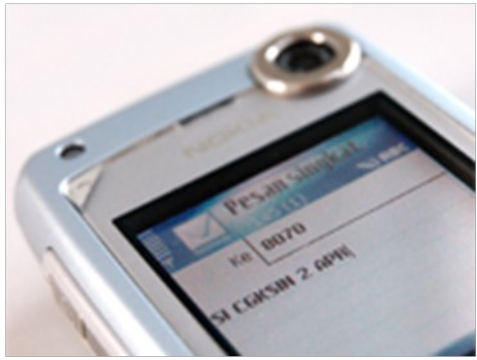


Computer Architectures

Multiprocessor Systems

Gábor Horváth, ghorvath@hit.bme.hu

2023. máj. 31.
Budapest

Forms of parallelism

- Implicit parallelism
  - The CPU tries to exploit parallelism without collaborating with the programmer
  - The programmer writes sequential algorithms/program
  - The program is not adjusted to the hardware specifics
  - Implementations:
    - Overlapped instruction execution → instruction pipeline
    - Multiple functional units → superscalar, VLIW CPUs

Forms of parallelism

- Explicit parallelism
  - The problems of implicit parallelism:
    - Data dependencies
    - Why do they decrease the efficiency?
    - Because the programmer knows nothing about the parallel execution capabilities of the CPU (or the programming language is not capable of expressing the parallelism)
  - Solution:
    - Let the programmer tell which parts of the program can be executed in parallel!
    - Control token multiplication and joining (FORK/JOIN)

Forms of parallelism

- Hardware support for explicit parallelism:
  - Multi-threading in CPUs
  - Multiprocessor systems

Multi-threaded CPUs

CPUs supporting multiple threads

- What are the root causes of reduced efficiency of the instruction pipeline?
  - Cache misses
  - TLB misses
  - Data dependencies
- What happens in these cases?
  - Nothing

    → A pipeline stall is required
- Remedy:
  - Let us introduce additional program counters
  - Multiple instruction sequences can be executed, but not at the same time
  - If the execution of one of the instruction sequences needs to stop and wait, let us take the other one and execute it till the issue is resolved

    → The CPU remains utilized during the stalls
  - This is how multi-threaded CPUs work

CPUs supporting multiple threads

- What does a „thread” mean?
- We are talking about multi-threaded CPUs, and not multi-task CPUs
- What is the difference??

Multi-thread vs. multi-task

- ***Multi-tasking***
  - = executing several programs „at the same time”
  - Professional operating systems support it for 50 years
  - It can be done in single-processor systems as well: instead of real parallel execution, the CPU makes the user believe that it is capable of executing several programs in parallel → ***by using time-shaing***
    - Ingredients:
      - A timer that generates interrupts according to a given frequency
      - An interrupt handler subroutine, that performs the ***task switching***
        1) Saves the state of the current task (current state of the registers, stack pointer, etc.)
        2) Selects another task that has been interrupted earlier
        3) Restores its state (loads the values of the registers according to the state when the task has been interrupted)
        4) Restores its program counter → from this point on the CPU executes the new task

Multi-thread vs. multi-task

- There are some more things to do when switching tasks:
  - Switching page tables (each task has its own page table)
  - The TLB has to be flushed
  - Maybe the instruction cache has to be flushed as well
- These take a while themselves, followed by a burst of TLB and cache misses till the system stabilizes

  → ***Switching tasks costs a lot of time!***
- Up to 2-3 thousands of clock cycles (Intel Pentium4)

Multi-thread vs. multi-task

- ***Multi-thread:***
  - Usually a program (task) consists of parts that can be executed in parallel, called ***thread***
  - Because the programmer used the FORK/JOIN primitives
  - Properties of the threads:
    - They share the memory of the task (even its local variables)
    - Threads have their own program counters and register sets
    - In a time sharing system the operating system sometimes switches between tasks, sometimes between threads belonging to the same task. In the latter case:
      - Page table remains the same → no TLB and cache flush
      - Only the registers are saved and restored

