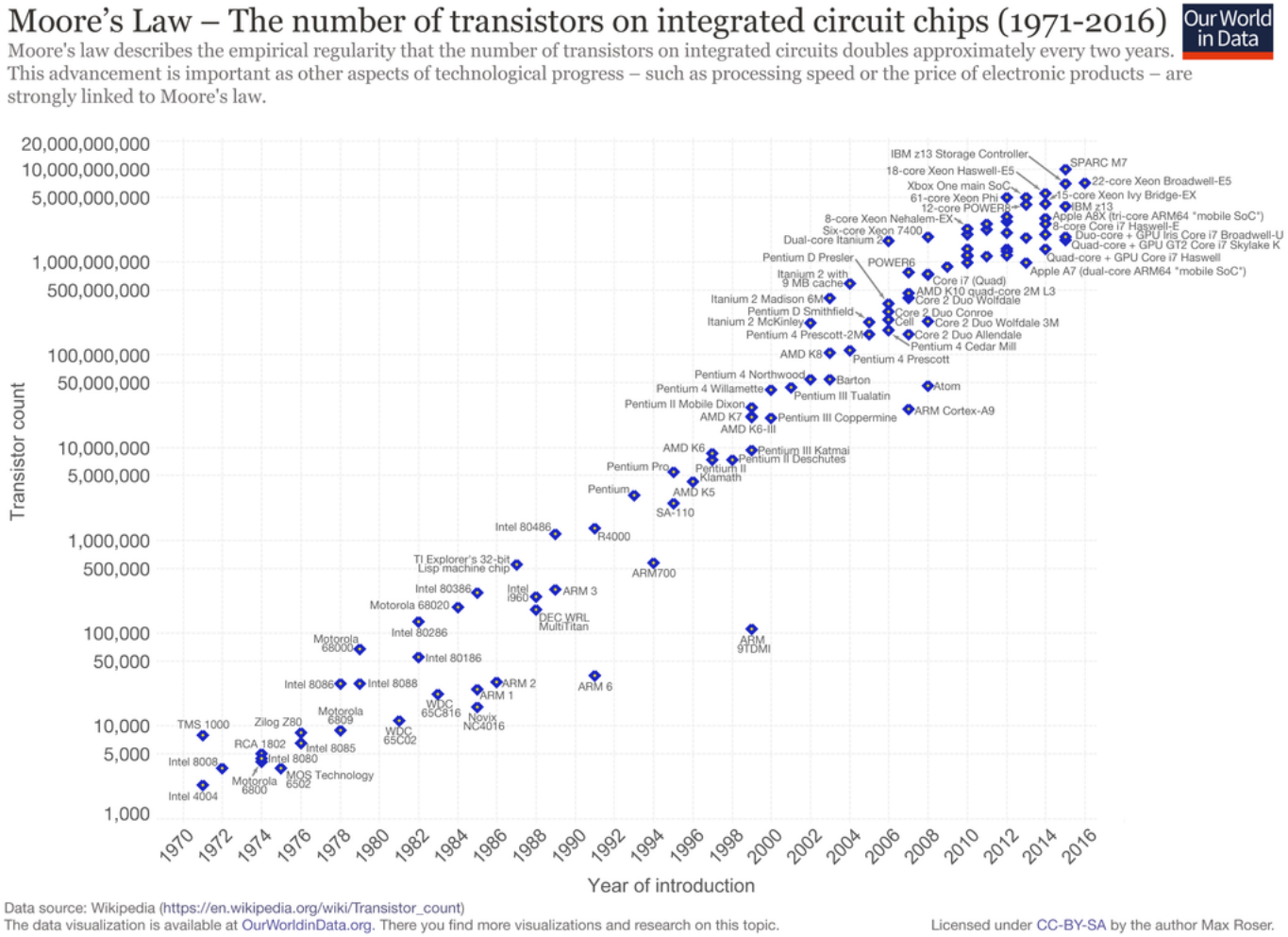
13B—Parallel Computing

How can we speed things up when processors can’t go any faster?

