§ Parallelism and its limits

E.g., matrix multiplication

```c
int A[DIM][DIM], /* src matrix */
    B[DIM][DIM], /* src matrix */
    C[DIM][DIM]; /* dest matrix */

/* C = A x B */
for (int i=0; i<DIM; i++) {
    for (int j=0; j<DIM; j++) {
        C[i][j] = 0;
        for (int k=0; k<DIM; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

Run time, with DIM=256, 500 iterations:

```
$ time ./a.out
./a.out 33.59s user
         0.00s system
           99% cpu
        33.596 total
```

E.g., 1 thread per result cell

```c
void run_with_thread_per_cell() {
    pthread_t ptd[DIM][DIM];
    int index[DIM][DIM][2];

    for(int i = 0; i < DIM; i ++)
        for(int j = 0; j < DIM; j ++) {
            index[i][j][0] = i;
            index[i][j][1] = j;
            pthread_create(&ptd[i][j], NULL,
                           row_dot_col,
                           index[i][j]);
        }

    for(i = 0; i < DIM; i ++)
        for(j = 0; j < DIM; j ++)
            pthread_join( ptd[i][j], NULL);
}
```

```c
void row_dot_col(void *index) {
    int *pindex = (int *)index;
    int i = pindex[0];
    int j = pindex[1];

    C[i][j] = 0;
    for (int x=0; x<DIM; x++)
        C[i][j] += A[i][x] * B[x][j];
}
```

```
$ time ./a.out
./a.out 115.69s user
        380.30s system
           149% cpu
        5:32.17 total
```

what happened?

E.g., variable # threads

```
void run_with_n_threads(int num_threads) {
    pthread_t tid[num_threads];
    int tdata[num_threads][2];
    int n_per_thread = DIM/num_threads;

    for (int i=0; i<num_threads; i++) {
        tdata[i][0] = i*n_per_thread;
        tdata[i][1] = (i < num_threads)
                      ? ((i+1)*n_per_thread)-1
                      : DIM;
        pthread_create(&tid[i], NULL,
                       compute_rows,
                       tdata[i]);
    }
    for (int i=0; i<num_threads; i++)
        pthread_join(tid[i], NULL);
}
```

```
void *compute_rows(void *arg) {
    int *bounds = (int *)arg;
    for (int i=bounds[0]; i<=bounds[1]; i++) {
        for (int j=0; j<DIM; j++) {
            C[i][j] = 0;
            for (int k=0; k<DIM; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

E.g., variable # threads

```
void run_with_n_threads(int num_threads) {
    pthread_t tid[num_threads];
    int tdata[num_threads][2];
    int n_per_thread = DIM/num_threads;

    for (int i=0; i<num_threads; i++) {
        tdata[i][0] = i*n_per_thread;
        tdata[i][1] = (i < num_threads)
                       ? ((i+1)*n_per_thread)-1
                       : DIM;
        pthread_create(&tid[i], NULL,
                       compute_rows,
                       tdata[i]);
    }
    for (int i=0; i<num_threads; i++)
        pthread_join(tid[i], NULL);
}
```

```
$ time ./a.out -nthreads=1
./a.out 35.59s user 0.01s system 99% cpu 35.617 total

$ time ./a.out -nthreads=2
./a.out 35.72s user 0.00s system 198% cpu 18.012 total

$ time ./a.out -nthreads=4
./a.out 37.48s user 0.03s system 389% cpu 9.641 total

$ time ./a.out -nthreads=8
./a.out 65.89s user 0.09s system 744% cpu 8.862 total

$ time ./a.out -nthreads=16
./a.out 67.02s user 0.22s system 766% cpu 8.776 total

$ time ./a.out -nthreads=32
./a.out 67.97s user 0.46s system 758% cpu 9.017 total

$ time ./a.out -nthreads=64
./a.out 68.48s user 0.99s system 763% cpu 9.101 total

$ time ./a.out -nthreads=128
./a.out 73.97s user 2.79s system 745% cpu 10.297 total

$ time ./a.out -nthreads=256
./a.out 67.52s user 4.23s system 639% cpu 11.224 total
```

test system = 8 CPUs, 2 cores per CPU

Embarrassingly parallelizable

- Matrix multiplication is a problem very well suited to parallelization
  - Solution can be easily broken into pieces that may run in parallel, with almost no interdependencies
  - Very little work needed to improve performance
    - I.e., would be embarrassing not to do it!
  - Not representative of most real world problems

