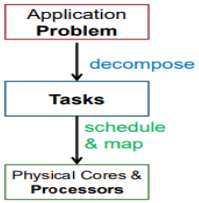
CS3210 Cheatsheet AY24/25 —— @JasonYapzx

1. Parallel Computing

- **Simultaneous use** of **multiple processing units** to solve problem fast / larger problem
- **Processing units:**
  1. Single processor with multi-core
  2. Single computer with multi-processors
  3. Number of computers connected by a network
  4. Combinations of above
- **von Neumann Model**
  - **Processor** performs instructions
  - **Memory** stores instructions and data in cells/addresses
  - **Control Scheme** fetch instructions from memory, shuttles data btw memory/processor
  - **Memory Wall** disparity between processor ($< 1 \times 10^{-9}$(nano) sec) vs memory speed ($100 - 1,000$ nano seconds)

How

1. Problem divided into m discrete parts (tasks) solved concurrently
2. Each part further broken down to a series of instructions (i)
3. Instructions from each part execute in parallel on different processing units (p)



- **Decomposition:** Potential parallelism of app → *how it should be split* into tasks — Size of tasks is called *granularity*
- **Scheduling**: Assignment of tasks to processes or threads
  - Manaully defined? Static? Dynamic? Execution Order?
- **Mapping**: Assignment of processes/threads to physical cores/processors for execution
- Tasks may depend on each other resulting in *data* or *control dependencies* → **impose execution order** of parallel tasks

Dependences and Coordination

- Dependences among tasks impose constraints on scheduling
- Correctness: processes/threads need *synchronization/coordination*
  - depends on information exchanges between processes and threads → depends on the hardware memory organization
- Memory organizations: shared-memory (threads) and distributed-memory (processes)

Concurrency vs Parallelism

Concurrency

- > 1 tasks start/run/complete in overlapping time period
- Might not be running (exec on CPU) **at the same instant**
- > 1 excution flows make progress by **interleaving** executions/exec instructions

Parallelism

- > 1 tasks can run (execute) simultaneously **at the same time**
- Tasks NOT ONLY makes progress **AND** execute simultaneously

Parallel Performance

Execution Time vs **Throughput**

- **Throughput:** (amt of computation observed in a specific amt time)
- Parallel Exec Time = Computation Time + Parallization Overheads
  - Overheads:
    - forking to distribute tasks/joining to combine results
    - information exchange or synchronization OR idle time
  - If computation is tiny, overhead >> performance optimization
  - Helpful only for **LONG** and **CONSISTENT** computations

2. Processes and Threads

- A **Process** is an instance of a program in execution, identified by a **Process ID (PID)**.
  - Includes: executable program, global data, stack, heap, and OS resources like open files and network connections.
  - Own address space, providing exclusive access to its data.
  - Communication between processes requires explicit mechanisms.
- A **Thread** is an extension of the process model:
  - Process consist of multiple independent control flows (threads)
  - Threads share the address space of the process, allowing shared-memory architecture.
  - Each thread has its own **Thread ID**, Program Counter (PC), Stack Pointer (SP), and register values.

Process Interaction with OS

Exceptions

- Executing a machine level instruction can cause exception
- For example: Overflow, Underflow, Division by Zero, Illegal memory address, Mis-aligned memory access
- Synchronous: due to program exec, **exception handler**

Interrupts

- External events can interrupt the execution of a program
- Usually hardware related: Timer, Mouse Movement, Keyboard Pressed etc
- Asynchronous: occurs independetly of program exec, **interrupt handler**

User Thread

Thread is implemented as a **user library**. Kernel is **NOT AWARE** of the threads in a process.
ADVANTAGES: Fast Context Switching
DISADVANTAGES:

- OS unaware of threads, scheduling performed at process level
- One thread blocked → Process blocked → all threads blocked, cannot exploit multiple CPU

Kernel Thread

Thread implemented in OS, thread operations handled as system calls.
ADVANTAGES

- Kernel can schedule on thread levels: More than 1 thread in the same process can run simultaneously on multiple CPUs

DISADVANTAGES

- Thread operations are now a syscall: slower, more resource intensive. Less flexible: Used by all multi-threaded processes

Mapping Strategies

- **Many-to-One:** All user-level threads → one process, thread library responible for scheduling user-level threads (executed by same kernel thread)
- **One-to-One:** User-level thread → exactly 1 kernel thread, no library scheduler. OS responsible for scheduling and mapping kernel threads (p-threads)
- **Many-to-Many:** Library scheduler assigns user-level threads → given set of kernel threads. Kernel scheduler maps kernel threads to available execution resource, at different points, user thread → different kernel thread

2.1 Synchronisation

Race Condition

Multiple execution paths finish in different order than expected, **critical** race conditions cause invalid execution → happens when processes/threads depend on shared state

Critical Section

- Protect parts of program where shared resource is accessed to avoid concurrent access.
- Cannot be entered by > 1 process/thread at a time
- Others are suspending until first leaves the section

Data Race

- 2 concurrent processes/threads access *shared resource(mem location)* without any **protection**
- AND at least 1 thread modifies the shared resource

Mechanisms

- **Locks:** `acquire()`, `release()`
- **Semaphores:** Integers that support `Semaphore::Wait()`, `Semaphore::Signal()` to decrement and increment respectively → value will always be ≥ 0
  - **Mutex** semaphore: binary, single access to resource `N = 1`
  - **Counting** semaphore: multiple threads can pass based on count

Deadlock

Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set. Conditions for deadlock (must hold simultaneously):

- Mutex: ≥ 1 resource held in non-sharable mode
- Hold and wait: one process holding 1 resource, waiting for another
- No pre-emption: Resources cannot be pre-empted (critical sections aborted externally)
- Circular wait: Processes `P1`, `P2` `P3`, `P1` waiting for `P2`, `P2` → `P3` ...

Starvation

Starvation is a situation where a process is prevented from making *progress* because some other process has the resource it requires

Livelock

The states of the processes involved in the livelock constantly change with regard to one another, none progressing

- Special case of resource starvation, specific thread not progressing

Classic Synchronization Problems

- **Producer-Consumer Problem:**
  - **Infinite Buffer:** The producer can always add items to the buffer without concern for capacity.
  - **Finite Buffer:** The producer must wait if the buffer is full, and the consumer must wait if the buffer is empty.
  - **Solution:** Use semaphores to manage buffer availability and ensure mutual exclusion between producer and consumer operations.
- **Readers-Writers Problem:**
  - Multiple readers can access shared data simultaneously, but writers require exclusive access to modify the data.
  - **Solution:** Implement a turnstile mechanism using semaphores to prioritize writer access when necessary, while still allowing concurrent reader access.
- **Dining Philosophers Problem:**
  - Philosophers must alternately think and eat. Each philosopher needs two forks to eat, but there are fewer forks than philosophers, leading to potential deadlock or starvation.

3. Parallel Computing Architectures

Bit Level Parallelism

Increasing word size, size of architecture → process more bits at the same time

- Unit of transfer between processor, memory
- Memory address space capacity
- Integer size
- Single precision floating point number size

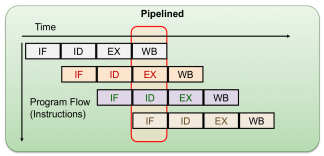
Instruction Level Parallelism

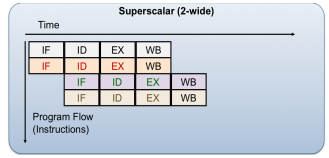
Pipelining

- Split instruction execution into multiple stages:
- Allow multiple instructions to occupy different stages in same clock cycle
- Number of pipeline stages == Maximum achievable speedup
- **Disadvantages**:
  - Independence, Bubbling, Hazards: data dependencies/control flow
  - How to solve? Speculation, Out-of-order execution (read-after-write)

Superscalar

- Duplicate pipelines allow multiple instr. to pass through same stage
- Scheduling is challenging (decide which instructions executed tgt)
- Dynamic (hardware decision) vs Static (compiler decision)
- **Disadvantages**: Structural hazards
  - How to solve? Convert cycles-per-instructions (CPI) to instructions-per-cycle (IPC)





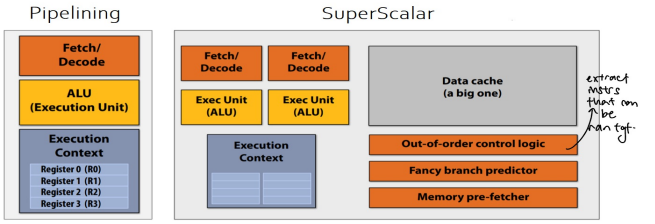
Pipelined

1. Determine instr. to run next
2. Execution unit: performs operation described by instr.
3. Registers: store val of variables used as I/Oputs to operations

Superscalar

1. Processor automatically finds independent instr. in instr. seq. and execute them in parallel on execution units
2. **Instructions come from same execution flow** (thread)

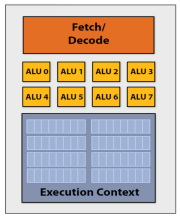
⋆ Instructions came from same execution flow, more instr. ran in parallel, requires **larger data cache**.
⋆ **Out-of-order control** logic unit extract instr. that can be ran tgt
⋆ **Fancy branch predictor** to guess which branch to exec.
⋆ **Memory pre-fetcher** brings data from memory into caches early to reduce overall execution time



SIMD — Single Instruction, Multiple data

Add more ALUs to increase compute capabilitys. Same instruction broadcasted to and executed by ALL **ALUs**.

- Suitable for processing sets of data for the same instruction.
- Better than superscalar as there might not be enough Instruction Level Parallelism, but math operations are long (compute heavy).
- Same instructions processes multiple times with **different data**
- Singular execution Context (registers/program counter/stack ptrs)



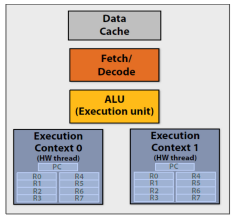
Thread Level Parallelism

Limited ILP as typical programs only 2-3 instr. executed in parallel due to data/control dependencies. We want processor to **execute threads in parallel**

SMT — Simultaneous Multi-Threading

Provide hardware support for multiple 'thread contexts'

- Run one scalr instruction per clock from one of the hardware threads
- 2 Logical cores → 2 execution contexts, stores PC, Register info
- Unlike *superscalar*, only store value of variables used for I/O from a **singular execution flow**
- a.k.a **hyper-threading** in Intel Processors



Processor Level Parallelism (Multiprocessing)

Add more cores to processor, application should have **multiple execution flows**

- Each processor/thread has independent context, can be mapped to multiple processor cores

Flynn's Parallel Architecture Taxonomy

Commonly used taxonomy of parallel architecture

- **Instruction Stream**: single exec. flow, e.g. Program Counter
- **Data Stream**: data being manipulated by instruction stream

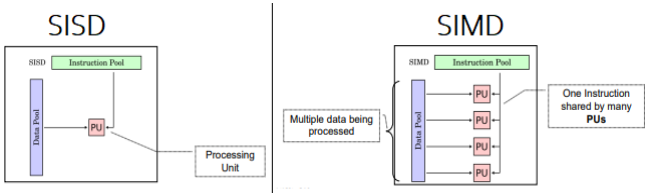
Single Instruction Single Data (SISD)

Single instruction stream executed, each instruction work on single data → most uniprocessors fall into this category

Single Instruction Multiple Data (SIMD)

Each instruction works on multiple data from a singular stream of instructions (e.g. 1 PC)

