#### 7usammenfassung

#### 1 MILL TT-THREADING

Parallelism (Subprograms run simultaneously for faster programs) VS. Concurrency (interleaved execution of pr grams for simpler programs)

Process: Program under Execution, own address space (heavy weight. Pros: Process isolation and responsiveness, Cons: Interprocess communication overhead, expensive in creation, slow context switching and process Thread: Parallel sequence within a process. Sharing the same

address space, but separate stack and registers (lightweight because

Multi-threads: Changes made by one thread to shared resources will be seen by other threads.

Context switch: Required when changing threads. Synchronous (Thread waiting for condition) Or Asynchronous (Thread gets released after defined time)

Multitasking: Cooperative (Threads must explicitly initiate context switches, scheduler can't interrupt) Or preemptive (scheduler can asynchronously interrupt thread via timer interrupt

JVM Thread Model: JVM is a process in the OS. It runs as long as threads are running ( man (town) will not be writed upon). Threads are realized by the thread class and the interface Runnable. Code to be run in a Thread is within a overridden run()

As a anonymous function (Lambda): var myThread = new Thread(() → { /\* thread behaviou \*/ }): mvThread.start():

As a named function: var myThread = new Thread(() → AssyFunct()); myThread.start(); With explicit runnable implementation:

class MyThread implements Runnable { @Override oublic void run() { /\* thread behavior \*/ }} var myThread = n Thread(new MyThread()): myThread.start();}

In C#: var myThread = new Thread(() => { ... }); myThread.Start(); ... myThread.Join();

# Multi-Thread Example (no synchronization) public class MultiThreadTest { public static void main(String[] args) { var a = new Thread(() → multiPrint("A")); var b = new Thread(() → multiPrint("B")); } a.start(); b.start(); System.out.println("main finished"); static void multiPrint(String label) { for (int i = 0; i < 10; i++) { System.out.println(label + ": " + i);</pre> 111

The printout of this function varies. It can be all possible combinations of A's and B's due to the non-deterministic schedul Thread Join: Waiting for a thread to finish (t2. join() blocks as long as t2 is running).

var a = new Thread(()  $\rightarrow$  multiPrint("A")); var b = new Thread(()  $\rightarrow$  multiPrint("B")) System.out.println("Threads start"): a.start(): b.start(): // a.join(); b.join(); System.out.println("Threads joined")

Thread States: Blocked (Thread is blocked and waiting for a monitor lock), New (Thread has not yet started), Runnable (Thread is runnable (Ready to run or running)), Terminated (Thread is terminated), Timed\_Waiting (Thread is waiting with a specified waiting time Thread, sleep (ms)/(hread, join (ms)), Waiting (Thread is waiting) Yield: Thread is done processing for the moment and hints to the scheduler to release the processor. The scheduler can ignore this. Thread enters into ready-state. (Thread wiels(C)) Interrupts: Threads can also be interrupted from the outside (nuThread.interrupt(). Thread can decide

what to do upon receiving an interrupt). If the thread is in the sleep(), wait() or ioin() methods. a InterruptException is thrown. Otherwise a flag is set that can be checked with interrupted()/ isInterrupted()

Exceptions: Exceptions thrown in run() can't be propagated to the Main thread. The exception needs to be handled within the code executed on the thread.

Thread Methods: currentThread(): (Reference to current thread), setTigemon(true): (Mark as doesnos). getId()/getName(): (Get thread ID/Name), isAlive(): (Tests if thread is alive), getState(): (Get thread

## 2 MONTTOR SYNCHRONTZATION

Threads run arbitrarily. Restriction of concurrency for de- class BankAcc terministic behavior nrivate int balance = 0: Communication between threads: Sharing access to fields and the objects they refer to. Efficient, but poses problems: Thread interference and memory consistency

public void deposit(int amount){
 // enter critical section
 synchronized(this) { this.balance += amount: } // exit critical section Critical Section: Part of the code which must be executed

by only 1 thread at a time for the values to stay

synchronized: Body of method with the synchronized keyword is a critical section. Guarantees memory consistency and a happens-before relationship. Impossible for two invocations of a synchronized method on the same object to interleave. Other threads are blocked until the current thread is done with the object. Every object has a Lock (Monitor-Lock). Maximum 1 thread can acquire the lock. Entry of a synchronized method acquires the lock of the object, the exit releases it public synchronized void deposit(int amount) { this halance += amount: } Can also be used within a method, the object that should be locked must be specified.

nized(this) { this.balance += amount: } Exit synchronized block: End of the block, return, unhandled exceptions

## 2.1.1. Monitor Lock

the Monitor There is no shortcut

A monitor is used for internal mutual exclusion. Only one thread operates at a time in Monitor. All non-private methods are synchronized. Threads can wait in Manitor for condition to be fulfilled Can be inefficient with different waiting conditions, has fairness-problems and no shared locks. Recursive Lock: A thread can acquire the same lock through recursive calls. Lock will be free by the last release.

Busy Wait: Running yield or sleep in a loop doesn't release the lock and is inefficient. Use wait. wait(): Waits on a condition. Temporarily releases Monitor-Lock so that other threads can run.

Needs to be wrapped into a while loop to check if the wake up condition has been met. Wakeup signal: Signalling a condition/thread in Monitor. notify() signals any waiting thread (sufficient if all threads wait for the same thing, so it does not matter which one comes next - uniform waiters or if only one single thread can continue like in a turnstile), notifuall() wakes up all threads (i.e. one denosit can satisfy mul tiple withdraws, does not augrantee fairness). If a thread is woken up, it goes from the inner waiting room (waiting on a condition) into the outer waiting room (Thread has not started yet) where it waits for entry to

IllegalMonitorStateException is thrown if notify, notifyAll or wait is used outside synchronized

private int balance = 0; private decimal balance; private object syncObject = new(); bronized void withdraw nublic void Withdraw(decimal amount) { lock (syncObject) { while (amount > balance) { while (amount > balance) { // not if Monitor.Wait(syncObject); wait(): // wait on a conditio halance -- emount: public void Deposit(decimal amount) { lic synchronized void deposit lock(syncObject) { (int amount) { balance += amount; Monitor.PulseAll(syncObject); balance += amount; notifyAll(); // Wakes up all waiting }}} eads in monitor inner waiting area

## 3. SPECIFIC SYNCHRONIZATION PRIMITIVES

#### 3.1 SEMAPHORE

Allocation of a limited number of free recourses is in essence a counter. If a resource is acquired count -- , if a resource is released, count ++ . Can wait until resource becomes available. Can also acquire/release multiple permits at once atomically.

public class Semaphore { private int value: public Semaphore(int initial) { value = initial: } ublic synchronized void acquire() throws In while (value ≤ 0) { wait(); } value—; } public synchronized void release() { value++; notify(): } }

General Semaphore: new Semaphore(N) (Counts from 0 to N for limited permits to access a resource) Binary Semaphore: new Semaphore (1) (Counter 0 or 1 for mutual exclusion, open/locked)

Fair Semaphore: new Semaphore(N, true) (Uses FIFO Queue aging for fairness. Slower than non-fair variant.) Semaphores are powerful, any synchronization can be implemented. But relatively low-level.

class BoundedBuffer<T> { private Oueue<T> queue = new LinkedList (): private Semaphore upperLimit = new Semaphore(Capacity, true); //how many free: private Semaphore lowerLimit = new Semaphore(0, true); // how many full? public void put(T item) throws InterruptedException { upperLimit.acquire(): // No. of free places - 1 chronized (queue) { queue.add(item):} lowerLimit.release(); } // No. of full place
public T get() throws InterruptedException { lowerLimit.acquire(); // No. of full places - 1 ynchronized (queue) { item = queue.remove(): upperLimit.release(); // No. of free places + return item: }}

# 3.2. LOCK & CONDITION

Monitor with multiple waiting lists and conditions. Independent from Monitor locks. Lock-Object: Lock for entry in the monitor (outer wniting room)

Condition-Object: Wait & Signal for a specific condition (inner we

Reentrantl ork: Class in Java alternative to sunchronized. Allows multiple locking operations by the same thread and supports nested locking (Thread is able to re-enter the same lock). Condition: Factors out the Object monitor methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. A Condition replaces the use of the Object monitor methods condition, await(): Throws an InterruptedException if the current thread has its interrupted sta-

tus set on entry to this method or is interrupted while waiting (finally frees the lock in case of interrupt).

