



Technical University of Denmark



High Performance Computing

FORTRAN, OpenMP and MPI

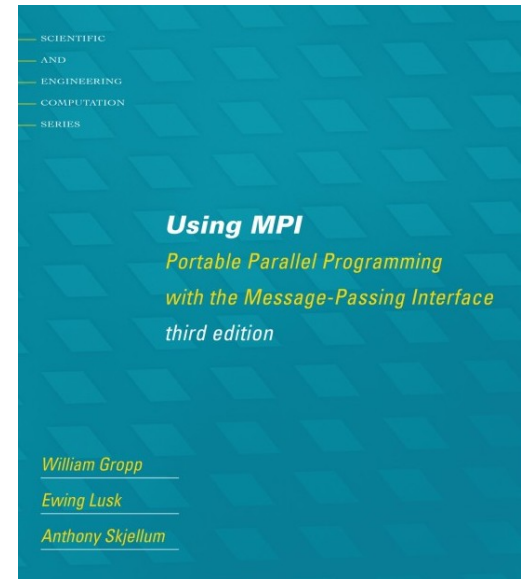
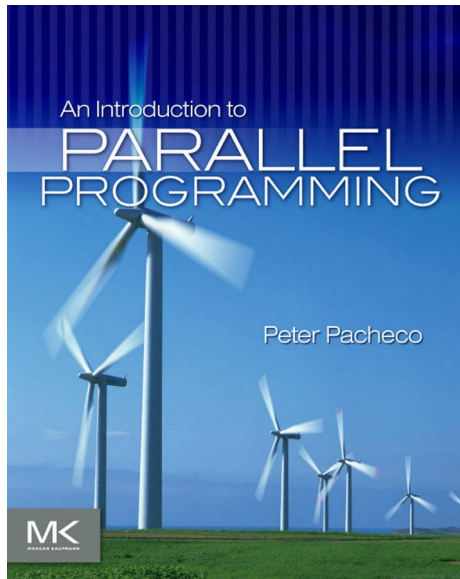
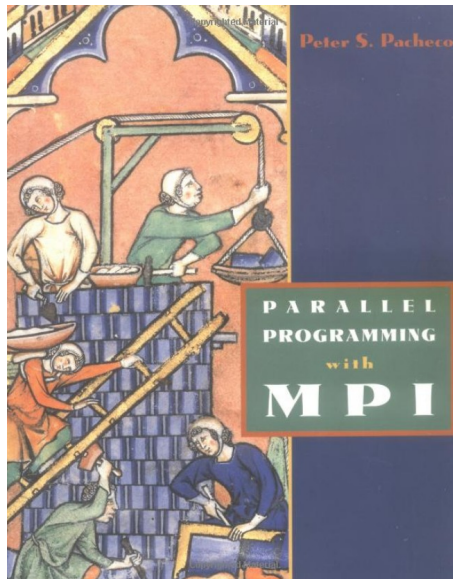
41391

Content – Week 3: MPI

- Monday:
Control structure, communication models, performance evaluations, pipelining and vectorization.
- Tuesday:
Message passing basics, MPI 1: point-to-point and collective communications.
- Wednesday:
MPI 2: derived data types, data packing, process groups, and communicators, virtual topologies, error handling, parallel debugging.
- Thursday:
Domain decomposition, load balancing, data abstractions.
- Friday:
Operator abstractions, parallel libraries.

Literature

- www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
- Other literature:



Content – MPI: Day 1

- Control structures and communication models.
- Performance evaluation.
- Pipelining.
- Vectorization.

Control structures and communication models

Control structures

A parallel program must specify:

- Control structure:
 - Define *concurrency* (concurrent computing: multiple tasks can be in progress simultaneously).
- Communication model:
 - *Interaction* between concurrent tasks.
- Compute model:
 - Conceptual view of the types of available *operations*.
- Independent of specific language syntax and hardware architecture.

Control structures

Loose categories:

- Data parallelism (same instruction to multiple data)
 - Vector processors.
 - OpenMP, HPF, Coarray Fortran.
- Shared memory:
 - Parallelism is not specified by data independence.
 - Programmer is in control.
 - Threads, UPC (Unified Parallel C), Linda.
- Message passing:
 - Data transfer between processes requires action from both of them.
 - Independent of interconnect or memory architecture.
 - MPI is one specific realization.

Control structures

Some definitions:

- Process:
 - A process is an *address space*:
 - 1) the executable machine language program
 - 2) A block of memory, which will include the executable code, a call stack that keeps track of active functions, a heap, and some other memory locations.
 - 3) Descriptors of resources that the operating system has allocated to the process – fx. file descriptors
 - 4) Security information
 - 5) Information about the state of the process, such as whether the process is ready to run or is waiting on some resource, the content of the registers and information about the process's memory.

Control structures

Some definitions:

- Thread:
 - A thread is characterized by a program counter (*counter*: processor register that indicates where a computer is in its program sequence) and an execution stack.
- A single process may have one or several threads associated with it (POSIX multi-thread standard).

Control structures

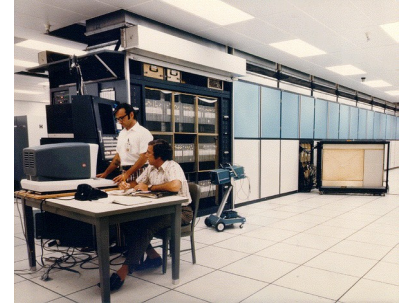
A parallel program must define the control structure:

- Organization:
 - Operate under centralized control of a single control unit (Master processor) – a *master-slave* paradigm, or:
 - Work independently, all processes are equal.
- **SIMD:**
 - **S**ingle Instruction (all proc execute the same ops).
 - **M**ultiple **D**ata (each proc works on different data).
- **MIMD:**
 - **M**ultiple Instructions (each proc executes different ops).
 - **M**ultiple **D**ata (each proc works on different data).

