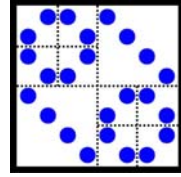


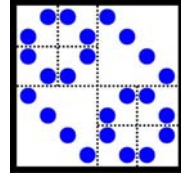
Parallel Numerics

Scope: Revise standard numerical methods considering parallel computations!

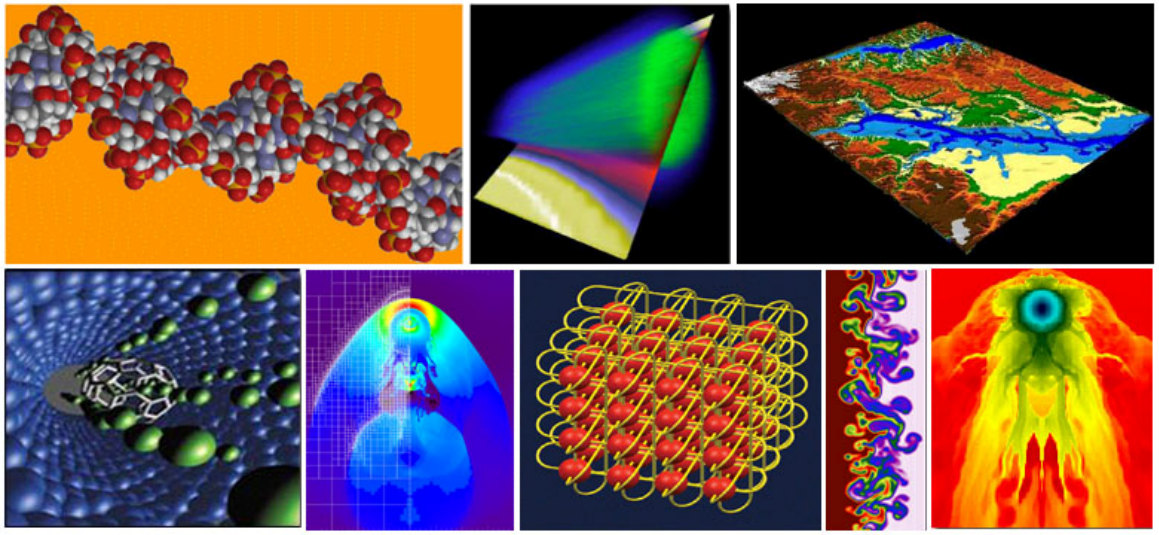
Required knowledge: Numerics
Parallel Programming
Graphs

Literature: Dongarra, Duff, Sorensen, van der Vorst: Numerical Linear Algebra for High-Performance Computers
Pacheco: A User's Guide to MPI (web)
Parallel Programming with MPI
Schüle: Paralleles Rechnen

Why parallel computing?

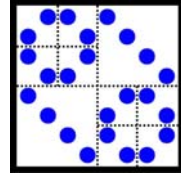


SETI, weather prediction, quantum simulation



TOP500

HLRB-II



I. Introduction
   1. Computer Science Aspects
   2. Numerical Problems
   3. Graphs

II. Elementary Linear Algebra Problems
   1. BLAS
   2. Matrix-Vector Operations
   3. Matrix-Matrix-Product

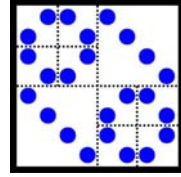
III. Linear Equations with Dense Matrices
   1. Gaussian Elimination
   2. Vectorization
   3. Parallelization
   4. QR-Decomposition with Householder matrices

IV. Sparse Matrices
   1. General Properties, Storage
   2. Sparse Matrices and Graphs
   3. Reordering
   4. Gaussian Elimination and Graphs

V. Iterative Methods for Sparse Matrices
   1. Stationary Methods
   2. Nonstationary Methods
   3. Preconditioning

VI. Domain decomposition

VII. Eigenvalues, (Quantum Computing, GPU)



1. Introduction

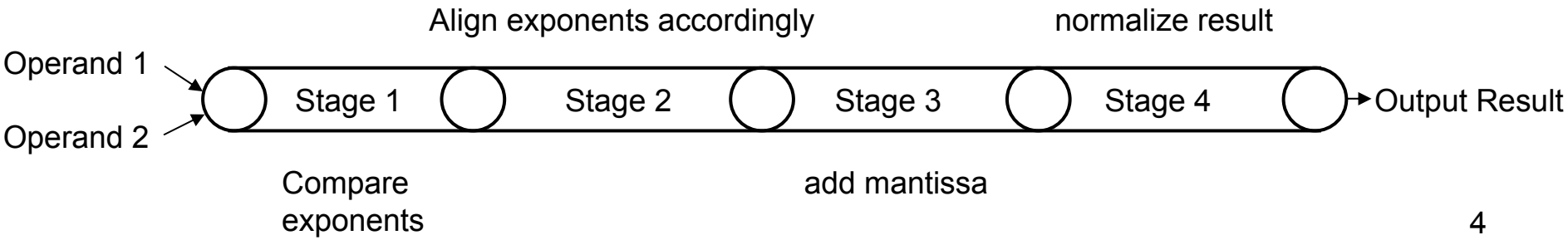
1.1 Computer Science Aspects of Parallel Numerics

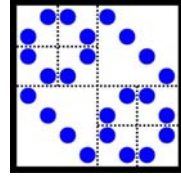
1.1.1 Parallelization in the CPU

Elementary operations in CPU are carried out in pipelines:

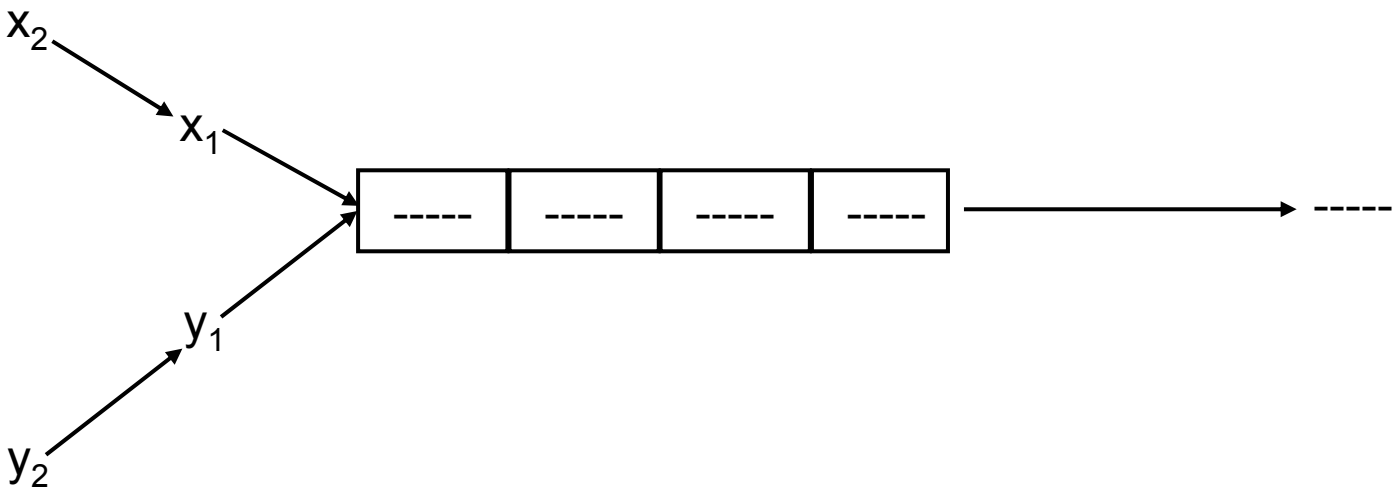
- Divide a task into smaller subtasks
- Each small subtask is executed on a piece of hardware that operates concurrently with the other stages of the pipeline.

Addition Pipeline:

