COSC 407
Intro to Parallel Computing

Topic 1: Introduction to Computer Parallelism

1

Changing times

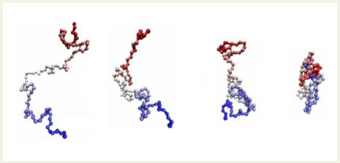
- From 1986 – 2002, performance of microprocessors increased, on average, 50% per year.
- Since then, it's dropped to about 20% increase per year.
- Now, it has dropped to even less % per year.
- But we still need high performance in many applications…
  - e.g., NP problems, gaming, big data analysis, …
- Why we need ever-increasing performance?
  - Computational power is increasing, but so are our computation problems and needs.
  - Problems we never dreamed of have been solved because of past increases, such as decoding the human genome.
  - More complex problems are still waiting to be solved.

2

Open Areas…

Climate modeling

Data analysis



Protein folding

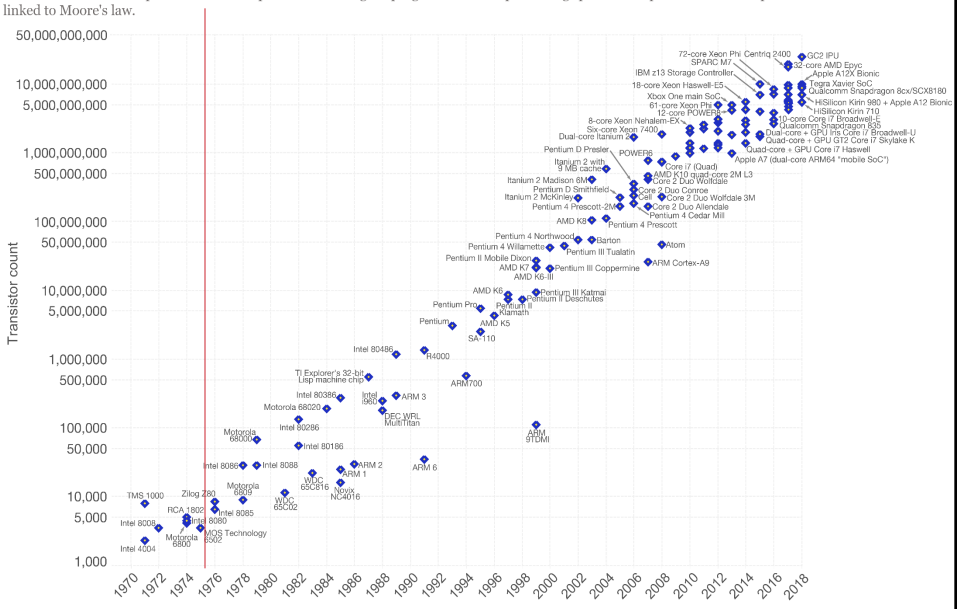
Energy research

3

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

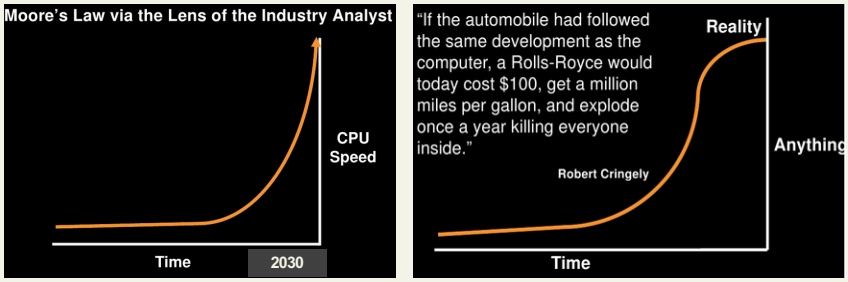
Our World in Data

