Context switching

- Synchron: Waits for condition
  - Queues itself as waiting and gives processor free
- Asynchron: Timing
  - After a defined time, the thread should release the processor
  - Prevents a thread from permanently occupying the processor

Thread states

Running, Waiting, Ready

JVM

- Scheduling of threads handled by the OS
- The current thread can be accessed with `Thread.currentThread()`

```
var t1 = new Thread(() ->
{ System.out.println("Hi from t1"); })
t1.setDaemon(true) // Stops running when main-
thread is finished
t1.start();
```

Interrupts

When `t2.interrupt()` is called, the thread doesn't terminate directly. It only stops when t2 calls `join`, `wait` or `sleep`.

Java Thread Lifecycle

- Blocked
- New
- Runnable
- Terminated
- Timed_Waiting: `sleep(timeout), join(timeout)`
- Waiting: `join()`

If `wait`, `notify` and `notifyAll` are called outside synchronized blocks: IllegalMonitorStateException.

When is a single notify sufficient?

Both must hold:

1. Only one semantic condition (uniform waiters):
   - Condition interests every waiting thread
2. Change applies to only one
   - Only one single thread can continue

Lock with conditions

```
private Lock monitor = new ReentrantLock(true);
private Condition nonFull =
monitor.newCondition();
nonFull.signal();
nonFull.signalAll();
nonFull.await();
```

Read-Write Locks

```
ReadWriteLock rwLock = new
ReentrantReadWriteLock(true);
rwLock.readLock().lock();
rwLock.writeLock().unlock();
```

Monitor vs. Locks + Conditions

- Monitor:
  - Simplicity, no complex wait-notify logic
  - Performance is critical
- Locks + Conditions:
  - More control over synchronisation required (e.g. fair locking)
  - More fine grained control on which Threads to wake up (instead of all)

Race Conditions

Race condition without Data Race

Synchronization can be skipped, if:

- Immutability is used/Read-Only Objects
- Confinement (Einsperrung): Objects belong to only one thread at a time

Confinement

- Thread Confinement: Object belongs to only one thread
- Object Confinement: Object is encapsulated in already synchronized objects

Threadsafe Java collections

Old Java-Collections like `Vector`, `Stack`, `Hashtable` are threadsafe. Modern collections (`HashSet`, `TreeSet`, `ArrayList`, `LinkedList`, `HashMap`, `TreeMap`) are **not** threadsafe. → `ConcurrentHashMap`, `ConcurrentLinkedQueue`, `CopyOnWriteArrayList`.

Concurrent collections have strong concurrency guarantees, but have weakly conisistent iterators! There's no `ConcurrentModificationException` and concurrent updates are likely not seen by others.

Deadlock avoidance

- Linear lock hierarchy
- Coarse (grob) granular locks:
  - Only one lock holder; e.g. entire bank is blocked while lock holder does work
- Partial order to the acquisition of mutexes: Any pair { M1, M2 } are always locked in the same order.

Starvation

```
do {
  success = account.withdraw(100);
} while(!success)
```

Correctness Criteria

- No raceconditions
- No deadlocks
- No starvation

.NET

An exception in a thread leads to the program to stop. Threads can be made daemon threads by calling `t.IsBackground = true`.

- `Monitor.Wait(obj)`
- `Monitor.PulseAll(obj)`

Concurrency at scale

Many Threads slow down the system:

- Longer time intervals in between threads
- Many thread start/stop
- Number limited
- Memory:
  - Stack for each thread
  - Full register backup at swap

Tasks

Tasks try to solve the problem of threads. They define potentially parallel work packages, they are purely passive objects describing the functionality. Tasks can run in parallel, but they don't have to.

#worker-threads = #processors + #pending IO-calls

Limitations

Tasks must run to completion, before its worker thread is free to grab another task.

