Parallel Computing

John Urbanic

Parallel Computing Scientist
Pittsburgh Supercomputing Center

Distinguished Service Professor
Carnegie Mellon University

Who am I?

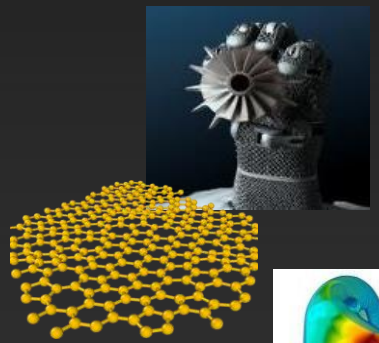
John Urbanic

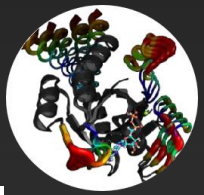
Distinguished Service Professor
Carnegie Mellon University

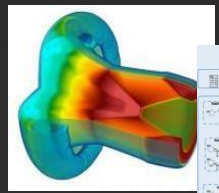
Undergrad Advanced Computational Physics
Graduate Large Scale Computing
Data Science Capstone Projects

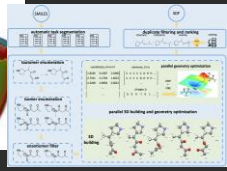
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Code, code, code, on
Parallel platforms: MPI, OpenMP, OpenACC, ...
Big Data platforms: Spark, ...
Machine Learning: Spark, TensorFlow, PyTorch, ...





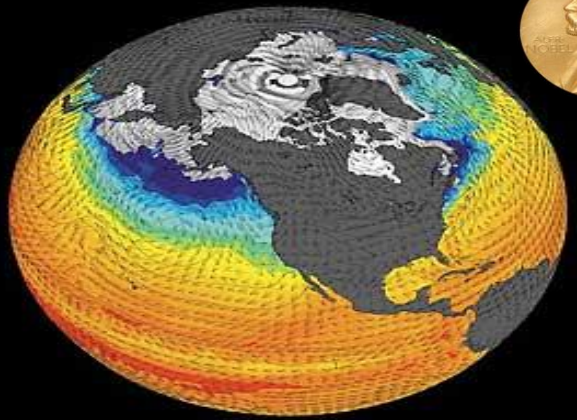


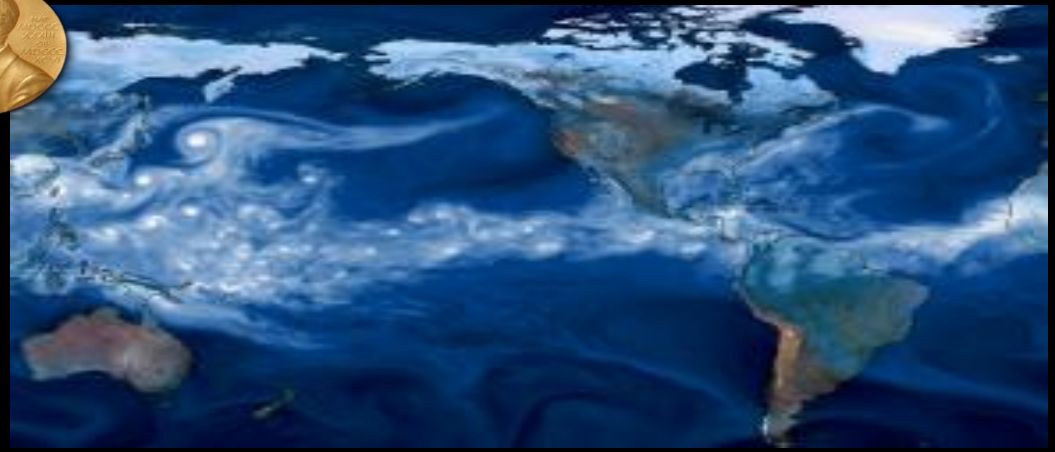


This Module

- Why is parallel computing so important?
- What does it look like today?
- How do we take advantage of it?
- Where is it headed?
- Let's try a few things...
- in preparation for the upcoming programming modules.

Compute bound problems abound: Climate change analysis



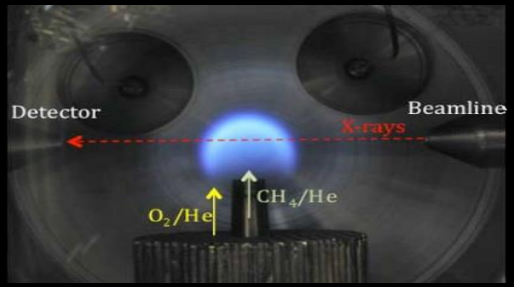


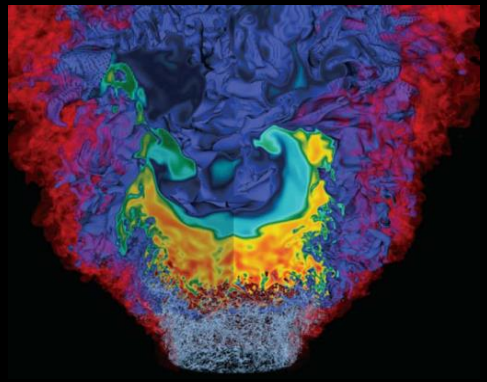
Simulations	Extreme data
• Cloud resolution, quantifying uncertainty, understanding tipping points, etc., will drive climate to exascale platforms • New math, models, and systems support will be needed	• “Reanalysis” projects need 100× more computing to analyze observations • Machine learning and other analytics are needed today for petabyte data sets • Combined simulation/observation will empower policy makers and scientists

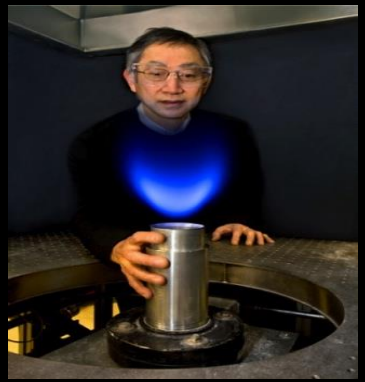
Courtesy Horst Simon, LBNL

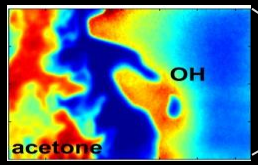
Exascale is needed at all scales: combustion simulations

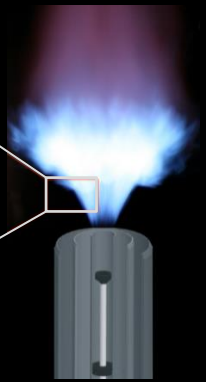
- **Goal: 50% improvement in engine efficiency**
- **Center for Exascale Simulation of Combustion in Turbulence (ExaCT)**
  - **Combines simulation and experimentation**
  - **Uses new algorithms, programming models, and computer science**

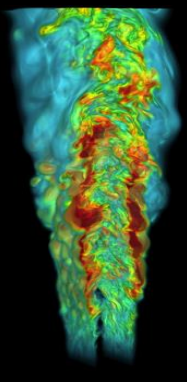








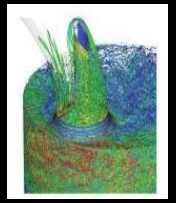
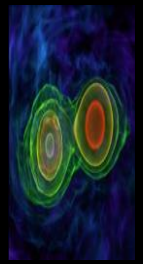
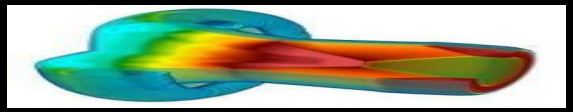
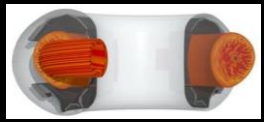
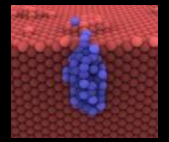
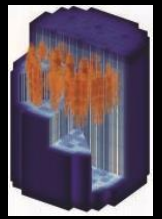
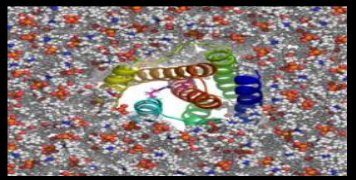
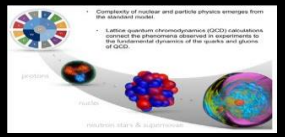




Courtesy Horst Simon, LBNL

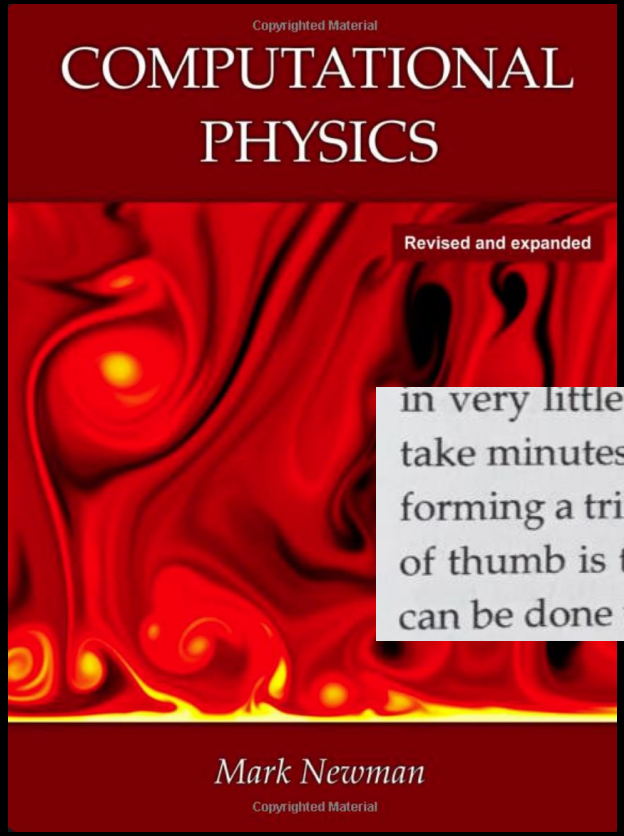
The list is long, and growing.

- Molecular-scale Processes: atmospheric aerosol simulations
- AI-Enhanced Science: predicting disruptions in tokomak fusion reactors
- Hypersonic Flight
- Modeling Thermonuclear X-ray Bursts: 3D simulations of a neutron star surface or supernovae
- Quantum Materials Engineering: electrical conductivity photovoltaic and plasmonic devices
- Physics of Fundamental Particles: mass estimates of the bottom quark
- Digital Cells

These and others are in an appendix at the end of our Outro To Parallel Computing talk.

And many of you doubtless brought your own immediate research concerns. Great!



in very little time. Performing a billion operations, on the other hand, could take minutes or hours, though it's still possible provided you are patient. Performing a trillion operations, however, will basically take forever. So a fair rule of thumb is that the calculations we can perform on a computer are ones that can be done with *about a billion operations or less.*

Where are those 10 or 12 orders of magnitude?

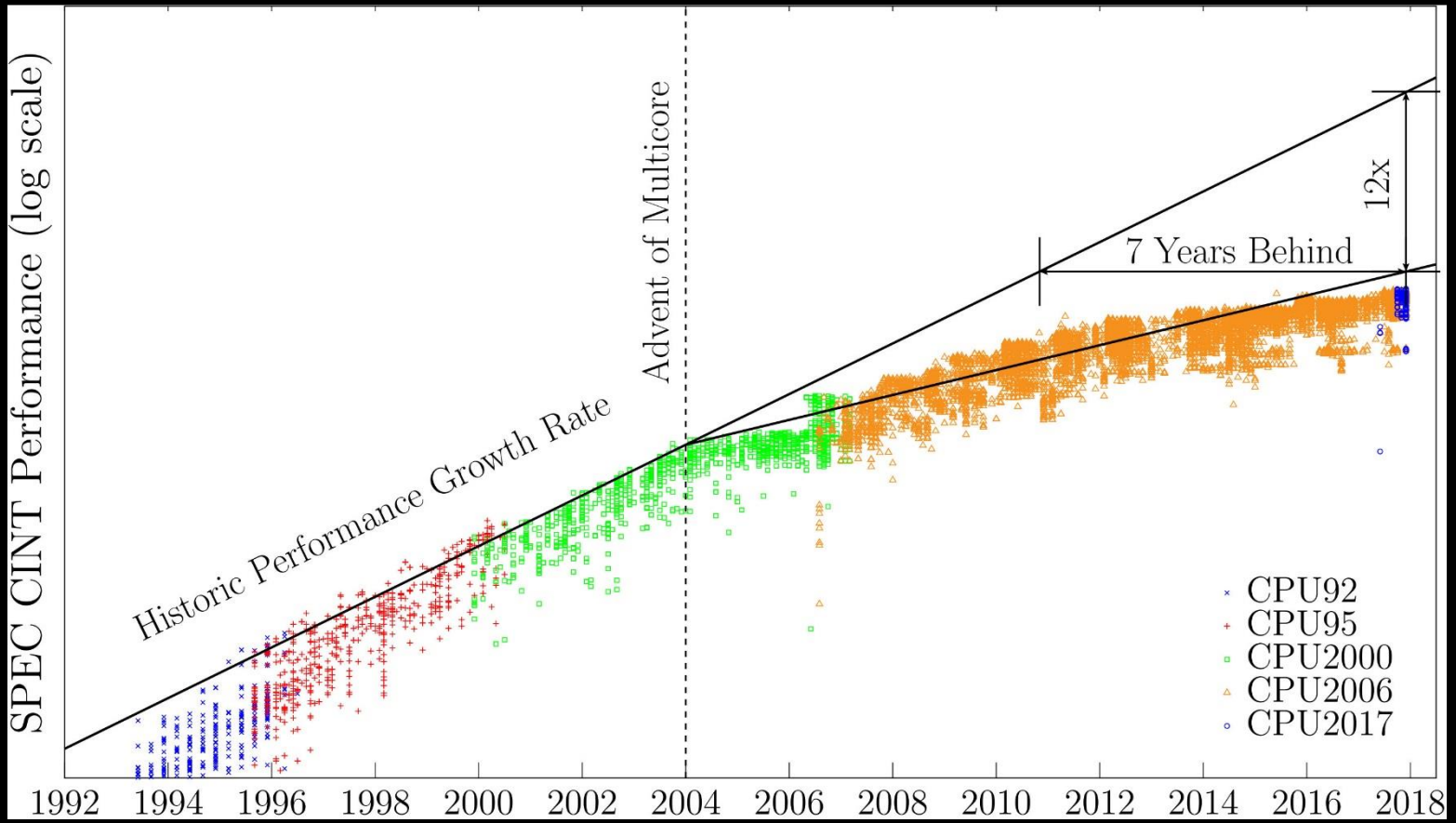
How do we get there from here?

BTW, that's a bigger gap than

VS.

IBM 709
12 kiloflops

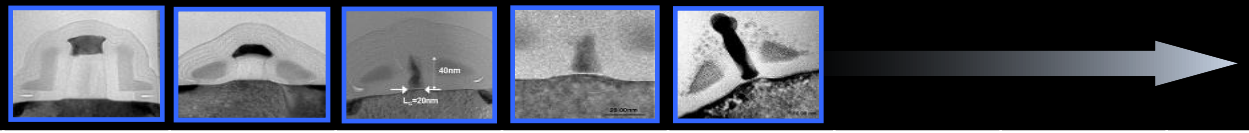
Moore's Law abandoned serial programming around 2004



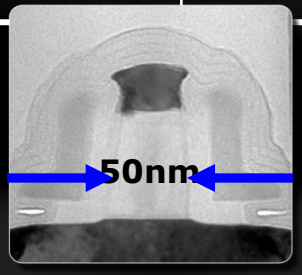
Courtesy Liberty Computer Architecture Research Group

But Moore's Law is only beginning to stumble now.

