Announcements

• HW#8 coming out later today. Due next Wednesday.
• I do will not have office hours tomorrow: moved to 4PM Friday this week only.
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 multiple 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 multiple 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 browsers 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 “super-nodes,” 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 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:

```
   3   4
  / \ / \  
 4   3
  |   |
 2   2
  |   |
 1   1
```

- 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 way to sort a list of items into ascending order goes like this:

```python
for i in range(len(L) - 1):
    for j in range(i, 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):

```
1    1    1    4    4    4    4    4
2    2    4    1    1    3    3
3    4    2    2    3    1    2
4    3    3    3    2    2    1
```

• 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 way to sort a list of items into ascending order goes like this:

  ```python
  for i in range(len(L) - 1):
      for j in range(i, 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):

```
1 ——— 1 ——— 1 ——— 4 ——— 4 ——— 4 ——— 4 ——— 4
2 ——— 2 ——— 4 ——— 1 ——— 1 ——— 3 ——— 3
3 ——— 4 ——— 2 ——— 2 ——— 3 ——— 1 ——— 2
4 ——— 3 ——— 3 ——— 3 ——— 2 ——— 2 ——— 1
```

- 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:

```python
for c in range(len(L) // 2):
    # Swap even/odd pairs
    for j in range(0, len(L) - 1, 2):
        if L[j] > L[j + 1]: L[j], L[j+1] = L[j+1], L[j]
    # Swap odd/even pairs
    for j in range(1, len(L) - 1, 2):
        if L[j] > L[j + 1]: L[j], L[j+1] = L[j+1], L[j]
```

1 ─── 1 ─── 2 ─── 2 ─── 2 ─── 4 ─── 4
2 ─── 2 ─── 1 ─── 4 ─── 4 ─── 2 ─── 3
3 ─── 4 ─── 4 ─── 1 ─── 3 ─── 3 ─── 2
4 ─── 3 ─── 3 ─── 3 ─── 1 ─── 1 ─── 1
Odd-Even Transposition Sorter

- Now suppose we repeatedly scrunch together adjacent columns of non-overlapping operations:

```
Data     Comparator     Separates parallel groups
```

————— Data       Comparator       Separates parallel groups
Odd-Even Sort Example

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

• What would have been 28 separate sequential operations (in general about \(N(N-1)/2\)) becomes 8 (\(N\)) parallel operations.

• If they can be carried out in parallel, we have sped things up by a factor proportional to \(N\).