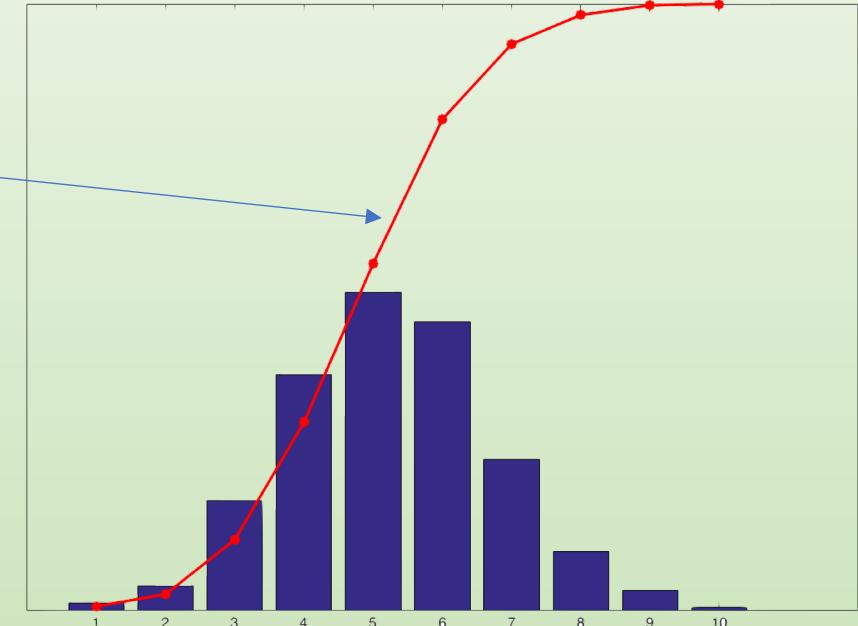
7

5B

9



Distribuzione cumulata empirica

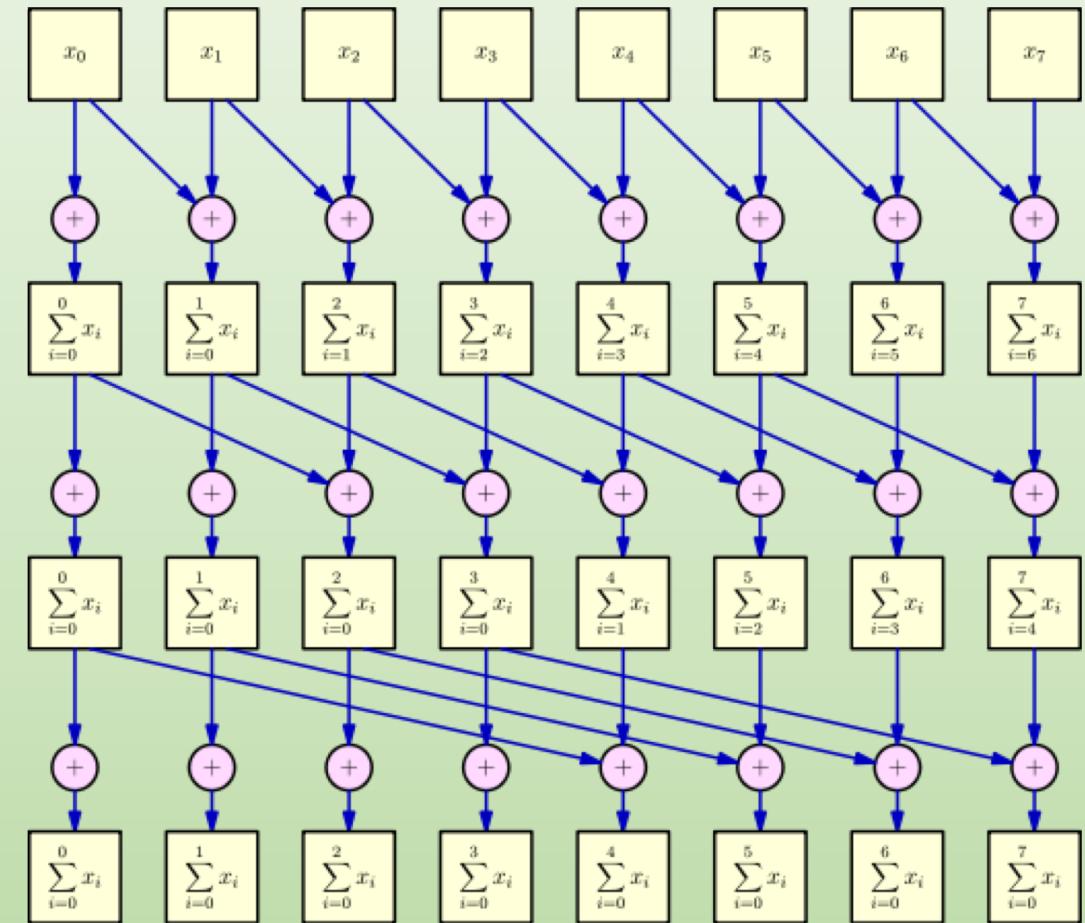


Parallel prefix sum

Codice sequenziale:

```
void scan_CPU(const float *a, float *b, int n) {  
    b[0] = a[0];  
    for (int i = 1; i < n; i++)  
        b[i] = b[i-1] + a[i];  
}
```

Schema parallelo->



Progettare mediante somma prefissa:

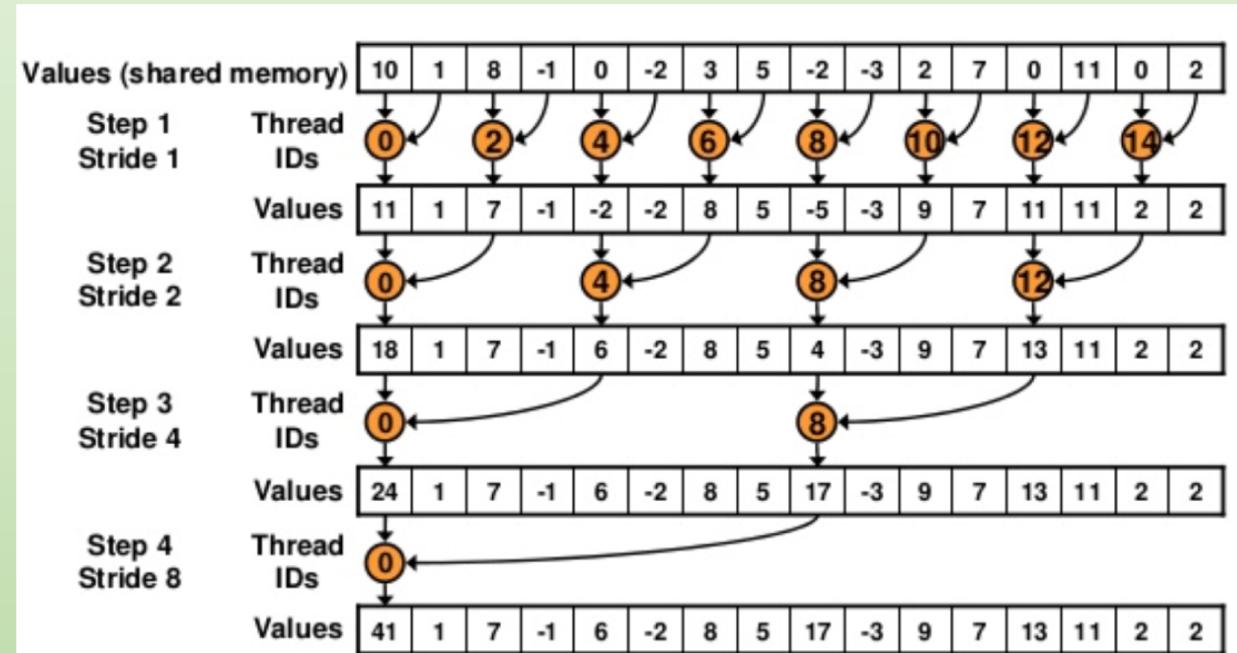
- Un kernel per l'operazione prefix-sum

Analogie con parallel reduction sum

Somma parallela su un solo blocco in shared:

```
for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2) {  
    __syncthreads();  
    smem[threadIdx.x] += smem[threadIdx.x - stride];  
}
```

Nota: Analogo allo schema di riduzione applicato alla somma anche se si hanno conflitti a livello di bank access nella memoria shared e con l'attenzione di mettere barriere di sincronizzazione se il dato supera la dimensione del warp



Schema: parallel prefix sum

Assunzione: n potenza di 2

$$n = 2^k$$

Passo 1:

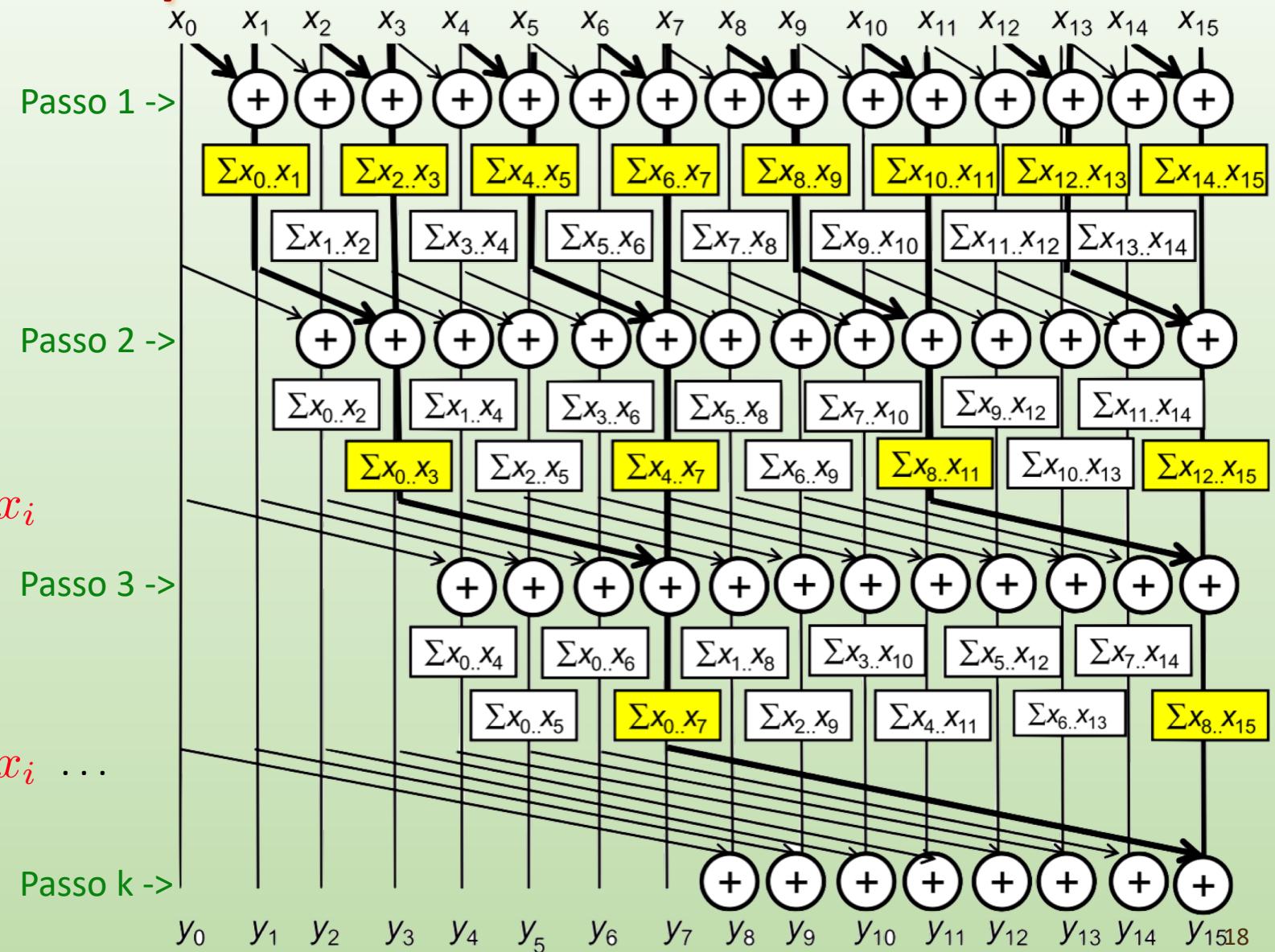
$$\forall i > 0 : y_i = x_{i-1} + x_i$$

Passo 2:

$$\forall i > 1 : y_i = x_{i-3} + \dots + x_i$$

Passo 3:

$$\forall i > 2 : y_i = x_{i-5} + \dots + x_i \dots$$



Algoritmo CUDA

```
__global__ void scan_GPU(float *x, float *y, unsigned int N) {
    __shared__ float smem[BLOCKSIZE];
    unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N)
        smem[threadIdx.x] = x[tid];

    // albero di riduzione: scan iterativo
    for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2) {
        __syncthreads();
        smem[threadIdx.x] += smem[threadIdx.x - stride];
    }
    y[tid] = smem[threadIdx.x];
}
```

