CS3210 Cheatsheet AY22/23 Sem 1

by Richard Willie

Parallel Computing

Definition

- Simultaneous use of multiple processing units to solve a problem fast.
- Terminologies:
  - PE = Processing Element = Processor (single or multi-core)
  - PU = Processing Unit = CPU = Core
  - Node = Computer
  - Cluster = Multiple nodes

Example

- We need to compute n values and add them together.
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
  x = compute_next_value(...);
  sum += x;
}
```

- Parallel solution:
  - Suppose we have p cores where $p \leq n$.
  - Each core can form a partial sum of approximately n/p values.

```
my_sum = 0;
my_begin = ...;
my_end = ...;
for (my_i = my_begin; my_i < my_end; my_i++) {
  my_x = compute_next_value(...);
  my_sum += x;
}
```

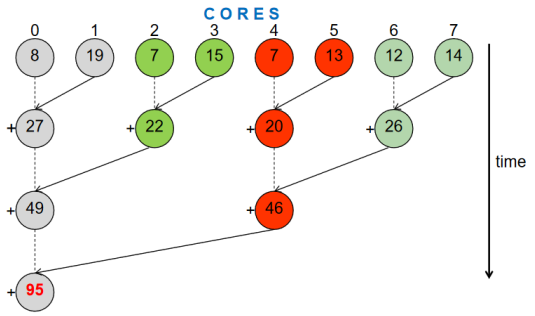
  - Suppose $n = 24$ and $p = 8$, the partial sum calculated at each core:

core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

  - The master core adds up all the values of `my_sum` to form the global sum:

```
if master {
  sum = my_x;
  for each core other than myself {
    receive value from other core;
    sum += value;
  }
} else {
  send my_x to the master;
}
```

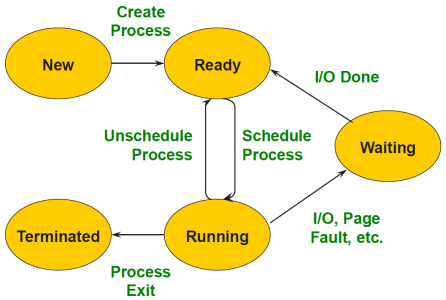
- Better parallel solution:
  - Share the work of the global summing:



Processes and Threads

Processes

- A **process** is a program in execution.
- Process state:



- **Interprocess Communication (IPC)** is a mechanism that allow processes to exchange information.
  - Two fundamental models:
    - **Shared memory**. A region of memory is shared. Processes exchange information by reading and writing to the shared region.
    - **Message passing**. Communication takes place by means of messages exchanged between the processes.
- Disadvantages of processes:
  - Creating a new process is costly.
    - Overhead of system calls.
    - All data structures must be allocated, initialized, and copied.
  - Communicating between processes is costly.
    - Communication goes through the OS.

Threads

- A multi-threaded program has more than one point of execution (i.e. multiple PCs).
- Threads in the same process share the same address space.
- Thread generation is faster than process generation.
  - No copy of address space is necessary.
- Two types of threads:
  - User-level threads.
  - Kernel threads.

User-level Threads

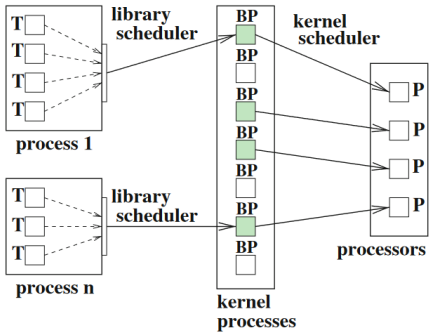
- Managed by a thread library, i.e. not visible to the OS.
- Advantages:
  - Switching thread contexts is fast.
- Disadvantages:
  - No parallelism.
  - OS cannot switch to another thread if one thread executes a blocking I/O operation.

Kernel Threads

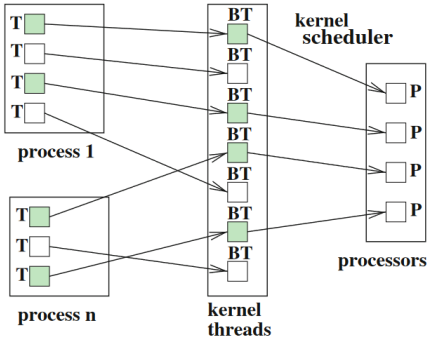
- Managed by the OS.
- Avoid disadvantages of user-level threads.

Multithreading Models

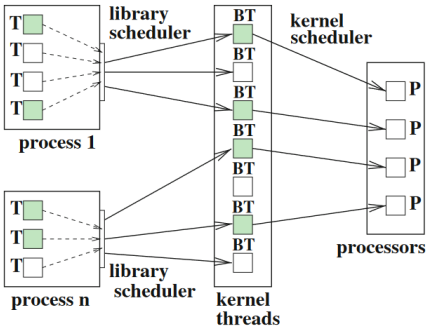
- A relationship must exists between user-level threads and kernel threads.
- Three common ways:
  - **Many-to-one Model**
    - Map many user-level threads to one kernel thread.
    - Thread management is done by the thread library in user space, so it is efficient.
    - The entire process will block if a thread makes a blocking system call.
    - Multiple threads are unable to run in parallel.



  - **One-to-one Model**
    - Map each user-level thread to a kernel thread.
    - Multiple threads run in parallel.
    - Creating a user-level thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of the system.



  - **Many-to-many Model**
    - Library scheduler assigns the user-level threads to a given set of kernel threads.
    - Kernel scheduler maps the kernel threads to the available execution resources.
    - In theory, avoid disadvantages of many-to-one and one-to-one models.
    - In practice, difficult to do an efficient implement.



Synchronization

Race Condition

- Criteria:
  1. Two concurrent threads access a shared resources without any synchronization.
  2. At least one thread modifies the shared resources.
- A solution to the critical-section problem must satisfy the following three requirements:
  1. **Mutual exclusion**
     - If one thread is in the critical section, than no other is.
  2. **Progress**
     - If thread T is not in the critical section, then T cannot prevent other threads from entering the critical section.
     - A thread in the critical section will eventually leave it.

3. **Bounded waiting (no starvation)**
   - There exists a bound on the number of times that other threads are allowed to enter the critical section after a thread has made a request to enter the critical section.
   - In other words, if thread T is waiting to enter its critical section, then T will eventually enter the critical section.

Mechanisms

- **Locks**
  - Two operations: `acquire` and `release`.
  - Locks can spin (a spinlock) or block (a mutex).
- **Semaphores**
  - Two operations: `wait` and `signal`.
  - Safety property: the semaphore value is always greater than or equal to 0.

Deadlock

- We say that a set of processes is in a deadlocked state when every processes in the set is waiting for an event that can be caused only by another process in the set.
- Deadlock can exist if and only if the following four conditions hold simultaneously:
  1. **Mutual exclusion** – at least one resource must be held in a non-sharable mode.
  2. **Hold and wait** – there must be one process holding one resource and waiting for another resource.
  3. **No preemption** – resources cannot be preempted, i.e. critical sections cannot be aborted externally.
  4. **Circular wait** – there must exist a set of processes $\{P_1, P_2, \ldots, P_n\}$ such that P_1 is waiting for P_2, P_2 for P_3, etc.

