CS3210 Finals Notes

Chapter 1 - Introduction

Motivation: Instead of trying to increase and optimize the computation power of a single processor, it is ideal to try to make multiple processors work together on the same task instead to obtain more computing power.

Challenges: We must write programs such that the subroutines/subtasks that we get our machine units to perform are synchronized, without which would result in ‘wasted’ computation power since some machines would just be sitting idle.

Performance improvement could come in the form of:

- Higher clock frequency (Increasing processing rate and speed of data transfer)
- Pipelining, superscalar processor (overlapping of instructions where we can)
- Replication (on multicore or performing the task on compute cluster)

Parallel Computing refers to the **simultaneous** use of **multiple processing units** to solve a problem faster/solve a larger problem. These processing units could be at the scale of

1. A single processor with multiple cores
2. A single computer with multiple processors
3. A number of computers connected by a network

In order to do this, we need to divide a problem into discrete parts that can be solved concurrently, which can be subdivided into a series of instructions that can potentially be executed in parallel on different processing units. Defining these tasks are challenging!

- If we pick too **large** of a task, it is computationally heavy and would take a long time to execute

- However, if we pick to **small** of a task, it means a lot of overhead will be incurred in switching tasks

Parallel computing does not necessarily achieve faster performance than sequential methods sometimes! This is due to the heavy **overhead** by performing the parallel execution, and it is only beneficial if the amount of work that has to be done allows us to take advantage of the speedup by executing on multiple processing units.

Some tasks could depend on each other in what is known as **data/control dependencies.** In order to make sure we are executing correctly, synchronization and coordination needs to be done between processes and threads.

Processes use **distributed memory** while threads use **shared-memory.**

Concurrency vs Parallelism

Concurrency: Two or more tasks run and complete in overlapping time periods (interleaving their execution), and they might not be executing on the CPU at the same instant.

Parallelism: Two or more tasks can run simultaneously, and they actually execute at the same instant.

When we talk about performance measures, we look at execution time vs throughput.

Execution time: Computation time + Parallelization overheads (distribution of work, information exchange, idle time)

Throughput: Work done by a process/system

Chapter 2 - Processes, Threads, and Synchronization

In order to parallelize a program, there are 3 main steps:

1. **Decomposition**
2. **Scheduling**
3. **Mapping**

We first need to **decompose** the computations into smaller tasks, then **schedule** these tasks on processes/threads before **mapping** the processes to physical processors/threads to cores

Typically, only the **decomposition** step needs to be done by the programmer and the other two steps are covered by the **OS & libraries.**

Processes

A unique process is identified by its PID. A process comprises an executable program, global data, local data (stack/heap) and register values. Each process has its own address space and therefore **exclusive** access to its data, but this data can be shared among processes using explicit communication techniques:

1. Shared memory
2. Shared variables
3. Message passing
4. pipes

When we have multiple processes to execute, there can be two types of execution:

1. Time slicing execution (concurrent) where tasks constantly contact switch between each other and have interleaving executions. When contact switch occurs, there is some overhead since the states of the suspended process must be saved/the new process must be loaded.
2. Parallel execution of processes on different resources

When we do data sharing using IPC techniques mentioned above,

- for shared memory/variable we have to protect access when reading/writing with locks
- for message passing, there can be blocking/nonblocking and synchronous/asynchronous
- for UNIX, pipes and signalling

Exceptions

Synchronous as they occur during a program's execution and we typically have to include an exception handler since it could lead to possible termination of the program

Interrupts

Caused by an external resource (i.e. mouse movement/keyboard key pressed). Asynchronous as they occur independently of a program's execution. We have to execute an interrupt handler.

Disadvantage of processes: Creating a new process is costly as there are overhead of system calls, and communication goes through the operating system and can be costly.

Threads

A single process may consist of multiple independent control flow called threads. Threads all share the address space of the process, and all threads belonging to the same process see the same values.

Each thread has its own stack and registers so that it can execute its own independent control flow.

Since no copying of address space is involved in the creation of a thread, thread generation is faster than process generation.

When multiple threads are created in a process, they can be assigned to run on different cores of a multicore processor, only when each thread is assigned to a separate kernel thread.

There are 2 kinds of threads: User-Level and Kernel

1. User-Level Threads

Context switching between threads is fast, but OS unaware of user-level threads, and it cannot map different threads of the same process to separate execution resources → no parallelism. Since the OS does not manage these threads, if one user-level thread executes a blocking operation the OS cannot switch to another thread.

2. Kernel Threads

Operating system which manages the threads only know about kernel threads (which processes are modelled as). Unlike user-level threads, these threads can execute in parallel. Therefore, in a multicore system we can make efficient use of all the cores.

Number of threads should be:

- suitable to parallelism degree of application
- suitable to available execution resources
- sufficiently small to keep overhead low, but at the same time sufficiently large such that maximize the degree of parallelisation

In multithreaded programs, threads share resources and access shared data structures. They coordinate/interleave their execution arbitrarily. Synchronization required.

Shared Resources

Problem: If two/more threads are accessing a shared variable and that variable is read/modified/written to by these threads, we need to control access to the variable in order to avoid erroneous behavior. If not it could lead to race conditions

when at least one of the threads modify the shared resource. This can be done through several mechanisms.

Mutual Exclusion: The code sequence that uses ME is called critical section. One thread at a time can execute in the critical section (blocked by several mechanims) and all other threads have to wait on entry, until the previous thread leaves the critical section.

Blocking mechanisms:

1. Locks

Using paired calls to acquire() and release(). An acquire call does not return until the previous holder releases. If these calls are not paired, a thread can hold a resource forever and result in deadlock.

2. Semaphores

Wait will decrement the semaphore, signal will increment the semaphore. We can initialize the semaphore to any (positive) whole number and its value will always be maintained at larger or equal to 0.

Mutex semaphores serve the purpose of mutual exclusion and represent a single access to a resource, while counting semaphores means multiple (set number) of threads can pass the semaphore, determined by the semaphore count. Upon releasing the resource, though there may be multiple other threads ‘waiting’, it is undefined which thread will run after the signal.

- Semaphores may be prone to bugs, if they are left at an unexpected state after usage (execution is complete)

Deadlock

Deadlock can only exist if the 4 conditions hold simultaneously:

1. Mutual exclusion - one resource is held, non shareable
2. Hold and wait - one process holding one resource and waiting for another resource

3. No pre-emption - critical sections cannot be aborted externally
4. Circular wait - P1 waits for P2, P2 waits for P3, P3 waits for P1

Producer-consumer problem

Producer

event = waitForEvent()

mutex.wait()

- buffer.add()

mutex.signal()

items.signal()

Consumer

items.wait()

mutex.wait()

- buffer.get()

mutex.signal()

event.process()

Reader-writers problem

Writer

roomEmpty.wait()

#critical section

roomEmpty.signal()

Reader

mutex.wait()

- increment reader, and if first reader then call roomEmpty.wait()

mutex.signal()

#critical section for reader

mutex.wait()

- decrement reader, if last reader is out then call roomEmpty,signal()

mutex.signal()

One key issue with reader writer is that the writer may experience starvation forever if there is a constant stream of readers. This can be prevented using turnstile implementation. When there is a writer that come and wants to enter, the turnstile block any subsequent reader from going in until the writer has finished its execution.

Chapter 3 - Parallel Computing Architectures

Previously in Chapter 2, we touched on Concurrency vs Parallelism. On a single core processor, are we able to have any parallelism?

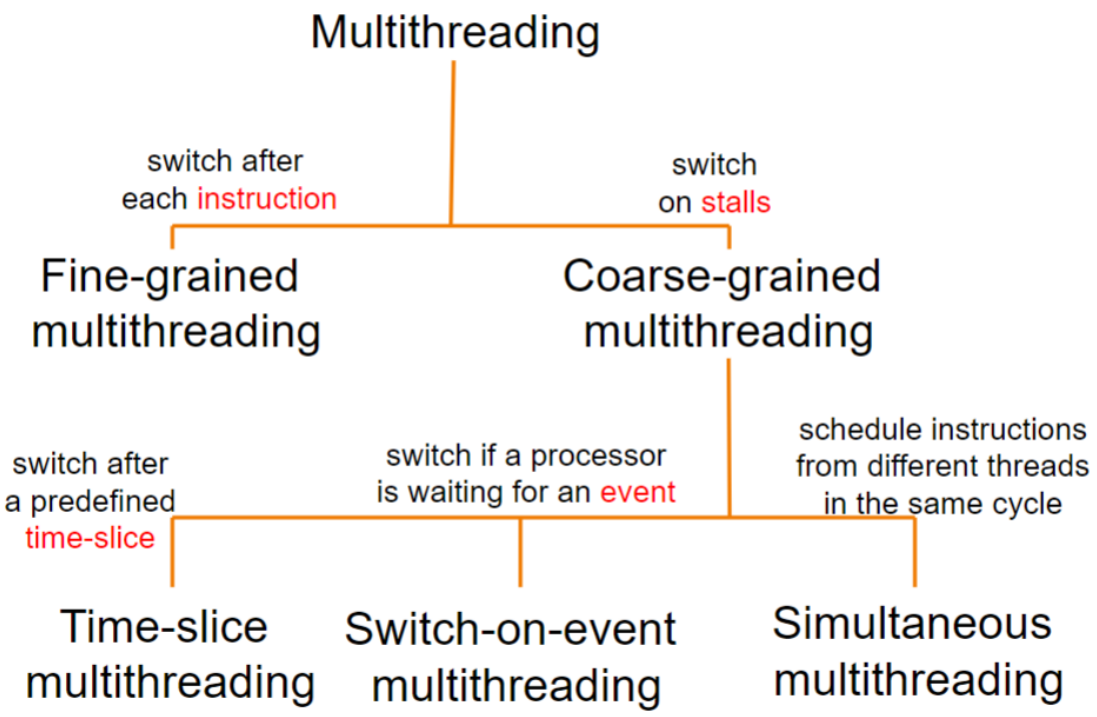
- Yes, because multiple data are used by a single instruction
- Yes, because multiple instructions can execute at the same time
- Yes, because multiple threads can execute at the same time on this core

Sources of Parallelism - Single Processor

Bit Level Parallelism: Work on multiple bits (words) at the same time

Instruction Level Parallelism: Work on multiple instructions at the same time using 2 different techniques - **pipelining and superscalar.**

Thread Level Parallelism: Processor works on multiple threads of the same program in parallel (simultaneous multithreading). Would incur more overhead since we need to manipulate multiple threads in a core.



