CMPS 2200

Introduction to Algorithms

Lecture 2: Parallelism

Today's agenda:

- Motivate why we are using parallel algorithms
- Overview of how to analyze parallel algorithms

What is parallelism? (aka parallel computing)

ability to run multiple computations at the same time

Why study parallel algorithms?

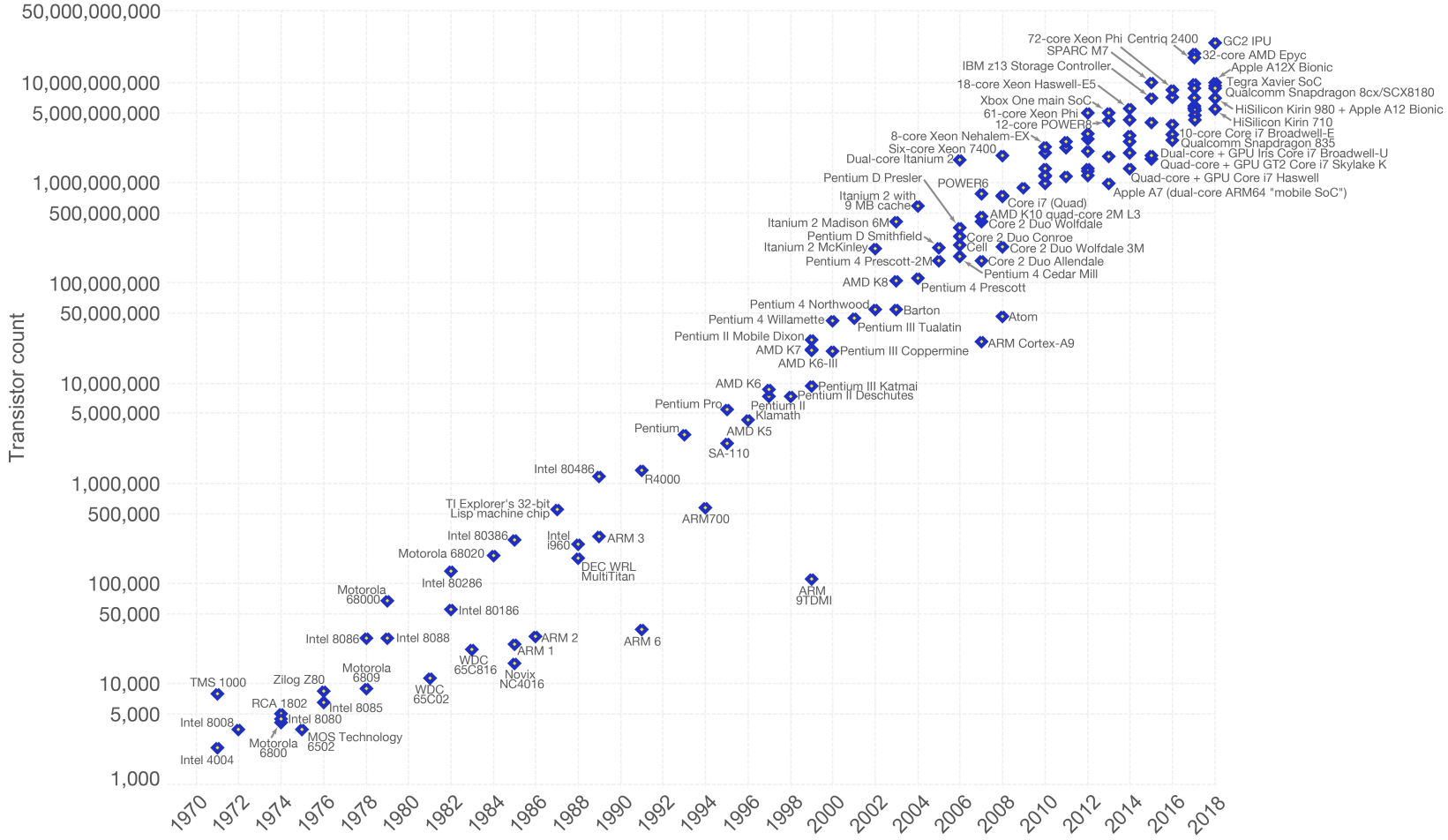
- faster
- lower energy usage
  - performing a computation twice as fast sequentially requires roughly eight times as much energy
  - energy consumption is a cubic function of clock frequency
- better hardware now available
  - multicore processors are the norm
  - GPUs (graphics processor units)

E.g., more than **one million** core machines now possible:
SpiNNaker (Spiking Neural Network Architecture), University of Manchester

supporting Moore's Law since 2004

Moore’s Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Example: Summing a list

Summing can easily be parallelised by splitting the input list into two (or k) pieces.

