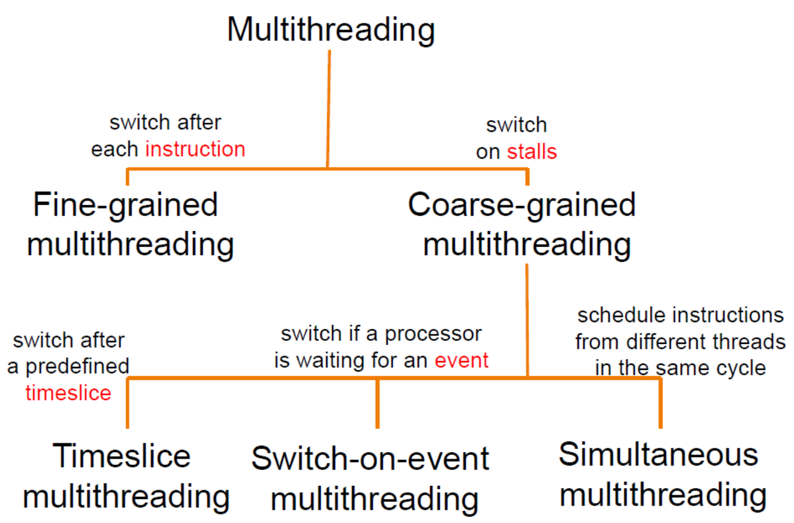
1 Chapter 3 - Parallel Computing Platforms

1.1 Source of Processor Performance Gain

Parallelism of various forms exist to have some performance gain

- Single Processor
  - Bit Level
    - We don't processs bit by bit, but instead word by word.
  - Instruction Level
    - Pipelining (parallelism across time)
      - Split instruction into different stages
      - allow multiple instructions to occupy different stages at same clock cycle.
      - Number of pipeline stages == Maximum achievable speedup
    - Superscalar (parallelism across space)
      - Duplicate the pipelines
      - Allow multiple instructions to pass through the same stage.
      - Hard to schedule, find instructions that can be run together. (Dymanic - Hardware decision, Static - Compiler decision)
      - Disadvantage: structural hazard
    - But is very limited, at most 2-3 instructions one time. due to data/control dependencies
    - Most CPUs have this. Usually we calculate Instructions per cycle, not Cycle-per-instruction.
  - Thread Level
    - Multithreading was originally a software mechanism
    - allow multiple parts of the same program to execute concurrently
    - Processor can provide hardware support for "thread context" like program counter and register (hyper-threading, etc..)
    - software threads can be executed in parallel


Multithreading
switch after each instruction
switch on stalls
Fine-grained multithreading
Coarse-grained multithreading
switch after a predefined timeslice
switch if a processor is waiting for an event
schedule instructions from different threads in the same cycle
Timeslice multithreading
Switch-on-event multithreading
Simultaneous multithreading

  - Process Level
    - Multiple processes work in parallel
    - independent memorry space, need special mechanism to communicate
    - Operating system provide IPC (Inter-Process communication) mechanism
    - Each process has independent context, can be mapped to multiple processor cores
- Multi Processor
  - Processor Level
    - Shared Memory
    - Distributed Memory

1.2 Flynn's Parallel Architecture Taxonomty

- SISD (Single Instruction Single Data)
  - A single instruction stream is executed
  - Each instruction work on single data
  - Most of the uniprocessor
- SIMD (Single Instruction Multiple Data)
  - A single stream of instructins
  - Each instruction work on multiple data
  - supercomputer during 1980s
  - To exploit data parallelism, also known as vector processor
  - Most modern processor has some form of SIMD
- MISD (Multiple Instruction Single Data)
  - Multiple stream of instructins
  - All instruction work on same data at any time
  - No actual implementaion, just here for completeness
- MiMD (Multiple Instruction Multiple Data)
  - Each Processing Unit fetch its own instruction
  - Each Processing operates on its data
  - Most popular model for multiprocessor

Variant - SIMD and MIMD. nVidia GPUs have a set of threads executing the same code (SIMD) and multiple set of threads executing in parallel (MIMD)

1.3 Multicore Architecture

1.3.1 Hierarchical design

Multiple cores share multiple caches. Cache size increase from the leaves to the root. Found in comon day desktop, GPUs.

1.3.2 Pipelined design

Data elements are processed by multiple (different) execution cores in a pipelined way. Bacuase same computation steps have to be applied to a long sequence of data elements. Found in networking application, or GPUs.

1.3.3 Network-based design

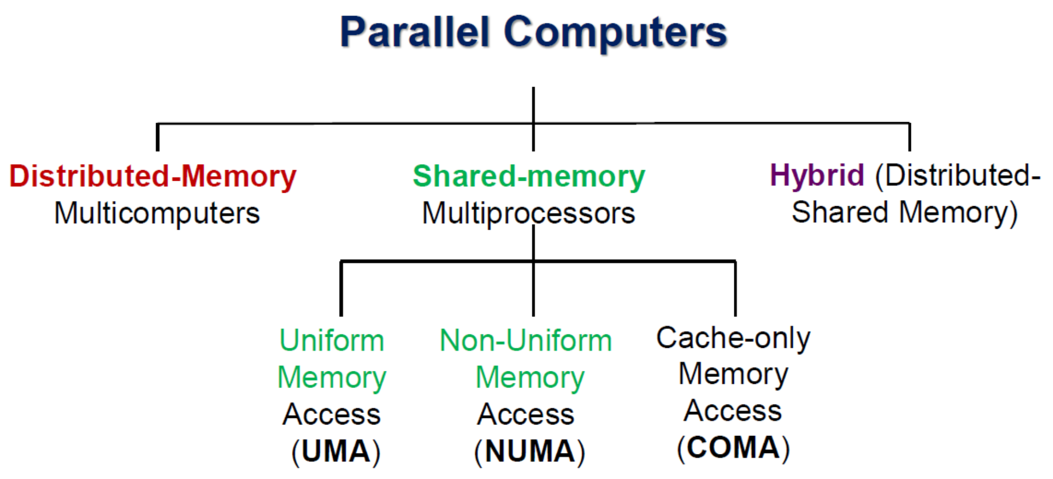
Cores and their local caches and memories are connected via an interconnection network Each core communicate with other cores / memory through this network.

