Lecture 33: Concurrency

- 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 (with a different exponent).
- 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:

1 2 4
3 1 2

- 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:

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

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

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

Example: Bitonic Sorter

Data Comparator Separates parallel groups
Bitonic Sort Example (I)

| 77 | 77 | 47 | 47 | 77 | 77 | 92 | 92 |
| 16 | 16 | 47 | 47 | 47 | 92 | 92 | 77 |
| 8  | 47 | 16 | 16 | 52 | 52 | 52 | 52 |
| 47 | 8  | 8  | 8  | 8  | 8  | 16 | 16 |
| 1  | 52 | 52 | 92 | 92 | 8  | 8  | 16 |
| 52 | 1  | 92 | 52 | 16 | 16 | 8  | 8  |
| 6  | 92 | 1  | 6  | 6  | 6  | 6  | 6  |
| 92 | 6  | 6  | 1  | 1  | 1  | 1  | 1  |
| 24 | 24 | 24 | 99 | 99 | 99 | 99 | 99 |
| 7  | 7  | 99 | 24 | 35 | 35 | 35 | 35 |
| 99 | 99 | 7  | 15 | 48 | 48 | 48 | 48 |
| 15 | 15 | 15 | 7  | 56 | 56 | 56 | 56 |
| 35 | 35 | 48 | 48 | 7  | 24 | 24 | 24 |
| 35 | 13 | 56 | 48 | 15 | 15 | 15 | 15 |
| 56 | 56 | 13 | 35 | 24 | 7  | 13 | 13 |
| 48 | 48 | 35 | 13 | 13 | 13 | 13 | 13 |

Bitonic Sort Example (II)

| 92 | 92 | 92 | 92 |
| 77 | 77 | 77 | 92 |
| 52 | 52 | 56 | 56 |
| 47 | 47 | 99 | 99 |
| 16 | 16 | 35 | 35 |
| 8  | 8  | 48 | 48 |
| 6  | 6  | 56 | 52 |
| 1  | 1  | 99 | 47 |
| 99 | 1  | 24 | 24 |
| 56 | 6  | 15 | 16 |
| 48 | 8  | 13 | 15 |
| 35 | 16 | 16 | 15 |
| 24 | 24 | 1  | 8  |
| 15 | 15 | 6  | 7  |
| 13 | 8  | 1  | 6  |
| 7  | 7  | 6  | 1  |

Mapping and Reducing in Parallel

- The map function in Python conceptually provides many opportunities for parallel computation, if the computations of individual items is independent.
- Less obviously, so does reduce, if the operation is associative. If list \( L \equiv L1 + L2 \), and \( op \) is an associative operation, then
  \[ reduce(op, L) \equiv op(reduce(op, L1), reduce(op, L2)) \]
  and the two smaller reductions can happen in parallel.

Map-Reduce

- Google™ patented an embodiment of this approach (the validity of which is under dispute). Here's a very simplified version.
- User specifies a mapping operation and a reduction operation.
- In the mapping phase, the map operation is applied to each item of data, yielding a list of key-value pairs for each item.
- The reduce operation is then applied on all the values for each distinct key.
- The final result is a list of key-value pairs, with each value being the reduction of the values for that key as produced by the mapping phase.
- Standard simple example:
  - Each input item is a page of text.
  - The map operation takes a page of text ("The cow jumped over the moon") and produces a list with the words as keys and the value \( 1 \) ("the", 1), ("cow", 1), ("jumped", 1),...).
  - The reduce phase now sums the values for each key.
  - Result: for each key (word), get the total count.

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
  \[ x = \text{square}(x) \]  READ x -> 5
  \[ \text{(calculate 5*5 -> 25)} \]  WRITE 25 -> x
  \[ y = 6 \]  WRITE 6 -> y
  \[ y += 1 \]  READ y -> 6
  \[ \text{(calculate 6+1 -> 7)} \]  WRITE 7 -> y
Conflict-Free Computation