Control structures

SIMD processors:

- Examples include many early parallel machines:
 - Illiac IV, Connection Machine (CM), MasPar, MPP,
- Found in today's co-processors:
 - SSE (streaming SIMD Extensions), AltiVec.
- Relies on regular structure of computations:
 - Media processing.
 - Kernel computations (linear algebra, FFTs,)



Above: Illiac IV
Below: CM



Control structures

MIMD processors:

- Current-generation supercomputers:
 - Shared memory:
 - Multi-processor PCs, SGI Origin, Cray X1.
 - Distributed memory:
 - Beowulf clusters, Cray, IBM, Dell, Fujitsu, SGI, Bull.
- Usually **SPMD** programming paradigm:
 - **Single Program Multiple Data**.
 - Same program runs on all processors, behavior is conditional on processor ID (e.g., MPI).
- **MPMD** programming paradigm:
 - **Multiple Program Multiple Data**.
 - Each processor executes a different program (e.g., PVM = Parallel Virtual Machine).



Control structures

SIMD vs. MIMD:

- SIMD platforms:
 - Special-purpose, not suited for arbitrary programs.
 - Expensive (special hardware required).
 - Single control unit = less hardware needed.
 - Need less memory (only one copy of the program).
- MIMD platforms:
 - Suitable for broad range of applications.
 - Inexpensive (off-the-shelf hardware).
 - Need more memory (OS and program on each proc).

Control structures

What about the others ?

- SISD:
 - Single Instruction Single Data.
 - Not a parallel platform, strictly sequential.
- MISD:
 - Multiple Instructions Single Data.
 - Not used since the result may depend on the sequence of instructions (non-deterministic).

Control structures

Communication models:

- Two primary forms of exchanging data between parallel processes:
 - Accessing a shared data space.
 - Exchanging messages (data).
- Platforms that provide a shared data space:
 - Shared memory platforms.
 - Multiprocessors.
- Platforms that support messaging:
 - Message passing platforms (e.g., MPI).
 - Multicomputers.

Control structures

Shared memory platforms:

- Components:
 - Set of processors.
 - One memory, accessible to all processors.
- Inter-processor communication:
 - Modify data objects stored in the same memory.
 - Locking and semaphores* are important.
 - Memory bottleneck!
- Flavors of shared memory:
 - UMA (uniform memory access).
 - NUMA (non-uniform memory access).

(*) a **semaphore** is a variable that provides a simple but useful abstraction for controlling access by multiple processes to a common resource in a parallel programming environment (wikipedia)

Control structures

UMA and NUMA shared memory platforms:

- UMA: memory access time is the same for all processors.
- NUMA: different procs have different access times.
- Differ with respect to locality:
 - Algorithms must exploit locality for performance of NUMA.
- Easy communication:
 - Read/write visible to all processors.
- Tricky coordination:
 - Locking/coordination of shared data.
 - Critical program sections (e.g. in OpenMP).
 - Cache coherence.
 - Automatic or explicit flush of cache to memory before sharing of data.

Control structures

Message passing platforms:

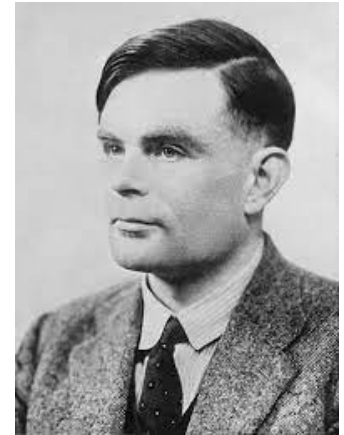
- Components
 - Set of processors.
 - Each processor has its own, exclusive memory (space).
- Inter-process communications:
 - Exchange of data using explicit send/receive operators (e.g. using MPI, PVM libraries).
- Examples:
 - Clusters of workstations.
 - Grids.
 - Multi computers (Cray, IBM BlueGene, Fugaku...)



Control structures

Advantages of message passing

- Universality:
 - Works on all hardware architectures.
 - Works on shared and distributed memory.
- Completeness:
 - All parallel algorithms can be expressed.
 - Turing-complete.
- Performance:
 - No memory bottlenecks.
 - Carefully hand-crafted communications.
 - No theoretical speedup limit.



Alan Turing

Turing complete

- In computability theory, a system of data-manipulation rules (such as an instruction set, a programming language, or a cellular automaton) is said to be Turing complete or computationally universal if and only if it can be used to simulate any single-taped Turing machine and thus in principle any computer. A classic example is the lambda calculus. The concept is named after Alan Turing.
- In practice, Turing completeness means that the rules followed in sequence on arbitrary data can produce the result of any calculation. In procedural languages this can be satisfied by having, at a minimum, conditional branching (e.g., an "if" and "goto" statement) and the ability to change arbitrary memory locations (e.g., variables).
- To show that something is Turing complete, it is enough to show that it can be used to simulate the most primitive computer, since even the simplest computer can be used to simulate the most complicated one.
- Ref: Wikipedia

Control structures

Message passing vs. shared memory:

- Message passing:
 - Only requires a network.
 - More involved to program since send/receive operations need to be coded explicitly.
 - In theory unlimited scalability.
- Shared memory:
 - Special hardware (memory controller, backplane) needed.
 - Easy to program. “automatic parallelization” of loops.
 - Limited scalability due to memory bottlenecks and race conditions.