Task must not wait for each other (except subtasks), otherwise potential deadlock (because current task in queue depends on the work of the next task in queue)

Java

```
var threadPool = new ForkJoinPool();
Future<Integer> future = threadPool.submit(() -
> {
  int value = ...;
  return value;
});
Integer result = future.get(); // blocking
```

`future.cancel(boolean mayInterruptIfRunning)`: Will fail, if task completed, cancelled or cannot be cancelled for some other reason

Recursive Task

```
class CountTask extends RecursiveTask<Integer>
{
  @Override
  protected Integer compute() {
    var left = new CountTask(lower, middle);
    var right = new CountTask(middle, upper);
    left.fork();
    right.fork();
    return left.join() + right.join();
  }
}
// ...
var threadPool = new ForkJoinPool();
int result = threadPool.invoke(new CountTask(2,
N))
```

Default Pool: `ForkJoinPool.commonPool()`:
`int result = new CountTask(2, N).invoke();`

Avoid Over-Parallelizing

```
protected Integer compute() {
  if (upper - lower > THRESHOLD) {
    // parallel count
  } else {
    // sequential count
  }
}
```

Special features

- Fire and forget might not finish (Worker threads are daemon threads)
- Automatic degree of parallelism

.NET

```
Task<int> task = Task.Run(() => {
  var left = Task.Run(() => Count(leftPart));
  var right = Task.Run(() => Count(rightPart));
  return left.Result + right.Result; //
task.Result is blocking
});
```

Parallel statements

```
Parallel.Invoke(
  () => MergeSort(l, m),
  () => MergeSort(m, r),
);
```

Parallel Foreach

```
Parallel.ForEach(list,
  file => Convert(file)
);
```

Parallel For

```
Parallel.For(0, array.Length,
  i => DoComputation(array[i])
);
```

Task Continuations

```
Task.Run(LongOperation)
  .ContinueWith(task2)
  .ContinueWith(task3)
  .Wait();
```

Multi-Continuation

```
Task.WhenAll(task1, task2)
  .ContinueWith(continutation);
```

Java

```
CompletableFuture
  .supplyAsync(() -> longOp())
  .thenApplyAsync(v -> 2 * v)
  .thenAcceptAsync(v -> println(v));
```

GUIs

Java

```
button.addActionListener(event -> {
  var url = textField.getText();
  CompletableFuture.runAsync(() -> {
    var text = download(url);
    SwingUtilities.invokeLater(() -> {
      textArea.setText(text);
    })
  })
})
```

.NET

```
void buttonClick() {
  var url = textBox.Text;
  Task.Run(() => {
    var text = Download(url);
    Dispatcher.InvokeAsync(() => {
      label.Content = text;
    })
  })
}
```

Or simpler with async/await:

```
var url = textBox.Text;
var text = await DownloadAsync(url);
label.Content = text;
```

If the thread is an UI thread, the part after the `await` instruction is guaranteed to be ran by the UI thread (instead of the separate Task where the await is ran).

Memory model

Visibility

Atomicity does not imply visibility! One thread may not see updates of another thread at all (or possibly much later). Guaranteed visible between threads are:

- Lock release & acquire:
  - Memory writes before release are visible after acquire
- Volatile variable
  - Memory writes up to including the volatile variable are visible when reading the variable
- Thread/Task-Start and join
  - Start: input to thread, Join: thread result
- Initialization of final variables
  - Visible after completion of constructor

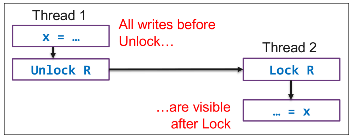

Figure 1: Visibility lock → unlock

Visibility also implies partial order.

```
volatile boolean a = false, b = false;

// thread 1:
a = true;
while(!b) {}

// thread 2:
b = true;
while(!b) {}
```

This code works, no reordering is done because of `volatile` → total order

Atomic operations

getAndSet(): Returns old value, writes new value.

```
public class SpinLock {
  private final AtomicBoolean locked = new
AtomicBoolean(false);
  public void acquire() {
    while(locked.getAndSet(true)) {}
  }
  public void release() {
    locked.set(false);
  }
}
boolean compareAndSet(boolean expect, boolean
update)
```

- Sets update only if read value is as expected (atomic)
- Returns true if successful

Lock free stack (Treiber 1986)

```
var top = new AtomicReference<Node<T>>();
void push(T value) {
  var newNode = new Node<T>(value);
  Node<T> current;
  do {
    current = top.get();
    newNode.setNext(current);
  } while(!top.compareAndSet(current,
newNode));
}
```

