# Lecture Summary

**Note** This document combines the Midterm I & II documents.

The contents are structured according to the lecture slides. Some of the texts are 1:1 copies from the lecture slides available at http://ait.inf.ethz.ch/teaching/courses/2014-SS-Parallel-Programming/. For the sake of readability and quicker typing, those excerpts are often simply in double quotes. And sometimes they are simply paraphrased. All credit goes to Prof. Dr. O. Hilliges and Dr. K. Kourtis.

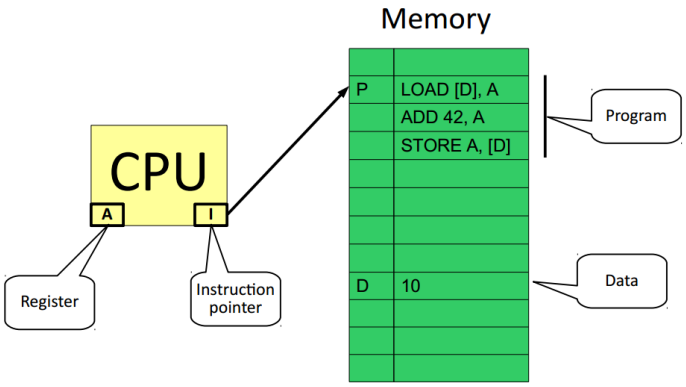
## Contents

# 1 Course Overview

- Even though Moore's Law[1] is still valid, heat and power consumption are of primary concern. These challenges can be overcome with smaller and more efficient processors or simply more processors. To make better use of the added computation power, parallelism is used.
- Parallel vs. concurrent: (quoted); in both cases, one of the difficulties is to actually determine which processes can overlap and which can't
  - Concurrent: focus on which activities may be executed at the same time (= overlapping execution)
  - Parallel: overlapping execution on a real system with constraints imposed by the execution platform
- Parallel/concurrent vs. distributed: In addition to parallelism/concurrency, systems can actually be physically distributed (e.g. BOINC).
- Concerns in PP: (quoted)
  - Expressing parallelism
  - Managing state (data)
  - Controlling/coordinating: parallel tasks and data

# 2 Parallel Architectures

- **Turing machine**: (quoted) infinite tape, head for r/w, state register
- Computers today: consist of CPU, memory and I/O; "stored program" i.e. program instructions are stored in memory and program data, too (von Neumann)
- Since access memory became slower than accessing CPU registers, CPUs now have caches which are closer (and thus faster but also smaller) to the CPU (locality of reference → principle of locality); L1, L2, L3, … cache

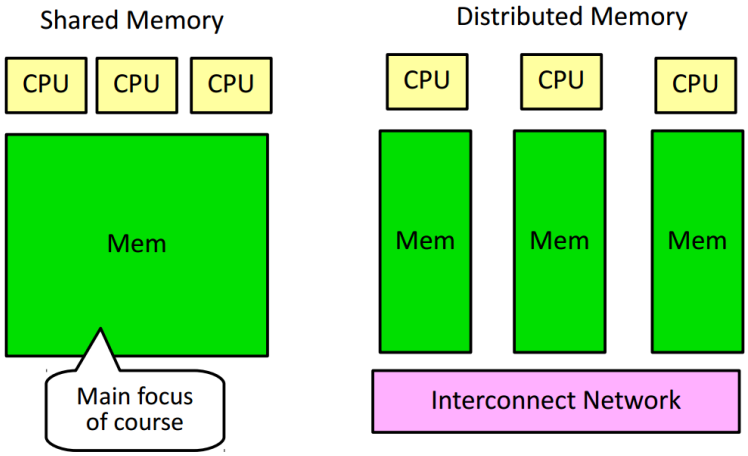



- "Applying parallelism to improve sequential processor performance: vectorization[2], pipelining[3], Instruction Level Parallelism (ILP)"
- Pipelining: There is a lead in and a lead out where system is warming up/cooling down (resp.) and full utilization (which is to be maximized). On a CPU:


- **ILP**: "Pipelining; Superscalar CPUs: Multiple instructions per cycle / multiple functional units; Out-of-Order (OoO) execution: Potentially change execution order of instructions, as long as the programmer observes the sequential program order; Speculative execution: predict results to continue execution"

[1] Actually an observation; "The number of transistors on integrated circuits doubles approximately every two years"
[2] Applying an operation on every element of a vector in parallel instead of sequentially; instead of 1-at-a-taime, N-at-a-time
[3] Think laundry

- For a long time, Moore's Law and ILP made sequential programs exponentially faster but due to power dissipation (expensive to cool CPUs), CPUs becoming faster than memory access and not being able to ILP a program anymore, it was "no longer affordable to increase sequential CPU performance". The solution was multicore processors which, however, first needs programmers to write programs which can actually take advantage of the new hardware.




- "Shared memory architectures: SMT (Intel's Hyperthreading; Simultaneous MultiThreading), Multicores, SMP, NUMA"
- SMT: "single core, multiple threads, ILP vs. multicore: ILP multiple units for one thread, SMT multiple units for multiple threads"
- CMP/multicores: "dual, quad, x8 etc. ; each has its own hardware, yet might share part of the cache hierarchy"
- SMP: "multiple chips (CPUs) on the same system: CPUs share memory – same cost to access memory; CPU caches coordinate – cache coherence protocol"
- NUMA/Non-uniform memory access: memory is distributed (local/remote) at the cost of speed, shared memory interface
- Distributed memory[4]: organized in clusters
- Flynn's taxonomy: [S|M]I[S|M]D where S = single, M =multi, I = instruction, D = data; used to classify different types of architectures
- GPUs are badass! (and they're great for data parallelism)

# 3 Basic Concepts

- Performance in sequential execution: computational complexity $O$, $\Theta$ and execution time
- Sequential programs are much easier to write, yet parallel programming has better performance
- **Parallel performance formulae**:

  $T_1$: sequential execution time, $T_p$: execution time on $p$ CPUs

  $T_p > \frac{T_1}{p}$, performance loss, normal; $T_p = \frac{T_1}{p}$, perfection; $T_p < \frac{T_1}{p}$, sorcery

  $S_p$: speedup on $p$ CPUs; $S_p = \frac{T_1}{T_p}$

  $S_p = p$, linear speedup, perfection; $S_p < p$, sublinear speedup, performance loss

  $S_p > p$, superlinear speedup, sorcery

  Efficiency: $\frac{S_p}{p}$

  Amdahl's Law[5] ($b$ = sequential part, $1 - b$ = parallel part):

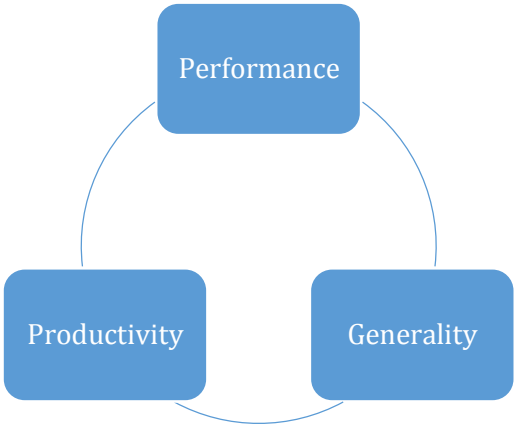
  $T_p = T_1 \cdot \left(b + \frac{1-b}{p}\right), S_p = \frac{p}{1 + b \cdot (p-1)}$

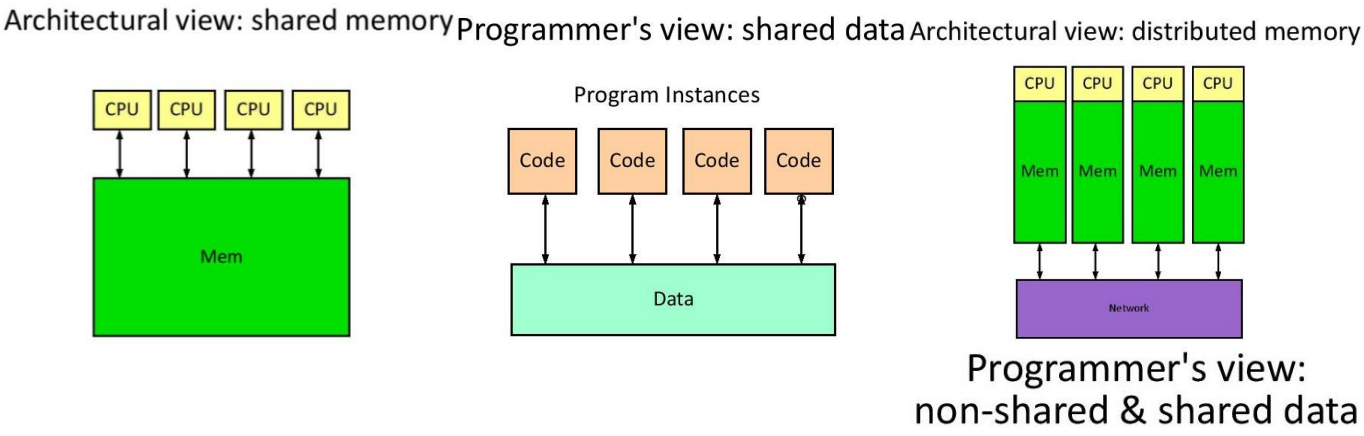
  Gustafson's Law ($b$ = sequential part): $T_1 = p(1-b) \cdot T_p + b \cdot T_p, S_p = p - b(p-1)$

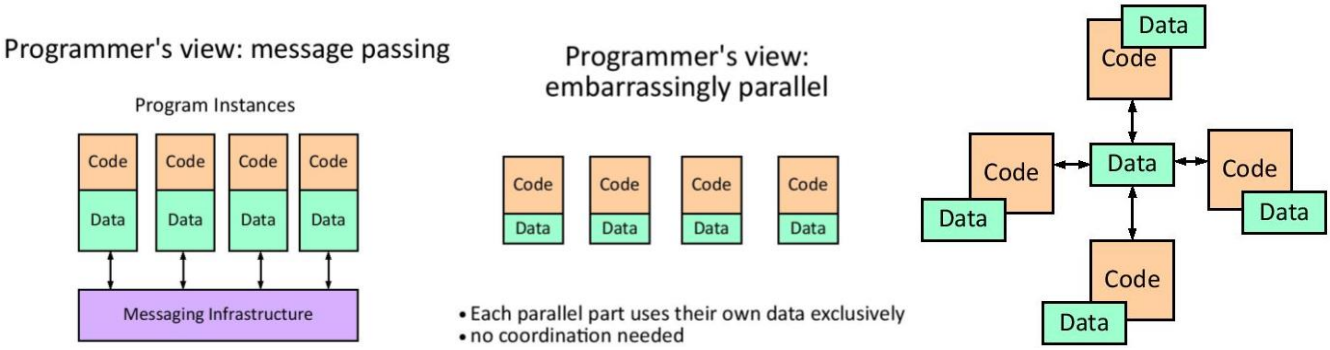
[4] See Top500

[5] Very optimistic approach, Gustafson was more realistic: it considers problem size, runtime (and not problem size) is constant, more process can solve larger problems in the same time, parallel part scales with the problem size

- **Scalability**: how well a system reacts to increased load; in PP: speedup with more processors (even to ∞), linear speedup is desirable
- Reasons for performance loss: program may not contain enough parallelism, overhead (due to pp), architectural limitations (think group work/presentation)
- Concurrency vs. parallelism: "Concurrency is: a programming model, programming via independently executing tasks, about structuring a program, example: network server, a concurrent program does not have to be parallel; Parallelism is about execution, concurrent programming is suitable for parallelism"
- Code and data – code doesn't change over time while data does