## Buffer with Lock & Condition

```
class BoundedBuffer<T> {
  private Oueue<T> queue = new LinkedList ():
  private Lock monitor = new ReentrantLock(true); // fair queue
private Condition nonFull = monitor.newCondition();
private Condition nonEmpty = monitor.newCondition();
   oublic void put(T item) throws InterruptedException {
     monitor.lock(); // Lock queue
try { while (queue.size() = Capacity) { nonFull.await(); }
       queue.add(item); nonEmpty.signal(); } finally { monitor.unlock(); }
       / signalAll() if uniform waiters
       ic T get() throws IntercuntedEvention (
                  k(): // wait for queue to be filled & signal to other queue
     try { while (queue.size() = 0) { nonEmpty.await(); }
        Titem = queue.remove(); nonFull.signal(); return item;
    } finally { monitor.unlock(); } // always release lock, even after Execption
```

3.3. READ-WRITE LOCK	Parallel	Read	Write
Mutual exclusion is unnecessary for read-only threads. So one should allow parallel reading access, but implement mutual ex-	Read	Yes	No
clusion for write access.	Write	No	No

ReadWriteLock rwLock = new ReentrantReadWriteLock(true); // true for fairness
rwLock.readLock().lock(); // shared Lock

rwLock.readLock().unlock(): rwLock.writeLock().lock(); // exclusive Lock

## 3.4 COLINT DOWN LATCH

Synchronization with a counter that can only count down. Threads can wait until counter  $\leq 0$ , or they can count down. The Latches can only be used once.

var ready = new CountDownLatch(N): var start = new CountDownLatch(1): ready.countDown(); // wait for N cars
start.await(); // await race start
start.countDown(); // start the race

# 3.5 CYCLIC BARRIER

Meeting point for fixed number of threads. Threads wait until everyone arrives. Is reusable, threads can synchronize in multiple rounds at the same barrier (Simplifies example above).

var start = new CvclicBarrier(N): start.await(): // all cars race as they're here

# 3.6. EXCHANGER

Rendez-Vous: Barrier with information exchange for 2 parties. Without exchange: new CyclicBarrier(2), with exchange: Exchanger.exchange(something). The Exchanger blocks until another thread also calls exchange(), returns argument x of the other thread.

```
var exchanger = new Exchanger<Integer>();
for (int count = 0; count < 2; count ++) { // odd n.of exch.: last one blocks</pre>
    new Thread(() \rightarrow {
     for (int in = 0; in < 5; in++) {
          int out = exchanger.exchange(in);
          System.out.println(Thread.currentThread().getName() + " got " + out);
       } catch (InterruptedException e) { }
     } ).start(): }
```

#### 4. CONCURRENCY HAZARDS

#### 4.1. RACE CONDITIONS

Insufficiently synchronized access to shared resources. The order of events affects the corre ness of the program. Leads to non-deterministic behavior. Can occur without data race, but data race is often the cause.

Race Condition without data race: The critical section is not protected. Data Race is eliminated using synchronization, but there is no synchronization over larger blocks, so race conditions are

#### 4.2. DATA RACE

Two threads in a single process access the same variable concurrently without synchronization Synchronize Everything? May not help and is expensive. So no.

# 4.3. THREAD SAFETY

Dispensable cases in synchronization: Immutable Classes (Declaring all fields private and final and don't provide setters), Read-only Objects (Read-only accesses are thread-safe,

Confinement: Object belongs to only one thread at a time. Thread Confinement (Object belongs to Thread safe: A data type or method that hehaves correctly when used from multiple threads as if it was running in a single thread without any additional coordination (Java concurrent coll Thread Safety: Avoidance of Data Races. When no sharing is intended, give each thread a private conv of the data. When sharing is important, provide explicit synchronization.

Happens when threads lock each other out, prohibiting both from running. Programs with poten tial deadlock are not considered correct. Threads can suddenly block each other

synchronized(listA) evnchronized(lietB)

synchronized(listB) { evecheonized(listA) listA.addALl(listB);

Both threads in this scenario have locked each other out, the program cannot continue Livelock: Threads have blocked each other permanently, but still execute wait instructions and therefore consume CPU during deadlock

b = false; while(!a) { } ... b = true; a = false; while(!b) { } ... a = true; 4.4.1. Resource Graph account2 Thread T waits for Lock Thread T acquires Lock of Resource R of Resource R **★** 

Deadlocks can be identified by cycles in the resource araph.

order, lock nested only in ascending order. Or use coarse granular locks

(When ordering does not make sense, e.g. black the whole Bank to black all accounts)

A thread never gets chance to access a resource, Avoidance; Use fair synchronization constructs. (Aging Englie fairness in previous supplying constructs. Maging and Thread priorities have a fairness problem.)

## 4.6 PARALIFIISM CORRECTNESS CRITERIA

Safety: No race conditions and no deadlocks, Liveness: No starvation

# 4.7. .NET SYNCHRONIZATION PRIMITIVES

Monitor with sync object: private object sync = new(); lock(sync){ ... }. Uses Monitor.Wait(sync), Monitor.PulseAll(sync). Uses fair FIFO-Queue. Lacks: No fairness flag, no Lock & Condition, Additional: ReadWriteLockSlim for Upgradeable Read/Write, Semaphore. can also be used at OS level. Mutex. Collections are not Thread-safe.

## 5. THREAD POOLS

Threads do have a cost. Many threads slow down the system. There is also a Memory Cost, be cause there is a stack for each thread. Recycle threads for multiple tasks to avoid this. Tasks: Define potentially parallel work packages. Passive objects describing the functionality. Thread Pool: Task are queued. A much smaller number of working threads grab tasks from the queue and execute them. A task must run to completion before a thread can grab a new one. Scalable Performance: Programs with tasks run faster on parallel machines. This allows the exploitation of parallelism without thread costs. The number of threads can be adapted to the system. (Rule of thumb: # of Worker Threads = # processors + 1 (Pendina I/O Calls)) Any task must complete execution before its worker thread is free to grab another task. Exception

nested tasks. Advantages: Limited number of threads (Too many threads slow down the system or exceed available me

Thread recycling (save thread creation and release), Higher level of abstraction (Disconnect task description from execution), Number of threads configurable on a per-system basis. Limitations: Task must not wait for each other (except sub-tasks), results in deadlocks (if one task  $T_1$  is

waiting for something the task  $T_2$  behind him in the Queue should provide, but  $T_2$  waits for  $T_3$  to finish, a deadlock occurs)

# 5.1. JAVA THREAD POOL

var threadPool = new ForkJoinPool(); Future<Integer> future = threadPool.submit(() → { // submit task into pool int value = ...; /\* long calculation \*/ return value;

# 5.1.1. Future<T>

Represents a future result that is to be computed (asynchronous). Acts as proxy for the result that may be not available yet because the task has not finished. Usage via . submit() (submits task into pool and launches task), . get() (waits if necessary for computation to complete and then retrieves its result) and .cancel() (Attempts to concel execution of this task removes it from queue). Task ends when a unhandled exception occurs. It is included in the ExecutionException thrown by get().