Performance evaluations

Computational cost:

- Total execution time on all processors:
 - $t(1)$: execution time (wall-clock) on a single proc.
 - $t_i(Nproc)$: execution time (wall-clock) on processor i out of $Nproc$ processors.
 - Cost: $C_p(Nproc) = Nproc * \text{MAX}_i(t_i(Nproc))$
 - Cost-optimal algorithm: cost to solve the problem on $Nproc$ processors is equal to cost to solve it on a single processor.

Performance evaluations

Speedup factor:

- How many times faster the problem is solved on Nproc processors than on a single processor:
 - Speedup :

$$S(Nproc) = \frac{t(1)}{\text{mean}_j \max_i t_{ij}(Nproc)} \frac{N(Nproc)}{N(1)}$$

- N(1) and N(Nproc) are the problem sizes.
- The maximum speedup is Nproc (linear speedup).
- Super linear speedup, i.e., $S(Nproc) > Nproc$ is an artifact of using a sub-optimal sequential algorithm.

Performance evaluations

Communication overhead:

- Ratio between time spend communicating and time spend computing in a program:
 - Communication overhead = (communication time)/(computation time) .
 - The communication overhead usually grows with increasing Nproc, leading to $S(Nproc) < Nproc$.

Performance evaluations

- **Strong scaling:**
 - Problem size does not change with the number of processors, so the amount of work per processor decreases.
- **Weak scaling:**
 - Problem size increases proportionally with the number of processors, so the amount of work per processor remains constant.
- Strong scaling is harder to achieve because the communication overhead grows.

Performance evaluations

Parallel efficiency:

- Fraction of the time the processors are being used for computation:

$$E(Nproc) = \frac{t(1)}{t_i(Nproc)Nproc} = \frac{S(Nproc)}{Nproc}$$

- The efficiency is between 0 and 1 (or 0% and 100%) with an ideal program achieving 100% efficiency.

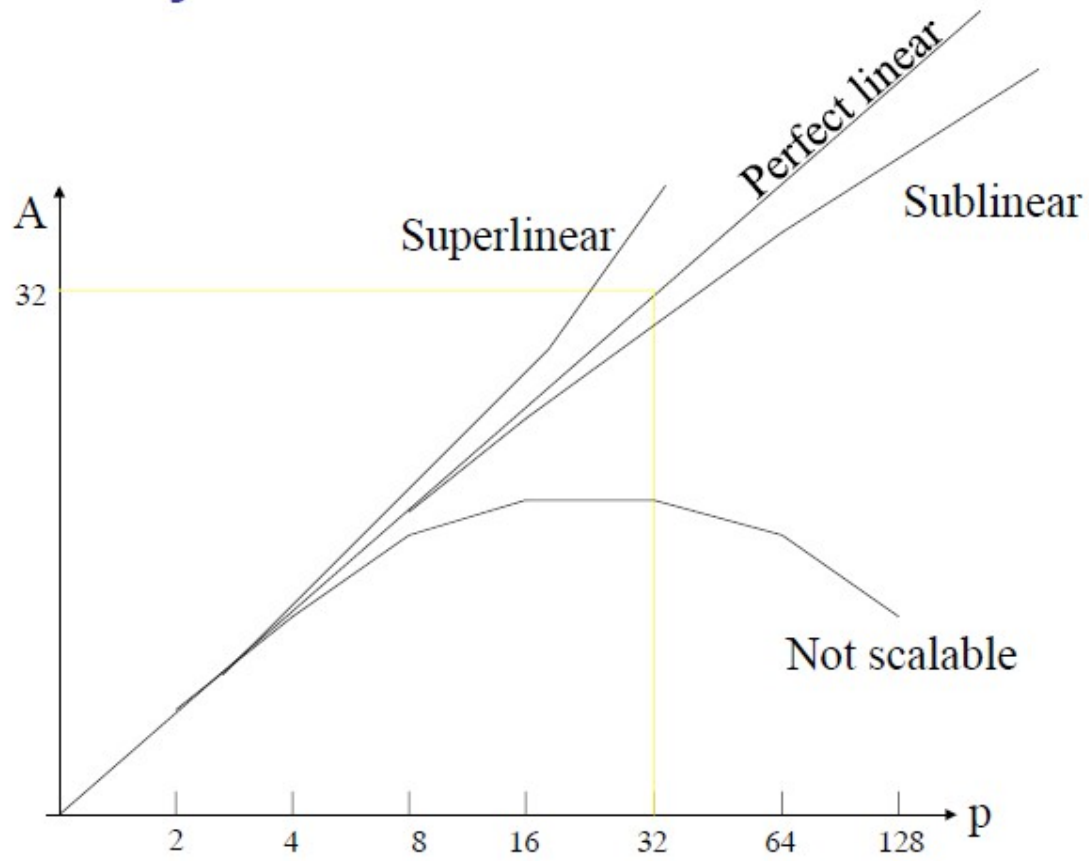
Performance evaluations

Scalability:

- Scalability analysis: estimation of the computational and communication requirements of a particular problem and how they change as the problem or hardware size changes:
- **Hardware scalability:** larger hardware (more processors) leads to proportionally better performance.
- **Software scalability:** increasing the input data leads to a bounded increase in the number of computational steps.