      → ***Switching between threads belonging to the same task is fast!***

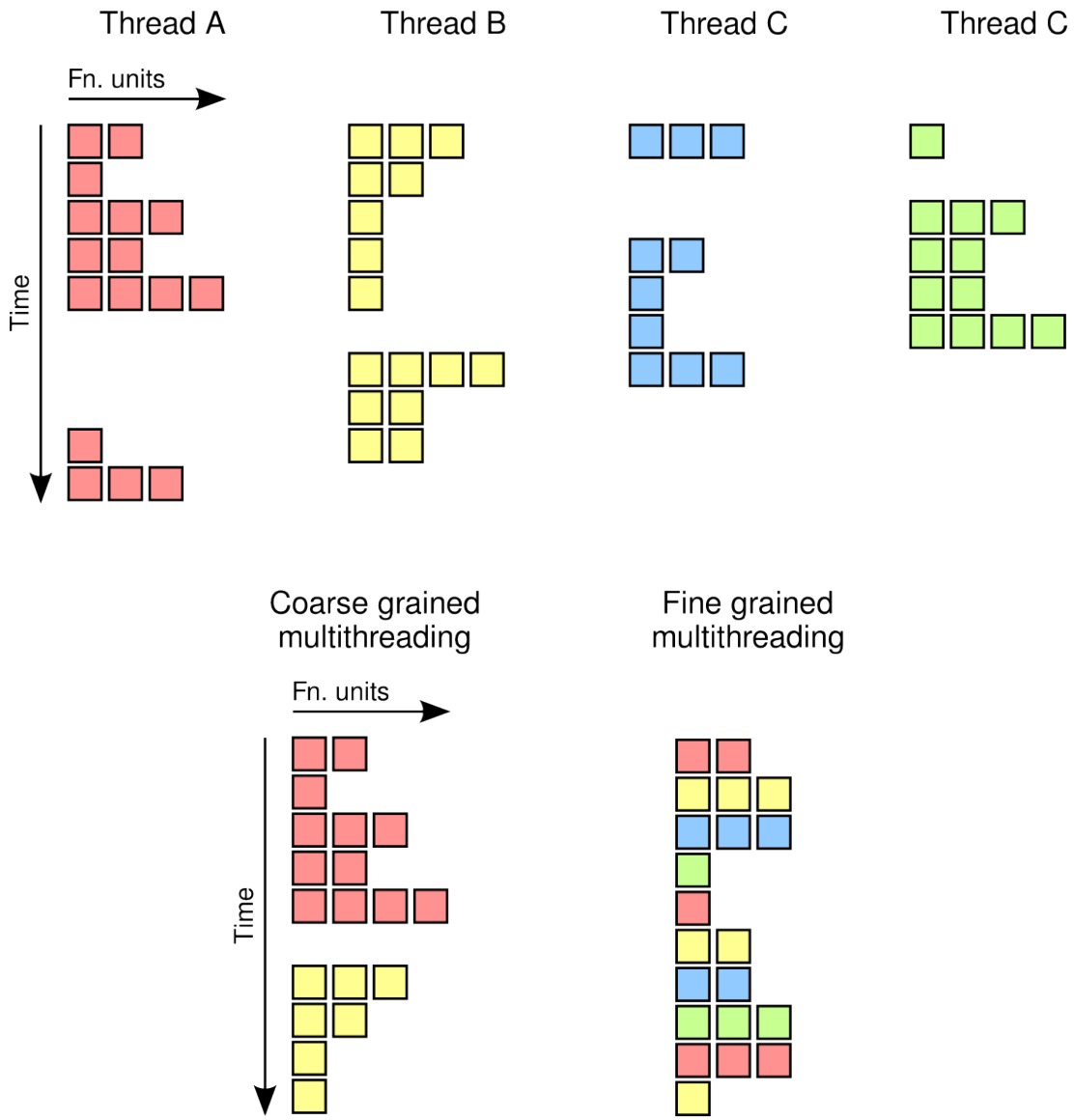
Multi-thread vs. multi-task

- Conclusion:
  - Time sharing is used by single processor systems to make the illusion of parallel execution
  - Switching between tasks is very slow
  - Switching between threads of the same task is very fast
- What are multi-threaded CPUs doing?
  - Surprise: they are in fact ***multi-tasking*** CPUs!
  - Confusing naming convention
  - We use the convention of computer architectures in the sequel: multi-threading, threads (in fact it is multi-tasking, tasks)

Time-sharing based multi-threading

- Some multi-threaded CPUs support
  - ***Fine-grained multi-threading***
    - Switch thread at every clock cycle
    - When the execution returns to a given thread, its problems are already solved with a high probability: data dependency, cache miss, TLB miss are already resolved, thus no pipeline stall is needed
  - ***Coarse-grained multi-threading***:
    - Thread switching is performed only if the current thread needs to stop
- Hardware implementation:
  - Thread switching latency: 0 or 1 clock cycles (fine-/coarse case)
  - A thread ID is assigned to the instructions after being fetched
  - The CPU has multiple program counters and register files (one for each thread)
  - TLB, cache, etc. are shared
    - Their entries have an additional field: „thread ID” (to which thread does the entry belong)
  - Every thread works on the register file, TLB and cache entries determined by its thread ID

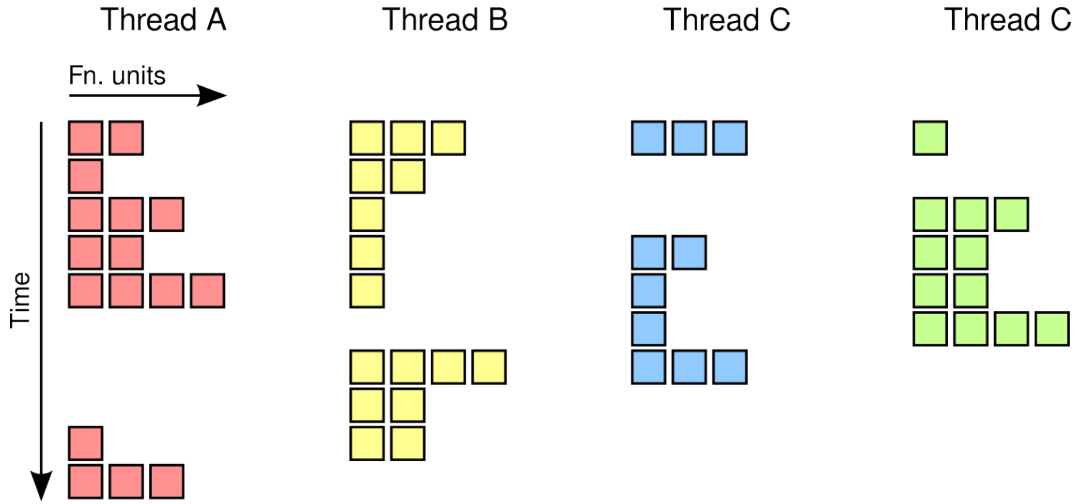
Time-sharing based multithreading



Simultaneous multi-threading

- An alternative to time-sharing based multi-threading
- It works only with superscalar CPUs
- In a superscalar CPU it is typical to have a lot of unutilized functional units
  - When the CPU was not able to find enough number of instructions in the program that can be executed in parallel
- Why not to use the unutilized functional units to execute instructions belonging to an other thread?

Simultaneous multithreading



Simultaneous multithreading

Multi-threading processors

- Relatively easy to implement

Processor	Year	Type of multi-threading	Number of threads supported
Intel Pentium 4	2002	Simultaneous	2
Intel Itanium 2	2006	Coarse-grained	2
IBM POWER5	2004	Simultaneous	2
IBM POWER6	2010	Simultaneous	4
UltraSPARC T1	2005	Fine-grained	4
UltraSPARC T2	2007	Fine-grained	8

- Note: in Intel terminology, simultaneous multi-threading is called as Hyper-Threading

