CS3210 — PARALLEL COMPUTING

DR. CRISTINA CARBUNARU

arsatis[*]

CONTENTS

[*]author: https://github.com/arsatis

1 INTRODUCTION

Lecture 1
9th August 2022

Parallel computing involves the use of multiple processing units to solve a problem quickly/at a greater scale. Processing units could be:

- **Processors** (a.k.a. *CPU, processing element*) with multiple **cores**.
- **Computers** (a.k.a. *nodes, machines*) with multiple processors.
- **Clusters**: a number of nodes connected by a network.

1.1 *Computational model attributes*

- **Operations**: the *primitive units* of computation.
- **Data**: the definition of *address spaces* available to the computation.
- **Control**: the *schedulable units* of computation.
- **Communication**: the modes and patterns of communication between parallel processing elements.
- **Synchronization**: mechanism to ensure needed information arrives at the right time.

1.2 *Basics of parallel computing*

Parallelization involves 3 main steps:

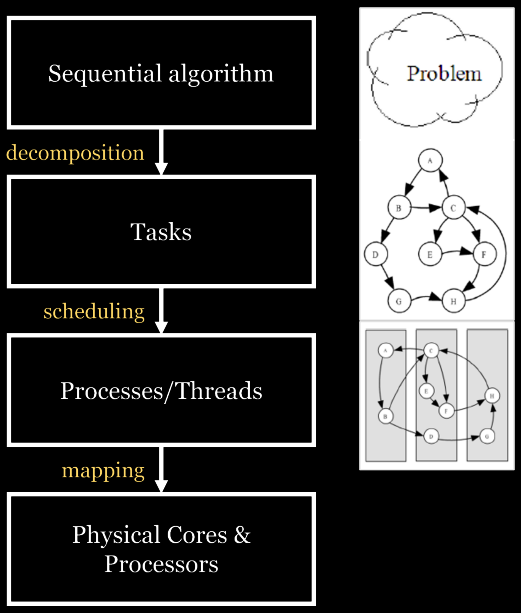


Figure 1: Steps for parallelizing a program.

1. **Decomposition** of the initial algorithm/problem into tasks.
    - Typically, there are many possible decompositions for an algorithm/problem.
    - An decomposition is deemed "optimal" if inter-task dependencies are minimized, based on some desired *granularity* (i.e., size of tasks).
2. **Scheduling**: assignment of tasks to processes/threads.
    - This dictates the execution order of parallel tasks.
    - **Dependencies** among tasks may impose constraints on scheduling, and may require *synchronization* and *coordination* between processes/threads.
    - The latter depends on how information is exchanged between processes/threads, based on the hardware's *memory organization*.
3. **Mapping** of processes/threads to physical processors/cores.
    - This step is typically handled by the operating system.

memory organization: **shared-memory** (threads) or **distributed-memory** (processes).

1.3 *Parallel performance*

Parallel algorithms are typically benchmarked based on their:

1. **Execution time**, which includes *parallelization overheads* such as:
    - Distribution of tasks/work to processors.
    - Initialization of tasks.
    - Synchronization and information exchange.
    - Idle time.
2. **Throughput**.

2 SYNCHRONIZATION

Lecture 2
15th August 2022

2.1 *Processes*

Processes are programs *in execution*. They are identifiable with a process ID, and comprise of:

- executable program (i.e., actual instructions contained by the program),
- global data (e.g., open files, network connections),
- stack/heap, and
- registers (e.g., general purpose and special registers).

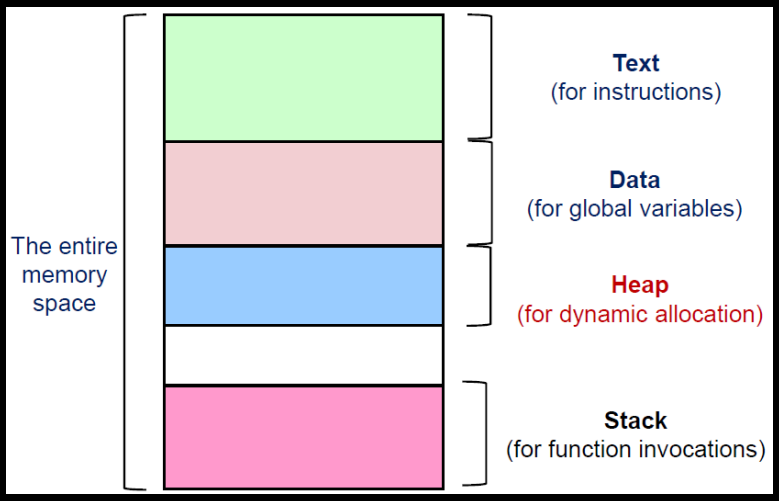


Figure 2: Illustration of a process's memory space.

Processes do not share any address space, thus forking/creating a new process is costly due to the overhead of system calls. Additionally, explicit inter-process communication (IPC) is needed to exchange data between processes.

Some IPC mechanisms include:

- **Shared memory** (locks are typically needed to protect access when reading/writing).
- **Message passing**: comes in various forms, including
  - Blocking vs non-blocking.
  - Synchronous vs asynchronous.
- **(Named) pipes**.
- **Signals**.

2.1.1 Multi-programming

There are two types of multitasking/multi-programming:

- **Time slicing execution**: processes take turns to execute on the same core, but don't actually execute in parallel.
  - **Context switches** are needed to switch between processes, which introduces additional computing overhead (since the states of the suspended process must be saved).
- **Parallel execution** of processes on different cores/resources.

2.1.2 Unix processes

In Unix, process P_1 can create a new process P_2 using either:

- `fork()` system call
- `int exec(char *prog, char *argv[])`

These functions return:

- `0` for the *child process*, and
- `child_pid` for the *parent process*.

P_2 becomes an identical copy of P_1 at the time of the function call, and it:

- works on a copy of the address space of P_1 (with *copy-on-write*, the copy is made only when there is a need to modify either P_1 or P_2).
- continues to execute the same program as P_1, starting with the instruction following the fork call.
- gets its own process ID.

To terminate a child process, we can use `exit(status)`. To wait for the termination of a child process, we can invoke either `wait()` or `waitpid(pid)`.

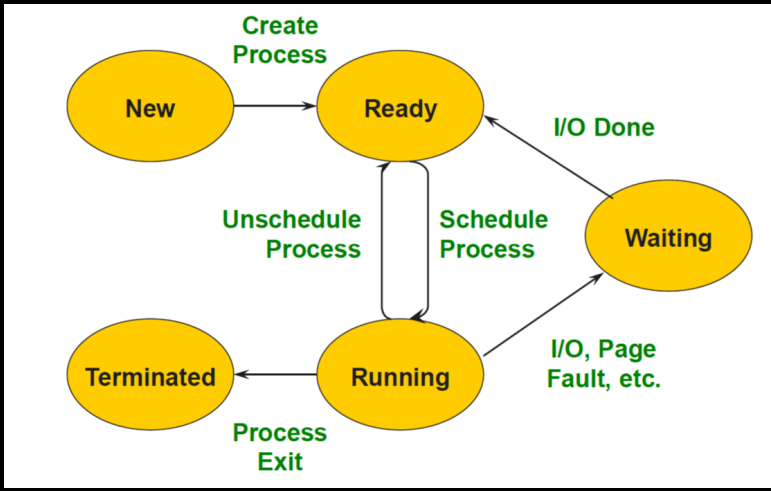


Figure 3: Process state graph illustrating the lifetime of a process.

2.1.3 Process interaction with OS

Exceptions	Interrupts
Caused by *machine-level instructions*	Caused by *external events*, usually hardware related
Synchronous, occur due to program execution	Asynchronous, occur independently of program execution
Executes an *exception handler*	Executes an *interrupt handler*

2.2 *Threads*

Threads define a sequential execution stream within a process; hence the run-time stack (i.e., PC, SP, and registers) is independent across threads. However, threads share the same address space of the process (i.e., the main thread).

Thus, thread generation and context switching is faster than processes.

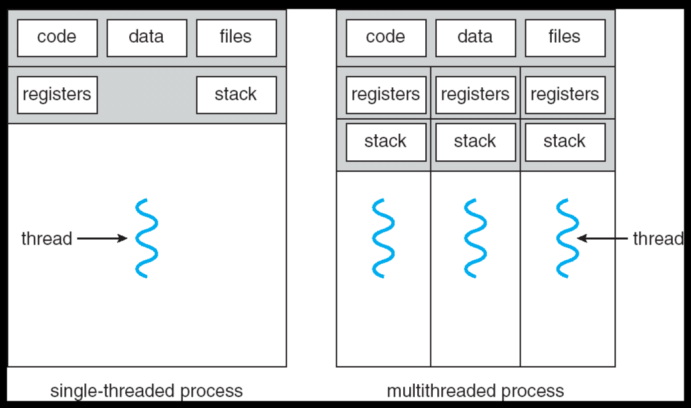


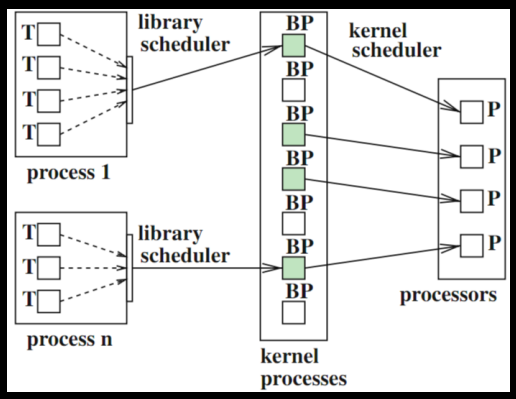
Figure 4: Difference between processes and threads.

There are 2 types of threads:

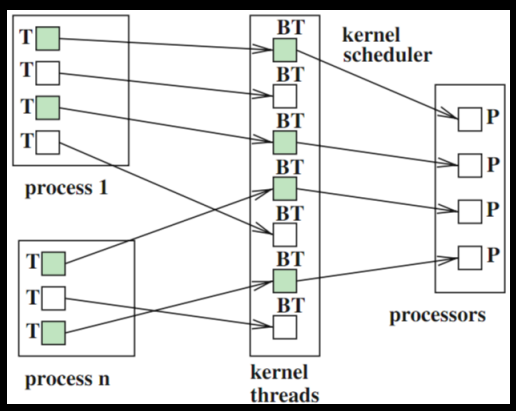
User threads	Kernel threads
OS unaware of their existence	OS aware of their existence
No parallelism, since OS is unable to map different user threads of the same process to different resources	Efficient use of cores in a multi-core system
Scheduling performed at the process level → OS cannot switch to another thread if one thread is blocked (e.g., by I/O)	Blocking of one thread does not block other threads
More configurable and flexible (e.g., thread scheduling policies can be customized)	Generally less flexible

User threads can be mapped to kernel threads in several ways:

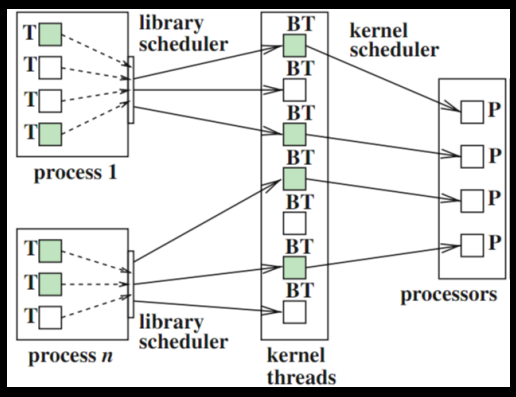
- **Many-to-one**: all user threads are mapped to one process, scheduling of user threads is managed by the thread library.



- **One-to-one**: each user thread is assigned to exactly one kernel thread (e.g., pthread), no library scheduler needed, and OS is responsible for scheduling and mapping of kernel threads.



- **Many-to-many**: library scheduler assigns user threads to a given set of kernel threads, and kernel scheduler maps kernel threads to available execution resources.



2.2.1 POSIX threads

In Unix, threads can be utilized by including the `pthread.h` header.

- Threads are created using `pthread_create(...)`.
- To wait for thread termination, we can invoke `pthread_join(thread, retval)`.

In practice, the number of threads should be suitable to:

- the degree of parallelism of the application, and
- the amount of available execution resources.

2.3 *Synchronization*

Synchronization enables us to restrict the possible interleaving of thread execution, thus ensuring correctness in multi-threaded/process systems.

A **race condition** may arise when:

1. two concurrent threads/processes access a *shared resource* without synchronization, and
2. ≥ 1 thread modifies the shared resource.

To avoid race conditions and erroneous behaviour in parallel systems, we can utilize **mutual exclusion** to synchronize access to shared resources by implementing **critical sections (CS)** with the following properties:

1. **Mutual exclusion**:
   - If one thread is in the CS, then no other is.
2. **Progress**:
   - If some thread T_1 is not in the CS, then T_1 cannot prevent some other thread T_2 from entering the CS.
   - A thread in the CS will eventually leave it.
3. **Bounded wait** (no starvation):
   - If some thread T is waiting on the CS, it will eventually enter it.
4. **Performance**:
   - The overhead of entering and exiting the CS is small w.r.t. the work being done within it.

Critical sections are also expected to satisfy various requirements:

- **Safety property**: nothing bad happens.

- **Liveness property**: a system has to make progress.
- **Performance requirement**.

2.3.1 Mechanisms

- **Locks**:
  - Operations: `acquire(lock)` and `release(lock)`.
  - Locks can spin (i.e., spinlocks) or block (i.e., mutex).
- **Semaphores**: abstract data type that provides mutual exclusion through *atomic counters*.
  - Operations: `wait(S)` and `signal(S)`.
  - Safety property: its value is always ≥ 0.
  - Types: binary/mutex and general/counting.
  - However, they are essentially shared global variables and can potentially be accessed anywhere in the program.
- **Monitors**: allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false.
- **Messages**: simple model of communication and synchronization based on atomic transfer of data across a channel.

2.3.2 Potential problems

- **Deadlock**: exists if every process in a set of processes is waiting for an event that can be caused only by another process in the set.
  - Exists iff the following conditions hold simultaneously:
    1. **Mutual exclusion**: ≥ 1 resource must be held in a non-shareable mode.
    2. **Hold and wait**: there must be one process holding one resource and waiting for another resource.
    3. **No pre-emption**: resources cannot be pre-empted (i.e., critical sections cannot be aborted externally).
    4. **Circular wait**: there must exist a set of processes $P_1, ..., P_n$ s.t. P_1 is waiting for P_2, and so on.
- **Starvation**: situation where a process is prevented from making progress because some other process has the resource it requires.
  - Typically a side effect of the scheduling algorithm.
- **Livelock**: two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work.