Performance evaluations

Scalability:



Performance evaluations

When does the speedup (A) have a maximum ?

$A = \frac{P}{1 + cP},$ $\frac{dA}{dP} = \frac{P}{(1 + cP)^2},$ <p style="text-align: center;">no max.</p>	$A = \frac{P}{1 + cP \log P},$ $\frac{dA}{dP} = \frac{1 - cP}{(1 + cP \log P)^2},$ <p style="text-align: center;">max. at $P = 1/c$</p>	$A = \frac{P}{1 + cP^{1+\alpha}}$ $\frac{dA}{dP} = \frac{1 - c\alpha P^{1+\alpha}}{(1 + cP^{1+\alpha})^2}$ <p style="text-align: center;">max. at $P^{1+\alpha} = 1/c\alpha$ (for $\alpha > 0$)</p>
--	--	--

- To get a max. in the speedup (A), the communication effort (c) has to grow faster than just with the number of procs. P. This can happen when synchronization operations with communication efforts have to be executed too regularly on a non-scalable network.

Performance evaluations

Interconnect characteristics:

- **Bandwidth:**
 - Number of bits that can be transmitted per unit time (bits/sec)
- **Latency:**
 - Time to send a message of length 0 bits. Software and hardware overhead to open a network connections (ms)
- **Diameter:**
 - Minimum number of hops between two nodes in the network (gives worse-case latency for ideal parallel efficiency, i.e., no waiting for busy processors)
- $\text{Comm. time} = \text{diam} * (\text{latency} + \text{msg_size}/\text{bdwth})$

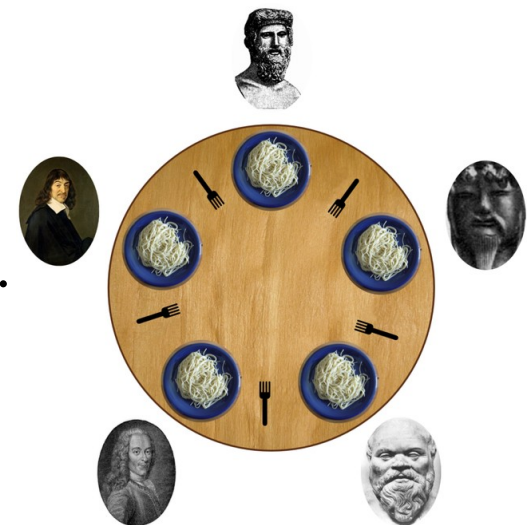
Performance evaluations

Deadlock:

- Inter-process communication is blocked eternally because of circular dependence
 - Example:
 - Node 1 wishes to send to node 2;
 - Node 2 wishes to send to node 3;
 - Node 3 wishes to send to node 1;
 - *Dining philosophers problem* (Dijkstra):
 - Deadlock if everyone holds the right chopstick and waits for the left one.



Prof. Dijkstra



Performance evaluations

Synchronicity:

- Synchronous communication: sender and receiver must execute the communication at the same time. If one is not ready the other one has to wait.
- Asynchronous communications: send and receive can occur at shifted times. The messages are buffered by the communication system and/or notification are sent to the receiver.

Performance evaluations

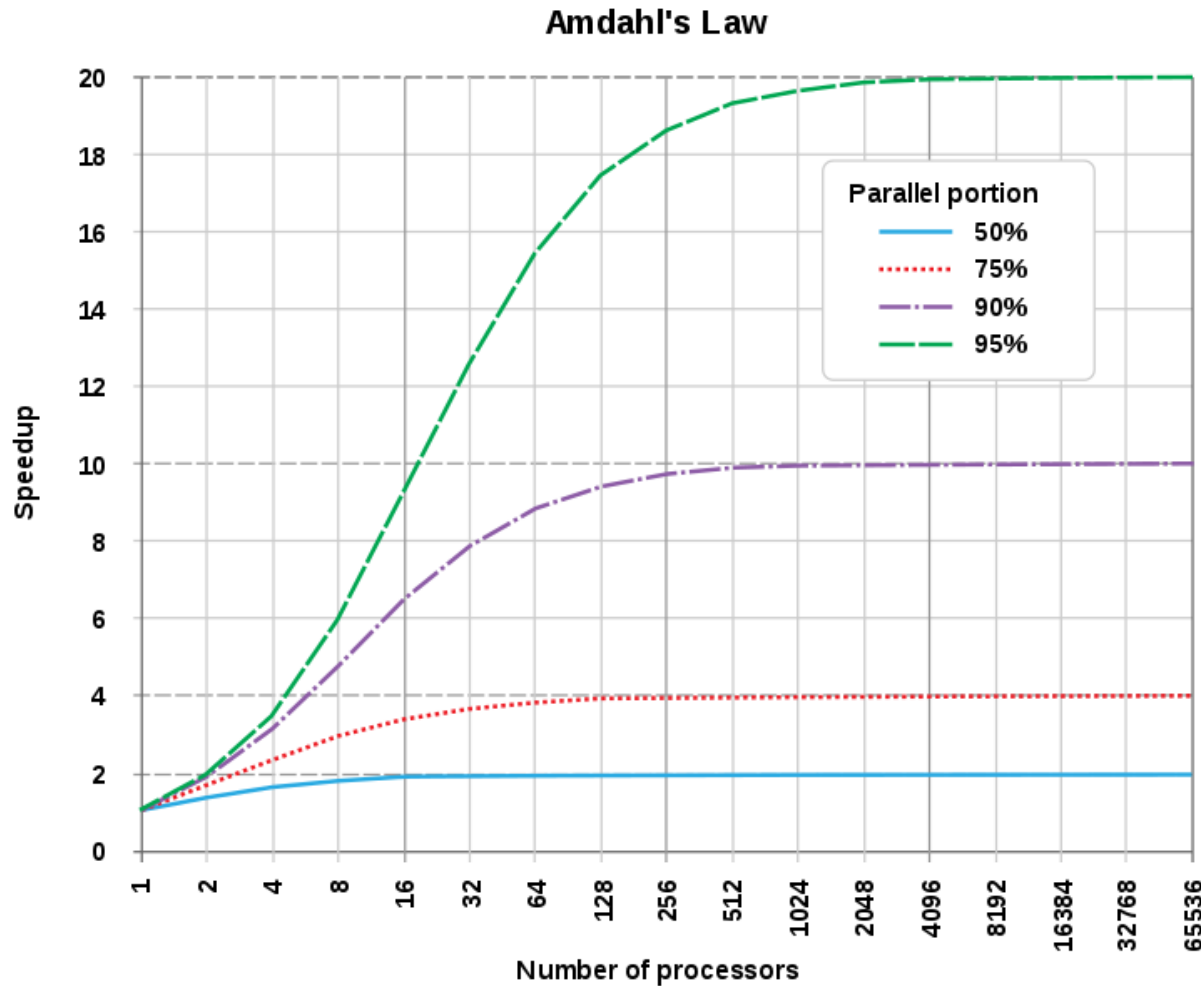
Amdahl's law:

- Upper bound for speedup of a given program (fixed-size problem, strong scaling):
 - f : serial factor, i.e., fraction of the program that cannot be parallelized (t_s = serial time):

$$S_s(N) = \frac{t_s}{ft_s + (1 - f)t_s / N} = \frac{N}{1 + (N - 1)f}$$

- Maximum speed with an infinite number of processors (N) is $1/f$.
If .e.g., 5% of the program is sequential, the speedup can never exceed 20!

Performance evaluations



Ref.: wikipedia

Performance evaluations

Gufstafson's law:

- Sequential part (f) of the program does not scale with the problem size (scaled-size problem, weak scaling):

$$S_s(N) = N - f(N - 1)$$

- The serial fraction decreases with increasing problem size, thus:

$$S_s \rightarrow N \text{ for } N \rightarrow \infty$$

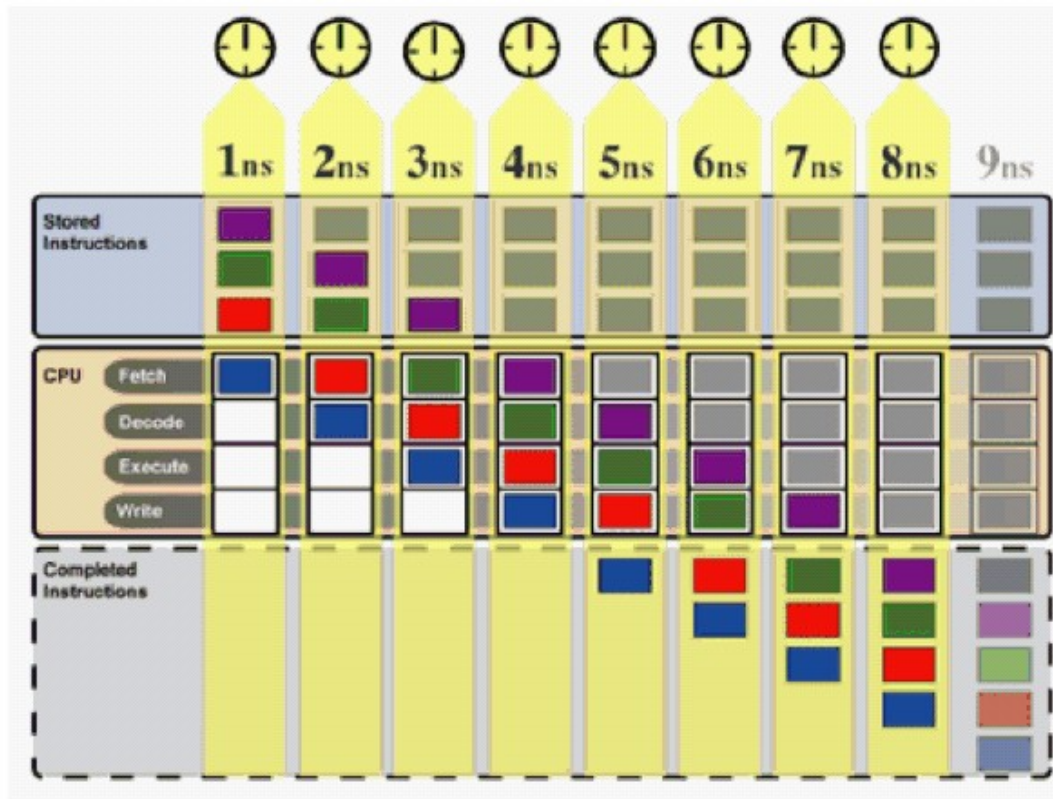
- “Any sufficiently large problem scales well”.

Pipelining

- *Instruction parallelism* with pipelining:
 - Overlapping multiple instructions in execution in a single processor.
 - Each instruction consists of:
 - Fetch.
 - Decode.
 - Execute.
 - Write.
 - Pipelining does not speed up the execution time for a single instruction, but speeds up program execution.