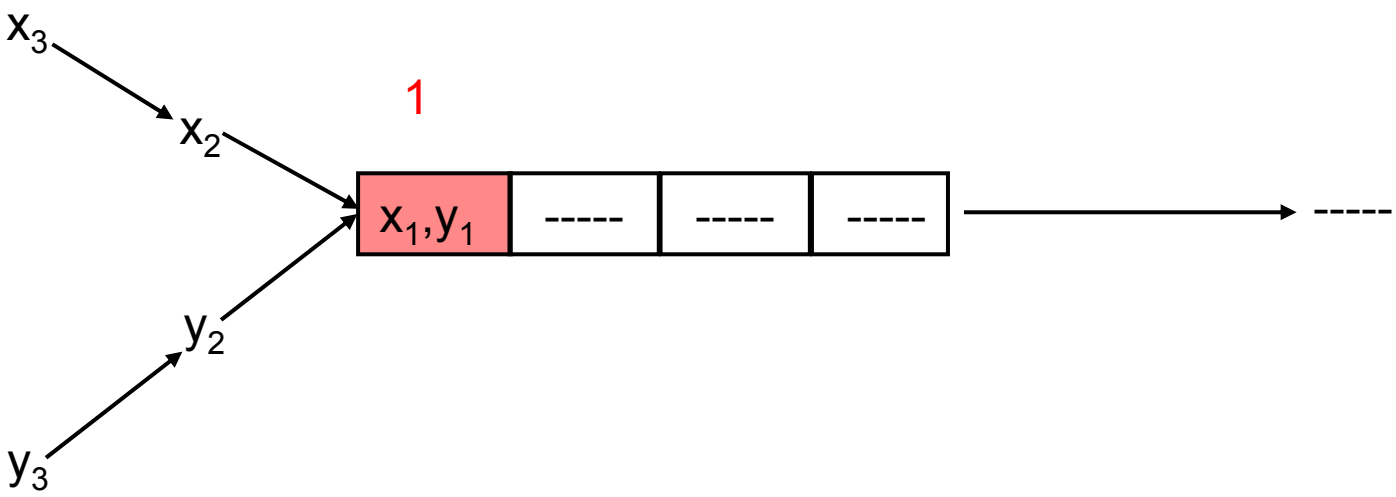


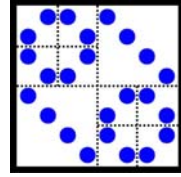


Visualisation Pipelining

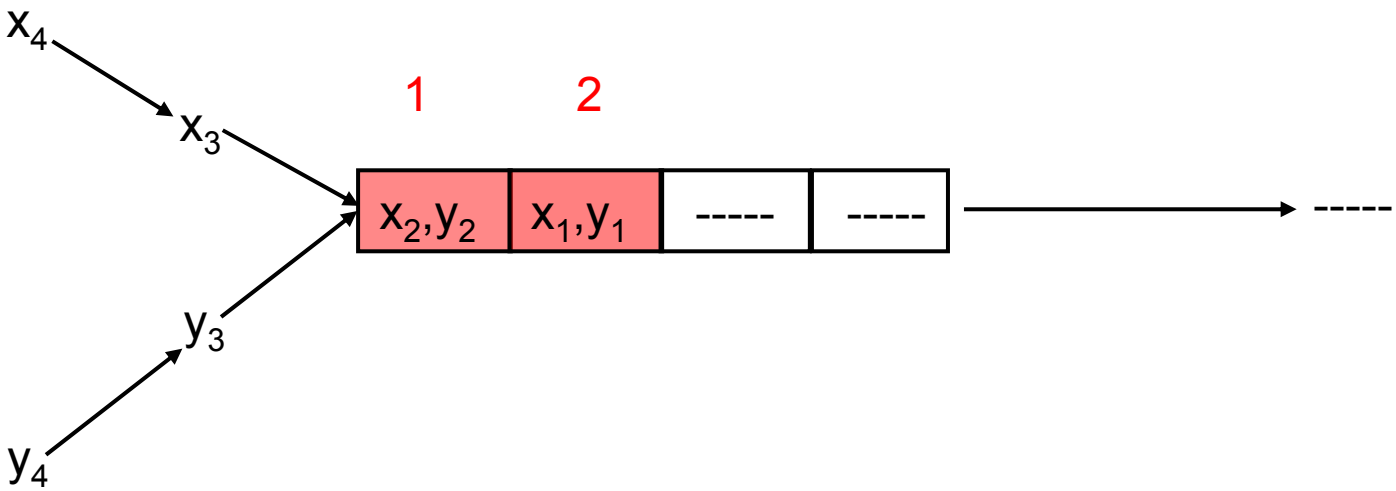


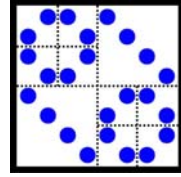
Visualisation Pipelining



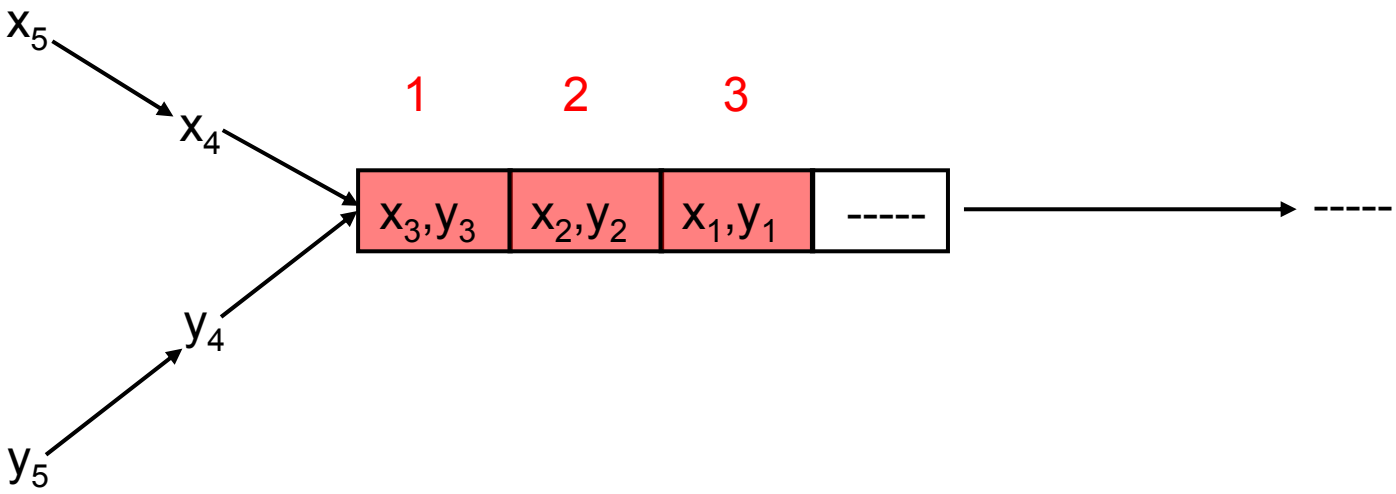


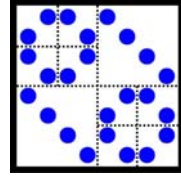
Visualisation Pipelining



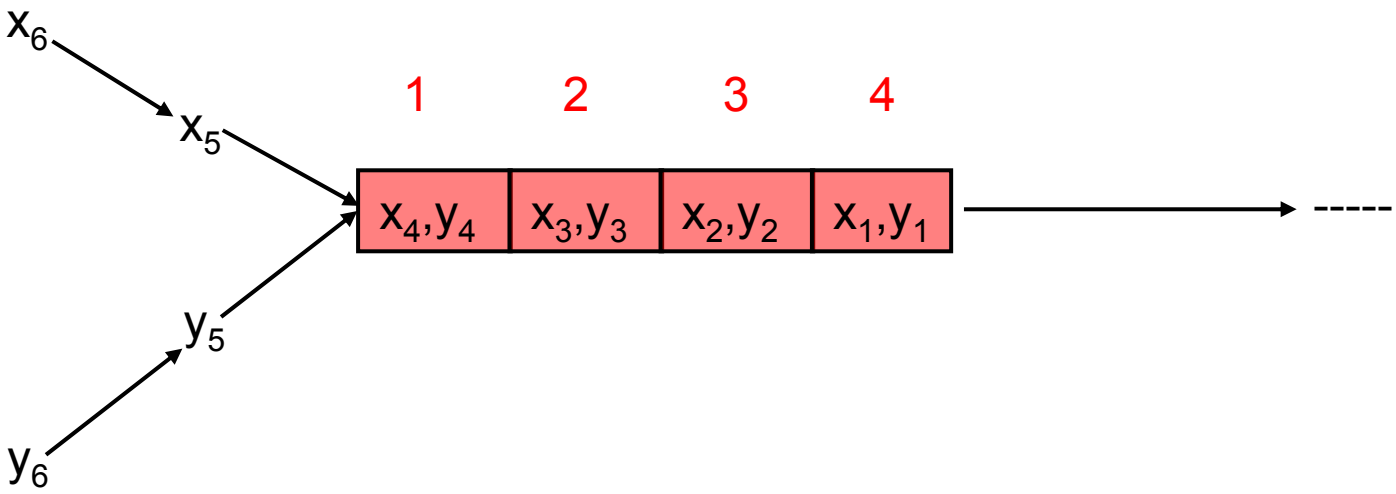


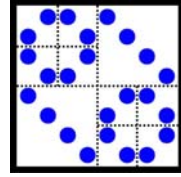
Visualisation Pipelining



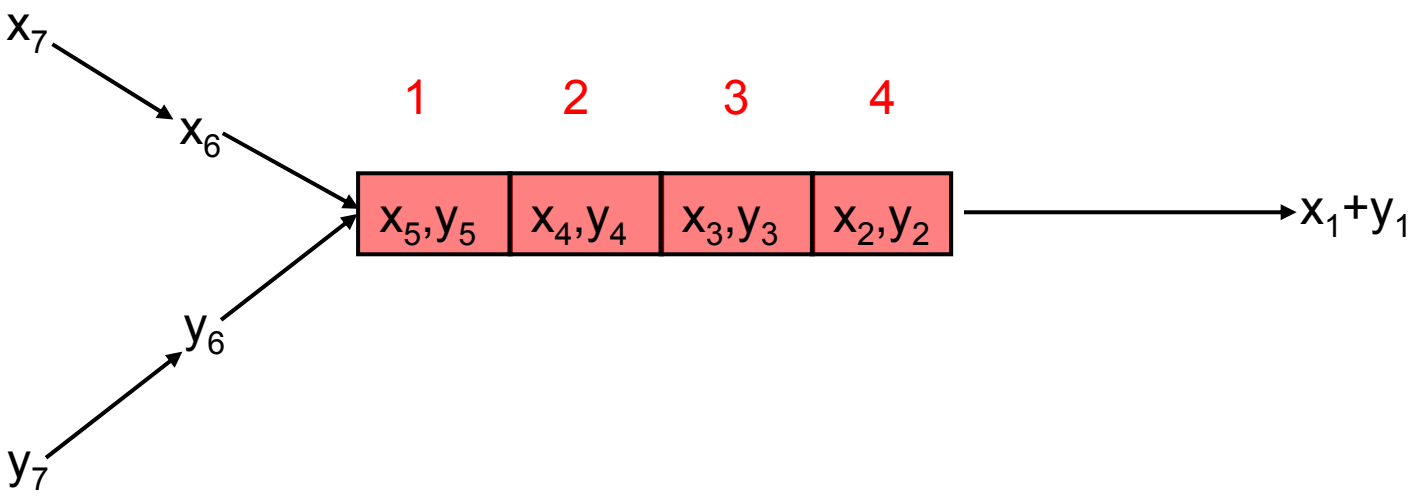


Visualisation Pipelining

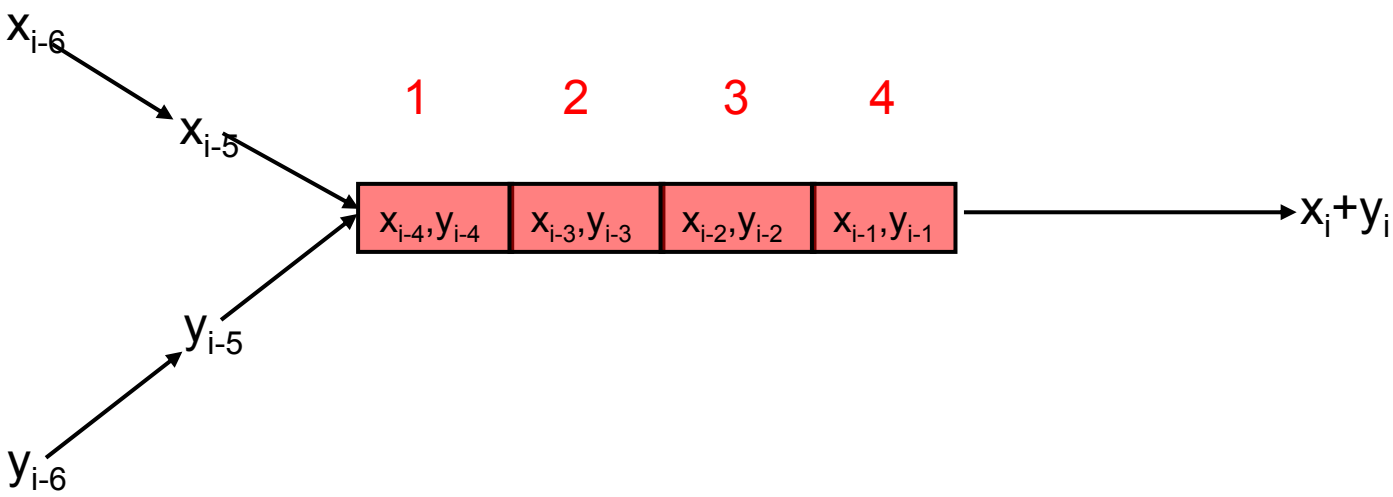




Visualisation Pipelining



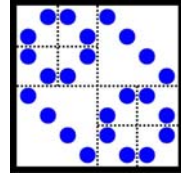
Visualisation Pipelining



Startup time = k(=4) clock units

Lateron on: per clock unit one result

Total time: k*u + n*u



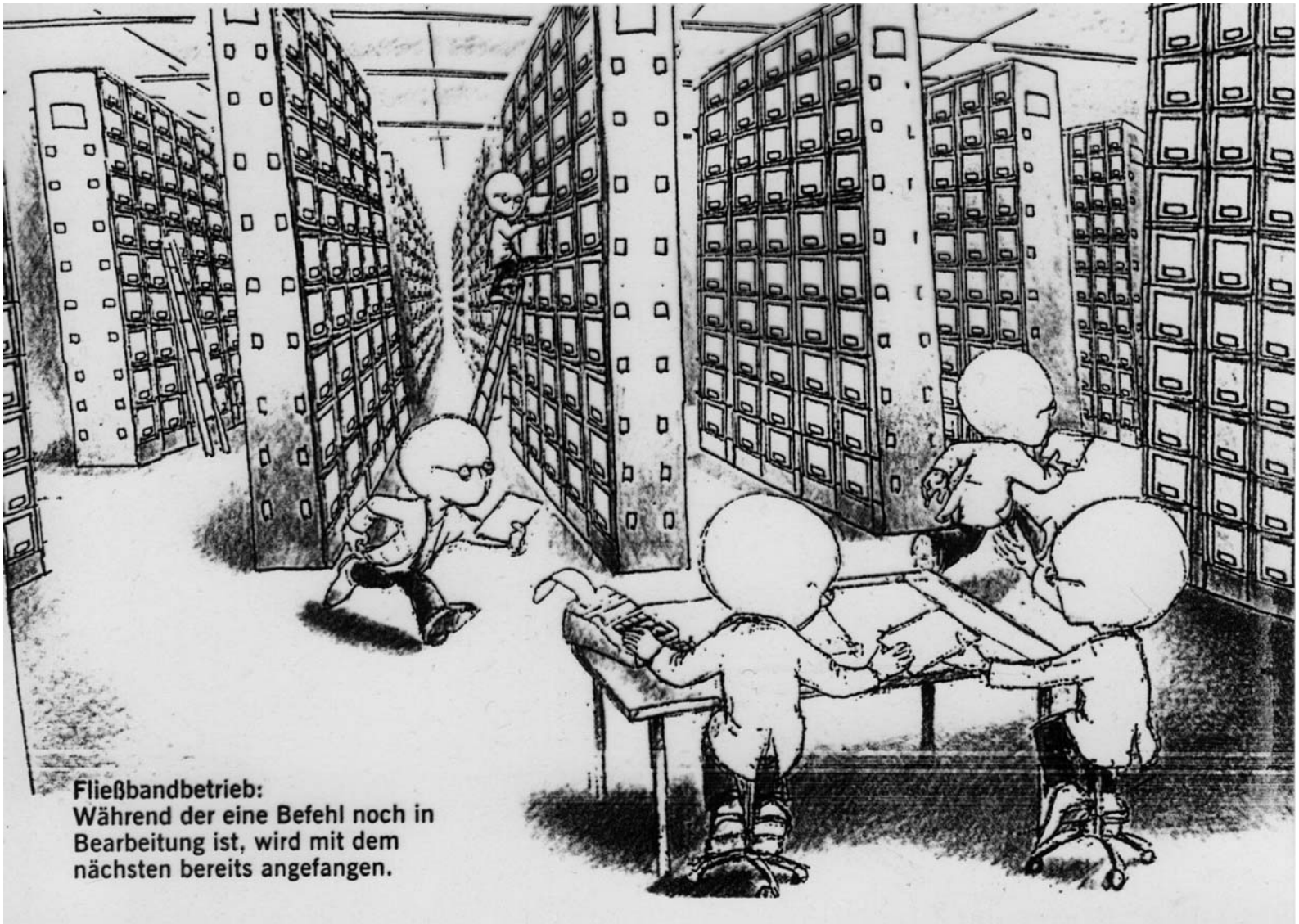
Advantages of Pipelines:

If pipeline is filled: per clock unit one result is achieved.
All additions should be organized such that the pipeline is always filled!

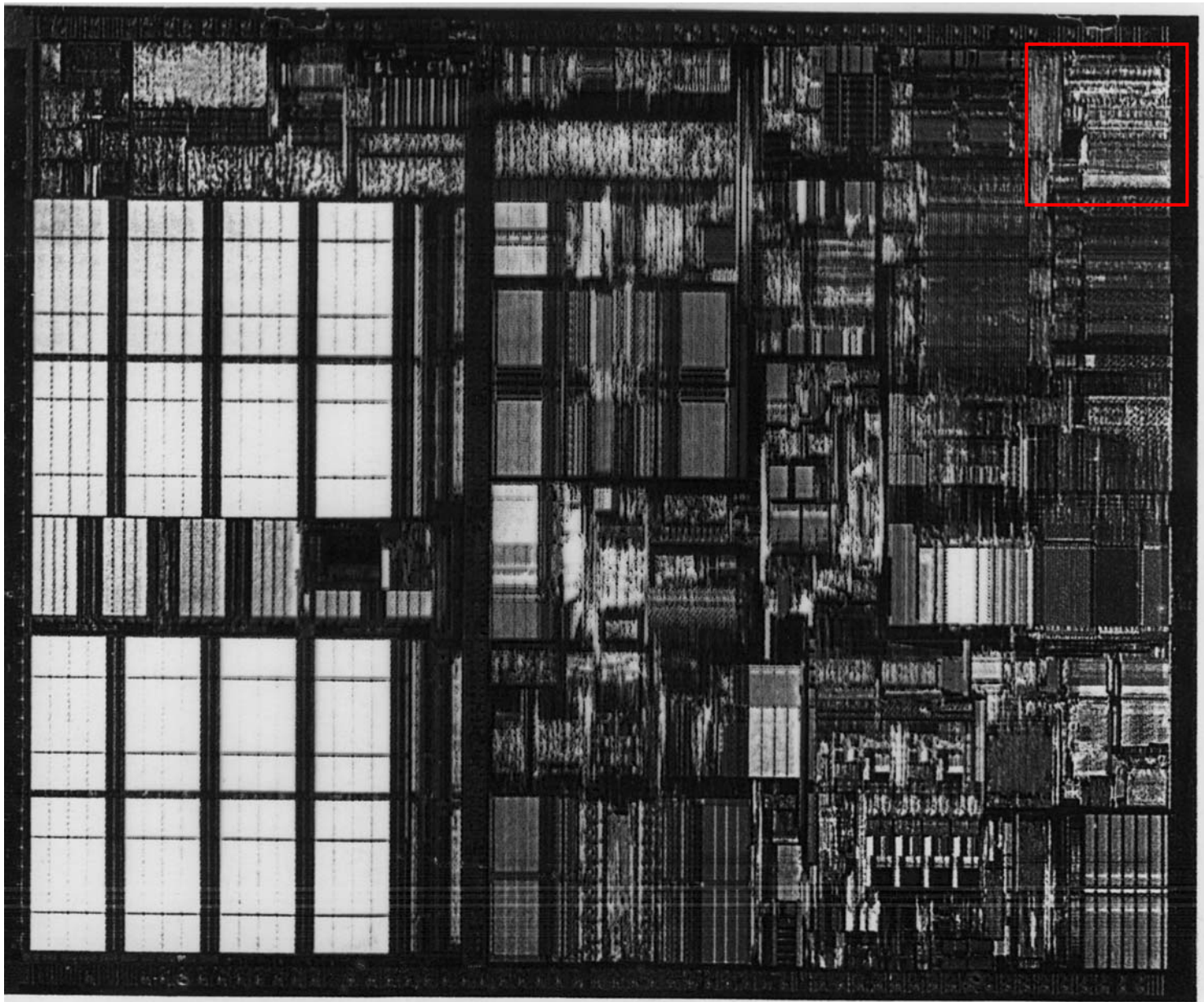
If the pipeline is nearly empty, e.g. in the beginning of the computations, it is not efficient!

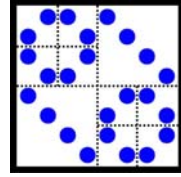