- Expressing parallelism: Work partitioning (splitting up work of a single program into parallel tasks) which can be done manually (task parallelism; user explicitly expresses tasks) or automatically by the system (data parallelism; user expresses and operation and the system) (quoted)
- **Work partitioning & scheduling** (quoted): work partitioning: split up work into parallel tasks, (done by user or system), a task is a unit of work, also called: task decomposition; scheduling: assign tasks to processors, (typically done by the system), goal: full utilization (no processor is ever idle)
- **Coarse vs. fine [task] granularity**: fine granularity is more portable and better for scheduling, parallel slackness[6], but overhead may grow (too) big
- Coordinating tasks: enforcing a certain order since e.g. task X needs the result of/has to wait for task A to finish; example primitives: barrier, send(), receive()
- **Managing state concerns**: shared vs. distributed memory architectures; which parallel task access which data and how (r/w); potentially split up data; task, then data or data, then tasks (quoted)

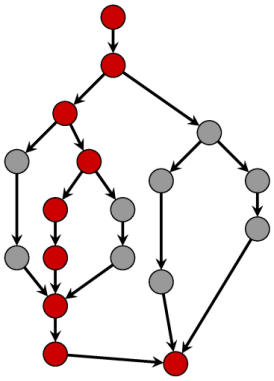
[6] expressed parallelism ≫ machine parallelism

- Coordinating data access (quoted): distributed data: no coordination (e.g., embarrassingly parallel), messages; shared data: controlling concurrent access, concurrent access may cause inconsistencies, mutual exclusion to ensure data consistency

# 4 Parallel Models

- PP is not uniform, many different approaches (!)
- PP paradigms (quoted): task parallel: Task Parallel: Programmer explicitly defines parallel tasks (generic, not always productive); Data parallel: An operation is applied simultaneously to an aggregate of individual items (e.g., arrays) (productive, not general)

## *Task Parallelism*



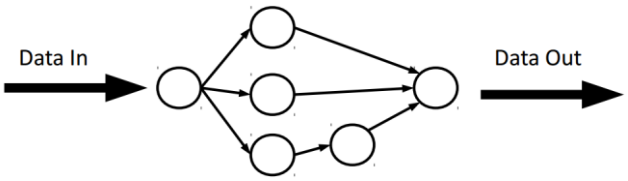
- Tasks execute code, spawn other task and wait for results from other tasks
- Tasks *can* execute in parallel (decided by the scheduler), task graph is dynamic (unfolds) – wider task graph = more parallelism
- $T_1$: $total\ work$ $(time\ for\ sequential)$

  $\frac{T_1}{T_p}$: speedup, lower bounds $T_p \geq \frac{T_1}{p}, T_p \geq T_\infty$

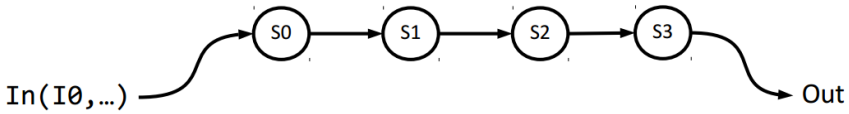
  $T_\infty$: span, critical path, longest path
- Scheduling: assigns task to processor, upper bound $T_p \leq \frac{T_1}{p} + T_\infty$ can be achieved with a greedy scheduler (all processors are executing tasks, if enough tasks available), optimal with a factor of 2, linear speedup for $\frac{T_1}{T_\infty} \geq p$
- Work stealing scheduler: provably $T_p = \frac{T_1}{p} + O(T_\infty)$, empirically $T_p \approx \frac{T_1}{p} + T_\infty$, linear speedup if $\frac{T_1}{T_\infty} \gg p$, parallel slackness (granularity)
- Common structure for divide & conquer (e.g. accumulator/$\sum a_i$):

```
Divide and Conquer:
    if cannot divide:
        return unitary solution (stop recursion)
    divide problem in two
    solve first (recursively)
    solve second (recursively)
    combine solutions
    return result
```

- Task graph can also be static, e.g. pipeline, streaming, dataflow
- Dataflow: Programmer defines: what each task does and how the tasks are connected




- **Pipeline**[7]: time unit is determined by the slower/slowest stage (→stalling), every stage should take the same amount of time; can be achieved by using splits and joins for parallel stages



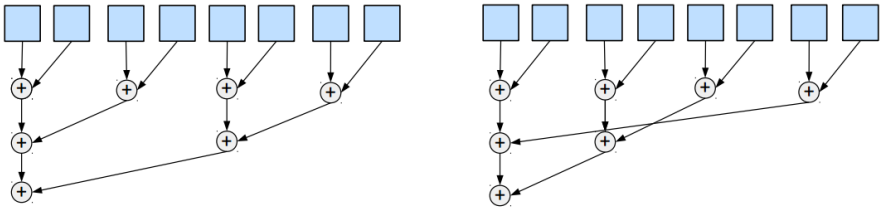

- Scheduling dataflow programs: scheduling: assigning nodes (tasks) into processors, $n < p$: cannot utilize all processors,$n = p$: one node per processor, $n > p$: need to combine tasks; portability, flexibility (parallel slackness), balancing, minimize communication (graph partitioning); dataflow is a good match for pp,

[7] Already discussed earlier to some extent

since the programmer isn't concerned with low-level/edge implementation details; can be generalized with feedback loops (performance becomes more difficult, though)

### *Data Parallelism*

- In data parallelism, the programmer describes an operation on an aggregate of data items (e.g., array); work partitioning is done by the system; declarative: programmer describes what, not how
- Map/reduce: map example: $B = 2 \cdot A$ where actually $b_i = 2 \cdot a_i \forall i$; reduce example: d&q accumulator making use of associativity and commutativity (second example: max()[8])



- Parallel loops can be used for work partitioning by adding generality yet possibly introducing "weird" bugs due to data races (e.g. operation on $a_i$ depends on $a_{i-1}$)

### *Managing State*

- Main challenge for parallel programs

| IMMUTABILITY | ISOLTEAD MUTABILIY | MUTABLE/SHARED DATA |
|---|---|---|
| – data do not change<br>– best option, should be used when possible | – data can change, but only one execution context can access them<br>– message passing for coordination<br>– State is not shared – each task holds its own state<br>– (async) messages<br>– Models: actors, CSP (Communicating Sequential Processes) | – data can change / all execution contexts can potentially access them<br>– enabled in shared memory architectures<br>– however: concurrent accesses may lead to inconsistencies<br>– Solution: protect state by allowing only one execution context to access it (exclusively) at a time; e.g. using locks (good performance, correctness issues) or transactional memory (correct, bad performance) |

## 5 Introduction to Programming

*I'm not going to talk about syntax and the like, you can read up on this in the lecture slides.*[9]

- Old Egyptian Multiplication: say you want to multiply $a$ with $b$. In one column you keep writing down $2^n$ as long as this is $\leq a$. In the other column you start with b (for row a=1) and then keep doubling the last row until you reach the last row of $2^i$. Then you figure out which $2^i$ form a, cross out the other rows and then add up the corresponding rows. Example: $27 \cdot 12 = (16 + 8 + 2 + 1) \cdot (12 + 24 + 96 + 192) = 324$

| **27 = 16+8+2+1** | **12** |
|---|---|
| 1 | 12 |
| 2 | 24 |

[8] Similar: prefix scan
[9] No offense!

| | |
|---|---|
| ~~4~~ | ~~48~~ |
| 8 | 96 |
| 16 | 192 |
| ~~32~~ | ~~384~~ |

- Russian Peasant Multiplication: In one column (division column) you keep dividing the number $a$ (while floor()[10]-ing if need be)until you reach 1 while in the other column (multiplication column) you keep multiplying $b$ as long as the corresponding row in the division row hasn't yet reached 1. In binary you keep deleting the LSB in the division column while adding 0s in the multiplication column. Then you cross out lines with an even number in the division column and sum up the values in the remaining multiplication column. This method is super great for CPUs! **Note:** If the multiplicand is odd, you have to add $a$ in the end (die to underestimating $a$)

  Formally: $a \cdot b \begin{cases} a, if\ b = 1 \\ 2a \cdot \frac{b}{2}, if\ b\ even \\ a + \left(2a \cdot \frac{b-1}{2}\right), else \end{cases}$, recursive: $f(a,b) = \begin{cases} a, if\ b = 1 \\ f\left(2a, \frac{b}{2}\right), if\ b\ even \\ a + f\left(2a, \frac{b-1}{2}\right), else \end{cases}$
- Important concept of Exception Handling; main keywords: try{…}catch(…){…} and throw[s] …

# 6 Java Basics

- Java is an interpreted (using compiled byte code) language running in the Java Virtual Machine (JVM), making it possible to run on virtually any computing device
- When writing an algorithm: KISS (keep it simple and stupid), group it logically, try to write re-usable code, DRY (don't repeat yourself)
- In Java, the 'main' method has a special significance, it gets called at runtime automatically as an entry point
- Java uses types (strongly typed) – primitive[11] (byte, short, int, long, float, double, char, boolean) and object (String, and all the rest); everything has a type (and needs to be declared as such); Types can be cast using myInt = (int) myFloat

# 7 Loops – Objects – Classes

- Loops can be definite (think 'for'), indefinite (think 'while') and sentinel[12] (until a sentinel value is seen)
- Fencepost problem (off-by-one error): (Wikipedia) "It often occurs in computer programming when an iterative loop iterates one time too many or too few.", can be avoided by e.g. using a do-while loop
- Arrays[13] are zero-based and use key-value pairings (or index-value), play well with for(each) loops; arrays are reference types; Arrays can throw ArrayIndexOutOfBoundsException if not implemented correctly/thought through
- Strings aren't created with 'new', they actually are a 'char'-array; strings can't be compared with '=='[14] (reference comparison of objects)
- Classes are code (just like a blueprint) while objects are instantiated classes (code vs. runtime); objects contain data (variables) and objects (methods); classes are (often) abstractions
- Null is special (can be used for a non-argument, return value for failed calls, default value of a variable etc.)
- **Encapsulation**: very important in OO – every object has internal and external view, it's also a form of protect (information hiding), methods maintain data integrity, different visibility keywords (public (eve-

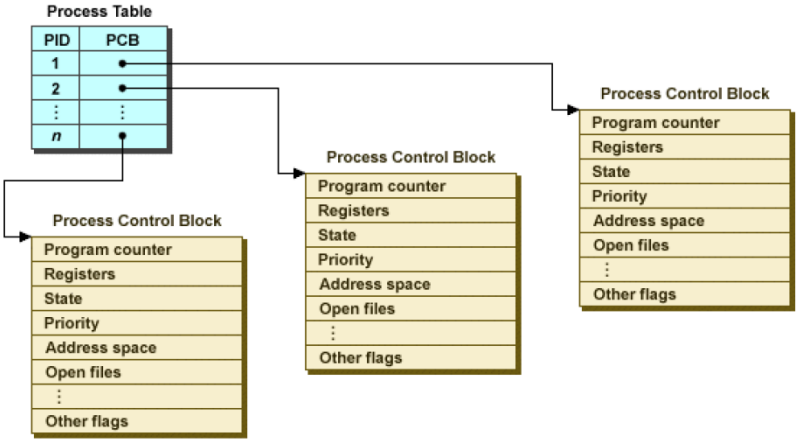
[10] Round down to the nearest integer, $\lfloor n \rfloor$
[11] Which are not real objects, instead they have a wrapper class
[12] German (here): Markierung
[13] 'int diaryEntriesPerMonth[] = new int [31]'
[14] 'equals()'

rywhere), private (only from this class), protected (current package and subclasses, regardless of package)); benefits: protects from unwanted access, implementation can be changed later, object's state can be constrained (invariants)

