Parallel Programming
PVW Script

Last Updated: June, 2019

Author: Lasse Meinen

pprog-pvw-skript@vis.ethz.ch

Disclaimer:
This script only serves as additional material for practice purposes and should not serve as a substitute for the lecture material. We neither guarantee that this script covers all relevant topics for the exam, not that it correct. If an attentive reader finds any mistakes or has any suggestions on how to improve the scrips, they are encouraged to contact the authors under the indicated email address or, preferably, through a gitlab issue on https://gitlab.ethz.ch/vis/luk/pvw_script_pprog.

Table of Contents

1 Introduction

1.1 How this script works

As mentioned in the disclaimer, this script is not meant to serve as a replacement for the lecture materials presented during the semester. However, this script does try to summarize the material with complementary exercises and examples. The exercises are inspired by former exam questions, but also include some other questions which might occur. The examples are simply meant to re-enforce the reader's understanding of the topics.

This script was heavily inspired by "The Art of Multiprocessor Progamming" by Maurice Herlihy and Nir Shavit and by "A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency" by Dan Grossman. Those who are interested in the subject and would like to read more or who feel that this script and the lecture materials didn't manage to cover the subject matter well enough will find these books to be good places to start.

1.2 Parallelism vs. Concurrency

The lecture and also this script are organized around a fundamental distinction between *concurrency* and *parallelism*. The definitions of these terms are not universal and many people use them differently. Nonetheless, these definitions are the most generally accepted and making the distinction is important to facilitate discussions on the subject.

Parallel programming is about using additional computational resources to solve a problem faster.

---

example 1.1. Consider the trivial problem of summing up all numbers in an array. As far as we know, there is no sequential algorithm that can do better than $\Theta(n)$ time. Suppose instead we had 4 processors. We could then produce the result (hopefully) 4 times faster by partitioning the array into 4 segments, having each processor sum one segment, and combining the results with an extra 3 additions.

Concurrent Progamming is about correctly and efficiently controlling access by multiple threads to shared resources.

---

example 1.2. Suppose we had several cooks (processes) working in a kitchen and they had some form of shared resource, e.g. an oven. Of course, it's important that a cook doesn't put a zucchini herb casserole in the oven unless the oven is empty. If the oven is not empty, we could keep checking until it is empty. In Java, you might write somehing like:

```
while(true){
    if(ovenIsEmpty()) {
        putCasseroleInOven();
        break;
    }
}
```

Unfortunately, code like this won't work when we have several threads run it at the same time, i.e. when we have several cooks. Problems like these are the primary complication in concurrent programming. Both cooks might observe the oven to be empty and then both put a casserole in, ruining both dishes in the process. We therefore need to think of ways to check if the oven is empty and put the casserole in without any other thread interfering in the mean time.

It's all-too-common for a conversation to become muddled because one person is thinking about parallelism, while the other is thinking about concurrency.

In practice, the distinction between parallelism and concurrency is not absolute. Many programs have aspects of each. Suppose you had a huge array of values you wanted to insert into a hash table. From the perspective of dividing up the insertions among multiple threads, this is about parallelism. From the perspective of coordinating access to the hash table, this is about concurrency. Also, parallelism does typically need some

coordination: even when adding up integers in an array we need to know when the different threads are done with their chunk of the work.

It's generally believed that parallelism is an easier concept to start with and we've ordered the chapters accordingly.

1.3 Basic Threads

Before writing any parallel or concurrent programs, we need some way of making multiple things happen at once and some way for those different things to communicate.

The programming model we will assume is explicit threads with shared memory. A thread is like a running sequential program, but one thread can create other threads that are part of the same program and those threads can create more threads, etc. Two or more threads can communicate by writing and reading fields of the same object. They can see the same objects because we assume memory to be shared among them.

Conceptually, all the threads that have been started but not yet terminated are "running at once" in a program (we can't tell the difference). In reality, they may be running at any particular moment, as there may be more threads than processors or a thread may be waiting for something to happen before it continues. When there are more threads than processors, it's up to the Java implementation, with help from the underlying operating system, to find a way to let the threads "take turns" using the available processors. This is called *scheduling* and is a major topic in operating systems. All we need to care about is that it's not under our control: We create the threads and the system schedules them.

We will now discuss some Java specifics for exactly how to create a new thread in Java. The details vary in different languages. In addition to creating threads, we will need other language constructs for coordinating them. For example, for one thread to read the result of another thread's computation, the reader often needs to know the writer is done.

To create a new thread in Java requires that you define a new class and then perform two actions at run-time:

1. Define a subclass of `java.lang.Thread` and override the `public` method `run`, which takes no arguments and has return type `void`. The `run` method will act like "main" for threads created using this class. It must take no arguments, but the example below shows how to work around this inconvenience.
2. Create an instance of the class created in step 1. Note that this doesn't create a running thread. It just creates an object.
3. Call the `start` method of the object you created in step 2. This step does the "magic" creation of a new thread. That new thread will execute the `run` method of the object. Notice that you do not call `run`; that would just be an ordinary method call. You call `start`, which makes a new thread that runs `run`. The new thread terminates when its `run` method completes.

---

example 1.3. Here is a useless Java program that starts with one thread and then creates 20 more threads:

```
public class Useless extends Thread{
    int i;
    Useless(int i){ this.i = i; }
    public void run(){
        System.out.printf("Thread %d says hi\n",i);
        System.out.printf("Thread %d says bye\n",i);
    }
}

public class M {
    Run | Debug
    public static void main(String[] args){
        for(int i = 0; i < 20; i++){
            Thread t = new Useless(i+1);
            t.start();
        }
    }
}
```

When running this program, it will print 40 lines of output, the order of which we cannot predict. In fact, if you run the program multiple times, you will probably see the output appear in different orders every run. This showcases that there is no guarantee that threads created earlier will run earlier. Therefore, multithreaded programs are *nondeterministic*. This is an important reason why multithreaded programs are much harder to test and debug.

We can also see how this program worked around the rule that `run` isn't allowed to take any arguments. Any "arguments" for the new thread are passed via the constructor, which then stores them in fields so that `run` can later access them.

We mentioned previously that we'd like to make a thread wait before reading a value until another thread has finished its computations, i.e. its `run` method. We can do this with the `join` keyword, which we'll introduce through another somewhat useless example.

---

example 1.4. Here is a Java program that starts with one thread which spawns 20 more threads and waits for all of them to finish.

```
public class Useless extends Thread{
    int i;
    Useless(int i){ this.i = i; }
    public void run(){
        System.out.printf("The double value of %d is %d\n",i,2*i);
    }
}

public class M {
    Run | Debug
    public static void main(String[] args){
        Thread[] threads = new Thread[20];
        for(int i = 0; i < 20; i++){
            Thread t = new Useless(i+1);
            t.start();
            threads[i] = t;
        }
        for(int i= 0; i < 20; i++){
            try{
                threads[i].join();
            } catch(InterruptedException e){}
        }
        System.out.println("All done.");
    }
}
```

The `join` method can throw an `InterruptedException`, which means we need to wrap it in a try-catch block.

1.3.1 Thread states

If we want to be able to talk about the effects of different thread operations, we need some notion of *thread states*. In short, a Java thread typically goes through the following states:

- *Non-Existing*: Before the thread is created, this is where it is. We don't know too much about this place, as it's not actually on this plane of reality, but it's somewhere out there.
- *New*: Once the `Thread` object is created, the thread enters the new state.
- *Runnable*: Once we call `start()` on the new thread object, it becomes eligible for execution and the system can start scheduling the thread as it wishes.
- *Blocked*: When the thread attempts to acquire a lock, it goes into a blocked state until it's actually obtained the lock, upon which it returns to a runnable state. In addition, calling the `join()` method will also transfer at thread into a blocked state.
- *Waiting*: The thread can call `wait()` to go into a waiting state. It'll return to a runnable state once another thread calls `notify()` or `notifyAll()` and the thread is removed from the waiting queue.
- *Terminated*: At any point during execution we can use the `interrupt()` to signal the thread that it should stop execution. It will then transfer to a terminated state. Note that when the thread is

in a runnable state, it needs to check whether its interrupted flag is set itself, it won't transfer to the terminated state automatically. Of course, exiting the `run` method is equivalent to entering a terminated state. Once the garbage collector realizes that the thread has been terminated and is no longer reachable, it will garbage collect the thread and it will return to a non-existing state, completing the cycle.

1.4 Bad Interleavings and Data Races

A *race condition* is a mistake in your program such that whether the program behaves correctly or not depends on the order that the threads execute. Race conditions are very common bugs in concurrent programming that, by definition, do not exist in sequential programming. We distinguish two types of race conditions.

One kind of race condition is a *bad interleaving*. The key point is that "what is a bad interleaving" depends entirely on what you are trying to do. Whether or not it is okay to interleave two bank-account withdraw operations depends on some specification of how a bank is supposed to behave.

---

example 1.5. Suppose we have the following implementation of a `peek` operation on a concurrent stack.

```
static <T> T peek(Stack<T> s){
    T ans = s.pop();
    s.push(ans);
    return ans;
}
```

Assume that the `pop` and `push` methods are implemented correctly. While `peek` might look like it's implemented correctly, the following interleaving might occur:

```
T ans = pop();

push(ans);

return ans;
```

Thread 1 (`peek`)

```
push(x);

push(y);

T e = pop();
```

Thread 2

A *data race* is a specific kind of race condition that is better described as a "simultaneous access error", although nobody uses that term. There are two kinds of data races:

- When one thread might read an object field at the same moment that another thread writes the same field.
- When one thread might write an object field at the same moment that another thread also writes the same field.

Notice it is not an error for two threads to both read the same object field at the same time.

Our programs must never have data races even if it looks like a data race would not cause an error - if our program has data races, the execution of your program is allowed to do very strange things.

---

example 1.6. Let's consider a very simple example.

```
class C {
    private int x = 0;
    private int y = 0;

    void f() {
        x = 1; // line A
        y = 1; // line B
    }
    void g() {
        int a = y; // line C
        int b = x; // line D
        assert(b >= a);
    }
}
```

Notice that `f` and `g` are not synchronized, leading to potential data races on fields `x` and `y`, Therefore, it turns out that the assertion in `g` can fail. But there is no interleaving of operations that justifies the assertion failure, as can be seen through a proof by contradiction:

Assume the assertion fails, meaning `!(b>=a)`. Then `a==1` and `b==0`. Since `a==1`, line B happened before line C. Since A must happen before B, C must happen before D, and "happens before" is a transitive relation, A must happen before D. But then `b==1` and the assertion holds.

There is nothing wrong with the proof except its assumption that we can reason in terms of "all possible interleaving" or that everything happens in certain orders. We can reason this way only if the program has no data races.

1.5 Other Models

We've introduced a programming model of explicit threads with shared memory. This is, of course, not the only programming model for concurrent or parallel programming. Shared memory is often considered convenient because communication uses "regular" reads and writes of fields to objects . However, it's also considered error-prone because communication is implicit; it requires deep understanding of the code/documentation to know which memory accesses are doing inter-thread communication and which are not. The definition of shared-memory programs is also much more subtle than many programmers think because of issues regarding data races, as discussed in the previous section.

Three well-known, popular alternatives to shared memory are presented in the following. Note that different models are better suited for different problems. Models can be abstracted and built of top each other as we wish or we can use multiple models in the same program (e.g. MPI with Java).

Message-passing is the natural alternative to shared memory. In this model, we have explicit threads, but they do not share objects. To communicate, there is a seperarte notion of a message, which sends a copy of some data to its recipient. Since each thread has its own objects, we do not have to worry about other threads wrongly updating fields. But we do have to keep track of different copies of things being produced by messages. When processes are far apart, message passing is likely a more natural fit, just like when you send email and a copy of the message is sent to the recipient.

Dataflow provides more structure than having "a bunch of threads that communicate with each other however they want." Instead, the progammer uses primitives to create a directed acyclic graph. A node in the graph performs some computation using inputs that arrive on its incoming edges. This data is provided by other nodes along their outgoing edges. A node starts computing when all of its inputs are available, something the implementation keeps track of automatically.

Data parallelism does not have explicit threads or nodes running different parts of the program at different times. Instead, it has primitives for parallelism that involve applying the same operation to different peices of data at the same time. For example, you would have a primitive for applying some function to every element of an array. The implementation of this primitive would use parallelism rather than a sequential for-loop. Hence all the parallelism is done for you provided you can express your program using the available primitives.

2 Parallelism

This chapter covers the basics of parallelism and how to examine the performance of a parallel program. Recall that when we talk about parallelism, we mean the use of additional computational resources to solve a problem faster. The exercises in this chapter will usually ask you to implement a parallel algorithm to solve a certain problem, to recognize the issue of a given implementation and to analyze the performance of an implementation.

2.1 Performance

2.1.1 Speedup

Before we begin to have a look at different ways of measuring performance, we first need to introduce some terminology. To that end, let P denote the number of processors available during the execution of a program.

Definition 2.1.1. $\mathbf{T_P}$ is the time it takes the program to execute on P processors.

Remark. T_∞ denotes the execution time when we have as many processors at our disposal as is required to get the best-possible execution time.

We will usually use T_1 to talk about the *sequential* and T_∞ to talk about the *minimum* execution time of a program. When talking about performance in general, we will use T_P.

Definition 2.1.2. The **speedup** of a program is

$$S_P := \frac{T_1}{T_P}$$

The speedup is essentially the ratio of the sequential execution time to the execution time given P processors. Naively, we might expect the speedup of our program to always be $S_P = P$, i.e. $T_P = T_1/P$. However, reality is often disappointing. In reality, additional overheads caused by inter-thread dependencies, creating threads and communicating between them and memory-hierarchy issues can greatly limit the speedup we gain from adding more processors.

2.1.2 Amdahl's Law

We noted in the previous section that the speedup of a given parallel program can be greatly decreased due to overhead caused by several issues introduced by parallelizing the program. In fact, any sequential parts of a program can drastically impact the maximum achievable speedup. We will derive Amdahl's law here to see why this is the case.

Let W_{ser} denote the time spent doing non-parallelizable, serial work and W_{par} denote the time spent doing parallelizable work.
We then write:

$$T_1 = W_{ser} + W_{par}$$

It's easy to see that T_{ser} remains constant as we increase the number of processors. Therefore, given P processors, we obtain the following lower bound on T_P:

$$T_P \geq W_{ser} + \frac{W_{par}}{P}$$

Recall the definition of speedup. Plugging in the relations derived above, we get:

$$S_P = \frac{T_1}{T_P} \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{P}}$$

Let $\mathbf{f}$ denote the non-parallelizable, serial fraction of the total work. We then obtain the following equalities:

$$W_{ser} = \mathbf{f} * T_1$$
$$W_{par} = (1 - \mathbf{f}) * T_1$$

This gives us the more common form of Amdahl's Law:

Theorem 2.1. *Let* $\mathbf{f}$ *denote the non-parallelizable, serial fraction of the total work done in a program and* P *the number of processors at our disposal. Then, the following inequality holds:*

$$S_P \leq \frac{1}{\mathbf{f} + \frac{1-\mathbf{f}}{P}}$$

When we let P go to ∞, we see:

$$S_\infty \leq \frac{1}{\mathbf{f}}$$

In order to see why this is such an important result, we can try plugging in a couple of values. Assume that 25% of a program is non-parallelizable. This means that even with the *IBM Blue Gene/P* supercomputer with its 164'000 cores, we can only achieve a speedup of at most 4. While a depressing result at first sight, this makes perfect sense when we consider the fact that these 25% are completely fixed, in the sense that the execution time can't possibly be reduced past this point.