2.3.3 Classic synchronization problems

- **Producer-consumer**:
  - Producers create some items and add them to a data structure.
  - Consumers remove the items and process them.
- **Reader-writer**:
  - ≥ 1 readers can be in the critical section simultaneously.
  - Writers must have exclusive access to the critical section.

3 PARALLEL COMPUTING ARCHITECTURES

Lecture 3
22nd August 2022

3.1 *Forms of parallelism*

Concurrency	Parallelism
Tasks execute in overlapping time periods (not necessarily at the same instant)	Tasks execute simultaneously (at the same time)

- **Bit level parallelism**: form of parallel computing based on increasing processor word size.
  - Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word.
- **Instruction level parallelism**: the parallel or simultaneous execution of a sequence of instructions.
  - **Pipelining**: (cf. cs2100)
    1. Split instruction execution into multiple stages (e.g., fetch [IF], decode [ID], execute [EX], write-back [WB]).
    2. Allow multiple instructions to occupy different stages in the same clock cycle.

    (–) **Independence** between instructions is needed for efficiency.
    (–) **Bubbles** (i.e., inefficient pipelines).
    (–) Data/Control flow **hazards**: we did not have the right data at the right time to execute some stage of an instruction.
    (–) **Speculation** methods need to be implemented on hardware (e.g., branch speculation).
  - **Superscalar**: duplicate pipelines so that multiple instructions are allowed to pass through the same stage simultaneously.

    (–) Scheduling of which instructions can be executed simultaneously is challenging (typically done by processor or compiler).

(–) **Structural hazard**: ≥ 2 instructions that are already in pipeline may need the same resource.

- **Thread level parallelism**: allow processors to execute multiple threads in parallel. Variants include:
  - **Fine-grained multithreading**: switch after each instruction.
  - **Coarse-grained multithreading**: switch on *stalls*.
    - **Time-slice multithreading**: switch on predefined time-slices.
    - **Switch-on-event multithreading**: switch if a processor is waiting for an event.
    - **Simultaneous multithreading**: schedule instructions from different threads in the same cycle.

  Specifically, in simultaneous multithreading, processors provide hardware support for multiple thread contexts:

  1. Registers are independent for each thread.
  2. There is sufficient space for multiple thread contexts to execute simultaneously *on the same core*, thus there is no need for context switching (i.e., threads can simply take turns to execute).
- **Processor level parallelism**: allow processors to execute multiple instructions at the same time on different cores by enabling each process/thread to have an independent context.

3.2 *Flynn's parallel architecture taxonomy*

Processors typically require handling 2 streams:

- **Instruction stream**: control/execution flow of a program.
- **Data stream**: contains data to be manipulated by the instruction stream.

Flynn's taxonomy includes the following categories:

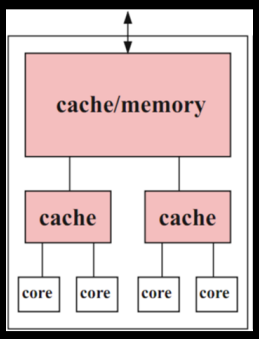
- **Single instruction single data (SISD)**.
- **Single instruction multiple data (SIMD)**: exploits *data parallelism*, e.g., when same operation applied to multiple objects on the same array.
- **Multiple instruction single data (MISD)**: only used for *systolic arrays*.
- **Multiple instruction multiple data (MIMD)**: each processing unit fetches its own instruction and operates on its data.

Modern **stream processors** consist of a mixture of SIMD and MIMD, where scheduling is required to:

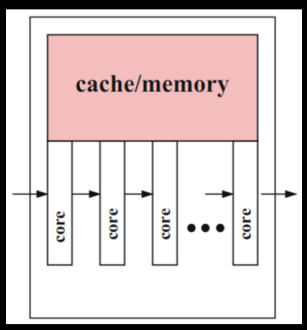
- Separate what is suitable for data parallelism for execution on GPUs.
- Utilize CPUs when the workflow does not require data parallelism.

3.3 *Multi-core processor architecture*

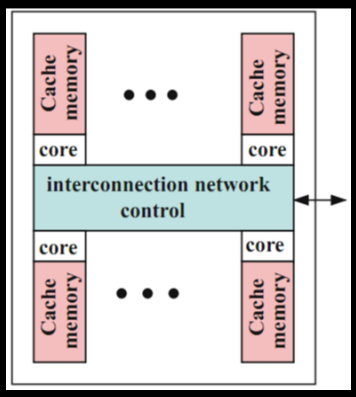
- **Hierarchical design**: each core has a separate L1 cache, and shares the L2 cache with some other cores; all cores share the common external memory (or L3 cache).



- **Pipelined design**: data elements are processed by multiple execution cores in a pipelined manner.
  - Typically utilized in the networking field, especially in routers.



- **Network-based design**: cores and their local caches and memories are connected via an interconnection network.



3.4 *Memory organization*

Memory organization affects program performance; it affects how we access our data, and how fast are we accessing our data.

Key ideas:

memory latency: amount of time for a memory request from a processor to be serviced by the memory system.

memory bandwidth: rate at which the memory system can provide data to a processor.

- Processors run more efficiently when data is stored in caches; as we move further away from the processor, the size of memory increases, but access to this memory space increases as well.
- To utilize processors efficiently, programs must organize computation in a way that minimizes memory access, via e.g.:
  - **Temporal locality optimizations**: reusing data previously loaded by the same thread.
  - **Inter-thread cooperation**: sharing data across threads.

Systems of memory organization in parallel computers:

- **Distributed memory**: each processor/node is an independent unit, *message passing* across the *interconnection network* is required to access data located at another processor.
- **Shared memory**: multiple processors can access the same memory via a *shared memory provider* (typically an interconnection network).
  - (+) No need for partitioning/sharding of data.
  - (+) Efficient communication: no need to physically move data between processors.
  - (−) **Memory contention** between processors becomes an issue as processor count increases → lack of scalability.
  - (−) **Cache coherence** problem:
    - Since multiple copies of the same data will exist on each processor's local cache, updates made by a processor may not be seen by other processors in the same system.
    - Protocols are needed to ensure cache coherence, but these introduce additional overhead.
  - (−) **Memory consistency** problem:
    - Output/Printing of a variable by one processor may differ from the output by another processor, as a result of a third processor having modified the same variable.
    - Protocols with additional overhead are needed to ensure memory consistency.
  - (−) Synchronization of memory access is required.

Shared memory systems can be further classified based on:

1. Processor to memory delay: whether delay to memory is uniform.
   - **Uniform memory access (UMA)**: latency of accessing main memory is the same for all processors.
     * Suitable for small number of processors (due to issues with memory contention).
   - **Non-uniform memory access (NUMA)**: latency of accessing main memory is different across processors; e.g., accessing local memory is faster than remote memory.
2. Presence of a local cache with cache coherence protocol (i.e., cache data will be maintained to be the same as that in memory).
   - **Non-cache coherent non-uniform memory access (nccNUMA)**.
   - **Cache coherent non-uniform memory access (ccNUMA)**: each node has cache memory to reduce memory contention.
   - **Cache-only memory access (COMA)**: each memory block works as cache; if data is not in cache, we will have to re-retrieve it from a separate storage space.

- **Hybrid** (i.e., distributed-shared memory): memory is distributed between machines but shared between cores (i.e., within machines).

4 PARALLEL PROGRAMMING MODELS

Lecture 4
29th August 2022

Parallelism describes the average units of work that can be performed in parallel per unit time. Some bottlenecks to parallelism include:

- Program dependencies:
  - **Data dependency**: situation in which an instruction refers to the data of a preceding instruction.
  - **Control dependency**: situation in which an instruction executes only if the previous instruction evaluates in a particular way (e.g., branch conditions).
- Runtime issues:
  - **Memory contention**: situation in which two different programs, or two parts of a program, try to read items in the same block of memory at the same time.
  - **Communication overhead**: e.g., communicating over a network.
  - **Thread/Process overhead**.
  - **Synchronization**.

Models for parallel processing differ in their level of abstraction.

1. **Machine model**: lowest level of abstraction, consists of a description of hardware and the operating system.
2. **Architectural model**: describes interconnection network, memory organization, sync/async processing, and execution mode (e.g., SIMD/MIMD).
3. **Computational model**: provides an analytical method for designing and evaluating algorithms.
4. **Programming model**: describes a parallel computing system in terms of the semantics of the programming language/environment.

4.1 *Levels of parallelism*

- **Data parallelism**: partitioning the data used in solving the problem among processing units (e.g., SIMD instructions).
  - Each processing unit carries out *similar operations* on its partition.
  - **Loop parallelism**: when algorithms perform computations by iteratively traversing a large data structure.
- **Task parallelism**: decomposing tasks employed in solving the problem into multiple tasks, and distributing them among processing units.
  - **Task dependence graphs** can be used to visualize and evaluate the task decomposition strategy, where:
    - Nodes rep. tasks (node values rep. expected execution time).
    - Edges rep. control dependencies between tasks.
    - **Critical path** length: maximum completion time for all tasks.
    - **Degree of concurrency**: indication of amount of work that can be done concurrently.

Frameworks such as **OpenMP** can be used for loop parallelism.

tasks: single statements, series of statements, loops, or function calls.

$$\deg = \frac{\text{total work}}{\text{critical path length}} = \frac{\sum_i v_i}{\sum_{i \in \text{critical path}} v_i}$$

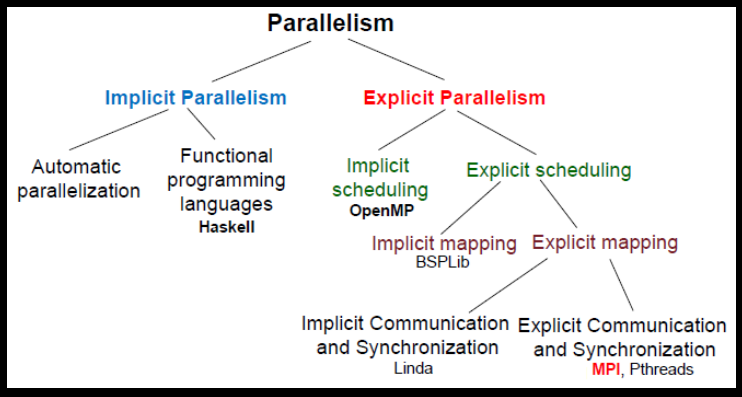


Figure 5: Explicit and implicit representation of parallelism.

In functional languages, each function can be treated as a task. Functions which are independent of each other can be automatically parallelized by the compiler.

However, implicit parallelization is challenging because it is difficult to:
(1) accurately determine the time taken to execute each task/function at compile time,
(2) perform dependence analysis for pointer-based computations, and
(3) extract parallelism at the "right" level of recursion.

4.2 *Coordination models*

- **Shared address spaces**: tasks communicate by reading from/writing to shared variables.
  - Locks are used to ensure mutual exclusion.
  - Processors can load/store at any address.
  - Shared address spaces can be utilized in both shared (most ideal) and distributed memory systems.
- **Message passing**: tasks operate within their private address spaces, and communicate by explicitly sending/receiving messages.
- **Data parallel models**: same function mapped onto large collection of data, no communication between distinct tasks.

Possible issues with shared address spaces when data cannot fit into memory:
1. Disk thrashing due to multiple page faults, since memory is unable to hold all the data.
2. High cache miss rate, especially if we are traversing column-by-column on a 2D matrix, since the elements on each column are unlikely to lie on the same cache line.

Each coordination model can be implemented (with slight differences) on different multi-core architectures/memory organization.

4.3 *Program parallelization*

Computation may come in different granularities:

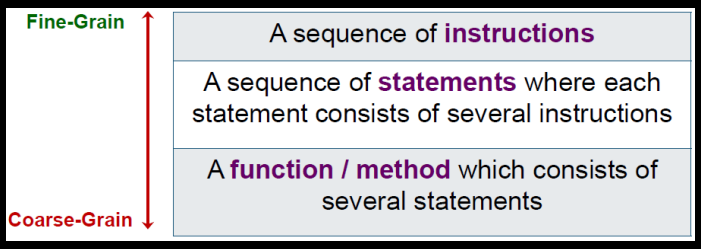
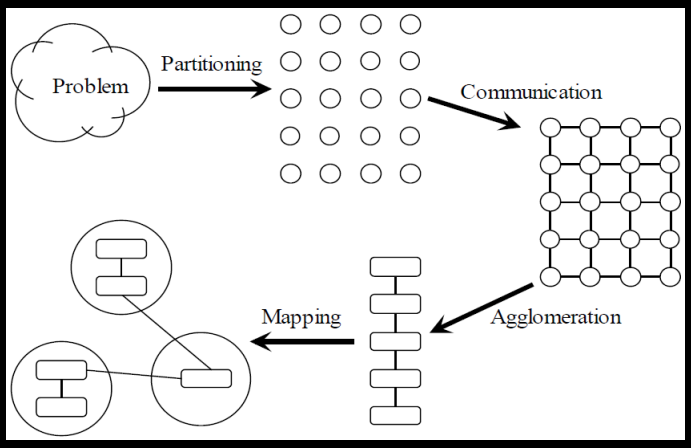


Figure 6: Different granularities of computation.

Foster's design methodology:



1. **Partitioning**: divide data and/or computation into smaller, independent pieces to maximize parallelism.
   - **Domain decomposition**: data centric/data parallelism.
   - **Functional decomposition**: computation centric/task parallelism.

   Rules of thumb:
   - At least 10x more primitive tasks than cores in target computer.
   - Minimize redundant computations and redundant data storage.
   - Primitive tasks should be of similar size.
   - Number of tasks should increase with problem size.
2. **Communication**: determine how data is passed among tasks.
   - **Local communication**: tasks require data from small number of other tasks (i.e., neighbours).
   - **Global communication**: tasks require data from most/all of the other tasks in the system (more costly than local communication).
     - E.g., centralised summation algorithm:
       * Does not distribute computation and communication (i.e., centralized).
       * Does not allow overlap of computation and communication operations (i.e., sequential).