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

\[
\begin{align*}
  x & = 5 \\
  x & = square(x) \\
  y & = 6 \\
  y & = y + 1 \\
\end{align*}
\]

\[
\begin{align*}
  \text{WRITE} & \; 5 \rightarrow x \\
  \text{READ} & \; x \rightarrow 5 \\
  \text{(calculate} & \; 5*5 \rightarrow 25) \\
  \text{WRITE} & \; 25 \rightarrow x \\
\end{align*}
\]

\[
\begin{align*}
  \text{WRITE} & \; 6 \rightarrow y \\
  \text{READ} & \; y \rightarrow 6 \\
  \text{(calculate} & \; 6+1 \rightarrow 7) \\
  \text{WRITE} & \; 7 \rightarrow y \\
\end{align*}
\]

\[
\begin{align*}
  x & = 25 \\
  y & = 7 \\
\end{align*}
\]

- 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.

\[
\begin{align*}
  x & = 5 \\
  x & = square(x) \\
  y & = x + 1 \\
\end{align*}
\]

\[
\begin{align*}
  \text{P1} & \quad \text{P2} \\
  \text{READ} \; x \rightarrow 5 & \quad \text{READ} \; x \rightarrow 5 \\
  \text{(calculate} \; 5+5 \rightarrow 25) & \quad \text{(calculate} \; 5+1 \rightarrow 6) \\
  \text{WRITE} \; 25 \rightarrow x & \quad \text{WRITE} \; 6 \rightarrow y \\
\end{align*}
\]

\[
\begin{align*}
  x & = 25 \\
  y & = 6 \\
\end{align*}
\]

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

Read-Write Conflicts (II)

- Here's another possible sequence of events

\[
\begin{align*}
  x & = 5 \\
  x & = square(x) \\
  y & = x + 1 \\
\end{align*}
\]

\[
\begin{align*}
  \text{P1} & \quad \text{P2} \\
  \text{READ} \; x \rightarrow 5 & \quad \text{READ} \; x \rightarrow 5 \\
  \text{(calculate} \; 5*5 \rightarrow 25) & \quad \text{(calculate} \; 5+1 \rightarrow 6) \\
  \text{WRITE} \; 25 \rightarrow x & \quad \text{WRITE} \; 6 \rightarrow y \\
\end{align*}
\]

\[
\begin{align*}
  x & = 25 \\
  y & = 26 \\
\end{align*}
\]

Read-Write Conflicts (III)

- The problem here is that nothing forces \( \text{P1} \) to wait for \( \text{P2} \) 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 \( x \):

\[
\begin{align*}
  x & = 5 \\
  x & = square(x) \\
  x & = x + 1 \\
\end{align*}
\]

\[
\begin{align*}
  \text{P1} & \quad \text{P2} \\
  \text{READ} \; x \rightarrow 5 & \quad \text{READ} \; x \rightarrow 5 \\
  \text{(calculate} \; 5+5 \rightarrow 25) & \quad \text{(calculate} \; 5+1 \rightarrow 6) \\
  \text{WRITE} \; 25 \rightarrow x & \quad \text{WRITE} \; 6 \rightarrow x \\
\end{align*}
\]

\[
\begin{align*}
  x & = 25 \\
\end{align*}
\]

- 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)

- This ordering is also possible; \( \text{P2} \) 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
Let's go back to bank accounts:

```python
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance
    @property
def balance(self): return self._balance
def withdraw(amount):
    if amount > self._balance:
        raise ValueError("insufficient funds")
    else:
        self._balance -= amount
        return self._balance
```

```python
acct = BankAccount(10)
acct.withdraw(8)
acct.withdraw(7)
```

At this point, we'd like to have the system raise an exception for one of the two withdrawals, and to set `acct.balance` to either 2 or 3, depending on which withdrawer gets to the bank first, like this...

```
    acct = BankAccount(10)
    acct.withdraw(8)
    acct.withdraw(7)
```

But instead, we might get...

```
    acct = BankAccount(10)
    acct.withdraw(8)
    acct.withdraw(7)
```

Oops!

Desired Outcome

```python
class BankAccount:
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
            return self._balance
```

```python
acct = BankAccount(10)
acct.withdraw(8)
acct.withdraw(7)
```

Undesirable Outcome

```python
class BankAccount:
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
            return self._balance
```

Serializability

- We define the desired outcomes as those that would happen if withdrawals happened sequentially, in some order.
- The nondeterminism as to which order we get is acceptable, but results that are inconsistent with both orderings are not.
- These latter happen when operations overlap, so that the two processes see inconsistent views of the account.
- We want the withdrawal operation to act as if it is atomic—as if, once started, the operation proceeds without interruption and without any overlapping effects from other operations.