Parallel Computing

- Understanding the parallelization of algorithms is important to improving the speed of solutions.
  - It's yet another example of reduction
  - Indeed, ever heard of map-reduce?
- We will be doing some exercises in this area and, in the term project, you will have the opportunity to put such techniques to practical use.



ourworldindata.org [CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)]

The end of Moore's Law

- Gordon Moore (founder of Fairchild Semiconductor and CEO of Intel) knew something about integrated circuits.
- Moore's "law" predicted (1965) that the number of transistors on an integrated circuit would double every year.
- Moore's "law" was revised in (1975) such that the number of transistors on an integrated circuit would double every *two* years.
- It's held up remarkably well until recently. What got in the way?
- *c*: the fundamental scale factor of space-time (popularly known as the speed of light)—299,792,458 m/sec. (3×10^8 m)
- In one nano-second, how far can an electric field propagate? 300×10^6 nm. How far is that? It's about a foot (300 mm).
- So, in one clock-cycle of this processor, it's only 115mm (4.25 inches).

Amdahl's Law

- Gene Amdahl (IBM and founder of Amdahl Corporation) knew something about super-computing (as anything involving more than one processor was called back in 1967).

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

- where *p* is the proportion of the original problem that benefits from increased resources (i.e. more processors);
- and where *s* is the factor by which that proportion is improved;
- and where $S_{latency}(s)$ is the overall speedup of the task.

Gustafson's Law

- The problem with Amdahl's law is that it is unduly pessimistic in that it assumes that there's only one task to be worked on. Gustafson's law is a little different:

$$S_{\text{latency}}(s) = 1 - p + sp,$$

- where the variables are the same as for Amdahl's law. Now, we can see that, as *s* increases to be much greater than one, the overall speed-up $S_{latency}(s)$ is close to linear with *s*.
- This is the law that more properly governs cluster computing.

How does Java allow us to do parallel processing?

- A long time ago, there was only one processor in a computer and therefore only one program counter (the register that points to the current instruction).
- “Interrupts” were introduced to allow peripherals to take control of the processor for a necessary high-priority task (such as reading the holes in a punched card, or reading a block from disk).
- The unix operating system introduced the concept of a process– owned by a particular user so that while that user’s process was blocked waiting for input, for example, another user’s process could run.

How does Java allow us to do parallel processing? (2)

- But then, things got complicated. More and more peripherals were added (mice, displays, keyboards, multiple disks, etc.). Networks were introduced. Instead of just a few peripherals, computers needed to be able to deal with many other devices, many of which were other computers (i.e. fast).
- The idea of sub-processes (threads in Java) arose. Now, one user could potentially continue processing even though one of his sub-processes was blocked.
- In the beginning, these threads all shared the same actual cpu (core). But, as soon as multiple cores were introduced, of course the threads could be assigned to their own independent core.

How does Java allow us to do parallel processing? (3)

- This is what Java (low-level) allows us to do:
  - Create a thread (allocated from a thread “pool”). Depending on the number of existing threads and the number of cores, this may or may not run on a different core.
  - Tell the thread what to do (i.e. give it a *Runnable* program to run).
  - Start the thread.
  - Wait for the thread to finish and get the result of the *Runnable.*
  - Release the thread (the wait and release part is called a “join” operation).
- Mutex and Synchronization:
  - Additionally, in Java—because most variables and fields are mutable–we need the concept of a “mutex” to lock a variable or field (or an entire object if we like), thus reserving it as the sole property of our thread. We use the *synchronize* keyword for this.
  - Without this feature, an algorithm could encounter inconsistent state if another thread interrupts it and changes the state.

How does Java allow us to do parallel processing? (4)

- But this low-level interface is hard to use.
- ”Functional” Java uses a much higher-level paradigm including some new concepts such as:
  - *actors*,
  - immutable state where possible,
  - functional composition (available from Java8 onwards).
- See later module for more details on this.