   Rules of thumb:
   - Communication operations should be balanced among tasks.
   - Tasks should employ local communication as far as possible.
   - Tasks should perform communication in parallel.
   - Computation should be overlapped with communication.
3. **Agglomeration**: combine tasks into larger tasks.

   Goals:
   - Decrease communication costs.
   - Decrease development costs (i.e., simplify programming).
   - Maintain program scalability and flexibility.

   Rules of thumb:
   - Locality of parallel algorithm should be increased.
   - Number of tasks should increase with problem size.
   - Number of tasks should be suitable for target systems.
   - Agglomeration can be skipped if the cost of code modification is too high.

4. **Mapping**: assignment of tasks to processors/cores.

   Goals:

   - **Maximize processor utilization**: place tasks on different processing units to increase parallelism.
   - **Minimize inter-processor communication**: place tasks that communicate frequently on same processing units to increase locality.

   Rules of thumb:

   - Utilize heuristics to find some decent mapping (since finding an optimal mapping is generally NP-hard).
   - Consider designs for one task per core and multiple tasks per core.
   - If *static task allocation* is used, ratio of tasks to cores should be at least 10:1; if *dynamic task allocation* is used, the task allocator should not be a bottleneck to performance.

4.4 *Parallel programming patterns*

Lecture 5
5th September 2022

Parallel programming patterns provide a coordination structure for tasks. Some examples include:

- **Fork-join**:
  1. Task T creates a number of child tasks $T_1, ..., T_m$ with a *fork* call, which work in parallel (usually independent of each other) and execute a given part/function of a program.
  2. T waits for the termination of $T_1, ..., T_m$ using a *join* call.

  Impl: language construct or library function (e.g., pthread, OpenMP).
- **Parbegin-parend**:
  1. Programmer specifies a sequence of statements/function calls (usually similar to each other) to be executed in parallel.
  2. When an executing thread reaches a *parbegin-parend construct*, a set of threads is created, and the statements of the construct are assigned to these threads for execution.
  3. Statements following the construct will only be executed after all threads have finished their execution.

  Impl: language construct (e.g., OpenMP) or compiler directives.
- **Master-worker**:
  1. Master thread controls the program execution and executes the main function.
  2. Master thread initializes and assigns work (e.g., computations) to worker threads as needed.

Master task is generally responsible for the initialization, coordination, timings, and output operations.

- **SIMD & SPMD**:
  - *SIMD*: single instructions are executed synchronously by different threads on different data.
    - Implicit synchronization.
  - *SPMD*: same program executed on different cores, and operate on different data.
    - No implicit synchronization, all threads have equal rights and work asynchronously with each other.
    - Explicit synchronization operations are required if necessary.

  Impl: MPI.
- **Client-server**: **MPMD** (multiple program multiple data) model.
  1. Server computes requests from multiple client tasks concurrently.
  2. For each new request, a client is spawned to handle the request.

  Note: client-server interactions happen within the same parallel machine.

  Note: unlike MW, CS works with requests (i.e., work done by clients are clear tasks), and clients are more independent of the server (as compared to workers' dependence on their master), e.g., they can spawn additional threads to help handle requests.
- **Task pool**:
  - Employs a common data structure from which a fixed number of threads retrieve tasks/requests for execution.
  - Program is completed when the task pool is empty, and each thread has terminated the processing of its last task.

  (+) Tasks can be generated dynamically → useful for adaptive apps.

  (+) Thread creation overhead is independent of the number of tasks.

  (−) For fine-grained tasks, the overhead of task retrieval and insertion (i.e., threads can generate new tasks) becomes significant.
- **Producer-consumer**:
  - *Producer threads* produce data, which are used by *consumer threads* as input.
  - Explicit synchronization is needed to ensure correctness.
- **Pipelining**: form of functional parallelism.
  1. Application data is partitioned into a stream of data elements.
  2. Data elements flow through each pipeline task one by one, to perform different processing steps.

5 PERFORMANCE OF PARALLEL SYSTEMS

Depending on the point of view, different criteria are important to evaluate performance.

- **Users**: small response times.
- **Computer centers/managers**: high throughput (i.e., average # work units executed per unit time).

5.1 *Response time*

The response time of a program A can be split into:

- **User CPU time**: time spent on executing A.
- **System CPU time**: time spent on executing OS routines issued by A.
- **Waiting time**: caused by waiting on I/O operations and the execution of other programs because of time sharing.

Generally, the system CPU time depends on the OS implementation, and waiting time depends on the load of the computer system.

On the other hand, user CPU time depends on:

1. Translation of program statements by the compiler into instructions.
2. Execution time for each instruction.

Therefore,

$$t_{\text{user}}(\text{A}) = \text{N}_{\text{cycle}}(\text{A}) \times t_{\text{cycle}} \tag{1}$$

$$= \text{N}_{\text{instr}}(\text{A}) \times \text{CPI}(\text{A}) \times t_{\text{cycle}} \tag{2}$$

since $\text{N}_{\text{cycle}}(\text{A}) = \sum_{\text{N}_{\text{instr}}} n_i(\text{A}) \times \text{CPI}_i$ where

- $t_{\text{user}}(\text{A})$ rep. user CPU time of program A.
- t_{cycle} rep. cycle time of CPU.
- N_{cycle} rep. total # CPU cycles req. for all instructions.
- N_{instr} rep. total # of instructions executed for A.
- CPI rep. the average # of CPU cycles needed per instruction.

If we include memory access time to the user time, we have:

$$\begin{aligned} t_{\text{user}}(\text{A}) &= \Big(\text{N}_{\text{cycle}}(\text{A}) + \text{N}_{\text{mm\_cycle}}(\text{A})\Big) \times t_{\text{cycle}} \\ &= \Big(\text{N}_{\text{instr}}(\text{A}) \times \text{CPI}(\text{A}) + \text{N}_{\text{rw\_op}}(\text{A}) \times \text{R}_{\text{miss}}(\text{A}) \times \text{N}_{\text{miss\_cycles}}\Big) \times t_{\text{cycle}} \end{aligned} \tag{3}$$

where

- $\text{N}_{\text{mm\_cycle}}(\text{A})$ rep. # additional clock cycles due to memory accesses.
- $\text{N}_{\text{rw\_op}}(\text{A})$ rep. total # of read and write operations.
- $\text{R}_{\text{miss}}(\text{A})$ rep. read and write miss rate.
- $\text{N}_{\text{miss\_cycles}}$ rep. # additional cycles needed for loading a new cache line.

and

$$t_{\text{rw\_access}}(\text{A}) = t^{\text{L1}}_{\text{rw\_hit}} + \text{R}^{\text{L1}}_{\text{rw\_miss}}(\text{A}) \times t^{\text{L1}}_{\text{rw\_miss}} \tag{4}$$

$$t^{\text{L}(x-1)}_{\text{rw\_miss}}(\text{A}) = t^{\text{L}x}_{\text{rw\_hit}} + \text{R}^{\text{L}x}_{\text{rw\_miss}}(\text{A}) \times t^{\text{L}x}_{\text{rw\_miss}}$$

$$\text{Global miss rate} = \prod_{x} \text{R}^{\text{L}x}_{\text{rw\_miss}}(\text{A}) \tag{5}$$

where

- $t_{\text{rw\_access}}(\text{A})$ rep. average read/write access time of A.
- $t_{\text{rw\_hit}}$ rep. time for read/write access to cache, irrespective of hit or miss.
- $\text{R}_{\text{rw\_miss}}(\text{A})$ rep. cache read/write miss rate of A.
- $\text{T}_{\text{rw\_miss}}$ rep. read/write miss penalty time.

Note that when doing performance analysis, we assume read memory access to be just as fast as write memory access.

In reality, read access is much faster than write access, since writing typically requires an overhead associated with running the cache coherence protocol.

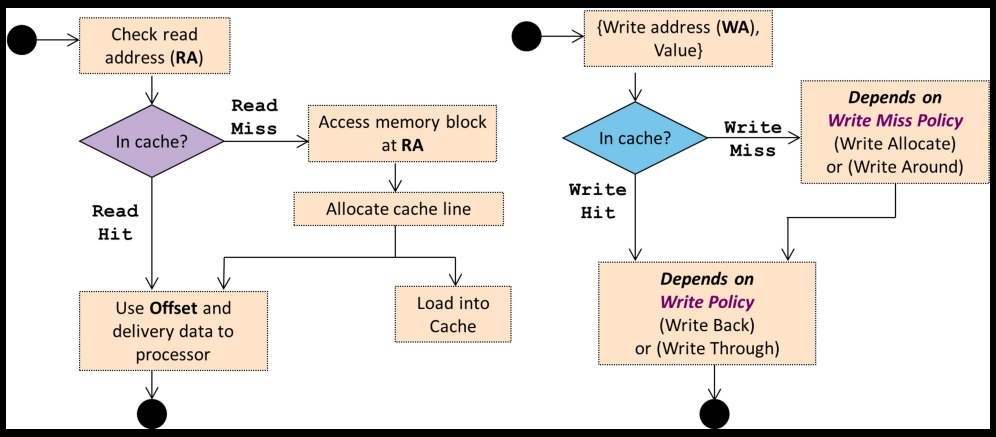


Figure 7: Read and write memory access workflows.

5.2 *Throughput*

Throughput is typically measured using either:

- **MIPS** (million-instructions-per-second):

$$\mathrm{MIPS(A)} = \frac{\mathrm{N_{instr}(A)}}{t_{\mathrm{user}}(\mathrm{A}) \times 10^6} = \frac{\mathrm{clock\_frequency}}{t_{\mathrm{user}}(\mathrm{A}) \times 10^6} \tag{6}$$

  (–) Only considers the number of instructions.

- **MFLOPS** (million-floating-point operations-per-second):

$$\mathrm{MFLOPS(A)} = \frac{\mathrm{N_{fl\_ops}(A)}}{t_{\mathrm{user}}(\mathrm{A}) \times 10^6} \tag{7}$$

  (–) No differentiation between different types of floating-point operations.

5.3 *Efficiency of parallel programs*

Key concepts when analyzing parallel programs include:

- **Execution time**: $\mathrm{T}_p(n)$ rep. execution time for a problem of size n, on p processing units. This includes:
  - Time for executing local computations.
  - Time for exchange of data between processors.
  - Time for synchronization between processors.
  - Waiting time (e.g., due to unequal load distribution, or accessing a shared data structure).

average concurrency: total time taken to execute all tasks / time taken along critical path = speedup.

maximum concurrency: maximum number of tasks running concurrently at any point in time.

- **Cost**:

$$\mathrm{C}_p(n) = p \times \mathrm{T}_p(n) \tag{8}$$

  - $\mathrm{C}_p(n)$ measures the total amount of work done by all processors.
  - Parallel programs are *cost-optimal* if they execute the same total number of operations as the *fastest* sequential program.

- **Speedup**:

$$\mathrm{S}_p(n) = \frac{\mathrm{T_{best\_seq}}(n)}{\mathrm{T}_p(n)} \tag{9}$$

  - $\mathrm{S}_p(n)$ measures the benefit of parallelism.
  - Theoretically, $\mathrm{S}_p(n) \leq p$ (< due to parallelization overheads).
  - In practice, $\mathrm{S}_p(n) > p$ can occur, e.g., due to caching.

i.e., *superlinear speedup*.

- **Efficiency**:

$$\mathrm{E}_p(n) = \frac{\mathrm{S}_p(n)}{p} = \frac{\mathrm{T}_*(n)}{p \times \mathrm{T}_p(n)} = \frac{\mathrm{T}_*(n)}{\mathrm{C}_p(n)} \tag{10}$$

  - $\mathrm{T}_*(n)$ is an approximation to the time taken for the (theoretical) best sequential algorithm.
  - $\mathrm{E}_p(n)$ measures the degree of speedup achieved.
  - $0 \leq \mathrm{E}_p(n) \leq 1$.

5.4 *Scalability*

Scalability refers to the ability to maintain execution time despite increasing problem size. Typically,

- For small problem sizes, parallelism overheads dominate parallelism benefits + inappropriate for large machines due to high task distribution overhead.
- For large problem sizes, small machines may not be able to withhold the working instruction set in memory.

There are several constraints associated with scaling:

- **Application-oriented**: parallelism is limited by how many small tasks we could obtain from an algorithm/application.
- **Resource-oriented**: parallelism is limited by the amount of resources.
  - **Problem constrained scaling**: scaling within the same problem.
  - **Time constrained scaling**: scaling within a fixed amount of execution time.
  - **Memory constrained scaling**: scaling within a fixed amount of memory.

5.5 *Performance analysis*

- **Amdahl's law**: the speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelized (i.e., the *sequential fraction* $0 \leq f \leq 1$).
  - Amdahl's law, $\mathrm{S}_p(n) = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$, holds only for *fixed problem sizes*.
  - For most computing problems, f is dependent on the problem size n, with $\lim_{n\to\infty} f(n) = 0$. Thus, $\lim_{n\to\infty} \mathrm{S}_p(n) = \frac{p}{1+(p-1)f(n)} = p$.
- **Gustafson's law**: if the sequential fraction f decreases with problem size, then $\mathrm{S}_p(n) = \frac{f(n)+p(1-f(n))}{f(n)+(1-f(n))} = p - (p-1)f(n) \leq p$.

6 GPGPU

Lecture 6
12th September 2022

Each GPU contains multiple **streaming multiprocessors (SMs)**, which connect with CPUs and the rest of the computer using a *connecting interface*. Each SM consists of multiple cores, and contains:

- Memories (i.e., registers, L1 cache, and shared memory).
- Logic for thread and instruction management.

6.1 *CUDA*

6.1.1 Programming model

CUDA is a general purpose programming model which acts as an extension to the C language. CUDA supports a fully concurrent read-concurrent write memory model (i.e., all threads can read/write from memory at the same time), and is used to launch batches of threads on the GPU.