1.4 Memory Organization

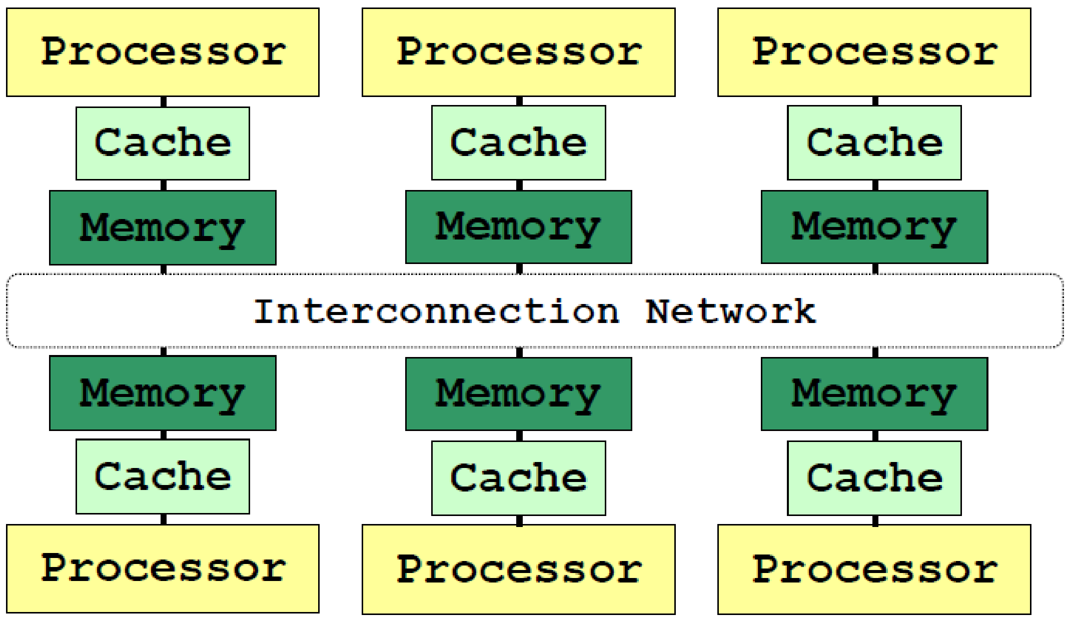
Memory can have consistency problem. If one processor updates value in memory, How will other cores also update their values. This also extends to the cache. Where we have cache coherence problem. If we update value in the cache, how will other processor also update the same value in their caches.

2 Factors differentiate shared memory systems

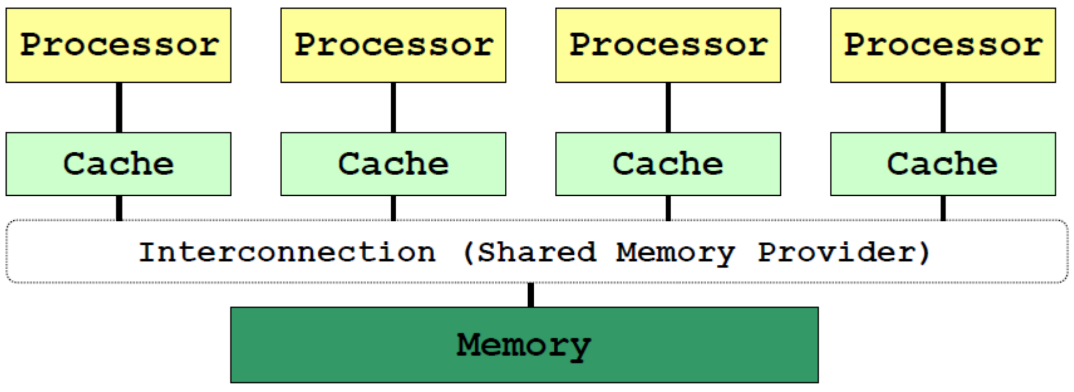
- Processor to Memory Delay (UMA/NUMA)
- presence of a local cache with cache coherence protocol (CC/NCC)


Parallel Computers
Distributed-Memory Multicomputers
Shared-memory Multiprocessors
Hybrid (Distributed-Shared Memory)
Uniform Memory Access (UMA)
Non-Uniform Memory Access (NUMA)
Cache-only Memory Access (COMA)

- Distributed Memory Systems
  - Each node is independent, communicate with message passing

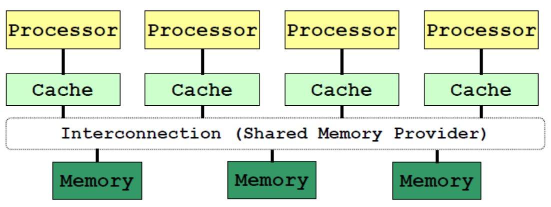


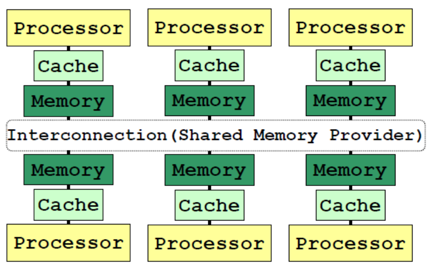
- Shared Memory
  - Uniform Memory Access (Time) (UMA)
    - Latency of accessing main memory is the same for each procesor
    - Good for low under of processors



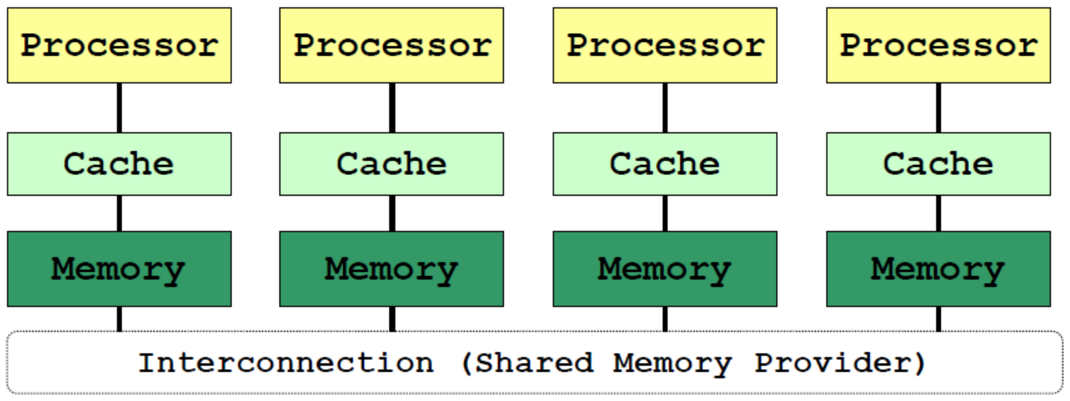
UMA organization for Multiprocessor

- Non-Uniform Memory Access
  * Physically distributed memory of all processing elemets are combined to form a global shared memory
  * Processor can access local memory faster than remote memory