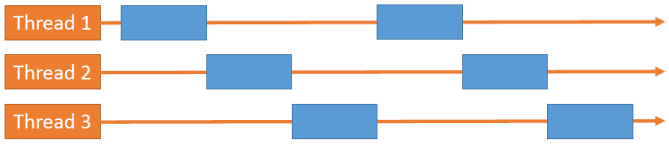
- Java uses packages (for namespacing)
- 'this' refers to the implicit parameter inside your class
- Class methods are marked with 'static' (can be called from a static context, e.g. main()); they're often generic and need no access to object variables and methods; serve as utility functions

# 8 Threads

- **Multitasking**: concurrent execution of multiple tasks: time multiplexing on CPU (creates impression of parallelism even on single core system); allows for asynchronous I/O
- **Process context**: instruction counter, register content, variable values, stack content, resourcing
- **Process states**: main memory: created, waiting, running, blocked, terminated; page file/swap space: swapped out waiting, swapped out blocked
- **Process management**: CPU time, memory; tasks managed by OS: start/terminate processes, control resource usage, schedule CPU time, synchronization of processes, inter-process communications
- Process control blocks (PCB)(see image) – process level parallelism can be complex and expensive

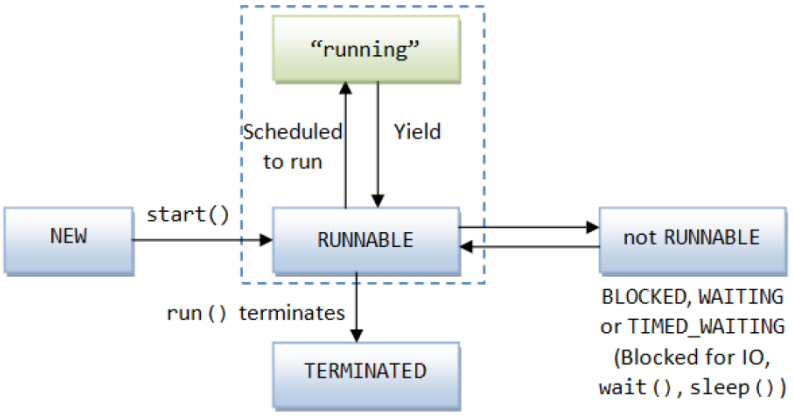



Multiple threads sharing a single CPU




Multiple threads on multiple CPUs


- Threads are light weight processes, they are independent sequences of execution but multiple threads share the same address space, they aren't shielded from each other but share resources and can communicate more easily, context switching is much more efficient; advantages: reactive system by constant monitoring, more responsive to user input (GUI interruption), server can handle multiple clients simultaneously, can take advantage of parallel processing




- Overriding methods: a subclass' method replaces a superclass' version of the same method
- **Interface**: list of method a class can implement; gives you an is-a relationship and without code-sharing (inheritance shares code)

- Creating threads in Java: extends 'java.lang.Thread' (override method, run()/start(); implement 'java.lang.Runnable'[15] (run()), if already inheriting from a class[16], 'Thread' implements 'Runnable'
- Every Java program has at least one execution thread (which calls main()), each call to Thread.start() creates a new thread (but not just the creation of a Thread object and run() doesn't start a thread either), program ends when all threads finish yet they can continue even if main() returns
- A thread has the following attributes (getters and setters): ID, name, priority (1…10), status (new, runnable, blocked, waiting, time waiting, terminated)
- A thread can throw 'InterruptedException'; can be requested by Thread.interrup() but can be ignored; fain grained control with isInterrupted(), interrupted()
- Checked exceptions (quoted): represent invalid conditions in areas outside the immediate control of the program (network outages, absent files); are subclasses of Exception; a method is obliged to establish a policy for all checked exceptions thrown by its implementation (either pass the checked exception further up the stack, or handle)
- Unchecked exceptions (quoted): represent conditions that, generally speaking, reflect errors in your program's logic and cannot be reasonably recovered from at run time (bugs); subclasses of RuntimeException, and are usually implemented using IllegalArgumentException, NullPointerException, or IllegalStateException; a method is not obliged to establish a policy for the unchecked exceptions thrown by its implementation (and they almost always do not do so)
- (quoted) Threads can make concurrent (and asynchronous) workflows faster even on single core machines. If execution units are well separated they can make programs even simpler to write.; for data heavy and compute intensive parallel programs there is usually no speed-up beyond the number of physical cores; Even then scheduling and communication overhead might reduce performance gains

# 9 Synchronization: Introduction to locks

- need for synchronization: races
- in a sequential program, 1:1 of program and data, in a parallel (multi thread) program, many different threads need to access data - data needs to be protected since concurrent access *might* (-> in sequential program, bugs become apparent quickly (Exception: boundary conditions)) lead to inconsistencies, concurrent access bugs often depend on execution conditions (#CPUs, load, timing) and (thus) they're difficult to reproduce
- sequential algorithms assume they act alone, thus them acting on data is unsafe; hardware/software optimizations assume sequential execution (which can mess up things)
- Example: circular doubly linked list: every node has a forward and a backward pointer and the list is circular (last and first are linked); easy to insert using four operations - remove is two operations (easy as well); remove() with 2 threads can create a mess (depending on the scheduling)
- **race condition**: correctness depends on relative timing; data race: unsynchronized access to shared mutable data; most race conditions are due to data races (but not always)
- avoiding race conditions: very difficult to consider all possible execution interleavings; instead use locks, locks - atomicity via mutual exclusions
- atomicity: operations A and B are atomic with respect to each other, if "thread 0 executes all of A, thread 1 executes all of B and Thread 0 **always** perceives either all of B or none of it"; intermediate state cannot be observed, only start and end
- atomic mutual exclusion works on sequential algorithms (with no or little adaption)
- thread safety: apply OO techniques (encapsulation): design "thread safe classes" which encapsulate any needed synchronization so that the clients don't need to worry about that; you should build your program by composing thread-safe classes (which isn't always easy); definition: "Behaves correctly when accessed by multiple threads" and doesn't require synchronization from users

[15] 'public class [Name] implements Runnable {}'
[16] Java doenst have multiple inheritance

- concurrent access breaks many assumptions from sequential programming (counter-intuitive); performance vs productivity
- we cannot make any assumptions about relative execution Speed of threads (-> synchronization with sleep() is wrong); even a single instruction is not atomic (value++ needs READ, INCREMENT, STORE); can be avoided using Java keyword "synchronized"
- Basic synchronization rule: Access to shared and mutable state needs to be **always** protected - Access includes both reads and writes (fine print: you can break it if you know and understand architectural details)
- **Locks**: a lock object instance defines a critical section; lock->enter, unlock->leave; there can be only one (thread)
- Java has intrinsic locks, each Java object contains a lock (built-in, for your convenience yet undesired implications (size)); can be used via they keyword "synchronized" on code blocks or entire function
- reentrant lock: (example: two synchronized functions, and one function calls the other) if a thread tries to acquire a lock it already holds it succeeds (normally this leads to deadlock, where the program is unable to proceed); in Java intrinsic locks are reentrant
- **reentrant**: per-thread acquisition, non-reentrant: per-invocation locks (some argue this is better since you need to think harder); trade-off: flexibility <-> productivity
- explicit locks: not a replacement for "synchronized", provides more flexibility (additional calls, non-block structured locks)
- synchronized vs function call: synchronized: part of the language, less error-prone but less flexible; function call: library, error-prone but more flexible
- reentrant lock using lock interface with try/catch/finally{unlock}
- Example: synchronizing circular doubly linked list: if every single call has its own "synchronized", interleavings are still possible, correctness depends on the semantics of the program, operations need to be synchronized *properly* (one of the reasons why PP is hard)
- rule: to preserve state consistency, update related state variables in a single atomic operation (simple approach: use "synchronized" on (every) method)
- Java servlets (implementing an interface), multiple threads for better performance are a good idea as long as they're thread-safe, they **have to**
- counters can be "synchronized", too, for thread-safety
- use built-in mechanisms where available (e.g. java.util.concurrent.atomic.AtomicLong) - caching results (memoization): cache input->result; might save expensive computation (followed by storing the result in the cache using .clone()); yet one might lose parallelism by using "synchronized" on the whole function -> use "synchronized" only on critical sections (for cache r/w access) instead of on the whole function
- locks and Amdahl's Law: at some point (if you keep adding processors), the "synchronized" parts will dominate (so keep them at minimum)

## 10 Synchronization: Using Locks and Building Thread-Safe Classes

- model: n threads, m resources
- coarse-grained locks (big lock), locking all resources (critical section), threads are serialized, but bad performance
- fine-grained locks, e.g. every resource is protected by one lock (note: 1 thread may have multiple locks), better performance; problems: using multiple locks for one thread, deadlock is possible
- deadlock: no thread is able to continue, caused by circular dependencies; runtime condition; necessary conditions: mutual exclusion (at least one resource must be non-shareable), had and wait (a thread holds at least a lock that it has already acquired, while waiting for another lock), no forced lock release (locks can only be released voluntarily by the threads (and not by system)), circular wait (e.g. $p_1$ waits for $p_2$] and so on and $p_n$ waits for $p_1$); mutual exclusion, hold and wait and no forced lock release can't be broken with "synchronized" and breaking them leads to complicate synchronization schemes

- breaking the circular wait condition: set of global order of locks, acquire locks respecting that order, impossible to create a condition of circular wait -> impossible to deadlock
- Example: Hash Table (key, hash function, buckets, collision lists); attempt to make it thread-safe: huge lock around the whole data structure (apply hash() to key, locate bucket, search for key in the list and, if found, return value) - bad, Amdahl's Law: synchronized part will dominate eventually; per-bucket lock (search for key in the list and, if found, return value)
- Example: Hash Table: per-bucket locks - discussion: operations involving only a single key (insert, lookup, remove) are easy; swap(key1, key2): exchanges values between those two keys *atomically*, locking with a single lock is trivial, but what about multiple locks?: 1st attempt: lookup() twice, then insert() twice, each one locked: intermediate state might be observed between the two insert()s, 2nd attempt: lock both buckets -> no intermediate state can be observed (locate both buckets and then, while synchronized, search for both keys and swap them), however there's the possibility of a deadlock which can be avoided with lock ordering (using the hash table's indexes; special case for the same bucket if locks aren't reentrant)
- Visibility – synchronization also enforces visibility
- in example (42) the reason(s) for the problem are: until a few years ago, sequential performance was the main focus, thus optimizations were done with respect to sequential programs (CPU/compiler tuning, out-of-order execution) are only guaranteed for sequential execution (and cancelling them isn't affordable) and thus counter-intuitive behavior in parallel settings is common
- building thread-safe classes: immutable and stateless are always thread-safe (assuming correct Initialization), mutable and shared data needs to be protected; try to avoid synchronization since it's difficult to reason about locks and there are performance issues (->immutable/stateless)
- 'final' keyword in Java: final class cannot be subclassed, final method cannot be overridden or hidden, final variable can only be initialized once (and only in the class constructor) (but referenced object *can* be changed)
- immutable objects in Java - properties: 1) its state cannot be modifier *after* construction, 2) all fields are final, 3) it's *properly constructed* (= final fields need to be set in the object constructor; when the constructor is done, the object is immutable; while the constructor runs, the object is mutable; it should not be access during that time); they are always thread-safe!
- Generics – motivation: the implementation of e.g. a linked list is independent from the underlying element used (implementor perspective); static checks for linked lists operations (user perspective); decoupling of data structure and algorithms that operate on them
- improper construction: 'this' escapes
- '**AtomicReference**<V>' is an atomic access to a reference of V (comparable (with care) with 'volatile')
- immutable vs non-immutable objects: immutable objects are special which is specified in the Java memory model and they don't require safe publication while non-immutable objects need to published safely since no ordering guarantees are proved by the memory model and a thread might observe a non-safely published object in an inconsistent state

