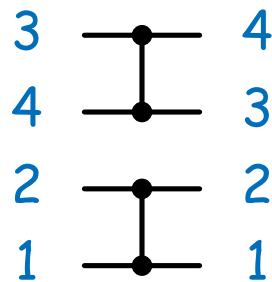


Lecture 32: Parallelism

- Moore's law ("Transistors per chip doubles every N years"), where N is roughly 2 (about $1,000,000\times$ increase since 1971).
- Has also applied to processor speeds (doubling a bit faster).
- But predicted to flatten: further increases to be obtained through *parallel processing* (witness: multicore/manycore processors).
- With distributed processing, issues involve interfaces, reliability, communication issues.
- With other parallel computing, where the aim is performance, issues involve synchronization, balancing loads among processors, and, yes, "data choreography" and communication costs.

Example of Parallelism: Sorting

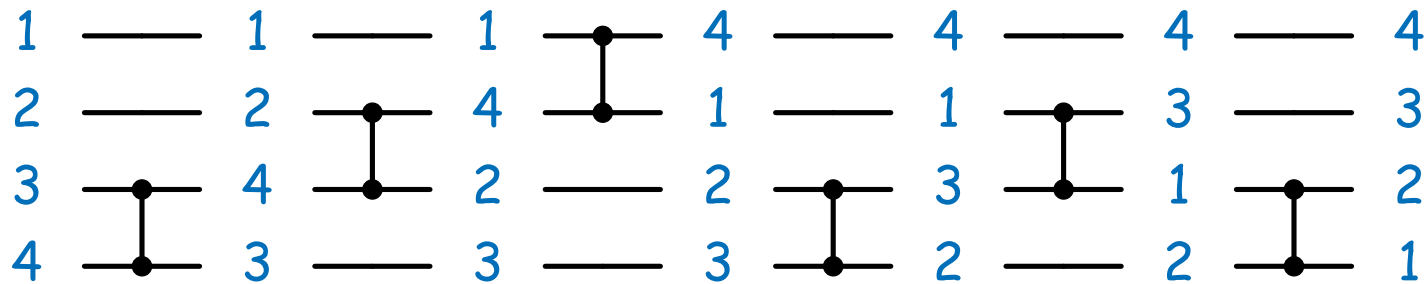
- Sorting a list presents obvious opportunities for parallelization.
- Can illustrate various methods diagrammatically using *comparators* as an elementary unit:



- Each vertical bar represents a *comparator*—a comparison operation or hardware to carry it out—and each horizontal line carries a data item from the list.
- A comparator compares two data items coming from the left, swapping them if the lower one is larger than the upper one.
- Comparators can be grouped into operations that may happen simultaneously; they are always grouped if stacked vertically as in the diagram.