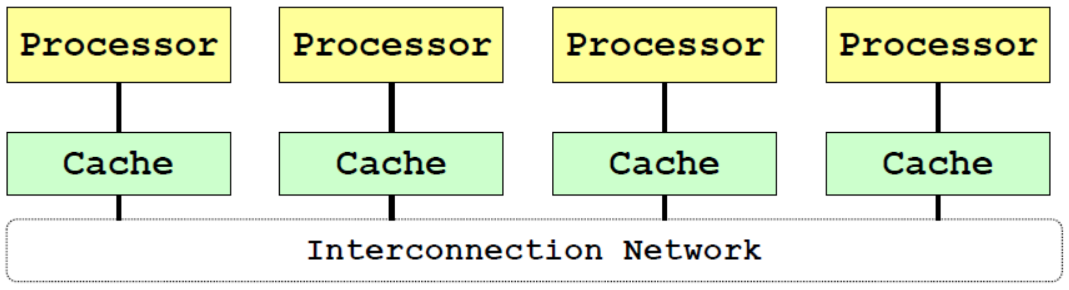




- Cache Coherent Non Uniform Memory Access (ccNUMA)
  * Each node has cache to reduce contention



- Cache Only Memory Architecture
  * Each memory block works as cache memory
  * Uses cache coherence scheme to migrate datq



- Advantages
  * No need to partition code or data
  * No need to physically move data among processors, there is effficient communication
- Disadvantages
  * Special synchronization constructs are required
  * Lack of scalability due to contention

• Hybird

  - Servers, use shared among distributed

2 Chapter 4

2.1 Parallelism

Parallelism: Average number of units of work that can be performed in parallel per unit time. MIPS, MFLOPS, etc.

Types of parallelism:

- Data Parallelism
  - Partition the data, each processing unit carries out similar operations on corresponding part of the data
  - If operations are independent, elements can be distributed among processors for parallel execution.
  - ex: SIMD, loop Parallelism if iterations are independent. (OpenMP)
  - SPMD (single program Multiple Data): One parallel program is executed by all processorts in parallel (both shared and distributed address space)
- Task Parallelism
  - Partition the tasks, each processing unit handles one task
  - A task can be one statement, loop, any number of statements or function calls
  - A single task can be further decomposed. to be executed sequentially by one processor or in parallel by multiple processors.
  - Example: in database query, different subqueries(different tasks) handled by different thread, then joined.

2.2 Models of Coordination

How do our processes communicate?

- Shared address space
  - Tasks communicate by reading/writing to shared variables
  - Ensure mutual exclusion via locks
  - Required hardware support. (any processor can load and store from any address - contention, costly to scale)
  - Very little structure, all threads can read and write to shared variable
  - Drawback: not all read and writes have the same cost
- Data parallel
  - Historically: Same operation on different element of an array (SIMD, vector processors)
  - No communication among distinct function calls, side-effect free execution
  - Model performance-oriented data-parallel languages do not strictly enforce this structure (CUDE, OpenCL, ISPC)
  - Very rigid computation structure
- Message passing
  - Tasks operate within their won private address spaces, communicate by explicitly sending / receiving messages.
  - MPI(Message Passing Interface), popular library
  - Imagine 2 servers, sending data through internet to each other. Similar to distributed memory systems.
  - Highly structured communication, all communication in the form of messages

2.3 Program Prallelization

1. Partitioning
    - Partition a problem into many smaller pieces or tasks
    - Divide computaion and data into independent pieces to maximize parallelism
2. Communication
    - Provides data required by partitioned tasks (cost of paralleism)
    - Deterine how tasks should commuinicate with one another.
    - Local communication: Tasks need data from some number of other tasks (neighbors)
    - Glocal communication: Don't create channels to contribute to final data early (don't have all threads add the same value, every few thread join, then join the joined)
3. Agglomeration
    - Decrease communication and development costs, while maintaining flexibility
    - Improve performance, maintain scalability, reduce communication overhead.
    - reduce granularity,
4. Mapping
    - Map tasks to processor (coress) to minimize total execution time
    - Maximize processor utilization, (place task on different processor)
    - minimize inter-processor communication (place tasks that communicate on same processor, contradiction above)
    - OS does this, or Users in distributed memory systems

Simplified Version:

- Decomposition of the computation (step 1 and 2)
- Scheduling (assignment of tasks to processes or threads) (step 3)
- Mapping of processes (or threads) to physical processors (or cores)

There existys parallelizing compilers the perform decomposition and scheduling. But the dependence analysis is difficult for pointer-based computations or indirect addresssing. Hard to predict at compile time, values known at run time.

2.4 Paralle Programming Patterns

Design pattern, but for parallel programming not software engineering

- Fork-Join
    - Create a child with fork, wait for termination (join)
    - Pthreads, OpenMP, MPI
- Parbegin-Parend
    - Specify region with parbegin-parend construct, when reach this construct a set of threads are created and the statements are assigned to threads for execution
    - Statements after this construct only run after all threads have finished
    - OpenMP or compiler directives
- SPMD
    - Same program run on different processors but operate on different data.
    - Different threads can execute different part of program (if-else bloock, different speed)

  - User needs to determine synchronization.
- SIMD
- Master-Worker (Master-Slave)
  - A single program (master) controls execution flow of the program (coordination and initializations, I/O)
  - Assigns work to worker threads to perform computation (wait for instructions)
- Client-Server
  - MPMD (Multiple Program Multiple Data) model
  - Server compute requests from multiple clients concurrently.
  - can have multiple threads to handle one request.
- Pipelining
- Task pool
  - A common data structure from which threads can access to retrieve tasks for execution
  - Fixed number of threads. statically created by main thread.
  - A thread generates thread, add to task pool
- Producer-Consumer

3 Performance

- Users want to reduce response time
- Computer managers want to have high thoughput

Response time of a program A includes:

1. User CPU time: time CPU spends for executing program
2. System CPU time:time CPU spends executing OS routines
3. Waiting time: I/O waiting time and the execution of other programs because of time sharing

User CPU Time

Depends on the instructions generated by compiler when translating program and execution time of each instruction.

$$Time_{\text{user}}(A) = N_{\text{cycle}}(A) \times Time_{\text{cycle}}$$

$Time_{\text{user}}(A)$	User CPU time of a program A
N_{cycle}	Total number of CPU cycles needed for all instructions
$Time_{\text{cycle}}$	Cyckle time of CPU (clcok cycle time $\frac{1}{\text{clock rate}}$)

But each instruction has different execution time. Consider each time instruction i differently.

