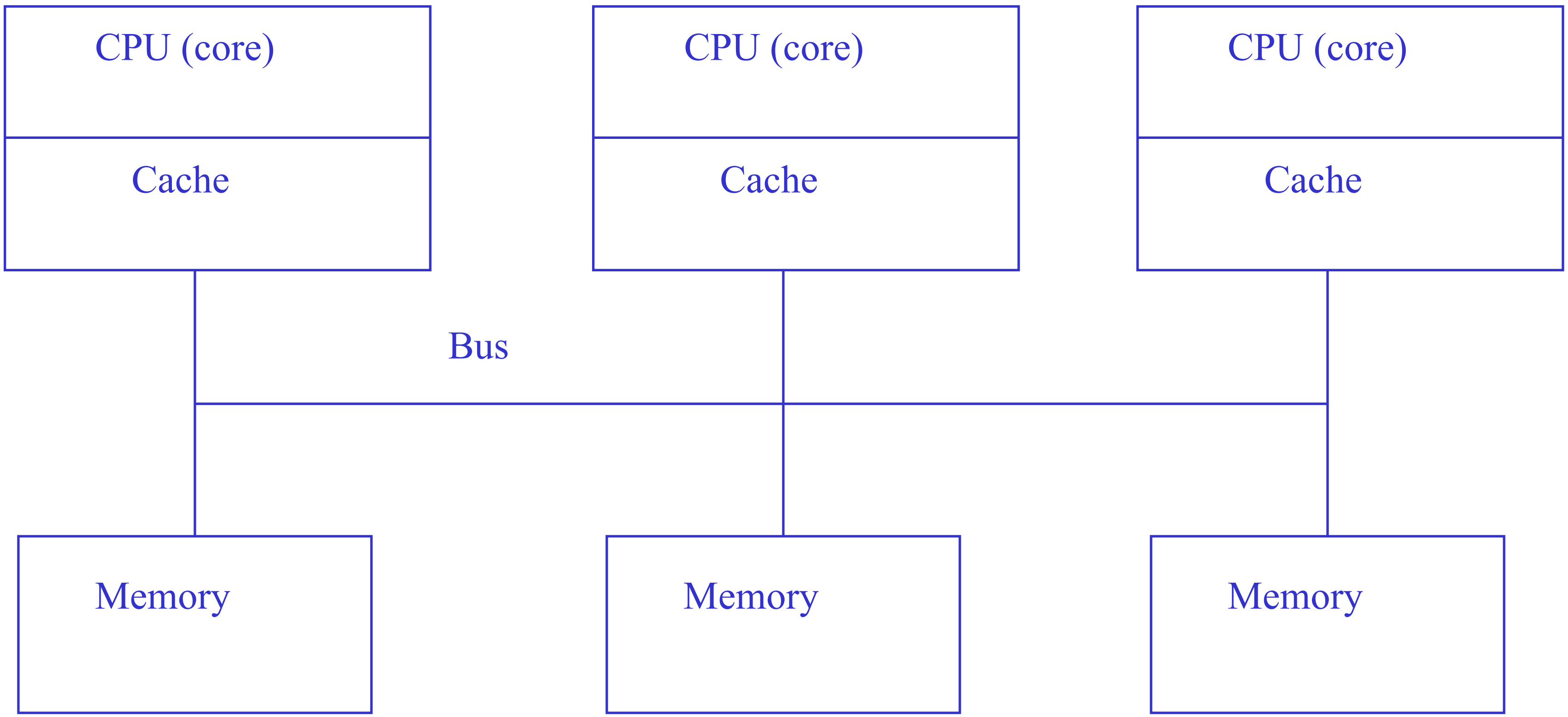
Parallel Scientific Programming

- Definitions
  - Speedup: T_s/T_p, where T_s is the sequential time and T_p is the time when using *p* cores
    - "Perfect speedup" is *p*, which really should be called "linear speedup"
    - Typically, speedup is less than *p*---but it can be larger because of memory hierarchy effects
  - Efficiency: Speedup/*p*
    - Intuition: how well am I using my *p* cores

“Superlinear” Speedup?