The conclusion we can draw from this result is that it's worth investing some more time into reducing the sequential fraction of our program, e.g. by reducing the overhead of communicating between threads or by reducing the granularity of locks in our code as much as possible.

2.1.3 Gustafson's Law

Amdahl's law considered a fixed workload and provides us with an upper bound on the speedup achievable when increasing the number of processors at our disposal. As the obtained result is rather depressing, we seek to find something slightly more optimistic. This leads us to an approach where we increase the problem size as we improve the resources at our disposal. In other words, we consider the time interval to be fixed and look at the problem size.
Let W denote the work done in a fixed time interval. We get

$$W = \mathbf{f} * W + (1 - \mathbf{f}) * W$$

As we increase the number of processors at our disposal, we can only speed up the parallel fraction of our program. The serial fraction remains the same. Letting W_P be the work done with P processors at our disposal, we get

$$W_P = \mathbf{f} * W + P * (1 - \mathbf{f}) * W$$

We can now follow *Gustafson's law.*

Theorem 2.2. *Let* $\mathbf{f}$ *denote the non-parallelizable, serial fraction of the total work done in the program and* P *the number of processors at our disposal. Then, we get*

$$\begin{aligned} S_P &= \mathbf{f} + P(1 - f) \\ &= P - \mathbf{f}(P - 1) \end{aligned}$$

If we again consider a program where 25% is non-parallelizable, we get a speedup of 4 when we increase the number of processors to 5.

2.2 Divide & Conquer

In divide & conquer we split a problem into smaller subproblems, solve the task recursively for each of these and combine the results such that we obtain the final result for the entire problem. To see how this approach works, let's construct a small example:

---

example 2.1. Through some weird twist of faith, we're given a list of the number of clocks each Swiss citizen posseses and are tasked with finding the maximum among all of these. For the sake of simplicity, suppose this list only has eight entries.

15 7 9 8 4 22 42 13

The first step of a divide & conquer algorithm is to divide the problem into smaller subproblems.

15 7 9 8 4 22 42 13

We then continue on like this until we reach a base case. Note that we break the list into single numbers and not into pairs of numbers. The reason for this is that the list might have an odd length, which of course couldn't be broken into pairs of numbers.

15 7 9 8 4 22 42 13

Then we begin to find the maximum, 2 numbers at a time.

15 9 22 42

We continue on like this until only one number remains.

42

We've thus come to the conclusion that the largest number of clocks possessed by a single person in Switzerland is 42.

This algorithm consisted of three steps. First, we broke the list into two smaller lists. This is known as the "divide" step. Then, we solved the task for each of these sublists. This is known as the "conquer" step. Finally, we combined the results of the subproblems by taking the maximum of the two returned maxima.

While divide & conquer algorithms seem very natural to parallelize, we currently have no easy way of determining the efficiency of these algorithms in a parallel setting and it's quite tedious to implement using regular Java threads. We will therefore first introduce a way of visualizing the performance of a divide & conquer algorithm and will then look at two different libraries to see how we can easily implement this.

2.2.1 Task Graphs

We will describe a program execution as a directed acyclic graph where:

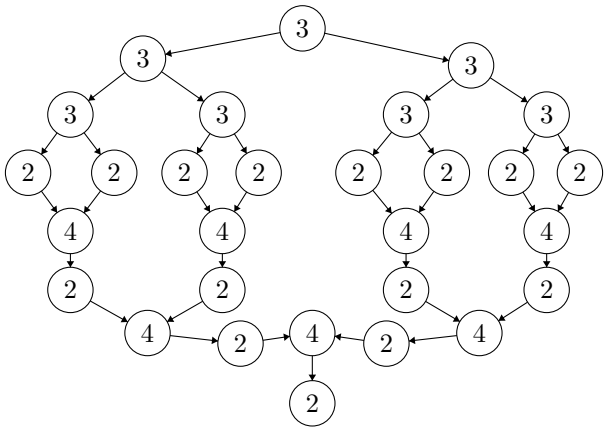
- Nodes are pieces of work the program performs. Each node will have a constant execution time per task. In most cases we will interpret the execution time per node as 1.
- Edges represent that the source node must complete before the target node begins, i.e. there is a data dependency along the edge.

We will now connect this visual representation of the execution of a program with the terminology introduced in the previous section by introducing two more terms:

- **Work** is the sum of the time cost of all nodes in the graph, i.e. T_1.
- **Span** is the sum of the time cost of all nodes along the critical, i.e. longest, path in the graph. The span determines the best possible execution time we can achieve by introducing more processors, i.e. T_∞.

example 2.2. Let's return to our example from the previous section where we try to find the largest number of clocks currently owned by a Swiss citizen. Assume that splitting a list into two sublists can be done in 3 "time units", finding the maximum can be done in 2 units and appending two lists can be done in 4 units.

The task graph will then look like this:



Let's say we now wish to compute the maximum achievable speedup we can gain by parallelizing this algorithm, i.e. S_∞. We know that the formula for speedup is

$$S_P = \frac{T_1}{T_P}$$

T_1 is defined as the sum of the time cost all nodes in the graph, which in this case would be 79. We can compute T_∞ as the sum of the time cost of all nodes along the critical path, which we see is 29. As a final result, we thus obtain

$$S_\infty = \frac{79}{29} \approx 2.72$$

We recognize from the width of the graph that we can issue at most 8 operations in parallel. This means that we achieve T_∞ with 8 processors, i.e. $S_8 = S_\infty$

2.2.2 Executor Service

Instead of managing the threads ourselves we can use a library which manages a threadpool to which we can submit tasks. A task is either:

- A `Runnable` object, which implements a method `void run()` and doesn't return a result
- or a `Callable<T>` object, which implements a method `T call()` and returns a result of type `T`.

Upon submitting a task a `Future<T>` is created, which contains the result of the submitted task. Implementing a divide & conquer algorithm now looks a lot more straightforward.

example 2.3. We wish to submit tasks to the `ExecutorService` such that we can obtain some sort of result from the task's execution. We therefore need to implement a `Callable` task (i). We've additionally provided the code for creating an `ExecutorService` and submitting the topmost task (ii). Try running it locally and see what happens.

```java
class MaxTask implements Callable {

    int l, h;
    int[] arr;
    ExecutorService ex;

    public MaxTask(ExecutorService ex, int lo, int hi, int[] arr){…

    public Integer call() throws Exception {
        // Check base case
        int size = h-l;
        if(size == 1)
            return arr[l];
        // Split work
        int mid = size / 2;
        MaxTask m1 = new MaxTask(ex, l, l + mid, arr);
        MaxTask m2 = new MaxTask(ex, l + mid, h, arr);
        // Start subtasks
        Future<Integer> f1 = ex.submit(m1);
        Future<Integer> f2 = ex.submit(m2);
        // Combine results
        try{
            return Math.max(f1.get(), f2.get());
        }catch(Exception e){
            return 0;
        }
    }
}
```

(i) implementation of `Callable` task

```java
public static void main(String[] args){
    int[] arr = new int[]{15,7,9,8,4,22,42,13};
    ExecutorService ex = Executors.newFixedThreadPool(4);
    MaxTask top = new MaxTask(ex, 0, arr.length, arr);
    Future<Integer> max = ex.submit(top);
    try{
        System.out.println(max.get());
    } catch(Exception e){}

    ex.shutdown();
}
```

(ii) implementation of `main` method

You should see that the program never terminates. The reason for this is that the `ExecutorService` limits the number of threads that can be spawned. Once all threads are occupied by tasks waiting for the result of spawned subtasks, execution is therefore halted. No threads are available to run the subtasks on, i.e. the subtasks will wait indefinitely.

With the `ExecutorService` the limited thread pool size caused the program to run forever, as all currently active threads were occupied by tasks waiting for the spawned subtasks to return. While there are possible approaches that could alleviate this problem, e.g. seperating the work partitioning from solving the sub-tasks, they are all non-trivial to implement and are therefore not well suited for use in practice.

2.2.3 Fork/Join Framework

We continue our search for a library well-suited to implementing divide & conquer algorithms. The `java.util.concurrent` package includes classes designed exactly for this type of fork-join parallelism.
Compared to Java threads, the usage of these classes is exactly the same, but with some different names and interfaces. First, we'll introduce the required terminology. Second, we'll have a look at the generic recipe for implementing fork-join parallelism.

We use the library as follows:

- Instead of extending `Thread`, we extend `RecursiveTask<T>` (with return value) or `RecursiveAction` (without return value)
- Instead of overriding `run`, we override `compute`
- Instead of calling `start`, we call `fork`
- Instead of a topmost call to `run`, we create a `ForkJoinPool` and call `invoke`

Also, note that in the case of `RecursiveTask<T>`, `join` now returns a result.

example 2.4. We again return to our previous example of trying to find the largest number of clocks currently owned by any Swiss citizen. We'd again like our tasks to return a result. We therefore need to implement a `RecursiveTask<T>` (i). We've additionally provided the code for creating a `ForkJoinPool` and submitting the topmost task (ii). Try running it locally and see if it works this time around.

```java
class MaxForkJoin extends RecursiveTask<Integer> {
    int l, h;
    int[] arr;

    public MaxForkJoin(int lo, int hi, int[] arr){…

    protected Integer compute(){
        // Check base case
        size = h-l;
        if(size == 1)
            return arr [l];
        // Split work
        int mid = size / 2;
        MaxForkJoin m1 = new MaxForkJoin(l, l+mid, arr);
        MaxForkJoin m2 = new MaxForkJoin(l+mid, h, arr);
        // Run subtasks
        m1.fork();
        int max2 = m2.compute();
        int max1 = m1.join();
        // Combine results
        return Math.max(max1, max2);
    }
}
```

(i) Implemention of `RecursiveTask`

```java
public static void main(String[] args){
    int[] arr = new int[]{15,7,9,8,4,22,42,13};
    MaxForkJoin tp = new MaxForkJoin(0, arr.length, arr);
    ForkJoinPool fjp = new ForkJoinPool();
    int res = fjp.invoke(tp);
    System.out.println(res);
}
```

(ii) Implementation of `main` method

Several things should be noted in this example. First, changing anything about the order in which the subtasks were called will result in a sequential execution (try it!). Second, similar to how we did it for regular threads, we call `compute()` instead of submitting another task using `fork()`.
Both in the case of the `ForkJoin`-framework and of `ExecutorService`, one should pay extra attention as to how to split the array such that all indices are covered exactly once. In this example and example 2.3, note the way the middle index was computed and think about what indices the `h` and `l` variables represented exactly.

2.3 Pipelining

There are three approaches to applying parallelism to improve sequential processor performance. The first is vectorization, where we essentially apply operations N-at-a-time, as opposed to the standard one-at-a-time approach. The second is instruction level parallelism, where we ask ourselves which instructions are independent of each other and can therefore be executed in parallel and/or reordered. This is where keywords such as superscalar CPUs and out-of-order execution come into play. The third and most important approach that we will consider is pipelining.

Pipelining is a technique where multiple independent instructions are overlapped in execution through the use of multiple execution units, provided they are available for use.

Definition 2.3.1. Throughput is the number of instructions that exit the pipeline per a given time unit. Throughput can be calculated as follows:

$$Throughput \approx \frac{1}{max(computationtime(stages))}$$

Note that we usually consider throughput when the pipeline is fully utilized, i.e. ignoring lead-in and lead-out time.

Definition 2.3.2. Latency is the time to perform a single computation, including wait time resulting from resource dependencies.

When designing a pipeline, it's of course always our aim to increase throughput and decrease latency as much as possible. However, the two goals are often conflicting.

Definition 2.3.3. A pipeline is **balanced** if the latency remains constant over time.

---

example 2.5. We return to the good old washing cycle pipeline, only with a little twist this time around. The stages of the washing cycle consist of first using the washing machine, then the dryer, folding the washing, and finally putting it away in the closet. The washing machine is quite a cheap brand and takes 15 seconds per washing (as opposed to the usual 5 seconds). The dryer takes 10 seconds, folding the washing 5 seconds, and putting it away in the closet another 5 seconds.

Time (s)	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90
Load 1	w	w	w	d	d	f	c												
Load 2				w	w	w	d	d	f	c									
Load 3							w	w	w	d	d	f	c						
Load 4										w	w	w	d	d	f	c			
Load 5													w	w	w	d	d	f	c

Looking at the definitions above, we find that the latency is 30 seconds and the throughput is 1 load per 15 seconds. In order to improve our throughput, we buy a new washing machine and give away the old one to charity. The new execution time for the washing stage is 5 seconds.

Time (s)	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90
Load 1	w	d	d	f	c														
Load 2		w	-	d	d	f	c												
Load 3			w	-	-	d	d	f	c										
Load 4				w	-	-	-	d	d	f	c								
Load 5					w	-	-	-	-	d	d	f	c						

We see that our new throughput is 1 load per 10 seconds. Latency, however, seems to be increasing, i.e. the pipeline is now unbalanced.

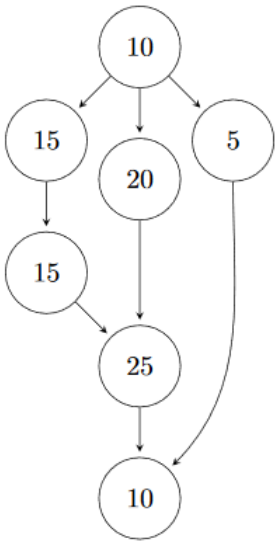
This example illustrated one important problem that can occur in pipelines. Despite achieving a high throughput, the latency increased indefinitely. Over the years it has become evident that improving sequential processor performance with methods such as these is not enough. This has led to the development of parallel architectures such as multicore processors and processors implementing simultaneous multithreading as we know them today.

2.4 Exercises

1. Amdahl's Law, Gustafson's Law, Performance.

1. Suppose a computer program has a method **M** that cannot be parallelized, and that this method accounts for 40% of the program's execution time. What is the limit for the overall speedup that can be achieved by running the program on an n-processor multiprocessor machine according to **Amdahl's Law**?
2. The analysis of a program has shown a speedup of 5 when running on 15 cores. What is the serial fraction according to **Amdahl's Law** (assuming best possible speedup)?
3. The analysis of a program has shown a speedup of 5 when running on 15 cores. What is the serial fraction according to **Gustafson's Law**?

2. Task Graph. The following figure shows the task graph for an algorithm. The number in each node denotes the execution time per task. What is the maximum overall achievable speedup that can be achieved by parallelism when the algorithm runs once compared to sequential execution? How many processors are required to achieve this speedup?



3. Fork/Join Framework. Given an array of integers, your task is to find the maximum subarray sum among all possible subarrays. For example,

$$\{2, -4, \mathbf{1, 9, -6, 7}, -3\} \rightarrow 11 \text{ (marked in } \mathbf{bold}\text{)}$$

Extend the following class such that it computes the maximum subarray sum as described above.

```
public abstract class MaxSubArraySum extends RecursiveTask<Integer> {
    private static final long serialVersionUID = 1L;
    int start;
    int length;
    int[] input;

    public MaxSubArraySum(int start, int length, int[] input){
        this.start = start;
        this.length = length;
        this.input = input;
    }

    public abstract Integer compute();
}
```

4. Pipelining. Over at UZH the law students have been tasked with writing a legal essay about the philosophy of Swiss law. In order to write the essay, a student first needs to inform themselves about the subject. To do so, they must read from four different books, each of which contains information necessary to understand the next book.