In [2]:
```
def sum_list(mylist):
    result = 0
    for v in mylist:
        result += v
    return result

sum_list(range(10))
```

Out[2]:
```
45
```

becomes

In [3]:
```
from multiprocessing.pool import ThreadPool

def parallel_sum_list(mylist):
    result1, result2 = in_parallel(
        sum_list, mylist[:len(mylist)//2],
        sum_list, mylist[len(mylist)//2:]
    )
    # combine results
    return result1 + result2

def in_parallel(f1, arg1, f2, arg2):
    with ThreadPool(2) as pool:
        result1 = pool.apply_async(f1, [arg1])  # launch f1
        result2 = pool.apply_async(f2, [arg2])  # launch f2
        return (result1.get(), result2.get())   # wait for both to finish

parallel_sum_list(list(range(10)))
```

Out[3]:
```
45
```

- How much faster should parallel version be?
- How much energy is consumed?

`parallel_sum_list` is twice as fast `sum_list` with same amount of energy

...almost. This ignores the **overhead** to setup parallel code and communicate/combine results.

$O(\frac{n}{2}) + O(1)$
$O(1)$ to combine results

some current state-of-the-art results:

application	sequential	parallel (32 core)	speedup
sort 10^7 strings	2.9	.095	30x
remove duplicated from 10^7 strings	.66	.038	17x
min. spanning tree for 10^7 edges	1.6	.14	11x
breadth first search for 10^7 edges	.82	.046	18x

The **speedup** of a parellel algorithm P over a sequential algorithms S is:

$$speedup(P, S) = \frac{T(S)}{T(P)}$$

Parallel software

So why isn't all software parallel?

dependency

> *The fundamental challenge of parallel algorithms is that computations must be **independent** to be performed in parallel. Parallel computations should not depend on each other.*

What should this code output?

In [4]:
```
import threading

total = 0

def count(size):
    global total
    for _ in range(size):
        total += 1

def race_condition_example():
    global total
    in_parallel(count, 100000,
                count, 100000)
    print(total)

race_condition_example()
```

```
100000
```

Counting in parallel is hard!

- motivates functional programming (next class)

This course will focus on:

- understanding when things can run in parallel and when they cannot
- algorithm, not hardware specifics (though see CMPS 4760: Distributed Systems)
- runtime analysis

Analyzing parallel algorithms

***work**: total number of primitive operations performed by an algorithm*

- For sequential machine, just total sequential time.
- On parallel machine, work is divided among P processors

***perfect speedup**: dividing W work across P processors yields total time $\frac{W}{P}$*

***span**: longest sequence of dependencies in computation*

- *time to run with an infinite number of processors*
- *measure of how "parallelized" an algorithm is*
- *also called: critical path length or computational depth*

intuition:
work: total energy consumed by a computation
span: minimum possible time that the computation requires

work: T_1 = time using one processor
span: T_∞ = time using ∞ processors

parallelism = $\frac{T_1}{T_\infty}$

maximum possible speedup with unlimited processors

What is work and span of `parallel_sum_list` algorithm using n threads?

