# CSCI 551
# Numerical and Parallel Programming

*Lecture 4-1 – Overall Resource Use and Algorithms that are and are not Embarrassingly Parallel*


February 11, 2025

# Announcements

- Quiz #1 – don't forget - take this week before class next week
  - Solutions will be posted on Canvas Tuesday next week
  - You can take up to 3x and scores are averaged
  - Feedback can help you improve
  - Covers material completed through Week-3
    - Notes
    - Reading
    - Starter code provided for Ex #1 and covered in class notes
- Grading on Ex #1 in progress – will return this week
- Turn in Ex #2 and use 3-days grace for good reasons if required
- Don't forget to verify your account for cluster use BEFORE Assignment #3 – Notes to help you

# Exercise #1 Grading

- Grades will be posted before end of week (Friday)
  - At least 24 hours before Ex #2 due date OR sooner!
  - I make sure I'm not holding more than 1 assignment at all times
- Ideal average for class is 70-80+ on exercises
- Attempt all problems first week, question, improve analysis, verification, and reporting second week
- **Not attempted, incomplete reports are lowest scores**
- **Complete, Clear**, Consistent, Correct!

# Parallel and Numerical Careers

- High Performance Computing (SC)
  - IT – Scaling clusters and nodes, Storage, Networking
    - 10G or 100G to 400G-Tbit Ethernet, Infiniband, Cluster Interconnection
    - Fiber channel – Qlogic, Broadcomm, Brocade
    - InfiniBand – Qlogic, Mellanox
    - GP-GPU - NVIDIA
  - Systems Programming for HPC
    - Scientific computing (numerical and parallel)
    - OS
    - Filesystems
    - Networking
  - Applications
    - Algorithms and Data Analytics
    - Domain specific
      - Business Intelligence & Machine Learning,
      - Digital Cinema & Computer Vision,
      - Scientific, e.g., ANSYS Multi-physics,
      - Geophysics,
      - Cryptanalysis, etc.
    - Data centers

Scale up – increase shared memory CPU cores per node

Scale out – increase nodes in cluster and size and speed of fabric interconnection between them

**Software, IT & Engineering Job Fair**(opens in new window)- Wednesday, February 19, 2025, 11 a.m.-3 p.m., BMU Auditorium

https://www.csuchico.edu/careers/employers/career-fair-information/index.shtml

# Questions on Large Semi-Prime Factoring – simple start

- Keep it simple
  - Use a simple sieve to generate primes between 0 … 1 billion as unsigned int
  - Should be 50 million primes (50,847,534 according to https://www.dcode.fr/prime-number-pi-count) – test with sieve (agree?)
  - Store them in an array (about 200 MB in size)
- Compute semi-prime that is product of any 2 primes in your array (randomly selected)
- Search for SP = $P_1$ x $P_2$ by looking for SP mod $P_n$ == 0
  - When $P_n$ is found, $P_m$ = SP / $P_n$
  - Report $P_m$ & $P_n$
  - Work on optimizations if you wish (e.g., methods learned in Math 317)
  - Make it parallel!
- Use parallel divide & conquer
- Analyze parallel speed-up
- Papers on cryptanalysis - here

Math 317, Cryptography as a CS/CINS/Math elective

**MATH 317 Cryptography** **4 Units**

**Prerequisite:** CSCI 111; MATH 217 or MATH 330W.

**Typically Offered:** Spring only

This is the first course in cryptography with an emphasis on public key cryptosystems, digital signature schemes, and the underlying mathematical principles on which they are based. Students implement algorithms and solve problems in programming-based assignments. Some time is devoted to getting familiar with the Python programming language and the SageMath Software system. 4 hours discussion. (022044)

**Grade Basis:** Graded

**Repeatability:** You may take this course for a maximum of 4 units

**Course Attributes:** Upper Division

# Advanced Notes on Semi-Prime Factoring

- Since DES56 broken, large key focus for encryption (SP)
  - https://en.wikipedia.org/wiki/RSA_numbers
  - https://mathworld.wolfram.com/RSANumber.html
  - **The largest cryptography-grade number that has been factored is RSA-250**
- Trap-door functions with large keys (need large primes)
  - https://en.wikipedia.org/wiki/Sieve_theory
  - https://en.wikipedia.org/wiki/Quadratic_sieve
  - https://en.wikipedia.org/wiki/General_number_field_sieve
- Lists of primes (generated by parallel computers)
  - https://en.wikipedia.org/wiki/List_of_prime_numbers
  - https://en.wikipedia.org/wiki/Prime-counting_function (density)
- Factoring large SP by Trial division (Ex #2)
  - https://en.wikipedia.org/wiki/Trial_division
  - https://en.wikipedia.org/wiki/Integer_factorization (complexity)
- Complexity Review
  - https://en.wikipedia.org/wiki/Time_complexity
  - https://en.wikipedia.org/wiki/NC_(complexity)
  - https://en.wikipedia.org/wiki/Time_complexity

Polylogarithmic:

$T(n)$ is $O((\log n)^k)$

Due to divide & conquer

Large SP and $P_1$, $P_2$ requires arbitrary precision (e.g., https://gmplib.org/)

Trap-door: SUBEX time or $2^{O(n)}$

Final Parallel Program?

