

# Parallel Programming | ParProg

## Zusammenfassung

### 1. MULTI-THREADING

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



**Process:** Program under Execution, own address space (heavy-weight). Proc: Process isolation and responsiveness, Cons: Interprocess communication overhead, expensive in creation, slow context switching and process termination.

**Thread:** Parallel sequence within a process. Sharing the same address space, but separate stack and registers (lightweight because most of the overhead already happened in the process creation).

**Multi-threads:** Changes made by one thread to shared resources can 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 *pre-emptive* (scheduler can asynchronously interrupt thread via time interrupt)

**JVM Thread Model:** JVM is a *process* in the OS. It runs as long as threads are running (Exception: threads marked as done with *setDaemon(true)* will not be waited upon). Threads are realized by the *class* and the *interface* *Runnable*. Code to be run in a Thread is within the overridden run()

#### 1.1. STARTING A THREAD

As a **anonymous function (Lambda)**: var myThread = new Thread() -> { /\* thread behaviour \*/ };

As a **named function**: var myThread = new Thread() -> AsyncFunction(); myThread.start();

With **explicit runnable implementation**:

```
class MyThread implements Runnable {
    @Override
    public void run() { /* thread behavior */ }
    var myThread = new 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); }
    }
}
```

The *printout* of this function varies. It can be all possible combinations of A's and B's due to the non-deterministic scheduler

**Thread Join:** Waiting for a thread to finish (t2.join() blocks as long as t2 is running).

var a = new Thread() -> multiPrint("A"); var b = new Thread() -> 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), *sleeps* (Thread.sleep(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.yield())

**Interrupts:** Threads can also be interrupted from the outside (myThread.interrupt(), Thread can decide what to do upon receiving an interrupt). If the thread is in the sleep(), wait() or join() methods, a *InterruptedException* is thrown. Otherwise a flag is set that can be checked with *isInterrupted()* / *isInterruptedOrThrow()*

**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), *setDaemon(true)*: (Mark a daemon thread), *getId()/getName()*: (Get thread ID/name), *isAlive()*: (Tests if thread is alive), *getState()*: (Get thread state)

### 2. MONITOR SYNCHRONIZATION

Threads run arbitrarily. *Restriction of concurrency* for de-terministic behavior.

**Communication between threads:** Sharing access to fields and the objects they refer to. Efficient, but poses problems: *Thread interference* and *memory consistency errors*.

**Critical Section:** Part of the code which must be executed by only 1 thread at a time for the values to stay consistent. Implementation with *mutual exclusion*.

```
class BankAccount {
    private int balance = 0;
    public void deposit(int amount) {
        // enter critical section
        synchronized(this) {
            this.balance += amount;
        } // exit critical section
    }
}
```

**parallelism:** Body of method with the *synchronized* keyword is a critical section. Guarantees *memory consistency* and a *happens-before relationship*. Important 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.balance += amount; }** Can also be used within a method, the object that *should be locked* must be specified.

**synchronized(this) { this.balance += amount; }**

**Exit synchronized block:** End of the block, return, unhandled exceptions

#### 2.1.1. Monitor Lock

A monitor is used for *small mutual exclusion*. Only one thread operates at a time in Monitor. All non-private methods are synchronized. Threads can *wait* in Monitor 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 while condition has been met.

**WakeUp signal:** Signalling a condition in thread in Monitor. notify()/ signals any waiting thread (infinite if all threads wait for the same thing, so it does not matter which one comes next - uniform waits or if only one single thread can continue like in a turnstile), notifyAll() wakes up all threads (i.e. one deposit can satisfy multiple withdrawals, does not guarantee 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 the Monitor. There is no shortcut.

InterruptedException is thrown if notify, notifyAll or wait is used outside synchronized.

```
// Java
class BankAccount {
    private int balance = 0;
    // Entry in the monitor:
    public synchronized void withdraw(int amount) {
        // while (amount > balance) { // not if
        // wait(); // wait on a condition
        // }
        balance -= amount;
    } // release / leave monitor
    public synchronized void deposit(int amount) {
        // while (amount > balance) { // not if
        // wait(); // wakes up all waiting
        // }
        balance += amount;
    }
    threads in monitor inner waiting area
}
```

### 3. SPECIFIC SYNCHRONIZATION PRIMITIVES

#### 3.1. SEMAPHORE

Allocation of a limited number of free resources. 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; }
    public synchronized void acquire() throws InterruptedException {
        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 upon fair. Slower than non-fair version)

Semaphores are *powerful*, any synchronization can be implemented. But relatively low-level.

```
class BoundedBuffer<T> {
    private Queue<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
        synchronized (queue) { queue.add(item); }
        lowerLimit.release(); // // No. of full places + 1
    } public T get() throws InterruptedException {
        T item;
        lowerLimit.acquire(); // No. of full places - 1
        synchronized (queue) { item = queue.remove(); }
        upperLimit.release(); // // No. of free places + 1
        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 (owner waiting room)

**Condition-Object:** Wait & Signal for a specific condition (owner waiting room)

**ReentrantLock:** Class in Java, *alternative to synchronized*. 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 status set on entry to this method or is interrupted while waiting (first try frees the lock in case of interrupt).

**Buffer with Lock & Condition**