.NET

- Volatile Write: Release semantics
- Volatile Read: Acquire semantics

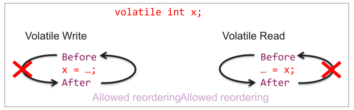

Figure 2: volatile semantics in .NET

Memory Barrier

To prevent reordering, we need to use `Thread.MemoryBarrier();`

```
volatile bool a = false, b = false;
// thread 1:
a = true;
Thread.MemoryBarrier();
while(!b) {}
// thread2:
b = true;
Thread.MemoryBarrier();
while(!a) {}
```

Cluster Programming

- Highest possible parallel acceleration
- Lots of CPU cores (instead of GPU cores)
- GPU often limiting because of SIMD
- Nodes close to each other
- Fast interconnect

Programming models

SPMD

- Single program, multiple data
- high level programming model
- Most commonly used for multi-node clusters

MPMD

- Multiple Program: Tasks may execute different programs simultaneously. Can be threads, message passing, data parallel or hybrid.

Memory Model: Hybrid Model

- Most modern supercomputers use a hybrid architecture (shared + distributed)
- All processors can share memory
- Can also request data from other computers (programmatically)

Message Passing Interface (MPI)

- Distributed programming model
- Industry standard (C, Fortran, .NET, Java, etc.)
- Process: Program + Data
- Multiple processes, working on the same task
- Each process only has direct access to its own data
- Usually one process per core

Message

- Id of sender
- Id of receiver
- Data type to be sent
- Number of data items
- Data itself
- Message type identifier

Compiling and running

```
mpicc HelloCluster.c
mpiexec -n 24 a.out # or -c 24 or -np 24
```

Send/receive

```
MPI_Send(void* data, int count, MPI_Datatype
datatype, int destination, int tag, MPI_Comm
communicator);
MPI_Recv(void* data, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm
communicator, MPI_Status* status)
```

`MPI_Barrier(MPI_COMM_WORLD)` blocks until all processes in the communicator have reached the barrier.

```
MPI_Reduce(&value, &total, 1, MPI_INT, MPI_SUM,
0, MPI_COMM_WORLD);
MPI_Allreduce(&value, &total, 1, MPI_INT,
MPI_SUM, MPI_COMM_WORLD);
```

Gather

```
MPI_Gather(&input_value, 1, MPI_INT,
&output_array, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

SIMD Vector extensions

Data Types and instructions for the parallel computing on short vectors (64 up to 512 bits). Easy to implement on chip.

Java Vector API

Features

- `Add()`, `Sub()`, `Div()`, `Mul()`
- `And()`, `Or()`, `Not()`
- Compare
- Casting
- Shuffle (important for encryption algorithm (rot13))

Info

- Platform agnostic
- Compiled to vector hardware instructions, if supported
  - Fallback: scalar code

```
private static final VectorSpecies<Integer>
SPECIES = IntVector.SPECIES_PREFERRED;

public static int[] vectorComputation(int[] a,
int[] b) {
  var c = new int[a.length];
  int upperBound = SPECIES.loopBound(a.length);
  int i = 0;
  for(; i < upperBound; i += SPECIES.length())
{
    var va = IntVector.fromArray(SPECIES, a,
i);
    var vb = IntVector.fromArray(SPECIES, b,
i);
    var vc = va.add(vb);
    vc.intoArray(c, i);
  }
  for (; i < a.length; i++) { // Cleanup loop
    c[i] = a[i] + b[i];
  }
}
```

OpenMP

```
pragma omp parallel
{
  const int np = omp_get_num_threads();
  const int thread_num = omp_get_thread_num();
} // here, the threads synchronize and
terminate (join)
```

Number of threads can be set via `omp_set_num_threads()`, through the env-variable `OMP_NUM_THREADS`, and they are numbered from 0 (master) to n - 1.

Parallel for loop

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  printf("Iteration %d, thread %d\n", i,
omp_get_thread_num());
}
```

- Launches multiple threads
- Each thread handles one iteration at a time
- Oversubscription (n > omp_get_max_threads()) is handled by OpenMP

Memory model

```
int A, B;
#pragma omp parallel for private (A) shared (B)
  for (...)
```

Or (for private):

```
#pragma omp parallel
int A;
#pragma omp for
for (...)
```

Each thread gets a private copy of variable A, but all threads access the same memory location for variable B.

