History

Jurassic Period (-1945)

- Astronomical/Geometric tools
- Schickard, Leibniz, Pascal (17th Cen.)
- Babbage: Analytic engine (1842) – first “programmable" machine.
- Scheutz (1853)
- Hollerith (1911) – IBM (1924)
- Mechanic Tools (Slide Rule, Marchand Calculator)
- Parallel Processing in room

Early Electronics (1937-53)

Often electromechanical

- Atanasoff (1937): PDE machine
- Zuse (1938): Z1 (binary system, high level language)
- Turing (1943): Colossus
- Eckert,Mauchly (1943): ENIAC
- Eckert,Mauchly,von Neumann : EDVAC (program memory)

Second Generation (1954-62)

- First commercial machines
- Magnetic cores
- Register
- Languages: FORTRAN (1956), ALGOL (1958), COBOL (1959)

Third Generation (1963-72)

Transition to semiconductor technology

- Cray (1964): CDC6600 – parallelism, 1MFlops
- IBM 360
- Illiac
- early UNIX

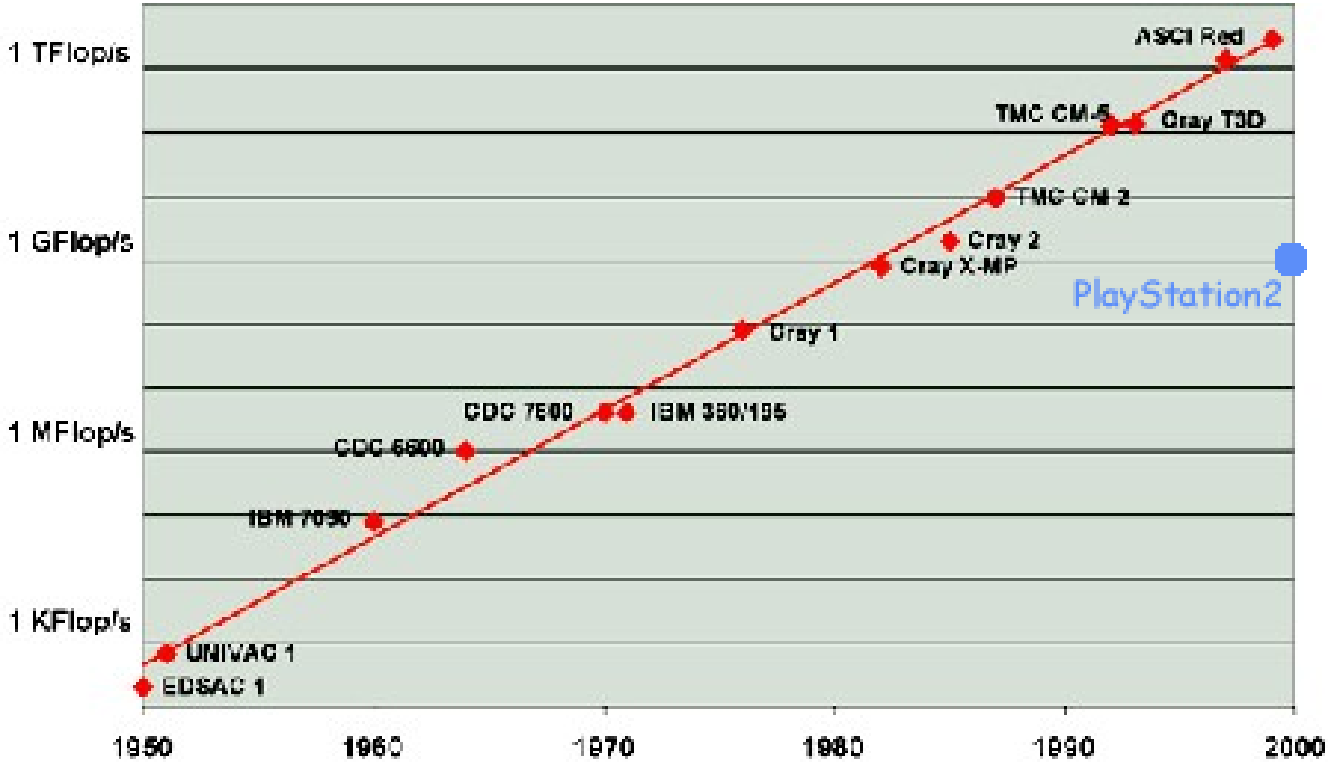
Fourth Generation (1973-84)