In [5]:
```
def in_parallel_n(tasks):
    """
    generalize in_parallel for n threads.

    Params:
      tasks: list of (function, argument) tuples to run in parallel

    Returns:
      list of results
    """
    with ThreadPool(len(tasks)) as pool:
        results = []
        for func, arg in tasks:
            results.append(pool.apply_async(func, [arg]))
        return [r.get() for r in results]


def parallel_n_sum_list(mylist):
    results = in_parallel_n([ (sum_list, [v]) for v in mylist ])
    # combine results...looks familiar...
    result = 0
    for v in results:
        result += v
    return result

parallel_n_sum_list(range(10))
```

Out[5]:
```
45
```

- work: $O(n)$
- span: $O(n)$

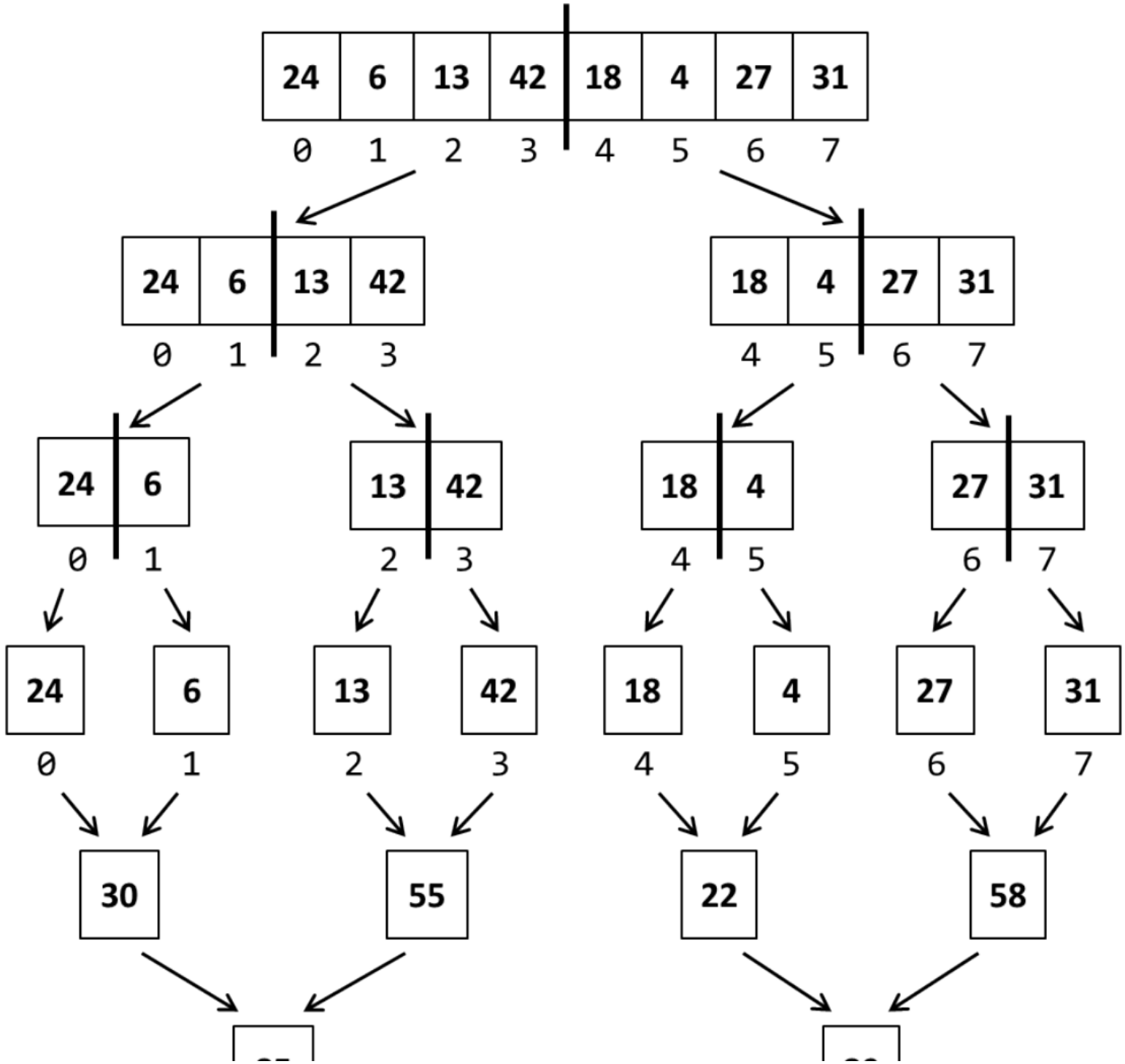
oops that didn't work...