Intel process technology capabilities

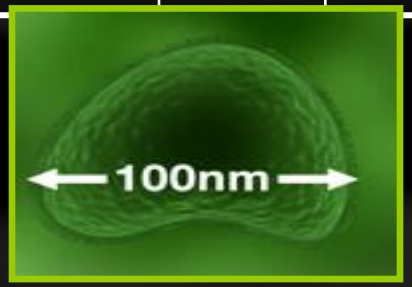


High Volume Manufacturing	2004	2006	2008	2010	2012	2014	2018	2021
Feature Size	90nm	65nm	45nm	32nm	22nm	14nm	10nm	7nm
Integration Capacity (Billions of Transistors)	2	4	8	16	32	64	128	256



Transistor for 90nm Process

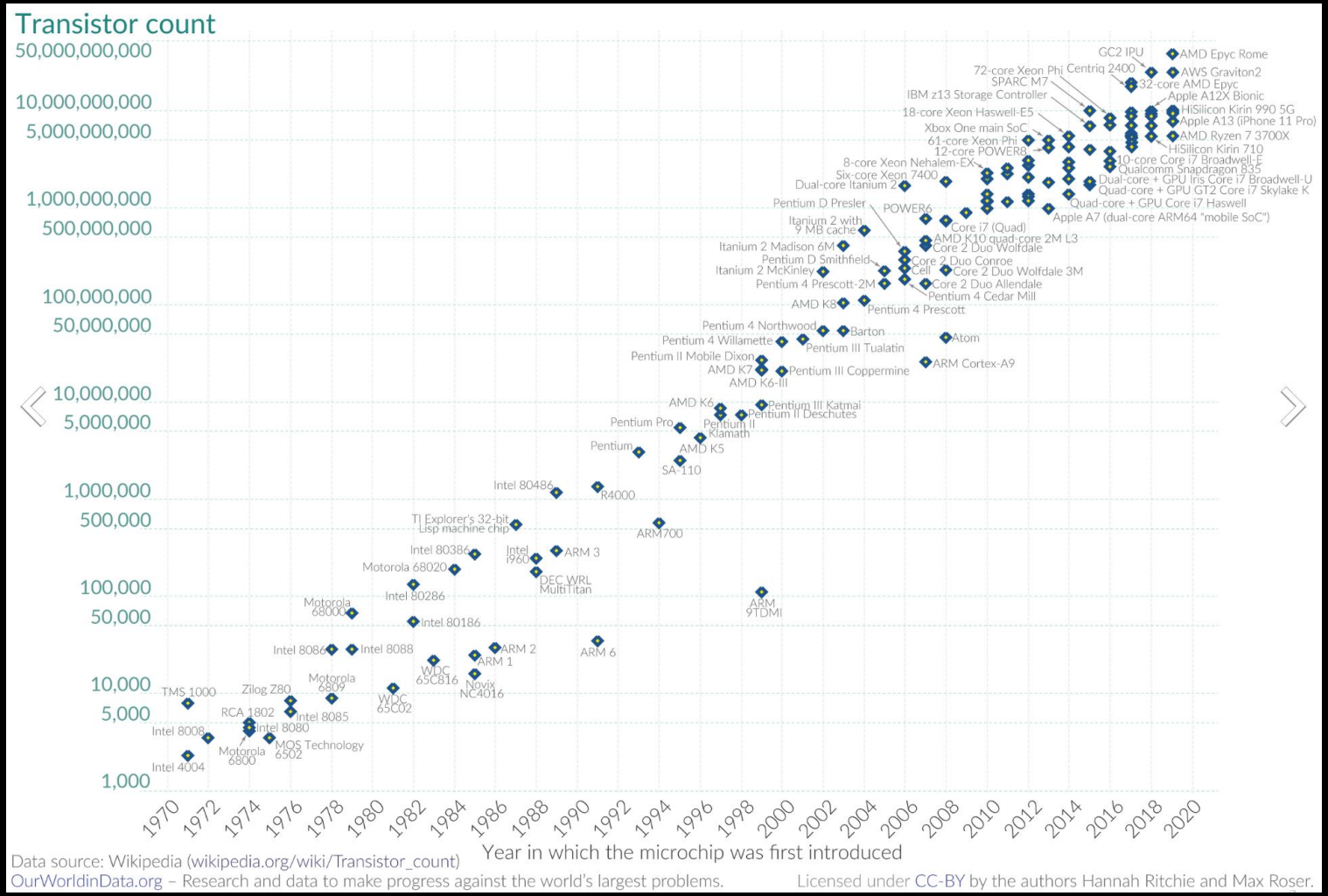
Source: Intel



Influenza Virus

Source: CDC

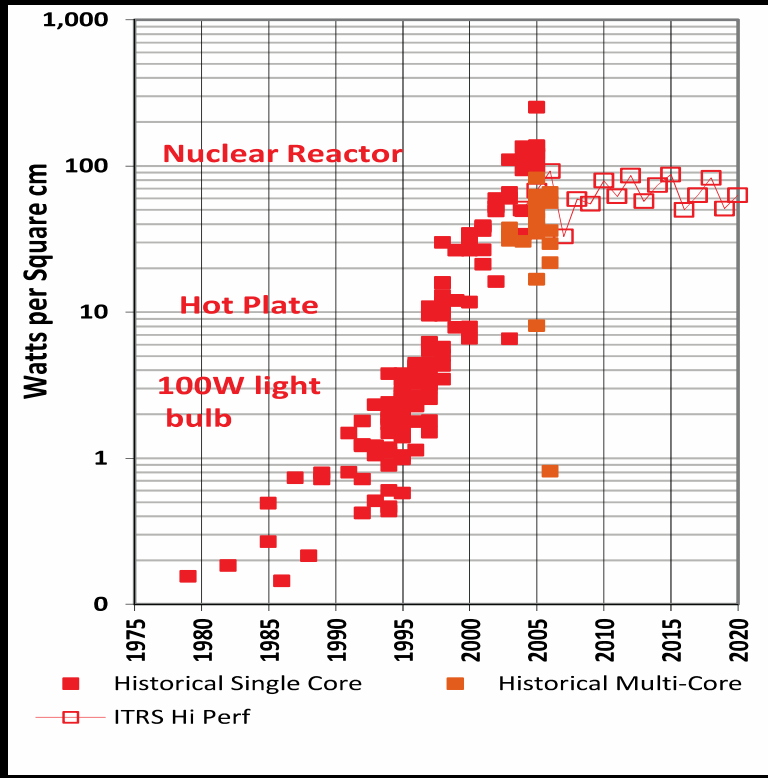
And at end of day we keep using getting more transistors.

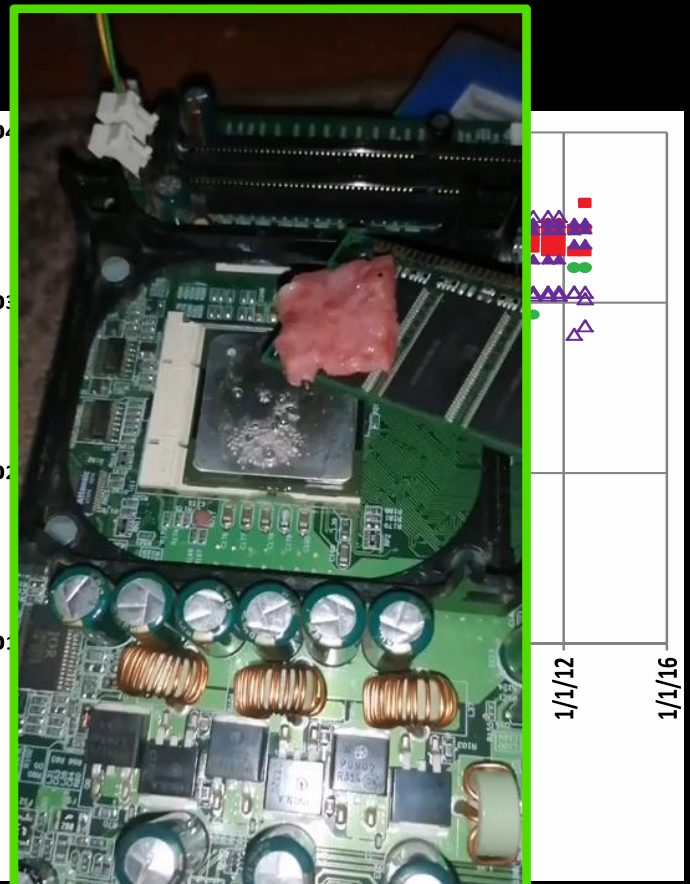


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

And run into the real problem. This is the central driver of 21st century computing!

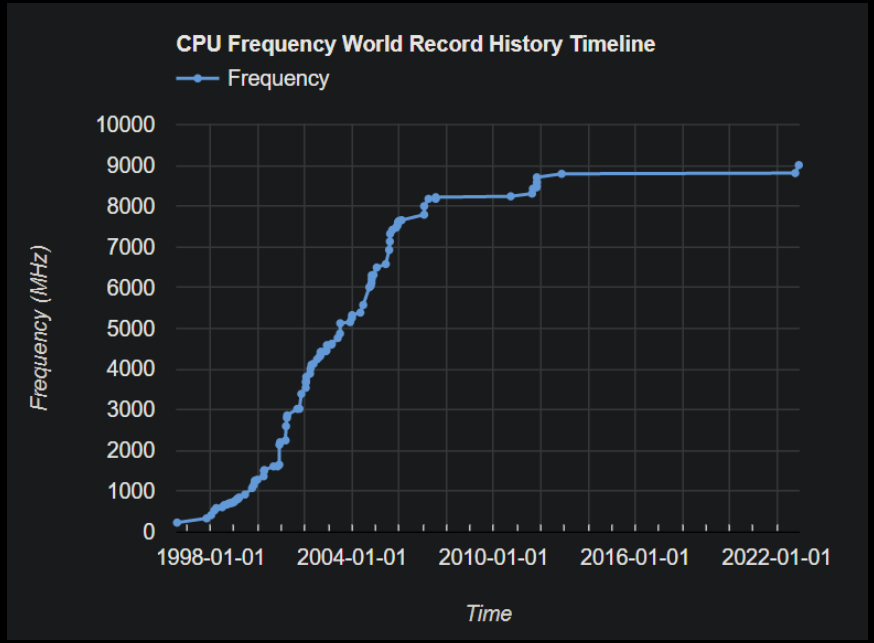




Fun fact: At 100+ Watts and <1V, currents are beginning to exceed 100A at the point of load!

Courtesy Horst Simon, LBNL

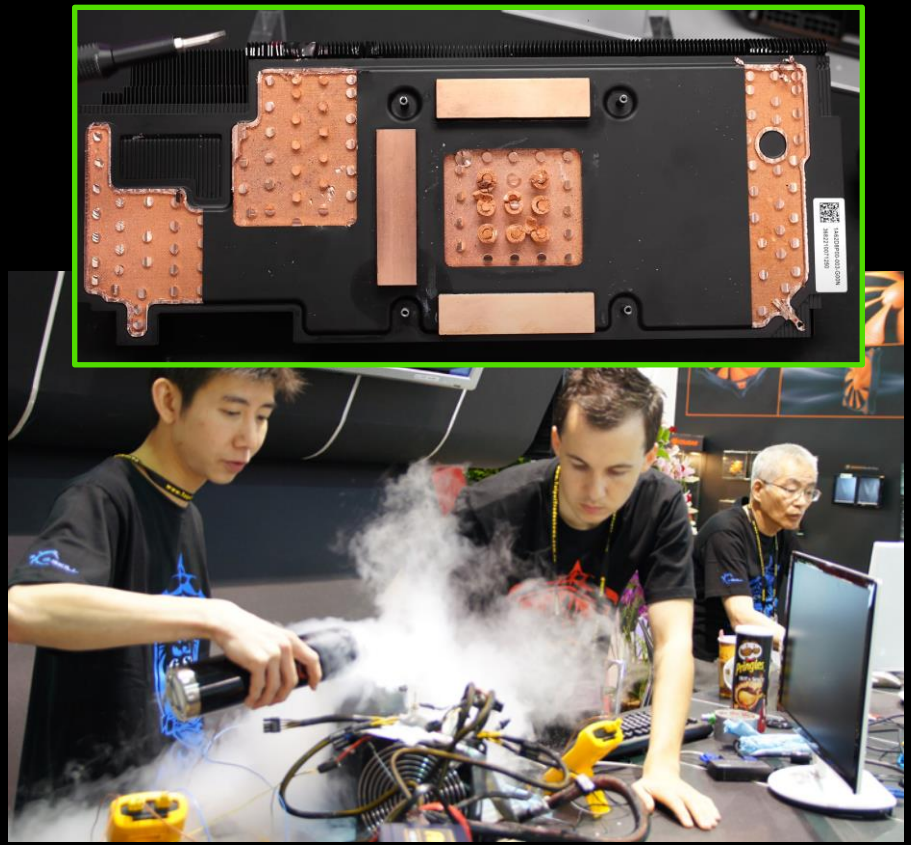
Even when you go extreme...



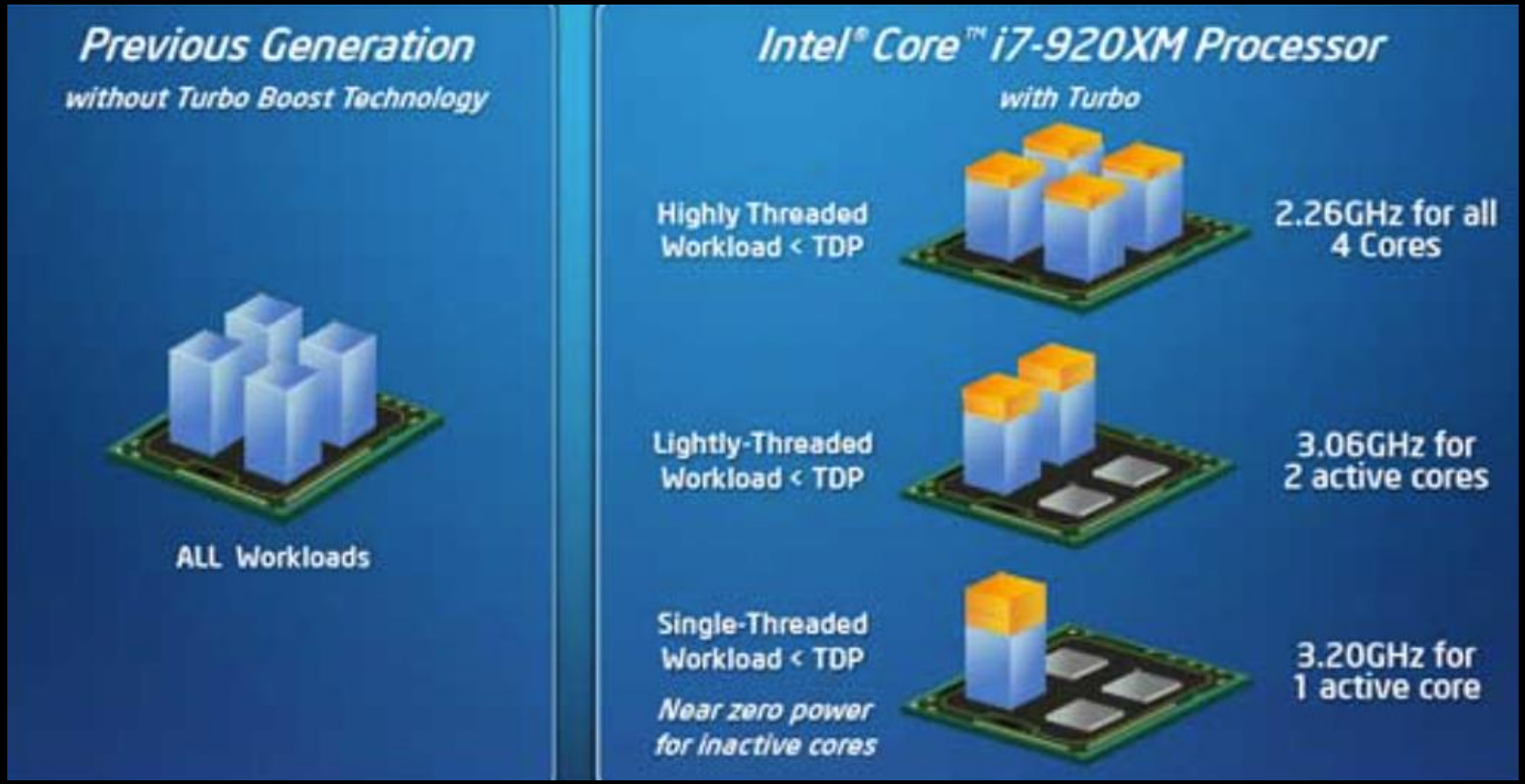
These are CPUs you can buy.

https://hwbot.org/benchmark/cpu_frequency/halloffame

Complex liquid cooling on a consumer GPU.