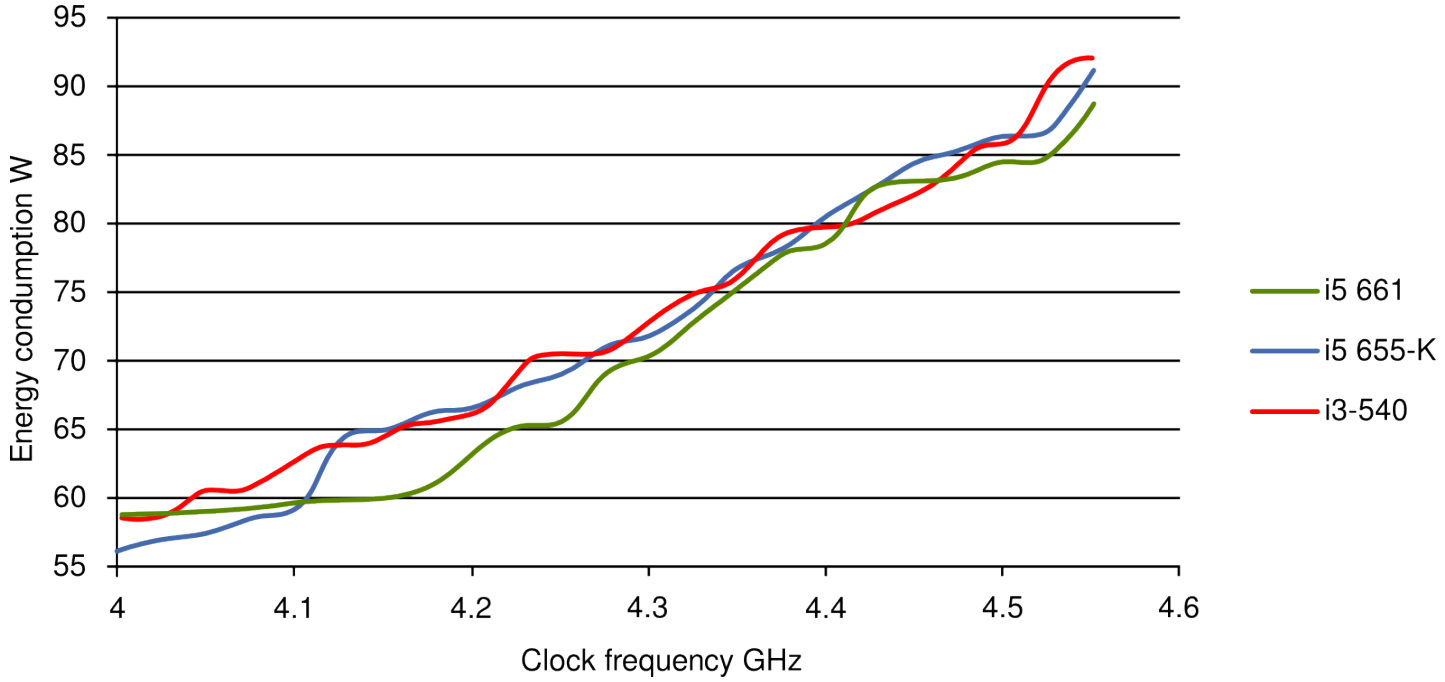
Multi-processor systems

Multi-processor systems

- Multiprocessor systems consist of multiple, autonomous, fully functional processors
- Outline:
  - Why do we need multi-processor systems?
  - How much do we gain by using them?
  - How to connect those processors?

Motivation

- Instruction level parallelism has its limits. Explicit parallelism can be more efficient.
- Energy-efficiency: the energy consumption increases drastically with the computational power (assuming identical manufacturing technology):



Motivation

- Cost efficiency: the cost of the processors grow exponentially with the computational power
  - It is cheaper to buy several slow CPUs than buying a single fast one
- Easy to extend:
  - If even more computational power is required, it is easy just to add some more CPUs to the multi-processor system
- Reliability:
  - If a CPU breaks down, the remaining CPUs can take its tasks and the system remains fully functional

Limitations

- There is a serious limitation. It is the ***software***
- Who writes parallel programs?
  - ***The compiler***:
    - The programmer writes sequential programs
    - The compiler detects the parts that can be executed in parallel
    - No such sophisticated compiler exists for the most popular programming languages (yet)
  - ***The programmer***:
    - It is difficult
    - Lots of potential mistakes can be introduced
    - This is the dominating approach today

The maximum expected improvement

- If we buy N processors, will my programs run N times as fast?

- No. Only if the program is perfectly parallelizable.
- In the general case, it is the Amdahl's law that specifies the speed improvement given by N processors

The maximum expected improvement

- Assume
  - „P” portion of the program is perfectly parallelizable
  - „1-P” portion of the program needs sequential execution
- Assume the execution speed in a single processor system is 1
- Question: what is the execution time by using *N* processors?
  - If the whole program is sequential: 1
  - If the whole program is perfectly parallelizable: 1/N
  - If „P” portion is parallelizable: (1-P)*1 + P/N
- ***Amdahl's law: the speedup using N processors compared to the single-processor system is:***

$$S_P(N) = \frac{1}{(1-P) + P/N}$$

The maximum expected improvement

- Example: we have 100 CPUs. We would like to execute our program 80 times faster than in a single CPU system. What is the maximal amount of sequential code we can put in the program?

$$P = \frac{S_P(N) - 1}{S_P(N)} \frac{N}{N-1} \simeq 0.9975$$

- Only 0.25% of the program can be sequential!
- If 5% of the program is sequential (*P=0.95*):

$$S_P(100) = \frac{1}{(1-0.95) + 0.95/100} = \frac{1}{0.05 + 0.0095} = 16.8$$

- The speedup in only 16.8! With 100 processors!