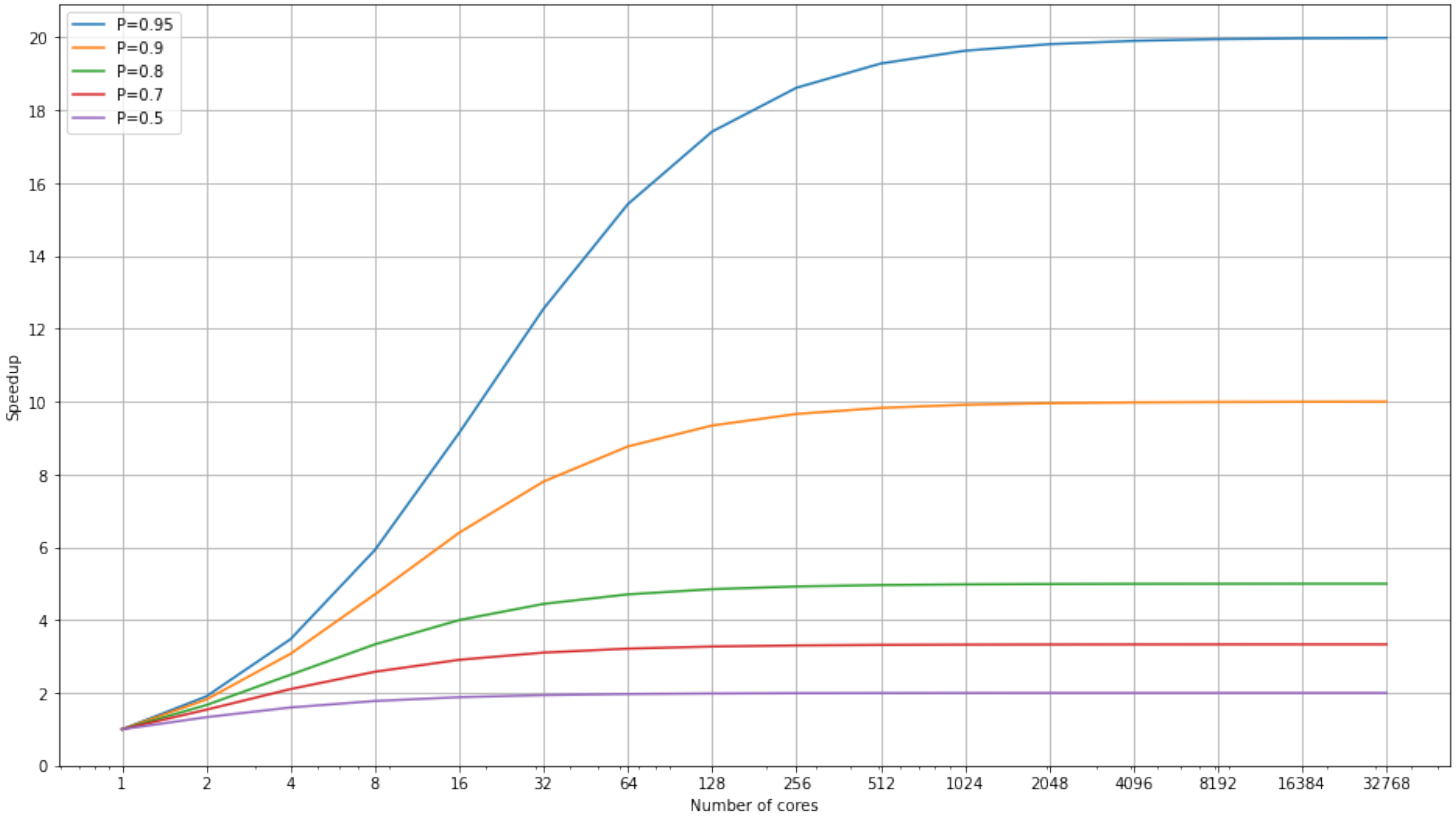
Parallelizability

- The extent to which we can parallelize a task imposes a theoretical cap on the potential speedup we might aim/hope for
- Two well known laws that relate parallelizability to potential speedup:
  - Amdahl's Law
  - Gustafson's Law