Every student takes the exact same amount of time to read a book, namely:

1) Reading book A takes 80 minutes
2) Reading book B takes 40 minutes
3) Reading book C takes 120 minutes
4) Reading book D takes 40 minutes

1. Let's assume all law students are a bit too competitive and don't return any books before they're done reading all of them. How long will it take for 4 students until all of them have started writing their essays?
2. The library introduces a "one book at a time" policy, i.e. the students have to return a book before they can start on the next one. How long will it now take for 4 students until all of them have started writing their essays? What is the throughput of this "pipeline" per hour? What is the latency?
3. Where is the problem with this system? What can the library do to solve this problem?

2.5 Solutions

1. Amdahl's Law, Gustafson's Law, Performance.

1. We see that the program has a serial fraction of 40%. Therefore, we set $\mathbf{f} = 0.4$ and insert this into the formula specified by Amdahl's Law:

$$S_P \leq \frac{1}{\mathbf{f} + \frac{1-\mathbf{f}}{P}} = \frac{1}{0.4 + \frac{0.6}{P}} \xrightarrow{P \to \infty} \frac{1}{0.4} = 2.5$$

2. Assuming best possible speedup, Amdahl's Law tells us the following holds true

$$S_P = 5 = \frac{1}{\mathbf{f} + \frac{1-\mathbf{f}}{P}} = \frac{1}{\mathbf{f} + \frac{1-\mathbf{f}}{15}}$$
$$\iff$$
$$5\mathbf{f} + \frac{1-\mathbf{f}}{3} = 5\mathbf{f} - \frac{1}{3}\mathbf{f} + \frac{1}{3} = 1$$
$$\iff$$
$$\mathbf{f} = \frac{1}{7}$$

3. Gustafson's Law tells us the following holds true

$$S_P = \mathbf{f} + P(1-\mathbf{f}) = \mathbf{f} + 15(1-\mathbf{f}) = 5$$
$$\iff$$
$$14\mathbf{f} = 10$$
$$\iff$$
$$\mathbf{f} = \frac{5}{7}$$

2. Task Graph. The formula for speedup is $S_P = \frac{T_1}{T_P}$. We know that T_1 is the simply the sum of the execution times of all nodes, i.e. $T_1 = 100$. We also know that the best-possible execution time achievable through parallelization is limited by the length of the critical path. The critical path has length 75, i.e. $T_\infty = 75$. Inserting these values into the formula for speedup gives us:

$$S_\infty = \frac{T_1}{T_\infty} = \frac{100}{75} \approx 1.33$$

We find that two processors are enough to achieve this speedup.

3. Fork/Join Framework. We stick to the generic recipe for the Fork/Join framework. The difficulty with this exercise is the merging of the results of the two subtasks, as the maximum subarray sum might contain elements of both array partitions. We therefore need to additionally compute the maximum subarray sum which crosses the middle border and return the maximum of the three sums.

```
public class MaxSubArraySumForkJoin extends MaxSubArraySum{

    private static final long serialVersionUID = 1L;

    public MaxSubArraySumForkJoin(int start, int length, int[] input, int CUTOFF) {
        super(start, length, input, CUTOFF);
    }

    public Integer compute(){
        // Check base case
        if(length <= CUTOFF){
            return MaxSumHelper.MaximumSum(input, start, start+length-1);
        }
        // Split work
        int mid = length / 2;
        MaxSubArraySumForkJoin l = new MaxSubArraySumForkJoin(start, mid, input, CUTOFF);
        MaxSubArraySumForkJoin r = new MaxSubArraySumForkJoin(start+mid, length-mid, input, CUTOFF);
        // Compute maximum subarray sums of left and right partitions
        l.fork();
        int rightMax = r.compute();
        // Find maximum subarray sum crossing middle border
        int rightMidMax = 0, sum = 0;
        for(int i = start+mid; i<start+length; i++){
            sum += input[i];
            if(sum > rightMidMax)
                rightMidMax = sum;
        }
        int leftMidMax = 0;
        sum = 0;
        for(int i = start+mid-1; i>=start; i--){
            sum += input[i];
            if(sum > leftMidMax)
                leftMidMax = sum;
        }
        // Combine results
        return Math.max(leftMidMax + rightMidMax, Math.max(l.join(), rightMax));
    }
}
```

4. Pipelining.

1. In the case of a sequential execution we simply need to add the execution times of the four stages together and multiply this by the number of students.

$$(80 + 40 + 120 + 40) * 4 = 280 * 4 = 1120 \text{ minutes}$$

2. We assume all students immediately pick up a book as soon as it's available and they've finished reading the previous one. We can then model execution in the following way:

Time (s)	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720
Load 1	A	A	B	C	C	C	D												
Load 2			A	A	B	-	C	C	C	D									
Load 3					A	A	B	-	-	C	C	C	D						
Load 4							A	A	B	-	-	-	C	C	C	D			
Load 5									A	A	B	-	-	-	-	C	C	C	D

We compute the throughput the standard way, i.e. Throughput $= \frac{1}{120}$. Therefore, throughput is 1 student per 2 hours.

When looking at the diagram above we see that the waiting time before students can read book C, and therefore also the latency, increases indefinitely.

3. Latency increasing indefinitely is a sign that the pipeline is unbalanced. There are several solutions to this problem. First, the library could force people to keep each book for exactly 120 minutes or, slightly more efficient, to keep book A for 120 minutes and book B for 80 minutes. Alternatively, the library could buy a second copy of book C. The reader is encouraged to model the execution and convince themselves that this would alleviate the issue. A last option would be to just split book C into two copies, effectively replacing the third stage with two stages, each with an execution time of 60 minutes.

Time (s)	0	40	80	120	160	200	240	280	320	360	400	440	480	520	560	600	640	680	720
Load 1	A	A	B	C	C‖C'	C'	D												
Load 2			A	A	B	C	C‖C'	C'	D										
Load 3					A	A	B	C	C‖C'	C'	D								
Load 4							A	A	B	C	C‖C'	C'	D						
Load 5									A	A	B	C	C‖C'	C'	D				

3 Concurrency

In the introductory chapter we saw how several threads accessing the same objects can lead to incorrect or undesirable executions and dubbed such cases **data races**. In this chapter, we'll first define the term mutual exclusion and the properties that can be associated with mutual exclusion. We'll then implement mutual exclusion as locks.
In order to widen our understanding of mutual exclusion implementations, we'll have a look at some of the more high-level synchronization primitives such as semaphores and barriers. Implementing what we've learned, we'll then look at different strategies of implementing concurrent data structures, moving from locked structures all the way to lock-free structures. After having seen some of the challenges of lock-free programming, we'll finally close with the introduction of some basic concurrency theory, which could help us show that our concurrent object is indeed correct or see how strong a certain synchronization primitive is.

For every memory location in your program, you must obey at least one of the following:

- **Thread local**: Do not use the location in more than one thread, e.g. by giving each thread its own copy of a resource, provided that threads don't need to communicate through this resource.
- **Immutable**: Do not write to the memory location. You can enforce this by declaring such a variable final.
- **Synchronized**: Use synchronization to control access to the location.

In practice, programmers usually over-synchronize their programs. When writing a concurrent program, you should focus on putting as much as possible in the first two categories, as synchronization is usually very expensive.

3.1 Mutual Exclusion

If a programmer isn't able to make a memory location thread local or immutable, they must ensure that no bad interleavings or data races occur by implementing **mutual exclusion**. This is done by defining critical sections, i.e. sections of code which only thread at a time is allowed to execute.

We will now first define progress-conditions and use these to explain the requirements for the correct implementation of a critical section. We will then turn our attention to finite state diagrams, which will allow us to formally determine whether all these requirements hold.

3.1.1 Progress Conditions

When talking about concurrent algorithms, we distinguish between *blocking* and *non-blocking* algorithms.

As one might think, in blocking algorithms threads might occasionally go into a blocked state, e.g. when attempting to acquire a lock. Among blocking algorithms, we distinguish two further cases:

- **Deadlock-free**: *At least one* thread is guaranteed to proceed into the critical section at some point.
- **Starvation-free**: *All* threads are guaranteed to proceed into the critical section at some point.

In non-blocking algorithms, threads never enter a blocked state, i.e. can always continue execution. This mainly suggests that the algorithms doesn't use any locks. We again distinguish two further cases:

- **Lock-free**: *At least one* thread always makes progress.
- **Wait-free**: *All* threads make progress within a finite amount of time.

When comparing the different definitions listed above, we notice a few things. We see that lock-freedom and starvation-freedom both imply deadlock-freedom. We further notice that wait-freedom implies both lock-freedom and starvation-freedom. We summarize the different conditions in the following table.

	Non-Blocking	Blocking
Someone makes progress	Deadlock-Free	Lock-Free
Everyone makes progress	Starvation-Free	Wait-Free

Of course, it's perfectly possible that an algorithm fits into none of these categories, e.g. when there is an execution that results in a deadlock.

We can now define the requirements for a correct implementation of a critical section. A critical section has to be:

- **Deadlock-free**: At least one thread is able to enter the critical section.
- **Mutually exclusive**: At most one thread is in the critical section at a time.

---

example 3.1. We've written a class `BooleanFlags` that increments a counter 1000 times:

```
public class BooleanFlags extends Thread {
    static int cnt = 0;
    static boolean flag = false;

    public void run(){
        for(int i = 0; i<1000; i++) {
            while(flag != false);
            flag = true;
            cnt++;
            flag = false;
        }
    }
}
```

When running the program with two threads the counter turns out to be unexpectedly low.

Looking at the implementation, we see that the critical section is in between the `flag = true` and `flag = false` statements. ¿The problem here is that there is a possible execution of the two threads, namely when both read `flag == false` before either of them could set `flag = true`, where both threads enter the critical section. Therefore, the second requirement of mutual exclusiveness is violated and this implementation is incorrect.

As a final note, it's important to mention **livelocks**. A livelock occurs when all or at least some threads are changing state, but none of them actually enter the critical section. One might think of this visually as two people in a narrow hallway continuously trying to get past each other by stepping aside, but always stepping in the same direction. Note that a livelock doesn't mean that there is no possible execution which results in a thread entering the critical section, but simply that there is the possibility of the threads returning to the same state without any thread entering the critical section. This also implies that the algorithm isn't deadlock-free by definition.

3.1.2 State Diagrams

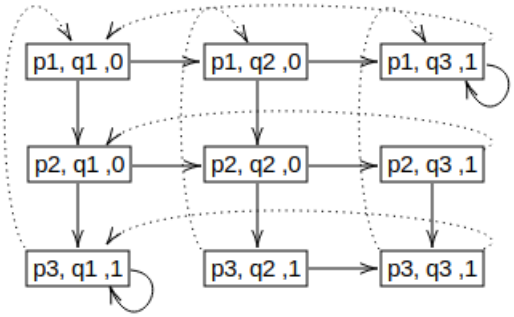
An execution state diagram visually represents the different states and state transitions between them a particular program might go through. A program state is determined by the instructions the threads are executing and the states of global variables and concurrent objects. A state transition is then simply represented by arrows between such boxes.

---

example 3.2. We continue the previous example and construct the state diagram corresponding to the given code snippet, where we restrict ourselves to two threads for simplicity. We represent a state by a tuple $(\mathrm{p}i, \mathrm{q}j, b)$ for $i, j \in \{1, 2, 3\}$ and $b \in \{0, 1\}$.
$\mathrm{p}i$ resp. $\mathrm{q}j$ represents the instruction the corresponding thread P or Q is about to execute. The represented instructions are:

- $i = 1$: `while(flag != false);`
- $i = 2$: `flag = true;`
- $i = 3$: `cnt++;`

b represents the value of `flag`.



We see that there is indeed a path of state transitions leading to a state in which both threads are in the critical section. Therefore, the code doesn't provide mutual exclusion.

As you might have noticed, we omitted/summarized non-critical states for the sake of brevity.

Using this state transition diagram, we can now easily recognize whether the critical section is implemented correctly.

- **Deadlocks** can be recognized by states which have no outgoing state transitions.
- **Livelocks** are any possible cycle of state transitions in which in none of the states a thread is in the critical sections.
- The critical section is **mutually exclusive** if there is no state in which more than one thread is in the critical section.

3.2 Mutex Implementation

We will now look at different ways of actually implementing mutual exclusion. We will first look at different algorithms, always considering their advantages and disadvantages, then their efficiency on modern multiprocessors and how to improve further. In order to understand how these implementations were constructed or what the concrete proofs of the characteristics look like, one should consult the lecture slides or Chapter 2 of the book "The Art of Multiprocessor Programming" by Herlihy and Shavit.

In the following, we assume all reads and writes are atomic. This assumption is necessary, as while we do add the `volatile` keyword to all array declarations, this only concerns the array references, not their entries. In practice, one would declare the arrays to be arrays of `AtomicInteger`s, i.e. as `AtomicIntegerArray`.

3.2.1 Peterson Lock

We assume only two threads attempt to acquire the lock, one with `ThreadID == 0` and the other with `ThreadID == 1`. The idea of the algorithm is as follows:

1. Set our flag, thereby indicating that we're interested in entering the critical section.
2. Indicate that the other thread is allowed to go first. The thread that arrives at this statement first will enter the critical section first.
3. Wait until the other thread is either no longer interested in entering the critical section or until we're allowed to go first.
4. Indicate that we're no longer interested.

```
class PetersonLock {
    private volatile boolean[] flag = new boolean[2];
    private volatile int victim;
    public void lock(int id){
        flag[id] = true;                    // 1.
        victim = id;                        // 2.
        while(flag[1-id] && victim == id) {} // 3.
    }
    public void unlock(int id){
        flag[id] = false;                   // 4.
    }
}
```

It's easy to see that the Peterson Lock satisfies the requirements for a correct implementation of a critical section. In fact, it's even starvation-free. One can prove this using a short proof by contradiction.

An obvious disadvantage of this implementation is that it only implements mutual exclusion for two threads. In most cases, we are interested in providing mutual exclusion for many more threads. The next section will adapt the Peterson Lock to do just that.

3.2.2 Filter Lock

As mentioned in the previous section, the Filter Lock is a generalization of the Peterson Lock. In order to implement mutual exclusion for up to n threads, we create $n-1$ "levels" that a thread must traverse (3) before entering the critical section.

We generalize the notion of a two-element `flag` array with an n-element `level` array (1), where the value of `level[T]` indicates the highest level thread T is trying to enter. Each level k has an entry `victim[`k`]` (2), indicating that the thread specified in this entry wants to enter the level.

Initially, a thread is at level 0. We say that T is at level k for $k > 0$, when it completes the waiting loop (4) with $level[T] \geq k$. Note that this also means that a thread at level k is also at level $k-1$ and so on. A thread can complete the waiting loop when another thread wants to enter its level or no more threads are in front of it.

```
class FilterLock{
    volatile int[] level;
    volatile int[] victim;
    volatile int n;
    public FilterLock(int n){
        this.n = n;
        level = new int[n];                          // (1)
        victim = new int[n];                         // (2)
        for(int i = 0; i < n; i++){
            level[i] = 0;
        }
    }
    boolean Others(int id, int lev){
        for(int k = 0; k < n; k++){
            if(k != id && level[k] >= lev) return true;
        }
        return false;
    }
    public void lock(int id) {
        for(int i = 1; i < n; i++){                  // (3)
            level[id] = i;
            victim[i] = id;
            while(Others(id, i) && victim[i] == id){} // (4)
        }
    }
    public void unlock(int id){
        level[id] = 0;
    }
}
```

We can prove by induction that for $k \in [0, n-1]$, there are at most $n-k$ threads at level k. Therefore, we can draw the key conclusion that at most 1 thread can be at level $n-1$, i.e. in the critical section. Again using induction, we can also show that the Filter Lock is starvation-free, which implies that it is also deadlock-free. In summary, the Filter Lock correctly implements mutual exclusion.