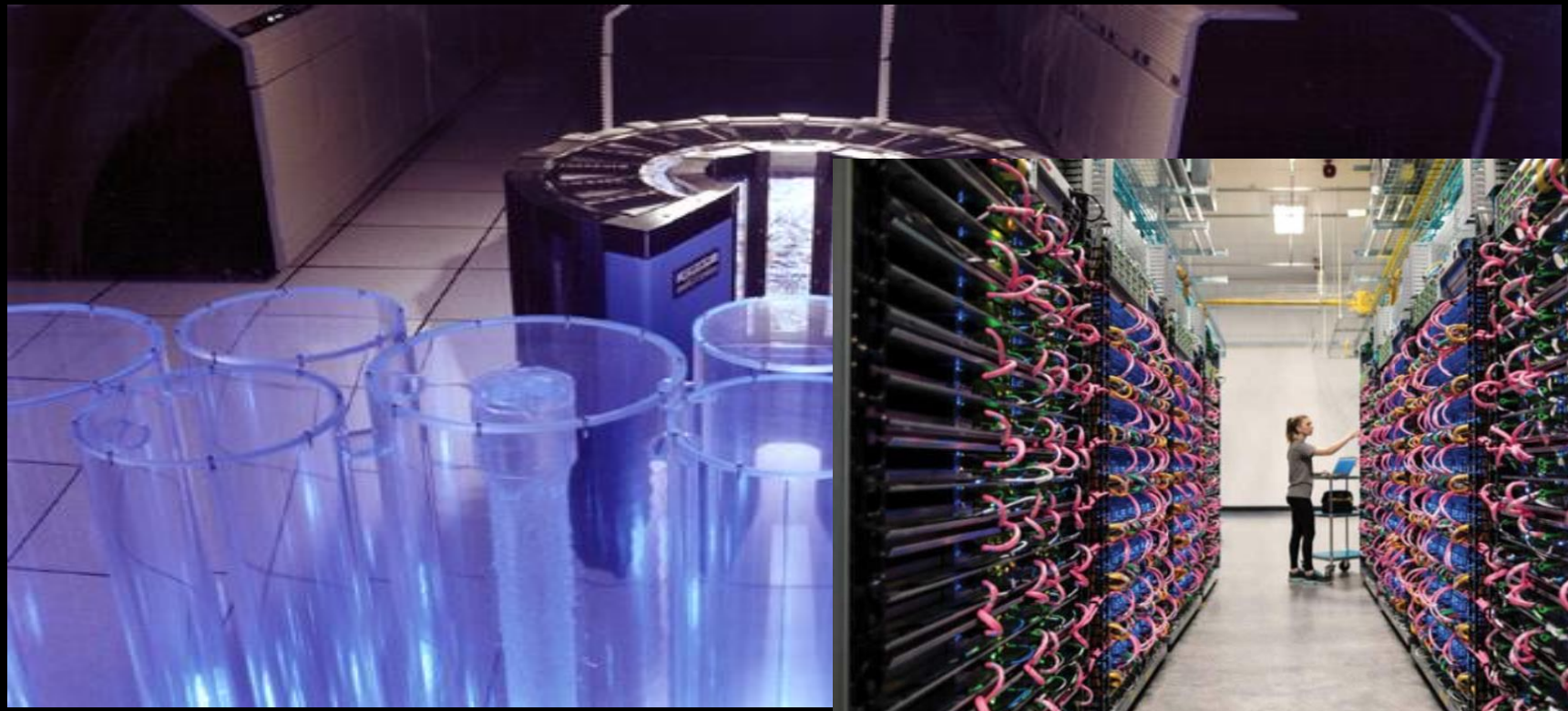


For those of you thinking, "Well, at least *my* CPU runs at 4+ GHz."



Maybe sometimes.

Not a new problem…just ubiquitous.



Cray-2 with cooling tower i

Google's liquid cooled TPU v2 servers deployed in racks. *Source: Google.*

Starting to see 200KW per cabinet in datacenters.

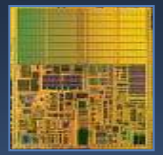
And how to get more performance from more transistors with the same power.

A 15% Reduction In Voltage Yields

RULE OF THUMB

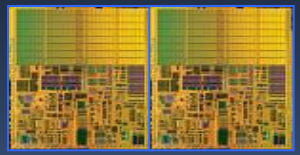
Frequency Reduction	Power Reduction	Performance Reduction
15%	45%	10%

SINGLE CORE



Area = 1
Voltage = 1
Freq = 1
Power = 1
Perf = 1

DUAL CORE

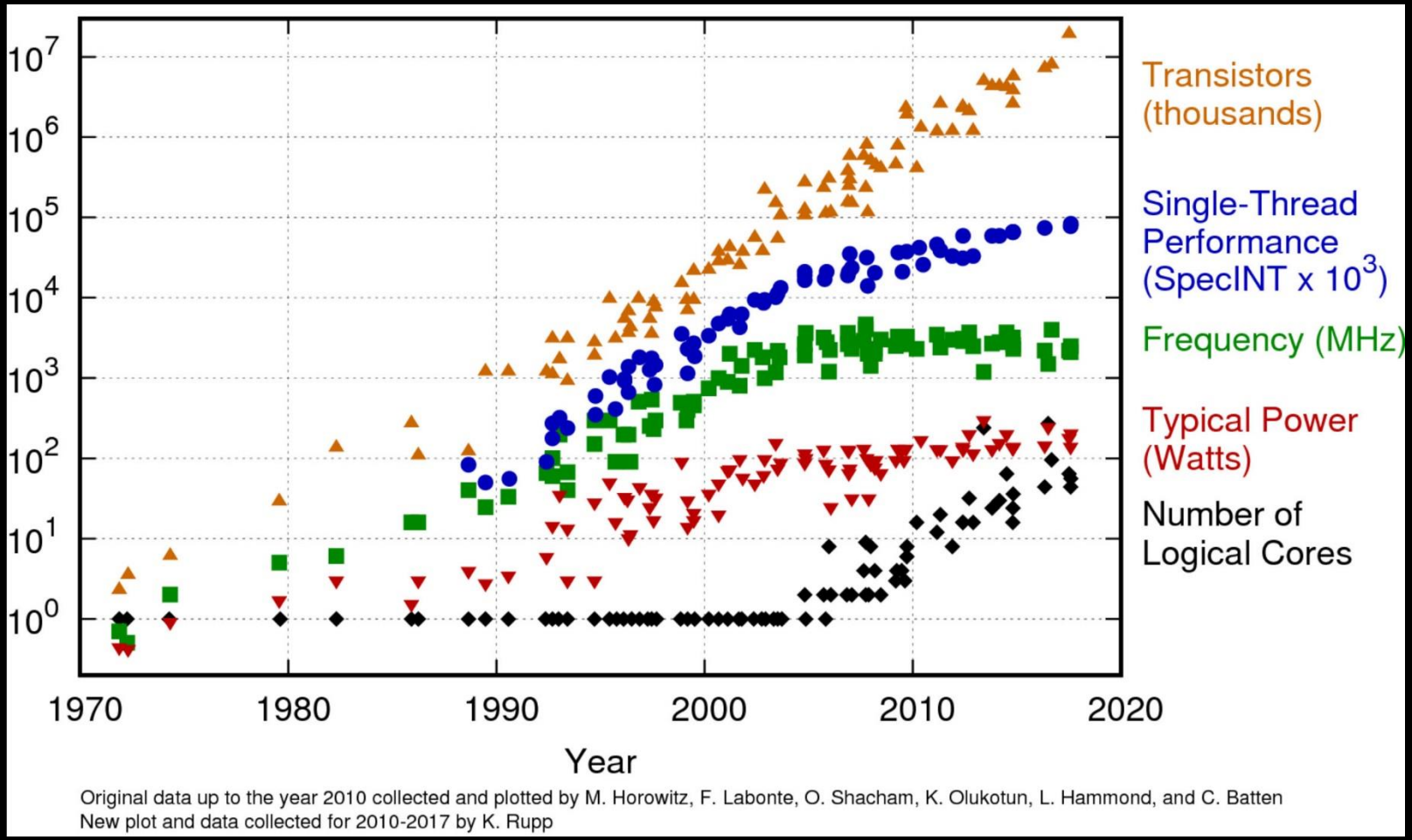


Area = 2
Voltage = 0.85
Freq = 0.85
Power = 1
Perf = ~1.8

Single Socket Parallelism

Processor	Year	Vector	Bits	SP FLOPs / core / cycle	Cores	FLOPs/cycle
Pentium III	1999	SSE	128	3	1	3
Pentium IV	2001	SSE2	128	4	1	4
Core	2006	SSE3	128	8	2	16
Nehalem	2008	SSE4	128	8	10	80
Sandybridge	2011	AVX	256	16	12	192
Haswell	2013	AVX2	256	32	18	576
KNC	2012	AVX512	512	32	64	2048
KNL	2016	AVX512	512	64	72	4608
Skylake	2017	AVX512	512	96	28	2688

Putting It All Together



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Parallel Computing

One woman can make a baby in 9 months.

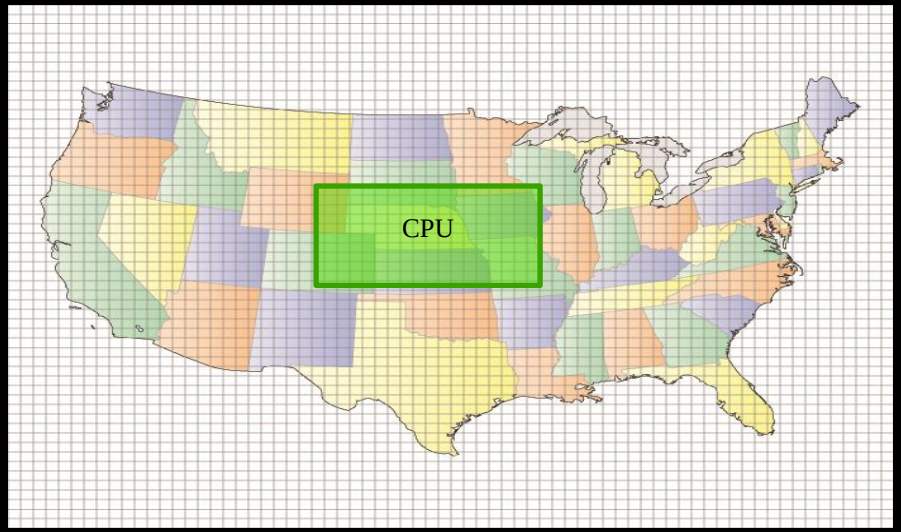
Can 9 women make a baby in 1 month?

But 9 women can make 9 babies in 9 months.

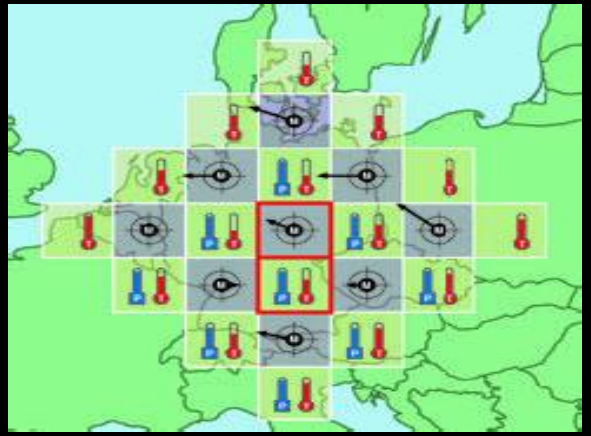
First two bullets are Brook's Law. From *The Mythical Man-Month.*

A must-read for serious project programmers that includes many other classics such as:
"What one programmer can do in one month, two programmers can do in two months."

Prototypical Application: Serial Weather Model

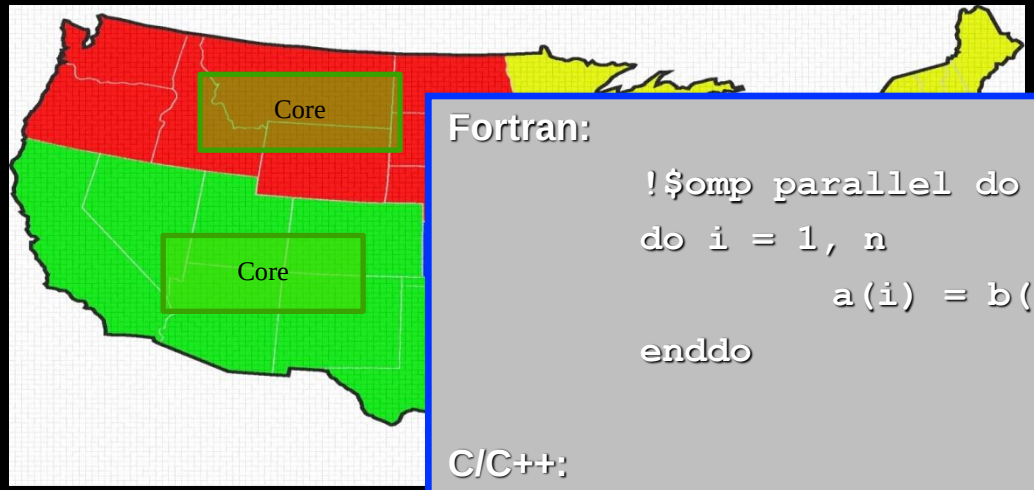


First Parallel Weather Modeling Algorithm: Richardson in 1917



Courtesy John Burkhardt, Virginia Tech

Weather Model: Shared Memory (OpenMP)



Four meteorologists in t

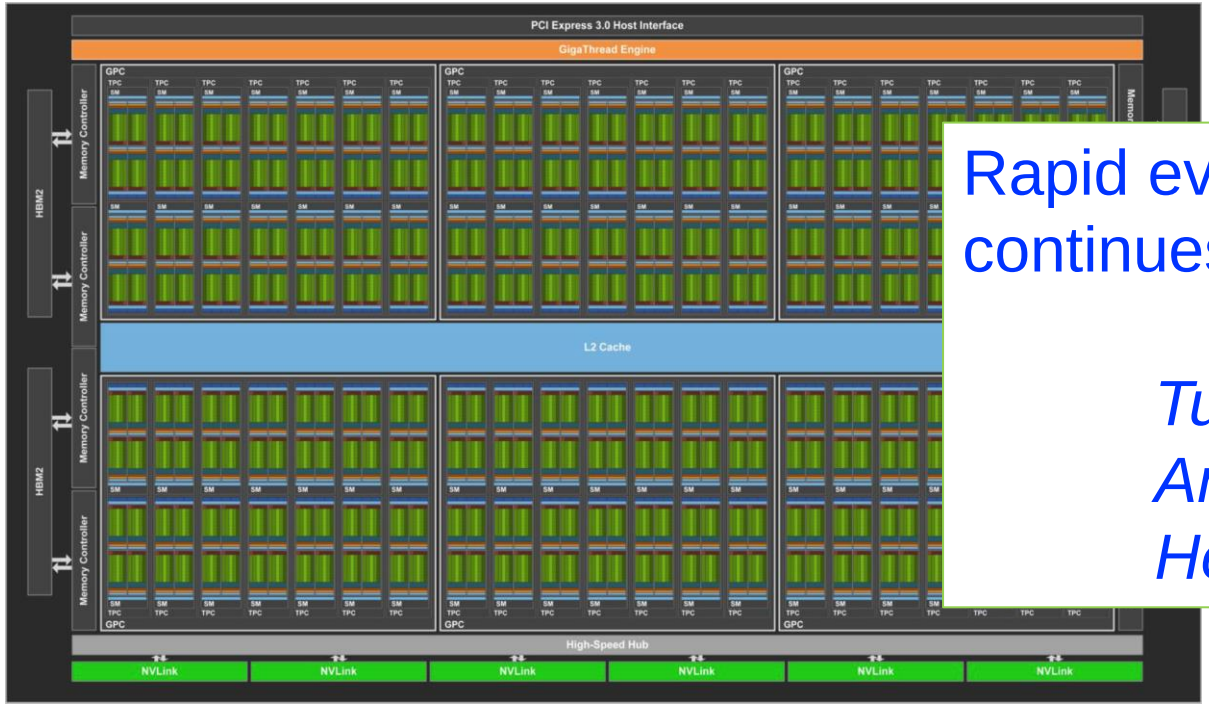
Fortran:

```
!$omp parallel do
do i = 1, n
        a(i) = b(i) + c(i)
enddo
```