Pipelining: Split instruction execution into multiple stages, and in the same clock cycle, we allow multiple instructions to occupy different stages. (i.e. instruction 1 at decode stage while instruction 2 at fetch stage at clock cycle 0), provided there is no **data dependencies.** Max speedup depends on the **number of pipeline stages.**

Disadvantages: If there are insufficient independent instructions, the pipeline will not be full and there are cycles with stages not filled by any instruction. Additionally, if the program contains if-else statements, there needs to be 'speculation' on which set of instructions should be loaded into the pipeline and if we speculate wrongly (bc. it is hard to know for sure), we need to clear the pipeline and reload it with new instructions. Finally, we need to identify out-of-order execution to fit the pipeline while preserving read-after-write ordering.

Superscalar: Duplication of the pipelines and allowing multiple instructions to go through the same stage at once → we can finish more instructions per cycle. It is however, challenging to decide which instructions can be executed together.

Disadvantages: It is hard to find enough instructions (from the same execution flow or **thread**) to maximize the superscalar pipeline.

Limitations: In a typical program, only 2-3 instructions can be executed in parallel due to data dependencies.

Sources of Parallelism - Multiple processor

We can enable this by adding more cores to the processor, and each process/thread needs independent context that can be mapped to multiple processor cores.

Flynn's Taxonomy

Single Instruction Single Data (SISD): Single instruction stream executed, each instruction work on single data (uniprocessor)

Single Instruction Multiple Data (SIMD): Single instruction stream executed, each instruction work on multiple data. One instruction is being shared by multiple Processing Units, with each PU working this instruction on different elements of the data pool.

Multiple Instruction Single Data (MISD): Multiple instruction streams which work on the same data at any time.

Multiple Instruction Multiple Data (MIMD): Each processor unit gets its own instruction and operates on its own data.

(See diagram representation from slides)

Multicore Architecture

Multiple cores share multiple caches, which increase in size from leaves to root. (L1 cache at the leaf, with each core having a separate one. L2 cache shared by

some cores. L3 is the shared memory, shared by all cores.)

Existence of caches reduces the memory access latency, because caches provide high **bandwidth** (rate of providing data to processor) data transfer to CPU. The smaller the cache the faster the access.

Idea: If we try to process an instruction at core level, we do not have to access the memory but rather just access data from the cache.

We should try to organize computations such that we fetch data from the memory less often. This can be done by reusing data previously loaded (exploiting temporal locality) or sharing data across threads (to avoid going to memory too often)

Memory Organization

Distributed memory systems: Each node is an independent unit, with its own processor, cache and memory. Node 1 is **unable** to access memory of Node 2 because the memory in a node is private due to the physically distributed memory. Data sharing is done through IPC.

Shared memory systems: Processors have a shared memory which is accessible by all processing units. Using the shared memory provider, parallel programs/threads are able to access the memory. It is important to maintain **cache coherence** and **memory consistency**.

Shared memory systems - UMA

Latency of accessing the main memory is the same for every processor.

Shared memory systems - NUMA

Physically distributed memory of all processing elements combined to form global shared memory space **(Distributed shared memory)**. But, using the shared memory provider, one processor can access the memory of another. However, accessing local memory is faster than remote memory, therefore Non-Uniform memory access time even though the memory is shared.

For ccNUMA, each node uses the cache memory to reduce contention.

COMA - Cache only memory

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

Chapter 4- Parallel Programming Models I

Parallelism is measured as the average number of units of work that can be performed in parallel per unit time.

The limits in exploiting parallelism include:

1. Control dependencies - the instructions cannot just run in any order
2. Data dependencies - Instructions need data provided by instructions executing before them
3. Memory contention - contest among threads executing on multiple cores to access memory to do work
4. Communication overhead
5. Thread/process creation overhead
6. Synchronization (waiting for other tasks to finish)

The total work time = time taken to perform tasks + time taken on dependencies

Data Parallelism

Partitioning the data used in solving the problem among processing units, which carry out similar operations on its assigned part of the data., if the operations are independent. Exploited by SIMD instructions.

Examples: executing the same instruction on different elements of an array at the same time in a for loop. If iterations are independent, they can be executed in arbitrary order and in parallel on multiple cores.

Can also be done for MIMD → i.e. scalar product on multiple processing units. The multiplications can be done in parallel by multiple PU's but at the end of the day the summations have to be done.

Task Parallelism

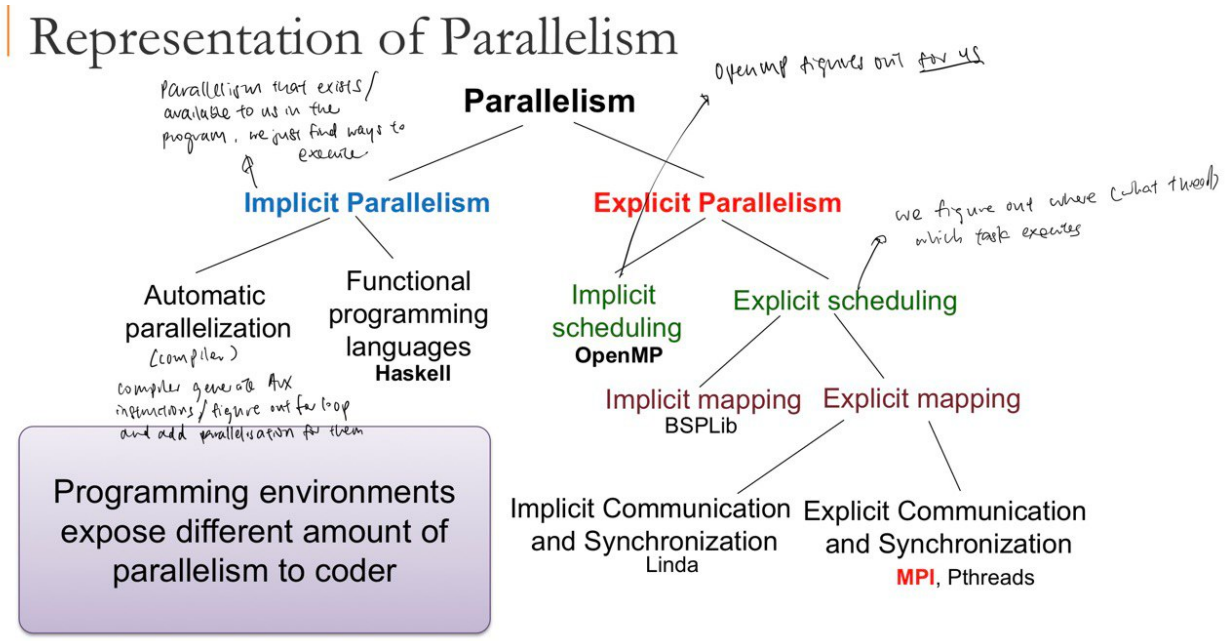
Partition tasks used in solving the problem among processing units. These tasks can potentially be further be broken down to be done in parallel among multiple processing units.

A **task dependence graph** is used to devaluate the task decomposition strategy. Nodes in the graph represent the expected execution time (not accurate since it is impossible to predict the possible overhead during execution), while edges represent control dependency. The critical path length is the longest completion time. Degree of concurrency : Total work / Critical path length.

Ideally, we want tasks equal in granularity to execute simultaneously so that cores do not idle while waiting for other dependent tasks to be ready.

Sample task: 3 TAs grading 60 scripts

- Data parallelism → all the TAs grade 20 scripts each of ALL questions
- Task parallelism → all the TAs grade 5 questions each of ALL scripts



Parallelism Overheads

These can come in the form of:

1. cost of initiating a parallel task
2. cost of management or coordination of a large amount of interprocessor/task interactions

Modes of Coordination

Shared address space: Tasks communicate by reading/writing to shared variables, while avoiding data races by ensuring mutual exclusion. However, needs hardware support to load/store to any address. We can use this on **any** memory systems, not only shared memory system! Shared AS is just simply easier to implement on shared memory since the hardware already exists

Data parallel: Map a function onto a large collection of data, without any communication between distinct function invocations. We need to ensure that we chunk data properly and do not overlap execution from different threads.

Message passing: Tasks operate within private address spaces and communicate through the means of explicit sending/receiving. This is used when there is no

shared memory within hardware that can be accessed by all cores (distributed memory systems) and is the programming model for compute clusters (Assignment3)

Any coordination mode can be implemented on any hardware, just more costly to implement (i.e. if we want to implement shared address space on distributed memory, we have to explicitly implement hardware layer that is used as shared memory provider, which will introduce overhead.

In general, it is preferable to use a model of coordination that matches the machine architecture.

Fosters Methodology

When we **partition** a program into tasks, we should follow these rules of thumb:

- at least 10x more primitive tasks than cores
- minimize redundant computations and data storage
- granularity of tasks roughly the same → minimize unnecessary waiting
- as the problem size increase, the number of tasks increase as well

Ideally, **communication** and computations should be overlapped. Computation rules of thumb:

- Same pattern of computation among tasks so we can overlap
- Task should communicate with small group of neighbors (local > global comm)
- Tasks can communicate in parallel

Task **agglomeration** → we can merge tasks into larger tasks by combining groups of similar tasks into a consolidated one. Granularity is important since it would affect the number of data transfers and the amount of work done by each processor. As much as possible, we should:

- Maintain locality

Task **mapping →** when we have more processing units, it may seem optimal just to place tasks on different processing units to increase parallelism. However, it is also optimal to place tasks that frequently communicate on the same processing unit to maximize locality and minimize inter-processor communication.