Major task for CPU: Organize all operations such that the operands are just in time at the right position to fill the pipeline and keep it full.

CPU - Pipelining



CPU





General Steps

Instruction Fetch: Get the next command

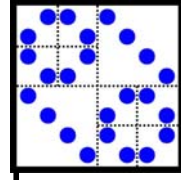
Decoding: Analyse instruction and compute addresses of operands

Operand Fetch: Get the values of the next operands

Execution step: Carry out command on operands

Result Write: Write result in memory

Pipelining of these steps, and also inside each step.

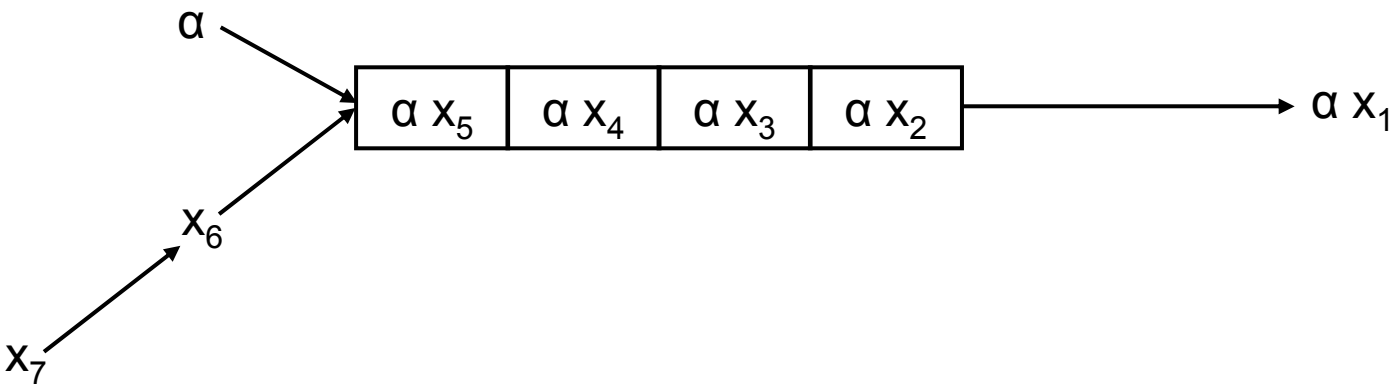


Special case: Vector instruction

For set of data the same operation has to be executed on all components.

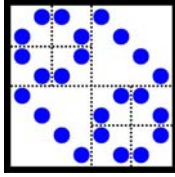
$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \alpha \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

For j = 1,2,..,n: $y_j = \alpha\, x_j$;

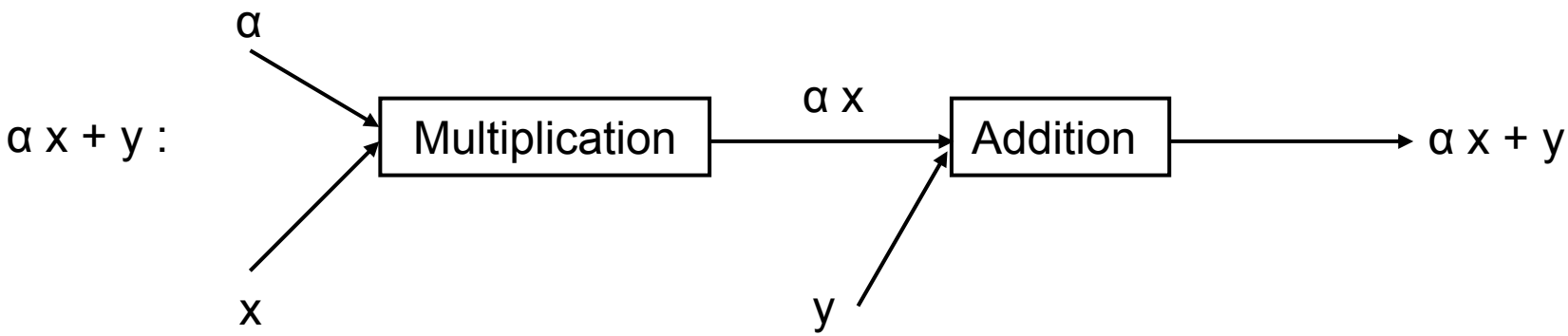


Total costs: Startup time + vector length * clock time

(pipeline length + vector length) * τ



Chaining: Combine pipelines directly



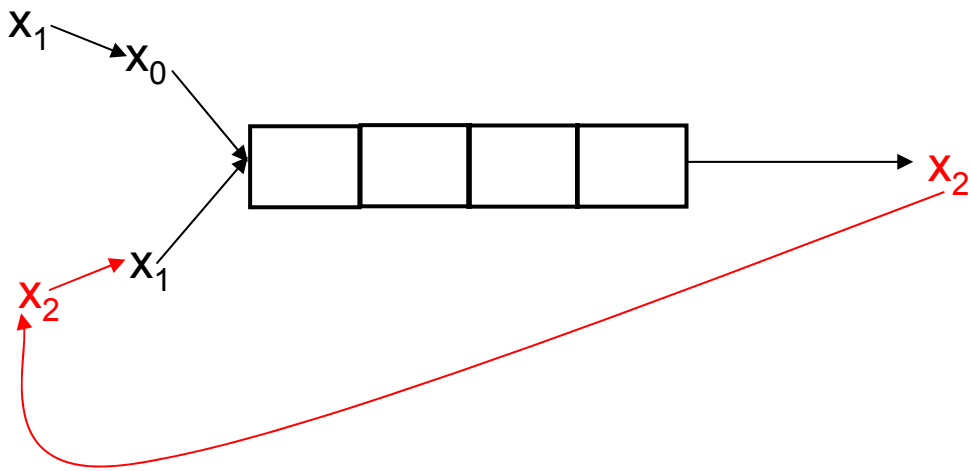
Advantage:

total cost = startup time + vector length * clock time

Problem: Data Dependency !