- Popular model for supercomputer during 1980s, exploiting **data parallelism**, commonly known as vector processor
- Not good for divergent executions
- Data Parallel Architectures: ACX instructions, GPGPUs
  - **AVX:** Intrinsic functions operates on vectors of 4 64-bit values (e.g. vector of 4 doubles)

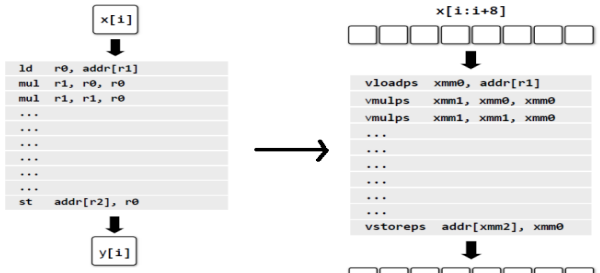


Original Program → Vector Program using AVX intrinsics

Original: processes 1 array element using scalr instr. on scalar registers (e.g. 32 bit floats)

Vector Program:

- Intrinsic functions operate on vectors of 8 32-bit values (vector of floats)
- Proceses eight array elements simultaneously using vector instructions on 256-bit vector registers




```
for (int i=0; i<N; i++)
{
   float value = x[i];
   float numer = x[i] * x[i] * x[i];
   int denom = 6;  // 3!
```

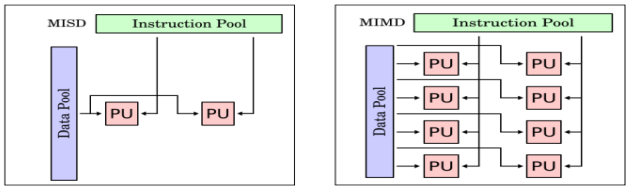
```
   float three_fact = 6;  // 3!
   for (int i=0; i<N; i+=8)
   {
      __m256 origx = _mm256_load_ps(&x[i]);
      __m256 value = origx;
      __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
      __m256 denom = _mm256_broadcast_ss(&three_fact);
```

Multiple Instruction Single Data (MISD)

Multiple instr. streams, all instructions work on same data at any time → **NO** actual implementation

Multiple Instruction Multiple Data (MIMD)

Each PU fetch its own instruction, and operates its own data



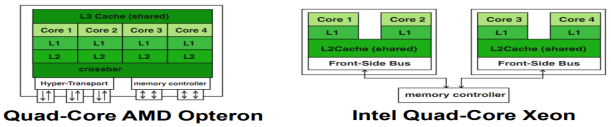
Variant — SIMD + MIMD (NVDIA GPUs)

- Set of threads executing the same code (effectively SIMD)
- Multiple set of threads executing in parallel (effectively MIMD)

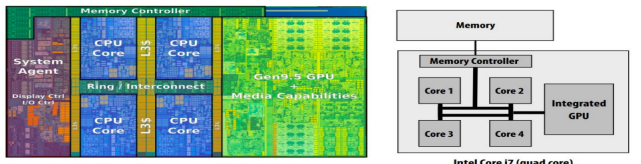
3.1 Multicore Architecture

Hierarchical Design

- Multiple cores share multiple cache, cache size ↑ from leaves to root
- Each core can have separate L1 cache and share their L2 cache
- All cores share common external memory



Quad-Core AMD Opteron **Intel Quad-Core Xeon**

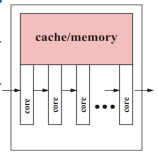


Example: Intel Core i7 processor (Kaby Lake)

Intel Core i7 (quad core) (interconnect is a ring)

Pipeline Design

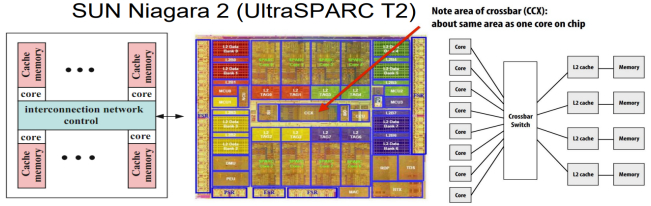
- Elements processed by multiple execution cores in **pipelined way**
- Useful if same computation steps have to be applied to a long sequence of data elements
- e.g. processors used in routers / graphics processors



Network-Based Design

Cores, local caches/memories connected by **interconnection network**

SUN Niagara 2 (UltraSPARC T2)



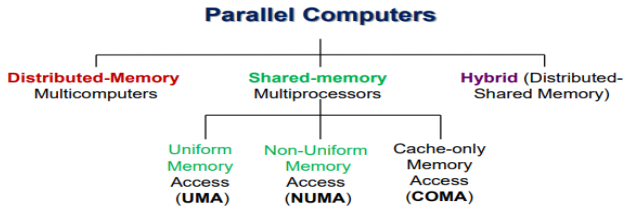
Future Trends

Efficient on-chip interconnection, Network on Chip (NoC)

- Enough bandwidth for data transfers between cores
- Scalable and Robust to tolerate failures
- Efficient energy management
- Reduce memory access time

3.2 Memory Organization

⋆ ↓ memory fetches and reuse data previously loaded by same thread
⋆ Favor performing additional compute instead of storing / reload values → **Programs must access memory infrequently to utilize modern processors efficiently**

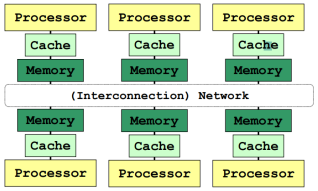


Components in a uniprocessor: Core $\rightarrow \geq 1$ or more levels of caches $\rightarrow$ Memory module $\rightarrow$ Other (e.g. I/O)

- **Memory Latency**: Amt of time for memory request (`load`, `store`) from processor to be serviced by memory system
- **Memory Bandwidth**: Rate at which memory system can provide data to processor

Distributed Memory

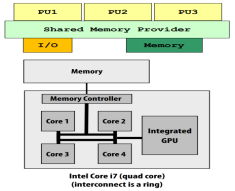
Each node is an *independent unit*, with processor, memory etc.



Physically distribute memory module, memory in a node is private. Requires explicit communication between 2 nodes to 'share memory'

Shared Memory System

- Parallel programs / threads access memory through shared memory provider
- Program unaware of actual hardware memory architecture
  - Cache coherence & memory consistency

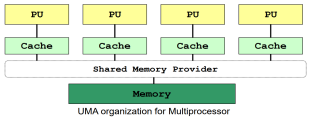


Cache Coherence

- Multiple choices of same data exists on different caches
- Local update by processing unit $\rightarrow$ Other PUs **should not see unchanged data**
- Factors to differentiate **shared memory systems**
  - Processor to Memory Delay (UMA / NUMA)
  - Delay to memory is uniform
- Presence of local cache with cache coherence protocol (CC / NCC)
  - Same shared variables may exist in multiple caches
  - Hardware ensures correctness via cache coherence protocol

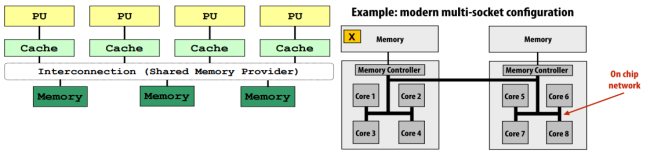
Uniform Memory Access (Time) UMA

- Latency of accessing main memory same for every processor
- Suitable for **small number** of PU due to *contention*



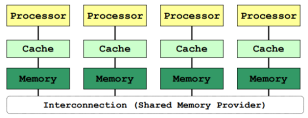
Non-Uniform Memory Access NUMA

- Physically distributed memory of all processing units combined to form a global shared-memory address space: **distributed memory**
- Access local memory faster than remote memory for PU



ccNUMA

- Cache Coherent Non-Uniform Memory Access
- Each node has cache memory to reduce *contention*



COMA — Cache Only Memory Architecture

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

ADVANTAGES	DISADVANTAGES
1. No need to partition code/data	1. Special synchronization constructs required
2. No need to physically move data among processors $\rightarrow$ efficient communication	2. Lack of scalability due to *contention*

4. Parallel Programming Models

Types of Parallelism

Data Parallelism

Partition data used to solve problem amongst PU, each PU carries out *similar operations* on its part of data

- Same operation applied to different elements
- If operations independent, elements can be distributed among cores for parallel execution (Loop Parallelism)
- Executed in arbitrary order and parallel in different cores

1. Parallel For in OpenMP: OpenMP API, multi-platform shared-memory multi-processing programming

```
// Parallelize the matrix multiplication (result = a x b)
// Each thread will work on one iteration of the outer-most loop
// Variables are shared among threads (a, b, result)
// and each thread has its own private copy (i, j, k)
#pragma omp parallel for num_threads(8)
        shared(a, b, result) private (i, j, k)
for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
        for (k = 0; k < size; k++)
            result.element[i][j] += a.element[i][k] *
                                    b.element[k][j];
```

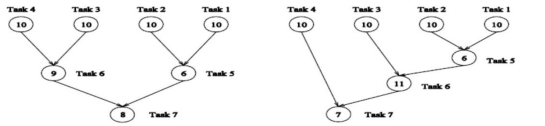
2. Data Parallelism on MIMD: SPMD (Single Program Multi Data)
   - One parallel program executed by all cores in parallel (shared / distrbuted address space) e.g. Scalar prroduct of $x \cdot y$ on p PU

```
1 local_size = size/p;
2 local_lower = me * local_size; // me = PU Index
3 local_upper = (me+1) * local_size - 1;
4 local_sum = 0.0
5 for (i=local_lower; i<local_upper; i++) {
6     local_sum += x[i] * y[i];
7 }
8 Reduce(&local_sum, &global_sum, 0, SUM);
```

Task (Functional) Parallelism

Partition tasks in solving problem amongst PU (singular/series of statements, loops or funcction calls)

1. **Further Decomposition:** single task executed sequentially by 1 PU, or in parallel by multiple PU
2. **Task Dependence Graph:** visualize/evaluate decomp. strategy
   - **Directed Acyclic Graph**
     - **Node:** Task, node value is expected execution time
     - **Edge:** *control dependency* between tasks
   - Properties:
     - Critical path length: Maximum (slowest) completion time
     - Degree of concurrency: $\frac{Total\ Work}{Critical\ Path\ length}$
       - Indication of amount of work that can be done concurrently



Models of Coordination

Any type of coordination can be implemented in any hardware, even those that do not match the architecture

Shared Address Space

- Communication Abstraction
  - Tasks communicate by read/write to/from shared variables
  - Ensure mutual exclusion via locks, logical extension of uniprocessor programming
- Requires hardware support to implement efficiently
  - Any processor can load store from any address — contention
  - Even with NUMA costly to scale
- Matches shared memory systems — UMA, NUMA

Data Parallel

- SIMD, Vector processors $\rightarrow$ same operation on each element of array
- Basic Structure: map function onto large collection of data
  - Functional: size-effect-free execution
  - **No communication** among distinct function invocations
    - Allows invocations to be scheduled in parallel
  - Stream programming model
- Modern performance-oriented data-parallel languages do not enforce this structure anymore

Message Passing

- Tasks operate within own private address space, communicate by explicitly sending/receiving messages
- e.g. Message Passing Interface (MPI)
- Hardware does not implement system-wide loads/stores
  - No shared memory within this entire system
  - Connect commodity systems tgt to form large parallel machine
- Matches distributed memory system $\rightarrow$ clusters/supercomputers etc.