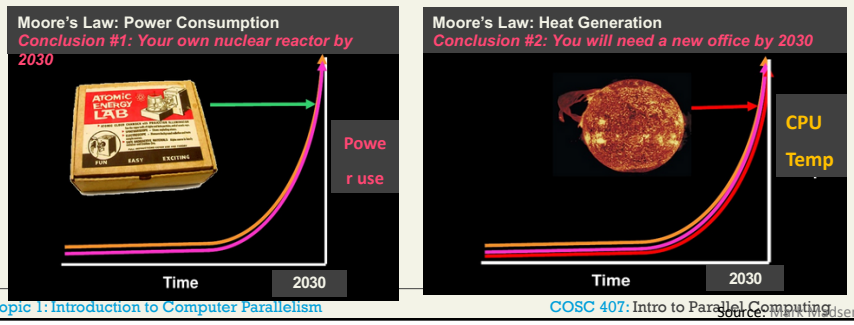


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

4

How to Predict Like an Analyst



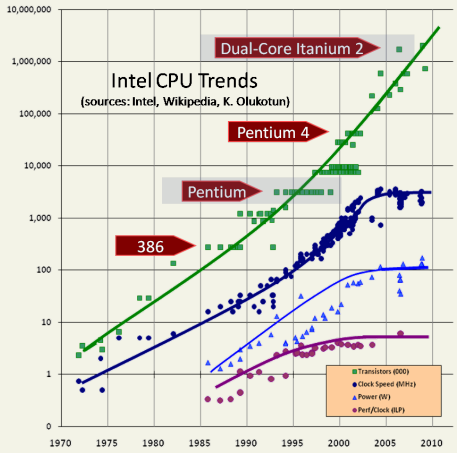


Source: Mark Hudsen

5

There's Plenty of Room at the Bottom…?

- More transistors doesn't always mean more speed or increased performance!
- Designs are approaching the physical limitations of processes

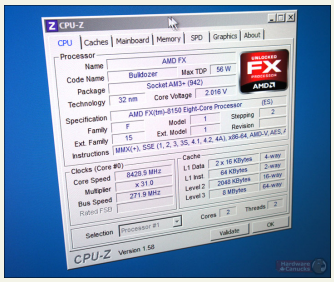


http://www.gotw.ca/publications/concurrency-ddj.htm

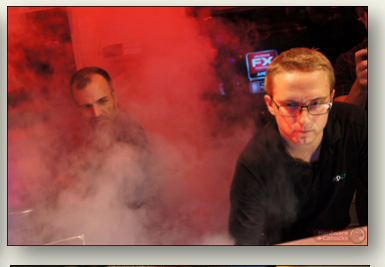
6

Harder to Increase frequency

- Heat
- Power consumption
- Leakage problems
- CPU frequency world record: Approaching **9 GHz**
  http://valid.canardpc.com/records.php



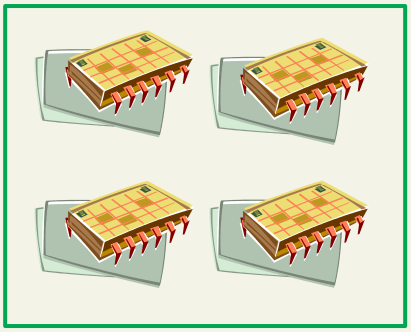
Harder to Increase Frequency cont'd

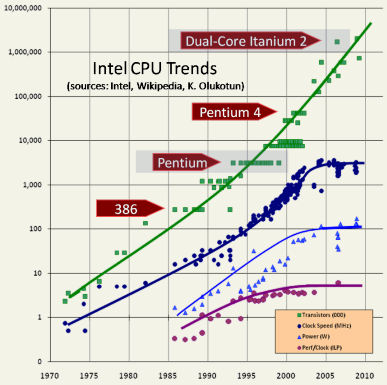


http://www.hardwarecanucks.com/forum/hardware-canucks-reviews/46413-amd-achieves-new-world-record-cpu-frequency.html

An intelligent solution

- Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.