Starvation

- **Starvation** is a situation where a process is prevented from making progress because some other process has the resource it requires.
- Starvation is a side effect of the scheduling algorithm:
  - A high priority process always prevents a low priority process from running on the CPU.
  - One thread always beats another when acquiring a lock.

Concurrency vs. Parallelism

Concurrency

- Two or more tasks can start, run, and complete in overlapping time periods.
- They might not be running (executing on the CPU) at the same time.
- Two or more execution flows make *progress* at the same time by interleaving their executions or by executing instructions (on CPU) at exactly the same time.

Parallelism

- Two or more tasks can run (execute) simultaneously, at the exact same time.
- Tasks do not only make progress, they actually execute at *simultaneously*.

Level of Parallelism

Bit-level Parallelism

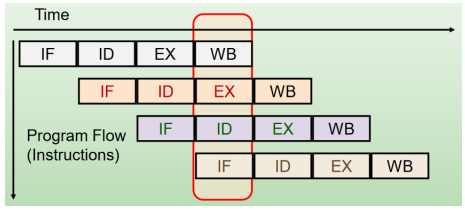
- Process one word at a time.
- The term **word** may refer to:
  - Unit of transfer between the processor and memory.
  - Memory address space capacity.
  - Integer size.
  - Single precision floating point number size.

Instruction-level Parallelism

- Execute instructions in parallel by:
  - Pipelining (parallelism across time)
  - Superscalar (parallelism accross space)

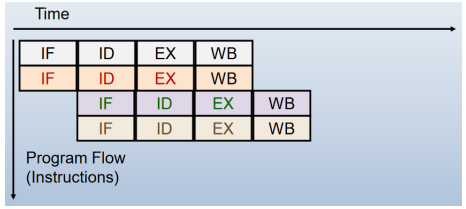
Parallelism by Pipelining

- Each instruction is split into several steps (e.g. fetch, decode, execute, write-back).
- Allow multiple instructions to occupy different stages in the clock cycle.



Parallelism by Superscalar

- Allow multiple instructions to pass through the same stage.



Thread-level Parallelism

Fine-grained Multithreading

- Interleave the execution of multiple threads by performing context switch after each instruction.

Coarse-grained Multithreading

- Similar to fine-grained multithreading, but only switch context on costly stalls.
- Two approaches:
  - **Timeslice Multithreading**
    - The processors switches between the threads after a predefined timeslice interval has elapsed.
  - **Switch-on-event Multithreading**
    - The processors switches to the next thread if the current thread must wait for an event ot occur as it can happen for cache misses.

Simultaneous Multithreading (SMT)

- Unlike SMT, the variants of multithreading discussed before do not truly execute multiple threads simultaneously in parallel.
- **SMT** schedules executable instructions from different threads in the same cycle.
- Requires hardware support for multiple thread contexts, which includes PC, registers, etc.

Processor-level Parallelism

- Add more cores to the processor.
- Each core represents an independent logical processor with separate execution resources, such as functional units or execution pipelines.

Flynn's Taxonomy

Single Instuction, Single Data (SISD)

- A single instruction stream is executed.
- Each instruction works on a single data.

Single Instuction, Multiple Data (SIMD)

- A single stream of instruction.
- Each instruction works on multiple data.
- Exploit data parallelism, commonly known as a vector processor.
- Modern examples:
  - SSE in Intel x86 processors.
  - AVX instructions, which operates on vectors of four 64-bit values.

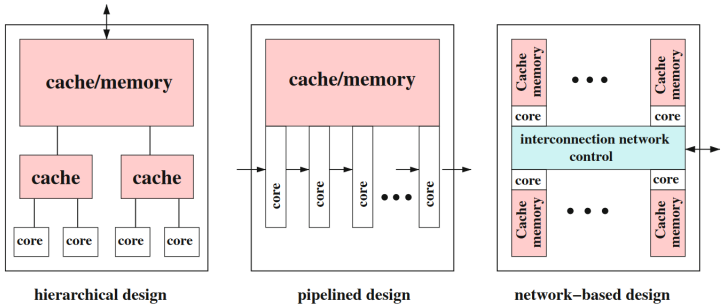
Multiple Instuction, Single Data (MISD)

- Multiple instruction streams.
- All instructions work on the same data at any time.
- No actual implementation except for the systolic array.

Multiple Instuction, Multiple Data (MIMD)

- Each processing unit fetch its own instruction.
- Each processing unit operates on its data.

Multicore Architecture



Hierarchical Design

- Multiple cores share multiple caches.
- The size of the caches increases from leaves to the root.
- Each core can have a separate L1 cache and shares the L2 cache with other cores.
- All cores share the common external memory.

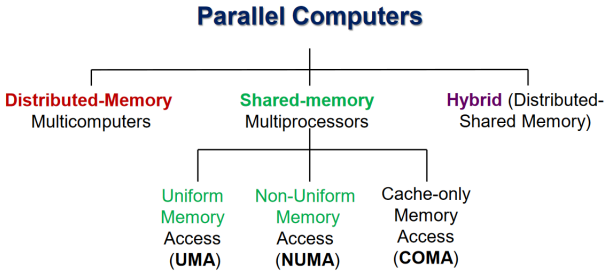
Pipelined Design

- Data elements are processed by multiple execution cores in a pipelined way.
- Each core performs specific processing steps on each data element.
- Useful for application areas in which the same computation steps have to be applied to a long sequence of data elements.

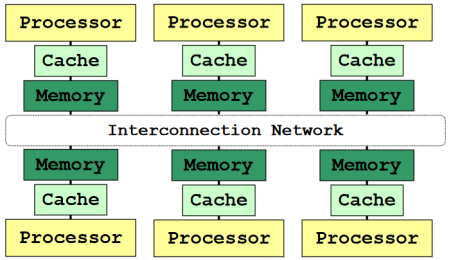
Network-based Design

- The cores and their local caches and memories are connected via an interconnection network with other cores of the chip.

Memory Organization of Parallel Computers

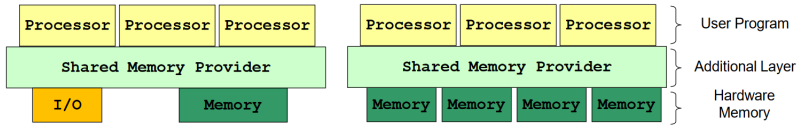


Distributed-memory



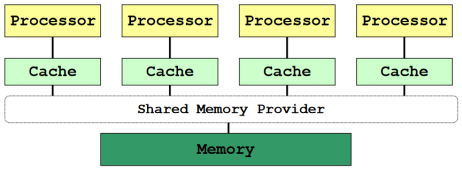
- Consist of a number of processing elements (called nodes).
  - Each node is an independent unit, with processor, local memory, and peripheral elements.
- An interconnection network connects nodes and supports the transfer of data between nodes.
  - The data exchange is usually implemented with message-passing.
- Program data are stored in the local memory of one or several nodes.
  - All local memory is private and only the local processor can access the local memory directly.

