CISC 372: Parallel Computing

Introduction

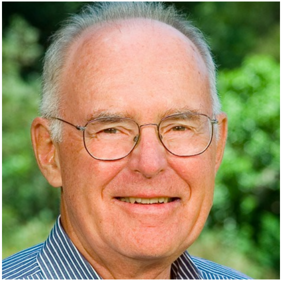
Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

August 31, 2020

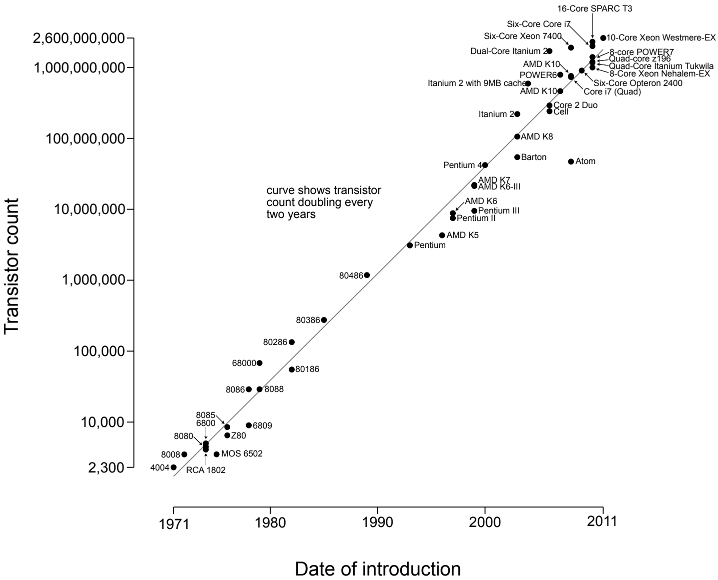
Moore's Law

- Gordon Moore, co-founder of Intel
- integrated circuits invented in 1958
- Moore made prediction in a 1965 paper
- the density of transistors in CPUs doubles every 1 or 2 years



- as number of transistors increases
  - CPUs become more complex
  - can do more in one clock cycle
  - more instructions
  - instruction-level parallelism: pipelining, ...
- as distance between transistors decreases
  - the time for current to travel between them decreases
  - clock frequency can be increased
  - computing capability increases

Microprocessor Transistor Counts 1971-2011 & Moore's Law

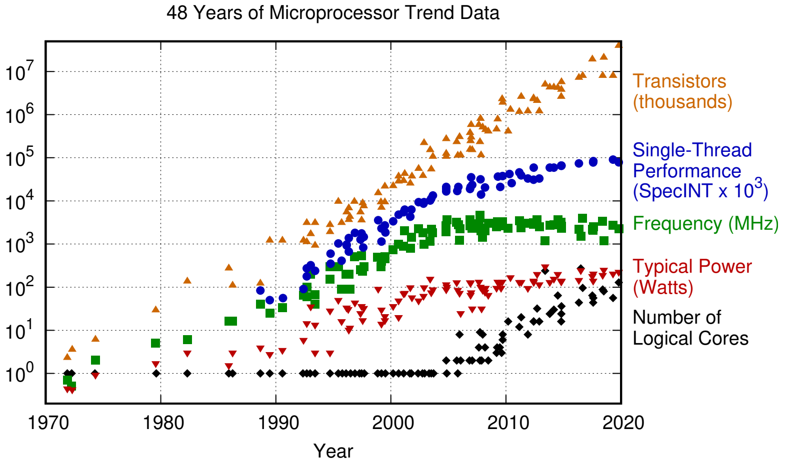


Source: `https://commons.wikimedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg`

Units of measurement

- CPU frequency is measured in Hertz (**Hz**)
  - number of cycles per second
    - cycle: smallest unit of time in which state of processor can change
  - equivalent to $1/\mathbf{s}$ ($\mathbf{s}$ = second)
  - typical speed of modern CPU: 2.5 **GHz** $= 2.5 \times 10^9$ **Hz**
- power consumption is measured in Watts (**W**)
  - power is energy per unit time
  - one Watt = 1 Joule per second (1 **J**/**s**)
    - example: a 30 **W** processor running for one hour consumes $(30\ \mathbf{J}/\mathbf{s})(3600\ \mathbf{s}) = 108,000\ \mathbf{J}$
  - 1 kilowatt hour (**kWh**) is also a unit of energy:
    - total amount of energy consumed by consuming 1000 Watts for one hour
    - $(1000\ \mathbf{W})(3600\ \mathbf{s}) = (1000\ \mathbf{J}/\mathbf{s})(3600\ \mathbf{s}) = 3,600,000\ \mathbf{J}$
  - one Watt = one Volt times one Ampere
    - rate at which work is done when one Ampere of current flows through an electrical potential difference of 1 Volt