Parallel Programming Patterns

Fork-Join: a task creates child tasks that run in parallel but are independent of each other. After performing these tasks, the children join back to the parent

Parbegin-Parend: the function calls that need to be executed in parallel are specified by the programmer, and a set of threads are created, with each thread assigned statements of the construct to be executed. Typically, threads execute the same code, but in the scenario that there are if statements, execution flow might diverge.

i.e. #pragma omp parallel for

SIMD: Single instructions executed synchronously different threads on different data. Similar to parbegin-parend

SPMD: Same program on different cores but on different data. Different threads execute different parts of the parallel program due to different speeds of the cores/control statements. No implicit synchronization - explicit sync operations are required.

Master-Worker: A master controls the execution of the program and assigns work to worker threads. The master is in charge of coordination and initializations, timings, I/O operations, while worker tasks just wait for instructions from master.

Task pools: Threads access a shared pool of tasks to receive a task for execution. The number of threads is fixed, as created statically by main thread. Once a task is finished, the worker retrieves another task from the pool. Work is not pre allocated but rather retrieved by worker thread when it is free. Good when the amount of work is adaptive, but if tasks are fine grained, the overhead of task insertion and retrieval becomes relevant.

Producer-Consumer: Producer threads produce data which are used as inputs by consumer threads. Synchronization is required to ensure coordination between

producer and consumer are correct. (Shared buffers)

Pipelining: Data is partitioned into a stream of data elements that flow through pipeline stages to perform different processing stepss.

Chapter 5 - Performance of Parallel Systems

Performance can be measured on multiple viewpoints:

1. Response time → Duration of program execution is shorter (as seen by user)
2. Throughput → More work can be done in the same duration

Ideally, as users we want to minimize the response time (time between the start and termination of a program) while computer managers want to maximize throughput (the average work units executed per unit time)

There are several factors that could affect performance, including low level factors such as the programmers ability to code an algorithm to higher level factors such as the interconnection network and memory organization of the computer architecture.

Response Time

This is a summation of:

1. User CPU time, which is the time taken for execution of the program
2. System CPU time, which is the time CPU spends on executing OS routines (depends on OS implementation)
3. Waiting time: I/O waiting time, execution of other programs due to time sharing (depends on the load of the computer system)

*in an ideal state, 2&3 should be minimal

User CPU Time: Number of CPU cycles needed for all instruction * Cycle time of CPU

Time(A) = Ninst(A) * Time * CPI(A)

If we include memory access time, it would include additional clock cycles for memory access time (reading and writing)

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

Memory Access

Processor → L1 Cache → L2 Cache → Memory

Memory size > L2 size > L1 size, and L2 typically contains L1 information and more

When we encounter a cache miss, we load the required data into the cache then read from it.

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

Throughput

Represented as Million-instruction-per-second = N instructions / User Time x 10^6 or Clock Frequency / CPI x 10^6

Million-FPO-per-second = N FPO / User Time x 10^6

Drawback is that the system could be good for FP operations but not for other types of computations

Quiz: Why does some sequential code run faster on i7? A: Clock frequency (cycles/sec) is higher for i7 core as compared to Xeon core

Wrong answers:

- Compiler for Xeon generate more instructions → the machine code generated is roughly similar.
- There is too much overhead in Xeon machine because it has more logical cores → since the code is sequential it execute on one core anyway! but if parallel program then it might be true

Parallel Execution Time

This is a summation of:

1. Time to execute local computations
2. Time to exchange data between multiple processors
3. Time for synchronization between multiple processors
4. Waiting time if there are uneven distribution of workload or to access a shared data structure

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

Cp measures the total amount of work performed by all processors, and a parallel program is cost optimal if it executes the same total operations as the fastest sequential program.

Speedup : Time of best sequential program / Time of parallel program.
Theoretically, it is not possible for the speedup to exceed p, the number of processors. But in practice it may be true if the task data fits into the cache, and the sequential process needs to access memory.

Scalability

This refers to the interaction between the size of the problem and the size of the parallel computer. This is seen by how overhead/locality of data access/arithmetic intensity scales.

When we have small problem size, parallelism overheads may dominate the parallelism benefits (splitting the task may be more complex than doing the full thing).

When we have overly large problem size, key working set may not fit in small machine (disk thrashing or cache capacity exceeding)

Amdahl's Law: The speedup of parallel execution is limited by the fraction **f** of the algorithm that cannot be parallelized.

i.e. If we have a hare and a tortoise in a relay race, no matter how fast the hare is, we will always be limited by how slow the tortoise is

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

For an effective parallel algorithm, the **f** scales well with the problem size and therefore as the problem size tends towards infinity, **f** tends towards 0. Thus, Amdahl's law **can be circumvented for large problem size.**

Gustafson's Law: By increasing the problem size, we can increase the scalability. (i.e. the sequential parts stay constant but the parallel parts increase → **f** increases)

Performance Measures Summary:

Measure	Definition	Unit
Bandwidth	Maximum rate at which data can be sent	bits (bytes) per second
Byte transfer time	Time to transmit a single byte	Seconds/byte
Time of flight	Time the first bit arrived at the receiver (channel propagation delay)	second
Transmission time	Time to transmit a message	second
Transport latency	Total time to transfer a message = transmission time + time of flight	second
Sender overhead	Time of computing the checksum, appending the header, and executing the routing algorithm	second
Receiver overhead	Time of checksum comparison and generation of an acknowledgment	second
Throughput	Effective bandwidth	bits (bytes) per second

Chapter 6 - GPGPU Programming

GPGPU: General Purpose Graphics Processing Unit

GPU Architecture

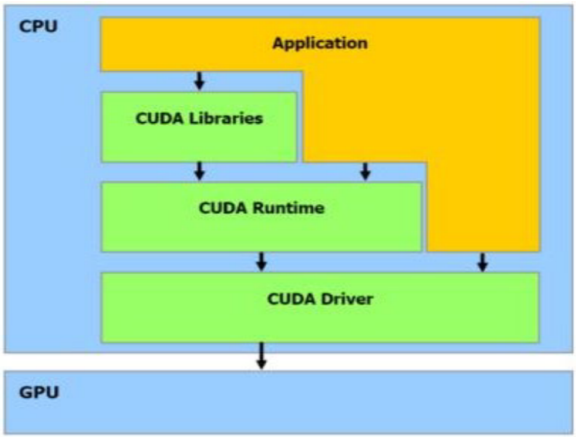
A GPU consists of multiple streaming multiprocessors. These multiprocessors have their own memory and cache, and there is a connecting interface between SMs.

Each SM consists of multiple compute cores, each with different levels of memory (registers, cache, shared memory) and corresponding logic for thread and instruction management.

CUDA: Compute Unified Device Architecture

CUDA Programming Model

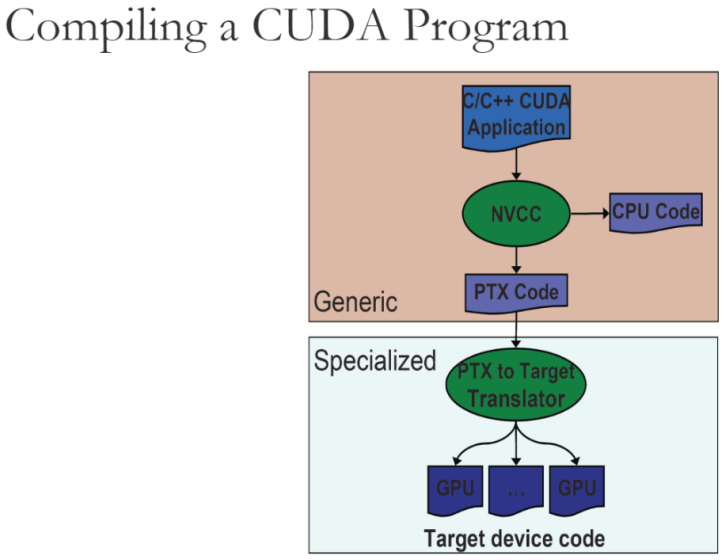
- General purpose programming model that can be used to program any type of problems
- Massively hardware multithreaded → Dedicated many-core co-processor
- User can launch batches of threads on the GPU



CUDA Runtime:
Minimal set of extensions to C, with kernels defined as C functions embedded in source code.

CUDA driver API:

API to load compiled kernels (Written in C and compiled to CUDA binary/assembly code), inspect their parameters and launch them



NVCC (nvidia compiler) works by invoking other compilers like cudacc, g++, and outputs C code (CPU Code) and PTX code which is interpreted at runtime by GPU

Although CUDA Programming allows transparent scaling to hundreds of cores and thousands of parallel threads where the workload can be divided, not every task is worth sending to the GPU to perform! Only **parallel computing intensive tasks** should be sent to GPGPUs.

Device = GPU

Host = CPU

Kernel = function that runs on the device (GPU)

The parallel tasks execute on the GPU as kernels, and one or more kernels are executed at a time.

CUDA threads are extremely lightweight - minimal creation overhead and instant context switching. Therefore, CUDA makes use of thousands of threads to achieve computational efficiency.

Each kernel is executed by an array of threads that run the same code. These threads run in '**lock step'** - they execute the same instructions at the same time and in sync. Threads are able to cooperate with each other, sharing results to save computations or sharing memory access. This would however make threads not as lightweight.

Thread Blocks

When we have an array of threads, we divide them into multiple blocks. Within each block, they have **shared memory, barrier synchronization and atomic operations.** However, threads across different blocks are not able to cooperate.

The hardware is able to schedule thread blocks to any processor (CUDA Core on any SM) at any time, and each block is able to execute in any order before any other.

Each kernel is executed by a **grid of thread blocks,** and one or more kernels can execute at a time. A grid of thread blocks can be split among SMs. Blocks are

simply a ‘virtual’ arrangement.

