# CS251B: Parallel Computer Architectures

Professor Tamir

Thilan Tran

Winter 2022

## Contents

**CS251B: Parallel Computer Architectures** 3

**Multiprocessors** 4

**Interprocess Communication** 5
- Caches . . . 6
  - Invalidate Protocol Implementation . . . 9
  - Directory-Based Cache Coherence . . . 12

**Synchronization** 15
- Locks . . . 16

**Interconnection Networks** 18
- Topology . . . 19
  - Multistage Interconnection Networks . . . 20
  - Fault Tolerance . . . 23
- Implementation . . . 24
- Switching . . . 28
- Flow Control . . . 30
- Forwarding and Latencies . . . 31
- Routing . . . 33
  - Livelock and Deadlock . . . 34
- Case Study . . . 36

**Message Passing** 38
- Design Space . . . 39

**SIMD and Vector Architectures** 43

Pipelined SIMD . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
Vectorization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
Parallel SIMD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**GPUs** **49**

# CS251B: Parallel Computer Architectures

- different definitions for **parallel systems**:
  - a system that supports the execution of multiple operations simultaneously
    - too broad a definition, a uniprocessor does perform concurrent hardware operations
  - a large collection of processing elements that can communicate and cooperate to solve large problems fast
- two main components of computer architecture are the ISA and machine organization:
  - parallel systems have an *explicit* specification of parallelism in their ISA:
    - as well as communication between hardware units
    - the ISA in single core processors do not support these
  - machine organization is the actual hardware
    - multiple functional units where multiple operations can be executed simultaneously
- limitations of uniprocessors:
  - upper bounds on individual functional units
  - interconnect latencies dominates
  - limited parallelism in sequential code
    - hardware is greatly constrained
  - but when performance is important, parallelism is often easy to identify and explicitly specify
  - does not efficiently address "big data" demands as cloud computing and SaaS becomes more prevalent
  - decreased efficiency
    - increasing marginal cost for added performance
  - power consumption is a huge limitation
    - with multiple slower processors, can achieve the same performance with lower power usage
- **Flynn's classification** is a classic way to classify parallel systems:
  - a **uniprocessor** is **single instruction**, **single data (SISD)** system
  - a **single instruction**, **multiple data (SIMD)** system e.g. GPU
    - vector or array architectures
  - a **multiple instruction**, **single data (MISD)** system e.g. systolic array
    - only a single data stream, but multiple operations are performed on it
  - a **multiprocessor** is **multiple instruction**, **multiple data (MIMD)** e.g. shared memory

# Multiprocessors

---

- multiprocessors use multiple processors instead of a single one
  - allows for improved performance
  - usually on a single chip AKA **chip multiprocessor (CMP)** or multicore chip per Intel
    - **uncore** refers to all hardware besides the actual cores themselves
- two typical workloads:
  - in a multi-programmed workload, we have multiple independent applications
    - each runs on a different processor
  - in a parallel application AKA multi-threaded workload, we have a single application
    - but partitioned into multiple **threads**, each runs on a different processor

# Interprocess Communication

---

- with multiple processors working on the same computation, a thread on one processor needs results from a thread on another processor
- requires **interprocess communication** mechanisms, such as explicit message passing or shared memory
- **shared memory** functionality:
    - all processors read / write from / to the same memory
    - all processors have the same view of memory
    - a read from some address `x` resolves from the most recent write to `x` from any processor
    - *pros*:
        * sharing data without storage overhead of replication
        * passing memory references simplifies sharing of complex data structures
        * simplifies selective parallelization of application hot spots
            · easy to parallelize critical parts of sequential code
    - use hardware arbitration so each processor gets its turn to use memory?
        * but processors spend too much time waiting for the memory bus
        * the performance solution would be to use a fast L1 cache for each processor to prevent resource contention
            · problem becomes keeping the caches coherent and consistent
        * desired functionality vs. desired performance
    - in modern implementation, for a multicore chip:
        * private L1 and L2 cache for each core
        * connected to an interconnection network
        * large single shared L2 cache for all cores
    - then, multichip systems connect multiple multicore chips together:
        * over an inter-chip interconnection network
        * different chips may have faster connection to different parts of memory
            · non-symmetric access time
- four general shared memory hierarchy approaches:
    1. shared cache
        - single shared cache and memory for all processors
    2. bus-based shared memory
        - private cache for each processor, attached to a shared bus and memory
    3. dancehall

   - private cache for each processor, attached to a scalable point-to-point interconnection network with memory modules
4. distributed memory
   - private cache and main memory portion for each processor, attached to an interconnection network (non-symmetric)

- specifically, in a **symmetric multiprocessor (SMP)**, all processors have **uniform access time (UMA)** to main memory:
  - symmetric relationship between memory and all processors
  - less contention on the buses, so throughput increases
- what are the correct semantics for processors?
  - in a uniprocessor, the result of execution is same as if the operations had been executed in program order i.e. a **sequential** uniprocessor
  - in a multiprocessor, depends on the **memory model** i.e. the permissible relationships between the order in which accesses are initiated and the order these accesses are observed:
    * AKA memory consistency, memory consistency model
    * thus expected behavior depends on hardware, system software, and compiler
      · all could have unique memory models
  - usual correctness semantics for a multiprocessor:
    * operations of all processors are executed in some sequential order, and operations of each individual processor appear as if done in program order (consider side effects) i.e. a **sequentially consistent** multiprocessor
      · sequentiality of each individual processor *does not* guarantee the multiprocessor is sequentially consistent
    * additional necessary requirements over uniprocessor correctness:
      1. each processor issues memory requests in program order
      2. memory requests from all processors issued to an individual memory model are serviced from a *single* FIFO queue
    * i.e. each core executes in-order, and the memory bus arbitrates access one by one

## Caches

---

- uniprocessor caches are **functionally transparent**:
  - a programmer cannot determine whether and where a system caches by analyzing the results of a program
  - however, program timing can reveal this

- with multiprocessors, caches provide both migration and replication of shared data items:
    - migrating from main memory to cache to lessen memory bandwidth demand
    - replicating shared data across cores that are reading it simultaneously
- these caches add complications to maintaining the memory model:
    1. data sharing
        - eg. writeback caches are not up to date between processors
    2. process migration
        - process switching invalidates caches
    3. I/O
    - simple solutions:
        * make shared data non cacheable
        * flush cache when process migrations and I/O
        * high performance costs
- **coherence** is, intuitively, where the caches are functionally transparent in a multiprocessor:
    - deals with the behavior of reads and writes to the *same* address
        * alternatively, defines what values can be returned by a read
    - want the global state i.e. main memory to be consistent with local state i.e. caches
    - should only process a single writer or multiple readers at a time
    - through **write propagation**, a write must eventually be seen by reads to the same address
    - through **write serialization**, writes to the same location are seen in the same order by all processors
    - invariants used in an equivalent definition:
        1. for any memory location, in any given epoch, there exists only a single core that may write and read it, or zero or more cores that can only read it
        2. the value of the memory location at the start of an epoch is the same as the value of the location at the end of its last read-write epoch
- **consistency** defines the behavior of reads and writes with respect to accesses to other memory locations:
    - system follows well-defined semantics with respect to accesses to *all* locations
        * alternatively, determines when a written value will be returned by a read
    - coherence is usually part of the consistency model
- multiprocessor **cache coherence protocols** allow for coherency:
    - protocol must track the state of any sharing of a data block
        * caches maintain state for every block

1. **directory-based** protocols keep the sharing status of blocks in a single location called the **directory**
   - consult directory before operating on shared blocks
2. in **snooping** protocols, every cache that has a copy of the data from a block could track the sharing status:
   - snoop controllers monitor or *snoop* on the shared broadcast medium connecting the core caches to determine whether they have a copy of a block that is requested on a bus or switch access
     * in addition to cache controllers communicating with the processor
   - block states are distributed among the caches allowing for parallel lookup

- types of snooping coherence protocols:
  1. in a **write invalidate protocol**, a processor has exclusive access to a data item before writing that item:
     - i.e. invalidates other copies on a write
     - then, when another processor tries to read the invalidated value from cache, the processor with the updated cache will snoop and respond with the value:
       * i.e. canceling the response from memory
       * if write-back, cache needs to respond, otherwise if write-through, could have gone to memory
     - finally, memory will be updated here if a write-back cache
       * can also force the write-back only if the block is replaced, but then we need an additional status bit for ownership of a block
     - note that if we have false sharing within a block between processors, we have degraded performance
       * need parallel-aware compiler
     - *pros*:
       * requires only one bus transaction per write run, better bandwidth
       * benefits from spatial locality
     - *cons*:
       * more latency between writes and reads, have to invalidate all copies and issue a read miss
  2. in a **write update / broadcast** protocol, all the cached copies of a data item are updated on a write:
     - takes much more bandwidth, must broadcast all writes to shared cache lines
     - *pros*:
       * lower latency between writes and reads
     - *cons*:
       * more bus activity and contention for the bus

    - famous implementation is the Dragon protocol
      * uses similar MESI states to optimize the broadcasts
    - most modern systems do not use write update