## 11 Synchronization: Beyond Locks

- Locks provide means to enforce atomicity via mutual exclusion yet they lack means for threads to communicate about changes
- Example: **producer/consumer (p/c) queues** (think: bakery): can be used for data-flow parallel programs, e.g. pipelines where a mean to transfer X from the producer to the consumer is needed. There might be multiple producers or (not xor) multiple consumers. For an implementation a circular buffer (with a fixed size) can be used with simple dequeue()/enqueue() and an "in" and "out" counter. Both functions use a shared (reentrant) lock and rely on helper functions to check for full/empty queue[17]. If you use a busy wait (while loop) there is a chance of a deadlock (and CPU running high). Using sleep for synchronization as another

[17] Note: If you have a try-catch-finally block and there is a return statement (assume it will be called no matter what) in the "try" part and an unlock() in the "finally" part, the finally part **will always** be executed (and thus also the lock released!).

approach is generally discouraged[18]. The solution is a condition variable which (ideally) notifies the threads upon change.

- A **condition** interface provides the following methods: .await() – the current thread waits until it is signaled; .signal() – wakes up one waiting thread; .signalAll() – wakes up all waiting threads. Conditions are **always** associated with a lock. .await() is called with the lock held, releases the lock *atomically* and waits for thread to be signaled and is *guaranteed* to hold the lock when returning and the threads *always* needs to check the condition. .signal[All]() is called with the lock held.
- Check then act![19]
- Conditions can also be used with intrinsic locks where each object can act as a condition, implementing .notify(), .notifyAll(), .wait(). They do not allow for multiple conditions.
- Object.wait and Codition.await: always have a condition predicate; always test the condition predicate: before calling wait and after returning from wait; always call wait in a loop; ensure state is protected by lock associated with condition
- **Semaphores**[20] have the following operations: *initialize* to an integer value and after initialization only wait/signal operations are allowed; *acquire*: integer value is decreased by one, if < 0 → thread suspends execution; *release*: integer value is increased by one if there is at least a thread waiting, one of the waiting threads resume execution; A thread cannot know the value of a semaphore and there is no rule about what thread will continue its operation after a release()
- Say you build a lock (mutex) with a semaphore, you initialize it to 1 and then 1 means unlocked, 0 is locked, -n means n threads are waiting to enter.
- You can (of course) also use semaphore for p/c queues, however you need to use two semaphores to order the operations (and to prevent a deadlock).
- **Barrier**: rendezvous for arbitrary number of threads i.e. every thread has to wait up for all other threads to arrive at a certain point; can be implemented for n threads with two semaphores (and one count variable), one as a mutex (used to atomically increment the counter) with default = 1 and one as a barrier with default = 0 (which is released if count == n and otherwise only acts as acquire-and-release-immediately).
- If you want a reusable barrier for n threads (aka 2-phase barrier) with semaphores, you need a count, a mutex and two barriers for it to be thread-safe.

## 12 Advanced (and other) Topics

- Locks can be implemented with low-level atomic operations and busy wait loops.
- Simple example: Peterson lock [for two threads] (see code on the right) where two AtomicBooleans (one per thread) and an AtomicInteger which decides which thread will be selected.
- Rich(er) atomic operations for AtomicInteger: getAndSet(val) (atomically { set to val, return old value }), getAndAdd(val), getAndIncrement, getAndDecrement, CompareAndSet (CAS for short)

```
AtomicBoolean t0 = new AtomicBoolean(false);
AtomicBoolean t1 = new AtomicBoolean(false);
AtomicInteger victim = new AtomicInteger(0);
lock:
my_t.set(true)
victim.set(me);
while (other_t.get() == true &&
victim.get() == me)
;
unlock:
my_t.set(false);
```

- **Lock using getAndSet**: mutex is an AtomicBoolean which is set to either true or false on lock or unlock (resp.)
- CAS(int old, int new): performs atomically the following (optimistically): if current_val == cold then current_val = new, return true else return false
- Lock using getAndSet: mutex is an AtomicBoolean which is either CAS'ed as compareAndSet(false, true) or set(false) on lock or unlock (resp.)

[18] Mentioned in an earlier lecture.
[19] This can also be helpful for bungee-jumping.
[20] Language background: semaphore is fancy for traffic light in English (see also Spanish).

- **Busy-waits** check continuously for a value and waste CPU-time (or as alternative: exponential backoff) which should be avoided using a notification mechanism of sorts
- **Mutexes**: locks that suspend the execution of threads while they wait are typically called mutexes (vs spinlocks); scheduler (typically from the OS) support is required; they do not waste CPU time but they have higher wakeup latency; hybrid approach: spin and then sleep
- **Locks performance**: Uncontended case: when threads do not compete for the lock, lock implementations try to have minimal overhead, typically just the cost of an atomic operation; Contended case: when threads do compete for the lock, can lead to significant performance degradation, also, starvation
- Disadvantages of locking: locks are pessimistic by design, they assume the worse/worst and enforce mutual exclusion; performance issues: overhead for each lock taken even in uncontended case, contended case leads to significant performance degradation; blocking semantics (wait until acquire lock), if a thread is delayed for a reason (e.g., scheduler) when in a critical section → all threads suffer, lead to deadlocks (and also livelocks)
- *Locks*: a thread can indefinitely delay another thread; *non-blocking*: failure or suspension of one thread cannot cause failure or suspension of another thread; *Lock-free*: at each step, some thread can make progress
- **Non-blocking algorithms**: typically built using CAS (more powerful than plain-atomic); see lecture slides for stack example
- Overview of what java.util.concurrent has to offer
  - Lock interface with lock(), lockInterruptibly(), tryLock([delay][21]), unlock(), newCondition(); implemented by ReentrantLock
  - ReadWriteLock interface with readLock() writeLock(); implemented by ReentrantReadWriteLock; multiple readers can concurrently access state whereas writers get exclusive access, beneficial for scenarios with comparably few writes; can be implemented with semaphores but fairness might be an issue leading to starvation[22] unless prevented by means to notify the read lock about waiting writers
  - Collections: objects that group multiple elements into a single unit; interfaces: Collection, List, Set, SortedSet, …; implementations: ArrayList, LinkedList, …; Algorithms: sort, …; based on Java generics
  - Synchronized Collections: Vector, HashTable, synchronizedList, synchronizedMap, synchronizedSet, synchronizedSortedMap, synchronizedSortedMap, synchronizedCollection; they are wrapper classes, basically wrapping every public method in a synchronized block, they are thread safe but poor concurrency due to a single, collection-wide lock
  - Concurrent Collections: thread safe, but not a single lock; ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArrayList
  - Queues: BlockingQueue: ArrayBlockingQueue, LinkedBlockingQueue (FIFO), PriorityBlockingQueue (ordered); TransferQueue: allows to wait until a consumer receives item; SynchronousQueue: hand-of, no internal capacity; Dequeue/BlockingDeque: allows efficient removal/insertion at both ends (head/tail), work stealing pattern
  - Synchronizers: Semaphores, CyclicBarrier, CountDownLatch (thread wait until countdown reaches zero)
  - Future: interface with get(), isDone(), cancel() representing a result for async computation

## 13 Parallel Tasks

Example for most of this lecture: $\sum x_i$

- When writing a parallel program, write a sequential version first! This is useful for knowing the results are correct and evaluate the performance of the parallel program.

```
public static int sum(int[] xs) {
 int sum = 0;
 for (int x: xs)
  sum += x;
 return sum;
}
```

[21] Using the PHP docs style where square brackets denote optional arguments

[22] Starvation: when a particular thread cannot resume execution; different from deadlock, where all the threads are unable to proceed

- Divide-and-conquer approach: recursive sum with the lower and upper half of the remaining part which cuts off at size = 1 is a lot slower (x10).[23]
- Task parallel model: basic operations: create a parallel task and wait for the tasks to complete; when using D&C a task for the first and second part (one each) are created and upon finishing their results are combined
- One thread per task: expensive to create, consumes many resources and is scheduled by the OS – generally inefficient
- **ExecutorService**: A (huge) amount of tasks is handled by an interface which assigns a thread from a thread pool to each task and returns a Future[24]
- Note: Runnable doesn't return a result, Callable does
- ExecutorService and recursive sum[25]: task is described as the array to be summed and the region for which the task is responsible for, additionally an instance to the ExecutorService is passed so that the task can spawn other tasks; problems (observation: no result returned): Tasks create other tasks and then wait for results, when they are waiting they are keeping threads busy, other tasks need to run so that the recursion reaches its bottom, system does not know that tasks waiting need to be removed so that other tasks can run *due to*: tasks create other tasks (which is not supported) and work partitioning (splitting up work) is part of the task – we can decouple work partitioning from solving the problem
- **Fork/Join framework** with ForkJoinTask (fork() creates a new task, join() returns the result when task is done, invoke() executes task without creating a new task; subclasses need to define compute()) implements Future and ForkJoinPool implements ExecutorService; Note fork(), fork(), join(), join() doesn't work (well) in Java, solved by using: t1.fork(), r2 = t2.compute(), return r2 + t1.join();[26]
- Problems of overhead: bad speedup due to too much overhead (scheduling etc.), can be solved by making each task work more, here: increase cutoff

# 14 Transactional Memory (TM)

- Aims at removing the burden of having to deal with locks from the programmer and place it on the system instead
- Problems with locks: ensuring ordering and correctness is really hard and locks are not composable; locks are pessimistic, performance overhead; locking mechanism is hard-wired to the program (separation not possible and change of synchronization scheme results in changing all of the program)
- With TM, the programmer explicitly defines atomic code sections and is only concerned with the *what* and not the *how* (declarative approach)
- **TM benefits**: easier, less error-prone, higher semantics, composable, optimistic by design; changes made by a transaction are made visible atomically; transactions run in isolation – while a transaction is running, effects from other transactions are not observed (as if transaction takes a snapshot of the global state when it begins and operates on that snapshot);note: while locks enforce atomicity via mutual exclusion, transaction does not require that
- TM is inspired by transactions in databases where transactions are vital; ACID – Atomicity, Consistency, isolation, Durability
- Implementation of TM: Keep track of operations performed by each transaction: concurrency control, system ensures atomicity and isolation properties
- Transactions can be aborted if a conflict has been detected by the concurrency control (CC) mechanism; aborts are possible e.g. if there's a deadlock; on abort, a transaction can be retried automatically or the user is notified

[23] Code not listed since the code in the slides is (by design) naïve and simple, everyone should know how to do that by heart.
[24] See notes about previous lecture
[25] Have a look at the code in the slides, pp 36 – 38
[26] "+" is in this case the arithmetic addition but can also be something else of a combining nature