- C language, Unix
- Vector Processors
- Functional Programming, Prolog
- Rise of PC
- Lax report (1982): National supercomputer centers

Fifth Generation (1985-9x)

- Massive parallel machines
- Distributed memory architecture
- Workstations
- RISC technology

Development

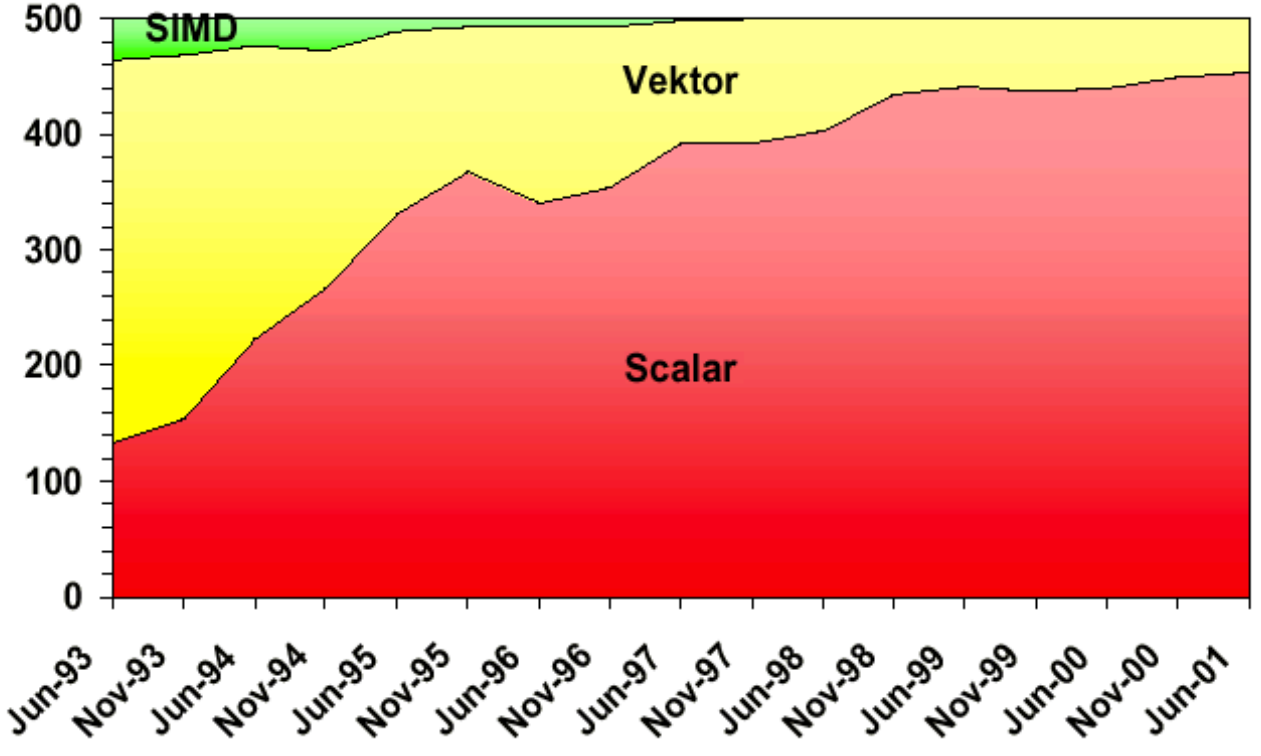


Source: Meuer/Strohmeier: Scientific Computing World

Now

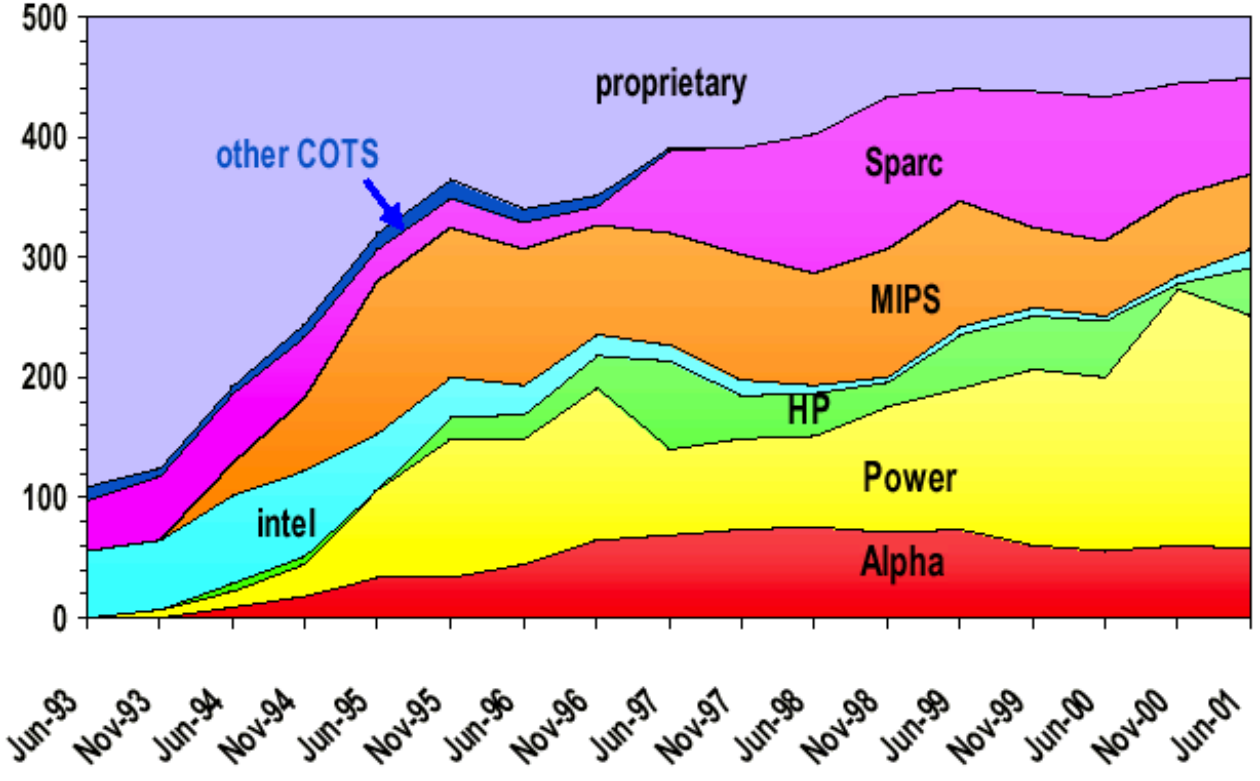
- Push for higher performance (ASCI)
- Web, Libraries, Linux
- Commodification
- Cluster (Network Of Workstations)