Correspondence with Hardward Implementations

1. Message-passing abstractions in shared memory system
   - Send Message $\rightarrow$ copy data into message library buffers
   - Receive Message $\rightarrow$ copying data from this library buffers
2. Possible to implement shared address space abstractions that do not support the hardware
   - Less efficient, modify shared variable: send message to invalidate all pages (memory)
   - Reading shared variable: page-fault handler to issue appropriate network requests (messages)

Program Parallization

Granularity of computation (from Fine-Grain $\rightarrow$ Coarse-Grain)

1. sequence of instructions
2. sequence of statements where each statement consists of several instructions
3. function / method which consists of several statements

Foster's Design Methodology

1. Partitioning

Divide computation & **data** into independent pieces to $\uparrow$ parallelism

1. **Data Centeric — Domain Decomposition**
   - Divide data into pieces of approximately equal size
   - Determine how to associate computations with data
2. **Computation Centric — Functional Decomposition**
   - Divide computation into pieces (tasks)
   - Determine how to associate data with computations
3. **Partitioning Rules of Thumb**
   - 10x more primitive tasks than cores in target computer
   - Minimize redudant computations and redundant data store
   - Primitive tasks roughly same size (keep tasks finishing tgt)
   - Number of tasks an increasing function of problem size

2. Communication (Coordination)

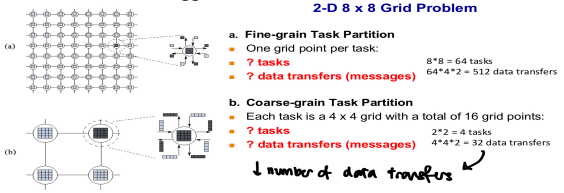
- Usually intended to execute in parallel, but not executed independently $\rightarrow$ determine data passed among tasks
  - **Local communication**
    - Tasks needs data from small number of other tasks (neighbours)
    - Create channels illustrating data flow
    - e.g. 2D grid, require 5 values from neighbour to update each element
  - **Global communication**
    - Significant number of tasks contribute data to perform computation
    - Do not create channels for them early in design

    - e.g. Unoptimized sum N numbers distributed amongst N tasks (centralised - no distribution of computation/communication and sequential - no overlap)
  - Ideally, distribute and overlap computation and communication
  - **Communication Rules of Thumb**
    - Operations balanced among tasks
    - Each tasks communicates with only small group of neighbours
    - Tasks can perform communication in parallel
    - Overlap computation with computation

3. Agglomeration

Combine tasks into larger tasks → Number of tasks $\geq$ number of cores

- Goals:
  - Improve performance (cost of task creation + communication)
  - Maintain scalability of program
  - Simplify programming
- Motivation:
  - Eliminate communication between primitive tasks agglomerated into consolidated tasks
  - e.g. combined groups of sending/receiving tasks
- **Agglomeration Rules of Thumb**
  - Locality of parallel program increased
  - Number of tasks increases with problem size
  - Number of tasks suitable for likely larger systems
  - Tradeoff between agglomeration and code modification costs is reasonable



4. Mapping

- Assignment of tasks to execution units
- Conflicting Goals
  - **Maximize processor utilization:** place tasks on different PU to ↑ parallelism
  - **Minimize inter-processor communication:** place tasks that communicate frequently on same PU to $uparrow$ locality
- Performed by: OS for centralized multiprocessor, or user for distrbuted memory systems
- **Mapping Rules of Thumb**
  - Finding optimal mapping is NP hard (heuristic needed)
  - Consider design based on 1 task/core and multiple tasks/core
  - Evaluate static/dynamic task allocation
    - Dynamic tasks allocation chosen, task allocator should not be bottleneck to performance
    - Static tasks allocation chosen ratio of tasks to cores is $\geq 10:1$

Automatic Parallization

Parallizing compilers perform decomposition + scheduling, drawbacks:

- Dependence analysis difficult for pointer-based computations / indirect addressing
- Execution time of function calls/loops with unknown bounds is difficult to predict at compile time

Functional Programming Languages

Describe computations of program as evaluation of mathematical functions without side-effects

- Advantages: new langugage constructs not necessary to enable parallel execution
- Challenges: Extract parallelism at right level of recursion

Parallel Programming Patterns

— design patterns in SWE, not mutually exclusive → best match

Fork-Join

- Task T creates child tasks
  - Children runs in parallel, independent of one another
  - Children execute at same time / different program part / different function
  - Children might join parent at different time
- **Implementation:**
  - Processes, threads and any parallel paradigm using this concepts
  - Independent **execution flows**

Parbegin-Parend

- Programmer specifies sequence of statements (fn calls) to be executed by set of cores in parallel
  - Executing thread a **parbegin-parend construct**, a set of threads created
  - Statement of construct assigned to these threads for execution
- Statement following **parbegin-parend construct** only executed after *all* these threads finished their work
- Like fork-join but **all forks and joins** are done at same time
- **Implementation:**
  - language construct like OpenMP/compiler directives

SIMD — Pattern

- **Single instructions** executed synchronously diff threads, diff data
  - Similar to parbegin-parend but threads execute synchronously
- **Implementation:** AVX/SSE instruction on Intel processor

SPMD

- **Same program** executed on different cores, operate on diff data
  - Different threads execute different parts of parallel program
    - Differnet speeds of executing coress / Control statements of program (`if` statements)
  - Similar to parbegin-parend, SPMD preferred name when we do not follow the pattern
- No implicit synchronization, can be acheived by using explicit synchronization operations
- **Implementation:** Programs running on GPGPU

Master-Work (previously Master-Slave)

- Single program (master) controls execution of program
  - Master executes main function, assigns work to worker threads
- **Master Task:** generally responsible for coordination and performs initializations, timings and outputs operations
- **Worker task:** wait from instructions from master tasks

Task (Work) Pools

- Common data structure, threads can access to retrieve tasks for execution
- Number of threads is fixed
  - Threads created statically by main thread
  - Tasks finished, worker thread retrieves another task from pool
  - Work not pre-allocated to the worker threads; instead new task is retrieved from pool by worker
- During processing of task, thread can generate new tasks and insert them into the pool
- Access to task pool must be **synchronised** to avoid race conditions
- Execution of parallel programs completed when
  - Task pool empty & each thread terminated processing of last task
  - For uneven tasks, to ensure (fairly) even distribution from tasks
- Advantages:
  - Useful for adaptive/irregular applications, generate tasks dynamically
  - Overhead for thread creation independent of problem size/number of tasks
- Disadvantages: Fine-grained tasks, overhead of retrievaland insertion of tasks becomes important

Producer-Consumer

Producer threads produce data → used as inputs by consumer threads

- Synchronisation has to be used to ensure correct coordination between producer and consumer threads

Pipelining

Data in application partitioned into stream of data elements, flow into pipeline stages sequentially → **perform different processing steps**

- Form of functional parallism: **Stream paralleism**

5. Performance of Parallel Systems

- **Users:** reduced response time (between start and termination of program)
- **Computer managers:** high *throughput*, avg number of work units/unit time

Sequential Programs

Response times in Sequential Programs

- **wall-clock time** — includes the following:
  - *User CPU*: Time CPU spends executing program
  - *System CPU*: time CPU spends executing OS routines
  - *Waiting*: I/O waiting time, execution of other programs (time sharing)
  - *Considerations:*
    - waiting time — depends on load of computer system
    - system CPU time — depends on the OS implementation

5.1 User CPU: Time CPU spends executing program

- translation of program statements by compiler into instructions + execution time ⇒ $\boxed{Time_{user}(A) = N_{cycle}(A) \times Time_{cycle}}$
  - $Time_{user}(A)$ — User CPU time of program A
  - $N_{cycle}(A)$ — Total no. of CPU cycles needed for all instructions
  - $Time_{cycle}$ — Cycle time of CPU (clock cycle $= \frac{1}{clock\ rate}$)
- instructions may have different execution times, instructions $I_1, \ldots, I_n$ ⇒ $\boxed{N_{cycle}(A) = \Sigma_{i=1}^{n} n_i(A) \times CPI_i}$
  - $n_i(A)$ — number of instructions of type I_i
  - CPI_i — avg. no. of CPU cycles needed for instructions of type I_i
- ⇒ $\boxed{Time_{user}(A) = N_{instr}(A) \times CPI(A) \times Time_{cycle}}$
  - $CPI(A)$ — depend on internal org. of CPU, memory, compiler
  - $N_{instr}(A)$ — total no. instr exec, depend on architecture/compiler

5.1.1. Refinement with Memory Access Time

- Include memory access time to user time ⇒ $\boxed{Time_{user}(A) = (N_{cycle}(A) + N_{mm\_cycle}(A)) \times Time_{cycle}}$
  - $N_{mm\_cycle}(A)$ — no. of additional clock cycles due to memory access
  - One level-cache: $N_{mm\_cycle}(A) = N_{read\_cycle}(A) + N_{write\_cycle}(A)$
  - $N_{read\_cycle}(A) = N_{read\_op}(A) \times R_{read\_miss}(A) \times N_{miss\_cycles}(A)$

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

  - $N_{rw\_op}(A)$ — total no. of read/write ops
  - $R_{miss}(A)$ — (read and write) miss rate
  - N_{miss_cycles} — no. of add. cycles needed for loading new cache line

5.1.2 Average Memory Access Time

- $\boxed{T_{read\_access}(A) = T_{read\_hit} + R_{read\_miss}(A) \times T_{read\_miss}}$
  - $T_{read\_access}(A)$ — avg. read access time of program A
  - T_{read_hit} — time for read access to cache regardless of hit/miss
  - $R_{read\_miss}(A)$ — cache miss rate of program A
  - T_{read_miss} — read miss penalty time
- applicable to multiple level of cache/ virtual memory
  - 1st Level: $\boxed{T_{read\_access}(A) = T^{L1}_{read\_hit} + R^{L1}_{read\_miss} \times T^{L1}_{read\_miss}}$
  - 2nd Level: $\boxed{T^{L1}_{read\_miss}(A) = T^{L2}_{read\_hit} + R^{L2}_{read\_miss} \times T^{L2}_{read\_miss}}$
  - **Global miss rate:** $\boxed{R^{L1}_{read\_miss} \times R^{L2}_{read\_miss}}$

Throughput: Million-Instruction-Per-Second

$$\boxed{MIPS(A) = \frac{N_{instr}(A)}{Time_{user}(A) \times 10^6}} \text{ — — } MIPS(A) = \frac{clock\_frequency}{CPI(A) \times 10^6}$$

- Drawbacks: consider only number of instructions
- Easily manipulated — more instructions that do less (depends on ISA)

Throughput: Million-Floating point-Operation-Per-Second

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

- $N_{fl\_ops}(A)$: no. of floating-point operations in program A
- Drawbacks: no differentiation between diff. types of floating-point ops

5.2 Parallel Programs $T_p(n)$

- T — End time Start time of parallel program on **all** processors
- p — p processing units & n — problem of size n
  - Comprised of time for: executing local **computations**, **exchange of data** & **synchronization** between PU. — **Waiting time** due to unequal load distribution of PU and wait to access shared data

5.2.1 Cost $C_p(n) = p \times T_p(n)$

- $C_p(n)$ measures total amount of work performed by all processors (PUs) i.e. **processor-runtime product**
- *Parallel Program* is cost-optimal if executes the same total no. of operations as the fastest sequential program

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

- Theorectically $S_p(n) \leq p$ always holds — In practice $S_p(n) > p$ (superlinear speedup) can occur, e.g. problem working tasks 'fits' cache

