Moore's Law

Transistor count doubles every two years.

Different types of parallelism

- Hyperthreading: Two Reg-Sets per core
- Multi-Core: Multiple Cores per CPU
- Multi-Processor: Multiple CPUs per machine
- Compute-Cluster

Parallelism	**Concurrency** (Nebenläufigkeit)
Decomposition of a program into several sub programs, which run simultaneously on several processors ⇒ Faster Programs	Interleaved (time shared) execution that accesses shared resources ⇒ Simpler programs. Sometimes with time slicing (but not necessarily).

Process	**Thread**
Heavyweight, OS only needs process context to run a program correctly, own address space.	Lightweight, a process can have multiple threads, parallel sequence in a program, same address space, separate stack and registers.

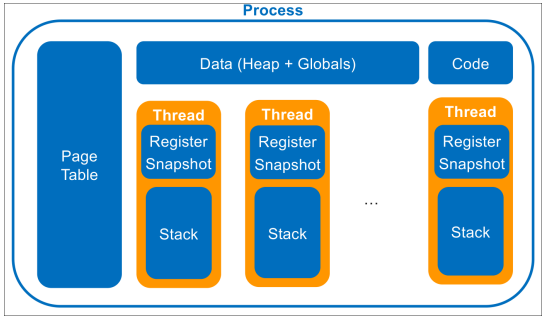

Figure 1: Memory utilization/resources

Multiple threads can write in the same memory locations ⇒ Needs explicit synchronization.

Multiplexing Interleaved execution by using context switching.

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

Multitasking (scheduling)

- Cooperative (rarely used nowadays)
  - Threads must explicitly initiate context switches at intervals
- Preemptive (nowadays standard)
  - Scheduler interrupts the running thread asynchronously via timer interrupt
  - Time sliced scheduling: Each thread has the processor for maximum time interval

Thread states

Running, Waiting, Ready

JVM

- Single process system
- Runs as long as threads are running (not until main() is done!), unless marked as daemon-thread
- `System.exit() / Runtime.exit()`: Uncontrolled stop of all threads
- Threads realized by `Thread`-class and `Runnable`-interface
  - `void run()` can be overridden for custom behaviour
- `thread.start()` starts a Thread, JVM calls `run()` (do not call run manually!)
- If exception is unhandled, other Threads still continue
- Just like any other application threads
- Scheduling of threads handled by the OS
- Allows setting priorities to threads → still managed by OS
- The current thread can be accessed with `Thread.currentThread()`

Non-deterministic

Threads run without any rules, interleaved or parallel. Many JVMs execute System Outputs without interleaving (but not specified!).

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

Instead of passing a lambda to `new Thread()` we can naturally also create a class, implement `Runnable` and pass it instead (functional interfaces).

Alternatively we can also just derive a custom class from `Thread` (not recommended).

Join

With `t2.join()`, we can block t1 as long as t2 is running.

Interrupts

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

Join currentThread

If `Thread.currentThread().join()` is called, the thread ends up in a deadlock.

Java Thread Lifecycle

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

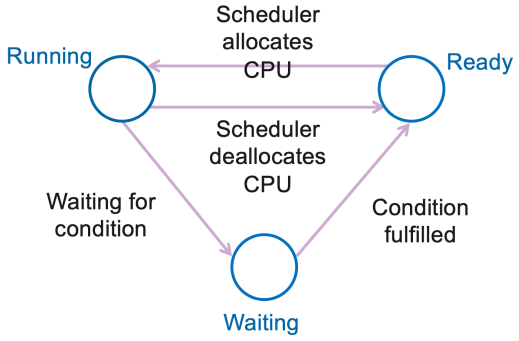

Figure 2: Thread states

Synchronization Restriction of concurrency

Challenges of communication between threads

- Thread Interference
- Memory consistency errors

Critical section Section that needs to be executed by only one thread at a time (atomic)

A java method can be forced to execute mutually exclusive with `synchronized`:

```
public synchronized void deposit(int amount) {
  this.balance += amount;
}
// or:
public void deposit(int amount) {
  synchronized(this) {
    this.balance += amount;
  }
}
```

A lock is acquired at the start of the synchronized block and freed:

- After the block
- At return
- If an unhandled exception is thrown

Nested locks are possible, but if locked on the same object is technically rendundant.