C/C++:

```
#pragma omp parallel for
for(i=1; i<=n; i++)
        a[i] = b[i] + c[i];
```

V100 GPU and SM



Volta GV100 GPU with 85 Streaming Multiprocessor (SM) units

Volta GV100 SM

Rapid evolution continues with:

Turing
Ampere
Hopper

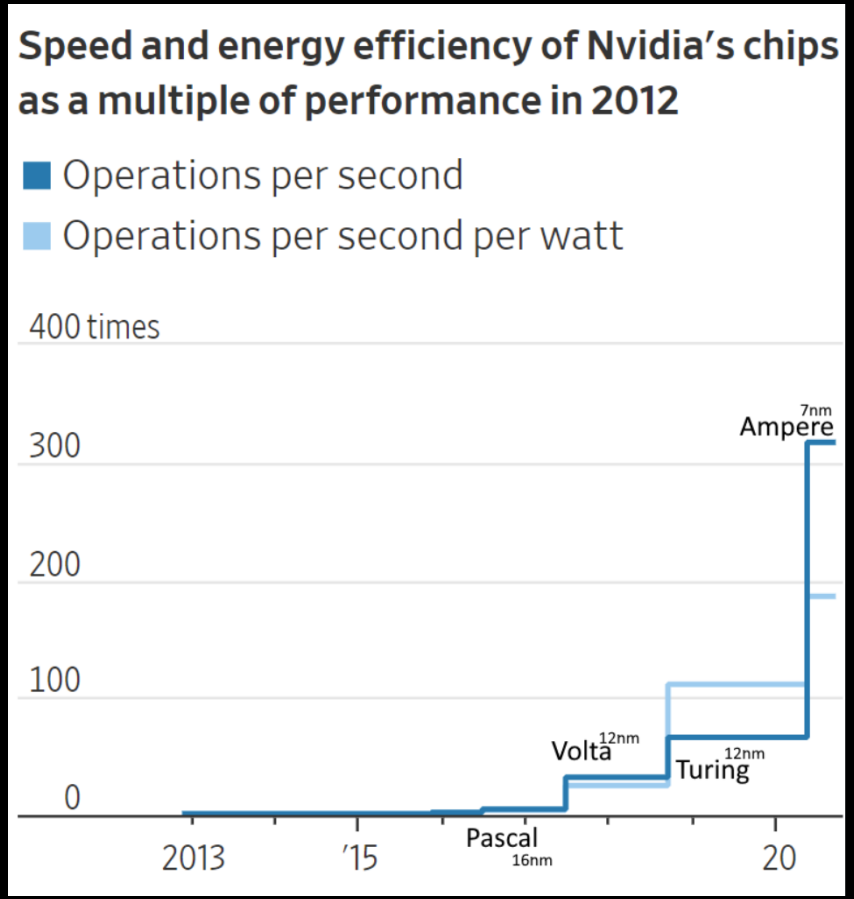
Huang's Law

An observation/claim made by Jensen Huang, CEO of Nvidia, at its 2018 GPU Technology Conference.

He observed that Nvidia's GPUs were "25 times faster than five years ago" whereas Moore's law would have expected only a ten-fold increase.

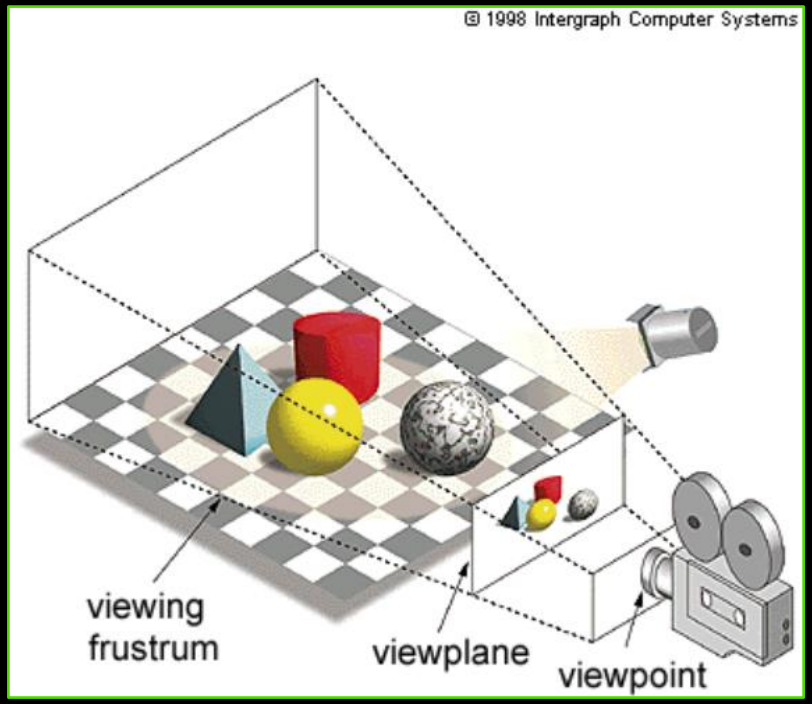
In 2006 Nvidia's GPU had a 4x performance advantage over other CPUs. In 2018 the Nvidia GPU was 20 times faster than a comparable CPU node: the GPUs were 1.7x faster each year. Moore's law would predict a doubling every two years, however Nvidia's GPU performance was more than tripled every two years fulfilling Huang's law.

It is a little premature, and there are confounding factors at play, so most people haven't yet elevated this to the status of Moore's Law.



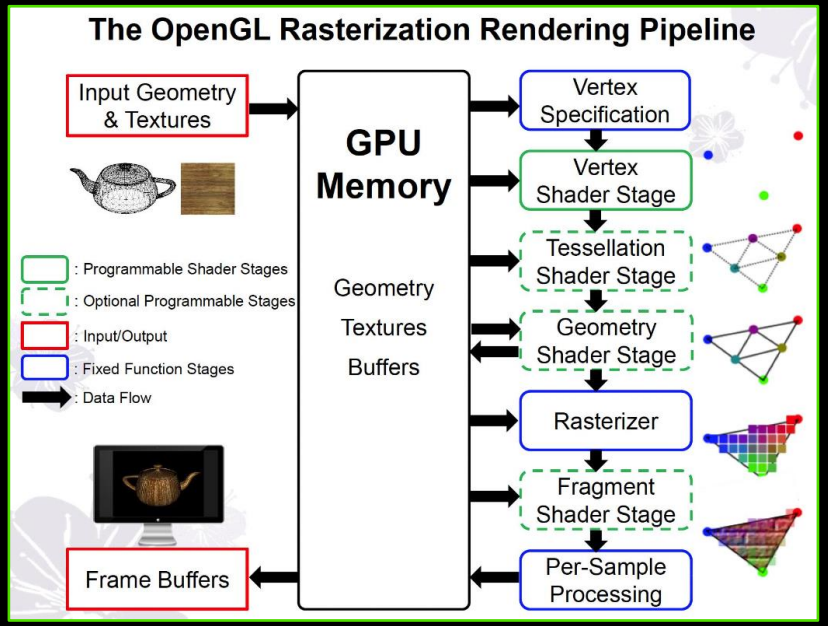
Source: NVIDIA

Why Video Gaming Cards?



By the turn of the century, the video gaming market has already standardized around a few APIs for rendering 3D video games in real-time.

None of these looked anything like scientific computing.



Heroic Efforts

An API in 2004 first demonstrated the potential use of this latent floating point ability.

By 2007 NVIDIA supported a dedicated API for their own hardware.

Note that these early devices were not at all engineered for scientific computing and lacked several very fundamental capabilities. In particular EEC and double precision.

Brook for GPUs: Stream Computing on Graphics Hardware

Ian Buck Tim Foley Daniel Horn Jeremy Sugerman Kayvon Fatahalian Mike Houston Pat Hanrahan
Stanford University

Abstract

In this paper, we present Brook for GPUs, a system for general-purpose computation on programmable graphics hardware. Brook extends C to include simple data-parallel constructs, enabling the use of the GPU as a streaming co-processor. We present a compiler and runtime system that abstracts and virtualizes many aspects of graphics hardware. In addition, we present an analysis of the effectiveness of the GPU as a compute engine compared to the CPU, to determine when the GPU can outperform the CPU for a particular algorithm. We evaluate our system with five applications, the SAXPY and SGEMV BLAS operators, image segmentation, FFT, and ray tracing. For these applications, we demonstrate that our Brook implementations perform comparably to hand-written GPU code and up to seven times faster than their CPU counterparts.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors D.3.2 [Programming Languages]: Language Classifications—Parallel Languages

Keywords: Programmable Graphics Hardware, Data Parallel Computing, Stream Computing, GPU Computing, Brook

1 Introduction

In recent years, commodity graphics hardware has rapidly evolved from being a fixed-function pipeline into having programmable vertex and fragment processors. While this new modern hardware. In addition, the user is forced to express their algorithm in terms of graphics primitives, such as textures and triangles. As a result, general-purpose GPU computing is limited to only the most advanced graphics developers.

This paper presents *Brook*, a programming environment that provides developers with a view of the GPU as a streaming coprocessor. The main contributions of this paper are:

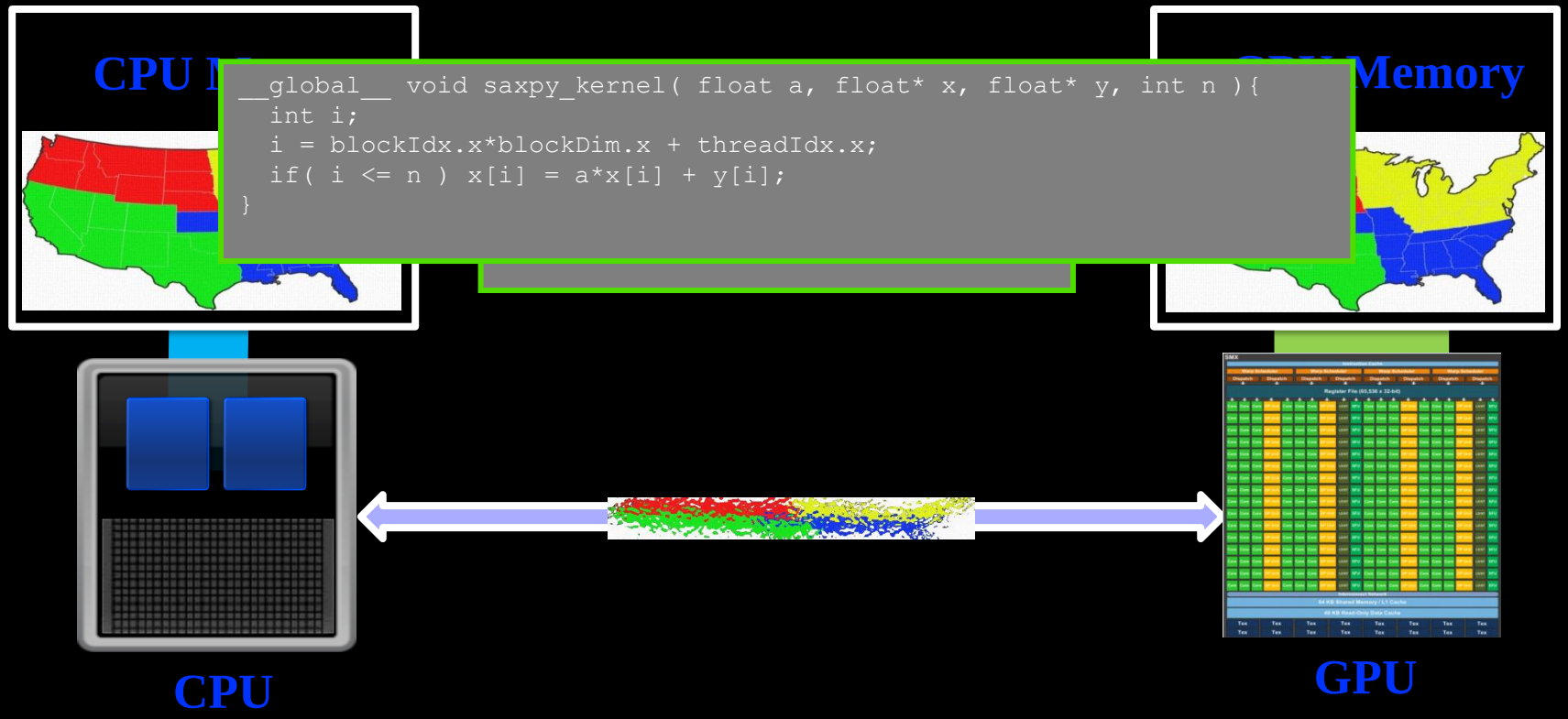
- The presentation of the Brook stream programming model for general-purpose GPU computing. Through the use of streams, kernels and reduction operators, Brook abstracts the GPU as a streaming processor.
- The demonstration of how various GPU hardware limitations can be virtualized or extended using our compiler and runtime system; specifically, the GPU memory system, the number of supported shader outputs, and support for user-defined data structures.
- The presentation of a cost model for comparing GPU vs. CPU performance tradeoffs to better understand under what circumstances the GPU outperforms the CPU.

2 Background

2.1 Evolution of Streaming Hardware

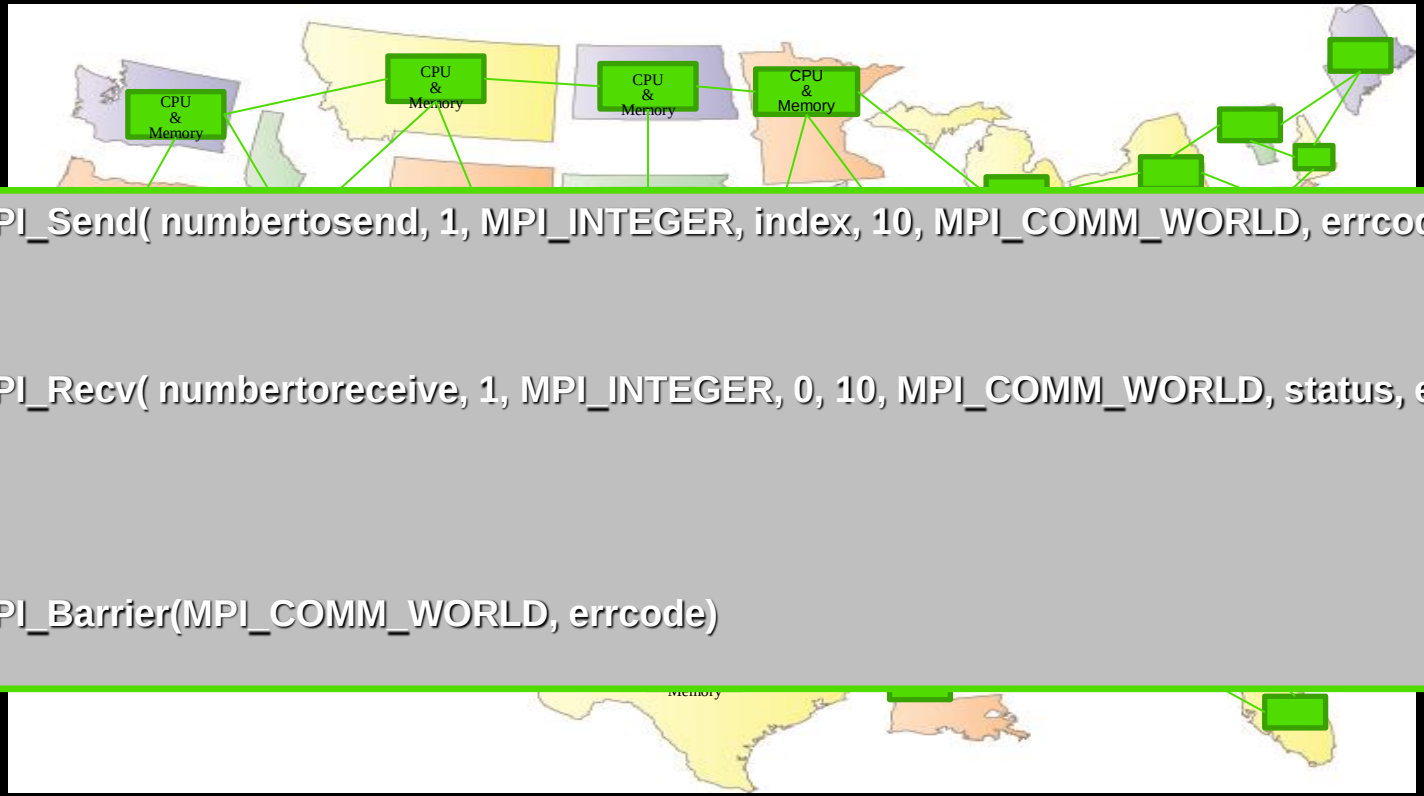
Programmable graphics hardware dates back to the original programmable framebuffer architectures [England 1986].