5.2.3 Efficiency $E_p(n) = \frac{T_*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T_*(n)}{p \times T_p(n)}$

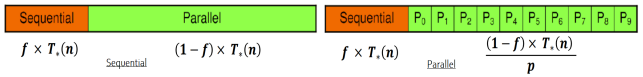
- T_* — shorthand for T_{best_seq}, ideal speedup $S_p(n) = p \Rightarrow E_p(n) = 1$

5.2 Scalability

Amdahl's Law: Speed up of parallel execution limited by fraction of algorithm that cannot be parallized (f)

- $f (0 \leq f \leq 1) \rightarrow$ sequential fraction or fixed-workload-performance
- In many computing problems f is not constant, commonly dependent on problem size $n \rightarrow f$ is function of $n, f(n)$
- **Effective parallel algorithm:** $\lim_{n\to\infty} f(n) = 0$
- **Speedup:** $\lim_{n\to\infty} S_p(n) = \frac{1}{1+(p-1)f(n)} = p$

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



Gustafson's Law: applications where main constraint is application time

- high computing power used to improve accuracy / better result
- If f is not constant but $\downarrow$ when problem size $\uparrow$ then $S_p(n) \leq p$
- T_f = constant execution time for sequential part
- $T_v(n,p)$ = execution time of parallizable part for problem size n and p processors: $S_p(n) = \frac{T_f + T_v(n,1)}{T_f + T_v(n,p)}$
- Assume parallel program is perfectly parallizable (without overheads)
  - $T_v(n,1) = T^*(n) - T_f$ and $T_v(n,p) = (T^*(n) - T_f)/p$
  - $S_p(n) = \frac{T_f + T_v(n,1)}{T_f + T_v(n,p)} = (\frac{T_f}{T^*(n) - T_f} + 1) \div (\frac{T_f}{T^*(n) - T_f} + \frac{1}{p})$
  - if $T^*(n) \uparrow$ strongly monotonically with n, then $\lim_{n\to\infty} S_p(n) = p$

Scaling with problem size

- Small problem size:
  - Parallelism overhead dominates parallelism benefits
  - Problem size may be appropriate for small machine but not for large ones
- Large problem size:
  - Key working set may not 'fit' small machine, $\rightarrow$ thrashing to disk, key working sets exceed cache capacity

Scaling Constraits

- **Application-Oriented scaling properties** — specific to application
  - combination of parameters not necessarily one number only
- **Resource-oriented scaling properties**
  - *Problem constrained (PC)*: use parallel com to solve same problem faster
  - *Time constrainted (TC)*: completing more work in fixed amount of time
  - *Memory constrained (MC)*: run the largest problem possible without overflowing memory

Arithmetic Intensity

- $= \frac{amount\ of\ computation}{amount\ of\ communication}$ e.g instructions $\div$ bytes
- if numerator = exec. time, ratio = avg bandwith required of code
- high arithmetic intensity (low communication-to-computation ratio) required to *efficienty* utilized modern parallel processors

Contention

- resource can perform operations at given throughput $\rightarrow$ memory, communication links, servers
- occurs when many requests to resource made within small window of time (**hot spot**)

Locality & Cache lines

- **Temporal Locality:**
  - **Exploit sharing:** co-locate tasks that operate on same data
  - schedule threads working on same data structure at same time on same processor $\rightarrow$ reduce communication
- **Spatial Locality:**
  - avoid sharing cache lines among tasks running on different cores in parallel
  - how data stored in memory (layout), **padding** to avoid cache line sharing
- allocate work to tasks to take advantage of prefetching

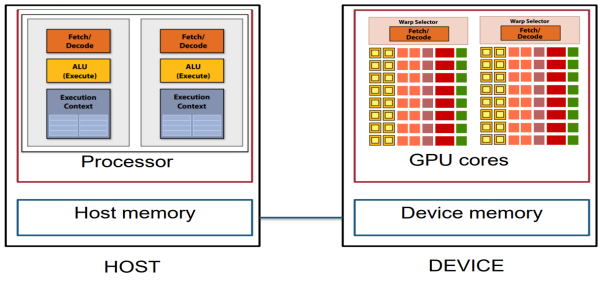
5.3 Performance Analysis

- **Bottlenecks:**
  - *Instruction-rate:* add 'math' (non-memory) instructions
  - *Memory bottleneck:* remove almost all math but load same data
  - *Locality of data acess:* change all data access to `A[0]`
  - *Sync overhead:* remove all atomic operations or locks (provided same amt of work done)

5.4 Communication Time

- **Sender**
  - **Sender processor:** When sending message — message $\rightarrow$ system buffer $\rightarrow$ checksum computer and header added
  - *After sending message:* ACK arrives, release system buffer, if timer elasped message is re-sent
- **Receiver**
  - **Receiving processor:** message copied from NI $\rightarrow$ system buffer
  - *After receiving message:* checksum computed, **mismatch** — discard and re-sent to sender, **identical** — message copied from system buffer into user buffer, application program gets notification and continue execution
- Total Latency of message of size `m`: $T(m) = O_{send} + T_{delay} + \frac{m}{B} + O_{recv} = T_{overhead} + \frac{m}{B} = T_{overhead} + t_B * m$
  - B network bandwith, T_{delay} time first bit arrives to receiver
  - $T_{overhead} = (O_{send} + T_{delay} + O_{recv})$ — indep. of message size
  - $t_B = \frac{1}{B}$ is byte transfer time

6. GPGPU Programming



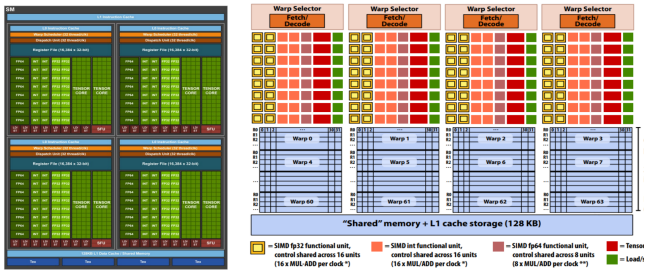
GPU Architecture

- Mutliple Streaming Multiprocessors (SMs)
  - Memory and cache, connectin interface(PCIE, HBM NVLink)
- SM consists of multiple compute cores
  - Memories (registers, L1/L2 cache and shared memory)
  - Logic for thread and instruction management
- **Hopper Streaming MultiProcessor:** 'latest' version
  - 64 INT32 CUDA Cores/SM
  - 128 FP32 CUDA Cores/SM
  - 64 FP64 CUDA Cores/SM
  - Four 4th Gen Tensor Cores/SM

Execution Model Mapping to Architecture

- **SIMT (single instruciton, multi-threaded)** execution model
- Multiprocessors creates,manages,schedules,executes threads in SIMT warps
- **Wraps:** Groups of 32 parallel thread
  - Threads in warp start together at same program address
  - Individual instr. program counter and register state
  - Ideally threads execute in **lock-step** $\rightarrow$ all of them in sync in *same step*
- Wrap executes one common instruction at a time
- Full efficiency realized $\rightarrow$ *all 32 threads* of warp agree on their exec. path

NVIDIA V100 SM



CUDA Programming Model — Compute Unified Device Architecture

- Proprietary programming model (NVIDIA)
- Massively hardware multi-threaded, designed to scale well over time
- General purpose programming model
  - Simple extension to standard `C`, mature software stack
  - Launch batches of threads on GPU, full general load/store memory model (CRCW) and enable *heterogenous systems* (CPU + GPU)

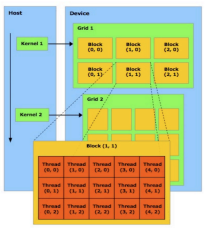
CUDA Kernel and Threads

Device = GPU, Host = CPU, Kernel = functions that run on **device**

- Parallel portions execute on device by multiple CUDA threads
- CUDA threads **lightweight**, little overhead creation and instant switching
- CUDA uses *thousands* of threads $\rightarrow$ transparently scales to hundreds of cores and 1000s of parallel threads
- CUDA Kernal executed by *array of threads*
  - Logically at program level, thread grouped into **blocks**
  - All threads run the kernel (SIMT model)
  - Each thread has ID, uses to compute mem address, make control decisions
- Thread in array *need not be completely independent*
  - Share results to save computation, share memory access
  - In a block the following are shared $\rightarrow$ different blocks cannot cooperate
    - Shared memory, atomic operations and barrier synchronization

Transparent Scalability

- Hardware free to schedule thread blocks to any SM
  - Kernel scales across any number of parallel Multiprocessors
  - Each block can execute in any order relative to other blocks
- **Logical (Virtual) Thread Hierarchy**
  - **Kernel** executed by a **grid** of thread blocks
  - **Grid** divided into **blocks**: Block ID: 1D/2D/3D
  - **Blocks** contain multiple threads:
    - Thread ID: 1D/2D/3D, share data through shared memory, sync exec.
    - Threads from diff blocks cannot cooperate
  - Thread may use ID to decide data to work on
    - Simplifies memory addressing when processing multi-dimensional data



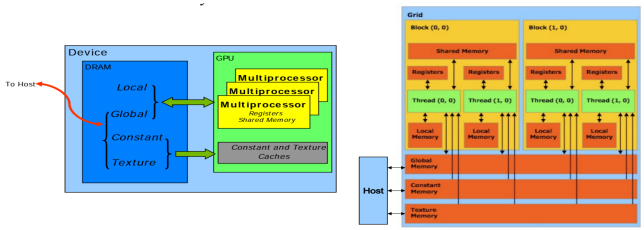
Execution Model Mapping to Architecture

- Kernels launch in **grid of blocks** ≥ 1 kernels execute at a time on an SM
- Block executes on streaming multiprocessor: DOES NOT MIGRATE DATA
- Several blocks reside concurrently on on SM
  - control limitations (depending on capability)
  - number limited by SM resources
    - register file is partitioned among all resident threads
    - shared memory is partitioned among all resident thread blocks''

- SM partitions threads (of a block) into warps and each warp scheduled by *warp scheduler* for execution → way warps are patition is **consistent**
  - Warp contains threads of consecutive, ↑ thread IDs (first warp thread 0)
  - Warps take turn to execute on SM until all threads of block finish

CUDA Memory Spaces

- Data must be explicitly transferred from CPU to device
- Global memory and Shared Memory — most important and commonly used
  - Global memory is cached while shared memory not cached
  - Part of shared memory is the *L1 Cache*
- Local, Constant and Texture memory for convenience/performance
  - **Local:** automatic array of variables allocated by compiler (cached)
  - **Constant:** useful for uniformly-accessed read-only data (cached)
  - **Texture:** useful for spatially coherent random-access read-only data → provides filtering, address clamping and wrapping (cached)



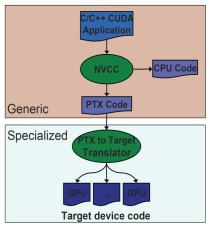
Programming in CUDA

CUDA Programming Interfaces

- CUDA C Runtime
  - Minimal set of extensions to the C language
  - Kernels defined as C functions embedded in application source code
  - Requires a runtime API (built on top of CUDA driver API)
- CUDA driver API
  - Low-level C API to load compiled kernels, inspect parameters, and launch
  - Kernels are written in C and compiled to CUDA binary or assembly code
  - Requires more code, harder to program and debug
  - Much like using the OpenGL API on GLSL shaders

Compiling and Linking CUDA

- NVCC: compiler driver for CUDA source
  - Works by invoking other tools and compilers like cudacc, g++, cl, ...
- NVCC outputs
  - C code (host CPU code) → must then be using another tool
  - PTX → object code/PTX source interpreted at runtime
- Linking with two static/dynamic libraries
  - CUDA runtime library (cudart)
  - CUDA core library (cuda)



