Parallel Computers and Parallel Computing

Today:

- How are parallel computers designed?
- How are parallel programs analyzed?

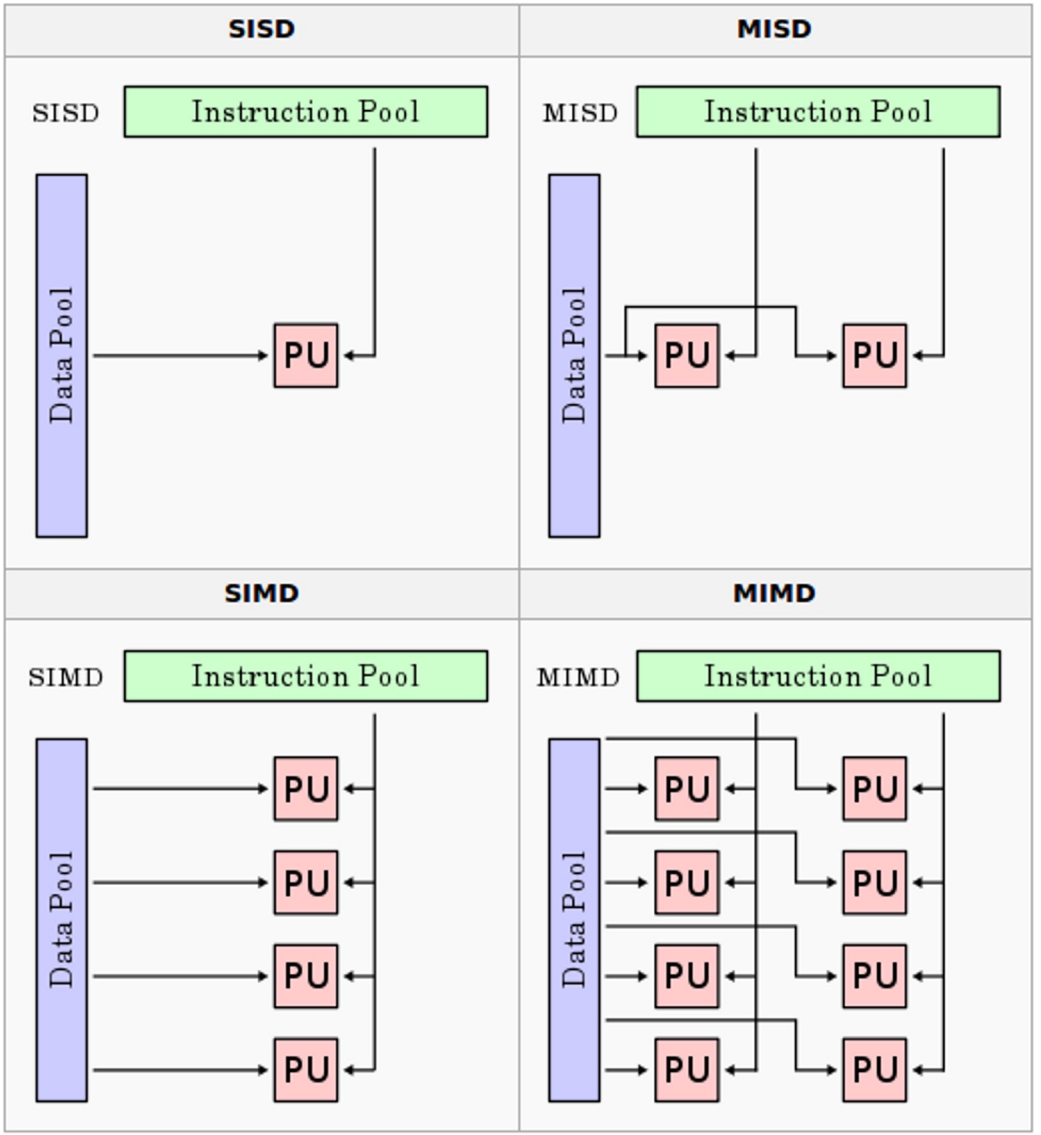
Flynn's Taxonomy [1966]

Computer processors classified by instruction and data stream:

- SISD: Single Instruction, Single Data
  - Purely sequential computer
- SIMD: Single Instruction, Multiple Data
  - PCs: Vector processing (2-32 words)
  - Graphic Processing Units (GPUs)
- MISD: Multiple Instruction, Single Data
  - Hardly ever used
- MIMD: Multiple Instruction, Multiple Data
  - Complete parallel computer (clusters, super computers, brake, ...)