Weather Model: Accelerator (OpenACC)



1 meteorologists coordinating 1000 math savants using tin cans and a string.

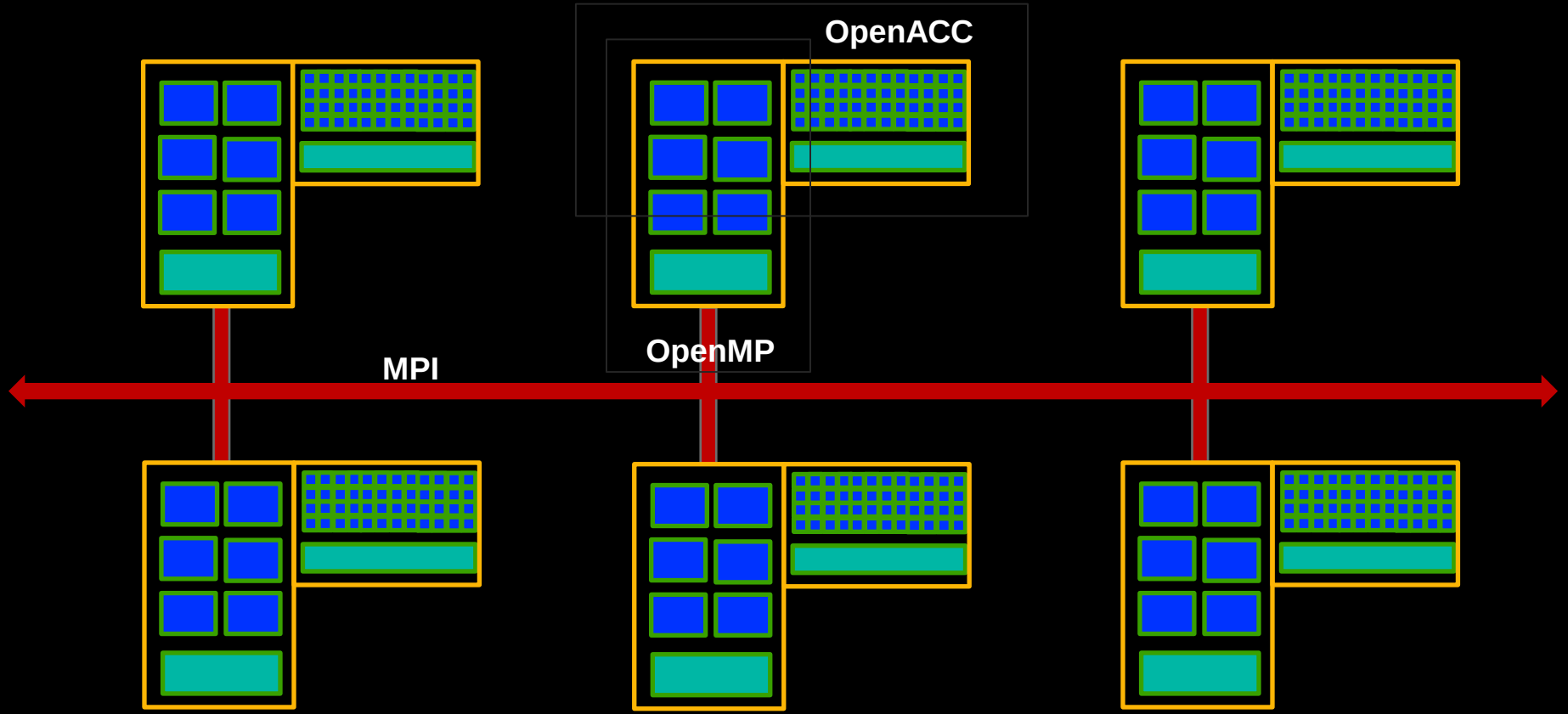
Weather Model: Distributed Memory (MPI)




```
call MPI_Send( numbertosend, 1, MPI_INTEGER, index, 10, MPI_COMM_WORLD, errcode)
.
.
call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, status, errcode)
.
.
.
call MPI_Barrier(MPI_COMM_WORLD, errcode)
.
```

50 meteorologists using a telegraph.

The pieces fit like this…



Cores, Nodes, Processors, PEs?

- A "core" can run an independent thread of code. Hence the temptation to refer to it as a processor.
- “Processors” refer to a physical chip. Today these almost always have more than one core.
- “Nodes” is used to refer to an actual physical unit with a network connection; usually a circuit board or "blade" in a cabinet. These often have multiple processors.
- To avoid ambiguity, it is precise to refer to the smallest useful computing device as a Processing Element, or PE. On normal processors this corresponds to a core.

I will try to use the term PE consistently myself, but I may slip up. Get used to it as you will quite often hear all of the above terms used interchangeably where they shouldn’t be. Context usually makes it clear.

Many Levels and Types of Parallelism

- Vector (SIMD)
- Instruction Level (ILP)
  - Instruction pipelining
  - Superscaler (multiple instruction units)
  - Out-of-order
  - Register renaming
  - Speculative execution
  - Branch prediction

Compiler
(not your problem)

OpenMP 4/5 can help!

OpenMP

- Multi-Core (Threads)
- SMP/Multi-socket

OpenACC

- Accelerators: GPU & MIC

MPI

- Clusters
- MPPs

Also Important

- ASIC/FPGA/DSP
- RAID/IO

MPPs (Massively Parallel Processors)

Distributed memory at largest scale. Shared memory at lower level.

Summit (ORNL)

- 122 PFlops Rmax and 187 PFlops Rpeak
- IBM Power 9, 22 core, 3GHz CPUs
- 2,282,544 cores
- NVIDIA Volta GPUs
- EDR Infiniband

Sunway TaihuLight (NSC, China)

- 93 PFlops Rmax and 125 PFlops Rpeak
- Sunway SW26010 260 core, 1.45GHz CPU
- 10,649,600 cores
- Sunway interconnect

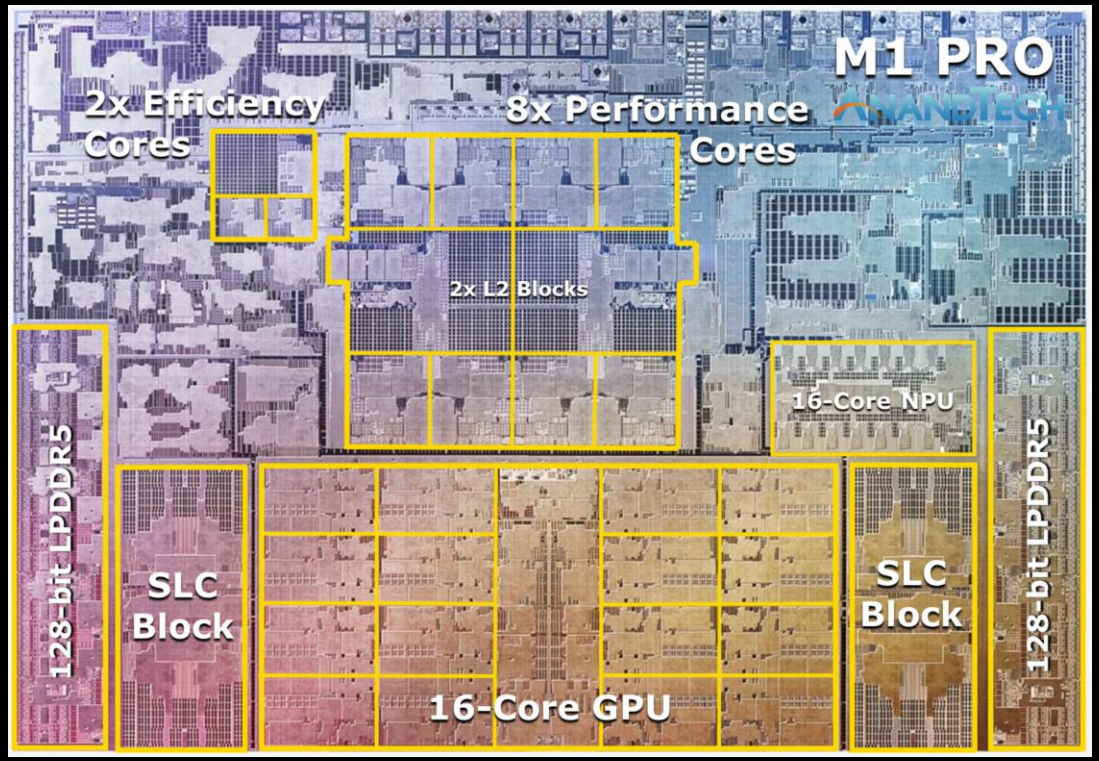
Top 10 Systems as of November 2024

#	Computer	Site	Manufacturer	CPU Interconnect [*Accelerator*]	Cores	Rmax (Pflops)	Rpeak (Pflops)	Power (MW)
1	**El Capitan**	Lawrence Livermore National Laboratory **United States**	HPE	AMD EPYC 24C 1.8GHz Slingshot-11 AMD Instinct MI300A	**11,039,616**	**1742**	**2746**	**30**
2	**Frontier**	Oak Ridge National Laboratory **United States**	HPE	AMD EPYC 64C 2GHz Slingshot-11 AMD Instinct MI250X	**9,066,176**	**1353**	**2055**	**25**
3	**Aurora**	Argonne National Laboratory **United States**	HPE	Intel Xeon Max 9470 52C 2.4GHz Slingshot-11 Intel Data Center GPU Max	**9,264,128**	**1012**	**1980**	**39**
4	**Eagle**	Microsoft **United States**	Microsoft	Intel Xeon 8480C 48C 2GHz Infiniband NDR NVIDIA H100	**1,123,200**	**561**	**846**	
5	**HPC6**	Eni S.p.A. **Italy**	HPE	AMD EPYC 64C 2GHz Slingshot-11 AMD Instinct MI250X	**3,143,520**	**477**	**606**	**8**
6	**Fugaku**	RIKEN Center for Computational Science **Japan**	Fujitsu	ARM 8.2A+ 48C 2.2GHz Torus Fusion Interconnect	**7,630,072**	**442**	**537**	**29**
7	**Alps**	Swiss National Supercomputing Center **Switzerland**	HPE	NVIDIA Grace 72C 3.1GHz Slingshot-11 *NVIDIA GH200*	**2,121,600**	**434**	**574**	**7**
8	**LUMI**	EuroHPC **Finland**	HPE	AMD EPYC 64C 2GHz Slingshot-11 AMD Instinct MI250X	**2,752,704**	**379**	**531**	**7**
9	**Leonardo**	EuroHPC **Italy**						**304**	**7**
10		Lawrenc							

500 ThinkSystem SR590, Xeon Gold 5218 16C 2.3GHz, 10G Ethernet, Lenovo Service Provider T China 108,800 2.31 4.00

The word is *Heterogeneous*

And it's not just supercomputers. It's on your desk, and in your phone.



AnandTech

How much of this can you program?

Networks

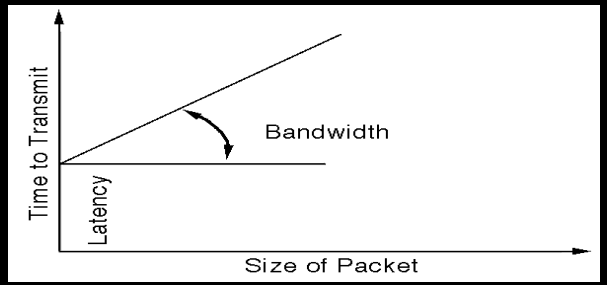
3 characteristics sum up the network:

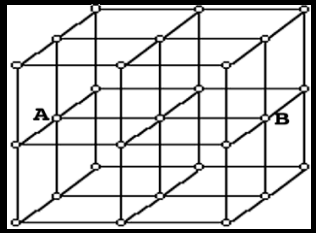
- **Latency**

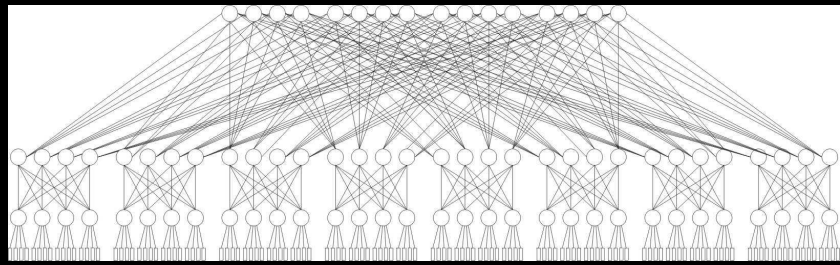
  The time to send a 0 byte packet of data on the network