# Complexity – POLY ≦ Parallel < EXP

| | | | |
|---|---|---|---|
| quadratic time | | $O(n^2)$ | $n^2$ |
| cubic time | | $O(n^3)$ | $n^3$ |
| polynomial time | P | $2^{O(\log n)} = \text{poly}(n)$ | $n^2 + n, n^{10}$ |
| quasi-polynomial time | QP | $2^{\text{poly}(\log n)}$ | $n^{\log \log n}, n^{\log n}$ |
| sub-exponential time (first definition) | SUBEXP | $O(2^{n^\epsilon})$ for all $\epsilon > 0$ | |
| sub-exponential time (second definition) | | $2^{o(n)}$ | $2^{\sqrt[3]{n}}$ |
| exponential time (with linear exponent) | E | $2^{O(n)}$ | $1.1^n, 10^n$ |
| factorial time | | $O(n)! = 2^{O(n \log n)}$ | $n!, n^n, 2^{n \log n}$ |
| exponential time | EXPTIME | $2^{\text{poly}(n)}$ | $2^n, 2^{n^2}$ |

Parallel programming:
T(n) < human time

Poly reduced to Poly(log n)
Poly degree=2 and larger
Nick's Class or NC
NC = P?

Parallel Optional:
O(1) – not needed
O(log n) – not needed
O(n) – large n (degree=1)
O(n log n) – large n

GNFS

General integer factoring!
(large keys, large n)
SP = $P_1$ x $P_2$

Quantum BQP reduces to P

- Shor's algorithm
- BQP

# Questions on Ex #2, Quest #1 - Activity

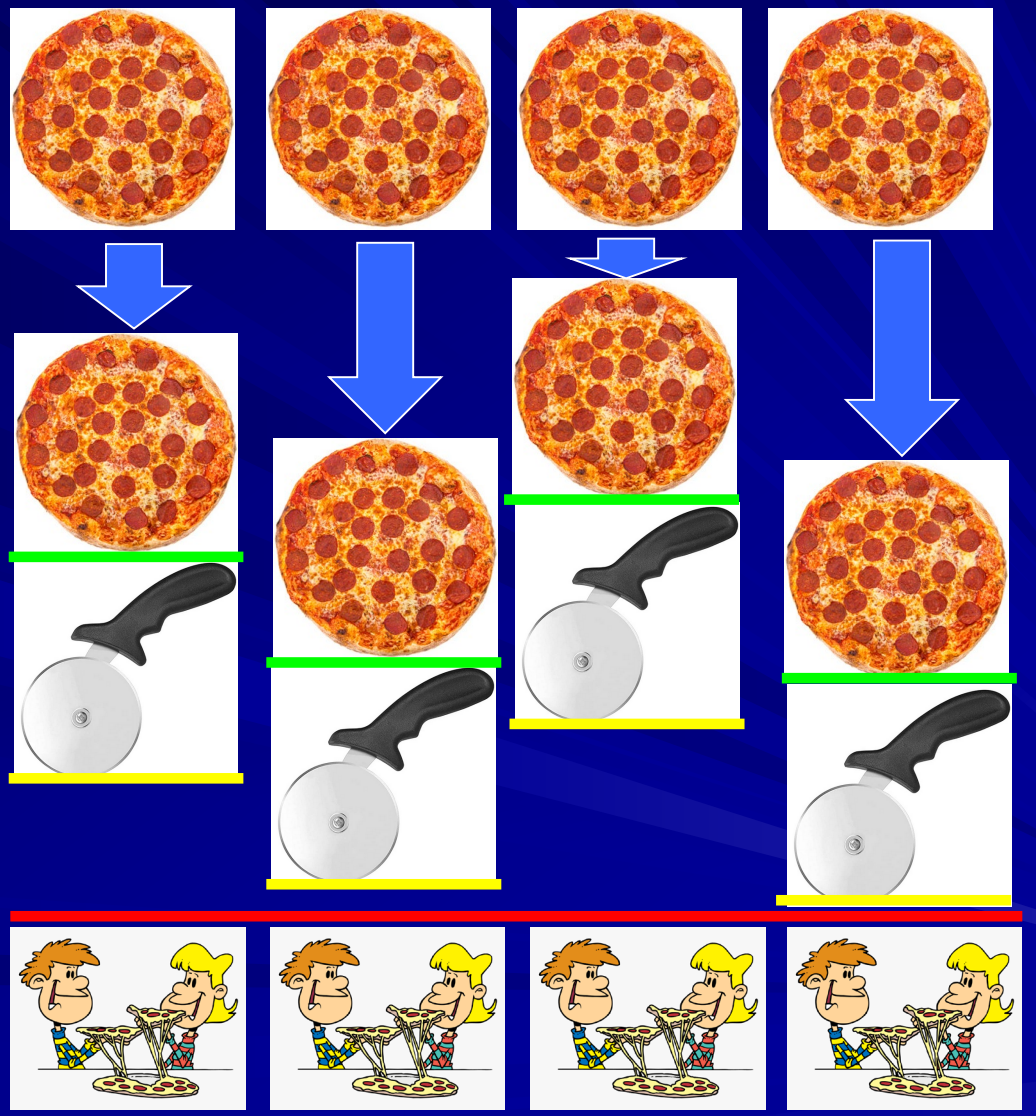
- Keep it simple (don't over complicate)
- #1a, part 1 – goal is to specify the # of threads interactively (backlog), no matter what they are doing (workload), fair or unfair (amount of work each does)
- #1a, part 2 – specify the work for each thread (worker) to do and make it equal
- #1b – Scale it, verify it based on inductive math, note whether order is deterministic and if not, why?
- #1c – for OpenMP version, explore use of reduction knowing that order of addition is flexible (commutative law)

## Parallel Scaling Pattern

1) Make # threads (workers) scalable
2) Make work each does scalable (E.g., O(n), with larger n)
3) Verify with math theory
    1) First verify for small scale, simple for sequential and parallel (eyeball it)
    2) Re-verify at larger scale simple problem (math principles)
    3) Verify at scale complex problem (does it make sense?)
4) Try various parallel methods verifying as above (which is best?)
5) Difficulty – simple map and reduce, harder?
6) Check Mr. G's law (saturates htop)
7) Check Amdahl's law – SU and % parallel