Flynn's Taxonomy

PU =
processing unit



Programming

SPMD: Single Program Multiple Data

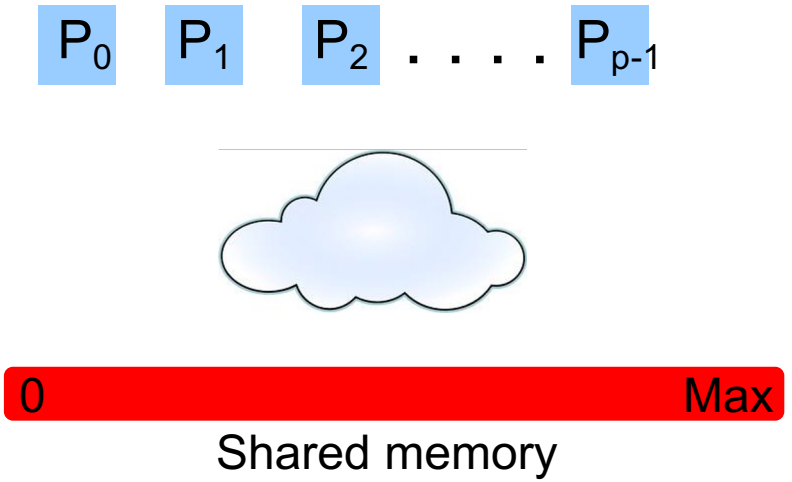
- Every process runs the same program
- Processes/threads are distinguished by process ids (0 through p-1)
- Used both for SIMD and MIMD computers
- We will only be using SPMD

```
main() {

    int size            // Number of processes
    int rank            // My process id: 0 to size-1

    if (rank == 0)
      printf("Process %d: I am the master! :-) \n",rank);
    else
      printf("Process %d: I am a worker! :-( \n",rank);
}
```

Shared Memory Computers

One common memory area

Visible to all processors



Size of system limited by interconnect

Medium size systems 1-128 processors

- But each processor can run several threads

Can have both local and shared variables.

- Local: Each thread has its own copy of the variable
- Shared: Only one variable shared by all threads

Main issue when programming: Avoiding race conditions:

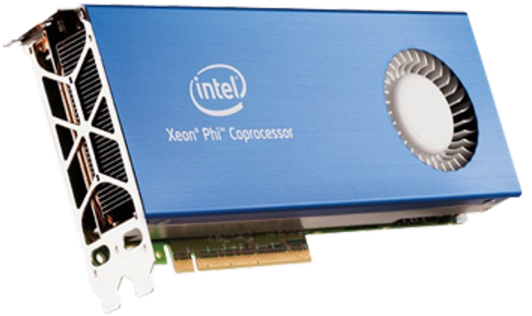
P_0: A = 5; // If A is shared, which instruction happens last?

P_1: A = 4;

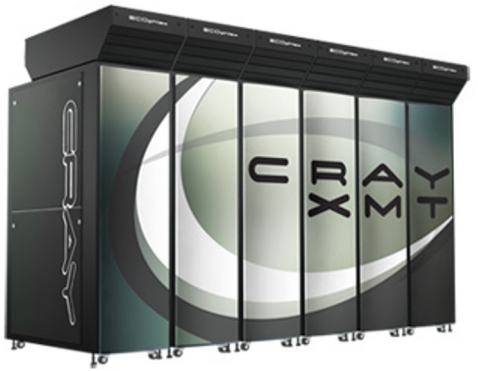
Examples of shared memory computers

Intel Xeon
Up to 60 cores

NVIDIA A100 GPU
6912 cores



Intel Xeon Phi
72 cores (discontinued)