Microprocessor trends through 2020



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 2010-2019 by K. Rupp

- Moore's law still holding!
- frequency peaked around 2005 ($\sim$ 3GHz) . . . multiple cores took off
- single-thread performance barely increasing now

Problems arose around 2005

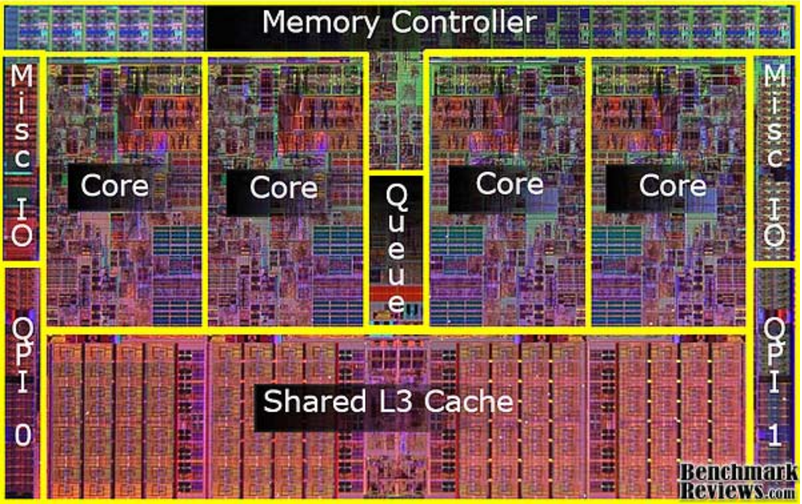
- harder to find ways to speed things up with more transistors
- increasing frequency requires much more power
  - relationship between power and CPU frequency is complicated
  - power consumption is roughly proportional to cube of frequency
- energy is a valuable resource! (climate change, . . .)
- energy consumed by processor is converted into heat
  - the heat must be removed in some way (liquid nitrogen cooling?!)
  - when transistors are so small and packed so closely together, they can be easily damaged by heat
- solution: instead of trying to make processors faster, put more processors on the integrated circuit ("chip")
- these "multi-core" chips have multiple processors (cores) on one chip

Example: one fast core vs. two slower cores

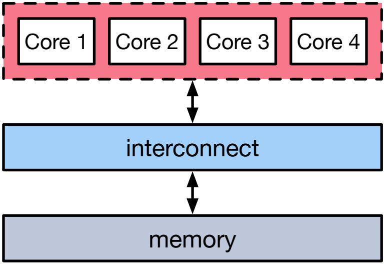
- CPU1 has one core
  - clock speed: x (measured in **Hz**)
  - power consumption: y (measured in **W**)
- CPU2 has two of the same cores running 30% slower
  - clock speed: $(.7)x$
  - power consumption: $2(.7)^3y = .686y$
- comparison
  - CPU2 uses less than 70% of the power of CPU1
  - CPU2 has $2(.7) = 1.4$ times the compute capability of CPU1
  - CPU2 is a win-win!
  - but this assumes programs can utilize the two cores concurrently!
    - and with 0 overhead

reference: `https://www.comsol.com/blogs/havent-cpu-clock-speeds-increased-last-years`

Intel Core i7 Nehalem CPU



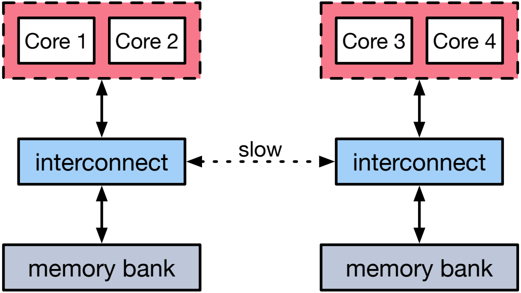
Multi-core memory organization: UMA



UMA: Uniform Memory Access

- all cores can access any memory location
- access time is independent of memory location