However, one issue remains. We split the `lock()` method into two sections:

1. A *doorway* section, whose execution interval consists of a bounded number of steps.
2. A *waiting* section, whose execution interval may take an unbounded number of steps.

We can now define a notion of fairness.

Definition 3.2.1. A lock is *first-come-first-served* if, whenever, thread A finishes its doorway before thread B starts its doorway, then A cannot be overtaken by B.

In the case of the `FilterLock` we can define the first two instructions of the for-loop to be the doorway and the waiting loop to be the waiting section. Finding an execution which doesn't adhere to the above definition is left as an exercise.

In summary, the `FilterLock` does implement starvation-free mutual exclusion, but isn't fair according to the *first-come-first-served* principle.

3.2.3 Bakery Lock

The Bakery Lock is inspired by the number-dispensing machines often found in bakeries (or in the case of Switzerland: Post offices).

The implementation is relatively straightforward: We have a `flag` array, indicating whether the thread want to enter the critical section or not, and a `label` array, where $label[i]$ contains the label assigned to the thread with $ThreadID == i$. In the *doorway* section, we take a label, which is the current highest label incremented by one, and set our entry in the `flag` array. Note that it's possible for several threads to receive the same label at this point. We then remain in the *waiting* section until we have the lowest label (or in the case of identical labels: the lowest id) among all threads wanting to enter the critical section.

```
class BakeryLock {
    volatile boolean[] flag;
    volatile int[] label;
    final int n;
    public BakeryLock(int n){
        this.n = n;
        flag = new boolean[n];
        label = new int[n];
    }
    boolean Conflict(int id){
        for(int i = 0; i < n; i++)
            if(i != id && flag[i]){
                int diff = label[i] - label[id];
                if(diff < 0 || diff == 0 && i < id)
                    return true;
            }
        return false;
    }
    public void lock(int id){
        flag[id] = true;
        // Find maximum label and increment by 1
        label[id] = label[0];
        for(int i = 1; i < n; i++){
            label[id] = Math.max(label[i],label[id]);
        }
        label[id]++;
        // Waiting section
        while(Conflict(id)){}
    }
    public void unlock(int id){
        flag[id] = false;
    }
}
```

Proving that the `BakeryLock` is deadlock-free follows directly from the fact that the labels are strictly increasing and that the id assigned to each thread is unique. Therefore, there is always some thread with a unique (label, ThreadID) pair which is allowed to proceed into the critical section.

It's very easy to see that the `BakeryLock` is also *first-come-first-serve*, as once thread A has finished the doorway section, any subsequent thread B will always choose a higher label, thereby allowing A to go first. Note that any algorithm that is both deadlock-free and *first-come-first-serve* is also starvation-free.

We can prove that the `BakeryLock` provides mutual exclusion using a proof by contradiction, where we make use of the uniqueness of the (label, ThreadID) pair.

3.2.4 Spin Lock

When implementing mutual exclusion, there are two different alternative choices on what to do when we cannot immediately acquire a lock. The first choice would be to continue trying to acquire the lock. This is called *spinning* or *busy waiting*. The `FilterLock` and `BakeryLock` are such spin locks. As spinning takes up CPU cycles, this approach only makes sense on a multiprocessor system. The second choice would be to ask the operating system's scheduler to schedule another thread on your processor until the lock becomes available again. This is called *blocking*. Because switching from thread to another is expensive, blocking only makes sense if you expect the lock delay to be long.

Unfortunately, our lock implementations up until now won't work on most modern processors and compilers. This is because the compiler and the underlying hardware architecture do not guarantee memory operations to occur in-order. We tried to somewhat alleviate this issue by introducing the `volatile` keyword, but this only guaranteed reads and writes to the array reference to be in-order, not to the entries of this array.

We therefore arrive at a lock implementation using TAS and TTAS operations, which will be introduced in the sub-chapter "Non-Blocking Algorithms", for those unfamiliar with the concepts. When measuring

```
public class TASLock{
    AtomicBoolean state = new AtomicBoolean(false);
    public void lock(){
        while(state.getAndSet(true)){}
    }
    public void unlock(){
        state.set(false);
    }
}
```

```
public class TATASLock{
    AtomicBoolean state = new AtomicBoolean(false);
    public void lock(){
        do{
            while(state.get()){}
        }while(!state.compareAndSet(false, true));
    }
    public void unlock(){
        state.set(false);
    }
}
```

performance, we see that both `TASLock` and `TATASLock`, while slightly better, both perform poorly. The reasons for this can be found in modern hardware architecture:

- Calls to `getAndSet()` and `set()` force other processors to invalidate their cached copies of the `state` variable. This means that the next call to `get()` and `getAndSet()` will need to read from main memory. Continuing on like this, it's easy to see that nearly each call will read from main memory.
- Nearly each call reading from main memory also means that the shared bus will be under heavy use. This means that all threads using the bus will be slowed down considerably.

The `TATASLock` performs slightly better, as the call to `get()` still tries to read from the local cached copy.

One possibility to alleviate this problem would be to implement an exponential backoff. This means that everytime we don't manage to acquire the lock, we wait for a random amount of time $m \in [0, t^x]$, where x is the number of times we've failed to acquire the lock and t is the initial wait time.

3.3 Synchronization

As mentioned above, when we have a memory location which is neither thread-local nor immutable, we need to make sure we synchronize the resource properly across all threads. We therefore introduce the `synchronized` keyword. When a thread encounters the `synchronized` keyword, it will always first attempt to obtain the lock to the specified object. Until it obtains the lock, it will block.

---

example 3.3. We return to the example from section 3.2.1. Removing the incorrect synchronization primitive from the class and implementing mutual exclusion using the `synchronized` keyword gives us:

```
public class SynchronizedCounter extends Thread {
    static int cnt = 0;
    static Object lock = new Object();

    public void run(){
        for(int i = 0; i < 1000; i++){
            synchronized(lock){
                cnt++;
            }
        }
    }
}
```

Note that it's not possible to synchronize on a primitive value. One might be tempted to turn `cnt` into an `Integer`, but this would result in erroneous behavior. The reason for this is that primitive wrapper objects such as `Integer`s are immutable. Therefore, Java will create a new object everytime we increment the `cnt` variable.

Every Java object, including classes itself, not just their instances, has an associated lock, meaning that we can use it to enforce mutual exclusion using `synchronized`. We can also include the `synchronized` in the method signature:

```
public synchronized void someMethod(){...}
```

This is equivalent to:

```
public void someMethod() { synchronized(this){...} }
```

This means that before proceeding with execution of the method body, we will first acquire the lock on the `this` object, which is either an instance of the class or the class itself if the method is declared `static`.

Note that Java locks are reentrant.

Definition 3.3.1. A lock is *reentrant* if it can be acquired multiple times by the same thread.

When creating a concurrent program with mutable, shared state, we think in terms of what operations need to be atomic. Locks do pretty much just that: Changes made inside of a `synchronized` block appear to other threads (provided they also acquire the lock before reading mutable, shared fields) to take place instantaneously. Nonetheless, locks aren't all rainbows and sunshine. When we have large critical sections which are all protected by locks, we reduce the parallelizable fraction in our program by a lot. Thinking back to Amdahl's Law, we know that this drastically reduces the possible speedup of our program.

3.3.1 Conditions

Quite often, one might come across a case where a thread is only allowed to proceed once a certain condition has been met. One might be tempted to put the condition in a while-loop and let the thread spin until the condition is fulfilled. This solution, however, would be grossly inefficient, as this spinning thread would still be taking up a lot of CPU cycles, thereby slowing down all other threads.

Lucky for us, Java locks provide methods created for this exact purpose. All of the following methods can only be called inside of a `synchronized` block, i.e. when the thread holds the lock.

- `wait()`: Thread is moved into an inactive state and is inserted into the waiting queue. The thread releases the lock on which `wait()` was called.

- `notify()`: Some thread is removed from the waiting queue and moved into an active state. The thread that was just woken up will then proceed to attempt to re-acquire the lock normally. Note that we do not have any control over what thread is woken up.
- `notifyAll()`: All threads are removed from the waiting queue and moved into an active state. All these threads will then proceed to attempt to re-acquire the lock.

A very important thing to remember here is that the operating system might randomly wake up a waiting thread (spurious wakeups) or another thread might interrupt the waiting thread. Both of these cases would lead to the thread being woken up, possibly with an `InterruptedException`. In order to prevent the thread from continuing execution normally despite the condition not being fulfilled, we need to put it in a while-loop, as opposed to a simple if-statement.

The `java.util.concurrent.locks` library also provides similar methods. In order to use them, we first need to create a `Condition` object, on which we can then call `await()`, `signal()` and `signalAll()`. The definitions for these three methods are identical to the ones listed above provided by the `synchronized` primitive, but for the separation of waiting queues. Instead of associating a waiting queue with a lock, we associate it with the `Condition` object. This means that we can have several waiting queues for a single lock. The advantage of this will become clear momentarily.

There are two very important rules to using `wait()`/`notify()`:

1. **Always** enclose a `wait()` statement in a `while` loop. The reason for this is that the operating system may randomly wake up a thread, without us having ever called `notify()`. These events are called *spurious wakeups*. In addition, `notifyAll()` wakes up *all* threads, meaning also those for which the entry condition might not have been fulfilled, yet.
2. **Only** call `wait()` or `notify()` when holding the lock to the corresponding object. Java will throw an exception otherwise.

The two rules listed above also hold for the `Condition` interface.

---

example 3.4. Let's say we work for some nameless company housed in a concrete monolith. The company has received complaints from the basement-housed employees that the nearest women's bathroom is two floors up. We're tasked with converting the basement's men's bathroom to a unisex bathroom, with the following constraints:

- There cannot be men and women in the bathroom at the same time.
- There should never be more than three employees squandering company time in the bathroom.

All of this, of course, without causing a deadlock.
We came up with a system which allows whoever is standing outside of the bathroom to tell how many people and of which gender are inside. Therefore, the protocol for entering and exiting the bathroom will look as specified in the code.

This problem allows for a very nice visual interpretation of the difference between using `wait()`/`notify()` and using `Condition`s. The implementation using `wait()`/`notify()` creates a single queue in front of the bathroom entrance. Whenever someone leaves the bathroom, everybody in the queue will try to enter the bathroom. If they see that they can't, i.e. one of the requirements listed above doesn't hold, they'll get back in the queue. Otherwise, they'll enter the bathroom.

By contrast, using the `Condition` interface allows us to split this queue into two queues, one for male employees and the other for female employees. Whenever someone leaves the bathroom, they'll tell one of the queues that they're allowed to try and enter the bathroom. In order to have some fairness, when an employee is the last person to leave the bathroom, they'll always `signal` the queue of the opposite gender.

```java
public class Bathroom extends Thread{
    private volatile static int men = 0;
    private volatile static int women = 0;
    private static Object lock = new Object();

    public void femaleEnter() {
        synchronized(lock){
            while(men > 0 || women > 2){
                try{
                    lock.wait();
                } catch(InterruptedException e){}
            }
            women++;
        }
    }

    public void femaleExit() {
        synchronized(lock){
            women--;
            lock.notifyAll();
        }
    }

    public void maleEnter() {
        synchronized(lock){
            while(women > 0 || men > 2){
                try{
                    lock.wait();
                }catch(InterruptedException e){}
            }
            men++;
        }
    }

    public void maleExit() {
        synchronized(lock){
            men--;
            lock.notifyAll();
        }
    }
}
```

```java
public class Bathroom extends Thread{
    private volatile static int men = 0;
    private volatile static int women = 0;
    private static Lock l = new ReentrantLock();
    private static Condition menQueue = l.newCondition();
    private static Condition womenQueue = l.newCondition();

    public void femaleEnter() {
        l.lock();
        while(men > 0 || women > 2){
            try{
                womenQueue.await();
            } catch(InterruptedException e){}
        }
        women++;
        l.unlock();
    }

    public void femaleExit() {
        l.lock();
        women--;
        if(women == 0)
            menQueue.signalAll();
        else
            womenQueue.signal();
        l.unlock();
    }

    public void maleEnter() {
        l.lock();
        while(women > 0 || men > 2){
            try{
                menQueue.await();
            }catch(InterruptedException e){}
        }
        men++;
        l.unlock();
    }

    public void maleExit() {
        l.lock();
        men--;
        if(men == 0)
            womenQueue.signalAll();
        else
            menQueue.signal();
        l.unlock();
    }
}
```

It's important to note that even when the creation of several waiting queues could in theory be a lot more efficient, the `synchronized` primitive is implemented so much more efficiently in Java, that using the `Condition` interface is usually not worth the effort.

3.3.2 Semaphores

Up until now locks have always ensured that only a single thread enters the critical section. Yet in some cases, this might not be what we want. If, for example, we have a server that can support up to 100 requests at a time, we want to be able to allow up to 100 threads into the critical section. This is where semaphores come into play.

A semaphore is a generalization of mutual exclusion locks. Each semaphore S has a capacity N and provides the following operations:

- `acquire(S)`: Blocks until $N > 0$, then decrements N.
- `release(S)`: Increments N.

Note that both operations above should be considered to be atomic, in order to avoid data race or bad interleavings. The simplest approach to implementing semaphores is therefore to associate a lock with the two methods. Actually implementing a semaphore is left as an exercise.

3.3.3 Barriers

We now want to go one step further. We wish to create a barrier which blocks all threads up until a certain threshold N. Once the threshold has been reached all threads are allowed to continue execution. We distinguish between non-reusable and reusable barriers.

The non-reusable barrier has a relatively simple implementation, which can be seen in Fig. 1. Each arriving thread increments the counter (taking the lock to avoid race conditions) and attempts to acquire the `barrier` semaphore, thereby going into a blocked state. Once all threads have arrived, i.e. `count == threshold`, the `barrier` semaphore is set to 1, allowing one thread to pass. Whenever a thread passes the barrier it immediately releases it again so that the next thread may proceed with its execution. This method of acquiring and releasing the `barrier` semaphore is called a *turnstyle*.

```
class SimpleBarrier {
    private int threshold;
    private int count = 0;
    private Semaphore barrier = new Semaphore(0);

    public SimpleBarrier(int threshold){
        this.threshold = threshold;
    }

    public void await(){
        synchronized(this){
            count++;
        }
        if(count == threshold)
            barrier.release();
        try{
            barrier.acquire();
        }catch(InterruptedException e){}
        barrier.release();
    }
}
```

The reusable barrier is implemented in such a way that it can be reused once all waiting threads are released, i.e. it assumes no additional threads call the `await()` method before all waiting threads have been released.
The reusable barrier consists of two parts. First, the threads increment the counter, attempt to acquire the `barrier1` semaphore and go into a blocking state. Once the threshold is reached the `barrier1` semaphore is incremented so that the threads are allowed to pass the turnstyle and enter the second part of the barrier. Additionally, the `barrier2` semaphore is decremented, thereby making the second part of the barrier behave identically to the first part, only now decreasing the counter. In the second part of the barrier, threads will be allowed to pass the turnstyle once the counter's value is 0.
Note that once all threads have exited the barrier, all values have been restored to their original state, thereby allowing the barrier to be reused again.