**Invalidate Protocol Implementation**

- key to implementation is to use the broadcast medium e.g. bus or shared cache connection to perform invalidates:
  1. to invalidate, simply acquire bus access and broadcast the invalidated address on the bus:
     - writes will be serialized when acquiring bus access
     - could be aware if block is even copied in other caches, otherwise there is no need to broadcast the write
       * can use an additional shared bit
  2. all processors continuously snoop on the bus
     - if the block number on the bus is in their cache, the corresponding data is invalidated via the valid bit
  3. locating data item on cache miss:
     - for write-through, just go to memory
     - for write-back, use a similar snooping scheme
       * if the processor has a dirty copy of the requested block, it provides the block on the bus and aborts the memory access
- uniprocessor cache controller:
  1. check if cache has block
     - check tags and valid bit
  2. if not, find a victim, and write-back if dirty
     - then get the block
  3. allow access to proceed
- multiprocessor cache controller:
  1. check if cache has block
  2. if not, find a victim, and write-back if dirty:
     - then get the block
     - if yes and we are writing, need to broadcast an invalidate
       * can additionally require an additional shared / exclusive bit so we only broadcast if others have copies
  3. allow access to proceed
  - write-backs, block retrieval from memory, and broadcasting are all **bus transactions**
    * transactions also send exact block number
- snoop controller:
  1. check if cache has block
  2. if invalidate, invalidate:
     - if read, supply block if exclusive i.e. modified owner

        * only this cache has the updated block
    - if write, write-back if modified and invalidate
- some definitions:
    - **validity** definition is the same as in a uniprocessor
    - a cache is the **owner** of a block if it must supply the data upon a request for that block
        * i.e. owner has responsibility
    - a cache has an **exclusive** copy of a block if it is the only cache with a valid copy of a block
        * exclusivity implies the cache can modify the block without notifying anyone else
    - a **modified** block is dirty
        * the cache it is in is the owner *and* it has exclusivity
- simple protocol with three states, as seen in Figure 1:
    - **shared** state indicates block in private cache is potentially shared:
        * main memory is up to date, shared i.e. shared-unmodified
        * on processor read hit, simply read data in cache
        * on processor read miss, we have an address conflict miss, place read miss on bus
        * on processor write hit, we need to place invalidate on bus
        * on processor write miss, we have an address conflict miss, place write miss on bus
    - **modified** state indicates block is dirty:
        * implies block is exclusive, main memory is stale
        * on processor read hit, simply read data in cache
        * on processor read miss, we have an address conflict miss, write-back block and place read miss on bus
        * on processor write hit, simply update data in cache
        * on processor write miss, we have an address conflict miss, write-back block and place write miss on bus
    - **invalid** state as previously described:
        * on processor read miss, we have a normal read miss, place read miss on bus
        * on processor write miss, we have a normal write miss, place write miss on bus
    - actions by snooping processor when broadcasted by bus:
        * if read miss and block is shared, allow shared cache or memory to service read miss
        * if read miss and block is modified, place cache block on bus, write-back block, and change the state to shared
        * if invalidate and block is shared, invalidate the block
        * if write miss and block is shared, invalidate the block
        * if write miss and block is modified, write-back block and invalidate

the block
  - possibilities for each block of memory:
    * clean in some caches and up-to-date in memory
    * dirty in exactly one cache
    * not in any caches

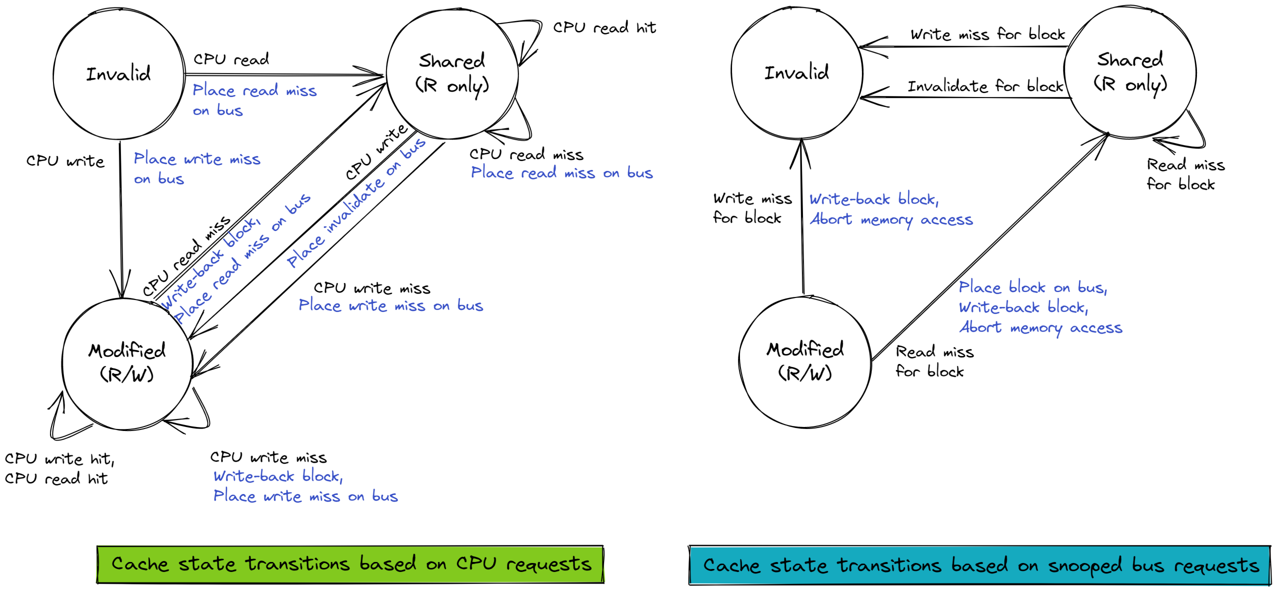


Figure 1: MSI Protocol FSMs

- this is a **MSI** protocol for its three states:
  - in a **MESI** protocol, we add the **exclusive** state:
    * indicates block is resident in only a single cache but is clean i.e. exclusive-unmodified
    * when exclusive but not owned, can be written without generating invalidates, without acquiring bus access
    * could be implemented with an additional shared bus line where caches report whether they have the requested block
  - in a **MOESI** protocol, we add the **owned** state:
    * indicates block is owned and out-of-date in memory, but still shared, i.e. shared-modified
    * prevents an unnecessary write out to memory in the case the block changes from modified to owned
      · allows for the case where written dirty blocks change between caches without updating main memory
  - in the Berkeley ownership protocol adds the owned state, but not the exclusive state
    * essentially a MOSI protocol
- implementing snooping caches:
  - in write-back, caches must reply to bus requests
  - need additional bits in the cache block to determine shared or not

  - every bus transaction checks cache tags:
    * could interfere with the CPU when checking the cache
    * bus transactions slow down the processor
    * *solutions*:
      · duplicate set of tags to allow checks in parallel with the CPU
      · second level cache that obeys inclusion
  - one complication is that operations are not atomic e.g. requesting or obtaining bus:
    * addressing the *transient* states that lie in-between major main states
    * e.g. multiple controllers invalidating block at the same time, or conflicting accesses to an owned-exclusive block
    * to solve this, cache controllers typically need to acquire the bus and reread block state on certain updates
    * alternatively, set a priority list to prevent the conflict
  - another complication is explicitly split transaction buses:
    * e.g. request followed by reply, address followed by data
    * one solution is to keep track of all outstanding requests, and wait until there are no outstanding requests for a block before requesting that block
    * alternatively, keep track of own outstanding requests, and after any response on bus for block in pending requests, discard reply when it arrives and re-issue
- performance of snooping caches:
  - we have an additional type of cache miss, **coherence misses**
  - true sharing vs. false sharing
    * want to minimize false sharing
  - increasing the number of processors can increase the coherency miss rate, but also decrease capacity miss rate
    * a "superlinear" speedup

**Directory-Based Cache Coherence**

- the major issue with snooping caches is scalability:
  - with heavy contention on a single shared bus, performance will degrade with additional nodes
  - instead, consider the directory-based approach:
    * in a shared directory, keep track of the state of each block in memory
    * who has a copy, is it modified, is there an operation on this block?
  - these directory-based schemes are classified based on:
    * the number of indices kept for each block in directory
    * and whether protocol is broadcast or no broadcast

    * e.g. $Dir_iB$
    * with no broadcast, $i$ is the max number of copies
  - practically, directory-based protocol with use distributed memory and distributed directories:
    * take advantage of a powerful interconnection network to connect all nodes, instead of a slower bus
    * typically use a mask on the address to determine which directory a block's state will be held in
- directory-based coherence scheme:
  - block state in the directory may be uncached, shared and who has copies, or exclusive and who has the block
  - block state in each cache may be invalid, shared, or modified
  - block location may be local node, home node (based on address), or remote node (from directory)
  - need to send various messages over network, similar to the bus in snoopy caches
    * e.g. read and write misses, invalidates, fetches, data replies and write backs
- using coherence directories as caches:
  - we do not need directory entries for uncached blocks
    * have a cache on the home directory for every block cached anywhere
  - no backing store for directory entries
  - blocks may have to be recalled from data / instruction caches if coherence directory cache entry is replaced
- e.g. Coherent HyperTransport protocol:
  - full MOESI, $Dir_0B$ protocol
  1. requestor messages home
  2. home broadcasts all nodes
  3. responsible node sends block to requestor
  4. requestor messages home to complete the transaction
- e.g. Intel QuickPath Interconnect (QPI):
  - MESIF, $Dir_1B$ protocol
    * additional forwarding state avoids memory access for clean blocks, access via the forwarding node instead
  - source snoop protocol:
    1. requestor broadcasts request to all
    2. each node sends a snoop response to home
    3. node with block in M, E, or F sends block to requestor
    4. once home has all responses, sends transaction complete (possibly with data) to requestor
    * transactions serialized by home based on when all snoop responses are received

  - home snoop protocol:
    1. requestor messages home
    2. home messages relevant nodes based on directory
    3. each node sends a snoop response to home
    4. node with block in M, E, or F sends block to requestor
    5. once home has all responses, sends transaction complete (possibly with data) to requestor
       * transactions directly serialized by home
  - note that all nodes need to respond in order for the home to have full information e.g. no one may have the block
  - source snoop protocol skips a transaction, since nodes respond to home and forward blocks in parallel
- implementation complications:
  - again, problems with non-atomic actions and no atomic broadcast
  - need to serialize requests
  - wait for explicit reply to complete operations
  - restart operation on explicit NACKs