Wait

`wait()` temporarily releases the monitor.

```
public synchronized void withdraw(int amount)
                    throws InterruptedException {
  while (amount > this.balance) {
    wait();
  }
  this.balance -= amount;
}
```

1. Go to Waiting room
2. Free Monitor
3. (Inactive, till wake up)
4. Gain lock again

Figure 3: Wait

notify vs notifyAll

```
class BankAccount {
  private int balance = 0;
  public synchronized void withdraw(int amount)
  throws InterruptedException {
    while (amount > balance) {
      wait();
    }
    balance -= amount;
  }
  public synchronized void deposit(int amount)
{
    balance += amount;
    notifyAll(); // notifyAll needed because
notify only alerts one thread and others might
still want to withdraw too
  }
}
```

A while is necessary because with an if, the thread doesn't check the condition again after being woken up. If `wait`, `notify` and `notifyAll` are called outside synchronized blocks: IllegalMonitorStateException.

Spurious wakeup A thread wakes up without being notified, interrupted or timing out (rare in practice)

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

Semaphore

- `acquire()` (P):
  - if count <= 0: Wait
    - Thread dormant until
      - other thread invokes `release` and the thread is the next one
      - Thread is interrupted (Throws InterruptedException)
- `release()` (V):
  - Free a permit
  - Increment counter
  - **No requirement that a thread must have acquired a permit to release it**

`new Semaphore(N, true)`: FIFO-queue for fairness, slower than (potentially) unfair variant

Java doesn't use OS semaphores → expensive

`acquire(int permits)`, `release(int permits)` for multiple acquires, releases at once

Lock with conditions

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

A specific condition can be either single signaled with `cond.signal()`, or multi signaled with `cond.signalAll()`. Wait with `cond.await()`.

Read-Write Locks

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

Data Races

- Happens when at least one threads writes while others read (single process) and
- Threads aren't using exclusive locks

Race condition without Data Race

```
account.setBalance(account.getBalance() + 100);
```

This can still lead to lost updates, even when account is synchronized, due to non atomic increment.

Race Conditions and Data Races

	Race Condition	**No Race Condition**
Data Race	Erroneous behaviour	Program works correctly, but formally incorrect
No Data Race	Erroneous behaviour	Correct behaviour

Synchronize everything?

- Expensive
- Other concurrency problems still exist
- Cache invalid, Optimization hindrances, ...

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

A datatype/method is *threadsafe*, if it behaves correctly when used from multiple threads, without requiring additional coordination from the caller. Therefore a threadsafe callee cannot put any synchronization requirements on the caller.

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.

Deadlocks

Nested Locks

```
synchronized(listA) {
  synchronized(listB) {
    listB.addAll(A);
  }
}

synchronized(listB) {
  synchronized(listA) {
    listA.addAll(B);
  }
}
```

Livelocks

Livelocks are deadlocks which still execute wait instructions and consume CPU.

```
b = false;
while(!a) {
}
b = true;

a = false;
while(!b) {
}
a = true;
```

Deadlock avoidance

- Linear lock hierarchy
- Coarse 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)
```

Starvation is a fairness problem and depends on scheduling.

Starvation avoidance

- Enforce fairness

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 starts/stops
- 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
  }
}
```

Work Stealing

A free worker thread can steal scheduled tasks of another worker-thread.

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

Graphical user interfaces should run on their own UI thread.

