Lecture 35: Concurrency, Parallelism, and Distributed Computing

Definitions

- *Sequential Process:* Our subject matter up to now: processes that (ultimately) proceed in *a single sequence* of primitive steps.
- *Concurrent Processing:* The logical or physical division of a process into multiple sequential processes.
- *Parallel Processing:* A variety of concurrent processing characterized by the *simultaneous execution* of sequential processes.
- *Distributed Processing:* A variety of concurrent processing in which the individual processes are physically separated (often using heterogeneous platforms) and communicate through some network structure.

Purposes

We may divide a single program into multiple programs for various reasons:

- *Computation Speed* through operating on separate parts of a problem simultaneously, or through
- *Communication Speed* through putting parts of a computation near the various data they use.
- *Reliability* through having mulitple physical copies of processing or data.
- *Security* through separating sensitive data from untrustworthy users or processors of data.
- *Better Program Structure* through decomposition of a program into logically separate processes.
- *Resource Sharing* through separation of a component that can serve mulitple users.
- *Manageability* through separation (and sharing) of components that may need frequent updates or complex configuration.

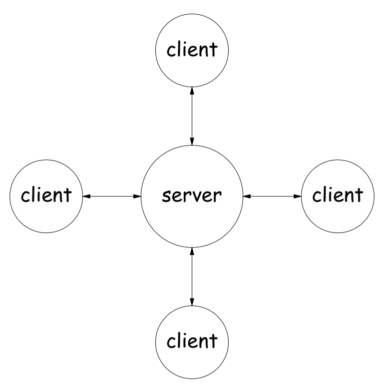
Communicating Sequential Processes

- All forms of concurrent computation can be considered instances of *communicating sequential processes*.
- That is, a bunch of "ordinary" programs that communicate with each other through what is, from their point of view, input and output operations.
- Sometimes the actual communication medium is *shared memory:* input looks like reading a variable and output looks like writing a variable. In both cases, the variable is in memory accessed by multiple computers.
- At other times, communication can involve I/O over a network such as the Internet.
- In principle, either underlying mechanism can be made to look like either access to variables or explicit I/O operations to a programmer.

Distributed Communication

- With sequential programming, we don't think much about the cost of "communicating" with a variable; it happens at some fixed speed that is (we hope) related to the processing speed of our system.
- With distributed computing, the architecture of communication becomes important.
- In particular, costs can become uncertain or heterogeneous:
  - It may take longer for one pair of components to communicate than for another, or
  - The communication time may be unpredictable or load-dependent.

Simple Client-Server Models



- Example: web servers
- Good for providing a service
- Many clients, one server
- Easy server maintenance.
- Single point of failure
- Problems with scaling

Variations: on to the cloud

- Google and other providers modify this model with redundancy in many ways.
- For example, *DNS load balancing* (DNS = *Domain Name System*) allows us to specify multiple servers.
- Requests from clients go to different servers that all have copies of relevant information.
- Put enough servers in one place, you have a *server farm*. Put servers in lots of places, and we have a *cloud*.

Communication Protocols

- One characteristic of modern distributed systems is that they are conglomerations of products from many sources.
- Web browers are a kind of universal client, but there are numerous kinds of browsers and many potential servers (and clouds of servers).
- So there must be some agreement on how they talk to each other.
- The *IP Protocol* is an agreement for specifying destinations, packaging messages, and delivering those messages.
- On top of this, the *transmission control protocol (TCP)* handles issues like persistent telephone-like connections and congestion control.
- The DNS handles conversions between names (inst.eecs.berkeley.edu) and IP addresses (128.32.42.199).
- The *HyperText Transfer Protocol* handles transfer of requests and responses from web servers.

Example: HTTP

- When you click on a link, such as

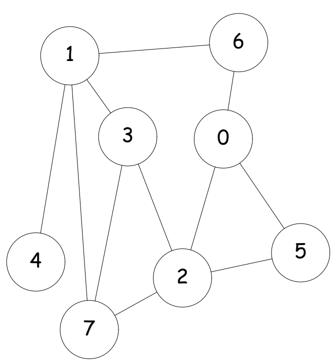
  `http://inst.eecs.berkeley.edu/~cs61a/lectures,`

  your browser:
  - Consults the DNS to find out where to look for `inst.eecs.berkeley.edu`.
  - Sends a message to port 80 at that address:

    ```
    GET ~cs61a/lectures HTTP 1.1
    ```
  - The program listening there (the web server) then responds with

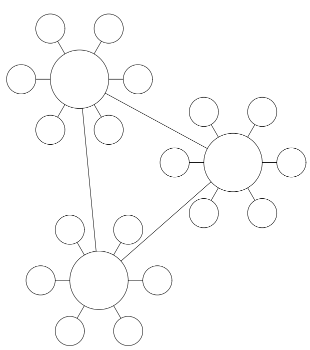
    ```
    HTTP/1.1 200 OK
    Content-Type: text/html
    Content-Length: 1354

    <html> ... text of web page
    ```
- Protocol has other messages: for example, POST is often used to send data in forms from your browser. The data follows the POST message and other headers.