Device Code

- C functions with the following restrictions
  - Only access **GPU Memory**
  - No variable number of arguments `varargs`
  - No static variables and no recursions

Function Types

- Function qualifiers: `__host__`, `__device__`, `__global__`
- `__global__` function is a kernel
- Must have `void` return type
- A call to a kernel must specify execution configuration
- Function parameters are passed via shared memory (*)
- `__host__` and `__device__` can be used together

Launching Kernels

- Modified C function call syntax:
  `kernel<<<dim3 grid, dim3 block, int smem, int stream>>>(...)`
- Execution Configuration (<<< >>>):
  - **Grid dimensions:** `x` and `y`
  - **Thread-block dimensions:** `x`, `y`, and `z`
  - **Shared memory:** number of bytes per block for external `smem` variables declared without size. → Optional, 0 by default
  - **Stream ID** → Optional, 0 by default

CUDA Built-In Device Variables

- All `__global__` and `__device__` functions have access to these automatically defined variables:
  - `dim3 gridDim;` → Dimensions of the grid in blocks (`gridDim.z` unused)
  - `dim3 blockDim;` → Dimensions of the block in threads
  - `dim3 blockIdx;` → Block index within the grid
  - `dim3 threadIdx;` → Thread index within the block

Variable Qualifiers (Device Code)

- `__device__`
  - Stored in global memory (large, high latency, no cache)
  - Read/write by all threads within grid
  - Written by CPU via `cudaMemcpyToSymbol()`, has *application lifetime*
- `__constant__`
  - Same as `__device__`, but cached and read-only by threads within the grid
  - Written by CPU via `cudaMemcpyToSymbol()`, has *application lifetime*
- `__shared__`
  - Stored in on-chip shared memory (very low latency)
  - Read/write by all threads in the same thread block, has *block lifetime*
- Unqualified variables (in device code):
  - Scalars and built-in vector types are stored in registers
  - Arrays > 4 elements or run-time indices are stored in local memory
  - Read/write by thread only, has *thread lifetime*

GPU Memory Allocation / Release / Copy

- GPU memory allocation / release:
  - `cudaMalloc(void **pointer, size_t nbytes);`
  - `cudaMemset(void *pointer, int value, size_t count);`
  - `cudaFree(void* pointer);`
- Data copy:
  - `cudaMemcpy(void *dst, void *src, size_t nbytes, enum ↩ cudaMemcpyKind direction);`
- `enum cudaMemcpyKind`:
  - `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`
- Unified memory model does not need data transfer.
  - `__managed__` → Page-locked host memory

Thread Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block:
  - Generates barrier synchronization instruction.
  - Used to avoid RAW/WAR/WAW hazards when accessing shared memory

Optimizing CUDA Programs

Overall strategies

1. Optimizing memory usage to acheive **maximum bandwidth**
   - Different memory spaces/access patterns have different performance
2. Maximizing **parallel execution**
   - Restructure algorithm to expose as much **data parallelism**
   - Map to hardware to increase occupancy (hardware utilization)
3. Optimizing instruction usage for maximum **instruction throughput**
   - High throughput through *arithmetic instructions*, avoiding different execution paths in same warp

Memory Optimizations

- Minimize data transfers between host/device
- Ensure global memory access coalesced whenever possible
- Minimize global memory access by using shared memory
- Minimize bank conflicts in shared memory access

Data transfer between Host and Device

Device memory → GPU >> host memory → device memory

- batching transfers into one larger transfer performs better than transferring separately
- Use page-locked/pinned memory transfer
  - Pinned memory is not cached, zero-copy feature allowing the threads to directly access host memory

Concurrent data transfers and executions

- Overlap async transfers with computation:
- `cudaMemcpyAsync()` instead of `cudaMemcpy()`
- CPU computation while data transfers, followed by device code executed
- Use different streams to concurrently copy/execute

Coalesced Access to Global Memory

- Simultaneous access to global memory by threads in a **warp** coalesced into number of transactions equal to number of 32-byte transactions necessary to service all threads of the warp
- k^{th} thread accesses k^{th} word in 32-byte aligned array, not all threads need to participate

Shared Memory

- Higher bandwidth/lower latency than local/global Memory
- Divided into equally-sized memory modules (**banks**)
- Bank conflict if 2 addresses of memory request fall into same bank → have to be serialized
- Each bank has bandwidth of 32 bits every clock cycle, successive 32-bit words assigned to successive banks
- Warp size is 32 threads and number of banks is also 32

Strided Access

- threads within warp access words in memory with stride of 2
- results in 50% of load/store efficiency
- half the elements in transactions are not used and represent wasted bandwidth

Execution Configuration

- Balance occupancy and resource utilization.
- Ensure more warps than multiprocessors for improved occupancy.
- Minimum 64 threads per block; multiples of warp size preferred.
- Use smaller thread blocks to avoid memory bank conflicts.
- Block size limited by registers and shared memory; ensure at least one block can run on an SM.
- Avoid multiple contexts per GPU to reduce inefficiencies.

Maximize instruction throughput

- Minimize low-throughput arithmetic instructions.
- Trade precision for speed; prefer single-precision floats.
- Avoid costly integer division and modulo; use bitwise operations.
- Use signed loop counters for aggressive optimizations.
- Optimize functions on char or short types to avoid conversions.

Control flow

- Minimize divergent warps from control flow instructions.
- Reduce instructions by optimizing synchronization points.

7. Cache Coherence

Cache Properties

- **Cache size:** larger cache increase access time, reduce cache misses
- **Block size:** data transferred between main memory and cache in blocks of a fixed length
  - Larger blocks reduces number of blocks but replacement lasts longer
- typical sizes for L1 cache blocks are 4/8 memory words (4 bytes)

Write Policy

- **Write-through:** write access immediately transferred to main memory
  - Advantage: always get newest value of a memory block
  - Disadvantages: slow down → many memory accesses (use write buffer)
- **Write-back:** write performed to main memory when cache block is replaced
  - Advantage: less write operations
  - Disadvantages: memory may contain invalid entries, uses a dirty bit

1. Processor performs write to address 'misses' in cache
2. Cache selection location to place line in cache, if dirty line in current location, its written out of memory
3. Cache loads line from memory (allocates line in cache)
4. Whole cache line is fetched, cache line marked as dirty

Memory Coherence

Coherence ensures that each processing unit has consistent view of memory (each memory location) through its local cache

- All PUs must agree on order of reads/writes to **SAME MEMORY location**
- **Program Order:**
  - Given this sequence, P should get the value written in 1.
    1. Given sequence `P (processing unit)` writes to `X`
    2. No write to `X` (from other processing units)
    3. `P` reads to `X`
- **Write Propagation**
  1. Given a system that meets program order property, the sequence:
     (a) `P_1` (PU) writes to `X`.
     (b) No further writes to `X`.
     (c) `P_2` reads from `X`.
  2. `P_2` should read value written by `P_1`
  3. writes become visible to other processing units **eventually**
- **Transaction Serialization**
  1. Given the sequence
     - Write `V_1` to `X` (by any PU) then Write `V_2` to `X` (by any PU)
     - Processing units can **never** read `X` as `V_2` then **later** `V_1`
  2. All writes to location (by same or different PUs) are seen in same order by all PUs

Tracking Cache Line Sharing Status

- **Snooping Based:** No centralised directory
  - each cache keeps track of sharing status
  - Cache *monitors* or *snoops* bus, to update status of cache line, takes appropriate action → contains a `dirty_bit`
  - Common protocol used in architectures with a a bus
    - all PUs on bus can observe every bus transaction (write propagation)
    - bus transactions visible to PUs in same order (Write serilization)
- **Directory Based:** sharing status kep in centralized location, used with NUMA architectures
- **Implications:** Overhead in shared address space, **Cache ping-pong**: multiple PU read/modify same *global* variable, **False sharing**

Coherence vs Consistency

- Memory consistency: constraints the order in which memory operations performed by 1 thread become visible to other threads for **DIFFERENT memory locations**
  - Consistency deals with when writes to X propagate to other PUs, relative to reads/writes to other addresses

Memory Operations on Multiprocessors

- A program defines a sequence of loads and stores, i.e., "program order."
- Four types of memory operation orderings:
  1. `W → R`: Write to `X` must commit before subsequent read from `Y`.
  2. `R → R`: Read from `X` must commit before subsequent read from `Y`.
  3. `R → W`: Read from `X` must commit before subsequent write to `Y`.
  4. `W → W`: Write to `X` must commit before subsequent write to `Y`.
- Memory operations reordered to hide *write latencies*

Sequential Consistency Model

Every PU issues its memory operations in program order

- Global result of all memory accesses of all PUs appears to all PUs in same **sequential order** irrespective of the arbitrary interleaving of the memory accesses by different PUs
- Effect of each memory operation must be visible to all PUs before the next operation on any PU → as if only *one memory space, one memory operation*
- Sequentially consistent memory system preserves all 4 memory operation orders (W → R, R → W, W → W, R → R)

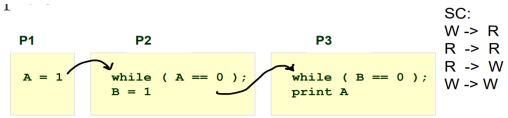
Relaxed Consistency Model

Used to hide latencies, gain performance, hiding memory access operations with other operations when they are independent (overlapping)

- relax the ordering of memory operations if data dependencies allow
- Dependencies: if two operations access the **same memory location**
  - $R \rightarrow W$ anti-dependency (WAR)
  - $W \rightarrow W$ output dependency (WAW)
  - $W \rightarrow R$ flow dependency (RAW)
  - MUST BE PRESERVED
- Program order relaxation: (1) Write → Read, (2) Write → Write, (3) Read → Read or Write

Write-to-Read Program Order

- **Total Store Ordering (TSO)**
  - PU `P` can read `B` before its write to `A` is seen by all PUs
  - PU can move its own reads in front of its own writes
  - Reads by other PUs cannot return new value of A until write to A is **observed** by all other PU (write atomicity property)
- **Processor Consistency (PC)**
  - Return value of any write before write is **observed by all PUs**
  - Write serilization/propagation are preserved buy write atomicity is not
    - Writes *observed eveneutally* by all PU
    - Writes to same memory location observed by all PUs in the same order
    - Writes can be read by some PUs before theya re observed by all PUs



SC:
W -> R
R -> R
R -> W
W -> W

- A = B = 0 initially
- Can A = 0 be printed under the models?
  - SC: A = 0 ×
  - TSO: A = 0 ×
  - PC: A = 0 √ (P3 may not have seen the change yet)

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

Write-to-Write Program Order

Writes can bypass earlier writes (to different locations) in write buffer, allow write miss to overlap and hide latency

- **Partial Store Ordering (PSO)** Relax $W \rightarrow R$ and $W \rightarrow W$

Relaxed consistency models does not address software complexity, exists to increase performance of programs (allowing programmers to write **faster code**), rather it increases software complexity further.

8. Performance Analysis and Instrumentation

Preparation → Profile → Plan → Implement → Commit → repeat Profile ...