- **Bandwidth**

  The rate at which a very large packet of information can be sent



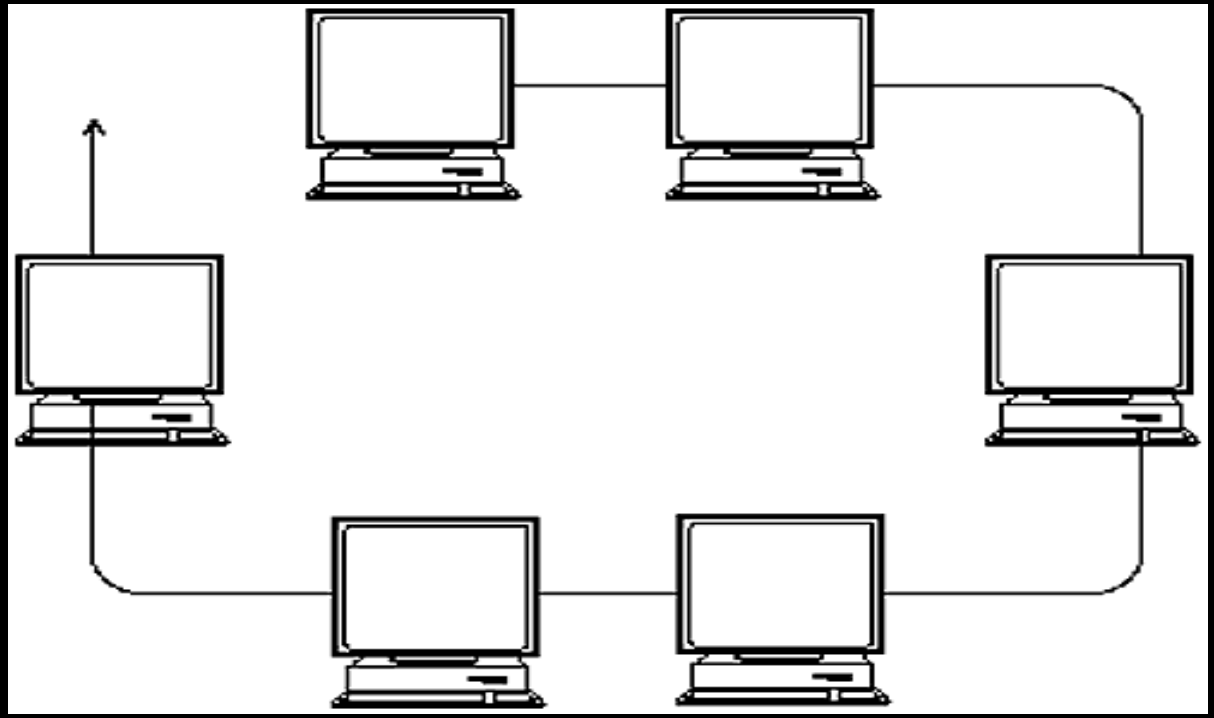




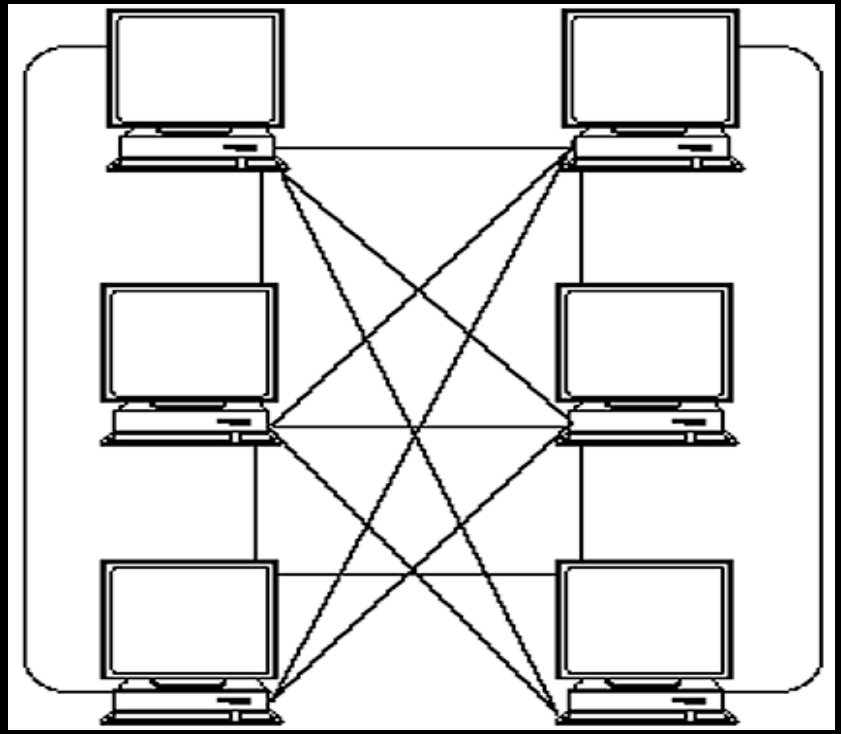
- **Topology**

  The configuration of the network that determines how processing units are directly connected.

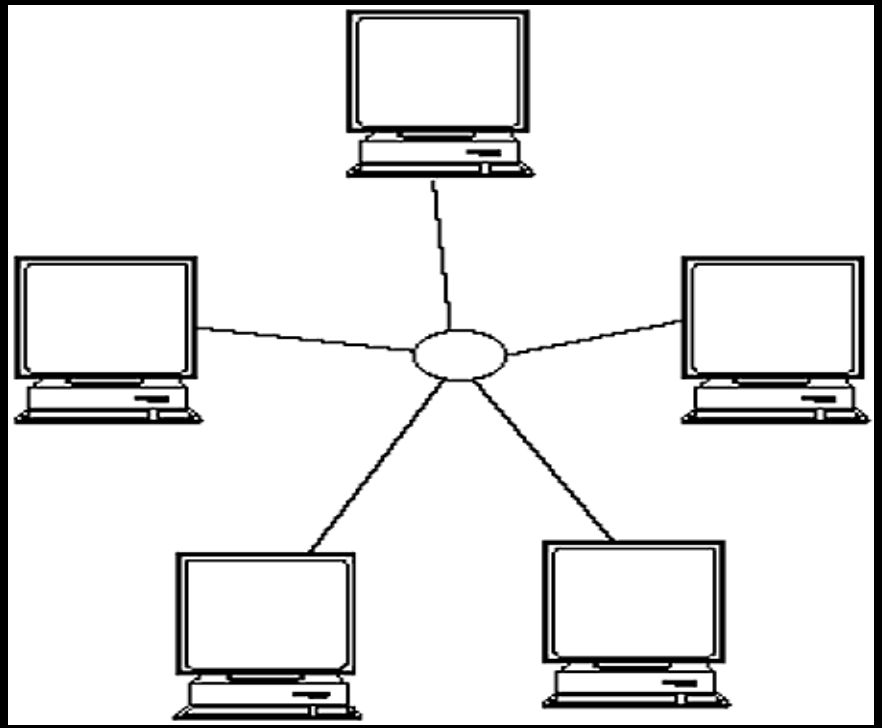
Ethernet with Workstations



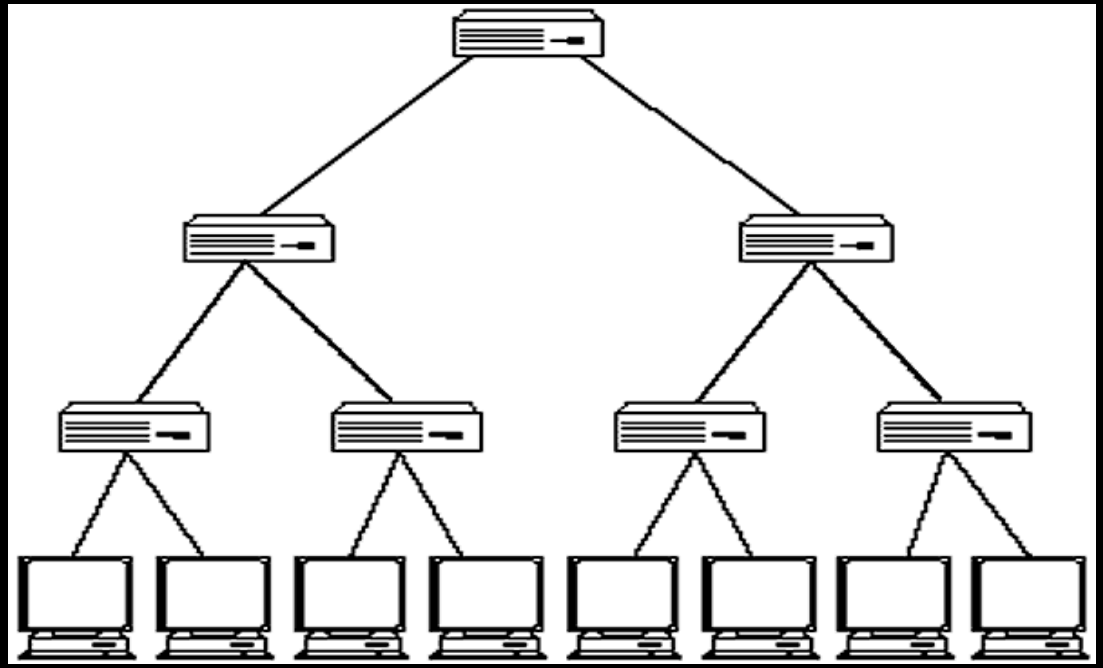
Complete Connectivity



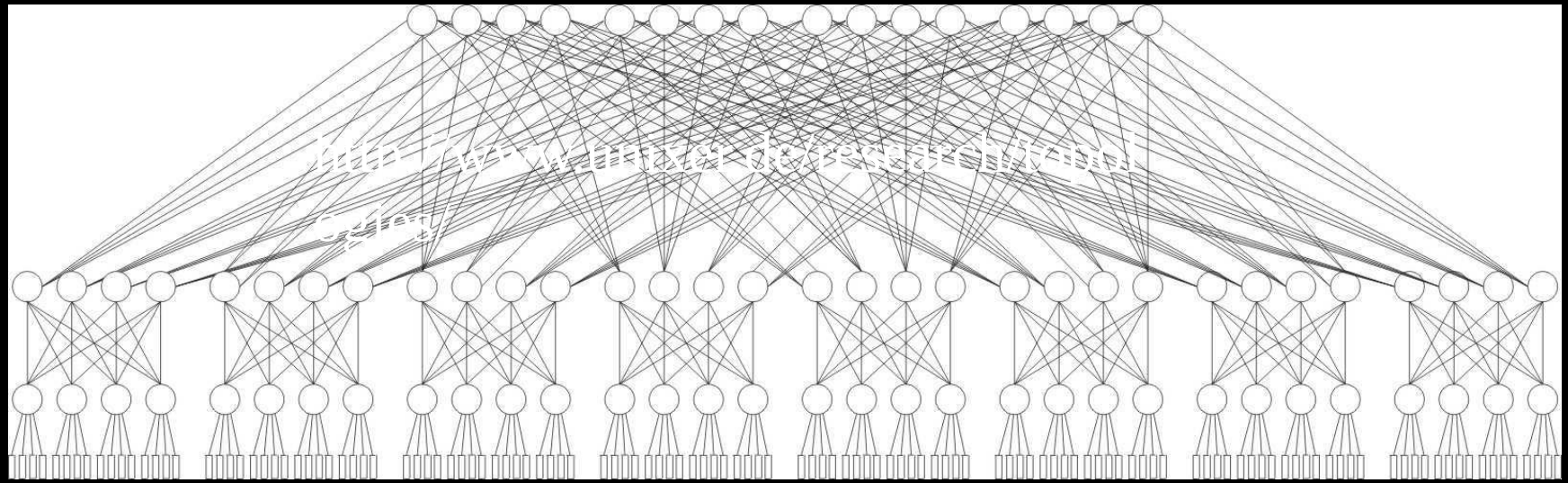
Crossbar



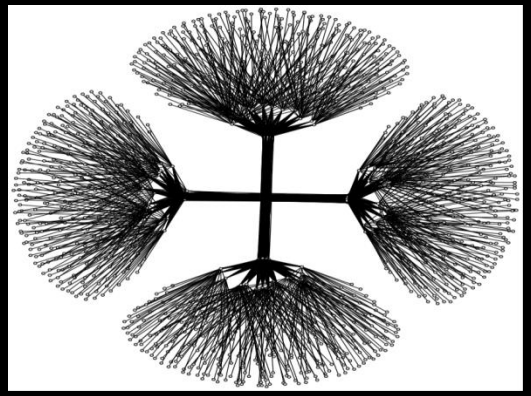
Binary Tree



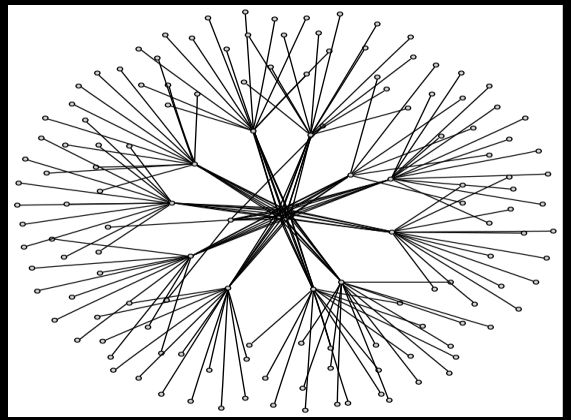
Fat Tree



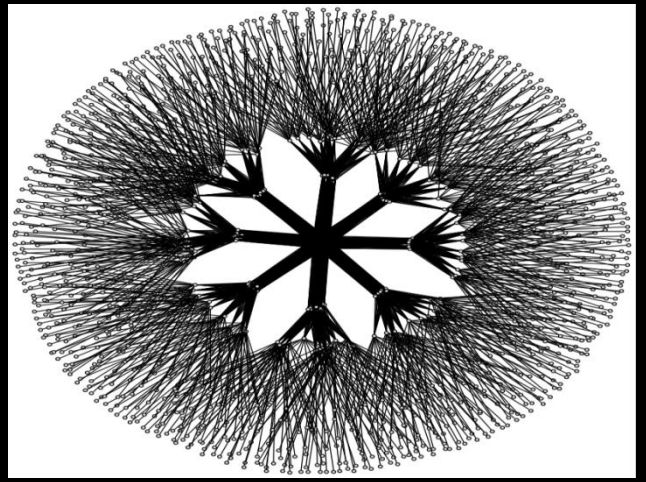
Other Fat Trees



Big Red @ IU

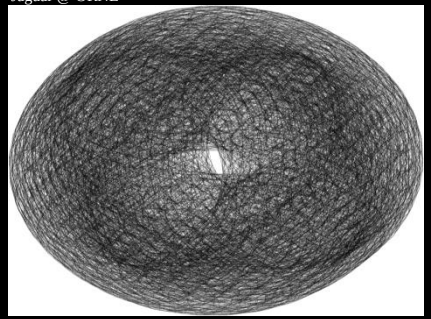


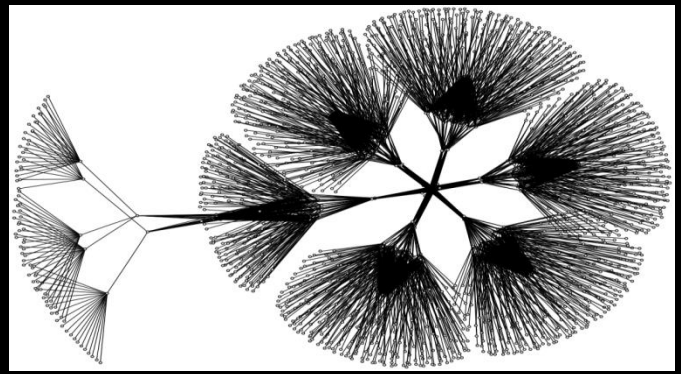
Odin @ IU



Atlas @ LLNL

Jaguar @ ORNL





Tsubame @ Tokyo Inst. of Tech

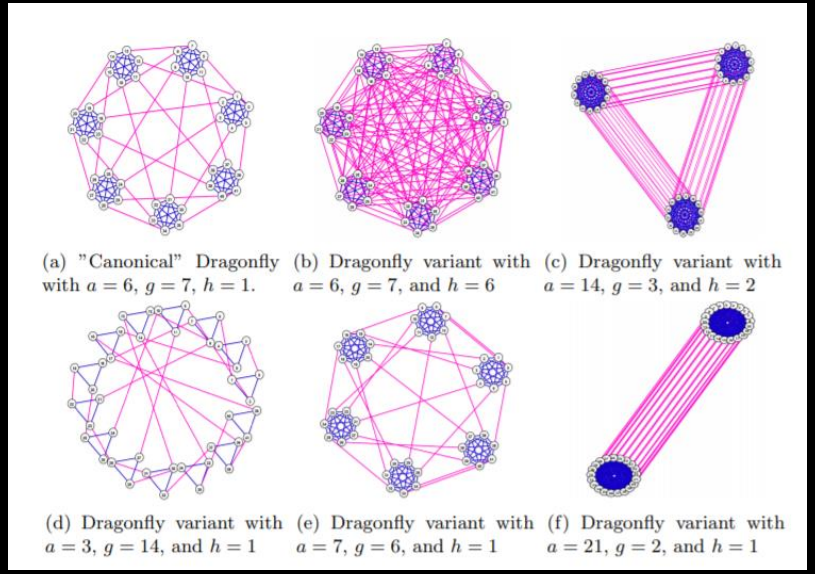
From Torsten Hoefler's Network Topology Repository at
http://www.unixer.de/research/topologies/

Dragonfly

A newer innovation in network design is the dragonfly topology, which benefits from advanced hardware capabilities like:

- **High-Radix Switches**
- **Adaptive Routing**
- **Optical Links**

Various 42 node Dragonfly configurations.