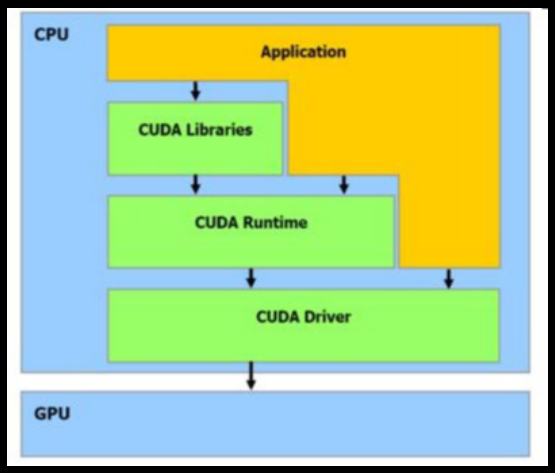
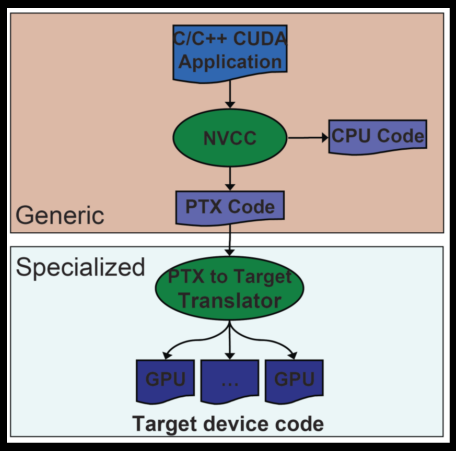


Figure 8: CUDA layers.

CUDA comes with:

- **Programming model**: which transparently scales to many cores and parallel threads, enabling heterogeneous systems (i.e., CPU + GPU) and allowing programmers to focus on parallelizing algorithms.
- **Programming interfaces**:
  - CUDA C runtime: minimal set of extensions to C; kernels are defined as C functions embedded in the application source code, and they require a runtime API which is built on top of the CUDA driver API.

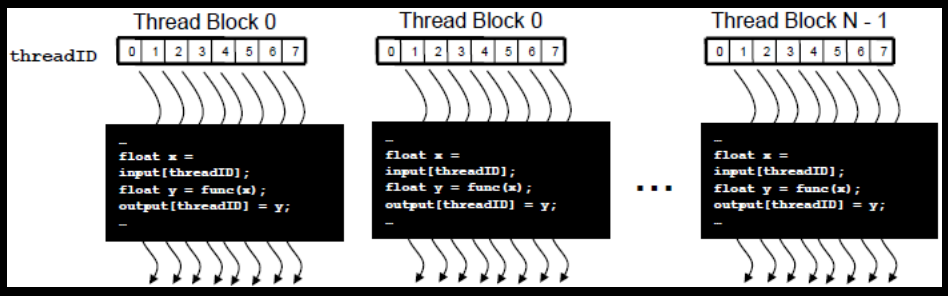
  - CUDA driver API: communicates with the GPU, loads compiled kernels, inspects their parameters, and launches them.
- **Software development environment**: tools (compiler [NVCC], debugger, profiler), C runtime, and libraries.
  - NVCC outputs host CPU code and PTX code, which is interpreted at runtime and runs on the GPU.



  - Linking is done with two static/dynamic libraries:
    * CUDA runtime library (cudart).
    * CUDA core library (cuda).

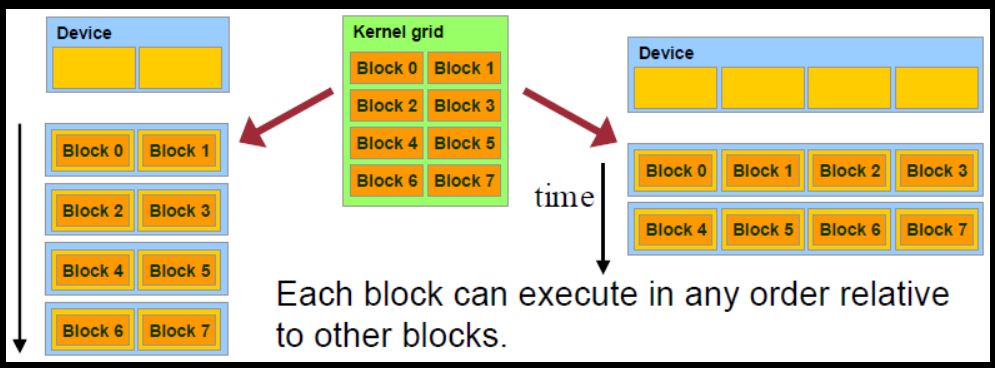
CUDA execution model:

- **Host**: CPU.
- **Device**: GPU.
- **Kernel**: *function* that runs on the device.
  - Each CUDA kernel is executed by an array of threads.
    * Each thread has an ID which can be used to compute memory addresses and make control decisions.
    * CUDA threads are extremely lightweight, and their overhead of creation and context switching is very small.
    * Threads in the same array may share results and memory accesses to save computation and reduce bandwidth.



  - Each thread array can be divided into multiple blocks.
    * Ideally, each block should have a thread count that is a multiple of 32.
    * Within the same block, threads can cooperate (e.g., via *shared memory*, *atomic operations*, and *barrier synchronization*).
    * Threads in different blocks are unable to cooperate, since they may be executed on different SMs.
    * Several blocks execute concurrently on one SM, subject to control and resource limitations (e.g., registers and shared memory, which are partitioned among all resident threads).

Note: the hardware is free to schedule thread blocks to any processor at any time, thus a kernel may scale across any number of SMs.



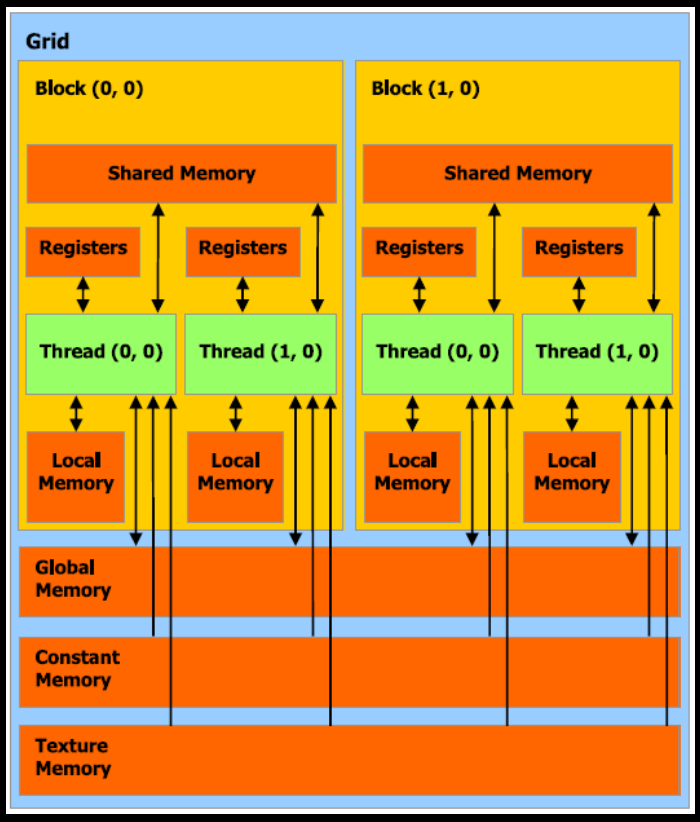
  - Kernels are launched in grids of thread blocks.

Mapping thread execution to architecture:

- CUDA employs a SIMT (single instruction multiple thread) execution model.
- SMs create, manage, schedule, and execute threads in SIMT **warps** (i.e., groups of 32 parallel threads).
  - Threads in a warp start together at the same program address, and execute instructions in lockstep.
  - Threads have individual instruction program counter and register state.
  - Each warp contains threads of consecutive, increasing thread IDs.

CUDA memory model:

Type	Scope	Access type	Speed	Explicit sync
Register	thread	RW	fastest	no
Local	thread	RW	slower than shm	no
Shared	block	RW	fast	yes
Global	program	RW	slow	yes
Constant	program	R	slow	yes
Texture	program	R	slow	yes



- **Global memory** is cached, but **shared memory** is not cached (because it is the cache).
- **Shared memory** has higher bandwidth and lower latency than local and global memory, and is divided into 32 **banks**.
  - Addresses from different banks can be accessed simultaneously.
  - **Bank conflict**: if ≥ 2 threads access different locations in the same bank at the same time $\rightarrow$ memory access has to be serialized.
    - There is no bank conflict if multiple threads access the *same location* in the same bank; broadcasting can be used to push the information out to all threads within a single memory txn.
  - **Strided accesses**: threads in a warp access memory in strides of 2+.

banks are equal-sized memory modules.

    - This results in a significantly reduced load/store efficiency (cf. figure above), since half of the elements in a txn are not used.
- Local, constant, and texture memory are cached and used for convenience/performance.
  - **Local**: automatic array variables allocated by compiler.
  - **Constant**: useful for uniformly-access read-only data.
  - **Texture**: useful for spatially coherent random-access read-only data; provides filtering, address clamping, and wrapping.
- **Memory coalescing**: simultaneous accesses to global memory by threads in the same warp are **coalesced** into a number of txns equal to the number of 32-byte txns necessary to service all the threads of the warp.

6.1.2 CUDA optimization strategies

- **Optimizing memory** to achieve maximum memory bandwidth:
  1. Minimize data transfer between host and device: so that host can engage in other computations.
     - Peak bandwidth between device memory and GPU is much higher than that between host and device memory.
     - Use page-locked/pinned memory transfer.
     - Use the zero-copy feature which allows threads to directly access host memory.
  2. Ensure global memory accesses are coalesced whenever possible.
  3. Minimize global memory accesses by using shared memory.
  4. Minimize bank conflicts in shared memory accesses.
- **Restructuring algorithms** to maximize parallel execution:
  - Overlap asynchronous transfers with computation: e.g., `cudaMemcpyAsync()`.
  - Improve occupancy of SMs: number of warps should be larger than the number of SMs, so that each SM has ≥ 1 warp to execute.
    - No idling when a block synchronizes.
    - Low occupancy $\rightarrow$ significant memory latency.
  - Threads per block should be a multiple of warp size (minimum 64).
  - Avoid multiple contexts per GPU within the same CUDA application.
- **Maximizing instruction throughput**:
  - Minimize use of arithmetic instructions with low throughput (e.g., replace division and modulo operations with bitwise operations).
  - Use signed (instead of unsigned) loop counters.
  - Use single-precision over double-precision floats.
  - Minimize divergent warps caused by diverging control flows.
  - Reduce the number of instructions.

7 COHERENCE AND CONSISTENCY

Lecture 7
26th September 2022

In a **shared address space** model, tasks communicate by reading/writing to shared variables, and mutual exclusion is ensured with the use of locks.

In this model, processors are allowed to load and store from any address, which translates to significant contention that does not scale well with the number of processors.

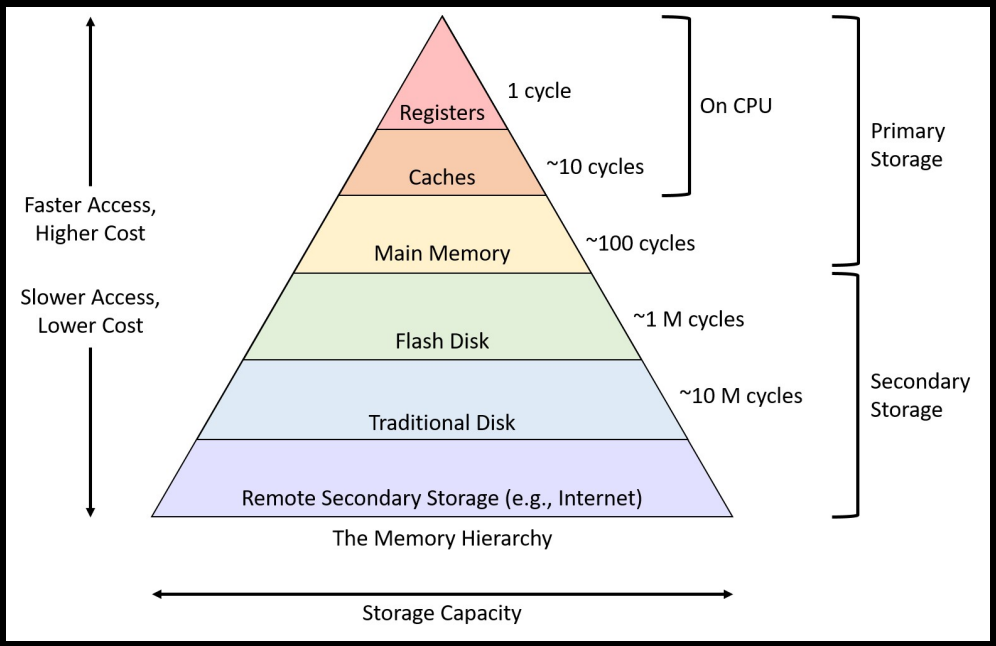


Figure 9: Memory hierarchy.

Modern processors run most efficiently when data is resident in caches, which reduce memory access latency and provide high bandwidth data transfer to CPU. Typically, cache coherence protocols are run in the background to ensure cache coherence.

7.1 *Cache coherence*

Cache properties:

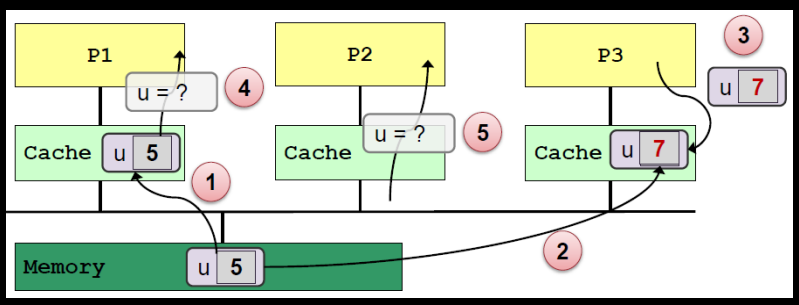
- **Cache size**: larger cache results in
  - ↑ *access time* (due to increased addressing complexity).
  - ↓ *cache misses*.
- **Block size**: data is transferred between main memory and cache in blocks of fixed length (i.e., cache lines). Larger blocks lead to
  - ↓ number of blocks transferred.
  - ↑ time of data transfer.
  - ↑ chance of cache hit.

Typically, L1 cache blocks have a size of 4-8 memory words.

Write policies:

- **Write-through**: writes are immediately transferred to main memory.
  - (+) Processors always get the newest value of a memory block.
  - (–) Slowdown due to multiple memory accesses (can be reduced by using a write buffer, i.e., buffer together several writes and write the entire buffer into memory at the same time).
- **Write-back**: writes are only performed in cache, and writes to main memory are only performed when cache blocks are replaced (i.e., using a dirty bit to indicate whether each cache line has been modified since it was brought in from memory).
  - (+) Less memory accesses.
  - (–) Memory may contain invalid entries.