Premises

- No long running operations in UI thread
- No access to UI-elements by other threads
  - .NET/Android: Exception
  - Java Swing: Race Condition

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;
```

An async method runs synchronously until it reaches its first `await`-expression → Method is suspended until the awaited task is complete. All code right *after* the `await`-expression runs asynchronously. All code up to including the first `await`-expression is called synchronously. At the `await`-expression, the thread returns to the caller, while the rest of the function is run asynchronously.

With a UI thread as the caller thread, the model is a bit different. Here, the await call is run in a different TPL thread (normal), but after its completion, the rest of the function is passed back to the UI thread again (so we can update the UI afterwards).

`void` should only be used in event handlers, only `Task` can be waited on. Async functions also have no support for `ref` and `out` parameters.

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

Async/Await thoughts

- One Async method, two scenarios (non UI thread vs. UI thread)
- Viral effect: Caller must also be async
  - Complicates debugging
  - Runtime overhead, code segmentation
- Should be orthogonal:
  - Caller should decide, not callee
  - Many libraries offer asynchronous and synchronous version of the same functionality

Memory model

Causes of problems

- Weak consistency: Memory in different order by different threads (except when synchronized and at memory barriers)
- Optimizations by compiler, runtime and CPU: Instructions are reordered or eliminated

→ No sequential consistency.

Java guarantees

Atomicity

A single read/write is atomic (primitives up to 32 bit, object references). Long and double are only atomic with the *volatile* keyword!

Visibility

Atomicity does not imply visibility! One thread may not see updates of another thread at all (or possibly much later).

```
// thread 1:
while(doRun) { // this possibly never stops.
}
// thread 2:
doRun = false;
```

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

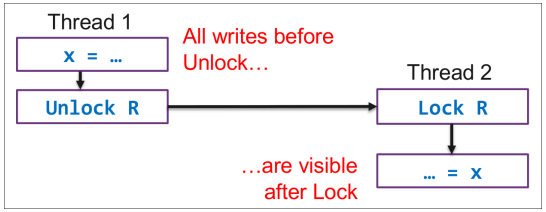

Figure 4: Visibility lock → unlock

```
private volatile boolean doRun = true;
// thread 1:
while(doRun) { // this possibly never stops.
}
// thread 2:
doRun = false;
```

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);
  }
}
```

Compare and set

`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)

```
AtomicReference<Node<T>> tope = new
AtomicReference<>();
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:
  - Preceding memory accesses are not moved below
  - Subsequent memory accesses can be moved above
- Volatile Read: Acquire semantics:
  - Subsequent memory accesses are not moved above
  - Preceding memory accesses can be moved below

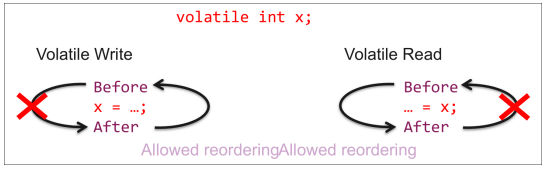

Figure 5: volatile semantics in .NET

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
- Single Program: All tasks execute their copy of the same program simultaneously.
- Multiple Data: All tasks may use different data
- 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

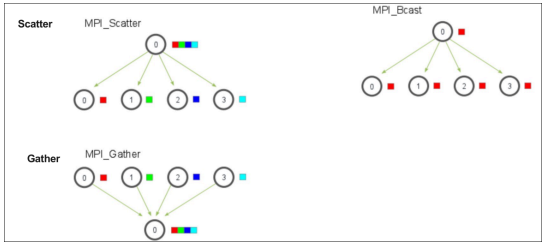

Figure 6: Scatter/Gather vs Broadcast

MPI example

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char** argv) {
  MPI_Init(&argc, &argv); // Passes arguments
to other processes via MPI middleware
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  char name[MPI_MAX_PROCESSOR_NAME];
  int len;
  MPI_Get_processor_name(name, &len); // Gets
the name of the processor (not process), e.g.
Node12
  printf("MPI Process %i, name: %s", rank,
name);
  MPI_Finalize();
}
```