Shared-memory



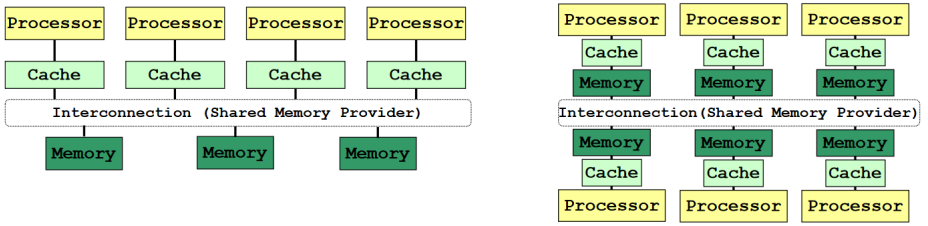
- Parallel programs/threads access memory through the shared memory provider, which maintain the illusion of a shared memory.
- Program is unaware of the actual hardware memory architecture.
  - This causes cache coherency and memory consistency problems.
- Two factors can further differentiate shared memory systems:
  - Processor to memory delay (UMA/NUMA)
  - Presence of a local cache with cache coherence protocol (CC/NCC)
- Advantages:
  - No need to partition code or data.
  - Communication is efficient.
- Disadvantages:
  - Special synchronization constructs are required.
  - Lack of scalability due to *contention*.

Uniform Memory Access (UMA)



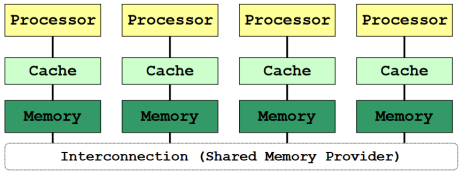
- Latency of accessing the main memory is the same for every processor, i.e. uniform access time.
- Suitable for small number of processors due to *contention*.

Non-uniform Memory Access (NUMA)



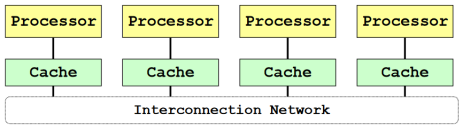
- Accessing local memory is faster than remote memory for a processor, i.e. non-uniform access time.

Cache-coherence Non-uniform Memory Access (ccNUMA)



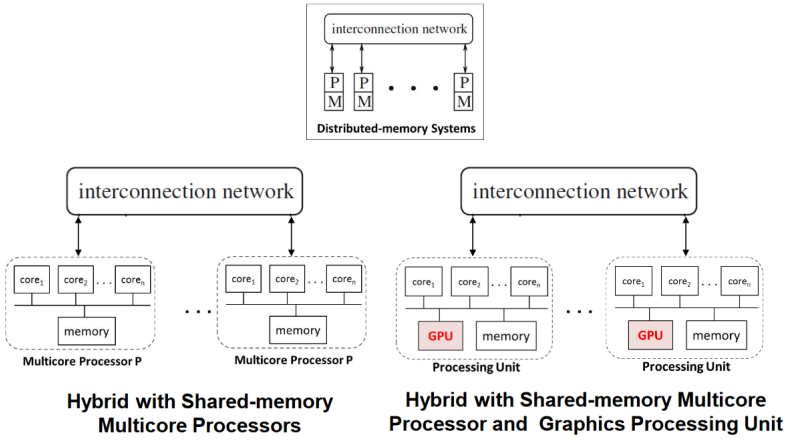
- Each node has cache memory to reduce contention.

Cache-only Memory Access (COMA)



- Each memory block works as cache memory.
- Data migrates dynamically and continuously according to the cache coherence scheme.

Hybrid (Distributed-shared Memory)



Types of Parallelism

Data Parallelism

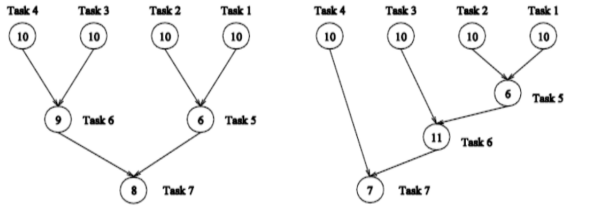
- Same operation is applied to different elements of a data set.
- Loop parallelism:
  - Many algorithm perform computations by iteratively traversing a large data structure.
  - If the iterations are independent, iterations can be executed in arbitrary order and in parallel on different cores.

Task Parallelism

- Independent program parts (tasks) can be executed in parallel.

Task Dependence Graph

- Used to visualize and evaluate the task decomposition strategy.
- A directed acyclic graph:
  - Node: represent each task, node value is the expected execution time.
  - Edge: represent control dependency between task.
- Properties:
  - Critical path = maximum completion time
  - Degree of concurrency = total work / critical path



Foster's Design Methodology

1. Partitioning

- Divide computation and data into independent pieces to discover maximum parallelism.
  - **Domain decomposition (data centric)**
    - Divide data into pieces of approximately equal size.
    - Determine how to associate computations with the data.
  - **Functional decomposition (computation centric)**
    - Divide the computation into pieces (tasks).
    - Determine how to associate data with the computations.

2. Communication

- **Local communication**
  - Tasks need data from a small number of other tasks (neighbors).
  - Create channels illustrating data flow.
- **Global communication**
  - Significant number of tasks contribute data to perform a computation.
  - Don't create channels for them early in design.
- Ideally, distribute and overlap computation and communication.

3. Agglomeration

- Combine tasks into larger tasks.
- Goals:
  - Improve performance by decreasing cost of task creation and communication.
  - Maintain scalability of program.

4. Mapping

- Assignment of tasks to execution units.
- Conflicting goals:
  - **Maximize process utilization** – place tasks on different processing units to increase parallelism.
  - **Minimize inter-processor communication** – place tasks that communicate frequently on the same processing units to increase locality.

Parallel Programming Patterns

Fork-join

- An existing thread T create a number of child threads $T_1, \ldots, T_m$ with a `fork` statement.
- The child threads work in parallel and execute a given program part or function.
- Thread T can execute the same or a different program part or function.
- Thread T wait for the termination of $T_1, \ldots, T_m$ by using a `join` call.

Parbegin-parend

- Programmer specifies a sequence of statements (function calls) to be executed by a set of cores in parallel.
- When an executing thread reaches a parbegin-parend construct, a set of threads is created and the statements of the construct are assigned to these threads for execution.
- The statements following the parbegin-parend construct are only executed after all these threads have finished their work.

SPMD and SIMD

- Same program executed on different cores but operates on different data.
- All threads have equal rights and different threads work asynchronously with each other.
- Synchronization can be achieved by explicit synchronization operations.

Master-worker

- A single program (master) controls the execution of the program.
- Master executes the main function.
- Master assigns work to worker threads to perform computations.

Client-server

- MPMD model.
- Server computes requests from multiple client tasks concurrently.
- Useful in heterogeneous systems such as cloud and grid computing.

Task Pools

- A common data structure from which threads can access to retrieve tasks for execution.
- Number of threads is fixed.
- During the processing of a task, a thread can generate new tasks and insert them into the task pool.
- Access to the task pool must be synchronized to avoid race conditions.
- Execution of a parallel program is completed when
  - task pool is empty and
  - each thread has terminated the processing of its last task
- Advantages:
  - Useful for adaptive and irregular applications, where tasks are generated dynamically.
