TODAY IN CS! BERKELEY POSTDOC IMPROVES UNDERSTANDING OF MATRIX MULTIPLICATION!
Virginia Vassilevska Williams used convex optimization to tighten the known worst-case upper bound on the complexity of $n$-by-$n$ mat. mult. from $O(n^{2.374})$ to $O(n^{2.3727})$. A great theoretical result via a method that suggests even tighter bounds exist & can be found soon!

http://www.scottaaronson.com/blog/?p=839
Parallel Processing: Familiar Obstacles

- Many hands make light work!
  - Execute instructions simultaneously
- But parallelization is hard....
  - More workers? More overhead!
  - Shared data is hard to coordinate
  - Whine whine whine whine whine
But once you have a parallel system...

... (after handling synchronization...)

... after finding a parallel algorithm...

... after finding a memory solution...

... and after handling worker failures)...

... just add more cores forever and win! ... r-right?
Array Copying Example

```c
for(i = 0; i < 100; i++) // With one core...
    y[i] = x[i];       // <-- 100 instr.
printf("DONE");     // <-- 10 instr.
```

- Takes about 110 instructions to run serially
  - Assume magical AMAT of 1 cycle
  - Assume magical cost-free 0-cycle comparator/increment
  - `printf()` is legacy code -- must be run serially
- **IF** we set up a successful parallelization scheme (threading?), each loop iteration could be run in parallel
  - Assume magical, no-collisions caching
  - Assume no increased work for each new thread added
Array Copying Example

```c
for(i = 0; i < 100; i++)
    y[i] = x[i];
printf("DONE");  // <-- 10 instr.
```

One core takes 110 instructions...

<table>
<thead>
<tr>
<th>With this many cores...</th>
<th>... loop takes...</th>
<th>... printing takes...</th>
<th>... totaling...</th>
<th>... for a speedup of:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>50 instr.</td>
<td>10 instr.</td>
<td>60 instr.</td>
<td>1.83x</td>
</tr>
<tr>
<td>4</td>
<td>25 instr.</td>
<td>10 instr.</td>
<td>35 instr.</td>
<td>3.14x</td>
</tr>
<tr>
<td>8</td>
<td>13 instr.</td>
<td>10 instr.</td>
<td>23 instr.</td>
<td>4.78x</td>
</tr>
</tbody>
</table>

2 to 4: 1.71x
4 to 8: 1.52x
Array Copying, Graphically

More cores can speed up the parallelizable copy loop, but not the serial-only print routine.

Even in the limit of a gazillion cores, need at least one instruction to compute the loop.

**BIGGEST POSSIBLE SPEEDUP:** $\frac{110}{11} = 10x$
Amdahl's Law

\[ f(N) = \frac{1}{(1 - P) + \frac{P}{N}} \]

- \( P \) := "Percentage" of code which is parallelizable
- \( N \) := Number of cores used
- \( f(N) \) := Amount of speedup code gains using \( N \) cores

suggests a maximum possible speedup:

\[ \lim_{N \to \infty} f(N) = \lim_{N \to \infty} \frac{1}{(1 - P) + \frac{P}{N}} = \frac{1}{1 - P} \]
Amdahl's Law

\[ f(N) = \frac{1}{(1 - P) + \frac{P}{N}} \]

For our copying example, \( P = \frac{100}{110} = \frac{10}{11} \) suggesting an asymptote of \( f(a \text{ gazillion}) = 11 \).
Amdahl's Law's Assumptions

- **No contention for shared resources!**
  - All threads have equal access to caches, memory, IO, etc.