At this point the reader is encouraged to think about why the different parts are necessary and what could go wrong if we changed anything about them. As an additional exercise, one should try to adapt the `CyclicBarrier` class such that it allows for later threads to start calling the `await()` method before all threads have exited the barrier.

```
class CyclicBarrier {
    private int threshold;
    private int count = 0;
    private Semaphore barrier1 = new Semaphore(0);
    private Semaphore barrier2 = new Semaphore(1);

    public CyclicBarrier(int threshold){
        this.threshold = threshold;
    }

    public void await(){
        synchronized(this){
            count++;
            if(count == threshold){
                try{
                    barrier2.acquire();
                }catch(InterruptedException e){}
                barrier1.release();
            }
        }
        try{
            barrier1.acquire();
        }catch(InterruptedException e){}
        barrier1.release();
        synchronized(this){
            count--;
            if(count == 0){
                try{
                    barrier1.acquire();
                }catch(InterruptedException e){}
                barrier2.release();
            }
        }
        try{
            barrier2.acquire();
        }catch(InterruptedException e){}
        barrier2.release();
    }
}
```

3.4 Lock Granularity

We've seen that the size of our critical section greatly influences the possible speedup we can achieve through parallelization. When programming with locks, it's common practice to start of essentially wrapping everything that remotely resembles a critical section in one huge lock. This is called *coarse-grained locking*. In this sub-section, we introduce different locking strategies which, with some customization, can be applied to many different concurrent data structures. We introduce these strategies at the hand of sorted linked lists.

---

example 3.5. A sorted linked list implements the following methods

- `add(x)`: adds `x` to the list in the position corresponding to the lexicographic ordering of x, returning `false` if `x` is already contained in the list.
- `remove(x)`: removes `x` from the list, returning `false` if the list doesn't contain `x`.
- `contains(x)`: returns `true` if the list contains `x`.

3.4.1 Coarse-Grained Locking

With *coarse-grained locking*, we take the sequential implementation of the data structure and synchronize all of its methods.

---

example 3.6. We only show the `remove` method for brevity:

```
public class SortedLinkedList<T> {
    private Node head;
    private Lock lock = new ReentrantLock();
    public SortedLinkedList(){
        head = new Node(Integer.MIN_VALUE);
        head.next = new Node(Integer.MAX_VALUE);
    }
    public boolean remove(T item){
        Node pred, curr;
        int key = item.hashCode();
        lock.lock();
        try{
            pred = head;
            curr = pred.next;
            while(curr.key < key){
                pred = curr;
                curr = curr.next;
            }
            if(key == curr.key){
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
        } finally {
            lock.unlock();
        }
    }
}
```

The disadvantage here is the considerable size of the critical section. As all the methods share a single lock and start by acquiring and end by releasing it, this implementation allows for barely any concurrency whatsoever. However, this strategy does have one advantage: its simplicity. Implementing it doesn't really require any effort on the side of the programmer.

3.4.2 Fine-Grained Locking

As a first step of improving *coarse-grained locking*, we can lock individual elements instead of the entire data structure. As a thread traverses the data structure, it locks each node when it first visits it and releases it once it has acquired the lock for the next node. This method of locking is called *hand-over-hand* locking. This method allows threads to traverse the data structure in a pipelined fashion.

In order to avoid deadlocks, it's important that all threads acquire the locks in some predefined order, e.g. in the case of the sorted linked list, all threads start at the head and don't try to "skip" nodes.

---

example 3.7. In order to avoid deadlocks, it's important that all threads acquire the locks in some predefined order. In the case of sorted linked lists, this requirement is easily met by always starting at the `head` sentinel and only proceeding in a hand-over-hand fashion.

```
public class SortedLinkedList<T> {
    private Node head;
    public SortedLinkedList(){
        head = new Node(Integer.MIN_VALUE);
        head.next = new Node(Integer.MAX_VALUE);
    }
    public boolean remove(T item){
        int key = item.hashCode();
        head.lock();
        Node pred = head;
        try{
            Node curr = pred.next;
            curr.lock();
            try{
                while(curr.key < key){
                    pred.unlock();
                    pred = curr;
                    curr = curr.next;
                    curr.lock();
                }
                if(key == curr.key){
                    pred.next = curr.next;
                    return true;
                } else {
                    return false;
                }
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }
}
```

We've now implemented an actual concurrent data structure, in the sense that several threads can actually operate on it simultaneously. But unfortunately, this strategy is still far from perfect. As threads iterate over this data structure in a pipelined fashion, the slowest thread sets the tempo for all threads that immediately follow it, meaning that a potentially "fast" thread might experience considerable slowdown. In addition, for large data structures, there is still a potentially long chain of lock acquisitions and releases.

3.4.3 Optimistic Locking

With *optimistic locking*, we take somewhat of a risk. We iterate the data structure without taking any locks. Once we've found the required elements we lock them and check if everything is still correct. If we find that in between finding the elements and taking the locks the state of the data structure changed to one where we can't execute our operation reliably anymore, we start over. As such conflicts are rare, we consider this approach to be optimistic.

Implementing such a verification method is non-trivial and requires careful thought. What state do we, as an operating thread, *expect* and how can we check that this expected state actually holds?

example 3.8. In the case of the sorted linked list, we verify the following two conditions:

- The `curr` node, i.e. the node we're operating on, is still reachable.
- `pred` still points to `curr`, i.e. the two nodes we're about to operate on are actually the ones we *should* operate on.

The reader is encouraged to check for themselves whether these two conditions are sufficient to guarantee a correct execution, e.g. by removing or weakening one condition and finding a case that would result in an incorrect execution.

```
public class SortedLinkedList<T> {
    private Node head;
    public SortedLinkedList(){
        head = new Node(Integer.MIN_VALUE);
        head.next = new Node(Integer.MAX_VALUE);
    }
    public boolean remove(T item){
        int key = item.hashCode();
        while(true){
            Node pred = head;
            Node curr = pred.next;
            while(curr.key < key){
                pred = curr;
                curr = curr.next;
            }
            pred.lock(); curr.lock();
            try{
                if(validate(pred, curr)){
                    if(key == curr.key){
                        pred.next = curr.next;
                        return true;
                    } else {
                        return false;
                    }
                }
            } finally {
                pred.unlock(); curr.unlock();
            }
        }
    }
    public boolean validate(Node pred, Node curr){
        Node node = head;
        while(node.key <= pred.key){
            if(node == pred)
                return pred.next == curr;
            node = node.next;
        }
        return false;
    }
}
```

We've now been able to alleviate most issues. Nonetheless, a few remain. First, by always calling `validate`, we're essentially iterating the list twice. If we then have a high amount of thread contention, i.e. a lot of threads operating on the same area of the data structure, these `validate` will often return `false`, thereby forcing most threads to re-iterate the entire list. Finally, we would like an operation as simple as the `contains` method to not have to acquire any locks whatsoever, as this does seem like a slight overkill.

3.4.4 Lazy Locking

Lazy synchronization builds on top of optimistic synchronization by adding a boolean `marked` field to each node, which - when `false` - holds the invariant that

- the node is in the set and
- the node is reachable.

The `remove` method now first *lazily* removes a node by setting its `marked` bit, then *physically* removes it, e.g. by redirecting the pointer from the previous node. We adjust the `validate` method so that it only checks whether the `marked` bit is set and whether the local state is still as expected. Therefore, the remainder of the `add` and `remove` methods can be left the same. Finally, we change the `contains` method such that it simply iterates the data structure without taking any locks and, if it finds the specified node, checks whether it's marked or not.

example 3.9. The concrete implementation of the sorted linked list according to the lazy locking strategy now looks as follows.

```
public class SortedLinkedList<T> {
    private Node head;
    public SortedLinkedList(){
        head = new Node(Integer.MIN_VALUE);
        head.next = new Node(Integer.MAX_VALUE);
    }
    public boolean remove(T item){
        int key = item.hashCode();
        while(true){
            Node pred = head;
            Node curr = pred.next;
            while(curr.key < key){
                pred = curr;
                curr = curr.next;
            }
            pred.lock(); curr.lock();
            try{
                if(validate(pred, curr)){
                    if(key == curr.key){
                        // logically remove
                        curr.marked = true;
                        // physically remove
                        pred.next = curr.next;
                        return true;
                    } else {
                        return false;
                    }
                }
            } finally {
                pred.unlock(); curr.unlock();
            }
        }
    }
    public boolean validate(Node pred, Node curr){
        // Both reachable and local state as expected
        return !pred.marked && !curr.marked && pred.next == curr;
    }
    public boolean contains(T item){
        int key = item.hashCode();
        Node curr = head;
        while(curr.key < key){
            curr = curr.next;
        }
        return curr.key == key && !curr.marked;
    }
}
```

3.5 Non-Blocking Algorithms

While locks allow us to relatively easily obtain atomicity of an operation, there are still several important disadvantages to locking:

1. Locks are **pessimistic** by design i.e. they assume the worst and enforce mutual exclusion
2. Severe **performance issues**: Even when uncontested, a lock acquisition must read and write from main memory. When the lock is contested, this degradation becomes much worse.
3. Locking is **hard**. A delayed thread might slow down all subsequent threads, deadlocks might occur, interrupt handlers become much harder to implement correctly, etc.

We therefore strive to create operations and data structures that don't use any locks whatsoever. To see how we can implement these we'll first have a very quick look at the most important different atomic primitives, which are also provided by the Java `java.util.concurrent` library. Then we'll direct our attention to implementing an actual lock-free concurrent data structure and the problems that might occur when implementing one.

3.5.1 Atomic Operations

All of the following methods are provided by the `java.util.concurrent.atomic` package. Note that some of them are not available in other programming languages.

The `testAndSet()` operation takes a byte or word which represents a boolean value, reads the stored boolean value and stores the value for `true`. It then returns the previously read value. We've already seen how we can implement a simple spin lock using this atomic operation.

The `compareAndSet()` method takes two arguments: an *expected* value and an *update* value. If the current stored value is equal to the expected value, it updates it to the specified update value, otherwise the value is left unchanged. Finally, it returns a boolean value indicating whether the value was replaced or not.

```
public boolean TAS(memref s){
    if(mem[s] == 0){
        mem[s] = 1;
        return true;
    } else
        return false;
}
```

```
public int CAS(memref a, int exptected, int update){
    oldVal = mem[a];
    if(expected == oldVal)
        mem[a] = update;
    return oldVal;
}
```

An `AtomicMarkableReference<T>` is an object that encapsulates both a reference to an object of type `T` and a boolean `mark` bit. Both of these fields can be updated atomically, either alone or both at once.
This leads us to a new version of the `compareAndSwap()` operation, namely the so-called double compare-and-swap, which takes four values: two *expected* values and two *update* values. It atomically checks whether both the reference and the mark bit are as expected, replaces them by the update values if so and returns a boolean value indicating whether the values were replaced or not.
It also provides the `attemptMark(T expectedReference, boolean newMark)` method, which checks whether the reference is as expected and, if it is, replaces the mark bit.
Finally, it's important to mention that the `get(boolean[] marked)` has a somewhat unusual interface, as it returns the object's reference value and stores the mark value in the boolean array which was passed as as argument.

3.5.2 ABA-Problem

Several different lock-free data structures were introduced during the lectures. For the sake of brevity, we'll focus on a lock-free stack implementation here.

Using the atomic operations introduced above, we can relatively easily implement a lock-free stack:

```
public Long pop() {
    Node head, next;
    do{
        head = top.get();
        if(head == null) return null;
        next = head.next;
    } while(!top.compareAndSet(head,next));
    return head.item;
}
```

```
public void push(Long item){
    Node newNode = new Node(item);
    Node head;
    do {
        head = top.get();
        newNode.next = head;
    } while(!top.compareAndSet(head, newNode));
}
```

We see that we create a new node for every single `push` operation. As an optimization, we implement a node pool which allows for reuse of the node objects:

```
public Node get(Long item){
    Node head, next;
    do {
        head = top.get();
        if (head == null) return new Node(item);
        next = head.next;
    } while(!top.compareAndSet(head, next));
    head.item = item;
    return head;
}
```

```
public void put(Node n) {
    Node head;
    do {
        head = top.get();
        n.next = head;
    } while (!top.compareAndSet(head, n));
}
```

When we adjust the stack implementation to use such a `NodePool`, we do indeed see a massive improvement in execution time. However, we see that the program exhibits erroneous behavior for some runs. This leads to a very common problem in lock-free concurrent programming, the **ABA Problem**.

Definition 3.5.1. The ABA problem occurs when one activity fails to recognize that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overall state has not been changed.

In the case of the lock-free stack with node reuse, this means the following scenario would exhibit an ABA-problem: We try to `pop` and observe that `head == a` and `head.next == b`. We then try to compare-and-set `head` to `b`. Suppose however, that another thread removes both `a` and `b`, pushes some other node `c` and then pushes `a`. We would then observe `head == a` as expected and set `head == b`, a node which is possibly not even in the stack at the time.

There are several possible alternatives to alleviate the ABA problem:

- **DCAS**: We can check whether both `head` and `head.next` are as expected, but doesn't exist on most platforms.
- **Garbage Collection**: Would eliminate the need for a `NodePool`, but is very slow and doesn't always exist either.
- **Pointer Tagging**: By incrementing the address bits made available by alignment, we can decrease the odds of the ABA problem occuring by a lot. Nonetheless, this doesn't actually alleviate the problem, only delay it.
- **Hazard Pointers**: We can associate an `AtomicReferenceArray<Node>` with the data structure where we temporarily store references which we've read and wish to write to in the future. Whenever we return a Node to the `NodePool`, we check whether its reference is stored in the `hazarduous` array. While this solution does work, the final product of a `NodePool` doesn't really improve performance when compared to regular memory allocation with a garbage collector.

3.6 Linearizability and Sequential Consistency

We've now looked at many different ways of implementing both blocking and non-blocking concurrent data structures and ensuring correctness. Interestingly enough, we did all this without ever properly defining what a correct parallel execution is. That is the aim of this section, defining different notions of correctness, some more strict than others, so that we may argue and prove things about our implementations.

3.6.1 Histories

We won't argue about implementations as a whole just yet. First, we'll consider one specific execution, called a *history*, and argue whether this specific history is correct.

A **method call** consists of a method *invocation* and *response*. An invocation that has no matching response is called *pending*. A **history** is a finite sequence of method invocations and responses. A **subhistory** is a subsequence of the events. *complete(**H**)* is the subsequence of a history **H** consisting of all matching invocation and response events in **H**. A history is *sequential* if every invocation is immediately followed by a matching response, i.e. when no method calls overlap.

---

example 3.10. We're given the following history:

```
A: s.push(2)
B: s.push(1)
A: void
B: void
A: s.pop()
B: s.pop()
A: 2
B: 1
```

We can see that the first two method calls and the last two method calls overlap.
An easy visualization of histories is through the use of time lines. In this case, it could look like this:

```
A s.push(1): *--------------*
B s.push(2):      *-------------*
B s.pop()->1:                                *----------------------*
A s.pop()->2:                                      *------------------*
```

There are many possibilities of representing a given history with a time line, as we have no way of telling how long a specific method call might take compared to others or how much time passes in between method calls.

We say that method call m_0 precedes a method call m_1 if m_0 finished before m_1 started, i.e. m_0's response event occurs before m_1's invocation event. We denote this by $m_0 \rightarrow_H m_1$.

3.6.2 Sequential Consistency

Definition 3.6.1. A program must fulfill two requirements to be deemed **sequentially consistent**:

- Method calls should appear to happen in a **one-at-a-time, sequential order**. This means that given a history **H**, every *thread subhistory* is a sequential history, i.e. every method call returns before the next one starts.
- Method calls should appear to take effect in **program order**. This means that we can order all method calls such that they
  1. are consistent with respect to the program order, i.e. they adhere to the ordering imposed by the thread subhistories

2. meet the object's sequential specification

Note that it's possible for a history to have several possible sequential orderings such that the above holds.

---

example 3.11. We continue our example from the previous section and check whether the history is sequentially consistent.

The first requirement states that method calls should appear to happen in a one-at-a-time, sequential order, i.e. that every thread subhistory is a sequential history. Indeed, we see that this requirement holds for both thread subhistories `H|A` and `H|B`, i.e. each method call finishes before the next one starts.