  - Overhead for thread creation is independent of the problem size and the number of tasks.
- Disadvantages:
  - For fine-grained tasks, the overhead of retrieval and insertion of tasks becomes important.

Producer-consumer

- Producer threads produce data which are used as input by consumer threads.
- Synchronization has to be used to ensure a correct coordination between producer and consumer threads.

Pipelining

- Data in the application is partitioned into a stream of data elements that flows through the each of the pipeline tasks one after the other to perform different processing steps.
- A form of functional parallelism: stream parallelism.

Performance of Sequential Programs

Response Time

- Known as **wall-clock time**.
- Response time of a program includes:
  - **User CPU time** = time CPU spends for executing program
  - **System CPU time** = time CPU spends executing OS routines
  - **Waiting time** = I/O waiting time and the execution of other programs because of time sharing
- Considerations:
  - Waiting time depends on the load of the computer system.
  - System CPU time depends on the OS implementation.

User CPU Time

- Depends on:
  - Translation of program statements by the compiler into instructions.
  - Execution time for each instruction.

$$T_{\text{user}}(A) = N_{\text{cycle}}(A) \cdot T_{\text{cycle}}$$

  - where:
    - $T_{\text{user}}(A)$ = user CPU time of a program A
    - $N_{\text{cycle}}(A)$ = total number of CPU cycles needed for all instructions
    - T_{cycle} = cycle time of CPU
- But instructions may have different execution times:

$$N_{\text{cycle}}(A) = \sum_{i=1}^{n} n_i(A) \cdot \text{CPI}_i$$

  - where:
    - $n_i(A)$ = number of instructions of type I_i
    - CPI_i = average number of CPU cycles needed for instructions of type I_i
- Thus using CPI:

$$T_{\text{user}}(A) = N_{\text{instr}}(A) \cdot \text{CPI}(A) \cdot T_{\text{cycle}}$$

  - where:
    - $\text{CPI}(A)$ = depends on the internal organization of the CPU, memory system, and compiler
    - $N_{\text{instr}}(A)$ = total number of instructions executed for A

- Include memory access time:

$$T_{\text{user}}(A) = (N_{\text{cycle}}(A) + N_{\text{mm\_cycle}}(A)) \cdot T_{\text{cycle}}$$

  - where:
    - $N_{\text{mm\_cycle}}(A)$ = number of additional clock cycles due to memory accesses
  - Consider a one-level cache:

$$N_{\text{mm\_cycle}} = N_{\text{read\_cycle}}(A) + N_{\text{write\_cycle}}(A)$$
$$N_{\text{read\_cycle}}(A) = N_{\text{read\_op}} \cdot R_{\text{read\_miss}}(A) \cdot N_{\text{miss\_cycles}}(A)$$
$$N_{\text{write\_cycle}}(A) = N_{\text{write\_op}} \cdot R_{\text{write\_miss}}(A) \cdot N_{\text{miss\_cycles}}(A)$$

Average Memory Access Time

$$T_{\text{read\_access}}(A) = T_{\text{read\_hit}} + R_{\text{read\_miss}}(A) \cdot T_{\text{read\_miss}}$$

  - where:
    - $T_{\text{read\_access}}(A)$ = average read access time of a program A
    - $T_{\text{read\_hit}}$ = time for read access to the cache irrespective of hit or miss
    - $R_{\text{read\_miss}}(A)$ = cache read miss rate of program A
    - $T_{\text{read\_miss}}$ = read miss penalty time

- Two-level cache example:

$$T_{\text{read\_access}}(A) = T^{\text{L1}}_{\text{read\_hit}} + R^{\text{L1}}_{\text{read\_miss}}(A) \cdot T^{\text{L1}}_{\text{read\_miss}}$$
$$T^{\text{L1}}_{\text{read\_miss}}(A) = T^{\text{L2}}_{\text{read\_hit}} + R^{\text{L2}}_{\text{read\_miss}}(A) \cdot T^{\text{L2}}_{\text{read\_miss}}$$

- Global miss rate = $R^{\text{L1}}_{\text{read\_miss}}(A) \cdot R^{\text{L2}}_{\text{read\_miss}}(A)$

Throughput: Million-Instruction-Per-Second

$$\text{MIPS}(A) = \frac{N_{\text{instr}}(A)}{T_{\text{user}}(A) \cdot 10^6}$$
$$\text{MIPS}(A) = \frac{\text{clock\_frequency}}{\text{CPI}(A) \cdot 10^6}$$

- Drawbacks:
  - Consider only the number of instructions.
  - Easily manipulated.

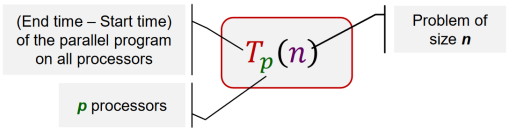
Million Floating-Point Operations Per Second (MFLOPS)

$$\text{MFLOPS}(A) = \frac{N_{\text{FLOPS}}(A)}{T_{\text{user}}(A) \cdot 10^6}$$

- $N_{\text{FLOPS}}(A)$ = number of floating-point operations of program A
- Drawback:
  - No differentiation between different types of floating-point operations.

Performance of Parallel Programs

Parallel Execution Time



- Consists of:
  - Time for executing local computations.
  - Time for exchange of data between processors.
  - Time for synchronization between processors.
  - Waiting time:
    - Unequal load distribution of the processors.
    - Wait to access a shared data structure.

Cost of Parallel Program

- Cost of a parallel program with input size n executed on p processors:

$$C_p(n) = p \cdot T_p(n)$$

- $C_p(n)$ measures the total amount of work performed by all processors, i.e. processor-runtime product.
- A parallel program is cost-optimal if it executes the same total number of operations as the fastest sequential program.

Speedup of Parallel Programs

$$S_p(n) = \frac{T_{\text{best\_seq}}(n)}{T_p(n)}$$

- Theoretically, $S_p(n) \leq p$ always hold.
- In practice, $S_p(n) > p$ (superlinear speedup) can occur.

Efficiency of Parallel Programs

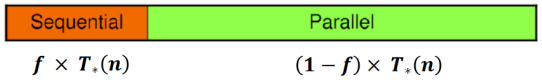
$$E_p(n) = \frac{T_{\text{best\_seq}}(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T_{\text{best\_seq}}}{p \cdot T_p(n)}$$

- Ideal speedup: $S_p(n) = p \longrightarrow E_p(n) = 1$

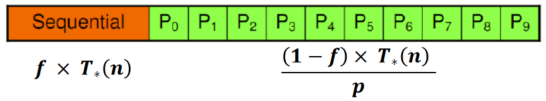
Scalability of Parallel Programs

Amdahl's Law

- Speedup of parallel execution if limited by the fraction of the algorithm that cannot be parallelized.