Amdahl's Law

- The maximum speedup of a task can be computed based on:
  - P : the parallelizable fraction of program, $0 \leq P \leq 1$
  - N : # of threads that can be run in parallel (number of cores)
- Maximum speedup $S_A(N) = \dfrac{1}{\frac{P}{N} + (1 - P)}$
  - As $P \to 1$, $S_A \to N$
  - As $N \to \infty$, $S_A \to \dfrac{1}{1 - P}$

Amdahl's Law



P=0.95
P=0.9
P=0.8
P=0.7
P=0.5
Speedup
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
Number of cores

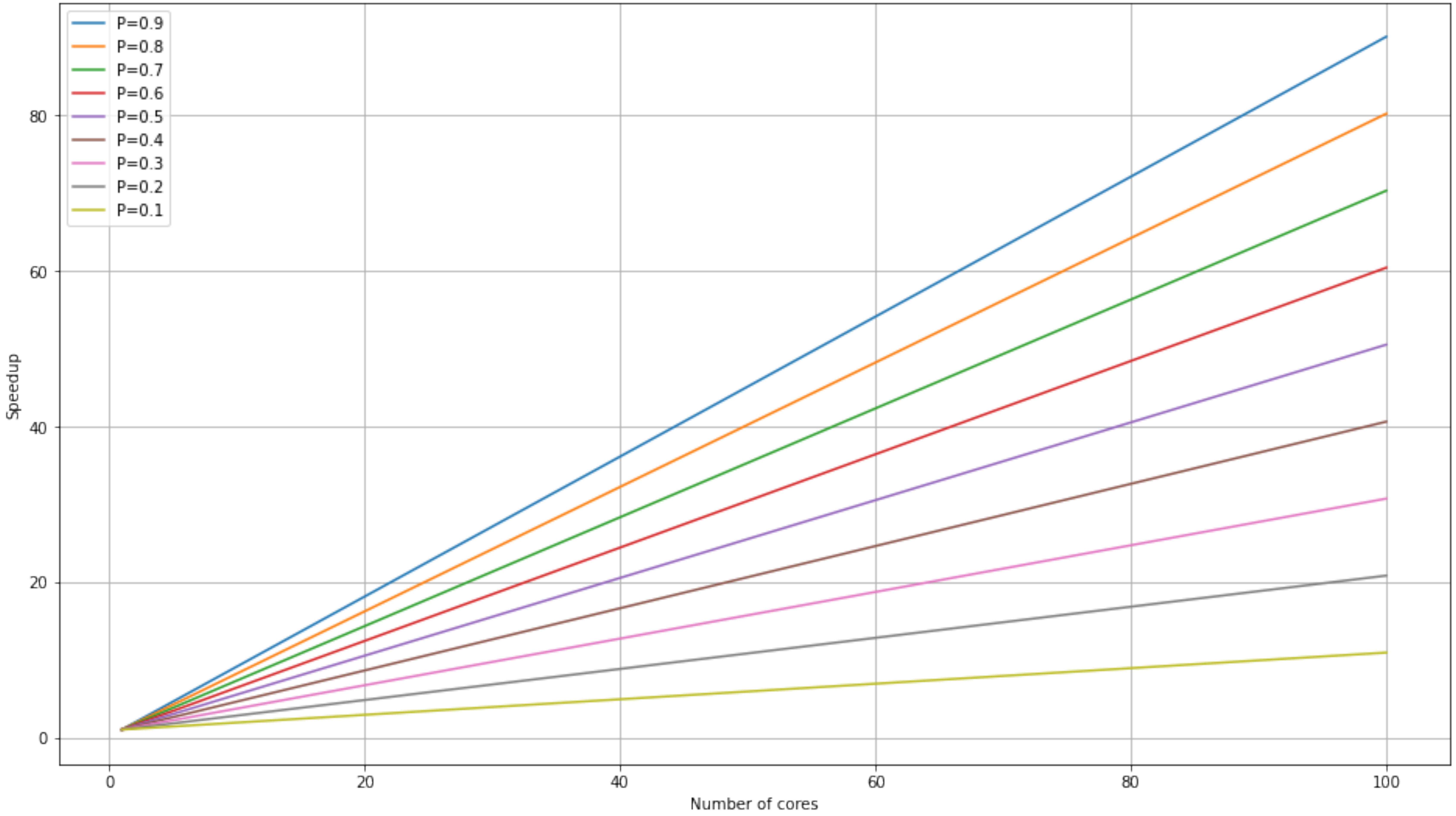
Amdahl's Law: pessimistic?