# Synchronization

---

- a **synchronization operation** is an operation in which two or more threads exchange information to coordinate their activities
- **synchronization points** are points in the program where the control flow requires threads to interact with one another
- a **data race** occurs when two threads access a shared address:
  - *and* at least one access is a write
  - *and* accesses are not separated by a synchronization operation
  - the outcome of a datarace depends on relative processor speeds
  - outcome may depend on relative speeds even if there are no data races
    - e.g. who gets to a lock first
  - a **synchronized program** is one where no data races are possible
- synchronization mechanisms typically built with user-level software routines that rely on hardware supplied primitives:
  - key hardware capability is some kind of atomic and indivisible instruction for retrieving and changing a value:
    - the **atomic exchange** interchanges a value in a register for a value in memory
    - the **test-and-set** tests a value and sets it if the value passes the test
    - the **fetch-and-increment** returns the value of a memory location and atomically increments it
    - also compare-and-swap, increment, decrement
  - acts as a building block to build other user-level synchronization operations like locks and barriers
- implementing an atomic memory operation (read-modify-write):
  - hardware cannot allow any operations between the read and write, but cannot deadlock
    - simplest implementation, just hold onto the bus
  - alternatively, use a pair of instructions:
    - second instruction returns a value from which it can be deduced whether the instructions were executed as though they were atomic
    - effectively atomic if it appears as though all other operations occurred before or after the pair
    - at no time do we have to hold the bus, can just loop the instruction pair until it succeeds
      - much more desirable for multiprocessors
  - RISC-V instruction pair approach:

    * **load reserved** AKA **load linked** instruction loads the memory contents given by `$rs1` into `$rd` and creates a reservation on that address
        · operates like a normal load
    * **store conditional** instruction conditionally stores the value in `$rs2` into the memory address given by `$rs1` :
        · if reservation is broken by a write to the same memory location, store conditional fails and writes a non-zero to `$rd`
        · e.g. a context switch between the instructions will always make the store conditional fail
    * reservation is implemented by keeping track of `lr` address in a **reserved register**:
        · if an interrupt occurs or the cache block is invalidated by another write, the link register is cleared
        · store conditional simply checks if the desired address matches the `lr` address
    * have to be careful which instruction can be inserted between the pair
        · e.g. register-register instructions are safe

RISC-V atomic memory operations:

```
; atomic exchange of memory location contents at $x1 with value in $x4
try:    mov  x3, x4
        lr   x2, x1
        sc   x3, 0(x1)
        bnez x3, try
        mov  x4, x2

; atomic fetch-and-increment
try:    lr   x2, x1
        addi x3, x2, 1
        sc   x3, 0(x1)
        bnez x3, try
```

## Locks

---

- with an atomic operation, we can implement basic **spin locks**:
    - *pros*:
        * spin locks are used when lock is expected to be held for a very short amount of time
            · otherwise, probably better to use a sleep-wakeup queue

    * locking process is also low latency
  - *cons*:
    * spinning ties up the processor i.e. busy wait
  - with no cache coherence, we can keep the lock variables in memory
  - with cache coherence, we can cache the locks using **test and test-and-set**:
    * only try to acquire the lock if there is a good chance we will actually succeed in getting the lock
      · i.e. first test by reading
    * *pros*:
      · spinning check is faster
      · locality in lock accesses
      · avoids overflooding the bus with invalidates
    * now, we have to spin by reading a local copy of the lock
    * then, can attempt to acquire the lock with a swap
    * when done, we broadcast an invalidate which leads to a race between the other processors to reacquire the new cached lock value

RISC-V spin lock examples:

```
; spin lock on memory address in $x1, no cache coherence
        addi x2, x0, 1
lockit: EXCH x2, 0(x1)  ; atomic exchange
        bnez x2, lockit ; already locked?

; spin lock with cache coherence
lockit: ld   x2, 0(x1)  ; load lock (don't need write permission, can check cache)
        bnez x2, lockit ; not available, spin
        addi x2, x0, 1  ; load locked value
        EXCH x2, 0(x1)  ; atomic exchange
        bnez x2, lockit ; branch if lock wasn't 0

; simple with LR/SC
lockit: lr   x2, 0(x1)
        bnez x2, lockit
        addi x2, x0, 1
        sc   x2, 0(x1)
        beqz x2, lockit
```

# Interconnection Networks

---

- **interconnection networks** connect together individual end nodes i.e. devices into a network of communicating devices:
  - i.e. allow for information transfer from any source node to any desired destination node
    - nodes are anything from multiple computers to components like memory and I/O modules in a single computer
  - unlike traditional networks, on a *parallel* machine, we value above all:
    - small latency (rather than throughput, for example)
    - large number of concurrent transfers
      - redundant paths and routing
    - satisfying limited geographic and physical constraints
  - switched networks are gradually replacing even buses
  - main domains:
    - **on-chip networks (OCNs)** are used for interconnecting microarchitecture modules such as register files, caches, cores, etc.
      - mitigate chip-crossing wire delay problems
    - **system / storage area networks (SANs)** are used for inter-processor and processor-memory interconnections for servers and data centers
    - **local area networks (LANs)** are used for autonomous computer systems distributed throughout a building
    - **wide area networks (WANs)** connect computer systems across the globe
- network definitions:
  - **links** are the wires or fibers that carry analog signals
    - while **channels** carry digital symbols
  - **hosts** or **terminal nodes** generate and remove traffic, while **switches** move traffic along i.e. connect input to output channels
  - the **topology** of a network is the physical interconnection structure of the network graph
    - can be regular or irregular, direct or indirect
  - the **routing algorithm** determines which routes messages follow through the graph
  - the **switching strategy** determines how the data traverses its route:
    - in **circuit switching**, the path is reserved until the message is transfered
    - in **packet switching**, the message is broken into a series of individually routed packets
  - the **flow control mechanism** determines when the message moves

along its route i.e. arbitration between network resources
  - in a **direct network**, every node is both a switch and a terminal node
    * i.e. nodes are directly connected to other nodes without going through some external switch or switch fabric
  - in an **indirect network**, every node is either a switch or a terminal node

## Topology

---

- two main kinds of networks:
  - **shared media** networks are like shared buses:
    * *pros*:
      · low cost
    * *cons*:
      · low performance and bandwidth
      · more contention
  - on the other hand, **switched media** networks have individual parts called **switches** that select between nodes:
    * *pros*:
      · supports multiple simultaneous operations
      · high scalable bandwidth
    * *cons*:
      · higher cost
- if our required number of ports is small enough, we can use a single crossbar switch:
  - otherwise, we need to use a **switch fabric** of interconnected switches
  - this interconnection structure is the network topology
  - most important network consideration for SANs and OCNs
    * greatly impacts routing, packaging, and scalability
  - can be implemented with a grid of connections, multiple multiplexers, intermediary memory, etc.
- the single stage **crossbar switch** in which every input is directly connected to every output in one large switch:
  - there are $n^2$ **crosspoints**
  - *pros*:
    * nonblocking topology
  - *cons*:
    * quadratic scaling
    * not modular, difficult to add ports
- the single stage **shuffle exchange** network shuffles the inputs into switches that connect pairs of inputs:
  - i.e. a perfect shuffle of a deck of cards, and then switches can be toggled

    between "straight" or "crossed"
  - i.e. mathematically, if we consider ports as binary addresses, shuffle is a left rotate on the bits, exchange flips the value of the least significant bits
  - can we achieve **full connectivity** if we have a *recirculating* shuffle exchange?
    * note that with only a recirculating shuffle or exchange alone, we cannot achieve full connectivity
      · can check on mathematical definition
    * with both shuffle and exchange, full connectivity is possible
    * worst case number of recirculations is based on the number of bits in the bit representation
- the single stage **k-ary n-cube** performs radix computations:
  - $N = k^n$ nodes with $k$ nodes in each dimension, with degree $d = 2n$
  - each node address is an $n$-digit radix $k$ number
  - the interconnection functions allows us to go up or down along each dimension
  - could simulate the functionality with recirculating shuffle exchange
    * need to flip one of the bits

**Multistage Interconnection Networks**

- instead of a recirculating approach, we can also divide the process into several stages in a **multistage interconnection network (MIN)**:
  - rather than recirculating, replicate in space rather than in time:
    * while still allowing any destination to be accessible from any source (full connectivity)
    * typically achieves logarithmic scaling:
      · $N$x$N$ network, with $M$x$M$ switches, each stage has $\frac{N}{M}$ switches, need $k$ stages to connect to $M^k$ ports
      · thus we need $k = \log_M N$ stages and $\frac{N}{M} \log_M N$ switches
      · but more contention and lower performance than crossbar
  - note that each individual switch is a subnetwork that implements its own interconnection functions
    * e.g. straight or crossed for a $2 \times 2$ switch
  - the interconnection patterns between stages are essentially a set of mathematical functions
  - to reduce blocking:
    * want **rearrangeably nonblocking** networks where nonconflicting paths to new source-destination pairs can be established
    * can add either extra stages to mirror the original topology
    * alternatively, add larger switches in the middle of other stages so that alternative paths are created