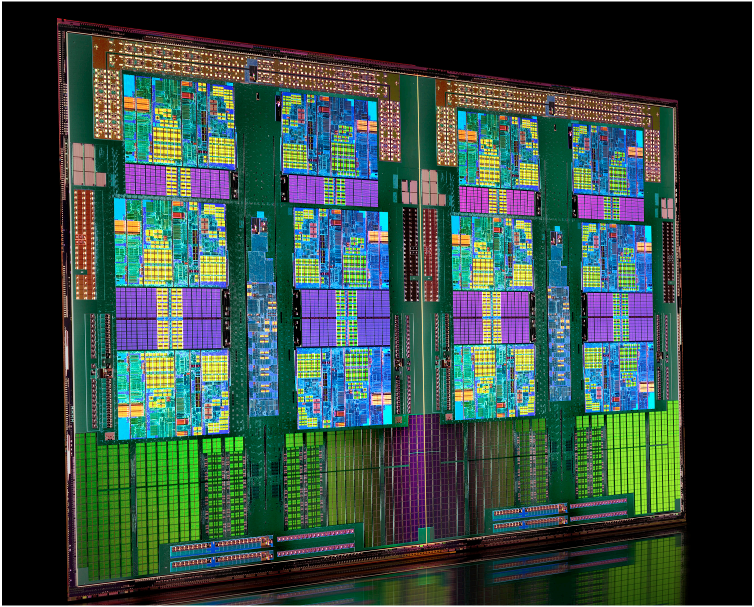
Multi-core memory organization: NUMA



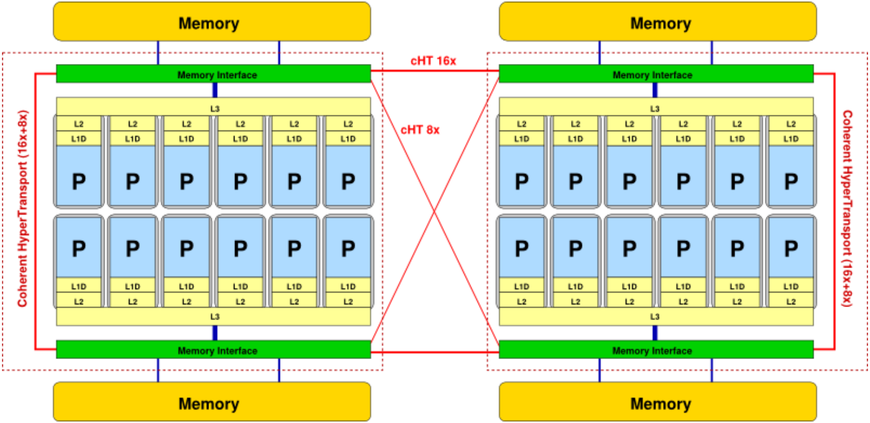
NUMA: Non-Uniform Memory Access

- all cores can access any memory location
- memory divided into banks; each core has an affinity to one bank
- core can access affiliated bank faster than another bank
- locality is vital for good performance
  - want most memory accesses to be to affiliated bank

12-core AMD Magny-cours CPU “package”



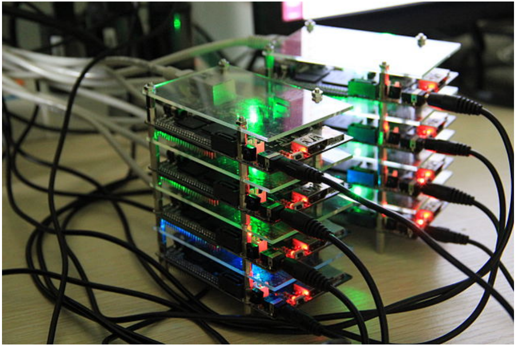
Structure of multi-socket AMD Magny-cours system



- system can have 2 or 4 sockets on a card
- each socket contains a CPU "package" consisting of 2 dies
- each die comprises 6 cores
- each die has L3 cache, memory interface, and affiliated memory bank (NUMA!)