When a block executes on a SM, it does not migrate (i.e. starts and finishes on the same SM). Several blocks can reside at a time on one SM, with the number of blocks limited by the resources on the multiprocessor. This is because

1. Register file is partitioned among all register threads
2. Shared memory is partitioned among all resident blocks
3. Resources could be scarce if the SM is running something else, or because one block is allocated too many resources

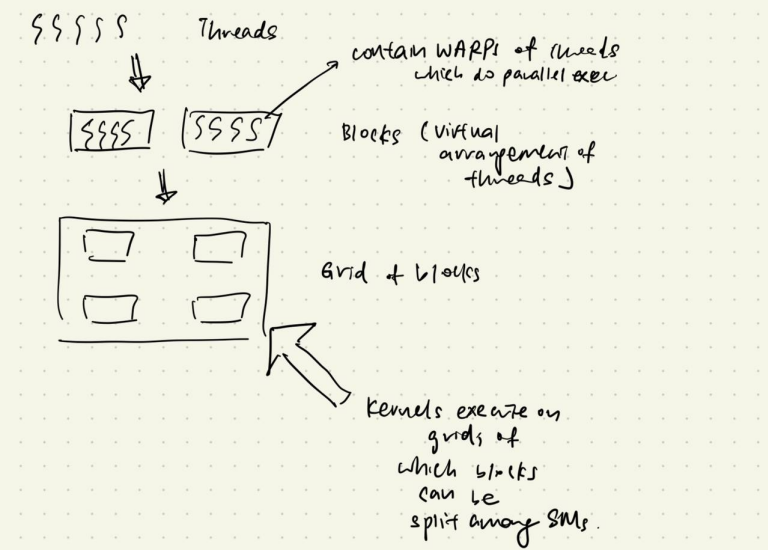
Thread Execution Mapping

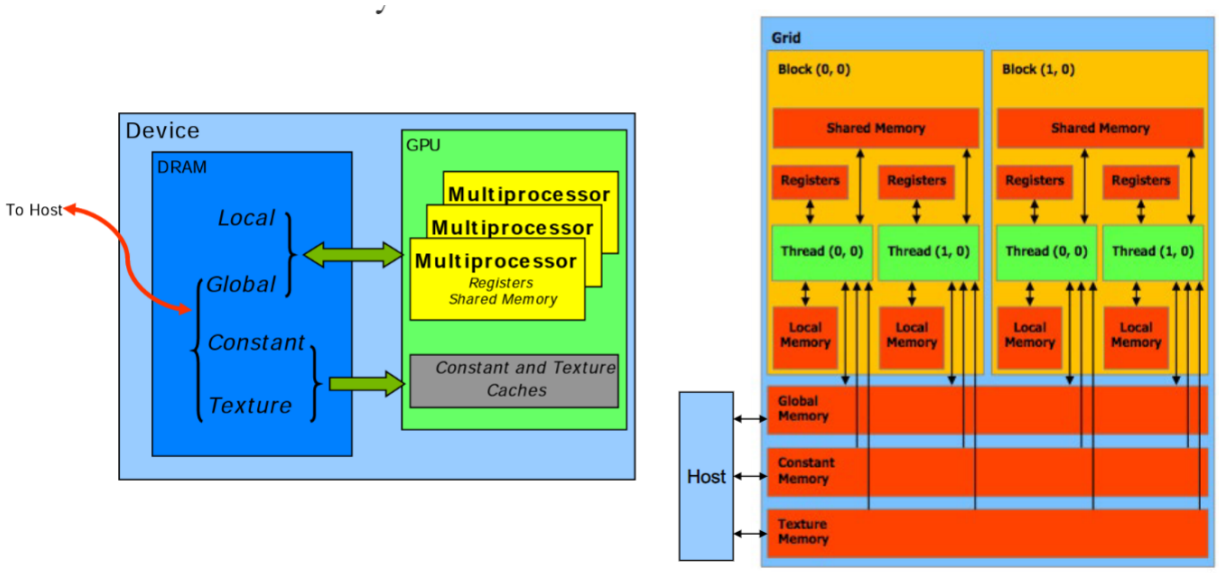
Multiprocessor creates, manages, schedules and executes threads in SIMT warps (groups of 32 threads that go for execution at the same time)

Threads in a warp start together at the same program address, but have their own program counter and register state.

A block of threads always splits into warps in the same way, with each warp consisting of threads of consisting threadIDs. Each warp executes one instruction at a time and within warps there are **no diverging control flows -** if there are if-else statements, the if takes one warp while the else takes another warp altogether.

Blocks have to be in multiples of 32 such that they can divide nicely into warps, which execute concurrently.





CUDA Memory

Data must be explicitly transferred from host to device (Assignment 2)

Global Memory is cached, while shared memory is not cached.

Local memory: Cached

Constant memory: Used for uniformly accessed read-only data, Cached

Texture memory: Used for spatially coherent random-access read-only data, Cached

Shared memory: higher bandwidth and lower latency than local/global memory.

Shared memory

Divided into equally-sized memory modules called **banks,** and successive 32bit words assigned to successive banks. Addresses from two different banks can be accessed simultaneously. But bank conflicts occur if two addresses of a memory request fall within the same memory bank - **serialized sequential access needed**.

Bank bandwidth: 32bits/clock cycle

Number of banks: 32 (same as warp size)

CUDA Optimization Strategies

(1) Optimize memory usage to achieve maximum memory bandwidth

- Different memory spaces and access patterns have different performance
- We should not do info transfer from host to device too often
- Minimize access to memory that has the lowest bandwidth (slowest access)

(2) Maximizing parallel execution

- Expose as much data parallelism as possible
- Map to hardware to increase hardware utilization

(3) Optimizing instruction usage to achieve maximum instruction throughput

- Avoid different execution path within the same warp
- High throughput arithmetic instructions

Memory Optimization Strategies

(1) Minimize data transfer between host and device

- Batch small transfers into one larger transfer

(2) Coalesce global memory access

(3) Use shared memory to minimize global memory access (higher bandwidth)

(4) Minimize bank conflict in shared memory access

Achieving concurrency in data transfer between host and device can be done with cudaMemcpyAsync() whereever possible. Use different streams to achieve concurrent copy and execute.

Optimally, there should be more warps than multiprocessor so that each SM has at least one warp that it can execute. Threads per block should be multiple of warp size and have minimally 64 threads (>1 warp per block) but its optimal to have more. Ensure threads within a warp execute the same control flow so that we can avoid memory bank conflicts.

Programming in CUDA

Device code should have no varargs, no static variable and no recursion. It can only access data loaded into the GPGPU memory.

Function types:

host: run on CPU and launched from the host

device: run on the device and launched from the device

global: run on the device but launched from the host

Launching kernels:

kernel <<dim3 grid, dim3 block, int sharedMem, int stream)

Lecture 7- Cache Coherence and Memory Consistency

Shared Address Space Model

Tasks communicate by reading/writing to shared variables

- Mutual exclusion using locking mechanism
- Requires hardware support to implement efficiently → allow any processing unit to load/store to or from any address

Problems: Memory contention overhead

Memory access recap: In order to memory access, we need to copy from memory to the cache, then access from the cache itself

Cache provides high bandwidth data transfer to CPU

Cache Properties

Tradeoff between size and access time: Larger cache increases the access time due to the increasing addressing complexity. However, it reduces the amount of cache misses since it is more likely you will be able to find your data in the cache

Cache line: Data is transferred btw main memory and cache in blocks of a fixed length. Larger blocks means the number of blocks is lesser, and increases the chance of a spatial locality cache hit but it takes longer time to replace a single block.

Case study: Matrix Multiplication

If we always have to read the column wise data in a matrix, all the required data are stored in non-contiguous memory, which would result in many cache misses each time.

Write Policy

1. Write through - write access is immediately transferred back to main memory

Advantage: The memory block always gets the newest value (constantly updated)

Disadvantage: slow down the program due to many memory accesses (which can be mitigated if we use a write buffer)

2. Write-back - write operation is performed only in the cache, and only done in main memory when the cache block is replaced (indicated by dirty bit - to signal that the cache line was written to, signaling to memory and other cache blocks that it has update, and then updated only when required)

Advantage: Less write operations to memory

Disadvantage: There could be invalid entries in the memory at any point in time

Cache Coherence Problem

There could be multiple copies of the same data that exists on different caches. When there is a local update by one core/processor on its own cache, other processors will see the unchanged data still)

Memory Coherence for Shared Address Space

Intuitively, reading a value at any address should return the last value written at that address by any processing unit. In a multiprocessor, there could be a problem with memory coherence because there are core caches as well as a global storage space.

Memory Coherence: Each processing unit has a consistent view of each memory location through its local cache. Dealing with SAME memory location, all processing units must agree on the order of read/write to that location (address).

Three properties of a coherent memory system:

- Given the sequence of operations where processing unit **P** writes to x and there are no other writes to x, if P reads from x again, it should get the value that it had written before.
- Given the sequence of operations where processing unit P1 writes to x and there are no other writes to x, if another processing unit P2 reads from x, it should get the value written by P1. (write propagation)
- Given the sequence of operations where processing unit writes v1 to x, and then writes v2 to x, the value of x can NEVER be read in the order v2 before v1. All writes must be seen in the same order (transaction serialization)

Cache line sharing status

Snooping based:

Each cache keeps track of the sharing status, and the cache monitors (snoops) on the bus, and listen to what other operations are put on the bus by other cache as well as updating its cache line whenever necessary.

Processing units can observe bus transactions (WP) and transactions visible in the same order (TS)

Directory based:

Sharing status is kept in a centralized location (so that each cache can be constantly informed about any changes to the cache line)

Implications of Cache Coherence

- Overhead in shared address space

if p1 write to x → write into memory → p2 load memory into its cache → p2 read from cache

Lowers the hit rate in cache

- Cache ping-pong

multiple PU's read/write the same one global variable value

- False sharing

The cache line is shared but not the memory location, and if P1 updates a value on the line, it will mark the line as dirty. If P2 wants another value on the line, it still needs to update the whole cache line though not actually necessary since it doesnt need to know the correct value that P1 updated.