Processor Type



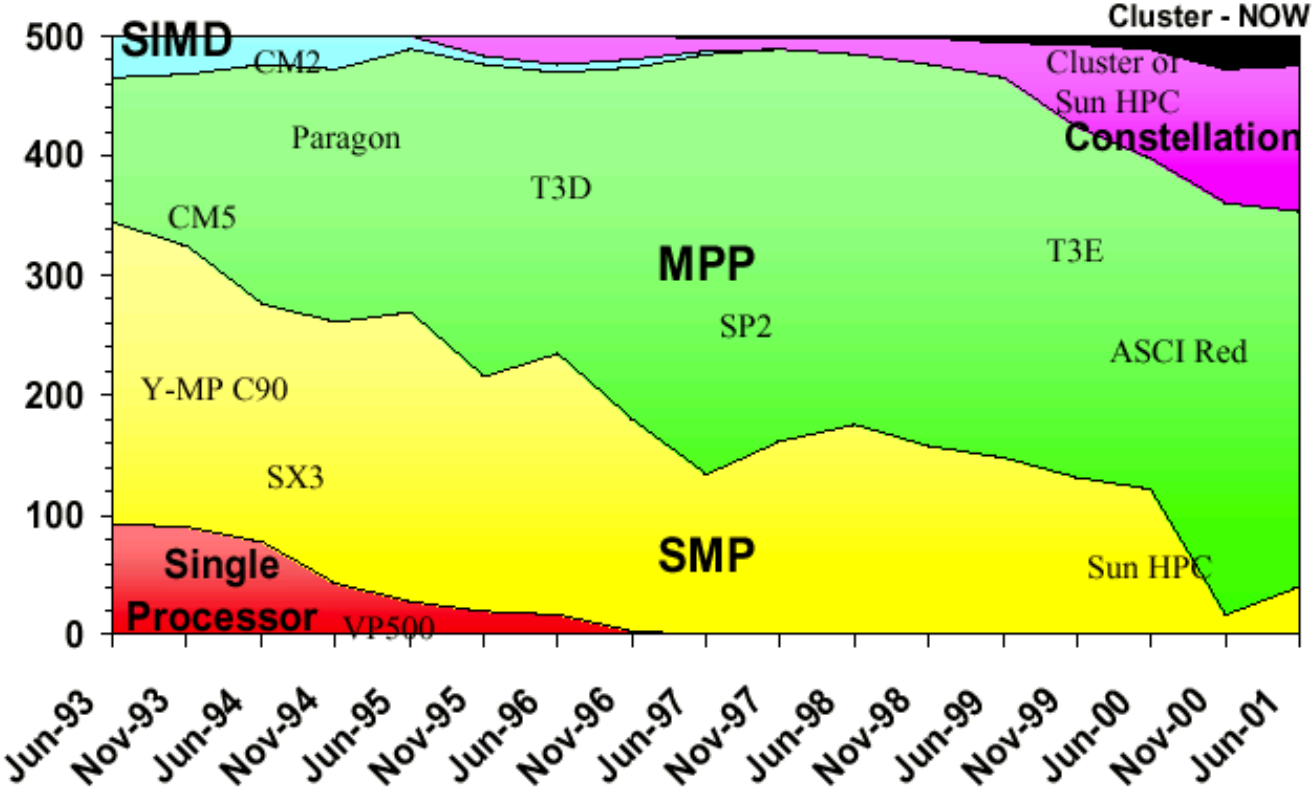
Source: www.top500.org

Chip Technology



Source: www.top500.org

Architecture



Source: `www.top500.org`

Parallelization

There are various (obvious) obstacles to have a single processor run faster.

If single-processor performance does not suffice, the only way therefore is to use several processors at once, that is to parallelize the calculation.

- One person takes a workday to carry 5000 bricks from A to B.
- 8 persons take one hour.
- 5000 persons take a minute.
- 300000 persons take 1 second ?
- A flight from Denver to London takes 10 hours. How long does it take using 2 planes in parallel?

Embarrasing parallelization

If I have to solve two different problems, I can solve them in parallel on two computers.

Some calculations can be parallelized in almost the same way: There are (comparatively) long calculations that do not interact, the results are collected and used afterwards.

Such a parallelization is easy and gives almost linear speedup.

Problems arise if different processes have to communicate or have to share resources.

Example

Sit in a square arrangement. Take a piece of paper and write on it a (random) positive integer between 1 and 100.

Now we perform the following process:

- Take the numbers of your (horizontal, vertical) neighbours (assume cyclic arrangement),
- average them (round to the nearest integer), and
- replace your number with this number.
- Iterate

What is the result?

Problems

- Communication
- When is new data available?
- Idling.
- When is the process done/ What to do with result?

Obstacle: Dependency

If task A depends on the result of task B, they cannot run in parallel.