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)
```

Barrier

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

Reduce

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

Allreduce

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

Allreduce distributes the aggregated value after reducing to all nodes. Therefore, one less argument (the node which collects the value).

Gather

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

Programming Models

	Single Instruction	**Multiple Instruction**
Single Data	SISD, Uniprocessor (single core)	MISD, FPGA, Google TPU
Multiple Data	SIMD, Vector Computing (GPU), Vector extension (MMX, SSE2, etc.)	MIMD, multi core, multi processors

Why SIMD?

- Many algorithms for media (e.g. black & white conversion, FFT, etc.)
- Careful with flow-control (if, switch, etc.)

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];
  }
}
```

The JVW often performs Auto-Vectorisation!

OpenMP

Why?

- No shared memory between nodes
- But: Shared memory for cores inside a node

OpenMP starts as a single initial/master thread. A pragma is used to spawn multiple threads (fork):

```
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.

Private vs firstprivate

Using Firstprivate(x) x is initialized for each thread with the value of x before the parallel part. Using private(x), x is initialized in the threads with 0 (default value).

Loop variables are private by default.

Race conditions with shared variables

Of course, race conditions can still happen:

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

Here, it's not guaranteed that the sum is added correctly, data races likely happen.

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;
}
```

However, this is:

- Extremely slow due to serialization
- Slower than a single thread
- For this code overkill
- Mutex is heavy weight too → large performance overhead

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:

- read
- write
- update (e.g. `x++` or `x = x + y`)
- capture (e.g. `v = x++`)

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

Writing fast parallel programs is hard

- Finding parallelism
- How big should the parallel part be (granularity)
- Moving data costs a lot
- Load balancing
- Coordination + Synchronization
- Performance debugging

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

Scalability Testing

- Scalability Testing = measuring the ability of an application to perform well/better with varying problem sizes and numbers of processors
- Impractical to test using the full problem size and number of processors right from the start
  - Problem size and number of processors is scaled down at first
  - The required resources for the full run is estimated
- Strong scaling vs weak scaling

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)

Amdahls law

The speedup is limited by the fraction of the serial part of the software that is not amenable to parallelization.

- 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 over-**

head for most algorithms increase in proportion to the processors used.

Mathematical definiton of Amdahls law

- $T =$ total time
- $p =$ part that can be parallelized
- $T = pT + (1 - p)T$
- Using N processors:
  - $T_N = \frac{pT}{N} + (1 - p)T$
- Serialized part $s = 1 - p$
- Speedup $= \frac{T}{\frac{pT}{N} + (1-p)T} = \frac{1}{s + \frac{p}{n}}$
- Efficiency $= \frac{T}{NT_N}$

speedup

N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02
100000	1.99	9.99	99.90

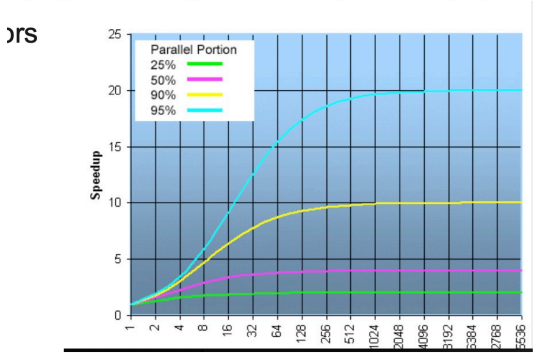

Figure 7: Speedup relative to the parallel portion of a program

Amdahls law ignores the **parallel overhead**:

- Task startup time
- Interprocess interactions:
  - Communication/data movement
  - Synchronization
- Idling due to load imbalance/synchronization
- Excess redundant computation
- Software overhead (language, libraries, OS, etc.)
- Task termination time

Gustafsons Law

In practice, the sizes of problems scale/change with the amount of available resources. A reasonable choice is to use small amounts of resources for small problems and large amount of resources for big problems.