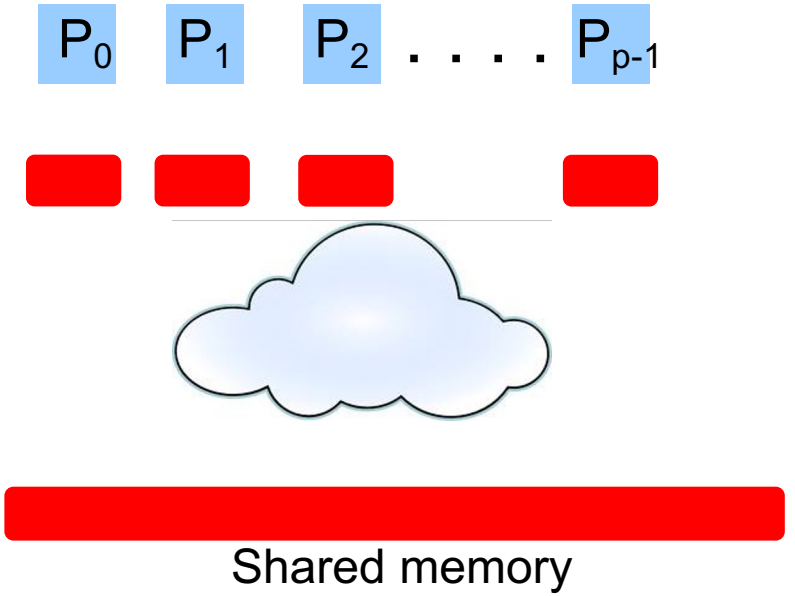


Cray XMT
Up to 1M cores
128TB memory
(discontinued)

Implementing Shared Memory Computers

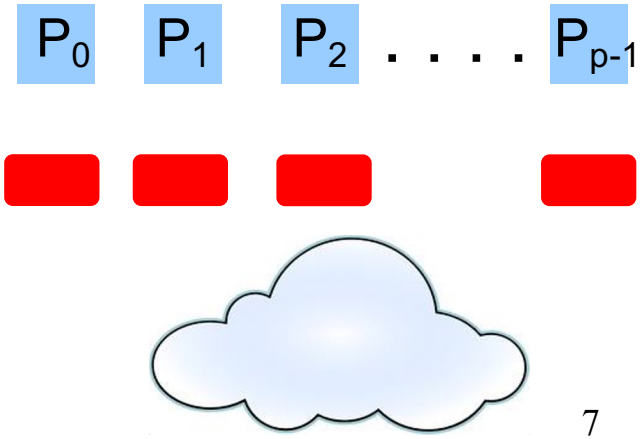
Different models of implementation:

- Can have some memory locally (caches etc.)
- Can have all memory locally
- System keeps track of where the data is

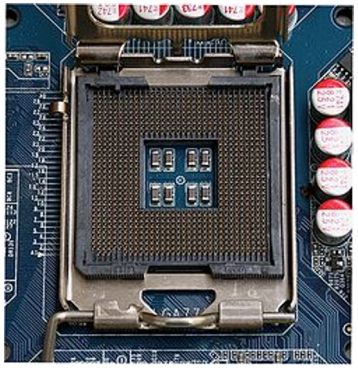


NUMA: Non-Uniform Data Access

- Takes more time to access remote memory
- Wish to map data close to thread(s) that use it



The Hardware



Socket

- Where a (multicore) processor can be plugged in

Multicore processor:

- A processor that can run several processes (i.e. threads) at the same time (one or two per core)
- Each process has access to the same shared memory
- Can have individual caches

Thread:

- Light weight process: One processor can run multiple threads

Hyper-threading:

- One processor (i.e. core) rapidly switching between multiple threads
- Good idea if one thread is waiting for data from memory

Can have multiple (multithreaded) processors sharing the same memory.