The second requirement asks us to find a method call ordering such that they are consistent w.r.t. program order and they meet the object's sequential specification. One such ordering is given by

`s.push(1)` → `s.push(2)` → `s.pop()->2` → `s.pop()->1`

Note that this is not the only possible ordering. Another possibility would've been

`s.push(2)` → `s.push(1)` → `s.pop()->1` → `s.pop()->2`

Recognizing that progam order, i.e. thread subhistories, is repsected by both of these orderings, is left up to the reader as an exercise.

3.6.3 Linearizability

The idea behind linearizability is that the concurrent history is equivalent to some sequential history. The rule is that if one method call precedes another, then the earlier call must have taken effect before the later call. By contrast, if two method calls overlap, then their order is ambiguous, and we are free to order them in any convenient way.

Definition 3.6.2. A history **H** is linearizable if it has an extension **H'** such that **H'** is complete and there is a legal sequential history **S** such that

1. complete(H') is equivalent to **S**, and
2. if $m_0 \rightarrow_H m_1$, then $m_0 \rightarrow_S m_1$

We can extend this notion by the following theorem:

Theorem 3.1. ***H** is linearizable if, and only if, for each object x, **H**|x, i.e. the subhistory of **H** with only all method calls concerning object x, is linearizable.*

Composability is important because we can design, implement and also prove things about concurrent systems in a modular fashion. Note that sequential consistency has no such property.

3.7 Consensus

To bring this chapter to a close, we want to define a notion of how *strong* a certain synchronization primitive, such as test-and-set, is. Using this we could for example show that there is an ordering on all synchronization primitives, such that no primitive at one level can be used for a wait-free or lock-free implementation of any primitives at higher levels.

The idea is as follows: Each class in the hierarchy has an associated *consensus number*, which is the maximum number of threads for which objects of the class can solve an elementary synchronization problem called *consensus*.

A *consensus object* provides a single method `decide()`. Each thread calls the `decide()` method with its input v at most once. The object's `decide()` method will then return a value meeting the following conditions:

- *consistent*: all threads decide the same value
- *valid*: the common decision value is some thread's input

In other words, a concurrent consensus object is linearizable to a sequential consensus object in which the thread whose value was chosen completes its call to `decide()` first.

We are interested in wait-free solutions to the consensus problem, that is, wait-free concurrent implementations of consensus objects. For historical reasons, any class that implements consensus in a wait-free manner is called a *consensus protocol*. We say that a class **C** solves *n-thread consensus* if there exists a consensus protocol using any number of objects of class **C** and any number of atomic registers. The *consensus number* of a class **C** is the largest n for which that class solves n-thread consensus.

A typical consensus protocol will look as follows:

```
public abstract class ConsensusProtocol<T> {
    protected T[] proposed = (T[]) new Object [N];
    // announce my input value to the other threads
    void propose(T value){
        proposed[TreadID.get()] = value;
    }
    // figure out which thread was first
    abstract public T decide(T value);
}
```

Let's quickly have a look at what this would actually look like in an example:

---

example 3.12. We're given an atomic implementation of the test-and-set primitive as specified in the previous section. We wish to show that test-and-set solves 2-thread consensus. To that end, we construct the following consensus protocol:

```
public class TASConsensusProtocol<T> {
    static int X = 0;
    protected T[] proposed = (T[]) new Object[2];
    // announce my input value to the other threads
    void propose(T value){
        proposed[ThreadID.get()] = value;
    }
    // figure out which thread was first
    public T decide(T value){
        propose(value);
        boolean val = TAS(X);
        if(val) return value;
        else return proposed[1-ThreadID.get()];
    }
}
```

We see that the call to `TAS(X)` will only return `true` once, namely the first time it's called. Otherwise, it'll always return false. It's therefore easy to see that the returned result is both consistent and valid. As the protocol doesn't contain any loops or other dependencies, it's wait-free. Therefore, we've provided a correct 2-thread consensus protocol and shown that test-and-set implements 2-thread consensus.

Note that we can't extend this implementation to n-treads, as a "loser" thread has no way of telling which entry of the `proposed` array it should use.

We've now defined what a consensus protocol is. Next, we would like to be able to prove things about concurrent object using these consensus protocols, e.g. that a concurrent object solves *at most* n-thread consensus for some n.

A *protocol state* consists of the states of the threads and the shared objects. When several threads call a consensus protocol, each thread makes *moves* until it decides on a value. Here, a move is a method call to a shared object, i.e. a transition from one state to the next.
A wait-free protocol's set of possible states forms a tree of finite depth, where each node represents a possible protocol state and each edge represents a possible move by some thread.
In the case of *binary consensus*, i.e. when the only possible decision values are 0 and 1, a protocol state is called *bivalent* if the decision value is not yet fixed, i.e. several final states are reachable in which the decision value differ. In contrast, we call a protocol state *univalent* if the outcome is fixed, i.e. all reachable final states decide on the same value. Finally, a *critical state* is a bivalent state in which any move leads to a univalent state.

---

example 3.13. One of the most important proofs in concurrency theory using consensus protocols shows that, surprisingly, atomic registers have consensus number 1. Note that this doesn't mean that only one thread is allowed to try and access an atomic register at a time, it simply means that we can't solve n-thread consensus for n>1 using atomic registers. The proof roughly looks as follows:

Suppose there exists a binary consensus protocol for two threads **A** and **B**. As every consensus protocol is wait-free, it goes through a finite number of bivalent states. We can therefore run the protocol until we reach a critical state s. Assume **A**'s next move carries it to a 0-valent state and **B**'s next move to a 1-valent state. We will now consider an exhaustive list of the methods that they might be about to call to cause such a state transition.

Suppose **A** is about to read a given register (we don't care about **B** here). Then there are two possible scenarios: either **B** moves first, carrying the protocol to a 1-valent state s', or **A** moves first, carrying the protocol to a 0-valent state s". Let's say that in both cases **B** would then run solo, eventually deciding 0 or 1, respectively. But as the read **A** issues only changes its thread-local state, states s' and s" should be indistinguishable to **B**, i.e. it should decide on the same value in both scenarios, a contradiction.

In the next case we consider, suppose that both threads are about to write to different registers, **A** to r_0 and **B** to r_1. Either **A** first writes to r_0 and **B** to r_1, carrying the protocol to 0-valent state, or **B** writes first, carrying the protocol to a 1-valent state. The problem is that, once again, the two resulting protocol states are indistinguishable from each other, which is a contradiction.

In the final case, suppose that both threads are about to write to the same register r. First, suppose that **A** writes first, carrying the protocol to a 0-valent state, and **B** then runs solo, eventually deciding 0. Second, suppose that **B** writes first and runs solo, therefore eventually deciding 1. The problem is that in both cases **B** overwrote whatever it is that **A** wrote into the register, i.e. **B** can't tell the difference between the two states. We've therefore reached another contradiction.

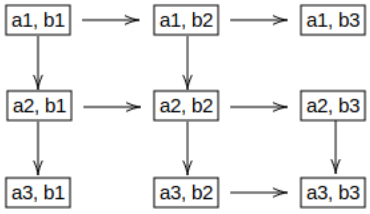
We've now shown that for all three possible types of consensus protocols one might construct using atomic registers, we reach a contradiction. It's therefore impossible to construct a 2-thread consensus protocol using atomic registers, i.e. atomic registers have consensus number 1.

In the proof above, one might be tempted to solve the problems leading contradictions using locks or by allowing one thread to go first, but then the protocol wouldn't be guaranteed to finish within in a bounded number of steps, i.e. it wouldn't be wait-free.

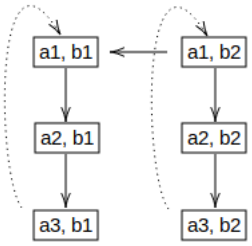
3.8 Exercises

5. Mutual Exclusion. A program is executed by two threads **A** and **B**. Each thread has three statements, labeled a_i and b_i, respectively. Statements a_3 and b_3 correspond to the critical section. In the following state diagrams, each state is described by the current statement for each thread in the form $\{a_i, b_i\}$. Arrows show the possible state transitions. For each of the diagrams, decide whether the program implements mutual exclusion correctly. If not, argue what the problem(s) are. Distinguish between livelock and deadlock.

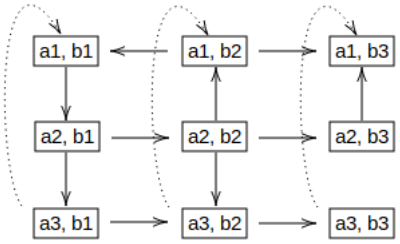
(i)



(ii)



(iii)



(iv)

6. Progress Conditions.

1. Consider the following implementation of a concurrent stack's `push` method.

```
public class MysteryStack<T> {
    AtomicReference<Node> top = new AtomicReference<Node>(null);
    static final int MIN_DELAY = 1;
    static final int MAX_DELAY = 42;
    Backoff backoff = new Backoff(MIN_DELAY,MAX_DELAY);

    protected boolean tryPush(Node node){
        Node oldtop = top.get();
        node.next = oldtop;
        return top.compareAndSet(oldtop, node);
    }
    public void push(T value){
        Node node = new Node(value);
        while(true){
            if(tryPush(node)){
                return;
            } else {
                backoff.backoff();
            }
        }
    }
}
```

   Name the progress conditions fulfilled by this implementation of the `push` method and describe your reasoning.
2. Prove that a wait-free program cannot use locks.
3. Suppose two cooks share a kitchen. In order to avoid both cooks using the same ingredient simultaneously (which would be really awkward), cooks need to know in advance what ingredients they need and then call the following program:

```
public class Kitchen {
    public void getIngredients(Ingredient... ingredients){
        for(Ingredient i : ingredients){
            i.lock.lock();
        }
    }
    public void releaseIngredients(Ingredient... ingredients){
        for(Ingredient i : ingredients){
            i.lock.unlock()
        }
    }
}
```

```
public class Ingredient {
    public final String name;
    public final Lock lock = new ReentrantLock();
}
```

What's the problem with this implementation? Describe a scenario that could cause the issue to occur and suggest a way to resolve it.

7. Peterson's.

1. Consider the following implementation of Peterson's lock.

```
boolean[] entryFlag = new boolean[2];

public void someProcess(){
    doStuff();
    while(true){
        entryFlag[threadID.get()] = true;
        while(entryFlag[1-threadID.get()]){}
        critical_section();
        entryFlag[threadID.get()] = false;
    }
}
```

   Draw the state diagram with all the relevant states and decide, based on the diagram, whether this implementation is correct or not. If not, describe what the problem is.
2. Another way to generalize the two-thread Peterson lock is to arrange a number of 2-thread Peterson locks in a binary tree. Suppose n is a power of two.

   Each thread is assigned a leaf lock which it shares with one other thread. Each lock treats one thread as thread 0 and the other thread as thread 1.

   In the tree-lock's `lock` method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's `unlock` method for the tree-lock unlocks each of the 2-thread Peterson locks that thread has acquired, from the root back to its leaf. At any time, a thread can be delayed for a finite duration (In other words, thread won't drop dead). For each property, either sketch a proof that it holds, or describe a (possibly infinite) execution where it's violated.
   1. Mutual exclusion
   2. Freedom from deadlock
   3. Freedom from starvation

8. Fairness. Recall the definition of *first-come-first-serve*: A lock is first-come-first-served if, whenever thread A finishes its doorway before thread B starts its doorway, A cannot be overtaken by B.

Describe an execution of the Filter Lock which doesn't adhere to the definition of first-come-first-served.

9. Lock Granularity.

1. Explain why the fine-grained locking algorithm isn't subject to deadlock.
2. Show a scenario in the optimistic locking algorithm where a thread is forever attempting to delete a node.

10. Linearizability and Sequential Consistency.

1. Draw a timeline for each of the following histories:

```
 A: s.push(2)
B: s.top()
A: void
C: push(1)
C: void
B: 1
C: s.push(2)
A: s.pop()
A: 1
C: void
```

```
B: r.write(1)
A: r.read()
C: r.write(2)
A: 1
C: void
B: void
B: r.read()
B: 2
```

```
A: s.push(x)
B: s.pop()
B: x
A: void
A: s.push(y)
B: s.push(z)
A: void
A: s.pop()
A: y
B: void
```

2. For each of the histories above, are they sequentially consistent? Linearizable? Justify your answer.
3. Is the result of composing several sequentially consistent objects itself always sequentially consistent? Either justify your answer or give a counterexample.

11. Consensus.

1. Argue why every n-thread with $n > 1$ consensus protocol has a bivalent initial state.
2. Which of the following are valid implementations wait-free consensus for the given number N of threads (or an arbitrary number of threads if no N is specified)? Justify your answer.

```
public class RandomConsensus extends ConsensusProtocol{
    private volatile int i = -1;
    public Object decide(Object value){
        propose(value);
        if(i == -1){
            i = ThreadID.get();
            return proposed[ThreadID.get()];
        }
        else {
            return proposed[1-ThreadID.get()];
        }
    }
}
```

(i) N = 2

```
public class CASConsensus extends ConsensusProtocol{
    private final int FIRST = -1;
    private AtomicInteger r = new AtomicInteger(FIRST);
    public synchronized Object decide(Object value) {
        propose(value); // Proposed array has size N
        int i = ThreadID.get();
        if (r.compareAndSet(FIRST, i))
            return proposed[i];
        else
            return proposed[r.get()];
    }
}
```

(ii)

```
public class CASConsensus extends ConsensusProtocol{
    private final int FIRST = -1;
    private AtomicInteger r = new AtomicInteger(FIRST);
    public Object decide(Object value) {
        propose(value); // Proposed array has size N
        int i = ThreadID.get();
        if (r.compareAndSet(FIRST, i))
            return proposed[i];
        else
            return proposed[r.get()];
    }
}
```

(iii)

3.9 Solutions

5. Mutual Exclusion.

(i) While this implementation does provide mutual exclusion in the sense that no two processes are in the critical section at the same time, it's not correct either. We see that the processes continuously transition from one state to the next, without ever entering the critical section, i.e. this implementation contains a livelock.

(ii) This time around the implementation doesn't contain a dead- or livelock. However, we see that there is a path of state transitions, i.e. an execution, which results in both threads being in the critical section at the same time. Therefore, this implementation isn't correct.

(iii) We see that the threads never get stuck in a state and there is path which doesn't result in one of the threads being in the critical section, i.e. the program contains neither a dead- nor a livelock. In addition, we see that there is no possible sequence of state transitions which results in both threads being in the critical section at the same time. Therefore, this implementation is correct. Note that thread **B** never enters the critical section. This doesn't mean the implementation is incorrect, only that it suffers from starvation.

(iv) We see that there is a cycle of state transition which doesn't result in one of the threads entering the critical section, i.e. the implementation suffers from livelock. In addition, there is a sequence of state transitions which results in both threads being in the critical section at the same time. In summary, this implementation suffers from livelock and doesn't provide mutual exclusion, i.e. it's incorrect.

6. Progress Conditions.

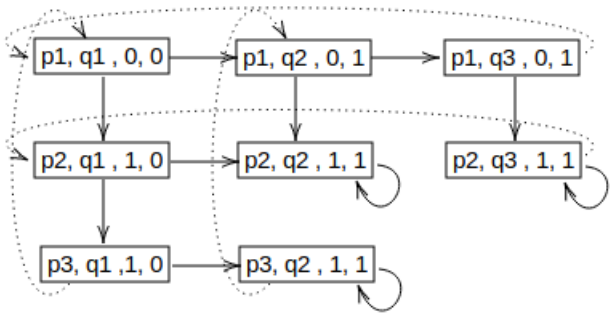
1. We see that this implementation is identical to the lock-free stack implementation from section 3.5 with an added backoff mechanism. Nonetheless, an argumentation is required.