- Built in to Amdahl's Law is the implication that as the size of a given problem increases, it has a *fixed parallelizable fraction*
  - Is this true?
    - Consider a renderer that is run on models of increasing detail/ resolution, or a game AI that can search deeper in a game tree
- We might argue that as we have more powerful computers we wish to throw larger / higher resolution problems at them
  - This could increase the parallelizable fraction of the workload!

Gustafson's Law

- Central argument: we tend to scale a workload to complete in the same amount of time, regardless of the processing power available
- Maximum speedup $S_G(N) = 1 - P + NP$
  - If $P = 0$, $S_G = 1$
  - As $P \rightarrow 1$, $S_G \rightarrow \mathrm{N}$
    - I.e. speedup is linear with respect to the # of cores

Gustafson’s Law



Amdahl vs. Gustafson

- Amdahl's law is appropriate if the problem or workload is fixed, and we are looking to estimate the maximum speedup via parallelization
- Gustafson's law has rosy implications for big data / data science
  - But not all datasets naturally increase in resolution — beware!
- Both stress the import of maximizing parallel fraction of workload

§ Writing concurrent programs

Goals & Challenges

- Identify opportunities for parallelization
  - Not always obvious
  - Domain specific — we will mostly ignore this part!
- Identify thread interdependencies & potential ramifications
  - Insidious problem: *race conditions*
  - Ensure correctness while maximizing concurrency

Revisiting increment bug

```c
int counter = 0;

void *inc(void *num) {
    for (int i=0; i<10000; i++) {
        counter += 1;
    }
    printf("Thread %ld counter = %d\n",
           pthread_self(), counter);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid;
    for (int i=0; i<5; i++){
        pthread_create(&tid, NULL, inc, NULL);
        printf("Created thread %ld\n", tid);
    }
    pthread_exit(NULL); // terminate main thread
    return 0; // never get here!
}
```

```
Created thread 139949404641024
Created thread 139949396248320
Created thread 139949387855616
Thread 139949396248320 counter = 20035
Created thread 139949379462912
Thread 139949404641024 counter = 10000
Created thread 139949371070208
Thread 139949387855616 counter = 20833
Thread 139949379462912 counter = 28523
Thread 139949371070208 counter = 34961
```

Problem: shared counter

- What is the final change in counter after completing both threads?

Thread A

```
a1  counter += 1
```

Thread B

```
b1  counter += 1
```

- Expected = +2, but may be different!

Non-atomic operations

- Factoring in machine-level granularity (assuming thread-local registers):

Thread A

```
a1  lw   (counter), %r0
a2  add  $1, %r0
a3  sw   %r0, (counter)
```

Thread B

```
b1  lw   (counter), %r0
b2  add  $1, %r0
b3  sw   %r0, (counter)
```

- Possible change in counter = +1 **or** +2
  - Actual result is *non-deterministic*

Race conditions & Critical sections

- **Race condition(s)** exist when the result of a program is dependent on the ordering of concurrent operations
- Shared resource(s) are the problem
  - More specifically, *concurrent mutability* of shared resources
- Code that accesses shared resource(s) = **critical sections**
- Concurrent program often requires the careful **synchronization** of critical section code to avoid race conditions

E.g., critical section

```
int counter = 0;

void *inc(void *num) {
    for (int i=0; i<10000; i++) {
        counter += 1;
    }
    printf("Thread %ld counter = %d\n",
           pthread_self(), counter);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid;
    for (int i=0; i<5; i++){
        pthread_create(&tid, NULL, inc, NULL);
        printf("Created thread %ld\n", tid);
    }
    pthread_exit(NULL); // terminate main thread
    return 0; // never get here!
}
```

← critical section
how to fix this code?

Lock-based synchronization

- A common form of synchronization is to ensure critical sections are only executed by one thread at a time
- One mechanism for ensuring this is a shared **lock** object
  - Can only be allocated to one thread at a time
    - I.e., mutually exclusive allocation (mutex)
  - Designed to be "thread-safe"
    - Multiple threads may try to acquire it concurrently, but its implementation ensures only one will succeed