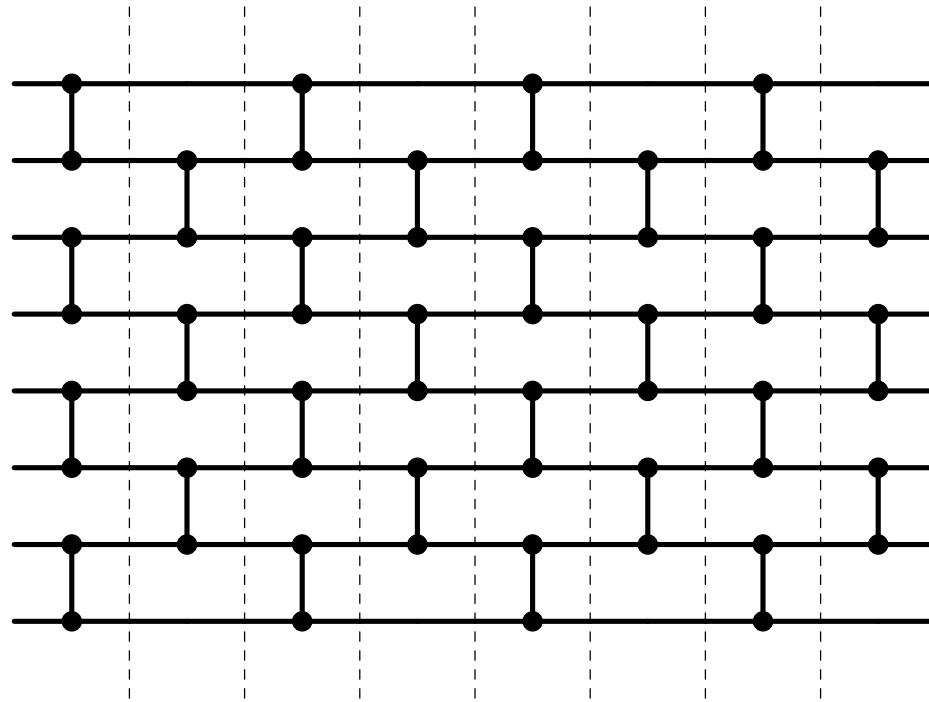
Sequential sorting

- Here's what a sequential sort (selection sort) might look like:



- Each comparator is a separate operation in time.
- In general, there will be $\Theta(N^2)$ steps.
- But since some comparators operate on distinct data, we ought to be able to overlap operations.