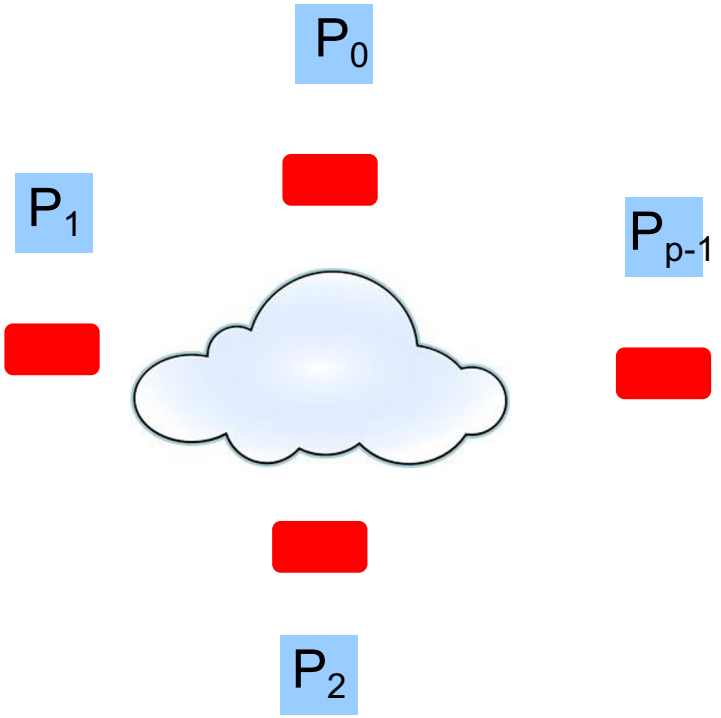
Distributed Memory Computers

- Memory is partitioned between the processors
- Each processor can only see and access its own memory

- Processes send and receive messages for accessing non-local memory

- API:
  Message Passing Interface (MPI)
  Bulk Synchronous Parallel Model (BSP)

- Offers more control and responsibility to the programmer. As a concequence: harder to program compared to shared memory computers.



+ Easier to build large systems

- Typically higher latency than shared memory systems

Examples of distributed memory computers

Hexagon
5.5K cores
6TB memory
Terminated 2017

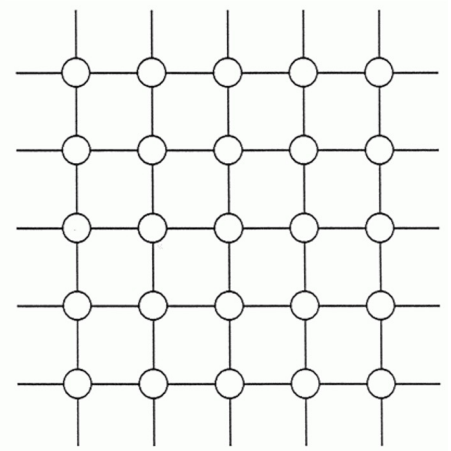
Rasberry Pi cluster

Interconnect Topologies

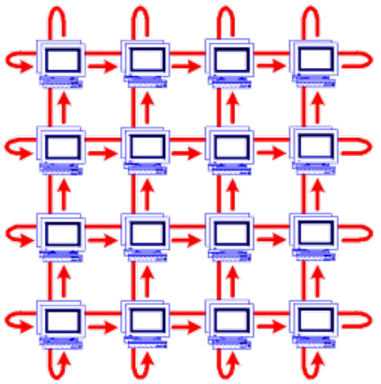
How are the processors connected?

- Maximum distance between two processors should be small (max number of hops for a message)
- Cost of building should be small (number of wires should grow "slowly" with num. of proc.)
- Able to send several messages simultaneously

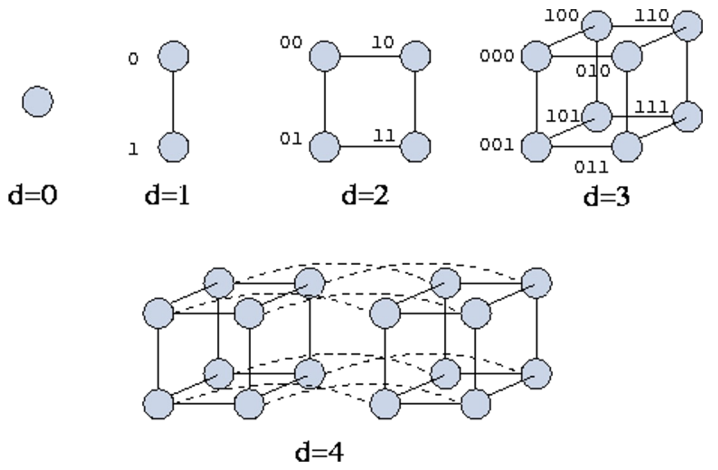
Examples: All-to-all connections, bus



Mesh



Torus



Hypercube

Hybrid Systems

Combining shared memory and message passing

- Typically, shared memory processors connected in a message passing network
- Need efficient interconnect to connect multiple systems
- Modern supercomputers are of this type