Distributed systems and Clusters

- parallel computing has been around a long time . . . long before multi-cores
- a cluster is a set of computers connected by a network to form a single computing system



Cubbiboard network

https://en.wikipedia.org/wiki/Computer_cluster#/media/File:Cubieboard_HADOOP_cluster.JPG

Clusters

- origins in the mid-1960s
- exploded in the 1980s and 1990s in high-performance computing
- enabled building of supercomputers from commodity hardware
  - commodity=inexpensive because components are mass-produced
  - examples: cheap PCs; mass-produced Intel or AMD CPUs
- extremely scalable: up to hundreds of thousands of nodes
- networking technologies: Ethernet, Infiniband, ...

Mills Cluster, UD

Multi-cores vs. clusters

Multi-cores:

- processors on same chip can share resources such as cache on chip
- communication and coordination on chip extremely fast and power-efficient
- limited scalability (no 100,000-core chip yet)

Clusters:

- no shared cache
- all communication must go over network; slower and less power-efficient than within a chip
- virtually no limit to scalability

Today all modern supercomputers combine features of both

- clusters of multi-core nodes
- see `https://www.top500.org/lists/2018/06/`

Programming models

1. message-passing
   - each node runs its own program or "process"
   - there are no variables shared by processes
   - processes communicate by calls to message-passing functions
     - process 1: "send the data in array a to process 17"
     - process 17: "receive data from process 1 and store in array b"
2. shared variables
   - one program spawns multiple "threads"
   - threads communicate by writing to and reading from shared variables
     - thread 1 : `x := (t1+t2)/2;`
     - thread 17: `t3 := x;`

Message-passing vs. shared variables

If you want to "parallelize" a sequential program. . .

- using message-passing:
  1. data structures (e.g., arrays) must be distributed across the processes
     - need to translate between the local index and original global index
  2. new communication commands/functions needed: `send`, `receive`, . . .
  3. local variables of one process can never be changed by another
     - you can safely reason about one process much like a sequential program
  4. locality is easily expressed — helping achieve good performance
- using shared variables:
  1. data structures mostly unchanged
     - "global view of data" : same as in original program
  2. communication accomplished by plain old assignment statements
  3. local variables of one thread might be changed at any time by another thread
     - local reasoning difficult; careful coordination/synchronization needed
  4. locality can be difficult to specify — achieveing good performance can be difficult

Message-passing vs. shared variables

Implementation:

- implementing message-passing systems is relatively easy
  - no need to modify the compiler or operating system
  - can be implemented as a C library (MPI)
  - many robust (and free) implementations exist — for many years
- implementing shared-variable systems is hard
  - new programming languages : research
  - libraries (Pthreads) : requires OS support
  - pragma/annotation systems (OpenMP): compiler and OS support

You might think:

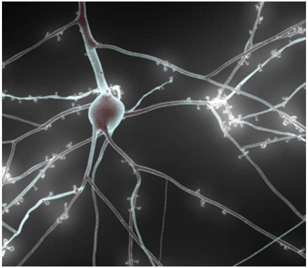
- message-passing is the best model for clusters
- shared variables is the best model for multi-cores

but it's not that simple.

- message-passing models can be used to program multi-cores
  - MPI, MCAPI. (actually very effective)
- shared variable models can be used to program clusters
  - Chapel, UPC. (this is more experimental)

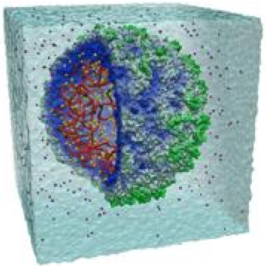
Some applications of scientific computing

- Blue Brain Project (IBM/EPFL)
  - brain model based on biologically realistic model of neuron
  - 2005: model of single neuron
  - 2008: simulation of neocortical column (10,000 neurons)
  - 2011: simulation of mesocircuit (100 neocortical columns)
  - 2015: simulation of part of rat's primary somatosensory cortex
    - 31,000 cells; 37 million synapses
  - findings: evidence for "innate knowledge"
    - http://actu.epfl.ch/news/new-evidence-for-innate-knowledge-5/
  - goal: simulation of full human brain