```
class BoundedBuffer<T> {
    private Queue<T> queue = new LinkedList<>();
    private Lock<T> monitor = new ReentrantLock(true); // fair queue
    private Condition nonFull = monitor.newCondition();
    private Condition nonEmpty = monitor.newCondition();
    ...
    public 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
    public T get() throws InterruptedException {
        monitor.lock(); // wait for queue to be filled & signal to other queue
        try { while (Queue.size() == 0) { nonEmpty.await(); }
            T item = queue.remove(); nonFull.signal(); return item; }
        finally { monitor.unlock(); } // // always release Lock, even after Exception
    }
}
```

#### 3.3. READ-WRITE LOCK

Mutual exclusion is *unnecessary for read-only* threads. So one can *select* allow parallel reading access, but implement mutual exclusion for write access.

	Parallel	Read	Write
Read		Yes	No
Write		No	No

ReadWriteLock rwlck = new ReentrantReadWriteLock(true); // true for fairness

rwlck.readLock().lock(); // shared Lock  
// read-only accesses  
rwlck.readLock().unlock();  
rwlck.writeLock().lock(); // exclusive Lock  
// write (and read) access  
rwlck.writeLock().unlock();

#### 3.4. COUNT DOWN LATCH

Synchronization with a counter that can only *count down*. Threads can wait until counter <= 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 ready  
start.await(); // wait for all cars ready  
start.countDown(); // await race start  
start.start(); // 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 CyclicBarrier(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 i = 0; i < 2; i++) { count++; } // odd n. of exch.: last one blocks
new Thread(() -> {
    for (int i = 0; i < 2; i++) {
        try {
            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 correctness 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 still possible (i.e. non-atomic incrementing).

#### 4.2. DATA RACE

Two threads in a single process access the *same variable* concurrently without synchronization, at least one of them is a *write access*.

**Synchronize Everything?** May not help and is expensive. So no.

#### 4.3. THREAD SAFETY

**Disposable classes in synchronization:** *Immutable Classes* (Declaring all fields private and final and don't provide setters. Read-only Objects (Read-only accessers are thread safe).

**Confinement:** Object belongs to only one thread at a time. *Thread Confinement* (Object belongs to only one thread), *Object Confinement* (Object is encapsulated in other synchronized objects)

**Thread safe:** A data type or method that behaves correctly when used from multiple threads as if it was running in a single thread without any additional coordination (Java collection collections).

**Thread Safety:** Avoidance of Data Races. When no sharing is intended, give each thread a private copy of the data. When sharing is important, provide explicit synchronization.

#### 4.4. DEADLOCKS

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

```
// Thread 1
synchronized(ListA) {
    synchronized(ListB) {
        ListA.addAll(ListA);
    }
}
// Thread 2
synchronized(ListB) {
    synchronized(ListA) {
        ListA.addAll(ListB);
    }
}
```

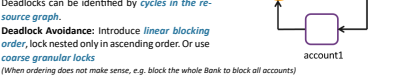
Both threads in this scenario have *locked each other* out, the program cannot continue.

**LiveLock:** Threads have blocked each other permanently, but still execute wait instructions and therefore consume CPU during deadlock.

```
// Thread 1
b = false; while(a) { ... b = true; }
// Thread 2
a = false; while(b) { ... a = true; }
```

#### 4.4.1. Resource Graph

Thread T waits for Lock T requires Lock of Resource R



Deadlocks can be identified by *cycles in the resource graph*.

**Deadlock Avoidance:** Introduce *linear blocking order*, lock nested only in ascending order. Or use *coarse granular locks* (When ordering does not make sense, e.g. block the whole Bank to block all accounts)

#### 4.5. STARVATION

A thread may not get chance to access a resource. *Avoidance:* Use fair synchronization constructs. (Aging, Release policies in previous synchronization constructs. Monitor and Thread priorities have a fairness problem.)

#### 4.6. PARALLELISM 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.putSuck(sync). Uses fair FIFO-Queue. Lacks: No fairness flag, no Lock & Condition. **Additional:** ReadWriteLocks for Upgradable Read/Write, Semaphores 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*, because 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:** Tasks 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 (Pending IO 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 memory), **Thread recycling** (avoid 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 (nested sub-tasks), results in deadlocks (if one task T<sub>1</sub> is waiting for something the T<sub>2</sub> behind him in the Queue should wait, but T<sub>2</sub> waits for T<sub>1</sub> to finish, a deadlock occurs)

#### 5.1. JAVA THREAD POOL

```
Task launch
var threadPool = new ForkJoinPool(poolSize);
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() (submit task into pool), .get() (waits if necessary for computation to complete and then returns its result) and .cancel() (Attempts to cancel execution of this task, removes it from queue). Task that has an unhandled exception occurs. It is included in the ExecutionException thrown by get().