Perspectives

Resource analysis (for system admins)

- analysis of system resources: CPU, memory, disks network interfaces, buses and interconnects → Performance issues investigations, capacity planning
- Focuses on *utilization*, demand -supply

Workload analysis (for app developers)

- Examines workload applied and how application is responding
- Targest:
  - Requests: on workload applied
  - Latency: response time of application
  - Completion: Looking for errors
- Metrics: *throughput* and *latency*

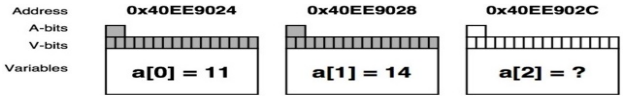
1. what makes you think there is a performance problem?
2. has system performed well?
3. what changed recently
4. can performance degredation be expressed in terms of latency/run time
5. does problem affect other people/applications(or just you)
6. what is the environment (soft/hardware) versions/configs?

Methodologies

- **Anti-methods:**
  - Random change anti-method
  - Streetlight anti-method — look for obvious issues in tools found elsewhere
  - Blame-someone-else-method
- **Methods**
  - Monitoring — records perf statistics over time, so past can be compared to present and *time-based* usage patterns can be identified
    - Useful for: capacity planning, quantifying growth/peak usage
    - Historic values can provide context for understanding current value
  - USE method
    - **U**tilization: busy time
    - **S**aturation: queue length/queue time
    - **E**rrors: easy to interpret
    - Helps if there is a diagram of system/env showing all resources
  - Tools method — list all perf tools for each tool list useful metrics provided, for each metric list possible ways to interpret
    1. *Observability:* watch activity under workload, safe depending on resource overhead, insert timing statements/check perf counters
    2. *Static:* attributes of system at rest instead of under active workload
    3. *Benchmarking:* Load test - prod tests may cause issues (contention)
    4. *Tuning:* change default settings - changes could hurt performance
       - Fixed counters: counters maintaine dby kernel (hardware)
       - Event-based counters: profiling (characterizes target by collecting set of samples), tracing (instruments occurence of event, store event-based details for later analysis)
       - Instrumentation code: modify source code, executable or runtime to understand performance
  - CPU profile method
    - Profile CPU usage by stack sampling/generating CPU flame graphs
      1. `perf` modern architectures expose hardware performance counters
      2. `perf_events` **multi-tool** for CPU profiling, cache profiling, static and dynamic tracing
      3. Flame graphs - each box represents function in stack, x-axis % of time on CPU, y-axis is stack depth

Debugging tools

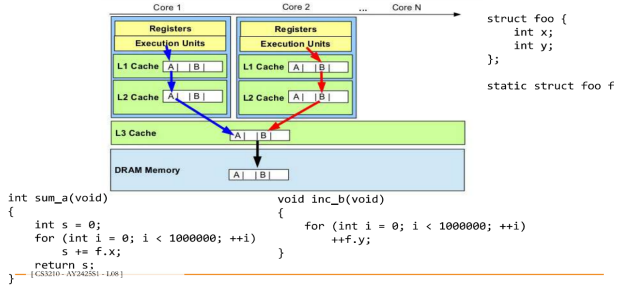
- `Valgrind`
  - Heavy-weight binary instrumentation $> 4 \times overhead$
  - Designed to shadow all program values, requires serializing threads
  - Usually used for memory check
- Santizers (compilation-based approach)
  - e.g. Thread/Memory Sanitizers (data races/unitialized reads)
  - e.g. UndefinedBehaviorSanitizer (Int Overlfow/NPE)
  - e.g. LeakSanitizer (for memory leaks)
- Shadow memory:
  - Used to track/store info on memory used by program during execution
  - Used to detect and report incorrect accesses of memory
    - A-bit: corresponding byte accessible
    - V-bit: corresponding bit initialized



False Sharing

- Identifying using `perf c2c`
- At high level shows cache lines where false sharing detected
- Readers/writers to cache lines and offsets where accesses occured
- pid, tid, instruction addr, fn name, binary object name for r/writers
- source file and line number for each reader/writer
- avg load latency for the loads to those cache lines

Problem: False Sharing



Benchmarking

- Results usually misleading, benchmark workload is synthetic and does not resemble real-world workload
- Common mistakes, testing/choosing wrong target (FS cache instead of disk)
- Benchmark `A` but measure `B` concluded measuring `C` instead
- Difficult to show benchmarks not relevant, so easier to just run them

9. Parallel Programming Models - II

Data Distribution — Blockwise vs Cyclic Data Distribution

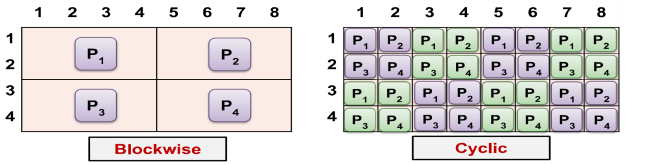
- **Blockwise:** divide into blocks of $B = \lceil \frac{n}{p} \rceil$
  - P_j takes maximum B elements starting from $((j-1) \times B + 1)$
  - suitable for programs that operate on spatially adjacent elements
- **Cyclic:** P_j takes elements $[j, j+p], \cdots j + (\lceil \frac{n}{p} \rceil - 1) \times p]$ if p divides n or $j \leq (n(modp))$
  - Otherwise $[j, j+p], \cdots j + (\lceil \frac{n}{p} \rceil - 2) \times p]$
  - many procecssors, computation per value expensive (need balanced load)

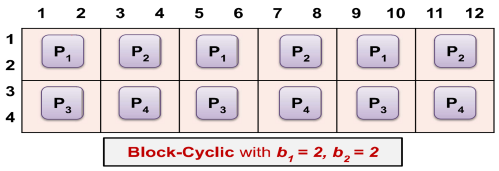
Data distribution for 2D Arrays: Block-Cyclic

- One-dimension distributions
  - Consider **Block-Cyclic** distributions
  - Form blocks of size b then perform cyclic (round robin) allocation

Data distribution for 2D Arrays: Checkerboard

- Processors **virtually organized into 2D mesh of R $\times$ C**
- **Blockwise:** elements split into blocks along both dimensions
- **Cyclic:** cyclic assignment of elements according to processor mesh
- **Block-cyclic:** elements split $b_1 \times b_2$ size nlocks, then cyclical assignment





- Example: heat transfer simulation, what other work do we need to do?
  - P_1 needs to ask P_2 and P_3 for latest data *across* data boundaries
  - If we have more processors but same N by N grids, more communication for each step, more granular work for each processor

Information Exchange

- **Purpose:** make sure every processor receives/has access to relevant information to do its intended task
- **Shared address space:** uses shared variables while **Distributed address space** uses communication operations

Distribution Memory: Communication Operations

- Shared variables do not apply in *distributed memory programming*
- Disjoint memory spaces, exchange data between processors through dedicated communication operations
- e.g. *message-passing programming model*
  1. *point-to-point*
  2. **global/collective communication**

Principles of Message Passing Model

- Data **explicitly partitioned** for each process, interactions require *both parties to participate — explicitly express parallelism*
- **Loosely synchronous paradigm**
  - *Synchronous:* tasks or subsets of tasks sometimes synchronize when communicating / interacting with one another
  - *Asynchronous:* **Between** these limited interactions, tasks execute async

Communication Protocols

Point-to-point, usually sends significant amount of data to one another

Blocking / Non-Blocking

Blocking call returns when resources used in the call may be *re-used safely* by programming

```c
int a = {0, 1, ... , N};
blocking_send(&a, P1);
a[5]=0;
// legal as variable a can be resued after send
```

```c
int a = {0, 1, ... , N};
nonblocking_send(&a, P1);
a[5]=0;
// not legal as variable a might not be safe to use yet
```

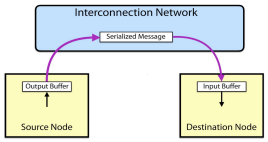
Non-blocking call resources may not be safe to use after returns, requires a `test` function to check if call completed

```c
int a = {0, 1, ... , N};
request = nonblocking_send(&a, P1);
do_other_useful_work();
while(!test(request));
a[5]=0;
// legal as test fn returned true before array reused
```

Buffer vs Non-buffered

Buffered call copies user-provided data into internal buffer then it returns control to the user

- Memory copy speed is faster than network speed, trading off space for time



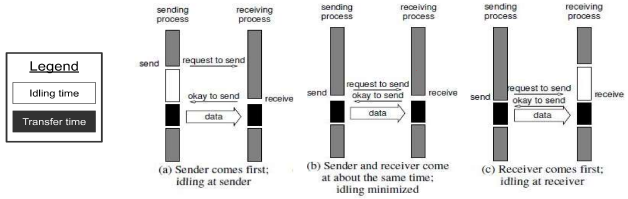
Synchronous vs Asynchronous

Synchronous call send complete only when **matching receive** has started to execute ('non-local' behavior)

- Non-local: requires **coordination with other processes**

Non-buffered + Blocking

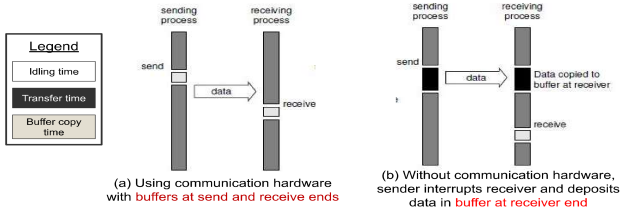
Considerable *idling* overheads, due to **mismatch** in timing btw send/receiver



Buffered + Blocking

Copying data to internal buffer removes idling time

- With hardward support, data transfer entirely in background
- Without, receiver is interrupted to transfer data to its buffer



Bounded Buffer Size: Impact

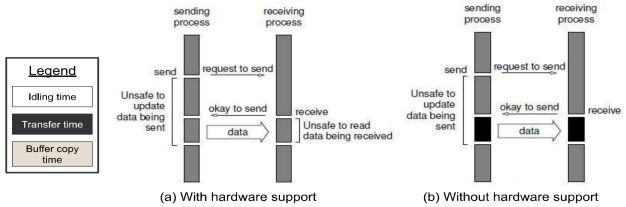
- Buffers have finite size, if consumer slower than producer
- Producer will be blocked, waiting for its buffer to clear
- unforseen delays if this was not already planned for
- Deadlock in blocking operations
  - When blocking `receive` without matching `send`

```c
receive(&a, P1);
send(&b, P1);
```

```c
receive(&a, P0);
send(&b, P0);
```

Non-Buffered + Non-Blocking

- Cannot re-use data until transfer complete, but no idle time
- Hardware support: program encounters little to no overhead



Summary

	Blocking Operations	Non-Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer.	Sending process returns after initiating the transfer to buffer. The copy might not be completed on return.
Non-buffered	Sending process typically blocks until the matching receive has completed.	Sending process typically returns after initiating send request.
	Send and receive semantics assured by when the operation **returns**	Programmer must explicitly ensure completion of the operation using a **test** function

"Local" behavior	"Non-local" behavior
Asynchronous Sender can execute its communication operations without coordination with any other receiver. ***Does not require interaction with other processes***	**Synchronous** Communication operation does not complete before both processes have started their communication operation. ***Requires interaction with other processes***

10. Message Passing

MPI **standardized specification** for message passing libs (fn signatures, behavior, defs)

Program Structure

- `MPI_Init()`: must be **called once and before** any other routines
- `MPI_Send()`, `MPI_Receive()` or other functions
- `MPI_Finalize()`: terminate MPI processing normally, **LAST MPI CALL**
- `MPI_Abort()`: force all MPI processes to terminate