- Sequential execution time:



- Parallel execution time:



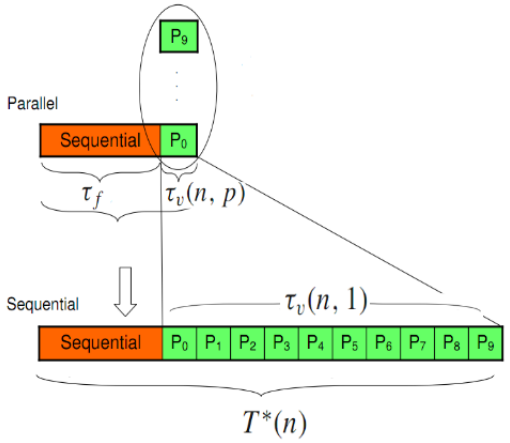
- Speedup:

$$S_p(n) = \frac{T_{\text{best\_seq}}(n)}{f \cdot T_{\text{best\_seq}}(n) + \frac{1-f}{p} \cdot T_{\text{best\_seq}}(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

- Rebuttal: in many computing problems, f is not constant.
  - f depends on the problem size n, i.e. f is a function of n, $f(n)$.
  - An effective parallel algorithm is $\lim_{n\to\infty} f(n) = 0$, thus $\lim_{n\to\infty} S_p(n) = p$.
  - Amdahl's law can be circumvented for large problem size.

Gustafson's Law

- There are certain applications where the main constraint is execution time.
  - Higher computing power is used to improve accuracy.



- Assume a parallel program is perfectly parallelizable, then:

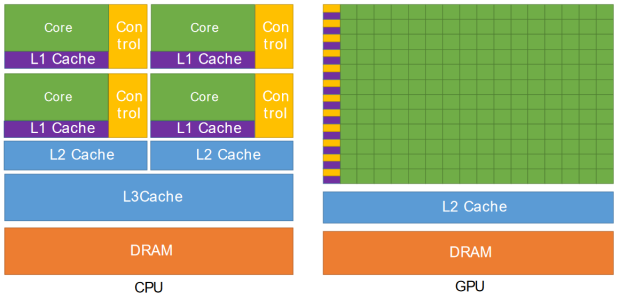
$$S_p(n) = \frac{\tau_f + \tau_v(n,1)}{\tau_f + \tau_v(n,p)} = \frac{\tau_f + T^*(n) - \tau_f}{\tau_f + (T^*(n) - \tau_f)/p} = \frac{\frac{\tau_f}{T^*(n)-\tau_f} + 1}{\frac{\tau_f}{T^*(n)-\tau_f} + \frac{1}{p}}$$

- If $T^*(n)$ increases strongly monotonically with n, then $\lim_{n\to\infty} S_p(n) = p$.

GPGPU Programming

Benefits of GPU

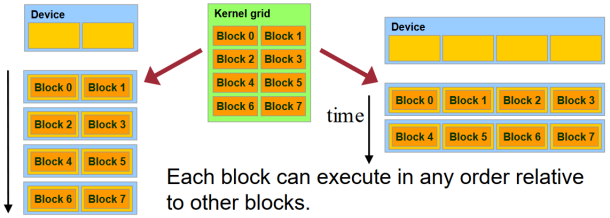
- GPU provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope.
- While the CPU is designed to excel at executing a sequence of operations as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).



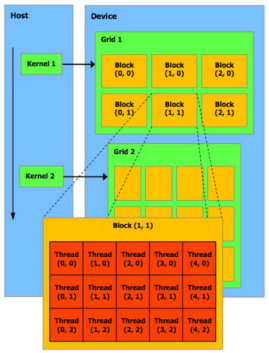
- GPU can hide memory access latencies with computation, instead of relying on large data caches and complex flow control to avoid long memory access latencies, both of which are expensive in terms of transistors.

CUDA Programming Model

- The challenge is to develop software that transparently scales its paralellism to leverage the increasing number of processor cores. The CUDA parallel programming model is designed to overcome this challenge.
- At its core are three key abstractions:
  1. A hierarchy of thread groups
  2. Shared memories
  3. Barrier synchronization
- Terminologies:
  - Device = GPU
  - Host = CPU
  - Kernel = functions that runs on the device
- A CUDA kernel is exeucted by an array of threads.
  - All threads run the same code, i.e. SPMD.
  - Each thread has an ID that it uses to compute memory addresses and make control decisions.
  - Performance is most efficient when all threads can execute in lockstep (parallel).
  - When there is a diverging execution flow, not all threads can be executed in parallel.
- The array of threads are divided into multiple blocks.
  - Hardware is free to schedule thread blocks to any processor at any time.
  - A kernel scales across any number of parallel multiprocessors.



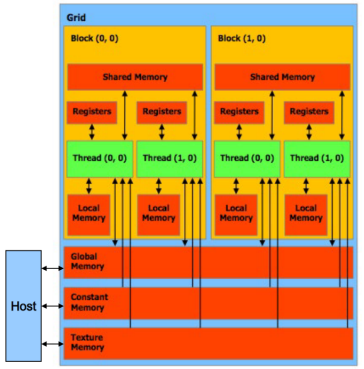
- Thread hierarchy:
  - A kernel is executed by a **grid** of thread blocks.
  - **Blocks** share data through shared memory and synchronize their execution.
  - **Threads** from different blocks cannot cooperate.



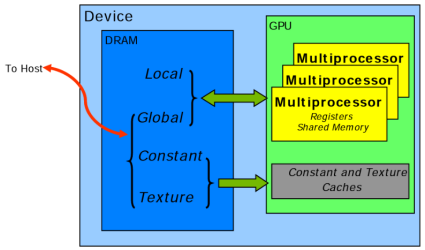
  - The virtual configuration (dimension) of the grid and block can be customized to fit the problem.
    - Matrix multiplication: arrange the threads to represent a matrix.
    - Volumetric computation: arrange the threads to represent a 3D object.
- The NVIDIA GPU architecture is built around a scalable array of multithreaded **Streaming Multiprocessors (SMs)**.
- When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity.
- A block executes on one SM, i.e. it doesn't migrate. However, multiple blocks can execute concurrently on one SM.
- An SM is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs an architecture called **SIMT (Single-Instruction, Multiple-Thread)**.
  - The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps**.
  - The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.
  - A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path.
  - If threads of a warp diverge via a data-dependent conditional branch, the warp executes each branch path taken, disabling threads that are not on that path.
  - Branch divergence occurs only within a warp; different warps execute independently.

CUDA Memory Model

- Access control diagram:

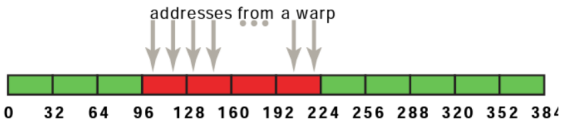


- Physical location diagram:

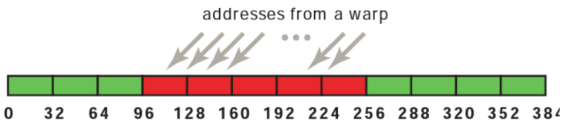


Coalesced Access to Global Memory

- Simultaneous accesses to global memory by threads in a warp are **coalesced** into a number of 32-byte transactions necessary to service all of the threads of the warp.
- Coalesced access, e.g. accessing address 96 to 223:



- Misaligned access, e.g. accessing address 112 to 241:



Stride Accesses

- Stride accesses reduce performance.