#### 5.1.2. Fire and Forget

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

#### 5.1.3. Count Prime Numbers

```
// Sequential
int counter = 0; for (int n = 2; n <= N; n++) { if (isPrime(n)) { counter++; } }
// Parallel and Recursive
class CountTask extends RecursiveTask<Integer> { // RecursiveAction: no return value
    private final int lower, upper;
    public CountTask(int l, int u) { this.lower = l; this.upper = u; }
    protected Integer compute() {
        if (lower == upper) { return 0; }
        if (lower + 1 == upper) { return 1; }
        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 CountTask(2, N).invoke(); // invokeAll() to start multiple tasks

#### 5.1.4. Pairwise sum (recursive)

```
class PairwiseSum extends RecursiveAction {
    private final int[] array;
    private final int lower, upper;
    private static final int THRESHOLD = 1; // configurable
    public PairwiseSum(int[] array, int lower, int upper) {
        this.array = array; this.lower = lower; this.upper = upper;
    }
    protected void compute() {
        if (upper - lower > THRESHOLD) {
            int middle = (lower + upper) / 2;
            invokeAll(
                new PairwiseSum(array, lower, middle),
                new PairwiseSum(array, middle, upper));
        } else {
            for (int i = lower; i < upper; 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 *global 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 thread's 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 Processors).

#### 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 ThreadPoo, supports algorithms to provide load balancing, tasks are lightweight), has *multiple abstraction layers* (Task Parallelization: use tasks explicitly, Data Parallelization: use parallel statements and queries using task implicitly), Asynchronous Programming and PLINQ.

```
// Task with return value in C#
Task<int> task = Task.Run(() -> {
    int total = ...; // some calculation
    return total;
});
Console.WriteLine(task.Result); //Blocks until task is done and returns the result
// Waited Task
var task = Task.Run(() => {
    var left = Task.Run(() -> Count(leftPart));
    var right = Task.Run(() -> Count(rightPart));
    return left.Result + right.Result;
});
static Task<int> Count(..., int) { ... }
```

#### 5.3.1. Parallel Statements in C#

Execute *independent statements potentially in parallel* (Once a task for each item, implicit barrier at the end).

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

Execute *loop-bodies potentially in parallel* (Once a task for each item, implicit barrier at the end).

```
Parallel.ForEach(task,
    file => Convert(file));
Parallel.For(0, array.Length,
    i => DoCompute(array[i]));
```

**Parallel Loop Partitioning:** Loop with lots of quickly executing bodies. It would be *inefficient* to give each iteration a parallel thread. 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-size portions, where n = Small number, sometimes 17 and Hash: partitioning (default: if input is divisible then range partitioning, else chunk partitioning)).

**PLINQ:** 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 reusability 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 ISBN // Random order

firstNumber in InputList.AsParallel().AsOrdered() select InputPrime(number)

// Maintains order but is slower

#### 5.3.2. PLINQ

**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 reusability 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 ISBN // Random order

firstNumber in InputList.AsParallel().AsOrdered() select InputPrime(number)

// Maintains order but is slower

#### 5.3.3. Thread Injection

TPL adds new worker threads *at runtime* every time a work item completes or every 500ms.

**Hit Climbing Algorithm:** Maximize throughput while using as few threads as possible. Measures throughput by scaling number of worker threads. Avoids deadlock with task-dependencies (but forcefully ends non-desired threads. Deadlocks with ThreadPoo, SetOf(Thread) are still possible). We should keep *parallel tasks short* to better profit from this automatic performance improvement.

### 6. ASYNCHRONOUS PROGRAMMING

**Unnecessary Synchrony:** Blocking method calls are often used without need (long running calculations, I/O calls, database 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 Asynchronism:** *Call-centric* ("pass" caller waits for the task end and gets the result, blocking call), *Caller-Centric* ("pass" task hands over the result directly to successor / followup task).

**Task Continuations:** Define task which is linked to the end of the predecessor task.

```
// C# .NET
var task = Task.Run(() => {
    // Java (there can be multiple Apply/AcceptAsync calls)
    CompletableFuture
        .Run(task1)
        .ContinueWith(task2)
        .ContinueWithAsync(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 is finished (other threads get lost after first thread is done): Task.WhenAny(task1, task2).ContinueWith(continuation);

(Exceptions in fire & forget task get ignored, i.e. Task.Run() { ...; throw e; })

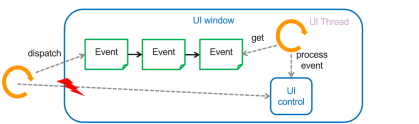
**Exception Handling:** Synchronously Wait() for the whole task-chain at the end. Register for un-observed exceptions with TaskScheduler.UnobservedTaskException (Receives unhandled exceptions from fire & forget tasks). This should be executed as soon as the task object is dead (Garbage Collection).

**Java CompletableFuture:** Modern asynchronous programming in Java. Also has *Multi-Continuation* with CompletableFuture.allOf(future1, future2) and CompletableFuture.anyOf(future1, future2).

#### 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-threaded* (Only a special UI-thread is allowed to access UI-components). The UI thread loops to process the event queue.



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