  - to create bidirectionality, can fold together these symmetric networks
- e.g. the **Omega** topology uses multiple stages of shuffle exchange:
  - although we can achieve full connectivity, we need to consider how many permutations are possible
    * i.e. with all permutations, we can pair together inputs and outputs in any one-to-one way, at the same time
  - with **stage control**, where each stage can be controlled to be straight or crossed:
    * we can achieve $2^k$ stage settings altogether
    * however, this is distinct from actual permutations!
    * e.g. consider trying to map input 0 to output 0, at the same time as input 1 to output 3:
      · need to flip bits in 0 an even number of times, but the bits in 1 an odd number of times
      · impossible with stage control
  - with **switch control**, we now can toggle the switch settings for every switch:
    * with a $2 \times 2$ switch, we have $\frac{N}{2}$ switches
    * after $k$ stages, we have $2^{\frac{kN}{2}}$ switch settings
    * if we wanted to implement $N!$ permutations, necessary number of switch settings must be greater than $N!$
      · e.g. $k = (2 \log N) - 1$, not just only $k = \log N$
    * however, it is still useful to have additional stages beyond $\log N$ even when we cannot implement all permutations
      · unlike in stage control, where some permutations are impossible
  - algorithmic routing:
    * represent source and destination as binary addresses
    * to get from source to destination, we need to interchange in stage $i$ if the XOR of the corresponding address bits are 1
    * alternatively just use the destination tag itself
- similarly, we can have a multistage implementation of the cube network
  - essentially, stage $i$ can perform $cube_i$ or is a NOP
- another common pattern for constructing MINs is to proceed *recursively*, with $\log n$ stages:
  - in first stage, inputs are divided in half so that exactly half go to the top and bottom halves respectively
    * then, next stages continue to proceed recursively
  - e.g. in the **baseline network**, top of each switch goes to the top half network, while bottom of each switch goes to the bottom half network
    * alternatively, in the **butterfly network**, we have the same pattern, except inputs to the next stage alternate between top and bottom halves of the initial stage

  - e.g. the **k-ary n-fly** is a generalization of the butterfly network with radix $k$ and dimension $n$
    * has $k^n$ processing nodes with $n$ stages and $k^{n-1}$ $k \times k$ switch nodes per stage
- another desirable property for networks is **rearrangeability**:
  - can achieve all possible $N!$ permutations between inputs and outputs
    * typically requires $k = (2 \log N) - 1$ stages so that the number of switch settings is greater than $N!$
  - however, with existing connections may have to be rearranged to allow new connection to be established
  - e.g. the **Benes network** is exactly two butterfly networks, with the second reversed, attached end to end
    * proof of rearrangeability is an inductive proof that builds out from the smaller inner Benes networks
- another property for networks is **non-blocking**:
  - can achieve all possible permutations between inputs and outputs
  - however, existing connections need not be rearranged to allow new connections to be established
    * i.e. allows for incremental construction of a permutation, without rearranging existing connections
  - e.g. crossbar, Clos
- e.g. **Clos networks** are a family of symmetric networks of the construction $(m, n, r)$:
  - for non-blocking, need $m \geq 2n - 1$:
    * condition can be found in the worst case of connections to and from the middle switches
    * in the worst case, inputs connect to $n - 1$ middle switches and outputs connect to a disjoint set of $n - 1$ middle switches, so we need greater than $2n - 2$ switches
    * this version has a lot of built-in redundancy and fault tolerance
  - for rearrangeability alone, just need $m \geq n$
    * Benes network is special case where $m = n$
  - $m$ is number of middle switches
    * middle switches can themselves be constructed as full crossbars, or even recursively built using smaller Clos networks
  - $n$ is in / out ports for each in / out switch
  - $r$ is number of input switches
- yet another property for networks is **non-interference**:
  - for packet-switched networks, non-blocking and rearrangeability is not important
    * i.e. circuit is not reserved, so we have more flexibility
  - what is important?
    * adequate channel bandwidth for all traffic

        * allocation of resources e.g. buffers, bandwidth so no flow is denied service for more than a set amount of time
- another common topology is using a tree-based approach where nodes are the leaves:
    - to get to another node, just have to go up to the lowest common ancestor
    - *cons*:
        * non-uniform latency
        * congestion with higher parts of the tree like the root
        * laying out on a 2D grid on chip (since been solved with the H-tree layout)
    - however, we can visualize Butterfly and Benes networks as trees by folding them down the middle, creating a **fat tree** with bidirectional links:
        * bidirectional and nonblocking
        * logarithmic number of hops
        * route to a common ancestor up, and then down:
            · however, unlike a traditional binary tree, there are more ancestors as you go up the tree
            · i.e. use enough links so that the bandwidth is constant across all levels

**Fault Tolerance**

- e.g. Omega and Cube networks do not have redundancy
    - there is a unique path for inputs, so when a switch fails, some connections become impossible
- for the Cube network, we can use the **extra stage Cube network**:
    - use an additional stage over the normal Cube network:
        * new stage is called stage $m$ i.e. closest to the input
            · note that this stage has the (redundant) functionality of flipping the lowest bit
        * stage 0 is the stage closest to the output
    - input and output switches need an extra *bypass* functionality to skip past the switch in the case it fails
- handing faults in the extra stage Cube network:
    - in normal operation, stage $m$ is disabled i.e. bypassed, and stage $0$ is enabled
        * like the original Cube network
    - if we have a fault in stage $0$, disable stage $0$ and instead route bit $0$ with stage $m$
    - if we have a fault in an in-between stage, all stages are enabled:
        * instead of routing the normal path $s$ to $d$, route using the path with the source address $s_n \dots s_1 \bar{s_0}$ with the lowest bit flipped
        * this path $s'$ to $d$ is a *disjoint* path to the *same* desired output

    * use stage $m$ to flip the lowest bit, and route through this new disjoint path, avoiding the fault
  - proof the path is disjoint i.e. uses different, non-failing switches:
    * only stage 0 and $m$ can modify bit 0
    * in all other stages, each switch's labels are either both odd or both even
    * e.g. if original connection went through even switches, it is *guaranteed* to go through odd switches when we flip the lowest bit, which means the entire path is now disjoint

## Implementation

---

- implementation considerations:
  - performance metrics:
    * **latency** or time to traverse the network
      · unloaded, under load, max average
    * **throughput** or number of bits per second from inputs to outputs
      · max, average
  - reliability through redundant paths and routing
  - implementation cost in terms of switches and wires
- topology characteristics:
  - **diameter** is the worst case number of hops for all the shortest paths between all source-destination pairs
  - average routing distance
  - bisection width
  - switch degree
  - connectivity
  - algorithmic routing without deadlocks
  - partitionability
  - scalability
  - upgradability
  - layout
- $N^*$ set of nodes, connected by set of channels $C$:
  - terminal nodes are $N \subseteq N^*$
  - a **cut** of the network $C(N_1, N_2)$ is a set of channels:
    * $N_1$ and $N_2$ are disjoint, $N_1 \cup N_2 = N^*$
    * the cut includes all channels $N_1 \to N_2$ or $N_2 \to N_1$
  - a **bisection** is a cut that partitions nearly in half:
    * nearly equal nodes and terminal nodes
    * can help measure the communication capability in the network
  - the **channel bisection** $B_C$ is the minimum channel count over all bisec-

tions
    * similarly **bisection bandwidth** $B_B$
  - the **switch cost** is the total number of crosspoints in all the switches of the network
  - the **link cost** is the total number of the links in the network
- e.g. crossbar switch with $N$ nodes:
  - switch cost of $N^2$
    * number of crosspoints
  - link cost of $2N$ unidirectional links
    * links from end nodes to crosspoint line and back
  - bisection width of $N$
    * to disconnect top and bottom, have to cut $N$ links
- e.g. fully-connected network with $N$ nodes:
  - switch cost of $2N(N-1)$:
    * for one node, $N-1$ incoming and $N-1$ outgoing links to each of the other nodes
    * typically a $1 \times k$ switch has linear switch cost, while a $k \times k$ switch has quadratic switch cost
  - link cost of $N(N+1)$ unidirectional links:
    * for one node, $N-1$ outgoing links, plus 2 links to and from end node
    * for bidirectional links, divide by 2
  - bisection width of $\frac{N^2}{2}$ unidirectional links
    * wires to cut to isolate halves of the network
  - diameter of 1, average hop count of 1
  - *pros*:
    * scalable latency
    * scalable bandwidth
    * simple routing
  - *cons*:
    * high switch and link costs
    * poor upgradability
- e.g. Omega network with $N$ nodes and $k$ size switches:
  - switch cost of $Nk \log_k N$
    * each switch has $k^2$ crosspoints
  - link cost of $N(\log_k N + 1)$ unidirectional links
    * $N$ links per stage, plus an additional $N$ from MIN to end device
  - bisection width of $N$ unidirectional links
  - diameter of $1 + \log_2 N$, average hop count of $1 + \log_2 N$
  - *pros*:
    * scalable latency
    * scalable bandwidth
    * simple routing

  - *cons*:
    - switch and link costs increase faster than $O(N)$
    - difficult upgradability
    - layout complexity
- e.g. $k$-ary $n$-fly network with $N = k^n$ nodes:
  - switch cost of $k^2 \times k^{n-1} \times n = k^{n+1}n = Nklog_kN$
    - $k^{n-1}$ switches per stage, $n$ stages
  - link cost of $N(\log_k N + 1)$ unidirectional links
    - $N$ links per stage, plus an additional $N$ to end device
  - bisection width of $\frac{N}{2}$ unidirectional links
    - cutting the links crossing between top and bottom of the network in the first stage
  - diameter of $1 + \log_2 N$, average hop count of $1 + \log_2 N$
  - *pros*:
    - scalable latency
    - scalable bandwidth
    - simple routing
    - upgradable due to hierarchical structure
  - *cons*:
    - switch and link costs increase faster than $O(N)$
    - layout complexity
    - bisection independent of switch radix
- e.g. unidimensional torus (ring) AKA k-ary 1-cube:
  - switch cost of $\sim 3N$:
    - $2 \times 1$ switch for other node links
    - end device switching should be cheaper (two unidirectional links)
  - link cost of $3N$
    - $N$ among switches, $2N$ to end device and back
  - bisection width of 2 unidirectional links
  - diameter of $N - 1$, average hop count of $\frac{N}{2}$
  - *pros*:
    - low switch and link costs
    - simple wiring
    - simple routing
    - simple to upgrade
  - *cons*:
    - poor latency and bandwidth scalability
- e.g. unidimensional mesh (array) with bidirectional links:
  - switch cost of $\sim 6N$:
    - $2 \times 2$ switch for 2 bidirectional node links has a cost of 4
    - end device switching should be cheaper (one bidirectional link)
  - link cost of $2N - 1$ bidirectional links
    - $N - 1$ among switches, $N$ to end device and back