   At no point during execution of the `push` method does the thread transition into a blocked state. Therefore, this implementation must be lock-free. This also implies that it's deadlock-free. We do realize, however, that a thread isn't guaranteed to successfully push a node within a finite amount of time, e.g. when other threads keep popping/pushing nodes just before the thread calls `compareAndSet`. Therefore, this implementation isn't wait-free. In addition, this also means that the method isn't starvation-free.
2. In a wait-free algorithm, any thread needs to be able to make progress independently of other threads. An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes. Thus, we cannot use locks to implement a wait-free algorithm (assuming locks are used to protect shared resources), as thread A might obtain the lock, then become really slow. During this period all other threads that want to obtain the lock are blocked and cannot make any progress.
3. This implementation suffers from deadlock. Suppose cook A wants to use ingredients I_1 and I_2 and cook B wants to use I_2 and I_1. Both proceed to call `getIngredients` simultaneously and obtain the first lock, i.e. cook A holds the lock for I_1 and cook B for I_2. Neither cook can now make progress as they are waiting for the other to release the lock.

   One possible solution would be to impose an ordering on the ingredients in which their locks must be obtained. For example, we could enforce that locks must always be obtained in alphabetical order of the ingredients' names. Note that this makes the assumption that no two ingredients have the same name, which is a sensible assumption to make when working in a kitchen.

7. Peterson's.

1. As can be read off the diagram below, this implementation suffers from livelock, e.g. when both threads reach the while-loop simultaneously. In this case, both threads would observe that the `entryFlag` field of the other thread is set and would therefore both decide to enter the while-loop. Therefore, this implementation is incorrect.



2. - **Mutual exclusion**: We see that only two threads share a single leaf node. As the Peterson lock provides mutual exclusion for two threads, the leaf node also provides mutual exclusion. If two subtrees of a single node both provide mutual exclusion, at most two threads can arrive at this node. As this node is also a Peterson lock, it provides mutual exclusion for two threads. By induction, we follow that this generelization of the Peterson's lock provides mutual exclusion.
   - **Freedom from deadlock**: Similarly, we show inductively that the generalization provides freedom from deadlock.
   - **Freedom from starvation**: Unfortunately, this property is violated, as is shown by the following execution. Suppose $n = 4$. We name the two leaf nodes P_1 and P_2 and the root P_0. Assume thread A first acquires P_1, but thread B manages to obtain P_0 first, i.e. A is blocked. Later, thread B releases both P_0 and P_2, but thread C immediately re-acquires them, then A must remain in a blocked state. Threads B and C continue to take turns this way, therefore making thread A block indefinitely. Thread A isn't guaranteed to make progress at any point, i.e. the generalization isn't starvation-free.

8. Fairness. Suppose $n = 3$, thread A is stuck at level 0, thread B at level 1 and thread C is at level 2, i.e. in the critical section. C finishes its critical section and releases the lock, thereby setting its level to 0. Now that there are no more threads at any higher levels, B enters the critical section. B finishes its critical section and also sets its level to 0. Let's say both B and C attempt to acquire the lock again, ending up with `victim[1] == C`. While A could, in theory, enter the next level, it might happen that B is faster and enters level 1 (and subsequently level 2, i.e. the critical section). Therefore, despite having completed the doorway section before B started its doorway sections, A was overtaken by B. Thus, the Filter Lock isn't fair.

9. Lock Granularity.

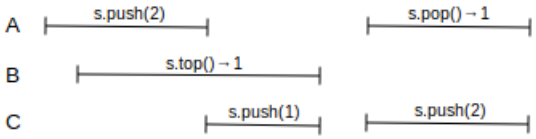
1. In order for a deadlock to occur, the dependency graph must allow for a cyclic dependency. In the fine-grained locking algorithm, all threads obtain the locks in a set order. In the case of the sorted linked list, for example, all threads acquire locks in increasing order. Therefore, a cyclic dependency becomes impossible.

   Suppose that the `add` method of the sorted linked list first acquired the lock for `curr`, then for `pred`. If another thread is executing the `remove` method, it might've already acquired the lock for `pred`, and be attempting to acquire the lock for `curr`. Both threads would be waiting for the other to release the lock. Thus, two threads acquiring locks in different orders can result in a deadlock.

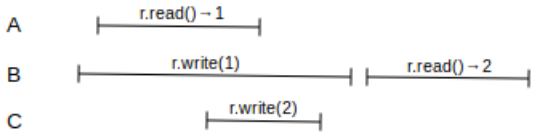
2. Suppose thread A wishes to remove node c. It can happen that the `validate` method always returns `false`, thereby forcing A to always retry and never successfully remove c. A scenario in which the `validate` method always returns `false` would be when other threads are continuously adding and removing nodes in between c and its predecessor, i.e. the comparison `pred.next == curr` may evaluate to `false` everytime. As long as no other thread removes c, A will continue to try to remove c.

10. Linearizability and Sequential Consistency.

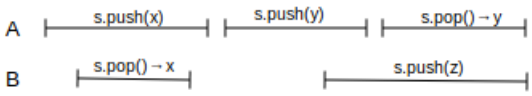
1. Note that there many possible different solutions. These are only some of them.



(i)



(ii)



(iii)

2. (i) We can order this history as

   `s.push(2)`→`s.push(1)`→`s.top():1`→`s.pop():1`→`s.push(2)`

   We see that this ordering preserves the precedence relationships and describes a legal sequential history. Therefore, this history is linearizable and thereby also sequentially consistent.

   (ii) We can order this history as

   `r.write(1)`→`r.read():1`→`r.write(2)`→`r.read():2`

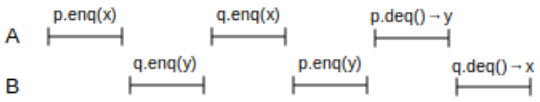
   We see that this ordering preserves the precedence relationships and describes a legal sequential history. Therefore, this history is linearizable and thereby also sequentially consistent.

   (iii) We can order this history as

   `s.push(x)`→`s.pop():x`→`s.push(y)`→`s.pop():y`→`s.push(z)`

   We see that this ordering preserves the precedence relationships and describes a legal sequential history. Therefore, this history is linearizable and thereby also sequentially consistent.

3. No, it's not. Consider the following execution. It's easy to see that both q and p are sequentially consistent, yet the execution as a whole is not sequentially consistent.



11. Consensus.

1. Consider the initial state where A has input 0 and B has input 1. If A finishes the protocol before B takes a step, then A must decide 0, because a valid consensus protocol must decide some thread's input, and 0 is the only input it has seen. Symmetrically, if B finishes the protocol before A takes a step, then B must decide, because it must decide some thread's input, and 1 is the only input it has seen. It follows that the initial state where A has input 0 and B has input 1 is bivalent.

2. (i) Suppose two threads A and B simultaneously see `i == -1`, i.e. they both enter the if-block, and both have different input values. They will then decide on their own input values, which means the `decide` method returns two different values, i.e. it's not correct.

   We could've also argued that the `volatile` keyword simply ensures that reads and writes are atomic. As we've shown that atomic registers can't solve two-thread consensus, we would've immediately seen that this protocol can't be correct.

   (ii) Note that the `decide` method is declared as `synchronized`. We've shown that wait-free implementations can't use locks, this consensus protocol isn't wait-free.

   (iii) The threads share an `AtomicInteger` object, initialized to a constant `FIRST`, distinct from any thread index. Each thread calls `compareAndSet` with `FIRST` as the expected value, and its own index as the new value. If thread A's call returns `true`, then that method call was first in the linearized order, so A decides its own value. Otherwise, A reads the current `AtomicInteger` value, and takes that thread's input from the `proposed[]` array.

4 Additional Topics

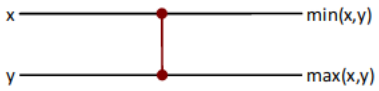
This chapter discusses those topics that didn't fit in nicely anywhere in the script. If you, as a reader, find that these topics do fit in nicely elsewhere, possibly with some slight rearrangement of the topics, feel free to send us an email.

4.1 Sorting Networks

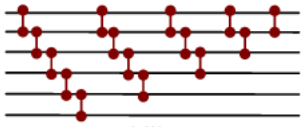
In this chapter we turn our attention to parallel sorting algorithms, which we'll represent using so-called *sorting networks*. We'll see what common sorting algorithms look like when parallelized and how to prove or disprove that a sorting network actually sorts any arbitrary sequence of numbers correctly.

We know that a sorting algorithm, with the exception of some specialized cases, cannot do better than $O(nlogn)$ in the worst case. The obvious next step is to try and break this lower bound by using the multiprocessors available to us, i.e. by parallelizing sorting algorithms.

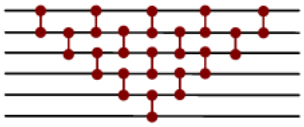
First off, we need to define what a sorting network looks like. To that end, we define the *comparator*, which swaps two values, such that the higher value ends up at the bottom.



We will now look at what good old Bubble Sort looks like as a sorting network. The idea of Bubble Sort is essentially that we first find the highest value among the first n values of the array, then among the first $n-1$ values, and so on until we've sorted the entire array. As a sorting network, that looks like this:



Let's take a quick break and think about what an actual actor could look like. From this visual representation it's pretty obvious which comparators can be executed in parallel. We "shift" all the comparators as far to the left as possible without having several comparators act on the same variables simultaneously, as this would result in a data race. This gives us the following sorting network for parallel Bubble Sort:



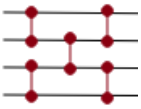
Let's take a quick break and think about what an actual actor could look like. Given an infinite number of processors, parallel Bubble Sort will therefore take $2n-3$. We can improve this even further to n by rearranging the comparators such that all "even" resp. all "odd" comparators operate in parallel. The algorithm can be improved even further, but that's far beyond the scope of this course.

Having seen what a sorting network is we're now interested in proving that a particular sorting network will sort any arbitrary sequence of numbers. As there are of course infinitely many such sequences, we need some easier way of proving such a property. This leads us to the *zero-one principle*.

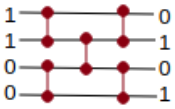
Theorem 4.1. *If a network with n input lines sorts all possible input sequences of 0s and 1s into a non-decreasing order, it will sort any arbitrary sequence of n numbers in non-decreasing order.*

By using this principle, we can prove the correctness of a given sorting network with relative ease.

example 4.1. We're given the following sorting network:



Tasked with determining whether this sorting network is correct or not, we turn our attention to the *zero-one-principle*. For the sake of brevity, we'll skip ahead to the conclusion and see that it's actually not correct, as can be seen by the following sequence:



4.2 Transactional Memory

We've learned how to correctly implement concurrent programs using locks, but a few problems still remain:

- **Deadlocks**: Threads might attempt to acquire locks in a different order, thereby introducing a cyclic dependency
- **Convoying**: A thread might be descheduled while other threads queue up waiting for it to continue
- **Priority Inversion**: Lower priority threads might hold a resource that a higher priority thread is waiting for
- Association of data and locks is established by **convention**, i.e. a correct use of the program by future developers depends on reasonable documentation
- **Non-composable**: Changing anything about the locking scheme requires changing the whole program
- **Pessimistic by design**: Assumes the worst and introduces expensive mutual exclusion as a consequence

We might attempt to resolve some of these problems using compare-and-swap, but, as one might see when attempting to implement a lock-free sorted linked list, this is often not enough. Double-compare-and-swap could be enough, but at this point it's only a theoretical concept. We therefore need some way of allowing the programmer to declare a block of code atomic without placing a large burden on him.

This is where *transactional memory* comes into play. In transactional memory, we define atomic code sections called *transactions*, which guarantee:

1. **Atomicity**: Changes made by transactions are made visible atomically, i.e. other threads either observe the initial or final state, but nothing in between. Note that this is done without mutual exclusion.
2. **Isolation**: While a transaction is running, effects from other transactions aren't observed.

It is convenient to say something about how transactions are implemented. Transactions are executed *speculatively*: as a transaction executes, it makes *tentative* changes to objects. If it completes without a synchronization conflict, it *commits* (the tentative changes become permanent) or it *aborts* (the tentative changes are discarded.

---

example 4.2. As one might have seen during the lecture, implementing a concurrent bounded producer/consumer queue requires a lot of thought (see: Sleeping Barber). However, using transactions implementing one becomes nearly trivial. Below we've depicted the enqueue method of such a bounded queue.

```
public void enq(T x) {
  atomic {
    if (count == items.length)
      retry;
    items[tail] = x;
    if (++tail == items.length)
      tail = 0;
    ++count;
  }
}
```

The method enters the `atomic` block, and tests whether the buffer is full or not. If so, it calls `retry`, which *rolls back* the enclosing transaction, pauses it, and restarts it when the object's state has changed. If not, it enqueues the item, increments the index modular the length of the queue, and exits the `atomic` block.

Transactional memory sound fantastic, but where is it implemented? There have been attempts to implement transactional memory in hardware and while these implementations can certainly be fast, their resources are bounded and they can therefore often not handle larger transactions. The implementation that we will work with is directly in software in parallel programming languages, which allows for greater flexibility, though achieving good performance might be more challenging.

4.2.1 Concurrency Control

In order to guarantee that a running transaction always sees consistent data, transactional memory implements a *concurrency control* (CC) mechanism. The CC will *abort* a transaction when a *conflict* occurs. An example of a conflict is a when a transaction that hasn't yet committed has read a value that is later changed by a committed transaction. Once a transaction is aborted, it can either be retried automatically (not necessarily immediately) or the user is notified.

Synchronization conflicts cause transactions to abort, but it's not always possible to halt a transaction's thread immediately after the conflict occurs. Instead, such *zombie* transactions may continue to run even after it has become impossible for them to commit. This raises the question: How do we prevent such a zombie from seeing an inconsistent state?

---

example 4.3. Such an inconsistent state could arise if an object has two fields x and y, where each transaction preserves the invariant that $x > y$. Transaction **Z** reads y, and sees the value 2. Transaction **A** changes x and y to 1 and 0, respectively, and commits. Suppose **Z** keeps running, i.e. **Z** is a zombie, since it will never commit. **Z** will later see an inconsistent state, when it reads the value 1 for x.

One might be tempted to say we can ignore this problem, as **Z** will never commit anyways, i.e. its updates will be discarded. But unfortunately, life isn't that easy. Let's say that after reading x, **Z** computes the value for $\sqrt{x-y}$. Under normal circumstances, evaluating this expression shouldn't throw an exception. **Z** has read an inconsistent state and will now throw an exception, which, depending on the implementation, could crash the program.

As there is no "practical" way of preventing exceptions in a programming model where invariants such as $x > y$ cannot be relied on, Transactional memory systems mostly guarantee that all transactions, even zombies, see consistent states.

Transactional memory organizes mutable state into *atomic objects*, to which it associates *timestamps*, which indicates when the object was last changed by a committed transaction. A transaction will keep a local *read-set* and a local *write-set*, holding all locally read and written objects, respectively. When a transaction calls *read*, it will check if the object is in the write-set and use this new version if it is. If not, it will check if the object's time stamp is smaller than the transaction's birthdate. If the object has been changed by some other transaction after the transaction's birthdate, it will abort. Otherwise, it'll add a new copy of the object to the read-set. When a transaction calls *write*, it'll create a copy of the object in the write-set, if there isn't one already.

When a transaction attempts to commit, all objects of the read-set and write-set are locked, and the timestamps of the objects in the read-set are compared to the birthdate of the transaction. If any of the objects were changed after the transaction started, it's aborted. Otherwise, all objects in the write-set are copied back to global memory with their new timestamps. All locks are released and a "commit" is returned.