- ✓ Il loop itera sull'albero di riduzione sommando su tutte le posizioni necessarie al thread per produrre l'output
- ✓ Nota l'uso delle barriere di sincronizzazione per assicurare che tutti i thread abbiano finito l'iterazione corrente prima di passare a quella successiva

Analisi efficienza

Versione sequenziale:

```
y[0] = x[0];  
for (int i = 1; i < n; i++)  
    y[i] = y[i-1] + x[i];
```

- Ottimizzando (lato compilazione) è richiesta una sola **addizione** e 2 operazioni tra **load** e **store** in memoria globale
- **Work efficiency** (numero di operazioni) : per un array di **N** elementi servono **un numero lineare di** operazioni... efficiente!

Work efficiency: $\mathcal{O}(N)$

Versione parallela su un solo blocco in shared:

- Tutti i thread iterano al più fino a **$\log N$** , con **N = BLOCKSIZE**
- Ad ogni passo il numero di thread che non effettua operazione è pari a **stride**

$$\sum_{\text{stride}=0}^{\log_2 N} (N - \text{stride}) = N \log_2 N - (N - 1)$$

- Tempo esecuzione **$O(\log N)$**
- **Work efficiency** (numero di operazioni) peggiore del caso sequenziale ... poco efficiente!

Work efficiency: $\mathcal{O}(N \log N)$

N	16	32	64	128	256	512	1024
N – 1	15	31	63	127	255	511	1023
N·log₂(N) – (N – 1)	49	129	321	769	1793	4097	9217

Parallel prefix sum work-efficient

Somma di 16 elementi in 4 passi:

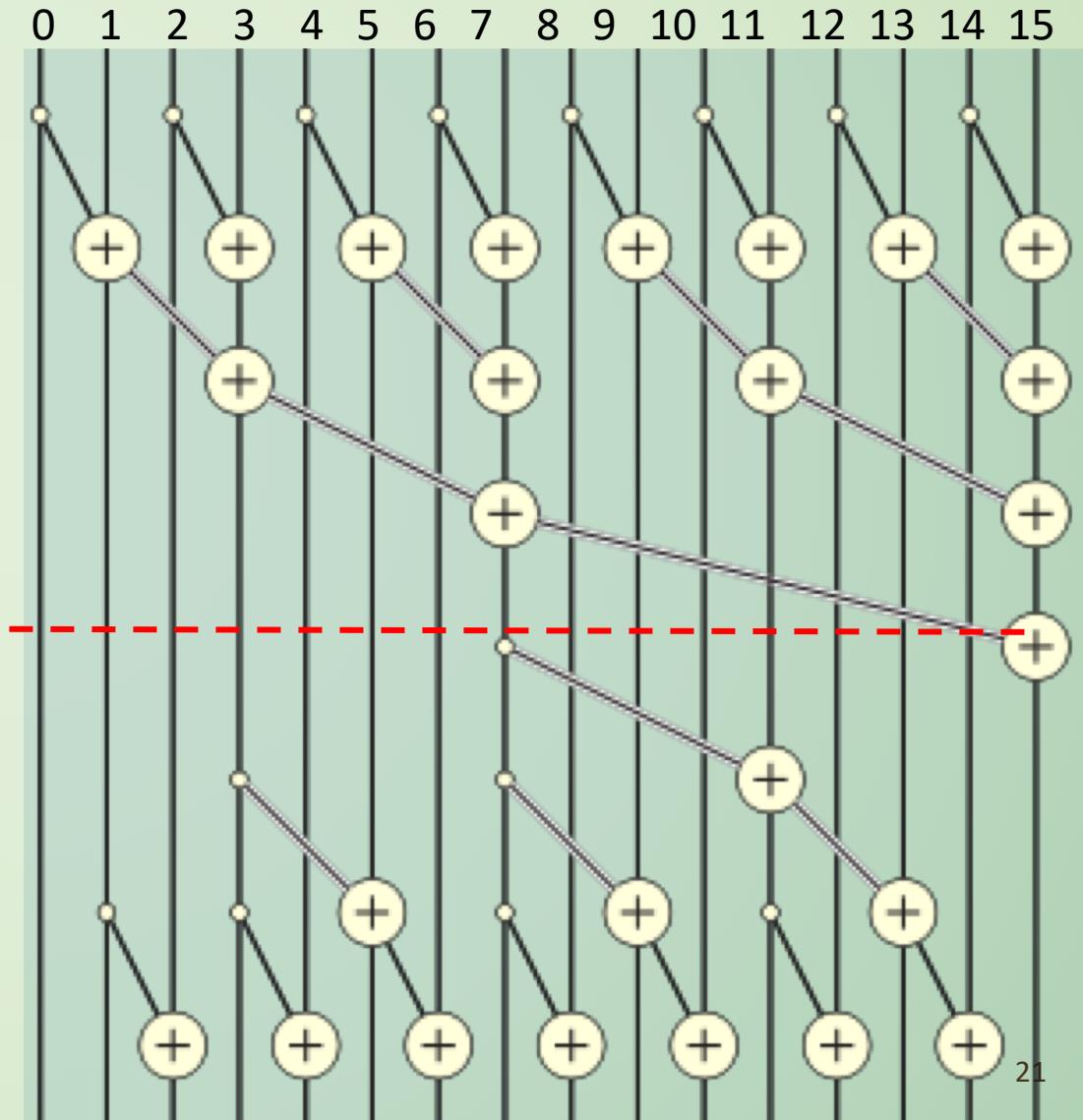
- **Step 1:** aggiorna solo gli elementi dispari
- **Step 2:** aggiorna solo gli elementi con indice della forma
 $4 \times n - 1$ (cioè 3, 7, 11, 15)

- **Step 3:** aggiorna solo gli elementi con indice della forma
 $8 \times n - 1$ (cioè 7, 15)

- **Step 4:** aggiorna solo l'ultimo (cioè 15)
- Numero operazioni: $8 + 4 + 2 + 1 = 15$
- In generale: $N/2 + N/4 + N/8 + \dots + 1 = N - 1$

Inversione dell'albero:

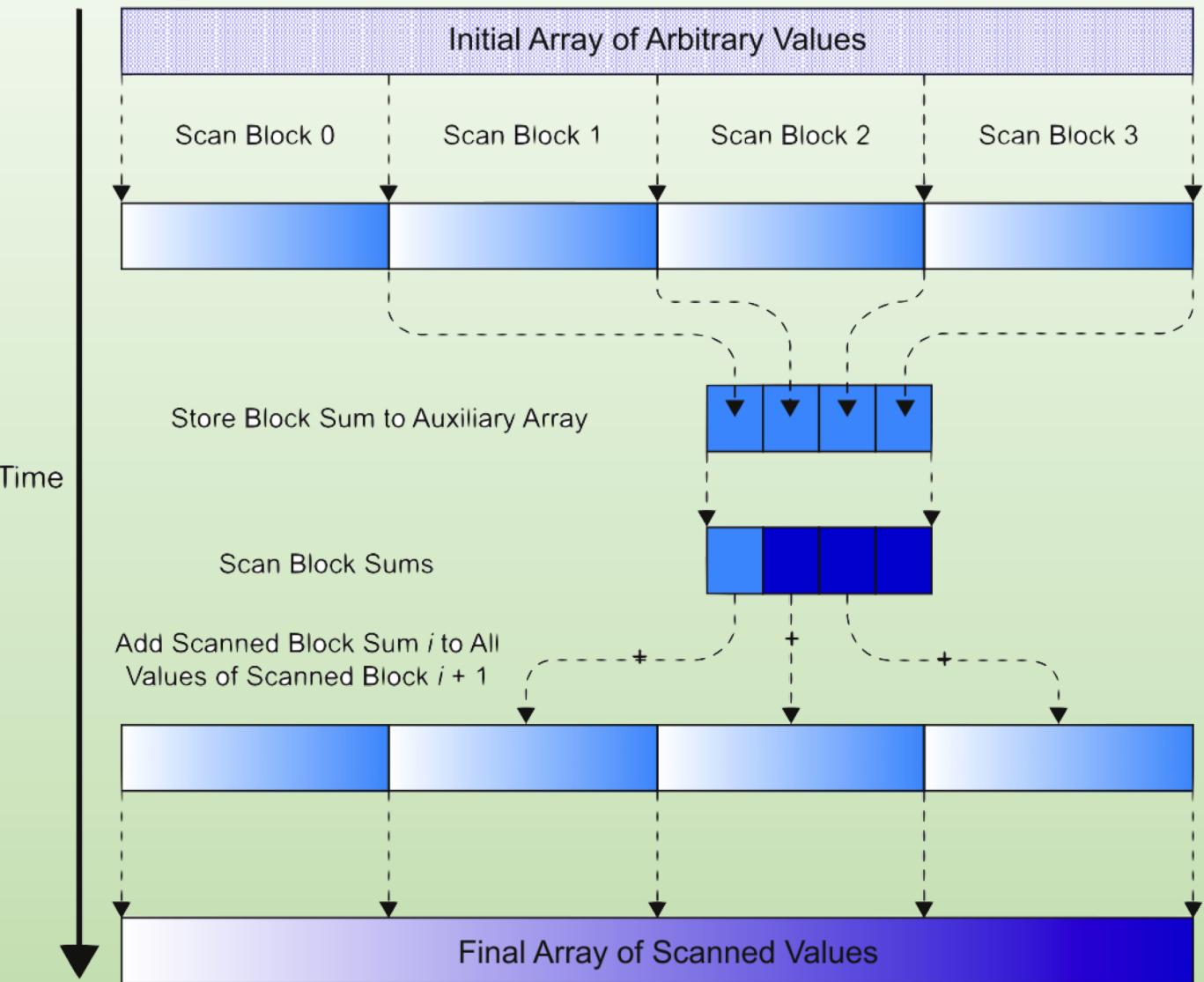
- Ridistribuire le somme parziali nelle posizioni che necessitano
- Usare schema rovesciato per la prima somma
- Work efficient: $\mathcal{O}(N)$



Scan parallelo per lunghezze arbitrarie

Come affrontare big data?