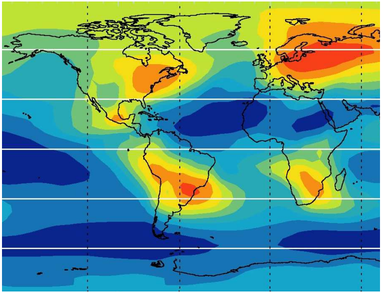
Some applications of scientific computing

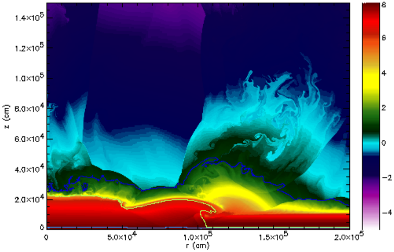
- Molecular Dynamics
  - 2006: complete atomic simulation of satellite tobacco mosaic virus
  - around 1 million atoms
  - software: NAMD
  - time resolution: femto-second (one millionth of one billionth sec.)
  - duration: 50 billionths sec.
  - findings
    - virus pulses asymmetrically
    - collapses without genetic material (implications on reproduction)



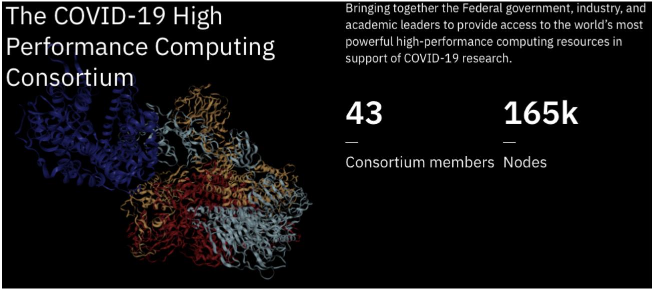
Some applications of scientific computing

- climate modeling and weather prediction
- drug discovery
- atomic structure
- development of new batteries
- design and simulation of nuclear reactors
- design of buildings, automobiles, aircraft, ships
- simulation of astronomical phenomena
  - supernovae
  - galaxy formation and collision
  - the Big Bang
- geological modeling





Applications: COVID-19 Research



- molecular simulations to screen 8,000 compounds
  - IBM's Summit supercomputer
  - find which bind to the coronavirus "spike" protein
  - 77 recommended for testing
- Simulation of urban transportation systems for return to operations
- Integrative modeling of SARS-COV2 envelope structure
- COVID Mapping and Modeling for City of Philadelphia

How this class will work

Before each class, you should...

- watch the video(s)
- read the notes if you want
- take the quiz

In class we will...

- review the quiz and selected homework solutions
- answer questions, work out any issues, ...
- work on breakout problems in small groups
  - you commit your solutions to your personal repo for participation credit
- present your solutions to breakout problems
- amuse bouche: preview of next subject/video

Homework

- homework is assigned each Tuesday
- due Wednesday morning (9:30 AM) of the next week
- late (5% penalty): between 9:30 AM and 9:30 PM Wednesday
- late (15% penalty): between 9:30 PM Wednesday and 9:30 AM Thursday
- solutions are released 9:30 AM Thursday
  - no solutions can be accepted after this time (0 points)
- it is always better to submit something rather than nothing
- it is always better to submit a program that compiles and gets some answers right, than a program that doesn't compile

Grading

- two exams, during class time: Oct 8, Nov 19
- no final exam
- final project (groups of 2)

Quizzes	5%
Participation	5%
Homework	40%
Exam 1	15%
Exam 2	15%
Final Project	20%

Resources available to you

- free online textbooks
- many example programs from other authors, sources
- free online tutorials
- office hours (10 hours per week between me and 2 TAs)
- Slack: ask and answer questions, discuss issues

See the syllabus for a complete list.

Mechanics

Two Subversion repositories:

1. public repo
   - `svn://grendel.cis.udel.edu/372-2020F`
   - how we distribute material to you
   - homework assignments, code examples, notes, documents
2. your personal repo
   - `svn://grendel.cis.udel.edu/372-USER`
   - how you submit material to us
   - homework solutions and breakout solutions

Canvas is used for gradebook and videos.

What you have to do next

- read the syllabus!
- get your EECIS account
- watch the two Unix videos and take the Unix quiz before class Thursday
- start working on HW1, due next week
- optional: submit an intro video of yourself (Canvas Assignment)
- take the Intro quiz (on today's lecture)
  - some time before end of next week