# Recall Divide and Conquer


- Map (backlog of work)
  - E.g., at least 2 threads per processing element (core or virtual core)
  - Too much backlog creates high overhead (start-up)
  - Make # of threads command line option!
  - More advanced – use POSIX message queues for workers
- Allow scheduler to balance workload over CPU cores (nodes)
- Synchronize on completion, then reduce (e.g., sums of sums after join)
- OpenMP – implicit and abstract (e.g., exit barrier)
- Pthreads – explicit with coarse grained control



Nobody eats until 4 pizzas are cut – **barrier or join**

# Notes on Big Numbers

- For Prime numbers, search up to 1 billion
  - This is still less than 32-bit address (4 billion +)
  - Check memory available with “free –h”, bit pack to save memory (1 billion / 8) locations
  - 64-bit address is much larger (16 exabytes)

C data types

Has unsigned long long int (largest)

Does not have infinite range

Can overflow (looks negative)

16 or 32-bit (int and unsigned int)
32-bit (long int and unsigned long)
64-bit (unsigned long long) – 16 exabytes

Be careful with types and overflow

https://en.wikipedia.org/wiki/C_data_types

| English Name | Power | Number | SI prefix |
|---|---|---|---|
| one | 0 | 1 | |
| ten | 1 | 10 | deca |
| hundred | 2 | 100 | hecto |
| thousand | 3 | 1,000 | kilo |
| ten thousand | 4 | 10,000 | |
| hundred thousand | 5 | 100,000 | |
| million | 6 | 1,000,000 | mega |
| ten million | 7 | 10,000,000 | |
| hundred million | 8 | 100,000,000 | |
| **billion** | **9** | **1,000,000,000** | **giga** |
| trillion | 12 | 1,000,000,000,000 | tera |
| quadrillion | 15 | 1,000,000,000,000,000 | peta |
| quintillion | 18 | 1,000,000,000,000,000,000 | exa |
| sextillion | 21 | 1,000,000,000,000,000,000,000 | |

https://en.wikipedia.org/wiki/Power_of_10

# Notes on Sequences, Sums and Averages

- Sum of the digits (commutative – can reduce)
  - Sum(1…n) = [n x (n+1)]/2, but why?
  - We know the average of the sequence
    - Average (Sum(1…n)) = Sum(1…n)/n, by definition
    - Average(1…n) is (n+1)/2, which can be proven by induction
    - E.g., Average(1…2) = [1+2]/2 = 1.5 = (n+1)2 = 3/2 = 1.5
    - E.g., Average(1…3) = [1+**2**+3] /3 = 2 = (n+1)/2 = 4/2 = **2**
    - E.g., Average(1…4) = [1+2+3+4]/4 = 2.5 = (n+1)/2 = 5/2 = 2.5
    - E.g., Average(1…5) = [1+2+**3**+4+5]/5 = 3 = (n+1)/2 = 6/2 = **3**
    - E.g., Average(1…2+**2.5**+3…4) = 12.5/5 = 2.5 = (n+1)/2 = **2.5 = ave**
    - **E.g., Average(1…n) = (1…n/2 + ave + (n/2)+1 … n)/(n+1)**
    - **Average(1…n)(n+1) = (1…n/2 + ave + (n/2)+1 … n)**
    - **ave x n + ave – ave = Sum(1…n), so, ave = Sum(1…n)/n**
  - **Average(Sum(1…n)) = Sum(1…n)/n**
  - Ave=(n+1)/2 = Sum(1…n)/n
  - [n x (n+1)]/2 = Sum(1…n), or Sum(1…n)=[n x (n+1)]/2

# What about Sum(m…n)?

- We know Sum(1…n) contains Sum(m…n)
  - Sum(1…n) = Sum(1…m-1) + Sum(m…n)
  - Sum(m…n) = Sum(1…n) – Sum(1…m-1)
  - Sum(m…n) = [n x (n+1)]/2 – [(m-1) x m]/2
  - E.g., Sum(1…20) = Sum(1…10) + Sum(11…20)
  - E.g., Sum(11…20) = [20 x (20+1)]/2 – [(11-1)x11]/2 = 210-55=155
- This observation is useful for verifying divide and conquer for Average(Sum(sequence)) and related computations
- Inductive math allows us to know the answer, but parallel computation allows us to enumerate it (trace)
- Addition is associative, so order does not matter to final results, but traces may be different

# What about other Sequences

- Fibonacci (See notes here) is fib(n) = fib(n-2) + fib(n-1)
  - fib(0)=0, fib(1)=1
  - E.g., fib(10) = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
  - Recall that Ave(sequence) = Sum(fib(0)…fib(n)) / n
  - Sum(fib(n-2)) = fib(n)-1 (see inductive proof for this)
  - E.g., [0+1+1+2+3+5+8+13+21+34+55]/10 = 14.3
  - 14.3 x 10 = 143
  - Sum(fib(0)…fib(8)) = fib(10) – 1 = 54
  - Average(fib(0)…fib(n)) = Sum(fib(0)…fib(n))/n = (fib(n+2) -1)/n

- So, for a Fibonacci sum or average, we can compute all fib(0)…fib(n) and save in an array, then add them all up
  - Or we could just compute (fib(n+2) -1)
  - Sum(fib(0)…fib(n) = fib(n+2)-1, and Average(fib(0)…fib(n) = [fib(n+2) – 1]/n