After the loop, threads terminate and A will be cleared from memory.

Race conditions with shared variables

```
const int n = 300;
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
  sum += i;
}
```

We can avoid race conditions with a Mutex:

```
const int n = 300;
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n; ++i)
#pragma omp critical
{
  sum += i;
}
```

Lightweight Mutex

```
const int n = 300;
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n; ++i)
#pragma omp atomic
{
  sum += i;
}
```

However atomic only works with simple expressions (r/w/arithmetics)

Reduction across threads

```
int sum = 0;
#pragma omp parallel for reduction (+: sum)
for (int i = 0; i < n; i++) {
  sum += i;
}
```

This returns the correct answer without synchronizing the code. The trick here is, that each thread calculates a partial sum. The partial sums are then later summed up atomically.

Hybrid OpenMP + MPI

```
int numprocs, rank;
int iam = 0, np = 1;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#pragma omp parallel default(shared)
private(iam, np) {
  np = omp_get_num_threads();
  iam = omp_get_thread_num();
  printf("Hello from thread %d out of %d from
  process %d out of %d\n"
  , iam, np, rank,
  numprocs,);
}
MPI_Finalize();
```

Performance scaling

Scalability

- Ability to handle more work as the size of the computer/program grows
- Widely used to describe the ability of hardware and software to deliver greater computational power when the number of resources is increased

Strong scaling

- Number of processors is increased while problem size remains constant
  - Reduced workload per processor
  - Individual workload must be kept high to keep processors occupied
- Used for long running CPU bound applications

Amdahls Law (strong scaling)

- Justification for programs that take long to run (CPU bound)
- Goal: Find sweet spot that allows computation to complete in a reasonable amount of time, while not wasting too many cycles due to parallel overhead
- **Harder to achieve good strong-scaling at larger process counts since the communication overhead for most algorithms increase in proportion to the processors used.**

Mathematical definiton of Amdahls law

- Speedup $= \frac{T}{\frac{pT}{N}+(1-p)T} = \frac{1}{s+\frac{p}{n}}$
- Efficiency $= \frac{T}{NT_N}$

Gustafsons Law

- Weak scaling mostly used for large memory bound applications

$$\text{Speedup} = s + pN$$

Latency vs Throughput

Pipelining

- Latency: How long does it take to execute a task from start to end: 120 minutes for a laundry
- Throughput: Number of tasks completed per second or per minute: 1/60 laundry per minute
- Transferring data from memory to device: 20ms
- Executing instructions on device: 60ms
- Latency = Time required to finish one operation = 80ms, resp. 120ms
- Throughput: Every 60ms an operation is finished. Throughput = 1/60 operations/ms

There is a tradeoff between latency and throughput. A high throughput by pipelining processing, the latency most often increases too. Rate of processing is determined by the slowest step.

If the compute time is longer → function is compute limited/compute bound. If the memory time is longer → memory limited/memory bound.

If an operation is memory bound, tweaking parameters to more efficiently use CPU is ineffective.

Operational intensity

$$\frac{\text{operations per second}}{\text{bytes per second}} = \frac{\text{FLOPs}}{\text{Bytes}}$$

If the IO is high, we have a more efficient utilization of modern parallel processors.

Roofline model

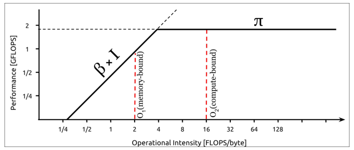

Figure 3: Roofline model

Attainable Perf =
min(Peak Perf, Peak Memory Bandwidth × Operational Intensity)

GPUs

SIMD is essentially vector parallelism.