- An example of strided memory accesses, because the `location` of different Train objects are not adjacent in memory:

```
struct Train {
  size_t id;
  size_t location;
}

for (size_t i = 0; i < trains.size(); i++) {
  trains[i].location += 1;
}
```

- Structure-of-array solution to avoid stride accesses:

```
struct Trains {
  size_t* id;
  size_t* location;
}
```

Shared Memory

- Higher bandwidth and lower latency than local or global memory.
- Divided into equally-sized memory modules, called **banks**.
- Three scenarios:
  1. Different threads access different memory locations from different banks: not a bank conflict.
  2. Different threads access different memory locations from the same bank: a bank conflict.
  3. Different threads access the same memory location from the same bank: not a bank conflict (it is a broadcast).

Optimizing CUDA Programs

- Overall optimization strategies:
  - **Maximize parallel execution** to achieve maximum utilization.
    - Restructure algorithm to expose as much data parallelism as possible.
    - Strike a balance between occupancy (work per threads) and resource utilization.
  - **Optimize memory usage** to achieve maximum memory throughput.
    - Minimize data transfer between host and device.
    - Ensure global memory accesses are coalesced.
    - Use shared memory to minimize global memory access.
    - Minimize bank conflicts in shared memory accesses.
    - Overlapping asynchronous transfers with computation.
  - **Optimize instruction usage** to achieve maximum instruction throughput.
    - Minimize the use of arithmetic instructions with low throughput.
    - Minimize divergent warps caused by control flow instructions.
    - Reduce the number of instructions.
  - **Minimize memory thrashing**

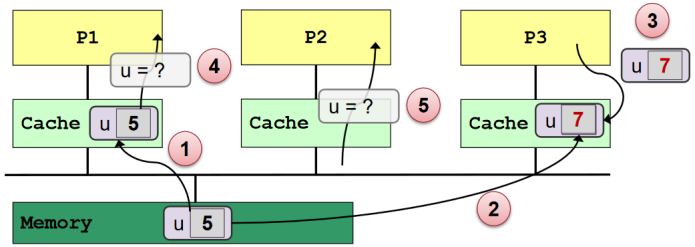
Cache Coherence

Write Policy

- **Write-through** – write access is immediately transferred to main memory.
  - Advantage: always get the newest value of a memory block.
  - Disadvantages: slow down due to many memory accesses.
- **Write-back** – write operation is performed only in the cache.
  - Write is performed to the main memory when the cache block is replaced.
  - Uses a dirty bit.
  - Advantages: less write operations.
  - Disadvantages: memory may contain invalid entries.
  - Example: processor executes `int x = 1`:
    1. Processor performs write to address that "misses" in cache.
    2. Cache selects location to place line in cache, if there is a dirty line currently in this location, the dirty line is written out to memory.
    3. Cache loads line from memory ("allocates line in cache").
    4. Whole cache line is fetched.
    5. Cache line is marked as dirty.

Cache Coherence Problem

- Multiple copies of the same data exists on different caches.



Cache Coherence Protocols

- **Coherence** ensures that each processor has consistent view of memory through its local cache.
  - All processors must agree on the order of reads/writes to the **same memory location** (address).
- Three properties of a coherent memory system:
  1. **Program Order**
  2. **Write Propagation**
  3. **Write Serialization** – all writes to a location (from the same or different processors) are seen in the same order by all processors.
- Cache coherence can be maintained by:
  - Software-based solution
  - Hardware-based solution
- Two major approaches for tracking cache line sharing status:

- **Snooping-based**
  - No centralized directory.
  - Each cache keeps track of the sharing status.
  - Cache **snoop** on the bus to update the status of a cache line.
  - Most common protocol used in architecture with a bus.
- **Directory-based**
  - Sharing status is kept in a centralized memory.
  - Commonly used with NUMA architectures.

Cache Coherence Overheads

- Cache coherence lowers hit rate in cache.
- **Cache ping-pong** – multiple processors read and modify the same address.
- **False sharing** – multiple processors write to different addresses on the same cache line.

Memory Consistency

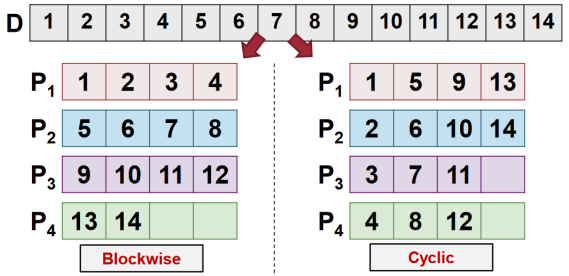
- **Memory consistency** constraints the order in which memory operations performed by one thread become visible to other threads for **different memory locations**.
- Four types of memory operations orderings:
  1. $W \rightarrow R$ – write to X must commit before subsequent read from Y.
  2. $R \rightarrow R$ – read from X must commit before subsequent read from Y.
  3. $R \rightarrow W$ – read from X must commit before subsequent write to Y.
  4. $W \rightarrow W$ – write to X must commit before subsequent write to Y.

Memory Consistency Models

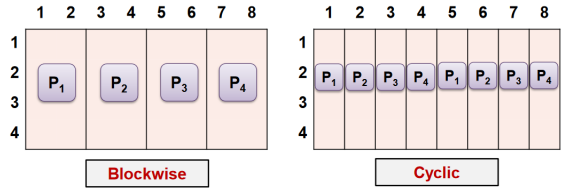
Property	Sequential Consistency (SC)	Total Store Ordering (TSO)	Processor Consistency (PC)	Partial Store Ordering (PSO)
Respects data dependency	Yes			
Preserves $R \rightarrow R$ and $R \rightarrow W$ order	Yes			
Preserves $W \rightarrow R$	Yes	No	No	No
Preserves $W \rightarrow W$	Yes	Yes	Yes	No
Guarantees propagation to all other processes at once	Yes	Yes	No	Yes

Data Distribution

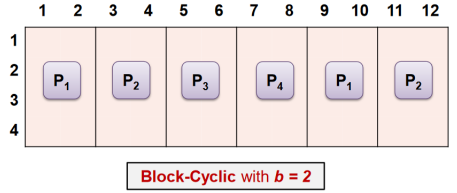
- For problems exhibiting data parallelism, data distribution can be used as a simple parallelization strategy.
- Given a one dimensional array, common distribution patterns:
  - Blockwise data distribution
  - Cyclic data distribution



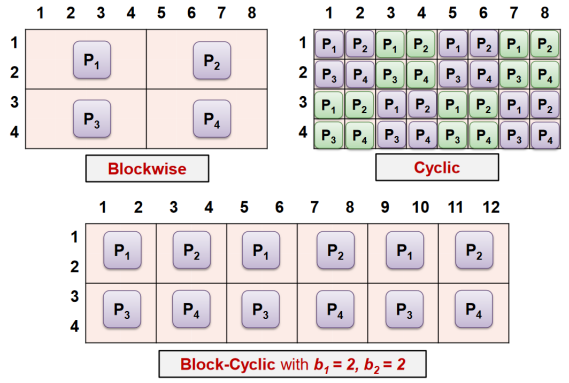
- Data distribution for 2D arrays:
  - Blockwise distribution
  - Cyclic distribution



  - Block-cyclic distribution



  - Checkerboard distribution



Information Exchange

Shared Variables

- Shared memory programming models assume a global memory accessible by all processors.
- Example: OpenMP.