# Prime number theorem - Encryption

```
sbsiewert@ecc-linux2:~/public_html/csci551/code/eratos_compare$ ./pinum
Starting pi test @ 2528307.143131

Primes[0..         10]=         4, density=40.000000000% in 0.000215 secs
Primes[0..        100]=        25, density=25.000000000% in 0.000262 secs
Primes[0..       1000]=       168, density=16.800000000% in 0.000480 secs
Primes[0..      10000]=      1229, density=12.290000000% in 0.002707 secs
Primes[0..     100000]=      9592, density=09.592000000% in 0.017275 secs
Primes[0..    1000000]=     78498, density=07.849800000% in 0.148093 secs
Primes[0..   10000000]=    664579, density=06.645790000% in 1.606489 secs
Primes[0.. 100000000]=   5761455, density=05.761455000% in 17.919246 secs
Primes[0..1000000000]=  50847534, density=05.084753400% in 189.874474 secs
sbsiewert@ecc-linux2:~/public_html/csci551/code/eratos_compare$
```

| $x$ | $\pi(x)$ |
|---|---|
| 10 | 4 |
| $10^2$ | 25 |
| $10^3$ | 168 |
| $10^4$ | 1 229 |
| $10^5$ | 9 592 |
| $10^6$ | 78 498 |
| $10^7$ | 664 579 |
| $10^8$ | 5 761 455 |
| $10^9$ | 50 847 534 |
| $10^{10}$ | 455 052 511 |
| $10^{11}$ | 4 118 054 813 |
| $10^{12}$ | 37 607 912 018 |
| $10^{13}$ | 346 065 536 839 |
| $10^{14}$ | 3 204 941 750 802 |
| $10^{15}$ | 29 844 570 422 669 |
| $10^{16}$ | 279 238 341 033 925 |
| $10^{17}$ | 2 623 557 157 654 233 |
| $10^{18}$ | 24 739 954 287 740 860 |
| $10^{19}$ | 234 057 667 276 344 607 |
| $10^{20}$ | 2 220 819 602 560 918 840 |
| $10^{21}$ | 21 127 269 486 018 731 928 |
| $10^{22}$ | 201 467 286 689 315 906 290 |
| $10^{23}$ | 1 925 320 391 606 803 968 923 |
| $10^{24}$ | 18 435 599 767 349 200 867 866 |
| $10^{25}$ | 176 846 309 399 143 769 411 680 |

- Primes are more prevalent in lower number ranges compared to higher
- Let $\pi(x)$ be the prime-counting function that gives the number of primes less than or equal to $x$, for any real number $x$ (*Prime_number_theorem*).
- Trap-door: SP = $Prime_1$ x $Prime_2$, which is easy to compute, but hard to factor
- Sequential code is slow and uses lots of memory

# Parallel Processing and HPC

- Parallel Processing – Using Concurrency to Speed-Up an algorithm (optimal or not)
- Algorithmic Complexity – inherent order of the number of operations required to compute a result
  - $O(\log_2(n))$
  - Sieve simple $O(n \times \log_2(\log_2(n)))$, but n memory locations!
  - $O(n)$ is linear, $O(\log_2(n)) < O(n)$
  - $O(n \times \log_2(n)) < O(n^2)$
  - $O(n) < O(n^2)$, $O(n^3)$, $O(n^m)$ is polynomial
  - Exponential time ($O(n^n)$) complexity cannot be solved for large n
  - E.g., O(n!) is NP at best and most often not computable in human time (perhaps with quantum computer)
  - Simpler parallel programs are polynomial bounded, but high degree
  - Recall P, NP, NP-hard (Non-polynomial, but we can test candidate answers in polynomial time)
  - Primality test complexity – NP in general (provides strength to encryption schemes)
  - Encryption often uses a very large semi-prime that must be factored into 2 prime numbers that compose it (key-based cryptography)

GeeksforGeeks on Sieve Complexity

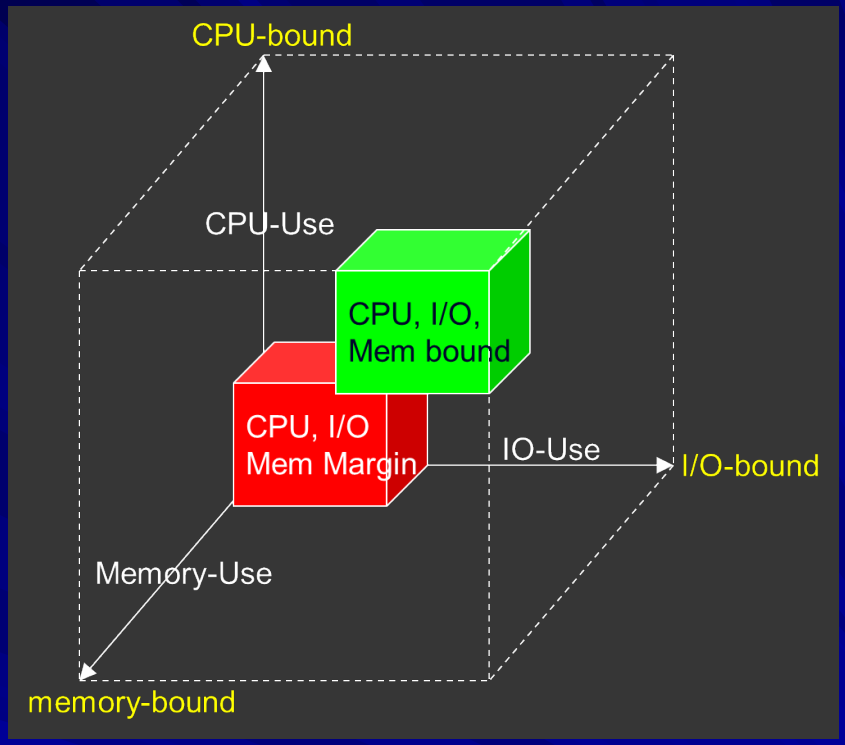
$$n * (\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + .....p)$$

