CS 61A/CS 98-52

Mehrdad Niknami

University of California, Berkeley
Today, we’re going to learn how to add & multiply. **Exciting!**

Let’s add two positive $n$-bit integers ($n = 8$ here):

- **Carry:** 1 111111
- **Augend:** 10110111
- **Addend:** + 10011101
  
  
  **----------**

- **Sum:** 101010100

This is called *ripple-carry addition*. Some questions:

1. How big can the sum be (at most)? What is the worst case?
2. How long does summation take in the worst case? **Why?**

...we’ll come back to this!
First computer design (difference engine) in 1822 (!!) and later, the analytical engine, by Charles Babbage (1791-1871)

First description of “MIMD” parallelism in 1842 (!!!!) in Sketch of The Analytical Engine Invented by Charles Babbage, by Luigi F. Menabrea

First theory of computation by Alan Turing in 1936

First electronic analog computer created in 1942 for bombing in WWII

First electronic digital computer created in 1943 ⇒ Electronic Numerical Integrator and Computer (ENIAC)

First description of parallel programs in 1958 (Stanley Gill)

First multiprocessor system (Multics) in 1969

Lots of parallel computing research starting in 1970s... then faded away

Multi-core systems reinvigorated parallel computing around 2001
Long story short...

- Parallel computing goes back longer than you think
- Lots of useful research from the 1900s finding life again since processors stopped getting faster
Terminology

Some basic terminology:

- **Process**: A *running program*
  Processes cannot access each others’ memory by default

- **Thread**: A *unit of program flow*
  \((N\text{ threads }= n\text{ independent executions of code})\)
  Threads maintain their own *execution contexts* in a given process

- **Thread context**: *All the information a thread needs to run code*
  This includes the location of the code that it is currently being executing, as well as its current stack frame (local variables, etc.)

- **Concurrency**: *Overlapping operations* \((X\text{ begins before } Y\text{ ends})\)

- **Parallelism**: *Simultaneously-occurring operations* (multiple operations happening *at the same time*)
Parallel operations are always concurrent by definition.

Concurrent operations need not be in parallel (open door, open window, close door, close window).

Parallelism gives you a speed boost (multiple operations at the same time), but requires $N$ processors for $N \times$ speedup.

Concurrency allows you to avoid stopping one thing before starting another, and can occur on a single processor.
Distributed computation (running on multiple machines) is more difficult:

- Needs fault-tolerance (more machines = higher failure probability)
- Lack of shared memory
- More limited communication bandwidth (network slower than RAM)
- Time becomes problematic to handle

Rich literature, e.g. actor-based models of computation (MoC) such as discrete-event, synchronous-reactive, synchronous dataflow, etc. for analyzing/designing systems with guaranteed performance or reliability.
Threading example:

```python
import threading
t = threading.Thread(target=print, args=('a',))
t.start()
print('b')  # may print 'b' before or after 'a'
t.join()   # wait for t to finish
```
**Race condition:** *When a thread attempts to access something being modified by another thread.* **Race conditions are generally bad.**

Example:

```python
import threading
lst = [0]
def f():
    lst[0] += 1  # write 1 might occur after read 2

if threading:
    t = threading.Thread(target=f)
    t.start()
    f()
    t.join()

assert lst[0] in [1, 2]  # could be any of these!
```
**Concurrency Control**

**Mutex (Lock in Python):** Object that can prevent concurrent access (mutual-exclusion). Example:

```python
import threading
lock = threading.Lock()
lst = [0]
def f():
    lock.acquire()  # waits for mutex to be available
    lst[0] += 1  # only one thread may run this code
    lock.release()  # makes mutex available to others

    t = threading.Thread(target=f)
t.start()
f()
t.join()

assert lst[0] in [2]  # will always succeed
```
Sadly, in CPython, multithreaded operations cannot occur in parallel, because there is a “global interpreter lock” (GIL). Therefore, Python code cannot be sped up in CPython.\textsuperscript{1}

To obtain parallelism in CPython, you can use multiprocessing: running another copy of the program and communicating with it.

Jython, IronPython, etc. can run Python in parallel, and most other languages support parallelism as well.

\textsuperscript{1}However, Python code can release GIL when calling non-Python code.
Threads/processes need to communicate. Common techniques:

- **Shared memory:** mutating shared objects (if all on 1 machine)
  - Pros: Reduces copying of data (faster/less memory)
  - Cons: Must block execution until lock is acquired (slow)

- **Message-passing:** sending data through thread-safe queues
  - Pros: Queue can buffer & work asynchronously (faster)
  - Cons: Increases need to copy data (slower/more memory)

- **Pipes:** synchronous version of message-passing (“rendezvous’’)

Message-passing example for parallelizing $f(x) = x^2$:

```python
from multiprocessing import Process, Queue

def f(q_in, q_out):
    while True:
        x = q_in.get()
        if x is None:
            break
        q_out.put(x ** 2)  # real work

if __name__ == '__main__':  # only on main thread
    qs = (Queue(), Queue())
    procs = [Process(target=f, args=qs) for _ in range(4)]
    for proc in procs: proc.start()
    for i in range(10): qs[0].put(i)  # send inputs
    for i in range(10): print(qs[1].get())  # receive outputs
    for proc in procs: qs[0].put(None)  # notify finished
    for proc in procs: proc.join()
```
Common parallelism technique: **divide-and-conquer**

1. Divide problem into separate subproblems
2. Solve subproblems in parallel
3. Merge sub-results into main result

XOR (and AND, and OR) are easy to parallelize:

1. Split each $n$-bit number into $p$ pieces
2. XOR each $n/p$-bit pair of numbers independently
3. Put back the bits together

Can we do something similar with **addition**?
Addition

Let’s go back to **addition**.