The maximum expected improvement

- Amdahl' law again:

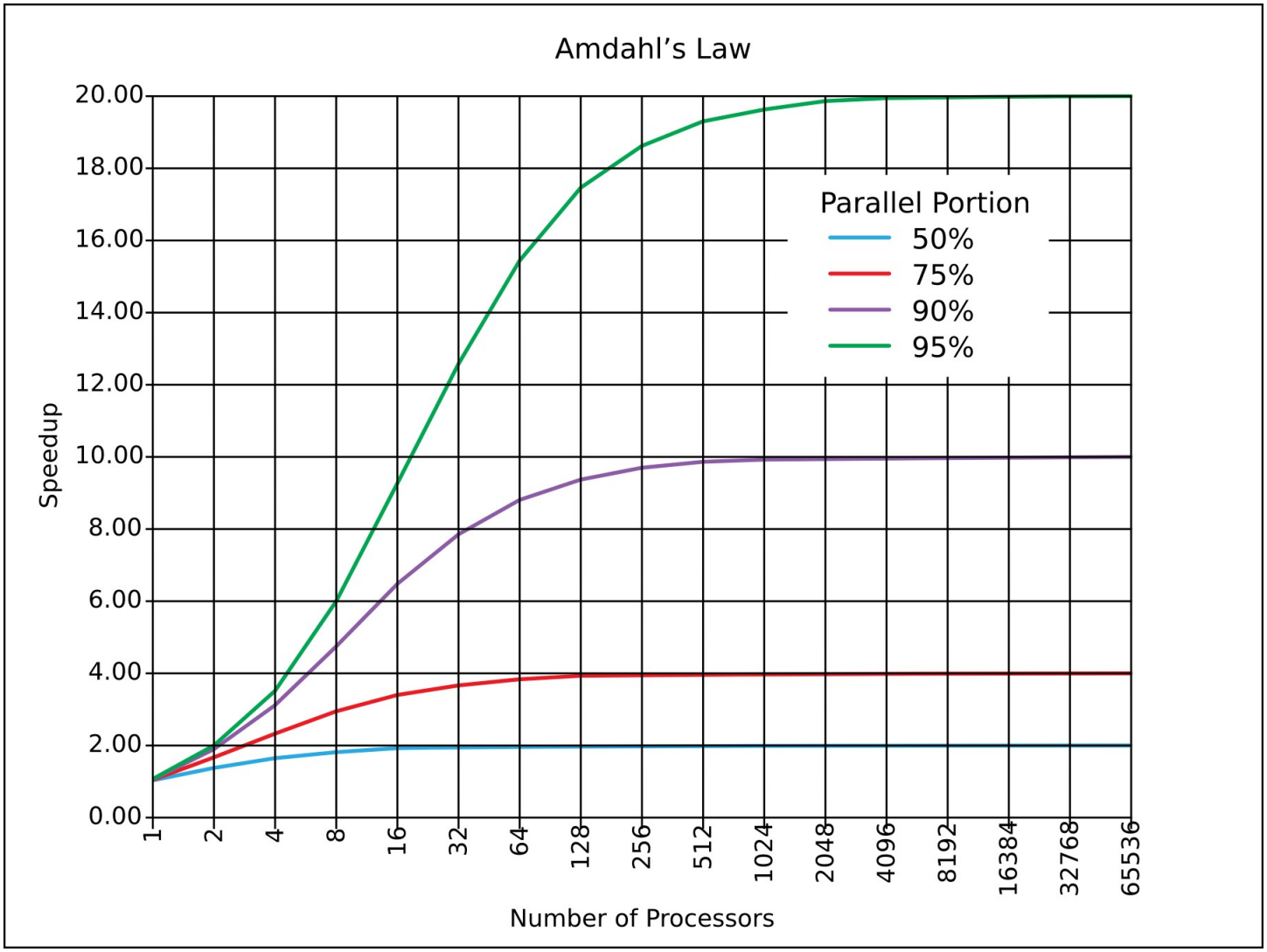
$$S_P(N) = \frac{1}{(1-P) + P/N}$$

- Assuming infinitely many processors we have:

$$S_P(\infty) = \lim_{N \to \infty} S_P(N) = \frac{1}{1-P}$$

- This is the maximum amount of speedup that can be achieved with ***any*** number of processors
  - The limitation is given by the sequential part of the program

The maximum expected improvement



A taxonomy of multi-processor systems

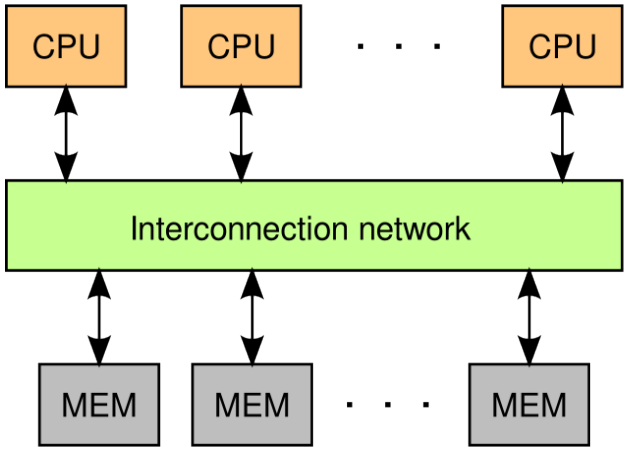
Taxonomy

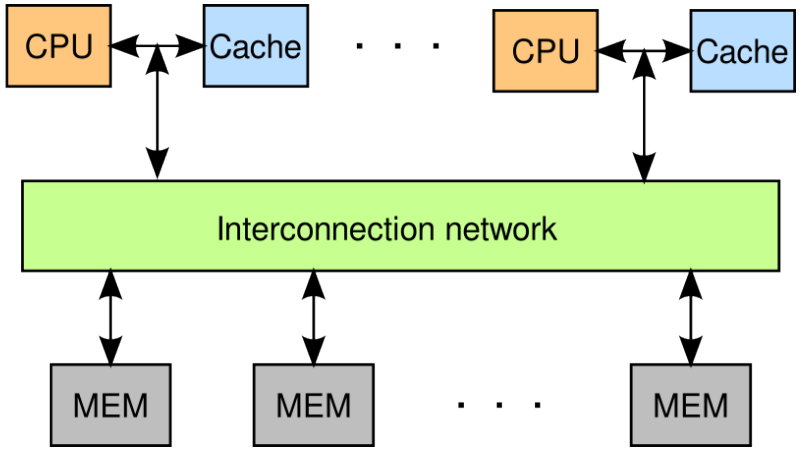
- According to the way the tasks running on different processors ***communicate***:
  - Shared-memory based multi-processor systems
    - Communication: one task writes the data to the shared memory, the other task is able to read it out
  - Message passing based multi-processor systems
    - Communication: tasks send message to each other on a communication network
- According to the speed of the ***memory operations***:
  - UMA (Uniform Memory Access)
    - Every memory operation takes the same time
    - The location of a data unit is irrelevant: each processor needs the same amount of time to read and write it
  - NUMA (Non-Uniform Memory Access)
    - It is important to put the data to the right place
    - It is important to put frequently used data closer to the processor