$$N_{\text{cycle}}(A) = \Sigma_{i=1}^{n} n_i(A) \times CPI_i$$

$n_i(A)$	number of isntruction of time I_i
CPI_i	average number of CPU cycles needed for instruction of type I_i
N_{cycle}	Total number of CPU cycles needed for all instructions

Thus if we use CPI

$$Time_{\text{user}}(A) = N_{\text{instr}}(A) \times CPI(A) \times Time_{\text{cycle}}$$

$Time_{\text{user}}(A)$	User CPU time of a program
$N_{\text{instr}}(A)$	Total number of instructions executed for A. Depends on architecture, and compiler
$CPI(A)$	Average Cycles per instruction. Depends on CPU, memory, compiler
$Time_{\text{cycle}}$	Time per cycle

If we incluude **Memory Access Time**

$$Time_{\text{user}}(A) = (N_{\text{instr}}(A) \times CPI(A) + N_{\text{rw\_op}}(A) \times R_{\text{miss}}(A) \times N_{\text{miss\_cycles}}) \times Time_{\text{cycle}}$$

$N_{\text{rw\_op}}(A)$	total number of read or write operation
$R_{\text{miss}}(A)$	(read and write) miss rate
$N_{\text{miss\_cycles}}$	number of additional cycles needed for leading a new cache line

If we only consider reading time

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

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

We can expand memory access time to consider different level of cache

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

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

In the end the global miss rate is $R^{\text{L1}}_{\text{read\_miss}}(A) \times R^{\text{L2}}_{\text{read\_miss}}(A)$

How to evaluate **throughput**

Million-Instruction-Per-Second (MIPS)

$$MIPS\,(A) = \frac{N_{\text{instr}}\,(A)}{Time_{\text{user}}\,(A) \times 10^6}$$

$$MIPS\,(A) = \frac{\text{clock\_frequency}}{CPI\,(A) \times 10^6}$$

But we can easily manipulate, a lot of short instruction vs low number of long instruction
Million-Floaring point-Operation-Per-Second (MFLOPS)

$$MFLOPS\,(A) = \frac{N_{\text{fl\_ops}}\,(A)}{Time_{\text{user}}\,(A) \times 10^6}$$

- $N_{\text{fl\_ops}}\,(A)$: number of floating point operations in program A
- But we do not differentiate between different types of floating point operation.

3.1 Speedup

We denote $T_p\,(n)$ as T the time of the parallel program on all processors on p processors for a problem of size n. This contains everything, waiting time, time for computation, time for synchronization, exchange of data.

The cost of a paralle program with input size n executed on p processors: $C_[\,(n) = p \times T_p\,(n)]$ We measuer the total amout of work performed by all processors $C_p\,(n)$. A parallel program is cost-optimal if it executes the same total number of operation as the fastest sequential program.

We measure speed up with $S_p\,(n) = \frac{T_{\text{best\_seq}}(n)}{T_p(n)}$ In theory $S_p\,(n) \leq p$ always holds, but in practice $S_p\,(n) > p$ can happen, ex: if problem working task fits in the cache. One issue is that we do not know what is the best sequential algorithm. best order of growth or lowest execution time. Also difficulty of implementaion

The Parallel Program Efficiency is the actual degree of speedup performance acheived comapred to the maximum

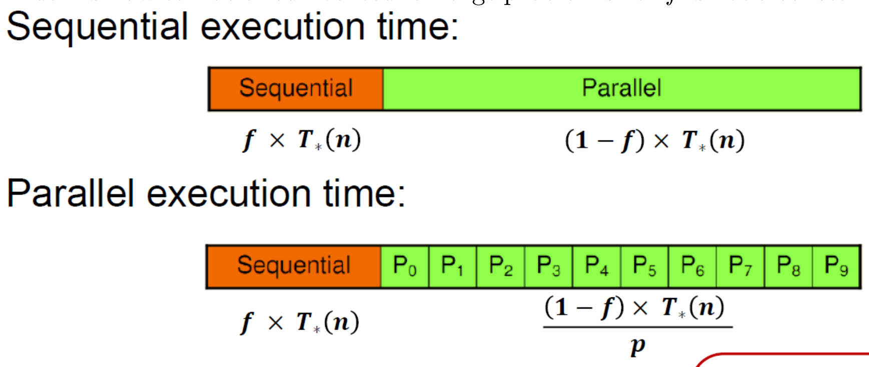
$$E_p\,(n) = \frac{T_{\text{best\_seq}}\,(n)}{p \times T_p\,(n)}$$

Ideal speedup $S_p\,(n) = p$, and idealy $S_p\,(n) = 1$ But if we space the problem, need to bnalance overhead, computation, locality of data. Small problem size has large overhead. Large program has a lot of trashing on disk.

Amdahl's Law (1967) Speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelized (f).

- $f\,(0 \leq f \leq 1)$ is called the sequential fraction.
- Also known as fixed-workload performance

But Amdahl's Law can be circumvented for large problem size! f is not a constant, dependent on problem



$$S_p(n) = \frac{T_*(n)}{f \times T_*(n) + \frac{1-f}{p} T_*(n)} = \boxed{\frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}}$$

size.

Gustafson's Law (1988) There are certain applications where the main constaint is execution time. if f is not a constant but decreases when probelm size increases then $S_p\left(n\right) \leq p$

4 GPGPU

General Purpose GPU definitions:

- Device: GPU
- Host: CPU
- Kernel: function that runs on the device

A GPU has Multiple Streaming Multiprocessors (SMs). Each has memory and cache. one SM has multiple compute cores.

In a GPU, threads are very lightweight, very little overhead for creation, instant switching.

A CUDA kernel is executed by an array of threads. All threads run the same code, SPMD (Single Program, Multiple Data). Each thread has an ID that it uses to compute memory addresses and make control decisions. The threads need not be completely independent.

Threads are grouped up into blocks. In a block, it has shared memory, atomic operations and barrier synchronization. Threads in different blocks cannot cooperate. We can scale to any number of processor by increasing blocks. Hardware is free to schedule thread blocks to any processor at any time. Kernel scales across any number of parallel multiprocessors.