Point-to-Point Communication

- **Blocking:** on return, can re-use resources in call: `MPI_Send`, `MPI_Recv`
- **Non-blocking:** on-return, resources may not be safe to reuse yet: `MPI_Isend`, `MPI_Irecv` $\rightarrow$ both blocking/non-block can be *mixed*
- **MPI Messages Format:**
  - Message = data (actual data to send/rcv) + envelop (how to route data)
  - Data = start-buffer (where data starts) + count (no. of elements in message) + datatype(type of data to be transmitted)
  - Envelop = dest/src (defined via process *rank* in communicator) + tag (arbitrary no. to distinguish msgs) + communicator (set of processes we communicate within)
- Every process is executing a (mostly) independent copy of the program
- Assume 2 processes, gurantee **non-overtaking**: if 1 sender sends 2 msg in succession to same receiver, msg delivered in order they were sent

Blocking Send and Receive

```c
int MPI_Send(void* buf, int count, MPI_Datatype dt, int dest, int tag, MPI_Comm c);
int MPI_Recv(void* buf, int count, MPI_Datatype dt, int src, int tag, MPI_Comm c, MPI_Status *status);
```

- User passes buffer with data (or empty one to rcv)
- src/dest is *rank* of the target process (unique ID)
- Recevied message must be $\leq$ length of rcv buf

- src = `MPI_ANY_SOURCE` from any process, tag = `MPI_ANY_TAG` from any tag (cannot SEND to any process only RECEIVE)
- `MPI_STATUS` is structure with: `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`

MPI Semantics

- **Blocking:** if resources are safe to reuse on function call return
- **Synchronous:** if communications operations (send/rcv) require *the other side* (rcv/send) to start participating before completing
- **Buffered:** if data being send can be copied to internal system buffer to avoid idling while waitng for matching receive
- may be buffered → leave it to MPI to decide buffer or not

This is nowhere near a comprehensive list!	Synchronous	May be asynchronous (asynchronous if buffered)
Blocking	Typically *not buffered:* `MPI_Ssend` `MPI_Rsend`	Send *may* be buffered: `MPI_Send` `MPI_Recv` Buffered (user provided): `MPI_Bsend`
Non-blocking (The letter "I" is used for non-blocking procedures as they are 'incomplete'/ 'immediate')	Typically *not buffered:* `MPI_Issend`	Send *may* be buffered: `MPI_Isend` `MPI_Irecv` Buffered (user provided): `MPI_Ibsend`

Deadlocks in MPI

- When messsage passing not completed (send/receive order)

```
if (my_rank == 0) {
  MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status)
  MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm)
} else if (my_rank == 1) { // 0 always waits for 1 and vice versa
  MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status)
  MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm)
}
```

- Relying on system buffer
  - If runtime system does not use system buffers or system buffers are too small → sends cannot complete

```
if (my_rank == 0) {
  MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm)
  MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status)
} else if (my_rank == 1) {
  MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm)
  MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status)
}
```

- Deadlock free logical ring:
  - Exchange data, process rank i sends data to $i+1$, receives from $i-1$
  - **Even** rank: send → receive, **Odd** rank: receive → send

Non-Blocking Send and Receive

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int
    tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int
    tag, MPI_Comm comm, MPI_Request *request);
```

- Non-blocking calls can be useful to avoid Deadlocks
- Check whether `buf` is safe to reuse/has finally receive data → have to *explicitly test* or *wait* on the `request` object
  1. `MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)` → Returns immediately, flag == true indicates request has completed
  2. `PI_Wait(MPI_Request *request, MPI_Status *status)` → Only returns when request is completed

Process Groups and Communicators

Process Groups ⇒ **ordered set** of processes, each process in group has a **unique rank** consecutively from 0

- processes may be member of **multiple groups** *different ranks* in these groups

Process Groups ⇒ handle that processes can use to comm. with one another

- **Intra-communicators:** supports execution of arbitrary point-to-point and *collective communication* operations on single group (default: `MPI_COMM_WORLD` contains all processes)
- **Inter-communicators:** support communication operations between *2 processor groups*
- Logical separation of processes based on tasks, *collective communication operations* across subset of related processes
- → faster if fewer ranks involved and provided basis for *user-defined* topology

Process Virtual Topologies

- Processes can be given specific logical organizations e.g. processes most communicate with neighbor processes that are
- organized as a mesh, or even an arbitrary graph pattern → Virtual topologies allow neighbors to be easily addressable

Collective Communication

- Operations that invovle **all** processes in communicator, otherwise deadlock!
- Both blocking and non-blocking versions of calls exists
- `int MPI_BArrier(MPI_Comm comm)` the only collective ***synchronization* operation** → processes block until all processes of communicator have started `MPI_Barrier` call

Single Broadcast

```
int MPI_Bcast(void* buf, int count, MPI_Datatype dt, int root, MPI_Comm c);
```

- Sender ('root' process) sends *same block of data* to all other processes
- Modelling tasks via collectives can make scaling up much easier than point-to-point operations
- Every process must call MPI_Bcast, *undefined behavior* not to do so.

Single Broadcast

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
    *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

- Each processor sends the same data block to every other processor, no root processor. Data blocks are collected in rank order

Scatter and Gather

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *
    recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm
    comm);
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *
    recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm
    comm);
```

- **Scatter:** root process divides data among itself and others
- **Gather:** root process collects data

Single-accumulation (Gather with Reduction Op)

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op, int root, MPI_Comm comm);
```

- Each processor provides a block of data with the same type and size
  - reduction (binary associated and communitative) operation applied to element by element to data blocks, results placed in root processor

Multi-accumulation

```
int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, const int recvcounts
    [], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

- Each processor provides for every other processor a potentially *different* data block → Data blocks for same receiver comibined and given reduction operation no root processor