- ✓ Partizionare i dati in blocchi (come d'uso) che stiano in memoria shared
- ✓ elaborare i singoli blocchi come visto in precedenza
- ✓ Memorizzare la somma dei singoli blocchi in memoria ausiliaria
- ✓ Ogni elemento della memoria ausiliaria contiene la somma del blocco precedente
- ✓ Lanciare un kernel per la somma dei risultati parziali con gli elementi dei blocchi



Esercitazione

Somma prefissa:

Implementare l'algoritmo di somma prefissa per array arbitrario nelle versioni:

- Non work efficient
- Work efficient

Parallel pattern: sorting

- ✓ È importante avere un indicatore di confronto tra i vari algoritmi possibili di ordinamento, indipendentemente dalla piattaforma hw/sw e dalla struttura dell'elemento informativo
- ✓ La complessità computazionale si basa sulla valutazione del numero di operazioni elementari necessarie (Confronti, Scambi)
- ✓ Si misura come funzione del numero n di elementi della sequenza
- ✓ Gli algoritmi di ordinamento interno si dividono in
 - Algoritmi semplici - complessità $O(n^2)$
 - Algoritmi evoluti - complessità $O(n * \log_2 n)$

QuickSort

- ✓ Si tratta di un algoritmo evoluto che ha **complessità computazionale** $n * \log(n)$ ed inoltre usa la **ricorsione**
- ✓ Ordinare un vettore di n con **algoritmi semplici** si ottiene un tempo di calcolo **proporzionale** a n^2
- ✓ Dividendo il vettore in due sottovettori di $n/2$ elementi ciascuno e ordinando le due metà separatamente con algoritmi semplici si avrebbe un tempo di calcolo pari a

$$(n/2)^2 + (n/2)^2 = n^2/2$$

- ✓ Se **riunendo** i due **sottovettori ordinati** si potesse riottenere il **vettore originario** tutto ordinato avremmo dimezzato il tempo di calcolo
- ✓ Se questo ragionamento si potesse applicare anche immaginando una decomposizione del vettore originario in quattro, si avrebbe un tempo di calcolo totale pari a

$$(n/4)^2 + (n/4)^2 + (n/4)^2 + (n/4)^2 = n^2/4$$

- ✓ Se si potesse dividere in 8, si avrebbe un tempo ancora più basso, e così via dicendo

- ✓ Lo schema descritto sopra non funziona sempre



- ✓ **Vincolo:** possiamo applicare questo metodo solo se il **massimo** elemento in **A1** è **minore o uguale** al **minimo** elemento di **A2** (ordinamento crescente)
- ✓ L'operazione che crea due parti con la suddetta caratteristica si dice **partizionamento** del vettore

Schema QUICKSORT

1. se $|A| = 1$ allora A è ordinato
2. altrimenti
 - a. scegli un elemento **pivot** in A e ridistribuisci gli elementi di A in due sottoinsiemi A_1 e A_2 , disponendo in A_1 gli elementi minori del pivot e in A_2 gli elementi maggiori o uguali al pivot
 - b. applica nuovamente l'algoritmo sui due sottoinsiemi A_1 e A_2
3. fine-condizione

Scelta del pivot

- ✓ Dato il vettore V di 10 elementi:

13	10	1	45	15	28	21	11	29	34
0	1	2	3	4	5	6	7	8	9

$i = \inf$ $j = \sup$

- ✓ **Scelta pivot:** elemento di indice $m = (\inf + \sup)/2$, cioè $V[4] = 15$
- ✓ L'indice i viene fatto incrementare fino a quando si trova un elemento **maggiore o uguale** al pivot
- ✓ L'indice j viene fatto decrementare fino a quando si trova un elemento **minore o uguale** al pivot

13	10	1	45	15	28	21	11	29	34
0	1	2	3	4	5	6	7	8	9
		i					j		

Movimento indici

- ✓ Gli elementi di indice i e j vengono scambiati, e l'indice i viene incrementato, mentre j viene decrementato, ottenendo:

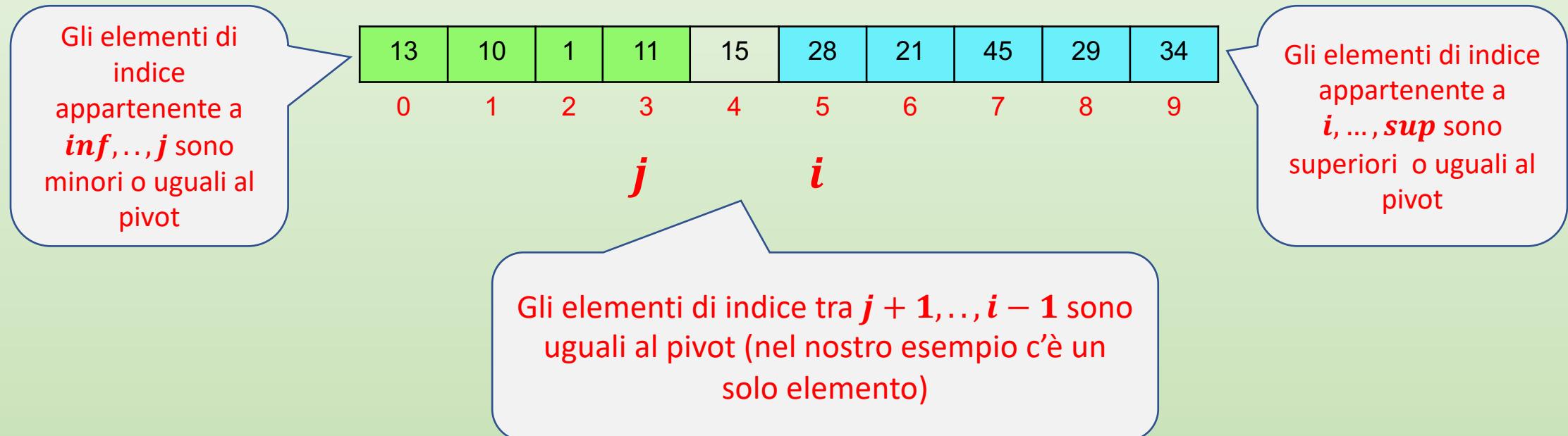
13	10	1	11	15	28	21	45	29	34
0	1	2	3	4	5	6	7	8	9
		i			j				

- ✓ L'indice i viene arrestato in quanto esso corrisponde al pivot
- ✓ L'indice j viene fatto decrementare fino a quando esso perviene all'elemento 15, che è uguale al pivot

13	10	1	11	15	28	21	45	29	34
0	1	2	3	4	5	6	7	8	9
		i,j							

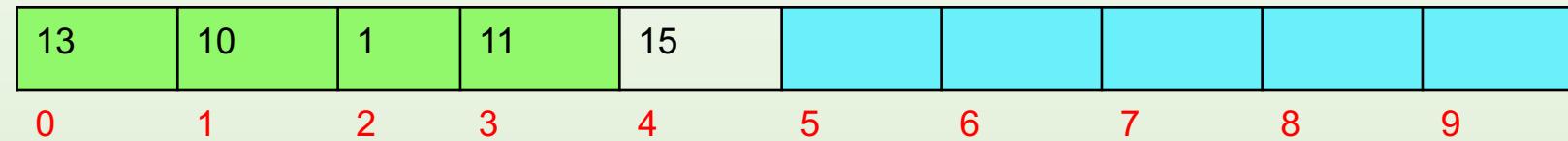
Primo step

- ✓ il processo termina quando si ha l'inversione degli indici $i > j$ e la conclusione del primo passo del QuickSort, Il vettore è così partizionato:



- ✓ L'algoritmo procede ricorsivamente operando sui vettori delimitati dagli indici (inf, \dots, j) e (i, \dots, sup)

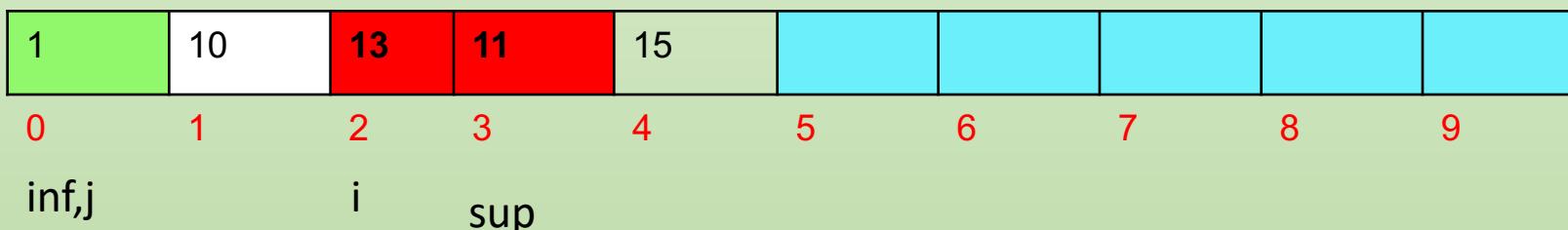
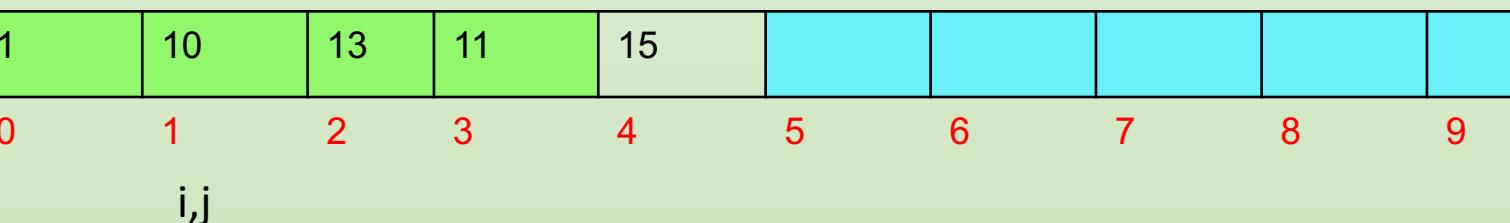
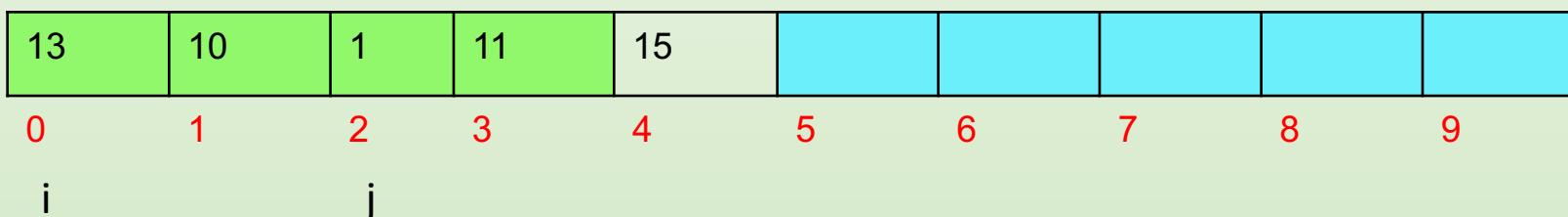
Secondo step



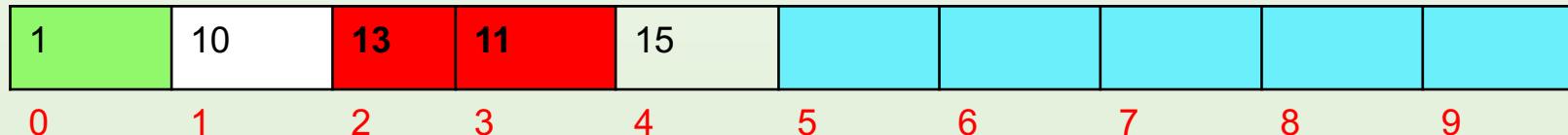
$i=inf$

$j=sup$

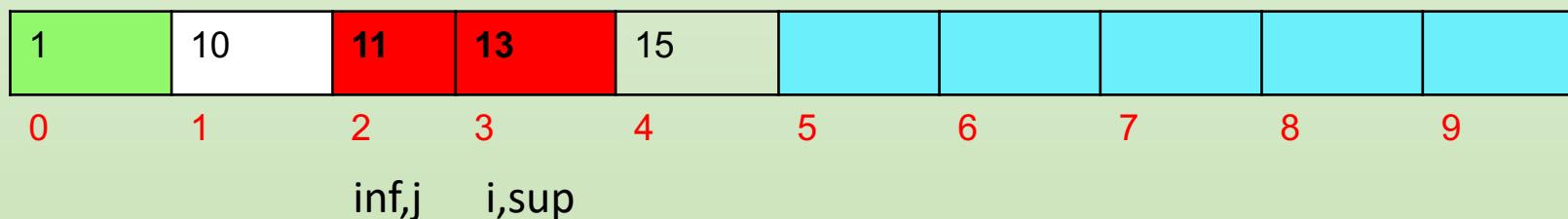
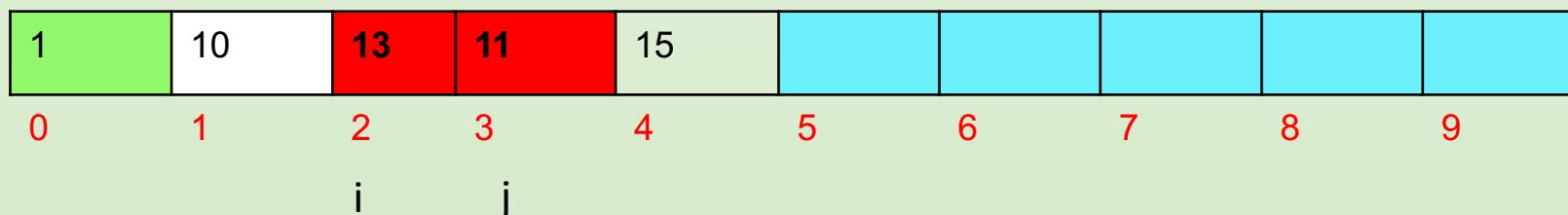
$pivot=10$