Cache coherence problem: local updates by a processor are not observed by other processors (i.e., they still observe the data before the update).



- This occurs because there is a global storage space (i.e., main memory) and per-processor local storage (i.e., local caches).

- Intuitively, reading from an address should yield the last value written at that address by any processor.

Coherence ensures that each processor has a consistent view of memory through its local cache. Properties of a coherent memory system include:

1. **Program order**: given the sequence
   (a) P writes to X.
   (b) No further write to X.
   (c) P reads from X.

   P should get the value written in (a).
2. **Write propagation**: writes will *eventually* become visible to other processors, i.e., given the sequence
   (a) P_1 writes to X.
   (b) No further write to X.
   (c) P_2 writes to X.

   P_2 should read the value written by P_1.
3. **Write serialization**: all writes to a location are seen in the same order by all processors, i.e., given the sequence
   (a) $X \leftarrow V_1$ (by any processor).
   (b) $X \leftarrow V_2$ (by any processor).

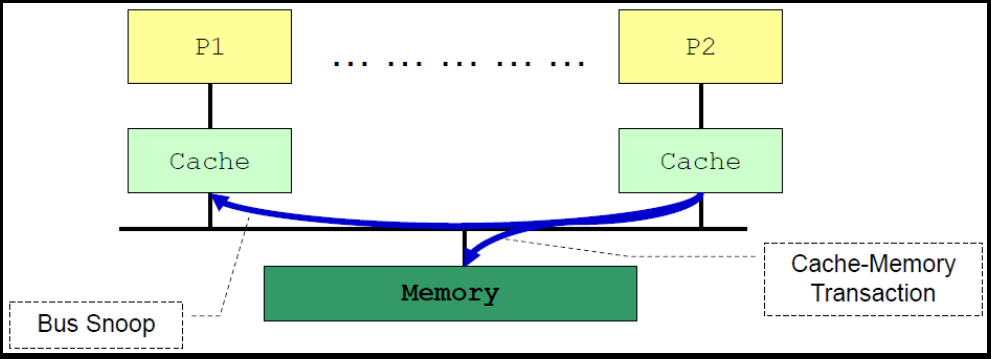
   Processors can never read X as V_2, and then later as V_1.

Cache coherence can be maintained by:

- Software-based solution: using OS's page fault to propagate writes.
- Hardware-based solution: *cache coherence protocols*, e.g.,:
  - **Snooping based**: each cache monitors/snoops on the *bus* and updates the status of each cache line (i.e., whether it has been modified) when necessary.

higher overhead, rarely used nowadays.

bus: medium used for communication between cores and memory.



  - **Directory based**: sharing status is kept in a centralized location.

more common in NUMA architectures.

Cache coherence implications:

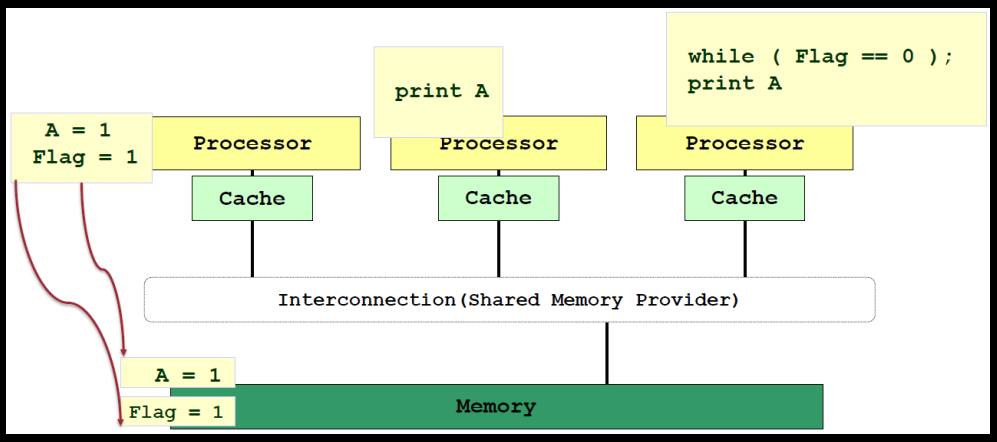
- **Cache ping-pong**: multiple processors read and modify the same global variable/cache line → cache invalidation for other processors.
- **False sharing**: when 2 processors write to different addresses which map to the same cache line; one processor writing to the cache line will result in cache misses for subsequent reads on the same cache line by the other processor.
- Overhead in shared address space: increased memory latency and lower cache hit rate (due to cache ping-pong and false sharing).

7.2 *Memory consistency*

consistency: constraints the order in which memory operations performed by one thread become visible to other threads for different memory locations.

Coherence guarantees that writes to address X will eventually propagate to other processors. On the other hand, **consistency** deals with when writes to X propagate to other processors, relative to reads and writes to other addresses.

Memory consistency problem: two distinct memory operations might be observed in a different order by other processors.

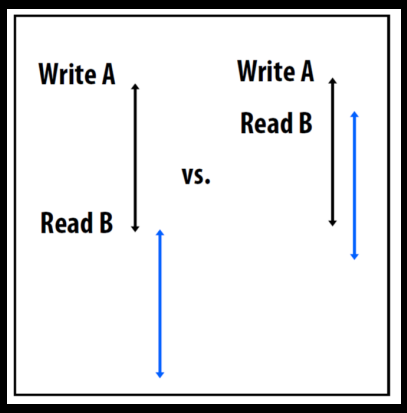


Typically, programs define a sequence of loads/reads (R) and stores/writes (W). Given these two memory operations, there are 4 possible orderings: W → R, R → R, R → W, and W → W.

- **Sequential consistency models**: every processor issues its memory operations in program order.
- **Relaxed consistency** models: ordering of memory operations may be relaxed if *data dependencies* and control flow (e.g., loops, branches) are preserved.

  This enables the hardware to hide memory latencies, by overlapping memory access operations with other operations when they are independent of each other.

data dependencies: occur if two operations R → W (anti-dep.), W → W (output dep.), W → R (flow dep.) access the *same memory location*.



- **Total store ordering (TSO)**: does not preserve W → R orderings.
- **Processor consistency (PC)**: does not preserve W → R orderings, and processors can execute reads before a previous write is observed by all processors.
- **Partial store ordering (PSO)**: does not preserve W → R and W → W orderings (i.e., total ordering on stores is still preserved).

In other words, the properties of cache coherence (i.e., program order, write propagation, and write serialization) are preserved by all models.

A summary of all consistency models is provided below:

Property	Sequential Consistency (SC)	Relaxed Consistency: Total Store Ordering (TSO)	Relaxed Consistency: Processor Consistency (PC)	Relaxed Consistency: Partial Store Ordering (PSO)
Respects data dependencies (e.g., don't touch: x = 5, read x) (but also W→W, R→W)	Yes			
Preserves R → R and R → W order	Yes			
Preserves W → R	Yes	**No**	**No**	**No**
Preserves W → W	Yes	Yes	Yes	**No**
Must wait for all processors to see same value before reading?	Yes	Yes	**No**	Yes

Relaxed consistency allows for increased performance during execution. Synchronization is needed with or without relaxed consistency.

8 PERFORMANCE INSTRUMENTATION

Lecture 8
3rd October 2022

Definitions:

- **Latency**: the time an operation spends waiting to be serviced.
- **Response time**: the time taken for an operation to complete (including time spent waiting & being serviced).
- **Throughput**: the rate of work performed.
- **IOPS** (input/output operations per second): the rate of data transfer.
- **Utilization**: measure of how busy a resource is (i.e., how much time in a given interval was it actively performing work).
- **Saturation**: degree to which a resource has queued extra work.
- **Bottleneck**: the resource that limits the performance of a system.
- **Workload**: the input/load applied to the system.

Different aspects of system performance include:

- Scalability.
- Reliability.
- Resource usage.
- Workload: normal vs overloaded.
- Timeline: not time sensitive (e.g., testing before release) vs time sensitive (e.g., incident performance response).

8.1 *Understanding system performance*

- **Perspectives**:
  - **Resource analysis**: examines the system's resources (e.g., CPU, memory, disks, network interfaces, interconnects, etc.), focusing on resource **utilization**. Some ways to measure utilization are:
    - **Time-based**: the average amount of time the server/resource was busy,

      $$U = B/T$$

      where T rep. the observation period and B rep. the total time the system was busy during T.
    - **Capacity-based**: the proportion of throughput the system/component is working at.

    Resource analysis includes:
    - Performance issue investigations.

    - Capacity planning: understanding of what will happen to the system if the load increases.
  - **Workload analysis**: examines the workload applied (i.e., *requests*) and how the application is responding (i.e., *latency* and *correctness*).
- **Methodologies**:
  - **Problem statement method**: asking concrete questions about w.r.t. system performance.
  - **USE method**: for every resource (e.g., CPU, memory, bus, etc.), check:
    - **U**tilization: busy time.
    - **S**aturation: queue length.
    - **E**rrors.
  - **Performance monitoring**: record performance statistics over time to compare past vs present performance, and identify time-based usage patterns.
  - **Performance analysis** in 60 seconds.
  - **Tools method**:
    1. List available performance tools.
    2. For each tool, list useful metrics it provides.
    3. For each metric, list possible ways to interpret it.

    Tools can be categorized as:
    - **Fixed counters**: kernels maintain various counters for providing system statistics, and expose these counters as metrics.
    - **Event-based counters**: enabled-as-needed, includes *profiling* and *tracing* capabilities.
  - **CPU profile method**: e.g., `perf`, `perf_events`, `gprof`, `vtune`, and *flame graphs*.

profiling: characterizes the target by collecting a set of samples/snapshots of its behaviour.
tracing: instruments every occurrence of an event, and stores event-based details for later analysis.

- **Instrumentation tools**: modify the source code, executable, or runtime environment to understand system performance. Types of tools include:
  - **Manual**: added to source code by the programmer.
  - **Automatic source level**: added to source code by an automatic tool.
  - **Intermediate language/compiler assisted**: added to assembly/decompiled bytecodes.
  - **Binary translation**: added to a compiled executable.
  - **Runtime instrumentation**: program run is fully supervised and controlled by the tool.
  - **Runtime injection**: code is modified at runtime to have jumps and helper functions.

  Typically, the addition of instrumentation tools introduces additional overhead to system execution. While time measurements may not be accurate, these measurements are still useful for (1) finding bottlenecks and (2) debugging (e.g., valgrind and *sanitizers*).

  - **Valgrind**: used for memory checks (e.g., leaks).
    * Heavy-weight binary instrumentation with significant (4x) overhead.
    * **Shadow memory** is used to track and store information on the memory that is used during a program's execution, and detect incorrect memory accesses.
  - **Sanitizers**: compilation-based approach to detect issues, typically incurs 2x overhead.
    * **AddressSanitizer**: detects out-of-bounds accesses and use-after-free.
    * **ThreadSanitizer**: detects data races.
    * **MemorySanitizer**: detects uninitialized reads.
    * **UndefinedBehaviourSanitizer**: detects undefined behaviour, e.g., integer overflow, null pointers.
    * **LeakSanitizer**: detects memory leaks.
- **Benchmarking**: while running benchmarks, analyze and confirm the performance limited using observability tools (e.g., `iostat`, `perf`, flame graphs, etc.).

9 PARALLEL PROGRAMMING MODELS II

Lecture 9
10th October 2022

9.1 *Data distribution*

Data distribution (a.k.a. data decomposition/partitioning): decomposing an (multidimensional) array across multiple processors.

To simplify discussion, we assume that we have p identical processors and n array elements:

- Distribution patterns on 1D arrays:
  - **Blockwise**:
    (+) Spatial locality of data → less cache misses.
    (−) Requires prior knowledge of the number of elements present in the array.
    (−) Generally poorer load distribution.
  - **Cyclic**:
    (+) Does not require prior knowledge of the size of the array.
    (+) Better load distribution.
    (−) High likelihood of cache miss when accessing elements (adjacent data elements are unlikely to be on the same cache line).
- Distribution patterns on 2D arrays:
  - **Column/Row** distributions:

    - Includes blockwise, cyclic, and **block-cyclic** (with block size $b > 1$) distributions.
  - **Checkerboard** distributions: processors are virtually organized into a 2D mesh.
    - Includes blockwise, cyclic, and block-cyclic distributions.

9.2 *Information exchange*

Information exchange is necessary for controlling the coordination of different parts of a parallel program execution. Some modes of information exchange include:

- **Shared address space**:
  - Assumes a global memory accessible by all processors.
  - Information exchange is done through **shared variables**, and **synchronization operations** are required for safe concurrent access.
    - Synchronized access can be guaranteed using *critical sections*, such as *mutexes* and *semaphores*.
    - Without synchronization, *data races* may occur, which leads to *non-deterministic behaviour*.

    data race: multiple threads accessing the sames shared variable.
  - (–) Overuse of shared address spaces results in significant overhead of resource contention.
- **Distributed address space**:
  - Assumes a disjoint memory space.
  - Information exchange is done through **communication operations** (e.g., *point-to-point* and *global communication*).
  - Common communication model: **message-passing**.

9.2.1 Message-passing model

Features:

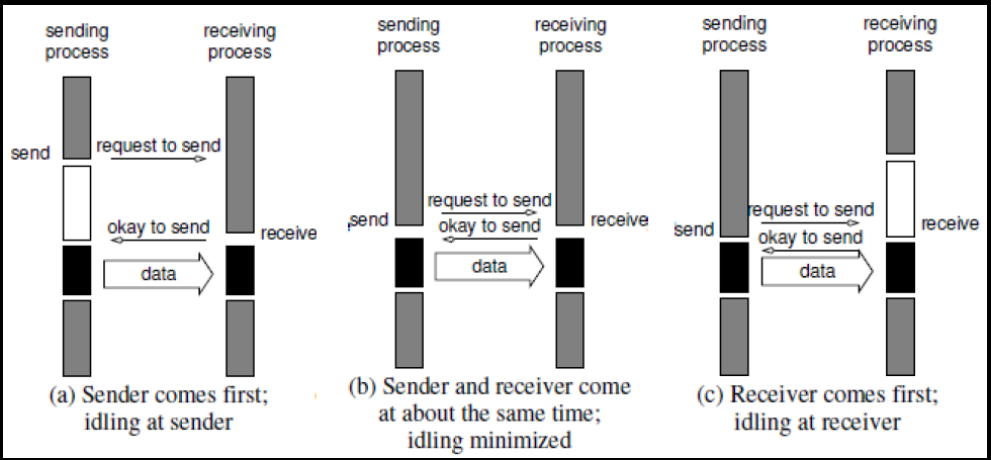
- Data is explicitly partitioned/separated for each process.
- Inter-process communication/interaction requires the participation of both parties.
- Programmer decides what gets to execute on each processor.