- Where TM is/can be implemented: Hardware TM[27]: can be fast but cannot handle big transactions; Software TM (STM)[28]: in the language, greater flexibility, performance might be challenging; Hybrid TM; TM is still work in progress with many different approaches and is still under active development
- **Design choice**: strong vs weak isolation: **Q**: What happens when shared state accessed by a transaction, is also accessed outside of a transaction? Are the transactional guarantees still maintained? **A**: Strong isolation: Yes, easier for porting existing code, difficult to implement, overhead; Weak isolation: No
- **Design choice**: Nesting[29]: **Q**: What are the semantics of nested transactions? (Note: nested transactions are important for composability) **A**: flat nesting (inner aborts → outer aborts; inner commits → changes visibly only if outer commits), closed nesting (inner abort does not result in an abort for the outer transaction; inner transaction commits → changes visible to outer transaction but not to other transaction; only when outer transaction commits, changes of inner transactions become visible), other approaches (e.g. open nesting)
- The more variables are part of a transaction (and thus protected) the easier it gets to port existing code but the more difficult to implement ,too (need to check every memory operation)
- Reference-based STMs: mutable state is put into special variables; these variables can only be modified inside a transaction, everything else is immutable (or not shared; see functional programming)
- Mechanism of retry: implementations need to track what reads/writes a transaction performed to detect conflicts, typically called read-/write-set of a transaction; when retry is called, transaction aborts and will be retried when any of the variables that were read, change
- Issues with transactions: it is not clear what the best semantics for transactions are; getting good performance can be challenging; I/O operations: can we perform I/O operations in a transaction?
- **I/O[30] in transactions**: in general, I/O operations cannot be rolled-back and thus generally cannot be aborted; that is why I/O operations are not allowed in transactions; one of the big issues with using TM; (some) STMs allow registering I/O operations to be performed when the transaction is committed

# 15 Designing Parallel Algorithms

- There are no rules whatsoever, yet – as (very) often – it is a matter of experience
- The following points can/should be considered
  - Where do the basic units of computation (tasks) come from? This is sometimes called "partitioning" or "decomposition". Depending on the problem partitioning in terms of input and/or output can make sense or functional decomposition might yield better results
  - How do the tasks interact? We have to consider the dependencies between tasks (dependency, interaction graphs). Dependencies will be expressed in implementations as communication, synchronization and sharing (depending upon the machine model).
  - Are the natural tasks of a suitable granularity? Depending upon the machine, too many small tasks may incur high overheads in their interaction. Should they be collected together into super-tasks?
  - How should we assign tasks to processors? In the presence of more tasks than processors, this is related to scaling down. The "owner computes" rule is natural for some algorithms which have been devised with a data-oriented partitioning. We need to ensure that tasks which interact can do so as (quickly) as possible.
- **D&C** is a very important technique and particularly helpful in PP since the recursive step can instead be parallelized
- **Number of threads** to be used: "Runtime.getRuntime().availableProcessors();" *might* be the right amount but your program may not get access to all cores; too few threads are bad because core(s) is/are idle; too many threads can be bad because of the overhead[31]

[27] Intel Haswell (4th generation i3/5/7 processors) is the first wide-spread implementation of hardware TM
[28] Haskell, Clojure, …
[29] See also https://www.facebook.com/groups/infstudents13/permalink/659004517480019
[30] send/receive data over the network, write data to disks, push a button to launch a missile; essentially escape the CPU & memory system
[31] This depends on the actual overhead the language introduces (in Java rather big)

- **Sorting**[32]: If the array is sorted the following condition must hold (equal only if $A_i = A_j$): $A_i \leq A_j \; for \; i < j$; features of a sorting algorithm: stable (duplicate data is allowed and the algorithm does not change duplicate's original ordering relative to each other), in-place ($O(1)$ auxiliary space), non-comparison; some sorting algorithms: horrible $\Omega(n^2)$: bogo, stooge; simple $O(n^2)$: insertion[33], selection[34], bubble, shell; fancier $O(n \log n)$: heap, merge, quick sort (**on average!**); specialized $O(n)$: bubble, radix
- Linked Lists and Big Data: Mergesort can very nicely work directly on linked lists; Heapsort and Quicksort do not; InsertionSort and SelectionSort can too but slower; Mergesort also the sort of choice for external sorting
- Quicksort and Heapsort jump all over the array; Mergesort scans linearly through arrays; In-memory sorting of blocks can be combined with larger sorts; Mergesort can leverage multiple disks
- **PRAM model**: processors working in parallel, each is trying to access memory values; when designing algorithms, the type of memory access required needs to be considered; scheme for naming different types: [concurrent|exclusive]READ[concurrent|exclusive]WRITE[35]; typically CR are not a problem since the memory isn't changed whereas EW requires code to ensure writing is exclusive; PRAM is helpful to envision how it works and the needed data access pattern but isn't necessarily the way processors are arranged in practice

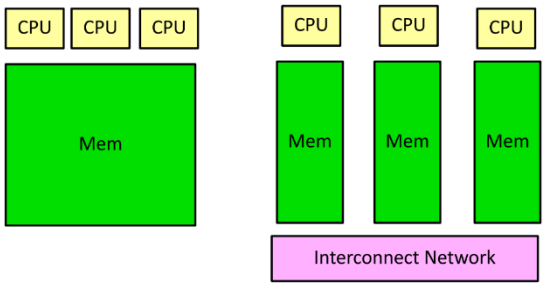
## 16 Java GUIs – MVC – Parallelism

*Don't get me wrong, but I'm having a hard time writing up this lecture…*

- (important) concepts: MVC (model (application domain, state and behavior) – view (display layout and interaction views) – controller (user input, device interaction)); layout managers; event-driven design[36] (listener, worker[37], callback, fire/handle), GUI (painting)
- Swing threads: initial[38], event dispatch[39] and worker thread[40]
- **MVC**: *model*: complete, self-contained representation of object managed by the application, provides a number of services to manipulate the data, computation and persistence issues; *view*: tracks what is needed for a particular perspective of the data, presentation issues; *controller*: gets input from the user, and uses appropriate information from the view to modify the model, interaction issues

## 17 Concurrent Message Passing

- goal: avoid (mutable) data sharing, instead use concurrent message passing (actor programming model) since many of the PP problems (so far) are due to shared state
- isolated mutable state: state is mutable, but not shared; each thread/task has its private state; tasks cooperate with message passing
- shared memory architecture (left side in image): message passing and sharing state is used; message passing: can be slower than sharing data yet is easier to implement and to reason about
- distributed memory architecture (right side in image):: sharing state is challenging and often inefficient, using almost exclusively message passing; additional concerns such as failures
- message passing works in both shared and distributed memory architectures making it more universal




[32] This is D&A thus not covered to its full extent
[33] At step $k$, put the $k^{th}$ input element in the correct position among the first $k$ elements
[34] At step $k$, find the smallest element among the unsorted elements and put it at position $k$
[35] Abbreviated as E/C and R/W; ERCW is never considered
[36] E.g. agents in Eiffel; .on() in jQuery, …
[37] In Swing, this implements Runnable
[38] Main thread
[39] Drawing/painting the GUI
[40] Background thread, can be used for (heavy) computation to keep GUI responsive

- example: shared state counting (i.e. atomic counter) with increase() and get(): approach #1: one counter thread, the other threads ask for its value; approach #2: every thread has its own (local) counter (Java: ThreadLocal), when sum is requested all threads return the value of their local counter
- example: bank account: sequential programing: single balance; PP shared state: single balance & protection; PP distributed state: each thread has a local balance (budget), threads share balance coarsely
- distributed bank account (cont.): each task can operate independently, only communicate when needed
- synchronous vs asynchronous messages: sync: send blocks until message is received (Java: SynchronousQueue); async: send does not block ("fire-and-forget"), placed into a buffer for receiver to get (Java: BlockingQueue, async as long as there is enough space (to prevent memory overflow))
- concurrent message passing programming models: actors: state-full tasks communicating via messages (e.g. erlang); channels[41]: can be seen as a level of indirection over actors, Communicating Sequential Process (CSP) (e.g. go)
- go (by Google): language support for: lightweight tasks (aka goroutines), typed channels for task communications which are synchronous (unbuffered) by default
- **actor programming model**: a program is a set of actors that exchange (async) messages; actor embodies: state, communication, processing
- An actor may: process messages, send messages, change local state, create new actors
- event-driven programming model: a program is written as a set of handlers (typical application: GUI)
- erlang: functional language; developed for fault-tolerant applications, if no state is shared, recovering form errors becomes much easier; concurrent, following the actor model; open-source
- actor example: distributor: forward received messages to a set of names in a round-robin fashion: state: an array of actors with the array index of the next actor to forward a message; receive: messages -> forward message and increase index (mod), control commands (e.g. add/remove actors)
- actor example: serializer: unordered input (e.g. due to different computation speed) -> ordered output; state: sorted list of received items, last item sent; receive: if we receive an item that is larger than the last item plus one, add it to the sorted list; if we receive an item that is equal to the past item plus one: send the received item plus all consecutive items form the last and reset the last item
- concurrent message passing in Java_ for simple applications, queues can be used which might be difficult especially for large tasks; instead use akka framework (written in Scala, interface for Java): follows the actor model (async messages), rich set of features[42]
- akka actors example: ping-pong: client sends n PINGs to server which responds with Pong upon receiving back to sender, master stops execution when receiving DONE[43]; version 2 with restart on DONE: add a message type SETUP to the client passing the server actor reference and the count, if the client receives SETUP before DONE it can either wait for DONE and the restart or discard the message
- collective operations: *broadcast*: send a message to all actors (related: multicast, sending a message to some actors), parallel broadcast using a tree where every parent forwards the message to its children until it reaches the leafs (top-down); *reduction*: perform a computation from values of multiple nodes (e.g. balance of all bank accounts), using a tree where a parent receives the message from its children, performs operation and sends it to parent (bottom-up)

# 18 Data Parallel Programming

**Data Parallel Programming**

- task vs data parallelism: task: work is split into parts, by parallelizing the algorithm, very generic but cumbersome; data: simultaneously applied operation on an aggregate of individual items (e.g. array), declarative (= what not how), splitting up the data for parallelism, less generic
- main operations in data parallelism: map, reduce, prefix scan, parallel loop
  - **map**: input: array ($x$), operation ($f(\cdot)$); output: aggregate with applied operation ($f(x)$); parallel execution: split array into chunks and assign chunks to processors (scheduling); generally more chunks leads to better load balancing (parallel slackness); order of execution must not influence the

[41] not an official term
[42] important methods to be overridden: preStart(), onReceive()
[43] code in slides

result (since order depends on scheduling), given by pure functions (no side effects, same result for same argument)