- High Performance Computing – Using scaled up and scaled out resources to solve hard problems
  - CPU scaling (number or cores and number of nodes)
  - Memory scaling (shared and distributed memory)
  - I/O scaling – bandwidth and number o IOPs (I/Os per second)
  - Storage scaling – size of a single file and # of files
- Challenges for primality testing – memory use, complexity, and size of numbers

# Exercise #2 – Eratosthenes Sieve

- Very simple algorithm to cross out all numbers that are multiples of basic prime (e.g., all multiples of 2 and so on…)

- If we want to find a large prime or the # of primes between 0 and a large number, we need
  1. Working algorithm – good?
  2. Efficient implementation
  3. Scaling – CPU & memory
  4. Correct implementation




On ECC I have multiple GB available

```
sbsiewert@ecc-linux:~/code/erast_compare$ free -h
              total        used        free      shared  buff/cache   available
Mem:           3.8Gi       605Mi       2.9Gi       0.0Ki       345Mi       3.0Gi
Swap:          5.9Gi       836Mi       5.0Gi
sbsiewert@ecc-linux:~/code/erast_compare$
```

On my R-Pi 3b+ I have only 670MB at most available

```
pi@raspberrypi:~/code/erast_compare-fast $ free -h
              total        used        free      shared  buff/cache   available
Mem:           926M        141M        535M         63M        249M        670M
Swap:           99M         23M         76M
pi@raspberrypi:~/code/erast_compare-fast $
```

Estimating prime density - https://primes.utm.edu/howmany.html

Better Methods - http://codinghelmet.com/exercises/finding-all-primes

# Sieve Principle – Simple, Slow …

- Crossing out non-primes (multiples of smallest prime, starting with 2), leaving only primes – not optimal, but works

```c
// experiment to see how big of a data segment array you can load.
//
// Hint, use "free - h" to determine how much memory is available and adjust the array
// size to a reasonable limit for your machine.
//
// Then, see whether the algorithm converges (finishes in a reasonable amount of time).
//
//#define MAX (1000000ULL)                      // primes between 0 and 1 million
//#define MAX (10000000ULL)                     // primes between 0 and 10 million
#define MAX (100000000ULL)                    // primes between 0 and 100 million
//#define MAX (200000000ULL)                    // primes between 0 and 200 million
//#define MAX (1000000000ULL)                   // primes between 0 and 1 billion
//#define MAX (0xFFFFFFFFULL)                   // primes between 0 and 2^32-1
//#define MAX (0xFFFFFFFFFFFFFFFFULL)           // primes between 0 and 2^64-1

#define FLOAT double

unsigned char isprime[MAX+1];
unsigned int value[MAX+1];

int main(void)
{
    int i, j;
    unsigned int p=2, cnt=0;
    struct timespec start, now;
    FLOAT fstart=0.0, fnow=0.0;

    clock_gettime(CLOCK_MONOTONIC, &start);
    fstart = (FLOAT)start.tv_sec  + (FLOAT)start.tv_nsec / 1000000000.0;
    clock_gettime(CLOCK_MONOTONIC, &now);
    fnow = (FLOAT)now.tv_sec  + (FLOAT)now.tv_nsec / 1000000000.0;
    printf("\nstart test at %lf\n", fnow-fstart);


    // not prime by definition
    isprime[0]=0; value[0]=0;
    isprime[1]=0; value[1]=1;

    for(i=2; i<MAX+1; i++) { isprime[i]=1; value[i]=i; }

    while( (p*p) <=  MAX)
    {
        // invalidate all multiples of lowest prime so far
        for(j=2*p; j<MAX+1; j+=p) isprime[j]=0;

        // find next lowest prime
        for(j=p+1; j<MAX+1; j++) { if(isprime[j]) { p=j; break; } }
    }


    clock_gettime(CLOCK_MONOTONIC, &now);
    fnow = (FLOAT)now.tv_sec  + (FLOAT)now.tv_nsec / 1000000000.0;
    printf("\nstop test at %lf\n", fnow-fstart);

    for(i=0; i<MAX+1; i++) { if(isprime[i]) { cnt++; } }
    printf("\nTested %u numbers per second up to [0..%llu]=%u\n\n", (unsigned int) (MAX / (fnow-fstart)), MAX, cnt);
}
```

Wikipedia Animation

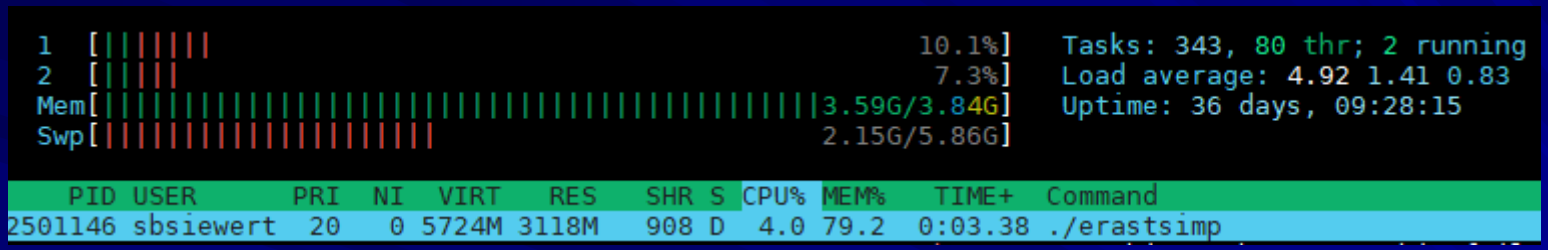
https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes


Sieve of Eratosthenes: algorithm steps for primes below 121 (including optimization of starting from prime's square).