Example: Fibonacci numbers

$x_0 = 0, x_1 = 1, x_2 = x_0 + x_1, \ldots, x_i = x_{i-2} + x_{i-1}$;

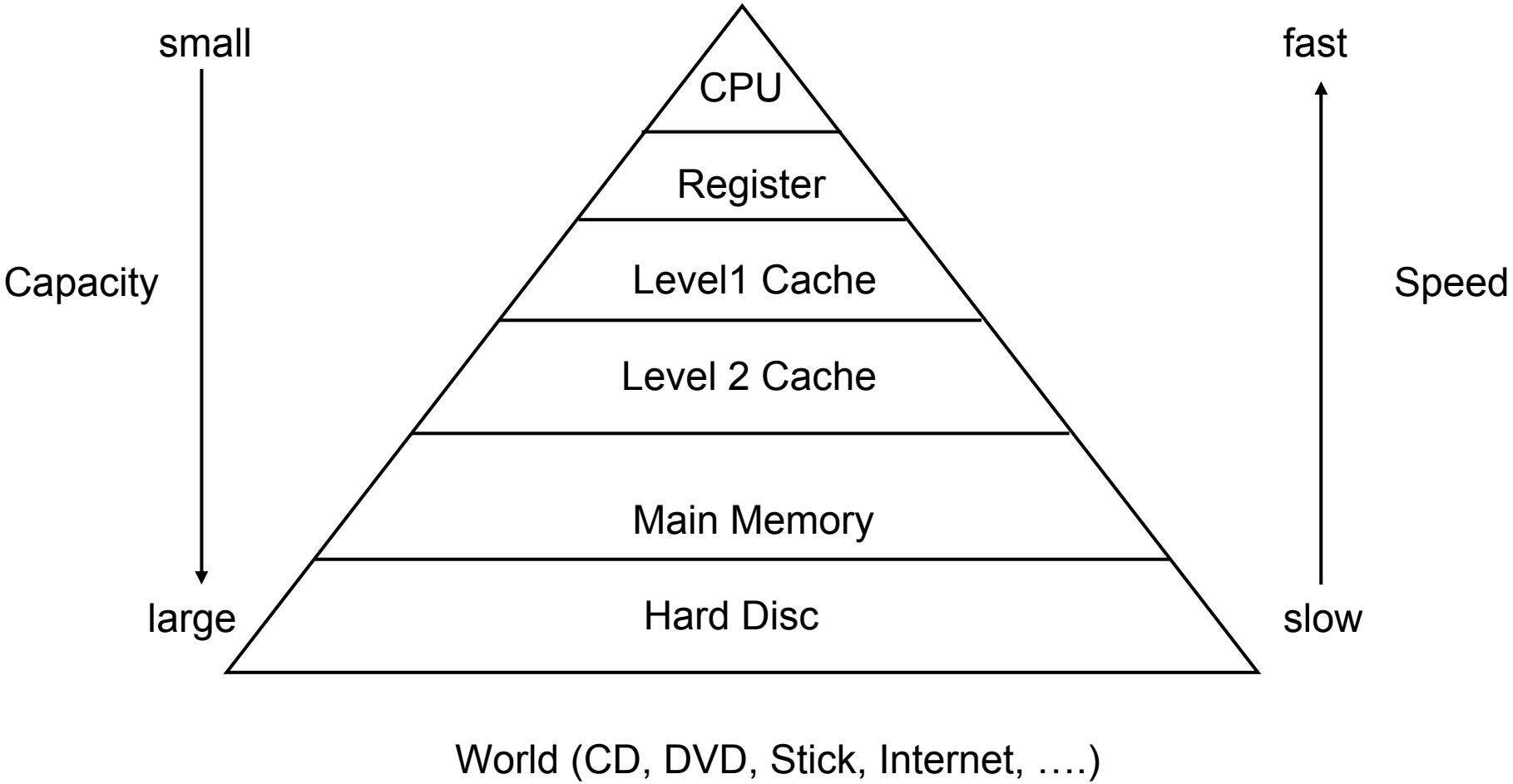


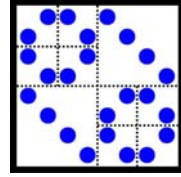
After filling in x_0 and x_1 the next pair needs x_2 which is known only after the first computation is finished!

Pipeline contains always only one pair – is nearly empty all the time!

Similar Problem for recursive subroutine calls.

1.1.2 Memory Organization





General Considerations

DRAM (dynamic): Fast, periodically refreshing is necessary
SRAM (static)
→ SDRAM (small part SRAM combined with large DRAM)

DDR: (double date rate) use both voltage flanks (side,shoulder)

Necessary time for reading:
- Transport addresses via bus to memory (bus speed)
- Time between arrival of adresses and arrival of data:
  latency time (~4 cycles)
- Refreshing of data: 20-40 cycles
- Transport of data from memory: 1 bus cycle

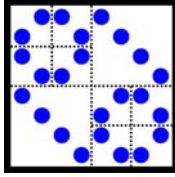
Cache Idea

Cache as memory buffer between large, slow memory and small, fast memory.

By considering the data flow (last used data), try to predict which data will be requested in the next step:

- keep the last used data in fast cache because it is likely that the same data will be used again
- keep also the neighborhood of the last used data in fast cache.

Memory is organized in pages (main memory, hard disc,..). Hence, together with the last used data put the whole page in the cache.
Page size ~ bus band width

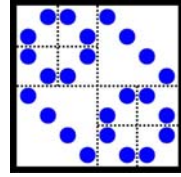


Cache hit:
The data requested from the small, fast memory is found in the cache: Copy the data to fast memory. Done.

Cache miss:
The data requested from the small, fast memory is not found in the cache:

Look for data in the large, slow memory.
Copy the related page to the cache (removing the oldest cache entry) and copy it to the fast, small memory.

Also: Reuse data as often as possible!
Working blockwise to ensure neighbouring!



Mapping between Memory

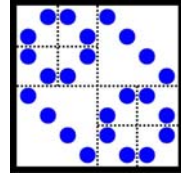
Direct mapping:

Address	10101	110	→ 110	in cache, modulo
	11100	001	→ 001	
	11110	110	→ 110	

Disadvantage: Immediately replacing of data in cache

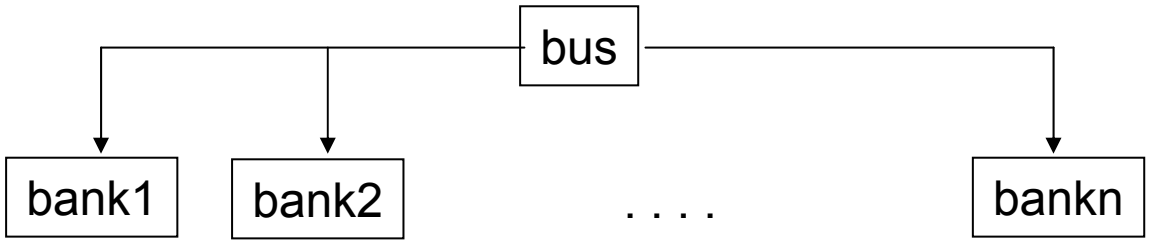
Associative mapping:

Partition cache in blocks.
Write data to direct mapped address in one of the blocks.
Replace oldest data in block.



Cyclic data distribution

Memory is often organized in banks connected by bus:



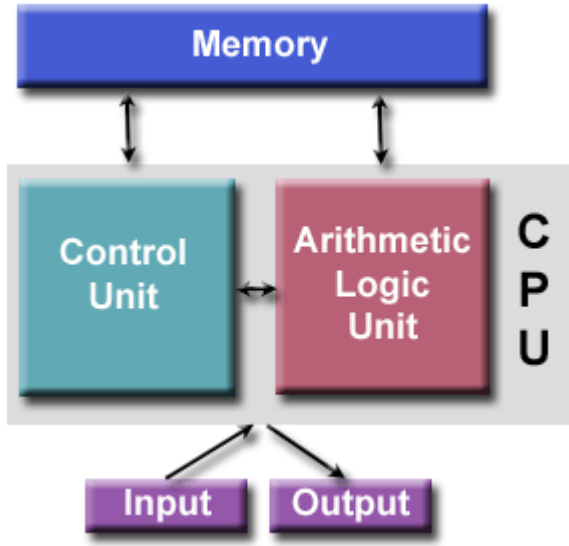
Per cycle n operands can be fetched out of the n banks.
Storing vectors!