- **reduce** (reduction)[44]: input: aggregate ($x$), binary associative operator ($\oplus$) with an identity I, output: $x_1 \oplus x_2 \oplus \ldots \oplus x_n$; result stays the same for sequential vs binary tree if operator is associative($(a + b) + c = a + (b + c)$); if operation is commutative ($a + b = b + a$), different scheduling is possible; e.g. sum, max
- **prefix scan**: if it is an addition, it is a prefix sum; input: aggregate ($x$), binary associative operator ($\oplus$) with an identity I, output: ordered aggregate ($x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \ldots \oplus x_n$);
- prefix[45] scan algorithm parallel version: addition example: 1st step is a reduction where two numbers are summed together and then pass their sum up the tree until it reaches the root i.e. bottom-up summing up all the values, two at a time; 2nd step is a down sweep where every node gets the sum of all the preceding leaf values passed whereas preceding is defined as pre-order[46]; **Have a look at slides 18 – 21** if in doubt
- application of pre-scan: line-of-sight, visible points (e.g. mountain tops) from a given observation point: point I is visible iff no other point between I and the observer has a greater vertical distance ($\theta_i = \arctan \frac{altititude_i - altitude_0}{i}$); compute angle for every point, do a max-pre-scan on angle array (e.g. 0,10,20,10,30,20 → 0,0,10,20,20,30), if $\theta_i > maxprevangle_i$ then $visible_i = true$ else $visible_i = false$; parallelizable parts: for loop to compute angles, for loop to compute visibility can be written as parfors (parallel for loops)
- **parfor**: iterations *can* be performed in parallel, work partitioning -> partition iteration space; potential source of bugs if thought of as a sequential loop (data races; think factorial)

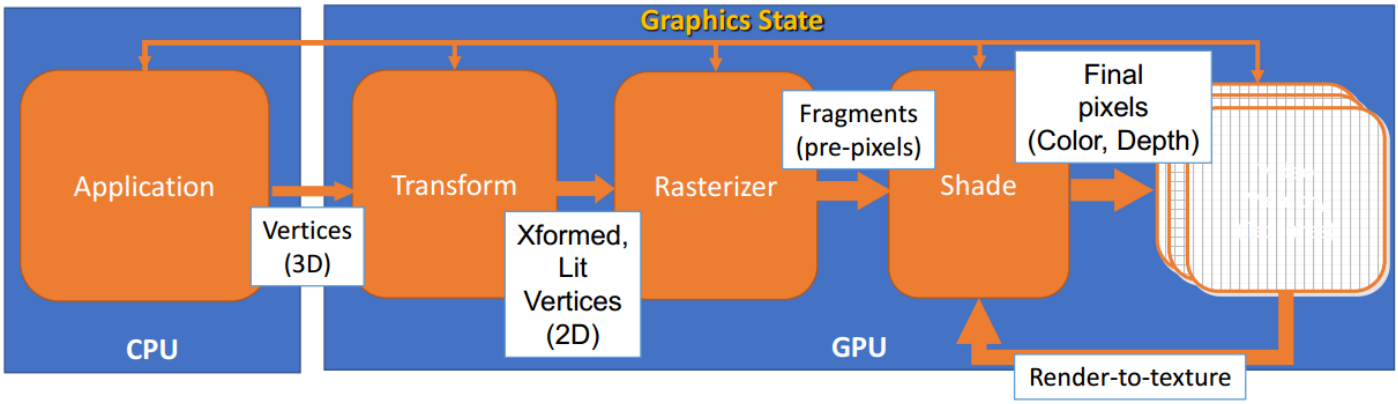
**Data Parallel Programming in Java 8**

- Functional programming crash course: functions are first-class values (composition), pure functions (immutability); such function are called lambdas or anonymous functions
- Functions as values: functions can be passed to other functions as arguments (such functions accepting such arguments are called high-order functions), e.g. $map(f, list): f$, $filter(fn, list): f$
- Lambdas make programming more convenient
- Data parallel programming in Java 8 is done using streams, providing means to manipulate data in a declarative way, allowing for transparent parallel processing;
- Menu example: input: stream, output: stream, map/filter/etc. are applied; collect in the end, doesn't create a stream; overall translates a stream into a collection[47]
- Parallel streams: created by applying .parallel() on a stream, splits it up into chunks for different threads; implemented using ForkJoin

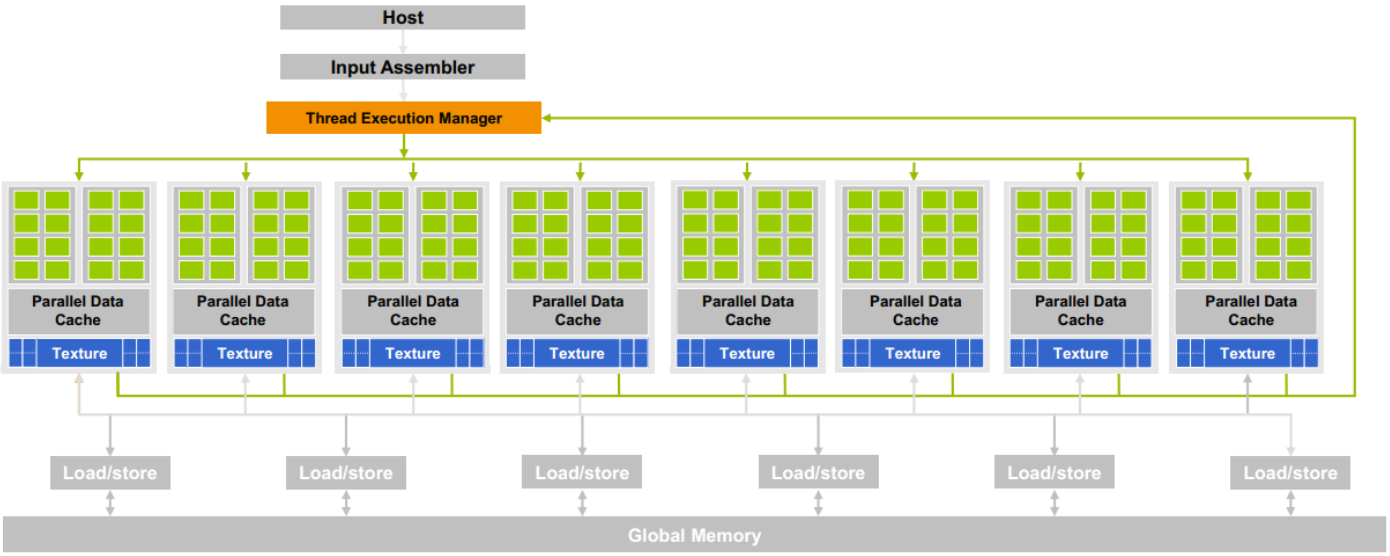
# 19 Introduction to GPGPU

- General purpose GPU programming/massively parallel GPU programming
- GPUs can calculate way more (G)FLOPS[48] than any CPU and have a much higher memory bandwidth but have only a (very) limited feature set
- This development is also due to the fact that increasing clock speed isn't a feasible option (anymore); energy consumption, length of data lanes, … thus GPUs, which have many cores on one chip, have to be reduced in terms of complexity/speed
- Vocabulary: rendering: generate an image from a 3D model; vertex: the corner of a polygon; pixel: smallest addressable screen element

[44] Remember the introductory lecture where we were asked how we could efficiently sum up all money in the lecture hall? There you go!
[45] If in node $i$, $in_i$ is *not* added , the operation is called a *pre*-scan
[46] Depth-first, left-to-right
[47] Think SQL
[48] Floating point operations per second




- above: fixed graphics pipeline, historical; the "transform" and the "shade" processes are programmable as the vertex and the fragment (resp.) processor
- **transform** / vertex processor: transform from would space to image space; computer per-vertex lighting
- **rasterizer**: convert geometric representation (vertex) to image representation (fragment), which is a pixel with associated data (color, etc.); interpolate per-vertex quantities across pixels
- **shade** / fragment processor: compute a color for each pixel and optionally read colors from textures (images)
- the potential of GPGPU lies in the following: the power and flexibility of GPUs makes them an attractive platform for general-purpose computation; examples are in a wide range from games to conventional CS; Goal: make the inexpensive power of the GPU available to developers as a sort of computational coprocessor
- GPUs used to be difficult to use since there were largely secret and only designed for games and you can't just port CPU code to a GPU; other problems wee: you had to deal with a graphics API, address modes limited size/dimension, the shader limited outputs, integer & bit operations lacked and communication was limited (between pixels); yet the underlying architecture is rapidly evolving and inherently parallel
- **CPU vs GPU** is low latency vs high throughput; whereas CPUs are optimized for low latency, cache out-of-order and speculative execution, GPUs are optimized for data-parallelism, have a memory-latency-tolerant architecture and much more transistors dedicated to computation
- Granularity: in a CPU cluster communication isn't necessary that soon since the CPU operates on a large chunk of data while in a GPU individual tasks are rather small thus requiring more communication
- GPUs are SIMD – single instruction multiple data; $(\text{inst}, (\text{data}_k)_k) \rightarrow (\text{result}_k)_k$, e.g. $(a_0, a_1, a_2, a_3) + (b_0, b_1, b_2, b_3) = (c_0, c_1, c_2, c_3)$
- Specializations of SIMD: vector processing (GPGPU), fixed HW & SW vector width; SIMT (T for threads), fixed HW but flexible SW vector width, similar to MIMD/shared memory; SIMD/MIMD: VLIW (very long instruction word), fixed HW vector width, one instruction with sub instructions, Multiple instructions of single instruction type multiple data
- The GPU is a fast, parallel array process, it's also a fast, highly multi-threaded processor

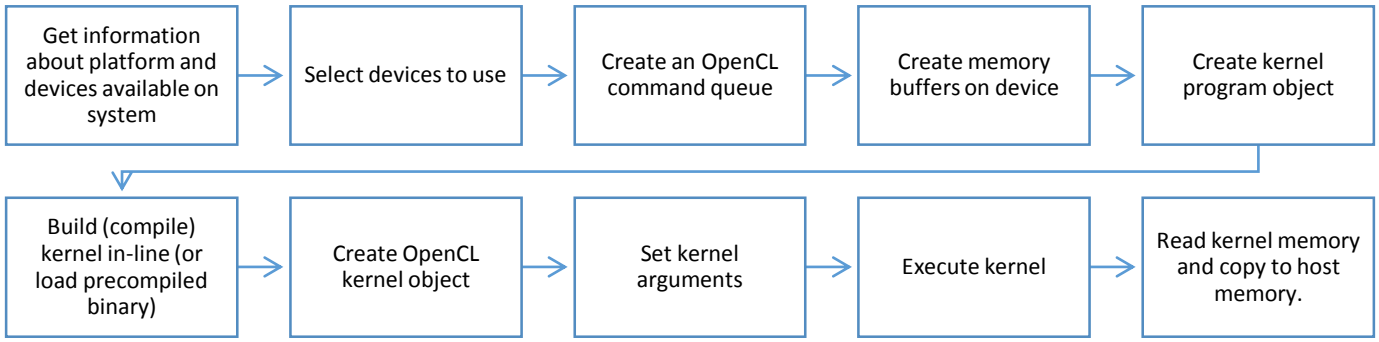


- Above: modern GPGPU architecture with: streaming processor array (SPA), all the green-grey-blue blocks; the streaming multiprocessors (SM), a green-grey-blue-block, multi-threaded processor cope, fundamental processing unit for a PGU thread block; streaming processor (SP), a single green block, scalar ALU for a single CUDA thread
- An SM (can) consists of single precision and double-precision cores, special function units and load/store units
- Heterogeneous computing: emerging intersection between CPU and GPU programming, taking advantage of both world; OpenCL/CUDA
- To enable scalable high-performance, both, CPU (task parallel, serial code) and GPU (data parallel, graphics-like code), need to be used
- **CUDA** is a general purpose programming model, its targeted software stack and drive for lading computation programs into the GPU and it's not another graphics API
- **OpenCL** is an open standard for heterogeneous parallel programming, is implemented on diverse platforms (and also runs) but it needs tuning for good performance; way younger than CUDA; it's an API and C-like language (ANSI-C99); the code is portable – correctness is guaranteed, performance not; targets a broader range than CUDA thus deals with much greater HW diversity; philosophy: avoid vendor specific details, enable high performance on different HW, write once – run everywhere, not only on GPUs (CPUs, FPGAs, …), data and task parallel models
- OpenCL technology stack: kernels: code executed by compute device; platform layer for device querying and management from the host; runtime layer: abstracts away device specifics, allows the programmer to schedule jobs
- OpenCL API specification is in C/C++, kernels are written in OpenCL-C (minimal extensions to ANSI C99)
- Comparison between C and Java:

| **Language Aspect** | **C** | **Java** |
|---|---|---|
| **type of language** | function oriented | object oriented |
| **basic programming unit** | function | class = ADT |
| **portability of source code** | possible with discipline | yes |
| **portability of compiled code** | no, recompile for each architecture | yes, bytecode is "write once, run anywhere" |
| **memory address** | pointer | reference |
| **manipulating pointers** | *, &, + | no direct manipulation permitted |
| **pass-by-value** | | |
| | primitive data types, structs, and pointers are passed by value; array decays to pointer | all primitive data types and references(which includes arrays), are passed by value |
| **allocating memory** | malloc | new |
| **de-allocating memory** | free | automatic garbage collection |