Pipelining

- Example: 4-stage pipeline

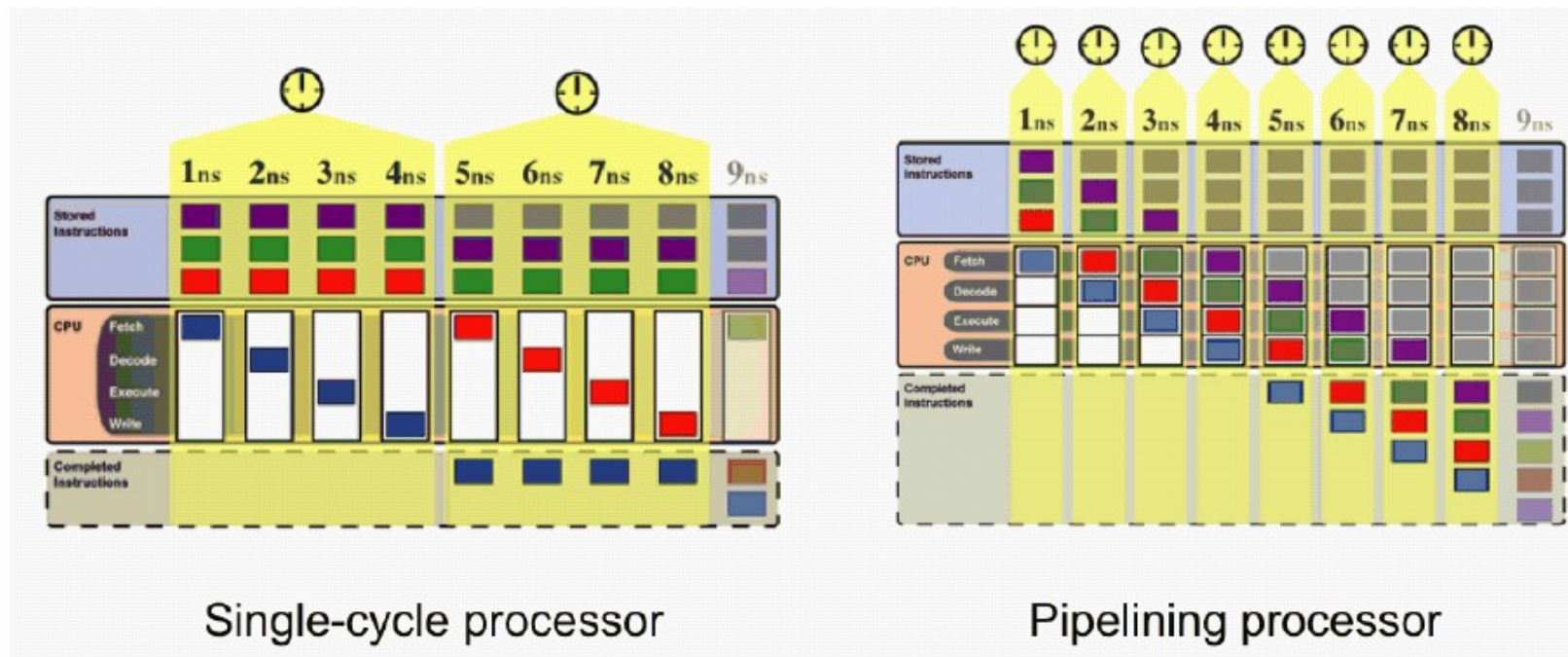


4-stage pipeline

Fetch instruction 2
While decoding
instruction 1, etc...

Pipelining

- Comparison to single-cycle processor



Single-cycle processor: 16ns to execute 4 instructions.

4-stage pipeline: 8ns (4ns for the first element, then the subsequent elements will appear every 1ns)

Pipelining

- Pipeline parallelism: history:
 - 1981: 4.77 MHz 8088 in IBM PC.
 - 1989: 33 MHz 80486.
 - 1994: 133 MHz Pentium.
 - 1998: 200 MHz Pentium PRO.
 - 2004: 3 GHz Pentium IV (20-stage pipeline)
 - 1000-fold increase in clock rate in 25 years.
 - 100s of instructions in processing at once.
 - Pipelining as main source of performance gain in last decade.
 - “Superscalar architecture”.