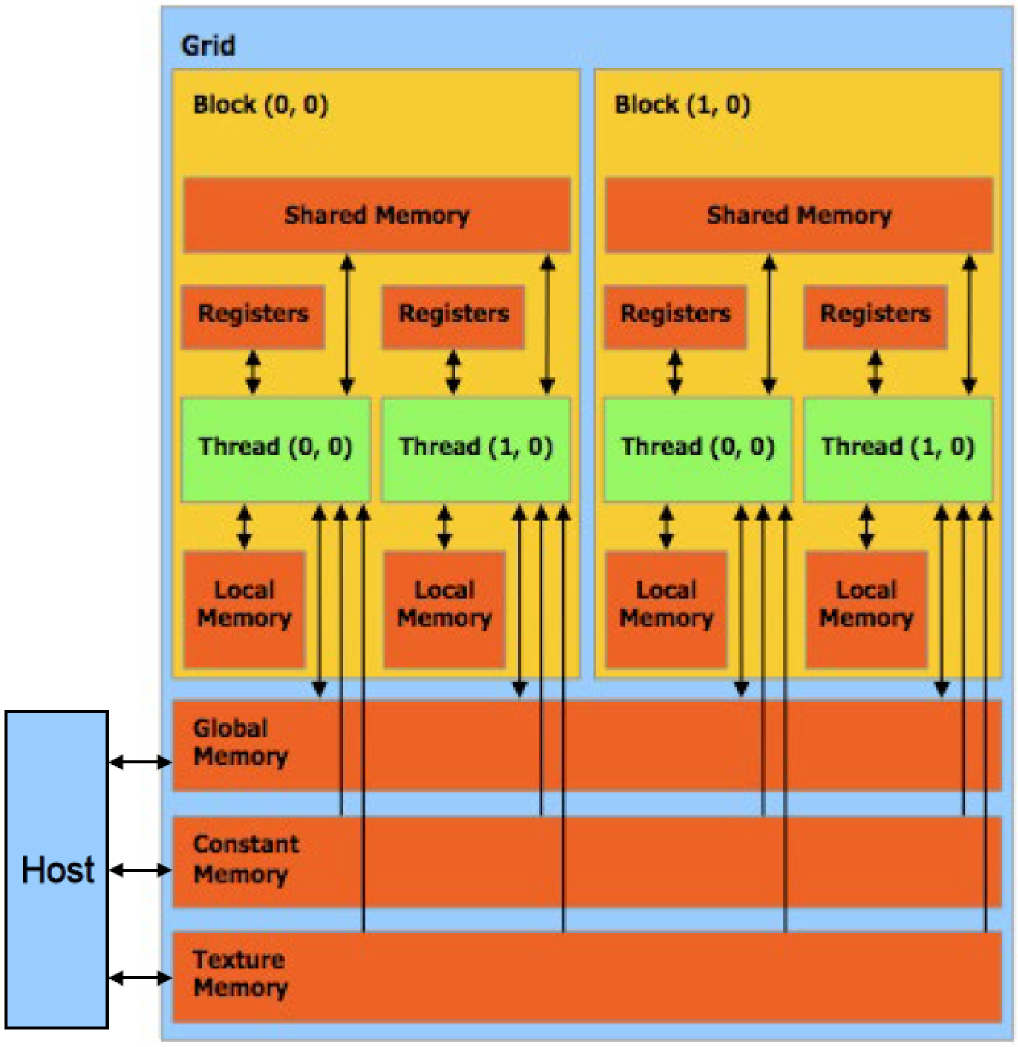
Each thread uses IDs to decide what data to work on. Can access block ID (1/2/3D), as well as thread ID. Users can specify organization of block and threads(1/2/3D)

A kernel is executed by a grid of thread blocks.

A block executes on one streaming multiprocessor. Does not migrate. Several blocks can reside concurrently on one SM. This is limited by SM resources.

Multiprocessor creates and manages, schedules and executes threads in SIMT warps(group of 32 parallel threads). Threads in a warp start together at the same program address. Each thread has individual instruction program counter and register state. A block is always split into warps the same way, a warp is consecutive threads in increasing IDs. In one warp, only one common instructuion is done in all threads. Avoid branching.

4.1 CUDA Memory Model



Type	Scope	Access type	Soeed	CUDA declaration syntax	Explicit sync
register	thread	RW	fastest	-	no
Local	thread	RW	depends*	float x	no
Shared	block	RW	fast	__shared__ float x	yes
Glocal	program	RW	slow	__device__ float x	yes
Constant	program	R	slow	__constant__ float x	yes
Texture	program	R	slow	__texture__ float x	yes

- Local memory is actually an abstraction of global memory that is private to the thread. So it is slower than shared memory.
- Shared memory has higher bandwidth, and lower latency than global and local memory. It is divided into equally sized momory modules, banks. Different addresses from different banks can be accessed simultaneously. if in the same bank, need to be serialized.
- Constant memory is useful for uniformly-accessed read-only data. It is cached
- Texture memory is useful for spatially coherent random-access read only data. It is cached. Provides filtering, address clamping and wrapping.
- simultaneous accesses to global memory by threads in a half-warp (16 threads) can be coalesced into as few as a single meory transaction of 32 / 64 / 128 bytes.

- when accessing global memory, it is 32 / 64 / 128 consecutive bytes segments are accessed together at one time.
- If all lie in the same region, one memory access, else need multiple transactions

5 Cache Coherence and Memory Consistency

5.1 Cache Coherence

Cache is used to reduce memory access latency.

- *Cache Size*: larger cache increases access time, increased addressing complexity. but reduces cache misses
- *Block Size*: data is transferred between main memory and cache in blocks of fixed length
  - Larger blocks reduce number of block but replacements take longer
  - Larger block increases probability of spacial locality

How do we write to cache and memory:

- Write-Through
  - write access is immediately transferredto main memory
  - Advantage: always gets neweest value of a memory block
  - Disadvantage: slow down due to many memory access (use write buffer)
- Write-Back
  - write operation is only on cache, written to main memory when cache block is replaced. Uses a dirty bit.
  - Advantage: uses less write operations
  - Disadvantage: Memory has invalid entries

Cache Coherence problem is when multiple entries of the same data exists on different cache. One processor updates the data, other processors may still see unchanged data. Issue arrises because we have one global memory, and many local memory.