## 20 OpenCL Basics

- Very simple example: say you have a this $\prod_{i=0}^{n} a_i \cdot b_i$, in an OpenCL Kernel you would write $a_{id} \cdot b_{id}$ and the $id$ is retrieved from *get_global_id(0)*.
- Data parallelism in OpenCL: you define an n-dimension, $n = 1 \dots 3$, computation domain, each element in this domain is called a work item, depending on the N-D domain you get a different number of work items which can be executed in parallel
- Problems that map well to a GPU: separation of problem into independent parts, linear algebra, random number generation, sorting (radix sort, bitonic sort), regular language parsing; problem that aren't that suitable: inherently sequential problems, non-local calculations, anything with strong communication dependence, device dependence
- Below: **structure** of an OpenCL application



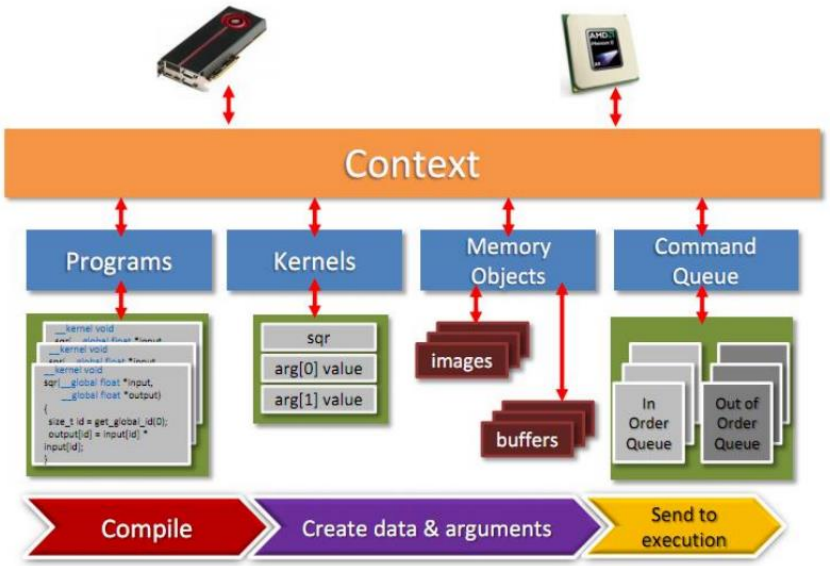

- OpenCL **programs**: An OpenCL "program" contains one or more "kernels" and any supporting routines that run on a target device; an OpenCL kernel is the basic unit of parallel code that can be executed on a target compute device
- OpenCL **execution model**. Host + device app C/C++ program[49] with serial or only modestly parallel host code and highly parallel parts in device SIMD kernel C code
- An OpenCL **kernel** is executed by an array of work items (threads): all work items run the same code (SIMD); each work item has an index that it uses to compute memory addresses and make control decisions
- Work groups: work items within a work group cooperate via shared memory, atomic operations and barrier synchronization; work items in different work groups cannot cooperate
- OpenCL **host code**: prepare and trigger device code execution: create and manage device context(s) and associate work queue(s), etc…, memory allocations, memory copies, etc. and launch (and compile [at runtime]) the kernel
- OpenCL **HW abstraction**: CPUs, GPUs, … are *devices*, containing one or more *compute units*, which in turn contain one or more SIMD *processing elements*
- An OpenCL **context** contains one or more devices; memory objects are associated with a context, not a device; each device need its own work queue(s); memory transfers are associated with a command queue
- OpenCL **platform**: one host is connected to one or more compute devices; each compute device is composed of one or more units and each unit has one or more processing elements
- Below: structure of a host program; *for details and how to implement this, please refer to exercise 10!*


- OpenCL scalar data types[50] ('u' stands for unsigned): void, cl_[u]char, cl_[u]short, cl_[u]int, cl_[u]long, cl_[u]half, cl_[u]float, cl_[u]double
- OpenCL vector data types ('u' stands for unsigned, $N \in \{2,4,8,16\}$): void, cl_[u]charN, cl_[u]shortN, cl_[u]intN, cl_[u]longN, cl_[u]halfN, cl_[u]floatN, cl_[u]doubleN
- OpenCl object types: cl_platofrm_id, cl_device_id, cl_context, cl_command_queue, cl_mem, cl_program, cl_kernel
- Objects are reference counted and GC[51]'ed
- OpenCL-specific extensions of ANSI-C99: declare functions with __kernel; address spaee classifiers: __global, __constant, __local, __private; access modifiers: __read_only, __write_only, get_global_id()/get_group_id
- Below: OpenCL execution model[52]

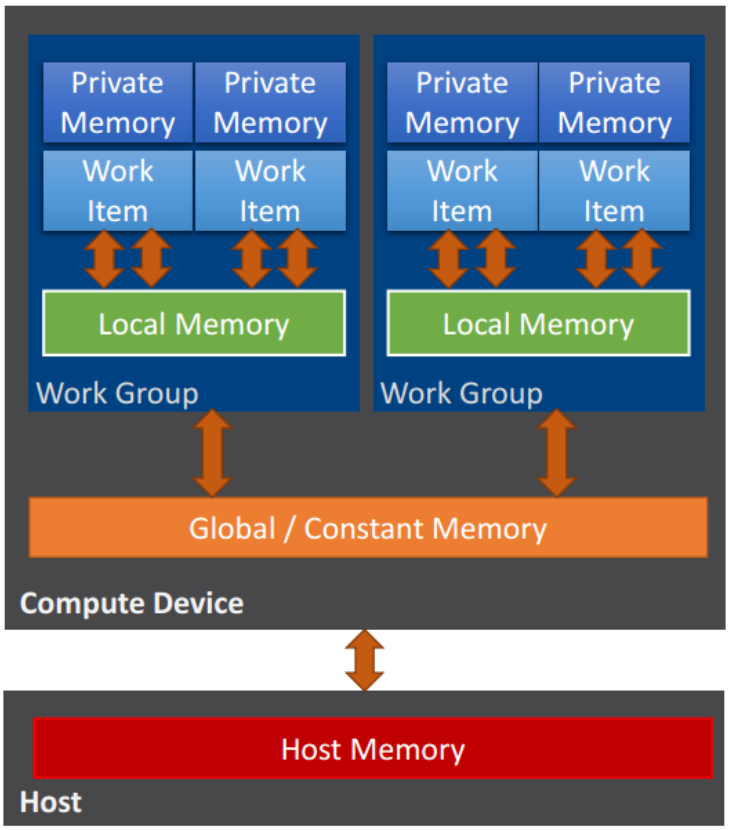
[49] Or other high-level language wrappers such as aparapi for Java
[50] For all data types listing: they are lazy and lack description yet I assume you can figure them out (and they are probably not super-relevant to know by heart anyway).
[51] Garbage collection
[52] Study this figure well, it's important!


## 21 OpenCL Memory Model & Synchronization

- Aside: OpenCL program flow
- Aside: OpenCL memory model
- **Memory**: private is per work item; local is shared within work groups; global/constant is not synchronized; host is on the CPU; memory management is explicit i.e. you have to move data from host to global to local and back
- __global is large, long latency; __constant is read-only constant cache; __local – is accessible form multiple work items, can be SRAM or DRAM; __private are on-chip device registers
- Default address space for arguments and local variables is __private but declaring something __private is illegal




- Program (global) variables must use __constant
- Casting between different address spaces is undefined
- Open CL **execution model**: queues are directly associated with a specifi device; commands execute either in-orde or out of order (they are always queued in-order); explicit synchronization is required for out-of-order quees and multiple command queus with data dependencies




**<synchronization examples>**[53]

- Single device in-order : no problems, one command executes after the prvouos one finishes, memory transaction have a consisten view
- Single device out-of-order: problem: memeory transactions overlap and clobber data
- Separate multi-device (2 contexts): doesn't make any sense (it works, but no benefit at all)! Command quees cannot synchronize across contexts, neither device sees the memory pool of the other unless you clFinish() it and copy across contexts
- Cooperative multi-device (1 combined context): problem since both devices start executing command ASAP and memory transactions overlap and clobber combined memory

[53] Slides 12-28

**</synchronization examples>**

**<synchronization methods>**

- Command queue control methods: brute force, not very fine-grained: clFlush() – send all commands on the queue to the compute device; clFinish() – flush and then wait for commands to finish
- Command execution barriers: clEnqueueBarrier():Enqueue a fence which insures that all preceding commands in the queue are complete before any commands which get enqueued afterwards are processed
- Event based synchronization: *event objects* are unique objects which can be used to determine command status; *event wait lists* are an array of events used to indicate commands which must be complete before further commands are allowed to proceed
- Possible statuses of an event[54]: queued, submitted, running, complete
- All clEnqueue…() methods cat return event objects which then can be used as synchronization points; these methods also accept event wait lists
- Events can be used for profiling

**</synchronization methods>**

**<kernel level synchronization mechanisms>**

- Memory model uses relaxed consistency: state of memory visible to a work item is not guaranteed to be consistent with all work items; if consistency is needed, synchronization is required
- Kernel barrier: barrier(FLAG) where flag is [GLOBAL|LOCAL]_MEM_FENCE
- Kernel fences ensure memory laods and stores order within a work item: mem_fence(), [read|write]_mem_fence()