Beowulf systems

- Use "standard" PC components + local area network for communication
- Cheaper but has higher latency

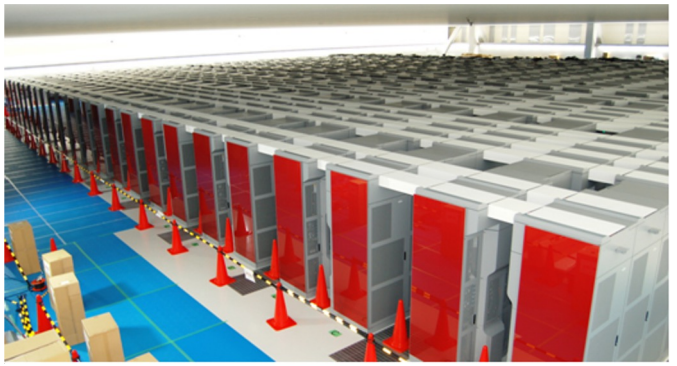
brake.ii.uib.no

"brake" = "juniper" (eng.) = "einer" (bokmål) = Computer used in this class

- Has 4 sockets
- Each socket holds an Intel 2.0GHz E7-4850 multicore processor
- Each processor has 32 GB memory (total of 128 GB)
- Each processor has 10 cores
- Each core can run 2 threads by hyper-threading
- In total 4 * 10 * 2 = 80 threads

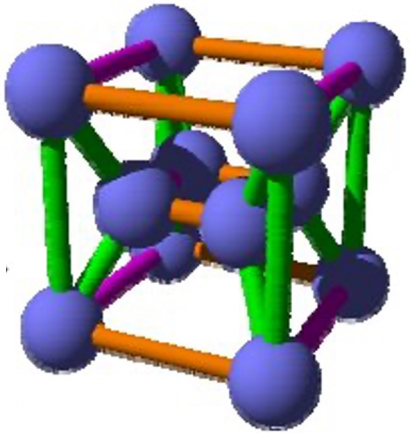
- Memory usage determines whether more than 40 threads is a good idea
- Accessing off-socket memory is slower than on-socket memory
- Capable of vector processing (i.e., pipeline) and SIMD
- Peak performance 2.0 * 40 * 4 Gflops = 320 Gflops
- Would have been the world's most powerful computer in 1996, and on the Top500 list until 2004

The K Computer



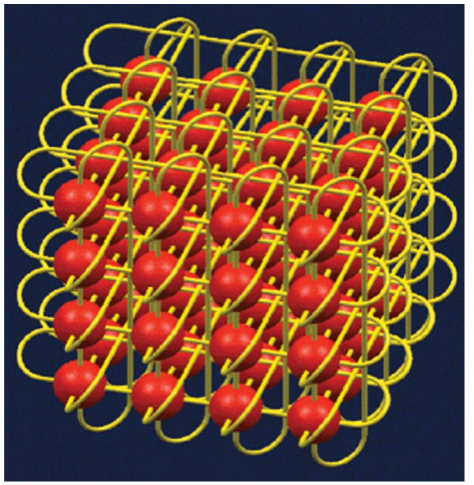
The world's most powerful computer until 2012.

Decommissioned 2019.



Used 88128 2.0GHz 8-core Fujitsu SPARC64 VIIIfx processors (in 864 cabinets), for a total of 705,024 cores

12 processors arranged in a cube-like structure (a node)



Nodes are organized in a torus network

One-to-one connection between processors in adjacent nodes

Can use shared memory on the processor level and then distributed memory between processors

Analyzing the Performance of Parallel Algorithms

Speedup

How fast is the parallel algorithm compared to the sequential one?