Total Exchange

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
    *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);
```

- Each processor provides for every other processor a potentially *different* data block → Effectively each processor executes a scatter operation, no root processor

Duality of Communication Operations

- A communication operation can be represented by a graph
  - Spanning tree where nodes are processes and directed edges are where communication occurs
- 2 communication operations are **duality** if same spanning tree can be used for both operations with the edge directions reversed
  - **Single-broadcast operation:** top-down traversal
  - **Single-accumulation operation:** bottom-up traversal

Stepwise Specialization

Communication operations can be ordered into a hierarchy:

- From the most general to the most specific: where specific operations can be implemented in terms of more generation operations
- Operations that are resulted from stepwise specialization are placed near to each other

11. Interconnection Networks

Motivation Examples: Sorting

- **Sorting on Linear Array:** Odd-Even Transposition sort n rounds given n processors ⇒ if 1 round done in 1 step (in parallel), $O(n)$ complexity
- **2-D Mesh**: PEs have 2 links at corners, and 3 links on the edges, wrap left to right, top to bottom ⇒ *Torus*. Sorting 2D Mesh:
  1. Each PE arranged in 2D Mesh with 1 number, $\sqrt{N}$ rows and columns, sort into 'snake like order'
  2. *Phase 1 Row Sorting:* Odd rows in ascending, Even rows in descending
  3. *Phase 2 Column Sorting:* All columns in ascending order (top to bottom)
  4. *Repeat until sorted*
  5. Complexity:
     - Use parallel odd-even transposition sorts for each row/column sorts, each needs $\sqrt{N}$ steps ⇒ For N numbers, $\log_2 N + 1$ phases, thus parallel shear sort $= \sqrt{N} \times (\log_2 N + 1)$ steps

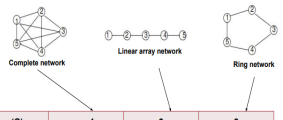
Interconect Topology

Direct Interconection

Static or **Point-to-Point**, often endpoints are of the same type (cores/memory)

- Topology modeled as a **graph** `G = (V,E)`, `V` = vertices, `E` = edges
- `V` are *processors*, `E` are *connections* e.g. network cable
- **Metrics:**
  1. **Diameter** $\delta(G)$**:** *maximum distance* between any pair of node
     - $\delta(G) = \max \min\{k \mid k$is the length of path ϕ from u to v$\}$, $u, v \in V$ & ϕ path from u to v
     - small diameters ⇒ small worst-case distance for message transmission
     - **Lower latency** delay for given message to travel from A to B
  2. **Degree** $g(v)$**:** *number of direct neighbour nodes* of node v
     - **Degree** $g(G)$**:** *maximum degree* of a node in graph G
     - $g(G) = \max\{g(v) \mid g(v)$ degree of $v \in V\}$
     - Small node degrees ⇒ fewer links between nodes, cheaper!
  3. **Bisection width** $B(G)$**:** *minimum number of edges* that have to be removed to divide network in **two equal halves**
     - **Bisection bandwidth** $BW(G)$**:** total *bandwidth* available between 2 bisected portions of network U_1, U_2 partition of V, $|U_1| - |U_2| \leq 1$:
     - $B(G) = \min|\{(u, v) \in E \mid u \in U_1, v \in U_2\}|$
     - Measure of network's capacity in worst case
     - Such a bisection has *minimum bandwidth* between 2 positions, therefore when many nodes transmitting simultaneously across bisection, this is the expected bandwidth/bottleneck of network
  4. **Connectivity:**
     - **Node** $nc(G)$**:** *min* no. of nodes when fail, disconnects network
       - $nc(G) = \min\{|M| \mid$ there exists $u, v \in V \setminus M$ s.t. there exists no path in $G_{V \setminus M}$ from u to $v\}$, where $M \subset V$
       - Determine the robustness of the network
     - **Edge** $ec(G)$**:** *min* no. of edges when fail, disconnects network
       - $ec(G) = \min\{|F| \mid$ there exists $u, v \in V \setminus M$ s.t. there exists no path in $G_{E \setminus F}$ from u to $v\}$, where $F \subset E$
       - Determine number of independent paths between any pair of nodes

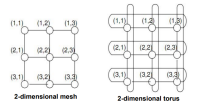
Complete, Linear Array & Ring



Degree g(G)	n - 1	2	2
Diameter δ(G)	1	n - 1	floor(n/2)
Edge Connectivity ec(G)	n - 1	1	2
Bisection Width B(G)	(n / 2)²	1	2

2-Dimensional Mesh and Torus

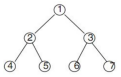
$n = r^2$



Degree g(G)	4	4
Diameter δ (G)	4 2 * (sqrt(N) - 1)	2 2 * (floor(sqrt(N) / 2))
Edge Connectivity ec(G)	2	4
Bisection Width B(G)	4 $n^{1/2}$ for square meshes	8 $2n^{1/2}$ for eq. mesh

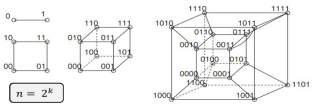
- Generalize to *d*-dimension
  - Number of node $n = r^d$

Complete Binary Tree



Degree g(G)	3
Diameter δ (G)	4 2 log((n+1)/2)
Edge Connectivity ec(G)	1
Bisection Width B(G)	1

Hypercube

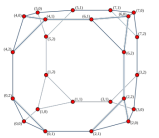


$n = 2^k$

Degree g(G)	$\log_2 n$
Diameter δ (G)	$\log_2 n$
Edge Connectivity ec(G)	$\log_2 n$
Bisection Width B(G)	n/2

Cube-Connected-Cycles (CCC)

- From k-dimensional hypercube ($k \geq 3$), each of the $k-$nodes take one of the original k links $\Rightarrow$ total nodes $= k2^k$
- Each node labelled as (X, Y)
  - $X =$ corresponding node index in hypercube, $Y =$ position in cycle
- Node (X, Y) is connected to
  - $(X, (Y+1) \mod K)$ AND $(X, (Y-1) \mod K)$
  - $(X \oplus 2^y, Y) \rightarrow$ link for corresponding dim in original hypercube



Degree g(G)	3 (always 3, vs hypercube k)
Diameter δ (G)	2k - 1 + floor(k / 2)
Edge Connectivity ec(G)	3
Bisection Width B(G)	n / (2k)

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

Indirect Interconnects

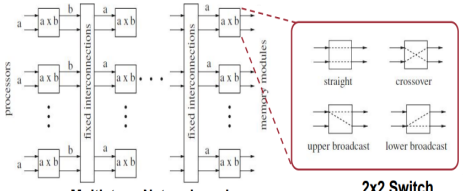
- Why?
  - Reduce hardware costs by sharing switches and links
  - Sometimes (but not necessarily) connects two different types of hardware, e.g., processors and memory modules
- How?
  - Switches provide indirect connections between nodes and can be configured dynamically
- What metrics?
  - Cost (number of switches / links) AND Concurrent connections

Bus Network

- set of wires to transport data from sender to receiver
- only one pair of devices can communicate at a time
  - bus arbiter used for coordination, typically for small no. of processes

Multistage Switching Network

- several intermediate switches with connecting wires btw neighbouring stages
- **Goal:** obtain a small distance for arbitrary pairs of input and output devices

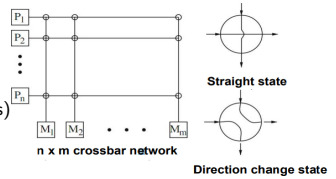


Multistage Networks using Regular 2x2 Crossbar Switches

2x2 Switch Settings

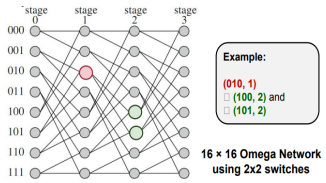
Crossbar Network

- A $n \times m$ crossbar network has n inputs and m outputs
- Two states of a crossbar switch: *straight* or *direction* change
- Hardware is costly ($n \times m$ switches) vs small number of processors



Omega Network

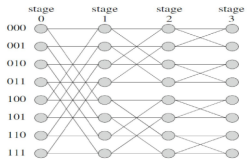
- **One unique path** for every input to output
- $n \times n$ omega network has $\log n$ stages $\Rightarrow \frac{n}{2}$ switches per stage
- connection between stages are regular $(\log n - 1)$ - dimension omega network (connect 16 processors to 16 memory nodes), using 2×2 switches
  - Total no. of switches $= \frac{n}{2}$ switches/stage $\times \log n$ stages $= 32$ switches
  - Crossbar $= 16 \times 16 = 256$ switches
- A specific switch's position: (α, i)
  - α: position of a switch within a stage; i: stage number
- Has an edge from node (α, i) to two nodes $(\beta, i+1)$ where
  - $\beta = \alpha$ with a cyclic left shift
  - $\beta = \alpha$ with a cyclic left shift + inversion of the LSBit



16 × 16 Omega Network using 2x2 switches

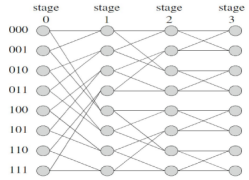
Butterfly Network

- Node (α, i) connects to:
  1. (α, i), i.e., a straight edge.
  2. (α', i), where α & α' differ in the $(i+1)^{\text{th}}$ bit from the left (cross edge)



Baseline Network

- Node (α, i) to **two** nodes $(\beta, i+1)$:
  1. $\beta =$ cyclic right shift of last $(k-i)$ bits of α, where k is the no. of bits
  2. $\beta =$ inversion of the LSBit of α, then cyclic right shift of last $(k-i)$ bits



Routing

- **Based on path length:** Minimal or Non-minimal routing: whether the shortest path is always chosen
- **Based on adaptivity:**
  - *Deterministic:* Always use the same path for the same pair of (source, destination) node
  - *Adaptive:* May take into account of network status and adapt accordingly, e.g. avoid congested path, avoid dead nodes etc

XY Routing for 2D Mesh

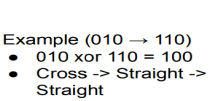
(X_{src}, Y_{src}) to (X_{dst}, Y_{dst}), move in X direction until $X_{src} == X_{dst}$ then move in Y direction until $Y_{src} == Y_{dst}$

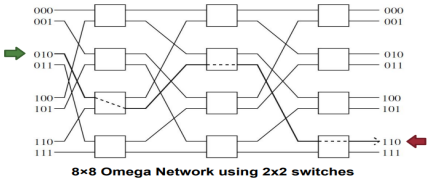
E-Cube Routing for Hyper-cube

- Let $(\alpha_{n-1}, \alpha_{n-2}, \cdot\alpha_0$ and $(\beta_{n-1}, \beta_{n-2}, \cdot\beta_0)$ be the bit representations of source and destination node address respectively:
- no. of bits diff in src/dst node addr $\rightarrow$ number of hops (*hamming distance*)
- Start from MSB to LSB (or LSB to MSB)
  - Find first different bit
  - Go to the neighbouring node with the bit corrected, at most n hops

XOR-Tag Routing for Omega Network

Example (010 → 110)
- 010 xor 110 = 100
- Cross -> Straight -> Straight





8×8 Omega Network using 2x2 switches

- Let T = Source Id $\oplus$ Destination Id
- At stage-k (and assuming MSB is bit 0):
  - Go straight if bit k of T is 0
  - Crossover if bit k of T is 1

12. Energy-efficient Computing

- **Energy:** (unit J Joules): capacity for doing work
- **Power:** (unit W or J/s watts or Joules/sec): amount of energy transferred/converted per unit time
  - Higher power *costs* more (more energy used, more to pay)
  - and increases *heat* as it limits the amount of power a processor can use

Increases in Power Density: End of Dennar Scaling

- **Dennard Scaling:** the theory that processors could always fit more transistors in the same area without using more power (false since 2005)
- more complex / performant processors (↑ transistors) → need more power
- processors improvements are currently **power-limited**

Per-Processor Efficiency

Performance-based efficiency metrics

- Performance-per-watt is a very common efficiency metric $\Rightarrow$ what is performance?
- Running particular workload/benchmark for a *score*
  - **HPL:** solve dense linear system (GFLOPs)
  - **POVRay:** Raytracing scene rendering (unitless value rendertime/FPS)
- *Performance-per-watt:* **Total score ÷ CPU Power**
  - Perf-per-watt vs Clock freq, performance-per-watt is not constant
  - Clock frequency of processor also cause power used to increase ↓ efficiency

Power vs Performance: Diminishing Returns

- *Very very broadly*, clock frequency ↑ can result in score ↑
- However, clock frequency ↑ results in processor power usage ↑↑↑
- Performance often does not increase linearly with power

Processor Power, Frequency, Voltage, Temperature

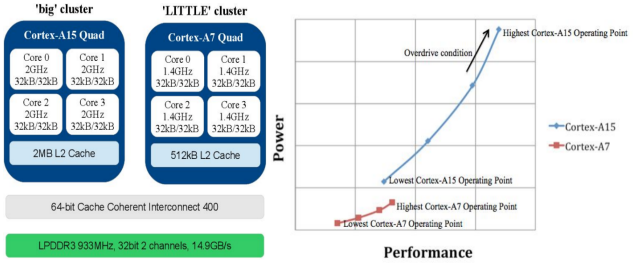
- ↑ clock freq. requires ↑ CPU voltage, or processor will not function correctly
- Processor voltage >>> effect on processor power and therefore temperature
- Factors affecting processor power usage:
  - $P_{total} = P_{dynamic} + P_{static}$
    - P_{total}: total power used
    - $P_{dynamic}$: power used for processor to perform operations
      - $= k \times V^2 \times f \rightarrow V/f$: processor voltage/frequency respecitvely
      - voltage and freq are not independent, ↑ freq require ↑ voltages
      - k: value depending on the **complexity** of the **program** being ran and underlying processor hardware
    - P_{static}: power used by independent of the work done by the processor
  - Every clock frequency value for a processor has a **minimum** safe voltage where the processor will operate correctly

Dynamic Voltage and Frequency Scaling (DVFS)

- Modern processors can adjust their voltage/clock frequency dynamically to ↓ power usage and heat
- Idle CPU / simple background tasks $\Rightarrow$ need less performance (clock frequency) $\Rightarrow$↓ frequency/voltage

Heterogeneous Cores

- Put **two types of procecssors** side-by-side
- e.g. P-cores (more efficient at higher performance levels) and E-cores (efficient at lower power levels) for Intel Meteor Lake



Questions to ask about processors

- Is the benchmark CPU bound? What benchmark program? At what settings?
- What absolute performance was achieved? At what processor wattage?
- How does the performance per watt vary for different loads, frequencies, etc?

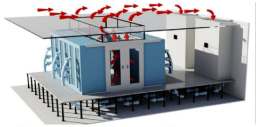
Data Centers/HPC Clusters

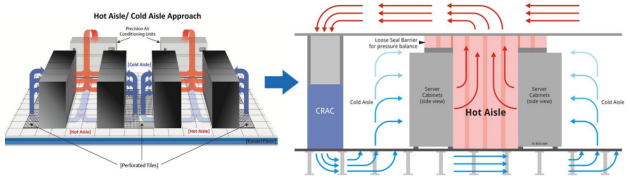
IT infrastructure / compute is often transferred to data centers

- Easy access from any location/machine
- Users don't have to manage their own hardware
- Centralizing hardware $\Rightarrow$ easier to manage power (UPS systems), cooling (centralized cooling solutions), etc
- Fast networking between nodes allows for fast, large distributed jobs
- **Cost of Datacenters / HPC**
  - $>$ 460 TWh per year, $>$ 2% of global electricity usage
  - 0.9% of energy-related greenhouse gas emissions, 0.6% of total

Metrics for Datacenter Efficiency

- **GFLOPs-per-watt:** similar to per-processor efficiency, but now applied across large, multi-node benchmarks
  - *Green500 List:* Tracks the energy efficiency of Top500 supercomputers in GFLOPs per watt $\Rightarrow$ measured using HPL benchmark
  - Power Usage only takes *compute power use* into account
  - *NVIDIA Grace Hopper Superchip* $\rightarrow$ high bandwidth CPU-GPU connection (900GB/s data rate, 5x less power usage for data transfers)
- **Power Usage Effectiveness:** overall data center energy efficiency: energy used for compute vs total energy usage
  - Supporting a datacenter's compute takes energy
  - a metric to determine how much of the total datacenter's energy is used for useful computation vs other overheads (cooling, lighting, pumps...)
  - $PUE = \frac{\text{Total Facility Energy}}{\text{IT Equipment Energy}} = 1 + \frac{\text{Non IT Facility Energy}}{\text{IT Equipment Energy}}$
  - Cyclic trends due to seasons, warmer seasons takes more energy to cool
  - **Issues:**
    - Hotter climates are automatically disadvantaged
    - Due to calculation being ((Overhead + IT) $\div$ IT), perversely, running larger compute loads often decreases PUE! $\Rightarrow$ incentives datacenters to use more absolute energy doing computation
    - Operators are incentivised to measure fewer overheads
  - **Techniques to reduce PUE:**
    - **Hot aisle/cold aisle approach**
      - Cool air enters one side of a server rack
      - Hot air comes out the other side
      - Containment: point hot air sides together, enclose within a "room", vent and cool
      - More efficient cooling!





- SG's National Supercomputing Centre: using **warm water** (40C) to directly cool CPUs and GPUs (45C output)
  - Still possible to efficiently cool the water w/ cooling tower (i4 building)
  - Compute nodes run at 60 - 80C