Odd-Even Transposition Sorter

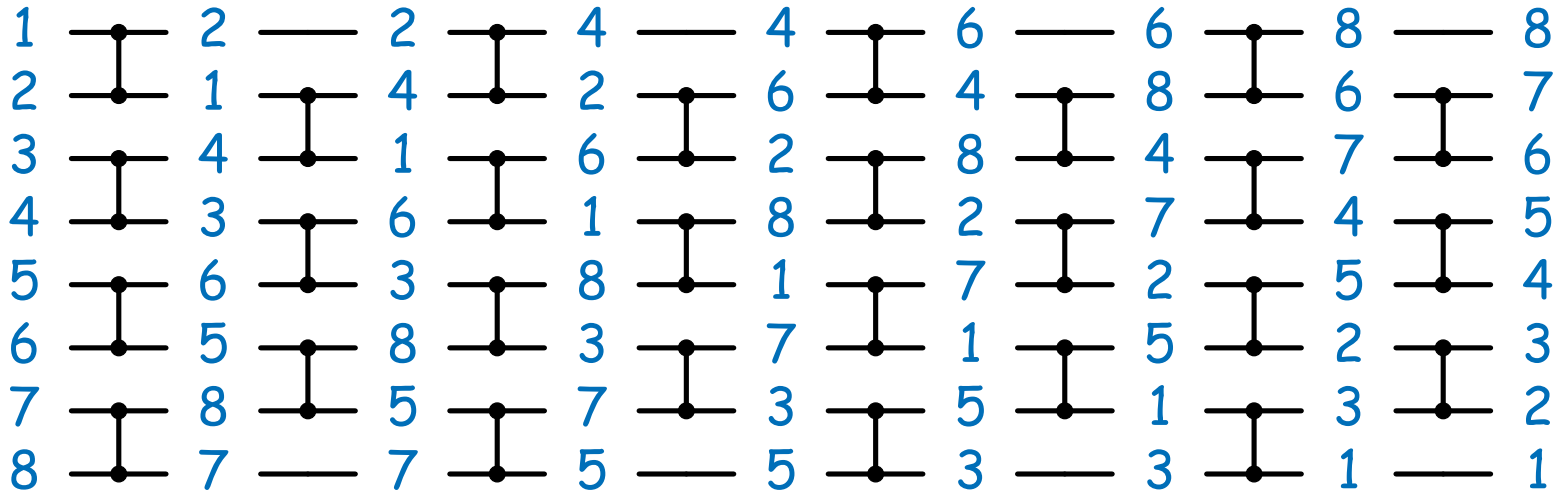


———— Data

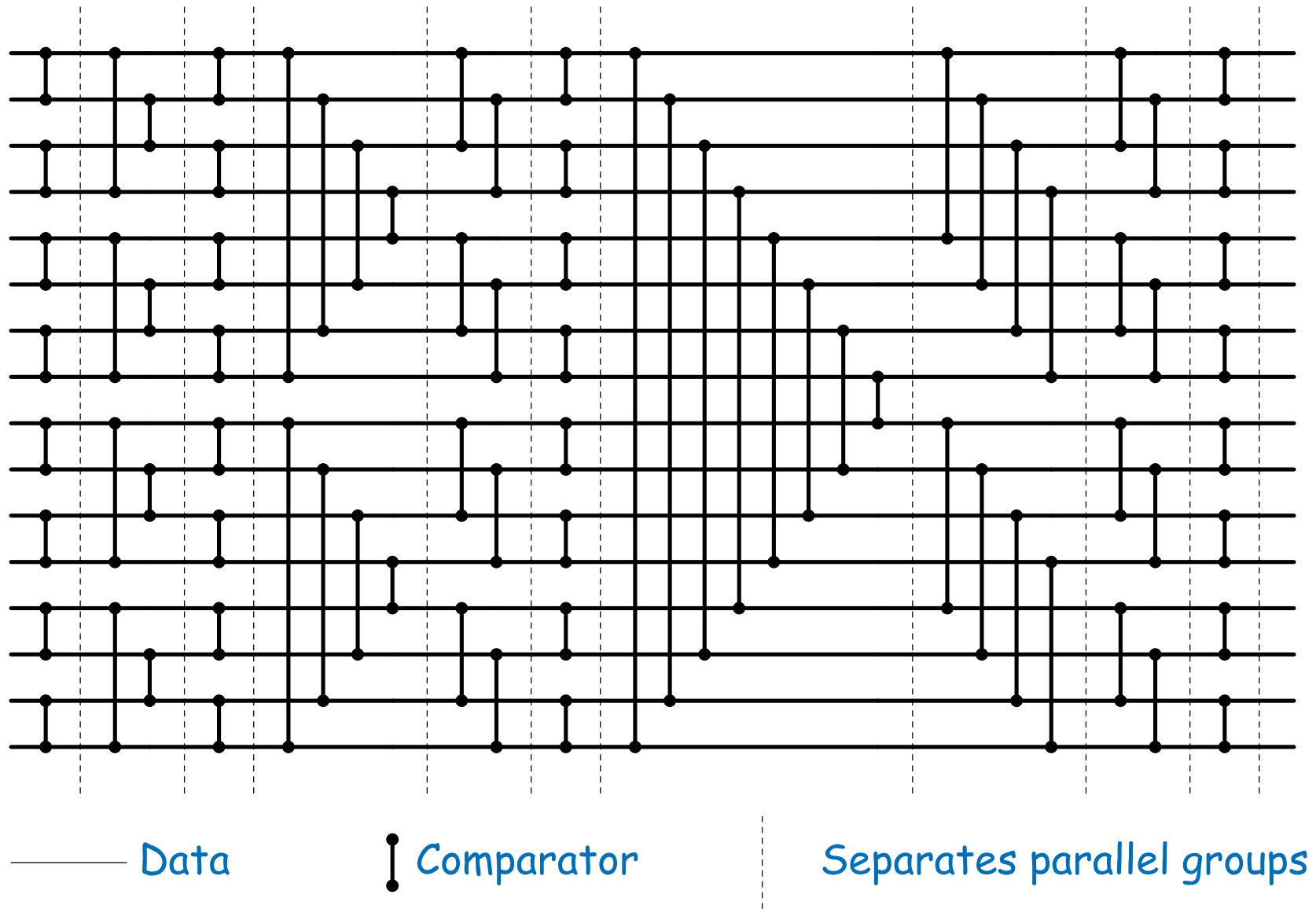
⋮ Comparator

⋮ Separates parallel groups

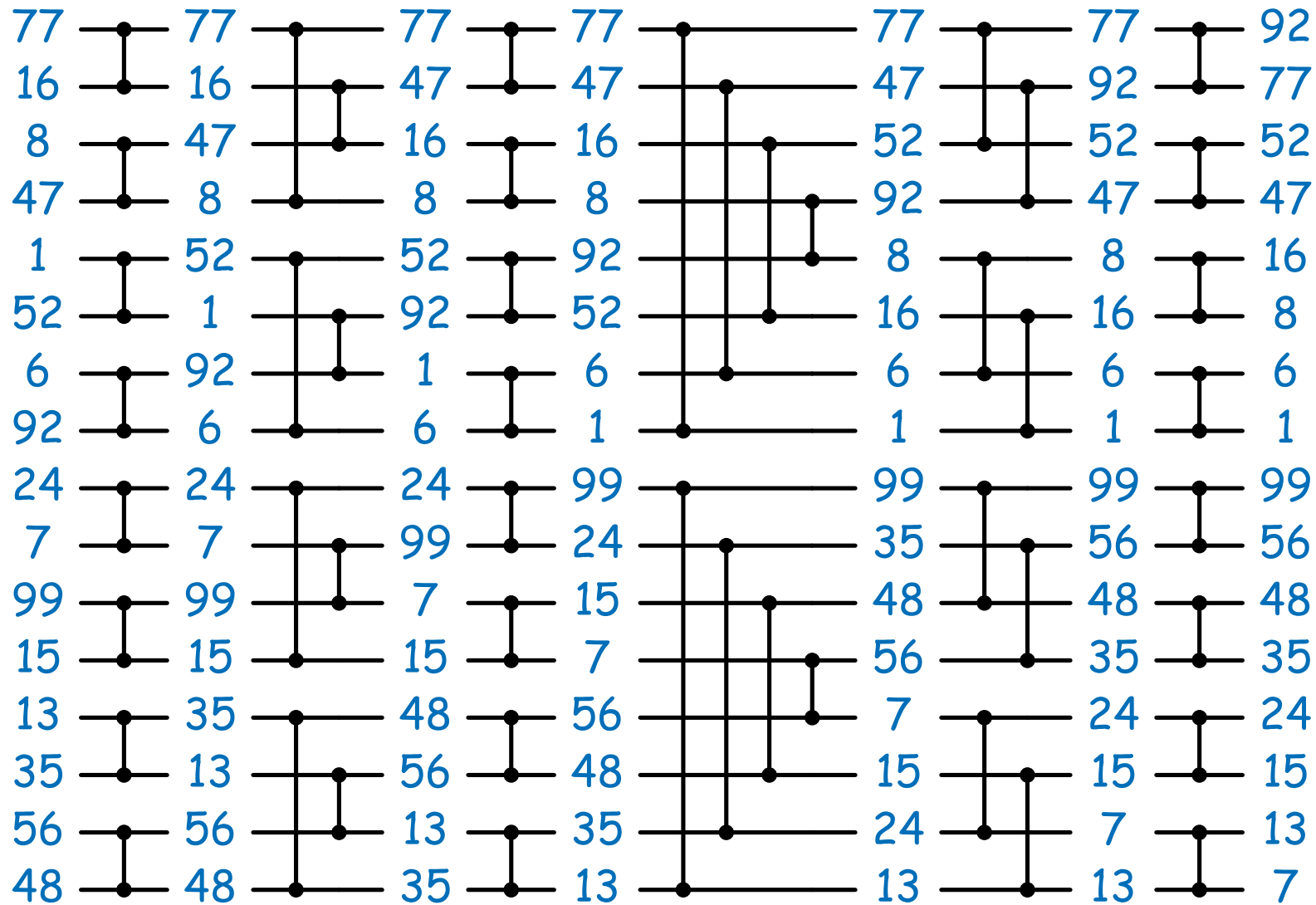
Odd-Even Sort Example



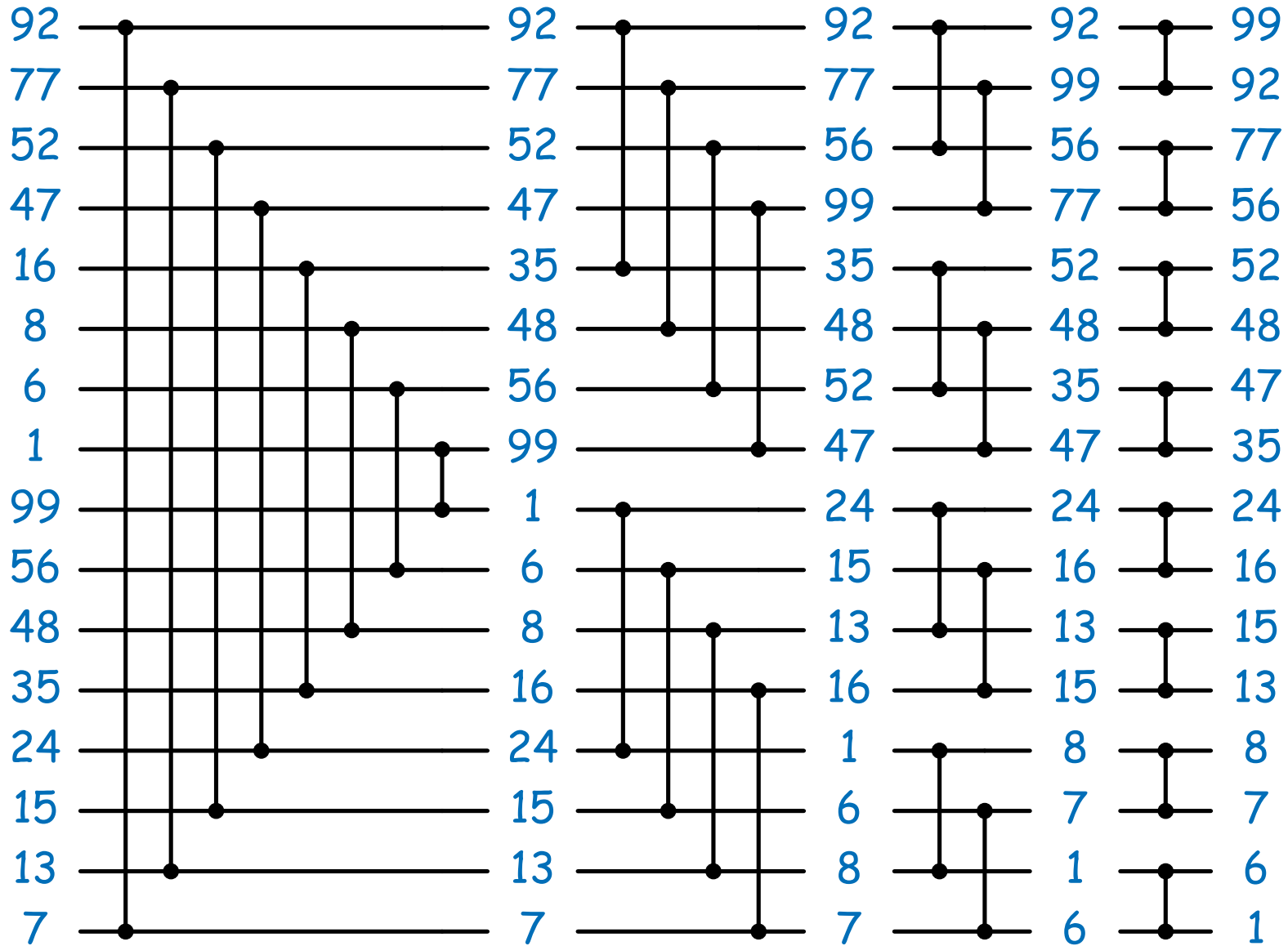
Example: Bitonic Sorter



Bitonic Sort Example (I)



Bitonic Sort Example (II)



Implementing Parallel Programs

- The sorting diagrams were abstractions.
- Comparators could be processors, or they could be operations divided up among one or more processors.
- Coordinating all of this is the issue.
- One approach is to use *shared memory*, where multiple processors (logical or physical) share one memory.
- This introduces conflicts in the form of *race conditions*: processors racing to access data.