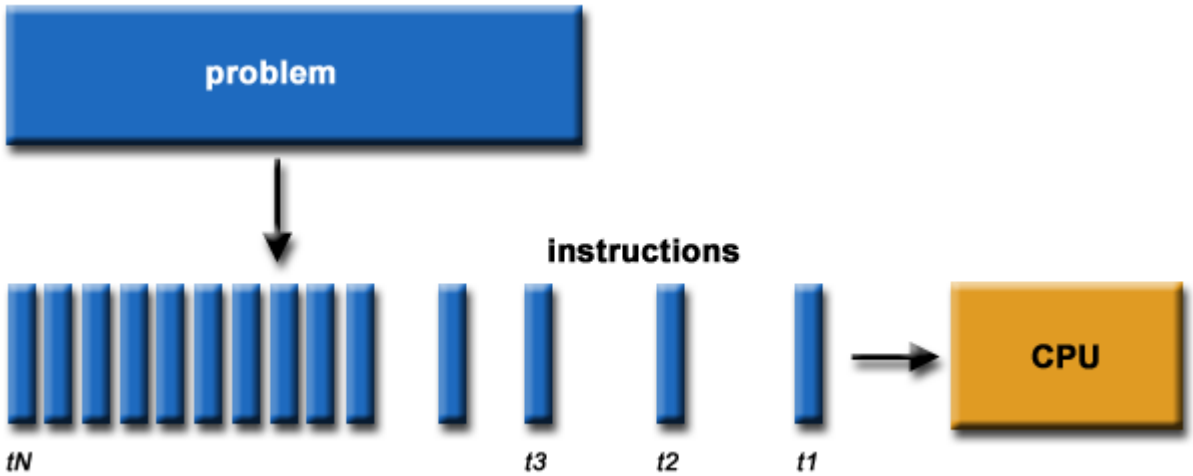
x_1 in bank1 x_2 in bank2,…. allows one step access to x

1.1.3 Parallel Processors

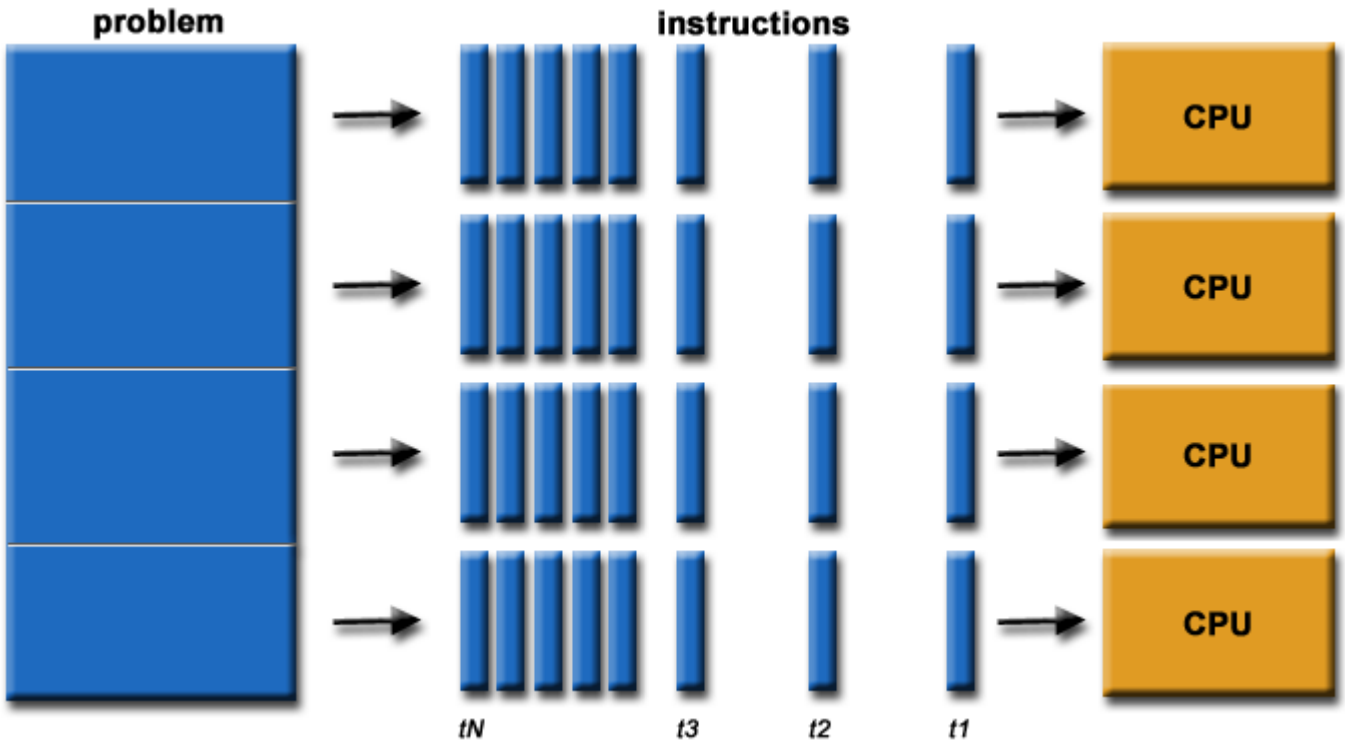
Classical von Neumann model:

Code and data in memory!
Control unit fetches instructions and data from memory and sequentially coordinates the operations.





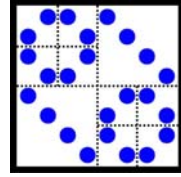
Parallel Computation



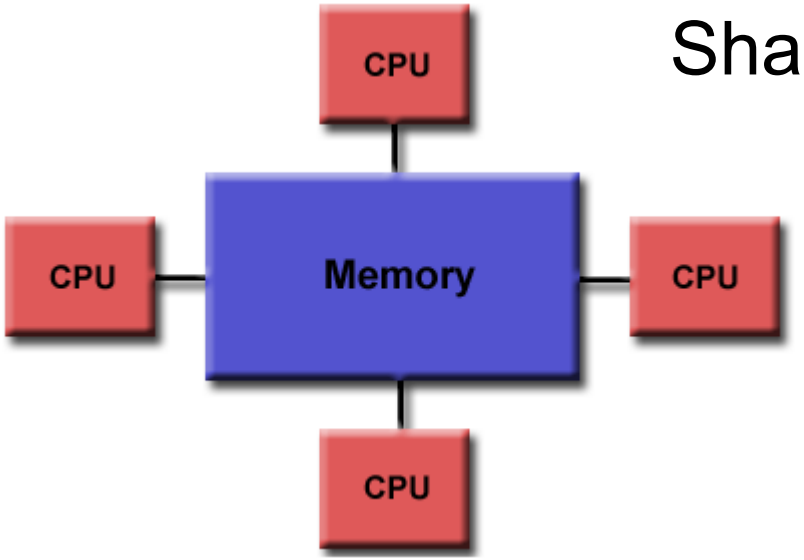
Flynn's taxonomy:

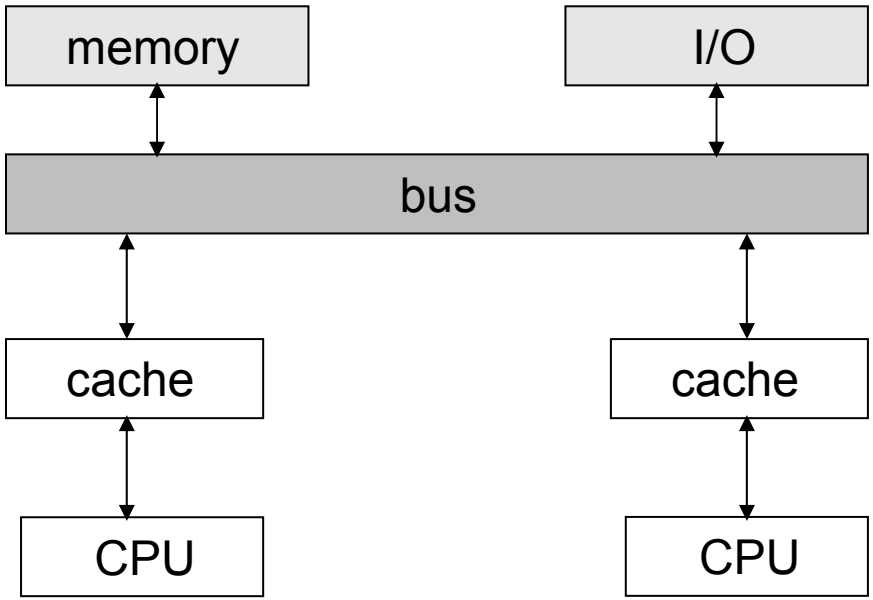
MIMD architecture: multiple instructions – multiple data

(compare to SISD = "single instructions – single data" , etc.)



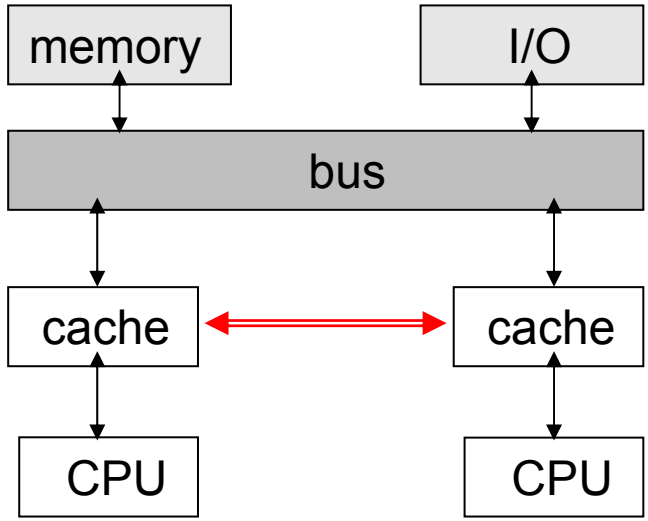
Shared Memory (SMP):





processors

Cache Coherence




```
start = proc_number;
for (S=0; s<S_max; S++)
   parallel for(i=start; i<N; i+=S+proc_number)
      x[i]=2.;
```

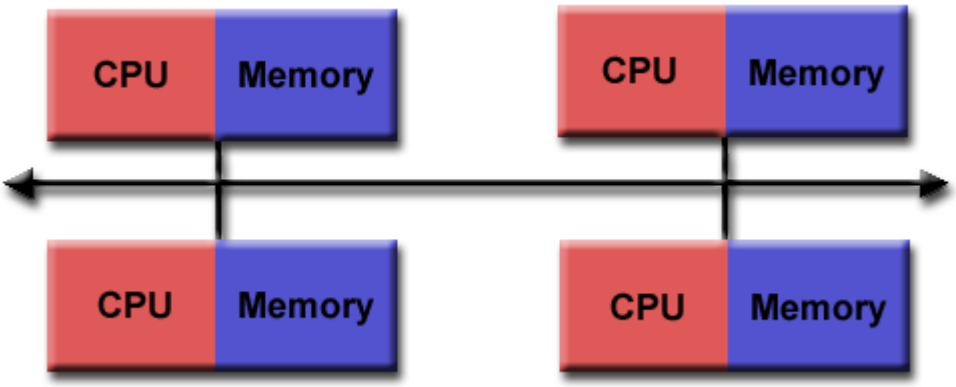
For S=0 and 2 threads:
Thread 1 changes x(0,2,4,…) and thread 2 changes x(1,3,5,…).

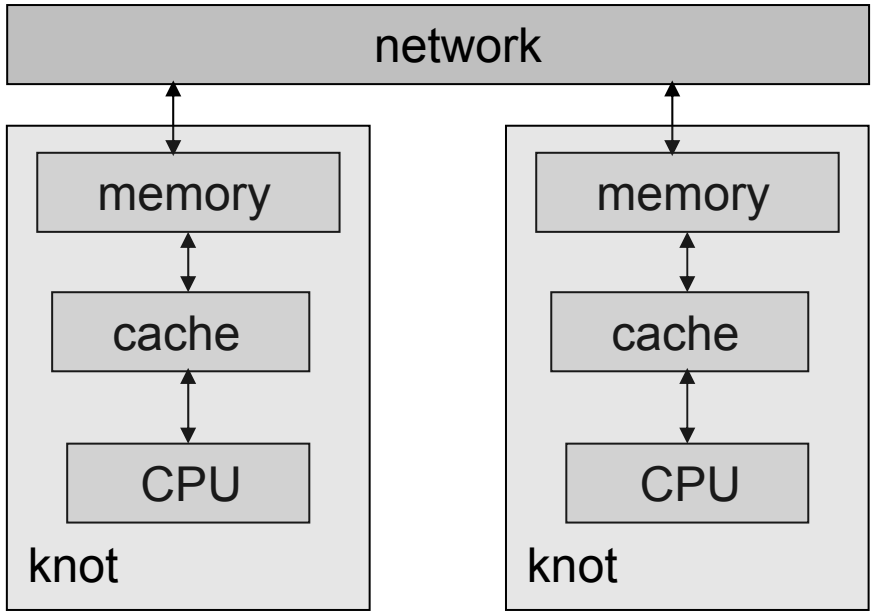
Does the cache contain 4 words (cache line = 4), then each changing step of thread 1 also changes data that is also contained in the cache of thread 2 (and vice versa). Otherwise the data in the two caches is not consistent anymore!