Stop cond



i=inf j=sup pivot=13



Fine ricorsione sul sottovettore di sinistra, si procede su quello di destra

Pseudocodice

QUICKSORT(A, p, r)

1. **pivot = FIND_PIVOT(A, p, r)**
2. se pivot \neq NULL allora
 - a. **q = PARTITION(A, p, r, pivot)**
 - b. **QUICKSORT(A, p, q)**
 - c. **QUICKSORT(A, q+1, r)**
3. fine-condizione

FIND_PIVOT(A, p, r)

1. per $k = p+1, \dots, r$ ripeti
 - a. se $a_k > a_p$ allora
 - restituisci a_k
 - b. altrimenti se $a_k < a_p$ allora
 - restituisci a_p
 - c. fine-condizione
2. fine-ciclo
3. restituisci NULL

PARTITION(A, p, r, pivot)

1. **i = p, j = r**
2. ripeti
 - a. fintanto che $a_j \geq \text{pivot}$ ripeti
 - $j = j - 1$
 - b. fine-ciclo
 - c. fintanto che $a_i < \text{pivot}$ ripeti
 - $i = i + 1$
 - d. fine-ciclo
 - e. se $i < j$ allora
 - scambia a_i e a_j
 - f. fine-condizione
3. fino a quando $i < j$
4. restituisci j

Divide et impera

- ✓ La riduzione mediante albero mostrata nella scan è un caso particolare di questa tecnica
- ✓ Si procede **ricorsivamente** come segue:
 1. Suddividi in sottoproblemi
 2. risovi il caso base
 3. Combina le soluzioni
- ✓ Particolarmente efficace negli algoritmi di **sorting**
- ✓ **Complessità:** $O(n \log_2 n)$

Algorithm *mergeSort*(S, C)

Input: sequence S with n elements,
comparator C

Output: sequence S sorted according
to C

if $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

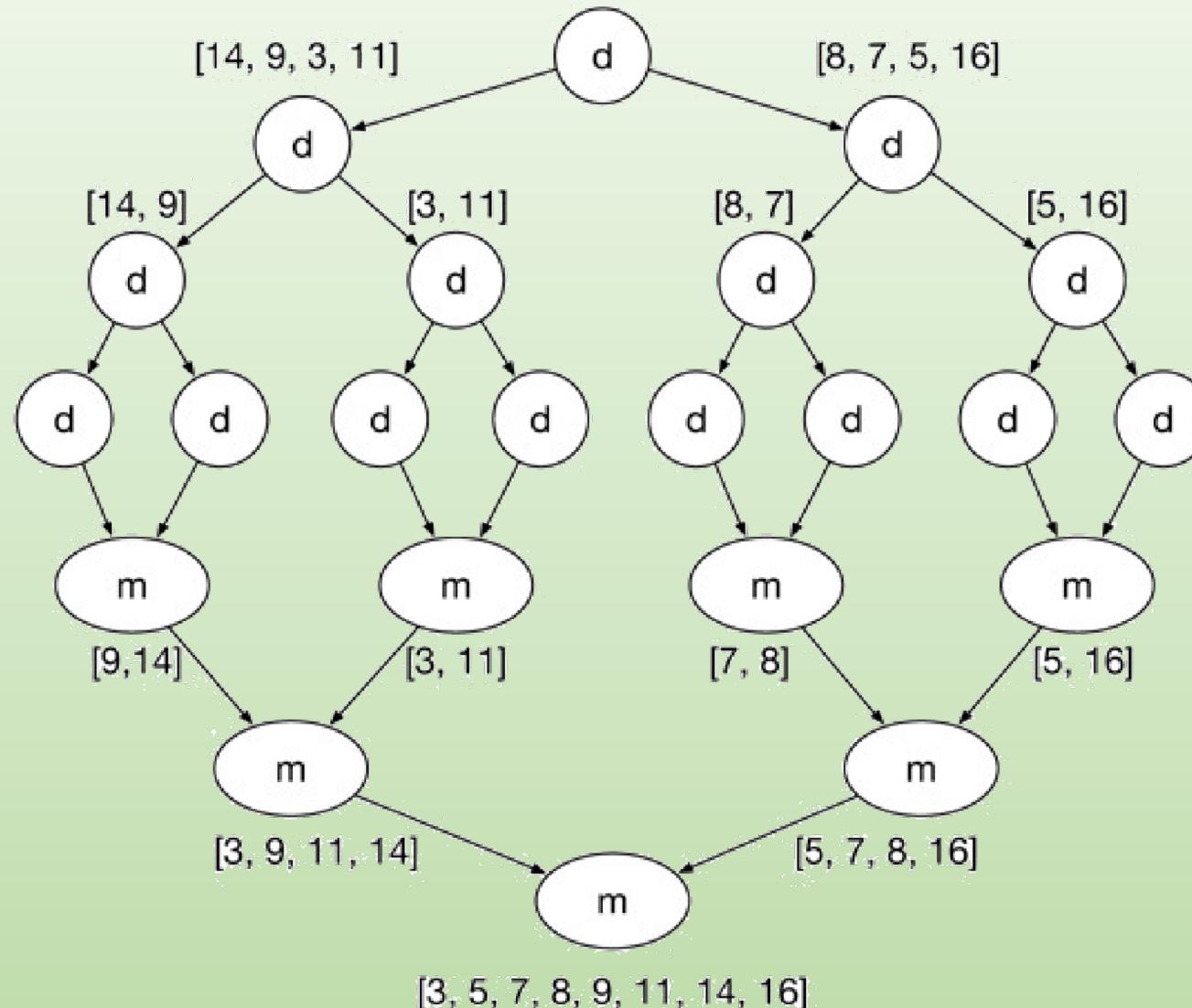
mergeSort(S_1, C)

mergeSort(S_2, C)

$S \leftarrow merge(S_1, S_2)$

Esempio: DG di merge sort

[14, 9, 3, 11, 8, 7, 5, 16]



Schema sequenziale

MergeSort(arr[], l, r)

If $r > l$

1. Divide the array into two halves:

middle $m = (l+r)/2$

2. Call mergeSort for first half:

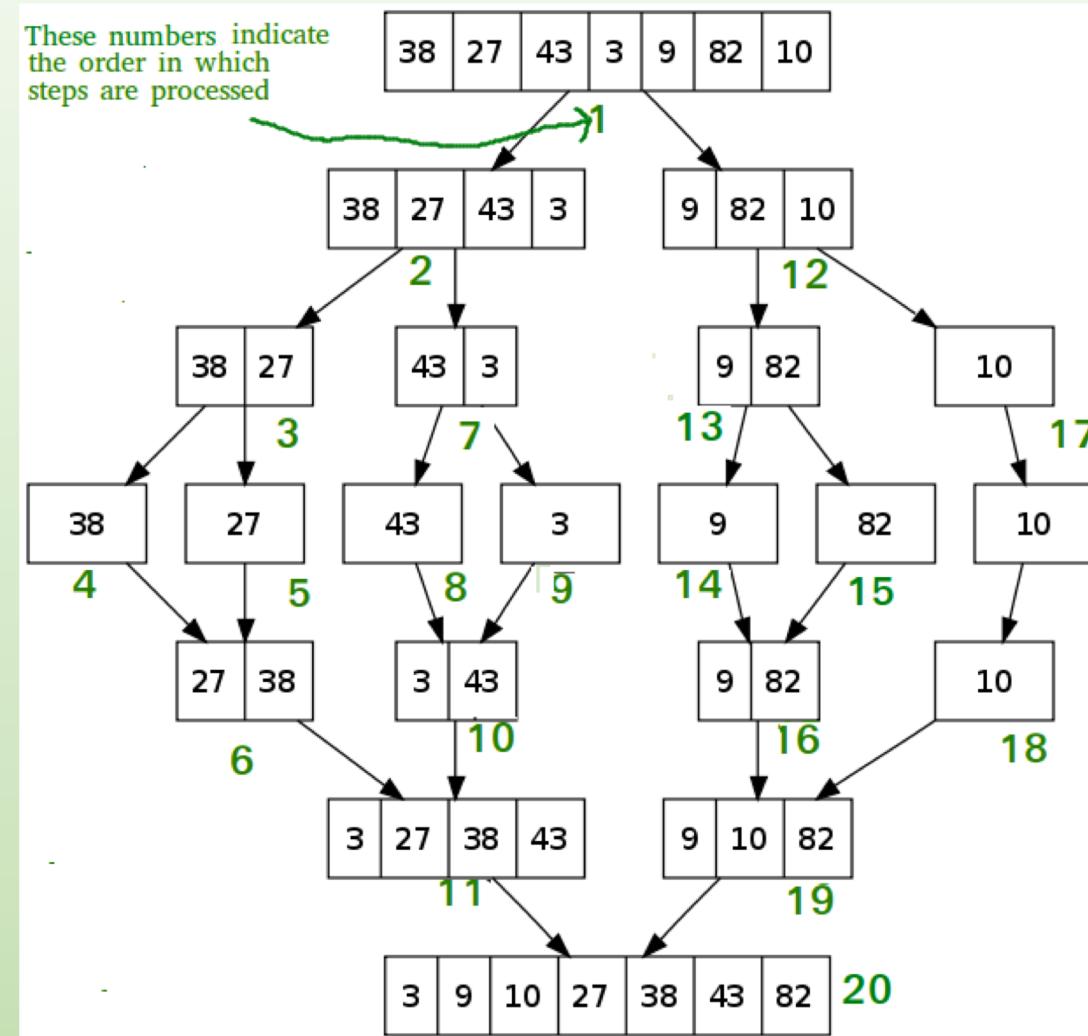
Call **mergeSort(arr, l, m)**

3. Call mergeSort for second half:

Call **mergeSort(arr, m+1, r)**

4. Merge the two halves sorted:

Call **merge(arr, l, m, r)**



Strategia parallela

Input:

8 3 1 9 1 2 7 5 9 3 6 4 2 0 2 5

Split:

8 3 1 9 - 1 2 7 5 - 9 3 6 4 - 2 0 2 5

Threads(4):

	t1		t2		t3		t4	
	8 3 1 9		1 2 7 5		9 3 6 4		2 0 2 5	
	3 8 1 9		1 2 5 7		3 9 4 6		0 2 2 5	

Merge:

	1 3 8 9		1 2 5 7		3 4 6 9		0 2 2 5	
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	

Threads(2):

	t1					t2				
	1 1 2 3 5 7 8 9					0 2 2 3 4 5 6 9				
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	