Memory Conflicts: Abstracting the Essentials

- When considering problems relating to shared-memory conflicts, it is useful to look at the primitive read-to-memory and write-to-memory operations.
- E.g., the program statements on the left cause the actions on the right.

```
x = 5  
x = square(x)
```

```
WRITE 5 -> x  
READ x -> 5  
(calculate 5*5 -> 25)  
WRITE 25 -> x  
WRITE 6 -> y  
READ y -> 6  
(calculate 6+1 -> 7)  
WRITE 7 -> y
```

```
y = 6  
y += 1
```

Conflict-Free Computation

- Suppose we divide this program into two separate processes, P1 and P2:

```
x = 5  
x = square(x)
```

```
y = 6  
y += 1
```

P1

```
WRITE 5 -> x  
READ x -> 5  
(calculate 5*5 -> 25)  
WRITE 25 -> x
```

P2

```
WRITE 6 -> y  
READ y -> 6  
(calculate 6+1 -> 7)  
WRITE 7 -> y
```

```
x = 25  
y = 7
```

- The result will be the same regardless of which process's READs and WRITEs happen first, because they reference different variables.

Read-Write Conflicts

- Suppose that both processes read from x after it is initialized.

$x = 5$	
$x = \text{square}(x)$	$y = x + 1$
P1	P2
READ $x \rightarrow 5$ (calculate $5*5 \rightarrow 25$) WRITE $25 \rightarrow x$ 	 READ $x \rightarrow 5$ (calculate $5+1 \rightarrow 6$) WRITE $6 \rightarrow y$
$x = 25$ $y = 6$	

- The statements in P2 must appear in the given order, but they need not line up like this with statements in P1, because the execution of P1 and P2 is independent.

Read-Write Conflicts (II)