  - bisection width of 1 bidirectional link
  - diameter of $N-1$, average hop count of $\frac{N+1}{3}$
  - *pros*:
    * low switch and link costs
    * simple wiring
    * simple routing
    * simple to upgrade
  - *cons*:
    * poor latency and bandwidth scalability
- e.g. k-ary d-cube (torus) with $N = k^d$ nodes:
  - switch cost of $\sim N((d+1)^2 - 1)$:
    * $d^2$ relationship for a $d \times d$ switch
    * switch degree is $2(d+1)$
  - link cost of $N(d+2)$ unidirectional links
    * $d$ links per switch, 2 more to end device and back
  - bisection width of $2\frac{N}{k} = 2N^{1-1/d}$ unidirectional links
    * bisection width changes based on number of dimensions
  - diameter of $d(k-1)$, average hop count of $\sim d\frac{k-1}{2}$
  - *pros*:
    * switch and link costs scale $O(N)$ for fixed $d$
    * simple routing
    * upgradable
  - *cons*:
    * moderate latency scalability, $< O(N), > O(\log n)$
    * moderate bandwidth scalability, $< O(N), > O(1)$
- e.g. $d$-dimensional $k$-ary array with $N = k^d$ nodes:
  - switch cost of $\sim N((2d+1)^2 - (2d+1))$:
    * bidirectional links scale quadratically vs. torus
    * switch degree is $2d+1$
  - link cost of $\sim N(d+1)$ bidirectional links
    * $d$ links per switch, 1 more to end device
  - bisection width of $\frac{N}{k}$ unidirectional links
  - diameter of $d(k-1)$, average hop count of $\sim d\frac{k}{3}$
  - *pros*:
    * switch and link costs scale $O(N)$ for fixed $d$
    * simple routing
    * upgradable
  - *cons*:
    * moderate latency scalability, $< O(N), > O(\log n)$
    * moderate bandwidth scalability, $< O(N), > O(1)$

# Switching

---

- the network **switching strategy** defines how network resources e.g. links and buffers are managed:
  - a **message** is the logical transfer unit between source and destinations
  - a **packet** is the smallest unit of routing and sequencing independently handled by the network:
    - has a restricted maximum length
    - has sequence header (and tail) in case of out-of-order delivery
      - i.e. envelope overhead with header and trailer
    - finer granularity than message
  - a **flit** or flow control unit is the unit of bandwidth and storage allocation:
    - offsets the necessity of having smaller packets for more network utilization vs. the overhead of tacking on too many header bits with a finer flow control unit
    - not routed independently, but stream of bits may be paused at the flit boundary
    - finer granularity than packet
  - a **phit** or physical transfer unit is the unit of information transferred across a channel in a single cycle:
    - how many bits can go through the channel at the same time i.e. channel width
    - finer granularity than flit
- in **circuit switching**, all link bandwidth between source and destination is allocated to a particular location:
  - *pros*:
    - very low-latency once circuit is set up
    - no control information sent with messages
    - FIFO delivery
  - *cons*:
    - high circuit setup time
    - idle circuits consume link bandwidth
- in **message switching**, messages are independently routed through the network:
  - *pros*:
    - improved link utilization
    - adaptive routing possible
  - *cons*:
    - sequencing / routing information sent with each message
    - higher latency due to routing at each hop
- in **packet switching**, messages are broken into bounded-length, independently-routed packets:

  - *pros*:
    * even more link utilization
    * even more adaptive routing possible
  - *cons*:
    * sequencing / routing information sent with each packet
    * higher latency due to routing at each hop
- with **static virtual circuits**:
  - benefits of circuit and packet switching
  - no sequencing information and less routing overhead information
  - each physical link is divided into multiple virtual channels
    * essentially a level of indirection
  - a virtual circuit is a sequence of virtual channels from the source to destination:
    * multiple circuits can go through the same physical wires
    * message still split into packets
  - source node creates a **circuit establishment packet (CEP)**:
    * includes virtual channel, destination node, source node
    * virtual channels on the desired path are allocated to the new circuit
    * at each hop, mapping is recorded into an input mapping table
    * CEP virtual channel field is updated as it is forwarded through the network
  - to disestablish the circuit, the source node sends a **circuit destruction packet (CDP)**
- switch scheduling or arbitration:
  - independent from HOL blocking
  - we have buffers at each input, need to decide which buffers to transfer to which outputs
    * at most one request per output can be granted
  - performance, quality of service, and starvation prevention goals
  - arbiter ports:
    * $n^2$ requests as input
    * $n$ output port blocked statuses as input
    * $n^2$ granted requests as "output" by controlling the crossbar
  - necessary part of the switch design
- overall switch operation upon packet header arrival:
  1. route computation, where the output port is selected
  2. virtual channel allocation, where the packet gains exclusive access to a downstream virtual channel
  3. switch allocation, where the flit competes for crossbar access upon available space in output virtual channel
  4. switch traversal, where the flit is actually transferred from input buffer to output port and on to downstream router
- the **rotary router** is an alternative switch microarchitecture:

  - uses rotary rings, avoiding central crossbar and central arbitration
  - avoid deadlocks with bubble flow control

## Flow Control

---

- dealing with resource conflicts:
  - blocking the network
  - misrouting somewhere else, can lead to livelock
  - buffering, generates **backpressure** that blocks previous nodes
    * in **head-of-line (HOL)** blocking, a packet at the front of the FIFO buffer prevents other buffered packets from proceeding, even if their outputs are free
  - dropping of packets, causes a lower offered rate due to retransmissions
- dealing with HOL blocking:
  - in HOL blocking, expected number of busy outputs has the recursive relationship $E(n, k+1) = E(n, k) + \frac{n-E(n,k)}{n}$
    * free outputs have potential to be filled by packets deeper in the buffer, but can't due to FIFO nature
  - one approach is to use buffers at the output ports:
    * these buffers must support multiple simultaneous writes
    * in addition, flow control is complicated since free buffer space is associated with particular outputs
      · more communication than just being aware of the neighbor's free buffer space, e.g. neighbor needs to know which output
  - alternatively, use input buffers that *behave* like output buffers:
    * each buffer is associated with a particular output
    * still complicates flow control in terms of free buffer space status over multiple partitions
  - in statically-allocated multi-queue buffers:
    * the multi-queue buffer is still read one entry at a time, allowing for normal crossbar circuitry
    * less extra connectivity required than the previous solutions
    * still requires prerouting and extra status complications
  - in dynamically-allocated multi-queue buffers:
    * once again, less extra connectivity
    * but no need to do prerouting, all the multi-queues add up to the normal size of $n$
  - with **virtual channel flow control**, flits are interleaved on a physical channel:
    * deals with HOL blocking that occurs with wormhole forwarding over *multiple* switches

      - thus we are talking about flits in each switch instead of packets
    - acts as a kind of lane
    - channel or lane number is transmitted with each *flit*
    - increases performance, but diminishing returns past a certain number of channels
    - essential for wormhole routing
- with a blocking strategy, we need a link-level flow control:
  - with *short* links, we can use a handshake protocol on each flit
  - however, with *long* links that are pipelined, there is significant latency for acknowledgements to reach the sender:
    - instead, use **credit-based flow control**, where receiver sends "credits" to sender when slots are freed
    - alternatively, send "stop" and "go" messages when passing a certain low and high mark
      - high mark affects lost flits, while low mark affects idling time
    - have to address lost flow control messages

## Forwarding and Latencies

---

- network forwarding strategies:
  - typically, unlike larger networks, we never drop packets in multiprocessor networks:
    - cannot send packets if the destination node has no buffer space
    - need switch communication
  - in **store-and-forward**, wait for the entire packet to be stored before forwarding
  - in **cut-through**, instead of waiting for the whole packet, we can immediately start forwarding when we get some data:
    - reduces latency
    - still requires buffer room for a whole packet
  - in **wormhole routing**, we only require there to be room for one flit in the buffer to be forwarded
    - the flits of packets may be stored across several switches
- latency in a switch network is the sum of the following:
  - **overhead** i.e. getting the message in and out of the network at the end points
  - **channel occupancy** i.e. time to transfer a packet through channels
  - **routing delay** i.e. time to move first bit of the message source to destination
  - **contention delay** for resources against other packets
  - hop count $h$, hop delay $\Delta$, channel width $w$, message length $n$, packet

length $n_p$

- the unloaded latency for circuit switching is as follows:

$$T_{cs} = \frac{n}{w} + h\Delta$$

  - $\frac{n}{w}$ is the channel occupancy, $h\Delta$ is the routing delay
- the unloaded latency for store-and-forward routing with whole messages is as follows:

$$T_{sf} = h(\frac{n}{w} + \Delta)$$

  - have to wait for messages to arrive before forwarding
- the unloaded latency for store-and-forwarding a single packet, and an entire message are as follows:

$$T'_{sf} = h(\frac{n_p}{w} + \Delta)$$

$$T_{sf} = \frac{n - n_p}{w} + h(\frac{n_p}{w} + \Delta) = \frac{n}{w} + h\Delta + \frac{(h-1)n_p}{w}$$