Pipelining

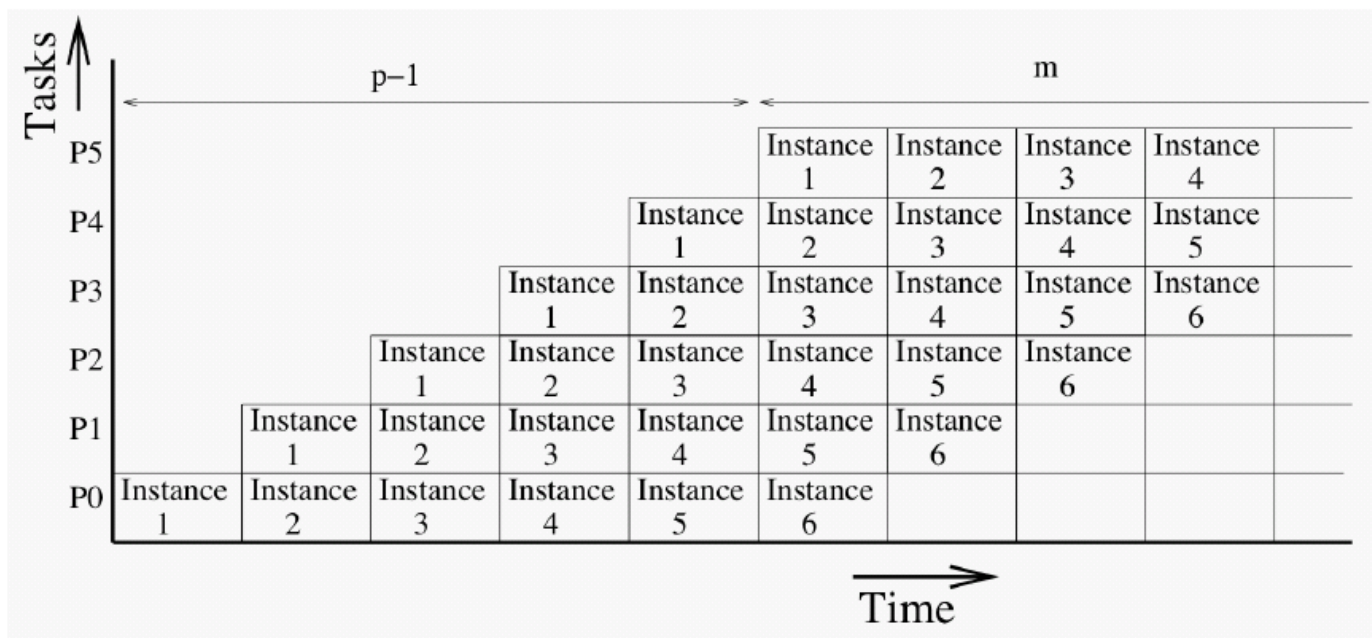
- Prerequisites and issues:
 - Memory must operate in one cycle (caches!).
 - Instructions within the pipeline must not depend on each other's results (otherwise the pipeline “stalls”).
 - Memory read/write must be coordinated in order not to overwrite new results with old ones.
 - Branches cannot be predicted and the pipeline stalls.
 - All instructions must fit into the common pipeline state structure.

Pipelining

- Data parallelism with pipelines (SIMD):
 - Execute same instruction (sequence) over a vector.
 - Similar to an assembly belt.
 - Assume each element has to be multiplied with 2, then add 3 and sqrt taken. Stream the data through the pipeline, each stage does one of the 3 operations on a piece of data concurrently passing through.
 - One complete result exits in each clock cycle once the pipeline is filled.

Pipelining

- Upper bound for pipeline speedup
 - Pipeline of length p has a fill-time (pipeline latency) of $p-1$ cycles:



Pipelining

- Upper bound for pipeline speedup (cont.)
 - Pipeline length p is upper bound.
 - Latency is $p-1$.
 - One work unit finishes per clock cycle.
 - So for N work units the speedup is max.:

$$S = \frac{Np}{(p - 1) + N}$$

Vectorization

What is vectorization ?

- Execution of an operation on a whole vector of data simultaneously.
- SIMD parallelism.
- Can be implemented by vector registers and vector ALUs (arithmetic logic unit) or by data pipelines:

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix} \quad \longrightarrow \quad C(1:5) = A(1:5) + B(1:5)$$

Vectorization

Where is it found ?

- Co-processors:
 - SSE, SSE2, SSE3, SSE4, AltiVec, 3DNow!
- Traditional supercomputers:
 - Cray YMP, NEC SX series.
- Digital signal processors:
 - GPUs, Cell BE.
- Game consoles:
 - X-Box, Wii, PlayStation.
- ARM X64FX Fujitsu (top500 2020, 2021).

Vectorization

Why does it work ?


- Exploit regular structure of computation to achieve data pipelining.
- Hardware acceleration for loops.
- Reduces the semantic gap between hardware and software.
- Allows more aggressive compiler optimization (regular data structure guaranteed).
- Important to eliminate branch mis-predictions penalty.

Vectorization

Prerequisite: Data independence


- Loop-carried dependence: dependence between instructions from different iterations of a loop:

```
DO I=1,N  
  A(I+1) = A(I)*B(I)  
ENDDO
```



Cyclic dependence

```
DO I=1,N  
  D(I) = A(I-1)*D(I)  
  A(I) = B(I)+C(I)  
ENDDO
```



Backward dependence

Vectorization

Loop vectorization

- Convert a single-statement loop into a vector operation.

```
DO I=1,N  
  A(I) = B(I)+C(I)  
ENDDO
```



```
A(1:N)=B(1:N)+C(1:N)
```


- Only inner-most loop is vectorized.

Vectorization

Problem: backward dependence

- Multi-statement loops can not be vectorized if there is a (backward) data dependence:

```
DO I=1,N
  D(I) = A(I-1)*D(I)      ! S1
  A(I) = B(I)+C(I)        ! S2
ENDDO
```



- S2 of iteration i delivers result to S1 of iteration $i+1$.
- Backward dependence: loop carried dependence on a syntactically preceding statement.