5.1.1 Memory Coherence

Three properties of memory Coherence

- **Program Order**
  - Given the sequence
    1. p write to $\mathbf{x}$
    2. No more writes to $\mathbf{x}$
    3. p reads from $\mathbf{x}$
  - p should get the value written in step 1
- **Write Propagation**
  - Given the sequence
    1. p_1 write to $\mathbf{x}$
    2. No more writes to $\mathbf{x}$
    3. p_2 reads from $\mathbf{x}$
  - p_2 should read value written in by p_1
  - Writes become visible to other processors
- **Write Serialization**
  - Given the sequence
    1. write v_1 to $\mathbf{x}$, by any processor
    2. write v_2 to $\mathbf{x}$, by any processor

  - any processor can never read **x** as v_2 and then later v_1
  - ALl writes to the same location (from same or different processors) are seen in the same order by all processors

Cache coherence can be solved using

- Software Based Solution
  - OS + compiler + Hardware aided solution
- Hardware Based Solution
  - Most common on multiprocessor system
  - Known as cache coherence protocols

Major tasks for Hardware Cache Coherence Protocols:

- Track the sharing status of a cache line
- Handle the update to a shared cache line

There are 2 major catagories of solutions:

- Snooping based:
  - No centralized directory
  - Each cache monitors or snoop on the bus, to update status of cache line and take appropiate action
  - Most common protocol used in architecture with a bus
- Directory based:
  - Sharing status is kept in a centralized location
  - commonly used with NUMA architectures

Bus based cache coherence:

- All the processors on the bus can observe every bus transaction, *Write Porpagation*
- Bus transaction are visible to the processors in the same order, *Write Serialization*

The cache controllers "snoop" on the bus and monitor transactions, takes relevant actions. Some issues that arrise due to cache coherence:

- Overhead in shared address space:
  - Cache Coherence appears as increased memory latency in multiprocessor
  - Cache Coherence lowers the hit rate in cache
- False Sharing
  - 2 processors write to different address in the same cache line

5.2 Memory Consistency Models

4 types of memory operation ordering:

1. $W \to R$
2. $R \to R$
3. $R \to W$
4. $W \to W$

These 4 need not be preserved. We can relax some restrictions, Relax consistencies, to hide write latencies. If there is no data dependency in one processor.

Some memory models:

- Sequential Consistency *SC*
  - Every processor issues its memory operations in program order
  - Extension of uniprocessor memory model, intuitive but can have loss in performance
  - As if one memory space, ne memory operation at one time
- Total Store Ordering *TSO*
  - Return the value written by P ealier without waiting for it to be serialized
  - Relaxes $W \to R$
  - Does not ensure *Write Serialization*
  - Data dependencies in one program must be preserved!
- Processor Consistency - *PC*
  - Return the value of any write (even from another processor) before the write is propagated or serialized
  - Relaxes $W \to R$
  - Does not ensure *Write Serialization* and *Write Propagation*
  - Data dependencies in one program must be preserved!
- Partial Store Order - *PSO*
  - Write can bypass earlier writes (to different locations) in write buffer. Allows write miss to overlap and hide latency
  - Relaxes $W \to R$, similar to TSO, and $W \to W$
  - Data dependencies in one program must be preserved!

The last 3 above are relaxed consistencies.

6 Parallel Programming Models - II

6.1 Data Distribution

There are many ways to distribute data on multiple processors. Known as data distribution / work distribution / decomposition / partitioning. Problems that exhibit data parallelism, data distribution can be used as a simple parallization strategy. Commonly distributed Blockwise or cyclic or some combination of the two.

6.2 Information Exchange

To communicate between different processes for coordination in a parallel process. For Shared address space, usually used **shared variables**. For distributed address space, use **communication operations**.

Shared memory programming models assume a global memory accessible by all processors. programmer responsible to ensure safe concurrent access. Each thread executed by one processor or core, have shared variables and may have private variables. Uses critical section such as mutex.

In distributed memory programming we assume disjoint memory space. Need to use dedicated communication operations. one sends, one receives. known as message-passing programming model. data exchanges can be point-to-point or global. Data is explicitly partitioned into each process. All interactions need both parties to participate. Programmer needs to explicitly express parallelism.

Principles of Message Passing Model: it is loosely synchronous paradigm. Tasks or subsets of tasks synchronize to perform interactions. between there interactions tasks execute completely asynchronous.

6.3 Communication Protocol

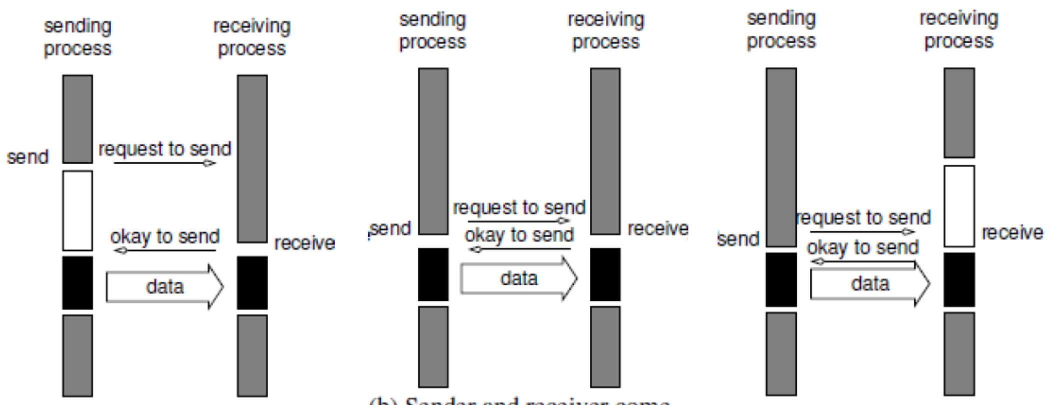
6.3.1 Send and Receive Operations

one process sends to another process. In a distributed memory system, over a network. THis is a one way transfer.

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

In a blocking operation, send operation blocks until it is safe to reuse the input buffer. "Safe" refers to the integrity of the data to be sent.

Non-buffered blocking send: the operation blocks until the matching receive operation has been performed by receiveing process. Can lead to idling and deadlocks. There are idling overhead, waiting sender and receiver to be ready

