Sign In

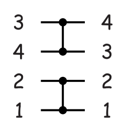
- Website: `https://goo.gl/forms/FzHSa5INKlavWIJC3`
- Enter the word of the day in the appropriate slot.

Lecture #5: Parallelism

- Moore's law ("Transistors per chip doubles every N years"), where N is roughly 2 (about $5{,}000{,}000\times$ increase since 1971).
- Similar rule applied to processor speeds until around 2004.
- Speeds have flattened: 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:



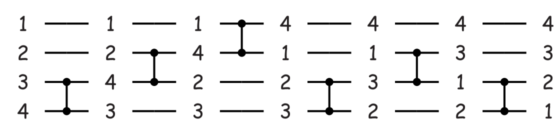
- Each vertical bar represents a *comparator*—a comparison operation or hardware to carry it out—and each horizontal line carries a data item from the list.
- A comparator compares two data items coming from the left, swapping them if the lower one is larger than the upper one.
- Comparators can be grouped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

Sequential sorting

- One (wasteful but simple) way to sort a list of items into ascending order goes like this:

```
for i in range(len(L) - 1):
    for j in range(len(L) - 1):
        if L[j] > L[j + 1]:
            L[j], L[j+1] = L[j+1], L[j]
```

- In general, there will be $\Theta(\ ?\ )$ steps.
- Diagrammatically (read bottom to top):



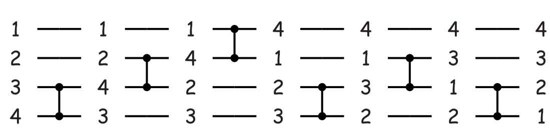
- Each comparator is a separate operation in time.
- Many comparators operate on distinct data, but unfortunately, there is an overlap between the operations in adjacent columns.

Sequential sorting

- One (wasteful but simple) way to sort a list of items into ascending order goes like this:

```
for i in range(len(L) - 1):
    for j in range(len(L) - 1):
        if L[j] > L[j + 1]:
            L[j], L[j+1] = L[j+1], L[j]
```

- In general, there will be $\Theta(N^2)$ steps.
- Diagrammatically (read bottom to top):

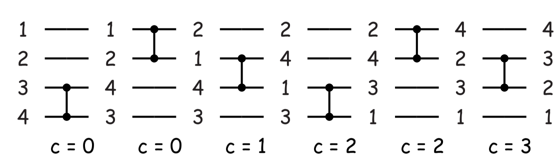


- Each comparator is a separate operation in time.
- Many comparators operate on distinct data, but unfortunately, there is an overlap between the operations in adjacent columns.

A Reorganization

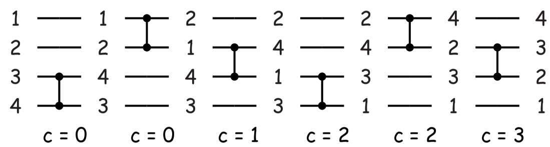
- It's not obvious, but we can accomplish the same final result with a different order of swaps:

```
for c in range(len(L)):
    # Swap even/odd pairs when c is even, odd/even pairs when c is odd
    for j in range(c % 2, len(L) - 1, 2):
        if L[j] > L[j + 1]: L[j], L[j+1] = L[j+1], L[j]
```

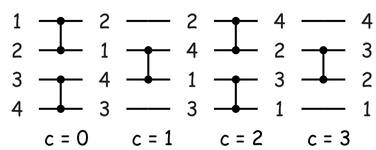


Exploiting Parallelism

- With this reorganization, can exploit parallelism, because not all columns need be executed in sequence. Thus, the sequential program:

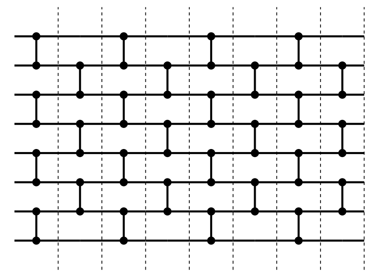


- Can be partially overlapped, saving two steps:



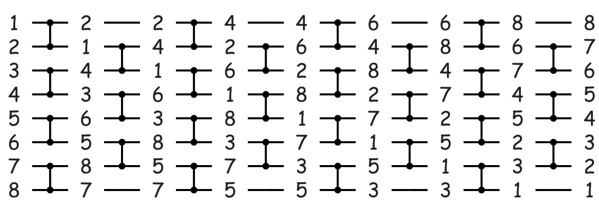
Odd-Even Transposition Sorter

- Here's a larger example:



The dashed lines separate parallel groups. Everything in one group can happen in parallel, one group at a time in sequence.

Odd-Even Sort Example



- What would have been 28 separate sequential operations (in general about $N(N-1)/2$) becomes 8 (N) *parallel operations*.
- Assuming we have enough processors, we have sped things up by a factor of about $N/2$.