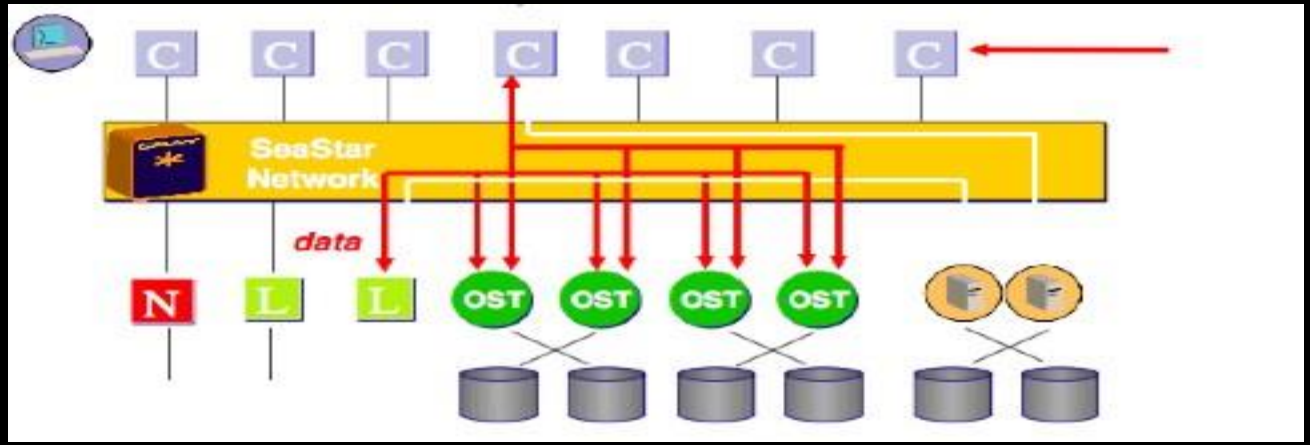


Purple links are optical, and blue are electrical.

Graphic from the excellent paper *Design space exploration of the Dragonfly topology* by Yee, Wilke, Bergman and Rumley.

Parallel IO (RAID…)

- There are increasing numbers of applications for which many PB of data need to be written.
- Checkpointing is also becoming very important due to MTBF issues (a whole ‘nother talk).
- Build a large, fast, reliable filesystem from a collection of smaller drives.
- Supposed to be transparent to the programmer.
- Increasingly mixing in SSD.



The Future Is Now!

Exascale Computing and you.

Today

- Pflops computing fully established with more than 500 machines
- The field is thriving
- Interest in supercomputing is now worldwide, and growing in many new markets
- Exascale projects in many countries and regions

Welcome to The Exascale Era!

exa = 10^{18} = 1,000,000,000,000,000,000 = quintillion

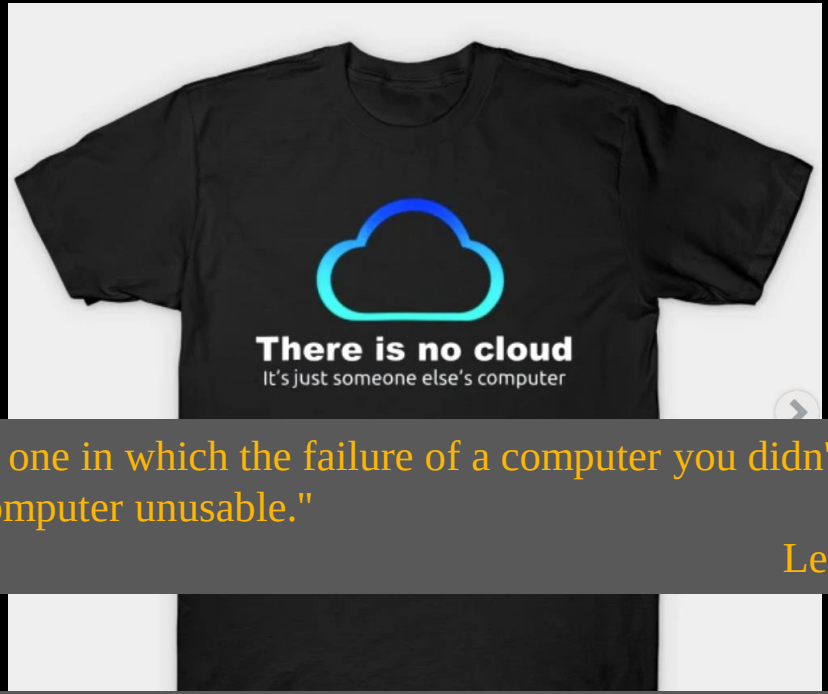
64-bit precision floating point operations per second

23,800 Cray RedStorms 2004 (42 Tflops)

23,833 NVIDIA V100 (125 Tflops)

There may also be a Chinese machine, OceanLight, or 3-letter-agency machines on the scene.

The Cloud



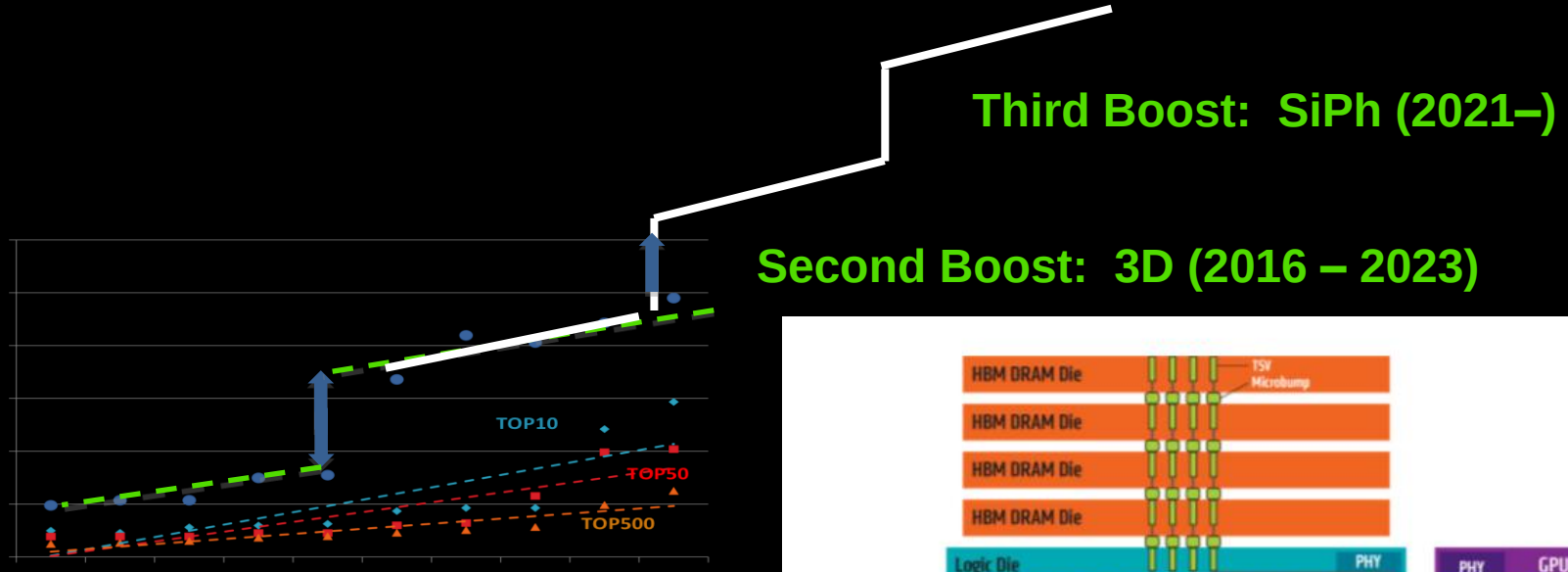
"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Leslie Lamport in 1987

"It's really more of a Fog."

Me, just now.

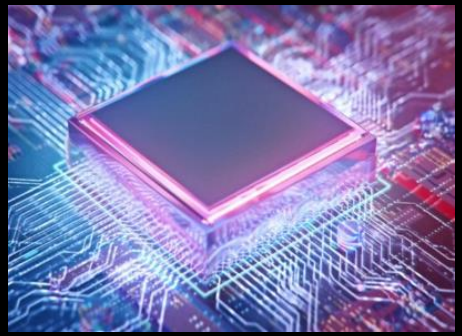
The path to Exascale has not been incremental.



Is Silicon Photonics a game changer?

Electrically switched networks can operate in “packet switching” mode to lower the effective latency and utilize all the available link bandwidth. The alternative to this mode is “circuit-switching” and it was abandoned by the electronic community long ago. Without practical means to buffer light, process photon headers in-flight, or reverting to switches with expensive optical-electrical-optical conversions, we would have to resort to circuit-switching with all the inherent deficiencies:

- **complex traffic steering calculations**
- **switching delays**
- **latency increase due to lack of available paths**
- **under-utilization of links**



Photonics is often cited as an enabler for extensive memory disaggregation, but this yields another challenge, specifically the speed of light. Photons travel at a maximum speed of 3.3 ns/m in fibers. This is equivalent to a level-2 cache access of a modern CPU, not including the disaggregation overhead (such as from the protocol, switching, or optical-electrical conversions at the endpoints). At 3–4 m distance, the photon travel time alone exceeds the first-word access latency of modern DDR memory.

A great dive into these topics can be found in *Myths and Legends in High-Performance Computing*, Matsuoka, Domke, et. al.

It is not just “exaflops” – we are changing the whole computational model

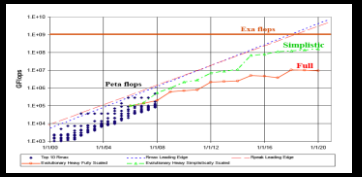
Current programming systems have WRONG optimization targets

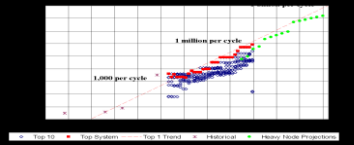
Old Constraints

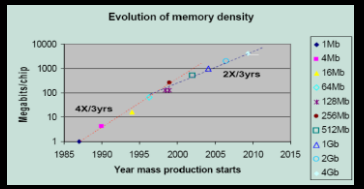
- **Peak clock frequency** ***as primary limiter for performance improvement***
- **Cost:** ***FLOPs*** **are biggest cost for system:** ***optimize for compute***
- **Concurrency: Modest growth of parallelism by adding nodes**
- **Memory scaling*: maintain byte per flop capacity and bandwidth***
- **Locality*: MPI+X model (uniform costs within node & between nodes)***
- **Uniformity:** ***Assume uniform system performance***
- **Reliability*: It’s the hardware’s problem***

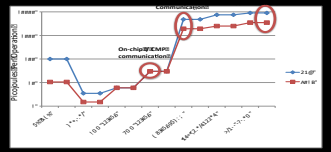
New Constraints

- **Power** *is primary design constraint for future HPC system design*
- **Cost:** *Data movement dominates: optimize to minimize data movement*
- **Concurrency:** *Exponential growth of parallelism within chips*
- **Memory Scaling:** *Compute growing 2x faster than capacity or bandwidth*
- **Locality***: must reason about data locality and possibly topology*
- **Heterogeneity**: *Architectural and performance non-uniformity increase*
- **Reliability***: Cannot count on hardware protection alone*









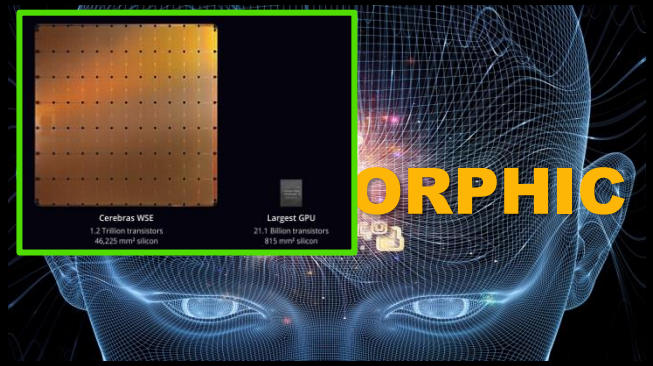
Adapted from John Shalf

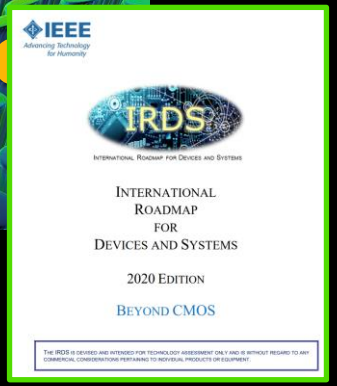
End of Moore's Law Will Lead to New Architectures

Non-von Neumann

ARCHITECTURE

von Neumann



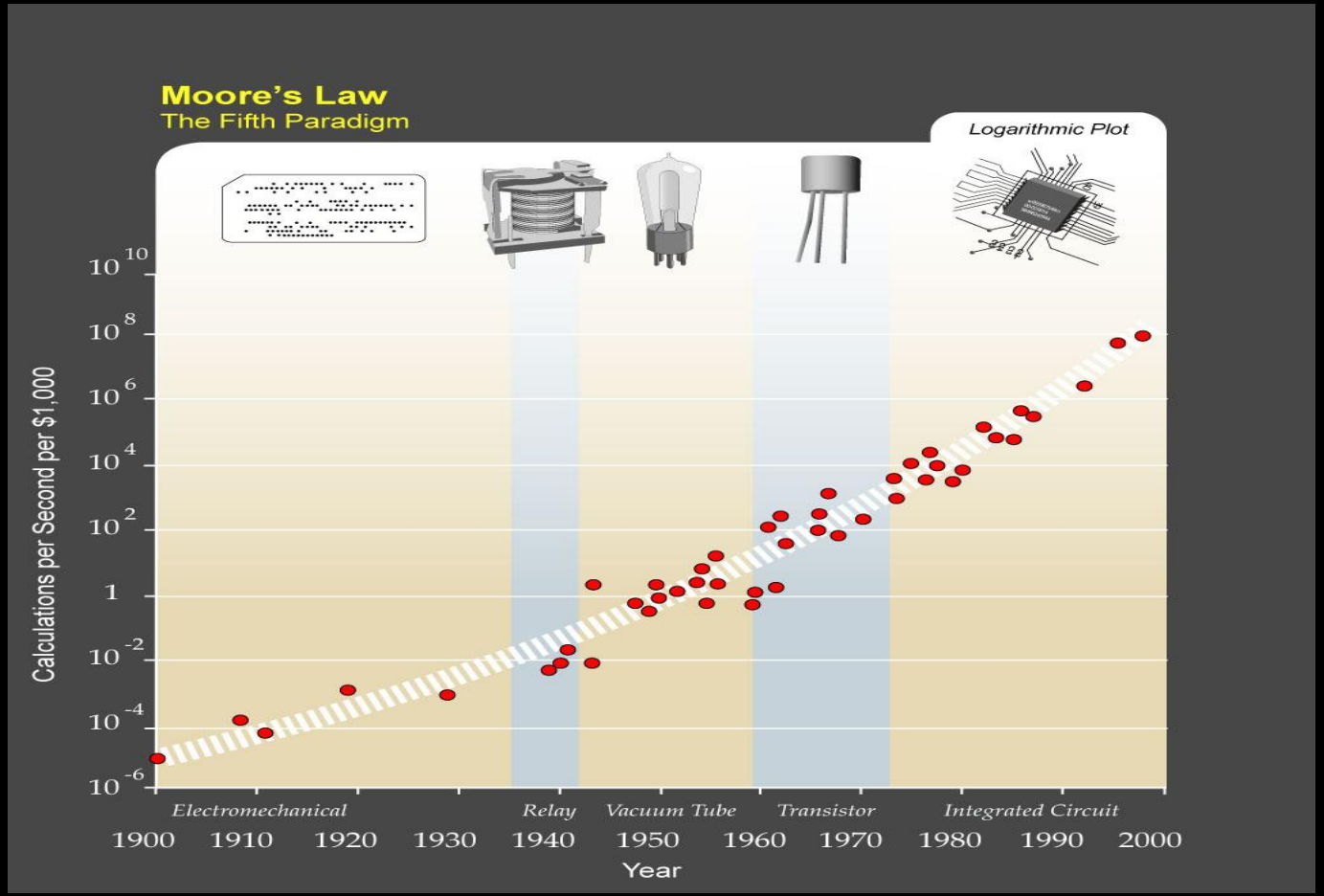


CMOS

Beyond CMOS

TECHNOLOGY

It would only be the 6th paradigm.