Message Passing

- Distributed memory programming models assume disjoint memory space.
- Exchange of data between processors through dedicated communication operations.
- Send and receive protocols:

	Blocking Operations	Non-Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating the transfer to buffer. This operation might not be completed on return.
Non-buffered	Sending process blocks until matching receive operation has been encountered.	
	Send and receives semantics assured by corresponding operation.	Programmer must explicitly ensure completion of the operation by polling.

- Semantics of send and receive operations:

Local view	Global view
Blocking Return from a library call indicates the user is allowed to reuse resources specified in the call	**Synchronous** Communication operation does not complete before both processes have started their communication operation
Non-blocking A procedure may return before the operation completes, and before the user is allowed to reuse resources specified in the call	**Asynchronous** Sender can execute its communication operation without any coordination with the receiver

- Implementation options:
  - Synchronous:
    - Send completes after matching receive and source data sent.
    - Receive completes after data transfer complete from matching send.
  - Asynchronous:
    - Send completes after input buffer may be reused.

Message Passing Interface (MPI)

- Send and receive operations in MPI:

	Synchronous	Asynchronous
Blocking	`MPI_SSend` `(MPI_Mrecv)`	`MPI_Send` `MPI_Recv`
Non-blocking	`MPI_ISSend` `(MPI_ImRecv)`	`MPI_ISend` `MPI_Irecv`

- Order of receive operations:
  - Two processes (one sender, one receiver): messages delivered in the order which they have been sent.
  - More than two processes: message delivery order not guaranteed.

Deadlock in MPI

- Deadlock occurs when message passing cannot be completed.
- Deadlock occurs if the runtime system does not use system buffers or if the system buffers used are too small.

Process Groups and Communicators

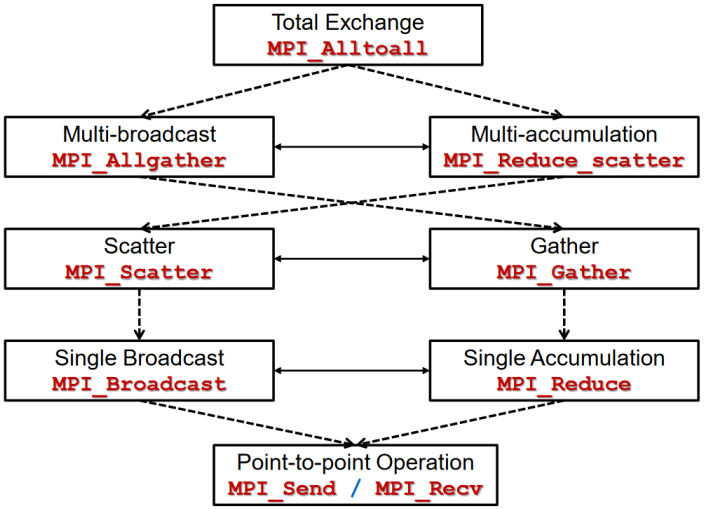
Process Groups

- An ordered set of processes, where each process has a unique rank.
- A process may be a member of multiple groups, and it may have different ranks in each of these groups.

Communicators

- Communicator is the communication domain for a group of processes.
- Two types:
  - **Intra-communicators**: support the execution of arbitrary collective communication operations on a single group.
  - **Inter-communicators**: support the point-to-point communication operations between two process groups.

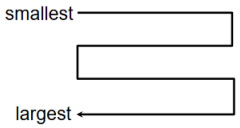
Collective Communication



Interconnection Networks

Example

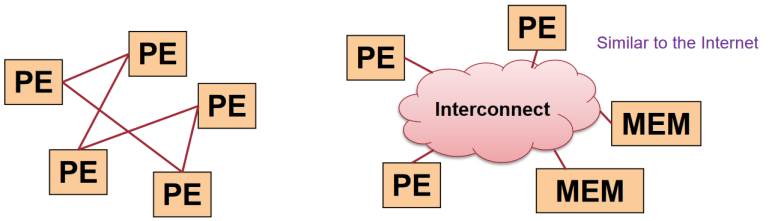
- Sorting n elements:
  - Fastest sequential algorithm = quicksort $\longrightarrow O(n \log n)$
  - Sort on n-PEs linear array:
    - Parallel odd-even sort $\longrightarrow O(n)$
  - Sort on n-PEs 2D mesh:
    - 2D mesh: $\sqrt{n}$ rows and $\sqrt{n}$ columns.
    - Sort into snake-like order:



    - **Shear sort**:
      1. **Row sorting**: odd rows sort in ascending order, while even rows sort in descending order.
      2. **Column sorting**: all columns sort in ascending order from top to bottom.
      3. Repeat until sorted.
    - Claim: for n numbers, shear sort has $O(\log N + 1)$ phases.
    - Analysis:
      - Sorting each row/column is $O(\sqrt{n})$ using parallel odd-even sort.
      - So, the time complexity is $O((\log N + 1) \cdot \sqrt{n}) < O(n)$.
- Conclusion:
  - Connection patterns enable different algorithms.
  - Connection patterns have a huge impact on execution.
  - Other impacts:
    - System scalability – size and extensibility.
    - System performance and energy efficiency.

Topology

- Major types of topology:



  - **Direct interconnection**:
    - Also known as static or point-to-point.
    - Usually endpoints are of the same type (core, memory).
  - **Indirect interconnection**:
    - Also known as dynamic.
    - Interconnect is formed by switches.

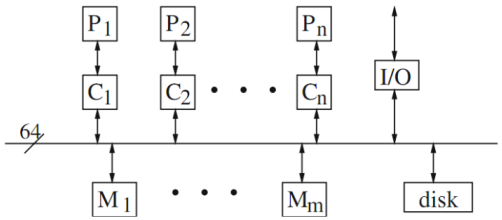
Direct Interconnection

- Topology can be model as a graph.
- Metrics:
  - **Diameter**
    - Maximum distance between any pair of nodes.
    - Usefulness: small diameter ensures small distances for message transmission.
  - **Degree**
    - Degree: number of direct neighbour nodes of node v.
    - Usefulness: small node degree reduces the node hardware overhead.
  - **Bisection width**
    - Minimum number of edges that must be removed in to divide the network into two equal halves.
      - **Bisection bandwidth**: total bandwidth available between the two bisected portion of the network.
    - Usefulness: a measure for the capacity of a network when transmitting messages simultaneously.
  - **Connectivity**
    - **Node connectivity**: minimum number of nodes that must fail to disconnect the network.
      - Usefulness: Determine the robustness of the network.
    - **Edge connectivity**: minimum number of edges that must fail to disconnect the network.
      - Usefulness: determine number of independent paths between any pair of nodes.