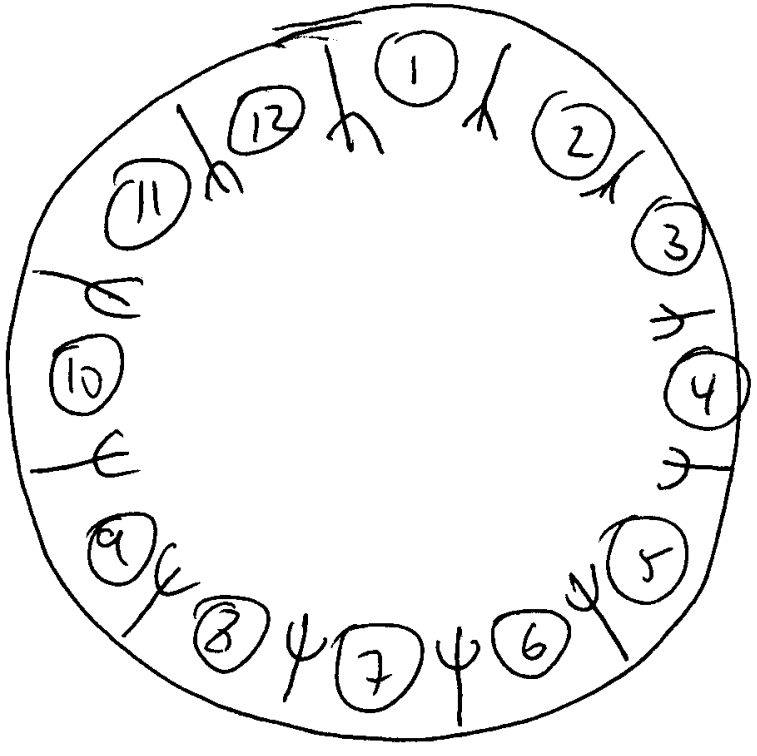
(There are fancy methods in project managment (PERT, C/PM) that can be used to identify such situations.)

For such tasks, parallelization gives no speedup.

Similarly it could happen that one process is idle while waiting for other processes. Proper load balancing tries to resolve this.

Obstacle: Deadlock

12 philosophers sit on a table, eating long spaghetti.



One needs 2 forks to eat, there is one fork between 2 plates. If 1 is eating, 2 cannot.

Suppose 1 and 3 are eating alternatively, 2 is permanently blocked.

If everyone takes up his left fork, the system is deadlocked. (The same can happen if different processes require the same resource.)

Remedy:

Coordinated Use signals (semaphores) to indicate the desire to eat. Then dispatch fork selection according to semaphore queue.

Uncoordinated If you are blocked, wait a random amount of time (based on maximal task time) and try again. (Used in ethernet protocol.)

Typically this is resolved on a low level in hardware or the communication protocol.

Obstacle: Communication

If two processes exchange data, this data has to be transferred from the processor executing process A to the processor executing process B:

$$A \longrightarrow B$$

If the processors have a direct communication link, this is easy.

If not, the data has to be transmitted via intermediate processors:

$$A \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \cdots \longrightarrow C_3 \longrightarrow B$$

This takes processing power away from the processors C_i. In the worst case, the system ends up spending almost all time in communication.

One way to remedy it is to give many different communication links (Hypercube architecture). However fast links are expensive.

Fortunately many “real world" calculations (for example weather forecast) require on the lowest level only communication between “adjacent cells". This limits communication requirements.

Common Network Another way to give total communication is to connect all processes to a common network. The problem then is potential deadlock.

Again, one can spend almost unlimited amounts of money to get “better" network connection (for example: fast switches).

Obstacle: Bootstrap/Crash Recovery

When running programs in parallel we have to start them all. Typically this has to be done indirectly.

The processes have to initialize properly and have to initiate communication.

Similarly, one has to deal with errors (or crashes) in single processes. Possible reqctions are:

- Terminate all calculations.
- Move load to other process.
- Restart dead process.