$\mathbf{t_s}$: Time of best sequential algorithm for a problem

$\mathbf{t_p}$: Time of parallel algorithm using p processes

Speedup: $S(p) = t_s/t_p$ Expect: $t_s/p \leq t_p \leq t_s$ (smaller t_p is better)

Thus $1 \leq S(p) \leq p$ (larger S(p) is better)

Two ways of measuring:

- Experiments: Using wall clock time
- Theoretically: Using number of operations (O- and Θ-notation)

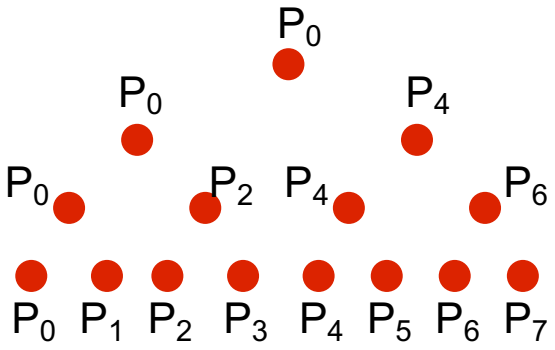
Example: Summing numbers

Input: n integers, Output: their sum

$t_s = \Theta(n)$ (n-1 additions)

Parallel algorithm:

Each process first sums n/p numbers: Yields p partial sums in $\Theta(n/p)$ time
Next, sum pairs of the partial sums recursively:



log p time steps

$t_p = \Theta(n/p + \log p)$

In theory:
$S(p) = n/(n/p + \log p)$
$= p/(1 + p/n * \log p)$

If $n \to \infty$ then $S(p) \to p$

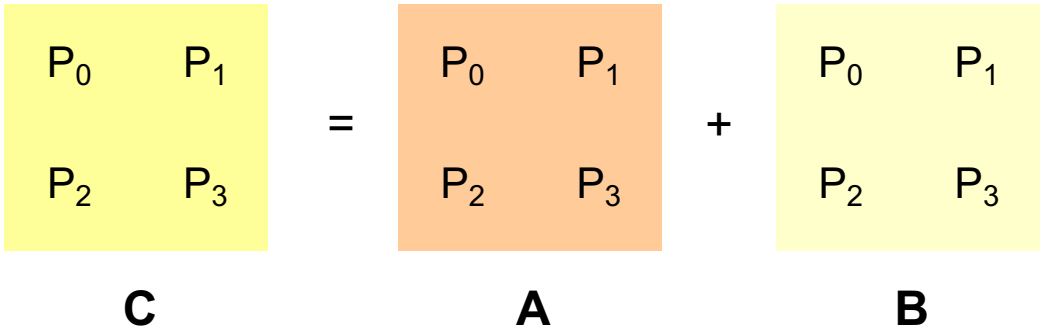
In practice:
Communication is much slower than computation.

In most cases faster to let only one process do the final summation.

Speedup in Practice

Linear speedup: $S(p) = p$

Calculating **C** = **A** + **B** where **A**, **B**, and **C** are nxn matrices



No communication needed
(carefull with the cache lines..)

Superlinear speedup: $S(p) > p$

Can happen if data does not fit into sequential cache, but parallel data does.

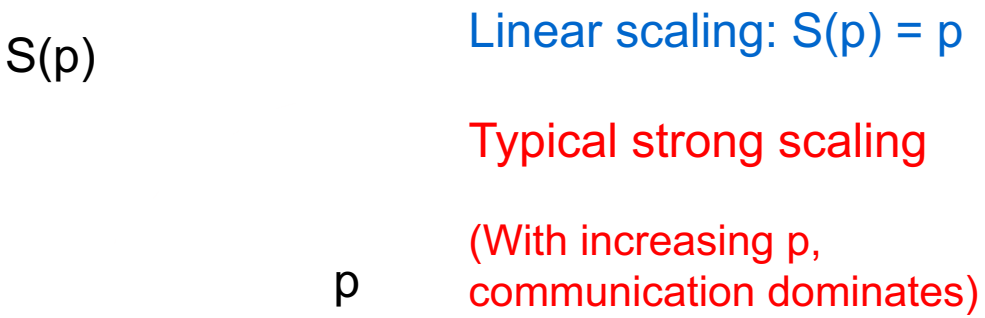
No speedup: $S(p) \leq 1$

Parallel processing makes sense only if needed for access to more memory.