The model is based on a **loosely synchronous paradigm**, i.e.:

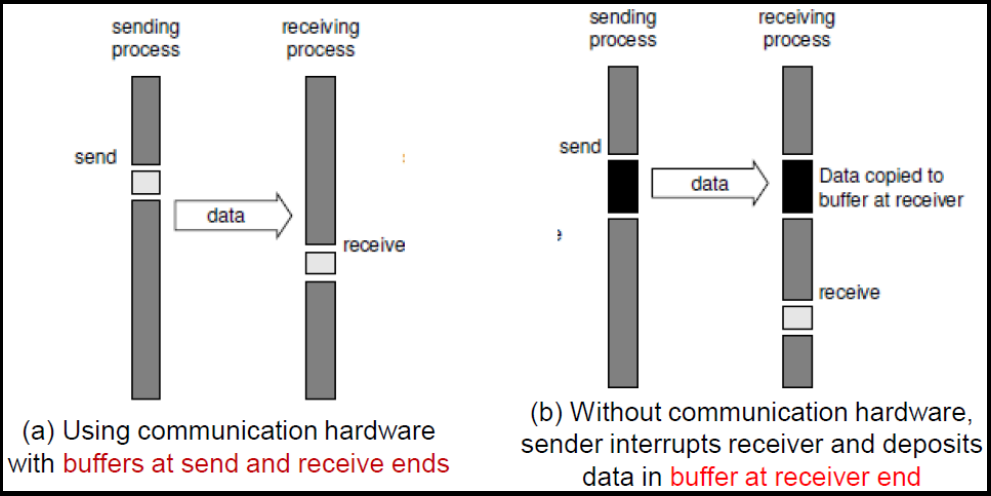
- Tasks/Subsets of tasks synchronize to perform interactions.
- Outside of interactions, tasks execute asynchronously.

9.2.2 Communication protocols

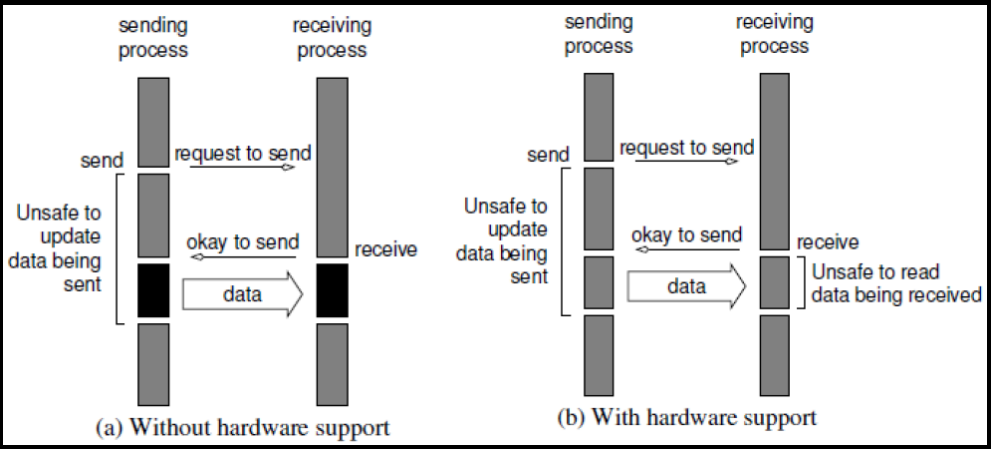
- **Blocking**:
  - **Buffered**: operation blocks until it is safe to reuse the input buffer.
    (+) No idling overhead.
    (–) Buffer copying overhead.
    (–) Problems with full/overflow buffer.
    (–) Possibility of deadlocks.



  - **Non-buffered**: operation blocks until the matching receive has been performed.
    (–) Considerable idling overheads (e.g. due to mismatch in timing between sender and receiver).
    (–) Possibility of deadlocks.



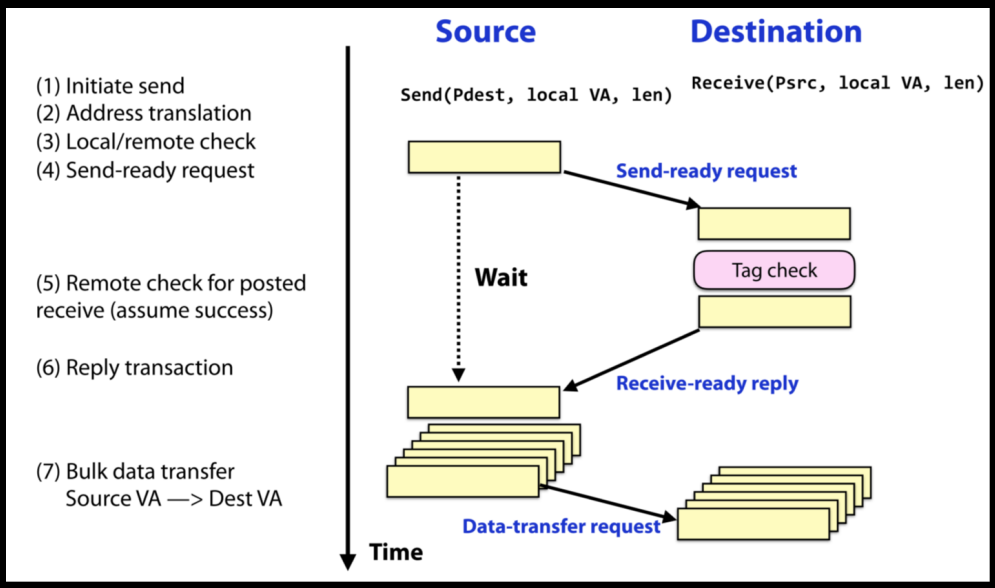
- **Non-blocking**: operation returns before it is semantically safe to use the data transferred.
  - Generally accompanied by a *check-status* operation.
  - Allows overlapping communication overheads with computation.



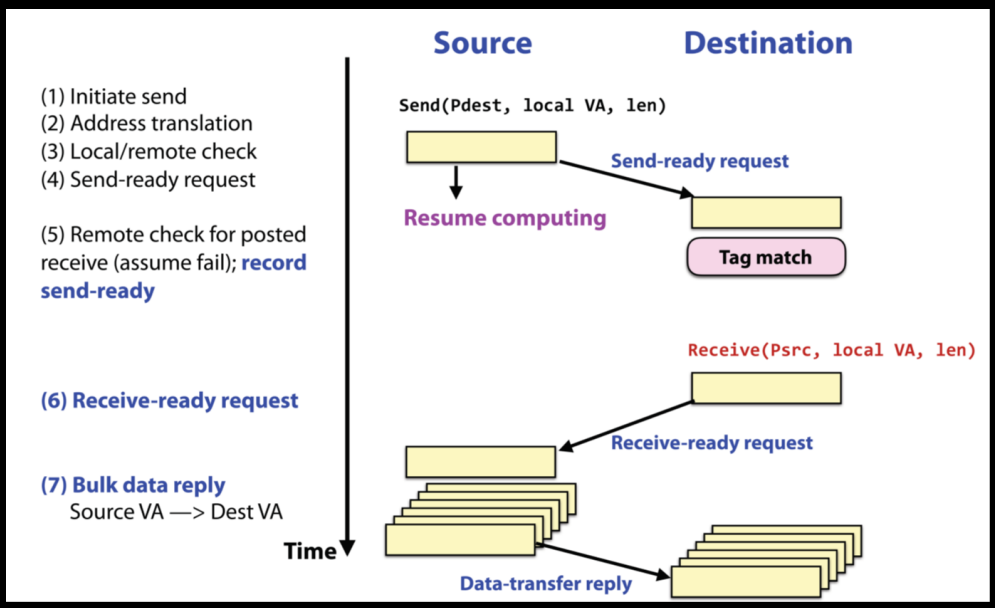
	Blocking	Non-blocking
Buffered	Sending process returns after data has been copied into the buffer.	Sending process returns after initiating the transfer to buffer. Operation might not be completed on return.
Non-buffered	Sending process blocks until matching receive operation has been encountered.	Sending process returns after initiating data transfer to receiving process.

In contrast to blocking, which only provides a **local view** (i.e., looks at a single side) of the entire communication process, protocols can also be differentiated based on a **global view** of the communication process.

- **Synchronous**: communication does not complete until both processes have started their communication operations.



- **Asynchronous**: both processes can execute their communication operations without any coordination with each other.



10 MESSAGE PASSING

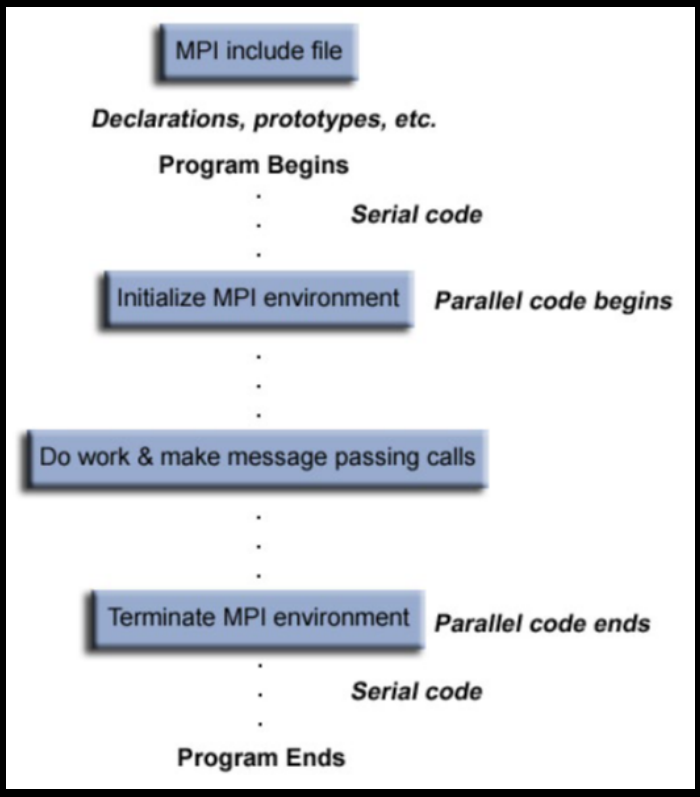
Lecture 10
17th October 2022

In a parallel computer with a *distributed address space*, **message-passing** is utilized to exchange data using communication operations. Under this paradigm, the programmer is responsible for identifying the parallelism.

10.1 *Message passing interface (MPI)*

MPI is a standardization of a message-passing library interface specification, provided as a set of library calls. Some implementations of MPI include MPICH, LAM/MPI, and OpenMPI.

In general, MPI programs have the following structure:



1. Initialize communications (e.g., `MPI_Init`).
2. Perform communications necessary for coordinating computation, e.g.:

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

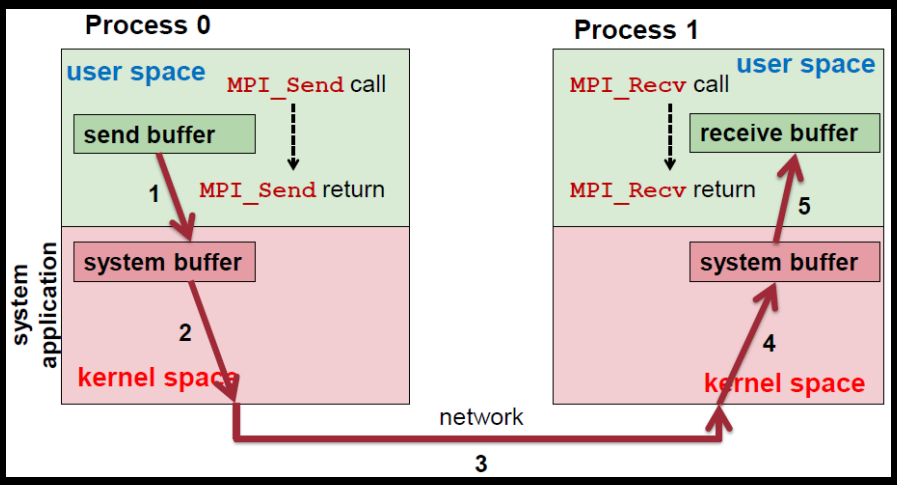
   Note: blocking and non-blocking operations can be mixed.
3. Exit from the message-passing system (e.g., `MPI_Finalize`).

In MPI, messages have the following format:

1. **Data**: actual data to send/receive.
   (a) **Start-buffer**: address where the data starts.
      - Received messages must be less than or equal to the length of the receive buffer.
   (b) **Count**: number of data elements in the message.
   (c) **Datatype**: type of data to be transmitted.
2. **Envelope**: how to route the data.
   (a) **Source/Destination**: rank of the process in a communicator.
      - For receiving messages, we can use src = `MPI_ANY_SOURCE`.
   (b) **Tag**: arbitrary number to distinguish between messages.
      - For receiving messages, we can use tag = `MPI_ANY_TAG`.
   (c) **Communicator**.

10.1.1 Point-to-point communication

The process of sending/receiving in MPI is illustrated in the figure below:



If there are only two processes, messages could be expected to be delivered in the order which they have been sent. However, there are no guarantees on delivery ordering if more than 2 processes are communicating with each other simultaneously.

Deadlocks may occur too, when:

- Message passing cannot be completed (e.g., both processes employ blocking recv before send).

- If runtime system does not use system buffers, or if the system buffers used are too small (e.g., both processes employ blocking send before recv → both sends cannot complete → deadlock).

To avoid deadlocks, MPI programs should not depend on assumptions about specified properties of the runtime system (e.g., that system buffers are used).