Task are started without retrieving results later (submit() without get()). Task is run, but Exceptions do not get catched.

# 5.1.3. Count Prime Numbers

int counter = 0; for (int n = 2; n < N; n++) { if (isPrime(n)) { counter++}};</pre> Recursive extends RecursiveTask<Integer> { //RecursiveAction: no return va private final int lower, upper: public CountTask(int lower, int upper) { this.lower = lower: this.upper = upper:} protected Integer compute() {
 if (lower = upper) { return 0; }
 if (lower + 1 = upper) { return isPrime(lower) ? 1 : 0;} int middle = (lower + upper) / 2; var left = new CountTask(lower, middle); var right = new CountTask(middle, upper); left.fork(); right.fork(); return right.join() + left.join(); int result = new CountTaks(2,N).invoke(); // invokeAll() to start multiple tasks

#### 5.1.4. Pairwise sum (recursive)

```
class PairwiseSum extends RecursiveAction 4
  private final int[] array;
private final int lower, upper;
private static final int THRESHOLD = 1; // configu
  public PairwiseSum(int[] array, int lower, int upper) {
    this.array = array; this.lower = lower; this.upper = upper;
  protected void compute() {
    if (upper - lower > THRESHOLD) f
       int middle = (lower + upper) / 2;
         nokeAll(
new PairwiseSum(array, lower, middle),
         new PairwiseSum(array, middle, upper));
       for (int i = lower: i < unner: i+) {
        array[2*i] += array[2*i+1]; array[2*i+1] = 0;
```

#### 5.1.5. Work Stealing Thread Pool

Jobs get submitted into the alobal queue, which distributes the jobs to the local queues of each worker thread. If one thread has no work left, it can "steal" work from another threads local queue instead of the global queue. This distributes the scheduling work over idle processors.

## 5.2. JAVA FORK JOIN POOL

Special Features: Fire-and-forget tasks may not finish, worker threads run as daemon threads. Automatic degree of parallelism (Default: As much worker threads as Processo

# 5.3. .NET TASK PARALLEL LIBRARY (TPL)

Preferred way to write multi-threaded and parallel code. Provides public types and APIs in System. Threading and System. Threading. Tasks namespaces. Efficient default thread pool (tasks are queued to the ThreadPool, supports algorithms to provide load balancing, tasks are lightweight), has multiple abstruction layers (Tark Parallelization: use tasks explicitly Data Pa tasks implicitly). Asynchronous Programming and PLINO.

Task<int> task = Task.Run(() ⇒ { int total = ...
return total:

Console.Write(task.Result): //Blocks until task is done and returns the result

var task = Task.Run(() ⇒ f var left = Task.Run(() ⇒ Count(leftPart));
var right = Task.Run(() ⇒ Count(rightPart)); return left.Result + right.Result; static Tasksint> Count(...nart) {...}

# Thread 2 5.3.1. Parallel Statements in C#

Execute independent statements potentially Execute loop-bodies potentially in parallel in parallel (Start all tasks, implicit barrier at the end). Parallel Invoke( () ⇒ MergeSort(l,m), () ⇒ MergeSort(m,r)

(Queue a task for each item, implicit barrier at the end), Parallel.ForEach(list. file ⇒ convert(rice)),
Parallel.For(0, array.Length i ⇒ DoComputation(array[i])):

Parallel Loop Partitioning: Loop with lots of quickly executing bodies. It would be inefficient to execute each iteration in parallel. Instead, TPL automatically groups multiple bodies into a single task. There are 4 kinds of partitioning schemes: Range (equally sized), Chunk (partitions start small and grow bigger), Stripe (n-sized partitions, where n "small number, sometimes 1") and Hash partitioning (default: if input is indexable then range partitioning, else chunk partitioning).

# 532 PLINO

LINQ: Set of technologies based on the integration of SQL-like query capabilities directly into C#. PLINQ: Is a parallel implementation of LINQ. Benefits from simplicity and readability of LINQ with the power of parallel programming by creating segments from its data. Analog to Java Stream API.

from book in bookCollection.AsParallel() where book.Title.Contains("Concurrency") select book TSRN // Pandom Or

from number in input list AsParallel() AsOrdered() select IsPrime(number)

## 5.3.3. Thread Injection

TPL adds new worker threads at runtime every time a work item completes or every 500ms. Hill Climbing Algorithm: Maximize throughput while using as few threads as possible. Measures throughput & varies number of worker threads. Avoids deadlock with task-dependencies (but inefwith Thougadon 1 SetMay Thougade (1 are still possible). We should keep parallel tasks short to better profit from this automatic performance improvement.

# ASYNCHRONOUS PROGRAMMING

Unnecessary Synchrony: Blocking method calls are often used without need (Long running color ase or file accesses). With an asynchronous call, other work can continue while waiting on the result of the long operation.

var task = Task.Run(LongOperation); /\* other work \*/ int result = task.Result; Kinds of Asynchronisms: Caller-centric ("pull", caller waits for the task end and gets the result, blocking call), Callee-Centric ("push", Task hands over the result directly to successor / follower task

Task Continuations: Define task whose start is linked to the end of the predecessor task.

// Java (there can be multiple Apply/AcceptAsync calls) CompletableFuture .supplyAsync(() → longOP) // runAsync for return void .Run(task1) ContinueWith(task2): thenApplyAsync(v → 2\*v) // returns value
ContinueWith(task3); thenAcceptAsync(v → ....println(v)); // returns void

Multi-Continuation: Continue when all tasks are finished: Task.WhenAll(task1, task2).ContinueWith(continuation);

Continue when any of the tasks are finished (other threads get lost after first thread is done): Task.WhenAnv(task1, task2).ContinueWith(continuation):

(Exceptions in fire & forget task get ignored, i.e. Task, Run(() ⇒ { ...: throw e: })) Exception Handling: Synchronously Wait() for the whole task-chain at the end. Register for unhearted excentions with TackSchoduler UncheartedTackExcention (Pers

from fire & forget tasks). This should be executed as soon as the task object is dead (Garbage Collector). Java CompletableFuture: Modern asynchronous programming in Java. Also has Multi-Continuation with CompletableFuture.allOf(future1, future2) and CompletableFuture.anv(...)

## 6.1 NON-BLOCKING GUI'S

Use case: If a UI is doing a long task, it should not freeze.

GUI Thread Model: Only single-threading (Only a special UI-thread is allowed to access UI-components). The UI thread loops to process the event queue.