Memory consistency: Constraint the order in which memory operations performed by one thread become visible to other threads for DIFFERENT MEMORY LOCATION

Touches on when writes to X propagate to other PUs, relative to reads/writes to other addresses by other PU

This is important because sometimes we are able to reorder the instructions of a program without changing its correctness in order to achieve better performance

Memory Operations on Multiprocessors

4 types of memory operation orderings:

1. W → R: write to X must be visible before read from Y
2. R → R: read from X must be visible before read from Y
3. R → W: read from X must be visible before write to Y
4. W → W: write to X must be visible before write to Y

Reordering in unintuitive way to hide write latencies (since write operations take longest time because we have to update both the cache and memory)

Memory Consistency Models

Sequential consistent memory system: Processing units issue their memory operations in program order, and the effect of each memory operation must be visible to all processing units before the next memory operation on any processing unit. The PUs take turns to write to one memory and every update has to be visible to all other processors at the same time. Together with the cache consistency protocol, this model becomes very slow.

Relaxed consistency memory system: If data dependencies allow, relax the ordering of memory operations. Dependencies are for two operations that access the same memory location, so we can reorder the operations for different memory locations.

This is able to hide latencies in order to gain performance (overlap independent memory access operations with other operations)

Write-to-read program order (remove W → R ordering)

TSO: Processing unit can read B before its write to A is seen by all PUs. Reads by other PUs cannot return the new value of A until the write is observed by ALL PUs.

PC: Return the value of any write (even from another processing unit) before the write is **observed** by all processing units. Write operations are eventually observed by all PUs, and all writes to the same memory location are observed by all PUs in the same order, but the writes can be read by one PU before they are observed by all.

Write-to-write program order

PSO: Not only isit like TSO where the W → R order is relaxed, but the W → W order is similarly relaxed as well.

Lecture 8 - Performance Instrumentation

Aspects of instrumentation:

- Scalability - ensuring that response time does not increase as work load increase
- Reliability
- Resource usage → Efficient utilization of resources

Timelines:

- Not time sensitive: Testing before release
- Time sensitive: Incident performance response

Overview: Approach for analyzing the performance of programs, by looking at performance bottlenecks and adjusting code to deal with normal/stressful scenarios

Workload: Input to the system or load applied

Latency: Measure of time that an operation spends waiting to be serviced

Response time: Time for an operation to complete, including latency and service time, and including time to transfer the results

Throughput: The rate of work performed (bytes/second)

Utilization: For resources that service requests, utilization is a measure of how busy a resource is (based on how much time in an interval that work was actively being performed)

Saturation: Degree to which a resource has queued work beyond what it is able to service

Bottleneck: Resource that limits the performance of a system. Identifying bottlenecks and removing them is a key to improving systemic performance.

Knee point: The place where the delay/response time starts increasing beyond control → after knee point, the delay increase is faster. We should ideally make sure system stays before the knee point so that the throughput keeps increasing.

From Amdahl's law, we know that many times, in order to improve the parallel performance of any program, we need to improve the parallel performance of our sequential code.

Resource analysis: High focus on utilization (supply-demand) and making sure that resources should not be underutilized.

Workload analysis: Examination of workload applied and how the application responds → measured by throughput and latency

Type	Characteristic
Observability	Watch activity under workload. Safe, usually, depending on resource overhead. -insert timing statements -check performance counters
Static	Examine attributes of the system at rest rather than under active workload. Should be safe.
Benchmarking	Load test. Caution: production tests can cause issues due to contention.
Tuning	Change default settings. Danger: changes could hurt performance, now or later with load.

Lecture 9 - Parallel Programming Models II

Parallel computing problems typically based on array of various dimensions, and it is useful to study how we decompose the arrays to distribute the work on multiple processors (data distribution/work distribution/decomposition/partitioning)

Data distribution for 1D Array

Given p identical processors and n elements (with n > p), there are 2 patterns of distribution:

- Blockwise → Split into blocks of size ceiling(n/p)
- Cyclic → Every processor is allocated one item, until we reach the last processor then we go back to the first one and give it one item again.

Which technique is better?

Blockwise maintains **data locality** since array items that are side by side will get put into the same processor. Cyclic allocates chunks of (almost) equal size and there are **minimal imbalances** observed. We need to identify which we prioritize - locality or workload balance. Also, sometimes we are not sure how much

workload n we need to process from the start → we are unable to use blockwise effectively since we may not know the size of blocks to split into.

Data distribution for 2D Array

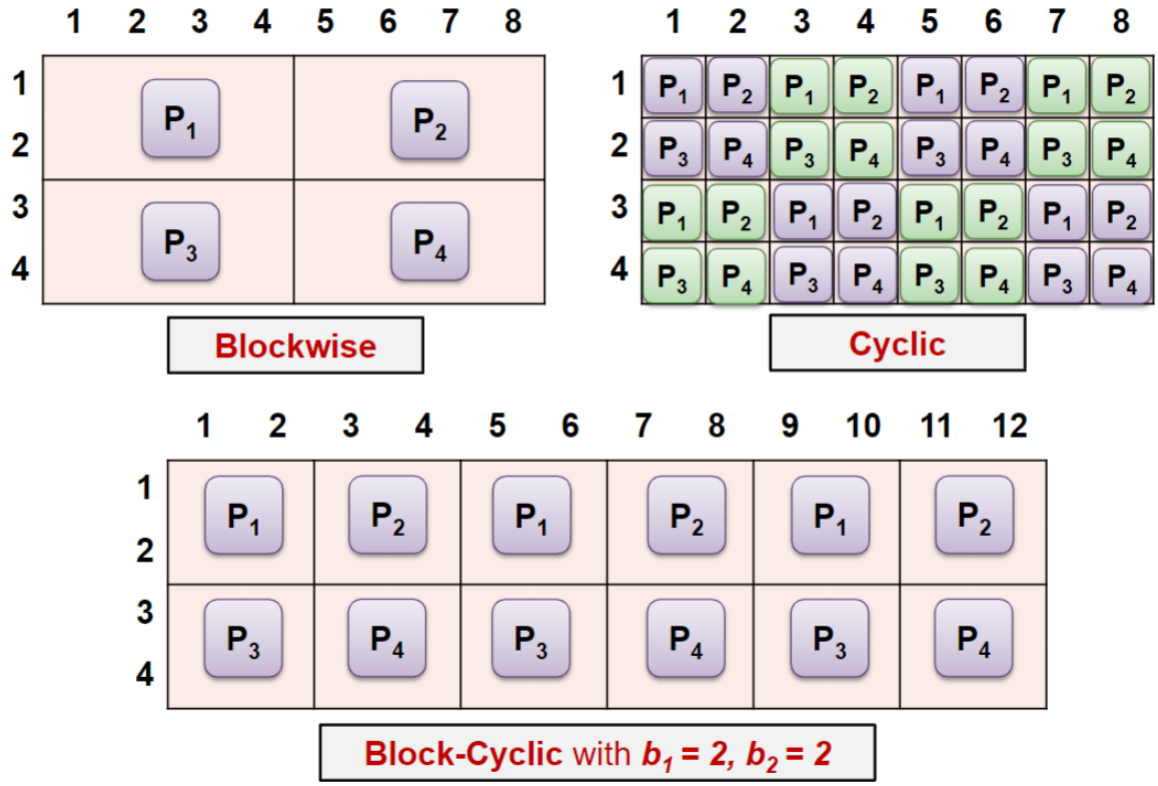
Blockwise distribution on column dimension

Cyclic distribution on column dimension

Block-cyclic combination → we form blocks of size **b** then we perform round robin (cyclic) allocation. This comes with the advantage that we are able to roughly get an even distribution while maintaining data locality → the advantages of both data distribution techniques. Programmer can specify the block size and has control over the granularity of the tasks.

Two dimension distributions

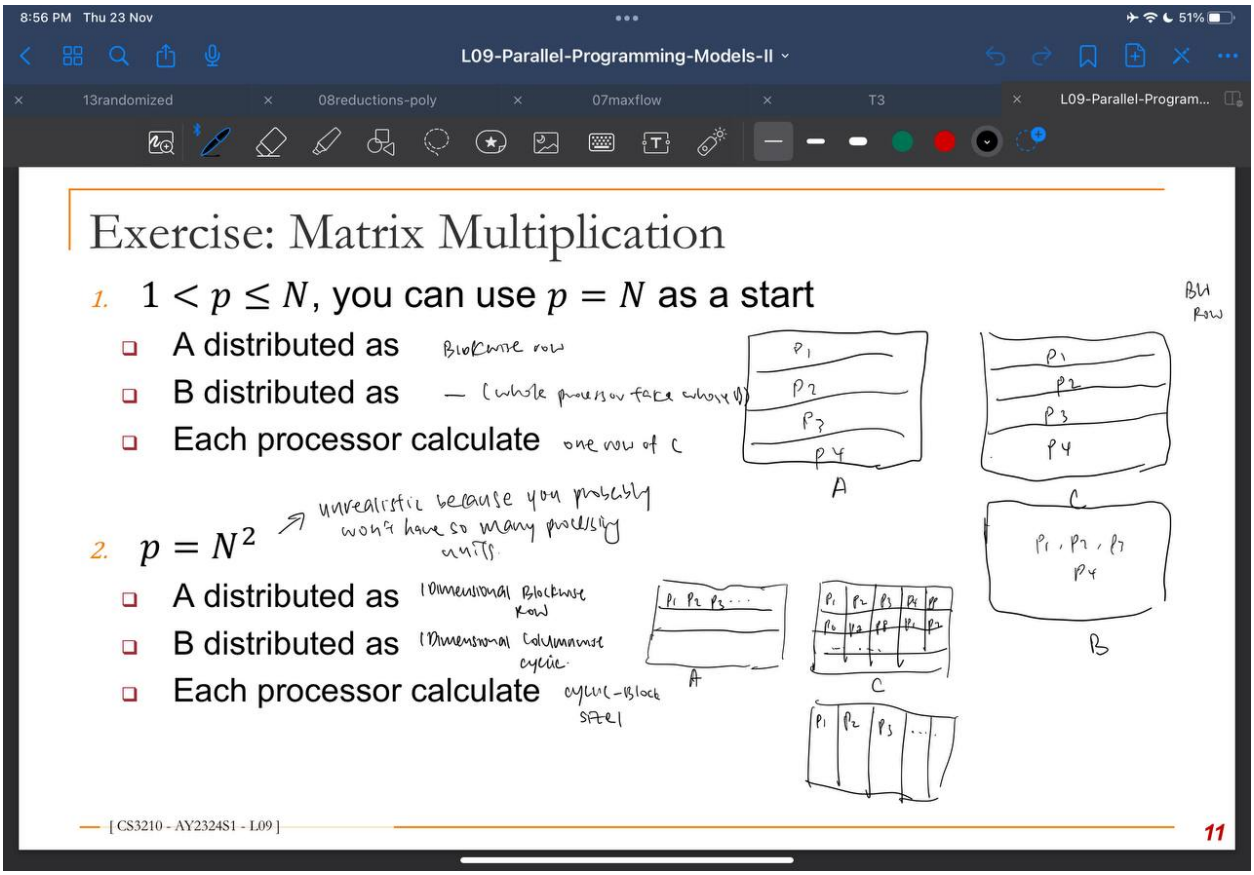
When processors are virtually organized into a 2D mesh of Row x Columns (checkerboard arrangement), based on the arrangement some processors communicate faster/slower with others depending on the number of ‘hops’.