10.1.2 Collective communication

Definitions:

- **Process group**: an ordered set of processes, each having a unique **rank**.
  - A process may be a member of multiple groups, and may have different ranks in each group.
- **Communicator**: communication domain for a group of processes. Types include:
  - **Intra-communicator**: supports the execution of arbitrary collective communication operations of a single group.
  - **Inter-communicator**: supports the point-to-point communication operations between two process groups.
- **Virtual topology**: a communicator with a Cartesian style of addressing the ranks of the processes.

Groups and communicators are useful because they:

- Allow us to organize tasks into task groups.
- Enable collective communication operations across sets of related tasks.
- Provide a basis for user-defined virtual topologies.
- Provide safe communication (e.g., only informing necessary processes w.r.t. some information).

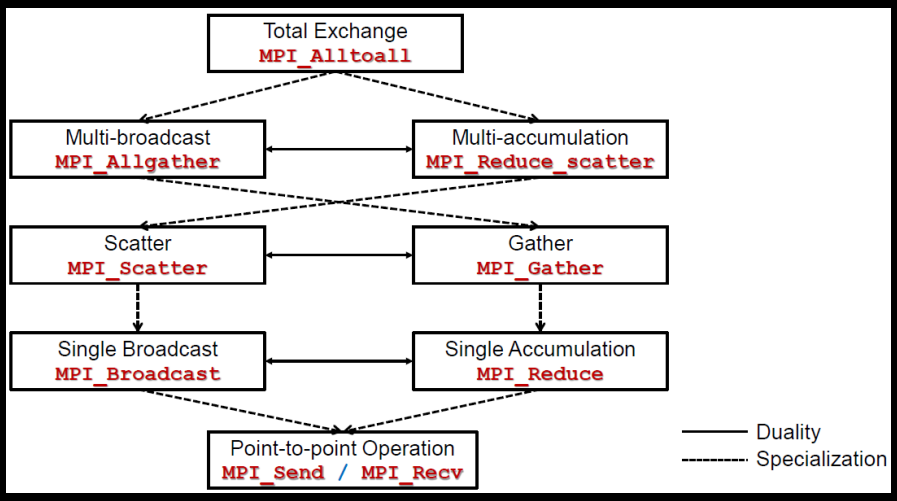
Collective communication consists of (blocking) operations that involve all processes in a communicator. If some process in a communicator does not call the same collective communication operation, deadlock results. Some types of operations include:

There also exists a collective synchronization operation, `MPI_Barrier`.

- **Single transfer**.
- **Gather/Scatter**: root processor sends/receives *different data* to/from different processors.
- **Single/Multi-broadcast**: sender(s) sends the *same data* block to all other processors.

- **Single/Multi-accumulation**: each processor provides a block of data with the same type and size, and a **reduction** operation is applied element-wise to the data blocks.
- **Total exchange**.

A collective communication operation can be presented as a graph/spanning tree illustrating how messages are passed to each node. Two communication operations are said to be a **duality** if the same spanning tree can be used for both operations.



Communication operations can also be ordered into a hierarchy from the most general to the most specific. An operation X is a **stepwise specialization** of another operation Y if it can be considered as a subset of Y (i.e., providing more specific functionality).

11 INTERCONNECTION NETWORKS

Lecture 11
31st October 2022

Some parallel sorting algorithms include:

- **Odd-even transposition sort**: i.e., standard linear sort done in parallel.
  1. Swap each odd element x_i with its right neighbour if $x_i > x_{i+1}$.
  2. Swap each even element with its right neighbour if necessary.
  3. Repeat from step 1 until no more swaps are needed.

  The time complexity of the algorithm is $O(n)$, which is faster than sequential sorting algorithms where comparisons are performed.

- **Shear sort**: given a 2D matrix, arrange its elements in a snake-like manner.

  1. Row sorting: sort odd rows in ascending order, and even rows in descending order.
  2. Column sorting: sort all columns in ascending order (i.e., top to bottom).
  3. Repeat from step 1 until all rows are sorted.

  Since the algorithm takes $\log_2 N + 1$ phases, and each phase involves transposition sorting (with a time complexity of $O(n)$, where $n = \sqrt{N}$), the overall complexity of shear sort is $O(\sqrt{N} \log N)$.

11.1 *Topology*

Interconnection networks assist in the communication between processors, memories, caches, and I/O devices. There are two main types of topologies:

- **Direct/Static/Point-to-point**: endpoints (e.g., processing elements, memory, etc.) are directly connected with each other, and are usually of the same type. Switches are found on each endpoint.
- **Indirect/Dynamic**: endpoints communicate with each other via an interconnect formed by switches; different types of endpoints may be involved.

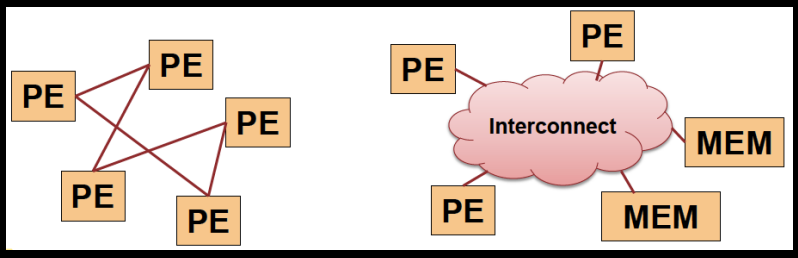


Figure 10: The left figure illustrates direct interconnection, whereas the right figure illustrates indirect interconnection.

11.1.1 Direct interconnection networks

Direct interconnection network topology may differ in shape and size:

- Basic shapes: e.g., linear array, ring, complete graph, binary trees.
- Multidimensional cubes: e.g., k-dimensional hypercube, k-dimensional CCC-network.
- Others: e.g., d-dimensional mesh, d-dimensional torus, k-ary d-cubes.

cube-connected-cycles (CCC) networks: networks where each node in a k-dimensional hypercube is subtituted with a cycle of k-nodes.

Key topology metrics include:

- **Diameter**, δ(G): maximum distance between any pair of nodes in network G.
  - A small diameter ensures small distances for message transmission.
- **Degree**, $g(v)$: number of direct neighbours of node v.

  Degree, g(G): maximum degree of a node in a network G.
  - A small node degree reduces the node hardware overhead.
- **Bisection width**, B(G): minimum number of edges that must be removed to divide network G into two halves of equal size.
  - The bisection width is a useful measure for the capacity of a network when transmitting messages simultaneously.
- **Node connectivity**, nc(G): minimum number of nodes that must fail to disconnect the network.
  - Node connectivity helps to determine the robustness of a network.
- **Edge connectivity**, ec(G): minimum number of edges that must fail to disconnect the network.
  - Edge connectivity helps to determine the number of independent paths between any pair of nodes.

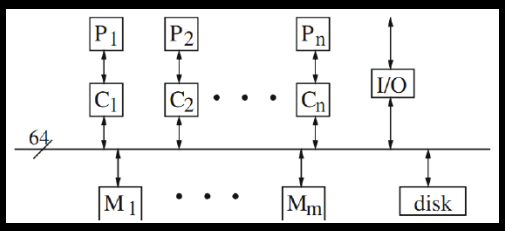
network G with n nodes	degree $g(G)$	diameter $\delta(G)$	edge-connectivity ec(G)	bisection bandwidth $B(G)$
complete graph	$n-1$	1	$n-1$	$\left(\frac{n}{2}\right)^2$
linear array	2	$n-1$	1	1
ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d-dimensional mesh ($n = r^d$)	$2d$	$d(\sqrt[d]{n}-1)$	d	$n^{\frac{d-1}{d}}$
d-dimensional torus ($n = r^d$)	$2d$	$d\left\lfloor \frac{\sqrt[d]{n}}{2} \right\rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-dimensional hypercube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
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 ($n = k^d$)	$2d$	$d\left\lfloor \frac{k}{2} \right\rfloor$	$2d$	$2k^{d-1}$

Figure 11: Summary of metrics for different network topologies.

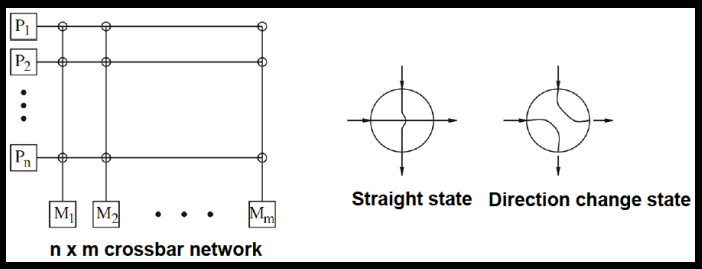
11.1.2 Indirect interconnection networks

Indirect interconnection networks also come in several different variants:

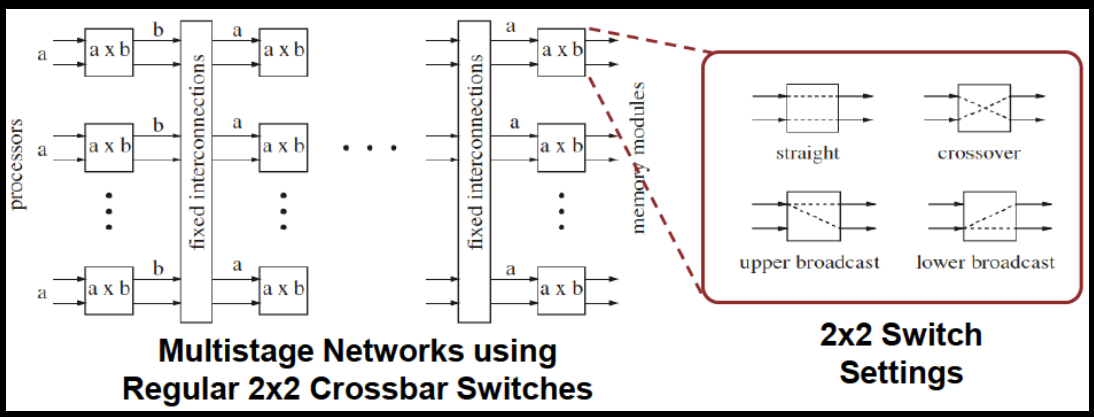
- **Bus network**: a set of wires are used to transport data from a sender to a receiver; only one pair of devices can communicate at a time.



- **Crossbar network**: a $n \times m$ crossbar network has n inputs and m outputs and $n \times m$ switches; each switch can be in one of two states.



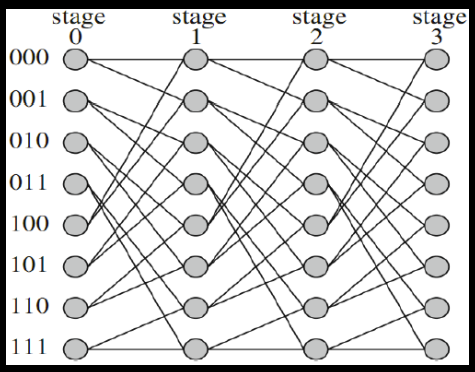
- **Multistage switching network**: several intermediate switches are employed, with connecting wires between neighbouring stages. The goal is to obtain a small distance for arbitrary pairs of input and output devices.



- **Omega network**: an $n \times n$ Omega network has $\log n$ stages, with $n/2$ switches per stage. The connections between stages are regular, and there is one unique path for every input to output.

  Each node (α, i) connects to two nodes $(\beta_1, i+1), (\beta_2, i+1)$ where:

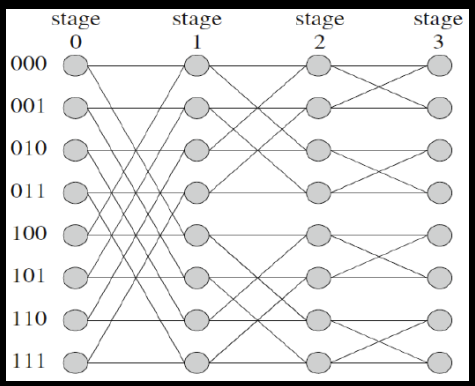
  1. $\beta_1 = \alpha$ by a cyclic left shift (e.g., $011 \rightarrow 110$).
  2. $\beta_2 = \alpha$ by a cyclic left shift, and invert the last bit (e.g., $011 \rightarrow 111$).



- **Butterfly network**:

  Each node (α, i) connects to two nodes $(\beta_1, i+1), (\beta_2, i+1)$ where:

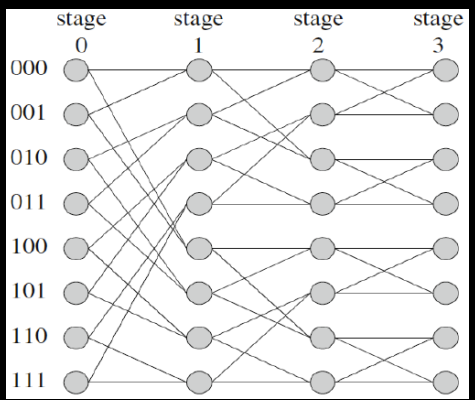
  1. $\beta_1 = \alpha$ (i.e., a straight edge).
  2. $\beta_2 = \alpha$ inverting the $(i+1)^{\text{th}}$ bit from the left (e.g., $011 \rightarrow 001$).



- **Baseline network**:

  Each node (α, i) connects to two nodes $(\beta_1, i+1), (\beta_2, i+1)$ where:

  1. β_1 = cyclic right shift of the last $k-i$ bits of α (e.g., $011 \rightarrow 101$).
  2. β_2 = invert the last bit of α, and cyclic right shift of the last $k-i$ bits (e.g., $011 \rightarrow 001$).



11.2 *Routing*

Routing algorithms determine the path(s) taken by a message from source to destination, within a given interconnect topology. They can be classified based on:

- **Path length** (minimal vs non-minimal): whether the shortest path is always chosen.
- **Adaptivity** (deterministic vs adaptive):
  - **Deterministic**: always use the same path for the same pair of <source, destination> nodes.
  - **Adaptive**: may take into account the network status and adapt accordingly (e.g., avoid congested paths).