GUI Premise: No long operations in UI events, or else blocks UI. No access to UI-elements by other threads, or else incorrect (Exception in .NET & Android, Race Condition in Javas Swing)

#### 6.1.1. Non-Blocking UI Implementation

```
// C# .NET
void buttonClick( ) {
                                      button.addActionListener(event -
  var url = textBox.Text:
                                         var url = textField.getText():
  Task.Run(() ⇒ {
 var text = Download(url);
                                          var text = download(url); // to worker thread
SwingUtilities.invokeLater(() → {
    Dispatcher.InvokeAsync(() ⇒ {
      label Content = text:
                                            textArea.setText(text): // to UT thread
```

Java invokeLater: To be executed asynchronously on the event dispatching thread. Should be used when an application thread needs to update the GUI.

#### 6.2. C# ASYNC/AWAIT

ronous (until the awaited task is complete)

More readable than the "spaghetti code" in the chapter before. This is the same code as before. async Task<string> DownloadAsync(string url) {

url = textBox.Text; var web = new HttpClient(); text = await DownloadAsync(url); string result = await web.GetStringAsync(url); label.Context = text; return result; 3

asunc for methods: Caller may not be blocked during the entire execution of the async method. wait for tasks: "Non-blocking wait" on task-end / result. Execution Model: Async methods run partly synchronous (as long as there is no blocking await), partly

Mechanism: Compiler disserts method into segments which are then executed completely syn chronously or asynchronously.

Different Execution Scenarios: Case 1: Caller is a "normal" thread (Usual case, Continuation is executed by a TPL-Worker-Thread), Case 2: Caller is a UI-thread (Continuation is dispatched to the UI thread and processes

Async Return Value Types: void ("fire-and-forget"), Task (No return value, allows waiting for end), Task<T>

Async without await: Execute long running operation explicitly in task with await Task.Run()

public async Task<bool> IsPrimeAsync(long number) { return await Task.Run(() => { for (long i = 2; i\*i ≤ number; i++) {
 if (number % i = 0) { return false; } } return true; 1): 1

# 7. MEMORY MODELS

Lock-Free Programming: Correct concurrent interactions without using locks. Use guarantees of memory models. Goal is efficient synchronization.

Problems: Memory accesses are seen in different order by different threads, except when synchronized and at memory barriers (weak consistency). Optimizations by compiler, runtime system and CPU. Instructions are reordered or eliminated by optimization.

Memory model: Part of language semantics, there exist different models: sequential co (SC) (Order of execution cannot be changed. Too strong a consistency model) and the Java Memory Model (a "weak" memory model).

#### 7.1. JAVA MEMORY MODEL (JMM)

Interleaving-based semantics. Minimum warranties: Atomicity, Visibility and Ordering. An atomic action is one that happens all at once (So no thread interference). Java guarantees that read/

#### writes to primitive data types up to 32 Bit. Object-References (strings etc.) and long and double (with volatile keyword) are atomic. A single read/write is atomic. Atomicity does not imply visibility

Guaranteed visibility between threads, Lock Release & Acquire (Memory writes before release are visible after acquire). Volatile Variable (Memory writes up to and including the write to volatile variables are visible when eading the variable), Thread/Task-Start and Join (Start: input to thread; Join: thread result), Initialization of final variables (Visible after completion of the constructor), final fields.

# 7.1.3. Ordering

Java Happens Before: "Happens before" defines the ordering and visibility augrantees between actions in a program. It ensures that changes made by one thread become visible to others. An unlock of a monitor happens-before every subsequent lock of that same monitor.

Java Ordering Guarantees: Writes before Unlock → reads after lock, volatile write → volatile read, Partial Order. Synchronization operations are never reordered. (Lock/Unlock, volatile-accesses

## 7.2. SYNCHRONIZATION

Rendez-Vous: Primitive attempt to synchronize threads.

volatile boolean a = false, b = false; volatile bool a = false, b = false; a = true; while( !b ) { ... } a = true; Thread.MemoryBarrier(); while (!b) { ... } b = true: while(!a) { ... } h = true: Thread.MemoryBarrier(); No reordering because a and b are volatile. while (!a) { ... }

## Spin-Lock with atomic Operation

public class SpinLock { private final AtomicBoolean locked = new AtomicBoolean(false): public void acquire() { while( locked.getAndSet(true) ) {...} }
public void release() { locked.set(false); }

Java Atomic Classes: Classes for boolean, Integer, Long, References and Array-Elements. Differen kinds of atomic operations, addAndGet(), getAndAdd() etc. Operations on atomic data classes: boolean getAndSet(bo

Lambda-Variants: AtomicInteger s = new AtomicInteger(2): s.undateAndGet(x \rightarrow x \times x):

Volatile Write

Volatila Read

boolean compareAndSet(boolean expect, boolean update) Sets update only when read value is equal to expect. Returns true when successful. Optimistic Synchronization:

do { oldV = v.get(); newV = result; } while(!v.compareAndSet(oldV, newV));

Reordering in one direction still possible. Volatile Write: Release seman-

#### 7.3. .NET MEMORY MODEL Main differences to JMM: Atomicity (long/double also not atomic with volatile),

Atomically sets to the given value and returns the previous value

Ordering and Visibility (only half and full fences). Atomic Instructions with the 7.3.1. Half Fence (Volatile)

# executed before the write). Volatile Read: Acquire semantics (Subsequent memory accesses are not moved a

7.3.2. Full Fence (Memory Barrier) Disallows reordering in both directions, Thread, MemoryBarrier():

tics (Preceding memory accesses are not moved below it, but later operations can be

Page 1

# ParProg | FS24 | Nina Grässli & Jannis Tschan

#### 8. GPU (GRAPHICS PROCESSING UNIT)

End of Moores Law: We can no longer gain performance by "growing" sequential processors. Instead, we improve performance by running code in parallel on multi-core (CPUs) (Low Latency) and many-core or massively parallel co-processors (GPUs) (high throughput).

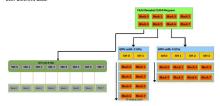
GPU's are specialized electronic circuits designed to accelerate the computation of computer graphics. They are faster than CPUs for suitable algorithms on large datasets. Useful for calculations which consist of multiple independent sub-calculations, not very useful for calculations where the results rely on the previous results (like Fibonacci).

High Parallelization: A CPU offers few cores (4, 8, 16, 64) and is very fast, Programming is easier, A GPU offers a very large number of cores (512, 1024, 3584, 5760) and has very specific slower processors. It is optimized for throughput. Programming is more difficult.

GPU Structure: A GPU consists of multiple Streaming Multiprocessors (SM) which in turn consist of multiple Streaming Processors (SP) (e.g. 1-30 SM, 8-192 SPs per SM).

SIMD: Single Instruction Multiple Data. The same instruction is executed simultaneously on multiple cores working on different data elements (Vector parallelism). Saves fetch & decode instructions. SISD: Single Instruction Single Data. Purely sequential calculations.

SIMT: Single Instruction Multiple Threads. The same instruction is executed in different threads



# 8.1 LATENCY VS. THROUGHPLIT

Latency: Elapsed time of an event (Walking from point A to B takes one minute, the latency is one minute) Throughput: The number of events that can be executed per unit of time (Bandy

There is a tradeoff between latency and throughput. Increasing throughput by pipelined processing, latency most often also increases. All pipeline stages must operate in lockstep. The rate of processing is determined by the slowest sten.