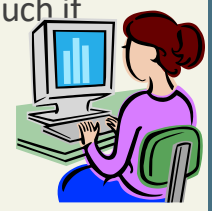




9

Now it's up to the programmers

- Adding more processors doesn't help much if programs aren't aware of them…
  - … or don't know how to use them.
- Serial programs don't benefit from this approach (in most cases).
- Running multiple instances of a serial program often isn't very useful.
  - Think of running multiple instances of your favorite game.
  - What you really want is for it to run faster!



10

Approaches to the Serial Problem

Rewrite serial programs so that they're parallel.

Write translation programs that automatically convert serial programs into parallel programs.

This is very difficult to do. Success has been limited.

An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelization of each of its steps. Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm.

11

Serial solution

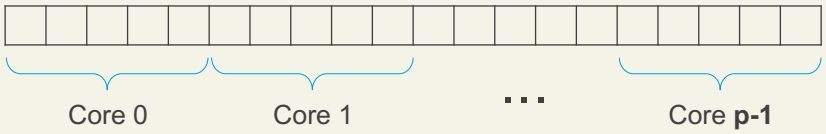
- Compute **n** values and add them together.
- Serial solution:

```
sum = 0;
for ( i = 0; i < n; i++ ) {
       sum += compute_next_value(…);
}
```


12

Concurrent Solution

- Assume we have p cores, p much smaller than n.
- Each core can perform a partial sum of approximately n/p values.




```
my_sum = 0;
my_a = ...;
my_b = ...;
for ( my_i = my_a; my_i < my_b; my_i++ ) {
      my_sum += compute_next_value(...);
}
```

Each core uses it's own ***private variables*** and executes this block of code independently of the other cores.

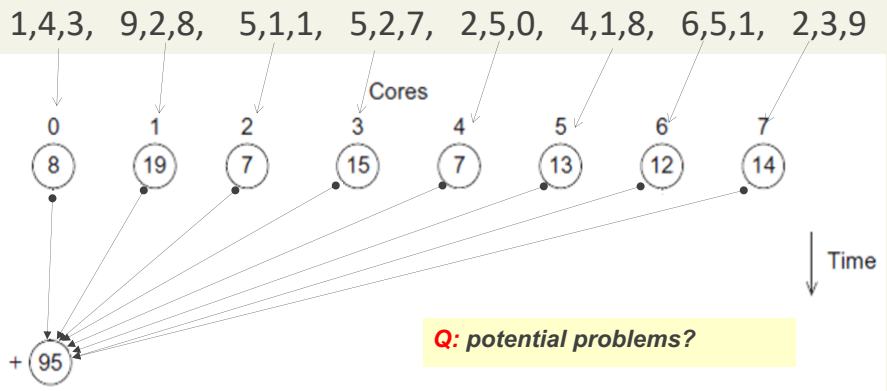
Let's Try Some Numbers…

- After each core completes execution of the code, a private variable my_sum contains the sum of the values computed by its calls to **compute_next_value(…)**
- Ex., 8 cores, n = 24, then the calls to **compute_next_value** return:
  - 1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9
- And the values stored in my_sum are

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Concurrent Solution (cont.)

- Once all the cores are done computing their private **my_sum**, they form a global sum by sending results to a designated "**master**" core which adds the final result.



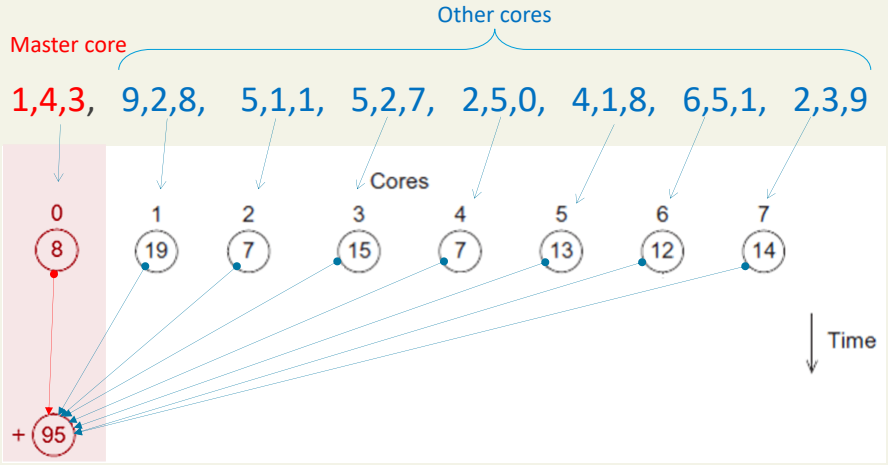
Concurrent Solution (cont.)

```
if(I'm the master core){
        global_sum = my_sum;
        for each core other than myself{
                receive value from core;
                global_sum += value;
        }
}else{
        send my_sum to the master core;
}
```