- **No per-thread overhead!**
  - Adding more threads to the parallel sections doesn't add more work for the serial section

- **No Pipelining!**
  - Some apps can send partial solutions off to one parallel thread at a time

(Also, let's just round off quantization, too!)
Amdahl: TO THE CLOUD

- Hourly computer rental
  - Speedup of 2x?
    - Twice the revenue!
    - Same rental fee!
- "Elastic" cluster size
  - Pay $x for 1 core?
  - Via virtualization: Pay $kx for k cores!
- Hardware price points
  - m1.small, $0.085/hr
    - 1x ~1.2 GHz
    - 1.7 GB RAM
  - c1.xlarge, $0.68/hr
    - 8x ~3 GHz
    - 7 GB RAM

(Most of these cost structures also hold even if you build your own rig -- more cores? Higher power bill!)
Amdahl: Costs and Benefits

- Benefits of more cores rise as Amdahl's Law
  - $f(N)$ speedup? $f(N)$ more customers served!
- Costs of more cores rises linearly in $N$
  - Steeper slope = cheap customers, pricey cores, or both.
Amdahl: Costs and Benefits

- Profit = Benefits - Costs; should *at least* be positive
  - Clear bounds on N for \( P = 50\% \) and \( P = 70\% \)
  - Note that both are quite asymptotic by that point anyway
- Insufficient to just have *positive* profit -- want the *maximum*!
Amdahl: Marginal Costs and Benefits

- Take the first derivative of both benefit and cost
- Find the point right before adding one more machine marginally costs more than it marginally benefits
Amdahl: **Marginal** Costs and Benefits

Amdahl's Marginal Costs v. Marginal Benefits

- P = 50%
  - Optimal N = 4
- P = 70%
  - Opt. N = 7
- P = 90%
  - Opt. N = 21

- Optimal N can occur quite a bit before asymptote kicks in
- If marginal cost rises a little, can cause Opt. N to drop a lot
Sum-of-Squares Example

```
s = 0;
for(i = 0; i < 100; i++)
    s += x[i]**2;    // 2 Inst per loop
```

- Each iteration depends on the result of the iteration before!
- As written, unparallelizable:
  - P = 0 %
  - max f(N) = 1/(1-P) = 1x speedup, max.
  - Have to run all 200 instructions serially!
  - DOOOOOOM!
Sum-of-Squares: One Good Idea

```c
s = 0;
for(i = 0; i < 100; i++)
  y[i] = x[i]**2;  // square
for(i = 0; i < 100; i++)
  s += y[i];    // accumulate
```

- Good idea: Break the loop into 2!
  - First square, then sum
  - Use more memory to save time
- First loop now parallelizable:
  - $P = 50\%$
  - $\text{max } f(N) = 1/(1-P) = 2x$ speedup, max.
  - Even 2x speedup requires a gazillion cores (a gazillion dollars).
  - dooooooooom.
Sum-of-Squares: One GREAT Idea

```c
s = 0;
for(i = 0; i < 100; i++)
    y[i] = x[i]**2;
parAccum(y,100); //parallel accumulator
```

- GREAT idea: build a parallelizable accumulator
  - Sum Reduction from 10.14.11's lecture is our friend here!
- How close can we get to full parallelizability?
  - The better we build `parAccum`, the closer P gets to 100%
Sum-of-Squares: \( \text{parAccum}(y, 100) \)

Level \( \log_2(N-1) \): 1 Instr.

Level 2: 1 Instr.

Level 1: \( \frac{100}{N} - 1 \) Instr.

TOTAL = \( \frac{100}{N} + \log_2(N-1) \) - 2 steps to complete.
Sum-of-Squares: One GREAT Idea

\[ s = 0; \]
\[ \text{for}(i = 0; i < T; i++) \] // squaring loop
\[ y[i] = x[i]^{**2}; \]
\[ \text{parAccum}(y, T); \] //parallel accumulator

- N cores provide:
  - Linear reduction in squaring loop
  - Almost linear reduction in accumulation
  - For large T, smallish N, it's awful close to \( P = 100\% \)
EC2 Usage

- Regular troughs at mid-day: Perfect for AWS!
- Peak usage: 292 instances
- Median usage: 52
- Mean usage: 81.44
- About $2,400!