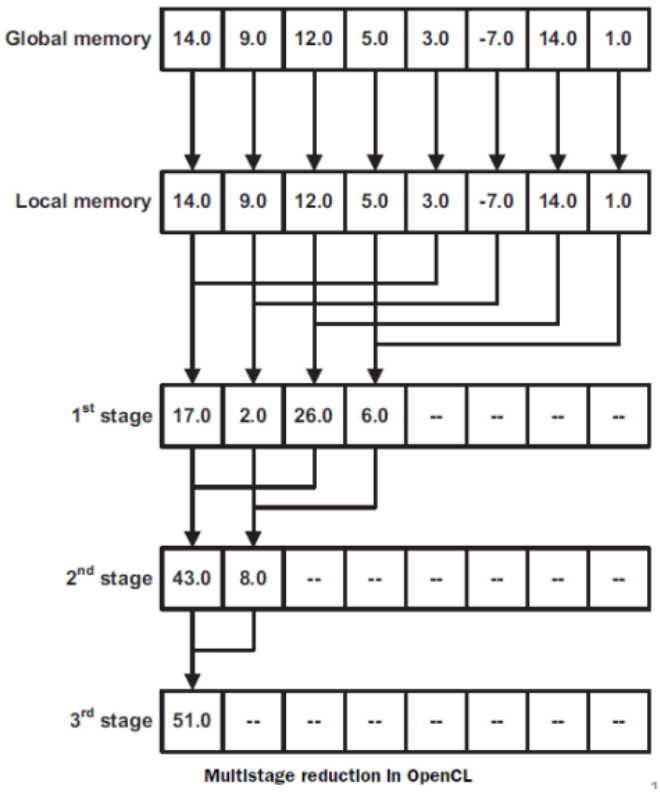
**</kernel level synchronization mechanisms>**

- map/reduce is very well suited for GPGPU/OpenCL

# 22 OpenCL Case Studies

**Vector Types & Operations**

- Vector literals: specifying only one value of a vector results in all elements being this value; you can also specify the values
- You can get the components in the first and second half by v.lo and v.hi (resp.) and the odd position (.odd)
- Vectors can be added to one another; you can apply abs() to get the absolute value of every component
- Using an appropriate vector width results in performance gain (exploiting the hardware)



Multistage reduction in OpenCL

**Numerical Reductions**

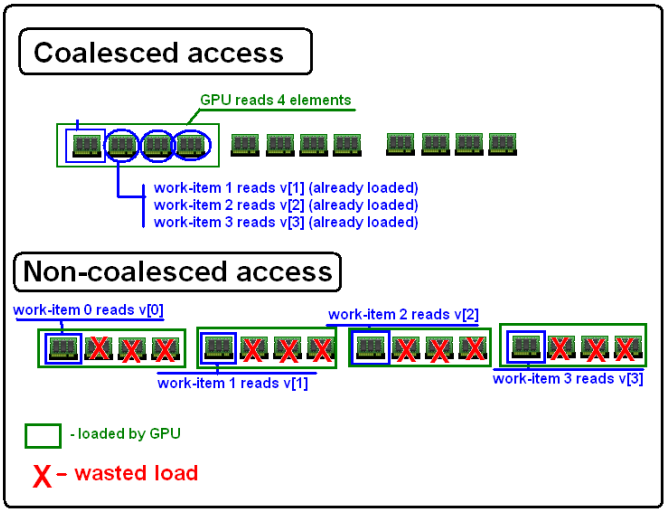
- Sum tree reduction, in-place, $\log n$ times; this OpenCL multistage reduction has the following properties: work items add elements $x^k$ and $x^{k+N/2}$; results are stored in sequence (sum 0 is stored at address 0, sum 1 is stored at address 1, and so on); this storage method reduces memory bank conflicts.
- Workgroup synchronization: no barrier function that synchronizes work-items in different workgroups; so long as a kernel is executing, there's no way to tell when

[54] All capital and prefixed with CL_

any work-group completes its processing; but: once a kernel is finished – we know that the entire work-group is finished; hence: launch multiple-kernels sequentially to synchronize across work groups
- General strategy: Single work-item → work-group → fully occupied device

**Memory Coalescing**

- Memory coalescing relates to how modern GPUs retrieve data from internal memory; memory fetch in global space causes the GPU to read a minimum number of elements (typically somewhere between 32-128bit); memory coalescing access means data access in a sequential, regular pattern: the $i^{th}$ work item should access the $(i+k)^{th}$ component of a __global vector, here $k$ is a constant value
- Not taking these considerations into account can have a huge performance penalty/loss

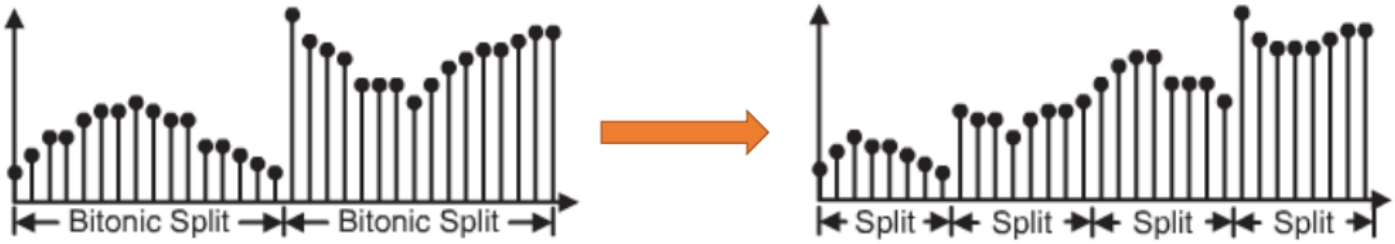



**Parallel Sorting**

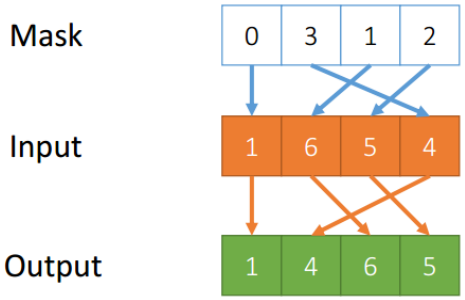
- You cannot (at least when using comparisons) sort faster than $O(n \log n)$, thus $\Omega(n \log n)$; proof is obtained by looking at a binary tree of all possible permutations ($n!$) and its height ($\log n$).
- Bitonic (merge) sort is a parallel sorting algorithm breaking the lower bound on sorting for comparison-based sorting algorithms with $O(\log^2 n)$ parallel time in any (worst, average, best) case
- A bitonic set is a set where the sign of the gradient changes once at most:

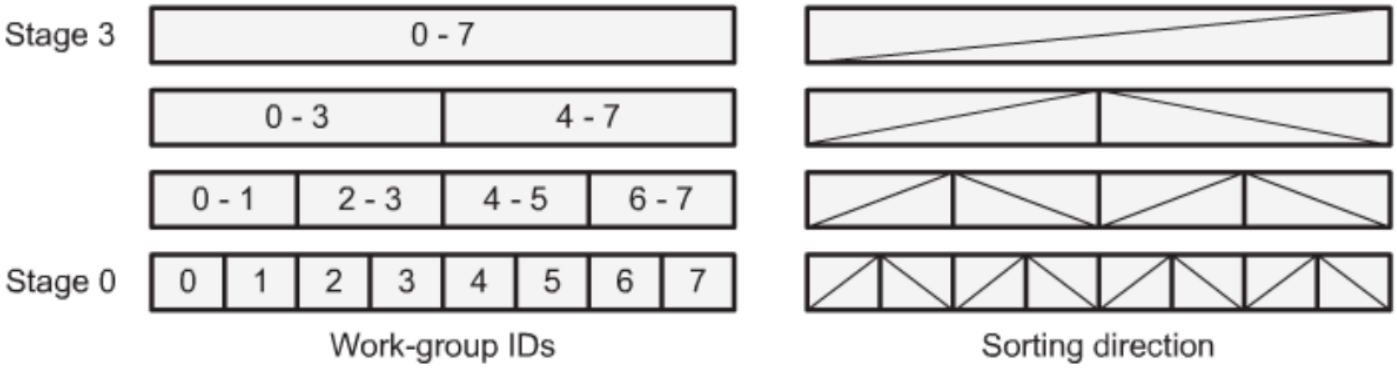
$$x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}, \text{for some } k, 0 \leq k < n$$

- A bitonic sequence is defined as a list with no more than one local maximum and no more than one local minimum.
- Binary split: 1) Divide the bitonic list into two equal halves. 2) Compare-Exchange each item on the first half with the corresponding item in the second half; The result: Two bitonic sequences where the numbers in one sequence are all less than the numbers in the other sequence. Because original sequence was bitonic, every element in the lower half of new sequence is less than or equal to the elements in its upper half.




- Bitonic merge: compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right; Given a bitonic sequence, recursively performing 'binary split' will sort the list.
- Forming bitonic sequences: 1. Pairs of adjacent numbers are already bitonic → merge into larger sequences 2. Compare the first two elements. If the first element is greater than the second, swap the two elements. Ensures that the elements are in ascending order. 3. Compare the second two elements. If the second element is greater than the first, swap the two elements. Ensures that the elements are in descending order.
- The shuffle operation: the shuffle function creates a vector whose components are taken from those in the input vector – according to a mask.
- Full bitonic sort: each work-item sorts 8 data points (as discussed); 1st stage each work-group sorts its own data; 2nd stage results are combined across work-groups; final stage sorts the entire data set with a bitonic merge






## Exercises

**Note** Thinking on your own is encouraged ;)

### Exercise 1

### Exercise 2

- Keep Amdahl's Law in mind
- Know how to draw/analyze/optimize a pipeline and calculate speedup etc.
- Height of a tree is $\log_2 n$ [55]
- Loop: if an element's operation depends on the value of a previous element, you can't parallelize that loop. If an element's operation depends only on its own value, you can parallelize that loop.

### Exercise 3

– Possible takeaway: PrimeSieve: it makes sense to first get rid of all even numbers;

### Exercise 4

– ParallelSieve: create an array of threads, each getting a different section of the sieve and foreach[56]-loop them

### Exercise 5

– **Dining Philosophers**: easy deadlock if every philosopher acquires left fork; cyclic dependency needs to be broken using e.g. lock ordering; to get maximum number of philosophers eating, bundle forks (maximum for five philosophers is two)

– **Banking System**: wrapping every method in "synchronized" will lock the whole system and thereby losing the parallelism (plus a bottleneck is created where many threads try to acquire a lock); Java's intrinsic locks are reentrant[57] (= same thread can acquire same lock multiple times), thus transfer(a, a, x) is not a problem; to prevent deadlocks, lock ordering can be used (e.g. account id); to get a correct sum (incorrect during transactions), lock all accounts before getting its balance and not releasing the lock until *all* accounts are summed up (aka two-phase-locking; one phase were all locks are acquired and no locks are released and another phase where all locks are released and no locks are acquired), do not forget lock ordering; summing up can easily be parallelized since a sum is associative (and more, actually)

### Exercise 6

– Hints/takeaways: static variables are not associated with an object, they are per-class and can be used as such (helpful for the boat); R/W lock: the last reader should notify waiting writers, when trying to acquire a writer lock, first wait for (if applicable) the writer to finish writing and then wait for all readers to finish reading, when releasing the writer lock, notify all waiting threads on the end, save current thread to check for release-without-acquire[58]

[55] Not always, but if you're asked to find some formula, keep log in mind
[56] What I mean to say: for (Thread t : threads) { t.start(); }; this is what I mean by a foreach loop
[57] And of course also explicit locks using ReentrantLock
[58] Thread.currentThread()

## Exercise 7

- Don't access any variable directly, wrap it in a STM call

## Exercise 8

- DRY – Don't Repeat Yourself; reuse your sequential code for the parallel version (+ verifying sequential code is easier)
- Know how merge sort works

## Exercise 9

- Streams are cool and fun! And somewhat similar to SQL
- Make use of the Collectors and predefined functions (math-related)
- Make sure all actors get all necessary signals and make use of the event-based functions (prestart, onReceive, …)

## Exercise 10

- Implement the structure of an OpenCL program (shown in lecture 20)

## Exercise 11

- Game of Life is great! And did I mention you can use colors? :)
- col = global_id % width, row = global_id / width;