Other Kinds of Sorting

- Another way to sort a list is *merge sort*:

```
def sort(L, first, last):
    if first < last:
        middle = (first + last) // 2
        sort(L, first, middle)
        sort(L, middle+1, last)
        L[:] = merge(L[first:middle+1], L[middle+1:last+1])
        # Merge takes two sorted lists and interleaves
        # them into a single sorted list.
```

- Assuming that merging takes time $\Theta(N)$ for two lists of size $N/2$, this operation takes $\Theta(\quad ?)$ steps.
- We can reorder its operations to get *(Batcher's) bitonic sort*, which can sort in $\Theta((\lg N)^2)$ time.

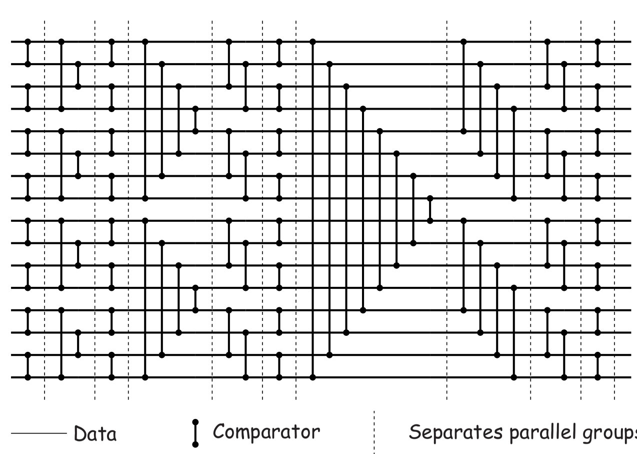
Other Kinds of Sorting

- Another way to sort a list is *merge sort*:

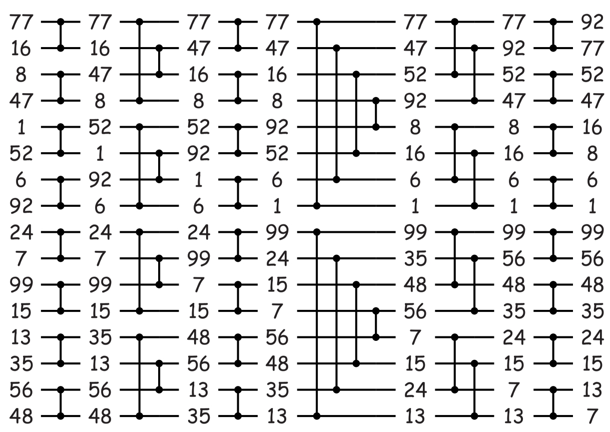
```
def sort(L, first, last):
    if first < last:
        middle = (first + last) // 2
        sort(L, first, middle)
        sort(L, middle+1, last)
        L[:] = merge(L[first:middle+1], L[middle+1:last+1])
        # Merge takes two sorted lists and interleaves
        # them into a single sorted list.
```

- Assuming that merging takes time $\Theta(N)$ for two lists of size $N/2$, this operation takes $\Theta(N \lg N)$ steps.
- We can reorder its operations to get *(Batcher's) bitonic sort*, which can sort in $\Theta((\lg N)^2)$ time.

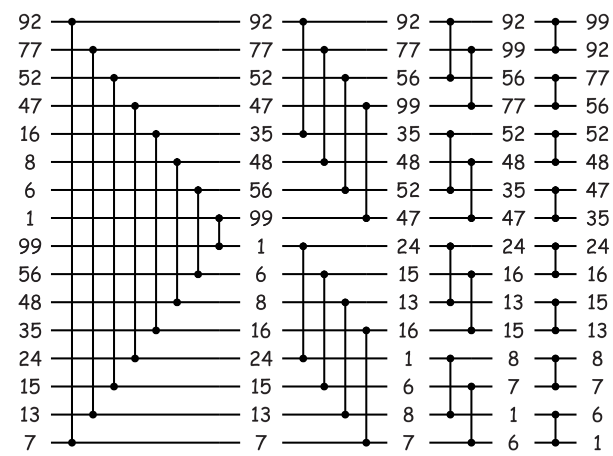
Example: Bitonic Sorter



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


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	**P2**

`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`

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 = square(x)`	`y = x + 1`
P1	**P2**

`READ x -> 5`
`(calculate 5*5 -> 25)`
`WRITE 25 -> x`
`|`
	`|`
	`READ x -> 5`
	`(calculate 5+1 -> 6)`
	`WRITE 6 -> 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 = square(x)	y = x + 1
P1	P2
READ x -> 5 (calculate 5*5 -> 25) WRITE 25 -> x | | |	| | | READ x -> 25 (calculate 25+1 -> 26) WRITE 26 -> 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:

x = 5

x = square(x)	x = x + 1
P1	P2
| READ x -> 5 | | WRITE 25 -> x	READ x -> 5 | WRITE 6 -> x |

x = 25

- 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 = square(x)	x = x + 1
P1	P2
| READ x -> 5 WRITE 25 -> x |	READ x -> 5 | | WRITE 6 -> x

x = 6

- 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 = square(x)	x = x + 1
P1	P2
| READ x -> 5 WRITE 25 -> x |	READ x -> 5 | | WRITE 6 -> x

x = 6

- 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