can we do better?

idea:

- let threads create threads recursively
- parallelize combination steps

divide and conquer


24 6 13 42 18 4 27 31
0 1 2 3 4 5 6 7
24 6 13 42
0 1 2 3
18 4 27 31
4 5 6 7
24 6
0 1
13 42
2 3
18 4
4 5
27 31
6 7
24
0
6
1
13
2
42
3
18
4
5
27
6
31
7
30
55
22
58

In [6]:
```
# recursive, serial
def sum_list_recursive(mylist):
    if len(mylist) == 1:
        return mylist[0]
    return (
        sum_list_recursive(mylist[:len(mylist)//2]) +
        sum_list_recursive(mylist[len(mylist)//2:])
    )

sum_list_recursive(range(10))
```

Out[6]: 45

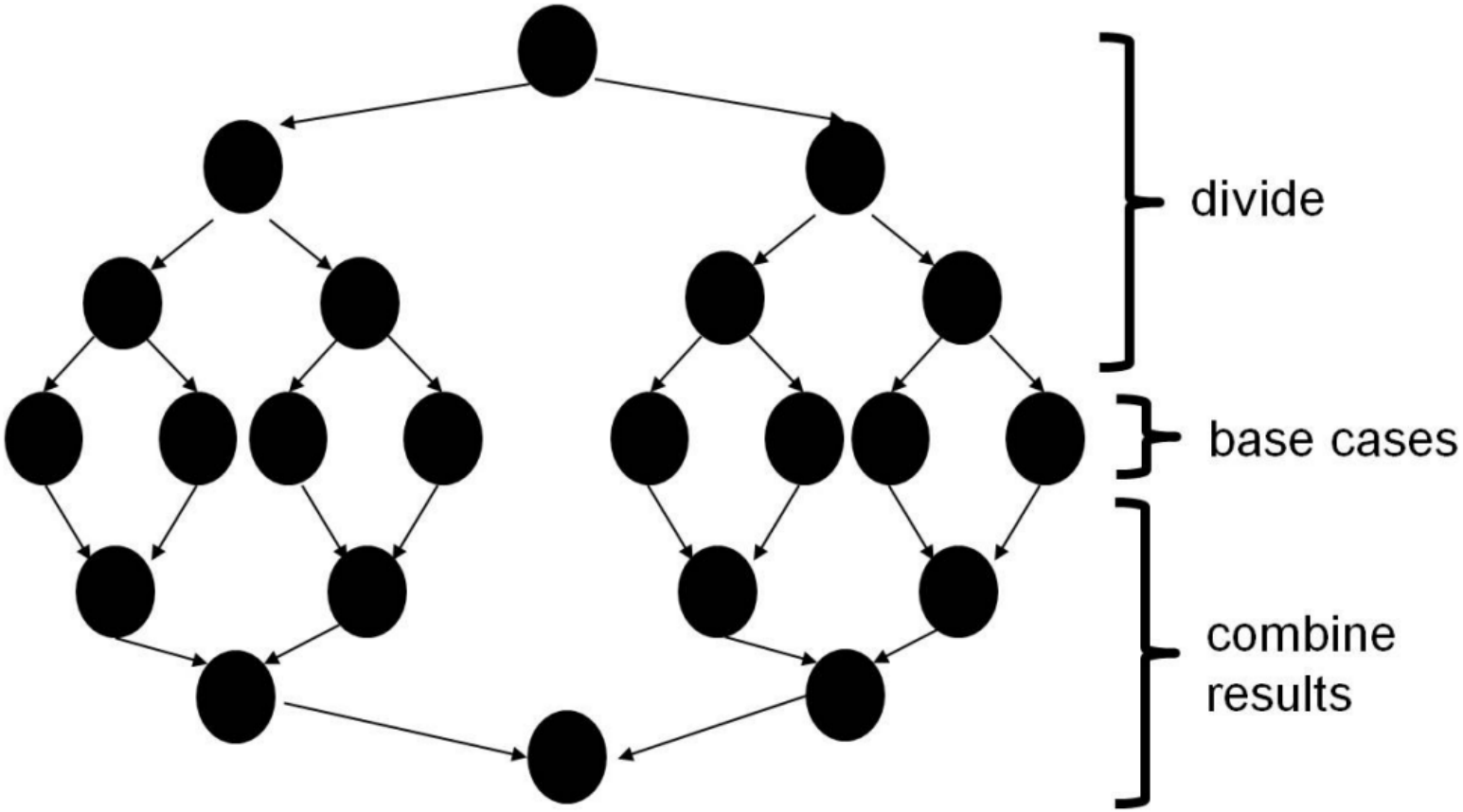
In [7]:
```
# recursive, parallel
def sum_list_recursive_parallel(mylist):
    if len(mylist) == 1:
        return mylist[0]

    # each thread spawns more threads
    result1, result2 = in_parallel(
        sum_list_recursive, mylist[:len(mylist)//2],
        sum_list_recursive, mylist[len(mylist)//2:]
    )
    return result1 + result2

sum_list_recursive(range(10))
```