Which do I run out of first?
Memory or Time?
Use a bitmap to compress

# Some ideas to figure this out …

- Stick with simple sieve vs. better algorithm option
- Malloc memory based on command line argument




- Run for limit (300 seconds)
  - What do I run out of? Memory? Time?
  - What is the largest prime I can find in 300 seconds?

1.2 billion is 80% memory quota

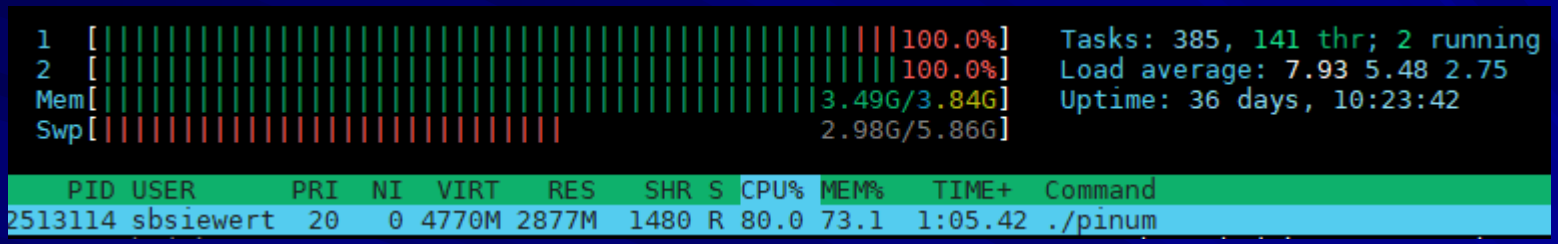
- I could write a script to figure out which I run out of first
- Beyond that, I must have a better algorithm or more resources (memory, clock scaling, parallel?)
- Does parallel really help? Do I need locks? – test and give me your best analysis and judgement answer

# Why do we care about large primes?

- Cryptography trap-door (semi-prime = prime1 x prime2)
- Multiplying 2 integers is simple, even if large
- Finding factors of large integers is hard (especially semi-primes), unless we have one of the 2 primes (a key)
- If n = prime1 x prime2, where 1 < prime1, prime2 < n then finding prime1 and prime2 is non-trivial factor of the large number n (huge search problem)
- https://en.wikipedia.org/wiki/Trapdoor_function
  - csci551/documents/Tutorials/Final-Parallel-Program-Cyber-Ideas.pdf
  - Product and factoring is simple function, can complicate with more complex function
- This principle is key to key-based cryptography

# Find largest < 300 seconds

- Limit on ECC is 1 billion (about)
- Limit on  R-Pi 3b+ less, R-Pi 4 depends on RAM




```
Number of primes [0..1000000]=78498
start test for at 0.000000

stopped test for at 1.097006

Number of primes [0..10000000]=664579
start test for at 0.000000

stopped test for at 10.681235

Number of primes [0..100000000]=5761455
start test for at 0.000000

stopped test for at 242.956881

Number of primes [0..1000000000]=50847534
sbsiewert@ecc-linux:~/code/erast_compare$
```

1,000,000 takes about 1 sec

1,000,000,000 takes 243 sec

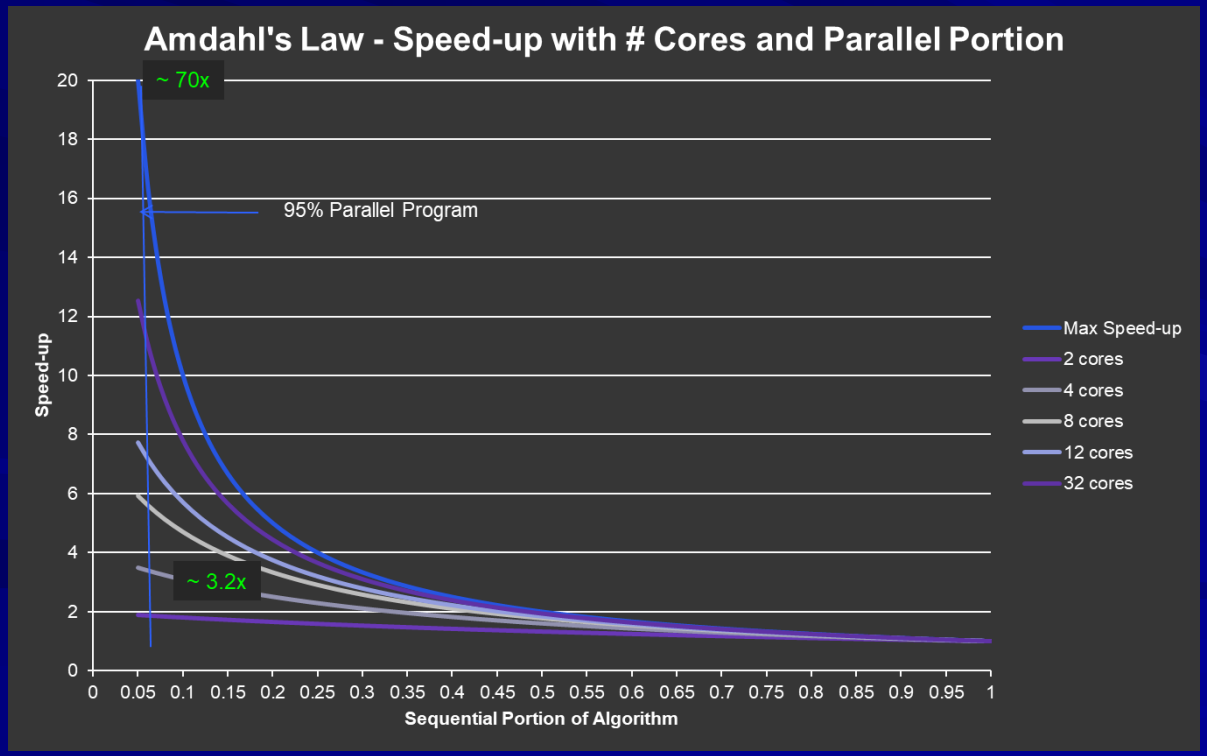
1,000,000,000,000 takes ? sec

Pi(1 billion) = 50,847,534
Pi(1 trillion) = ?