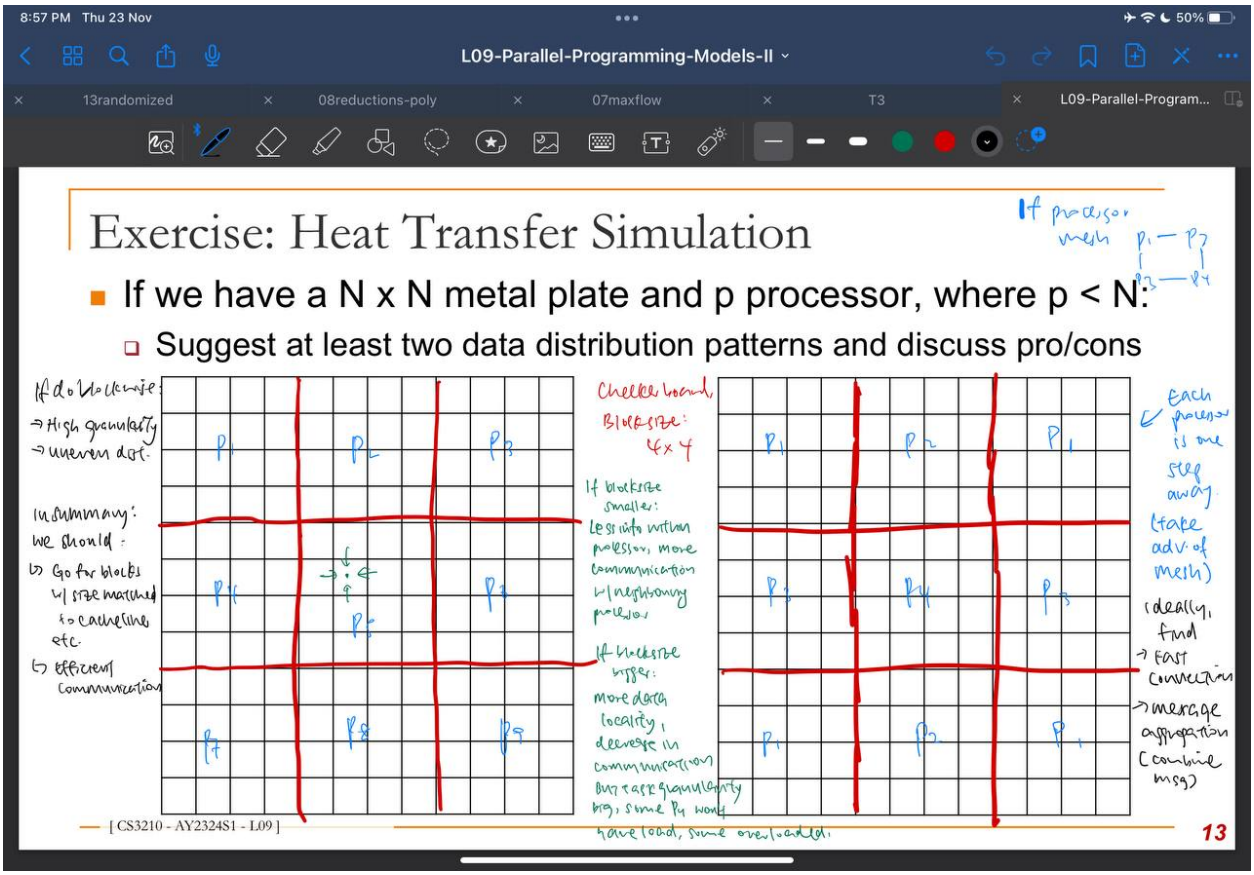


Blockwise: Split elements into blocks along both dimensions according to R and C

Cyclic: According to processor mesh arrangement, do cyclic assignment

Block-cyclic: Elements are split into r1 x c1 sized blocks then cyclical assignment is done to processors based on the processor mesh arrangement. The programmer can choose r1/c1.


8:56 PM Thu 23 Nov
51%
L09-Parallel-Programming-Models-II
13randomized
08reductions-poly
07maxflow
T3
L09-Parallel-Program...
Exercise: Matrix Multiplication
1. $1 < p \leq N$, you can use $p = N$ as a start
A distributed as Blockwise row
B distributed as — (whole processor take whole)
Each processor calculate one row of C
2. $p = N^2$
unrealistic because you probably won't have so many processing units.
A distributed as 1Dimensional Blockwise Row
B distributed as 1Dimensional Columnwise cyclic
Each processor calculate cyclic-Block size 1
[CS3210 - AY2324S1 - L09]
11



First pattern → Blockwise

If we do blockwise, there will be high task granularity and (likely) uneven distribution

To counter this, we can potentially:

1. Decrease blocksize. Task granularity will be smaller, since less information is written to processor. Likelihood of an even distribution increase. However, more communication with neighbouring processor is needed (incurring communication overhead)
2. Increase blocksize. Task granularity is big, but less communication overhead since there is more data locality. More likelihood of uneven distribution → some processors have less workload while some are overloaded

To optimize this pattern, we need to:

- Go for blocksize with size that match cache line

- Efficient communication techniques

Second pattern → Assuming processor is in a mesh

Each processor that it needs data from is a single hop away in the actual mesh. In order to optimize this even farther, we need:

- Fast connection
- Message aggregation (when requesting data from neighbouring node, we can combine requests so less total requests → less overhead)

Information Exchange

Purpose: Necessary for controlling the coordination of different parts of a parallel program execution.

Recap: Shared address space → shared variables; Distributed address space → communication operations

Shared address space

Assumes a global memory accessible by all processors → information exchange through shared variables, with synchronization needed for safe concurrent access

Data races (multiple threads accessing the same shared variable) could lead to non-deterministic behavior, but can be avoided through critical section mechanism

Distributed address space

Exchange of data between processors done through dedicated communication operations, known as the message passing programming model

Two main types of data exchange:

1. Point to point communication (one-one communication between one processor and another)

2. Global communication (one-all communication between one processor and all other processors)

Message passing model

Data is explicitly partitioned for each process, and all interaction requires both parties to participate.

Loosely synchronous paradigm: Tasks or subsets synchronize to perform interactions but between these interactions, tasks execute completely asynchronously. (Programs can execute anything independent of each other - we do not need tasks to execute in lockstep)

Communication Protocol

Fundamentally, the value received should be equivalent to the value sent.

Buffered point to point communication: One way transfer done over the interconnection network. The message is stored in the output buffer in the source node, then sent over the ICN to the input buffer of the destination node).

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

Problem with Non-Buffered Blocking operations: Idling and deadlocks

(a) If sender comes first, it has to wait for the corresponding receive message and idles meanwhile

(b) If receiver is ready first, it has to (idle) while the corresponding send operation is not ready yet.

Buffering will make the sender copy the data to be sent into the designated buffer and return after the copy operation is complete. The receiver receives the data into its buffer as well. This strategy trades off idling overhead for buffer copying overhead.

In addition, the CPU needs to send the buffer data over the network before the data can be retrieved into the receive buffer.

If the consumer is much slower than the producer, we need some sort of buffer management to sort out the time difference. Deadlocks can also occur if both processors call receive before send since the receive is a blocking operation.

Non Blocking operations have the send/receive returning before it is semantically safe to use the transferred data, so such operations are typically paired with a **check status** operation. When used correctly, these primitives are capable overlapping communication overhead with useful computation operations.

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

Sync: Send completes after matching receive and source data send, receive completes after data transfer complete from matching send

Async: Send completes after the input buffer may be reused

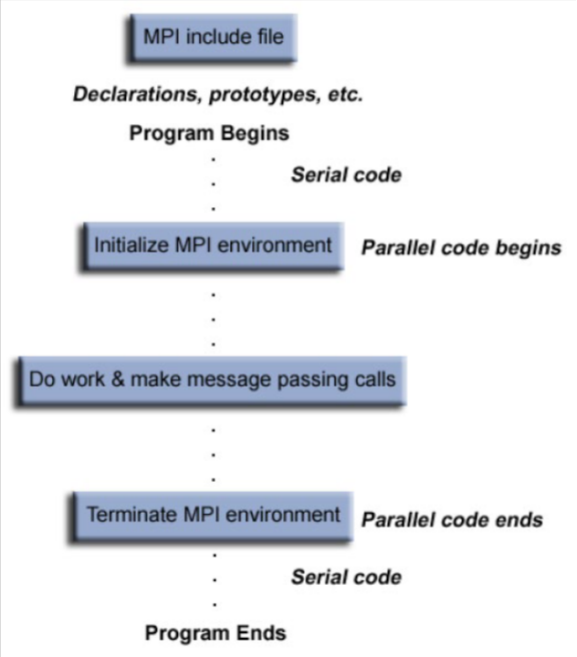
Blocking =/= Synchronous and Non-blocking =/= Asynchronous

Chapter 10 - Message Passing Programming

Abstraction of a parallel computer with distributed address space, with processes exchanging data using explicit messages (communication operations)

MPI

MPI is not just a library/set of API calls that allow send and receive operations, btu rather a standardization of what we should provide in a message-passing library



Initialization of MPI Call

Before any other MPI routines are called, we need to call MPI_Init to initialize the MPI program

int MPI_Init(int* argc, char** argv[])

Basic MPI Functions to write a program

MPI_Comm_size(MPI_Comm comm, int* size)→ size of the group of processes communicating

MPI_Comm_rank(MPI_Comm comm, int* rank) → to get the index of the process within the communicator (based on different rank, can get different process to execute different code)

MPI_Send (Blocking)

MPI_Recv (Blocking)

MPI_Isend (Non-Blocking)

MPI_Irecv (Non-Blocking)

MPI_Barrier (used for a synchronization point → processes all block and wait until every single process in the specified communicator has reached the barrier point)

Blocking and non-blocking operations can be mixed → Isend can be received by normal recv and Send can be received by Irecv.