Strictly more cache when more cores are used

Can result in fewer cache misses when using more cores

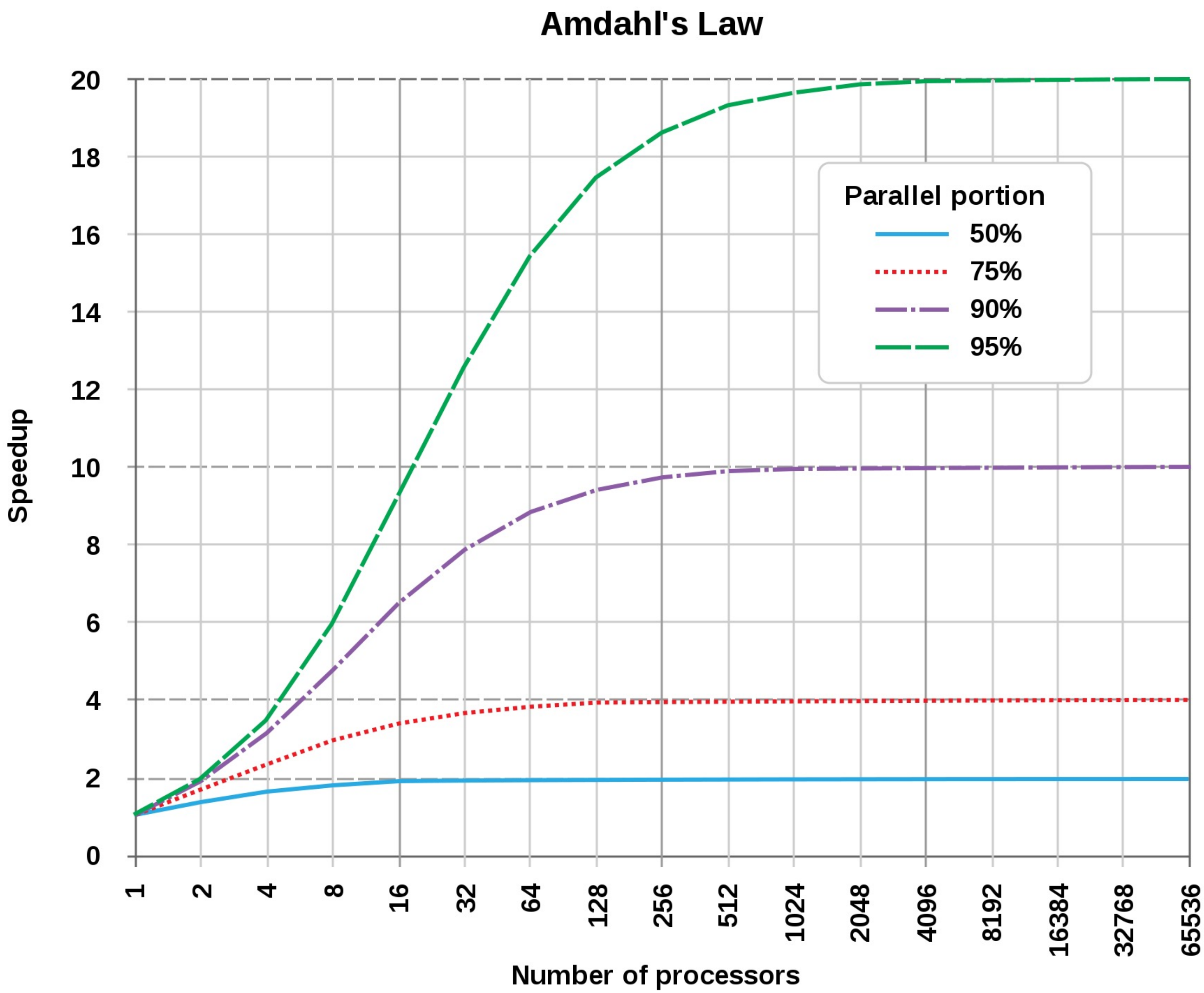
Same argument can hold for any level of the memory hierarchy

Graphs of Speedup and Efficiency

(shown on board)

Parallel Scientific Programming

- Definitions, continued
  - Amdahl's law: if T_s is sequential time, then:
    - $T_p \approx T_s * (1-f) + (T_s / p) * f$, where f is the fraction of the program that is parallelizable, and p is the number of cores.
    - Intuition: the non-parallelizable portion doesn't speed up at all, and the parallelizable part scales linearly
    - Of course, this isn't true in general; one might, for example, have load imbalance in a parallelizable portion---also ignored is process/thread creation, communication, synchronization



Source: By Daniels220 at English Wikipedia, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=6678551

Example use of Amdahl's Law

- Suppose program takes 50 seconds sequentially, but of that 50 seconds:
  - 5 seconds is initialization that can't be parallelized
  - 5 seconds is finalization that also can't be parallelized
- Then the **maximum speedup** possible is 5!
  - Even if we have an arbitrarily large number of cores!
  - Lesson: try to avoid sequential portions of code

Different parallel programming styles (end of Chapter 3)

- Iterative: SPMD (e.g., Jacobi iteration)
- Recursive: adaptive quadrature
- Task parallel: independent portions of programs
- Bag of tasks: implementation of recursive, generally, but also can be used for iterative

Data Parallel Algorithms

- Execute identical code on different parts of a data structure
  - Usually we mean “SPMD algorithms”, which stands for Single Program Multiple Data
  - Data Parallel implies barrier after every instruction (arose from programming SIMD architectures; recall SIMD is Single Instruction Multiple Data)
  - SPMD allows barriers at arbitrary points (arose from MIMD architectures)
    - It is a relaxation of SIMD

Picture: finding the sum of an array in parallel (SPMD program)

Finding the sum of an array in parallel (SPMD program)

```
int sum[n], old[n], a[n]  // Array a initialized to arbitrary values
co i := 0 to n-1
  int d = 1
  sum[i] = a[i]
  while (d < n) {
     old[i] = sum[i]
     barrier                                  Why?
     if (i – d >= 0)
        sum[i] = old[i-d] + sum[i]
     barrier                                  Why?
     d = d * 2
   }
oc
```

Picture: Jacobi Iteration