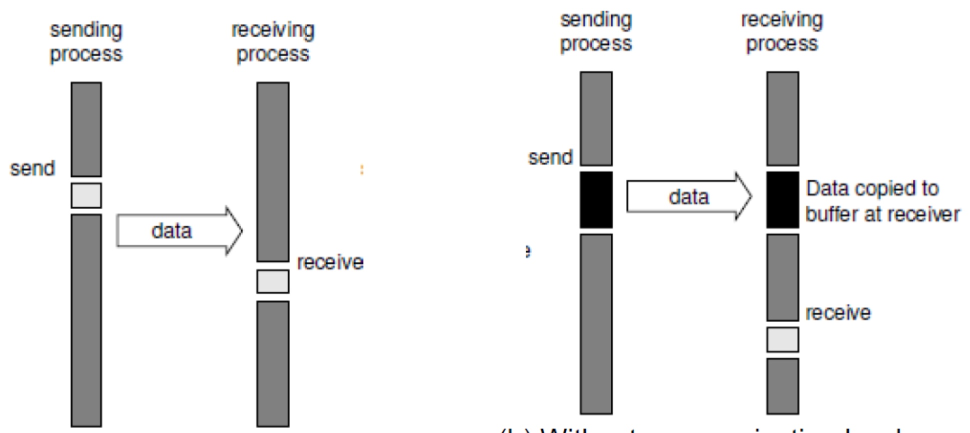


(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

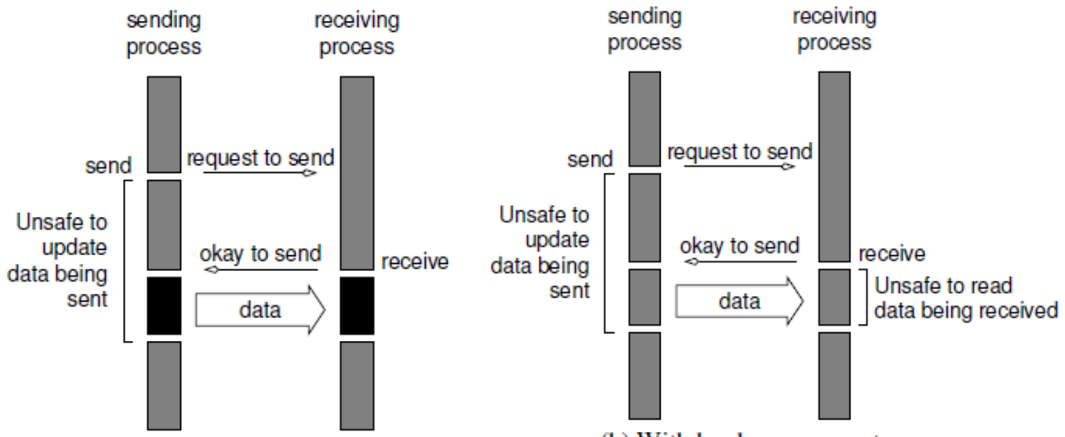
To reduce the idling overhaed, we use buffers. Send data to designated buffer and return after the copy operation has been completed. Receiver similarly buffers the incoming data. Buffering trades off idling overhead for buffer copying overhead.



(a) Using communication hardware with buffers at send and receive ends

(b) Without communication hardware, sender interrupts receiver and deposits data in buffer at receiver end

In Non-blocking operation, Send/Receive returns before it is semantically safe to do so. Non-blocking operations are generlly acompanied by a checking mechanism. Programmer checks if it is safe to use data. If used correctly, can overlap communication overhead.



(a) Without hardware support

(b) With hardware support

Semantics of Send/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

7 Message Passing

Coding see lecture slides.

7.1 Collective Communication

There are many communicatoin operations: single tranfer, gather (scatter) Typically is like this: sencer (root) sends to slave processeors to do work. then gather accumulate their results. Processors are given an id / rank.

- Single broadcast: Sender (root processor) sends the same data to all other processes.
- Multi broadcast: Each processor sends the same data block to every other processor. NO root processor. Data blocks are collected in rank order
- Single accumulation (Gather with reduction): Each processor provides a block of data with the same type and size. A reduction (binary, associative and commutative) operation is applied element by element to the data blocks. Root process receives result.
- Multi accumulation: Each processor provides for every other processor a potentially different data block.Data blocks for the same receiver are combined with a given reduction operation. No root processor.
- Total Exchange: Each processor provides for each other processor a potentially different data block. Effectively each processor executes a scatter operation. No root processor

8 Interconnection

Interconnection network is cool.

8.1 Topology

What is the geometrical shape of the connection.
There are 2 major types of Topology:

- Direct Interconnection
  - Static or point-to-point
  - endpoints are usually the same (core, memory)
- Indirect Interconnection
  - Dynamic
  - Interconnect formed by switches

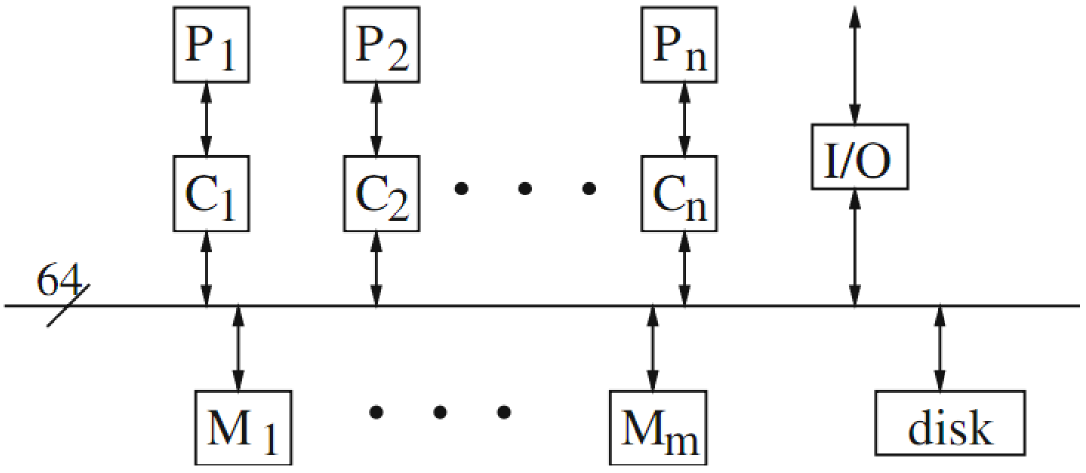
We can model this as a graph. $G = (V, E)$, V = vertices / node, E = edges We are concerned about:

- Diameter $\delta(G)$
  - Maximum distance between any pair of nodes
  - small diameter ensures small distances for message transmission
- Degree
  - $g(v)$: number of direct neighbour nodes of a node v
  - $g(G)$: maximum degree of a node in a network G
  - small node degree reduces the node hardware overhead
- Bisection Width $B(G)$
  - minimum number of edges that must be removed to divide the network into 2 equal halves
  - Bisection Bandwidth $BW(G)$: Total bandwidth available between the two bisected portion of the network.
  - A measure for the capacity of a network when transmitting messages simultaneously
- Connectivity
  - Node Connectivity $nc(G)$: minimum number of nodes that must fail to disconnect the network.
  - To determine the robustness of the network
  - Edge Connectivity $ec(G)$: minimum number of edges that must fail to disconnect the network.
  - Determine number of independent paths between any pair of nodes

8.2 Indirect Interconnection

Reduce hardware cost by sharing switches and links. Use switches to provide indirect connection between nodes and can be configured dynamically. metrics to consider are cost / number of switch/link and concurrent connections.