Message received must be less than or equal to the length of the receive buffer.

Termination of MPI Call

After all MPI processes are completed, we need to call MPI_Finalize to terminate all MPI processing

int MPI_Finalize(void)

Force Stopping MPI Process

If for any reason we need to force all processes to terminate, we use MPI_Abort to terminate them, and return the error code to mpirun

int MPI_Abort(MPI_Comm comm, int errorCode)

Each MPI Message Comes with Data and an Envelope

Data includes:

1. Start buffer (data address)
2. Count (number of elements within the data message)
3. Datatype

Envelope includes:

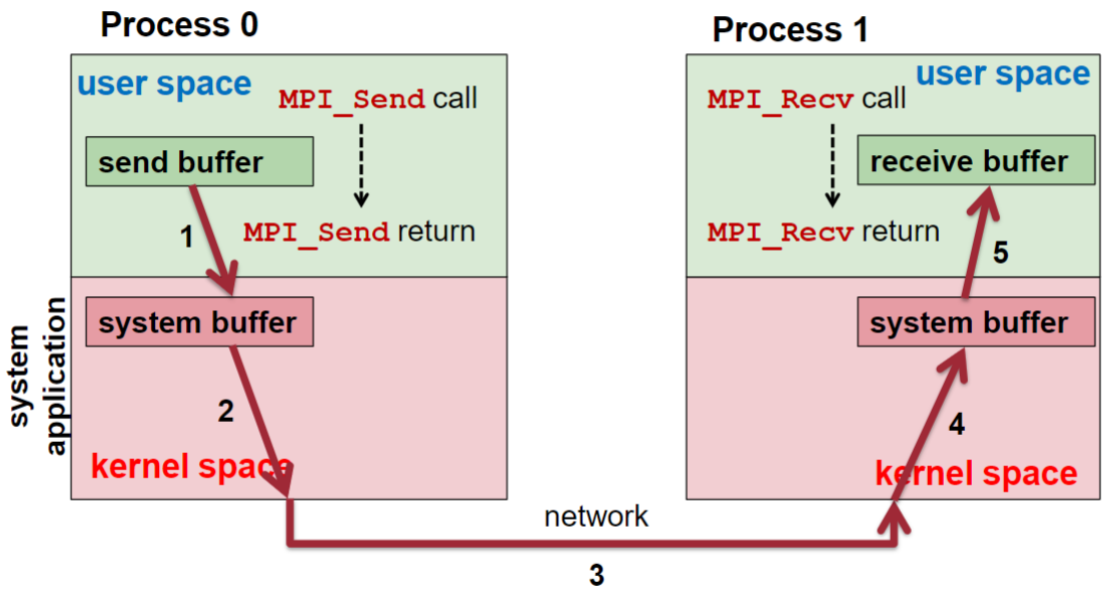
1. Destination/source rank
2. Tag (used to categorize message)
3. Communicator (Group of processes between which communication occurs → rank refers to the index within the communicator)

	Synchronous	Asynchronous

Blocking	`MPI_SSend`
	`(MPI_Mrecv)`
	`MPI_RSend`
		`May be buffered:`
		`MPI_Send`
		`MPI_Recv`
		`Buffered:`
		`MPI_Bsend`

Non-blocking	`MPI_ISSend` `(MPI_ImRecv)`	`MPI_ISend` `MPI_Irecv`

Sample MPI Send Receive (Take Process 0 to be on node 0 and Process 1 to be on node 1)



(1) Send buffer is in the local space of P0. Since a blocking send call is used, the send operation waits until the send buffer is completely copied into the system buffer before returning. Since in this scenario we see that the return is initiated before the P1 matching receive is started, it is an asynchronous operation.

(2) The system buffer is transferred over the network from the node 0 kernel space to the node 1 kernel space

(3) Kernel helps the MPI create a message sent over the network to process 1 [TCP Message] → this is only required because the P0 and P1 are on separate nodes. If they were on the same node, there is no need for message transfer over the network.

(4) The system buffer of node 1 in its kernel space receives the message in its system buffer.

(5) The message is copied into the receive buffer of P1. Since a blocking receive is used, the receive operation waits until the system buffer is completely copied into the receive buffer before returning.

Order of receive operations

If there are two processes, P0 and P1, and P0 sends 2 messages over to P1, the order of message delivery is always the order in which they have been sent.

However, if there are more than two processes doing the communications then we are unable to determine the ordering with certainty.

i.e. Imagine the scenario if:

(1) P0 sends M1 to P1 and M2 to P2.

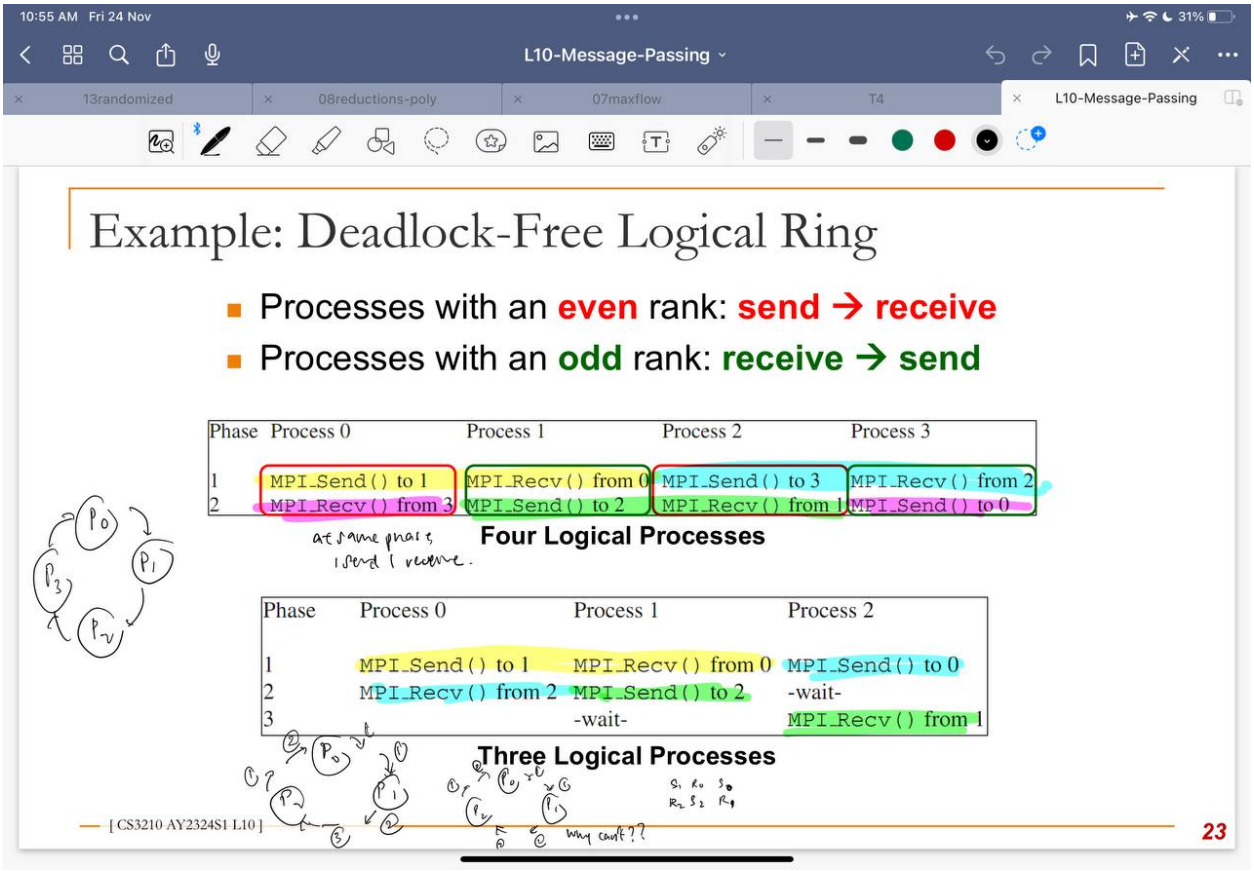
(2) P1 receives M1 from P0, then sends M1 to P2.

We are unable to determine the exact order of which P2 receives the messages!

Deadlocks in MPI

- If between two processes, both of them call receive before send, then a deadlock occurs because they will (infinitely) wait on each other.
- If the runtime system does not use system buffer or if the system buffer used is too small, if two processes both call send before receive, it is possible that there is deadlock. This is because the (blocking) send operation only return after the send buffer is safe to reuse (i.e. when it is copied into the system buffer) and if the data is unable to be copied from the send buffer to the system buffer (because there is no buffer/insufficient space) , the send will be unable to complete.

- If a MPI program's correctness does not depend on assumptions about properties of the MPI runtime system, it is known as **secure**. If we do send-recv/recv-send, we do not have to assume that the system buffer has sufficient space for both send operations in order for the program to be correct and not deadlock.
- If there are more processes, we need a deadlock free logical ring.



At each phase, for each send there is a matching receive. For odd number, when P2 send to P0, P0 can only receive on the next phase. Therefore, it wait until the corresponding message is received by P0 (so it waits on phase 2).