Scalability

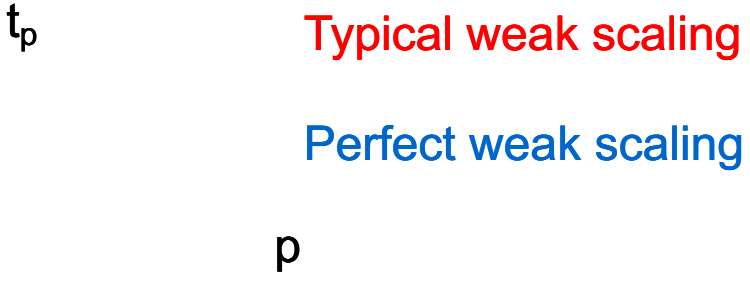
How does the program performance improve with increasing numbers of processes?

Strong scalability:
For a fixed problem, how does the speedup change when p increases?



Useful for analyzing *cpu bound* problems

Weak scalability:
Increase both number of processing elements and problem size while keeping amount of sequential work per processing element constant.



Increase resources while keeping work per processor fixed.

Note, work might grow faster than problem size.

Weak scalability for matrix multiplication

Best sequential runtime is $\Theta(n^3)$.

1. experiment — Run algorithm on $k_1 \times k_1$ matrix using 1 process.
Each process gets k_1^3 work.

2. experiment — Run algorithm using 2 processes. What should k_2 be?
Amount of work per process will be $k_2^3 / 2$.
Thus we want $k_2^3 / 2 = k_1^3$
This gives $k_2 = 2^{1/3} k_1 = 1.26\, k_1$

In general, when using p processes on problem of size k:
Amount of work per process = k^3/p
Must solve $k^3 / p = k_1^3$
Gives $k = p^{1/3} k_1$

Deviations from a straight line could mean that overhead (such as communication) is increasing with p.

t_p

p

Efficiency

How well are we using the parallel system?

Efficiency: E = "Best parallel time we could get" / "Parallel time we actually got"

= $(t_s/p) / t_p$

= $(t_s/t_p) / p$

= S(p) / p = Speedup / processes - ratio

Assuming $1 \le S(p) \le p$, we have $1/p \le E \le 1$ (best)

Sum example: $E = S(p)/p = (n / (n/p + \log p)) / p$

$= 1 / (1 + p/n * \log p)$ If $n \to \infty$ then $E \to 1$

Note that $E = t_s/(pt_p)$ = time(Sequential resources) / total time(Parallel resources)

- How well are you using what you are paying for?

Amdahl´s Law

Suppose only part of an application can be run in parallel

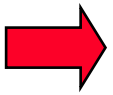
Amdahl's law

- Let s be the fraction of work done sequentially
- → parallelizable fraction is 1-s
- $t_s = 1 = s + (1 - s)$

$$S(p) = t_s / t_p$$

$$\leq 1/(s + (1-s)/p)$$

$$\leq 1/s$$ (theoretical bound approached as $p \rightarrow \infty$)



Even if the parallel part speeds up perfectly, performance is limited by the sequential part!

Overhead

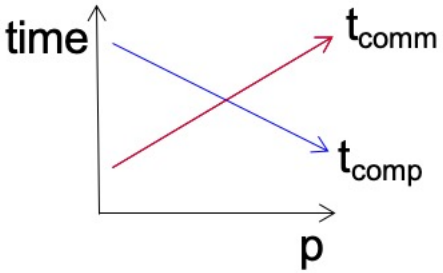
Why is S(p) < p?

- Code that cannot be parallelized (Amdahl's law)
- Uneven load
- More computations in parallel code
- Time needed for communication and syncronization

Communication overhead

$t_p = t_{comm} + t_{comp}$ (communication + computation)

Higher values of p → Data is more fine-grained → More communication
Computation-communication ratio: $r = t_{comp} / t_{comm}$



Indicator of good speedup: Fast growth in r with problem size
- Time to perform work dominates time to communicate

Sum example: When is n "large" in comparison to log p?

(constants do matter).