We have two $n$-bit numbers to add.

What if we take the same approach for $+$ as for XOR?

1. Split each $n$-bit number into $p$ pieces
2. Add each $n/p$-bit pair of numbers independently
3. Put back the bits together
4. ...

5. Profit? No? **What’s wrong?**

We need to propagate carries! How long does it take? $\Theta(n)$ time

(How) can we do better?
Key idea #1: A carry can be either 0 or 1... and we add different pieces in parallel... and then select the correct one based on carry! ⇒ This is called a carry-select adder.

Key idea #2: We can do this recursively. ⇒ This is called a conditional-sum adder.

How fast is a conditional-sum adder?

- Running time is proportional to maximum propagation depth
- We solve two problems of half the size simultaneously
- We combine solutions with constant extra work
- Therefore, parallel running time is $\Theta(\log n)$

However, we do more work: $T(n) = 2T(n/2) + c = \Theta(n \log n)$
Other algorithms also exist with different trade-offs:

- Carry-skip adder
- Carry-lookahead adder (CLA)
- Kogge–Stone adder (“parallel-prefix” CLA; widely used)
- Brent-Kung adder
- Han–Carlson adder
- Lynch–Swartzlander spanning tree adder (fastest?)

...I don’t know them. But $\Theta(\log n)$ is already asymptotically optimal. :-)

Some algorithms are better suited for hardware due to lower “fan-out”: e.g. 1 bit is too “weak” to drive 16 bits all by itself.
How do we multiply?

Multiplicand: 10110111
Multiplier: * 10011101

-----------------

10110111
+ 00000000
+ 10110111
+ 10110111
+ 10110111
+ 00000000
+ 00000000
+ 10110111

-----------------

Product: 1110000000111011
For two $n$-bit numbers, how long does it take in parallel?

- Multiplication by 1 is a copy, taking $\Theta(1)$ depth
- There are $n$ additions
- Divide-and-conquer therefore takes $\Theta(\log n)$ additions
- Each addition takes $\Theta(\log n)$ depth
- Total depth is therefore $\Theta((\log n)^2)$

...can we do better? :-) How?
**Carry-save addition:** reduce every $a + b + c$ into $r + s$ in parallel:

- Compute all carry bits $r$ independently
  \[ \Rightarrow \text{This is just OR, so } \Theta(1) \text{ depth} \]

- Compute all sums-excluding-carries $s$ independently
  \[ \Rightarrow \text{This is just XOR, so } \Theta(1) \text{ depth} \]

- Recurse on new $r_1 + s_1 + r_2 + s_2 + \ldots$ until final $r + s$ is obtained.
  \[ \Rightarrow \text{This takes } \Theta(\log n) \text{ levels of recursion} \]

- Compute final sum in additional $\Theta(\log n)$ depth

Total depth is therefore $\Theta(\log n)$!\(^2\)

---

\(^2\)Simplified; detailed analysis is a little tedious. See [here](#)
Parallel Prefix

There isn’t too much special about *addition* from basic arithmetic. Often the same tricks apply to any binary operator $\oplus$ that is associative!

Parallel addition can be generalized this way, called “parallel prefix”:

- Say we want to compute cumulative sum of 1, 2, 3, ...
- First, group into binary tree: (((1 2) (3 4)) ((5 6) (7 8))) ...
- Then, evaluate sums for all nodes recursively toward root
- Finally, propagate sums back down from root to right-hand children

This is a very flexible operation, useful as a basic parallel building block. (More notes can be found on MIT’s website.)
A common pattern for parallel data processing is:

```python
from functools import reduce
outputs = map(lambda x: ..., inputs)
result = reduce(lambda r, x: ..., outputs, initial)
```

- **map** you have already seen: it *transforms* elements
- **reduce** is anything like $+$, $\times$ to *summarize* elements
- Transformations assumed to ignore order (to allow parallelism)
MapReduce

Google recognized this and built a fast framework called MapReduce for automatically parallelizing & distributing such code across a cluster

- *MapReduce: Simplified Data Processing on Large Clusters*
  by Jeffrey Dean and Sanjay Ghemawat (2004)

- System and method for efficient large-scale data processing
  U.S. Patent 7,650,331

Fault-tolerance is handled automatically (why is this possible?)

Apache Hadoop later developed as an open-source implementation

“MapReduce” became a general *programming model* for distributed data processing

*Spark* (Matei Zaharia, UCB AMPLab, now at Databricks) developed as a faster implementation that processes data in RAM
Parallel map is easy in Python!

```python
>>> import math
>>> from multiprocessing.pool import Pool
>>> pool = Pool()
>>> pool.map(math.sqrt, [1, 2, 3, 4])
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
```

This a *higher-level* threading construct that makes your life simpler.
Not everything fits into a MapReduce model
- Inputs may be generated on the fly
- Mappers might depend on many inputs
- Mappers may need lots of communication
- Computation may not be nicely “layered” at all
- ...

Parallel & distributed computation still an open research problem.