Process Groups

Refer to an ordered set of processes where each process within the group has a unique rank. A process may be a member of multiple groups and hold different ranks in each of the groups.

Communicator

Communicator refers to the communication domain for a group of processes.

- Intra-communicators facilitate the execution of communication operations within a certain group
- Inter-communicators facilitate the point to point communications between two separate groups

Groups and Communicators allow us to stay organized → organizing tasks based on their function into separate task groups, and enabling communication operations across a subset of related tasks (i.e. producers grouped into one group while consumers grouped into another group. such grouping enables messages such as 'buffer space available' to send to ONLY those who need to be aware of them.)

Virtual process topologies are possible → Process topologies can be created where the neighbors with which most communication operations are done are most easily addressable.

Gather operation: Message passed from **many processes to one process**

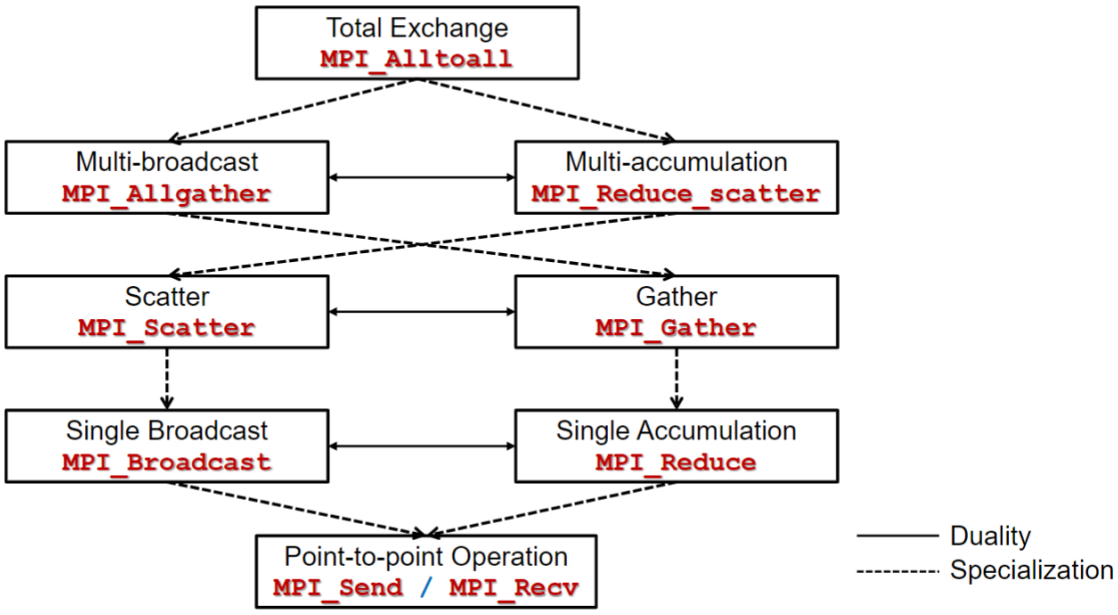
Scatter operation: Message passed from **one process to many processes**

Communication operations

- Single transfer (point → point) : Send/Recv
- Scatter gather (one point → all/ all → one point). Each send takes 1 stage! (not simultaneous and cannot do parallel send to different processor)
- Single broadcast: Single processor (root) send the same data block to all the other processor

- Multi broadcast: Each processor sends the same data block to all other processes, and these data blocks are collected in rank order (Rank 1 collects from 2, then 3 then 4)
- Single Accumulation: Gather with reduction → Each processor provides a block of data with the same type and size, then the root processor gathers all of them and applies a reduction (binary, associative and commutative operation → +-x/) and stores result in root
- Multi Accumulation: Every processor sends to all other data blocks (potentially different unlike broadcast) and receives from every other block, applying the same reduction operation on all data blocks it gets.
- Total Exchange: Like Multi Accumulation but without the reduction operation → each processor just send to all, then collect and store all it receive.

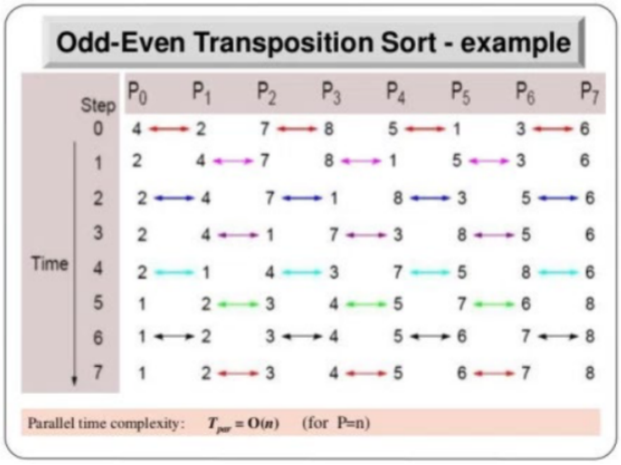
When we have communication operations, we can put it in a spanning tree and if 2 communication operations have the same spanning tree they are known as a duality.



Chapter 11 - Interconnections

The interconnection network forms the backbone of communication between processors, processor and memories, processor and caches and I/O devices.

Sort ***N*** numbers on ***N***-PEs Linear Array



stions:

- At each step, we perform a comparison and swap in parallel, for a total of n rounds. This O(1) operations, and the total n rounds means that this works better than the fastest sort (merge sort) which is in O(nlogn).
- Although this seems ideal, keep in mind that it is unrealistic to assume that we have N processors to process a data of length N!

In a processor mesh, each corner processor has two links, each edge processor has 3 links while those in the middle have 4 links. If we wrap around from left to write and from top to bottom, then every processor element has 4 links. This is also known as a **Torus** arrangement.

Shear sorting phases:

(1) sort odd rows in ascending order and even rows in descending order

(2) sort columns in ascending order from top to bottom

Repeat 1&2 until the numbers are sorted, into a snake like arrangement

For N numbers, we need Log2N + 1 phases, and at each phase we do odd-even transposition sort where every processor takes one row/column with sqrt(N) elements. Therefore, time complexity is O(sqrt(N) * (Log2N + 1))

Interconnection Network Impact

- System scalability - size and extensibility (what happens when we try to add new devices)
- System performance and energy efficiency - judged based on speed of communications, memory access latency and energy spent transferring data (if hardware interconnects are connected too heavily, transform into hardware overload that could overheat devices)

Interconnect factors - Topology

Direct interconnection: Known as static or point-to-point connection, as its name suggests, the switches are at the endpoint that is interconnected, and usually the endpoints are of the same type (core, memory)

Indirect interconnection: Known as dynamic connection, interconnect are formed by switches within a interconnect network, that can be configured.

Diameter: Maximum distance between any pair of nodes - minimizing diameter ensures small distances for message transmission

Degree: Number of direct neighbor nodes of v - better to maintain as small as possible since it lowers the hardware overhead (w high hardware overhead, cannot scale easily since we cannot infinitely overload the hardware → eventually will hit maximum overheating)

Bisection width: Minimum number of edges that must be removed to divide the network into 2 equal halves - a good measure of the capacity of a network when transmitting messages simultaneously (think about 3230)

Node Connectivity: Minimum number of nodes that must fail to disconnect the network - or robustness, better to maintain as high as possible

Edge Connectivity: Minimum number of edges that must fail to disconnect the network - to determine the number of independent paths between any two nodes, better to maintain high since high EC → many independent paths between two 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 hyper-cube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k-dimensional CCC-network ($n = k2^k$ for $k \geq 3$)	3	$2k-1+\lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
complete binary tree ($n = 2^k - 1$)	3	$2\log\frac{n+1}{2}$	1	1
k-ary d-cube ($n = k^d$)	$2d$	$d\lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

As we can observe, for binary tree, it is not the most ideal from a connectivity perspective since all it takes is one edge to fail before the tree is disconnected. However, a direct fix to this is to use a FAT tree instead → more edges between any 2 nodes depending on level, can be 2, 4, 8 edges to make it more “failproof”

For a hypercube, it has good interconnect since the degree of the node is kept constant. With a cube connected cycle, as the network increases the degree of a node stays as constant as possible.

Indirect Interconnection

By using sharing switches and links, we can reduce hardware costs, since switches provide indirect connection between nodes and are able to be configured dynamically → useful for concurrent connections.

Bus network: a set of wires to transport data from a sender to a receiver. Only one pair of devices can communicate at once, and this is typically used for a small number of processors. This does not scale very well if many processors try to communicate at the same time - they share the bus so messages will not arrive on time at the destination if many messages are sent simultaneously.

Crossbar network: Number of switches = n (Number of input processors) * m (Number of output processors). Every switch has two states → straight or direction change , and this hardware is costly since there are many switches for a small number of processors. There are in face more switches than direct interconnect!

Multistage switching network: There are 4 different switch settings with the goal of obtaining the smallest distance for arbitrary pairs of input and output devices.

Omega Network

One unique path from every input to output → a n x n network has logn stages with n/2 switches per stage

Each switch has 2 connections to switches in the next stage. If A is the position of the first switch, then it connects to A leftshift 1 and A leftshift 1 invert lastbit

Butterfly network

One unique path from every input and output, same number of switch and stage as omega

Each switch has 2 connections → If A is the position of the first switch, then it connects to A and A invert (nextstagenumber th) bit from left

Baseline network

Similar to the other two, each switch connect to 2 others. If A is the position of the first switch, then it connects to A cyclic right shift of last (k - stagenumber) bits, A invert LSB and right shift of last (k - stagenumber) bits.

Interconnect factors - Routing

Algorithms classification:

1. Based on path length (minimal = shortest path)
2. Based on adaptivity (deterministic = always same path for the same pair of source, destination vs adaptive = take into account the network status and adapt, may avoid congested paths or dead nodes → perform better but slightly more computation overhead)

Examples:

1. X-Y routing: always move horizontal then vertical
2. E-Cube routing for hypercube: Number of hops: number of bit difference → either from MSB to LSB or vice versa, find the first different bit and move to the corrected bit node neighbor.
3. XOR tag routing for Omega Network: XOR the sourceID and destinationID, then move straight for bit 0 and crossover if bit 1
4.