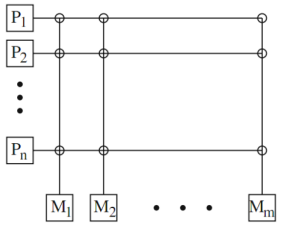
network G with n nodes	degree $g(G)$	diameter $\delta(G)$	edge-connectivity $ec(G)$	bisection bandwidth $B(G)$
complete graph	$n-1$	1	$n-1$	$\left(\frac{n}{2}\right)^2$
linear array	2	$n-1$	1	1
ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d-dimensional mesh ($n = r^d$)	$2d$	$d(\sqrt[d]{n}-1)$	d	$n^{\frac{d-1}{d}}$
d-dimensional torus ($n = r^d$)	$2d$	$d\lfloor \frac{\sqrt[d]{n}}{2} \rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-dimensional hyper-cube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k-dimensional CCC-network ($n = k2^k$ for $k \geq 3$)	3	$2k-1+\lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
complete binary tree ($n = 2^k - 1$)	3	$2\log \frac{n+1}{2}$	1	1
k-ary d-cube ($n = k^d$)	$2d$	$d\lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

Indirect Interconnection

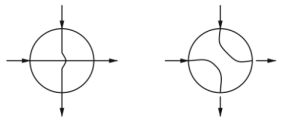
- Why: reduce hardware costs by sharing switches and links.
- How: switches provide indirect connection between nodes and can be configured dynamically.
- Metrics:
  - Cost (number of switches per links)
  - Concurrent connections
- **Bus network**:
  - Only one pair of devices can communicate at a time.



- **Crossbar network**:
  - Hardware is costly (n x m switches), but small number of processors.

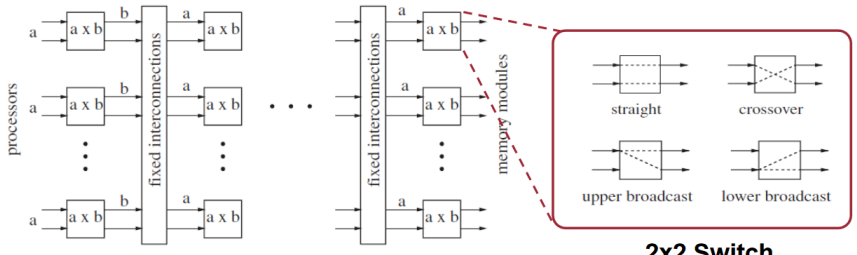


n x m crossbar network



Straight state Direction change state

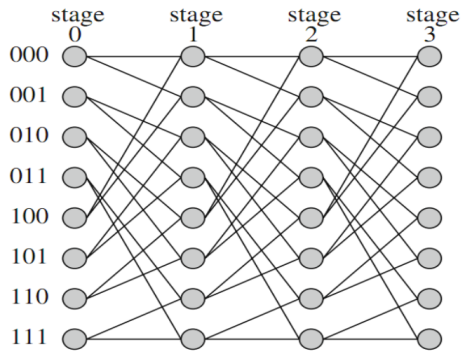
- **Multistage switching network**:
  - Several intermediate switches with connecting wires between neighbouring stages.
  - Goal: obtain a small distance for arbitrary pairs of input and output devices.



Multistage Networks using Regular 2x2 Crossbar Switches

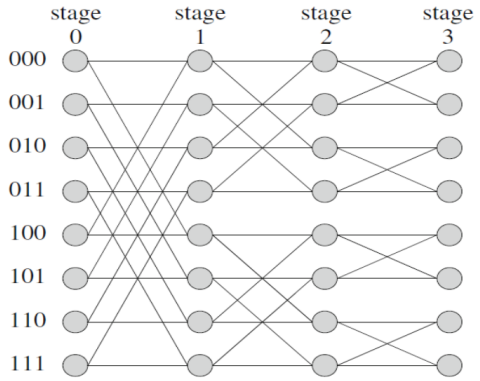
2x2 Switch Settings

- **Omega network**:
  - One unique path for every input to output.
  - An $n \times n$ Omega network has $\log n$ stages.
    - $n/2$ switches per stage.
    - Connections between stages are regular.
    - Also known as $(\log n - 1)$-dimension omega network.

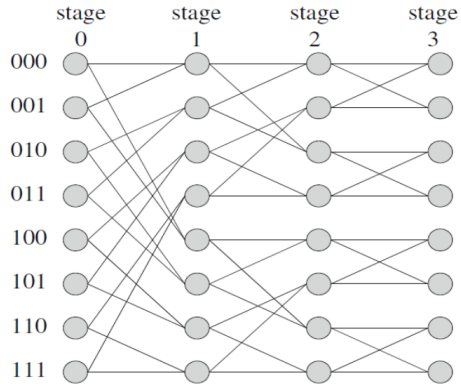


16 × 16 Omega Network

  - Construction:
    - A switch position = (α, i)
      - α = position of a switch within a stage
      - i = position of a switch within a stage
    - Has an edge from node (α, i) to two nodes $(\beta, i+1)$ where:
      - $\beta = \alpha$ by a cyclic left shift
      - $\beta = \alpha$ by a cyclic left shift + inversion of the LSB
- **Butterfly network**:
  - Node (α, i) connects to:
    - $(\alpha, i+1)$, i.e. straight edge
    - $(\alpha', i+1)$, i.e. cross edge where α and α' differ in the $(i+1)$-th bit from the left

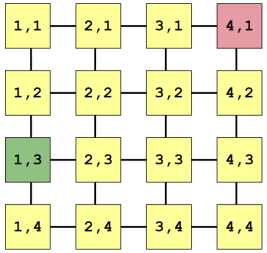


- **Baseline network**:
  - Node (α, i) connects to $(\beta, i+1)$ where:
    - β = cyclic right shift of last $(k-1)$ bits of α
    - β = inversion of the LSB of α + cyclic right shift of last $(k-1)$ bits

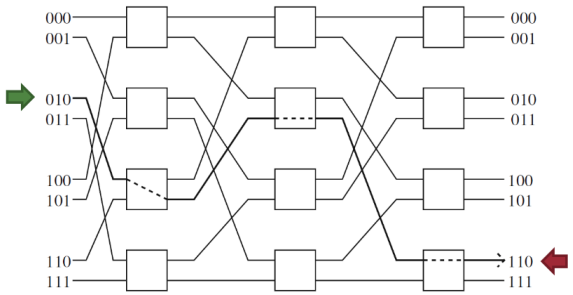


Routing

- Routing algorithm determines path(s) from source to destination within a given interconnection topology.
- Classification:
  - Based on **path length**:
    - **Minimal** or **Non-minimal** routing: whether the shortest path is always chosen.
  - Based on **adaptivity**:
    - **Deterministic**: always use the same path for the same pair of (source, destination) node.
    - **Adaptive**: may take into account of network status and adapt accordingly, e.g. avoid congested path, avoid dead nodes, etc.
- Tree deterministic examples:
  - **XY Routing for 2D Mesh**



    - $(X_{\text{src}}, Y_{\text{src}})$ to $(X_{\text{dst}}, Y_{\text{dst}})$:
      - Move in X direction until $X_{\text{src}} = X_{\text{dst}}$.
      - Move in Y direction until $Y_{\text{src}} = Y_{\text{dst}}$.
  - **E-Cube Routing for Hypercube**
    - Number of bits difference in source and target node address = number of hops = hamming distance
    - Start from MSB to LSB (or LSB to MSB):
      - Find the first different bit.
      - Go to the neighboring node with the bit corrected.
  - **XOR-Tag Routing for Omega Network**



8×8 Omega Network using 2x2 switches

    - Let T = source id $\oplus$ destination id.
    - At stage k:
      - Go straight if bit k of T is 0.
      - Crossover if bit k of T is 1.