Out[7]: 45

Computation Graph



source (https://homes.cs.washington.edu/~djg/teachingMaterials/spac/sophomoricParallelismAndC

- Directed-acyclic graph (DAG) where
    - Each node is a unit of computation ($O(1)$)
    - An edge is a **computational dependency**
        - Edge from node A and B means A must complete before B begins

***work**: total number of primitive operations performed by an algorithm*
***span**: longest sequence of dependencies in computation*

So, what is work and span for `sum_list_recursive_parallel` ?

work: number of nodes

span: length of longest path

What is the height of a balanced binary tree with n nodes?

$O(log_2\ n)$

- Number of leaf nodes in a perfect binary tree is 2^h.
- To add an array of length n, the computation graph has $2 * 2^{(\log_2 n)} = 2n$ nodes.

so,

work: $2n \in O(n)$
span: $2 \log_2 n \in O(\log_2 n)$

parallelism: $O(\frac{n}{\log_2 n})$ = **exponential speedup**

Work Efficiency

> *A parallel algorithm is (asymptotically) **work efficient** if the work is asymptotically the same as the time for an optimal sequential algorithm that solves the same problem.*

Since $T_{\text{sequential}} = T_1 = O(n)$, our parallel algorithm is work efficient

Using p processors

Of course, we don't have ∞ processors. What if we only have p?

It can be shown that (and will be shown in lecture 4):

$$T_p \leq \frac{T_1}{p} + T_\infty$$
$$T_p \in O(\frac{T_1}{p} + T_\infty)$$

So, assuming $\log_2 n$ processors, our parallel sum algorithm has time

$$O(\frac{n}{\log_2 n} + \log_2 n)$$

compared to $O(n)$ of serial algorithm.

Amdahl's Law

As seen above, some parts of parallel algorithms are sequential.

We expect that the bigger the "sequential" part of the parallel algorithm is, the lower its speedup.

How bad is a little bit of sequential code?

- Let T_1 be the total time of the parallel algorithm on one processor. Let's set this value to 1.
- Let S be the amount of time that cannot be parallelized.
- Assume (generously) that the remaining time (1-S) gets perfect speedup using p processors.
- Then we have