# Embarrassingly Parallel

- Recall that Speed-up is a function of the parallel section and sequential
  - 1-P = sequential %
  - P = parallel %
  - S = scaling factor (most often, number of processors)

$$Multicore\_Speed\_Up = \frac{1}{(1-P)+P/S}$$




For PSF, 70x speed-up with 128 cores

If linear scaling, or close to it is easy to implement, this is embarrassingly parallel problem

Can we speed up the prime hunter?

# Harder to Make Parallel

- Data dependencies are OK – Compute A, use A to Compute B, use B to get C
- Loop carried data dependencies – data dependency where previous loop iteration determines data for next iteration
  - Compute A in parallel and save
  - Use pre-computed A to compute B in parallel
  - Use pre-computed B to get C in parallel
  - Will reduce P, because phases must be sequential (1-P)
  - E.g., numerical integration of equations of motion (acceleration, velocity, and position)
- Shared memory with readers and writers
  - Without synchronization – data corruption!
  - With synchronization, becomes sequential – lowers Amdahl's P (parallel portion)
  - MUTEX locks often used for protection
  - Lock scope determines (1-P) – sequential portion, and therefore P (parallel portion) in Amdahl's law

## Data dependencies

- Within a loop – OK
- Between loops – challenge
- Parallel and sequential sections
- Making whole program parallel may not be possible

## Nick's class - NC

- Complexity P, NC, NP
- Examples
- Complexity P = polynomial with "n"
- Complexity NP = Nondeterministic P
- High degree polynomial, large dimensions scale with parallel
- Factorial, exponential, and anything with complexity greater than P, may benefit, but not solved by parallel
- Useful for NP hard – Heuristic with P bounded test for correctness

Complexity Zoo (simpler) – Collection of algorithms: P, NC, NP and points between

# How do we classify Parallel Challenge?

- P – polynomial bounded algorithms
- NP – non-polynomial – algorithm that is P is not known for this problem (exponential time)
- NP-hard – non-polynomial algorithm, which can however test a hypothesized solution in P time (heuristic)

P vs. NP

Can every problem verifiable in P time actually be solved in P time?

- P-complete – hard to speed-up with parallel programming (probably not possible)
  - Hard to divide up (mapping to parallel processing units in S)
  - Hard to combine results (reduction of results to answer question)
  - Or both
- NC – Nick's Class (Polynomial bounds – Nick Pippenger) – P problems that can be sped up

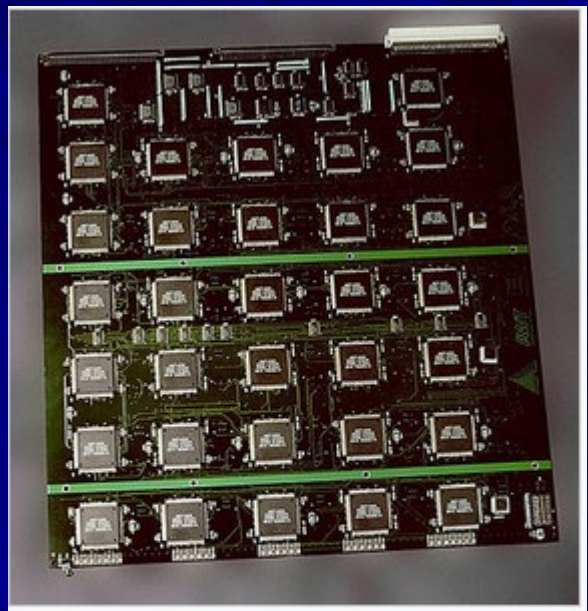
Unknown in CS

$NC =^{?} P$

I.e., can any polynomial time algorithm be sped-up with some map-reduce?

# Our hardest problems (still NC)

- Simulation with equations of motion – NC, but not easy
- Primality testing – NC, but not easy (Locks? Memory?)
- Cryptanalysis – P-Complete (not NC for strong with large keys)
  - Exponential time required based on key size – e.g., $2^{256}$
  - NP-hard attacks do exist (heuristics testable in P time, that are NC) and DES was cracked in human time (small key of $2^{56}$)
    - E.g., dictionary attacks, RockYou hash checking, etc. can work, but not always – (weak passwords)
  - Lost or no-key decryption is not NC for AES or RSA large keys
    - If cracking strong encryption was NC, we'd be in real trouble – it would be weak (encryption like DES is NC, small key size and was broken!)
  - Is Parallel a threat to Encryption – not likely as long as key sizes are large enough, but other forms of computing …?
    - Quantum supremacy – idea that only Quantum computers can solve these NP-complete and P-Complete problems that have defied solution with Turing class computers and parallel
- I/O bound problems (if S = CPU cores, no speed-up)
- Memory bound problems
- MPI node scaling can help with multi-resource scaling
  - E.g., data analytics and machine learning
  - Computer vision and digital media processing (movie magic!)