#### Pipelining: Run processes in an overlapping manner

Example: A program consists of two operations: Transfer data from CPU memory to GPU memory  $(T_1 \text{ units} = 20ms)$ . Execute computation on the device  $(T_2 \text{ units} = 60ms)$ . What is the *latency* (non-pipelined)? 20 + 60 = 80 ms.

What is the throughput (pipelined)? Every 60ms an operation is finished Throughput =  $\frac{1}{co}$  operations/ms.

# 8.2. CPU VS GPU

CPUs	GPUs	
Low latency     Few but optimized cores     General purpose     Architecture and Compiler help to run any code fast	Can execute highly parallel data operations     Simple but a lot of cares with cache per core     very useful for problems which consist of a lot of in     pendent data elements     Efficiency must be achieved by optimizing the progra	
Aim: low latency per thread	Aim: high throughput	

# 8.3. NUMA MODEL

NUMA stands for Non-Uniform Memory Access. CPUs on host and GPU devices each have local memories. There is no common main memory between the two, so explicit transfer between CPU and GPU is needed. There is also no garbage collector on the GPU.

# 84 CLIDA

Computer Unified Device Architecture, Is a parallel computing platform and an API for Nvidia GPU that allows the host program to use GPUs for general purpose processing.

## CUDA Execution

1. cudaMalloc: GPU memory allocate 4. cudaMemopu: Transfer results from GPU to cuda/Hemcpy: Data transfer to GPU (Host ToDevice)
 CPU (DeviceToHost)
 Kernel <<<1, N>>>: Kernel execution
 CPU (DeviceToHost)
 cudaFree: Deallocate GPU memory

## Example: Array addition

for (int i = 0; i < N; i++) { C[i] = A[i] + B[i]; } // sequential

(i = A .. N-1): Cfil = Afil + Rfil: // narallel using n threads

## **CUDA Kernel**

A kernel is a function that is executed n times in parallel by m different CUDA threads.

```
// kernel definition on GPU
 nlohal
 oid VectorAddKernel(float *A, float *B, float *C) {
  int i = threadIdx.x; C[i] = A[i] + B[i];
VectorAddKernel<<<1, N>>>(A, B, C); // N is amount of threads
```

Only the GPU knows when the task is finished.

## Boilerplate Orchestration Code

void CudaVectorAdd(float\* h A. float\* h B. float\* h C. int N) { size\_t size = N \* sizeof(float);
float \*d\_A, \*d\_B, \*d\_C; // data on GPU
cudaMalloc(&d\_A, size); cudaMalloc(&d\_B, size); cudaMalloc(&d\_C, size); cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice); mcpv(d B, h B, size, cudaMemcpvHostToDevice): VectorAddKernel«<1, N>>>(d\_A, d\_B, d\_C, N); cudaHemcpy(h\_C, d\_C, size, cudaHemcpyDeviceTo cudaFree(d\_A); cudaFree(d\_B); cudaFree(d\_C);

# 8.5. PERFORMANCE METRICS

The performance is either limited by memory bandwidth or comnutation. Compute Bound: Throughout is limited by calculation (Cores are at the limit, but the memory bus could transfer more data). This is hetter and reached if Al Kernel > Al GPU Memory Bound: Throughput is limited by data transfer (Memory bus

handwidth is at its limit, but cores could Arithmetic intensity: Defined as FLOPS (Floating Point Operations per . Second) per Byte. The higher, the better.

Number of operations  $\_$  FLOPS Number of transferred bytes Bytes

•  $t_{compute} < t_{memory}$ Example for (i=0: i< N. i++) {  $z[i] = x[i] + v[i] + x[i]: }$ Read x and y from memory, write z to memory. That's 2 reads and 1 write (x is used twice but read only once). In case x, y and z are ints, we have 12 (3-4) bytes transferred and 2 arithmetic ops (+, \*). The arithmetic intensity is therefore  $\frac{2}{3} = \frac{1}{3}$ .

Compute Bound •  $t_{compute} > t_{memory}$ 

•  $\frac{\#ops}{BW_{compute}} > \frac{}{B}$  $BW_{memory}$  $\frac{\#ops}{\#bytes} > \frac{BW_{compute}}{BW_{res}}$ 

## Memory Bound

#ops <

 $BW_{compute}$ 

Copy needed, but other new rules. aMallocManaged(&A, size); // ... same for &B and &C

A[0] = 8; ... // initialize A and B assuming they reside on the host // A and B are automatically transferred to the device  $\frac{\#ops}{\#bytes} < \frac{BW_{compute}}{BW_{memory}}$ std::cout « C[0];

Provides performance estimates of a kernel running on differently sized architectures. Has three parameters: Peak performance, peak bandwidth vs. arithmetic intensity. rmance is derived from benchmarking FLOPS or GFLOPS (Giga-FLOPS, 10° FLOPS). The peak bandwidth from manuals of the memory subsystem. The ridge point is where the horizontal and diagonal lines meet = minimum AI required to achieve the peak performance.

# π 1/4 1/2 1 2 4 8 16 32 64 128

# 9. GPU ARCHITECTURE

Because there are so many cores on GPUs, it is possible to run many threads in parallel without context switches. This allows better parallelism without a performance penalty.

Just-in-time Compilation: The NVCC compiler compiles the non-CUDA code with the host C compiler and translates code written in CUDA into PTX instructions (assembly language represented as ASCII text). The graphics driver compiles the PTX into executable binary code. The assembly of PTX code is postponed until application runtime, at which time the target GPU is known. The disadvantage of this is the increased application startup delay. However, thanks to cache this only happens

Programming Interface: Runtime (The cudart library provides functions that execute on the host to (de-)allocate device memory, transfer data etc.) OT driver API (The CUDA driver API is implemented in the cuda. dll or cuda. so which is copied on the system during installation of the driver. This provides an additional level of control by exposing lower-level

Asynchronous Execution: The command pipeline in CUDA works asynchronous, commands and data can be transferred from/to the GPU at the same time

#### 9.2. CUDA SIMT EXECUTION MODEL

Single instruction, multiple Threads, The kernel is executed N times in parallel by N different CUDA threads.

Blocks: Threads are grouped in blocks. The host can define how many threads each block has (up to 1024). Threads in one block can interact with each other but not with threads in other blocks. Execution Model: One thread runs on one virtual scalar processor (one GPU core). One block runs on one virtual multi-processor (one GRU Streaming Multiprocessor). Blocks must be independent Thread Pool Abstraction: The compiled CUDA program has e.g. 8 CUDA blocks. The runtime can

choose how to allocate these blocks to multiprocessors. For a larger GPU with 8 SMs, each SM gets one CUDA block. This enables performance scalability without code changes. Guarantees: CUDA guarantees that all threads in a block run on the same SM at the same time

and that the blocks in a kernel finish before any block from a new, dependent kernel is started. Mapping: One SM can run several concurrent blocks depending on the resources needed. Each kernel is executed on one device. CUDA supports running multiple kernels on a device at one time.