Threads(1):

	t1																
	0 1 1 2 2 2 3 3 3 4 5 5 6 7 8 9 9																
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	

Esercitazione

Merge sort:

Implementare l'algoritmo **mergesort** a partire da **quicksort**

Ordinamento bitonico

L'ordinamento **Bitonic Mergesort** è un algoritmo di ordinamento parallelo progettato da **Ken Batcher** basato su **sorting network** (rete di comparatori) con proprietà:

- Numero comparatori: $\mathcal{O}(N \log N)$
- Complessità in tempo: $\mathcal{O}(\log^2 N)$

Sequenza bitonica (Batcher 1968)

- Una **sequenza bitonica** è una sequenza con due “toni”, uno crescente e uno decrescente, cioè

data $s = \langle a_1, \dots, a_{n-1} \rangle$ esiste $0 \leq i \leq n - 1$

tale che $a_0 \leq a_1 \leq \dots \leq a_i$ e $a_{i+1} \geq a_{i+2} \geq \dots \geq a_{n-1}$

- **PROP.:** una **permutazione ciclica** di una sequenza bitonica è ancora bitonica

- Se la sequenza è tale che (n è un potenza di 2):

$$a_0 \leq a_1 \leq \dots \leq a_{n/2-1} \quad \& \quad a_{n/2} \leq a_{n/2+1} \leq \dots \leq a_{n-1}$$

- Allora le sequenze:

$$s_1 = \langle \min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\}, \dots, \min\{a_{n/2-1}, a_{n-1}\} \rangle$$

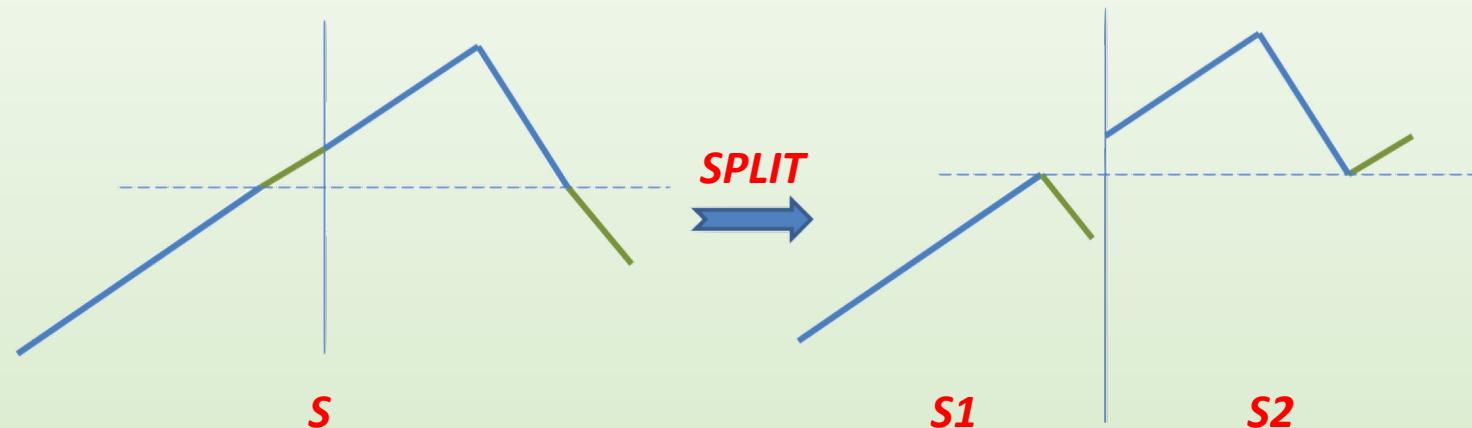
$$s_2 = \langle \max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\}, \dots, \max\{a_{n/2-1}, a_{n-1}\} \rangle$$

- Sono ancora sequenze bitoniche (**bitonic split**)!
- **IDEA:** applicare **ricorsivamente** questa procedura alla sequenza data per ordinarla!!

Split x ordinamento bitonico

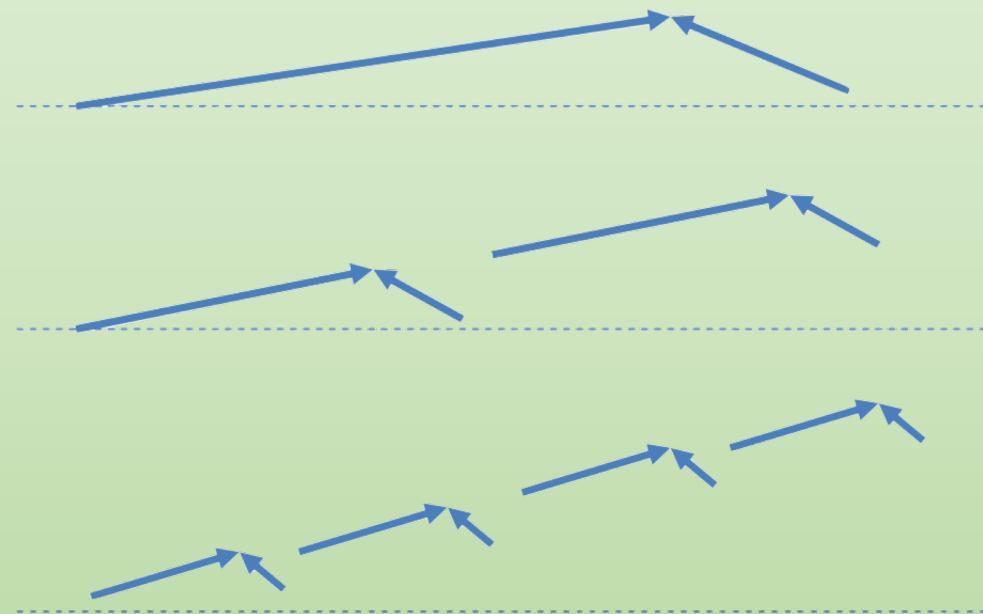
Proprietà:

le due sequenze **S1** e **S2** sono
bitoniche e gli elementi di **S1**
sono minori degli elementi di **S2**



Applicazione ricorsiva del teorema:

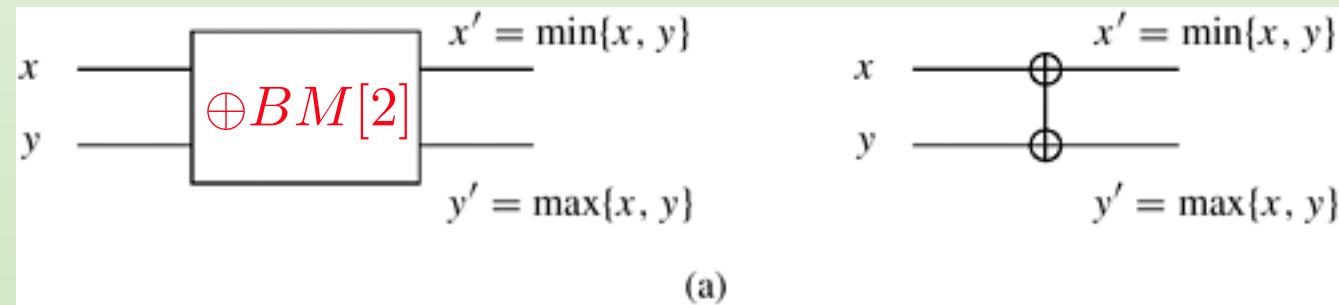
- Dopo **log n - 1** passi, ogni sequenza bitonica avrà solo due elementi...
- Facile da ordinare!!



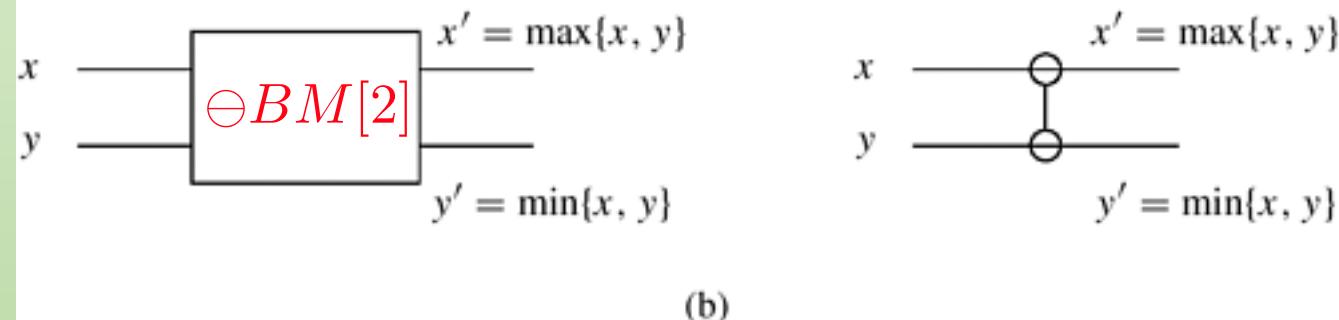
Bitonic merging networks

Bitonic merging network con n input:

- Produce una sequenza **crescente** con il comparatore $\oplus BM[n]$
- Produce una sequenza **decrescente** con il comparatore $\ominus BM[n]$

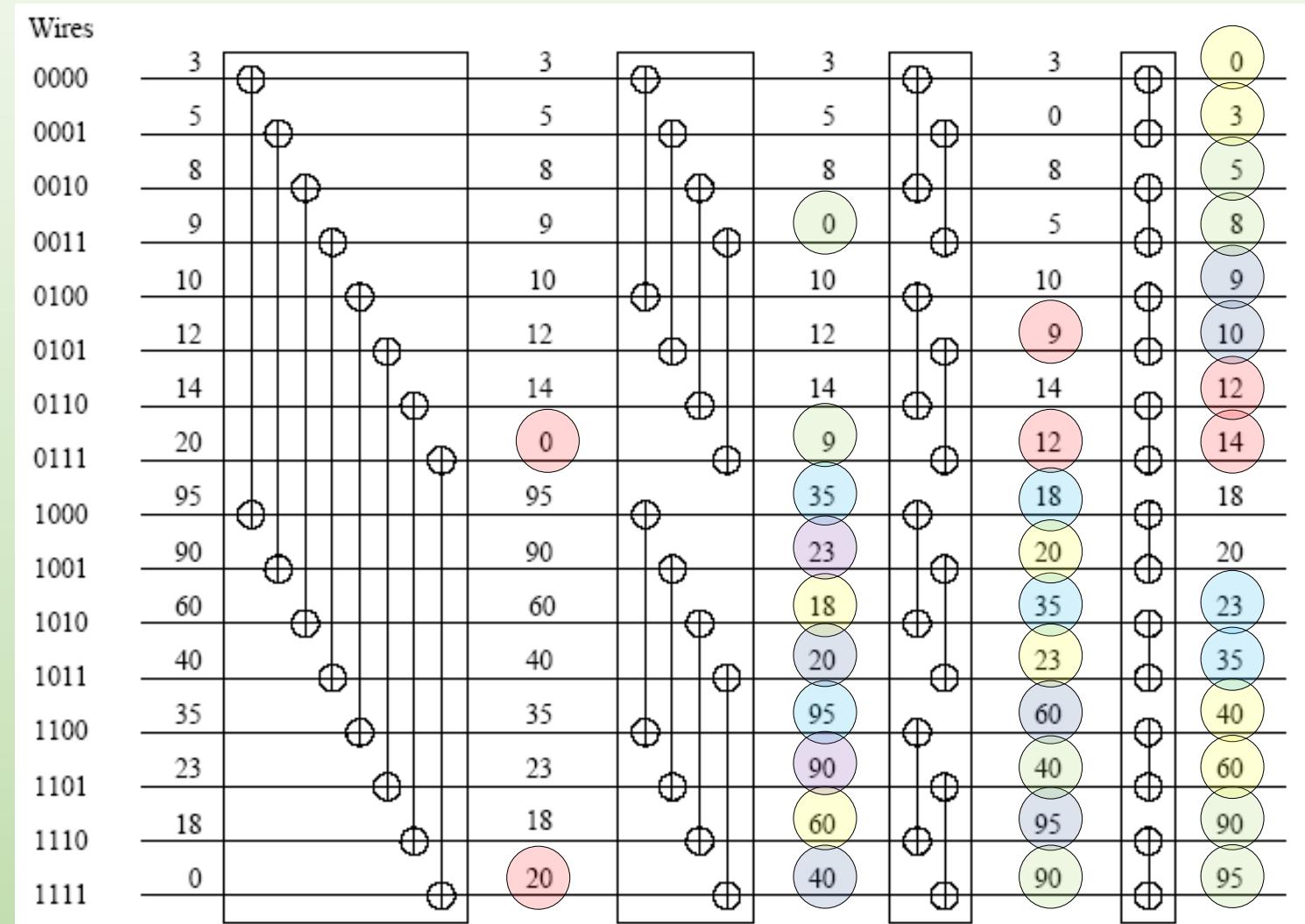


Comparatori $n = 2$



Bitonic merging network (n=16)