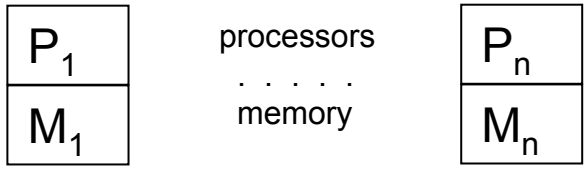
To retain the right values in both caches after each changing step also the Value in the other cache has to be renewed!

Leads to a dramatical increase of computational time,
ev. slower than sequential computation!

Locally distributed memory:





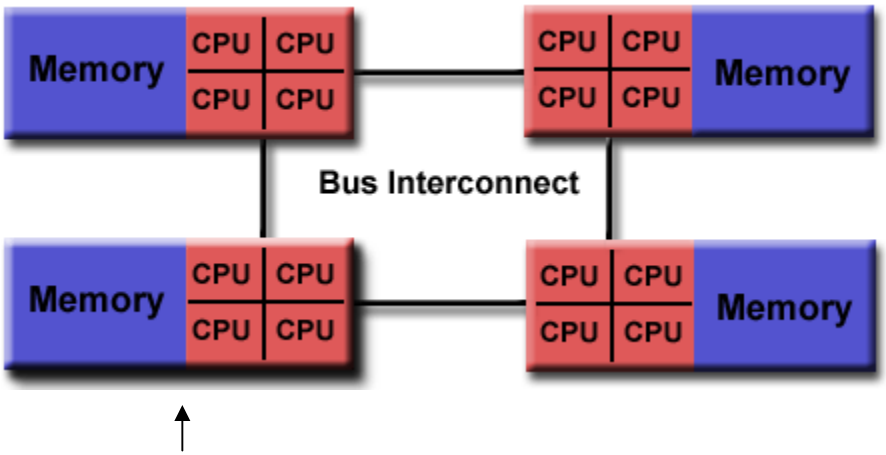


Virtual shared memory:

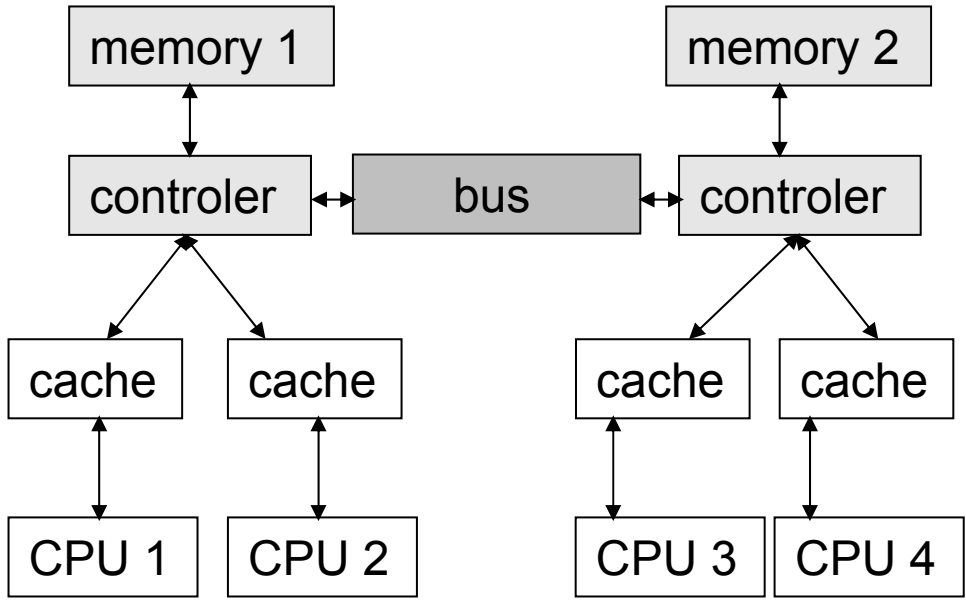
Distributed data but organized as shared memory.

Nonuniform Memory Access

Cluster of multiple CPU processors



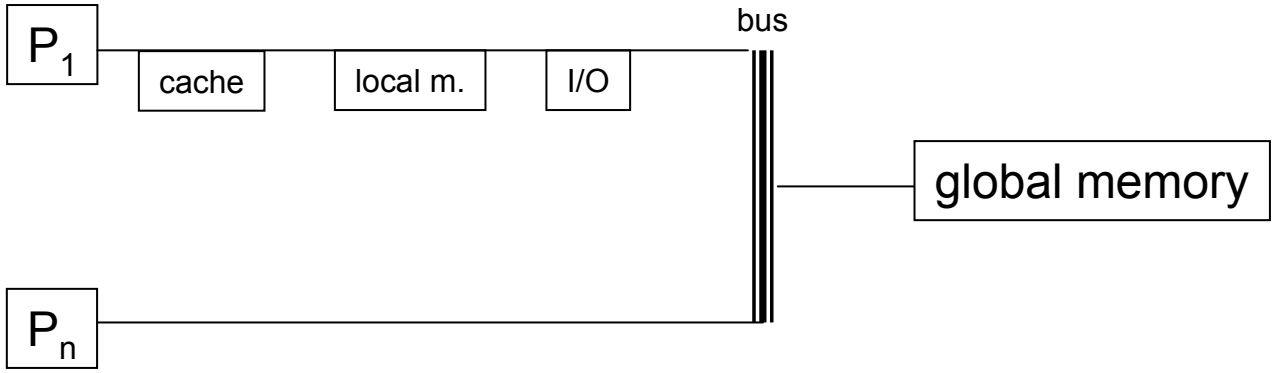
Symmetric multiprocessor



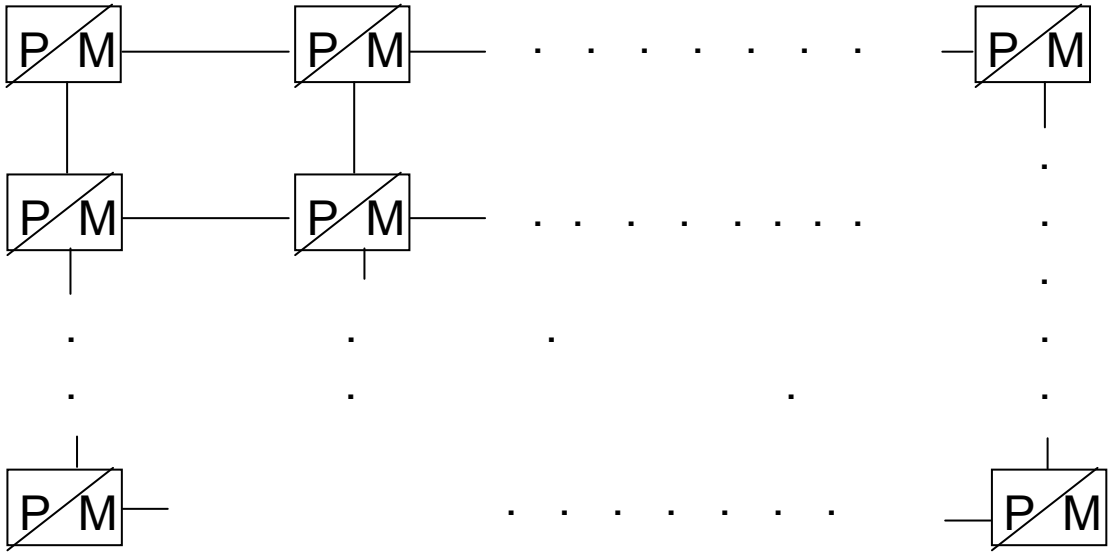
Different types of communication!
Shared memory and distributed memory!

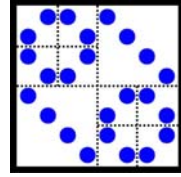
Topology of the processor/memory interconnection

Bus:



Mesh (Array, Grid): p processors, longest path sqrt(p)





Time for sending data from one processor to another depends on the connection network topology:

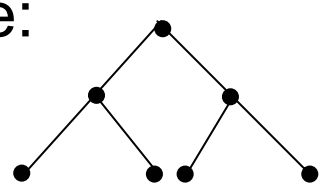
Mesh: 2*sqrt(p)

vector or ring: p-1 or p/2

tree: 2 log(p)

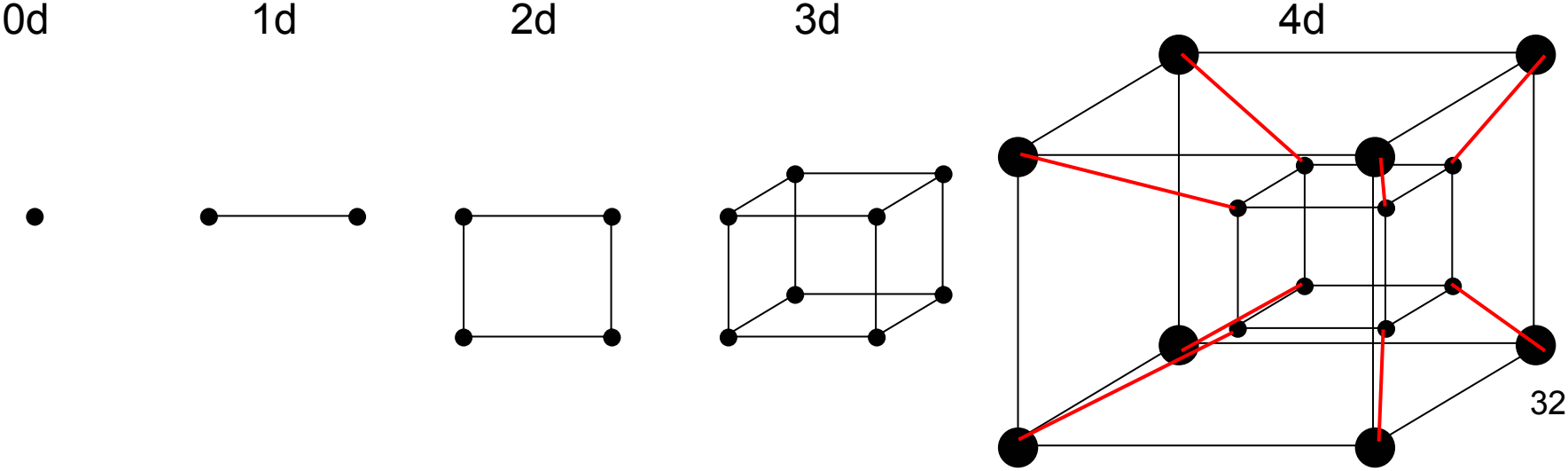
hypercube: log(p)

Tree:



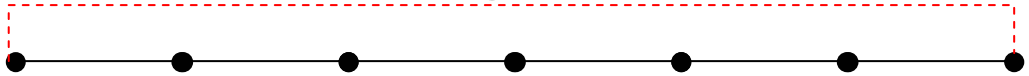
Hypercube:

0d 1d 2d 3d 4d

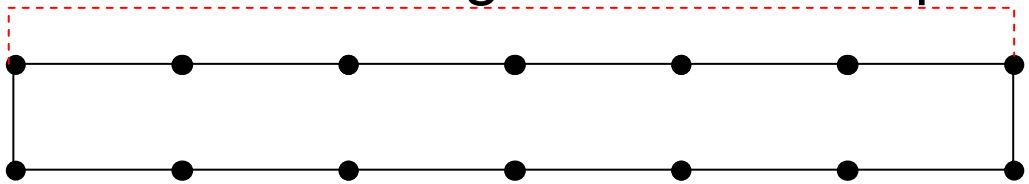


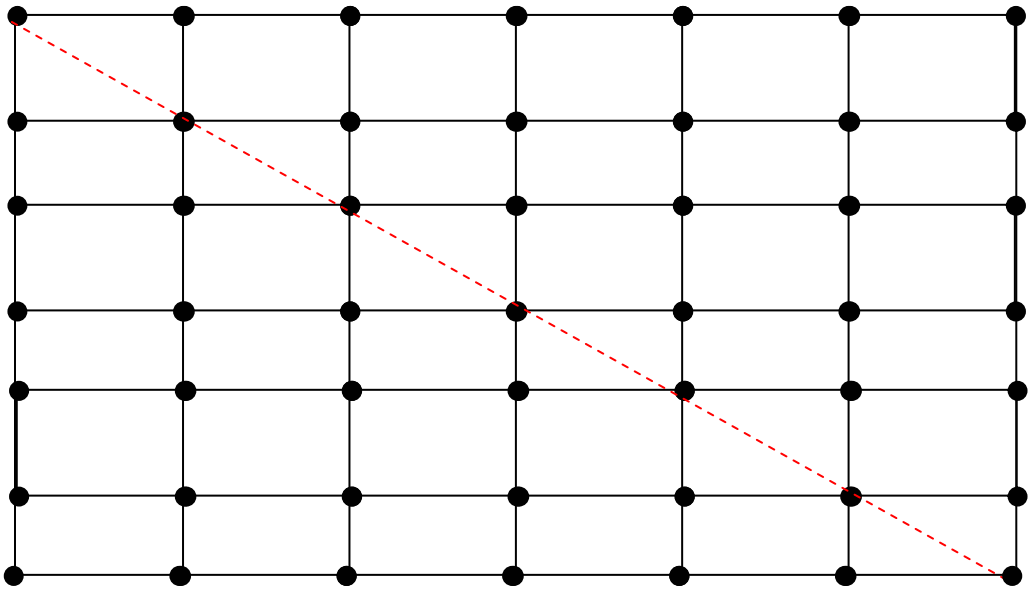
Communication - Topology

diameter = largest distance = p-1:



diameter = largest distance = p/2:

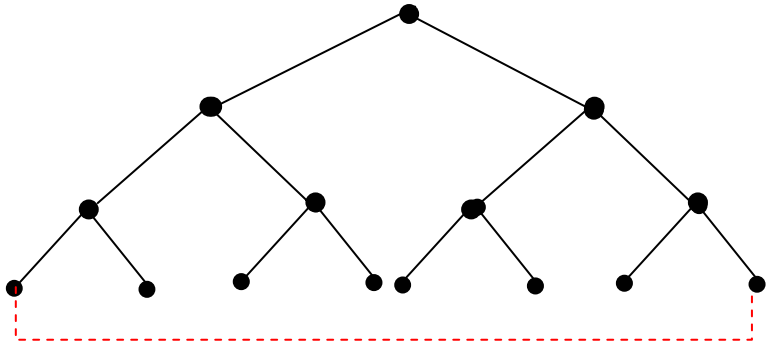




diameter =

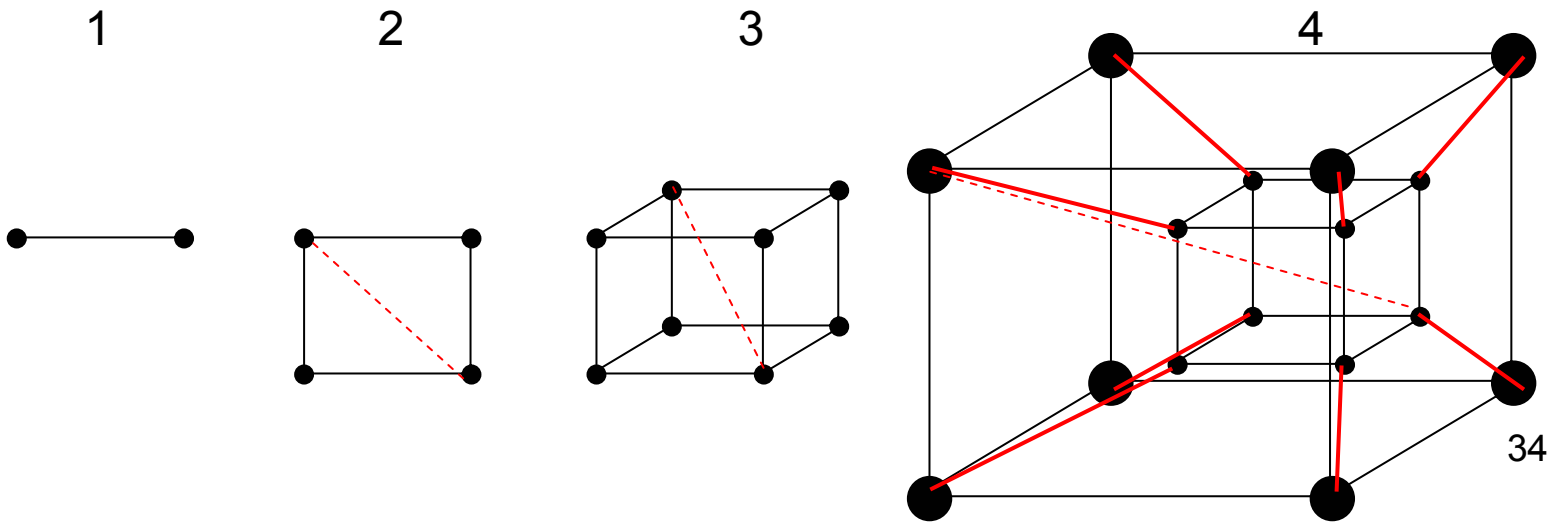
largest distance =

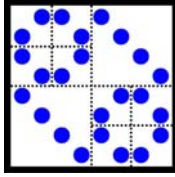
2 sqrt(p)



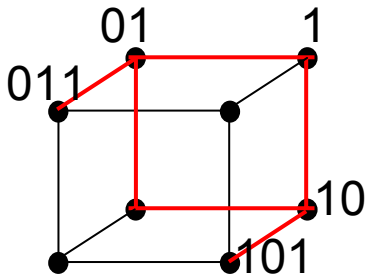
Diameter = largest distance = 2 ln(p)

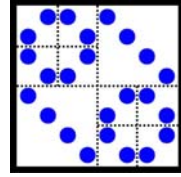
Diameter:





Tree in Hypercube





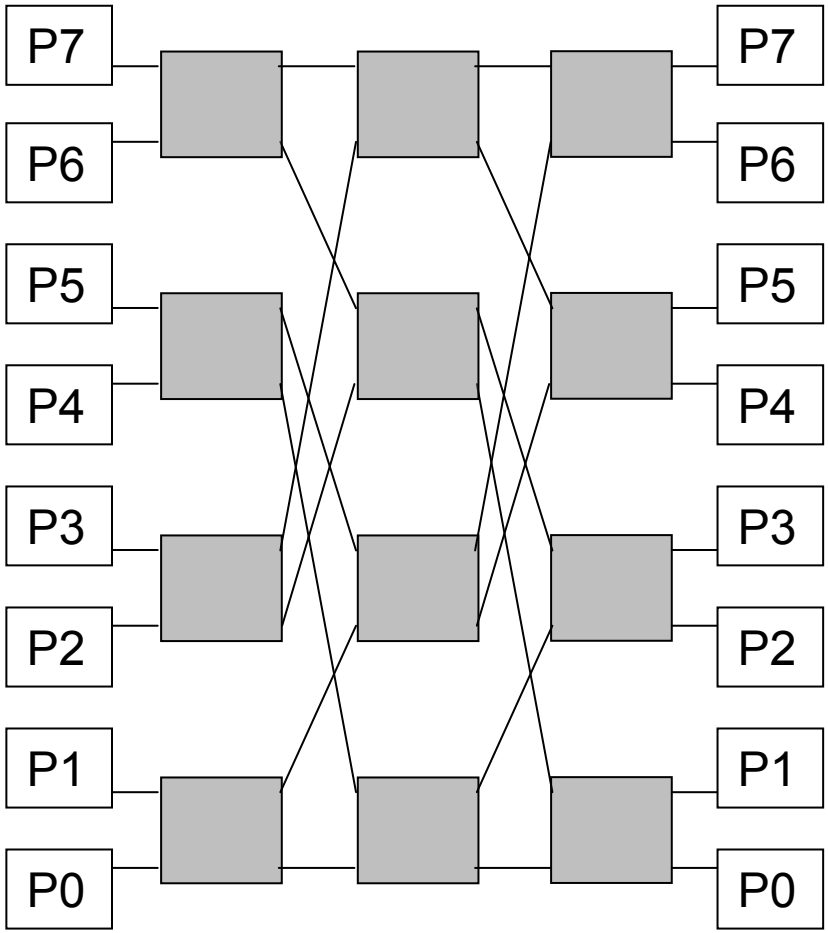
Different Topologies:

G	p	Diam(G)	Degree(G)	Edges(G)
$G_1(n)$	n	n-1	2	n-1
$T_1(n)$	n	floor(n/2)	2	n
$G_2(n,n)$	n^2	2n-2	4	$2n^2$ -2n
$T_2(n,n)$	n^2	2*floor(n/2)	4	$2n^2$
BT(h)	2^{h+1} - 1	2h	3	2^{h+1} -2
HC(k)	2^k	k	k	2^{k-1} k

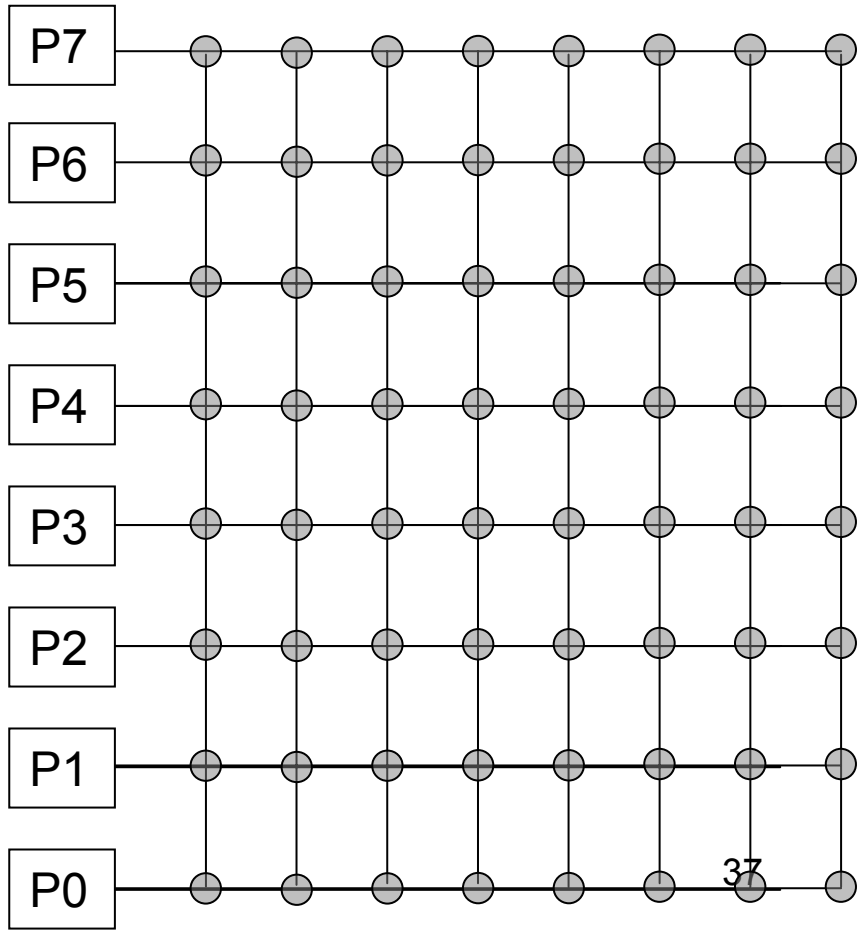
G: Grid, T: Torus, BT: binary Tree, HC: Hypercube

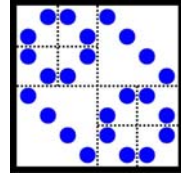
Network based on Switches

3-level Omega network



Crossbar

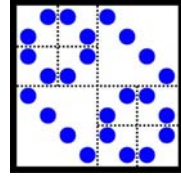




Communication

Crossbar: Direct, independent connection between all processors. Nonblocking!