## 9.3. CUDA KERNEL SPECIFICATION

Specifying Kernel: VectorAddKernel <<< GRID\_dimension, BLOCK\_dimension>>>> (A,B,C) ns can be 1D, 2D or 3D and specified via dim3 which is a structure designed for storing block and grid dimensions; struct dim3{x: v: z}. dim3 dimGrid(2) == dim3 dimGrid(2,1,1) (Unassigned o

VectorAddKernel <<<di>dimBrid. dimBlock>>>(A.B.C): Number of blocks in a grid: dimGrid.x \* dimGrid.y \* dimGrid.z

Number of threads in a block: dimBlock.x \* dimBlock.v \* dimBlock.z

1D Grid: We can simply use integers. VectorAddKernel <<<1, N>>> creates 1 Block with N Threads. 2D Grid: dim3 gridS(3.3): dim3 blockS(3.3): VectorAddKernel << gridS, blockS>>> 3D Grid: dim3 gridS(3.2.1): dim3 blockS(4.3.1): VectorAddKernel <<<or>
 blockS
 Device Limits: Max threads per block: 1024, Max thread dimensions per block: (1024, 1024, 64) Max arid size: (2'147'483'647, 65'535, 65'535)

## Calculation Examples

VectorAddKernel <<<dim3(8,4,2), dim3(16.16)>>>(d A. d B. d C): Amount of Blocks:  $8 \cdot 4 \cdot 2 = 64$ Amount of threads per block:  $16 \cdot 16 = 256$ Threads in total: 64 , 256 - 16'384

If we have 1024 threads in a block, how many blocks are needed to launch N threads? int blocksPerGrid = (N + threadsPerBlock - 1) /

threadsPerBlock; Need to round up because for 1025 threads we need 2 blacks

1 \* 2 \* 1 2 \* 2

## 9.4. DATA PARTITIONING WITHIN THREADS

Data Access: Each kernel decides which data to work on. The programmers decide data partitioning scheme, threadId, x/y/z (Thread no. in block), blockId, x (Block no.), blockDim, x (Block size) Partitioning in Blocks:

C[i] = A[i] + B[i]: // without this if, some threads will be idle N = 4897: int blockSize = 1824: int gridSize = (N + blockSize - 1) / blockSize: VectorAddKernel <<< gridSize, blockSize >>> (A, B, C);

Boundary Check: More threads than necessary work on the data. If N = 4097, 5 Blocks with 1024 Threads are needed which results in 1023 unused threads. Threads with i > N must not be al lowed to write to array C because they might corrupt the working memory of some other thread.

## 9.5. FRROR HANDLING

Some functions have return type cudaError. Need to check for cudaSuccess. It's best to write your own helper function and wrap every fucking line in it. E.g. handleCudaError() which prints the error and exits the program.

## 9.6. UNIFIED MEMORY

Unified memory allows automatic transfer from CPU to GPU and vice versa. No explicit Memory

// C is transferred automatically to the host and can be read directly cudaFree(A); cudaFree(B); cudaFree(C);

#### 10. GPU PERFORMANCE OPTIMIZATIONS

Hardware: A scalable array of multithreaded Streaming Multiprocessors (SMs), the threads of a thread block execute *concurrently* on one multiprocessor, multiple *thread blocks* can execute concurrently on one multiprocessor. When thread blocks terminate, new blocks are launched on the free multiprocessors.

## 10.1. MATRIX ADDITION

```
__global__
void MatrixAddKernel(float *A, floa *B, float *C) {
    int column = blockIdx.x * blockDim.x + threadIdx.x:
     int row = blockfdx.y * blockDim.y * threadIdx.y;
if (row < A_ROWS && col < A_COLS) { // boundary checking
        C[row * A_COLS + col] = A[row * A_COLS + col] * B[row * A_COLS + col];
const int A_COLS, B_COLS, C_COLS = 6;
const int A_ROWS, B_ROWS, C_ROWS = 4;
dim 3 block = (2,2); dim3 grid = (3,2)
MatrixAddKernel << grid.block >>> (A.B.C):
```

#### 10.2. MATRIX MULTIPLICATION

 $\textbf{Parallelization:} \ \textbf{Every Thread computes one element of the result matrix} \ C. \ \textbf{Can be parallelized}$ because results do not depend on each other

global oid multiply(float +A floa +B float +C) { int i = blockIdx.x \* blockDim.x + threadIdx.x; int j = blockIdx.y \* blockDim.y + threadIdx.y; if (i < N && j < M) { // boundary checking float sum = A for (int k = A: k < K: k++) { sum += A[i \* K + k] \* B[k \* M + j]; C[i \* M + j] = sum;

#### 10.3. MAPPING THREADS / BLOCKS TO GPU WARPS

Warns: Blocks are split into warns (1 Warn = 32 Threads) and all threads within execute the same code If there aren't enough threads to fill a warp, "empty" threads are launched. A number of warps constitutes a thread block. A number of thread blocks are assigned to a Streaming Multiproce sor. The whole GPU consists of several SM.

Thread blocks are scheduled in parallel or sequentially. Once a thread block is launched on a SM. all of its warps are resident until their execution finishes. Therefore, a new block on a SM is not launched until there is a sufficient number of free registers and shared memory for all warps of the new block

date all warps of a block, but only a subset is running in parallel at the same time (1 to 24). Divergence: Not all threads of a warp may branch the same way. The branches do not run simulily, so the other threads need to wait until one branch is finished. So branches within one warp should be avoided because of performance problems (Branches are born at if / switch / )

if (threadIdx.x > 1) { } else { } if (threadIdx.x / 32 > 1) { } else { }

DRAM (Dynamic Random Access Memory): Global memory of a CUDA device is implemented with DRAMs. If a GPLI kernel accesses data from consecutive locations, the DRAMs can supply the data at a much higher rate than if a random sequence of locations were accessed. Memory Coalescing: Thread access natterns are critical for performance. If the threads in a warn

simultaneously access consecutive memory locations, their reads can be combined into a single access (burst). Else there are expensive individual accesses. Coalesced Accesses: Read/Write the burst in one transaction per warn burst section, swapped

read/write within the same burst, only individual elements in the burst accessed Not Coalesced Accesses: Read/Write in different warp bursts, one action that spans multiple

Coalescing in Use: data[(Expression without threadId.x) + threadId.x] Coalescing with Matrices: Matrices get linearized to a 1D array. The row of the matrix should be the longer side so that there are as many coalescing accesses as possible

All threads have the access to the same global memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block (Has higher bandwidth and lower latency than local or global memory but longer latency and lower bandwidth than registers which are private to a thread). Each thread has private local memory (in device memory, high latency and low bandwidth, same as global). Constant, texture and surface memory also reside in device memory.