We can do better. We have a role model.

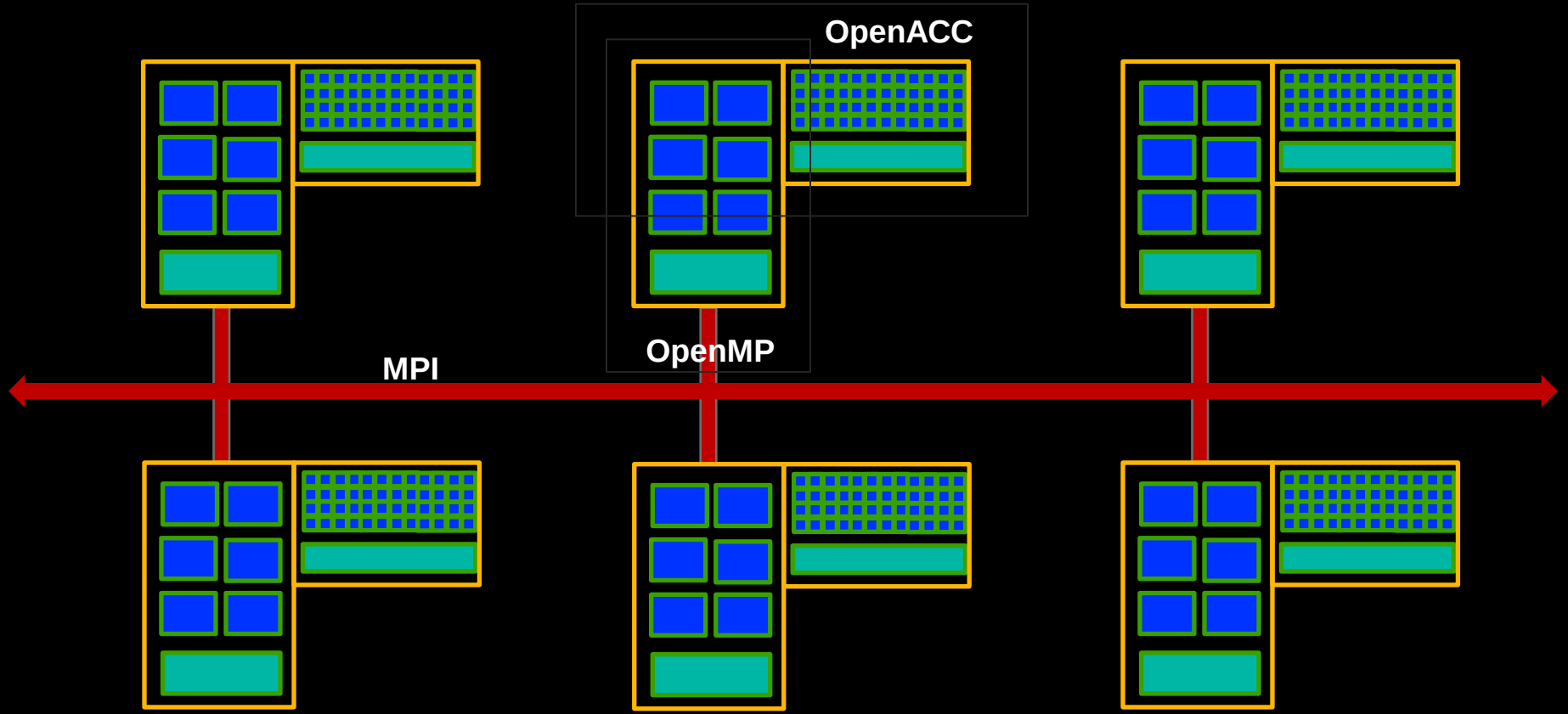
- We hope to "simulate" a human brain in real time on one of these Exascale platforms with about 1 - 10 Exaflop/s and 4 PB of memory
- These newest Exascale computers use 20+ MW
- The human brain runs at 20W
- Our brain is a million times more power efficient!

Why you should be (extra) motivated.

- This parallel computing thing is no fad.
- The laws of physics are drawing this roadmap.
- If you get on board (the right bus), you can ride this trend for a long, exciting trip.

Let’s learn how to use these things!

In Conclusion…



Compiling

We will be using standard Fortran and C compilers on this Pathway. They should look familiar.

We will slightly prefer the NVIDIA compilers (the Intel or gcc or AMD or Clang ones would also be fine).

Note that on Delta you would normally have to enable this compiler with

```
module load nvhpc/24.1
```

I have put that in the .bashrc file that we will all use.

Configure Your Environment

Let's get the boring stuff out of the way now. This will configure your environment by copying over the exercise files and setting up your .bashrc so that you can do our initial exercises, as well as exercises for the following two modules.

- Log on to Delta.

- Run the setup script that will copy over the Exercises directory we will all use. It will also automatically load the right compiler using your .bashrc script whenever you login. Note that linux shells are case sensitive.

```
/projects/becs/urbanic/Setup
```

- As told, logout and log back on again to complete the setup. You won't need to do that in the future.

Our Files For This Pathway

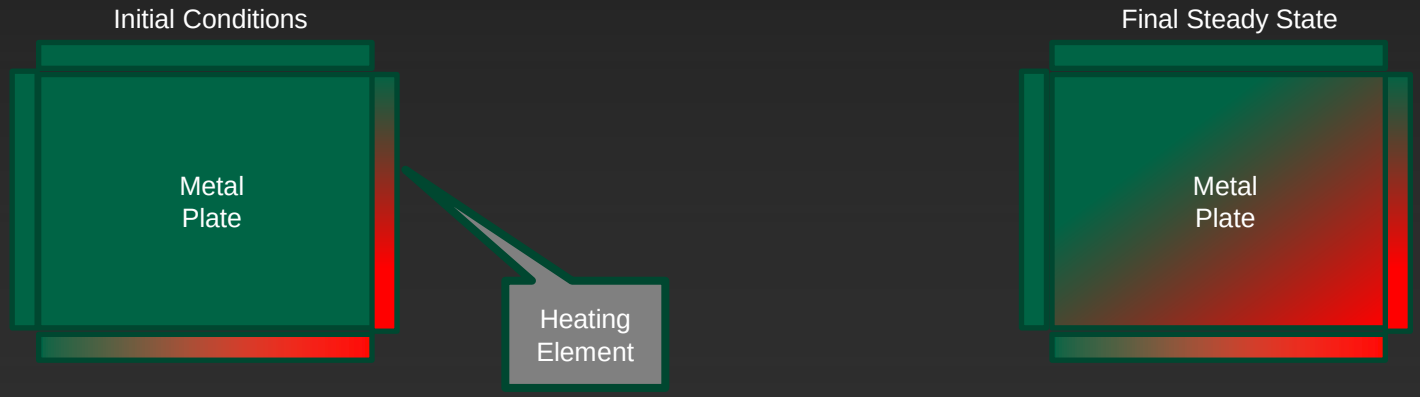
After you run the setup script, you will have the following directories in your home directory. We will be using them in future modules.

```
/Exercises
        /Test
        /OpenMP
        /OpenACC
        /MPI
```

Our Foundation Exercise: Laplace Solver

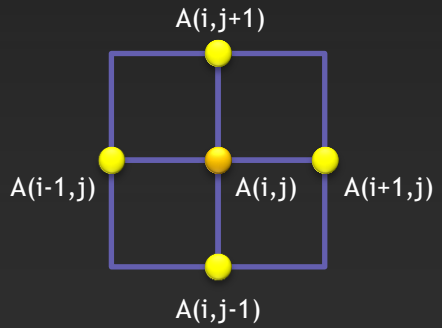
- I've been using this for MPI, OpenMP and now OpenACC. It is a great simulation problem, not rigged for MPI.
- In this most basic form, it solves the Laplace equation: $\nabla^2 f(x, y) = 0$
- The Laplace Equation applies to many physical problems, including:
  - Electrostatics
  - Fluid Flow
  - Temperature

- For temperature, it is the Steady State Heat Equation:



Exercise Foundation: Jacobi Iteration

- The Laplace equation on a grid states that each grid point is the average of its neighbors.
- We can iteratively converge to that state by repeatedly computing new values at each point from the average of neighboring points.
- We just keep doing this until the difference from one pass to the next is small enough for us to tolerate.



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Serial C Code (kernel)

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

     for(i = 1; i <= ROWS; i++) {
          for(j = 1; j <= COLUMNS; j++) {
               Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                           Temperature_last[i][j+1] + Temperature_last[i][j-1]);
          }
     }

     dt = 0.0;

     for(i = 1; i <= ROWS; i++){
          for(j = 1; j <= COLUMNS; j++){
               dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
               Temperature_last[i][j] = Temperature[i][j];
          }
     }

     if((iteration % 100) == 0) {
          track_progress(iteration);
     }

     iteration++;

}
```

Done?

Calculate

Update temp array and find max change

Output

Whole C Code

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

// size of plate
#define COLUMNS    1000
#define ROWS       1000

// largest permitted change in temp (This value takes about 3400 steps)
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2];      // temperature grid
double Temperature_last[ROWS+2][COLUMNS+2]; // temperature grid from last iteration

//   helper routines
void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

    int i, j;                                            // grid indexes
    int max_iterations;                                  // number of iterations
    int iteration=1;                                     // current iteration
    double dt=100;                                       // largest change in t
    struct timeval start_time, stop_time, elapsed_time;  // timers

    printf("Maximum iterations [100-4000]?\n");
    scanf("%d", &max_iterations);

    gettimeofday(&start_time,NULL); // Unix timer

    initialize();                   // initialize Temp_last including boundary conditions

    // do until error is minimal or until max steps
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        // main calculation: average my four neighbors
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                            Temperature_last[i][j+1] + Temperature_last[i][j-1]);
            }
        }

        dt = 0.0; // reset largest temperature change

        // copy grid to old grid for next iteration and find latest dt
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
              Temperature_last[i][j] = Temperature[i][j];
            }
        }

        // periodically print test values
        if((iteration % 100) == 0) {
            track_progress(iteration);
        }

        iteration++;
    }

    gettimeofday(&stop_time,NULL);
    timersub(&stop_time, &start_time, &elapsed_time); // Unix time subtract routine

    printf("\nMax error at iteration %d was %f\n", iteration-1, dt);
    printf("Total time was %f seconds.\n", elapsed_time.tv_sec+elapsed_time.tv_usec/1000000.0);

}

// initialize plate and boundary conditions
// Temp_last is used to to start first iteration
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}

// print diagonal in bottom right corner where most action is
void track_progress(int iteration) {

    int i;

    printf("---------- Iteration number: %d ------------\n", iteration);
    for(i = ROWS-5; i <= ROWS; i++) {
        printf("[%d,%d]: %5.2f  ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```

Whole Fortran Code

```
program serial
      implicit none

      !Size of plate
      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      double precision, parameter    :: max_temp_error=0.01

      integer                        :: i, j, max_iterations, iteration=1
      double precision               :: dt=100.0
      real                           :: start_time, stop_time

      double precision, dimension(0:rows+1,0:columns+1) :: temperature, temperature_last

      print*, 'Maximum iterations [100-4000]?'
      read*,   max_iterations

      call cpu_time(start_time)      !Fortran timer

      call initialize(temperature_last)

      !do until error is minimal or until maximum steps
      do while ( dt > max_temp_error .and. iteration <= max_iterations)

         do j=1,columns
            do i=1,rows
               temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                      temperature_last(i,j+1)+temperature_last(i,j-1) )
            enddo
         enddo

         dt=0.0

         !copy grid to old grid for next iteration and find max change
         do j=1,columns
            do i=1,rows
               dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
               temperature_last(i,j) = temperature(i,j)
            enddo
         enddo

         !periodically print test values
         if( mod(iteration,100).eq.0 ) then
            call track_progress(temperature, iteration)
         endif

         iteration = iteration+1

      enddo

      call cpu_time(stop_time)

      print*, 'Max error at iteration ', iteration-1, ' was ',dt
      print*, 'Total time was ',stop_time-start_time, ' seconds.'

end program serial
```

```
! initialize plate and boundery conditions
! temp_last is used to to start first iteration
subroutine initialize( temperature_last )
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,j

      double precision, dimension(0:rows+1,0:columns+1) :: temperature_last

      temperature_last = 0.0

      !these boundary conditions never change throughout run

      !set left side to 0 and right to linear increase
      do i=0,rows+1
         temperature_last(i,0) = 0.0
         temperature_last(i,columns+1) = (100.0/rows) * i
      enddo

      !set top to 0 and bottom to linear increase
      do j=0,columns+1
         temperature_last(0,j) = 0.0
         temperature_last(rows+1,j) = ((100.0)/columns) * j
      enddo

end subroutine initialize

!print diagonal in bottom corner where most action is
subroutine track_progress(temperature, iteration)
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,iteration

      double precision, dimension(0:rows+1,0:columns+1) :: temperature

      print *, '---------- Iteration number: ', iteration, ' ---------------'
      do i=5,0,-1
         write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no'), &
                    rows-i,columns-i,temperature(rows-i,columns-i)
      enddo
      print *
end subroutine track_progress
```

Exercises For This Module.

Our first exercises will be to compile, run and time the Laplace code using the two programming models in this Pathway: OpenMP and MPI.

This will get you familiar with the programming environment in preparation for our actual parallel programming in the next two modules.

It will also allow you to experience some parallel scaling first hand.

Specifically, our goal will be to compile the Laplace code in OpenMP and run it on varying numbers of *threads* to see how much it speeds up.

We will do the same thing using MPI to run with multiple *processes*.

Don't worry if it seems like we are skipping over some details today - we are. But those details will be made clear in our following modules.

OpenMP (Exercise 1)

Go into the OpenMP folder inside the Exercises folder. You will see codes there called *laplace_omp.c* and *laplace_omp.f90*. Select whichever language most interests you.

To compile with OpenMP we do either

```
nvc -mp laplace_omp.c
```

or

```
nvfortran -mp laplace_omp.f90
```

Now we have an executable called *a.out*. But we need to ask for a compute node with multiple cores allocated in order to run. The Slurm command to get us a command line (*--pty bash*) on a compute node (*--nodes=1*) with 32 cores (*--cpus-per-task=32*) is:

```
srun --account=becs-delta-cpu --partition=cpu-interactive  --nodes=1 --cpus-per-task=32 --pty bash
```

*I am assuming the we all either have the same *--account*, or you know which one you should be using.

OpenMP

You will notice a few messages as Slurm find the resources, and then your command line will change to something with a compute node number in it.

From the command line on this compute node we can run up to 32 cores. OpenMP allows us to control core usage with the *OMP_NUM_THREADS* environment variable. You learned about these in the Intro to Delta module.

Set this variable to request 1 core (*export OMP_NUM_THREADS=1*) and run with *a.out* to find the baseline run time for the Laplace code. If you select 4000 iterations, the code will run to a complete solution. It reports its own runtime for you.

Now, try varying number of cores up to 32 and see what kind of speedup you experience. Note that you do not need to recompile the code. Just change the environment variable and run a.out. Record these times for our discussion.

exit the compute node when you are finished. You should find yourself returned to the login node.

MPI (Exercise 2)

Now we will do a similar exercise using MPI. Go into the MPI folder inside the Exercises folder. You will see codes there called *laplace_mpi.c* and *laplace_mpi.f*. Select whichever language most interests you.

To compile with MPI we do either

```
mpicc laplace_mpi.c
```

or

```
mpif90 laplace_mpi.f90
```

Again, you have an executable called *a.out*. Now you need to ask for a compute node with multiple processes allocated in order to run. Similar, but not identical, to the previous Slurm command, the one to get us a command line on a compute node with 4 processes (`--nodes=1 --tasks=4 --tasks-per-node=4`) is:

```
srun --account=becs-delta-cpu --partition=cpu-interactive  --nodes=1 --tasks=4 --tasks-per-node=4 --pty bash
```

MPI

With MPI we will limit ourselves to a single timing run, using 4 processes. The command to run our *a.out* executable on our four available processes is

```
mpirun -n 4 a.out
```

Record this time for our later discussion.

Exit the compute node when you are finished.