- ✓ Esempio di una rete
 $\oplus BM[16]$
- ✓ Struttura rete con **n** fili:
 - $\log_2 n$ colonne
 - $n/2$ comparatori x colonna
- ✓ Ogni colonna è composta da **comparatori** con un numero di fili pari alla metà del livello precedente
- ✓ La rete prende in ingresso una sequenza **bitonica** e produce in output una sequenza **ordinata**



Ordinamento bitonico

Due passi:

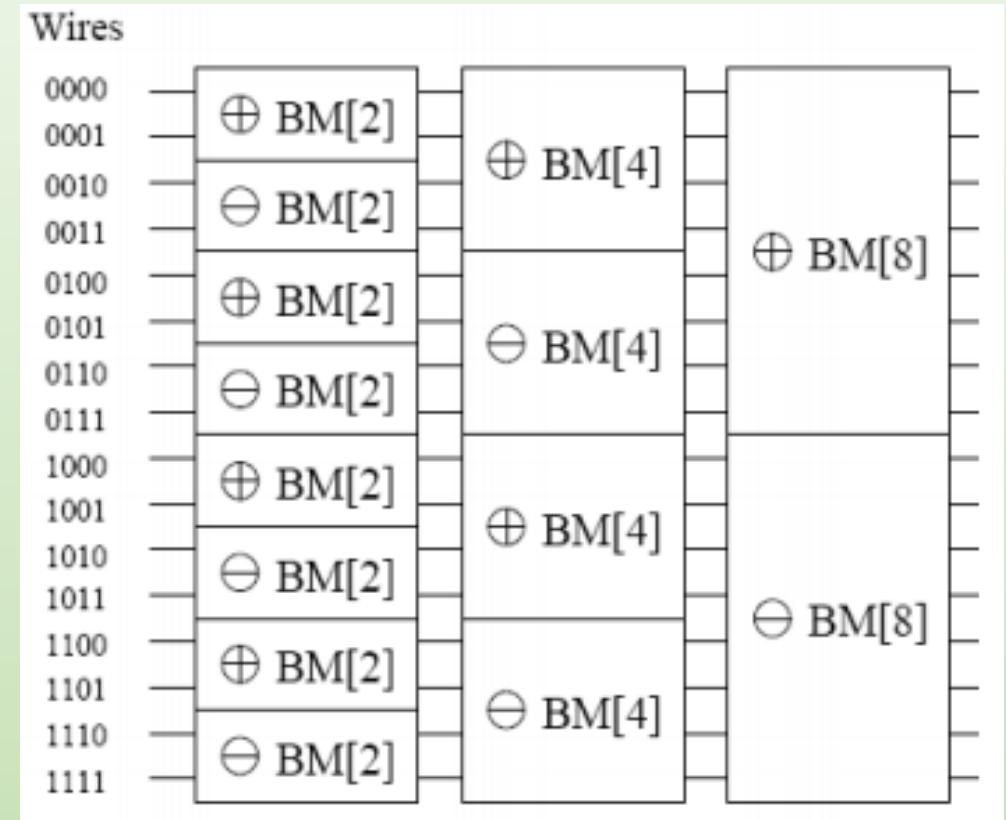
1. Costruisci una **sequenza bitonica**
2. Ordina usando una **bitonic merging network**

Costruire una sequenza bitonica da una data sequenza:

- ✓ Ogni sequenza di lunghezza 2 è una sequenza bitonica
- ✓ Per costruire una sequenza bitonica di lunghezza 4
 - Ordina i primi due elementi con $\ominus BM[2]$
 - Ordina i due successivi con $\oplus BM[2]$
- ✓ Per costruire sequenze bitoniche di lunghezza arbit.
 - ✓ Unire comparatori di dimensione crescente k

$\ominus BM[k]$

$\oplus BM[k]$



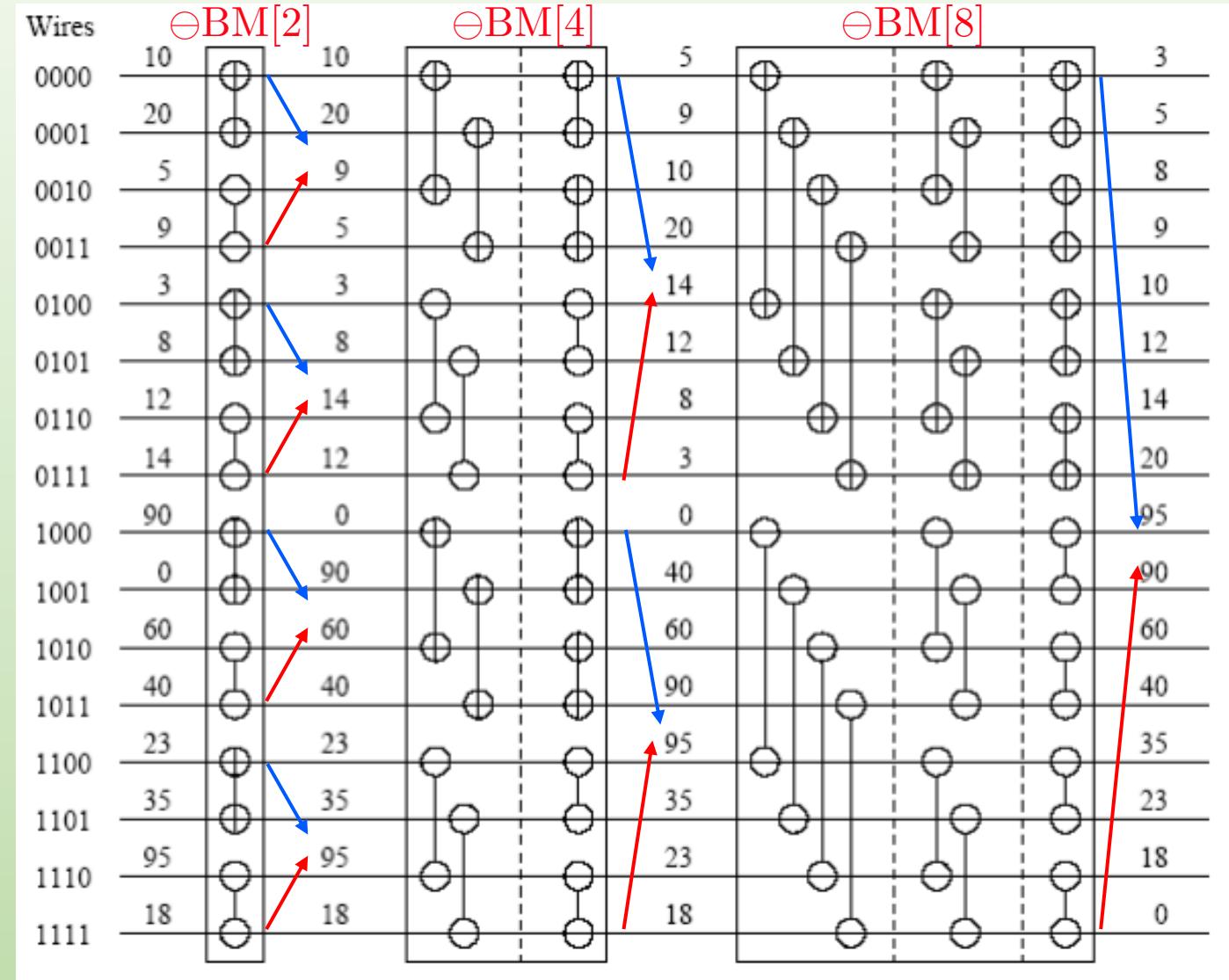
Costruzione di sequenza bitonica

Input:

- Sequenza arbitraria

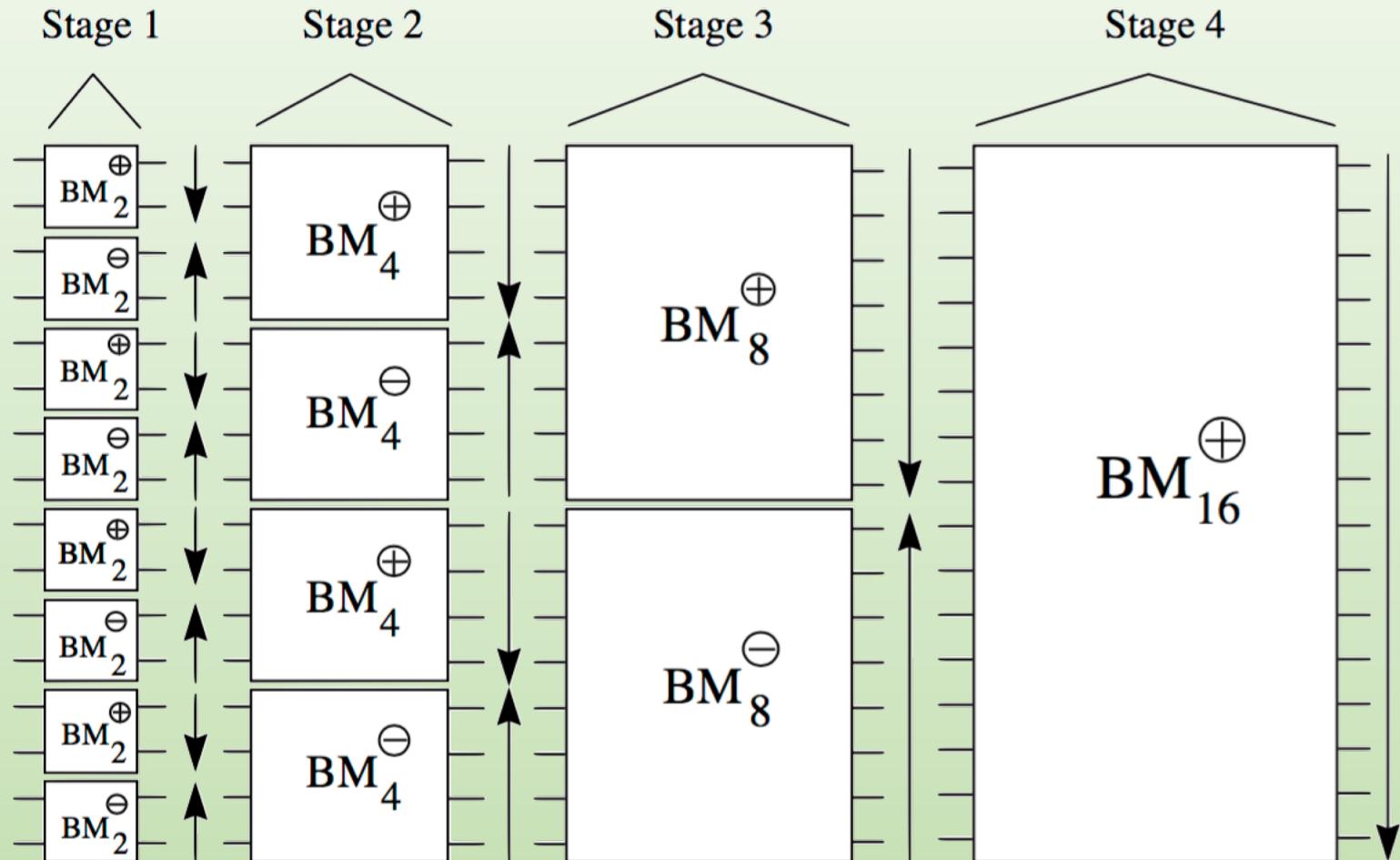
Output:

- Sequenza bitonica



Ordinamento bitonico ($n = 16$)

- ✓ I primi 3 stage costruiscono una sequenza bitonica
- ✓ L'ultimo ordina la sequenza con il comparatore
- ✓ Numero comparatori:
 $BM2 = 1, BM4 = 4,$
 $BM8 = 12, BM16 = 32$
- ✓ Complessità: $8 \times 1 + 4 \times 4 + 12 \times 2 + 32 \times 1 = 80$



Simulazione: bitonic merge sort example

- ✓ Repeatedly build bitonic lists and then sort them
 - ✓ Bitonic list is two monotonic lists concatenated together, one increasing and one decreasing.
 - ✓ List A: (3, 4, 7, 8) monotonically increasing
 - ✓ List B: (6, 5, 2, 1) monotonically decreasing
 - ✓ List AB: (3, 4, 7, 8, 6, 5, 2, 1) bitonic

from paper GPUterasort: High Performance Graphics Co-processor Sorting for Large Database Management.
Govindaraju, Naga K. et. al.