Shared memory multi-processor systems

- Communication between the tasks is simple

  UMA case:



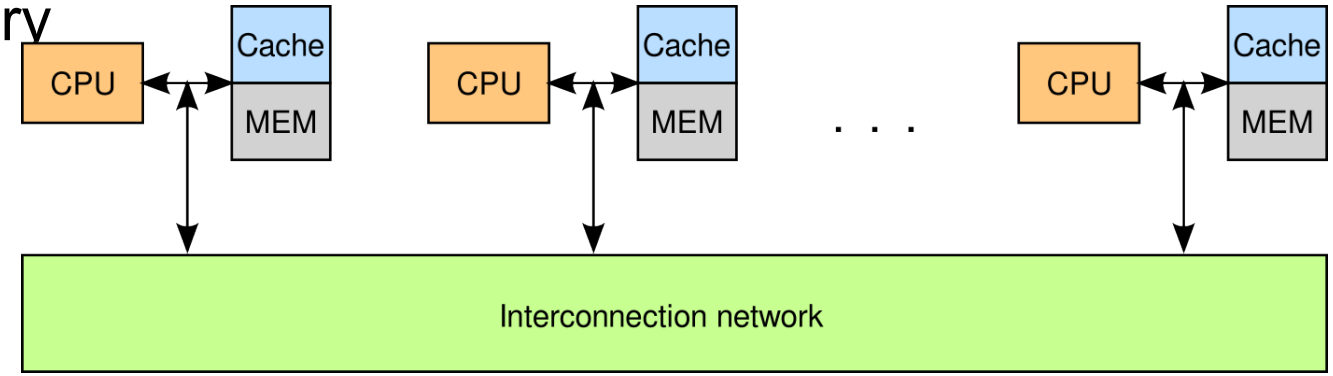


- Easy to implement
- Does not scale well:
  - Adding a new CPU increases the load of the interconnect
- Practical limit: max. 100 CPUs

Shared memory multi-processor systems

NUMA case:

- Nodes of the system consists of a CPU, a Cache, and a Memory



- The address space is shared
  - Everybody has fast access to its own part of the address space
  - If the CPU wants to access an address belonging to a different CPU, the data is transmitted through the interconnect → slower
  - Only the inter-task communication is carried on the interconnect → scaleable!

Message-passing multi-processor systems

- Nodes: complete computers, address space is not shared
- Communication: with messages
  - Explicit message passing primitives:
    - The sender sends a message (calls „send” primitive)
    - The target needs to be prepared to receive it (calls the „receive” primitive)
  - Every task has its own identifier
  - Content of the messages:
    - Payload (the data to send)
    - Sender identifier
    - The identifier of the target
  - The interconnect is responsible to transmit the message to the target
- Message passing can be done
  - Synchronously: The source and the target are waiting till the message arrives to the destination
  - Asynchronously: The sender does not wait till the arrival of the message

Interconnection networks

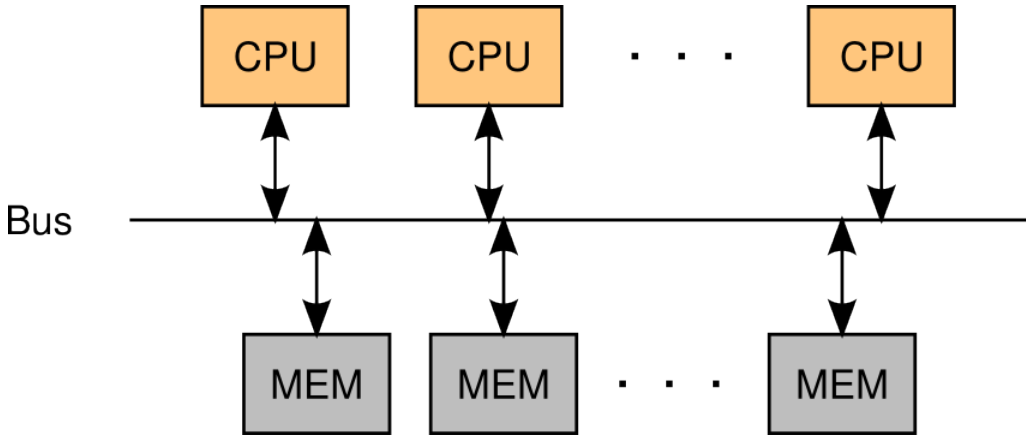
Interconnection networks

- Shared-memory (UMA / NUMA) / message-passing:
  The components must be connected somehow
- What kind of interconnections do we need?
  - Shared-memory UMA:
    - To connect the CPU and the memory
  - Shared-memory NUMA:
    - To connect the nodes (consisting of CPU, cache and local memory)
  - Message-passing systems:
    - To connect complete computers
- What is transmitted by the interconnection networks?
  - Shared-memory systems: memory requests / replies
  - Message passing systems: messages

Properties

- Direct interconnection networks:
  - Connect the nodes (processor or memory modules) directly with each other
- Indirect interconnection networks:
  - The components of the multi processor systems are not connected to each other in a direct way
  - They are connected to a component belonging to the interconnection network