Some routing algorithms include:

- XY **routing** (for 2D mesh):
  1. Move in X direction until $X_{src} == X_{dst}$.
  2. Move in Y direction until $Y_{src} == Y_{dst}$.
- **E-cube routing** (for hypercube):

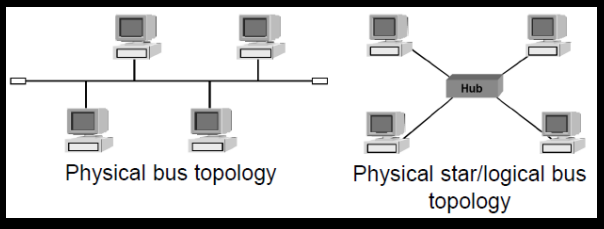
  Let $(\alpha_{n-1}\alpha_{n-2}...\alpha_1\alpha_0)$ and $(\beta_{n-1}\beta_{n-2}...\beta_1\beta_0)$ be the bit representations of source and destination node addresses respectively. Starting from the MSB to LSB,
  1. Find the first different bit.
  2. Go to the neighbouring node with the bit corrected.
  3. Repeat from step 1 until $(\alpha_{n-1}\alpha_{n-2}...\alpha_1\alpha_0) = (\beta_{n-1}\beta_{n-2}...\beta_1\beta_0)$.

  The number of different bits between the source and target node addresses is known as the **hamming distance**.
- **XOR-tag routing** (for Omega network):

  Let $T = (\alpha_{n-1}\alpha_{n-2}...\alpha_1\alpha_0) \oplus (\beta_{n-1}\beta_{n-2}...\beta_1\beta_0) = (\gamma_{n-1}\gamma_{n-2}...\gamma_1\gamma_0)$.
  1. At each stage k,
     - Go straight if $\gamma_k = 0$.
     - Crossover if $\gamma_k = 1$.

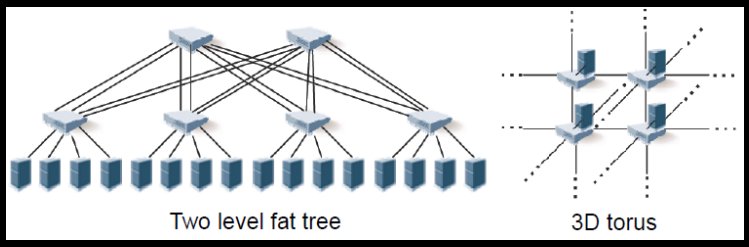
Modern interconnect systems include:

- **Ethernet**:



in **broadcast**, a message is delivered to all the endpoints on the network; in **multicast**, a message is only delivered to intended recipients.

  - Supports *point-to-point* and *broadcast*.
  - Commonly-used topologies: physical bus, physical star with logical bus, etc.
- **InfiniBand**:



  - Supports *point-to-point* and *multicast*.
  - Commonly-used topologies: fat tree, torus, etc.

12 ENERGY-EFFICIENT COMPUTING

Ways of achieving energy efficiency include:

- **Heterogeneous computing**: using brawny/high-performance CPUs for compute intensive applications (e.g., gaming, social media tools), and wimpy/low-power CPUs for most other workloads (e.g., push notifications, reminders).

Power Use Effectiveness (PUE) is a measure of energy efficiency in data centers.

- **Energy-efficient data centers**:
  - Continuously measure efficiency.
  - Building custom, highly-efficient servers (e.g., strategic rack positioning, remove redundant parts, minimize power loss in AC/DC conversions, decreasing fan speed to optimize cooling).
  - Extending equipment lifecycle (e.g., reuse and resell components).
  - Controlling equipment temperature (e.g., using thermal modelling to avoid hot spots, adopt hot/cold aisles, and manage airflow).
  - Employ water instead of air cooling.
- Reduce data movement: use compression techniques and exploit locality.
- Use specialized processing: e.g., using a mix of CPU-like and GPU-like cores (i.e., throughput optimized), FPGAs, etc.

A C CHEATSHEET

A.1 *Processes*

- `pid_t fork();` `#include <unistd.h>`
  - creates a new process.
- `pid_t wait(...);` `#include <sys/wait.h>`

  `pid_t waitpid(pid_t pid, ...);`
  - suspends execution of the calling thread until a child (specified by pid argument) has changed state.

A.1.1 Shared memory

- `key_t ftok(const char *pathname, int proj_id);` `#include <sys/ipc.h>`
  - uses the identity of the file named by the given `pathname` and the least significant 8 bits of `proj_id` to generate a `key_t` type System V IPC key.
- `int shmget(key_t key, ...);` `#include <sys/shm.h>`
  - returns the identifier of the System V shared memory segment associated with the value of the argument `key`.
- `void *shmat(int shmid, const void *shmaddr, ...);` `#include <sys/shm.h>`
  - attaches the shared memory segment associated with the shared memory identifier specified by `shmid` to the address space of the calling process.
- `int shmdt(const void *shmaddr);` `#include <sys/shm.h>`
  - detaches the shared memory segment located at the address specified by `shmaddr` from the address space of the calling process.
- `int shmctl(int shmid, int cmd, ...);` `#include <sys/shm.h>`
  - performs the control operation (e.g., `IPC_RMID`: destroying the shared memory) specified by `cmd` on the System V shared memory segment whose identifier is given in `shmid`.

A.1.2 Semaphores

Need to compile with `-pthread`.

- `sem_t *sem_open(const char *name, int oflag, ...);`
  ```
  #include <fcntl.h>
  #include <sys/stat.h>
  #include <semaphore.h>
  ```
  - initialize and open a named semaphore.

`#include <semaphore.h>`

- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - initialize an unnamed semaphore at the address pointed to by `sem`, with initial value `value`.

`#include <semaphore.h>`

- `int sem_unlink(const char *name);`
  - removes the named semaphore referred to by `name`.

`#include <semaphore.h>`

- `int sem_close(sem_t *sem);`
  - closes the named semaphore referred to by `sem`, allowing any resources that the system has allocated to the calling process for this semaphore to be freed.

`#include <semaphore.h>`

- `int sem_destroy(sem_t *sem);`
  - destroys the unnamed semaphore at the address pointed to by `sem`.

`#include <semaphore.h>`

- `int sem_wait(sem_t *sem);`
  `int sem_trywait(sem_t *sem);`
  `int sem_timedwait(sem_t *restrict sem, ...);`
  - decrements the semaphore pointed to by `sem`.

`#include <semaphore.h>`

- `int sem_post(sem_t *sem);`
  - increments the semaphore pointed to by `sem`.

A.2 *Threads*

Need to compile with `-pthread`.

`#include <pthread.h>`

- `int pthread_create(...);`
  - starts a new thread in the calling process.

`#include <pthread.h>`

- `int pthread_join(pthread_t thread, ...);`
  - waits for the thread specified by `thread` to terminate.

`#include <pthread.h>`

- `void pthread_exit(void *retval);`
  - terminates the calling thread and returns a value via `retval` that (if the thread is joinable) is available to another thread in the same process that calls `pthread_join(3)`.

A.2.1 Mutexes

Need to compile with `-pthread`.

`#include <pthread.h>`

- `int pthread_mutex_init(*restrict mutex, *restrict attr);`
  - initialize the mutex referenced by `mutex` with attributes specified by `attr`.

- `pthread_mutex_destroy(pthread_mutex_t *mutex);` `#include <pthread.h>`
  - destroy the mutex object referenced by `mutex`; the mutex object becomes uninitialized.
- `int pthread_mutex_lock(pthread_mutex_t *mutex);` `#include <pthread.h>`

  `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - acquires the mutex object referenced by `mutex`.
- `int pthread_mutex_lunock(pthread_mutex_t *mutex);` `#include <pthread.h>`
  - release the mutex object referenced by `mutex`.

A.2.2 Conditional variables

Need to compile with `-pthread`.

- `int pthread_cond_init(*restrict cond, *restrict attr);` `#include <pthread.h>`
  - initialize the condition variable referenced by `cond` with attributes referenced by `attr`.
- `int pthread_cond_destroy(pthread_cond_t *cond);` `#include <pthread.h>`
  - destroy the given condition variable specified by `cond`; the object becomes uninitialized.
- `int pthread_cond_wait(*restrict cond, *restrict mutex);` `#include <pthread.h>`

  `int pthread_cond_timedwait(*restrict cond, *restrict mutex, ...);`
  - atomically release `mutex` and cause the calling thread to block on the condition variable `cond`.
  - e.g., `pthread_mutex_lock(&mutex);`

    ```
    while (cond) pthread_cond_wait(&cond_var, &mutex);
    pthread_mutex_unlock(&mutex);
    ```
- `int pthread_cond_signal(pthread_cond_t *cond);` `#include <pthread.h>`
  - unblock at least one of the threads that are blocked on the specified condition variable `cond` (if any threads are blocked on `cond`).
- `int pthread_cond_broadcast(pthread_cond_t *cond);` `#include <pthread.h>`
  - unblock all threads currently blocked on the specified condition variable `cond`.

A.3 *Signals*

- `sighandler_t signal(int signum, sighandler_t handler);` `#include <signal.h>`
  - sets the disposition of the signal `signum` to `handler`.

#include <unistd.h>

- `int pthread_sigmask(...);`
  - used to fetch and/or change the signal mask (i.e., set of signals whose delivery is currently blocked for the caller) of the calling thread.

A.4 *OpenMP*

Need to compile with `-fopenmp`.

A.4.1 Directives & constructs

#include <omp.h>

- `#pragma omp parallel [clause]...`
  - creates a team of OpenMP threads that execute the region, each with a unique thread id.
  - by default, OpenMP creates a number of threads equal to the number of cores on the machine.
  - clause `shared(list)`: variables in `list` are shared between threads or explicit tasks executing the construct.

#include <omp.h>

- `#pragma omp for`
  - specifies that the iterations of associated loops will be executed in parallel by threads in the team.
  - clause `schedule (static, chunk_size)`: iterations are divided into chunks of size `chunk_size`, and assigned to team threads in round-robin fashion in order of thread number.

#include <omp.h>

- `#pragma omp sections` → `#pragma omp section`
  - specifies a `parallel` construct containing a `sections` construct and no other statements.

#include <omp.h>

- `#pragma omp barrier`
  - synchronizes all threads (threads wait until all threads arrive at the barrier).

#include <omp.h>

- `#pragma omp master`
  - specifies a region that must be executed only by the master thread.

#include <omp.h>

- `#pragma omp critical`
  - specifies a critical section that must be executed only by one thread at a time.

#include <omp.h>

- `#pragma omp atomic`
  - specifies that a specific memory location must be updated atomically.

A.4.2 Routines

- `void omp_set_num_threads(int num_threads);` `#include <omp.h>`
  - sets the value of the first element of the `nthreads-var` ICV of the current task to `num_threads`.

ICV: internal control variables.

- `int omp_get_num_threads();` `#include <omp.h>`
  - returns the number of threads in the current team.
- `int omp_get_thread_num();` `#include <omp.h>`
  - returns the thread number of the calling thread.

A.4.3 Environment variables

- `OMP_NUM_THREADS`
  - sets the initial value of `nthreads-var` ICV for the number of threads to use for parallel regions.

More OpenMP constructs and routines can be found here and here (short version).

A.5 *CUDA*

Refer to this page.

A.6 *MPI*

- `MPI_Init(...);`
  - initializes the MPI execution environment.
- `MPI_Comm_rank(MPI_Comm comm, int *rank);`
  - determines the rank of the calling process in the communicator.
- `MPI_Comm_size(MPI_Comm comm, int *size);`
  - determines the size of the group associated with a communicator.
- `MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`

  `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm);`
  - performs a send/receive operation.

- `MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`
  - broadcasts a message from the process with the given rank to all other processes of the communicator.
- `MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);`
  - reduces values on all processes to a single value.
- `MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);`
  - gathers together values from a group of processes.
- `MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype datatype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);`
  - sends data from one process to all other processes in a communicator.
- `MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);`
  - gathers data from all tasks and distribute the combined data to all tasks.
- `MPI_Reduce_scatter(void *sendbuf, void *recvbuf, int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);`
  - combines values and scatters the results.
- `MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);`
  - sends data from all processes to all processes.
- `MPI_Barrier(MPI_Comm comm);`
  - blocks until all processes in the communicator have reached this routine.
- `MPI_Wait(MPI_Request *req, MPI_Status *status);`

  `MPI_Waitall(int count, MPI_Request reqarr[], MPI_Status statarr[]);`
  - waits for an MPI request to complete.
- `MPI_Finalize();`
  - terminates MPI execution environment.

B LINUX CHEATSHEET

- `hostname`
  - print the name of the system.
- `ps`
  - report a snapshot of the current processes.
- `lscpu`
  - displays information about the CPU architecture.
- `less`
  - filter for paging through text one screenful at a time.
- `cat [OPTION]... [FILE]...`
  - concatenate `FILE`s and print on standard output.
- `grep [OPTION...] PATTERNS [FILE...]`
  - searches for `PATTERNS` in each `FILE`.
- `top`
  - provides a dynamic real-time view of a running system (e.g., system summary information, list of processes/threads).
- `kill [-signal|-s signal|-p] ... pid|name...`

  `pkill`

  `killall`
  - sends the specified signal to the specified processes or process groups.
- `watch [options] command`
  - runs `command` repeatedly, displaying its output and errors (the first screenful).
- `perf stat [-e <EVENT> | -M <METRICS>] [-r` k`] [-a] COMMAND`
  - runs a `COMMAND` and gathers performance counter statistics from it.
  - optional: include a comma-delimited list of `EVENT`s or `METRIC`s to specify which events/metrics we wish to measure.
  - optional: include `-r` k to repeat runs k times.
- `perf list`
  - displays the symbolic event types which can be selected in the various `perf` commands with the `-e` option.

B.1 *Debugging*

- `gdb`
- `valgrind`

C SLURM CHEATSHEET

Link to CS3210 Slurm guide.

- `sinfo`: view the status of node partitions in the lab.
- `sacct`: view running/completed jobs and Slurm accounting data.
- `squeue`: view running jobs.
- `sbcast`: broadcast a file to all allocated nodes.
- `scancel`: kill submitted Slurm jobs.
- `sbatch`: submit a Bash script to Slurm for execution; runs in a **batch**, **non-interactive** mode. Params:
  - `--job-name`
  - `--nodes`
  - `--ntasks`
  - `--mem`
  - `--time`
  - `--output`
  - `--error`
- `srun [command]`: sends `command` as a job to the Slurm system, in an **interactive**, **blocking** mode.
  - `-Nk`: run the job on k separate nodes.
  - `-nk`: run k tasks in total across all nodes.
  - `--partition P`: execute the job on a node within the partition P.
  - `-w X`: execute the job on the node X.

More Slurm commands can be found here.

* * *