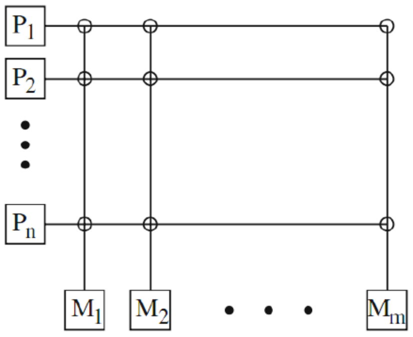
8.2.1 Bus network



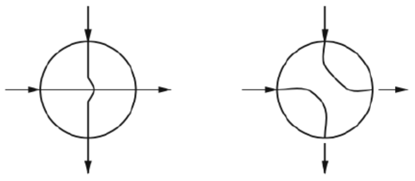
A set of wires to transport data from a sender to a receiver. Only one pair of devices can communicate at a time. A bus arbiter is used for the coordination. Typically used for a small number of processors.

8.2.2 Crossbar network

A $n \times m$ crossbar network has n inputs and m outputs.



n x m crossbar network

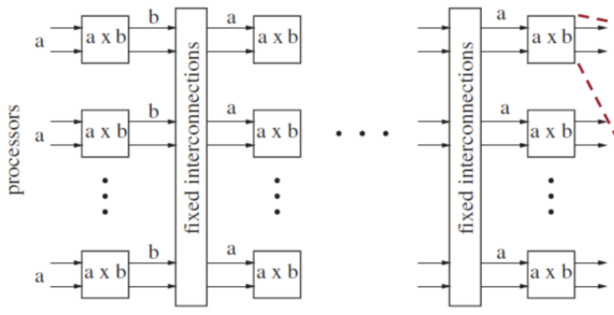


Straight state Direction change state

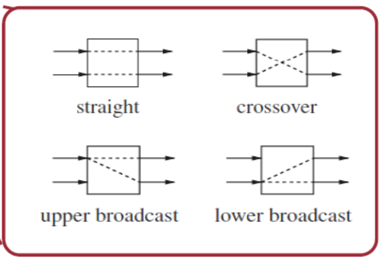
The switches can either be straight or direction change. But it is expensive. $n \times m$ switches for few number of processors.

8.3 Multistage Switching Network

Several intermediate switches with connecting wires between neighbouring stages. Goal: obtain a small distance for arbitraty pairs of input and output devices. each switch has 2 input and 2 outputs.



Multistage Networks using Regular 2x2 Crossbar Switches

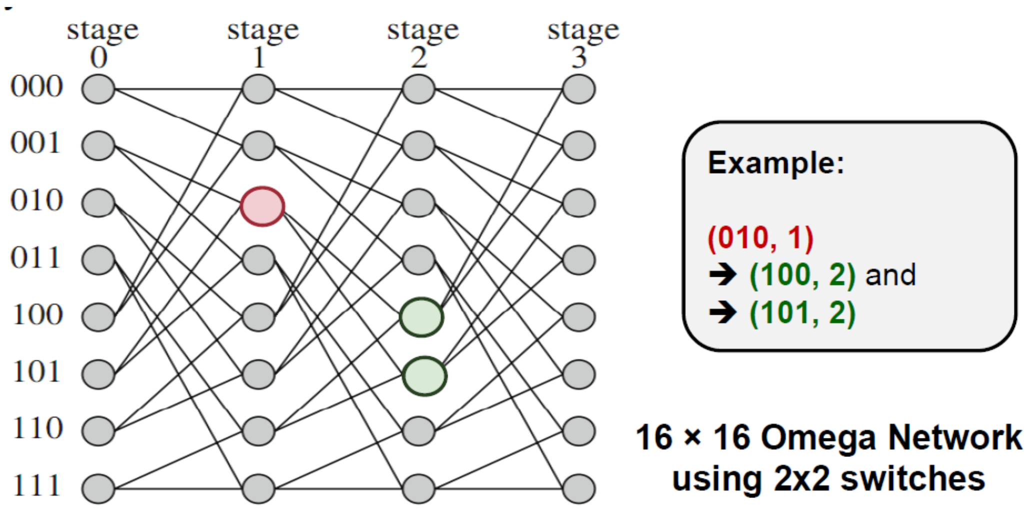


2x2 Switch Settings

8.4 Omega Network

One unique path for every input to output an $n \times n$ omega network has $\log n$ stages and $n/2$ switches per stage. The connections between stages are regular. Also known as $(\log n - 1)$ - dimension Omega Network

- A switch position (α, i), where α is the position of a stage within a stage; i: stage number
- 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 LSBit



In a 16 processor to 16 memory nodes, crossbar needs $16 \times 15 = 256$ switches. Omega needs $\log n \times n/2 = 4 * 8 = 32$ switches.

8.5 Butterfly Network

- Node (α, i) connets to
  - $(\alpha, i+1)$, i.e. straight edge
  - $(\alpha', i+1)$, α and α' differ in the $(i+1)$th bit from the left, i.e. cross edge

8.6 Baseline Network

- A switch position (α, i), where α is the position of a stage within a stage; i: stage number
- Has an edge from node (α, i) to two nodes $(\beta, i+1)$ where
  - $\beta =$ cyclic right shift of last $(k-i)$ bits of α
  - $\beta =$ inversion of the LSBit of α + cyclic right shift of last $(k-i)$ bits

8.7 Routing

How to go from processor to memory.
Routing algorithms Classifications

- Based on path length
  - Minimal or Non-minimal routing: whether the shortest path is always chosen
- Based on adapticity
  - 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.

Some deterministic routing examples:

1. 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 $Y_{\text{src}} == Y_{\text{dst}}$
2. E-Cube Routing for Hypercube
   - Let $(\alpha_{n-1}\alpha_{n-2}\ldots\alpha_0)$ and $(\beta n - 1 \beta n - 2 \ldots \beta)$ be the bit representations of source and destination node address respectively.
   - Number of bits difference in source and target node address means number of hops, also known as 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, at most n hops.
3. XOR-Tag Routing for Omega Network
   - 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

9 Definitions

Blocking call: Control returns only when the call completes.

Non blocking call: Control returns immediately. Later OS somehow notifies the process that the call is complete.

Synchronous program: A program which uses Blocking calls. In order not to freeze during the call it must have 2 or more threads (that's why it's called Synchronous - threads are running synchronously).

Asynchronous program: A program which uses Non blocking calls. It can have only 1 thread and still remain interactive.

Concurrent and parallel are effectively the same principle as you correctly surmise, both are related to tasks being executed simultaneously although I would say that parallel tasks should be truly multitasking, executed "at the same time" whereas concurrent could mean that the tasks are sharing the execution thread while still appearing to be executing in parallel.

Asynchronous methods aren't directly related to the previous two concepts, asynchrony is used to present the impression of concurrent or parallel tasking but effectively an asynchronous method call is normally used for a process that needs to do work away from the current application and we don't want to wait and block our application awaiting the response.