Peer-to-Peer Communication



- No central point of failure; clients talk to each other.
- Can route around network failures.
- Computation and memory shared.
- Can grow or shrink as needed.
- Used for file-sharing applications, botnets (!).
- But, deciding routes, avoiding congestion, can be tricky.
- (E.g., Simple scheme, broadcasting all communications to everyone, requires N^2 communication resource. Not practical.
- Maintaining consistency of copies requires work.
- Security issues.

Clustering



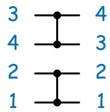
- A peer-to-peer network of "supernodes," each serving as a server for a bunch of clients.
- Allows scaling; could be nested to more levels.
- Examples: Skype, network time service.

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 flattend: further increases to be obtained through *parallel processing* (witness: multicore/manycode 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

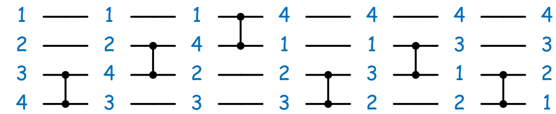
- Sorting a list presents obvious opportunities for parallelization.
- Can illustrate various methods diagrammatically using *comparators* as an elementary unit:



- 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.

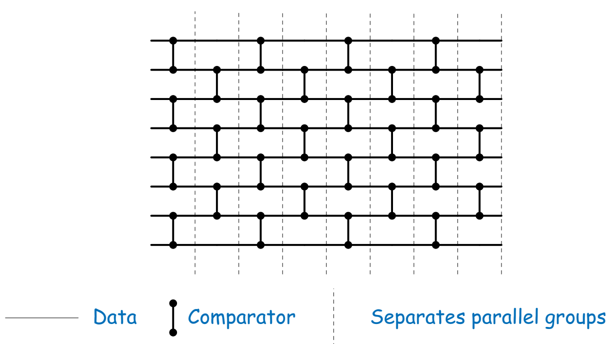
Sequential sorting

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

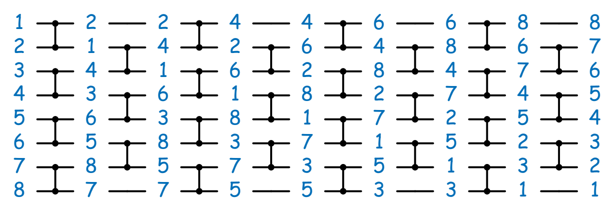


- Each comparator is a separate operation in time.
- In general, there will be $\Theta(N^2)$ steps.
- But since some comparators operate on distinct data, we ought to be able to overlap operations.

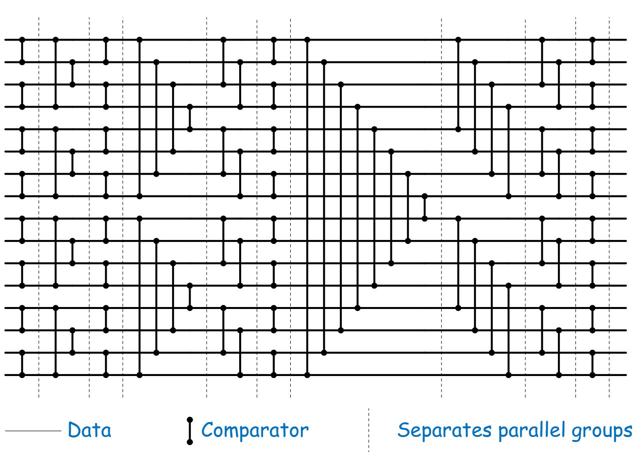
Odd-Even Transposition Sorter



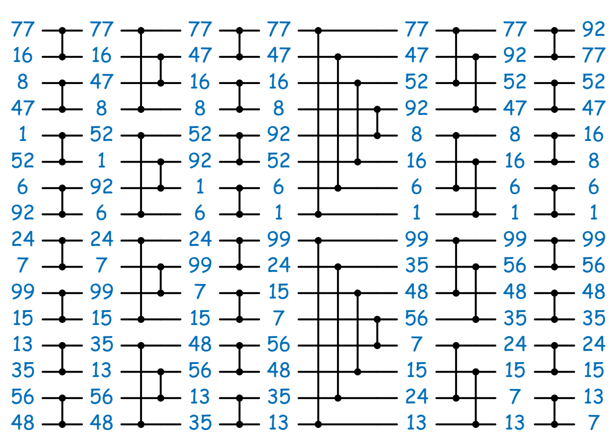
Odd-Even Sort Example



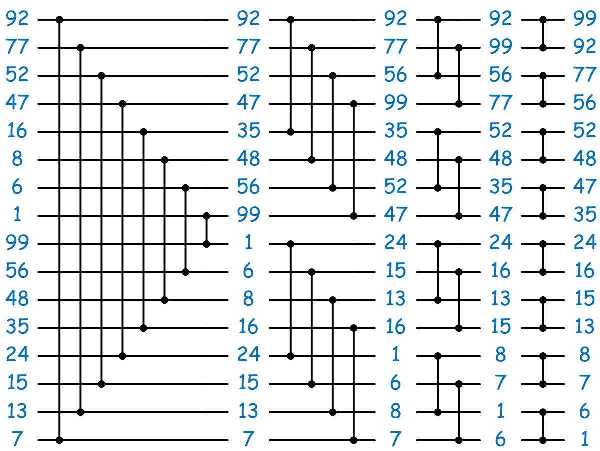
Example: Bitonic Sorter



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`	`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 = 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