Amdahl's Law

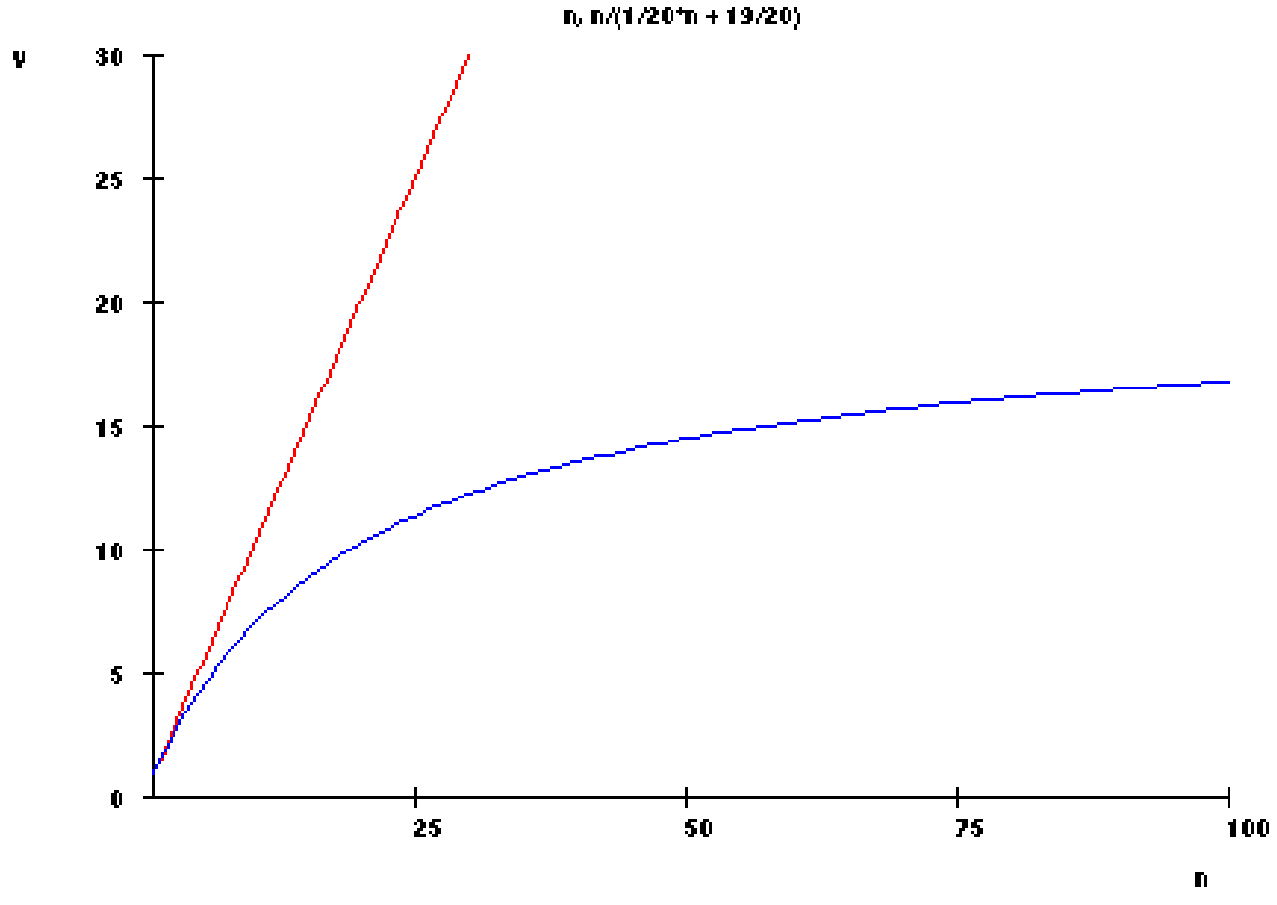
Not all parts of a program can be parallelized, there is always a serial "residue". Suppose this residue is a ratio of b of the total calculation.

Then the speedup of a program using n processors (over a single processor) is

$$S = \frac{n}{bn + (1 - b)}$$

For a constant nonzero b this yields a curve that grows substantially slower than n, in fact it has limit $\frac{1}{b}$ for $n \to \infty$.

For example for $b = 1/20$ we get:



For problems with nonparallelizable part this can substantially limit the gain due to parallelization.

Note that initialization and communication always stay nonparallelizable.