- Here's another possible sequence of events

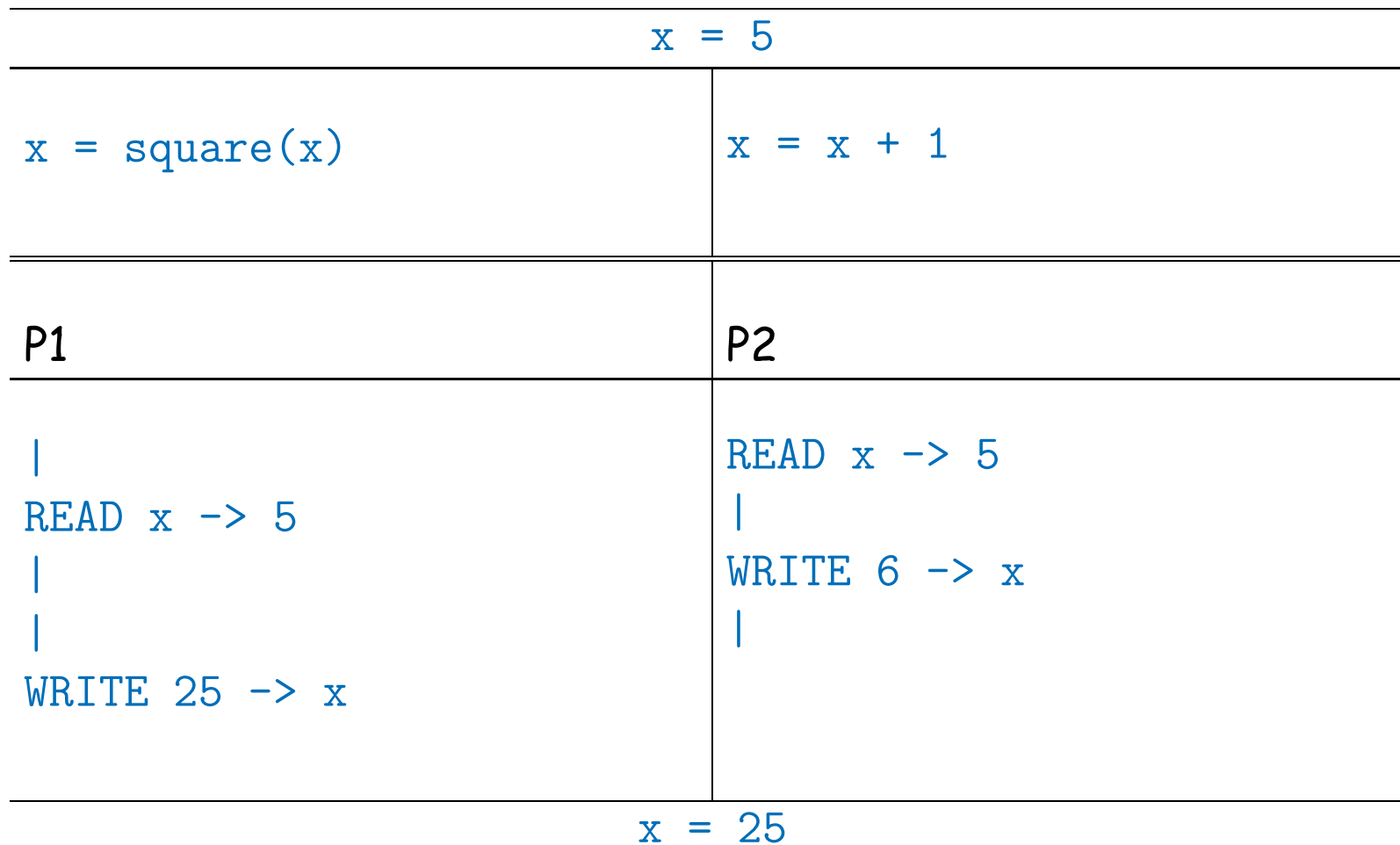
$x = 5$	
$x = \text{square}(x)$	$y = x + 1$
P1	P2
READ $x \rightarrow 5$ (calculate $5*5 \rightarrow 25$) WRITE $25 \rightarrow x$ 	 READ $x \rightarrow 25$ (calculate $25+1 \rightarrow 26$) WRITE $26 \rightarrow y$
$x = 25$ $y = 26$	

Read-Write Conflicts (III)

- The problem here is that nothing forces P1 to wait for P1 to read x before setting it.
- Observation: The "calculate" lines have no effect on the outcome. They represent actions that are entirely local to one processor.
- The effect of "computation" is simply to delay one processor.
- But processors are assumed to be delayable by many factors, such as time-slicing (handing a processor over to another user's task), or processor speed.
- So the effect of computation adds nothing new to our simple model of shared-memory contention that isn't already covered by allowing any statement in one process to get delayed by any amount.
- So we'll just look at READ and WRITE in the future.

Write-Write Conflicts

- Suppose both processes write to x :



- This is a *write-write conflict*: two processes race to be the one that “gets the last word” on the value of x .

Write-Write Conflicts (II)

$x = 5$	
$x = \text{square}(x)$	$x = x + 1$
P1	P2
 READ $x \rightarrow 5$ WRITE $25 \rightarrow x$ 	READ $x \rightarrow 5$ WRITE $6 \rightarrow x$
$x = 26$	

- This ordering is also possible; P2 gets the last word.
- There are also read-write conflicts here. What is the total number of possible final values for x ?

Write-Write Conflicts (II)

$x = 5$	
$x = \text{square}(x)$	$x = x + 1$
P1	P2
 READ $x \rightarrow 5$ WRITE $25 \rightarrow x$ 	READ $x \rightarrow 5$ WRITE $6 \rightarrow x$
$x = 26$	

- This ordering is also possible; P2 gets the last word.
- There are also read-write conflicts here. What is the total number of possible final values for x ? **Four: 25, 5, 26, 36**