Is this the best we can do?

Here is one of the problems…

- Only the master will combine the results while other cores are idle. Can we improve the algorithm?



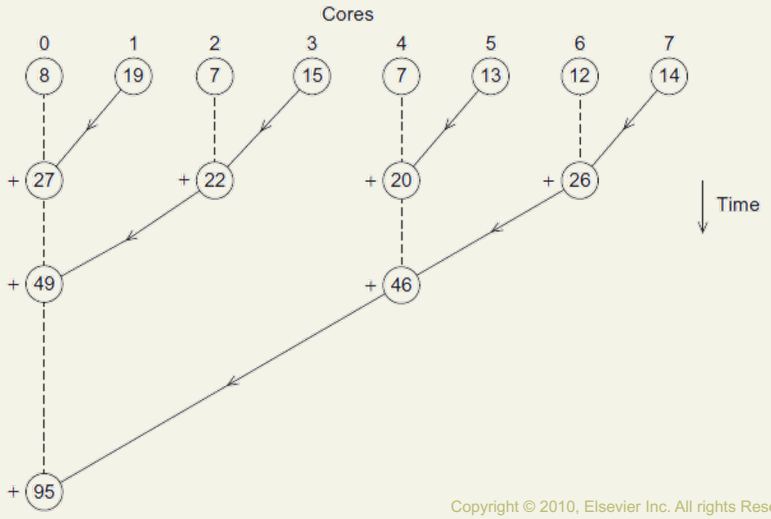
17

Better Parallel Algorithm

- There's a better way to compute the global sum, especially with large number of cores.
- Don't make the master core do all the work. Share it among the other cores!!
  - Pair the cores so that
    - core 0 adds its result with core 1's result.
    - Core 2 adds its result with core 3's result, etc.
  - Repeat the process now with only the evenly ranked cores.
    - Core 0 adds result from core 2.
    - Core 4 adds the result from core 6, etc.
  - Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.

18

Multiple Cores Forming a Global Sum



Analysis

- In the first example, the master core performs 7 receives and 7 additions.
- In the second example, the master core performs 3 receives and 3 additions.
  - The improvement is more than a **factor** of 2
- The difference is more dramatic with a larger number of cores.
  - If we have 1000 cores:
    - The first example would require the master to perform 999 receives and 999 additions.
    - The second example would only require 10 receives and 10 additions.
    - That's an improvement of almost a **factor** of 100

How Do We Write Parallel Programs?

- **Basic Idea:**
  - **Partition the work to be done among the cores!**
- **Two approaches:**
  1. **Task parallelism**
     - Partition various tasks carried out solving the problem among the cores.
  2. **Data parallelism**
     - Partition the data used in solving the problem among the cores.
     - Each core carries out similar operations on it's part of the data.

21

How Do We Write Parallel Programs?

- The professor example
  - 15 questions
  - 300 exams
  - 3 TAs

22

Division of Work: Data Parallelism


TA#1


100 exams


100 exams


TA#3


TA#2


100 exams

23

Division of work: Task Parallelism


TA#1


Questions 1 - 5
All exams



TA#3
Questions 11 - 15
All exams


Questions 6 - 10
All exams


TA#2

Issues?
Communication, load balancing, & sync issues

All TAs will be totaling the marks

24

Coordination

- Cores usually need to coordinate their work
- **Communication** – one or more cores send their current partial sums to another core
- **Load balancing** – share the work evenly among the cores so that one is not heavily loaded.
  - If one core has to compute most of the values, then the other cores will finish much sooner and their computational power will be wasted.
- **Synchronization** – because each core works at its own pace, make sure cores do not get too far ahead of the rest (e.g., if the tasks of one core **depend** on the results from another core).