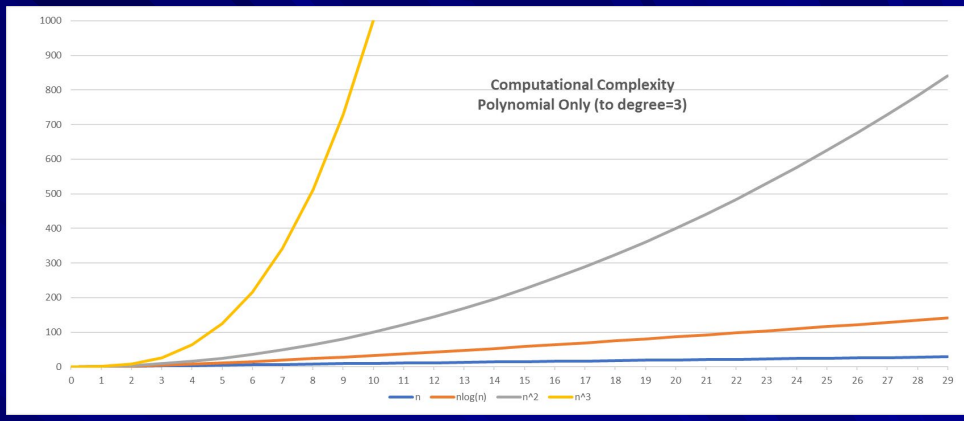
The EFF's US$250,000 DES cracking machine contained 1,856 custom chips and could brute force a DES key in a matter of days — the photo shows a two-sided DES Cracker circuit board fitted with 64 Deep Crack chips

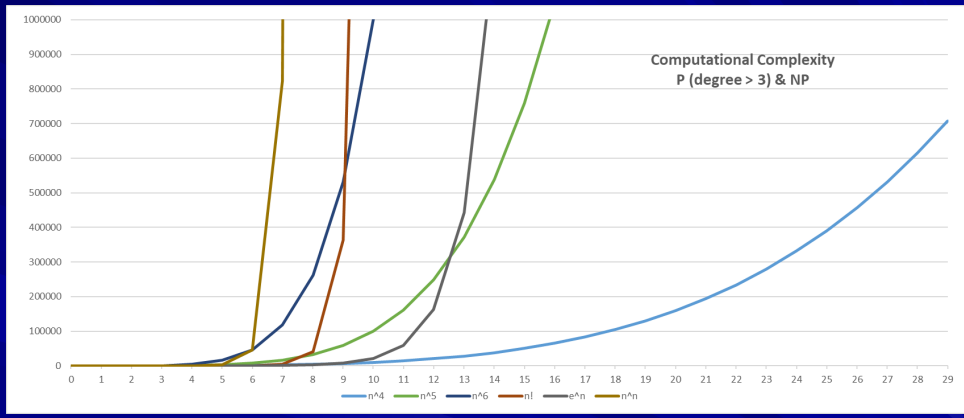
Wikipedia EFF DES Cracker –

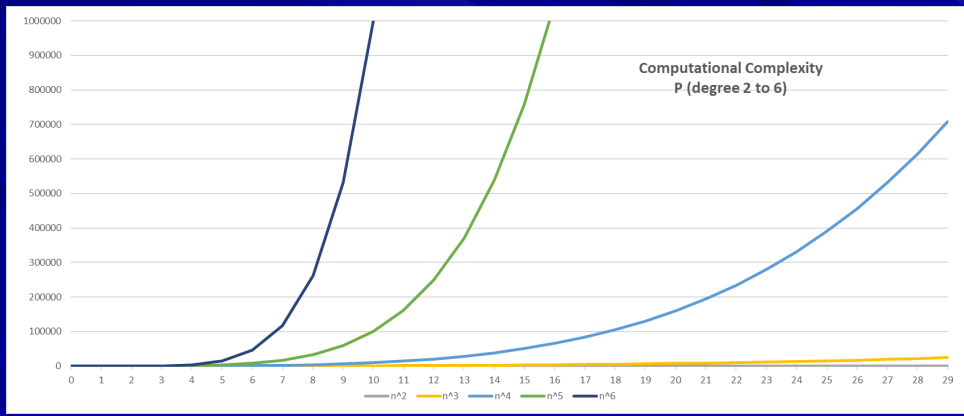
"DES uses a 56-bit key, meaning that there are $2^{56}$ possible keys under which a message can be encrypted. This is exactly 72,057,594,037,927,936, or approximately 72 quadrillion possible keys. One of the major criticisms of DES, when proposed in 1975, was that the key size was too short."

# Problems we can solve with parallel

- Polynomial bounded – P time
  - but inconvenient (days to run …)
  - large n, high degree and dimension (e.g., $n^5$ or $n^4$, but not $n^n$ or n!)
  - Not exponential, high degree polynomial
  - Divide and Conquer – log(N)
  - Large N complexity becomes $O((\log(N))^k)$ – Polylogarithmic
  - Dimensions are big (lots of data)
- Step sizes, grid size, and/or resolution determines "n" in polynomial bound and smaller is better (larger n) for fidelity, accuracy, or for a given precision
- Scaling has other issues – memory, I/O, network and size of numbers (integers) and precision (digits)
- Iterations must be significant
- Degree > 3 - parallel high value!



# General Suggestion for Class

- Always scale to run > 3 seconds
- Running longer than 5 minutes, only if required to solve
- Overcome start-up issues
  - E.g., cache warm up
  - Loading files into memory
  - Ignore user interaction time (prompts for input)
- I/O Bound will defeat parallel processing – need parallel I/O instead or in addition!
- Memory bound can result in more I/O
- Network overhead limits MPI scaling (messages / sec)