Omega network: Blocking network.
Simultaneous connection P0 – P6 and P1 – P7 is not possible!
Turn-over of switches necessary!



1.1.4 Performance Analysis

Definition: *Computational Speed* $r = N / t$ *Mflops,*
N floating point operations in t microseconds

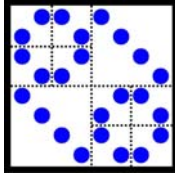
or by known speed r:
time for N flops is given by $t = N / r$

Amdahls's Law:

Setting:
An algorithm takes N flop's.

A fraction f is carried out with speed of V Mflops (good in parallel)
A fraction 1-f is carried out with S Mflops (bad)

f : high speed parallel	1 – f : low speed, strongly sequential

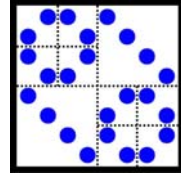


Total CPU time: $t = \frac{f \cdot N}{V} + \frac{(1-f) \cdot N}{S} = N \cdot (\frac{f}{V} + \frac{1-f}{S})$

Overall speed (performance): $r = \frac{N}{t} = \frac{1}{\frac{f}{V} + \frac{1-f}{S}}$

(Amdahl's Law)

f must be close to 1 in order to benefit significantly from parallelism



Discussion

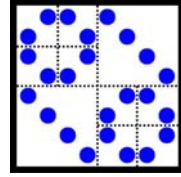
$$r = \frac{N}{t} = \frac{1}{\frac{f}{V} + \frac{1-f}{S}} \leq \frac{1}{1-f} \cdot S$$

with S the slow speed

To achieve large speed, 1-f has to be small!

For very large “parallel” speed V: $r = \frac{N}{t} = \frac{1}{\frac{f}{V} + \frac{1-f}{S}} \approx \frac{1}{0 + \frac{1-f}{S}} = \frac{S}{1-f}$

The total speed is governed by the fraction of the “strongly sequential” part of the algorithm that cannot be parallelized.



Speedup

Executing a job using p processors in parallel we can achieve a speedup.

Define t_p := wall clock time to execute the job on p parallel processors

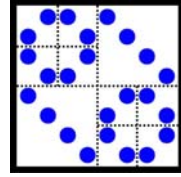
Speedup: $S_p := t_1 / t_p$ is the ratio of execution time with 1 versus p processors

In the ideal case it would hold $t_1 = p\, t_p$.

Efficiency using p processors: $E_p = S_p / p$. $0 <= E_p <= 1$

$E_p \approx 1$: very good parallelizable, because then $S_p \approx p$ or $t_1 \approx p\, t_p$.
Problem scales.

$E_p \approx 0$: bad, because $E_p = S_p / p = t_1 / (p\, t_p)$ and $t_1 << p\, t_p$.



Ware's Law

Using the same definition of speed and fraction f as above:

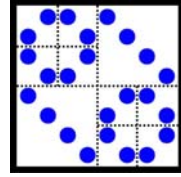
ideally parallel

$$t_p = \overbrace{\frac{f \cdot t_1}{p}} + \underbrace{(1-f)t_1}_{\text{strongly sequential}} = t_1 \cdot \frac{f + (1-f)p}{p} \geq (1-f)t_1$$

$$S_p = \frac{t_1}{t_p} = \frac{1}{\frac{f + (1-f)p}{p}} = \frac{p}{f + (1-f)p} \leq \frac{1}{1-f}$$ Ware's Law

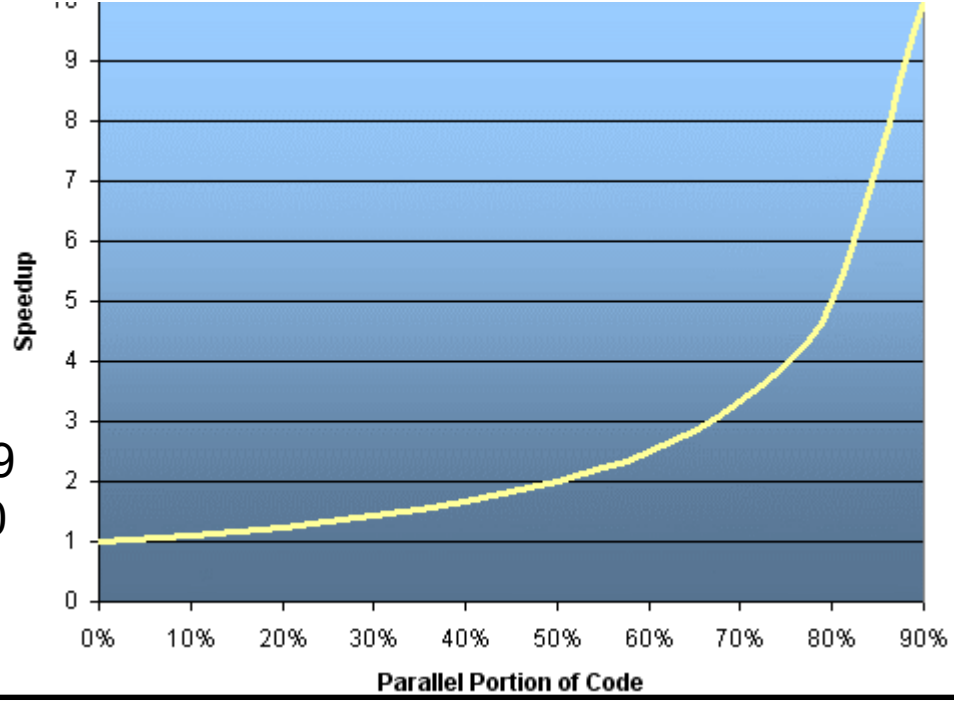
$$E_p = \frac{S_p}{p} = \frac{1}{f + (1-f)p} \leq \frac{1}{(1-f)p} \Longrightarrow \quad p \to \infty : \quad E_p \to 0$$

We always will have a small portion of our algorithm that is not parallelizable and therefore the efficiency will always be zero in the limit!

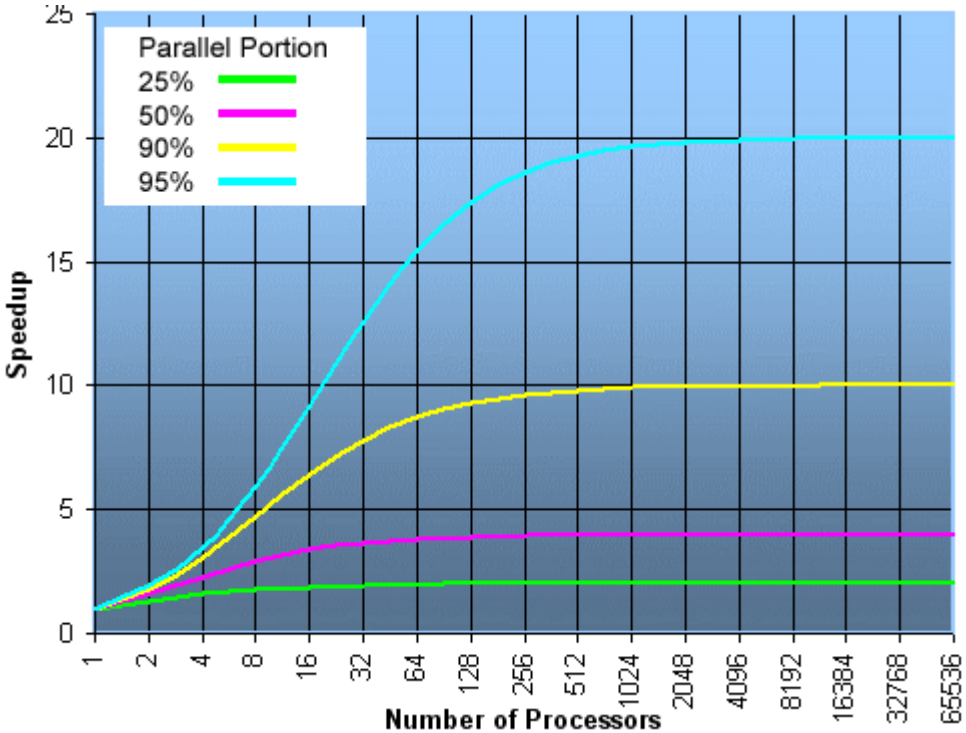


1/(1-f):

f:	0.1	0.2	0.5	0.9
S:	10/9	10/8	2	10

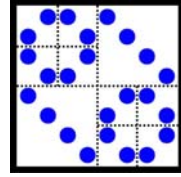


Reachable Speedup for large p.



Speedup depending on p.
Saturation 1/(1-f).

Gustafson's Law



Other model:
We assume that the problem can be solved in 1 unit of time on a parallel machine with p processors.

Fraction f is good parallelizable, 1-f not

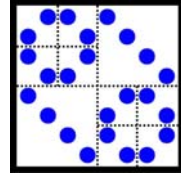
Compared with this parallel implementation an uniprocessor would perform

(1-f) + f p

for the same job.

Speedup: $$S_{pf} = \frac{t_1}{t_p} = \frac{1 - f + fp}{1} = p + (1-p)(1-f)$$

Efficiency: $$E_{pf} = \frac{S_{pf}}{p} = \frac{1-f}{p} + f \overset{p\to\infty}{\to} f$$



Example

f = 0.99, p = 100 or 1000:

Amdahl/Ware: $S = p/(f+(1-f)p))$ $E = 1/(f+(1-f)p)$

p=100: $S_{100} = 100/1.99 \sim 50$, $E_{100} = 0.5$,

p=1000: $S_{1000} = 1000/10.99 \sim 100$, $E_{1000} = 0.1$,

Gustafson: $S = 1-f+fp$ $E = (1-f)/p+f$

p=100: $S_{100f} = 99.01$, $E_f = 0.9901$

p=1000: $S_{1000f} = 990.01$, $E_f = 0.99001$