$$T_1 = S + (1 - S) = 1$$
$$T_P = S + \frac{(1-S)}{p}$$

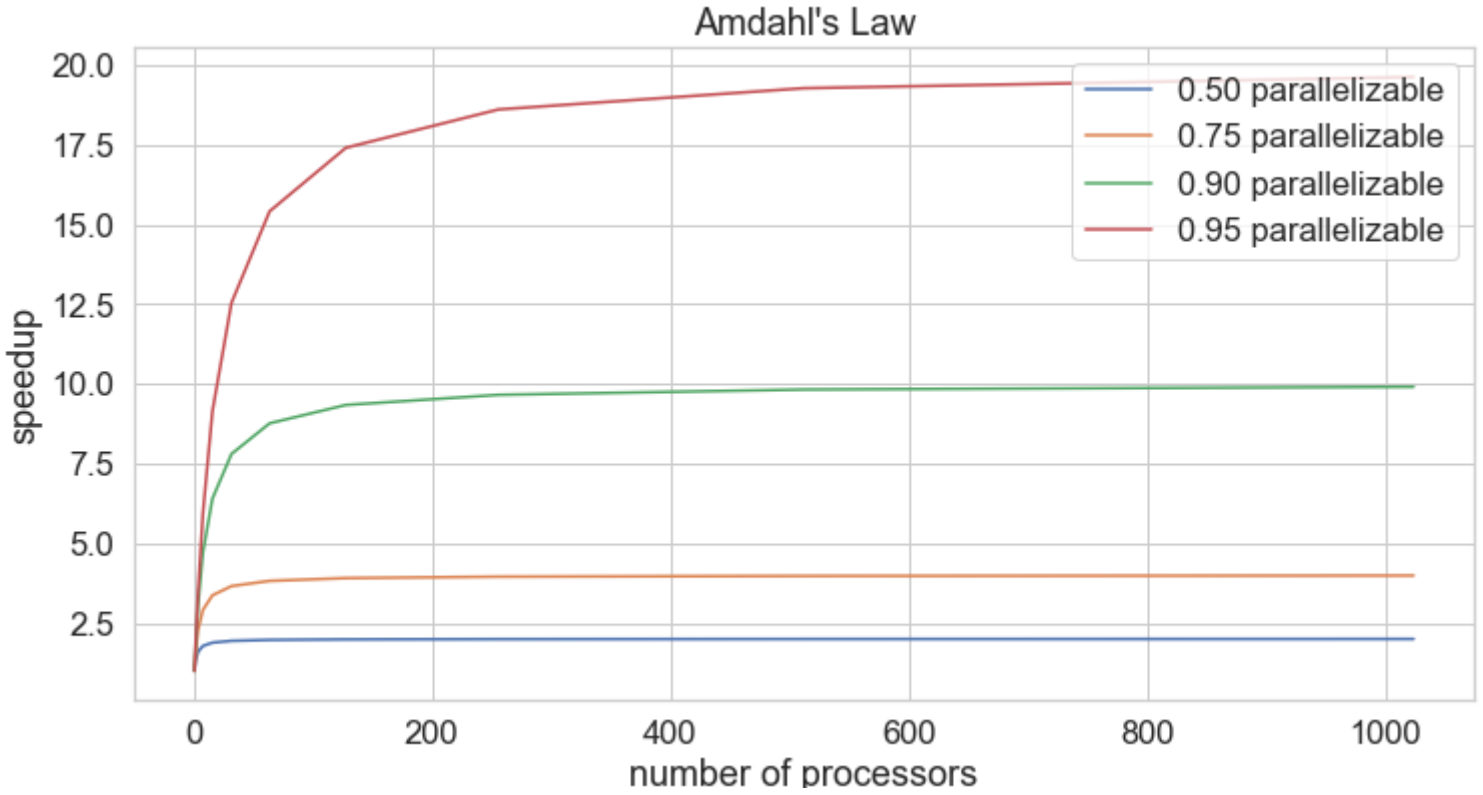
Amdahl's Law

$$\frac{T_1}{T_p} = \frac{1}{S + \frac{1-S}{p}}$$

speedup using p processors is limited by the fraction of the algorithm that is parallelizable.

In [8]:

```
# plot amdahl's law
processors = [1,2,4,8,16,32,64,128,256,512,1024]
parallel_portions = [.5, .75, .9, .95]
plt.figure()
for parallel_portion in parallel_portions:
    S = 1 - parallel_portion
    speedups = [1 / (S + (1-S)/p) for p in processors]
    plt.plot(processors, speedups, label='%.2f parallelizable' % parallel_portion)
plt.legend()
plt.ylabel('speedup')
plt.xlabel('number of processors')
plt.title("Amdahl's Law")
plt.show()
```



If 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times.

If 50% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 2 times.

even if you have a billion processors!

Reasons for hope:

As number of processors grow, span becomes more important than work.

- scalability matters more than performance
- we can tradeoff work and span in algorithm choice

Allows us to do new things we couldn't do before:

- consider **computer graphics**: rendering many pixels in parallel allows more accurate images