Vectorization

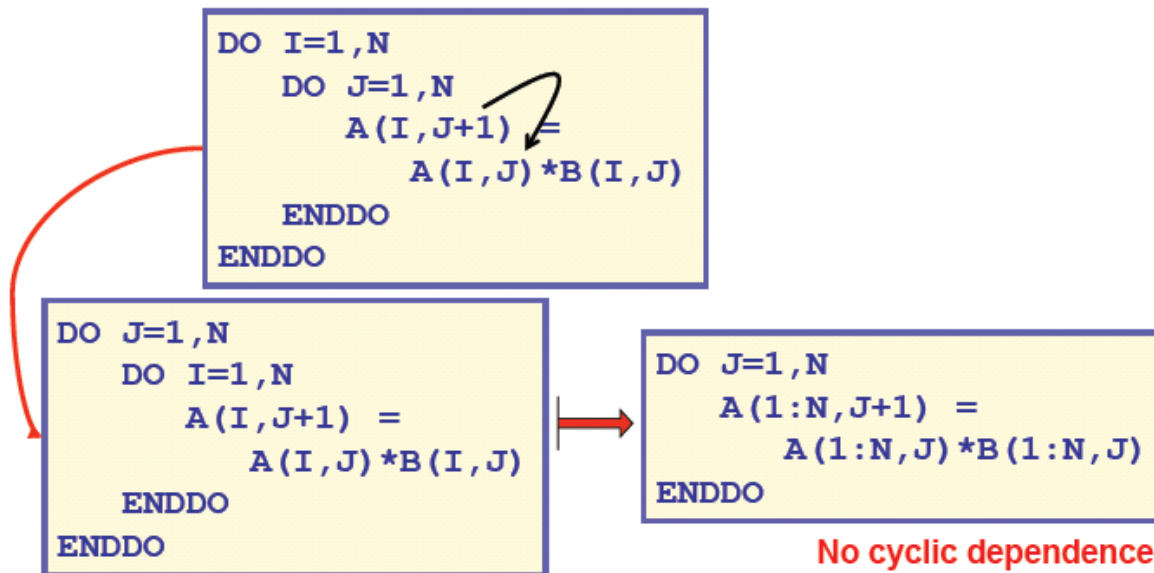
Problem: backward dependence (cont.)

- Result would be invalid if loop was distributed and S1 and S2 vectorized.
- Only loops without backward dependence can be distributed.
- Solution: statement re-ordering (if possible).

Vectorization

Loop interchange:

- Some cyclic dependence can be resolved by interchanging inner and outer loops:



Vectorization

Loop sectioning:

- The length of the vector registers in general does not match the number of loop iterations.
- Loops are broken (sectioned) into several loops, whose length corresponds to the length of the vector register, and a remainder loop.
- The longer the remainder loop the better the vectorization performance (longer vectors).

Vectorization

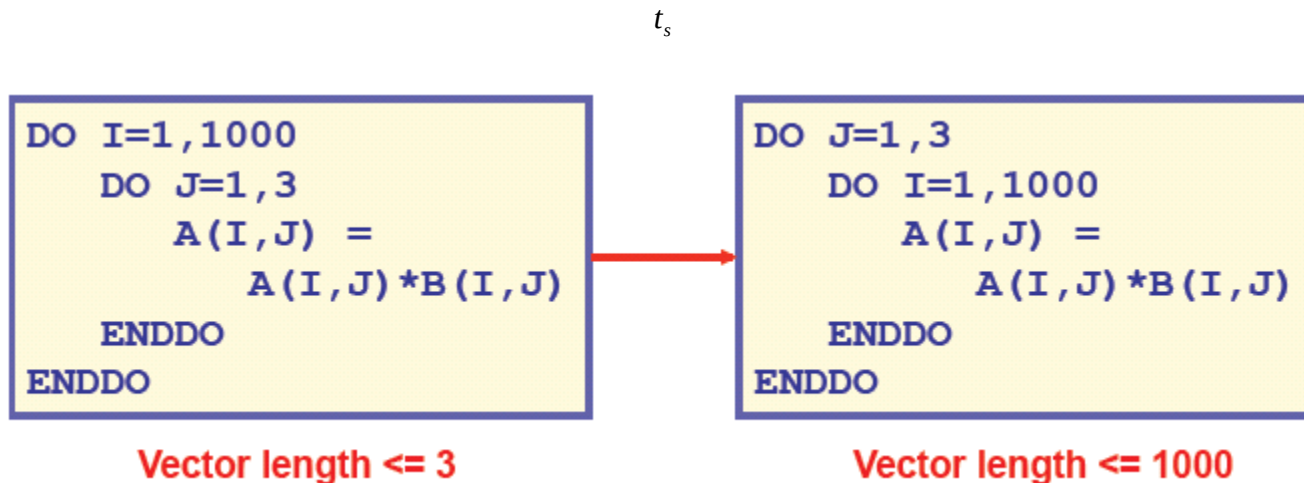
Vector length:

- Each operation in the vector ALU (arithmetic logic unit) takes the same amount of time, regardless of how full the vector registers are.
- Longer vectors (ideally completely filling the vector registers all the time) lead to better performance.
- Performance tools report average vector length and % of vectorized loop iteration at run time.

Vectorization

Vector length (cont.):

- Only the innermost loop can be vectorized.
- Swap loops to have the longest one vectorized (larger vector length).



Vectorization

Loop unrolling:

- Faster execution even on scalar machines.
- Less loop overhead.
- Longer vectors.

t_s

```
DO I=1,1000
  DO J=1,3
    A(J,I) =
      A(J,I)*B(J,I)
  ENDDO
ENDDO
```



```
DO I=1,1000
  A(1,I) = A(1,I)*B(1,I)
  A(2,I) = A(2,I)*B(2,I)
  A(3,I) = A(3,I)*B(3,I)
ENDDO
```

Vectorization

Fortran vector notation:

- Fortran supports explicit vector.
- But it is MUCH slower than loops! (semantic gap to hardware)
- DO NOT USE IT!