4.2.2 Scala-STM

ScalaSTM is a Java API through which we can access the methods provided by the Scala STM library. ScalaSTM is a so-called *reference-based STM*, which means that mutable state, i.e. state which can only be modified *inside a transaction*, is put into special variables.

```
private final Ref.View<Integer> count = STM.newRef(0);
```

Arrays can be declared as follows:

```
private TArray.View<E> items = STM.newTArray(capacity);
```

Everything else is immutable, which means any other variable accessed inside an atomic block must be declared `final`.

We can create an atomic block as follows:

```
STM.atomic(new Runnable(){...});  or  STM.atomic(new Callable<T>(){...});
```

Note that the passed `Runnable` or `Callable` object must implement the `public void run()` or the `public T call()` method, respectively.

---

example 4.4. We can now try to implement the `enq()` method of the bounded queue using ScalaSTM.

First, we need to distinguish between mutable state and immutable state. We see that we modify the variables `count, tail` and `items`. As these variables should indeed only be accessed from within a transaction, they are part of the mutable state. The method also accepts a parameter `x`. As we only read object and don't actually try to modify, we'll include it in the immutable state. All in all, our class will look something like this (omitting the other methods for brevity):

```
public class CircularBufferSTM<T> {
    private final Ref.View<Integer> count = STM.newRef(0);
    private final Ref.View<Integer> tail = STM.newRef(0);
    private TArray.View<T> items;
    public CircularBufferSTM(int capacity){
        items = STM.newTArray(capacity);
    }
    public void enq(final T x){
        STM.atomic(new Runnable(){
            public void run(){
                if(count.get() == items.length())
                    STM.retry();
                items.update(tail.get(), x);
                tail.set((tail.get()+1) % items.length());
                STM.increment(count, 1);
            }
        });
    }
}
```

4.3 Message Passing

We've now seen a lot of different ways of dealing with processes concurrently accessing the same data and we've come to one definite conclusion, it's hard. We might try to avoid sharing state by partitioning the dataset using the `ForkJoin` framework, we might use locks to guarantee consistency or we might even work with transactional memory, but all of these approaches are still difficult to work with, near impossible to implement or both. Taking a step back, we see that the root cause of the problem is the fact that we have shared mutable state. So, what if we didn't? This is the core idea of *message passing*, namely to have processes run in isolation and only communicate via messages which must be explicitly received.

4.3.1 Message Passing Interface

The *Actor Model* is a model for concurrent computation. It uses *actors* as computational agents which react to received messages. Received messages are mapped to a set of messages sent to other actors, a new behavior and a set of new actors created. In other words (and slightly informal), an actor is a thread which reacts to received messages and has a local state.

---

example 4.5. Let's take a quick break and think about what an actual actor could look like.

A **distributor** is a type of actor which forwards received messages to a set of actors in a round-robin fashion. There are two questions that we need to answer to be able to model an actor. What *local state* should be kept by the actor and what should the actor do upon receiving a message?

The first question can be answered quickly. We know that we have a set of actors. In order to guarantee that the messages are distributed in a round-robin fashion, we store the actors in an array and we keep an index which indicates the entry in the array which contains the next actor to send a message to.

Now that we know what internal state we're going to keep, defining the behavior of the actor upon receiving a message seems obvious. The message can immediately be forwarded to the actor stored in the array entry indicated by the stored index. Then, we increment the index modular the array length. We also need to consider adding control commands which could, for example, allow us to add or remove actors from the set of stored actors.

Note that in the actor model messages are sent in an *asynchronous fashion*, i.e. the sender places the message into the buffer of the receiver and continues execution. In contrast, when the sender sends *synchronous* messages, it blocks until the message has been received.

As always, Java has a library for it. The *Message Passing Interface* (MPI) is a library which provides us with a message passing API which we can use to write message-passing programs.

MPI collects processes into groups, where each group can have multiple "colors" and a group paired with its color uniquely identifies a communicator. Initially, all processes are collected in the same group and communicator `MPI_COMM_WORLD`. Within each communicator, a process is assigned a unique number to identify it by, called a *rank*.

4.3.2 Point-to-Point Communication

The methods to send and receive messages are declared as follows:

```
public void Send(
    Object buf,        // Pntr to data to be sent
    int offset,
    int count,         // Number of items to be sent
    Datatype datatype, // Datatype of items
    int dest,          // Destination process id
    int tag            // Data id tag
);
```

```
public void Recv(
    Object buf,        // Pntr to buffer to receive to
    int offset,
    int count,         // Number of items to be received
    Datatype datatype, // Datatype of items
    int source,        // Source process id
    int tag            // Data id tag
);
```

Note that in the case of the `Recv` method it's not necessary to declare `src` or `tag`, instead one could use `MPI_ANY_SOURCE` or `MPI_ANY_TAG`. Both methods are declared in the `COMM` class, i.e. can only be used in combination with a communicator.

The two methods declared above are so-called *blocking* operations, which means they won't return until the action has been completed *locally*. Their *non-blocking* variants `recv` and `send` return *immediately*. We can also send synchronous messages, i.e. the operation blocks until the message has been received, using `Ssend`. Note that the `Recv` method declared above already is synchronous.

To summarize and complete what we've learned up until now, we can write a simple MPI program using the following six functions:

- `MPI.Init()`: initialize the MPI library (first routine called)
- `MPI.COMM.Size()`: get the size of a communicator `COMM`
- `MPI.COMM.Rank()`: get the rank of the calling process in the communicator
- `MPI.COMM.Send()`: Send a message to another process in the communicator
- `MPI.COMM.Recv()`: Receive a message from another process in the communicator
- `MPI.Finalize()`: Clean up all MPI state (last routine called)

---

example 4.6. Given an array of integers, let's compute the number of prime factors for each entry. The resulting array should be present in the process with rank 0 at the end. For simplicity, assume the array length is divisible by the number of processes.

```
public static void computePrimeFactors(int[] arr){
    // We assume MPI.Init() has already been called
    int size = MPI.COMM_WORLD.Size();
    int rank = MPI.COMM_WORLD.Rank();
    int partSize = arr.length / size;
    int[] res = new int[partSize];
    for(int i = rank*partSize, j = 0; i < arr.length; i++, j++){
        if(rank == 0)
            arr[i] = primeFactors(arr[i]);
        else
            res[j] = primeFactors(arr[i]);
    }
    if(rank != 0)
        MPI.COMM_WORLD.Send(res, 0, partSize, MPI.INT, 0, 42);
    else {
        for(int i = 1; i < size; i++){
            MPI.COMM_WORLD.Recv(arr, i * partSize, partSize, MPI.INT, i, 42);
        }
    }
    // We assume MPI.Finalize() will be called
}
```

4.3.3 Group Communication

Up until now we've only considered communication between two specific processes. MPI also supports communications among groups of processes. In theory, this isn't really necessary to write a program, but it is essential to performance.

`MPI.COMM.Reduce()` takes an array of input elements on each process and returns an array of output elements to the specified root process. The output elements contain the reduced result. `MPI.COMM.ReduceAll()` does the exact same thing, only returning the result to all processes instead of a single root process.

```
public void Reduce(
    int sendoffset,
    Object recvbuf,
    int recvoffset,
    int count,
    Datatype datatype,
    Op op,
    int root
);
```

`MPI.COMM.Scan()` takes an array of input elements on each process and returns distinct arrays of output elements to each process, where the operation is applied iteratively, with the final result array being returned

to the specified root process. The method declaration is identical to the declaration of `Reduce()`, but for the method name.

Using `MPI.COMM.Bcast()` one process can *broadcast*, i.e. send, the same data to all processes in a communicator. Note that both the receiver and sender processes call the same method.

```
public void Bcast(
    int sendoffset,
    Object buf,
    int count,
    Datatype datatype,
    int root
);
```

Finally, we arrive at the `MPI.COMM.Scatter()` and `MPI.COMM.Gather()` methods. `Scatter()` involves a root process sending data to all processes in a communicator. The main difference to `Bcast()` is that `Scatter()` sends *chunks of an array* to different processes, while `Bcast()` sends the *entire array* to all processes. `Gather()` does the exact inverse of `Scatter()`, in that it takes data from many processes and gathers them to a single root process. The elements are ordered by the rank of the process from which they were received.

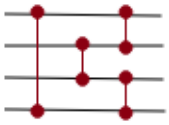
```
public void Scatter(
    Object sendbuf,
    int sendoffset,
    int sendcount,
    Datatype sendtype,
    Object recvbuf,
    int recvoffset,
    int recvcount,
    Datatype recvtype,
    int root
);
```

It's important to know that the method declarations aren't necessarily accurate, as there are many different MPI implementations with variations in parameters and parameter ordering.

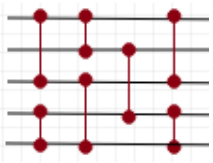
4.4 Exercises

12. Sorting Networks.

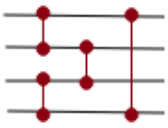
1. How many inputs does one have to check correctness of a sorting network that takes integer inputs of length 4, using the 0-1 principle?
2. For each of the following sorting networks, decide whether they're correct. If not, give a counterexample.



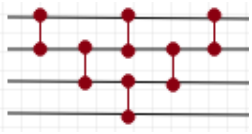
(i)



(ii)



(iii)



(iv)

3. Draw the parallel sorting network corresponding to insertion sort for an input sequence of length 5.

13. Software Transactional Memory.

1. Implement the `deq` method of the `CircularBufferSTM<T>` class.
2. Implement a bounded stack (i.e. a stack which isn't allowed to grow past a certain size) using ScalaSTM.

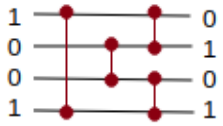
14. Message Passing.

1. Adapt the `computePrimeFactors` method such that the resulting array is present in all processes at the end. *Hint: Look up the* ***`Allgather`*** *method.*
2. Implement a method `randAvg` that works as follows:
    1. Generate a random array of numbers on the root process (process 0).
    2. Distribute the numbers among all processes, giving each process an equal-sized chunk of the array. Assume that the array length is divisible by the number of processes.
    3. Each process computes the average of their subset of the numbers.
    4. Gather all averages to the root process. The root process then computes the average of these numbers to get the final average.

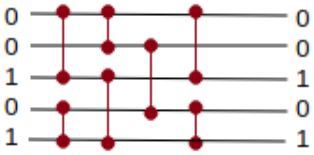
4.5 Solutions

12. Sorting Networks.

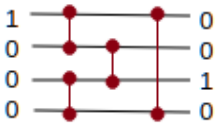
1. The 0-1 principle states that if the network sorts all possible input sequences of 0s and 1s into a non-decreasing order, it will sort any arbitrary sequence of input numbers in a non-decreasing order. As there are 2^n possible input sequences of length n of 0s and 1s, one has to check 16 sequences to ensure correctness of the sorting network.
2. (i) This sorting network is incorrect, as can be seen by the following incorrectly sorted input sequence.
   (ii) This sorting network is incorrect, as can be seen by the following incorrectly sorted input sequence.
   (iii) This sorting network is incorrect, as can be seen by the following incorrectly sorted input sequence.



(i)

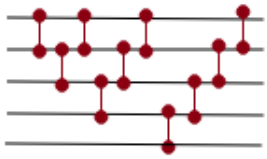


(ii)

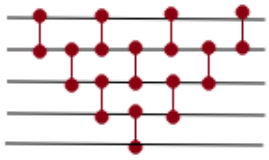


(iii)

   (iv) We see that this sorting network is a parallelized version of bubble sort. Checking all 16 possible input sequences yields the expected result: the network is correct.
3. The sequential sorting network corresponding to insertion sort looks as depicted in (i). "Shifting" the comparators as far t0 the left as possible to obtain the maximum amount of parallelism gives us (ii). Interestingly enough, this sorting netwrk is identical to the sorting network we obtain when parallelizing bubble sort.



(i)



(ii)

13. Software Transactional Memory.

1. We add a `head` variable which tracks the front of the queue and use a `Callable` class instead of a `Runnable`, as we need to be able to return a result.

```
public T deq(){
    return STM.atomic(new Callable<T>(){
        public T call(){
        if (count.get() == 0)
            STM.retry();
        T item = items.refViews().apply(head.get()).get();
        items.update(head.get(), null);
        head.set(next(head.get()));
        STM.increment(count, -1);
        return item;
        }
    });
}
```

2. Our implementation is nearly identical to the previously implemented `CircularBuffer`, only we don't need to track a tail and head of the buffer, but only the number of items on the stacks. As we know the maximum size of the stack in advance, we store the items in an array.

```
public class BoundedStackSTM<T> {
    private final Ref.View<Integer> count = STM.newRef(0);
    private TArray.View<T> items;
    public BoundedStackSTM(int capacity){
        items = STM.newTArray(capacity);
    }
    public void push(final T val){
        STM.atomic(new Runnable(){
            public void run(){
                if(count.get() == items.length())
                    STM.retry();
                items.update(count.get(), val);
                STM.increment(count, 1);
            }
        });
    }
    public T pop(){
        return STM.atomic(new Callable<T>(){
            public T call(){
                if(count.get() == 0)
                    STM.retry();
                T item = items.refViews().apply(count.get()).get();
                items.update(count.get(), null);
                STM.increment(count,-1);
                return item;
            }
        });
    }
}
```

14. Message Passing.

1. We use the `Allgather` method to make the code more concise.

```
public static void computePrimeFactors(int[] arr){
    // We assume MPI.Init() has already been called
    int size = MPI.COMM_WORLD.Size();
    int rank = MPI.COMM_WORLD.Rank();
    int partSize = arr.length / size;
    int[] res = new int[partSize];
    for(int i = rank*partSize, j = 0; i < arr.length; i++, j++){
        res[j] = primeFactors(arr[i]);
    }
    MPI.COMM_WORLD.Allgather(res,0,partSize,MPI.INT,arr,0,partSize);
    // We assume MPI.Finalize() will be called
}
```

Given a set of elements distributed across all processes, `Allgather` will gather all of the elements to all the processes. In the most basic sense, `Allgather` is a `Gather` followed by a `Bcast`.

2. We use the `scatter` and `gather` methods to compute the final average efficiently.

```
public void randAvg(int len){
    // We assume MPI.Init() has already been called
    int size = MPI.COMM_WORLD.Size();
    int rank = MPI.COMM_WORLD.Rank();
    int[] arr = new int[len];
    if(rank == 0){
        Random r = new Random();
        for(int i = 0; i < len; i++){
            arr[i] = r.nextInt();
        }
    }
    int partSize = arr.length / size;
    // Create a buffer that will hold a subset of the random numbers
    int[] subArr = new int[partSize];
    // Scatter the random numbers to all processes
    MPI.COMM_WORLD.Scatter(arr,0,partSize,MPI.INT,subArr,0,partSize,MPI.INT,0);
    // Compute the average of your subset
    int sum = 0;
    for(int i = 0; i < partSize; i++){
        sum += subArr[i];
    }
    double[] avg = new double[]{(double)sum / partSize};
    // Gather all partial averages down to the root process
    int[] avgs;
    if(rank == 0)
        avgs = new int[size];
    MPI.COMM_WORLD.Gather(avg,0,1,MPI.DOUBLE,avgs,0,1,MPI.DOUBLE,0);
    // Compute the total average of all numbers
    if(rank == 0){
        sum = 0;
        for(int i = 0; i < size; i++){
            sum += avgs[i];
        }
        double average = (double)sum / size;
    }
    // We assume MPI.Finalize() will be called
}
```