NUMA Model

NUMA: Non-Uniform Memory Access

CUDA

```
__global__
void VectorAddKernel(float *A, float *B, float
*C) { // GPU (Device)
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int CudaVectorAdd(float* h_A, float* h_B,
float* h_C, int N) { // CPU (HOST)
  size_t size = N * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc(&d_A, size);
  cudaMalloc(&d_B, size);
  cudaMalloc(&d_C, size);

  cudaMemcpy(d_A, h_A, size,
cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size,
cudaMemcpyHostToDevice);

  VectorAddKernel<<<1, N>>>(A, B, C);

  cudaMemcpy(h_C, d_C, size,
cudaMemcpyDeviceToHost);

  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}

int main() {
  CudaVectorAdd(a, b, c);
}
```

CUDA Execution model

- Thread = Virtual Scalar Processor
- Block = Virtual Multiprocessor
- Blocks must be independent
- Each block contains (usually) 1024 threads, each thread has an ID
- Threads & Blocks must complete
- All threads in a block run on the same SM at the same time

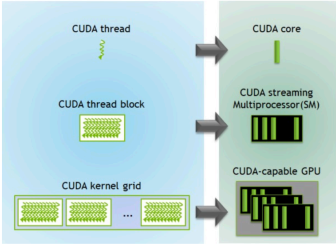

Figure 4: CUDA Architecture

```
dim3 gridSize(3, 2, 1);
dim3 blockSize(4, 2, 1);
VectorAddKernel<<<gridSize, blockSize>>>(...);
```

Product of blockSize cannot be greater than 1024! e.g. 33 x 32 x 1 is not allowed an results in an error.

Max x and y dimensions are 1024, max z dimension is 64.

Data partitioning

```
__global__
void VectorAddKernel(float *A, float *B, float
*C) {
  int i = blockIdx.x * blockDim.x +
threadIdx.x;
  if (i < N) {
    C[i] = A[i] + B[i];
  }
}
// ...
N = 4097;
int blockSize = 1024;
int gridSize = (N + blockSize - 1)/
blockSize; // ceiling
VectorAddKernel<<<gridSize, blockSize>>>(A, B,
C, N);
```

Error handling

```
cudaError error;
error = cudaMalloc(&d_A, size);
if (error != cudaSuccess) {
  char * errStr = cudaGetErrorString(error);
}
```

Unified Memory

Unified memory allows automatic memory transfer from CPU to GPU:

```
A = (float *)malloc(size);
B = (float *)malloc(size);
C = (float *)malloc(size);

vectorAdd(A, B, C, N);

free(A);
free(B);
free(C);

// ...
cudaMallocManaged(&A, size);
cudaMallocManaged(&B, size);
cudaMallocManaged(&C, size);

VectorAddKernel<<<..., ...>>>(A, B, C, N);
cudaDeviceSynchronize(); // Wait for GPU to
finish

cudaFree(A);
cudaFree(B);
cudaFree(C);
```

VectorAdd on a multi dimensional grid

```
__global__
void VectorAddKernel(float *A, float *B, float
*C) {
  int col = blockIdx.x * blockDim.x +
threadIdx.x;
  int row = blockIdx.y * blockDim.y +
threadIdx.y;

  if (row < A_ROWS && col < A_COLS) {
    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;
dim3 block = (2,2);
dim3 grid = (3,2);
VectorAddKernel<<<grid, block>>>(A, B, C);
```

Warps

Blocks are allocated internally in warps of 32 threads each. So a block may have 32 warps at max. All threads in a warp execute the same instruction. The SM executes instructions of one branch (same instruction) in parallel, the other branches have to wait. This can be a performance problem.

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

The global memory of CUDA devices is implemented using DRAMs. DRAMs parallelize data access, and if data is accessed, different data close to that single entry are accessed too really fast. If we can achieve consecutive accesses to data close to each other, we can gain a significant speedup. → Memory coalescing. This is called a memory burst.

Therefore it's crucial, how we align the items in memory and where we run the threads (row vs column-first algorithm).

We should always try to redesign access as follows:

```
data[(expression without threadIdx.x) +
threadIdx.x]
```

So we go linearly through the data.

Register spilling

Variables in a thread are usually stored in registers. If we have too many variables for the registers to hold, the variables are put on the global memory → Register spilling (slow).

We can save variables in shared memory with:

```
__shared__ float x;
```

Only 48 KB. So for example in the matrix multiplication, it makes sense to store chunks of data in the shared memory (tiled matrix multiplication).

In tiled matrix multiplication we need `__syncthreads()` to avoid data races.