Memory Hierarchy: Shared Memory (per SM. fast, shared between threads in 1 block, a few KB. shared float x), Global Memory ("Main Memory" in GPU Device, slow, accessible to all threads, in GB, cudaMalloc()) Registers (private to a thread fastest but very limited storage)

Constant memory: Constant variables are stored in the global memory but are cached. Shared Memory Declaration: With keyword shared . A static array size is necessary. Limited

memory, around 48KB. Multidimensionality is allowed. Fast Matrix Multiplication: By reducing global memory traffic. Partition data into subsets called tiles which fit into shared memory (the row & column that should be multiplied and the result cell). The kernel computation on these tiles must be able to run independently of each other. Because the shared memory is limited, load the tiles in several steps and calculate the intermediate result from this

\_global\_\_void MatrixMulKernel(float\* d\_M, float\* d\_N, float\* d\_P, int Width) {
 \_shared\_\_float Mds[TILE\_WIDTH][TILE\_WIDTH]; shared float Nds[TTLE WIDTH][TTLE WIDTH]: int bx = blockIdx.x; int by = blockIdx.y; int tx = threadIdx.x; int ty = threadIdx.y; the d P element to work on int Row = by \* TILE WIDTH + ty: int Col = bx \* TILE\_WIDTH + tx; float Pvalue = 0;
// loop over d\_M and d\_N tiles required to compute d\_P element for (int m = 0: m < Width/TILE WIDTH: ++m) { Most(ty][tx] = d\_M[Row=Width + m=TILE\_WIDTH + tx];
Mos[ty][tx] = d\_M[Row=Width + m=TILE\_WIDTH + tx];
Mos[ty][tx] = d\_M[[cm=TILE\_WIDTH + ty]=Width + Col]; \_\_syncthreads(); // CUDA equivalent to for (int k = 0; k < TILE\_WIDTH; ++k) { Pvalue += Mds[ty][k] \* Nds[k][tx] d P[Row:Width + Coll = Pvalue: ids() is only allowed in if/else if all threads of a black choose the same branch, otherwise undefined be

11. HIGH PERFORMANCE COMPUTING (HPC) CLUSTER PARALLELIZATION

Cluster programming is the highest possible parallel acceleration (Factor 100 and more). Used for general purpose programming, lots of CPU cores. Combination of CPUs and GPUs possible. Computer Cluster: Network of powerful computing nodes, firmly connected at one location. Very

fast interconnect (like 100GBit/s), used for big simulations (Fluids, Weather, Traffic, etc.) SPMD: This is the most commonly used programming model, "high level". Single Program (All tasks ecute their conv of the same program simultaneously). Multiple Data (all tasks may use different data). The MPI program is started in several processes. All processes start and terminate synchronously. Synchronization is done with harriers

MPMD: Also a "high level" programming model. Multiple Program (Tasks may execute different programs ously). Multiple Data (all tasks may use different data)

Hybrid Memory Model: All processors in a machine can share the memory. They also can request data from other computers. (non-uniform memory access; not all accesses take the same tin

Message Passing Interface (MPI): Distributed programming model. Is a common choice for Parallelization on a cluster, Industry-Standard libraries for multiple programming languages. MPI Model: Notion of processes (Process is the running program plus its data), parallelism is achieved by running multiple processes, co-operating on the same task. Each process has direct access only to

its own data (variables are private). Inter-Process-Communication by sending and receiving messages. SPMD in MPI: All processes run their own local copy of the program & data. Each process has a unique identifier, processes can take different paths through the program depending on their IDs. Usually, one process per core is used (to maximize the benefit of parallelization).

Formalizing Message: A message transfers a number of data items from the memory of one process to the memory of another process (Typically contains ID of sender and receiver, data type to be sent. Communication modes: Point to Point (very simple, one sender and one receiver. Relies on matching send and

receive calls) and Collective communications (between groups of processes. Broadcast: one to all, Scatter: Split data and send each chunk to different node. Gather: Collect the chunks back at the originating node.

#### 11 1 MPI ROII FRPI ATE CODE

int main(int argc, char \* argv[]) {
 MPI\_Init(&argc, &argv); // MPI Initialization int rank; int len; nk(MPI\_COMM\_WORLD, &rank); // Process Identification char name[MPI\_MAX\_PROCESSOR\_NAME]; MPI\_Get\_processor\_name(name, &len); printf("MPI process %i on %s\n", rank, name); return 0: } MPT Init: Must be the first MPI call. Allows the mni init to broadcast to all the processes. Does

not create processes, they are only created at launch time. All MPI global and internal variables are constructed. A communicator is formed around all the processes that were spawned an unique ranks (IDs) are assigned to each process, MPI\_COMM\_WORLD encloses all processes in the iob. Communicator: Group of MPI processes, allows inter-process-communication.

NPT Comm nank: Returns the rank of a process in a communicator Used for sender/receiver IDs MPI\_Comm\_size: Returns the total number of processes in a communicator.

MPI\_Finalize: Is used to clean up the environment. No more MPI calls after that. MPT Reprier: Blocks until all processes in the communicator have reached the barrier. Compila-

tion & Execution: mpicc HelloCluster.c && mpiexec -c 24 a.out && sbatch -hi.sub Process Identification: Rank = number within a group, incremental numbering from 0. Unique Identification = (Rank, Communicator) MPI\_Send(void \* data, int count, MPI\_Datatype datatype, int destination, int tag,

MPI\_Comm communicator) // tag: freely selectable number for msg type (≥ 0) MPI Recv(void \* data, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm

communicator, MPI\_Status\* status) // status: error information Fach send should have a matching receive.

#### Example direct communication:

PI\_Send(&value, 1, MPI\_INT, receiverRank, tag, MPI\_COMM\_WORLD); MPI\_Recv(&value, 1, MPI\_INT, senderRank, tag, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); Array send: int array[LENGTH];

MPT Send(array, LENGTH, MPT INT, receiverRank, tag. MPT COMM WORLD): MPI\_Recv(array, LENGTH, MPI\_INT, senderRank, tag, MPI\_COMM\_WORLD, MPI\_STATUS\_IGN); MPI Boast: Is efficient, because the root node does not send the signal individually to each node. the other nodes help spread the message to others. (signal spreads like corona): MPI\_Bcast(void data, int count, MPI\_Datatype datatype, int root, MPI\_Comm\_World communicator)

MPI\_Reduce: Reduction is a classic concept: reducing a set of numbers into a smaller set of numbers via a function (e.a. [1, 2, 3, 4, 5] ⇒ sun ⇒ 15). Each process contains one integer, MPI Reduce is called with a root process of 0 and using MPI SUM as the reduction operation. The four numbers are added and stored on the root process. Job is done in a distributed in MPT Reduce(void\* send data, void\* recy data, int count. MPT Datatyne datatyne. MPI\_Op op, int root, MPI\_Comm comm) (send\_data: array of elements of type datatype to reduce from

each process, pecu\_data: relevant on the root process, contains the reduced result and has a size of size of (datatype) \* count. op the operation you wish to apply to your data: MPI\_MAX\_MPI\_MIN\_MPI\_SUM\_MPI\_PROD: multiplies all, MPI\_BAND/ MPI\_LAND: Bitwise/Logical AND, MPI\_LOR: Logical OR, MPI\_MAXLOC: Same as max plus rank of process that owns it) MPI\_AllReduce: Many parallel applications require accessing the reduced results across all processes. This function reduces the values and distributes the result to all processes. Does not

need a root node. This is an implicit broadcast to all processes MPI\_Allreduce(void\* send\_data, void\* recv\_data, int count, MPI\_Datatype datatype, MPT On on. MPT COMM comm)

#### MPI\_Gather: Gather together multiple values from different processors. I\_Gather(&input\_value, 1, MPI\_INT, &output\_array, 1, MPI\_INT, 0, MPI\_COMM\_WORLD)

11.2. APPROXIMATION OF  $\pi$  VIA MONTE CARLO SIMULATION Draw a circle inside of a square and randomly place dots in the square. The ratio of dots inside the circle to the total number of dots will approximately equal  $\frac{\pi}{4}$ .

long count hits(long trials) { long hits = 0, i; for (i = 0; i < trials; i++) { double x = (double)rand()/RAND\_MAX; double y = (double)rand()/RAND\_MAX; if  $(x * x + y * y \le 1)$  { hits++;} // distance to center is bigger } return hits; }