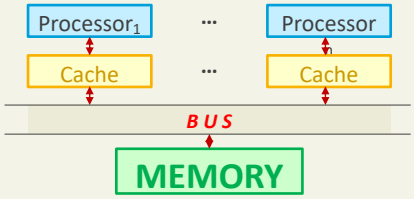
25

A few more concepts…

- Computer architectures classified by memory organizations:

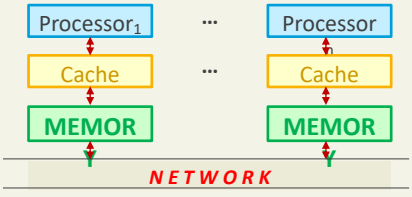
Multiprocessor with Shared Memory

Cores share same memory.
- Multiple cores on a single chip
  - e.g. CPU
- Multiple cores on separate chips
  - e.g. multiple processors, GPUs



Multi-node with Distributed Memory

Cores don't share memory
But they are connected on a network



26

Flynn's Taxonomy

Not this Flynn….

Retrieved from: https://disney.fandom.com/wiki/Kevin_Flynn

Computer architectures may also be classified into four groups:

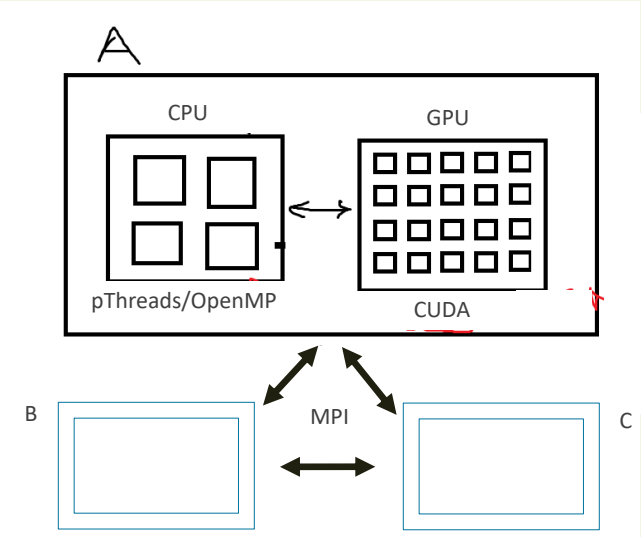
- Single Instruction Single Data (**SISD**)
  - 1 core (Serial architecture)
    - 1 instruction at any time, operations on single data stream
- Single Instruction Multiple Data (**SIMD -> GPU**)
  - Multiple cores (Parallel architecture)
    - All cores execute the same instruction at any time
    - Each core operates on a different portion of the data
- Multiple Instruction Single Data (**MISD**)
  - Multiple cores, each operate on the same data stream
  - *Uncommon architecture*
- Multiple Instruction Multiple Data (**MIMD**)
  - Multiple cores operate on multiple data streams, each core runs independent instructions

More about Flynn's taxonomy later…..

Recap: What We'll Be Doing in this Course…

- Learning to write programs that are explicitly parallel (or at least contain some parallel aspects…..)
  - Different types of parallelism
- Using different extensions to **C**
  - Posix Threads - pThreads (shared memory – low level)
  - OpenMP (shared memory)
  - CUDA (shared memory)
  - Message-Passing Concurrency (distributed memory)
- We may also have a look at **Java** Concurrency Mechanisms
  - Time permitting

Where Things Sit



29

Homework

- Read: Mark Handley, "An introduction to C Programming for Java Programmer",
  - Link on Canvas.
- read the course notes on C

30