  - note due to pipelining, the channel occupancy makes up the latency after the initial packet
- the unloaded latency for cut-through routing is as follows:

$$T_{ct} = \frac{n}{w} + h\Delta$$

  - ideally, approaches circuit switching latency
- **raw link bandwidth** is $b = wf$, but **effective link bandwidth** when transmission is blocked for routing decision is as follows:

$$b(\frac{n}{n + n_E + w\Delta})$$

  - hop delay is lost opportunity to transmit $w\Delta$ bits
  - $n$ packet length, $n_E$ envelope length, $\Delta$ hop delay
- the **global bandwidth** is measured among terminal nodes:
  - **bisection bandwidth** is channel bisection times channel bandwidth
    * not always a good measure, depending on the locality or uniformity of the routing
  - **peak bandwidth** is every channel delivering to terminal nodes at peak channel bandwidth
    * e.g. direct network
  - **application-specific bandwidth** depends on average number of hops
- average link utilization:
  - injection rate of $\frac{1}{M}$ packets per cycle
  - aggregate injection rate is $N\frac{n}{M}$ bits per cycle
  - each packet is "re-injected" $h - 1$ times

- total traffic rate is $\frac{Nhn}{M}$ bits per cycle
- peak network bandwidth is $Cw$ bits per cycle
- link utilization is then traffic rate divided by peak network bandwidth
- since the utilization must be less than or equal to 1, we have the inequality $M \geq \frac{Nhn}{Cw}$

- impact of dimension in a torus:
  - increase dimension to minimize hops, otherwise, decrease dimension to minimize cost
  - purely considering latency, low dimension networks scale poorly
  - however, we need to consider cost as well:
    - with fixed pin consideration, we fix the number of pins per node at $2wd$:
      - one possible measure to fix the cost per node
      - there is now an optimal dimension that is relatively small to balance latency and cost
    - with fixed wire bisection, we fix the number of wires in the bisection:
      - increasing dimension decreases the channel width and increases the channel time
      - again, there is an optimal dimension earlier on
    - these optimization problems depend on assumptions about the hardware
  - under load i.e. more channel utilization, higher dimensions typically achieve better throughput and latency

## Routing

---

- in a network, we have nodes $N$ and a set of channels $C \subset N \times N$:
  - **routing** determines the path from source to destination
  - the **routing relation** $R \subset C \times N \times C$ identifies *all* permissible paths
    - sometimes, source node may affect permissible paths
  - the **selection function** $\rho : P(C) \to c$ specifies the next channel to use
    - may be affected by the state of the network
  - in **source routing**, the entire path is determined at the source node
    - simpler switches, but longer headers and slower adaptation
  - in **incremental routing**, determine the next channel at each intermediate node
    - worse switch complexity, but smaller headers and better adaptation
  - in **minimal routing**, the routing mechanism always chooses the shortest path in number of hops
    - but sometimes congestion or a network fault should come into play,

cannot take shortest path

- selection function considerations:
  - in **deterministic** selection, there are no choices, routing relation yields one choice:
    * simple and fast, poor handling of non-uniform traffic and faults
    * **incremental algorithmic** requires a regular topology but is simple and fast
    * **incremental table-driven** works with arbitrary topologies, but requires storage and slower access for tables
    * **source routing** is usually deterministic once leaving source
  - in **oblivious** selection, the choice is *independent* of network state
    * key difficult is ensuring progress especially when non-minimal i.e. preventing livelock
  - in **adaptive** selection, we take network state into account
    * more complex, but best use of available bandwidth

**Livelock and Deadlock**

- in **livelock**, we have active state changes, but the task does not monotonically advance towards completion, and thus never completes:
  - different from deadlock and starvation
  - in non-minimal routing, a packet may be misrouted forever
  - in dropping flow control, retransmitted packets may also be dropped
  - through **deterministic avoidance**, we add state to every packet to ensure eventual progress towards destination
    * stop misroutes after a certain packet age
  - through **probabilistic avoidance**, guarantee a non-zero probability of progress at every step
- in **deadlocks**, we have blocked agents that are unable to make progress or release resources in a cycle:
  - not an issue with dropping flow control, but serious problem with blocking flow control
  - a cycle in the resource dependence graph is a necessary but not sufficient condition for deadlock
  - deadlock avoidance by eliminating cycles:
    * impose a partial order on resources
    * agents acquire resources in ascending order (simple labelling approach to prove against deadlock)
    * consequences of increased resource requirement or restricted agent actions or both
  - many different specific deadlock avoidance implementations, as follows
- e.g. in a **structured buffer pool**, partition buffer space into multiple buffer classes:

    - packets use buffers from strictly increasing buffer class
    - buffer class can be distance of packet from sender
- e.g. switch with multiple virtual channels:
    - buffer space divided up into different virtual channels
        * packet header decided which virtual channel it should go into (like virtual circuits)
    - example policy is, once a packet *crosses* a certain node, start using a certain virtual buffer class
        * no more cycle in the graph
    - buffering virtual channels identify buffer resource to be used by the packet:
        * number of channels relates to topology and routing algorithm
        * motivated by deadlock avoidance
    - while routing virtual channels (with static virtual circuits) identifies next link to be used by the packet:
        * number of channels related to number of connections through a port
        * motivated by smaller packet header and simpler / faster routing
- we can also restrict physical packet routes to avoid cycles:
    - e.g. in a mesh, use dimension-order routing
    - e.g. other directional routing techniques dealing with turns:
        * avoids deadlocks by essentially prohibiting certain "turns" AKA **turn restriction routing**
        * prohibiting one clockwise and one counter-clockwise turn alone is not necessarily enough to remove cycles
        * e.g. west-first (no turns into negative x), north-last (no turns out of positive y), negative-first all remove cycles
    - e.g. with up-down routing, create a spanning tree, route up to the root, and down to the destination:
        * label the buffers with the tree layer number going up and *past* the root, and then in the other direction
        * root becomes a congestion point
- e.g. Duato's deadlock-free routing theory is that we can route adaptively without constraints, but always have an "escape route" to a deadlock-free network
    - not necessarily a separate network, but separate resources i.e. virtual network
- e.g. in **bubble flow control**, we have a deadlock-free ring *without* virtual channels:
    - a ring cannot deadlock if there is one free packet slot in *one* of the buffers
    - only inject into a ring if there are *two* free packet slots at the injection point

## Case Study

---

- the Summit and Sierra supercomputers use an InfiniBand interconnection network in a three-level fat tree topology:
  - features thousands of nodes (both CPUs and GPUs), advanced adaptive routing, high level of redundancy and reliability, ability to isolate traffic among partitions and subsystems
  - "fat nodes" combine POWER9 CPUs and NVIDIA GPUs, that are then connected together into a global network:
    - the ConnectX-5 HCA network adapter was offloaded from CPUs via PCIe lanes
    - ConnectX-5 has two bidirectional ports with 100 gigabit per second speeds
    - packs together two CPUs per leaf node for easier packaging
  - first layer above fat nodes are 18 by 18 "top-of-rack" switches
  - second and third layers are made out of "core switches" that are constructed out of 18 18 by 18 switches, creating 648 ports altogether
    - creates a full bisection fat tree
  - multiple networks for different dedicated functions e.g. collectives, Ethernet, control, etc.
  - redundant connections in the first level of the fat tree
  - top level can be dynamically sized based on cost and performance tradeoffs
- topology details:
  - fat tree excels at scalability, global and local bandwidth, isolation, traffic balance, and fault tolerance
  - fewer virtual channels required to avoid deadlock compared to Dragonfly alternative
    - however, Dragonfly has a lower switch and link cost
- ConnectX-5 HCA features:
  - decoupled from the end node devices makes network easier to upgrade
  - hardware-based tag matching
  - CORE-Direct coordinates between different queue pair messages
  - hardware-based out-of-order data handling
- switch features:
  - adapting routing takes network status to be considered when choosing the packet route:
    - deciding whether to reroute, and choosing an output port
    - double performance comapred to static routing, while using more than 95% of peak link bandwidth
    - with fat tree, latency scales well with higher node counts as well
  - support for collective operation acceleration e.g. parallel reductions, ag-

gregations
- PAMI messaging stack:
  - on demand paging to pin certain pages, preventing the OS does not move certain pages during remote communication transfer
  - dynamically connected transport scaling
  - uses hardware tag matching

# Message Passing

---

- **message passing** across networks is done through **network transactions**:
  - one-way transfers of information from source output buffer to destination input buffer
    - e.g. deposit data, execute a handler
  - *difficulties*:
    - source and destination operations are decoupled
    - no global information or scheduling
    - completion detection
    - delivery guarantees
    - transaction ordering
    - deadlocks
  1. format conversion e.g. placing into envelope
  2. output buffering
  3. message arbitration to get access to network
  4. route through network
  5. input buffering at destination
  6. protection check at destination
  7. action at destination
- popular example **Message Passing Interface (MPI)**:
  - point-to-point APIs include:
    - `MPI_Send(buffer, count, type, dest, tag, comm)`
    - `MPI_Recv(buffer, count, type, source, tag, comm, status)`
  - collective APIs include:
    - `MPI_Barrier(comm)`
    - `MPI_Bcast(buffer, count, datatype, root, comm)`
  - `tag` identifies certain type of message from the same source
  - `comm` is the set of communication nodes
- by utilizing input buffers at destinations, we can allow for many simultaneous operations without central coordination:
  - efficient resource management of buffers is difficult
  - balancing under-utilization i.e. unnecessary sender restrictions
    - missed opportunities to use storage resources
  - vs. over-commitment i.e. backpressure
    - leads to blocking and discarding
- message operations vs. message functions:
  - **functions** provide *interfaces* to actual **operations**
    - completion of operations are distinct from return of functions
  - a **synchronous operation** completes once a matching receive has been executed and data has arrived at the destination:

    * uses three-way handshaking to confirm the buffer space
    * explicit resource availability check, but incurs delay
  - an **asynchronous operation** completes once the source buffer can be reused:
    * problem of where to place arriving message
    * requires unbounded temporary buffer space in case matching receive may have not been posted, so receive is *not* guaranteed
      · poor performance if we must copy from system buffer
    * note that the receive always completes when data is at destination and is always synchronous
    * AKA **asynchronous optimistic** message passing
  - with **asynchronous conservative** message passing, we can use the optimistic or credit scheme for shorter messages
    * but use a synchronous handshake approach for long messages
  - a **blocking function** returns only after the operation completes
  - a **non-blocking function** returns immediately
- possibility of **fetch deadlock**:
  - occurs with an unbounded number of request and insufficient buffer space
  - requests back up to sender's outgoing port, so sender cannot send replies
    * processing head request requires a reply, leading to deadlock
  - *solutions*:
    * independent request and response networks or virtual networks
    * large input buffers
    * ability to NACK requests
    * active messaging that causes an action at the receiver
      · reduces latency, and buffer pressure
    * partitioned global address spaces

## Design Space

---

- with **communication architecture (CA)**, there have been many different approaches to design network transactions in parallel systems:
  - want to base off of regular I/O e.g. programmed, interrupt-driven, DMA, I/O processor approaches
  - key issue is the fraction of work offloaded from the CPU
    * as well as protection, bandwidth, latency, granularity of I/Os
  - key questions:
    * how to initiate send and receive?
    * how to protect the network and data?

    * who moves the bits into and from the network?
    * how to deal with unwanted messages?
    * where to store the message?
    * how to inform the application?

- with **blind physical DMA**, just use DMA controllers to move bits between CPUs:
  - OS protects with system calls, and notifies with interrupts
  - this requires multiple memory copies for end-to-end operation, expensive
- with the NIC approach, use network adaptors that translate through a stack of layers, from application down to socket and link layers
- **push** messaging pattern:
  - "eagerly" push data from the source to the destination, and buffer it for the receiving processor

  1. sender uses fetch and increment against the receiver's fetch and increment register
     - gets a unique index into a preallocated message buffer
  2. execute remote write operations to move data into buffer
     - data transfer will then occur over network
  3. receiver reads the message from the buffer in its local memory when it wishes to process the message

  - *pros*:
    * minimizes latency at low network loads
  - *cons*:
    * performance degrades with contention
    * if receiver's memory starts to fill up, we get back pressure onto the network
- **pull** messaging pattern:
  - "lazily" move data when the receiver is ready to process it
  - requires a shared address space

  1. sender copies data into local buffer
  2. use atomic swap to link the message into receiver's receive queue:
     - queue is a distributed linked list whose head and tail are stored at the receiver
     - remote stores to old tail pointer
  3. when receiver wants to process the message, pulls message into local memory using remote read
  4. receiver deallocates message buffer

  - *pros*:
    - nodes are never swamped with data, eliminating output contention and buffer overflow
    - enqueuing messages is comparatively low overhead
    - data is matched to receiver's polling rate
  - *cons*:
    - pulling data across the network requires more work, more latency

- ideally, we want to allow for **user-level access**:
  - network ports are mapped to product address space, including I/O queues and control registers
  - then, sender can process these queues and registers *without* OS involvement
    - key issue here is pinning memory so that we do not need to search for swapped out pages via OS
  - sender CA wraps in envelope and injects into network
  - receiver CA places into input queue and updates status registers
  - receiver process notified with interrupts or polling

- **virtual interface architecture (VIA)**:
  - allows for user-level access
  - goal is to reduce the *software* overhead behind a high-performance processor and network:
    - low latency and high bandwidth communication between two nodes on the network
    - motivated by the many layers of software required to get to and from network
  - with different classes of communication traffic, different optimizations matter more
    - decreasing software overhead benefits smaller messages, while increasing network bandwidth benefits larger messages
  - VIA defines a simple set of operations and follows some basic guidelines:
    - eliminate intermediate data copies by copying directly rather than indirectly through a system buffer
    - avoid interrupts and traps
    - avoid drivers running in kernel mode
    - minimize instruction count
  - in VIA, each process has the illusion of owning the interface to the network:
    - called **instances**, each has one send and receive queue and is owned by a single process
    - each of the queues is formed by a linked list of variable-length descriptors

        - **descriptors** are posted onto the tail of the work queue, and writing to a **doorbell** register to notify the NIC
    - descriptors can be sends, receives, and remote reads and writes
    - the NIC gradually works through the work queue using DMA, updating the descriptor and transferring ownership when it is done
    - notifications on description completion can be done through polling or blocking calls
  - VIA parts:
    - NIC hardware with DMA, doorbells, completion queues, and user-managed TLB
    - OS agent for memory registration, VI creation and destruction, connection setup and teardown
    - VI consumers that are applications and libraries
  - InfiniBand networks and RDMA use parts of VIA

- with virtual network interfaces, the network interface needs to know the address translations of the virtual addresses:
  - application process uses virtual memory whereas the network interface accesses physical memory
    - ideally want to eliminate OS calls (except for pinning)
  - additionally, the NIC has no control over paging and swapping in the OS, so the memory buffers must be pinned before data transfer can take place
  - one approach is to use **user-managed TLBs (UTLBs)**:
    - through **demand-driven page-pinning**, pin the local buffer when it is used in communication for the first time
      - keep it pinned so subsequent transfers can be initiated at the user level
    - then, provide the user a handle into the memory region e.g. the TLB entry number
    - establish a *protected* translation table for pinned virtual pages:
      - invisible to the user process but entries and indices are managed by the user, NIC can easily read the physical address
    - create a fast *user-level* lookup data structure that tracks translation table indices and pinned virtual pages
    - need OS to initially pin pages and set up UTLB entries, but afterwards, communication can proceed without OS intervention

# SIMD and Vector Architectures

---

- motivations for SIMD:
  - efficiently exploit finer grained parallelism
  - design simplicity, since we only have one control unit
    - no coordination or synchronization mechanisms
  - performance and power efficiency through ALUs
  - simple to program and debug
  - many graphics, media, and scientific applications
- with **parallel SIMD**, we have an **array computer** where we have multiple ALUs each connected to multiple memory:
  - while with **pipelined SIMD**, we have a **vector processor** where CPU is connected to multiple memory, and we have a longer ALU pipeline due to the fact that we have no dependencies
  - both can be combined in a single implementation

## Pipelined SIMD

---

- why vector processors:
  - instruction fetch and decode need not keep up with execution rate of pipelined functional units
  - no need for complex hazard detection among many instructions
    - critical parameter is number of in-flight instructions rather than operations
  - simple memory access patterns allows for optimized utilization of interleaved memory
  - intelligent software controlled caches via vector register file
  - reduced impact of control dependencies
- the **vector register file** allows for dynamic register file configuration before use:
  - registers can be enabled and disabled
  - data type and width stored with each register
  - configuration and implementation impacts contention
  - e.g. `vsetdcfg 1*FP32, 3*FP64` sets 1 32-bit floating point and 3 64-bit floating points

Comparing code for DAXPY i.e. double precision $ax + y$:

```
; scalar
      fld       f0, a
      addi      x28, x5, $256
```

```
loop: fld       f1, 0(x5)
      fmul.d    f1, f1, f0
      fld       f2, 0(x6)
      fadd.d    f2, f2, f1
      fsd       f2, 0(x6)
      addi      x5, x5, $8
      addi      x6, x6, $8
      bne       x28, x5, loop

; vectorized
vsetdcfg 4*FP64
fld      f0, a
vld      v0, x5
vmul     v1, v0, f0
vld      v2, x6
vadd     v3, v1, v2
vst      v3, x6
vsdisable
```

- a **convoy** is a set of vector instructions that can begin execution together, without structural or data hazards:
  - a **chime** is the execution time of one convoy
  - in the above example, the `vmul` and second `vld` have no dependencies and thus form a convoy
- pipelined SIMD performance:
  - $T_{serial} = dn$
  - $T_{pipe} = d - 1 + n$
    * $d - 1$ forms the startup penalty due to pipelining
  - thus, we have a speedup of $S = \frac{T_{serial}}{T_{pipe}} = \frac{dn}{d-1+n}$
    * as $n \to \infty$, $S = d$
  - alternatively, $T_{pipe} = T_{start} + nT_{element}$ :
    * with a deeper timeline, $T_{element}$ drops while $T_{start}$ increases
    * define $r_\infty = \frac{1}{T_{element}}$ and $n_{1/2} = T_{start} r_\infty$
    * then, we have $T_{pipe} = \frac{n_{1/2}}{r_\infty} + \frac{n}{r_\infty}$
    * $n_{1/2}$ determines the cutoff time at which vectorization becomes worthwhile
- one common technique is to use multiple **lanes** to combine pipelining with parallel SIMD:
  - we can perform array calculations in any order, since there are no inter-array dependencies
  - "stripe" the array elements across multiple lanes
    * achieves speedup proportional to number of lanes

- with **chaining**, we connect together pipelines on an element by element basis:
  - e.g. feeding multiply result directly into add instead of waiting for the entire array calculation to complete
  - pipelining the pipelines
- since array lengths will vary, we can use **strip mining** to deal with arbitrary array lengths:
  - "strip" and calculate parts of the arrays at a time, based on **maximum vector length (MVL)**
  - `setvl` sets the vector length register
  - as vector size increases, element execution time drops, but the overhead based on having to strip mine appears at each MVL

**Vector mask registers** are used to conditionally execute vector calculations:

```
; for ...
;    if (X[i] ≠ 0)
;       X[i] = X[i] - Y[i]

vsetdcfg    2*FP64
vsetpcfgi   1         ; enable 1 predicate register
vld         v0,x5
vld         v1,x6
fmv.d.x     f0,x0
vpne        p0,v0,f0 ; set p0(i) if v0(i)≠0
vsub        v0,v0,v1 ; subtract under vector mask
vst         v0,x5
vdisable
vpdisable             ; disable predicate registers
```