- Weak scaling mostly used for large memory bound applications

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

GPU

- Graphics Processing Unit
- Co-processor of the system
- Specialized electronic circuit designed to manipulate and alter memory to accelerate the creation of images in a frame buffer
- Efficient at manipulating computer graphics and image processing
- Parallel structure makes them more efficient than general purpose CPUs for algorithms that process large blocks of data in parallel.

CPUs vs GPUs

- CPUs offer few cores
  - Very fast (in general)
  - Few but fast
- GPUs offer very large number of cores
  - e.g. 3584, 5760 cores
  - Very specific slower processors
  - Many but slow

Latency vs Throughput

Pipelining

Washing machine takes 30 minutes, drawer takes 60 minutes.

The first laundry therefore takes 90 minutes, **but every subsequent laundry takes 120 minutes** (not 60)! Every 60 minutes a laundry is finished.

- Latency: How long does it take to execute a task from start to end
- 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

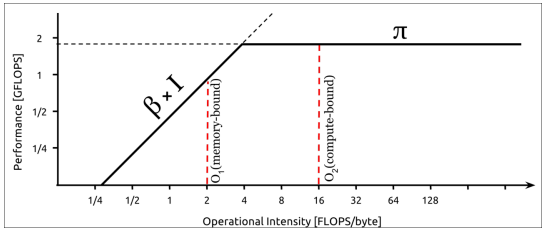

Figure 8: Roofline model

Example

- Peak floating point performance of 17.6 GFlops/s
- Peak memory bandwith of 15 GB/s

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

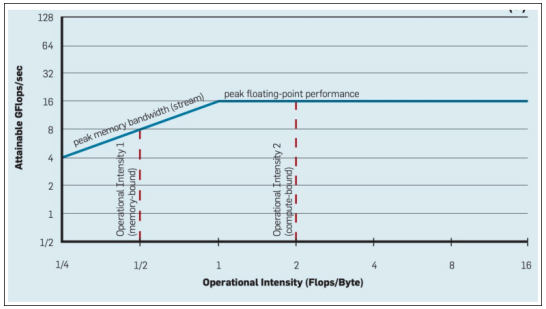

Figure 9: Roofline model example

GPUs

GPUs use more transistors. Streaming multiprocessors composed by cores. All cores perform the same instructions but on different data → SIMD.

SIMD is essentially vector parallelism.

GPU	CPU
Video Gaming	General purpose
Extremely high data parallel	Low data parallelism
Simple but many cores	Few but powerful cores
Small caches per core	Large caches in chip
Aim: high throughput	Aim: low latency per thread

NUMA Model

- NUMA: Non-Uniform Memory Access
- No shared main memory between CPU and GPU
  - Explicit transfer
- Different instruction set/architecture
  - Compile and design code for GPU

CUDA

A typical CUDA application has CPU and GPU code and is 'C' with extensions. It contains following code:

- Allocate GPU memory: `cudaMalloc`
- Copy data from CPU to GPU: `cudaMemcpy`
- Launch kernel on GPU to process data
- Copy data back to CPU: `cudaMemcpy`
- Memory is freed : `cudaFree`

```
__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);
}
```

We talk about SIMT: Single Instruction Multiple Threads.

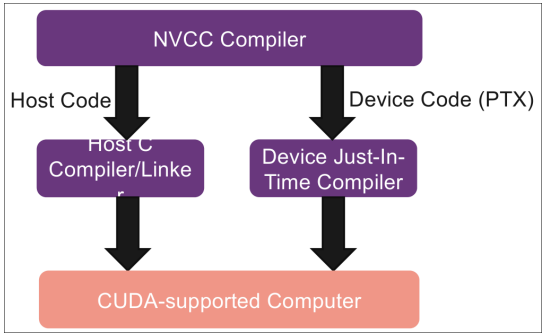

Figure 10: Cuda compilation