BITONIC MERGE SORT



8x monotonic lists: (3) (7) (4) (8) (6) (2) (1) (5)
4x bitonic lists: (3,7) (4,8) (6,2) (1,5)

BITONIC MERGE SORT



Sort the bitonic lists

BITONIC MERGE SORT



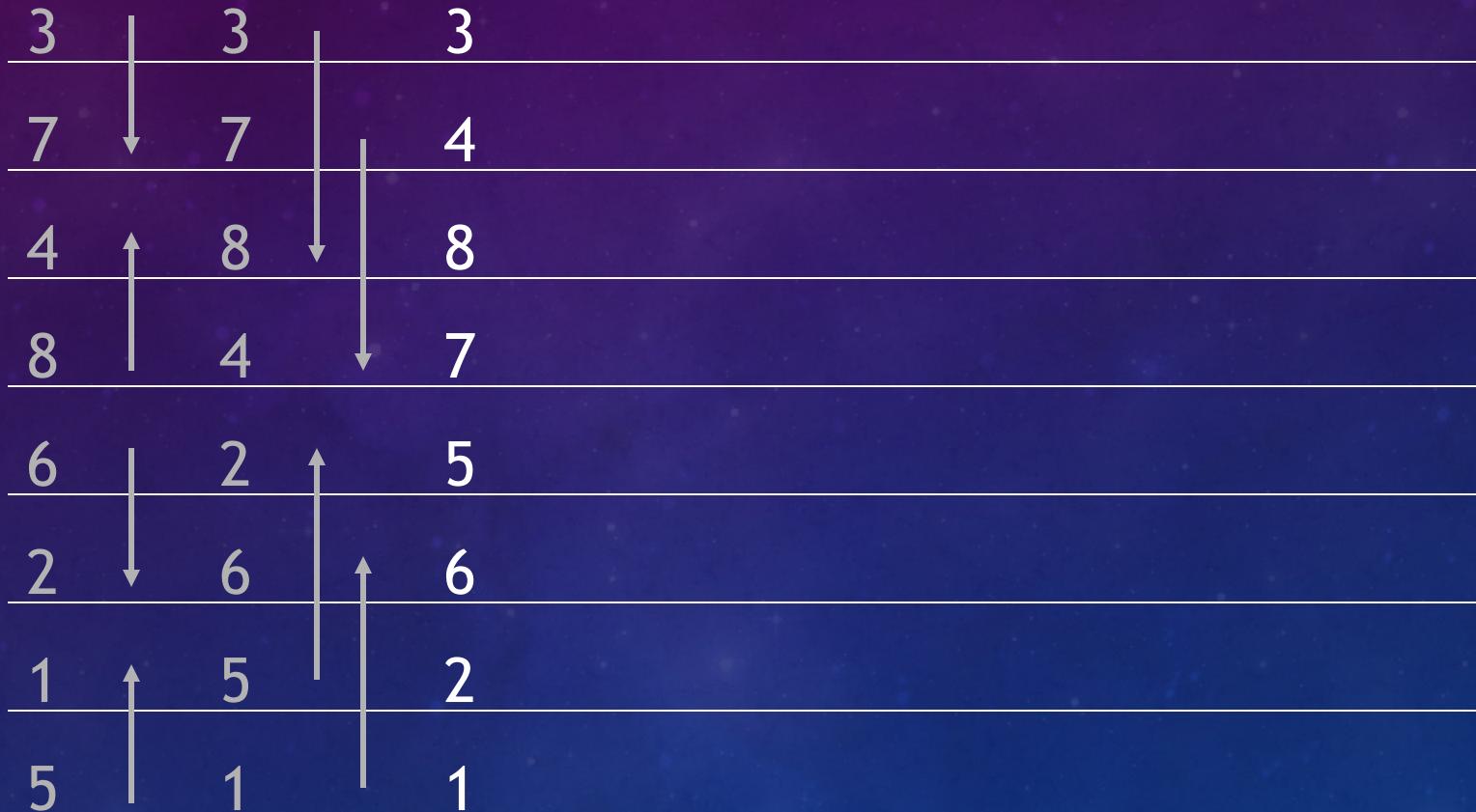
4x monotonic lists: (3,7) (8,4) (2,6) (5,1)
2x bitonic lists: (3,7,8,4) (2,6,5,1)

BITONIC MERGE SORT



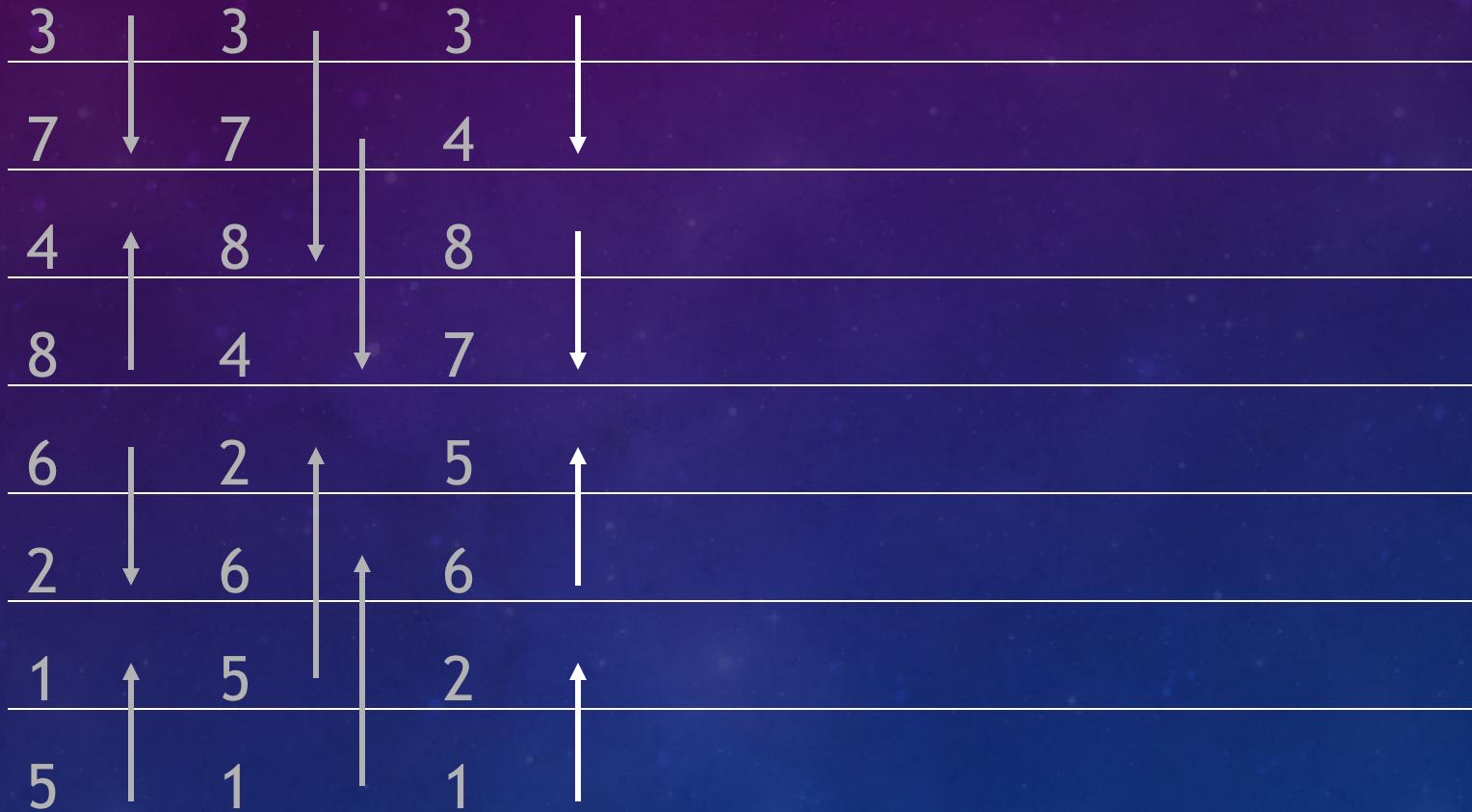
Sort the bitonic lists

BITONIC MERGE SORT



Sort the bitonic lists

BITONIC MERGE SORT



Sort the bitonic lists

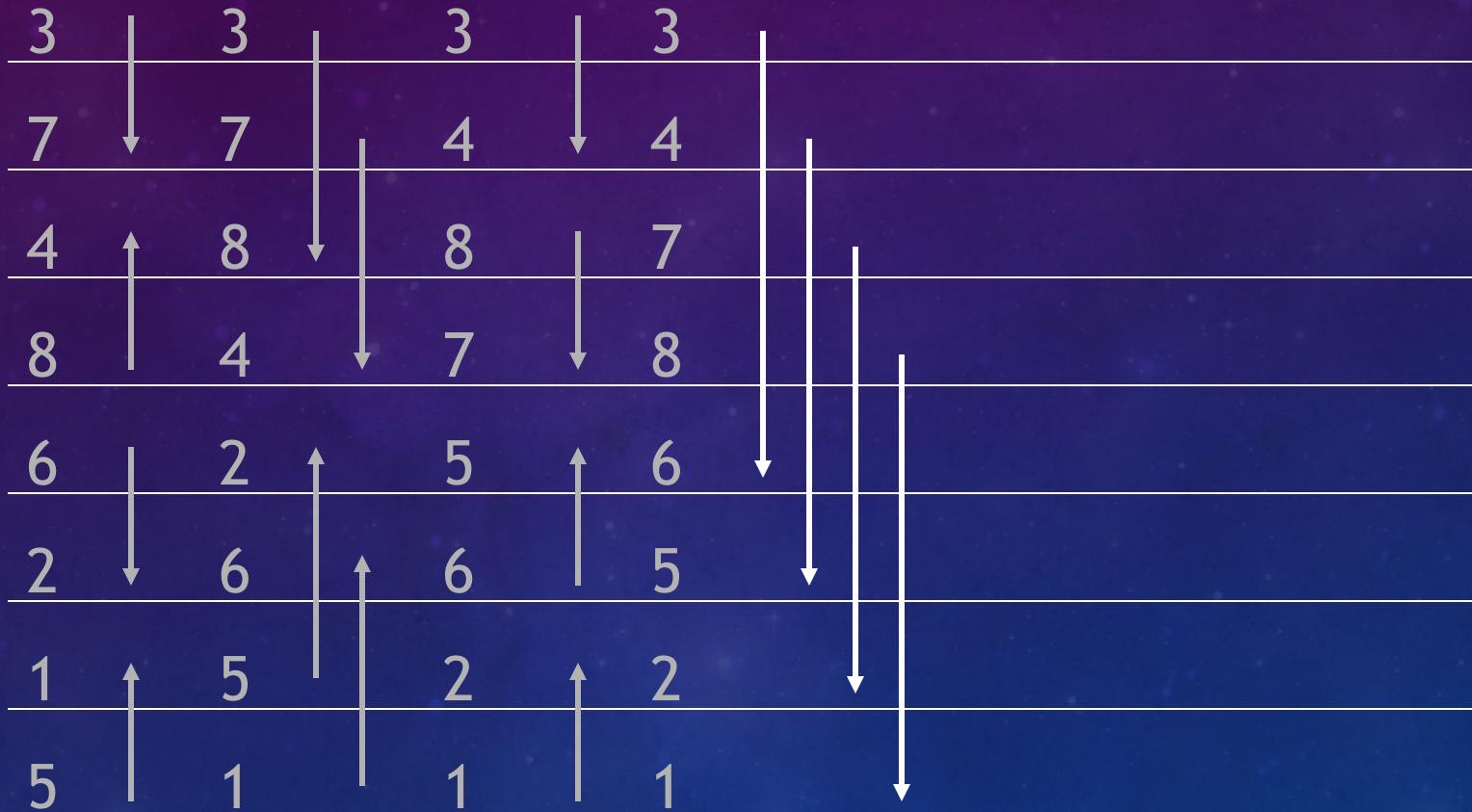
BITONIC MERGE SORT



2x monotonic lists: (3,4,7,8) (6,5,2,1)