- the vector **stride** is the distance between consecutive vector elements:
  - common necessity in 2D arrays, need to support loading given a stride
  - e.g. `vlds v1,(x3,x9)` loads `v1` from address at `x3` stride in `x9`
    - similarly `vsts` stores at a stride
- if array elements are scattered at a non-uniform stride, we need another mechanism to load:
  - with **gather / scatter**, index array elements through an index vector
  - e.g. `vldx v1,(x5,v2)` loads `v1` with vector whose elements are at `x5` indexed by `v2`
    - similarly `vstx`
- the memory bandwidth bottleneck is a critical issue for vector processors:
  - need many reads and writes per cycle to keep up with the functional unit pipelines
  - use **interleaved memory** in order to divide up memory into multiple **modules**

        - MSBs specify word in module, while LSBs specify module
    - e.g. if we have 8 modules with an access time of 2 cycles, we can achieve 4 reads or writes per cycle
    - however, with interleaved memory, we need to carefully configure the **data layout** with respect to module accesses for maximum performance:
        - need to coordinate which accesses go where so that all operations happen without conflicts (no delay, so elements have to arrive pair by pair into the ALU)
        - alternatively, add in variable delays so that we can simplify the data layout
            - each ALU is responsible for a certain array index calculation, and use delays to line up the calculation of two elements
    - for laying out matrices on modules:
        - with layout by rows or columns, one pattern of access is always poor
            - e.g. layout by rows is poor for column vector access
        - instead, layout by rows, but skip one module
            - column vector access is still striped across all modules
    - mathematically, accesses are directed to at most $\frac{M}{GCD(s,M)}$ different modules, where $M$ is number of modules and $s$ is stride:
        - if $s, M$ are relatively prime, we get $M$ distinct accesses
        - if $M$ is a power of 2, use an odd stride
        - however, the stride for the major diagonal access pattern in a matrix is 1 more than the column stride
            - have to use prime number of modules (difficulty decoding)

## Vectorization

---

- data dependence definitions:
    - **true dependence** or $S_1 \delta S_2$ is a read after write dependency
        - $S_1$ must precede $S_2$
    - **antidependence** or $S_1 \bar{\delta} S_2$ is a write after read dependency
        - $S_2$ must come after $S_1$
    - **output dependence** or $S_1 \delta^\circ S_2$ is a write after write dependency, to the same place
        - $S_2$ must come after $S_1$
    - dependencies can be within loops, or loop-carried between different iterations
        - for vector processors, loop-carried dependencies matter because we execute all array element calculations in parallel!
    - if we have a cycle-free dependence graph, we can achieve full vectoriza-

tion
    * with cyclic dependencies, we can only achieve partial loop vectorization

Example full vectorization:

```
do i = 1,N
    A(i) = B(i)        ; S1
    C(i) = A(i) + B(i) ; S2
    E(i) = C(i+1)      ; S3
end do

; Dependencies: S1→S2, S3→S2

A(1:N) = B(1:N)          ; S1
E(1:N) = C(2:N+1)        ; S3
C(1:N) = A(1:N) + B(1:N) ; S2
```

- **reduction operations** are a common special pattern e.g. sum, max
- more vectorization techniques:
    - recognize **induction variables** i.e. loop variables whose values form an arithmetic progression like decrementing backwards
    - support for **wraparound variables**
    - allow for **symbolic data dependence testing** i.e. handling subscript expressions containing terms that are unknown at compile time
        * may only be able to partially vectorize in the general case
    - variable renaming when possible, similar to register renaming
    - node splitting to eliminate some dependence cycles
    - scalar expansion
    - loop interchanging:
        * not always possible, may change the execution result
        * test by drawing the dependency graph and traversing column-first vs. row-first
    - loop collapsing multiple dimensions
        * memory is one dimensional

Vectorizing with node splitting:

```
do I = 1,N
    A(I) = B(I) + C(I)
    D(I) = A(I) + A(I+1)
end do

do I = 1,N
```

```
    ATEMP(I) = A(I+1)
    A(I) = B(I) + C(I)
    D(I) = A(I) + ATEMP(I)
end do

ATEMP(1:N) = A(2:N+1)
A(1:N) = B(1:N) + C(1:N)
D(1:N) = A(1:N) + ATEMP(1:N)
```

Vectorizing with loop interchanging:

```
do J = 1,N
    do I = 2,N
        A(I,J) = A(I-1,J) + B(I)
    end do
end do

do I = 2,N
    A(I,1:N) = A(I-1,1:N) + B(I)
end do
```

## Parallel SIMD

---

- with parallel SIMD, we have a single control unit that controls many functional units:
  - can be organized as either a processor-to-processor or processor-to-memory architecture
  - crucially, we have to deal with routing information between the processor elements:
    * layout of the SIMD network becomes important, often a grid layout
    * need to broadcast scalars, etc.

Matrix multiplication with parallel SIMD:

```
for k = 1 to N
    for i = 1 to N
        C[i,k] = 0
        for j = 1 to N
            C[i,k] = C[i,k]+A[i,j]*B[j,k]
        endfor
    endfor
endfor
```

```
for i = 1 to N
    C[i,k] = 0 ; (1 < k < n in parallel)
    for j = 1 to N
        C[i,k] = C[i,k]+A[i,j]*B[j,k] ; (1 < k < n in parallel)
    endfor
endfor
```

- in the above matrix multiplication implementation:
  - we have row elements distributed across processor elements
  - need to perform a scalar broadcast of elements of `A`

Sum of an array of numbers with parallel SIMD:

```
for j = 1 to log2n
    for all k in parallel
        if (k+1)mod(2^j)=0
            x[k] = x[k-2^(j-1)] + x[k]
        fi
    endfor
endfor
```

- parallel SIMD usually appears as ISA extensions:
  - original motivation for media applications:
    * many operate on narrow data types
    * inherent data parallelism
    * runs on processors with wide ALUs
    * e.g. packed compare operations
  - easy to partition registers and ALUs
  - e.g. Intel MMX, AVX

# GPUs

---

- in a conventional CPU:
  - a **thread** is a program in execution, with its state being the contents of all registers and memory
  - **multithreading** is multiple threads executing on a CPU at the “same” time:
    * this is done by a software in a normal CPU
    * must stop running thread, save state, select next thread, load register state, run thread
- there is a motivation for hardware multithreading instead:
  - hide stalls incurred by memory latency or data dependencies

    - software thread switching is too expensive, so switch threads without software intervention
    - implemented via multiple register sets and a hardware thread scheduler
    - can achieve **simultaneous multithreading (SMT)** i.e. **hyperthreading**
        * even more interspersed than coarse and fine-grained multithreading
- **graphical processing units (GPUs)** are not only tied to graphics applications:
    - general purpose GPUs (GPGPUs) are generally applicable
    - motivations with SIMD:
        * many applications have lots of parallelism
        * useful work is done by ALUs, minimizing complexity of control units
    - problems with SIMD, specifically large SIMD machines:
        * communication is expensive
        * data-dependent execution where many PEs are off much of the time
    - with GPUs, take the SIMD motivations:
        * and remember that shared memory SIMD is a good starting point
        * use multiple "small" SIMD machines
        * critically, we may have poor memory performance due to no room for large caches (stuffed with ALUs):
            · try and exploit parallelism using hardware multithreading
            · use some memory resources to support many hardware threads
        * overall, should allow for the best power overhead amortization compared to conventional CPUs
- GPGPU programming:
    - users are familiar with SPMD (single program, multiple data) programming, so we can use SPMD to approximate SIMD
    - with SPMD, no need for implicit synchronization at instruction granularity
        * different portions of the SIMD approximation instruction need not be executed simultaneously (as long as execution paths of threads do not diverge)
    - burden on the programmer / compiler:
        * which threads are part of a thread block
        * which threads are in the same WARP
            · impacts branch divergence, memory coalescing, etc.
        * explicitly moving data between host and GPU
        * data placement in memory hierarchy
- NVIDIA terminology:
    - streaming processor core (SP)
    - special function unit (SFU)
    - streaming multiprocessor (SM)
        * made up of instruction and data caches, SPs, and SFUs

  - Compute Unified Device Architecture (CUDA) is NVIDIA's API basis
  - CUDA thread is lowest granularity thread
    * essentially a single instruction targeting single data
  - a SIMD thread or **warp** is composed of multiple CUDA treads
    * warp is executed simultaneously, passed to each SP
  - a thread block contains multiple warps
    * *logically*, the thread block executes all at once, but in reality, they are staggered in execution at the warp level
  - a grid contains multiple thread blocks

DAXPY in CUDA:

```
__host__
int nblocks = (n+255)/256;
daxpy<<<nblocks, 256>>>(n, 2.0, x, y); // 256 threads per thread block
__device__
void daxpy(int n, double a, double* x, double* y) {
    int i = blockIdx.x*bloxkDim.x + threadIdx.x;
    if (i < n)
        y[i] = a*x[i] + y[i];
}
```

- however, it is possible for execution paths of SPMD threads to diverge:
  - need to use predicated execution
    * if condition is true, instruction is executed, otherwise instruction impact is `NOP`
  - then, we only have a single execution path
  - thus we need the following logic:
    * if divergence is within a warp, use predication to serialize code
    * otherwise, execution resources are used only for the correct execution path, and there is no problem since there is no simultaneous execution
  - usually, GPUs have support for efficient branch divergence and reconvergence:
    * use a branch synchronization stack per SIMD thread
    * an active mask vector per SIMD thread can be pushed and restored from this stack