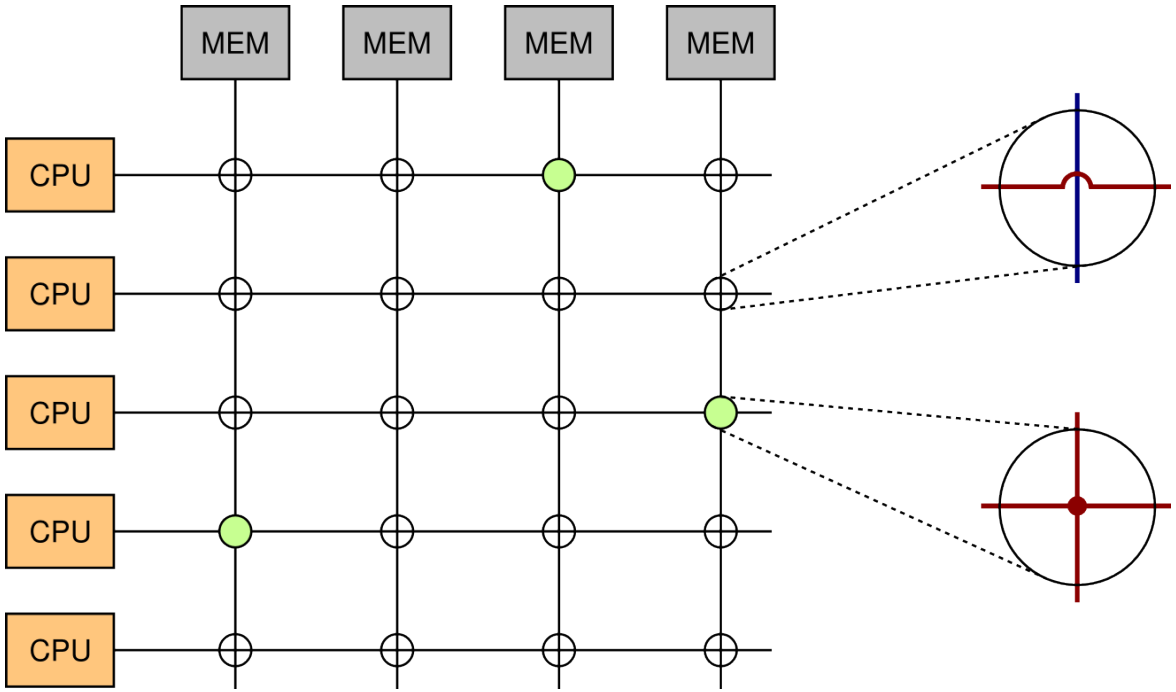
Buses



- Indirect
- Advantages:
  - Not only point-to-point, but also broadcast communication is supported
  - The distance between all the components of the system is the same (in terms of latency)
  - The cost of the network scales well: the cost does not change when a new component is added
- Drawbacks:
  - The throughput does not scale well
    - It has a fixed transmission capacity, which is shared by the components of the system
- Used by Sun Enterprise servers (1996-2001), Intel Pentium based systems

Crossbar

- Indirect
- To connect P processors with M memory modules: P*M switches are required

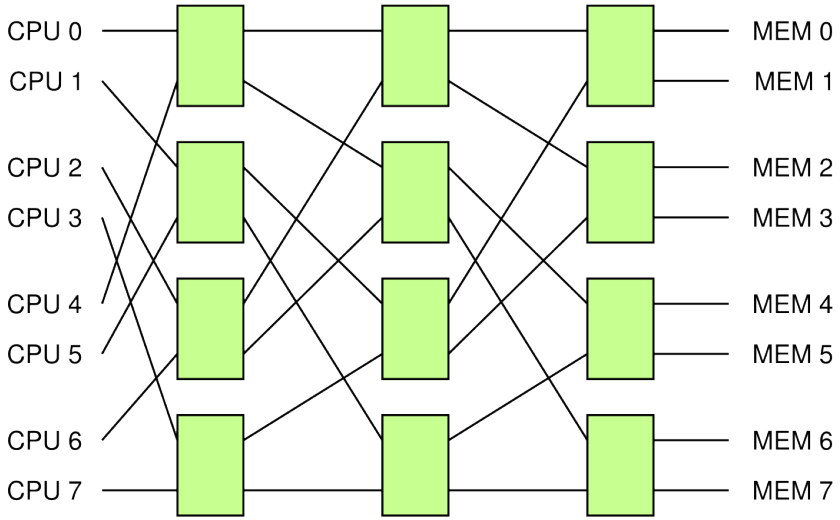


- Several components can communicate with each other at full speed if there is no blocking
- Adding a new CPU: 1 additional wire and M additional switches are needed (expensive)
- Used by IBM POWER4, POWER5, Sun Niagara

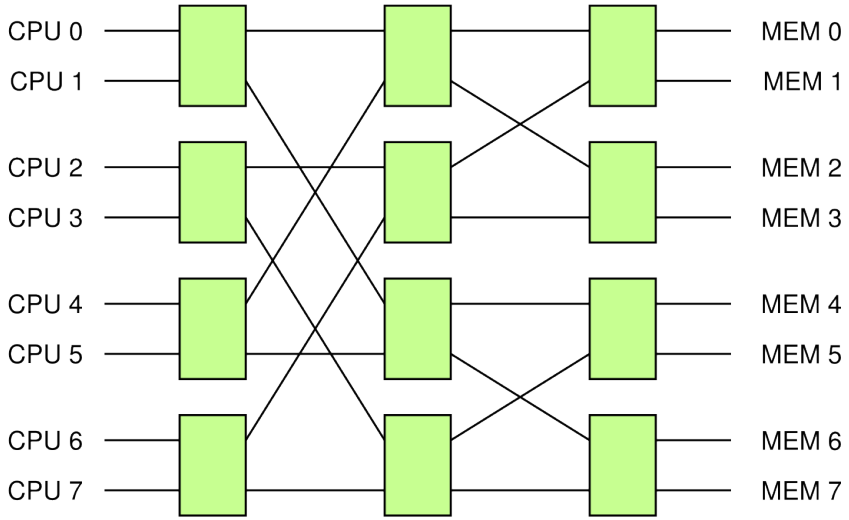
Multistage switching networks

- Indirect
- Composed by small (2x2) crossbars
- Scales well
- Several possibilities to connect the small crossbars:

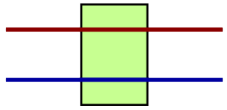
Omega network



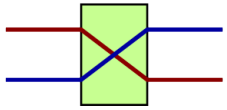
Butterfly network



"Straight" setting:

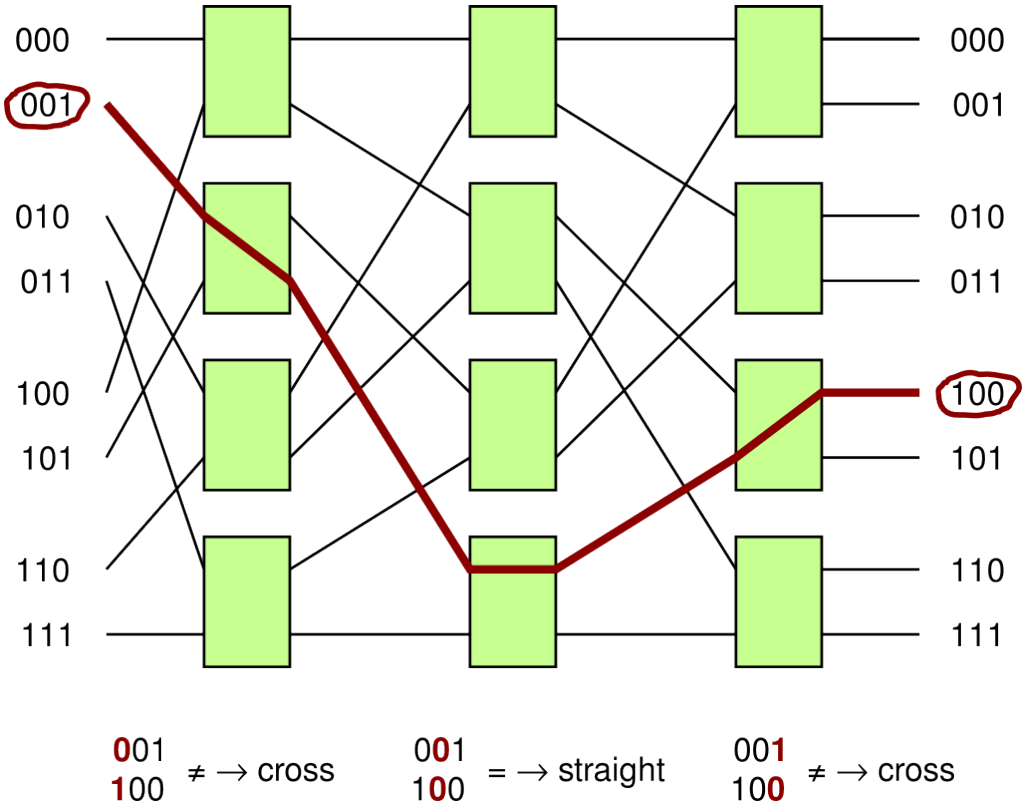


"Cross" setting:



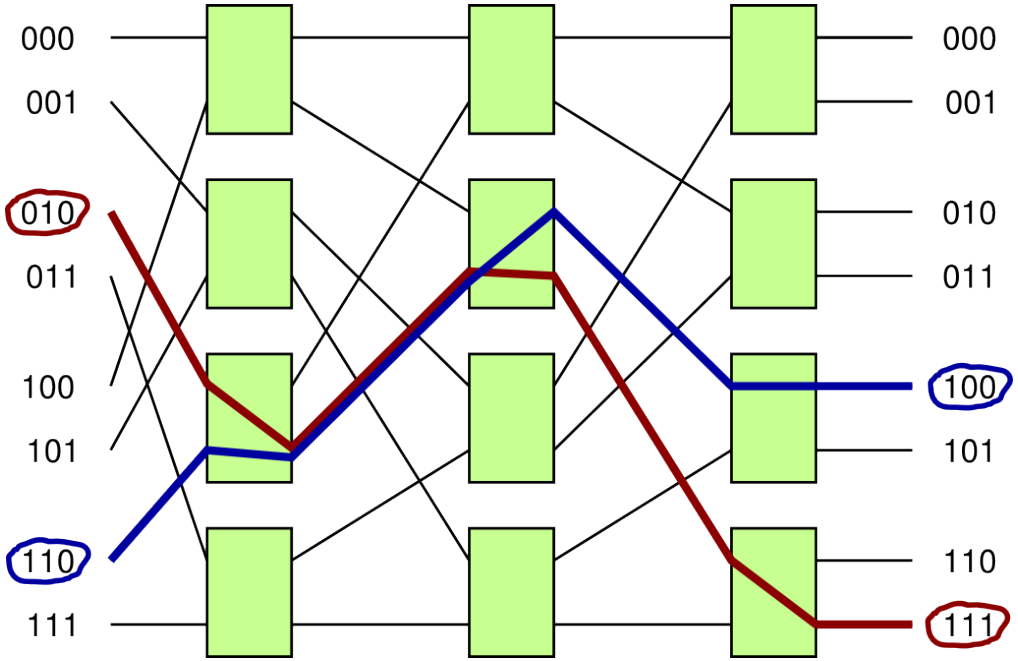
Multistage switching networks

- Example: Omega network
  - Routing in the i-th stage: the switch compares the i-th bits of the sender ID and the target ID



Multistage switching networks

- Not non-blocking:

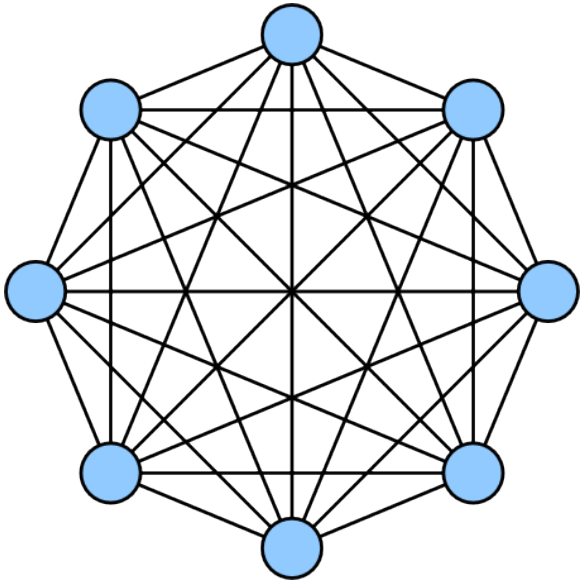


Multistage switching networks

- Number of switches needed to connect *P* components:
  - With multistage switching network: $P*\log_2(P)$
  - With simple crossbar: P*P
- Example: IBM RP3

Complete graph

- Direct interconnection network



- Does not scale well in terms of cost:

  1 additional component → has to be connected to all other components

- It scales well in terms of throughput:

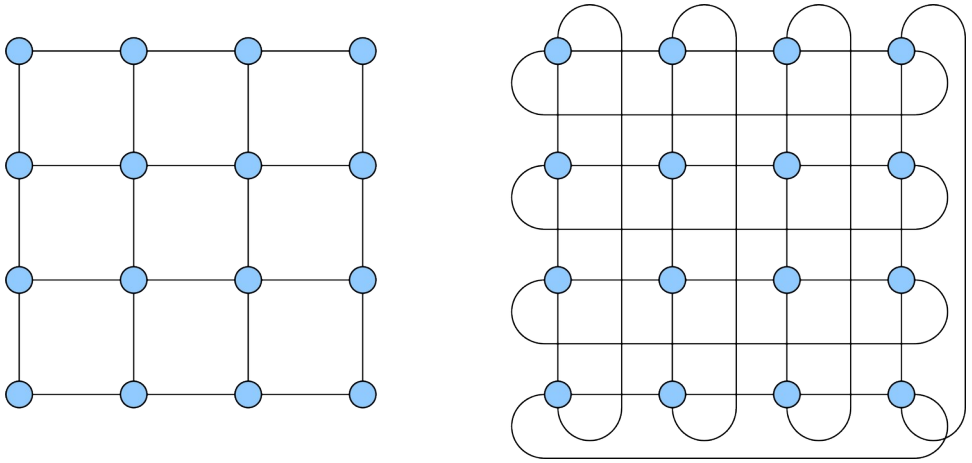
  1 additional component → does not increase the load of the existing interconnections

Linear array and ring networks

- Direct interconnections

- Scales well in terms of cost:
  - 1 additional component → requires only a single new connection
- Does not scale well in terms of throughput:
  - The additional traffic of the additional component increases the load of all the connections

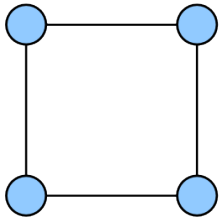
Two-dimensional mesh and torus



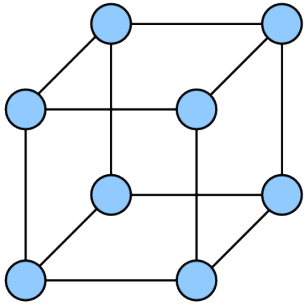
- Direct interconnections
- It scales quite well in terms of cost and throughput as well: good compromise
- Examples:
  - Cray T3E (3D mesh, for weather forecast)
  - IBM Blue Gene/L

Hypercube

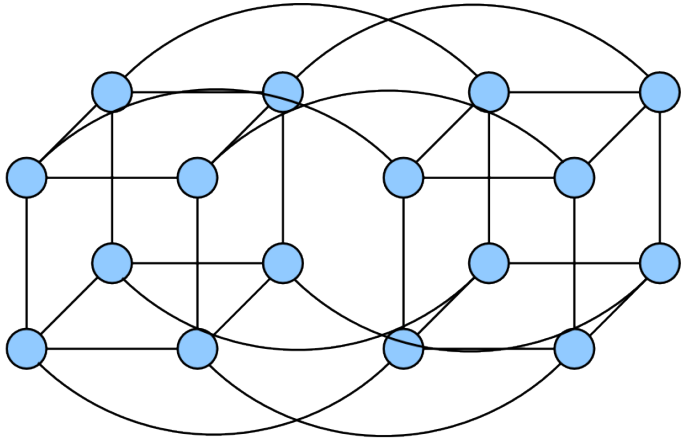
- Direct interconnection



2D hypercube



3D hypercube



4D hypercube

- N components → $\log_2(N)$ dimensional hypercube
- Very wide-spread interconnection network topology

Routing

Routing

- Routing: selects a suitable path in the network to transmit messages
- Routing decisions are made by
  - The components of the multi-processor system
    - In case of direct interconnections
      - Nodes composed by CPU, memory and cache
      - Processor or memory modules
    - In case of indirect interconnections
      - The message forwarding devices of the interconnection network
- Two classes:
  - Static: the route depends on the sender and the target only
  - Dynamic: the current network conditions are also taken into consideration

Static routing

- Three wide-spread solutions

1) Dimension order routing:

- Can be used if the topology has a regular geometry
- Nodes are identified by their locations
- Example: we have a 2D mesh
  - Target: (x,y)
  - Switch checks „x” first, and compares it to its own x coordinate:

    x is to the left: it transmits the message to the left neighbor, x is to the right: it sends it to the right
  - If „x” of the target matches the x coordinate of the switch: it checks „y” and sends the message up or down depending on the location of „y” relative to the placement of the switch

2) Source routing

- The source node determines the entire path of the message and writes the path into the message
- The source needs to have a correct picture about the network
- The message size increases as it now holds the routing path as well

Static routing

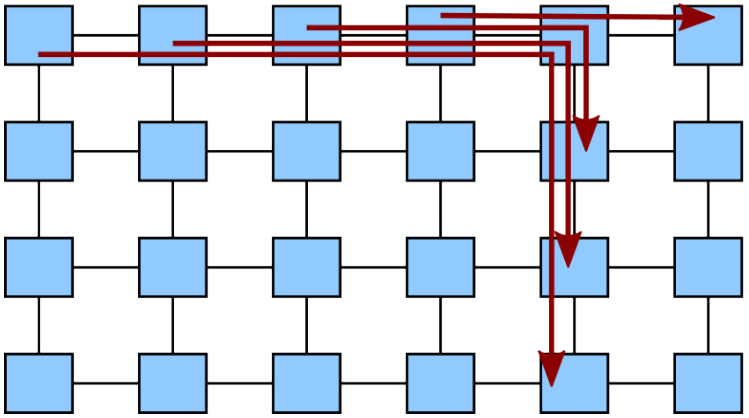
3) **Table-driven routing**

- Each node contains a routing table:
  - For each destination node it stores to which direction the message needs to be forwarded
- If the network is large, the routing table is large as well
  - Consumes a lot of energy
  - Finding the necessary entry may take a lot of time

Dynamic routing

- Adapts to the current conditions of the interconnection network
  - Tries to avoid over-loaded connections
  - Tries to by-pass failed connections
- Introduces varying message transmission latency
- With an appropriate implementation it improves both the transmission latency and the throughput

Static routing



Dynamic routing