//Parallel, the trials are split across different nodes int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); srand(rank \* 4711); // each process receives a different seed
long hits = count\_hits(TRIALS / size); // Each process computes a subtask long total: MPI\_Reduce(&hits, &total, 1, MPI\_LONG, MPI\_SUM, 0, MPI\_COMM\_WORLD);
if (rank = 0) { double pi = 4 \* ((double)total / TRIALS); }

# 12. OPENME

Node: A standalone "computer in a box". Usually comprised of multiple CPU/processors/cores. memory, network interfaces etc. Nodes are networked together to comprise a supercomputer. Each node consists of 20 cores. The processes do not share memory, they must use messages. Threads: Default are 24 on a single Node in OST cluster. Can be set with onp\_set\_num\_threads() or with the OMP\_NUM\_THREADS environment variable. Threads range from 0 (master thread) to N-1. HPC Hybrid memory model: Run a program on multiple nodes. No Shared Memory (NUMA) hetween Nodes. Shared Memory (SMP) for Cores inside a Node. OpenMP: Is a programming model for different languages, Allows to run multiple threads, dis-

tribute work using synchronization and reduction constructs. Shared Memory (shared memory process consists of multiple threads), Explicit Parallelism (Programmer has full control over parallelization) and Compiler Directives (Most OpenMP parallelism is specified through the use of compiler directives (pragmas) in the source code).

#### Fork and Join #include <stdio.b:

the company of the main(int argc, char\* argv[]) {
 const int np = omp\_get\_max\_threads(); // executed by initial thread
 printf("OpenMP with threads %d\n", np); // executed by initial thread #pragma omp parallel // pragma spawns multiple threads (fork) const int np = omp\_get\_num\_threads(); // executed in parallel
printf("Hello from thread %d\n", omp\_get\_thread\_num()); // executed in parallel } // order not fixed, after execution, threads synchronize and terminat

#### For loops

oma omo narallel for

Each thread processes one loop-iteration at a time. Execution returns to the initial threads. Oversubscription (too many threads for a problem) is handled by OpenMP. The iteration variable (i.e. i) is im plicitly made private for the duration of the loop.

#### Memory Model

pragma omp parallel for private (A) shared (B) firstprivate (C) for(...) int A, B, C //an

Each thread has a private copy of A and use the same memory location for B. C is also private, but gets its initial value from the global variable. After the loop is over, threads die and both A and B will be cleared/removed from memory.

#oragma omn narallel ically private because inside of pragma #pragma omp for ...

# Avoiding Race conditions: Mutex

int sum =  $\theta$ ; #pragma omp parallel for for (int i = 0: i < n: i++) #pragma omp critical { sum += i; } // only one thread at a time

This is extremely slow due to serialization, slower than single threading. Critical section is overkill for this code, with a heavy weight mutex the performance overhead is large.

# Lightweight mutex: Atomic

int sum = 0; int i #pragma omp parallel for for (i = 0; i < n; i++)
#pragma omp atomic { sum += i }</pre>

#### Reduction across threads

When using reduction (operator: variable), a copy of the reduction variable per thread is created, initialized to the identity of the reduction operator (+ = 0.4 = 1). Each thread will then reduce into its local variable. At the end of the parallel region, the local results are combined into the alobal variable. Only associative operators allowed (+. + not -. /).

// Code using the reduction clause int sum = A: pragma omp parallel for reduction(+: sum)
for (int i = 0; i < n; i++) { sum += i; }</pre> The same code without the reduction clause int sum = 0: ma omp parallel { int intermediate\_sum = 0; // private #pragma omp for
 for (int i = 0; i < n; i++) { intermediate\_sum += i; } // thread partial sum</pre> pragma omp atomic // reduction is final\_sum += intermediate\_sum; }

Hybrid: OpenMP + MPI numprocs, rank; int iam = 0, np = 1; I\_Init(&argc, &argv);
I\_Comm\_size(MPI\_COMM\_WORLD, &numprocs); MPT Comm cank (MPT COMM WORLD Scank) a omp parallel default(shared) private(iam, np) { omp\_get\_num\_threads() iam = omn get thread num(): printf("I am I %d out of %d from P %d out of %d\n", iam, nn, rank, numprocs);

# Sequential count\_hits from Section 11.2 in OpenMP

t\_hits(long trials) { long hits = 0; long i; double x,y; #pragma omp parallel { #oragma omp for reduction(+:hits) private(x.v) for (i = 0; i < trials; i++) {
 double x = random\_double(); double y = random\_double();
 if (x \* x + y \* y < 1) { hits +; } return hits: }

## 13. PERFORMANCE SCALING

Difficulties with parallel programs: Finding parallelism, granularity of a parallel task, moving data is expensive, load balancing, coordination & synchronization, performance debugging. Scalability: The ability of hard- and software to deliver greater computational power when the number of resources is increased.

Scalability Testing: Primary challenge of parallel computing is deciding how best to break up a problem into individual pieces that can be computed separately. It is impractical to develop and test large applications using the full problem size. The problem and number of processors are scaled down at first. Scalability testing: measuring the ability of an application to perform well or better with varying problem sizes and numbers of processors. It does not test the applications general functionality or correctness.

## 13.1 STRONG SCALING

The number of processors is increased while the problem size remains constant. Results in a reduced workload per processor. Mostly used for long running CPU bound applications. Amdahls Law: The speedup is limited by the fraction of the serial part of the software that is not amenable to parallelization. Sweet spot needs to be found. It is reasonable to use small amounts

of resources for small problems and larger quantities of resources for big problems T= total time, p= part of the program that can be parallelized., N= amount of processors  $T = (1 - p)T + T_p = T_s + T_p$  $T_N = T_n/N + (1-p)T$ , Speedup  $\leq 1/(s+p/N)$ , Efficiency =  $T/(N \cdot T_N)$ 

Amdahls law ignores the parallel overhead. Because of that, it is the upper limit of speedup for a problem of fixed size. This seems to be a bottleneck for parallel computing. Examples: 90% of the computation can run parallel, what is the max speedup with 8 processors

 $1/(0.1 + 0.9/8) \approx 4.7$ 25% of the computation must be serial. What is the max speedup with ∞ Processors?  $1/(0.25 \pm 0.75/\infty) \approx 1/0.25 - 4$ 

To gain a 500 × speedup on 1000 processors. Amdahls law requires that the proportion of serial part cannot exceed what?  $500 = 1/(s + \frac{1-s}{1000}) \Rightarrow s + \frac{1-s}{1000} = \frac{1}{500} \Rightarrow 1000s + (1-s) = 2 \Rightarrow 999s = 1 \Rightarrow s = \frac{1}{999} \approx 0.1\%$ 

## 13.2. WEAK SCALING

The number of processors and the problem size is increased. Mostly used for large memory bound applications where the required memory cannot be satisfied by a single node. Gustafson's law: Based on the approximations that the parallel part scales linearly with the amount of resources, and that the serial part does not increase with respect to the size of the

problem. Speedup =  $s+p\cdot N = s+(1-s)\cdot N$ In this case, the problem size assigned to each processing element stays constant and additional elements are used to solve a larger total problem. Therefore, this type of measurement is justification for programs that take a lot of memory or other system resources.

Example: 64 Processors. 5% of the program is serial, What is the scaled weak speedup?  $0.05 + 0.95 \cdot 64 = 60.85$