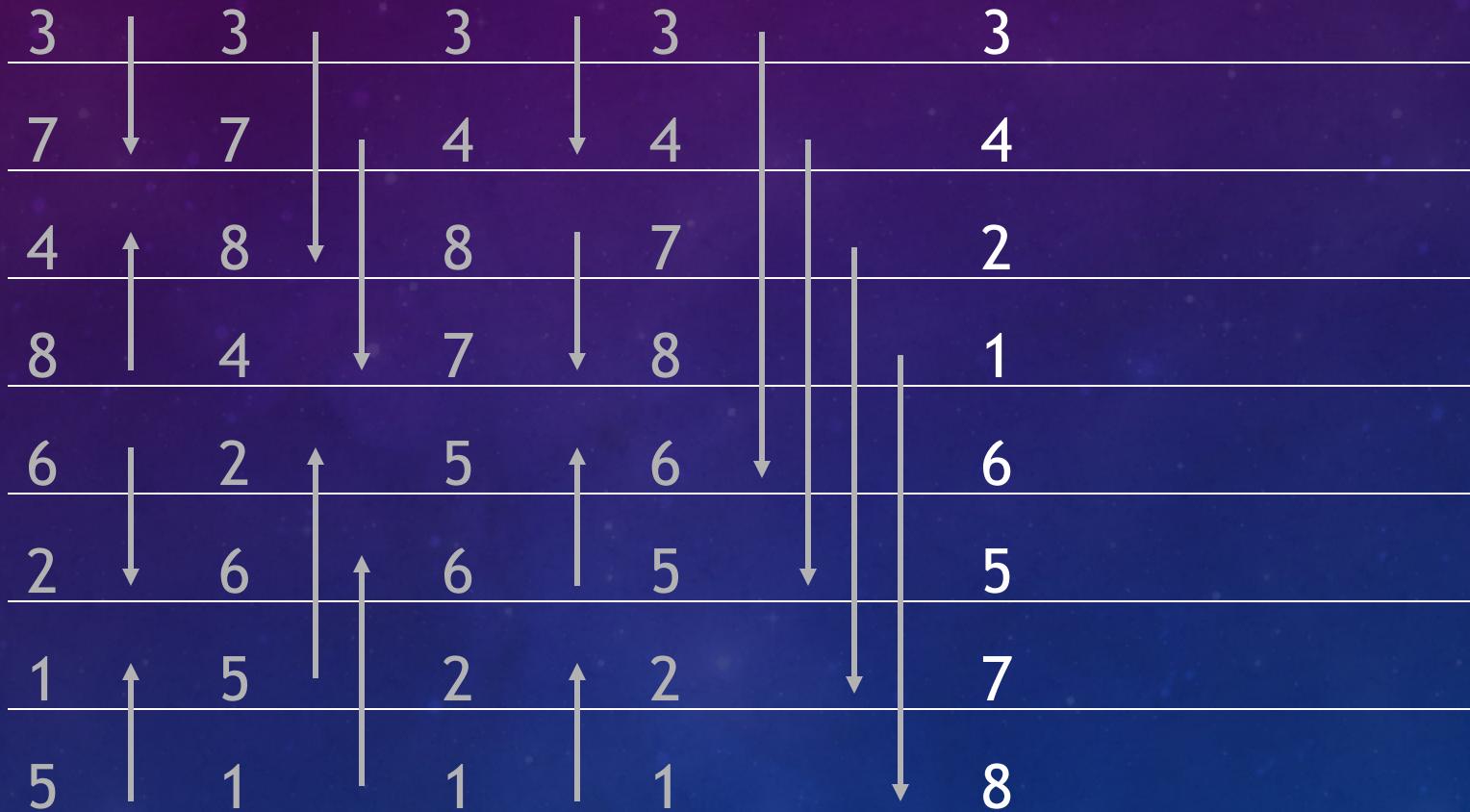
1x bitonic list: (3,4,7,8, 6,5,2,1)

BITONIC MERGE SORT



Sort the bitonic list

BITONIC MERGE SORT



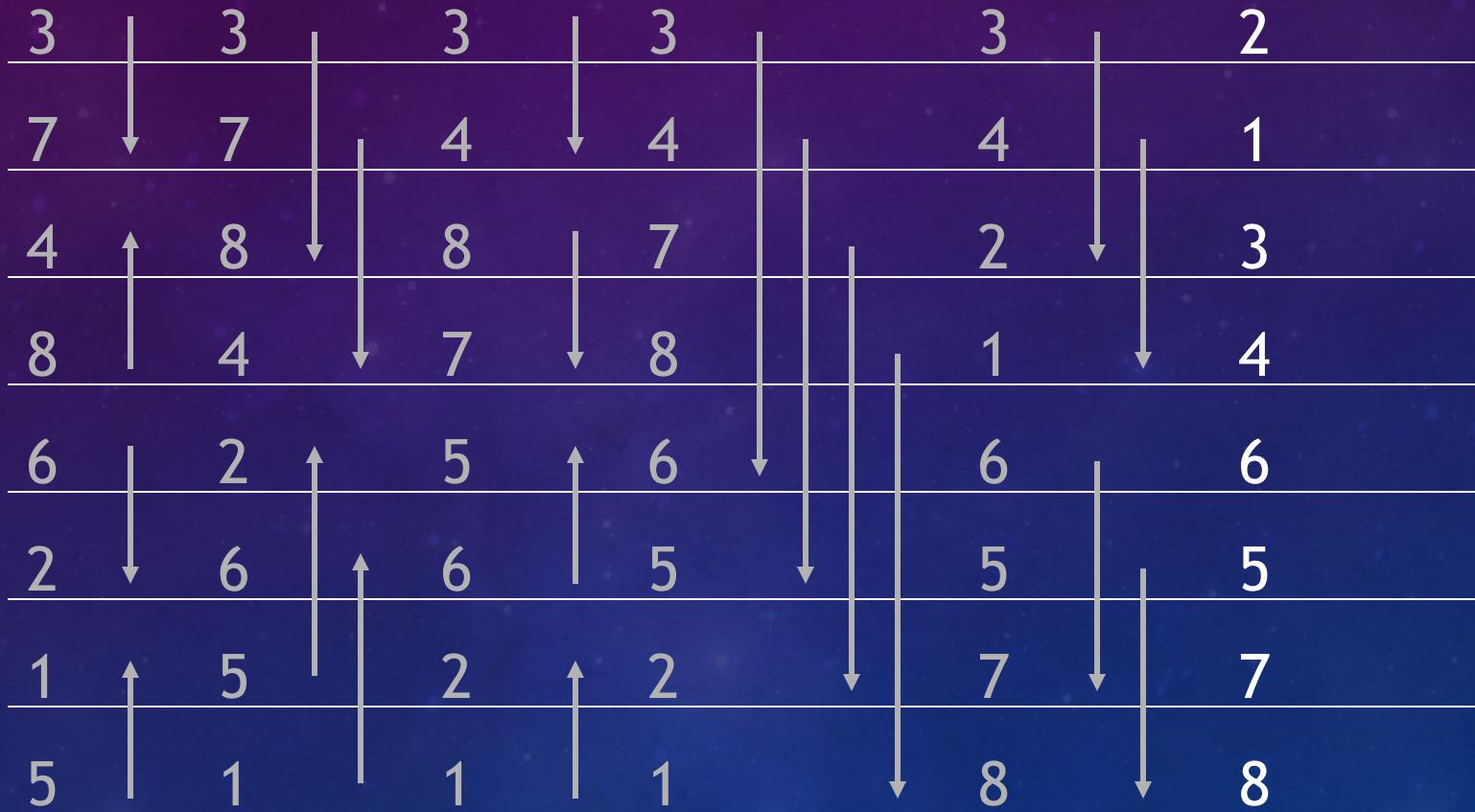
Sort the bitonic list

BITONIC MERGE SORT



Sort the bitonic list

BITONIC MERGE SORT



Sort the bitonic list

BITONIC MERGE SORT



Sort the bitonic list

BITONIC MERGE SORT



Done!

Complessità di Bitonic sort parallelo

Parallel bitonic sort with n processor

- ✓ n unsorted numbers
- ✓ $n/2$ group of **2-number** bitonic sequence
- ✓ $n/4$ group of **4-number** bitonic sequence
- ✓ ...
- ✓ **1** group of **n -number** bitonic sequence
- ✓ The last stage of an n -element bitonic sorting need to merge n -element, and has a depth of **$\log(n)$**
- ✓ Other stages perform a complete sort of $n/2$, $n/4$, $n/8, \dots, 2$ elements
- ✓ Depth, $d(n) = d(n/2) + \log(n)$
- ✓ $d(n) = 1 + 2 + 4 + \dots + \log(n) = \Theta(\log^2 n)$
- ✓ **Complexity: $T(n) = \Theta(\log^2 n)$**

Esercitazione

Ordinamento bitonico:

Sviluppare l'algoritmo per sequenza bitonica tale che:

- L'ordinamento si possa effettuare in memoria SMEM
- Pensare a come estenderlo a dimensioni dei dati arbitraria

```
/*
It recursively sorts a bitonic sequence in ascending order,
if dir = 1, and in descending order otherwise (means dir=0).
The sequence to be sorted starts at index position low, the
parameter cnt is the number of elements to be sorted.
*/
void bitonicMerge(int a[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        for (int i = low; i < low + k; i++)
            compAndSwap(a, i, i + k, dir);
        bitonicMerge(a, low, k, dir);
        bitonicMerge(a, low + k, k, dir);
    }
}
```

```
/*
This function first produces a bitonic sequence by
recursively sorting its two halves in opposite sorting
orders, and then calls bitonicMerge to make them in the
same order
*/
void bitonicSort(int a[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        // sort in ascending order since dir here is 1
        bitonicSort(a, low, k, 1);
        // sort in descending order since dir here is 0
        bitonicSort(a, low + k, k, 0);
        // Will merge whole sequence in dir order
        bitonicMerge(a, low, cnt, dir);
    }
}
```

```
/*
The parameter dir indicates the sorting direction,
ASCENDING or DESCENDING; if (a[i] > a[j]) agrees with
the direction, then a[i] and a[j] are interchanged.
*/
void compAndSwap(int a[], int i, int j, int dir) {
    if (dir == (a[i] > a[j])) {
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
}
```